

softwarearen ingeniaritza

[I. ATALA:
softwarearen garapenaren
zenbait arlo]

Jose Ramon **Zubizarreta Aizpuru**

Udako **Euskal Unibertsitatea**

SOFTWAREAREN INGENIARITZA

[I. ATALA: Softwarearen garapenaren zenbait arlo]

Jose Ramon Zubizarreta Aizpuru

Udako **Euskal Unibertsitatea**
Bilbo, 2006

EUSKO JAURLARITZA
GOBIERNO VASCO

HEZKUNTZA, UNIBERTSITATE
ETA IKERKETA SAILA
DEPARTAMENTO DE EDUCACIÓN
UNIVERSIDADES E INVESTIGACIÓN

«Liburu hau Hezkuntza, Unibertsitate eta
Ikerketa Sailaren laguntzaz argitaratu da»

© Udako Euskal Unibertsitatea
© Jose Ramon Zubizarreta Aizpuru

ISBN: 84-8438-085-8
Lege-gordailua: BI-2571-06

Inprimategia: CUATROAS, Bilbo
Azalaren diseinua: Iñigo Ordozgoiti
Hizkuntza-zuzenketen arduraduna: Ander Altuna Gabiola

Banatzaileak: UEU. Erribera 14, 1. D BILBO telf. 946790546 Faxa. 944793039
Helbide elektronikoa: argitalpenak@ueu.org
www.ueu.org

Elkar Banaketa: Igerabide, 88 DONOSTIA

Galarazita dago liburu honen kopia egitea, osoa nahiz zatikakoa, edozein modutara delarik ere, edizio honen Copyright-jabeen baimenik gabe.

Aurkibidea

IRAKURTZAILEARI 9

I. ATALA: Softwarearen garapenaren zenbait arlo 13
1. SARRERA. 15

2. SOFTWARE-PROZESUA 17
2.1. Sarrera 17
2.2. CMM (Capability Maturity Model) 20
2.3. SPICE (Software Process Improvement and Capability dEtermination) 28
2.4. PSP (Personal Software Process) eta TSP (Team Software Process) 34
2.5. Muturreko programazioa 49
2.6. Laburpena 56
2.7. Oinarrizko bibliografia 57
2.8. Helbide interesgarriak 58

3. OBJEKTUETARA ZUZENDUTAKO PROGRAMAZIOA. 59
3.1. Java: Historia 59
3.2. Jatorrizko datu motak eta mota horien kapsulaketa 61
3.3. Sarrera/irteeren kudeaketa 63
3.3.1. StringTokenizer 69
3.4. Dokumentazioaren antolaketa: Javadoc 71
3.5. Salbuespenak salbuespen 75
3.6. Klaseak eta objektuak 82
3.7. Klase-azpiklase erlazioak: herentzia 102
3.8. Klase abstraktuak eta interfazeak 110
3.9. Liburutegi generikoak: JGL 120
3.9.1. Funtzio eta predikatu-klaseak 126
3.9.2. Iteratzaileak 139
3.10. Laburpena 146
3.11. Oinarrizko bibliografia. 147
3.12. Helbide interesgarriak 148

4. PROBA BATERATZAILEAK: JUNIT 149
4.1. Proba bateratzaileak 149
4.2. Laburpena 174
4.3. Oinarrizko bibliografia 175
4.4. Helbide interesgarriak 175

5. ASPEKTUETARA ZUZENDUTAKO PROGRAMAZIOA 177
5.1. Sarrera 177
5.2. Adibide konplexuak: deiaren fakturazioaren kudeaketa 184
5.3. Adibide konplexuak: Observer patroia 201
5.4. Laburpena 207
5.5. Oinarrizko bibliografia 208
5.6. Helbide interesgarriak 209

6. INFORMAZIO-SISTEMEN ANALISI ETA DISEINUA 211
6.1. Sarrera 211
6.2. Informazio-sistemen analisi eta diseinu egituratua 216
6.3. Objektuetara zuzendutako garapena: UML. 223
6.3.1. UMLren kondaira txikia 223
6.3.2. UML Diagramak 224
6.3.3. Erabilera Kasu (Use case) Diagramak. 225
6.3.4. Klase Diagramak 229
6.3.5. Objektu Diagrama 234
6.3.6. Pakete (package) Diagrama. 234
6.3.7. Ezarpen (deployment) Diagrama. 235
6.3.8. Osagai (component) Diagrama 236
6.3.9. Egoera-Makina (state machine) Diagrama. 236
6.3.10. Jarduera (activity) Diagrama 238
6.3.11. Elkarrekintza Diagramak. 238
6.3.11.1. Kolaborazio Diagrama 239
6.3.11.2. Sekuentzia Diagrama 241
6.4. Prozesu bateratzailea 249
6.5. MDA (Model Driven Architecture). 252
6.6. Laburpena. 262
6.7. Oinarrizko bibliografia. 262
6.8. Helbide interesgarriak 263

7. DISEINU-PATROIAK 265
7.1. Sarrera 265
7.2. Eraikitze-prozesurako patroiak 271
7.2.1. Abstract Factory. 271
7.2.2. Builder 273

7.2.3. Factory Method 275
7.2.4. Prototype 277
7.2.5. Singleton 278
7.3. Egituratze-prozesurako patroiak 280
7.3.1. Adapter 281
7.3.2. Bridge 283
7.3.3. Composite 285
7.3.4. Decorator 288
7.3.5. Façade 290
7.3.6. Flyweight 291
7.3.7. Proxy 293
7.4. Portaera kudeatzeko patroiak 294
7.4.1. Chain of Responsibility 295
7.4.2. Command 297
7.4.3. Interpreter 299
7.4.4. Iterator 301
7.4.5. Mediator 303
7.4.6. Memento 306
7.4.7. Observer 308
7.4.8. State 311
7.4.9. Strategy 312
7.4.10. Template Method 314
7.4.11. Visitor 316
7.5. Adibide konplexuak 318
7.6. Adibide konplexuak: Composite 318
7.7. Adibide konplexuak: Façade 325
7.8. Adibide konplexuak: Iterator 335
7.9. Adibide konplexuak: Strategy 339
7.10. Adibide konplexuak: Observer 344
7.11. Adibide konplexuak: Observer + Iterator 353
7.12. Adibide konplexuak: Strategy + FactoryMethod [+ JGL] 358
7.13. Laburpena 362
7.14 Oinarrizko bibliografia 362
7.15 Helbide interesgarriak 363

8. BIBLIOGRAFIA 365

Irakurtzaileari

Egun batez, konpaiñia on batean, euskaldunik baizen etzen lekhuan nengoela, ekharri zuen solhasak, izan zen perpausa, etzuela, deusek ere hanbat kalte egiten arimako, eta ez gorputzeko ere, nola egitekoen geroko utzteak, egunetik egunera, biharretik biharrera luzatzeak. Eta on zatekeiela, **Softwarearen Ingeniaritza**z liburu baten, euskaraz, guztiz ere euskararik baizen etziakitenentzat, egitea: eta hartan, gero dioenak bego dioela erakustea. Konpaiñia berean galdegin zen berehala, ea nori emanen zeikan liburuaren egiteko kargua. Eta bertze guztien artetik, hasi zeizkidan niri neroni aditzera emaiten, lehenbizian kheinuz eta aieruz, eta gero azkenean, klarki eta agerriz, nik behar nituela, egiteko hartan eskuak sarthu. Ordea nola ezpainintzen neure buruaz fida, desenkusatu nintzen ahal bezanbat. Baiña alferrik, zeren hain zinez eta batetan lothu zeizkidan, non ezetz erraiteko bide guztiak hertsi baitzerauzkidaten. Eta hala deliberatu nuen, buru-eragotzkarri bezala, liburutto baten, bi partetan partiturik, gero hunen gaiñean egitera.

Eta nahi nituzkeien bi parteak elkharrekin, eta batetan athera. Baiña ikhusirik zein gauza guti edireiten den **informatika**z eskiribaturik, gogan behartu naiz eta beldurtu, etziren bideak asko segur eta garbi, baden bitartean zenbait trabu edo behaztopa-harri. Eta halatan hartu dut gogo, lehenbiziko parte hunen, lehenik benturatzeko, eta berri iakitera bezala aitzinerat igortzeko.

Hunek zer iragaiten den, zer begitarte izaiten duen, eta nor nola mintzo den, abisu eman diazadan. Gero abisu haren arauaz ethorkizunerat gobernatzeko: eta bigarren partearen kanporat atheratzeko, edo barrenean gelditzeko eta estaltzeko.

Hala zuzentzen zitzaion Sarako erretorea *irakurtzaileari* hala zuzentzen natzaizu ni ere. Berak hala dio eta nik ere hala diotsut: honako hau lehenengo atala da eta laster beste batek jarraituko dio.

Miresten dut *Gero*ren egilea baina ez Aita Villasantek beste. Berak azal eta mamia miresten zituen; nik, azala, *euskara jator oparoan borborrean dariona*. Mamian, ez ginateke ados jarriko ez baikara kofradia berekoak. Jainkoaz ari garenean Spinozatar proiektuak eta sortaldeko haizeek gehiago erakartzen naute. Eta haragiaz ari garenean, nekez dakusat Sarako nekazaria haragiaren lohian lizundua eta idoiztatua; bai, ordea, adi, beldurtua eta txunditua, pulpitutik ari zaion euskal predikariari.

Azalaz eta mamiaz ari garenez berehala konturatuko zara irakurle, batzuetan, liburuan zehar, azalari —iturburu-kodeari— soilik utzi diodala hitz egiten, irudi-

tzen baitzait edukiontzi on batek asko laguntzen duela edukia azalarazten. Hala, ahal izan dudan guztietan iturburu-kodeaz hornitu dut ohiko jarduna.

Iturburu-kode hori laster ikusiko duzun legez Javaren teknologia erabilita idatzi dut. Irakaskuntzan aritzen naizenez maite dut teknologia hori. Softwarearen ingeniaritzaren edozein arlo ukitu, teknologiak ez dizu hutsik egingo, baliabideak eskaintzen dizkizu arlo hori landu eta inplementatzeko. Beti esan izan dut Java ez balitz existituko, asmatuko egin beharko litzatekeela, behintzat irakaskuntzarako.

Erakunde informatiko bakoitzak bere era propioa izaten du aplikazio informatikoak eraikitzeko. Batzuetan, erakundea hasiberria da eta nahikoa du aplikazio informatikoa inplementatzearekin. Beste batzuetan, erakundea nahiko heldua da eta kalitatea eta eraginkortasuna aplikatzen saiatzen da softwarea eraiki behar duenean. Bai batak eta bai besteak, softwarea eraikitzeko, prozesuak aplikatzen dituzte, arras desberdinak jakina. Liburuaren lehenengo atala prozesuez arituko da. Nola neurtu eta hobetu prozesu horiek izango da atal horren zeregina. Lan hori gauzatzeko eta ez aritzeko gurpila asmatzen behin eta berriz, *de facto*, estandar bihurtu diren zenbait erreferentzia-eredu aztertuko dira. Hurrengo ataletan ere, antzeko estrategiaz baliatuko gara zerbait uniformea eta hedatua azaldu behar dugunean.

Bigarren atalean, objektuetara zuzendutako programazioaren ezaugarri nagusiak aztertuko dira. Hurrengo ataletan, Javaren teknologia erruz erabiliko denez, atal honetan, objektuetara zuzendutako programazioa gauzatzeko teknologiak dauzkan baliabideak aurkeztuko dira. Atalaren azken aldera JGL (Java Generic Library) liburutegi generikoaren ezaugarri nagusiak aurkeztuko dira.

Ingeniariek eta arkitektoek zubiak eta urtegiak eraikitzen dituztenean proba zorrotzak prestatzen dituzte oniritziak emateko. Informatikariak ere, aplikazio informatikoa eraikitzen ari den bitartean, sortzen ari den iturburu-kodearen funtzionalitatea probatu behar du eta egiaztatu, akatsik barik dagoela. Hirugarren atalean, nola gauzatu proba bateratzaile horiek JUnit tresna informatiko aurreratua erabilita azalduko dugu.

Programazioaren arloarekin amaitzeko programazio-paradigma berri bat aztertuko dugu: aspektuetara zuzendutako programazioa. Atal berri horretan, aspektuetara zuzendutako programazioaren ezaugarri nagusiak azalduko dira AspectJ tresna informatikoa erabilita. Orobat, adibide konplexuak erabilita, aztertuko da nola integratzen den programazio mota hori objektuetara zuzendutako programazioarekin.

Aplikazio informatiko ertainak eta handiak eraikitzen dituen erakunde informatikoak ondo daki primerako programatzaileak behar dituela, baina jakin badaki ere, aplikazioaren analisia eta diseinua kaxkarrak izan badira, ez dela zilegi programatzaileari botatzea errua. Nik ere hala pentsatzen dudanez atal bat idatzi dut analisiaren eta diseinuaren nondik norakoak argitzeko. Atal honetan, *de facto*,

estandar bihurtu den UML (Unified Modeling Language) modelatze-lengoaiaren ezaugarri nagusiak azalduko dira eta orobat, aztertuko da, nola integratzen diren modelatze-lengoaiaren diagramak prozesu bateratzailean. Atal hau ixteko, MDA (Model Drive Architecture) arkitektura berriaren azalpen txiki bat egingo da.

Liburuaren azken atala diseinu-patroiez arituko da. Irakurle, sarritan aipatzen den irla galdu hartara erbesteratzen bazaituzte, eraman itzazu Gamma, Helm, Johnson eta Vlissides-en *Design Patterns: Elements of Reusable Object-Oriented Software* liburua eta nirea noski :-). Behintzat aztarnategia aurkitzen dutenean zer pentsa emango diezu arkeologoei.

Informatikariak eraginkortasuna itsatsi nahi badio diseinuari, diseinu-patroien arlo hau, nahitaez, ezagutu behar du. Ezagutu eta ahal den neurrian menderatu buruhauste bat baino gehiago ekiditeko. Liburuan, aipatutako liburuko 23 patroi kanonikoak aztertzen dira banan-banan eta amaieran, zenbait adibidetan, haien erabilera praktikoa aurkezten da.

Horratx liburuaren atalen edukien aurkezpena, hona hemen orain, zerk bultzatu nauen idaztera. Bi akuilukada izan dira ukuilutik atera nautenak: azken mohikanoaren konplexuak. Zuk ala inork ez; orain ala inoiz ez. Gure-gurea dugu konplexu hori, ez duenak ezagutzen ez gaitu ulertuko. Kolonizatzaile-sena duenarentzat beti izango gara espezie exotiko bat, bitxikeria bat, Galapagoetako kolonia. Ez du sekula santan ulertuko biodibertsitateak on egiten diola, on egiten digula. Forunkulutzat hartzen gaituzte, hori da gure desgrazia. Nola esan han, urrun daudenei, geziak eta arkuak ez direla giza ehizarako, bufaloetarako baizik.

Bestalde, irakurle, susmagarri oso nire belaunaldiko mohikanoak, gazteegi heldu baikinen iraultzetara eta zaharregi blogetara. Irakurle, mesfidantzarekin hartu nire belaunaldiko iraultzakidea. Oraindik blogik ez duelako ari da lubakietan iraultza-oihuak botatzen.

Bigarren akuilukada aspaldian jaso nuen, gazte nintzela, Lekeitioko Gora Buru zinema-aretoan. Baserritarren eta arrantzaleen mintzairan ari zen gizona baina laia, aitzurra eta salabardoa aipatu ordez uranioaz, turbinaz eta erradioaktibitateaz ari zen, zentral nuklearren arriskuez. Aitzurretik atomoaren nukleora, salabardotik fisio nuklearrera. Bai badakit, irakurle, egun alukeria dirudiela horrek guztiak gure seme-alaben testuliburuak ikusita, baina orduan, niretzat, bizitzan izaten dituzun epifaniatxo horietako bat izan zen. Berandu samar baina eskerrik asko Jose Ramon.

Eta eskerrak ematen hasita sortu nindutenekin hasi eta ordezkatuko nautenekin jarraitu behar. Pello eta Xabier, Xabier eta Pello, a ze parea, karakola eta barea; nire deskonektatzaile ofizialak. Pellok zein urrun dagoen Kamerun erakutsi zidan eta Xabierrek bizpahiru irudi marraztu dizkit maisuki, copyrighta berarentzat. Bestela, zer esan, barkatzeko jakina, egon behar nintzena baino gutxiago egon naizelako eta egin behar nuena baino gutxiago egin dudalako.

Rosa zuri zer esango dizut zuk ez dakizuna. Azken urteetako indar ikusezina. Yang-energia, egoerak, Yin-era egiten zuenean; Yin barea nire Yang-a ateratzen zenean. Behar izan dudanean lasaitasuna eman didazu enbatei aurre egiteko eta azken urteetan beti izan zara hor, etxaldearen erretagoardian zaindari. Ez dakit zelan eskertu, kendutakoa asko izan baita eta emandakoa gutxi. Mila esker eta musu bat.

Berako amabitxiri eta aitabitxiri musu bat eta besarkada bat eta *ia lehengusuei* sexty-sex gutxiago eta sexty-nine gehiago. Solasean paregabeak Larratxetarrak; harremanetarako aristokratak. Eskerrik asko beti zabal hartzeagatik zuen etxaldean.

Beratik Hondarribira, Berazaditarrak lekuko, zein baino zein jatorrago, otorduak eta solasa ez dizkigute kenduko.

Donostiako Informatika Fakultatean, Carmenekin hasi behar, hainbat urtetako bulegokidea. Azken boladan makal dabil buruko minak direla eta, senda zaitez azkar, gurekin nahi zaitugu eta. Juanan ordezkoa ere ez da herrena, zorionez Carmenen ohiturak ditu, hala nola, hamaiketako kafea.

Alfredo eta Jon, irakasgai-kideak, bikote ederra, bata nafarra bestea nafartzen. Nafarrik gabe ez dago Euskal Herririk.

Mahaikide guztiei agur bero bat.

Amaitzeko sarrera hau bi erraldoiren bizkarretara igoko naiz. Garaikideak biak: Montaigne eta Axular. Hurbil espazioan eta denboran baina urrun predikuan.

Hala dio Montaigne handiak, eta hala, nik ere: «Nire hutsak naturalak eta zuzentezinak dira gaurgero; baina, gizon zintzoek publikoari on egiten diotena beren burua imitaraziz, menturaz eginen dut nik neure ereduari itzuraraziz. Nire akatsak argitaratuz eta salatuz, norbaitek ikasiko du haien beldur izaten. Neure baitan gehien onesten ditudan zatiek nire gomendatzetik baino ohore gehiago erdiesten dute nire salatzetik. Horra zergatik haietara itzultzen eta gelditzen naizen maizago. Solasean aritzen banaiz gogo azkar batekin eta lehiakide gogor batekin, saihetsak hertsatzen dizkit, ezker-eskuin narabil; haren irudimenek abiarazten dituzte nireak. Irabazi-nahiak, ospeak, indartzeak bulkatzen eta goratzen naute nire buruaz gainera. Eta solasean adostasuna nolakotasun osoki aspergarria da. Ezein ustek ez nau harritzen, ezein sinesmenak ez nau zauritzen, nirearekin kontrakotasun handiagoa badu ere».

Eta hala Axularrek: «Aitzitik haur da nik nahi nukeien gauzetarik bat, ene enseiu aphur hunek kilika zinitzan eta gutizia, enseiu hobeago baten egitera eta ene hemengo falten ere erremediatzera. Zeren halatan, ez lizateke euskara hain labur, eskas eta ez hertsi, nola munduak uste baitu, eta baitaduka, dela».

Donostian, 2006ko apirila

I Atala

Softwarearen garapenaren zenbait arlo

1. Sarrera

Mundu osoan nagusi den sistema ekonomikoaren abiadura gorantz joan da urtetik urtera. Edozeinek, egunero, lanera bidean, abiadura horren ondorioak pairatzen ditu zirkulazioaren joan-etorrian. Lehengaiak direla trailerretan, produktu erdilanduak kamioietan edota gu geu autoetan, dena mugitzen da azkar eta konpultsiboki gizarte hiperzinetikoan. Zinetika hori guztia kontrolatzeak informazioaren trataera egokia eta azkarra eskatzen du sistema ez kolapsatzeko. Herrialde kapitalista helduetan, hots, informazioaren gizartean bizi omen diren horietan, XXI. mendeko azti berriak ei dira informatikariak.

Azti-hitzak zailak omen deszifratzen eta alfabetizazio digitala gobernu guztien lehentasunen agendan txertatu da. Ez da harritzekoa, informatikaren eragina gure eguneroko bizimoduan edonon nabaritzen baita. Sukaldeko tresna elektriko eta elektroniko gehienen portaera automatikoki kudeatzen da software informatikoaren bidez. Domotikaren baliabideak erabilita etxetresnen funtzionamendua telefono mugikorrean monitoriza dezakegu. Oraintxe bertan, gure haurra lotan, negarrez edo algaraka dagoen kuxkuxea dezakegu haurtzaindegiko webcama atzituta. Aurtengo aitorpen fiskala gauzatu baino lehen aurreko urteetako aitorpenen informazio guztia etxean eskura dezakegu Internet erabilita.

Informazio hori guztia kudeatzeko behar den softwarea konplexua eta egundokoa izaten da eta hori dena eraiki behar duen erakunde informatikoak ondo antolatuta egon behar du proiektu informatikoen kostuak eta denborak ondo egokitzeko aurrekontuetara. Erakundeak ondo zaindu beharko du softwarearen garapenaren bizi-zikloa aipatutako helburuak lortzeko; bestela, bai kostuak bai denborak ikaragarri handitu daitezke. Softwarearen garapenaren bizi-zikloaren kalitatea bermatzeko erakundeak, bi arlo osagarritan jarri behar du arreta nagusia: produktuaren ingeniaritzan, hots, softwarearen eraiketan eta softwarearen prozesuaren kudeaketan. Lehenengoarekin, garapen eta proiektuaren kudeaketaren prozesuak kontrolatzen saiatzen da metodo estandarizatuak eta tresna informatiko aurreratuak erabilita. Bigarrenarekin, ostera, erakundean aplikatzen diren prozesuen kalitatea monitorizatu egiten da jakiteko non dagoen erakundea kalitate mailan eta zer egin behar duen kalitate hori hobetzeko.

Liburu honetan, lehenengo eta behin, prozesuen arlo horri ekingo diogu CMM, SPICE, TSP eta PSP erreferentzia-ereduak aztertuta. Ondoren, garapen-

prozesua behetik gora arakatuko dugu objektuetara zuzendutako programazioaren nondik norakoak aurkeztuta lehendabizi. Atal horretatik aurrera erabiliko dugun teknologia, garapenaren arlo desberdinak aurkezteko, Java izango da. Horrela, objektuetara zuzendutako programazioa menderatuz gero, liburuaren kapituluak erraz eta azkar ulertuko dira.

Objektuetara zuzendutako programazioa erabiltzea, bakarrik, ez da nahikoa izango programa ziurrak eta eraginkorrak eraikitzeko. Aplikazioa bezeroari igorri baino lehen probatu eta egiaztatu egin beharko da proba bateratzaileak egikarituta ezusteak ekiditeko. Laugarren kapituluan aurkeztuko dugu nola zertzen dituen JUnit tresna informatikoak proba bateratzaile horiek.

Bosgarren kapituluan, programazio-paradigmaz aldatuko dugu objektuetara zuzendutako programazioaren ezintasunak apaltzeko eta ahuleziak sendotzeko. Aspektuetara zuzendutako programazioak laguntza paregabea eskaintzen du aipatutako ahuleziaren ondorioak leuntzeko eta konpontzeko.

Seigarren kapituluan, igo egingo dugu abstrakzio maila eta informazio-sistemen analisiaren eta diseinuaren nondik norakoak azalduko ditugu azterketa diakroniko txiki bat gauzatuta. Halaber, hasieran, xehetasun batzuk plazaratuko ditugu aztertuko dugun testuinguruaren ezaugarri esanguratsu batzuk azaleratzeko. Hori eginez gero, interesgunea informazio-sistemen analisi eta diseinu egituratuan finkatuko dugu. Atal horretan, analisi eta diseinu egituratuaren alde estatikoa eta dinamikoa zehazteko erabiliko diren teknika diagramatikoak eta grafikoak azalduko ditugu.

Analisia eta diseinua gauzatzeko, arreta ikuspegi egituratuan finkatu beharrean objektuetara bideratzen badugu, metodo egituratuak ez dira egokienak ebazpen sendo eta iraunkorra bilatzeko. Testuinguru berri horretan, UML metodoaren modelatze bisualak erraz erakartzen ditu domeinuaren objektuak ebazpenaren ingurumarietara informazio-sistemaren ikuspegi ortogonal bat eraikita; hots, analistak eta diseinatzaileak unean uneko teknika diagramatiko zehatz bat aukeratzen dute objektuaren ezaugarri berezi bat aztertzeko.

Erakunde informatiko batean heldutasun maila hobetzen denean aldaketa kualitatibo bat gertatzen da eraikitzen dituen diseinuetan: erregulartasunak antzematen dira diseinuen toki batzuetan. Egoera horretan, behin eta berriz, arazo bera sortzen da diseinuaren atal horretan eta diseinatzaileak, azkenean, beharra sentitzen du ebazpen uniforme eta normalizatu bat eraikitzeko diseinatu behar duen arazoa bera baita beti. Zazpigarren kapituluan aurkeztuko ditugun diseinu-patroiek aipatutako erregulartasunak era sistematikoan egituratu eta aztertzen dituzte ebazpen orokor eta estandarizatu bat eskainiz.

2. Software-prozesua

2.1. SARRERA

Informatikaren bilakaeran, pitean-pitean, edukiontziak, aurrea hartu dio edukiari. Adibidez, laurogeiko hamarkadan, CASE[1] tresnek izugarrizko ezarpena izan zuten softwarearen garapenean aplikazioen analisia eta diseinua garatzeko. Erakunde gehienetan, CASE tresna bat erosten zen eta analista berehala hasten zen erabiltzen emaitza oparoen bila. Askotan baina, emaitza oparo horiek ez ziren atzematen hain azkar. Beste behin, edukiontzia, tresna, edukiaren aurretik jartzen zen. CASE tresnak, jakina, lana errazten zuen aplikazioaren analisia eta diseinua mekanizatuz eta proiektuen gisara kudeatuz, baina aplikazioa mekanizatzeko eta sistematizatzeko, analistak, CASE tresnak inplementatzen zuen metodologia —edukia— ondo ezagutu behar zuen metodologiaren ezaugarri nagusiak kontrolatzeko. Sarritan, baldintza hori ez zen betetzen eta etsipena nagusitzen zen erakundean, emaitzak kaskarrak izaten baitziren. Ondorioa: CASE tresnaren proposatzaileak deabruaren ordezkari maltzurraren estatusa bereganatzen zuen eta, jakina, CASE tresna zokoratu egiten zen *in saecula saeculorum*.

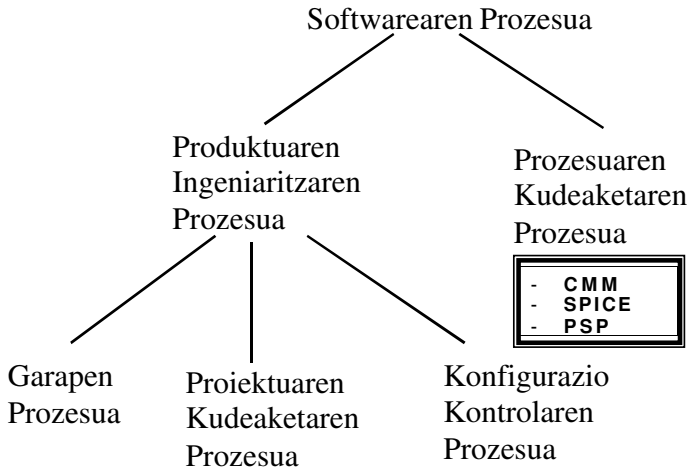
Gure egunetara etorrita, eta beharbada bizi dugun kultura nartzisistaren eraginez, badirudi orain ere edukiontziak tokia hartzen jarraitzen diola edukiari hainbat arlotan, eta informatika ez da, zoritxarrez, salbuespen bat testuinguru horretan.

Egun, lasai-lasai esango dizu batek, berak, RUP[2] erabiltzen duela beti analisiak eta diseinuak gauzatzeko, eta Microsoft Project, proiektu informatikoak kudeatzeko. Jakina, inplementatzeko orduan, objektuetara zuzenduko du programazio-paradigma eta horretarako, Java, C# edota C++ erabiliko ditu barra-barra. Kontu egin berriro ere, edukiontzi mailan ari garela hitz egiten. Entzuleak, goiko paragrafoaren informazioaren aurrean, nahitaez edukietara jo beharko luke eta galdetu, garatzen ari den aplikazio informatikoan, nola, noiz eta zergatik erabiltzen diren RUP eta Microsoft Project tresnak.

Tresna horien eraginkortasuna kaskarra izango da aplikatzen diren tokian softwarearen prozesuak ez badu garrantzi handirik. Softwarearen prozesua erabat lotuta dago softwarearen proiektu eta produktuarekin. Hala nola, softwarearen

1. Computer Aided Software Engineering
2. Rational Unified Process

proiektua, garapen-proiektu bat izango da software-prozesu bat aplikatuta kudeatzen dena. Softwarearen proiektuaren ebazpen nagusia softwarearen produktua izango da. Gehienetan, iturburu-kodearekin soilik identifikatzen da produktu hori, proiektuaren bizi-zikloaren beste produktuen kalterako. Azkenik, softwarea garatzeko balio duen metodoari software-prozesua deitzen zaio. Software-prozesuak adierazten du proiektuak jarraituko duen jardueren sekuentzia. Jardueren sekuentzia nahitaez abstrakzio mailan adierazi behar da erakundearen proiektu kopuru handienera heltzeko. Hala, proiektu bakoitzak, bere instantziazio propioa emango dio jarduera generiko bakoitzari proiektuaren bizi-zikloan.



2.1. irudia. Softwarearen Prozesua.

Beraz, orain arte esandakotik honako *urrezko arau* hau ondoriozta dezakegu: softwarearen ingeniaritzan, software-prozesuak bideratzen du beti proiektua eta ez aldrebes.

2.1. irudiak orain arte azaldutako terminologia zehazten lagunduko digu. Argi dago softwarearen prozesuaren helburua produktuak lortzea dela, eta horretarako, produktuaren ingeniaritzaren prozesu bat aplikatu behar da produktuaren kostua eta kalitatea egokiak izan daitezen. Helburu horiek lortzeko, hiru azpiprozesu desberdin baina osagarri, erabiltzen dira: garapen-prozesua, proiektuaren kudeaketaren prozesua eta konfigurazio-kontrolaren prozesua. Lehenengoak, gauzatu behar diren garapen- eta kalitate-jarduerak aztertzen ditu. Bigarrenak, jardueren planifikazioa eta kontrola kudeatzen ditu, eta hirugarrenak, aldaketen eta produktuaren integritatearen kudeaketa kontrolatzen du, kostuen eta kalitatearen arabera. Erakunde bakoitzak bere heldutasun mailaren arabera, hiru prozesuok barra-barra erabiliko ditu eguneroko jardunean edo akaso, garapen-prozesu kaskar batekin konformatu beharko du. Tokian tokiko eta unean uneko egoerak bideratuko du aukera.

Softwarearen prozesua hankamotz geldituko litzateke produktuaren ingeniaritzaren kalitatea ez balu kontuan hartuko. Softwarearen prozesuak beste prozesu mota berri bat behar du, aiduru eta kuxkuxean egongo dena, produktuaren ingeniaritza arakatzen: prozesuaren kudeaketaren prozesua.

Azken hamarkadan kalitatearen inguruko berriak eta txostenak ugalduz joan dira eta gizartearen arlo gehienetara hedatu dira. Hezkuntzan, kalitateak du lehentasuna; ekonomian, beste hainbeste. Ingeniaritza estandarrean, kalitateak, garapen-prozesu ezagun bati jarraitzen dio beti eta oso lotua egoten da ingeniaritza mota horren heldutasun mailarekin. Hasiera batean, ingeniaritzak artisautza tankerako ohiturak jartzen ditu praktikan produktua eraikitzeko, prozesu orokorrak ez dira ondo ezagutzen eta kalitatea banakako atazen kontrolean oinarritzen da. Produktuaren eta prozesuaren kalitatea hobetzeko, akatsak antzeman eta konpontzen dituen sistema bat eraikitzen da. Kontu egin, prozesuak baino produktuak duela lehentasuna lehenengo urrats honetan.

Bigarren urratsean, akatsei aurre egiteko neurriak hartzen dira eta horretarako, kalitatearen kontrol estatistikoa gauzatuko duen sistema bat eraikitzen da. Akatsen iragarpen egokiak egitea izaten da helburu nagusia.

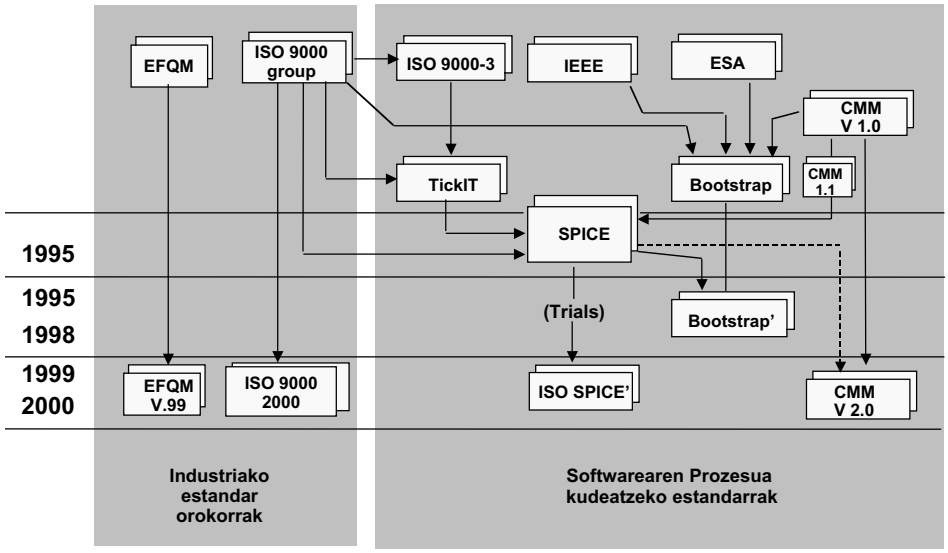
Bukatzeko, azken urratsean, produktuaren merkatu mota aldatu egiten da eta eskaera motako merkatu izatetik eskaintza motako merkatu izatera igarotzen da. Merkatuak finkatzen du salneurria eta lehia nagusitzen da partaideen artean. *Dura lex, sed lex*[3].

Softwarearen ingeniaritzak, ingeniaritza den heinean, aipatutako hiru urrats horien unibertsoan mugitu beharra du kalitatearen ezaugarriak bereganatu nahi baditu. Kalitatea bermatzeko, kalitatearen hobekuntza prozesuetan oinarritu behar da eta ez jendearengan. Bestalde, hobekuntza neurtu egin behar da eta hobekuntza-prozesuak etengabea izan behar du erakundean. Ez legoke gaizki, halaber, hemen eta orain, Watts Humphrey jakintsuaren hitzak errepikatzea berriz: erraza da azaltzen kalitatea zer den baina zeharo zaila gauzatzen erakundean. Nork ez du ezagun bat bertsolaritzari buruz orduak eta orduak egon daitekeena hitz egiten, baina gero ez dena gauza kopla arrunt bat botatzeko bat-batean afari batean? Kalitatearen inguruan ere, *zakur goseak okela amets*.

Softwarearen ingeniaritzaren egungo egoera erakunde gehienetan, aski ezaguna da[4]: proiektu informatikoen % 23k porrot egiten du, % 28k arrakasta izaten du eta % 49 bukatu egiten da, baina epez kanpo eta aurreikusitako kostuak zeharo gaindituta. Ehuneko horiek ez dute inola ere islatzen ingeniaritza heldu baten profila eta, beraz, lan asko dago oraindik egiteko artisautza-ikuspegia utzi eta ingeniaritzari atxikitzeko. Hori lortzeko, ondoko orrietan, puri-purian dauden hainbat

3. Lege gogorra, baina lege, hala ere.
4. Standish Group International: www.standishgroup.com

eredu eta prozesuaren kudeaketa-prozesu mota aurkeztuko ditugu, produktuaren ingeniaritzaren prozesua hobetzeko.



2.2. irudia. Softwarearen prozesuen estandarren bilakaera historikoa.

2.2. irudiak softwarearen prozesuaren inguruan garatu diren estandar ezagunen bilakaera historikoa islatzen du. Bakoitzaren ezaugarriak zehatz-mehatz azaltzeak luze joko ligukeenez, hautaketa pragmatiko bat egingo dugu eta eredu paradigmatiko bi azalduko ditugu ondoko orrietan: CMM (Capability Maturity Model) eta SPICE (Software Process Improvement and Capability dEtermination). Bai batak eta bai besteak, softwarea garatzen duten erakundeen heldutasun maila neurtu eta hobetzea dute jomuga.

2.2. CMM (CAPABILITY MATURITY MODEL)

Informatikaren buruhauste nagusietariko bat da softwareak kudeatu behar duen domeinuaren konplexutasunaren aldaketa-abiadura askoz ere handiagoa dela gure ahalmena baino softwarea garatu eta mantentzeko. Horrek, nahitaez, softwarearen produktuaren kalitatea, hobekuntza-prozesu etengabean sartzea behartzen du kostuak murrizteko eta epeak txukun betetzeko.

Aldez aurretik, behin eta berriz errepikatu dugu softwarearen produktuaren kalitatea oso lotua dagoela prozesuaren kalitatearekin, erabat bideratuz eta baldintzatuz haren eraginkortasuna. CMMk testuinguru horretan, azpiegitura bat eskaintzen du softwarea eraikitzen aritzen diren erakundeek kontrola ditzaten beren garapena eta mantentze-prozesuak. Hala, erakundeak bere softwarearen prozesura bideratuko du softwarearen ingeniaritzaren kultura, eta horrek, bat-bateko eragina izango du haren kostu eta garapen-denboretan.

CMMren sorkuntza 80ko hamarkadan kalitatearen inguruan sortu zen arretarekin lotu behar da. Hamarkada horren hasieran, IBM erakundea, Watts Humphrey-ren zuzendaritzapean, ingurune bat hasi zen garatzen softwarearen prozesua kontrolatzeko. Hala, 1986ko azaroan, SEI (Software Engineering Institute) erakundearen[5] azpiegitura erabilita ingurune estandar baten lehen urratsak gauzatu ziren eta, 1987ko irailean, SEIk, proposamen zehatz bat aurkeztu zuen softwarearen prozesuaren heldutasuna neurtu eta garatzeko. Handik gutxira, Watts Humphrey-k *Managing the Software Process* liburuan aurkeztu zuen proposamen hori.

CMMk kalitate totalaren kudeaketaren (TQM-Total Quality Management) ezaugarriak aplikatzen ditu softwarearen ingeniaritzaren prozesuen kudeaketan, batez ere, produktuaren kontrol kuantitatiboan. Emaitza desiragarriak lortzeko, metodologia logiko eta zentzuzkoa erabiltzen du CMMk: lehenengo eta behin, egungo egoera identifikatzen da egungo sistemaren heldutasun maila neurtuta. Gero, egoera desiragarria identifikatzen da hurrengo mailaren ezaugarriak deskribatuz. Egungo egoeratik egoera desiragarrira dagoen distantzia laburtzeko, erakundeak planifikatu, inplementatu eta instituzionalizatu egiten ditu hurrengo mailaren praktika nagusiak. Praktika horiek behin eta berriz errepikatzen dira beren erabilera optimizatu arte.

CMM arloan, berdin da liburu sakonena kontsultatu edo dokumentazio oinarrizkoena irakurri, beti agertzen zaigu barra-barra eta bat-batean *heldutasun* hitza, dela heldutasun maila adierazteko, dela erakundearen heldutasuna aztertzeko. Erakundeak heldutasun maila apala duela adierazten bada aztertzen ari garen testuinguruan, honako hau azpimarratu nahi da:

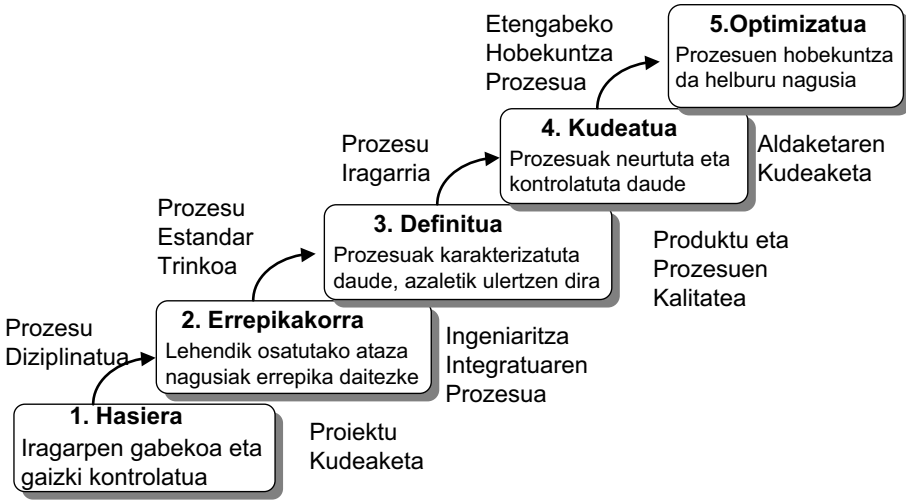
- Erakundearen kudeatzaileek ez dute softwarearen prozesua kudeatzen, *egunean eguneko* kultura dago hedatua eguneroko jardunean.
- Produktuaren kalitatea eraikitzeko eta kudeatzeko behar den prozesua ez da irizpide objektiboetan oinarritzen.
- Erakunde horietan, aurreikusitako kostuak eta denborak ez dira betetzen, aurreikuspenak errealistak ez direlako. Proiektuaren arduradunak, halabeharrez, epeak aldatzen dituenean, produktuaren funtzionalitatearen eta kalitatearen ordainetan izaten da.

Bestalde, erakundeak heldutasun maila handia duenean, gauza izaten da softwarearen prozesu eta produktuak monitorizatzeko. Halaber, tresna egokiak (prozedurak, teknologiak eta giza baliabideak) izaten ditu azterketa kuantitatiboak eta objektiboak gauzatzeko, softwarearen garapenean arazoak sortzen direnean. Erakundea prozesu diziplinatuari jarraitzeko gauza da eta erabat instituzionalizatuta dauzka mailaren praktika nagusiak eguneroko jardunean.

5. http://www.sei.cmu.edu/

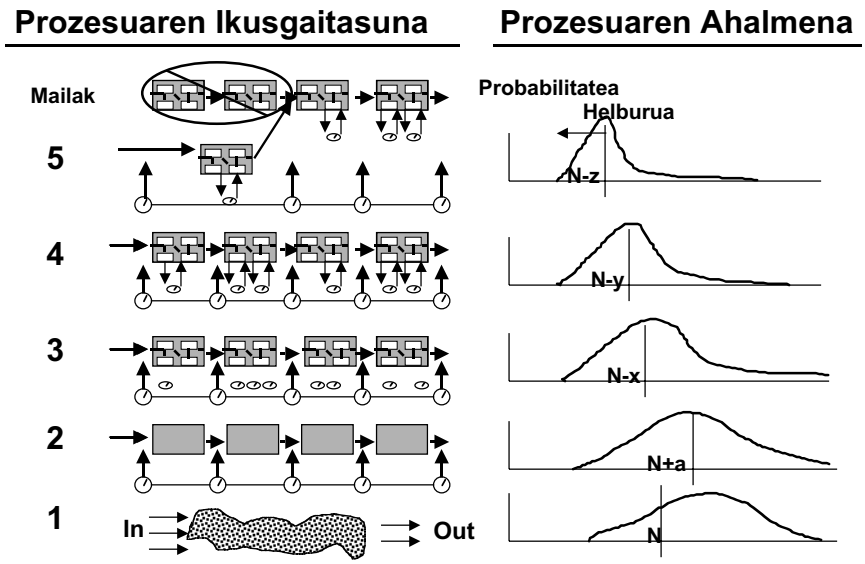
CMMk bost heldutasun maila (**2.3. irudia**) definitzen ditu softwarearen prozesuaren hobekuntza aurrera eramateko. Hala, prozesuaren heldutasuna, gaitasuna eta eraginkortasuna nabari hobetuko dira eta, horrek, eragina izango du softwarearen produktuan.

Hasieran [bigarren maila], iragarri gabeko eta gaizki kontrolatua den prozesua diziplinatu bihurtzen saiatuko gara eta horretarako proiektuaren kudeaketaren oinarrizko kontrolak sortuko ditugu. Hori egin ondoren, prozesua estandarizatu eta instituzionalizatu egingo dugu [hirugarren maila] proiektu guztietan softwarearen ingeniaritzaren eta kudeaketaren prozesuak aplikatuta. Laugarren mailan, ezaguera kuantitatiboak hartzen du lehentasuna, softwarearen prozesuan eta produktuan. Amaitzeko, bosgarren mailak irakurketa sinbolikoa du ez baita propio maila bat. Hemen, sistematizatutako prozesuaren hobekuntzak hartzen du lehentasuna eta, horretarako, etengabe eta era neurtuan aztertuko dira prozesuak; hala, akatsak eta beharrezko aldaketak finkatuta eta gauzatuta gelditzen dira.



2.3. irudia. CMM ereduaren heldutasun mailak.

2.4. irudiak azaldu dugunaren irakurketa grafiko bat egiten du prozesuaren ikusgaitasuna eta prozesuaren ahalmena kontuan izanda. Grafikoaren ezkerraldeak adierazten du nola aldatzen den softwarearen prozesuaren ikusgaitasuna. Hasierako mailan, softwarearen prozesua kutxa beltz bat da erakundearentzat, prozesuak sarrera batzuk jasotzen ditu kanpotik eta irteera bat ematen du agerkunde magiko bat izango balitz bezala. Maila horretan, analistak arazoak ditu proiektuaren garapena kontrolatzeko eta jarduerak finkatzeko. Betekizunak, kontrolik gabe iristen dira erakundera, eta zorionez, zergatik ondo jakin gabe, software-produktua ateratzen da jarraian. Software-produktuaren bezeroak bukaeran bakarrik egiazta dezake hasieran emandako betekizunak gauzatu diren ala ez produktuan.



2.4. irudia. Prozesuaren ikusgaitasuna eta ahalmena neurtzen.

Bigarren mailan, hasierako kutxa beltz monolitikoa zatitu egiten da ikusgaitasunaren mesederako. Bezeroaren betekizunak eta jarduera desberdinen produktuak kontrolatu egiten dira proiektuaren oinarrizko kudeaketa bat txertatuta eguneroko jardunean. Softwarearen eraiketa mugarriz mugarri gauzatzen da eta mugarri bakoitzean datuak eta informazioa jasotzen da, egiaztatzeko proiektuaren epeak betetzen ari direla eta kontrolpean daudela. Horrela, proiektuaren arduradunak ez daki zehatz-mehatz zer gertatzen ari den kutxa beltz bakoitzean baina emaitzak aztertuz ondoriozta dezake aurreikuspenak betetzen ari direla. Bezeroak, sukaldariek bezala, produktuaren *zaporea* dasta dezake mugarrietan eta informazioa eman erakundeari betekizunen funtzionalitatea hobetzeko. **2.5. irudia**n maila honetan gauzatzen diren jarduerak eta lortzen diren gaitasun orokorrak adierazten dira.

Hirugarren mailan —maila interesgarriena eta erabakigarriena— kutxa beltzen barne-egitura azaleratu egiten da eta prozesuak karakterizatuta gelditzen dira. Erakundeak prozesu estandar bat finkatzen du aurrerantzean proiektu guztietan aplikatuko dena. Hemen esan dezakegu benetan hasten dela finkatzen produktuen eta prozesuen kalitatea. Proiektuaren kudeaketa eraginkorra denez, desbideratutako atazak berehala ekartzen dira beren onera proiektua mugarrietara heldu baino lehen. Bezeroak, une oro, erabat monitorizatuta dauka bere produktua, garapenaren bilakaeraren aztarnak zeharo ugaldu baitira maila honetan.

Erakundeak burututako jarduerak:	**Proiektuek burututako jarduerak:**	**Lortutako prozesuaren gaitasuna:**
Software-proiektuaren kudeaketaren ***kontrol*** eta ***prozedurak*** finkatzen dira	Aurreko proiektuen emaitzak eta egungo proiektuaren betekizunak kontuan hartuz proiektuaren ***planifikazio errealista*** bat egin	Software-prozesuaren gaitasuna ***diziplinatua*** da. Proiektuaren planifikazioa eta jarraipena ***egonkorrak*** dira eta hasierako arrakastak ***errepika*** daitezke
Instituzionalizatu proiektuaren prozesuak. Horrela, proiektu berriak aurreko proiektuetan sortutako praktika eraginkorrak ***errepika*** ditzake	Softwarearen kostuak, denborak eta funtzionalitateak jaso ***arazoak identifikatzeko*** proiektuaren planifikazioan	
	Betekizunak eta ***lan-produktuak*** (*work products*) kontrolatu. Ziurtatu finkatutako ***proiektuaren estandarrei*** jarraitzen zaiela	Proiektuaren prozesua, proiektu kudeaketaren sistemaren ***kontrolpean*** dago ***plan errealei*** jarraituz

2.5. irudia. CMM ereduaren bigarren mailaren ezaugarri nagusiak.

Laugarren mailan, aurreko mailan definitutako software-prozesua kuantitatiboki kontrolatzen da. Aurreikuspen zehatzak egiten direnez, kostuen desbideraketak txikiak izaten dira eta epeak zehatz-mehatz betetzen dira. Bezeroak hasieratik daki proiektuaren arriskua norainokoa den eta bukaerako ezusteak[6] baztertuta gelditzen dira.

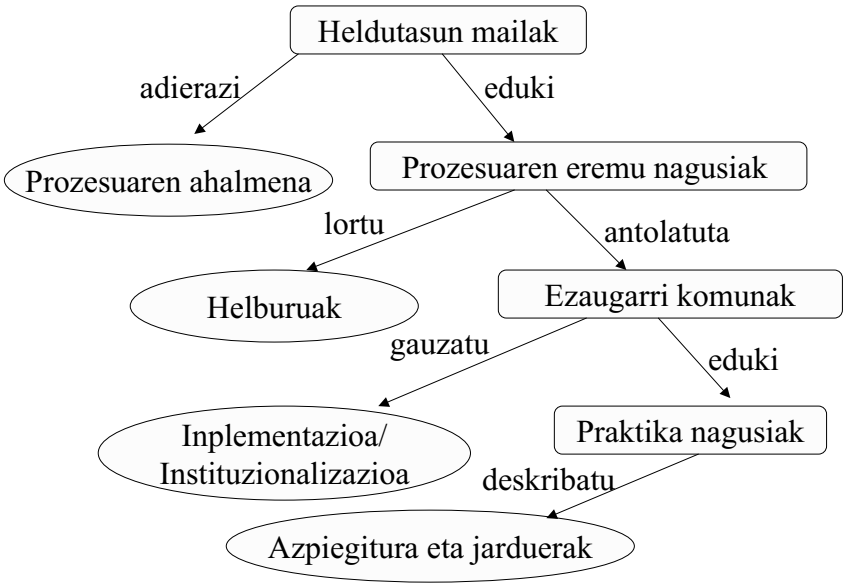
Bukatzeko, bosgarren mailan, prozesuaren hobekuntza etengabe lantzen da. Prozesuaren edozein aldaketaren eragina aurreikusi egin daiteke eta kuantitatiboki neurtu. Aldaketak, diziplinatuak izaten dira eta helburu nagusia, jarduera ez eraginkorrak saihestea izaten da.

2.4. irudiak badu, halaber, bigarren zutabe bat prozesuaren ahalmenaren profila adierazten duena. Profil horietan, erakundearen heldutasun mailan igoz doan heinean, erakundeak iragarpen hobeak eta zehatzagoak egiten ditu proiektuaren desbideraketak txikiagoak izango direlako. Epeak eta kostuak jaitsi egiten direnez, proiektua gauzatzeko baliabideen beharra txikiagotu egiten da.

CMMren azalpen orokorra burututa, joan gaitezen haren barne-egitura (**2.6. irudia**) nola dagoen antolatuta aztertzera. CMMren heldutasun maila bakoitzaren helburu nagusia softwarearen prozesuaren ahalmena islatzea izaten da eta hori lortzeko, prozesuaren eremu nagusietan (Key Process Areas) antolatzen da heldutasun maila bakoitza. Maila bakoitzaren eremu nagusi bakoitzarentzat helburu zehatz batzuk finkatzen dira. Helburu horiek adieraziz gero, eremu nagusi bakoitzarentzat

6. Espero zen funtzionalitatea ez da lortu, kostuak handiagoak izan dira edota epeak ez dira bete.

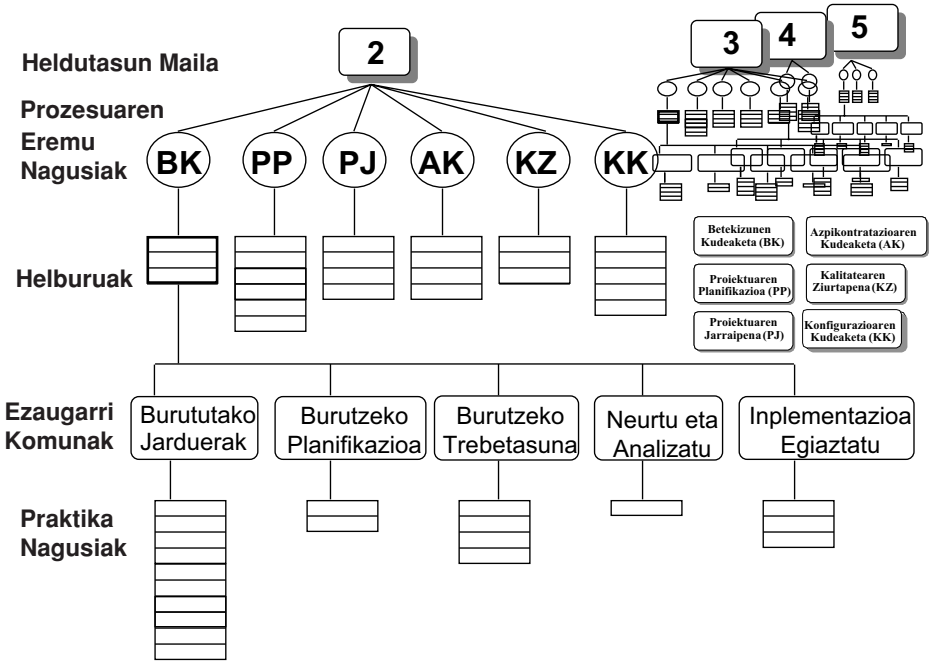
haren ezaugarri komunak zehazten dira, nola geldituko diren gauzatuta finkatzeko. Bukatzeko, ezaugarri komun bakoitzak bere praktika nagusiak edukiko ditu ziurtatzeko praktikan prozesuaren eremu nagusiaren helburuak ondo gauzatzen direla, jarduera egokiak egikarituta. Azpimarratu behar da praktika nagusi horiek zer egin behar den deskribatzen dutela eta ez nola inplementatu behar den jarduera bakoitza.



2.6. irudia. CMM ereduaren barne-egitura.

Aitzinean aipatu dugun legez, CMMren bigarren mailak proiektuaren kudeaketaren oinarrizko kontrolak finkatu nahi ditu eta horretarako honako prozesuaren eremu nagusi hauek (**2.7. irudia**) landu eta antolatuko ditu:

- Betekizunen kudeaketa (BK). Eremu hau, erabiltzailearen betekizunak dokumentatu eta kontrolatzen saiatuko da. Kontuan hartu behar da erabiltzailearen betekizunak software-proiektuaren aurrebaldintza nahitaezkoak direla proiektua planifikatu, iragarri eta gauzatzeko.
- Softwarearen proiektuaren planifikazioa (PP). Arlo honetan, iragarpen errealak egingo dira baliabideak, arriskuak eta denborak identifikatu eta zehazteko.
- Softwarearen proiektuaren jarraipena (PJ). Eremu nagusi honekin, proiektuaren ikusgaitasuna handitu egiten da. Eremu hau txukun lantzen bada, software-produktuaren eguneroko garapena kontrolatuta eta kudeatuta egoten da erakundean. Proiektuaren desbideraketak lehenbailehen antzematen dira eta berehala jartzen dira martxan ekintza egokiak, arazoak zuzentzeko.



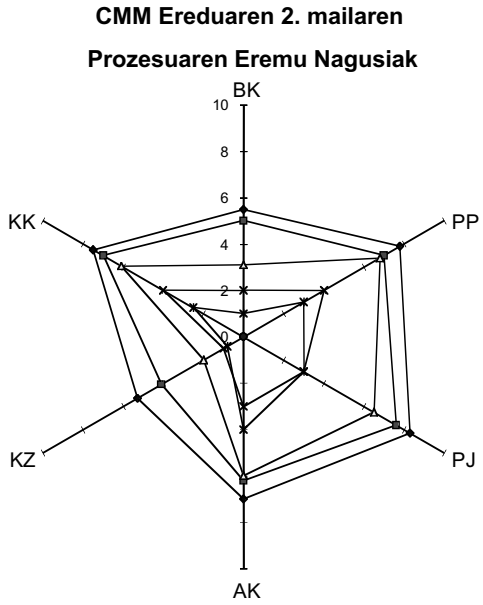
2.7. irudia. CMM ereduaren bigarren mailaren prozesuaren eremu nagusiak.

- Azpikontratazioen kudeaketa (AK). Arlo hau ez da beti beharrezkoa. Softwarearen konplexutasunak eskatzen duenean edo erakundearen lan egiteko erak inposatzen duenean, software-produktua azpikontratatu egiten da eta beharrezkoa zaigu azpikontratazioaren kudeaketa egoki bat eratzea.

- Softwarearen kalitatearen ziurtapena (KZ). Eremu honetan, software-produktuak eta praktika-eginkizunak ikuskatu eta egiaztatu egiten dira baieztatzeko finkatu diren estandarrak eta prozedurak aplikatzen ari direla.

- Softwarearen konfigurazioaren kudeaketa (KK). Hemen, proiektuaren bizi-zikloan zehar, proiektuaren produktuen integritatea mantentzen eta finkatzen saiatuko gara. Produktuen integritatea mantentzeko, proiektuaren planifikazioa denbora-mugarriz osatua egongo da. Horrela, proiektuaren arduradunak proiektuaren aztarnak jasoko ditu proiektuaren bizi-zikloan zehar eta konfigurazio-aldaketak sistematikoki kudeatuko ditu.

2.8. irudiak adierazten du eremu nagusien ezaugarri komunen heldutasunaren maila norainokoa den, eta zehazten du nola dauden kudeatuta eta kontrolatuta bigarren mailaren eremu nagusiak.



2.8. irudia. Ezaugarri komunen heldutasun maila.

Jakin badakigu CMMren hirugarren mailak prozesua estandarizatu eta instituzionalizatu egiten duela. Hori gauzatzeko, bigarren mailak legez, maila honek ere bere prozesuaren eremu nagusiak ditu:

- Erakundearen prozesuaren fokusa. Erakundea gauzatzen ari den prozesua kontrolatu eta garatu nahi bada nahitaezkoa da ondo koordinatzea softwarearen prozesuaren ekintzak erakundean zehar. Horrela, ardurak non kokatzen diren identifikatuak egongo dira eta posible izango da prozesuaren garapena.

- Erakundearen prozesuaren definizioa. Prozesuaren kudeaketa kuantitatiboa erabilita, prozesuaren definizio zehatza lortzen da, eta horrela, softwarearen prozesu estandarra finkatuta geldituko da. Erakundeak, hortik aurrera, prozesua garatu, neurtu eta kudeatu ahal izango du.

- Eguneratze-programa. Prozesuak, estandarizatzen den heinean, eragina izango du erakundearen giza baliabideetan. Batzuetan, eguneratze-prozesu hori erakundean bertan gauzatuko da inongo laguntzarik gabe; beste askotan, ostera, kanpoko laguntza beharko da eta ikastaroak antolatu beharko zaizkio erakundeko pertsonalari.

- Softwarearen kudeaketa integratua. Prozesua estandarizatuz gero, bermatuta gelditu behar da proiektuaren planifikazioa eta kudeaketa ondo integratuta gelditzen direla prozesu berrian.

- Softwarearen produktuaren ingeniaritza. Goian planifikazioari buruz esan dugunak berdin balio digu proiektuaren jarduera teknikoen arloak kudeatzeko, hau da, betekizunen analisia, diseinua, kodeketa eta probak.
- Taldeen arteko koordinazioa. Erakunde osoaren integrazioa eta koordinazioa bermatu nahi badugu era eraginkorrean, proiektuan parte hartzen duten talde guztien elkarrekintzek eta interfazeek ondo planifikatuta eta kudeatuta egon behar dute. Gardentasuna nagusitu egingo da erakundean, talde guztiak jakitun egongo baitira besteen planifikazioez eta egoerez.
- Ikuskapen-politika. Softwarearen produktuen akatsak ezabatu nahi badira azkar eta era eraginkorrean, ikuskapen-politika bat finkatu behar da erakundean akatsak berehala aurreikusi eta jakinaren gainean jartzeko.

Horratx, beraz, bigarren eta hirugarren mailen eremu nagusien azalpena. Kontuan izanda erakunde gehienen maila lehenengoa edo bigarrena izaten dela eta laugarren eta bosgarren mailek ere antzeko ibilbideari jarraitzen diotela, ez zaitut irakurle gehiago nekatuko eta hemen utziko dugu eremu nagusien azalpena. Informazio egarriz gelditu bazara eta ur handiagoetan murgildu nahi baduzu, M. C. Paulk, C.V. Weber, B. Curtis eta M. B. Chrissis-en *The Capability Maturity Model: Guidelines for Improving the Software Procces* liburua kontsulta dezakezu egarri hori asetzeko.

2.3. SPICE (SOFTWARE PROCESS IMPROVEMENT AND CAPABILITY DETERMINATION)

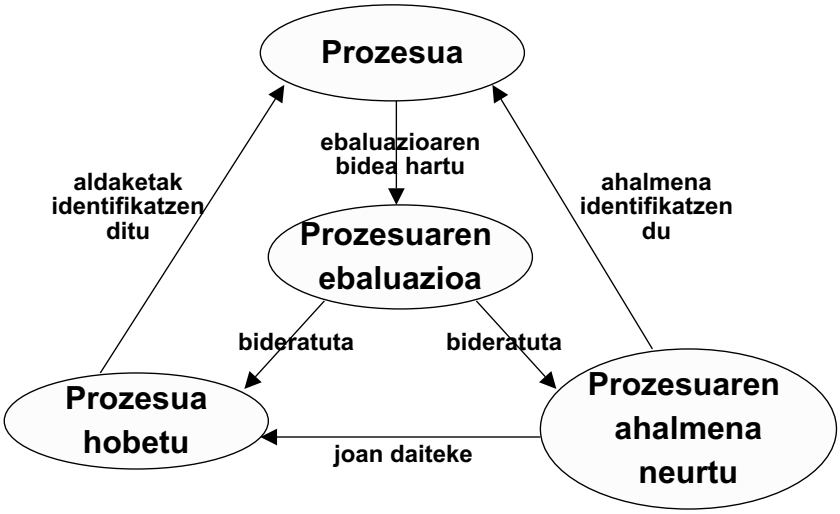
SPICE (Software Process Improvement and Capability dEtermination) erreferentzia-eredua CMMren europar bertsio eguneratua dela esan dezakegu. Eredua, ISO/IEC[7] 15504 estandarrarekin lerrokatzen da orpoz orpo eta ISO/IEC 12207 estandarraren ezaugarriak ere hartzen ditu kontuan. 1987an, ISO eta IEC erakundeek lan-talde bat (JTC - Joint Technical Committee) eratzea erabaki zuten informazioaren teknologien arloan estandarrak eraikitzen joateko. ISO/IEC 12207 estandarrak softwarearen bizi-zikloan gertatzen diren prozesu guztiak katalogatzen dituenez, SPICE ereduak prozesu horiek hartu zituen kontuan bere prozesu-kategoriak antolatzeko.

80ko hamarkadan, bai Ameriketako Estatu Batuetako baita Europako armaden arduradunak ere, berehala konturatu ziren softwarearen konplexutasuna izugarri ugaldu zela aurreko urteetan eta, tamalez, gehiago handituko zela etorkizunean. Hornitzaileekiko software-kontratuak arriskutsuagoak bihurtzen ari ziren kostuak igo ahala. Egoera horretan, zerbait egin behar zen softwarearen kalitatea bermatzeko eta arriskuak ekiditeko edo gutxienez apaltzeko. Aurreko orrietan ikusi dugun

7. ISO: International Organization for Standardization
IEC: International Electrotechnical Commission

legez, Ameriketako Estatu Batuetan, CMM eredu estandarra proposatzea izan zen erantzuna. Europako indarrak, bestalde, Erresuma Batuan kokatu ziren eta hango gobernuak lan-talde berezi bat (CPCC - Computing Policy Consultative Committee) antolatu zuen arazoari aurre egiteko. Arazoaren ikerketa Harry Barker-en esku gelditu zen eta horrek bi ibilbide proposatu zituen: hornitzaileekin sortzen ziren arazoak konpontzeko, defentsa arloan bakarrik aplikatuko zen eredu bat proposatu edo beste eredu irekiago bat antolatu, softwarearen industriari, oro har, aplikatuko zitzaiona. Deman, bigarren aukera atera zen garaile.

1992ko ekainean *The Need and Requirements for a Software Process Assessment Standard* txostena argitaratu zuten. Txostenak azpimarratzen zuen softwarearen prozesuaren kudeaketa gauzatzeko estandar bat behar zela. Urtearen bukaeran, ISO erakundeak, txostenari bultzada bat emateko, WG10 (Working Group 10) lan-taldea sortu zuen aipatutako estandarra garatu eta finkatzeko. WG10 lan-taldea 1993ko urtarrilean hasi zen lanean Peter Simms eta Alec Dorling-en ardurapean.



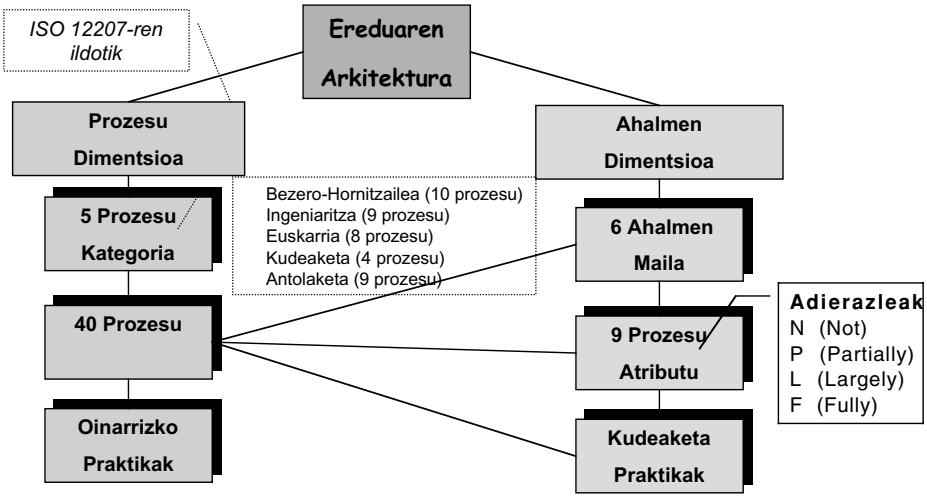
2.9. irudia. SPICE erreferentzia-ereduaren egituraketa.

SPICE erreferentzia-ereduak bi helburu (**2.9. irudia**) nagusi ditu: softwarearen prozesuaren hobekuntza planifikatu eta kudeatzea, eta prozesuaren ahalmena neurtzea. Bestalde, ereduaren diseinatzaileek hainbat ezaugarri izan zituzten kontuan erreferentzia-eredu estandarraren hedapena zabala izan zedin:

- Ereduak erakunde gehienetan aplikagarria izan behar zuen eta, beraz, generikoa izan behar zuen domeinu desberdinetan aplikatzeko, kontuan izan gabe erakundearen neurria eta egitura.

- Ereduak independentea izan behar zuen erakundearen softwarearen bizi-ziklotik, softwarearen teknologiatik eta softwarea garatzeko metodoetatik.
- Ereduak, irizpide kuantitatiboetan oinarrituta, erabilera objektiboa eduki behar zuen nahitaez.
- Ereduaren irteera estandarrak prozesuen profiletan gauzatu behar zuen. Bai prozesuaren ahalmena neurtzeko bai prozesuaren hobekuntza ikuskatzeko, profilak erabili behar ziren. Horrela, prozesuaren arduradunak informazio zehatzagoa zuen prozesuaren nondik norakoak aztertzeko.

Ezaugarri horiek guztiak ondo finkatzeko eta estandarizatuta uzteko, ereduaren diseinatzaileek bederatzi zatitan antolatu zuten eraikitzen ari ziren dokumentazioa (**2.10. irudia**). Bederatzi zati horietatik bigarren, hirugarren eta bederatzigarren zatiak normatiboak ziren eta beste guztiak informazio-emaileak.



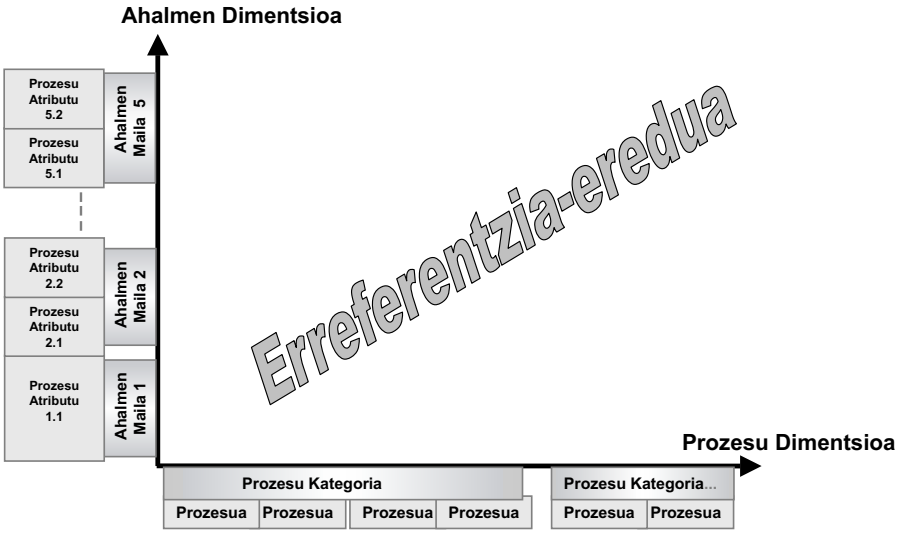
2.10. irudia. SPICE erreferentzia-ereduaren arkitektura orokorra.

Ondoko paragrafoetan, zati normatibo horiek aztertuko ditugu. SPICE erreferentzia-ereduaren egiturak bi dimentsio mota (**2.11. irudia**) erabiltzen ditu erakundearen softwarearen prozesuaren profila eraikitzeko. Dimentsio batek prozesuaren kategoriak islatzen ditu eta besteak, ostera, prozesuaren ahalmena. Prozesuaren ahalmenak bost maila dauzka hautatutako eta ebaluatutako prozesuen ahalmena neurtzeko. Irudian ikusten den legez, bederatzi prozesu-atributu erabiltzen dira ahalmena neurtzeko.

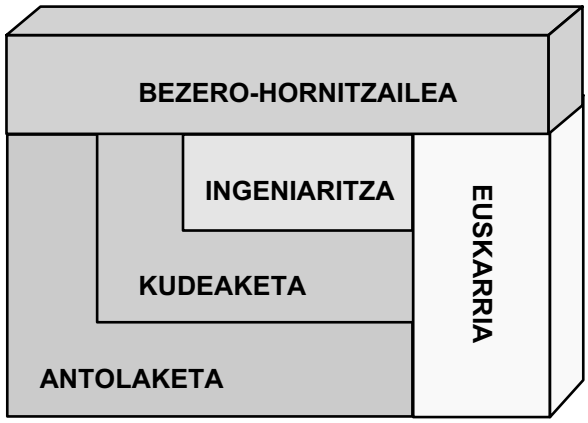
Prozesuaren kategoriak (**2.12. irudia**) bost arlotan sailkatzen dira eta bakoitzak softwarearen eremu bat aztertzen du. Grafikoak adierazten duen bezala ingeniaritza kategoria dimentsio horren muina da eta beste kategoriak haren inguruan antolatzen dira. Ingeniaritzarekin batera badago beste kategoria garrantzitsu

bat: bezero-hornitzaile kategoria. Softwarearen ingeniaritzaren produktuaren eraikitzailea erakundea bada, kategoria horrek adierazten du nolako harremanak izan behar dituen; bestalde, softwarearen ingeniaritzaren produktuaren kontsumitzaileak bagara, kategoria horrek adierazten du nolako baldintzak bete behar dituen gurekin harremanetan jarri behar den hornitzaileak.

Beste kategoriak bi horien inguruan lerrokatzen dira. Batek, euskarria ematen du adierazteko nola antolatu nahi ditugun prozesuaren dokumentazioa, kalitatea, egiaztapenak eta konfigurazioaren kudeaketa. Beste bi kategoriek, ostera, softwarearen bizi-zikloan antolaketa-arloak egituratzeko balio dute.

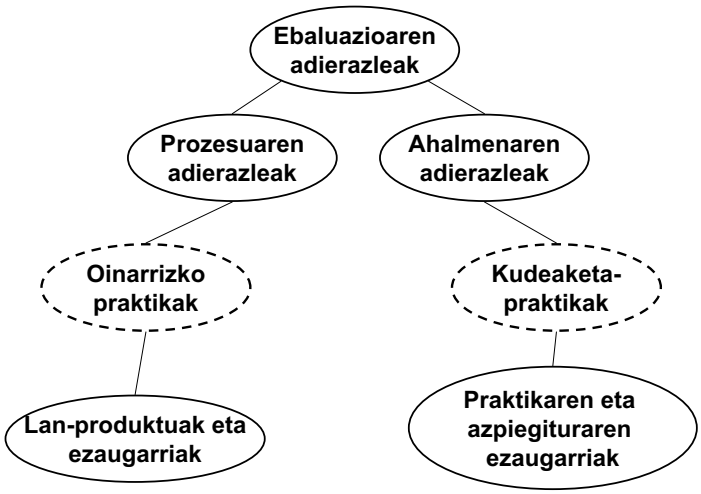


2.11. irudia. SPICE erreferentzia-ereduaren dimentsio motak.



2.12. irudia. SPICE erreferentzia-ereduaren prozesuaren kategoriak.

Erakundearen softwarearen prozesuaren ahalmena neurtzeko, erreferentzia-eredua (**2.10. irudia**) erakunde horretan gauzatzen diren praktiketan oinarritzen da, hori baita erakundeak eskain dezakeen informazio objektiboena. Hala, prozesuen bilakaera kontrolatzeko, oinarrizko praktika batzuk proposatzen ditu erreferentzia-ereduak. Praktika horiek prozesuaren adierazleekin (**2.13. irudia**) lotzen dira eta baliagarriak dira ebaluatzailearentzat, jakiteko, prozesuaren helburuak betetzen ari diren ala ez. Oinarrizko praktikak lan-produktuetan gauzatzen dira, horiek baitira erreferentzia-ereduaren irteera estandarrak.



2.13. irudia. Prozesuaren ebaluazioaren adierazleak.

Prozesuaren dimentsioa alde batera utzita, azter dezagun ahalmenaren dimentsioa. Dimentsio horrekin ebaluatzaileak prozesuaren heldutasun maila neurtzen du jakiteko (a) prozesuaren hobekuntza zenbatekoa izan den erakundea hobekuntza-prozesuan sartuta badago edo (b) zein den prozesuaren ahalmen maila ebaluazio-prozesu batean ari bada. Prozesu desberdinen ahalmena kontrolatzeko, kudeaketa-praktika batzuk proposatzen ditu erreferentzia-ereduak. Praktika horiek ahalmenaren adierazleekin lotzen dira eta baliagarriak dira ebaluatzailearentzat jakiteko noraino heltzen den prozesuaren ahalmena. Eskuratzen den informazioa objektiboa izan dadin, erreferentzia-ereduak bederatzi prozesu-atribututan egituratzen du ahalmen-dimentsioa:

- Atributu 1.1: *Prozesua gauzatu*. Atributu honek ziurtatzen du prozesuaren oinarrizko praktikak burutu egiten direla. Sarrerako lan-produktu batzuk ditugu prozesuaren sarreran eta prozesua egikaritzen denean irteerako beste lan-produktu batzuk eskuratzen dira. Lan-produktuen kalitatea ez da neurtzen bakarrik, ziurtatu egiten da prozesua burutu egiten dela.
- Atributu 2.1: *Prozesua kudeatu*. Atributu honekin prozesua planifikatzen den ala ez adierazten da. Maila honetan lan-produktuen kudeaketa

gauzatzen da ziurtatuz epeak ondo betetzen direla eta lan-produktuak eraikitzeko behar diren baliabideak kontrolpean daudela.

- Atributu 2.2: *Lan-produktuak kudeatu*. Atributu honek ziurtatzen du lan-produktuak ondo dokumentatuak, kontrolatuak direla eta eskatzen zaien funtzionalitatea lortzen dutela.
- Atributu 3.1: *Prozesua definitu*. Prozesuaren ahalmena maila honetaraino heldu bada prozesua estandarizatuta dagoelako da.
- Atributu 3.2: *Prozesuaren baliabideak kudeatu*. Prozesua estandarizatuz gero, nahitaezkoa da prozesuaren azpiegitura ondo finkatzea prozesua erakundean instituzionalizatzeko.
- Atributu 4.1: *Prozesua neurtu*. Maila honetan bai produktuaren kalitatea bai prozesuaren kalitatea, kuantitatiboki neurtzen da. Erakundeak mekanismo formal bat instituzionalizatzen du datuak jaso, neurtu eta aztertzeko.
- Atributu 4.2: *Prozesuaren kontrola gauzatu*. Atributu honek prozesuaren eraginkortasuna neurtu eta kudeatu egiten du. Produktuaren eta prozesuaren kalitatea aurreikusi egiten da eta ziurtatu mugarrietan eta kontrolpean dagoela. Prozesuaren ahalmena ahuldu egiten bada, erakundeak baliabideak dauzka lehengo maila berreskuratzeko.
- Atributu 5.1: *Prozesuaren aldaketak kudeatu*. Erakundearen prozesuaren definizioa, kudeaketa edota eraginkortasuna aldatu egiten badira erakundearen helburuak aldatu egin direlako, atributu honetara heldu den prozesuak ahalmena du aldaketaren eragina iragarri eta ebaluatzeko prozesuaren aldaketa sistematikoa gauzatuta.
- Atributu 5.2: *Prozesuaren hobekuntza etengabea osatu*. Maila honetan aldaketen beharrak identifikatu egiten dira eta prozesuaren hobekuntzak aurreikusi. Prozesuaren aldaketak era eraginkorrean gauzatzen dira.

Bukatzeko, adibide (**2.14. irudia**) txiki bat aztertuko dugu azaldu ditugun prozesuaren dimentsio biak elkarrekin ipintzen dituena. Grafikoaren lerro bakoitzak hausnartzeko hautatu diren prozesuen egoera islatzen du eta eskuineko aldean, ebaluatzaileak eman dion maila adierazten da. Kontu egin ingeniaritza dela irudia egituratzeko aukeratu den prozesu-kategoria. Zutabe bakoitzak prozesuaren atributu baten egoera[8] islatzen du. Gure adibidean argi ikusten da, erakundeak, nahiko

8. Bazterrak ez nahasteko, prozesuaren atributu bakoitzari emango diogun balioa erreferentziaereduak proposatzen duen multzotik aukeratuko da. Hau da, atributuaren ahalmena txikia bada N (No) bat jarriko da eta atributuaren ezaugarriak partzialki betetzen badira P (Partially) bat idatziko da. Atributuaren kudeaketa-praktikak nahiko ondo betetzen badira baina ez erabat L (Largelly) bat jarriko da eta bete-betean gauzatzen badira F (Fully) bat. Tresna aurreratu bat erabiltzen bada prozesu guztia kudeatzeko, adierazle horiek balio kuantitatiboak har ditzakete eta azterketa estatistikoak egin daitezke prozesuaren ahalmena neurtzeko eta hobetzeko.

ondo kudeatzen duela proben esparrua, produktuaren ingeniaritzaren arloan. Esan dezakegu halaber, softwarearen diseinuak maila txukuna duela. Bestalde, softwarearen analisiaren eta mantentze-lanen ahalmen maila ez da oso handia baina arazoen jatorria zehaztuta dago. Bukatzeko, softwarearen produktuaren kalitate maila handitu nahi bada, zalantzarik gabe, zeharo hobetu behar da betekizunen azterketa.

Prozesuak (Ingeniaritza)	Prozesuaren atributuak									Maila
	PA1.1	PA2.1	PA2.2	PA3.1	PA3.2	PA4.1	PA4.2	PA5.1	PA5.2	
Betekizunen azterketa	L	L	P	N	N	N	N	N	N	M 1
Softwarearen analisia	F	L	L	F	F	P	N	N	N	M 2
Softwarearen diseinua	F	F	F	F	F	P	L	N	N	M 3
Probak	F	F	F	F	F	L	F	P	N	M 4
Mantentze-lanak	F	F	L	P	N	F	N	N	N	M 2

2.14. irudia. Prozesuaren dimentsioak neurtzen.

2.4. PSP (PERSONAL SOFTWARE PROCESS) ETA TSP (TEAM SOFTWARE PROCESS)

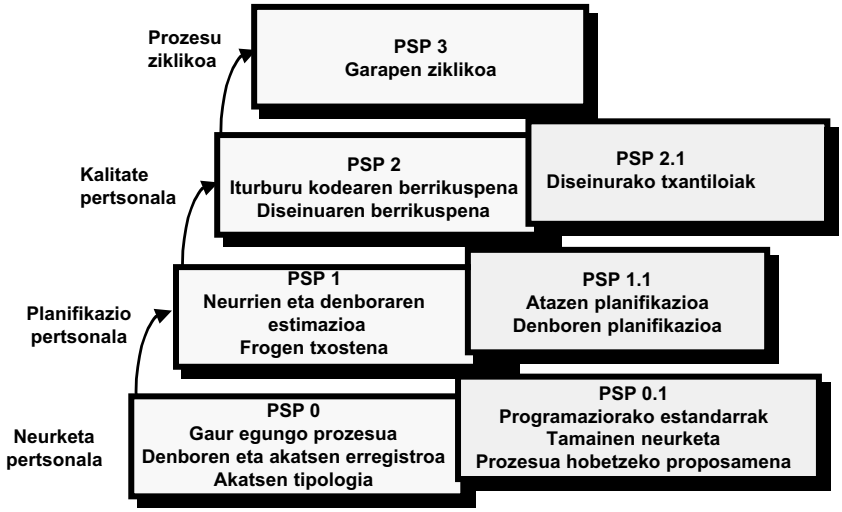
CMM eta SPICE erreferentzia-ereduak tresna aproposak dira erakundearen heldutasun maila neurtu eta zer egin behar den zehazteko. Beren azterketa-ingurunea erakundearen goi-mailara zuzentzen denez taxuz betetzen dute lan hori. Prozesuaren hobekuntza-plan bat onartuta, erakundean nola gauzatu behar den erabakitzean sortzen da arazoa, zeren erreferentzia-ereduak oso orokorrak baitira eta ez baitute laguntza handirik eskaintzen arlo horretan. Watts Humphrey ere, berehala konturatu zen hutsune horretaz eta arazoa bideratzeko, lehenengo PSP (Personal Software Process) eta ondoren TSP (Team Software Process) ereduak diseinatu eta egituratu zituen.

Softwarearen eraiketa lan konplexua eta neketsua da hainbat partaide motataz baliatzen baita erakundea, softwarearen ezaugarri guztiak ondo egituratzeko. Erakundearen software-prozesuaren heldutasun maila hobetu nahi bada aldaketak gauzatu behar dira erakunde horretan. CMM eta SPICE erreferentzia-ereduak aplikatzen direnean erakunde informatikoan, gauza jakina da erakundeak harresi bat aurkitzen duela hirugarren heldutasun maila gainditzeko. Alde batetik, tresnak, metodoak eta prozesuak egokitu behar dira, eta bestetik, giza baliabideak. Software-erakundea eraginkorra izateko software-ingeniari eraginkorrak izan behar ditu lantalde ondo eratuetan txertatuta. PSP eta TSP ereduen zeregina, izan ere, giza eta talde-eraginkortasun horien maila hobetzea izango da.

CMM eta SPICE erreferentzia-ereduak aztertu ditugunean proiektuaren planifikazioaren eta kudeaketaren berebiziko garrantzia azpimarratu dugu behin eta berriro. PSP eta TSP ereduek ikuspegi berari jarraituko diote software-ingeniariaren eta lan-taldearen eraginkortasun maila hobetzeko.

Watts Humphrey-k PSP eredua egituratu zuenean, jakin bazekien software-ingeniariaren eguneroko jardunean ohiturak sustraituta egoten direla erakundeetan eta zaila izaten dela aldaketa sakon bat proposatzea atoan. Hori dela eta, software-laborategi kontrolatu bat eratu zuen bere proposamena aurrera eramateko. Gauzak errazteko, eta zergatik ez diru apur bat eskuratzeko, *A Discipline for Software Engineering* liburua idatzi eta argitaratu zuen. Liburuak, aipatutako laborategi kontrolatuaren nondik norakoak azaltzen ditu eta, aldi berean, PSP ereduaren aurkezpen zabala eta zehatza gauzatzen du. Software-ingeniariaren ohiturak hobetu nahi direnez, liburuak egundoko garrantzia ematen dio alde praktikoari. Horretarako, hamar ariketa kontrolatu proposatzen dira azalpen teorikoa osatzeko.

Teorikoki, PSP ereduak, CMM erreferentzia-ereduaren heldutasun kontzeptua berreskuratu eta kontzeptu horren inguruan antolatzen du bere arkitektura orokorra. Hau da, CMM erreferentzia-ereduan bezala, PSP ereduak (**2.15. irudia**) hiru heldutasun maila izango ditu software-ingeniariaren garapena neurtu eta egituratzeko. Maila bakoitzak helburu jakin bat du eta osagarria da bere gainean dagoen mailarentzat. Adibidez, hasierako mailak neurketaren garrantzia azpimarratu nahi du, eta horretarako, azpiegitura kontrolatu bat antolatzen du software-ingeniariak, proiektuaren denbora eta akatsen neurketaren garrantziaz ohartarazteko.



2.15. irudia. PSP ereduaren heldutasun mailak.

Maila horretan ingeniariak hasierako hiru ariketa[9] kontrolatuak gauzatzen ditu. Ariketa bakoitzarentzat, diseinu txiki bat eraikitzen du eta programatzen hasten da. Hori egiten ari dela, eta prozesuaren gidoi txiki bati jarraituz, denborak eta akatsak erregistratzen ditu propio diseinatutako agirietan. Horrela, ariketa bat bukatzen denean eta *postmortem*[10] egoerara heltzen denean, hainbat informazio izango du proiektu txiki bakoitzaren nondik norakoak aztertzeko. Zenbat eta ariketa gehiago gauzatu metodologia hori erabilita, orduan eta informazio gehiago izango du ingeniariak, haren portaeraren alde ahul eta sendoak aztertzeko. Azkenean, gauzak ondo joan badira, prozesua hobetzeko proposamen zehatz bat egingo da eta programaziorako estandarrak finkatuko dira.

Behin neurketaren arloa menderatzen dela baieztatzen bada, ingeniariaren ardurak kualitatiboki aldatuko dira. Ingeniariaren jarrera pasibo izatetik aktibo izatera igaroko da. Hau da, proiektu bat planifikatzerakoan ez da nahikoa izango proiektu horren informazioa pasiboki jasotzea proiektua garatuz doan heinean. Proiektua hasi baino lehen, estimazioak egitea eskatuko zaio ingeniariari, proiektuak gauzatzeko zein neurri eta zenbat denbora beharko duen aurreikusteko.

2.16. irudiak azaltzen du nola egituratzen diren maila honen ezaugarri nagusiak bizi-zikloan zehar. Proiektuaren betekizunak definituz gero, aplikazioaren diseinu kontzeptuala gauzatzen da. Hortik aurrera, estimazioen unean sartzen da ingeniaria, proiektuaren egutegia eraikitzeko. Une horretan, aipatutako liburuan, PROBE (PROxy-Based Estimating) metodoa erabiltzea proposatzen du Watts Humphrey-k, ingeniariari estimazioen lana errazteko eta egituratzeko.

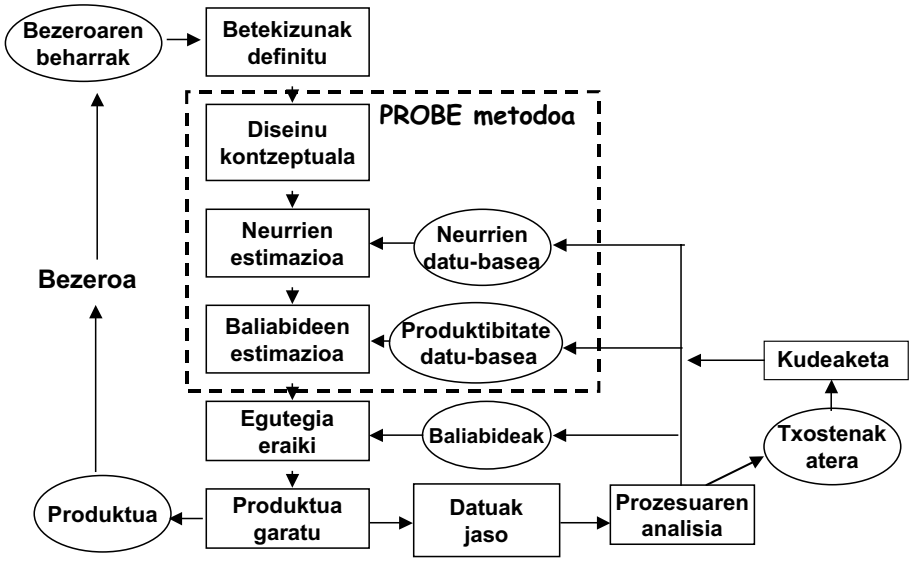
PROBE metodoa, akronimoak dioen bezala, *proxy*etan oinarritzen da estimazioak gauzatzeko. Proxy hitzak beharbada beldurtu egin dezake baina atzean dagoen esanahia, ez da batere beldurgarria. Diseinu kontzeptuala bukatu eta gero, aplikazioak izango dituen objektuak identifikatzeko eskatzen dio metodoak ingeniariari. Objektuak identifikatu ondoren, bakoitzari kategoria bat itsasten zaio eta objektuak zenbat metodo izango dituen zehazten da. Neurrien estimazioak eraikitzen joateko metodoen batez besteko neurri kualitatiboa[11] adierazten da eta informazio hori guztia Humphrey-k berak proposatutako txantiloi (**2.17. irudia**) estandarizatu batera iraultzen da.

9. Ariketen zailtasuna hasierako mailetan ez da handia, helburua ez baita ariketa azkar ebaztea. Helburua da aztertzea ebazpenaren garapena nola joan den egituratuz eta gauzatuz denboran zehar.

10. Egoera horretan, agirietan jasotako informazioa biltegiratu egiten da, gehienetan, euskarri informatiko bat erabilita.

11. Kualitatiboa diogu ez baita zenbakiz adierazten, hitzez baizik: Very Small, Small, Medium, Large eta Very Large.

Geroago ikusiko dugu termino bakoitza zenbaki batez ordezkatzen dela kalkuluak gauzatzeko estimazioetan.



2.16. irudia. Proiektuaren bizi-zikloa.

Informazio kualitatiboa kuantitatibo bihurtzeko aipatutako liburuan taula (**2.17. irudia**) estandar batzuk agertzen dira bihurketa gauzatzeko. **2.17. irudia**k objektuaren kategoria zehaztu eta gero zein den bere batez besteko neurria adierazten du, aplikazioa C++ programazio-lengoaia erabilita garatzen bada. Jakina, taula hori, bere beharretara egokitu beharko du ingeniariak haren programazio-inguru-nean, kategoriak eta zenbakiak alda baitaitezke erabiltzen duen teknologian.

Lerro kopurua C++ programazio-lengoaian

Mota	VS	S	M	L	VL
Calculation	2.34	5.13	11.25	24.66	54.04
Data	2.60	4.79	8.84	16.31	30.09
I/O	9.01	12.06	16.15	21.62	28.93
Logic	7.55	10.98	15.98	23.25	33.83
Set-up	3.88	5.04	6.56	8.53	11.09
Text	3.75	8.00	17.07	36.41	77.66

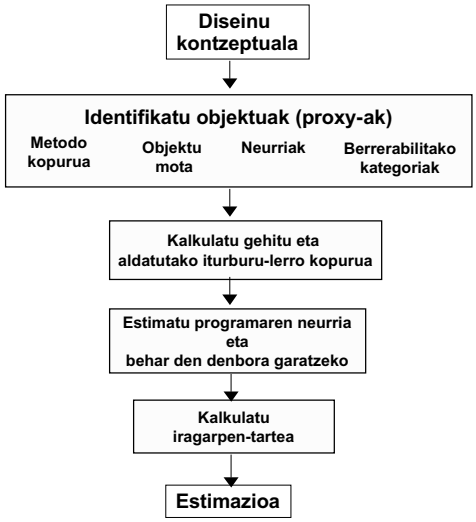
2.17. irudia. Informazio kualitatibotik informazio kuantitatibora.

Table C39 Size Estimating Template

BASE PROGRAM LOC				ESTIMATE	ACTUAL
BASE SIZE (B) => => => => => => => => => =>				0	
LOC DELETED (D) => => => => => => => => => =>				0	
LOC MODIFIED (M) => => => => => => => => => =>				0	
OBJECT LOC					
BASE ADDITIONS	TYPE	METHODS	REL. SIZE	LOC	LOC
TOTAL BASE ADDITIONS (BA) => => => => => =>				0	
NEW OBJECTS	TYPE	METHODS	REL. SIZE	LOC (New Reused*)	
Sarrera	**I/O**	**1**	**large**	**22**	
Zerrenda	**Data**	**3**	**medium**	**27***	
Kalkulatu BB	**Calc**	**1**	**medium**	**11**	
Kalkulatu DS	**Calc**	**1**	**medium**	**11**	
Idatzi emaitza	**I/O**	**1**	**large**	**22**	
TOTAL NEW OBJECTS (NO) => => => => => => =>				**93**	
REUSED OBJECTS					
REUSED TOTAL (R) => => => => => => => =>					
				SIZE	TIME
Estimated Object LOC (E):		E = BA+NO+M		**93**	
Regression Parameters:		β_0 Size and Time		**38**	**110**
Regression Parameters:		β_1 Size and Time		**.8**	**1.5**
Estimated New and Changed LOC (N):		$N = \beta_0 + \beta_1 * E$		**112**	
Estimated Total LOC:		T = N + B - D - M + R		**112**	
Estimated Total New Reuse (sum of * LOC):				**27**	
Estimated Total Development Time:		Time = $\beta_0 + \beta_1 * E$			**250**
Prediction Range:		Range		N/A	N/A
Upper Prediction Interval:		UPI = N + Range		N/A	N/A
Lower Prediction Interval:		LPI = N - Range		N/A	N/A
Prediction Interval Percent:				N/A	N/A

2.18. irudia. Neurriak estimatzeko txantiloia.

Objektu berrien neurriak estimatu eta gero, gehitutako eta aldatutako iturburu-lerro kopurua kalkulatzen da **2.19. irudia**n ikusten den legez. Eragiketa hori burututa, programaren neurri orokorra eta beharko den denbora estimatzen dira erregresio lineala erabilita. Bukatzeko, iragarpen-tartea kalkulatzen da estimazioaren fidagarritasuna zehazteko.

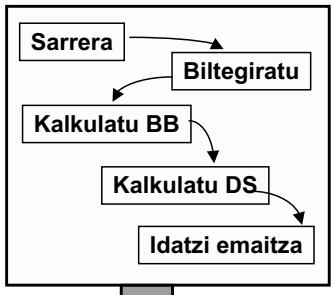


2.19. irudia. Estimazioen egituraketa.

Adibide txiki bat garatuko dugu azaldutako prozesu guztia urratsez urrats aztertzeko. Garatuko dugun aplikazio informatikoak, betekizunen azterketa burututa, bost objektu/metodo nagusi izango ditu eta **2.20. irudia**n agertzen diren ezaugarriak izango ditu. Jakina, ezaugarri horiek zehazteko, ingeniariaren eskarmentua erabili beharko da eta aitzinean (**2.17. irudia**) aipatu dugun taularen informazioa. Informazio hori guztia eraikita eta aipatutako txantiloi estandarrera (**2.18. irudia**) iraulita, historikoki biltegiratuz joan garen informazioa erabilita, neurrien eta denboren estimazioa (**2.21** eta **2.22. irudia**k) gauzatzen da erregresio lineala aplikatuta. Kalkulatutako informazio hori txantiloian biltegiratzen da eta, hala nahi izanez gero, iragarpen-tartea kalkulatzen da. β_0 eta β_1 erregresio-parametroen balioak kalkulatzeko honako formula hauek aplikatzen dira:

$$\beta_1 = \frac{\sum_{i=1}^{n} x_i y_i - n x_{bb} y_{bb}}{\sum_{i=1}^{n} x_i^2 - n(x_{bb})^2}$$

$$\beta_0 = y_{bb} - \beta_1 x_{bb}$$



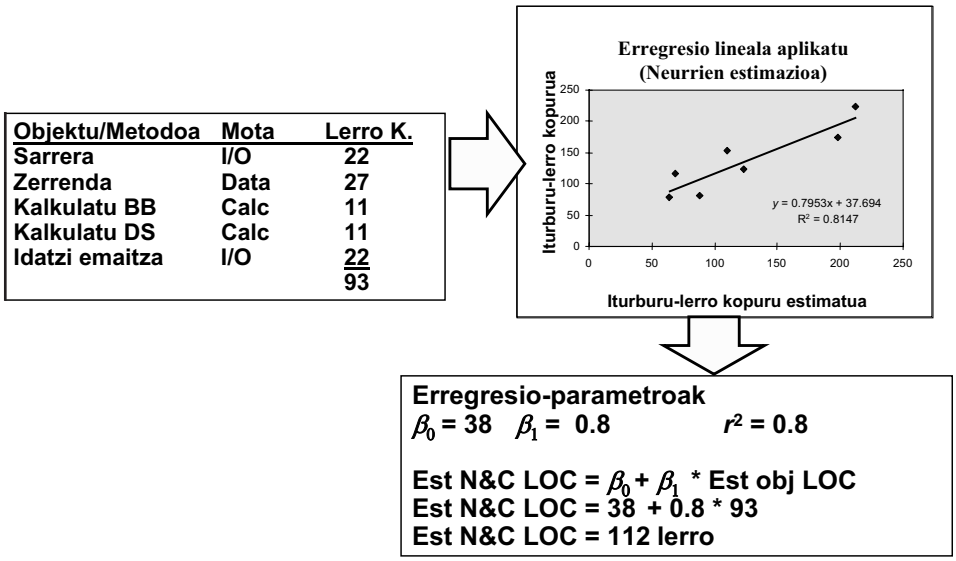
Objektu/Metodoa	Mota	Metodo K.	Neurria	Iturburu L.K.
Sarrera	I/O	1	Large	22
Zerrenda	Data	3	Medium	27
Kalkulatu BB	Cal.	1	Medium	11
Kalkulatu DS	Cal.	1	Medium	11
Idatzi emaitza	I/O	1	Large	22
				93

2.20. irudia. Aplikazioaren objektu/metodoen ezaugarriak.

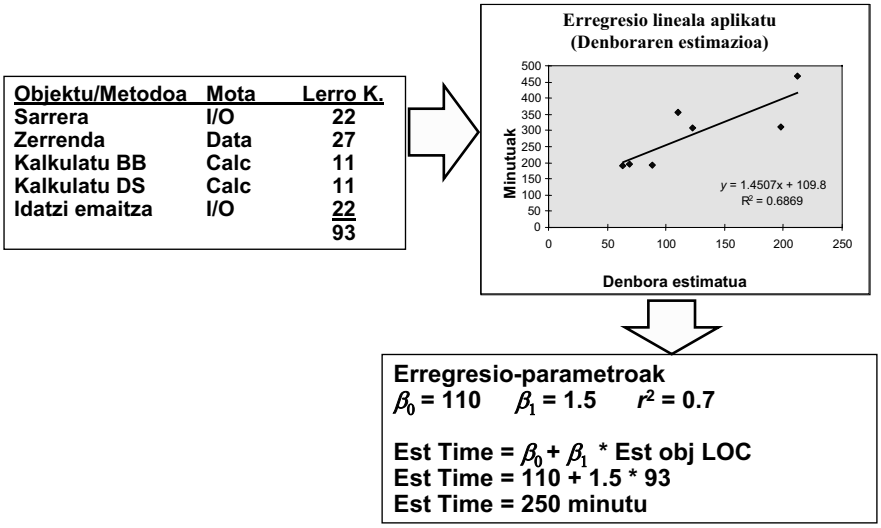
Datuen korrelazio estatistikoa handia edo txikia den jakiteko beste honako formula honetaz baliatuko gara korrelazio-koefizientearen balioa eskuratzeko:

$$r(x,y) = \frac{n\sum_{i=1}^{n} x_i y_i - \sum_{i=1}^{n} x_i - \sum_{i=1}^{n} y_i}{\sqrt{\left[n\sum_{i=1}^{n} x_i^2 - \left(\sum_{i=1}^{n} x_i\right)^2\right]\left[n\sum_{i=1}^{n} y_i^2 - \left(\sum_{i=1}^{n} y_i\right)^2\right]}}$$

Hasieran aipatu dugu ingeniariaren ohiturak, gizakiaren ohitura gehienak legez, trinko sustraituta egoten direla eta nahiko zaila izaten dela ingeniariaren portaera atoan aldatzea. Watts Humphrey-k gauza bera pentsatu zuen PSP metodoa egiaztatzeko orduan, bazekien ez zela nahikoa bat edo bi ingeniarirekin baieztatzea metodoa eta SEI (Software Engineering Institute) erakundeko 300 ingeniari aukeratu zituen aipatutako ariketa kontrolatuak diseinatu eta programatzeko.



2.21. irudia. Neurrien estimazioa.

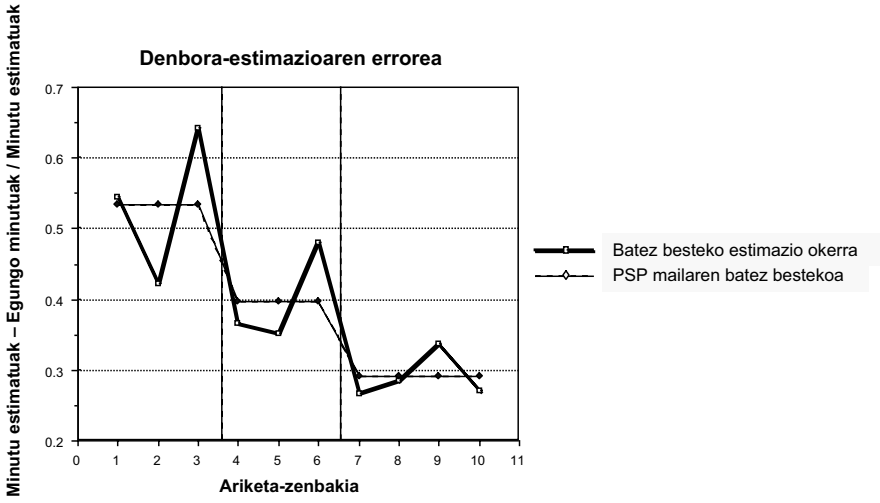


2.22. irudia. Denboraren estimazioa.

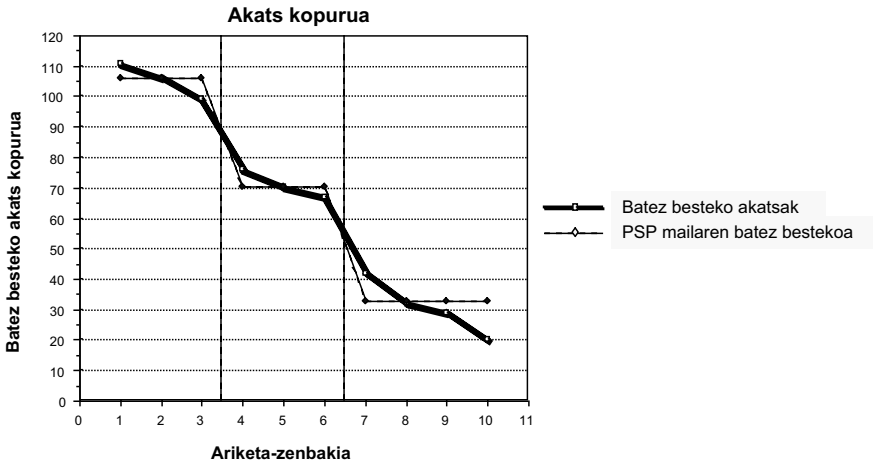
2.23. iruditik **2.25. irudi**ra doazen grafikoek burututako azterketaren emaitzak jasotzen dituzte. Grafiko guztietan antzeko portaera antzematen da, batez ere, laugarren ariketatik aurrera. Gogoratu laugarren ariketa horretatik aurrera, ingeniaria, PROBE metodoa hasten dela erabiltzen estimazioak egiteko. Hala, ez gara harrituko ingeniariaren errorea denbora estimatzeko nabarmen jaisten bada

laugarren ariketatik aurrera, **2.23. irudia**n ikusten den legez. Batez beste, denboraren estimazioaren errorea % 55etik % 27ra jaisten da. Gauza bera gertatzen da konpilazioan eta frogetan sartu ditugun akatsak aztertzen badira (**2.24. irudia**). Hasieran, batez beste 110 bat akats sartzen dira 1.000 lerrotan, eta amaierarako, 20 bat akatsetara jaisten da kopuru hori. **2.25. irudia** ere nahiko deigarria da beste informazio zeharo interesgarri bat ematen digulako, hau da, nahiz eta softwarearen garapen-prozesuaren konplexutasuna handitu laugarren ariketatik aurrera, ingeniariari planifikazio zehatzak eskatuta, beraren produktibitatea ez da jaisten eta ez du lerro gutxiago programatzen, aitzitik, orduko programatzen duen lerro kopurua bertsua da esperimentuaren hasieran eta bukaeran.

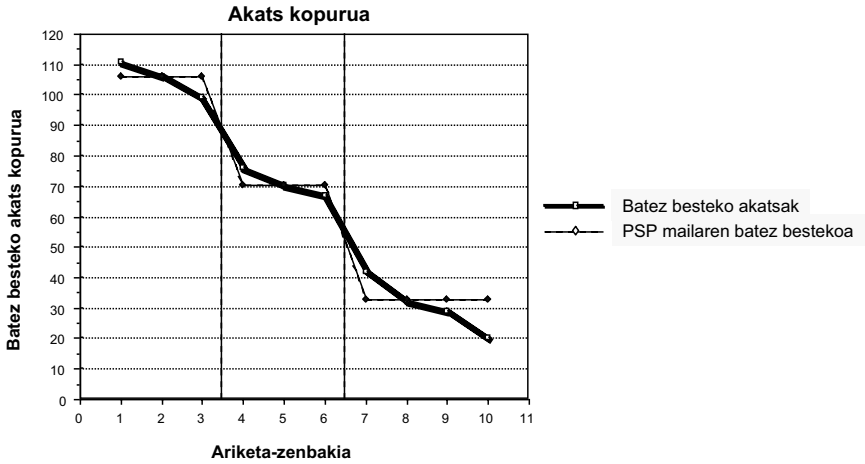
Bukatzeko, **2.26. irudia**k beste honako ezaugarri hau jasotzen du: bizizikloaren atal bakoitzak hartzen duen garrantzia ariketa bakoitzaren garapenean. Hasieran, deigarria egiten da diseinuaren garrantzi xumea eta nola ariketaz ariketa handituz doan nabarmenki. Bestalde, azpimarragarria egiten da ere, kodeketaren, konpilazioaren eta frogen bilakaerak ingeniarien denborak aztertzen direnean nabarmenki jaisten direla diseinuaren kalitatea hobetuz doan heinean. Kontu egin, bukaeran, ingeniariak denbora gehiena diseinatzen igarotzen duela horrek dakarren guztiarekin: akats gutxiago iturburu-kodean eta estimazio hobeak proiektuaren planifikazioan.



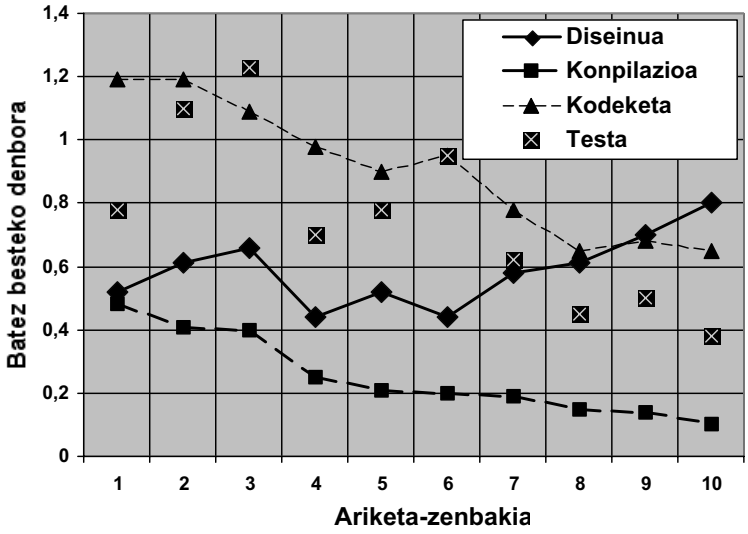
2.23. irudia. Denbora-estimazioaren errorea.



2.24. irudia. Batez besteko akatsak.

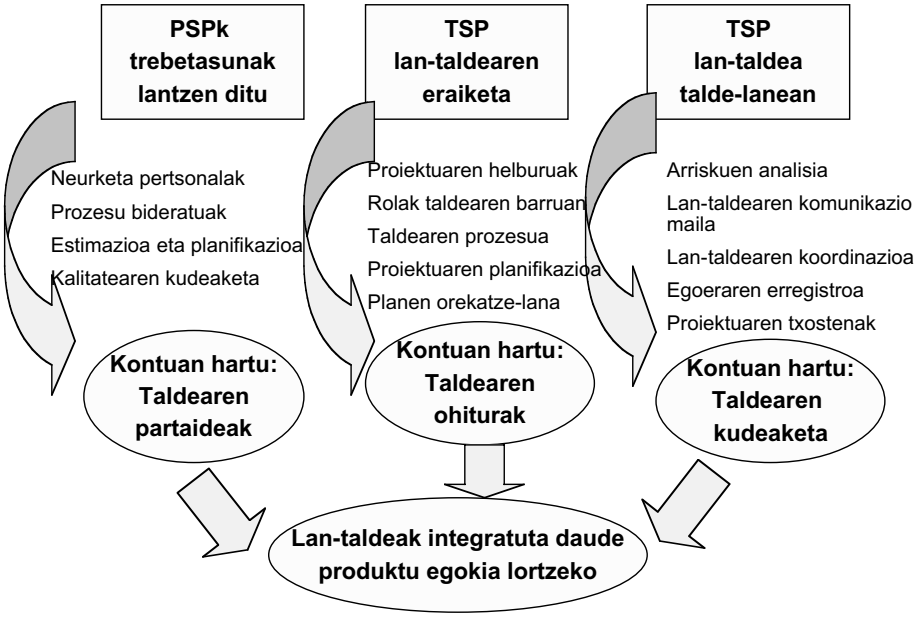


2.25. irudia. Batez besteko lerro kopurua orduko.



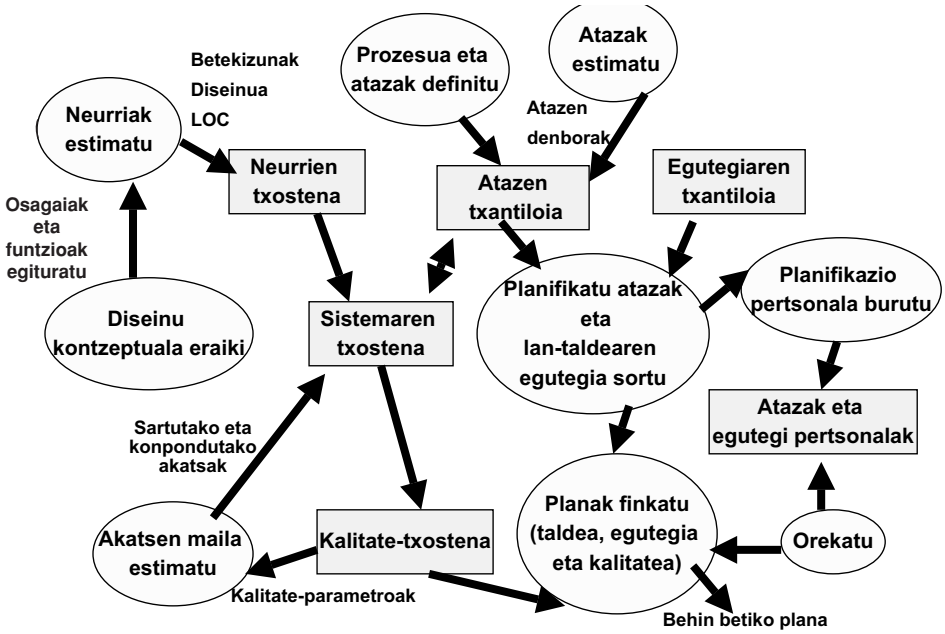
2.26. irudia. Bizi-zikloaren arlo bakoitzaren batez besteko denborak

Aurreko orrietan azaldu dugu CMM eta SPICE erreferentzia-ereduek erakunde osoa hartzen zutela jomugatzat softwarearen prozesuaren heldutasun maila kudeatzeko; bestalde, PSP metodoak ingeniariaren trebetasun pertsonalak hobetzen zituen prozesu estandarizatu bat finkatuz haren eguneroko jardunean. Horretarako, metodoak neurketaren arloa hobetzen zuen hasierako unean eta proiektuaren planifikazioaren kalitate maila geroago, estimazio egokiak gauzatuz. Agerian gelditzen da, jauzi handi xamarra burutu dugula erakundearen goi-mailatik behemailako ingeniariarengana heltzeko. Erreferentzia-ereduetan antzeman egiten zen eta PSP metodoan susmatu, ingeniariaren lana, salbuespenak salbuespen, ez dela era isolatuan gauzatzen proiektuaren bilakaera aurrera doan heinean. Softwareingeniariaren fruituak hutsalak lirateke lan-taldearen talde-lanean ez baleude ondo txertatuta. Aztertzen ari garen software-prozesuaren arloan, TSP (Team Software Process) metodoaren eskuetan gelditzen da betebehar hori. **2.27. irudia**k PSP metodoa eta TSP metodoa nola erlazionatzen diren islatzen du lan-taldearen ikuspegitik. Laburbilduz, PSP metodoa taldearen partaideen eraginkortasunaz arduratzen da eta TSP metodoa, ostera, taldearen kudeaketaz.



2.27. irudia. PSP eta TSP metodoen lan-talde erlazioak.

Proiektu guztietan, hasiera batean, proiektua dimentsionatu ondoren eta sistemaren betekizunak aztertuta, lan-taldea eraiki beharko da proiektua garatzeko. Urrats horretan, argi utzi beharko da zein diren proiektuaren helburuak eta arriskuak eta nola gauzatuko den proiektuaren planifikazioa. Orobat, rolak finkatu beharko dira eta atazak banatu partaideen artean. Horretarako, organigrama estandarizatu (**2.28. irudia**) bati jarraitzen zaio diseinu kontzeptualetik hasita planak finkatu arte. Hori eginez gero, software-ingeniariak, atazak gauzatzen ditu banaka edo azpitaldeka, PSP metodoaren teknikak erabilita.



2.28. irudia. TSP metodoaren organigrama kontzeptuala.

Atazak garatzen ari den aldi berean, PSP metodoa aplikatuta, kalitatezko softwarea eraikitzen saiatuko da ingeniaria proiektuaren kalitate-planaren gomendioei jarraituz. Aholku horietan software-osagai bakoitzaren kalitate-profila zehazten da eta ingeniariak, une oro, osagaiarekin lortzen ari den profila, parekatu egin beharko du, jomuga gisa ipinita dagoenarekin. Profilak prozesuaren metrika batzuk zehazten ditu ziurtatzeko eraikitzen den osagaiak eskatutako kalitatea daukala. Erabiltzen diren metrikak nahiko ezagunak eta orokorrak dira softwarearen ingeniaritzan eta erabat adostuta daude, ez baitira propio diseinatu TSP metodoan erabiltzeko. Honako hiru arlo hauetan sailkatzen dira aipatutako metrikak:

1. Diseinuarekin lotuta daudenak. Diseinuak proiektu osoan duen garrantzia neurtzen dute metrika hauek. Helburua ohartaraztea da zaila dela kalitatezko softwarea eraikitzea, software hori itsumustuan eraiki bada zuzenean igarota betekizunetatik programaziora. Metrika hauek kontrolatu nahi dute, ingeniariak diseinuari eman dion garrantzia egokia dela haren eguneroko jardunean.

2. Berrikuste-ekintzekin kateatuta daudenak. Egindakoa egiaztatzea da metrika hauen helburua, arazoak aurreikusi eta alboratzeko albait arinen. Berrikuskatze-prozesua diseinuari eta iturburu-kodeari aplikatuko zaie bakoitzarentzat metrika propioa zehaztuta. Horrela, aurreko puntuan diseinuarekin legez, hemen ere, ingeniariak, egiaztatu ahal izango du berrikuskatzeari eskaintzen dion denbora egokia den ala ez.

3. Hirugarren arloko metrikek akatsen dentsitatea neurtzen dute konpilazioan eta proba bateratzaileetan. Metrika hauekin, erakundeak edota lan-taldeak akatsen dentsitate desiragarria finkatu behar dute eta kopuru horretara gerturatzen saiatu behar da ingeniaria edo lan-taldea. Konpilazioan aurkitzen den akatsen dentsitateak kodeketaren eta iturburu-kodearen berrikuspenaren kalitatea neurtzen du. Egin kontu ez dela prozesu horien bukaerako emaitzaren kalitatea neurtzen, prozesua bera baizik. Emaitzaren kalitatea, hots, bukaerako softwarearen kalitatea, proba bateratzaileetan aurkitzen den akatsen dentsitateak neurtzen du.

Horrenbestez, metrika guztiak azaleratzen baditugu, osagai bat eraikitzeko aplikatuko den prozesuaren kalitate-profilak bost dimentsio izango ditu:

Kalitate-profilaren dimentsioa	Irizpidea
Diseinu/Kodeketa-denbora	Diseinuari eskainitako denborak kodeketari eskainitakoaren erdia izan behar du gutxienez
Diseinuaren berrikuskatze-denbora	Diseinuaren berrikuskatzeari eskainitako denborak diseinuari eskainitakoaren erdia izan behar du gutxienez
Kodeketaren berrikuskatze-denbora	Kodeketaren berrikuskatzeari eskainitako denborak kodeketari eskainitakoaren erdia izan behar du gutxienez
Akatsen dentsitatea konpilazio-garaian	Konpilazio-garaian akatsen dentsitateak honako muga hauetan mugitu behar du: 10 akats baino gutxiago azaleratu behar dira 1.000 iturburu-kode lerroko
Akatsen dentsitatea proba bateratzaileetan	Proba bateratzaileetan akatsen dentsitateak honako muga hauetan mugitu behar du: 5 akats baino gutxiago azaleratu behar dira 1.000 iturburu-kode lerroko

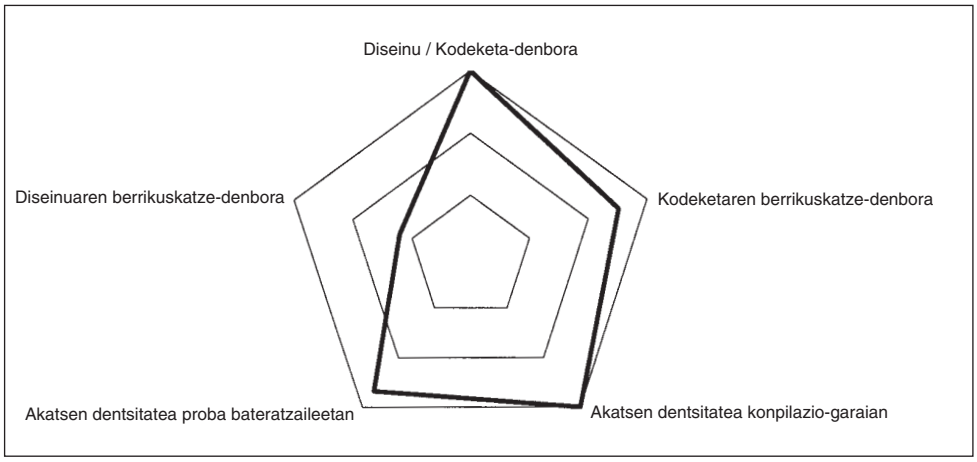
Dimentsio bakoitzaren irizpidearen balioa Humphrey-ren taldeak proposatu zuen bere garaian eta osagai estandarraren kalitate-profil desiragarria finkatzeko erabili ohi izan da harrezkero.

Bukatzeko, ingeniari batek osagai bat garatzerakoan bildu dituen balioak aztertuko ditugu prozesuaren kalitate-profilaren ikuspegitik. **2.1. taula**k ingeniariak biltegiratu dituen balioak jasotzen ditu eta balio bakoitzak kalitate-profilaren dimentsio bat du jomuga. Balio denak batuta aztertzen baditugu, argazki orokorrak **2.29. irudia**ren itxura hartuko luke. Irudiak irizpidearekiko dimentsioaren kalitatea neurtzen du eta argazki azkar baina esanguratsua ematen digu, ingeniariak nola

bete duen ibilbidea diseinu kontzeptualetik proba bateratzaileetaraino. Gure adibidean esan dezakegu ingeniaria txintxo samar portatu dela orokorki, dimentsio gehienetan eskatutakoa bete baitu. Agian, ohartarazi beharko litzaioke osagai bat eraikitzerakoan diseinuaren berrikuskatze-denbora zaindu egin beharko lukeela, oso gutxi ematen baitu berrikuskatze-arlo hori betetzen.

Kalitate-profilaren dimentsioak	Kalitate-profilaren dimentsioaren balioak
Neurria	160 iturburu-kode lerro
Diseinu-denbora	40 minutu
Diseinuaren berrikuskatze-denbora	8 minutu
Kodeketa-denbora	74 minutu
Kodeketaren berrikuskatze-denbora	31 minutu
Akats kopurua konpilazioan	0 akats
Akats kopurua proba bateratzaileetan	1 akats
Diseinu-denbora/ Kodeketa-denbora	0.54
2 x (Diseinuaren berrikuskatze-denbora / Diseinu-denbora)	0.40
2 x (Kodeketaren berrikuskatze-denbora / Kodeketa-denbora)	0.84
Akatsen dentsitatea konpilazio-garaian	0 akats / 1.000 iturburu-kode lerro
Akatsen dentsitatea proba bateratzaileetan	6.25 akats / 1.000 iturburu-kode lerro

2.1. taula. Kalitate-profilaren dimentsioen balioak.



2.29. irudia. Kalitate-profilaren dimentsioak.

2.5. MUTURREKO PROGRAMAZIOA

Txina zaharrak Taoa eratu zuen Naturaren ezaugarriak antolatzeko. Testuinguru horretan, neguak uda dakar, iluna argian urtzen da eta gaztaroak aurretik dauzka biltegiratuta zahartzaroko ajeak. Yin eta Yang indar osagarrien eragina, nonahi eta noiznahi, azaleratzen da. Batak, bestea behar du une oro, ez baitira gauza isolatuta agertzeko.

Parajeago etorrita eta asagoago joanda berriro, denboran, koka gaitezen Vatikanoan, hain zuzen, XVI. mendearen hasieran. Vatikanoko Signatura gelan, Rafael Sanzio (1483-1520), gerora entzute handiko Rafael, *Atenasko Eskola* fresko handia margotzen ari da Julio II.a aita santuaren agindupean. Perspektiba maisuki erabilita, ikuslearen arreta berak nahi duen tokira erakartzen du pintoreak. Eta zein da toki hori? Freskoaren erdigunea.

Erdigunean, Platon eta Aristoteles daude bakoitza liburu batekin, Platonek bere *Timeoa* eta Aristotelek bere *Etika*. Teoria eta ideien mundua alde batetik; Naturaren formena eta behaketena bestetik. Batak, goi zerutiarrera zuzentzen du hatz erakuslea; besteak, esku-ahurra, behegaineruntz. Milaka urte geroago eta berriro ere, Yin eta Yang indarrak pil-pilean. Areago, Rafaelek, perspektibaz baliatuta, bertikaltasunarekin bateratzen du sakontasuna, arreta bilatzeko. Baina jakin badaki, perspektibaren sakonera orekatu egin behar duela lehen planoarekin eta gainera horizontalki, indarrak parekatzeko. Horretarako, filosofoz betetzen du eszena, poliki-poliki, hierarkikoki pilatuta, Platon eta Aristotelesen ingurumarian.

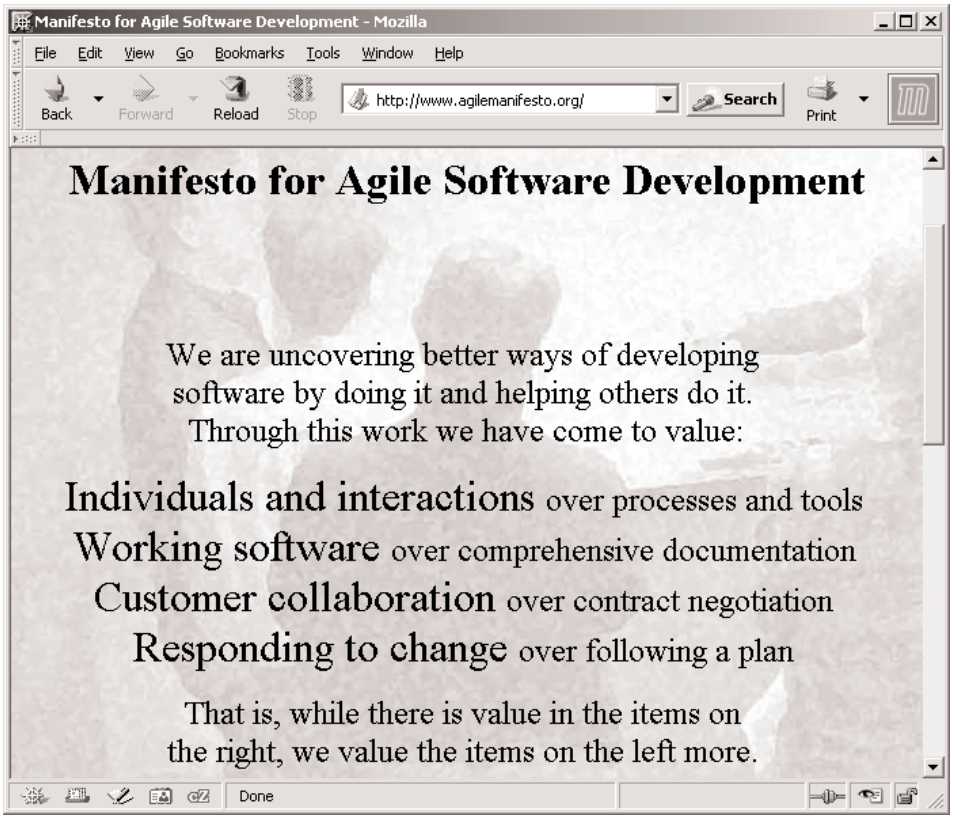
Freskoan, bai edukian baita edukiontzian ere, indar bi jartzen dira jokoan. Indar bi edukiontzian, perspektiba orekatzeko; indar bi edukietan, munduaren ikuspegi bi orekatzeko. Badirudi indar bat bestearen gehiegikeriak moteltzeko eraiki dela *aurea mediocritas* baten mesederako. Informatikan ere, azken boladan, erreferentzia-ereduen arloari bere Yang indarra sortu omen zaio. Indar berri hori gehiegikeriak apaltzeko etorri omen da, erreferentzia-ereduaren burokrazia astunari hegoak ipintzeko. Mugimendu iraultzaile postmoderno gehienak legez, indar berri horren zaleek, Interneten (`www.agilemanifesto.org`) kokatu dute beren manifestua garaiek hala agintzen baitute. Agiriak (**2.30. irudia**) aipatutako Yin eta Yang indarren borroka irudikatzen du, hots, nirekin edo nire kontra, dena edo ezer ez dinamika ezaguna. Metodo arinen, azkarren eta leunen zaleak, pertsona eta elkarrekintzak maite ditu, ez prozesuak eta tresnak. Funtzionatzen duen softwareak lehentasuna du dokumentazioaren aurrean. Bezeroaren laguntza nahitaezkoa da proiektuaren bizi-ziklo osoan. Aldi berean, laguntza hori ondo gauzatzen bada, ez dago zertan denbora gehiegirik galdu kontratuen hitzarmen astunetan. Aldaketak gertatzen direnean erantzunak azkarra izan behar du eta horretarako ez dugu ur handitan sartu behar proiektuaren planifikazioa antolatzerakoan. Emaitza azkarrak nahi baditugu zenbat eta planifikazio gutxiago hobe.

Gehienok ados egongo gara, agiriaren bultzatzaileek dioten bezala, agiriaren ezkerreko ezaugarriak desiragarriak direla eta norbaitek esan lezake, gainera, beste hiruzpalau gehiago jarriko lituzkeela. Arazoa eskuineko ezaugarriekin sortzen da: norbaitek ezagutzen ote du proiektu-arduradun bat prozesuak eta tresnak baztertuko lituzkeena elkarrekintzak bultzatuta horren ordez? Eta gauza bera beste ezaugarri guztiekin. Nork zokoratuko lituzke erabat eskuineko ezaugarriak eskerrekoez ordezkatuz? Kent Beck eta enparauek badirudi baietz. Ez naiz ni orain hemen hasiko Beck jaunaren merezimenduak[12] zalantzan jartzen, asko baitira. Hemen ez gara informatikari lurtarraz arituko, harago joango gara eta haren izaera mesianikoaz jardungo gara. Israelgo erreinua berreraikitzeko beste mesias batek aurrea hartu zionez, gure salbatzaile berriak informatikara bideratu zituen bere ahaleginak softwarearen erreinua berreraikitzeko.

Mesianiko guztiek legez, gure informatikari mesianikoak bere teofania propioa izan zuen eta kontenplazio-egoera hartako estasiaren emaitza, liburu batean jaso zuen: *Extreme Programming Explained. Embrace Change*.

Egoerak hala behartuta eta telepredikarien zantzuei jarraituta ideiak barreiatzeko, liburuak sekulako arrakasta izan zuen salmentetan. Laburbilduz, esan dezagun liburuak muturreko programazioaren (Extreme Programming) dogma nagusiak jasotzen dituela. Saltzaile onen teknikak erabilita, hitz egoki eta zirraragarria bilatu behar zen eta hor zegoen *extreme* hitza prest horretarako, kiroletan erruz erabiltzen zena rafting, snowboard, sky surfing eta halako kirol motak izendatzeko. Hitzak[13], azkartasuna, arriskuen aurrean ausardia eta, batez ere, berritasuna islatzen zituen. Jakin badakigu berritasunak batez ere haustura adierazten duela egungo kultura postmodernoan; haustura, lehen egindakoarekin; haustura, iraganarekin eta haustura historiarekin. Beck jaunak, lurzoru urratu gabe horretan ereiten ditu bere dogma nagusiak aldiko, sasiak sutuz.

12. Smalltalk Best Practice Patterns. 1996.
    Refactoring: Improving the Design of Existing Code. 1999.
    Test-Driven Development: By Example. 2002.
    JUnit Pocket Guide. 2004.
13. Gerora, *agile* hitzak ere antzeko fortuna izango zuen.



2.30. irudia. Garapen bizkorraren aldeko manifestua.

Hona hemen erlijio berriaren hamabi aginduak:

1. Planifikatu behar baduzu, planifikatu azkar planifikazioaren jokoa erabilita.
2. Produktuaren bertsioek txikiak izan behar dute eta azkar plazaratu behar dira.
3. Metaforak erabili garapena bideratzeko. Metaforak kontakizun xume bat eraikiko du sistemak nola funtzionatzen duen adierazteko. Kontakizun horrek garapen osoa gidatu beharko luke.
4. Diseinu xume bat nahikoa da garapena aurrera eramateko. Diseinuak ahal denik eta txikerrena izan behar du. Konplexutasuna, antzeman bezain pronto baztertu behar da.
5. Une oro probak gauzatu. Aldi oro, programatzaileek proba bateratzaileak idatzi behar dituzte osagaiak egiaztatzeko. Bezeroek ere proba funtzionalak idatzi behar dituzte betekizunak ondo betetzen direla ziurtatzeko.

6. Birfaktorizatu (Refactoring) zure iturburu-kodea. Eragiketa honekin programazioaren eraginkortasuna hobetzen da. Horretarako, osagaiaren iturburu-kodea aldatu egiten da haren portaera ukitu gabe, osagaiaren interfazeak bere horretan uzten baitira.

7. Programazioa bikoteka gauzatuko da. Iturburu-kode guztia binaka idatziko da. Parekatze-prozesua dinamikoa izango da erakundean programatzaileen arteko promiskuitatea bultzatuz. Programatzaile bat inplementazio-lanetan arituko da eta besteak ikuspegi estrategikoa landuko du garatzen ari diren osagaia sistema orokorrean ondo txertatzeko.

8. Jabetza kolektiboa bultzatu. Edozeinek edozein unetan alda dezake sistemaren iturburu-kodearen edozein zati.

9. Iturburu-kodearen etengabeko integrazioa gauzatu une oro. Ataza bukatuta eraiki den iturburu-kode guztia berehala integratu eta probatu behar da sistema osoa berreraikita. Akatsen bat aurkitzen bada, integratzen ari direnek konpondu behar dute arazoa.

10. Astean, ez lan egin 40 ordu baino gehiago. Aparteko orduak sartu behar badira erakundean, ez sartu bi aste baino gehiago jarraian. Proiektu batek aparteko orduak behar baditu gauzatzeko, arazo larriak izango ditu etorkizunean.

11. Bezeroak *in situ* egon behar du. Txertatu bezeroa lan-taldean hasiera-hasieratik eta era egonkorrean. Bezeroak une oro prest egon behar du galderei erantzun eta zalantzak argitzeko.

12. Kodeketaren estandarizazioa bultzatu. Iturburu-kodea eraikitzeko, programatzaile guztiek erakundean onartutako estandarrei jarraituko diete iturburu-kodearen itxura bateratu bat lortzeko. Estandarrak komunikazioa bultzatzeko eraikitzen dira eta haien erabilera ez da behartuta inposatu behar.

Agian, irakurle, irakurri dituzun hamabi aginduek *totum revolutum* nahaspilatsu bat dirudite, aldarrikapen sindikalak (10. agindua) eta lan-promiskuitatea (7. agindua) nahasten baitira bizpahiru ideia onekin: proba bateratzaileak, integrazioa eta birfaktorizazioa. Beste guztiak tautologia hutsa dirudi eta begi-bistakoa gainera. Akaso irakurle, nire esku-hartzea egon dela suma dezakezu hainbat agindutan. Hori hala ez dela egiaztatzeko hona hemen beste perla batzuk:

- Binakako programazioak hobeto funtzionatu zuen monitorea atzera mugitu zenuenean hogeita hamar zentimetro (7. orrialdea).

- Betekizunak ez daude argi hasieran. Bezeroek ezin digute esan zehatzmehatz zer nahi duten (19. orrialdea).

- Softwarean dena aldatzen da. Betekizunak aldatzen dira. Diseinua aldatzen da. Salmentagunea aldatzen da. Teknologia aldatzen da. Taldea aldatzen da (28. orrialdea).
- Zenbat eta xumeago izan zure sistema, orduan eta gutxiago komunikatu behar duzu (31. orrialdea).
- … infernura jaisteko unea da. Ez bazoaz ziztu bizian beste norbait joango da zure ordez eta zure janaria jango du (33. orrialdea).
- Erretorikaren logikak eta ahalmenak ez diote eraginik egiten kodeari. Unibertsitate-tituluek eta soldata handiek ez dute kodea txunditzen. Hor ari da pozik egiten, zuk esan diozuna egiteko. Ez badu egiten zuk pentsatzen duzuna egin behar duena, zure arazoa da (44. orrialdea).
- Apala izan zaitez eta bila ezazu xumetasuna. Programatzaile, joan zaitez albait arinen Zen ikastaro batera. Azken hau nirea da !!!!

Edonondik irekita bitxitegia dirudi liburuak. Planifikazioaren eta analisiaren arloek huskeria dirudite. Diseinua beharrezkoa da baina zenbat eta gutxiago hobe, arinak eta azkarrak izan behar baitugu. Ordainetan zer dugu? Barry Boehm-ek dioen bezala, programatzaile cowboya. Harreman promiskuoetan endredatuta dabilena, gaineratuko nuke nik.

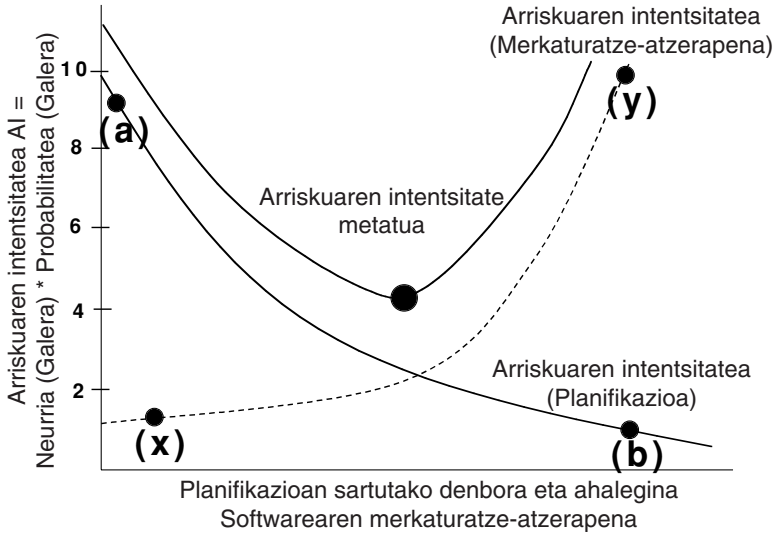
Informatikari lurtar arruntak gauza berri gutxi aurkituko du metodo arinen eta muturreko programazioaren zaleen artean ikono bihurtu den liburu horretan. Batez ere programazioaren zenbait teknika on eta bizpahiru ohitura egoki: proba bateratzaileak, birfaktorizazioa eta integrazioaren garrantzia. Eta kito!

Azken boladan, Barry Boehm-ek ahalegin handiena egin du ikuspegi erabat kontrajarri biak batera aztertzeko eta bakoitzaren egoitza non dagoen zehazteko.

Bai ikuspegi tradizionalak[14] eta bai metodo arin guztiek, arriskuen kontrolarekin lotzen dute proiektu informatikoaren kudeaketa ona. Arriskua handia bada, baliabideek ugaldu egin beharko dute eta faktore bakoitzak zer eragin duen zehatzmehatz aztertu beharko da. Horrela, adibidez, planifikazio astunegiak burokrazia gehiegi sar dezake erakundean eta prozesua mantsotu. Horrek, produktuaren merkaturatze-data atzera dezake eta galerak sortu. Egoera horretan, komenigarria izango da planifikazioa optimizatzea plan leunagoak eta dinamikoagoak eratuta. Baina hori bakarrik egin ahal izango dugu, erakundearen heldutasun maila egokia bada honelako aldaketak gauzatzeko era sistematikoan eta kontrolatuan. Baldintza hori betetzen bada, Horaziok zioen *aurea mediocritas* hura aurkitu beharko da, sasi

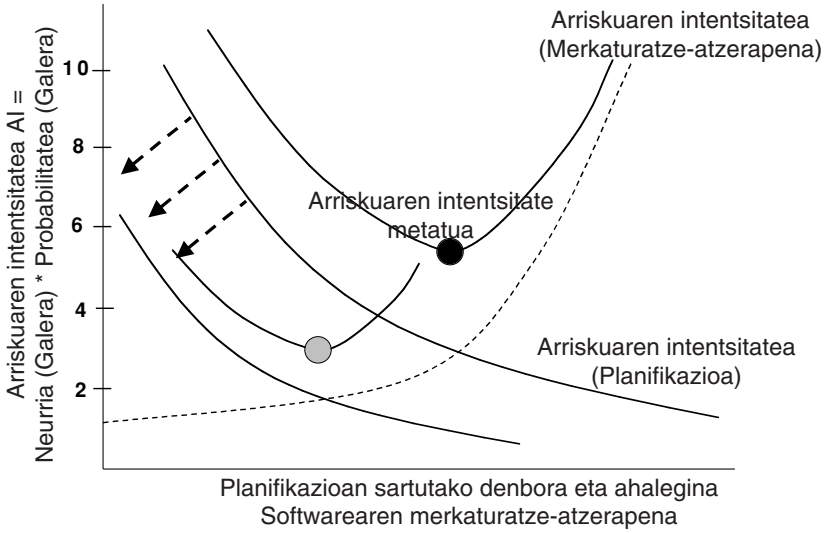
14. Gogoratu ikuspegi tradizionalak berebiziko garrantzia ematen diela proiektuaren aurreplanifikazioari, betekizunen azterketa sakonari eta analisi txukunari. Aitzitik, metodo arinak gehiago arduratzen dira bizi-zikloaren azken urratsez, hots, diseinuaz eta batez ere programazioaz.

guztien gainetik eta hodei guztien azpitik, oreka bilatuz, metodo arinen eta ikuspegi tradizionalen artean. Azaldu dugun egoera irudikatzeko eta beraren portaera aztertzeko **2.31. irudia**n agertzen den grafikoaz baliatuko gara.



2.31. irudia. Arriskuaren intentsitatea neurtzen.

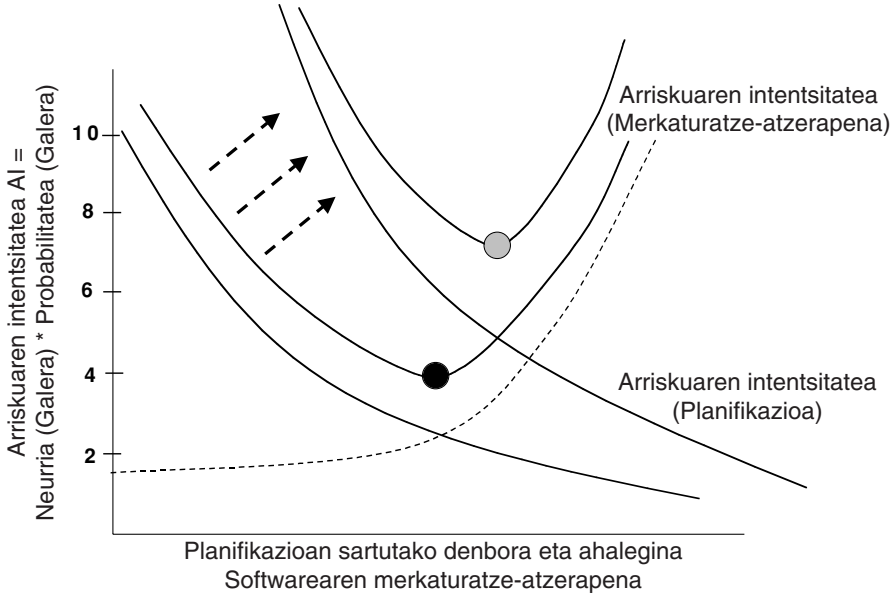
Grafikoan (a) eta (b) puntuek honako hau adierazten dute: proiektu informatikoaren planifikazioan sartutako denbora eta ahaleginaren baliabideak nola islatzen diren arriskuaren intentsitatean. Puntuak kokatuta dauden lerro kurbatu beherakorrak joera hori du baliabideen eragina handia delako arriskuaren intentsitatea jaisteko. Beheranzko joera hori goranzko beste joera batzuekin parekatu behar da arriskuaren intentsitate orekatua aurkitzeko. (*x*) eta (*y*) puntuek beste honako hau adierazten dute grafikoan; softwarearen merkaturatzearen atzerapenak nolako eragina duen arriskuaren intentsitatean. Honako honetan, lerro kurbatua gorakorra da arriskuaren intentsitatearekiko. Zenbat eta gehiago atzeratzen dugun produktu horren merkaturatze-data, orduan eta arrisku gehiago dugu kale egiteko gure ahaleginean. Grafikoaren erdian dagoen puntuak argi uzten dio arriskuen arduradunari zein den hartu behar duen arrisku maila egokiena. Puntu horretatik ezkerretara igo egiten da arriskuaren intentsitatea eta gauza bera gertatzen da eskuineko aldera jotzen badugu.



2.32. irudia. Arriskuaren intentsitatea tokiz aldatzen planifikazioa hobetuta.

Ondorioetara jotzen badugu, argi ikusten da tokian tokiko eta unean uneko egoerak zeharo baldintzatzen duela arriskuaren intentsitatearen maila, hau da, proiektu berak erakunde desberdinetan portaera zeharo desberdina izan dezake erakundearen ekosistema ez badugu ezagutzen eta ezin dugu jakin, proiektuaren ezaugarriak ezagututa bakarrik, nolako arriskuetan sartuko den. Erakundeak planifikazio arloa ondo menderatzen badu eta ohituta badago planifikazio egokiak antolatzen bere heldutasun maila softwarearen prozesuaren arloan ona delako, proiektu batean egoerak hala eskatzen badio, gauza izango da aipatutako lerro kurbatu beherakorra tokiz aldatzeko. Egoera berri horretan (**2.32. irudia**) arriskuaren intentsitate metatua ere hobera aldatuko da.

Bestalde, erakundeak sistema kritiko bat eraiki behar badu eta merkaturatze-atzerapenaren profila aldatzen ez bada, erabaki dezake planifikazioaren arloak izugarrizko eragina izango duela proiektuaren arrakastan. Bigarren egoera horretan ere, lerro kurbatu beherakorra aldatu (**2.33. irudia**) egin da tokiz. Mugimendu horrek, jakina, arriskuaren intentsitate metatua mugiarazi egingo du.



2.33. irudia. Arriskuaren intentsitatea tokiz aldatzen planifikazioa makalduta.

Azaldu ditugun adibide txiki horiek ondo kokatuta uzten dute gaia. Arazoa ez da konpontzen ikuspegi bat aukeratuta eta bestea baztertuta. Bakoitzak bere alde mengelak eta sendoak ditu eta unean uneko aukerak egin beharko dira proiektu informatikoa ondo bideratzeko, hori bai, beti kontuan izanda, metodo arinak osatuko duela ikuspegi tradizionala eta ez alderantziz. Nik gainera, erakundeari, beste aurrebaldintza bat jarriko nioke metodo arinetan eta muturreko programazioetan sartu baino lehen: softwarearen prozesuaren heldutasun maila homologatua egon dadila CMM erreferentzia-ereduaren hirugarren mailarekin. Azken batean, bai metodo arinak bai muturreko programazioak programatzaile bikainak behar dituzte proposatzen dituzten teknikak ondo eratzeko. Zalantzan jartzen dutena hauxe da: programatzaile horiek zuzendaririk behar ote duten ala ez, eta partiturak aurretik idatzita egon behar duen ala ez. Ziur musika klasikoaren zaleek gauza bat diotela eta jazz-zaleek beste bat.

2.6. LABURPENA

Hainbat urtetan softwarearen ingeniaritzaren ardura nagusia produktuaren ingeniaritzan egon da kokatua. Hori dela eta, hainbat baliabide eta tresna teknologiko aurreratu garatu dira produktu horren bilakaera hobetzeko eta azkartzeko. Egoera horrek lilura eta, ez gutxitan, zaletasun patologiko bat sortu du informatikariongan. Sarritan teknologiarekiko xarmak ezkutatu egiten du edozein teknologiaren atzean dagoena ez dela teknologia bera, teknologia horrek automatizatzen duen prozesua baizik.

Egun erakunde informatikoen heldutasun maila erreferentzia-eredu estandarrak (CMM, SPICE, PSP...) erabilita neurtzen da. Hala, edozeinek, edozein tokitan eta edozein unetan erakunde informatikoaren *argazki* bat atera dezake eta erabili, erakunde beraren heldutasun maila hobetzeko, edo, beste erakundeekin konparaketak egiteko, erreferentzia-eredu neutral bat erabilita.

2.7. OINARRIZKO BIBLIOGRAFIA

Ahern, D.; Clouse, A. eta Turner, R. (2003): *CMMI Distilled: A Practical Introduction to Integrated Process Improvement*, Addison Wesley.

Beck, K. (2002): *Una explicacion de la Programación Extrema. Aceptar el cambio*, Addison Wesley.

Beck, K. eta Fowler, M. (2001): *Planning Extreme Programming*, Addison Wesley.

Chrissis, M.; Konrad, M. eta Shrum, S. (2003): *CMMI: Guidelines for Process Integration and Product Improvement*, Addison Wesley.

El Eman, K.; Drouin, J. eta Melo W. (1998): *SPICE. The theory and practice of Software Process Improvement and Capability Determination*, IEEE Computer Society Press.

Humphrey, W. S. (1995): *A discipline for Software Engineering*, Addison Wesley.

————, (1999): *Introduction to the Team Software Process*, Addison Wesley.

————, (2001): *Introducción al Proceso del Software Personal*, Addison Wesley.

Kasse, T. (2004): *A Practical Guide to CMMI Implementation*, Artech House.

Paulk, M.; Weber, C.; Curtis, B. eta Chrissis, M. (1995): *The Capability Maturity Model for Software: Guidelines for Improving the Software Process*, Addison Wesley.

2.8. HELBIDE INTERESGARRIAK

Zer dago bertan	Web gunea
Software Engineering Institute (SEI)	`http://www.sei.cmu.edu/`
European Software Institute(ESI)	`http://www.esi.es/`
Software Process Improvement and Capability dEtermination(SPICE)	`http://www.sqi.gu.edu.au/spice/`
Software Process Dashboard Project	`http://processdash.sourceforge.net/`
Personal Software Process	`http://www.sei.cmu.edu/tsp/psp.html`
Team Software Process	`http://www.sei.cmu.edu/tsp/tsp.html`
Manifesto for Agile Software Development	`http://agilemanifesto.org/`
Agile Alliance	`http://www.agilealliance.org/`
Extreme Programming: A gentle introduction.	`http://www.extremeprogramming.org/`
Software Process Engineering Metamodel, version 1.1	`http://www.omg.org/technology/documents/formal/spem.htm`

3. Objektuetara zuzendutako programazioa

3.1. JAVA: HISTORIA

Programazio-lengoaien historia deigarria izan ohi da askotan eta bitxia ez gutxitan. Javaren historia deigarria eta bitxia da. Egun, Kinnerley ikerlariaren arabera 2.000 programazio-lengoaia dagoke munduan. Hainbeste programazio-lengoaia egonda zerk erakartzen du programatzaile bat Javan programatzera? Erantzuna ez da begi-bistakoa.

Ezin da ukatu Javak halako lilura sortzen duela programatzaileengan. Ez ginateke gauza berririk esaten ariko, askotan esaten dena errepikatuko bagenu hemen: Javak jakin izan du egon behar zuen tokian eta egon behar zuen unean egoten.

Arrakastaren hasiera-data ezaguna da dagoeneko, 1995a, tokia ere bai, Sun Microsystems enpresa informatikoaren ikerkuntza-laborategian. Urte horretan James Gosling-ek eta bere taldeak aurretik eraikitako Oak programazio-lengoaia birbataiatu zuten eta Java izena jarri zioten. Baina zergatik bataio berri hori eta zergatik izen berriarekin? Gurasoek jakin jakingo ote zuten egindako haur-hilketak eta identitate berriarekin sortutako programazio-lengoaiak hain arrakasta handia izango zuenik? Nik nire zalantzak ditut.

Aipatutako ikertzaileek 1995ean argi ikusi zuten informatikaren etorkizuna sistema banatuetan eta Interneten zegoela eta hortik jo behar zela aurrera. Baina saltsa horietan sartzeak arriskuak zituen, aztertu behar zen sistemaren konplexutasuna biderkatu egiten baitzen. Aplikazioen segurtasuna asko zaindu behar zen, amaraunean sistema eragile desberdinekin lan egin behar zen eta programak ordenagailu mota desberdinetan egikaritu behar ziren.

Suneko ikertzaileek antzeko arazoak konpontzen ari ziren 1991tik. Urte hartan Patrick Naughton-ek, Chris Warth-ek, Ed Frank-ek eta Mike Sheridan-ek, James Gosling-en gidaritzapean, ikerkuntza-talde bat sortu zuten programazio-lengoaia berri bat garatzeko. Lengoaia horrek kontsumorako diseinatutako etxetresna elektronikoak kudeatzeko balio behar zuen. Horretarako, diseinatu behar zen programazio-lengoaiak kode fidagarria sortu behar zuen eta kode hori erraza izan behar zen garatzen. Hasieran Gosling-ek eta bere taldeak C++-ean finkatu zuten beren interes nagusia baina konturatu ziren C++-en eraginkortasuna

ona izanda ere, baliabide asko eskatzen zituela proba- eta arazketa-garaian. Zerbait sinpleagoa behar zen, eta gainera, tresna elektroniko desberdinetan funtziona zezakeena. Aldi berean Sunek FirstPerson enpresa filiala sortu zuen proiektuari bere identitatea emateko, argi utziz zer egin nahi zen eta zertarako.

Testuinguru horretan garatu zuen Suneko ikertzaile-taldeak Oak programazio-lengoaia. Lengoaiaren eraginkortasuna eta egokitasuna neurtzeko, bi proiektu desberdinetan erabili zen: Green eta VOD (Video On Demand). Green proiektua, egun domotika deritzon ingurunean kokatzen zen eta haren helburua etxeetan dauden tresna elektronikoak kontrolatzea zen. Bigarren proiektuak telebista interaktiboa zuen helburu. Bai bata eta bai bestea ez ziren inoiz komertzialak izatera heldu baina baliagarri eta ezinbesteko bihurtu ziren Javaren (**3.1. taula**) garapenean. 1994ko udaberrian FirstPersonek ateak itxi zituen.

FirstPerson itxi ondoren Gosling-ek eta bere taldeak ez zuten etsi, berehala konturatu ziren Internetek etorkizunean arrakasta handia izango zuela eta lehia horretan parte hartu nahi zuten. Horretarako, ez zuten hutsetik hasi behar izan tresna elektronikoen munduak antzekotasun handiak baititu Internetekin. Bietan, kudeatu behar diren inguruneak banatuak dira, sistema eragile eta tresna desberdinetan funtzionatu behar du eta sortu behar den softwareak eramangarria izan behar du. Beraz, bidearen hiru laurdenak eginda zeuden, bakarrik gelditzen zen marketing egokia antolatzea eta Microsoften alternatiba *erreal* gisa aurkeztea.

Ezaugarriak	Javak eskaintzen duen teknologia
Objektuetara zuzendua	`Egun ba ote dago beste era bat?`
Aplikazioari datu-baseetara atzitzeko bideak eskaini SQL aginduak erabilita	`Java Database Connectivity(JDBC)`
Osagaietan oinarritutako softwarea garatzeko aukera eman	`EJB`
Klase-hierarkiak eskaini, erabiltzailearen interfaze grafikoak garatzeko	`AWT/SWING`
Garapen-tresna aurreratu asko	`Eclipse/JDeveloper/ NetBeans`
Liburutegi generikoak	`JGL/JDSL/`...
Aplikazio banatuak sortzeko azpiegitura	`RMI`
Web zerbitzari batean eta haren ingurunean egikaritzen diren aplikazioen arteko informazioa trukatzeko baliabideak	`SERVLET`
Erreflexio mekanismoak eduki, klaseak, interfazeak eta objektuak aztertzeko exekuzio-garaian	`REFLECT`
Aplikazioaren dokumentazioa eta laguntza-fitxategiak automatikoki sortzeko ahalmena	`JAVADOC`
Zubi bateragarriak CORBA eredu estandarrarekin	`ORG.OGM.CORBA`
Tresnak eskaini sortzen den softwarea enkriptatu, ziurtatu eta baimentzeko	`SECURITY/JAR`
Web zerbitzuak	`Java Web Services Developer Pack`

3.1. taula. Javaren ezaugarri nagusiak.

3.2. JATORRIZKO DATU MOTAK ETA MOTA HORIEN KAPSULAKETA

Programatzaileak objektuetara zuzendutako programa bat egituratzen duenean aldagaiak erazagutzen ditu une oro, objektuen egoera ezagutzeko. Erazagutzen dituen aldagai horiek objektuaren atributuak izango dira. Objektuetara zuzendutako programazio-lengoaiek aldagai edo atributu horiek erazagutzeko jatorrizko datu motak eskaintzen dituzte programatzailea hutsetik programatzen ez hasteko. Gero, programatzaileak, oinarrizko datu mota horiekin klaseak sortuko ditu eta erabili ahal izango ditu klase horiek beste aldagai berri batzuk erazagutzeko.

Mota	Biltegiratzea	Balio Minimoa	Balio Maximoa
byte	8 bit	-128	+127
short	16 bit (2 byte)	-32768	+32767
int	32 bit (4 byte)	-2147483648	+2147483647
long	64 bit (8 byte)	-9223372036854775808	+9223372036854775807
float	32 bit (4 byte)	$-3.4 \times 10^{+38}$ 7 digitu esanguratsu	$+3.4 \times 10^{+38}$ 7 digitu esanguratsu
double	64 bit (8 byte)	$-1.7 \times 10^{+308}$ 15 digitu esanguratsu	$+1.7 \times 10^{+308}$ 15 digitu esanguratsu
boolean	8 bit (1 byte)	`true/false`	`true/false`
char	16 bit (2 byte)	0	65536

3.2. taula. Oinarrizko datu motak.

Javan, oinarrizko zortzi datu mota erabil dezakete programatzaileek: `byte`, `short`, `int`, `long`, `float`, `double`, `boolean` eta `char`. **3.2. taula**n datu mota bakoitzaren ezaugarriak jasotzen dira. Aldagai bat erazagutzeko honako formatu honi jarraituko diogu:

datu_mota aldagaia [, aldagaia];

Adibidea: `int balioBat, balioBi = 69;`

Lerro horretan aldagai bi erazagutzen dira eta bigarren aldagaia gainera hasieratu egiten dugu balio bat txertatuz.

Aldagai bat `String` motakoa bada eta eragiketa matematikoak burutzeko `int` motakoa bihurtu nahi badugu, ezin dugu honela egin:

```
int balioInt ;
String balioString = "50";
......
balioInt = balioString;
```

Alegia, datu-bihurketa (`String` → `int`) egin behar dugu lehenago; `int` aldagaiak `int` motako aldagai batera erreferentziatu behar baitu. Kode zati hori konpilatzen saiatzen bagara, honako errore hau emango digu konpiladoreak:

```
xxx.java:xx : incompatible types
found: java.lan.String
required : int
        balioInt =  balioString;
                    ^
```

Konpiladoreak `String` objektu bat aurkitu du lengoaiaren (`java.lang.String`) klase baten motakoa eta programak objektu horren erreferentzia esleitu nahi dio `int` motako oinarrizko datu mota bati, baina ez dio uzten. Zergatik gertatzen da hori? Darabilgun mota (`int`) ez da objektu bat horrek dakarren guztiarekin eta ezin

dugu esleipen hori horren *gordinki* gauzatu. Arazoa konpontzeko, lengoaiak klase batzuk (`Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `Boolean` eta `Character`) eskaintzen ditu jatorrizko datu motak kapsulatzeko eta objektu bihurtzeko. Objektua eraikiz gero datu-bihurketa gauza dezakegu eta esleipena eguneratuz programak funtzionatuko du:

```
balioInt = Integer.parseInt(balioString);
```

Esleipenaren eskuineko aldeak adierazten digu `Integer`[15] klasea erabiliko dugula bihurketa gauzatzeko eta klase horren `parseInt()` metodoari dei eginez `String` baten balioa eskuratzen dugula `int` formatuan. Bihurketa ondo burutzeko, `String`-ak zenbaki hamartarrak izan behar ditu eta zenbaki negatibo bat nahi izanez gero edo behar izanez gero, lehen karakterea minus zeinua izan daiteke.

Nondik	Nora	Metodoa	Adibidea
int	Integer	Eraikitzailea[16]	`Integer number = new Integer(5)`
double	Double	Eraikitzailea	`Double number = new Double(5.6)`
int	String	`toString` `valueOf`	`String string=Integer.toString(5)` `String string=String.valueOf(5)`
double	String	`toString` `valueOf`	`String string=Double.toString(5.6)` `String string=String.valueOf(5.6)`
Integer	int	`intValue`	`int i = zenbaki.intValue()`
Double	double	`doubleValue`	`double x = zenbaki.doubleValue()`
Integer	String	`toString`	`String string = zenbaki.toString()`
Double	String	`toString`	`String string = zenbaki.toString();`
String	Integer	Eraikitzailea `valueOf`	`Integer number = new Integer("5")` `Integer number=Integer.valueOf("5")`
String	Double	Eraikitzailea `valueOf`	`Double number = new Double ("5.6");` `Double number=Double.valueOf ("5.6")`
String	int	`parseInt`	`int number = Integer.parseInt("5");`
String	double	`parseDouble`	`double number=Integer.parseDouble("5.6");`

3.3. taula. Datu-bihurketak.

3.3. taulan bihurketa arruntenak, eta aldi berean, praktikan gehien ematen direnak jasotzen dira. Konturatu, batzuetan, bihurketa bat gauzatzeko era bat baino gehiago dagoela.

3.3. SARRERA/IRTEEREN KUDEAKETA

Jakina da programa bat konpilatu ondoren exekutatzen hasten denean datuak sartu, eguneratu eta ezabatu egiten dituela. Datu horiek memorian dauden bitartean eta

15. `Integer` klasea `Javaren java.lang.Number` klase abstraktutik eratortzen da. Klase abstraktuak geroxeago aztertzen dira.

16. Metodo eraikitzaileak geroxeago aurkezten dira.

programa exekutatzen ari den artean, programaren kontrolpean egongo dira, memoria nagusia behin-behineko euskarri gisa erabilita. Programa bukatzen denean, memoria askatuko du beste programa batzuek erabil dezaten eta gure datu guztiak desagertu egingo dira. Datu batzuk galtzea ez zaigu asko ardura baina beste batzuk gordetzea interesatzen zaigu, gero, irakurri eta prozesatzeko programa berean edo beste batean.

Hori dela eta, programazio-lengoaiek iraunkortasun-mekanismoak izaten dituzte datuak euskarrietan gorde eta irakurtzeko eta halaber, programa exekutatzen ari denean erabiltzailearekin harremanetan jartzeko, datuak teklatutik sartzeko edo, pantailan ikuskatzeko.

Javan, datu-fluxu (`stream`) bidez gauzatzen da sarrera/irteera prozesu hori. Bai sarreran eta bai irteeran, datu-fluxu motak erabiliz datuak egokituz joaten dira euskarritik memoriara eta memoriatik euskarrira. Aplikazioen sarrera/irteera kudeatzeko, `java.io` paketea erabiltzen da hor baitaude behar izango diren klase eta interfazeak.

Has gaitezen sarrera/irteera sinpleena aztertzen. Programa arruntena ere egikaritzen denean, erabiltzailearekin jartzen da harremanetan teklatua eta pantaila erabiliz. Horretarako, teklatutik irakurtzen du informazioa eta zerbait egin ondoren informazio horrekin, pantailan idazten du emaitza. Lengoaia batzuetan *irakur* eta *idatz* agindu soilekin burutzen da prozesu hori guztia, programak ez baitu inongo arazorik sarrera eta irteera estandarrak identifikatzeko. Javan, ostera, gauzak ez dira hain erraz gauzatzen.

Javan, programa batek sarrera/irteera eragiketa bat eratu nahi duenean ez da zuzenean harremanetan jartzen euskarriarekin, datu-fluxu bidez komunikatzen da. Datu-fluxu hauek bitartekari-lana gauzatzen dute euskarriaren eta egikaritzen ari den klasearen artean. Hala independentzia lortzen da euskarriarekiko eta biltegiratzeko orduan datuak erraz eta era malguan kudeatzen dira.

Hortaz, has gaitezen gure lehen sarrera/irteera eratzen:

```
import java.io.*;
class LerroarenIrakurketa {
public static void main( String args[] ) throws IOException {
int karakterea;
int kontagailua = 0;
// irakurri egiten da lerro bukaera(\n) aurkitu arte
while( (karakterea = System.in.read() ) != '\n' )
{
kontagailua++;
System.out.print( (char) karakterea );
}
System.out.println(); // Lerroz aldatu
System.err.println( "Irakurritako byte kopurua:  "+ kontagailua );}}
```

Programa aztertu ondoren erraz ikus daiteke iturburuak egiten duena: irakurri sarrera estandarretik eta "\n" adierazpena detektatzen duenean pantailan idazten du irakurritakoa karakterez karaktere, aldi berean, zenbat aldiz gertatzen den ekintza hau zenbatu egiten da eta amaitzeko, irakurri duen byte kopurua pantailaratu egiten du. Ikusten den legez, sarrera/irteera guztia, `System` klasea erabilita gauzatzen da. Klase hori, `final` motakoa da eta `Object` klase nagusitik eratortzen da, beraz, zuzenean erabil dezakegu instantziarik sortu gabe. Klaseak, hiru atributu (`in`, `out` eta `err`) estatiko (`static`) ditu sarrera/irteera fluxuak kudeatzeko:

- `System.in`-ek `InputStream` objektu bat erreferentziatzen du eta prest dago `InputStream` klasearen edozein metodo egikaritzeko. Guk behar duguna eta erabiliko duguna `read()` izango da. Metodo horrek, sarrera-fluxuko hurrengo byte irakurtzen du.

- `System.out`-ek `PrinStream` objektu bat erreferentziatzen du eta prest dago `PrintStream` klasearen edozein metodo egikaritzeko. Guk behar duguna eta erabiliko duguna `print(char c)` izango da. Metodo gainkargatu horrek karaktere bat idatziko du irteera-fluxuan.

- `System.err`-ek erroreen irteera-fluxu estandarra modelatzen du eta haren portaera `System.out`-enaren parekoa da, `PrintStream` objektu bat erreferentziatzen baitu.

Datu-fluxuen abstrakzioak malgutasuna eskaintzen dio programatzaileari eta geroxeago ikusiko dugun legez, ez dago era bat eta bakarra teklatua modelatzeko. Aitzinean bat ikusi dugu eta laster batean beste bat agertuko zaigu.

Orain arte ikusitakoak sarrera/irteera oinarrizkoak tratatzeko balio digu, eman dezagun beste urrats bat eta sar gaitezen fitxategiak tratatzera. Memorian dagoen informazioa irautea nahi badugu programaren exekuzioa amaitu eta gero, biltegiratu egin beharko dugu. Hori egiteko, lehen bezala, sarrera/irteera fluxuak kudeatu beharko ditugu. Has gaitezen irteerarekin, alegia, informazioa idazten fitxategietan. Irteerako fitxategi bat sortu nahi badugu hiru urrats egin behar dira:

1. `File` motako objektu bat eraiki, sortuko dugun fitxategiaren izena pasatuz argumentu gisa objektu eraikitzaileari.

2. `FileOutputStream` motako beste objektu bat eraiki, goian eraiki dugun `File` motako objektu bat pasatuz argumentu gisa objektu eraikitzaileari. Klase honen bidez irteerako fitxategira bideratzen ditugu datuak. Klaseak informazio-fluxua datu-unitate txikietan (bytetan) modelatzen du baina ez da gauza unitate handiak (karaktere-kateak eta karaktere-lerroak) kudeatzeko beren osotasunean. Horretarako, beste objektu mota bat eraiki behar da.

3. Bukatzeko, `PrintStream` motako objektu bat eraikiko dugu `FileOutputStream` motako objektu bat pasatuz argumentu gisa objektu eraikitzaileari. Azken urrats honek funtzionalitate egokia itsasten dio irteerako fluxuari (`FileOutputStream`-i) irteerako eragiketak (`print`, `println`, `write`, `close`,...) egiteko.

Laburbildu dezagun prozesu guztia adibide batekin:

```
import java.io.*;
class IdaztenFitxategi {
public static void main(String arg[]) throws Exception {
File fitxategiF= new File("fitxategi.irteera");
FileOutputStream FitxategiStream =
               new FileOutputStream(fitxategiF);
PrintStream fitxategi=
      new PrintStream(FitxategiStream);
fitxategi.println("Irteera fitxategian idazten");}}
```

Adibideak *Irteera fitxategian idazten* `String`-a idatziko luke `fitxategi.irteera` fitxategian. Programak fitxategi berri bat sortu eta idatzi egiten du bertan. Lehendik izen bereko fitxategi bat bazegoen, programa egikaritu eta gero, informazioa ordezkatu egiten da eduki berriarekin.

Horratx, irteera-fitxategiak sortzeko modua. Sarrerako fitxategi bat irakurtzeko antzeko bideari jarraitzen zaionez laburragoak izango gara azalpenean. Sarrerako fitxategi bat irakurtzeko, lehen irteerako fitxategiekin egin dugun bezala, hiru urrats bete behar dira:

1. `FileInputStream` motako objektu bat eraiki fitxategi bat modelatzen ari bagara, edo `BufferedInputStream` motakoa, teklatua modelatzen ari bagara. Teklatua modelatzeko bide hau aukeratzen badugu, konturatu irakurle, beste ordezko bide bat aurkitu dugula teklatuko informaziora atzitzeko; alegia, teklatuari sarrerako fitxategi-kutsua eman eta handik fitxategi bat izango balitz bezala irakurri.

2. `InputStreamReader` motako objektu bat eraiki, `FileInputStream` motako objektu bat pasatuz argumentu gisa objektu eraikitzaileari.

3. Bukatzeko, `BufferedReader` motako objektu bat eraiki, `InputStreamReader` motako objektu bat pasatuz argumentu gisa objektu eraikitzaileari.

Lehen bezala, laburbildu dezagun esandako guztia adibide batean:

```
import java.io.*;
class FitxategiaKopiatu{
public static void main(String arg[]) throws IOException {
     InputStreamReader isrTeklatua;    // Teklatua modelatzeko
     BufferedReader Teklatua;

  String jatorrizkoFIzena; // Sarrerako fitxategia modelatzeko
     File fIturri;
     FileInputStream   fisIturri;
     InputStreamReader isrIturri;
     BufferedReader rdrIturri;

  String kopiaFIzena; // Irteerako fitxategia modelatzeko
     File fKopia; // sarrerakoaren kopia izango da
     PrintStream psKopia;
     FileOutputStream fosKopia;

     String s;

isrTeklatua = new InputStreamReader(System.in);//Atzitu Teklatua
     Teklatua = new BufferedReader(isrTeklatua);

     // Irakurri izena eta irteerako fitxategiaren izena sortu
System.out.print("Kopiatu nahi dugun fitxategiaren IZENA(luzapen
gabe): ");
     System.out.flush();

     jatorrizkoFIzena = Teklatua.readLine();
     kopiaFIzena = jatorrizkoFIzena.concat(".kopia");

     // Sarrerako fitxategia atzitzeko prestatu datu fluxuak
     fIturri = new File(jatorrizkoFIzena);
     fisIturri = new FileInputStream(fIturri);
     isrIturri = new InputStreamReader(fisIturri);
     rdrIturri = new BufferedReader(isrIturri);

     // Prestatu irteerako fitxategiaren datu fluxuak
     fKopia = new File(kopiaFIzena);
     fosKopia = new FileOutputStream(fKopia);
     psKopia = new PrintStream(fosKopia);

     // Jatorrizko fitxategitik BI lerro irakurtzen ditugu
     //  eta fitxategi berrian kopiatu
         s = rdrIturri.readLine();
         psKopia.println(s);
         s = rdrIturri.readLine();
         psKopia.println(s);  }}
```

Irakurleak orain artekoa menderatu ondoren jauzi erraldoia egin dezake eta bere ordenagailuaz baliatuta Interneten murgildu. Ez ikaratu irakurle, ez duzu beste hogei orri gehiago irakurri beharrik. Gehiena eginda daukagu, makillajea aldatuko dugu eta kitto.

Hasteko, Javari laguntza eskatuko diogu eta `java.net` liburutegia erabiliko dugu. Sarean dauden ordenagailuak atzitzeko balioko digu liburutegi horrek eta hala aplikazio informatikoak beste ordenagailuetako informazioa eskura lezake arazorik gabe.

Demagun Donostiako Informatika Fakultateko web gune nagusia atzitu nahi dugula eta hango HTML informazioa eskuratu gure ordenagailutik. Zer egin beharko dugu?

1. Donostiako Informatika Fakultatearen URLa (Universal Resource Locator) aurkitu: `http://www.sc.ehu.es/siwebso/index.html`
2. URL motako objektu bat eraiki:
   `URL url = new URL(“http://www.sc.ehu.es/siwebso/index.htm”)`
3. URL klasearen `openstream()` metodoa erabilita, urruneko ordenagailuarekin jartzen gara harremanetan, konexio bat irekitzen dugu eta bukatzeko, `InputStream` motako objektu bat eraikitzen dugu.
4. Sortu berri dugun `InputStream` motako objektuarekin `InputStreamReader` motako objektu bat eraikitzen dugu.
5. Bukatzeko, `InputStreamReader` motako objektuarekin `BufferedReader` motako objektu bat eraikitzen dugu eta prest dugu azpiegitura guztia gure web gunea irakurtzeko eta saltseatzen hasteko

Laburbilduz, honako hau da konpilatu beharko genukeen iturburu-kodea:

```
import java.net.*;
import java.io.*;

class WebOrria {
public static void main( String[] args ) {
  String karaktereak;

  try {
    // URL motako objektu bat eraikitzen dugu
    // Hemen jartzen duguna Donostiako Informatika Fakultatekoa da
    URL url = new URL(
    “http://www.sc.ehu.es/siwebso/index.html”);

    BufferedReader htmlOrria =
      new BufferedReader( new InputStreamReader(
        url.openStream() ) );
```

```
        // Irakurri eta pantailaratu edukia lerroz lerro

    while( (karaktereak = htmlOrria.readLine()) != null ) {
        System.out.println( karaktereak );
      }
    } catch( UnknownHostException e ) {
      e.printStackTrace();
      System.out.println(
               "Sarean konektatu egon behar zara !!!!!" );
    } catch( MalformedURLException e ) {
      e.printStackTrace();
       } catch( IOException e ) {
      e.printStackTrace();  }}}
```

Dena ondo joan bada, karaktere andana agertuko da pantailan, HTML kodean idatzita.

3.3.1. StringTokenizer

Objektuetara zuzendutako programa batek, memoria nagusian egikaritzen hasten denean, klase desberdinetako hainbat metodo eraikitzaileri egiten die dei objektuak sortzeko eta haiekin lanean hasteko. Mota bereko objektu asko behar ditugunean, hainbat `new <objektu> (<parametroZerrenda>)` programatu behar ditugu objektu horiek guztiak eraikitzeko. Gehienetan, metodo eraikitzailearen parametro-zerrenda objektuaren atributuen balioak hasieratzeko erabiltzen da eta egokia litzateke balio horiek nonbaitetik irakurriko bagenitu programaren hasieratzea errazteko. Hala, adibidez, fitxategi bat sor dezakegu editore bat erabilita objektuen atributuen hasierako balioekin. Ondoren, programak, metodo eraikitzaileei deitu baino lehen, fitxategi hori irakurriko luke hasierako balioak memorian edukitzeko, eta amaitzeko, metodo eraikitzaileei deituta objektuak sortuko lituzke.

Prozesu hori guztia errazteko, Javak, `StringTokenizer` klasea eskaintzen digu. Aurretik irakurritako `String` baten unitate lexikografikoak eskuratzen ditu klase horrek. Unitate lexikografiko horiek nondik nora hedatzen diren jakiteko, unitateak nahitaez nola zatitzen diren jakin behar du programak. Hori guztia `StringTokenizer` klaseak ebazten du metodo eraikitzailean: metodo eraikitzaileak gutxienez[17] bi parametro behar ditu `StringTokenizer` objektu bat eraikitzeko: lehenengoa, zatitu behar den `String` nagusia, eta bigarrena, zatitzeko erabiliko dugun `String` mugatzailea. Adibidez:

```
StringTokenizer gure Tokena=new StringTokenizer(karaktereKatea, "/");
```

17. Metodo eraikitzailea gainkargatua dago eta bi formatu nagusi erabil dezake programatzaileak:
    - `StringTokenizer(String s, String muga)`, guk erabiliko duguna eta,
    - `StringTokenizer(String s, String muga` , boolean itzuliToken). Aldagai boolearrarekin zatiak (tokenak) itzuli nahi ditugun ala ez adierazten da.

Adibide horretan, `StringTokenizer` objektu bat eraiki dugu eta aldi berean sistemari esan diogu `String` nagusiaren barruan zati (token) desberdinak bereizteko, "`/`" karakterea erabiliko dugula. Gero programan, `nextToken()` metodoa erabilita eta arazo handirik gabe, zati desberdinak eskuratu ahal izango ditugu. Azter dezagun `StringTokenizer`-en portaera adibide konplexu eta praktikoago batean:

```
static void kargatuBezeroak()
      throws FileNotFoundException, IOException {
      BufferedReader sarrera =
        new BufferedReader(new FileReader("bezeroak.txt"));
  // bezeroak.txt TESTU FITXATEGIA DA
  // lerro bakoitzean <bezeroaren> informazioa dator
  // ondoko formatuan: nan/izena/helbidea/telefonoa/kreditua
  // Adibidea:  12341234/Pello Korta/
  //Zugarramurdi 32-4.B/12344321/1200.50
      String lerro;
    public static final String FITXATEGI_AMAIERA=null;
      while ((lerro=sarrera.readLine())!=FITXATEGI_AMAIERA)
      {
        StringTokenizer st=new StringTokenizer(lerro,"/");
        Bezeroa bezeroa= new Bezeroa(
                st.nextToken(),
                st.nextToken(),
              st.nextToken(),
              st.nextToken(),
                Float.valueOf(st.nextToken()).floatValue());
          bezeroa.ikuskatu();
          Bezeroak.gehituBezeroa(bezeroa);  }}
```

Adibidean lau eragiketa nagusi gauzatzen dira:

1. Aurretik editore batekin sortu dugun fitxategi (`bezeroak.txt`) bat lerroz lerro irakurtzen dugu eta irakurketa bakoitzaren emaitza, `lerro` `String`-ean uzten dugu.
2. Irakurritako lerroa `StringTokenizer` klasearen metodo eraikitzaileak hartzen du eta `StringTokenizer` motako objektu bat sortzen du. Objektu hori, `StringTokenizer` motako aldagai batekin erreferentziatzen dugu.
3. `StringTokenizer` motako aldagaia erreferentziatzen ari den objektuari `nextToken()`[18] metodoa egikaritzeko eskatzen diogu. Horrela, irakurri dugun lerroaren unitate lexikografikoak eskuratzen ditugu.
4. Unitate lexikografikoak erabilita, `Bezeroa` motako objektu bat eraikitzen dugu. Konturatu metodo eraikitzaileak `String` motakoa ez den parametro bat behar duenean, datu-bihurketa bat gauzatu behar duela, parametroaren balioa metodo eraikitzaileari pasatu baino lehen.

18. `nextToken ( )` metodoak `String` bat itzultzen du hurrengo unitate lexikografikoarekin.

3.4. DOKUMENTAZIOAREN ANTOLAKETA: JAVADOC

Informatikari gutxi ezagutzen ditut dokumentazio-zaleak direnak. Informatikako ikasketa gehientsuenetan dokumentazioaren auzia azkar eta azaletik tratatzen da. Badirudi garrantzia, programan, kodean dagoela eta beste guztia horren menpe ipini behar dela. Azken urteetan softwarearen konplexutasuna dezente handitu da eta egun zaila egiten da aplikazio informatiko bat aztertzea kode soila erabilita bakarrik. Aplikazio horrek ehunka metodo eta milaka aldagai eduki ditzake eta, agian, garatu zuen programatzailea beste saltsetan ari da lanean, edo larriagoa dena, enpresa utzi du eta gelditzen zaigun bere arrasto bakarra ordenagailu alboan zituen maskotak eta milaka kode-lerro huts dira. Maskotak, paperontzira bota edo programatzaileen artean bana ditzakegu; kodea, ostera, ezin paperontzira bota eta ezin bana dezakegu. Hortik bizi gara eta hori da gure diru-iturria.

Irakurleak pentsa dezake aipatutako egoera gehiegizkoa dela eta hori gutxitan gertatzen dela; horrela pentsatzen badu, heldu berria da informatikara eta oraindik *dokumentitis* gaitzaren atzaparretan erortzeke egon daiteke.

Beraz, gure softwarea aldameneko lankideak edo erabiltzaileak ulertzea nahi badugu, gero aldaketak edo hedapenak gauzatzeko, ez diezaiegun eman kode hutsa eta esan, “hor konpon, Mari Anton”.

Aplikazio bat ondo dokumentatzeak garrantzia du eta handia gainera, ulertzeko zuk egindakoa, beste pertsona batzuen eskuetatik pasatu behar bada, ondoren, erabili, aldatu, konpondu edota hedatzeko aukera eduki dezan.

Esanak esan, ondo dokumentatutako aplikazio informatiko batek hiru dokumentazio mota izango ditu:

1. Aplikazioaren betekizunen analisia eta diseinua aztertzen dituena. Hemen aplikazioa egituratzeko erabiliko dugun paradigma (objektuetara zuzendutako metodoa/metodo egituratua) aukeratu ondoren, normala izaten da metodologia bat (UML, Metrica, SSADM, Yourdon...) aukeratzea eta metodologia horren notazioa erabilita aplikazio informatikoa dokumentatzea. Egun, tresna aurreratuak (Rational Rose, Visio 2002, Eclipse...) erabilita gauzatzen da lan hau. Aplikazioa proiektu bat bezala antolatzen da eta beraren arlo guztiak, kontrolpean eta interkonektatuta mantentzen dira une oro etengabe.

2. Softwarea kodetu eta probatu eta gero, instalatu egiten da erabiltzailearen egoitzan. Hanka hutsik utziko genuke erabiltzailea instalazioa bukatu ondoren ez bagenio erabiltzailearen eskuliburua emango. Eskuliburuan, aplikazioaren ohiko gertaera-fluxua nola egikarituko den azalduko da eta komeni izaten da, ezohiko gertaeren aurrean, zer egin behar den adieraztea eta ez maiz gertatzen den legez, telefono mugikorraren zenbakia luzatzea,

arazoa konpontzeko. Zenbat eta dei gutxiago, orduan eta produktu eta eskuliburu hobea egin dugun seinale.

3. Eta zer esan kodeari buruz. Lehenengo dokumentazio motarekin, aplikazioaren bizi-zikloaren analisi eta diseinua dokumentatzen ditugu; bigarren motarekin, ostera, instalazioa eta mantentzea, tartean, inplementazioa gelditzen zaigu, kodea, alegia.

Aipatutako dokumentazio motek asko laguntzen dute softwarea kontzeptualki eraikitzen ari garenean, hau da, analisian eta diseinuan. Gauzak korapilatu egiten dira softwarea instalatu eta tarte bat igaro ondoren, bezeroak dei egiten digunean softwarea hedatzeko betekizun berriak itsatsi nahi dizkiolako programari.

Analisi eta diseinuaren dokumentazioak asko lagunduko du funtzionalitate berria kokatzen eta aztertzen, ez baikara hutsetik abiatzen, baina, azkenean, programatzaileak kodea hartu beharko du eta kode horretan funtzionalitate berria txertatu. Une horretan programatzaileak asko eskertuko luke iturburu-kodean bertan laguntza aurkituko balu. Testu-iruzkinak jarrita gauzatzen da laguntza hori eta askotan denbora faltagatik edo nagitasunagatik gelditzen da baztertuta.

Javaren sortzaileek tresna bat txertatu zuten JDK (Java Development Kit) estandarrean testu-iruzkinen lan hau errazteko eta, batez ere, programatzailearentzat erabilgarri bihurtzeko: javadoc.

Tresnak aplikazioaren iturburuak aztertzen ditu eta HTML orriak sortzen ditu. Orri horietan, aplikazioaren klaseen informazio zehatza islatzen da era egituratuan eta oso lagungarria da programatzailearentzat. Prozesu guztia ondo burutzeko, programatzaileak ondo kokatu behar ditu iruzkinak iturburuan. Iruzkin batek ondo interpretatua izateko klasearen, interfazearen, metodoaren edo atributuaren aurre-aurrean joan behar du eta honela adierazita:

```
/**
*   Hau iruzkin bat da eta
*   klaseen aurrean ipintzen dut
*/
```

Horrekin, iruzkin arrunt bat sortu dugu programaren iturburuan eta oraingoz ez diogu gauza berririk itsatsi aurretik genekienari eta egiten genuenari. Dokumentazioa egituratu nahi badugu eta javadoc tresnari etekina atera, dokumentazioa asko aberastuko da iruzkinak etiketez (**3.4. taula**) atontzen baditugu. Etiketa horiek txertatuta, javadoc-ek iruzkinen analisi sintaktikoa gauzatzen du eta sortzen duen informazioa HTML orri eran egituratuta uzten du irakurgarriagoa izan dadin programatzailearentzat. **3.4. taula**n ikusten den legez, etiketa guztiek @ ikurra daramate javadoc-ek lehen karaktere gisa ezagutu ditzan analisian.

Etiketa	Helburua
`@author` testua	• Klasearen edo interfazearen egilea adierazteko
`@param` parametroIzena deskripzioa	• Metodo baten parametroen informazioa argitzeko
`@return` deskripzioa	• Metodo batek itzultzen duen balioa deskribatzeko
`@deprecated` testua	• Klaseak, interfazeak, aldagaiak eta metodoak markatzeko erabiltzen da eta ohartarazten digu zaharkitua gelditu dela edo ordezkatua izan dela elementu hori
`@see` erreferentzia	• Erabiltzen ari garen metodotik beste klase edo metodo batera erreferentzia bat adierazteko
`@link` paketea.klasea#partaidea testu	• Estekadura bat adierazten du dokumentazioaren beste elementu batera
`@throws` / `@exception` klaseIzena deskripzioa	• Salbuespen-egoerak tratatzeko
`@version` testua	• JDKren bertsioari buruzko iruzkina

3.4. taula. Javadoc tresnaren etiketak.

Programatzailea HTML orriak sortzen aditua bada, HTML kodea gordin-gordin txerta dezake iruzkinean eta, ondoren, javadoc-ek iturburua aztertzen duenean, kontuan hartuko du eta era egokian interpretatu. Adibidez, demagun kodean programatzailearen helbide elektronikoa txertatu nahi dugula arazoren bat gertatzen denean erabiltzaileak laguntza eske nora idatzi jakin dezan. Horretarako, honako hau idatziko genuke:

```
/**
*  Arazoak aurkitzen badituzu
*  programa egikaritzerakoan, erabili helbide hau:
*   <ADDRESS> zubizarreta@ehu.dif.es </ADDRESS>
*/
```

Adibide gisa, hona hemen, Javaren teknologiak berak, nola antolatzen duen `Enumeration` interfazearen dokumentazioa:

```
package java.util;

/**
 * An object that implements the Enumeration interface generates a
 * series of elements, one at a time. Successive calls to the
 * <code>nextElement</code> method return successive elements of the
 * series.
 * <p>
 * For example, to print all elements of a vector <i>v</i>:
 * <blockquote><pre>
 *     for (Enumeration e = v.elements() ; e.hasMoreElements() ;) {
 *         System.out.println(e.nextElement());<br>
 *     }
 * </pre></blockquote>
 * <p>
 * Methods are provided to enumerate through the elements of a
 * vector, the keys of a hashtable, and the values in a hashtable.
 * Enumerations are also used to specify the input streams to a
 * <code>SequenceInputStream</code>.
 * <p>
 * NOTE: The functionality of this interface is duplicated by the
 * Iterator
 * interface.  In addition, Iterator adds an optional remove
*operation, and
 * has shorter method names.  New implementations should consider
* using
 * Iterator in preference to Enumeration.
 *
 * @see     java.util.Iterator
 * @see     java.io.SequenceInputStream
 * @see     java.util.Enumeration#nextElement()
 * @see     java.util.Hashtable
 * @see     java.util.Hashtable#elements()
 * @see     java.util.Hashtable#keys()
 * @see     java.util.Vector
 * @see     java.util.Vector#elements()
 * @author  Lee Boynton
 * @version 1.16, 06/29/98
 * @since   JDK1.0
 */

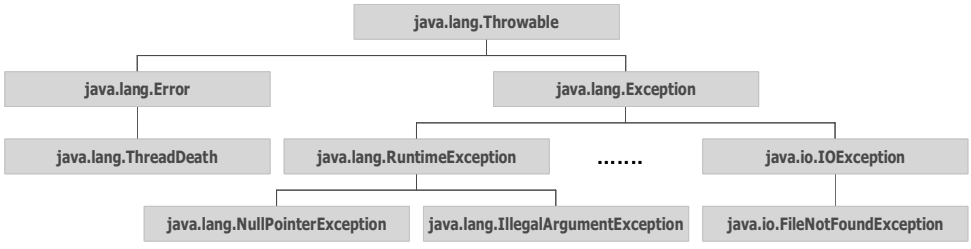

public interface Enumeration {
    /**
     * Tests if this enumeration contains more elements.
     *
  * @return <code>true</code> if and only if this enumeration object
     *           contains at least one more element to provide;
     *         <code>false</code> otherwise.
     */
```

```
    boolean hasMoreElements();
    /**
  * Returns the next element of this enumeration if this enumeration
     * object has at least one more element to provide.
     * @return     the next element of this enumeration.
     * @exception  NoSuchElementException  if no more elements exist.
     */
    Object nextElement();}
```

3.5. SALBUESPENAK SALBUESPEN

Aplikazio informatiko baten exekuzio-fluxu arruntenak baldintzapeko makina bat adarkatze izaten ditu, aldagaiak balio desberdinak hartu eta hartu aritzen dira eta eragiketa aritmetikoak hamaika egoeratan egikaritzen dira. Gehienetan exekuzio-fluxua arazorik gabe egikarituko da eta erabiltzaileak aplikazioa lasai eta ziur itxiko du. Baina noizbait, eta bat-batean, ezer berezirik egin gabe, aplikazioak porrot egingo du eta erabiltzailea harrituta gelditu da bere buruari galdetuz, zer egin ote duen oker egun horretan. Haserre, aplikazioaren analistari deitzen dio eta horrek diseinatzaileari. Diseinatzailea ez dago alukerietarako eta programatzaileari deitzen dio arazoa konpon dezan. Programatzaileak goiz osoa ematen du programa arakatzen eta azkenean, hara non aurkitzen duen bilatzen ari zen buruhaustearen iturburua: programak sarrera-irteerako eragiketa bat gauzatu nahi zuen baina arazoak zituen fitxategiarekin eta ezin izan du eragiketa burutu. Une horretan, salbuespen-egoera bat sortu da eta aplikazioak ez zekienez zer egin ez nora joan, exekuzioa bertan behera utzi du.

Salbuespen-egoera gehienek aipatutako bi ezaugarri horiek izaten dituzte: ez dira sarri agertzen baina agertzen direnean letalak izaten dira ez badira baliabideak ipintzen arazoari aurre egiteko. Horrenbestez, programatzaileak, tentu handiz programatu behar ditu aplikazioaren lurralde arriskutsu horiek eta hemen, inon baino gehiago, Paretoren Legearen[19] aholkuak kontuan hartu behar ditu.



3.5. taula. Salbuespenak kudeatzeko klaseak.

19. Iturburu-kodearen % 20k aplikazioaren konplexutasunaren % 80 tratatzen du.

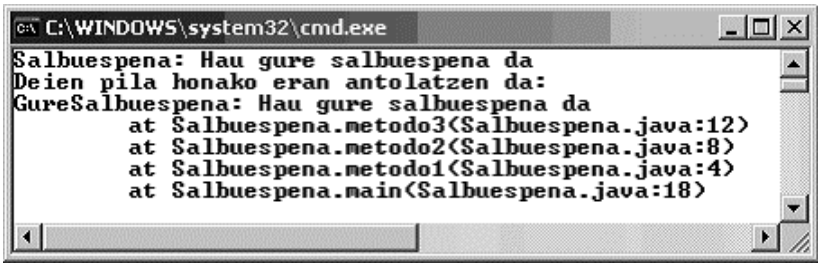
Javaren teknologiak, salbuespen-egoera horiek tratatzeko, azpiegitura modular bat eskaintzen dio programatzaileari ezohiko egoera horiek harrapatu eta kudeatzeko. **3.5 . irudia**n ikusten den legez, `Throwable` klaseak azpiegitura nagusia prestatzen du aparteko egoera horiei aurre egiteko. Laburbilduz, aplikazioaren exekuzioan, bi ezohiko egoera mota gerta daitezke: erroreak eta salbuespenak.

Erroreak, Javaren makina birtualak tratatzen ditu eta oso egoera larrietan altxatzen dira. Egikaritzen ari den aplikazio informatikoak ezer gutxi egin dezake egoera horietan.

Salbuespenak, eskuragarriagoak dira aplikazioarentzat eta programatzaileak baliabideak presta ditzake exekuzioaren fluxua kontrolatzeko eta programa bertan behera ez uzteko.

Programatzaileak, metodo bat programatzen ari dela, sumatzen badu iturburu-kodeak salbuespen-egoera bat altxa dezakeela, `try` bloke baten barruan sartuko du iturburu-kode zati hori, adierazteko berezi samarra dela, eta exekuzio-fluxuak arazoren bat aurkitzen badu, `catch` blokean sartuko da salbuespen-egoera tratatzeko. Metodoa egikaritzerakoan arazorik egon ez bada, `try` blokea egikarituko da eta exekuzio-fluxuak aurrera jarraituko du hurrengo aginduarekin. Batzuetan, ordea, bai egoera batean bai bestean egon, iturburu-kode zati bat egikaritu nahi da beti; hori adierazteko aplikazioari, `finally` bloke berri bat antolatu behar du programatzaileak eta bloke horretan sartu egikaritu nahi den iturburu-kodea. Esanak esan, metodo batek salbuespen bat kudeatu nahi duenean honako egitura hau izango du:

```
public void metodo( )
      {  ..................
         ..................
    try {
    ..................
    // Aztertu nahi den iturburu-kodea
         ..............}
    catch (  ) {
    ..................
    // Salbuespena harrapatzen da eta tratatu
         ..............}
    finally {
    ..................
    // Beti egikaritu nahi den zatia
         ..............}
               }
```



3.6. taula. Salbuespenen hierarkia.

Behin salbuespen bat altxatuta, sistemak metodo bat bilatzen du salbuespen hori tratatzeko. Deien pila aztertuta eskuratuko da metodoa. Lehenengo eta behin, salbuespena altxatu duen metodoarekin hasiko da eta hor ez bada tratatzen, pilako hurrengo metodoarekin jarraituko du. Tamalez, pila amaitzen bada eta salbuespena tratatu barik gelditzen bada, aplikazioaren exekuzio-fluxua gelditu egingo da. Hona hemen (**3.6. irudia**) adibide honetan nola garatzen den aipatutako prozesua:

```
class Salbuespena
{  public static void metodo1() throws GureSalbuespena
   {
     metodo2();
  }
public static void metodo2() throws GureSalbuespena
   {
     metodo3();
  }
public static void metodo3() throws GureSalbuespena
   {
     throw new GureSalbuespena();
  }
public static void main(String args[])
{
   try
    {
      metodo1();
    }
    catch (GureSalbuespena e)
    {System.out.println("Salbuespena: "+e.getMessage());
     System.out.println("Deien pila honako eran antolatzen da:");
     e.printStackTrace();
    }}}
class GureSalbuespena extends Exception
{
   GureSalbuespena()
   {
     super("Hau gure salbuespena da");
   }}
```

Azaldu dugun guztia zehatz-mehatz finkatuta uzteko aplikazio informatiko txiki bat garatuko dugu. Bertan, `Exception` klasetik zuzenean eratortzen diren salbuespenak eta aplikazioarentzat propio diseinatu direnak tratatuko dira. Aplikazioak artikulu baten erosketa gauzatu nahi du, eta horretarako, aplikazioa egikaritzerakoan zenbaki bat idatzi behar dugu aplikazioak irakur dezan. Artikuluaren salneurria izango da zenbaki hori eta `Artikulu` motako objektu bat eraikiz gero, balio hori itsatsiko zaio bere salneurriari. Hala gauzak, erosle bat eraikiko da `Bezeroa` motakoa eta erosketa bat gauzatuko da `erosketaBurutu( )` metodoa egikarituta:

```
class Aplikazioa {
   public static void main(String[] args) {

       int Salneurri = 0;
       if (args.length > 0) {
            try {
                Salneurri =
                    Integer.parseInt(args[0]);
           }
            catch(NumberFormatException e) {
                System.out.println(
      " Zenbaki Oso bat sartu behar duzu lehen argumentu bezala.");
                return;
            }
        }
        else {
            System.out.println(
              "Salneurria sartu behar duzu.");
            return;
        }

        // Eraiki artikulu bat eta
        // itsatsi salneurria.
        Artikulu artikulu = new Artikulu();
        artikulu.setSalneurri(Salneurri);
        // Eraiki eroslea eta burutu erosketa.
        Bezeroa erosle = new Bezeroa();
        Eroski.erosketaBurutu(erosle, artikulu);
    }
}
```

Aplikazioaren iturburu-kodean `try` bloke bat programatu da erabiltzailearen sarrera kontrolatzeko eta arazoak aurkitzen badira, `catch` bloke bat prestatu da salbuespena oratzeko. Zati honek azalpen txiki bat merezi du. Iturburu-kodeak arazoak aurkitzen baditu `try` blokea egikaritzeko, `catch` blokeak `NumberFormatException` motako salbuespenak oratuko ditu eta galdera, berehala datorkigu ahora: zergatik, hain zuzen, mota horretako salbuespena?

Erantzuna aurkitzeko, `Integer` klasearen `parseInt( )` metodo estatikora joan behar dugu:

```
public static int parseInt(String s) throws NumberFormatException
```

`parseInt( )` metodo estatikoak `NumberFormatException` motako salbuespen bat altxa dezakeela erazagutzen du eta horregatik gure metodo nagusiak aukera bi zituen, edo salbuespena erazagutu, edo salbuespena oratu eta tratatu. Bistan da bigarrena aukeratu duela.

Iturburu-kodearekin jarraitzen badugu, `Aplikazioa` klase nagusiaren `main` metodo nagusia, `erosketaBurutu( )` metodoa egikarituta amaitzen da:

```
  class Eroski {

 public static void erosketaBurutu(Bezeroa erosle,
     Artikulu artikulu) {

     try {
         erosle.erosiArtikulu(artikulu);
         System.out.println("Erosketa burutu da.");
     }
     catch (LarBehetikException e) {
         System.out.println("Salneurria lar behetik dabil.");

     }
     catch (LarGoitikException e) {
         System.out.println("Salneurria lar goitik dabil.");
}}}
```

Espero genuen bezala `Eroski` klaseak `erosketaBurutu( )` metodo estatikoa definitzen du `try` bloke bat kontrolatuta. Blokea ondo egikaritzen bada, ez da salbuespenik altxatuko eta aplikazioa `println( )` metodoa egikarituta amaituko da. Arazorik sortzekotan, `Bezero` motako objektuaren `erosiArtikulu( )` metodoarekin sortuko da:

```
  class Bezeroa {

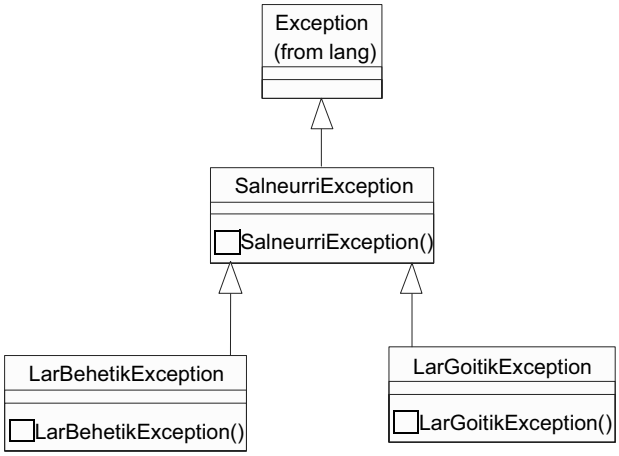
  private static final int larBehetik = 50;
  private static final int larGoitik = 80;

 public void erosiArtikulu(Artikulu artikulu) throws
     LarBehetikException, LarGoitikException {

     int Salneurri = artikulu.getSalneurri();
     if (Salneurri <= larBehetik) {
         throw new LarBehetikException();
     }
     else if (Salneurri >= larGoitik) {
```

```
                throw new LarGoitikException();
            }
            //...
        }
        //...
}
```

Hasieran sartu dugun artikuluaren salneurria 50 baino txikiagoa bada edo 80 baino handiagoa, salbuespenak altxatuko dira eta hori adierazteko `throw` hitz erreserbatua erabiliko da. Iturburu-kodeak adierazten duen legez, gure aplikazioan (**3.7. irudia**) `LarBehetikException` edo `LarGoitikException` motako objektu bat instantziatzen da.



3.7. irudia. Aplikazioaren salbuespen klaseak.

3.7. irudiak adierazten du nola egituratu diren aplikazioaren salbuespenak. Kontu egin salbuespen bat baino gehiago tratatu nahi denean kontuan izan behar dela nola jartzen diren `catch` blokeak. Gure kasuan, adibidez, ez du inongo zentzurik honako hau jartzeak:

```
catch(SalneurriException e)  {........}
catch(LarBehetikException e)  {........}
```

Lehenengo `catch` blokeak `LarBehetikException`, `LarGoitikException` eta `SalneurriException` salbuespenak harrapatzen ditu eta horrelako egoeretan, hau da, salbuespen desberdinen artean klase-azpiklase erlazioa azaltzen denean, egokiago izaten da klase orokorrak bukaera aldera eramatea.

Kontu egin, orain arte salbuespen bat oratzen zenean, `println( )` metodoa egikaritzen zela soilik irteera estandarretik mezu xume bat ateratzeko. Testuinguru horretan, salbuespenen metodo eraikitzailea parametrorik gabe egikaritzen zen.

Egoera biziki aldatzen da salbuespena tratatu behar duen klasea Klase Diagramaren beste klase baten erara antolatzen badugu eta ez bakarrik `Exception` klasearen eratorpen ñimiño bat bezala. Demagun gure adibidean, bai `LarBehetikException` salbuespena harrapatzen dugunean baita `LarGoitikException` salbuespena oratzen dugunean ere, birfindu egin nahi dugula trataera, salneurrien balioa arakatuta. Hori guztia, `catch` blokearen barruan programa daiteke inongo arazorik gabe:

```
class Eroski {

     public static void erosketaBurutu(
         Bezeroa erosle, Artikulu artikulu) {

         try {
             erosle.erosiArtikulu(artikulu);
             System.out.println("Erosketa burutu da.");
         }
         catch (LarBehetikException e) {

             int Salneurri = e.getSalneurri();
             System.out.println("Artikuluaren salneurria : "
                 + Salneurri + " da.");

             if (Salneurri > 1 && Salneurri <= 10) {
                 System.out.println(
                     "Artikuluaren salneurria behea jota dabil.");
                }
             else if (Salneurri > 10
                 && Salneurri <= 50) {
   System.out.println("Artikuluaren salneurria behetik dabil.");
                             }
             else if (Salneurri <= 0) {
   System.out.println("Salneurri hau iraingarria da !!!!!.");
                             }
         }
         catch (LarGoitikException e) {

             int Salneurri = e.getSalneurri();
             System.out.println("Artikuluaren salneurria : "
                 + Salneurri + "   da.");
             if (Salneurri >= 80 && Salneurri < 100) {
                 System.out.println("Artikulua garestia da.");
                             }
             else if (Salneurri >= 100
                 && Salneurri < 200) {
                System.out.println("Artikulua garestiegia da.");
                             }
             else if (Salneurri >= 200
   && Salneurri < 1000) {
```

```
System.out.println(
          "Erotuta zaude!!!!.Bill Gates naizela uste duzu!!!");
                              }
      }}}
```

`SalneurriException`, `LarBehetikException` eta `LarGoitikException` salbuespen klaseak honako era honetan egokitu behar izan ditugu funtzionalitate berri hori inplementatzeko:

```
abstract class SalneurriException
     extends Exception {

     private int Salneurri;
     public SalneurriException(int Salneurri) {
         this.Salneurri = Salneurri;
     }
     public int getSalneurri() {
               return Salneurri;
          }}

class LarBehetikException
           extends SalneurriException {

           public LarBehetikException(int Salneurri) {
                super(Salneurri);
           }}
class LarGoitikException
            extends SalneurriException {

            public LarGoitikException(int Salneurri) {
                super(Salneurri);
           } }
```

Hala, nahikoa zaigu salbuespena altxatzen denean, `throw` hitz erreserbatua erabilita, salbuespen-instantzia egokia sortzea. Instantzia berria eraiki baino lehen, metodo eraikitzaileari hasieran sartu dugun salneurria pasatuko diogu. Gero, `catch` bloke bakoitzak, arazorik gabe, `getSalneurri( )` metodoa erabilita eskuratuko du.

3.6. KLASEAK ETA OBJEKTUAK

Javak iturburu-kodean dagoen informazioa klaseetan egituratzen du. Konpiladoreak klasea nondik nora hedatzen den jakin dezan, `{` eta `}` ikurren artean kokatzen da klasea eta `class` hitz erreserbatua jartzen da klasearen izenaren aurretik bera identifikatzeko.

Lengoaiak bereak egiten ditu C eta C++ lengoaien hainbat ezaugarri eta hori dela eta, Java, C/C++ *eguneratu* eta *leundu* bat dela aldarrikatzen da. Has gaitezen lehenengo eta behin, antzekotasunarekin: Javan, C++-ean legez, programa bat egikaritzen hasteko, salbuespenak salbuespen[20], `main` metodoa izan behar du programatua eta klase batean egon behar du txertatua. Metodo horri ez dio inork deitzen programatik, `java` komandoa erabiltzen dugunean interpreteak fitxategi egokian bilatzen du eta egikaritzen hasten da, hau da, `>java gurePrograma` idazten badugu, `gurePrograma.class` fitxategia bilatu eta egikaritzen hasiko da hango `main` metodora joanda.

Lehen urratsik gabeko biderik ez dagoenez, abia gaitezen gure lehen programarekin, milaka aldiz errepikatua eta ehundaka argitaratua:

```
class kaixoPatxi {
      public static void main (String [] args) {
      System.out.println(“Kaixo Patxi !!!!! “);
                                                }
               }
```

Patxi esnatzeko darabilgun programa hori ezaguna egingo zaie C++ ezagutzen dutenei eta ondoren datorren azalpena gainetik pasa dezakete nahi izanez gero. Besteontzat, hona hemen azalpena: lehen lerroan, zintzo-zintzo, aurretik esandakoa bete dugu, hasieran, `class` hitz erreserbatua, ondoren klasearen izena, `kaixoPatxi`. Gogoratu izen horrek eta `.class` fitxategiaren izenak[21] bat etorri behar dutela programa ondo egikaritzeko eta bukatzeko, `{` karakterea, klasearen hasiera adierazteko. Ondoren datorrena klasearen edukia da, gure kasuan metodo soil bat, hori bai, era kriptiko xamarrean idatzita.

Metodo horren izena `main` da eta dagoeneko ezaguna da guretzat. Parentesi artean `String [] args` adierazpena agertzen da eta honakoa adierazten du: metodoa deitzen dugunean egikaritze-unean, parametro formalak eta errealak bat egingo dute eta `String` motako array bat izango dugu prest metodoan erabiltzeko. Gurea kasuan, `java kaixoPatxi bat bi`, idatzi izan bagenu array-ak balio bi izango lituzke: `args[ 0]` -k “`bat`” balioko luke eta `args[ 1]` -k “`bi`”. Geroxeago, luze arituko gara parametroei buruz, oraingoz, utz dezagun gaia bere horretan.

Aztertzen ari garen metodoaren ezkerraldean beste hiru hitz esoteriko xamarrak jarri ditugu: `public, static` eta `void`. Lehenengoak oztopoak ekiditeko balioko digu ikusgaitasuna adierazten baitu. `public` hitz erreserbatua jarri dugu lehendabizi, `main` metodoa atzigarri bihurtzeko. Hala, Javaren interpreteak ez du inongo arazorik izango bertara atzitzeko. Ondoren, `static`, hau

20. Geroago ikusiko dugun bezala `applet` motako programa batek ez du `main` metodorik behar eta inongo arazorik gabe egikaritu daiteke bera gabe.

21. Kontuz irakurle !!!! Javan hizki larriak eta xeheak desberdindu egiten dira:
`patxieuskaltel` ≠ `PatxiEuskaltel`

ere hitz erreserbatua. Hitz horrekin programak interpreteari esaten dio ez duela inongo `kaixoPatxi` klaseren instantziarik bilatu behar; `main` metodoa klasearen metodoa da eta ez bere instantziarena, horregatik ez du `kaixoPatxi` klasearen instantziarik behar egikaritzeko. Bukatzeko, `void` hitz erreserbatuak adierazten du metodoa prozedura bat dela eta ez duela baliorik itzuliko egikaritzerakoan.

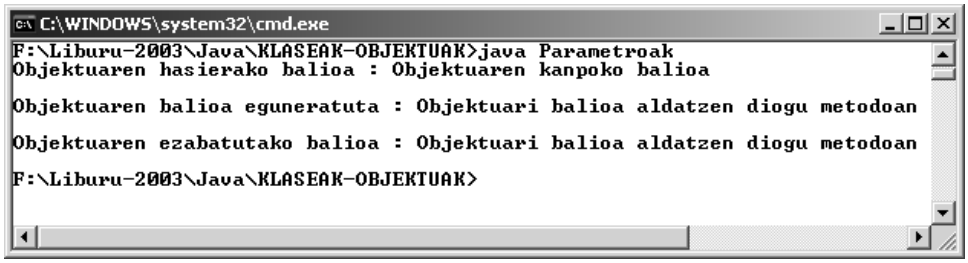
Programaren bigarren lerro korapilatsua utzita, ondoren, metodoaren gorputza azaltzea gelditzen zaigu: programak irteera-fluxu estandarra erabilita "`Kaixo Patxi!!!`" `String`-a idazten du fluxuak modelatzen duen irteera estandarrean eta kitto, horratx, gure programa guztia!!!

Orain arte esandakoarekin badakigu programa bat egikaritzeko, `.class` motako fitxategi bat behar dugula, arazoa da nola sortu horrelako fitxategi bat. Joan gaitezen urratsez urrats: demagun Internet bidez Sun Microsystems enpresak duen web gunera (www.sun.com) joan garela eta Javaren bertsio bat eskuratu ondoren `C:\jdk` direktorioan instalatu dugula. Hori egin ondoren eta programa-editore bat erabiliz `kaixoPatxi.java` fitxategia idazten dugu. Urduri eta presaka, gauza berrien xarmak sortzen duen lilurak itsututa, gure lehen programa konpilatzen saiatzen gara. Hori egiteko, `javac kaixoPatxi.java` komandoa idazten dugu eta... kale!!! ordenagailuak: `Bad command or file name`, mezua ematen digu. Gure ordenagailu azkarrak ez du konpiladorea aurkitu; gure kasuan, `C:\jdk\bin` bidea gehitzen diogu sistemaren PATH aldagaiari eta aurrera!!

Berriro idazten dugu komandoa eta... bapo!!! sortu zaigu nahi genuen `kaixoPatxi.class` fitxategia. Lehen baino urduriago, idazten dugu `java kaixoPatxi`, komandoa eta... kaka zaharra!!! berriro mezu berri bat: `Exception in thread "main" java.lang.NoClassDefFoundError: kaixoPatxi`. Nola ez duela klasea bilatzen? Hor zegon/k eta direktorioan!! Lasai irakurle, hemen dator suhiltzailea, idatzi honako hau: `set CLASSPATH=.` eta ondoren komandoa. Orain bai!!! hor dago harro-harro, aurrean, gure lehen programak sortua: `Kaixo Patxi!!!!!` mezua, kostata baina azkenean atera da.

Aperitiboa dastatu ondoren sar gaitezen menu nagusiarekin. Hasieran esan dugu Javan informazioa klaseetan egituratzen dela eta adibide txiki bat aztertu dugu klase bat zuena eta haren barruan metodo soil bat. Horrelako klaseak ez dira maiz agertzen programa estandarretan, normalean klasearen gorputzak atributuak eta metodoak izango ditu. Atributuek klasearen barne-egitura islatzeko balio dute eta objektuaren egoera definitzen dute une oro; metodoek, ostera, klasearen portaera deskribatzen dute eragiketen bidez. Objektuen ikuspegitik, metodo bat mezu bat da, zerbitzu bat eskatuz objektu batek beste bati bidaltzen diona. Objektu igorleak beharrizan bat du egikaritzen jarraitzeko eta badaki nork gauza dezakeen lan hori, mezua bidaltzen dio eta objektu jasotzaileak burutu egiten du lana, mezuari dagokion metodoa egikarituz.

Javan metodo bati dei egiten diogunean egikaritzeko eta metodo horrek argumentuak baditu, balio bidez pasatzen dizkiegu argumentu errealak argumentu formalei. Argumentuak jatorrizko datu motakoak badira, datu motaren kopia bat pasatzen dugu eta metodoak balioa aldatzen badio datu motari, aldaketak ez du inongo eraginik metodoa bukatu ondoren, aldagaiak metodora sartu aurreko balioa hartzen baitu berriro. Metodoaren barnean aldagaiak lokalak dira eta metodoa egikaritu ondoren desagertu egiten dira. Argumentua erreferentzia motakoa denean, erreferentziatzen ari garen objektuan aldaketak gauza daitezke eta aldaketa horiek iraun egingo dute metodotik programara itzultzen garenean. Hona hemen adibide txiki bat:



3.8. irudia. Metodoak eta argumentuak aztertzen.

```
class Parametroak {

      public static void main(String[] args) {
            Parametroak param = new Parametroak();
            param.frogatuParam();
            }

      public void frogatuParam() {
      Objektua objektua = new Objektua ();
      objektua.balio1 = "Objektuaren kanpoko balioa" ;
      System.out.println("Objektuaren hasierako balioa : " +
objektua.balio1);
      System.out.println();
      balioAldatu(objektua);
      System.out.println("Objektuaren balioa eguneratuta : " +
objektua.balio1);
      System.out.println();
      balioaEzabatu(objektua);
      System.out.println("Objektuaren ezabatutako balioa : " +
objektua.balio1);

}

      public void Parametroak() {  }

      public static void balioAldatu(Objektua objek1)
            {
```

```
      objek1.balio1 = "Objektuari balioa aldatzen diogu
metodoan" ;
            }
      public static void balioaEzabatu(Objektua objek1)
            {
            objek1 = null; // Ez du eraginik
            }}

class Objektua {
      String balio1, balio2;

public Objektua() { }
}
```

Objektuen ikuspegitik, metodoei buruz esan dugunak berdin-berdin balio du atributu bat atzitzen dugunean ere.

Gauzak korapilatu egiten dira, klasearen metodoek eta atributuek ikusgaitasun-adierazlez (`public, protected, private` eta `package`) atontzen ditugunean. Orduan, klasearen ezaugarria (metodoa edota atributua) kapsulatuta gelditzen da eta erabat (`public`) atzigarria izan daiteke beste objektuentzat edo erabat (`private`) izkutua utz dezakegu aplikazioaren beste objektu guztientzat. **3.9. taula**k adierazle bakoitzaren atzipen maila islatzen du:

Atzipen maila	klasea	azpiklasea	paketea	besteak
`private`	X			
`protected`	X	X	X	
`public`	X	X	X	X
`package`	X		X	

3.9. taula. Klasearen metodoen eta atributuen ikusgaitasuna.

Taulan ikus daitekeen legez lehen zutabeak (*klasea*) mezu argi bat bidaltzen digu: klase bateko metodo guztiek atzipena dute klasearen ezaugarri guztietara kontuan hartu gabe atzipen maila, alegia, klaseari berdin-berdin zaio `private, public, protected` edo `package` ipintzea, berdin-berdin atzitzen baita ezaugarrietara.

Bigarren zutabean (*azpiklasea*), ostera, gauzak nahasi egiten dira. Zutabe horretan klase-azpiklase erlazioak nola egituratu daitezkeen atzipen mailan adierazten da. Espero genuen bezala, azpiklaseko objektu batek ez du inongo arazorik klasearen `public` ezaugarrietara atzitzeko. Ezaugarria, `protected`–ekin erazagututa dagoenean klasean, azpiklaseko objektu batek atzitu dezake, klasea eta azpiklasea pakete (`package`) berean badaude.

Hirugarren zutabeak eragina izan dezan, klaseak pakete (`package`) berean kokatuko ditugu. Hori egiten badugu klase guztien `protected` eta `public` ezaugarriak atzigarri bihurtzen ditugu.

Eta zer gertatzen da ez klase berean, ez pakete berean eta ez azpiklase berean dauden metodoekin, alegia, laugarren zutabeko metodoekin? Begi-bistakoa da erantzuna: metodo horiek nahiko isolatuta daude objektuen munduan, klasearen `public` ezaugarrietara atzitu baitezakete bakarrik.

Esan dugun guztiak gauzak argitzen lagundu izan badizu irakurle, atzipenen zati hau amaitzeko, ez zaizu zaila egingo bigarren eta azken taula hau ulertzea:

Ikusgaitasuna klaseen arteko erlazioetan	adierazlerik gabe	private	protected	public
Klase berean	Bai	Bai	Bai	Bai
Pakete berean azpiklasea	Bai	Ez	Bai	Bai
Paketean baina ez azpiklasea	Bai	Ez	Bai	Bai
Pakete desberdinean baina azpiklasea	Ez	Ez	Bai	Bai
Pakete desberdinean baina ez azpiklasea	Ez	Ez	Ez	Bai

3.10. taula. Ikusgaitasuna klaseen arteko erlazioetan.

Dagoeneko `private` eta `public` adierazleen esanahia guztiz argituta dago eta erabat logikoa da, alegia, ezin duzu inola ere atzitu (`private`) eta atzitu (`public`) zabal hartuko zaitugu. Bestela, aipatu behar da, deigarria egiten dela adierazlerik gabeko ezaugarri bat, pakete berean dagoenean, atzitu daitekeela inongo arazorik gabe eta gainera, ez dela azpiklasea izan behar. Halaber bitxia, `protected` adierazlearen portaera, nahiz eta azpiklasea pakete desberdinean egon, ez da arazorik sortuko, ezaugarria, `protected` adierazlearekin atontzen badugu.

Bukatzeko, `Erabiltzailea` klase nagusia programatu dugu Javaren atributuen eta metodoen ikusgaitasuna aztertzeko. Klase Diagramaren diseinuan, gainera, beste hiru klase erabiliko ditugu: `Aita, Semea eta Osaba`. Hiru klaseak honela daude definituta:

```
package paketeBat;
class Aita {
      int aitaHutsik;
      protected int aitaProtected;
      static protected int aitaStatic;
      public Aita() {  aitaHutsik = 1;
                       aitaProtected = 10;
                       aitaStatic = 20;}
      }

package paketeBat;
public class Semea extends Aita {
      public int semeaPublic;
             int  semeaHutsik;
        protected int semeaProtected;
        private int semeaPrivate;
```

```
  public Semea () {
       semeaHutsik  =1;
       aitaHutsik = 10;

  }}

package paketeBi;
public class Osaba {
       public int osabaPublic;
              int  osabaHutsik;
         protected int osabaProtected;
         private int osabaPrivate;

  public Osaba () {
       osabaHutsik  =1;
       osabaProtected = 10;

  }}
```

Eta hona hemen, `Erabiltzailea` klasea:

```
1: package paketeBi;
2: import paketeBat.*;
3: //import paketeBi.*;
4: //
5: class Erabiltzailea extends Semea {
6: private int erabiltzailea;
7: //
8:    Aita objektuAita = new Aita();
9:    Semea objektuSemea = new Semea();
10: Osaba objektuOsaba = new Osaba();
11: //
12: Erabiltzailea () {
13: objektuSemea.aitaProtected = 2;
14: objektuSemea.aitaHutsik = 2;
15: objektuSemea.aitaStatic = 2;
16: objektuSemea.semeaPublic    =2;
17: objektuSemea.semeaHutsik   =2;
18: objektuSemea.semeaProtected =2;
19: objektuSemea.semeaPrivate   =2;
20:
21: objektuOsaba.osabaPublic = 10;
22: objektuOsaba.osabaHutsik = 10;
23: objektuOsaba.osabaProtected = 10;
24: //
25: }
26: public static void main(String [] args){
27: Erabiltzailea objektuErabiltzailea = new Erabiltzailea();
28: objektuErabiltzailea.semeaProtected = 2;
```

```
29: System.out.println("KAIXO PATXI !!!!!"); }
30: }
31: //
32: //
33: class Beste  {
34: private int beste;
35: Beste () {beste=1;}
36: }
```

Ikusgaitasun motak aztertzeko `Erabiltzailea` klasearen iturburuaren lehen hiru lerroak, aldiko, aldatu egingo ditugu eta bi eratara jarriko ditugu:

```
a.-    import paketeBat.*;
       import paketeBi.*;

b.-    package paketeBi;
       import paketeBat.*;
```

Iturburuak honela daude biltegiratuta azpidirektorioetan:

```
........>Softwarea>Erabiltzailea.java
........>Softwarea>paketeBat>Aita.java
........>Softwarea>paketeBat>Semea.java
........>Softwarea>paketeBi>Osaba.java
```

Hala gauzak, `Softwarea` azpidirektorioa azpidirektorio-sustrai gisa erabilita, `Erabiltzailea.java` fitxategia konpilatzen dugu:

```
........>Softwarea>javac Erabiltzailea.java
```

1. Zer gertatuko litzateke (a) zatia jarriko bagenu goiburu gisa?
2. Zer gertatuko litzateke (b) zatia jarriko bagenu goiburu gisa?

Konturatu orain (b) egoeran dagoela `Erabiltzailea.java` fitxategia eta, beraz, bigarren azpigaldera erantzuteko eran dago jarrita.

Konpiladoreak iturburu-kodea lehenengo azpigalderan adierazten den egoeran aurkitzen badu, honako errore-txosten hau argitaratzen du irteera estandarretik:

```
Error: (8) class paketeBat.Aita is not public in package paketeBat;
cannot be accessed from outside package.
Error: (8) class paketeBat.Aita is not public in package paketeBat;
cannot be accessed from outside package.
Error: (13) variable aitaProtected not found in class paketeBat.Semea.
Error: (14) variable aitaHutsik is not public in class paketeBat.Aita;
cannot be accessed from outside package.
Error: (15) variable aitaStatic not found in class paketeBat.Semea.
Error: (17) variable semeaHutsik is not public in class paketeBat.Semea;
cannot be accessed from outside package.
```

```
Error: (18) variable semeaProtected has protected access in class
paketeBat.Semea.
Error: (19) variable semeaPrivate has private access in class
paketeBat.Semea.
Error: (22) variable osabaHutsik is not public in class
paketeBi.Osaba; cannot be accessed from outside package.
Error: (23) variable osabaProtected has protected access in class
paketeBi.Osaba.
```

Konpiladoreak, 8. lerroan, erroreak atzeman ditu, `Aita` klaseak ez du inongo adierazlerik eta `public` beharko, beste pakete batean baitago. Txostenaren azpiko lerroetan, arazoak sortzen dira `Semea` motako objektuarekin. Hasieran, objektuak, `Aita` motako objektuaren ezaugarrietara atzitu nahi du baina ezin du, ez dituelako aurkitzen edota `public` adierazlea falta zaielako. Bitxia da gainera, `Semea` motako objektuaren ezaugarriekin ere, konpiladoreak arazoak izatea. Berez, ematen du `Erabiltzailea` klasea `Semea` klasetik eratortzen bada, ez duela arazorik egon behar. Jakina, horrela dira gauzak, objektua `Erabiltzailea` motakoa denean, baina kontu egin, gure kasuan, objektua `Semea` motakoa dela eta beste pakete batean dagoela. Antzeko gauza gertatzen da `Osaba` motako objektuarekin `public` adierazlea ez duen ezaugarri batera atzitu nahi duenean.

Bigarren azpigalderan, errore-txostenaren hasierako egoera ez da batere aldatzen baina bai, ordea, `Osaba` motako objektuarena Hemen, bi klaseak pakete berean daudenez, atzipenen murriztapenak leundu egiten dira:

```
Error: (8) class paketeBat.Aita is not public in package paketeBat;
cannot be accessed from outside package.
Error: (8) class paketeBat.Aita is not public in package paketeBat;
cannot be accessed from outside package.
Error: (13) variable aitaProtected not found in class paketeBat.Semea.
Error: (14) variable aitaHutsik is not public in class
paketeBat.Aita; cannot be accessed from outside package.
Error: (15) variable aitaStatic not found in class paketeBat.Semea.
Error: (17) variable semeaHutsik is not public in class
paketeBat.Semea; cannot be accessed from outside package.
Error: (18) variable semeaProtected has protected access in class
paketeBat.Semea.
Error: (19) variable semeaPrivate has private access in class
paketeBat.Semea.
```

Azken paragrafoetan objektuen instantzia-atributuak eta metodoak nola atzitu daitezkeen azaltzen saiatu gara baina ez dugu ezer esan klase eta objektuei buruz, alegia, zer den bat eta zer bestea, zein den beren arteko desberdintasuna eta objektuez ari garenean nola sortzen diren beren instantziak eta nola egikaritzen diren metodoak. Uste dut garaia dela galdera horiei ere erantzuten hasteko.

Klase bat datu mota abstraktu bat da gehienetan inplementazio partziala duena. Oro har, software-testu estatiko bat da, ez zaiona exekuzio bati lotu behar. Objektua, ostera, klase baten eratorpena (klasearen instantzia) da, dinamikoki sortua eta programaren exekuzio-garaian ordenagailuaren memorian kokatzen dena.

Azaldu dugun bezala, klasearen instantzia, programaren exekuzioan, memorian kokatzen den objektu bat da. Jarraian objektu hori nola sortzen den azalduko dugu. Objektuetara zuzendutako programazio-lengoaia gehienetan bezala, Javan, klasearen instantzia bat sortzeko metodo eraikitzaileak erabiliko ditugu. Metodo horiek klasearen izen berbera eduki behar dute eta, normalean, instantziaren atributuak hasieratzeko balio ohi dute. Ipin dezagun adibide bat:

```
Nireklasea nirea = new Nireklasea(bat);
Nireklasea nireaEre = new Nireklasea(bat, bi);
```

Lerro bakoitzari hebreerazko irakurketa egingo diogu eta eskuinetik ezkerrera azalduko dugu: `new` hitz erreserbatuarekin `Nireklasea` klasearen instantzia bat sortzen dugu eta erazagututa dugun `nirea` aldagaiari, haren erreferentzia esleitzen diogu. Konturatu instantzia sortzerakoan parametro bat pasatu diogula eraikitzaileari, orduan zer gertatzen da bigarren lerroarekin bi parametro pasatzen baitizkiogu eraikitzaileari? Bada, gauzak ondo egin baditugu, metodo eraikitzailea gainkargatu egin dugula eta parametro kopuru eta moten arabera, sistemak metodo eraikitzaile egokiena aukeratuko duela. Javak berak ere, erruz erabiltzen du teknika hau bere paketeetan objektuak hasieratzeko. Adibide gisa hona hemen, `java.awt.Dialog` klasearen metodo eraikitzaileak:

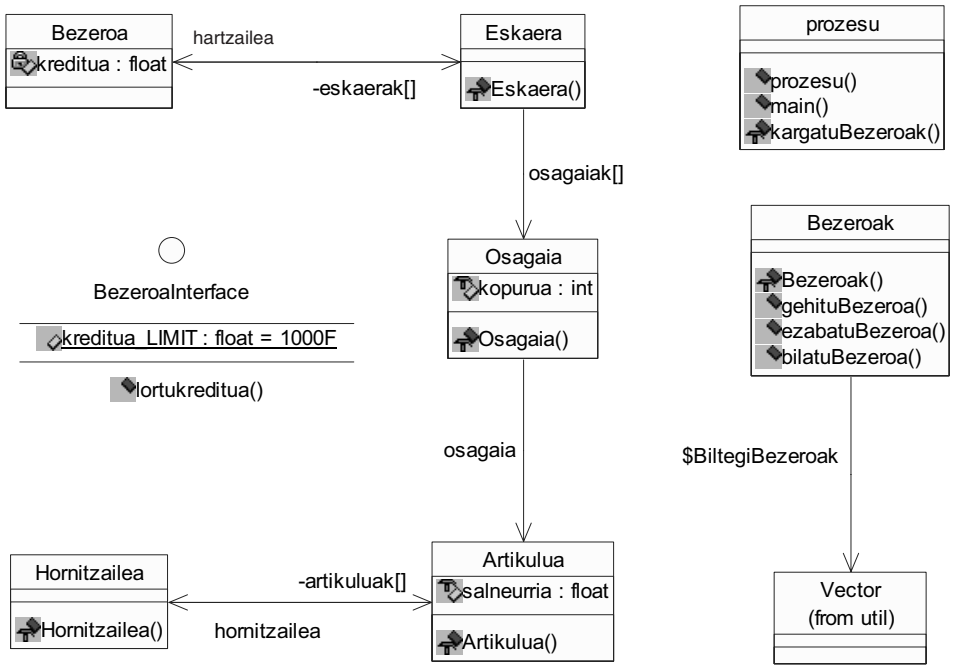
```
Dialog(Dialog owner)
Dialog(Dialog owner, String title)
Dialog(Dialog owner, String title, boolean modal)
Dialog(Frame owner)
Dialog(Frame owner, boolean modal)
Dialog(Frame owner, String title)
Dialog(Frame owner, String title, boolean modal)
```

Argibide batzuk metodo eraikitzaileei buruz: metodo horiek ez dira heredatzen, ezin dute baliorik itzuli, metodo eraikitzaile batek klase bereko beste metodo eraikitzaile bati dei diezaioke `this` hitz erreserbatua erabilita eta klase-azpiklase erlazioetan azpiklase motako instantzia bat sortzeak ez du esan nahi automatikoki haren klasearen metodo eraikitzaileak ere egikaritzen direla. Hori egiteko `super` hitz erreserbatua erabili behar dugu eta parentesi artean parametro-zerrenda adierazi:

```
<azpiklaseIzena> (<parametroZerrenda>) {
       super(<parametroZerrenda>);
 // azpiklasearen instantzia sortzeko aginduak
                                    }
```

Bukatzeko, aplikazio informatiko bakarrean laburbilduko dugu aurreko orrietan azaldu duguna. Aplikazio informatikoak **3.11. irudia**n agertzen den Klase Diagrama izango du eta bezeroen eskaerak kudeatzen saiatuko da. Jakina, egoera erreal batean, eskaerak datu-base batean biltegiratuko lirateke baina gu ez gara saltsa horietan sartuko oraingoz eta `Vector` erako behin-behineko biltegi motak

erabiliko ditugu bakarrik. Hala, `Bezeroak` klasean, `Vector` motako hainbat `BiltegiXXX` objektu eraikiko dugu informazioa biltegiratzeko.



3.11. irudia. Aplikazioaren Klase Diagrama.

Bestela, aplikazioak ez du egundoko lanik gauzatuko: eskaerak sortu eta biltegiratu eta eskaerak eskuratzeko aukera eman, `lortuEskaera()` eta `lortuEskaerak()` metodo gainkargatuak erabilita:

```
import java.util.*;

import java.util.Vector;          // java.util.* eta java.io.*
ordez
import java.util.StringTokenizer; //     Ikusteko non dauden
import java.io.BufferedReader;    // erabili beharreko klaseak
import java.io.FileReader;
import java.io.IOException;
import java.io.FileNotFoundException;
import java.io.InputStreamReader;


public class prozesu {

        public static final String FITXATEGI_AMAIERA=null;
        public static void main(String[] args)
```

```
            throws FileNotFoundException, IOException
             {

                  kargatuBezeroak();
                  kargatuHornitzaileak();
                  kargatuArtikuluak();


      // Osagai generikoak sortzen ditugu

      for (int i=50;i<56;i++)  {
            Artikulua artikulua=
              (Artikulua)Bezeroak.BiltegiArtikuluak.elementAt(i-
49);
            Osagaia osagaia = new Osagaia(artikulua,i);
            osagaia.ikuskatu();
               Bezeroak.gehituOsagaia(osagaia);
            }

      //Eskaerak sortzen

            Osagaia[ ]  osagaiak = new Osagaia[ 20] ;

            for (int k=0;k<5;k++)  {
            Bezeroa
bezeroa=(Bezeroa)Bezeroak.BiltegiBezeroak.elementAt(k);
            for (int i=0;i<6;i++)  {
                 int j = (int)Math.round(Math.random()*5);
          Osagaia osagaia=
(Osagaia)Bezeroak.BiltegiOsagaiak.elementAt(j);
                osagaiak[ i]  = osagaia;
                              }
            Eskaera eskaera = new Eskaera(bezeroa,osagaiak);
            Bezeroak.gehituEskaera(eskaera);
            }


      // Ikuskatu bezero guztiak
          for (int i=0;i<Bezeroak.BiltegiBezeroak.size();i++)

((Bezeroa)Bezeroak.BiltegiBezeroak.elementAt(i)).ikuskatu();

      // Bezero bat bilatu bere izena erabilita
            BufferedReader f=new BufferedReader(
                    new InputStreamReader(System.in));
            System.out.print("Emaidazu bezeroaren NAN-a: ");
            System.out.flush();
            String nan=f.readLine();
            Bezeroa bezeroa = Bezeroak.bilatuBezeroa(nan);
          if (bezeroa ==null)
```

```
          System.out.println("Bezero Ez aurkitua: NAN = "+ nan);
                   else bezeroa.ikuskatu();

        }


       static void kargatuBezeroak()
             throws FileNotFoundException, IOException {
             BufferedReader sarrera =
               new BufferedReader(new FileReader("bezeroak.txt"));
             // bezeroak.txt TESTU FITXATEGIA DA
             // lerro bakoitzean <bezeroaren> informazioa dator
        // ondoko formatuan:  nan/izena/helbidea/telefonoa/kreditua
       // Adibidea:  12341234/Pello Korta/Zugarramurdi 32-
       //4.B/12344321/1200.50
             String lerro;


             while ((lerro=sarrera.readLine())!=FITXATEGI_AMAIERA)
             {
                StringTokenizer st=new StringTokenizer(lerro,"/");
                Bezeroa bezeroa= new Bezeroa(
                        st.nextToken(),
                        st.nextToken(),
                     st.nextToken(),
                     st.nextToken(),
                        Float.valueOf(st.nextToken()).floatValue());
              bezeroa.ikuskatu();
                  Bezeroak.gehituBezeroa(bezeroa);
             }  }

static void kargatuHornitzaileak()
             throws FileNotFoundException, IOException {
             BufferedReader sarrera =
              new BufferedReader(new FileReader("hornitzaileak.txt"));
             // hornitzaileakak.txt TESTU FITXATEGIA DA
             // lerro bakoitzean <hornitzailearen> informazioa dator
  // ondoko formatuan: nan/izena/helbidea/telefonoa/fidagarritasuna
             // Adibidea:  12341234/Pello Korta/Zugarramurdi 32-
             //4.B/12344321/txikia
             String lerro;
             while ((lerro=sarrera.readLine())!=FITXATEGI_AMAIERA)
             {
                StringTokenizer st=new StringTokenizer(lerro,"/");
                Hornitzailea hornitzailea= new Hornitzailea(
                        st.nextToken(),
                        st.nextToken(),
                     st.nextToken(),
                     st.nextToken(),
                        st.nextToken());
```

```
             hornitzailea.ikuskatu();
                Bezeroak.gehituHornitzailea(hornitzailea);
            }  }

static void kargatuArtikuluak()
            throws FileNotFoundException, IOException {
            BufferedReader sarrera =
              new BufferedReader(new FileReader("artikuluak.txt"));
            // bezeroak.txt TESTU FITXATEGIA DA
            // lerro bakoitzean <artikuluaren> informazioa dator
            // ondoko formatuan:    izena/deskripzioa/salneurria
            // Adibidea:  LAM001/Txoria txori lamina.Zumeta-Artze/12
            String lerro;
          Hornitzailea
hornitzailea=(Hornitzailea)(Bezeroak.BiltegiHornitzaileak.elementAt(2)
);
            while ((lerro=sarrera.readLine())!=FITXATEGI_AMAIERA)
            {
               StringTokenizer st=new StringTokenizer(lerro,"/");
               Artikulua artikulua= new Artikulua(
                       st.nextToken(),
                       st.nextToken(),
                    Float.valueOf(st.nextToken()).floatValue(),
                    hornitzailea);

             artikulua.ikuskatu();
                Bezeroak.gehituArtikulua(artikulua);
            }}  }
// prozesu amaiera

   //=============================================================

class Bezeroak {

        static Vector BiltegiBezeroak = new Vector();
        static Vector BiltegiHornitzaileak = new Vector();
        static Vector BiltegiArtikuluak = new Vector();
        static Vector BiltegiOsagaiak = new Vector();
        static Vector BiltegiEskaerak = new Vector();


      public static void gehituHornitzailea(Hornitzailea hornitzailea)
{
             BiltegiHornitzaileak.addElement(hornitzailea);
                   }

      public static void gehituArtikulua(Artikulua artikulua) {
             BiltegiArtikuluak.addElement(artikulua);
                   }
```

```
      public static void gehituOsagaia(Osagaia osagaia) {
            BiltegiOsagaiak.addElement(osagaia);
                  }

      public static void gehituEskaera(Eskaera eskaera) {
            BiltegiEskaerak.addElement(eskaera);
                  }

  // ------------------------------------------------------

      /** gehitu Bezero berria BiltegiBezeroak Bektorean */
      public static void gehituBezeroa(Bezeroa bezeroa) {
            BiltegiBezeroak.addElement(bezeroa);
      }

      /** ezabatu Bezeroa BiltegiBezeroak Bektoretik */
      public static Bezeroa ezabatuBezeroa(String izena) {
            Bezeroa bezeroa;
            Enumeration total = BiltegiBezeroak.elements();
            while (total.hasMoreElements()) {
                  bezeroa = (Bezeroa)total.nextElement();
                  if (bezeroa.lortuizena().equals(izena)) {
                        BiltegiBezeroak.removeElement(bezeroa);
                        return bezeroa;
                  }
            }
            return null;
      }



      public static Bezeroa bilatuBezeroa(String nan) {
          for (int i=0;i<BiltegiBezeroak.size();i++)

if(((Bezeroa)(BiltegiBezeroak.elementAt(i))).lortunan().equals(n
an))
                return (Bezeroa)(BiltegiBezeroak.elementAt(i));
            return null;
         }}

   //============================================================

class Bezeroa {

      private String nan;
      private String Izena;
      private String Helbidea;
      private String Telefonoa;
      private float kreditua;
```

```
      private Eskaera[ ] eskaerak = new Eskaera[ 20];

      Bezeroa(String nanB, String izenaB,String helbiB,String
telB, float kredB)
            {
              this.nan = nanB;
              this.Izena = izenaB;
              this.Helbidea = helbiB;
              this.Telefonoa = telB;
              this.kreditua = kredB; }


      /** Eskuratu bezeroaren nan-a */
      public String lortunan() {
            return nan;
      }

      /** Gorde bezeroaren nan-a */

      public void gordenan(String nan) {
            this.nan = nan;
      }

      /** Eskuratu bezeroaren izena */
      public String lortuizena() {
            return Izena;
      }

      /** Gorde bezeroaren izena */

      public void gordeizena(String izena) {
            this.Izena = izena;
      }

      /** Eskuratu bezeroaren Helbidea */
      public String lortuHelbidea() {
            return Helbidea;
      }

      /** Gorde bezeroaren Helbidea  */
      public void gordeHelbidea(String Helbidea) {
            this.Helbidea = Helbidea;
      }

      /** Eskuratu bezeroaren Telefono zenbakia */
      public String lortuTelefonoa() {
            return Telefonoa;
      }
```

```
      /** Gorde bezeroaren Telefonoa */
      public void gordeTelefonoa(String Telefonoa) {
            this.Telefonoa = Telefonoa;
      }

      /** Eskuratu bezeroaren kreditua */
      public float lortukreditua() {
            return kreditua;
      }

      /** Gorde bezeroaren kreditua */
      public void gordekreditua(float kreditua) {
            this.kreditua = kreditua;
      }

      /** Ikuskatu bezeroaren informazio pertsonala*/
      public void ikuskatu()
          { System.out.println(" <<BEZEROA: NAN = "+this.nan+ "\n");
        System.out.println("          IZENA = "+this.Izena+ "\n");
  System.out.println("          HELBIDEA= "+this.Helbidea+ "\n");
  System.out.println("          TELEFONOA= "+this.Telefonoa+ "\n");
  System.out.println("          KREDITUA= "+this.kreditua+ "\n");
       }


      /** Hemendik beherakoa Bezeroaren Eskaerak tratatzeko zatia */
      /** ================================================== */

      /** Eskuratu bezeroaren Eskaera bat. Ez badu bilatzen
      *   null itzultzen du.
      */
      public Eskaera lortuEskaera() {
            if (eskaerak == null) return null;
            Eskaera temp_Eskaera;
            for (int i = 1; i < eskaerak.length; i++) {
                  if (eskaerak[ i]  != null) {
                        temp_Eskaera = eskaerak[ i] ;
                        eskaerak[ i]  = null;
                        return temp_Eskaera;
                  }}
            return null;  }

/** Eskuratu bezeroaren Eskaera bat izena emanda. Ez badu bilatzen
      *   null itzultzen du.
      */
      public Eskaera lortuEskaera(String izena) {
            if (eskaerak == null) return null;
            Eskaera temp_Eskaera;
            for (int i = 1; i < eskaerak.length; i++) {
                  if (eskaerak[ i]  != null &&
```

```
eskaerak[ i] .hartzaile.Izena.equals(izena)) {
                        temp_Eskaera = eskaerak[ i] ;
                        eskaerak[ i]  = null;
                        return temp_Eskaera;
                  }}
            return null;  }

      /** Eskuratu bezeroaren Eskaera bat hartzailearen arabera.
      *     Ez badu bilatzen null itzultzen du.
      */
      public Eskaera lortuEskaera(Bezeroa hartzaile) {
            if (eskaerak == null) return null;
            Eskaera temp_Eskaera;
            for (int i = 1; i < eskaerak.length; i++) {
   if (eskaerak[ i]  != null && eskaerak[ i] .hartzaile == hartzaile) {
                        temp_Eskaera = eskaerak[ i] ;
                        eskaerak[ i]  = null;
                        return temp_Eskaera;
                  } }
            return null;
      }
/** Eskuratu bezeroaren Eskaera bat hartzailearen izenaren arabera.
      *     Ez badu bilatzen null itzultzen du.
      */
      public Eskaera[ ]  lortuEskaerak(String izena) {
            if (eskaerak == null) return null;
            Eskaera[ ]  temp_eskaerak = new Eskaera[ eskaerak.length] ;
            int j = 0;
            for (int i = 0; i < eskaerak.length; i++)
                  if (eskaerak[ i]  != null &&
eskaerak[ i] .hartzaile.Izena.equals(izena))
                        temp_eskaerak[ j++]  = eskaerak[ i] ;
            Eskaera[ ]  eskaerak = new Eskaera[ j] ;
            for (int i = 0; i < j; i++)
                  eskaerak[ i]  = temp_eskaerak[ i] ;
            return eskaerak;
      }

      /** Eskuratu hartzaile batek jasoko dituen Eskaerak. */
      public Eskaera[ ]  lortuEskaerak(Bezeroa hartzaile) {
            if (eskaerak == null) return null;
            Eskaera[ ]  temp_eskaerak = new Eskaera[ eskaerak.length] ;
            int j = 0;
            for (int i = 0; i < eskaerak.length; i++)
      if (eskaerak[ i]  != null && eskaerak[ i] .hartzaile == hartzaile)
                        temp_eskaerak[ j++]  = eskaerak[ i] ;
            Eskaera[ ]  eskaerak = new Eskaera[ j] ;
            for (int i = 0; i < j; i++)
                  eskaerak[ i]  = temp_eskaerak[ i] ;
            return eskaerak;
```

```
      }
      /** Gehitu Eskaera berri bat Bezeroari bere
      *   kreditua 1000 baino handiagoa bada. Eragiketa ondo joan bada
      *   true itzultzen du bestela false.
      */
      public boolean gehituEskaera(Eskaera eskaera) {
            if (this.lortukreditua() <
BezeroaInterface.kreditua_LIMIT) return false;
            if (eskaerak == null) eskaerak = new Eskaera[ 20] ;
            for (int i = 1; i < eskaerak.length; i++) {
                  if (eskaerak[ i] == null) {
                        eskaerak[ i] = eskaera;
                        return true;
                  } }
            return false;  }   }

class Eskaera {
      Bezeroa hartzaile;
      Osagaia[ ] osagaiak;

      Eskaera(Bezeroa harE, Osagaia[ ] osaE) {
            this.hartzaile= harE;
    System.out.println("ESKAERA SORTUTA DAGO !!!!!!!! \n \n \n");
            this.osagaiak = osaE;   }}

class Osagaia {
      Artikulua osagaia;
      int kopurua;

      Osagaia(Artikulua artikuluaO, int kopO) {
            this.osagaia= artikuluaO;
            this.kopurua= kopO;
            }

public void ikuskatu()
          {
         System.out.println("<<ESKATUTAKO OSAGAIA:     \n");
          osagaia.ikuskatu();
        System.out.println("Eskatutako KOPURUA = "+this.kopurua+"\n");
        }  }

class Artikulua {
      String izena;
      String deskripzioa;
      float salneurria;
      Hornitzailea hornitzailea;

Artikulua(String izenA, String deskA,float salnA,Hornitzailea horA)
            {
            this.izena = izenA;
```

```
            this.deskripzioa = deskA;
            this.salneurria = salnA;
            this.hornitzailea = horA;
            }

public void ikuskatu()
      {  System.out.println("<<ARTIKULUA: IZENA = "+this.izena+ "\n");
         System.out.println("            DESKRIPZIOA =
"+this.deskripzioa+ "\n");
System.out.println("         SALNEURRIA= "+this.salneurria+ "\n");
             hornitzailea.ikuskatu();
        }}


 class Hornitzailea {

      private String nan;
      private String Izena;
      private String Helbidea;
      private String Telefonoa;
      private String fidagar;

      private Artikulua[ ] artikuluak= new Artikulua[ 10] ;

      Hornitzailea(String nanH, String izenaH,String helbiH,String
telH, String fidH)
            {
              this.nan = nanH;
              this.Izena = izenaH;
              this.Helbidea = helbiH;
              this.Telefonoa = telH;
              this.fidagar = fidH; }


      public void ikuskatu()
       {  System.out.println("<<HORNITZAILEA: NAN = "+this.nan+ "\n");
         System.out.println("          IZENA = "+this.Izena+ "\n");
System.out.println("           HELBIDEA= "+this.Helbidea+ "\n");
System.out.println("           TELEFONOA= "+this.Telefonoa+ "\n");
System.out.println("           FIDAGARRITASUNA= "+this.fidagar+
"\n");
        }  }

interface BezeroaInterface {
      static final float kreditua_LIMIT = 1000f;
      abstract float lortukreditua();
}
```

3.7. KLASE-AZPIKLASE ERLAZIOAK: HERENTZIA

Aplikazio informatikoaren bizi-zikloaren garapena aurrera joan ahala, eta domeinuaren betekizunak aztertu ondoren, aplikazioaren ikuspegi estatikoa egituratzen saiatzen da analista. Azterketa gauzatu ahala, lehenengo eta behin domeinuaren objektu nagusiak identifikatzen ditu eta, ondoren, objektu abstraktu horiek zehaztu, eta klase bihurtuko ditu. Hasieran, klaseak isolatuta agertuko dira Klase Diagraman, gero, analistak, erlazioak atzemango ditu eta Klase Diagrama behin-behineko forma hartuz joango da.

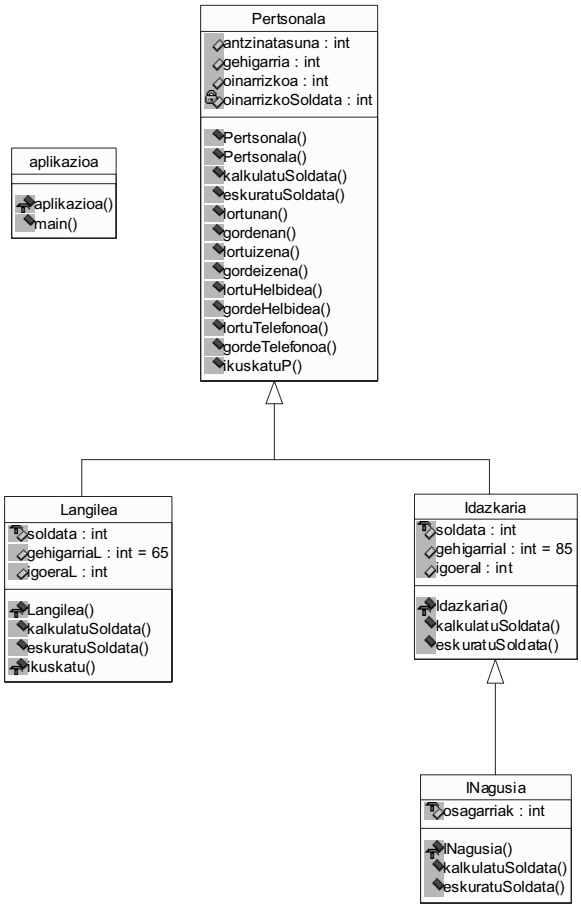
Lehenengo fase horretan, diagramaren egitura laua izango da eta hala izan behar du, konplexutasunak, gero etorriko baitira. Klase Diagramaren lehen bertsioa izanez gero, klaseen arteko antzekotasunak aurkitzen ditu analistak eta Klase Diagramaren berregituraketarekin hasten da. Bigarren fase horretan, herentziak, agregazioak eta antzeko portaerak azaltzen dira klaseen artean eta Klase Diagramaren bigarren bertsioan islatzen dira. Diseinua bukatu ondoren, Klase Diagramak erlazio mota desberdinak izango ditu, batzuk erraz programa daitezkeenak programazio-lengoaien teknologiei esker, eta beste batzuk, ostera, arretaz eta tentuz diseinatu eta programatu behar direnak. Azken sail horretako klaseak programatzeko, programazio-lengoaiak tresnak eskaini behar dizkio programatzaileari lana ondo burutu dezan. Ondoko paragrafoetan Javak eskaintzen dituenak azalduko ditugu.

Has gaitezen klase-azpiklase erlazioa egituratzen. Javan herentzia-erlazioek honako formatu hau izango dute:

```
class azpiklasea extends klasea {
                azpiklasearen gorputza
                    }
```

Alegia, klase bat hedatu nahi dugula adierazteko `extends` hitz erreserbatua erabili behar da eta ondoren hedatu nahi dugun klasea adierazi. Konturatu, `extends`-aren ostean, klase bakarra onartzen dela bakarrik, horrek dakarrenarekin, hau da, eratorpena klase batetik eta bakarretik gauzatu behar dela nahitaez. Beraz, Javan, herentzia anizkuna ezin da programatu zuzen-zuzenean. Geroxeago ikusiko dugu lengoaiak trikimailuak eskaintzen dituela delako herentzia anizkuna izan daitekeena programatzeko.

Herentziaren zehaztasunen azalpena errazteko, adibide bat eratuko dugu eta horren inguruan gauzatuko dugu hausnarketa. Demagun enpresa bateko pertsonalaren soldata kudeatu nahi dugula. Hil-bukaeran enpresako pertsona bakoitzak jasoko duen soldata lanpostuaren arabera egokituko da. Horrela, esaterako, idazkari nagusiak soldata bat izango du, idazkari arruntak beste bat eta langileek beste bat.



3.12. irudia. Aplikazioaren Klase Diagrama.

Domeinuaren azterketa egin ondoren **3.12. irudia**n agertzen den Klase Diagrama eraiki dugu. Klase Diagrama eraiki ondoren eta enpresan eman dizkiguten soldatei buruzko kudeaketa-arauak jasota, aplikazioa programatzen hasi gara:

```
class Pertsonala {

      private String nan;
      private String Izena;
      private String Helbidea;
      private String Telefonoa;
      public int antzinatasuna;
      public int gehigarria;
      public int oinarrizkoa;
      private int oinarrizkoSoldata;
```

```
      public Pertsonala() { };
      public Pertsonala(String nanB, String izenaB,String
helbiB,String telB, float kredB)
            {
             this.nan = nanB;
             this.Izena = izenaB;
             this.Helbidea = helbiB;
             this.Telefonoa = telB;
              }
      public void kalkulatuSoldata( ) {

      oinarrizkoSoldata= antzinatasuna*gehigarria + oinarrizkoa;
                                       }

      public int eskuratuSoldata() {
  return(oinarrizkoSoldata );
                                       }

      }


/** Gorde Pertsonalaren nan-a */
      public void gordenan(String nan) {
            this.nan = nan;
      }
/** Eskuratu Pertsonalaren izena */
      public String lortuizena() {
            return Izena;
      }
// ......
// beste datu pertsonalak antzera
//.......

class Langilea extends Pertsonala {
 int soldata;
 public final int gehigarriaL = 65;
 public int igoeraL;

 // Langilearen kalkulatuSoldata() eta eskuratuSoldata() inplementatzen

 public void kalkulatuSoldata( int orduak ) {

  soldata = orduak * oinarrizkoa + igoeraL*antzinatasuna+ gehigarriaL;
  System.out.println(
   "Langilearen soldata(kalkulatuSoldata()) : "
   + soldata+ "(orduak * oinarrizkoa + igoeraL*antzinatasuna+
gehigarriaL)" );
  }
```

```
 public int eskuratuSoldata() {
  return(soldata );
  }

 // ikuskatu() metodoa definitzen dugu Langilearen soldata ikusteko
 // Kontuz !!!!!, ez dago NireInterfazea interfazean
 void ikuskatu() {
  System.out.println(
   "Langilearen soldata ikuskatzeko metodoa(ikuskatu()) = " + soldata);
  } }


class Idazkaria extends Pertsonala {
 int soldata;
 public final int gehigarriaI = 85;
 public int igoeraI;
 // Idazkariaren kalkulatuSoldata() eta eskuratuSoldata() inplementatzen

public void kalkulatuSoldata( int orduak ) {

  soldata = orduak * oinarrizkoa + igoeraI*antzinatasuna+ gehigarriaI;
      System.out.println(
   "Idazkariaren soldata(kalkulatuSoldata()) : "
   + soldata+" (orduak * oinarrizkoa + igoeraI*antzinatasuna+
gehigarriaI)" );
  }


 public int eskuratuSoldata() {
  return( soldata );
  }
 }

class INagusia extends Idazkaria {

int osagarriak;

// Idazkari Nagusiaren kalkulatuSoldata() eta eskuratuSoldata()
inplementatzen

public void kalkulatuSoldata( int orduak ) {

  soldata = orduak * oinarrizkoa + igoeraI*antzinatasuna+
gehigarriaI+ osagarriak*antzinatasuna;
      System.out.println(
   "Idazkariaren soldata(kalkulatuSoldata()) : "
   + soldata+" (orduak * oinarrizkoa + igoeraI*antzinatasuna+
gehigarriaI+ osagarriak*antzinatasuna)" );
  }
```

```
 public int eskuratuSoldata() {
  return( soldata );
  }
}
```

Listatuan argi ikusten da pertsonal-kategoria bakoitzak soldata desberdina izango duela eta ez bakarrik sartu dituen lan-orduengatik, baita aplikatzen zaion kudeaketa-arauagatik ere. Pertsona bakoitzaren kudeaketa-araua soldata kalkulatzeko aplikatzen den formula matematikoan dago jasota. Eragiketa matematiko hau `kalkulatuSoldata()` metodoan egikaritzen da klase bakoitzean. Hala, programatzaileak atzipen uniforme eta bateratua izango du, zeren soldata kalkulatzeko erabiltzen duen metodoaren signatura bera baita beti.

Konturatu `kalkulatuSoldata()` metodoa birdefinitu[22] egiten dugula azpiklaseetan, alegia, metodoa birdefinitzerakoan, metodoak izen bera izango du, itzultzen duen balioak ere mota berekoa izan behar du eta parametroen kopuruak eta motak ere bat etorri behar dute klasean eta azpiklasean. Hori eginda, superklasearen metodoa ezkutatuta gelditzen da azpiklasean. Ezkutatuak ez du esan nahi suntsitua, metodoak bizirik dihardu, arazoa da atzitzeko era estandarra aldatu egin behar dela. Honako adibideak esango digu nola egin:

```
INagusia idazNagusia = new INagusia() ;
idazNagusia.kalkulatuSoldata(35);
```

Eraiki dugun `INagusia` motako objektuak `INagusia` klaseko `kalkulatuSoldata()` metodoa egikaritzen du. Orain arte arazorik ez, baina zer gertatzen da objektuak `Idazkaria` klasekoa egikaritu nahi badu? nola deitu behar diogu? Erantzuna ez da hain begi-bistakoa: Javak `super` hitz erreserbatua eskaintzen du arazoa konpontzeko: `super.kalkulatuSoldata(40)` idatzi eta kito. Arazoa klasearen atributuetan sortzen bada hemen ere, aldaketa bera gauzatu behar da.

Azpiklaseen metodo eraikitzaileetan erruz erabiltzen da `super` hitz erreserbatu hori. Gehienetan, azpiklase motako objektu bat eraiki nahi denean superklasearen atributuak hasieratu nahi izaten dira. Hori egiteko, superklasearen eraikitzaileari deitu behar zaio eta deiak forma berezia izan behar du: `super(parametro_zerrenda)`. Gure kasuan `INagusia` klasearen eraikitzailean `super(parametro_zerrenda)` ipintzen badugu, `Idazkaria` klasearen eraikitzaileari[23] deituko diogu. Komeni da agindu hori azpiklasearen metodo

22. Metodoa gainkargatua dagoela esaten da izen bera badu baina parametroen kopurua eta mota aldatu egiten bada.

23. Eraikitzailea gainkargatua badago bertsio egokia aukeratzen da pasatzen diren parametro kopuru eta moten arabera.

eraikitzailearen lehen agindua izatea. Metodo eraikitzaileen arteko exekuzioa hierarkikoa da, herentzia bera den legez, alegia, goitik behera exekutatzen da; lehenengo, superklaseen metodo eraikitzaileak exekutatzen dira, eta gero, azpiklaseenak.

Eraikitzaileekin eta hitz erreserbatuekin bukatzeko, klase bereko metodo eraikitzaile batek gainkargatua dagoen beste metodo eraikitzaile bati deitu nahi badio, `this.(parametro_zerrenda)` agindua txertatzea nahikoa du bere aginduen zerrendan.

Halaber, gogoratu metodo bat birdefinitzen badugu bere ikusgaitasuna ez dela murriztu behar, alegia, metodo bati superklasean[24] `protected` maila jarri badiogu ikusgaitasunean, birdefinitzen badugu, `protected` edo `public` maila ipini behar diogu azpiklasean.

Klase-azpiklase erlazioak diseinatu ditugunean gure aplikazioan, klase eta azpiklaseen arteko esleipenak tentu handiz programatu behar dira, ez baita gauza bera superklase motako objektu bat erreferentziatzen dagoen aldagai bati azpiklase motako objektu bat erreferentziatzen dagoen aldagai bat esleitzea edo azpiklase motako objektu bat erreferentziatzen dagoen aldagai bati superklase motako objektu bat erreferentziatzen dagoen aldagai bat esleitzea.

Ahal bada, bigarren mota hori baztertu egin behar da eta ez erabili. Bigarren esleipen mota hori ondo egikaritzeko, baldintza batzuk gertatu behar dira eta gainera egoera nahiko nahasiak sor ditzake kodean. Azter dezagun kontua, lehen aurkeztutako adibidearen ingurune bera erabilita:

```
public class aplikazioa {

public static void main( String args[] ) {

Pertsonala pertsona = new Pertsonala();
Idazkaria idazkari = new Idazkaria();
Idazkaria idazkari1 = new Idazkaria();
Langilea langile = new Langilea();
INagusia idazNagusia= new INagusia();

idazkari = pertsona;

..................... }
........}
```

Goiko kode hori programa nagusian txertatzen badugu eta konpilatzen saiatzen bagara, konpiladoreak errore bat emango digu:

24. Egoerak behartzen gaituenean eta nahasketak saihesteko, klase-azpiklase erlazioa superklase-azpiklase erara aurkeztuko dugu.

```
aplikazioa.java:178: incompatible types
found: Pertsonala
required: Idazkaria
idazkari = pertsona
           ^
```

Alegia, ez digu onartzen `azpiklase=superklase` motako esleipen bat programatzea. Arazoa konpondu nahian datu-bihurketa bat egiten dugu eta orain bai, konpiladoreak ez digu errorerik emango:

```
Idazkari = (Idazkaria)pertsona
```

Egikaritzen dugu kodea eta berriro errorea!

```
java.lang.ClassCastException
      void aplikazioa.main(java.lang.String[])
Exception in thread main
```

Konpilazio-garaian onartzen da datu-bihurketa baina exekuzioan nahitaez `pertsona` aldagaiak `Idazkaria` motako objektu bat erreferentziatzen egon behar du lerroa ondo exekutatzeko. Hori jakinda honako hau idazten dugu:

```
pertsona = idazkari1;
idazkari = (Idazkaria)pertsona;
idazkari.kalkulatuSoldata();  // Pertsonala klaseko metodoa
idazkari.kalkulatuSoldata(45); // Idazkaria klaseko metodoa
```

Lau lerrook arazorik gabe prozesatzen dira bai konpilazio-garaian bai exekuzioan. Espero genuen legez lehenengoko `kalkulatuSoldata()` metodoa, `Pertsonala` klasekoa izango da eta bigarrena, ostera, `Idazkaria` klasekoa.

Alda dezagun esleipenaren norabidea eta azter dezagun `superklase=azpiklase` esleipena. Bigarren esleipen mota hori asko erabiltzen da metodo birdefinituak ditugunean. Programatzaile trebeak etekin handia atera diezaioke esleipen mota horri polimorfismoa[25] aplikatuz. Geroxeago aurkeztuko dugu adibide bat kontu hori gehiago lantzen duena. Oraingoz, azter ditzagun metodo birdefinituak eta `superklase=azpiklase` erlazioa. Lehen bezala hona adibide txiki bat:

25. Polimorfismoan atzipen bera erabiliz metodo desberdinak egikaritzen dira. Programa egikaritzerakoan sistemak aztertuko du zein metodo egikaritu behar duen erabakitzeko, aldagaia, zein objektu mota dagoen erreferentziatzen.

```
pertsona = idazkari ;
pertsona.kalkulatuSoldata(35); // Idazkariaren soldata kalkulatu

pertsona = langile ;
pertsona.kalkulatuSoldata(35); // Langilearen soldata kalkulatu

pertsona.kalkulatuSoldata();   // Pertsonalaren soldata
//                             kalkulatu

// pertsona.ikuskatu();  Ez du funtzionatzen Pertsonala
//                             objektuan bilatzen saiatzen da

idazkari = idazNagusia;
idazkari.kalkulatuSoldata(12); // Idazkari Nagusiaren soldata
//                             kalkulatzen du !!!!!!
```

Adibide hori konpilatzen badugu, konpiladoreak ez digu inongo erroremezurik emango eta gainera lasai-lasai egikaritu dezakegu kodea. Konturatu 1 eta 3 lerroetan egin dugun `superklase=azpiklase` esleipenak ondorio nabariak eragin dituela ondoko lerroetan: superklasearen (`pertsona`) erreferentzia bat erabiltzen dugu azpiklase (`idazkari` eta `langile`) motako objektu batera atzitzeko. Baina kontuz, metodoei dei egiten diegunean egikaritzeko, superklasearen metodoak bakarrik onartzen dira, alegia, superklasearen motak mugatu egiten du zein metodo erabil dezakegun eta zein ez. Salbuespen batekin, metodo birdefinituekin, hain zuzen.

Metodo birdefinitu bati dei egiten badiogu, egikarituko den bertsioa erreferentziatzen ari garen objektuarena (`Idazkaria` eta `Langilea`) izango da eta ez aldagaiaren (`Pertsonala`) motakoa. Estekadura dinamikoa da eta ez da konpilazio-garaian auzia konpontzen, egikaritze-garaian erabakitzen da zein bertsio egikaritu. Horratx polimorfismoa, signatura bera iturburuan (2 eta 4 lerroak) baina portaera desberdina egikaritzerakoan.

Esandako guztia azalpen teoriko huts eta antzua izango litzateke ez bagenio inongo erabilera praktikorik aurkituko. Goazen, bada, esandako guztiari praktikotasun bat bilatzera.

Demagun kontratatu gaituen enpresak soldatak kudeatzeko honako funtzionalitate berria eskatzen digula: soldatak kudeatuko dituen aplikazioak pertsonalaren soldata guztiak kalkulatu eta pantailan ikuskatu behar ditu. Eman dezagun gauzak errazteko, enpresaren pertsonal guztia `Pertsonala` motako array batean biltegiratu dugula. Pertsona bakoitzak enpresan sartu dituen orduak beste array batean etorriko dira.

Pertsonalaren klaseak eta azpiklaseak aurreko paragrafoetan esan dugun erara

egituratu baditugu, nahikoak izango zaizkigu hiru (!!!) lerro[26] funtzionalitate berria inplementatzeko:

```
Pertsonala pertsona = new Pertsonala[100];
pertsona[0] = new Idazkaria(.....);
pertsona[1] = new INagusia(.....);
pertsona[2] = new Langilea(.....);
pertsona[3] = new INagusia(.....);
.....
for(int i=0;pertsona[i]!= null; i++) {
System.out.print(pertsona[i].lortunan()+":"); // Hemen Pertsonala
System.out.println(pertsona[i].kalkulatuSoldata()); // Hemen AUSKALO!!
                                                    };
```

Egin duguna nahiko sinplea da eta ezaguna egingo zaigu: erreferentzia egokiak prestatu eta polimorfismoa aplikatu. Egikaritze-garaian, `for` aginduaren barruan dauden beste bi aginduek ematen digute eskatutako funtzionalitatea. Lehenengoak, pertsonalaren izena eskuratzen digu eta bigarrenak, ostera, polimorfismoa erabiliz, eskatutako soldata. Soldatak kalkulatzeko sistemak erabiliko duen `kalkulatuSoldata()` metodoaren bertsioa[27] egikaritze-garaian erabakiko du eta aldagaia erreferentziatzen ari den objektuarena izango da.

3.8. KLASE ABSTRAKTUAK ETA INTERFAZEAK

Aplikazio informatiko bat eraikitzen ari garenean, diseinua gauzatu ondoren, klaseen arteko erlazioak agertuko dira Klase Diagraman, aplikazioaren egitura estatikoa zein izango den finkatzeko. Klase horien arteko erlazioak, hierarkikoak direnean, klase-azpiklase erlazio bidez adieraziko dira. Erlazio mota horiek herentzia bidez inplementatzen dira programazio-lengoaietan. Hala, klaseek dituzten pareko atributu eta portaerak, superklasean kokatzen dira software berrerabilgarriagoa eta modularragoa sortzeko. Gehienetan, superklasean kokatzen ditugun metodoek, oinarrizko inplementazioa izaten dute, gero, herentzia bidez, inplementazio hori birfindu egiten da metodoa birdefinituz edo gainkargatuz. Beste batzuetan, superklasean kokatzen dugun metodoak ez du inongo oinarrizko inplementazioarik izan behar, kasu hauetan, metodoaren signatura adierazi nahi dugu bakarrik. Hori egiteko, superklasea, abstraktu gisa definitu behar dugu:

```
public abstract class klaseIzena
{ ..............
public abstract void metodoa( );
......}
```

26. `for` aginduaren gaineko lerroek objektuak eraiki eta erreferentziatzeko bakarrik balio dute. Funtzionalitate guztia `for` aginduaren gorputzean kokatzen da.

27. Kontu egin, adibide honetan, metodoa birdefinitu egiten dela azpiklaseetan.

Konturatu, klase osoa abstraktu gisa definitu behar dela, klaseak, gutxienez, metodo bat abstraktu gisa erazagututa badauka. Aurreko atalean aztertu dugun adibidea berreskuratuko dugu berriro, esan duguna azaltze aldera. Demagun, orain, gure klase-azpiklase hierarkian `Petsonala` klaseak ez diola `kalkulatuSoldata(int)` metodoari inongo inplementaziorik itsatsi nahi, alegia, klase-hierarkian hasierako superklasea generiko samar bihurtu nahi dugu eta metodo batzuk erazagututa eta ez definituta utzi nahi ditugu. Gero, klase hori hedatzen duten azpiklaseek, bakoitzak bere beharretara egokituta, inplementazio egokia erantsiko diete metodo abstraktu horiei.

```
abstract class Pertsonala {
..............
public abstract void kalkulatuSoldata( int orduak );
..........}
```

Klase abstraktu bat ezin dugu instantziatu, hori egiten saiatzen bagara konpiladoreak errore-mezu hau emango digu:

```
Pertsonala.java:xx: class Pertsonala is an abstract class.
      It canÏt be instantiated.
new Pertsonala
^
```

Klase abstraktu bat erabiltzeko heredatu egin behar dugu eta hemen hasten dira buruhausteak: azpiklase batek klase abstraktu bat heredatu ondoren ezin du gehiagorik heredatu, bukatu zaio kupoa; Javak ez du herentzia anizkuna onartzen. Eta aipatutako azpiklaseak, ezaugarri gehiago eratortzea nahi badu, ezin du ezer egin? Egin, egin dezake zerbait: interfazeak erabili. Zehatz-mehatz, interfazeak erabilita ez ditugu ezaugarriak eratortzen orain arte esan dugun eran. Javan eratorri ordez inplementatu egiten dira horrelako ezaugarriak:

```
class klasea extends superklasea

      implements [ interfazeak]*
```

Goiko formatuarekin non inplementatzen den interfazea adierazten dugu, honako honekin, nola definitzen den:

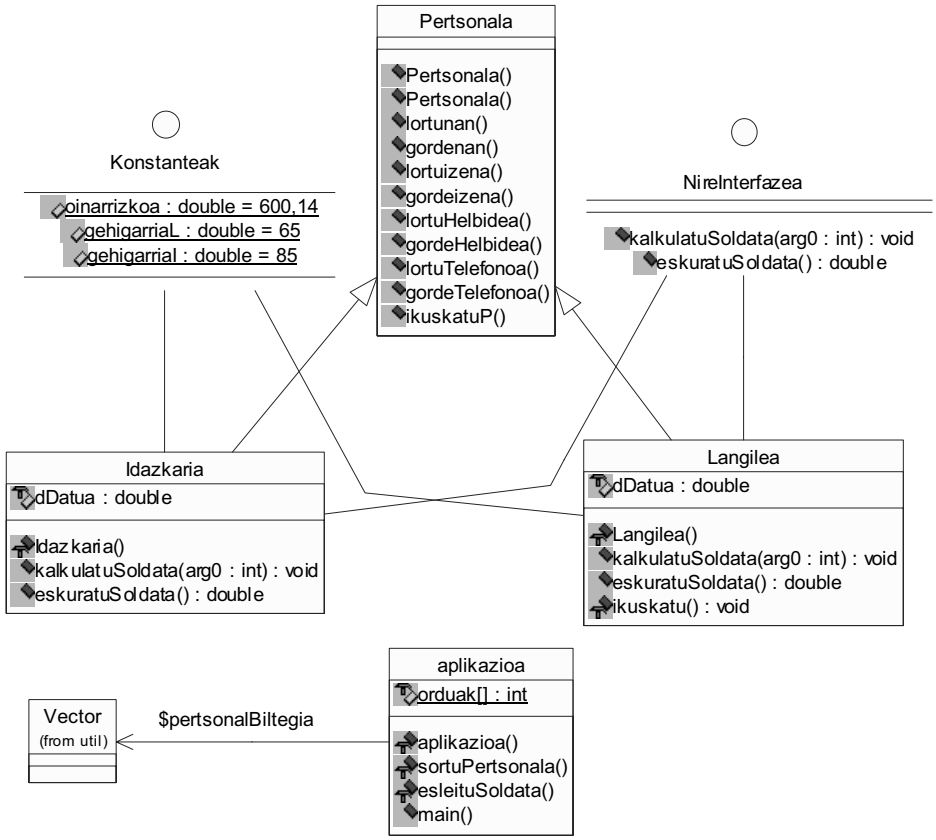
```
     [ public]  interface  <interfazearenIzena>  extends
[ <interfazeak>]*
     {
           <interfazearen gorputza>
     }
```

Ikusgaitasuna `public` izaten denean, interfazea atzigarri bihurtzen da edozein klaserentzat edozein paketetatik. Interfazearen gorputzak metodoen eta aldagai konstanteen erazagupenak bakarrik eduki ditzake. Interfaze batean

erazagutzen diren metodo guztiak publikoak (`public`) eta abstraktuak (`abstract`) dira eta konstante guztiak publikoak (`public`), bukaerakoak (`final`) eta estatikoak (`static`).

Diseinuaren ikuspegitik interfaze bat portaera-protokolo bat da klase-hierarkian edozein klasek inplementa lezakeena. Klaseak ez du interfaze bat heredatzen, ez baitago klase-azpiklase erlaziorik beraien artean. Klaseak, portaera bat eskuratzen du norbaitek diseinatu duena eta patroi bezala erabiliko duena.

3.13. irudian, aurreko atalean landu dugun adibide bera eskuratu dugu, baina oraingoan diseinua beste era batera bideratu dugu interfazeen ezaugarriak aztertzeko.



3.13. irudia. Aplikazio berriaren Klase Diagrama.

```
      import java.util.Vector;

      interface Konstanteak {
        public final  double oinarrizkoa = 600.14;
        public final  double gehigarriaL = 65;
        public final  double gehigarriaI = 85;
        }

      interface NireInterfazea {
        void kalkulatuSoldata( int orduak );
        double eskuratuSoldata();
        }

      class Pertsonala {

            private String nan;
            private String Izena;
            private String Helbidea;
            private String Telefonoa;

            public Pertsonala() {  };
            public Pertsonala(String nanB, String izenaB,String
helbiB,String telB, float kredB)
                     {
                       this.nan = nanB;
                       this.Izena = izenaB;
                       this.Helbidea = helbiB;
                       this.Telefonoa = telB;
                        }

            /** Eskuratu Pertsonalaren nan-a */
            public String lortunan() {
                     return nan;
            }
            /** Gorde Pertsonalaren nan-a */
            public void gordenan(String nan) {
                     this.nan = nan;
            }
// ......
// beste datu pertsonalak antzera
//.......

/** Ikuskatu Pertsonalaren informazio pertsonala*/
       public void ikuskatuP()
        { System.out.println(" <<PERTSONALA: NAN = "+this.nan+ "\n");
        System.out.println("            IZENA = "+this.Izena+ "\n");
    System.out.println("           HELBIDEA= "+this.Helbidea+ "\n");
   System.out.println("           TELEFONOA= "+this.Telefonoa+ "\n");
        }}
```

```
class Langilea extends Pertsonala implements Konstanteak,
NireInterfazea {
  double soldata;

// Langilearen kalkulatuSoldata() eta eskuratuSoldata() inplementatzen

  public void kalkulatuSoldata( int orduak ) {

    soldata = (double)orduak * oinarrizkoa + gehigarriaL;
    System.out.println(
      "Langilearen soldata(kalkulatuSoldata()) :  "
         + Math.round(soldata)+ "(orduak * oinarrizkoa +
Langile_gehigarria)" );
    }

  public double eskuratuSoldata() {
    return(soldata );
    }

  // ikuskatu()  metodoa definitzen dugu Langilearen soldata ikusteko
  // Kontuz !!!!!, ez dago NireInterfazea interfazean
  void ikuskatu() {
    System.out.println(
      "Langilearen soldata ikuskatzeko metodoa(ikuskatu()) = " +
Math.round(soldata) );
    }
  }

class Idazkaria extends Pertsonala implements Konstanteak,
NireInterfazea {
  double soldata;


   //Idazkariaren kalkulatuSoldata() eta eskuratuSoldata()
// inplementatzen
   public void kalkulatuSoldata( int orduak ) {

    soldata = (double)orduak * oinarrizkoa + gehigarriaI;
      System.out.println(
      "Idazkariaren soldata(kalkulatuSoldata()) :  "
         + Math.round(soldata)+ "(orduak * oinarrizkoa +
Idazkari_gehigarria)" );
    }

  public double eskuratuSoldata() {
    return( soldata );
}}
```

Interfaze bat berez ezin da instantziatu, beti izan behar da inplementatua. Horrek ez du esan nahi ezin dugula erazagutu interfaze motako aldagairik. Guk bat erazagutuko dugu:

```
NireInterfazea pertsona=
    (NireInterfazea)aplikazioa.pertsonalBiltegia.elementAt(i);
```

Agindu horretan, objektu bat eskuratu dugu bektoretik `elementAt()` metodoa erabilita eta datu-bihurketa `(NireInterfazea)` aplikatu diogu erreferentzia ondo gauzatzeko. Konturatu eskuratzen dugun objektuak (`Langilea` edo `Idazkaria` motakoa) interfazea inplementatzen duela, inplementatu behar duela, esango genuke, arazorik ez sortzeko. Kodean ondoren agertzen dena ezaguna egingo zaigu: polimorfismoa[28] aplikatuta objektu mota bakoitzaren `kalkulatuSoldata()` metodoari dei egiten diogu metodoaren bertsio egokia egikaritzeko. Hona hemen, aplikazio osoaren iturburu-kodea eta irteera estandarretik ateratzen den informazioa:

```
class aplikazioa {

       static int [] orduak= {10,20,30,40,50,15,25,35,45,55};
       static Vector pertsonalBiltegia = new Vector();

static void sortuPertsonala() {
       Langilea langilea = new Langilea();
       Idazkaria idazkaria = new Idazkaria();
       for (int i=0;i<5;i++) pertsonalBiltegia.addElement(langilea);
       for (int i=0;i<10;i++) pertsonalBiltegia.addElement(idazkaria);

       } // sortuPertsonala

static void esleituSoldata() {

       for (int i=0;i<10;i++) {
       NireInterfazea pertsona =
(NireInterfazea)aplikazioa.pertsonalBiltegia.elementAt(i);

       pertsona.kalkulatuSoldata(orduak[ i] );
       double soldata= pertsona.eskuratuSoldata();
       System.out.println("\n Pertsonal [ "+i+" ]"+" soldata :"+
Math.round(soldata));

                          }
} // esleituSoldata
```

28. Polimorfismoaren bertsio gogorra dela esango genuke: hemen metodoak ez dira birdefinitzen klase-azpiklase erlazioetan legez, ez baitago hasierako inplementaziorik aldatu beharrik; hemen metodoak *inplementatu* egiten dira. Beraz, interfazearen metodo guztiei dei egin diezaiekegu.

```
public static void main( String args[ ] ) {
    System.out.println(
      " objLangilea instantzia Langilea motakoa, sartu datuak eta
ikuskatu soldata" );
    Langilea objLangilea = new Langilea();
    objLangilea.kalkulatuSoldata( 22 );
    objLangilea.ikuskatu();

    System.out.println(
      "\n objLangilea esleitzen dio NireInterfazea motako
erreferentzia bati :objLangileaA" );
    NireInterfazea objLangileaA = objLangilea;
    System.out.println(
      " objLangileaA-ren kalkulatuSoldata()metodoarei deitzen soldata
kalkulatzeko");
    objLangileaA.kalkulatuSoldata( 24 );
    System.out.println(
      " objLangileaA-ren eskuratuSoldata()metodoarei deitzen soldata
ikuskatzeko");
    System.out.println("soldata objLangilea = " +
objLangileaA.eskuratuSoldata() );

    System.out.println(
      "\n Idazkaria motako objektu bat sortu : \n" +
      " eta NireInterfazea motako erreferentzia bati :objIdazkaria
\n" );
    NireInterfazea objIdazkaria = new Idazkaria();
    System.out.println(
      " kalkulatuSoldata()metodoari deitzen soldata gordetzeko \n");

       objIdazkaria.kalkulatuSoldata(35);
    System.out.println(
      "eskuratuSoldata()metodoarei deitzen soldata ikuskatzeko" );
    System.out.println( "datua objIdazkaria = " +
objIdazkaria.eskuratuSoldata() );

    System.out.println(
      "\n objIdazkaria-ri objLangilea esleitzen diogu eta ondoren :
objIdazkaria.eskuratuSoldata()" );
    objIdazkaria = objLangilea;
    System.out.println( "soldata objIdazkaria = " +
objIdazkaria.eskuratuSoldata() );

    System.out.println(
      " objLangileaA-ren kalkulatuSoldata() metodoari deitzen soldata
aldatzeko");
    objLangileaA.kalkulatuSoldata( 16 );
    System.out.println(
      " objIdazkaria-ri objLangileaA esleitzen diogu eta ondoren:
```

```
objIdazkaria.eskuratuSoldata()");
    objIdazkaria = objLangileaA;

    System.out.println( "soldata objIdazkaria = " +
objIdazkaria.eskuratuSoldata() );

    System.out.println(
      "\n objIdazkaria  berriro hasieratzen dugu" );

    objIdazkaria = new Idazkaria();
    objIdazkaria.kalkulatuSoldata( 11 );
    System.out.println(
      " objLangileaA-ri objIdazkaria esleitzen diogu eta ondoren :
objLangileaA.eskuratuSoldata()" );
    objLangileaA = objIdazkaria;
    System.out.println( "soldata objLangileaA = " +
objLangileaA.eskuratuSoldata() );

    System.out.println(
      "\n objLangilea-ri objIdazkaria esleitu nahi diogu. " +
      "\n Ez du funtzionatzen datu-bihurketa burutu behar da "+
      "\n NireInterfaze-tik Langilea-ra" );
    // objLangilea = objIdazkaria;   // Ohar bezala ondo egikaritzeko

    System.out.println(
      "\n objLangileaA-ren ikuskatu()  metodoari deitzen diogu" +
      " ikuskatu() metodoa ez dago erazagutua " +
      " NireInterfazea interfazean. EZ dago ondo programatua!!!!" );
    // objLangileaA.ikuskatu();  // Ohar bezala ondo egikaritzeko


System.out.println("=========================================");

      aplikazioa.sortuPertsonala();
      aplikazioa.esleituSoldata();

      } // main amaiera
        } // aplikazioa klasearen amaiera
```

oOo

```
objLangilea instantzia Langilea motakoa, sartu datuak eta ikuskatu
soldata
Langilearen soldata(kalkulatuSoldata()) :  13268(orduak * oinarrizkoa
+ Langile_gehigarria)
Langilearen soldata ikuskatzeko metodoa(ikuskatu()) = 13268

 objLangilea esleitzen dio NireInterfazea motako erreferentzia bati
:objLangileaA
 objLangileaA-ren kalkulatuSoldata()metodoarei deitzen soldata
kalkulatzeko
```

```
Langilearen soldata(kalkulatuSoldata()) :  14468(orduak * oinarrizkoa
+ Langile_gehigarria)

 objLangileaA-ren eskuratuSoldata()metodoarei deitzen soldata
ikuskatzeko
soldata objLangilea = 14468.36

 Idazkaria motako objektu bat sortu :
 eta NireInterfazea motako erreferentzia bati :objIdazkaria

 kalkulatuSoldata()metodoari deitzen soldata gordetzeko

Idazkariaren soldata(kalkulatuSoldata()) :  21090(orduak * oinarrizkoa
+ Idazkari_gehigarria)
eskuratuSoldata()metodoarei deitzen soldata ikuskatzeko
datua objIdazkaria = 21089.899999999998

 objIdazkaria-ri objLangilea esleitzen diogu eta ondoren :
objIdazkaria.eskuratuSoldata()
soldata objIdazkaria = 14468.36
 objLangileaA-ren kalkulatuSoldata() metodoari deitzen soldata
aldatzeko
Langilearen soldata(kalkulatuSoldata()) :  9667(orduak * oinarrizkoa +
Langile_gehigarria)
 objIdazkaria-ri objLangileaA esleitzen diogu eta ondoren:
objIdazkaria.eskuratuSoldata()
soldata objIdazkaria = 9667.24

 objIdazkaria  berriro hasieratzen dugu
Idazkariaren soldata(kalkulatuSoldata()) :  6687(orduak * oinarrizkoa
+ Idazkari_gehigarria)
 objLangileaA-ri objIdazkaria esleitzen diogu eta ondoren :
objLangileaA.eskuratuSoldata()
soldata objLangileaA = 6686.54

 objLangilea-ri objIdazkaria esleitu nahi diogu.
 Ez du funtzionatzen datu-bihurketa burutu behar da
 NireInterfaze-tik Langilea-ra

 objLangileaA-ren ikuskatu()  metodoari deitzen diogu ikuskatu()
metodoa ez dago erazagutua  NireInterfazea interfazean. EZ dago ondo
programatua!!!!
==================================================
Langilearen soldata(kalkulatuSoldata()) :  6066(orduak * oinarrizkoa +
Langile_gehigarria)

 Pertsonal [  0 ]  soldata :6066
Langilearen soldata(kalkulatuSoldata()) :  12068(orduak * oinarrizkoa
+ Langile_gehigarria)
```

```
 Pertsonal [  1 ] soldata :12068
Langilearen soldata(kalkulatuSoldata()) :  18069(orduak * oinarrizkoa
+ Langile_gehigarria)

 Pertsonal [  2 ] soldata :18069
Langilearen soldata(kalkulatuSoldata()) :  24071(orduak * oinarrizkoa
+ Langile_gehigarria)

 Pertsonal [  3 ] soldata :24071
Langilearen soldata(kalkulatuSoldata()) :  30072(orduak * oinarrizkoa
+ Langile_gehigarria)

 Pertsonal [  4 ] soldata :30072
Idazkariaren soldata(kalkulatuSoldata()) :  9087(orduak * oinarrizkoa
+ Idazkari_gehigarria)

 Pertsonal [  5 ] soldata :9087
Idazkariaren soldata(kalkulatuSoldata()) :  15089(orduak * oinarrizkoa
+ Idazkari_gehigarria)

 Pertsonal [  6 ] soldata :15089
Idazkariaren soldata(kalkulatuSoldata()) :  21090(orduak * oinarrizkoa
+ Idazkari_gehigarria)

 Pertsonal [  7 ] soldata :21090
Idazkariaren soldata(kalkulatuSoldata()) :  27091(orduak * oinarrizkoa
+ Idazkari_gehigarria)

 Pertsonal [  8 ] soldata :27091
Idazkariaren soldata(kalkulatuSoldata()) :  33093(orduak * oinarrizkoa
+ Idazkari_gehigarria)

 Pertsonal [  9 ] soldata :33093
```

Klase berean interfaze bat baino gehiago inplementatzen dugunean, arazoak sor daitezke metodoa inplementatzerakoan, metodoak izen bera badu interfaze bitan edo gehiagotan. Hiru egoera sor daitezke:

1. Metodo guztiek signatura bera dute. Kasu honetan metodoa klasean behin definitzen dugu eta kito.
2. Metodoen signatura desberdina da. Hemen ere ez dugu arazo handirik: metodo guztiak definitzen dira klasean bakoitza bere inplementazioarekin eta aurrera.
3. Oraingoan itzultzen den balioan aurkitzen da desberdintasuna. Diseinu-arazoak ditugu hemen, konpiladoreak errorea emango digu klase batek ezin baititu eduki bi metodo, den-dena berdina dutenak, eta aldi berean, itzultzen den balioaren mota desberdina izatea.

3.9. LIBURUTEGI GENERIKOAK: JGL

1996ko ekainean, ObjectSpace software-enpresak, STL[29] liburutegiaren Java bertsioa ateratzen du merkatura doan. Berehala, IDEak eraikitzen dituzten enpresa nagusiek beren tresnetan txertatzen dute liburutegi berria. ObjectSpace enpresak JGL (Java Generic Library) izena jartzen dio liburutegi berriari hasiera batean, baina Sunen departamentu juridikoari ez zaio izena atsegin eta azkenean, Generic Collection Library izenarekin egingo da ezaguna liburutegi berria, nahiz eta, bere akronimoak JGL izaten jarraitu.

Liburutegiak, denboran zehar, egokitzapen txikiak izango ditu eta 1997ko urte-bukaeran, ObjectSpacek JGLren 3.1 bertsio egonkorra aterako du. Bertsio horretan, pakete nagusiak azpipaketeetan zatituko dira eta Voyager ORBa[30] integratuko zaio liburutegi nagusiari. Voyager ORBk azpiegitura eskaintzen zuen sistema banatuak eraikitzeko JGL erabilita.

Ia bost urtean JGLren 3.1 bertsioak ez zuen aldaketa berririk izan baina 2002ko uztailean, Recursion Software enpresak ObjectSpace bereganatu ondoren, 4.0 bertsio komertziala kaleratu zuen. Agur *freeware*, ongi etorri, *shareware*. Bertsio berria eguneratuta atera zen eta erabat bateragarria eta integratua Sunen Collections APIarekin. Hona hemen integrazioa zertan datzan:

- JGLren `Container` klase guztiek `java.util.Collection` inplementatzen dute.
- JGLren `Sequence` klase guztiek `java.util.List` inplementatzen dute.
- JGLren iteratzaile guztiek, `java.util.Iterator` interfazea hedatzen dute.
- JGLren `BidirectionalIterator` interfazeak `java.util.ListIterator` interfazea hedatzen du.
- JGLren algoritmo guztiek `Collection` eta `List` objektu estandarrekin egin dezakete lan.

Oraingoz, azkena den bertsio[31] horretan, eta arrakasta ezak bultzatuta, Voyager ORB azpiegitura baztertua izan da behin betiko.

Hasiera batean, JDKren hutsune nabari bat betetzeko sortu zen JGL liburutegia. Lehen bertsioetan, JDKren funtzionalitatea data-bildumak eta algoritmo generikoak tratatzeko nahiko apala zen eta programazio generiko oinarrizkoena ere ezin zen programatu JDK erabilita.

---

29. `http://www.sgi.com/tech/stl/`

30. Object Request Broker.

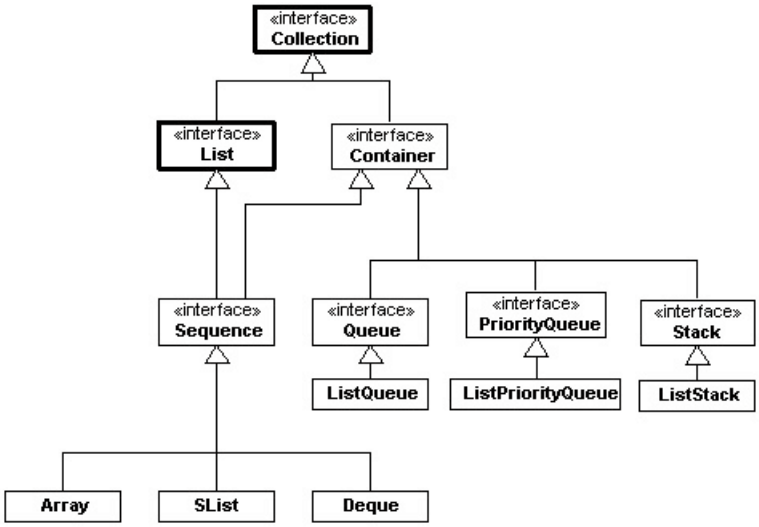
31. Txikikeria dirudien arren, JGL liburutegia erabili nahi duen programatzaileak kontuan izan behar du zein bertsiorekin ari den lanean, paketeak izendatzeko era aldatu egin baita. Orain arte paketeak, `com.objectspace.jgl` izen-eremuan kokatuta zeuden; 4.0 bertsioan, ostera, `com.recursionsw.jgl` izen-eremu berrian kokatuta daude.

JGL da STL liburutegiaren ondorengo zuzena. JGLk STLren oinarrizko arkitektura heredatzen du liburutegiaren paketeak antolatzeko orduan. Hala, STLren antzera, liburutegiak hiru zutabe nagusi erabiliko ditu programazio generikoaren paradigma egituratzeko: edukiontziak, algoritmoak eta iteratzaileak.

Programatzaileak algoritmo generikoak bere beharretara egokitu nahi baditu, liburutegiak funtzio-objektuak eta predikatu-objektuak eskaintzen ditu egokitzelana gauzatzeko. Oinarrian, programazio generikoak, datu-egiturak eta algoritmoak banatuta aztertzen ditu. Algoritmoarentzat, datu-egitura edukiontzi generiko bat izango da eta horren gainean aplikatuko da algoritmoaren funtzionalitatea. Hori guztia, mota parametrizatuak (template) erabiliz gauzatzen zen STL liburutegian, oraingoz[32] Javak ez ditu mota horiek onartzen eta hori dela eta JGL liburutegia interfazeak erabiliz eraiki zen.

Sunen `Collections` APIak antzekotasun handiak ditu JGL liburutegiarekin, nahiz eta, bata eta bestea eraikitzeko erabili diren software-arkitekturak desberdinak izan. Hainbat funtzionalitate antzekoak dira eta askotan programatzaileak zalantzak izango ditu noiz erabili baliabide bat eta noiz bestea.

Programatzaileari lagungarri gertatuko zaiolakoan, JGLren azalpenean `Collections` APIaren ezaugarriak tartekatuz joango gara datozen lerroetan, irakurleak konparaketak egin ditzan eta hala, erabaki, noiz erabili bat eta noiz bestea. **3.14. irudia**k JGLren edukiontzi nagusiak zein diren eta nola dauden egituratuta azaltzen du. Halaber zehazten du nola gauzatzen den integrazioa Java estandarraren `Collections` APIarekin.



3.14. irudia. JGL liburutegiaren edukiontzi nagusiak.

32. JSR 14 lan-taldeak mota generikoen prototipo bat landu eta proposatu du oraintsu.

Desberdintasunak bilatzen hasita, nagusiarekin hasiko gara: `Collections` APIak edukiontzietan kokatzen du erabat bere arkitekturaren diseinu nagusia; JGLk bestalde, aitzinean aipatu dugun legez, programazio generikoaren ildoari ematen dio lehentasuna, hau da, edukiontziak eta algoritmoak zatituta diseinatzeari kontzeptualki. Hala, paketeen antolaketa desberdina izango da batean eta bestean. **3.15. taula**k argi erakusten digu JGLren antolaketa xehatuagoa dela, pakete desberdinetan antolatzen baititu liburutegiaren arlo funtzional guztiak. Taulako zenbakiek azterketa txiki bat merezi dute, adibidez `jgl.adapters` paketearen 29 klaseak nahiko oinarrizkoak dira. Klase horiek klase egokitzaileak dira Javaren array natiboak JGLren edukiontzietara bihurtzeko, eta hala, JGLren algoritmoekin lan egin. Horrela, algoritmoak edukiontzian edozein aldaketa gauzatzen badu, aldaketa hori islatuta gelditukо da Java estandarraren array natiboan ere. Bestalde, `jgl.algorithms` paketearen 22 klaseek metodo estatikoak dituzte bakarrik algoritmoak egikaritzeko.

Bukatzeko, `jgl.functions` eta `jgl.predicates` paketeen 71 klaseak algoritmoak egokitzeko erabiltzen dira eta lehen hurbilpen batean, ez dira beharbeharrezkoak izaten. Behin programatzailea liburutegiaren nondik norakoak menderatzen hasten denean, pakete horiek oso erabilgarriak izaten dira, algoritmoaren portaera programatzailearen beharretara egokitzeko, geroago ikusiko dugun bezala.

	.jgl	.jgl.adapters	.jgl.algorithms	.jgl.functions	.jgl.predicates	.jgl.util	java.util
Interfazeak	12	0	0	0	0	0	10
Klaseak	22	29	22	22	49	7	13
Salbuespenak	1	0	0	0	0	0	2

3.15. taula. JGL liburutegiaren antolaketa.

JGLren edukiontzi bakoitzak `Container` interfazea inplementatzen du. Interfaze horrek `clone()`, `maxSize()`, `start()` eta `finish()` metodoak definitzen ditu. Lehenengoak, edukiontziaren kopia[33] bat itzultzen du; bigarrenak, ostera, une horretan edukiontziak duen elementu kopurua. Hirugarrenak, iteratzaile bat itzultzen du edukiontziaren hasieran kokatua eta, azkenik, laugarrenak ere iteratzaile bat itzultzen du, edukiontziaren bukaeran kokatua.

3.14. irudian ikusten den legez, JGLren edukiontziek beren kategoriei dagokien interfaze propioa inplementatzen dute hierarkiaren bukaeran. Adibidez, `Array`, `Slist` eta `Deque` edukiontziek `Sequence` interfazea inplementatzen dute, interfaze horrek dituelako definituak metodo egokienak edukiontzi horientzat.

33. Kopia horri “shallow copy” deritzo, hau da, jatorrizko edukiontziak eta kopiak objektu multzo bera erreferentziatzen dute. Kopia mota hori azkarrena da eta gehienetan egokiena izaten da programatzailearentzat.

Gauza bera gertatzen da `ListQueue`, `ListPriorityQueue` eta `ListStack` edukiontziekin.

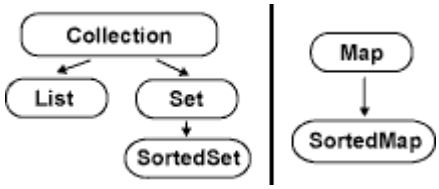
`Stack` eta `Queue` motako edukiontziak `push()` eta `pop()` metodoak erabilita kudeatzen dira. JGLk hiru edukiontzi ditu honako mota hauetakoak:

- `Stack` edukiontzia, FILO (First-in, last-out) motako datu-egitura da. Elementuak biltegiratutako alderantzizko ordenan eskuratzen dira `pop()` metodoa erabilita.
- `Queue` edukiontzia, FIFO (First-in, first-out) motako datu-egitura da. Elementuak biltegiratutako ordena berean eskuratzen dira `pop()` metodoa erabilita.
- `PriorityQueue` edukiontzian elementuen hash balioa hartzen da kontuan eta konparazio bat gauzatzen da. Konparazioaren arabera itzultzen dira balioak, `pop()` metodoa egikaritzen denean. Beraz, konparatzaileak adierazten du eskuratze-ordena. Funtzio-objektu batek eskuratze-ordenaren portaera alda dezake hash balioaren ordez, beste irizpide bat erabilita elementuak eskuratzeko.

`Array`, `SList` eta `Deque` edukiontziek `Sequence` interfazea inplementatzen dute. Gogoratu, `Sequence` interfazeak JGLren `Container` eta `java.util.List` interfazeak hedatzen dituela. `Sequence` bat, edukiontzi lineal bat da eta atzipenak, indizeak erabilita gauzatzen dira.

- `Array` edukiontziak JDKren `Vector` klasearen funtzionalitate bera du. Elementuak oso era eraginkorrean gehitzen dira edukiontziaren bukaeran
- `SList` edukiontzian, elementu berriak, hasieran edo bukaeran txertatzen dira eraginkortasun handiarekin. Horretarako, egituraren nodo bakoitzak erreferentzia bat du elementura eta beste erreferentzia bat hurrengo nodora.
- `Deque` edukiontziak array batean biltegiratzen ditu elementuak. Hala, ausazko atzipenak `Array` edukiontzi batean baino motelago gauzatuko dira, ez ordea, elementuak txertatzen direnean edukiontziaren hasieran edo bukaeran, zeharo azkartuko baita txertatze-abiadura.

Honaino, JGL liburutegiaren edukiontzien nondik norakoa. Gainetik bada ere, azter ditzagun Java estandarraren `Collections` APIaren ordezko interfazeak. **3.16. irudia**k islatzen du APIa nola dagoen egituratuta edukiontzi mailan. `Collection` interfazea ezaguna zaigu dagoeneko, ez ordea beste interfaze guztiak. `Collection` interfazea, oinarrizko interfazea da, `Set` eta `List` interfazeentzat eta ez du zuzeneko inplementaziorik onartzen. `Set` interfazea inplementatzen duen edukiontziak elementu bilduma bat kudeatuko du eta ez du bikoizketarik onartuko. `HashSet` eta `TreeSet` edukiontziek inplementatzen dute `Set` interfazea. `TreeSet` sailkatuta dagoenez `SortedSet` interfazea ere inplementatzen du.



3.16. irudia. Java estadarraren Collections APIa.

`List` interfazea inplementatzen duen edukiontziak bilduma ordenatuak kudeatuko ditu, eta atzipena, indize bidezkoa edo sekuentziala izan daiteke. `ArrayList` eta `LinkedList` edukiontziek inplementatzen dute `List` interfazea.

`Map` interfazea inplementatzen duen edukiontziak gako-balioen bikote-bilduma bat kudeatuko du. `HashMap` eta `TreeMap` edukiontziek inplementatzen dute `Map` interfazea. `TreeMap`-ek, `TreeSet` bezala sailkatuta dagoenez, `SortedMap` interfazea ere inplementatzen du.

JGLren eta `Collections` APIaren edukiontzien azterketa bukatzeko **3.17. taula**k bataren eta bestearen antzekotasunak aztertzen ditu, edukiontziak eta edukiontzien interfazeak inplementatzeko erabiltzen diren klaseak erlazionatuz.

JGL edukiontzia	Inplementatutako JGL interfazeak	`Collections` edukiontzi hurbilena	Inplementatutako `Collections` interfazeak
Array	Container, Sequence	ArrayList, Vector	Collection, List
Deque	Container, Sequence		
DList	Container, Sequence	LinkedList	Collection, List
SList	Container, Sequence	(LinkedList)	Collection, List
HashMap	Container	HashMap, Hashtable	Map
OrderedMap	Container	TreeMap	Map, SortedMap
HashSet	Container, Set	HashSet	Collection, Set
OrderedSet	Container, Set	TreeSet	Collection, Set, SortedSet
PriorityQueue	Container		
Queue	Container		
Stack	Container	Stack	Collection, List

3.17. taula. JGL liburutegiaren eta `Collections` APIaren antzekotasunak.

Ikusitakoak ikusita, erraz ondorioztatzen da `Collections` APIak edukiontziekin bakarrik egituratzen duela APIaren arkitektura nagusia; JGLk, bestalde, STLren oinarrizko arkitekturari jarraituta, edukiontziak eta algoritmoak parean jarriko ditu liburutegia egituratzerako orduan.

JGLk 50 bat algoritmotik gora eskaintzen ditu, erabat generikoak eta hainbat datu-egituratan aplika daitezkeenak. Algoritmoak, JGLren edukiontzietan ez ezik, JDKren `Collections` eta Javaren array natiboetan ere aplika daitezke inongo

arazorik gabe. Horretarako, algoritmoak klaseetan daude antolatuta kategorien arabera. Klase barnean, algoritmo bakoitza metodo publiko eta estatiko bat izango da. Beraz, nahikoa izango zaigu algoritmoa eta haren klasea adieraztea soilik[34] zuzen-zuzenean erabiltzeko programan, metodo estatiko batek ez baitu eskatzen, nahitaez, klasearen instantzia bat izatea metodoa egikaritzeko.

Liburutegiaren bi osagai nagusiak banaka aztertuta, sasoia heldu zaigu batera nola funtzionatzen duten ikusteko. Horretarako, adibide[35] bat aztertuko dugu edukiontzi desberdinetako elementuak sailkatuko dituena:

```
// Copyright(c) 1996-2002 Recursion Software, Inc.
import java.util.Vector;

import com.recursionsw.jgl.Deque;
import com.recursionsw.jgl.adapters.IntArray;
import com.recursionsw.jgl.algorithms.Sorting;
import com.recursionsw.jgl.predicates.LessString;

/**
 * Sorting provides algorithms for sorting a sequence.
 *
 * @see com.recursionsw.jgl.algorithms.Sorting
 * @see com.recursionsw.jgl.functions.LessString
 * @see com.recursionsw.jgl.Deque;
 * @see com.recursionsw.jgl.adapters.IntArray;
 */
public class Algorithms7
{
  public static void main(String[] args)
      {
            // Sort a Vector of Integers
            Vector vector = new Vector();
            vector.add(new Integer(3));
            vector.add(new Integer(-2));
            vector.add(new Integer(4));
            vector.add(new Integer(-5));
            System.out.println("unsorted vector = " + vector);
            Sorting.sort(vector);
            System.out.println("sorted vector = " + vector);

            // Sort a native array of int values
            int[] nativeArray = new int[]{ 3, -2, 4, -5 };
            IntArray intArray = new IntArray( nativeArray );
            System.out.println( "unsorted int[] = " + intArray
);
```

34. KategoriKlasea.algoritmoa(). Adibidez: `Sorting.sort()`

35. Adibidea liburutegiarekin batera dator eta bere horretan aurkezten dizut irakurle. Zehatzmehatz, `Algorithms7.java` fitxategian datorkizu iturburu osoa, konpilatu eta probatu nahi baduzu.

```
        Sorting.sort(intArray);
        System.out.println( "sorted int[] = " + intArray );

        // Sort a Deque of String values alphabetically.
        Deque deque = new Deque();
        deque.add("triangle");
        deque.add("square");
        deque.add("pentagon");
        deque.add("hexagon");
        System.out.println("unsorted deque = " + deque);
        Sorting.sort(deque, new LessString());
        System.out.println("sorted deque = " + deque);
}  }
```

Irteera estandarra:

```
unsorted vector = [ 3, -2, 4, -5]
sorted vector = [ -5, -2, 3, 4]
unsorted int[] = int[] ( 3, -2, 4, -5 )
sorted int[] = int[] ( -5, -2, 3, 4 )
unsorted deque = [ triangle, square, pentagon, hexagon]
sorted deque = [ hexagon, pentagon, square, triangle]
```

Adibideak, bektore bat, Javaren array bat eta JGLren edukiontzi (`Deque`) bat sailkatzen ditu. Bektorea sailkatu baino lehen, `Integer` motako lau elementuz betetzen da, eta ondoren, `Sorting` klasearen `sort()` metodo estatikoa egikaritzen da bektorea sailkatzeko. Metodoak quicksort motako algoritmo eraginkor bat erabiltzen du elementuak sailkatzeko. Ezer adierazten ez den bitartean, elementuak hash balioaren arabera sailkatzen dira. Elementua `Integer` motakoa denean, beraren hash balioa objektuaren balio osoa izango da.

Algoritmoaren jatorrizko portaera aldatu nahi badugu, predikatu bitarren objektu bat jarri behar dugu `sort()` metodoaren bigarren parametroan. Predikatu-objektua, liburutegia erabilita sor dezakegu edo programatzaileak bere nahietara egokitutakoa izan daiteke. Adibidean, hirugarren edukiontziarekin (`Deque`) gauzatzen den sailkapenak JGL liburutegiaren predikatu bitar bat (`LessString`) erabiltzen du edukiontziaren elementuak sailkatzeko:

```
Sorting.sort(deque, new LessString());
```

Predikatu horren bidez elementuak alfabetikoki sailkatuko dira, irteera estandarrean ikusten den legez.

3.9.1. Funtzio eta predikatu klaseak

Funtzio eta predikatu-objektuekin, algoritmoen jatorrizko portaera aldatu eta hedatu egiten da. Objektu horiek, parametro gisa pasatzen zaizkie algoritmoei jatorrizko portaera alda dezaten. Funtzio-objektuek, parametro bat edo gehiago hartzen dute eta objektu bat itzultzen dute. Predikatu-objektuek parametro bat edo

gehiago hartzen dute eta `boolean` motako balio boolear bat itzultzen dute. Bai funtzioek baita predikatuek ere, `execute()` metodo bakarra definitzen dute. Programatzaileak aplikazioan, bere funtzioak eta predikatuak programatu nahi baditu, `execute()` metodo hori egokitu beharko du exekuzioa ondo burutzeko.

Funtzioak `Object` motako parametro bat jasotzen duenean eta `Object` motako emaitza bat itzuli `execute()` metodoarekin, `UnaryFunction` motako interfazea inplementatu duelako izango da. Bestalde, funtzioak, `Object` motako bi parametro jasotzen baditu eta `Object` motako emaitza bat itzuli `execute()` metodoarekin, `BinaryFunction` motako interfazea inplementatu behar da.

Aldi berean, predikatuak, `Object` motako parametro bat jasotzen badu eta boolear motako emaitza bat itzuli `execute()` metodoarekin, `UnaryPredicate` motako interfazea inplementatu duelako izango da. Eta azkenik, predikatuak `Object` motako bi parametro jasotzen baditu eta `boolean` motako emaitza bat itzuli `execute()` metodoarekin, `BinaryPredicate` motako interfazea inplementatu behar du.

Programatzailearen lana errazteko, JGL liburutegiak hainbat funtzio eta predikatu unario eta bitar ditu prest berehala erabiltzeko. Hona hemen, JGLk eskaintzen dituen funtzio eta predikatuak[36] zuzen-zuzenean erabiltzeko:

- **UnaryFunction.**

Izena	Eragiketa
`BindFirst`	P(V, x)
`BindSecond`	P(x, V)
`ConstantFunction`	V
`Hash`	x.hashCode()
`IdentityFunction`	x
`LengthString`	x.toString().length()
`NegateNumber`	-x
`SelectFirst`	x.first
`SelectSecond`	x.second
`ToString`	x.toString()
`UnaryCompose`	P(Q(x))
`UnaryPredicateFunction`	P(x)

- **BinaryFunction.**

Izena	Eragiketa
`BinaryCompose`	P(Q(x), R(y))
`BinaryPredicateFunction`	P(x, y)
`ConstantFunction`	V
`DividesNumber`	x / y
`MinusNumber`	x - y
`ModulusNumber`	x % y
`PlusNumber`	x + y
`PlusString`	x.toString() + y.toString()
`SwappedBinaryFunction`	P(y, x)
`TimesNumber`	x * y

36. *x*-k eta *y*-k `execute()` metodoaren argumentuak adierazten dituzte.

P-k, *Q*-k eta *R*-k beste funtzio/predikatu-objektu batzuk adierazten dituzte. *V-k* balio konstante bat adierazten du.

C-k `java.lang.Class` objektuaren instantzia bat adierazten du.

- **UnaryPredicate.**

Izena	Eragiketa
`BindFirstPredicate`	P(V, x)
`BindSecondPredicate`	P(x, V)
`ConstantPredicate`	V
`InstanceOf`	x instanceof C
`LogicalNot`	!x
`NegativeNumber`	x < 0
`PositiveNumber`	x > 0
`UnaryAnd`	P(x) && Q(x)
`UnaryComposePredicate`	P(Q(x))
`UnaryNot`	!P(x)
`UnaryOr`	P(x) || Q(x)
`UnaryTern`	P(x) ? Q(x) : R(x)

- **BinaryPredicate.**

Izena	Eragiketa
`BinaryAnd`	P(x, y) && Q(x, y)
`BinaryComposePredicate`	P(Q(x), R(y))
`BinaryNot`	!P(x, y)
`BinaryOr`	P(x, y) || Q(x, y)
`BinaryTern`	P(x, y) ? Q(x, y) : R(x, y)
`ConstantPredicate`	V
`EqualCollationKey`	x.compareTo(y) == 0
`EqualCollator`	collator.compare(x, y) == 0
`EqualNumber`	x == y
`EqualString`	x.toString().equals(y.toString())
`EqualTo`	x.equals(y)
`GreaterEqualCollationKey`	(collator.getCollationKey(x.toString())).compare(collator.getCollationKey(y.toString())) >= 0
`GreaterEqualCollator`	collator.compare(x.toString(), y.toString()) <= 0
`GreaterEqualNumber`	x >= y
`GreaterEqualString`	x.toString().compareTo(y.toString()) >= 0

`GreaterCollationKey`	(collator.getCollationKey(x.toString())).compare(collator.getCollationKey(y.toString())) > 0
`GreaterCollator`	collator.compare(x.toString(), y.toString()) > 0
`GreaterNumber`	x > y
`GreaterString`	x.toString().compareTo(y.toString()) > 0
`HashComparator`	x.hashCode() < y.hashCode()
`IdenticalTo`	x == y
`LessEqualCollationKey`	(collator.getCollationKey(x.toString())).compare(collator.getCollationKey(y.toString())) <= 0
`LessEqualCollator`	collator.compare(x.toString(), y.toString()) <= 0
`LessEqualNumber`	x <= y
`LessEqualString`	x.toString().compareTo(y.toString()) <= 0
`LessCollationKey`	(collator.getCollationKey(x.toString())).compare(collator.getCollationKey(y.toString())) < 0
`LessCollator`	collator.compare(x.toString(), y.toString()) < 0
`LessNumber`	x < y
`LessString`	x.toString().compareTo(y.toString()) < 0
`LogicalAnd`	x && y
`LogicalOr`	x || y
`NotEqualCollationKey`	x.compareTo(y) != 0
`NotEqualCollator`	collator.compare(x, y) != 0
`NotEqualNumber`	x != y
`NotEqualString`	!x.toString().equals(y.toString())
`NotEqualTo`	!x.equals(y)
`NotIdenticalTo`	x != y
`SwappedBinaryPredicate`	P(y, x)

Badirudi irakurle, funtzio eta predikatuen prediku hau guztia, zu zorabiatzeko prestatu dudala, sinets iezadazu, ez da izan hori nire asmoa. JGL liburutegiaren oinarrizko ezagutzatik harantzago joan nahi duenari, funtzio eta predikatuen arlo hau ezagutzea eta menderatzea nahitaezkoa zaio, aurrera egin nahi badu eta ur handiagotan sartu.

Ikusi duzun legez irakurle, funtzio eta predikatuen azalpen teorikoak ez du misterio handirik. Erabilerak ere ez dezan eduki, jarraian, adibide batzuk aztertuko ditugu edukiontziak, algoritmoak, funtzioak eta predikatuak jokoan jartzeko.

Lehen adibidean, liburutegiak berak dakarren adibide bat hartuko dugu eta aurreprogramatutako funtzio bitar bat (`TimesNumber`) birprogramatu egingo dugu funtzio bitar hori euskaratuz. Jakina, euskaratzeak ez du ekarriko itzulpen soila egitea bakarrik, `execute()` metodoa ere birprogramatu egin beharko dugu gauzak ondo joateko. Hona hemen esandakoa nola burutu dugun:

```
import com.objectspace.jgl.*;
import com.objectspace.jgl.adapters.*;
import com.objectspace.jgl.algorithms.*;
import com.objectspace.jgl.functions.*;

public class Algorithms9
  {
  public static void main( String[] args )
    {
    int ints2[] = {  2, 4, 2, 3 };
    int ints3[] = {  3, 6, 2, 1 };
    SList list = new SList();
    IntArray intArray2 = new IntArray( ints2 );
    IntArray intArray3 = new IntArray( ints3 );

    // Honako hau da liburutegiak dakarren jatorrizko iturburu-kodea:
    //BinaryFunction function2 = new TimesNumber();
    //Transforming.transform( intArray2, intArray3, list, function2 );

    BinaryFunction biderka = new Biderkatu();
    Transforming.transform( intArray2, intArray3, list, biderka );
    System.out.println( "ints2 = " + intArray2 );
    System.out.println( "ints3 = " + intArray3 );
    System.out.println( "list = " + list );
    System.out.println();
    }
  }

  class Biderkatu implements BinaryFunction {
   public Object execute(Object objektu1, Object objektu2){
        Integer bat= (Integer)objektu1;
        Integer bi= (Integer)objektu2;
        int bat1=bat.intValue();
        int bi1=bi.intValue();
        int hiru= bat1*bi1;
        Integer emaitza= new Integer(hiru);
        return emaitza;
   }  }
```

Irteera estandarra:

```
ints2 = int[] ( 2, 4, 2, 3 )
ints3 = int[] ( 3, 6, 2, 1 )
list = SList( 6, 24, 4, 3   )
```

`transform()` metodoak lau parametro erabiltzen ditu egikaritzeko. Lehenengo biak, jatorrizko edukiontziak izango dira eta horien gainean aplikatuko da laugarren parametroan adierazten dugun funtzio bitarra (`BinaryFunction`), hau da, `TimesNumber` motako funtzio-objektua (`function2`). Funtzio horrek, aldiko, bi edukiontzietako elementuak hartuko ditu binaka eta beraien biderketa gauzatuko du. Emaitza hirugarren edukiontzian utziko da.

Funtzio bitarraren hesteak ikusi nahi ditugunez gure funtzio bitar propioa (`Biderkatu`) programatuko dugu aurreko emaitza bera lortzeko. Horretarako, kodean ikusten den legez, `BinaryFunction` interfazea inplementatzen dugu `execute()` metodoa programatuta. Konturatu, metodoak, `Object` motako bi parametro hartzen dituela eta `Object` motako emaitza bat itzuli. Edukiontzien elementuen mota (`Integer`) nahiko arrunta denez parametro bakoitzaren balio osoa eskuratzen dugu eta biderketa egiten da. Biderketaren emaitza `SList` motako edukiontzi batean gordeko da.

Konturatu biderketaren ordez batuketa bat programatu nahiko bagenu, nahikoa izango litzatekeela, `hiru = bat1*bi1`-en ordez beste lerro hau jartzea:

```
int hiru = bat1+bi1;
```

Funtzio bitar berriaren beste kode guztia, `Biderkatu` klasean agertzen den bera izango da.

Hemen ikusten da, funtzio eta predikatu-objektuen nondik norakoak ezagututa, programatzaileak eskuratzen duen ahalmena. Programatzailea elkarrizketan jartzen da liburutegiarekin, lan astuna[37] liburutegiaren esku uzten da eta lan malgua, ostera, bere ardurapean gelditzen da.

Aurkeztu dugun adibidea sarrerako adibide bat zen. Alabaina, JGL liburutegiari etekina atera nahi badiogu, ohitu egin behar dugu edukiontzietan domeinuaren objektuak ikusten eta tratatzen. Bigarren adibidean, ildo hori hartuta, lehenengo eta behin, edukiontzia `Pertsona` motako objektuz beteko dugu. Edukiontzia eraikiz gero, aplikazio informatikoak 25 urte baino gehiagoko pertsonak bilatu behar ditu edukiontzian eta bukatzeko, edukiontziaren edukia irteera estandarretik aterako du. Hori guztia egikaritzeko eta JGL liburutegiari esker, nahikoa izango zaigu lau lerro programatzea programa nagusian. Jakina, lau lerro horiek ondo exekutatzeko, funtzio eta predikatu-objektuak antolatu beharko dira. Hona hemen, aipatutako betekizunak aztertuta, atera zaigun aplikazio informatikoa:

37. Edukiontzien eraiketa eta kudeaketa, eta algoritmoen exekuzioa.

```
import com.objectspace.jgl.*;
import com.objectspace.jgl.algorithms.*;
import com.objectspace.jgl.predicates.*;

public class Adibidea
  {
  public static void main( String[] args )
    {
    DList list = new DList();
    list.add( new Pertsona("Xabier", 30) );
    list.add( new Pertsona("Lorea", 15) );
    list.add( new Pertsona("Amaia", 40) );
    list.add( new Pertsona("Peru", 55) );
    list.add( new Pertsona("Nerea", 20) );

    UnaryPredicate predikatua = new BindSecondPredicate( new
Handiago(),new Integer(25) );
    int n = Counting.countIf( list, predikatua );
    Applying.forEach(list, new Idatzi());
    System.out.println( "25 urte baino gehiagoko pertsonak zerrendan
= " + n );
    }}

class Pertsona {
   private String izena;
   private int adina;

   Pertsona(String izena,int adina) {
      this.izena = izena;
      this.adina = adina;
   }
   public String izena() {
      return izena;
   }
   public int adina() {
      return adina;
   }
}
 class Handiago implements BinaryPredicate
  {
public boolean execute( Object objektu1, Object objektu2 )
    {
        Pertsona per= (Pertsona)objektu1;
        int bat= per.adina();
        Integer bi= (Integer)objektu2;
        int bi1=bi.intValue();
        return (bat>bi1);
    }}
  class Idatzi implements UnaryFunction {
    public Object execute(Object objektu)
```

```
    {  Pertsona per= (Pertsona)objektu;

 System.out.println("IDATZI BALIOAK-> Izena:  "+per.izena()+"
Adina: "+per.adina());

    return null;}
  }
```

Irteera estandarra:

```
IDATZI BALIOAK-> Izena:  Xabier  Adina: 30
IDATZI BALIOAK-> Izena:  Lorea  Adina: 15
IDATZI BALIOAK-> Izena:  Amaia  Adina: 40
IDATZI BALIOAK-> Izena:  Peru  Adina: 55
IDATZI BALIOAK-> Izena:  Nerea  Adina: 20
25 urte baino gehiagoko pertsonak zerrendan = 3
```

Aplikazioak `countIf()` eta `forEach()` algoritmoak erabiliko ditu nahi dugun funtzionalitatea eskuratzeko. Lehenengo algoritmoak zenbatuko du zenbat pertsona dauden 25 urte baino gehiagokoak. Horretarako, predikatu unario bat antolatu behar da ezaugarri txiki batekin: edukiontziaren elementu bakoitzaren, hau da, pertsona bakoitzaren adina, 25 zenbaki osoarekin parekatu behar dugu. Hori adierazteko tramankulu txiki bat antolatu behar dugu; lehenengo eta behin, predikatu bitar bat (`Handiago`) programatu behar dugu, eta bigarrenez, predikatu horren bigarren parametroa finkatu behar dugu prozesu guztirako. Hori egiteko, `BindSecondPredicate`[38] motako predikatu-objektu bat eraiki behar da:

```
UnaryPredicate predikatua = new BindSecondPredicate(
   new Handiago(), new Integer(25) );
```

38. `BindSeconPredicate`-rekin predikatu bitarraren bigarren argumentua finkatu egiten dugu. Lehen argumentua finkatu nahiko bagenu, `BindFirstPredicate` motako predikatu-objektu bat eraiki beharko genuke.

Predikatu-objektuen ordez funtzio-objektuak erabili behar baditugu argumentuak finkatzeko, `BindFisrt` eta `BindSecond` motako funtzio-objektuak eraiki behar dira.

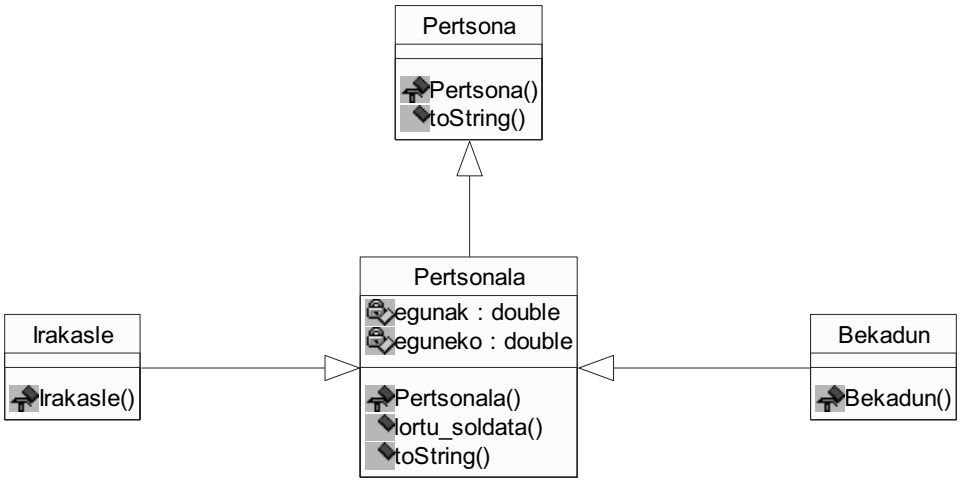
Adibidez, edukiontzi batean zirkunferentzia desberdinen diametroak baditugu, honako agindu sorta honek zirkunferentzia bakoitzaren luzera kalkulatuko luke:

```
// Copyright(c) 2002 Recursion Software, Inc.
import com.recursionsw.jgl.UnaryFunction;
import com.recursionsw.jgl.adapters.*;
import com.recursionsw.jgl.algorithms.Transforming;
import com.recursionsw.jgl.functions.*;
 // Use function binders to create a higher-order
// UnaryFunction from a BinaryFunction and a value.

public class Functions7
{
    public static void main(String[] args)
    {
```

Predikatu unarioa eraikiz gero, `countIf()` metodoari pasatzen diogu parametro gisa, algoritmoak `list` edukiontzia ebalua dezan eta zenbatu zenbat pertsona dauden 25 urte baino gehiagokoak. Hori eginda, `forEach()` metodoak edukiontziaren elementu bakoitzari `Idatzi` funtzio unarioa aplikatzen dio pertsonaren informazio pertsonala (izena eta adina) eskuratu eta irteera estandarretik ateratzeko.

Adibideen arlo hau bukatzeko, azken aplikazio bat garatuko dugu funtzio eta predikatu-objektuen azalpena txukun ixteko. Horretarako, JGL liburutegia erabilita "*datu-base*" txiki bat kudeatuko dugu. Datu-basea, JGLren edukiontzi batean kokatuko dugu eta predikatu-objektuak erabilita "*SQL aginduak*" eraikiko ditugu datu-basearen "*tuplak*" aztertzeko. Datu-basearen objektuek honako klase-hierarkia honi jarraituko diote:



3.18. irudia. Aplikazioaren Klase Diagrama.

```
                double[] diameters = { 3, 1, 5 };
                DoubleArray diameterArray = new DoubleArray(diameters);
                System.out.println( "diameter values = " + diameterArray );

   UnaryFunction calcCircumference = new BindFirst( new
TimesNumber(Double.class), new Double(Math.PI) );

        DoubleBuffer circumferences = new DoubleBuffer();
        Transforming.transform(diameterArray, circumferences,
calcCircumference);
        System.out.println( "circumference values = " + circumferences );  }}
```

Bestalde, klase-hierarkiaren iturburu-kodea honako hau izango da:

```
class Pertsona {

   String izena;
   String deitura;

   Pertsona(String deitura) {
      this.deitura = deitura;
   }

   public String toString() {
      return deitura;
   }}

class Pertsonala extends Pertsona {

   private double egunak;   //Lan egindako egunak
   private double eguneko;  //Eguneko soldata
   String hiri;

   Pertsonala(String deitura, double egunak, double eguneko, String
hiri) {
      super(deitura);
      this.egunak = egunak;
      this.eguneko = eguneko;
      this.hiri = hiri;
   }

public double lortu_soldata() {
      double guztira;
      guztira = egunak * eguneko;
      return guztira;
 }

   public String toString() {
      return super.toString() + " " + lortu_soldata() + " zzz";
   }}

class Bekadun extends Pertsonala {

   String proiektu;

Bekadun(String deitura,double egunak,double eguneko,String hiri,
String proiektu) {
      super(deitura, egunak, eguneko, hiri);
      this.proiektu= proiektu;
   }}
```

```
class Irakasle extends Pertsonala {
      String saila;
Irakasle(String deitura,double egunak,double eguneko,String
hiri,String saila ) {
      super(deitura, egunak, eguneko, hiri);
      this.saila= saila;
   } }
```

Datu-basearen tuplak eskuratzeko orduan, JGLren `select()` metodoak egikaritzen duen algoritmoa erabiliko dugu:

```
Filtering.select(Container, UnaryPredicate)
```

Behin algoritmoa egikarituta, emaitzak, edukiontzi berri batean gordeko ditugu. Bukatzeko, emaitzak ikuskatzeko, `forEach()` metodoak egikaritzen duen algoritmoa egikarituko dugu honako formatu honekin:

```
Applying.forEach(Container, UnaryFunction)
```

SQL agindu baten benetako antza izan dezan `select()` metodoaren `UnaryPredicate` motako predikatu unario parametroak, honako `UnaryPredicate` motako predikatu unario berri hau erabiliko dugu:

```
UnaryAnd(UnaryPredicate, UnaryPredicate)
```
[39]

Predikatu horrekin, *eta* logikoa egikaritzea lortzen dugu. Predikatuak `true` balio boolearra itzuliko du bere `UnaryPredicate` biek `true` itzultzen badute aurretik. Azpiegitura horrekin guztiarekin galdera konplexuak antola daitezke ahalegin handirik gabe. Hona hemen aplikazioa ondo programatu dugula baieztatzeko erabiliko dugun galdera:

"Bilatu 150.000 baino gehiago irabazten duten Lezoko irakasle guztiak"

Betekizunen azterketa burututa eta kontuan izanda nolakoa den datu-basearen egitura hierarkikoa (**3.18. irudia**), honako iturburu-kodea erabiliko dugu aplikazioa ondo egikaritzen dela probatzeko:

```
import com.objectspace.jgl.*;
import com.objectspace.jgl.algorithms.*;
import com.objectspace.jgl.functions.*;
import com.objectspace.jgl.predicates.*;
import java.util.Enumeration;

public class Nagusi {
```

39. Galderen semantika hedatu nahi badugu. `UnaryNot`, `UnaryOr` edo `UnaryTern` predikatuak ere erabil daitezke.

```
public static void main (String[] args) {

   double muga_soldata = Double.valueOf("150000").doubleValue();
   String hiria = new String("Lezo");

   System.out.println("Aztertu nahi ditugun balioak : ");
   System.out.println("Soldata : "+muga_soldata+"  Hiria : "+hiria);

   Container datuak = new Array();

   datuak.add( new Bekadun("Leizarraga",20.0, 1300.0, "Aduna","EHU") );
   datuak.add( new Bekadun("Lizarralde",25.0, 1200.0,
"Lekeitio","DIPUTAZIO") );
   datuak.add( new Bekadun("Goitia", 20.0, 1300.0, "Bilbo","EHU") );
   datuak.add( new Bekadun("Zubiria",120.0, 400.0, "Bera","EJ") );

  datuak.add( new Irakasle("Fernandez",120.0, 3000.0, "Lezo","LSI") );
   datuak.add( new Irakasle("Zelaia",50.0, 4000.0, "Lezo","KAT") );
   datuak.add( new Irakasle("Gonzalez",12.0, 5400.0, "Bera","LSI") );

     UnaryPredicate irakasleak  = new InstantziI(Irakasle.class);
     UnaryPredicate soldatak    = new Soldata_mugak(muga_soldata);
     UnaryPredicate hiritar     = new hiribateko(hiria);

// predikatu konplexua sortzen "eta"  logiko erakoa, predikatu
// arruntak erabiliz
UnaryPredicate predikatu1 =
      new UnaryAnd(hiritar,soldatak);

// predikatu konplexua sortzen "eta"  logiko erakoa, predikatu
// arruntak erabiliz
   UnaryPredicate Irakasleen_Soldatak =
      new UnaryAnd(irakasleak, predikatu1);

      Container emaitzak =
      Filtering.select(datuak, Irakasleen_Soldatak);

//Applying.forEach(emaitzak, new Idatzi() ); Ebazpen posible bat

      // Beste ebazpen posible bat -Enumeration- erabiliz
      Enumeration iteratzaile = emaitzak.elements();
         while(iteratzaile.hasMoreElements())
                  System.out.println(iteratzaile.nextElement());
} // main()-en bukaera

} // Nagusi-ren bukaera


class Soldata_mugak implements UnaryPredicate {
```

```
   protected double muga;

   Soldata_mugak(double muga) {
      this.muga = muga;
   }

   public boolean execute(Object o) {
      Pertsonala Pertsonal1 = (Pertsonala) o;
      return (Pertsonal1.lortu_soldata() > this.muga );
   }
}


class hiribateko implements UnaryPredicate {

   protected String hiria;

   hiribateko(String hiria) {
      this.hiria = hiria;
   }

   public boolean execute(Object o) {
      Pertsonala Pertsonal1 = (Pertsonala) o;
      return ( (Pertsonal1.hiri).equals(this.hiria) );
   }
}


class InstantziI implements UnaryPredicate {

protected Class instantziClass;

public InstantziI(Class instantziClass) {
   this.instantziClass = instantziClass;
}

public boolean execute(Object ObjektuA) {
   return instantziClass.isInstance(ObjektuA);
}

} // InstantziI-ren bukaera


class Idatzi implements UnaryFunction {

public Object execute(Object ObjektuA) {
       System.out.println(ObjektuA.toString());
       return null;
}

} // Idatzi-ren bukaera
```

Aplikazioak, lehenengo eta behin, datu-basearen funtzioa beteko duen

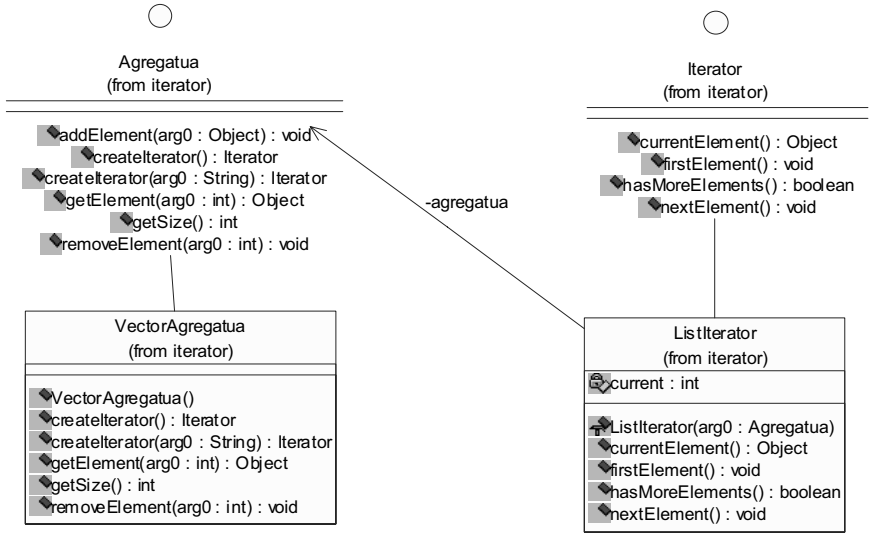
edukiontzia eraikitzen du. Hori eginda, beharrezkoak zaizkigun predikatu unarioak eraikitzen dira eta `UnaryAnd` predikatu-objektua erabilita, eraiki berriak izan diren predikatu unarioak kateatu egiten dira. Bukatzeko, `select()` eta `forEach()` metodoak egikaritzen dira, datu-basetik tupla egokiak eskuratu eta irteera estandarrean ikuskatzeko.

3.9.2. Iteratzaileak

Programatzaile bat, Java estandarrean, bilduma motako objektuekin lanean ari denean edo JGL liburutegiaren edukiontzi batekin, lehenago edo geroago mugitu nahi izango du bilduman edota edukiontzian barna, bertako elementuak atzitu eta nahi izanez gero eguneratzeko.

Java estandarrean `Collection` interfazeak `iterator()` metodoa du erazagututa, aipatutako lana egiteko. `iterator()` metodoak `Iterator` motako objektu bat itzultzen du. Objektu horrek, `hasNext()`, `next()` eta `remove()` metodoak erabiliko ditu bilduman zehar mugitzeko, elementuak eskuratuz (`next()`) eta ezabatuz (`remove()`).

Bai Java estandarraren iteratzaileen diseinatzaileak bai JGL liburutegiarenak Iterator (**3.19. irudia**) patroi ezagunean oinarritu ziren iteratzaileak programatzeko. Iterator patroiak interfaze bat eta zenbait metodo erazagutzen ditu. Metodo horiek bilduma (`Agregatua`) bateko objektuetara sekuentzialki atzitzeko erabiliko dira, hau da, `Agregatua` interfazea inplementatzen duen edukiontzira. Deigarriena da ez dugula datuen barne-adierazpidea ezagutu behar bilduman barna mugitzeko.



3.19. irudia. Iterator patroiaren Klase Diagrama.

Iteratzaile batek `Collections` APIan betetzen duen funtzionalitatea, tradizionalki, `Enumeration`[40] motako objektu batek betetzen zuen Java estandarrean, lortu behar zen funtzionalitatea txikia zenean. Jakina da `Enumeration` motako objektu batek murrizketa batzuk dituela, hau da, enumerazioa gauzatzen denean, norabidearen zentzuak nahitaez aurrerakoa izan behar du eta edukiontziaren elementuak ezin dira aldatu, atzitu (read-only) bakarrik baizik.

Beraz, iteratzaileen erabilera arrunta denean, askotan nahikoa izango zaigu `Enumeration` motako objektu bat eraikitzea. Programak erabilera aurreratua eskatzen badu, Java estandarraren iteratzaileen edo JGL liburutegiak eskaintzen dituenen artean egin beharko du aukera programatzaileak.

Antola dezagun aplikazio informatiko txiki bat `Enumeration` motako objektuak tratatzeko. Horretarako, JGL liburutegiaren `Array` motako edukiontzi bat erabiliko dugu `Enumeration` motako objektu bat sortzeko, eta Java estandarraren `Vector` motako objektu bat, beste bigarren `Enumeration` motako objektu bat eraikitzeko. Bai bata eta bai bestea eraikitzeko `elements()` metodoa egikaritzen dugu.

`Enumeration` motako objektuak eraikita, iteratzaileak prest egongo dira atzitu eta bertako informazioa eskuratzeko. Horretarako, `hasMoreElements()` eta `nextElement()` metodoak erabiliko ditugu. Lehenengoak, `true` balio boolearra itzuliko du iteratzaileak elementu gehiago baditu eta bigarrenak, ostera, iteratzailearen hurrengo elementua eskuratzen digu. Aipatu dugun guztia iturburukodera itzuliko bagenu, honako aplikazio informatikoa gelditzuko litzaiguke:

```
import com.objectspace.jgl.*;
import java.util.Enumeration;
import java.util.Vector;

public class Iteratzaile1
  {
  public static void main( String[] args )
    {
    Array array = new Array();
      array.add( "Linux" );
      array.add( "Microsoft" );
      array.add( "IBM" );
      array.add( "Oracle" );
      array.add( "Sun" );
      array.add( "Ibermatica" );
      array.add( "WebLogic" );

    Vector bektore = new Vector();
      bektore.add( "Linux" );
```

40. Gogoratu irakurle, `Enumeration` interfazea inplementatzen duen edukiontziak `iterator()` metodoaren ordez `elements()` metodoa inplementatzen duela `Enumeration` motako objektu bat eraikitzeko.

```
    bektore.add( "Microsoft" );
    bektore.add( "IBM" );
    bektore.add( "Oracle" );
    bektore.add( "Sun" );
    bektore.add( "Ibermatica" );
    bektore.add( "WebLogic" );

  Enumeration iterator = array.elements();
  while ( iterator.hasMoreElements() )
    System.out.println( iterator.nextElement() );
    System.out.println( );
    System.out.println( );
  Enumeration iteratzailea = bektore.elements();
  while ( iteratzailea.hasMoreElements() )
    System.out.println( iteratzailea.nextElement() );
  }}
```

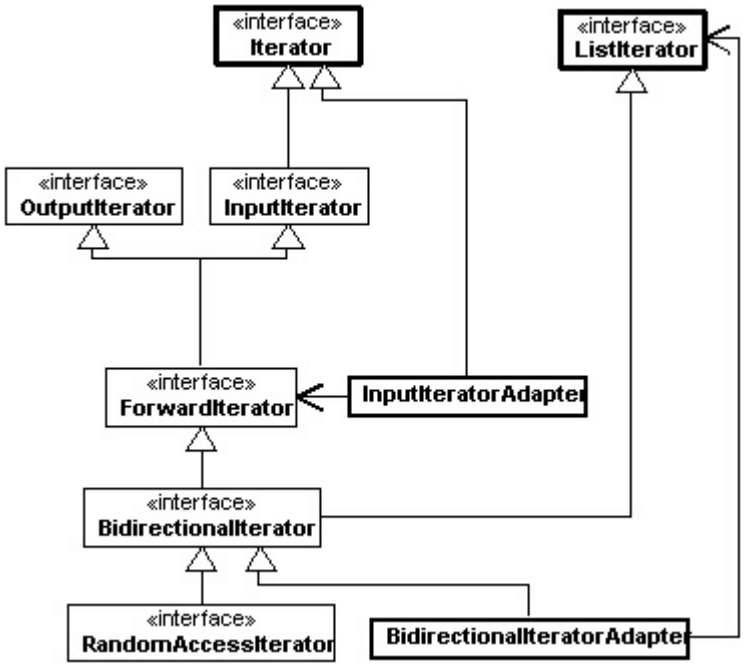
Ur handiagotako enbatei aurre egin behar badie, programatzaileari ondo etorriko zaio JGL liburutegiak eskaintzen dizkion iteratzaileak ezagutzea eta erabiltzea. Liburutegiak iteratzaile mota desberdinak eskaintzen ditu edukiontzien elementuak kudeatzeko. Batzuetan, nahikoa izango zaigu iteratzaile arrunt bat erabiltzea, zeinak aurrerako norabidea erabiltzen duen datuak atzitzeko bakarrik; beste batzuetan, ostera, atzera eta aurrera mugitzea interesatuko zaigu edukiontzian, elementuak atzituz eta berriak txertatuz, malgutasun osoa eskatuta iteratzaileari.

Egoera horiek guztiak tratatzeko, liburutegiak bost iteratzaile mota eskaintzen ditu egoera bakoitza kudeatzeko. **3.20. taula**k iteratzaile bakoitzaren deskripzio txiki bat ematen du gauzak argitzen joateko. Kontuan hartu behar da halaber, bateragarritasuna bultzatzeko Java estandarraren `java.util.Collection`-ekin, JGLren `Container` interfazea inplementatzen duen edozein edukiontzik `iterator()` metodoa ere izango duela, iteratzaile egokia itzultzeko dei egiten zaionean. Dakigun legez, Java estandarraren `Collection` interfazea inplementatzen duen edukiontziak gauza bera egiten du. Ildo beretik, liburutegiaren `Sequence` interfazea inplementatzen duen edukiontziak `ListIterator` interfazearen metodoak inplementatuko ditu. Esandako guztia klase-hierarkia (**3.21. irudia**) egoki bat erabilita inplementatzen da espezializazio-estrategia bat aplikatuz.

Iterator Interfazea	Deskripzioa
`InputIterator`	Elementu bakarra irakurtzen du aldiro aurrerako norabidean
`OutputIterator`	Elementu bakarra idazten du aldiro aurrerako norabidean
`ForwardIterator`	Elementu bakarra irakurtzen/idazten du aldiro aurrerako norabidean
`BidirectionalIterator`	Elementu bakarra irakurtzen/idazten du aldiro aurrerako eta atzerako norabidean
`RandomAccessIterator`	Elementu bakarra irakurtzen/idazten du aldiro aurrerako eta atzerako norabidean eta gainera, jauziak egitea ere onartzen du.

3.20. taula. JGL liburutegiaren iteratzaile motak.

Jakin badakigu JGL liburutegiak hainbat edukiontzi eskaintzen diola programatzaileari, bestalde, aurreko paragrafoetan ikusi dugu iteratzaile kopurua ere ez dela makala, beraz, programatzaileak argi izan behar du edukiontzi bakoitzari zein iteratzaile esleitzen zaion eta iteratzaile horrek zein interfaze inplementatzen duen. **3.22. taula**k edukiontzi bakoitzaren esleipena eta inplementatzen den interfazea jasotzen ditu alfabetikoki ordenatuta.



3.21. irudia. Iterator moten klase-hierarkia.

Edukiontzia	Asoziatutako iteratzailea	Iterator Interfazea
`Array`	`ArrayIterator`	`RandomAccessIterator`
`BooleanArray` `BooleanBuffer`	`BooleanIterator`	`RandomAccessIterator`
`ByteArray` `ByteBuffer`	`ByteIterator`	`RandomAccessIterator`
`CharArray` `CharBuffer`	`CharIterator`	`RandomAccessIterator`
`Deque`	`DequeIterator`	`RandomAccessIterator`
`DoubleArray` `DoubleBuffer`	`DoubleIterator`	`RandomAccessIterator`
`FloatArray` `FloatBuffer`	`FloatIterator`	`RandomAccessIterator`
`IntArray` `IntBuffer`	`IntIterator`	`RandomAccessIterator`
`DList`	`DListIterator`	`BidirectionalIterator`
`HashMap`	`HashMapIterator`	`ForwardIterator`
`ObjectArray`	`ObjectIterator`	`RandomAccessIterator`
`OrderedMap`	`OrderedMapIterator`	`BidirectionalIterator`
`OrderedSet`	`OrderedSetIterator`	`BidirectionalIterator`
`PriorityQueue`	`ArrayIterator`	`RandomAccessIterator`
`Queue`	Erabiltzen den `Sequence` motako sekuentziaren arabera	Erabiltzen den `Sequence` motako sekuentziaren arabera
`HashSet`	`HashSetIterator`	`ForwardIterator`
`ShortArray` `ShortBuffer`	`ShortIterator`	`RandomAccessIterator`
`SList`	`SListIterator`	`ForwardIterator`
`Stack`	Erabiltzen den `Sequence` motako sekuentziaren arabera	Erabiltzen den `Sequence` motako sekuentziaren arabera
`VectorArray`	`VectorIterator`	`RandomAccessIterator`

3.22. taula. Edukiontzien eta iteratzaileen arteko erlazioak.

Iteratzaileen azalpen hau ixteko adibide praktiko bat garatuko dugu ikusteko iteratzaileak nola konbinatzen diren edukiontzi eta algoritmoekin. Horretarako, lehenengo eta behin, `Array` motako edukiontzi bat eraikiko dugu eta balio batzuekin hasieratu. Hori eginda, adibidearen lehen zatian, iteratzaileak eta algoritmoak jarriko ditugu jokoan.

Aurreko orrietan ikusi dugu programatzaileak algoritmo baten portaera estandarra aldatu nahi badu, erraz alda dezakeela predikatu eta funtzio-objektuak erabilita. Egoera horietan, predikatu eta funtzio-objektuen portaera edukiontzi osoari aplikatzen zitzaion, batzuetan halaber, portaera hori edukiontziaren zati zehatz bati aplikatzea interesatzen da. Horrelakoetarako, oso ondo datorkigu iteratzaileen funtzionalitatea algoritmoaren portaera edukiontziaren elementu zehatz batzuetara murrizteko. Gure adibidearen lehen zatiak hori berori egingo du, iteratzaile baten bidez edukiontziaren laugarren elementuan kokatuko gara eta hortik, edukiontziaren amaiera arte `replace()` metodoa egikarituko dugu. Metodoak “Ibermatica” balioa aurkitzen duen bakoitzean, “Euskomatika” balioa

txertatuko du jatorrizko balioa ordezkatuz. Hori guztia egiteko nahikoak izango zaizkigu honako lerro hauek:

```
// Iteratzailea eskuratu hirugarren elementuaren alboan kokatuta
      ArrayIterator last = array.begin();
      last.advance( 3 );
// Aldatu hirugarren elementutik aurrera <Ibermatica> aurkitzen
//  den toki guztietan balio hori eta jarri <Euskomatika>
      Replacing.replace( last, array.end(), new
String("Ibermatica"), new String("Euskomatika"));
```

Adibidearen bigarren zatian, iteratzaileez baliatuko gara begizta bat kontrolatzeko. Bigarren egoera honetan, ez dugu algoritmorik erabiliko edukiontzia aldatzeko eta horren ordez, `get()` eta `put()` metodoak aplikatuko ditugu; edukiontziaren hurrengo elementuan kokatzeko, `advance()` metodoa erabiliko dugu. Bigarren zati honetan, `get()` metodoa erabiliz eskuratzen dugun `String` motako balioa hizki larritan jarriko dugu `toUpperCase()` metodoa egikarituta. Itxuraldaketa gauzatuz gero, edukiontzian txertatuko dugu balio berria `put()` metodoa egikarituta. Hona hemen esandakoa egiteko erabiliko dugun iturburu-kodea:

```
// Iteratzailea eskuratu lehen elementuan kokatuta
    ForwardIterator iterator = container.start();
// Iteratzailea eskuratu azken elementuaren alboan kokatuta
    ForwardIterator end = container.finish();

// Edukiontziaren elementuak aztertu banan-banan
    while ( !iterator.equals( end ) )
      {
      String balioa = (String) iterator.get();
      // Eskuratutako elementua hizki larriz jarri
      iterator.put( balioa.toUpperCase() );
      iterator.advance(); // Hurrengo elementuan kokatu
```

Adibidearen hirugarren eta laugarren zatiak, helburu bera lortzeko estrategia bi izango dira. Batean, iteratzaile motak (`ReverseIterator`) eta bestean, `retreat()` metodoak lagunduko digute arazoa konpontzen. Inplementatu nahi dugun betekizuna ez da beste mundukoa: edukiontziaren balioak eskuratu baina atzetik aurrera.

`ReverseIterator` iteratzailea erabilita, betekizuna inplementatzeak ez du inongo zailtasunik, enumerazio arrunt gisa trata baitaiteke edukiontzia, nahi dugun emaitza lortzeko:

```
ReverseIterator iteratze = new ReverseIterator( array.end() );
    while ( iteratze.hasMoreElements() )
      { System.out.print( iteratze.nextElement() );
       System.out.print("  " );}
```

Iteratzaileekin gustura lan egiten badugu eta `ReverseIterator` iteratzaileari buruz berririk ez badugu, adibidearen bigarren zatian erabili dugun estrategia bera aplika dezakegu hemen ere; iteratzaileak erabili begizta bat kontrolatzeko eta begiztaren barnean `get()` metodoarekin edukiontziaren elementua eskuratu eta `retreat()` metodoarekin aurreko elementuan kokatu.

Zati guztiak iturburu-kode berean itsasten baditugu prozesu guztian edukiontzi bera erabilita, honako aplikazio informatiko hau geldituko zaigu:

```
import com.objectspace.jgl.*;
import com.objectspace.jgl.util.*;
import com.objectspace.jgl.algorithms.*;

public class Iteratzaile2
  {
  public static void main( String[] args )
    {
    Array array = new Array();
      array.add( "Linux" );
      array.add( "Microsoft" );
      array.add( "IBM" );
      array.add( "Oracle" );
      array.add( "Sun" );
      array.add( "Ibermatica" );
      array.add( "WebLogic" );

      System.out.println( "jatorrizko array-a = " + array );

      // Iteratzailea eskuratu hirugarren elementuaren alboan kokatuta
      ArrayIterator last = array.begin();
      last.advance( 3 );
      // Aldatu hirugarren elementutik aurrera <Ibermatica> aurkitzen
      //  den toki guztietan balio hori eta jarri <Euskomatika>
      Replacing.replace( last, array.end(), new String("Ibermatica"),
new String("Euskomatika"));
      System.out.println( "aldatutako array-a = " + array );
      System.out.println( );
      System.out.println( );

    System.out.println( "jatorrizko array-a = " + array );
    aldatu( array );
    System.out.println( "array-a eguneratuta= " + array );
    System.out.println( );
    System.out.println( );
        // Iteratzailea eskuratu azken elementuaren alboan kokatuta
    ArrayIterator iterator = array.end();
        // Iteratzailea eskuratu lehen elementuan kokatuta
    ArrayIterator begin = array.begin();
```

```
    while ( !iterator.equals( begin ) )
      {
      iterator.retreat();
      System.out.print( iterator.get());
      System.out.print("  " );
        }
      System.out.println( );
      System.out.println( );

      ReverseIterator iteratze = new ReverseIterator( array.end() );
    while ( iteratze.hasMoreElements() )
      { System.out.print( iteratze.nextElement() );
      System.out.print("  " );}
    }

  static void aldatu( Container container )
    {
       // Iteratzailea eskuratu lehen elementuan kokatuta
    ForwardIterator iterator = container.start();

    // Iteratzailea eskuratu azken elementuaren alboan kokatuta
    ForwardIterator end = container.finish();

    // Edukiontziaren elementuak aztertu banan-banan
    while ( !iterator.equals( end ) )
      {
      String balioa = (String) iterator.get();
      // Eskuratutako elementua hizki larriz jarri
      iterator.put( balioa.toUpperCase() );
      iterator.advance(); // Hurrengo elementuan kokatu
      }}}
```

Irteera estandarra:

```
jatorrizko array-a = Array( Linux, Microsoft, IBM, Oracle, Sun,
Ibermatica, WebLogic )
aldatutako array-a = Array( Linux, Microsoft, IBM, Oracle, Sun,
Euskomatika, WebLogic )

jatorrizko array-a = Array( Linux, Microsoft, IBM, Oracle, Sun,
Euskomatika, WebLogic )
array-a eguneratuta= Array( LINUX, MICROSOFT, IBM, ORACLE, Sun,
EUSKOMATIKA, WEBLOGIC )
```

3.10. LABURPENA

Aplikazio informatikoak garatzea beti izan da lan konplexua eta zaila. Lan horren erremina leundu eta errazteko, informatikariak, tresna ahaltsu bi izan ditu: *osagaien kapsulazioa* eta *abstrakzioa*. Programazio egituratuarekin, prozedurek eta funtzioek, goiti-beheiti, kapsulazio-lan apala burutzen zuten. Geroago,

programazio modularrak gauzak dezente hobetu zituen objektuetara zuzendutako programazioari bideak irekita.

Programazio-paradigma horretan domeinuaren objektu fisikoak ebazpenaren testuingurura lekualda daitezke arazo handirik gabe. Ebazpenaren unibertso berrian, objektu fisikoak, klaseetan antolatzen dira objektu horien egoera eta portaera egituratzeko. Horrenbestez, informatikariak ondo baino hobeto jakin behar du klase horiek nola antolatzen diren eta programatzen diren.

Testuinguru horretan liburutegi generikoek (JGL) egundoko laguntza ematen diote programatzaileari eguneroko jardunean iturburu-kodea sinplifikatu eta murriztu egiten baita abstrakzio maila igota.

3.11. OINARRIZKO BIBLIOGRAFIA

Arnold, K.; Gosling, J. eta Holmes, D. (2000): *The Java™ Programming Language*, Addison Wesley.

Arnow, D. eta Weiss, G. (2000): *Introducción a la programación con Java. Un enfoque orientado a objetos*, Addison Wesley.

Bishop, J. (1999): *Java. Fundamentos de programación*, Addison Wesley.

Campione, M.; Walrath, K.; Zakhour, S. eta Bodoff, S. (2005): *The Java™ Tutorial: A Short Course on the Basics*, Addison Wesley.

Ceballos, F. J. (2000): *Java 2. Curso de programación*, Rama.

Deitel, H. eta Deitel, P. (2004): *Cómo Programar en Java*, Prentice Hall.

Li, L. (1998): *Java. Data Structures and Programming*, Springer-Verlag.

Lima Díaz, F. (2000): *Manual avanzado de Java 2*, Anaya Multimedia.

Meyer, B. (1998): *Construcción de Software Orientado a Objetos*, Prentice Hall.

Musser, D. eta Saini, A. (1995): *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*, Addison Wesley.

Riel, A. (1996): *Object-Oriented Design Heuristics*, Addison Wesley.

3.12. HELBIDE INTERESGARRIAK

Zer dago bertan	Web gunea
Javaren teknologia	`http://java.sun.com/`
Java 2 plataforma Edizio estandarra	`http://java.sun.com/j2se/`
Thinking in Java (Doaneko liburuak `pdf` formatuan)	`http://mindview.net/Books`
JGL Tresna	`http://www.recursionsw.com/jgl.htm`
Informazioa Javaren teknologiari buruz	`http://www.javahispano.org/`
Informazioa Javaren teknologiari buruz	`http://java.programacion.com/`
Javaren teknologiaren web ataria	`http://www.theserverside.com/`
Tresna aurreratua aplikazioak garatzeko	`http://www.eclipse.org/`
Tresna aurreratua aplikazioak garatzeko	`http://www.netbeans.org/`
IntelliJ™ IDEA	`http://www.jetbrains.com/idea/`

4. Proba bateratzaileak: JUnit

4.1. PROBA BATERATZAILEAK

Erakunde informatiko askotan, programatzaileak aplikazio informatikoaren diseinua programa bihurtzen duenean eta lehenengo emaitzak bistaratu, presio izugarria sortzen da aplikazio hori agudo merkaturatzeko. Merkaturatze-prozesuan programarekin arazoak sortzen badira, programatzaileak, gehienetan azkar eta korrika suhiltzaile-lana beteta, identifikatzen ditu eta zuzentzen saiatzen da. Denboraren eta baliabideen arabera, hiru estrategia mota erabiliko ditu gauzak beren onera ekartzeko. Denbora urria, presioa handia eta baliabideak kaxkarrak badira, informatikarien artean hain ezaguna den `println`-aren estrategia ezaguna erabiliko du arazoa konpontzeko. Programatzaileak iturburu-kodea `print` aginduz josiko du, batzuetan, aldagaien balioak bistaratzeko eta besteetan, ostera, exekuzio-fluxuaren nondik norakoa aztertzeko. Hala, poliki-poliki, arazoa inguratuz eta mugatuz joango da eta noizbait programa martxan jarriko du berriro. Edozelan ere, nahiz eta programa martxan jarri beti geldituko zaio harra konpondutakoak ez ote duen konpondu behar ez zen zerbait ukitu eta aldatu. Behin sua piztuta, nork jakin hurrengoko sutea nork eragin duen?

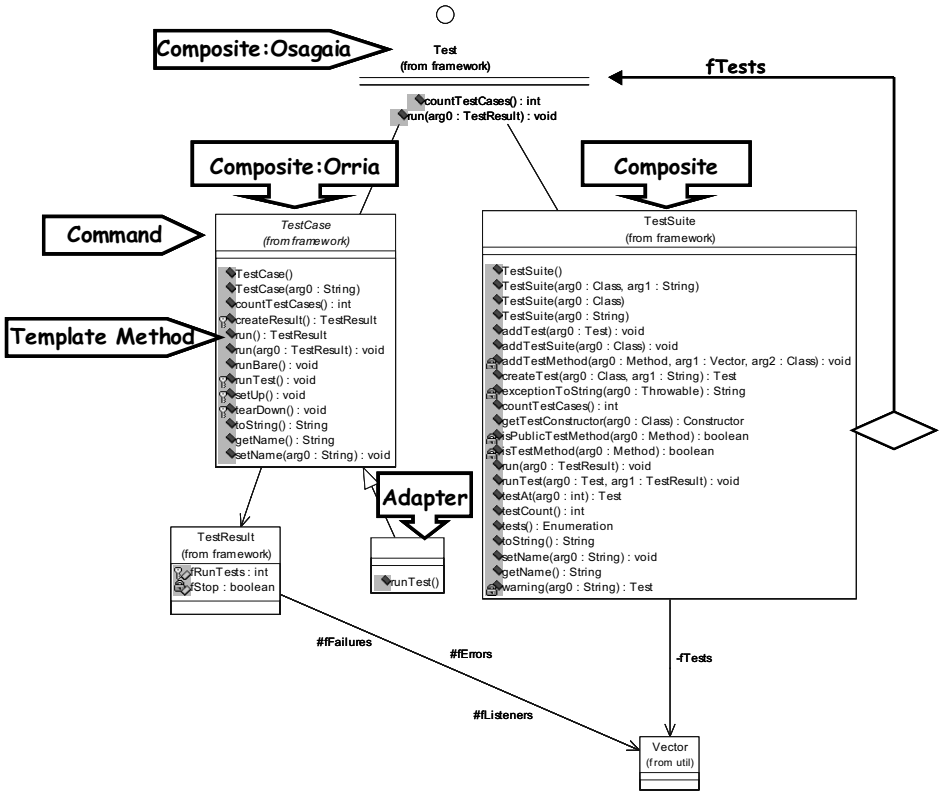
Arazoak zertxobait leunduko dira, programatzaileak tresna aurreratu batekin garatu badu programa. Tresnak ziur asko ingurune grafiko bat eskainiko dio programaren exekuzioa bistaratzeko, eta araztaile bat, exekuzio-fluxuan gertatzen diren akatsak detektatu eta konpontzeko. Betiere, programatzailearen trebetasunaren garrantzia kritikoa izango da bigarren estrategia honetan. Araztailea ona izan daiteke arazoa detektatzen baina beti geldituko da programatzailearen eskuetan arazoaren zergatikoa interpretatzea eta konpontzea. Estrategia honetan arazoa ez da era sistematikoan eta metodikoan konpontzen eta etorkizunean, arazoa izan den inguruan, programatzaileak iturburu-kode gehiago programatu behar badu eta berriro arazoak sortzen badira, ez da jakingo aurreko arazoaren eraginaren ondorioa den ala ez.

Programatzaile batek, jakin badaki idazten ari den iturburu-kodea piteanpitean egiaztatu egin behar dela eta lan hori komeni dela zati txikerretan eta aldiko egitea. Tamalez, egiaztapen hori aipatutako presiopean gauzatzen da eta zenbat eta presio handiagoa sumatu, orduan eta, egiaztapen gutxiago egiten da. Egiaztapena gutxitzen bada iturburu-kodearen kalitatea jaitsi egingo da programatzailearen

produktibitatearekin batera. Horrek, presio handiagoa ekarriko du programatzailearentzat, sorgin-gurpil ero bat eratuz.

Epe luzera, dinamika horrek programatzailea kiskaltzera darama. Ostera, erakundeetan ohiturak sustraituta daudenez, aldaketak kontrolatuta eta kudeatuta egin behar dira, erakundea ohiturazalea baita eguneroko jardunean eta kontserbadorea jarreretan.

Aipatutako testuinguruan, programatzailearen produktibitatea hobetu nahi bada presioa igo barik, egiaztapen arloa automatizatu egin behar da ahal denik eta gehien, tresna informatiko aurreratuak erabilita. Hala, programatzaileak aplikazioaren iturburu-kodean ipiniko du arreta nagusia eta tresnari lagako dio egiaztapen-lana. Bada, horrelako tresna bati bi baldintza eskatuko dizkiogu: erraza izan dadila txertatzen aplikazioaren iturburu-kodean, eta egiaztapen-lana bere ardura bakarra izan dadila. JUnit tresnak (`http://www.junit.org`) baldintza biak betetzen ditu.



4.1. irudia. JUnit tresnaren arkitektura orokorra.

4.1. irudian agertzen den legez, JUnit tresnaren arkitektura orokorra objektuetan egituratzen da aplikazioak egikarituko dituen testak antolatu eta emaitzak

gordetzeko. Diseinu txukuna eratuz, tresnak diseinu-patroi ezagunak inplementatzen ditu: `Command` eta `Composite` lan orokorra egituratzeko eta `Adapter` eta `Template Method` zehaztasunak kudeatzeko.

`Command` diseinu-patroiarekin, bezeroaren eskaera kapsulatu egiten da objektu bat bezala eta objektu horri interfaze publiko eta ezagun bat itsasten zaio. Hala, bezeroak eskaera gauzatzen duenean, egikaritu behar den ekintza banatuta gelditzen da.

`Composite` diseinu-patroiarekin, objektuak zuhaitz-egituran antolatzen dira zati-oso (part-whole) erlazio hierarkikoak islatzeko. Horretarako, oinarrizko (`Composite:Orria`) osagaiak eta osagai konplexuak (`Composite`) eraikitzen dira. Diseinu-patroiaren bezeroak zuhaitz-egitura eratzeko eta kudeatzeko erabiliko ditu osagai konplexuak. Eratuz gero, ez du arazorik izango osagai bati inplementatu dituen eragiketak eskatzeko. Gauzak ondo diseinatu badira, zuhaitzaren osagai bakoitzak bere eragiketa propioa egikarituko du.

Gurera etorrita has gaitezen `TestCase` klasea aztertzen:

```
package junit.framework;

import java.lang.reflect.*;

/**
 * A test case defines the fixture to run multiple tests. To define a
* test case<br>
 * 1) implement a subclass of TestCase<br>
 * 2) define instance variables that store the state of the
fixture<br>
 * 3) initialize the fixture state by overriding
 *__<code>setUp</code><br>
 * 4) clean-up after a test by overriding <code>tearDown</code>.<br>
 * Each test runs in its own fixture so there
 * can be no side effects among test runs.
 * Here is an example:
 * <pre>
 * public class MathTest extends TestCase {
 *     protected double fValue1;
 *     protected double fValue2;
 *
 *    protected void setUp() {
 *         fValue1= 2.0;
 *         fValue2= 3.0;
 *     }
 * }
 * </pre>
 *
 * For each test implement a method which interacts
 * with the fixture. Verify the expected results with assertions
```

```
  * specified
 * by calling <code>assertTrue</code> with a boolean.
 * <pre>
 *    public void testAdd() {
 *        double result= fValue1 + fValue2;
 *        assertTrue(result == 5.0);
 *    }
 * </pre>
 * Once the methods are defined you can run them. The framework
*  supports
 * both a static type safe and more dynamic way to run a test.
 * In the static way you override the runTest method and define the
* method to
 * be invoked. A convenient way to do so is with an anonymous inner
 * class.
 * <pre>
 * TestCase test= new MathTest("add") {
 *        public void runTest() {
 *            testAdd();
 *        }
 * };
 * test.run();
 * </pre>
 * The dynamic way uses reflection to implement <code>runTest</code>.
* It dynamically finds
 * and invokes a method.
 * In this case the name of the test case has to correspond to the
 *   test method
 * to be run.
 * <pre>
 * TestCase= new MathTest("testAdd");
 * test.run();
 * </pre>
 * The tests to be run can be collected into a TestSuite. JUnit
 * provides
 * different <i>test runners</i> which can run a test suite and
 * collect the results.
 * A test runner either expects a static method <code>suite</code> as
* the entry
* point to get a test to run or it will extract the suite
 * automatically.
 * <pre>
 * public static Test suite() {
 *      suite.addTest(new MathTest("testAdd"));
 *      suite.addTest(new MathTest("testDivideByZero"));
 *      return suite;
 *  }
 * </pre>
 * @see TestResult
 * @see TestSuite
 */
```

```
public abstract class TestCase extends Assert implements Test {
       /**
        * the name of the test case
        */
       private String fName;

       /**
        * No-arg constructor to enable serialization. This method
*is not intended to be used by mere mortals without calling setName().
        */
       public TestCase() {
              fName= null;
       }
       /**
        * Constructs a test case with the given name.
        */
       public TestCase(String name) {
              fName= name;
       }
       /**
* Counts the number of test cases executed by run(TestResult result).
        */
       public int countTestCases() {
              return 1;
       }
       /**
        * Creates a default TestResult object
        *
        * @see TestResult
        */
       protected TestResult createResult() {
           return new TestResult();
       }
       /**
* A convenience method to run this test, collecting the results with a
        * default TestResult object.
        *
        * @see TestResult
        */
       public TestResult run() {
              TestResult result= createResult();
              run(result);
              return result;
       }
       /**
        * Runs the test case and collects the results in TestResult.
        */
       public void run(TestResult result) {
              result.run(this);
```

```
}
/**
 * Runs the bare test sequence.
 * @exception Throwable if any exception is thrown
 */
public void runBare() throws Throwable {
	setUp();
	try {
		runTest();
	}
	finally {
		tearDown();
	}
}
/**
 * Override to run the test and assert its state.
 * @exception Throwable if any exception is thrown
 */
protected void runTest() throws Throwable {
	assertNotNull(fName);
	Method runMethod= null;
	try {
		// use getMethod to get all public inherited
		// methods. getDeclaredMethods returns all
		// methods of this class but excludes the
		// inherited ones.
		runMethod= getClass().getMethod(fName, null);
	} catch (NoSuchMethodException e) {
		fail("Method \""+fName+"\" not found");
	}
	if (!Modifier.isPublic(runMethod.getModifiers())) {
		fail("Method \""+fName+"\" should be public");
	}

	try {
		runMethod.invoke(this, new Class[0]);
	}
	catch (InvocationTargetException e) {
		e.fillInStackTrace();
		throw e.getTargetException();
	}
	catch (IllegalAccessException e) {
		e.fillInStackTrace();
		throw e;
	}
}
/**
 * Sets up the fixture, for example, open a network connection.
 * This method is called before a test is executed.
 */
```

```
protected void setUp() throws Exception {
}
/**
 * Tears down the fixture, for example, close a network
 * connection.
 * This method is called after a test is executed.
 */
protected void tearDown() throws Exception {
}
/**
 * Returns a string representation of the test case
 */
public String toString() {
    return getName() + "(" + getClass().getName() + ")";
}
/**
 * Gets the name of a TestCase
 * @return returns a String
 */
public String getName() {
      return fName;
}
/**
 * Sets the name of a TestCase
 * @param name The name to set
 */
public void setName(String name) {
      fName= name;
}}
```

`TestCase` klaseak `Assert` klasea hedatzen du eta `Test` interfazea inplementatzen du. `Test` interfazea geroago aztertuko dugu eta behar-beharrezkoa izango zaigu `Composite` diseinu-patroia egituratzeko. Bestela, `TestCase` klase horretan, azpimarragarriena testa egikaritzeko erabiliko diren `run` metodoak dira, bereziki, `runBare()` metodoa, horrek egituratzen baitu `Template Method` diseinu-patroia. Patroiak, test bakoitzean, egikarituko den algoritmoaren eskeletoa biltzen du eragiketa bakarrean. Algoritmoak hiru zati izango ditu: (a) testa egikaritu baino lehen egin behar diren eragiketak jasoko dituen `setUp()` metodoa, (b) testa egikaritzeko behar den iturburu-kodea (`runTest()`) eta (c) testa egikarituta egin behar diren gainerako eragiketak antolatuko dituen `tearDown()` metodoa. Iturburu-kodean ikusten den legez, `setUp()` eta `tearDown()` metodoek inplementazio hutsa daukate eta domeinu bakoitzak bere nahietara egokituko ditu testa antolatzerakoan.

Orain arte aztertutakoarekin test bat nola egituratzen den azaldu dugu `Command` diseinu-patroia erabilita. Laburtuz, prozesuan bi multzo bereiz daitezke: (a) testaren aurretik (`setUp()`) eta atzetik(`tearDown()`) gauzatzen diren

eragiketak eta (b) testa bera egikaritzeko egiaztatzen diren asertzioak. Gehienetan, ordea, klase baten portaera orokorra aztertu nahi dugu eta horretarako test bat baino gehiago egikaritu behar dugu aldi bakoitzean. Test horiek guztiak multzokatzeko eta taldeka antolatzeko `Composite` diseinu-patroia erabiliko dugu. `TestSuite` klasea izango da patroiaren osagai nagusia, bera arduratuko baita testak antolatzen multzoka, `Vector` datu mota erabilita:

```
package junit.framework;

import java.util.Vector;
import java.util.Enumeration;
import java.io.PrintWriter;
import java.io.StringWriter;
import java.lang.reflect.*;
import java.lang.reflect.Constructor;

/**
 * A <code>TestSuite</code> is a <code>Composite</code> of Tests.
 * It runs a collection of test cases. Here is an example using
 * the dynamic test definition.
 * <pre>
 * TestSuite suite= new TestSuite();
 * suite.addTest(new MathTest("testAdd"));
 * suite.addTest(new MathTest("testDivideByZero"));
 * </pre>
 * Alternatively, a TestSuite can extract the tests to be run
 * automatically.
 * To do so you pass the class of your TestCase class to the
 * TestSuite constructor.
 * <pre>
 * TestSuite suite= new TestSuite(MathTest.class);
 * </pre>
 * This constructor creates a suite with all the methods
 * starting with "test" that take no arguments.
 *
 * @see Test
 */

public class TestSuite implements Test {

      private Vector fTests= new Vector(10);
      private String fName;

    /**
       * Constructs an empty TestSuite.
       */
      public TestSuite() {
      }
```

```
      /**
  * Constructs a TestSuite from the given class with the given name.
       * @see TestSuite#TestSuite(Class)
       */
      public TestSuite(Class theClass, String name) {
            this(theClass);
            setName(name);
      }

      /**
 * Constructs a TestSuite from the given class. Adds all the methods
       * starting with "test" as test cases to the suite.
 * Parts of this method was written at 2337 meters in the Hüffihütte,
       * Kanton Uri
       */
       public TestSuite(final Class theClass) {
            fName= theClass.getName();
            try {
          getTestConstructor(theClass); // Avoid generating
                                        // multiple error messages
            } catch (NoSuchMethodException e) {
                addTest(warning("Class "+theClass.getName()+" has no
public constructor TestCase(String name) or TestCase()"));
                  return;
            }

            if (!Modifier.isPublic(theClass.getModifiers())) {
      addTest(warning("Class "+theClass.getName()+" is not public"));
                  return;
            }

            Class superClass= theClass;
            Vector names= new Vector();
            while (Test.class.isAssignableFrom(superClass)) {
                  Method[] methods= superClass.getDeclaredMethods();
                  for (int i= 0; i < methods.length; i++) {
                        addTestMethod(methods[i], names, theClass);
                  }
                  superClass= superClass.getSuperclass();
            }
            if (fTests.size() == 0)
   addTest(warning("No tests found in "+theClass.getName()));
      }

      /**
       * Constructs an empty TestSuite.
       */
      public TestSuite(String name) {
            setName(name);
```

```
	}

	/**
	 * Adds a test to the suite.
	 */
	public void addTest(Test test) {
		fTests.addElement(test);
	}

	/**
	 * Adds the tests from the given class to the suite
	 */
	public void addTestSuite(Class testClass) {
		addTest(new TestSuite(testClass));
	}

	private void addTestMethod(Method m, Vector names, Class
theClass) {
			String name= m.getName();
			if (names.contains(name))
				return;
			if (! isPublicTestMethod(m)) {
				if (isTestMethod(m))
			addTest(warning("Test method isn't public: "+m.getName()));
				return;
			}
			names.addElement(name);
			addTest(createTest(theClass, name));
	}

	/**
	 * ...as the moon sets over the early morning Merlin, Oregon
	 * mountains, our intrepid adventurers type...
	 */
	static public Test createTest(Class theClass, String name) {
			Constructor constructor;
			try {
				constructor= getTestConstructor(theClass);
			} catch (NoSuchMethodException e) {
				return warning("Class "+theClass.getName()+" has no
public constructor TestCase(String name) or TestCase()");
			}
			Object test;
			try {
				if (constructor.getParameterTypes().length == 0) {
				     test= constructor.newInstance(new Object[0]);
					if (test instanceof TestCase)
						((TestCase) test).setName(name);
				} else {
					test= constructor.newInstance(new
```

```
Object[ ]{ name} );
                  }
            } catch (InstantiationException e) {
                  return(warning("Cannot instantiate test case:
"+name+" ("+exceptionToString(e)+")"));
            } catch (InvocationTargetException e) {
                  return(warning("Exception in constructor: "+name+"
("+exceptionToString(e.getTargetException())+")"));
            } catch (IllegalAccessException e) {
                  return(warning("Cannot access test case: "+name+"
("+exceptionToString(e)+")"));
            }
            return (Test) test;
      }

      /**
       * Converts the stack trace into a string
       */
      private static String exceptionToString(Throwable t) {
            StringWriter stringWriter= new StringWriter();
            PrintWriter writer= new PrintWriter(stringWriter);
            t.printStackTrace(writer);
            return stringWriter.toString();

      }

      /**
   * Counts the number of test cases that will be run by this test.
       */
      public int countTestCases() {
            int count= 0;
            for (Enumeration e= tests(); e.hasMoreElements(); ) {
                  Test test= (Test)e.nextElement();
                  count= count + test.countTestCases();
            }
            return count;
      }

      /**
       * Gets a constructor which takes a single String as
       * its argument or a no arg constructor.
       */
      public static Constructor getTestConstructor(Class theClass)
throws NoSuchMethodException {
            Class[ ] args= { String.class };
            try {
                  return theClass.getConstructor(args);
            } catch (NoSuchMethodException e) {
                  // fall through
            }
```

```
            return theClass.getConstructor(new Class[ 0] );
      }

      private boolean isPublicTestMethod(Method m) {
            return isTestMethod(m) &&
Modifier.isPublic(m.getModifiers());
       }

      private boolean isTestMethod(Method m) {
            String name= m.getName();
            Class[ ] parameters= m.getParameterTypes();
            Class returnType= m.getReturnType();
            return parameters.length == 0 && name.startsWith("test")
&& returnType.equals(Void.TYPE);
       }

      /**
       * Runs the tests and collects their result in a TestResult.
       */
      public void run(TestResult result) {
            for (Enumeration e= tests(); e.hasMoreElements(); ) {
                  if (result.shouldStop() )
                        break;
                  Test test= (Test)e.nextElement();
                  runTest(test, result);
            }
      }

      public void runTest(Test test, TestResult result) {
            test.run(result);
      }

      /**
       * Returns the test at the given index
       */
      public Test testAt(int index) {
            return (Test)fTests.elementAt(index);
      }

      /**
       * Returns the number of tests in this suite
       */
      public int testCount() {
            return fTests.size();
      }

      /**
       * Returns the tests as an enumeration
       */
      public Enumeration tests() {
```

```
            return fTests.elements();
      }

      /**
       */
      public String toString() {
            if (getName() != null)
                  return getName();
            return super.toString();
       }

      /**
       * Sets the name of the suite.
       * @param name The name to set
       */
      public void setName(String name) {
            fName= name;
      }

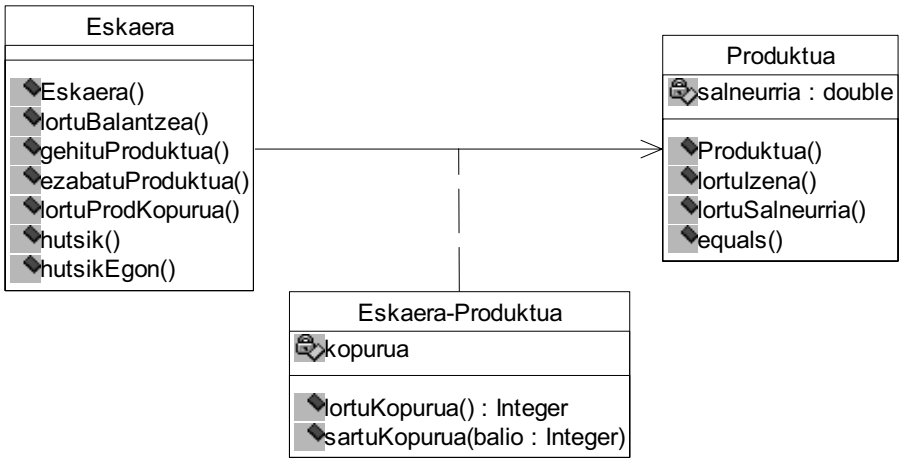
      /**
       * Returns the name of the suite. Not all
       * test suites have a name and this method
       * can return null.
       */
      public String getName() {
            return fName;
      }

      /**
       * Returns a test which will fail and log a warning
message.
       */
      private static Test warning(final String message) {
            return new TestCase("warning") {
                  protected void runTest() {
                        fail(message);

                  }

            };  }}
```

`TestSuite` klaseak biltegiratu dituen testak `TestCase` motakoak izango dira eta Adapter diseinu-patroia erabilita egokituko dira domeinuaren nahietara. Test bakoitzaren emaitzak `TestResult` motako objektu batean biltegiratuko dira.



4.2. irudia. Aplikazioaren Klase Diagrama.

JUnit tresnaren nondik norakoak azalduta, buka dezagun hausnarketa adibide bat aztertuz. Demagun enpresa bateko eskaerak kudeatu nahi ditugula eta horretarako, **4.2. irudia**n agertzen den Klase Diagramarekin hasiko gara gure garapena. Klase Diagramaren bi klaseek honako hasierako iturburu-kode hau izango dute:

```
import java.util.*;

/**
 * Eskaera klasearen ezaugarriak.
 */

public class Eskaera {

	private ArrayList Produktuak;

	/**
	 * Eraiki <code>Eskaera</code> instantzia.
	 */
	public Eskaera() {
		Produktuak = new ArrayList();
	}

	/**
	 * Itzuli balantzea.
	 *
	 * @return balantzea.
	 */
	public double lortuBalantzea() {
```

```
            Iterator i = Produktuak.iterator();
            double balantzea = 0.00;
            while (i.hasNext()) {
                  Produktua p = (Produktua)i.next();
                  balantzea = balantzea + p.lortuSalneurria();
            }

            return balantzea;
      }

      /**
       * Gehitu Produktua.
       *
       * @param p Produktua.
       */
      public void gehituProduktua(Produktua p) {
            Produktuak.add(p);
      }

      /**
       * Ezabatu Produktua.
       *
       * @param p Produktua.
       * @throws ProduktuaNotFoundException. Produktua ez badago.
       */
      public void ezabatuProduktua(Produktua p) throws
ProduktuaNotFoundException {
            if (!Produktuak.remove(p)) {
                  throw new ProduktuaNotFoundException();
            }
      }

      /**
       * Itzuli eskaeraren produktu kopurua.
       *
       * @return Produktu kopurua.
       */
      public int lortuProdKopurua() {
            return Produktuak.size();
      }

      /**
       * Hustu eskaerak.
       */
      public void hutsik() {
            Produktuak = new ArrayList();
      }

      /**
       * Adierazi eskaera hutsik dagoen ala ez.
       *
       * @return <code>true</code> Eskaera hutsik badago;
```

```
         *         <code>false</code> bestela.
         */
        public boolean hutsikEgon() {
              return (Produktuak.size() == 0);
        }
}


/**
              * Produktua klasearen ezaugarriak.
             */

public class Produktua {

        private String izena;
        private double salneurria;

        /**
         * Eraiki <code>Produktua</code>.
         *
         * @param izena Produktua izena.
         * @param salneurria Produktua salneurria.
         */
        public Produktua(String _izena, double _salneurria) {
              izena = _izena;
              salneurria = _salneurria;
        }

        /**
         * Itzuli Produktuaren izena.
         *
         * @return izena.
         */
        public String lortuIzena() {
              return izena;
        }

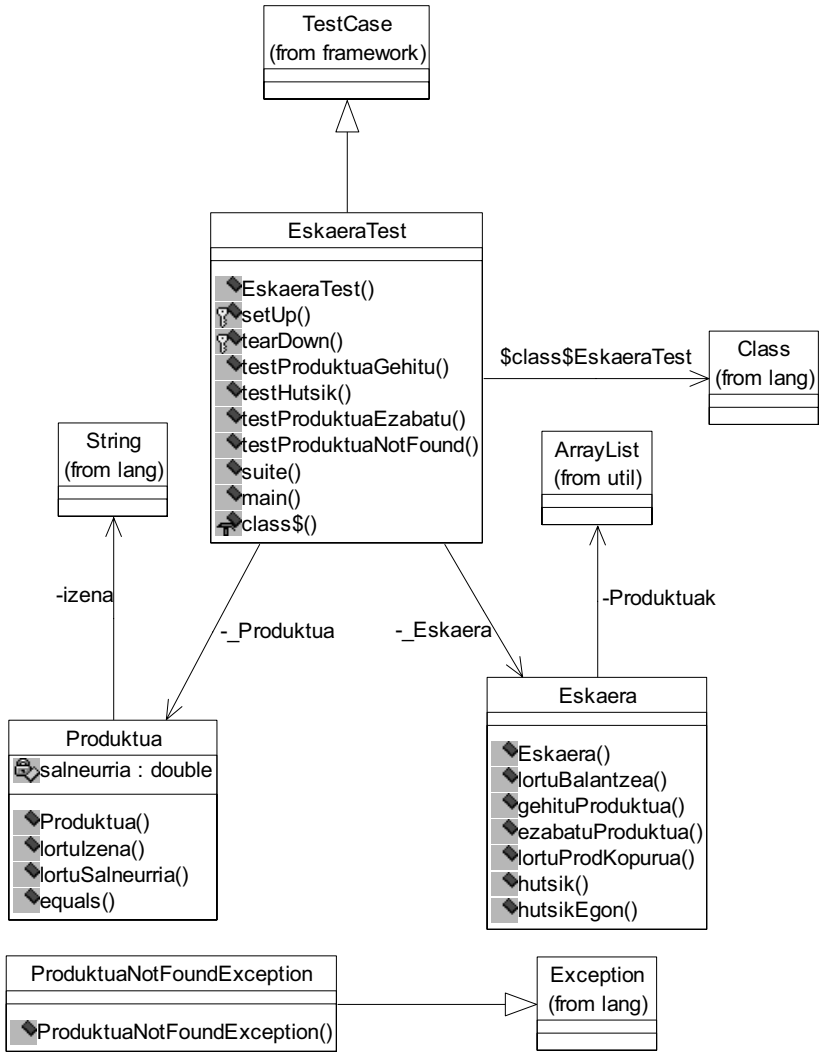
        /**
         * Itzuli Produktuaren salneurria.
         *
         * @return salneurria.
         */
        public double lortuSalneurria() {
              return salneurria;
        }

        /**
         * Testeatu Produktuen berdintasuna.
         *
         * @return <code>true</code> Produktuak berdinak
         *                           badira.
```

```
 */
public boolean equals(Object o) {

      if (o instanceof Produktua) {
            Produktua p = (Produktua)o;
            return p.lortuIzena().equals(izena);
      }

      return false; }}
```



4.3. irudia. Test automatikoak txertatzen aplikazioaren Klase Diagraman.

`Eskaera` eta `Produktua` klaseak diseinatu eta programatuz gero, klase horien portaeraren egokitasuna egiaztatu nahi dugu, eta horretarako, test automatiko batzuk diseinatu ditugu eta JUnit tresna erabilita `EskaeraTest` klasea (**4.3. irudia**) programatu dugu:

```
import junit.framework.Test;
import junit.framework.TestCase;
import junit.framework.TestSuite;

/**
 *  <code>EskaeraTest</code> ezaugarri nagusiak
  */

public class EskaeraTest extends TestCase {

      private Eskaera _Eskaera;
      private Produktua _Produktua;

      /**
       * Eraiki <code>EskaeraTest</code> instantzia
       * izen egokiarekin.
       *
       * @param izena Test case izena.
       */
      public EskaeraTest(String izena) {
            super(izena);
      }

      /**
       * Hasieraketak adierazi.
       *
       * Metodo hau testaren aurretik deitzen da.
       */
      protected void setUp() {

            _Eskaera = new Eskaera();

            _Produktua = new Produktua("Erloju digitala", 23.95);

            _Eskaera.gehituProduktua(_Produktua);
      }

      /**
       * Test amaierak adierazi.
       *
       * Metodo hau testa egikarituta deitzen da.
       */
      protected void tearDown() {
            _Eskaera = null;
      }
```

```
       /**
        * Testeatu: Produktua gehitu eskaerara.
        */
     public void testProduktuaGehitu() {

  Produktua produktua = new Produktua("Ordenagailua", 653.95);
             _Eskaera.gehituProduktua(produktua);

             double gureBalantzea = _Produktua.lortuSalneurria() +
produktua.lortuSalneurria();

   assertEquals(gureBalantzea, _Eskaera.lortuBalantzea(), 0.0);

             assertEquals(2, _Eskaera.lortuProdKopurua());
       }

       /**
        * Testeatu: Eskaera hutsik dagoen.
        */
       public void testHutsik() {

             _Eskaera.hutsik();

             assertTrue(_Eskaera.hutsikEgon());
       }

       /**
        * Testeatu: Produktua ezabatu eskaeratik.
        *
        * @throws ProduktuaNotFoundException. Altxatu salbuespena
        *                           produktua ez badago eskaeran.
        */
       public void testProduktuaEzabatu() throws
ProduktuaNotFoundException {

             _Eskaera.ezabatuProduktua(_Produktua);

             assertEquals(0, _Eskaera.lortuProdKopurua());

             assertEquals(0.0, _Eskaera.lortuBalantzea(), 0.0);
    }

       /**
        * Testeatu: Ezabatu ez dagoen produktua eskaeratik.
        */
     public void testProduktuaNotFound() {

             try {
```

```
                    Produktua produktua = new Produktua("Telefono
mugikorra", 24.95);
                    _Eskaera.ezabatuProduktua(produktua);

   fail("ProduktuaNotFoundException salbuespena altxatu behar zuen");

            } catch(ProduktuaNotFoundException success) {
            }
    }

    /**
     * Burutu behar diren test guztiak egituratzen
     * eta kontrolatzen
     *
     * @return <code>Test</code>.
     */
    public static Test suite() {

        //
        // Erreflexio mekanismoa erabili
        // testXXX() metodoak gehitzeko.
        //
        TestSuite suite = new TestSuite(EskaeraTest.class);

        //
        // Ordezko bidea gauza bera egiteko baina
         // programatzaileak kontrolatuta
         //
        // TestSuite suite = new TestSuite();
        // suite.addTest(new EskaeraTest("testProduktuaGehitu"));
        // suite.addTest(new EskaeraTest("testHutsik"));
        // suite.addTest(new EskaeraTest("testProduktuaEzabatu"));
        // suite.addTest(new EskaeraTest("testProduktuaNotFound"));
        //

        return suite;
    }

      /**
       * Main metodo nagusia.
       */
      public static void main(String args[]) {

            junit.textui.TestRunner.run(suite());

      }}
```

`EskaeraTest` klaseak JUnit ingurunearen `TestCase` klasea hedatzen du, gure aplikazioaren iturburu-kodea JUnit ingurunearekin konektatzeko. Hori eginez gero, `TestCase` klasearen hainbat baliabide (`setUp()`, `tearDown()`...) gure beharretara egokituko ditugu testak gauzatzeko. Esaterako, test bakoitza egikaritu

baino lehen, `setUp()` metodoan, prozesuaren hasieraketa nolakoa izango den adierazten dugu. Ildo beretik, `tearDown()` metodoarekin, test bakoitza egikarituta, hurrengoa hasi baino lehen zer egin behar den zehazten da.

Hurrengo urratsa egikarituko ditugun testak definitzea da. Horretarako, `testXXX()` motako metodoak programatzen ditugu, diseinatu eta programatu ditugun klaseen portaera arakatzeko. Konturatu metodo gehienak `assertxxx()` motako deiekin bukatzen direla, egiaztatzeko, aurreko aginduetan egin denak ondo funtzionatu duela `Eskaera` klasean. Azken batean, `Eskaera` klasearen portaera arakatzen dugu asertzioak erabilita klase horren metodoak egikarituz eta beren emaitzak aztertuz.

Bukatzeko, testak egikaritzea gelditzen zaigu. Horretarako, `TestSuite` motako objektu bat eraiki behar dugu, `testXXX()` motako metodo guztiak multzokatzeko `Composite` diseinu-patroia eratuta. Une honetan, bide bi ditugu `TestSuite` motako objektua egikaritzeko: bide batekin, lengoaiaren erreflexio-mekanismoa erabiltzen da `testXXX()` motako metodo guztiak gehitzeko:

```
TestSuite suite = new TestSuite(EskaeraTest.class);
```

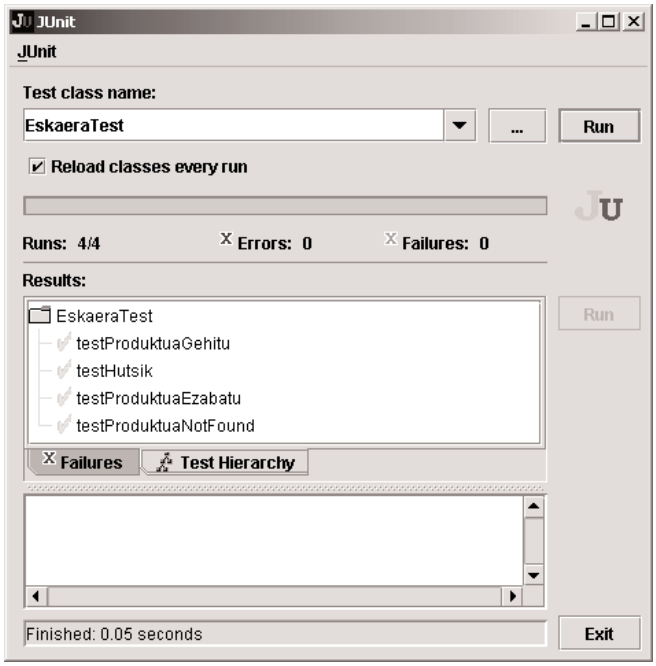
Beste bidearekin, `TestSuite` motako objektu bat eraikitzen da lehenengo, eta gero, `addTest()` metodoa erabilita, `testXXX()` motako metodo guztiak gehitzen dira:

```
TestSuite suite = new TestSuite();
    suite.addTest(new EskaeraTest("testProduktuaGehitu"));
    suite.addTest(new EskaeraTest("testHutsik"));
    suite.addTest(new EskaeraTest("testProduktuaEzabatu"));
    suite.addTest(new EskaeraTest("testProduktuaNotFound"));
```

Konturatu bigarren bide horrekin eskuz antolatzen direla testak eta programatzailearen eskuetan gelditzen dela kudeaketaren ardura guztia. Etorkizuneko edozein aldaketa edo eguneraketa, berak, zuzendu beharko du.

Gure lanaren fruituak ikuskatu (**4.4. irudia**) nahi baditugu, berriz ere bi aukera nagusi ditugu testak automatikoki egikaritzeko: lehenengo aukerarekin, `textui` paketearen `TestRunner` klasearen `run`() metodoa egikaritzen da testen bilakaera eta emaitzak ikuskatzeko. Lehenengo mota honetan, informazio guztia testu eran agertzen zaigu pantailan.

Bigarren aukerarekin, `swingui` paketearen `TestRunner` klasearen `run`() metodoa egikaritzen da testen bilakaera eta emaitzak ikuskatzeko. Bigarren mota honetan, informazio guztia (**4.4. irudia**) grafikoki egituratuta azaltzen zaigu leiho berri batean. Gure adibidean, zehatzak izan garenez diseinuan eta arduratsuak programazioan, ez dugu inongo arazorik (`Errors:0` eta `Failures:0`) aurkitu diseinatutako testetan.



4.4. irudia. JUnit tresnaren interfaze grafikoa.

Jakina, egindakoari etekina ateratzeko, non edo non testak kale egin beharko luke. Beti esan izan da, testak kale egiten duenean informazio gehiago ematen digula ondo dabilenean baino. Orduan, noiz da interesgarria testa? Gaizki ibili beharrekoan ondo dabilenean eta ondo ibili beharrekoan txarto dabilenean. Egoera horietan lortzen du programatzaileak informazio aberatsa aplikazioaren iturburu-kodea birfindu eta txukun programatzeko.

Egin duguna asko aldatu gabe, eraiki dezagun aipatutako egoera horietako bat. Horretarako, aurretik egituratu dugun test-ingurune bera erabiliko dugu egoera ezohiko horiek tratatzeko. Ezaugarri berriak aztertzeko aldaketa txiki batzuk gauzatuko ditugu `EskaeraTest` klasearen iturburu-kodean. Egindako lana ez dugunez asko aldrebestu nahi, aurretik erabili ditugun testak bere horretan lagako ditugu eta berri bat eraikiko dugu:

```
public void testErrorea() {
            try {

            _Eskaera = null;
            _Eskaera.ezabatuProduktua(_Produktua);

            } catch(ProduktuaNotFoundException success) {

            }}
```

Test berri horrek, ondo badoa prozesua, errore bat sortu behar luke eta JUnit inguruneak hala adierazi beharko lioke programatzaileari. Iturburu-kodean azaltzen den modura, exekuzio-fluxua objektu baten metodo bat egikaritzen saiatzen da, konturatu gabe aurretik `null` erreferentzia esleitu diogula. Jakina, egoera dezente sinplifikatu dugu errorea bat-batean ipinita. Egoera erreal batean, `null` erreferentziaren esleipena eta metodoaren deia asago egongo lirateke bata bestearengandik.

Bestela, `EskaeraTest` klasearen iturburu-kodean gauzatu dugun aldaketa bakarra `setUp()` metodoan zertu da eta nahiko txikerra izan da: `Produktua` motako beste objektu berri bat eraiki eta `Eskaera` motako objektuan biltegiratu hori ere. Hala, egoera berri horretan, eskaerak bi produktu izango ditu test bakoitzera sartu baino lehen eta ez bat lehen legez:

```
protected void setUp() {

      _Eskaera = new Eskaera();

      _Produktua  = new Produktua("Erloju digitala", 23.95);
      _Produktua1 = new Produktua("Telebista HP", 595.95);

      _Eskaera.gehituProduktua(_Produktua);
      _Eskaera.gehituProduktua(_Produktua1);

            }
```

Azter dezagun aldaketa horrek zer eragin dakarkion test bakoitzari:

- `testProduktuaGehitu()`. Lehen, test honekin, egoera zehatz bat aztertzen genuen. Jakinda eskaerak produktu bat zeukala, produktu berri bat eraikitzen zen eta eskaerari gehitzen zitzaion. Testak, bere aldetik, balantzea kalkulatzen zuen eta balioa `gureBalantzea` aldagaian biltegiratzen zuen.

  Hori eginda, asertzio bi (`assertEquals()`) erabiltzen genituen eskaeraren egoera aztertzeko eta egiaztatzeko gauzak ondo joan direla. Gure hasierako egoera horretan, testak ez zuen inongo arazorik aurkitzen eta ondo egikaritzen zen.

  Bestelako egoeran gaude `setUp()` metodoan bi produktu biltegiratzen baditugu eta test bakoitza eskaeran bi produkturekin egikaritzen badugu. Egoera berri honetan (`Failure`), testak huts egingo du, asertzioa egikaritzerakoan arazoak aurkitu baititu. Hona hemen, JUnit tresnak sortzen duen informazioa:

```
 testProduktuaGehitu(EskaeraTest1)
     junit.framework.AssertionFailedError:
     expected:<677.9> but was:<1273.85>

at EskaeraTest1.testProduktuaGehitu(EskaeraTest1.java:59)

at sun.reflect.NativeMethodAccessorImpl.invoke0
(Native Method)
at sun.reflect.NativeMethodAccessorImpl.invoke
(NativeMethodAccessorImpl.java:39)
at sun.reflect.DelegatingMethodAccessorImpl.invoke
(DelegatingMethodAccessorImpl.java:25)
```

- `testHutsik()`. Test honetan, `setUp()` metodoan gauzatu dugun aldaketak ez du inongo eraginik eta lehen bezala, arazorik gabe egikarituko da asertzioa.

- `testProduktuaEzabatu()`. Hemen ere lehenengo testean aipatu dugunarena errepikatzen da. Testak eskaeran produktu bakarra dagoela uste du eta hala antolatzen du asertzioa. Lehen gertatu den legez, hemen ere, nahiz eta produktu bat ezabatu eskaeratik, oraindik beste bat gelditzen da eskaeran eta, beraz, asertzioak kale egingo du. Hona hemen JUnit tresnak sortzen duen informazioa:

```
testProduktuaEzabatu(EskaeraTest1)
     junit.framework.AssertionFailedError:
     expected:<0> but was:<1>
atEskaeraTest1.testProduktuaEzabatu(EskaeraTest1.java:84)

      at sun.reflect.NativeMethodAccessorImpl.invoke0
     (Native Method)
      at sun.reflect.NativeMethodAccessorImpl.invoke
     (NativeMethodAccessorImpl.java:39)
      at sun.reflect.DelegatingMethodAccessorImpl.invoke
     (DelegatingMethodAccessorImpl.java:25)
```

- `testProduktuaNotFound()`. Azken test honetan, `setUp()` metodoarekin gauzatu dugun aldaketaren eragina ez da kontuan hartzen[41] eta, beraz, lehen arazorik gabe egikaritzen zenez, buruhauste handirik gabe egikarituko da orain ere.

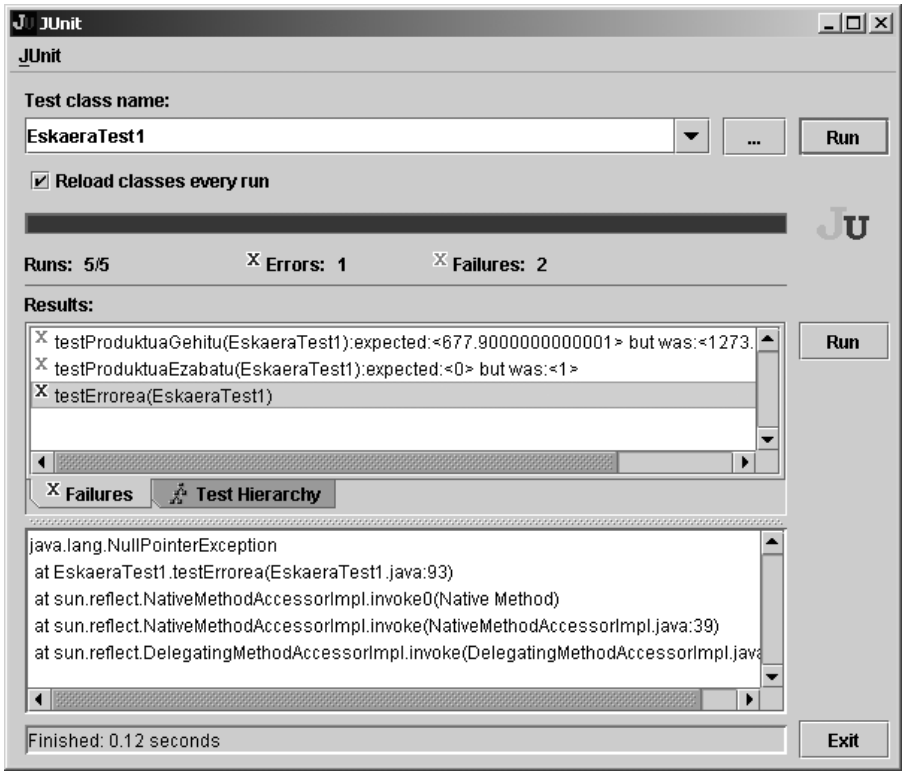
4.5. irudiak jasotzen du JUnit tresnak nola bistaratzen duen aipatutako informazio guztia. JUnit tresnaren baliabide grafikoak (**4.6. irudia** eta **4.7. irudia**) erabili nahi baditugu, lehen bezala, nahikoa zaigu `swingui` paketearen `TestRunner` klasea erabiltzea testen bilakaera eta emaitzak ikuskatzeko.

41. Kontu egin testak `Produktua` motako objektu berri bat eraikitzen duela eta bertatik bertara objektu hori ezabatzen saiatzen dela aurretik inolako `gehituProduktua()` metodorik egikaritu gabe.

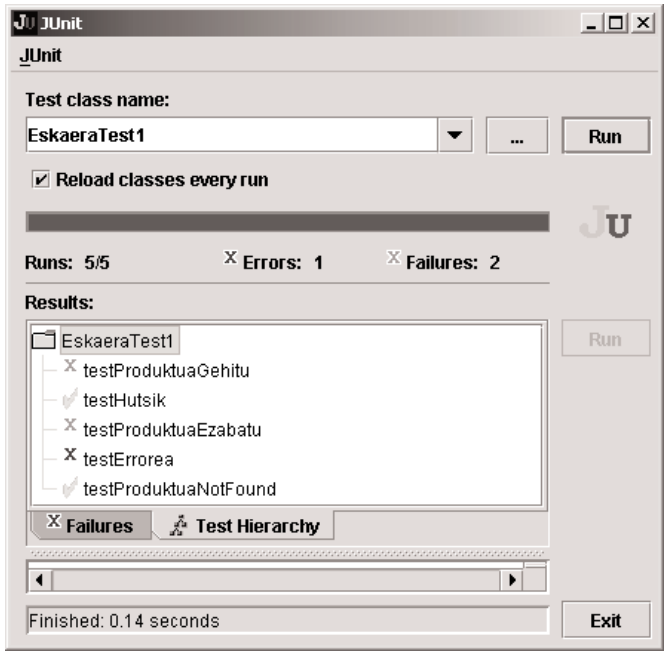
Beraz, espero genuen `ProduktuaNotFoundException` salbuespena altxatu denez altxatu behar zen unean, testa arazorik gabe egikaritu da.



4.5. irudia. JUnit tresnak sortzen duen informazioa.



4.6. irudia. JUnit tresnaren irteera grafikoa (I).



4.7. irudia. JUnit tresnaren irteera grafikoa (II).

4.2. LABURPENA

Programatzaileak klase bat programatzen duenean iturburu-kodearen funtzionalitatea egiaztatu nahi izaten du egoera ohikoak eta ezohikoak txertatuta exekuzio-garaian. Egoera horiek arazoak sor ditzakete, eta zorionez, aplikazio informatikoa oraindik programatzailearen eskuetan badago, bezeroa ez da gertatutakoaz jabetuko eta nola edo hala, gehienetan araztaile bat erabilita, buruhaustea konpondu ahal izango du baina beti geldituko zaio arra konponketa behin-behinekoa izan den ala behin betikoa. Lehentxeago edo geroxeago konturatuko da gauzatzen dituen proba xumeak produktua merkaturatu baino lehen sistematizatu egin behar dituela. Une horretan, JUnit bezalako tresna batek arazorik gabe eta erraz asko antolatuko dizkio proba bateratzaileak probatu nahi duena txukun eta era modularrean probatzeko, hau da, proba bateratzailean egiaztatu nahi den klasea ez da *kutsatu* beharrik iturburu-kode gehiago itsatsita, zeren probak egikaritzeko programatu behar den iturburu-kodea tresnaren beste klase batzuetan programatzen baita.

4.3. OINARRIZKO BIBLIOGRAFIA

Beck, K. (2004): *JUnit Pocket Guide*, O`Reilly.

Cohen, F. (2004): *Java Testing and Design : From Unit Testing to Automated Web Tests*, Prentice Hall.

Hightower, R.; Onstine, W.; Visan, P.; Payne, D. eta. Gradecki, J. (2004): *Professional Java Tools for Extreme Programming: Ant, XDoclet, JUnit, Cactus, and Maven (Programmer to Programmer)*, Wrox.

Hunt, A. eta Thomas, D. (2003): *Pragmatic Unit Testing in Java with JU*nit, The Pragmatic Programmers.

Lane, D. (2004): *JUnit: The Definitive Guide*, O`Reilly.

Link, J. eta Fröhlich, P. (2003): *Unit Testing in Java: How Tests Drive the Code*, Morgan Kaufman.

Massol, V. (2004): *JUnit in Action*, Manning Publications.

Niemeyer, G. eta Poteet, J. (2003): *Extreme Programming with Ant: Building and Deploying Java Applications with JSP, EJB, XSLT, XDoclet, and JUnit,* Sams.

Rainsberger, J. B. (2004): *JUnit Recipes: Practical Methods for Programmer Testing*, Mannig Publications.

Walnes, J.; Abrahamian, A.; Cannon-Brookes, M. eta Lightbody, P. (2003): *Java Open Source Programming: with XDoclet, JUnit, WebWork, Hibernate*, Wiley.

4.4. HELBIDE INTERESGARRIAK

Zer dago bertan?	Web gunea
JUnit tresnaren web ataria	`http://www.junit.org/index.htm`
Artikulu txikia *JUnit-i* buruz	`http://www.clarkware.com/articles/JUnitPrimer.html`
Nola txertatzen den *JUnit NetBeans* tresna aurreratuan	`http://junit.netbeans.org/`
JUnit-i buruz tutorial txikia	`http://www.adictosaltrabajo.com/tutoriales/tutoriales.php?pagina=junit`
jWebUnit tresnaren tutorial txikia	`http://www.adictosaltrabajo.com/tutoriales/tutoriales.php?pagina=jwebunit`

5. Aspektuetara zuzendutako programazioa

5.1. SARRERA

Objektuetara zuzendutako garapenak klaseetan egituratzen ditu domeinuaren objektu guztiak domeinuaren Klase Diagrama osatuz. Diagrama horretan, asoziazio- eta agregazio-erlazioek klaseak jartzen dituzte harremanetan domeinuaren objektuen arteko harremanak islatzeko. Portaera berdintsuak antzematen direnean, herentziaz baliatzen da programatzailea berrerabilera bultzatzeko. Jakin badakigu objektuak atributuak izango dituela bere egoera adierazteko eta prozedurak eta funtzioak bidaltzen zaizkion mezuak tratatzeko. Paradigma horretan, analistek eta diseinatzaileek erabat modularizatu eta kapsulatu dezakete aztertu behar den domeinua. Hots, objektuetara zuzendutako garapenak hainbat abantaila dakarkio programatzaileari: osagaien berrerabilera erraztu egiten da, modularitatea berezko baliabidea da, inplementazioaren konplexutasuna txikiagotu egiten da eta mantentze-lanen kostuak nabari jaisten dira.

Egoera horretan, domeinuaren funtzionalitatea klaseetan kapsulatzen da eta klaseen arteko antzekotasunak herentzia bidez konpontzen dira. Batzuetan, ordea, domeinuaren funtzionalitate bat edo betekizun zehatz bat ez dago erabat kapsulatuta eta ez da kokatzen klase batean edo klase baten inguru hurbilean, aitzitik, barreiatua dago Klase Diagramaren klaseetan. Objektuetara zuzendutako paradigmak erraztasunak ematen ditu berrerabiltze-prozesua bertikalki aplikatzen denean, baina arazoak ditu horizontalki aplikatu nahi denean.

Oraintsu, Gregor Kiczales-ek eta bere lankideek PARC (Palo Alto Research Center) ikerkuntza-zentroan arazoa aztertu eta tresna bat (AspectJ[42]) garatu dute auzi horri irtenbide bat emateko. Kiczales-ek eta bere lankideek, lehenengo eta behin, aspektuetara zuzendutako programazioaren ezaugarriak finkatu zituzten, arazoa teorikoki kudeatzeko. Hala, programazio-paradigma horretan, aspektuak bereganatzen du arreta nagusia domeinuaren interesgune gurutzatuak tratatzeko.

Aurrera jarraitu baino lehen argitu dezagun paradigma horrek ez duela objektuetara zuzendutako paradigma ordezkatzen, aitzitik, osatu egiten du. Adibide pare bat nahikoa izango zaigu gaiaren nondik norakoak bideratzeko. Delako aspektuak

42. `http://eclipse.org/aspectj/`

interesgunea edo ikuspegia adierazi nahiko luke eta ildo horretatik, zuhaitz bera ez litzateke gauza bera diputazioko teknikariarentzat, margolariarentzat eta biologoarentzat. Batentzat, kudeatu behar den baliabidea litzateke, bestearentzat, edertasunaren adierazpen bitxi bat, eta biologoarentzat, dendrometriaren eta estereometriaren ezaugarriak aplikatzeko txokoa. Hari beretik, gure auto matxuratuak harrera desberdina izango luke txapa-tailerrean, mekanika-tailerrean eta zer esanik ez, elektrizitate-tailerrean. Informatikaren testuinguruan, sarritan, antzeko arazoak sortzen dira funtzionalitate bat barreiatuta aurkitzen denean klaseen arkitekturan zehar, hau da, funtzionalitatea ezin dugunean erabat modularizatu eta ezin garenean herentziaz bakarrik baliatu auzia konpontzeko.

Demagun sistema informatiko bat eraiki behar dugula bezeroaren eskaerak kudeatzeko. Domeinuaren Klase Diagrama garatzen badugu, berehala agertuko zaizkigu, besteak beste, `Eskaera`, `Bezeroa`, `Produktua`, `Aurrekontua` eta `Ordainagiria` klaseak eraikitzen ari garen diseinuan. Aplikatzen den iraunkortasun-mekanismoa zeinahi den, Klase Diagramako hainbat klase `lortuXXX()`[43], `ezabatuXXX()` eta `biltegiratuXXX()` metodoez josita egongo da aplikazio informatikoaren iraunkortasuna kudeatzeko. Tamalez, metodo horiek sakabanatuta daude Klase Diagraman zehar eta edozein unetan, adibidez, iraunkortasunaren atzipen-mekanismoen portaera ikertzeko, `lortuXXX()` metodo guztien jarrera aztertu nahi badugu, zeregina zaila eta astuna bihurtuko zaigu.

Iker ditzagun Javaren teknologiak eskaintzen dizkigun baliabideak aipatutako arazoa bideratzeko. Teknologiak API[44] berezi bat eskaintzen dio programatzaileari horrelako arazoak kudeatzeko: `java.util.logging`. APIak informazioa atzematen du segurtasun-akatsak, konfigurazio-erroreak, performantzia-arazoak edota aplikazioaren programazio-erroreak kudeatzeko. APIaren baliabideak egikaritzen direnean aplikazioarekin batera, programa informatikoak datu-erregistratze (data logging) prozesu bat gauzatzen du eta gai da txosten bereziak eraikitzeko egikaritze-prozesua osatzen ari den bitartean. Hala, azkenean, programatzaileak edota sistemaren administratzaileak egikaritze-prozesuaren xehetasunak biltegiratuta dauzka eta prozesuaren nondik norakoak patxadaz azter ditzake.

Demagun, egikaritzen ari garen aplikazio informatikoak APIaren baliabideak erabili nahi dituela egikaritze-prozesuaren nondik norakoak aztertzeko eta Klase Diagramak `EskaeraKudeatzailea` klasea duela bezeroaren eskaerak kudeatzeko. Klase horretan, `lortuEskaera()`, `ezabatuEskaera()` eta `biltegiratuEskaera()` metodoak programatu ditu programatzaileak eta gauza bera egin du `ProduktuKudeatzailea`, `AurrekontuKudeatzailea` eta `OrdainagiriKudeatzailea` klaseetan.

43. `lortuXXX()` notazioa, `lortuEskaera()`, `lortuBezeroa()`, `lortuProduktua()`, `lortuAurrekontua()` eta `lortu Ordainagiria()` adierazteko erabiltzen da. Gauza bera beste metodoei dagokienez.

44. `http://java.sun.com/j2se/1.4.2/docs/guide/util/logging`

Programatzaileak, aldi berean, kontrolatuta eduki nahi ditu informazioaren biltegiratze-prozesua eta aplikazioaren egikaritze-prozesua, eta jakin nahi du baita ere, noiz egikaritzen diren metodo horiek. Hori egiteko, honako hau programatzen du `EskaeraKudeatzailea` klasean[45]:

```
      import java.util.logging.*;
      public class EskaeraKudeatzailea {
      ..............
static Logger _logger = Logger.getLogger("arrastoa");
            ...........

public Eskaera lortuEskaera(Eskaera eska) {
      _logger.logp(Level.INFO, "EskaeraKudeatzailea", "lortuEskaera",
"MetodoanSartzen");
      ..........
      return eskaera;
    }

    public boolean ezabatuEskaera(Eskaera eska) {
_logger.logp(Level.INFO, "  EskaeraKudeatzailea ", "ezabatuEskaera",
"MetodoanSartzen");
      ..............
      return balioa;
    }

    public boolean biltegiratuEskaera(Eskaera eska) {
_logger.logp(Level.INFO, " EskaeraKudeatzailea ",
"biltegiratuEskaera", "MetodoanSartzen");
      ..............
      return balioa;
    }  }
```

Argi ikusten da datuen erregistratze-prozesuak klaseen modularitatea apurtu egiten duela ezaugarri *ezezagunak* itsatsiz klaseetan eta metodoetan. Bestalde, prozesua erabat mekanikoa eta aspergarria suertatzen da eta kudeatzaile berri bat programatu behar denean, programatzaileak iturburu-kode bera kopiatu eta itsatsi behar du behin eta berriz.

Aztertzen ari garen adibideak arazoa azaleratzen lagundu digu eta objektuetara zuzendutako programazioaren mugak non aurkitzen diren nabarmen finkatu digu. Bestalde, begi-bistakoa da arazoa txukun konpondu nahi bada, aipatutako APIaren estrategia ez dela egokiena eta beste dimentsio edo ikuspegi berri bat behar dugula aplikazioaren funtzionalitate berri horiek inplementatzeko: aspektuetara zuzendutako programazioa.

45. Gauza bera beste `XXXKudeatzailea` klase guztientzat. Bakarrik, `logp()` metodoaren parametroak aldatzen dira iturburu-kodean.

Programazio-paradigma berriari bi ezaugarri eskatuko dizkiogu objektuetara zuzendutako programazioarekin bateragarria izan dadin:

1. Diseinatu eta inplementatu nahi dugun aspektuak edo interesguneak ez du diseinu nagusiaren modularitatea auzitan jarri behar.

2. Tresna aurreratu eta automatiko bat diseinatzen eta inplementatzen bada aspektuak adierazteko, gardena izan behar du aplikazio informatikoarentzat, hau da, inplementatu nahi dugun aspektua arazorik gabe integratu behar da aplikazio informatikoaren iturburu-kodearekin eta programatzaileak ez du arduratu behar integrazioaren xehetasunak programatzeaz.

Azter dezagun ezaugarri horiek nola inplementatzen eta kudeatzen diren AspectJ tresna automatikoan aspektu bat itsatsi nahi diogunean aplikazio informatikoari. Bazterrak ez nahasteko, lehen adibide honetan, aplikazioaren iturburu-kode guztia klase bakarrean bilduko dugu eta honako iturburu-kode hau izango du:

```
public class KaixoKlasea {
  public static void idatziKaixo () {
    System.out.println ("Kaixo eta ongi etorri !!!!");
  }

  public static void idatziZerbait (String s) {
    System.out.println (s);
  }

  public static void main (String[] args) {
    idatziKaixo ();
          idatziZerbait ("Hello and welcome");
       }}
```

Programa guztiak, mezu bi idazten ditu `idatziXXX()` motako metodo bi egikarituta. Ezer gutxi alajaina!!!

Has gaitezen, bada, gauza interesgarriagoak egiten. Horretarako, programa nagusia ukitu gabe aspektu bat inplementatuko dugu aipatutako tresna erabilita eta hona hemen haren iturburu-kodea:

```
public aspect NireAspektua {

   public pointcut idatziMetodoDeia (): call (public void
                                              KaixoKlasea.idatzi*() );
   public pointcut idatziMetodoDeiaArg (String str): call
                    (public void KaixoKlasea.idatziZerbait (String))
                    && args(str);
```

```
   before(): idatziMetodoDeia() {
   System.out.println("\n KaixoKlasea." +
       thisJoinPointStaticPart.getSignature().getName() +
       "  hasiera..." );
   }

   after(): idatziMetodoDeia() {
   System.out.println("\n KaixoKlasea." +
       thisJoinPointStaticPart.getSignature().getName() +
       "  bukaera...");
   }

   before(String str): idatziMetodoDeiaArg(str) {
     if (str .length() < 3) {
     System.out.println ("Errorea: Hiru karaktere baino gutxiago
idatzi dituzu");
     return;
           }
         }}
```

Iturburu-kodea nahiko ezaguna egiten zaigu baina hitz batzuen (`pointcut`, `before()`, `after()`...) esanahia ulertzeko arazoak ditugu, ez baitira ohikoak Javaren teknologiaren testuinguruan. Hitz berri horien esanahia zehazten hasi baino lehen, inplementatu berri dugun aspektuaren nondik norakoak azalduko ditugu. Aspektuak iturburu-kode nagusiaren `idatziXXX()` metodo biak kontrolatu nahi ditu, eta horretarako, programatzaileak AspectJ tresnari adierazi behar dio iturburu-kode nagusiaren zein tokitan finkatu behar duen arreta. Eragiketa hori gauzatzeko, `pointcut` motako ezaugarri bi eraikitzen dira AspectJ tresna erabilita. Arretaguneak definitu eta adierazten dira gero, zer egin nahi den exekuzio-fluxua gune horietara heltzen denean, `after()` eta `before()` erazagupenak zehaztuta. Gure adibidean, klase nagusiaren (`KaixoKlasea`) edozein `idatziXXX()` metodo argumenturik gabea egikaritu baino lehen, mezu bat idatziko da tresnaren baliabideak erabilita. Halaber, metodoa egikarituz gero, beste mezu bat idatziko da gertaera hori adierazteko.

Azken `before()` erazagupena `idatziZerbait()` metodoari lotzen zaio eta metodoari pasatu zaion `String` motako argumentuaren luzera kontrolatzeko diseinatu da, hau da, metodoari pasatzen diogun argumentuak hiru karaktere baino gutxiago badu, aspektuak mezu bat idatziko du irteera estandarrean.

Kontu egin aipatutako eragiketa guztiak iturburu-kode nagusia ukitu[46] gabe burutu direla eta hortaz, aplikazioaren klaseen modularitatea ziurtatuta gelditzen dela.

46. Geroxeago ikusiko dugu hau ez dela guztiz egia aplikazioaren kodearentzat baina oraingoz bere horretan utziko dugu eztabaida.

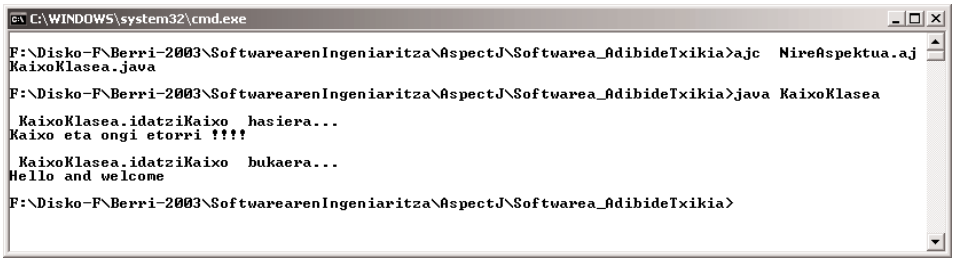
Aspektua eta aplikazioaren iturburu-kodeak eraikita, biak erlazionatzea gelditzen zaigu soilik. Hori egiteko, AspectJ tresnaren konpilatzailea erabiliko dugu eta honako hau idatziko dugu:

```
> ajc NireAspektua.aj KaixoKlasea.java
```

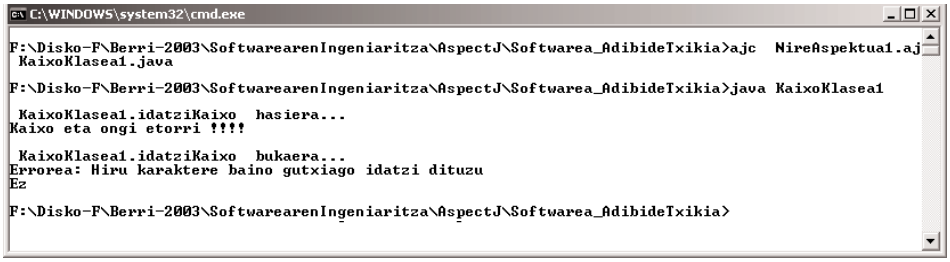
Aplikazioa egikaritu nahi badugu nahikoa zaigu honako hau idaztea:

```
> java KaixoKlasea
```

5.1. irudiak eta **5.2. irudia**k irteera estandarrean zer idazten den islatzen dute behin aplikazioa egikarituta. Kontu egin, bukaeran, emaitza desberdinak jasotzen direla irudi bakoitzean.



5.1. irudia. AspectJ tresnaren irteera estandarra (I).



5.2. irudia. AspectJ tresnaren irteera estandarra (II).

Prozesu guztian benetan deigarri egiten zaigu nola ezer aldatu gabe aplikazioaren iturburu-kodean Javaren teknologiak txukun eta zintzo gauzatzen dituen eragiketa guztiak. Non gauzatzen da miraria? Balizko mirari guztietan legez, hemen ere, mirariak mirari. `KaixoKlasea.class` fitxategia deskonpilatzen[47] badugu, misterioa argitu egingo da eta lurtar bihurtu:

47. Guk zeregin horretarako jad tresna deskonpilatzailea erabili dugu.

```
// Decompiled by Jad v1.5.7f. Copyright 2000 Pavel Kouznetsov.
// Jad home page:
http://www.geocities.com/SiliconValley/Bridge/8617/jad.html
// Decompiler options: packimports(3)
// Source File Name:   KaixoKlasea.java

import java.io.PrintStream;
import org.aspectj.runtime.reflect.Factory;

public class KaixoKlasea
{

    public KaixoKlasea()
    {
    }

    public static void idatziKaixo()
    {
        System.out.println("Kaixo eta ongi etorri !!!!");
    }

    public static void idatziZerbait(String arg0)
    {
        System.out.println(arg0);
    }

    public static void main(String args[])
    {
        try
        {

NireAspektua.aspectOf().ajc$before$NireAspektua$14b(ajc$tjp_0);
            idatziKaixo();
        }
        catch(Throwable throwable)
        {

NireAspektua.aspectOf().ajc$after$NireAspektua$1f6(ajc$tjp_0);
            throw throwable;
        }
        NireAspektua.aspectOf().ajc$after$NireAspektua$1f6(ajc$tjp_0);
        String s = "Hello and welcome";
// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
//  Bigarren Kaixo klasearentzat -KaixoKlasea1- beste honako hau
//  agertzen da aurreko lerroan:
//      String s = "Ez";

        NireAspektua.aspectOf().ajc$before$NireAspektua$29f(s);
        idatziZerbait(s);
    }
```

```
 public static final org.aspectj.lang.JoinPoint.StaticPart ajc$tjp_0;

    static
    {
        Factory factory = new Factory("KaixoKlasea.java",
Class.forName("KaixoKlasea"));
        ajc$tjp_0 = factory.makeSJP("method-call",
factory.makeMethodSig("9-idatziKaixo-KaixoKlasea--void-"), 11);
         }
}
```

Tresnak, espero genuen legez, iturburu-kode berria txertatzen du gure hasierako iturburu-kodean, alabaina, txertaketa gardena da programatzailearentzat eta automatikoki gauzatzen da.

5.2. ADIBIDE KONPLEXUAK: DEIAREN FAKTURAZIOAREN KUDEAKETA

Igerian ikasi ondoren murgil gaitezen ur handiagotan. Horretarako, AspectJ tresnaren programazio-gidan azaltzen diren bi adibide konplexutan finkatuko dugu arreta nagusia. Batean, interesguneen arloa geruza desberdinak eraikita antolatuko da oinarrizko egitura batekin hasita. Bestean, ostera, Observer diseinu-patroi ezaguna inplementatuko da aspektuak erabilita.

Lehenengo adibidearekin (**5.3. irudia**) sistema telefoniko bat eraiki nahi da bezeroaren deiak egituratu, kontrolatu, neurtu eta fakturatzeko. Aplikazio informatikoa hiru interesgune hauen inguruan antolatuko da: konexio telefonikoa, denboraren neurketa eta deiaren fakturazioa. Hiru interesguneek bezeroaren deia dute jomuga baina bakoitzak dei hori bere nahietara egokitu beharko du. Gainera, hiru interesguneen artean erlazio interesgarri bat gertatzen da: mendekotasun-erlazioa.

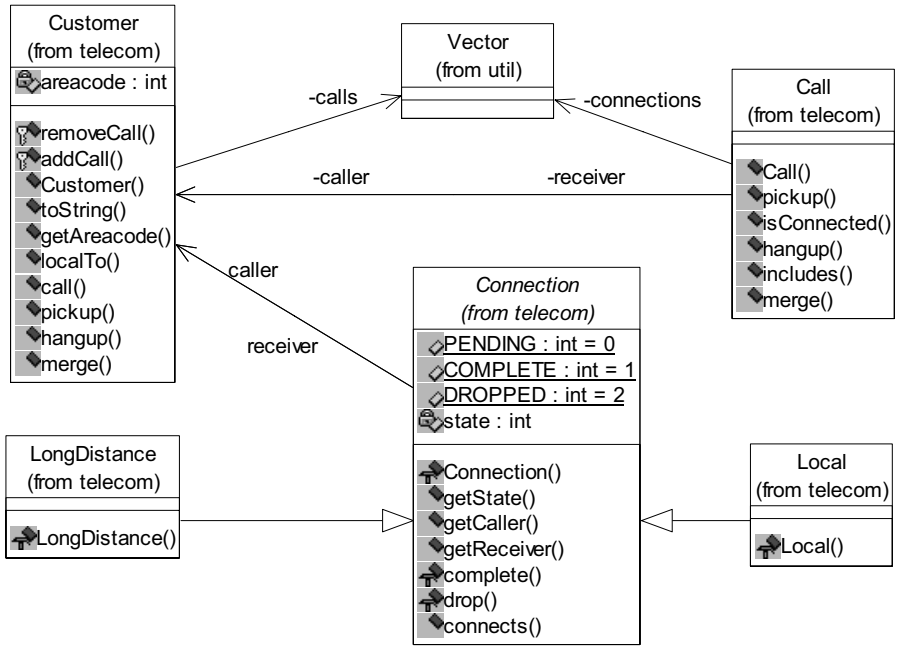


5.3. irudia. Sistema telefonikoaren geruza-egiturak.

Aipatutako erlazioak erabat baldintzatuko ditu aplikazioaren diseinua eta inplementazioa adibide honetan. Hori dela eta, aplikazioaren arkitekturak hiru geruza izango ditu:

- Hasierako eta behe-mailako geruzak aplikazioaren oinarrizko objektuak (**5.4. irudia**) egituratuko ditu bezeroaren deiak eta konexioak kudeatzeko.
- Erdi-mailako geruzak konexioaren denboraren nondik norakoak kontrolatuko ditu. Geruza hau behe-mailako geruzak burututako lanaz baliatuko da, eta horretarako, aspektuetara zuzendutako ikuspegia erabiliko da haien artean dagoen mendekotasun-erlazioa kudeatzeko.
- Hirugarren geruzak beste bien lana xurgatuko du, bezeroari deiaren faktura egokitzeko. Hemen ere, lehen bezala, aspektuez baliatuko gara baina ikuspuntu berri bat landuta: erdi-mailako geruza kudeatzeko antolatuko den aspektua habiaratu egingo da aspektu berri honetan, fakturazioa gauzatzeko.

Ezohiko arkitektura bitxi horrekin, birtualki, hauxe esango genuke: deiaren denbora neurtu behar duenari ez zaizkiola ardura konexioaren eta deiaren trikimailu guztiak, bera, mugetan mugitzen da, hau da, noiz hasten den eta noiz amaitzen den deia interesatzen zaio bakarrik, erloju-kontuak alegia. Gainera, lan *iluna* behe-mailako geruzak beteko duenez, biziki eskertuko dio laguntza. Ostera, deiaren faktura kudeatu behar duen gibel-handiak, endosinbiosiz hordituta, denbora neurtzailearen emaitzak zurrupatuko ditu.



5.4. irudia. Aplikazioaren Klase Diagrama.

Horra hor, bada, aplikazioaren arkitekturaren diseinu orokorraren ezaugarri nagusiak.

Demagun orain, Nontzebarri jaunak Kontakatilu anderearekin harremanetan jarri nahi duela auzoko azken berriez berba egiteko. Nontzebarrik Kontakatiluri deitzen dionean (`Nontzebarri.call(Kontakatilu)`) dei —`Call` motako objektu bat— berri bat sortzen da aplikazio informatikoan:

```
public Call call(Customer receiver) {
        Call call = new Call(this, receiver);
        addCall(call);
              return call;
          }
```

Jarraian Kontakatiluk deia onartzen du:

```
      Kontakatilu.pickup(call1);

public void pickup(Call call) {
        call.pickup();
              addCall(call);
          }
```

Hartzaileak deia onartzen badu, konexioa gauzatuta gelditzen da eta kontukontari hasten dira. Hartzaileak edota igorleak eman diezaioke deiari amaiera, hau da:

```
      Nontzebarri.hangup(call1);
            edota
      Kontakatilu.hangup(call1);

public void hangup(Call call) {
      call.hangup(this);
      removeCall(call);
          }
```

Hona hemen, ahotan erabili ditugun hiru klaseen (`Customer`, `Call` eta `Connection`) iturburu-kodea:

```
package telecom;
import java.util.Vector;

/**
 * Customers have a unique id (name in this case for didactic purposes
 * but it could be telephone number) and area code.
 * They also have protocol for managing calls: call, pickup, etc.
 */
public class Customer {
```

```
private String name;
private int areacode;
private Vector calls = new Vector();

/**
 * unregister a call
 */
protected void removeCall(Call c){
   calls.removeElement(c);
}

/**
 * register a call
 */
protected void addCall(Call c){
   calls.addElement(c);
}

/**
 * Make a new customer with given name
 */
public Customer(String name, int areacode) {
    this.name = name;
    this.areacode = areacode;
}

/**
 * String rendition of customer
 */
public String toString() {
    return name + "(" + areacode + ")";
}

/**
 * what area is the customer in?
 */
public int getAreacode(){
   return areacode;
}

/**
 * Is the other customer in the same area?
 */
public boolean localTo(Customer other){
   return areacode == other.areacode;
}

/**
 * Make a new call to receiver
 */
```

```
    public Call call(Customer receiver) {
        Call call = new Call(this, receiver);
        addCall(call);
        return call;
    }

    /**
     * pick up a call
     */
    public void pickup(Call call) {
        call.pickup();
        addCall(call);
    }

    /**
     * hang up a call
     */
    public void hangup(Call call) {
      call.hangup(this);
      removeCall(call);
    }

    /**
     * Merge a pair of calls - conference them
     * PRE: call1.includes(this)
     *      call2.includes(this)
     *      call1.connected()
     *      call2.connected()
     * POST: call1 includes all customers connected by call1@pre and
call2@pre
     */
    public void merge(Call call1, Call call2){
      call1.merge(call2);
      removeCall(call2);
    }
}
```

oOo

```
package telecom;
import java.util.Vector;
import java.util.Enumeration;

public class Call {

    private Customer caller, receiver;
    private Vector connections = new Vector();

    /**
     * Create a new call connecting caller to receiver
```

```
     * with a new connection. This should really only be
     * called by Customer.call(..)
     */
    public Call(Customer caller, Customer receiver) {
        this.caller = caller;
        this.receiver = receiver;
        Connection c;
        if (receiver.localTo(caller)) {
           c = new Local(caller, receiver);
        } else {
           c = new LongDistance(caller, receiver);
        }
        connections.add(c);
    }

    /**
     * picking up a call completes the current connection
     * (this means that you shouldnt merge calls until
     * they are completed)
     */
    public void pickup() {
       Connection connection = (Connection)connections.lastElement();
         connection.complete();
    }


    /**
     * Is the call in a connected state?
     */
    public boolean isConnected(){
       return ((Connection)connections.lastElement()).getState()
            == Connection.COMPLETE;
    }

    /**
     * hanging up a call drops the connection
     */
    public void hangup(Customer c) {
        for(Enumeration e = connections.elements();
e.hasMoreElements();) {
           ((Connection)e.nextElement()).drop();
        }
    }

    /**
     * is Customer c one of the customers in this call?
     */
    public boolean includes(Customer c){
       boolean result = false;
       for(Enumeration e = connections.elements();
```

```
e.hasMoreElements();) {
          result = result ||
((Connection)e.nextElement()).connects(c);
      }
      return result;
    }

    /**
     * Merge all connections from call 'other' into 'this'
     */
    public void merge(Call other){
      for(Enumeration e = other.connections.elements();
e.hasMoreElements();){
          Connection conn = (Connection)e.nextElement();
          other.connections.remove(conn);
          connections.addElement(conn);
      }
    }}
```

oOo

```
package telecom;

public abstract class Connection {

    public static final int PENDING = 0;
    public static final int COMPLETE = 1;
    public static final int DROPPED = 2;

    Customer caller, receiver;
    private int state = PENDING;

    /**
     * Creatte a new Connection between a and b
     */
    Connection(Customer a, Customer b) {
        this.caller = a;
        this.receiver = b;
    }

    /**
     * what is the state of the connection?
     */
    public int getState(){
      return state;
    }

    /**
     * get the customer who initiated this connection
     */
    public Customer getCaller() { return caller; }
```

```
    /**
     * get the customer who received this connection
     */
    public Customer getReceiver() {  return receiver; }

    /**
     * Called when a call is picked up.  This means the b side has
picked up
     * and the connection should now complete itself and start passing
data.
     */
    void complete() {
        state = COMPLETE;
        System.out.println("Konexioa bukatu");
    }

    /**
     * Called when the connection is dropped from a call.  Is intended
to
     * free up any resources the connection was consuming.
     */
    void drop() {
      state = DROPPED;
        System.out.println("Konexioa utzi");
    }

    /**
     * Is customer c connected by this connection?
     */
    public boolean connects(Customer c){
      return (caller == c || receiver == c);
          }}
```

Kontu egin hartzailearen eta igorlearen area-kodea bera bada, konexioa lokala izango dela, eta bestela, urrunekoa.

Azpiegitura guztia prestatuz gero, simulazio txiki bat gauzatuko dugu:

```
package telecom;

public class BasicSimulation extends AbstractSimulation {

    public static void main(String[] args){
      simulation = new BasicSimulation();
      simulation.run();
    }

    protected void report(Customer c) { }

}
```

oOo

```
package telecom;

public abstract class AbstractSimulation {

    public static AbstractSimulation simulation;

    /**
     * Creates objects and puts them to work.
     */
    public void run() {
        Customer Itxaso = new Customer("Itxaso", 650);
        Customer Ibai = new Customer("Ibai", 650);
        Customer Aintzira = new Customer("Aintzira", 415);

        say("Itxasok Ibairi deitzen dio");
        Call c1 = Itxaso.call(Ibai);
        wait(1.0);
        say("... Ibaik onartu egiten du... ");
        Ibai.pickup(c1);
        wait(2.0);
        say("... Itxasok eskegi egiten du... ");
        Itxaso.hangup(c1);
        report(Itxaso);
        report(Ibai);
        report(Aintzira);

        say("Ibaik Aintzirari deitzen dio");
        Call c2 = Ibai.call(Aintzira);
        say("... Aintzirak onartu egiten du... ");
        Aintzira.pickup(c2);
        wait(1.5);
        say("Aintzirak eskegi egiten du... ");
        Aintzira.hangup(c2);
        report(Itxaso);
        report(Ibai);
        report(Aintzira);
    }

    /**
     * Print a report of the connection time for customer
     */
    abstract protected void report(Customer c);

    /**
     * Wait 0.1 seconds per "second" for simulation
     */
    protected static void wait(double seconds) {
        Object dummy = new Object();
        synchronized (dummy) {
```

```
            //XXX cheat and only wait 0.1 seconds per second
              try { dummy.wait((long)(seconds*100)); }
              catch (Exception e) {}
        }
    }

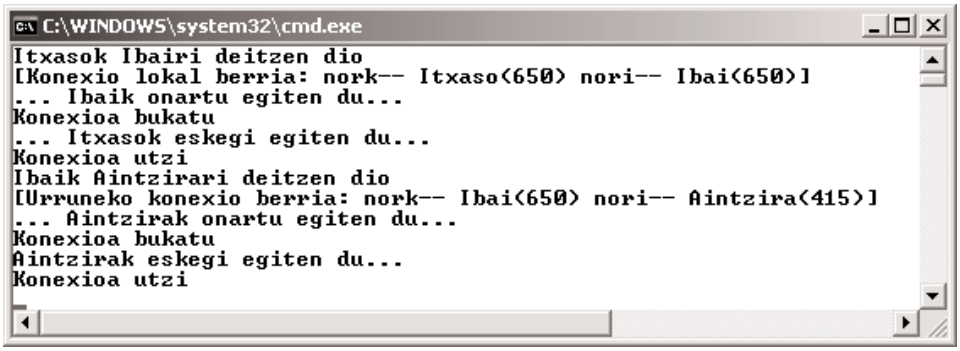
    /**
     * Put a message on standard output
     */
    protected static void say(String s){

       System.out.println(s);
    }}
```

Simulazioak (**5.5. irudia**) dei bi kudeatzen ditu:

1. Lehenengo deian, Itxasok Ibairi deitzen dio eta bai batak eta bai besteak area-kode bera dutenez, konexio lokala eratzen da.
2. Bigarren deian, Ibaik Aintzirari deitzen dio, baina oraingoan area-kodea desberdina denez, urruneko konexioa zertzen da.

Simulazioak, bai Klase Diagraman baita inplementazioan ere, objektuetara zuzendutako paradigmari jarraitzen dio orpoz orpo eta exekuzioarekin egiaztatu dugun legez, deiaren hartzailea eta igorlea harremanetan jartzen ditu konexio lokala edota urruneko konexioa eraikita. Hori eskatzen zitzaion egiteko eta hori gauzatzen du.



5.5. irudia. Aplikazioaren konexio telefonikoa gauzatzen.

Demagun aplikazioa erosi digun erakundearen arduradunak funtzionalitate berri bat eskatzen digula premia berriak sortu direlako erakundean. Orain arte deiaren iraupena ez zen neurtzen konexioaren kostua doan zelako erakundearentzat baina datorren hiletik aurrera, konexio bakoitzak, kostu finko batzuk izango ditu eta erakundeak egokitu egin behar ditu bere diru-sarrerak kostu berriei aurre egiteko. Hori dela eta, egokitzapena bi urratsetan gauzatuko da: lehenengoan, konexioaren iraupena neurtuko da, eta bigarrenean, ostera, konexioaren kostua kalkulatuko zaio igorleari.

Objektuetara zuzendutako hurbilketa erabiliko bagenu funtzionalitate berri hori diseinatzeko eta inplementatzeko, Klase Diagrama nagusia aldatu egin beharko genuke lehengo klase zaharrei klase berriak sortuz eta, batez ere, atributu eta metodo berriak itsatsiz, horrek dakarren arriskuarekin. Bide hori aukeratu beharrean aspektuetara zuzendutakoa hartuko dugu. Ikuspegi berri horrek abantaila nagusi bat du: orain arte diseinatu eta inplementatutako guztia bere horretan utziko du, amiñi bat ere ukitu eta aldatu gabe. Orain arte kapsulatuta egon den guztiak kapsulatuta egoten jarraituko du. Horretarako, funtzionalitate berria geruza berri baten gisara eraikiko da egindakoaren gainean, egindakoa informazio-iturri gisa erabiliz.

Diseinatu nahi dugun funtzionalitateak klase berri bakarra izango du deiaren konexioaren hasiera-ordua eta amaiera-ordua neurtzeko:

```
package telecom;

public class Timer {
    public long startTime, stopTime;

    /**
     * set the start time
     */
    public void start() {
        startTime = System.currentTimeMillis();
        stopTime = startTime;
    }

    /**
     * set the end time
     */
    public void stop() {
        stopTime = System.currentTimeMillis();
    }

    /**
     * set how much time passed between last start and stop?
     */
    public long getTime() {
              return stopTime - startTime;
    }}
```

Denbora neurtuko duen baliabidea eratuz gero, ezin dugu isolatuta utzi, orain arte egindakoarekin *konektatu* behar dugu. Horretarako, bi aspektu garatuko ditugu:

1. Batekin, hasierako eta bukaerako orduak irteera estandarretik aterako ditugu. Hori egiteko nahikoa zaigu aspektuan `after` erazagupen bi adieraztea, `Timer` motako objektu batekin erlazionatzen direnak:

```
package telecom;

public aspect TimerLog {

after(Timer t): target(t) && call(* Timer.start())  {
  System.err.println("Hasierako ordua: " + t.startTime);
}

after(Timer t): target(t) && call(* Timer.stop()) {
            System.err.println("Bukaerako ordua: " + t.stopTime);
          }}
```

   Horrela, `Timer` motako objektu baten `start()` metodoa egikaritzen den bakoitzean hasierako ordua idazten da irteera estandarrean.

   Gauza bera objektuaren `stop()` metodoa egikaritzen denean, baina oraingoan, amaierako ordua idazten da.

2. Bestearekin, erlojua (`Timer` objektua) eta deiaren konexioa jarriko ditugu harremanetan, honako prozesu hau eratuta: konexioa eratzen denean (`Connection.complete()`) erlojua martxan hasten da (`getTimer(c).start()`) eta konexioa ixten denean (`Connection.drop()`), ostera, erlojua gelditu egiten da (`getTimer(c).stop()`) eta denborak kalkulatzen dira deiaren igorlearentzat eta hartzailearentzat:

```
c.getCaller().totalConnectTime += getTimer(c).getTime();
c.getReceiver().totalConnectTime += getTimer(c).getTime()
```

   Denborak bezeroari metatzeko eta konexio bakoitzak bere erlojua izan dezan aldagai bat eta `Timer` motako objektu bat erazagutzen ditugu:

```
public long Customer.totalConnectTime = 0;
private Timer Connection.timer = new Timer();
```

   Konexioaren hasiera eta amaiera deiaren iraupenarekin lotzeko, `after` motako bi erazagupen erabiliko ditugu `Timer` motako objektuaren `start()` eta `stop()` metodoekin erlazionatzeko. Hala, konexio bat eratzen denean, martxan hasiko da erlojua, eta konexioa ixten denean, gelditu egingo da erlojua.

Hona hemen azaldutako guztia nola gelditzen den bigarren aspektu honetan:

```
package telecom;

public aspect Timing {
    /**
     * Every Customer has a total connection time
     */
    public long Customer.totalConnectTime = 0;

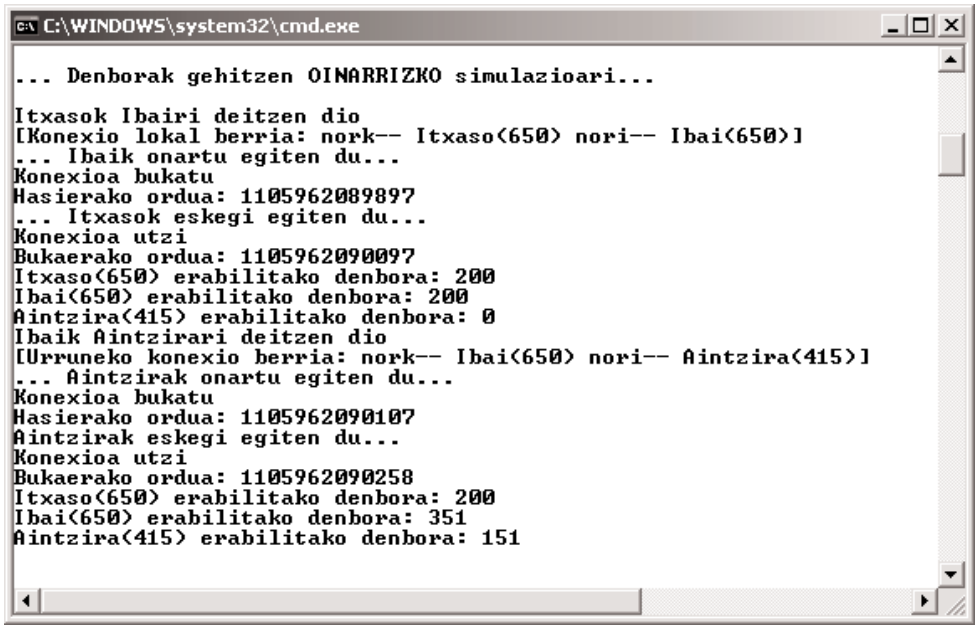
    public long getTotalConnectTime(Customer cust) {
        return cust.totalConnectTime;
    }
    /**
     * Every connection has a timer
     */
    private Timer Connection.timer = new Timer();
    public Timer getTimer(Connection conn) { return conn.timer; }
    /**
     * Start the timer when call completed
     */
    after (Connection c): target(c) && call(void
Connection.complete()) {
        getTimer(c).start();}
    /**
     * When to stop the timer
     */
    pointcut endTiming(Connection c): target(c) &&
        call(void Connection.drop());
    /**
     * Stop the timer when call dropped and update the involved
parties
     */
    after(Connection c): endTiming(c) {
        getTimer(c).stop();
        c.getCaller().totalConnectTime += getTimer(c).getTime();
        c.getReceiver().totalConnectTime += getTimer(c).getTime(); }}
```

Funtzionalitate berriaren zati hau probatzeko (**5.6. irudia**), aurreko simulazioaren datuak erabiliko ditugu, eta horretarako, `AbstractSimulation` klase abstraktua hedatuko dugu simulazio berria egituratzeko:

```
package telecom;
public class TimingSimulation extends AbstractSimulation {
    public static void main(String[] args){
      System.out.println("\n... Denborak gehitzen OINARRIZKO
simulazioari...\n");
      simulation = new TimingSimulation();
      simulation.run();
```

```
    }
    /**
     * Print a report of the connection time for customer
     */
    protected void report(Customer c){
              Timing t = Timing.aspectOf();
               System.out.println(c + " erabilitako denbora: " +
t.getTotalConnectTime(c)); }}
```

Kontu egin simulazio berri honetan `report()` metodoan `aspectOf()` metodo estatikoa[48] egikaritzen dela aspektu berriaren baliabidetara atzitu ahal izateko.



5.6. irudia. Aplikazioaren oinarrizko simulazioari denborak gehitzen.

Funtzionalitate berriaren lehen urratsa burutuz gero, hau da, deiaren konexioaren iraupena neurtuta, bigarren eta azken urratsa gauzatuko dugu igorlearen konexioaren kostua kalkulatzeko. Funtzionalitatearen bigarren urrats hori bideratzeko, berriro ere, aspektuetara zuzendutako ikuspegia erabiliko dugu. Urrats honetan inplementatuko dugun aspektu berria aplikazioaren azken geruza eratzeko erabiliko dugu eta aurretik programatutako aplikazioaren klaseen eta aspektuen gainean eraikiko da:

48. Metodo estatikoak aspektuaren instantzia bat itzultzen du.

```
package telecom;
public aspect Billing {
 // precedence required to get advice on endtiming in the right order
    declare precedence: Billing, Timing;

    public static final long LOCAL_RATE = 3;
    public static final long LONG_DISTANCE_RATE = 10;

    public Customer Connection.payer;
    public Customer getPayer(Connection conn) { return conn.payer; }
    /**
     * Caller pays for the call
     */
    after(Customer cust) returning (Connection conn):
        args(cust, ..) && call(Connection+.new(..)) {
        conn.payer = cust;
    }

    /**
     * Connections give the appropriate call rate
     */
    public abstract long Connection.callRate();


    public long LongDistance.callRate() { return LONG_DISTANCE_RATE; }
    public long Local.callRate() { return LOCAL_RATE; }


    /**
     * When timing stops, calculate and add the charge from the
     * connection time
     */
    after(Connection conn): Timing.endTiming(conn) {
        long time = Timing.aspectOf().getTimer(conn).getTime();
        long rate = conn.callRate();
        long cost = rate * time;
        getPayer(conn).addCharge(cost);
    }


    /**
     * Customers have a bill paying aspect with state
     */
    public long Customer.totalCharge = 0;
    public long getTotalCharge(Customer cust) { return
cust.totalCharge; }

    public void Customer.addCharge(long charge){
              totalCharge += charge;
             }}
```

Aplikazioa puzten joan zaigunez, aspektuen oharren ordena kontrolatu egin behar da lehentasunak erazagutuz aspektu berrietan. Hori dela eta, gure aspektu

berriak, mota (`declare precedence`) horretako erazagupen bat behar du aspektuaren oharren lehentasuna azpimarratzeko eta tresna informatiko aurreratuari hala adierazteko. Behin hori eginda, aspektu berriak aldagai eta metodo berriak erazagutzen ditu konexioaren fakturazioa egiteko. Horretarako, hiru baliabide sortzen ditu:

1. Konexioari tarifa esleitzeko, `callRate()` metodoa erazagutzen du.

```
  public abstract long Connection.callRate();

public long LongDistance.callRate() { return LONG_DISTANCE_RATE; }
  public long Local.callRate() { return LOCAL_RATE; }
```

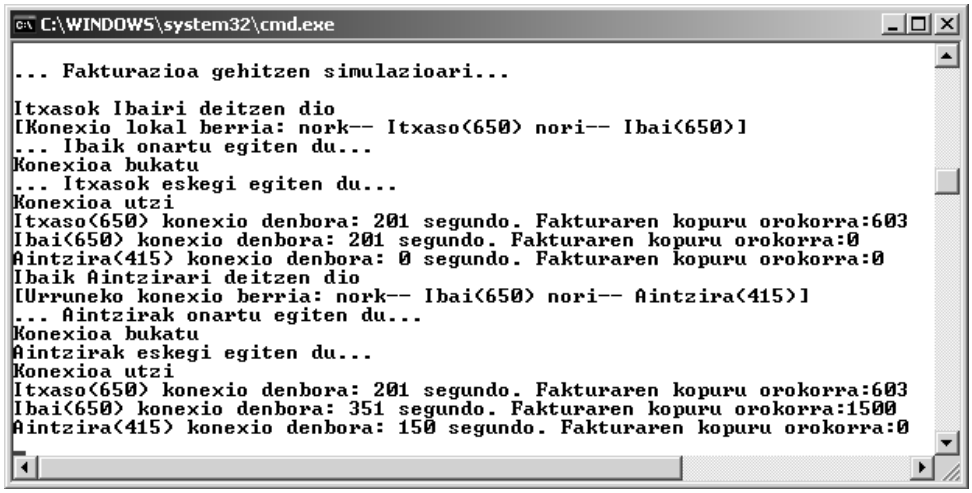
2. Konexioa ixten denean, denbora gelditzen da eta ordaindu beharreko kopurua kalkulatu eta konexioa hasi duen bezeroari esleitzen dio.

```
after(Connection conn): Timing.endTiming(conn) {
  long time = Timing.aspectOf().getTimer(conn).getTime();
  long rate = conn.callRate();
  long cost = rate * time;
  getPayer(conn).addCharge(cost); }
```

3. Bezeroak ordaindu beharreko kopurua kudeatuko duten baliabideak erazagutzen ditu.

```
public long Customer.totalCharge = 0;
      public long getTotalCharge(Customer cust) { return
  cust.totalCharge; }

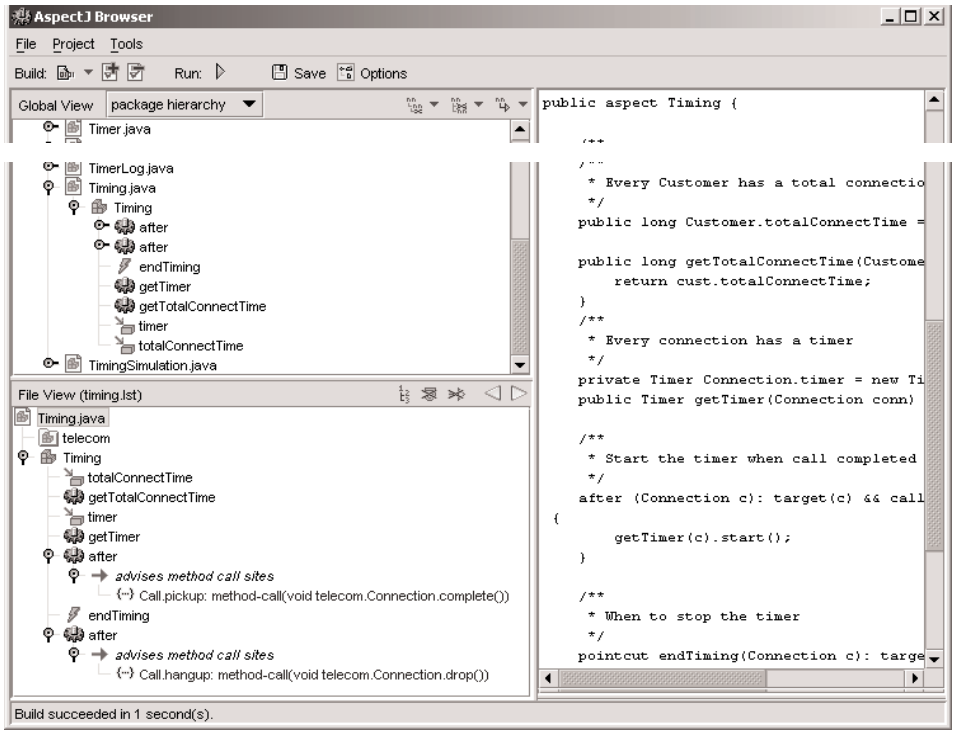
public void Customer.addCharge(long charge){
    totalCharge += charge;}
```



5.7. irudia. Fakturazioa gehitzen simulazioari.

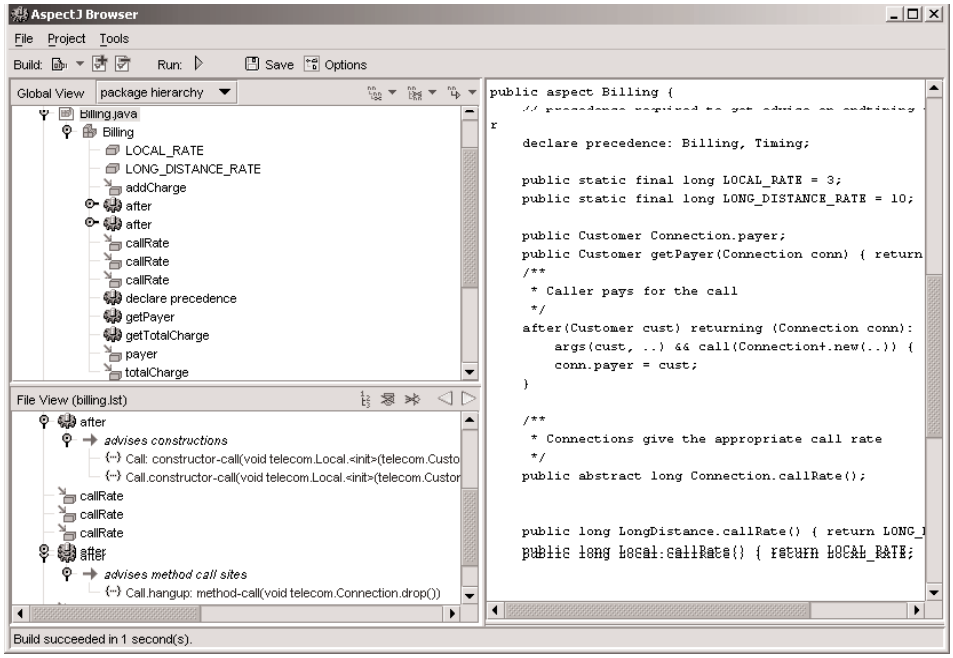
Bukatzeko, simulazio berri bat antolatzen dugu aplikazio eguneratuaren portaera (**5.7. irudia**) bistaratzeko:

```
package telecom;
public class BillingSimulation extends AbstractSimulation {
    public static void main(String[] args){
      System.out.println("\n... Fakturazioa gehitzen
simulazioari...\n");
      simulation = new BillingSimulation();
      simulation.run();}
    /**
     * Print a report of the connection time and the bill for customer
     */
    protected void report(Customer c){
        Timing t = Timing.aspectOf();
        Billing b = Billing.aspectOf();
        System.out.println(c + " konexio denbora: "
                    + t.getTotalConnectTime(c)
                    + " segundo. Fakturaren kopuru orokorra:"
                    + b.getTotalCharge(c));}}
```



5.8. irudia. AspectJ tresnaren interfaze grafikoa (I).

Gainetik bada ere, **5.8. irudia**n eta **5.9. irudia**n azaltzen da nola kudeatzen dituen AspectJ tresnak aplikazioaren zenbait aspektu. Tresnak aspektuak detektatu eta bakoitzaren ezaugarriak kudeatzeko baliabide grafikoak eskaintzen ditu. Hala, aplikazio bat eraikitzeko nahikoa zaigu `Project` menuko `Build` aukera hautatzea eta tresnak, automatikoki, fitxategi guztiak aztertu eta aplikazioa eraikitzen du. Aplikazioa egikaritu nahi badugu aski da `Options` aukera klikatzea eta sistemari adieraztea non bilatu behar duen `main()` metodo nagusia.



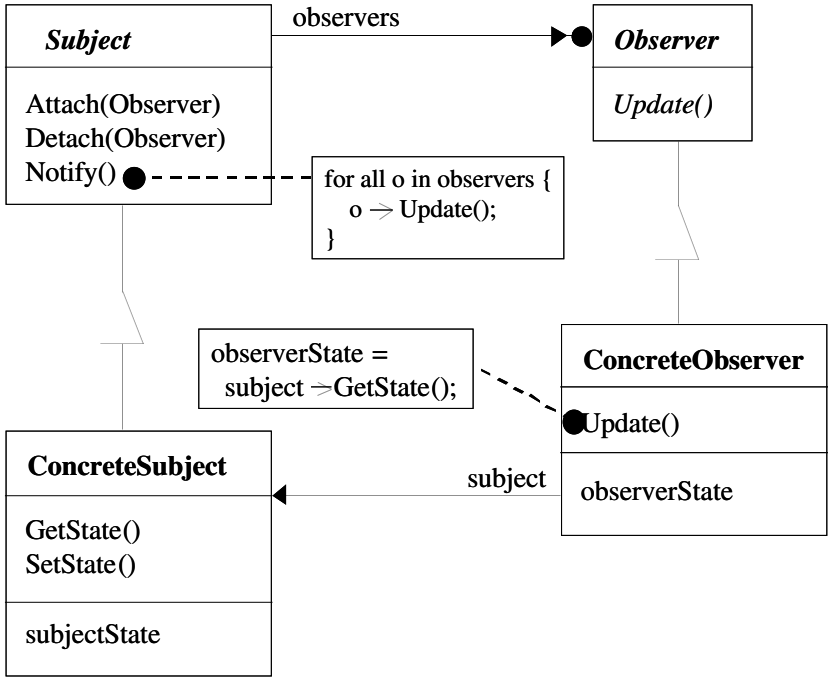
5.9. irudia. AspectJ tresnaren interfaze grafikoa (II).

5.3. ADIBIDE KONPLEXUAK: OBSERVER PATROIA

Diseinu-patroien atzean dagoen ideia nagusia beharbada ez da lar originala ezta oraintsukoa ere. Edozein informatikarik daki informatikaren garapenean abstrakzioaren kontuak zeresan handia izan duela lengoaia eta tresna berrien kontzeptzioan. Abstrakzio mailaren igoerak ikuspegi berri bat sortzen du psikologikoki pertsonengan, arazoak kudeatzeko erak aldatu egiten dira eta paradigma berri bat sortu izan balitzaio bezala interesgune mentalak jauzi egiten du eta dimentsio berri batean kokatzen da. Adibidez, klaseak erabiltzen badira aplikazioaren funtzionalitatea kapsulatu eta inplementatzeko, tentu handiz eta poliki-poliki idatzi beharko ditu klase guztiak programatzaileak. Ostera, klaseak erabili beharrean osagaiak erabiltzen badira, lan pila bat osagaiak kudeatzen dituen edukiontziak gauzatuko du eta programatzailea diseinuaren eta proben arloetan murgildu ahal izango da, zeren programazioaren arlo batzuk automatizatu egingo baitira.

Diseinu-patroiak ere testuinguru batean kokatuta dagoen arazo bati erantzun orokor eta berrerabilgarri bat ematen saiatzen dira. Orokorra, arazoa tipifikatu egin daitekeelako eta berrerabilgarria, arazoa konpontzeko erabiltzen den diseinu informatikoa ez delako ezkontzen domeinu zehatz batekin.

Testuinguru horretan kokatzen da jarraian aztertuko dugun bigarren adibide konplexua. Hain zuzen ere, diseinu-patroi ezagun baten ezaugarriak aurkeztuko ditugu eta, haren osagaiak eta osagaien arteko erlazioak azaleratuz gero, inplementatu egingo dugu aspektuetara zuzendutako ikuspegia erabilita. Patroiak (**5.10. irudia**) bi osagai generiko ditu eta horietako batek ematen dio izena: Observer.



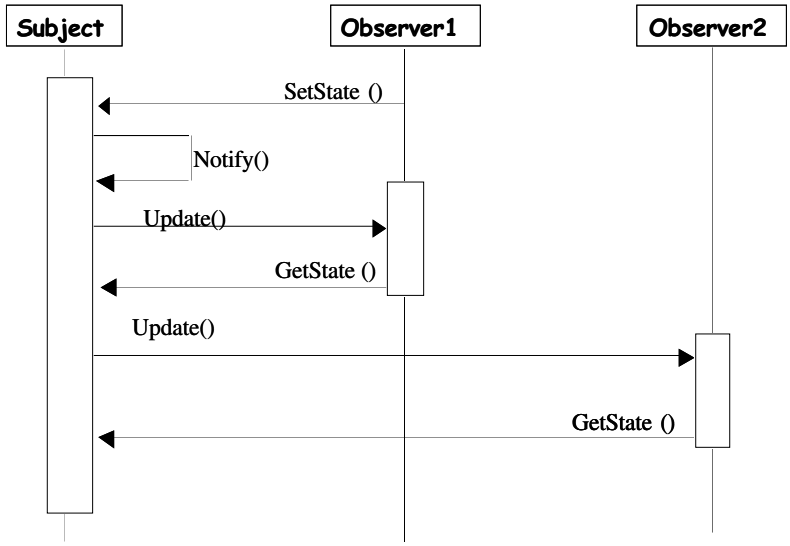
5.10. irudia. Observer patroiaren Klase Diagrama.

Osagai batek begirale (`Observer`) rola jokatzen du patroian eta besteak, ostera, begiratuarena (`Subject`). Bai bata eta bai bestea, interfaze bat bezala inplementatzen dira gehienetan eta aplikazioan, begiratuaren lana beteko duen objektu zehatzak `Subject` interfazea inplementatuko du. Begiralearen rola bete nahi duen objektu zehatzak, ostera, `Observer` interfazea inplementatuko du:

```
interface Subject {
    void addObserver(Observer obs);
    void removeObserver(Observer obs);
    Vector getObservers();
    Object getData(); }
```

```
interface Observer {
    void setSubject(Subject s);
          Subject getSubject();
          void update(); }
```

Observer patroiak objektuen arteko mendekotasunak era egituratuan kudeatzen ditu, horrela, begirale bat objektu begiratu batekin harremanetan jarri nahi bada, esplizituki adierazi beharko dio objektu horren begirale bat izan nahi duela eta gauza bera egin behar du harremanak hautsi nahi baditu. Behin lotura formalizatuta, objektu begiratuak bere objektu begiraleen zerrendan biltegiratzen du eta exekuzio-fluxuan (**5.11. irudia**) egoera-aldaketa bat suertatzen zaionean, jakinarazi (`Notify()`) egiten die begirale guztiei. Begirale bakoitzak egoera berriari ematen dion erantzuna (`Update()`) berezia eta berezitua izaten da gehienetan.

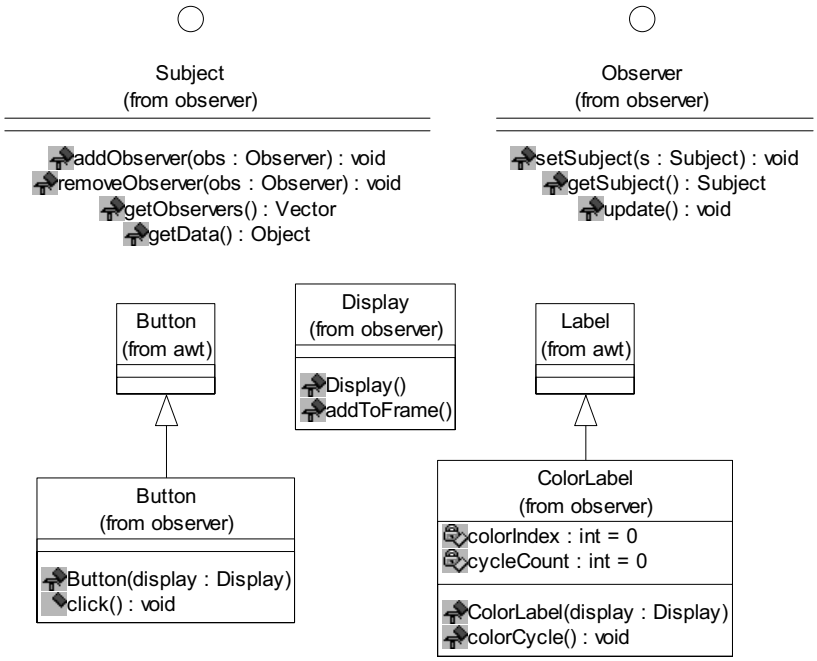


5.11. irudia. Observer patroiaren Sekuentzia Diagrama orokorra.

Adibidea ez nahasteko, jostailuzko domeinu (**5.12. irudia**) txiki batean inplementatuko dugu patroia. Domeinua, bi botoiz eta hiru etiketaz egongo da osatua eta botoiek `Subject` rola jokatuko dute eta etiketek, ostera, `Observer`rena. Bestalde, Javaren teknologiak interfaze grafikoak eraikitzeko hainbat osagai eskaintzen dituenez, horietaz baliatuko gara botoiak eta etiketak eraikitzeko.



5.12. irudia. Aplikazioaren interfaze grafikoa.



5.13. irudia. Aplikazioaren Klase Diagrama.

Aplikazioak izango dituen klase nagusiak azaleratuz gero, aplikazioaren Klase Diagrama (**5.13. irudia**) eraikitzen da diseinua osatzeko. Irudian ikusten den legez, aplikazioaren botoi bakoitza, `Button` motako objektu bat izango da eta haren funtzionalitate gehiena Javaren teknologia beraren `Button` osagaiatik eratorriko da. Era berean, aplikazioaren etiketa bakoitza `ColorLabel` motako objektu bat izango da eta bere funtzionalitate gehiena ere Javaren teknologiaren `Label` osagaiatik eratorriko da.

```
import java.awt.Color;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;

class Button extends java.awt.Button {

    static final Color  defaultBackgroundColor = Color.gray;
    static final Color  defaultForegroundColor = Color.black;
    static final String defaultText = "<<Botoia>>";

    Button(Display display) {
        super();
        setLabel(defaultText);
        setBackground(defaultBackgroundColor);
        setForeground(defaultForegroundColor);
        addActionListener(new ActionListener() {
```

```
                public void actionPerformed(ActionEvent e) {
                    Button.this.click();
                }
            });
        display.addToFrame(this);
    }

                        public void click() {}
}
                                oOo
import java.awt.Color;
import java.awt.Label;

class ColorLabel extends Label {

    ColorLabel(Display display) {
        super();
        display.addToFrame(this);
    }

    final static Color[] colors = {Color.red, Color.blue,
                                   Color.green, Color.magenta};
    private int colorIndex = 0;
    private int cycleCount = 0;
    void colorCycle() {
        cycleCount++;
        colorIndex = (colorIndex + 1) % colors.length;
        setBackground(colors[colorIndex]);

              setText("" + cycleCount);

          }}
```

Interfaze grafikoaren elementu guztiak `Panel` motako `Display`[49] objektu batekin kudeatzen dira eta objektu horren erreferentzia pasatzen zaie `Button` eta `ColorLabel` motako objektuen eraikitzaileei:

```
public class Demo {
    public static void main(String[] args) {

        Display display = new Display();
        Button b1 = new Button(display);
        Button b2 = new Button(display);
        ColorLabel c1 = new ColorLabel(display);
        ColorLabel c2 = new ColorLabel(display);
        ColorLabel c3 = new ColorLabel(display);
```

49. `Display` motako objektua `Panel` motako objektutik eratortzen da eta bere funtzionalitate guztia bereganatzen du.

```
        b1.addObserver(c1);
        b1.addObserver(c2);
        b2.addObserver(c3);
    }}
```

`Demo` klasearen iturburu-kodearen azken hiru lerroek zehazten dute nola erlazionatzen diren botoiak eta etiketak, hau da, lehenengo botoiak (`b1`) bi begirale izango ditu eta, bigarrenak (`b2`), ostera, bat. Hala, erabiltzaileak lehenengo botoia zapaltzen duenean, lehenengo bi etiketei transmitituko zaie botoiaren egoera-aldaketa, eta bigarren botoia zapaltzen duenean, hirugarren etiketa ohartaraziko dugu.

Horratx, aplikazioaren testuinguru orokorra: Observer patroiaren Subject eta Observer osagaiak eta aplikazioaren botoiak eta etiketak. Baina, iturburu-kodea zehatz-mehatz aztertzen badugu, berehala datozkigu ahora honako galdera hauek: nola eta non gauzatzen da elementu desberdinen arteko komunikazioa? nola daki botoi batek objektu begiratua dela? eta, etiketa batek objektu begiralea dela? Aplikazioaren aspektuek hala diotelako:

```
abstract aspect SubjectObserverProtocol {

    abstract pointcut stateChanges(Subject s);

    after(Subject s): stateChanges(s) {
        for (int i = 0; i < s.getObservers().size(); i++) {
            ((Observer)s.getObservers().elementAt(i)).update();
        }
    }

    private Vector Subject.observers = new Vector();
    public void   Subject.addObserver(Observer obs) {
        observers.addElement(obs);
        obs.setSubject(this);
    }
    public void   Subject.removeObserver(Observer obs) {
        observers.removeElement(obs);
        obs.setSubject(null);
    }
    public Vector Subject.getObservers() {  return observers; }

    private Subject Observer.subject = null;
    public void     Observer.setSubject(Subject s) {  subject = s; }
    public Subject  Observer.getSubject() {  return subject; }  }
                                oOo
import java.util.Vector;

aspect SubjectObserverProtocolImpl extends SubjectObserverProtocol {

    declare parents: Button implements Subject;
```

```
public Object Button.getData() {  return this; }

declare parents: ColorLabel implements Observer;
public void    ColorLabel.update() {
    colorCycle();
}
pointcut stateChanges(Subject s):
           target(s) &&
           call(void Button.click());  }
```

Lehenengo aspektuak Observer patroiaren protokoloa inplementatzen du `Subject` eta `Observer` osagaien ezaugarriak erazagutuz eta inplementatuz. Halaber, interesgune (`stateChanges()`) abstraktu bat erazagutzen du eta berari dagokion `after` erako portaera. Portaerak objektu begiratuaren begiraleen `update()` metodoei deitzen die objektuaren egoera aldatzen den bakoitzean.

Bigarren aspektuak, ostera, Observer patroiaren protokoloa hedatu egiten du azken ukituak emanda aplikazioaren osagaiei. Esaterako, hemen erazagutzen da `Button` motako objektuak Subject rola jokatuko duela patroian eta `ColorLabel` motako objektuak, Observerrena. Halaber, aurreko aspektuan erazagututako interesgune (`stateChanges()`) abstraktua hemen inplementatzen da. Horrela, aplikazioaren erabiltzaileak botoi bat zapaltzen duen bakoitzean, botoi horren begiraleek `update()` metodoa egikarituko dute. Kontu egin metodoaren gorputzak barne-dei bat gauzatzen duela `colorCycle()` metodoa egikaritzeko. Azken metodo horrek honako bi gauza hauek egiten ditu soilik: etiketaren kolorea aldatu eta klik-kontagailu bat eguneratu.

5.4. LABURPENA

Objektuetara zuzendutako paradigmak bezeroaren betekizunen funtzionalitatea klaseetan kapsulatzen du eta klaseen arteko harremanak hainbat erlazio mota erabilita kudeatzen ditu. Erlazio mota horien artean herentziak eta asoziazioak funtsezko zeregina izango dute klaseak egituratzerakoan. Batzuetan, ordea, domeinuaren funtzionalitate bat ez da ondo adierazten klase zehatz batzuk erabilita, barreiatuta aurkitzen da Klase Diagraman zehar. Jakin badakigu klase batean funtzionalitatearen zati bat egitura dezakegula baina ez guztia. Harantzago, beste klase batek funtzionalitatearen beste alde bat lantzen lagun diezaguke. Hori guztia objektuetara zuzendutako ikuspegia erabilita bakarrik programatuko bagenu, funtzionalitatea inplementatuko lukeen iturburu-kodea sakabanatuta geldituko litzateke han-hemenka.

Aspektuetara zuzendutako ikuspegiak inplementatu behar diren funtzionalitate txiki horiek guztiak kapsulatu egiten ditu aspektuetan. Hala programatzaileak toki bakarrean biltegiratzen du funtzionalitatearen semantika guztia eta ez da ibili behar Klase Diagraman zehar funtzionalitatearen xehetasunak bilatzen. Gure

testuinguruan, AspectJ tresna informatikoak automatikoki hezurmamitzen du lan hori. Aspektuaren integrazioa Klase Diagraman eta inplementazioan automatikoki eta era gardenean gauzatzen da.

5.5. OINARRIZKO BIBLIOGRAFIA

Colyer, A.; Clement, A.; Harley, G. eta Webster, M. (2005): *Aspect-Oriented Programming in Eclipse with AspectJ and the AspectJ Development Tools*, Addison-Wesley.

Gradecki, J. eta Lesiecki, N. (2003): *Mastering AspectJ*, Wiley.

Jacobson, I. eta Pan-Wei Ng. (2005): *Aspect-Oriented Software Development with Use Cases*, Addison Wesley.

Kiselev, I. (2002): *Aspect-Oriented Programming with AspectJ*, Sams.

Laddad, R. (2003): *AspectJ in Action*, Manning Publications.

Miles, R. (2005): *AspectJ Cookbook*, O´Reilly Media.

Pawlak, R.; Retaillé, J-P. eta Seinturier, L. (2005): *Foundations of Aop for J2ee Development*, Apress.

5.6. HELBIDE INTERESGARRIAK

Zer dago bertan?	Web gunea
AspectJ tresnaren egoitza	http://eclipse.org/aspectj/
Aspect-Oriented Software Development biltzarren web ataria	http://aosd.net/
Eclipse tresnaren AspectJ Development Tools tresna	http://www.eclipse.org/ajdt/
The AspectJ™ Programming Guide	http://eclipse.org/aspectj/doc/progguide/
I want my AOP!, Part 1. Separate software concerns with aspect-oriented programming (Artikulua)	http://www.javaworld.com/javaworld/jw-01-2002/jw-0118-aspect_p.html
I want my AOP!, Part 2 Learn AspectJ to better understand aspect-oriented programming (Artikulua)	http://www.javaworld.com/javaworld/jw-03-2002/jw-0301-aspect2.html
I want my AOP!, Part 3 Use AspectJ to modularize crosscutting concerns in real-world problems (Artikulua)	http://www.javaworld.com/javaworld/jw-04-2002/jw-0412-aspect3.html
Introduction to Aspect-Oriented Programming (Artikulua)	http://www.onjava.com/lpt/a/4448
Second-generation aspect-oriented programming (Artikulua)	http://www.javaworld.com/javaworld/jw-07-2004/jw-0705-aop_p.html
Elkarrizketa Jonas Bonerekin	http://www.javahispano.org/text.viewer.action?file=jonas_es

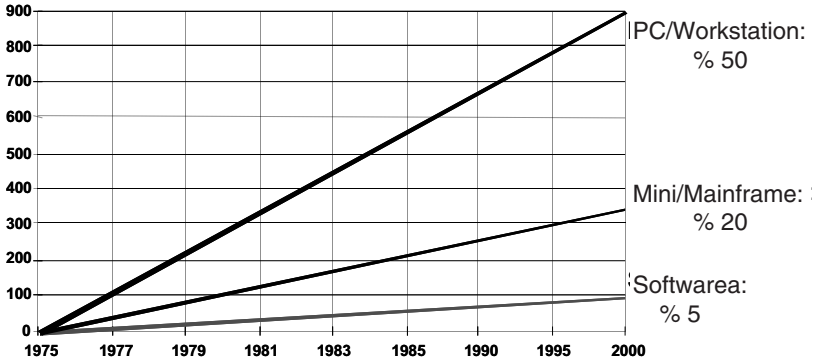
6. Informazio-sistemen analisi eta diseinua

6.1. SARRERA

Software ona eraikitzea zeregin konplexua eta zaila da. Softwarea eraikitzea ez hainbeste. Hasieran, analista bezeroarekin jartzen da harremanetan domeinuaren nondik norakoak ulertzeko. Bezeroak domeinua ondo egituratuta eduki dezake eta erraza egiten zaio domeinuaren prozesuak zehaztea eta informazioaren ibilbidea zein den azaltzea. Zoritxarrez, egoera idiliko hori gutxitan gertatzen da eta ohikoagoa izaten da bezeroak argi ez izatea zer lortu nahi duen aplikazio informatikoarekin.

Egoera motak asko eta askotarikoak izaten dira. Sarritan, garatu nahi den aplikazio informatikoa berria da eta dena egin behar da hasiera-hasieratik. Beste batzuetan, ostera, bezeroak badu aplikazio informatiko bat, baina aldatu edota eraberritu nahi izaten du. Gainera, ez du argi zer arraio den Internet delako hori, baina susmatzen du horretara bideratu behar dituela etorkizunean enpresa eta eraiki nahi duen aplikazio informatikoa.

Horregatik, ez da harritzekoa irakurtzea ButlerBloor Computer Research erakundeak aspaldian argitara eman zuen txostenean agertzen diren zenbakiak: proiektuen % 2 ez da sekula santan bukatzen, % 47 bukatzen da baina gero ez da erabiltzen, % 19 erabiltzen da baina zuzenketa sakonak gauzatu eta gero, % 5 aldaketak eginda jartzen da operatibo eta % 3 bakarrik garatzen da era itxurosoan. Beraz, ez da inor txundituko **6.1. irudia**n agertzen dena ikusita. Irudiak jasotzen du azken urteetan salneurri/eraginkortasun ratioa nola hazi den hiru ingurunetan. Ageri-agerian gelditzen da hardwarearen garapenaren abiadura eta eraginkortasuna askoz ere hobeak izan direla softwarearenak baino.



6.1. irudia. Salneurri/eraginkortasun ratioa.

Software ona eraikitzeko hainbat urrats bete behar dira eta ez edozelan. Erakundeak, teknologia eta software-prozesu egokiak izan behar ditu finkatuta, softwarearen bizi-zikloan sortuko diren erroreak detektatu eta kudeatzeko.

Analisian sortu den errore bat hainbat urrats geroago detektatzen bada, errore horren konpontze-kostua handia eta desatsegina izango da. **6.2. irudia**ren zenbakiek bizi-zikloan zehar gertatzen diren desbideraketen kostuak aztertzen dituzte. Lehenengo zutabeak, bizi-zikloan zehar gertatzen diren kostu-desbideraketak jasotzen ditu eta nahiko ezagunak dira informatika arloan. Arazoaren gakoa ondorengo hiru zutabeetan dago. Lehenengoak, erroreen jatorria identifikatzen du eta hemen dago arazoaren argibideetako bat: erroreen ia % 90, betekizunen analisian eta aplikazioaren diseinuan sortzen da.

Softwarearen garapenaren urratsak	Kostu-desbid.	Sartutako erroreak	Aurkitut . erroreak	Konpontze-kostua
Betekizunen analisia	% 5	% 55	% 18	1
Diseinua	% 25	% 30	% 10	1 - 1.5
Kodeketa	% 10			
Integrazioa eta testa	% 50	% 10	% 50	1 - 5
Egiaztapena/Dokumen.	% 10			
Funtzion./Mantenua		% 5	% 22	10 - 100

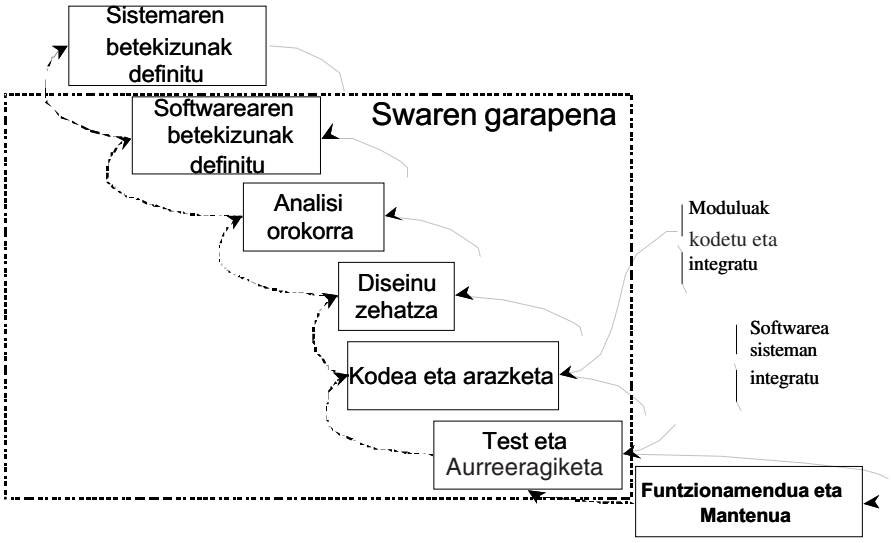
6.2. irudia. Softwarearen garapenaren kostu-desbideraketak.

Zer gertatzen da alboko zutabearekin? Zenbakiak aztertzen baditugu, berehala konturatzen gara erroreak bizi-zikloaren azken urratsetan daudela, hau da, aplikazioa probatzen hasten garenean, erroreen % 50 *hemen nago ni* aldarrikatzen hasten da eta gaitzerdi hor aurkitzen baditugu; funtzionamenduan edota mantenuan agertzen badira, alboko zutabera begiratu besterik ez dago jakiteko konpontze-kostua zenbat handitzen den. Agian, egoera hori ez da errepikatzen software-proiektu

informatiko guztietan zehatz-mehatz, baina nago, zenbakiak antzekoak izango direla gehienetan.

Softwarearen garapena korapilatsua bada, analistak software-prozesu finkatu bat, teknologia eraginkor bat, analisi eta diseinurako metodo zehatz bat eta bizi-ziklo ezagun bat erabiliko ditu software-produktua eraikitzeko. Lau arloak kateatuta daude ia beti erakunde informatiko gehienetan eta zaila izaten da ikusten bizi-ziklo aurreratu bat software-prozesu makal batekin edo teknologia koskor bat analisi eta diseinu kalitatezkoetan. Zoritxarrez edo zorionez, arlo guztiak paraleloan garatzen dira erakundean eta oso lotuta daude erakunde[50] beraren bizi-zikloarekin.

Erakundea gaztea denean, asko ikasi behar du oraindik, baina kemena du horretarako. Porrota ekiditeko, hasieran, softwarea garatzeko aplikatuko duen bizi-zikloa ur-jauzi bizi-zikloa (**6.3. irudia**) izango da. Badaki bizi-ziklo sinplea dela, baina badaki jakin ere, buruhausteak uxatuko dituela bide batez, arlo horretan. Bizi-ziklo horretan urratsak sekuentzialki betetzen dira bata bestearen atzetik eta urrats bakoitzaren irteera hurrengoaren sarrera izango da. Ur-jauzian bezala, informazioa (ura) mailaz maila jausten da inor eta ezer zipriztindu gabe. Zoritxarrez egoera idiliko hori ez da hain goxoa izaten, eta bidean, informazioa galdu, zikindu edota ahantzi egiten da eta amuarrainak bezala, gora jo behar da informazio eske. Igo behar den maila bakarra bada, gaitzerdi, jauzi egin eta aurreko mailan jartzen gara; aitzitik, mailak igo behar baditugu, berriro, **6.2. irudia**n aztertu dugun arazoaren aurrean aurkituko gara.

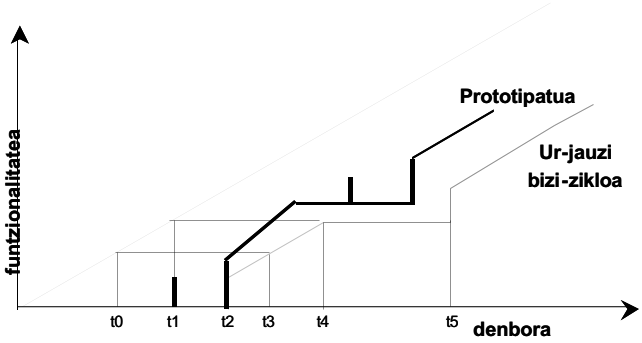


6.3. irudia. Ur-jauzi bizi-zikloa.

50. Erakundeak, denok bezala, jaio, hazi, heldu eta hil egiten dira eta askotan gaztaroko grina faltan botatzen da zahartzaroan, batez ere, aldaketak gauzatu behar direnean.

Badago beste ezaugarri bat softwarearen garapenean exogenoa dena eta erakunde informatikoak aurre egin behar diona: bezeroak edo erabiltzaileak aplikazio informatikotik espero duen funtzionalitatea. Askotan handia izaten da eta asmo irudikatuak ez badira finkatuta gelditzen betekizunetan, nekez izango da jasoa gero iturburu-kodean.

6.4. irudiak aipatutako egoera islatzen du. Irudian, bizi-ziklo mota bi agertzen dira: ur-jauzi bizi-ziklo ezaguna eta prototipatua. Erdiko marrak lortu nahi den funtzionalitate birtuala islatzen du. Ikusten den legez, eta ur-jauzi bizi-zikloarekin ari gara, bezeroak t_0 unean nahi zuen funtzionalitatea t_3–an lortzen da, atzerapenarekin beraz. Norena ote errua? Askotan, informatikariaren jauregira begiratuko beharko da gaizkilearen bila baina halaber, beste askotan, bezeroari betekizunen azterketa erakutsi beharko zaio galdetzeko ea non dauden zehaztuta berak aipatzen dituen balizko betekizun horiek. Horregatik, komeni izaten da sistemaren eta softwarearen betekizunak txosten batean jaso badira, bertan esaten dena hitzartzea. Zalantzak badaude edota informatikariak erabakitzen badu bezeroak eman dion edo eman diezaiokeen informazioa urria dela, hobe du prototipo bat eraikitzea lehenbailehen, falta zaion informazioa, betekizun mailan, eskuratzeko.



6.4. irudia. Aplikazioaren funtzionalitatearen bilakaera.

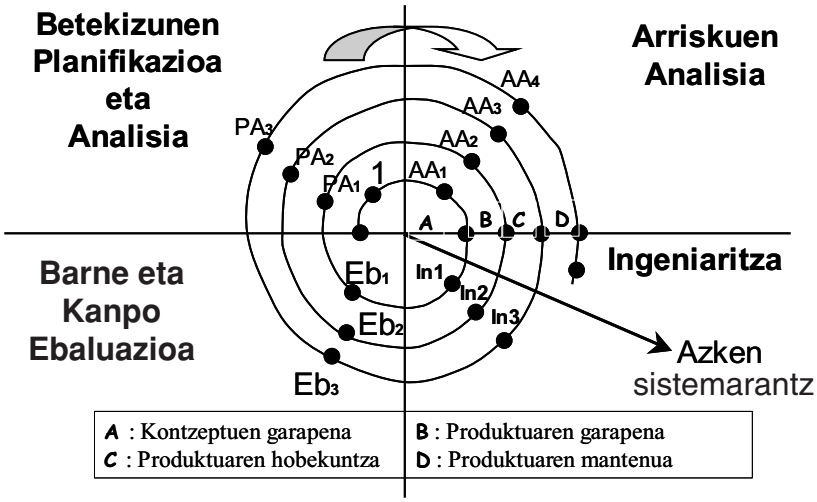
Estrategia horrek berehalako eragina izango du bizi-zikloan edo izan beharko luke gauzak ondo egiten badira: une batean (t_1) funtzionalitate maila asko igotzen da eta geroago tarte (t_2-t_1) batean badirudi ez dela ezer egiten. Sekuentzia hori ezaguna da prototipo bat eraikitzen denean bezeroaren betekizunak eskuratzeko. Ondorioak t_2 unean azaleratuko dira, prototipatuak funtzionalitate handiagoa emango digu analisiaren hasieran eta, abantaila horrek, bizi-ziklo osoan zehar iraungo du.

Sistemaren betekizunekin bukatuz gero, bizi-zikloaren bilakaerak hazkunde (t_2-t_4) nabaria ezagutuko du oraindik, zeren aplikazioaren analisiak eta diseinuak funtzionalitate berriak azaleratuko baitituzte. Diseinu zehatza bukatuz gero, aplikazioaren programazioak (t_4-t_5) ez dio funtzionalitate berririk itsatsiko bizi-zikloari eta haren hazkunde-ratioaren grafikoa erabat laua izango da. Denbora

aurrera doa baina kodeketak, arazketak eta aplikazioaren mantenuak ez diote funtzionalitate berririk ekartzen aplikazio informatikoari.

Unean uneko teknologiak zeharo baldintzatzen du bizi-zikloaren egituraketa eta ez da gauza bera ur-jauzi bizi-zikloa egun aplikatzea edo orain dela hamar urte. Egun, hibridazioa da nagusi ur-jauzi bizi-zikloa aplikatzerakoan. Batzuetan, ur-jauzi bizi-zikloa oinarri gisa hartuta, garapen inkrementalean[51] bat gauzatzen da eta hasieran prototipo bat eraikitzen dugu betekizunak eskuratzeko. Beste batzuetan, ostera, garapena ez da inkrementala ebolutiboa baizik. Prototipo batekin hasten gara eta garapen-tresna aurreratu bat erabilita, software-produktua hazten joaten da produktuak berak elikatuz produktuaren hurrengo bertsioa. Aukerak asko dira eta erakundeak ondo neurtu beharko ditu bere giza baliabideak eta baliabide teknologikoak, bizi-ziklo egokia aukeratzeko orduan.

Arazoak ekiditeko, tentagarria izan daiteke erakundearentzat espiral bizi-zikloa (**6.5. irudia**) aplikatzea enpresa sortu eta berehala, neurtu gabe erakundearen heldutasuna zein den eta gai ote den horrelako bizi-ziklo konplexu bat aurrera eramateko.



6.5. irudia. Espiral bizi-zikloa.

Ez dago zalantzarik bizi-zikloa oso aberatsa dela, garapen informatikoaren arlo gehienak lantzen baititu kontzeptuen garapenetik produktuaren mantenura. Oinarrian, bizi-ziklo horretan, aplikazio informatikoaren lau ezaugarriak lantzen dira: *kontzeptuen garapena*, *produktuaren garapena*, *produktuaren hobekuntza* eta

51. Garapen inkrementalean aplikazioa moduluetan zatitzen da eta modulu bakoitza miniaplikazio bat bezala garatzen da. Miniaplikazio bakoitzak bere betekizun, diseinu eta inplementazio partikularrak izango ditu. Batzuetan, betekizunen azterketa orokorra hasieran egiten da eta bakarra izaten da aplikazio osoarentzat, horrela, integrazio-arazoak hasieratik ekiditen dira.

produktuaren mantenua. Ezaugarri bakoitza garatzeko begizta metodologiko bera erabiltzen da behin eta berriz: hasieran, aztertzen ari garen ezaugarriaren betekizunen planifikazio eta analisia gauzatzen da. Bigarren urratsean, arriskuen analisia egiten da, erabakitzeko planifikatutakoa bideragarria den ala ez, eta merezi duen aurrera jarraitzea begiztarekin. Aurrera jarraitzea erabakitzen bada, planifikatutakoari ingeniaritza aplikatzen zaio emaitzak eskuratzeko. Emaitzek ez dute zertan izan emaitza informatiko gordinak, hau da, iturburu-kodea. Hori, azken sistemarantz hurbiltzen ari garenean gertatuko da. Hasierako urratsetan, emaitzak kontzeptualagoak izango dira: betekizunen azterketa, analisiaren prozesuen eta datuen egitura, eta diseinuaren nondik norakoa.

Urrats bakoitzaren ingeniaritza bukatuta, erakunde informatikoan eta bezeroarekin aztertzen dira emaitzak, ikuspegi batua edukitzeko. Hori eginez gero, ezaugarria birfindu egiten da edo ezaugarriz aldatzen da, espiralaren hurrengo begizta martxan jarriz.

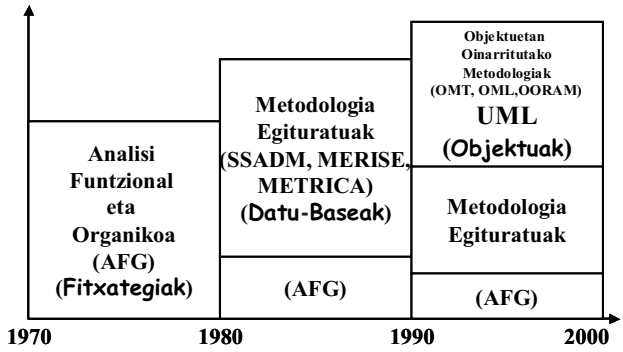
6.2. INFORMAZIO-SISTEMEN ANALISI ETA DISEINU EGITURATUA

Agian anakronikoa dirudi, egun, analisi eta diseinu egituratuaz hitz egiteak, joera eta interesgunea objektuetan finkatuta daudenean. Baina nago, nondik gatozen jakitea ezinbestekoa dela nora joan nahi dugun erabakitzeko.

Urteetan zehar, softwarearen eraikuntza, informazioaren biltegiratze erarekin erabat lotua egon da softwareak izan duen bilakaera historikoan. **6.6. irudia**k laburbildu eta sintetizatzen du azaldu nahi duguna. Hasiera batean, analisi funtzionala eta organikoa nagusi zenean, informazioa fitxategietan biltegiratzen zen hala eskatzen baitzuen garai haietako teknologiak. Orduan, analisia eta diseinua eskuz egiten zen eta ordenagailu handietan inplementatzen zen aplikazio informatikoa. Informazioaren atzipen era erabakitzea izaten zen arazo larriena. Fitxategia sekuentziala bazen, aplikazioaren atzipen-abiadura bermatuta gelditzen zen baina atzipen era baldintzatuta. Fitxategi indexatua[52] erakargarria izaten zen baina, noiz aukeratu ausazkoaren[53] ordez?

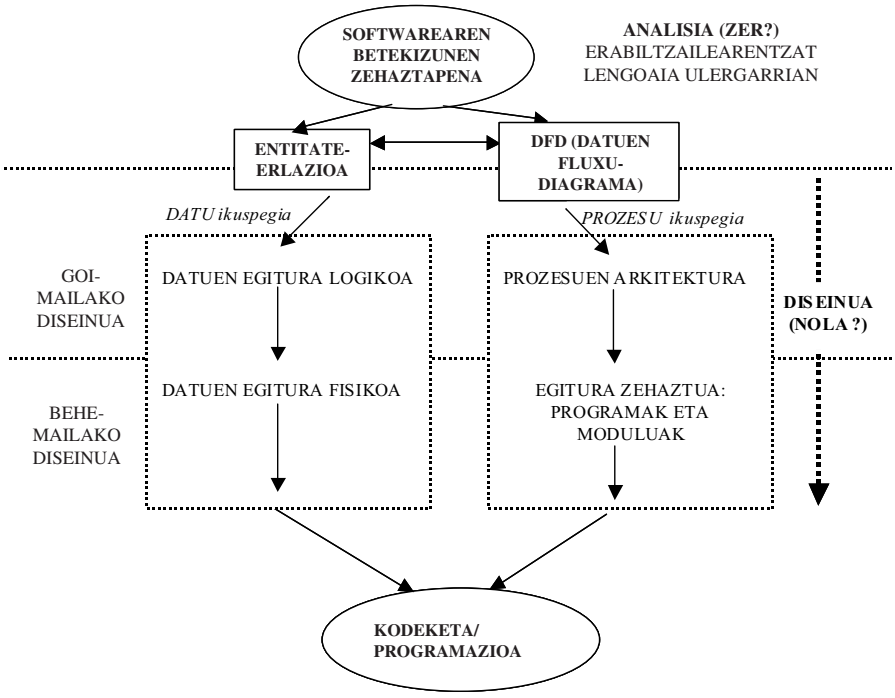
52. *Fitxategi indexatua*: Fitxategiko edozein erregistro gako batetik abiatuta atzitzeko bidea ematen duen taula bat (indize deitua) duen fitxategia.

53. *Ausazko atzipeneko fitxategia*: Bigarren mailako memorian gordetako fitxategi mota, erregistroak zuzenean atzitzeko bidea ematen duena, atzipen-denbora erregistroak fitxategian daukan posiziotik independentea izanik.



6.6. irudia. Softwarearen garapenaren bilakaera historikoa.

Laurogeiko hamarkadan, datu-baseen arrakasta gertatu zen eta horrek erabat aldatu zuen egoera: analisia eta diseinua bete-betean bideratu ziren biltegiratze eta antolakuntza mota berrira. Informatikan sarritan gertatzen den legez, metodo bateratu eta estandar ireki bat garatu ordez, hainbat metodo desberdin jaio ziren. Norbaitek aniztasuna ona dela esango du, eta hala da, aniztasunak aberastasuna dakarrenean baina, aniztasuna interes komertzial eta monetarioekin nahasten denean, susmagarria izaten da.



6.7. irudia. Analisi eta diseinu egituratuaren ezaugarri nagusiak.

Eta hala gertatu zen, hainbat eta hainbat metodo merkaturatu ziren eta beste hainbeste, CASE[54] tresna, liburu, ikastaro eta aholkularitza-lan saldu ere. Sakonean, denek saltzen zuten produktu (**6.7. irudia**) bera, baina bezeroa nahastu eta ez konturatzeko, azaleko aldaketak egiten ziren.

Hori izan zen Ameriketako Estatu Batuetan jarraitu ohi zen bidea. Europan, zertxobait aldatu zen ikuspegia. Herrialdeetako gobernuetako administrazioetan, software asko kontratatzen zen eta kontratazio-prozesuak homologatua eta bateratua izan behar zuen. Beraz, estandarrak behar ziren eta hala egin zen. Frantzian, MERISE metodologia garatu zen eta Erresuma Batuan, ostera, SSADM (Structured Systems Analysis and Design Methodology). Espainian, beranduago eta aurreko bi aitzindarien lana xurgatuz, METRICA metodologia eraiki zen, MAP[55] erakundeak bultzatuta.

Aldaketa txikiak gorabehera, sakon-sakonean, metodologia guztiek **6.7. irudia**n agertzen den egitura orokorrari jarraitzen diote. Softwarearen betekizunen azterketarekin analistak zer egin behar den zehazten du. Domeinuan aplikatzen diren kudeaketa-arauak antzematen ditu eta bezeroaren nahiak igartzen saiatzen da. Lehen urrats horretan, bezeroa analistarengana hurbiltzen da domeinuaren ezaugarriak azalarazteko. Enpresaren mekanizazioa egin nahi bada, bezeroak nola dagoen egituratua azaldu beharko dio: zein diren sailak eta nola funtzionatzen duten prozesu mailan. Batzuetan, bezeroak berak ere ez ditu lar formalizatuta prozesuak eta orduan lana astuna eta zaila bihurtuko da. Analistak argitu egin beharko dio, horrelako egoera batean, aplikazio informatikoak agian hobetuko diola enpresaren kudeaketa baina, lehenengo eta behin adierazi beharko dio, prozesuak ondo zehaztu eta formalizatzea ere komeni zaiola.

Gauza bera gertatzen da agiriekin eta agiriek jasotzen duten informazioarekin. Informazio-fluxuak ondo identifikatuta egon behar du eta agiri bakoitzaren barnean dagoen informazio guztiak sailkatuta eta lotua egon behar du domeinuko prozesuekin.

Beraz, informazioaren eta prozesuen arloak aurretik ondo definituta badaude, zeharo erraztuko da analistaren lana eta aplikazioaren garapena eraginkorragoa izango da.

Proiektuaren fase honetan, egoera askoren aurrean egon daiteke analista. Batzuetan, bezeroak aurkeztu dion domeinua erabat mekanizatu beharko du goitik behera eta ez da baldintzatua egongo aurretik egin denarekin. Beste batzuetan,

54. *Computer Aided Software Engineering*: Aplikazio informatikoaren analisia eta diseinua mekanizatzeko erabiltzen den tresna informatiko aurreratua. Aplikazioa proiektu gisa antolatzen da eta aplikazioaren ikuspegi ortogonal bat eraikitzen da. Tresnak aplikazioaren elementu bakoitzaren ikuspegi bateratua izango du nahiz eta, elementu hori, analistak eta diseinatzaileak toki desberdinetan erabili prozesuan zehar.

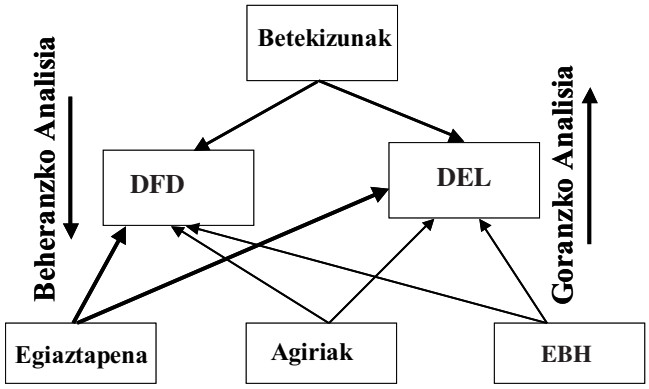
55. Ministerio para las Administraciones Públicas.

ostera, domeinuaren zati batzuk partzialki mekanizatuta daude eta, aurretik egindakoarekin integratu beharko du haren ebazpena. Dena dela, bai batean eta bai bestean, prozesuetan sailkatu beharko du informazio-sistema guztia, prozesu bakoitzari bere informazio-egitura itsatsiz.

Diseinatzaileak prozesuen ikuspegia landu nahi badu, Datuen Fluxu Diagrama (DFD) erabiliko du prozesuen arkitektura zehazteko. Datuen Fluxu Diagramak sistemaren barnean informazio-fluxua nola mugitzen den egituratzen du: informazioa nola sartzen den eta ateratzen den, sistemara eta sistematik; nork aldatzen duen informazioa eta non biltegiratzen den.

Kanpo-entitateak sistemarekin harremanetan jartzen diren elementuak izango dira eta normalean, grafikoki, obalo-itxura batekin marrazten dira. Prozesuak laukizuzen batekin marrazten dira eta hiru ezaugarri izaten dituzte: identifikadorea, non gauzatzen den prozesua eta prozesuaren deskribapen txikia. Bukatzeko, informazioa sisteman biltegiratzeko, fitxategiak erabiltzen dira. Hiru elementu horiekin, prozesuen informazio adierazgarriena egituratuta gelditzen da.

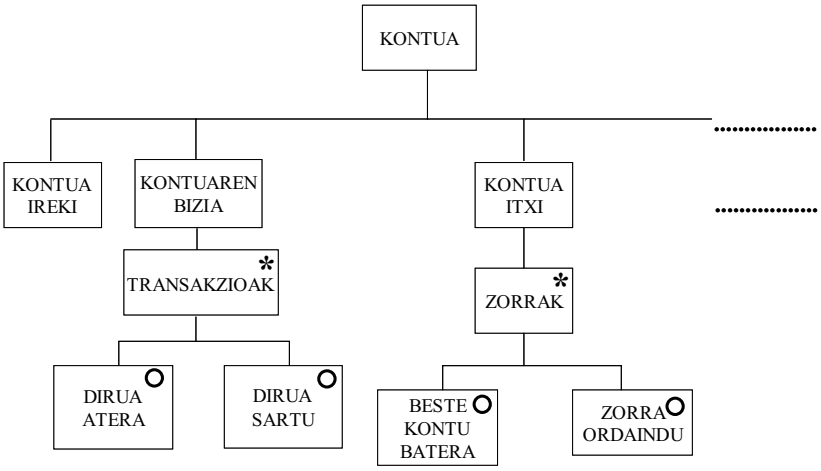
Diseinua bukatzeko, datuen ikuspegia lantzen du diseinatzaileak eta horretarako, Datuen Egitura Logikoa (DEL) eraikitzen da. Datuen Egitura Logikoa (**6.7. irudia**) eraikitzeko, datu-baseetan erabiltzen den eredu erlazionalaz baliatuko da diseinatzailea datuak egituratzeko eta haien arteko erlazioak adierazteko. Agirietan agertzen den informazioak asko lagunduko du entitateak identifikatzen eta entitate horietako atributuak zehazten.



6.8. irudia. Ezaugarrien egiaztapena.

Lehenago edo beranduago, entitateetan agertzen den informazioa egiaztatu (**6.8. irudia**) egin beharko da DFDan agertzen denarekin. Nahitaez eta ondo burutu bada diseinua, DFDan agertzen diren fitxategiek eta fitxategien eremuek islatuak egon behar dute DELaren entitateetan, sistemaren prozesuetatik aplikazioa programatzera igarotzeko eta datuen egitura logikotik datuen egitura fisikoa lortzeko, hau da, diseinuaren behe-mailara igarotzeko.

Entitatearen Biziaren Historiak (EBH) egiaztapenaren fidagarritasuna hobetzen laguntzen du. Teknika horrek, entitateak izango dituen egoera-aldaketak jasotzen ditu sisteman gertatuko diren gertaerak aztertuta. Lehendabizi, gertaerak identifikatzen dira eta matrize bat erabilita gertaerak eta entitateak ipintzen dira harremanetan. Gertaera ipini duen lerroan, diseinatzaileak adierazi behar du gertaera horrek zein entitatetan duen eragina. Matrizea eraikita, diseinatzaileak entitate bakoitzarentzat bere EBHa eraikitzen du gertaerak grafikoki egituratuz. **6.9. irudia**k banketxe batek nola tratatuko dituen bezeroen kontuak adierazten du. Laukizuzenak, goiko aldean, biribil txiki bat badu baldintza-egoera bat adierazten du EBHan. Aitzitik, goiko aldean asterisko txiki bat badu, begizta baten aurrean gaudela ematen du aditzera.



6.9. irudia. Entitatearen Biziaren Historia.

Metodologia egituratuen atal hau bukatzeko, adibide bat aztertuko dugu aurkeztu ditugun teknika guztien erabilera azalduko duena.

Adibidearen helburua enpresa bateko proiektuetan erabiltzen diren lan-bonuen kontrola kudeatzea da. Horretarako, sistema informatiko bat eraiki nahi da prozesua mekanizatzeko. Proiektuaren zereginak (atazak) eta bere inguruko informazioa gorde egiten da proiektua zenbat desbideratu den jakiteko. Laburbilduz, proiektuaren urrats guztiak eta haien egituraketa nolakoa izango den etorkizunean aztertuko dugu.

Sistema informatikoaren erabiltzaileak proiektuko pertsonala eta proiektuaren arduraduna izango dira. Proiektuko parte-hartzaile bakoitzak egunero bere lan-bonua sartu behar du sistema informatikoan adieraziz zenbat ordu sartu dituen proiektuan eta zein den gaurko data.

Arduradunak sistema informatikoa erabiliko du datak ondo betetzen ari direla ziurtatzeko. Arduradunak berak proiektuaren egituraketa sartuko du (ezaugarri nagusiak, pertsonala, arduradunak, eta abar) ere. Edozein unetan proiektuaren egoera azter dezake eta nahi dituen aldaketak egin.

Proiektuaren ezaugarri nagusiak honako hauek dira:

– Proiektua atazetan zatitzen da eta gaitasun desberdinak eskatzen ditu.
– Proiektu batek arduradun bakarra du.
– Ataza bat pertsona baten ardurapean egongo da.
– Langile batek proiektu batean baino gehiagotan egin dezake lan.
– Ataza bat gauzatzeko pertsona bat baino gehiago behar izaten da normalean.
– Proiektu bakoitzak hasiera- eta bukaera-data ditu, eta langile kopuru bat egokitzen zaio. Proiektuak kodearen arabera daude antolatuta eta aste jakinetan listatuak ateratzea interesatzen zaigu.
– Proiektu gehienak bezeroek eskatzen dituzte baina badaude proiektu batzuk barne-proiektutzat hartzen direnak.
– Langile bakoitzaren gaitasun maila guztiak gorde nahi ditugu.

Honako gertaera hauek eragin zuzena dute proiektuaren bilakaeran:

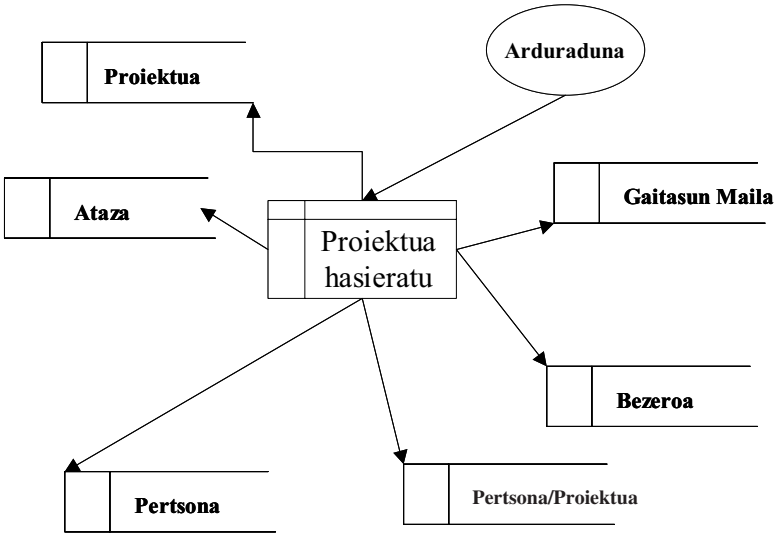
– Proiektu bat martxan jartzeko bezeroaren eskaera bat edo barne-eskaera bat jaso behar da. Lehenengo eta behin, arduradun bat aukeratzen da eta ondoren, pertsonala esleitzen zaio proiektuari. Proiektuaren arduradunak plan bat garatzen du eta sisteman sartzen du. Proiektua urratsetan antolatzen da eta hasierako eta bukaerako datak sisteman sartzen dira. Urrats bakoitza bukatzen denean kalitate-kontrola aplikatzen zaio.
– Proiektua bukatzen denean proiektu osoaren berrikuspena egiten da eta emaitzak gorde egiten dira sisteman. Urtea igarota berrikuspena artxibatu egiten da.

Honako hiru gauza hauek landuko ditugu adibide honetan:

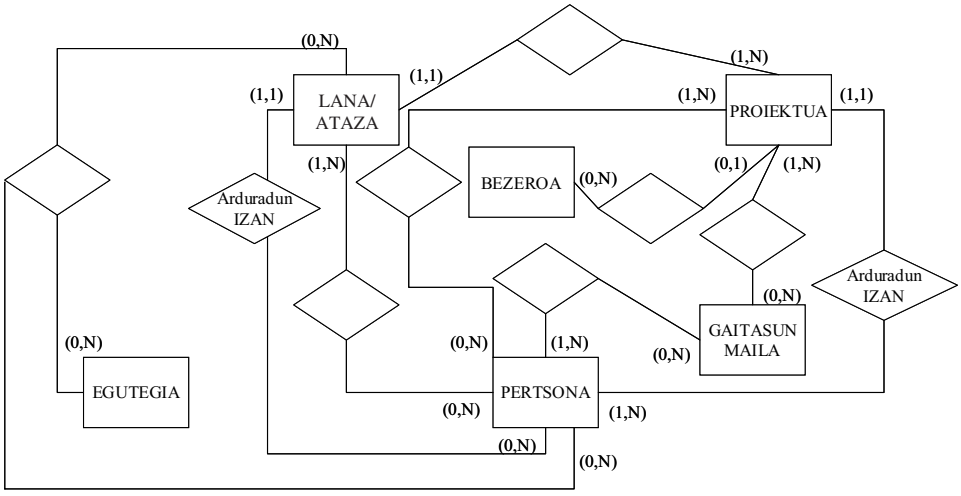
1. Sistema informatikoaren proiektua hasieratzeko DFDa (Datuen Fluxu Diagrama).
2. DELa (Datuen Egitura Logikoa).
3. Entitatearen Biziaren Historia (EBH).

Prozesu mailan, Datuen Fluxu Diagramak (**6.10. irudia**) ez du arazo handirik emango. Prozesu orokor bat antolatuko dugu DFDaren erdian kokatuko duguna eta horren inguruan zehaztuko ditugu aplikazio informatikoak erabiliko dituen fitxategi guztiak. Arduradunak, prozesuari dagokion interfaze grafikoa erabilita,

proiektuaren zehaztasunak adierazten dizkio sistemari, horrek fitxategi egokietan biltegira ditzan. Kontu egin fitxategi horietako informazio guztiak bat etorri behar duela DELan azaltzen denarekin, hala, gero, egiaztapenean ez da arazorik aurkituko bai bata eta bai bestea inplementaziorantz bideratzeko.

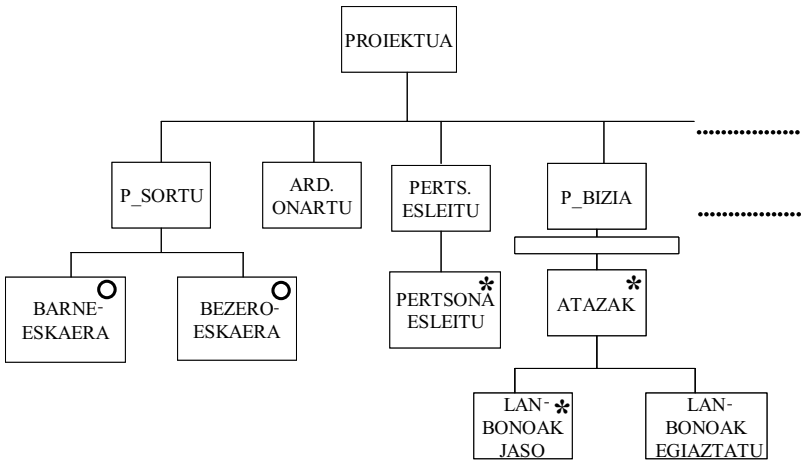


6.10. irudia. Adibidearen Datuen Fluxu Diagrama.



6.11. irudia. Adibidearen Datuen Egitura Logikoa.

Datuen Egitura Logiko guztiak (**6.11. irudia**) `Proiektua` entitatearen inguruan pibotatuko du. Proiektuak eta proiektuaren atazek bereganatuko dute arreta guztia bai proiektua bera egituratzeko baita proiektuari eta proiektuaren ataza guztiei esleitu behar zaien pertsonala antolatzeko ere. Kontu egin proiektuak arduradun bakarra izango duela baina ataza bakoitzak, bere aldetik, bere arduraduna ere izango duela. Jakina, proiektua aurrera eramateko ez da nahikoa arduradunak esleitzea, lana ere jarri beharko diegu, alajaina. Ohi den legez, arduradunen lana besteen lana kontrolatzea denez, pertsonal gehiago esleitu behar diogu proiektuari ataza bakoitza gauzatzeko. Bestalde, diseinuan agertzen den erlazio hirutarrean, langileak ataza bakoitzean sartzen dituen lan-orduak biltegiratuko dira.



6.12. irudia. Adibidearen Entitatearen Biziaren Historia.

Proiektuaren Entitatearen Biziaren Historian, `Proiektua` entitateak aplikazioan izango duen bizi-zikloa adierazten da. Grafikoki (**6.12. irudia**) adierazten den legez, proiektua barne-eskaera baten bidez edo bezeroaren eskaera batetik sortzen da. Sortuz gero, arduradun bat esleitzen zaio eta jarraian, proiektua gauzatuko duen pertsonala. Pertsonala esleituta, proiektua martxan jartzen da eta atazako haien lan-bonoak jasotzen dira eta amaieran, egiaztatu egiten dira guztiak. Proiektuaren atazak prozesatuz gero, azken eragiketak egiten dira txostenak aterata eta proiektua biltegiratuta.

6.3. OBJEKTUETARA ZUZENDUTAKO GARAPENA: UML

6.3.1. UMLren kondaira txikia

Laurogeita hamarreko hamarkadaren hasieran, softwarea garatzen zuen profesionalak aukera mordoa zuen analisi eta diseinu mailan metodoen merkatuan.

Estandarizazio ezak, lehenago metodo egituratuetan gertatu bezala, metodoen *gerla* piztu zuen egile-proposatzaileen artean. Askotan, lehen bezala, marketina izaten zen eragile nagusia metodo bat aukeratzerakoan, informatikaren kaltetan.

Egoera horretan, 1994an Grady Booch-ek, Rational Software Corporation enpresaburuak eta aldi berean Booch metodoaren proposatzaileak, James Rumbaugh OMT (Object Modeling Technique) metodoaren sortzaileetariko bat kontratatu zuen OMT eta Booch metodoak elkartuz metodo berri eta eguneratu batean lan egiteko. Booch eta Rumbaugh-ek Unified Method-aren 0.8 bertsioa kaleratu zuten 1995eko urrian eta urte-bukaeran, europar bat, Ivar Jacobson OOSE (Object-Oriented Software Engineering) metodoaren sortzailea, Rationalekin elkartu zen metodo bateratuan lan egiten jarraitzeko. Elkarlanaren ondorioz, Booch-ek, Rumbaugh-ek eta Jacobson-ek metodoaren 0.9 bertsioa atera zuten 1996ko ekainean.

Garai hartan, aipatutako gerla-giroa baretzeko eta egoera kaotiko hari —une batez 50 bat metodo metatu ziren bataila-eremuan— irtenbide bat emateko OMG (Object Management Group) erakundeak proposamen-dei bat zabaldu zuen modelatze-lengoaia baten ezaugarriak jaso eta estandarizazio-prozesu bat abian jartzeko. Une horretan, Rationalekoak konturatu ziren prest zeudela gero UML (Unified Modeling Language) deituko zen modelatze-lengoaia bateratuaren 1.0 bertsioa proposatzeko.

OMGren barnean pisutsuak ziren enpresen —IBM, Oracle, Hewlett-Packard,...— proposamenak jaso ondoren, 1997ko uztailean UMLren bertsio eguneratua azaldu zen eta urte hartako azaroaren 14an OMGk estandartzat aukeratu zuen.

Ondoren, 1998ko ekainean, 1.2 bertsioa argitaratu zen eta urte bereko udazkenean 1.3 bertsio eguneratua. Eta oraingoz, 2.0 bertsioaren zain, 2001eko irailean ateratako 1.4 bertsioarekin itxiko dugu behin-behingoz atea.

6.3.2. UML Diagramak

Diagrama bat eredu baten ikuspegi bat da sistemaren alde partikular bat aztertzen duena sistemaren adierazpen partzial bat eskainiz. Diagrama bat semantikoki trinkoa da beste ikuspegiekin.

UMLn hamar diagrama estandar daude:

Ikuspegi estatikoa: Klase Diagrama, Objektu Diagrama, Osagai (component) Diagrama, Ezarpen (deployment) Diagrama eta Pakete (package) Diagrama.

Ikuspegi dinamikoa: Erabilera Kasu (Use case) Diagrama, Sekuentzia Diagrama, Kolaborazio Diagrama, Egoera-Makina (state machine) Diagrama eta Jarduera (activity) Diagrama.

6.3.3. Erabilera Kasu (Use case) Diagramak

Erabilera-kasuak euskarri ezinbestekoak dira sistemaren arkitektura baieztatu eta garapena ondo gauzatzen ari dela egiaztatzeko. Erabilera-kasuetan sistemak zer egiten duen azpimarratzen da, sistemaren kanpo-ikuspegia azalduta. Ondoren, eta beste diagrama batzuk erabilita, batez ere Kolaborazio Diagramak eta Sekuentzia Diagramak, sistemak nola funtzionatu behar duen adierazten dugu. Erabilera Kasu Diagrama bat oso egokia da baita ere sistemaren proba-kasuak prestatzeko.

Erabilera-kasuak sistemaren funtzionalitatea atzematen du erabiltzailearen ikuspegia hartuta. Garapenaren hasierako faseetan eraikitzen da eta aztertu nahi den sistemaren portaera-patroi bat da. Aplikazioaren lehen urratsetan lantzen denez, analistak eta domeinuan aditua denak garatzen dute, horrela, haien arteko komunikazioa erraztu egiten da eta eraginkortasuna handitu. Eskuratu edota gordetzen den informazio hori ortogonala izango da eta semantikoki trinkoa, hau da, islatua eta jasoa egon beharko du beste hainbat diagramatan, hala nola, Klase, Sekuentzia eta Kolaborazio Diagrametan hurrenez hurren.

Erabilera-kasu bat honako helburu hauekin eraikitzen da: sistemaren ingurunea zehazteko, sistemaren betekizunak atzemateko, sistemaren arkitektura egiaztatzeko eta, inplementazioa bideratu eta test-kasuak sortzeko.

Erabilera-kasu bat ondo deskribatzeko bi arlo nagusi landu behar dira: a) haren interfaze grafikoak diseinatu eta marraztu egin behar dira eta, b) erabilera-kasu bat egikaritzen denean zein izango den gertaera-fluxua azaldu behar da.

- Interfaze grafikoa diseinatu behar da erabiltzaileak jakiteko zer ikusi eta nolako elkarrizketa gauzatu behar duen sistemarekin. Interfaze grafikoak adieraziko du zein informaziorekin lan egingo duen sistema informatikoak.
- Erabilera-kasu bakoitzeko, gertaeren agiri bat prestatu behar da eta honako hau jaso behar du:
  - Nola hasi eta bukatzen den erabilera-kasua.
  - Erabilera-kasuaren gertaeren fluxu normala.
  - Erabilera-kasuaren gertaeren fluxu ezohikoa.

Goiko bi arlo horiek deskribatuta, Erabilera Kasu Diagrama marraztu behar du analistak, aktoreak eta erlazioak erabilita. Azal dezagun, beraz, zer den bat eta zer diren besteak:

Aktorea norbait edo zerbait da garatu behar den sistemarekin elkarrekintzan jartzen dena. Erabiltzaileak sisteman betetzen duen *rola* adieraziko du. Aktoreak asoziazio-erlazioa erabiliko du erabilera-kasu batekin konektatzeko. Asoziazioak aktorearen eta erabilera-kasuaren arteko mezuen joan-etorria adierazteko balioko

du. Erabilera-kasuaren eta aktorearen artean mezuak bidali eta jaso egingo dira eta asoziazioa izango da euskarria.

Asoziazio-erlazioak oinarrizko komunikazioa adierazteko balio du baina Erabilera Kasu Diagrama batean beste hiru erlazio mota ere ager daitezke: generalizazioa, hedapena (extend) eta barneratzea (include/use).

Klaseen arteko generalizazioa ulertzen duenak oso erraz uler dezake erabilera-kasuan erabiltzen den generalizazio-erlazioaren semantika, oso antzekoak baitira. Alegia, erabilera-kasu semeak erabilera-kasu aitaren portaera eta esanahia heredatzen ditu eta gainera, semeak, aitaren portaera birdefinitu eta osatu egin dezake eta aita ordezkatu, portaera zehatzago bat egikarituta aplikazioaren exekuzioan.

Hedapen-erlazioarekin oinarrizko erabilera-kasua osatu egiten dugu beste erabilera-kasu bat txertatuz oinarrizko erabilera-kasuan. Gertaera-fluxua ondo dokumentatzeko argi eta garbi utzi behar dugu non eta noiz txertatzen den erabilera-kasu berria eta zein den egikaritu behar den baldintza txertaketa gauzatzeko. Erabiltzaileak ulertu beharko luke hautazko portaera baten aurrean dagoela honelako erlazio bat atzematen duenean Erabilera Kasu Diagraman.

Barneratze-erlazioak balioko digu erabilera-kasu batek beste erabilera-kasu bat erabiltzen duela adierazteko. Oinarrizko erabilera-kasuak bere portaera propioa izango du baina betekizunak aztertzerakoan konturatu gara beste portaera gehigarri bat eskatzen duela eta gainera portaera hori beste erabilera-kasu batek gauzatzen duela. Informazio horrekin eta gertaera-fluxua alferrik ez errepikatzeko, barneratze-erlazioa erabiltzen dugu analisiaren eraginkortasuna hobetzeko.

Modelatze bisualaz ari garenez, adibide konplexu bat (**6.13. irudia**) landuko dugu orain arte azaldu duguna ondo finkatuta uzteko.

AloKamioi enpresa ertain bat da gidari gabeko kamioi txiki, ertain eta handiak alokatzen dituena. Enpresa honek 50 bulego ditu eta bulego bakoitzak bere garajea du kamioiak bertan aparkatzeko. Aplikazio informatiko bat sortu nahi dugu eta enpresaren bulego bakoitzean kokatu. Aplikazioak honako betebehar nagusi hauek izango ditu: bezeroen erreserbak kudeatu eta sukurtsal bakoitzeko bezeroen datu pertsonalak eta bakoitzaren kreditua kudeatu. Bezeroak zenbat kamioi aloka dezakeen adierazten du kredituak.

Aplikazioak, bestalde, sukurtsaleko kamioien informazioa ere kudeatu beharko du. Horretarako, kamioiak motetan banatzen dira. Mota bakoitzak tarifa batzuk izango ditu alokatzeko denbora-unitatearen arabera (eguna edo egun-zatia, astea eta hilabetea) eta egindako kilometroen arabera (frankizian: kilometro hauek ez dira ordaintzen, eta kilometro × salneurria: hurrengo kilometroen fakturazioa). Kamioi bakoitza aloka daitekeen ala ez jakiteko, haren egoera (libre, alokatuta edo konpontzen) kontrolatu egin beharko da.

Antolatzen ari garen aplikazioak bezeroen erreserbak kudeatu nahi ditu eta haien egoera kontrolatu. Erreserba bat baieztatu, onartu, baztertu eta ezeztatu egin daiteke. Onartua denean erreserbaren zati bat egiazta dezakegu baina ez osoa. Zati bat ere ezin dugunean erreserbatu, baztertu egiten dugu erreserba. Baztertua denean bezeroak krediturik ez duelako izango da eta aukera ematen diogu bere kreditua igotzeko. Bezeroak ezezta egin ditzake erreserba baieztatuak eta onartuak. Sukurtsal batek beste sukurtsal baten eskaerak prozesa ditzake baina kamioiak sukurtsal eskatzailera eraman beharko dira. Sukurtsalek bananduta egiten dute lan. Aplikazioa sukurtsal bakoitzean instalatuko dugu eta sukurtsalek ez dute atzipenik Internetera.

Bezeroen eskaerak zuzenean egin daitezke sukurtsalean edo telefonoz zehaztu. Gogoratu beste sukurtsalek ere kamioiak eska ditzaketela eta, guretzat, erreserbak izango direla.

Aplikazio informatikoa kamioien kudeaketaz ere arduratuko da. Horretarako, administrazioko pertsonalak kamioiak sartuko ditu datu-basean: gureak eta inguruko sukurtsalek *uzten* dizkigutenak zerbitzu bat egiteko.

Erreserba honela gauzatuko da: lehendabizi, bezeroa identifikatu egingo da, eta ondoren, bere kamioi-premiak azalduko ditu data bi zehaztuta (erreserbaren hasiera- eta bukaera-datak). Eskaera honetan, bezeroak kamioiak eskatuko ditu mota batekoak edo gehiagokoak. Adibidez: A motako 3 kamioi txiki eta D motako 7 kamioi, otsailaren 5etik 10era Donostiako sukurtsalean hartu eta Malagan uzteko. Bezeroa ez badago erregistratua, haren datu pertsonalak eskatzen dira eta erregistratu egiten dugu.

Administrazioko pertsonalak itxaron-zerrendaren ardura bere gain hartuko du. Lan hau gauzatzeko, zintzilik dauden erreserbak tratatuko ditu bakoitzaren lehentasunaren arabera: lehendabizi, martxan dauden erreserbak, hau da, erreserba-garaia hasi eta arazoak (matxurak edota istripuak) izan dituztenak eta ondoren, hurbilen daudenak denboran zehar. Prozesua honela gauzatuko da: hasieran behar den kamioi mota hori sukurtsalean dagoen ala ez ikusten da. Badago (posible da arazoak izan dituen kamioiren bat konpondu ahal izatea) erreserba esleitzen da. Ez badago, ezaugarri horiek dituen kamioi bat beste sukurtsal batzuetan bilatzen saiatuko da sukurtsaleko pertsonala. Horretarako, telefono-deiak egingo ditu. Mota horretako kamioi bat libre aurkitzen badu, egiaztatu egiten da gure datu-basean aurretik sartuta dagoen ala ez. Ez badago, alta ematen zaio eta esleitu egiten da erreserbara.

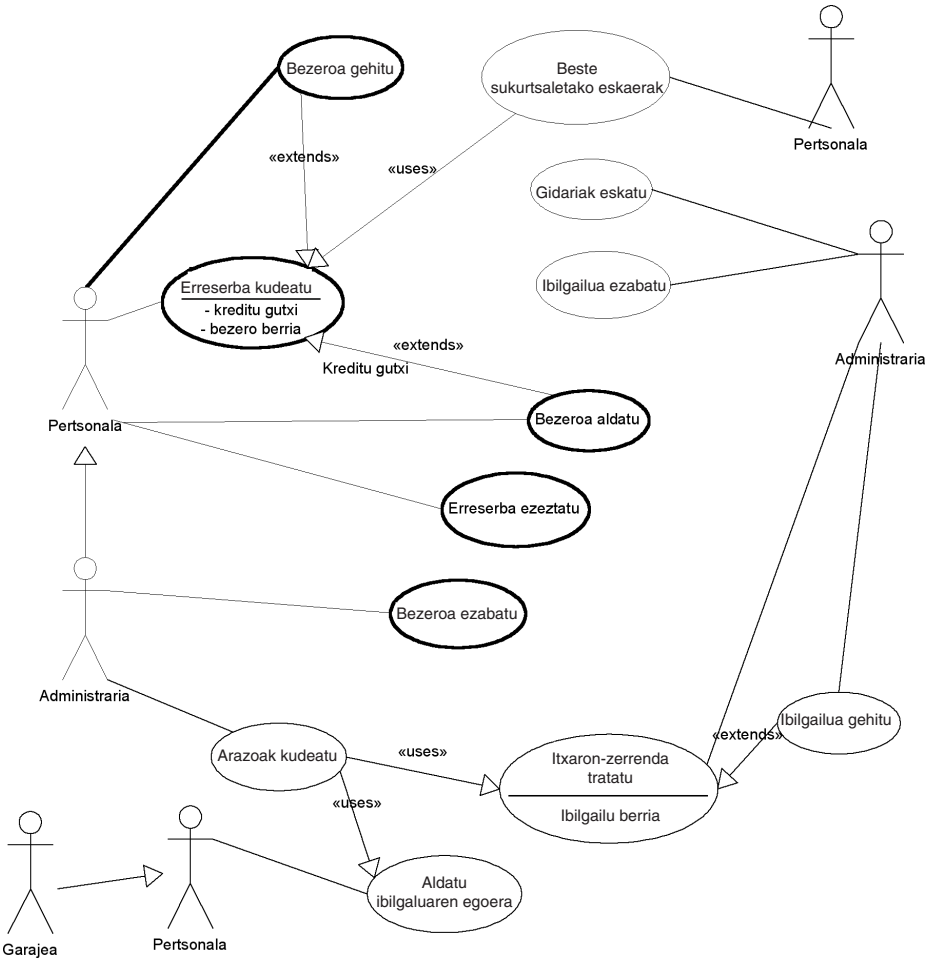
Erreserba bat ezeztatu egiten denean, kamioien esleipena desegiten da eta bezeroari ez zaio zigorrik jartzen ezeztapena data egokietan gauzatu bada, alegia, erreserbaren hasiera-data baino hiru egun lehenago ezeztatu badu, erreserbaren % 10arekin zigortzen da eta ezeztapena, erreserba-egunean edo erreserba hasita gauzatzen badu, % 100arekin zigortzen zaio.

Erreserba hasi eta gero, bezeroak arazoak eduki ditzake kamioiarekin. Egoera horretan, bezeroak telefonoz azalduko du arazoa (matxura edota istripua) eta sukurtsaleko pertsonala arazoa konpontzen saiatuko da. Horretarako kamioiaren egoera aldatuko du eta erreserba itxaron-zerrendan jarriko du eta lehenbailehen arazoa konpontzen saiatuko da itxaron-zerrenda tratatuta.

Eguneko lehen lanorduan garajeko pertsonalak telefonoz adierazten du zein kamioi konpondu diren garajean.

Erreserba-prozesua bukatzeko, administrazioko pertsonalak beste sukurtsaletatik egiten zaizkien eskaerak tratatuko ditu. Eskaera horiek erreserba gisa tratatuko dira sukurtsalean.

Egunero, 17:00etan, gidarien eskaera-orria inprimatzen da sukurtsal bakoitzean. Orri hori, faxez bidaltzen da gidariak kontratatzen dituen enpresa batera. Orri horretan, gidari-beharrizanak agertzen dira gidabaimen motaren arabera sailkatuta. Kamioien bidalketa gauez egiten da hurrengo goizean prest egon daitezen dagozkien lekuetan. Ibilgailu bat, honako kasu hauetan bidaltzen dugu: (1) bezeroak arazo bat izan du eta inguruan sukurtsal bat dago; (2) bezeroak, beste sukurtsal batean jaso nahi du kamioia; (3) bezeroak, kamioi bat entregatu du eta beste sukurtsal batekoa da, eta (4) beste sukurtsal batek ibilgailua eskatu du datorren egunerako.



6.13. irudia. Adibidearen Erabilera Kasu Diagrama.

6.3.4. Klase Diagramak

Klase Diagrama batek sistemaren ikuspegi estatikoa islatzen du klaseak eta klaseen arteko harremanak erabilita. Klaseak, garatu nahi den sistemaren datuen hiztegitik egituratzen dira.

Klase Diagrama batek honako osagai hauek ditu:

- Klaseak (egitura eta portaera)
- Asoziazio-, agregazio- eta herentzia-erlazioak
- Kardinalitatea eta nabigatze-adierazleak
- Rolen izenak

Klase bat objektu-bilduma bat da. Bilduma horretako objektuek egitura, portaera, erlazio eta semantika berdinak izango dituzte. Klasea, laukizuzen bat erabilita marrazten da eta hiru zati izango ditu: izena, atributuak eta eragiketak (metodoak).

Datuen hiztegiak jasotzen du zer den garrantzitsua erabiltzailearentzat eta analistarentzat, alegia, betekizunak zehazteko orduan bai bezeroak, etorkizuneko sistemaren erabiltzaile gisa, eta bai analistak, sistema garatu behar duen heinean, ados jarri behar dute eta *lexiko* bera erabili behar dute domeinuaren kontzeptuak adierazteko orduan, eta garrantzitsuagoa dena, semantika berarekin. Adibidez, etorkizuneko sistema mekanizatuaren inplementazioak arazoak sortuko dizkigu erabiltzaileak eskaerak ordaintzeko garaian bezeroak ordainagiria epeka ordain dezakeela esan badio analistari eta horrek bere betekizunetan ordainketa-data zehatz batean egingo dela jaso badu.

Klasearen egitura bere atributuek adieraziko dute. Modelatzen ari garen elementuaren propietate estatikoak atributuetan egongo dira eta klase-definizioetan, betekizunetan eta domeinuaren azterketa burututa aurkitzen dira. Analistak honako formatu hau erabilita adieraziko ditu atributuaren ezaugarri guztiak:

[ikusgaitasuna] izena [kardinalitatea] [: mota] [= hasiera_balioa] [{propietateak}]
ikusgaitasuna[56] **= +, # , -**

Adibidea: **– Kreditua : float = 3000**

Klase baten portaera bere eragiketek adieraziko dute. Eragiketa bat da klase baten objektuei eska diezaiekegun zerbitzu baten inplementazioa. Eragiketak elkarrekintza-diagramak aztertuta aurkitzen dira. Analistak honako UML signatura hau erabilita adieraziko ditu eragiketaren ezaugarriak:

[ikusgaitasuna] izena [(parametroak)] [: itzul_mota] [{propietateak}]

Adibidea: **+ lortuKreditua( ) : float**
+ hasieratuKreditua(balioa : float) : void

Analista klaseak aurkitzen hasten denean, erabiltzaileak pasa dizkion betekizunen artean, berehala konturatzen da klaseak ez direla isolatuta agertzen. Batzuetan, klase batek beste bat erabiltzen du bere funtzionalitatea edo bere semantika propietate mailan osatzeko. Beste batzuetan, ostera, klase batek beste bat birfintzen du lehenengoaren portaera berriz erabilita.

Lehenengo egoera mendekotasun-erlazioa erabilita tratatuko da. Erlazio mota hori, erabilera-erlazioa da, alegia, klase batek beste klase bat erabiliko du eta

56. + : Programazio-lengoaietako `public` ezaugarria adierazten du.
: Programazio-lengoaietako `protected` ezaugarria adierazten du.
– : Programazio-lengoaietako `private` ezaugarria adierazten du.

edozein diseinu-aldaketak bigarren klase horretan eragina, eta askotan eragin ez desiragarria, izango du klase erabiltzailean. Erlazio mota hori adierazteko lerro eten bat erabiltzen da gezi batekin ertz batean. Geziaren aldeak adierazten du nondik eratortzen den mendekotasuna.

Aitzinean aipatutako bigarren egoerak klase/azpiklase erlazioa islatzeko balioko du. Erlazioa adierazteko triangelu txiki bat kokatuko da klase biak erlazionatzeko erabiliko den lerroaren gainean. Triangeluaren oinarrian kokatzen den klaseak goian ipintzen denaren portaera heredatuko du diseinuan. Jakina, portaera orokor horri berarena propio dena gehituko zaio.

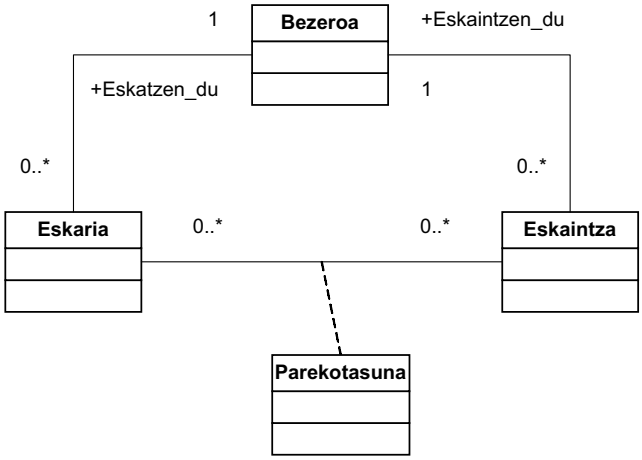
Hirugarren erlazio mota, asoziazio-erlazioa, klaseen instantzien arteko erlazio egituratzailea da eta objektuen artean *nabigatzeko* balioko du. Erlazio horren bidez, klaseen arteko lotura potentziala adieraz dezakegu analisi eta diseinu mailan. Exekuzio-garaian benetakoak bihurtuko dira lotura hauek klaseen instantziak erabilita.

Asoziazioak bi klase soilik konektatzen dituenean bitarra dela esango dugu. Nahiz eta oso ohikoa ez izan, batzuetan beharrezkoa izaten da bi klase baino gehiago konektatzea Klase Diagraman asoziazio n-arrak sortuz.

Asoziazio bat zehatz-mehatz deskribatu nahi badugu, honako ezaugarri hauek izan beharko ditu: izena, rola eta kardinalitatea. Izenak erlazioaren natura deskribatu behar izango luke eta gezi bat itsatsi diezaiokegu erlazioaren irakurketa errazteko.

Rolak erlazioaren klase bakoitzaren bisaia zein den erlazioan adieraziko du. Eta bukatzeko, kardinalitateak adierazten du erlazioaren instantzien alde batean eta bestean zenbat objektu konekta daitezkeen. Ohiko kardinalitateak honako hauek dira: zero edo bat (`0..1`), zehatz-mehatz bat (`1`), asko (`0..*`) eta, bat edo asko (`1..*`). Izartxoaren balio zehatza ezagutzen denean balioa jartzen da (`1..4`, `0..7`) izartxoaren ordez.

Klaseen arteko asoziazio-erlazio batek, berak ere, propietate berriak sor ditzake. Adibidez, etxeen salmentan aritzen den agentzia batek bere kudeaketa mekanizatu nahi badu seguru asko ondoko edo antzeko Klase Diagrama bat erabili beharko du:



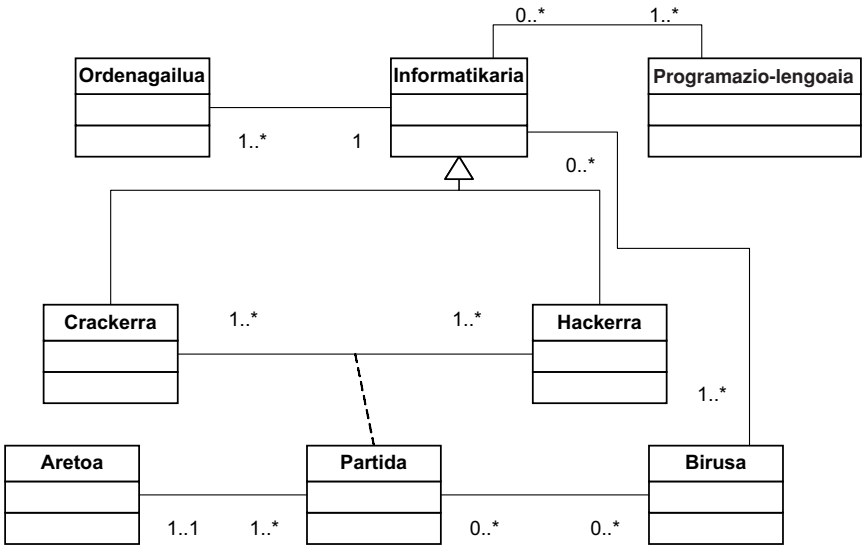
6.14. irudia. Domeinuaren Klase Diagrama.

Diagraman (**6.14. irudia**) ikusten den bezala agentziaren bezeroak etxe-saltzaileak eta etxe-erosleak izango dira. Erosleak eskariak egingo ditu adieraziz erosi nahi duen etxeak nolako ezaugarriak bete behar dituen; aitzitik, saltzaileak etxeak eskainiko dizkio agentziari, horrek sal diezazkion eta haien ezaugarri guztiak agentziaren datu-basean gordeko dira. Bezeroaren eskaria salmentan eskaintzen den etxe batekin parekatzen dugunean klase berri bat marraztu beharko dugu gure klase-diagraman gertaera hori jasotzeko. Klase hori, irudian ikusten den legez, ez da klase arrunt bat, asoziazio batetik eratortzen dena baizik, horratx, asoziazio-klase izenaren jatorria. Beraz, esan daiteke klase berri hori asoziazio bat dela klaseen propietateak dituena edota klase bat aldi berean asoziazioaren propietateak adierazten dituena. Klase bat asoziazio-klasea dela adierazteko, lerro eten bat marraztuko da aipatutako klasetik sorburu izan den asoziaziora.

Asoziazio-erlazioan parte hartzen duten klaseak baliokideak dira kontzeptualki, alegia, maila berean daude baina batzuetan gertatzen da Klase Diagramak asoziazio-erlazioarekin soilik ezin duela asoziazio mota (agregazioa) berezi bat deskribatu. Asoziazio mota horretan *guzti/zatia* erlazioa modelatzen da eta horretarako hedatu egin behar dugu asoziazio-erlazio arrunta. Agregazio horren bidez eta erlazio bat erabilita, klase baten (guztia) osagai kontzeptualak (zatiak) modelatu nahi ditugu, adierazteko klase orokorra *zerez* dagoen osatua. Klase nagusi hori erlazioan identifikatzeko erronbo bat marraztuko dugu aipatutako klasearen aldean, adierazteko nork nor subsumitzen duen erlazioan. Erronboa beltzez marrazten denean agregazioaren semantika zertxobait aldatzen da, alegia, klase orokorraren (guztia) zatiak klase orokorra sortu ondoren eraiki daitezke, ondoko edozein unetan, baina sortuz gero, klase orokorraren patuari jarraituko diote beren bizitza eta heriotza erabat lotuak egongo baitira klase orokorrari. UMLn, agregazio mota hori izendatzeko, konposizio hitza erabiliko da.

Orain arte ikusitakoak Klase Diagramaren erlazio egituratuak modelatzeko balio izan digu. Batez ere, betekizunen eta analisi mailan mugitu izan gara eta ordua heldu zaigu bizi-zikloan behera jotzeko eta diseinuan sartzeko. Hemen ohi denez, gure sistema berriaren goi-azterketa zehaztu beharko dugu eta klase bakoitzaren alde estatikoa (klasearen egitura: atributuak) eta dinamikoa (klasearen portaera: eragiketak) zehatz-mehatz adierazi beharko ditugu, programatzailea, sistemaren diseinua hartuta, gauza izan dadin aplikazioa programatzen hasteko.

Alde estatikoan, alegia, klasearen atributuetan argi eta garbi utzi beharko da atributu bakoitzaren ikusgaitasuna zein izango den. Diseinatzaileak, une honetan, erabaki beharko du klasearen zein informazio ezkutatu eta ondo ezkutatu nahi duen, eta zein erakutsi erraztasunak emanez inguruko klaseei bertara atzitzeko.



6.15. irudia. Adibidearen Klase Diagrama.

Alde dinamikoan, bestalde, klasearen eragiketetan aitzinean atributuetan esan dugun legez, ikusgaitasuna itsatsi beharko diegu eragiketa guztiei. Klasearen portaera adierazterakoan, UMLk bereizketa egiten du eragiketa eta metodo hitzen artean. Eragiketa hitzaren esanahia orokorragoa da eta klase batek eskaintzen duen zerbitzuekin lotzen da; metodo hitzarekin, ostera, eragiketaren inplementazioaz ari gara. Klase baten eragiketa ez abstraktu batek beti izan behar du metodo bat esleitua nahitaez.

Klase Diagramaren azalpena amaitzeko adibide txiki bat landuko dugu diagramaren arloak azaleratzeko. Demagun Eusko Jaurlaritzaren Informatika Sailak txapelketa bat antolatu nahi duela crackerren eta hackerren artean informatikaren erabilera eskoletan bultzatzeko. Jakina denez, nekez ikusiko du norbaitek, hackerren artean cracker bat programatzen, etsai amorratuak baitira. Hori dela eta, hackerrak

crackerren aurka arituko dira txapelketan zehar. Hackerrak crackerren ordenagailuak kutsatzen saiatuko dira eta crackerrak hackerrenak. Sistema informatiko bat antolatu nahi dugu partiden zehaztasunak gordetzeko.

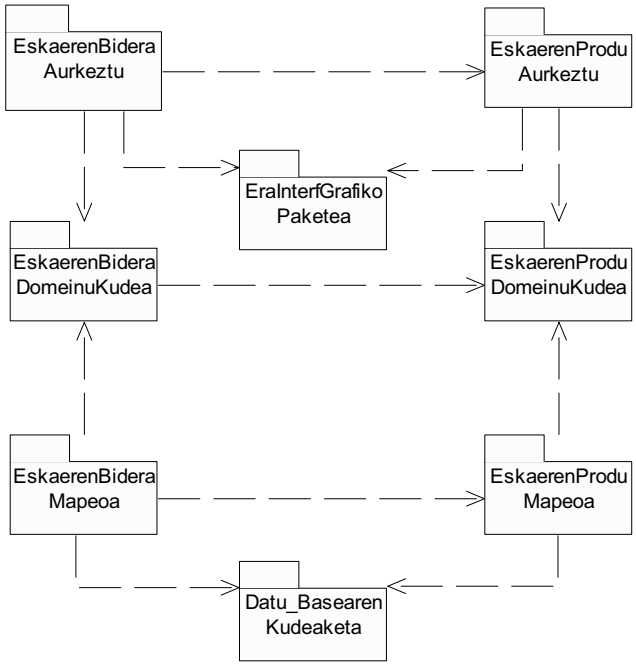
Aplikazioaren betekizunak aztertuz gero, **6.15. irudia**n agertzen den Klase Diagrama eraiki dugu. Diagramak partida bakoitzaren informazioa klaseetan egituratzen du informazio mota bakoitzaren alde estatikoa eta portaera adierazita. Partidak areto ezagunetan jokatzen dira eta ez dira errepikatzen. Jokalari bakoitzak etsaiaren ordenagailua kutsatzen du eta bakoitzak ahalegindu egin behar du aurkitzen eta ezabatzen besteak sartu dizkion birusak.

6.3.5. Objektu Diagrama

Diagrama mota honek, exekuzio-garaian eta une jakin batean, sistema informatikoaren objektuak nola dauden antolatuta islatzen du. Sistema informatikoaren Klase Diagramaren klaseen instantziak islatzen direnez, diagrama mota honek Instantzia Diagrama izena ere hartzen du batzuetan.

6.3.6. Pakete (package) Diagrama

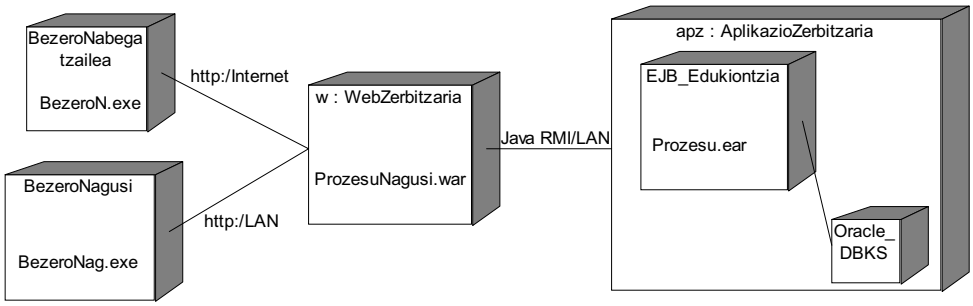
Diagrama honekin, paketeak eta paketeen arteko mendekotasunak jasotzen dira. Sistema lar handia ez denean, gehienetan, nahikoa izaten da Klase Diagrama marraztea klaseen arteko erlazioak egituratzeko. Egituraketa birfinduago bat lortu nahi badugu sistemak ehunka klase dituenean edota, nahiz eta Klase Diagrama oso konplexua ez izan baina bertako klaseen egituraketa birfinduago bat lortu nahi dugunean, interesgarria izaten da klaseak paketeetan biltzea Klase Diagramaren irakurgarritasuna errazteko. Demagun sistema informatiko konplexu bat eraiki nahi dugula eskaeren bideragarritasuna eta eskaeren produkzioa aztertu eta kudeatuko dituena. Azpisistema biak paraleloki garatuko dira eta arkitektura bera izango dute eta klase askoz osatuta egongo direnez, klaseak paketeetan antolatuko ditugu hiru mailetako arkitekturaren ezaugarriak kontuan izanda. Analisia gauzatu eta gero, erabaki dugu sistema zortzi paketetan egituratuko dugula. **6.16. irudia**k islatzen du sistemaren Pakete Diagrama nola antolatu dugun. Azpisistema bakoitzarentzat aurkezpen maila bat prestatu dugu erabiltzailea sistemarekin harremanetan jartzeko. Erabiltzailearengandik eta erabiltzailearengana datorren eta doan informazioa tratatzeko, domeinuaren kudeaketa-arauak prozesatuko dituzten paketeak kokatzen ditugu diagramaren erdialdean. Aplikazioaren semantika nagusia pakete horretan egongo da zalantzarik gabe. Bukatzeko, informazioaren mapeoa nonbait gauzatu beharko da, hau da, exekuzio-garaian memoria nagusian eraikiko diren objektuen atributuen balioak nonbaitetik ekarri edota norabait bidali beharko dira. Gehienetan datu-base erlazionalak erabiliko dira iraunkortasun hori ziurtatzeko. Gure sisteman hori guztia gauzatzeko, diagramaren beheko aldeko hiru paketeak erabiliko ditugu.



6.16. irudia. Aplikazioaren Pakete Diagrama.

6.3.7. Ezarpen (deployment) Diagrama

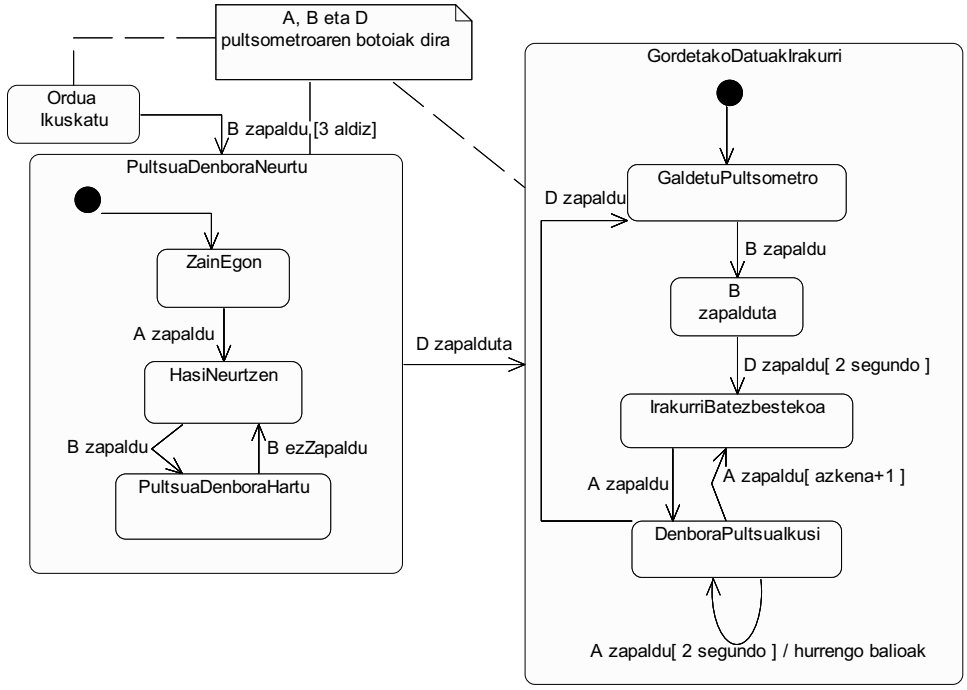
Diagrama mota honetan azalduko da nola ezarriko den aplikazio informatikoa hardware mota bakoitzean. Diagramak (**6.17. irudia**) nodoak eta konexioak nola antolatuko diren adierazten du eta nodo bakoitzean aplikazioaren zein osagai egikarituko den.



6.17. irudia. Aplikazioaren Ezarpen Diagrama.

6.3.8. Osagai (component) Diagrama

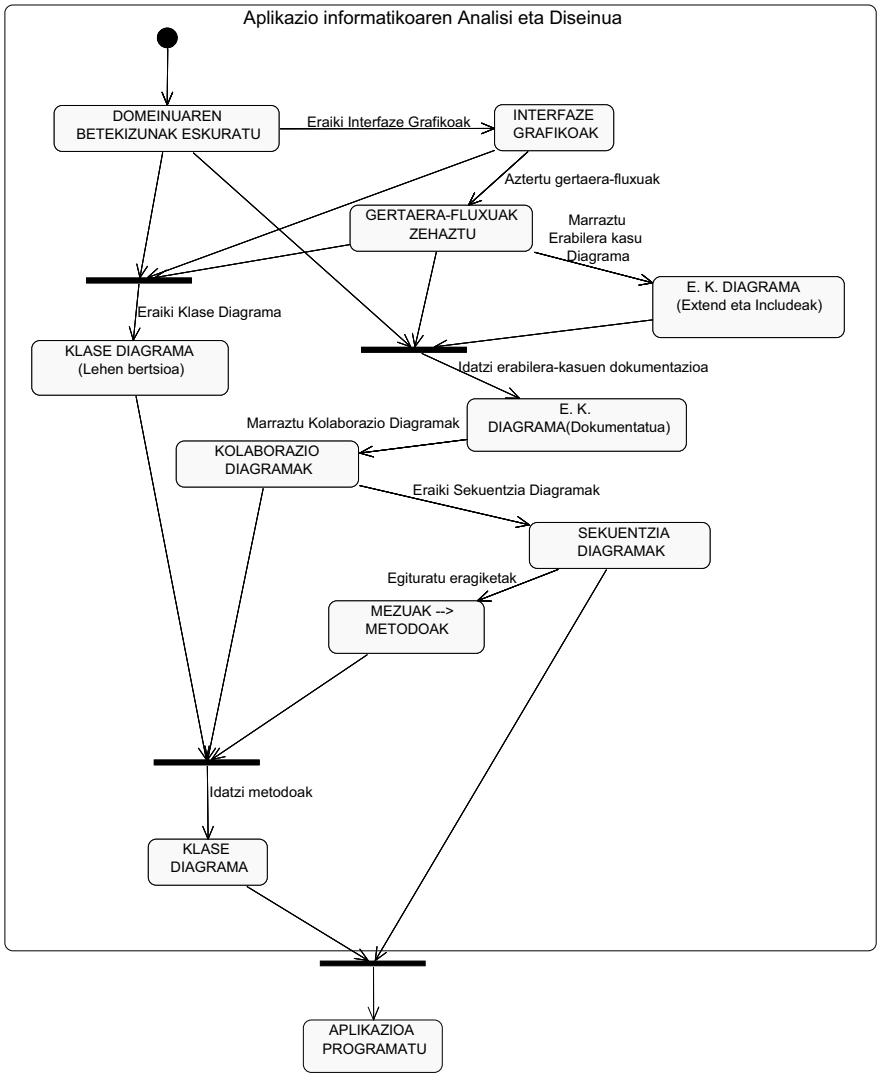
Nodo batean dauden elementu fisikoak modelatzeko erabiltzen dira osagaiak. Beraz, nodo horretan kokatu dugun elementu fisikoa objektu bat bada, osagaiak elementu fisiko hori paketatu egingo du eta osagai batekin parekatuko da baina, konturatu, halaber, nodoan dagoen elementu fisikoak ez duela nahitaez klase soil bat izan behar beti. Osagai onek abstrakzio zehatzak definituko dituzte eta interfaze argiak izango dituzte, sistema osoa ondo modularizatuta edukitzeko. Askotan sistema osagaietan zatitzea erabaki teknikoa izaten da baina, ez gutxitan, merkatuak eta marketinak inposatutako baldintza bat ere.



6.18. irudia. Pultsometroaren Egoera-Makina Diagrama.

6.3.9. Egoera-Makina (state machine) Diagrama

Testuinguru informatikoan, Egoera-Makina batek elementu informatiko baten bizitza modelatzen du bere alde dinamikoa azpimarratuz. Elementu informatikoa klasearen instantzia izan daiteke behe-mailakoa denean baina halaber, erabilerakasu bat edota sistema informatiko osoa. Egoera-Makinak (**6.18. irudia**) hasierako eta bukaerako egoerak izango ditu adierazteko, non hasi eta bukatzen den gertaerafluxuen exekuzioa. Egoeren arteko erlazioak trantsizioen bidez adieraziko dira diagraman. Trantsizioak bost ezaugarri izango ditu:



6.19. irudia. Aplikazio informatiko baten analisi eta diseinuaren Jarduera Diagrama.

- Iturburu-egoera. Egoera honekin, trantsizioa nondik abiatuko den adierazten da.
- Helmuga-egoera. Trantsizioa bukatu ondoren, diagramaren zein egoeratan gelditukо den Egoera-Makinaren exekuzioa adierazten da egoera honekin.
- Gertaera aktibatzailea. Ezaugarri hau iturburu-egoeran dagoen objektuak jaso behar duen gertaera izango da. Beti ez da beharrezkoa, egoera batzuetan, trantsizioa gauzatzeko nahikoa izango da iturburu-egoeraren exekuzioa bukatu dela jakitea.

- Baldintza-jagolea. Baldintza boolear hau, gertaera aktibatzailearen alboan jartzen da eta gertaera aktibatzailea gertatu ondoren ebaluatzen da. Beraz, iturburu-egoera batetik hainbat trantsizio izan ditzakegu gertaera aktibatzaile berarekin baina baldintza-jagole desberdinekin.
- Ekintza. Trantsizioa gauzatzen denean egikarituko den sistemaren portaera bat adierazten da ezaugarri honekin. Kontzeptualki, konputazio atomiko bat da, hau da, ekintzaren exekuzioa hasiz gero, beste gertaera batek ezin du moztu eta itxaron egin behar du ekintza bukatu arte.

6.3.10. Jarduera (activity) Diagrama

Jarduera Diagrama (**6.19. irudia**) fluxu-diagrama bat da jardueren artean eraikitzen den kontrol-fluxua kudeatzen duena. Diagrama honek antz handia du Egoera Diagramarekin, biek sistemaren portaera dinamikoa modelatzen baitute. Egoera Diagramak egoeren arteko kontrol-fluxua aztertzen du azpimarratuz trantsizioen semantika; diagrama hauetan trantsizioak konplexuak izaten dira eta egoerak ez hainbeste. Jarduera Diagrametan, bestalde, jarduerak konplexu xamarrak izaten dira gehienetan eta trantsizioak, ostera, nahiko oinarrizkoak. Analistak Jarduera Diagramak erabilita deskribatzen ditu sistemaren logika prozedurala, domeinuaren kudeaketa-arauak eta lan-fluxuak. Diagrama horiekin, analistak, objektuen artean pasatzen diren eragiketak islatzen ditu. Diagrama horiek Elkarrekintza Diagramen osagarriak dira, azken horiek objektuak adierazten baitituzte eta objektuen artean pasatzen diren mezuak, hau da, egikaritzeko eskatzen diren eragiketak. **6.19. irudia**k Jarduera Diagrama bat islatzen du UMLren oinarrizko software-prozesu bat adierazteko.

6.3.11. Elkarrekintza Diagramak

Software-produktu bat eraiki nahi dugunean, aplikazioaren betekizunak aztertzen hasten gara lehendabizi. Betekizun horiek eskuratzeko, bezeroarekin jartzen gara harremanetan eta UML erabiltzea erabaki badugu, datuen hiztegia —glosategia bezala ere ezagutzen dena— sortuko dugu lehendabizi, domeinuko kontzeptuak argi geldi daitezen analista eta bezeroaren artean. Hori egin eta gero, analisiarekin hasten gara eta Erabilera Kasu Diagramak eta Klase Diagramak erabilita analisia gauzatzen dugu, batez ere, sistemak (Erabilera Kasu Diagramak) *zer* egingo duen argituz baina *nola* egingo duen azaldu gabe oraindik.

Ondoren, Erabilera Kasu Diagrama eta datuen hiztegia erabilita Klase Diagramaren lehen hurbilpena egiten da. Logikoa den bezala, lehen hurbilpen horretan, ez da Klase Diagrama erabat landua eraikitzen, gehienez klase bakoitzaren atributu nagusiak identifikatzen dira eta klaseen arteko erlazio egituratuak eta estatikoak marrazten dira. Analisi maila horretan eta dugun informazioarekin, ezinezkoa egiten zaigu klase bakoitzaren portaera aurreikustea. Beraz, ebazpenaren

abstrakzio maila jaitsi egin behar dugu eta diseinuan sartu klase bakoitzaren portaera adierazteko.

Eraiki nahi dugun sistema informatikoan nahiko arraroa izango da klaseak erabat isolatuta egotea, normalean klase batek beste baten zerbitzuak beharko ditu bere eginbeharrak betetzeko eta berak ere zerbitzuak eskain ditzake beste klase batzuek erabil ditzaten. Zerbitzu horien inplementazioa mezuen bidez gauzatzen da. Objektu batek mezu bat bidaltzen dio beste bati horrek eskatutako eragiketa egikari dezan. Aldi berean eta batzuetan, mezua jaso duen objektuak beste objektu baten laguntza behar du eragiketa gauzatzeko eta hark ere beste mezu bat sortzen du eta hirugarren objektu bati bidaltzen dio. Horrela, hasierako mezuaren eragina Klase Diagramaren objektuetara transmititzen da eta hori guztia ondo modelatzea garrantzizko lana bihurtzen da aplikazioaren diseinatzailearentzat.

Lanak errazteko, UMLk Elkarrekintza Diagramak eskaintzen dizkio diseinatzaileari. Elkarrekintza Diagrama batek, oro har, erabilera-kasu bat nola egikaritzen den azalduko du, dinamikoki adieraziz objektuen arteko harremanak zein izango diren eta nolakoak.

UMLn Elkarrekintza Diagrama mota bi daude: Kolaborazio Diagramak eta Sekuentzia Diagramak. Diagrama horiek isomorfoak dira, alegia, semantikoki baliokideak, batetik bestera pasa gaitezke informazioa bidean galdu gabe.

6.3.11.1. Kolaborazio Diagrama

Kolaborazio Diagramek objektuen arteko elkarrekintzak azaltzeko balio dute eta objektuak nola antolatzen diren adierazten dute. Diagrama horietan, objektuen antolaketa egituratua azpimarratzen da, batez ere, haien portaera dinamikoa aztertuta.

Notazio mailan bi formatu erabiltzen dira Kolaborazio Diagrametan baliokideak direnak. Lehen formatuan, objektuak laukizuzenak erabiliz marrazten dira eta gertaeren kontrol-fluxua zenbakiak erabiliz adierazten da. Zenbaki horiek, mezu bakoitzaren alboan joango dira adieraziz nola garatzen den denboran zehar objektuen arteko elkarrekintza.

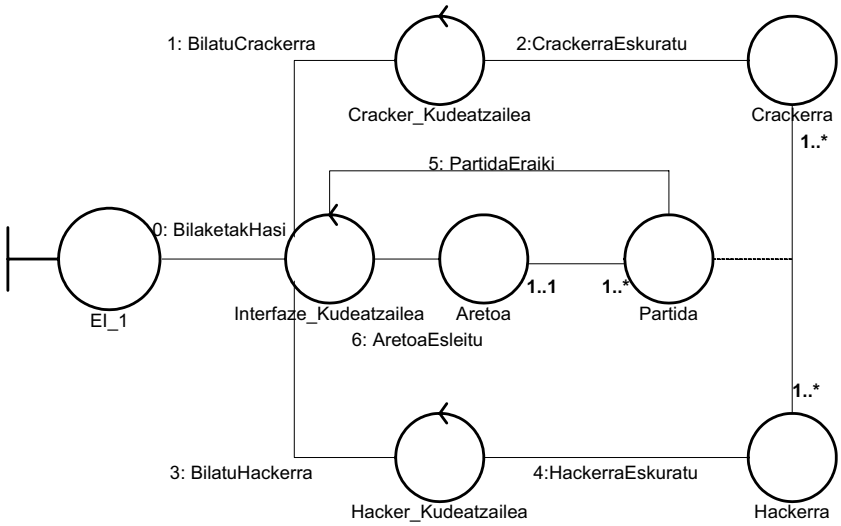
Badago beste notazio mota bat aberatsagoa dena eta Jacobson-ek proposatu zuena. Notazio horretan, hiru zirkunferentzia mota (**6.20. irudia**) erabilita adierazten dira objektuak. Zirkunferentzia mota bakoitzak, hau da, objektu mota bakoitzak, betebehar zehatza du sistemaren inplementazioan. Hiru objektu mota horiek honako izen hauek dituzte: *interfaze-objektua*, *entitate-objektua* eta *kontrol-objektua*.

Interfaze-objektuak, sistemaren eta aktoreen arteko elkarrekintzak modelatzeko erabiltzen dira. Informazioa kanpotik jasotzen denean edo erabiltzaileari erakutsi nahi diogunean, objektu mota hau erabiltzen dugu. Beraz, objektu mota hau erabat lotua dago erabiltzailearen interfaze grafikoekin.

Bigarren notazio mota honetan, interfaze-objektuarekin ikusi dugun legez, objektu mota bakoitza espezializatu egiten dugu, hau da, aplikazioan objektu bakoitzak izango dituen ardurak eta funtzionalitateak kapsulatu egiten ditugu.

Ildo horretatik, bigarren objektu motak, entitate-objektuak, objektuen iraunkortasuna kudeatzen du. Objektu horiek Klase Diagrametan aurkitzen dira eta Klase Diagramaren egitura berari orpoz orpo jarraitzen diote, hau da, domeinuaren objektuak izango dira eta iraunkorrak bihurtu nahi ditugunean aplikazioaren entitate nagusiak izango dira. Adibidez, datu-base erlazional bat aukeratu badugu sistemaren iraunkortasuna ziurtatzeko, datu-base horretan entitate eran gorde dugun informazioa objektu mota hau erabilita eraikiko da.

Bukatzeko, hirugarren objektu motak, kontrol-objektuak, domeinuaren logika kudeatzeko balioko digu. Objektu mota horretan, sistemaren portaera dinamikoa egongo da adierazita, eta bera arduratuko da ziurtatzeaz objektuen arteko koordinazioa eragiketa mailan ondo gauzatzen dela. Kontrol-fluxu nagusiak eta ekintzen koordinazioa, kontrol-objektuen esku gelditzen direnez, sistemaren funtzionalitate desberdinak ondo berezituak agertuko dira Kolaborazio Diagraman. Laburtuz, kanpo-harremanak *interfaze-objektuen* esku gelditzuko dira; iraunkortasunaren kudeaketa antolatzeko, *entitate-objektuak* erabiliko ditugu, eta domeinuaren logika *kontrol-objektuen* ardurapean gelditzuko da.



6.20. irudia. Adibidearen Kolaborazio Diagrama.

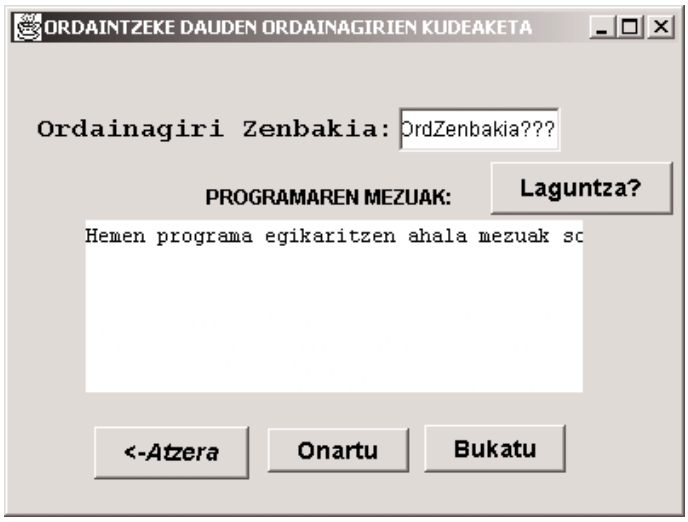
6.3.11.2. Sekuentzia Diagrama

Sekuentzia Diagramek objektuen arteko elkarrekintzak islatzen dituzte denbora-sekuentzian. Grafikoki (**6.24. irudia**) Sekuentzia Diagrama bat taula bat da ezkerretik eskuinera eta goitik behera irakurtzen dena. Ezkerretik eskuinera irakurtzen ari garenean, mezuak nondik nora mugitzen diren ikusten ari gara eta bertikalki aztertzen dugunean, ostera, diagraman zein objektuk duen kontrola une horretan adierazten da.

Sekuentzia Diagrama bat antolatzeko, lehenengo eta behin, eta goiko aldean, elkarrekintzan parte hartuko duten objektuak kokatzen dira, elkarrekintza martxan jarriko duen objektua ezker aldean marraztuz beti. Hori egin eta gero, eta goitik behera, objektuek bidaltzen dituzten mezuak denboran zehar ipiniko dira. Laukizuzen bat marraztuko da bertikalki objektu baten zutabean, objektu horrek kontrola hartzen duela adierazteko.

Askotan, Sekuentzia Diagramak kontrol-fluxu arrunt eta sekuentzialak adierazi beharko ditu eta aitzinean aipatu dugun notazioarekin fluxu horiek marraztea erraza izango da baina, beste batzuetan, kontrol-fluxu konplexuagoak (baldintzak eta begiztak) adierazi beharko ditugu Elkarrekintza Diagramak hala eskatzen duelako. Bigarren egoera hau antolatzeko, notazioa hedatu egin beharko dugu, kontrol-fluxuan baldintzak eta begiztak adierazteko. Begizta bat adierazteko, dagoeneko ezaguna[57] egingo zaigun asteriskoa (*) erabiliko dugu edota, `*[ i:=1 .. n]` adierazpena. Begizta mezu bat baino gehiagoz osatua badago, lerro etenez marraztutako laukizuzen baten barruan sartuko dugu mezu multzoa. Baldintza bat modelatzeko, antzeko bidea erabiliko dugu laukizuzena marrazteko orduan. Adarkatzepuntuek adierazita gelditu behar dutenez, puntu horietan baldintza-klausulak (`[ adina<45]`,`[ (x>7)Λ(kopuru=100)]`...) idatziko dira mezuaren alboan, horrela, argi geldituko da kontrol-fluxua nondik nora mugituko den eta zein baldintzaren pean.

57. Ezaguna, bere semantika Klase Diagraman erabiltzen dugun berbera baita.



6.21. irudia. Aplikazioaren interfaze grafiko nagusia.



IZENA	TELEFONOA	ERAIKINA	BISITA ZENBAKIA	BISITA ORDUA	ARDURADUNA
Xabier Lete	6666666	44553	123	10:30	9999
Izaskun Mendikate	654333	466553	1773	9:30	7799
Arantza Lizarralde	9933499	11553	1443	16:30	9009
Joseba Sarrionandia	9878999	1953	1453	14:30	9609
Peter Sloterdijk	95678999	1183	3243	18:30	9409
Martin Heidegger	9990899	1553	1783	19:30	9209
Xabier Agirregomezk...	9678999	16753	8743	20:30	9059
Pello Korta	9998909	11763	1343	11:30	9079
Arantza Txanponzale	9546799	11233	5443	10:30	9309
Koldo Karanbola	9998999	11623	1663	17:30	9609

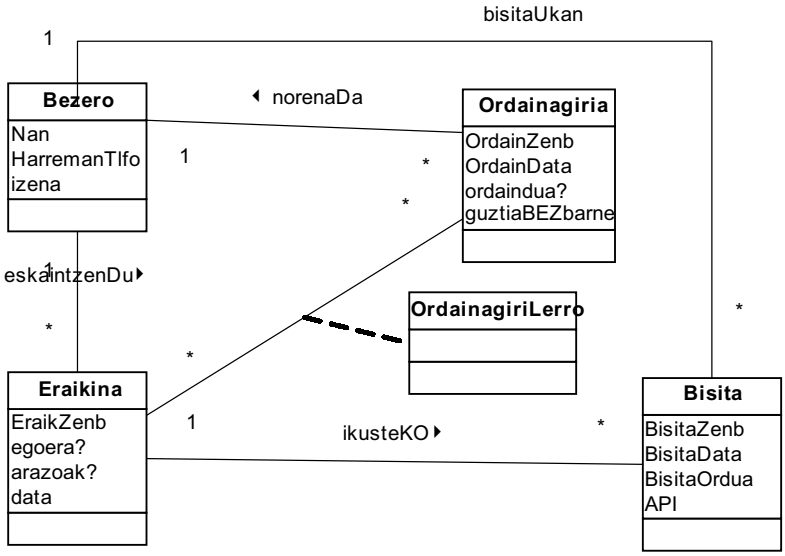
6.22. irudia. Baliogabetutako bisitak.

Hona hemen azaldutakoa zehatz-mehatz aztertuko duen adibide bat. TELETXE eraikuntza-enpresaren erabilera-kasu kritiko bat aztertu nahi dugu. Erabilera-kasu horrek, bankuak itzuli dizkigun ordaindu gabeko ordainagiriak tratatuko ditu. Aplikazioan, bezeroak bere eraikinak saldu nahi ditu eta horretarako, bisitak antolatzen dira eraikinak ikusteko. Eraikin bakoitzaren informazioa, `Eraikina` klaseak kudeatuko du eta **6.23. irudia**n agertzen den Klase Diagraman txertatuko da. `Eraikina` klasearen **arazoak?** atributuak era boolearrean azalduko digu eraikinak arazorik duen ala ez; **data** atributuak, ostera, noiz gertatu den arazoa adierazten du, eta **egoera** atributuak eraikinaren egoera jasoko du honako

balioren bat hartuta: (1) eskaintza: eraikina datu-basean dago eta harentzat ez dago bisitarik antolatuta; (2) bisitan: eraikinak bisita bat edo batzuk dauzka programatuta eta (3) alokatuta: eraikina beste bezero bati alokatu zaio. Eraikin bakoitzaren bisitak `Bisita` klasean biltegiratzen dira. Bertan bisitaren ordua eta data gordetzen dira eta zein arduradunek joan behar duen gauzatzera.

Bezeroak, hilero, ordainagiriak jasoko ditu zenbat ordaindu behar duen jakiteko. Bertan, ordainagiriaren jaulkipen-**data**, ordaindu behar den **kopuru** nagusia (BEZa barne) eta **ordaindua** dagoen ala ez adieraziko dira. Ordainagirian bezeroari fakturatu behar zaizkion eraikin guztiak agertuko dira.

Prestatu nahi dugun erabilera-kasuari *ordaindu gabeko ordainagiriak tratatu* izena jarriko diogu eta honako gertaera-fluxua izango du: administrariak ordaintzeke dagoen ordainagiriaren zenbakia (**6.21. irudia**) sartzen du. Sistemak ordainagiria ez-ordaindu gisa jartzen du (`ordaindua?=false`). Gainera egiaztatu egiten da bezeroak ez dituela beste hiru ordainagiri ordaindu gabe. Hiru baino gehiago ez badu, bukatu egiten da erabilera-kasua. Bestela, hau da, hiru ordainagiri baino gehiago baditu ordaindu gabe, honako hau egiten da: lehendabizi, bezeroaren eraikin guztiak markatzen ditugu datu-basean (`arazoak?=true`). Ondoren, bezeroak eskaintzan (`egoera=eskaintza`) dituen eraikin guztiak ezabatu egiten ditugu. Eta bukatzeko, bisita hitzartuta (`egoera=bisita`) duten eraikinen zerrenda ateratzen dugu eta haien bisitak ezabatu egiten ditugu. Badago egoera bat non bisita ez den ezabatu behar, alegia, eguneko bisitak, bisita horiek ez dira ezabatzen eta haiekin zerrenda bat antolatzen da. Lortutako zerrenda eta bisitara joan behar duen bezeroaren informazioa pantailaratu eta inprimatu egiten dira. Lerro bakoitzak honako informazioa (**6.22. irudia**) izango du: bisita-zenbakia, data, ordua, arduradunaren zenbakia, eraikinaren zenbakia, bezeroaren izena eta haren harremanetarako telefonoa.

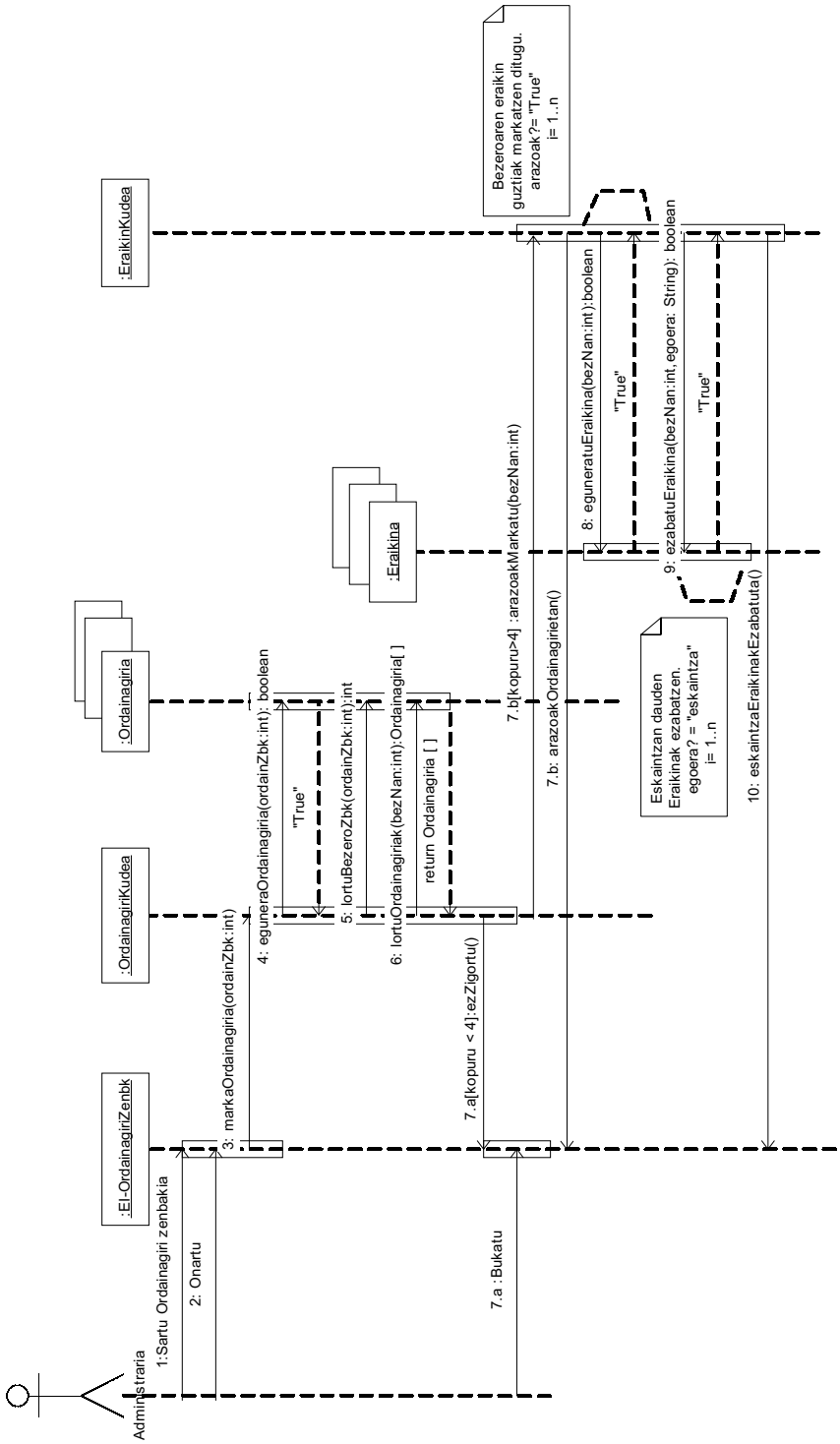


6.23. irudia. Erabilera Kasuaren Klase Diagrama.

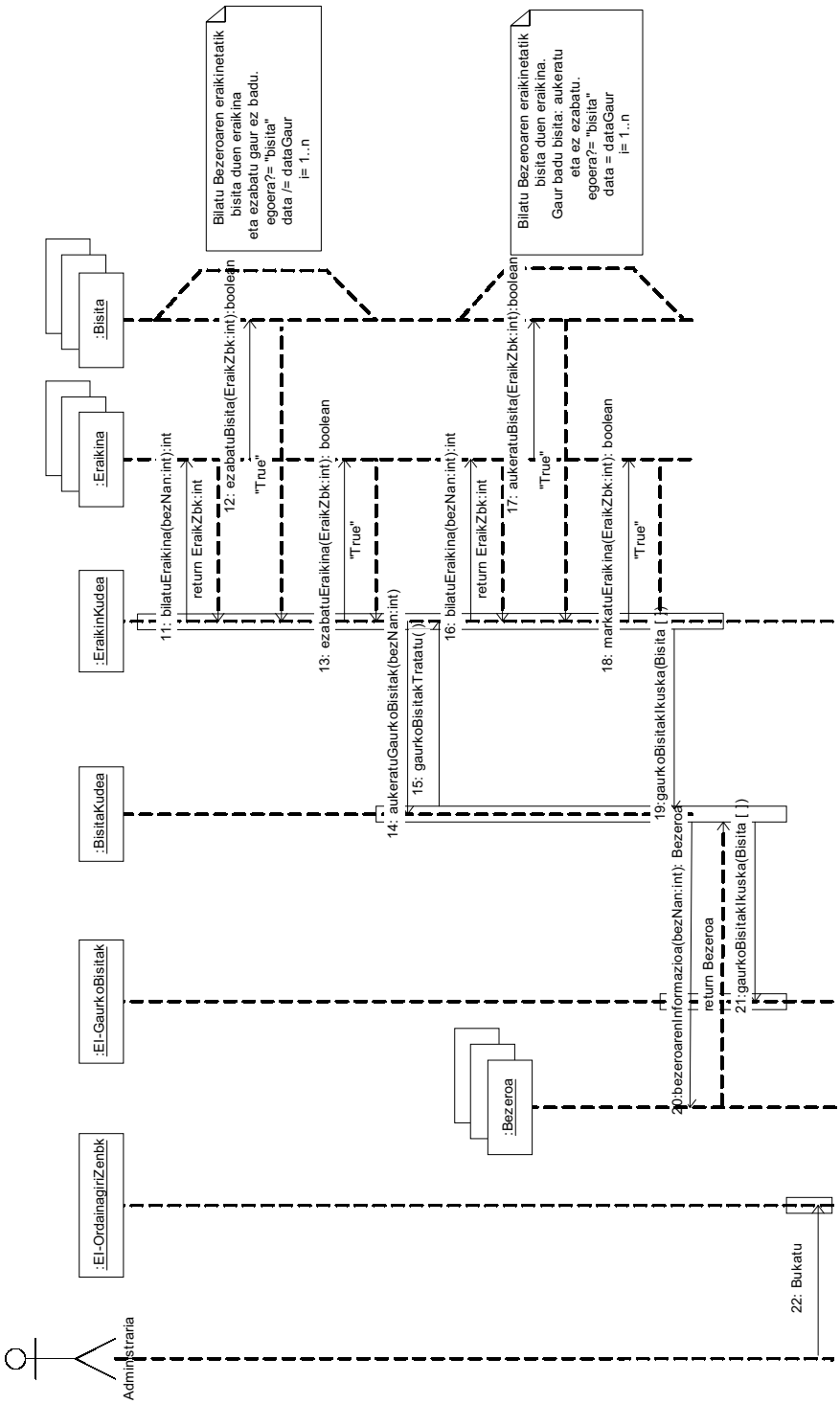
Adibide honetan, erabilera-kasuaren bi Sekuentzia Diagrama mota eraikiko ditugu: bata, bete-betean objektuetara zuzendua (**6.24.** eta **6.25. irudia**k) eta bestea, objektuetan sustraitua ere, baina biltegiratze-prozesua datu-base erlazionalak (**6.26.** eta **6.27. irudia**k) erabilita gauzatzen dena.

Lehenengo Sekuentzia Diagrama motan, erabiltzaileak ordaindu gabeko ordainagiriaren gakoa idazten du erabilera-kasuaren hasierako interfaze grafikoan. `Onartu` botoia zapalduta, sistemak, bezeroaren ordainagiria bilatzen du eta egunera-tu egiten dio bere egoera. Ordainagiriaren informazioa erabilita bezeroaren gakoa eskuratzen da eta halaber bezeroaren ordaindu gabeko ordainagiri guztiak. Kopurua finkatutako mugara heltzen ez bada, erabilera-kasuaren exekuzioa hor amaitzen da. Ostera, mugara heldu bada, aplikazioak hainbat ekintza gauzatu behar ditu bezeroa zigortzeko. Lehenengo eta behin, bezeroaren eraikin guztiak markatzen dira `arazoak?` aldagaian `true` balio boolearra sartuta. Gero, bezeroak eskaintzan jarri dituen eraikin guztiak ezabatu egiten dira, eta bukatzeko, eraikin bakoitzarentzat antolatuta dauden bisitak ezabatu egiten dira. Bisita horiekin salbuespen bat apli-katzen da bisita gaur gauzatu behar bada, egoera horretan, bisitaren informazioa pantailaratu egiten da egun horretako bisita kudeatzeko.

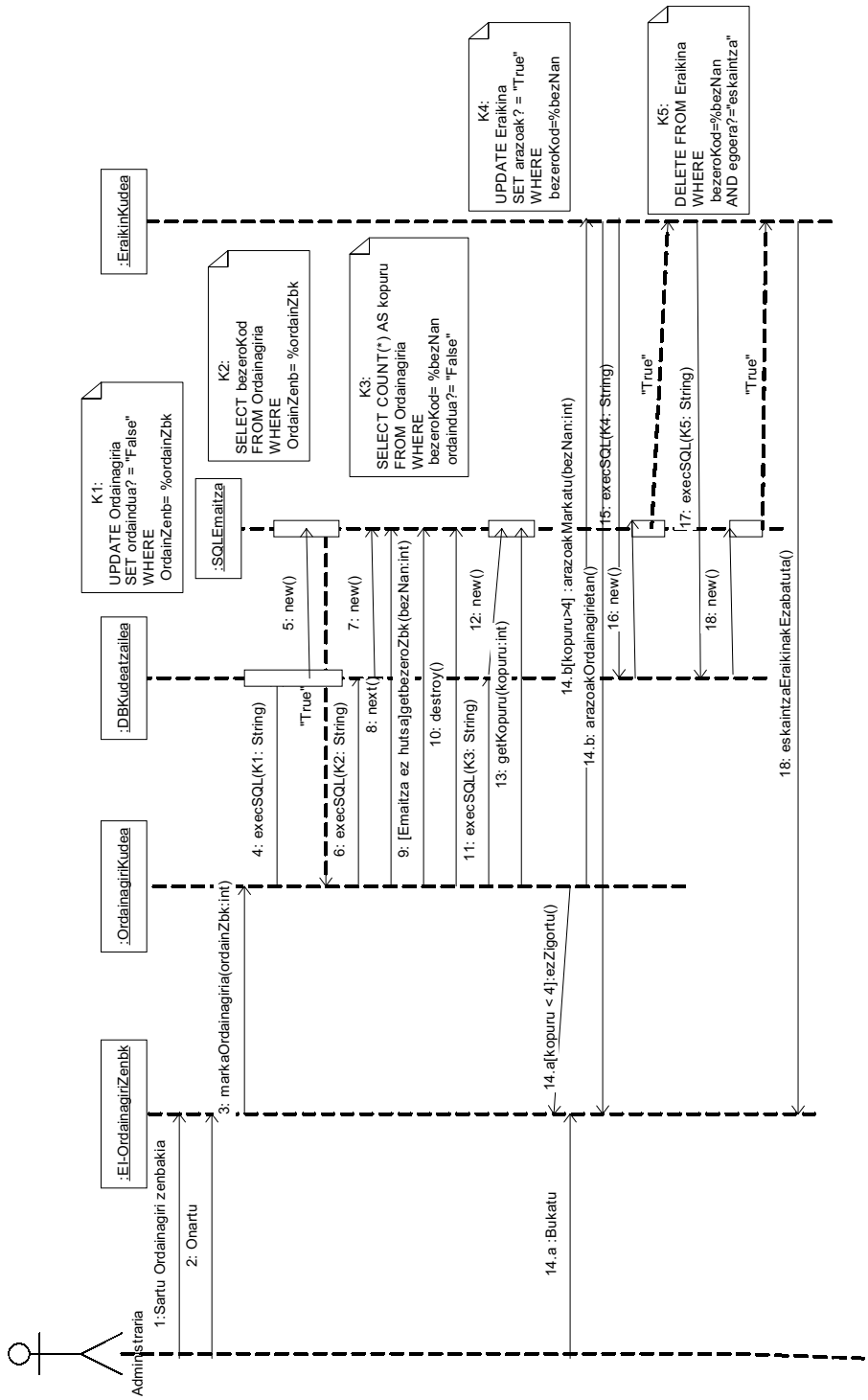
Bigarren Sekuentzia Diagrama motarekin (**6.26.** eta **6.27. irudia**k) objektu bitartekariak erabiliko ditugu datu-basean dagoen informazioa atzitzeko. `DBKudeatzailea` klaseak SQL aginduak egikarituko ditu `execSQL()` metodoa erabilita eta `SQLEmaitza` motako objektu batean utziko ditu emaitzak. Hala, SQL agindua bidali duen objektuak beti daki nora joan aginduaren emaitzak eskuratzeko. Bestela, Sekuentzia Diagramaren exekuzio-fluxua lehenengo motaren hari beretik joango da.



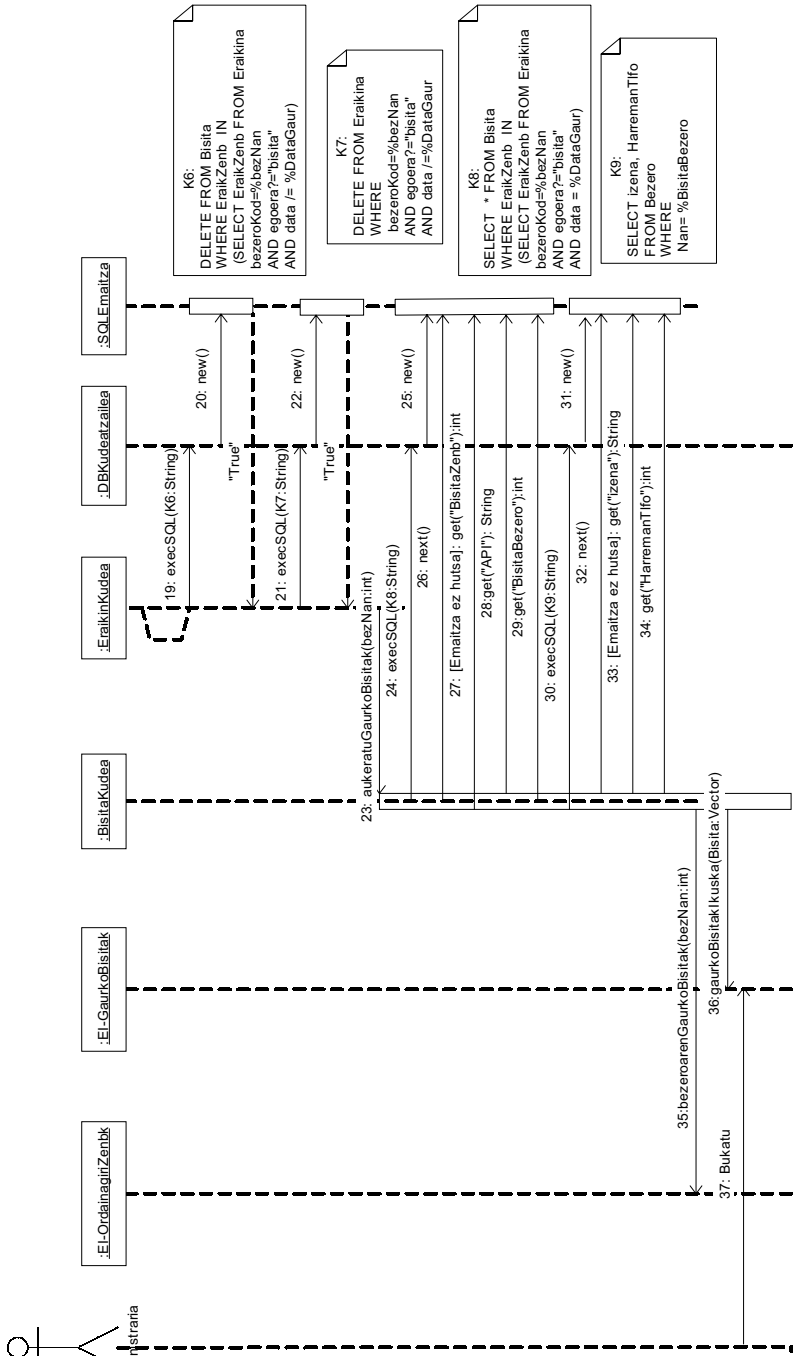
6.24. irudia. Erabilera Kasuaren Sekuentzia Diagrama (I.a)



6.25. irudia. Erabilera Kasuaren Sekuentzia Diagrama (II.a)



6.26. irudia. Erabilera Kasuaren Sekuentzia Diagrama (I.b)



6.27. irudia. Erabilera Kasuaren Sekuentzia Diagrama (II.b)

6.4. PROZESU BATERATZAILEA

Informazio-sistema bat mekanizatzeko, UML erabiltzea erabaki bada analisia eta diseinua gauzatzeko, garapen-taldea inplizituki onartzen ari da notazio bisual estandar bat erabiliko duela garapenaren urrats desberdinetan. Aurreko ataletan, UMLren diagrama bakoitza nolakoa den eta zertarako balio duen azaldu dugu banan-banan eta azaldu ahala nabari da, bakoitzak, garapenaren bizi-zikloaren urrats desberdinetan parte hartzen duela. Batzuk, egokiagoak dira garapenaren betekizunen azterketa gauzatzeko; beste batzuk, ostera, hobeto datoz diseinuaren zehaztasunak adierazteko. Bada, UML hankamotz geldituko litzateke ez balu prozesu bat, diagrama bakoitza noiz eta non erabili behar den zehaztuko lukeena.

UMLren liburu kanonikoetan behin eta berriz aipatzen da UMLren notazioa eta diagramak antzuak liratekeela prozesu estandar bat ez balitz aplikatuko aldi berean.

Prozesu estandar hori prozesu bateratzailea da. Prozesu bateratzaileak hiru ezaugarri nagusi ditu: iteratiboa eta inkrementala izango da, batez ere softwarearen arkitekturan zentratuko da eta erabilera-kasuek bideratuko dute. **6.28. irudia**k prozesu bateratzailearen egitura orokorra jasotzen du. Irudiak bertikalki eta goitik behera prozesuaren lan-fluxuaren garapena zein izango den adierazten du. Horizontalki, lan-fluxu bakoitza garapenaren zein fasetan landuko den eta zenbat landuko den adierazten da. Konturatu fase bakoitzak iterazioak izango dituela eta ez kopuru bera fase guztietan. Iterazioak burututa eta prozesu inkrementala bat garatuta, ur-jauzi bizi-zikloaren flakeziak konpondu nahi dira prozesu bateratzailean.

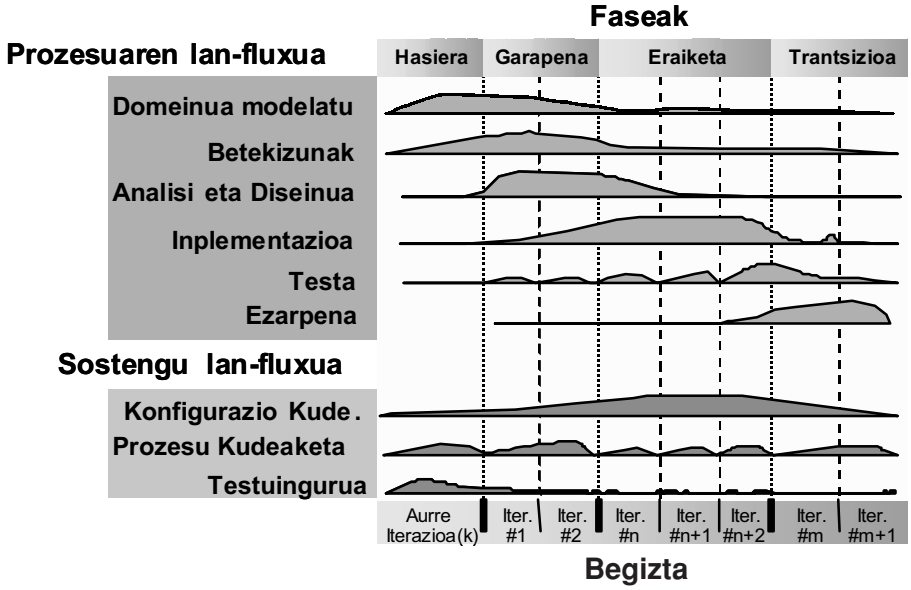
Alde batetik, iterazio[58] laburrak burututa, arriskuak ekiditen dira eta egokitzapenak errazago txertatzen dira garapen-prozesuan. Aldi berean, iterazioa bukatzeko epeak jartzen direnez, garapen-taldeak produktuz ekoitzi behar ditu analisia eta diseinua, bukaera arte itxaron gabe software-produktua edukitzeko. Emaitzak zatika lortzen dira eta ez dena bukaeran. Horretarako, ondo zehaztu behar dira iterazio bakoitzean lortu nahi diren emaitzak eta emaitza horiek lortzeko erabiliko ditugun baliabideak. Baliabideak denboran zehar nola banatuko diren adierazteko, iterazioaren planifikazioa gauzatu behar da agerian utzita nork zer lortu behar duen eta noiz.

Aplikazio informatikoaren analisi eta diseinuaren funtzioaren eta formaren oreka bilatu behar da ematen diren urrats guztietan. Aplikazioaren funtzionalitatea aztertzeko, UMLk erabilera-kasuak erabiliko ditu sistemaren funtzio nagusiak identifikatu eta deskribatzeko, hots, sistemaren betekizun funtzionalak jasotzeko.

58. Larman-en iritziz iterazio batek ez luke bi aste baino gutxiago iraun behar ezta sei aste baino gehiago ere.

Erabilera-kasu horiek garapen-prozesua hasteko ez ezik, garapen osoa bideratzeko ere erabiliko dira. Erabilera-kasu bat ondo deskribatzen bada, oso lagungarria izango zaigu analisian, aplikazioaren objektuak identifikatu eta Klase Diagraman txertatzeko, aplikazioaren ikuspegi estatikoa lortuta. Aldi berean, diseinuan aplikazioaren ikuspegi dinamikoa eraiki nahi badugu, Kolaborazio eta Sekuentzia Diagramak marraztuta, erabilera-kasuaren deskripzio zehatza nahitaezkoa izango zaigu adierazteko nola gauzatuko diren objektuen arteko harremanak.

Erabilera-kasuak erabilita adierazi ditugun betekizun funtzionalak, testuinguruan kokatu behar dira hasiera-hasieratik forma emanez. Forma hori aplikazioaren arkitekturak ematen digu. Halaber, batzuetan, arkitekturaren zati bat emanda etorriko da eta independentea izango da. Aplikazioa plataforma informatiko zehatz batean eraiki behar da eta ezaugarri zehatz batzuk izan behar ditu, eta hori guztia, erabilera-kasuetatik urruti kokatuko da. Adibidez, banketxe batek kutxa automatikoak jarri nahi ditu zenbait herritan eta horrek, nahitaez, baldintzatu egiten du analistak egin behar duen analisi informatikoa. Gauza bera gertatzen da salmenta elektronikoa bultzatzeko sistema informatiko bat eraiki nahi badugu. Aplikazio motak, bai batean eta bai bestean, dezente baldintzatuko ditu softwarearen arkitekturaren zenbait arlo garrantzitsu. Beraz, garapen-prozesua ondo garatzeko, erabilera-kasuen eta arkitekturaren bilakaera paraleloki garatu behar da prozesuaren une bakoitzean. Prozesu bateratzailean zehar, batak bestea berrelikatzen du.



6.28. irudia. Prozesu bateratzailearen egitura orokorra

Bi hitzetan laburtu nahi badugu prozesu bateratzailea nola gauzatzen den, esan dezakegu prozesuaren garapena, beste arlo askotan legez, Paretoren Legea aplikatuta gauzatzen dela, hau da, hasiera-hasieratik, erabilera-kasu guztien % 20 garrantzitsuena aukeratzen da eta baliabideen % 80 inbertitzen da hartan. Hori eginez gero, etorkizuneko sistema informatikoa azpisistematan banatzen da sistemaren konplexutasunak hala eskatzen badu. Eta bukatzeko, aplikazioaren osagaiak eta klase nagusiak identifikatzen dira.

Prozesu bateratzailearen lan-fluxuak (**6.28. irudia**) aztertzen badira, ez da ustekabe handirik aurkitzen, *in illo tenpore*tik ezagutzen baitira lan-fluxu horiek[59]. Laburtuz, honela gauzatzen dira lan-fluxuak: softwarearen betekizun funtzionalak adierazi baino lehen, domeinua modelatu behar da erabilera-kasuak erabilita prozesu nagusiak identifikatuz eta deskribatuz. Domeinua prozesuetan banatu ondoren, informazioa nondik nora mugitzen den zehaztu behar da, non sortzen den, non aldatzen den eta non biltegiratzen den azalduz. Domeinuan erabiltzen diren agiriak iturburu egokiak izaten dira aipatutako informazioa eskuratzeko.

Domeinuaren modelatzea bukatuz gero, aplikazioaren betekizun funtzionalak eraikitzen dira erabilera-kasuak identifikatuta eta deskribatuta. Analisiaren eta diseinuaren lan-fluxuetan, aplikazioaren ikuspegi estatiko eta dinamikoak eraikitzen dira. Ikuspegi estatikoa, gehienetan, aplikazioaren Klase Diagrama erabilita adierazten da hobekien. Ikuspegi dinamikoa, aplikazio estandarretan, Kolaborazio Diagrama eta Sekuentzia Diagrama landuta eskuratzen da. Halaber, batzuetan, beharrezkoa izaten da alde dinamikoa gehiago aztertzea eta horretarako ondo datozkigu Jarduera Diagramak eta Egoera-Makina Diagramak.

Inplementazioan, software-produktua garatzen da aitzinean aipatutako ikuspegiak kontuan izanda. Garapen-prozesua bukatzeko, proba orokorrak eta integrazio-probak gauzatzen dira softwarearen trinkotasuna eta heldutasuna neurtzeko.

Ezarpen lan-fluxuarekin garatutako softwarea bezeroaren egoitzan instalatu egiten da konfigurazio-arazo guztiak behingoan konponduz eta hardwarearen topologia finkatuz. Softwareak agerian utzi behar du egoera errealean funtzionatzen duela, probatuz egoera ohikoei eta, batez ere ezohikoei, aurre egiteko gauza dela.

Aipatutako prozesuaren lan-fluxuak ondo egikaritzeko komeni izaten da paraleloki beste hiru lan-fluxu ere lantzea: konfigurazio-kudeaketa, prozesu-kudeaketa eta testuingurua. Azken lan-fluxuak aztertzen du sistema garatzeko behar dugun azpiegiturak zein izan behar duen eta bizi-zikloaren hasieran lantzen da. Konfigurazio-kudeaketak bizi-zikloan zehar izango diren sistemaren aldaketak kontrolatzen ditu proiektuaren langai[60] desberdinen integritatea mantenduta.

59. Beste hark esango lukeen legez: *nihil novum sub sole*.

60. Langaia ez da soilik iturburu-kodea edo programazioarekin soilik zerikusi duena, analisiaren eta diseinuaren irteera desberdinak ere langaiak izango dira proiektuaren kudeatzailearentzat.

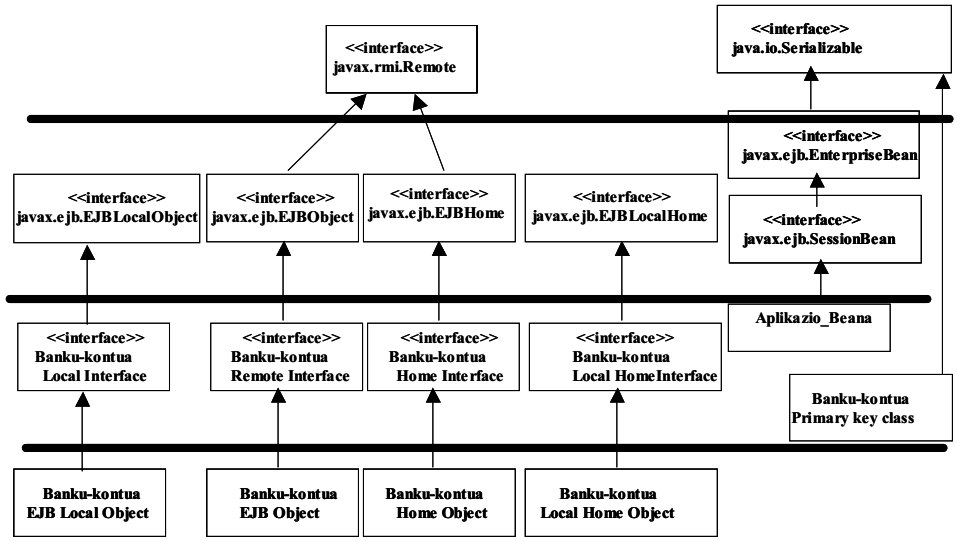
Lan egiteko era hobetu nahi badugu nahitaezkoa da prozesu-kudeaketa bat gauzatzea erakundean. Prozesu-kudeaketa erakundearen *begia* da, ikusten eta neurtzen baitu nola aplikatzen den prozesu bateratzailea erakundean. Nekez hobetuko dugu zerbait ez badugu ondo ezagutzen hobetu nahi duguna, beraz, prozesu bateratzaileak proiektuaren kudeaketa bat izan behar du aplikazioaren baliabide eta emaitzak planifikatzeko eta neurtzeko. Hala, analistak, proiektuaren akatsak identifika ditzake eta ahaleginak egin etorkizuneko proiektuetan ez errepikatzeko. Zenbat eta egokiago egin lan hori, orduan eta prozesu bateratzaile hobea izango du erakundeak esku artean.

6.5. MDA (MODEL DRIVEN ARCHITECTURE)

Informatikaren bilakaerak, bere ibilbide korapilatsuan, abstrakzioarekin egin du topo behin eta berriz; esaterako, analistak, bezeroaren betekizunak abstrakzioak erabilita erakartzen ditu, domeinuaren arlotik ebazpenaren esparrura. Gauza bera egiten du programatzaileak aplikazioaren diseinua programatzeko eta zer esanik ez iturburu-kodea ordenagailuan prozesatzeko.

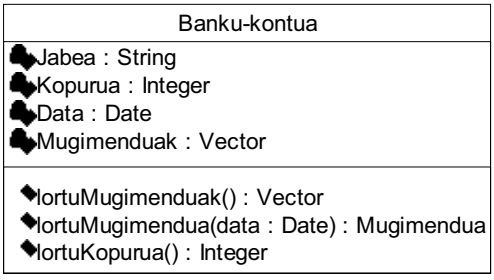
Aurreko orrietan, UMLren diagramak aztertzen ari ginela abstrakzioarekin jolasten ari ginen une oro betekizunak zehazteko erabilera-kasuetan eta erabilera-kasuak egokitzeko Sekuentzia eta Kolaborazio Diagrametan. Zerbait egiaztatu, eguneratu edo konpondu nahi genuenean gorantz eta beherantz mugitzen genuen abstrakzioaren maila aplikazioaren ikuspegi ortogonalak kudeatuz. Hori guztia, egun, eta aplikazioa ertaina edo handia bada areago, tresna informatiko aurreratuak erabilita kudeatzen da. Hala, diagrama batean edozein aldaketa egiten denean, aldaketa hori berehala islatzen da gainerako diagrametan. Testuinguru horretan, aldaketa berak hartzen du garrantzia eta ez aldaketa hori gauzatzeko egikaritu behar den lan astun eta mekanikoak.

Automatizazio-prozesu horretan, beste urrats bat emango dugu eta igo egingo dugu abstrakzio maila. Demagun banketxe bateko banku-kontuak kudeatzeko osagai bat diseinatu eta programatu nahi dugula Javaren J2EE teknologia erabilita. Osagaiak, urruneko eta hurbileko deiak tratatu behar baditu, urruneko (Remote) eta hurbileko (Local) EJB interfazeak inplementatu (**6.29. irudia**) beharko ditu. Era berean, osagaia eraiki, bilatu eta suntsitzeko, faktoria-objektuak diseinatu eta programatu egin beharko dira.



6.29. irudia. Osagaiaren inplementazioa J2EE teknologia erabilita.

Aipatutako prozesua behin eta berriz, eta beti era berean, errepikatu egin beharko da osagai berri bat eraiki nahi dugunean, J2EE teknologiak azpiegitura hori guztia behar baitu osagaiak prozesatzeko. Bestalde, jakin badakigu, diseinatzen ari garen banku-kontu osagaia biluztu egiten badugu teknologiaren jantzia aldenduta, osagaiaren muinarekin (**6.30. irudia**) gelditzen garela eta hori dela benetan interesatzen zaiguna diseinatzaile gisa, bestela esanda, teknologiaren erregulartasunak teknologiak berak konpon ditzala.

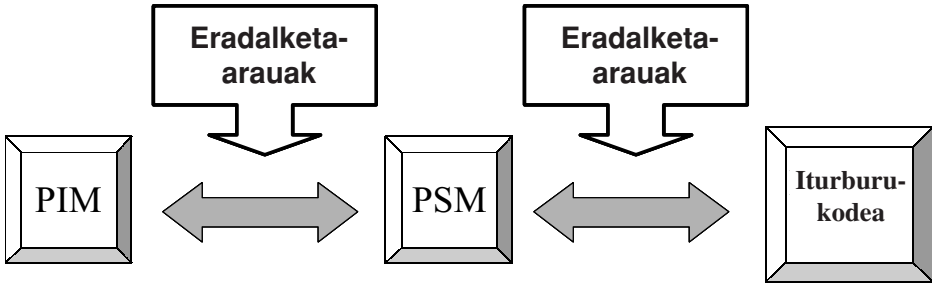


6.30. irudia. `Banku-kontua` klasearen diseinua.

Aztertu dugun adibideak argi uzten du analistak UML diagrama erabilita kontzeptualki sortu duenak eraldaketak izaten dituela inplementaziora eta exekuziora heldu baino lehen. Arazoa gehiago korapilatzen da, gure osagai sortu berria J2EE teknologia erabilita beste osagai batekin harremanetan jarri nahi badugu, .NET

teknologiarekin eraiki denarekin, alegia. Kontzeptualki, analisi mailan, UML erabilita esaterako, ez litzateke inongo arazorik egongo bi osagai horiek harremanetan jartzeko, analistarentzat, bai bata eta bai bestea, klaseak izango baitira eta abstrakzio maila horretan eroso eta lasai aurkitzen da. Beraz, azkenean, arazoaren muinarekin egin dugu topo: nola egitura dezakegu aplikazio informatikoa, plataforma teknologikoen arteko ika-mikak aintzakotzat hartu gabe?

2000ko azaroan, Orlandon bildu ziren OMG erakundearen partaideek galdera bera izan zuten aztergai. Ordura arte, sistema banatuen arkitekturaren azpiegituraren estandarizazioa zuen jomuga OMG erakundeak baina, ikusita softwarearen industriaren aldetik borondate gutxi zegoela arazoa konpontzeko[61], lemari sendo oratu eta haize-orratzari kasu eginda, lurralde berrietara bideratu zuen branka. Lurralde berri horietan, ereduen azterketak hartuko du lehentasun nagusia eta arretagune berri hori tratatzeko, lemazain informatikoak igo egin beharko du abstrakzio maila. Ur berri horietan aplikazio informatiko bat tresna kontzeptual berriekin kudeatu eta eraikiko da eta gehienetan, tresna horiek, halabeharrez, bere tresna informatiko aurreratua izan behar dute asoziatua arrakasta lortzeko.



6.31. irudia. MDA arkitekturaren urrats nagusiak.

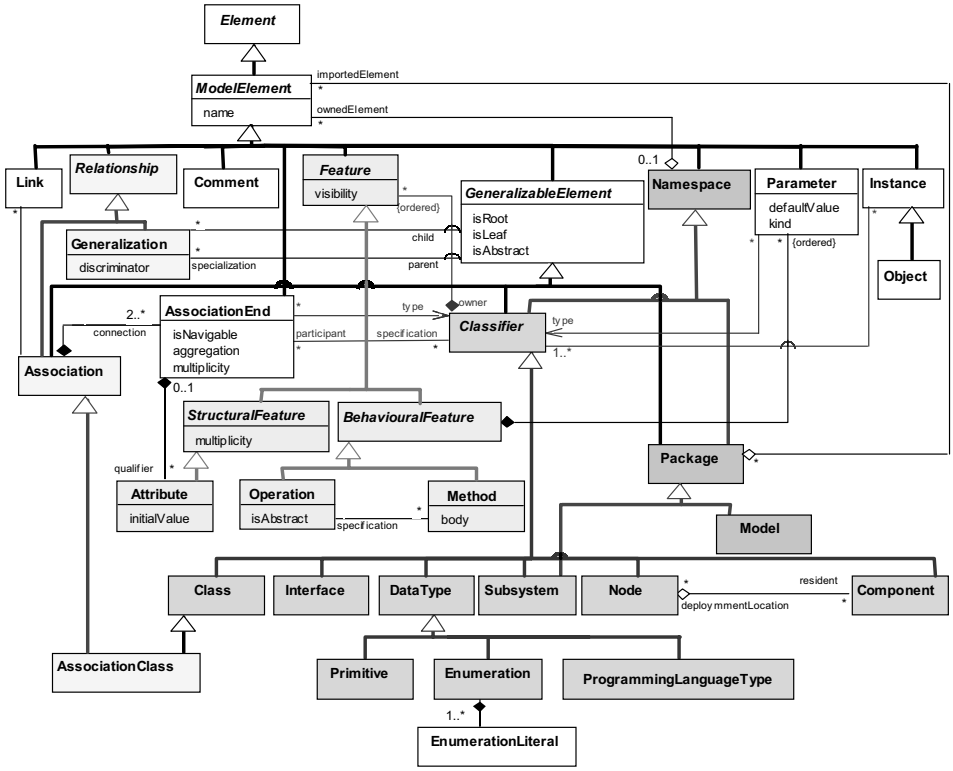
Beraz, OMG erakundearen baitan plataformen arteko gerla bukatutzat emango da eta ereduen trataerak bereganatuko du arreta nagusia. Hemendik aurrera, aplikazio informatiko baten analisia ez da teknologien mosaiko heteroklitoan itota ibiliko, hots, plataforma teknologikoek ez dute baldintzatuko azken ebazpena. Aplikatuko diren arkitekturetan (MDA - Model Driven Architecture) ereduaren azterketak eta kudeaketak gidatuko dute garapen guztia. Arkitektura horietan, hainbat eredu eraikiko dira aplikazioaren logika jasotzeko alde batetik, eta logika hori plataforma teknologiko zehatz batean gauzatzeko, bestetik. Hala, proiektu

61. Ez dago esan beharrik erakunde informatiko ezagun eta arrakastatsu batek dezente lagundu zuela egoera trauskiltzen.

Une horretan OMGk middleware-en arteko gerla bukatutzat eman zuen arkitektura bakoitzak bere lurraldea bereganatu baitzuen eta antzua zatekeen middleware bakarra proposatzea eta bultzatzea. OMG ondorio batera heltzen da: epe luzera plataforma desberdinen arteko zubiak eraikitzen jardutea ez dela estrategia egokiena eta emankorrena.

informatiko baten lehen urratsa (**6.31. irudia**) eratzeko plataformarekiko independentea (PIM - Platform Independent Model) den eredu bat eraiki behar da aplikazioaren prozesuak eta datuak era kontzeptualean adierazteko. Urrats horretan, aplikazioa inplementatzeko erabiliko diren plataformaren ezaugarriak ez dira kontuan hartzen, are gehiago, hor adierazitakoak independentea izan behar du plataformarekiko. Arkitekturaren bigarren geruzak, eraldaketa-arau egokiak aplikatuta, lehenengo urratsean eraikitako eredua egokitu egiten du plataforma (PSM - Platform Specific Model) zehatz batera, plataformarekiko berariazko eredua sortuz. Azken urratsean, eta berriro eraldaketa egokiak aplikatuta, aplikazioaren iturburu-kodea eraikitzen da. Bukatzeko, behin-behineko iturburu-kode sortu berria osatu egin beharko du programatzaileak xehetasunak gehituz.

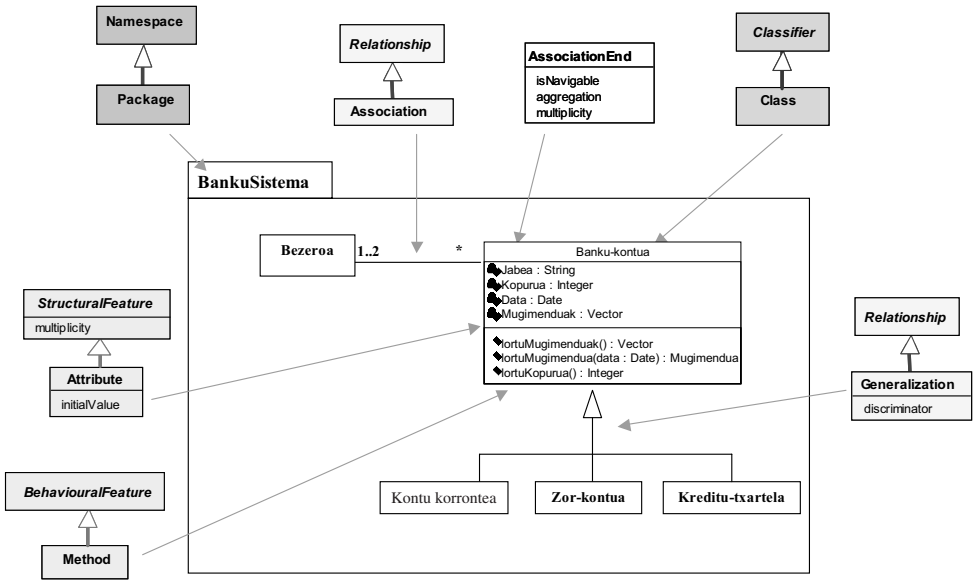
Gehienetan, arkitektura horien konplexutasuna, tresna informatiko aurreratu bat erabilita kudeatzen da, edozein aldaketa edo eguneratze kontrolpean edukitzeko. Arkitektura horietan, edozein *goi-aldaketa* kontzeptual berehala transmiti daiteke iturburu-kodera, eta ondorioz, aplikazioaren diseinuaren sendotasuna bermatu. Halaber, iturburu-kodearen edozein *behe-aldaketa* ere, eredu kontzeptualean islatzen da goranzko ingeniaritza aplikatuta.



6.32. irudia. UMLren metaeredua.

Beraz, abstrakzioa igotzeak lanaren konplexutasuna ere igotzen du, ereduen tipologia era askotakoa izan baitaiteke. Hala nola, prozesuen ereduak izan ditugu baina baita zerbitzuenak, zerbitzuaren kalitatearenak, plataformenak, eraldaketenak, objektuenak eta abar ere. Konplexutasun hori egituratzeko eta gaia bere tokian kokatuz joateko, lau mailako arkitektura proposatu zuen OMG erakundeak. Hona hemen maila bakoitzaren ezaugarri nagusiak:

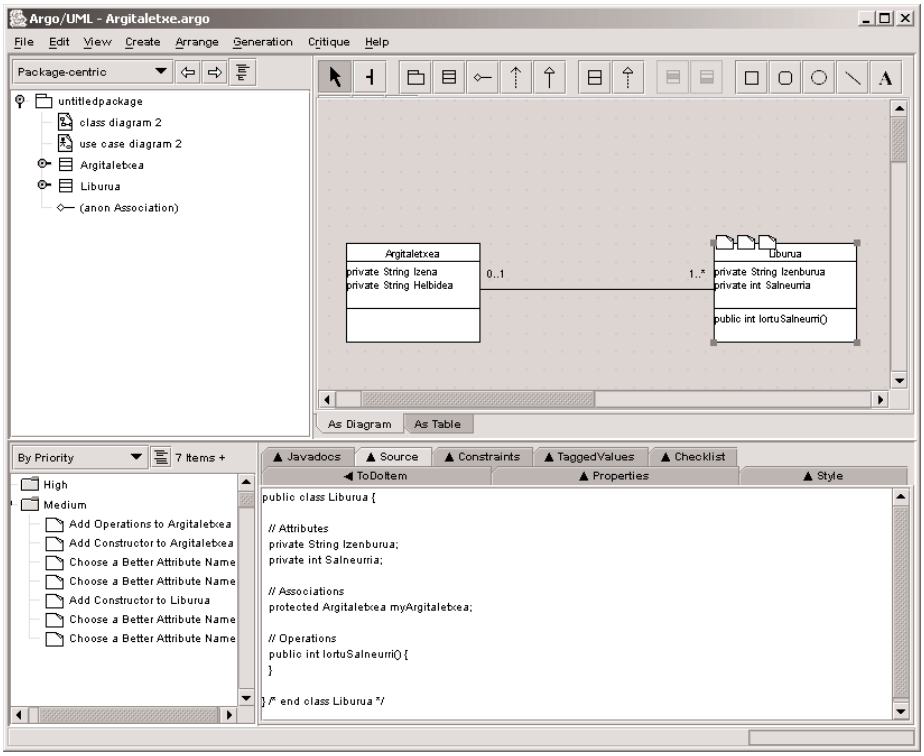
- M0 maila: Maila hau datu errealen maila da. Maila honetan sistema informatiko baten instantzia erreal guztiak kokatzen dira, hau da, aplikazioaren objektuak.
- M1 maila: Maila honek M0 mailako instantzien kategoriak irudikatzen ditu. Aplikazio informatikoaren ereduak maila honetan adierazten dira. Adibidez, UML erabilita sortzen diren diagrama guztiak maila honetakoak dira.
- M2 maila: M1 mailako informazio-ereduak definitzeko erabiltzen diren lengoaiak maila honetan zehazten dira. Lengoaia horiek, M1 mailako ereduen metaereduak dira. Adibidez, UMLren metaeredua (**6.32. irudia**) maila honetan kokatzen da.
- M3 maila: Maila honen entitate bakarra M2 mailako metaereduak definitzeko erabiliko den lengoaia bakarra izango da, hots, metaereduen meta-metaeredua (MOF - Meta Object Facility). Hala, M2 mailako metaeredu guztiak MOF meta-metaereduaren instantziak izango dira.



6.33. irudia. Mailen eta ereduen arteko erlazioak.

6.33. irudiak islatzen du nola erlazionatzen diren maila desberdinak eta eredu desberdinak. Irudiaren erdian, banku-sistema bateko aplikazio informatikoaren Klase Diagrama nagusiaren zati baten egituraketa adierazten da. Aldiz, inguruan azaltzen den informazioa, diagrama hori eraikitzeko eta ulertzeko behar diren UMLren metaereduaren elementuez osatua dago. Kontu egin, irudian, M1 mailaren eta M2 mailaren arteko *elkarrizketa* islatzen dela.

Egun, aplikazio informatiko bat analizatu nahi duenean tresna informatiko aurreratu bat erabilita, programatzailea, M1 mailan kokatzen da kontzeptualki eta maila horretako elementuak grafikoki erabilita, eredu desberdinak eraikitzen ditu. Eredu horien informazioa bateragarria izan dadin tresna guztietan, eta baten adierazitakoa bestean irakurgarri izan dadin, ez da lan erraza. Lehenengo eta behin, estandar bat proposatu behar da eta, gainera, estandar horrek irekia izan behar du. OMG erakundearen XMI (XML Metadata Interchange) estandarrak ezaugarri biak betetzen ditu. Analistak, XML lengoaia erabilita, diseinatu duen ereduaren ezaugarriak biltegiratu, saretik transmititu eta tresna desberdinen artean informazioa elkarrekin trukatu nahi badu, nahikoa du ereduaren informazioa XMI estandarra erabilita egituratzea. Beste aldean, beste tresna informatiko aurreratu batek prozesatuko du informazio hori kontuan izanda XMI erabili dela egituratzeko, eta gauza izango da berriro hasierako eredua arazorik gabe berreraikitzeko.



6.34. irudia. Aplikazioaren Klase Diagrama.

Azter dezagun guztia adibide txiki batekin. Demagun liburutegi batek aplikazio informatiko bat eraiki nahi duela katalogoan dauzkan liburuak kudeatzeko. Hori dela eta, analisiaren kostuak murrizteko, Argo/UML (`http://argouml.tigris.org/`) doaneko tresna informatiko aurreratua erabiliko da analisian. Hasierako betekizunak aztertu ondoren, analistak, **6.34. irudia**n agertzen den Klase Diagrama eraiki du liburutegiko liburuak kudeatzeko. Konturatu, analistak leiho nagusian marrazten duen guztiak bere ordezkoa duela beste leiho guztietan. Hala, ezkerreko leihoan, aplikazioaren paketeak eta paketearen osagaiak nola egituratzen diren adierazten da automatikoki. Leiho nagusiaren beheko aldean beste honako informazio hau agertzen da: tresnak automatikoki sortzen duen aplikazioaren osagai bakoitzaren iturburu-kodea, dokumentazioa eta beste hainbat informazio.

Gainera, analistak eraiki berri den eredua egituratu nahi badu XMI estandarra erabilita, tresnak arazorik gabe egingo du lan hori. Gure adibidean, honako XML fitxategia eraiki du:

```
<?xml version = "1.0" encoding = "ISO-8859-1" ?>
<!DOCTYPE XMI SYSTEM "uml.dtd" >
<XMI>
  <XMI.header>
    <XMI.documentation>
      <XMI.exporter>Argo/UML</XMI.exporter>
      <XMI.exporterVersion>0.7.0</XMI.exporterVersion>
    </XMI.documentation>
    <XMI.metamodel name="uml" version="1.2" />
  </XMI.header>
  <XMI.content>
    <Model XMI.id = "S.100027">
      <name>untitledpackage</name>
      <visibility XMI.value = "public" />
      <isAbstract XMI.value="false" />
      <isLeaf XMI.value="false" />
      <isRoot XMI.value="false" />
      <ownedElement>
        <Class XMI.id = "S.100028">
          <name>Argitaletxea</name>
          <visibility XMI.value = "public" />
          <isAbstract XMI.value="false" />
          <isLeaf XMI.value="false" />
          <isRoot XMI.value="false" />
          <isActive XMI.value = "false" />
          <feature>
            <Attribute XMI.id = "S.100079">
              <name>Izena</name>
              <visibility XMI.value = "private" />
              <ownerScope XMI.value="instance" />
              <changeable XMI.value="none" />
```

Argitaletxea klasea

Argitaletxea klasearen **Izena**

```
            <multiplicity>1</multiplicity>
            <targetScope XMI.value="instance"/>
            <initialValue>
            </initialValue>
            <type> <XMI.reference target="S.100003"/> </type>
            <!- Type is: String ->
          </Attribute>
        </feature>
        <feature>
          <Attribute XMI.id = "S.100086">
            <name>Helbidea</name>
            <visibility XMI.value = "private"/>
            <ownerScope XMI.value="instance"/>
            <changeable XMI.value="none"/>
            <multiplicity>1</multiplicity>
            <targetScope XMI.value="instance"/>
            <initialValue>
            </initialValue>
            <type> <XMI.reference target="S.100003"/> </type>
            <!- Type is: String ->
          </Attribute>
        </feature>
      </Class>
    </ownedElement>
    <ownedElement>
      <Class XMI.id = "S.100029">
        <name>Liburua</name>
        <visibility XMI.value = "public"/>
        <isAbstract XMI.value="false"/>
        <isLeaf XMI.value="false"/>
        <isRoot XMI.value="false"/>
        <isActive XMI.value = "false"/>
        <feature>
          <Operation XMI.id = "S.100101">
            <name>lortuSalneurri</name>
            <visibility XMI.value = "public"/>
            <ownerScope XMI.value="instance"/>
            <isQuery XMI.value="false"/>
            <concurrency XMI.value="sequential"/>
            <isPolymorphic XMI.value="false"/>
            <operationSpecification>
            </operationSpecification>
            <parameter>
              <Parameter XMI.id = "S.100106">
                <name>return</name>
                <visibility XMI.value = "public"/>
                <defaultValue>
                </defaultValue>
                <kind XMI.value="return"/>
                <type> <XMI.reference target="S.100006"/> </type>
```

Argitaletxea klasearen **Helbidea**

Liburua klasea

Liburua klasearen `lortuSalneurri()` metodoa

```
            <!– Type is: int –>
          </Parameter>
        </parameter>
      </Operation>
    </feature>
    <feature>
      <Attribute XMI.id = "S.100096">
        <name>Izenburua</name>
        <visibility XMI.value = "private"/>
        <ownerScope XMI.value="instance"/>
        <changeable XMI.value="none"/>
        <multiplicity>1</multiplicity>
        <targetScope XMI.value="instance"/>
        <initialValue>
        </initialValue>
        <type> <XMI.reference target="S.100003"/> </type>
        <!– Type is: String –>
      </Attribute>
    </feature>
    <feature>
      <Attribute XMI.id = "S.100100">
        <name>Salneurria</name>
        <visibility XMI.value = "private"/>
        <ownerScope XMI.value="instance"/>
        <changeable XMI.value="none"/>
        <multiplicity>1</multiplicity>
        <targetScope XMI.value="instance"/>
        <initialValue>
        </initialValue>
        <type> <XMI.reference target="S.100006"/> </type>
        <!– Type is: int –>
      </Attribute>
    </feature>
  </Class>
</ownedElement>
<ownedElement>
  <Association XMI.id="S.100080">
    <name></name>
    <visibility XMI.value = "public"/>
    <isAbstract XMI.value="false"/>
    <isLeaf XMI.value="false"/>
    <isRoot XMI.value="false"/>
      <connection>
        <AssociationEnd XMI.id = "S.100081">
          <name></name>
          <visibility XMI.value = "public"/>
          <isNavigable XMI.value="true"/>
          <isOrdered XMI.value="false"/>
          <aggregation XMI.value="none"/>
          <multiplicity>0..1</multiplicity>
```

Liburua klasearen **Izenburua**

Liburua klasearen **Salneurria**

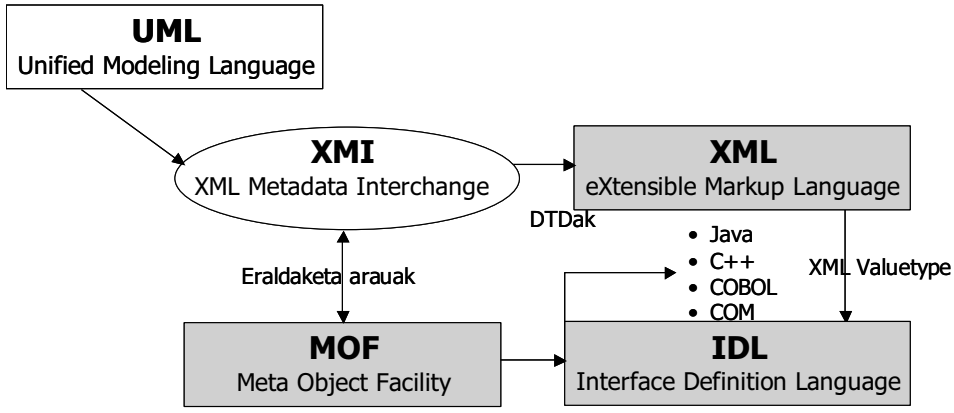
Argitaletxea eta **liburua** klaseen arteko **asoziazio**-erlazioa

Asoziazioaren alde bateko kardinalitatea

```
                <changeable XMI.value="none"/>
                <targetScope XMI.value="instance"/>
                <type> <XMI.reference target="S.100028"/> </type>
                <!- Type is: Argitaletxea ->
   </AssociationEnd></connection><connection>
      <AssociationEnd XMI.id = "S.100082">
                <name></name>
                <visibility XMI.value = "public"/>
                <isNavigable XMI.value="true"/>
                <isOrdered XMI.value="false"/>
                <aggregation XMI.value="none"/>
                <multiplicity>1..*</multiplicity>
                <changeable XMI.value="none"/>
                <targetScope XMI.value="instance"/>
                <type> <XMI.reference target="S.1000  "/> </type>
                <!- Type is: Liburua ->
              </AssociationEnd>
            </connection>
         </Association>
       </ownedElement>
     </Model>......
  </XMI.content>
</XMI>
```

Asoziazioaren beste aldeko kardinalitatea

Kontu egin fitxategi hori prozesagarri bihurtzen dela beste tresna aurreratu batzuentzat, ez bakarrik XML erabilita eraikia izan delako, baizik eta XMI estandarrari (**6.35. irudia**) jarraitzen diolako. **6.35. irudia**k islatzen du lengoaia moten arteko eta mailen arteko informazioa nola egituratzen den.



6.35. irudia. Lengoaia moten arteko erlazioak.

6.6. LABURPENA

Informazio-sistemen analisi eta diseinu motek modelatze bisuala erabili ohi dute bezeroaren betekizunak atzemateko eta komunikazio sendoa eta aberatsa eratzeko softwarearen garapenean parte hartuko duten partaide guztien artean. Analisi eta diseinurako metodologien ereduek eta teknikek informazio-sistemaren abstrakzio mailak arakatzen dituzte sistemaren ikuspegi ortogonala eratzeko. Hala, partaide batek sistemaren arlo estatikoa edo dinamikoa isolatuta ikuska dezake eta arlo horretako eragiketak eta elementuak erabilita eredu partikular hori garatu eta eguneratu. Partaideak jakin badaki, eredu eraiki berriak osatu egiten duela beste ereduek adierazten dutena. Ereduak gauzatzen duen edozein aldaketa berrik bateragarria izan behar du aurretik adierazitakoarekin beste ereduetan. Gehienetan tresna informatiko aurreratuak erabiliko dira bateragarritasuna eta integrazioa bermatzeko. Bertan, informazio-sistemaren osagai bakoitza elementu polimorfikotzat hartuko da eta automatikoki kudeatuko ditu analistek edo diseinatzaileek bizizikloan zehar adierazten dituzten aldaketa guztiak.

6.7. OINARRIZKO BIBLIOGRAFIA

Booch, G.; Rumbaugh, J. eta Jacobson, I. (1999): *El Lenguaje Unificado de Modelado*, Addison Wesley.

Bruegge, B. eta Dutoit, A. (2003): *Object-Oriented Software Engineering: Using UML, Patterns and Java*, Prentice Hall.

Frankel, D. (2003): *Model Driven Architecture: Applying MDA to Enterprise Computing*, John Wiley & Sons.

Hunt, J. (2003): *Guide to the Unified Process Featuring UML, Java and Design Patterns*, Springer.

Jacobson, I.; Booch, G. eta Rumbaugh, J. (2000): *El Proceso Unificado de Desarrollo de Software*, Addison Wesley.

Lano, K. (2005): *Advanced Systems Design with Java, UML and MDA*, Butterworth-Heinemann.

Larman, C. (2002): *UML y Patrones*, Prentice Hall.

Lee, R. eta Tepfenhart, W. (2003): *Practical Object-Oriented Development with UML and Java*, Prentice Hall.

Piattini, M.; Calvo-Manzano, J. A.; Cervera, J. eta Fernández, L. (1996): *Análisis y diseño detallado de Aplicaciones Informáticas de Gestión*, Rama.

Rumbaugh, J.; Jacobson, I. eta Booch, G. (2000): El Lenguaje Unificado de Modelado Manuel de Referencia, Addison Wesley.

6.8. HELBIDE INTERESGARRIAK

Zer dago bertan?	Web gunea
Objektuetara zuzendutako analisia eta diseinua	http://www.cetus-links.org/oo_ooa_ood_methods.html
UMLren baliabideak eta espezifikazioak	http://www.uml.org/
UML materiala	http://www.cetus-links.org/oo_uml.html
UML eta web zerbitzuak	http://www.jeckle.de/unified.htm
Rational tresnaren web ataria	http://www-306.ibm.com/software/rational/uml/
ArgoUML tresna aurreratua	http://argouml.tigris.org/
UML eta patroiak	http://www.craiglarman.com/
Liburu osoa pdf formatuan	http://manuales.astalaweb.com/Manuales/UML.asp
Upgrade aldizkari elektronikoaren zenbaki berezia	http://www.upgrade-cepis.org/issues/2004/2/upgrade-vol-V-2.html
ATI erakundearen *Novatica* aldizkariaren zenbaki berezia	http://www.ati.es/novatica/2004/168/nv168sum.html

7. Diseinu-patroiak

7.1. SARRERA

1987an Ken Beck estatubatuarra aholkulari ari zen Ward Cunningham-ekin Tektronix enpresarentzat. Enpresa horretako Semiconductor Test Systems Group garapen-taldeak arazoak zituen erabiltzailearen interfazearen diseinu zehatz bati amaiera emateko. Beck-ek eta Cunningham-ek aukera ezin hobea aurkitu zuten azken boladan buruan bueltaka zuten ideia bat garatzeko.

Berriki, Christopher Alexander arkitekto eta pentsalariaren *The Timeless Way of Building* liburuaren irakurketak liluratuta utzia zuen Ken Beck. Liburuan aurkezten den arkitekturaren ikuspegiak antzekotasun handiak zituen softwarea garatzeko erabiltzen den metodologiarekin. Zergatik ez, orduan, planteatu softwarearen garapena, Alexanderrek proposatzen zuen ildotik?

Aipatutako liburuaren oinarriak eta nondik norakoak sendotzeko, Alexanderren *Notes on the Synthesis of Form* eta *A Pattern Language* liburuak aztertu zituen Beck-ek proposamen osatuago bat egituratzeko. Laburbilduz, Alexanderrek, arkitekturan ematen ziren antzekotasunak kudeatzeko patroiak proposatzen zituen etxegintza eta hiri-diseinua egituratzeko:

> Each pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution.
>
> The pattern is, in short, at the same time a thing, which happens in the world, and the rule which tells us how to create that thing, and when we must create it. It is both a process and a thing; both a description of a thing which is alive, and a description of the process which will generate that thing. (Alexander, 1979: 247)
>
> These patterns in our minds are, more or less, mental images of the patterns in the world: they are abstract representations of the very morphological rules which define the patterns in the world. However, in one respect they are very different. The patterns in the world merely exist. But the same patterns in our minds are dynamic. They have force. They are generative. They tell us what to do; they tell us how we shall, or may, generate them; and they tell us too, that under certain circumstances, we must create them. Each pattern is a rule which describes what you have to do to generate the entity which it defines. (Alexander, 1979: 181-182)

Softwarearen garapenean ideia horien analogiak bilatuz, Cunningham-ek eta Beck-ek *Where do Objects come from?* lana aurkeztu zuten OOPSLA-87 biltzarrean. Aurkeztu zituzten patroiak abstrakzio maila handikoak zirenez entzuleak liluratuta gelditu ziren lanarekin, baina aldi berean egarriz patroi zehatzagoak eskatuz.

Garai horietan, Erich Gamma bere doktore-tesia idazten ari zen Zuricheko Unibertsitatean eta Andre Weinand-ekin batera ET++ tresna informatikoa garatu zuen. Tesia idatzi eta tresna garatu ahala, Gamma-k argi ikusten zuen diseinu-egitura batzuk sarri errepikatzen zirela eta klase soilak ez zirela nahiko egitura horien semantika eta egitura islatzeko.

1990ean, EcoopOopsla-90 biltzarrean, Bruce Anderson-ek hitzaldi bat eman zuen eta aukera izan zuen Gamma-rekin eta Richard Helm-ekin patroiei buruz hitz egiteko. Biltzarra bukatu eta gero, Gamma-k eta Helm-ek harremanetan jarraitu zuten eta bost urteren buruan ezaguna eta arrakastatsu bihurtuko zen *Design Patterns* liburuaren lehen urratsak gauzatu zituzten klasiko eta kanoniko bihurtuko zen patroien katalogoa proposatuz.

1991n, eta beste OOPSLA biltzar batean, Anderson-ek harremanetan jartzen ditu Gamma eta Helm, Ralph Johnson-ekin eta John Vlissides-ekin. Laurek[62] hiru urte geroago 1994ko OOPSLA biltzarrean *Design Patterns: Elements of Reusable Object-Oriented Software* liburua aurkezten dute. Softwarearen ingeniaritzaren arloan liburu kanonikoez hitz egin behar baduzu irakurle eta ondo gelditu zure entzuleen aurrean, hemen duzu aukera ederra. Addison-Wesley argitaletxearentzat ere ez zen izan mauka txikia, aipatutako biltzarrean 750 ale saldu baitzituen, inoiz baino zazpi aldiz gehiago.

Pantxo eta Pello aspaldiko lagunak, hiriko autobus-geltokian aurkitu dira ezustean. Pantxok, oraintsu, izena aldatu du eta Swami Shunyata deitzen da:

Pantxo - Kaixo Pello, aspaldiko !

Pello - Kaixo Pantxo.

– Barka, orain Swami Shunyata deitzen nauk, hain zuzen, oraintxe bertan, Hatha-yoga praktikatzera noak.

– Ni ere iaztik ari nauk.

– Guk asanak lantzerakoan, Swami Sivananda jakintsuaren seriea lantzen diagu: Sarvangasana eta Halasana asana indartsuekin hasten gaituk lehendabizi eta Savasana leuna, bukaerarako uzten diagu.

– Guk bestetara egiten diagu, berotzeko, Suryanamaskar pixka bat eta bukatzeko, Sarvangasana eta Shirshasana.

62. Harrezkero, irakurle, softwarearen ingeniaritzaren arloan *Gang of Four* edo *GOF* akronimoa agertzen zaizunean, lau izen horiekin ordezka ezazu delako akronimoa.

Pantxoren asana-serieak gogorretik bigunera egiten du jauzia, beti asana osagarriak tartekatuz. Pellok, ostera, estrategia orokor bera erabiltzen du baina saioaren hasieran gorputzeko muskuluen tenplea egokitu egiten du asanak egikaritzeko. Batek, egoera lasaian bukatzen du asana-sesioa, Savasana gozoarekin, eta besteak, ostera, egoera patxadatsuan hasi, eta bukaerarako uzten ditu Sarvangasana eta Shirshasana asana klasikoak.

Yoga-praktikatzaile bati azken paragrafoa eta goiko elkarrizketako bi parte-hartzaileen azken hitzen esanahia ulertzea ez zaio batere kostatuko, lexiko hori ezaguna baitzaio eta aurrezagueraz baliatuko da esanahia aurkitzeko. Badaki, Savasana asana, hildakoaren posturarekin parekatu behar dela eta etzanda patxada lasaian gauzatzen dela. Halaber, badaki jakin ere, Suryanamaskar seriea hamabi mugimenduz osatutako ariketa-bilduma dela eta Sarvangasana/Shirshasana/ Halasana asanak, ostera, buruz behera jarri beharreko horietarikoak direla. Beraz, delako yoga-praktikatzaileak, elkarrizketako parte-hartzaileek bezala, ez du azalpen handirik behar elkarrizketa ulertzeko, Shirshasana asana gauzatzeko era bera baita Delhin eta Lekeition.

Patroiekin antzeko egoerak aurki ditzakegu aplikazio informatiko bat garatu nahi dugunean eta analista bi ditugunean aurrez aurre:

> A1 – Nik datu-basearen kudeaketarako Singleton eta Façade patroiak erabiliko nituzke.
>
> A2 – Ados, baina proposatzen duzun interfaze grafiko horrentzat primeran letorke Observer eta Composite patroiak erabiltzea.

Yogarena bezain kriptikoa elkarrizketa hori ere. Argi ikusten da bai batean eta bai bestean, parte-hartzaileek ezaguera adostu zehatza eta ez-arrunta erabiltzen dutela informazioa transmititzeko. Ez dute denborarik galdu behar, zer diren Observer eta Singleton patroiak azaltzen, bai batak eta bai besteak baitakite zer diren eta zertarako diren egokiak.

Patroi bat *testuinguru* batean kokatua dagoen *arazo* bati ematen zaion *erantzun orokor* bat dela badiogu bakarrik, ez dugu asko argitzen zertarako balio dezakeen patroi batek softwarearen ingeniaritzan. Definizioa oso orokorra da, nahiz eta ezagunena eta hedatuena izan. Normala da hori gertatzea zerbait konplexua definitu nahi dugunean, ez baitira nahikoak bi lerro eta hiruzpalau hitz, eremu semantiko zabalekoak erabiltzea. Definizioak berdin-berdin balio dezake patroiak zer diren definitzeko edo nola atera zaitezkeen depresio-egoera batetik antsiolitikoak erabili gabe adierazteko. Beraz, utz ditzagun definizio orokorrak eta pertsonaliza dezagun patroien azterketa, hurbilpena eta zehaztasuna bilatuz. Niretzat patroiak, klase arrunt bat eta diseinu osoaren tartean dauden abstrakzioak izango

lirateke[63]. Batzuetan patroia klaseen egiturara hurbilduko da klase espezializatu baten tankera hartuz; beste batzuetan, ostera, klase bakarra ez da aski izango patroiaren semantika jasotzeko eta Klase Diagrama txiki bat beharko dugu patroia egituratzeko.

Hedapena	Eraikitze-prozesurako patroiak	Egituratze-prozesurako patroiak	Portaera kudeatzeko patroiak
Klasea	*Factory Method*	*Adapter (*klasea*)*	*Interpreter* *Template Method*
Objektua	*Abstract Factory* *Builder* *Prototype* *Singleton*	*Adapter (*objektua*)* *Bridge* *Composite* *Decorator* *Facade* *Flyweight* *Proxy*	*Chain of Responsibility* *Command* *Iterator* *Mediator* *Memento* *Observer* *State* *Strategy* *Visitor*

7.1. irudia. Diseinu-patroien katalogoa.

Agerikoa da objektuetara zuzendutako programazio-lengoaiek ez dutela lehenetsia tresnarik klaseen instantziak kudeatzeko. Baina batzuetan adibidez, klaseen instantzia bakarra eduki nahi dugu atzipen-puntu bakarrarekin. Horretarako, patroietan aditua denak badaki Singleton patroiak arazoari dotore erantzuten diola eta ez du asko pentsatu beharrik ebazpena egituratzeko.

Beste batzuetan, objektuen arteko mendekotasun ezagun bat kudeatu nahi da: objektu baten egoera aldatzen bada, berehala banatu nahi da egoera berriaren informazioa beste objektu batzuetara. Horretarako ere, gure patroi-adituak badaki Observer patroiak buruhausteak ekiditeko balioko diola eta ahalegin txiki bat eginda, burututa izango duela aplikazioaren arlo hori.

63. *Hireak ere ez dik distira handirik,* esango du norbaitek eta egia da baina, nirea da. Gainera, Groucho Marx parafraseatuz: definizio hau dut hemen eta orain baina ez bazaizu atsegin beste bat ere laster eskura dezaket.

Geroago ikusiko dugun legez, aipatutako patroiek objektuaren bizi-zikloaren urrats desberdinetan dute eragina eta, gainera, eragin hori era desberdinean gauzatzen da. Batzuetan (Singleton), objektuaren eraikitze-aldian du eragina patroiak eta beste batzuetan (Observer), ostera, bizi-zikloan zehar izango duen portaerak eskuratuko du garrantzia. Geroago aztertuko ditugun patroiak sailkatzeko, GOF taldeak *Design Patterns* liburu ezagunean erabili zuen taxonomia (**7.1. irudia**) bera erabiliko dugu hemendik aurrera. **7.1. irudia**n ikusten den legez, patroien katalogoak hogeita lau patroi ditu eta patroi bakoitza zutabe batean kokatzen da, non izango duen eragina adierazteko. Halaber, patroiaren hedapena hartzen da kontuan, non aplikatuko den patroia azpimarratzeko, klasean edo objektuan. Klase mailan aplikatzen denean patroia, erlazio estatiko bat adierazten ari garela esan nahi du eta gainera konpilazio-garaian eratuko dela. Bestalde, objektu mailan aplikatzen denean, erlazioa, dinamikoagoa izango da eta gainera exekuzio-garaian gauzatuko da.

GOF taldeak patroi bat deskribatu nahi duenean bere *Design Patterns* liburuan, Alexanderrek proposatutako formatuan oinarrituko da. Beraz, nahitaezkoa zaigu Alexanderren proposamena ezagutzea gero liburuan agertzen dena ulertzeko:

> For convenience and clarity, each pattern has the same format. First, there is a picture, which shows an archetypal example of that pattern. Second, after the picture, each pattern has an introductory paragraph, which sets the context for the pattern, by explaining how it helps to complete certain larger patterns. Then there are three diamonds to mark the beginning of the problem. After the diamonds there is a headline, in bold type. This headline gives the essence of the problem in one or two sentences. After the headline comes the body of the problem. This is the longest section. In describes the empirical background of the pattern, the evidence for its validity, the range of different ways the pattern can be manifested in a building, and so on. Then, again in bold type, like the headline, is the solution the heart of the pattern which describes the field of physical and social relationships which are required to solve the stated problem, in the stated context. This solution is always stated in the form of an instruction so that you know exactly what you need to do, to build the pattern. Then, after the solution, there is a diagram, which shows the solution in the form of a diagram, with labels to indicate its main components.
>
> After the diagram, another three diamonds, to show that the main body of the pattern is finished. And finally, after the diamonds there is a paragraph which ties the pattern to all those smaller patterns in the language, which are needed to complete this pattern, to embellish it, to fill it out.
>
> There are two essential purposes behind this format. First, to present each pattern connected to other patterns, so that you grasp the collection of all 253 patterns as a whole, as a language, within which you can create an infinite variety of combinations. Second, to present the problem and solution of each pattern in such a way that you can judge it for yourself, and modify it, without losing the essence that is central to it.
>
> (Alexander *et al.,* 1977: x- xi)

GOF taldeak Alexanderren proposamena hartu eta ingurune informatiko batera egokitu zuen formatua. Horretarako, patroi bakoitza deskribatzeko, hamabi ezaugarri erabiltzen ditu patroiaren arloak arakatzeko:

– Intentzioa edo xedea. Ezaugarri honek patroiaren deskribapen sintetiko bat gauzatu behar du, laburki adieraziz, zer egiten duen patroiak eta zein diseinu-arazo konpontzen duen.

– Honela ere ezagutua. Patroiari beste ordezko izen bat jarri nahi bagenio zein izango litzatekeen adierazten du ezaugarri honek.

– Motibazioa. Atal honetan patroiaren proposatzaileak jokaleku informatiko bat aurkezten du diseinu-arazo bat azaltzeko. Atalak, hasieran, patroiak nola konpontzen duen diseinu-arazo zehatza azaltzen du eta gero, abstrakzio bat egiten da eta ebazpen orokorra proposatzen da diseinu-arazo berbera aurkitzen denean patroia aplikatzeko. Jokalekuak patroiaren deskribapen abstraktua ulertzen lagunduko digu.

– Aplikazioa. Atal honetan azaltzen da zeintzuk diren egoera egokienak patroia aplikatzeko.

– Egitura. OMT[64] notazioa erabilita patroiaren klaseen arteko erlazioak eta loturak adierazten dira grafikoki atal honetan. Beharrezkoa ikusten denean Sekuentzia Diagrama ere marrazten da.

– Parte-hartzaileak. Goiko atalean marraztutako klaseak ezaugarri honetan deskribatzen dira banan-banan, bakoitzaren ardurak zehaztuz.

– Lankidetza. Atal honetan adierazten da nola gauzatzen den elkarlana patroiaren parte-hartzaileen artean.

– Ondorioak. Atal honetan patroiaren aplikazioak nola hobetzen duen diseinuaren eraginkortasuna azaltzen da. Azpimarratu behar da halaber, batzuetan patroiak diseinuaren arlo bat edo batzuk dezente hobetzen dituela diseinuaren beste eremu batzuk sakrifikatuz. Egoera horiek ere jaso behar dira atal honetan.

– Inplementazioa. Aipatutako liburuan Smalltalk eta C++ lengoaietan nola inplementa daitekeen patroia adierazten da atal honetan. Patroiaren elementuak zatika aztertzen dira inplementazioaren ezaugarriak zein izan daitezkeen identifikatzeko.

– Iturburu-kodea. Inplementazioaren nondik norakoak azaldu ondoren, atal honetan, iturburu-kode zatiak jartzen dira, adierazteko, nola inplementatzen den zehatz-mehatz patroia.

64. Gogoratu irakurle, OMT (Object Modeling Technique) metodoa UML metodoaren arbaso zuzena dela eta Klase Diagramak ia orpoz orpo jarraitzen diola UMLrenari. Beraz, erraza egingo zaizu ulertzea liburuan agertzen diren Klase Diagramak.

– Erabilera ezagunak. Patroia arrakastatsua dela probatzeko eta haren erabilera bultzatzeko softwarearen industrian, atal honetan, nork, non eta nola erabiltzen duen adierazi behar da.

– Patroi erlazionatuak. Patroiak ez dira isolatuta egoten aplikazioaren diseinu orokorrean. Batzuetan, patroi bat beste baten barruan txertatzen da; beste batzuetan, ostera, harremanetan jartzen da beste batekin diseinu orokorrago bat eratzeko. Atal honetan, patroiak, norekin eta nolako harremanak dituen deskribatzen da eta, halaber, norekin dituen antzekotasunak eta norekin desberdintasun handiak.

Ondoko orrietan, **7.1. irudia**n aurkezten ditugun diseinu-patroien egitura eta aplikazio informatikoari itsasten dioten funtzionalitatea azalduko ditugu era kanonikoan, hau da, orpoz orpo jarraituko diegu aipatutako liburuan erabilitako notazioari eta diseinu-patroi bakoitzaren osagaien egituraketari.

7.2. ERAIKITZE-PROZESURAKO PATROIAK

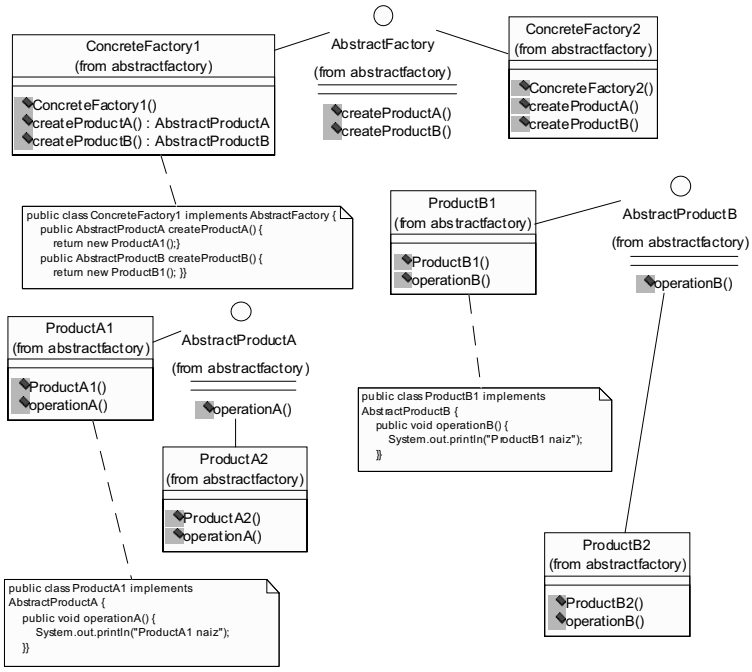
Diseinu-patroi hauekin objektuen instantziazio-prozesua mantsotu eta malgutu egiten da. Diseinu-patroi hauetan objektuen instantziazioa ez da zuzenean eta batbatean egiten, prozesu bat burutzen da instantziazioa kontrolatzeko eta egokitzeko aplikazioaren nahietara. Horrela, aplikazioak beti izango du kudeatuta nork zer eraiki duen eta halaber, nola eta noiz burutu duen prozesua.

7.2.1. Abstract Factory

Diseinu-patroi honek interfaze bat eskaintzen du erlazionatutako objektuak zuzenean eraikitzeko klase zehatzak adierazi gabe.

Patroi honek lau parte-hartzaile ditu:

- AbstractFactory: Parte-hartzaile honek interfaze bat erazagutzen du faktoriak eraikitzeko. Interfazeak AbstractProduct motako objektuak eraikitzeko eragiketak izango ditu definituta, ConcreteProduct motako instantziak eraikitzeko.
- ConcreteFactory: `ConcreteProduct` motako objektuak eraikitzeko klase honetan inplementatuko dira eragiketak.
- AbstractProduct: Interfaze bat erazagutzen du produktu mota bakoitzarentzat. Produktuak motaka sailkatzen dira eta mota bakoitzarentzat `AbstractProduct` bat erabiliko dugu produktuak multzotan antolatzeko.
- ConcreteProduct/Product: Klase honek `AbstractProduct` interfazea inplementatzen du, hala, produktuaren motara egokituta gelditzen dira klasearen eragiketak, hau da, eragiketa berak ondorio desberdinak izan ditzake produktu mota bakoitzean.



7.2. irudia. Abstract Facory diseinu-patroia.

7.2. irudiak islatzen du nola egituratzen diren aipatutako lau parte-hartzaileen erlazioak. Ondoko listatuak azaltzen du bezero batek nola erabiliko lukeen patroiaren osagai bakoitza.

```
public class Bezeroa {
    public static void main(String[] args) {
        // faktoriak eraiki
        AbstractFactory factoryOne = new ConcreteFactory1();
        AbstractFactory factoryTwo = new ConcreteFactory2();

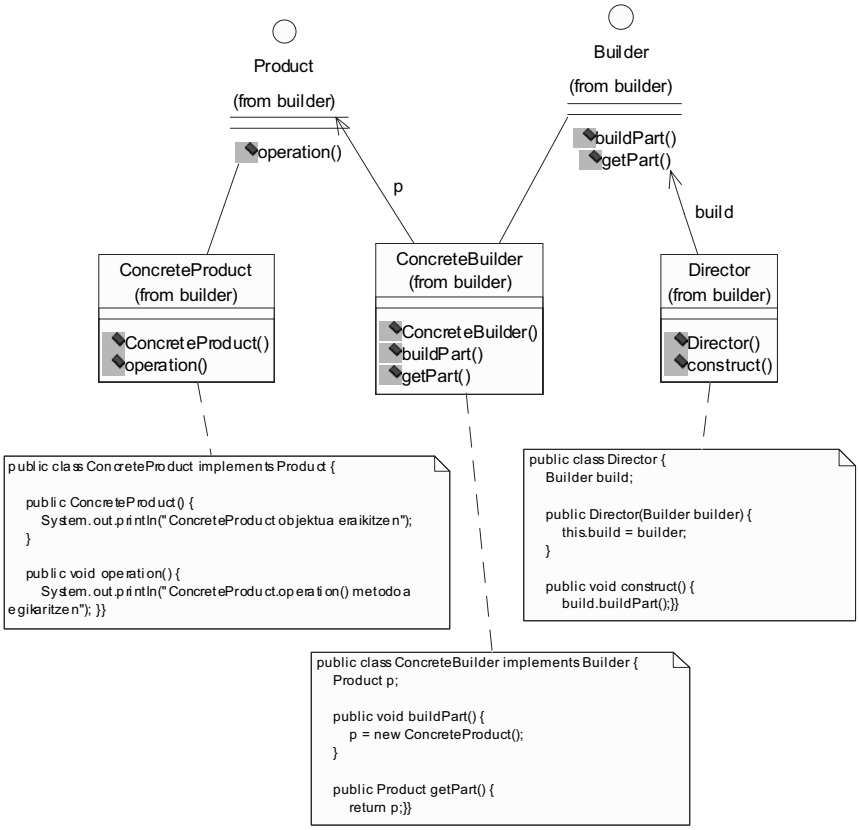
        // faktoriak erabili produktuak eraikitzeko
        AbstractProductA productA1 =
factoryOne.createProductA();
        AbstractProductB productB1 =
factoryOne.createProductB();

        AbstractProductA productA2 =
factoryTwo.createProductA();
        AbstractProductB productB2 =
factoryTwo.createProductB();

        // produktuen metodoei dei egin
        productA1.operationA();
        productB1.operationB();
              productA2.operationA();
           productB2.operationB();}}
```

Patroi honekin zeharka gauzatzen dugu produktuen instantzien eraiketa, `ConcreteFactory` motako objektu bat eraikita. Listatuan ikusten den legez, exekuzio-garaian, `ConcreteFactory` motako instantzia bat eraikitzen da lehendabizi. Hori eginda, `ConcreteFactory` motako objektuak erabilita, produktuak eraikitzen dira eta eragiketak produktu motetara egokituta gelditzen dira.

Aplikazio informatikoan produktuak familietan antolatu nahi ditugunean antzekotasunak eta portaerak ahal denik eta era bateragarrienean egituratuta, patroi honek, zeharo erraztuko digu egituratze-lan hori. Halaber, produktuen liburutegi bat antolatu nahi badugu software berrerabilgarriago bat sortzeko, patroi honek erabiltzen duen teknika inplementazioa ezkutatzeko eta interfazeak eskaintzeko, oso egokia da aipatutako liburutegia eraikitzeko.



7.3. irudia. Builder diseinu-patroia.

7.2.2. Builder

Diseinu-patroi honekin objektu konplexu baten eraiketa banatu egiten da bere adierazpidetik. Horrela, eraiketa-prozesu berak objektuen adierazpide desberdinak sortzen ditu.

Patroi honek lau parte-hartzaile ditu:

- Builder: Parte-hartzaile honek interfaze abstraktu bat erazagutzen du `Product` motako objektuaren zati desberdinak eraikitzeko.
- ConcreteBuilder: Klase honek `Builder` interfazea inplementatzen du eta ardura hartzen du produktuaren zatiak eraiki eta elkartzeko. Baliabide bat ematen digu produktua eskuratzeko. Klase hau izango da ere, objektu konplexuaren adierazpidea kontrolatuko duena.
- Director: Klase honek `Builder` interfazea erabilita eraikiko du objektua.
- Product: Eraikitzen ari garen objektu konplexuaren adierazpide zuzena da. Klase-produktu zehatzak inplementatuko dituen eragiketak definitzen ditu.
- ConcreteProduct: `Product` interfazean definitu ditugun eragiketak klase honetan inplementatzen dira.

7.3. irudiak islatzen du nola egituratzen diren aztertutako bost parte-hartzaileen harremanak. Ondoko listatuak azaltzen du bezero batek nola erabiliko lukeen patroiaren osagai bakoitza.

```
public class Bezeroa {
    public static void main(String[] args) {
        // Builder motako objektua eraiki
        Builder b = new ConcreteBuilder();

        // Director motako objetua eraiki
        Director d = new Director(b);

        // Produktu konplexua eraiki, eskuratu eta erabili
        d.construct();
              Product p = b.getPart();
              p.operation();}}
```

Listatuan ikus daitekeen bezala, bezeroak `Director` motako objektu bat eraikitzen du `Builder` interfazea inplementatzen duen `ConcreteBuilder` motako objektu bat metodo eraikitzaileari erantsita. Gogoratu, metodo eraikitzaileak `Builder` interfazea inplementatzen duen edozein `ConcreteBuilder` motako objektu bat onartzen duela, horrek dakarren malgutasunarekin.

`Director` motako objektu bat eraikiz gero, `construct()` metodoak `ConcreteBuilder` objektuaren `buildPart()` metodoari deitzen dio eraiki nahi dugun produktu zehatza (`ConcreteProduct`) eraikitzeko.

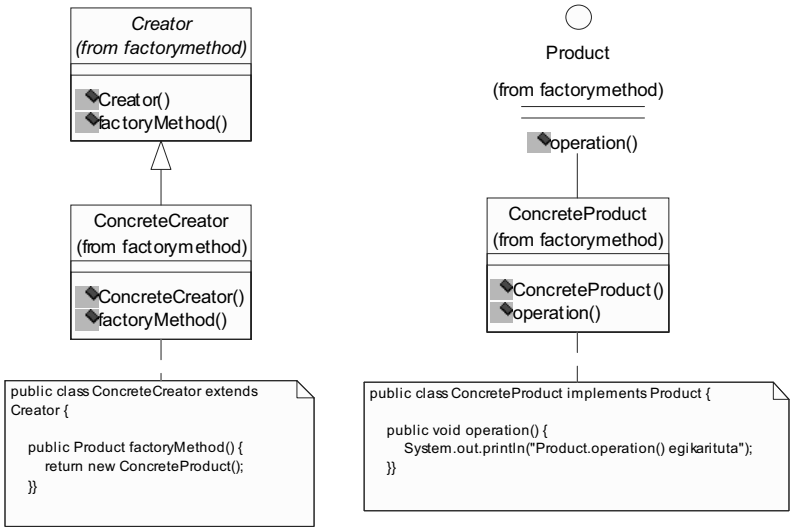
Produktua eraikita, bezeroak, produktua eskuratu egiten du `ConcreteBuilder` motako objektuaren `getPart()` metodoa egikarituta. Bukatzeko eta egiaztatzeko

dena ondo egikaritu dela, bezeroak eraikitako produktu berria erabili egiten du `operation()` metodoa egikarituta.

Patroi honetan, eraiketa-prozesuak adierazpide desberdinak onartzen ditu objektu bat eraikitzerakoan, hau da, objektu konplexu bat eraikitzeko erabiliko den algoritmoa independentea da objektuaren zatietatik. Zatien kontrola eta zati horiek nola lotzen dioten elkarri alde batetik joango da, eta bestetik, objektu konplexua eraikiko duen algoritmoa.

Produktu konplexua ez da bat-batean eta eragiketa bakarrean eraikitzen, objektu konplexua urratsez urrats eraikitzen da `Director` motako objektuaren kontrolpean, produktuaren barne-egitura eta eragiketa-prozesua kontrolatuz une oro.

`Builder` interfazea erabilita, objektu konplexuaren barne-egitura eta bere adierazpidea ezkutatuta gelditzen dira. Horrela, objektuaren barne-egitura aldatu nahi bada, erraz eta era malguan gauzatuko da. Kodean ikusi dugun bezala, bezeroak ez du ezer jakin behar objektuaren barne-egitura definitzeko erabiltzen ditugun klaseez.



7.4. irudia. Factory Method diseinu-patroia.

7.2.3. Factory Method

Diseinu-patroi honetan interfaze bat definitzen da objektu bat eraikitzeko, baina azpiklaseei uzten zaie ardura, adierazteko, zein klase instantziatu. Metodo honek atzeratu egiten du klasearen instantziazioa, lan hori, azpiklaseen bizkarrean utzita.

Patroi honek lau parte-hartzaile ditu:

- Creator: Gehienetan klase abstraktu bat izango da `factoryMethod()` metodoa erazagutzen duena, `Product` motako objektu bat itzuliko du egikaritzen denean. Metodoa `abstract` gisa egon daiteke erazagutua inongo inplementazioarik gabe edo inplementazio bat eduki dezake lehenetsia.
- ConcreteCreator: Klase honetan birdefinitu egiten da `factoryMethod()` metodoa produktu egokia eta zehatza (`ConcreteProduct`) itzultzeko.
- ConcreteProduct: Patroiak eraikitzen duen objektuaren ezaugarriak klase honetan egongo dira. Klase honek `Product` interfaze generikoa eta abstraktua inplementatzen du, `Product` interfazearen metodoei inplementazioa itsatsita.
- Product: Eraikitzen ari garen objektuaren adierazpide zuzena da. `ConcreteProduct` motako objektuak inplementatuko dituen eragiketak definitzen ditu.

7.4. irudiak islatzen du nola egituratzen diren aztertutako lau parte-hartzaileen harremanak. Hurrengo listatuak azaltzen du bezero batek nola erabiliko lukeen patroiaren osagai bakoitza.

```
public class Bezeroa {
    public static void main(String[] args) {
        // Creator motako objektua eraiki
        Creator c = new ConcreteCreator();

        // Produktua eraiki -factoryMethod()- metodoa erabilita
        Product p = c.factoryMethod();

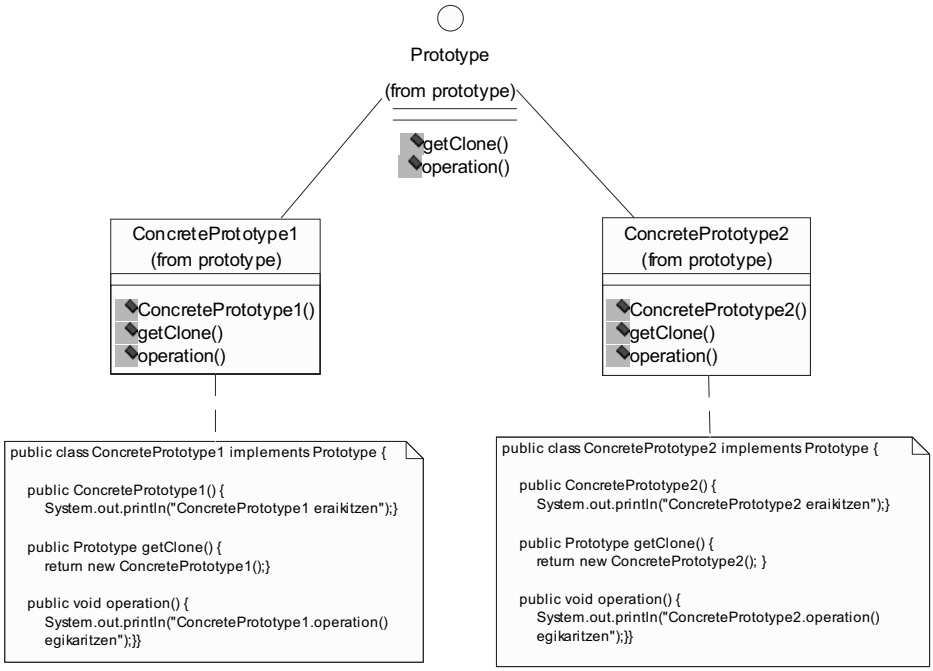
              // Produktua erabili
              p.operation();}}
```

Bezeroak, lehendabizi, `Creator` motako `ConcreteCreator` objektu bat eraikitzen du, `factoryMethod()` deia eraikitzaile zehatz eta kontrolatura egokitzeko. Lan hori, programaren bigarren lerroak gauzatzen du `factoryMethod()` metodoa egikarituta. Emaitza `Product` motako `p` aldagaiari esleitzen zaio.

Bukatzeko, eta ziurtatzeko objektua ondo eraiki dela, eragiketa bat egikaritzeko eskatuko diogu objektu eraiki berriari.

Patroi hau oso erabilgarria da klase batek aurreikusi ezin duenean eraiki behar den objektuaren klase mota zein den. Programaren exekuzio-fluxuak bideratuko du aplikatu behar den eraikitze-prozesua zein izango den.

Patroi honetan, klaseak delegatu egiten dio azpiklaseari eraikitze-prozesuaren ardura. Horrela, eraikitze-prozesuaren klase-hierarkia eta eraikiko diren objektuena, paraleloak izan daitezke, diseinu malguak eta berrerabilgarriak sortuz.



7.5. irudia. Prototype diseinu-patroia.

7.2.4. Prototype

Diseinu-patroi honetan instantzia-prototipo bat erabilita eraiki behar diren objektu motak adierazten dira. Horrela, prototipo hori kopiatuta, objektu berriak eraikiko dira.

Patroi honek bi parte-hartzaile ditu:

- Prototype: Parte-hartzaile honek interfaze bat erazagutzen du prototipoak klonatzeko. Prototipo zehatzak (`ConcretePrototype`) inplementatuko dituen eragiketak hemen definitzen dira. Halaber, hemen definitzen da klonatutako prototipo zehatzak itzuliko duen metodoa.
- ConcretePrototype: Klase honetan inplementatzen dira `Prototype` interfazean definitu diren metodoak. Hau da, hemen inplementatzen dira prototipoa klonatzeko erabiliko den metodoa (`getClone()`) eta prototipo zehatzak izango dituen eragiketak.

7.5. irudiak islatzen du nola egituratzen diren aztertutako bi parte-hartzaileen harremanak. Ondoko listatuak azaltzen du bezero batek nola erabiliko lukeen patroiaren osagai bakoitza.

```
public class Bezeroa {
    public static void main(String[] args) {
        // Prototipo zehatzak eraiki
        Prototype p1 = new ConcretePrototype1();
        Prototype p2 = new ConcretePrototype2();

        // Objektuak sortu prototipo zehatzak erabilita
        Prototype gp1 = p1.getClone();
        Prototype gp2 = p2.getClone();

        // Egikaritu klonatutako objektuen metodoak
              gp1.operation();
        gp2.operation();}}
```

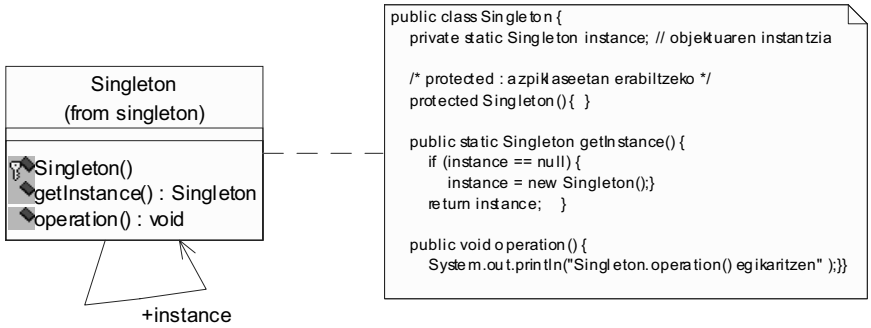
Bezeroak, klonatutako objektuen metodoei dei egin baino lehen, eragiketa bi egin behar ditu:

1. Objektu prototipoak eraiki. Horretarako, `Prototype` motako `ConcretePrototype` motako objektuak eraikitzen dira. Horrela, prototipo zehatzak prest ditugu eraikita eta zain, klonatuak izateko.

2. Objektuak eraiki objektu prototipoak erabilita. Bigarren eragiketa hau egiteko, ez da lan asko egin behar, lehen urratsak egin baitu lan astunena. Objektu prototipoaren `getClone()` metodoa egikaritzen da eta prototipo zehatz (`ConcretePrototype`) bakoitzaren klon bat sortzen da. Hori eginda, arazorik gabe egikaritu ditzake bezeroak klonatutako objektuen metodoak.

Konturatu, patroi honekin aplikazioak, lasai-lasai eta inongo arazorik gabe, objektuak gehitu eta ken ditzakeela dinamikoki exekuzio-garaian. Patroi honekin ez da eraiki behar eraikitze-prozesuaren beste klase-hierarkia berri bat domeinuaren klase-hierarkiarekin batera eta paraleloki.

AbstractFactory eta Builder patroietan bezala, patroi honek bezeroari ezkutatu egiten dizkio eraikitzen ari garen objektuaren klase zehatzak.



7.6. irudia. Singleton diseinu-patroia.

7.2.5. Singleton

Diseinu-patroi honek klase batek instantzia bakarra izango duela ziurtatzen du eta gainera, atzipen-puntu bakarra eta ezaguna sortzen du instantzia eskuratzeko.

Patroi honek parte-hartzaile bakarra izango du:

- Singleton: Klase honek ezaugarri berezi bat du eta eragiketa bat inplementatuta klasearen instantzia bakarra eskuratzeko. Eragiketak (`getInstance()`) `Singleton` motako objektu bat itzultzen du aldi bakoitzean, hain zuzen, `Singleton` klasearen instantzia bakarra izango dena. Eragiketa hori izango da bezeroak `Singleton` klasearen instantzia eskuratzeko bidea.

7.6. irudiak islatzen du nolakoa izango den aztertutako parte-hartzailearen egitura. Ondoko listatuak azaltzen du bezero batek nola erabiliko lukeen patroiaren osagaia.

```
public class Bezeroa {
public static void main(String[] args) {
  // Erabili -getInstance()- metodoa Singleton motako
  // objektuaren instantzia eskuratzeko
    Singleton s = Singleton.getInstance();

          // Singleton motako objektuaren eragiketa bat egikaritu
          s.operation();}}
```

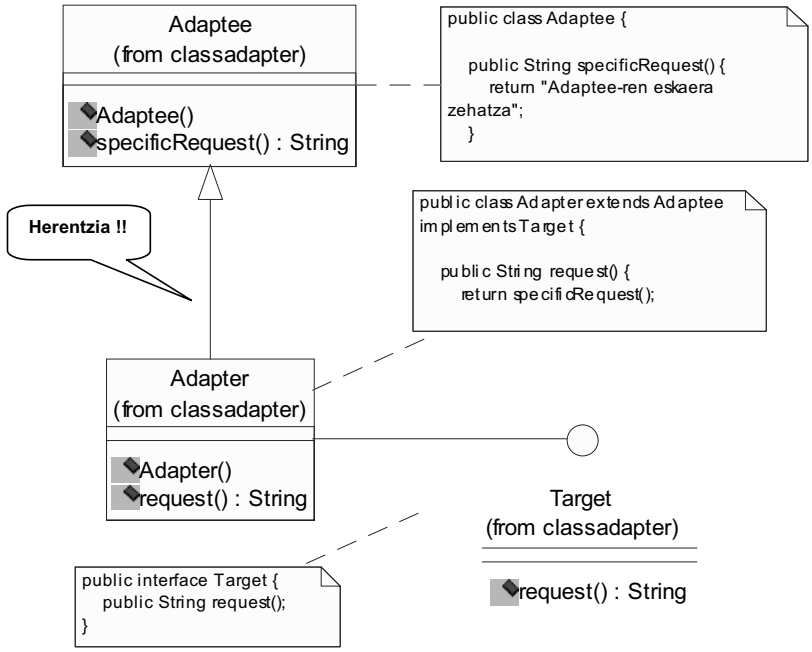
Patroi honekin bezeroari bi gauza adierazten zaizkio: bat, klaseak instantzia bakarra izango duela eta instantzia hori ondo kontrolatuta dagoela, eta bi, klasearen instantziara atzitu nahi badu, bide uniforme eta ezagunari jarraitu behar diola.

Klaseak bere aldetik, ostera, aplikazio informatikoak `Singleton` motako instantzia bakarra izango duela ziurtatzen dio bezeroari eta berak hartuko duela ardura hori hala izan dadin.

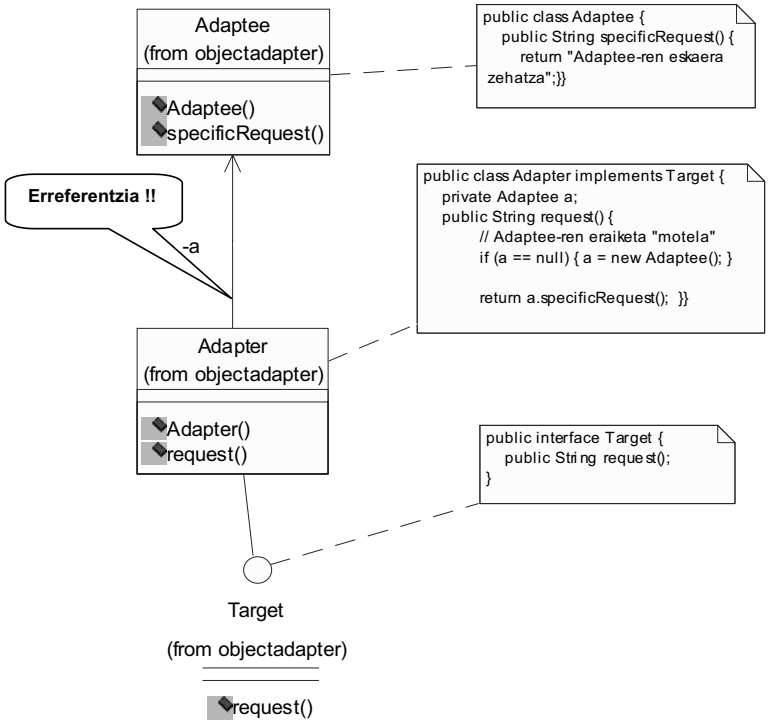
7.3. EGITURATZE-PROZESURAKO PATROIAK

Aplikazio informatikoen bizi-zikloa luzea eta gehienetan konplexua izaten da. Aplikazioa martxan jartzen da eta berehala ikusten da arlo batzuk hobetu egin behar direla. Hobetuta ere, teknologia berrien garapenak eta funtzionalitate berriek presioa sortzen dute aplikazioaren diseinua egoera berri horietara egokitzeko. Hala, klase-egitura berriak eraiki beharko dira diseinu zaharrean eta tentu handiz kokatu aurretik eraikitako ingurunean.

Atal honetan aztertuko ditugun diseinu-patroiek aipatutako lana leundu eta erraztu egiten dute. Batzuetan, diseinuaren zenbait arlo kapsulatu eta komunikatzen jartzen dituzte aurretik egindakoa diseinu berrian integratzeko. Beste batzuetan, ostera, egitura konplexuak eraiki behar dira eta tokian tokiko eta behin-behineko ebazpena bilatu beharrean, irtenbidea estandarizatu egiten da behin eta berriro aplikatzeko.



7.7. irudia. Adapter diseinu-patroia (klasea).



7.8. irudia. Adapter diseinu-patroia (objektua).

7.3.1. Adapter

Diseinu-patroi honetan klasearen interfazea bezeroak espero duen beste interfaze mota batera egokitzen da. Horrela, interfaze desberdinen abstrakzio uniforme bat lortzen da.

Patroi honek hiru parte-hartzaile ditu:

- Target: Parte-hartzaile honek interfaze bat erazagutzen du. Bezeroak domeinuaren interfazea identifikatzeko erabiliko duen bidea izango da. Interfaze honetan definitzen den eskaera (`request()`), klase egokitzaileak (`Adapter`) inplementatuko du bezeroaren eskaera egokituz.

- Adapter: Klase honek eskaera egokitzen duen klasea (`Adaptee`), eta hain zuzen, domeinuak erabiliko duena, eta `Target` interfazean definitu duguna lotzen ditu. Horrela, bezeroa ez da konturatzen, ezkutatu egiten baitzaio metodo mota birekin ari dela lanean eta batek bestea erabiltzen duela metodo-deia egokitzeko. Prozesu hori guztia, ez dago esan beharrik, gardena da bezeroarentzat.

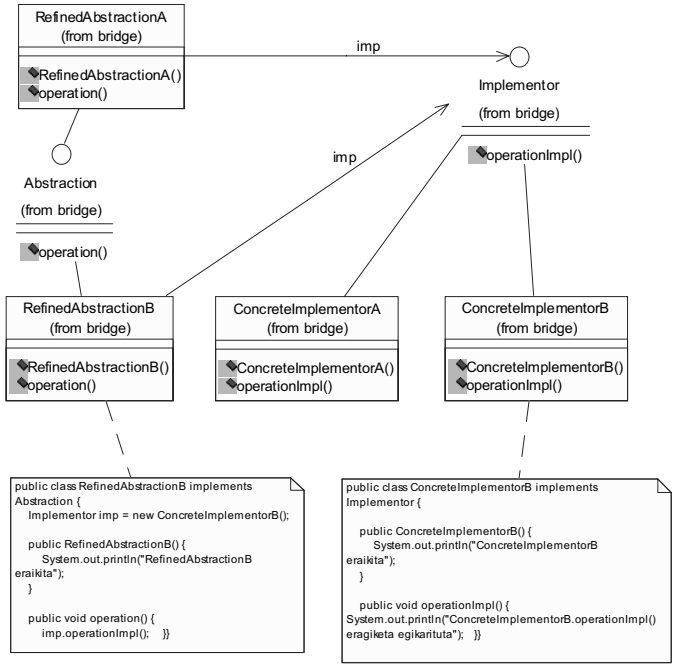
- Adaptee: Klase honek inplementatzen du benetan bezeroaren eskaera. Baina diseinu-arazoengatik, egokitu egin behar da bere notazioa, eta horretarako, `Adapter` klasearen eskaeran txertatzen da bezeroarentzat gardena izan dadin. Horrela, bezeroa ez da konturatzen eskaera egiten dagoenean, benetan, `Adaptee` klasearen eskaera bat egikaritzen ari dela.

Hedapenaren (**7.1. irudia**) ikuspuntutik, patroi hau da bakarra klase edo objektu gisa izango dena tratatua. Lehenengo egoeran, herentzia erabiltzen da egokitze-lana gauzatuko duen klasea (`Adapter`) kudeatzeko; bigarren egoeran, ostera, `Adaptee` motako aldagai bat erazagutzen da eta prozesuan zehar `Adaptee` motako objektu bat esleitzen zaio egokitze-lana gauzatu baino lehen. **7.7. irudia**k lehenengo egoera islatzen du eta **7.8. irudia**k, ostera, bigarrena.

Ondoko listatuak azaltzen du bezero batek nola erabiliko lukeen patroiaren osagai bakoitza.

```
    //  Klasea erabiltzen dugu
//  patroia egituratzeko herentzia bidez
public class Bezeroa {
    public static void main(String[] args) {
        // Target motako objektua eraiki eta eskaera burutu
        Target t = new Adapter();
            System.out.println(t.request());}}
    ____________________________
    //  Oraingo honetan, aldagai bat erazagutzen da
//     patroia egituratzeko
public class Bezeroa {
    public static void main(String[] args) {
        // Target motako objektua eraiki eta eskaera burutu
              Target a = new Adapter();
        System.out.println(a.request());}}
```

Bezeroaren programak adierazten digun bezala, lehenengo eta behin, `Adapter` motako objektu bat eraikitzen dugu eskaera egokituko duen objektua (`Adaptee`) eta bezeroa lotuko dituena. Hori egin eta gero, `Target` interfazeak informatuta, bezeroak domeinuaren eskaera egiten du. Eskaera hori `Adapter` objektuak egokitu eta beste eskaera batera bihurtzen du, `Adaptee` motako objektuak arazorik gabe egikaritu dezan eta emaitza itzuli. `Adapter` motako objektuak bezeroari itzultzen dio emaitza. Bezeroak `Adapter` motako objektuari ematen dizkio eskaerak, baina benetan `Adaptee` motakoari eman beharko lizkioke, hau da, bezeroarentzat erabat gardena da eskaera nork gauzatzen duen.



7.9. irudia. Bridge diseinu-patroia.

7.3.2. Bridge

Diseinu-patroi honetan abstrakzio bat bere inplementaziotik askatu egiten da. Horrela, bakoitza bere erara kudeatuko da, abstrakzioa alde batetik eta inplementazioa bestetik.

Patroi honek lau parte-hartzaile ditu:

- Abstraction: Parte-hartzaile honek interfaze abstraktu bat erazagutzen du. Interfaze hau inplementatzen duen klaseak erreferentzia bat izango du `Implementor` motako objektu batera. Horrela, abstrakzioa eta haren hierarkia alde batetik diseinatuko ditugu, eta inplementazioa eta haren hierarkia bestetik.

- RefinedAbstraction: Klase honek `Abstraction` interfazeak definitzen dituen eragiketak inplementatzen ditu. Horretarako, `Implementor` motako objektu bat eraikitzen du eta abstrakzioan definitu ditugun eragiketak klase honetan inplementatzen dira. Prozesua burutzeko, inplementazio zehatz bat erabiliko da `ConcreteImplementor` motako objektu bat eraikita eta objektu horren eragiketa egikarituta. Konturatu azken prozesu hau, Adapter patroian gertatzen zen bera dela, baina hemen bezeroa, abstrakzio bera izango da eta, objektu egokitzailea, ostera, inplementazio zehatza.

- Implementor: Parte-hartzaile honek interfaze bat erazagutzen du. Interfaze horrek inplementazio zehatzetan (`ConcreteImplementor`) inplementatuko diren eragiketak definitzen ditu. Konturatu, hemen definitzen diren eragiketek eta abstrakzioan agertzen direnek ez dutela maila berean egon behar. Diseinu egokietan, inplementazioaren aldeko eragiketak oinarrizkoagoak izango dira, abstrakzioan definitzen eta inplementatzen direnak baino.
- ConcreteImplementor: Klase honek `Implementor` interfazea inplementatzen du. Patroiaren barnean klase hau izango da benetan lan egiten arituko dena. Exekuzio-garaian, abstrakzioak mota honetako objektuaren eragiketak egikarituko ditu, bezeroak, abstrakzioaren eragiketak egikaritzen dituenean.

7.9. irudiak islatzen du nola egituratzen diren aipatutako lau parte-hartzaileen erlazioak. Ondoko listatuak azaltzen du bezero batek nola erabiliko lukeen patroiaren osagai bakoitza.

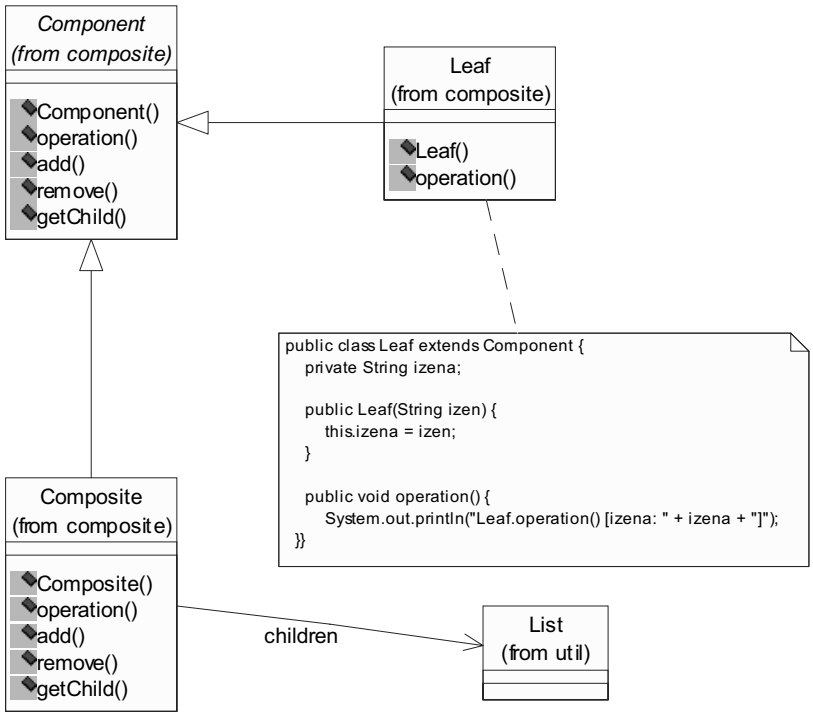
```
public class Bezeroa {
   public static void main(String[] args) {

        Abstraction a = new RefinedAbstractionA();
        a.operation();

              Abstraction b = new RefinedAbstractionB();
              b.operation();}}
```

Bezeroak `Abstraction` motako abstrakzio zehatzak (`RefinedAbstraction`) eraikitzen ditu lehendabizi. Abstrakzioak eraiki eta gero, abstrakzio zehatz bakoitzaren eragiketak egikaritzen dira. Une horretan, konexio ezkutua sortzen da abstrakzioaren eta inplementazioaren artean, zeren abstrakzioak egikaritzen baitu inplementazioak eskaintzen dion eragiketa.

Konturatu, patroiak bezeroarekin sortzen dituen harremanak beti abstrakzio mailan gauzatzen direla eta abstrakzioak erabat banatzen dituela inplementazioak eta inplementazio horiek erabiltzeko interfazeak. Eragiketen inplementazioen nondik norakoak ezkutatuta gelditzen dira bezeroarentzat. Patroi honetan abstrakzioen eta inplementazioen hierarkiak, independenteak direnez, erraz eta era malguan heda daitezke.



7.10. irudia. Composite diseinu-patroia.

7.3.3. Composite

Diseinu-patroi honetan objektuak zuhaitz-egituran antolatzen dira zati-oso (part-whole) erlazio hierarkikoak islatzeko.

Patroi honek hiru parte-hartzaile ditu:

- Component: Klase abstraktu honetan dago patroiaren muina. Bezeroak mota honetako objektu bat erabiliko du zuhaitz-egitura aztertzeko eta aldi berean, zuhaitzaren osagai guztiek objektu mota hau hedatuko dute beren ezaugarriak adierazteko. Osagaiak bi motatakoak izan daitezke: zuhaitzaren hostoak edo zuhaitzaren nodoak. Patroi kanonikoak, zuhaitzaren hostoak eraikitzeko, `Leaf` motako objektuak eraikitzen ditu eta zuhaitzaren nodoak eraikitzeko, ostera, `Composite` motakoak. Bigarren motako osagai horiek tratatzeko, hainbat metodo (`add()`, `remove()` eta `getChild()`) definitzen eta inplementatzen ditu `Component` klase abstraktuak. Metodoak gero birdefinitu egingo dira `Composite` motako objektuetan.

- Composite: Klase honek `Component` motako objektu bat hedatu egiten du. Zuhaitzaren osagaiak semeak dituenean, hau da, zuhaitzaren elementua nodo bat denean, `Composite` motako objektu bat eraiki behar da. Klase

horrek bilduma motako objektu[65] bat eraiki behar du bere elementuak gordetzeko. Bilduma motako objektua eraikiz gero, `add()`, `remove()` eta `getChild()` metodoak birdefinitu egiten dira, inplementazio berri bat itsatsita bakoitzari. Ondoko listatuak adierazten du nola gauzatzen den aipatu duguna iturburu-kode mailan.

```
public class Composite extends Component {
    List children = new ArrayList();
    public void operation() {
        Iterator iter = children.iterator();
        while (iter.hasNext()) {
            ((Component) iter.next()).operation();   }}

    public void add(Component c) {
        try {
            children.add(c);
        } catch (Exception e) {
            System.out.println(e.getMessage());  }}

 public void remove(Component c) {
        try {
            children.remove(c);
        } catch (Exception e) {
            System.out.println(e.getMessage());  }}

    public Component getChild(int index) {
            Component child;
            try {
                child = (Component) children.get(index);
            } catch (IndexOutOfBoundsException e) {
                return null;}
         return child; }}
```

- Leaf: Klase honekin zuhaitzaren elementu atomikoak adierazten dira. Klase honek zuhaitzaren hostoak izan behar duen portaera jasotzen du.

7.10. irudiak islatzen du nola egituratzen diren aipatutako hiru parte-hartzaileen erlazioak. Hurrengo listatuak azaltzen du bezero batek nola erabiliko lukeen patroiaren osagai bakoitza.

65. Guk aztertuko dugun adibidean, `ArrayList` motako objektu bat eraikiko dugu `Composite` klasearen osagaiak gordetzeko.

```
public class Bezeroa {
/*
                          componentsCollection
                                  |
                           -----------
                          |           |
                    components     component3
                          |
                 ---------------------
                |                     |
        component1     component2
*/
    public static void main(String[] args) {
        // Zuhaitzaren orriak eraiki
        Component component1 = new Leaf("orri bat");
        Component component2 = new Leaf("orri bi");
        Component component3 = new Leaf("orri hiru");

        // Composite motako objektuak(nodoak) eraiki
        Component components = new Composite();
        Component componentsCollection = new Composite();

        // Orriak gehitu nodoari
        components.add(component1);
        components.add(component2);

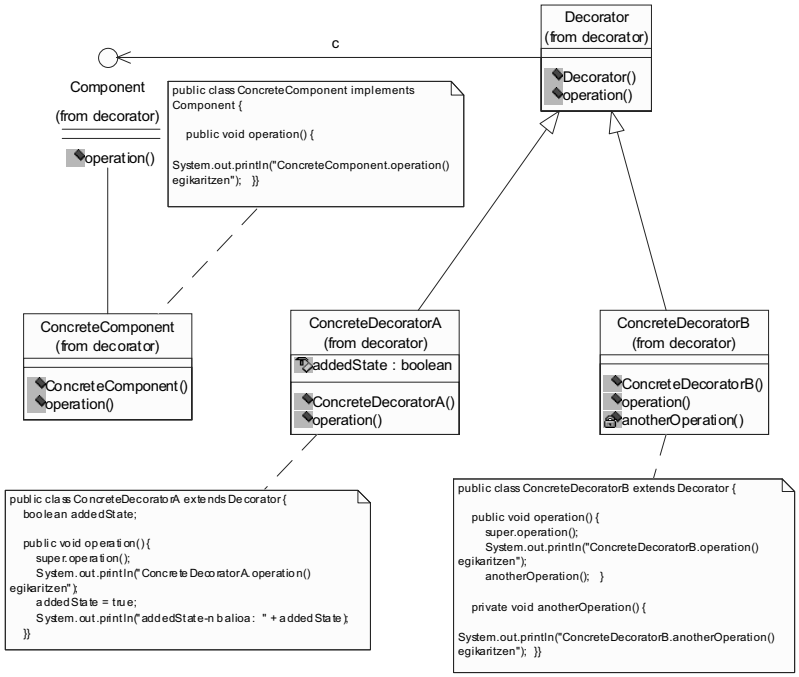
        // Nodoari beste nodo bat gehitu
        componentsCollection.add(components);

        // Orriak gehitu nodoari
                componentsCollection.add(component3);

    //   Egikaritu   zuhaitzaren   operation()   metodoak
componentsCollection.operation();   }}
```

Lehenengo eta behin, bezeroak zuhaitz-egitura eraikitzen du. Horretarako, oinarrizko (`Leaf`) hiru osagai eraikitzen ditu eta bi osagai konplexu (`Composite`). Bezeroak osagai konplexu (`componentsCollections`) horietariko bat zuhaitz-egiturara sartzeko erabiliko du. Sartuz gero, inplementatuta duen eragiketa (`operation()`) egikaritzeko eskatuko dio osagai bakoitzari eta gauzak ondo badoaz, zuhaitzaren osagai bakoitzak bere eragiketa egikarituko du.

Bigarren osagai konplexua (`components`) elementu atomikoz (`Leaf`) beteko dugu eta goian aipatutako osagai konplexuari (`componentsCollections`) gehituko diogu. Zuhaitza eraikiz gero, bezeroak osagai konplexu nagusiaren eragiketa egikaritzen du, egitura osoaren partaide bakoitzari ohartarazteko bere eragiketa partikularra egikaritu egin behar duela.



7.11. irudia. Decorator diseinu-patroia.

7.3.4. Decorator

Diseinu-patroi honekin objektu bati dinamikoki itsasten zaizkio ardurak. Patroi honekin klase-azpiklase erlazioa ordezkatu egiten da, funtzionalitatea hedatzeko.

Patroi honek lau parte-hartzaile ditu:

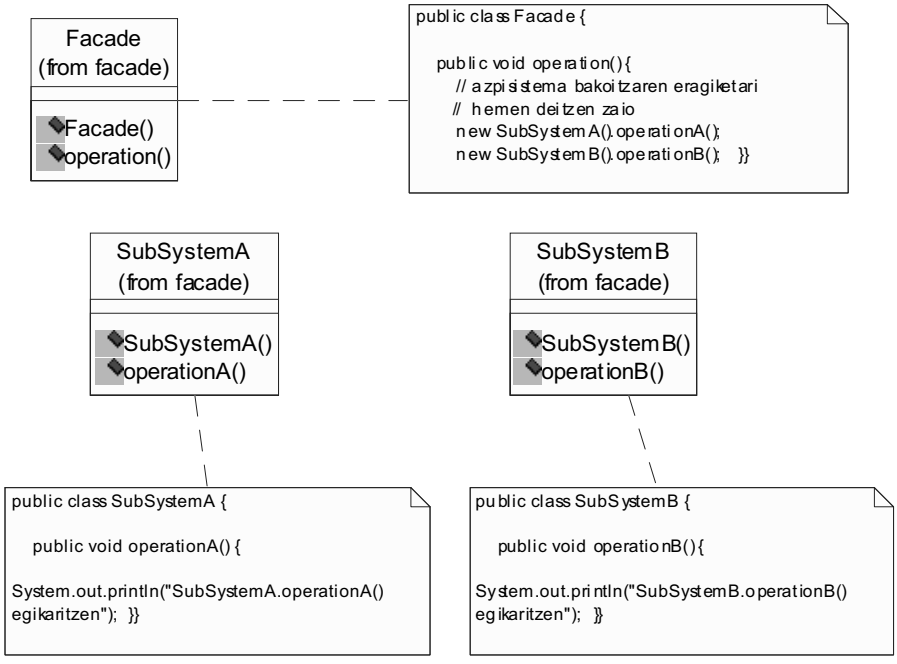
- Decorator: Klase honek `Component` interfazea inplementatzen du eta halaber, osagai zehatza (`ConcreteComponent`) eraiki eta erabili egiten du inplementatzen duen eragiketan (`operation()`). Hala, `Decorator` motako objektu bat erabiltzen duen edozein objektuk badaki zein den *dekoratzen* ari den osagai zehatza.
- Component: Parte-hartzaile honek interfaze bat erazagutzen du. Hortxe definitzen da osagai zehatzaren interfazea. `Decorator` motako objektuek, dinamikoki, funtzionalitate berriak itsasten dituzte era malguan, `Decorator` motako objektu bat erabilita, bitartekari-lanak gauzatzeko.
- ConcreteComponent: Klase hau izango da patroiaren osagai zehatza eta honi gehituko zaizkio ardura edo funtzionalitate berriak. Klase honek `Component` interfaze orokorra inplementatzen du eragiketa nagusia (`operation()`) inplementatuta.

- ConcreteDecorator: Klase hauek `Decorator` klasearen funtzionalitatea hedatzen dute eragiketa (`operation()`) birdefinituta. Hala, osagai zehatzaren funtzionalitatea hedatu egiten da ezaugarri berriak itsatsita exekuzio-garaian.

7.11. irudiak islatzen du nola egituratzen diren aipatutako lau parte-hartzaileen erlazioak. Ondorengo listatuak azaltzen du bezero batek nola erabiliko lukeen patroiaren osagai bakoitza.

```
public class Bezeroa {
public static void main(String[] args) {
    // Eraiki objektuak apaintze lanak burutzeko
    Component dA = new ConcreteDecoratorA();
    Component dB = new ConcreteDecoratorB();
    // Eragiketak egikaritu
          dA.operation();
          dB.operation();   }}
```

Bezeroak `Decorator` motako objektu desberdin bi eraikitzen ditu lehendabizi. Bai batak eta bai besteak, osagai zehatzaren (`ConcreteComponent`) eragiketak ezagutzen dituzte eta bakoitzak eragiketa horiek apaindu ditu bere erara funtzionalitate berria erantsita. Batek, true balioa esleituko dio aldagai boolear bati, eta besteak, ostera, eragiketa (`anotherOperation()`) berri bat egikarituko du metodoei (`operation()`) dei egiten zaienean.



7.12. irudia. Façade diseinu-patroia.

Patroi honekin ardurak banatu egiten dira eta dinamikoki esleitzen zaizkie objektu zehatzei, objektu nagusiari, enbarazurik egin gabe. Patroi hau sarri erabiltzeak arazoak sor ditzake klaseen diseinuan, hainbat objektu txiki sortzen baitira apainketa-lana bakarrik gauzatuko dutenak. Konturatu, halaber, patroi hau zenbait egoeratan herentzia estatikoa ordezkatzeko ere balio dezakeela ebazpen malguagoa eskainiz.

7.3.5. Façade

Diseinu-patroi honekin, interfaze bateratu bat eraikitzen da azpisistema batean, interfaze multzo baten lana gauzatzeko. Goi-mailako interfaze bat eraikitzen da azpisistemaren erabilera errazteko.

Patroi honek bi parte-hartzaile ditu:

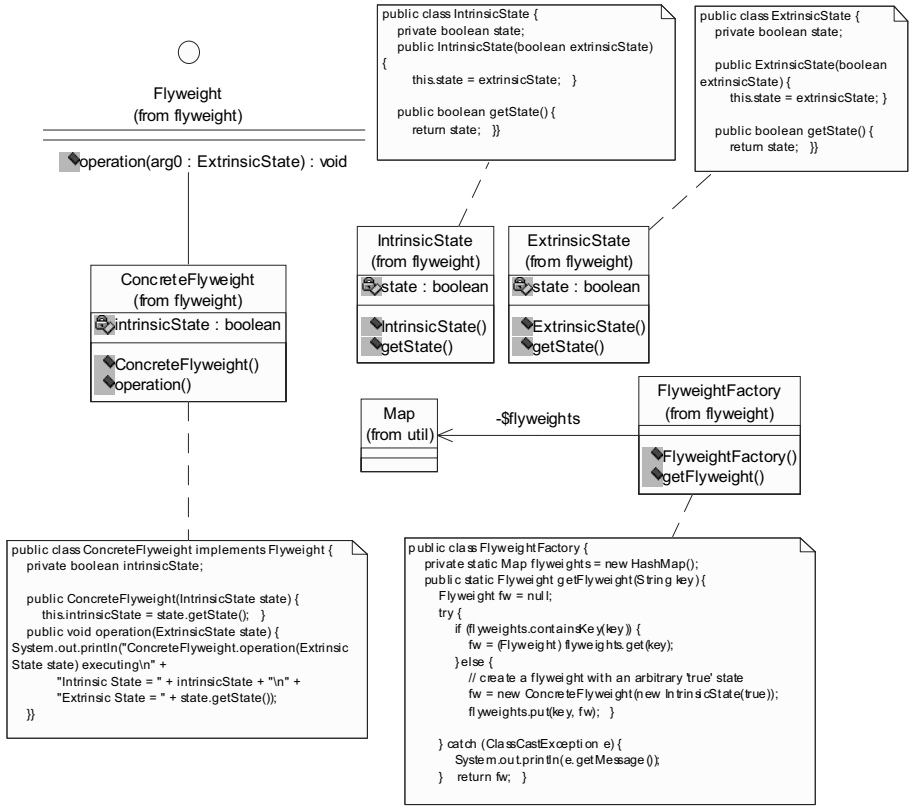
- Façade: Klase honek bitartekari-lana egiten du bezeroen eta azpisistemen artean. Klase hau erdian kokatuta, bezeroak nahikoa du mota honetako objektuari eskaera bat eta bakarra egitea. Façade-n kokatuko dugun eskaerak berak hartuko du ardura, azpisistema desberdinetara egin behar diren eskaera birfinduagoak kudeatzeko. Horrela, bezeroak ez du arduratu behar eskaeren nondik norakoak ezagutzeaz eta kudeatzeaz.
- SubSystem: Klase hauek azpisistema bakoitzaren funtzionalitatea inplementatzen dute. Klase horietako eragiketak kudeatuz, `Façade` klaseak erantzun egokia eraikiko dio bezeroari, horrek eskaera bat bidaltzen dionean.

7.12. irudiak islatzen du nola egituratzen diren aipatutako bi parte-hartzaileen erlazioak. Ondoko listatuak azaltzen du bezero batek nola erabiliko lukeen patroiaren osagai bakoitza.

```
public class Bezeroa {
   public static void main(String[] args) {

          // Façade motako objektu bat eraiki eta erabili
       new Façade().operation();   }}
```

Bezeroak nahikoa du `Façade` motako objektu bat eraikitzea eta objektu horren eragiketa (`operation()`) egikaritzea, azpisistema desberdinen eragiketak (`operationX()`) egikaritzeko. Bezeroak ez du arduratu behar inondik ere, azpisistema desberdinen eragiketak nola antolatu eta egikaritzen diren; `Façade` motako objektuak beteko du lan aspergarri hori.



7.13. irudia. Flyweight diseinu-patroia.

7.3.6. Flyweight

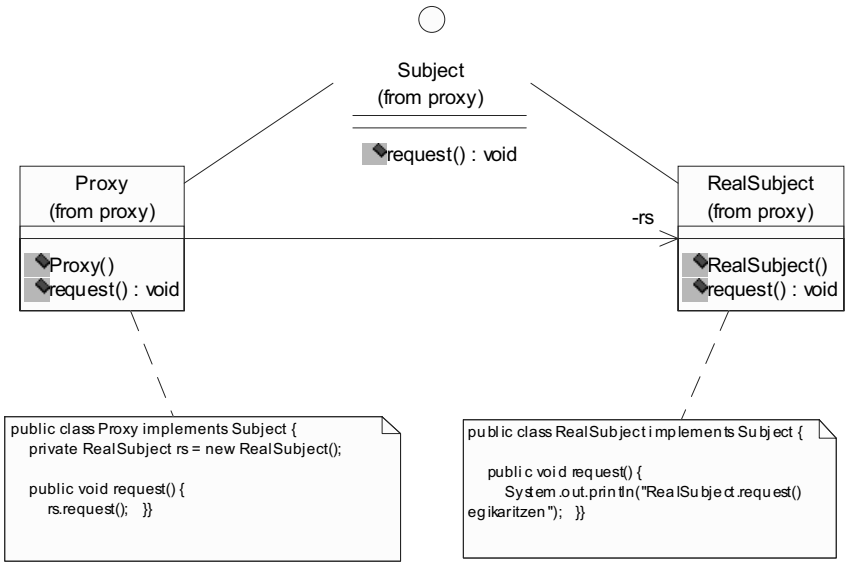
Objektu multzo batek portaera antzekoa duenean, antzekotasun hori identifikatu eta patroi bihurtzen da objektu guztiak zati horretaz balia daitezen.

Patroi honek bost parte-hartzaile ditu:

- Flyweight: Parte-hartzaile honek interfaze bat erazagutzen du. Interfaze horretan, gero `ConcreteFlyweight` motako objektu batek inplementatuko duen eragiketa definitzen du patroiak. Bezeroak bidalita, eragiketak kanpoko egoeraren (`ExtrinsicState`) informazioa jasoko du parametroan eta konbinatu egingo du bere barneko egoerarekin (`IntrinsicState`).
- ConcreteFlyweight: Klase hauek `Flyweight` interfazea inplementatzen dute eta aldi berean, barne-egoera (`IntrinsicState`) gordetzeko ere erabiliko dira. Mota honetako objektuek partekatu egin behar dute informazioa eta nahitaez, barnekoa izan behar du informazio horrek. Gero, testuinguruak osatu egingo du informazio hori kanpoko egoerak (`ExtrinsicState`) informazio horri erantsita.

- FlyweightFactory: Klase honek `Flyweight` motako objektuak eraiki eta kudeatzen ditu. `Flyweight` motako objektuak kudeatzeko `getFlyweight()` metodoa erabiltzen da. Metodo horri, `Flyweight` motako objektuaren gakoa pasatzen zaio parametro gisa eta metodoak, objektua bere biltegian aurkitzen badu, itzuli egiten du, bestela, `Flyweight` motako objektu bat eraikitzen du lehendabizi eta gero, biltegiratu egiten du, pasa diogun gakoaren balioa gako nagusi gisa erabilita. Bukatzeko, lehenengo egoeran bezala, `Flyweight` motako objektu eraiki berria itzultzen du.
- IntrinsicState: Klase honek patroiaren barne-egoerak eraiki eta eskuratzeko (`getState()`) metodoak izango ditu inplementatuta. Flyweight zehatzek (`ConcreteFlyweight`) mota honetako objektuak erabiliko dituzte eraikitzen direnean, barne-egoerak flyweight zehatzei itsasteko.
- ExtrinsicState: Klase honetan patroiaren kanpoko egoerak eraiki eta eskuratzeko (`getState()`) metodoak inplementatzen dira.

Patroi hau baliagarria da baldin eta eraikitzen ari garen Klase Diagrama hainbat klase taldetan antolatzen bada eta, taldetan antolatuz gero, talde bakoitzean objektu guztiek egoera (`IntrinsicState`) jakin bat izaten badute beti. Taldeko objektu bakoitzak, gero, egoera jakin hori aberastu egiten du kanpoko egoera (`ExtrinsicState`) erantsita, zeren eraikitzen den uneko testuingurua kontuan hartzen baita kanpoko egoera hori eraikitzeko.



7.14. irudia. Proxy diseinu-patroia.

7.3.7. Proxy

Patroi honek zeharkako bide bat eraikitzen du objektu batera, gauzatuko den atzipena kontrolatzeko.

Patroi honek hiru parte-hartzaile ditu:

- Proxy: Klase honek zeharkako bidea egituratzen du objektu (`RealSubject`) batera egingo diren atzipenak kontrolatzeko. `Proxy` motako objektu bat eraikitzen denean berehala sortzen da atzituko den objektura erreferentzia bat. Proxy-ak eta atzipena kontrolatu nahi duen objektuak interfaze bera izango dute eta `Subject` interfazean egongo da definitua. Hala, proxy-a erabilita objektua ordezkatu nahi dugunean, ez dugu inongo arazorik izango. Proxy-ak berak, erreferentzia ez ezik, ardura ere har dezake atzitu nahi den objektua eraiki eta ezabatzeko.
- Subject: Osagai honek interfaze bateratua definitzen du proxy-arentzat eta atzipena kontrolatu nahi dugun objektuarentzat. Horrela, objektuaren aiduru dauden tokietan haren proxy-a erabil dezakegu arazorik gabe.
- RealSubject: Klase honek `Subject`-aren interfazeak inplementatzen ditu eta atzipena kontrolatu nahi dugun objektuaren rola beteko du. Proxy-ak erreferentziatuta izango du objektu hau, eta bezeroak, objektu honek duen eragiketa egikaritu ahal izango du zeharka.

7.14. irudiak islatzen du nola egituratzen diren aipatutako hiru parte-hartzaileen erlazioak. Ondoko listatuak azaltzen du bezero batek nola erabiliko lukeen patroiaren osagai bakoitza.

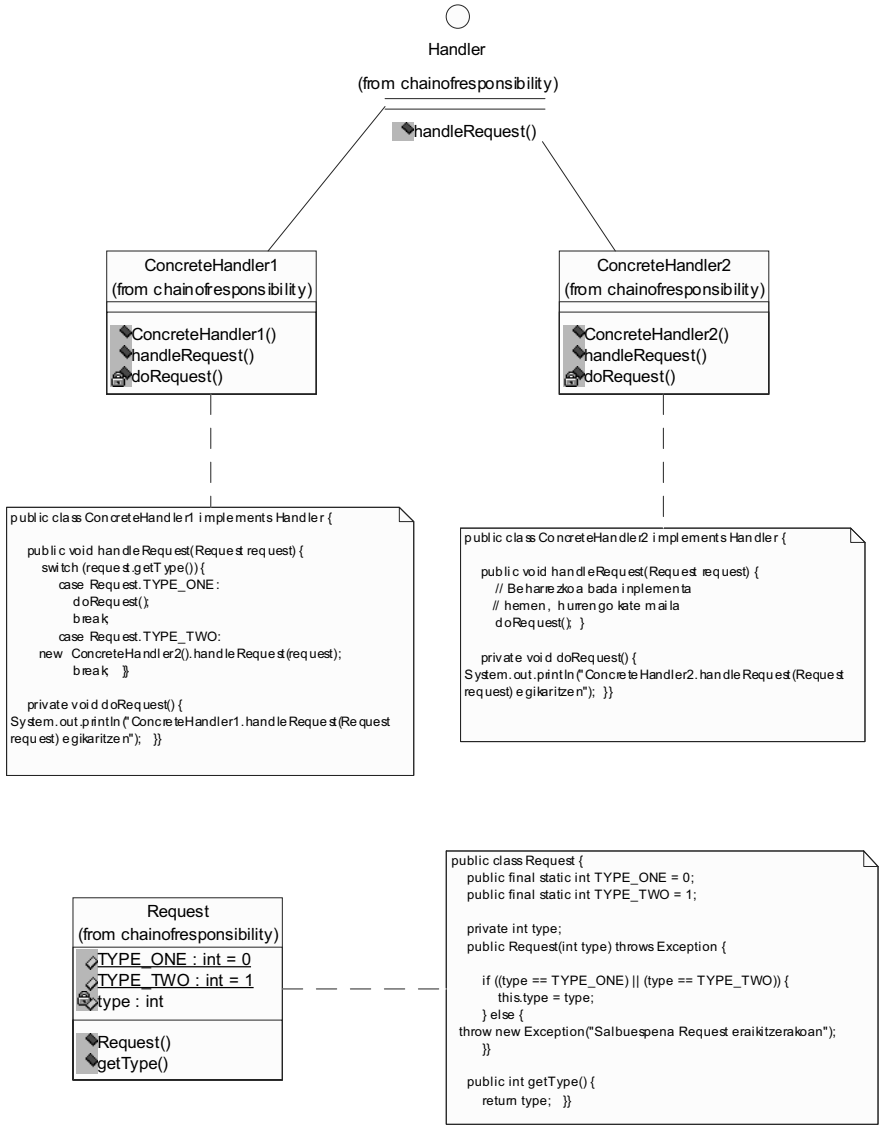
```
public class Bezeroa {
         public static void main(String[] args) {
       // Subject motako objektua eraiki
 //  proxy-a egituratzeko
       Subject s = new Proxy();

             // Eskaera burutu proxy-a erabilita
       s.request();   }}
```

Bezeroak `Subject` motako objektu bat eraikitzen du `Proxy` klasearen metodo eraikitzailea egikarituta. Dakigun bezala, proxy objektuak erreferentzia pribatu bat sortzen du atzipena kontrolatu nahi dugun objektura. Horrela, bezeroak eskaera (`request()`) gauzatzen duenean, eskaera hori proxy objektutik igaroko da eta atzipena kontrolatuta dugun objektura helduko da, hantxe egikaritzeko.

7.4. PORTAERA KUDEATZEKO PATROIAK

Aurreko atalean aztertu ditugun diseinu-patroiek klaseak eta objektuak egituratzen dituzte diseinu malgu eta eraginkorrak lortzeko. Oraingo atal honetan, objektuen arteko komunikazioak bereganatuko du arreta, hau da, objektuaren portaeran gertatzen diren aldaketak kontrolatuko dira eta ondorioak diseinuan zehar barreiatuko dira, hala nahi izanez gero.



7.15. irudia. Chain of Responsibility diseinu-patroia.

Atal honetako diseinu-patroiek, normalean, klase abstraktu bat definitzen dute kapsulatu nahi dugun objektua deskribatzeko. Kapsulatzen dugunaren izaera era askotakoa izan daiteke: adibidez, Strategy patroiak algoritmoak kapsulatzen ditu; Mediator patroiak objektuen arteko komunikazio-protokoloa, eta Iterator patroiak, ostera, objektu agregatu baten osagaietara atzitzeko bidea. Objektuaren egoera kapsulatu nahi bada State patroiak txukun-txukun beteko du lan hori. Ostera, objektuaren egoera-aldaketak beste objektu batzuetara transmititu nahi badira, Observer patroiak ez du inongo arazorik izango betebehar hori gauzatzeko.

7.4.1.Chain of Responsibility

Diseinu-patroi honetan eskaera baten igorlea ez da zuzen-zuzenean jartzen harremanetan eskaeraren jasotzailearekin. Patroiak aukera ematen die beste objektu batzuei eskaera prestatzeko. Eskaera kateatutako objektu multzo batera bidaltzen da eta objektuz objektu bidaiatzen hasten da, objektu batek eskaera tratatzeko ardura hartu arte.

Patroi honek hiru parte-hartzaile ditu:

- Handler: Parte-hartzaile honek interfaze bat erazagutzen du. Eskaerak kudeatzeko interfazea hemen definitzen da. Batzuetan, ondorengo esteka ere hemen inplementatzen da.
- ConcreteHandler: Klase hauek `Handler` interfazea inplementatzen dute. `ConcreteHandler` motako objektu bakoitzak bi eragiketa nagusi egiten ditu eskaera (`Request`) bat jasotzen duenean: lehendabizi, eskaeraren mota aztertzen da eta objektua konturatzen bada berak tratatu behar duela eskaera, dagokion erantzuna (`doRequest()`) egikaritzen du eta bukatutzat ematen du trataera orokorra. Objektua konturatzen bada bera ez dela eskaera tratatu behar duena, beste eskaera-kudeatzaile (`ConcreteHandler`) bati pasatzen dio eskaera tratatzeko ardura. Bigarren objektu horrek gauza bera egingo du, hau da, eskaera tratatu edo beste hirugarren objektu bati pasatu eskaera tratatzeko ardura. Pentsatzekoa da, eskaera ondo egin bada, katearen kate-mailaren batek beti tratatuko duela eskaera.
- Request: Klase hau erabilita, eskaera motak eraiki eta kudeatzen dira. Klase honetan antolatzen dira onartuko diren eskaera motak eta erabiliko diren metodo atzitzaileak.

7.15. irudiak islatzen du nola egituratzen diren aipatutako hiru parte-hartzaileen erlazioak. Ondoko listatuak azaltzen du bezero batek nola erabiliko lukeen patroiaren osagai bakoitza.

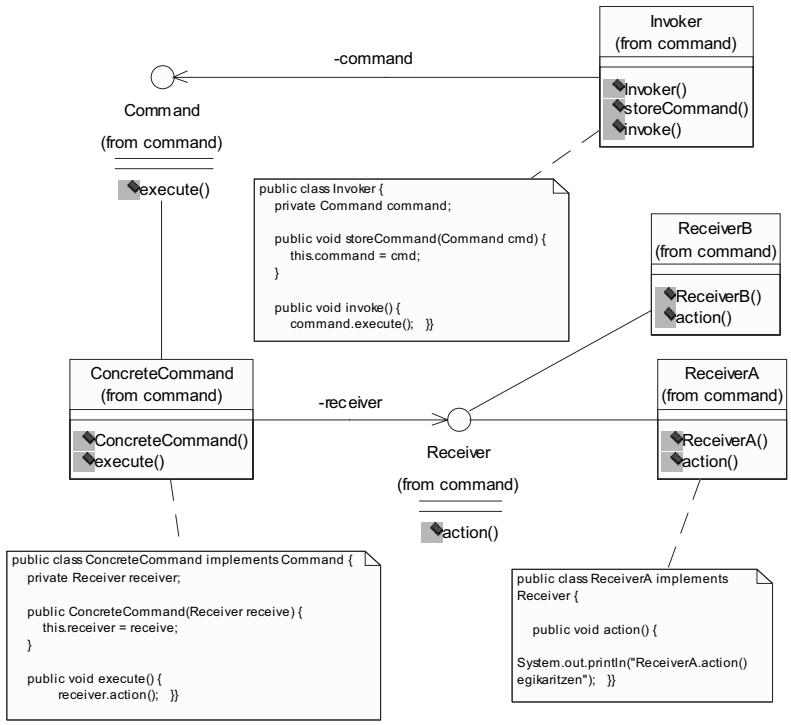
```
public class Bezeroa {
public static void main(String[] args) {

    try {
        // Eskaerak eraiki
        Request r1 = new Request(Request.TYPE_ONE);
        Request r2 = new Request(Request.TYPE_TWO);

        // Eskaera tratatuko duen objektua eraiki
        Handler h = new ConcreteHandler1();

        // Eskaerak kudeatu
        h.handleRequest(r1);
                h.handleRequest(r2);

    } catch (Exception e) { // Salbuespenen trataera
    System.out.println(e.getMessage());   }}}
```



7.16. irudia. Command diseinu-patroia.

Bezeroak, patroia erabiltzen hasteko, lehendabizi, eskaera mota desberdinak eraiki behar ditu. Behin hori eginda, `Handler` motako `ConcreteHandler` objektu bat eraikiko du, hau da, eskaeraren kudeatzaile lehenetsia. Objektu hori izango da patroiak eskaera tratatzeko erabiliko duen lehen objektua. Eskaeraren

mota aztertuta konturatzen bada berak ez duela mota hori tratatu behar, beste objektu-kudeatzaile bati egiten dio dei. Gure kasuan, bezeroak egiten duen lehenengo eskaera (`handleRequest(r1)`) lehentsia dagoen objektuak kudeatuko du eta bigarrena (`handleRequest(r2)`), ostera, bigarren objektuak. Konturatu objektu-kudeatzailea eskaeraren motaren arabera aukeratzen dela. Horregatik, beti izango da interesgarriagoa gehien erabiltzen diren eskaera motak katearen hasierako kate-mailetan tratatzea, horrela, patroiaren eraginkortasuna asko hobetuko da.

7.4.2. Command

Patroi honetan bezeroaren eskaera kapsulatu egiten da objektu gisa eta objektu horri interfaze publiko eta ezagun bat itsasten zaio. Hala, bezeroak eskaera gauzatzen duenean, egikaritu behar den ekintza banatuta gelditzen da eta diseinatzaileak ez du arazo handirik izango erantzunaren ekintza aldatu edo birprogramatu nahi badu, bezeroaren programa ez baitago zuzen-zuzenean lotua erantzunaren ekintzara.

Patroi honek bost parte-hartzaile ditu:

- Command: Parte-hartzaile honek interfaze bat erazagutzen du. Interfaze horretan definitzen da egikaritu behar den eragiketa (`execute()`).
- ConcreteCommand: Klase honek `Command` interfazea inplementatzen du. Interfazean definitu dugun eragiketa (`execute()`) hemen inplementatzen da eta horretarako `Receiver` motako objektu bat erabiliko da bitartekarilana gauzatzeko. `Receiver` motako objektu baten `action()` metodoa izango da eragiketa benetan egikarituko duena.
- Receiver: Parte-hartzaile honek interfaze bat erazagutzen du. Interfaze honetan definitzen da `execute()` metodoak erabiliko duen `action()` metodoa. Eskaera bat aktibatzen denean `execute()` metodoa egikaritzeko eskatuz gero, interfaze hau inplementatzen duen objektuak inplementazio egokia erantsiko dio `action()` metodoari.
- ReceiverX: Klase hauek `Receiver` interfazea inplementatzen dute bakoitzak bere inplementazio propioa itsatsita `action()` metodoari.
- Invoker: Klase honek `Command` motako objektu bati eskatzen dio bere `execute()` metodoa egikaritzeko. Horretarako, bi metodo antolatzen dira: batekin, `Command` motako objektu bat biltegiratzen da objektuan (`storeCommand()`) eta bestearekin, `Command` motako objektuaren `execute()` metodoa egikaritzen (`invoke()`) da.

7.16. irudiak islatzen du nola egituratzen diren aipatutako bost parte-hartzaileen erlazioak. Ondoko listatuak azaltzen du bezero batek nola erabiliko lukeen patroiaren osagai bakoitza.

```
public class Bezeroa {

   public static void main(String[] args) {
       // Receiver motako objektu bi eraiki
       Receiver rcv1 = new ReceiverA();
       Receiver rcv2 = new ReceiverB();

       // Command motako objektuak eraiki
       // Receiver motako objektuak erantsiz
       Command cmd1 = new ConcreteCommand(rcv1);
       Command cmd2 = new ConcreteCommand(rcv2);

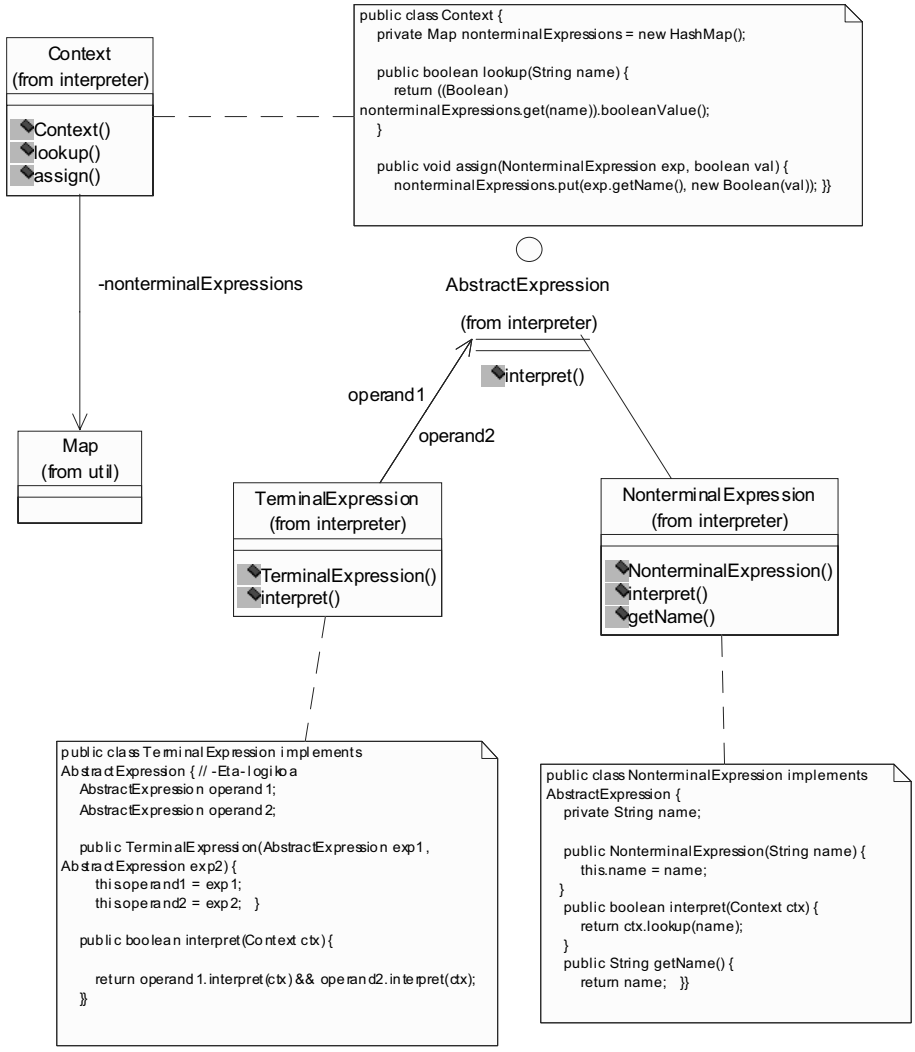
       // Invoker-ak eraiki
       Invoker inv1 = new Invoker();
       Invoker inv2 = new Invoker();

       // Biltegiratu aginduak Invoker motako objektuan
       inv1.storeCommand(cmd1);
       inv2.storeCommand(cmd2);

       // Deitu Invoker motako objektuei
//aginduak egiaztatzeko
           inv1.invoke();
       inv2.invoke();    }}
```

Bezeroak bost urrats nagusi bete behar ditu patroi hau erabili nahi badu:

- Lehenengo eta behin, `Receiver` motako objektuak eraiki behar ditu, `Command` motako objektu batek mota horretako objektua eskatzen baitu metodo eraikitzailean.
- Bigarren urratsean `Receiver` motako objektu eraiki berriak `Command` motako objektuak eraikitzeko erabiliko dira. Hala, `Command` motako objektu bati `execute()` metodoa egikaritzeko eskatzen zaionean, ez du inongo arazorik izango, aurretik `Receiver` motako objektu bat erabili izan baitu objektua eraikitzeko.
- Hirugarren eta laugarren urratsak kateatuta daude. Hirugarrenean, `Invoker` motako objektuak eraikitzen dira eta laugarrenean, `Invoker` motako objektu bakoitzean bigarren urratsean eraiki ditugun `Command` motako objektuak biltegiratu (`storeCommand()`) egiten dira.
- Prestatu dugun azpiegitura guztiak ondo funtzionatzen duela egiaztatu nahi badugu, nahikoa zaigu azken urrats honetan `Invoker` motako objektu bakoitzaren `invoke()` metodoa egikaritzea. Aplikazioak ondo funtzionatzen badu, aldiko, `Receiver` motako objektuaren `action()` metodoa egikaritu behar da.



7.17. irudia. Interpreter diseinu-patroia.

7.4.3. Interpreter

Aplikazio informatiko batzuetan komenigarria izaten da gauzatu behar diren eragiketak lengoaia abstraktu bat erabilita egituratzea. Lengoaia horrek, lengoaia naturalak legez, gramatika bat izango du *esaldi* egokiak eta ezegokiak bereizteko.

Patroi hau arduratuko da, hain zuzen, esaldi egoki horiek *interpretatzeaz* eta beraien esanahia azalarazteaz.

Patroi honek lau parte-hartzaile ditu:

- AbstractExpression: Parte-hartzaile honek interfaze bat erazagutzen du. Interfaze horretan eragiketa (`interpret()`) abstraktu bat definitzen da. Eragiketaren izena eta signatura bera izango da, lengoaia abstraktuaren adierazpen oinarrizkoak eta konplexuak tratatzeko.
- TerminalExpression: Klase honek `AbstractExpression` interfazea inplementatzen du `interpret()` metodoa inplementatuta. Gramatikaren bukaerako ikur bakoitza mota honetakoa izango da eta instantzia bat izan beharko du eraikita.
- NonterminalExpression: Gramatikaren erregela bakoitza klase hau erabilita eraikiko da. Halaber, erregelaren ikur bakoitzaren `AbstractExpression` motako instantzien kudeaketa, klase honen ardura izango da. Bukatzeko, `interpret()` metodoa inplementatzen du gramatikaren bukaerako ikurrak ez diren ikurrentzat.
- Context: Interpreteak erabiltzen duen informazio orokorra klase honetan biltegiratzen da. Geroago ikusiko dugun adibidean, `assign()` metodoa erabiliko dugu bukaerako ikurra ez den ikur bati balio boolear bat esleitzeko. Gero, lookup() metodoa erabilita balio boolear hori eskuratu ahal izango dugu.

7.17. irudiak islatzen du nola egituratzen diren aipatutako bost parte-hartzaileen erlazioak. Hurrengo listatuak azaltzen du bezero batek nola erabiliko lukeen patroiaren osagai bakoitza.

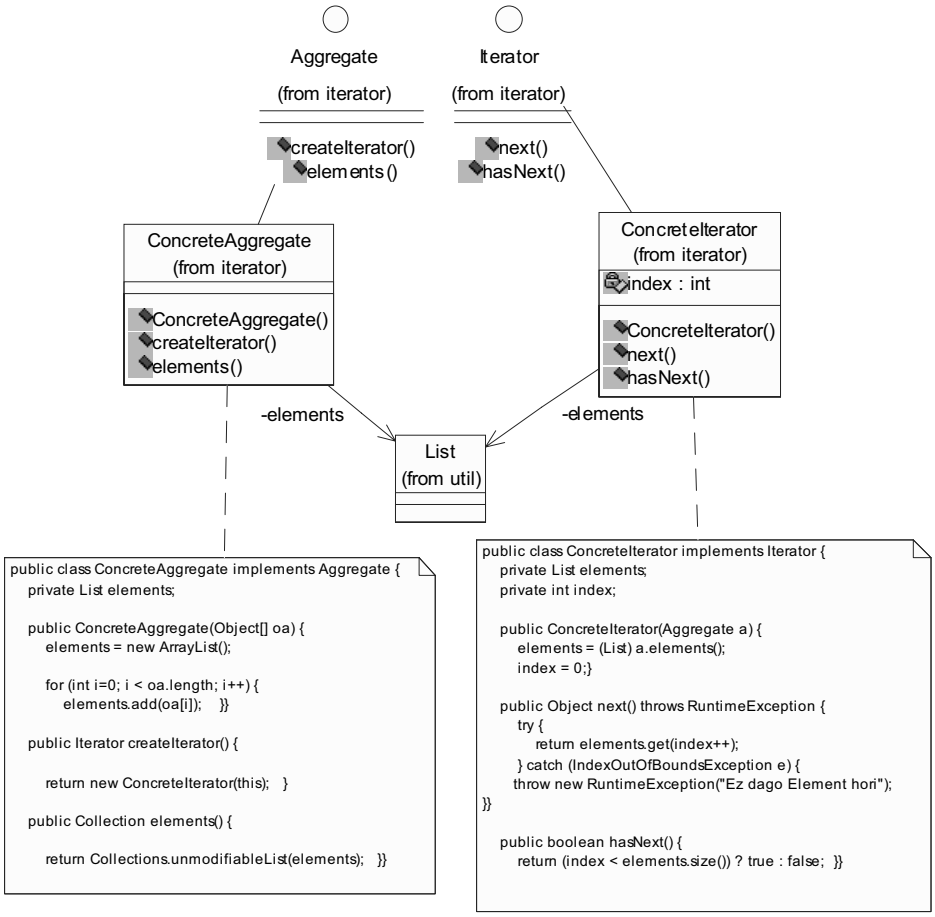
```
   public class Bezeroa {
      public static void main(String[] args) {
            // Eraiki eragileak
            AbstractExpression x = new
NonterminalExpression("X");
            AbstractExpression y = new
NonterminalExpression("Y");

      AbstractExpression expression = new TerminalExpression(x,
y);

            Context context = new Context();
            context.assign((NonterminalExpression) x, true);
            context.assign((NonterminalExpression) y, true);

                boolean result = expression.interpret(context);
       System.out.println("AbstractExpression.interpret() value [" +
result + "]");   }}
```

Adibidean, bezeroak *eta* logikoa egikaritu nahi du, horretarako `AbstracExpression` motako hiru objektu eraikitzen ditu. Horietako bi ez dira bukaerako ikurrak izango eta bat, ostera, bukaerakoa (`expression`) da. Bukaerakoak ez diren bi ikurrak kudeatzeko, `Context` motako objektu bat eraikitzen da eta `true` balio boolearra esleitzen zaio bukaerakoa ez den ikur bakoitzari (`x` eta `y`). Hori guztia egin eta gero, *eta* logikoa egikaritzen da bukaerako ikurraren `interpret()` metodoa egikarituta. Metodoak, Java estandarraren `&&` eragiketa gauzatzen du ikurren balio boolearra aztertuta. Emaitza, Javaren irteera estandarretik ateratzen da.



7.18. irudia. Iterator diseinu-patroia.

7.4.4. Iterator

Datu multzo baten barna mugitu nahi badugu datu multzo horren barneadierazpidearen zehaztasunak ezagutu gabe, Iterator patroiak baliabide egokiak

eskaintzen dizkigu lan hori gauzatzeko. Patroi honek bide erraz eta sinple bat eraikitzen du datu multzoaren elementuak aztertu eta eskuratzeko.

Patroi honek lau parte-hartzaile ditu:

- Iterator: Parte-hartzaile honek interfaze bat erazagutzen du elementuak atzitu eta kudeatzeko.
- ConcreteIterator: Klase honek `Iterator` interfazea inplementatzen du. `next()` metodo generikoarekin, datu multzoaren hurrengo elementua eskuratzen da, eta `hasNext()` metodoarekin, ostera, datu multzoaren bukaerara heldu garen ala ez egiaztatzen da. Klasearen metodo eraikitzaileak `Aggregate` motako objektu bat behar du parametro gisa. `Aggregate` motako objektu batek `elements()` metodo bat izango du `Collection` motako objektu bat itzuliko duena. Iterator motako iteratzaileak (`ConcreteIterator`) `elements()` metodoa erabiliko du datu multzoaren elementuak eskuratzeko eta biltegiratzeko. Hori eginda, `next()` eta `hasNext()` metodoek arazorik gabe erabil dezakete datu multzoa.
- Aggregate: Parte-hartzaile honek interfaze bat erazagutzen du eta `Iterator` motako objektu bat eraikitzeko erabiliko da. `createIterator()` metodoak `Iterator` motako objektu bat itzuliko du eta `elements()` metodoak, ostera, `Collection` motako beste bat.
- ConcreteAggregate: Osagai honek `Aggregate` interfazea inplementatzen du, `createIterator()` eta `elements()` metodoak inplementatuta. `Iterator` motako iteratzailea eraikitzeko, `createIterator()` metodoak `ConcreteAggregate` klasearen instantzia bat pasatzen dio `ConcreteIterator` objektu bati. Horrela, iteratzaileak beti daki datu multzoaren jatorria zein den.

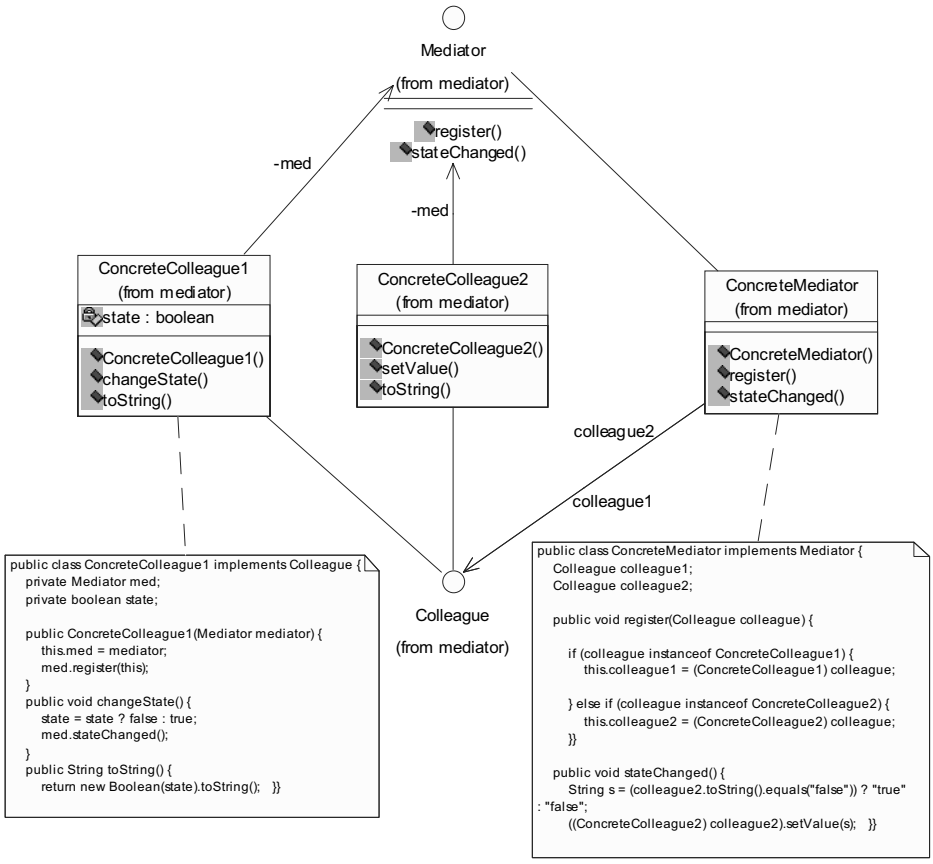
7.18. irudiak islatzen du nola egituratzen diren aipatutako lau parte-hartzaileen erlazioak. Ondoko listatuak azaltzen du bezero batek nola erabiliko lukeen patroiaren osagai bakoitza.

```
 public class Bezeroa {
    public static void main(String[] args) {

     String[] list = {"bat", "bi", "hiru", "lau", "bost", "sei"
};
     Aggregate ag = new ConcreteAggregate(list);
         // Iterator motako objektua eraiki
         //agregatua erabilita
        Iterator iter = ag.createIterator();

        // Datu multzoaren edukiak aztertu
            while (iter.hasNext()) {
                System.out.println(iter.next());   }}}
```

Bezeroak, lehendabizi, `String` array bat eraikitzen du eta `Aggregate` motako objektu bat eraikitzeko erabiliko du array hori. Metodo eraikitzaileak Java estandarraren `List` motako objektu batean gordeko du array-aren elementu bakoitza. Gero, `ConcreteAggregate` motako objektu batek itzuliko du objektu hori, `elements()` metodoa egikarituta. `Aggregate` motako objektua ondo eraiki bada, bezeroak `createIterator()` metodoa egikaritzen du `Iterator` motako objektu bat eskuratzeko. Iteratzailea eraikita, `hasNext()` eta `next()` metodoak egikaritzen ditu bezeroak elementuak iteratzailetik eskuratzeko. Eskuratutako elementuak irteera estandarretik ateratzen ditu bezeroak.



7.19. irudia. Mediator diseinu-patroia.

7.4.5. Mediator

Aplikazio informatiko batzuetan objektuen arteko elkarrekintzak sarritan aldatzen dira eta aplikazioaren diseinatzaileak Sekuentzia Diagrama berriak sortu behar ditu eta hainbat kode-lerro aldatu eta sortu. Egoera horietan, interesgarriagoa

izaten da objektu bat eraikitzea, objektuen arteko portaera kapsulatuko duena. Horrela, objektuen arteko elkarrekintzak zeharka tratatzen dira, desbideratu egiten baitira `Mediator` objektu-kapsulatzaile berri horretara. Beste era batera esanda, objektuen arteko *akoplea* leundu egiten da, `Mediator` objektuak bitartekari lana beteko baitu batzuen eta besteen mezuak kudeatuta.

Patroi honek lau parte-hartzaile ditu:

- Mediator: Parte-hartzaile honek interfaze bat erazagutzen du. Interfaze horrek metodoak antolatzen ditu `Colleague` motako objektuen arteko komunikazioa bideratzeko.
- ConcreteMediator: Klase honek `Mediator` interfazea inplementatzen du `register()` eta `stateChanged()` metodoak inplementatuta. Lehenengoak (`register()`), `Colleague` motako objektu bat erregistratu egiten du egiaztatuz pasatzen den parametroa `ConcreteColleague` motakoa dela. Egiaztatuz gero, `Colleague` motako aldagai bati esleitzen zaio parametroa. `stateChanged()` metodoak, `Colleague` motako objektu baten `false` balio boolearra `true`-ra aldatzen du eta alderantziz.
- Colleague: Parte-hartzaile honek interfaze bat erazagutzen du.
- ConcreteColleague: Osagai honek `Colleague` interfazea inplementatzen du. `Colleague` motako objektu batek `Mediator` motako objektu baten eskaerak jasotzen ditu. `Mediator` objektuak eskaeraren portaera kapsulatu egiten du eta `Colleague` motako objektu bakoitza `Mediator` motako objektuarekin komunikatzen da beste `Colleague` motako objektu batekin harremanetan jartzeko. `Mediator` objektuak bitartekari lana beteko du batzuen eta besteen mezuak kudeatzeko.

`Colleague` motako objektu batek beti eduki behar du erreferentzia bat `Mediator` motako objektu batera. Erreferentzia eraikiz gero, `Mediator` objektuaren `register()` metodoa egikaritzen da `Colleague` motako objektua erregistratzeko. Eragiketa bi horiek, `Colleague` motako objektu bat eraikitzen denean zertzen dira.

7.19. irudiak islatzen du nola egituratzen diren aipatutako lau parte-hartzaileen erlazioak. Ondoko listatuak azaltzen du bezero batek nola erabiliko lukeen patroiaren osagai bakoitza.

```
public class Bezeroa {
public static void main(String[] args) {
    // Eraiki Mediator motako objektua
    Mediator med = new ConcreteMediator();

    // Colleague objektuak eraiki
    //  Mediator objektua erabilita
```

```
        Colleague colleague1 = new ConcreteColleague1(med);
        Colleague colleague2 = new ConcreteColleague2(med);

        // Eskuratu hasierako balioak
  System.out.println("colleague1.toString() [" + colleague1 + "]");
  System.out.println("colleague2.toString() [" + colleague2 + "]");

      // Egoeraren balioa aldatu
        ((ConcreteColleague1) colleague1).changeState();

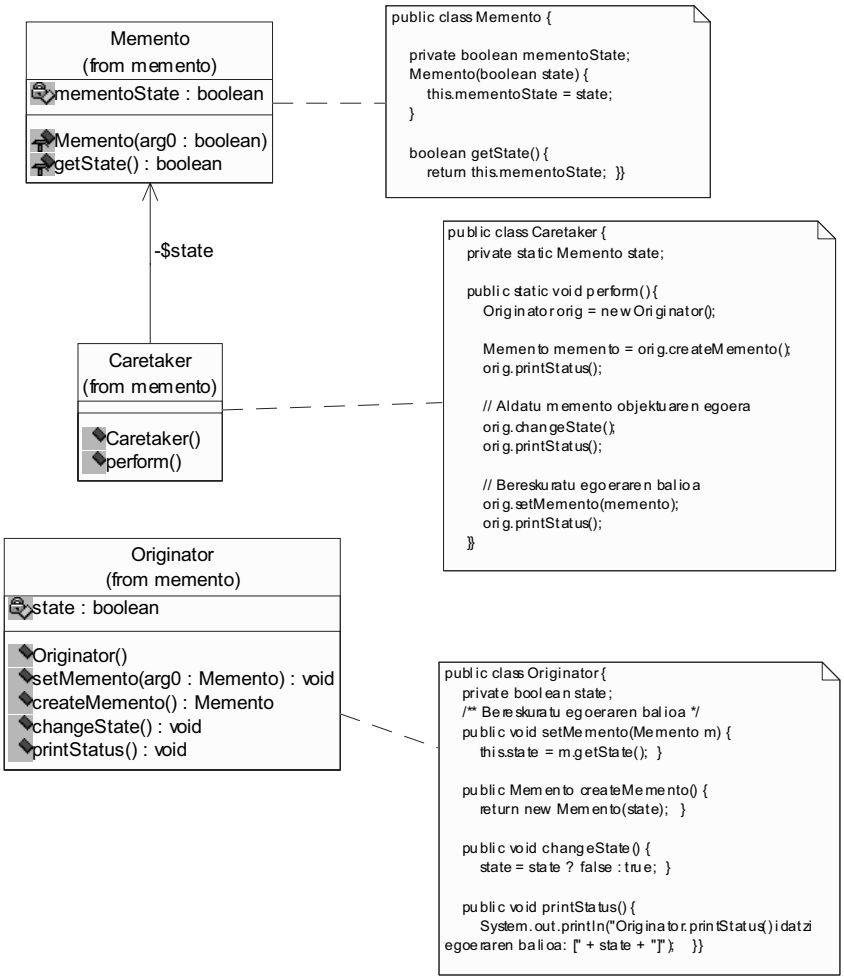
        // Eskuratu egoeraren azken balioa
          System.out.println("colleague1.toString() [" +
colleague1 + "]");
           System.out.println("colleague2.toString() [" +
colleague2 + "]");   }}
```

Bezeroak, `Colleague` motako objektuak eraiki baino lehen, `Mediator` motako objektu bat eraikitzen du. Objektu hori, `Colleague` motako objektu bakoitza eraikitzeko erabiliko du metodo eraikitzaileak.

`Colleague` bakoitzaren hasierako egoera aztertzeko, bezeroak, bakoitzaren egoera eskuratu eta irteera estandarretik ateratzen du. Hasieran, `Colleague` bakoitzak `false` balio boolearra du.

Bezeroak baieztatu nahi du mediator patroiaren funtzionalitatea ondo programatuta dagoela, eta horretarako, `Colleague` motako objektu bati (`colleague1`) bere egoera aldatzen dio `changeState()` metodoa egikarituta. Aldaketa gertatzen ari dela, `changeState()` metodoak `Mediator` objektua erabiltzen du gainerako beste aldaketa batzuk gauzatzeko. Bigarren aldaketa horiek, `Mediator` objektuaren `stateChanged()` metodoa egikarituta gauzatzen dira.

Aldaketak ondo burutu direla egiaztatzeko, bezeroak `Colleague` objektuen egoera berria eskuratzen du eta irteera estandarretik ateratzen du. Oraingoan, espero genuen bezala, `Colleague` bakoitzak `true` balio boolearra izango du. Gogoratu balio bat esplizituki aldatu duela bezeroak `changeState()` metodoa egikarituta; aitzitik, bigarrena, `Mediator` objektuak aldatu du inplizituki.



7.20. irudia. Memento diseinu-patroia.

7.4.6. Memento

Patroi honek objektu baten barne-egoera atzeman eta kanporatu egiten du, geroago berriro itzultzeko egoera horretara.

Patroi honek hiru parte-hartzaile ditu:

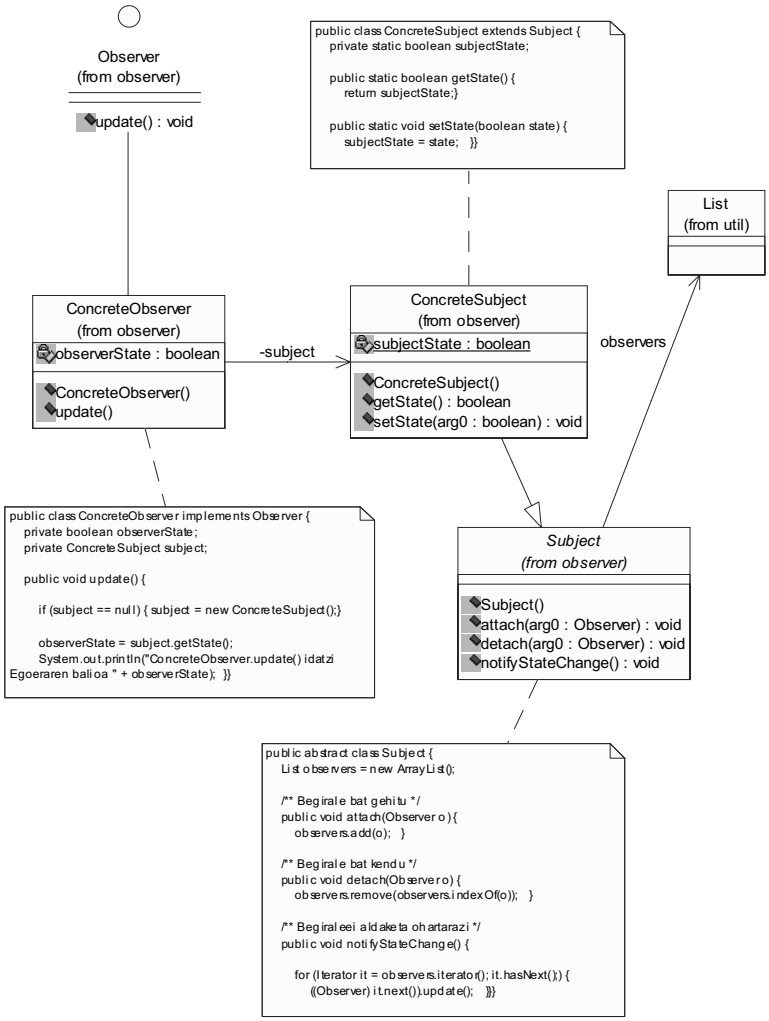
- Memento: Klase honek `Originator` motako objektuaren barne-egoera gordetzen du. Barne-egoera hori, une batean, `Originator` motako objektuak eskura dezake objektua egoera horretara berriro itzultzeko eta egikaritze-prozesua jarraitu, beste bide bat hartuta. Barne-egoera aztertzeko, zeharkako bidea ere erabil daiteke, horretarako `Caretaker` motako objektu bat erabiliko da baina objektu mota horrek `Originator` motako objektu bat beharko du barne-egoera atzitzeko.

- Originator: Klase honek sortuko ditu `Memento` motako objektuak. Bere barne-egoera gorde nahi duen bakoitzean, `Memento` motako objektu bat eraikiko du, eta gero, objektu hori erabiliko du berriro bere aurreko barne-egoera eskuratzeko.
- Caretaker: Bezeroak klase hau erabiliko du `Originator` motako objektuaren barne-egoera atzitu eta kudeatzeko. Atzipena ez da zuzenean egiten, beti, `Originator` motako objektu bat erabiliko da barne-egoera tratatzeko.

7.20. irudiak islatzen du nola egituratzen diren aipatutako hiru parte-hartzaileen erlazioak. Ondoko listatuak azaltzen du bezero batek nola erabiliko lukeen patroiaren osagai bakoitza.

```
public class Bezeroa {
public static void main(String[] args) {
        // Egikaritu Caretaker
        Caretaker.perform();   }}
```

Bezeroak objektu (`Originator`) baten barne-egoerak kudeatu nahi baditu, `Caretaker` motako objektu bat erabiltzen du. Objektu horrek, eragiketa (`perform()`) estatiko bat dauka programatuta, `Originator` motako objektua eraiki eta haren barne-egoerak tratatzeko. Iturburu-kodean ikusten den legez, `Originator` objektuaren barne-egoerak ez dira zuzenean tratatzen, `Originator` objektuak `Memento` motako objektu bat eraikitzen du eta haren barne-egoera biltegiratu egiten du. Barne-egoera aldatu nahi badugu edo objektuaren aspaldiko barne-egoera eskuratu, beti, `Originator` motako objektua erabili behar da bitartekari gisa.



7.21. irudia. Observer diseinu-patroia.

7.4.7. Observer

Patroi honekin objektuen arteko mendekotasunak kudeatzen dira. Erlazionatuta dauden objektuen artean, baten baten egoera aldatzen bada, begirale-lana betetzen ari den objektuak berehala banatzen du egoera berriaren informazio guztia beste objektuetara. Objektu bakoitzak egoera berriari ematen dion erantzuna, berezia eta berezitua izaten da gehienetan.

Patroi honek lau parte-hartzaile ditu:

- Observer: Parte-hartzaile honek interfaze bat erazagutzen du. Interfaze horrek begiraleen egoera eguneratzeko erabiliko den eragiketaren metodoa (`update()`) definitzen du. Konturatu, horrela eginda, begirale zehatz bakoitzak eguneraketa-prozesua bere idiosinkrasia propiora egokitu dezakeela.
- ConcreteObserver: Klase honek `Observer` interfazea inplementatzen du. Horretarako, `update()` metodoari inplementazio partikular bat emango zaio `Observer` motako objektu bakoitzean. `Observer` motako objektu bakoitzak erreferentzia bat izango du `Subject` motako objektu batera. Gero, `update()` metodoan, erreferentzia hori erabiliko du `Subject` motako objektuaren egoera berria eskuratzeko.
- Subject: Klase honek bere begiraleak (`Observer`) zein diren ondo baino hobeto daki. Begiraleek mota honetako objektua kontrolatzen dute eta objektu horri aldaketa bat gertatzen zaionean, `notityStateChange()` metodoa erabilita ematen da aldaketaren berri. Aurretik, begiraleen kopuruak ezaguna izan behar du `Subject` objektuarentzat eta horretarako, `Subject` klaseak metodoak (`attach()`/`detach()`) izango ditu begiraleak itsasteko eta `Subject` objektuan gordetzeko.
- ConcreteSubject: Klase honek `Subject` klasearen funtzionalitatea hedatu egiten du, hau da, `Subject` klasearen funtzionalitate bera izango du begiraleak tratatzeko, baina hemen, hedatu egingo da funtzionalitate hori, `Subject` zehatz bakoitzaren egoera hemen kudeatzen baita.

7.21. irudiak islatzen du nola egituratzen diren aipatutako hiru parte-hartzaileen erlazioak. Hurrengo listatuak azaltzen du bezero batek nola erabiliko lukeen patroiaren osagai bakoitza.

```
public class Bezeroa {
public static void main(String[] args) {
    // Begiraleak eraiki
    Observer observer1 = new ConcreteObserver();
    Observer observer2 = new ConcreteObserver();

    // Subject zehatza eraiki
// begiraleak kudeatzeko
    ConcreteSubject subject = new ConcreteSubject();

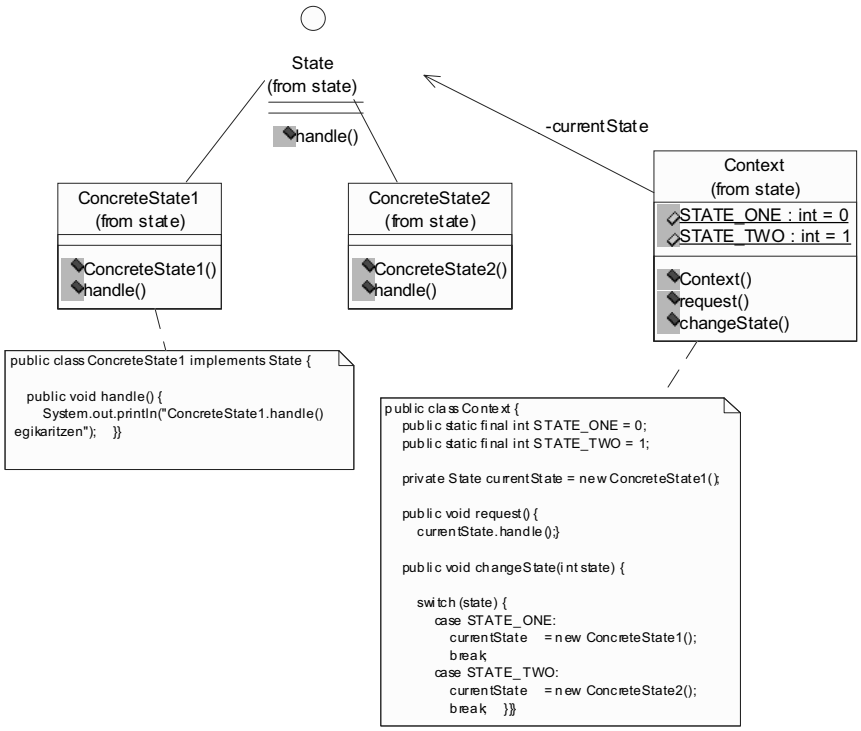
    // Begiraleak erantsi Subject objektuari
    subject.attach(observer1);
    subject.attach(observer2);

    // Egoera finkatu eta
     // aldaketaren berri, begiraleei eman
    subject.setState(true);
          subject.notifyStateChange();
```

```
// Begirale bat ezabatu
      subject.detach(observer1);

      // Egoera -berria- finkatu eta berriz,
      // aldaketaren berri, begiraleei eman
            subject.setState(false);
      subject.notifyStateChange();  }}
```

Bezeroak, lehendabizi, bi begirale (`Observer`) eta `Subject` motako objektu bat eraikitzen ditu. Horrela, patroiaren lehengaiak sortuta gelditzen dira. Bigarren urrats bezala, bezeroak lehengaiak nahastu egiten ditu eta horretarako `Subject` motako objektuari esaten dio zein begirale izango dituen aplikazioan. Lan hori `attach()` metodoa egikarituta gauzatzen da. Orain arte egindakoa ondo egituratuta dagoela egiaztatzeko, `Subject` motako objektuari egoera aldatzen zaio `setState()` metodoa egikarituta eta, berehala, begirale guztiei aldaketaren berri ematen zaie `notifyStateChange()` metodoa egikarituta. Gauzak korapilatzen joateko, begirale bat ezabatu egiten da begiraleen biltegitik eta egiaztatzeko begirale gisa ezabatuta gelditu dela, `Subject` motako objektuari egoera aldatzen zaio berriz, eta lehen bezala, begirale guztiei aldaketaren berri ematen zaie. Orain-goan, irteera estandarretik begirale bat gutxiago aterako da eta beste begiraleek egoera aldatuta edukiko dute.



7.22. irudia. State diseinu-patroia.

7.4.8. State

Diseinu-patroi honekin objektu baten barne-egoera aldatzen denean, objektuaren portaera ere aldatu egiten da. Kanpotik, badirudi objektua klasez aldatu dela eta klase-izaera berri bat eskuratu duela.

Patroi honek hiru parte-hartzaile ditu:

- State: Parte-hartzaile honek interfaze bat erazagutzen du. Osagai honek `Context` motako objektuaren egoera partikular baten portaera kapsulatzeko balioko digu. Interfaze honetan egoera partikularraren portaera kudeatuko duen metodoa (`handle()`) definitzen da.
- ConcreteState: Klase honek `Context` motako objektuaren egoeraren portaera inplementatzen du. Horretarako, `State` interfazean definitu den portaera (`handle()`) inplementatu egiten da hemen. Egoera zehatz (`ConcreteState`) bakoitzak egoera partikular bakoitzaren portaera inplementatuko du.
- Context: Klase honek unean uneko egoera kudeatzen du. Klaseak portaera desberdina izango du egoeraren balio partikularraren arabera, hau da, mota honetako objektu bati eskaera (`request()`) bat egiten zaionean, eskaera horren erantzuna, une horretako egoeraren balioaren menpe egongo da.

Mota honetako objektu batek beti edukiko du erreferentzia bat State motako objektu batera. Egikaritze-prozesuak aurrera egin ahala, erreferentzia aldatuz joango da.

7.22. irudiak islatzen du nola egituratzen diren aipatutako hiru parte-hartzaileen erlazioak. Ondoko listatuak azaltzen du bezero batek nola erabiliko lukeen patroiaren osagai bakoitza.

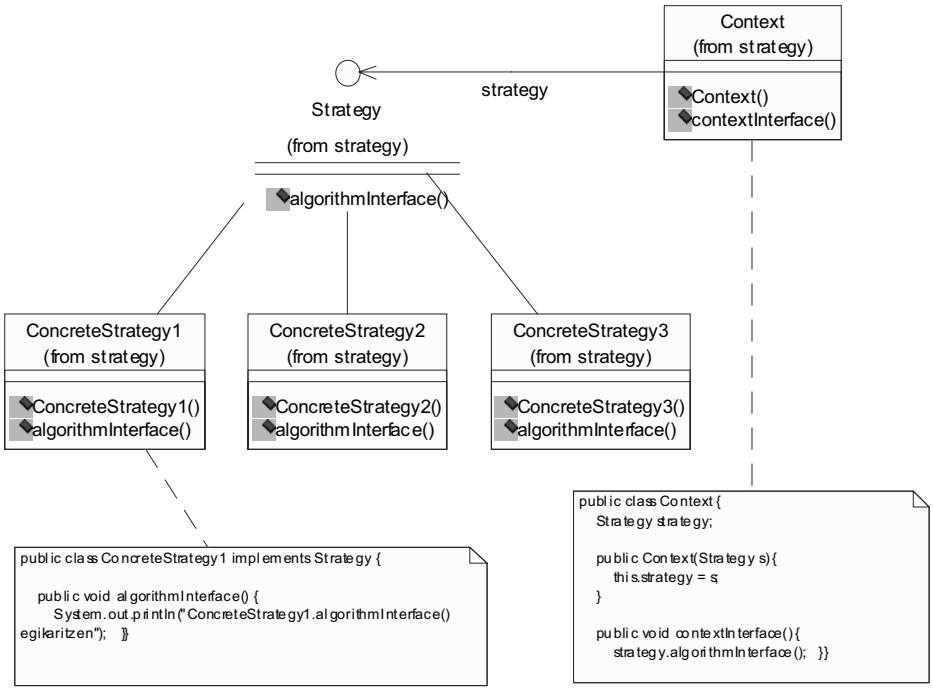
```
public class Bezeroa {
   public static void main(String[] args) {
     // Eraiki testuingurua
     Context ctx = new Context();

     // Egikaritu eskaera
     ctx.request();

     // Aldatu egoeraren balioa
     ctx.changeState(Context.STATE_TWO);

     // Egikaritu berriz eskaera
           ctx.request();   }}
```

Bezeroak, lehendabizi, `Context` motako objektu bat eraikitzen du gertatuko diren egoera-aldaketak tratatzeko. Iturburu-kodean ikusten den legez, lehenengo eskaerak (`request()`) `ConcreteState1` motako objektuaren `handle()` metodoa egikarituko du, hori baita `State` motako egoera lehenetsia. Gero, portaerak aldatzen direla egiaztatzeko, `Context` motako objektuari egoera aldatzen dio bezeroak, `changeState()` metodoa egikarituta. Egoera-aldaketak portaera-aldaketa dakarrela ziurtatzeko, bezeroak, `request()` metodoa egikaritzen du berriro, egiaztatzeko portaera benetan aldatu egiten dela.



7.23. irudia. Strategy diseinu-patroia.

7.4.9. Strategy

Patroi honekin algoritmo-familia bat definitzen dugu eta algoritmo bakoitza kapsulatu egiten dugu. Erabiltzerako orduan, hau da, exekuzio-garaian, algoritmoak trukatu egin daitezke eta bezeroaren programak algoritmoaren bertsio egokiena eskura eta exekuta dezake arazorik gabe.

Patroi honek hiru parte-hartzaile ditu:

- Strategy: Parte-hartzaile honek interfaze bat erazagutzen du. Interfaze horrek algoritmo guztien interfaze generikoa (`algorithmInterface()`) definitzen du. Gero, interfaze generiko horren izena eta signatura erabiliko ditu `Context` motako objektuak algoritmo partikular bat egikaritzeko.

- ConcreteStrategy: Algoritmo partikular bakoitzaren portaera, mota honetako klaseak instantziatuta kudeatuko da. Klase honek `Strategy` interfazea inplementatzen du eta inplementazio egokia itsasten dio `algorithmInterface()` metodoari, hau da, patroi honek `Strategy` interfazea erabiltzen du algoritmo bakoitza inplementatzeko. Horrela, `Context` motako objektuak ez du ezer aldatu beharko algoritmo partikular bat egikaritzeko.
- Context: Mota honetako objektu bat eraikitzen denean, `Strategy` motako objektu bat erreferentziatuko du hasiera-hasieratik. Horrela, eskaera bat egiten zaionean algoritmo bat egikaritzeko, erreferentziatuta duen `Strategy` motako objektuari pasatuko dio eskaera (`algorithmInterface()`) eta `Strategy` motako objektuak egikarituko du benetan algoritmoa, berak baitaki algoritmoaren portaerak nolakoa izan behar duen.

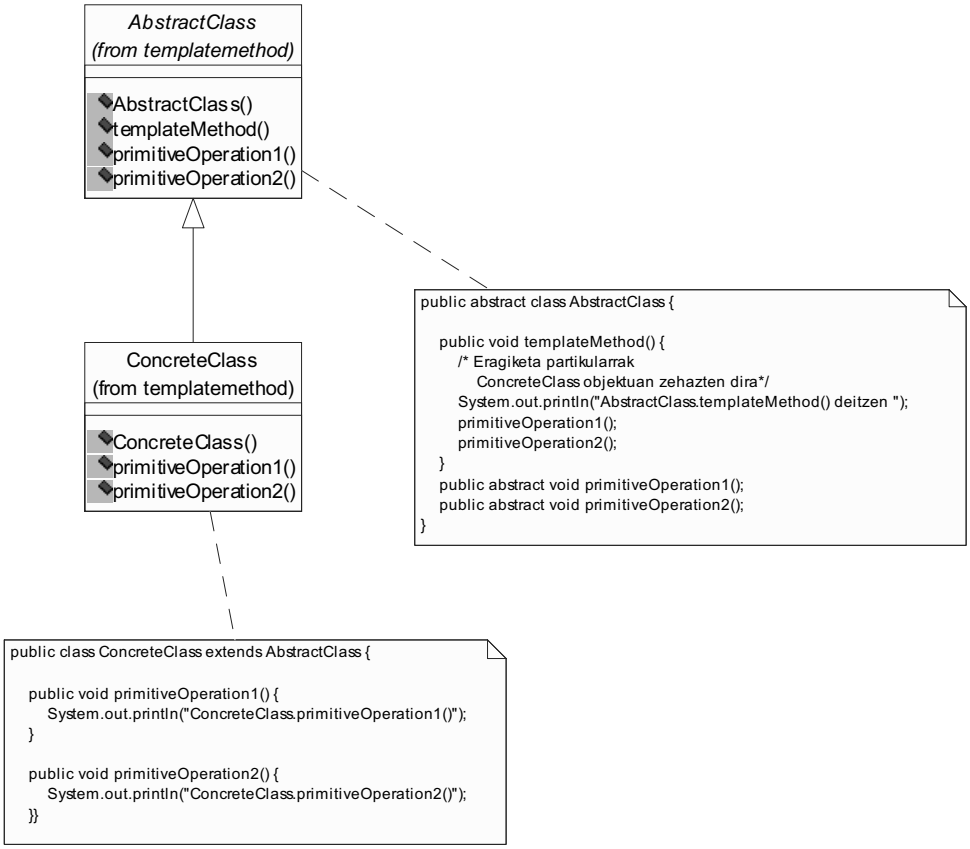
7.23. irudiak islatzen du nola egituratzen diren aipatutako hiru parte-hartzaileen erlazioak. Ondoko listatuak azaltzen du bezero batek nola erabiliko lukeen patroiaren osagai bakoitza.

```
public class Bezeroa {
 public static void main(String[] args) {
     // Algoritmoak egikaritzeko estrategia
     //  desberdinak eraiki
     Strategy strategy1 = new ConcreteStrategy1();
     Strategy strategy2 = new ConcreteStrategy2();
     Strategy strategy3 = new ConcreteStrategy3();

     // egikaritu estrategiak Context objektua erabilita
     new Context(strategy1).contextInterface();
         new Context(strategy2).contextInterface();
         new Context(strategy3).contextInterface();  }}
```

Bezeroak, algoritmo desberdinen portaera probatu baino lehen, `Strategy` motako objektu desberdinak eraikitzen ditu. Konturatu objektu bakoitzak ezagutzen duela algoritmo partikular baten portaera eta berak kontrolatu eta kudeatuko ditu algoritmoaren xehetasunak.

Algoritmoek ondo funtzionatzen dutela ziurtatzeko, bezeroak `Context` motako objektuak eraiki eta erreferentziak sortzen ditu aipatutako algoritmoetara. Hori eginez gero, `Context` objektu bakoitza konfiguratuta gelditzen da algoritmo zehatz bat kudeatzeko, hau da, erreferentzia bat du algoritmoa tratatuko duen `Strategy` motako objektu batera eta metodo (`contextInterface()`) bat izango du prest algoritmoa egikaritzeko. Bezeroak metodoak egikaritzen ditu eta, dena ondo badoa, `Context` motako objektu bakoitzak erreferentziatuta duen algoritmoa egikarituko du.



7.24. irudia. Template Method diseinu-patroia.

7.4.10. Template Method

Demagun aplikazio informatiko batean kudeaketa-arau bi, oso antzekoak, inplementatu nahi ditugula. Kudeaketa-arau batekin, bezeroaren eskaera osorik prestatuta egon arte ez da banatzen bezeroaren biltegira. Bigarren kudeaketa-arauarekin, zatika bana daiteke bezeroaren eskaera. Kudeaketa-arau biak diseinatu eta inplementatu nahi baditugu, Template Method diseinu-patroia oso ondo datorkigu, zeren inplementatu nahi dugun eragiketaren algoritmoa klase abstraktu batean defini baitezakegu. Hori eginda, azpiklaseak erabiliko ditugu algoritmoaren urratsak birfintzeko eta gure nahietara egokitzeko, algoritmoaren urrats bat edo beste birdefinituta eta aldi berean, egitura orokorra mantenduta.

Patroi honek bi parte-hartzaile ditu:

- AbstractClass: Klase abstraktu honek oinarrizko eragiketa abstraktuak definitzen ditu. Eragiketa abstraktu horiek azpiklaseetan egongo dira erabat inplementatuta. Klase honek algoritmo baten eskeletoa inplementatzen du

`templateMethod()` metodoan. Metodoak soilik ezagutzen du algoritmoaren eragiketa abstraktuak zein diren eta nola antolatu behar diren, baina ez, algoritmoa egikaritu ondoren sortuko diren emaitzak eta egikaritze-prozesuan zehar atzitu eta tratatuko diren datu motak. `ConcreteClass` motako objektuen ardura izango da hori.

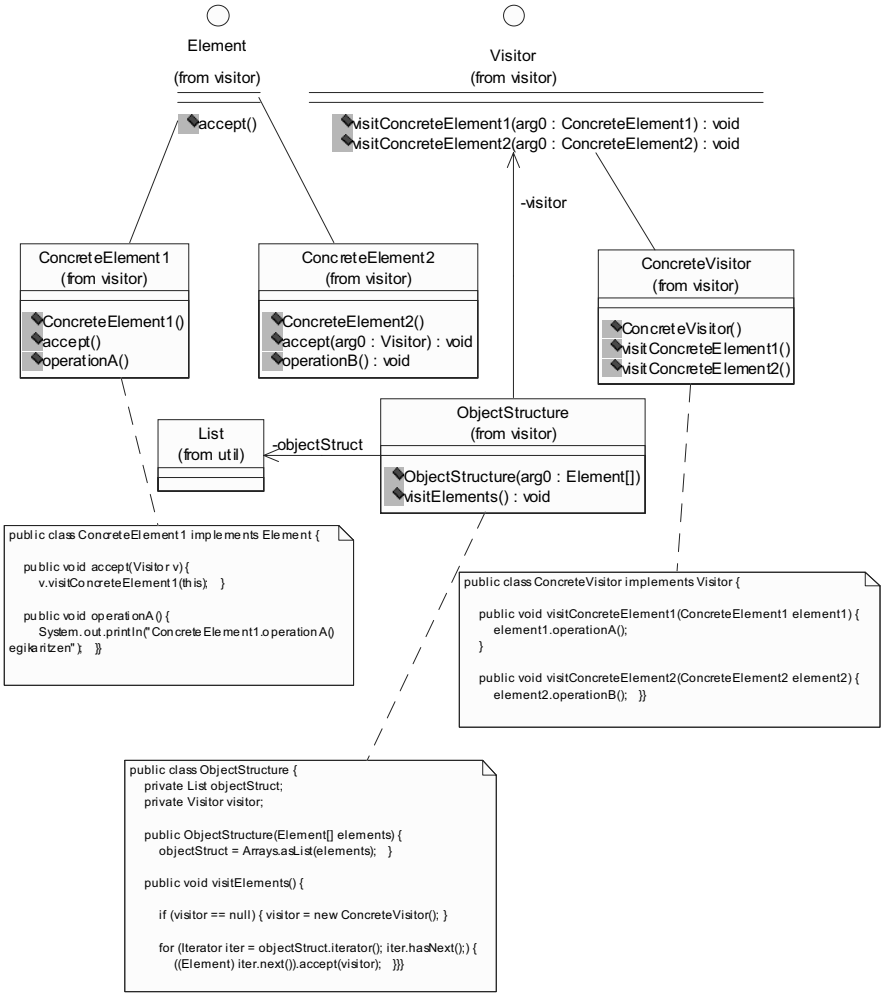
- ConcreteClass: Klase honek `AbstractClass` klasean definitzen diren oinarrizko eragiketa abstraktuak inplementatzen ditu. Klase honetan adierazten da argi eta garbi, oinarrizko eragiketa bakoitzak zer egiten duen eta nolako portaera izango duen. Portaera jakin hauek `templateMethod()` metodoan txertatuko dira, algoritmoak bere urratsak zintzo bete ditzan. Ikusten den legez, `templateMethod()` metodoak algoritmoaren zati inbarianteak kudeatzen ditu zuzenean, eta aldakorrak diren zatiak, ostera, `ConcreteClass` motako objektuari uzten dizkio berak trata ditzan.

7.24. irudiak islatzen du nola egituratzen diren aipatutako bi parte-hartzaileen erlazioak. Ondoko listatuak azaltzen du bezero batek nola erabiliko lukeen patroiaren osagai bakoitza.

```
public class Bezeroa {
 public static void main(String[] args) {
     // Eraiki klase zehatza
     AbstractClass ac = new ConcreteClass();

           // Egikaritu klase zehatzaren portaera zehatza
     ac.templateMethod(); }}
```

Bezeroak, `templateMethod()` metodoa egikaritu baino lehen, oinarrizko eragiketen portaera adierazi behar du. Patroi honetan portaera hori `ConcreteClass` motako objektu bat eraikita egituratzen da. Bezeroak horrela, `AbstractClass` motako objektuaren `templateMethod()` metodoa egikaritzen duenean, prest izango ditu osagai guztiak, hau da, oinarrizko eragiketak eta algoritmoaren fluxu nagusia.



7.25. irudia. Visitor diseinu-patroia.

7.4.11. Visitor

Demagun eragiketa polimorfiko bat inplementatu nahi dugula baina aldi berean, ez dugula aplikazioaren klase-hierarkia ukitu nahi. Horretarako, `Visitor` patroia erabilita, objektu bat prestatzen dugu eragiketa gauzatuko duena. Klase-hierarkiako objektuek, metodo bat erabilita, eragiketa delegatu egiten dute `Visitor` objektuan.

Patroi honek bost parte-hartzaile ditu:

- Visitor: Parte-hartzaile honek interfaze bat erazagutzen du. `ConcreteElement` motako klase bakoitzarentzat, `visit` eragiketa bat definitzen du. Eragiketaren izenak eta signaturak klase bat identifikatzen dute. Klase horrek `Visitor` motako objektu bati bidaltzen dio eskaera bat.

- ConcreteVisitor: Klase honek `Visitor` interfazearen eragiketak inplementatzen ditu. Eragiketa bakoitzak algoritmoaren zati bat inplementatzen du.
- Element: Parte-hartzaile honek interfaze bat erazagutzen du. `accept()` metodoa definitzen du, `Visitor` motako objektu bat parametro gisa eskatzen duena.
- ConcreteElement: Klase honek `Element` interfazea inplementatzen du `accept()` metodoari inplementazio egokia erantsita. Metodoaren inplementazioak `Visitor` motako objektuaren `visitConcreteElementX()` metodoa egikaritzen du, `Element` motako objektu bat parametro gisa pasatuta.
- ObjectStructure: Klase honek goi-mailako interfaze bat kudeatzen du `Visitor` motako objektu batek bere elementuak *bisitatzeko*. Elementuak bisitatzeko, elementu bakoitzaren `accept()` metodoa egikaritzen da. Elementu zehatz (`ConcreteElementX`) bakoitzak `accept()` metodoaren inplementazio propioa izango du. Horretarako, `Visitor` motako objektuaren `visitConcreteElementX()` metodo zehatza egikarituko da. `Visitor` motako `ConcreteVisitor` objektu bakoitzak inplementatuko ditu `visitConcreteElementX()` metodo horiek.

7.25. irudiak islatzen du nola egituratzen diren aipatutako hiru parte-hartzaileen erlazioak. Ondoko listatuak azaltzen du bezero batek nola erabiliko lukeen patroiaren osagai bakoitza.

```
public class Bezeroa {
 public static void main(String[] args) {
     // Eraiki elementuak eta array-an biltegiratu
     Element[] elements = new Element[2];
     elements[0] = new ConcreteElement1();
     elements[1] = new ConcreteElement2();

     // Eraiki ObjectStructure motako objektua eta
         //  elementuak bisitatu
     new ObjectStructure(elements).visitElements();  }}
```

Bezeroak `Element` motako objektuak eraikitzen ditu lehendabizi eta `Element` motako array batean biltegiratzen ditu. Eraikitako elementuak bisitatzeko, `ObjectStructure` motako objektu bat eraikitzen du bezeroak eta elementuak bisitatuz joaten da `ObjectStructure` motako objektuaren `visitElements()` metodoa egikarituta. Metodoak `Visitor` motako objektu bat eraikitzen du eta, begizta bat erabilita, `Element` motako objektu bakoitzaren `accept()` metodoa egikaritzen du `Visitor` motako objektua parametro gisa erantsita. Dakigun bezala, `accept()` metodoak `Visitor` motako objektu eraiki berriaren `visitConcreteElementX()` metodoa egikaritzen du eta, horretarako, `Element` motako objektua erabiltzen du parametro gisa.

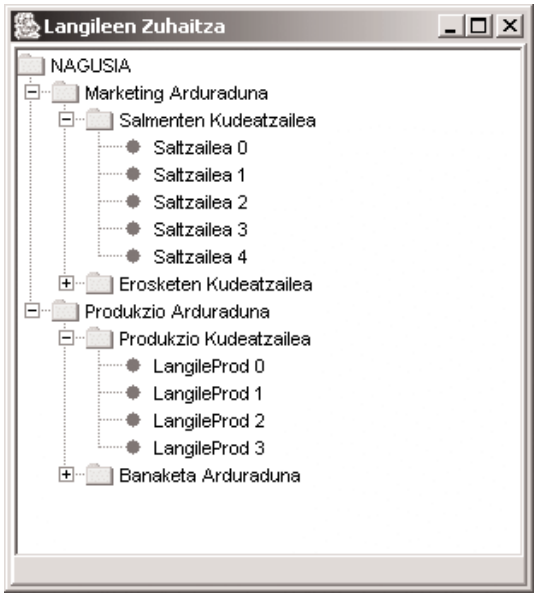
7.5. ADIBIDE KONPLEXUAK

Diseinu-patroien atal hau amaitzeko adibide batzuk landuko ditugu patroien portaera berezi hori aplikazio informatiko errealetan aztertzeko. Hasieran landuko ditugun adibideetan, aplikazio informatikoa patroi bakarraren inguruan eraikiko da eta ahal denik eta etekin handiena ateratzen saiatuko da patroiaren zirrikitu guztiak arakatuta.

Patroiak banaka aztertuz gero, ur handiagotan murgilduko gara eta azken adibideetan, aplikazio informatikoak patroi bat baino gehiago erabiliko du bere funtzionalitate guztiak inplementatzeko.

7.6. ADIBIDE KONPLEXUAK: COMPOSITE

Enpresa bateko nagusiak aplikazio informatiko bat erosi nahi du enpresaren pertsonalaren soldatak kudeatzeko. Enpresan lan egiten duten pertsonek antolaketa hierarkikoa dute pertsonala egituratzeko. Enpresak langileak (`Langile` klasea), arduradunak (`Nagusi` klasea) eta nagusia ditu. Arduradunak langileak ditu bere menpe eta langile horietariko bat edo batzuk azpiarduradunak izan daitezke eta, jakina, langileak eduki beraien menpe.



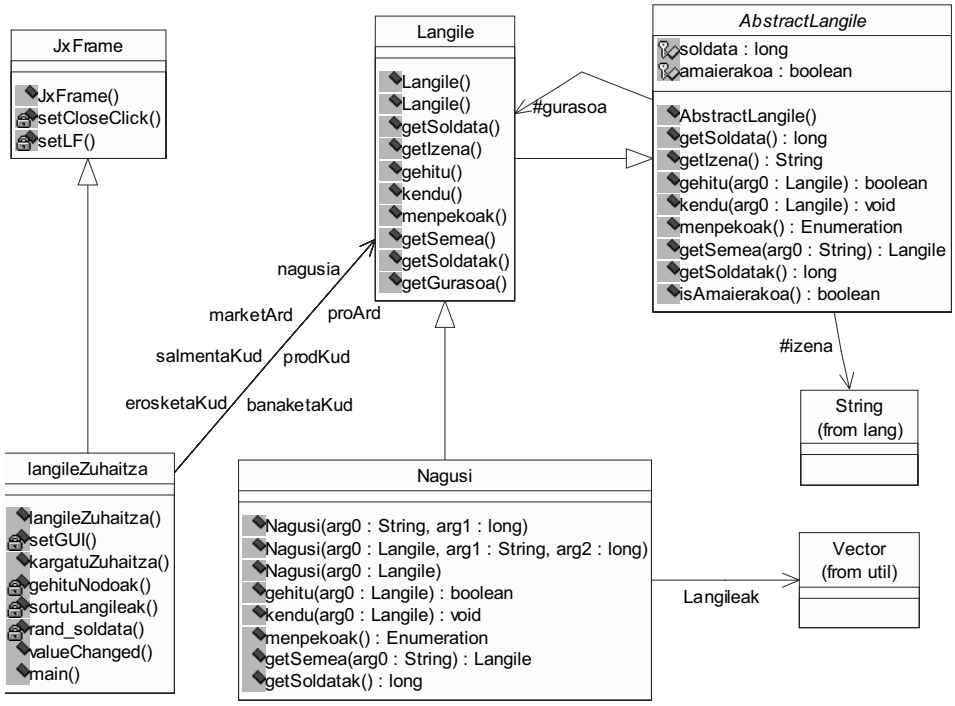
7.26. irudia. Aplikazioaren interfaze grafikoa.

Inplementatuko den lehen prototipoan, enpresaren nagusiak gauza gutxi eskatuko dizkigu. Hasteko nahikoa du interfaze grafiko (**7.26. irudia**) bat, enpresaren egitura islatzen duena. Egitura islatuz gero, langile edo sail oso baten soldata eskuratzearekin konformatuko da oraingoz.

Etorkizunean aplikazioa garatzen jarraitu nahi dugunez eta, aldi berean, gauzak txukun egin, pentsatu dugu patroiak erabiltzea aplikazioaren Klase Diagrama eraikitzeko.

Behin aztertuta lortu nahi dugun interfaze grafikoa, erabakitzen dugu Composite diseinu-patroia primeran datorkigula interfazea egituratzeko, zeren interfaze grafikoak zuhaitz-egitura bat islatzen baitu urratsez urrats. Patroiaren Component osagaia eraikitzeko `AbstractLangile` klase abstraktu bat sortuko dugu. Klase horrek honako metodoak izango ditu definituta osagaia konplexua denean, hau da, zuhaitzaren nodoa kudeatu nahi dugunean osagai horren elementuak[66] antolatzeko: `kendu()`, `gehitu()` eta `menpekoak()`.

Patroiaren Component osagaia eraikiz gero Leaf eta Composite parte-hartzaileak bilatzen hasten gara. Interfazearen langileen zuhaitza aztertuta, argi ikusten da Leaf rola enpresako langile arruntek beteko dutela eta Composite rola arduradunen eta kudeatzaileen esku gelditzen dela. Beraz, aipatutako osagai biak eraikitzeko, `Langile` eta `Nagusi` klaseak sortuko ditugu eta osagai guztiak harremanetan jarriko ditugu Klase Diagraman erlazio egokiak eraikita. **7.27. irudia**k aplikazioak erabiliko duen Klase Diagrama jasotzen du.



7.27. irudia. Aplikazioaren Klase Diagrama.

66. Gogoratu elementuak zuhaitzaren hostoak izan daitezkeela baina baita beste osagai konplexu batzuk ere.

`Langile` klaseak, hau da, patroiaren Leaf osagaiaren rola jokatuko duen elementuak, ondoko listatuan agertzen den iturburu-kodea izango du aplikazioan.

```
import java.util.*;
public class Langile extends AbstractLangile {
    public Langile(String _izena, long _soldata) {
        izena = _izena;
        soldata = _soldata;
        amaierakoa = true;}
 public Langile(Langile _gurasoa, String _izena, long _soldata) {
        izena = _izena;
        soldata = _soldata;
        gurasoa = _gurasoa;
        amaierakoa = true; }
 public long getSoldata() {
        return soldata;  }
 public String getIzena() {
        return izena;   }
 public boolean gehitu(Langile e) throws NoSuchElementException {
        throw new NoSuchElementException("Ez menpekoak");    }
 public void kendu(Langile e) throws NoSuchElementException {
        throw new NoSuchElementException("Ez menpekoak");    }
 public Enumeration menpekoak () {
        Vector v = new Vector();
        return v.elements ();  }
 public Langile getSemea(String s) throws NoSuchElementException {
        throw new NoSuchElementException("Ez Semearen");   }
 public long getSoldatak() {
        return soldata;   }
     public Langile getGurasoa() {
        return gurasoa;     }}
```

Konturatu nola `Langile` klaseak, iturburu-kodean, ez dien inongo inplementaziorik eman behar `gehitu()`, `kendu()` eta `getSemea()` metodoei, ez baitu inongo zentzurik zuhaitzaren bukaerako elementua izango den osagai bati elementu berriak gehitzeak edo kentzeak eta, are gutxiago, zein diren haien semeak eskatzeak. Aipatutako metodoek zentzua izango dute `Nagusi` klasean, klase horrek Composite osagaiaren rola beteko baitu patroian eta, beraz, elementuz osatua egongo da. `Nagusi` klaseak zuhaitzaren bukaerakoak ez diren elementuak kudeatu behar dituenez, iturburu-kode konplexuagoa izango du.

```
import java.util.*;
public class Nagusi extends Langile {
    Vector Langileak;

    public Nagusi(String _izena, long _soldata) {
        super(_izena, _soldata);
```

```
        amaierakoa = false;
        Langileak = new Vector();}
    public Nagusi(Langile _gurasoa, String _izena, long _soldata) {
        super(_gurasoa, _izena, _soldata);
        amaierakoa = false;
        Langileak = new Vector();}
    public Nagusi(Langile emp) {
        // Langile bat Nagusi bihurtzen dugu
        // eta horrek Langileak izateko aukera ematen dio
        super(emp.getIzena (), emp.getSoldata());
        Langileak = new Vector();
        amaierakoa = false;}
    public boolean gehitu(Langile e) throws NoSuchElementException {
        Langileak.add(e);
        return true;}
    public void kendu(Langile e) throws NoSuchElementException {
        Langileak.removeElement(e);}
    public Enumeration menpekoak () {
        return Langileak.elements ();}
    public Langile getSemea(String s) throws NoSuchElementException {
        Langile newLangile = null;

        if (getIzena().equals(s))
            return this;
        else {
            boolean bilatu= false;
            Enumeration e = menpekoak();
            while (e.hasMoreElements() && (! bilatu)) {
                newLangile = (Langile)e.nextElement();
                bilatu = newLangile.getIzena().equals(s);
                if (! bilatu) {
                    if (! newLangile.isAmaierakoa ()) {
                   newLangile = newLangile.getSemea(s);  } else
                        newLangile = null;
                    bilatu=(newLangile != null);  }}
            if (bilatu)
                return newLangile;
            else
                return null;  }}
    public long getSoldatak() {
        long sum = soldata;
        for (int i = 0; i < Langileak.size(); i++) {
            sum += ((Langile)Langileak.elementAt(i)).getSoldatak(); }
         return sum; }}
```

Esan bezala, `Nagusi` motako objektuak langileak kudeatu beharko ditu, eta horretarako, mota horretako objektu bakoitzarentzat `Vector` objektu bat eraikiko dugu zuhaitzean, haren azpian dauden objektuak —`Langile` objektuak edota beste `Nagusi` objektu batzuk— kudeatzeko. Hori dela eta, hortxe programatu

beharko ditugu `gehitu()`, `kendu()` eta `getSemea()` metodoak. Konturatu, `getSemea()` metodoan, semearen bilaketa, azpizuhaitza aztertuta gauzatzen dela, eta azkenean, `Langile` motako objektu bat itzuliko dela edo `null` balioa. `getSoldatak()` metodoarekin estrategia berari jarraitzen zaio soldatak kalkulatzeko.

Bukatzeko, programa nagusiak hurrengo listatuan agertzen den iturburukodea izango du aipatutako funtzionalitate guztia eraikitzeko.

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;

//swing klaseak
import javax.swing.text.*;
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.border.*;
import javax.swing.tree.*;


public class langileZuhaitza extends JxFrame
implements TreeSelectionListener {
    Langile nagusia, marketArd, proArd;
    Langile salmentaKud, erosketaKud;
    Langile prodKud, banaketaKud;

    JScrollPane sp;
    JPanel zuhaitzPanela;
    JTree zuhaitza;
    DefaultMutableTreeNode troot;
    JLabel soldataOrokorra;

    public langileZuhaitza() {
        super("Langileen Zuhaitza");
        sortuLangileak();
        setGUI();   }
    private void setGUI() {
        zuhaitzPanela = new JPanel();
        getContentPane().add(zuhaitzPanela);
        zuhaitzPanela.setLayout(new BorderLayout());

        sp = new JScrollPane();
        zuhaitzPanela.add("Center", sp);
  zuhaitzPanela.add("South", soldataOrokorra = new JLabel("       "));

        zuhaitzPanela.setBorder(new BevelBorder(BevelBorder.RAISED));
        troot = new DefaultMutableTreeNode(nagusia.getIzena());
        zuhaitza= new JTree(troot);
        zuhaitza.setBackground(Color.lightGray);
        kargatuZuhaitza(nagusia);
        /* Jarri zuhaitza scroller batean */
```

```
        sp.getViewport().add(zuhaitza);
        setSize(new Dimension(200, 300));
        setVisible(true);
    }
        public void kargatuZuhaitza(Langile goikoa) {
        DefaultMutableTreeNode troot;
        troot = new DefaultMutableTreeNode(goikoa.getIzena());
        zuhaitzPanela.remove(zuhaitza);
        zuhaitza= new JTree(troot);
        zuhaitza.addTreeSelectionListener(this);
        sp.getViewport().add(zuhaitza);

        gehituNodoak(troot, goikoa);
       zuhaitza.expandRow(0);
        repaint();
    }
   private void gehituNodoak(DefaultMutableTreeNode pnodoa, Langile
lang) {
        DefaultMutableTreeNode nodoa;

        Enumeration e = lang.menpekoak();
        if (e != null) {
            while (e.hasMoreElements()) {
                Langile newLang = (Langile)e.nextElement();
         nodoa = new
DefaultMutableTreeNode(newLang.getIzena());
                pnodoa.add(nodoa);
                gehituNodoak(nodoa, newLang);   }  }}
  private void sortuLangileak() {
        nagusia = new Nagusi("NAGUSIA", 200000);
        nagusia.gehitu(marketArd = new Nagusi("Marketing Arduraduna",
100000));
        nagusia.gehitu(proArd = new Nagusi("Produkzio Arduraduna",
100000));

        marketArd.gehitu(salmentaKud = new Nagusi("Salmenten
Kudeatzailea", 50000));
        marketArd.gehitu(erosketaKud = new Nagusi("Erosketen
Kudeatzailea", 50000));

        for (int i=0; i<5; i++)
            salmentaKud .gehitu(new Langile("Saltzailea "+ i,
rand_soldata(30000)));
        erosketaKud.gehitu(new Langile("Erosketen Admini", 20000));

        proArd.gehitu(prodKud = new Nagusi("Produkzio Kudeatzailea",
40000));
        proArd.gehitu(banaketaKud = new Nagusi("Banaketa Arduraduna",
35000));
```

```
        for (int i = 0; i < 4; i++)
            prodKud.gehitu( new Langile("LangileProd "+i,
rand_soldata(25000)));

        for (int i = 0; i < 3; i++)
            banaketaKud.gehitu( new Langile("Banaketa Idazkari "+i,
rand_soldata(20000)));
    }
 private long rand_soldata(long soldata) {
        return soldata +(long)(Math.random () -0.5)*soldata/5;
    }
 public void valueChanged(TreeSelectionEvent evt) {
        TreePath path = evt.getPath();
        String aukera = path.getLastPathComponent().toString();

        Langile lang = nagusia.getSemea(aukera);
        if (lang != null)
            soldataOrokorra.setText(new
Float(lang.getSoldatak()).toString());
    }
    static public void main(String argv[]) {
        new langileZuhaitza();   }}
```

Konturatu, `sortuLangileak()` metodoan `superklase=azpiklase` esleipenak ondorioak izango dituela objektuaren metodoak egikaritzen direnean. Adibidez:

```
nagusia = new Nagusi("NAGUSIA", 200000);
    nagusia.gehitu(marketArd = new Nagusi("Marketing Arduraduna",
100000));
    nagusia.gehitu(proArd = new Nagusi("Produkzio Arduraduna",
100000));
```

Lerro horietan gauza bitxi bat gertatzen da: `nagusia` aldagaia `Langile` motakoa bezala erazagutu da aurretik, eta lehen esan dugu `Langile` motako objektu batek ez duela `gehitu()` metodo bat inplementatu behar, zuhaitzaren bukaerako elementu bat delako; orduan, zergatik ez du errorerik ematen iturburu-kode horrek?

Azalpena liburu honen beste toki batean aurkeztu dugu eta honako hau da: gure egoeran, metodo birdefinitu bati dei egiten diogunean, egikarituko den metodoaren bertsioa erreferentziatzen ari garen objektuarena —gure kasuan `Nagusi` motakoa—izango da eta ez aldagaiaren (`Langile`) motakoa. Estekadura dinamikoa da eta auzia konpilazio-garaian konpondu beharrean, egikaritze-garaian erabakitzen da zein bertsio egikaritu.

7.7. ADIBIDE KONPLEXUAK: FAÇADE

Jakin badakigu aplikazio informatiko bat garatzen denean hiru mailako arkitektura bat erabilita, programatzaileak, hiru kategoriatan banatzen dituela aplikazioaren objektuak: batean, aurkezpen maila kudeatuko duten objektuak bilduko ditu; bestean, domeinuaren kudeaketa-arauak tratatuko dituzten objektuak kokatzen ditu eta hirugarren kategorian, ostera, domeinuaren objektuen iraunkortasuna ziurtatuko duten objektuak paratzen ditu. Objektuen iraunkortasunaren arazoa gehiago aldrebesten da iraunkortasun hori datu-base erlazionalak erabilita gauzatzen bada. Egoera horretan, objektuak mapeatu egin beharko dira eta prozesua ez da zaila izango objektuak arruntak direnean baina, objektuak egitura konplexua duenean arazoa ez da erraz konpontzen.

Beraz, arkitekturaren hirugarren maila hori ondo diseinatzen ez bada, programatzaileak arazo larriak izango ditu iraunkortasuna kudeatu nahi duenean.

Java estandarrean idatzitako aplikazio batek JDBC (Java Database Connectivity) APIa erabiltzen du datu-baseen kudeaketa sistema desberdinekin komunikatzeko. Tamalez, komunikazio hori ez da bat-batean eta automatikoki sortzen. Programatzaileak, kontrolatzaile bat kargatu behar du lehenengo, eta jarraian, konexio bat sortu datu-basearekin. Behin hori eginda, programatzailea, datu-basearekin jar daiteke harremanetan SQL aginduak egikarituta.

Programatzaileak aipatutako iraunkortasun mota hori behin bakarrik programatu behar badu, ez da saltsa handiagoetan sartuko eta zuzen-zuzenean programatuko ditu iraunkortasun erlazionalak eskatzen dizkion urrats guztiak. Aitzitik, programatzaileak, barra-barra programatu behar badu objektuen iraunkortasuna datu-base erlazionalak erabilita, pentsatzen hasiko da ez duela zentzu handirik behin eta berriz gurpila asmatzen egoteak.

Arazoari irteera bat bilatu nahian gure programatzailea gogoratuko da behin norbaitek ikasgai bat eman ziola Softwarearen Ingeniaritza irakasgaian horrelako buruhausteak bideratzeko. Trastelekua arakatuko du eta ustekabean agian, bilatzen ari zena agertuko zaio: Façade patroia. Gogoratuko da, halaber, garai batean, patroiaren izen frantsestuak hainbat oroitzapen goxo ekartzen zizkiola gogora, batez ere, uda hartakoak. Errealitateak irudimenari lekua hartuta, patroiari oratu eta objektuaren iraunkortasuna automatizatzen eta uniformizatzen saiatuko da gure programatzailea.

Badakigu oraintsu azaldu dugulako, Façade patroiak interfaze bateratu bat eraikitzen duela objektu multzo baten lana gauzatzeko. Patroiak erantzun egokia eraikiko dio aplikazioaren bezeroari, horrek eskaera bat bidaltzen dionean. Gure kasuan, iraunkortasuna ziurtatzeko, bezeroak ez du arduratu behar Java estandarrean egin behar diren eragiketak programatzeaz. Horretarako, `Façade` motako objektu bat eraikiko da eta objektu horrek gauzatuko ditu bere mailako eragiketa

guztiak, hau da, kontrolatzailea kargatu eta konexio egokiak sortu. Gure adibidean, façade rola beteko duen objektua, `Datubase` motako objektu bat izango da:

```
public Datubase(String driver) {
        types = new String[ 1];
      types[ 0]  = "TABLE";          //Hasieratu array mota
       // types[ 0]= null;
        try {
            Class.forName(driver);
        }  //   Kargatu zubia
        catch (Exception e) {
            System.out.println(e.getMessage());   }}

public void Open(String url, String cat) {
        catalog = cat;
        database_url = url;
        try {
            con = DriverManager.getConnection(url);
            dma =con.getMetaData();   // Eskuratu Metadatuak
              } catch (Exception e) {
                  System.out.println(e.getMessage());  }}
```

Aplikazioaren bezeroak `Datubase` motako objektu bat eraikitzen duenean metodo eraikitzailea egikarituta, kontrolatzailea ere kargatzen du, `Class` klasearen `forName()` metodo estatikoa egikarituta. Hori egin eta gero, bezeroak `Datubase` motako objektuaren `Open()` metodoa egikaritu nahiko du datu-basearekin konektatzeko. Hori egiteko, `DriverManager` klasearen `getConnection()` metodoa egikarituko da lehendabizi.

Hasierako eragiketak burutuz gero, aplikazioaren bezeroak SQL aginduak egikaritu nahiko ditu datu-basera atzitu eta balioak txertatu, ezabatu eta eguneratzeko. `Datubase` motako objektuak `Execute()` metodoa erabiliko du aipatutako funtzionalitatea egituratzeko:

```
    public Results Execute(String sql) {
        // SQL agindua egikaritu
        results = null;
        try {
            Statement stmt = con.createStatement();
            results = new Results(stmt.executeQuery(sql));
        } catch (Exception e) {
                    System.out.println("egikaritze errorea:
"+e.getMessage());
        }  return results;}
```

`Execute()` metodoak `Statement` motako objektu bat eraikitzen du `Connection` motako objektuaren `createStatement()` metodoa egikarituta eta prest uzten digu azpiegitura teknologiko guztia SQL aginduak egikaritzeko. SQL

agindua egikaritzeko, `Statement` motako objektu eraiki berriaren `executeQuery()` metodoa egikaritzen da.

```
    public String[] getTableNames() {
        String[] tbnames = null;
        Vector tname = new Vector();
    // Gehitu taularen izenak Bektore bati

        try {
   results = new Results(dma.getTables(catalog, null, "%", types));
        } catch (Exception e) {
            System.out.println(e);  }
        while (results.hasMoreElements())
            tname.addElement(results.getColumnValue("TABLE_NAME"));
        // Kopiatu taularen izenak String motako array batean
        tbnames = new String[tname.size()];
        for (int i=0; i< tname.size(); i++){
            tbnames[i] = (String)tname.elementAt(i);
            System.out.println("Taula IZENAK:  "+tbnames[i]);};
        return tbnames;  }

public String[] getColumnNames(String table) {
        // Itzuli zutabeen izenak
        String[] tbnames = null;
        Vector tname = new Vector();
        try {
  results = new Results(dma.getColumns(catalog, null, table, null));
            while (results.hasMoreElements() )
   tname.addElement(results.getColumnValue("COLUMN_NAME"));
        } catch (Exception e) {  System.out.println(e);}
        tbnames = new String[tname.size()];
        for (int i=0; i< tname.size(); i++)
            tbnames[i] = (String)tname.elementAt(i);
        return tbnames;}

public String getColumnValue(String table, String columnName) {
        // Zutabe baten izena itzuli
        String res = null;
        try {
            if (table.length()>0)
       results = Execute("Select "+columnName+" from "+table+" order
by "+columnName);
            if (results.hasMoreElements())
             res = results.getColumnValue(columnName);
        } catch (Exception e) {
System.out.println("Column value error" +columnName+ e.getMessage());
        }

        return res;   }
```

```
public String getNextValue(String columnName) {
        // Zutabearen hurrengo balioa itzuli

        String res = "";
        try {
            if (results.hasMoreElements())
                res = results.getColumnValue(columnName);
              } catch (Exception e) { System.out.println("errorea
hurrengo balioan"+columnName+ e.getMessage());  }
              return res;   }
```

Javaren JDBC APIak SQL agindu bat egikaritzen duenean `Statement` motako objektuaren metodoak erabilita, `ResultSet` motako objektu batean uzten ditu SQL aginduaren emaitzak. Emaitzak `ResultSet` motako objektuan daudenean, bezeroak objektuaren metodoak erabiltzen ditu emaitzak banan-banan eskuratzeko. Horrek nahitaez eskatzen du bezeroak ezagutu egin behar dituela ResultSet objektuaren diseinua eta eskaintzen duen funtzionalitate guztia. Badago beste estrategia bat aplika daitekeena: ResultSet objektuaren portaera kapsulatu beste objektu batean eta bigarren objektu hori eskaini bezeroari emaitzak kudeatzeko. Estrategia hori erabiliko da aztertzen ari garen adibidean.

`Results` motako objektuak JDBCren `ResultSet` objektuaren portaera kapsulatu egingo du bezeroari errazteko SQL aginduen emaitzen kudeaketa. Objektuak, metodoak eskaintzen dizkio bezeroari `ResultSet` motako objektuaren datuak eta metadatuak aztertzeko:

```
public Results(ResultSet rset)
   {
      rs = rset;
      try
      {
      // Eskuratu aldi berean, metadatuak eta zutabe kopurua
      rsmd = rs.getMetaData();
      numCols = rsmd.getColumnCount();
      }
            catch (Exception e)
          { System.out.println("ResultSet errorea"+e.getMessage());
}}
```

`Results` motako objektuaren metodo eraikitzaileak `ResultSet` motako objektu bat erabiltzen du eraikitze-prozesuari SQL aginduaren emaitza eransteko.

Metodo eraikitzailean, lehenengo eta behin, `ResultSet` objektua `Results` objektuan erazagutu den `ResultSet` motako `rs` aldagaiari esleitzen zaio. Horrela, aldagai hori erabiliko dugu gero, objektuaren metodoetan, `ResultSet` objektuaren balioak aztertzeko. Metodo eraikitzaileak, aldi berean, `ResultSet` objektuaren

metadatuak eta zutabe kopurua eskuratzen ditu `getMetadata()` eta `ResultSetMetaData` objektuaren `getColumnCount()` metodoak egikarituta.

```
public String[ ] getMetaData()
   {   // Array bat itzuli zutabe guztien izenekin
       String md[ ] = new String[ numCols] ;
       try
          {   for (int i=1; i<= numCols; i++)
          md[ i-1] = rsmd.getColumnName(i);  }
       catch (Exception e)
       { System.out.println("metadatu errorea"+e.getMessage());}
       return md;   }

   public boolean hasMoreElements()
   {   try{
       return rs.next();  }
   catch(Exception e){
       System.out.println("next metodoaren errorea "+e.getMessage());
       return false;}  }

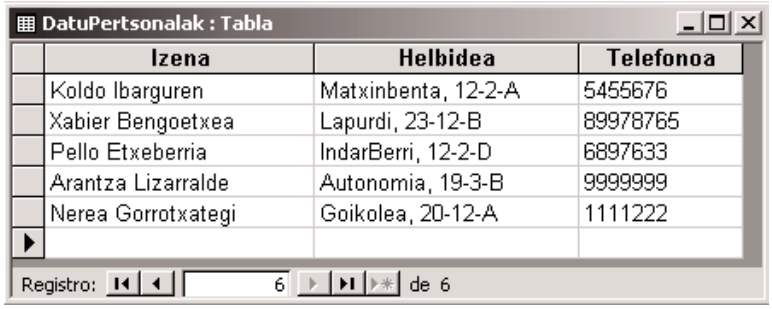
 public String[ ] nextElement()
   {   //  Lerroaren edukiak array batean Kopiatu
       String[ ] row = new String[ numCols] ;
       try
       {   for (int i = 1; i <= numCols; i++)
             row[ i-1] = rs.getString(i);  }
       catch (Exception e)
       { System.out.println("nextElement metodoan
errorea"+e.getMessage());}
    return row;  }

public String getColumnValue(String columnName)
   {   String res = "";
     try   {
     res = rs.getString(columnName);  }
   catch (Exception e)
   { System.out.println("Errorea Zutabe balioan
:"+columnName+e.getMessage());}
   return res;
public String getColumnValue(int i)
   {   String res = "";
     try   {
     res = rs.getString(i);  }
   catch (Exception e)
   { System.out.println("Errorea Zutabe balioan :" + i + " " +
e.getMessage());}
   return res;  }

public void finalize()
```

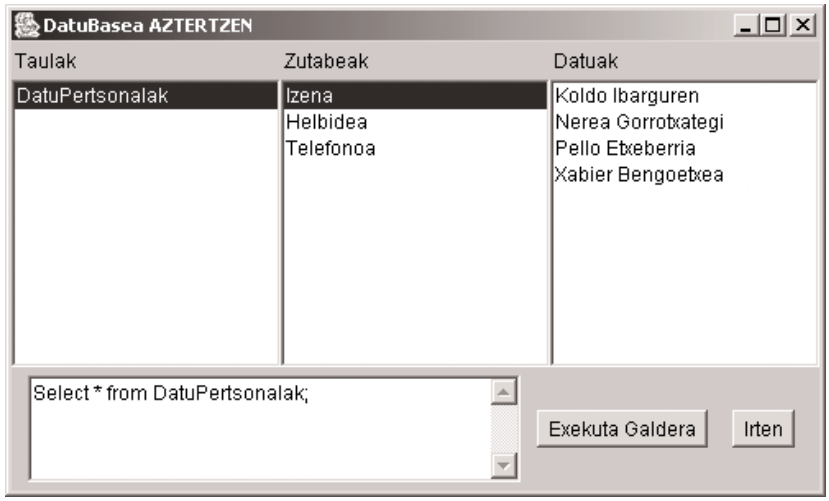
```
{    try{ rs.close();}
        catch (Exception e)
              { System.out.println(e.getMessage());}    }}
```

`Datubase` eta `Results` motako objektuekin eraiki dugun Façade patroiak automatizatu eta uniformizatu egiten du Javaren JDBC APIaren kudeaketa, zeren abstrakzio maila igo egiten baita eta bezeroak buruhauste gutxiago izango baititu datu-basearekin harremanetan jartzeko.

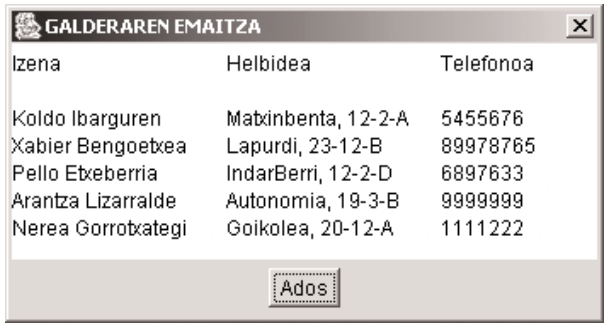


7.28. irudia. Aplikazioaren datu-basea.

Atal hau bukatzeko, eraiki dugun baliabidea aplikazio informatiko batean erabiliko dugu. Aplikazioak datu-base baten balioak (**7.28. irudia**) kudeatu nahi ditu eta horretarako **7.29. irudia**n agertzen den interfaze grafiko nagusia erabiliko du. Interfaze grafikoak funtzionalitate bi inplementatzen ditu: goiko zatian, datu-basearen taulak, taulen zutabeak eta zutabeen balioak ikuskatzen dira hiru azpileihotan. Beheko zatian, erabiltzailea gauza bada, SQL aginduak idatziko dira zuzenean eta `ExekutaGaldera` botoia zapalduta agindua egikaritu egiten da. Aplikazioak `Dialog` motako objektu batean aterako (**7.30. irudia**) ditu emaitzak.

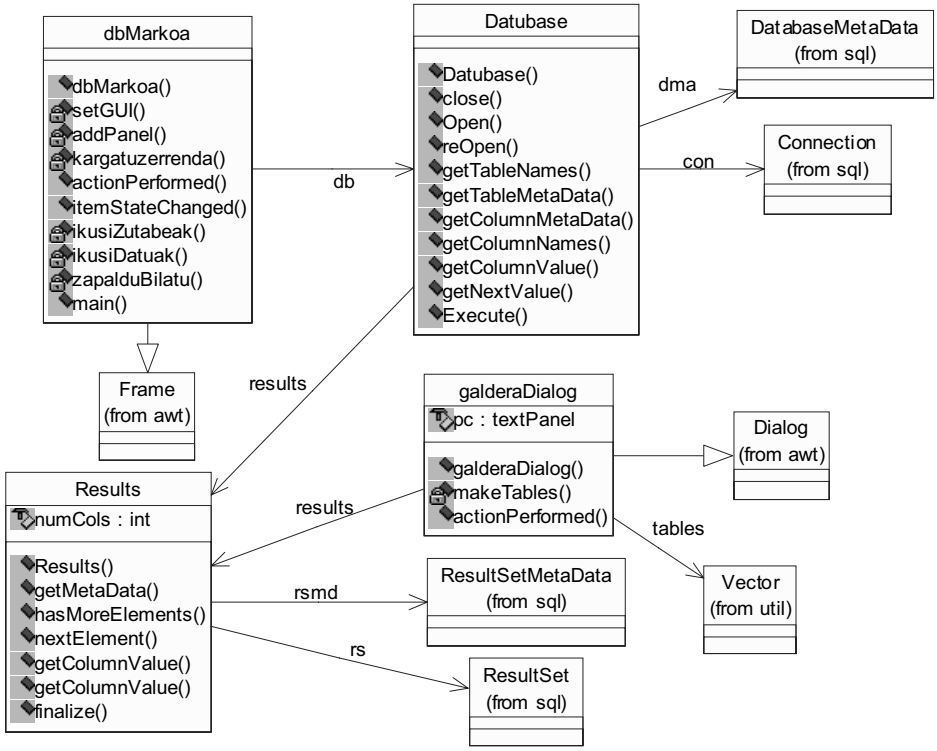


7.29. irudia. Aplikazioaren interfaze grafiko nagusia.



7.30. irudia. SQL aginduaren emaitzak.

Aplikazioaren betekizunak aztertuta, **7.31. irudia**n agertzen den Klase Diagrama erabiliko dugu aplikazioa inplementatzeko. Aplikazioak arestian eraiki dugun Façade patroiaren baliabideak erabiliko ditu datu-basearen informazioa kudeatzeko.



7.31. irudia. Aplikazioaren Klase Diagrama.

`dbMarkoa` da aplikazioaren informazio-fluxua kontrolatu eta kudeatuko duen objektua. Bera jarriko da harremanetan patroiaren Façade objektu kanonikoaren rola betetzen duen `Datubase` motako objektuarekin. **7.32. irudia**k `dbMarkoa`

objektuak jarraituko duen Sekuentzia Diagrama islatzen du. Sekuentzia Diagramak zati nagusi bi ditu: zati batek, `ExekutaGaldera` botoia zapalduta egikaritu behar den mezu multzoa biltzen du eta `zapalduBilatu()` metodoarekin egikaritzen da. Beste zatia, ostera, interfaze grafikoaren goiko atalaren kudeaketaz arduratuko da, eta horretarako, `itemStateChanged()` metodoa programatuko da interfaze grafikoaren azpileihoak eguneratzeko.



7.32. irudia. Aplikazioaren Sekuentzia Diagrama orokorra.

Ondoko listatuak jasotzen du nola gelditukо diren inplementatuta Sekuentzia Diagramaren mezuak.

```
import java.net.URL;
import java.sql.*;
import java.util.*;
import java.awt.*;
import java.awt.event.*;

public class dbMarkoa extends Frame
implements ActionListener, ItemListener {
    Datubase db;
    java.awt.List Taulak, Zutabeak, Datuak;
    TextArea galdera;
    Button Bilatu, Irten;
    public dbMarkoa() {
        super("DatuBasea AZTERTZEN");
        setGUI();

        db = new Datubase("sun.jdbc.odbc.JdbcOdbcDriver");
        db.Open("jdbc:odbc:Pertsonak", null);
      String tizenak[] = db.getTableNames();

        kargatuzerrenda(Taulak, tizenak);  }

private void setGUI() {
        setBackground(Color.lightGray);
        setLayout(new BorderLayout());
        Panel pn = new Panel();
        add("North", pn);
        pn.setLayout(new GridLayout(1,3));
        pn.add(new Label("Taulak"));
        pn.add(new Label("Zutabeak"));
        pn.add(new Label("Datuak"));
        Panel pc = new Panel();
        add("Center", pc);
        pc.setLayout(new GridLayout(1,3));
        pc.add(Taulak = new java.awt.List(15));
        pc.add(Zutabeak = new java.awt.List(15));
        pc.add(Datuak = new java.awt.List(15));
        Taulak.addItemListener(this);
        Zutabeak.addItemListener(this);

        Panel ps = new Panel();
        add("South", ps);
        ps.add(galdera = new TextArea("", 3,40));
        addPanel(ps, Bilatu = new Button("Exekuta Galdera"));
        addPanel(ps, Irten = new Button("Irten"));
        Bilatu.addActionListener(this);
        Irten.addActionListener(this);
        setBounds(100, 100, 500, 300);
        setVisible(true);  }
    //------------ ----------------- -----
    private void addPanel(Panel ps, Component c) {
        Panel p = new Panel();
```

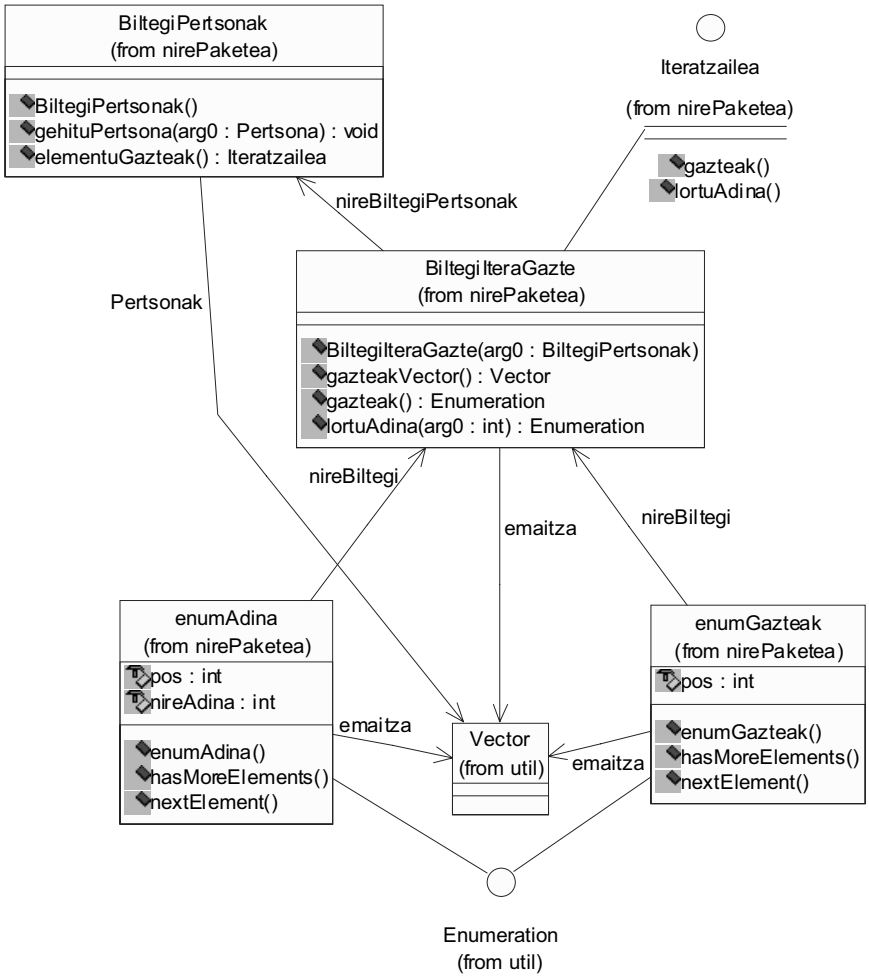
```
        ps.add(p);
        p.add(c);  }
    //------------ ------------------ -----
    private void kargatuzerrenda(java.awt.List zerrenda, String[] s) {
        zerrenda.removeAll();
        for (int i=0; i< s.length; i++)
            zerrenda.add(s[ i]);  }
    //------------ ------------------ -----
    public void actionPerformed(ActionEvent e) {
        Object obj = e.getSource();
        if (obj == Irten)
            System.exit(0);
        if (obj == Bilatu)
            zapalduBilatu();  }
    //------------ ------------------ -----
    public void itemStateChanged(ItemEvent e) {
        Object obj = e.getSource();
        if (obj == Taulak)
            ikusiZutabeak();
        if (obj == Zutabeak)
            ikusiDatuak();  }
    //------------------
    private void ikusiZutabeak() {
        String zutIzenak[] =
db.getColumnNames(Taulak.getSelectedItem());
        kargatuzerrenda(Zutabeak, zutIzenak);   }
    //------------ ------------------ -----
    private void ikusiDatuak() {
        String zutIzena = Zutabeak.getSelectedItem();
        String zutBalioa = db.getColumnValue(Taulak.getSelectedItem(),
zutIzena);
        Datuak.setVisible(false);
        Datuak.removeAll();
        Datuak.setVisible(true);

        zutBalioa = db.getNextValue(Zutabeak.getSelectedItem());

        while (zutBalioa.length()>0) {
            Datuak.add(zutBalioa);
         zutBalioa =
db.getNextValue(Zutabeak.getSelectedItem());}}
    //------------ ------------------ -----
    private void zapalduBilatu() {
        Results rs = db.Execute(galdera.getText());
        String zutIzenak[] = rs.getMetaData();
        Zutabeak.removeAll();
        galderaDialog g = new galderaDialog(this, rs);
        g.show();  }
    //------------ ------------------ -----
    static public void main(String argv[]) {
        new dbMarkoa();   }}
```

7.8. ADIBIDE KONPLEXUAK: ITERATOR

Aplikazio informatiko bat eraiki nahi dugu datu-base txiki bat kudeatuko duena. Datu-baseak pertsonen informazioa gordeko du eta gauzak ez nahasteko, `Vector` motako objektu batean biltegiratuko dugu informazio hori. Datu-basearen informazio gehiena behin eta berriz atzitu behar dugunez, pentsatu dugu Iterator patroia erabiltzea ondo datorkigula funtzionalitate hori lortzeko, hau da, datu-basearen barna mugitu nahi dugu haren barne-adierazpidearen zehaztasunak ezagutu gabe. Eraiki nahi ditugun iteratzaileek pertsona bakoitzaren adina aztertuko dute eta lehenengo egoeran, 25 urte baino gehiagoko pertsonen multzoa aterako da irteera estandarretik. Bigarren egoeran 35 urte baino gehiagoko pertsonen multzoa aterako da; hori ere irteera estandarretik. Aplikazioaren betekizunak aztertuta **7.33. irudia**n agertzen den Klase Diagrama diseinatu dugu Iterator patroia egituratzeko.



7.33. irudia. Aplikazioaren Klase Diagrama.

Datu-basearen funtzioa beteko duen objektua `BiltegiPertsonak` izango da eta ondoko listatuan agertzen den iturburu-kodea izango du:

```
import java.util.*;
public class BiltegiPertsonak {

 Vector Pertsonak;

    public BiltegiPertsonak() { Pertsonak = new Vector();}

    public void gehituPertsona(Pertsona pertsona)
                { Pertsonak.addElement(pertsona);}

    public Iteratzailea elementuGazteak (){
    return new BiltegiIteraGazte(this);}  }
```

Aplikazio nagusia egituratzen joateko, lehenengo eta behin, datu-basearen funtzioa beteko duen `BiltegiPertsonak` motako objektua eraiki eta hasieratu egingo dugu:

```
BiltegiPertsonak pertsonenBiltegia = new BiltegiPertsonak();

 Pertsona per1= new Pertsona("Koldo", 27);
 Pertsona per2= new Pertsona("Amaia", 17);
 Pertsona per3= new Pertsona("Lorea", 37);
 Pertsona per4= new Pertsona("Xabier", 15);
 Pertsona per5= new Pertsona("Nerea", 55);

 pertsonenBiltegia.gehituPertsona(per1);
 pertsonenBiltegia.gehituPertsona(per2);
 pertsonenBiltegia.gehituPertsona(per3);
 pertsonenBiltegia.gehituPertsona(per4);
 pertsonenBiltegia.gehituPertsona(per5);
```

Datu-basea eraikiz gero, `BiltegiPertsonak` motako objektuaren `elementuGazteak()` metodoa egikaritzen dugu `Iteratzailea` motako objektu bat eraikitzeko.

```
import java.util.*;
public interface Iteratzailea {
         public Enumeration gazteak();
                public Enumeration lortuAdina(int adina);
         public Vector gazteakVector();  }
```

`Iteratzailea` interfazea inplementatzen duen klaseak (`BiltegiIteraGazte`) nahitaez, `gazteak()`, `lortuAdina()` eta `gazteakVector()` metodoak inplementatu behar ditu. Lehenengo bi metodoek, `Enumeration` motako objektu bat itzultzen dute eta hirugarrenak, ostera, `Vector` motakoa.

- `Vector` motako objektu bat itzultzen duen metodoak (`gazteakVector`) aztertuko du eraiki dugun pertsonen datu-basea eta 25 urte baino gehiagoko pertsonak beste `Vector` motako objektu berri batean biltegiratuz joango da. Datu-basearen azterketa bukatzen duenean, `Vector` motako objektu hasieratu berria itzultzen du emaitza gisa.
- `gazteak()` metodoak `gazteakVector()` metodoaren funtzionalitate bera inplementatzen du, baina kasu honetan, `Enumeration` motako objektu bat itzulita. Konturatu funtzionalitatea inplementatzeko estrategia hau erabiltzen badugu, nahitaez, `Enumeration` interfazearen `nextElement()` eta `hasMoreElements()` metodoak ere inplementatu behar direla.
- `lortuAdina()` metodoak, aurreko bi metodoen antzeko funtzionalitatea inplementatzen du baina, hemen, iteratzailearen bezeroari aukera ematen zaio adina adierazteko datu-basea aztertzen hasi baino lehen.

Hurrengo listatuak jasotzen du nola inplementatuko genituzkeen aitzinean aztertu ditugun hiru metodoak `Iteratzailea` interfazea inplementatzen duen `BiltegiIteraGazte` klasearen iturburu-kodean.

```
import java.util.*;
public class BiltegiIteraGazte implements Iteratzailea {

 Vector emaitza;
 BiltegiPertsonak nireBiltegiPertsonak;

 public BiltegiIteraGazte(BiltegiPertsonak pertsonak){

 nireBiltegiPertsonak= pertsonak;
 emaitza = new Vector();   }


public Vector gazteakVector() {
 for (int i=0; i < nireBiltegiPertsonak.Pertsonak.size();i++) {
 Pertsona per=
(Pertsona)nireBiltegiPertsonak.Pertsonak.elementAt(i);
 if(per.adina > 25) {emaitza.addElement(per);}
              } // for amaiera
 return emaitza;   }

public Enumeration gazteak() {
return new enumGazteak(this); }

public Enumeration lortuAdina(int adina) {

return new enumAdina(this,adina);  }  }
```

`enumGazteak` eta `enumAdina` klaseen iturburu-kodea honako hau izango da:

```
import java.util.*;
public class enumGazteak implements Enumeration {

 int pos=0;
 Vector emaitza;
 BiltegiIteraGazte nireBiltegi;

 public enumGazteak(BiltegiIteraGazte iterGazte) {

 nireBiltegi = iterGazte;
 emaitza= new Vector();

 for (int i=0; i <
nireBiltegi.nireBiltegiPertsonak.Pertsonak.size();i++) {
 Pertsona per=
(Pertsona)nireBiltegi.nireBiltegiPertsonak.Pertsonak.elementAt(i);
 if(per.adina > 25) {emaitza.addElement(per);}
   } // for amaiera
 } // enumGazteak bukaera

 public boolean hasMoreElements() {
     return (emaitza.size()>pos);  }
   public Object nextElement() {
     return emaitza.elementAt(pos++);  }}

//------------ ------------------ -----

import java.util.*;
public class enumAdina implements Enumeration {

 int pos=0;
 Vector emaitza;
 BiltegiIteraGazte nireBiltegi;
 int nireAdina;

 public enumAdina(BiltegiIteraGazte iterGazte, int adina) {

 nireBiltegi = iterGazte;
 nireAdina = adina;
 emaitza= new Vector();

 for (int i=0; i <
nireBiltegi.nireBiltegiPertsonak.Pertsonak.size();i++) {
 Pertsona per=
(Pertsona)nireBiltegi.nireBiltegiPertsonak.Pertsonak.elementAt(i);
 if(per.adina > nireAdina) {emaitza.addElement(per);}
   } // for amaiera
 } // enumAdina bukaera
```

```
    public boolean hasMoreElements() {
        return (emaitza.size()>pos);  }
        public Object nextElement() {
        return emaitza.elementAt(pos++);  }}
```

Azpiegitura guztia eraikita aplikazioaren klase nagusia programatuko dugu baliabide eratu berriak erabiltzeko:

```
   Iteratzailea iter = pertsonenBiltegia.elementuGazteak();
    Vector bektorea = iter.gazteakVector();

       for (int i=0; i < bektorea.size();i++) {
    Pertsona pertsona= (Pertsona)bektorea.elementAt(i);
  System.out.println("Bektore: Pertsonaren IZENA:
"+pertsona.izena+"   ADINA:"+pertsona.adina);
      } // for bukaera

    Enumeration e1 = iter.gazteak();
    System.out.println("Pertsonak: > 25 URTE--");
    while (e1.hasMoreElements()) {
                Pertsona pertsona=(Pertsona)e1.nextElement();
      System.out.println("Iteratzaile: Pertsonaren IZENA:
   "+pertsona.izena+"   ADINA:"+pertsona.adina);  }
     Enumeration e2 = iter.lortuAdina(35);

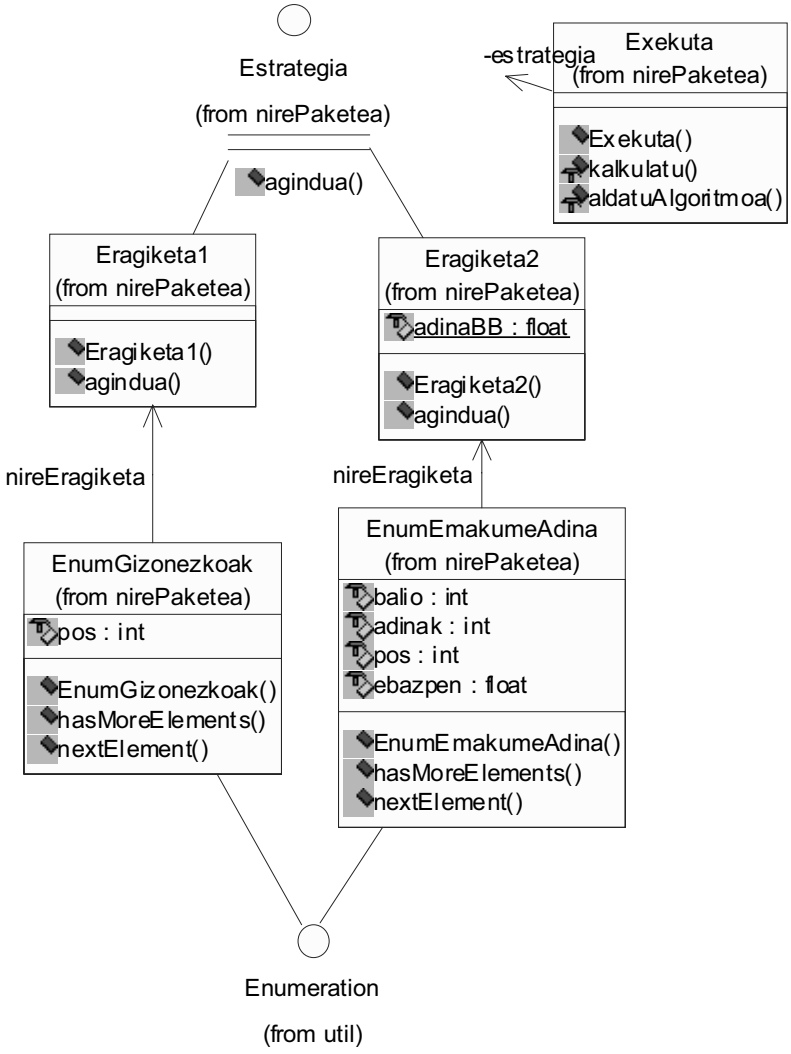
    System.out.println("Pertsonak: > 35 URTE--");
    while (e2.hasMoreElements()) {
         Pertsona pertsona=(Pertsona)e2.nextElement();
        System.out.println("BigarrenIteratzailea: Pertsonaren IZENA:
"+pertsona.izena+"   ADINA:"+pertsona.adina);  }
```

Konturatu klase nagusiarentzat gardena dela nola egituratzen diren `Enumeration` motako objektu biak eta `Vector` motako objektua. Bektorearekin, klase nagusiak `Vector` klasearen `size()` eta `elementAt()` metodoak erabiliko ditu bektorearen elementu kopurua eskuratzeko eta elementuak atzitzeko. Gauza bera egingo du `Enumeration` motako objektuekin baina, oraingoan, `hasMoreElements()` eta `nextElement()` metodoak egikarituta.

7.9. ADIBIDE KONPLEXUAK: STRATEGY

Hurrengo adibide honetan garatuko dugun aplikazio informatikoak Iterator patroiaren adibidean landu dugun datu-base txikia berreskuratuko du adibidearen zehaztasunak lantzeko. Adibidean algoritmo familia bat egituratu nahi dugu, eta horretarako, Strategy patroia erabiltzea erabaki dugu. Ikuspegi hori erabilita, egikarituko diren algoritmoak kapsulatu egingo ditugu eta exekuzio-garaian, algoritmoak trukatu egingo dira eta unean uneko algoritmoaren bertsio egokiena eskuratu eta egikarituko da.

Strategy patroia aurkeztean ikusi dugu patroiak interfaze bat erazagutzen duela `Strategy` motako objektu kanonikoak eraikitzeko erabiliko dena. Gure adibidean, interfaze horrek `Estrategia` izena hartuko du eta `Eragiketa1` eta `Eragiketa2` klaseak izango dira interfazeari inplementazioa erantsiko diotenak. **7.34. irudia**k aplikazio informatikoak erabiliko duen Klase Diagrama jasotzen du.



7.34. irudia. Aplikazioaren Klase Diagrama.

Ondoko listatuak jasotzen du `Estrategia` interfazearen eta `Eragiketa1` eta `Eragiketa2` klaseen iturburu-kodea.

```
package nirePaketea;
import java.util.*;
interface Estrategia {
Enumeration agindua();  }
//-----------------------
package nirePaketea;
import java.util.*;
class Eragiketa1 implements Estrategia {
 Vector nirePertsonak;

 public Eragiketa1 (Vector pertsonak){
 nirePertsonak= pertsonak;}

 public Enumeration agindua() {
 return new EnumGizonezkoak(this);  }}
//-----------------------
package nirePaketea;
import java.util.*;
class Eragiketa2 implements Estrategia {
 Vector nirePertsonak;
 static float adinaBB;
 public Eragiketa2 (Vector pertsonak){
 nirePertsonak= pertsonak;}

    public Enumeration agindua() {
    return new EnumEmakumeAdina(this);  }}
```

Bai `Eragiketa1` motako objektu bat eta bai `Eragiketa2` motako bat eraikitzeko erabiliko ditugun metodo eraikitzaileek hasieran aipatutako datu-base txikia erabiliko dute instantzia bat eraiki eta instantzia horretako `Vector` motako objektua hasieratzeko. Hori eginez gero, klase bakoitzak, `Estrategia` interfazean definitu dugun `agindua()` metodoaren inplementazio propioa erantsiko dio metodoari.

Listatuan ikusten den legez, `agindua()` metodoak delegatu egiten dio algoritmoaren benetako exekuzioa `Enumxxx` motako objektu bati. `Enumxxx` motako objektu bakoitzak `Enumeration` interfazea inplementatzen du, eta horrela, `agindua()` metodoak `Enumeration` motako objektu bat itzultzen dio deitzen dion bezeroari. Alegia, `Enumxxx` motako objektuek inplementatuko dute benetan algoritmo partikular bakoitzaren portaera. Hona hemen bakoitzaren inplementazioa:

```
package nirePaketea;
import java.util.*;
class EnumGizonezkoak implements Enumeration {
 Eragiketa1 nireEragiketa;
 Vector emaitza;
  int pos=0;
```

```
 public EnumGizonezkoak(Eragiketa1 eragiketa){
 nireEragiketa = eragiketa;
 emaitza= new Vector();
 for (int i=0; i < nireEragiketa.nirePertsonak.size();i++) {
 Pertsona per=
(Pertsona)nireEragiketa.nirePertsonak.elementAt(i);
 if(per.sexua.equals("G")) {emaitza.addElement(per);}  }   }

public boolean hasMoreElements() {
  return(emaitza.size()>pos);}

  public Object nextElement() {
  return emaitza.elementAt(pos++);} }
//-----------------------
//package nirePaketea;
import java.util.*;
class EnumEmakumeAdina implements Enumeration {
 Eragiketa2 nireEragiketa;
 Vector emaitza;
 int balio;
 int adinak;;
   int pos =0;
   float ebazpen;
 public EnumEmakumeAdina(Eragiketa2 eragiketa){
 nireEragiketa = eragiketa;
 emaitza= new Vector();
  int ind=1;
 for (int i=0; i < nireEragiketa.nirePertsonak.size();i++) {
 Pertsona per=
(Pertsona)nireEragiketa.nirePertsonak.elementAt(i);
 if(per.sexua.equals("E")) {
 adinak += per.adina;
  System.out.println("Emakumezkoen adinak: "+adinak);
 emaitza.addElement(per);
  nireEragiketa.adinaBB = adinak/ind;
  System.out.println("Emakumezkoen BATEZBESTE:
"+nireEragiketa.adinaBB);
  ind=ind+1;  } } }

  public boolean hasMoreElements() {
  return(emaitza.size()>pos);}

     public Object nextElement() {
     return emaitza.elementAt(pos++);}  }
```

Lehenengo klasean (`EnumGizonezkoak`), datu-basearen pertsonen mota aztertzen da eta, gizonezkoak direnak, beste `Vector` motako objektu berri batean biltegiratzen dira. Gero, klase nagusian `hasMoreElements()` eta `nextElement()` metodoak erabiliko dira balio berriak atzitu eta eskuratzeko.

`EnumEmakumeAdina` klasean antzeko funtzionalitatea inplementatuko da, hemen ere datu-basearen pertsona mota aztertzen da eta emakumezkoak direnak beste `Vector` motako objektu berri batean biltegiratzen dira. Hemen gainera, aldiko, emakumeen batez besteko adina kalkulatzen da begiztaren exekuzio-ziklo bakoitzean.

Strategy patroiaren osagai guztiekin bukatzeko, patroiaren Context rol kanonikoa jokatuko duen klasea diseinatu behar dugu. `Exekuta` klaseak aipatutako rola beteko digu gure aplikazio informatikoan.

```
package nirePaketea;
import java.util.*;
public class Exekuta {

 private Estrategia estrategia;
 public Exekuta(Estrategia estra){
 estrategia = estra;}
 Enumeration kalkulatu() {
 return estrategia.agindua();}

    void aldatuAlgoritmoa(Estrategia estraBerri){
    estrategia = estraBerri;   } }
```

`Exekuta` klaseak bi metodo nagusi izango ditu: `aldatuAlgoritmoa()` metodoak estrategia berriak esleitzeko balioko digu exekuzio-garaian eta `kalkulatu()` metodoak, ostera, `Estrategia` motako objektu egokiaren `agindua()` metodoa egikarituko du, hau da, algoritmoaren portaera partikularra bereizteko balioko digu.

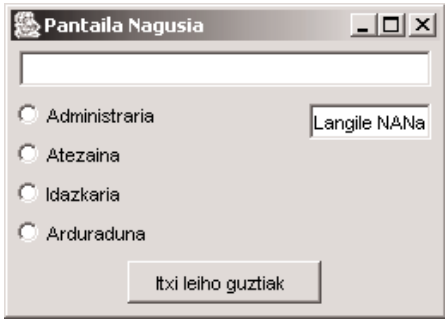
Dena prest dugunean, aplikazioa eraikitzen dugu `Nagusi` klasea programatuta. Klase horrek bizpahiru lerrotan egituratuko du aplikazioaren funtzionalitate guztia Strategy patroia erabilita, eta gero geldituko zaion gauza bakarra izango da eskuratutako emaitza guztiak irteera estandarretik ateratzea.

```
Exekuta ex = new Exekuta(new
Eragiketa1(pertsonenBiltegia.Pertsonak));
 Enumeration enum = ex.kalkulatu();
 ex.aldatuAlgoritmoa(new
Eragiketa2(pertsonenBiltegia.Pertsonak));
 Enumeration enum1 = ex.kalkulatu();
 float adinaBB = Eragiketa2.adinaBB;

  while(enum.hasMoreElements()){
  Pertsona pertsona = (Pertsona)enum.nextElement();
  System.out.println("Gizonezkoak: "+pertsona.izena+"
sexua= "+pertsona.sexua);}
```

```
      while(enum1.hasMoreElements()){
      Pertsona pertsona = (Pertsona)enum1.nextElement();
      System.out.println("Emakumezkoak: "+pertsona.izena+"    sexua=
   "+pertsona.sexua+
"     adina= "+pertsona.adina);}
 System.out.println("Emakumezkoen batezbesteko adina: "+adinaBB);
```

7.10. ADIBIDE KONPLEXUAK: OBSERVER



7.35. irudia. Aplikazioaren hasierako interfaze grafikoa.

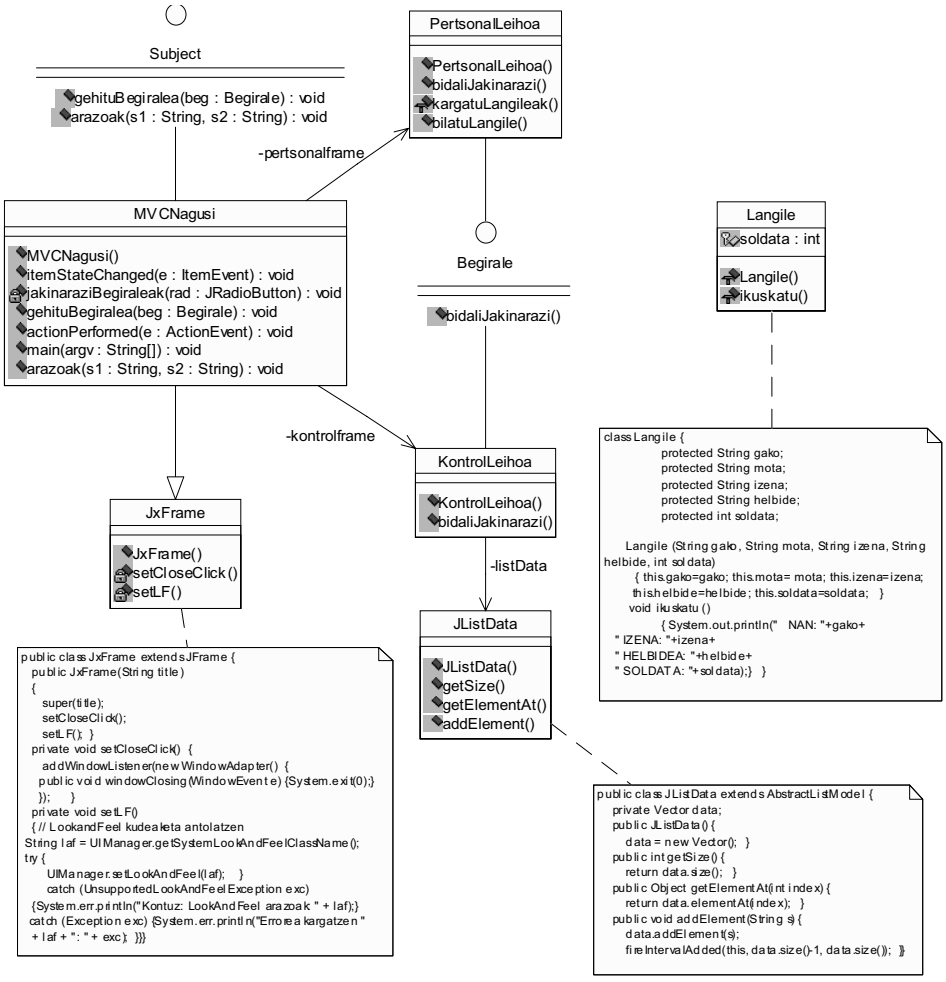
Aplikazio informatiko bat eraiki nahi dugu enpresa bateko pertsonalaren sarrerak kontrolatuko dituena. Aplikazioak **7.35. irudia**n agertzen den interfaze grafiko nagusia izango du. Enpresako pertsonala lanera sartzen denean pantaila nagusi hori erabiliko du bere NA-a eta mota adierazteko. Dena ondo badoa sistemak pertsona identifikatuko du eta zain geldituko da beste pertsona bat etorri eta eragiketa bera errepikatu arte. Pantaila nagusia enpresaren atarian kokatu dugun ordenagailuaren[67] monitorean agertuko da egunero. Bestalde, pertsonal-arduradunak eskatu digu enpresako sarrerak monitorizatuta eduki nahi dituela bere bulegoan duen ordenagailuaren pantailan eta antzeko gauza eman digu aditzera datu-basearen zerbitzariaren ardura duen informatikariak ere. Aplikazioa prototipo mailan eraikiko dugunez, bai batari eta bai besteari, interfaze grafiko (**7.38.** eta **7.39. irudia**k) xume bat prestatuko diegu. Beraz, aplikazioak hiru interfaze grafiko desberdin izango ditu eta bakoitzak bere funtzionalitatea izango du.

Aplikazioaren betekizunak aztertu ondoren erabaki dugu Observer patroia primeran datorkigula betekizunen funtzionalitate guztiak kudeatzeko.

Erabakiz gero aplikazio informatikoan Observer patroiaren arkitektura inplementatuko dugula, patroiaren parte-hartzaileak identifikatu eta diseinatuko ditugu. Begirale motako objektu kanonikoak identifikatzeko, ez dugu asago joan

67. Benetan, adibidean inplementatuko dugun aplikazioa ordenagailu bakarrean egikarituko da. Benetako sistema banatu bat inplementatu nahiko bagenu, RMI (Remote Method Invocation) eta antzeko saltsetan sartu beharko ginateke eta ez da hori adibidearen helburua.

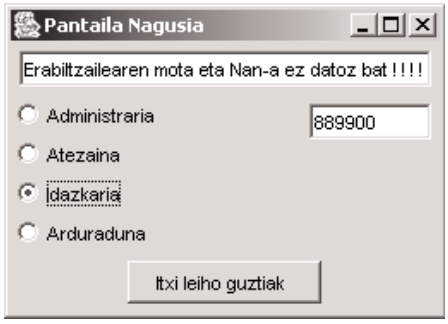
beharrik, pertsonal-arduradunaren ordenagailuan kokatuko dugu bat eta datubasearen zerbitzarian beste bat. `Subject` motako objektu kanonikoa ere paraje dago pertsonalarentzat, interfaze grafiko nagusia kontrolatuko duen objektua izango baita. Objektu kanonikoei izena jarri eta instantziatu ondoren, **7.36. irudia**n agertzen den hasierako Klase Diagrama erantsi diegu.



7.36. irudia. Aplikazioaren Klase Diagrama.

Klase Diagraman ikusten den legez, `Subject` motako objektua eraikitzeko `MVCNagusi` klasea erabiliko dugu. Klaseak, hiru metodo nagusi izango ditu bere begiraleak kudeatzeko: `jakinaraziBegiraleak()`, `gehituBegiraleak()` eta `arazoak()`.

- `jakinaraziBegiraleak()` metodoa, gertaera berri bat gertatzen denean interfaze grafiko nagusiaren `JRadioButton` motako objektu batean egikaritzen da. Metodoak bere begiraleei jakinarazten die gertaera berria begirale bakoitzaren `bidaliJakinarazi()` metodoa egikarituta. Konturatu begirale bakoitzak, gure kasuan gertatuko den bezala, `bidaliJakinarazi()` metodoaren inplementazio partikularra izango duela, zentzuzkoa baita begirale bakoitzaren erantzuna desberdina izatea gertaera beraren aurrean.
- Jakin badakigu `Subject` motako objektuak bere begiraleak identifikatuta eta kontrolpean izan behar dituela egoera-aldaketak jakinarazteko, haatik, objektuak metodoak izan behar ditu begiraleak biltegiratzeko eta nahi izanez gero ezabatzeko. `gehituBegiralea()` metodoak hori egingo du: begirale berri bat gehitu `Subject` motako objektuaren begiraleen `Vector` motako objektu-biltegira.
- Datu-basearen zerbitzariak arazorik badu pertsona aurkitzeko interfaze grafiko nagusian sartu ditugun mota eta NA-a erabilita, `Subject` motako objektuari ohartaraziko dio `arazoak()` metodoa egikarituta. `Subject` motako objektua ohartuko da arazo bat gertatu dela (**7.37. irudia**) eta berehala jakinaraziko die bere begirale guztiei. Jakinarazteko erabiliko den estrategia `jakinaraziBegiraleak()` metodoak aplikatu duen bera izango da.



7.37. irudia. Erabiltzailearen NA-a eta mota gaizki sartu dira sisteman.

Hona hemen `Subject` objektuaren rola jokatuko duen `MVCNagusi` klasearen iturburu-kodea:

```
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.text.*;
import javax.swing.border.*;
import javax.accessibility.*;
```

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.io.*;
import java.applet.*;
import java.net.*;

public class MVCNagusi extends JxFrame
implements ActionListener, ItemListener, Subject {
    private JButton Itxi;
    private JTextField NanEremua, arazoak;
    private JRadioButton adminis, atezain, idazkari, ardura;
    private Vector begiraleak;
    private Vector langileak = new Vector();
    private KontrolLeihoa kontrolframe;
    private PertsonalLeihoa pertsonalframe;
//-----------------------------------------
    public MVCNagusi() throws FileNotFoundException, IOException{
        super("Pantaila Nagusia");
        begiraleak = new Vector();   // Begiraleen zerrenda
 langileak = new Vector();

 JPanel p = new JPanel(true); // Panel-a gehitu ContentPane bati
        p.setLayout(new BorderLayout());
        getContentPane().add("Center", p);

 JPanel p3 = new JPanel();
   p.add("North", p3);
        p3.add( arazoak=new JTextField(30));

        Box box = new Box(BoxLayout.Y_AXIS);
        p.add("Center", box);
        box.add(adminis = new JRadioButton("Administraria"));
        box.add(atezain = new JRadioButton("Atezaina"));
   box.add(idazkari = new JRadioButton("Idazkaria"));
        box.add(ardura = new JRadioButton("Arduraduna"));

 idazkari.addItemListener(this); //Osagaien entzuleak prestatzen
        adminis.addItemListener(this);
        atezain.addItemListener(this);
        ardura.addItemListener(this);

        // Botoi guztiak taldekatu
        ButtonGroup bgr = new ButtonGroup();
        bgr.add(adminis);
        bgr.add(atezain);
        bgr.add(idazkari);
        bgr.add(ardura);

        JPanel p2 = new JPanel();
```

```
        p.add("East", p2);
        p2.add( NanEremua =new JTextField("Langile NANa"));

 // ITXI botoia markoaren bukaeran prestatzen dugu
        JPanel p1 = new JPanel();
        p.add("South", p1);
        p1.add( Itxi =new JButton("Itxi leiho guztiak "));
        Itxi.addActionListener(this);     // Entzuleak prestatu
        setBounds(200, 200, 400, 300);
        pack();

 //---------Begiraleak  eraiki---------
        kontrolframe = new KontrolLeihoa(this);
        pertsonalframe = new PertsonalLeihoa(this);
        setVisible(true);  }

 //--------------------------------------
    public void itemStateChanged(ItemEvent e) {
        // Gertaerak tratatzen
        if (e.getStateChange() == ItemEvent.SELECTED) {

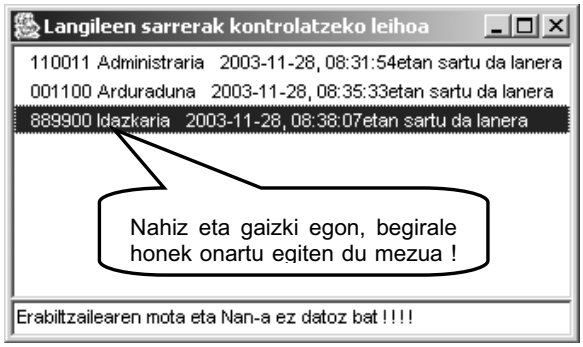
         jakinaraziBegiraleak((JRadioButton)e.getSource());  }}

    //---------------------------------
public void arazoak(String mota, String nan) {
      System.out.println("MENU NAGUSIRA ITZULI NAIZ!!!!!");
String mezua="Erabiltzailearen mota eta Nan-a ez datoz bat
! ! ! !";
      arazoak.setText(mezua);
      jakinaraziBegiraleak1(mezua);  }
//--------------------------------------
    private void jakinaraziBegiraleak1(String mezua){
 for (int i=0; i< begiraleak.size(); i++) {
     ((Begirale)(begiraleak.elementAt(i))).bidaliJakinarazi1(
mezua); }}
    //--------------------------------------
    private void jakinaraziBegiraleak(JRadioButton rad) {
        // Begiraleei jakinarazi aldaketa eman dela
        String langile = rad.getText();
 String LangileNana= NanEremua.getText();
        for (int i=0; i< begiraleak.size(); i++) {

((Begirale)(begiraleak.elementAt(i))).bidaliJakinarazi(langile,Lan
gileNana);}}
    //--------------------------------------
    public void gehituBegiralea(Begirale beg) {
        // Gehitu begirale bat zerrendara
        begiraleak.addElement(beg);   }
    //--------------------------------------
    public void actionPerformed(ActionEvent e) {
        // Itxi botoiaren erantzuna
```

```
            Object obj = e.getSource();
            if (obj == Itxi)
                System.exit(0);   }
        //-------------------------------------
         static public void main(String[] argv)throws
FileNotFoundException,IOException {
           new MVCNagusi();   }}
```

Pertsonal-arduradunaren ordenagailuan kokatuko dugun `Begirale` motako objektuaren diseinuak ez du zailtasun handirik, objektu horrek ez baitu ia ezer egiten aplikazioaren funtzionalitatea handitzeko. Horregatik, haren `bidaliJakinarazi()` metodoa xumea eta arrunta izango da.



7.38. irudia. Pertsonal-arduradunaren interfaze grafikoa.

Hona hemen pertsonal-arduradunaren pantaila kudeatuko duen `KontrolLeihoa` klasearen iturburu-kodea:

```
public class KontrolLeihoa extends JFrame
implements Begirale {
    private JList list;
    private JPanel p;
    private JScrollPane lsp;
    private JListData listData;
    private Subject nirea;
    private TextField  testua= new TextField(25);

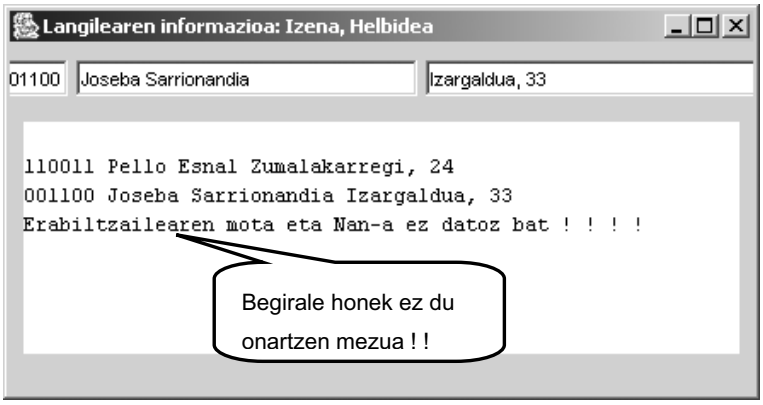
  public KontrolLeihoa(Subject s) {
        super("Langileen sarrerak kontrolatzeko leihoa");
        // Subject objektuaren erreferentzia gorde
        nirea = s;
        // Txertatu panel-a markoaren barnean
        p = new JPanel(true);
        getContentPane().add("Center",  p);
        getContentPane().add("South",testua);
        p.setLayout(new BorderLayout());
        // Subject objektua ohartarazi
        s.gehituBegiralea(this);
```

```
        // Eraiki zerrenda
        listData = new JListData();  // Zerrendaren eredua
eraiki
        list = new JList(listData);
        lsp = new JScrollPane();
        lsp.getViewport().add(list);
        p.add("Center", lsp);

        lsp.setPreferredSize(new Dimension(200,300));
        setBounds(450, 200, 400, 200);
        setVisible(true);  }
    //-----------------------------
    public void bidaliJakinarazi(String s, String Nana) {
        Date gaur = new Date();
 String patroia = " yyyy-MM-dd, HH:mm:ss";
 SimpleDateFormat formato = new SimpleDateFormat(patroia);
 String irteera = formato.format(gaur);
 listData.addElement("   "+Nana+" "+s+"  "+irteera+"etan sartu
da lanera"); }
 //-----------------------------
 public void bidaliJakinarazi1(String mezua) {

 String nireMezua = mezua;
 testua.setText(nireMezua);  }}
```

`KontrolLeihoa` motako objektuaren metodo eraikitzaileak `JFrame` motako interfaze grafikoa eraikitzen dio pertsonal-arduradunari enpresako pertsonalaren sarrerak monitorizatzeko. Metodo eraikitzailean halaber, `Subject` motako objektuaren erreferentzia gordetzen da eta `Subject` motako objektuari ohartarazten zaio begirale berri bat eraiki dela eta gehitu egin behar duela begiraleen biltegian.



7.39. irudia. Datu-basearen zerbitzariaren arduradunaren interfaze grafikoa.

Hona hemen datu-basearen zerbitzariaren pantaila kudeatuko duen `PertsonalLeihoa` klasearen iturburu-kodea:

```
public class PertsonalLeihoa extends JFrame
implements Begirale {

    private Subject nirea;
    private Font font;
    Vector langileak = new Vector();
    JTextField NANa, Izena, Helbidea;
    JTextArea LangileInformazioa = new JTextArea(8, 60);

    private JPanel p = new JPanel(true);

    public PertsonalLeihoa(Subject s) throws
FileNotFoundException, IOException{
        super("Langilearen informazioa: Izena, Helbidea");
 kargatuLangileak();
 nirea = s;

 JPanel p1 = new JPanel();
       p.add("Center", p1);
       p1.add( NANa =new JTextField("Langilearen NANa"));
       p1.add( Izena =new JTextField("Langilearen Izena",25));
 p1.add( Helbidea =new JTextField("Langilearen Helbidea",25));


 JPanel p2 = new JPanel();
p.add("South", p2);
 p2.add(LangileInformazioa);

       getContentPane().add("Center",  p);
       s.gehituBegiralea(this);
       font = new Font("Sans Serif", Font.BOLD, 14);
 setBounds(100, 400, 900, 200);
       setVisible(true);  }
   public void bidaliJakinarazi(String s, String Nana) {

 String labur= s.substring(0,2);
 System.out.println("LANGILE MOTA: "+labur);
 Langile langilea= this.bilatuLangile(labur, Nana);

 if (langilea == null){
   nirea.arazoak(s, Nana);
 System.out.println("ARAZOAK HEMEN!!!!!");}

 String Gakoa = langilea.gako;
 NANa.setText(Gakoa);
 String izena = langilea.izena;
 Izena.setText(izena);
 String helbidea = langilea.helbide;
 Helbidea.setText(helbidea);
```

```
LangileInformazioa.setText(LangileInformazioa.getText()+"\n
"+Gakoa+" "+izena+" "+helbidea);  }

public void bidaliJakinarazi1(String mezua) {
  String nireMezua = mezua;
LangileInformazioa.setText(LangileInformazioa.getText()+"\n"+
nireMezua); }

  void kargatuLangileak()
        throws FileNotFoundException, IOException {

        String FITXATEGI_AMAIERA=null;
        BufferedReader sarrera =
            new BufferedReader(new FileReader("langileak.txt"));

       String lerro, izena, nan, mota, helbide;
        int soldata;
           while ((lerro=sarrera.readLine())!=FITXATEGI_AMAIERA)
           {  StringTokenizer st=new StringTokenizer(lerro,"/");

        nan = st.nextToken();
       mota= st.nextToken();
        izena =  st.nextToken();
        helbide = st.nextToken();
        soldata= Integer.valueOf(st.nextToken()).intValue();

    Langile lan = new Langile(nan,mota,izena, helbide, soldata);
              System.out.println("LANGILE NAN-a:"+nan+
"\n       IZENA:"+izena+
"\n       HELBIDE:"+helbide+
"\n       SOLDATA:"+soldata);
          langileak.addElement(lan);  }}

public Langile bilatuLangile(String mota1, String nan) {
          for (int i=0;i<langileak.size();i++)

if(((Langile)(langileak.elementAt(i))).gako.equals(nan)
        && ((Langile)(langileak.elementAt(i))).mota.equals(mota1))
               return (Langile)(langileak.elementAt(i));
          return null;  }}
```

langileak.txt

```
110011/Ad/Pello Esnal/Zumalakarregi, 24/2000
001100/Ar/Joseba Sarrionandia/Izargaldua, 33/1500
111000/Id/Irene Lamarka/Lasarte Karrika, 26/2500
222000/Id/Pedro Sloterdijk/Normandia Kalea, 25/1200
111111/At/Juantxo Txapapote/Marea Beltz, 40/2000
000222/Ad/Miren Amalur/Haizeorratz, 80/1000
220022/Ad/Amaia Gaztelugatxe/Mendebaldea, 69/2000
002200/At/Il Yeol Song/Tao the King, 45/1000
333000/Ar/Kyu Young Whang/Taejeon, 30/800
```

7.40. irudia. Pertsonalaren datu pertsonalak.

Datu-basearen zerbitzarian kokatuko dugun bigarren begiraleak enpresako pertsonalaren informazioa kudeatuko du. Horretarako, metodo eraikitzaileak `kargatuLangileak()` metodoa egikarituko du `txt` (**7.40. irudia**) motako fitxategi batetik pertsonalaren informazioa eskuratzeko. Fitxategiaren informazioa `Vector` motako objektu batean biltegiratuko da. Bektorea `Langile` motako objektuz eratuko dugu.

Bestalde, `KontrolLeihoa` motako objektuak bezala, `PertsonalLeihoa` motako objektuak `bidaliJakinarazi()` metodoa inplementatuta izan behar du `Subject` motako objektuaren mezuak tratatzeko. Konturatu metodo horretan `bilatuLangile()` metodoa egikaritzen dela aztertzeko interfaze nagusian idatzi diren datuak egokiak direla. Metodoak ez badu inor aurkitzen ezaugarri horiekin, `null` balioa itzultzen du eta `Subject` motako objektua ohartarazten du `arazoak()` metodoa egikarituta. `Subject` motako objektuak bere begirale guztiei ohartarazten die enpresako atarian dagoen pertsonak ez duela informazio egokia sartu interfaze grafiko nagusian.

7.11. ADIBIDE KONPLEXUAK: OBSERVER + ITERATOR

Telefoniako operadore batek zerbitzu berri bat eskaini nahi du bezeroek mezuak trukatzeko erabil dezaten. *Deien Kontrolatzaile* zerbitzuak bezeroak aurretik aukeratu dituen dispositiboetara (bilatzailea, posta elektronikoa, telefono finkoa edo telefono mugikorra) bidaltzen ditu deiak, bezeroaren dispositiboak dei bat jasotzen duenean. Mezuan deia egin duen dispositiboaren identifikazioa adierazten da.

Dispositiboak `izenpetu(dispositiboa)` metodoa erabiliko du beste dispositiboei ohartarazteko zeintzuk diren bere ardurapeko dispositiboak. Halaber, `KenduIzenpe(dispositiboa)` metodoarekin baja eman dezake dispositiboak

Aplikazio informatikoaren klase nagusiak honako informazio-fluxu hau egikarituko du:

```
TelefonoFinko tfinko=new TelefonoFinko("943015052");

TelefonoMugikor tmugikor=new TelefonoMugikor("Nire mugikor-1");
TelefonoMugikor tmugikor2=new TelefonoMugikor("Nire mugikor-2");

PostaElektroniko posta=new PostaElektroniko("iso@si.ehu.es");

Bilatzaile bila=new Bilatzaile("Jaxintoren bilatzailea");

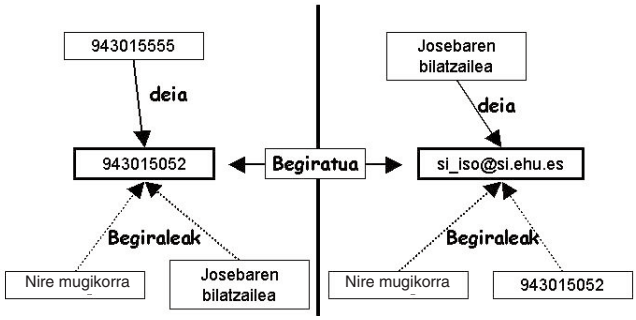
tfinko.izenpetu(tmugikor);
tfinko.izenpetu(bila);
tfinko.deia(new TelefonoFinko("943015555"));
System.out.println("----LEHEN DEIA --");

tfinko.KenduIzenpe(bila);
tfinko.deia(posta);
System.out.println("----BIGARREN DEIA --");

posta.izenpetu(tmugikor);
posta.izenpetu(tfinko);
posta.deia(bila);
System.out.println("----HIRUGARREN DEIA --");

 Enumeration e=tmugikor.getZenbakiak();
while (e.hasMoreElements()) {
       String s=(String)e.nextElement();
                    System.out.println(s);
```

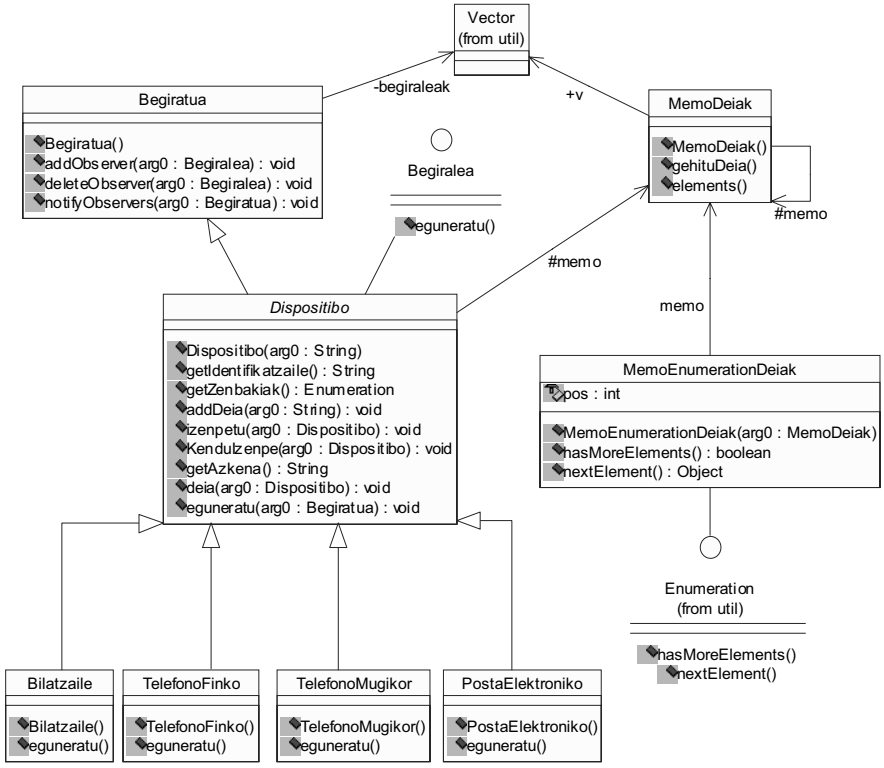
Dispositibo batek beste bati deia luzatzen diola adierazteko, `deia()` metodoa programatu behar da. Informazio-fluxuan agertzen den adibidean, 943015555 zenbakia duen telefono finkoak dei bat luzatzen dio 943015052 telefono finkoari. Deia jasoz gero, telefono finkoak bere ardurapean hartu dituen dispositiboei jakinarazten die deiaren berri. **7.41. irudia**k aipatutako adibidea islatzen du grafikoki. Deiak jasotzen ari diren bitartean posible da dispositibo bakoitzak bere ardurapean dituen beste dispositiboak kentzea edo berriak eranstea.



7.41. irudia. Dispositiboen arteko deiak.

Telefoniako operadoreak gainera, azterketa bat egin du eta modulu berri bat eraiki nahi du goian aipatu dugun zerbitzua osatzeko. Modulu horrek deiak gordetzeko balioko du eta horretarako `addDeia()` eta `getZenbakiak()` metodoak erabiliko ditu. Lan hau gauzatzeko kontuan izan behar da moduluaren bezeroari ez zaiola moduluaren inplementazioaren zehaztasunik eman nahi.

Aplikazioaren betekizunak aztertuta, konturatzen gara aplikazioak bi modulu nagusi dituela eta bakoitza diseinatzeko patroi egoki bat dugula. Observer patroiarekin dispositiboak jasotzen dituen deiak eta dei horien banaketa erraz prozesa daitezke. Iterator patroiak, ostera, deiak gorde eta kudeatzeko balioko digu. Patroi biak konbinatzen baditugu aplikazio informatikoan **7.42. irudia**n agertzen den Klase Diagrama sortzen zaigu. Konturatu Klase Diagraman bitxitasun bat gertatzen dela Observer patroiarekin: dispositibo bakoitza begirale bat izan daiteke beste dispositibo batzuentzat eta, aldi berean, patroiaren `Subject` rola ere joka dezake `Begiratua` klasearen funtzionalitatea eratorrita. Horregatik, Klase Diagraman ikusten den legez, `Dispositibo` klaseak `Subject` motako objektuaren funtzionalitatea izango du, `addObserver()`, `deleteObserver()` eta `notifyObservers()` metodoak `Begiratua` klasetik eratorrita eta, aldi berean, patroiaren `Observer` rola ere jokatuko du `Begiralea` klasea inplementatuta.



7.42. irudia. Aplikazioaren Klase Diagrama.

Hona hemen, `Begiratua` eta `Dispositibo` klaseen eta `Begiralea` interfazearen iturburu-kodea:

```
import java.util.*;
public class Begiratua
{
 private Vector begiraleak = new Vector();
      public Begiratua() {};

 public void addObserver (Begiralea disp) {
begiraleak.addElement(disp);}

 public void deleteObserver (Begiralea disp) {
begiraleak.removeElement(disp);}


 public void notifyObservers(Begiratua disp)  {
 Begiratua beg = disp;
 for (int i=0; i < begiraleak.size();i++) {
((Dispositibo)(begiraleak.elementAt(i))).eguneratu(beg);}} }

//-------------------------

import java.util.*;
public interface Begiralea
{  public void eguneratu(Begiratua t); }

//-------------------------

import java.util.*;
public abstract class Dispositibo extends Begiratua implements
Begiralea
{   protected MemoDeiak memo;
  String azkenzenbaki;

  String identifikatzaile;
  public Dispositibo(String pIdent)
  { identifikatzaile=pIdent;
    memo=new MemoDeiak();}
  public String getIdentifikatzaile() {return identifikatzaile;}
  public Enumeration getZenbakiak() {
   return memo.elements();  }
  public void addDeia(String pZenbaki) {
   memo.gehituDeia (pZenbaki);  }
  public void izenpetu(Dispositibo pDisp) {
   addObserver(pDisp);  }
  public void KenduIzenpe(Dispositibo pDisp) {
   deleteObserver(pDisp); }
```

```
  public String getAzkena(){
   return azkenzenbaki;  }
  public void deia(Dispositibo disp) {
      azkenzenbaki=disp.getIdentifikatzaile();
         notifyObservers(this); }}
```

Iterator patroia egikaritzeko `Vector` motako objektu bat eta `Enumeration` interfazea erabiliko ditugu. Moduluen arteko komunikazioa `MemoDeiak` motako objektu bat erabilita gauzatuko da. `Enumeration` motako objektua eskuratzeko, `MemoDeiak` motako objektuak `elements()` metodo ezaguna inplementatuko du. Metodo horrek, aldi berean, `MemoEnumerationDeiak` motako objektu bat eraikitzen du `Enumeration` motako instantzia bat eskuratzeko.

Hona hemen `MemoDeiak` eta `MemoEnumerationDeiak` klaseen iturburu-kodea:

```
import java.util.*;
public class MemoDeiak
{
  public Vector v;
  protected MemoDeiak memo;
  public MemoDeiak()
  {   v=new Vector();  }
  public void gehituDeia(String s) {
   v.addElement(s);  }
  public Enumeration elements() {
   return new MemoEnumerationDeiak(this);  }}

//-------------------------

import java.util.*;
public class MemoEnumerationDeiak implements Enumeration
{
  MemoDeiak memo;
  int pos=0;
  public MemoEnumerationDeiak(MemoDeiak m)
  {   memo=m;  }
  public boolean hasMoreElements() {
   return (memo.v.size()>pos);  }
     public Object nextElement() {
return memo.v.elementAt(pos++);  }}
```

Dispositibo zehatz bakoitza eraikitzeko nahikoa da `Dispositibo` klasearen metodo eraikitzaileari dei egitea, deian dispositiboaren identifikadorea erantsita. Metodo eraikitzaileak, identifikadorea biltegiratuko du eta `MemoDeiak` motako objektu bat eraikiko dio dispositiboari deiak gordetzen joateko. Hurrengo listatuak `TelefonoFinko` klasearen iturburu-kodea islatzen du. `Bilatzaile`,

`TelefonoMugikor` eta `PostaElektroniko` klaseek `TelefonoFinko` klasearen egitura bera dute.

```
    import java.util.*;
    public class TelefonoFinko extends Dispositibo
    {  String zenbaki;

       public TelefonoFinko(String pZenbaki)
       {   super(pZenbaki);  }

       public void eguneratu(Begiratua t){
         Dispositibo disp=(Dispositibo)t;
     System.out.println("Mezua finkora —"+identifikatzaile+ "-jaso
duen dispositiboa "+disp.getIdentifikatzaile()+"  bidali duena
"+disp.getAzkena());
     memo.gehituDeia(disp.getAzkena());  }}
```

7.12. ADIBIDE KONPLEXUAK: STRATEGY + FACTORYMETHOD [+ JGL]

Java egitura batean pertsona multzo bat dugu. Egitura horren gainean aplikazio bat eraiki nahi dugu honako betekizun hauek izango dituena:

1. Kontsulta bat definitu nahi da egituraren gainean. Kontsulta bakoitzak elementu kopuru bat itzuliko du emaitza gisa, kontsulta horren baldintzak betetzen dituena. Adibidez, kontsultak, honelakoak izan daitezke:

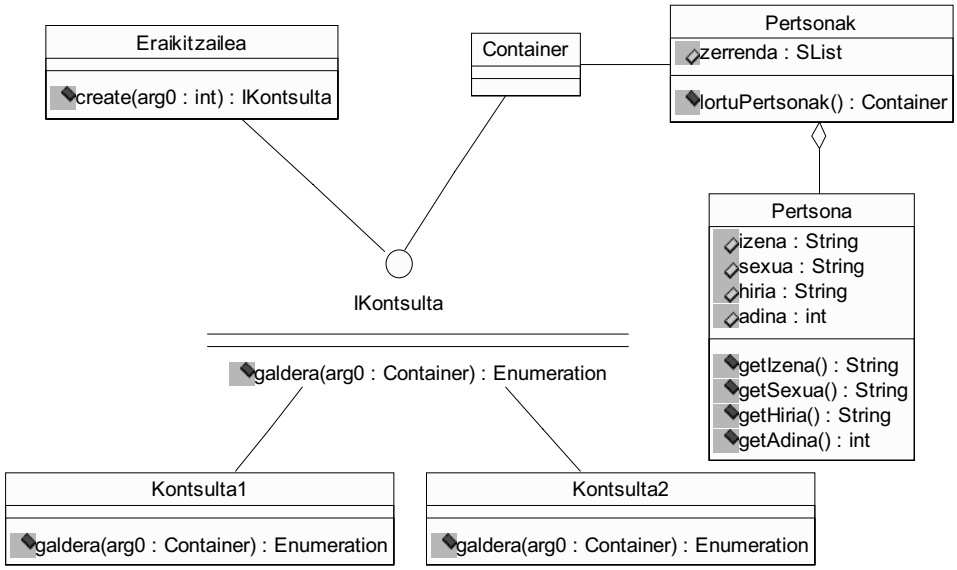
   K1. Eskuratu 18 urte baino gehiago dituzten eta gizonak diren pertsonak, izenaren arabera sailkatuta.

   K2. Eskuratu Donostian bizi diren eta emakumeak diren pertsonak.

2. Aplikazioak kontsulta berriak txertatzen utzi behar du dinamikoki.

3. Klase-erabiltzaileari kontsultak inplementatzeko erabiltzen diren klaseak ezkutatu egin behar zaizkio. Hau da, klase-erabiltzaileak egin nahi duen kontsulta (k1, k2, k3...) zenbaki oso batez (1, 2, 3...) finkatuko du eta zenbakiaren balioaren arabera kontsulta egokia gauzatuko da.

Lehenengo eta hirugarren betekizunek argi adierazten digute eraiki nahi dugun aplikazio informatikoak egituraren gainean kontsultak gauzatu nahi dituela. Azken batean, algoritmo generiko baten instantziazioak lortu nahi ditugu kontsultak egikaritzeko. Ez dago asko aztertu beharrik ondorioztatzeko, Strategy patroia ondo datorkigula funtzionalitate hori eskuratzeko. Hala egingo dugu eta horretarako, **7.43. irudia**n agertzen den Klase Diagrama erabiliko dugu aplikazio informatikoa eraikitzeko.



7.43. irudia. Aplikazioaren Klase Diagrama.

Bigarren betekizunari ere patroien ikuspegia aplikatuko diogu eta Factory Method patroiaren egitura eskuratu, eta inplementatzen saiatuko gara. Konturatu proposatu dugun Klase Diagramak aipatutako patroi biak paraleloki inplementatzen dituela `IKontsulta` interfazea eta `Kontsultaxx` klaseak erabilita.

Hona hemen `IKontsulta` interfazearen eta `Eraikitzailea` klasearen iturburu-kodeak:

```
public interface IKontsulta
{
  Enumeration galdera(Container c);  }

//---------------------

public class Eraikitzailea
{
  IKontsulta create(int mota) {

    if (mota==1) return new Kontsulta1();
      if (mota==2)  return new Kontsulta2();
      return null;  };}
```

Betekizunetan eskatzen den kontsulten funtzionalitatea aztertuta, konturatzen gara kontsultak nahiko konplexuak izan daitezkeela zeren sailkapenak, azpimultzoak eta iterazioak eskatzen baitira. Hori guztia guk programatu beharko bagenu aldiko, kontsulta bakoitza programatzeak luze joko liguke. Horren ordez,

kontsultako JGL liburutegia erabiliko dugu kontsultaren semantika tratatzeko, eta emaitza, `Enumeration` motako objektu batean itzuliko da. Horrela, klase nagusiak nahikoa izango du `Eraikitzailea` motako objektu bat eraiki eta objektu horren `create()` metodoari adierazi zein kontsulta gauzatu nahi den. Metodoak `IKontsulta` motako objektu bat itzuliko du eta nahikoa izango da objektu horren `galdera()` metodoa egikaritzea adierazi dugun kontsulta egikaritzeko. Hurrengo listatuak azaldutako prozesu guztia laburbiltzen du.

```
public class Nagusia
{
  public Nagusia()  {}

  public static void main(String[] args)
  {   Pertsonak pertsonala=new Pertsonak();
    Eraikitzailea eraiki=new Eraikitzailea();
    IKontsulta galdera=eraiki.create(1);
    Enumeration e=galdera.galdera(pertsonala.lortuPertsonak());
    while (e.hasMoreElements()) {
      Pertsona p=(Pertsona)e.nextElement();
      p.ikuskatu();    }}}}
```

Kontu egin aplikazioaren funtzionalitateak erabat kapsulatzen direla diseinu-patroiak erabilita. Aldi berean, Strategy patroiaren algoritmo bakoitzaren inplementazio zehatza JGL liburutegiaren esku gelditzen da liburutegiaren klaseak erabiltzen baitira algoritmo horiek inplementatzeko. Azaldutakoa nola gauzatzen den zehatz-mehatz ondoko listatuak adierazten du:

```
public class Kontsulta1 implements IKontsulta
{
  public Enumeration galdera(Container c) {
    UnaryPredicate gizona=new BindSecondPredicate(new
EqualString(),new String("gizona"));
    UnaryPredicate nagusia=new BindSecondPredicate(new
GreaterNumber(),new Integer(18));
    UnaryPredicate izanGizona=new
UnaryComposePredicate(gizona,new LortuSexua());
    UnaryPredicate izanNagusia=new
UnaryComposePredicate(nagusia,new LortuAdina());
    UnaryPredicate betetzen=new UnaryAnd(izanGizona,izanNagusia);
    Sequence c1=(Sequence)Filtering.select(c,betetzen);
    BinaryPredicate bp=new BinaryComposePredicate(new
GreaterString(),new LortuIzena(),new LortuIzena());
    Sorting.sort(c1,bp);
    return c1.elements();   }}

//-----------------------
```

```
    public class Kontsulta2 implements IKontsulta
    {  public Enumeration galdera(Container c){
        UnaryPredicate emakume=new BindSecondPredicate(new
EqualString(),new String("emakume"));
        UnaryPredicate hiria=new BindSecondPredicate(new
EqualString(),new String("donosti"));
        UnaryPredicate izanEmakume=new
UnaryComposePredicate(emakume,new LortuSexua());
        UnaryPredicate izanDonosti=new UnaryComposePredicate(hiria,new
LortuHiria());
        UnaryPredicate betetzen=new UnaryAnd(izanEmakume,izanDonosti);
        Sequence c1=(Sequence)Filtering.select(c,betetzen);
        BinaryPredicate bp=new BinaryComposePredicate(new
GreaterNumber(),new LortuAdina(),new LortuAdina());
        Sorting.sort(c1,bp);
        return c1.elements();   }}

    //-------------------------

    import com.objectspace.jgl.*;
    public class LortuHiria implements UnaryFunction
    {
      public Object execute(Object o) {
        return ((Pertsona)o).getHiria();  }}

    //-------------------------

    import com.objectspace.jgl.*;
    public class LortuAdina implements UnaryFunction
    {
      public Object execute(Object o) {
        return new Integer (((Pertsona)o).getAdina() );  }}

    //-------------------------

    import com.objectspace.jgl.*;
    public class LortuIzena implements UnaryFunction
    {
      public Object execute(Object o) {
        return ((Pertsona)o).getIzena();   }}

    //-------------------------

    import com.objectspace.jgl.*;
    public class LortuSexua implements UnaryFunction
    {
      public Object execute(Object o) {
          return ((Pertsona)o).getSexua();  }}
```

`Kontsulta1` klaseak pertsonen egiturako 18 urte baino gehiagoko gizonezkoak detektatu eta sailkatu nahi ditu; horretarako, JGL liburutegiaren predikatuak eta algoritmoak erabiliko ditu. Predikatuekin adina eta gizonezkoak direla kontrolatu nahi da, eta algoritmoekin, ostera, edukiontzian zenbat pertsona dauden baldintza horiek betetzen dituztenak. Lehen eragiketa batean, `Filtering` klasearen `select()` metodo estatikoak aipatutako baldintzak betetzen dituzten pertsonak aukeratzen ditu eta `Sorting` klasearen `sort()` metodoak aukeratutako pertsonak sailkatu egiten ditu. Emaitzaren edukiontziari `elements()` metodoa egikaritzen zaio `Enumeration` motako objektu bat eraikitzeko. Bukatzeko, objektu hori izango da `galdera()` metodoak itzuliko duena emaitza gisa. `Kontsulta2` klaseak `Kontsulta1` klasearen egitura berari jarraitzen dionez, ez zaitut irakurle aspertuko haren xehetasunak aurkezten.

7.13. LABURPENA

Objektuetara zuzendutako programazioak klaseak erabili izan ohi ditu domeinuko elementuak egituratu eta inplementatzeko. Programatzailearen diseinuetan, klaseak berak adierazten du informazioaren egoera eta portaera kapsulatzeko muga non dagoen. Testuinguru hori programatzaile hasiberriarentzat egokia eta aldi berean malgua da; aplikazio informatikoak, klaseetan antolatzen ditu eta klaseen arteko erlazio mota desberdinak Klase Diagraman azaleratzen ditu. Aplikazioak eraiki ahala, ordea, konturatzen da diseinuen ezaugarri batzuk behin eta berriz errepikatu egiten direla, hots, ebazpenaren domeinuan erregulartasunak izaten direla.

Egoera horrek tentsio-uneak sortzen ditu erakundean eta aukeratu egin behar da lehen bezala jarraitzea ala baliabideak jartzea abstrakzio maila berri batera igarotzeko. Abstrakzio maila berrian erregulartasunak deskribatu eta diseinatu egingo dira erakundeko partaide bakoitzak ezagutu eta erabil ditzan. Horrela, astiro-astiro, erakundea diseinu-patroien katalogo bat eratuz eta hedatuz joango da. Hutsetik ez hasteko, eta batez ere gurpila asmatzen ez egoteko, ezagunenak eta hedatuenak aztertu ditugu atal honetan.

7.14. OINARRIZKO BIBLIOGRAFIA

Alur, D.; Malks, D. eta Crupi, J. (2003): *Core J2EE Patterns: Best Practices and Design Strategies*, Prentice Hall.

Crawford, W. eta Kaplan, J. (2003): *J2EE Design Patterns*, O´Reilly.

Freeman, E. eta Freeman, E. (2004): *Head First Design Patterns*, O´Reilly Media.

Gamma, E.; Helm, R.; Johnson, R. eta Vlissides, J. (1995): *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley.

Grand, M. (1998a): *Patterns in Java. Vol I. A catalog of reusable Design Patterns illustrated with UML*, John Wiley & Sons.

————, (1998b): *Patterns in Java. Vol II*, John Wiley & Sons.

Kuchana, P. (2004): *Software Architecture Design Patterns in Java*, Auerbach Publications.
Larman, C. (2002): *UML y Patrones*, Prentice Hall.
Riel, A. J. (1996): *Object-Oriented Design Heuristics*, Addison Wesley.
Thomas, J.; Young, M.; Brown, K. eta Glover, A. (2004): *Java Testing Patterns*, Wiley.

7.15. HELBIDE INTERESGARRIAK

Zer dago bertan?	Web gunea
Informazioa diseinu-patroiei buruz	`http://hillside.net/patterns/`
Baliabideak diseinu-patroiei buruz	`http://c2.com/cgi/wiki?DesignPatterns`
Christopher Alexander arkitektoari buruzko informazioa	`http://c2.com/cgi/wiki?ChristopherAlexander`
Douglas C. Schmidt-en web ataria	`http://www.cs.wustl.edu/~schmidt/patterns.html`
Thinking in Patterns with Java liburua pdf formatuan	`http://www.mindview.net/Books/TIPatterns/`
Tutorial txikia	`http://www.industriallogic.com/papers/learning.html`
Diseinu-patroiak C# lengoaian	`http://www.dofactory.com/Patterns/Patterns.aspx`
J2EE diseinu-patroiak	`http://java.sun.com/blueprints/corej2eepatterns/index.html`
GRASP patroien azalpena	`http://www.adictosaltrabajo.com/tutoriales/tutoriales.php?pagina=grasp`
Tutorial txikia J2EE diseinu-patroiei buruz	`http://www.adictosaltrabajo.com/tutoriales/tutoriales.php?pagina=patronesj2ee`

8. Bibliografia

Agile Alliance (2001a): *Manifesto for Agile Software Development*, <http://www.agilealliance.org/>.

————, (2001b): *Principles: The Agile Alliance*, <www.agilealliance.org/principles.html>.

Ahern, D.; Clouse, A. eta Turner, R. (2003): *CMMI Distilled: A Practical Introduction to Integrated Process Impro*vement, Addison Wesley.

Aho, A. V.; Kernighan, B. W. eta Weinberger, P. J. (1979): "Awk: a pattern scanning and processing language", *Software Practice and Experience*, **9(4)**, 267-79.

Alexander, C. (1979): *The Timeless Way of Building*, Oxford University Press.

Alexander, C.; Ishikawa, S. eta Silverstein, M. (1977): *A Pattern Language*, Oxford University Press.

Alur, D.; Malks, D.eta Crupi, J. (2003): *Core J2EE Patterns: Best Practices and Design Strategies*, Prentice Hall.

Ambler, S. W. (1995): *The Object Primer: Application Developer's Guide to Object Orientation*, Cambridge University Press.

————, (1997): *Building Object Applications That Work: Your Step-By-Step Handbook for Developing Robust Systems with Object Technology*, Cambridge University Press, <www.ambysoft.com/buildingObjectApplications.html>.

————, (1998): *Process Patterns: Building Large-Scale Systems Using Object Technology*, Cambridge University Press, <www.ambysoft.com/processPatterns.html>.

————, (1999): *More Process Patterns: Delivering Large-Scale Systems Using Object Technology*, Cambridge University Press, <www.ambysoft.com/moreProcessPatterns.html>.

————, (2001a): *The Object Primer, Second Edition: The Application Developer's Guide to Object Orientation*, Cambridge University Press, <www.ambysoft.com/theObjectPrimer.html>.

————, (2001b): *Enterprise Unified Process White Paper*, <www.ronin-intl.com/publications/unifiedProcess.htm>.

————, (2001c): *Agile Modeling Home Page*, <http://www.agilemodeling.com/>.

————, (2001d): *The Design of a Robust Persistence Layer*, <www.ambysoft.com/persistenceLayer.html>.

————, (2001e): *Mapping Objects to a Relational Database*, <www.ambysoft.com/mappingObjects.html>.
Arnold, K.; Gosling, J. eta Holmes, D. (2000): *The Java™ Programming Language*, Addison Wesley.
Arnow, D. eta Weiss, G. (2000): *Introducción a la programación con Java. Un enfoque orientado a objetos*, Addison Wesley.
Bass, L.; Clements, P. eta Kazman, R. (1998): *Software Architecture in Practice*, Addison Wesley .
Beck, K. (2000): *Extreme Programming Explained: Embrace Change*, Addison Wesley .
Beck, K. eta Cunningham, W. (1989): *A Laboratory for Teaching Object-Oriented Thinking*, Proceedings of OOPSLA'89, 1-6.
Beck, K. eta Fowler, M. (2001): *Planning Extreme Programming*, Addison Wesley.
Beedle, M. eta Schwaber, K. (2001): *Agile Software Development With SCRUM*, Prentice Hall.
Beck, K. (2002): *Una explicacion de la Programación Extrema. Aceptar el cambio*, Addison Wesley.
————, (2004): *JUnit Pocket Guide*, O`Reilly.
Bennett, D. (1997): *Designing Hard Software: The Essential Tasks*, Manning Publications.
Bezroukov, N. (1999a): *Open Source Software Development as a Special Type of Academic Research*, FirstMonday, Volume 4 - Number 10.
————, (1999b): *A Second Look at the Cathedral and Bazaar*, FirstMonday, Volume 4 - Number 12.
Bishop, J. (1999): *Java. Fundamentos de programación*, Addison Wesley.
Boehm, B. W. (1981): *Software Engineering Economics*, Prentice Hall.
————, (1988): "A Spiral Model Of Software Development and Enhancement", *IEEE Computer*, **21(5)**, 61-72.
Booch, G. (1994): *Object-Oriented Analysis and Design with Applications*, Addison Wesley.
Booch, G.; Rumbaugh, J. eta Jacobson, I. (1999): *El Lenguaje Unificado de Modelado*, Addison Wesley.
Bremer, M. (1999): *UnTechnical Writing: How to Write About Technical Subjects and Products So Anyone Can Understand*, UnTechnical Press.
Brooks, F. P. (1995): *The Mythical Man Month: Essays on Software Engineering Anniversary Edition*, Addison Wesley.
Brown, W. J.; McCormick, H. W. III, eta Thomas, S. W. (2000): *AntiPatterns in Project Management*, John Wiley & Sons.
Bruegge, B. eta Dutoit, A. (2003): *Object-Oriented Software Engineering: Using UML, Patterns and Java*, Prentice Hall.

Buschmann, F.; Meunier, R.; Rohnert, H.; Sommerlad, P. eta Stal, M. (1996): *Pattern-Oriented Software Architecture: A System of Patterns*, John Wiley & Sons.

Campione, M.; Walrath, K.; Zakhour, S. eta Bodoff, S. (2005): *The Java™ Tutorial: A Short Course on the Basics*, Addison Wesley.

Ceballos, F. J. (2000): *Java 2. Curso de programación*, Rama.

Chrissis, M.; Konrad, M. eta Shrum, S. (2003): *CMMI: Guidelines for Process Integration and Product Improvement*, Addison Wesley.

Christel, M. G. eta Kang, K. C. (1992): *Issues in Requirements Elicitation*, Software Engineering Institute (SEI) Technical Report CMU/SEI-92-TR-12, <http://www.sei.cmu.edu/>.

Coad, P.; Lefebvre, E. eta DeLuca, J. (1999): *Java Modeling in Color with UML: Enterprise Components and Process*, Prentice Hall .

Coady, Y.; Kiczales, G.; Feeley, M. eta Smolyn, G. (2001): *Using AspectC to Improve the Modularity of Path-Specific Customization in Operating System Code*, Proceedings of the Joint European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-9), Vienna, Austria, September, 78-88.

Cockburn, A. (1998): *Surviving Object-Oriented Projects: A Manager's Guide*, Addison Wesley.

————, (2001a): *Writing Effective Use Cases*, Boston, MA: Addison Wesley.

————, (2001b): *Crystal Clear: A Human-Powered Software Development Methodology for Small Teams*, <http://members.aol.com/humansandt/crystal/clear/>.

————, (2001c): *Characterizing People as Non-Linear, First-Order Components in Software Development*, <members.aol.com/humansandt/papers/nonlinear/ nonlinear.htm>.

————, (2002): *Agile Software Development*, Addison Wesley.

Cohen, F. (2004): *Java Testing and Design : From Unit Testing to Automated Web Tests*, Prentice Hall.

Colyer, A.; Clement, A.; Harley, G. eta Webster, M. (2005): *Aspect-Oriented Programming in Eclipse with AspectJ and the AspectJ Development Tools*, Addison Wesley.

Constantine, L. L. eta Lockwood, L. A. D. (1999): *Software For Use: A Practical Guide to the Models and Methods of Usage-Centered Design*, ACM Press.

Coplien, J. O. (1997): *Idioms and Patterns as Architectural Literature*, IEEE Software Special Issue on Objects, Patterns, and Architectures, January.

————, (1998): *Software Design Patterns: Common Questions and Answers*. Rising L. (Ed.), The Patterns Handbook: Techniques, Strategies, and Applications, Cambridge University Press, January, 311-320.

Coplien, J. eta Harrison, N. (2001): *Organizational Patterns Site*, <www.bell-labs.com/cgi-user/OrgPatterns/OrgPatterns>.

Crawford, W. eta Kaplan, J. (2003): *J2EE Design Patterns*, O´Reilly.
Davis, A. M. (1995): *201 Principles of Software Development*, McGraw Hill.
Deitel, H. eta Deitel, P. (2004): *Cómo Programar en Java*, Prentice Hall.
Dijkstra, E.W. (1976): *A Discipline of Programming*, Prentice Hall.
–––––––––, (1982): *Selected writings on computing: a personal perspective*, Springer Verlag.
Douglass, B. P. (1999): *Doing Hard Time: Developing Real-Time Systems With UML, Objects, Frameworks, and Patterns*, Addison Wesley.
D'Souza, D. F. etaWills, A. C. (1999): *Objects, Components, and Frameworks with UML: The Catalysis Approach*, Addison Wesley.
El Eman, K.; Drouin, J. eta Melo W. (1998): *SPICE. The theory and practice of Software Process Improvement and Capability Determination*, IEEE Computer Society Press.
Elrad, T.; Aksit, M.; Kiczales, G.; Lieberherr, K. eta Ossher, H. (2001): *Discussing Aspects of AOP*, Communications of the ACM, October, 33-38.
European Space Agency. Ariane 5: Flight 501 failure. Report by the enquiry board, <www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html>.
Evans, G. (2001): *Palm Sized Process: Point of Sale Gets Agile*, Software Development, September 2001.
Fowler, M. (1997): *Analysis Patterns: Reusable Object Models*, Addison Wesley.
————, (1999): *Refactoring: Improving the Design of Existing Code*, Addison Wesley.
————, (2001a): *The New Methodology*, <www.martinfowler.com/articles/newMethodology.html>.
––––––––––, (2001b): *Is Design Dead?* <www.martinfowler.com/articles/designDead.html>.
Fowler, M. eta Scott, K. (1999): *UML Distilled Second Edition: A Brief Guide to the Standard Object Modeling Language*, Addison Wesley .
Frankel, D. (2003): *Model Driven Architecture: Applying MDA to Enterprise Computing*, John Wiley & Sons.
Freeman, E, eta Freeman, E. (2004): *Head First Design Patterns*, O´Reilly Media.
Gane, C.eta Sarson, T. (1979): *Structured Systems Analysis: Tools and Techniques*. Prentice Hall.
Gamma, E.; Helm, R.; Johnson, R. eta Vlissides, J. (1995): *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley.
Gilb, T. eta Graham, D. (1993): *Software Inspection*, Addison Wesley.
Goldberg, A. (1984): *Smalltalk-80: The Interactive Programming environment*, Addison Wesley.
Goldberg, A. eta D. R. (1983): *Smalltalk-80: The Language and Its Implementation*, Addison Wesley.
Gottesdiener, E. (2001): *Specifying Requirements With a Wall of Wonder*, <http://www.therationaledge.com/content/nov_01/t_wallOfWonder_eg.html>.

Gradecki, J. eta Lesiecki, N. (2003): *Mastering AspectJ*, Wiley.
Graham, I.; Henderson-Sellers, B. eta Younessi, H. (1997): *The OPEN Process Specification*, ACM Press.
Grand, M. (1998): *Patterns in Java. Vol I. A catalog of reusable Design Patterns illustrated with UML*, John Wiley & Sons.
————, (1998): *Patterns in Java, Vol II*, John Wiley & Sons.
Greenbaum, J. eta Kyng, M. (1991): *Design At Work: Cooperative Design of Computer Systems*, Lawrence Erlbaum Associates.
Henderson-Sellers, B. (1992): A book of object-oriented knowledge, Prentice-Hall.
Highsmith, J. A. III (2000): *Adaptive Software Development: A Collaborative Approach to Managing Complex Systems*, Dorset House.
Hightower, R.; Onstine, W.; Visan, P.; Payne, D. eta. Gradecki, J. (2004): *Professional Java Tools for Extreme Programming: Ant, XDoclet, JUnit, Cactus, and Maven (Programmer to Programmer),* Wrox.
Hock, D. W. (2000): *Birth of the Chaordic Age*, Berrett-Koehler.
Hohmann, L. (1996): *Journey of the Software Professional: The Sociology of Computer Programming*, Prentice Hall.
Humphrey, W. S. (1995): *A discipline for Software Engineering*, Addison Wesley.
————, (1999): *Introduction to the Team Software Process*, Addison Wesley.
————, (2001): *Introducción al Proceso del Software Personal*, Addison Wesley.
Hunt, J. (2003): *Guide to the Unified Process Featuring UML, Java and Design Patterns*. Springer.
Hunt, A. eta Thomas, D. (2000): *The Pragmatic Programmer: From Journeyman to Master*. Addison Wesley.
Hunt, A. eta Thomas, D. *Pragmatic Unit Testing in Java with JU*nit. The Pragmatic Programmers.
Jacobson, I.; Booch, G. eta Rumbaugh, J. (1999): *The Unified Software Development Process*, Addison Wesley .
————, ————, eta ————, (2000): *El Proceso Unificado de Desarrollo de Software*, Addison Wesley.
Jacobson, I.; Christerson, M.; Jonsson, P. eta Overgaard, G. (1992): *Object-Oriented Software Engineering: A Use Case Driven Approach*, ACM Press.
Jacobson, I.; Griss, M. eta Jonsson, P. (1997): *Software Reuse: Architecture, Process, and Organization for Business Success*, ACM Press.
Jacobson, I. eta Pan-Wei Ng. (2005): *Aspect-Oriented Software Development with Use Cases*, Addison Wesley.
Jeffries, R.; Anderson, A. eta Hendrickson, C. (2001): *Extreme Programming Installed*, Boston, MA: Addison Wesley.
Jeffries, R. (2001a): *Essential XP: Card, Conversation, Confirmation*, <www.xprogramming.com/xpmag/expCardConversationConfirmation.htm>.
————, (2001b): *Essential XP: Documentation*, <www.xprogramming.com/xpmag/expDocumentationInXp.htm>.

Jones, C. (2000): *Software Assessments, Benchmarks, and Best Practices*, Addison Wesley.

Kasse, T. (2004): *A Practical Guide to CMMI Implementation*, Artech House.

Kerievsky, J. (2001): *Patterns and XP. Extreme Programming Examined*, 207-220, Eds. Succi, G. eta Marchesi, M. Addison Wesley.

Kerth, N. (2001): *Project Retrospectives: A Handbook for Team Reviews*, Dorset House.

Kiczales, G.; Rivières, J. eta Bobrow, D. G. (1991): *The Art of the Metaobject Protocol*, MIT Press.

Kiczales, G. (1992): *Towards a New Model of Abstraction in the Engineering of Software*, Proceedings of the International Workshop on New Models for Software Architectures (IMSA): Reflection and Metalevel Architecture, Tokyo, Japan, November, 1-11.

Kiczales, G.; Hilsdale, E.; Hugunin, J.; Kersten, M.; Palm, J. eta Griswold, W. (2001): *An Overview of AspectJ*, European Conference on Object-Oriented Programming (ECOOP), LNCS 2072, Springer-Verlag, Budapest, 327-353.

Kiselev, I. (2002): *Aspect-Oriented Programming with AspectJ*, Sams.

Kruchten, P. (1995): "The 4+1 View Model of Architecture", *IEEE Software*, **12(6)**, 42–50.

————, (2000): *The Rational Unified Process, Second Edition: An Introduction*, Addison Wesley.

Kuwabara, K. (2000): *Linux: A Bazaar at the Edge of Chaos*, FirstMonday, Volume 5 - Number 3.

Kuchana, P. (2004): *Software Architecture Design Patterns in Java*, Auerbach Publications.

Laddad, R. (2002): *I Want My AOP! Parts1-3*, Java World, <www.javaworld.com/javaworld/jw-01-2002/jw-0118-aspect.html>.

————, (2003): *AspectJ in Action*, Manning Publications.

Lane, D. (2004): *JUnit: The Definitive Guide*,O`Reilly.

Lano, K. (2005): *Advanced Systems Design with Java, UML and MDA*. Butterworth-Heinemann.

Larman, C. (2002): *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*, Prentice Hall.

————, (2002): *UML y Patrones*, Prentice Hall.

Lee, R. eta Tepfenhart, W. (2003): *Practical Object-Oriented Development with UML and Java*, Prentice Hall.

Lesieki, N. (2002): *Test Flexibly with AspectJ and Mock Objects*, IBM DeveloperWorks, <www-106.ibm.com/developerworks/java/library/j-aspectj2/>.

Leuf, B. eta Cunningham, W. (2001): *The Wiki Way: Quick Collaboration on the Web*, Addison Wesley .

Li, L. (1998): *Java. Data Structures and Programming*, Springer-Verlag.

Lima Díaz, F. (2000): *Manual avanzado de Java 2*, Anaya Multimedia.

Link, J. eta Fröhlich, P. (2003): *Unit Testing in Java: How Tests Drive the Code*, Morgan Kaufman.

Linthicum, D. S. (2000): *Enterprise Application Integration*, Addison Wesley .

Martin, R. (2001): *The Process*, <www.objectmentor.com/ publications/ RUPvsXP.pdf>.

Massol, V. (2004): *JUnit in Action*, Manning Publications.

Mayhew, D. J. (1992): *Principles and Guidelines in Software User Interface Design*, Prentice Hall.

McConnell, S. (1993): *Code Complete: A Practical Handbook of Software Construction*, Microsoft Press.

Meyer, B. eta Dino Mandrioli, D. (1992): *Advances in Object-Oriented Software Engineering*, Prentice Hall.

Meyer, B. (1992): "Applying Design by Contract", *IEEE Computer,* **25(10)**, 40-51.

————, (1995): "On formalism in specifications", *IEEE Software,* **2(1)**, 6-26.

————, (1997): *Object-Oriented Software Construction*, Prentice Hall, 2. argitalpena.

————, (1998): *Construcción de Software Orientado a Objetos*, Prentice Hall.

Microsoft Corporation (1995): *The Windows Interface Guidelines for Software Design: An Application Design Guide*, Microsoft Press.

----------, (2001): *The Microsoft Solutions Framework (MSF)*, <www.microsoft.com/business/services/mcsmsf.asp>.

Miles, R. (2005): *AspectJ Cookbook*, O´Reilly Media.

Musser, D. eta Saini, A. (1995): *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*, Addison Wesley.

Musser, D. R.; Derge, G. J. eta Saini, A. (2001): *STL Tutorial and Reference Guide, Second Edition: C++ Programming with the Standard Template Library*, Addison Wesley.

Naiburg, E. J. eta Maksimchuk, R. A. (2001): *UML for Database Design*, Boston, MA: Addison Wesley.

Newkirk, J. eta Martin, R. C. (2001): *Extreme Programming in Practice*, Addison Wesley.

Niemeyer, G. eta Poteet, J. (2003): *Extreme Programming with Ant: Building and Deploying Java Applications with JSP, EJB, XSLT, XDoclet, and JUnit*, Sams.

Object Management Group (2001a): *The Unified Modeling Language (UML) Specification*, <www.omg.org/technology/documents/formal/uml.htm>.

————, (2001b): Model Drive Architecture (MDA), <ftp.omg.org/pub/docs/ ormsc/01-07-01.pdf>.

Page-Jones, M. (1988): *Practical Guide to Structured Systems Design 2/e*, Prentice-Hall.

————, (2000): *Fundamentals of Object-Oriented Design in UML*, Dorset-House Publishing.

Parnas, D. (1972): *On the Criteria To Be Used in Decomposing Systems into Modules*, Communications of the ACM, December, 1053-1058.

Paulk, M.; Weber, C.; Curtis, B. eta Chrissis, M. (1995): *The Capability Maturity Model for Software: Guidelines for Improving the Software Process*, Addison Wesley.

Pawlak, R.; Retaillé, J-P. eta Seinturier, L. (2005): *Foundations of Aop for J2ee Development*, Apress.

Piattini, M.; Calvo-Manzano, J. A.; Cervera, J. eta Fernández, L. (1996): *Análisis y diseño detallado de Aplicaciones Informáticas de Gestión*, Rama.

Pressman, R. S. (2002): *Ingeniería del Software. Un enfoque práctico*, McGraw Hill, 5. argitalpena.

Rainsberger, J. B. (2004): *JUnit Recipes : Practical Methods for Programmer Testing*, Mannig Publications.

Raskin, J. (2000): *The Human Interface: New Directions for Designing Interactive Systems*, Addison Wesley.

Rational Corporation (2001): *Rational Unified Process Home Page*, <www.rational.com/products/rup/index.jsp>.

Raymond, E. (1998): *The Cathedral and the Bazaar*, FirstMonday, Volume 3 – Number 3.

————, (2001): *The Cathedral and the Bazaar*, O`Reilly.

Reingruber, M. C. eta Gregory, W. W. (1994): *The Data Modeling Handbook: A Best-Practice Approach to Building Quality Data Models*, John Wiley & Sons.

Riel, A. J. (1996): *Object-Oriented Design Heuristics*, Addison Wesley.

Roman, E.; Ambler, S. W.; Jewell, T. eta Marinescu, F. (2002): *Mastering Enterprise Java Beans, Second Edition*, John Wiley & Sons.

Rosenberg, D. eta Scott, K. (1999): *Use Case Driven Object Modeling With UML: A Practical Approach*, Addison Wesley .

Ross, R. G. (1997): *The Business Rule Book, Second Edition*, Business Rules Solutions.

Rumbaugh, J.; Blaha, M.; Premerlani, W.; Eddy, F. eta Lorensen, W. (1991): *Object-Oriented Modeling and Design*, Prentice Hall.

Rumbaugh, J.; Jacobson, I. eta Booch, G. (1999): *The Unified Modeling Language Reference Manual*, Addison Wesley .

————–, eta ————–, eta ————–, (2000): *El Lenguaje Unificado de Modelado Manuel de Referencia*, Addison Wesley.

Software Engineering Institute. (1995): *The Capability Maturity Model: Guidelines for Improving the Software Process*, Addison Wesley, Inc.

Stapleton, J. (1997): *DSDM: Dynamic Systems Development Method*, Addison Wesley.

Stroustrup, B. (1997): *The C++ Programming Language*, 3. argitalpena, Addison Wesley.

Tarr, P.L. (1999): *N Degrees of Separation: Multi-Dimensional Separation of Concerns*, Proceedings of the International Conference on Software Engineering (ICSE), ACM, Los Angeles, 107-119.

Thomas, J.; Young, M.; Brown, K. eta Glover, A. (2004): *Java Testing Patterns*, Wiley.

Vermeulen, A.; Ambler, S. W.; Bumgardner, G.; Metz, E.; Misfeldt, T.; Shur, J. eta Thompson, P. (2000): *The Elements of Java Style*, Cambridge University Press.

Wake, W. C. (2002): *Extreme Programming Explored*, Addison Wesley.

Walnes, J.; Abrahamian, A.; Cannon-Brookes, M. eta Lightbody, P. (2003): *Java Open Source Programming: with XDoclet, JUnit, WebWork, Hibernate*, Wiley.

Warmer, J. eta Kleppe, A. (1999): *The Object Constraint Language: Precise Modeling With UML*, Addison Wesley.

Weiss, E. H. (1991): *How To Write Usable User Documentation*, The Oryx Press.

Wells, J. D. (2001): *Extreme Programming: A Gentle Introduction*, http://www.extremeprogrammarnming.org/

Wiegers, K. (1999): *Software Requirements*, Microsoft Press.

Wilkinson, N. M. (1995): *Using CRC Cards: An Informal Approach to Object-Oriented Development*, Cambridge University Press.

Williams, L.; Kessler, R. R.; Cunningham, W. eta Jeffries, R. (2000): "Strengthening the Case for Pair Programming", *IEEE Software*, July/August, 19-25.

Wood, J. eta Silver, D. (1995): *Joint Application Development Second Edition*, John Wiley & Sons.

Xerox (2001): *Aspect-Oriented Programming*, <http://www.parc.xerox.com/csl/projects/aop/>.

Yourdon, E. (1989): *Modern Structured Analysis*, Prentice-Hall.

————, (1997): *Death March: The Complete Software Developer's Guide to Surviving "Mission Impossible" Projects*, Prentice-Hall.

Sailean argitaratu diren beste liburu batzuk

Algoritmika
Rosa Arruabarrena
1997an argitaratua
ISBN: 84-86967-82-1

Ordenadore bidezko irudigintza
Joseba Makazaga, Asier Lasa
1998an argitaratua
ISBN: 84-86967-90-2

Oinarrizko programazioa. Ariketa-bilduma
Arantza Diaz de Illarraza, Kepa Sarasola
1999an argitaratua
ISBN: 84-8438-002-5

Zirkuitu elektriko eta elektronikoen oinarrizko analisia
Olatz Arbelaitz, Txelo Ruiz
2001ean argitaratua
ISBN: 84-8438-018-1

Linux sistemaren eta sarearen administrazioa
Iñaki Alegria
2003an argitaratua
ISBN: 84-8438-040-8

Sistema digitalen diseinu-hastapenak. Oinarrizko kontzeptuak eta adibideak
Olatz Arbelaitz eta beste
2005ean argitaratua
ISBN: 84-8438-069-6