Author(s): Leppänen, Mauri; Käkölä, Timo; Nurminen, Miika

Title: ITKA111 Oliosuuntautunut analyysi ja suunnittelu

Year: 2009

Version:

Please cite the original version:

Leppänen, M., Käkölä, T., & Nurminen, M. (2009). ITKA111 Oliosuuntautunut analyysi ja suunnittelu. Jyväskylän yliopisto.

ITKA111 OLIOSUUNTAUTUNUT ANALYYSI JA SUUNNITTELU

LUENTOMONISTE

Mauri Leppänen
Timo Käkölä
Miika Nurminen

Jyväskylän yliopisto
Informaatioteknologian tiedekunta
Kevät 2009

Tämä luentomoniste on tarkoitettu *Oliosuuntautunut analyysi ja suunnittelu* -kurssin tukimateriaaliksi. Moniste perustuu pääosin Mauri Leppäsen *Oliokeskeinen tietojärjestelmien kehittäminen* -luentomonisteeseen 2000-2005 ja Timo Käkölän lisäyksiin 2006. Materiaali ei yksinään sovellu itseopiskeluun, vaan toimii luentojen ja harjoitusten tukena. Syvällisempi perehtyminen asiaan vaatii kirjallisuuteen tutustumista.

Monisteen ja kurssin sisältöön ovat lisäksi vaikuttaneet ainakin Jonne Itkonen, Tommi Kärkkäinen, Eetu Luoma, Vesa Korhonen ja Pekka Makkonen. Kiitokset.

Palaute ja korjausehdotukset ovat tervetulleita.

Jyväskylässä, toukokuu 2009

Miika Nurminen

Muutoshistoria

Pvm	**Muutokset**	**Tekijä(t)**
11.05.2009	2009-kurssin luentomoniste, painoversio	MN
01.06.2009	Lisätty sekvenssikaavio UML-liitteeseen	MN

Sisällysluettelo

1 OLIOLÄHESTYMISTAPA

Oliolähestymistapa (eli oliokeskeisyys[1]) tarkoittaa kehittämis- ja ohjelmointiparadigmaa, jonka mukaan todellisuus nähdään joukkona toisiinsa vuorovaikutuksessa olevia olioita (Koskimies, 1995). Kuva 1.1. havainnollistaa oliolähestymistavan "maailmankuvaa". Muita paradigmoja ohjelmoinnin puolella ovat

- proseduraalinen
- funktio-ohjelmointi
- logiikkaohjelmointi

Tietojärjestelmien kehittämisessä tunnistetaan seuraavia vaihtoehtoisia paradigmoja:

- tietokeskeinen
- toimintokeskeinen
- protoilu
- osallistuva.

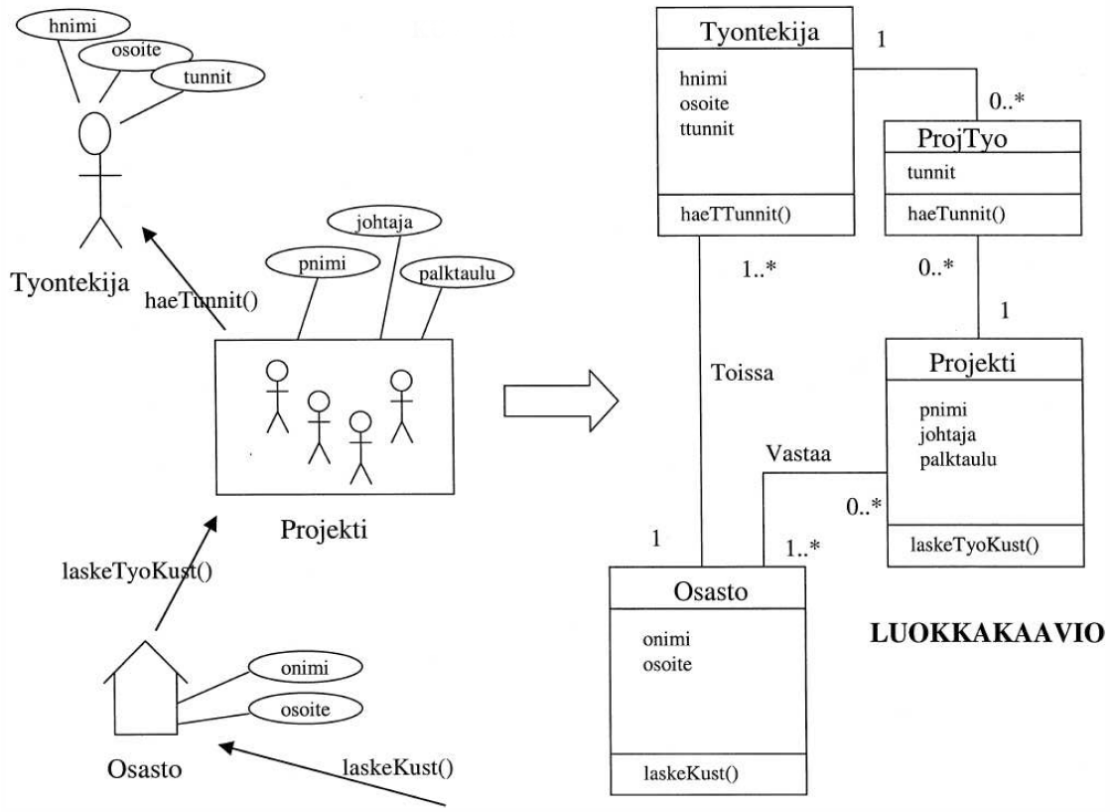
Seuraavaksi pyritään tarkentamaan kuvaa oliolähestymistavasta luomalla silmäys sen historiaan, määrittelemällä sen keskeiset käsitteet ja periaatteet sekä vertaamalla sitä perinteiseen tietojärjestelmien kehittämiseen.

1.1 Historiaa

Seuraavat viitteet oliolähestymistavan historiasta voidaan kirjata (vrt. Berard, 1993):

- 1963: Sutherlandin Sketchpad (object-oriented graphics)
- 1966: Simula (Dahl, Nygaard) (object-oriented programming language)
- 1972: Smalltalk (Kay, Xerox), termi 'object-oriented' tuli yleisemmin käyttöön
- 1982: Boochin oliosuunnittelua käsittelevä artikkeli "Object-Oriented Design".
- 1980-luvun alkupuoli: oliopohjaisia laitteistoarkkitehtuureja: iAPX 432 (Intel) ja Rekursiv. Parnas julkaisi ohjelmistoperheistä (program families) ja hänen ajatuksiaan otettiin laajasti käyttöön tietoliikenneteollisuudessa.
- 1980-luvun puolivälistä alkaen: uusia olio-ohjelmointikieliä: Objective-C, C++, Self, Beta, Eiffel, Flavors, Trellis/Owl, Oberon-2
- 1980-luvun puolivälissä ensimmäiset kaupalliset oliotietokannat tulivat markkinoille ja konferenssisarjoja (esimerkiksi OOPSLA, ECOOP) käynnistettiin.
- 1980-luvun lopussa: menetelmiä, jotka kattavat myös analyysin: OOA, OOSE ym.
- 1990-luvulla: object-oriented domain analysis, olio-ohjelmistojen testaus ja metriikat, oliopohjaiset CASE-välineet,
- 1998: OMG (Object Management Group) hyväksyi UML:n (Unified Modeling Language) standardiksi; UML-kirjoja, menetelmiä ja CASE-välineitä julkaistiin
- 2005: UML 1.4.2 julkaistiin ja hyväksyttiin kansainväliseksi ISO 19501 -standardiksi. UML 2.0 julkaistiin.

[1] Lähellä olevia termejä: oliosuuntautunut, oliokeskeinen ja oliopohjainen lähestymistapa. Englanninkielessä vastaavat termit ovat 'object-oriented', 'object-centered' ja 'object-based'. Edellä mainituille on joskus määritelty eksplisiittisesti käsitteellisiä erojakin. Tässä niihin ei oteta kantaa.



Kuva 1.1: Oliolähestymistapa ja olioiden vuorovaikutus

Oliolähestymistapa on levinnyt ohjelmointikielistä ohjelmointiin, tietojärjestelmien analyysiin ja suunnitteluun, liiketoiminnan (prosessien) analyysiin ja suunnitteluun, ja jopa strategiseen suunnitteluun.

1.2 Peruskäsitteitä

Oliolähestymistapa määrittyy seuraavilla peruskäsitteillä ja –periaatteilla.

Olio:

- rajattavissa ja yksilöitävissä oleva asia tai käsite, joka on merkityksellinen käsillä olevan tarkastelun kannalta, ja joka kattaa sekä rakenteen (tilan) että käyttäytymisen[2]
- voidaan käyttää myös nimitystä olioilmentymä (object instance – olio on luokan ilmentymä)
- voi olla esimerkiksi:
  - ◊ liiketoimintaolio (business object, application-*domain object*), joka vastaa liiketoiminnan käsitettä tai asiaa; esim. Matti Mainio, Markkinointiosasto.
  - ◊ käyttöliittymäolio (*boundary object*), joka on toteutettavan järjestelmän käyttäjärajapinnan tekninen komponentti; esim. tietty Ikkuna, Painike tai Anturi,
  - ◊ ohjausolio (*control object*), jonka vastuulla on ohjata ja koordinoida muiden olioiden toimintaa.

[2] "A discrete entity with a well-defined boundary and identity that encapsulates state and behavior" (UML, 1999)

(Olio)luokka (kuva 1.1):

- kuvaa joukkoa, joka muodostuu rakenteeltaan ja käyttäytymiseltään (lyh. ominaisuuksiltaan) samanlaisista olioista; on eräänlainen malli tai "muotti"; esim. Työntekijä, Osasto.
- olioiden luokittelun perustana käytetään näkemystä niiden todellisesta luonteesta eikä pelkästään niiden ilmituotuja ominaisuuksia (vrt. Lato ja Hevonen),
- olioluokat muodostavat hierarkian, jossa on *yliluokka* (esim. Henkilö) ja sen *aliluokkia* (esim. Sihteeri ja Sorvari; Työntekijä, Ohjelmoija).

Attribuutti:

- kuvaa luokkaan kuuluvien olioiden rakenteellista ominaisuutta (esim. Henkilön nimi, ikä ja paino).
- attribuuttiarvo yksilöi ominaisuuden (esim. Ville, 30, 75),
- arvoilla ei ole identiteettiä (vrt. 30), toisin kuin olioilla[3].

Operaatio:

- on olion käyttäytymisen aikaansaama toimenpiteen määritys (esim. *laskeKustannukset()*, *haeTunnit()*; se voidaan ymmärtää myös määritykseksi palvelusta, jonka olio tarjoaa.
- saman luokan olioilla on samat operaatiot
- joskus sama operaatio voidaan kohdistaa eri luokkiin kuuluviin olioihin (vrt. monimuotoisuus, polymorfismi),
- operaatioon voi liittyä parametreja (esim. *muutaPalkka(muutos)*)
- *metodi* on luokan olioihin sovitettu operaation toteutus

Viesti (message)

- oliot kommunikoivat keskenään lähettämällä viestejä toisilleen,
- viesti voidaan ymmärtää palvelupyynnöksi; viestin vastaanottaminen saa aikaan olion aktivoitumisen ja viestissä mainitun operaation (tarkemmin metodin) toteuttamisen. Viestin lähettäjä ei välttämättä tiedä täsmällisesti vastaanottavan olion luokkaa – suoritettava metodi määräytyy dynaamisen sidonnan avulla.
- Viesti voi kantaa mukanaan arvoja parametreina,

Assosiaatio:

- on kahden tai useamman luokan välinen rakenteellinen suhde (esim. Töissä –assosiaatio Työntekijän ja Osaston välillä)

Oliolähestymistavan perusperiaatteita:

Abstrahointi (abstraction):

- periaate, jonka mukaan nostetaan esille ongelman kannalta relevantit piirteet ja jätetään huomiotta epärelevantit
- tehdään tarkastelukohteen yksinkertaistamiseksi tiettyä tarkoitusta varten

[3] Ohjelmointikielestä riippuen attribuutit voivat olla myös olioita tai viitteitä olioihin, jolloin niillä on identiteetti. Toisaalta puhtaissa oliokielissä (esim. Smalltalk) myös perustyyppien arvot ovat olioita.

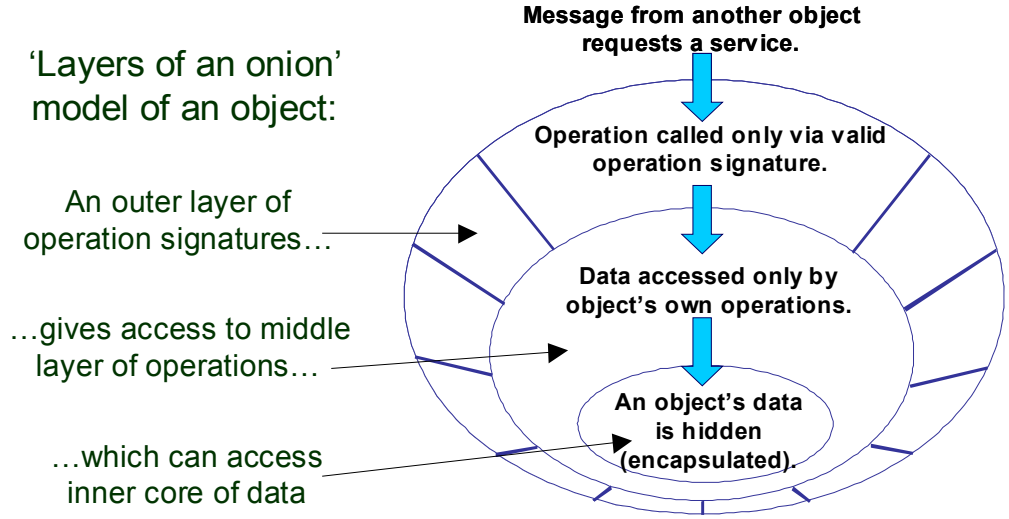
- mahdollistaa kuvaamisen, analysoinnin ja suunnittelun ilman häiritseviä yksityiskohtia (esim. olio vs. olioluokka, aliluokka vs. yliluokka),
- *tietoabstrahoinnilla* erotetaan luokan ulkoinen liittymä eli rajapinta (interface) sisäisestä toteutuksesta (implementation).

Identiteetti (identity):

- periaate, jonka mukaan oliot ovat yksikäsitteisesti tunnistettavissa muiden kuin ominaisuuksiensa (rakenteensa tai käyttäytymisensä) avulla (vrt. relaationaalinen käsitys).
- on verrattu käsitteelliseen "kahvaan" (handle, UML), jonka avulla muut oliot voivat tunnistaa olion ja lähettää sille viestejä.
- teknisesti esim. muistipaikan osoite, jossa olio sijaitsee

Kapselointi (encapsulation):

- periaate, jonka mukaan kootaan yhteen toisiinsa liittyvät asiat: olion (tieto)rakenne (attribuutit) ja käyttäytyminen (operaatiot)
- käytetään joskus myös synonyyminä tiedon suojaukselle



Kuva 1.2: Kapselointi ja tiedon piilotus: oliota ympäröivät suojaavat kerrokset

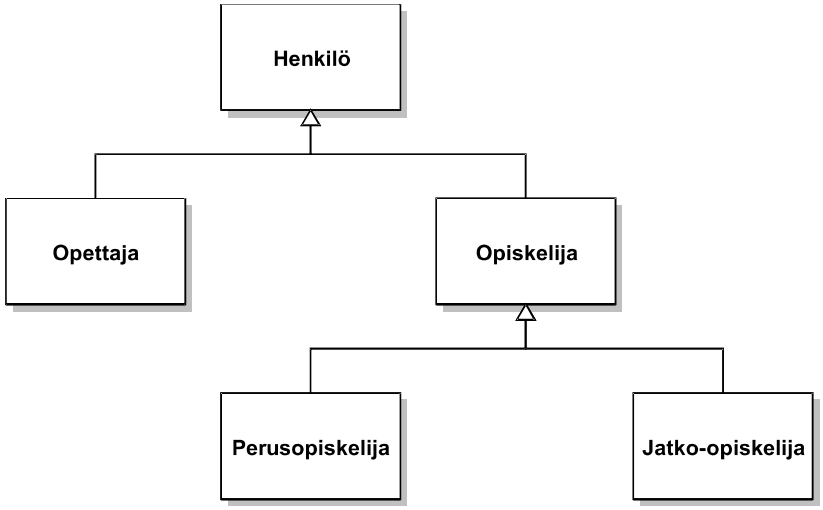
Tiedon suojaus (information hiding) (kuva 1.2; Bennett ym. 2006):

- periaate, jonka mukaan tietyt (yksityiskohtaiset) piirteet oliosta salataan sellaisilta olioilta, joiden ei kuulu niitä tuntea.
- tehdään määrittelemällä olioille rajapintoja, joihin toiset oliot voivat viitata. Rajapinnan takana oleviin toteutuksen yksityiskohtiin (implementation) ei ole muilla pääsyä.

Periytyminen (inheritance):

- periaate, jonka mukaan aliluokat perivät yliluokkansa ominaisuudet, jolloin aliluokkien olioilla on samantapainen rakenne ja käyttäytyminen kuin vastaavan yliluokan olioilla. Aliluokille voidaan määritellä lisäominaisuuksia, jotka eivät kuitenkaan saa olla täysin ristiriidassa perittyjen ominaisuuksien kanssa (kuva 1.3).

- *moniperinnässä* (multiple inheritance) aliluokalla on useampia yliluokkia, joilta se voi periä ominaisuuksia (esim. Asunto, Laiva → Asuntolaiva); huonosti käytettynä johtaa luokkahierarkian rämettymiseen[4]. Kaikissa oliokielissä (esim. Java, Smalltalk) ei ole suoraa tukea moniperinnälle, mutta voidaan usein korvata rajapintojen ja koostamisen avulla[5].



Kuva 1.3: Periytyminen

Monimuotoisuus, polymorfismi (polymorphism, overriding):

- periaate, jonka mukaan tietty operaatio (loogisella tasolla) voidaan toteuttaa eri tavoin (metodein) eri luokissa ja niiden olioissa riippuen luokan (ja olioiden) yksityiskohtaisesta luonteesta tai rakenteesta (esim. *Luo()* –operaation toteuttamiseksi voidaan määritellä erilainen metodi Tiedoston, Kuvakeen ja Ikkunan yhteyteen; tai *laskePalkka()* toteutetaan eri tavalla riippuen siitä, onko kysymyksessä kuukausi-, viikko- tai tuntipalkalla oleva Henkilö). Polymorfismia sovelletaan erityisesti abstraktien luokkien ja rajapintojen yhteydessä syrjäytettäessä aliluokissa yliluokassa määriteltyjä (abstrakteja) operaatioita (katso kuva 1.4 ja luku 5.3.1.1).

Dynaaminen sidonta (dynamic binding)

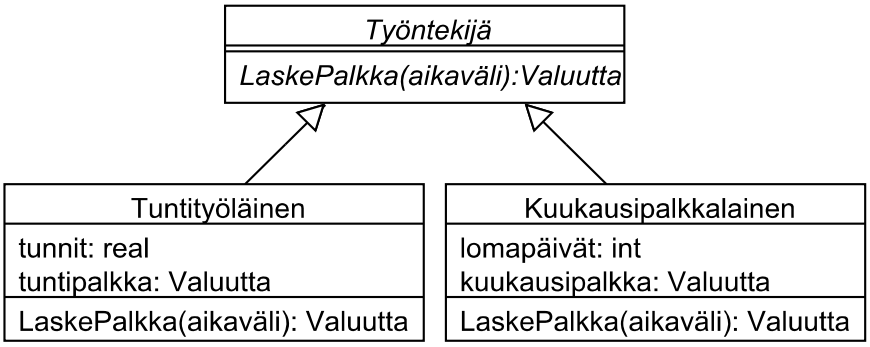
- periaate, jonka mukaan vasta ajoaikana päätetään siitä metodista, joka kutsutaan toteutettavaksi. Dynaaminen sidonta mahdollistaa polymorfismin.

Jakaminen (sharing):

- periaate, jonka mukaan olioiden samanlaiset ominaisuudet kuvataan kootusti yhdessä paikassa, yleensä oliojoukon määrittelevässä olioluokassa. Johtaa lyhyempiin kuvauksiin ja tukee luokkien uudelleenkäyttöä (reuse)

[4] Käytännön tietojärjestelmätyössä moniperintää on kritisoitu. Erityisesti seuraavan kaltainen käyttötapa on tyrmätty. Halutaan luokka A, jolla on luokkien B ja C käyttäytyminen. Toteutetaan A määrittelemällä se B:n ja C:n aliluokaksi. Tuloksena spagettia (vrt. Jacobson, 1992)

[5] Ks. esim. H. Kainulainen (2002). Moniperintä vastaan rajapinnat. http://www.mit.jyu.fi/opetus/opinnayte/LuK/MoniperintaVastaanRajapinnat/



Kuva 1.4: Abstrakti luokka (=luokalla ei ole olioita) Työntekijä määrittää operaation LaskePalkka, jonka aliluokat Tuntityöläinen ja Kuukausipalkkalainen toteuttavat eri tavoin (polymorfismi). Dynaamisen sidonnan ansiosta suoritus ohjautuu ajonaikana oikeaa tyyppiä olevalle oliolle (esim. laskettaessa kaikkien Työntekijöiden palkat käytetään Tuntityöläisen tai Kuukausipalkkalaisen metodeja olion konkreettisesta tyypistä riippuen, vaikka kutsujan kannalta tietorakenne sisältää vain Työntekijä-viitteitä).

Kyseiset periaatteet on otettu huomioon ja toteutettu eri tavoin eri menetelmissä ja kielissä. Oliokielten puolella vaihtelua on mm. siinä, miten pitkälle oliomaisuus on viety (C++ on moniparadigmakieli, Smalltalk puhdas oliokieli) on kielessä viety, tuessa moniperinnälle (C++:ssa on, Javassa ei) ja kielen tyypityksessä (Javassa staattinen tyypitys, Smalltalkissa dynaaminen).

1.3 Olioteknologia

Olioteknologialla tarkoitetaan niitä kieliä, ohjelmistoja ja arkkitehtuureja, joita on kehitetty tukemaan oliolähestymistavan mukaista analyysia, suunnittelua ja toteutusta:

- Arkkitehtuureja ja standardeja
  - ◊ Open Distributed Processing (ODP)
  - ◊ Common Object Request Broker Architecture (CORBA), (Distributed) Component Object Model (COM/DCOM/COM+), Enterprise JavaBeans (EJB)
  - ◊ Yhdistelmädokumentit (OLE, ActiveX, OpenDoc),
  - ◊ ODMG-93, ODMG-97, ODMG 3.0
- Oliokieliä ja mallinnuskieliä
  - ◊ Java, C#, Smalltalk, Python, Ruby, Eiffel, CLOS (the Common Lisp Object System), C++, Beta, Modula-3, Oberon-2, ...
  - ◊ UML 1.4, UML 2.0
- Oliomenetelmiä
  - ◊ OOA/OOD (Coad, Yourdon, 1991), OOSE (Jacobson ym., 1992), OOAD (Martin, Odell, 1992), OMT (Rumbaugh ym., 1991), OOSA (Shlaer, Mellor, 1992), DOOS (Wirfs-Brock ym., 1990), Booch, 1991, OPEN (Graham et al., 1997), USDP (Jacobson 1999), RUP (Kruchten 2000), ...
  - ◊ GOOD, HOOD, MOOD, ROOM, OOD/LVM, JSD/OOD, OOSD, OBA, SOMA, SOOA, OOREM, Objectory, OOZ, ...
- CASE-välineitä[6]
  - ◊ StarUML, ArgoUML

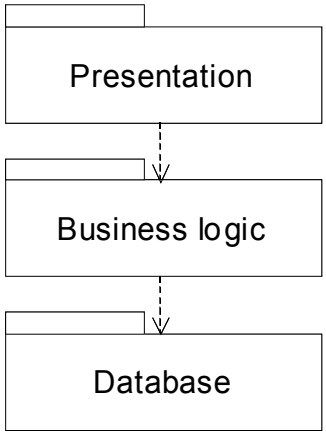
[6] http://en.wikipedia.org/wiki/List_of_UML_tools

- ◊ Rational Rose, Prosa, Magic Draw, Visual Paradigm, Together
- ◊ MetaEdit+ (ei varsinaisesti UML-työkalu, mutta sisältää UML:n yhtenä kohdealueen metamallina)

- Oliopohjainen tiedonhallinta
  - ◊ Oliokannan hallintajärjestelmät (OODBMS)[7]
  - ◊ Oliorelaatiokannan hallintajärjestelmät (ORDBMS)[8],
  - ◊ Relaatiokannan abstrahointityökalut (object-relational mapping)[9]
- Integroidut kehitysympäristöt (IDE) oliokielille
  - ◊ Eclipse, NetBeans, Visual Studio, Delphi, Squeak, …
- Sovelluskehykset (katso luku 5.2.2)

1.4 Järjestelmäarkkitehtuuri

Järjestelmäarkkitehtuuri ryhmittelee karkealla tasolla järjestelmään kuuluvat komponentit. Tyypillisessä kolmitasoisessa (3-tier) arkkitehtuurissa (kuva 1.5; Bennett 1999 s. 262) järjestelmän käyttöliittymän muodostamat luokat muodostavat oman tasonsa (presentation), pysyväisluonteisista tiedoista huoltapitävät luokat oman tasonsa (database) ja sovelluslogiikasta vastaavat luokat oman tasonsa (business logic). Kuvassa 1.6 on annettu yksityiskohtaisempi esimerkki 3-tasoisesta arkkitehtuurista. Arkkitehtuurin mallintamista käsitellään tarkemmin luvussa 4.1.

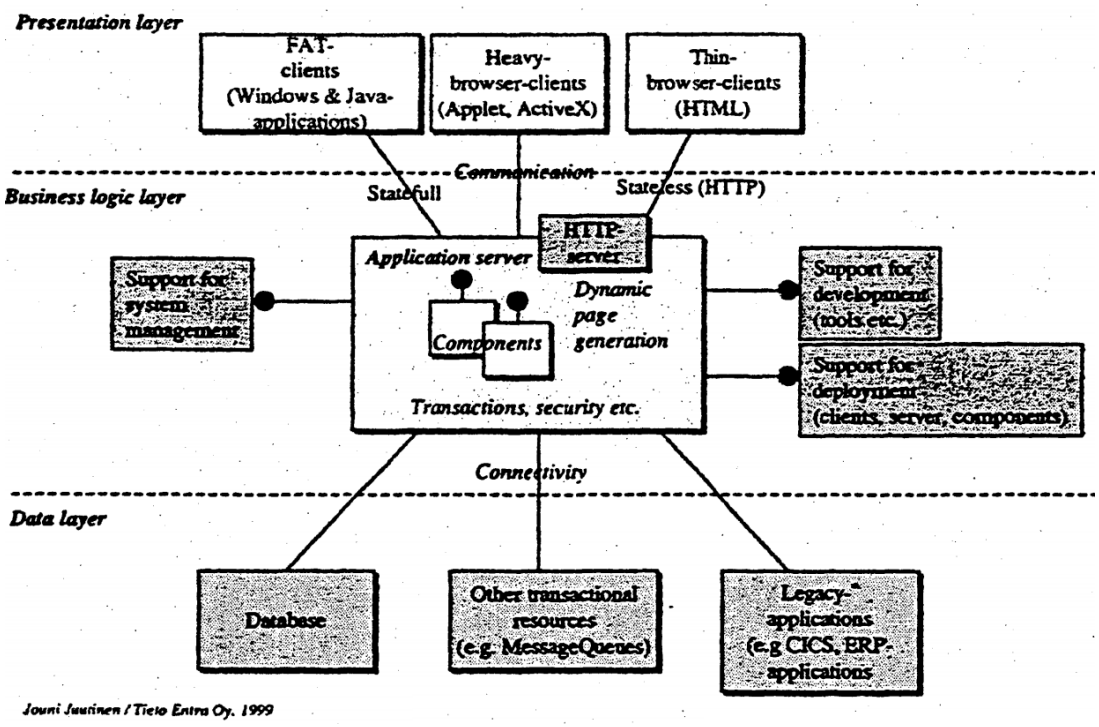


Kuva 1.5: Kolmitasoarkkitehtuuri

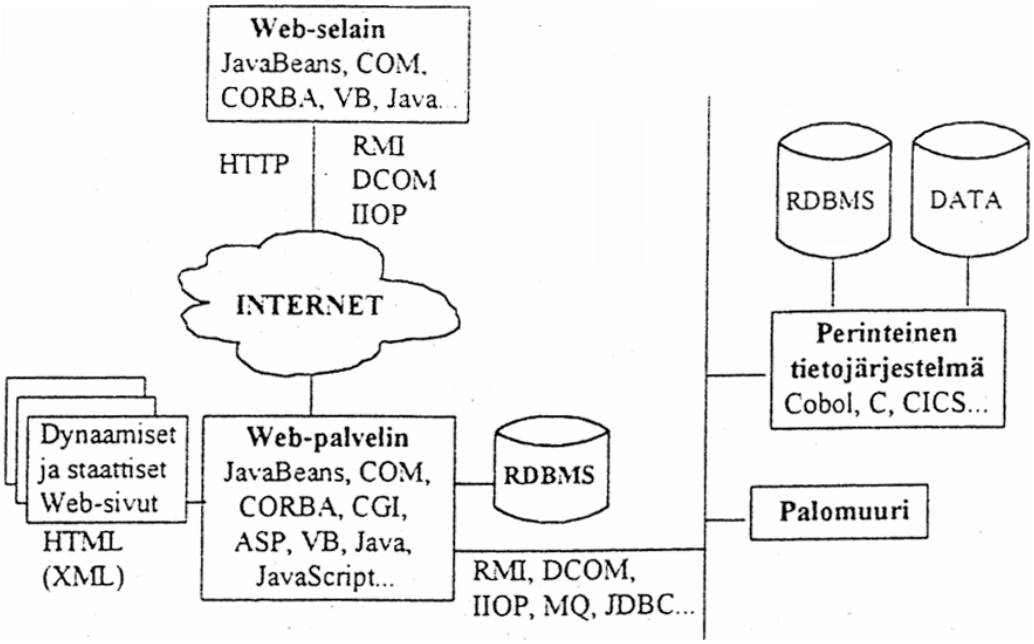
[7] http://en.wikipedia.org/wiki/Comparison_of_object_database_management_systems

[8] http://en.wikipedia.org/wiki/List_of_object-relational_database_management_systems

[9] http://en.wikipedia.org/wiki/List_of_object-relational_mapping_software

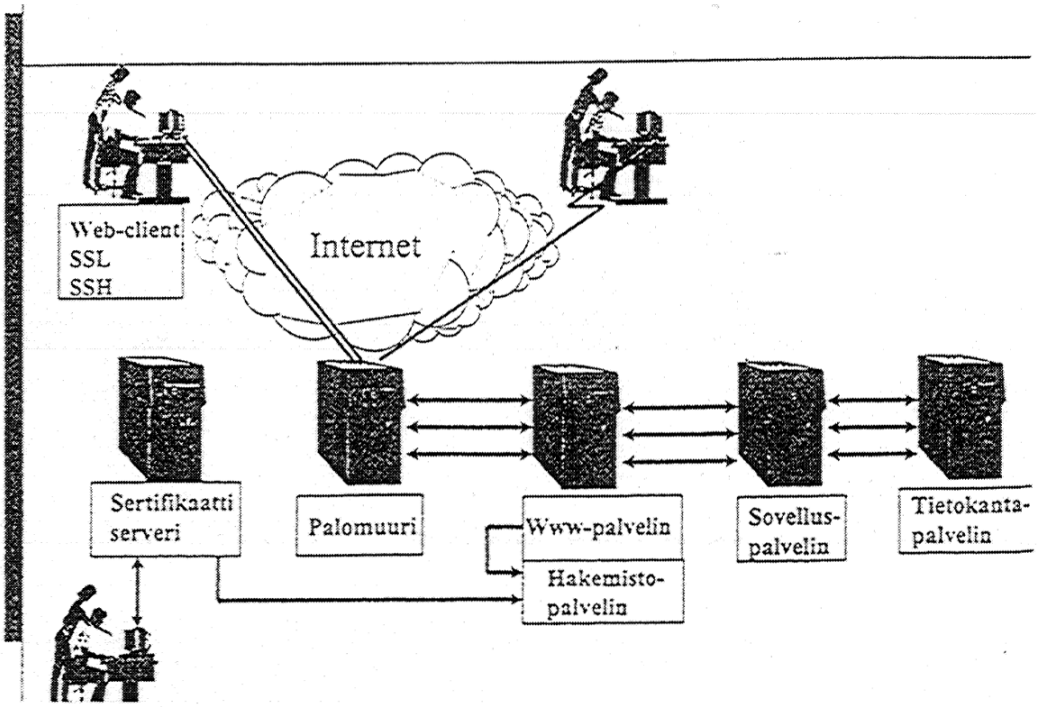


Kuva 1.6: Esimerkki kolmitasoarkkitehtuurin toteutuksesta

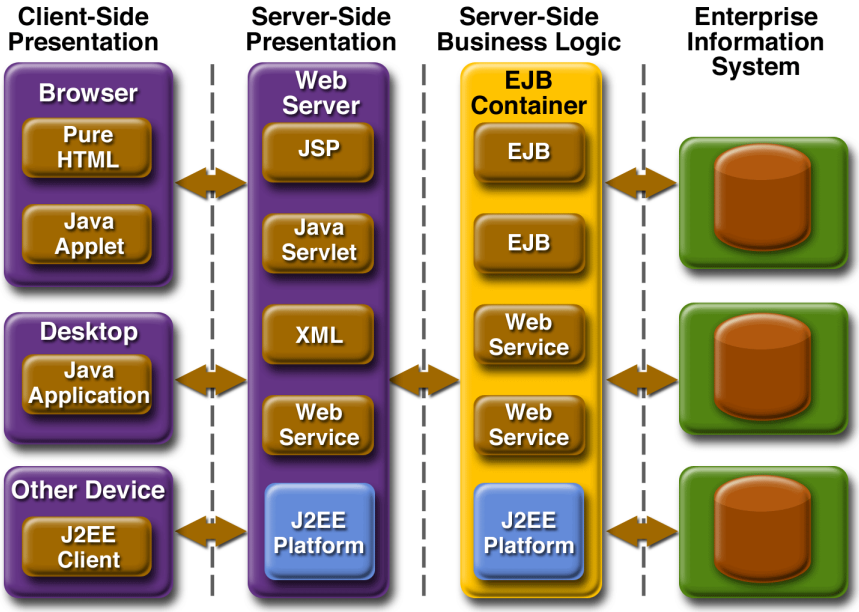


Kuva 1.7: Esimerkki internet-arkkitehtuurista ja teknisistä vaihtoehdoista

Web-perusteisissa järjestelmissä käytetään usein n-tasoista arkkitehtuuria, jossa n (> 3) tarkoittaa tasojen määrää ja verkkoa (**n**etwork). Kuvissa 1.7-1.9 on havainnollistettu sitä, miten monista verkon toisiinsa kytkemistä osista järjestelmä voidaan koostaa. Oliokeskeisyys omalta osaltaan edesauttaa osien yhteentoimivuutta (interoperability).



Kuva 1.8: Esimerkki internet-arkkitehtuurista ja teknisistä vaihtoehdoista



Kuva 1.9: Java Platform, Enterprise Edition (Java EE) (Sun Microsystems, Inc.)

Kuvassa 1.9 esitetään Java Platform, Enterprise Edition (Java EE, aikaisemmalta nimeltään J2EE), joka on hajautettujen monikerrosarkkitehtuuriin perustuvien Java-ohjelmointikielellä toteutettavien sovellusten kehittämistä ja suorittamista tukeva ohjelmointialusta. Java EE:n mukaiset sovellukset kehitetään pääosin modulaarisia, sovelluspalvelimella suoritettavia komponentteja hyödyntäen. Sovelluskehittäjät voivat kehittää tehokkaasti skaalautuvia ja monilla alustoilla toimivia yrityssovelluksia, jotka integroituvat perinteisiin tietojärjestelmiin ja keskittyä ensisijaisesti liiketoimintalogiikan ymmärtämiseen, kuvaamiseen ja toteuttamiseen sovelluspalvelimen tarjotessa pal-

velut transaktioiden hallintaan, tietoturvaan, komponenttien hallintaan ja muihin matalamman tason tehtäviin. Ohjelmointialusta ja sen määrittävä spesifikaatio kehittyvät jatkuvasti eri intressiryhmien verkostomaisena yhteistyönä (Java Community Process). Sen toteuttava lähdekoodi on julkisesti ja ilmaiseksi saatavissa. Spesifikaatio muodostaa samalla epävirallisen standardin, jonka mukaisesti intressiryhmien on toimittava voidakseen määrittää tuotteensa ja palvelunsa Java EE -yhteensopiviksi.

1.5 Oliolähestymistavan arviointia

Oliolähestymistavan todellisia/väitettyjä etuja:

- vastaa ihmisen luonnollista hahmotus- ja ajattelutapaa
- tukee kehittämistyötä myös uusilla sovellusalueilla (reaaliaikasovellukset, CAD/CAM, animaatio, simulointi, teollisuustuotannon järjestelmät ym.)
- helpottaa kokonaisuuden jakamista yhteensopiviksi osiksi (interoperability), jotka edustavat sovellusmaailman olioita
- pakottaa tarkastelemaan yhtä aikaa tietoja ja niiden käsittelyä olioina,
- olioluokat tarjoavat luonnollisen lähtökohdan ohjelmiston toteutukselle,
- takaa aidosti oliopohjaiset toteutukset (vrt. C++, CLOS, oliokannat)
- helpottaa jäljitettävyyttä (traceability), so. tietyn ratkaisun “syiden ja seurausten” selvittämistä,
- helpottaa uudelleenkäytettävien (reusable) ohjelmaosien tuottamista,
- helpottaa ohjelmistojen laajentamista ja ylläpitoa

Vaikeuksia oliolähestymistavan soveltamisessa:

- suurimman hyödyn saamiseksi lähestymistapaa on sovellettava laajassa mitassa (“to reduce a need of deobjectification and objectification”)
- rakenteisen lähestymistavan perinteen painolasti (osaaminen, aiemmat sovellukset (legacy systems), dokumentit, menetelmät, kehitysympäristöt, tottumukset)
- vaarana liian aikainen keskittyminen koodiin ja sen tuottamiseen eikä sitä edeltävään analyysiin ja suunnitteluun,
- käsitteiden vakiintumattomuus
- vaihtoehtoisten menetelmien kirjo (UML:n aseman vahvistuminen on helpottanut tilannetta)
- kehittymättömät organisointimuodot.

Olioajattelun hyötyjen on todettu kertyvän sitä varmemmin,

- mitä perusteellisemmin ja laaja-alaisemmin ajattelu on onnistuttu viemään läpi koko organisaatiossa,
- mitä paremmin opitaan ymmärtämään oliolähestystavan perusajatus (vrt. oppimiskäyrä)
- mitä useammin päästään uudelleenkäyttämään aiemmin määriteltyjä olioluokkia myöhemmissä projekteissa.

2 OLIOKESKEISET KEHITYSMENETELMÄT

1980-luvun loppupuolella ja 1990-luvun alussa julkaistiin kymmeniä oliomenetelmiä. Myöhemmin joukosta alkoi erottua muutamia kaupallisesti menestyviä menetelmiä, näiden joukossa OMT (Rumbaugh ym, 1991). OMT:n suosiota on selitetty muiden muassa sillä, että sitä käsittelevässä kirjassa on kuvattu havainnollisesti ja selkeästi vaihejako ja se sisälsi joitakin rakenteisen lähestymistavan puolella tutuksi tulleita malleja. Konvergenssi menetelmien joukossa jatkui; syntyi UML-kieli kolmen merkittävän menetelmän kehittäjän ("three amigos": Booch, Jacobson ja Rumbaugh) toimesta, sekä UML-kieltä hyödyntävä kehitysmenetelmä Rational Unified Process (Jacobson ym, 1999; Kruchten 2000).

Tällä kurssilla pyritään esittelemään yhtenäisen oliomenetelmän piirteitä painottuen ohjelmistoprosessin toteutusta edeltäviin vaiheisiin. Sen pohjana käytetään UML-notaatiota ja ICONIX-prosessia. Yksittäisten vaiheiden toiminnoissa hyödynnetään myös RUP:n, OMT:n ja OMT++:n tekniikoita.

2.1 Menetelmistä yleisesti

Menetelmällä tarkoitetaan tässä yhteydessä (deskriptiivistä ja preskriptiivistä) kuvausta organisationaalisten ja teknisten muutosten suorittamisesta organisaation tietojenkäsittelyssä tehokkaasti ja laadukkaasti. Menetelmä rakentuu osista, joita yhdistävät yleiset filosofiset ja paradigmaattiset olettamukset sekä määritellyt lähestymistavat.

Henderson-Sellersiä (1995) mukaillen voidaan erottaa seuraavat menetelmän osat:

- täydellinen vaihejakomalli
  - ◊ sovellettu prosessimalli (esim. waterfall, prototyping, incremental, spiral)
  - ◊ toiminnot koko elinkaaren osalta
  - ◊ toimintojen ajalliset suhteet
- keskenään johdonmukaiset käsitteet ja mallit
- kokoelma sääntöjä ja ohjeita
- vaadittujen tulosten kuvaus
  - ◊ mitä tulee tuottaa (esim. toiminto/tietokeskeisyys) ja milloin (checkpoints)
- mukailtava notaatio (määrämuotoisuus: formaali/ei-formaali)
- metriikat sekä laatua, standardeja ja testausstrategioita koskevia ohjeita
- ohjeita projektin hallintaan
- ohjeita luokkakirjaston hallintaan ja uudelleenkäyttöön
- roolikuvaukset (esim. osallistumisvastuu)

Menetelmää noudattamalla voidaan kehittämistyö saada määrämuotoisemmaksi, toistettavaksi ja kokemukset aiemmista hankkeista voidaan saattaa näkymään ja tuntumaan uudistetun menetelmän muodossa luotettavammin ja tehokkaammin.

Oliolähestymistavan näkökulmasta tietojärjestelmien kehittämismenetelmät voidaan karkeasti jakaa kahteen ryhmään (Jacobson, 1992, Coleman ym. 1994):

- rakenteista kehittämistä tukevat menetelmät (esim. SDM, SSADM, SA/SD, SADT ym.);
  - ◊ perustuvat joko toimintojen ja tietovirtojen mallintamiseen (toimintoperusteiset) tai käsiteltävän tiedon sisällön ja rakenteen mallintamiseen (tietoperusteiset)
  - ◊ toiminnot edustavat aktiivista, tieto passiivista puolta
  - ◊ tekevät eron analyysin, suunnittelun ja toteutuksen välillä,
- oliopohjaista kehittämistä tukevat menetelmät (esim. OMT, OOA, OOD, OOSE, Fusion, RUP/UML, OPEN ym.);
  - ◊ perustuvat olioiden mallintamiseen, joissa yhdistyvät sekä tieto että toiminnot,
  - ◊ sisältävät seuraavia perustehtäviä: olioluokkien tunnistaminen, olioluokkien organisointi rakenteeksi, olioiden vuorovaikutuksen kuvaaminen, olioluokkien operaatioiden kuvaaminen, olioluokkien sisäisen rakenteen kuvaaminen
  - ◊ ero analyysin, suunnittelun ja toteutuksen välillä hämärtynyt (Martin, Odell, 1992).

Oliomenetelmät voidaan jakaa edelleen kahteen luokkaan (Berard, 1993):

- *revolutionaariset* menetelmät: esim. Booch (1992)
  - ◊ perustuvat suuressa määrin olio-ohjelmointikielten käsitteisiin ja rakenteisiin (C++, Smalltalk)
  - ◊ monimuotoisuus ja periytyminen keskeisiä, samoin tiedon suojaus
  - ◊ käyttävät abstrakteja luokkia, parametrisoituja luokkia ja/tai metaluokkia,
  - ◊ kehittämistyön alusta alkaen järjestelmää tarkastellaan yhtenäisenä kokoelmana toisiinsa kiinteästi liittyneitä olioita,
  - ◊ oliot voivat olla aktiiveja,
  - ◊ tekevät selvän eron luokan ja ilmentymän välillä
  - ◊ luokilla on operaatioita mm. ilmentymien luomiseen ja tuhoamiseen
  - ◊ esim. Booch, 1991, Berard, 1993, Wirfs-Brock ym., 1990
- *evolutionaariset* menetelmät:
  - ◊ käyttävät graafisia kuvaustekniikkoja, joista monet tunnetaan myös rakenteisissa menetelmissä (esim. tietovirtakaaviot, ER-mallit)
  - ◊ käsittelevät olioita ikään kuin ne olisivat tietoa
  - ◊ mallintavat olioita tiedonmallintamisperiaatteilla (vrt. normalisointi)
  - ◊ pitävät olioita staattisina
  - ◊ tiedon suojaus toisarvoista
  - ◊ kytkevät oliot kiinteästi tietovaraston käsitteeseen
  - ◊ esim. Rumbaugh (1991), Shlaer, Mellor (1988), Martin, Odell (1995), Fusion (Coleman ym., 1995), Embley, Kurtz (1992)

1990-luvun jälkeen varteenotettavien oliomenetelmien määrä on vähentynyt fuusioiden, liittoutumien ja parhaiden käytäntöjän kokoamisen myötä. Notaation tasolla edelleen kehittyvä UML (Unified Modeling Language, Booch, Jacobson, Rumbaugh, 1999) on muodostunut standardiksi, jota USDP, RUP ja monet muut menetelmät käyttävät. 2000-luvun uusista kehitysmenetelmistä ja lähestymistavoista merkittäviä ovat mm. ketterät menetelmät (agile methods) ja mallipohjainen kehitys (Model Driven Engineering).

- Ketterät menetelmät (esim. XP (Extreme Programming), SCRUM, joissakin yhteyksissä myös RUP) korostavat kommunikointia, nopeita julkaisuja, asiakaskeskeisyyttä ja reagointia vaatimusten muuttumiseen vastakohtana joidenkin perinteisten menetelmien raskaudelle esim. prosessin tai dokumentoinnin suhteen[10]. UML:n käyttö soveltuu ketterien menetelmien osaksi[11], kuitenkin hyöty/kustannusperiaatteella valikoiden, esim. tiettyihin ongelmakohtiin keskittyen ja kommunikoinnin apuna – ei välttämättä kattavana dokumentaationa.
- Mallipohjaisen kehityksen (Model Driven Engineering; esim. Domain Specific Modeling, MDA (Model Driven Architecture), Software Product Lines) tavoitteena on minimoida perinteisen ohjelmoinnin määrä sisällyttämällä suurin osa järjestelmän logiikasta malleihin, jotka käännetään suoraan koodiksi. Mallinnuskielenä voi olla esim. UML (MDA-lähestymistavassa) tai kohdealuepohjainen malli (DSM). Mallipohjaisesta kehityksestä on lupaavia esimerkkejä rajatuilla sovellusalueilla (esim. matkapuhelinten käyttöliiittymät[12]) ja tuoteperheissä, mutta käytännössä MDE vaatii erittäin edistyneitä mallinnustyökaluja ja tarkkoja malleja, eikä sitä käsitellä tällä kurssilla tarkemmin.

Käytettyjen menetelmien määrä on vähentynyt myös siksi, että tietojärjestelmiä ja ohjelmistotuotteita kehitetään nykyään yhä verkostoituneemmassa globaalissa ympäristössä, jossa järjestelmäintegraattorit tuottavat asiakkailleen entistä kokonaisvaltaisempia ja modulaarisempia ratkaisuja ja niiden partnerit ja alihankkijat toteuttavat järjestelmien osia ympäri maailmaa. Sekä ratkaisuja että niiden toteuttamiseen tarvittavia menetelmiä ja välineitä säätelevät myös yhä monimutkaisemmat sisäiset ja ulkoiset normit. Verkostoitunut järjestelmäkehitys onkin erittäin vaikeaa, mikäli eri toimijat käyttävät eri menetelmiä ja työkaluja. Menetelmien kansainvälisen standardoinnin merkitys korostuu. Työkalujen on nojattava entistä enemmän avoimiin rajapintoihin, jotta menetelmien edellyttämän tiedon välitys, varastointi, ja käsittely työkalujen välillä helpottuu ja voidaan rakentaa integroituja kehitysympäristöjä tietojärjestelmien ja tuotteiden hallitsemiseen läpi niiden elinkaarien.

2.2 RUP-menetelmä ja UML-kieli

RUP (Rational Unified Process) (Kruchten 2000) on UML-kieltä käyttävä kehitysmenetelmä. Menetelmään on pyritty kokoamaan parhaita käytäntöjä eri menetelmistä mahdollisimman yleiskäyttöisessä muodossa. RUP luetaan inkrementaalisen ja iteratiivisen kehitysmallinsa vuoksi joskus ketteriin menetelmiin, mutta menetelmän laajuuden vuoksi ”ketteryyden” toteutuminen voi vaatia sovittamista ja joustavaa organisaatiokulttuuria. RUP pohjautuu seuraaville perusperiaatteille:

1. Iteratiivinen kehitys
2. Vaatimusten hallinta
3. Komponenttipohjaiset arkkitehtuurit
4. Visuaalinen mallinnus
5. Jatkuva ohjelmiston laadunvarmistus
6. Muutosten hallinta

[10] Ks. Agile Manifesto, http://agilemanifesto.org/

[11] Ks. esim. Agile Modeling, http://www.agilemodeling.com/

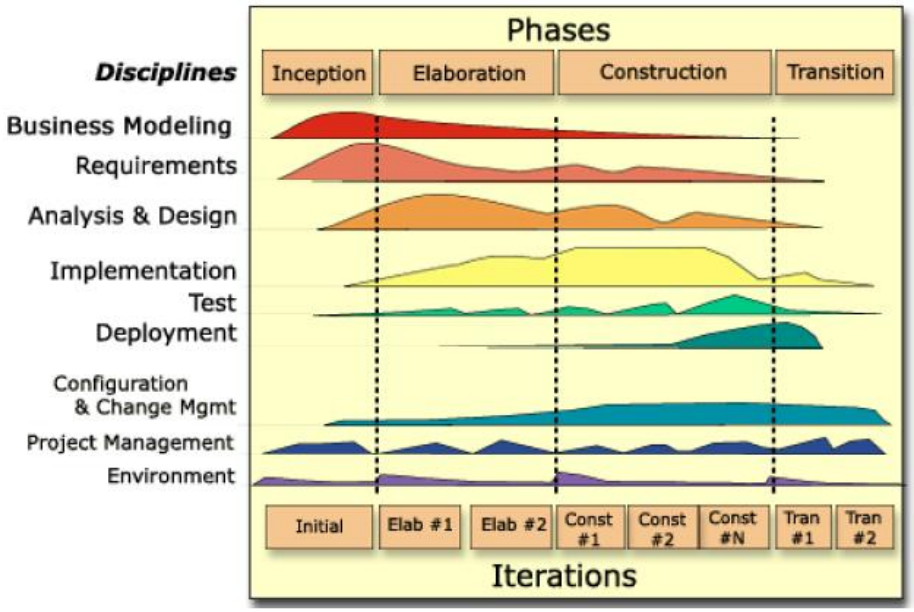
[12] Juha-Pekka Tolvanen, Domain-Specific Modeling in Practice, http://tfs.cs.tu-berlin.de/gtvmt08/Program/DSMinPractice_Tolvanen_29March2008.pdf

RUP-menetelmän mukaisissa dokumenteissa käytetään pääosin UML-kieltä, ja RUP:ia edeltävän USDP-menetelmän (Unified Software Development Process) kehittäjät ovatkin samat kuin UML:llä. Tästä huolimatta UML-kieli ei ole RUP-menetelmään sidottu eikä itsessään ota kantaa prosessiin tai kehitysmenetelmään.

RUP-menetelmän ja UML-kielen historiaa:

- 1988: Jacobsonin Objectory-prosessin ensimmäinen versio ja OOSE-notaatio - sis. mm. käyttötapaukset (Jacobson 1992)
- 1980-luvun lopulla General Electric Research and Development Center:issä kehitetty OMT (Object Modeling Technique) (Rumbaugh ym, 1991) kattaa analyysin, suunnittelun ja osan toteutusta
- 1991: Boochin menetelmä (Booch 1991)
- 1994-95 vaihteessa Rumbaugh siirtyi Grady Boochin Rational-yritykseen ja vuonna 1996 heidän joukkoonsa liittyi myös Jacobson,
- 1995-96: Unified Method versio 0.8 ja UML (Unified Modeling Language) 0.9 standardoinnin pohjaksi
- 1998: OMG (Object Management Group) hyväksyi UML:n (Unified Modeling Language) standardiksi
- 1999: UML 1.3 (Booch, Rumbaugh, Jacobson, 1999) ja Unified Software Development Process (USDP) (Jacobson, Booch, Rumbaugh, 1999)
- 2000: Rational Unified Process (Kruchten 2000)
- 2005: UML 1.4.2 hyväksyttiin kansainväliseksi ISO 19501 -standardiksi. UML 2.0 julkaistiin.

Kuva 2.1 havainnollistaa RUP-prosessin etenemistä. Menetelmä voidaan jakaa kolmen tekijän mukaan seuraavasti (Kruchten 2000).



Kuva 2.1: RUP-tehtäväkokonaisuudet suoritetaan iteratiivisesti vaiheissa

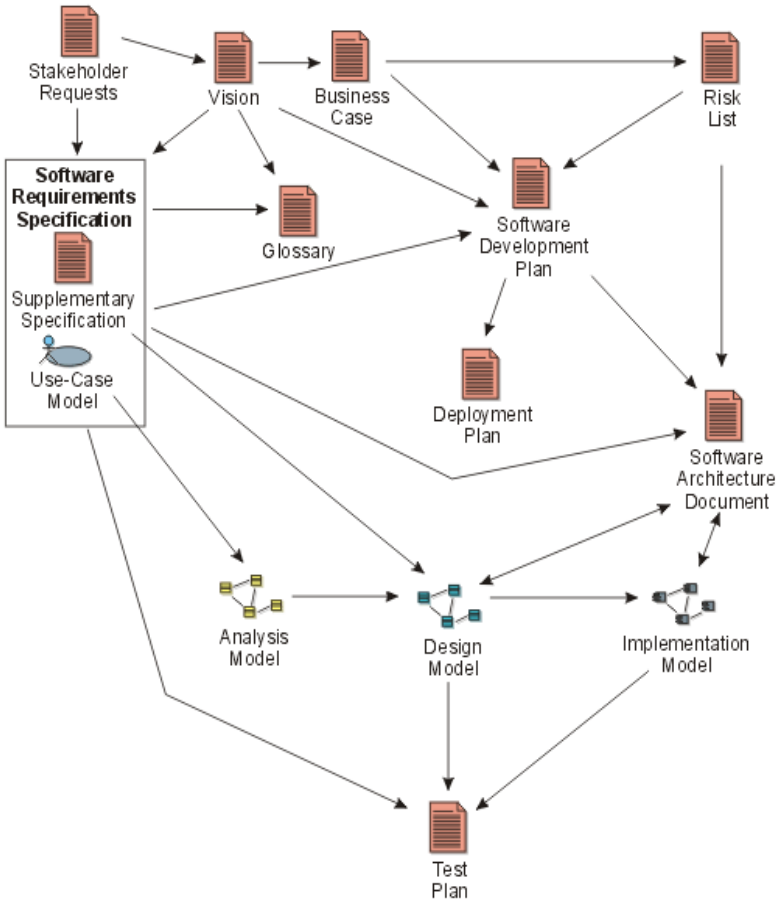
1. *Tehtäväkokonaisuudet* (core workflows):
- Kehitystehtävät
  - ◊ **Liiketoiminnan mallinnus** (business modeling): jäsennetään kehityksen kohteena olevien prosessien nykytila ja priorisoidaan kehityskohteet
  - ◊ **Vaatimusten määritys** (requirements): kerätään, kootaan ja määritetään toiminnalliset ja ei-toiminnalliset vaatimukset järjestelmälle.
  - ◊ **Analyysi** (analysis): Tarkennetaan ja jäsennetään edellä koottuja vaatimuksia, jotta voidaan muodostaa ymmärrettävä, kattava ja ylläpidettävä kuvaus järjestelmästä (MITÄ järjestelmän tulee tehdä?).
  - ◊ **Suunnittelu** (design): tarkennetaan ja työstetään tuotettua analyysimallia siten, että pystytään vastaamaan kysymykseen "MITEN järjestelmä vaatimukset tyydyttää?".
  - ◊ **Toteutus** (implementation): toteutetaan suunnittelun tuottamat kuvaukset (suunnitteluvaiheen malli) käyttäen komponentteja, ohjelmointikieliä, tietokantateknologiaa yms
  - ◊ **Testaus** (test): varmistetaan jokaisen toteutetun osan toimivan vaatimusten mukaisella tavalla.
  - ◊ **Sijoitus** (deployment): ohjelmiston julkaisu, paketointi ja jakaminen loppukäyttäjille
- Tukitehtävät
  - ◊ **Konfiguraation- ja muutostenhallinta** (configuration & change management): versiohistorian ja rinnakkaisten julkaisujen hallinta, muutospyyntöjen (esim. bugiraportit) käsittely
  - ◊ **Projektinhallinta** (project management): vaiheiden ja iteraatioiden suunnittelu, riskienhallinta, projektin seuranta jne
  - ◊ **Ympäristö** (environment): työkalujen valinta, ohjelmistoprosessin mukauttaminen, mikrotuki, laitteet ja muu infrastruktuuri

2. *Vaiheet*:
- Aloitusvaihe (inception):
  - ◊ kiinnitetään ja rajataan liiketoiminta-alue ja perusprosessit (liiketoiminnan mallintaminen),
  - ◊ määritetään kriittiset menetystekijät ja riskit, hahmotellaan perusarkkitehtuuri, arvioidaan resurssitarve ja tehdään projektisuunnitelma,
  - ◊ tehdään mahdollisesti prototyyppi,
  - ◊ lopuksi päätetään, kannattaako kehittämistä jatkaa.
- Tarkentamisvaihe (elaboration):
  - ◊ analysoidaan ongelma-alue,
  - ◊ määritetään vaatimukset,
  - ◊ tarkennetaan perusarkkitehtuuria,
  - ◊ priorisoidaan järjestelmäosia,
  - ◊ tehdään yksityiskohtaisempi projektisuunnitelma.
- Konstruointivaihe (construction):
  - ◊ Toteutetaan ja testataan suunnitellut ratkaisut
- Luovutusvaihe (transition):
  - ◊ luovutetaan järjestelmä ensin beta-versiona ja myöhemmin tuotantoversiona käyttäjien käyttöön,
  - ◊ tehdään välttämättömät korjaus- ja ylläpitotehtävät
  - ◊ tehdään päätös projektin päättämisestä
  - ◊ kootaan kokemukset ja korjataan niiden mukaisesti prosessimallia.

3. *Iteraatiot* (iterations):
   - Kunkin vaiheen sisällä voi olla useampia iteraatiokierroksia, joiden aikana vaatimuksia, kuvauksia, suunnitelmia ja toteutuksia tarkennetaan ja tarpeen mukaan muutetaan.
   - Iteraatioiden tarve on tapauskohtainen.

RUP-prosessin tuotoksia kutsutaan artefakteiksi, jotka voivat olla malleja (käyttötapausmalli, suunnitteluvaiheen malli, toteutusmalli, testimalli jne), dokumentteja, lähdekoodia tai ajettavia ohjelmia (kuva 2.2; Kruchten 2000). Artefaktit on jaettu tehtäväkokonaisuuksien seuraaviin tietojoukkoihin *(information sets)*:

- *Hallintakokonaisuus* (management set) sisältää liiketoimintaan, ohjelmistoprosessiin ja projektinhallintaan liittyvät artefaktit
- *Vaatimuskokonaisuus* (requirements set) sisältää vaatimukset, käyttötapaukset ja kehitettävään järjestelmään liittyvien liiketoimintaprosessien kuvaukset
- *Suunnittelukokonaisuus* (design set) sisältää suunnitteluvaiheen mallin, arkkitehtuurikuvauksen ja testimallin
- *Toteutuskokonaisuus* (implementation set) sisältää lähdekoodin, ajettavat ohjelmat ja muun järjestelmän ajamiseen liittyvän datan
- *Sijoituskokonaisuus* (deployment set) sisältää järjestelmän asennukseen ja käyttöönottoon liittyvän materiaalin

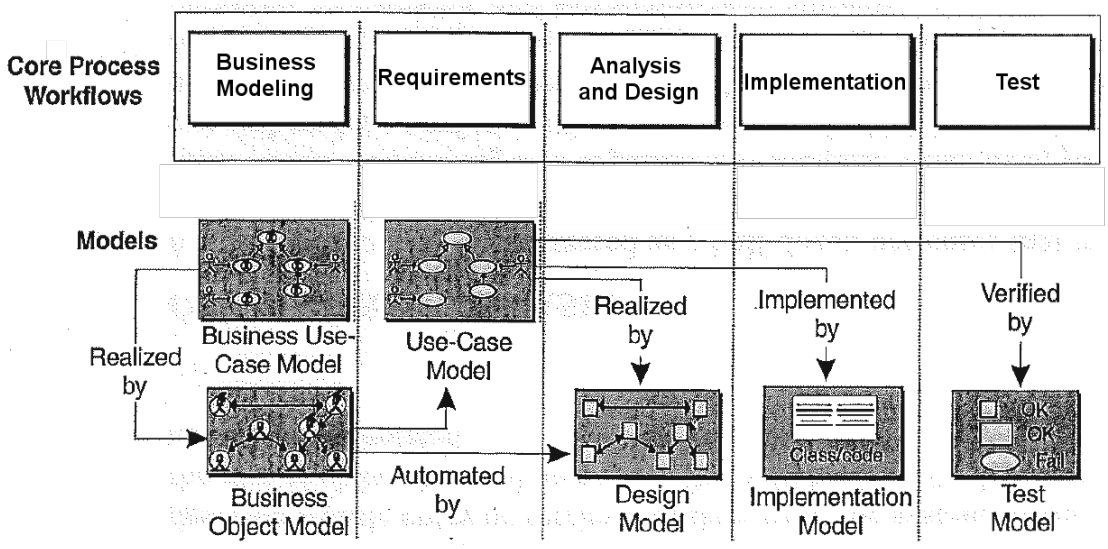


Kuva 2.2: RUP-prosessin tärkeimmät tuotokset (artefaktit).

Kehittämisen kulun konkretisoimiseksi voidaan esittää myös mallit, joita eri yhteyksissä tuotetaan (Jacobson 1999):

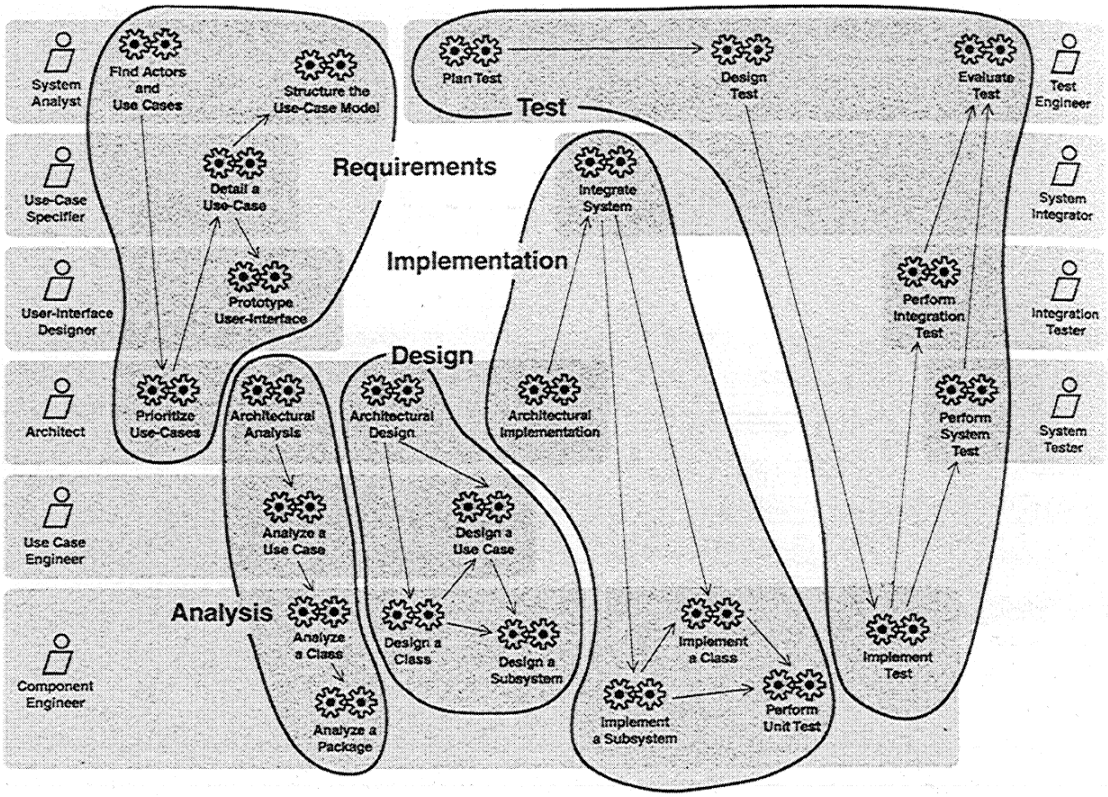
- *Liiketoimintakäyttötapausmalli* (business use case model) kuvaa liiketoimintaa tekevät aktorit ja heidän tavoitteensa liiketoimintaprosessien suorittamisen ja ohjaamisen näkökulmista sekä prosessit ja niiden tuottamat palvelut asiakkaille. Yhteen malliin voi liittyä useita käyttötapausmalleja, joista kukin kuvaa yhden järjestelmän (sovelluksen) toimintaa.
- *Käyttötapausmalli* (use case model) kuvaa järjestelmän käyttäjät ja käyttötapaukset (so. toiminnot, joita järjestelmän tuella voidaan suorittaa),
- *Suunnitteluvaiheen malli* (design model) kuvaa järjestelmän teknisen rakenteen ja toiminnan. Tarvittaessa osa mallista voidaan jakaa erilliseen *analyysimalliin* (analysis model), joka tarkentaa käyttötapausmallia niin järjestelmän rakenteen (luokkakaavio, class diagram) kuin käyttäytymisenkin (sekvenssikaavio, sequence diagram; yhteistoimintakaavio, collaboration diagram) osalta,
- Sijoitusmalli (deployment model) kuvaa järjestelmän solmut (laitteet) ja järjestelmän ohjelmistokomponenttien sijoittelun niihin,
- Toteutusmalli (implementation model) kuvaa järjestelmän ohjelmistokomponentit ja niiden sisällön olioluokkina,
- Testimalli (test model) kuvaa testaukset ja niiden suhteet käyttötapauksiin.

RUP-menetelmä on käyttötapauslähtöinen (kuva 2.3; Kruchten 2000). Käyttötapaukset toimivat yhteisenä kielenä asiakkaiden ja järjestelmänkehittäjien välillä. Analyysi- ja suunnitteluvaiheissa käyttötapaukset toimivat pohjana dynaamisille malleille, jotka kuvaavat olioiden vuorovaikutusta. Käyttötapausten läpikäynti auttaa myös olioiden ja luokkien etsimistä staattisissa malleissa. Suunnitteluvaiheen malli toimii spesifikaationa toteutukselle. Testauksen aikana käyttötapaukset toimivat pohjana testitapauksille. Varsinaisten UML-mallien lisäksi käyttötapausten kirjoitustapa mahdollistaa niiden käytön käyttöohjeiden osana.



Kuva 2.3: Käyttötapaukset ovat pohjana monille RUP-menetelmän malleille.

Yksityiskohtaisimmillaan prosessia voidaan havainnollistaa kuvan 2.4 (Jacobson ym. 1999, s. 324) tavoin, jolloin osoitetaan myös roolihenkilöiden vastuut eri tehtäväkokonaisuuksien/kuvausten avulla.



Kuva 2.4: RUP-roolit on kuvattu vasemmalla ja oikealla pystysarakkeella. Kullekin työroolille määritellään oma ”uimarata”, johon määritellyistä tehtävistä roolin haltija on vastuussa. Eri vaiheisiin liittyvät aktiviteetit on ryhmitelty. Esimerkiksi komponenttisuunnittelija analysoi luokkia ja paketteja analyysikokonaisuudessa, suunnittelee luokkia ja osajärjestelmiä suunnittelukokonaisuudessa, toteuttaa niitä toteutuskokonaisuudessa, ja tekee yksikkötestausta testauskokonaisuudessa.

RUP:n mallit esitetään UML-kielellä, josta keskeisimmät kaaviot (ja arkkitehtuurinäkymät, joihin kaaviot ensisijaisesti liittyvät) on listattu kuvassa 2.5. UML-kieli sisältää laskutavasta riippuen 13 erilaista kaaviota, joilla eri malleja ja arkkitehtuurinäkymiä esitetään. UML-kieli itsessään ei ole RUP-menetelmästä riippuvainen ja on notaationa käytössä useissa muissakin kehitysmenetelmissä.

Major Area	*View*	*Diagrams*	*Main Concepts*
structural	static view	class diagram	class, association, generalization, dependency, realization, interface
	use case view	use case diagram	use case, actor, association, extend, include, use case generalization
	implementation view	component diagram	component, interface, dependency, realization
	deployment view	deployment diagram	node, component, dependency, location
dynamic	state machine view	statechart diagram	state, event, transition, action
	activity view	activity diagram	state, activity, completion transition, fork, join
	interaction view	sequence diagram	interaction, object, message, activation
		collaboration diagram	collaboration, interaction, collaboration role, message
model management	model management view	class diagram	package, subsystem, model
extensibility	all	all	constraint, stereotype, tagged values

Kuva 2.5: UML 1.4 - näkymät ja kaaviot. UML 2:ssa on lisäksi interaktio-yleiskaavio ja ajoituskaavio. Staattista näkymää voidaan mallintaa lisäksi luokkakaavion tyylisillä olio-, paketti- ja rakennekaavioilla.

2.3 OMT-menetelmä

OMT (Object Modeling Technique)-menetelmä oli suosittu oliosuunnittelumenetelmä ennen RUP:n ja UML:n yleistymistä. Osa UML-kielen kaavioista perustuu OMT:n vastaaviin. OMT:n (Rumbaugh ym., 1991) vaihejako koostuu seuraavista vaiheista:

- *Analyysi*:
  - ◊ MITÄ järjestelmän pitäisi tehdä?
  - ◊ luodaan kuvaus rakenteista, toiminnoista ja toimintaympäristöstä
  - ◊ keskitytään ongelma-alueeseen
  - ◊ riippumaton toteutusympäristöstä
  - ◊ peruskäyttäjiltä edellytetään aktiivista osanottoa
  - ◊ lopputuloksena:
    - ongelman kuvaus
    - luokkakaavio ja tietohakemisto
    - tilakaaviot ja sekvenssikaaviot, tietovirtakaaviot

- *Järjestelmän suunnittelu:*
  - ◊ arkkitehtuurin suunnittelua
  - ◊ järjestelmän ositus osajärjestelmiksi
  - ◊ luokkakirjastojen käyttö
  - ◊ rajoitteet toteutukselle
  - ◊ lopputuloksena:
    - järjestelmän arkkitehtuuri ja ositus osajärjestelmiksi
- *Yksityiskohtainen suunnittelu (oliosuunnittelu):*
  - ◊ MITEN kukin osajärjestelmä toteutetaan?
  - ◊ sidotaan toteutusympäristön välineisiin
  - ◊ suunnittelumalli rakennetaan lisäämällä analyysimalliin yksityiskohtia:
  - ◊ tietorakenteiden ja algoritmien valinta
  - ◊ teknisten eli toteutusolioiden (käyttöliittymät, tiedonhallinta ym.) käyttöönotto
  - ◊ lopputuloksena:
    - tarkennetut luokkakaaviot
    - tarkennetut dynaamiset mallit
    - tarkennetut funktionaaliset mallit
- *Toteutus*
  - ◊ luokkien toteutus ohjelmointikielellä
  - ◊ rajapintamäärittelyt tulevat oliomallista
  - ◊ lopputuloksena:
    - luokkamääritykset, ohjelmat

Mallit yhdessä kattavat järjestelmän piirteet kokonaisuudessaan. Mallien välillä on monenlaisia yhteyksiä. Esimerkiksi oliomallissa esitetään kunkin luokan yhteydessä operaatiot. Dynaamisessa mallissa esitetään, miten oliot reagoivat vastaanottaessaan viestejä. Funktionaalisessa mallissa osoitetaan, miten ko. operaatiot muodostavat prosesseja, joka vaikuttavat järjestelmän tietojenkäsittelyyn. Kukin malli tarkentuu edettäessä analyysista suunnittelun kautta toteutusvaiheeseen. Ehdottomia määrityksiä sille, mitkä piirteet otetaan malleista käyttöön missäkin vaiheessa, ei ole esitetty. Menetelmä sisältää siis tilannekohtaista liikkumavaraa, eli se on *kontingentti* menetelmä.

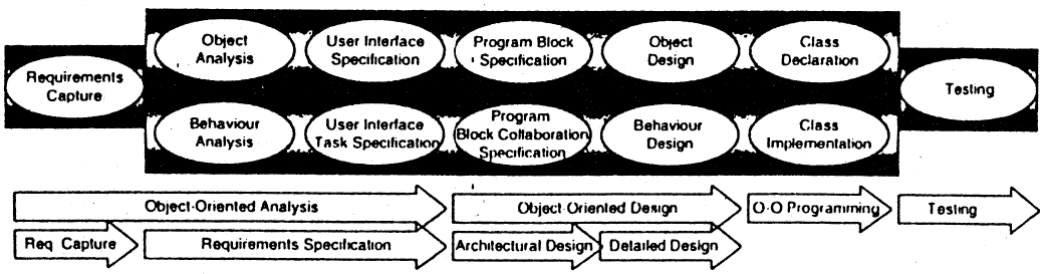
Liitteessä B on kuvattu alun perin OMT:llä mallinnettu pankkiautomaattijärjestelmä staattisesta ja dynaamisesta näkökulmasta (käyttötapausmalli on lisätty alkuperäiseen OMT-menetelmään nähden ja luokkakaaviot on muutettu UML:n mukaisiksi).

2.4 OMT++ -menetelmä

OMT++ on Nokian joissakin yksiköissä käytössä oleva oliomenetelmä. Se perustuu OMT- (Rumbaugh ym., 1991), OOSE- (Jacobson, 1992) ja Fusion-menetelmiin (Coleman, 1995).

OMT++:n rakenne on muodostettu kahden rinnakkaisen polun, staattisen mallintamisen ja dynaamisen (functional) mallintamisen varaan (kuva 2.6; Jaaksi, 1997, 32). Polkujen lähtökohtana on vaatimusten kerääminen ja päätepisteenä testaaminen. Kuvassa on esitetty, millaisia kehittämistehtäviä poluilla suoritetaan ja millaisin vaihein. Oliomallintaminen ja käyttäytymisen mallintaminen tuottavat tarkentuvia kuvauksia vaihe vaiheelta. Seuraavassa on luettelo kunkin vaiheen päätuotoksista:

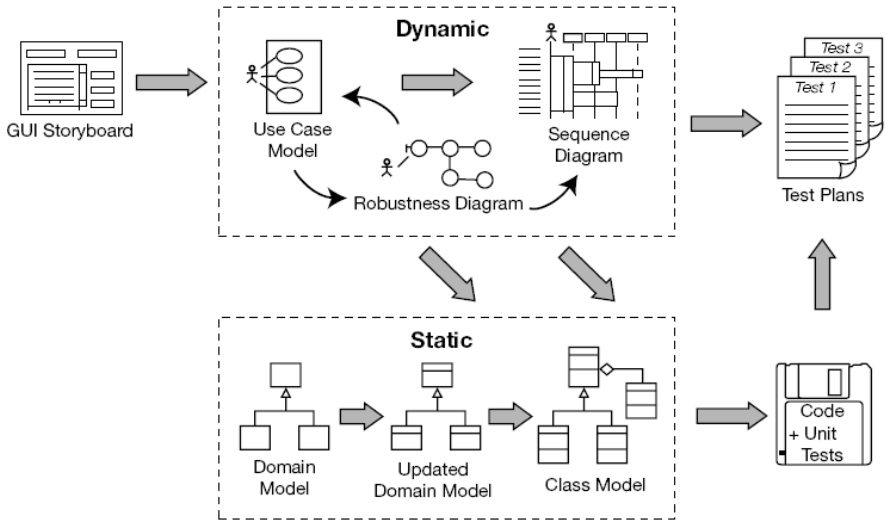
- vaatimusten keruu:
  - ◊ käyttötapaukset ja tekstikuvaukset
- määrittely (analysis):
  - ◊ määrittelyoliomalli
  - ◊ operaatiolista
  - ◊ käyttöliittymäkuvaus
- suunnittelu:
  - ◊ suunnitteluoliomalli
  - ◊ toimintokuvaukset
- ohjelmointi:
  - ◊ lähdekoodi



Kuva 2.6: Staattinen ja dynaaminen mallintaminen OMT++-menetelmässä

2.5 ICONIX

ICONIX (kuva 2.7) on Rosenbergin ym. (Rosenberg & Scott 1999; Rosenberg & Stephens 2007) kehittämä kevyt UML-kielen osajoukkoa käyttävä menetelmä. ICONIX on käyttötapauslähtöinen menetelmä, jossa mallintaminen on jaettu staattisiin ja dynaamisiin osiin. Menetelmän vahvuutena on tarkasti määritelty siirtymä analyysi- ja suunnitteluvaiheiden välillä, jota tuetaan UML-yhteistoimintakaavioiden variantilla (*robustness diagrams*).

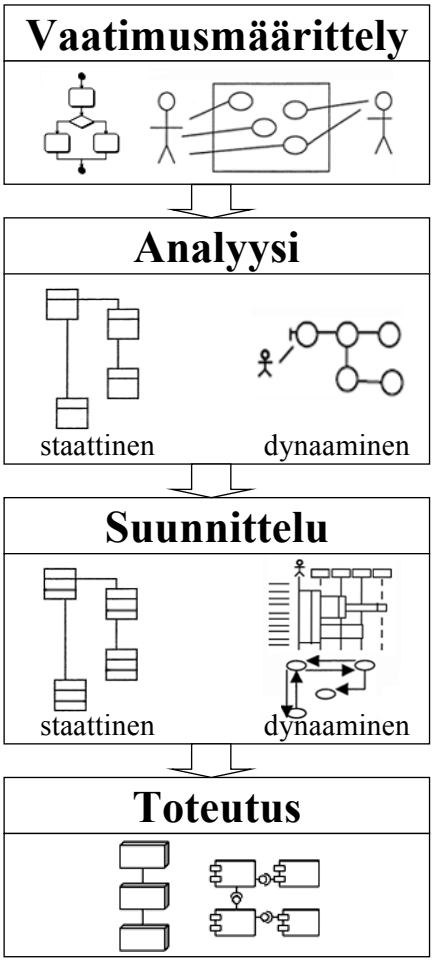


Kuva 2.7: ICONIX-prosessi

2.6 Oliomenetelmä tällä kurssilla

Seuraavissa luvissa esitellään oliokeskeisen tietojärjestelmien kehittämisprosessia pääosin RUP- ja ICONIX-menetelmien mukaisesti. Koska RUP jättää käyttöliittymän suunnittelun melko vähäiselle huomiolle, laajennetaan siltä osin tarkastelua OMT++:n ja Bennettin ym. (1999, 2006) mukaisesti. OMT-menetelmäkirjasta on tekstiin sisällytetty esimerkki pankkiautomaattijärjestelmästä. Eri vaiheiden tuotoksia havainnollistetaan luvuissa useilla esimerkeillä, joista merkittävimpänä toistuvana on Bennettin (2006) Agate-mainostoimistoesimerkki. Kunkin vaiheen yhteydessä määritellään keskeiset käsitteet, esitellään mallit notaatioineen sekä kuvataan eri kehitysmenetelmissä suositeltuja askeleita ja muita hyödyllisiksi havaittuja tekniikoita. Päävaiheet ovat (kuva 2.8):

1. Vaatimusmäärittely
2. Analyysi
3. Suunnittelu
4. Toteutus



Kuva 2.8: Tämän kurssin oliomenetelmän prosessimalli

Vaatimusten määrityksen osalta keskitytään erityisesti käyttötapausten kirjoittamiseen. Toteutuksen osalta ohjelmointia tai suurinta osaa muistakaan toteutuksen aktiviteeteista ei käsitellä, vaan rajoitutaan toteutusta ja ajonaikaista komponenttien sijoittelelua kuvaaviin UML-kaavioihin. Analyysiin ja suunnitteluun liittyvät asiat pyritään käsittelemään perusteellisesti.

3 VAATIMUSTEN MÄÄRITYS

Vaatimusten määrittämisen (requirements engineering) tarkoituksena on tuottaa kattava ja johdonmukainen kuvaus kehitettävän tietojärjestelmän toiminnallisista (*functional* – mitä järjestelmän tulisi tehdä) ja ei-toiminnallisista (*non-functional* – laadullisia vaatimuksia, joita ei voida johtaa suoraan ohjelman toimintoihin) ominaisuuksista. Ei-toiminnallisia vaatimuksia voidaan jakaa tarkemmin eri luokkiin, joita ovat mm.

- Käytettävyysvaatimukset (*usability*) (Bennett ym. nostavat tämän omaksi vaatimusluokakseen)
- Luotettavuusvaatimukset (*reliability*) – mm. tulosten oikeellisuus, vioista toipumisaika
- Suorituskykyvaatimukset (*performance*)
- Tuettavuus (*supportability*) – mm. ylläpidettävyys, testattavuus, konfiguroitavuus.
- Tietoturva

Toiminnalliset-, käytettävyys-, luotettavuus-, suorituskykyvaatimukset ja tuettavuusvaatimukset ovat HP:n FURPS-mallista[13], jolla voidaan luokitella ohjelmiston laatuattribuutteja.

Vaatimuksiin vaikuttavat (mutta eivät ole varsinaisesti vaatimuksia) myös järjestelmästä itsestään johtuvat *rajoitteet*, jotka voivat liittyä suunnitteluun, toteutukseen, laitteistoon, rajapintoihin tai toimintoihin. Rajoitteet voivat lähteä myös ympäristöstä (yritys, toimiala, sopimukset, standardit, lait ym.)

Määrityksen tulee olla kuvaus tarpeista, ei ratkaisuista. Toteutukseen viitataan vain epäsuorasti (ehdottomat suorituskykyvaatimukset, laitteistoalusta, sovellettavat standardit, tulevaisuuden laajennustarpeet yms). Seuraavassa kerrotaan lyhyesti vaatimusten määritysstrategioista, -menetelmistä ja eräistä keskeisimmistä tekniikoista.

3.1 Määritysstrategiat ja -tulokset

Tietojärjestelmien kehittämisen olennaisimpana tavoitteena on etsiä ratkaisuja todellisiin ongelmiin ja sellaisia ratkaisuja, joita todella halutaan. Käytännössä pidetään liian usein itsestään selvänä, että käyttäjiltä saadaan helposti heti aluksi täsmälliset ja lopulliset ongelma- ja tavoitelistat suunnittelun pohjaksi. Kysymys on kuitenkin huomattavasti monimutkaisemmasta ja hankalammasta asiasta. Määrittelytyö edellyttää monipuolista kokemusta oikean psykologisen, organisatorisen ja poliittisen lähestymistavan valitsemiseksi kulloiseenkin tilanteeseen.

Erityisen tärkeää on löytää kaikki järjestelmän keskeiset käyttäjä- ja muut intressiryhmät, joiden vaatimukset on huomioitava, sekä asiantuntevat edustajat näistä ryhmistä ja saada nämä ryhmät ja niiden edustajat sitoutumaan määrittelytyöhön koko järjestelmän kehittämisprojektin aikana. Näin järjestelmiä voidaan kehittää asiakasohjautuvasti suunnittelijoiden ja käyttäjäryhmien välisissä oppimisprosesseissa.

[13] http://en.wikipedia.org/wiki/Furps

Davisin and Olsonin (1985) mukaisesti voidaan erottaa neljä päästrategiaa vaatimusten määrittelylle:

1. Kyseleminen
   - ◊ Perustuu olettamukseen, että käyttäjät pystyvät jäsentämään ongelmakentän ja ilmaisemaan tarpeensa,
   - ◊ Sovelias tilanteissa, joissa on suhteellisen stabiili järjestelmä ja jäsentynyt ongelmakenttä,
2. Nykyisestä tietojärjestelmästä päättely
   - ◊ Tarkoituksena on ottaa oppia ratkaisuista, joita on tehty aiemmin ko. sovellusalueella,
   - ◊ Lähtökohtana voi toimia korvattava järjestelmä, samantapainen järjestelmä muualla, sovelluspaketti, käsikirjat tms.
   - ◊ Sovelias tilanteissa, joissa vakiintuneet toiminnot ja ympäristö
3. Hyväksikäyttävän järjestelmän piirteiden selvittäminen
   - ◊ Tarkoituksena on hyödyntävän järjestelmän piirteiden selvittämisen kautta rajata ja jäsentää ongelmakentää, johon vaatimukset voidaan sitten johdonmukaisesti johtaen "ankkuroida",
   - ◊ Sovelias tilanteissa, joissa hyväksikäyttävä järjestelmä on muutosaltis ja ei ole olemassa aikaisempaa ratkaisua,
4. Kokeilujen myötä (vrt. protoilu) vähitellen löytäminen
   - ◊ Tarkoituksena on tuottaa jo alkuvaiheessa jotain konkreettista, jonka avulla käyttäjät voivat määrittää luotettavammin vaatimuksiaan,
   - ◊ Soveltuu tilanteisiin, joissa käyttäjiltä ja/tai suunnitelijoilta puuttuu riittävä kokemus vaatimusmäärittelystä ja käyttäjien tarpeet ovat muutosalttiita.

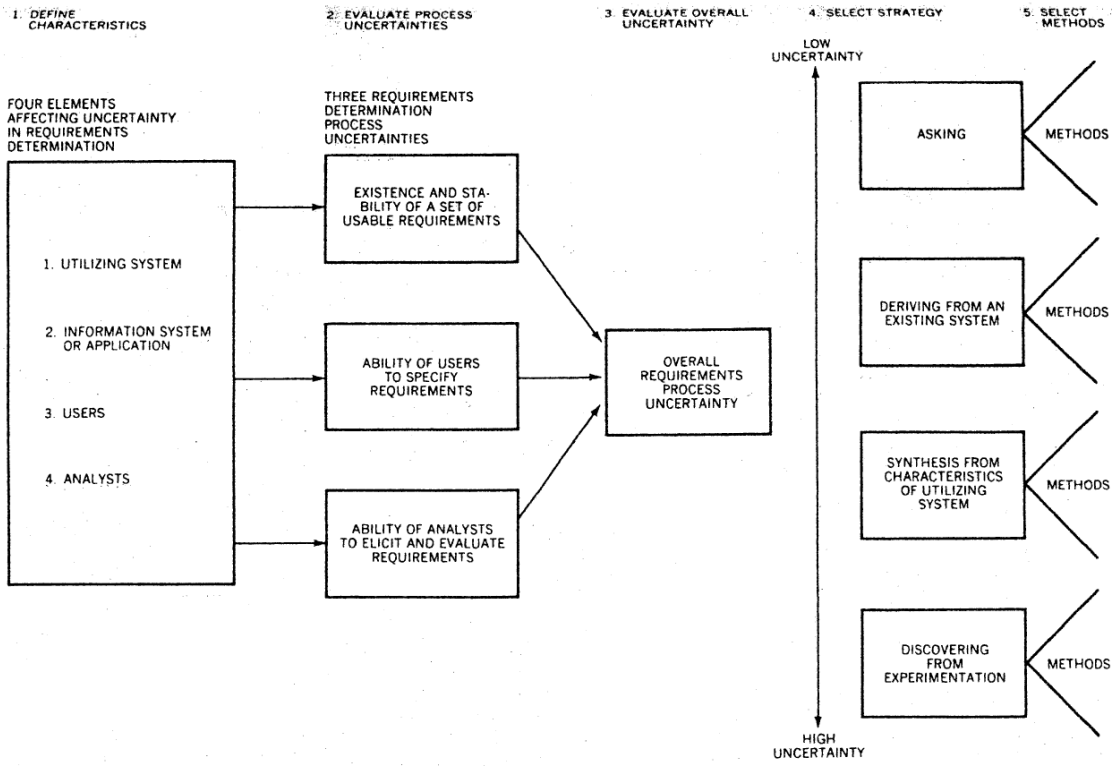
Strategian valinta suoritetaan neljällä askeleella (kuva 3.1; Davis, Olson, 1985), jotka alkavat hyväksikäyttävään järjestelmään, kehitettävään järjestelmään, käyttäjiin ja kehittäjiin liittyvien epävarmuustekijöiden tunnistamisella ja niiden arvioinnilla ja etenee kokonaisepävarmuuden johtamiseen. Strategian valinta ohjaa sopivan määritysmenetelmän valintaan. On itsestään selvää, että käyttäjien rooli on ehdottoman tärkeä vaatimusten määrittelyn kaikissa strategioissa ja menetelmissä.

RUP-menetelmässä on esitetty yksinkertainen jäsennys vaatimusmäärittelylle (huom. vaatimusmäärittely RUP-menetelmässä on tehtäväkokonaisuus, joka ulottuu vaihejaon mukaan konstruointivaiheeseen saakka) ja sen tuotoksille (Jacobson ym. 1999 s. 117).

Toiminto	Tuotos
listaa harkinnanalaiset vaatimukset	piirrelistat (feature lists)
kartuta ymmärrystä järjestelmän ympäristöstä (so. hyväksikäyttävästä järjestelmästä),	Liiketoimintamalli / kohdemaailman malli (domain model, liiketoimintaprosessit)
Määritä toiminnalliset vaatimukset	Käyttötapausmalli
Määritä ei-toiminnalliset vaatimukset	Täydentävät vaatimukset (eivät käyttötapauskohtaisia)

Vaatimusmäärittelyn tuloksena saaduille vaatimuksille on IEEE:n toimesta kehitetty seuraavat laatukriteerit:

- Oikeita: vaatimukset vastaavat käyttäjien todellisia toiveita,
- Yksiselitteisiä (unambiguous): vaatimukset on ilmaistu tavalla, joka ei jätä tulkinnan varaa,
- Kattavia: sisältävät kaikki tarpeelliset vaatimukset
- Johdonmukaisia (consistent): vaatimusten välillä ei ole ristiriitoja,
- Luokiteltuja (ranked): vaatimukset on luokiteltu: esim. pakolliset, valinnaiset, pohdinnanalaiset, mahdollisesti muuttuvat,
- Todennettavia (verifiable): vaatimuksen toteutumisen testaamiseksi tulee voida tehdä testitapaus,
- Muutettavia (modifiable): vaatimusten joukkoon on helppo lisätä uusia vaatimuksia,
- Jäljitettäviä (traceable): vaatimuksista käsin voidaan paikantaa vastaavat suunnittelu- ja toteutusosat ja päinvastoin,



Kuva 3.1: Neljä strategiaa vaatimusten määrittelylle

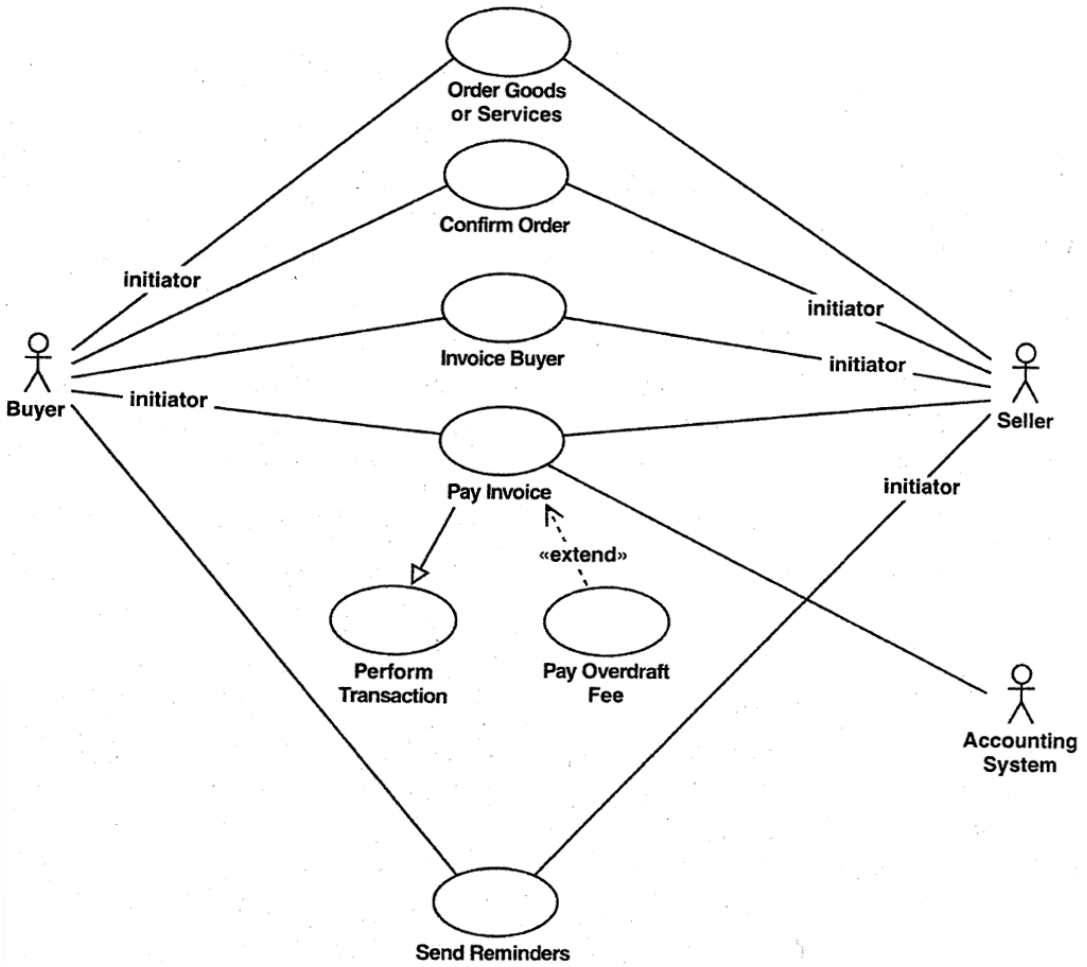
3.2 Käyttötapaukset

Alun perin Jacobsonin ym. (1992) esittämästä käyttötapausten (use cases) kuvaamisesta on tullut hyvin suosittu tekniikka, jolla voidaan edesauttaa järjestelmän rajaamista, toiminnallista jäsentämistä ja vaatimusten systemaattista määrittämistä. Tämä tekniikka on osana useissa uusimmissa oliomenetelmissä (joissakin ketterissä menetelmissä, kuten XP:ssä käyttötapausten sijaan käytetään kevyempiä käyttäjäkertomuksia, *user stories*), myös RUP:ssa. Seuraavassa kuvataan sen pohjana olevan mallin käsitteet ja käyttö RUP:n mukaisesti.

3.2.1 Käyttötapauskaavio: käsitteitä

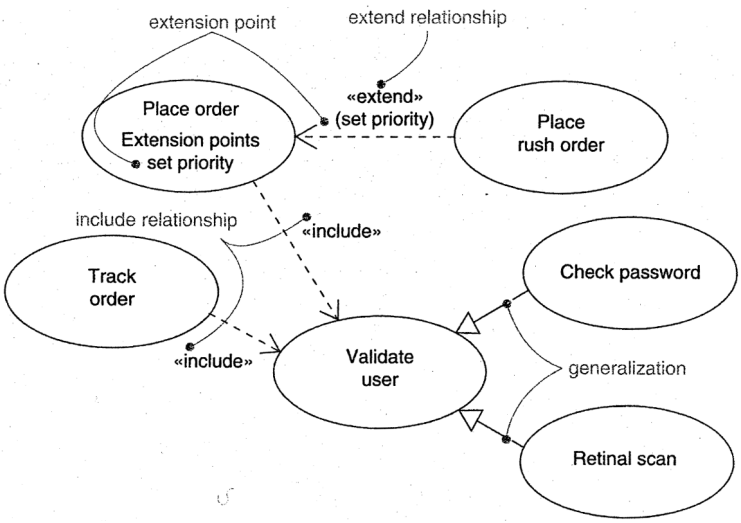
Käyttötapauksella (use case) tarkoitetaan toimintosarjaa, jota järjestelmä toteuttaa käyttäjien (aktorien) tarpeiden tyydyttämiseksi (Booch et al., 1999). Pääperiaatteena on, että yhden käyttötapauksen tulisi mudoostaa looginen kokonaisuus, jolla on selvä lähtökohta ja merkityksellinen lopputulos. Lähtökohtana eli herätteenä on tapahtuma tai tarve, joka käynnistää käyttötapauksen. *Käyttötapausmalli* koostuu käyttötapauskaaviosta sekä käyttötapauskuvauksista. Malli kuvaa aktorit, järjestelmän tarjoamat palvelut sekä yhteydet aktorien ja järjestelmän välillä, lyhyesti tavan käyttää järjestelmää.

Käyttötapauskaaviota (use case diagram) (kuva 3.2; Booch ym, 1999, s. 227): käytetään kuvaamaan järjestelmän tarkoitusta, rajausta, toimintoja ja käyttäjiä ja toimii lähtökohtana analyysimallin laadinnalle. Käyttötapauskaavio kuvaa yleisellä tasolla ne tavat ja tilanteet, joita järjestelmä tukee, toimivat aktorit (actors) ja sen, mitkä aktorit missäkin käyttötilanteissa ovat osallisina viestien lähettäjinä/vastaanottajina tai muulla tavalla toimijoina.



Kuva 3.2: Laskutusjärjestelmän käyttötapauskaavio liiketoimintaprosessiin Myynti: Tilauksesta toimitukseen liittyville käyttötapauksille. Initiator-rooli kuvaa, kuka aloittaa tapauksen suorituksen.

- Semantiikka:
  - ◊ *Aktori* on eräänlainen rooli, jossa käyttäjät (esim. operaattori, tietokannan hoitaja, peruskäyttäjä), laitteet (esim. tulostin) tai toiset järjestelmät (kirjanpitojärjestelmä) toimivat ollessaan yhteydessä järjestelmään.
  - ◊ *Kommunikointisuhde* aktorin ja käyttötapauksen välillä osoittaa järjestelmän kytkennän ympäristöön,
  - ◊ *Yleistyssuhde* käyttötapausten välillä: esimerkiksi käyttäjän tunnistaminen (validate user) voi tapahtua joko salasanan (Check password) tai silmän verkkokalvon skannaamisen (Retinal scan) avulla; merkitys samantapainen kuin myöhemmin käsiteltävässä luokkien välisessä yleistyssuhteessa (perintä)
  - ◊ *Sisältymissuhde* (include) käyttötapausten välillä: tarkoittaa, että toinen käyttötapaus (supplier) voi sisältyä toiseen käyttötapaukseen (client) viimeksi mainitun kontrollissa ja päättämässä tilanteessa. Esimerkiksi tilauksen tekeminen edellyttää aina käyttäjän tunnistamisen
  - ◊ *Laajennossuhde* (extend) käyttötapausten välillä tarkoittaa, että toinen käyttötapaus laajentaa toisen toiminta-alaa. Esimerkiksi tilausten tekemisessä noudatetaan normaalia toimintatapaa, paitsi kiireellisten tilausten tekemisen (poikkeustapaus) yhteydessä, jossa sovelletaan normaalin toimintatavan lisäksi erityistoimintoja. Edelleen laskun maksamisessa joudutaan joskus suorittamaan lisäksi tilinylitysmaksu (kuva 3.2; Jacobson ym, 1999, s. 146).



Kuva 3.3: Yleistys-, sisältymis-, ja laajennossuhteet

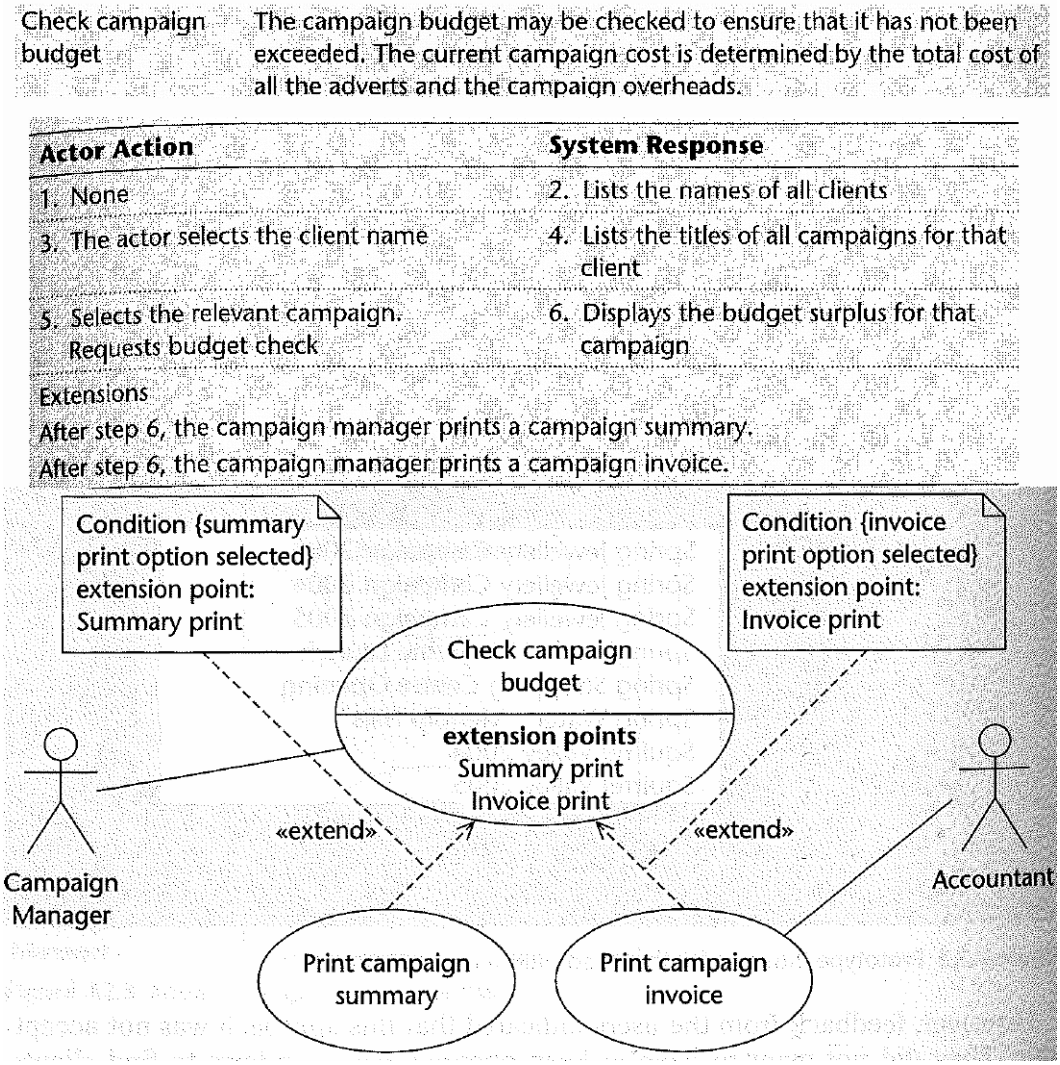
- Notaatio*:*
  - ◊ Käyttötapaus: ellipsi, jonka sisällä on käyttötapauksen nimi,
  - ◊ Aktori: tikku-ukko,
  - ◊ Kommunikointisuhde: yhtenäinen viiva
  - ◊ Yleistyssuhde: yhtenäisellä viivalla piiretty nuoli, jonka pää osoittaa yleistämisen suuntaan,
  - ◊ Sisältymissuhde: katkoviivalla piiretty nuoli, jonka pää osoittaa sitä käyttötapausta, joka sisällytetään. Nuolen yhteydessä käytetään lisäksi merkintää <<include>>
  - ◊ Laajennossuhde: katkoviivalla piirretty nuoli, jonka nuolenpää osoittaa sitä käyttötapausta, jota laajennetaan. Nuolen yhteydessä käytetään lisäksi merkintää <<extend>>.

Kaavio on tarkoitettu antamaan yleiskuva järjestelmästä. Sen ilmaisuvoima ei riitä kuitenkaan yksityiskohtien kuvaamiseen. Tästä syystä käyttötapauksia tarkennetaan lisäksi tekstimuotoisella *käyttötapauskuvauksella* (use case description). Kuvassa 3.4 on annettu esimerkki kuvauksen rakenteesta.

Use Case:	Assign Seat
Summary:	A passenger with a reservation on a flight requests a seat assignment. The system obtains information from the passenger and then attempts to make an assignment. The assignment is given to the passenger or the passenger is told that no assignment is presently available.
Actors:	Passenger
Preconditions:	The passenger has a reservation on a flight by the given airline
Description:	A passenger requests a seat assignment on a flight. (This may be implicit as part of checking in or may be an explicit request by the passenger.) The system (in the form of the agent) asks for the date of the flight, the flight number, departure airport, and passenger name. The passenger supplies the information. Instead of name, the passenger can supply frequent flyer number with the airline. [Exception: Too early to make assignment.] The system finds the reservation. [Exception: No reservation found.] If the reservation already has a seat assignment, it is given to the passenger who is offered the opportunity of changing the assignment. If there is no assignment or the passenger wants to change, the system requests seat preference (window, aisle) and smoking preference (yes, no). The system uses the information, including frequent flyer level, to try to find a suitable seat assignment subject to previous assignments and the policies of the airline. If necessary, the system asks additional questions: would the passenger accept a bulkhead seat, would the passenger accept a seat in an emergency exit row? The system proposes a seat assignment. If all the passenger's preferences cannot be satisified, then the system proposes the best match it can find. [Exception: No assignment possible.] The passenger can accept the assignment or ask for changes, in which case another assignment is attempted. The use case concludes when an assignment is made and accepted.
Exceptions:	Too early: Raised if the current date is too early, based on an airline algorithm that includes fare category and frequent flyer level. The passenger is advised when seat assignments will be possible and the use case terminates. No reservation found: Raised if no reservation can be found. The information is rechecked and searched if necessary for a partial match. If still no reservation can be found, then the passenger is advised to obtain one and the use case terminates. No assignment possible: Raised if no assignment is possible based on an airline-dependent formula that includes number of unassigned seats, days until departure, fare category, and frequent flyer level. The passenger is advised to obtain an assignment on check in. If this is part of check in, then the passenger is placed on standby in the order of arrival. The use case terminates.
Postconditions:	If this is part of check in, then the passenger either has a seat assignment or is on the standby queue in order of arrival. Otherwise none (the attempt may fail).

Kuva 3.4: Esimerkki käyttötapauskuvauksesta

Käyttötapauksia voidaan kirjoittaa useilla eri mallipohjilla (use case templates). Vaihteoehtoinen tapa käyttötapauksen kuvaamiseen on kahden sarakkeen muoto, jossa käyttäjän ja järjestelmän toiminnot on selkeästi eroteltu (kuva 3.5; Bennett 2006).



Kuva 3.5: Mainostoimistoesimerkin (Bennett 2006) käyttötapaus "tarkasta kampanjabudjetti" lyhennelmänä, 2-sarakemuodossa ja käyttötapauskaavion osana. Huom. käyttötapauksen vaiheiden kuvauskäytännössä on pieniä ongelmia: "none"-niminen vaihe ja yleinen "Actor"-niminen rooli.

Esimerkkinä laajasta ja useita poikkeuksia tai vaihtoehtoisia polkuja sisältävästä käyttötapauksesta esitetään Cockburnin (2000) RUP-tyylipohjalla (numeroidut askeleet ja poikkeukset) kirjoitettu *Register for courses* -käyttötapaus[14]. Askeleissa on myös kommentteja esimerkkinä käyttötapauksen kirjoitusprosessista (osa kommenteissa mainituista asioista kuuluisi ennemmin huomautuksiin kuin varsinaiseen käyttötapauksen tekstiin).

[14] Tässä käytetty esitys perustuu sivuun http://www.victoria.ac.nz/lals/research/usecase/data/files/042.aspx

Use case name	**Register for Courses**
Description	This use case allows a Student to register for course offerings in the current semester. The Student can also modify or delete course selections if changes are made within the add/drop period at the beginning of the semester. the Course Catalog System provides a list of all the course offerings for the current semester.
Actors	The main actor of this usecase is the Student. The Corse Catalog System is an actor within the use case.

Flow of events

The use case begins when the Student selects the "maintain schedule" activity from the Main Form. *(Refer to user-interface prototye for screen layout and fields.)*

Basic Flow

1 Create a Schedule

1.1 The Student selects "create schedule."

1.2 The system displays a blank schedule form. *(Refer to user-interface prototype for screen layout and to the domain model for required fields.)*

1.3 The system retrieves a list of available course offerings from the Course Catalog System. *(How is this selected and displayed? Text? Drop-down lists?)*

1.4 The Student selects 4 primary course offerings from the list of available offerings. Once the selections are complete the Student selects "submit." *(Define "primary course offerings" and "alternative course offerings" in project glossary. Must exactly 4 and 2 selections be made? Or "up to 4...," etc.)*

1.5 The Add Course Offering subflow is performed at this step for each selected course offering.

1.6 The System saves the schedule. *(When is the master schedule updated? Immediately? Nightly (batch)?)*

Alternative Flows

2 Modify a Schedule

2.1 The Student selects "modify schedule."

2.2 The system retrieves and displays the Student's current schedule (e.g., the schedule for the curent semester). *(Is this only available for the current semester?)*

2.3 The system retrieves a list of all the course offerings available for the current semester from teh Course Catalog System. The system displays the list to the Student.

2.4 The Student can then modify the course selections by deleting and adding new courses. The Student selects the courses to add from the list of available courses. The Student also selects any course offerings to delete from the existing schedule. once the edits are complete the Student selects "submit."

2.5 The Add Course Offering subflow is performed at this step for each selected course offering.

2.6 The System saves the schedule.

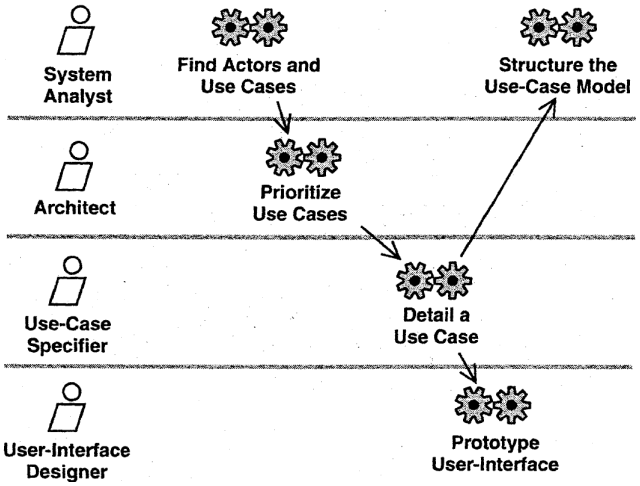
3 Delete a Schedule

3.1 The Student selects the Delete Schedule activity

	3.2 The system retrieves and displays the Student's current schedule. 3.3 The Student selects "delete." 3.4 The system prompts the Student to verify the deletion. 3.5 The Student verifies the deletion. 3.6 The system deletes the schedule. *(At what point are the student slots freed up?)* 4 *Save a Schedule* At any point, the Student may choose to save a schedule without submittingit by selecting "save." the current schedule is saved, but the student is not added to any of the selected course offerings. The course offering sare marked as "selected" in the schedule. 5 *Add Course Offering* The system verifies that the Student has the necessary prerequisites and that the course offering is open. The system then adds the Student to the selected course offering. the course offeringis marked as "enrolled in" in the schedule. 6 *Unfulfilled Prerequisites or Course Full* If in the Add Course subflow the system determines that the Student has not satisfied the necessary prerequisites or that the selected course offering is full, an error message is displayed. The Student can eithr select a different course offering or cancel the operation, at which point the use case is restarted. 7 *No Schedule is Found* If in the Add Course subflow the system determines that the Student has not satisfied the necessary prerequisites or that the selected course offering is full, an error message is displayed. The Student can either select a different course offering or cancel the operation, at which point the use case is restarted. 8 *Course Catalog System Unavailable* If the system is unable to communicate with the Course Catalog System after a specified number of tries, the system will display an error message to the Student. the Student acknowledges the error message and the use case terminates. 9 *Course Registration Closed* If, when the student selects "maintain schedule," registration for the current semester has been closed, a message is displayed to the Tudent and the use case terminates. Students cannot register for courses after registration for the current semester has been closed.
Special Requirements	None
Preconditions	**Login**: Before this use case begins the Student has logged onto the system.
Postconditions	None
Extension Points	None

3.2.2 Käyttötapausten mallintaminen

Kuvassa 3.6 (Jacobson ym, 1999 s. 143) on esitetty yksinkertainen työnkulku käyttötapausten mallintamiseksi. Siinä esitetään suoritettaviksi seuraavat askeleet:



Kuva 3.6: RUP-menetelmän työnkulku käyttötapausten mallintamiseksi

1. Etsi aktorit ja käyttötapaukset:
   - ◊ Hahmottele rajausta
   - ◊ Tunnista aktorit, jotka kommunikoivat "ilman välikäsiä" järjestelmän kanssa,
   - ◊ Tunnista ja hahmottele pääkäyttötapaukset; voit aloittaa tekemällä skenaarioita tyypillisistä tapauksista,
   - ◊ Kuvaa järjestelmä käyttötapauskaaviona; täydennä kaaviota tekstikuvauksilla; laadi sanastoa keskeisistä asioista,
2. Priorisoi käyttötapaukset tärkeysjärjestykseen, jos niitä kaikkia ei ole resursseja/tarvetta toteuttaa kerralla,
3. Saatetaan kuvauksia käyttötapauksista tarkemmiksi
   - ◊ Tarkenna käyttötapausten sisäistä ja välistä rakennetta; aloitus- ja lopetuskohdat, vaihtoehtoiset etenemispolut, poikkeustapaukset, jne.
   - ◊ Tekstikuvausten lisäksi käyttötapausten kulkua voi visualisoida aktiviteettikaavioilla,
4. Rakenna protyyppi käyttöliittymien konkretisoimiseksi
   - ◊ Järjestelmän toiminnan ymmärtämiseksi ja parempien/tarkempien vaatimusten esille saamiseksi on usein tarpeen rakentaa kevyt prototyyppi, joka simuloi lähinnä esityskerrosta (vrt. kuva 1.5).
5. Määritä käyttötapausten välille rakenteelliset suhteet:
   - ◊ Ota käyttöön käyttötapauksia koskevat yleistys-, laajennos- ja sisältymissuhteet.

Huomattakoon, että RUP-menetelmässä prototyyppi on tarkoitus toteuttaa vasta seuraavan vaiheen aikana, mutta ICONIX-menetelmässsä sitä voidaan käyttää jo aiemmin käyttötapausten kulun selvittämisessä. ICONIX-menetelmässä korostetaan käyttötapausten mahdollisimman yksityiskohtaista mallintamista erityisesti poikkeustapausten osalta, mutta rakenteellisten suhteiden (include, extends) ja metatietokenttien (esi- ja jälkiehdot, laajennospisteet ym.) määrittämistä ei pidetä yhtä merkittävänä. Perusteena tälle on, että käyttötapauksen kulku on jäljitettävissä suoraan myöhemmissä kehitysvaiheissa aina toteutukseen asti, mutta rakenteelliset suhteet liittyvät enemmän käyttötapauskaavion muotoiluun.

Vinkkejä käyttötapausten kirjoittamiseen (tarkemmin ks. esim. Cockburn 2000, Rosenberg & Scott 1999):

- Käyttötapausten nimeäminen käskymuodossa (*Tarkasta* kampanjabudjetti, *Lisää* asiakas jne) – tämä korostaa tavoitetta, jonka suoritukseen pyritään
- Käyttötapausten askeleiden sanajärjestys subjekti-predikaatti-objekti-muodossa. Esim. ”kampanjapäällikkö valitsee asiakkaan tiedot”
- Käyttötapauksen lauseiden tulee olla yksinkertaisia ja helppolukuisia erityisesti pääskenaarion osalta. Tietojen kenttäkohtaisia yksityiskohtia ei tarvitse merkitä askeleisiin (tiedot mallinnetaan joka tapauksessa tarkasti analyysivaiheen aikana). Askeleita täydentäviä lisätietoja tai esim. toistorakenteiden kuvauksia voidaan merkitä erilliseen *Huomautukset*-kohtaan.
- Tarkkuustaso yksittäisille käyttötapauksille tulee valita tilanteen mukaan. Käyttötapaus ei voi olla kuitenkaan liian pieni (käyttötapaus on toimintojen sarja – ei toiminto[15]) eikä liian laaja (jotta kuvaus pysyy hallinnassa). Keskikokoisessa järjestelmässä voi olla muutamia kymmeniä ja isoimmissa järjestelmissä jopa satoja käyttötapauksia. Esimerkiksi tietokantasovelluksissa yleiset ”CRUD” (create, read, update, delete) -tyyliset päivitystoiminnot voidaan usein pakata samoihin käyttötapauksiin (esim. *Päivitä* tietoja – erittäin laajoissa järjestelmissä jopa niin, että monen päivityslogiikaltaan samanlaisen kohdealueluokan päivitystoiminnot on kuvattu samassa käyttötapauksessa), jonka poikkeuksia, vaihtoehtoisia polkuja tai variaatioita yksittäiset toiminnot ovat.
- Käyttötapaukseen liittyvät poikkeukset ja vaihtoehtoiset toiminnot tulee kuvata huolellisesti. Jokainen jo varhaisessa vaiheessa löydetty poikkeustilanne helpottaa myöhempien vaiheiden mallintamista.
- Käyttötapaukset antavat järjestelmän ulkoisen kuvan (miltä järjestelmä näyttää käyttäjän näkökulmasta). Järjestelmän sisäinen rakenne (esim. osajärjestelmäjako) voi olla aivan toisenlainen.

Käyttötapauskaavio ja -kuvaukset toimivat apuna *toiminnallisten vaatimusten* määrittämisessä. Kukin käyttötapaus tai sen osa tulee liittää vaatimuksiin (yleisesti: jokaista käyttötapausta voi potentiaalisesti vastata useita vaatimuksia ja päinvastoin), jolloin käymällä käyttötapauksia läpi voidaan varmistaa vaatimusten toteutuminen. Kullekin käyttötapauksille erikseen tai niille yhteisesti voidaan lisäksi määritellä *ei-toiminnallisia vaatimuksia* (esim. tietojen tarkkuus- ja luotettavuustaso, turvallisuus, toimintahäiriöttömyys, käyttäjäystävällisyys, siirrettävyys ym.), mutta nämä eivät yleensä ole helposti kuvattavissa pelkillä käyttötapauksilla. Vaatimusmäärityksiä tarkennetaan analyysin kuluessa ja myöhemmin myös muissa vaiheissa. Samalla tulee täydentää niiden priorisointia, toteutusaikataulua ja toteutumista (esim. jaetaan eri iteraatioihin toteutettavat vaatimukset).

3.3 Aktiviteettikaaviot

UML-aktiviteettikaavio (toimintakaavio) on yleiskäyttöinen dynaamisen mallintamisen työkalu (ks. kuva 3.7). Toimintakaavioita käytetään usein mallintamaan organisaa-

[15] Ks. K. Bittner (2000). *Why Use Cases Are Not "Functions"*. http://download.boulder.ibm.com/ibmdl/pub/software/dw/rationaledge/dec00/WhyUseCasesAreNotFunctionsDec00.pdf

tioiden liiketoimintaprosesseja (tässä *liiketoimintaprosessi*[16] määritellään järjestetyiksi työvaiheiksi, joilla on alku ja loppu sekä syötteitä ja tuotoksia. Liiketoimintaprosessit kulkevat tyypillisesti useiden organisaatioiden läpi), niihin kuuluvia rakenteisia työnkulkuja (workflow) ja niitä tukemaan suunniteltavia tietojärjestelmiä. Muita käyttökohteita ovat esimerkiksi monimutkaisten algoritmien kuvaus, järjestelmän suorittamien rinnakkaisten prosessien mallinnus sekä käyttötapausten visualisointi vaihtoehtona tekstikuvaukselle (mitä toimintoja tulisi suorittaa ja milloin). Aktiviteettikaaviolla voidaan kuvata myös ohjelmistoprosessia – mitä aktiviteetteja kenenkin pitäisi suorittaa prosessin eri vaiheissa. Aktiviteettikaaviot ovat erikoistapauksia tilakaavioista. Näitä käsitellään tarkemmin suunnittelun yhteydessä, ks. luku 5.3.2.1.

Ilmaisuvoimaltaan aktiviteettikaaviot ovat lähellä vanhemmissa menetelmissä käytettyjä vuokaavioita (flowchart) - periaatteessa kaaviolla pystyisikin kuvaamaan toteutusta yksittäisten koodirivien tasolla. Tämä ei kuitenkaan ole ”oliomainen” tapa järjestelmän toiminnan kuvaamiseen, koska se vie huomion olioiden välisestä vuorovaikutuksesta toteutuksen yksityiskohtiin. Tällä kurssilla aktiviteettikaavioita käytetään ensisijaisesti vaatimusmäärittelyn yhteydessä kehitettävän järjestelmän ympäristön kuvaamiseen (esim. millaista liiketoimintaprosessia kehitettävä järjestelmä tehostaa) ja käyttötapausten etsimiseen (liiketoimintaprosessin tasoisessa aktiviteettikaaviossa monet järjestelmän käyttöön liittyvät yksittäiset aktiviteetit vastaavat käyttötapauksia).

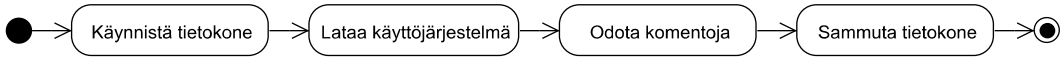
3.3.1 Aktiviteettikaavio: käsitteitä

Aktiviteettikaaviossa käytetään seuraavaa notaatiota:

Lähtötila (Initial state): Toiminnan alkaminen, merkintänä tummennettu piste.

Tila (State): Prosessin selkeä vaihe, jolla on jokin kesto. Merkitään kulmista pyöristetyllä suorakaiteella, jonka sisälle kirjoitetaan tilan nimi.

Tilasiirtymä (Transition): Siirtyminen tilasta toiseen. Merkintänä nuoli tilojen välillä.

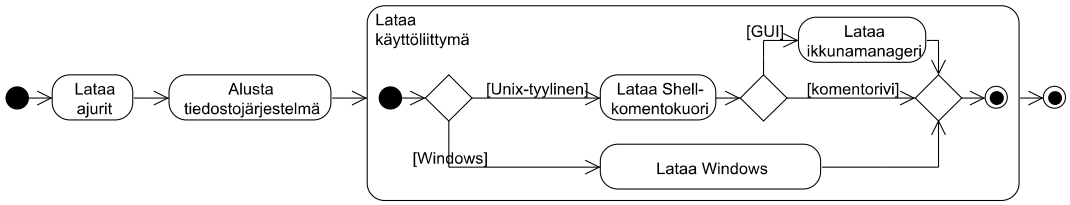


Kuva 3.7: Esimerkki yksinkertaisesta tilakaaviosta: tietokoneen käynnistys, komentojen odottaminen ja sammutus.

Haarautuminen(Branch)/Koostaminen(Merge): Tilasiirtymä saattaa jonkin ehdon (guard condition) perusteella haarautua kahdeksi tai useammaksi toisensa poissulkevaksi toiminnoksi. Jokaiseen haarautumiseen liittyy aina koostaminen, jossa muodostetaan vaihtoehtoisten tilasiirtymien jälkeen uusi tila. Ehto määritetään hakasulkeiden

[16] T.H. Davenport (1993). Process Innovation - Reengineering Work through Information Technology ja A. Sharp & P. McDermott (2001). Workflow Modeling.

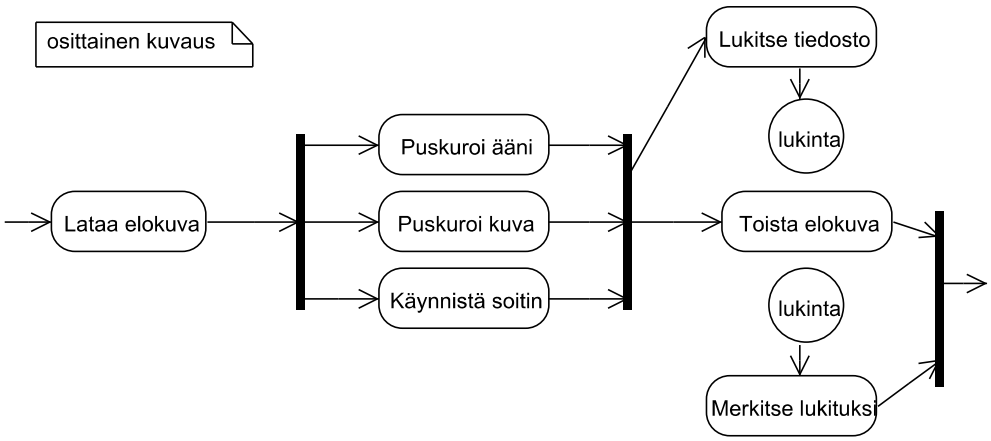
[ ] sisällä. Haarautuminen ja koostaminen merkitään värittämättömän salmiakkikuvion avulla.



Kuva 3.8: Esimerkki haarautumisesta ja alitiloista. Aktiviteettikaavio on kokonaisuutena kuvan 3.7 Lataa käyttöjärjestelmä-aktiviteetin tarkennettu kuvaus. Lataa käyttöliittymä-aktiviteetin sisäinen rakenne on kuvattu samaan kaavioon.

Jakautuminen(Fork)/Yhdistäminen(Join): Tilasiirtymä saattaa jakautua kahdeksi tai useammaksi yhtäaikaiseksi toiminnoksi (parallel activity), kunnes palataan yhdistämisen kautta uuteen tilaan. Tällöin käytetään merkintänä paksua viivaa yhtäaikaisten toimintojen alun ja lopun yhteydessä. Merkintöjen välissä rinnakkaisia toimintoja suoritetaan siis yhtäaikaisesti.

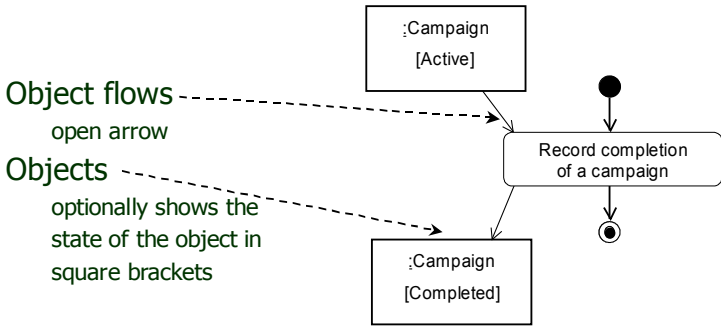
Toimintokaari (Activity edge): Joissain tapauksissa toimintakaavioon voidaan joutua lisäämään etenemisreittejä, jotka leikkaisivat häiritsevästi jo olemassa olevia symboleita. Jotta toimintakaavio säilyisi selkeänä, käytetään näissä tilanteissa toimintokaarta reitin visuaaliseen kuvaukseen. Notaationa on ympyrä, jonka sisällä toiminnan nimi. Kuviossa täytyy olla aina samalla nimellä nimetty toiminnan rajan alku- ja loppukohdat. Näiden kohtien välille ei piirretä varsinaista viivaa, vaan kaaviossa siirrytään toiminnan rajan vastaan tullessa sen vastinpariin, josta matkaa jatketaan.



Kuva 3.9: Esimerkki rinnakkaisesta toiminnasta ja aktiviteettikaarista: elokuvan katsominen tietokoneella ja tiedoston lukitseminen elokuvaa katsoessa. Huom. kuvaus on vain osittainen aktiviteettikaavio, koska alku- ja lopputiloja ei ole merkitty.

Kaistaloiminen (Swimlane): Toimintakaavio voidaan jakaa erillisiin kaistaleisiin (”uimaradat”) aktorien avulla. Prosessiin osallistuvista aktoreista kukin saa kaaviossa oman kaistansa, josta nähdään helposti mitä toimintoja kunkin aktorin tulee kulloinkin suorittaa.

Edellisissä toimintakaavio-esimerkeissä on käytetty UML 2.0:n toimintokaavionotaation kontrollivirtoja (Control flow), joiden avulla tarkastellaan erityisesti toimintojen kulkua, suoritusjärjestystä ja järjestelmän logiikkaa. Notaatioon kuuluu toinenkin osio eli oliovirta (Object flow, tietovirta). Järjestelmässä saattaa olla tilasiirtymien yhteydessä tai niistä riippumatta olioita, joiden toiminta muuttuu riippuen siitä, vastaanottavatko ne tietoa vai lähettävätkö ne sitä. Esim. mainostoimisto-esimerkissä (kuva 3.10) kampanjan päättäminen edellyttää, että ennen toimintoa päätettävän kampanjan tila on aktiivinen. Toiminnon yhteydessä kampanja kirjataan päätetyksi.



Kuva 3.10: Esimerkki oliovirroista

3.3.2 Aktiviteettikaavioiden mallintaminen

Aktiviteettikaavioita laadittaessa tavoitteena on luoda kokonaiskuva järjestelmän käyttäytymisestä ympäristössään kuvaamalla toimintojen logiikan kulku, rinnakkaiset tapahtumat ja kunkin aktorin osallistuminen prosessin vaiheisiin.

1. Aloita tarkastelemalla käyttötapauskuvauksia ja skenaarioita. Etsi toimintaan osallistuvat aktorit ja rajaa kaavioon kullekin oma kaistaleensa (swimlane).
2. Pyri luomaan aluksi suoraviivainen malli prosessin kulusta. Huomioi tässä vaiheessa mitkä toiminnot olisivat mahdollisesti rinnakkaisia (yhtäaikaisia) ja mille aktorille mikin tarkasteltava tila kuuluu.
3. Pyri selkeyteen aktorien kommunikoinnissa (esim. vastuullinen osapuoli)
4. Älä kiinnitä vielä tässä vaiheessa huomiota yksityiskohtaisiin tietoihin, kuten välitettäviin viesteihin tai niiden parametreihin.
5. Lisää kaavioon poikkeustapaukset käyttäen haarautumista (branch/merge). Tällä tavoin saat ilmaistua ehdolliset poikkeavat reitit.
6. Tarkista onko järjestelmän logiikassa joitakin selviä virheitä ja korjaa ne ennen kuin mallinnat muita.

Hyvä käytäntö aktiviteettikaavioiden piirrossa on merkitä jokaiseen aktiviteettiin tasan 1 lähtevä ja 1 tuleva kontrollivirta. Rinnakkaisten tai ehdollisten toimintojen mallintamiseen ja kontrollivirtojen ryhmittelyyn käytetään Branch/Merge-salmiakkeja tai Fork/Join-viivoja (silmukan alussa olevat Merge-elementit voi jättää pois, jos kaavion muotoilun haluaa pitää minimaalisena, koska tämä ei muuta kaavion merkitystä). Väl-

tä tilanteita, joissa aktiviteetista puuttuu kokonaan lähtevä tai tuleva kontrollivirta – rakenteen tulkinta on epäselvä (esim. mistä prosessiin saavutaan? Milloin toiminta loppuu?).

Esimerkki käyttötapauksen visualisoinnista aktiviteettikaaviolla: Rahan nostaminen ja tilitietojen tarkastaminen pankkiautomaatilla (ks. Liite B). Tässä esimerkissä (kuva 3.11) esiintyvät kaikki edellä esitetyt notaatiot kootusti, mutta muutamia poikkeustapauksia on jätetty käsittelemättä esimerkin selkeyttämiseksi.

Käyttötapaus: Nosta rahaa ja tarkista tilitiedot

Tiivistelmä: Yksittäiseen pankkiin yhteydessä oleva pankkiautomaatti (ATM) antaa asiakkaalle rahaa ja tilitietoja pankkikortin avulla. Käyttötapaus ja toimintakaavio on luotu liitteen B (ATM) toimintamallin pohjalta, mutta Consortium–aktori on poistettu. Käyttötapaus käsittelee selkeyden vuoksi ainoastaan poikkeustapaukset "Väärä tunnusluku" ja "Riittämätön tilin saldo". Muita poikkeustapauksia (kuten asiakkaan valitsemat keskeytykset) ei käsitellä tässä käyttötapauksessa, jotta kaikkien esiteltyjen toimintakaavion notaatioiden käyttömahdollisuudet tulisivat selkeästi esille.

Aktorit: Asiakas, ATM ja Pankki

Esiehdot: Asiakkaalla on voimassa oleva tili pankissa, jonka kanssa hän asioi pankkiautomaatin avulla. Hänen pankkikorttiinsa on liitetty täsmälleen yksi voimassa oleva tili ko. pankissa.

Kuvaus: Asiakas antaa korttinsa pankkiautomaattiin. Automaatti kysyy tunnuslukua (PIN), jonka asiakas antaa [Poikkeus: Väärä tunnusluku]. Automaatti kysyy asiakkaalta tilitoiminnon tyyppiä, jolloin asiakas valitsee joko noston tai tilitiedot. Tilitiedot–valinnalla automaatti tulostaa kuitin ja kysyy asiakkaalta jatketaanko vai lopetetaanko. Mikäli valitaan lopetus, niin automaatti palauttaa kortin ja palaa perustilaansa. Nosto–valinnalla automaatti kysyy summaa, jonka asiakas antaa. Tämän jälkeen automaatti tarkistaa pankilta, onko tilillä riittävästi katetta [Poikkeus: Riittämätön tilin saldo]. Automaatti antaa yhtäaikaisesti rahat ja veloittaa valitun summan tililtä. Asiakas ottaa rahat ja automaatti tulostaa kuitin, jonka jälkeen automaatti kysyy, jatketaanko vai lopetetaanko.

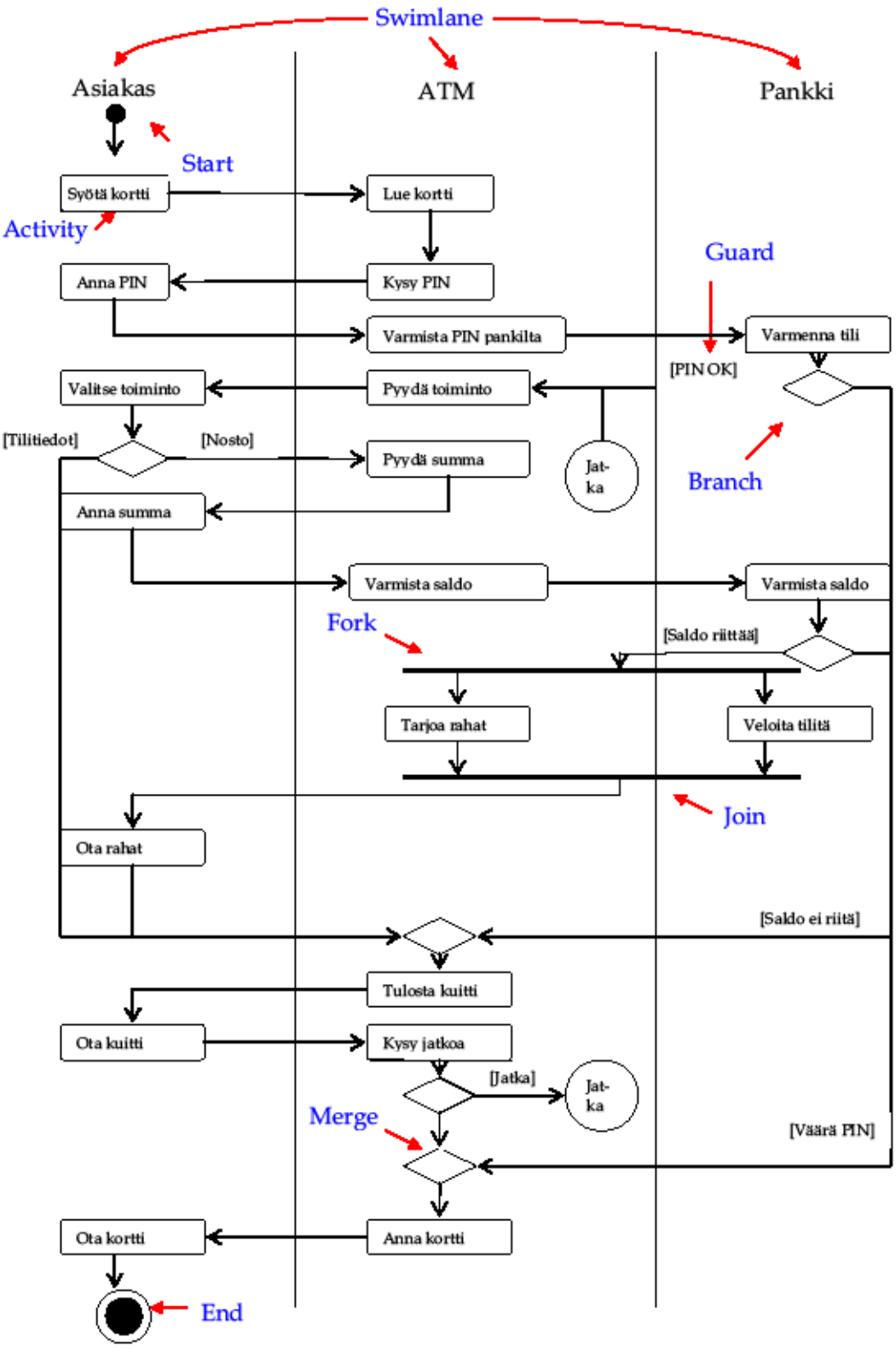
Lopetus–valinnalla automaatti palauttaa kortin ja palaa perustilaansa. Jatka–valinnalla automaatti palaa tilaan ’Valitse toiminto’.

Poikkeukset:

Väärä tunnusluku: Automaatti palauttaa kortin ja palaa perustilaansa.

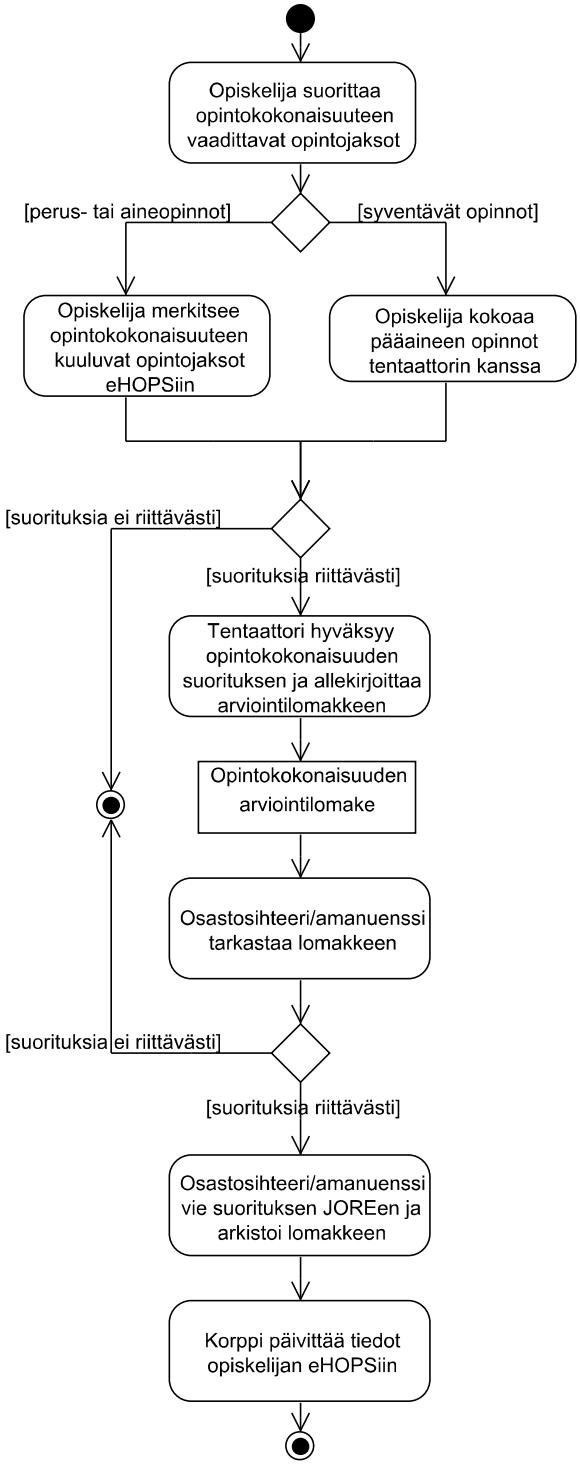
Riittämätön tilin saldo: Automaatti tulostaa kuitin ja kysyy jatketaanko vai lopetetaanko.

Loppuehdot: Käyttäjä poistuu rahojen kanssa automaatilta ja automaatti jää odottamaan seuraavaa käyttäjää.



Kuva 3.11: Toimintakaavio asioinnille pankkiautomaatilla.

Esimerkkinä prosessitasoisesta aktiviteettikaaviosta esitetään JY:n IT-tdk:n prosessien[17] pohjalta muokattu kuvaus *Opintokokonaisuuden kokoaminen (tavoitetila)* (kuva 3.12).



Kuva 3.12: Esimerkki liiketoimintaprosessista: Opintokokonaisuuden kokoaminen.

[17] http://prosessit.it.jyu.fi/ - suurin osa sivuston prosesseista on kuvattu tavanomaisia liiketoimintaprosesseja yksityiskohtaisemmin, jolloin kuvaukset toimivat samalla toimintaohjeina.

4 ANALYYSI

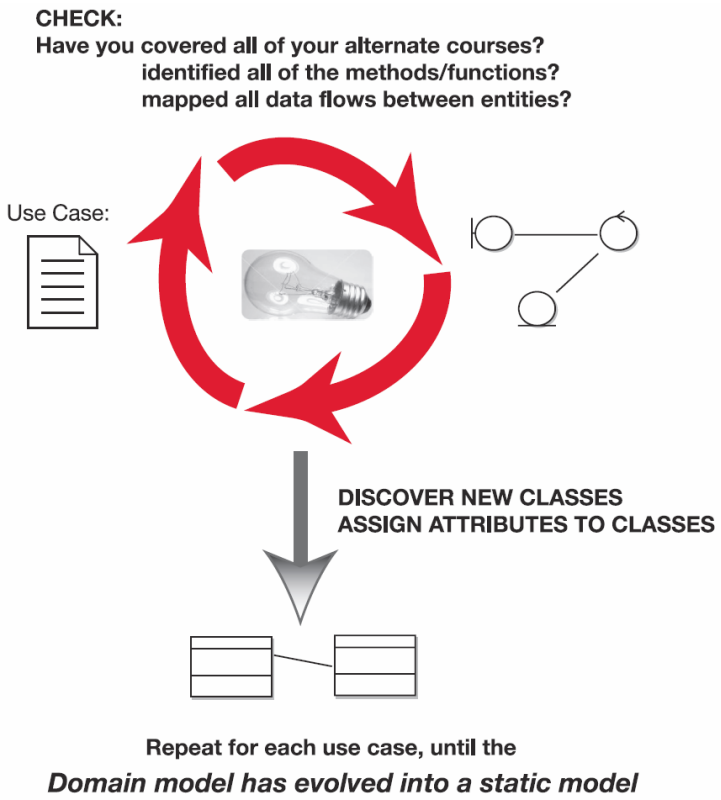
Analyysi (myös: vaatimusanalyysi, *requirements analysis*) ei esiinny tämännimisenä vaiheena RUP:ssa (vaan tehtäväkokonaisuutena). Tässä esityksessä analyysivaiheeseen on sisällytetty seuraavat tehtävät:

- Arkkitehtuurin mallintaminen, jossa hahmotellaan järjestelmän perusarkkitehtuuri,
- Staattinen mallintaminen (oliomallintaminen OMT:ssa, domain modelling RUP:ssa), jonka pääasiallisena tuloksena saadaan alustava luokkakaavio
- Dynaaminen mallintaminen (ICONIX-menetelmässä robustness analysis), jossa vuorovaikutuskaavioiden (sekvenssikaaviot ja yhteistoimintakaaviot) avulla tarkennetaan järjestelmän käyttäytymistä
- Käyttöliittymän suunnittelu (OMT++:n mukaan), jossa dialogikaavioiden ja näyttömallien avulla hahmotellaan käyttöliittymien rakennetta ja toimintaa,

Analyysivaiheen staattisten ja dynaamisten mallien suhdetta vaatimusmäärittelyyn ja mallintamisjärjestystä voidaan hahmottaa useilla eri tavoilla:

- OMT-menetelmässä ja kurssilla käytetyssä prosessissa mallinnetaan aluksi vaatimukset ja käyttötapaukset, tämän jälkeen staattinen kohdealuemalli (*domain model*, käsitemalli), jonka jälkeen edetään dynaamisiin malleihin (kuitenkin staattista mallia iteroiden).
- RUP-menetelmässä eri aktiviteetteja voidaan vaiheiden ja tehtäväkokonaisuuksien erottelun ansiosta tehdä samanaikaisesti, mutta käyttötapauslähtöisyyttä korostetaan niin, että vaatimukset johdetaan käyttötapauksista toisin kuin muissa tässä esitellyissä menetelmissä.
- Bennett ym. (2006) suosittelevat prosessia, jossa käyttötapausten mallintamisen jälkeen mallinnetaan aluksi yhteistoimintakaaviot ja vasta tämän jälkeen kohdealuemalli.
- ICONIX-menetelmä (Rosenberg & Scott 1999) poikkeaa järjestykseltään edellä mainituista menetelmistä siten, että alustava kohdaluemalli määritellään vaatimusten pohjalta jo ennen käyttötapauksia, jonka jälkeen niitä tarkennetaan iteratiivisesti. Kohdealuemallin varhainen määritys edesauttaa yhtenäisen terminologian käyttöä käyttötapauskuvauksissa ja myöhemmissä malleissa.

Sekä Bennett että ICONIX-menetelmä korostavat prototyyppien laatimista mahdollisimman varhaisessa vaiheessa käyttötapausten mallintamisen apuna. Käytännössä käyttötapausten ja vaatimusten etsintää sekä ja analyysivaiheen staattista ja dynaamista mallintamista kannattaa menetelmästä riippumatta tehdä iteratiivisesti tarkentaen aloittaen mallista, joka tuntuu luonnollisimmalta tai helpoimmin lähestyttävältä ongelma-aluetta ajatellen. Mallinnuksen eri aktiviteetit tukevat toisiaan ja lopputuloksen pitäisi järjestyksestä riippumatta olla sama. ICONIX-menetelmän staattisen ja dynaamisen mallin suhteita kuvaava silmukka (kuva 4.1; Rosenberg & Stephens 2007) havainnollistaa asiaa.



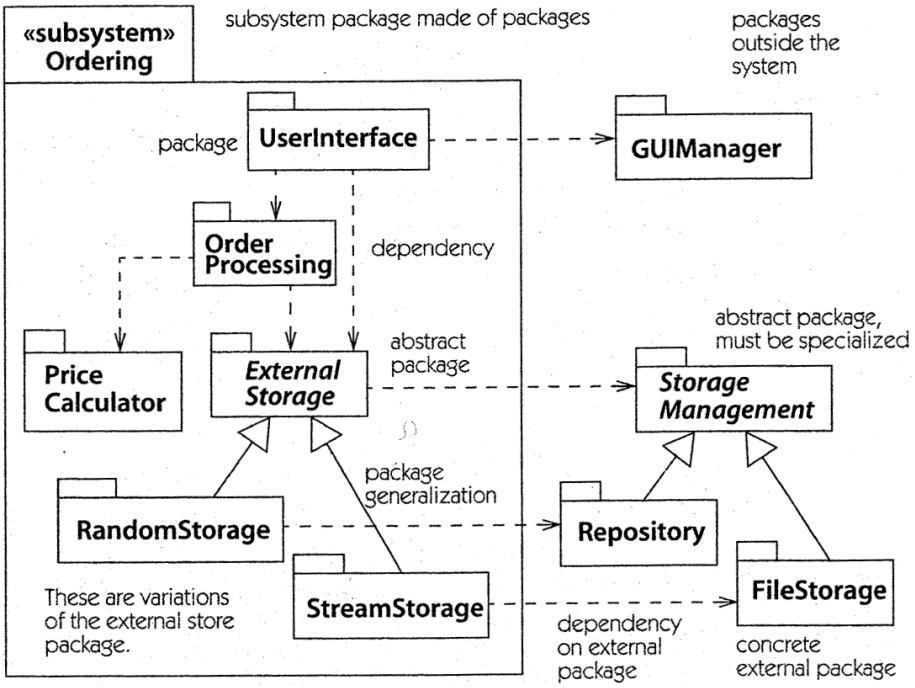
Kuva 4.1: Vaatimusmäärittelyn ja analyysin eteneminen ICONIX-menetelmässä

4.1 Arkkitehtuurin mallintaminen

Jo kehittämisen alkuvaiheessa on tarpeellista hahmotella arkkitehtuurin perusratkaisu. Lähtökohdat tähän ratkaisuun saadaan käyttötapauksista sekä seuraavista ympäristötekijöistä (Jacobson ym. 1999, s. 66):

- millaisia olemassa olevia järjestelmiä (legacy systems) halutaan käyttää tai mihin luoda yhteys,
- millaisia hajautusratkaisuja organisaatio aikoo käyttää,
- millaisia systeemiohjelmistoja ja palvelimia on tarkoitus käyttää (esim käyttöjärjestelmä, tietokannan hallintajärjestelmä, web-palvelin),
- millaisia väli- ja integrointiohjelmistoja (sovelluspalvelimet, middleware) on tarkoitus käyttää (esim. EJB-sovelluspalvelimet, ORB, Transactional/Message Oriented Middleware, Enterprise Service Bus)
- millaisiin standardeihin, protokolliin ja politiikkoihin organisaatio on sitoutunut,

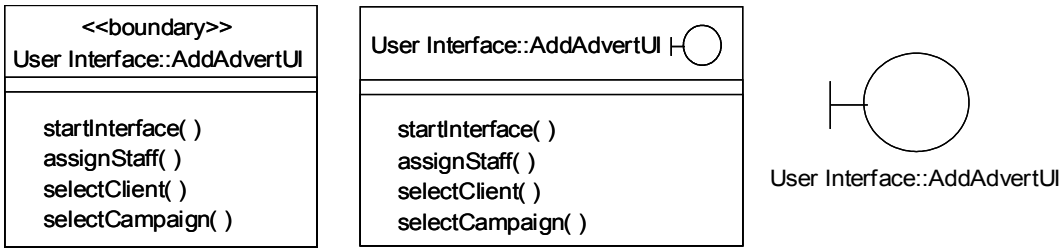
Yksityiskohtaisempia ratkaisuja arkkitehtuurin suhteen tehdään myöhemmissä vaiheissa. Tässä vaiheessa on hyödyllistä luoda "luuranko", jonka mukaisesti voidaan luokkia ryhmitellä ja "paketoida" eli muodostaa luokista *paketteja* (package). Kuvassa 4.2 (Rumbaugh ym., 1999, s. 380) on esitetty arkkitehtuuri, jossa ovat muiden muassa käyttöliittymään (UserInterface, GUIManager), tilausten käsittelyyn (Order Processing) ja muistinhallintaan liittyviä paketteja. Kaaviossa esiintyvät nuolet kuvaavat pakettien (tarkemmin pakettien sisältämien joidenkin luokkien) välisiä riippuvuussuhteita (dependency). Riippuvuussuhde kahden luokan välillä tarkoittaa sitä, että toista luokkaa muutettaessa täytyy varautua muuttamaan myös toista luokkaa.



Kuva 4.2: Paketteja ja niiden välisiä suhteita

Kurssilla keskitytään pääosin arkkitehtuurijäsennykseen, jossa järjestelmä jaetaan kolmeen tasoon (kerrokseen) niin, että käyttöliittymä on eristetty tietojen esitysmuodosta. Tämän mukaisesti järjestelmä koostuu kolmentyyppisistä olioluokista (vrt. kuvat 1.5 ja 4.6):

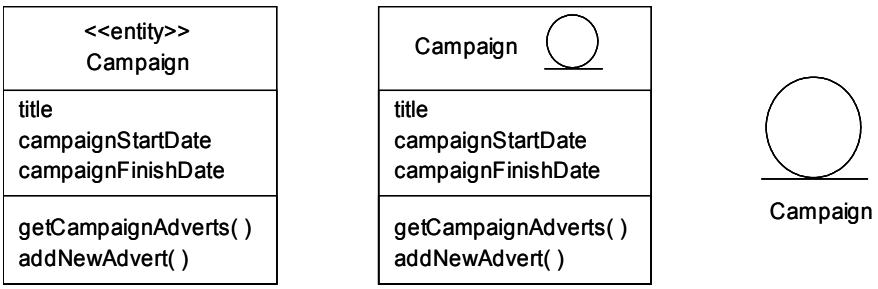
- Käyttöliittymäluokat, rajaluokat (*boundary* class, user interface class) (kuva 4.3; Bennett ym. 2006):
  - ◊ koostuvat niistä järjestelmän osista, joiden kautta järjestelmä on yhteydessä ympäristöönsä (esim. Anturi, Tutka, Ikkuna, Painike, Ohjauspaneeli, jne.)
  - ◊ löytyvät käyttötapauksista tai tarkastelemalla aktoreita yksi kerrallaan sen selvittämiseksi, millä tavalla ne ovat yhteydessä järjestelmään.



Kuva 4.3: Käyttöliittymäluokan vaihtoehtoiset esitystavat UML-luokka- ja yhteistoimintakaavioissa. <<boundary>> on stereotyyppi (yksi UML:n laajennusmekanismeista), jonka avulla tässä tarkennetaan luokan kuulumista käyttöliittymäluokkiin. Ikoninen notaatio on stereotyypin vaihtoehtoinen, visuaalinen esitys.

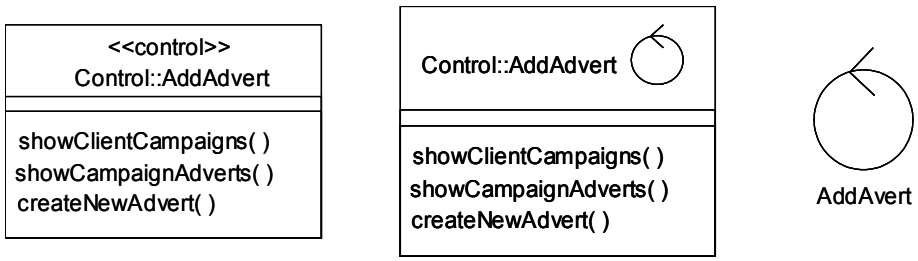
- Liiketoiminta- eli sovellusluokat, kohdealueluokat (*entity* class, domain classs) (kuva 4.4; Bennett ym. 2006),
  - ◊ ovat pysyväisluonteisia tieto-olioluokkia; esim. Asiakas, Tili, Pankki, Tilaus

◊ luokkien attribuutit kuvaavat käyttäjiä kiinnostavia asioita (esim. Asiakkaan nimi ja osoite, Tilin saldo)
◊ tyypillisiä operaatioita: attribuuttiarvojen tallentaminen ja hakeminen, arvojen muuttaminen/laskeminen sekä olion luonti ja poistaminen; joidenkin operaatioiden suorittamiseksi olio voi tarvita toisten olioiden apua, jota varten se lähettää viestejä,



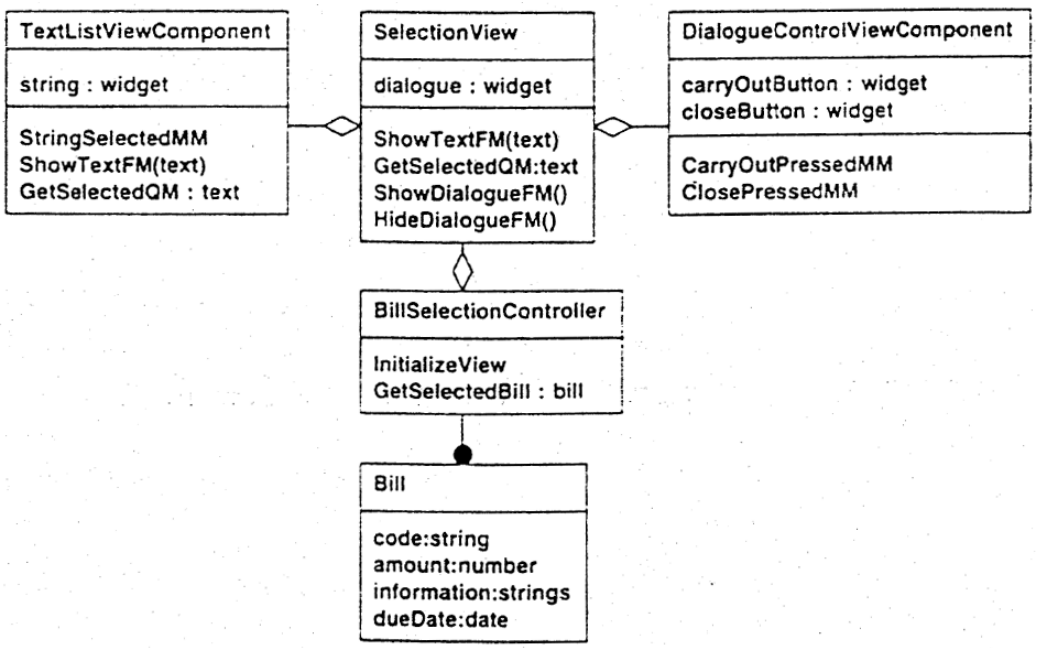
Kuva 4.4: Kohdealueluokan vaihtoehtoiset esitystavat UML-luokka- ja yhteistoimintakaavioissa.

● Ohjausluokat, ohjaimet (*control* class, controller) (kuva 4.5; Bennett ym. 2006):
◊ huolehtivat sellaisen käyttäytymisen aikaansaamisesta, jota ei voida luontevasti delegoida muille olioille; esim. yhteenvedon tuottaminen, ohjauksen jakaminen käyttöliittymäolioiden kesken ja käyttöliittymäolioista kohdealueluokkiin, palveluvuorojen jakaminen aktoreille jne
◊ yhtä käyttötapausta kohti voi olla nolla, yksi tai useampia ohjausluokkia. Edelleen kutakin ohjausluokkaa kohti voi olla yksi tai useampia ohjausolioita.
◊ ICONIX-menetelmässä analyysivaiheen dynaamisessa mallintamisessa etsittävät ohjaimet eivät välttämättä ole luokkia vaan operaatioita, vaikka niitä kuvataankin yhteistoimintakaaviossa luokkasymbolilla

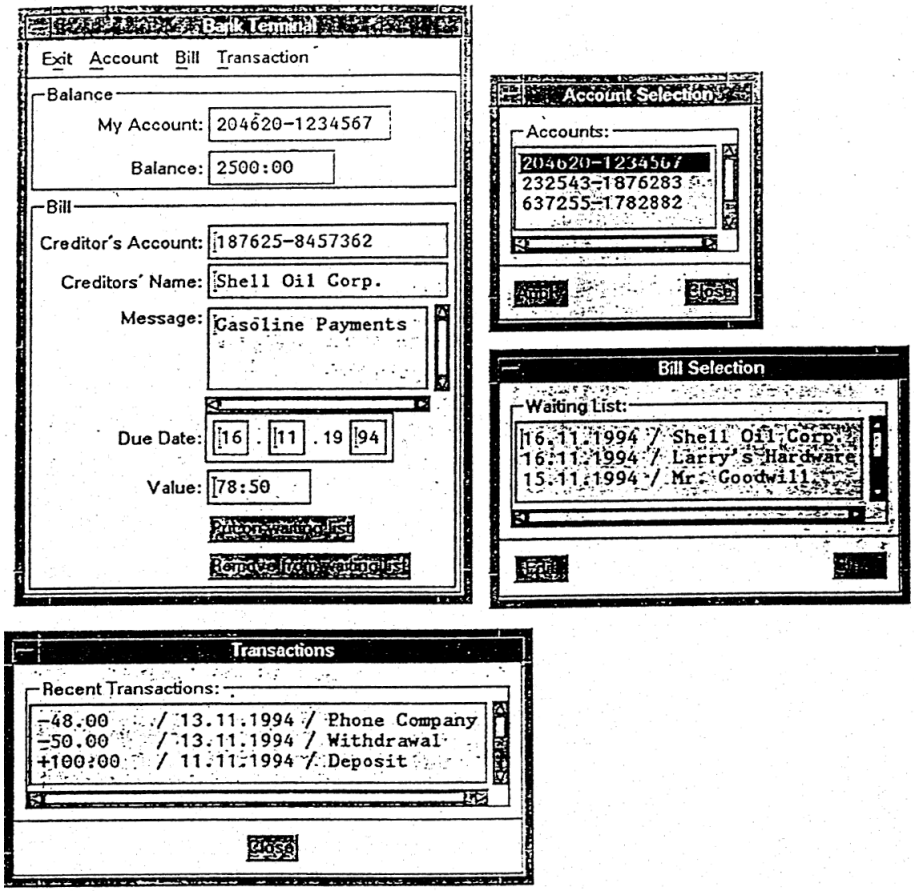


Kuva 4.5: Ohjausluokan vaihtoehtoiset esitystavat UML-luokka- ja yhteistoimintakaavioissa

Jacobson ym. (1992) kehottaa lähtemään mallintamisessa liikkeelle liiketoimintaluokista ja myöhemmin tarkastelemaan käyttöliittymäluokkia. Tämän jälkeen katsotaan, tulevatko kaikki käyttäytymispiirteet mallinnettua; jos eivät, otetaan käyttöön ohjausluokkia. Dynaamisessa sovelluksessa ohjausluokkien tunnistaminen jo alkuvaiheessa voi helpottaa luokkarakenteen hahmottumista.

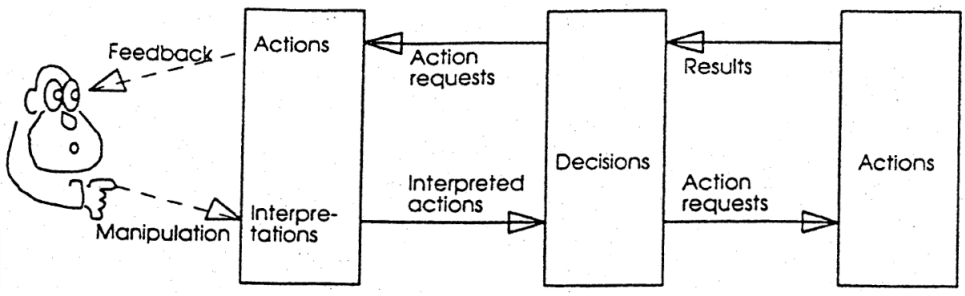


Kuva 4.6: Esimerkki pankkiautomaattijärjestelmän osan käyttöliittymä-, ohjaus-, ja liiketoimintaluokista



Kuva 4.7: Esimerkkikäyttöliittymä (Bill Selection -dialogi) kuvassa 4.6 esitetyille käyttöliittymä-, ohjaus-, ja liiketoimintaluokille.

Nokia on kehittänyt ko. luokkajakoon perustuvan ns. MVC++ -lähestymistavan (huom. nimestään huolimatta ei ole sama kuin luvussa 5.2.1.2 käsiteltävä MVC-arkkitehtuurimalli, vaan pikemminkin erikoistapaus 3-tasoarkkitehtuurista). Seuraavat periaatteet määrittävät luokkien välisiä suhteita (kuva 4.8):

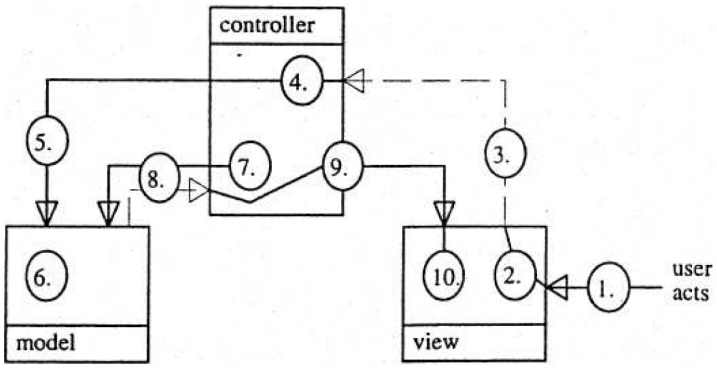


Kuva 4.8: MVC++-sovelluksen rakenne: käyttöliittymä-, ohjain- ja liiketoimintakerrokset

- kussakin järjestelmässä on yksi pääohjausluokka, jonka mukaiset oliot luovat ja ohjaavat pääkäyttöliittymäolioita,
- pääkäyttöliittymäolio muodostaa järjestelmän päänäytön,
- kukin ohjausolio alustaa ja ohjaa omaa käyttöliittymäoliotaan,
- pääohjausolio kontrolloi muita ohjausolioita ja välittää niiden sanomia toisille,
- ohjausoliot välittävät tietoa käyttöliittymäoliioden ja liiketoimintaolioiden välillä. Ne tekevät kaikki sovelluslogiikkaan kuuluvat päätökset. Vain ohjausoliot tietävät, mihin muut oliot pystyvät.
- käyttöliitymäolio tietää, miten tietoja esitetään käyttäjälle, miten käyttäjän tekemät valinnat tai antamat käskyt välitetään järjestelmälle (ohjausoliolle).

Kyseisten luokkien roolin ymmärtämiseksi tarkastellaan vielä tyypillistä toimintojen sarjaa (kuva 4.9):

1. Käyttäjä painaa käyttöliittymän painiketta.
2. Käyttäjän toimenpide tulkitaan käyttöliittymäolion toimesta.
3. Käyttöliittymäolio lähettää tulkinnastaan tiedon ohjausoliolle.
4. Ohjausolio päättää, mitä tällaisessa tilanteessa tulisi tehdä.
5. Omaan tietämykseensä (lue: operaatioihinsa) perustaen ohjausolio lähettää sanoman liiketoimintaoliolle/-olioille.
6. Liiketoimintaolio toimii itsenäisesti sanoman edellyttämällä tavalla.
7. Liiketoimintaolion lopetettua toimintansa kontrolli palaa ohjausoliolle. Se päättää, edellyttääkö tilanne käyttöliittymäolion päivitystä.
8. Jos edellyttää, ohjausolio hankkii tarvittavat tiedot liiketoimintaoliolta/-olioilta.
9. Ohjausolio välittää tiedot käyttöliittymäoliolle
10. Käyttöliittymäolio esittää tiedot käyttäjälle.



Kuva 4.9: MVC++ -sovelluksen toimintosarja

Kyseisen luokkajaon ymmärtäminen on hyödyllistä jo analyysivaiheessa, jotta osataan oikealla tavalla suhtautua "teksteistä poimittuihin" luokkiin ja ymmärretään niiden rooli. Kuitenkin vasta yksityiskohtaisen suunnittelun aikana olioluokkavalikoima tulee kokonaisuudessaan esiinotettavaksi.

Oikein toteutetun 3-tasoarkkitehtuurin seurauksena sekä käyttöliittymä että tietojen tallennusmekanismi ovat periaatteessa vaihdettavissa tai käytettävissä toisistaan riippumatta (esim. menupohjaisen tekstikäyttöliittymän päivitys graafiseen käyttöliittymään). Toisaalta eri kerroksissa olevissa luokissa voi olla toisteisuutta (esim. useimpia kohdealueluokkia vastaa käyttöliittymäkerroksella tietty ikkuna tai näyttö). On myös huomattava, että luokkien *looginen* jako kolmeen kerrokseen ei välttämättä vastaa järjestelmän *fyysistä* hajautusta. Arkkitehtuurin mallintamista käsitellään tarkemmin sekä loogisesta että fyysisestä näkökulmasta luvussa 5.1.

4.2 Staattinen mallintaminen (oliomallintaminen)

- tarkoituksena on rakentaa ymmärrettävä kuvaus niistä rakenteellisista ilmiöistä, joita vaatimusmäärittelyt koskevat
- lähtökohtina toimivat vaatimusmäärittelyt, käyttötapausmalli sekä sovellusaluetta ja sen ympäristöä koskeva asiantuntemus,
- tuloksena luokkakaavio (kohdealuemalli) + tietohakemisto

Kohdealuemalliin merkitään tavallisesti vain liiketoimintaluokat. Jos käyttöliittymän tai soveluslogiikan ominaisuuksiin halutaan ottaa kantaa tai se helpottaa muuten mallintamista, riittää merkitä yleiset (mahdollisesti käyttötapauskohtaiset) käyttöliittymä- ja ohjausluokat, mutta näiden tarkkoja omianaisuuksia (esim. käyttöliittymään kuuluvia yksittäisiä komponentteja) ei merkitä – nämä mallinnetaan joka tapauksessa suunnitteluvaiheen aikana tarkemmin. Sen sijaan liiketoimintaluokkien loogista rakennetta tulee päivittää myös suunnitteluvaiheen edetessä, jolloin järjestelmän sisäisestä tietomallista on jatkuvasti saatavilla korkealla abstraktiotasolla oleva malli, jota voi käyttää keskusteltaessa järjestelmän ominaisuuksista asiakkaan kanssa.

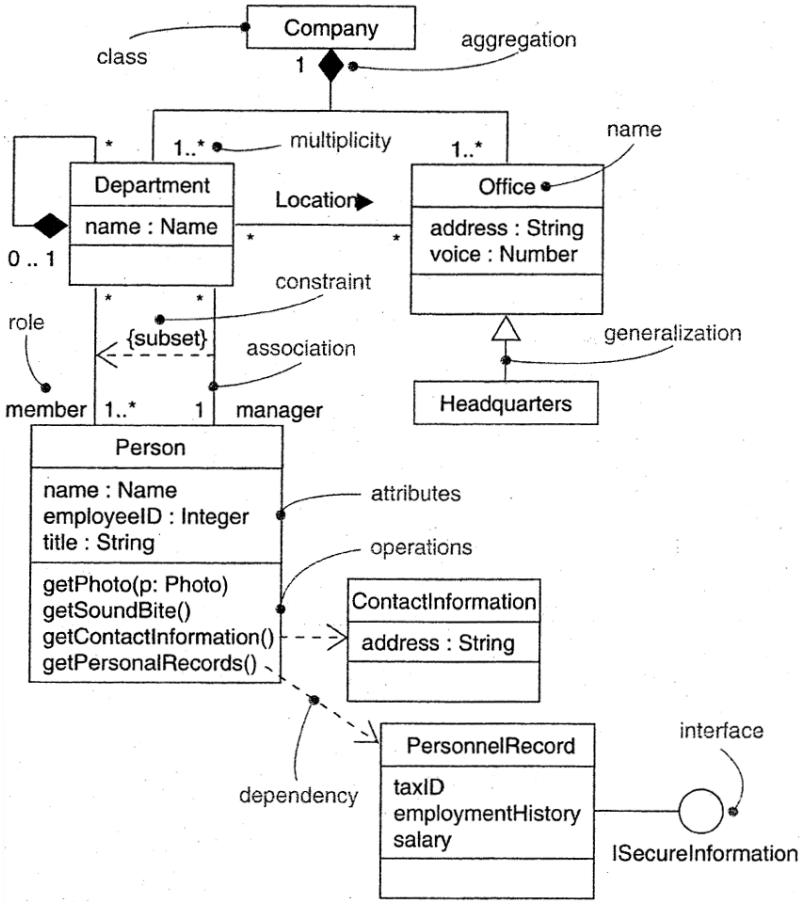
Seuraavassa määritellään ensin luokkakaavion peruskäsitteet, rakenteet ja notaatio UML:n mukaan ja sen jälkeen esitetään lyhyt kuvaus mallintamisaskeleista.

4.2.1 Luokkakaavio: peruskäsitteitä

Staattisessa mallintamisessa voidaan käyttää kahdenlaisia malleja: oliokaavioita ja luokkakaavioita. *Oliokaaviot* (object diagrams) kuvaavat yksittäisiä olioita ja niiden välisiä rakenteellisia suhteita. Näitä käytetään lähinnä monimutkaisten tietorakenteiden havainnollistamiseen. Varsinainen analyysi ja suunnittelu tapahtuu luokkakaavioilla.

Luokkakaavio (class diagram) kuvaa olioluokat ja niiden väliset suhteet. Luokkakaaviota voidaan pitää eräänlaisena kaavana tai mallina, joka määrittää, millaiset ajonaikaiset oliorakenteet ovat mahdollisia.

Kaaviot rakentuvat seuraavista käsitteistä: olio, luokka, attribuutti, operaatio, assosiaatio, linkki, kardinaalisuus, rooli, kooste, kompositio ja yleistys. Osaa näistä on käsitelty jo luvussa 1.2. Seuraavassa määritellään käsitteet ja esitetään luokkakaavion notaatio UML:n mukaisesti. Notaatiot on esitetty tiivistetysti myös liitteessä A.



Kuva 4.10: Luokkakaavio

Olio on rajattavissa ja yksilöitävissä oleva asia tai käsite, joka on merkityksellinen käsillä olevan tarkastelun kannalta ja joka kattaa sekä rakenteen (tilan) että käyttäytymisen

- notaatio: suorakulmio, jonka yläreunassa esitetään nimi alleviivattuna. Nimi koostuu tavallisesti kahdesta kaksoispisteellä erotellusta osasta: olion nimi ja luokan nimi.

Luokka (olioluokka) (kuva 4.10; Booch ym. 1999, s. 166) kuvaa joukkoa rakenteeltaan ja käyttäytymiseltään samanlaisia olioita

- notaatio: suorakulmio, joka koostuu 1-3 osasta: yläosassa ilmaistaan luokan nimi, keskiosassa attribuutit ja alaosassa operaatiot.

Attribuutti kuvaa luokkaan kuuluvien olioiden rakenteellista ominaisuutta

- voidaan tulkita olion sisäiseksi, nimetyksi tietokentäksi (paikallinen tietorakenne)
- attribuutille voidaan antaa tyyppi (data type) ja oletusarvo
- attribuuttiarvo yksilöi ominaisuuden
- arvo voi olla yksinkertainen (luku, merkki, joissakin kielissä myös merkkijono; esim. paino), rakenteinen (tietue tai lueteltu tyyppi; esim. osoite) tai kokoelma yksinkertaisia tai rakenteisia arvoja (esim. taulukko tai järjestetty joukko koordinaattipareja),
- arvona voi olla myös osoitin tai viite olioon (*viitearvoinen* eli olioarvoinen attribuutti) tai kokoelma viitteitä; tällainen attribuutti tarkoittaa, että on olemassa yhteys attribuutin sisältävän olion ja attribuutin arvon osoittaman olion välillä.
- perustyyppien ja tietueiden (jos kielessä tuki tietueille) arvoilla ei ole identiteettiä, toisin kuin olioilla (huom. olion käsite vaihtelee eri kielissä – tarkennettava päätettäessä olioiden esitystavasta, katso luku 5.3.1.4)
- *perusattribuutin* (base attribute) arvo on riippumaton toisten attribuuttien arvoista (esim. osoite)
- *johdetun* attribuutin (derived attribute) arvo voidaan laskea tai päätellä muiden attribuuttien arvoista (esim. vuosipalkka)
- attribuuttiarvo esitetään vastaavan olion yhteydessä ja attribuutti luokan yhteydessä

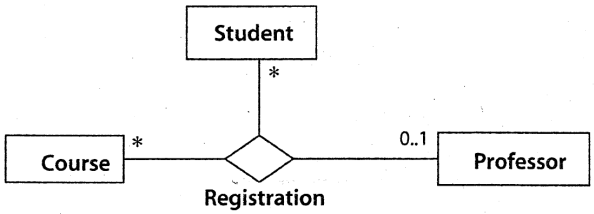
Operaatio on olioon kohdistuva tai olion suorittaman toimenpiteen määritys

- *Metodi* on tietyn luokan olioihin sovitettu operaation toteutus. Joissakin uml-työkaluissa luokkakaavion operaatiolokeron rivejä kutsutaan suoraan metodien määrittelyksi, joten termejä voi pitää synonyymeinä, ellei ole erityistä tarvetta korostaa metodien määrittelyn ja toteutuksen eroa
- operaatioilla ja metodeilla voi olla parametreja ja palautustyyppi (return type)
- operaation kaikilla metodeilla tulee olla sama otsake eli kutsumuoto (signature) ja sama parametrimäärä (esim. *move(delta: vector)*)
- esitetään vastaavan luokan yhteydessä

Seuraavaksi määritellään käsitteet, joilla luokista ja olioita voidaan rakentaa rakenteellisia kokonaisuuksia (so. luokkakaavio ja oliokaavio)

Assosiaatio on kahden tai useamman luokan välinen pysyväisluonteinen suhde (esim. Töissä), myöhemmin assosiaatio voidaan mallintaa jollakin spesifisemmällä suhteella (esim. koostesuhde)

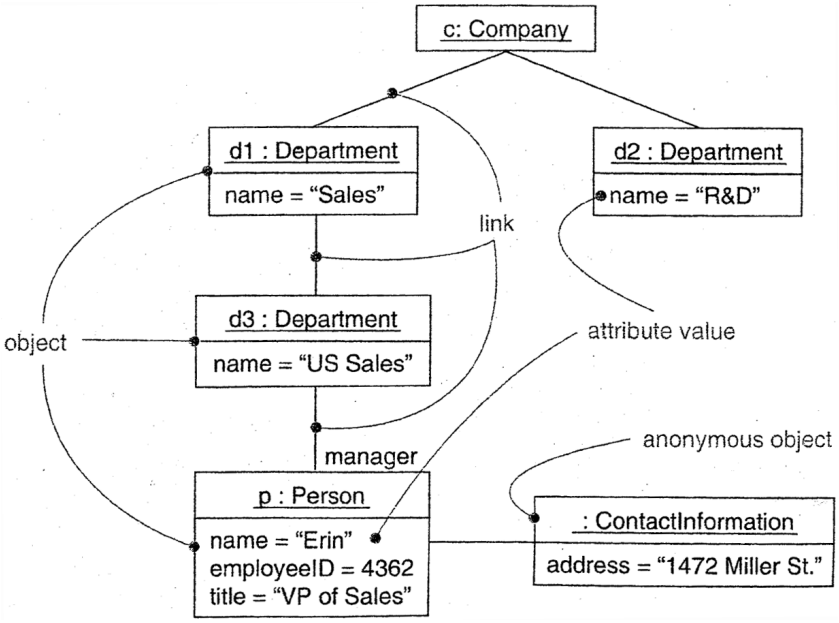
- notaatio:
  - ◊ binaarinen assosiaatio esitetään viivalla, joka yhdistää luokat,
  - ◊ useamman kuin kaksi luokkaa yhdistävä assosiaatio (monikkoassosiaatio, yleisesti n-ary association) esitetään vinoneliöllä, joka yhdistetään viivoin luokkiin (kuva 4.11; Rumbaugh ym., 1999 s. 351),
  - ◊ assosiaation nimi kirjoitetaan viivan yhteyteen; lisäksi voidaan käyttää nuolenpäätä tekstin vieressä osoittamaan, mihin "suuntaan ko. nimi luetaan".



Kuva 4.11: Kolmipaikkainen (ternary) assosiaatio

Linkki (kuva 4.12; Booch ym. 1999 s. 196) on kahden tai useamman olio(ilmentymä)n välinen suhde,

- assosiaatiota vastaava ilmentymätasoinen käsite (toteutus osoittimella tai viitteellä),
- notaatio: esitetään viivalla ja tarvittaessa (monikkoassosiaation yhteydessä) lisäksi vinoneliöllä, tässä tapauksessa kuitenkin olioiden yhteydessä,



Kuva 4.12: Oliokaavio ja linkki

Assosiaatioita halutaan usein kuvata tarkemmin. Tätä varten on käytettävissä useita lisäkäsitteitä ja notaatioita, joista muutamia selvitetään seuraavassa.

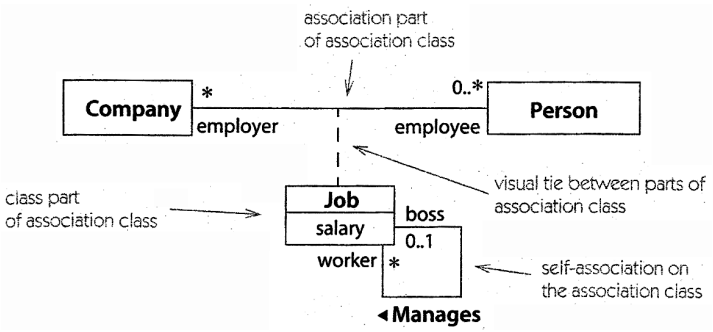
Kardinaalisuus eli *kertautuminen* (cardinality, multiplicity) (kuva 4.13) ilmaisee, kuinka monta oliota vähintään/enintään kustakin olioluokasta osallistuu kerrallaan assosiaatioon,

- notaatio: assosiaatioviivan päiden yhteydessä merkinnöillä 0..1 (yksi tai ei yhtään), 0..* (ei yhtään, yksi tai monta), 1..* (yksi tai monta), * (ei yhtään, yksi tai monta) ja 1 (täsmälleen yksi); voidaan käyttää myös tarkempia numeerisia merkintöjä (esim. 3..5)

Joskus assosiaatioon liittyy luontaisesti omia attribuutteja (esim. vihkimispäivä, työntekijän tekemät työtunnit tietylle projektille). Tällöin voidaan määritellä *assosiaatioluokka* joka sisältää sekä assosiaation että ko. assosiaatioon liittyvien attribuuttien piirteet (kuva 4.13; Rumbaugh ym, 1999 s. 159). Assosiaatioluokkaan voidaan sijoittaa myös operaatioita, joilla ko. attribuutteihin päästään käsiksi.

Rooli (kuva 4.13) viittaa tehtävään, asemaan tai tilaan, joka olioluokan mukaisilla olioilla on assosiaatiossa (esim. työntekijä, työnantaja). Roolit ovat tarpeellisia erityisesti unaarisuhteissa (saman luokan olioiden välinen suhde) tai jos kahden luokan välillä on useampia assosiaatioita.

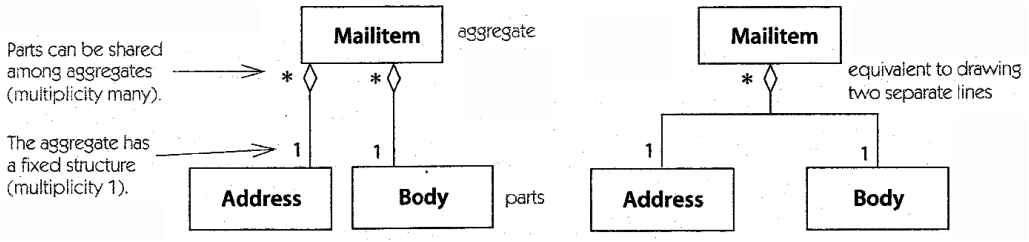
- notaatio: roolinimet kirjataan assosiaatioviivan molempiin päihin,



Kuva 4.13: Assosiaatioluokka, linkit ja roolit

Kooste (aggregation) (kuva 4.14; Rumbaugh ym. 1999, s. 159) on erityinen assosiaatio kahden luokan välillä, joista toinen on kokonaisuus (whole) ja toinen osa (part).

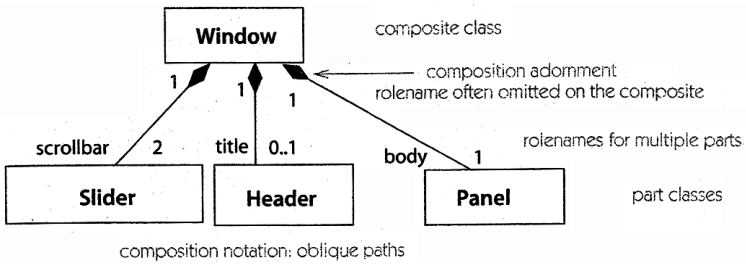
- tarkoittaa, että kokonaisuusluokan mukainen olio koostuu osaluokan mukaisista olioista (esim. Auto koostuu Korista, Moottorista, Pyöristä, jne) - *has-a*-suhde,
- koostesuhteen ominaisuuksia: transitiivisuus (jos a:lla on b ja b:llä on c, a:lla on c), ei-symmetrisyys,
- notaatio:
  - ◊ esitetään sijoittamalla vinoneliö (salmiakki) assosiaatioviivan kokonaisuusluokan puoleiseen päähän,
  - ◊ Viivan yhteydessä voidaan käyttää kardinaalisuusmerkintöjä tavallisen assosiaation tapaan,
  - ◊ vaikka kooste kytkee kerrallaan vain kaksi luokkaa toisiinsa, voidaan esityksessä koosteita yhdistellä esimerkiksi hierarkkiseksi puuksi.



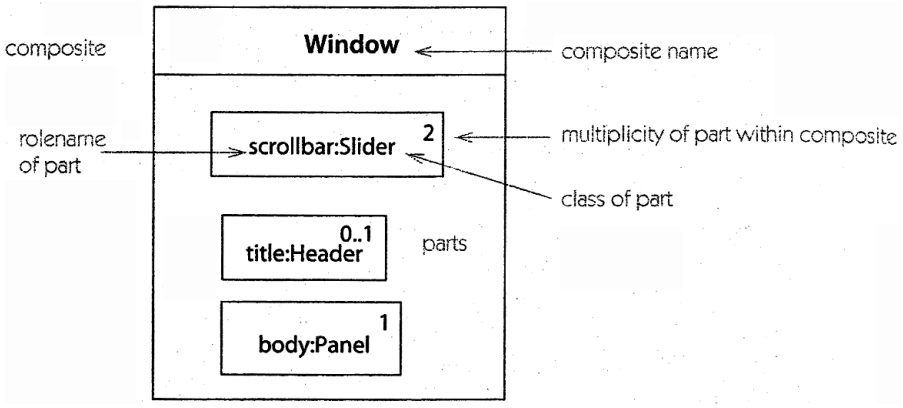
Kuva 4.14: Kooste ja sen osat vaihtoehtoisilla asetteluilla

Kompositio (composition) (kuva 4.15; Rumbaugh ym. 1999, 229-230) on erityinen kooste, jossa osaolio voi sisältyä vain yhteen kokonaisuusolioon ja kerran "synnyttäjään" osaolio on sidottuna kiinteästi kokonaisuuteen. Siten esimerkiksi kokonaisuusolion poisto johtaa myös osaolioiden poistamiseen. Osaolion poisto ei johda kuitenkaan välttämättä kokonaisuusolion poistamiseen.

- Notaatio: esitetään kuten kooste, mutta vinoneliö on tummennettu



Kuva 4.15: Kompositio käyttöliittymän ikkunan mallintamiseksi



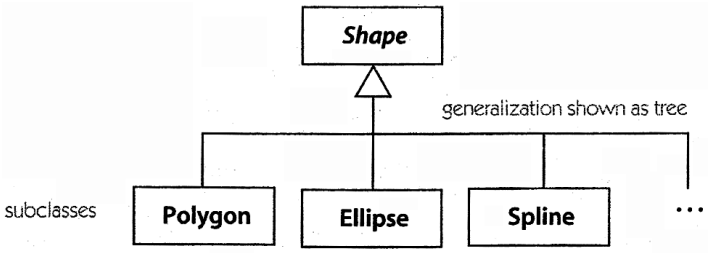
Kuva 4.16: Sama kompositio graafisesti yhdisteltynä kokonaisuutena

Lopuksi tarkastellaan luokkien välisiä yleistyssuhteita. Nämä ovat erittäin tärkeitä koodin ja mallien uudelleenkäytön kannalta.

Yleistys eli perintä (generalization) (kuva 4.17; Booch ym, 1999 s. 141):

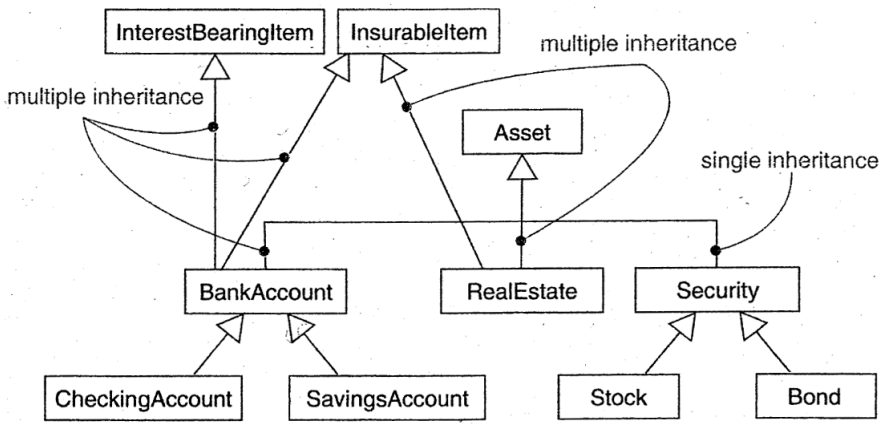
- on erityinen assosiaatio *yliluokan* (superclass, parent) ja sen *aliluokan* (subclass, child) välillä
- tarkoittaa, että aliluokka tarkentaa (refine) eli erikoistaa (specialize) yliluokkaansa ja yliluokka yleistää (generalize) aliluokkaansa,
- samaistetaan joskus is_a -suhteeseen (Sihteeri is_a Henkilö),

- aliluokat perivät (inherit) yliluokkiensa ominaisuudet; aliluokilla voi olla lisäksi omia erityisominaisuuksiaan (attribuutteja ja operaatioita)
- aliluokka voi *määritellä uudelleen* (override) perimänsä attribuutin tai operaation eli tehdä siitä erikoistuneen version,
- yleistys/erikoistaminen voi johtaa seuraaviin aliluokkiin:
  - ◊ *erillisiin* (disjoint) tai *päällekkäisiin* (overlapping) aliluokkia vastaaviin olio-joukkoihin,
  - ◊ *kattavaan* (complete) tai *osittaiseen* (incomplete) olioiden jakoon,
- notaatio: esitetään viivalla, jonka iso nuolenpää osoittaa yliluokkaan (kuva 4.18; Rumbaugh ym, 1999 s. 289). Yleistys-/erikoistamismuoto voidaan ilmaista rajoitteena (kuva 4.19; Rumbaugh ym. 1999, s. 290).



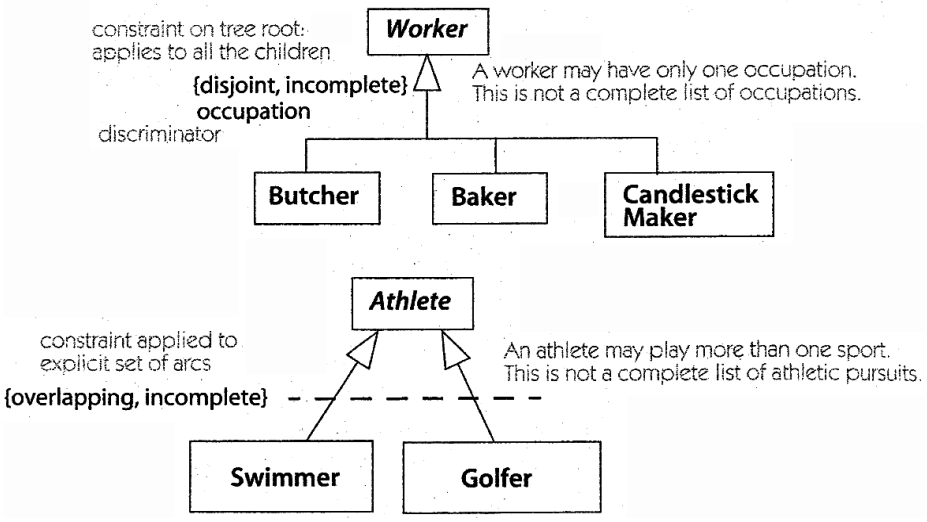
Kuva 4.17: Yleistys ja erikoistaminen

Kohdealuetta mallinnettaessa yleistyshierarkian tärkein motiivi on kuvausvaivan säästäminen ja erilaisten sääntöjen täsmällinen esittäminen. Jos usealla luokalla on samoja attribuutteja ja luokille voidaan määritellä yhteinen yliluokka, voidaan yhteiset attribuutit liittää yliluokkaan ja määritellä ne vain kertaalleen. Analyysivaiheessa lähtökohtana ovat kohdemaailman ilmiöiden väliset suhteet. Myöhemmin suunnittelun kuluessa määritellään yleistys /perintäsuhteita toteutusratkaisujen näkökulmasta.



Kuva 4.18: Yleistys ja erikoistaminen (perintä ja moniperintä)

Luokkakaavio sisältää paljon muitakin erikoiskäsitteitä ja –rakenteita. Niitä tarvitaan kuitenkin lähinnä suunnitteluvaiheessa. Tästä syystä ne esitellään luvussa 5.3.1.1.



Kuva 4.19: Rajoitteet yleistyksessä ja erikoistamisessa

4.2.2 Staattisen mallintamisen askeleet

Seuraavassa kuvataan lyhyesti, millä askeleilla luokkakaaviota ja sitä tukevaa tietohakemistoa tehdään (Rumbaugh ym, 1991; Jacobson ym, 1999).

4.2.2.1 Luokkien tunnistaminen

Olioiden ja luokkien tunnistamiselle ei voida antaa ehdottomia ja yksiselitteisiä ohjeita. Sen sijasta seuraavassa on listattu joitakin vihjeitä (esimerkit viittaavat liitteenä olevaan ATM-esimerkkiaineistoon, Rumbaugh ym, 1991):

- lähtökohtana toimivat vaatimusmäärittelyt, käyttötapaukset ja keskustelut käyttäjien kanssa,
- kiinnitä huomiota erityisesti usein toistuviin substantiiveihin,
- aloita liiketoimintaluokista ja vältä toteutukseen liittyvien seikkojen mukaanottamista alkuvaiheessa,
- pyri löytämään mahdollisimman kuvaavat nimet,
- älä tässä vaiheessa ole huolestunut monimutkaisista rakenteista (koosteet, yleistykset),
- luokkalistasi voi sisältää alussa sellaisiakin, joista jotkut myöhemmin karsiutuvat.

Luokaksi sopivuuden tarkastelua:

- *redundantteja* (päällekkäisiä) luokkia: sisältävät samoja ominaisuuksia vaikka erinimisinä (esim. Customer, User), valitse kuvaavin nimi
- *epärelevantteja* luokkia: eivät ole ongelman eivätkä sen ratkaisun kannalta olennaisia (esim. Cost)
- *epämääräisiä* luokkia: ovat rajaukseltaan tai sisällöltään täsmentymättömiä (recordkeeping provision).
- pikemminkin *attribuutteja*: kuvaavat johonkin muuhun luokkaan kuuluvia ominaisuuksia, eivät ole itsenäisiä (esim. Account data)
- pikemminkin *operaatioita*: kuvaavat johonkin luokkaan kuuluvien olioiden käyttäytymistä (esim. Call)

- pikemminkin *rooleja*: nimi jo viittaa enemmän olion asemaan suhteessa kuin sen perimmäiseen luonteeseen
- heijastaa toteutuspiirteitä: analyysin yhteydessä keskitytään sovellusalueen ilmiöihin, ei toteutukseen, minkä vuoro tulee suunnittelun aikana (esim. Software, Communication line). Tyypillisiä suunnitteluun ja toteutukseen liittyviä luokkia ovat säiliöluokat, tietojen pysyvyyteen liittyvät luokat, käyttöliittymäkomponentit, käyttöjärjestelmään tai yksittäiseen ohjelmointikieleen liittyvät luokat sekä muut esim. tietojen haussa ja muussa käsittelyssä käytettävät ohjausluokkien apuluokat.

4.2.2.2 Tietohakemiston alustaminen

Luokkia, assosiaatioita, attribuutteja ja operaatioita kuvaavat nimet eivät mitenkään riitä kertomaan, mitä tarkasti ottaen kullakin käsitteellä tarkoitetaan. Tästä syystä käytetään tietohakemistoa eli mallisanastoa lyhyen luonnehdinnan esittämiseen kustakin keskeisestä käsitteestä. Olioluokkien osalta sen sisältönä on:

- luokan lyhyt kuvaus, joka sisältää myös mahdolliset olettamukset ja rajoitukset luokan jäsenyyden suhteen (vrt. ATM:n tietohakemisto liitteessä).

4.2.2.3 Assosiaatioiden tunnistaminen

Mikä tahansa kytkentä tai riippuvuus luokkien välillä on potentiaalinen assosiaatio. Seuraavia vihjeitä voidaan antaa niiden löytämiseksi:

- lähde liikkeelle vaatimusmäärittelystä, käyttötapauksista sekä luokkalistasta,
- etsi tekstistä verbejä,
- kiinnitä huomiota ilmiöihin, jotka kuvaavat fyysistä sidosta, johonkin kohdistuvaa toimintaa, yhteydenpitoa, omistusta tms.
- Mieti, mitkä olioiden väliset kommunikointiyhteydet (vrt. sekvenssi- ja yhteistoimintakaaviot) edellyttävät assosiaation olemassaoloa,
- älä ole alussa turhan valikoiva.

Assosiaatioiden sopivuuden tarkastelu:

- poistettujen luokkien väliset assosiaatiot: jos luokat on poistettu, voivat assosiaatiotkin olla turhia (esim. ATM prints receipts),
- epärelevantit tai toteutusta koskevat assosiaatiot: poistetaan (esim. System handles concurrent access)
- pikemminkin operaatio: assosiaation tulee kuvata rakenteellista piirrettä ongelma-alueella, ei sanomaa eikä operaatiota (esim. ATM interacts with user)
- monimutkaiset assosiaatiot: useampaa kuin kahta olioluokkaa koskevat assosiaatiot voidaan usein pilkkoa binaarisiksi assosiaatioiksi (esim. Company employes person with a salary)
- johdetut assosiaatiot: poistetaan assosiaatiot, joiden merkitys käy ilmi muiden assosiaatioiden kautta (esim. Consortium shares ATM’s).

Lisäohjeita:

- käytä sopivia assosiaationimiä
- lisää roolinimiä tarvittaessa,

- lisää kardinaalisuuksia kuvaavia merkintöjä
- selvitä, puuttuuko joidenkin luokkien väliltä assosiaatioita (vrt. sekvenssi- ja yhteistoimintakaaviot) ja lisää ne

4.2.2.4 Attribuuttien tunnistaminen

Attribuutit kuvaavat luokkaan kuuluvien olioiden rakenteellisia ominaisuuksia (esim. nimi, paino, väri). Ohjeita attribuuttien tunnistamiseksi:

- löytyvät harvemmin tavoitemäärittelystä, tarvitaan lisänä yleistietämystä,
- esiintyvät usein substantiiveina, ovat alisteisia luokille (esim. Auton väri)
- adjektiivit ilmaisevat yksittäisiä attribuuttiarvoja (esim. Auto on sininen)
- ota mukaan vain todella tarpeelliset attribuutit,
- vältä johdettujen attribuuttien mukaanottamista,
- ota huomioon myös assosiaatioita kuvaavat attribuutit

Attribuuttien sopivuuden tarkastelu:

- pikemminkin olioluokkia: jos attribuutista ollaan kiinnostuneita muutenkin kuin karakterisoivana ominaisuutena (esim. City), se on mahdollisesti olioluokka
- pikemminkin tarkentimia: jos attribuutin arvo on kontekstisidonnainen (esim. Employee number, Department name)
- pikemminkin oliotunniste; jos attribuutilla ei ole todellista vastinetta, jätä se olion pysyvyyden toteuttamismekanismin haltuun suunnitteluvaiheessa (esim. Transaction ID),
- pikemminkin assosiaatiota kuvaava attribuutti: jos ominaisuus liittyy assosiaatioon eikä yksittäiseen luokkaan, tee assosiaatioluokka,
- sisäisiä arvoja: jos attribuutti kuvaa olion sisäistä tilaa, jonka ei ole tarkoituskaan näkyä ulkopuolelle, poistetaan se,
- yhteensopimattomia attribuutteja: jos jotkut attribuutit poikkeavat merkittävästi luokan muista attribuutteista luonteensa puolesta, voi olla syytä jakaa luokka kahtia.

4.2.2.5 Luokkakaavion iterointi

Luokkien uudelleenorganisointi

Luokat on tähän mennessä kytketty toisiinsa assosiaatioilla. Osalla luokista voi olla paljonkin samanlaisia ominaisuuksia (esim. attribuutteja). Yleistyksellä on mahdollista selkiyttää luokkakaaviota:

- alhaalta-ylös: etsitään samanlaisia attribuutteja ja assosiaatioita sisältäviä luokkia ja selvitetään, ovatko ne osa yleistyshierarkiaa (Remote transaction, Cashier transaction).
- ylhäältä alas: tarkastellaan luokkia, joilla on paljon attribuutteja. Ovatko attribuutit relevantteja kaikille ko. luokan olioille?
- tukeudu moniperintään luokista vain perustelluissa tapauksissa (useista rajapinnoista periminen Java-kielen tapaan on huomattavasti luokista perintää yksinkertaisempaa) – moniperinnän käyttö edellyttää käytetyn kielen perintämekanismin

perusteellista ymmärtämistä ja vaatii luokkahierarkian kokonaisvaltaista suunnittelua[18]. Jos toteutuskieli ei tue moniperintää, suunnitteluvaiheessa tulee myös määrittää, miten moniperintä korvataan koostamisella.

Tarpeellisten saantipolkujen tarkistus

Nojaten luokkien välisiin assosiaatioihin ja niiden yhteyteen liitettyihin kardinaalisuusmäärityksiin tarkistetaan, vastaako rakenne sitä kuvaa, jonka esimerkiksi käyttötapauskuvaukset antavat.

- *ristiriitainen*, jos esimerkiksi vastauksena kyselyyn tulisi saada yhden olion tiedot ja rakenteen mukaan assosiaatioviivan päässä on kardinaalisuusmääre “moneen”.
- *puutteellinen*, jos kyselyyn ei löydy rakenteesta vastausta ollenkaan,
- *vääristynyt*, jos kuva yksinkertaisesta todellisuuden ilmiöstä esiintyy kompleksisena rakenteena kaaviossa

Luokkakaavion parantaminen

Luokkakaaviota ei saada koskaan valmiiksi kertasuorituksena, vaan tarvitaan useampia iterointikierroksia. Osa näistä kierroksista tehdään rinnan dynaamisen mallintamisen kanssa. Seuraavassa on joitakin ohjeita korjaustarpeiden löytämiseksi ja korjausten toteuttamiseksi:

- Merkkejä puuttuvista luokista:
  - ◊ assosiaatioiden ja yleistyssuhteiden epäsymmetrisyys: lisää uusia luokkia
  - ◊ epäyhtenäinen joukko attribuutteja ja operaatioita luokassa: jaa luokka osiin
  - ◊ vaikeuksia yleistyksessä (esim. jollakin luokalla voi olla kaksi roolia): jaa luokka
  - ◊ operaatio, jolla ei ole selvää olioluokkaa: lisää luokka
  - ◊ kaksi samannimistä ja samansisältöistä assosiaatiota: yleistä ja luo yliluokka yhdistämään niitä
- Merkkejä tarpeettomista luokista:
  - ◊ luokkaan ei löydy attribuutteja, assosiaatioita eikä operaatioita
- Merkkejä puuttuvista assosiaatioista:
  - ◊ puuttuvat saantipolut; lisää assosiaatioita, joiden kautta “kulkien” kysely voidaan suorittaa.
- Merkkejä tarpeettomista assosiaatioista:
  - ◊ toisteista tietoa assosiaatioissa,
  - ◊ ei löydetty operaatioita, jotka hyödyntäisivät assosiaatioiden muodostamia polkuja

4.2.3 Vinkkejä staattiseen mallintamiseen

- Koeta ensin ymmärtää ongelma-alue, vasta sitten voit määritellä luokkia.
- Pyri pitämään luokkakaavio yksinkertaisena.
- Valitse huolellisesti nimet ja selvennä tarvittaessa määrittelyä tietohakemistossa.
- Mallinna analyysivaiheessa viitteet ja osoittimet luokasta toiseen assosiaatioina, älä attribuutteina

[18] Moniperinnän käytöstä C++:ssa ks. esim. V. Lappalainen & R. Lahdelma (1996), Olio-ohjelmointi ja C++. http://users.jyu.fi/~vesal/kurssit/winohj/html/cpp/m65.htm

- Useimmat monimutkaiset assosiaatiot voidaan jakaa binaarisiksi assosiaatioiksi
- Kardinaalisuusmääritteitä voidaan tarkentaa pitkin matkaa.
- Suhteita koskevat attribuutit assosiaatioluokiksi, ei assosiaation yhdistämien luokkien attribuuteiksi
- Vältä liian syviä yleistyshierarkioita
- Varaudu siihen, että luokkakaavion rakentaminen edellyttää lukuisia korjaus- ja täydennyskierroksia.
- Keskustele käyttäjien kanssa kaavion sisällöstä
- Ole huolellinen dokumentoinnissasi.
- Käytä vain niitä oliomallin käsitteitä ja rakenteita, jotka tuntuvat tarpeellisilta ko. tilanteessa.

4.3 Dynaaminen mallintaminen

Dynaamisen mallintamisen

- tarkoituksena on määritellä olioiden käyttäytyminen: olioiden keskinäisenä vuorovaikutuksena (sekvenssikaavio, yhteistoimintakaavio) ja yksittäisten olioiden tiloina ja tilanvaihtoina (tilakaaviot)
- lähtökohtina toimivat vaatimusmäärittelyt, käyttötapaukset, luokkakaavio sekä yleinen asiantuntemus
- tuloksena saadaan skenaariota, sekvenssikaavioita (sequence diagrams), yhteistoimintakaavioita (collaboration diagrams, robustness diagrams), toimintakaavioita (activity diagrams, katso luku 3.3) ja tilakaavioita (state diagrams, katso luku 5.3.2.1) niiden olioiden tai olioluokkien osalta, joiden käyttäytymisen selvittäminen yksityiskohtaisesti erityisesti kiinnostaa.

Seuraavassa esitellään dynaamisten mallien peruskäsitteet, rakenteet ja notaatiot ja sen jälkeen kuvataan askeleet eri menetelmillä. Luvussa käsitellään myös CRC-kortteja, joita voidaan käyttää sekä staattisen että dynaamisen mallintamisen yhteydessä hahmotettaessa luokkien vastuualueita ja yhteistoimintaa.

4.3.1 Dynaamiset mallit: peruskäsitteitä

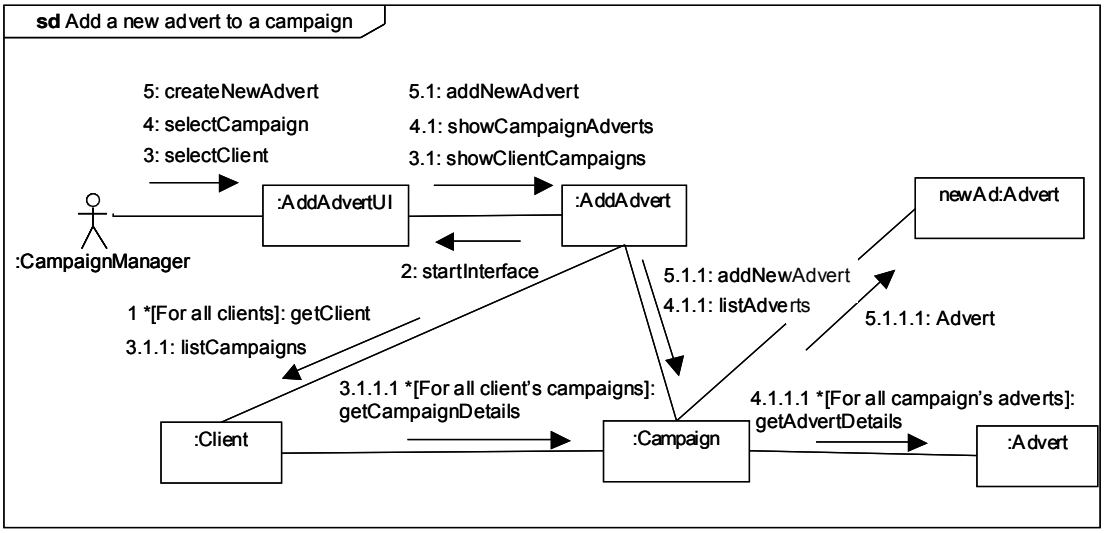
Dynaamiset mallit kuvaavat järjestelmän käyttäytymistä joko olioiden yhteistyönä (collaboration) vuorovaikutuksen (interaction) muodossa tai yksittäisten olioiden tilasiirtyminä. Luokkakaaviossa näkyvät olioiden suorittamat toiminnot operaatioiden muodossa, mutta ei se, millaista yhteistoimintaa (tietojen tai palvelujen pyytämistä ja saamista) operaatioiden suorittaminen edellyttää.

4.3.1.1 Yhteistoimintakaavio

Yhteistoimintakaavio (collaboration diagram, UML 2.0:ssa myös viestintäkaavio, communication diagram) (kuva 4.20; Bennett ym, 2006) kuvaa tavallisesti yhden käyttötapauksen osalta viestien kulkua oliolta toiselle tavalla, joka tuo erityisesti esille sen, miten yhteistoiminnassa käytetään hyväksi olioiden välisiä linkkejä (huom. olio voi lähettää viestin vain sellaiselle olioille, jonka olemassaolon se tietää),

- notaatio:
  - ◊ toimintaan osallistuvat oliot esitetään nimetyillä suorakulmioilla (tai ikonista notaatiota käytettäessä visuaalisilla boundary/control/entity-stereotyypeillä)
  - ◊ viivat olioiden välillä vastaavat linkkejä,
  - ◊ viivan yhteyteen piirretään nuoli kuvamaan viestinnän suuntaa sekä kirjoitetaan viestin nimi.
  - ◊ Kunkin viestin eteen kirjoitetaan järjestysnumero, joka osoittaa, missä järjestyksessä ko. viestit lähetetään; järjestysnumero voi ilmaista peräkkäisyyttä (1, 2, ..) ja/tai sisäkkäisyyttä (2., 2.1, 2.2, ...)
  - ◊ Toisteisen toiminnan (”for/while-silmukat”) kuvaaminen voidaan tehdä sijoittamalla toiston aloittavan viestin nimen eteen tähti (*).
  - ◊ Ehdollisen toiminnan (”if/else/switch-lauseet”) kuvaaminen voidaan tehdä sijoittamalla ehdollisen viestin eteen hakasulkeilla *[]* ympäröity vapaamuotoinen ehtolause (*guard condition*).

Sekvenssikaavioita ja yhteistoimintakaavioita kutsutaan yhtenäisellä nimellä *vuorovaikutuskaaviot* (interaction diagrams). Sekä sekvenssi- että yhteistoimintakaavioita voidaan mallintaa useilla eri tarkkuustasoilla, joista tarkempia esimerkkejä on luvussa 4.3.2.

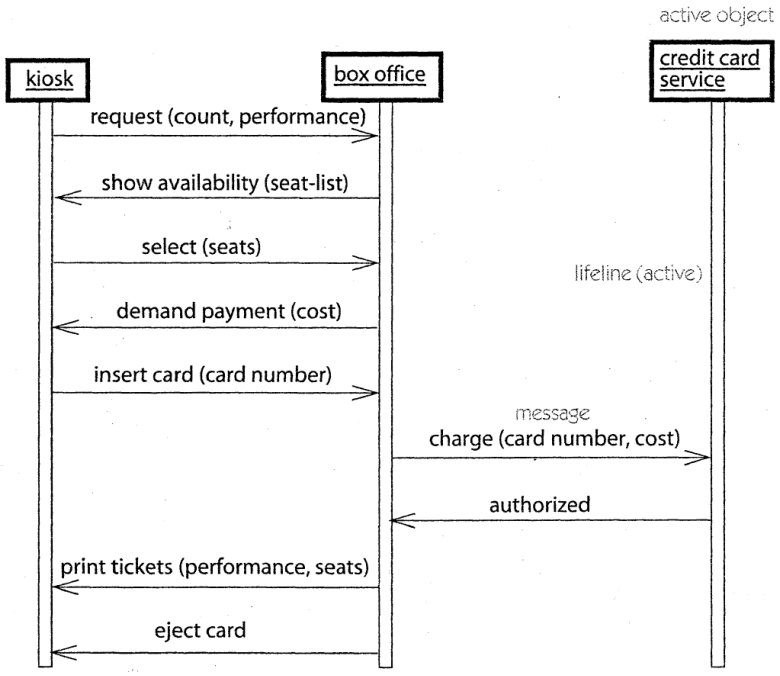


Kuva 4.20: Yhteistoimintakaavio käyttötapaukselle "lisää uusi mainos kampanjalle".

4.3.1.2 Sekvenssikaavio

V*iesti* (message):

- on *signaali*, joka välitetään lähettävältä oliolta vastaanottavalle oliolle, tai *operaatiokutsu* (call), jonka vastaanotettuaan olio käynnistää nimetyn operaation (metodin) toteuttamisen,
- voi kantaa mukaan parametriarvoja (esim. *nappainPainettu(n)*, *hiiriPainikePainettu(painike,sijainti)*)



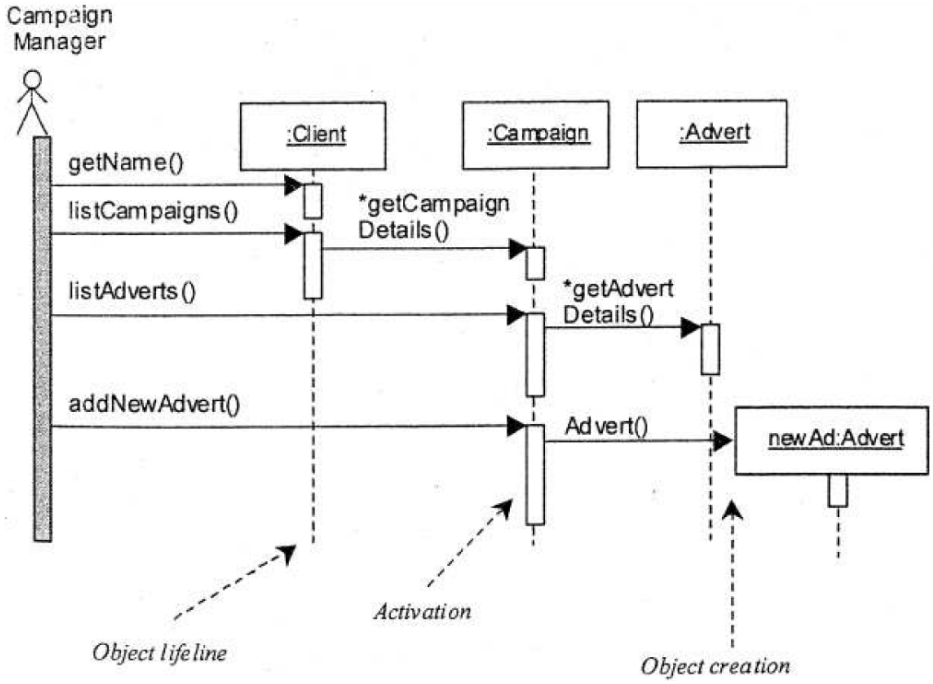
Kuva 4.21: Sekvenssikaavio

Sekvenssikaavio (sequence diagram, OMT:ssä *event trace diagram*) (kuva 4.21; Rumbaugh, 1999 s. 28; kuva 4.22; Bennett ym, 1999; kuva 4.23; Bennett ym, 1999):

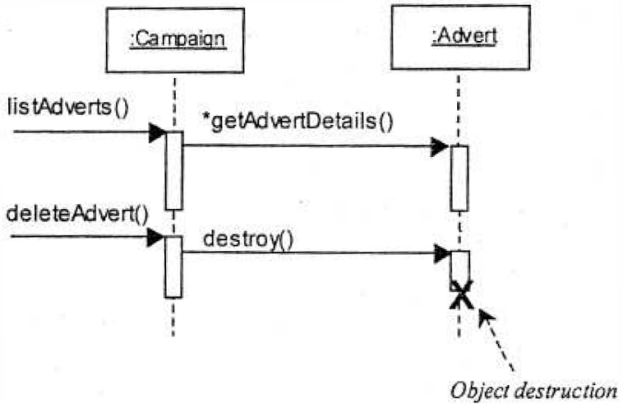
- kuvaa tavallisesti yhden käyttötapauksen mukaisen vuorovaikutuksen olioiden välillä (aika kulkee ylhäältä alas),
- esitys:
  - ◊ vuorovaikutukseen osallistuvat oliot kuvataan nimetyillä suorakulmioilla (vasemmalta oikealle suunnilleen siinä järjestyksessä kuin oliot lähettävät toisilleen viestejä),
  - ◊ olion elinaika kuvataan pystysuunnassa olevalla katkoviivalla (lifeline);
  - ◊ olion aktiivisen toiminnan aika kuvataan paksunnoksella (palkilla). Tänä aikana kontrolli on ko. operaation suorittamisen osalta oliolla.
  - ◊ Viesti kuvataan nuolena lähettävästä oliosta vastaanottavaan olioon sekä nuolen yhteydessä esitetyllä tekstillä. Nimi muodostuu yleensä operaation kutsumuodosta (signature).
  - ◊ Viestin käynnistämä operaatio voi tuottaa palautearvon, mutta vastaavaa palauteviestiä ei useinkaan kuvata, ellei sitä käytetä esim. myöhempien kutsujen parametrina.
  - ◊ Viestin yhteydessä voidaan esittää myös ehto, jonka voimassa ollessa viesti lähetetään
  - ◊ Olion luonti voidaan esittää suuntaamalla viestinuoli (create) suoraan ko. oliota vastaavaan suorakaiteeseen (konstruktori, muodostin),
  - ◊ Olion itsensä tuhoaminen osoitetaan palkin loppupään yli vedetyllä ristillä ja palauteviestillä; olion tuhoaminen voi olla myös toisen olion lähettämän viestin (destroy) seurauksena (destruktori, hajotin – Javan tapauksessa kuvaa tilannetta, kun olioon ei ole enää viitteitä, jolloin se voidaan poistaa muistinsiivouksen yhteydessä)
  - ◊ UML 1.4:ssä toisteinen tai ehdollinen toiminta kuvataan yhteistoimintakaavion tapaan *- ja *[]*-symboleilla (kannattaa käyttää myös UML 2.0-kaavioissa, jos yksinkertaistaa kaavion rakennetta)

◊ UML 2.0:ssa sekvenssikaavioihin on lisätty huomattavasti uusia mallinnuselementtejä, joiden ansiosta sekvenssikaavioilla voidaan kuvata kontrollirakenteita ohjelmointikielen tarkkuustasolla. Uusia elementtejä ovat mm. seuraavat:

- *Kehykset* (frames), joilla sekvenssikaavio voidaan nimetä ja parametrisoida, jolloin siihen voidaan viitata muista kaavioista
- *Yhdistetyt fragmentit* (combined fragments, ”lohkot, blokit”), joilla sekvenssikaavio voidaan jakaa rakenteisiin osiin ohjelmointikielten lohkorakenteiden tapaan. Fragmentin merkitys riippuu operaattorista, joita ovat mm. *ref* (viittaus toiseen sekvenssikaavioon), *opt* (ehdollinen lohko, ”if-lause”), *alt* (useita vaihtoehtoisia lohkoja, ”switch/case”) ja *loop* (silmukka ”for/while/repeat-until”). Operaattorista riippuen fragmenttiin voidaan liittää myös parametreja (ks. kuva 4.29 - silmukan raja-arvot) ja *[]*-ehtoja (ehdolliset lohkot).



Kuva 4.22: Sekvenssikaavio tapaukselle ”lisää uusi mainos kampanjalle”



Kuva 4.23: Sekvenssikaavio käyttötapaukselle ”poista mainos kampanjasta”

4.3.2 Dynaamisen mallintamisen askeleet

Dynaamisella mallintamisella pyritään kuvaamaan järjestelmän käyttäytyminen. Seuraavassa kuvataan mallintamista OMT:n, Bennettin ym. ja ICONIX-menetelmien mukaisesti. Lähtökohtana toimivat

- tavoitemäärittelyt
- käyttötapaukset
- luokkakaaviot, ja
- aktiviteettikaaviot (esim. liiketoimintaprosessimallit)

Dynaamisen mallintamisen tuloksena myös kohdealuemalli tarkentuu, joten menetelmästä riippumatta dynaamisen mallintamisen jälkeen on tarkastettava eri mallien johdonmukaisuus (katso luku 4.5.3).

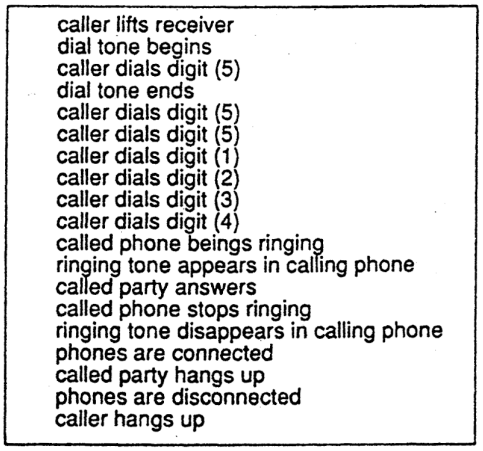
4.3.2.1 Dynaaminen mallintaminen: OMT

Mallintaminen etenee seuraavin askelin:

1. skenaarioiden tekeminen
2. vuorovaikutuksen mallintaminen
3. tilakaavioiden laadinta (suunnitteluvaiheessa, katso luku 5.3.2.1)

Skenaarioiden tekeminen:

Järjestelmän käyttäytymistä voidaan selvittää ottamalla lähtökohdaksi käyttötapaukset ja kuvaamalla niiden perusteella esimerkkitapauksia skenaarioina (scenario). *Skenaario* on luonnollisella kielellä esitetty kertomus toimintosarjasta, joka kattaa aktorien ja järjestelmän välisen vuorovaikutuksen kuten myös järjestelmän sisäisen toiminnan. Skenaario tarjoaa erinomaisen keinon keskustella ja suunnitella käyttäjien kanssa. Kuvassa 4.24 on esitetty pelkistetty esitys skenaariosta. Skenaarioita voidaan seuraavaksi määrämuotoistaa ja samalla yleistää kaavioiksi.



Kuva 4.24: Skenaario puhelinsoitolle.

Tarkoituksena on kuvata ensin yleisesti ja sitten yksityiskohtaisemmin dialogia käyttäjän ja järjestelmän välillä sekä järjestelmän sisällä. Lähtökohtana ovat käyttötapaukset ja luokkakaavio. Tehtävä edellyttää tiivistä yhteistyötä käyttäjien ja kehittäjien välillä:

- laadi sanalliset kuvaukset tyypillisistä viesteistä käyttäjän ja järjestelmän välillä (mainline scenario) ja tarpeen mukaan järjestelmän sisäisten olioiden välillä
- laadi sanalliset kuvaukset poikkeuksellisista (harvinaisista, virhetapauksista) tapauksista ja viesteistä (special scenario)
- tuo esiin viestien olennaisimmat parametrit

Vuorovaikutuksen mallintaminen:

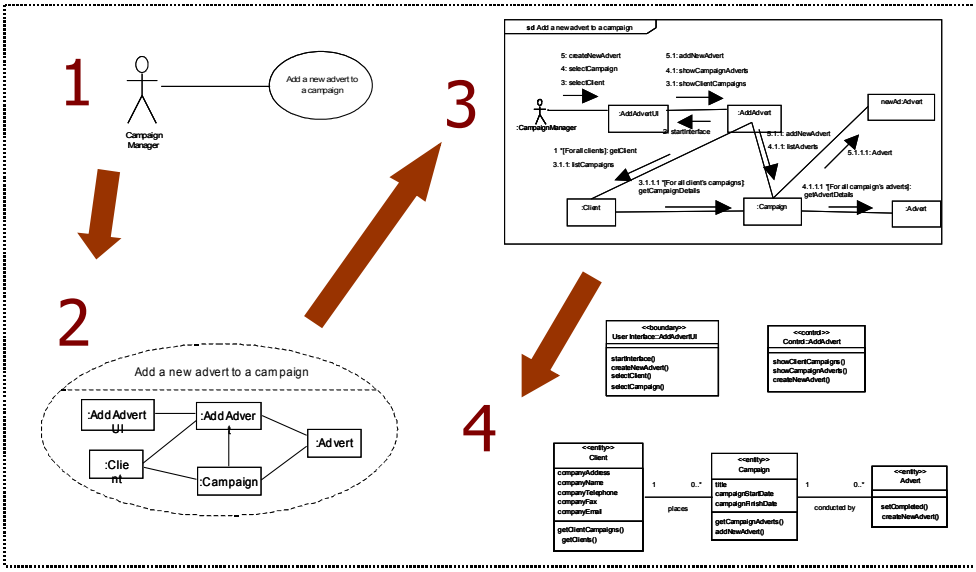
Tarkoituksena on tunnistaa viestit ja herätteet sekä tarkentaa niiden avulla järjestelmän sisäistä käyttäytymistä:

- Ota lähtökohdaksesi käyttötapausmallit, CRC-kortit ja mahdolliset skenaariot. Kunkin käyttötavan osalta selvitä, minkä olioiden osallistumista kuvatun toiminnan aikaansaaminen edellyttää. Piirrä ne suorakaiteiksi sekvenssikaavioon/yhteistoimintakaaavioon.
- Selvitä, millaisia viestejä liittyy olioiden sisäiseen vuorovaikutukseen sekä järjestelmän ja aktoreiden väliseen kanssakäymiseen. Esitä ne ajallisesti ja loogisesti oikeassa järjestyksessä sekvenssikaaviossa (jos aikajärjestys on tärkeä) tai yhteistoimintakaavioissa (jos rakenteelliset suhteet ovat tärkeitä).
- Käytä normaalin vuorovaikutuksen esittämiseen yksiä kaavioita ja poikkeustilanteita varten luodun toiminnan esittämiseen toisia kaavioita, jos kaikki eivät sovi yhteen kaavioon,
- Analyysivaiheen aikana käyttöliittymä- ja ohjausluokkia ei tarvitse mallintaa tarkasti eikä viestien yhteydessä ole tarpeellista ilmaista parametreja. Ne tarkennetaan suunnitteluvaiheessa.

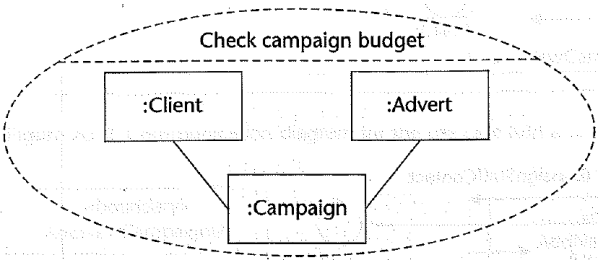
4.3.2.2 Dynaaminen mallintaminen: Bennett ym.

Bennett ym. (2006) suosittelevat sekä staattiseen että dynaamiseen mallintamiseen prosessia (kuva 4.25), jossa lähtökohtana ovat vaatimuksista johdetut käyttötapaukset (1), joista hahmotellaan keskeisimmät luokat ja niiden assosiaatiot alustavaksi yhteistoimintakaavioksi (*collaboration*) (2). Tämän jälkeen luokkien suhteita ja tarvittavia metodeja tarkennetaan käyttötapauksen askeleita seuraamalla yhteistoimintakaavioon (3), ja vasta tämän jälkeen määritellään kohdealumalli (4) käyttötapaus kerrallaan. Lopuksi käyttötapauskohtaiset luokkakaaviot yhdistetään yhdeksi kohdealuemalliksi ja yhteistoimintakaaviot tarkennetaan sekvenssikaavioiksi.

Yhteistoimintakaavio mallinnetaan aluksi karkealla *collaboration*-tasolla (kuva 4.26; Bennett 2000). Kaavion avulla etsitään käyttötapauksesta potentiaalisia luokkia ja niiden välisiä suhteita. Metodien nimiä tai suoritusjärjestystä ei merkitä.

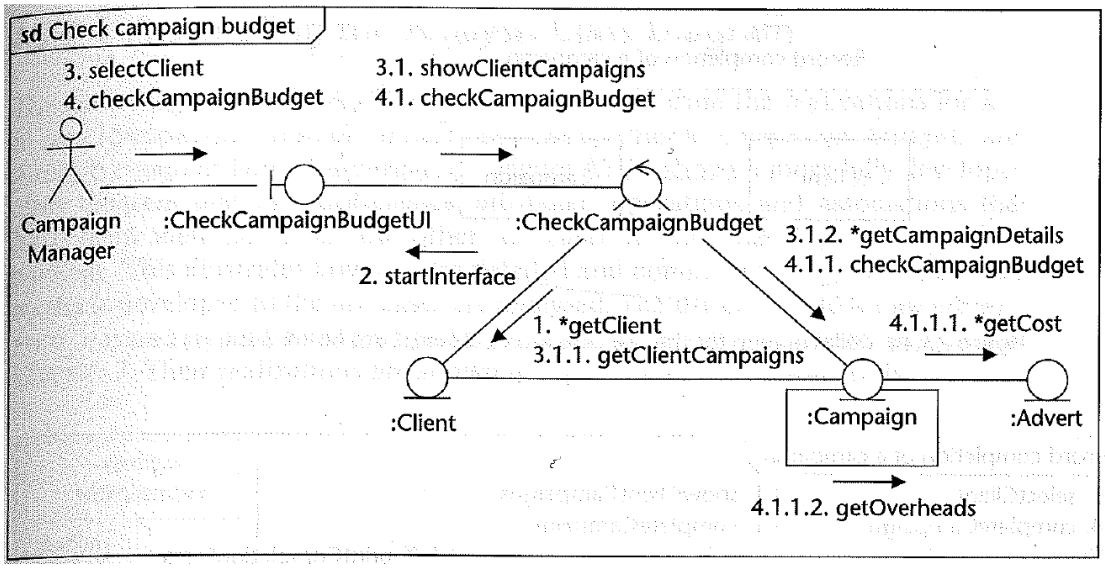


Kuva 4.25: Staattisen ja dynaamisen mallintamisen askeleet analyysivaiheen alkupuolella Bennetin ym. (2006) mukaan.

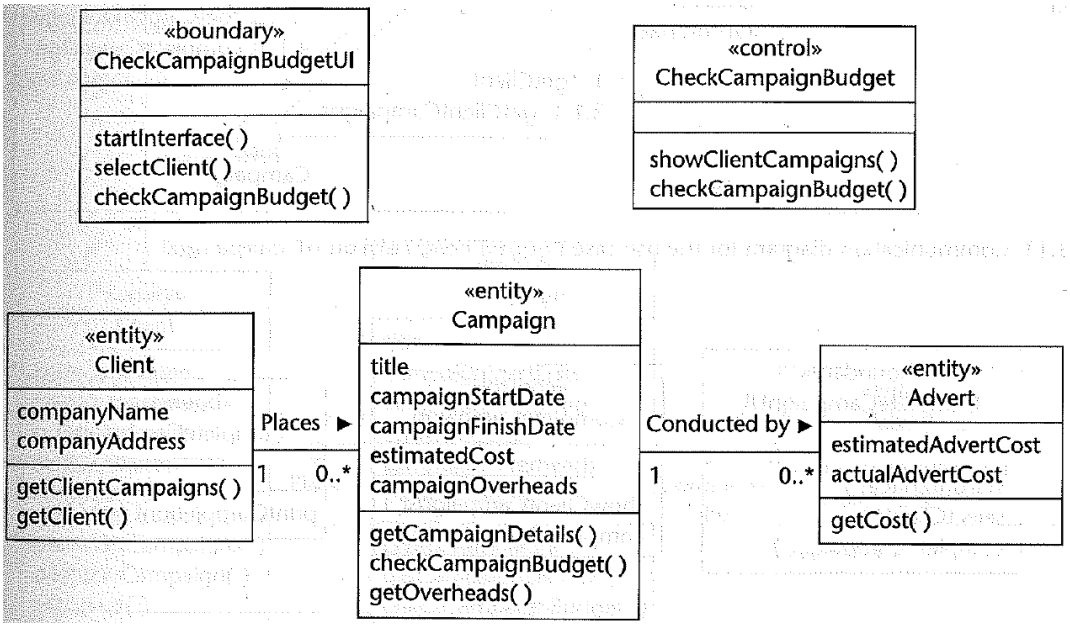


Kuva 4.26: Alustava yhteistoimintakaavio (collaboration) käyttötapaukselle ”tarkasta kampanjabudjetti”.

Yhteistoimintakaavio tarkennetaan käyttötapaksen kulkua seuraamalla aluksi yhteistoimintakaavioksi (kuva 4.27 – katso myös kuvan 3.5 käyttötapaus), jonka avulla päätellään edelleen kohdealuemallin yhteistoimintakaavion liittyvä osa (kuva 4.28).

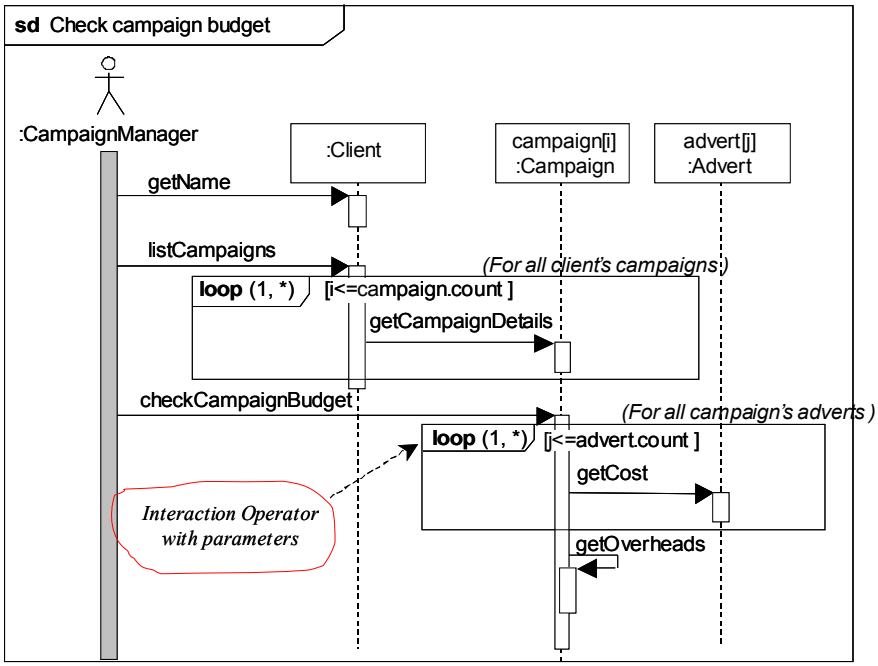


Kuva 4.27: UML-yhteistoimintakaavio (communication diagram) käyttötapaukselle ”tarkasta kampanjabudjetti”.



Kuva 4.28: Käyttötapaukseen ”tarkasta kampanjabudjetti” liittyvä kohdealuemallin osa – käyttöliittymä – ja ohjainluokka merkitty mukaan malliin.

Yhteistoimintakaavio tarkennetaan lopuksi sekvenssikaavioksi, jossa UML 2 -notaatiota käyttäen voidaan kuvata tarkasti silmukat, ehtolauseet ja mahdolliset viittaukset muihin sekvenssikaavioihin. Bennett ym. suosittelevat sekvenssikaavion laadintaa sekä analyysi- että suunnitteluvaiheessa.

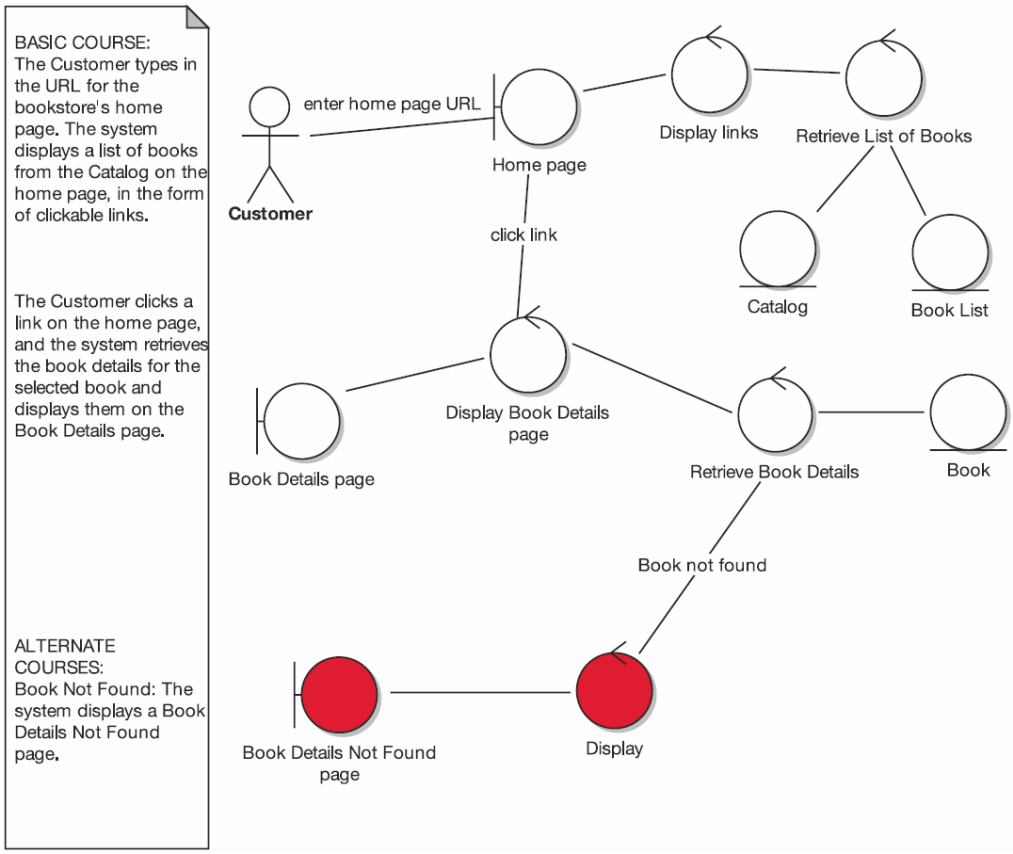


Kuva 4.29: Analyysivaiheen UML 2.0-sekvenssikaavio käyttötapaukselle ”tarkasta kampanjabudjetti” (käyttöliittymä- ja ohjausluokkia ei merkitty)

4.3.2.3 Dynaaminen mallintaminen: ICONIX

Dynaamisen mallintamisen kaaviot on ICONIX-menetelmässä jaettu niin, että yhteistoimintakaavio on analyysivaiheen artefakti ja sekvenssikaavio suunnitteluvaiheen. Tämä minimoi mallinnettavien kaavioiden määrän ja selkeyttää eri kaaviotyyppien käyttötarkoitusta. Muissa menetelmissä kaavioiden käytössä on enemmän liikkumavaraa.

ICONIX-menetelmässä käytetään yhteistoimintakaavioista varianttia *robustness diagram*, jota tässä kutsutaan myös korkean tason yhteistoimintakaavioksi. *Collaboration*-kaavion tapaan yhteistoimintakaaviossa käytetään yleensä ikonisia stereotyyppejä luokkien esittämiseen. Viestit luokkien välillä voivat olla nimettyjä, ja lisäksi kaavioon voi olla merkitty ehtoja tai silmukoita UML-yhteistoimintakaavioiden tapaan. Yhteistoimintakaavio muodostetaan käymällä kaaviota vastaava käyttötapauskuvaus lause kerrallaan läpi ja muodostamalla substantiiveista käyttöliittymä- ja kohdealueluokkia ja verbeistä ohjaimia. Lisäksi luokkien välillä olevia viestejä voi tarvittaessa tarkentaa metodien mahdollisilla nimillä tai merkitsemällä nuolella viestin pääasiallisella suunnan. Myös käyttötapauksen poikkeustilanteet tulee merkitä kaavioon tarvittavien ehtojen kera. Kuvassa 4.30 on esimerkki ICONIX-tyylisestä yhteistoimintakaaviosta.



Kuva 4.30: Yhteistoimintakaavion johtaminen käyttötapauksesta ”Näytä kirjan tiedot” ICONIX-menetelmässä

Kaavion erikoisuutena ovat piirtosäännöt, joiden avulla käyttöliittymä-, ja kohdealueluokat tulevat varmasti erotetuiksi toisistaan. Lisäksi säännöt tukevat käyttötapausten

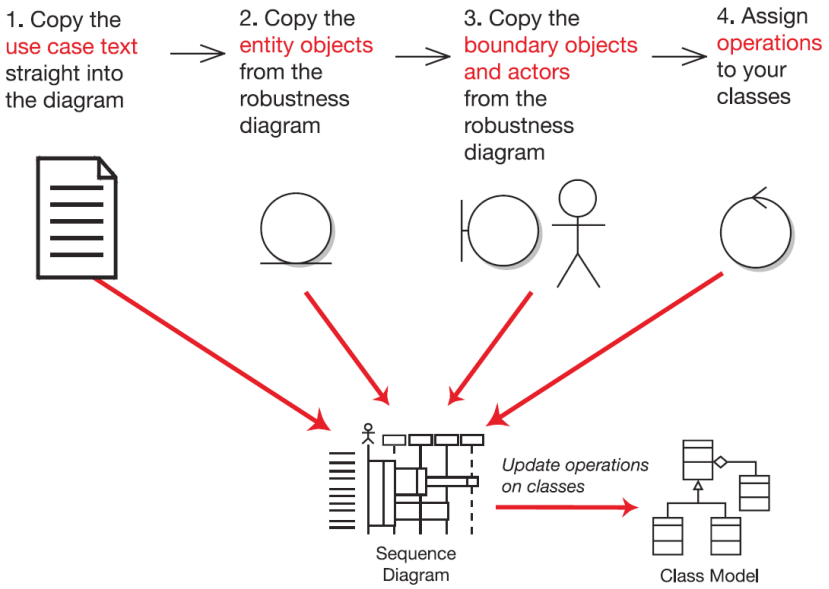
hyvää kirjoitustyyliä (subjekti-predikaatti-objekti -lausejärjestys). Kytkentäsäännöt ovat seuraavat:

- Käyttöliittymäluokat ja kohdealueluokat voivat kutsua ohjaimia ja päinvastoin *(nouns can talk to verbs and vice versa)*
- Käyttöliittymäluokat ja kohdealueluokat eivät voi kutsua suoraan toisiaan *(nouns can't talk to other nouns)*
- Ohjaimet voivat kutsua toisiaan *(verbs can talk to other verbs)*

Käyttöliittymä- ja kohdealueluokkien suoran keskinäisen kommunikoinnin kieltäminen johtaa ohjainten määrän kasvamiseen, mutta toisaalta helpottaa kaavion asettelua ja viestien reitittämistä (tavanomaisessa UML-yhteistoimintakaaviossa suurin osa viesteistä saattaa kulkea samojen kaarien välillä - erityisesti, jos käytössä on vain yksittäiset käyttöliittymä- ja ohjausluokat käyttötapausta kohti). ICONIX-yhteistoimintakaavion ohjaimia ei pidäkään ajatella luokkina, vaan potentiaalisina metodeina, jotka sijoitellaan luokkien vastuualueiden määrityksestä riippuen kohdealue- tai käyttöliittymäluokkiin (ja osa jätetään tai yhdistetään erillisiksi ohjainluokiksi) suunnitteluvaiheessa, sekvenssikaavioita mallinnettaessa. Vasta tässä vaiheessa on mielekästä merkitä ohjainluokkia tarkasti luokkakaavioon.

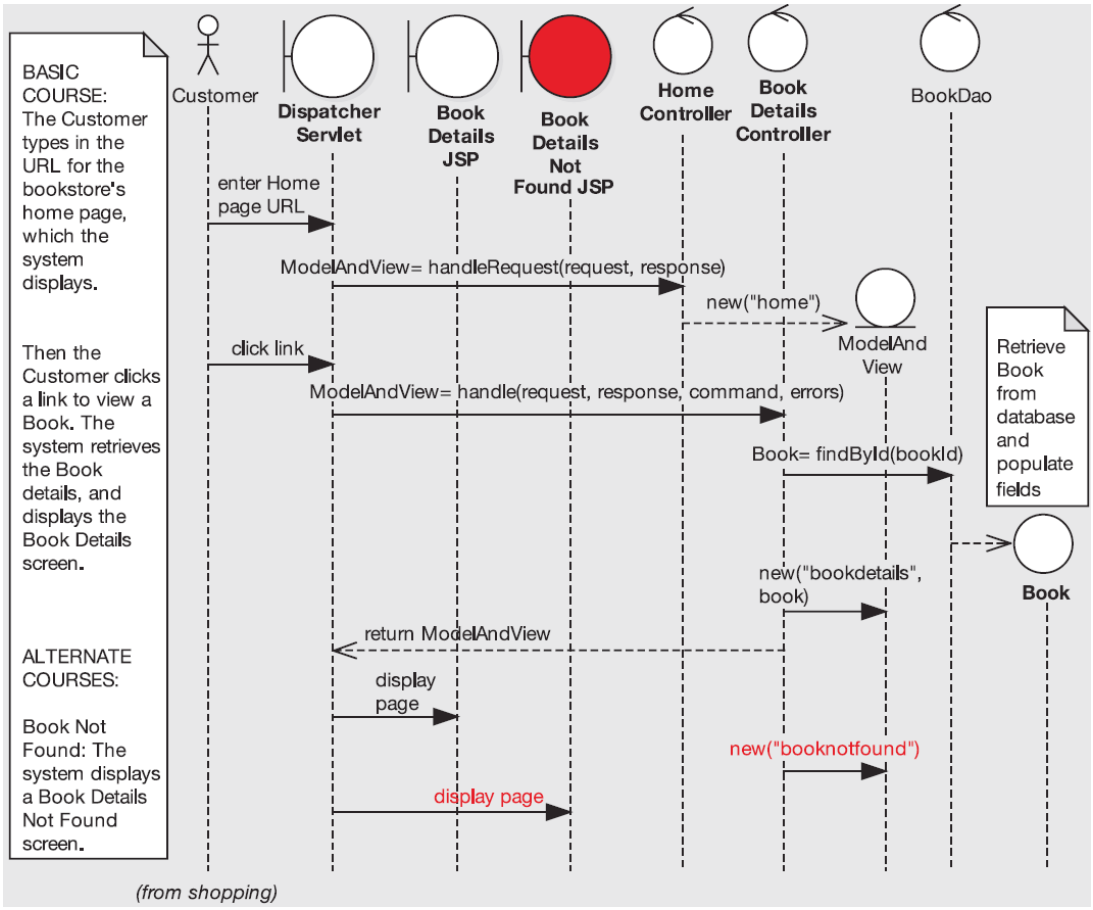
Sekvenssikaavioiden mallintaminen etenee ICONIX-menetelmässä seuraavasti (kuva 4.31; Rosenberg & Stephens 2007): lähtökohtana ovat yhteistoimintakaavion tapaan käyttötapauksen askeleet sekä yhteistoimintakaavioon jo mallinnetut käyttöliittymä- ja kohdealueluokat. Erona yhteistoimintakaavioon sekvenssikaaviossa tarkennetaan olioiden kommunikoinnin määrityksiä ja otetaan kantaa järjestelmän sisäiseen toimintaan ja mahdollisiin toteutustekniikoihin. Keskeisenä tekniikkana on yhteistoimintakaavion ohjainten sijoittelusta päättäminen luokkien eri vastuualueiden (voidaan mallintaa esim. CRC-korteilla) mukaan. Ohjaimien sijoittelussa on kolme eri perustekniikkaa, joista kaikkia tarvitaan laajemman käyttötapauksen tarkentamisessa:

1. Säilytä kontrolliluokka sellaisenaan
2. Yhdistä useita samaan asiaan liittyviä kontrolliluokkia yhteen
3. Muuta ohjain käyttöliittymä- tai kohdealueolion metodiksi



Kuva 4.31: Sekvenssikaavion mallinnusprosessi ICONIX-menetelmässä

Kuvassa 4.32 (Rosenberg & Stephens 2007) on esitetty kuvan 4.30 yhteistoimintakaavio sekvenssikaavioksi tarkennettuna.



Kuva 4.32: ”Näytä kirjan tiedot”-käyttötapauksen sekvenssikaavio.

4.3.3 CRC-kortit

CRC (Class-Responsibility-Collaboration) -kortit ovat Beckin & Cunninghamin alun perin oliolähestymistavan opettamiseen kehittämä tekniikka[19], joka on sittemmin osoittautunut myös yleisemmin toimivaksi oliosuunnittelun apuvälineeksi, erityisesti vastuualuepohjaisen suunnittelun (Responsibility-Driven Design; Wirfs-Brock 1990) yhteydessä.

CRC-kortit (kuva 4.33; Bennett ym. 2006) koostuvat luokan nimestä, vastuukuvauksista ja avustajista (yhteistoiminta). Korteissa voidaan käsitellä kerralla kohdealuemallin keskeisimpiä luokkia kokonaisuutena tai sitten käsitellä käyttötapausta kerrallaan, jolloin myös käyttöliittymä- ja ohjausluokat ovat mukana.

[19] K. Beck & W. Cunningham (1989). A Laboratory For Teaching Object-Oriented Thinking. http://c2.com/doc/oopsla89/paper.html

Class Name	*Client*
Responsibilities	Collaborations
Provide client information. *Provide list of campaigns.*	*Campaign provides campaign details.*

Class Name	*Campaign*
Responsibilities	Collaborations
Provide campaign information. *Provide list of adverts.* *Add a new advert.*	*Advert provides advert details.* *Advert constructs new object.*

Class Name	*Advert*
Responsibilities	Collaborations
Provide advert details. *Construct adverts.*	

Kuva 4.33: Mainostoimistoesimerkin CRC-kortteja

Vastuu (responsibility) on korkean tason kuvaus toiminnosta, jota luokka voi tehdä (tai pyytää muilta luokilta) tai palvelu, jote se voi tarjota muille luokille. Vastuuta voi verrata sopimukseen tai rajapintamääritykseen. Luokka voi suoriutua vastuusta jollakin seuraavista periaatteista (Wirfs-Brock 1990):

1. Tee kaikki työ itse
2. Pyydä apua muilta olioilta työn osien suoritukseen yhteistyössä
3. Delegoi vastuu jollekin toiselle oliolle

Eri luokkien vastuun pitäisi jakautua järkevästi (ei välttämättä kuitenkaan tasaisesti) niin, ettei yksittäisellä luokalla ole vastuuta liikaa tai liian vähän. Vastuualueiden jakamisesta päättäminen on yleisessä muodossaan epätriviaali tehtävä. Esimerkiksi tilanteessa, jossa tietoja on koottava kahdesta luokasta tietyn tehtävän suorittamista varten voidaan edetä useilla mahdollisilla tavoilla – sopivin tapa riippuu järjestelmän arkkitehtuurista, laajuudesta ja muista vaatimuksista esim. yleiskäyttöisyyden suhteen:

1. Ensimmäinen olio kysyy tietoja toiselta oliolta ja hoitaa koostamisen ja tarvittavat toiminnot (yksinkertaisin mutta ei välttämättä ylläpidettävin ratkaisu)
2. Ensimmäinen olio luovuttaa itsensä parametrina toiselle oliolle, joka suorittaa toiminnon käyttäen omia ja ensimmäisen olion tietoja (vaatii sopivan kyselyrajapinnan määritystä)
3. Käytetään kolmatta oliota tietojen kyselyyn tai toimintoon tarvittavien tietojen välittämiseen (erotellaan sovelluslogiikkaan liittyvät toiminnot kohdealueluokista)
4. Abstrahoidaan olioilta vaaditut tiedot, koostamisen määritys ja suoritettava toiminto omiksi olioikseen, jolloin erilaisia hakuja ja toimintoja voidaan yhdistellä ja

muuttaa helposti (tällöin kyse voi olla enemmän yleisen sovelluskehyksen kuin tiettyä yksittäistä ongelmaa ratkaisemaan suunnitellun järjestelmän määrityksestä)

Yhteistoiminta (collaboration) tarjoittaa CRC-korttien yhteydessä kuvausta muista luokista (avustajista), joita käsiteltävänä oleva luokka tarvitsee suoriutuakseen vastuualueistaan. Kohdealuemallin assosiaatiot voivat olla lähtökohtana avustajien etsinnässä, mutta yhteistoimintaa määritellessä kannattaa miettiä, kumpi assosiaatiossa mukana oleva luokka käyttää pääsääntöisesti toisen luokan palveluja (eli toimii korttiin merkityn luokan avustajana) - joskus assosiaatio voi olla aidosti 2-suuntainen, useimmiten ei. Assosiaatioiden suunnat tarkennetaan lopulliseen muotoonsa suunnitteluvaiheessa. Varsinaisten assosiaatioiden lisäksi luokka voi saada viitteen avustajaan toisen luokan metodin paluuarvona, oman metodinsa parametrina jne.

Korttien täyttämiseen suositellaan seuraavaa prosessia:

1. Tunnista luokat
2. Jaa kutakin luokkaa edustavat oliot (kortit) ryhmän jäsenille
3. Käy läpi käyttötapaus askel kerrallaan ja täytä vastuualueet ja avustajat. Läpikäynnin aikana ryhmän pitäisi päästä yhteisymmärrykseen tai neuvotella vastuualueista ja niiden hoitamisesta (esim. yhtenä strategiana osallistujat voivat pyrkiä mahdollisimman vähäiseen määrään vastuita)
4. Tunnista puuttuvat tai päällekkäiset oliot ja vastuut

Kurssilla käytettävässä ”menetelmässä” oletetaan, että CRC-kortteja käytettäessä luokat ja pääosin myös luokkien väliset assosiaatiot ovat selvillä, mutta luokkien operaatioiden lopullista jakoa tai toteutustapaa ei ole vielä päätetty. CRC-kortit tarjoavat hyödyllistä tietoa siirryttäessä analyysivaiheesta suunnitteluvaiheeseen, erityisesti sekvenssikaavioita mallinnettaessa. Operaatioiden toteutusta voi luokkien yhteistoiminnan osalta hahmotella karkealla tasolla jo yhteistoimintakaavioissa, mutta CRC-korteilla voidaan perustella tarkemmin, mitä apuluokkia (tai esim. parametrien välitystä, jos luokkien välille ei haluta määrittää suoraa assosiaatiota) toteutus vaatii. Lisäksi CRC-korttien avulla voidaan löytää aiemmissa vaiheissa ohitettuja tarpeellisia operaatioita ja mahdollisesti myös assosiaatioita luokkakaavioon.

4.3.4 Yhteenveto dynaamisen mallintamisen kaavioiden välisistä suhteista

Sekvenssikaavioissa painotetaan prosessin hallinnan omistamista ja prosessissa esiintyvien metodien välittämistä. Sekvenssikaavioista nähdään, milloin suoritusvastuu on milläkin oliolla. Aktiviteettikaavioista selviää samankaltainen tieto käyttäen eri toimijoiden mukaista prosessin kaistaloimista. Sekvenssikaavioissa esitetään lisäksi eri olioiden elinkaaret ja täten pystytään huomioimaan se, milloin olioiden tulisi syntyä ja milloin tuhoutua ja minkä olion vastuulla toiminnot milloinkin ovat. Sekvenssikaavioista löydetään myös metodit, joilla olioiden välinen kommunikointi tapahtuu. Aktiviteettikaavioista puuttuvat nämä konkreettisen tason ohjelmointia helpottavat tiedot.

Yhteistoimintakaavioiden ja sekvenssikaavioiden välinen ero on siinä, että sekvenssikaaviot keskittyvät erityisesti prosessiin osallistuvien olioiden välisten metodien kutsujen ajalliseen hallintaan, kun taas yhteistoimintakaaviot painottavat rakenteellisia ominaisuuksia. Aktiviteettikaavioiden ja yhteistoimintakaavioiden suurin ero puoles-

taan on se, että yhteistoimintakaaviosta näkyvät myös tarvittavat metodien kutsut, joita aktiviteettikaaviossa ei yleensä ole. Mallintamisen kannalta toimintakaaviot ovat erittäin korkean tason työväline, jonka avulla huomio saadaan kohdennettua erityisesti mallinnettavan prosessin loogiseen rakenteeseen. Muut dynaamisen mallintamisen työvälineet esittävät paljon konkreettisemmin ne viestit ja toiminnot, joita varsinaisessa ohjelmointivaiheessa tarvitaan.

Lyhyt toimintasuunnitelma hyvän dynaamisen mallin luomiseen on se, että suunnittelija lähestyy ensin ongelmaa yleisellä tasolla toimintakaavioiden avulla, minkä jälkeen muodostetaan yhteistoimintakaaviot käyttötapausten pohjalta. Suunnitteluvaiheessa yhteistoimintakaaviot tarkennetaan edelleen sekvenssikaavioiksi, tarvittaessa CRC-kortteja apuna käyttäen. Tilakaaviot muodostetaan vasta yksityiskohtaisen suunnittelun aikana hyödyntäen edellä luotuja kaavioita. Mikäli esimerkiksi ohjelmistoa testattaessa havaitaan ongelmia jossain loogisen tason ratkaisussa, yhteistoimintakaavioista on helppo tarkistaa mitä rakenteellisia muutoksia voitaisiin toteuttaa ongelman ratkaisemiseksi. Suoraan tilakaavioista katsomalla loogisen tason ongelmat voivat jäädä huomaamatta yksityiskohtaisten tietojen runsauden vuoksi.

4.4 Käyttöliittymän mallintaminen

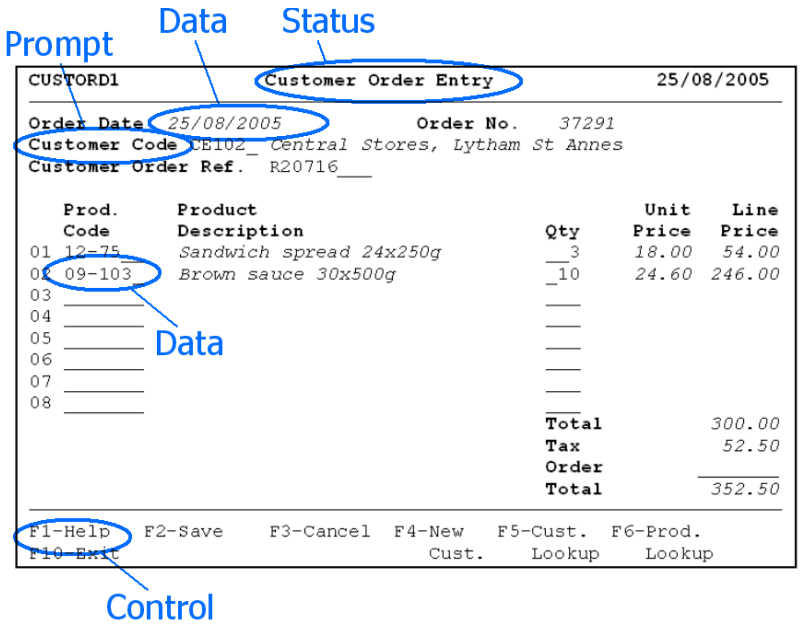
Käyttöliittymä (UI, user interface) on rajapinta, jonka välityksellä järjestelmä on yhteydessä ympäristöönsä. Oletetaan seuraavassa yksinkertaisuuden vuoksi, että nämä yhteydet ovat käyttäjään (toisin sanoen suljetaan tarkastelun ulkopuolelle yhteydet laitteisiin ja muihin järjestelmiin). Käyttöliittymän suunnittelussa voidaan soveltaa kahta metaforaa:

1. *Dialogi-metafora*: tämän mukaan käyttäjä ja järjestelmä käyvät vuoropuhelua dialogin muodossa käyttöliittymän kautta. Dialogiin kuuluu viestiminen molempiin suuntiin. Kuvassa 4.34 (Bennett ym, 2006) on esitetty tyypilliset viestimuodot järjestelmään päin (in) ja järjestelmästä poispäin (output). Konkreettisena käyttöliittymänä voi olla esimerkiksi kuvan 4.35 (Bennett ym. 2006) mukainen merkkipohjainen näyttö asiakastilausten tallentamista varten.
2. *Suorakäsittelyn* (direect manipulation) *-metafora*: tämän mukaan käyttöliittymä on areena, jolla käyttäjä voi toimia mahdollisimman samantapaisesti kuin reaalimaailmassa (esim. tiedon hävittämiseksi sen laittaminen "roskakoriin"). Toisin sanoen: "*What you see is what you get*" (WYSIWYG). Tätä voidaan soveltaa graafisten käyttöliittymien yhteydessä. Käyttöliittymän suunnittelu käynnistyy tällöin käyttäjän tehtävän semantiikan selvittämisestä ja sitä vastaavan mallin luomisesta käyttöliittymäkomponenteista ja niiden käsittelystä (vedä_ja_pudota, pienennä_ ja_ kutista_ikkunaa, paina_painiketta, vedä_alas_ valikko). Kuvassa 4.36 (Bennett ym, 2006) on esitetty tyypillinen graafinen käyttöliittymä, joka toteuttaa aiemmin käsitellyn käyttötapauksen ”Tarkasta kampanjabudjetti”.

Lähtökohtana käyttöliittymän mallintamiselle ovat pääasiassa käyttötapausten kuvaukset. Kun käyttötapauksessa viitataan käyttäjän toimintoon tai järjestelmän palautteeseen/tulosteeseen, on kysymys käyttöliittymän alaan kuuluvasta toiminnnasta. Tästä syystä on varsin luonnollista, että jo varsin aikaisessa vaiheessa järjestelmän kehittämistä pyritään saamaan aikaan hahmotelma käyttöliittymän rakenteesta ja toiminnasta.

Output	prompt	request for user input
	data	data from application following user request
	status	acknowledgment that something has happened
	error	processing cannot continue
	help	additional information to user
Input	control	user directs which way dialogue will proceed
	data	data supplied by user

Kuva 4.34: Tyypilliset viestityypit käyttöliittymässä

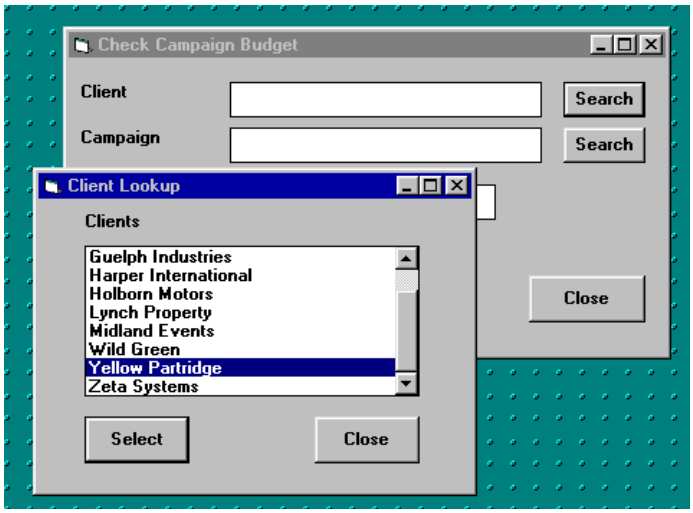


Kuva 4.35: Erään yhtiön asiakastilauksen syöttökäyttöliittymä

Koska RUP-menetelmässä käyttöliittymän suunnittelu on hyvin vähäisessä roolissa, on tämä askel koostettu lähinnä OMT++-menetelmän mukaan. Kuvaus on suppea eikä paneudu esimerkiksi käyttöliittymän käytettävyyden (usability) suunnitteluun.

Käyttöliittymä koostuu dialogeista, jotka edelleen koostuvat komponenteista (esim. paneelit, välilehdet), jotka edelleen voivat koostua toisista komponenteista (esim. painikkeet, tekstikentät, valintalistat). Seuraavassa on mainittu yleisimmät komponentit:

- *Ikoni* (icon) eli kuvake: symboloi joko tietoa, käsittelytoimintoa tai tietojärjestelmän osaa; voidaan luokitella muodon, tyypin ja värin mukaan,
- *Ikkuna* (window): tarkoittaa näytöllä olevaa suorakulmion muotoista ja kehyksin rajattu aluetta tietyn toiminnallisen ja tietonäkymän esittämiseksi,
- *Dialogilaatikko* (dialog box): sisältää huomautuksia, varoituksia, ehdotuksia tai muuta lisäinformaatiota, kuten myös vahvistus- ja peruutuspainikkeet,
- *Valikko* (menu): tarjoaa valittaviksi tietoja tai toimintoja; valikkomuotoiona: putkahdusvalikko, valikkopalkki, alasvetovalikko,
- *Kenttä* (field): käytetään tiedon syöttämiseen ja esittämiseen; voi olla rivikenttä tai taulukkokenttä; näihin voi liittyä vierityspalkkeja molempiin suuntiin.

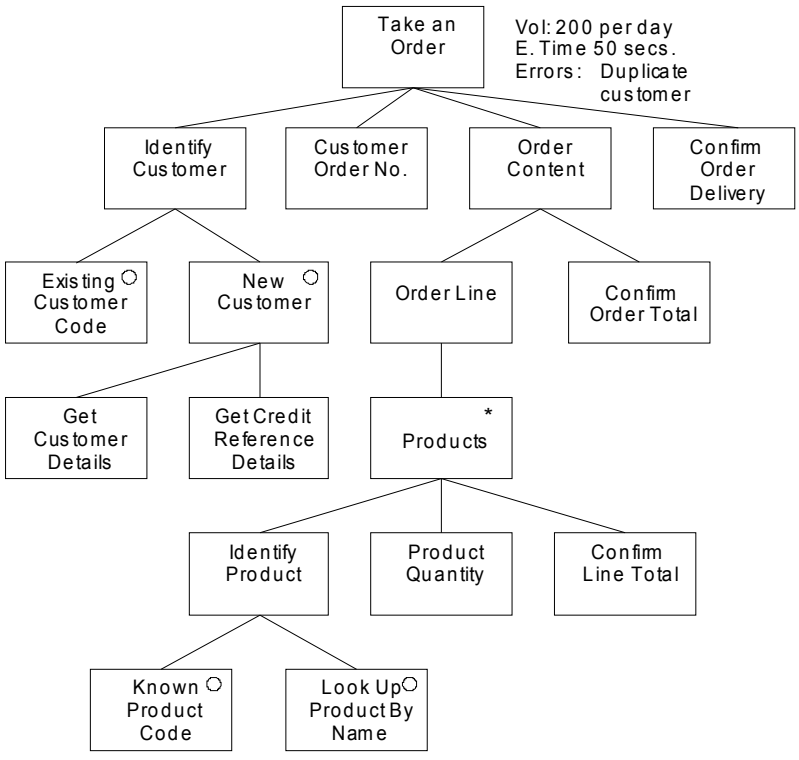


Kuva 4.36: Graafinen käyttöliittymä (Visual Basic -prototyyppi) käyttötapaukselle ”Tarkasta kampanjabudjetti”.

Suorakäsittely tapahtuu kohdistinlaitteen avulla. Laite voi olla esim. hiiri, valokynä, ohjaintikku, rullapallo tai sormi (kosketusnäyttöjen yhteydessä).

Käyttöliittymän suunnittelun lähestymistapoja on useita (Bennett 1999, s. 332):

- *Rakenteinen* lähestymistapa:
  - ◊ käytetään yleisesti rakenteisten kehittämismenetelmien yhteydessä,
  - ◊ käyttäjien tehtävät jäsennetään (top-down) ja esitetään hierarkioina (kuva 4.37; Bennett ym. 1999), joiden perusteella suunnitellaan dialogikaaviot,



Kuva 4.37: Tehtävähierarkiakuvaus tehtävälle ”vastaanota tilaus”

- *Etnograafinen* lähestymistapa:
  - ◊ painottaa käyttökontekstin huomioon ottamista käyttäjätehtävien ja käyttöliittymien suunnittelussa ja arvioinnissa,
  - ◊ käyttäjien osallistuminen suunnitteluun ja arviointiin tärkeää,
- *Skenaarioperusteinen* lähestymistapa:
  - ◊ perustuu skenaarioiden (eli konkreettisten tarinoiden) tekemiseen käyttäjien tehtävistä (kuva 4.38; Bennett 1999). Näiden avulla suunnitellaan ja testataan käyttöliittymän toimivuutta.

The user selects Add a Note from the menu. A new window appears
From the list box at the top of the window she selects the name of the client.
A list of campaigns appears in the list box below, and she selects a particular campaign.
A list of adverts appears in the next list box, and she selects a specific advert.
She types a few paragraphs into a text box to describe her idea for the advert.
She fills the space on screen and a vertical scrollbar appears and the text in the text box scrolls up.
She enters her initials into a text box, and the system checks that she is allocated to work on that campaign.
The date and time are displayed by the system, and the Save button is enabled.
She clicks on the Save button and the word Saved appears in the status bar.
The text box, the text field for initials and the date and time are cleared.

Kuva 4.38: Skenaario kuvaa, miten käyttäjä voisi lisätä lyhyen viestin

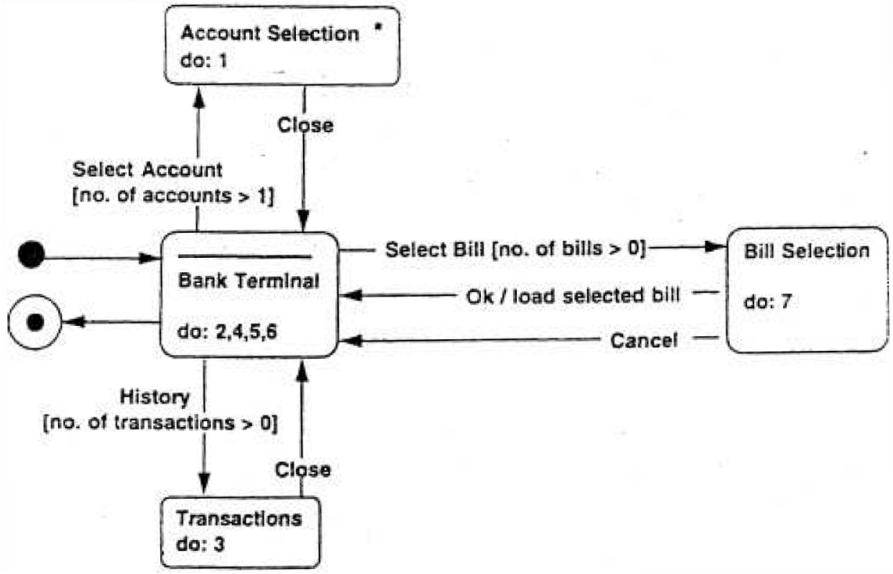
OMT++ -menetelmässä käyttöliittymän suunnittelu on muunnelma rakenteisesta ja skenaarioperusteisesta lähestymistavasta. Se etenee seuraavien askeleiden mukaisesti:

1. erotellaan käyttötapausten kuvauksista ne käyttäjien suorittamat tehtävät (esim. tilisaldon kysyminen), joissa ollaan yhteydessä järjestelmään (kuva 4.39). Kuvataan tehtävät tarkemmin. Jos tehtäviä on paljon, jaetaan ne kahteen ryhmään: ensisijaiset tehtävät (usein toistuvat, pitkäkestoiset, todennäköisimmin pääikkunaan sijoittuvat) ja toissijaiset tehtävät (harvoin esiintyvät, poikkeukselliset, tukitehtävän luonteiset),
2. Piirretään dialogikaavio (*dialogue diagram*, erikoistapaus tilakaaviosta), josta näkyvät järjestelmän dialogit (näytöt) ja niiden väliset siirtymät ja sijoitetaan tehtävät dialogikaavioon (kuva 4.40). Kaavion muodostusta ohjaavia kriteerejä ovat:
   - ◊ Loogisesti toisiinsa liittyvät tehtävät samaan näyttöön,
   - ◊ Siirtymät näyttöjen välillä mahdollisimman lyhyet ja luontevat (vrt. käyttötapausten kuvaukset)
   - ◊ Ensisijaiset tehtävät päänäyttöihin, toissijaiset eri näyttöihin,

Number	Task
1.	Select a user's account.
2.	Read the balance of the selected account.
3.	Read transcations related to the selected account.
4.	Modify information concerning the bill to be paid.
5.	Put the bill on the waiting list for payment on the due date.
6.	Remove the bill from the waiting list.
7.	Select a bill that is waiting to be paid on the due date.

Kuva 4.39: Pankkipääteohjelmiston suorittamat tehtävät

3. Määritellään dialogikomponentit (käytetään apuna komponenttikirjastoa),
4. Suoritetaan visualisointi, jolloin luodaan kullekin hahmotellulle komponentille käyttäjäystävällinen rakenne ja toimintalogiikka,
5. Tarkennetaan tehtävämäärityksiä kuvaamalla, miten tehtävät suoritetaan dialogikomponenttien avulla; karkealla tasolla apuna voidaan käyttää sekvenssikaavioita.

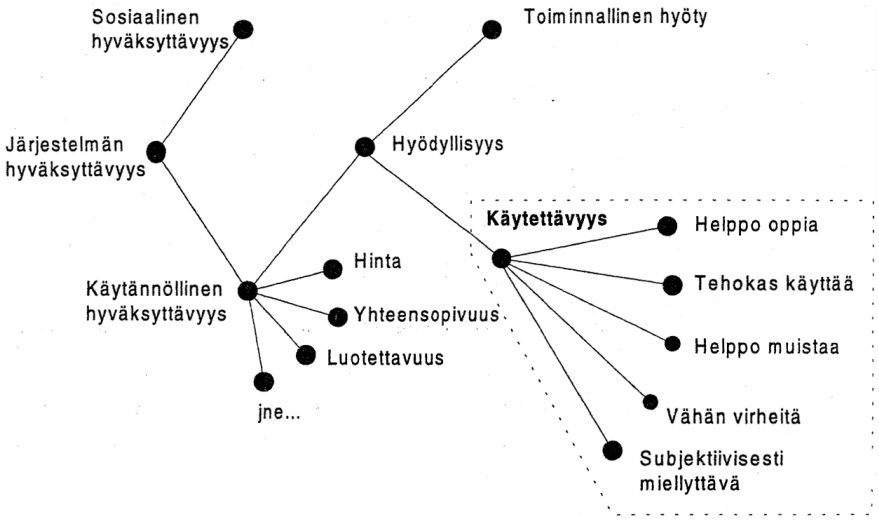


Kuva 4.40: Pankkipääteohjelmiston dialogikaavio

Analyysivaiheessa riittää tavallisesti kahden ensimmäisen askeleen suorittaminen. Jos tässä vaiheessa rakennetaan prototyyppejä, suoritetaan myös kolme viimeistä askelta.

Käytettävyydelle on esitetty monia määrityksiä. Nielsen (1993) jakaa käytettävyyden käsitteen kuvan 4.41 mukaisesti. ISO 9241 -standardin mukaan käytettävyys on suure, joka ilmaisee käyttäjäryhmän kyvyn pystyä käyttämään tuotetta (=tietojärjestelmää)

- tehokkasti (effectiveness),
- tuottavasti (efficiency) ja
- miellyttävästi (satisfaction)



Kuva 4.41: Nielsenin käytettävyysmalli

Käyttöliittymän suunnittelussa ja arvioinnissa voidaan käyttää erilaisia heuristisia menetelmiä, joita tukee laaja valikoima käytettävyyssääntöjä. Alla on esitetty näistä joitakin:

- Käytä yksinkertaisia ja luonnollisia dialogeja
- Käytä käyttäjien omaa kieltä
- Minimoi käyttäjän muistikuorma
- Tee käyttöliittymästä kauttaaltaan yhdenmukaisia
- Anna käyttäjille palautetta toiminnoista
- Anna selkeä poistumistapa eri tilanteista ja toiminnoista
- Anna käyttäjille mahdollisuus käyttää oikopolkuja
- Anna virhetilanteista selkeät virheilmoitukset
- Anna riittävä ja selkeä apu ja dokumentaatio.

Edelleen käytössä on useita tyylioppaita, jotka tähtäävät yhdenmukaisten käyttöliittymien suunnitteluun. Valitettavasti tässä ei ole mahdollista käsitellä käytettävyyttä tämän enempää.

4.5 Integrointi ja tarkistaminen

Tähän asti käsitellyt analyysin osat ovat tarkastelleet järjestelmää pääasiassa yhdestä näkökulmasta kerrallaan, joko rakenteellisesta tai käyttäytymisestä käsin. Tämän osan tarkoituksena on yhdistää tarkastelukulmat mallien keskinäisen johdonmukaisuuden varmistamiseksi sekä rikastaa ja täydentää kuvauksia toisista kaavioista saatujen ideoiden pohjalta. Vastaavat tarkistukset on tehtävä myös suunnittelun yhteydessä, kun malleja päivitetään.

4.5.1 Vaatimusten uudelleenläpikäynti

Analyysin ja suunnittelun aikana tuotetut mallit tarjoavat konkreettisen perustan keskustella käyttäjien kanssa heidän alun perin esittämiensä vaatimusten merkityksestä, tärkeydestä ja realistisuudesta sekä mahdollisista rajauksista ja painotuksista. On täysin luonnollista, että myös vaatimukset “elävät”. Tästä syystä muutokset vaatimuksiin (ja vaikutukset käyttötapauksiin) tulee kirjata, jotta seuraavat iteraatiot voidaan ottaa todella haluttuun suuntaan.

4.5.2 Operaatioiden lisääminen luokkakaavioon

Tässä käsitellään ensin sitä, mistä operaatioita löydetään, ja sitten, mihin luokkaan operaatiot tulisi sijoittaa. Koska analyysivaiheessa luokkien operaatiot eivät ole keskeisimpiä, annetaan tässä asiasta vain yleiskuva.

Operaatiot voivat olla

- olioiden attribuuttien arvoja ja linkkejä koskevia toimintoja
    - ◊ analyysivaiheen aikana oletetaan, että luokkakaaviossa näkyvät attribuutit ja assosiaatiot ovat hyväksikäytettävissä, joten näitä koskevia perusoperaatioita (esim. Javassa yleensä get/set-alkuiset asetus- ja saantimetodit) ei tässä vaiheessa eksplisiittisesti tarvitse määrittää,

- viesteihin, herätteisiin ja tiloihin liittyviä toimintoja
  - ◊ käyvät ilmi dynaamisista malleista
  - ◊ olioon saapuvaan herätteeseen liittyvä toiminto vastaa luokan operaatiota. Arkkitehtuurista riippuen operaatio voidaan toteuttaa monella tavalla. Analyysin aikana operaatiota ei kiinnitetä vielä mihinkään.
  - ◊ olion tilaan liittyvä toiminto edellyttää operaation, joka voidaan sisällyttää vastaavan luokan määritykseen (esim. *verifyBankCode* - Consortium).
- sekalaisia ("shopping list operations"):
  - ◊ eivät riipu tietystä sovelluksesta eivätkä edellytä määrättyä suoritusjärjestystä (Meyer, 1988) vaan laajentavat kuvaa olion käyttäytymisestä.
  - ◊ pääteltävissä olioiden käyttäytymisestä todellisuudessa
  - ◊ ATM:ssä: tili (*sulje*), tili (*oikeutaPankkikortinKäyttö*), pankkikortin oikeutus (*lakkauta*), pankki (*luoTiliAsiakkaalle*)

Yleisperiaatteita yksittäisten operaatioiden sijoittamisesta:

- Olion attribuutteja kysyvät ja muuttavat sekä olion linkkejä luovat ja poistavat operaatiot sijoitetaan vastaavaan loukkaan (kapseloinnin periaate),
- Jotta olio pystyy tunnistamaan saapuvan viestin, sillä tulee olla vastaava operaatio (viestin nimi = operaation kutsumuoto).

Käyttötapauksissa kuvattujen toimintasarjojen jäsentäminen operaatioiksi ja niiden sijoittaminen luokkiin voidaan tehdä useammalla tavalla. Jacobson ym. (1992) on esittänyt kolme päästrategiaa, jotka eroavat toisistaan siinä, millaiset roolit käyttöliittymä-, ohjaus- ja liiketoimintaluokilla halutaan olevan:

- *upotetun ohjauksen strategia*:
  - ◊ käyttäytymisen ohjaus määritetään liiketoiminta- tai ohjausluokkien yhteydessä
  - ◊ voi tuottaa tehokkaan toteutuksen, mutta on protoiluvaiheessa hankala
- *dialogikeskeinen strategia*:
  - ◊ käyttäytymisen ohjaus määritetään pääosin käyttöliittymäluokissa
  - ◊ protoiluvaiheessa helppo, mutta toteutusvaiheessa strategia voi johtaa huonosti ylläpidettäviin käyttöliittymäluokkiin (=sovelluslogiikka hajaantunut käyttöliittymä- ja ohjausluokkiin)
- *sekastrategia*:
  - ◊ edellisten sekoitus
  - ◊ joustavampi mutta vaatii toteutuksen ja ylläpidon aikana enemmän kuria rajapintojen pitämiseksi yhdenmukaisina
- *tasapainostrategia*:
  - ◊ päävastuu käyttäytymisen ohjauksesta annetaan yhdelle tai useammalle ohjausoliolle, joka ottaa vastaan viestejä muilta olioilta ja jakaa niitä muille.
  - ◊ ohjauksen eristäminen yhteen luokkaan lisää lokaalisuutta ja voi siten tehdä muutosten tekemisen muihin luokkiin helpommaksi.

Aiemmin kuvattu MVC++ -lähestymistapa noudattaa tasapainopainostrategiaa. Esimerkkinä tästä strategiasta on kuvissa 4.42-4.43 esitetty kierrätysautomaatin toimintaa käyttötapauksen, luokkakaavion ja sekvenssikaavion avulla tilanteessa, jossa asiakas

syöttää pulloja, tölkkejä ja koreja automaattiin ja saa lopuksi kuitin (esitystapa ei noudata täysin UML:n notaatiota).

The use case starts in *Customer Panel* when *Customer* presses the start button. Sensors are then activated.

Customer can now turn in his returnable items via the *Customer Panel*. *Customer Panel* will tell the active object *Deposit Item Receiver* that an item has been handed in, as well as the measures of the item.

Deposit Item Receiver will use the entity objects *Can*, *Bottle* and *Crate*, together with the measured values from *Customer Panel*, to find out what type of item has been received. The daily total of the received object of this type will be enumerated in the object.

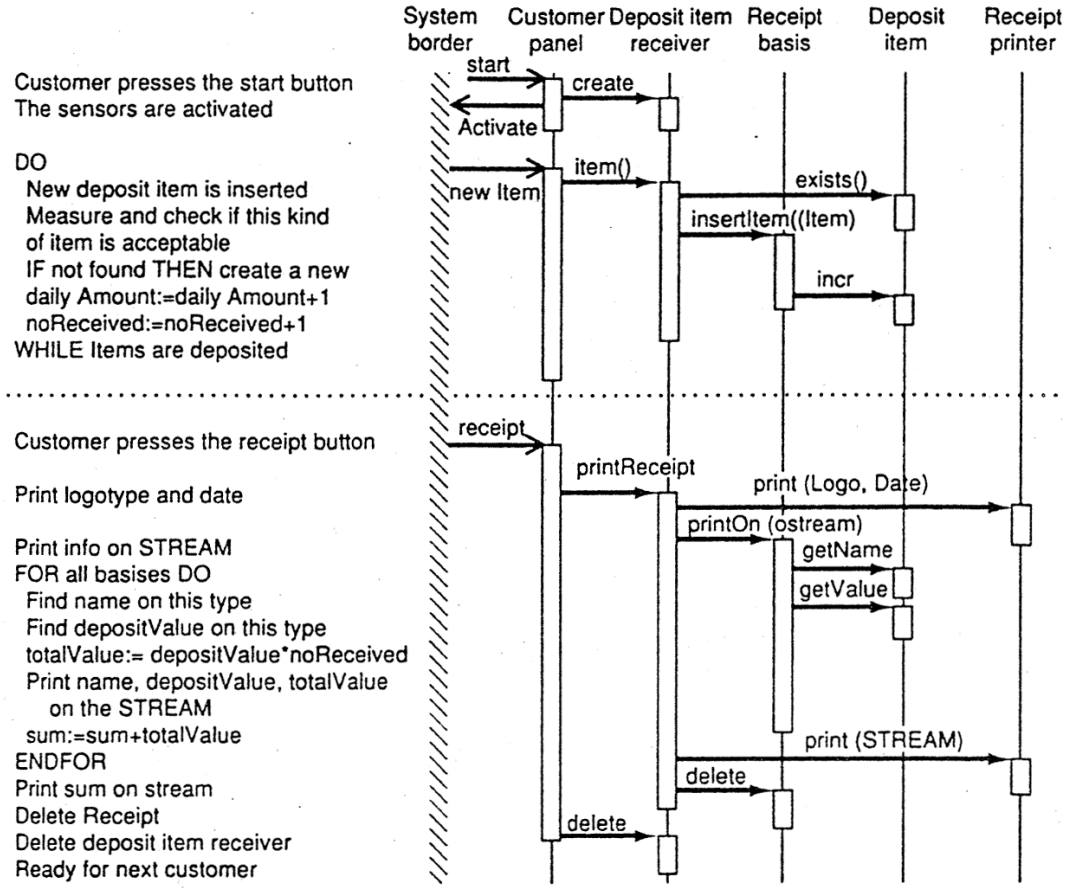
By means of the entity object *Receipt Basis*, *Deposit Item Receiver* will keep track of how many items of the current type the customer has turned in so far.

After turning in his returnable items, *Customer* will ask for his receipt. This is done via *Customer Panel*.

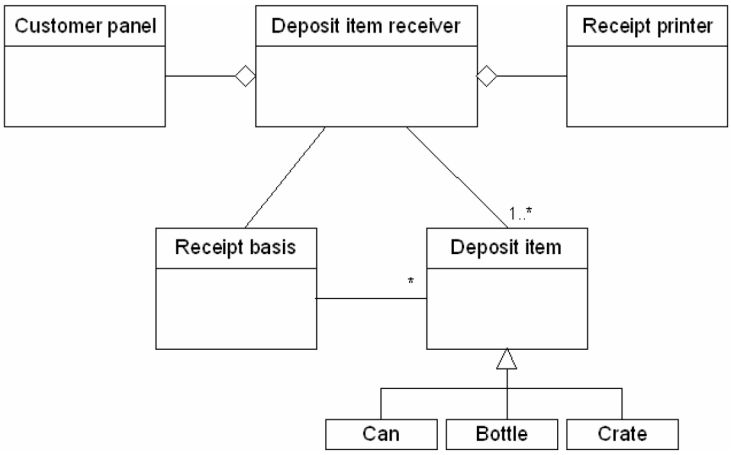
Deposit Item Receiver finds the information to be printed on the receipt. A line is printed for each item that has been turned in, stating the number of each type as well as the return value for this number. Finally, the total return value sum is printed on the receipt. The information on the receipt is taken from the *Receipt Basis*.

The information is printed on the interface object *Receipt Printer*, which is also told when the printout is completed.

Kuva 4.42: Käyttötapaus kierrätysautomaatille (Jacobson 1992, s. 189-190)



Kuva 4.43: Sekvenssikaavio kierrätysautomaatille (Jacobson, 1992, 189-190)



Kuva 4.44: Luokkakaavio kierrätysautomaatille (Jacobson 1992, s. 189-190)

4.5.3 Johdonmukaisuuden varmistaminen

Ylläpidettävien mallien johdonmukaisuus tulee tarkistaa aina, kun johonkin mallin osaan on tehty merkittäviä muutoksia. Analyysivaiheen malleista tärkein on kohdealuemalli (luokkakaavio), jota kannattaa ylläpitää myöhemmissäkin kehitysvaiheissa, vaikka kohdealueluokkien yksityiskohtainen toteutustapa tarkentuu suunnitteluvaiheen aikana – kohdealuemallia voidaan käyttää kommunikoinnin apuna päätettäessä järjestelmän jatkokehityksestä. Sen sijaan alustavat yhteistoimintakaaviot ja CRC-kortit toimivat paremmin kertaluontoisina työkaluina luokkien ominaisuuksien etsinnässä ja kontrollilogiikan hahmottamisessa. Kun tieto on tarkennettu suunnitteluvaiheen malleihin ja niiden päivityksestä huolehditaan (suunnitteluvaiheen luokkakaaviot, sekvenssikaaviot, tilakaaviot), alkuperäisten mallien ylläpito ei ole tarpeellista.

Esimerkkejä ohjeista:

- käy läpi malleja epäjohdonmukaisuuksien ja epäyhtenäisyyksien löytämiseksi ja korjaamiseksi,
- tarkasta, että luokkien, attribuuttien ja operaatioiden nimet esiintyvät samoina eri kaavioissa,
- tarkista, että kaikki käyttötapausmallissa esiintyvät toiminnalliset piirteet löytyvät mallinnetusta järjestelmästä; myös poikkeustapausten käsittely.
- tarkista, onko jokaista saapuvaa sanomaa vastaava operaatio vastaanottavan olion luokassa,
- tarkista, ovatko tilakaavioissa esiintyvät toiminnot vastaavan luokan operaatioina,
- yritä löytää luonnollisempia ratkaisuja “kummallisiin” osiin malleissa
- poista osuudet, jotka eivät sittenkään tunnu tarpeellisilta.

Jotkut UML-työkalut tukevat eri kaavioissa olevien riippuvuuksien seuraamista vähintään kaavioiden elementeistä kootulla mallitietokannalla (esim. StarUML) tai varoittavat mahdollisista suunnitteluvirheistä (esim. ArgoUML). Myös mallien luonti olemassa olevasta koodista (reverse engineering) tai synkronointi koodin muutosten kanssa (roundtrip engineering) on mahdollista sovelluskehittimillä. Kytkettäessä kaaviot ajettavaan koodiin malleista voi tulla tarpeettoman yksityiskohtaisia suunnittelua ajatellen, mutta tiettyyn koodin osiin rajatuista kaavioista voi olla hyötyä ratkaisujen dokumentoinnissa tai hahmotettaessa puutteellisesti dokumentoitua järjestelmää.

5 SUUNNITTELU

Suunnittelulla tarkoitetaan tässä tehtäväkokonaisuutta, jonka tarkoituksena on vastata kysymykseen, miten järjestelmä pystyy vastaamaan käyttäjien vaatimuksiin: esim.

- millainen on järjestelmän arkkitehtuuri,
- millaiseen teknologiaan ratkaisu tulee perustumaan,
- mitkä osat on tarkoitus toteuttaa milläkin teknologialla sekä
- miten kukin luokka on tarkoitus toteuttaa.

Analyysin tavoin myös suunnittelu voidaan jakaa neljään osaan: arkkitehtuurin suunnitteluun, staattiseen suunnitteluun, dynaamiseen suunnitteluun ja käyttöliittymien suunnitteluun. Koska tällä kurssilla ei ole mahdollisuutta käydä systemaattisesti läpi jokaista osa-aluetta, tarkastellaan seuraavassa ensin arkkitehtuurin suunnittelua ja sen jälkeen staattista ja dynaamista suunnittelua yhdessä oliosuunnittelun otsikon alla.

5.1 Arkkitehtuurin suunnittelu

Järjestelmän arkkitehtuurin suunnittelun tarkoituksena on analyysivaiheessa tehtyjen ratkaisujen (luku 4.1) nojalla suunnitella ja päättää osajärjestelmistä, komponenteista ja niiden välisistä rajapinnoista (interfaces). Järjestelmän *arkkitehtuurilla* tarkoitetaan puolestaan kuvausta osajärjestelmistä ja komponenteista ja niiden välisistä suhteista. Seuraavassa käsitellään arkkitehtuurin suunnittelua sekä loogisesta että fyysisestä näkökulmasta ja UML:n tarjoamia mahdollisuuksia arkkitehtuurien mallintamiseen[20].

RUP-menetelmässä arkkitehtuuri on jaettu viiteen eri näkymään, jotka kuvastavat vastaavien UML-mallien arkkitehtuurillisesti merkittäviä osia. RUP:n mallikokokonaisuuksilla (ks. luku 2.2) ja arkkitehtuurinäkymillä on yhteys: *malli* on järjestelmää kokonaisuutena kuvaava johdonmukainen kokonaisuus, *näkymä* on mallin osajoukko. Näitä molempia voidaan havainnollistaa *kaavioilla*, jotka ovat mallien tai näkymien visuaalisia esityksiä. RUP-menetelmän arkkitehtuurinäkymät ovat seuraavat:

- **Käyttötapausnäkymä** kuvaa arkkitehtuurisesti merkittävät käyttötapaukset (käytetään muiden näkymien johtamiseen)
- **Looginen näkymä** kuvaa alijärjestelmiin jaon ja pakettirakenteen
- **Toteutusnäkymä** kuvaa itse toteutettujen ja ulkopuolelta hankittujen komponenttien käytön
- **Prosessinäkymä** mallintaa järjestelmän rinnakkaisuutta: prosessit, säikeet, synkronointi
- **Sijoitusnäkymä** kuvaa järjestelmän ajonaikaista (fyysistä) konfiguraatiota

Kurssilla käsiteltävistä arkkitehtuurinäkymistä looginen näkymä vastaa karkeasti loogista arkkitehtuuria; toteutus- ja sijoitusnäkymät fyysistä arkkitehtuuria. Arkkitehtuurin mallintamiseen voivat vaikuttaa myös kokonaisarkkitehtuurit (*enterprise architec-*

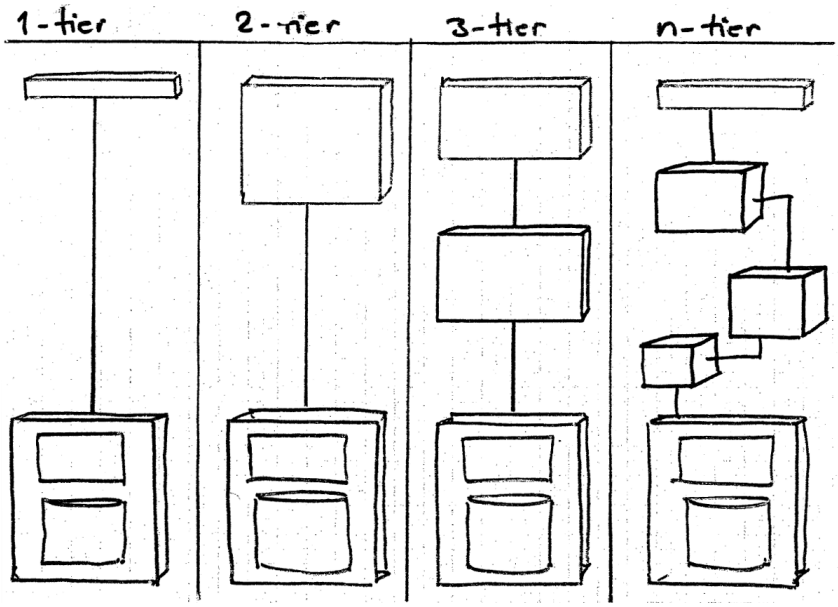
[20] Järjestelmän arkkitehtuuri ei ole riippuvainen siitä, onko järjestelmä toteutettu olioilla, mutta olioiden kapselointi ja tiedon piilotus auttavat komponenttien määrityksessä ja osajärjestelmiin jaossa. Esimerkki arkkitehtuurin esittämisestä moduuleilla (ennen olioiden yleistymistä): D.L. Parnas (1972). *On the Criteria To Be Used in Decomposing Systems into Modules*. http://sunnyday.mit.edu/16.355/parnas-criteria.html

ture - koko organisaation laajuisia kuvauksia prosessien, tiedon, järjestelmien ja laitteiston hallinnasta) tai toimialakohtaiset referenssiarkkitehtuurit.

UML tarjoaa useampia käsitteitä ja kaavioita arkkitehtuurin ja osajärjestelmien kuvaamiseksi. Loogista arkkitehtuuria havainnollistavaa pakettirakenteen kuvausta (luku 5.1.1.2) voidaan käyttää jo analyysivaiheessa, fyysistä arkkitehtuuria kuvaavat komponentti- ja sijoituskaaviot (luvut 5.1.2.1 ja 5.1.2.2) soveltuvat paremmin käytettäviksi järjestelmäsuunnittelussa ja toteutuksen aikana.

Esimerkkinä yksinkertaisesta arkkitehtuurien jaottelusta esitetään tunnettu arkkitehtuuriluokittelu, joka perustuu kerrokselliseen osittamiseen (Simon, 1996). Tässä luokittelussa kerrokset voidaan rinnastaa myös laitteisiin, joissa järjestelmää ajetaan, mutta välttämättä tällaista suoraa vastaavuutta ei ole (kuva 5.1):

- *yksitasoinen* (1-tier) arkkitehtuuri:
  - ◊ järjestelmän osat toimivat yhtenäisenä kokonaisuutena keskuskoneessa, johon päätteet tai PC:t emulaattorin kautta ovat yhteydessä,
- *kaksitasoinen* (2-tier) arkkitehtuuri:
  - ◊ järjestelmän osat on sijoitettu kahdenlaisiin koneisiin, asiakas- ja palvelinkoneiseen;
  - ◊ joko “lihava” asiakas: esityslogiikan lisäksi sovelluslogiikka asiakaskoneessa,
  - ◊ tai “lihava” palvelin: sovelluslogiikka on sijoitettu tiedonhallinnan osien kanssa palvelinkoneeseen (tyypillinen ohjelmistopakettiratkaisu)
- *kolmetasoinen* (3-tier) arkkitehtuuri:
  - ◊ järjestelmän osat on jaettu kolmenlaisiin koneisiin:
    - asiakaskoneessa on esityslogiikka,
    - sovelluspalvelimessa on sovelluslogiikka,
    - tietokantapalvelimessa on tiedonhallinta.



Kuva 5.1: Kerrosarkkitehtuureihin perustuva arkkitehtuuriluokittelu.

Edellisten lisäksi on alettu käyttää myös *N-tasoista* arkkitehtuuria (Panttaja, 1996) eli verkkokeskeistä arkkitehtuuria (Malmberg, 1996):

- "ultraohut" asiakaskone toimii pelkästään www-selaimena tai yleisemmin sovelluselaimena,
- verkkopalvelimet (www-palvelimet, sovelluspalvelimet) sisältävät sovelluslogiikan,
- tietokanta-/oliokantapalvelimessa on tiedonhallinta,
- asiakaskoneelta lähetetään palvelupyyntöjä yhdelle tai useammalle verkkopalvelimelle, jotka voivat edelleen kutsua toisiaan ja tietokantapalvelimia,
- verkosta haetaan asiakaskoneelle myös käyttöliittymän osia sitä mukaa, kun käyttäjä etenee sovelluksen eri osiin
- selaimessa ajettavat "rikkaat" internet-sovellukset: esim. Ajax-pohjaiset, erilaisia plugineja (Java-appletit, ActiveX-kontrollit, Flash) tai ajoympäristöjä (Adobe Flex, Java FX, MS Silverlight) - sisältävät usein enemmän selaimessa ajettavia sovelluslogiikan osia kuin perinteiset web-sovellukset.
- avainteknologioita ovat: Internet/WWW, Java/C#/Ruby, web-sovelluspalvelut (web services).

5.1.1 Looginen arkkitehtuuri

Looginen arkkitehtuuri kuvaa järjestelmän loogista rakennetta – järjestelmän komponenttien sisäisiä kytköksiä ja rajapintoja.

Osajärjestelmiin ja komponentteihin jakamisella saavutettavia etuja:

- saadaan aikaan pienempiä kokonaisuuksia, joita helpompi ymmärtää, hallita ja kehittää,
- osajärjestelmät ja erityisesti komponentit ovat potentiaalisia uudelleenkäytön kohteita,
- järjestelmien ylläpidettävyys paranee ja ylläpito näin tehostuu,
- siirrettävyys parantuu.

Arkkitehtuurisilla ratkaisuilla on yhteys ei-toiminnallisiin vaatimuksiin (laatuvaatimuksiin). Esimerkiksi järjestelmän suorituskyky ja ylläpidettävyys voivat olla ristiriitaisia tavoitteita, joihin voidaan ottaa kantaa suunniteltaessa arkkitehtuuria. Tyree & Akerman[21] toteavatkin, että arkkitehtuuri *on* dokumentoituja *päätöksiä* järjestelmän toteutuksen yleisistä periaatteista.

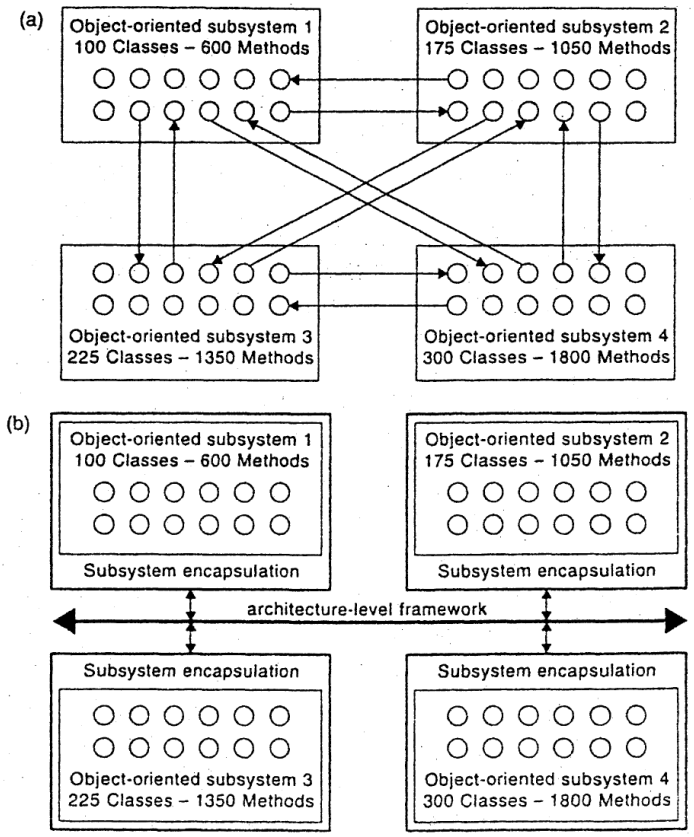
5.1.1.1 Osajärjestelmiin jako

Osajärjestelmä

- koostuu sellaisista järjestelmän osista, joilla on paljon yhteyksiä toisiinsa (high cohesion), samantapainen käyttäytyminen, sama fyysinen sijainti ja/tai toteutus samanlaisella laitteistolla (esim. avaruusaluksen osajärjestelmät: suunnistus, moottorin ohjaus, astronauttien huolto, tieteellisten kokeiden tuki),
- koostuu joukosta luokkia ja niitä yhdistäviä assosiaatioita,

[21] J. Tyree & A. Akerman (2005). Architecture decisions: demystifying architecture. IEEE Software, 22(2).

- on mahdollisimman vähän ja selväpiirteisesti yhteydessä muihin osajärjestelmiin; tämä tarkoittaa, että osajärjestelmällä tulee olla hyvin määritelty rajapinta, joka määrittää liitännät muihin osajärjestelmiin
- voidaan suunnitella ja toteuttaa itsenäisesti, kunhan rajapinta säilyy eheänä.



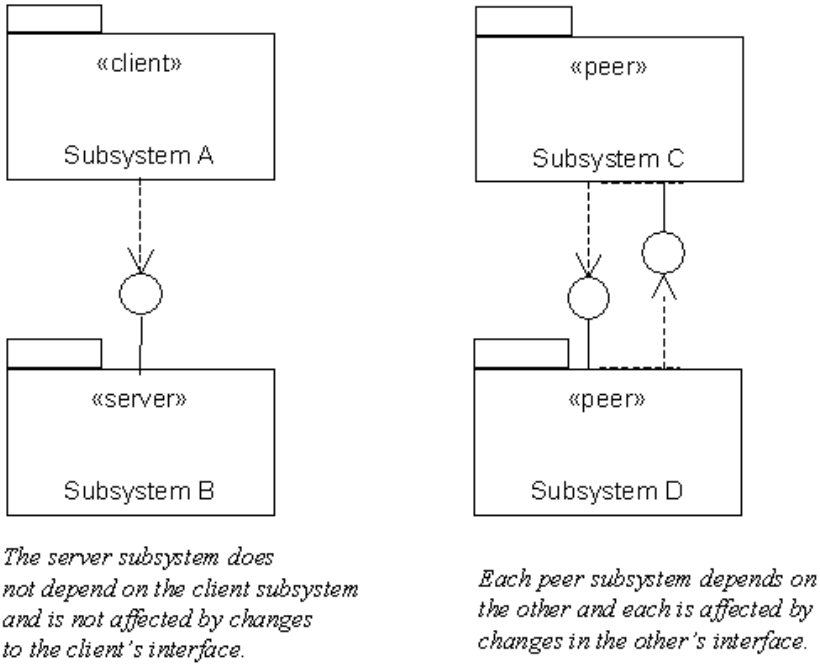
Kuva 5.2: Kuvassa (a) osajärjestelmillä on paljon yhteyksiä, epäselvästi määritellyt rajapinnat, ja mahdollisesti vääriin osajärjestelmiin sijoitettuja luokkia. Kuvassa (b) tilanne on korjattu selkeän arkkitehtuurin avulla.

Hyvällä arkkitehtuurin suunnittella voidaan vaikuttaa siihen, miten hyvin edellä mainitut vaatimukset voidaan toteuttaa (vrt. kuva 5.2). Yleisperiaatteena on, että luokkien (komponenttien, osajärjestelmien...) väliset riippuvuudet eli *kytkennät* (coupling) tulisi minimoida ja luokkien omat vastuualueet (sisäisten toimintojen yhtenäisyys, *koheesio*) ovat mahdollisimman selväpiirteisesti määriteltyjä.

Osajärjestelmien väliset suhteet voivat olla (kuva 5.3; Bennett ym., 1999 s. 258):

- asiakas-palvelin (client-server) -tyyppisiä: asiakas (client) pyytää palvelimelta (server) palvelusta; asiakas tuntee palvelimen rajapinnan, mutta palvelin ei välttämättä tunne asiakkaan rajapintaa
- tasavertaisia (peer-to-peer): kukin osajärjestelmä voi pyytää palveluja toisilta, kukin tuntee toistensa rajapinnat; monimutkaisempi hallita ja toteuttaa.

Osajärjestelmiin jakaminen voidaan tehdä useilla eri periaatteilla (arkkitehtuurityylit, arkkitehtuurimallit), joista tässä käsitellään pääosin kerros- ja ositusarkkitehtuureja.



Kuva 5.3: Client-Server ja Peer-to-Peer kommunikaatiotyylit

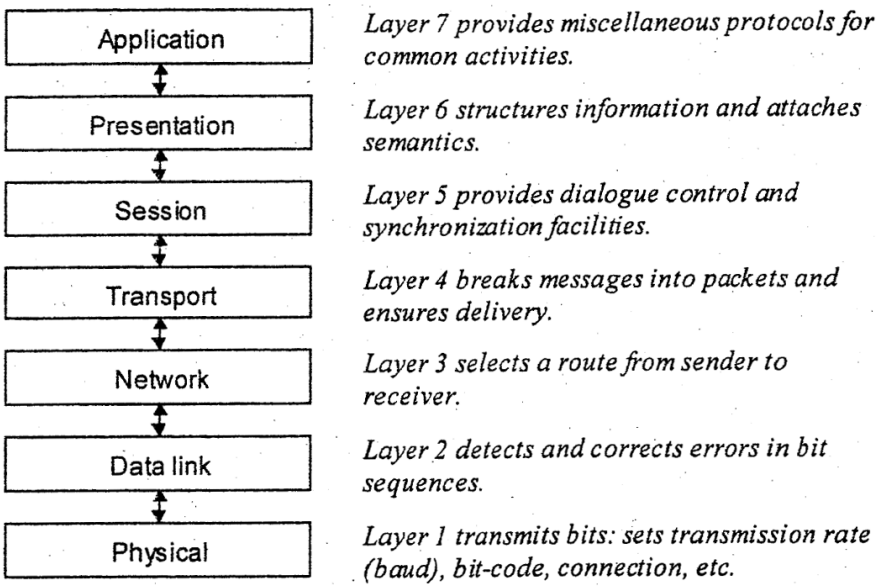
Kerroksellinen (layered) jakoperiaate:

- järjestelmä ositetaan kerroksittain (horisontaalisesti), jolloin osajärjestelmät kattavat koko järjestelmän toiminta-alan mutta vain tietyiltä rakenteellisilta ja/tai käyttäytymispiirteiltään (esim. käyttäjälle näkyvät osat, käyttöjärjestelmän kannalta tärkeät osat).
- kerrokset muodostavat tavallisesti hierarkian, jossa tietyn kerroksen osajärjestelmä on tietoinen alemmista kerroksista mutta ei ylemmistä (vrt. asiakas - palvelin -suhteet)
- perusmuodot (kuva 5.4; Bennett ym. 1999 s. 259):
  - ◊ suljettu arkkitehtuuri: kukin kerros on rakennettu *välittömästi* alemman kerroksen palvelujen varaan; edistää sitkeyttä (robustness) eli toteutusriippumattomuutta alemmissa kerroksissa tehdyistä muutoksista
  - ◊ avoin arkkitehtuuri: kukin kerros tuntee kaikkien alempien kerrosten palvelut; vähentää tarvetta määritellä esim. operaatio uudelleen joka kerrosta varten, mutta ei tue tiedon piilottamista.
- tavallisesti ylin kerros vastaa haluttua sovellusta, alin kerros vastaa käytettävissä olevaa laitteisto- ja varusohjelmistokantaa; järjestelmän siirrettävyyteen (portable) erilaisille laitteistoalustoille vaikuttaa se, kuinka monta kerrosta näiden välillä on.
- Tunnetuin esimerkki kerroksellisesta jaosta on ISO:n (International Organization for Standardization) määrittelemä verkkoprotokollapinon referenssimalli OSI (Open System Interconnection) (kuva 5.5; Bennett ym., 1999 s. 260).
- Kuvassa 1.5 esitettiin yksinkertainen kolmitasoinen eli -kerroksellinen arkkitehtuuri, jossa yli kerros vastaa käyttöliittymää, keskimmäinen liiketoimintalogiikkaa (ml. liiketoimintaluokat) ja alimmainen tietokantakohtaisia määrityksiä tallennettavista olioista.

Closed architecture—messages may only be sent to the adjacent lower layer.

Open architecture—messages can be sent to any lower layer.

Kuva 5.4: Suljettu kerroksellinen arkkitehtuuri vasemmalla ja avoin oikealla.



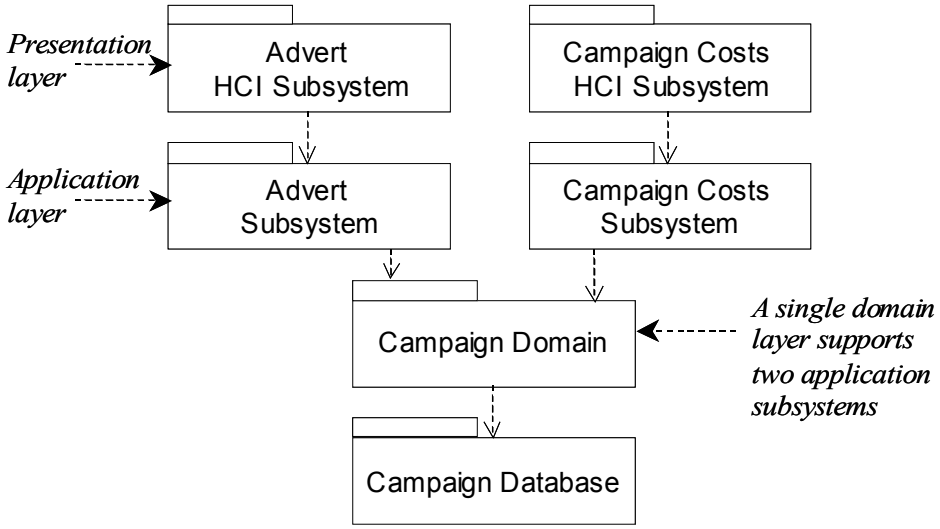
Kuva 5.5: ISO:n OSI (Open System Interconnection)-referenssimalli noudattaa kerrosarkkitehtuuria.

Partitioiva jakoperiaate:

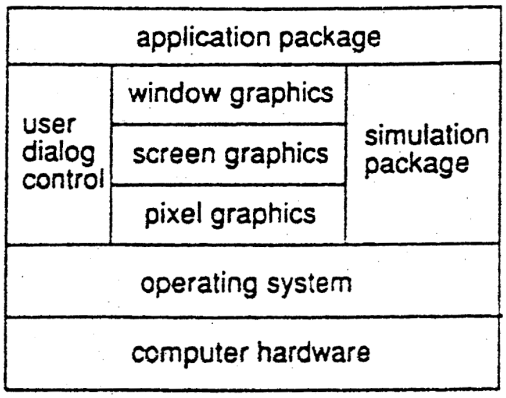
- Järjestelmä jaetaan vertikaalisesti mahdollisimman riippumattomiin osajärjestelmiin, joista kukin tarjoaa määrätyn tyyppistä palvelua (esim. käyttöjärjestelmä jakaantuu tiedostojen hallintaan, prosessin ohjaukseen, näennäismuistin hallintaan, laitehallintaan jne). Laajoilla kohdealueilla jakoperusteena voi olla myös käyttötapausten ja kohdealueluokkien ryhmittelystä lähtevä toiminnallinen jako (esim. mainosten ja kulujen hallintaan liittyvät alijärjestelmät mainostoimistoesimerkissä, kuva 5.6)
- osajärjestelmät tuntevat toisistaan vain rajapinnan tai osan siitä.

Sekamuoto:

- järjestelmä jaetaan sekä kerroksellisesti että vertikaalisesti osiin
- Kuvassa 5.6 (Bennett ym. 1999 s. 263) on esitetty mainostoimiston järjestelmän arkkitehtuuri ja kuvassa 5.7 (Rumbaugh ym. 1999) liikennevalojen simulointiohjelman arkkitehtuuri.



Kuva 5.6: Mainoskampanjajärjestelmän arkkitehtuurissa on neljä kerrosta, joista käyttöliittymä- ja sovelluslogiikkakerrokset on ositettu. Lisäksi mainosten ja kulujen hallintaa varten on määritelty erilliset käyttöliittymät.



Kuva 5.7: Simulointisovelluksen arkkitehtuuri.

5.1.1.2 Pakettirakenteen kuvaaminen

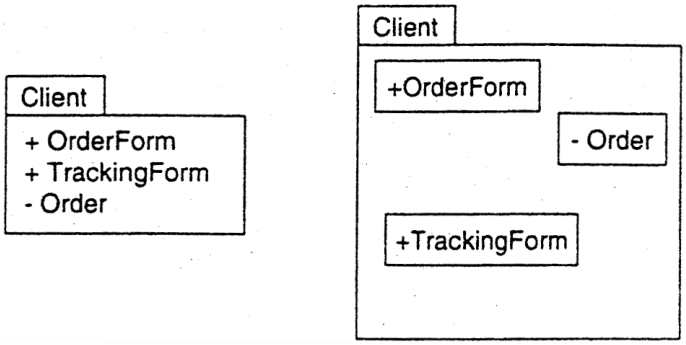
Paketti (package):

- kuvaa järjestelmän osaa, joka koostuu elementeistä. Elementteinä voivat olla luokat, komponentit, solmut, toiset paketit jne.
- rajaa "suljetun" maailman: kaikki paketin sisällä olevat elementit ovat muille paketin elementeille (periaatteessa) tunnettuja, mutta paketin ulkopuolella olevien elementtien tekeminen tunnetuksi edellyttää niiden tunnetuiksi tekemistä eli "tuontia" (import) (tuonnin tarkka merkitys vaihtelee eri ohjelmointikieliellä; vrt. Javassa *import*, C++:ssa *include* ja nimiavaruudet, Object Pascalissa *uses*)
- notaatio:
  - ◊ paketti esitetään kansiona, jonka yläosaan kirjataan paketin nimi ja alapuoliseen laatikko-osaan elementit.
  - ◊ Elementit voidaan esittää pelkillä nimillä tai kaaviolla (esim. luokkakaavio) (kuvat 5.8 ja 5.9; Booch ym, 1999).
  - ◊ Paketit yhdistetään toisiinsa riippuvuutta (dependency) osoittavalla katkoviivalla, jonka päässä oleva nuolenpää osoittaa siihen pakettiin, jonka jokin

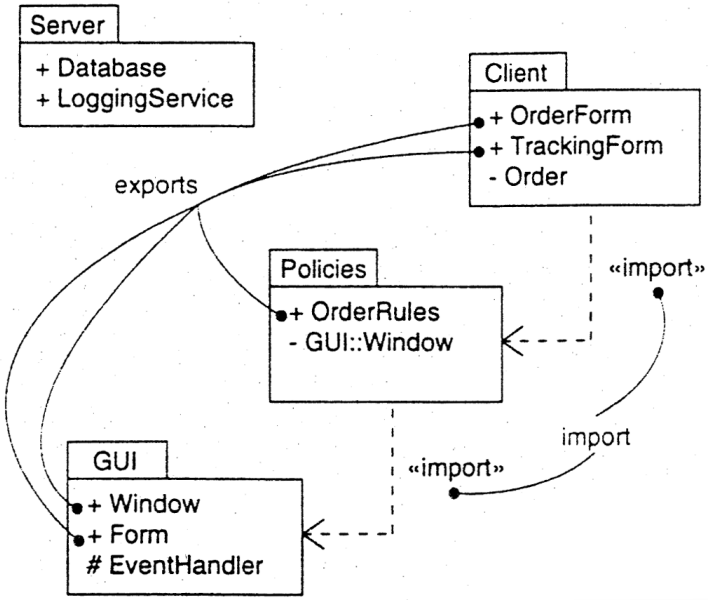
elementti halutaan tehdä tunnetuksi (import) toisen paketin elementeille. Viivan yhteydessä esitetään teksti <<import>>.

◊ Paketin kunkin elementin osalta voidaan erikseen merkitä, "näkyykö" elementti paketin ulkopuolelle:

- + - merkki (public): elementti näkyy,
- - -merkki (private): elementti ei näy paketin ulkopuolelle,
- # -merkki (protected): elementti näkyy vain sellaisille paketeille, jotka ovat yleistyshierarkiassa ko. pakettia alempana.

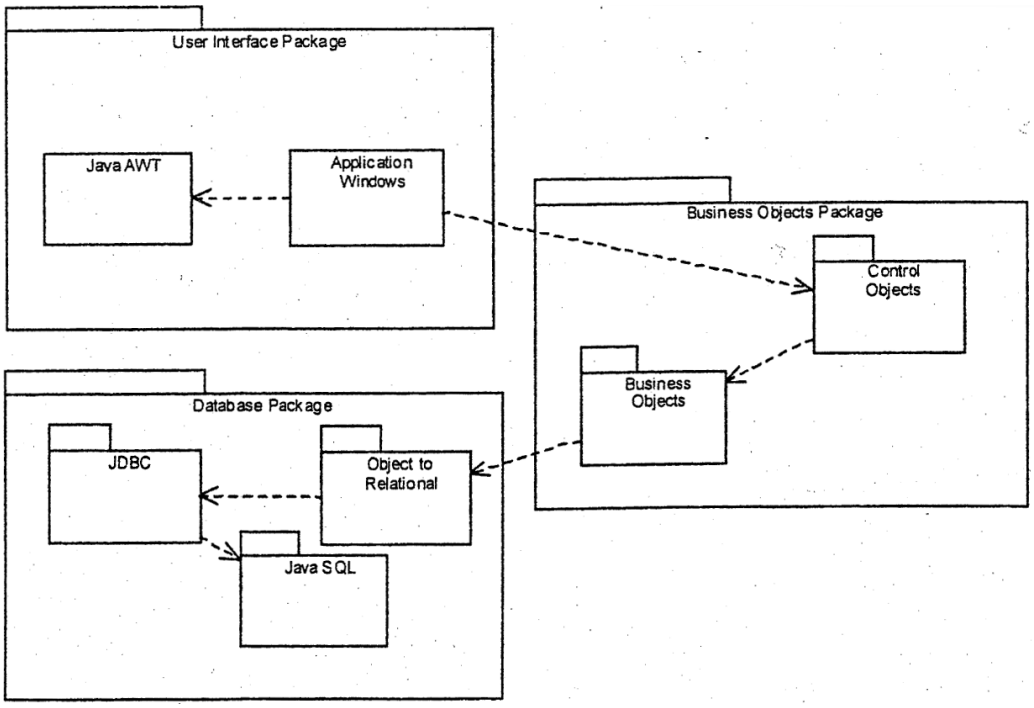


Kuva 5.8: Tekstuaalinen ja graafinen paketin elementtien kuvaus.



Kuva 5.9: Pakettien välisten riippuvuuksien kuvaus.

Kuvassa 5.10 (Bennett ym, 1999, s. 243) on esitetty 3-tasoisen arkkitehtuurin kuvaus, jossa käyttöliittymä koostuu kahdesta paketista, liiketoimintataso kahdesta paketista ja relaatiotietokantatekninen osuus kolmesta paketista. Viimeksi mainitussa osajärjestelmässä "Object to Relational" -paketti toimii rajapintana; ts. Business objects -paketista tulevat oliomallin mukaiset viestit muunnetaan relaatioympäristössä ymmärrettäviksi ennen kuin ne välitetään eteenpäin. Kuvasta nähdään edelleen, minkä pakettien tulee olla tunnettuja millekin paketeille (esim. Control objects -paketin tulee tuntea Business objects -paketin rajapinta, so. jokin kyseisen paketin elementeistä).



Kuva 5.10: UML-paketit esittävät kerroksia kolmitasoarkkitehtuurissa. Kaksi korkeinta kerrosta toimivat olioparadigman mukaisesti. Tietokantakerros on kuitenkin toteutettu perinteistä relaatiotietokantaa käyttäen. Tästä syystä tietokantakerroksessa joudutaan hyödyntämään "Object to Relational" pakettia, joka kuvaa oliot relaatioiksi ja päinvastoin mahdollistaen kahden eri paradigman rinnakkaiselon.

5.1.2 Fyysinen arkkitehtuuri

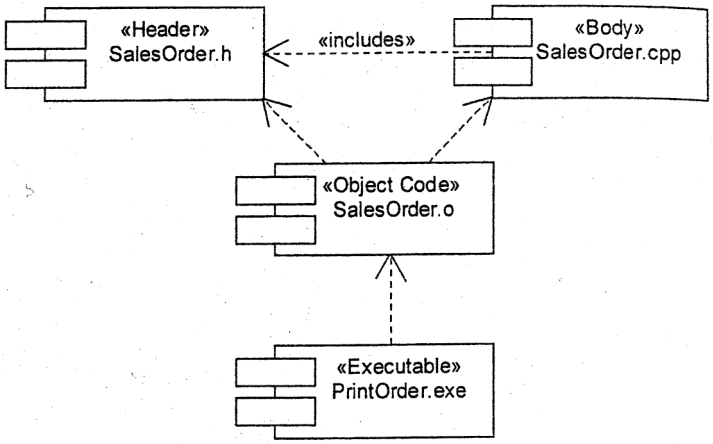
UML-toteutuskaavioita käytetään fyysisen arkkitehtuurin mallintamiseen ja järjestelmän ajonaikaisen kokoonpanon (sijoitus, deployment) kuvaamiseen.

5.1.2.1 Komponenttikaaviot

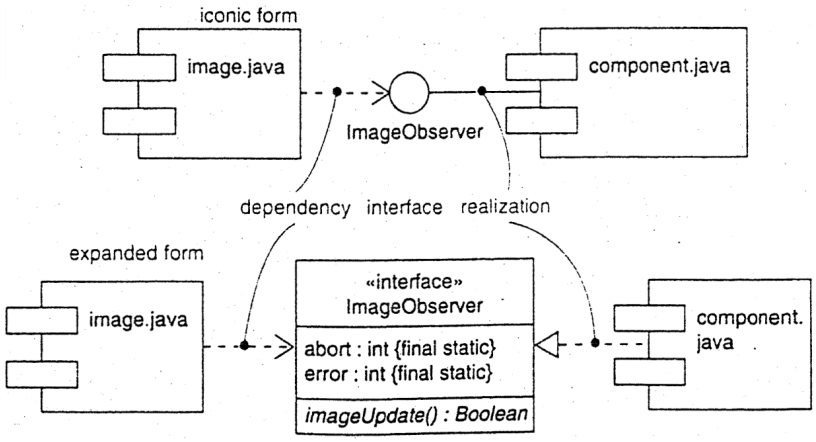
*Komponentilla (*component)

- tarkoitetaan fyysistä järjestelmän osaa, joka toteuttaa rajapintamääritykset (set of interfaces). Rajapinta muodostuu operaatioiden kutsumuodoista. Kun luokat ovat loogisia abstraktioita järjestelmän osista, komponentit ovat fyysisiä järjestelmän ohjelmoituja osasia.
- Komponentti on eräänlainen "standardi" palikka, jota voidaan toivon mukaan käyttää sellaisenaan muuallakin (uudelleenkäyttö!). Komponentit voivat olla esim. ohjelmia, dynaamisia kirjastoja (Windows: DLL, BPL; Linux: so), Java-luokkakirjastoja tai paketteja (class, JAR) tai esim. hajautettuja CORBA-komponentteja. Myös Web-sovelluspalveluita (web services) voidaan pitää komponentteina.
- notaatio:
  - ◊ Komponentti esitetään suorakulmiolla, jonka toisessa laidassa ovat rajapintaa kuvaavat pienemmät suorakaiteet.
  - ◊ Komponentteja voidaan ryhmitellä ja sijoittaa paketteihin,
  - ◊ Komponenttien välisiä suhteita voidaan kuvata komponenttikaaviolla (component diagrams). Yksinkertaisimmassa muodossaan komponentit yhdistetään toisiinsa samalla tavalla kuin paketit edellä (kuva 5.11; Bennett ym, 1999). Jos on tarvetta kuvata tarkemmin myös rajapinnat, joiden "läpi" kom-

munikointi tapahtuu, voidaan käyttää kuvan 5.12 (Booch ym. 1999) muotoa. Kuvassa on esitetty kaksi vaihtoehtoista tapaa (ikoninen tapa ja laajennettu tapa) sen osoittamiseksi, miten Image.java (ajonaikana po. class) -niminen komponentti käyttää Component.java-komponentin palveluja. Käyttöä on tarkennettu nimeämällä komponentin toteuttama rajapinta (ImageObserver), jonka operaatioita Image.java -komponentti kutsuu. Ylemmässä kuvassa rajapinta on esitetty vain pienellä ympyrällä. Alemmassa kuvassa rajapinta on kuvattu luokkasymbolia käyttäen, jolloin rajapintaan kuuluvat attribuutit ja metodit näkyvät. Katkoviiva kuvaa riippuvuutta (dependency – Image.java riippuu ImageObserver-rajapinnasta) ja yhtenäinen/katkoviiva kuvaa toteutusta (realization – Component.java toteuttaa ImageObserver-rajapinnan).



Kuva 5.11: Komponenttikaavio esittää C++ -ohjelmiston kehitysaikaista näkymää.



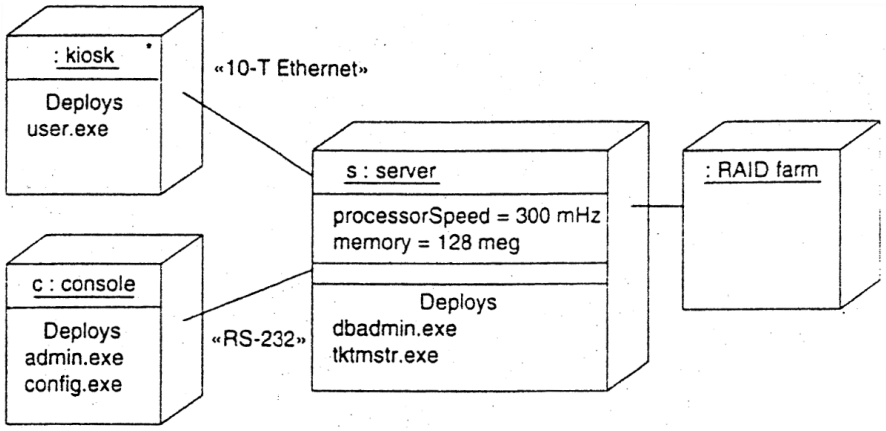
Kuva 5.12: Ikoninen ja laajennettu kuvaustapa komponenteille ja rajapinnoille.

5.1.2.2 Sijoituskaaviot

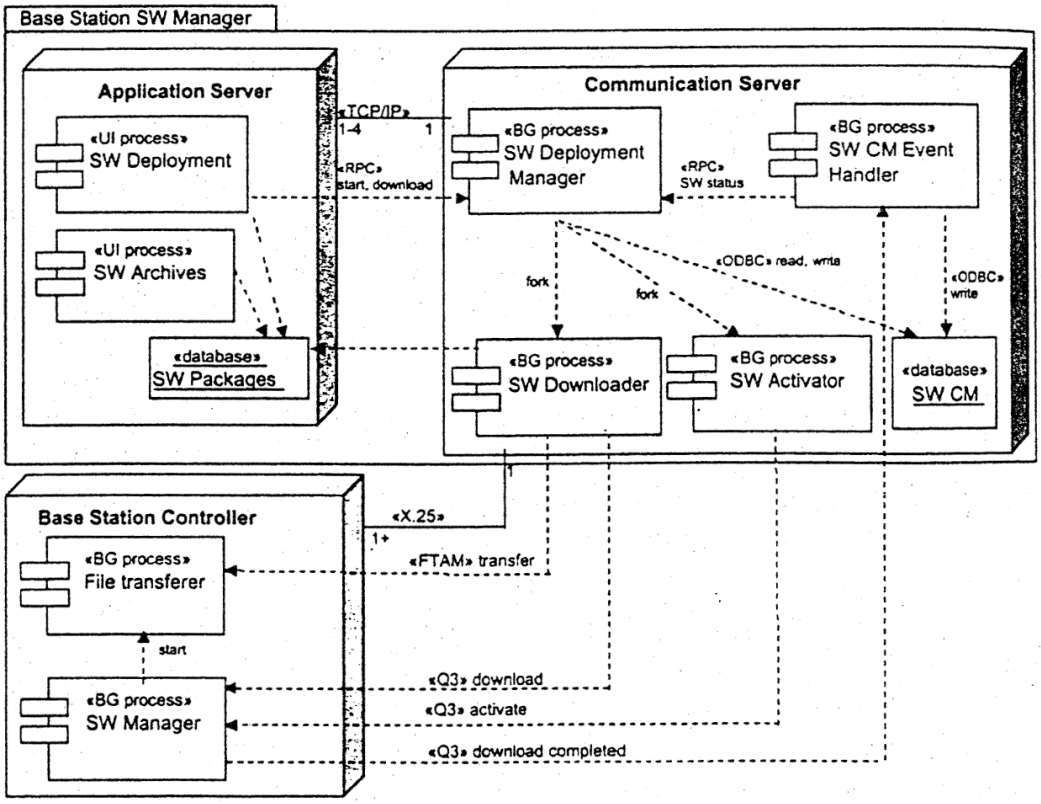
Solmulla (node)

- tarkoitetaan fyysistä järjestelmän osaa, jolla voi olla jonkin verran myös omia käsittely- ja muistiresursseja. Solmu on komponenttien sijoitus- ja ajopaikka.
- osoitetaan, mihin komponentit on tarkoitus sijoittaa.
- notaatio:
  - ◊ solmu esitetään kuutiona, jonka sisällä voidaan esittää komponentteja, joko nimien tai kaavioiden avulla.

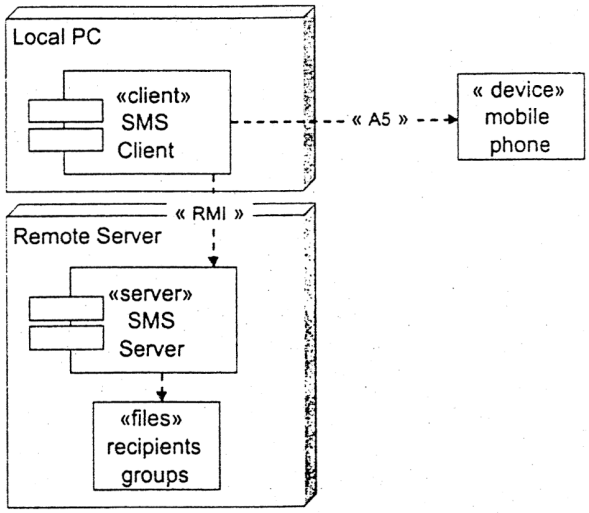
Sijoituskaavio (deployment diagram) kuvaa fyysisen (ajoaikaisen) järjestelmän solmut, niiden yhteydet sekä niiden sisältämät komponentit. Solmujen yhteydet (connection) ovat fyysisiä, esim. Ethernet-yhteys. Yhteyden luonne voidaan kuvata tarkemmin sopivalla stereotyypillä (kuva 5.13; Booch ym. 1999, 366; kuva 5.14; Jaaksi ym., 1999, 218). Kuvassa 5.15 (Jaaksi ym., 1999, 43) on esitetty sijoituskaavio järjestelmästä, jonka avulla voidaan lähettää tekstiviestejä matkapuhelimiin.



Kuva 5.13: Komponenttien sijoittamisen mallintaminen.

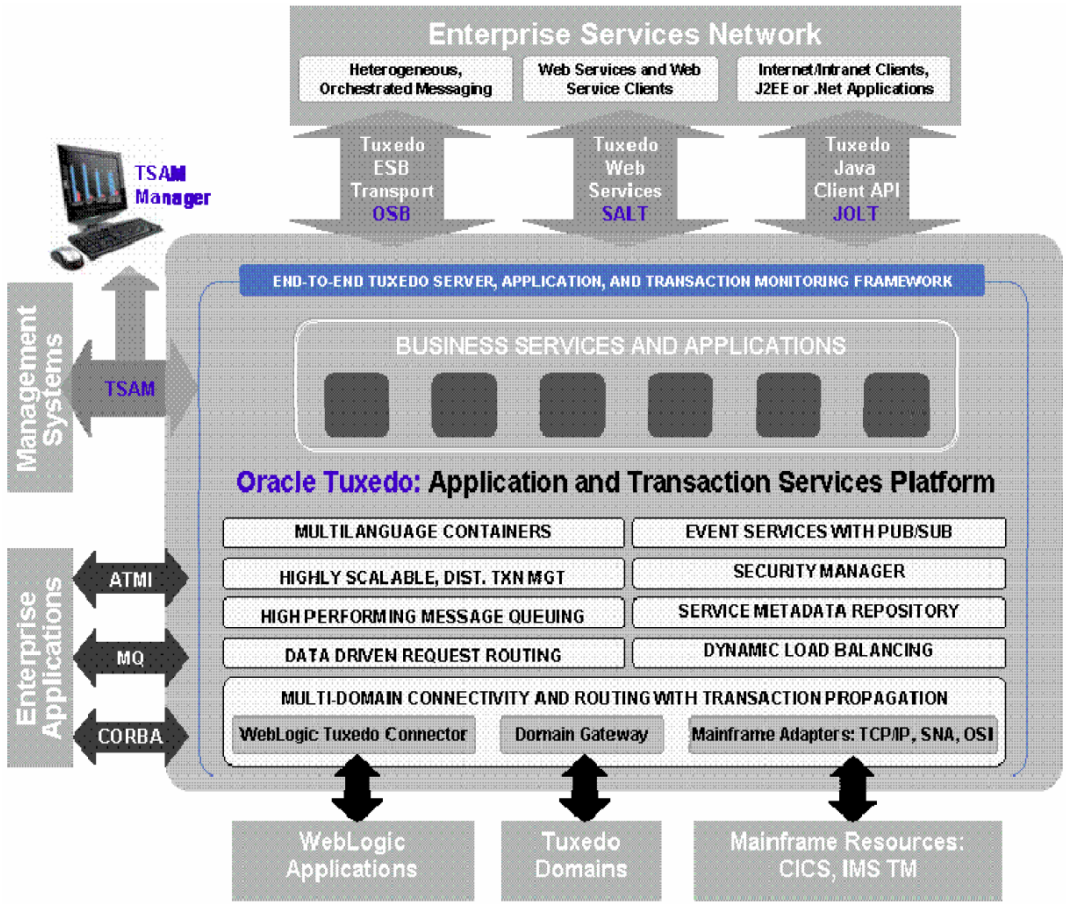


Kuva 5.14: Komponenttien sijoittamisen mallintaminen matkapuhelinverkoissa.



Kuva 5.15: Ajonaikainen näkymä komponenttiarkkitehtuurista.

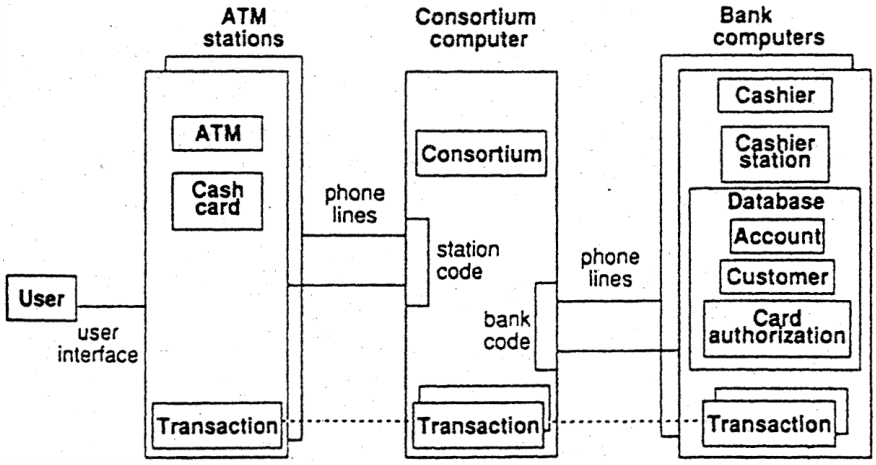
Järjestelmän osien sijoittaminen useampaan koneeseen (solmuun) johtaa komponenttiarkkitehtuuriin. Kuvassa 5.16 on annettu esimerkki erään väliohjelmiston (Tuxedo) avulla toteutusta komponenttiperusteisesta arkkitehtuurista. Tällöin sovelluslogiikka kirjoitetaan "itsenäisiksi" palveluiksi, joita kutsutaan tarpeen mukaan. Komponenttien tuotanto vaatii erillisen organisoinnin ja laadunvarmistamisen.



Kuva 5.16: Middleware-komponenttiarkkitehtuuri.

Lopuksi kuvataan *ATM-järjestelmän arkkitehtuuri* (kuva 5.17; Rumbaugh ym, 1991) ja vaatimukset sen käytölle:

- koostuu kolmesta osajärjestelmästä, joista kullakin on erilainen sisäinen arkkitehtuurinsa:
- ainoat pysyvät tietovarastot ovat pankkien järjestelmissä tietokantoina/oliokantoina; tapahtumankäsittelyn ajaksi kyselyn/muutoksen kohteena olevat tilit lukitaan; tapahtumien välittäminen ja/tai käsittely koskettaa kaikkia kolmea osajärjestelmää
- liittoutuman tietokoneen tulee olla tarpeeksi tehokas, jotta se pystyy selviytymään myös käyttöhuipuista; tilapäisesti tapahtuma voi joutua jonoon, mutta siitä on ilmoitettava asiakkaalle,
- pankin koneen tulee myös olla tehokas ja tarjota riittävästi levymuistikapasiteettia
- Suojaus- ja oikeellisuusvaatimukset priorisoidaan hyvin korkealle. Kunkin fyysisen yksikön osalta on varmistettava, että tapahtuma välittyy oikeana verkon kautta. Tarvitaan siis aukoton vahvistusmenettely, ennen kuin lopullinen päivitys voidaan tehdä tietokantaan.
- pankkiautomaatin tulee ilmoittaa käyttäjälle, jos yhteys on poikki; muistakin poikkeustilanteista (kuten käteisen loppuminen, kuittipaperin loppuminen) tulee huolehtia.



Kuva 5.17: ATM-järjestelmän arkkitehtuuri (kuvattu sijoituskaavion mukaisia elementtejä – kuvassa ei käytetty UML-notaatiota).

5.2 Mallit ja uudelleenkäyttö

Tietojärjestelmien kehityksessä törmätään toistuvasti hyvin samantapaisiin rakenteisiin. Tästä syystä on pyritty löytämään yleisluonteisia ratkaisuja, joiden soveltaminen tilannekohtaisesti johtaisi vähemmällä vaivalla laadukkaampaan tulokseen kuin "from scratch"-periaatteella toimittaessa. Tällaisia ratkaisuja ovat mallit (patterns) ja sovelluskehykset (application frameworks). Malleista tunnetuimpia ovat suunnittelumallit[22] (design patterns), jotka sijoittuvat tarkastelutasoltaan arkkitehtuurisuunnittelun ja yksityiskohtaisen suunnittelun välille. Sovelluskehykset ovat suoraan toteutuksessa hyödynnettäviä komponentteja (joiden käyttötapa on kuitenkin "käänteinen" perinteisiin komponentteihin ja kirjastoihin nähden), jotka ohjaavat tietynlaisen arkkitehtuurin käyttöön sovellusta rakennettaessa ja sisältävät usein tarvittavaa toiminnallisuutta.

[22] Design pattern on käännetty myös ratkaisumalliksi, joka itse asiassa paremmin kuvaa sitä, mihin mallilla pyritään. Suunnittelumalli on kuitenkin vakiintuneempi suomennos.

5.2.1 Suunnittelumallit

Suunnittelumalli (design pattern)

- tarkoittaa nimettyä tapaa suunnitella järjestelmän osa tietyn usein esiintyvän ongelman ratkaisemiseksi tietyssä kontekstissa,
- koskee yleensä muutamia (3 - 5) luokkia ja niiden välisiä suhteita.
- ei anneta ohjelmistona vaan määrämuotoisena (pattern template) dokumenttina, jossa esitetään mallin käyttämiseen tarvittavat oleelliset tiedot, esim.
  - ◊ Nimi
  - ◊ Tarkoitus (ongelma jonka malli ratkaisee) ja konteksti (esiehdot tai tilanne, jossa ongelma ilmenee)
  - ◊ Vaikuttavat tekijät (forces – ongelmat, jotka ratkaisussa tulee huomioida)
  - ◊ Ratkaisu (kuvaus mallin komponenttien staattisista ja dynaamista ominaisuuksista)
  - ◊ Mallin kuvaukseen voi sisältyä myös tunnettuja sovellusesimerkkejä, etujen ja haittojen arviointia, toteutukseen ja soveltamiseen liittyviä huomioita ja viitteitä muihin ongelma-alueeseen liittyviin malleihin
- suunnittelumallit *löydetään* olemassa olevista järjestelmistä toistuvina ja hyväksi havaittuina ratkaisuina, niitä ei *keksitä* itse
- tarjoavat kehittäjille yhteisen sanaston keskusteltaessa suunnitteluratkaisuista

Varsinaisten suunnittelumallien lisäksi eritasoisia malleja voidaan tarkastella prosessin eri vaiheissa (analyysimallit vs. suunnittelumallit), tarkkuustasolla (organisaatiotason mallit vs arkkitehtuurimallit – toteutukseen ja kielen ominaisuuksien tasolla puhutaan idiomeista) sekä positiivisina tai negatiivisina ilmiöinä (antimallit, *antipatterns*). Yhteenliittyviä malleja voidaan ryhmitellä edelleen mallikieliksi. Malleja on kehitetty myös yksittäisille sovellusalueille ja ohjelmistoprosessin muihinkin vaiheisiin, kuten Adolphin ym. laatima mallikieli käyttötapausten kirjoittamiseen[23].

Historiaa:

- Suunnittelumallien dokumentointi sai alkunsa (rakennusten) arkkitehtuuriratkaisuja kuvanneesta kirjasta (Alexander ym, 1977).
- 1984: Knuth: kirjallinen ohjelmointi (literate programming) – lähestymistapa, jossa ohjelmaa ja sen dokumentaatiota pidetään kirjallisena tuotoksena. Dokumentaatiossa mukana ratkaisujen perusteluja ja vaihtoehtoisten ratkaisujen pohdintaa
- 1987: Beck & Cunningham sovelsivat Alexanderin ideoita ohjelmistotekniikkaan kehittämällä mallikielen Smalltalkin käyttöliittymäkehityksen opettamisen tueksi[24]
- 1990-luvun vaihteessa Coplien kokosi C++:n kooditason malleja (idiomeja)
- 1993: muodostui ryhmä (Hillside-ryhmä), joka otti tavoitteekseen ajattelutavan istuttamisen oliosuunnitteluun.
- 1994: Ensimmäinen suunnittelumalleja käsittelevä Pattern Languages of Programs (PloP)-konferenssi.
- 1995: Design Patterns -kirja (GoF) ja PortlandPatternRepositoryn c2-wiki[25]. Suunnittelumallien käyttö ohjelmistokehityksessä alkaa yleistyä.

---

[23] S. Adolph, P. Bramble, A. Cockburn, A. Pols (2002). Patterns for Effective Use Cases.

[24] K. Beck & W. Cunningham (1987). Using Pattern Languages for Object-Oriented Programs http://c2.com/doc/oopsla87.html

[25] http://c2.com/cgi/wiki?WikiHistory

Kirjallisuutta:

- Suunnittelumallien alueelta tunnetuin julkaisu on ns. “gang-of-four book” eli GoF (Gamma ym., 1995)
- Arkkitehtuurimallit: ns. Siemens-kirja. Buschmann ym., Pattern-Oriented Software Architecture - A System of Patterns (1996)
- Analyysimallit: M. Fowler (1996), Analysis Patterns: Reusable Object Models
- Antimallit: W. Brown, R. Malveau, H. "Skip" McCormick, T. Mowbray (1998), AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis.
- Tiivis yleisesitys arkkitehtuurin, yleisten oliosuunnitteluperiaatteiden ja suunnittelumallien suhteesta: R. Martin (2000), Design Principles and Design Patterns[26]
- Laajahko linkkilista ja mallien historiasta ja käsitteistä: B. Appleton (2000), Patterns and Software: Essential Concepts and Terminology[27]

5.2.1.1 Organisaatiotason (anti)malleja

Organisaatiotason mallit liittyvät projektinhallintaan ja kehitystyön yleiseen organisointiin. Kaikki tässä käsitellyt organisaatiotason mallit ovat antimalleja. Analysis Paralysis ja Mushroom Management -mallit on esitetty alun perin Brownin ym. kirjassa, Architects Don't Code on määritelty c2-wikissä[28].

Analyysihalvaus (Analysis Paralysis)

- Kuvaa erityisesti dokumentoinnin osalta ”raskaissa” projekteissa tai menetelmissä ilmenevää tarpeettoman tarkkoihin yksityiskohtiin meneviä analyysivaiheen malleja. Analyysihalvauksesta kärsivässä projektissa tunnusomaista on, etteivät kohdealueen asiantuntijat enää ymmärrä helposti dokumentaatiota niiden liian teknisiin piirteisiin menevien yksityiskohtien vuoksi.
- Ratkaisuja analyysihalvaukseen ovat inkrementaalinen kehitys, ohjelmistoprosessin keventäminen ja ketterien menetelmien käyttö. ICONIX-menetelmässä (Rosenberg & Scott 1998) analyysihalvausta pyritään välttämään systemaattisesti erityisillä varoituksilla, joita esitetään prosessin eri vaiheiden yhteydessä[29].

Mushroom Management

- Antimalli kuvaa projektiorganisaatiota, jossa kehittäjät on erotettu asiakkaista - ”älä päästä ohjelmistokehittäjiä puhumaan loppukäyttäjien kanssa”. Antimalli voi esiintyä, jos kehittäjille halutaan antaa ”työrauha” kehitystä varten tai jo käytetty prosessi ja organisaation roolijako noudattaa liian tiukasti vesiputoumallia (esim. kokonaan erillinen ryhmä vastaa vaatimusmäärittelystä ja analyysistä – toteutus ulkoistettu muualle). Antimallin riskejä ovat ”väärän” järjestelmän rakentaminen tai järjestelmän todellisten tavoitteiden puutteellinen ymmärrys. Jos vaatimuksia ei ole alun perin mallinnettu riittävän täsmällisesti eikä kehittäjillä ole toteutusvaiheessa suoraa yhteyttä käyttäjiin, kehittäjät saattavat joutua tekemään omia vaati-

[26] http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf

[27] http://www.cmcrossroads.com/bradapp/docs/patterns-intro.html

[28] http://c2.com/cgi/wiki?ArchitectsDontCode

[29] http://www.iconixsw.com/UMLBook/AnalysisParalysis.html

muksiin tai kohdealueeseen liittyviä oletuksia, jotka eivät pahimmillaan tule dokumentoitua muualle kuin koodiin.
- Antimallia voidaan riskianalyysin sisältävällä spiraalimaisella kehitysprosessilla, käyttäjäpalautteen järjestelmällisellä keräämisellä ja kohdealueen asiantuntijan (tai asiakkaan) ottamisella mukaan kehitystiimiin (mikä on vakiokäytäntö mm. XP-kehitysmenetelmässä).

Arkkitehti ei koodaa (Architects don't Code)
- Antimalli kuvaa tilannetta, jossa arkkitehdilla ei ole riittävän syvällistä käsitystä järjestelmän teknisestä ympäristöstä, toteutusratkaisuista tai käytettävien komponenttien toimivuudesta. Arkkitehdin rooliin liittyy sekä asiakkaan kanssa kommunikointia (erityisesti laatuvaatimusten määrittämisessä) että teknistä suunnittelua. Työ voi painottua enemmän suunnitteluun ja muiden kehittäjien ohjaukseen kuin ohjelmointiin, mutta jos suunnittelun vaikutuksesta toteutukseen ei ole ymmärrystä, riskinä ovat toteutuskelvottomat tai tehottomat suunnitelmat, joita kehittäjät voivat joutua muokkaamaan arkkitehdista riippumatta.
- Ihannetapauksessa arkkitehdin tulisi rajapintojen ja pakettirakenteen määrityksen lisäksi myös toteuttaa järjestelmästä tiettyjä kriittisiä osia – mahdollisesti luoda yleistä sovelluskehystä järjestelmän ympärille, jota muut kehittäjät voivat laajentaa edelleen. Useimpien komponenttien ja alijärjestelmien toteutus kuuluu kuitenkin arkkitehdin sijaan kehittäjille.

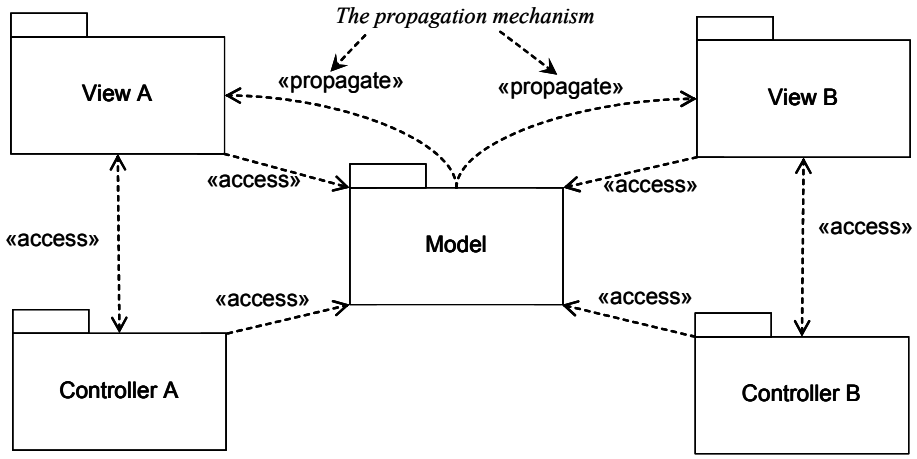
5.2.1.2 Arkkitehtuurimalleja

Luvussa 5.1.1 käsitellyt kerros- ja ositetut arkkitehtuurit ovat esimerkkejä koko järjestelmän laajuisista arkkitehtuurimalleista (arkkitehtuurityylit, architectural styles). Tässä esitellään järjestelmän tiettyyn osaan vaikuttavia (mutta kuitenkin suunnittelumalleja laajempia rakenteita koskevia) arkkitehtuurimalleja.

MVC (*Model-View-Controller*) (kuva 5.18; Bennett 2006)

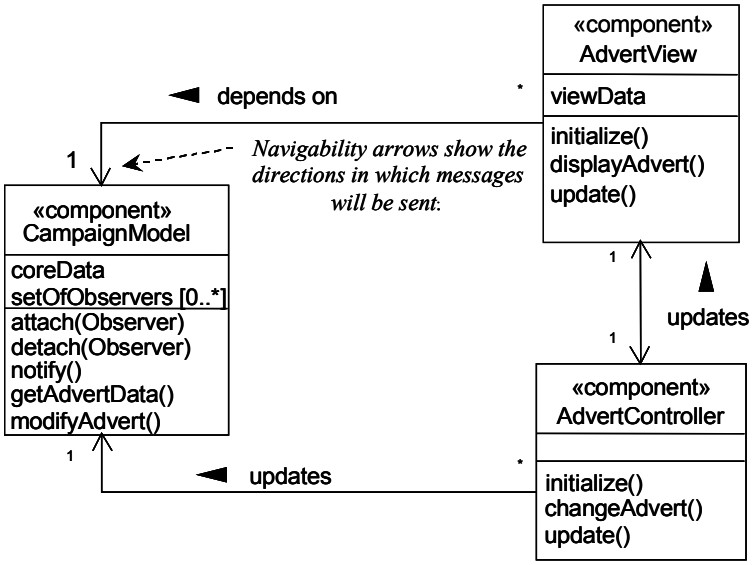
MVC on T. Reenskaugin 1970-luvun lopulla Smalltalk 80 -kielen käyttöliittymäkirjastoa varten kehittämä malli. MVC:n päätavoitteena on tukea käyttäjän mentaalista mallia kohdealueesta ja mahdollistaa kohdealueen tietojen suora muokkaus ("direct manipulation"). Näkymän tarkoitus on näyttää käyttäjälle tietty osa mallista, ohjaimen tarkoitus on muokata mallia käyttäjän toimintojen vasteina (tiettyyn malliin voi liittyä useita näkymä- ja ohjainolioita, joiden synkronoinnista malli huolehtii).

MVC poikkeaa kerrosarkkitehtuureista sekä käsitteellisesti ja toimintalogiikaltaan. MVC:n malliluokat vastaavat karkeasti kolmikerrosarkkitehtuurin tiedonhallintakerrosta - ohjaus - ja näkymäluokat sijoittuvat puolestaan käyttöliittymäkerrokselle. Kolmikerrosarkkitehtuurin sovelluslogiikkakerroksella ei ole suoraa vastinetta MVC:ssä.



Kuva 5.18: MVC-malli kahdella samaan malliin liittyvällä näkymällä ja ohjaimella

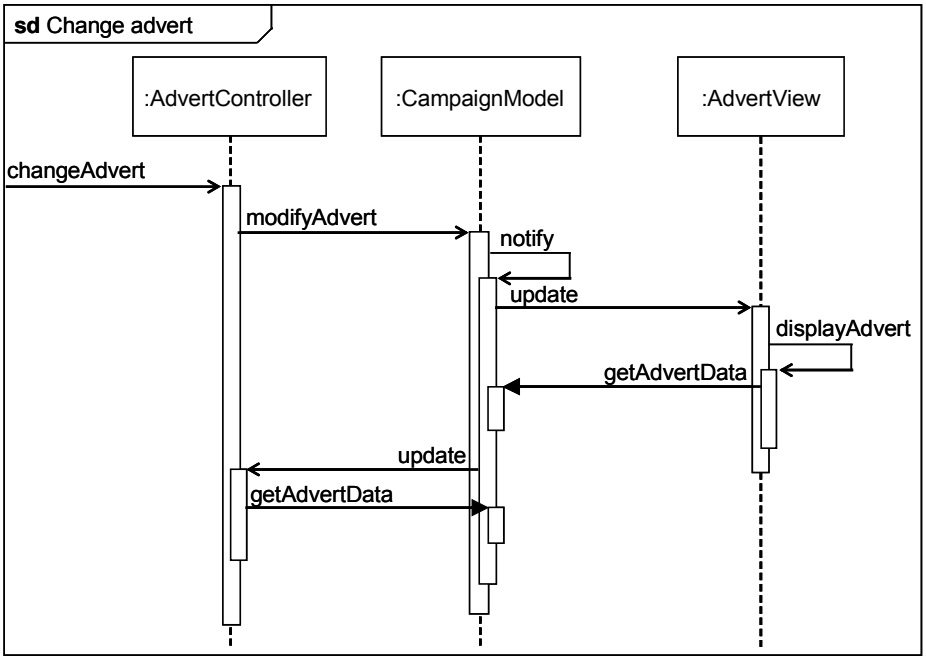
Kuvat 5.19 ja 5.20 havainnollistavat MVC-mallin käyttöä mainostoimistojärjestelmässä tilanteessa, jossa kampanjan tietoja näytetään ja muokataan useilla eri käyttöliittymillä samanaikaisesti. *CampaignModel*-luokka sisältää kampanjan tiedot. *AdvertView-luokka* sisältää kampanjaluokan käyttäjän näkymän, *AdvertController* käsittelee käyttäjän toiminnot, jotka liittyvät näkymäluokkaan ja tarvittaessa muokkaa mallia (näkymäluokan ei tarvitse tietää ohjainluokan yksityiskohtia, vaan ohjainluokkaa voidaan kutsua esim. rajapintojen avulla määriteltyjen tapahtumien kautta). Jos malli on muuttunut, malliluokka ilmoittaa tietojen muuttumisesta rekisteröityneille näkymä- ja tarvitaessa ohjausluokille (setOfObservers-taulukko) niiden *update()*-metodia kutsumalla. Malliluokka ei tiedä käyttöliittymäluokkien yksityiskohtia, joten update-kutsun seurauksena näkymäluokka kysyy malliluokalta uudet tiedot (näkymien päivitykseen liittyvä osuus mallista on samalla esimerkki Tarkkailija-suunnittelumallista).



Kuva 5.19: MVC-malli sovellettuna mainostoimiston kampanjaluokkaan

MVC:ssä näkymä/ohjain-pareja voidaan koota kerroksiin esim. Rekursiokooste-suunnittelumallin tyylisellä mekanismilla (graafinen käyttöliittymä koostuu päällekkäisistä komponenteista, joista kukin saa tapahtumia käyttäjän toimintojen perusteella: ikkuna koostuu paneeleista, joissa voi edelleen olla tekstikenttiä jne). MVC:ssä oleellista ei

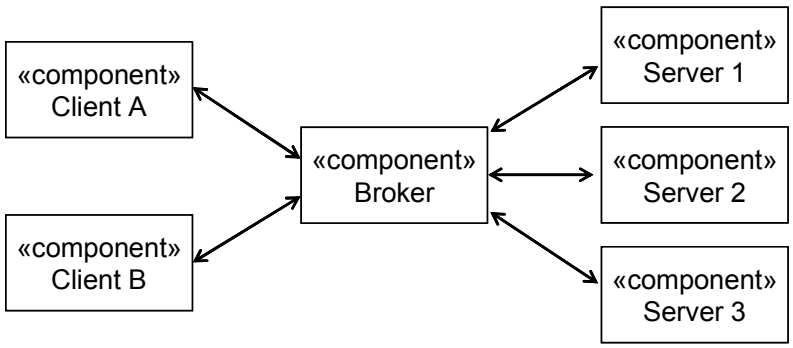
ole täsmällinen luokkajako, vaan kohdealueen tietojen suoran muokkauksen mahdollistaminen. Esimerkiksi, jos malli ja sen käyttöliittymä ovat riittävän yksinkertaisia tai esim. ei tarvita useita käyttöliittymiä, ohjain ja näkymä (esim. ikkunan menut) tai jopa malli (esim. ikkunan vierityspalkki) voidaan yhdistää samoihin luokkiin[30].



Kuva 5.20: Kampanjan muokkaus MVC-mallissa

Broker (välittäjä) (kuva 5.21; Bennett 2006)

Välittäjä on yksi Buschmannin ym. (1996) esittämistä arkkitehtuurimalleista. Mallia käytetään hajautetuissa järjestelmissä, joissa asikkaat ja palvelimet halutaan erottaa toisistaan (esim. tilanteessa, jossa palvelinten kuormitusta halutaan jakaa, palvelimen sijainti halutaan peittää asiakkaalta tai osapuolet toimivat erilaisissa verkoissa). Välitinkomponentti vastaa asiakkailta tulevien viestien välityksestä oikealle palvelimelle ja mahdollisesti viestimuodon muuttamisesta. Välittimen kutsurajapinta on sama kuin palvelimilla, joten asiakkaiden näkökulmasta välitinkomponentilla tai palvelimilla ei ole eroa. Välittäjä-arkkitehtuurimalli on käytössä mm. CORBA (Common Object Request Broker Architecture)-pohjaisissa järjestelmissä.



Kuva 5.21: Välittäjä-arkkitehtuurimalli

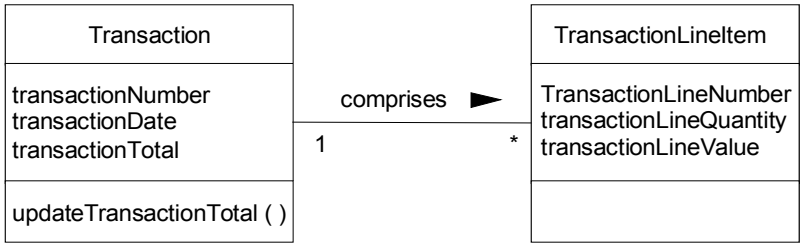
[30] T. Reenskaug (2003). The Model-View-Controller (MVC) - Its Past and Present. http://folk.uio.no/trygver/2003/javazone-jaoo/MVC_pattern.pdf

5.2.1.3 Analyysi- ja suunnittelumalleja

Analyysimallit ovat suunnittelumallien kaltaisia ratkaisuja kohdealuemallinnuksen yhteydessä usein esiintyviin mallinnusongelmiin. Analyysimallit ovat tyypillisesti varsinaisia suunnittelumalleja yksinkertaisempia ja vähemmän toteutukseen tai toteutuskieleen sidonnaisia.

Transaction-Transaction Line Item (Coad 1995)

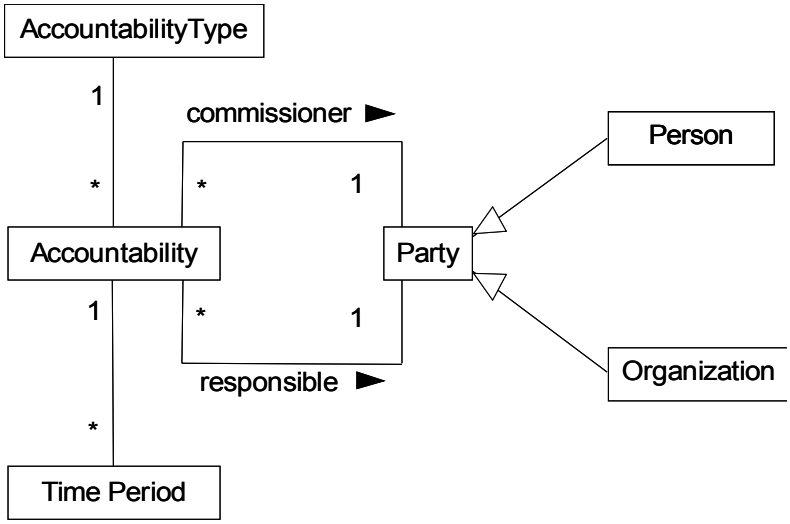
Esimerkki yksinkertaisesta analyysimallista (kuva 5.22; Bennett 2006). Sen avulla voidaan mallintaa tapahtumia (transaktioita), joiden suoritukseen liittyy useita (mahdollisesti hinnoiteltuja) yksiköitä. Tyypillinen esimerkki mallin soveltamisesta on verkkokaupan ostoskori, mutta vastaava rakenne ilmenee kaikenlaisten tilaus- ja toimituslistojen yhteydessä.



Kuva 5.22: Transaction-Tranaction Line Item -analyysimallin esitys luokkakaaviona

Vastuuvelvollisuus (Accountability)-malli (Fowler 1996; kuva 5.23; Bennett 2006)

Esimerkki monipuolisemmasta analyysivaiheen mallista. Mallilla kuvataan vastuun määrityksiä organisaatiossa: tietty osapuoli (henkilö tai organisaatio) voi määrätä toisen osapuolen vastuulliseksi tietystä asiasta tietylle ajankohdalle.



Kuva 5.23: Vastuuvelvollisuus -analyysimallin esitys luokkakaaviona

Agate-mainostoimistoesimerkkiin sovellettuna Vastuuvelvollisuus-mallin toimeksiantaja on asiakas ja vastuullinen henkilö on henkilökunnan jäsen. Varsinainen vastuu on asiakkaan kontaktihenkilönä toimiminen kampanjan kulun ajan.

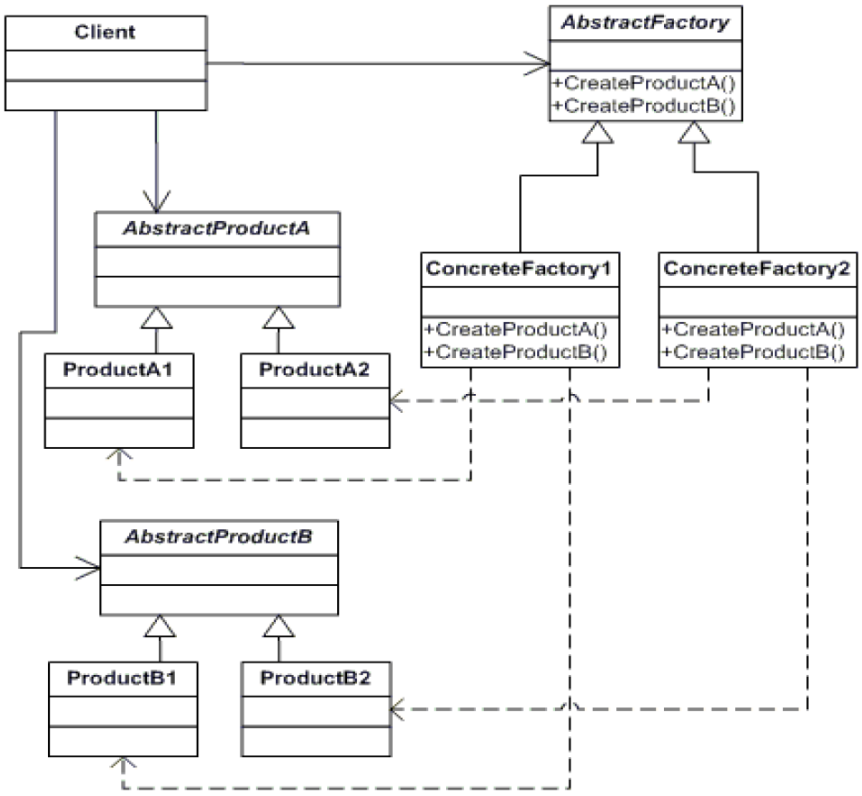
Suunnittelumallit ovat analyysimalleja spesifimpiä kuvauksia, joiden soveltaminen liittyy arkkitehtuurisuunnittelun ja yksityiskohtaisen suunnittelun välimaastoon. Gamman ym. (1995) esityksessä mallit on jaettu *luontimalleihin* (creational patterns), *rakennemalleihin* (structural patterns) ja *käyttäytymismalleihin* (behavioral patterns)[31].

Esimerkkinä luontimallista kerrotaan seuraavassa lähemmin *Abstrakti tehdas* (Abstract Factory*)*-mallista (Koskimies, 1994). Mallia on havainnollistettu kuvassa 5.24.

Tarkoitus	Malli tarjoaa liittymän, jonka avulla voidaan luoda tietyn luokan aliluokkien ilmentymiä tuntematta aliluokkia.
Motivointi	Oletetaan, että halutaan toteuttaa käyttöliittymien tekemiseen tarkoitettu työkalu, joka tukee useita ikkunointityyppejä (esim. Motif, Mac). Eri ikkunointityypeissä on esim. erilainen ulkoasu vierityspalkeille, painikkeille ja ikkunoille. Jotta sovellus olisi helposti siirrettävissä ympäristöstä toiseen, sen koodissa ei saisi viitata tietyn standardin mukaisiin käyttöliittymän osiin. Ongelma voidaan ratkaista antamalla abstrakti luokka AbstractFactory, joka määrittelee luontioperaation vierityspalkille, painikkeelle jne. Toisaalta jokaiselle näistä käyttöliittymän osista on oma abstrakti luokka, jonka aliluokkina olevat konkreettiset luokat määrittelevät osan tietyn standardin mukaisena. Myös AbstractFactory:llä on oma konkreettinen aliluokka kutakin standardia kohden. Sovellukset kutsuvat abstraktin AbstractFactory-luokan operaatioita käyttöliittymän osien luomiseksi tietämättä, minkä konkreettisen luokan mukaisia ne tulevat olemaan. Ratkaisu sitoo yhteen tietyn standardin mukaiset konkreettiset käyttöliittymäosat: sovellus voi käyttää vain niitä osia, joiden luontioperaatio on annettu sovelluksen tuntemassa AbstractFactory:n aliluokassa.
Soveltuvuus	Abstract Factory -mallia voidaan käyttää, kun ● järjestelmän tulisi olla riippumaton siitä, miten sen tuotteita luodaan, kootaan ja esitetään, ● järjestelmän tulisi olla konfiguroitavissa tietylle tuoteperheelle ● järjestelmän tulisi taata, että tietyn tuoteperheen jäseniä käytetään yhdessä ● halutaan esittää erilaiset järjestelmän tuotteet luokkakirjastona ja halutaan paljastaa vain niiden abstraktit liittymät, ei toteutustapaa.

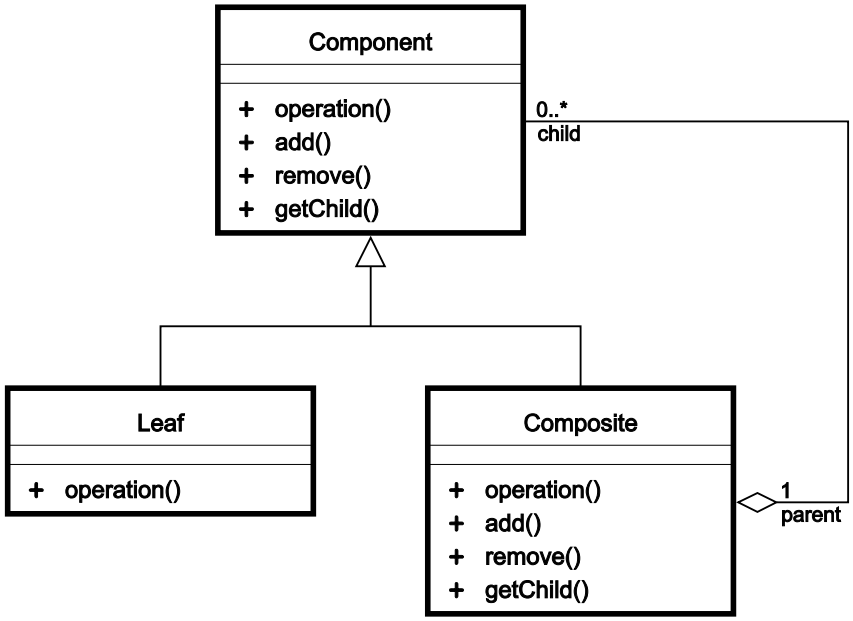
Rakenne-, Osallistujat- ja Seuraukset- kohdat tulisi lisäksi kuvata (ei ole sisällytetty tähän esitykseen).

[31] Tiivis yleiskatsaus GoF-suunnittelumalleihin: J. Itkonen (2005), Suunnittelumallit (”GoF-referaatti”), http://users.jyu.fi/~ji/opetus/oa2005/luennot/oa2005_dp.pdf



Kuva 5.24: Abstrakti tehdas -mallin esitys luokkakaaviona.

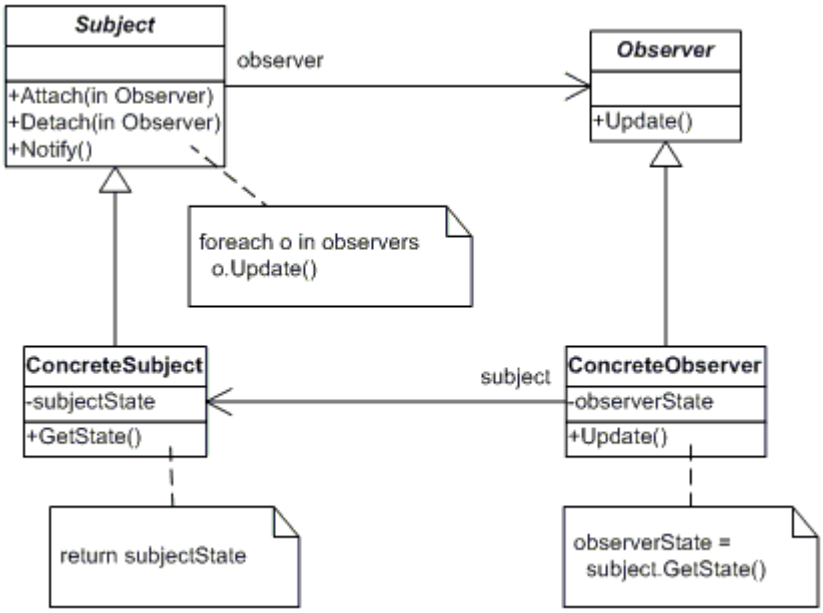
Esimerkkinä rakennemallista käsitellään *Rekursiokooste*-mallia (composite). Malli soveltuu tilanteeseen, jossa oliot ja niiden osaoliot muodostavat hierarkian ja on tarvetta käsitellä kaikkia puussa olevia olioita ja osaolioita samalla tavalla esimerkiksi tietorakenteen läpikäynnin yhteydessä.



Kuva 5.25: Rekursiokoosteen yleinen esitys luokkakaaviona

Kuvassa 5.25 on esitetty mallia kuvaava luokkakaavio[32]. Lehtiluokka (Leaf) tarkoittaa hierarkian päässä olevia olioita, joilla ei ole omia osaolioita. Koosteluokka (Composite) kuvaa hierarkian huipulla ja välisolmuissa olevia olioita, jotka omistavat aina muita osaolioita. Sekä kooste- että lehtisolmut on peritty Component-luokasta, joka tarjoaa hierarkiassa oleville olioille yhteiset toiminnot (erityisesti *operation*-metodi – muut metodit liittyvät osaolioiden lisäykseen ja poistoon). Operation-metodi toteutetaan koosteluokassa niin, että varsinaisen toiminnan suorittamisen lisäksi kutsutaan kaikkien osaolioiden operation-metodia. Lehtiluokassa operation-metodilla ei ole erityistä lisätoiminnallisuutta.

Esimerkkinä käyttäytymismallista käsitellään *Tarkkailija*[33] (observer) (kuva 5.26). Tarkkailija-malli soveltuu tilanteeseen, jossa yksi tai useampi olio (Observer) tarvitsee tietoa Subject-olion tilan muuttumisesta ilman, että Subject-olion tarvitsee tietää Observerin yksityiskohdista.



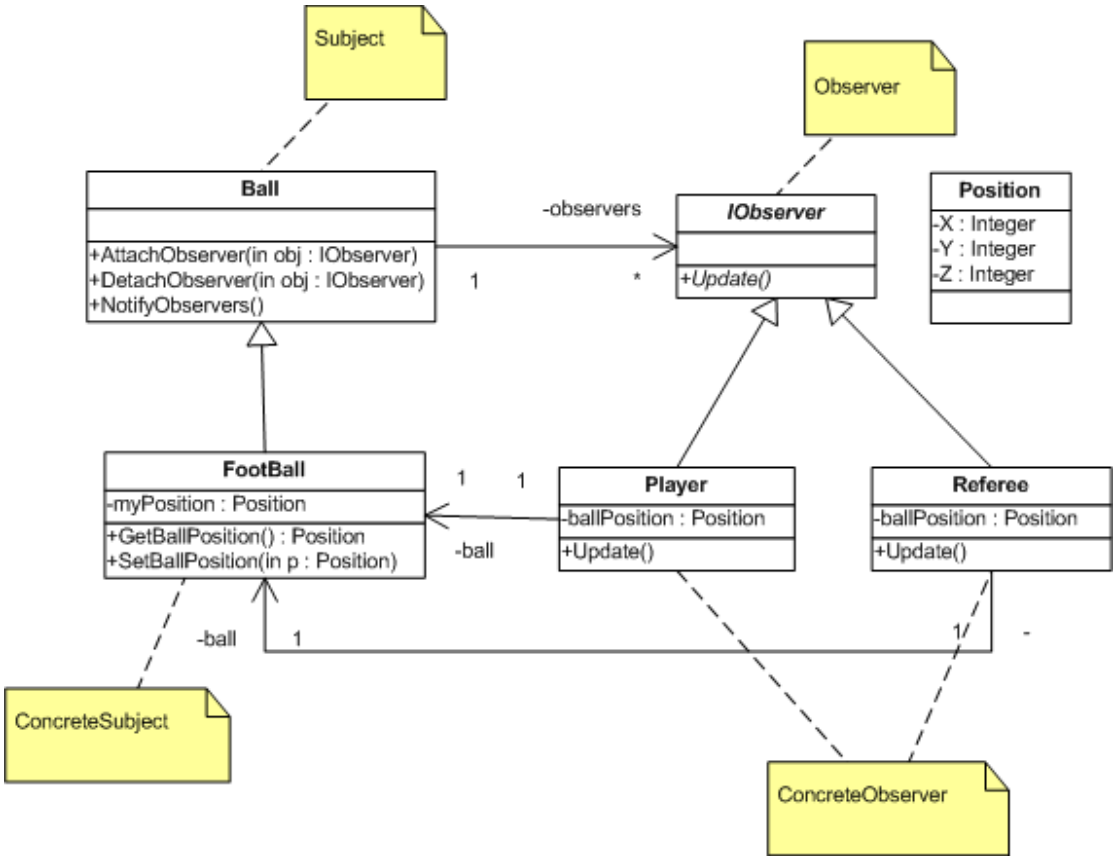
Kuva 5.26: Tarkkailija-malli

Malli toteutetaan samaan tapaan kuin näkymä- ja malliolioiden yhteys MVC-arkkitehtuurimallissa: Subject-olioille määritellään rekisteröitymismekanismi (*attach*- ja *detach*-metodit) sekä *notify*-metodi, joka kutsuu kaikkien rekisteröityneiden olioiden (tarkkailijoiden) *update*-metodia. Update-metodin toteutus Observerissa päivittää tarkkailijaan liittyvät tiedot, mm. kysymällä Subject-olion nykytilan.

Kuva 5.27 havainnollistaa tarkkailijamallin käyttöä jalkapallopelin olioilla: pelaajat ja tuomarit ovat tarkkailijoita, joille pallo-olio ilmoittaa tilansa muutoksista.

[32] Kuvan lähde: http://en.wikipedia.org/wiki/Composite_pattern

[33] Kuvat ja esimerkki: An 'OOP' Madhusudanan (2005). Design Your Soccer Engine, and Learn How To Apply Design Patterns (Observer, Decorator, Strategy and Builder Patterns) - Part I and II http://www.codeproject.com/KB/architecture/applyingpatterns.aspx



Kuva 5.27: Tarkkailija-malli sovellettuna jalkapallon pelilogiikkaan

5.2.2 Sovelluskehykset

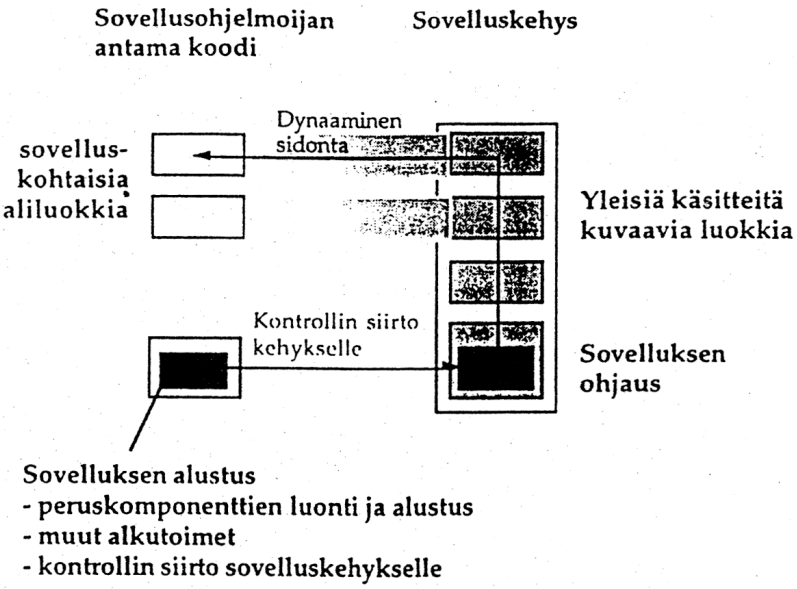
Suunnittelun yhteydessä käsitellyt mallit tarjoavat suuntaviivoja ja parhaimmillaankin yleisiä rakenteita tietojärjestelmien kehittämiseen. Mutta aiemmissa kehittämisprojekteissa on voitu tuottaa paljon sellaisia luokkamäärityksiä, joita ainakin periaatteessa voitaisiin myöhemmin uudelleenkäyttää samankaltaisissa kehittämisprojekteissa. Teknisten olioluokkien (erit. käyttöliittymäluokkien) osalta luokkakirjastoja on rakennettu ja yleisesti markkinoitu jo vuosia. Käytännössä on osoittautunut kuitenkin hankalaksi löytää juuri sopivia luokkia ja istuttaa niitä kohtuullisella vaivalla uuteen ympäristöön. Erityisesti toiminnallisten suhteiden määrittely siirrännäisten ja muiden luokkien välillä on koettu ongelmalliseksi.

Sovelluskehyksellä (application framework) tarkoitetaan sovellussuuntautunutta luokkakokoelmaa, josta sopivasti täydentämällä saadaan valmis järjestelmä tai sen merkittävä osa (Koskimies, 1994). Yleensä sovelluskehys muodostuu abstrakteista luokista (ja mahdollisesti joistakin konkreeteista luokista), joihin sovellussuunnittelija kytkee konkreettisia aliluokkia ja muodostaa näiden luokkien ilmentymistä tarvitsemiaan rakenteita.

Hollywood -periaate (inversion of control): *don't call us, we call you*

- sovelluskehys sisältää järjestelmän peruslogiikan, joka vastaa sovittamisen jälkeenkin järjestelmän muiden osien ohjauksesta (Kuva 5.28; Koskimies, 1996),

- tavanomaisessa uudelleenkäytössä puolestaan ohjelmoijan tekemä koodi kutsuu uudelleenkäytettävää koodia.



Kuva 5.28: Sovelluskehyksen rakenne.

Tällä hetkellä valmiita kehyksiä on tarjolla mm.:

- graafisiin käyttöliittymiin (GUI, graphical user interface): esim. MacApp, Visual Component Library (Borland), Java AWT/SWT/Swing, Windows Presentation Foundation, NeXTStep, wxWidgets
- web-sovelluksiin (esim. Ruby on Rails, Spring, Zend, Django)
- skaalautuvien ja hajautettujen yrityssovellusten kehittämiseen (esim. Java EE- ja CORBA-pohjaiset kehykset)
- CAD-sovellusten kehittämiseen
- käyttöjärjestelmien ja verkon hallintajärjestelmien kehittämiseen

Suurin osa kehyksistä on sovellusalueriippumattomia (eivät sisällä liiketoimintaluokkia): OMG:n piirissä on viime vuosien aikana ollut toiminnassa useita sovellusaluekohtaisia työryhmiä (DTF, Domain Task Force), jotka ovat työstäneet kehyksiä mm. rahoitukseen, elektroniseen liiketoimintaan, terveydenhoitoon, logistiikkaan ja tietoliikenteeseen. Ehkä merkittävin voima viime vuosina tapahtuneen sovelluskehysten määrän ja laadun nopean kasvun takana on tullut avoimeen lähdekoodiin perustuvia kehyksiä kehittäneiltä lukuisilta projekteilta. Toisaalta myös suuret ohjelmistotalot kuten Microsoft ovat panostaneet kehyksiin. Sovelluskehitystyön tuottavuus onkin lähtenyt selvään nousuun mm. web-palveluiden kehittämisen alueella.

Kehyksen suunnittelu muodostaa aivan oman ongelma-alueensa. Vain osa järjestelmän suunnittelusta voidaan hyödyntää siinä. Seuraavat periaatteet ja toimintatavat liittyvät kehyksen suunnitteluun:

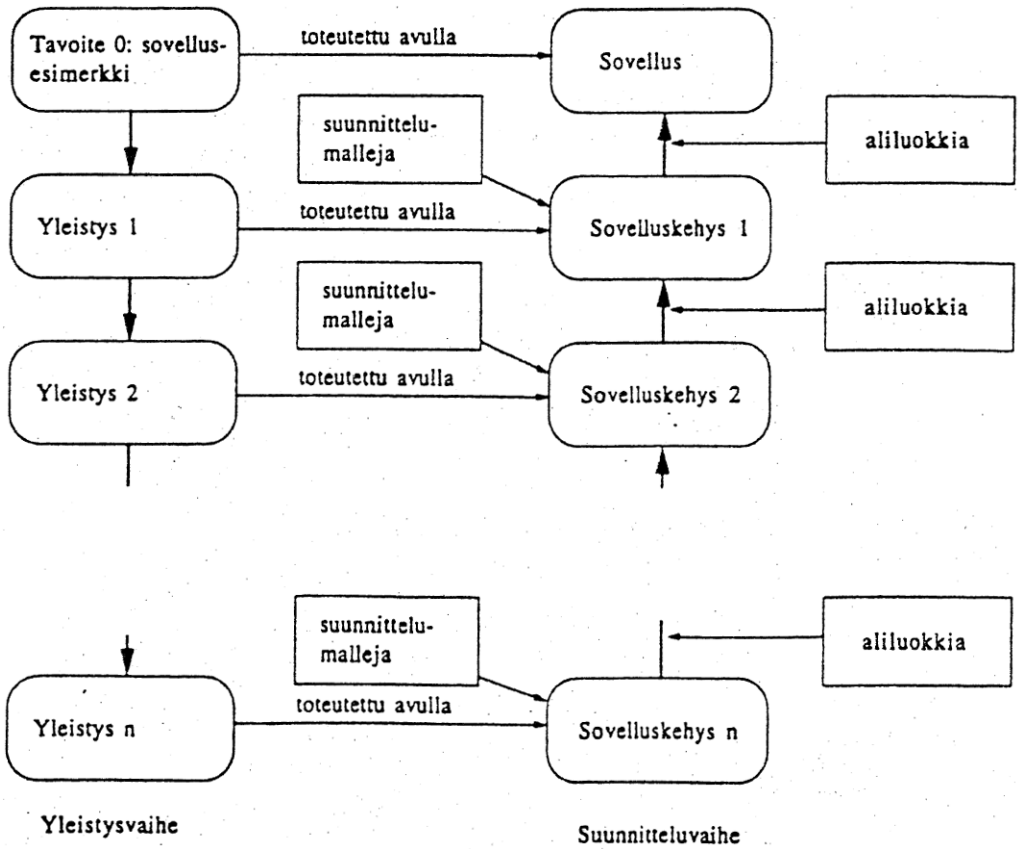
- Kehys on tarkoitettu periaatteessa äärettömän monen järjestelmän toteutukseen.
- Kehyksen käytön kannalta on olennaista, että se soveltuu myös tapauksiin, joihin sen suunnittelija ei ole etukäteen varautunut.
- Kehyksen suunnittelu etenee iteratiivisesti:

1. aloitetaan tekemällä yksittäisiä järjestelmiä yhdellä sovellusalueella,
2. irroitetaan näiden yhteisiä osia kehyksen ensimmäiseksi versioksi
3. kokeillaan kehystä muihin järjestelmiin ja täydennetään sitä näiden vaatimusten mukaisesti

- Olennaista on kiinnittää huomiota myös "kuumiin kohtiin" (hot spots): ne ovat osia, jotka voivat vaihdella eri järjestelmissä. Nämä muodostavat kehyksen rajapintoja, joihin tulee ripustaa järjestelmästä riippuvia koodiosia tai luokkia.

Seuraavassa on kuvattu kehyksen suunnittelun kulkua Koskimiehen (1994) mukaisesti (huom. järjestelmää kutsutaan sovellukseksi) (kuva 5.29): Suunnittelu aloitetaan spesifioimalla jokin tyypillinen sovellus, joka toivotaan voitavan toteuttaa sovelluskehyksen avulla. Tätä spesifiointia ryhdytään sitten asteittain yleistämään niin, että päästään lopulta mahdollisimman yleiseen muotoon, joka kuitenkin vielä sanoo jotain oleellista sovellusalueesta. Tätä kutsutaan *yleistysvaiheeks*i. Se johtaa peräkkäisiin, asteittain yleistyviin tehtäväkuvauksiin.

Yleistysvaiheen jälkeen siirrytään *suunnitteluvaiheeseen*. Siinä aloitetaan yleistysvaiheen viimeisestä tehtäväkuvauksesta ja pyritään antamaan sille toteutus sovelluskehyksenä. Toteutuksen apuna voidaan käyttää suunnittelumalleja. Käyttämällä hyväksi tätä sovelluskehystä pyritään sitten toteuttamaan seuraavaksi yleisin tehtäväkuvaus, Näin jatkaen saadaan lopulta toteutus alkuperäiselle sovellusesimerkille. Haluttu sovelluskehys on tätä edeltävän tehtäväkuvauksen toteuttava sovelluskehys.



Kuva 5.29: Sovelluskehyksen kehitysprosessi

Kirjallisuudessa on esitetty erityisiä ohjeistoja, nk. keittokirjoja (cook books, receipts), jotka ohjaavat kehysten käyttöä (Pree, 1995). Viime vuosina markkinoille on tullut useita sekä kaupallisia että avoimeen lähdekoodiin perustuvia yleiskäyttöisiä välineitä kehysten suunnitteluun, hallintaan ja hyödyntämiseen (esim. Eclipse).

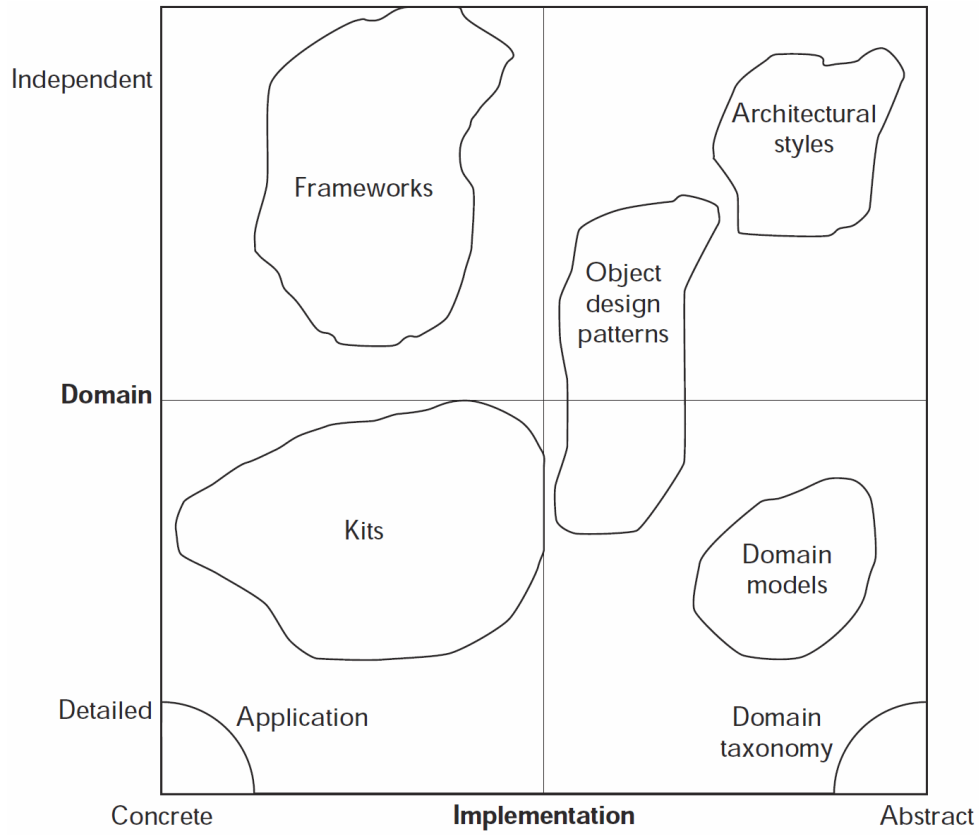
Eroja kehyksen ja mallin välillä:

- kehykset ovat laajempia kuin mallit; kehys voi koostua useammasta mallista
- kehykset ovat konkreettisia luokkamäärityksiä, kun taas mallit ovat dokumentoitu- ja suunnitteluideoita
- mallit ovat toteutusriippumattomia, kun taas kehykset ovat ainakin ohjelmointikielen osalta kiinnitettyjä.

Kuvassa 5.30 on esitetty Tempfenhartin ja Cusickin (1997) jäsennys, jolla osoitetaan suunnittelumallien ja kehysten rooli järjestelmänkehitykselle. Jäsennyksen ulottuvuudet ovat:

- sovellusalueriippuva – sovellusalueriippumaton (Domain)
- toteutusriippuva – toteutusriippumaton (Implementation)

Suunnittelumallien ja kehysten lisäksi jäsennykseen sisältyvät myös arkkitehtuurimallit, abstraktit sovellusmallit ja sovelluskohtaiset työkalupaketit (kits).



Kuva 5.30: Suunnittelumallien, sovelluskehysten ja muiden suunnittelua helpottavien tekniikoiden jäsennys sovellusaluekeskeisyyden ja konkreettisuuden mukaan.

5.3 Yksityiskohtainen suunnittelu (oliosuunnittelu)

Suunnittelu on edennyt tähän mennessä tarkentuneeseen arkkitehtuurirakenteeseen, jonka avulla voidaan osoittaa, mihin pakettiin ja mahdollisesti mihin komponenttiin kukin luokka kuuluu. Luokkien attribuutit ja assosiaatiot on määritelty. Sen sijaan useimpien luokkien operaatiot ovat vielä selvittämättä. Keskeisimpänä työstämisen kohteena on tässäkin vaiheessa luokkakaavio. Oliosuunnittelun tarkoituksena on

- operaatioiden löytäminen, muodostaminen ja sijoittelu luokkiin
- ohjausrakenteen ja algoritmien suunnittelu
- luokkarakenteen muokkaaminen periytymistä hyödyntäen
- assosiaatioiden toteutuksen suunnittelu
- olioiden esittämistavasta päättäminen

Oliosuunnittelu etenee iteratiivisesti, jolloin kierros kierrokselta suunnitellaan ja päätetään yhä yksityiskohtaisemmista asioista. Siten yllä olevia tehtäviä käydään läpi tilannekohtaisesti useampaan kertaan samanaikaisesti dynaamisten mallien kanssa. Ratkaisuissa otetaan huomioon tehokkuuden (mm. muistitila, toteutusaika), ylläpidettävyyden ja laajennettavuuden kriteerit.

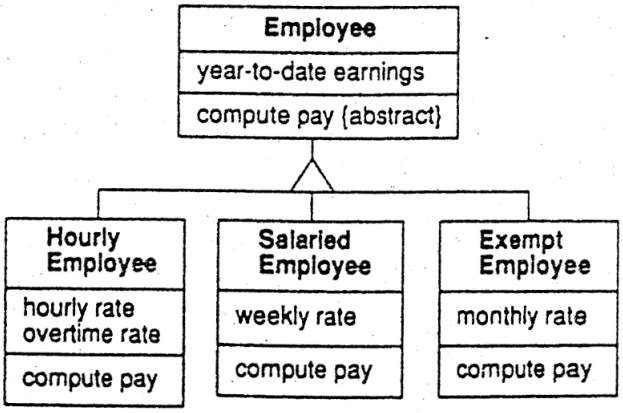
5.3.1 Luokkakaavioiden tarkentaminen

Luokkakaavioiden tarkentamisen lähtökohtana ovat analyysivaiheen mallit ja CRC-kortit. Suunnitteluvaiheessa luokkakaaviota tarkennetaan iteratiivisesti.

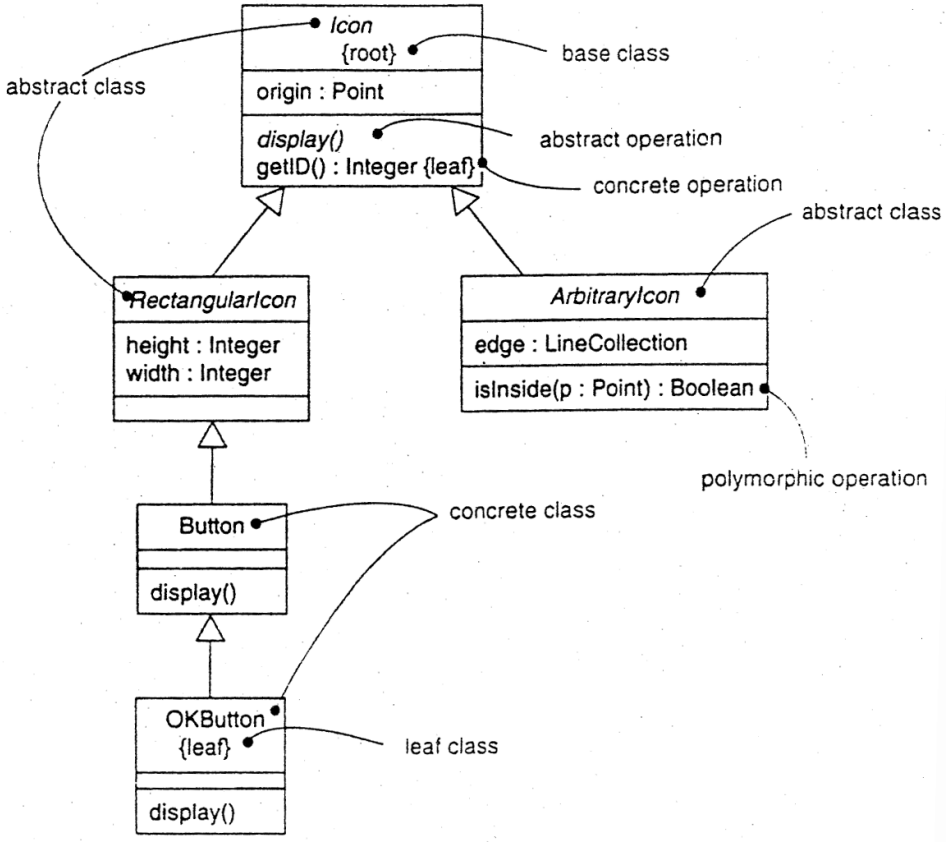
5.3.1.1 Luokkakaavio: erityiskäsitteitä

Luvussa 4.2.1 esiteltyjen yleiskäsitteiden ja -rakenteiden lisäksi luokkakaavion esittämisessä voidaan käyttää seuraavia erityiskäsitteitä: abstrakti/konkreetti luokka, attribuuttien ja operaatioiden näkyvyys sekä navigointi.

Tähän mennessä luokkia on pääosin käsitelty vain yleisesti. On olemassa kuitenkin kahdenlaisia luokkia, abstrakteja ja konkreetteja luokkia (kuvat 5.32 ja 5.31). Rajapinnat ovat erikoistapaus abstrakteista luokista.



Kuva 5.31: Abstraktit ja konkreetit luokat ja operaatiot.

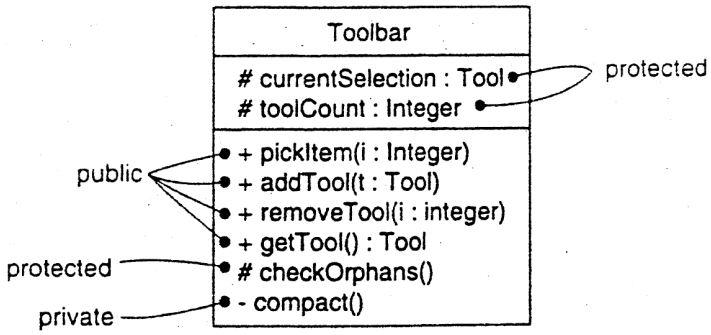


Kuva 5.32: Abstraktit ja konkreetit luokat ja operaatiot.

- *Abstrakti* luokka:
  - ◊ on luokka, jolla ei ole suoranaisia olioilmentymiä,
  - ◊ esiintyy luokkahierarkian yläosassa, ei koskaan lehtisolmuna,
  - ◊ käytetään kokoamaan konkreettien luokkien yhteisiä ominaisuuksia,
  - ◊ sisältää yhden tai useampia abstrakteja operaatioita, jolle annetaan konkreetti merkitys konkreetin luokan yhteydessä (esim. compute pay). Näin konkreetit luokat voivat toteuttaa metodin eri tavoin.
  - ◊ Jos abstrakti luokka ei sisällä lainkaan attribuutteja eikä toteutettuja metodia, sitä kutsutaan (varsinkin Java-terminologiassa) *rajapinnaksi* (interface)
- *Konkreetti* luokka:
  - ◊ on luokka, jolla voi olla olioilmentymiä (instantiable),
  - ◊ esiintyy luokkahierarkiassa lehtisolmuna tai välisolmuna.

Tähän asti olemme painottaneet sitä, että kaikki olion tieto on saatavissa esille vain ko. luokan operaatioiden kautta. Seuraavaksi kerrotaan mahdollisuuksista poiketa tästä periaatteesta. Luokan attribuuttien ja operaatioiden yhteydessä voidaan ilmaista, miten *näkyviä* (visible) ne ovat toisten luokkien olioille; tarkemmin ottaen voidaanko ko. attribuutteja/operaatioita käyttää toisten luokkien operaatioiden määrittelyssä. Seuraavat vaihtoehdot ovat tarjolla (kuva 5.33; Booch ym.,1999, 123):

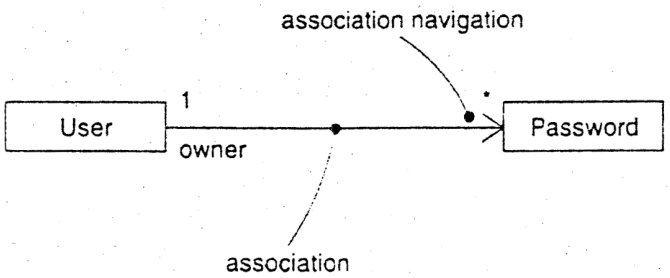
- yleinen (+): attribuutti tai operaatio on muiden luokkien (olioiden) käytettävissä,
- yksityinen (-): attribuutti tai operaatio ei ole muiden luokkien (olioiden) käytettävissä,
- suojattu (#): attribuuttia tai operaatiota voivat käyttää vain ko. luokan aliluokkien oliot.



Kuva 5.33: Luokan olioiden näkyvyys muille olioille.

Yleensä kaikki attribuutit tulisi määrittää yksityisiksi, ellei kyse ole vakiosta. Jos erityismerkintöjä ei ole käytetty, kannattaa olettaa attribuuttien olevan yksityisiä ja metodien julkisia (luokkien yksityisten apumetodien mallintaminen UML:n tasolla ei ole muutenkaan tarkoituksenmukaista edes suunnitteluvaiheessa).

Assosiaatio kahden luokan välillä määrittää rakenteellisen yhteyden ko. luokkien olioiden välille. Tämä ”viestintäkäytävä” mahdollistaa ko. luokkien olioiden välisen viestinnän. Oletusarvoisesti assosiaatio mahdollistaa suoran viestinnän molempiin suuntiin. Jos suunnittelun aikana haluamme määrittää tarkemmin, onko viestintäyhteys molempiin suuntiin vai vain toiseen suuntaan, voimme tehdä sen "avoimella" nuolenpäällä (association navigation). Esimerkiksi jos salasanaa etsitään pelkästään Käyttäjä-olion kautta, voidaan navigointisuunta määrittää kuvan 5.34 (Booch ym, 1999, 144) kaltaisesti. Yksisuuntaisilla assosiaatioilla voidaan vähentää luokkien välisiä riippuvuuksia ja näin yksinkertaistaa esim. muistinhallintaa kielillä, joissa ei ole automaattista muistinsiivousta.



Kuva 5.34: Viestintäkäytävä ja –suunta olioiden välillä.

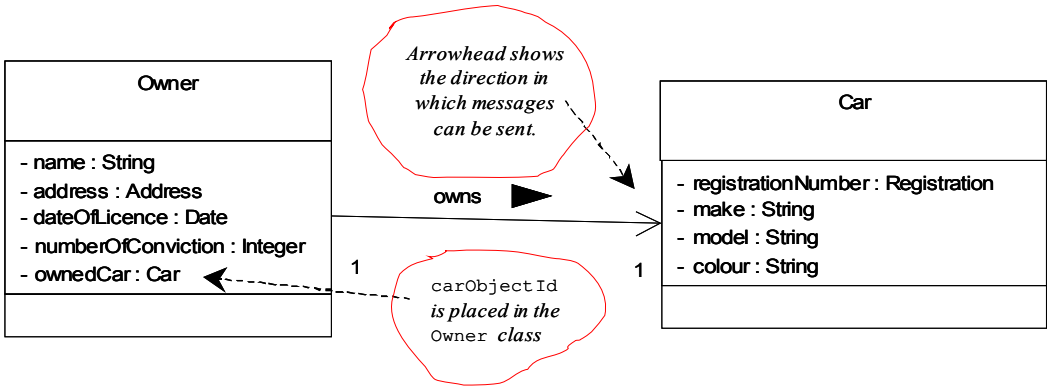
5.3.1.2 Assosiaatioiden toteutuksesta päättäminen

Assosiaatiot osoittavat polut, joita "pitkin" viestejä voidaan lähettää. Tässä vaiheessa tehdään periaatepäätös, minkälaisina polut toteutetaan. Ratkaisun tekeminen edellyttää, että tiedetään, millä tavalla luokkien mukaisten olioiden halutaan viestivän keskenään. Viesti voi saavuttaa vastaanottavan olion vain, jos sen viite tunnetaan. *Viite* (reference) eli identiteetti on pysyvä tunnus, jonka järjestelmä antaa oliolle sen syntymisen yhteydessä. Koko viestimisen perusedellytys on se, että lähettävä olio A tuntee vastaanottavan olion B viitteen. Tähän on kaksi mahdollisuutta:

- joku olio C "kertoo" lähettävälle oliolle A olion B viitteen esimerkiksi lähettämällä A:lle viestin, jonka parametrina on kyseinen viite.
- A:llä on viitearvoinen attribuutti, johon B:n viite on pysyvästi tallennettu. Viitearvoinen attribuutti voidaan toteuttaa esimerkiksi C++ -kielessä viitemuuttujalla (reference data member) tai osoitinmuuttujalla (pointer data member). Javassa oliomuuttujat ovat aina viitearvoisia.

Tarkastellaan viitearvioisten attribuuttien käyttöä assosiaatioiden toteuttamisessa.

Yhden_suhde_yhteen -assosiaatio tarkoittaa sitä, että luokan A mukainen olio voi olla kytketty linkillä vain yhteen luokan B mukaiseen olioon ja päinvastoin (esim. *owns* -assosiaatio: kuva 5.35; Bennett ym, 2006)



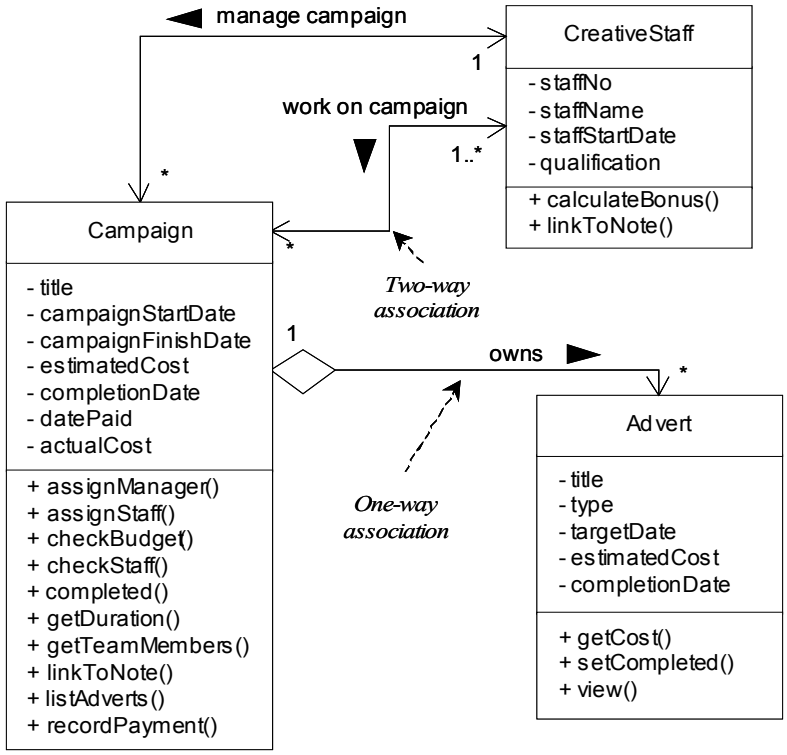
Kuva 5.35: Esimerkki yksisuuntaisesta 1-1 viestinnästä assosiaatiossa.

- yksisuuntainen viestintä:
  - ◊ jos viestinnän on tarkoitus tapahtua vain toiseen suuntaan, määritellään lähettävälle luokalle viitearvoinen attribuutti (carObjectId), joka sisältää vastaanottavan olion viitteen (kuva 5.35).
- kaksisuuntainen viestintä:
  - ◊ Jos viestinnän on tarkoitus tapahtua molempiin suuntiin, määritellään viiteattribuutti molempiin luokkiin.
- Viestintäsuunnan osoittamiseksi käytetään erityistä navigointiassosiaatiota.

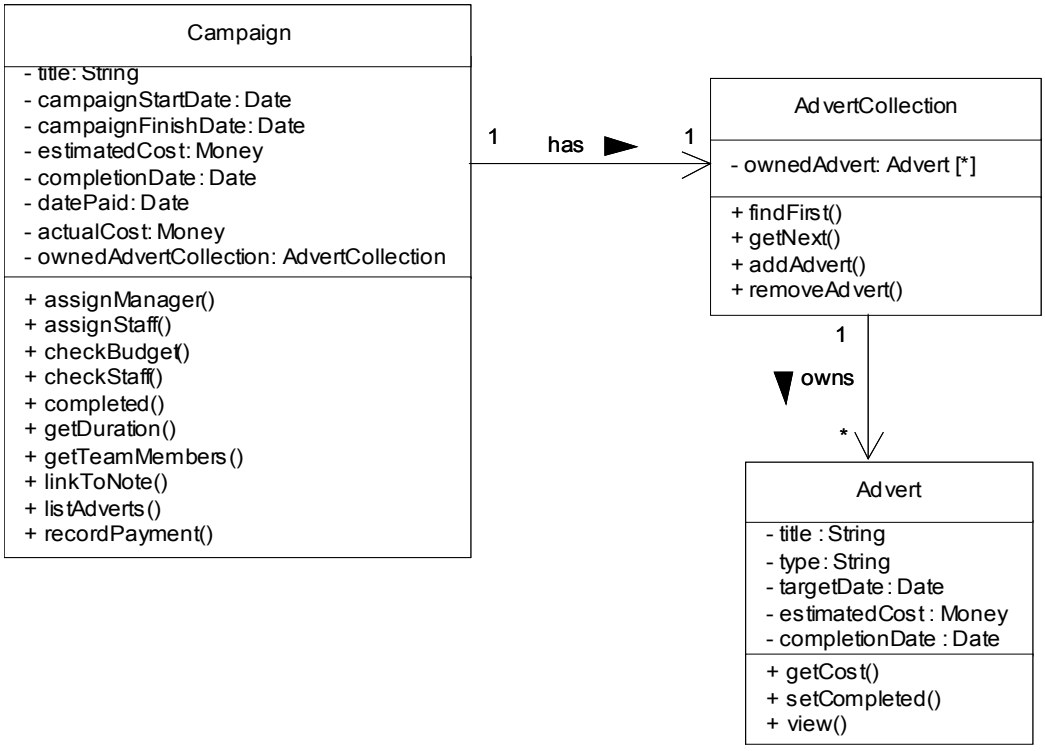
Yhden_suhde_moneen -assosiaatio tarkoittaa sitä, että luokan A mukainen olio voi olla kytketty linkeillä useampaan luokan B mukaiseen olioon, mutta B mukainen olio voi olla kytketty vain yhteen luokan A olioon (esim. *owns* -kooste, kuva 5.36).

- Yksisuuntainen viestintä:
  - ◊ Tarkastellaan kuvan 5.36 esimerkkiä. Oletetaan, että Campaign -olio haluaa lähettää viestin jokaiselle Advert -oliolle, joka on koostesuhteessa siihen. Tämä voidaan toteuttaa siten, että määritellään taulukkotyyppinen viitearvoinen attribuutti. Parempi ratkaisu on kuitenkin erillinen "säiliö"luokka (Collection, toteutus esim. listana, joukkona tai säiliöluokan sisäisenä taulukkona), jonka mukaiseen olioon tallennetaan ko. viitteet (kuva 5.37). Tämän AdvertCollection -olion viite on puolestaan tallennettu Campaign -olioon viitearvoiseksi attribuutiksi (advertCollectionId). Tällöin viestintä

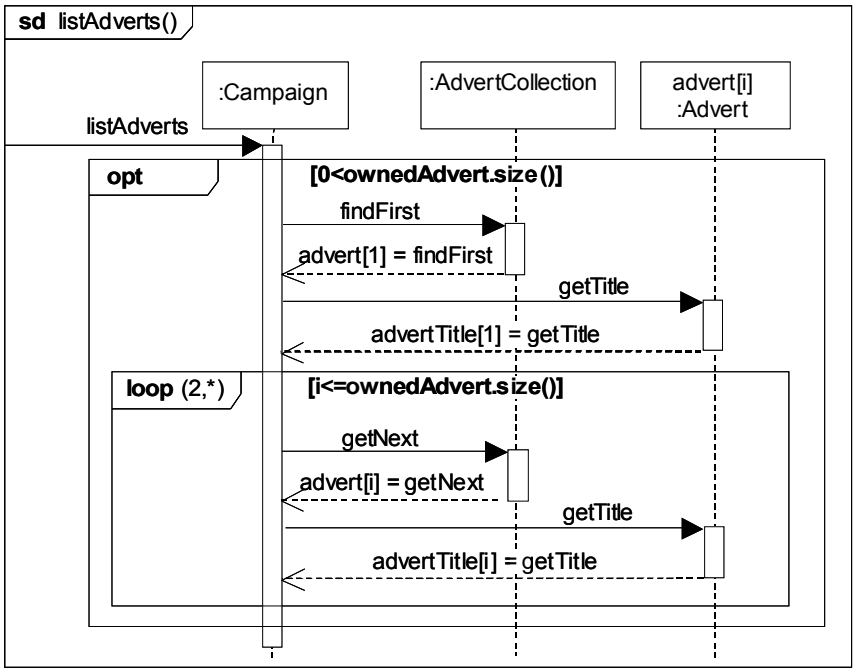
voi tapahtua esimerkiksi seuraavasti (kuva 5.38; Bennett ym. 2006): Campaign -olio lähettää ensin viestin AdvertCollection -oliolle ensimmäisen Advert -olion viitteen saamiseksi. Tämän viitteen avulla Campaign -olio voi viestiä suoraan Advert- oliolle. Tämän jälkeen Campaign -olio pyytää AdvetCollection -oliolta toisen viiteen, jonka avulla se lähettää viestin, jne.



Kuva 5.36: Esimerkki yksisuuntaisesta 1- viestinnästä assosiaatiossa (owns) sekä kaksisuuntaisesta monesta-moneen -suhteesta (works on campaign).*



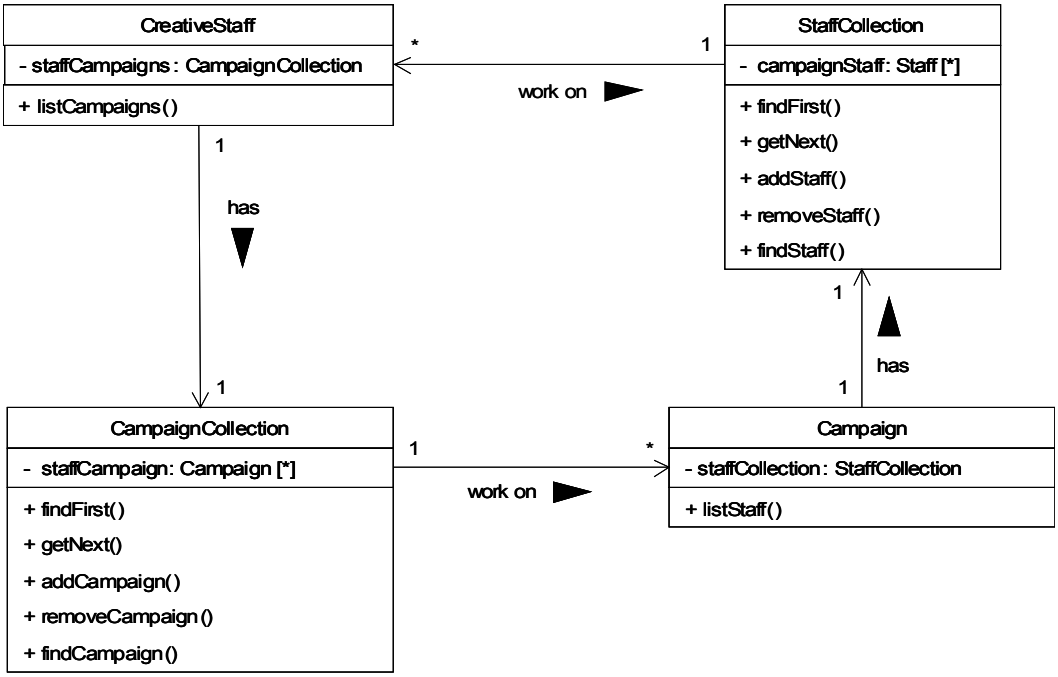
Kuva 5.37: Esimerkki 1- viestinnästä assosiaatiossa säiliöluokan avulla.*



Kuva 5.38: Sekvenssikaavio kampanjan kaikkien mainosten otsikoiden hakuun.

- Kaksisuuntainen viestintä:
  - ◊ Määritellään edellisen lisäksi viitearvoinen attribuutti "yhden" -puoleiselle luokalle (esim. kuvan 5.36 Advert-luokasta Campaign-luokkaan).

Monen_suhde_moneen -assosiaatio tarkoittaa sitä, että luokan A mukainen olio voi olla kytketty linkeillä useampaan luokan B mukaiseen olioon ja B mukainen olio voi olla kytketty useampaan luokan A olioon (esim. worksOnCampaign, kuva 5.35).



Kuva 5.39: Kaksisuuntaisen assosiaation toteutus säiliöluokilla (vrt. kuva 5.35)

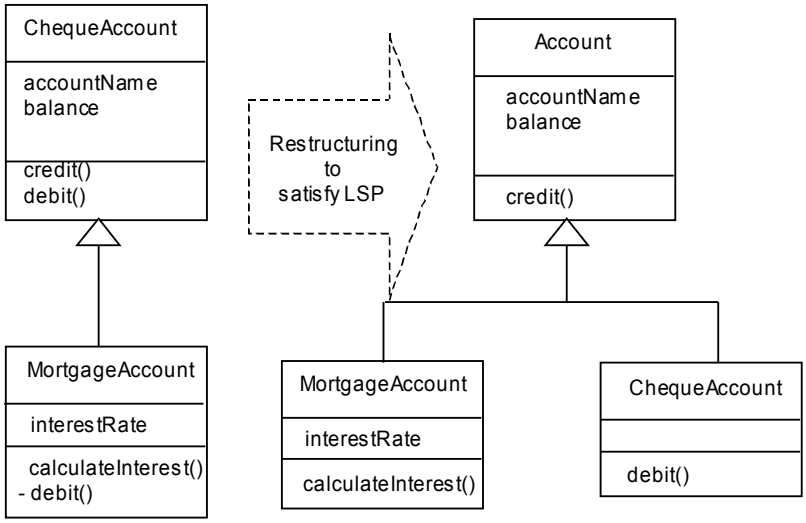
- Yksisuuntainen viestintä:
  - ◊ Kuten edellisessä tapauksessa,
- Kaksisuuntainen viestintä:
  - ◊ Tarkastellaan kuvan 5.35 esimerkkiä, jossa CreativeStaff -luokan ja Campaign -luokan välille on piirretty kaksisuuntainen assosiaatio. Se voidaan toteuttaa kahdella säiliöluokalla (kuva 5.39; Bennett ym. 2006).

Lopullinen päätös teknisestä toteutustavasta riippuu käytettävästä oliokielestä.

5.3.1.3 Luokkakaavion muokkaaminen periytymistä hyödyntäen

Luokkakaavioon tehtyjen lisäysten ja muutosten jälkeen on syytä jälleen tutkia, voidaanko luokkarakennetta muokata periytymistä hyödyntäen. Seuraavat tehtävät kuuluvat tähän askeleeseen:

- Joskus samoja tai samantapaisia operaatioita on sisällytetty useampaan luokkaan. Määrittelyn yksinkertaistamiseksi on hyödyllistä tutkia, voidaanko operaatioita periyttää, vaikka sitten joitakin operaatioita (niiden kutsumuotoa (signature) tai sisäistä rakennetta) hieman muuttaen.
- Kokonaisuuden tarkastelu voi johtaa uusien luokkien ja operaatioiden lisäämiseen. Edelleen luokkia voidaan erikoistaa yleistyshierarkian mukaisesti. Yliluokat voivat tällöin olla abstrakteja luokkia. Yliluokan määrittely voi olla mahdollista, vaikka sillä olisi vain yksi aliluokka, jos ratkaisu yleisemmin nähdään tarpeelliseksi. Tällainen yleinen syy voi olla sellaisen uudelleenkäytettävän luokan tekeminen, jota tiedetään/oletetaan tarvittavan jossakin muussa yhteydessä. Abstraktit yliluokat voivat parantaa myös laajennettavuutta. Esimerkiksi Sensori, joka on tarkoitus hieman erityyppisenä ottaa myöhemmin käyttöön toisella tehtaalla.



Kuva 5.40: Kuvan vasemmassa osassa oleva rakenne ei nodata korvausperiaatetta, koska lainatililtä ei voi veloittaa. Tilanne on ratkaistu määrittelemällä tileille uusi, yhteinen yliluokka, jonka rajapintaa on rajoitettu (veloitusmetodi poistettu), josta lainatili ja shekkitili on peritty.

Luokkahierarkiaa suunnitellessa perittyjen luokkien tulisi noudattaa *Liskovin korvausperiaatetta* (substitution principle): aliluokan olion pitäisi olla aina järkevästi käsitel-

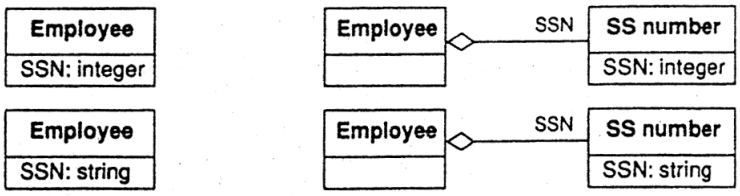
tävissä myös yliluokan oliona (=korvattavissa yliluokan metodeilla). Esimerkki korvausperiaatteen soveltamisesta on kuvassa 5.40. Luokkien rakenteellisten ominaisuuksien lisäksi myös metodien polymorfinen syrjäyttäminen (virtuaalifunktiot) voi rikkoa periaatetta, jos alaluokassa oleva metoditoteutus on ristiriidassa yliluokan toteutuksen kanssa.

Tunnettu esimerkki korvausperiaatteen rikkomisesta on ns. Ympyrä-Ellipsi (variantti: Neliö-Suorakulmio) -ongelma[34]. Ympyrä on ellipsin erikoistapaus, mutta oliosuunnittelun kannalta ympyrä ei ole ellipsi, koska ellipsin rajapinta määrittää sille erillisen pituuden ja leveyden. Ympyrän toteutuksessa on mahdollista syrjäyttää pituuden ja leveyden asetusmetodit (esim. niin, että asetettaessa pituus asetetaan samalla myös leveys), mutta tällöin olio ei enää käyttäydy ellipsin määrittelemän rajapinnan mukaan (ellipsejä käsittelevissä metodeissa ei voida olettaa, että pituuden muuttaminen vaihtaa myös leveyttä) – korvausperiaatetta on siis rikottu.

5.3.1.4 Olioiden esittämistavasta päättäminen

Luokkia voidaan määritellä toisten luokkien avulla tai perustietotyyppien avulla (kielestä riipuen Integer, Char, String ym. - kokonaisluvut ja merkit ovat perustietotyyppejä, Javassa käsiteltävissä myös luokkina; merkkijono on C++:ssa ja Javassa luokka, Object Pascalissa perustyyppi). C++:ssa ja Object Pascalissa luokat ja tietueet on selkeästi eroteltu toisistaan, Smalltalkissa kaikki ovat olioita perustyypit mukaan lukien. Päivämääriä ja aikoja (Date, Datetime, Timestamp), valuuttoja (Currency) ja lueteltuja tyyppejä (enum) voidaan usein käsitellä kuten perustyyppejä ja vähintään näiden olemassaolo voidaan olettaa UML-malleissa (tarkat toteutustavat vaihtelevat eri kielillä). Suunnitteluvaiheen aikana joudutaan ratkaisemaan mm.

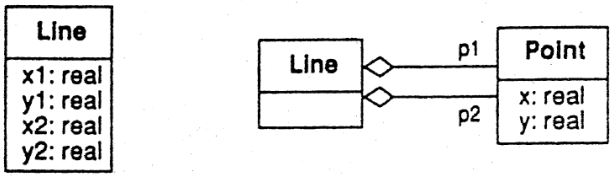
- Missä tapauksessa käytetään luokkia ja milloin määrittely tehdään suoraan perustietotyyppien avulla (vrt. Employee ja SS number) (kuva 5.41)



Kuva 5.41: Luokkien ja perustietotyyppien hyödyntäminen luokkien määrittelyssä. Kokonaisluvut ja merkkijonot tulkitaan tässä perustietotyyppeinä.

- Missä tapauksessa olioiden suhteet toteutetaan yhdellä tai useammalla luokalla (esim. kuvataanko viivan päätepisteet suoraan lukuina olion sisään vai erillisinä pisteoliona). Koostamismekanismi vaihtelee hieman eri kielissä – esim. Javassa olioita voidaan koostaa vain viitteillä, C++:ssä voidaan valita osoittimen, muuttumattoman viitteen tai aidon koostamisen välillä. (kuva 5.42)

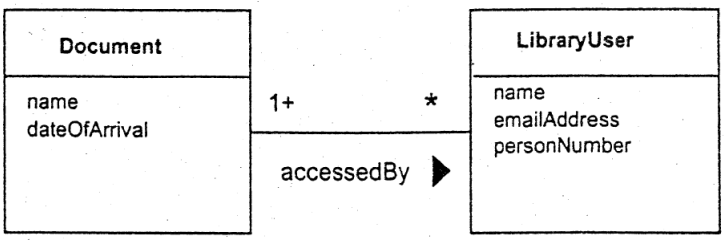
[34] http://www.parashift.com/c++-faq-lite/proper-inheritance.html#faq-21.6



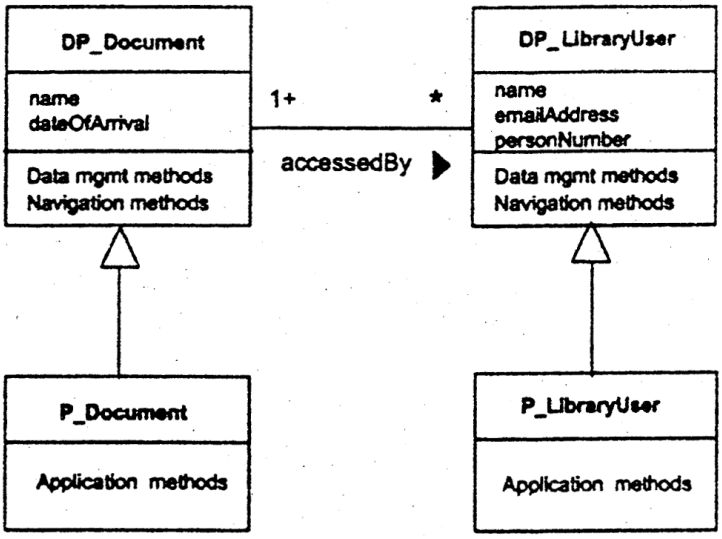
Kuva 5.42: Olioiden suhteiden toteuttaminen implisiittisesti ja eksplisiittisesti.

- Missä tapauksessa kannattaa jakaa luokkia osiin, jotta voidaan tuottaa uudelleenkäytettäviä komponentteja.

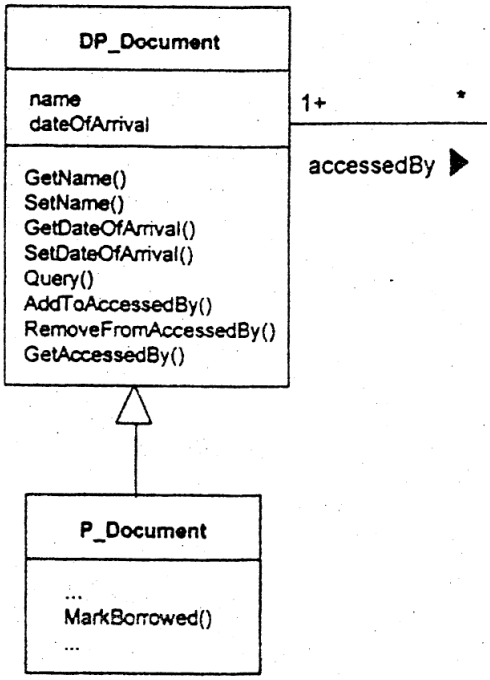
Viimeksi mainittuun tapaukseen liittyy Jaaksin ym. (1999) esittämä esimerkki (katso myös luku 5.3.4.2): Oletetaan, että dokumenttikanta toteutetaan tietokantana. Osa alkuperäisestä luokkakaaviosta on esitetty kuvassa 5.43. Jos haluamme standardoida tietokantaan sijoitettavien tietojen haun, määrittelemme kullekin liiketoimintaluokalle yleiskäyttöiset yliluokat (DP-luokat). Näihin voidaan sijoittaa perustoimintoja suorittavia operaatioita (*getName*, *setName* ym). Näiden aliluokiksi määritellään luokkia, joiden yhteydessä voidaan antaa sovelluskohtaisempia operaatioita (esim. *markBorrowed*) (kuvat 5.44 ja 5.45).



Kuva 5.43: Osa dokumenttitietokannan luokkakaaviosta.



Kuva 5.44: Dokumenttitietokannan DP-yliluokat ja sovelluskohtaiset aliluokat.



Kuva 5.45: Yliluokan ja aliluokan operaatioita dokumenttitietokannassa.

5.3.2 Ohjausrakenteiden tarkentaminen

Ohjausrakenteiden suunnittelu täydentää luvussa 4.3 kuvattua dynaamista mallintamista.

5.3.2.1 Tilakaaviot

Tilakaavio (state diagram) kuvaa yhden olion osalta käyttäytymistä: sen eri tiloja ja tilojen välisiä siirtymiä. Seuraavassa esitellään ensin kaavion keskeiset käsitteet ja sitten notaatio.

Tilakaaviot muistuttavat paljon aktiviteettikaavioita (luku 3.3). Erona näiden välillä on se, että tilakaavio keskittyy prosessin aikana tapahtuvaan olion sisäiseen toimintaan, kun taas toimintakaavio keskittyy prosessin toimintojen ohjaukseen ja kulkuun. Toimintakaaviosta nähdään miten toiminnot riippuvat toisistaan, missä järjestyksessä ne suoritetaan, mitä tietoa toiminnot tuottavat ja mitkä toiminnot tätä tietoa käyttävät, ja mikä on prosessin logiikka. Toimintakaavio on tilakaavion erikoistapaus, sillä toimintakaavio on abstrahoitu tilakaavio. Toimintakaaviosta on jätetty pois esimerkiksi tilakaavioissa esiintyvien herätteiden nimet, jotta looginen toiminnallisuus saataisiin paremmin näkyviin.

Heräte (event):

- on tapahtuma, jolla voi olla seurauksia jonkin olion tilaan; se voi olla viestin saapuminen, olion jonkin (attribuutin) arvon muuttuminen tai esimerkiksi määräajan täyttyminen;
- on hetkellinen, siihen ei liity kestoa,

Tila (state):

- määrittelee aikavälin, jolloin (a) olion tila pysyy samanlaisena, (b) olio odottaa seuraavaa herätettä ja/tai (c) olio on suorittamassa jotakin toimintaa (esim. Henkilö voi olla naimaton, naimisissa, eronnut tai leski)
- tilalla on kesto,
- voi tarkoittaa jatkuvanluonteista toimintaa (esim. puhuminen puhelimessa), tai hallinnollisten lakien (esim. naimisissa) tai luonnonlakien mukaista olotilaa (esim. elohopea on juoksevaa); joskus tilat erottuvat attribuuttiarvojen mukaan (esim. vaihteen asento)
- voidaan mallintaa eri tarkkuustasoilla (granularity)

Tilasiirtymä (transition)

- kuvaa siirtymän olion kahden tilan välillä;
- aiheutuu herätteestä ja saa olion toimimaan tavalla, joka muuttaa tilaa lähtötilasta (source state) maalitilaan (target state),

Ehto (guard condition):

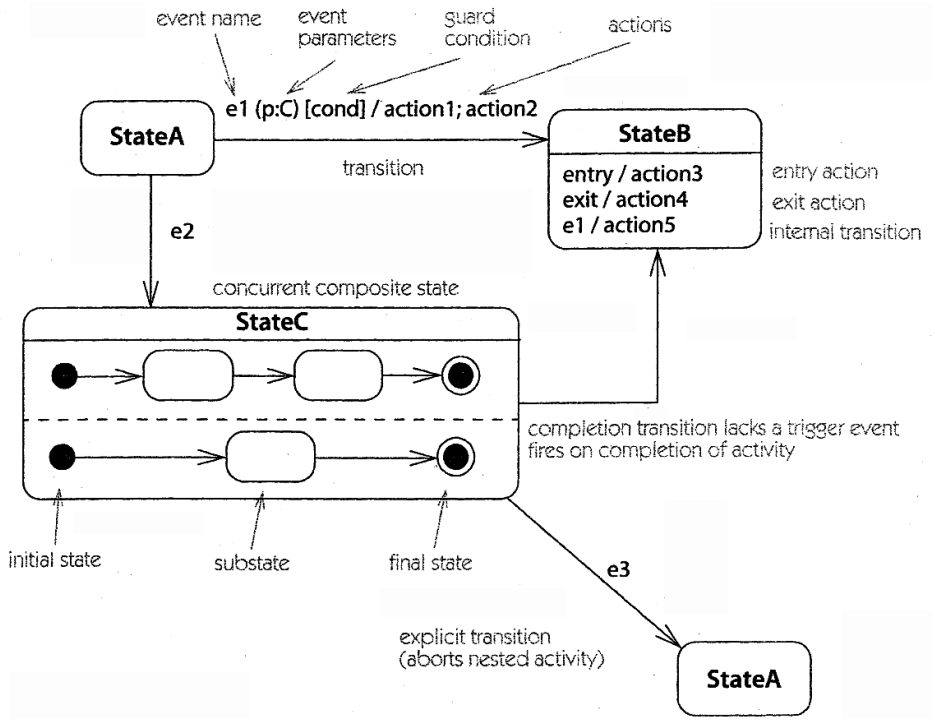
- voidaan käyttää lisämääreenä herätteen yhteydessä ilmaisemaan, missä olosuhteissa tilasiirtymä on mahdollinen
- ilmaistaan usein Boolean (loogisella) -lausekkeella, joka on rakennettu olion attribuuttiarvoista (esim. lämpötila on alle pakkasrajan helmikuussa 1996)
- tarkistetaan vasta, kun heräte on tapahtunut (esim. Hissi lähtee liikkeelle (heräte: VarattuNäppäin_painettu) vain mikäli ehto ‘Hissi_ei_ole_käytössä’ on tosi)

*Toiminto (*action):

- tilasiirtymän yhteydessä olio voi suorittaa jonkin toiminnon (esim. *tilillePano()*);
- samoin tilan yhteydessä voidaan määrittää (a) tilaantulotoimintoja (entry action)), (b) tilastalähtötoimintoja (exit action) ja (c) tilan sisäisiä toimintoja (internal activity) (esim. yritettäessä soittaa ja linjan ollessa varattu puhelin alkaa tuottaa ‘varattu’-ääntä)

Tilakaavio (state diagram) (kuva 5.46; Rumbaugh ym. 1999, s.72, kuva 5.47; Rumbaugh ym, 1999, s. 72):

- kuvaa tietyn olion käyttäytymistä tuomalla esiin kaikki kiinnostavat tilat ja olion mahdollisuudet siirtyä tilasta toiseen,
- notaatio:
  - ◊ suunnattu graafi, joka koostuu solmuista (tilat) ja nuolista (tilasiirtymät)
  - ◊ solmuina esiintyvät tilat kuvataan suorakaiteilla, joilla on pyöristetyt kulmat. Symbolin sisällä ilmaistaan tilanimi ja mahdolliset toiminnot,
  - ◊ tilasiirtymät esitetään nuolina, joiden yhteydessä ilmaistaan vastaavan herätteen nimi, mahdollinen ehto (hakasuluissa) sekä toiminnot (muodossa /toiminto)
  - ◊ elinkaaren alku (olion luonti) esitetään pisteellä ja loppu “häränsilmällä”.
- voi sisältää yhden tai useampia silmukoita (loops)

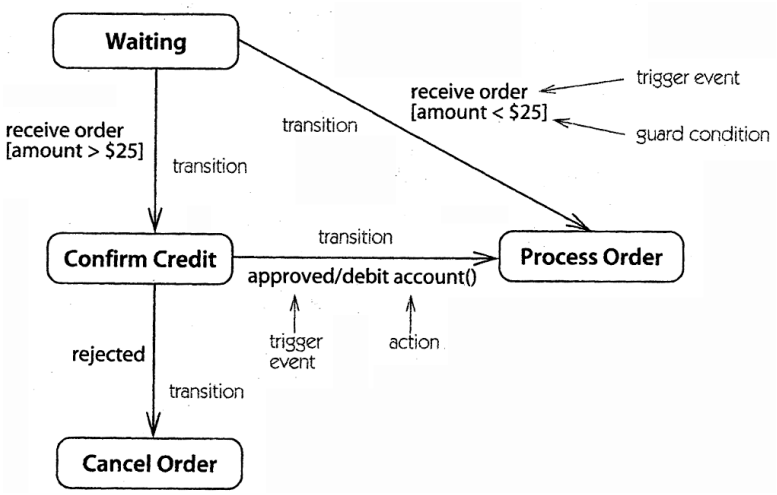


Kuva 5.46: Tilakaavio

Tilakaavioiden laadinnan tarkoituksena on määrittää keskeisten olioiden osalta niiden käyttäytyminen siten, että tilat, tilasiirtymät sekä herätteet (ts. elinjakso) ovat nähtävissä yhden olioiden osalta kerrallaan (vrt. Liitteen 2 kuvat B.15-B.17, Rumbaugh ym, 1992):

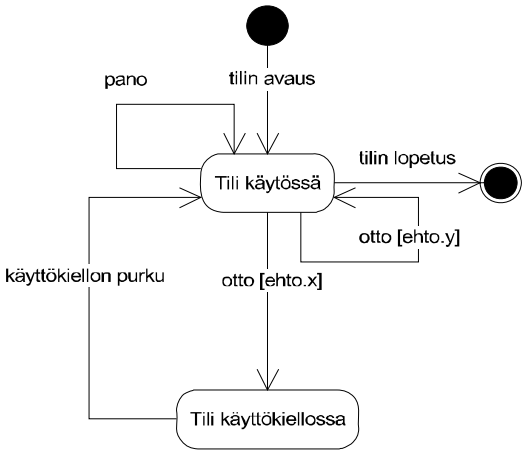
- valitse luokka, jonka olioiden käyttäytymisen tarkempi selvittäminen on tarpeen
- etsi sekvenssikaavio tai yhteistoimintakaavio, jonka viestit vaikuttavat ko. luokan olioihin
- poimi ne kaikki tulevat ja lähtevät viestit, jotka vaikuttavat ko. luokan olioihin.
- harkitse, onko luontevaa pitää kahden peräkkäisen saapuvan viestin väliä erillisenä tilana; jos on, tee väleistä tilakaavion tiloja ja liitä viestit vastaavien tilasiirtymien herätteiksi,
- nimeä tila, jos löytyy luonteva nimi; muuten jätä nimeämättä
- jos toimintasarjaa on tarkoitus toistaa loputtomasti, esitä se tilakaaviossa silmukkana. Muussa tapauksessa esitä elinkaaren alku ja loppu.
- valitse seuraava sekvenssikaavio tai yhteistoimintakaavio, jonka viestit vaikuttavat ko. luokan olioihin ja suorita edelliset askeleet sille. Mieti tarkkaan, milloin on kysymyksessä uusi tila ja milloin jo tilakaaviossa esiintyvä tila, johon tulee tai josta lähtee erilaisia viestejä (polkuja). Jälkimmäisessä tapauksessa järjestelmän ei tarvitse muistaa, minkä herätteen seurauksena tilaan on tultu.
- tarkastele kaavion johdonmukaisuutta: Ovatko kaikissa silmukoissa mahdollisesti tarvittavat lopetusehdot? Ovatko kaikki tilat aidosti erilaisia? Onko tilojen välillä mahdollisesti muita herätteitä? Jos on, täydennä vastaavia vuorovaikutuskaavioita.
- jaa kaavio osiin, jos siitä tulee liian monimutkainen (esim. normaalit tilat, poikkeustilat); toinen mahdollisuus on käyttää yleistysrakennetta,
- valitse seuraava keskeinen olioluokka. Toista sille edelliset askeleet.

Huom. Tilakaaviota ei ole tarkoitus tehdä jokaisen olioluokan olioille.

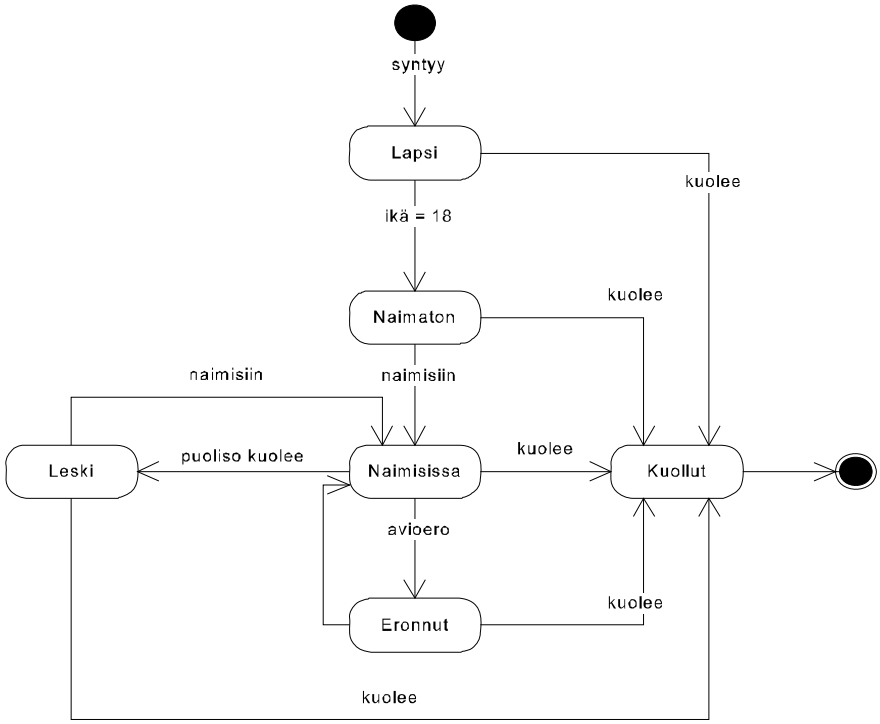


Kuva 5.47: Tilakaavio tilauksen käsittelylle. Tilauksen käsittelyjärjestelmä odottaa tilausta. Kun tilaus otetaan vastaan, se käsittelee tilauksen, mikäli tilauksen arvo on alle 25 dollaria. Yli 25 dollarin tilauksissa se varmistaa ensin luoton. Mikäli luottoa on tarpeeksi, se käsittelee tilauksen. Muussa tapauksessa se peruu tilauksen.

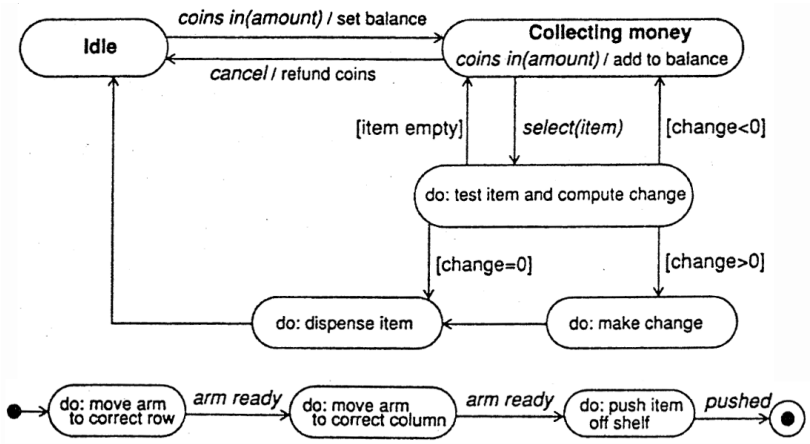
Tilakaavion käyttöä dynaamisessa mallintamisessa on havainnollistettu vielä seuraavilla kuvilla: tili (kuva 5.48), henkilön siviilisääty (kuva 5.49), juoma-automaatti (kuva 5.50), mikroaaltouuni (kuva 5.51) ja auto (kuva 5.52).



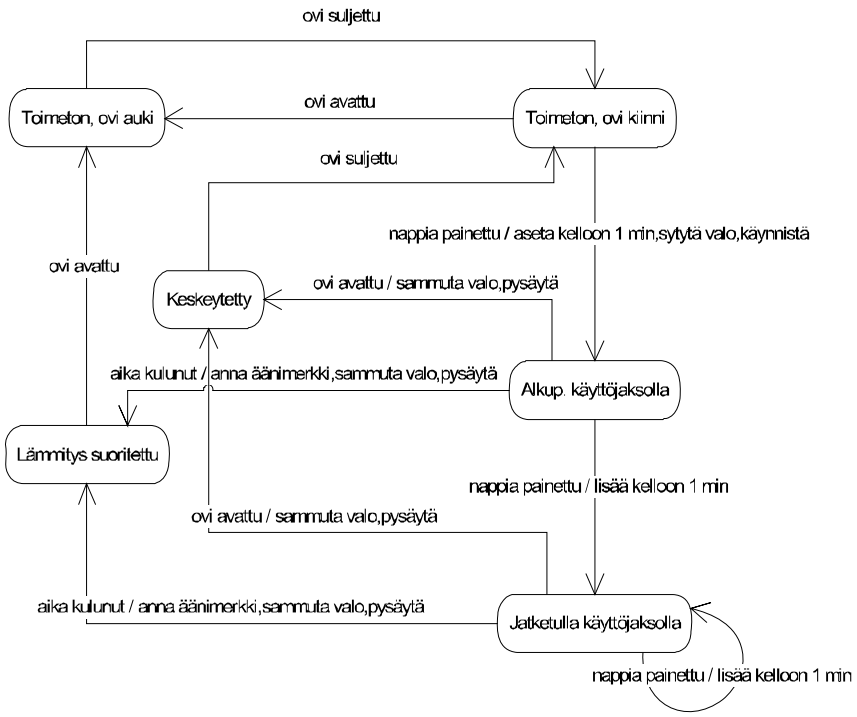
Kuva 5.48: Tilin tilat elinkaarensa aikana. Henkilö voi avata tilin, jolloin hän saa tilin käyttöönsä. Tilille voi panna rahaa ja sieltä voi nostaa sitä. Jos rahaa nostetaan normaalisti, ongelmia ei esiinny. Mikäli tililtä kuitenkin nostetaan yli nostorajan, tili joutuu käyttökieltoon. Asiakas voi purkaa käyttökiellon sopimalla siitä pankin kanssa. Näin tili on jälleen käytössä, kunnes asiakas päättää sen lopettaa.



Kuva 5.49: Henkilön siviilisääty elinkaarensa aikana. Kun henkilö syntyy, hän on lapsi ensimmäiset 18 elinvuottaan. Normaalitapauksessa lapsi saavuttaa 18 vuoden iän ja hänet merkitään naimattomaksi. Henkilö saa uuden statuksen "Naimisissa", kun hän astuu avioon. Naimisissa oleva henkilö voi erota tai hänen puolisonsa voi kuolla. Eronnut tai leskeksi jäänyt henkilö voi mennä uudelleen naimisiin tai elää elämänsä loppuun eronneena tai leskenä. Henkilö voi kuolla milloin tahansa.



Kuva 5.50: Juoma-automaatin tilakaavio. Automaatti odottaa asiakkaan kolikkoa. Kun kolikko syötetään, automaatti antaa valita tuotteen. Mikäli asiakas valitsee tuotteen, jota ei ole saatavana tai jonka maksamiseksi annettu raha ei riitä, automaatti palaa tuotteen valintaan. Asiakas valitsee uuden tuotteen tai lisää rahaa. Kun asiakas on valinnut tuotteen, joka on saatavana ja johon rahamäärä on riittävä, automaatti palauttaa ylimääräiset rahat, jos niitä on. Sen jälkeen automaatti antaa ostetun tuotteen. Asiakas voi myös keskeyttää ostonsa, jolloin automaatti palauttaa asiakkaan syöttämät rahat. Kuvan alareunassa tilakaavio "dispense item" -toiminnolle.



Kuva 5.51: Mikroaaltouuni. Alkutilassa mikrooouuni on toimeton ja ovi on kiinni. Oven avaamisen jälkeen uuni pysyy edelleen toimettomana. Mikro käynnistetään nappia painamalla (oven pitää olla kiinni), jolloin mikroon syttyy valo ja asetetaan minuutin käyttöaika. Aikaa saadaan minuutti lisää jokaisella uudella painalluksella. Jos ovi avataan kun aikaa on vielä jäljellä, lämmitys keskeytetään, valo sammutetaan ja aika nollataan. Mikrouuni palaa toimettomaksi, kun ovi suljetaan. Normaalitapauksessa ajan annetaan kulua loppuun, jonka jälkeen mikrouuni antaa äänimerkin, uuni pysäytetään ja valo sammutetaan. Kun ovi avataan, mikrouuni palaa toimettomaksi.

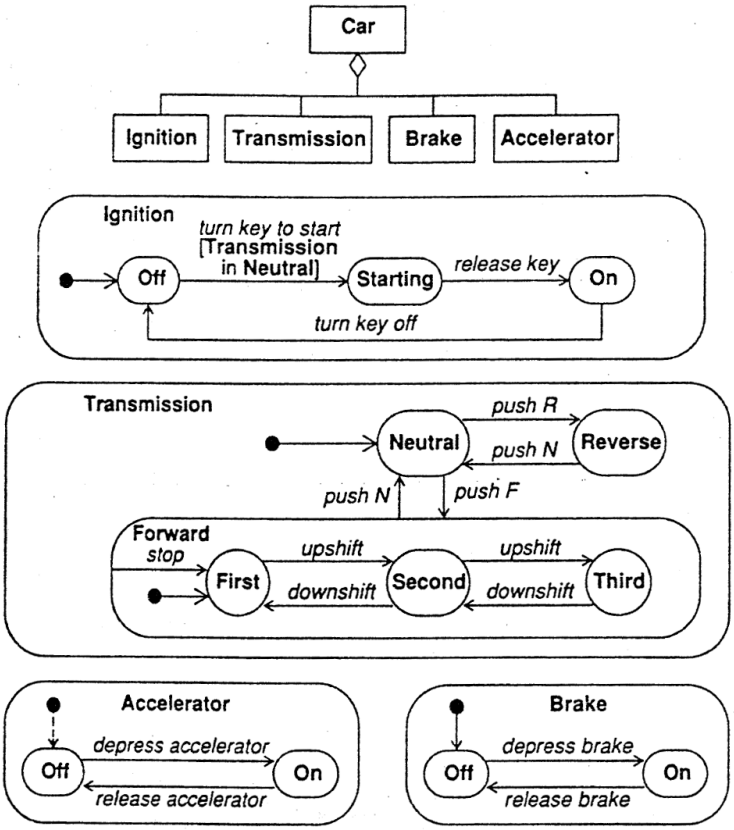
Tilakaavioista tulee usein laajoja ja vaikeaselkoisia. Tällöin voidaan käyttää normaaleja abstrahointiperiaatteita (yleistys, koostaminen) kuvausten rakenteistamiseksi:

- tilaan liittyvä toiminta voidaan kuvata yksityiskohtaisemmin omana tilakaavionaan, jossa kukin tila vastaa yhtä askelta ylemmän tason tilakaavion tilaan liittyvässä toiminnassa (kuva 5.50).
- tila voidaan kuvata yksityiskohtaisemmin useampana alitilana tilakaaviossa (vrt. Vaihteen tilakaavio) (kuva 5.52).

Vinkkejä tilakaavioiden laadintaan:

- Tee tilakaaviot vain niille luokille, joiden olioiden käyttäytymisen yksityiskohtainen selvittäminen on tarpeellista
- Tarkista tilakaavioiden sisäinen ja välinen johdonmukaisuus
- Käytä tarvittaessa sisäkkäisiä (nested) rakenteita
- Esitä aliluokkien tilakaaviot erillisinä yliluokan tilakaaviosta
- onko käyttäytyminen kuvattu jokaisen relevantin olioluokan/attribuutin osalta?
- jokaisella viestillä tulee olla lähettäjä ja vastaanottaja (joissakin tapauksissa sama olio)
- jokaisella tilalla tulisi olla edeltäjä ja seuraaja (alkua ja loppua lukuunottamatta)

- jokainen tilasiirtymä tulisi olla kuvattuna jossakin sekvenssi- ja vuorovaikutuskaaviossa johdonmukaisella tavalla
- viestien/herätteiden rinnakkaisuus tulee mallintaa oikein



Kuva 5.52: Kooste auton eräistä järjestelmistä ja rinnakkaisista tilakaavioista. Auton järjestelmiä ovat mm. sytytys, vaihteisto, kiihdyttäminen ja jarruttaminen. Kuvataan ne tässä erikseen. Tässä tapauksessa käytössä on automaattivaihteisto.

Sytytysjärjestelmä: Auto on sammuksissa. Kun virta-avaimesta käännetään, auton moottori alkaa toimia (vaihteen pitää olla vapaa). Kun avaimesta päästetään irti, moottori käy. Moottori sammutetaan kääntämällä avainta vastakkaiseen suuntaan.

Vaihteisto: Vaihde on ensin vapaa. Vaihteen voi vapaasta vaihtaa joko peruutusvaihteeksi tai eteenpäinmenovaihteeksi valitsemalla asianmukaisen asennon vaihdekepistä (R=peruutus, N=vapaa, F=eteenpäin). Peruutusvaihteen voi vaihtaa ainoastaan vapaaksi. Eteenpäinmenovaihteeksi vaihdettu vaihde on ensin ykkönen ja vaihtuu nopeuden kasvaessa kakkoseksi, sitten kolmoseksi jne. Nopeuden hidastuessa vaihde vaihtuu samalla tavalla alaspäin, ja lopulta ykköseksi, kun auto on pysähtynyt. Eteenpäinmenovaihteen voi vaihtaa vain vapaalle.

Kiihdytys: Kiihdytysvaiheessa kiihdytin menee päälle, kun kaasupoljinta painetaan ja sammuu, kun kaasupoljin vapautetaan.

Jarrutus: Jarrutusvaiheessa jarrut menevät päälle, kun jarrupoljinta painetaan ja sammuvat, kun jarrupoljin vapautetaan.

Kohdealueluokkien mallintamisen lisäksi tilakaavioilla voidaan kuvata käyttöliittymän logiikkaa. Tätä käsitellään tarkemmin luvussa 5.3.3.3.

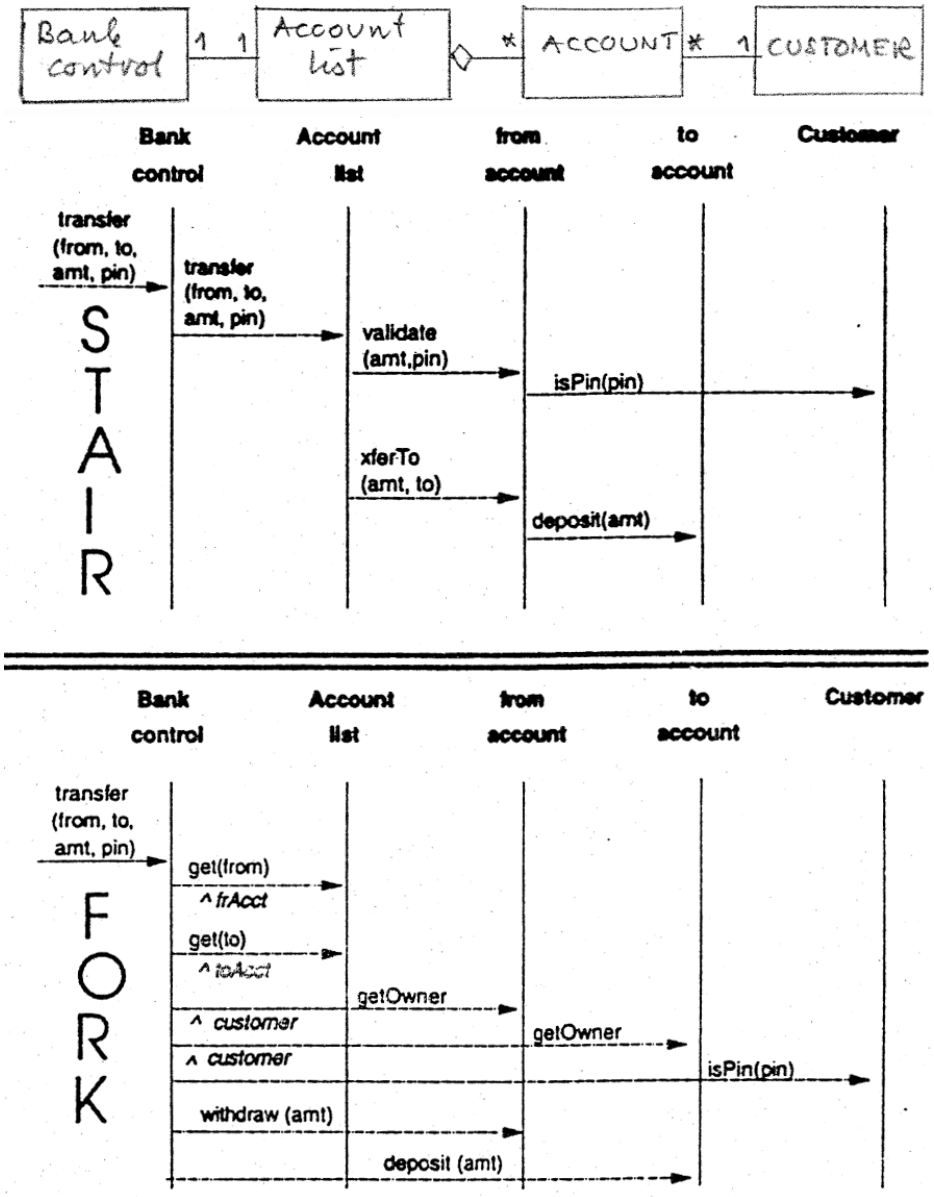
5.3.2.2 Operaatioiden sijoittaminen luokkiin

Tämän oliosuunnittelutehtävän tarkoituksena on löytää ja muodostaa operaatioita sijoitettaviksi luokkiin. Lähtökohtana toimivat dynaamiset mallit ja CRC-kortit:

- sekvenssikaaviot ja yhteistoimintakaaviot:
  - ◊ kuvaavat olioiden välistä vuorovaikutusta,
  - ◊ jokainen viesti, jonka on esitetty saapuvan oliolle, tulee tulkituksi ja huomionotetuksi vasta sitten, kun vastaanottaneella oliolla on vastaava operaatio.
  - ◊ jokainen viesti, jonka on osoitettu lähtevän, edellyttää, että olemassa olevien operaatioiden osana on jokin sellainen käsky, joka muodostaa viestin, laittaa sen liikkeelle ja odottaa sitä seuraavaa palautetta.
- tilakaaviot:
  - ◊ kuvaavat olioiden elinkaarta; ts. mitä oliot tekevät
  - ◊ jotta heräte saisi aikaan halutun tilasiirtymän, tulee usein suorittaa toimintoja. Näiden toimintojen tulee olla operaatioina tai niiden osina ao. luokassa.
  - ◊ Kutakin tilassa suoritettua toimintoa (merkitty *do:*) varten tarvitaan operaatio tai sen osa ao. luokassa

Edellä annetut ohjeet ovat varsin yleisluonteisia. Käytännössä joudutaan hyvinkin tarkkaan miettimään, minkä luokkien vastuulle järjestelmän toimintaa jaetaan. Tästä annetaan seuraavaksi tarkempia sunnitteluohjeita:

1. Yksittäisten tehtävien suorittaminen (esim. nimen hakeminen, varastosaldon pienentäminen),
   - ◊ tehtävä kohdistuu tyypillisesti ja rajatusti tietyn luokan olioihin, jolloin operaatio sijoitetaan asianomaiseen luokkaan,
2. Laajemman toimintakokonaisuuden jakaminen operaatioiksi:
   - ◊ toiminnan kohteena ja suorittajana useampia olioita,
   - ◊ ongelmana sen päättäminen, millä oliolla on ohjausvastuu,
   - ◊ ohjausvastuu voidaan antaa (harvemmin) keskeiselle liiketoimintaoliolle tai erityiselle ohjausoliolle. Jacobson (1994) on esittänyt jälkimmäiselle tapaukselle kaksi perusmallia: (1) porrasmallin ja (2) haarukkamallin (kuva 5.53):
     - *Porrasmalli* hajauttaa velvollisuudet toiminnan ohjauksen ja operaatioiden käynnistämisen osalta hajautetaan useammalle luokalle,
       - + luokkaan kuuluva peruskäyttäytyminen (esim tiliin läheisesti kuuluva pin-koodin tarkastus) tulee sisällytettyä luokkaan, jolloin sen varaan voidaan rakentaa muissakin yhteyksissä,
       - + monimuotoisuuden soveltamismahdollisuudet paranevat,
       - + kehysten käyttömahdollisuudet paranevat.
     - *Haarukkamalli* sijoittaa velvollisuudet toiminnan ohjauksen ja operaatioiden käynnistämisen osalta sijoitetaan pelkästään ohjausluokalle (Bank control)
       - + ohjauksen hallinta paranee,
       - + liiketoimintaluokkien operaatioista tulee yksinkertaisia ja sen myötä uudelleenkäytettäviä,
       - + jos joskus joudutaan ohjaustapaa muuttamaan, muutos rajautuu vain yhden luokan (so. ohjausluokan) sisälle.



Kuva 5.53: Porras- ja haarukkamalli.

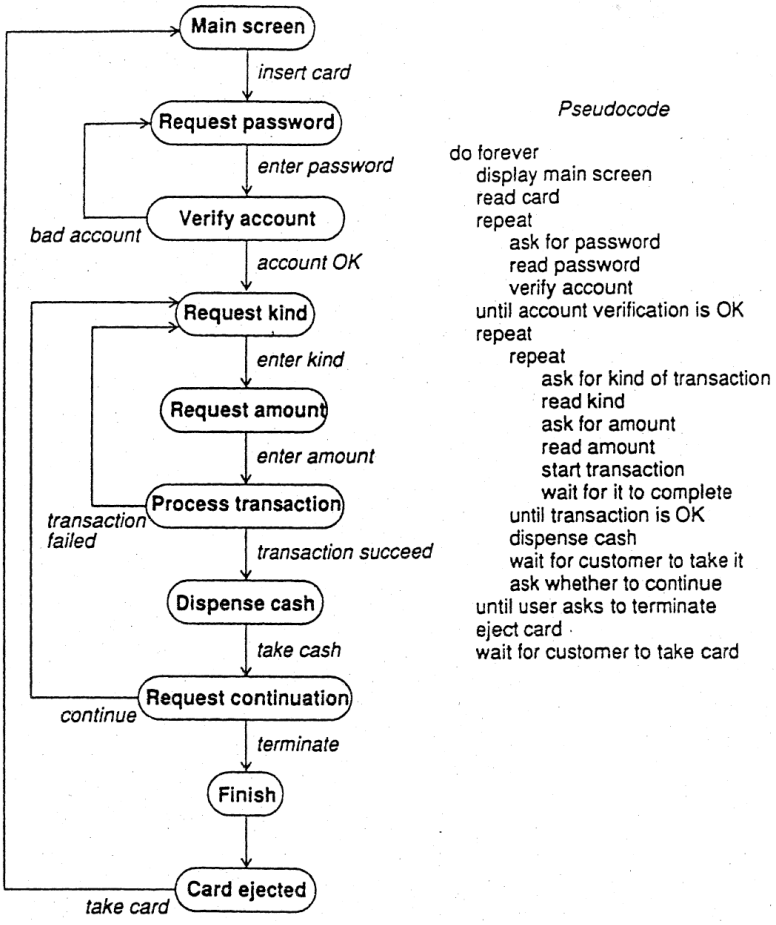
3. Oikean yleistystason löytäminen:
   ◊ operaatiolle tulee hakea luokka, joka on sopivalla tasolla yleistyshierarkiassa.
4. Liiallisen kuormittamisen välttäminen:
   ◊ Operaation tulee tehdä ”yksi” asia kunnolla mieluummin kuin, että siihen sisällytetään useita erilaisia velvollisuuksia. Tällä edistetään ylläpidettävyyttä ja uudelleenkäyttöä.
5. Liian suurien luokkien välttäminen:
   ◊ Yleinen ohje (OMT): jos luokka sisältää enemmän kuin 10 attribuuttia tai enemmän kuin 20 operaatiota tai jos se on osallisena useammassa kuin 10 assosiaatiossa, luokka tulisi jakaa osiin, edellyttäen että attribuutit, operaatiot ja assosiaatiot ovat jaettavissa sopivasti uusien luokkien kesken.

5.3.2.3 Operaatioiden määritys ja algoritmien suunnittelu

Tämän tehtävän tarkoituksena on määrittää kullekin ei-triviaalille operaatiolle ohjausrakenne, algoritmi ja sopiva tietorakenne:

Operaatioita toteuttavien ohjelmien ohjausrakenteen määrittämiseksi voidaan käyttää useita strategioita. OMT-kirjassa kuvataan kolme, joista yhtä tarkastellaan seuraavaksi tarkemmin (katso ATM-esimerkki kuvassa 5.54).

1. ota lähtökohdaksi ko. luokan tilakaavio
2. tunnista pääpolku aloitustilasta mahdolliseen päätöstilaan ja kirjoita polun varrella olevien tilojen nimet allekkain.
3. tunnista vaihtoehtoiset haarat, jotka myöhemmin yhtyvät pääpolkuun. Nämä toteutetaan myöhemmin valintarakenteiden (if..then..else) avulla
4. tunnista palautuspolut. Nämä tulevat esiintymään silmukkoina ohjelmassa.
5. Jäljelle jääneet polut vastaavat poikkeustapauksia. Niistä voidaan huolehtia alirutiineilla, ohjelmointikielen tarjoamilla poikkeustenkäsittelymekanismeilla tms.



Kuva 5.54: Pankkiautomaatin ohjaus. Pankkiautomaatin päänäyttö pyytää asiakasta laittamaan automaattikorttinsa korttiaukkoon. Tunnistettuaan kortin automaatti pyytää asiakkaalta tunnusluvun, jonka se tarkistaa. Jos tunnusluku oli oikein, automaatti jatkaa tehtävänäytölle. Muussa tapauksessa automaatti pyytää tunnuslukua uudelleen. Automaatti pyytää asiakasta valitsemaan tehtävän tyypin (esim. nosto). Tämän jälkeen automaatti pyytää asiakkaalta nostettavaa rahamäärää. Määrän antamisen jälkeen automaatti suorittaa tapahtuman ja antaa asiakkaalle rahat. Mikäli tapahtuman suorittaminen jostain syystä epäonnistuu, järjestelmä pyytää asiakasta valitsemaan tehtävän tyypin uudelleen. Kun asiakas on ottanut rahat, automaatti tarjoaa mahdollisuutta tehdä uuden toimenpiteen. Mikäli asiakas ei haluaa jatkaa, automaatin käyttö lopetetaan ja automaatti palauttaa asiakkaalle automaattikortin.

Ohjausrakennerungon jälkeen voidaan suorittaa:

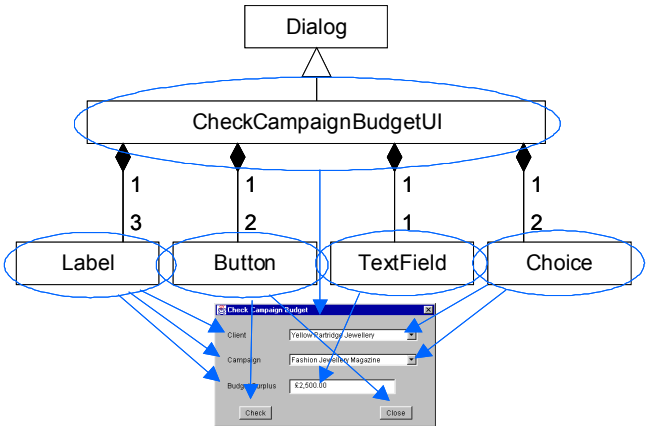
- algoritmin valinta:
  - ◊ valintakriteerejä:
    - • laskennallinen kompleksisuus: käsittelyaika tietomäärän funktiona
    - • toteutuksen helppous ja ymmärrettävyys
    - • joustavuus tulevaisuuden muutosten ja laajennusten varalta
    - • mahdollisuus tehokkuuden parantamiseen luokkakaaviota muuttamalla
- tietorakenteen valinta:
  - ◊ attribuuteille ja attribuuttijoukoille
  - ◊ vaihtoehtoja: taulukko, lista, jono, pino, joukko, hakemisto, puu, jne (useimmat kielet sisältävät perustietorakenteet standardikirjastossaan, esim. Javassa Collections Framework[35], C++:ssa STL[36]).
- sisäisten luokkien ja operaatioiden määrittäminen (ks. luku 5.3.1.4)

5.3.3 Käyttöliittymäluokkien suunnittelu

Suunnitteluvaiheessa tulee mallintaa käyttöliittymäluokkien keskeiset toiminnot. Kaikkien käyttöliittymäkomponenttien yksityiskohtien mallintaminen UML:llä ei ole tarkoituksenmukaista (sovelluskehitinten visuaaliset suunnittelutyökalut toimivat tässä huomattavasti paremmin ja samalla käyttöliittymästä saadaan helposti prototyyppejä), vaan lähtökohtana voidaan pitää analyysivaiheessa mallinnettuja *boundary*-luokkia. Käyttöliittymäluokkia voidaan tarvittaessa jakaa uudelleen esim. lomakekohtaisesti - jopa niin, että analyysivaiheen käyttöliittymäluokka on peritty käyttöliittymäkirjaston lomake- tai ikkunaluokasta (esim. Javassa JFrame, Object Pascalissa TForm).

5.3.3.1 Käyttöliittymän integrointi luokkakaavioon

Käyttöliittymän komponentit ovat samantapaisia olioita kuin liiketoimintaoliotkin: niillä on rakenne (attribuutit) ja käyttäytyminen. Niiden tulee olla tietoisia toisistaan ja myös liiketoimintaolioista, jotta ne voivat viestiä keskenään. Tästä syystä luokkakaaviota tulee täydentää käyttöliittymäluokilla. Käyttöliittymäkomponentteja löydetään usein luokkakirjastosta. Niitä voidaan käyttää sellaisenaan tai erikoistamista suorittaen. kuvassa 5.55 (Bennett ym, 2006) on esitetty käyttöliittymäluokkia, joihin liittyvät ohjaus- ja kohdealueluokat on esitetty kuvassa 5.56.

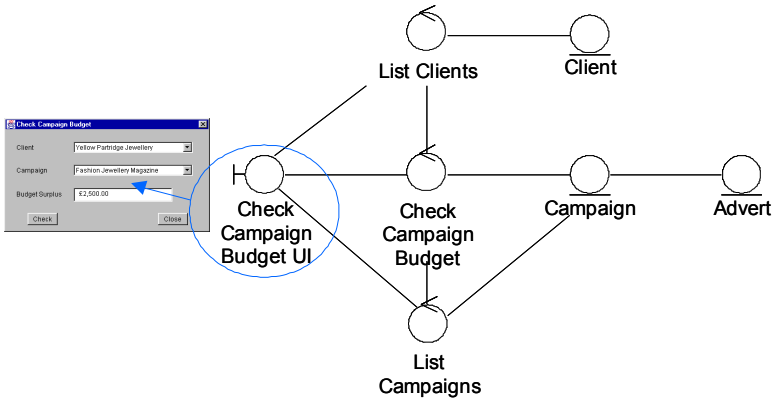


Kuva 5.55: Kuvan 3.5 käyttötapaukseen liittyviä käyttöliittymäluokkia.

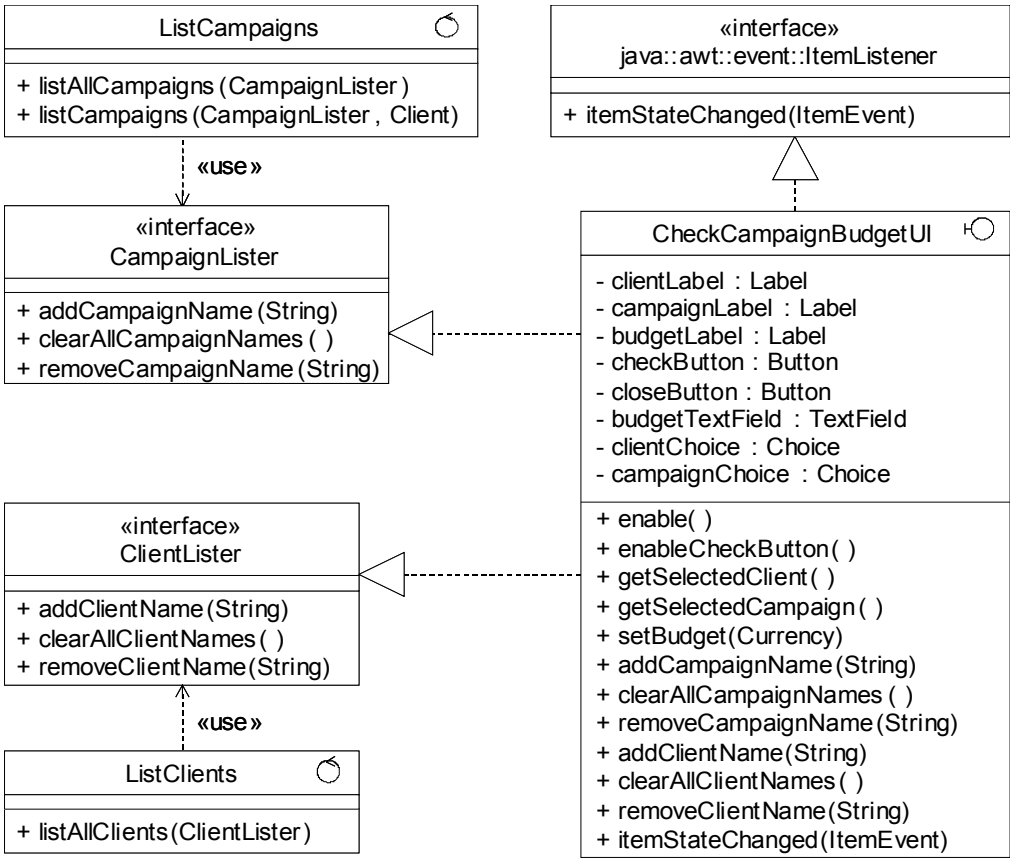
[35] http://java.sun.com/docs/books/tutorial/collections/

[36] http://www.cplusplus.com/reference/stl/

Käyttöliittymäluokat kutsuvat pääsääntöisesti ohjausluokkia (=käyttäjä ohjaa sovelluksen toimintaa), mutta on myös tilanteita, jossa ohjausluokka kutsuu käyttöliittymäluokkaa (esim. uusia lomakkeita avatessa tai päivitettäessä tietoja lomakkeiden välillä). Tällöin käyttöliittymä ja sovelluslogiikka voidaan erottaa toisistaan abstrahoimalla käyttöliittymäluokan muualta kutsuttavat toiminnot rajapinnoiksi. Ohjausluokan ei tarvitse tietää käyttöliittymäluokan toteutustapaa, vaan ainoastaan rajapinta, jossa toiminnon keskeisimmät piirteet on kuvattu. Esimerkki rajapintojen käytöstä mainostoimistoesimerkin yhteydessä on kuvassa 5.57 (Bennett 2006).



Kuva 5.56: Käyttöliittymäluokat ja ohjaimet huomioiva yhteistoimintakaavio (collaboration, robustness diagram) käyttötapaukselle ”tarkasta kampanjabudjetti”.

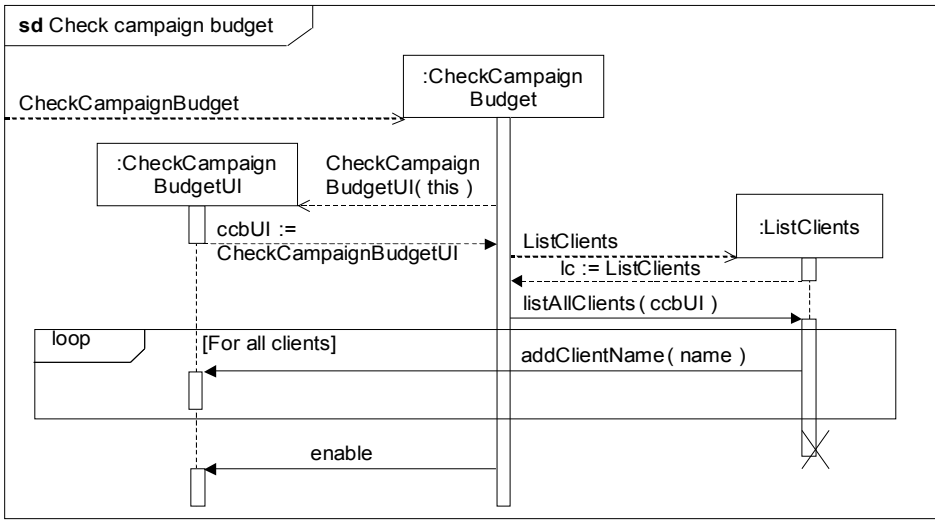


Kuva 5.57: Luokkakaavio kampanjabudjetin tarkistukseen liittyvistä käyttöliittymä- ja ohjainluokista. Rajapintojen CampaignLister ja ClientLister käyttö mahdollistaa käyttöliittymän toteutuksen vaihdon ohjainluokista riippumatta.

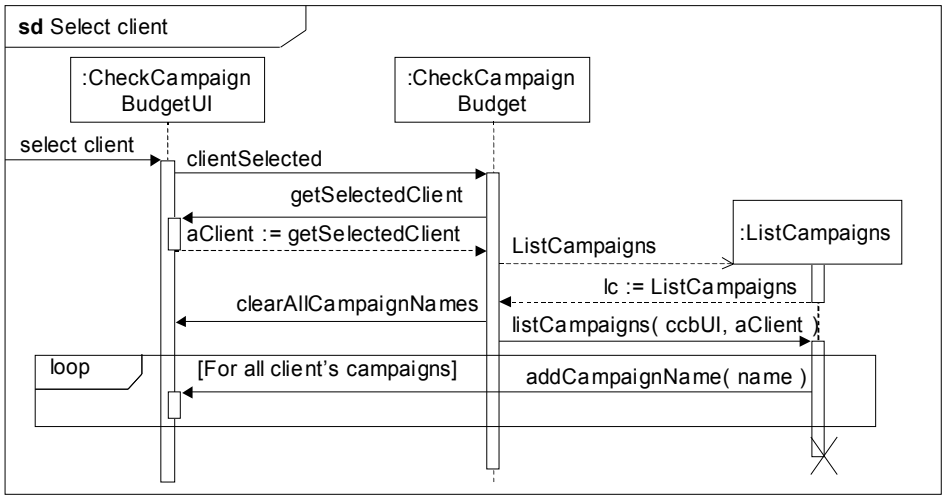
5.3.3.2 Käyttöliittymän integrointi sekvenssikaavioihin

Sen jälkeen kun on saatu jäsennettyä käyttöliittymä komponenteiksi voidaan sekvenssikaavioilla (tai yhteistoimintakaavioilla) osoittaa, miten kukin niistä käyttäytyy ja toimii yhdessä muiden olioiden kanssa. Tällöin sekvenssikaavioon otetaan mukaan käyttöliittymäoliot ja täydennetään viestinnän kuvausta ko. olioiden välisillä viesteillä.

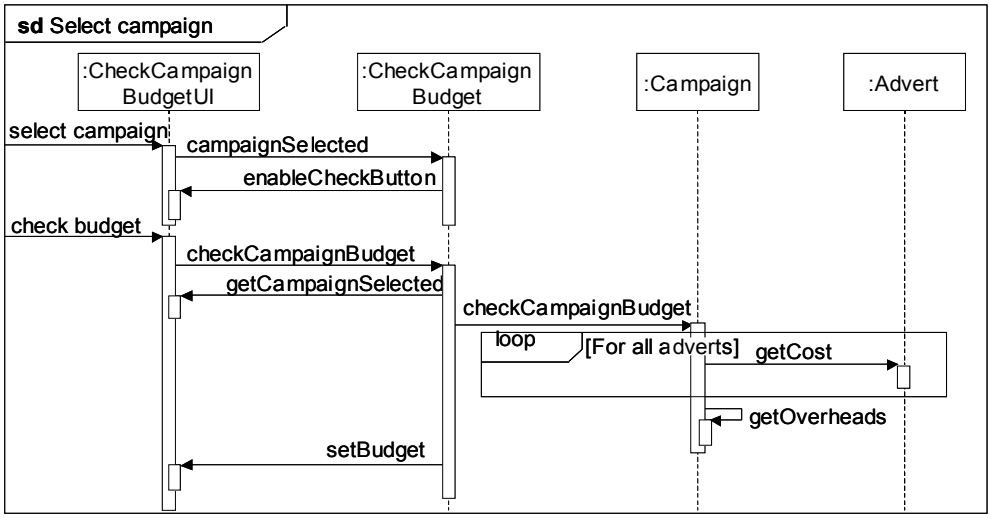
Kuvissa 5.58-5.60 (Bennett ym. 2006) on esitetty tarkennetut sekvenssikaaviot, jotka liittyvät kuvassa 5.55 (Bennett, ym. 2006) esitettyyn käyttöliittymään sekä luvuissa 3.2 ja 4.3.1.1 esitettyihin käyttötapaukseen ja yhteistoimintakaavioon. Sekvenssikaavio kuvaa toimintaa, jolla pyritään selvittämään mainoskampanjan budjetti. Käyttäjälle tarjotaan mahdollisuus etsiä tietoa tunnistamalla ensin ao. asiakas ja sitten kampanja (CheckCampaignBudgetUI-komponenttiin kuuluvilla Choice-valinta). Tämän jälkeen käydään hakemassa kampanjaan kuuluvien mainosten hinnat ja yleiskulut.



Kuva 5.58: Tarkennettu sekvenssikaavio ”tarkasta kampanjabudjetti”-käyttötapauksen alusta (analyysivaiheen kuvaus kuvassa 4.29). Dialogiin (kuva 5.55) lisätään aluksi asiakkaat.



Kuva 5.59: Jatkoa kuvan 5.58 sekvenssikaaviolle. Käyttäjä valitsee asiakkaan, jonka jälkeen näytetään asiakkaaseen liittyvät kampanjat.



Kuva 5.60: "Tarkasta kampanjabudjetti"-käyttötapauksen lopun kuvaus tarkennettuna sekvenssikaaviona. Kampanjan valinnan jälkeen kustannukset lasketaan yhteen.

5.3.3.3 Käyttöliittymälogiikan mallintaminen tilakaavioilla

Käyttöliittymän dynaamisia ominaisuuksia voidaan sekvenssikaavioiden lisäksi mallintaa tilakaavioilla. Tilojen mallinnuksen lähtökohtana ovat tällöin näytöt ja lomakkeet, joissa käyttäjä navigoi.

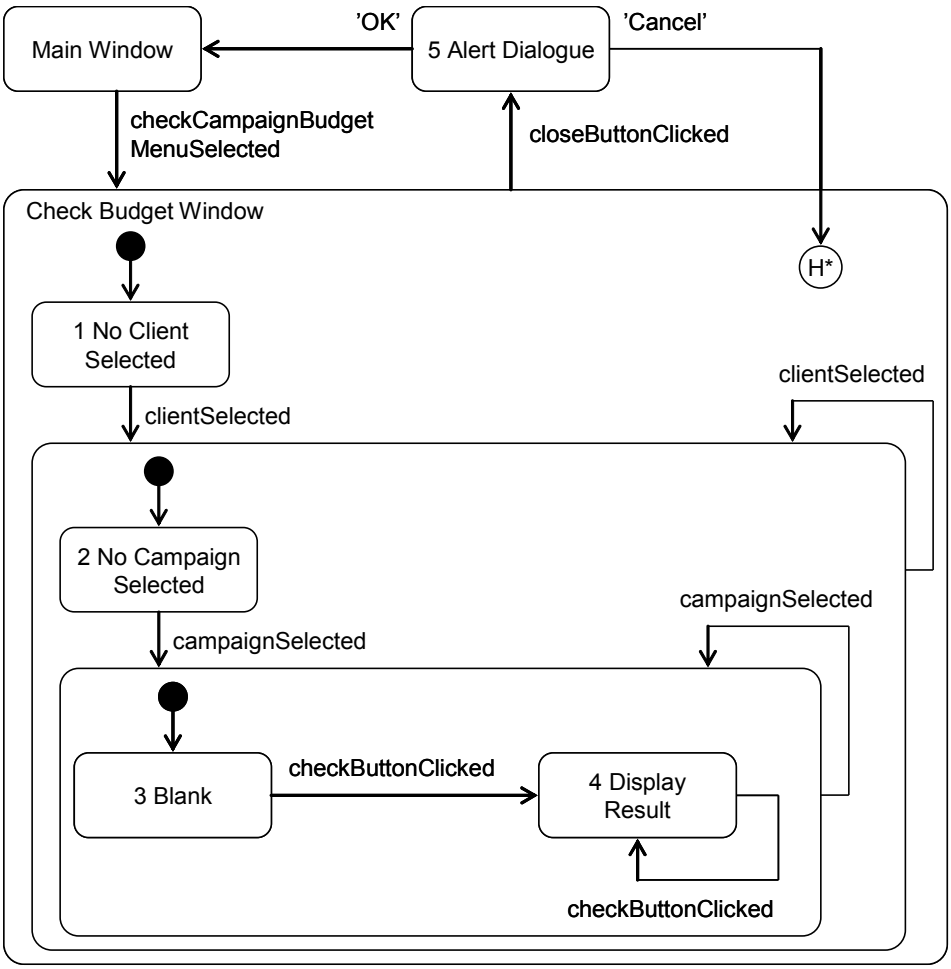
Esimerkkinä tilakaavioihin perustuvasta käyttöliittymän mallintamisesta esitetään Horrocks'n[37] menetelmä (Bennett 2006). Sen pääkohdat ovat seuraavat:

1. Kuvaa korkealla tasolla vaatimukset ja käyttäjän päätoiminnot
2. Kuvaa käyttöliittymän toiminnallisuus (yksittäisten komponenttien tasolla)
3. Määrittele käyttöliittymän säännöt (esim. käyttöliittymäkomponenttien alkutila ja tärkeimmät tapahtumat; ikkunan tai lomakkeen saapumis- ja poistumisehdot)
4. Piirrä tilakaavio ja tarkenna iteratiivisesti
5. Laadi tapahtuma/toiminto-taulukko (event action table)

Tilakaavion piirtäminen aloitetaan käyttöliittymän ylimmältä tasolta, jolloin tilakaavio on lähellä rakenteisen käyttöliittymäsuunnittelun hierarkiakuvauksia tai OMT++ -menetelmän dialogikaavioita (luku 4.4). Tarkemmilla tasoilla voidaan kuvata sallittuja tiloja tietyn lomakkeen tai yksittäisen käyttöliittymäkomponentin sisällä. Tilakaavioista voi tulla helposti monitasoisia, minkä takia varsinaista tilakaaviota on yksinkertaistettu (kuva 5.61): osa tiloista voidaan jättää nimeämättä ja tiloihin liittyvät toiminnot kootaan erilliseen tapahtuma/toiminto-taulukkoon (kuva 5.62). Viittaamisen helpottamiseksi tilat on numeroitu.

Tilakaavion mallintamisen seurauksena myös muita malleja voidaan joutua päivittämään. Mainostoimistoesimerkissä kuvan 5.58 kaavion loppuun tulisi lisätä *disable*-viesti kampanjalistaan ja budjetin laskentaan liittyvään painikkeeseen: asiakasta ei ole valittu tässä vaiheessa, joten myöskään kampanjan valinnan tai budjetin laskennan ei pidä vielä olla käytettävissä.

[37] I. Horrocks (1999). Constructing the User Interface with Statecharts



Kuva 5.61: Mainostoimiston kampanjoiden ja asiakkaiden hallintaan liittyvän käyttöliittymän mallintaminen tilakaaviolla (yksinkertaistettu notaatio)

Current State	Event	Action	Next State
–	Check Campaign Budget menu item selected.	Display CheckCampaignBudgetUI. Load client dropdown. Disable campaign dropdown. Disable check button. Enable window.	1
1	Client selected.	Clear campaign dropdown. Load campaign dropdown. Enable campaign dropdown.	2
2, 3, 4	Client selected.	Clear campaign dropdown. Load campaign dropdown. Clear budget textfield. Disable check button.	2
2	Campaign selected.	Clear budget textfield. Enable check button.	3
3	Check button clicked.	Calculate budget. Display result.	4
3, 4	Campaign selected.	Clear budget textfield.	3
4	Check button clicked.	Calculate budget. Display result.	4
1, 2, 3, 4	Close button clicked.	Display alert dialogue.	5
5	OK button clicked.	Close alert dialogue. Close window.	–
5	Cancel button clicked.	Close alert dialogue.	H*

Kuva 5.62: Tapahtuma/toiminto -taulukko

5.3.4 Tiedonhallinnan suunnittelu

Kohdealueluokat ovat lähes poikkeuksetta pysyviä[38] (persistent), jolloin niiden tietojen täytyy olla ladattavissa ja tallennettavissa järjestelmän käyttökertojen välillä. Tähän asti laadituissa malleissa ei pääosin ole otettu kantaa tietojen tallennustapaan, mutta käytännössä tiedonhallinta on kriittinen osa järjestelmän sisäistä toimintaa ja voi vaikuttaa merkittävästi erityisesti kohdealueluokkien toteutukseen.

5.3.4.1 Tallennusteknologian valinta

Pysyväisluonteiset oliot tulee tallentaa tiedosto-, tietokanta-, oliokanta- tai oliorelaatiokantateknologiaa hyväksikäyttäen. Liiketoimintaoliot ovat miltei poikkeuksetta pysyväisluonteisia olioita. Luonnollisin tallentamisalusta on oliokanta, koska silloin oliomäärityksiin ei tarvitse tehdä mitään muutoksia (eli “deobjektifikaatiota”). Olioihin päästään käsiksi tehokkaasti navigoinnin avulla eli seuraamalla osoittimia, kunhan vain tiedon käyttötavat kyetään riittävän tarkoin ennakoimaan jo suunnitteluvaiheessa.

Oliokannat sopivat erityisesti sellaisiin sovelluksiin, joissa on

- komplekseja tietorakenteita,
- hyvin dynaaminen ja vuorovaikutteinen käyttöliittymä,
- tarkoitus käyttää toteutuksessa oliokieltä, joka tukee olioiden sarjallistamista (serialisointia), esim. Java, Python.

Tällaisia sovellusalueita ovat esimerkiksi:

- tietoliikennejärjestelmät (verkonhallintajärjestelmät, reititys),
- paikkatietojärjestelmät (GIS),
- insinöörisovellukset (CAD/CAM, sis. versioinnin),
- tuotemallintaminen,
- asianhallintajärjestelmät (workflow systems) ja dokumenttien hallinta,
- tietokoneavusteinen suunnittelu ja toteutus (CASE)
- kuvien käsittely (lääketieteelliset ja satelliittisovellukset),
- rahoituksen suunnittelu (portfoliot, vakuutus- ja verosuunnittelu),
- simulointi ja animointi.

Oliokantateknologia on vielä osittain vakiintumatonta eikä esimerkiksi kyselyjen tekemistä helpottavia standardeja ole kattavasti noudatettu. On esimerkiksi epäselvää, voidaanko tietyllä ohjelmointikielellä luotuja ja tietokantaan talletettuja olioita hakea tietokannasta millään muulla ohjelmointikielellä toteutetulla järjestelmällä. Tärkeimmät perinteiset kaupallishallinnollisia tietojärjestelmiä hyödyntävät toimialat, jotka vaativat erittäin tehokasta, skaalautuvaa ja luotettavaa tiedonkäsittelykapasiteettia, eivät näin ole voineet täysimittaisesti hyödyntää oliotietokantoja. Yritykset, jotka ovat sijoittaneet suuria summia perinteiseen teknologiaan (kuten relaatiokannat), eivät ole olleet halukkaita siirtymään uuteen teknologiaan, jonka asema markkinoilla ja liiketoimintahyödyt ovat epävarmoja. Oliotietokantojen hyödyntämistä edesauttavat kuitenkin viime vuosina markkinoille tulleet avoimeen lähdekoodiin perustuvat edulliset (lisensseiltään jopa ilmaiset) ratkaisut (esim. Java-pohjainen ObjectDB).

[38] Olio on pysyväisluonteinen, jos sen elinkaari on pitempi kuin sen luoneen prosessin elinkaari. Olioita, jotka eivät ole pysyväisluonteisia, kutsutaan tilapäisiksi (transient).

Toinen mahdollisuus on käyttää rakenteista tallennustapaa. Tällöin yksinkertaisin (mutta käyttäjien tai datan kasvaessa huonosti skaalautuva) tapa tallentaa olioita on käyttää tiedostoja. Jos käytettävä kieli tukee serialisointia, olion lataus ja tallennus on suoraviivaista (kehittäjän ei tarvitse ottaa kantaa olion esitystapaan – tallennuksen toteutuksessa on kuitenkin huomioitava pysyvän olion riippuvuudet muihin pysyviin olioihin, että viite-eheys säilyy), muuten olion tila täytyy muuntaa tiedostomuotoon manuaalisesti, ja lataus- ja tallennusmekanismit on toteutettava itse tai käytettävä tiettyyn formaattiin erikoistunutta kirjastoa (esim. XML-prosessorit) Tiedosto:

- soveltuu tilanteisiin, joissa tietorakenne poikkeaa tavanomaisesta (esim. monet binaariformaatit), tietoa tallennetaan lyhytaikaisesti, tieto on yhteenvetotyyppistä,
- arvioita:
  + halpa ja yksinkertainen rakenteeltaan,
  – käsittely edellyttää usein määrittelyä matalatasoisella kielellä,
  – käsittelyn tehokkuus laskee nopeasti tiedon määrän kasvaessa.

Tiedostopohjainen ratkaisu ei skaalaudu laajempiin tietomalleihin, suurempiin datamääriin tai useille samanaikaisille käyttäjille, ellei järjestelmään toteuteta omia indeksointirakenteita, hakumekanismia tai huolehdita samanaikaisuuden hallinnasta – tietokantapalvelinta käyttäessä suurin osa näistä on eristetty järjestelmästä palvelimen vastuulle, jolloin tiedonhallinta helpottuu oleellisesti. Tiedostoille on kuitenkin tietokantapohjaisissakin järjestelmissä käyttöä esim. konfigurointiin tai ulkoisten tietojen muuntamiseen järjestelmään tai järjestelmästä, esim. XML:ää siirtoformaattina hyödyntäen.

Pysyvien olioiden rakenteiseen tallentamiseen laajemmassa mittakaavassa on siis käytännössä käytettävä tietokantoja. Tietokantana voidaan käyttää perinteistä relaatiomalliin perustuvaa kantaa. Relaatiotietokanta soveltuu tilanteisiin, joissa tiedolla on useita samanaikaisia käyttäjiä; tietoa on hyvin paljon; tietojen välillä on runsaasti loogisia sidoksia; käyttäjien on kyettävä etsimään ja yhdistelemään tietoja tavoilla, joita suunnitteluvaiheessa ei täysin tiedetä; ja tietoa voidaan käsitellä tehokkaasti tietokannan hallintajärjestelmillä (tkhj) (esim. Oracle, OpenIngres, IBM DB2, MS SQL Server. Rajoitetuissa ympäristöissä myös MS Access). 1990-luvun loppupuolelta asti markkinoilla on ollut myös avoimeen lähdekoodiin perustuvia hyvin edullisia ratkaisuja (esim. MySQL, PostgreSQL), minkä seurauksena esimerkiksi Oraclen ja IBM:n kaupallisista tuotteista on julkaistu ilmaisia variantteja pienten tietokantojen rakentamiseksi.

- arvioita:
  + tkhj:t tarjoavat valmiit mekanismit (infrastruktuurin) toipumiseen, rinnakkaisuuden hallintaan, hajautukseen, johdonmukaisuuden varmistamiseen, suojaukseen, jne.
  + tkhj:t tarjoavat yhtenäiset rajapinnat (ODBC, JDBC, ActiveX Data Objects jne) ja kielen (SQL92, SQL99) sovelluksille
  – relaatiomallin rajoittuneisuus mm. kompleksisten rakenteiden käsittelyssä (esim. vaatimusten muuttuessa 1-M-suhteen muuttaminen M-M-suhteeksi),

– SQL-kielen ei-proseduraalisuus, joka vaikeuttaa rajapinnan määrittelyä olio-ohjelmiin päin (”Object-Relational Impedance Mismatch”),

– tehokkuusongelmat (jos kyselyjä ei ole optimoitu tai indeksejä ei ole määritelty oikein).

SQL 1999-standardi on helpottanut olioparadigman hyödyntämistä myös relaatiotietokantojen yhteydessä. Relaatiotietokantateknologian kehittäjät ovatkin vastanneet oliolähestymistavan haasteeseen laajentamalla omia tuotteitaan oliopiirteillä. Tuloksena on syntynyt oliorelaatiomalliin (OR-malli) perustuvia kaupallisia tuotteita, joista tärkeimmät ovat Oracle10 ja IBM:n DB2. Relaatiorakenteiden lisäksi OR-malli antaa käyttäjälle mahdollisuuden määritellä tietotyyppejä, toteuttaa periytymistä, ja käsitellä komplekseja multimediatietorakenteita. Oliorelaatiokannan hallintajärjestelmät pystyvät tarjoamaan perinteisten hallintajärjestelmien palvelujen lisäksi siis myös oliomuotoisten tietojen käsittelyä. Ne ovat myös teknisessä mielessä riittävän tehokkaita toimimaan laajoja ja monimutkaisia tietorakenteita sisältäviä tietokantoja tarvitsevien sovellusten alustana. Valitettavasti tietokantatoimittajat ovat noudattaneet SQL 1999 standardia ja sen 2003 parannettua versiota olioparadigman edellyttämien uusien piirteiden osalta vaihtelevasti ja jokainen toimittaja on tehnyt räätälöidyt ratkaisunsa.

Näistä teknisistä, liiketoiminnallisista, organisatorisista, ja standardeihin liittyvistä syistä johtuen oliorelaatiomalliin perustuvat järjestelmät ovat yleistymässä ja puhtaat oliotietokannat ovat vallanneet vain pienen osan tietokantamarkkinoista. Tekniikan vakiintuessa ja avoimeen lähdekoodiin perustuvien ratkaisujen myötä on kuitenkin syytä olettaa, että oliotietokannat yleistyvät edelleen.

5.3.4.2 Tiedonhallintaluokkien suunnittelu

Kohdealueolioiden pysyvyyden toteutustapaan vaikuttavia tekijöitä (Bennett 2006):

- Voiko tallennukseen käyttää tiedostoja?
- Onko järjestelmä ”aidosti” oliopohjainen vai onko kyseessä vain rajapinta relaatiotietokantaan (esim. kysely/raportointisovellukset)?
- Onko käytössä (relaatio/olio)tietokanta?
- Mikä on järjestelmän fyysinen/looginen kerrosjako vai käytetäänkö jotain muuta arkkitehtuurityyliä?
- Onko järjestelmä tai sen tietovarasto hajautettu?
- Millä protokollilla järjestelmä kommunikoi?

Jos järjestelmän kohdealueen kaikki *tiedot* ovat relaatiotietokannassa ja tietokannassa olevia tietoja on tarpeen hakea ja yhdistellä mielivaltaisilla tavoilla tai suorituskykyvaatimukset ovat kovat, erillisiä kohdealueluokkia ei välttämättä tarvitse toteuttaa. Tällöin ohjausluokat käsittelevät tietokantaa suoraan sql-kielellä ja hakutuloksia rivejä ja sarakkeita sisältävinä tauluina. Tämä vaatii kuitenkin huomattavaa lisätyötä sovelluslogiikkakerroksessa ja mahdollisia ylläpidettävyysongelmia, koska kohdealueluokkia vastaava *toiminnallisuus* jäisi tietokannasta riippuville ohjausluokille. Ratkaisua ei myöskään voi pitää oliopohjaisena (vaikka toteutuksessa olisi käytetty oliokieltä), vaan pikemminkin rakenteisena, koska kohdealueen tiedot ja toiminnot ovat toisistaan erillään.

Koska relaatiotietokannat ovat ei-oliomaisuudestaan huolimatta vallitseva tekniikka laajojen tietomassojen säilömiseen, ne on otettava huomioon myös tilanteessa, jossa järjestelmä toteutetaan muuten oliopohjaisesti. Tavoitteena on, että tallennustapa vaikuttaisi mahdollisimman vähän kohdealueluokkiin.

Kohdealueluokkien tietojen tallentaminen tietokantaan edellyttää, että luokkien pohjalta luodaan määritykset oliomallia vastaaville tietokantatauluille. Perussääntönä on, että yksittäiset luokat vastaavat relaatiokannan tauluja ja attribuutit kenttiä – kuitenkin usein niin, että tietyn luokat tiedot jakaantuvat usealle taululle. Luokat voivat sisältää rakenteista ja monimuotoista tietoa, mutta relaatiomallin normalisointiperiaate vaatii, että taulujen rivien arvot ovat atomisia. Tietokantamääritykset voidaan määritellä joko lähtemällä luokissa olevista tiedoista ja normalisoimalla niitä tavanomaisen tietokantojen suunnittelun tapaan tai sitten käyttämällä muutamia oliospesifisiä muunnossääntöjä (luokat viedään tauluiksi, oliotunnisteet pääavaimiksi, kokoelmat, koosteattribuutit ja m-m-suhteet uusiksi tauluiksi jne), joiden avulla luokkamallista päästään relaatiomalliin.

Tietokantaan tehtävien hakujen käsittelyvastuu voidaan jakaa useilla eri tavoilla. Olk. esim. tietokantahaku (Bennett 2006), jossa halutaan hakea kansainvälisten kampanjoiden otsikot sijainnin mukaan:

```
SELECT campaignTitle
FROM Campaign c, InternationalCampaign ic, Location l
WHERE c.campaignCode = l.locationCode
AND ic.locationCode = l.locationCode
AND l.locationName='Hong Kong'
```

1. SQL-lause voidaan ajaa sellaisenaan (=yo. Lauseen osalta sijainnin mukaan parametrisoituna) ja kannasta palautetut arvot sijoitetaan vastaaville olioille. Tämä on suorituskyvyltään tehokkain tapa, mutta sovelluslogiikkaan sekoittuu SQL-koodia, jota täytyy ylläpitää esim. kohdealueen muuttuessa.
2. Palautetaan kannasta kaikki kyselyn tauluihin (Campaign, InternationalCampaign, Location) liittyvät oliot ja jätetään taulujen liitoksista ja hakuehdoista huolehtiminen kohdealuemallille. Ylläpidettävä ja oliopohjainen tapa, mutta suorituskyky- ja muistivaatimuksiltaan epärealistinen laajoilla tietomalleilla, koska suuri osa kohdealueolioiden tiedoista täytyisi pitää kerralla muistissa.
3. Palautetaan kannasta kyselyä vastaavat tiedot taulu kerrallaan navigoimalla kohdealuemallia ja tarkentaen hakuehtojen mukaan. Oliopohjainen tapa ja muistivaatimuksiltaan kohtuullisempi verrattuna 2. vaihtoehtoon, mutta vaatii tavanomaista useampien indeksien määrittelyä tietokannassa liitosten tehokasta hakua varten (kyselyjen nopeuttamiseksi indeksien määritys täytyy joka tapauksessa huomioida tietokantasuunnittelussa riippumatta tietojen käsittelyvastuusta). Suorituskyky on huonompi kuin 1. vaihtoehdossa, koska järjestelmä ottaa vastuulleen normaalisti tietokantapalvelimen vastuulla olevan kyselyjen optimoinnin ja välitulosten käsittelyn.

Lopuksi arvioidaan erilaisia vaihtoehtoja tiedonhallinnan toteutukseen (Bennett 2006).

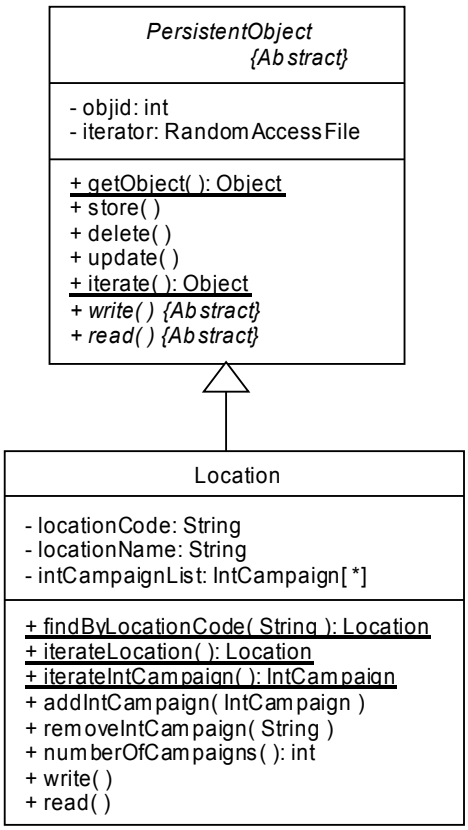
1. Lisää kohdealueluokkiin lataus- ja tallennusmetodit

Kohdealueluokkiin liittyviä tai luokkakohtaisia lataus- ja tallennusmetodeja käytetään lähinnä tilanteessa, jossa pysyvyyttä hallitaan pelkästään tiedostoilla. Edellisen luvun arvion mukaisesti ratkaisu ei skaalaudu useille käyttäjille tai laajoihin tietomassoihin. Lisäksi luokkien uudelleenkäyttömahdollisuudet vähenevät, kun luokassa sitoudutaan kiinteästi tiettyyn tallennusratkaisuun.

2. Tee latauksesta ja tallennuksesta luokkakohtaisia (staattisia) metodeja

Luokkakohtaisten lataus- ja tallennusmetodien (2) etuna oliokohtaisiin on lähinnä uusien olioiden luonnissa, koska staattista metodia käytettäessä ”tyhjän” olion ei tarvitse ladata itseään, vaan staattinen metodi voi luoda olion osana latausmetodia – uudelleenkäyttöön tai skaalautuvuuteen liittyvät perusongelmat ovat kuitenkin samat.

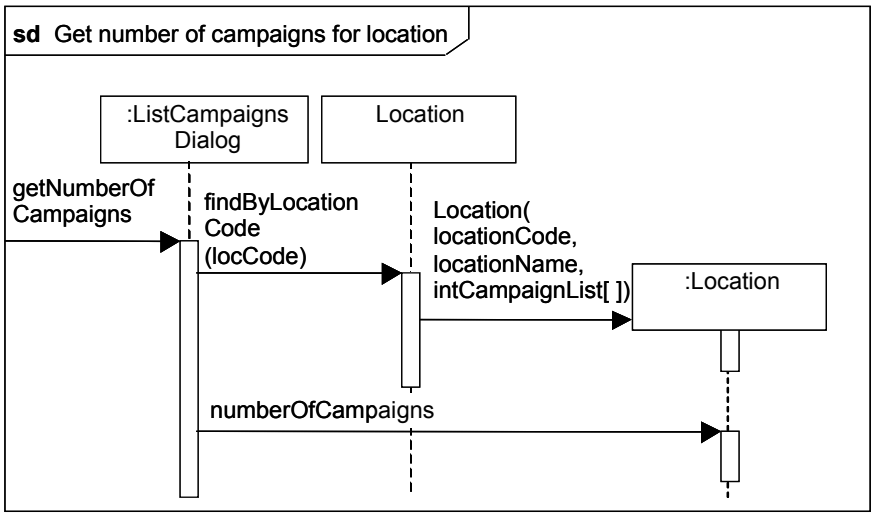
3. Peri pysyväisluonteiset oliot PersistentObject-yliluokasta



Kuva 5.63: Tiedonhallintatoimintojen määritys yhteisessä yliluokassa

Yksinkertainen tapa vähentää (mutta ei poistaa) kohdealueluokkien riippuvuutta tallennusmekanismista on toteuttaa tietokantaan liittyvät yleiset käsittelytoiminnot (haku, tallennus, poisto) yhteiseen yliluokkaan (aiemmin mainituissa tavoissa oletet-

tiin, että perintää ei käytetä). Latauksen ja tallennuksen yksityiskohdat aliluokkien pitää kuitenkin toteuttaa itse (kuva 5.63; Bennett 2006). Uuden olion haku ja olioiden iterointi eri hakujen mukaan on toteutettu luokkakohtaisina metodeina, mikä yksinkertaistaa sekvenssikaavioiden mallinnusta (kuva 5.64; Bennett 2006), koska hakuoperaatioita ei tarvitse osoittaa millekään tietylle oliolle (vrt. luvun 5.3.1.4 dokumenttikantaesimerkki).



Kuva 5.64: Yhteiseen yliluokkaan perustuvan tiedonhallinnan käyttö. Hakumetodi on toteutettu luokkakohtaisena, uusi Location-olio luodaan hakutulosten perusteella.

Yhteisen yliluokan käyttö tietokantayhteyden määrittelyssä on hyväksyttävä ratkaisu, jos ei ole syytä olettaa käytettävän tietokannan muuttuvan tai kohdealueluokkia ei tarvitse uudelleenkäyttää osana muihin tietokantoihin liittyviä järjestelmiä. Edellisten oletusten ollessa voimassa etuna voidaan pitää myös sitä, että kohdealueluokat osaavat ladata ja tallentaa itsensä, eikä muilla kerroksilla olevien luokkien tarvitse ottaa kantaa tallennustekniikkaan. Jos PersistentObject-yliluokkaan on koottu kunnollinen kirjasto tietokannan käsittelymetodeja, aliluokkien *read()*- ja *write()*-metodien toteutukset voivat olla suoraviivaisia. Ratkaisun ongelmana on kuitenkin kohdealueluokkien kytkeytyminen ”apuluokkaan” (PersistentObject) ja tiettyyn tietokantaan. Kohdealueluokat sisältävät tällöin sekä sovellusalueen yleistä tietämystä että pysyvyyden hallintaan liittyvää yksittäiseen sovellukseen liittyvää tietoa, mikä heikentää niiden uudelleenkäyttöä.

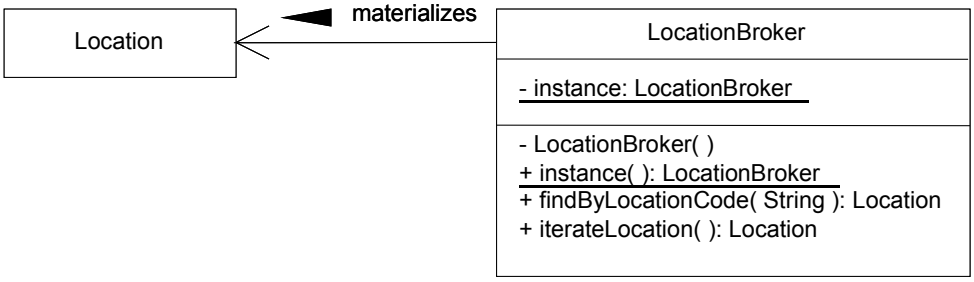
4. Hallinnoi pysyvyyttä kokoelmaluokkien avulla

Suunnitteluvaiheessa määriteltävien kokoelmaluokkien käyttö on yhteistä yliluokkaa parempi ratkaisu, koska luokat eivät kuulu kohdealuemalliin. Ratkaisun heikkouksia ovat lähinnä tallennusratkaisun sitominen suunnitteluvaiheen malliin (esim. jos kokoelmaluokkia halutaan uudelleenkäyttää toisella tallennuratkaisulla) ja se, että kokoelmaluokkiin tulee vahva riippuvuus kohdealueluokista.

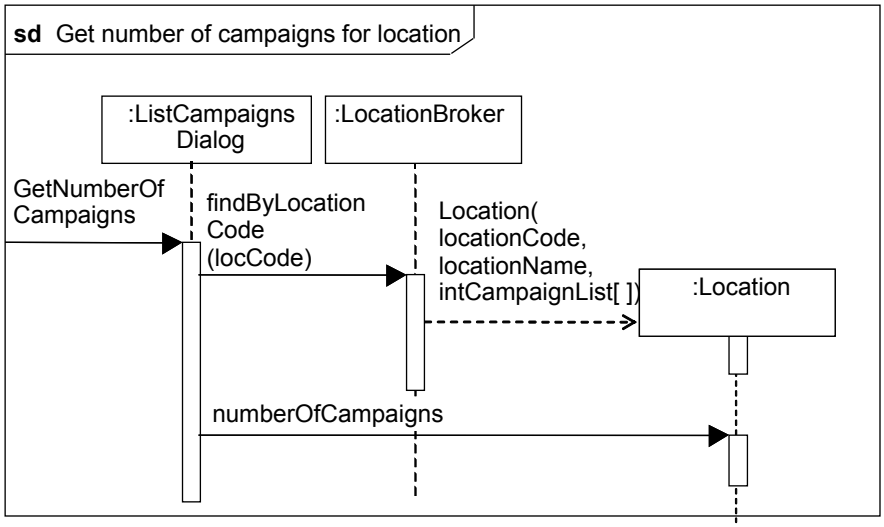
5. Hallinnoi pysyvyyttä omassa ”kerroksessaan” välitinluokkien (broker) *avulla*

Välitinluokat ovat kohdealueluokista erillisiä luokkia, jotka vastaavat tiedonhallinnasta. Jokaista kohdealuemallin luokkaa vastaa oma välitinluokka, joka sisältää oman ”kohdeluokkansa” lataukseen, tallennukseen ja iterointiin liittyvät metodit (kuva

5.65). *PersistentObject*-ratkaisun tyyliin välitinluokilla voi olla yhteinen yliluokka tiedonhallintatoimintojen yleistä määrittelyä varten. Koska välitinluokkia käytetään vain muiden olioiden luontiin, jokaisesta välitinluokasta luodaan vain yksi olio (Singleton-suunnittelumalli[39]). Myös hakutoimintojen käyttö muistuttaa muiden kerrosten kannalta *PersistentObject*-ratkaisua, mutta nyt sovelluslogiikkakerros hakee kohdealueolioita Broker-luokan kautta, ei luokalla itsellään (kuva 5.66).



Kuva 5.65: Välitin-malli erottaa tallennusmekanismin kohdealueluokista



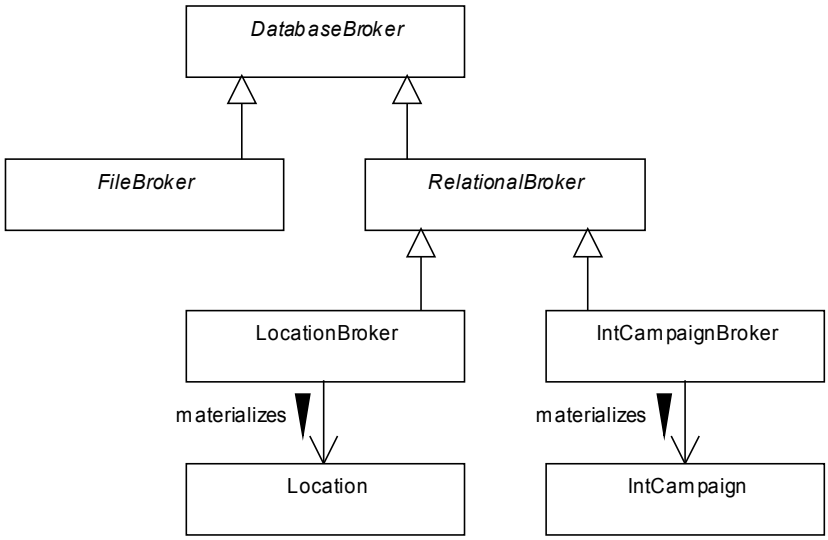
Kuva 5.66: Välitinpohjaisen tiedonhallintaratkaisun käyttö (vrt. kuvaan 5.64)

Välitinpohjainen ratkaisu mahdollistaa tallennustavan muuttamisen kohdealueluokista riippumatta. Kuvassa 5.67 on esitetty luokkahierarkia, jossa yleisestä välitinluokasta on peritty tiedosto- ja tietokantapohjaista käsittelyä varten omat välittimet. Tallennusratkaisua muutettaessa yksittäisiin kohdealueluokkiin liittyviin välitinluokkiin pitäisi kuitenkin todennäköisesti tehdä muutoksia.

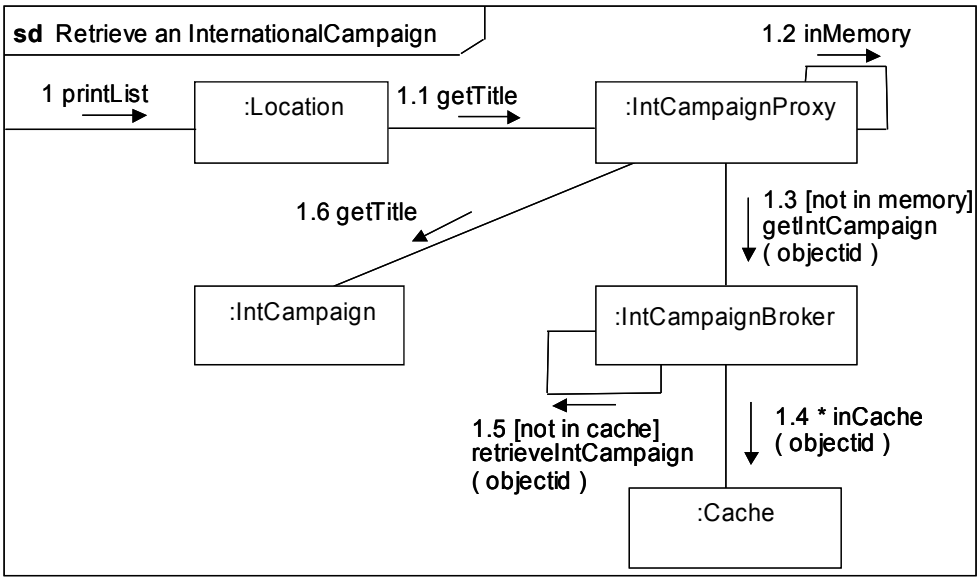
Välitinluokkiin perustuvan tiedonhallintamekanismin toteutus edellyttää lisäksi, että viitteet muihin olioihin huomioidaan erillisillä edustajaluokilla (Proxy-suunnittelumalli[40]) ja oliot säilötään tietokannasta lukemisen ja kirjoittamisen välillä välimuisteihin. Edustajaluokat mahdollistavat kohdealueluokkaan liittyvien tietojen haun osissa, mikä vähentää muistissa pidettävien olioiden määrää. Välimuistien avulla voidaan pitää kirjaa olioista, jotka ovat vasta luotuja, muokattuja tai poistettuja. Muutokset voidaan viedä tietokantaan hallitusti (esim. vasta transaktion hyväksymisen jälkeen).

[39] http://en.wikipedia.org/wiki/Singleton_pattern

[40] http://en.wikipedia.org/wiki/Proxy_pattern



Kuva 5.67: Abstrahoimalla yleiset lataus-, tallennus- ja hakumekanismit tiedonhallintatapa voidaan vaihtaa kohdealueluokista riippumatta.



Kuva 5.68: Välitinluokkien käyttö edustajaluokkien (proxy) kanssa tilanteessa, jossa viitattua oliota ei ole haettu välimuistiin (cache). Edustajaolio näkyy Location-oliolle kampanjaoliona, joka haetaan tietokannasta vasta, kun oliota todella tarvitaan (luetaan tai muutetaan tietoja). Tietokannasta haetut oliot tallennetaan välimuistiin.

Välitinluokkia käyttävä mekanismi eristää kohdealueluokat kokonaan tiedonhallinnasta oliopohjaisella tavalla. Ratkaisun ongelmana on kuitenkin toteutuksen monimutkaisuus ja tarvittavien luokkien suuri määrä – kohdealuemallin ominaisuuksia täytyy viedä sekä välitin- että edustajaluokkiin (joista osa on tosin generoitavissa automaattisesti, jos tallennusmekanismi on riittävän yleiskäyttöinen). Ratkaisu on ongelmallinen myös tilanteessa, jossa kohdealuemalli ei ole vakiintunut: jokaista kohdealuemallin muutosta kohti täytyy muuttaa useita tiedonhallintaluokkia.

6. *Käytä parametrisoituja luokkia* (=geneerisiä luokkia, luokkamalleja) *pysyvyyden hallinointiin eri luokilla*

Tämä on hienostunein ratkaisu, jonka avulla voidaan välttää kohdealuemallin ”kahdentuminen” (samaa luokkamallia voidaan kohdealuemallin eri luokkien tallennukseen). Ratkaisu on kuitenkin oleellisesti muita hankalampi toteuttaa, koska tallennusmekanismista pitää saada tarpeeksi yleiskäyttöinen (mm. indeksoidut ja tyypeiltään vaihtelevat kentät, m-m -suhteiden ja muiden viitteiden hallinta, tietojen validointi uusia olioita luotaessa, tyhjien viitteiden korjaus/uudelleenohjaus tuhoamisen yhteydessä jne). Lisäksi malliluokissa käytetyt parametrit pitää ”konfiguroida” varsinaisten tiedonhallintaluokkien ulkopuolella (mitkä kohdealueluokkien attribuutit vastaavat mitäkin tietokannan kenttiä).

7. *Käytä erillistä pysyvyyden hallintaan erikoistunutta kirjastoa tai sovelluskehystä*

Tiedonhallintamekanismia ei välttämättä kannata suunnitella itse, ellei esim. suorituskykysyistä ole syytä kontrolloida tietokantakyselyjä tarkasti. Koska tiedonhallintaongelmat toistuvat samankaltaisina eri järjestelmissä on saatavilla useita valmiita kirjastoja tietokannan abstrahointiin (erityisesti oliomallin muuttamiseen relaatiomalliksi). Tekniikan yleisnimitys on *Object-Relational mapping* (ORM) ja eri toteutusten yksityiskohdat vaihtelevat, mutta yleensä muunnoskirjastoon pitää kuvata kohdealuemallin rakenne (esim. xml-kuvauksilla tai ohjelmointikielen omalla metadatalla), jonka jälkeen kirjasto huolehtii suorasta kommunikoinnista tietokannan kanssa. Haku- ja päivitysoperaatioita varten kirjasto saattaa sisältää oman SQL-tyylisen kyselykielen. Java-ympäristössä suosittuja pysyvyyden hallintaan erikoistuneita kirjastoja ovat JDO[41] (Java Data Objects)-rajapintaa tukevat kirjastot sekä Hibernate[42]-kirjasto. Spring[43] on esimerkki pysyvyyden hallintaa helpottavasta sovelluskehyksestä, joka vähentää kohdealueen olioiden konfigurointitarvetta ja helpottaa yksittäisten kirjastojen käyttöä tukien sekä Hibernatea että JDO:ta.

Toinen lähestymistapa (joka ei sinänsä sulje pois erillisten kirjastojen käyttöä) pysyvyyden hallintaan ovat hajautetut komponenttiarkkitehtuurit, kuten Java Enterprise Editionin (kuva 1.9) EJB-määritys (Enterprise JavaBeans). EJB:n mukaan määritellyt kohdealueluokat (Entity Beans) voivat vastata pysyvyyden hallinnastaan joko itse (Bean-managed persistence) tai sitten sovelluspalvelimen toimesta (Container-managed persistence – EJB 3.0-määrityksessä käytetään Java Persistence API:a). Komponenttiarkkitehtuuri tarjoaa pysyvyyden lisäksi myös muita hajautetuissa yrityssovelluksissa usein tarvittavia palveluita (hajautetut oliot, olioiden transaktiot, kuormituksen tasaus, hakemistopalvelu, viestinvälitys, tunnistautuminen jne). Yleiskäyttöisyyden takia yksinkertaisenkin sovellus saattaa vaatia runsaasti rajapinnan vaatimia luokkamäärityksiä ja konfigurointitietoja.

[41] http://en.wikipedia.org/wiki/Java_Data_Objects

[42] https://www.hibernate.org/

[43] http://www.springsource.org/

KIRJALLISUUTTA

Alexander ym., A pattern language, Oxford University Press, 1977.

Awad M., Kuusela J., Ziegler J., Object-oriented technology for real-time systems - a practical approach using OMT and Fusion, Prentice-Hall, 1995.

Bennett S., McRobb S., Farmer R., Object-oriented systems analysis and design using UML, McGraw-Hill, 1999.

Bennett S., McRobb S., Farmer R., Object-oriented systems analysis and design using UML, Third Edition, McGraw-Hill 2006.

Berard E., Essays on object-oriented software engineering, Vol. 1, Prentice-Hall, 1993.

Booch G., Object-oriented design with applications, The Benjamin/Cummings Pub., 1991.

Booch G., Rumbaugh J., Unified method for object-oriented development, Document Set, version 0.8, 1995.

Booch G., Jacobson I., Rumbaugh J., The unified modelling langrage user guide, Addison-Wesley, 1999.

de Champeaux D., Lea D., Faure P., Object-oriented system development, Addison-Wesley, 1993.

Coad P., Yourdon E., Object-oriented analysis, Yourdon Press, 1991.

Coad P., Yourdon E., Object-oriented design, Yourdon Press, 1991.

Coad P., North D., Mayfield M., Object models - strategies, patterns & applications, Yourdon Press, 1995.

Cockburn A., Writing effective use cases, Addison-Wesley, 2000.

Coleman D., Arnold P., Bodoff S., Dollin C., Gilchrist H., Hayes F., Jeremiaes P., Object-oriented development: the Fusion Method, Prentice-Hall, 1994.

DeMarco T., Structured analysis and systems specifications, Yourdon Press, 1978.

Dock P., OOD: research or ready, Hotline on Object-oriented Technology, Vol. 3, No. 9, 1992.

Drake J., Tsal W., Lee H., Object-oriented analysis: criteria and case study. Intelnational Journal of Software Engineering and Knowledge Engineering, Vol. 3, No. 3, 1993.

D'Souza F., Wills A., Objects, components, and f'rameworks with UML - The catalyst approach, Addison-Wesley, 1999.

Embley D., Kurtz B., Woodfield S., Object-oriented systems analysis: a model-driven approach, Yourdon Press, 1992.

Fayad M., Tsai W.-T., Fulghum M., Transition to object-oriented software development, Comm. of the ACM., Vol 39, No. 2, 1996.

Fowler M., Scott K., UML distilled - appearing the standard object modelling language, Addison-Wesley, 1997.

Gamma E., Helm R., Johnson R., Vlissides J., Design patterns: elements of reusable object-oriented software, Addison-Wesley, 1995.

Graham I., Object-oriented methods: a practical introduction, Addison-Wesley, 1991.

Graham I., Henderson-Sellers B., Younessi H., The OPEN process specification, Addison-Wesley, 1997.

Harmon P., Watson M., Understanding UML - the developer's guide with a web-based application in Java, Morgan Kaufmann, 1998.

Hutt A., Object analysis & design: comparison of methods, Wiley & Sons, 1994.

Jaaksi A., Implementing interactive applications in C++, Software - Practice and Experience, March 1995.

Jaaksi A., Object-oriented specification of user interfaces, Software - Practice and Experience, November 1995.

Jaaksi A., Aalto J.-M., Aalto A., Vättö K., Tried & true object development - industry-proven approaches with UML, Cambridge University Press, 1999.

Jacobson I., Christerson M., Jonsson P., Övergaard G., Object-oriented software engineering, Addison-Wesley, 1992.

Jacobson I., Booch G., Rumbaugh J., The unified software development process - the complete guide to the unified process from the original designers, Addison-Wesley, 1999.

Koskimies K., Olio-ohjelmointi ja oliokielet, Raportti C-1994-2. Tietojenkäsittelyopin laitos, Tampereen yliopisto, 1994.

Kruchten P., The Rational Unified Process: An Introduction, 2nd Edition, Addison-Wesley, 2000.

Lee R., Tepfenhart W., UML and C++ - a practical guide to object-oriented development, Prentice-Haall, 1997.

Lorenz M, Object-oriented software development: a practical guide, Prenctice-Hall, 1993.

Martin J., Odell J., Object-oriented analysis and design, Prentice-Hall, 1991.

Monarchi D., Puhr G., A research typology for object-oriented analysis and design, Comm. of the ACM, Vol. 39, No. 2, March 1993.

Pree W., Design patterns for object-oriented software development, Addison-Wesley, 1995.

Priestly M., Practical object-oriented design, McGraw-Hill, 1997.

Rosenberg D., Scott K., Use Case Driven Object Modeling with UML: A Practical Approach, Addison-Wesley, 1999.

Rosenberg D., Stephens M., Use Case Driven Object Modeling with UMLTheory and Practice, Apress, 2007.

Rumbaugh J., Jacobson I., Booch G., The unified modeling language reference model, Addison-Wesley, 1999.

Rumbaugh J., Blaha M., Premerlani W., Eddy F., Lorensen W., Object-oriented modeling and design, Prentice-Hall, 1991.

Shlaer S., Mellor S., Object lifecycles: modelling the world in states, Prentice-Hall, 1992.

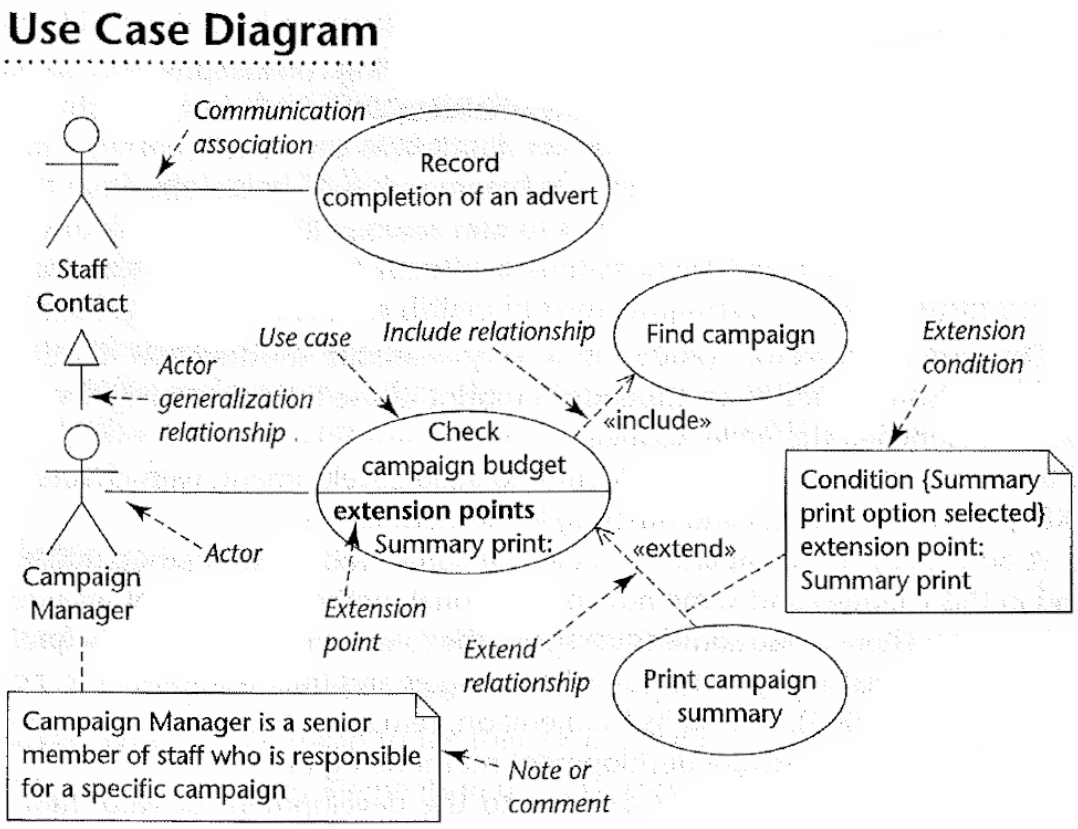
Schultz R., A comparison of object-oriented development methodologies, 2nd Edition, Berard Software Engineering 1992.

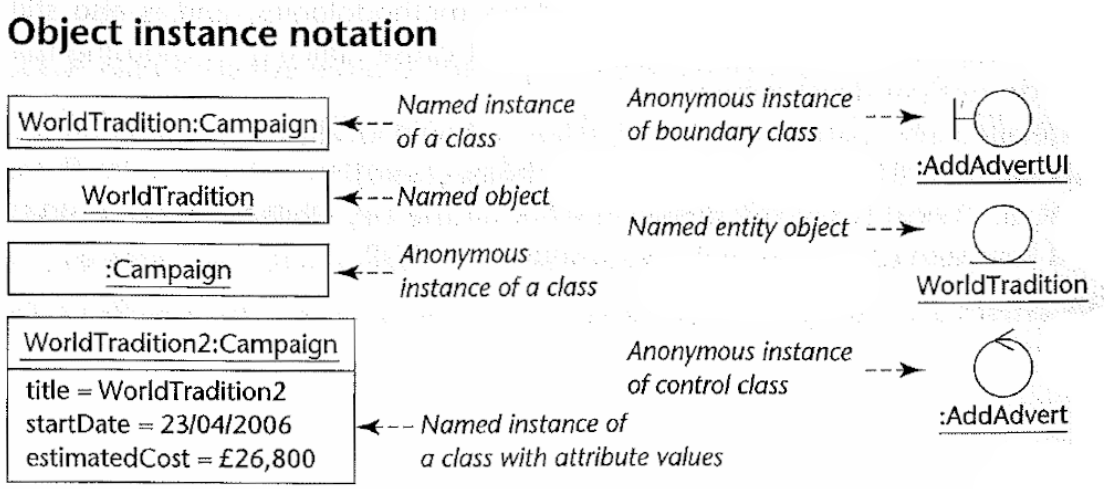
Tempfenhart W., Cusick J., A unified object topology, IEEE Software, January, 1997.

Wirfs-Brock R., Wilkerson B., Wiener L., Designing object-oriented software, Prentice-Hall, 1990.

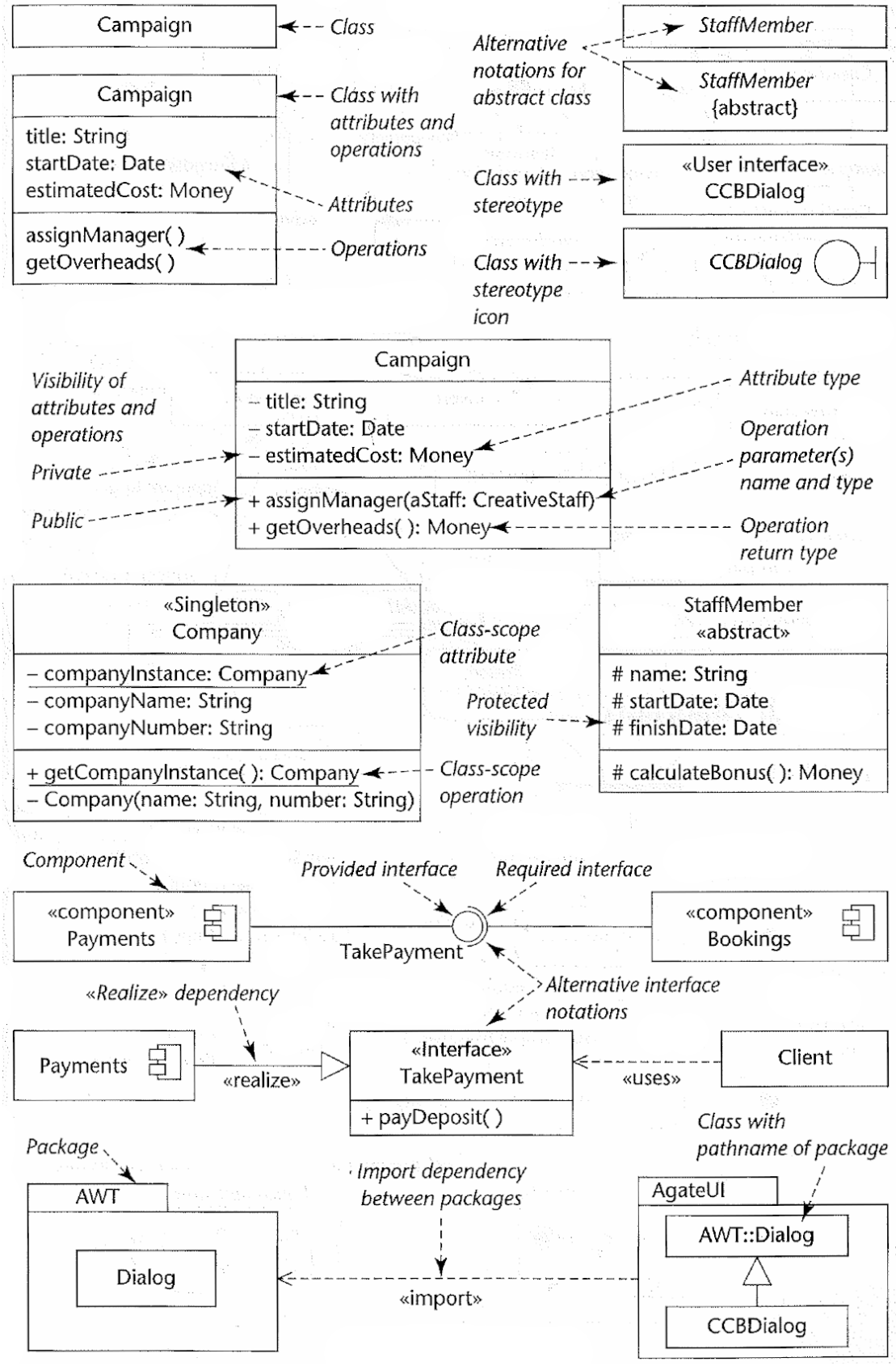
LIITE A. UML 2.0 -NOTAATIO

Lähde: Bennett ym. (2006). Interaktio-yleiskaavio, ajoituskaavio ja sekvenssikaavio aikarajoitteisilla viesteillä jätetty pois.

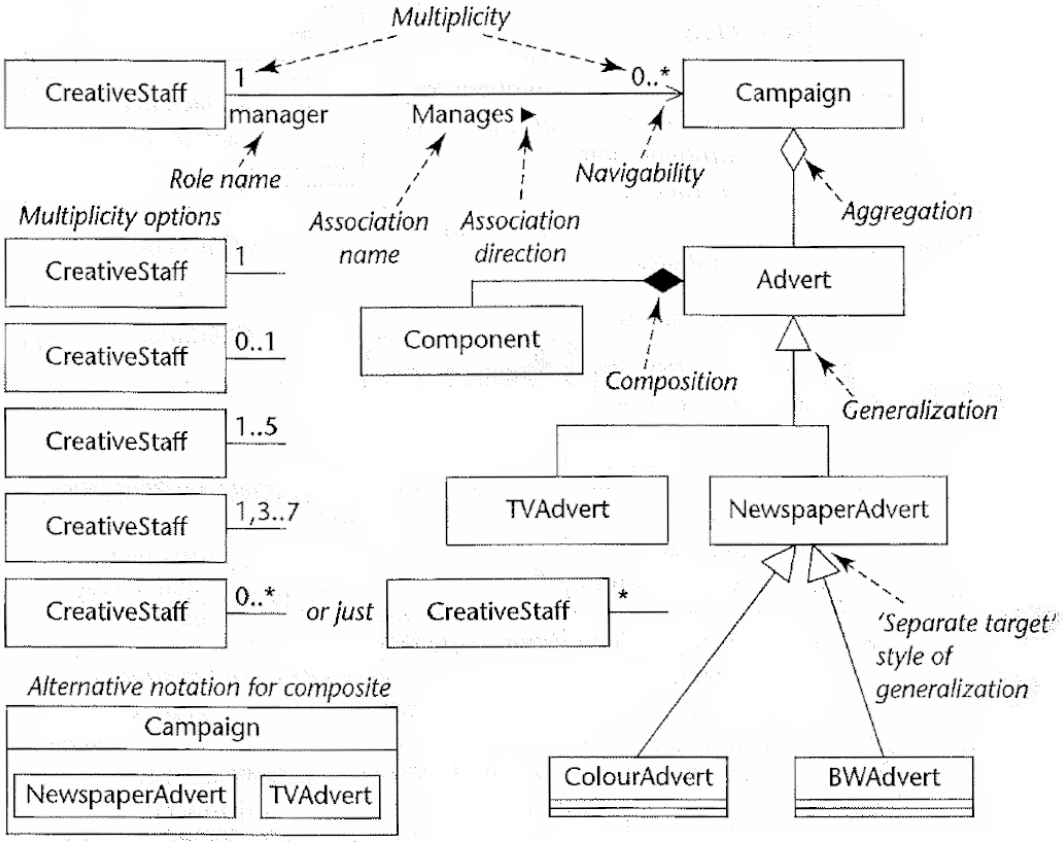




Class notation

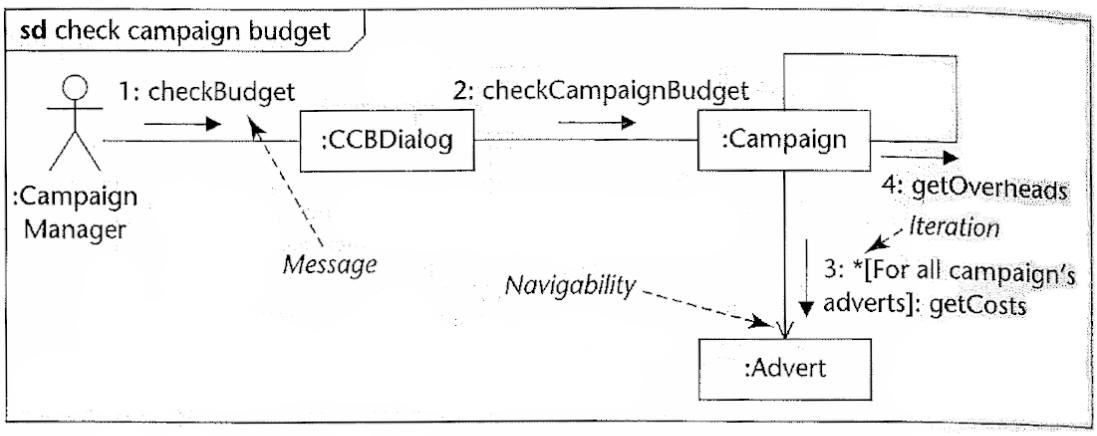


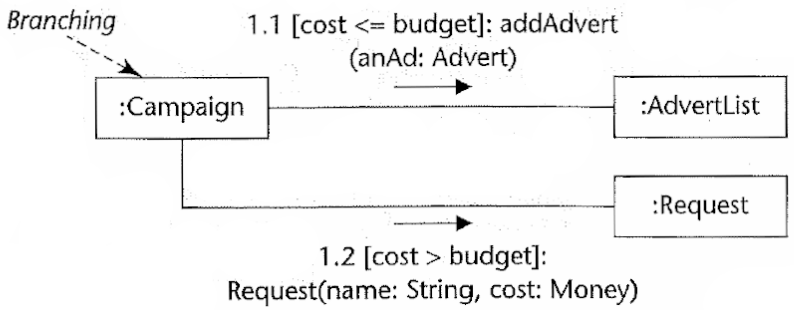
Associations



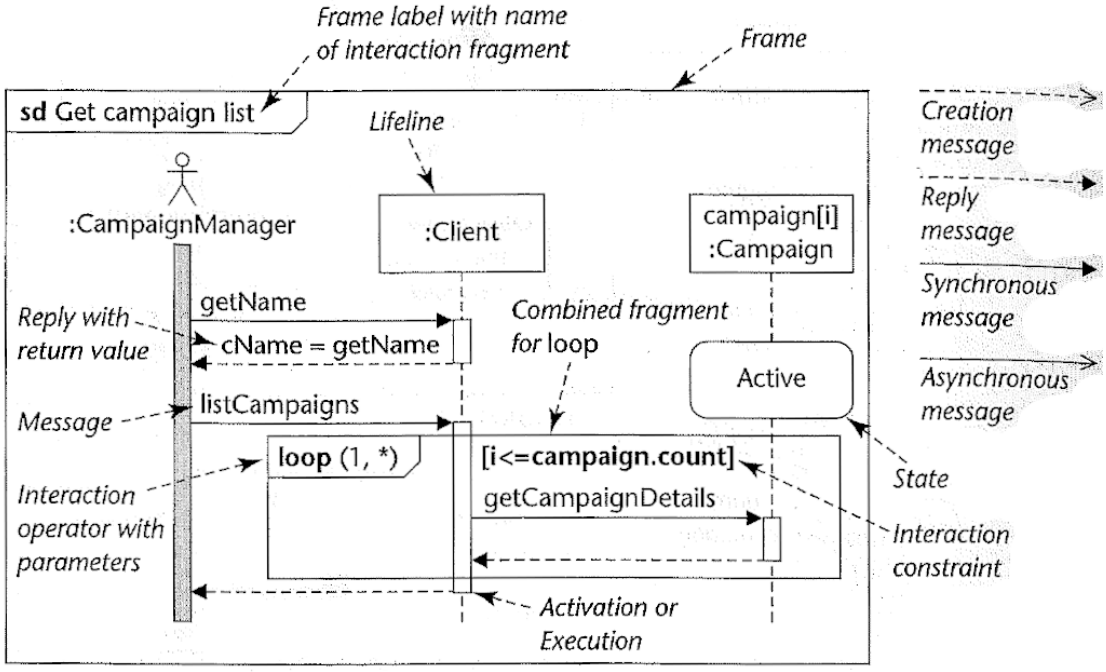
Interaction Diagrams

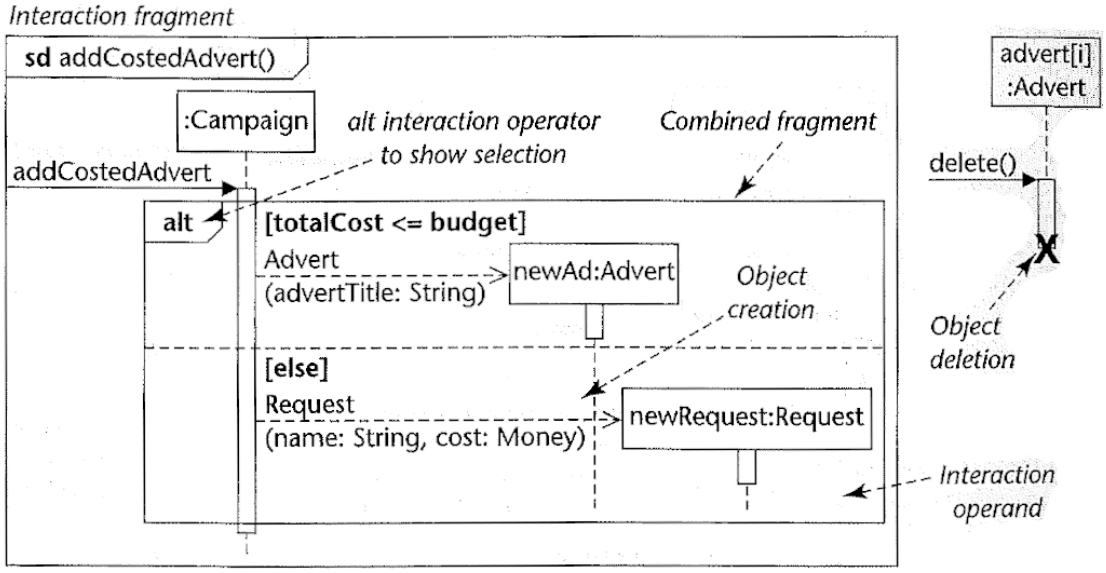
Communication diagram





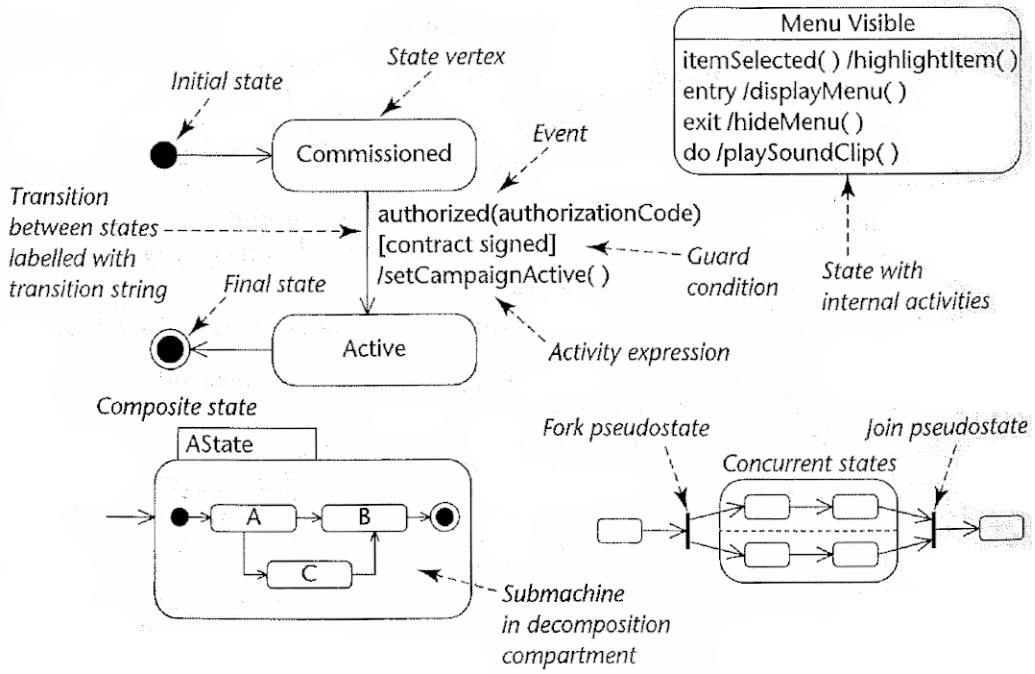
Sequence diagram



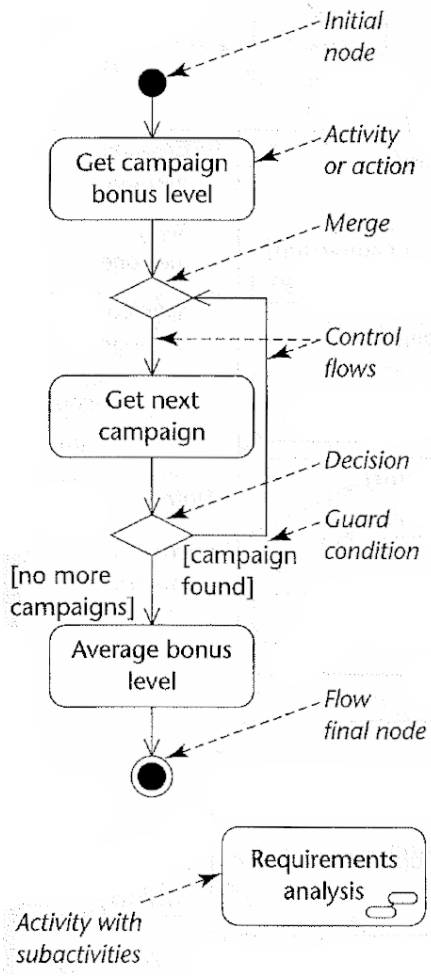


Behaviour Diagrams

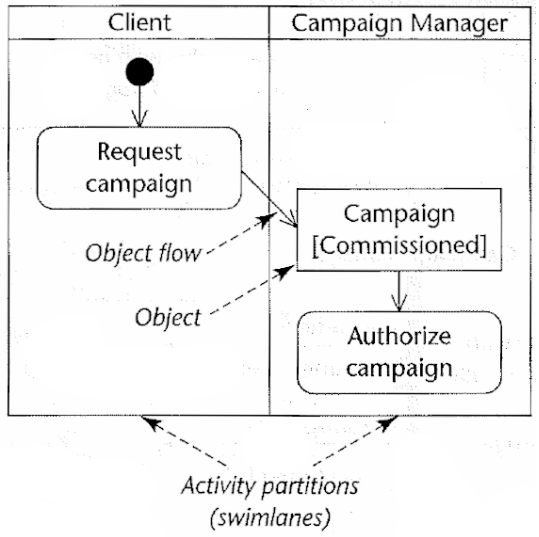
State Machine

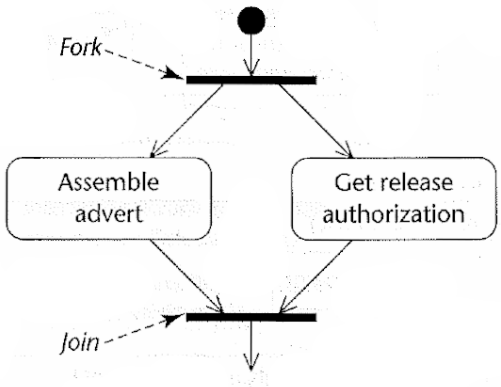


Activity diagram



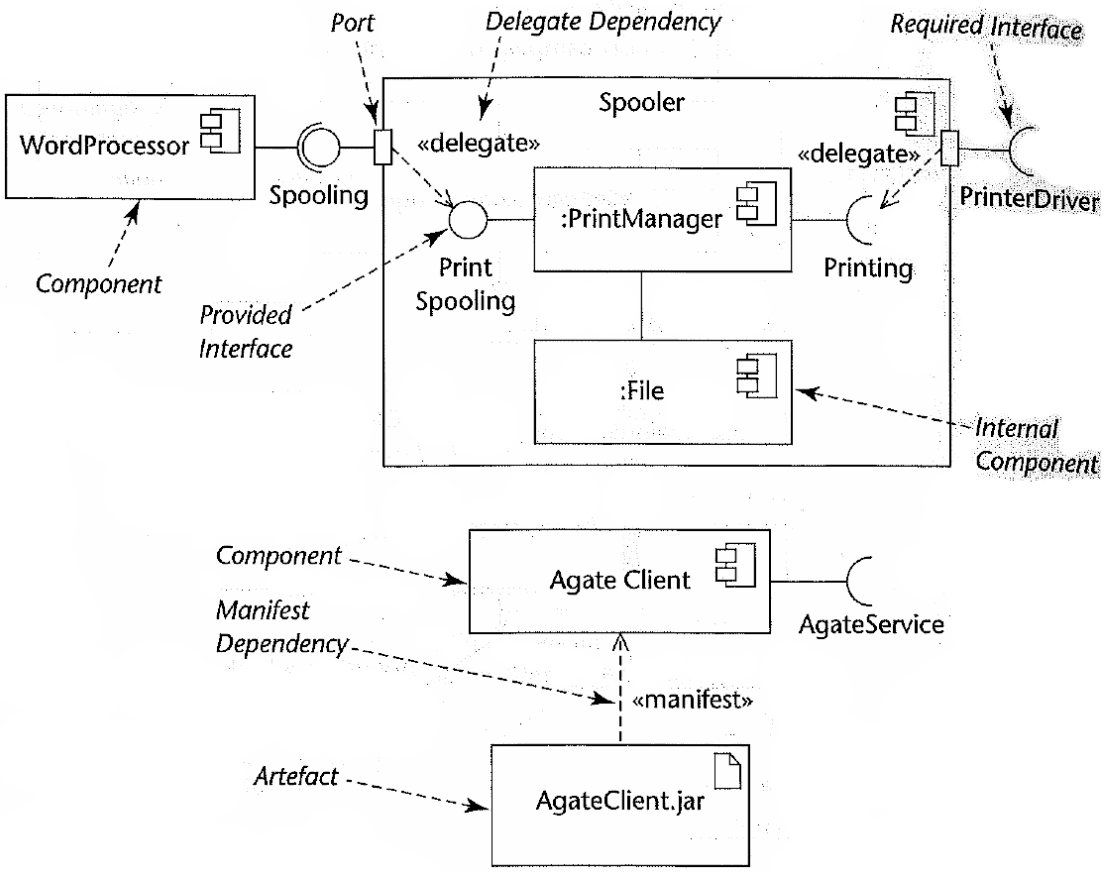
Activity diagram with activity partitions



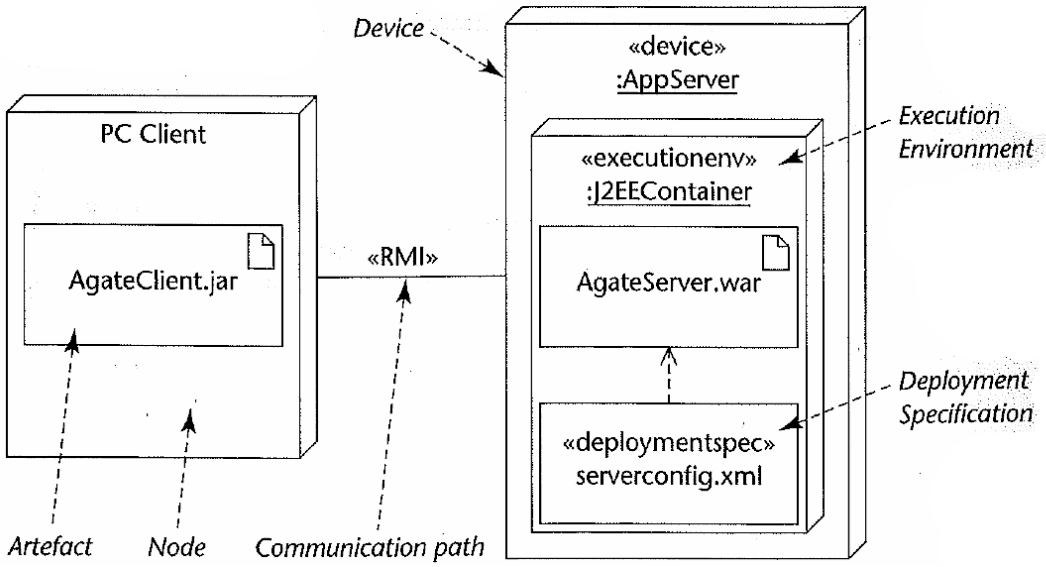


Implementation Diagrams

Component diagram



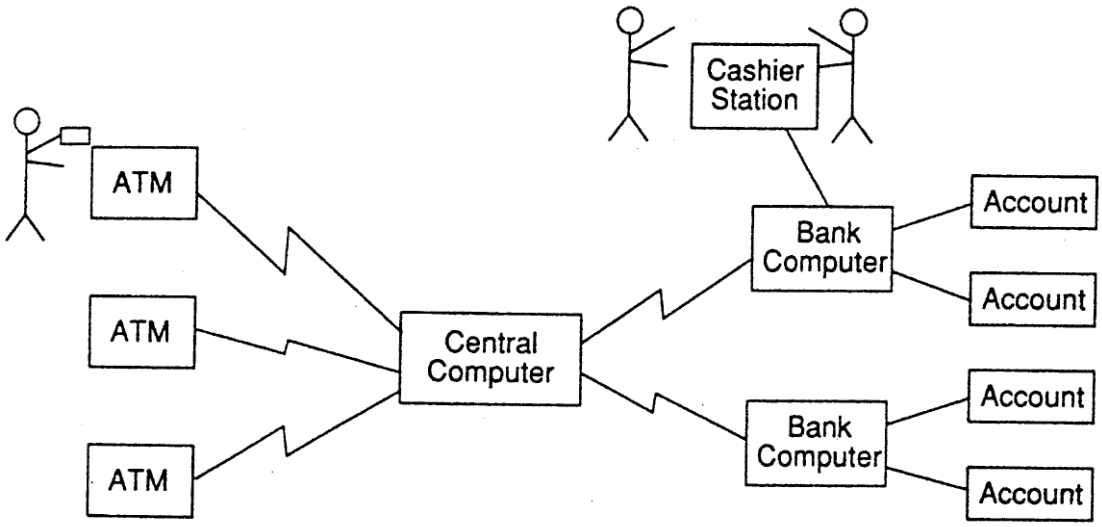
Deployment diagram



LIITE B. ESIMERKKI: PANKKIAUTOMAATTIJÄRJESTELMÄ

Lähde (pl. käyttötapaukset): Rumbaugh ym. (1991). Luokkakaaviot muutettu OMT:stä UML-notaatioon.

Suunniteltava ohjelmisto, joka tukee kassoista ja pankkiautomaateista (ATM, Automated Teller Machine) muodostuvaa verkkoa. Kassat ja pankkiautomaatit kuuluvat liittoutuman muodostaneille pankeille. Kunkin pankin oma tietokone pitää kirjaa omista tileistä ja huolehtii tapahtumista eli transaktioista. Kassojen työasemat kommunikoivat suoraan oman pankin tietokoneen kanssa. Kassanhoitajat tekevät kirjauksia tileistä ja tapahtumista. Pankkiautomaatit ovat yhteydessä keskuskoneeseen, joka ohjaa tapahtumat ao. pankin koneen käsiteltäviksi. Pankkiautomaattia voi käyttää pankkikortin avulla. Automaatti suorittaa tapahtumia (keskuskoneen avustuksella) käyttäjän antamien ohjeiden mukaan, antaa rahaa ja tulostaa kuitin. Järjestelmän tulee tarjota asianmukainen kirjanpito ja suojaus sekä mekanismi saman tilin rinnakkaisen käsittelyn hallintaan. Kullakin pankilla on ohjelmat tiliensä käsittelyyn. Tehtävänä on suunnitella ohjelmisto pankkiautomaatteja ja verkkoliikennettä varten. Järjestelmän kustannukset jaetaan pankkien kesken pankkikortteja omistavien asiakkaiden määrän mukaisessa suhteessa.



Kuva B.1: ATM-verkko

ATM USE CASE—FUNCTION ACCESS
(REUSED FOR OTHER USE CASES)

A machine is available outside the *bank* for *customers* to perform typical *teller* functions. *Customers* have *cards* that have a unique *identification* and *password* that they insert into a slot in a *card reader.* The *customer* is prompted for the *password* that is enscribed on the *card*. If the two match, the *customer* is given a menu choice of actions on the *screen*.

If the two do not match for *three* times in a row, the *card* is kept by the *card reader* and the *customer* is shown a *message* on the screen.

ATM USE CASE—
CASH WITHDRAWAL

<Function Access Use Case prerequisite> If the *withdrawal button* is pressed, the *customer* is asked for the *account* type and an *amount* of money to be *withdrawn*. The *money dispenser* then dispenses the requested amount of money. A *receipt* is then *printed* which contains the *account name* and *number,* date, and amount of money *withdrawn* from the *account*. The *customer* is allowed to then request another action, or to exit.

<Variation—$200 maximum withdrawal per day> If the *withdrawal* for a particular *account* would put the daily *amount* over the $200 limit for the day, a *receipt* is immediately *printed* with a *message* explaining why no money can be *dispensed*.

<Variation—Default withdrawal amount> If the *customer* picks the *default button,* then $30 is immediately *dispensed,* without asking for an amount.

ATM USE CASE—BALANCE INQUIRY

<Function Access Use Case prerequisite> If the *balance inquiry button* is pressed, the *customer* is asked for the *account* type, after which a *receipt* is *printed* which contains the *account name* and *number,* date, and amount of money in the *account*. The *customer* is allowed to then request another action, or to exit.

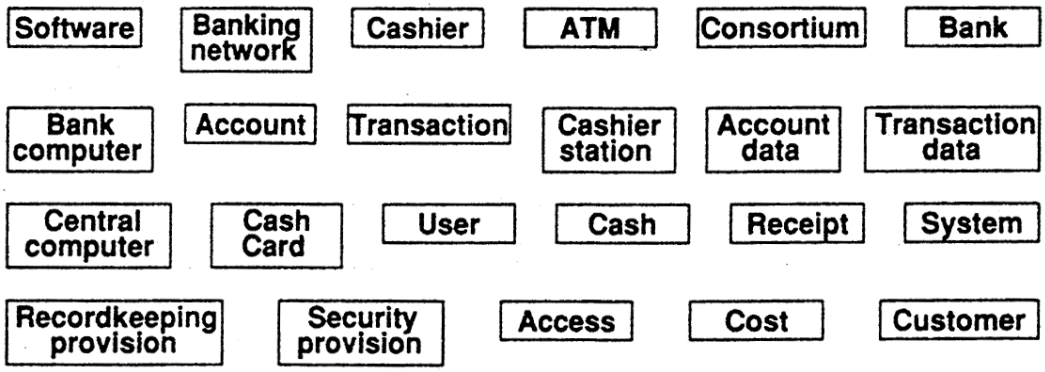
ATM USE CASE—CASH DEPOSIT

<Function Access Use Case prerequisite> If the *deposit button* is pressed, the *customer* is asked for the *account* type and an *amount* of money to be *deposited*. The *envelope slot* then opens, and the *customer* puts the *envelope* into it. A *receipt* is then *printed* which contains the *account name* and *number,* date, and amount of money *deposited* to the *account*. The *customer* is allowed to then request another action, or to exit.

ATM USE CASE—
ACCOUNT FUND TRANSFER

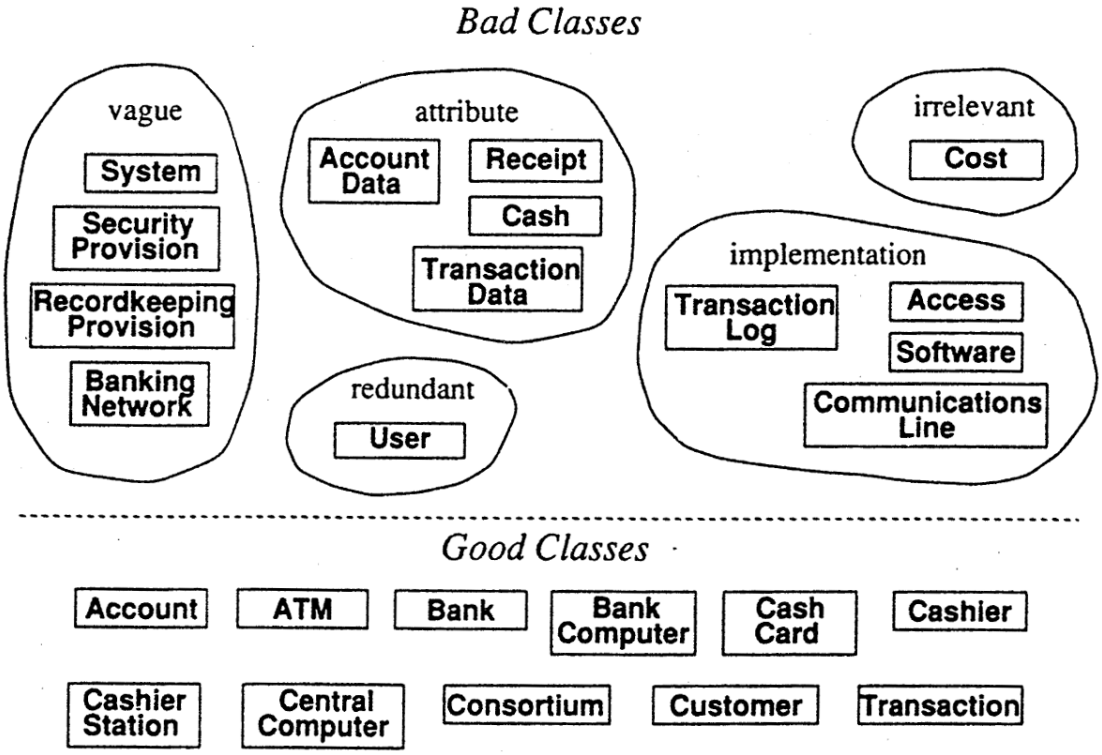
<Function Access Use Case prerequisite> If the *transfer button* is pressed, the *customer* is asked for the source *account* type and the target *account* type and an *amount* of money to be *transferred*. A *receipt* is then *printed* which contains the *account names* and *numbers,* date, and amount of money *transferred* between *accounts*. The *customer* is allowed to then request another action, or to exit.

Kuva B.2: Pankkiautomaatin käyttötapaukset (Lorenz 1993)



Kuva B.3: Aihekuvauksen pohjalta löydettyjä kandidaattiluokkia

Kuva B.4: Sovellusalueelta yleisesti tiedossa olevista käsitteistä tunnistettuja kandidaattiluokkia



Kuva B.5: Sopivien luokkien valinta kandidaattiluokista

Account—a single account in a bank against which transactions can be applied. Accounts may be of various types, at least checking or savings. A customer can hold more than one account.

ATM—a station that allows customers to enter their own transactions using cash cards as identification. The ATM interacts with the customer to gather transaction information, sends the transaction information to the central computer for validation and processing, and dispenses cash to the user. We assume that an ATM need not operate independently of the network.

Bank—a financial institution that holds accounts for customers and that issues cash cards authorizing access to accounts over the ATM network.

Bank computer—the computer owned by a bank that interfaces with the ATM network and the bank's own cashier stations. A bank may actually have its own internal network of computers to process accounts, but we are only concerned with the one that talks to the network.

Cash card—a card assigned to a bank customer that authorizes access of accounts using an ATM machine. Each card contains a bank code and a card number, most likely coded in accordance with national standards on credit cards and cash cards. The bank code uniquely identifies the bank within the consortium. The card number determines the accounts that the card can access. A card does not necessarily access all of a customer's accounts. Each cash card is owned by a single customer, but multiple copies of it may exist, so the possibility of simultaneous use of the same card from different machines must be considered.

Cashier—an employee of a bank who is authorized to enter transactions into cashier stations and accept and dispense cash and checks to customers. Transactions, cash, and checks handled by each cashier must be logged and properly accounted for.

Cashier station—a station on which cashiers enter transactions for customers. Cashiers dispense and accept cash and checks; the station prints receipts. The cashier station communicates with the bank computer to validate and process the transactions.

Central computer—a computer operated by the consortium which dispatches transactions between the ATMs and the bank computers. The central computer validates bank codes but does not process transactions directly.

Consortium—an organization of banks that commissions and operates the ATM network. The network only handles transactions for banks in the consortium.

Customer—the holder of one or more accounts in a bank. A customer can consist of one or more persons or corporations; the correspondence is not relevant to this problem. The same person holding an account at a different bank is considered a different customer.

Transaction—a single integral request for operations on the accounts of a single customer. We only specified that ATMs must dispense cash, but we should not preclude the possibility of printing checks or accepting cash or checks. We may also want to provide the flexibility to operate on accounts of different customers, although it is not required yet. The different operations must balance properly.

Kuva B.6: Pankkiautomaattijärjestelmän tietohakemisto

Verb phrases:

Banking network includes cashiers and ATMs
Consortium shares ATMs
Bank provides bank computer
Bank computer maintains accounts
Bank computer processes transaction against account
Bank owns cashier station
Cashier station communicates with bank computer
Cashier enters transaction for account
ATMs communicate with central computer about transaction
Central computer clears transaction with bank
ATM accepts cash card
ATM interacts with user
ATM dispenses cash
ATM prints receipts
System handles concurrent access
Banks provide software
Cost apportioned to banks

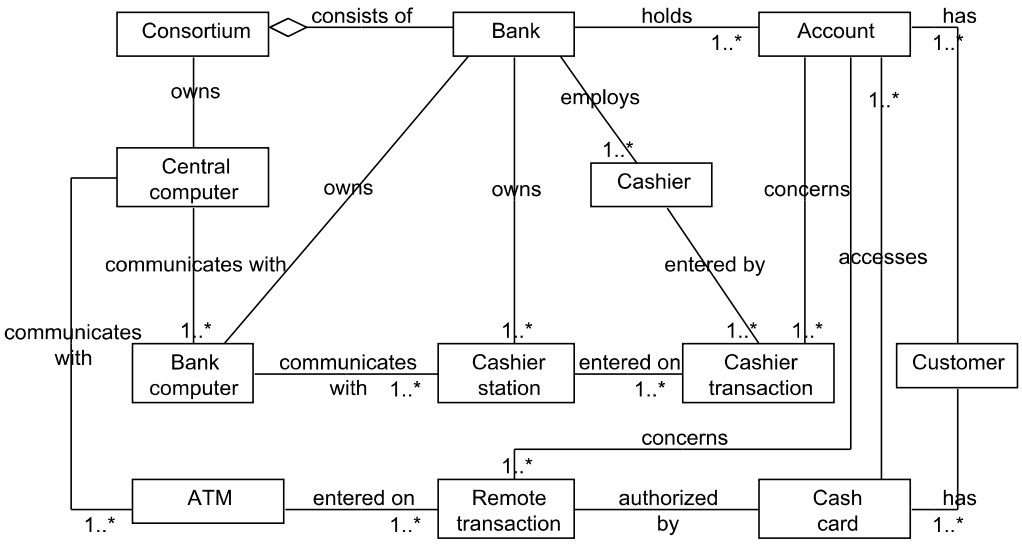
Implicit verb phrases:

Consortium consists of banks
Bank holds account
Consortium owns central computer
System provides recordkeeping
System provides security
Customers have cash cards

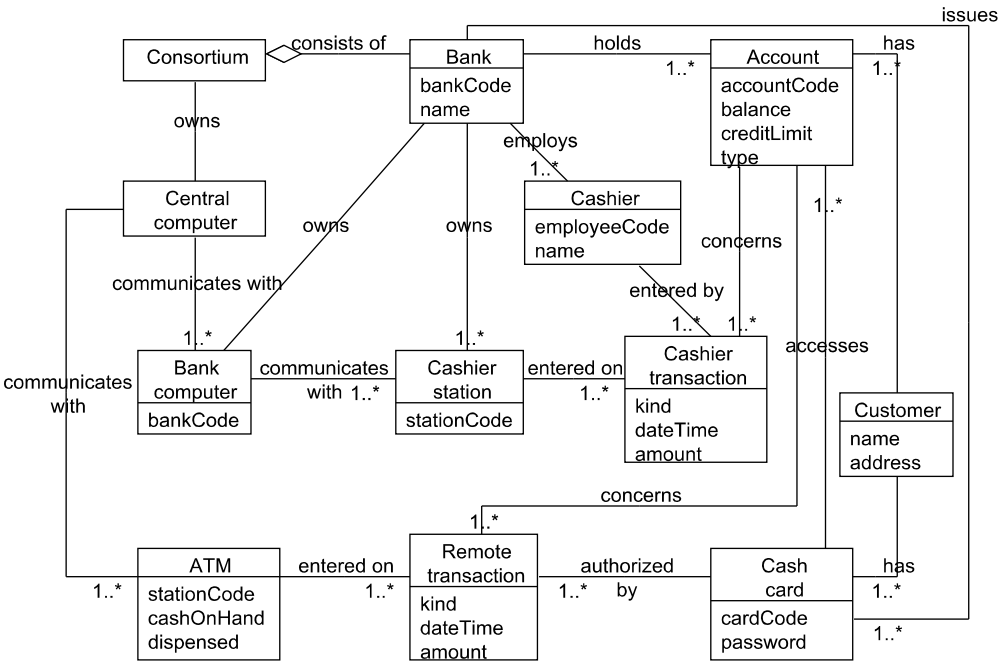
Knowledge of problem domain:

Cash card accesses accounts
Bank employs cashiers

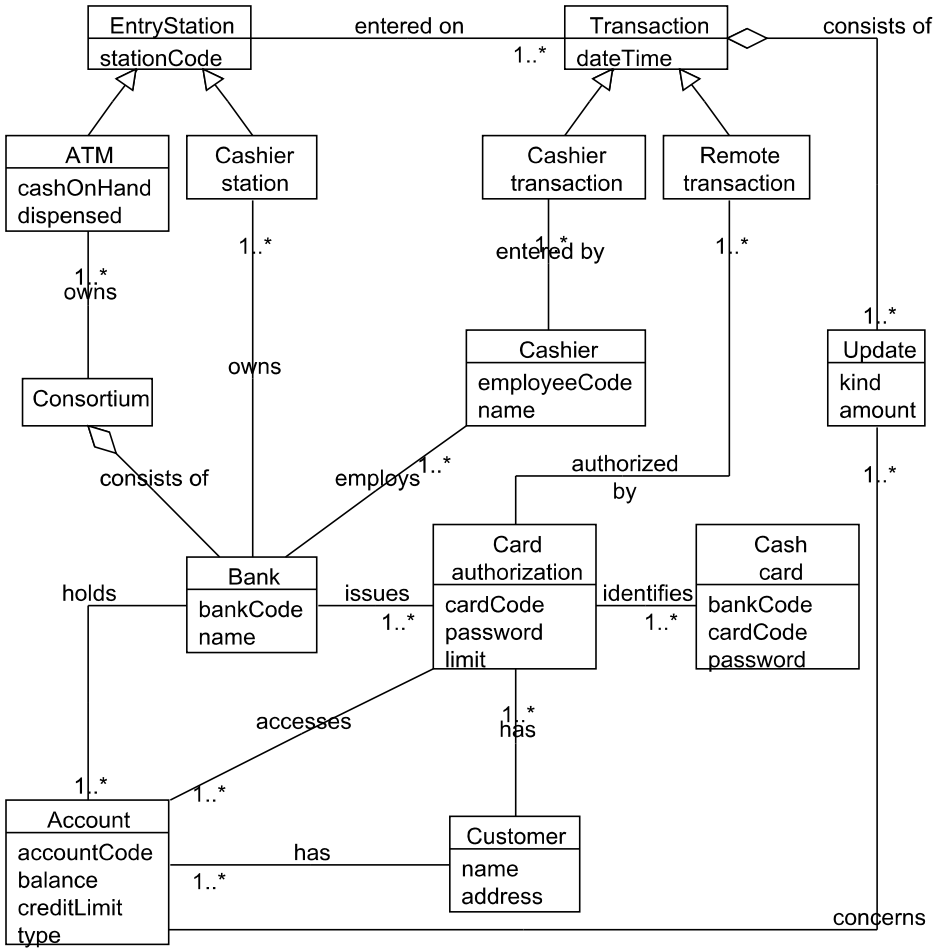
Kuva B.7: Pankkiautomaattijärjestelmän assosiaatioita



Kuva B.8: Alustava luokkakaavio



Kuva B.9: Tarkennettu luokkakaavio, jossa lisätty attribuutteja



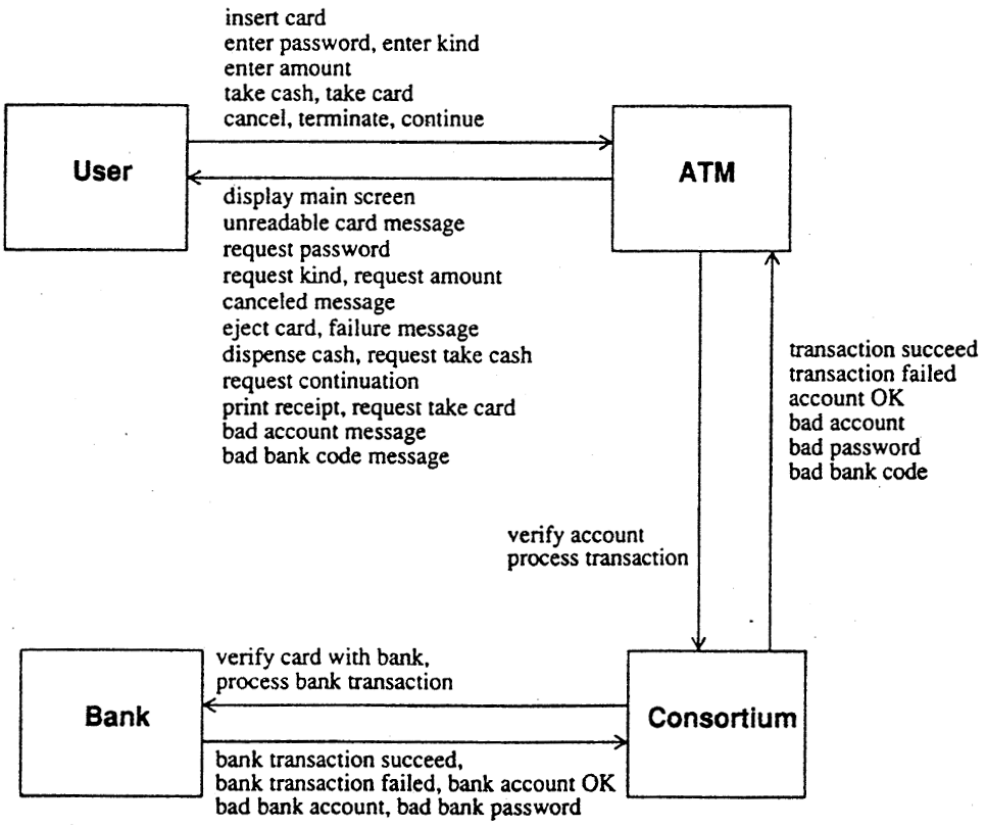
Kuva B.10: Luokkamallia muokattu edelleen ja perintä otettu käyttöön

The ATM asks the user to insert a card; the user inserts a cash card.
The ATM accepts the card and reads its serial number.
The ATM requests the password; the user enters "1234."
The ATM verifies the serial number and password with the consortium; the consortium checks it with bank "39" and notifies the ATM of acceptance.
The ATM asks the user to select the kind of transaction (withdrawal, deposit, transfer, query); the user selects withdrawal.
The ATM asks for the amount of cash; the user enters $100.
The ATM verifies that the amount is within predefined policy limits and asks the consortium to process the transaction; the consortium passes the request to the bank, which eventually confirms success and returns the new account balance.
The ATM dispenses cash and asks the user to take it; the user takes the cash.
The ATM asks whether the user wants to continue; the user indicates no.
The ATM prints a receipt, ejects the card, and asks the user to take them; the user takes the receipt and the card.
The ATM asks a user to insert a card.

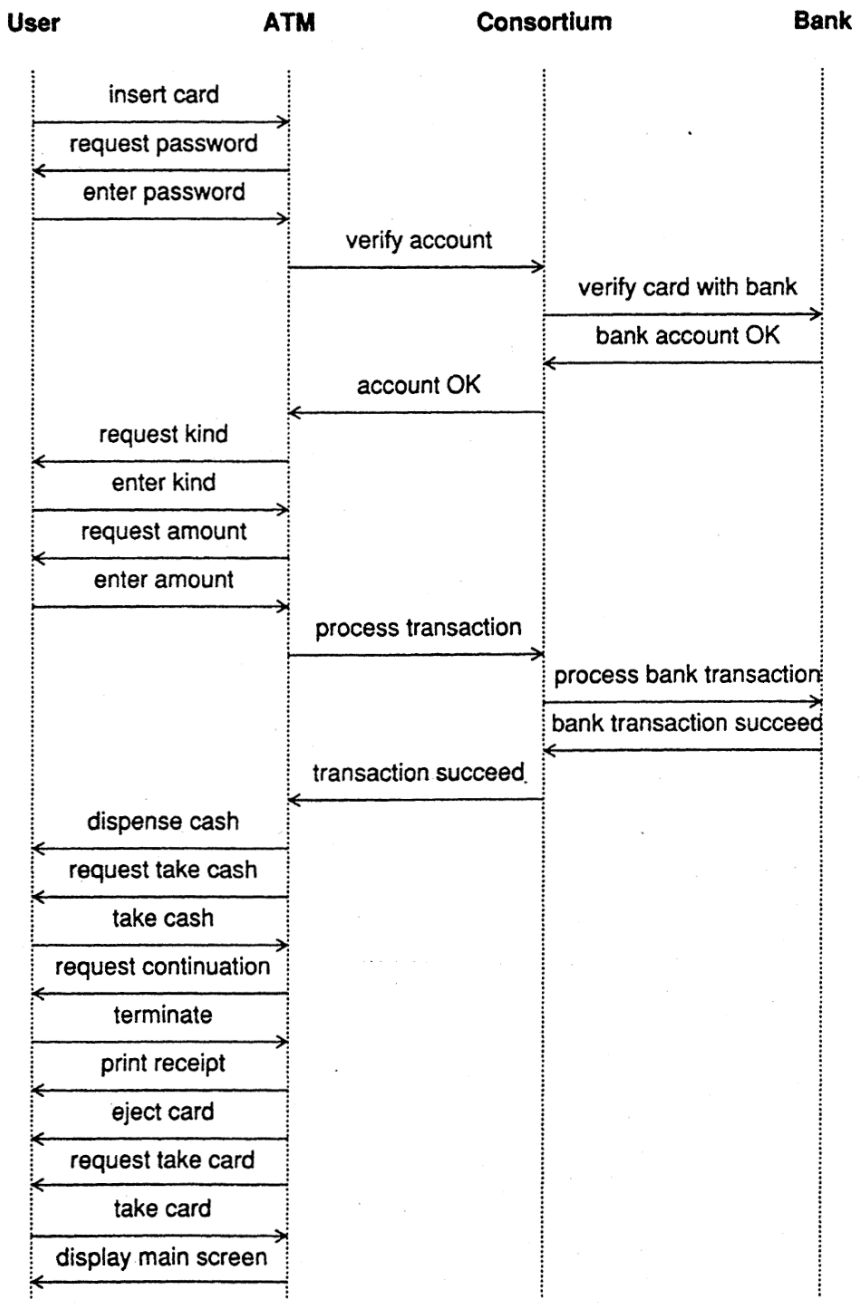
Kuva B.11: Esimerkkiskenaario (=käyttötapauksen läpikäynti) pankkiautomaatin käytöstä

The ATM asks the user to insert a card; the user inserts a cash card.
The ATM swallows the card and reads its serial number.
The ATM requests the password; the user enters "9999."
The ATM verifies the serial number and password with the consortium, which rejects it after consulting the appropriate bank.
The ATM indicates a bad password and asks the user to renter it; the user enters "1234" which the ATM successfully verifies with the consortium.
The ATM asks the user to select the kind of transaction; the user selects withdrawal.
The ATM asks for the amount of cash; the user has a change of mind and hits "cancel."
The ATM ejects the card and asks the user to take it; the user takes it.
The ATM asks a user to insert a card.

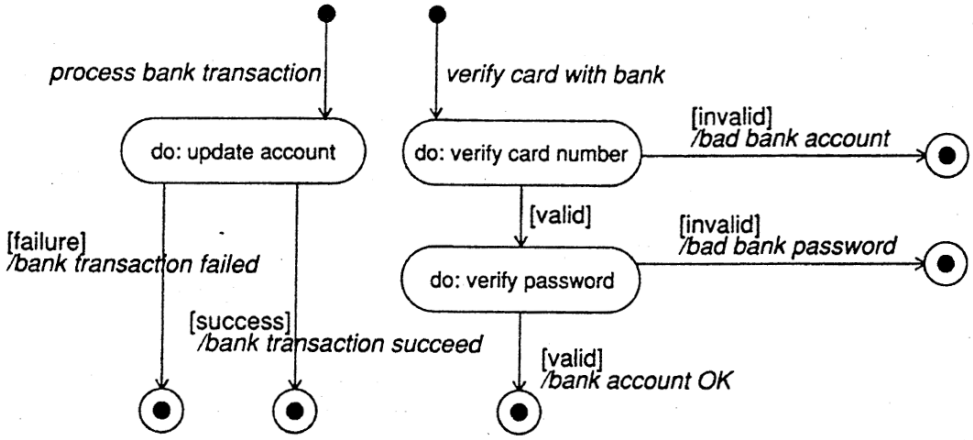
Kuva B.12: Poikkeuksen sisältävä esimerkkiskenaario



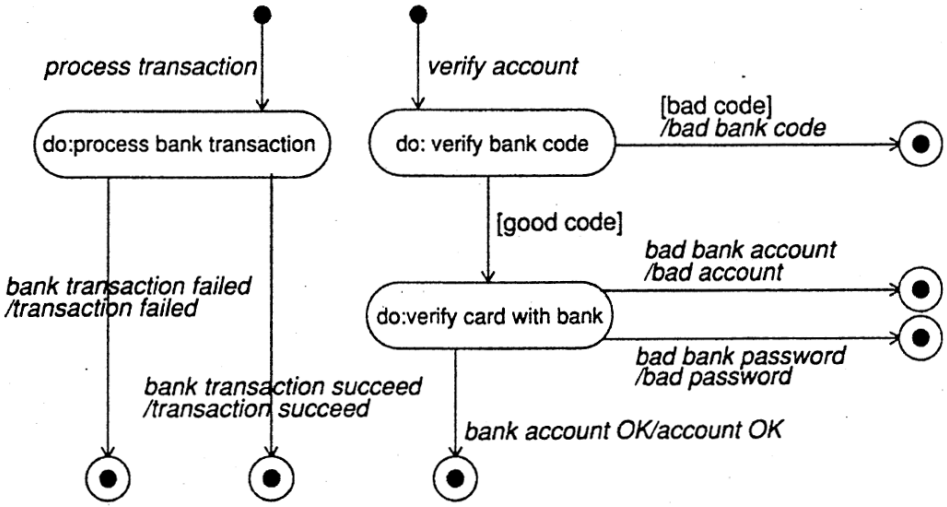
Kuva B.13: Skenaarioiden pohjalta laadittu yhteistoimintakaavio



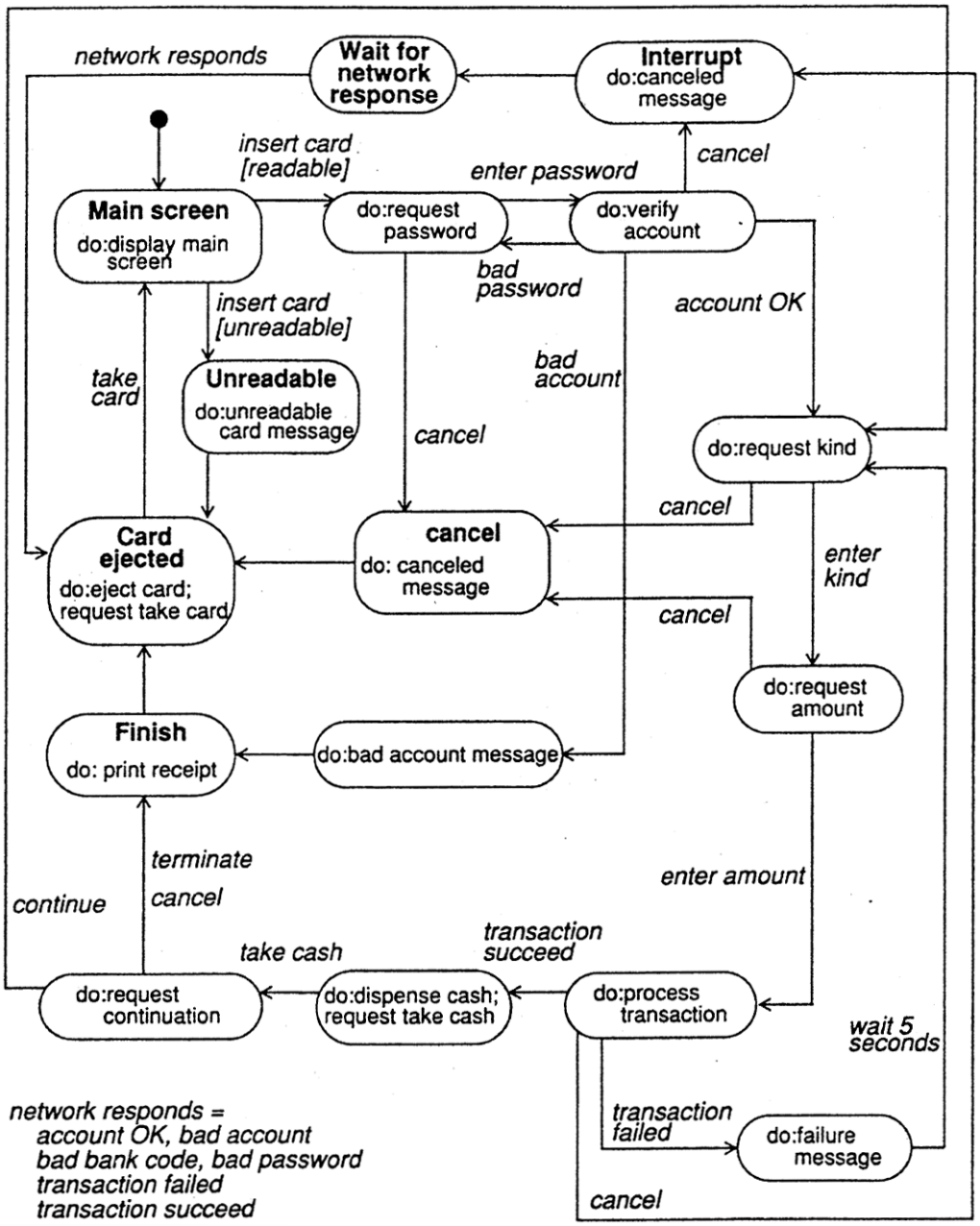
Kuva B.14: Skenaarion B.11 pohjalta laadittu sekvenssikaavio



Kuva B.15: Bank-luokan tilakaavio



Kuva B.16: Consortium-luokan tilakaavio



Kuva B.17: ATM-luokan tilakaavio