Ohjelmistotuotanto

Ohjelmistosuunnittelu 2

1. Käyttötapalähtöinen suunnittelu

- Lähtökohdat: käyttötapamalli (käyttötapaukset, use cases), määrittelytason luokkakaavio
- Tehtävät:
  1. Käyttöliittymän periaatesuunnittelu
     - Mitkä käyttötapaukset hoidetaan saman käyttöliittymän kautta?
     - Ulkoasu, "look and feel"
     - Periaateratkaisu käyttöliittymän ja sovelluslogiikan suhteista
       - miten käyttöliittymä erotetaan sisältöluokista
       - arkkitehtuuri (MVC / komennot /....)
     - Käytettävä liittymäkirjasto
     - Syntyy liittymäluokkia, jotka vastaanottavat syötteitä ja välittävät tuloksia

Käyttötapalähtöinen suunnittelu

2. Tietojen säilytykseen liittyvät periaateratkaisut
   - tiedostot, tietokannat, ohjelman ja tietokannan kommunikointi - kerrostus, eristäminen
   - kirjastot
   - sisältöluokat määrittelytason luokkien pohjalta
3. Hajautukseen ja tietoliikenteeseen liittyvät periaateratkaisut
   - sanomanvälitys, kirjastot

Käyttötapalähtöinen suunnittelu

4. Käyttötapausten läpikäynti:

4.1. Ratkaise, miten käyttötapaus hoituu käyttäjän kannalta

4.2. Ratkaise käyttötapauksen kulku järjestelmän kannalta
   - osallistuvat oliot, luokat, palvelut, attribuutit
   - olioiden yhteistyö - mitä palveluja käytetään
   - lisää tarvittaessa palveluita, luokkia ja attribuutteja, jotta saat yhteistyön sujumaan
   - simuloi käyttötapausta luokkamallin suhteen

Käyttötapalähtöinen suunnittelu

- Pitäisi saada aikaan tietty toiminto
  - Minkä olion vastuulla toiminto on?
    - luokka, olio; lisää tarvittaessa
  - Onko luokalla jo tarvittava palvelu?
    - lisää tarvittaessa
  - Onko oliolla tarpeellinen tietosisältö palvelun hoitamiseksi?
    - lisää tarvittaessa attribuutteja, yhteydet muihin olioihin
  - Hoitaako olio palvelun yksinään vai tarvitaanko yhteistyötä?
    - mahdollista yhteistyö; vain tunnettujen olioiden palveluita voi käyttää

Käyttötapalähtöinen suunnittelu

4.3. Tehdyt muutokset voivat vaikuttaa aiemmin tehtyihin ratkaisuihin - iteroi

5. Täsmällinen kuvaus luokista, attribuuteista ja palveluista
6. Pakkausten, osajärjestelmien yms. määrittely

- Hyödynnä standardoituja ratkaisumalleja ja olemassa olevia kirjastoluokkia

Olioyhteistyö

Olioyhteistyö (UML):

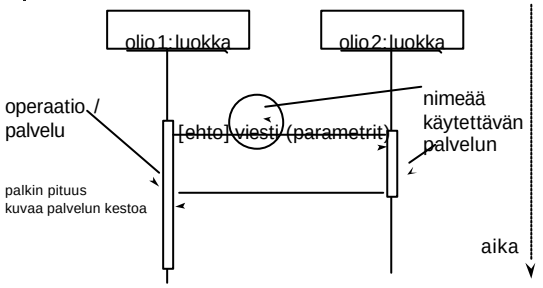
- **Sekvenssikaavio (yhteistyöpolut)** (sequence diagram)
  - Keskitytään erityisesti kuvaamaan operaatioiden tapahtumajärjestystä ja toimintaan liittyvien viestien kulkua. Sekvenssikaavion toinen ulottuvuus on aika.
- **Yhteistyörakennekaavio** (collaboration diagram)
  - Keskitytään kuvaamaan sitä, miten yhteistyö hyödyntää olioiden välisiä kytkentöjä. Viestien ajallinen kulku merkittävä kaavioon erikseen.

Olioyhteistyö

Sekvenssikaavio

- Kuvaa, miten kontrolli ajallisesti etenee oliolta toiselle jotain toimintoa (esim. käyttötapausta) suoritettaessa
  - mitä olioita palvelun suoritukseen osallistuu ja mitä näiden palveluja käytetään
  - missä järjestyksessä olioiden palveluita käytetään

Olioyhteistyö



Olioyhteistyö

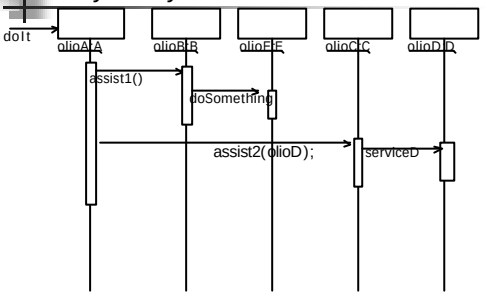
- Viestit ovat yleensä palvelujen kutsuja (palvelupyyntöjä)
  - palvelun nimi (ja parametrit)
  - käytännössä metodikutsu olio-ohjelmassa
  - ehto ei ole välttämätön
  - palveluun liittyvä paluunuoli jätetään yleensä piirtämättä, vaikka palaute saataisiinkin (kaavio tulee näin yksinkertaisemmaksi)

Olioyhteistyö

```
class A {
  B olioB;
  C olioC;
  D olioD;

  public int doIt() {
    olioB.assist1();
    olioC.assist2(olioD); }
}

class D {....
public serviceD {...}
}
```

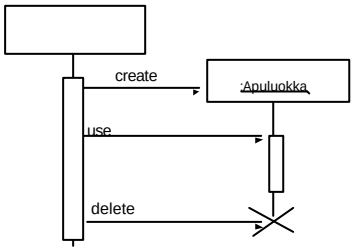
```
class B {
  E olioE;
  public void assist1() {

olioE.doSomething();} }

class C {....
public void assist2(D od){
  od.serviceD() }
}
```


Olioyhteistyö



Olioyhteistyö

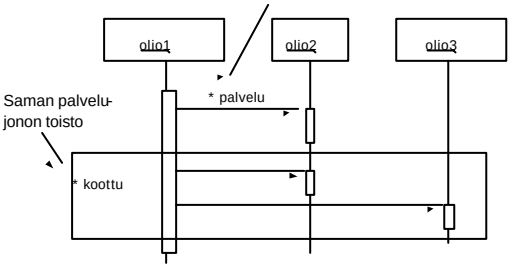
Luonti ja tuhoaminen



Olioyhteistyö

SILMUKAT

Palvelua pyydetään useilta olioilta



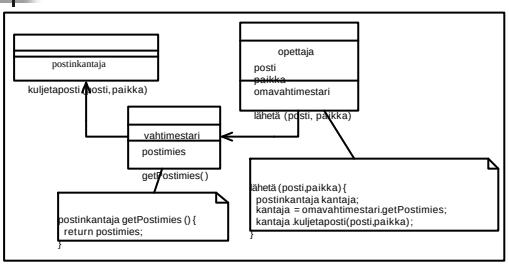
Olioyhteistyö

- Yhteistyöketjuja käytetään havainnollistamaan oliorakenteen toimintaa
  - ei kannata laatia jokaiselle palvelulle / käyttötapaukselle, vaikka jokaisen kohdalla yhteistyö on mietittävä
  - laaditaan keskeisille palveluille ja tilanteisiin, joissa kaavioiden käyttö edistää suunnitelman ymmärtämistä
  - 3-4 viestiä pidemmät ketjut pikemminkin sotkevat kuin havainnollistavat

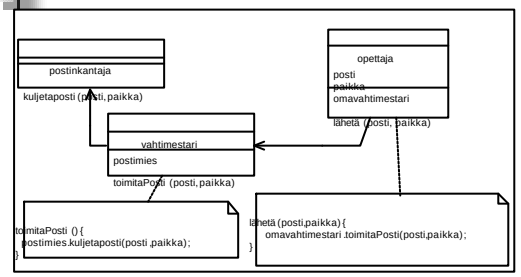
2. Hyvä oliorakenne

- Oliorakennetta suunniteltaessa pitäisi pyrkiä minimoimaan olioiden välisiä yhteyksiä (kytkentää)
  - olio 'tuntee' mahdollisimman vähän muita olioita ja näiden luokkia
  - olion tarvitsee käyttää mahdollisimman vähän muiden olioiden palveluja (kiinteys, cohesion)
  - palvelujen käyttöön liittyvät sanomat ovat mahdollisimman yksinkertaisia

Hyvä oliorakenne ?

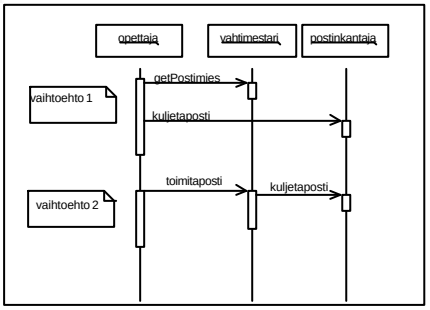


Hyvä oliorakenne



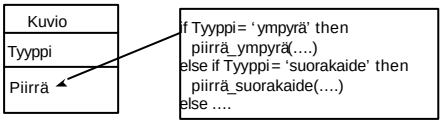
Löyhä "paikallinen" kytkentä: tämä on parempi !

Hyvä oliorakenne



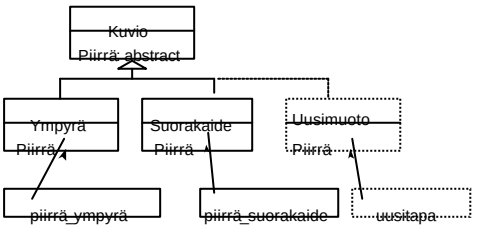
Hyvä oliorakenne

- Perinnän ja dynaamisen sidonnan hyödyntäminen
- Tarkastellaan ohjelmaa, jonka tehtävänä on tuottaa erilaisista kuvioista muodostuva kuvaesitys
- Ratkaisu 1: perinteinen malli - ei perintää
  - laajentaminen edellyttää luokan Kuvio muuttamista



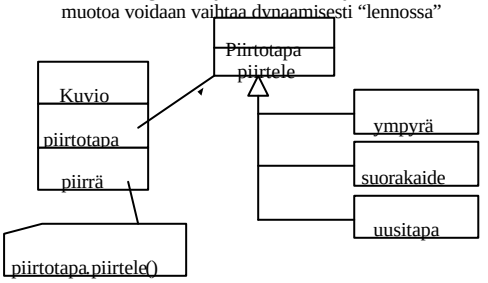
Hyvä oliorakenne

- Perintää ja syrjäyttämistä hyödyntävä. Uusia muotoja voidaan lisätä helposti. Kuvion muoto määräytyy luonnin yhteydessä.



Hyvä oliorakenne

Erillinen algoritmi, jolle toiminta ohjataan:
muotoa voidaan vaihtaa dynaamisesti "lennossa"



3. Ratkaisumallit

- Edellisellä kalvolla oli esimerkki suunnittelutason ratkaisumallista (Strategy), joka lisää joustavuutta ja muunneltavuutta järjestelmään.
- Esim. UML-kaavioiden laatimisohjelma, jossa käyttäjä saisi määritellä omat kuvasymbolinsa, voisi käyttää tätä ratkaisumallia (tai jotain vielä paremmin soveltuvaa).

Ratkaisumallit

- Ratkaisumallit (patterns) ovat systemaattisia hyviksi havaittuja tapoja ratkaista tietty usein esiintyvä ongelma
  - taitotiedon, kokemuksen dokumentti
  - yleiskäyttöisiä ratkaisumalleja
  - sovellusaluekohtaisia ratkaisumalleja
- Ratkaisumalleja voi liittyä
  - kehittämisprosessiin (process pattern)
  - määrittelyvaiheen ongelmiin (analysis pattern)
  - suunnitteluvaiheen ongelmiin (design pattern)
  - ohjelmointitason ongelmiin (idiom)

Ratkaisumallit

- Analyysimallit (analysis patterns)
  - Kuinka jäsentäisi ongelman, jotta saavutettaisiin joustavuutta/muunneltavuutta/....?
- Suunnittelumallit (design patterns)
  - Kuinka ongelma olisi hyvä ratkaista teknisesti, jotta saavutettaisiin joustavuutta / muunneltavuutta / ylläpidettävyyttä / tilansäästöä / toimivuutta / siirrettävyyttä
- Idiomit (idioms)
  - Kuinka asia pitäisi ratkaista esim. Javalla

Ratkaisumallit

- Parantelumallit eli anti-mallit (anti-patterns)
  - Normaalisti ratkaisumalleissa kuvataan ongelma ja esitetään siihen ratkaisutapa.
  - Parantelumalleissa lähtötilanteena on se, että jotain on jo tehty tavalla, joka aiheuttaa ongelmia. Mallit esittävät ratkaisutavan, jolla päästään nykyistä parempaan ratkaisuun.
  - Parantelumalli = huono ratkaisu usein esiintyvään ongelmaan + ohjeet huonon ratkaisun muuttamiseksi hyväksi (esim. ratkaisumallin mukaiseksi)

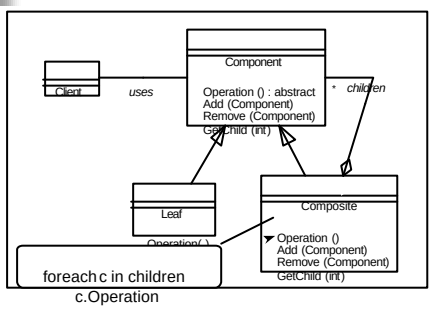
Ratkaisumallit

- Tunnetuin lähde:
  - E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns - Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995
  - esittelee 23 ohjelmista etsittyä olioperustaista suunnittelumallia
- Seuraavaksi käydään läpi muutama esimerkki yllä mainitun lähteen ratkaisumalleista

Ratkaisumallit – Kooste (Composite)

- Tarkoitus: Kokonaisuuden ja sen osien käsitteleminen samalla tavalla
- Esimerkki: piirto-ohjelmassa kuvio voi koostua hierarkkisesti pienemmistä osakuvioista, ja halutaan piirtää kaikki kuvioelementit samalla tavoin (ts. samannimisellä operaatiolla hyödyntäen olio-ohjelmoinnin monimuotoisuutta ja dynaamista sidontaa)

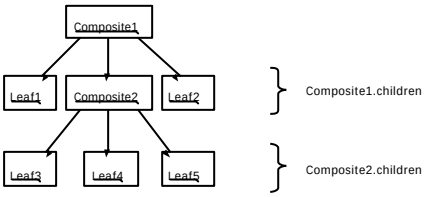
Ratkaisumallit – Kooste (Composite)



Ratkaisumallit – Kooste (Composite)

- Määritellään abstrakti luokka, jonka konkreettisina aliluokkina (siis sellaisina, joilla on ilmentymiä) ovat alkeisluokat ja kokoelmat. Palvelut määritellään abstraktin luokan tasolla. Palvelut jakautuvat kokoelman hallintapalveluihin ja varsinaisiin palveluihin. Kokoelman varsinaiset palvelut suoritetaan välittämällä pyyntö kokoelman osille.
- Esimerkiksi abstrakti luokka voisi olla *kuvio* ja sen aliluokat olisivat *alkeiskuvio* ja *koottu_kuvio*. Varsinainen palvelu olisi *piirrä*. Kootun kuvion *piirrä* -palvelu toteutetaan kutsumalla kunkin sen osan *piirrä* -palvelua.

Ratkaisumallit – Kooste (Composite)



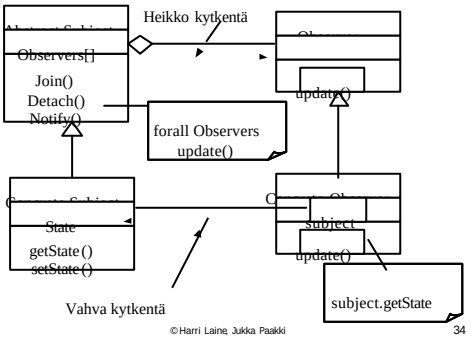
Ratkaisumallit – Kooste (Composite)

- Ratkaisun etuna on se, että asiakas pystyy käsittelemään myös koosteoliota missä tahansa yhteydessä, jossa se käsittelee alkeisoliota
- Haittana on se, että kaikki koosteoliot ovat rakenteeltaan samanlaisia (ei voi olla eri tavoin käsiteltäviä, täysin erilaatuisia osia)

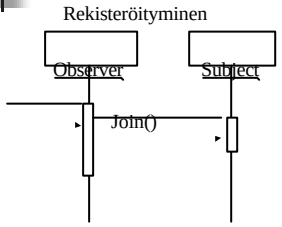
Ratkaisumallit – Tarkkailija (Observer)

- Tarkkailija (Observer) esittää tavan välittää olion tilan muuttuminen tästä riippuville olioille ilman, että olioiden välillä olisi kumpaankin suuntaan vahvaa kytkentää.
- Kaksi roolia, *kohde* ja *tarkkailija*. Tarkkailija (Observer) rekisteröityy tarkkailemaan kohdetta (Subject). Kohde tiedottaa muutoksesta kaikille rekisteröityneille tarkkailijoilleen. Saatuaan ilmoituksen tarkkailija selvittää kohteen tilan.

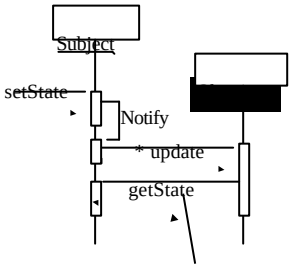
Ratkaisumallit – Tarkkailija (Observer)



Ratkaisumallit – Tarkkailija (Observer)



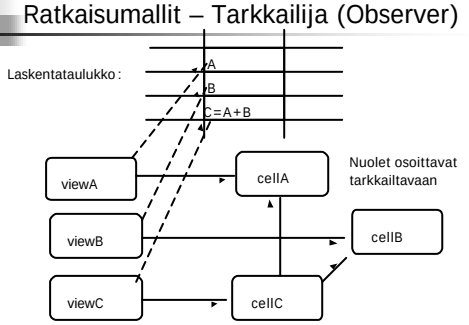
Ratkaisumallit – Tarkkailija (Observer)



Ratkaisumallit – Tarkkailija (Observer)

- Käyttötilanteita
  - käyttöliittymän näkymä - kohde
  - taulukon (taulukkolaskimen) johdettu alkio - perusalkio

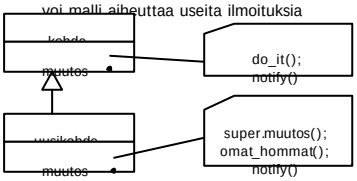
Ratkaisumallit – Tarkkailija (Observer)



Ratkaisumallit – Tarkkailija (Observer)

Vaaroja:

- Tarkkailija saa useita ilmoituksia saman kohteen muutoksista - turhia päivityksiä
  - esim. jos kohdeluokan erikoistamiseen ei ole varauduttu, voi malli aiheuttaa useita ilmoituksia

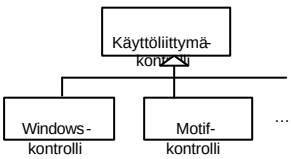


Ratkaisumalli – Tehtaat (Factory-mallit)

- Olioiden luominen on oliomalleissa ongelmallinen tehtävä
  - olion toiminnallisuus määräytyy sille luontihetkellä annetun konkreettisen luokan perusteella
  - konkreettiset luokat eivät ole tiedossa kirjastoa laadittaessa - miten siis kirjaston pitäisi huomioida luonti?
    - erityiset luontiluokat, joilla abstraktit luontipalvelut
    - luontipalvelut konkretisoidaan luontiluokkien aliluokissa

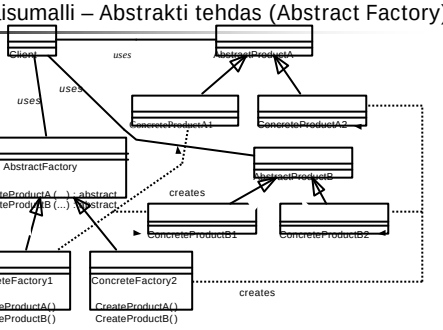
Ratkaisumalli – Abstrakti tehdas (Abstract Factory)

- Abstract Factory: olioiden luontiin liittyvä olioiden välisiä suhteita hyödyntävä malli
- Tarkoitus: malli esittää tavan luoda tietyn luokan jälkeläisluokkien ilmentymiä tuntematta näitä luokkia
- Esimerkki: yleiskäyttöinen käyttöliittymäkirjasto



kontrollit erilaisia eri ympäristöissä; kirjaston pitäisi olla yleinen, mutta miten voidaan yleisellä tasolla luoda ilmentymiä ympäristökohtaisille luokille?

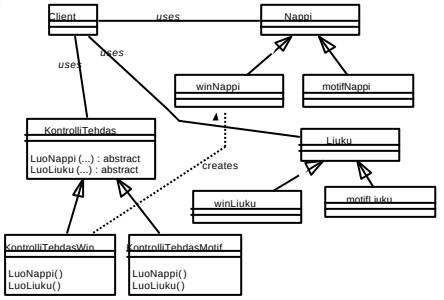
Ratkaisumalli – Abstrakti tehdas (Abstract Factory)



Ratkaisumalli – Abstrakti tehdas (Abstract Factory)

- Client (siis hyödyntävä ohjelma) tuntee abstraktin tehtaan ja abstraktit tuotteet ja kutsuu näiden tarjoamia palveluja. Abstraktin tehtaan palvelukutsu ohjautuu konkreettisen tehtaan vastaavaan luontipalveluun, joka siis luo ilmentymän kyseisen tyyppiselle tuotteelle
- Esim. Windows-kontrollitehtaan LuoNappi-palvelu luo ilmentymän winNappi-luokkaan.

Ratkaisumalli – Abstrakti tehdas (Abstract Factory)



Ratkaisumalli – Abstrakti tehdas (Abstract Factory)

- Mallilla eristetään konkreettiset tehtaat asiakkaasta
- Tuoteperhe voidaan helposti vaihtaa toiseksi
- Asiakkaan on tunnettava tuotetyypit, joten uusia tuotetyyppejä ei voida lisätä pelkästään aliluokkia lisäämällä, vaan tällöin on muutettava asiakkaan koodia ja abstraktin tehtaan koodia
  - esim. kilistimen lisäys käyttöliittymäkontrolleihin

Ratkaisumallit – Komento (Command)

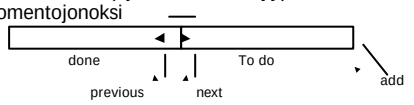
- Komento (Command) -suunnittelumalli soveltuu tilanteisiin, joissa
  - operaation käynnistämiseen on erilaisia vaihtoehtoja, mutta lopputuloksen pitäisi kuitenkin olla sama
  - järjestelmään pitäisi pystyä helposti liittämään uusia operaatioita puuttumatta järjestelmän perusrakenteisiin
  - operaatioita pitäisi pystyä perumaan ja uudelleensuorittamaan
- Suunnittelumallissa varsinainen toiminta eristetään toimintapyynnöstä muodostamalla pyynnöstä erillinen komento -olio
- Kaikilla komennoilla on yhteinen rajapinta, joka pitää sisällään vähintään palvelun *suorita* (do, execute)

Ratkaisumallit – Komento (Command)

- Komennon rajapintaan voi lisäksi kuulua
  - peruutus (undo)
    - toimiakseen peruutus saattaa edellyttää edeltäneen tilan kirjaamista komennon tietorakenteisiin
    - esim. kun teksti korvataan toisella, on peruutusta varten taltioitava
      - korvattu teksti
      - sen alkukohta ja
      - korvaavan tekstin pituus
  - uudelleensuoritus (redo)
    - edellyttää myös tilatietojen tallennusta

Ratkaisumallit – Komento (Command)

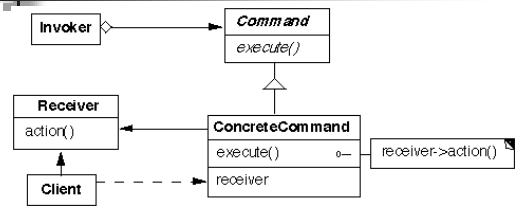
- Komentojen suorituksen ohjausta voidaan hallita erityisen komentokäsittelijän (command processor) avulla
  - komentokäsittelijä ei ole aina välttämätön, mutta se on tarpeen esimerkiksi peruutusten yhteydessä
  - komentokäsittelijä ottaa vastaan komentoja (addCommand) ja tallentaa ne tyypillisesti komentojonoksi



Ratkaisumallit – Komento (Command)

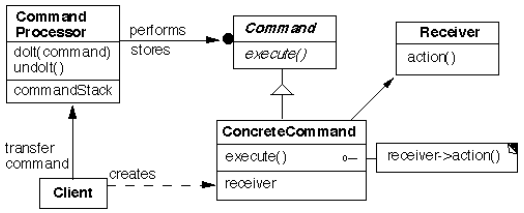
- Peruskuvio 1:
  - Kun aiheuttaja (client, controller) haluaa saada aikaan toiminnon, se luo komennon ja joko
    - aktivoi luomansa komennon suorituksen tai
    - luovuttaa sen komentokäsittelijälle aktivoitavaksi
- Peruskuvio 2:
  - Aiheuttaja reagoi käyttäjän toimenpiteeseen aktivoimalla siihen ennalta kytketyn komennon suorituksen

Ratkaisumallit – Komento (Command)

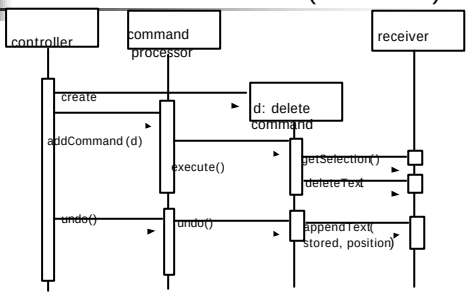


Client (esim. pääohjelma) luo konkreettisen komennon (esim. OpenFile). Se kytketään jotenkin aiheuttajaan Invoker (esim. MenuVaihtoehto - mainmenu. addItem('Open', new OpenFile())

Ratkaisumallit – Komento (Command)

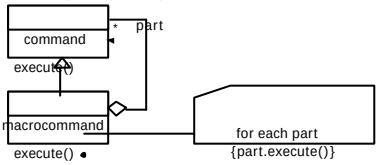


Ratkaisumallit – Komento (Command)



Ratkaisumallit – Komento (Command)

- Yleensä on järkevää mahdollistaa myös komennot, jotka kootaan alkeellisemmista komennoista
  - tällöin koottu komento (makro) on syytä toteuttaa koosteen (Composite) mukaisesti:



Ratkaisumallit – Komento (Command)

- Komentoja on suhteellisen helppo lisätä
  - määritellään uusi komentoluokka
- Sama toiminto on kytkettävissä eri käynnistystapoihin
- Suuren komentomäärän välttämiseksi komentoja kannattaa yleistää ja parametroida

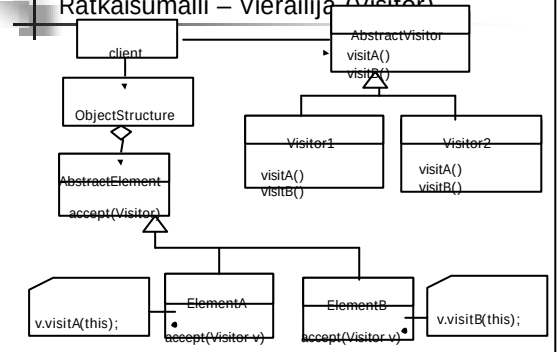
Ratkaisumalli – Vierailija (Visitor)

- Vierailija (Visitor) soveltuu käytettäväksi
  - kun erilaiset liittymät ja rakenteet omaaville kohteille halutaan suorittaa toimenpiteitä, jotka riippuvat konkreettisista luokista
  - yhteenliittyvää operaatiokokonaisuutta ei haluta sirotella luokkiin
    - eri käyttötilanteisiin liittyvän tilannekohtaisen koodin mukanaolo luokissa ei edistä uuskäyttöä
  - rakenne on staattinen mutta operaatiot dynaamisia

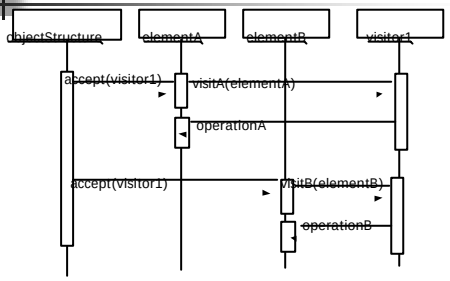
Ratkaisumalli – Vierailija (Visitor)

- Ratkaisumallissa kootaan luokkakohtaiset palvelut vierailijan metodeiksi. Vierailijalla on kutakin luokkaa kohti oma erityismetodinsa.
- Oliot rekisteröivät vierailijan ja suorittavat palvelunsa kutsumalla vierailijan luokkakohtaista metodia.
- Vierailijan vaihtaminen vaihtaa palvelun.
- Vierailijan abstraktissa määrittelyssä on otettava huomioon kaikki konkreettiset luokat, joissa vierailija voi käydä => rakenne on ylläpidon kannata hankala.

Ratkaisumalli – Vierailija (Visitor)



Ratkaisumalli – Vierailija (Visitor)



Ratkaisumalli – Vierailija (Visitor)

- Toiminnallisia kokonaisuuksia saa lisättyä laatimalla uuden konkreettisen vierailijan
- Elementeissä voidaan aina varautua vierailijoihin
- Vierailija erottaa toiminnan ja tietorakenteen
- Tiukka kytkentä vierailijasta vierailtavaan. Vierailtavan muutos aiheuttaa yleensä aina muutoksen myös vierailijaan
- Epäsuora suoritusaikainen kytkentä käynnistyvään toimintaan
  - normaalisti (C++, Java) olion luokka ja palvelun nimi määräävät toiminnon, ratkaisumallia käytettessä mukaan tulee lisäksi vierailija

Suunnittelumallit (Gamma)

- Luonti
  - Abstract Factory: samankaltaisten olioiden abstrakti luonti
  - Factory Method: virtuaalinen luontioperaatio
  - Builder: olion rakenteen eristäminen sen luonnista
  - Prototype: kloonattavissa olevan prototyyppi-olion luonti
  - Singleton: yhden ainokaisen luokkailmentymän luonti

Suunnittelumallit (Gamma)

- Rakenne
  - Adapter : luokan rajapinnan muuntaminen
  - Bridge: abstraktin luokan ja konkreettisen luokan eristäminen toisistaan
  - Composite: hierarkkisen rakenteen yhdenmukainen käsittely
  - Decorator: olion toiminnallisuuden laajentaminen dynaamisesti
  - Facade: yhtenäisen rajapinnan luominen erityyppisille luokille

Suunnittelumallit (Gamma)

- Rakenne
  - Flyweight: yleiskäyttöiset, jaetut oliot
  - Proxy: olion edustaminen erillisellä valvojalla

Suunnittelumallit (Gamma)

- Toiminta
  - Chain of Responsibility: palvelupyynnön delegointi olioketjua pitkin
  - Command: komento oliona
  - Interpreter : kielen kielioppipohjainen tulkki
  - Iterator: rakenteen osissa vierailu
  - Mediator: olioiden välisen vahvan kytkennän muuttaminen löyhäksi
  - Memento: olion tilan tallentaminen

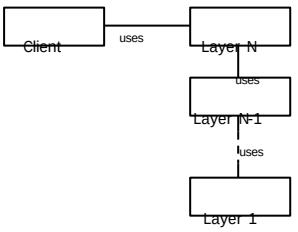
Suunnittelumallit (Gamma)

- Toiminta
  - Observer : olion tilamuutosten valvonta
  - State: olion toiminnallisuuden muuttaminen tilamuutoksen yhteydessä
  - Strategy: algoritmiperheen toteutus
  - Template Method: algoritmin luuranko
  - Visitor: rakenteen osien käsittely järjestyksessä

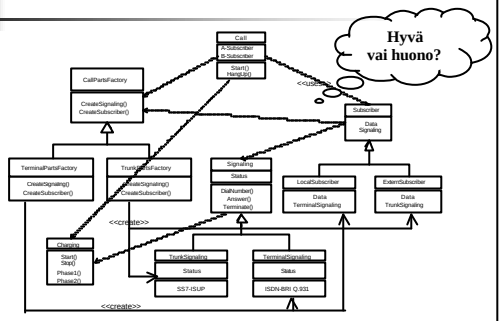
4. Arkkitehtuurimallit

- F. Buschmann et al.: *Pattern-Oriented Software Architecture – A System of Patterns.* John Wiley, 1996
  - arkkitehtuuritason ratkaisumalleja
  - myös suunnittelumalleja ja idiomeja
  - kuvaustapa vähemmän systemaattinen kuin Gamman kirjassa
  - mallit eivät välttämättä olioperustaisia

4. Arkkitehtuurimallit – Tasot (Layers)

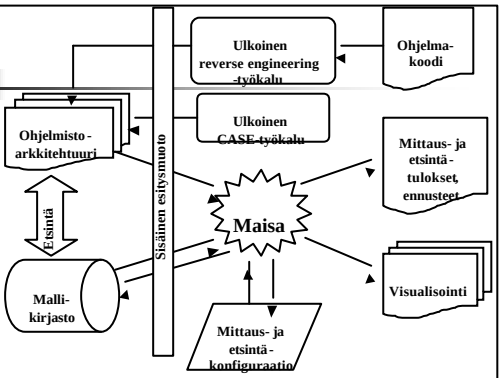


5. Arkkitehtuurin mittaaminen – Maisa

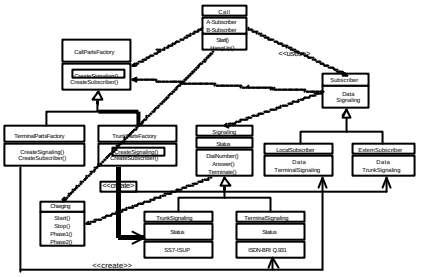


Arkkitehtuurin mittaaminen – Maisa

- **Maisa** : Laitoksen tutkimusprojekti (Metrics for Analysis and Improvement of Software Architectures)
- ***Hypoteesi*** : (anti-)suunnittelumallit ilmentävät arkkitehtuurin laatua
- ***Hypoteesi*** : arkkitehtoniset mallit ennustavat ohjelmiston laatua
- Malleja voi löytää arkkitehtuurista automaattisesti
- Arkkitehtuuria voi mitata (lisäksi perinteisin ohjelmistomittarein)



Arkkitehtoninen löydös – Abstract Factory



Maisa - Perinteisiä suunnittelumittareita

- *Koko* : 11 luokkaa, 8 attribuuttia, 13 operaatiota
- *Mutkikkuus* : syvyys 2, leveys 2, kytkentä 4, McCabe 5
- *Suoritusaika* : best-case 500, worst-case 1000, average-case 725

- Lasketaan UML:n luokka-, tila-, aktiviteetti- ja sekvenssikaavioista
- Malliesiintymät toistaiseksi vain luokkakaavioista