

### Ohjelmistosuunnittelu 2

1

## 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
    - Syntyvät liittymäluokkia, jotka vastaanottavat syötteitä ja välittävät tuloksia

© Harri Laine, Jukka Paakki

2

## 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

© Harri Laine, Jukka Paakki

3

## Käyttötapalähtöinen suunnittelu

4. Käyttötapauksen 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 käyttötapauksia luokkamallin suhteen

© Harri Laine, Jukka Paakki

4

## Käyttötapalähtöinen suunnittelu

- Pitäisi saada aikaan tietty toiminto
  - Minkä oliion 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

© Harri Laine, Jukka Paakki

5

## Käyttötapalähtöinen suunnittelu

- 4.3. Tehdyt muutokset voivat vaikuttaa aiemmin tehtyihin ratkaisuihin - iterointi
  5. Täsmällinen kuvaus luokista, attribuuteista ja palveluista
  6. Pakkausten, osajärjestelmien yms. määrittely
- Hyödynnä standardoituja ratkaisumalleja ja olemassa olevia kirjastoluokkia

© Harri Laine, Jukka Paakki

6

## Olioyhteistyö

Olioyhteistyö (UML):

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

© Harri Laine, Jukka Paakki

7

## Olioyhteistyö

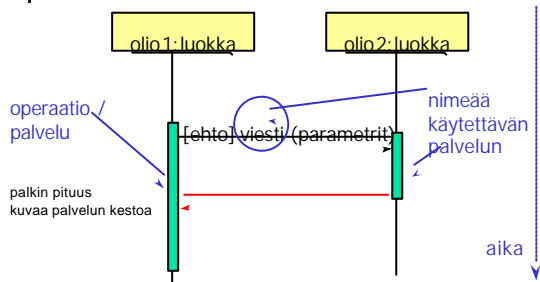
### Sekvenssikaavio

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

© Harri Laine, Jukka Paakki

8

## Olioyhteistyö



© Harri Laine, Jukka Paakki

9

## Olioyhteistyö

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

© Harri Laine, Jukka Paakki

10

## Olioyhteistyö

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

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

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

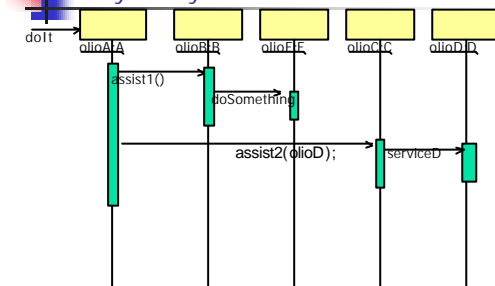
class B {
    E olioE;
    public void assist1() {
        olioE.doSomething();
    }

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

© Harri Laine, Jukka Paakki

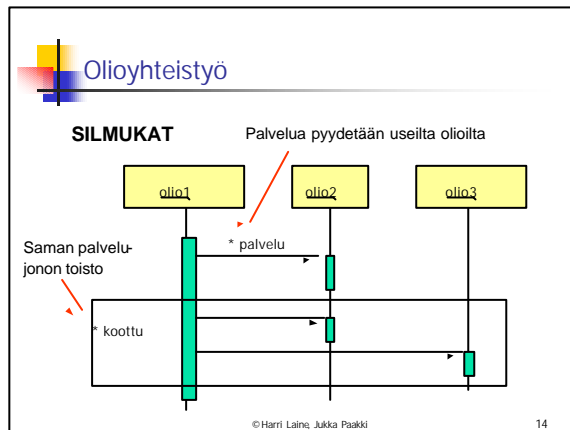
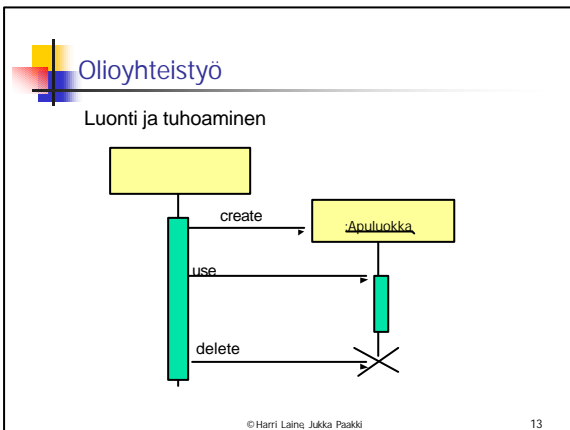
11

## Olioyhteistyö



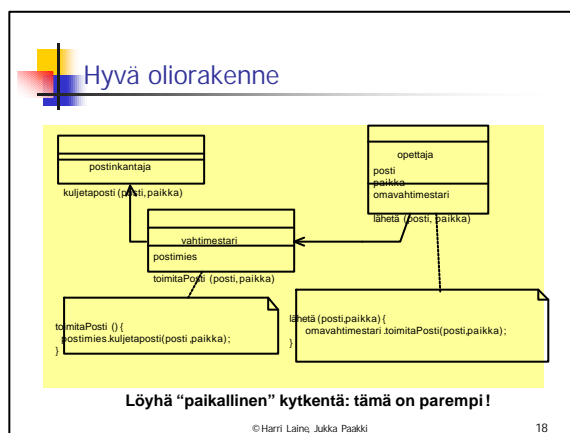
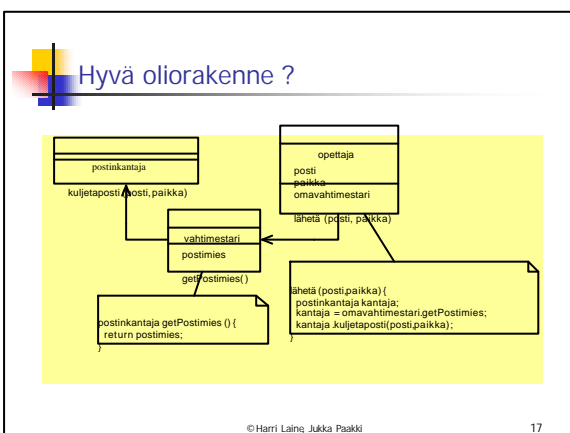
© Harri Laine, Jukka Paakki

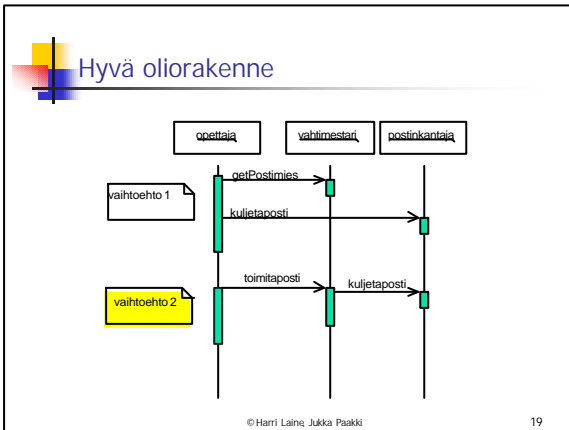
12



- ### Olioyhteistyö
- Yhteistyökettuja käytetään havainnollistamaan oliorakenteen toimintaa
    - ei kannata laatia jokaiselle palvelulle / käytötapaukselle, vaikka jokaisen kohdalla yhteistyö on mieltävä
    - laaditaan keskeisille palveluille ja tilanteisiin, joissa kaavioiden käyttö edistää suunnitelman ymmärtämistä
    - 3-4 viestiä pidemmät ketjut pikemminkin sotkevat kuin havainnollistavat
- © Harri Laine, Jukka Paakki 15

- ### 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 (kiinteyts, cohesion)
    - palvelujen käyttöön liittyvät sanomat ovat mahdollisimman yksinkertaisia
- © Harri Laine, Jukka Paakki 16





### Hyvä oliorakenne

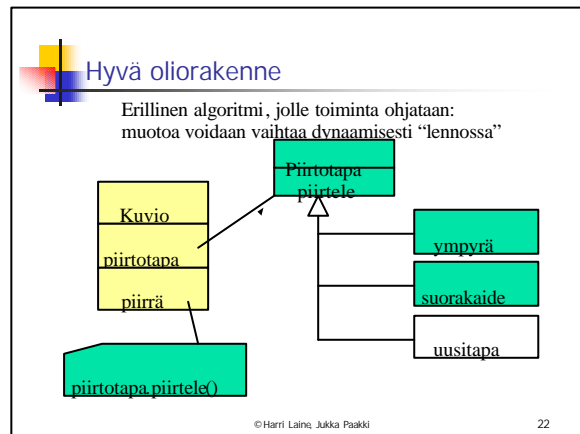
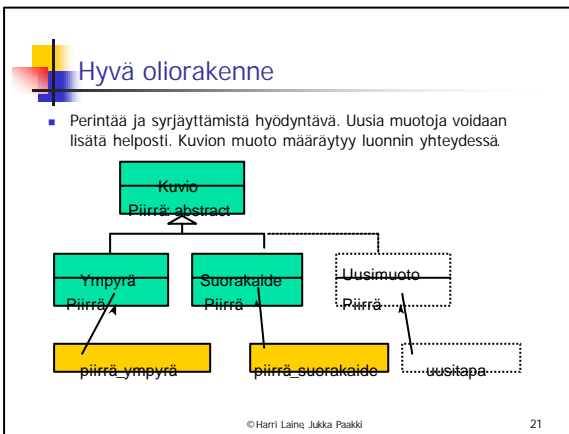
- Perinnän ja dynaamisen sidonnan hyödyntäminen
- Tarkastellaan ohjelmaa, jonka tehtävänä on tuottaa erilaisista kuvioista muodostuva kuvaesitys
- Ratkaisu 1: perinteinen malli - ei perintää
  - laajentaminen edellyttää luokan Kuvio muuttamista

Kuvio
Tyyppi
Piirrä

```

f Tyyppi= 'ympyrä' then
  piirrä_ympyrä(...)
else if Tyyppi= 'suorakaide' then
  piirrä_suorakaide(...)
else ....
  
```

© Harri Laine, Jukka Paakki 20



### 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).

© Harri Laine, Jukka Paakki 23

### Ratkaisumallit

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

© Harri Laine, Jukka Paakki 24

## 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

© Harri Laine, Jukka Paakki

25

## 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)

© Harri Laine, Jukka Paakki

26

## 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

© Harri Laine, Jukka Paakki

27

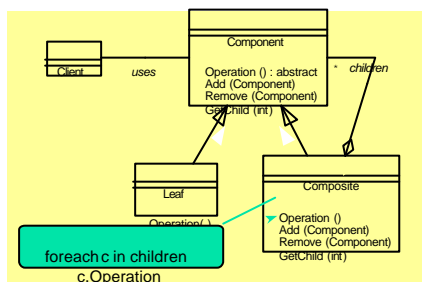
## Ratkaisumallit – Kooste (Composite)

- 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)

© Harri Laine, Jukka Paakki

28

## Ratkaisumallit – Kooste (Composite)



© Harri Laine, Jukka Paakki

29

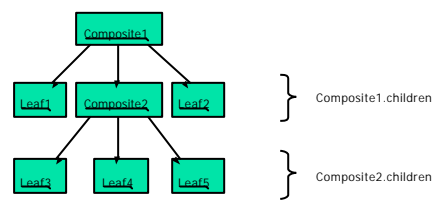
## Ratkaisumallit – Kooste (Composite)

- 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.

© Harri Laine, Jukka Paakki

30

## Ratkaisumallit – Kooste (Composite)



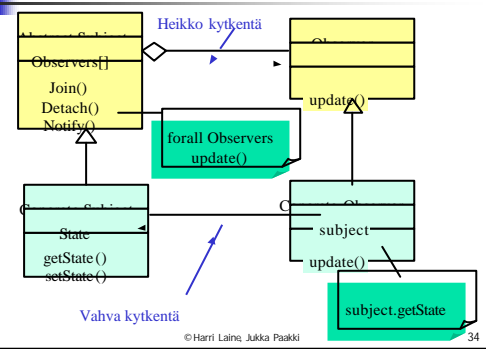
## Ratkaisumallit – Kooste (Composite)

- 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)

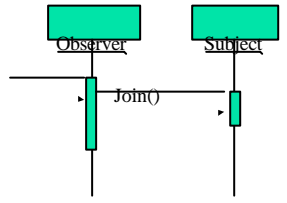
- Tarkkailija (Observer) esittää tavan välittää olion tilan muuttuminen tästä riippuville olioille ilman, että olioiden välillä olisi kumpaankin suuntaan vahvaa kytkentää.
- Kaksi roolia, *kohde* ja *tarkkailija*. Tarkkailija (Observer) rekisteröityy tarkkailemaan kohdetta (Subject). Kohde tiedottaa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

## Ratkaisumallit – Tarkkailija (Observer)

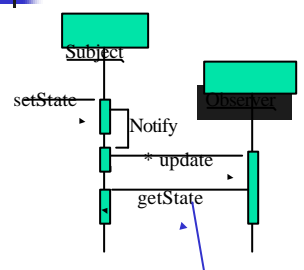


## Ratkaisumallit – Tarkkailija (Observer)

### Rekisteröityminen



## Ratkaisumallit – Tarkkailija (Observer)



Vahva kytkentä

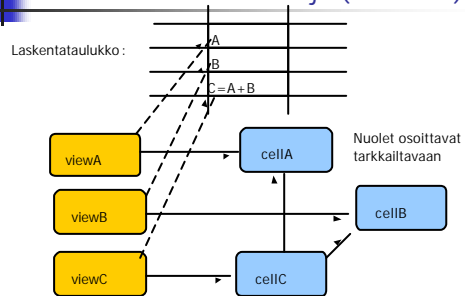
## Ratkaisumallit – Tarkkailija (Observer)

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

© Harri Laine, Jukka Paakki

37

## Ratkaisumallit – Tarkkailija (Observer)



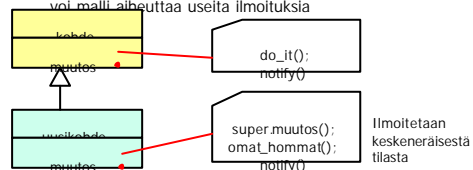
© Harri Laine, Jukka Paakki

38

## 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



© Harri Laine, Jukka Paakki

39

## Ratkaisumalli – Tehtaat (Factory-mallit)

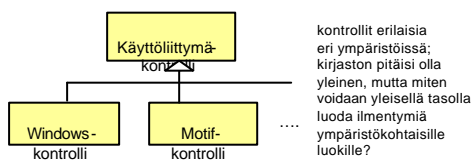
- Olioiden luominen on oliomalleissa ongelmallinen tehtävä
  - oliion toiminnallisuus määräytyy sille luontihetkellä annetun konkreettisen luokan perusteella
  - konkreettiset luokat eivät ole tiedossa kirjastoa laadittaessa - miten siis kirjaston pitäisi huomioida luonti?
    - erityiset luontiluokat,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

© Harri Laine, Jukka Paakki

40

## Ratkaisumalli – Abstrakti tehdas (Abstract Factory)

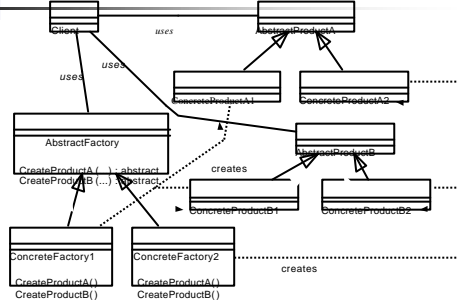
- **Abstract Factory**: olioiden luontiin liittyvä olioiden välisiä suhteita hyödyntävä malli
- Tarkoitus: malli esittää tavan luoda tietyn luokan jälkeläisluokkien ilmentymiä tuntematta näitä luokkia
- Esimerkki: yleiskäyttöinen käyttöliittymäkirjasto



© Harri Laine, Jukka Paakki

41

## Ratkaisumalli – Abstrakti tehdas (Abstract Factory)



© Harri Laine, Jukka Paakki

42

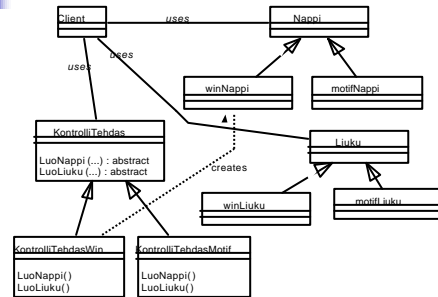
## Ratkaisumalli – Abstrakti tehdas (Abstract Factory)

- 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 .

© Harri Laine, Jukka Paakki

43

## Ratkaisumalli – Abstrakti tehdas (Abstract Factory)



© Harri Laine, Jukka Paakki

44

## 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 käyttöliittymäkontroleihin

© Harri Laine, Jukka Paakki

45

## 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 toimintapyyntöstä muodostamalla pyynnöstä erillinen komento-olio
- Kaikilla komennoilla on yhteinen rajapinta, joka pitää sisällään vähintään palvelun *suorita* (do, execute)

© Harri Laine, Jukka Paakki

46

## Ratkaisumallit – Komento (Command)

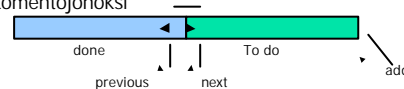
- Komennon rajapintaan voi lisäksi kuulua
  - peruutus (undo)
    - toimiakseen peruutus saattaa edellyttää edeltäneen tilan kirjaamista komennon tietorakenteisiin
    - esim. kun teksti korvataan toisella, on peruutusta varten tallioitava
      - korvattu teksti
      - sen alkukohta ja
      - korvaavan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

© Harri Laine, Jukka Paakki

47

## 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 tyyppillisesti komentojonoksi



© Harri Laine, Jukka Paakki

48



## 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

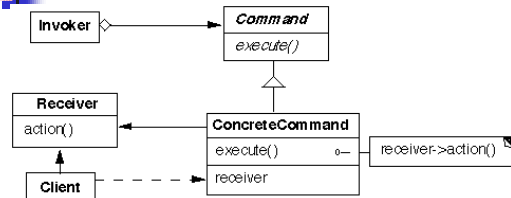
### Peruskuvio 2:

- Aiheuttaja reagoi käyttäjän toimenpiteeseen aktivoimalla siihen ennalta kytketyn komennon suorituksen

© Harri Laine, Jukka Paakki

49

## Ratkaisumallit – Komento (Command)

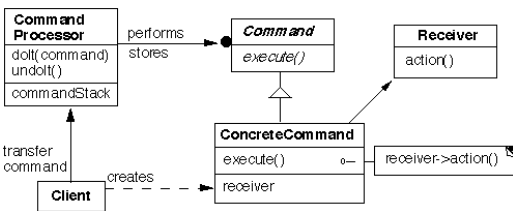


Client (esim. pääohjelma) luo konkreettisen komennon (esim. OpenFile). Se kytketään jotenkin aiheuttajaan Invoker (esim. MenuVaihtoehto - mainmenu.addItem('Open', new OpenFile())

© Harri Laine, Jukka Paakki

50

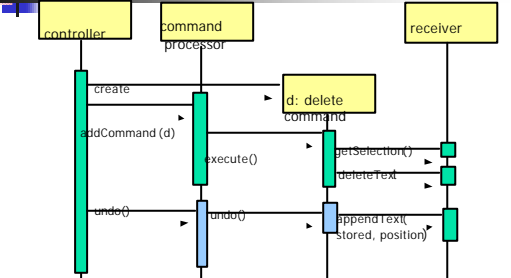
## Ratkaisumallit – Komento (Command)



© Harri Laine, Jukka Paakki

51

## Ratkaisumallit – Komento (Command)

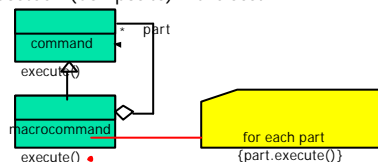


© Harri Laine, Jukka Paakki

52

## Ratkaisumallit – Komento (Command)

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



© Harri Laine, Jukka Paakki

53

## Ratkaisumallit – Komento (Command)

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

© Harri Laine, Jukka Paakki

54

## 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 uskäyttöä
  - rakenne on staattinen mutta operaatiot dynaamisia

© Harri Laine, Jukka Paakki

55

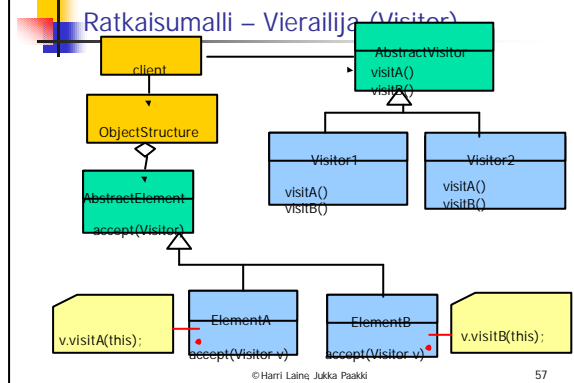
## Ratkaisumalli – Vierailija (Visitor)

- 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 kannalta hankala.

© Harri Laine, Jukka Paakki

56

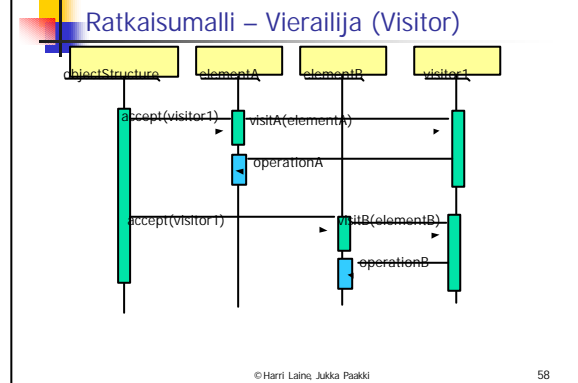
## Ratkaisumalli – Vierailija (Visitor)



© Harri Laine, Jukka Paakki

57

## Ratkaisumalli – Vierailija (Visitor)



© Harri Laine, Jukka Paakki

58

## Ratkaisumalli – Vierailija (Visitor)

- Toiminnallisia kokonaisuuksia saa lisättyä laatimalla uuden konkreettisen vierailijan
- Elementeissa 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 käytettäessä mukaan tulee lisäksi vierailija

© Harri Laine, Jukka Paakki

59

## 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

© Harri Laine, Jukka Paakki

60



## 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

© Harri Laine, Jukka Paakki

61



## Suunnittelumallit (Gamma)

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

© Harri Laine, Jukka Paakki

62



## 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

© Harri Laine, Jukka Paakki

63



## 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

© Harri Laine, Jukka Paakki

64



## 4. Arkkitehtuurimallit

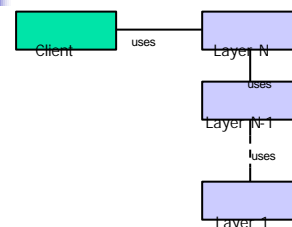
- F. Buschmann et al.: *Pattern-Oriented Software Architecture – A System of Patterns*. John Wiley, 1996
  - arkkitehtuuritason ratkaisumalleja
  - myös suunnittelumalleja ja idiomeja
  - kuvaustapa vähemmän systemaattinen kuin Gamman kirjassa
  - mallit eivät välttämättä olioperustaisia

© Harri Laine, Jukka Paakki

65



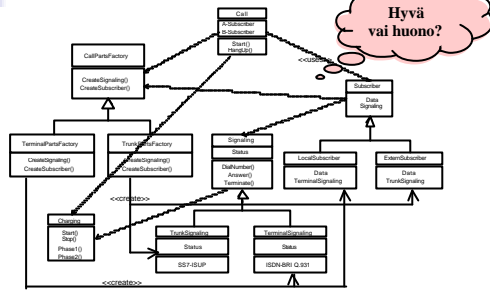
## 4. Arkkitehtuurimallit – Tasot (Layers)



© Harri Laine, Jukka Paakki

66

## 5. Arkkitehtuurin mittaaminen – Maisa



© Harri Laine, Jukka Paakki

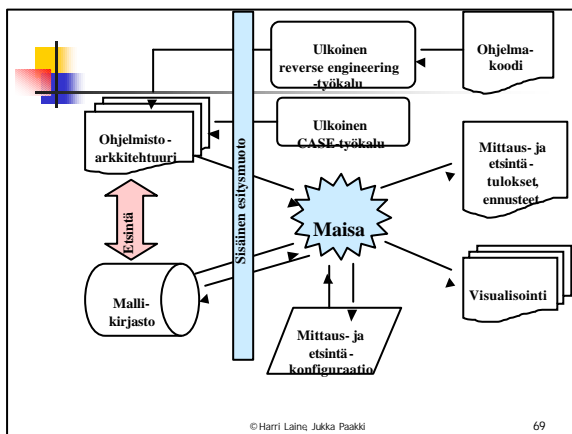
67

## 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)

© Harri Laine, Jukka Paakki

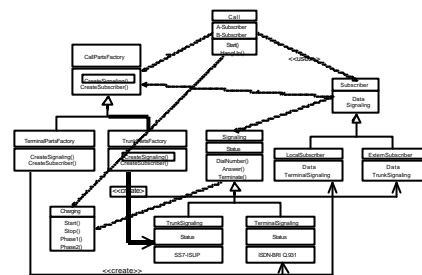
68



© Harri Laine, Jukka Paakki

69

## Arkkitehtoninen löydös – Abstract Factory



© Harri Laine, Jukka Paakki

70

## Maisa - Perinteisiä suunnittelumittareita

- **Koko** : 11 luokkaa, 8 attribuuttia, 13 operaatiota
- **Mutkikkuus** : syvyys 2, leveys 2, kytkentä 4, McCabe 5
- **Suoritus aika** : best-case 500, worst-case 1000, average-case 725
- Lasketaan UML:n luokka-, tila-, aktiviteetti- ja sekvenssikaavioista
- Malliesiintymät toistaiseksi vain luokkakaavioista

© Harri Laine, Jukka Paakki

71