Koodimalli
Code Model

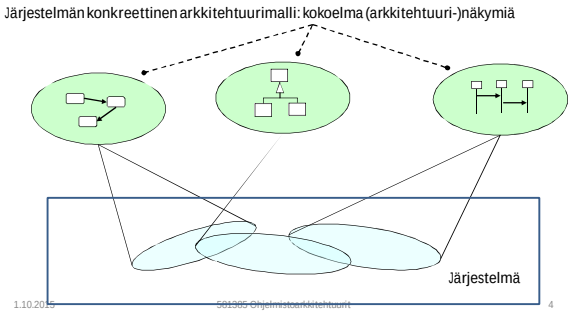
Luento 10

Oppimistavoitteet

- Näkymätyypit (suunnittelumalliasiaa)
- Koodimalli
  - Arkkitehtuurisuunnittelun ja implementaation välinen kuilu
  - Arkkitehtuurin tekeminen näkyväksi koodissa
- Arkkitehtuurikielet (ADL)

NÄKYMÄTYYPIT

Yksi arkkitehtuuri – monta näkymää



Näkymätyypit

- *Näkymätyyppi* on helposti yhteen sovitettavien ja yhdessä ymmärrettävien mutta erilaisten näkymien joukko
  - Esim. toiminnallinen skenaario ja komponenttikokoonpano suunnittelumallissa !
  - Käytetään jaottelemaan *suunnittelu-* ja *koodimallien* näkymiä
- Kolme standardinäkymätyyppiä
  - Moduulinäkymätyyppi (module viewtype, *staattinen* olomuoto)
    - Käännösaikana näkyvät elementit
  - Ajonaikainen näkymätyyppi (runtime viewtype, *dynaaminen* ol.)
    - Suorituksen aikana näkyvät elementit ja niiden suhteet
  - Sijoittelunäkymätyyppi (allocation viewtype, *operatiivinen* ol.)
    - Ohjelmiston ja sen osien suoritusympäristö ja -laitteisto

Näkymätyypit

- Usean näkymätyypin yli menevät näkymät (spanning viewtype)
  - Laatuominaisuudet ja niiden tasapainottelu (trade offs)
    - Ylläpidettävyys vs. suorituskyky

Näkymätyypit

Näkymätyyppi	Esimerkkejä tyypin näkymien sisällöstä
Moduulinäkymät	Moduulit, kerrokset, riippuvuudet, vastuut (CRC), tietokantaskeema, rajapinnat, luokat, komponenttityypit, konnektorityypit
Ajonaikaiset näkymät	Olio-instanssit, komponentti-instanssit, konnektori-instanssit, käyttäytymismallit (tilamallit, skenaariot),
Sijoittelunäkymät	Ohjelmiston sijoittelu, maantieteellinen sijainti, verkkotopologia, laitteet (nodes)
Poikkimenevät näkymät	Laatuattribuuttiskenaariot, trade offs (laatuominaisuudet, liiketoiminnan tavoitteisiin liittyvät näkökohdat)

Yinzer -suunnittelumallin näkymät

Tyyppi	Näkymät
Moduuli	Moduulikaavio Komponentti-, konnektori- ja portti*tyypit* Käyttötapauskaavio Komponenttien vastuut
Ajonaikainen	Systeemikontekstikaavio Toiminnalliset skenaariot Komponentti-, konnektori- ja portti-*instanssit* Komponenttikokoonpanot
Sijoittelu	Sijoittelukaavio Suoritusympäristön elementtien kuvaus
Ylimenevät	Laatuattribuuttiskenaariot Ominaisuuksien ja vaatimusten tasapainottelut (trade off)

Näkökohtia

- Ohjelman suoritusaikaista käyttäytymistä on vaikea päätellä koodia lukemalla
  - Koodi kannattaa kirjoittaa niin, että arkkitehtuuriratkaisut ja –tyyli näkyvät koodissa
    - Helpottaa analysoijaa ylittämään ohjelman staattisen koodin ja sen suoritusaikaisen dynaamisen olomuodon välinen kuilu
  - Älä yritä päätellä asioita näkymistä, joiden tyyppi ei siihen sovi
- Ohjelmiston jonkin tietyn suunnitteluratkaisun ymmärtämiseksi on yleensä tarpeen tutkia (joitain) näkymiä kaikista näkymätyypeistä

KOODIMALLI

Koodimalli

- Koodimalli = implementoidun ohjelmiston koodi
  - Tarkentaa (refinement) suunnittelumallin
- Fairbanks keskittyy koodin ja suunnittelun suhteen tarkasteluun kolmesta näkökulmasta
  - Mallien ja koodin väliset erot
  - Mallien ja koodin erojen hallinta
  - Arkkitehtuuria korostava ohjelmointityyli

Mallien ja koodin erot

- Ohjelmiston koodi on kehitysprojektin varsinainen lopputuote
  - Suunnitteluratkaisut on aina lopulta ilmaistava ohjelmointikiel(t)en keinoin
  - Mallit ovat hyödyllisiä vain, jos niiden välittämä kuva on riittävän yhdenmukainen koodin ilmaiseman todellisuuden kanssa
- *Huom:* MDE – Model Driven Engineering – menetelmissä *malleista generoidaan implementaatio* tai sen osa automaattisesti
  - Eivät vielä laajasti käytössä

Mallien ja koodin erot - kieli

- Monilla suunnittelumallin käsitteillä ei ole suoraa vastinetta koodissa
  - Laatuominaisuus, arkkitehtuurityyli, rajoite, komponentti (riippuu tilanteesta), vastuu, …
- Jotkin käsitteet ovat lähellä toisiaan
  - Moduuli ja pakkaus
- Koodi puhuu luokista, metodeista, funktioista, parametreista, muuttujista, kentistä, lauseista, lausekkeista jne.

Mallien ja koodin erot - abstraktisuus

- Yksi arkkitehtuurielementti suunnittelumallissa vastaa usean koodielementin yhteenliittymää
  - Palvelinkomponentti suunnittelumallissa vs. sen toteutuksen muodostavat Java-luokat (projektin itse koodaamat + kehysten/alustan mukana tulevat)
- Tietyn elementin arkkitehtuurikuvauksessa on vähemmän yksityiskohtia kuin saman elementin toteutuksessa
  - Koodin oltava yksityiskohdiltaan ”täydellinen”, jotta sitä voi suorittaa tietokoneessa

Mallien ja koodin erot - suunittelupäätökset

- Arkkitehtuurimallissa voidaan päättää käyttää tiettyjä teknologioita, mutta
  - Jätetään koodauksen asiaksi yksityiskohtaiset päätökset siitä, miten teknologiaa käyttäen implementoidaan komponentit ja konnektorit
- Malli voi määrätä rajoitteita ja esimerkiksi konkreettisia suorituskykytavoitteita, mutta
  - Lähdekoodi määrittelee tietorakenteet, algoritmit ja prosessikonfiguraation, joilla tavoite saavutetaan

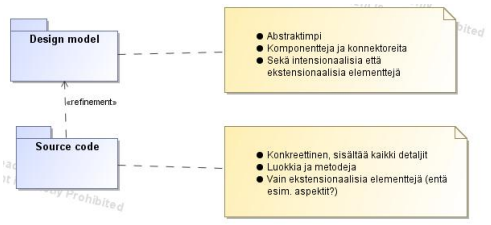
Mallien ja koodin erot – ekstensionaalisuus ja intensionaalisuus

- Ekstensionaalinen – vain *tiettyjä, erikseen nimettyjä* elementtejä koskeva asia tai väite
- Intensionaalinen – *kaikkia tietynlaisia* elementtejä koskeva asia tai väite

Ekstensionaalisuus ja intensionaalisuus

Intensionaalinen / Ekstensionaalinen	Arkkitehtuurimallin elementti	Vastaavuus lähdekoodissa
Ekstensionaalinen	Moduulit, komponentit, konnektorit, portit, komponenttikokoonpanot	Vastine koodissa voidaan yleensä osoittaa, joskin arkkitehtuurikuvaus on yleensä abstraktimpi
Intensionaalinen	Arkkitehtuurityylit, invariantit ja rajoitteet, vastuut, suunnittelupäätökset, perustelut, protokollat, laatuominaisuudet	Koodin pitää noudattaa näitä, mutta yleensä ei ole suoraa tapaa ilmaista niitä koodissa.

Mallien ja koodin erot



Mallien ja koodin erot

- Kuilu on nykyteknologioilla väistämätön
  - Arkkitehtuurimallit auttavat hallitsemaan järjestelmän monimutkaisuutta ja laajuutta, koska ne ovat *abstrakteja* ja intensionaalisia
  - Koodi voidaan suorittaa tietokoneessa, koska se on *konkreettista* ja ekstensionaalista
- Erojen kanssa on opeteltava tulemaan toimeen
  - Mallit ja koodi on ajoittain synkronoitava ohjelmiston elinkaaren aikana

Erojen hallinta

- Mallien ja koodin erot pyrkivät yleensä kasvamaan ohjelmistojen evoluution myötä
  - Uudet piirteet ja bugikorjaukset lisätään kyllä implementaatioon, mutta malleja ei aina välitetä päivittää
- Yleensä halutaan välttää mallien ja toteutuksen välisiä harhaanjohtavia tai suorastaan haitallisia ristiriitaisuuksia

Erojen hallintastrategiat

Strategia	Selitys
Älä välitä poikkeamista	Käytetään vanhentunutta mallia mutta muistetaan, mitkä asiat ovat muuttuneet
Ad hoc -mallinnus	Pidetään malli mielessä ja tehdään siitä uusi instanssi (jokin näkymä) vasta tarpeen vaatiessa
Vain "korkean tason" malleja	Arkkitehtuurin kaikkein perustavimmat ratkaisut muuttuvat yleensä hitaasti, joten niitten mallit pysyvät parhaiten ajan tasalla
Syknronointi elinkaaren virstanpylväissä	Mallien ja koodin erot sovitetaan elinkaaren virstanpylväissä/merkkikohdissa (milestone), esimerkiksi iteraation lopussa tai ohjelmistoversion toimituksen yhteydessä
Kun hätä on kädessä	Sovitetaan erot, kun jotain menee pahasti pieleen, tai kun valmistaudutaan katselmointiin (pakon edessä). Yllättävän yleinen strategia.
Jatkuva synkronointi	Kallista ja hyvin harvinaista.

Strategian valintaan vaikuttavia tekijöitä

- *Työkalutuki*
  - Esimerkiksi takaisinmallinnustyökalujen ja korkean tason ohjelmointikielten (koodin generointi) käyttö voi kaventaa kuilua ja vähentää synkronointikustannuksia
- *Yksityiskohtaisuus*
  - Paljon yksityiskohtia sisältävä malli happanee nopeammin ja vaati enemmän synkronointityötä
- *Poikkeavuuksien sietokyky*
  - Missä tilanteissa mallien on tärkeää heijastaa toteutusta oikein, ja kuka niitä käyttää?

Arkkitehtuuria korostava ohjelmointityyli

- Olio-ohjelmoinnin tunnettu perusperiaate on, että ohjelmiston luokkien (tyyppien) ja suoritusaikaisten oliorakenteiden ja vuorovaikutusten tulisi heijastaa sovellusalueen käsitteitä ja prosesseja
  - Sovellusaluetta tuntevan (ulkopuolisen) kehittäjän on helpompi ymmärtää, miten ohjelmisto toimii ja mitä eri oliot tekevät tai tietävät (vastuut)
  - Jos sovellusalueen käsitteet ovat vakaita ja niiden muutokset ovat paikallisia, myös ohjelmiston toteutus on niiden osalta vakaa ja muutokset ovat paikallisia

Arkkitehtuuria korostava ohjelmointityyli

- Sovellusalueen lisäksi koodi voi heijastaa myös *arkkitehtuurisuunnittelua*
  - Ohjelmoijat voivat upottaa koodiin vinkkejä, jotka auttavat ylittämään mallin ja implementaation välisen kuilun
  - *Model-in-code* –periaate
- Etuja
  - Arkkitehtuurin rapautumisen ehkäiseminen
  - Helpottaa suunnitteluratkaisujen tunnistamista
  - Vähentää dokumentointitaakkaa
  - Madaltaa uusien kehittäjien oppimiskynnystä

Suunnittelun ilmaiseminen koodissa

- Pehmeät mekanismit
  - Esimerkiksi metodien nimeäminen niin, että nimi kuvaa metodin tarkoitusta (*miksi*) eikä vain toimintaa (*miten*)
- Kovat mekanismit
  - Luokkien ja metodien invariantit sekä esi- ja jättöehdot, jotka voidaan automaattisesti tarkistaa suoritusaikana
  - Ohjelmointikielten laajentaminen tukemaan rajoitteiden ja muiden ehtojen noudattamisen automaattista valvontaa

Teknisen velan hallinta

- Tekninen velka tarkoittaa (mm.) korkoa korolle kasvavaa toteutuksen erkaantumista parhaan ymmärryksen mukaisesta ratkaisusta
  - Tekninen velka on helpompaa havaita ja korjata, jos koodi heijastaa arkkitehtuurisuunnittelua
  - Koodia voi helpommin verrata malliin

Mitä asioita koodissa pitäisi (yrittää) kertoa?

- Mitkä arkkitehtuurimallien kertomat asiat on vaikea päätellä suoraan koodista ilman apuja?
- Moduulinäkymissä
  - Moduulien (pakkausten) väliset riippuvuudet
  - Mistä luokista/tietorakenteista + funktioista komponentti- tai konnektorityypit muodostuvat?
  - Luokkien tarjoamat rajapinnat on usein helppo tunnistaa mutta niiden muilta *vaatimia* rajapintoja/palveluja on työläämpää nähdä koodista
  - Protokollien toimintalogiikka ja osallistuvat elementit

Mitä asioita koodissa pitäisi (yrittää) kertoa?

- Ajonaikaisissa näkymissä
  - Suorituksen kulkua ja suoritusaikaisia tietorakenteita on usein vaikea kuvitella mielessään koodia lukiessaan
  - Implementaatio näyttää oliomassalta, josta on vaikea hahmottaa komponentteja, ja konnektorien toimintaa on vaikea erottaa muusta olioiden välisestä kommunikaatiosta
  - Komponenttien ja konnektorien ajonaikaista käyttäytymistä ja konfiguraatioita koskevat intensionaaliset rajoitteet (tyylit)
  - Protokollien tilakäyttäytyminen

Mitä asioita koodin pitäisi (yrittää) kertoa?

- Sijoittelunäkymissä
  - Ohjelmistokomponenttien allokointia suoritusympäristöön ja laitteistolle on lähes mahdotonta nähdä lähdekoodista
    - Ajatellaan esimerkiksi palvelimien virtualisointia pilvipalveluissa
  - Suoritusympäristön ja tietoliikenneväylien ja topologian vaikutusta ohjelmiston suoritusaikaiseen käyttäytymiseen on hyvin vaikea päätellä koodista

Ratkaisumalleja (design pattern) arkkitehtuurin ilmaisemiseen

- Seuraavat patternit olettavat, että toteutuksessa käytetään valtavirran olio-ohjelmointikieliä
  - Patternien käyttö lisää toteutukseen ylimääräistä koodia, joka ei sinänsä ole tarpeen ohjelman oikean toiminnan kannalta (suhteessa vaatimuksiin) mutta joka ei tuo liikaa lisärasitetta suorituskyvyn tai resurssien käytön kannalta
- Monen patternin perusperiaate on *käsitteen* (abstraktin mallielementin tai piirteen) *konkretisointi* (reification) *oliona*, perittävinä luokkina tai annotaatioina
  - Esim. Command -design pattern tekee *suoritettavasta operaatiosta* olion, jolla on `execute()` -metodi

Ratkaisumalleja (design pattern) arkkitehtuurin ilmaisemiseen

Suunnittelukäsite	Patternit
Komponentti	Määrittele komponenttia vastaava luokka, jolla on abstrakti yläluokka tai rajapinta tunnisteena (tag). Liitä luokan nimeen sana Component. Käytä instanssimuuttujia edustamaan portteja ja määrittele komponentti-invarianteista metodeja. Nimeä moduulit / hakemistot komponenttien mukaan.
Konnektori	Määrittele konnektoria vastaava luokka; luokan nimentä (...Connector); abstraktin yliluokan tai rajapinnan käyttäminen tunnisteena. Ohjaa kaikki komponentin ulkopuolelle menevä kommunikaatio porttien kautta.
Portti	Määrittele tarjottua porttia vastaava rajapinta. Määrittele tarjottuja ja vaadittuja portteja vastaavat luokat; luokan nimentä (...Required/ProvidedPort); abstraktin yliluokan tai rajapinnan käyttäminen tunnisteena.

Ratkaisumalleja (design pattern) arkkitehtuurin ilmaisemiseen

Suunnittelukäsite	Patternit
Protokolla	Kytke protokollan tilakone portti-luokkaan. Käytä staattista analyysiä protokollan noudattamisen varmistamiseen, annotaatioita, kommentteja
Ominaisuudet / Property	Käytä annotaatioita ja staattista analyysiä. Käytä nimentää: AsynchronousSend...
Tyylit ja suunnittelumallit	Käytä tyylin ilmaisevaa nimentää: FeatureExtractFilter Sijoita tyylin elementtien yliluokat/tunnisterajapinnat tyylin mukaan nimettyyn pakkaukseen.
Invariantit ja rajoitteet	Kirjoita API:t niin, että ne estävät invarianttien rikkomisen. Käytä assert-lausekkeita tai muita rajoitteiden määrittely- ja tarkastusformalismeja. Kommentoi.

Ratkaisumalleja (design pattern) arkkitehtuurin ilmaisemiseen

Suunnittelukäsite	Patternit
Moduulien riippuvuudet	Käytä kielen ja sovelluskehyksen/alustan tarjoamia keinoja riippuvuuksien ilmaisemiseen (tarjotut ja vaaditut palvelut/rajapinnat/moduulit). Käytä ulkoisia työkaluja riippuvuuksien analysointiin, annotaatioita, staattista analyysiä.
Moduulien käytön rajoittaminen	Nimentä: InternalFoo... Sovellukehysten/alustojen tarjoama vivutus.
Suoritusaikaiset rakenteet	Jos mahdollista, keskitä komponenttien luonti, alustaminen ja kytkentä toisiinsa yhteen tiettyyn paikkaan koodissa. Vivuta järjestelmän käynnistystä ja alustus; mielellään niin, että halutun konfiguraation voi ilmaista deklaratiivisesti (esim XML) proseduraalisen koodin (esim skripti) sijaan.

Anti-Pattern

- Haudattu aarre (buried treasure)
  - Koodi tekee jotain muuta kuin mitä koodiin lisätyt vihjeet antavat ymmärtää
    - Esim olion `getPropertyX()` –tyyppisellä attribuutin arvon saantimetodilla on yllättävä olion tilaa muuttava *sivuvaikutus*
  - Asian voi ajatella myös toisinpäin: jos epäilet, että koodin lukija saattaa yllättyä siitä, mitä koodi tekee, liitä siihen riittävä vihje

ARKKITEHTUURIKIELET

Arkkitehtuurin määrittelykielet

- ADL – *Architecture Description Language*
- Perustuvat yleisiin arkkitehtuurielementteihin
  - Komponentit
  - Konnektorit (connector)
  - Portit
  - Rajapinnat (interface)

  - Pitkälti samoja asioita kuin UML:n komponenttimallissa
  - Usein sekä tekstuaalinen että graafinen esitys
  - Kuvauksen riittävyys analysointitarpeisiin
    - Esim. vaadittu ja tarjottu rajapinta yhdenmukaisia
    - Simuloitavuus
    - Kapasiteettilaskelmat

Arkkitehtuurin määrittelykieliä

- Arkkitehtuurikieliä - kiinnitetty käsitteistö, yleiskäyttöisiä
  - Darwin
  - Rapide
  - Wright
- Arkkitehtuurikieliä sovellusaluekohtaisia
  - Koala (viihdeelektroniikka)
  - Weaves (väylät ja suodattimet)
  - AADL
- Arkkitehtuurikieliä – laajennettava käsitteistö
  - Acme
  - ADML
  - xADL

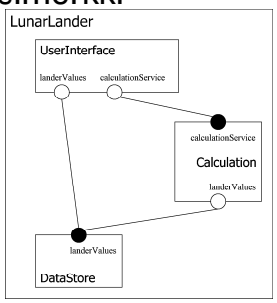
Darwin -esimerkki

```
component DataStore{
    provide landerValues;
}

component Calculation{
    require landerValues;
    provide calculationService;
}

component UserInterface{
    require calculationService;
    require landerValues;
}

component LunarLander{
inst
    U: UserInterface;
    C: Calculation;
    D: DataStore;
bind
    C.landerValues -- D.landerValues;
    U.landerValues -- D.landerValues;
    U.calculationService -- C.calculationService;
}
```



(Taylor..)

Arkkitehtuurikielet

- Erityisellä arkkitehtuurikielellä laadittu kuvaus
  - - kielet lähinnä tutkimuskäytössä
  - - ei juuri käytössä teollisuudessa
  - + käsitteet täsmällisesti määriteltyjä
  - + analysointimahdollisuuksia
- MDE – Model Driven Engineering
  - Aktiivista tutkimusta, toistaiseksi vähän käytännön sovelluksia
- DSL – Domain Specific Language
  - Sovellusaluekohtainen, usein graafinen kieli, josta generoidaan sovelluksen toteutus arkkitehtuureineen kaikkineen (yleensä suppea erikoisalue)
  - MDE:tä laajemmin käytössä (esim suomal. MetaEdit+)