Mallid tarkvara disainis (LAI8320)

Ülevaade

Disainimallide määratlus. Disainimallide esitusviisid, mallide keeled, erinevad tähistused. Disainimallide klassifikatsioon: loovad-, struktuuri- ja käitumismallid. Disainimallide kasutamine OO tarkvara disainis. Programm kui idee ülestähendus. Tarkvara arhitektuuride korduvkasutus. Raamistikud ja metamallid. Peamised probleemid korduvkasutatava OO tarkvara disainimisel. Disainimallide ja metamallide kasutamine keerukate raamistike disaini mõistmisel. Disainimallide rakendamine praktikas. Mallide avastamine olemasolevais rakendusis. Disainimallide kataloog: üldised-, hajatöötlus-, ärisüsteemide-, side- ja sündmuste haldamise disainimallid.

Sisukord

1. *Disainimallide* määratlus. ("Mis on üks disaini mall")

   1.1. *Disainimallide* osad:.
   a) probleem või teema,
   b) lahendus,
   c) järeldus.

2. *Disainimallide* esitusviisid.

   2.1. Alexander'i kuju.
   2.2. Portland'i kuju.
   2.3. Coad'i kuju.
   2.4. Hüperklass.

3. *Disainimallide* klassifikatsioon.

   3.1. Loovad *disainimallid*.
   3.2. Struktuuri *disainimallid*.
   3.3. Käitumise *disainimallid*.

4. *Disainimallide* kasutamine objekt-orienteeritud tarkvara disainis. Programm kui idee esitus.
5. Korduvkasutatava tarkvara disainimise peamised probleemid.
6. *Disainimallide* rakendamine praktikas.
7. Raamistikud.
8. *Disainimallide* kataloog.

   8.1. Üldised *disainimallid*.
   8.2. *Disainimallid* hajatöötluses.
   8.3. *Disainimallid* ärisüsteemides.
   8.4. *Disainimallid* sides.
   8.5. *Disainimallid* sündmuste haldamises.

> *So the real work of any process of design lies in the task of making up the language, from which you can later generate the one particular design.*
>
> *And, more subtly, we find also that different patterns in different languages have underlying similarities, which suggest that they can be* ***reformulated*** *to make them*

more general, and usable in a greater variety of cases.

C. Alexander, *The Timeless Way of Building*

1. Disainimallide määratlus

1.1. Disainimallid

Objekt-orienteeritud keeltes ja programmeerimissüsteemides on toimunud areng klassiteekidest (1970) erinevaid ülesandeid komplektselt lahendavateks raamistikeks (*frameworks*). See areng kulmineerub arhitektuurielementide korduvkasutamisega.
Klassiteekides sisalduvad objektid on korduvkasutamise suurem ühik, kui alamprogrammid ja funktsioonid tavalistes programmiteekides, aga nad ei tõsta oluliselt arhitektuuri elementide korduvkasutamise taset. Põhiprobleem on, et laialt kasutatava klassi skoop pole küllalt suur, et oluliselt vähendada käsitsi kirjutatavat koodi.

Disaini mall on *korduv arhitektuuriline element*, mis lahendab mingit disainiprobleemide hulka kindlas kontekstis. Disainimallid aitavad parendada tarkvara arendust esitades edukaid ekspertlahendusi süstemaatilisel ja kergeltkasutataval kujul.
Mallide klassifitseerimine osutub üha tähtsamaks mallide kataloogide kasvades ja tarkvaratööstuse küpsedes punktini, kus disainerid kasutavad mitmeid mallide katalooge (mallide keeli) üheskoos. Tänapäeval on mallide katalooge alates suhteliselt abstraktsetest arhitektuuri raamistike mallidest, disainimallideni ja konkreetsete idioomideni. Arhitektuuri raamistike mallid (nagu MVC) on tavaliselt sõltumatud rakendusvaldkonnast. Vastupidiselt idioomideni (nagu viidete loendamine), mis on sageli tugevalt seotud konkreetse programmeerimiskeelega. Disainimallid pole sama süstemaatilised kui arhitektuuri raamistikud, aga ka mitte nii seotud programmeerimiskeelega kui idioomid.

Tarkvarasüsteemi, eriti aga suuremõõtmelise tööstustarkavarasüsteemi ehitamine on keerukas protsess. Tuleb teha mitmeid disainiotsuseid: sisse tuua mitmeid komponente, tarkvara funktisonaalsus tuleb nende komponentidega siduda, nende vahelised suhted tuleb määratleda ja kogu arhitektuur peab vastama teatud mittefunktsionaalsetele nõuetele.
Mallid koosnevad ettevalmistatud disainistruktuuridest, mida saab kasutada ehituskividena tarkavaraarhitektuuri ehitamiseks.
Iga mall:

- Annab ettemääratud skeemi mingi teatud struktuurse või funktsionaalse põhimõtte realiseerimiseks tarkvarasüsteemis, kirjeldades tema erinevaid osi ja nende koostegevust ning vastutusalasid.
- Kirjeldab olemasolevat, järeleproovitud disaini kogemust.
- Identifitseerib, nimetab, ja spetsifitseerib abstraktsioone, mis on klassidest ja isenditest kõrgemal.
- Pakub ühist sõnastiku ja disainipõhimõtetest arusaamist.
- Aitab tarkvara keerukust hallata.
- On tarkvara arenduses taaskasutatav ehituskivi.
- Võib olla kas rakendusvaldkonnast sõltumatu, või sõltuv.
- Puudutab nii tarkvaradisaini funktsionaalseid kui mittefunktsionaalseid külgi.

1.2. Mallide süsteem

Mallide süsteem, mida saab kasutada suvalise soovitava tarkvara arhitektuuri ehitamiseks ja koostamiseks, koosneb paljudest erinevatest mallidest, mida kasutatakse paljudel erinevatel otstarvetel. Kuna sellise süsteemi eesmärgiks on ettemääratud omadustega tarkvarasüsteemide süstemaatiline väljatöötamine, ei piisa mallide lühikirjelduste loetelust, et süsteemi saaks effektiivselt kasutada.
Kasutuskõlblik mallide süsteem peab:

- Toetama süsteemi arengut. Teatud mallid võivad muutuda süsteemi elutsükli jooksul, uusi malle võib lisanduda ja olemasolevad võivad kaduda.
- Kirjeldama kõiki malle, mida ta sisaldab ühesugusel kujul. Need kirjeldused peavad puudutama kõiki aspekte, mis on tähtsad malli iseloomustamisel, detailne kirjeldus, realisatsioon, valik, ja võrdlused teiste mallidega.
- Klassifitseerima malle, mida ta sisaldab, et juhatada mallide valikut konkreetse disainisituatsiooni jaoks. Selline klassifikatsioonisüsteem peab sisaldama kategooriaid kriteeriumite või disaini teemade kohta, mis mängivad tähtsat rolli tarkvara arenduses.
- Vaatlema teemasid, mis puudutavad mallidest keerukate ja heterogeensete struktuuride ehitamist. Mitte igat malli ei saa effektiivselt koos kasutada mistahes teise süsteemis oleva malliga. Lisaks mõjutab see, kuidas malle on koos kasutatud tulemuse omadusi nagu taaskasutatavus või muudetavus.

2. Disainimallide esitusviisid

Üldjuhul on mallil neli olulist osa:

- ·**Nimi** -- see on probleemi kirjelduse, lahenduse ja selle tagajärgede lühike (ühe-kahe sõnaline) iseloomustus. Malli nimi lisandub disaini-sõnastikule ja võimaldab rääkida, arutleda ja kirjutada mallist.
- ·**Probleem** kirjeldab, kus/kunas malli rakendada. Ta selgitab probleemi ja selle konteksti, kirjeldades sümptomaatilisi klassi- või objektistruktuure. Mõnikord sisaldab probleem tingimusi, mille korral mall on rakendatav.
- ·**Lahendus** kirjeldab elemente, mis moodustavad disaini, nende vahelisi suhteid, nende ülesandeid ja koostööd. Lahendus ei kirjelda konkreetset disaini või realisatsiooni, kuna malle võib rakendada mitmes situatsioonis.
- ·**Tagajärjed** on malli rakendamise tulemused ja hind. Tagajärjed on sageli tähtsad disaini alternatiivide hindamisel.

2.1. Alexander'i kuju

Alexander eristas mallidel kolme vajaliku osa:

- ·**Kontekst**: Mingi konkreetne korduv situatsioon
- ·**Probleem**: Jõudude süsteem, mis eksisteerib antud kontekstis
- ·**Lahendus**: Ruumiline konfiguratsioon, mis lubab diasineril probleemi lahendada

2.2. Gamma kuju

Erich Gamma'l on disainimallid (kasutab ka terminit mikro-arhitektuurid) kategoriseeritud (klassifitseeritud):

- *Loovad* mallid
- *Struktuuri* mallid
- *Käitumis*mallid

ja esitatud järgmisel kujul:

Malli nimi ja Klassifikatsioon
: Malli nimi annab edasi malli olemust.

Eesmärk
: Lühike tekst, mis vastab küsimustele:
Mida antud mall teeb?
Mis on põhimõte või eesmärk?
Millist disaini küsimust või probleemi antud mall lahendab?

Alias
: Malli teised tuntud nimed (kui neid on)

Motivatsioon
: Stsenaarium, mis illustreerib disaini probleemi ja seda, kuidas malli klassi- ja objektistruktuurid lahendavad seda probleemi.

Rakendatavus
: Situatsioonide kirjeldus, kus antud malli saab rakendada. Millised on halva disaini näited, mida antud mall parandab ja kuidas neid situatsioone ära tunda.

Struktuur
: Mallis osalevate klasside graafiline esitus (OMT notatsioonis). Samuti interaktsiooni diagrammid, mis kirjeldavad meetodikutsete järjestust ja koostööd.

Osalejad
: Klassid ja objektid, mis disainis osalevad ning nende ülesanded.

Koostöö
: Kuidas osalejad üheskoos täidavad oma ülesandeid.

Tagajärjed
: Kuidas mall saavutab oma eesmärgid? Milline on malli kasutamise hind? Milliseid süsteemi struktuuri aspekte on võimalik sõltumatult varieerida?

Realisatsioon
: Millised ohud võivad olla seotud realiseerimisega ja milliseid tehnikaid tuleks kasutada? Kas ja millised on keelest sõltuvad küsimused?

Näitekood
: Koodi fragmendid, mis illustreerivad, kuidas malli võiks realiseerida C++ või Smalltalk'is.

Teada kasutused
: Antud malli näited tegelikes süsteemides (vähemalt kaks erinevatest valdkondadest pärit näidet).

Seotud mallid
: Millised disainimallid on antud malliga seotud? Millised on tähtsamad erinevused sarnastest mallidest? Milliseid teisi malle antud mall kasutab?

2.2. Portland'i kuju

Pattern Name: Malli nimi **Aliases:** ***Aliases (or none)*** Malli teised nimed

Problem

Antud malli poolt lahendatava probleemi kirjeldus. Probleem võib olla esitatud küsimusena

Context

Probleemi konteksti kirjeldus

Forces

Probleemi ja lahendust mõjutavate jõudude kirjeldus. See võib olla loetelu
- Esimene jõud
- Teine jõud
- ...

Solution

Probleemi lahenduse kirjeldus

Resulting Context

Lahenduse konteksti kirjeldus

Rationale

Lahenduse taga oleva mõttekäigu selgitus

Known Uses

Loetle, kus seda malli on kasutatud

Related Patterns

Antud malliga seotud mallide loetelu või kirjeldus

Sketch

Joonis ja joonise kirjeldus

Author(s): Autori nimi
Date: Kuupäev, näit. 3/1/96
Send email to author(s)
Pattern Source: *Näide: AG Communication Systems, Writers Workshop, etc.*

References

Viidete nimestik

Keywords: Komadega eraldatud loetelu võtmesõnadest

Example
Näide malli kasutamisest

2.3. Coad'i kuju

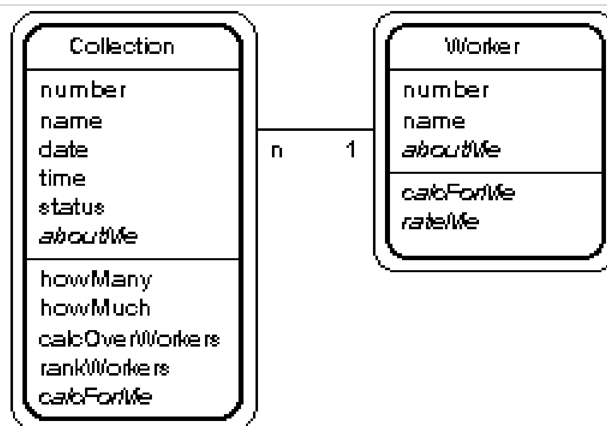
Coad'il on mallid stereotüüpsed objektide, vastutuste (*responsibilities*) ja interaktsioonide komplektid, mida võib korduvalt analoogiale põhinedes rakendada. Mallide isendid (*instances*) on objektmudelite ehituskivideks.
Malle võib kategoriseerida peredesse:

- fundamentaalsed (*fundamental*)
- transaktsioonid (*transactions*)
- liitolemid (*agregates*)
- vahendid (*devices*)
- interaktsioonid (*interactions*)
- kombinatsioonid (*combinations*)

Mallid on esitatud kujul:

#number. "Nimi" tüüp kategooria
skeem (Coad'i notatsioon)
kirjeldus

Näide:

<table>
<tr><td>#1. Mall "Kogum-Töötaja" (Collection-Worker)
fundamentaalne mall</td></tr>
<tr><td>
</td></tr>
<tr><td>• Kogum-Töötaja on objektmudeli fundamentaalne mall
• Kõik teised objektimudeli mallid on antud malli variatsioonid
• Tüüpilised objektide vahelised interaktsioonid:
<code>howMany --> calcForMe</code>
<code>howMuch --> calcForMe</code>
<code>calcOverWorkers --> calcForMe</code>
<code>rankWorkers --> rateMe</code></td></tr>
</table>

- Teisi märkusi
  `aboutMe` -- aitab mõtelda, milliseid atribuute veel vaja on
  `calcForMe` -- aitab mõtelda, milliseid konkreetseid arvutusi on vaja teostada
  `rankMe` -- aitab mõtelda, milliseid järjestus- ja võrdlusteenuseid vaja oleks
  `rateMe` -- aitab mõtelda, milliseid hinnanguteenuseid vaja oleks

Strateegia on tegevuste plaan, mis on ette nähtud teatud eesmärgi saavutamiseks.
Strateegiad jaotuvad nelja põhikategooriasse:

- süsteemi eesmärkide ja omaduste identifitseerimine
- objektide valimine
- vastutuste kindlaksmääramine
- dünaamika väljatöötamine stsenaariumite abil

Strateegiad on esitatud kujul:

#number. "Nimi" tüüp kategooria
kirjeldus

Näide:

#1. Strateegia "Neli Põhilist Tegevust, Neli Põhilist Komponenti" peamised tegevused ja komponendid
• Organiseeri oma töö nelja põhitegevuse ja nelja põhikomonendi ümber • Neli põhitegevust: Identifitseeri eesmärgid (*purpose*) ja omadused (*features*), vali objektid, sea paika ülesanded (*responsibilities*), tööta stsenaariumeid kasutades välja dünaamika • Neli põhikomponenti: Probleemivaldkond, iniminteraktsioon, andmete haldamine, süsteemi interaktsioon

2.4. Hüperklass

Jiri Soukup leiab, et Gamma ja Coad'i poolt kirjeldatud mallide probleemiks on, et nad aitavad disainida tarkvara abstraktsel tasemel aga ei säili realisatsioonis (koodis). Lahenduseks pakub ta ühe spetsiifilise klassi -- malli klassi (*pattern class*) või hüperklassi lisamist disaini. See klass oleks sõber (*friend*) kõikide klassidega, mis moodustavad malli ja ta sisaldaks malli kogu liidest (rakenduse klassides poleks ühtegi malliga seotud meetodit).
Hüperklass esitab abstraktsemaid mõisteid, kui tavaline klass tema jaoks on mallid isenditeks/objektideks.

3. Disainimallide klassifikatsioon

Alexander esitab oma töös mallide klassifikatsiooni aga mitte nende vaheliste suhete klassifikatsiooni. Gamma ja teised esitavad hulga hästikirjeldet ja klassifitseerit malle, kuid nende vahelised suhted pole klassifitseeritud. Nende klassifikatsiooniskeem, mis põhineb skoobil (klass, objekt, liitobjekt) ja iseloomustusel (loov, struktuurne, käitumuslik), on ortogonaalne siinkirjeldatule.

3.1. Disainimallide klassifikatsiooni skeemid E.Gamma järgi

3.1.1. Loovad disainimallid

Loovad disainimallid abstraheerivad isendite loomise protsessi (*instantiation*). Sellega aitavad nad teha tarkvarasüsteemi sõltumatuks sellest, kuidas objekte luuakse, ühendatakse ja kujutatakse. *Klassi* loovad disainimallid kasutavad pärimist et varieerida klassi, mille isendeid luuakse, aga *objekti* loovad disainimallid delegeerivad loomise teisele objektile.

Loovad disainimallid muutuvad tähtsaks, kui süsteem areneb rohkem kompositsioonist kui pärimisest sõltuvaks. Sellisel juhul siirdub rõhk ettemääratud käitumiste (*behavior*) kodeerimisest väiksema hulga fundamentaalsete käitumiste kirjeldamisele, mida saab kombineerida suvaliseks arvuks keerukamateks käitumisteks.

Kõik loovad disainimallid kapseldavad teadmise süsteemi kuuluvate konkreetsete klasside kohta ja nendesse klassidesse kuuluvate isendite loomise ja ühendamise viisid.

3.1.2. Struktuuri disainimallid

Struktuuri disainimallid tegelevad sellega, kuidas klassid ja objektid moodustavad suuremaid struktuure. *Klassi* struktuuri mallid kasutavad pärimist liideste või realisatsioonide komponeerimiseks (lihtsaim näide on mitmene pärivus). *Objekti* struktuuri disainimallid kirjeldavad kuidas koostada objekte, et tekiks uus funktsionaalsus. Sellisel juhul osutub võimalikuks muuta funktsionaalsust töö ajal.

3.1.3. Käitumise disainimallid

Käitumise disainimallid tegelevad algoritmide ja objektide vaheliste kohustuste jagamisega. Käitumise disainimallid ei kirjelda ainult objektide ja klasside malle vaid samuti nende vahelise suhtlemise malle. Neid malle iseloomustab keerukas juhtimisvoog, mida on töö ajal raske jälgida, selleks juhivad nad tähelepanu juhtimisvoolt viisile, kuidas objektid on ühendet.

Klassi käitumise disainimallid kasutavad pärimist, et käitumist klasside vahel jaotada. *Objekti* käitumise disainimallid kasutavad kompositsiooni pärimise asemel ja mõned neist näitavad, kuidas grupp objekte koostöös lahendab ülesandeid, mida ükski neist üksikult võttes ei saaks lahendada.

			Eesmärk	
		Loomine	**Struktuur**	**Käitumine**
Skoop	**Klass**	Tehasmeetod	Adapter (klass)	Interpretaator Näidismeetod
	Objekt	Abstraktne Tehas Ehitaja Prototüüp Üksik	Adapter (objekt) Sild Liitobjekt Dekoraator Fassaad Kärbeskaallane Asemik	Vastutuse Jada Käsk Iteraator Vahendaja Momentvõte Vaatleja Olek Strateegia Külastaja

3.2. Disainimallide klassifikatsiooni skeemid W. Zimmer'i järgi

Nagu eespool üteldud peab iga mallide süsteem sisaldama klassifikatsiooni skeemi. Mõistlik kategooriate hulk mallide klassifitseerimisel aitab kasutajat leida vajalikud mallid. Iga kategooria peab esitama selget kriteeriumit või disaini teemat, mis mängib tähtsat rolli tarkvara arenduses. Võib eristada järgnevaid kolme kategooriat.

3.2.1. Granulaarsus

Tarkvarasüsteemi väljatöötamine nõuab tegutsemist erinevatel abstraktsioonitasemetel. Granulaarsust võib määratleda:

- ·**Arhitektuurilised raamistikud**: Iga tarkvaraarhitektuur on ehitet vastavalt üldisele struktureerimise põhimõttele. Neid põhimõtteid kirjeldavad arhitektuurilised raamistikud (*architectural frameworks*):
  Arhitektuuriline raamistik esitab põhimõttelist vormide kogu (paradigmat) tarkvarasüsteemide struktureerimiseks. Ta annab ettemääratud alamsüsteemide hulga ja reeglid suhete loomiseks nende vahel.
  Arhitektuurilised raamistikud on kui näidised konkreetsetele tarkvara arhitektuuridele, nad määravad süsteemi struktuuri ja mõjutavad alamsüsteemide arhitektuuri. Seega lubavad arhitektuurilised raamistikud hallata struktuurset keerukust tarkvarasüsteemides. Mingi kindla arhitektuurilise raamistiku valik tarkvarasüsteemi jaoks on fundamentaalne disaini otsus.
  Näide: Mudel-Vaade-Kontroller.
- ·**Disainimallid**: Tarkvara arhitektuur koosneb tavaliselt mitmetest väiksematest arhitektuuri ühikutest, neid kirjeldavad mallid.
  Disaini mall kirjeldab tarkvara arhitektuuri alamsüsteemide ja komponentide struktureerimise põhiskeemi, ja nende vahelisi suhteid. Ta identifitseerib, nimetab ja abstraheerib üldist disainipõhimõtet, kirjeldades tema erinevaid osi ja nende koostöödning ülesandeid [Gamma].
  Disaini malle võib vaadelda mikroarhitektuuridena, disaini mall on väiksem, kui terviklik tarkvara arhitektuur või arhitektuuriline raamistik.
  Disaini mall võib olla seotud teiste disainimallidega ja koosneda mitmest väiksemast mallist.
  Näide: Mediaator.
- ·**Idioomid**: Idioomid tegelevad mingi disaini küsimuse konkreetsete realisatsioonidega.
  Idioom kirjeldab, kuidas realiseerida mingit malli osa, osa funktsionaalsust, või osade vahelist suhet. Nad on sageli konkreetsest programmeerimiskeelest sõltuvad.
  Idioomid on madalaima taseme mallid, nad on tugevalt seotud mingi kindla programmeerimiskeelega. Kui sama idioomi eksisteeribki erinevates programmeerimiskeeltes, on tema kuju erinev.
  Näide: Viidete loendamine C++ (see idioom ei oma mõtet Smalltalk'is, kuna seal on automaatne prügikoristus (*garbage collection*)).

3.2.2. Funktsionaalsus

Teine mallide klassifitseerimisekategooria on funktsionaalsus. Iga mall on eeskujuks (*template*) konkreetse funktsionaalsuse realiseerimisel. Võib eristada järgnevaid funktsionaalsuse klasse:

- ·**Objektide loomine (Loomine)**: Mallid mis kirjeldavad, kuidas luua keerukaid, rekursiivseid või kompleksseid objektide struktuure.
- ·**Objektide vaheline suhtlemine (Suhtlemine)**: Mallid mis kirjeldavad, kuidas

organiseerida koos töötavate objektide (mis võivad olla iseseisvalt välja töötatud või hajutatud) hulgas suhtlemist.

- ·**Juurdepääs objektidele (Juurdepääs)**: Mallid mis kirjeldavad, kuidas kasutada objektide poolt pakutavaid teenuseid ja pääseda juurde nende olekule ilma kapseldumist rikkumata.
- ·**Keerukate protsesside organiseerimine**: Mallid mis kirjeldavad, kuidas jagada ülesandeid koos töötavate objektide vahel, et lahendada keerukaid ülesandeid.

3.2.3. Struktuursed Põhimõtted

Et realiseerida oma funktsionaalsust, toetuvad mallid kindlatele arhitektuurilistele põhimõtetele, mis moodustavad kolmanda kategooria.

- ·**Abstraktsioon** (*abstraction*): Mall vaatleb abstraktselt või üldistatult konkreetset (sageli keerukat) olemit (*entity*) tarkvarasüsteemis
- ·**Kapseldumine** (e*ncapsulation*): Mall kapseldab mingi kindla objekti, komponendi või teenuse detailid, et teha tema kliendid neist sõltumatuteks või kaitsta neid juurdepääsu eest.
- ·**Huvide lahusus** (*separation of concerns*): Mall eristab konkreetsed ülesanded (*responsibilities*) eristatud objektidesse või komponentidesse, et konkreetset ülesannet (*task*) lahendada või konkreetset teenust osutada.
- ·**Seotus ja Kokkukuuluvus** (*coupling and cohesion*):Mall kaotab või nõrgendab struktuurseid või suhlemis seoseid ja sõltuvusi muidu tugevalt seotud objektide vahel.

Iga malli võib süsteemis klassifitseerida eelpooltoodud kategooriate järgi. Et konkreetses disainiolukorras klassifikatsioonisüsteemi kasutada, tuleb teha järgmist:

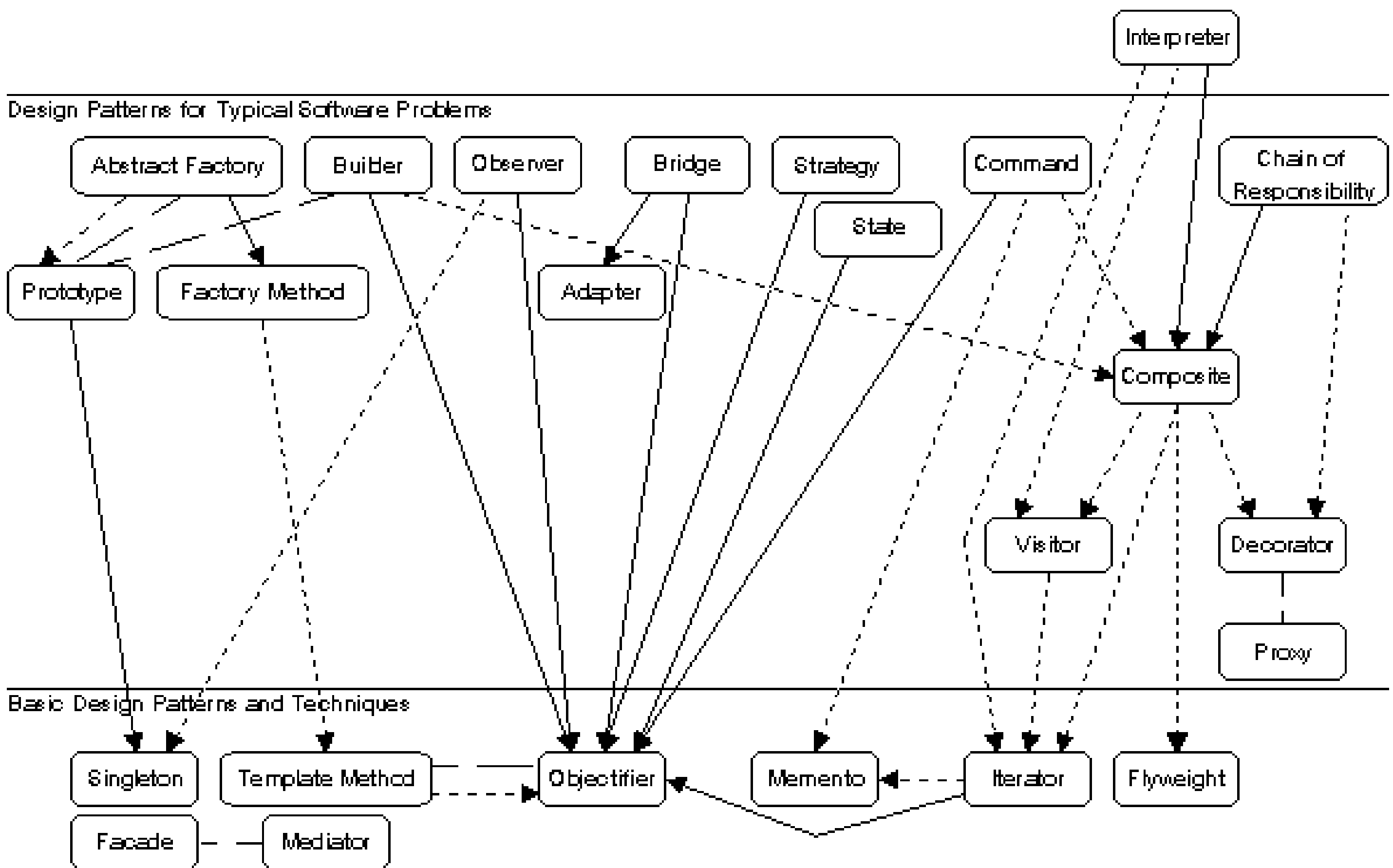
- *Määrata malli nõutud granulaarsus*. Kas mallil põhineb kogu rakendus, alamsüsteem või komponent, või konkreetse disaininõude realisatsioon..
- *Valida vajalik funktsionaalsus*. Kui on vaja kombineerida mitut funktsionaalsust, tuleb valida mall, mis annab kõik nõutavad funktsionaalsused ise, või mallid, mis annavad need koostöös.
- *Määrata soovitavad struktuursed põhimõtted*. Mall tuleb valida vastavalt antud olukorras kõige kasulikumale ja tähtsamale struktuursele põhimõttele.

Klassifikatsiooni süsteem pole mõeldud õige mall valikuks -- otsuse peab tegema disainer, aga ta aitab disainereid otsida antud olukorra jaoks sobivat malli.

Mallid saame jaotada semantiliselt erinevatesse tasemetesse:

- Põhilised disainimallid ja tehnikad
- Disainimallid tüüpiliste tarkvarasüsteemides esinevate probleemide lahendamiseks
- Rakendusvaldkonnale spetsiifilised disainimallid

Disainimallide jaotamine tasemetesse:



4. Disainimallide kasutamine objekt-orienteeritud tarkvara disainis

Kuidas valida disainimalli:

- *Vaadelda, kuidas disainimallid probleemi lahendavad*. Uurides, kuidas mallid aitavad leida vajalike objekte, määrata objektide granulaarsust, kirjeldada objektide liideseid, ja muid vahendeid, mida mallid kasutavad probleemide lahendamisel, võib leida oma disaini sobiva malli.
- *Lugedes mallide eesmärgikirjeldusi*. Tuleb leida mall, mille eesmärgi kirjeldus on lähedaseim lahendust nõudvale probleemile.
- *Uurida, kuidas on mallid omavahel seotud*. Mallide vaheliste suhete järgimine juhatab kätte vajaliku malli või mallide rühma.
- *Võrrelda sarnase otstarbega malle*. Kataloog (Gamma) jaotub kolmeks osaks: loovad mallid, struktuuri mallid ja käitumismallid.
- *Juurelda ümberdisainimise põhjuste üle*.
- *Mõtelda, mis peab olema teie disainis muutuv*. Keskenduda tuleb muutuva mõiste kapseldamisele.

Kuidas kasutada disaini malli:

- *Loe mall läbi, et saada ülevaadet*. Eriti kinnita tähelepanu rakendatavuse ja tagajärgede kirjeldustele, et tagada malli sobivus.
- *Uuri struktuuri, osalejate kirjeldusi ja koostöö kirjeldust*.
- *Vaatle näitekoodi*. See näitab, kuidas malli oleks õige realiseerida.
- *Vali nimed mallis osalejatele, mis on sinu rakenduse kontekstis mõtestatud*. Malli kirjelduses

kasutatud osalejate nimed on tavaliselt liialt abstraktsed.
- *Kirjelda klassid*. Kirjelda nende liideseid, pärivussuhteid ja isendimuutujaid. Erista rakenduses olemasolevad klassid, mida mall puudutab ja muuda neid vastavalt.
- *Kirjelda rakendusele omased nimed malli operatsioonide jaoks*.
- *Realiseeri operatsioonid, et mallis olevaid ülesandeid (responsibility) ja koostööd (collaborations) täita*.

Smalltalk'i MVC sisaldab mitmeid disaini malle, millest põhilised:

- Mudel ja Vaated on lahtiühendet Vaatlejat kasutades
- Vaated võivad olla paigutet üksteise sisse, mis viitab Liitobjekti kasutamisele
- Vaate ja Kontrolleri vaheline suhe vastab Strateegia'le

Esineb ka muid nagu Vaikimisi antava Kontroller'i klassi määrab Tehasmeetod ja Vaatele lubab lisada elemente Dekoraator

5. Korduvkasutatava tarkvara disainimise peamised probleemid

Kaks kõige laiemalt kasutatud tehnikat funktsionaalsuse taaskasutamiseks objekt-orienteeritud süsteemides on pärimine ja objektide kompositsioon. Pärimise korral kirjeldatakse ühe klassi liidest teise klassi liidese abil, sellist taaskasutamist nimetatakse sageli valge-kasti taaskasutuseks (viitates nähtavusele). Objektide kompositsioon on alternatiiviks pärimisele, siin saavutatakse uus funktsionaalsus objektide liitmisega. Objektide kompositsioon nõuab objektidelt hästi määratletud liideseid. Seda taaskasutuse stiili nimetatakse musta-kasti taaskasutuseks, kuna objektide sisemised detailid pole nähtavad.

Mõned olukorrad, mis tekitavad taaskasutamisel ümberdisainimist põhjustavaid probleeme, mida vastavate mallide abil saaks vältida:

- *Objekti loomisel klassi avalik näitamine*. See seob programmi konkreetse liidese asemel konkreetse realisatsiooniga. Selle vältimiseks tuleb luua objekte kaudselt.
  Kasutatavad mallid: Abstraktne Tehas, Tehas Meetod, Prototüüp.
- *Sõltuvus konkreetsetest operatsioonidest*. Näidates konkreetset operatsiooni seotakse programm slle konkreetse realisatsiooniga.
  Kasutatavd mallid: Vastutuse Jada (*Chain of Responsibility*), Käsk.
- *Sõltuvus riistvarast ja tarkvarast*. Välised operatsioonisüsteemi liidesed ja rakendusprogrammeerija liidesed (API) on erineva riist- ja tarkvaraplatvormidel erinevad.
  Kasutatavd mallid: Abstraktne Tehas, Sild.
- *Sõltuvus objektide esitusest ja realisatsioonist*. Kliente, kes teavad, kuidas objekt on ehitet, salvestet ja leitav, tuleks objekti realisatsiooni muutumisel muuta.
  Kasutatavd mallid: Abstraktne Tehas, Sild, Momentvõte (*Memento*), Asemik (*Proxy*).
- *Sõltuvus konkreetsetest algoritmidest*. Algoritme laiendatakse, optimiseeritakse ja vahetatakse sageli arenduse ja taaskasutuse käigus. Objekte, mis nendest sõltuvad tuleks vastavalt muuta.
  Kasutatavd mallid: Ehitaja, Iteraator, Strateegia, Näidismeetod (*Template Method*), Külastaja (*Visitor*).
- *Tihe side*. Klasse, mis on tihedalt seotud (*coupled*) on raske taaskasutada. Tihe side viib monoliitsete süsteemide tekkimisele, kus eisaa midagi muuta ilma mitmest klassist aru saamata ja neid muutmata.
  Kasutatavad mallid: Abstraktne Tehas, Sild, Vastutuse Jada, Käsk, Fassaad, Vahendaja, Vaatleja.

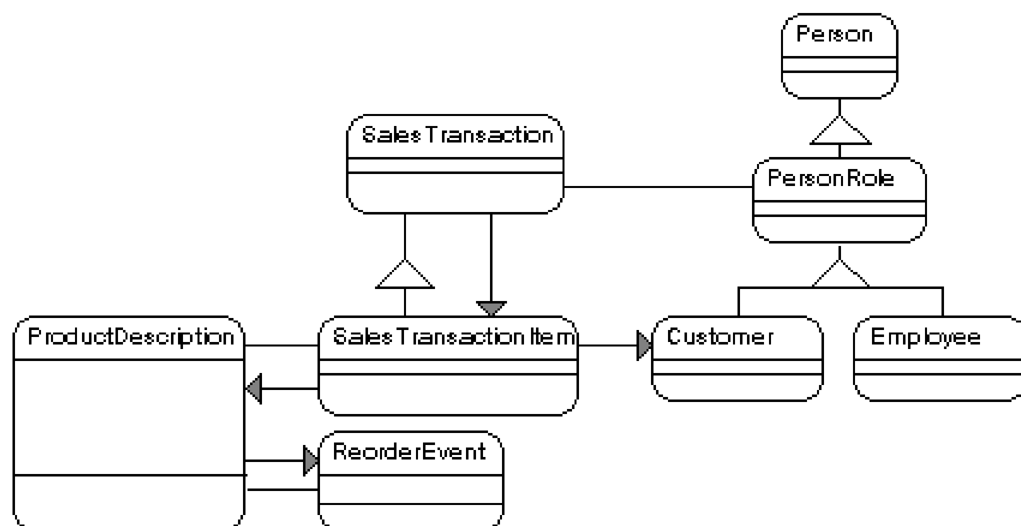
- *Funktsionaalsuse laiendamine pärivuse abil*. Objekti spetsialiseerimine pärimist kasutades on sageli raske. Iga uus klass lisab koos funktsionaalsusega ka kindla hulga vältimatuid tegevusi (alustamine, lõpetamine, ...), samuti nõuab pärimise kasutamine päritava klassi täielikuu tundmist. Pärimise kasutamine võib viia plahvatuslikule klasside hulga kasvule (ühe alamklassi tegemine võib nõuda teiste alamklasside tegemist). Objektide kompositsioon ja delegatsioon pakuvad paindliku alternatiivi pärimise kasutamisele käitumise muutmisel. Teisest küljest kompositsiooni laialdane kasutamine raskendab disainist arusaamist.
  Kasutatavd mallid: Sild, Vastutuse Jada, Liitobjekt (*Composite*), Dekoraator, Vaatleja, Strateegia.
- *Võimatus klasse muuta*. Vahel on vaja muuta klassi, mida ei saa kas lähtekoodi puudumise tõttu või mõnel muul põhjusel lihtsalt muuta.
  Kasutatavd mallid: Adapter, Dekoraator, Külastaja.

6. Disainimallide rakendamine praktikas

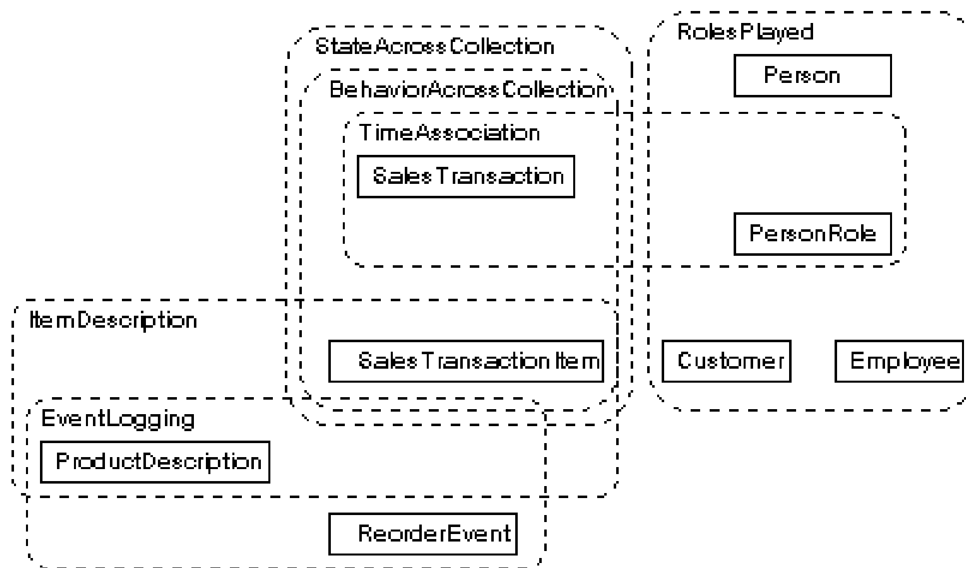
Probleemid mallide realiseerimisel:

1. Praeguse realiseerimisstiili korral kaovad mallid kodeerimisel. See põhjustab hilisemaid probleeme silumisel ja hooldamisel.
2. Mitme malli kooskasutamine põhjustab suure hulga üksteisest sõltuvate klasside teket.
3. Siiani puudub konkreetsete taaskasutatavate mallide teek.

Eelnevast loetelust probleemid 1. ja 2. saab lahendada esitades igat malli erilise klassiga, mida nimetame Malli klass (ka hüperklass). See klass kapseldab kogu malli käitumise ja loogika. Coad'i poolt toodud mallide kasutamise näites:



On kaasatud kuus malli:



Lisaks rakendusest tulenevatele klassidele tuleks lisada malliklassid, mis omaksid suhteid rakenduse klassidega (*friend*), jättes rakenduse klassid üksteisest sõltumatuks.

7. Raamistikud

Raamistik (*framework*) on hulk koostöötavaid klasse, mis moodustavad taaskasutatava disaini mingisse konkreetsesse klassi kuuluva tarkvara jaoks. Raamistik määrab kogu rakenduse arhitektuuri, tema jaotumise klassidesse ja objektidesse, tähtsamad kohustused, seega ka kuidas klassid ning objektid koos töötavad ja juhtimisvoo. Raamistik tähtsustab disaini taaskasutamist koodi taaskasutamise asemel, ehkki tavaliselt sisaldavad raamistikud konkreetseid klasse, mida saab koheselt kasutada.

Raamistike erinevused disainimallidest:

- *Disaini raamistikud on raamistikest abstraktsemad*. Raamistike saab kehastada koodis, aga ainult disaini raamistike näiteid võib koodis kehastada. Raamistike jõud ongi selles, et neid võib valmis kodeerida ja seega valmilt taaskasutada. Disainimallid tuleb iga kord uuesti realiseerida aga nad kirjeldavad eesmärke ja lahenduse hinda ning tagajärgi.
- *Disainimallid on väiksem arhitektuuriline ühik, kui raamistikud*. Tavalisekt sisaldab raamisti mitut disaini malli.
- *Disainimallid on vähem spetsialiseeritud, kui raamistikud*. Raamistikel on alati konkreetne rakendusvaldkond aga disaini malle saab kasutada rakendusvaldkonnast sõltumata. Isegi kui siduda disaini mall mingi konkreetsema valdkonnaga nagi hajussüsteemid või paralleelprogrammeerimine ei dikteeri nad rakenduse arhitektuuri, nagu raamistikud.

Raamistikud sisaldavad rakendusvaldkonna üldisi abstraktsioone -- nende struktuuri ja mehhanisme, jättes rakenduse-spetsiifilised struktuurid ja käitumise rakenduse väljatöötajale. Raamistike tüüpilised esindajad on MacApp, ET++, Interviews, Choices, MFC, ... Raamistik on korduvalt kasutatav disain, mis on esitatud abstraktsete klassidega ja viisiga kuidas nende klasside isendid koos töötavad.
Raamistik:

- sisaldab integreeritud, rakendusvaldkonnast sõltuvat funktsionaalsust (klassiteegi klassid on tavaliselt valdkonnast sõltumatud)
- omab töö ajal juhtimist (klassiteegile baseeruv disain asetab juhtimisvoo rakendusse)

- on poolik rakendus (rakenduse lõpetamine toimub parametriseerimisega ja abstraktsete klasside spetsialiseerimisega)

Raamistiku juhtimisvoog on juhitud tagasikutsete (*callback*) poolt.

Raamistikele orienteeritud disainimallide põhiline eesmärk on kirjeldada raamistikku ja temas olevaid klasse ilma realisatsiooni paljastamata. Võib sisse viia metamalli mõiste, mis kirjeldab kuidas sõltumata rakendusvaldkonnast koostada raamistike.
Raamistike disainimisel on põhiliseks printsiibiks kitsa pärimisliidese printsiip (*narrow inheritance interface principle*) -- Weinand, mis ütleb: 'käitumine, mis on hajutet üle mitme meetodi, peab baseeruma minimaalsel meetodite hulgal, mida peab ülekirjeldama'. Vastasel juhul peavad kliendid alamklassides palju meetodeid ülekirjeldama, et ühte käitumist seada.
Mõned rusikareeglid:

- *Nõrk side klasside (nende isendite) vahel* -- objektid peaks vahetama nii vähe andmeid kui võimalik.
- *Tugev side klassi sees* -- ei tohi olla ei meetodeid ega andmeid ilma suheteta; saavutatav kirjeldades klassiga vaid ühte abstraktsiooni.
- *Minimaalne vajalik liides* -- tuleb vältida sama teenuse osutamist mitme väikeste erinevustega liidese kaudu.
- *Testitavus* -- klassi õigsust peab olema võimalik kontrollida teadmata millises kontekstis klassi kasutatakse.

Klasside/Objektide liidesed ja suhtlemine

Näidis- ja konksmetodid (*Template and Hook methods*)
Näidismeetodid kirjeldavad abstraktset käitumist, üldist juhtimisvoogu või objektide vahelisi suhteid.
Näidismeetodid põhinevad konksmeetoditel, mis võivad olla abstraktsed meetodid.
Keerukad meetodid on näidismeetodid ja nad on realiseeritud konksmeetodite abil.

Klasside/Objektide kompositsioon (Class/Object Composition)

Näidis- ja konksklassid (*Template and Hook classes*)
Konksklass parametriseerib näidisklassi. Üldiselt võib näidisklassi ja konksklassi vahel luua sideme:

- isendimuutuja abil, mis viitab konksklassile
- edastades viite objektile kui meetodi parameetri
- globaalse muutujaga, mis viitab konksklassile

8. Disainimallide kataloog

Gamma disainimallide kataloog:

Otstarve	**Disaini Mall**	**Eesmärk**
Loomine (*Creational*)	Abstraktne Tehas (*Abstract Factory*) {Komplekt -- *Kit*}	tekitada liides seotud sõltuvate objektide loomiseks ilma nende konkreetset klassi näitamata

	Ehitaja (*Builder*)	eraldada keeruka objekti ehitamine tema esitusest, et sama ehitusprotsess võiks luua erinevaid esitusi
	Tehasmeetod (*Factory Method*) {Näivkonstruktor -- *Virtual Constructor*}	kirjeldada liides objekti loomiseks jättes alamklassidele võimalus otsustada, millisesse klassi isend luua
	Prototüüp (*Prototype*)	määrata millised objektid tuleb luua prototüüpse isendi alusel ja luua uued objektid kopeerides seda isendit
	Üksik (*Singleton*)	tagada, et klassil on ainult üks isend ja anda talle globaalne juurdepääs
Struktuur (*Structural*)	Adapter (*Adapter*) {Ümbrik -- *Wrapper*}	muundada klassi liides selliseks mida ootab klient; lubab koos töötada klassidel, millede liidesed ei sobi
	Sild (*Bridge*) {Käsik/Keha -- *Handle/Body*}	eraldada abstraktsioon oma realisatsioonist nii, et mõlemad võivad iseseisvalt muutuda
	Liitobjekt (*Composite*)	liita objekte puustruktuurideks, mis kujutavad osa-tervik hierarhiaid; lubab klientidel üksikobjekte ja liitobjekte ühetaoliselt käsitleda
	Dekoraator (*Decorator*) {Ümbrik -- *Wrapper*}	lisada dünaamiliselt objektile lisafunktsionaalsust (*responsibilities*); pakub paindlikku alternatiivi pärimisele funktsionaalsuse laiendamisel
	Fassaad (*Facade*)	pakkuda ühetaolist liidest tervele liideste hulgale alamsüsteemis; kirjeldab kõrgema taseme liidese, mis teeb alamsüsteemi kergemini kasutatavaks
	Kärbeskaallane (*Flyweight*)	kasutada jagamist (*sharing*) suure hulga väikeste objektide effektiivseks realiseerimiseks
	Asemik (*Proxy*) {Surrogaat -- *Surrogate*}	pakkuda surrogaatobjekti mingi teise objekti asemel, et kontrollida talle juurdepääsu
Käitumine (*Behavioral*)	Vastutuse Jada (*Chain of Responsibility*)	vältida päringu saatja ja vastuvõtja vahelist tihedat sidestust (*coupling*), andes rohkemale, kui ühele objektile võimalus

		päringut täita; paigutada vastuvõtjad jadasse ja anda päringut piki jada edasi kuni leidub objekt, kes ta täidab
	Käsk (*Command*) {Tegevus, Transaktsioon -- *Action, Transaction*}	kapseldada päring kui objekt, lubades parametriseerida kliente erinevate päringutega, paigutada päringuid järjekorda, pidada päringute kohta logi ja toetada tühistatavaid operatsioone
	Interpretaator (*Interpreter*)	kirjeldada antud keele grammatikale esitus koos interpretaatoriga, mis kasutab seda esitust
	Iterator (*Iterator*) {Kursor -- *Cursor*}	pakkuda järjestikust juurdepääsu objektide struktuuri (hulga) elementidele ilma realisatsiooni paljastamata
	Vahendaja (*Mediator*) {Haldur -- *Manager*}	kirjeldada objekt, mis kapseldab hulga teiste objektide suhtlemise viisi; vahendaja hoiab objekte üksteisele viitamast ja lubab nende vahelist suhtlemist neist sõltumatult varieerida
	Momentvõte (*Memento*) {Suveniir? -- *Token*}	esitada ilma kapseldumist lõhkumata väliselt objekti sisemine olek, et seda saaks salvestada ja hiljem taastada
	Vaatleja (*Observer*) {Sõltlased, Avalda-Telli -- *Dependents, Publish-Subscribe*}	määrata üks-mitmele suhe objektide vahel nii, te kui ühe objekti olek muutub, kõik temast sõltuvad objektid saavad teada ja võivad end uuendada
	Olek (*State*)	lubab objekti muuta oma käitumist vastavalt sisemisele olekule
	Strateegia (*Strategy*) {Poliitika -- *Policy*}	määrata algoritmide pere, kapseldada nad ja muuta vahetatavateks; lubab muuta algoritmi klientidest sõltuvalt
	Näidismeetod (*Template Method*)	määrata algoritmi struktuur (*skeleton*) viivitades mingite osade kirjeldust alamklassideni; lubada alamklassidel varieerida algoritmi osi
	Külastaja (*Visitor*)	esitada operatsiooni, mida tuleb rakendada objektide struktuuri osadele; lubab kirjeldada

		uusi operatsioone, muutmata nende elementide klasse, milledel ta opereerib
Otstarve	**Disaini Mall**	**Aspektid, mis võivad muutuda**
Loomine	Abstraktne Tehas (*Abstract Factory*)	toodetavate objektide pered
(*Creational*)	Ehitaja (*Builder*)	viis, kuidas liitobjekti luuakse
	Tehasmeetod (*Factory Method*)	objekti alamklass, millesse isend luuakse
	Prototüüp (*Prototype*)	objekti klass, millesse isend luuakse
	Üksik (*Singleton*)	klassi ainus isend
Struktuur	Adapter (*Adapter*)	objekti liides
(*Structural*)	Sild (*Bridge*)	objekti realisatsioon
	Liitobjekt (*Composite*)	objekti struktuur la koosteviis
	Dekoraator (*Decorator*)	objekti ülesanded ilma pärimiseta
	Fassaad (*Facade*)	alamsüsteemi liides
	Kärbeskaallane (*Flyweight*)	objketide mäluvajadus
	Asemik (*Proxy*)	juurdepääs objektile; objekti asukoht
Käitumine	Vastutuse Jada (*Chain of Responsibility*)	objekt, mis päringu täidab
(*Behavioral*)	Käsk (*Command*)	kus ja kuidas päring täidetakse
	Interpretaator (*Interpreter*)	keele grammatika ja semantika
	Iterator (*Iterator*)	juurdepääs hulga elementidele ja viis kuidas hulka läbitakse
	Vahendaja (*Mediator*)	kuidas ja millised objektid suhtlevad
	Momentvõte (*Memento*)	milline sisemine info ja kunas on objektist

		väljaspoole salvestatud
	Vaatleja (*Observer*)	objektide hulk, mis sõltub objektist; viis kuidas sõltuvaid objekte uuendatakse
	Olek (*State*)	objekti olekud
	Strateegia (*Strategy*)	algoritm
	Näidismeetod (*Template Method*)	algoritmi sammud
	Külastaja (*Visitor*)	operatsioonid, mida rakendatakse objektidele, ilma nende klasse muutmata

Gamma disainimallide kataloogi üldine struktuur:

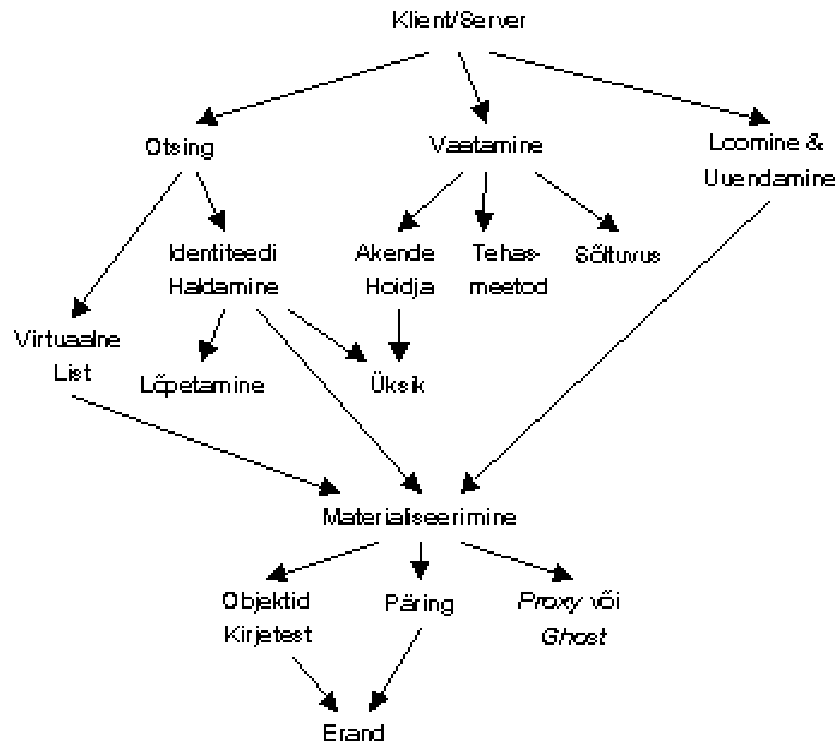
Mallid klient/server süsteemide tegemiseks mis sisaldavad liideseid vanadesse (*legacy*) süsteemidesse:

Nimi	**Probleem**	**Lahendus**

Erand (*Exception*)	Ebanormaalsed asjad juhtuvad tavaliselt kaugel punktist kus programm peaks antud olukorrale vastama.	Probleemide määratlemine klassi *Exception* täpsustustena lubab ehitada tingimuste hierarhiat ja uhendada lahti probleemi tuvastamist ning lahendamist.
Üksik (*Solitaire (Singleton)*)	Mõnedest objektidest on rakenduses vaid üks isend.	Üksiku kaks rolli, on tähtsaimad: *identifikaatorid* -- objektide identiteedi leidmiseks ja *hoidjad* -- mingi ressursi haldajad.
Objektid Kirjetest (*Objects from Record*)	Andmevahetus OO klientide ja vanade andmebaasisüsteemide vahel on komplitseeritud keeruka struktuuriga objektmudeli ja lameda relatsioonilise mudeli vaheliste erinevuste tõttu.	Andmetele juurdepääs nihutatakse Äriobjektist selleks ettenähtud Kirjeobjekti (*Record object*). Äriobjekt ümritseb objekti isendiandmeid esitava Kirje vajaliku käitumisega.
Päring (*Request*)	Klient/Server süsteemid kasutavad erinevaid sideliideseid/-protokolle andmevahetuseks keskarvuti rakendustega, millede erinevused peaksid olema nähtamatud kliendile.	Klient loob Päringu objekti, mis atomaarselt vahetab hulga sisendkirjeid hulga väljundkirjete vastu. Seega kapseldab Päring ühe serveri transaktsiooni (tööühiku).
Materialiseerimine (*Materialization*)	Objekt-orienteeritud klient/server süsteemi põhiomaduseks on kliendi objektide kujutamine serveris, hoolimata sellest, kas server toetab objekte või ei. Lisaks peab kliendi rakenduskood olema sõltumatu sellest kujutusest.	Materialiseerimise mall kasutab Päringu ja Objektid Kirjetest malle serveri andmete objektideks muundamiseks, kasutades *proxy* või *ghost*'i malli. *Proxy* asub mingi objekti ja tema kasutajate vahel, ning näib olevat antud objekt, viies sisse kaudsuse, mis maskeerib, millise astmeni antud objekt on materialiseerunud. *Ghost* ei oma käitumist ja esimese teate saamisel materialiseerib ta tegeliku objekti ning "muundub" temaks. *Proxy* omab tegeliku objekti liidest ja jääb tegelikku objekti ümbritsema, olles tema ainuke väline liides.
Lõpetamine (*Finalization*)	Rakendustes on sageli vaja midagi ette võtta, kui objekt või viimane viit objektile eemaldatakse -- näiteks operatsioonisüsteemi ressursid.	Lõpetamise malli realiseerimiseks võib kasutada viitade lugemist (*reference counting*).
Identiteedi Haldamine (*Identity Management*)	Kuna objektid materialiseeritakse identiteedita kirjetest, võib rakendus materialiseerida sama objekti mitu koopiat, mis võtavad kasutult ruumi ja lisavad ebajärjekindlate	Et takistada äriobjektide paljusust, kasutatakse Identiteedi Haldamise malli. Vastutus lasub abstraktsel klassil Fail (*File*). Fail on Üksik, mis haldab äriobjektide identiteeti, kindlustades, et

	versioonide (*inconsistent versions*) tekkimise riski.	kui kaks päringut tagastavad sama objekti, viitab teine olemasolevale objektile selle asemel, et luua uus koopia.
Virtuaalne List (*Mega-Scrolling (Virual Lists)*)	Päringu poolt tagastatud objektide hulk võib olla liiga suur, et seda kliendis materialiseerida.	Virtuaalne List piirab tagastatava hulga suurust. Kui selline piiratud suurusega alamhulk ei rahulda kasutajat, koostab Virtuaalne List uue päringu eelneva alamhulgaga külgneva alamhulga saamiseks. Virtuaalse Listi realisatsioon võib sisaldada küllalt infot, et iga järgnev alamhulk oleks serveri jaoks uue transaktsiooni tulemus, või toetuda ka püsivatele ressurssidele serveris, nagu SQL kursor.
Äriobjektide Otsing (*Searching for Business Objects*)	Huvipakkuva objekti leidmine on nii elementaarne tegevus, et on risk seda võtta enesestmõistetavana.	Äriobjektide Otsingu võib objektiseerida (*reify*) kirjeldades abstraktse klassi Otsing. Otsingu klassi kuuluvad objektid määravad otsingukriteeriumi. Fail saab otsingu protsessis Otsingu objekti kui parameetri ja tulemuseks on List või üksik objekt. Otsing kasutab Identiteedi Haldamist.
Sõltuvus (*Dependency (Model-View, Broadcast, Observer, MVC)*)	Äriobjektide ja neid kuvavate ning juhtivate vaateobjektide ühendamine on halb, kuna ta teeb koodi keerukaks ja varjab kontseptuaalse mudeli -- mudeli käitumine osutub seotuks kasutajaliidese käitumisega.	Esmatööd mudeli objektide ja nende vaadete lahti ühendamisel tehti 70-ndatel Xerox'i PARC'is MVC (Model-View-Controller) malli väljatöötamisel. Tänapäeval on algne MVC triaad liitunud MV düaadiks. Olles vanim ja OO mall on teda aegade jooksul kirjeldatud paljude erinevate nimede all. Kuna seda malli saab rakendada ka kasutajaliideseta, nimetame teda Sõltuvuseks.
Äriobjektide Loomine ja Uuendamine (*Creating and Updating Business Objects*)	Äriobjektid peavad tekkima ja muutuma ajas. Need tegevused saavad alguse kliendis ja peavad lõpuks peegelduma serveris. Need tegevused muutuvad keerukamaks, kui objektid sõltuvad üksteisest.	Fail kasutab materialiseerimist muudatuste rakendamiseks äriobjektidele. Uuendused peavad levima sõltuvatele objektidele, et ka need saaksid uueneda, seda saab teha kasutades malli Sõltuvus.

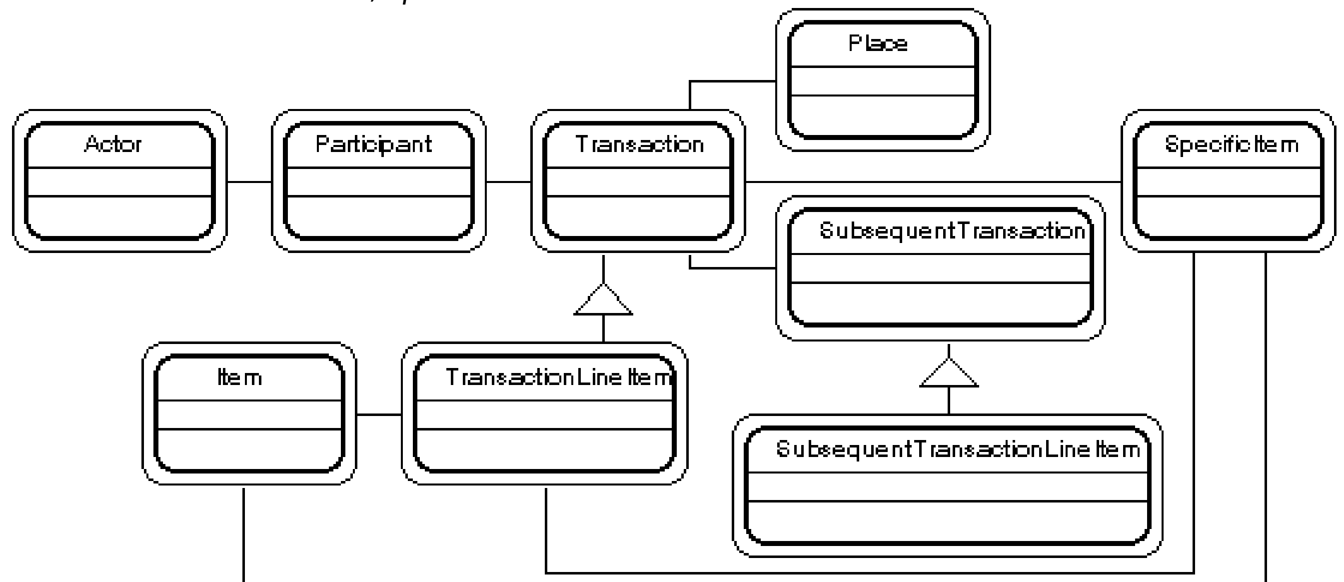
Tehas-meetod (*Factory Method*)	Luues vaateobjekti mudelile, on parem, kui kliendi kood teem minimaalselt eeldusi. Üldjuhul ei tohiks klient teada isegi vaate klassi.	Tehas-meetod (Gamma), loob olemasoleva objektiga seotud objekti ja vabastab oma kliendi vajadusest teada, milline see objekt peab olema.
Akende Hoidmine (*Window-Keeping*)	Keerukate akende ehitamine iga kord algusest peale võib olla kallis. Samuti tekib probleem, kui rakendus soovib avada akent äriobjektile, millele on juba aken avatud -- uue akna loomise asemel peaks olemasolev aken kõikide peale tõstetama.	AknaHoidja klass on Üksik, mis haldab kõiki rakenduse aknaid. AknaHoidja võib suletud aknaid varuks hoida, mitte hävitada, et neid järgmise avamisnõude korral uuesti kasutada. Samuti võib ta juba avatud akna leida ja teiste akende peale tuua.
Vaatamine (*Viewing*)	Äriobjektide leidmise ja uuendamise kõrval on rakenduse poolt nõutavatest funktsioonidest puudu veel vaatamine.	Vaatamine nõuab tavaliselt laia funktsioonide valikut, mida pakub mingi kasutajaliidese tööriistade komplekt. Lisaks sellele vajab Vaatamine veel Tehas-meetodi, Sõltuvuse, ja Akende Hoidmise malle eelpoolmainitud probleemide lahendamiseks.
Klient/Server Raamistik (*A Client/Server Framework*)	Objekt-orienteeritud klient/server raamistik, mis on ette nähtud vanade suurarvutite süsteemidega suhtlemiseks, peab lubama kasutajal leida teda huvitavaid objekte, neid vaadelda ja muuta (või luua).	Seega on Klien/Server Raamistik Otsingu, Vaatamise ja Loomise/Uuendamise mallide kogum.



Coad'i strateegiad ja mallid:

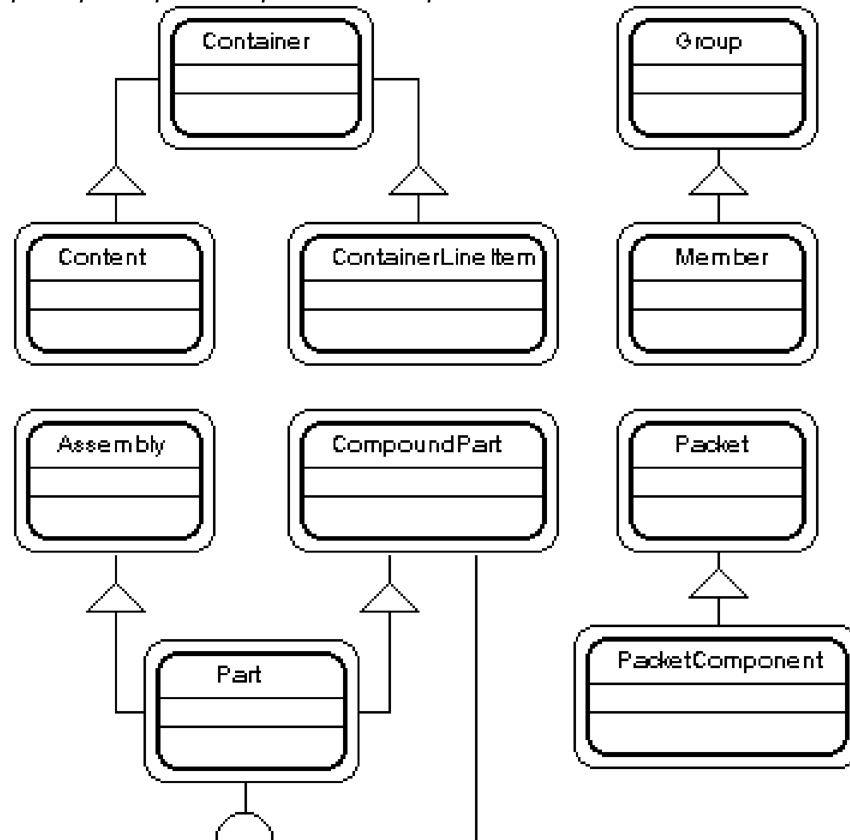
Objektimudeli ehitamise mallid:

- Fundamentaalne mall
- Transaktsiooni mallid
  actor-participant, participant-transaction, place-transaction, specific item-transaction, transaction-transcation line item, transaction-subsequent transaction, transaction line item-subsequent transaction line item, item-line item, specific item-line item, item-specific item, associate-other associate, specific item-hierarchical item

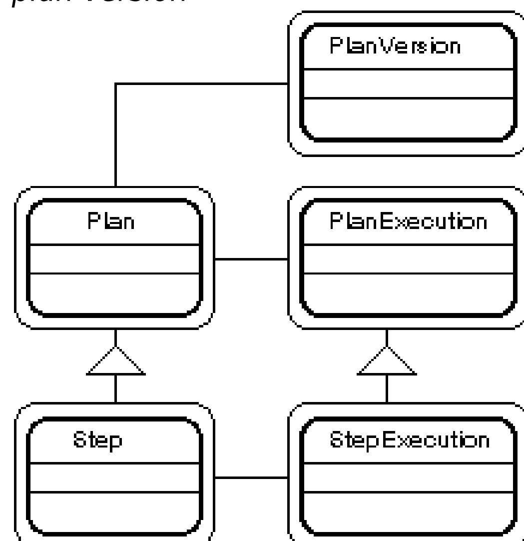


- Liitobjektide mallid
  container-content, container-container line item, group-member, assembly-part, compound

part-part, packet-packet component



- Plaanide mallid
  plan-step, plan-plan execution, plan execution-step execution, step-step execution, plan-plan version



- Interaktsiooni mallid
  peer-perr, proxy-specific item, publisher-subscriber, sender-pass through-receiver, sender-lookup-receiver, caller-dispatcher-caller back, gategeeper-request-resource

Strateegiad oma strateegiate ja mallide leidmiseks:

#142. Strateegia "Uute Strateegiate avastamine" avastamine (strateegiad)
• Analüüsi igat väikest sammu, mida sa teed objektmudeli mingi osa ehitamisel, millist nõu võiksid anda teistele, et nad sama ülesandega hakkama saaksid

- Analüüsi igat objektmudeli parandust, millist nõu võiksid anda teistele, et vältida samu vigu

#143. Strateegia "Strateegiate täpsustamine"

avastamine (strateegiad)

- Kirjelda, mida sa tegid -- sinu strateegia
- Rakenda seda mitu korda (kui võimalik, eri valdkondades)
- Anna talle nimi ja kategoriseeri ta
- Jaga teda teistega -- õpi nende reaktsioonidest

#144. Strateegia "Strateegiate kirjeldamine"

avastamine (strateegiad)

- Lisa kirjeldusse:
  - nimi (vahetu siht) ja kategooria
  - strateegia ise (kasutades "kuidas teen ..." vormi)
  - näited

#145. Strateegia "Uute Mallide avastamine"

avastamine (mallid)

- Uuri igat suhtlevate objektide paari, kolmikut, nelikut (jne.)
- Üldista osaliste nimed
- Analüüsi kas teda saab rakendada teistes valdkondades

#146. Strateegia "Uute Mallide nimetamine"

avastamine (mallid)

- Otsi nime, mis kirjeldab osalisi suhtluses, mida nad tüüpiliselt teevad ja teavad
- Anna mallile osaliste järgi nimi
  - Vaatle sünonüüme
  - Vaatle üldisemaid nimesid
  - Vaatle metafoore analoogsetes süsteemides

#147. Strateegia "Mallide täpsustamine"

avastamine (mallid)

- Kirjelda mall
- Kasuta teda mitu korda (kui võimalik, eri valdkondades)
- Kategoriseeri mall (transaktsioon, liitobjekt, seade, interaktsioon, kombinatsioon, ...)
- Jaga teda teistega -- õpi nende reaktsioonidest

#148. Strateegia "Mallide kirjeldamine"

avastamine (mallid)

- Lisa kirjeldusse:
  - nimi ja kategooria
  - mall ise -- objektmudeli näidis (*template*)
  - tüüpilised interaktsioonid
  - näited
  - kombinatsioonid

Kategooria	Strateegia	Kirjeldus (tõlge)
Peamised tegevused ja komponendid	1. Neli Põhitegevust, Neli Põhikomponenti	• Organiseeri oma töö nelja põhitegevuse ja nelja põhikomonendi ümber • Neli põhitegevust: Identifitseeri eesmärgid (*purpose*) ja omadused (*features*), vali objektid, sea paika ülesanded (*responsibilities*), tööta stsenaariumeid kasutades välja dünaamika • Neli põhikomponenti: Probleemivaldkond, iniminteraktsioon, andmete haldamine, süsteemi interaktsioon
Identifitseeri eesmärgid ja omadused	2. Süsteemi Eesmärk 3. Töö välja peal, Pildid ja Näited 4. Selgita peamised stressi allikad 5. Tööta välja omaduste nimekiri 6. Nelja liiki omadused 7. Arvutustulemused ja Otsustuspunktid 8. Parimad ja Halvimad omadused 9. Esikümme 10. Nüüd ja hiljem 11. Ümberehitused piiridel 12. Nupukad seadmed	2. *System Purpose* 3. *Field Trips, Pictures, and Examples* 4. *Identify Major Sources of Stress* 5. *Develop a Features List* 6. *Four Kinds of Features* 7. *Calculation Results and Decision Points* 8. *Best and Worst Features* 9. *Top 10* 10. *Now and Later* 11. *Reengineering on the Boundaries* 12. *Smarter Devices*
Vali objektid (malli mängijad)	13. Vali tegutsejad (*actors*) 14. Vali osalejad 15. Vali kohad 16. Vali kombatavad asjad 17. Vali transaktsioonid 18. Vali liitlased 19. Vali elemendid ja konkreetsed elemendid 20. Vali suhtlevad süsteemid ja seadmed 21. Vali objektide kogumid 22. Vali sisaldavad (konteiner-) objektid 23. Nimeta kogum 24. Kasuta vähimat võimaliku kogumit	13. *Select Actors* 14. *Select Participants* 15. *Select Places* 16. *Select Tangible Things* 17. *Select Transactions* 18. *Select Associates* 19. *Select Items and Specific Items* 20. *Select Interacting Systems and Devices* 21. *Select Collections of Objects* 22. *Select Container Objects* 23. *Select a Collection* 24. *Select the Smallest Applicable Collection*

Kirjandus

1. E. Gamma et al., **Design Patterns**: Elements of Reusable Object-Oriented Software, 1994, pp.

395.
2. W. Pree, **Design Patterns for Object-Oriented Software Development**, 1995, pp. 268.
3. Ed. by J. O. Coplien & D. C. Schmidt**, Pattern Languages of Program Design**, 1995, pp. 562.
4. P. Coad, D. North, M. Mayfield**, Object Models**: Strategies, Patterns, & Applications, 1995, pp. 505.
5. J. Soukup, **Taming C++**: Pattern Classes and Persistence for Large Projects, 1994, pp. 416.