

Ce presupune ingineria sistemelor soft?

1. Modelare = gestionarea complexității, prin focusarea pe aspectele relevante
  2. Rezolvare de probleme = - modele folosite pt. căutarea unei soluții  
- evaluarea diferențelor alternative
  3. Stocurile de cunoștințe = modelarea presupun căutare de info., organizarea și formalizarea lor
  4. Argumentare = înțelegerea impactului unei eventuale schimbări asupra sistemului
- Model = reprezentare abstractă a unui sistem; reprezentare simplificată
- ale domeniului problemei: pt. a înțelege mediul în care sistemul operează
  - ale domeniului soluției: pt. a înțelege sistemul care va fi construit și pt. a evalua soluțiile și alternativele posibile

ISS 00: modelul domeniului de soluție se obține natural ca & transformare a modelului dom. problemei

Rezolvare prob leme: 1. Formularea problemei

2. Analiza problemei
3. Căutarea de soluții
4. Alegerea unei soluții
5. Specificarea soluției

Dexvoltarea: 1. Colecțarea cerințelor (\*): definirea scopului sistemului

2. Analiza cerințelor:
3. Proiectare de sistem: descompunerea sistemului în subsisteme: dom prob → dom soluție
4. Proiectarea obiectuală: definirea de obiecte, clase
5. Implementare: tranzformarea modelului obiectual în cod cursă
6. Testarea: identificarea diferențelor între sistemul implementat și modelele sale

ISS - activitate inginerescă, nu algoritmică: schimbările ce au loc în domeniul problemei și al soluției în timpul rezolvării problemei

System = un ansamblu de părți interconectate

Cerinte funcționale = specificarea a unei funcționalități pe care sistemul va trebui să o ofere  
• nefuncționale = constrângere legată de operarea sistemului

Metodologii de dezvoltare a softului: descompun procesul în activități:

- OMT - Analiză + Proiectare de sistem + Proiectare obiectuală
- USPT - Colecțarea cerințelor + Analiză + Proiectare

Activități tehnice ale ISS 00: 1. Colecțarea cerințelor, (\*)

Testare unitară: se compară modelul obiectual de proiectare cu fiecare obiect și subsistem  
de integrare: se integrează subsisteme  
de sistem: comportamentul sistemului

Tipuri de modele: • funcțional: descrie funcționalitățile → diagrama unei case

• obiectual (structural): clase, atribute, associeri → clasă diagramă

1. modelul de analiză/conceptual: descrie conceptele din domeniul problemei relevante pt. sistem

2. modelul proiectare de sistem: rafinarea modelului conceptual

3. model de proiectare: rafinarea modelului obiectual → descrierea detaliată a obiectelor din domeniul soluției

• dinamic: comportamentul intern al sistemului - diagrama de interacțiune (mesajele schițate); tranzitivitatea stăriilor, de activități (fluxuri de date și de control)

## Diagramme

### 1. Use case : ablon :

- nume
- actori
- flux de evenimente
- preconditii
- postconditii
- cerinta de calitate

- un caz de utilizare este o abstractizare se acopera toate scenariile posibile aferente functionalitatii descrise

- scenario = instanta a unui caz de utilizare, ce descrie o secventa concreta de actiuni

- relatii:
  - de comunicare : caz de utilizare si actor comunica : asociere UML binara, bidirectionala
  - de inclusiune : dependenta intre 2 cazeuri de utilizare « include »
  - de extindere : dependenta intre 2 cazeuri de utilizare « extend »
  - de generalizare :  $\rightarrow$

### 2. Clas diagram

- obiecte : pot primi nume

- asocieri si legaturi : conexiune

• asociere unidirectionala  $\rightarrow$   
bidirectionala  $\leftrightarrow$  OR  $\text{---}$

- multiplicatii si roluri

• rol = clarificare scopul asociiei si permit diferenzierea intre asocii diferite

• multiplicitate = indica nr. de obiecte legaturi pe care o instanta a clasei de la capatul opus il poate avea cu instantele clasei de la acel capat

- clase asociate  $\leftrightarrow$  clasa + asocieri simple + constrainte de unicitate

- generalizare: metoda de reducere a multiplicitatilor, prin utilizarea cheilor

- agregare: o relatie de tip parte - intreg (oare este format din)

• partajata : partile pot exista si independent

• compunere : o parte poate fi continuta in cel mult un intreg  
- intregul controland durata de viata a partilor

- generalizarea: permite factorizarea atributelor si operatiilor comune unei multimi de clase

- utilizarea .. in etapa de analiza a cerintelor : clasele corespunz entitatilor + asocieri,

stabilirea tipurilor atributelor

• in proiectare de sistem si obiectuala : clasele sunt grupate in subsisteme

### 3. De interacsiune

- descriu subloane de comunicare ; schimb de mesaje

- tipuri de secventi

de comunicare:

### 4. De tranzitie a starii

- pt. a descrie succesiunea de stari prin care trece un obiect sub actiunea evenimentelor externe

5. De activitati : modul de realizare a unui anumit comportament in termenii uneia sau a mai multor secvente de activitati si a fluxurilor de obiecte necesare

Flux de control = sezonierarea acțiunilor într-un sistem.

## Mécanisme:

- flux de control procedural
  - flux de control dirigit de evenimente
  - thread - uni

## Proiectarea obiectuală

- = activitățile tehnice ale ingineriei software reduc, în măsură graduală, discrepanța problemă-mașină fizică /calculator, prin identificarea /definirea obiectelor care compun viitorul sistem.
  - Analiză = identificarea obiectelor aferente conceptelor din domeniul problemei.
  - Proiectare de sistem
    - = definirea VM, prin selectarea de componente predefinite pentru servicii standard
    - = soluții

#### • Proiectare obiectivă

- = identificarea de noi obiecte din domeniul soluției, rafinarea celor existente, adaptarea componentelor reutilizate, specificarea riguroasă a interfețelor subsistemelor și dosetelor componente

freativitate : reutilizare

- specificarea interfețelor
  - restructurarea modelului obiectual
  - optimizarea modelului obiectual

Obiecte din domeniul problemei / soluției

problemei: concepte ale domeniului relevant pt. sistem

~~Soluției~~: concepte fără corespondență la nivelul domeniului problemei

Etapele identificării obiectelor din domeniul problemei/soluției

- analiză: se identifică obiectele din teren și se determină problema
  - proiectare des.: — " — soluției
  - proiectare obiectuală: se refinează și detaliază obiectele

Moștenirea specificării vs. moștenirea implementării

- în etapa de analiză, măștenirea este utilizată pt. clasificarea obiectelor în taxonomii
  - în etapa de proiectare obiectuală, rolul măștenirii este acela de a elimina redundanțele din modelul obiectual și de a asigura modificabilitatea/extenzibilitatea sistemului

⇒ m. implementării = utilizarea moștenirii în cadrul dreptului unic scop reutilizarea codului

m. specificării / interfeței = clasificarea conceptelor în hierarhii

Delegare = o alternativă la moștenirea implementării, atunci când se dorește implementarea unei funcționalități prin reutilizare.

- clasa delegă unei alte clase doar implementarea operației retransmînd mesajul aferent către cea din urmă

Principiul Rirkov al substitutiei = oferă o definiție formală conceptului de moștenire a specificării

= dacă pt. fiecare obiect  $O_1$  de tipul  $S$  există o altă de tipul  $T$ , a.î. orice program  $P$  definit în termenul lui  $T$  își păstrează comportamentul când  $O_1$  este substituit cu  $O_1'$ , atunci  $S$  este un subtip al lui  $T$ .

### Principiile SOLID

1. Single responsibility principle = nu trebuie să existe niciodată un motiv pentru ca o clasa să sufere modificări.  $\Leftrightarrow$  nu trebuie să existe mai mult de o responsabilitate/clasă.
2. Open-closed principle: Software entities should be open for extension, but not for modification.
3. Interface Segregation Principle: Clients should not be forced to depend upon interfaces that they do not use.
4. Dependency Inversion Principle: High level modules should not depend on low level modules, both should depend upon abstractions.  
Abstractions should not depend upon details, details should depend upon abstractions.

### Design Patterns

#### Adapter

Tranf. interfața unei clase existente într-o altă interfață care este așteptată de client, a.î. clientul și clasa să poată colabora fără nici o modificare la nivelul acestora.

#### Bridge

Decouplează o abstracție de implementarea ei, astfel încât ele două să poată varia independent. Implementările posibile vor putea fi astfel substituite la execuție.

#### Strategy

Defineste o familie de algoritmi, încapsulă fiecare algoritm și îi face interschimbabilă. Permite algoritmului să varieze independent de clientii care il utilizează.

#### Composite

Permite reprezentarea unor ierarhii de lătime și adăunarea variabilei, astfel î. frunzele și agăreatele să fie tratate uniform, prin intermediul unei interfețe comune.

#### Abstract Factory

Furnizează o interfață pentru crearea familiilor de obiecte înrudite sau dependente, fără specificarea clărelor lor con concrete.

#### Command

Încapsulează o cerere ca și un obiect, permitând parametrizarea clientilor cu diferite cereri, precum și formarea unei cozi sau a unui register de cereri și configurarea suportului pentru operațiile ce pot fi anulate.

+ State & Singleton

## Principiul baza de arhitectura

Proiectarea de sistem = procesul de transformare a modelului rezultat din ingineria cerintelor într-un model arhitectural al sistemului.

- produse:
  - obiectivele de proiectare (design goals): calitate și optimizare
  - arhitectura sistemului: subsistemele componente
    - responsabilitatele subsistemelor și dependențele între ele
    - moșona subsistemelor la hardware
- activități ale proiectării de sistem:
  1. Identificarea obiectivelor de proiectare
  2. Descompunerea inițială a sistemului
  3. Rafinarea descompunerii inițiale în vederea atingerii obiectivelor de proiectare

10.  
ion.

- concepte:

• Subsisteme și clase:

• Subsistem = parte înlocuibilă a unui sistem, caracterizată prin interfețe bine definite, care încapsulează starea și comportamentul elementelor componente

Serviciu = multime de operații înrudite

Interfață = multime de operații UML înrudite, complet specificate (nume, tipuri parametri, tip API (Application Programming Interface) = specificarea unei interfețe subsistem într-un <sup>retinut</sup> limbaj de programare

Cozieruire: măsura dependențelor din interiorul unui subsistem

Cuplare: măsura a dependenței dintre două subsisteme.

Descompunerea ierarhică a unui sistem

Arhitectură închisă: fiecare strat poate accesa doar servicii din stratul imediat inferior

Arhitectură deschisă: un strat poate accesa servicii din oricare dintre straturile superioare

Stiluri arhitecturale:

1. Repository: subsistenele accesează și modifică o structură de date centralizată, repositoriu
2. MVC: subsistene încadrăte în cele 3 categorii
3. Client-Server: serverul oferă servicii clientilor
4. Peer-to-Peer: un subsistem poate fi și client și server
5. Three/Tour-tier architecture (6.): subsistenele organizate pe nivele
6. Pipes and filters: filter - subsistene care efectuează un pas de procesare  
pipe - conexiune dintre 2 pași de procesare

Facade - permite reducerea dependențelor (cupărării) dintre un subsistem și clientii săi, oferă o interfață unică, de nivel înalt, pentru un grup de interfețe ale unui subsistem, care facilitează utilizarea acestuia.

Aplicabilitate: oferirea unei interfețe simple către un subsistem complex.

Diminuarea nemăreștilor de dependențe între clienti și clasele de implementare ale unei abstractizări

Stratificarea unui subsistem

## Proxy

Sablonul asigură, peține un obiect, un surrogat sau un înlocuitor, în scopul controlării acestuia la acesta.

Proxy la distanță - oferă un reprezentant local peține un obiect dintr-un spațiu de adresă diferit.

Proxy virtual - crează la cerere obiecte existente.

Cerintă = funcționalitatea pe care sistemul trebuie să îl ofere sau o constrângere

Ingineria cerințelor = scop: definirea cerințelor

- activități ale colectării cerințelor bazate pe scenarii: (2)

1. identificarea acționilor:

2. identificarea scenariilor

3. identificarea cazurilor de utilizare

4. refinarea cazurilor de utilizare

5. identificarea relațiilor dintre cazurile de utilizare

6. identificarea cerințelor nefuncționale

- validarea continuă a cerințelor reprezintă un imperativ în procesul de dezvoltare

- validarea cerințelor presupune verificarea completitudinii, consistenței, clarității și corectitudinii

Ingineria Greenfield = procesul de dezvoltare începe de la 0

: cerințele furnizate doar de clienti și utilizatori

Re-inginerie = reproiectarea și reimplementarea unei sisteme existente

Ingineria interfețelor = reproiectarea și reimplementarea doar a interfeței

Colectarea cerințelor - activități tehnice: (2)

Tipuri de cerințe nefuncționale: Utilizabilitate

2. Fiabilitate

3. Suportabilitate

4. Performanță

5. Implementare

6. Interfață

7. Instalare

8. Operare

9. Legal

Metoda JAD (Joint Application Design)

activități: 1. Project definition

2. Research

3. Preparation

4. Session

5. Final document preparation

Modelul obiectual de analiză: concepte manipulabile de sistem, proprietățile și relațiile acestora

| diagrama de clase

o clasa din model → abstractizare pentru i sau + close din codul sursă

Modelul dinamic: surprinde comportamentul sistemului

diagrama de sevență și tranziție a stărilor

• «entity»: responsabile de informația persistență din sistem

• «boundary»: responsabile de interacțiunea actorilor cu sistemul

• «control»: responsabile de realizarea cazurilor de utilizare

Generalizarea: identificarea unor concepte abstrakte, pornind de la unele de nivel înălțat

Specializarea: identificarea unor concepte specializate, pornind de la unele de nivel înălțat

Diagramale de sevență, în etapa de analiză, permit identificarea unor descrieri de

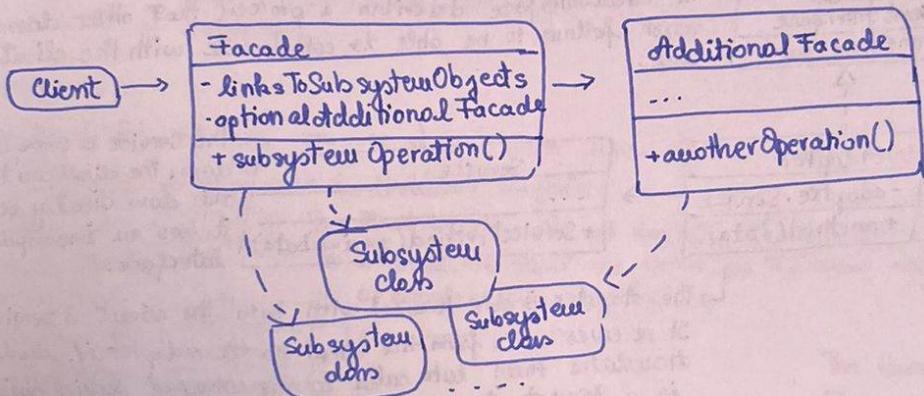
comportamente ambigue sau a unor clase/obiecte participante omise.

## Design Patterns

- Facade
- Proxy
- Adapter
- Bridge
- Strategy
- Composite
- Abstract Factory
- Command
- State
- Singleton

### 1. Facade

- structural design pattern



The Facade provides convenient access to a particular part of the subsystem's functionality. It knows where to direct the client's request and how to operate all the moving parts. An additional Facade class can be created to prevent polluting a single facade with unrelated features that might make it yet another complex structure.

The complex subsystem consists of dozens of various objects. To make them all do something meaningful, you have to dive deep into the subsystem's impl. details, such as initiating objects in the correct order and supplying them with data in the proper format. Subsystem classes aren't aware of the facade's existence. They operate within the system and work with each other directly.

The Client uses the facade instead of calling the subsystem objects directly.

### 2. Proxy

- structural design pattern that lets you provide a substitute or placeholder for another object. A proxy controls access to the original object, allowing you to perform something either before or after the request gets through to the original object.

Producători existenți până în momentul etapei de specificare a interfețelor.

- modelul obiectual de analiză
- decompunerea sistemului
- maparea hardware / software
- tabloane de proiectare reutilizate, componente de bibliotecă reutilizate pt. structuri de date și servicii de bază

Subactivități în specificarea interfețelor:

- identificarea atributelor și operațiilor lipsă
- specificarea vizibilității și semnaturii operațiilor
- specificarea pre/post - condițiilor pentru operații și a invariантelor de tip

OCL (Object Constraint Language)

- tipuri primitive: Integer, UnlimitedNatural, Real, Boolean, String
- tipuri specifice: Ocl Any, Ocl Void, Ocl Invalid, Ocl Message
- tipuri colecție: Collection, Set, Ordered Set, Sequence, Bag
- Enumeration, TupleType
- operații:
  - ocl IsKindOf (type : Classifier)
  - ocl Type()
  - ocl AsType()
  - ocl IsNew()
  - ocl IsUndefined()
  - ocl IsValid()

- constrângeri de tip invariant: context X  
inv invX1: self.a >= 0.

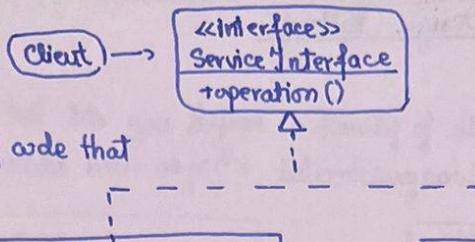
- constrângeri de tip precondition / postcondition: context X:: subtract (a: Integer)  
pre subtract Pre: self.a >= m  
post subtract Post: self.a = self.a @ pre-n

- mecanismul set: context X  
inv invX2: let all T: Bag (T) = self. #.t in  
all T → size () = all T → asSet () → size

- constrângeri de tip definition: context X  
def: has Y: Boolean = not self.y. ocl IsUndefined()  
def: has Z With BValue (value: String): Boolean =  
self.z → exists (z # | z.b = value)

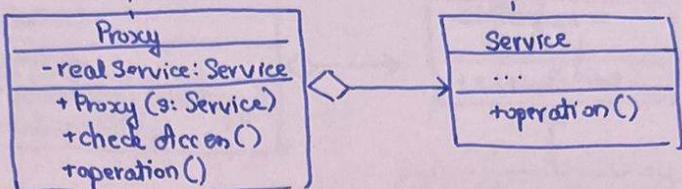
- iteratori: collection → select (v: Type | boolean - exp - with - v)  
collection → collect (v: Type | expression - with - v)  
collection → iterate (elem: Type ; acc: Type = <expr> | expr-with-elem-&-acc)  
e.x. collection → iterate (x: T; acc: Bag (T) = Bag { }) | acc → including (x.property))

The Client should work with both services and proxies via the same interface. This way you can pass a proxy into any code that expects a service object.



The Service Interface defines the API of the Service. The proxy must follow this if. to be able to disguise itself as a service object.

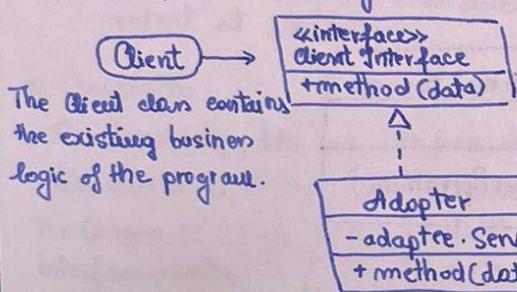
The Proxy has a ref. field to the service. After the proxy finishes processing, it passes the requests to the service obj.



The Service is a class that provides some useful business logic.

### 3. Adapter

- structural, that allows objects with incompatible interfaces to collaborate.



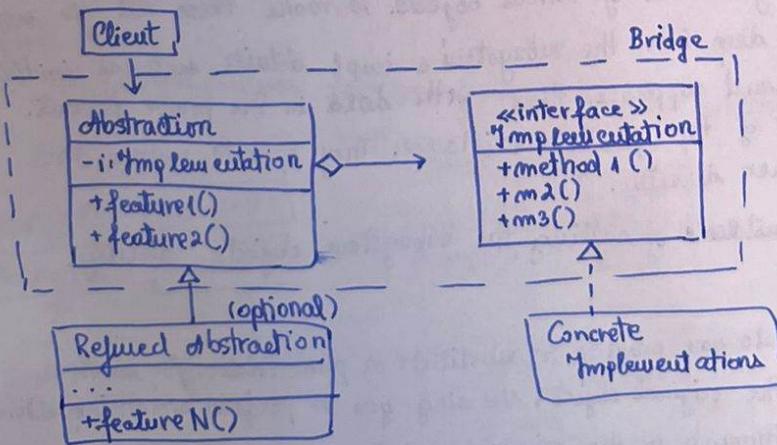
The Client Interface describes a protocol that other classes must follow to be able to collaborate with the client code.

The Service is some useful class. The client can't use this class directly because it has an incompatible interface.

The adapter is able to work with both the client & service. It receives calls from the client via the adapter if. and translates them into calls to the wrapped service object in a format it can understand.

### 4. Bridge

- structural, lets you split a large class or a set of closely related classes into 2 separate hierarchies - abstraction & implementation - which can be developed independently of each other



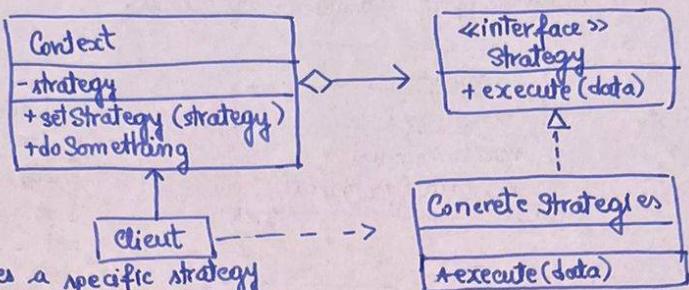
The abstraction provides high-level control logic. It relies on the implementation object to do the actual low-level work.

Usually, the Client is only interested in working with the abstraction. However, it's the Client's job to link the abstraction object with one of the implementation objects.

## 5. Strategy

behavioral design pattern that lets you define a family of algorithms, put each of them into a separate class and make their objects interchangeable.

The Context maintains a reference to one of the concrete strategies and communicates with them via the same strategy interface.



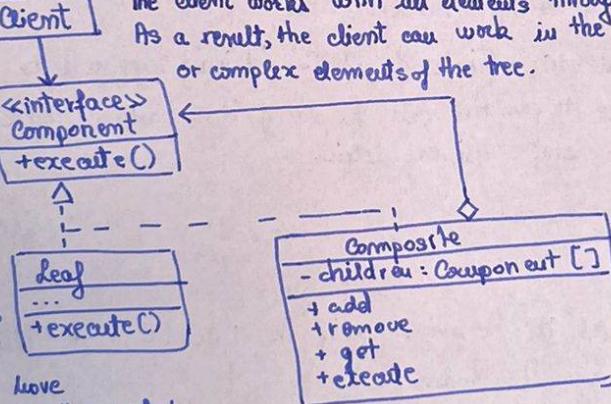
## 6. Composite

structural d.p., lets you compose objects into tree structures and then work with these structures as if they were individual objects.

The Component interface describes operations that are common to both simple and complex elements of the tree.

Leaf elements do the "real work" since they don't have anyone to delegate the work to.

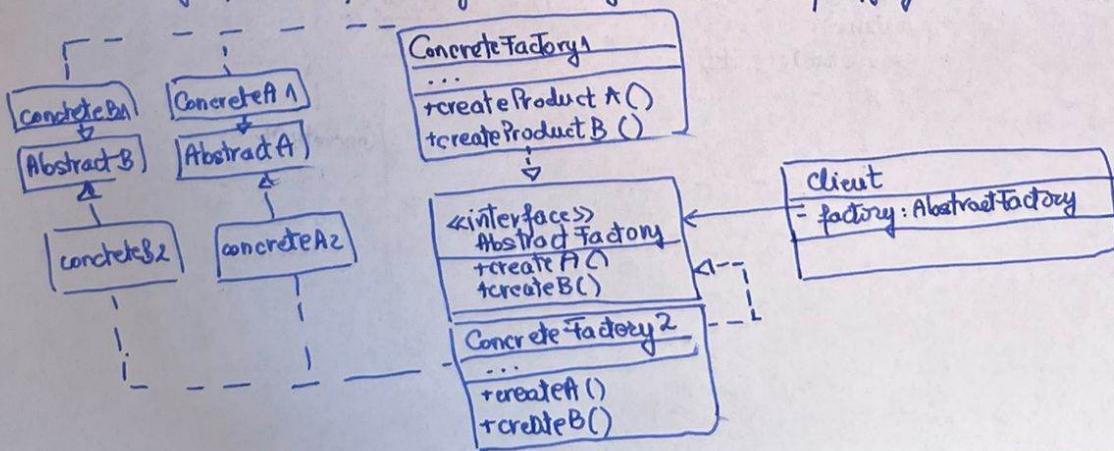
The Client works with all elements through the component interface. As a result, the client can work in the same way with both simple or complex elements of the tree.



The Component has sub-elements. Upon receiving a request, a container delegates the work to its sub-elements, processes intermediate results and returns the final result to the client.

## 7. Abstract Factory

-creational, lets you produce families of related objects without specifying their concrete classes



## Menținerea contradicțiilor

- precondiții: unei metode dintr-o subclasa îi este permis să slăbească precondiția metodelor pe care o supradefineste
- postcondiții: o metodă care supradefineste trebuie să asigure o postcondiție cel puțin la fel de puternică precum cea supradefinită.
- invariante: o subclasa trebuie să respecte toti invariantele superclasei sale; poate, eventual, introduce invariante mai puternice decât cele menținute

## Modele și transformări

- transformare: îmbunătățește caracteristici a unui model cu păstrarea ceealetă proprietăți ale acestuia

optimizare

representarea asocierilor

representarea contradicțiilor

represențarea entităților persistente

- transformări la nivelul modelului: transformarea unui atribut

- refacțiori: operație pe cod surșă  
îmbunătățesc un aspect al sistemului, fără a-i afectua funcționalitatea

- inginerie directă: produce un fragment de cod aferent unui model obiectual  
multe dintr-conceptele de modelare pot fi trans. în cod surșă

- inginerie inversă: produce un model, pe baza unui fragment de cod surșă  
utilă atunci când modelul de proiectare nu (mai) există sau atunci când  
modelul și codul au evoluat desincronizat

- principiu de transformare:

1. fiecare t. trebuie să vixere optimizări din perspectiva unui singur criteriu
2. - " - trebuie să fie locală
3. - " - trebuie aplicată izolat de alte schimbări
4. - " - trebuie urmată de validări aferente

## Reprezentarea contradicțiilor

• verificarea precondițiilor/postcondițiilor/invariantei

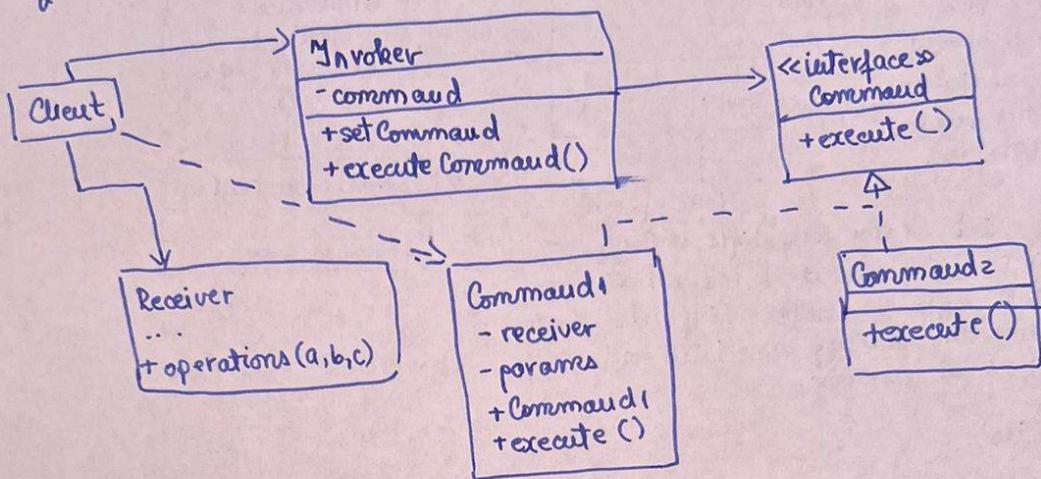
## Reprezentarea entităților persistente

• maparea verticală: adăugarea unui atribut în clasa de bază => adăugarea unei coloane în tabelul aferent => interogări mai lente

• maparea orizontală: adăugarea unei att. în clasa de bază => adăugarea unei coloane în fiecare altă tabel aferent claselor derivate; adăugarea unei noi clase derivate  
... => interogări rapide

### 8. Command

- behavioral, turns a request into a stand-alone object that contains all information about the request. This transformation lets you pass requests as method arguments, delay or queue a request's execution, and support undoable operations.



The **Invoker** is responsible for initiating requests. The sender triggers that command instead of sending the request directly to the receiver.

Concrete Commands implement various kinds of requests. It isn't supposed to perform the work on its own, but rather to pass the call to one of the business logic objects. The Receiver class contains some business logic.

### 9. State

- behavioral, lets an object alter its behavior when its internal state changes. It appears as if the object changes its class.

