Principii SOLID

conf.dr. Cristian KEVORCHIAN
ck@fmi.unibuc.ro

Cuplarea

- Ori de câte ori o clasă **A** folosește o altă clasă sau interfață **B**, atunci **A** depinde de **B**. A nu își poate desfășura activitatea fără B, iar A nu poate fi refolosit fără a reutiliza B. În asemenea situație, clasa A se numește „dependentă” și clasa sau interfața B se numește „dependență”. Orice dependent depinde de dependențele sale.
- Două clase care se folosesc reciproc se numesc „cuplate”. Cuplarea dintre clase poate fi slabă sau puternică sau undeva în acest interval.

Managementul Dependențelor (MD)

Când crește dependența o serie de atribute cum ar fi reutilizarea, flexibilitatea și mentenanța codului, scad.

Managementul dependențelor este identificat prin controlul interdependențelor.

Legătura dintre MD și procesul dezvoltării de software

Cuplarea reprezintă modul în care unobiect nu modifică sau modifică direct starea sau comportamentul unui alt obiect.

Coeziunea reprezintă gradul în care elementele unui modul interoperează împreună. Pe de o parte, este o măsură a puterii relației dintre metodele și datele unei clase și un anumit scop sau concept unificator expus de acea clasă. Pe de alta parte, este o măsură a puterii relației dintre metodele clasei și datele în sine.

Putem spune că OO este o familie de instrumente și tehnici pentru managementul dependențelor

Consecințele unei practici defectuase a MD

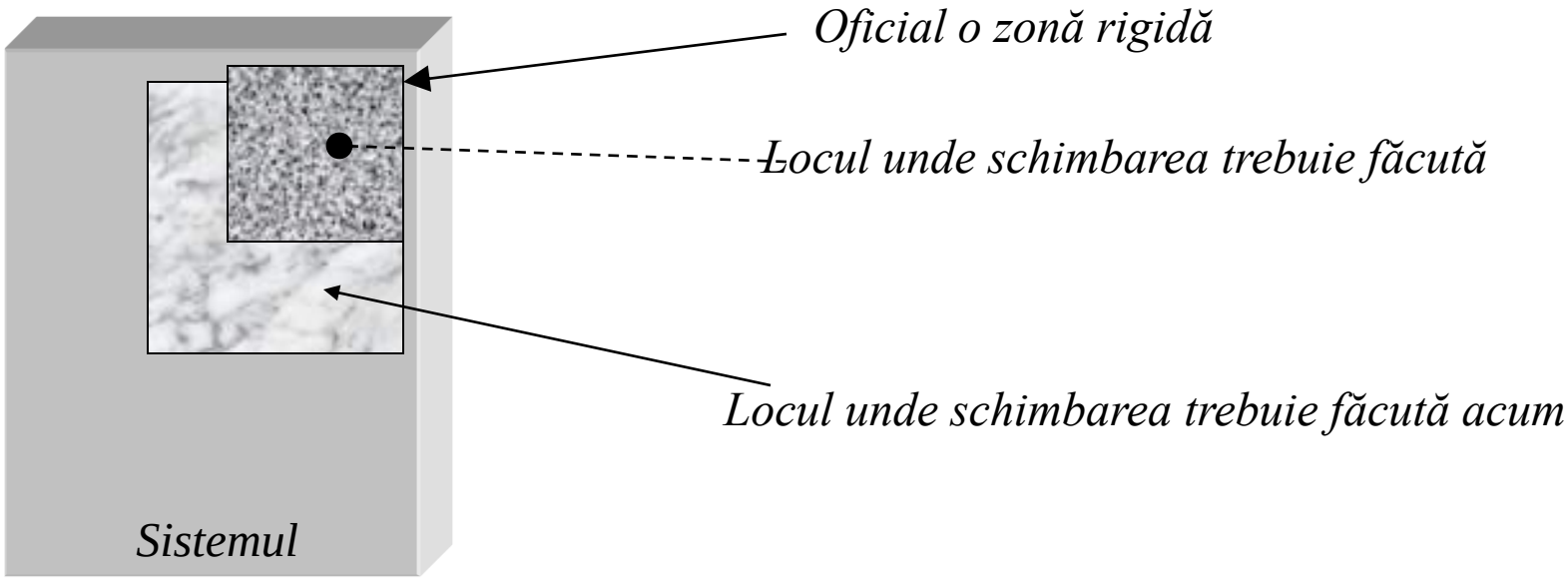
- **Rigiditatea**
- **Fragilitatea**
- **Reutilizarea limitată**
- **Vâscozitate** ridicată(*dificil de adăugat cod cu păstrarea design*-ului)

Rigiditatea

- Impactul unei modificări nu poate fi prognozat
- Pentru că nu poate fi prognozat nu poate fi estimat
- Timpul și costurile nu pot fi cuantificate
- Managerii devin reticenți în a autoriza schimbarea
- Rigiditatea se identifică cu prezența unor module de tip “Roach Motel” (ușor de intrat, greu de ieșit)

Rigiditatea reprezintă inabilitatea de a putea fi schimbat

Schimbări în condiții de rigiditate



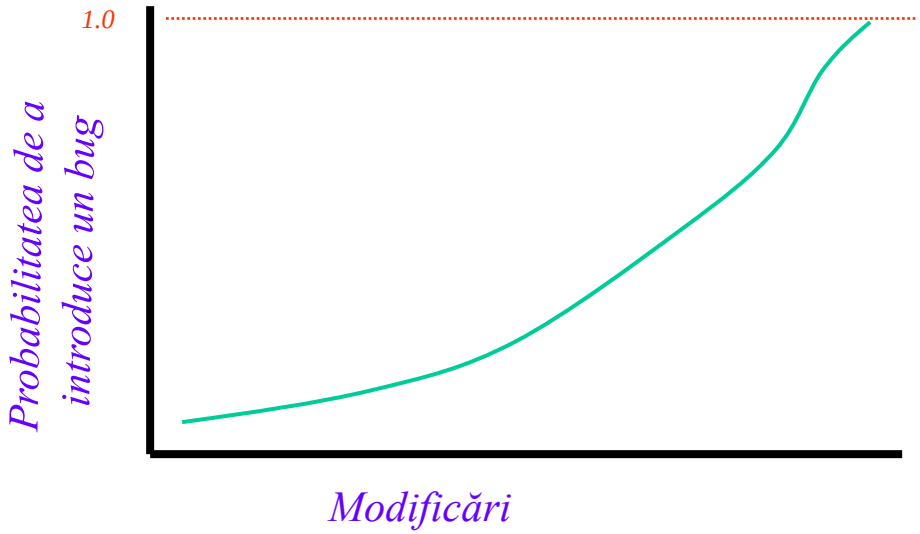
Există riscuri de răspândire a zonei ”maligne”

Fragilitatea

- O singură modificare generează o avalanșă de modificări ulterioare
- Noile erori apar în zone care nu par să fie legate de zonele modificate
- Calitatea nu poate fi predictibilă.
- Echipa de dezvoltare poate pierde credibilitatea
- Modificările software pot genera efecte non-locale

Creșterea Riscului

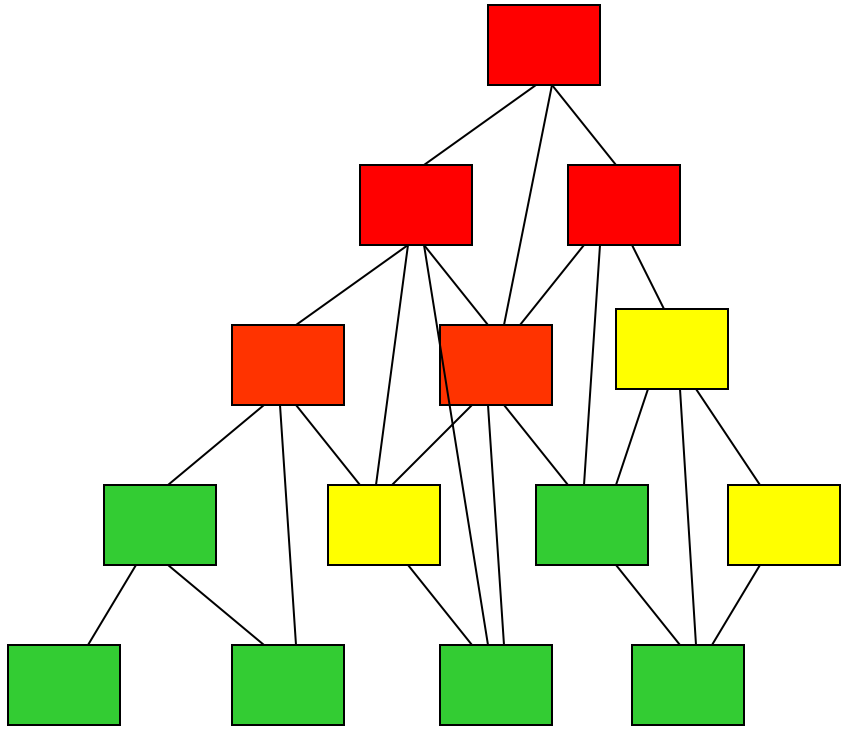
Defecte vs. Modificări cumulative



Sistemele tind să devină, în timp, din ce în ce mai fragile.
Rescrierea parțială, planificată poate fi necesară pentru a susține dezvoltarea și întreținerea sistemului.

Imposibilitatea reutilizării

- Componentele dorite ale proiectului sunt dependente de părțile nadorite.
- Munca și riscul de a extrage partea dorită pot genera depășirea costului reproiectării.



Vâscozitate ridicată

Viscozitatea este rezistența la curgere a unui fluid.

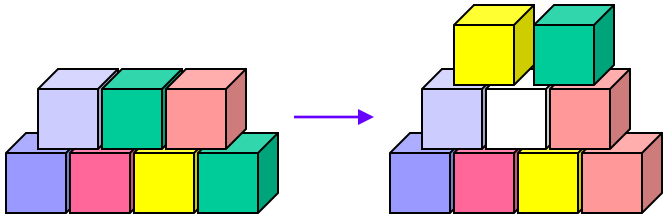
Când "schimbările corecte" sunt mult mai dificile decât hacking-ul, vâscozitatea sistemului este ridicată.

În timp, va deveni din ce în ce mai greu să continuăm dezvoltarea produsului software.

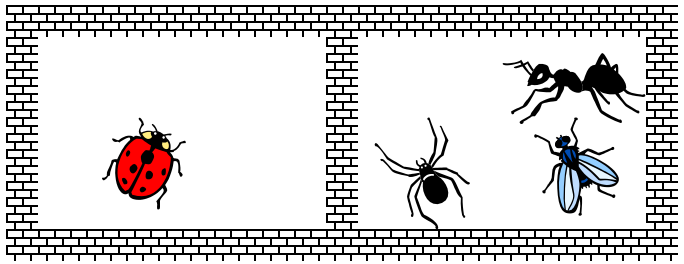
Beneficiile unui MD corect aplicat

Interdependențele sunt gestionate prin intermediul unor firewall-uri care să separe zonele care trebuie să varieze independent.

Flexibilitate crescută



Fragilitate redusă,iar bug-urile sunt izolate



Ușor de reutilizat

Ușor de făcut modificările potrivite

Metrici pentru Arhitectură, Analiză și Design

Analiza Codului

- Instrumentele software pentru ALM(**A**pplications **L**ifeCycle **M**anagement) nu includ metrici pentru activitățile ce țin de zona arhitecturii de sistem.
- Prin utilizarea metricilor asociate codului putem obține informații despre modul în care arhitectura și design-ul sistemului au fost concepute și interoperează:
  - Analiză statică a codului dezvoltat
  - Analiza dinamică a codului.

Analiza statica

- Analiza modului in care functioneaza codul dezvoltat fara a fi executat. Putem realiza acest obiectiv prin:
  - **Reflection** este o varianta de metaprogramare prin care se realizeaza "introspecția de tip", care permite(in contextul OOP) **identificarea tipului unui obiect la runtime**.

Ex.
```
/ Utilizam GetType pentru a obtine tipul variabilei:
int i = 1955;
Type type = i.GetType();
Console.WriteLine(type);
```

  - FxCop vs. .NET Analyzer
  - Profilling
    - CPU Sampling Report-raport relativ la utiliz. CPU
    - Instrumentarea codului cu LOC pentru a raporta timpii fiecarei metode. Monitorizarea timpilor mari de la niv. comp. Software
    - Memory profiler – timpii pina la actiunea garbage collector
    - Profilingul concurentei-urmărire threadurilor

Analiza dinamică a codului

- Analiza dinamică a programelor – Analiza software realizată prin execuția programelor pe un mediu de procesare real sau virtual.
- Pentru ca analiza dinamică să fie efectivă programul țintă al analizei trebuie executat cu suficiente input-uri de testare pentru a permite sinteza comportamentului computational al acestuia.
- Metrici în testarea produsului software:
  - ***Code Coverage – măsură utilizată a descrie gradul în care codul sursă al programului este testat cu o suită de teste particulare. Un program cu un "code coverige" mare are șanse mari de a include un număr mic de bug-uri.***
- **Detectează dependențele imposibil de identificat cu analiza statică. Pot colecta informații temporale.**

Metrici VS 2019-Complexitate ciclomatica

Complexitate Ciclomatica-Masoara complexitatea structurala a codului.

Numarul “drumurilor parcurse în procesul de execuție” al codului generate de instructiunile **if, instructiunile de ciclare(loop),** etc**.**

Un numar ciclomatic mare este un indicator pentru refactoring.

Valoarea recomandată : aprox. 10

Metrici pentru Cod
Adincimea mostenirii

- Indică numarul definițiilor de clase, care se extinde pornind de la rădacina ierarhiei de clase.
- Spre deosebire de mostenirea in sine care nu are un efect negativ, nivelul mostenirii poate face codul greu de inteles.
- Complexitatea ciclomatica ma conduce la o reducere a adâncimii
- Valoarea recomandată este una inferioară lui 6.

Metrici de Cod

Cuplarea și Coeziunea Claselor

Măsurile de cuplare la clase unice prin parametri, variabile locale, tipuri de return, apeluri de metode, instanțieri generice sau orientate șablon, clase de bază, implementări de interfață, câmpuri definite pe tipuri externe, și ”atribute cu rol decorativ”.

Coeziunea este măsura gradului în care o clasă are un rol bine determinat în cadrul sistemului.

Software-ul de bună calitate se caracterizează prin tipuri și metode de coeziune mare și cuplaj scăzut. Cuplajul indică un design dificil de reutilizat din cauza interdependențelor sale cu alte tipuri.

Indicat este a se realiza o valoare redusa a cuplarii claselor

Linii de cod

Indica numărul liniilor de cod executabile

Comentariile și spațiile sunt incluse.

Nu reprezinta o masura a progresului.

Indexul de mentenanță

O combinatie a metricilor:

- Complexitatea ciclomatica
- Liniile de cod
- Complexitatea computationala

$$Max(0, (171 - 5.2 * \ln(HV) - 0.23 * CC - 16.2 * \ln(LC)) * \frac{100}{171})$$

Unde HV este numarul drumurilor executabile intr-un proces

Performanță si Profiling

- Determinarea cauzelor care genereaza functionarea lenta sau ineficienta a aplicatiilor software.
- Analiză dinamica a codului(în opoziție cu cea statică)
- Visual Studio 2019 Ultimate include “**profiling tool**” integrate in IDE.

Tipuri de “profile-re”

- Majoritatea instrumentelor de profileing:
  - Esantionare(sampling)
    - “Snapshot-uri” periodice (esantioane) a aplicatiei executate, inregistrind liniile de cod executate.
  - Instrumentare(Instrumentation)
    - Tracing markers(probe) la inceputul si sfirsitul fiecarei functii-instrumentarea aplicatiei.
- Visual Studio –ofera tool-uri pentru esantionare si instrumentare.

Microsoft Profiling

- Sampling - Call Atributed Provider(CAP)
- Instrumentation-Low-Overhead Profiler(LOP)
- Nu a fost o simpla includere in VS. Au fost adaugate noi capabilitati.

Modul de utilizare al profiler-ului.

- Crearea unei sesiuni:
  - selectarea metodei:
    - CPU sampling
    - Instrumentare
    - Esantionarea memoriei
    - Concurenta
  - Target-ul
- Utilizarea “Performance Explorer” pentru a administra proprietatile sesiunii
- Lansarea sesiunii – executarea aplicatiei si sesiunii
- Raportarea

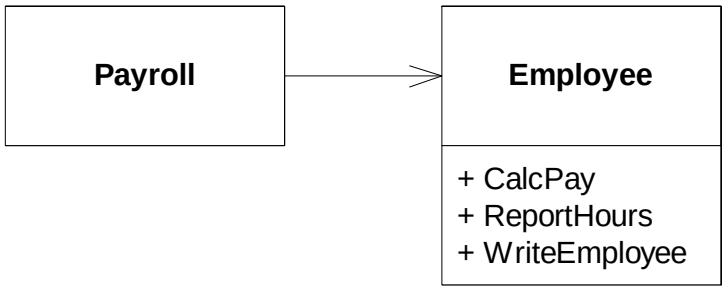
Principiile de proiectare a claselor

https://www.codeproject.com/Articles/703634/SOLID-architecture-principles-using-simple-Csharp

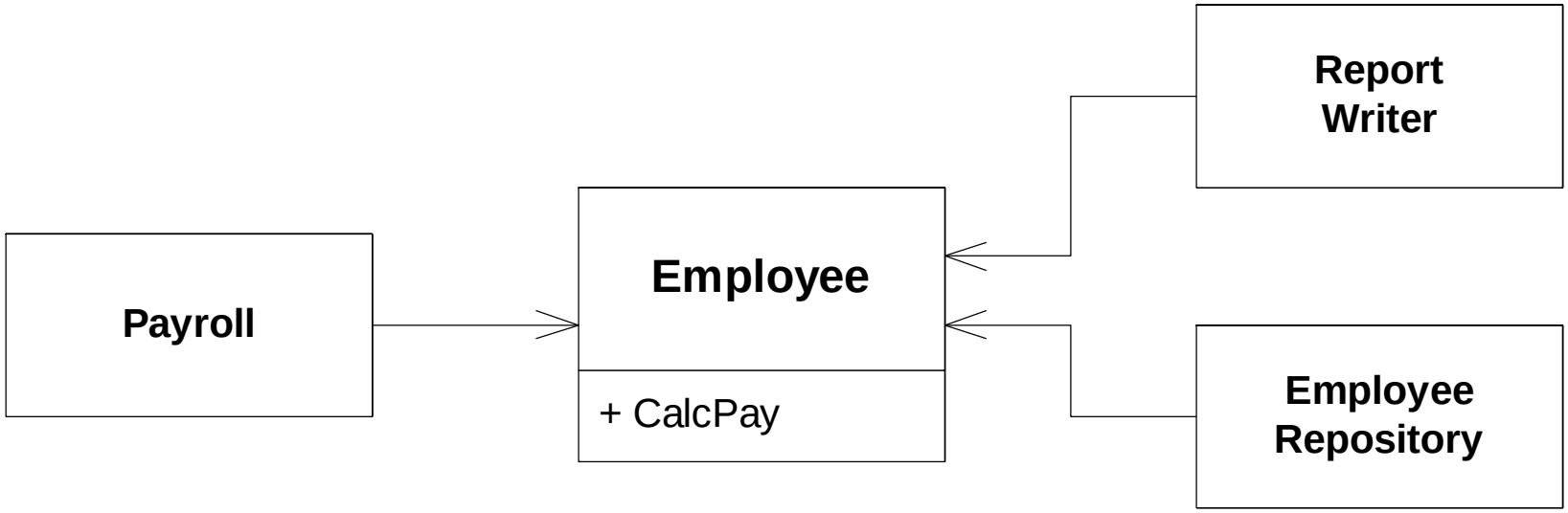
- SRP: Single Responsibility Principle
- OCP: Open/Closed Principle
- LSP: Liskov Substitution Principle
- ISP: Interface Segregation Principle
- DIP: Dependency Inversion Principle

Principiul Singurei Responsabilități

- Fiecare clasă/funcție trebuie să realizeze un singur task
- O clasă ar trebui să aibă un singur motiv de modificare(responsabilitate).



Principiul singurei responsabilități



Principiul Open/Closed

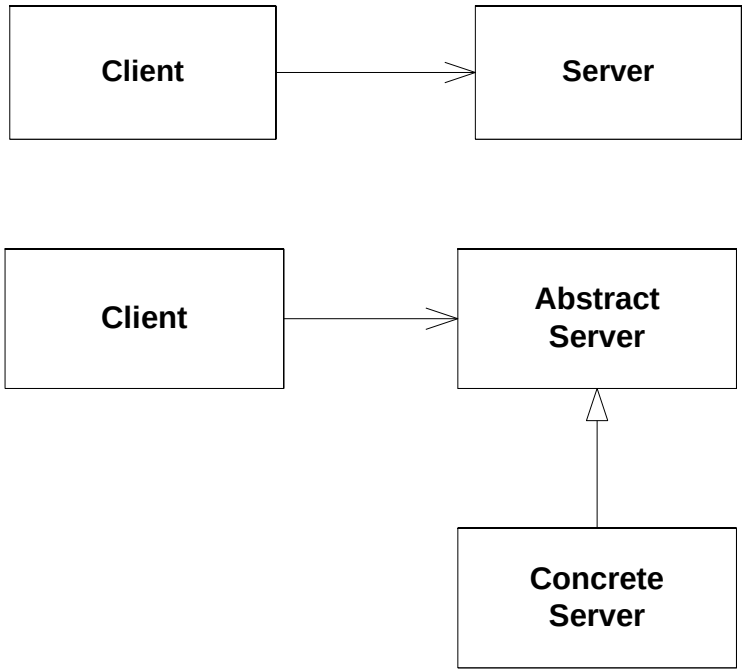
“Modulele trebuie să fie deschise pentru extensie, dar închise pentru modificare”
-Bertrand Meyer

- Un principiu care spune că ar trebui să adăugăm noi funcționalități prin adăugarea de cod nou, nu prin editarea codului vechi.
- Definește elementele care dau valoare adăgată programării OO
- Abstracția este cheia abordării

Abstracția este cheia

Abstracția este cel mai important cuvânt din OOD

- Relația de tip Client/Server este “open”
- Schimbările la nivel server generează schimbări la nivelul clienților
- Serverele abstracte generează o stare “close" clienților la schimbări în implementare.



Inchidere Strategică

Niciun program nu este 100% închis.

- Închiderea este strategică. Trebuie să alegem modificările pe care le vom izola.
- Formele individuale trebuie instanțiate.
- Este mai bine dacă putem limita dependențele.

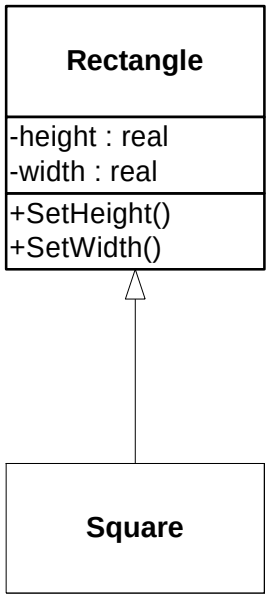
Principiul Substituției Liskov

Derived classes must be usable through the base class interface, without the need for the user to know the difference.

- Toate clasele derivate trebuie să fie substituibile pentru clasele de bază ale acestora
- Acest principiu ne ghidează în crearea abstractizărilor.

Pătrat/Dreptunghi

Un pătrat este un dreptunghi. Fie un Pătrat ca un Subtip de Dreptunghi..




```
void Square::SetWidth(double w)
{
   width = w;
   height = w;
}
void Square::SetHeight(double h)
{
   width = h;
    height = h;
}
```

Substituția… respinsă!

- Este rezonabil ca utilizatorii conceptului de dreptunghi să se aștepte ca lungimea și lățimea să se schimbe independent.
- Aceste așteptări sunt precondiții și postcondiții
- Bertrand Meyer o numește “Proiectare prin contract”
  - Contractul post condiție pentru dreptunghi este
    - lungime = new lungime
    - lățime= vecheaa lățime
- Pătratul violează contractile dreptunghiului

Substituția Liskov (cont.)

- Un client al dreptunghiului se așteaptă ca lungimea si lățimea să fie schimbate independent
  - void setAspectRatio(Rectange* r, double ratio);
- Prin derivarea Pătratului din dreptunghi, permiteți cuiva să stabilească raportul de aspect al unui Pătrat
- Putem obține
  - if (typeid(r) == typeid(Dreptunghi))
  - Încalcă principiul Open/Closed

Substitutia Liskov

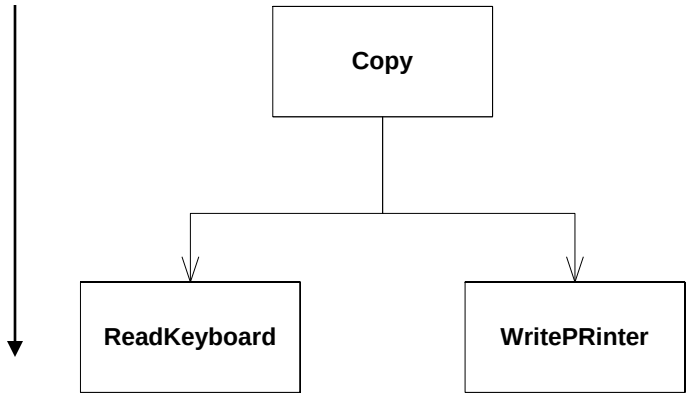
- Proiectarea prin Contract
  - Bertrand Meyer
  - Precondiții, postcondiții, invarianți
- Postcondițiile Dreptunghiului pentru setWidth()
  - width = newWidth
  - length = oldLength
- Pătratul nu poate necesita mai mulți clienți, nici nu poate promite mai puțini
  - Nu menține invariantă lungimea
  - Violază contractul

LSP Ghidează Crearea Abstracțiilor

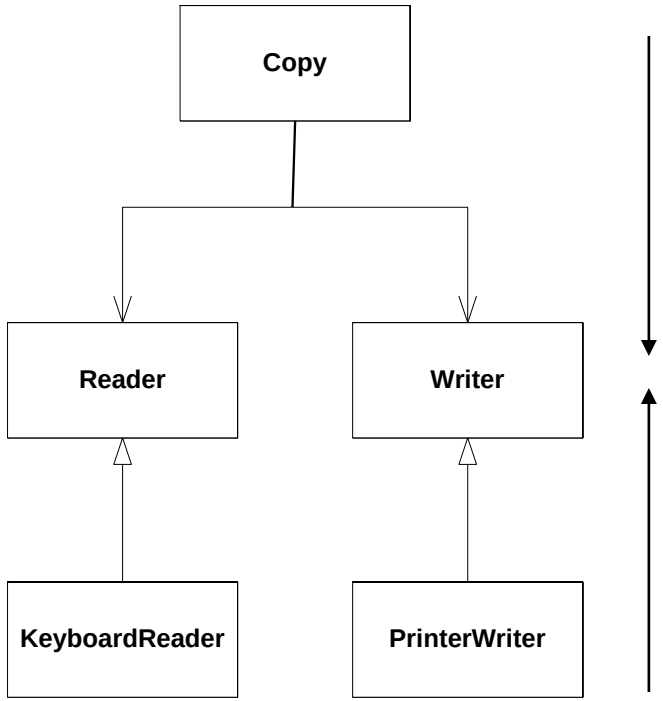
- Abstracțiile nu au o existentă izolată
- Abstracțiile nu se potrivesc întotdeauna cu așteptările lumii reale
- Violarea LSP este echivalent cu încălcarea OCP

Principiul Dependenței Inverse

Detaliile ar trebui să depindă de abstracții.
Abstracțiile nu ar trebui să depindă de detalii.



V.



Implicații DIP

Totul ar trebui să depindă de abstractizări

- De evitat derivarea de la clasele concrete
- De evita asocierea cu clasele concrete
- De evitat agregarea claselor concrete
- De evitat dependențele de componentele concrete

Dependency Inversion Principle (cont.)

- Motive de încălcare a Principiului Dependenței Inverse
  - Crearea de Obiecte
    - new Cerc crează creates o dependență de o clasă concretă
    - Localizarea dependențelor utilizând ”factories„(biblioteci, de preluare in afara ierarhiei)
  - Clase Nonvolatile
    - string, vector, etc.
      - Cu condiția să fie stabile

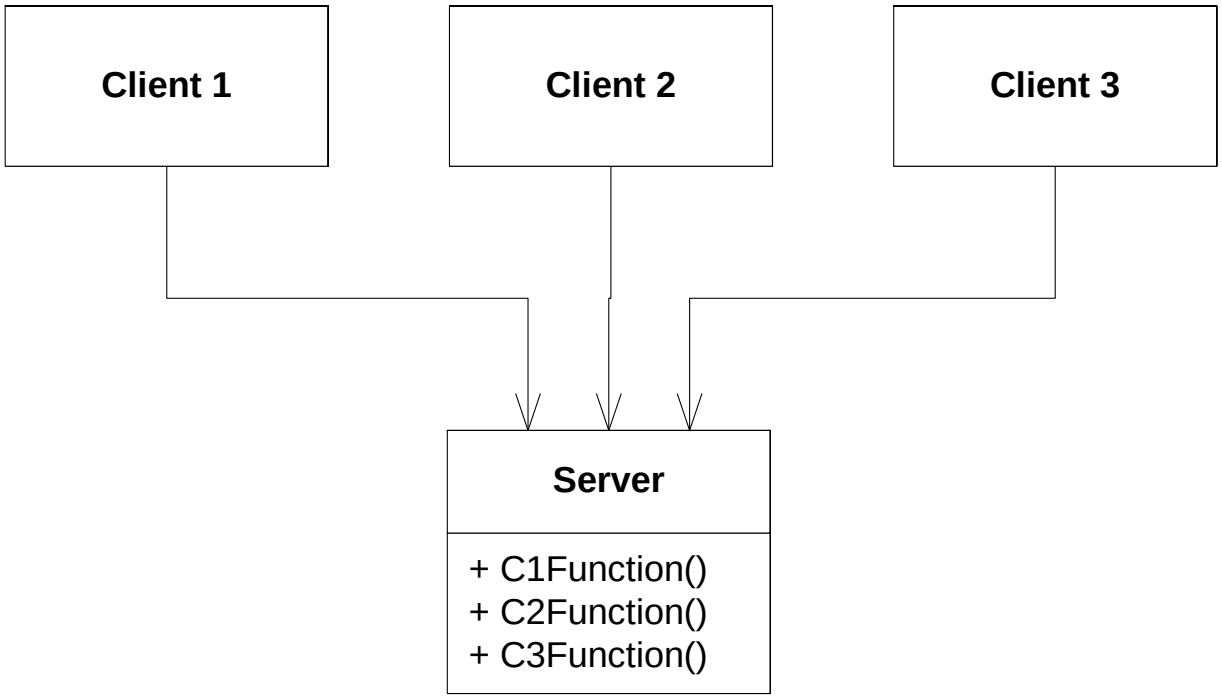
Principiul Segregării Interfețelor

Ajută la abordarea interfețelor "fat" sau necorespunzătoare

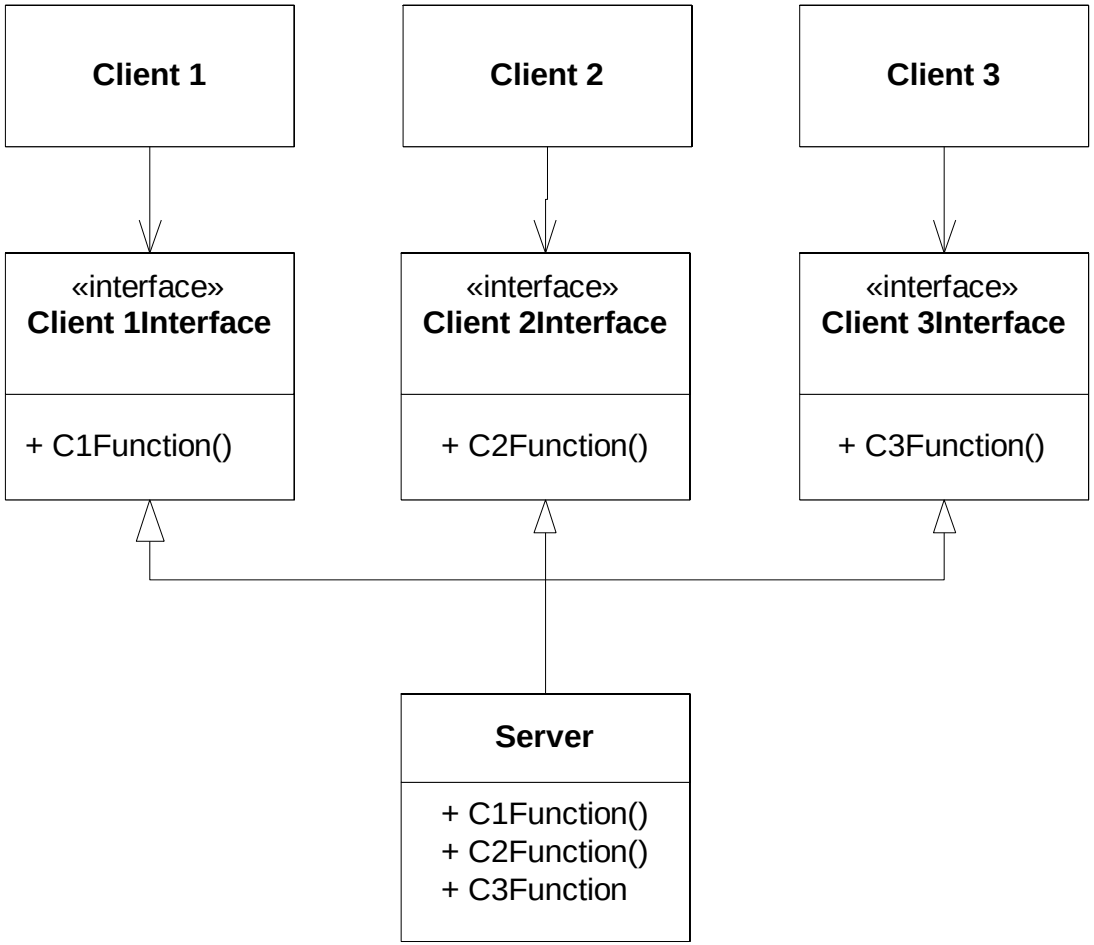
- Uneori metodele claselor includ diferite grupări.
- Aceste clase sunt folosite în scopuri diferite.
- Nu toți utilizatorii folosesc toate metodele.
- Această lipsă de coeziune poate cauza probleme grave de dependență
- Aceste probleme pot fi supuse reproiectării.

Poluarea Interfeței prin “collection”

Clienți diferiți ai clasei noastre au nevoi distincte legate de interfață.



Un exemplu de segregare



Concluzii

- OCP Extinde funcția fără editarea codului
- LSP Instanțele ”copil” înlocuiesc pe cele de bază
- DIP Depența se realizează prin abstracții în loc de detalii
- ISP Segregarea interfețelor pentru un management corect al dependențelor