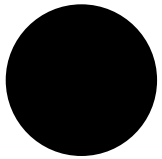



# Principii SOLID

**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”. Un 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 cu controlul interdependențelor.

# Legătura dintre MD și procesul dezvoltării de software

Cuplarea și coeziunea(măsura în care o clasa implementază o anumită sarcină specifică) sunt permanente provocări pentru dezvoltatorii de aplicații

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

Consecințele unei practici defectuase a MD

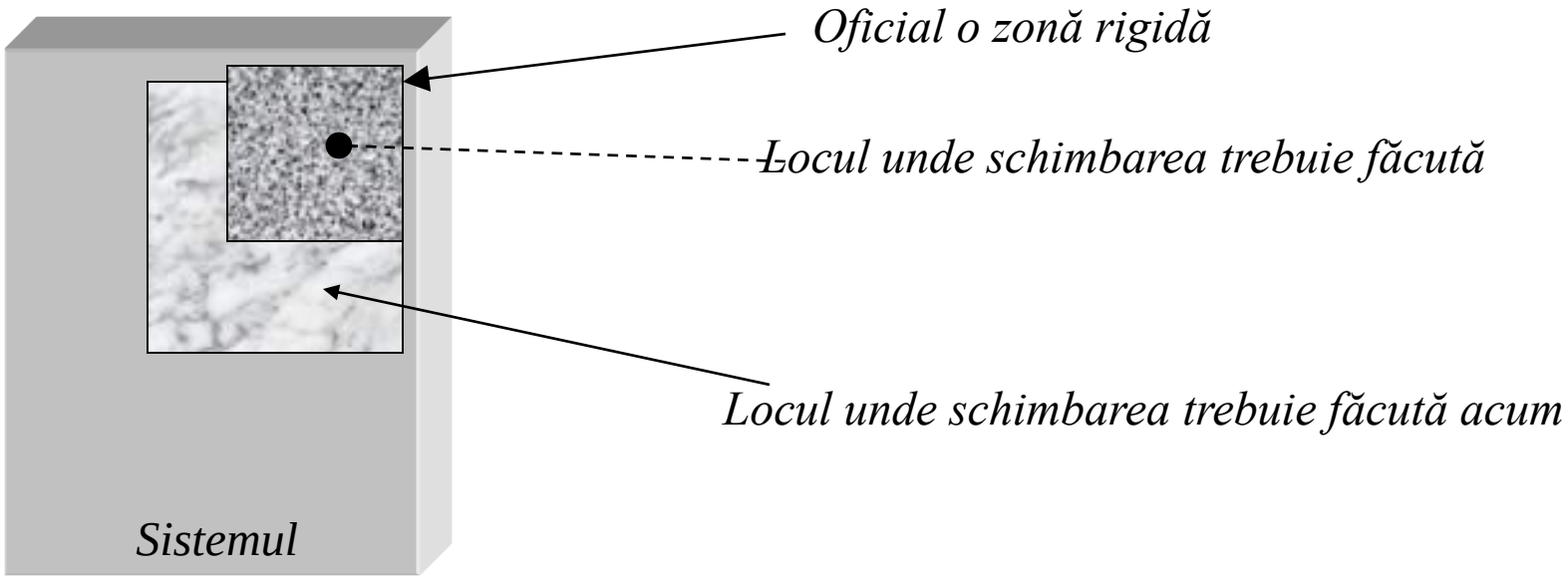
- Rigiditatea
- Fragilitatea
- Reutilizarea limitată
- Vâscozitate ridicată(dificil de adăugat cod cu păstrarea design-ului)

# 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 este inabilitatea de a putea fi schimbat*

# Schimbări în condiții de rigiditate




***Există riscuri de răspândire a zonei ”maligne”***

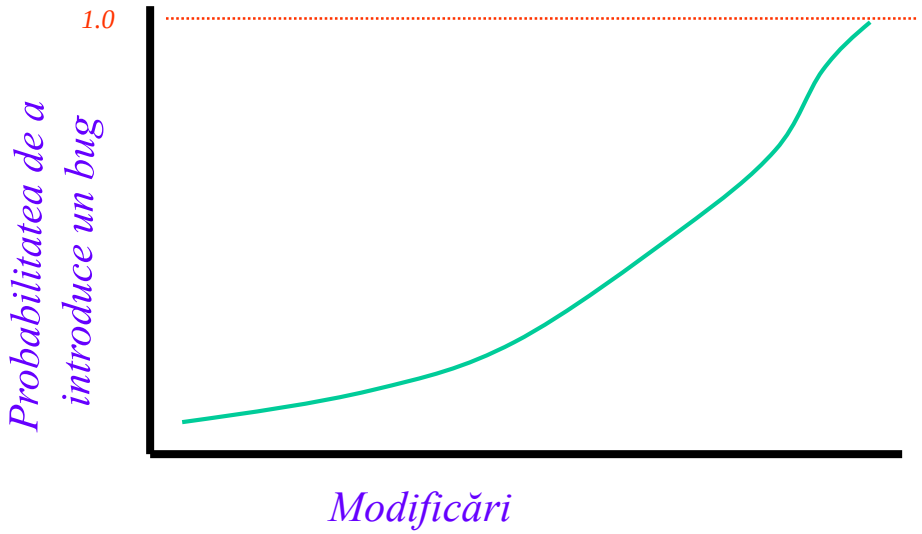
# Fragilitatea

- 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

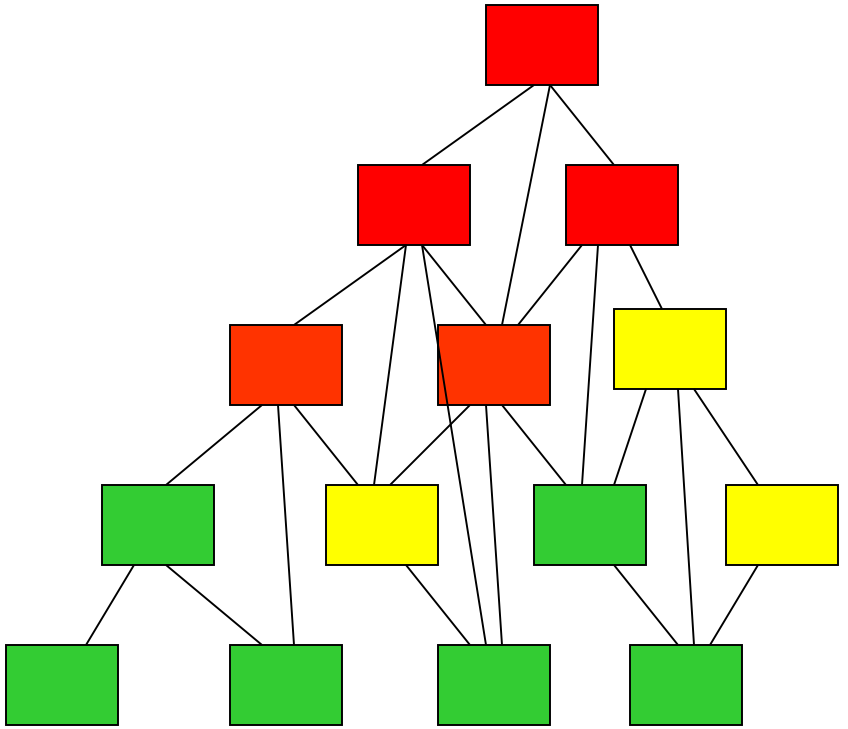
*Defecte vs. Modificări cumulative*




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 depășire costului reproiectării.



# Vâscozitate ridicată

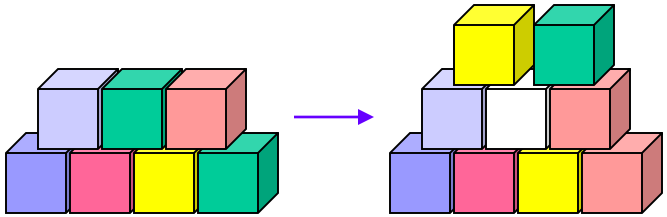
**Viscozitatea este rezistența la curgere a unui fluid**.

- 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.

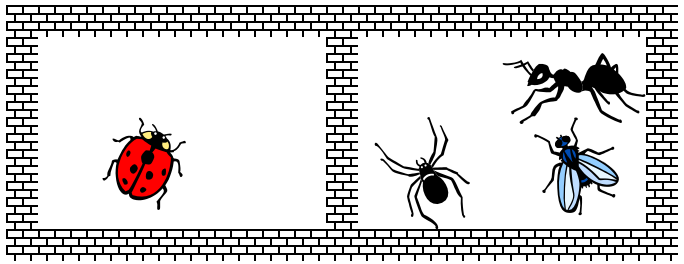
# Beneficiile unui MD corect aplicat

*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*

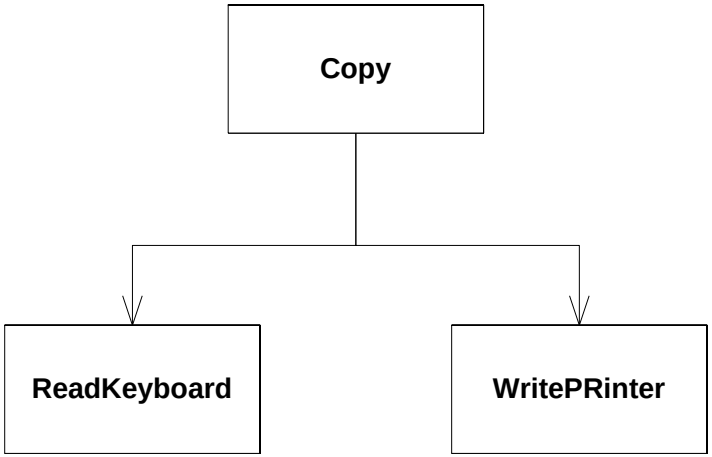


*Ușor de reutilizat*

*Ușor de făcut modificările potrivite*

# Prima Versiune

*Toți designerii încep bine*




```
void copy(void)
{
     int ch;
     while( (ch=ReadKeyboard()) != EOF)
          WritePrinter(ch);
}
```

*Programul este un succes peste noapte!*

*Cum ar putea fi mai simplu, mai elegant și mai ușor de întreținut?*

# Versiunea a Doua

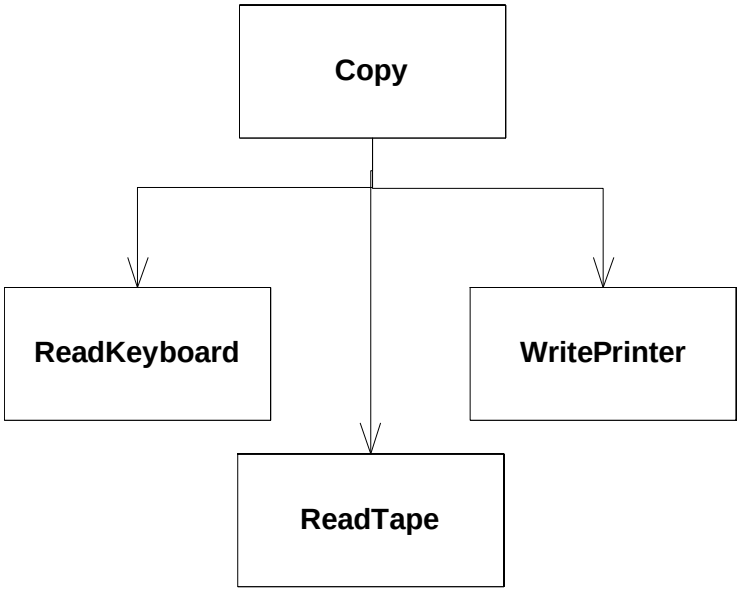
*Nimeni nu se gândește că cerințele ar trebui modificate!*

- Vrem uneori să citim de la cititorul de bandă perforată.
- Am putea pune un parametru în apel, dar deja avem sute de utilizatori!
- Nu este mare lucru, este doar o excepție ... putem face să funcționeze.

# A doua versiune a codului

```
bool GtapeReader = false; // nu uita sa clarifici

void copy(void)
{
    int ch;
    while( (ch=GtapeReader ? ReadTape() : ReadKeyboard()) != EOF)
       WritePrinter(ch);
}
```



# A treia versiune

*Cerințele sunt modificate din nou!*

Se pare că uneori trebuie să scriem pe o bandă perforată. Am avut această problemă înainte și tocmai am adăugat un flag. Se pare că ceea ce am făcut ar trebui să funcționeze din nou.

```
bool GtapeReader = false;
Bool GtapePunch = false;
// remember to clear

void copy(void)
{
    int ch;
    while( (ch=GtapeReader ? ReadTape() : ReadKeyboard()) != EOF)
       GtapePunch ? WritePunch(ch) : WritePrinter(ch);
}
```

# Este OK

*Prima și ultima versiune.*

```
void Copy()
{
   int c;
   while( (c=getchar()) != EOF)
      putchar(c);
}
```

*Nu ar trebui să aplicăm programarea OO-ceea ce am făcut nu este OO*

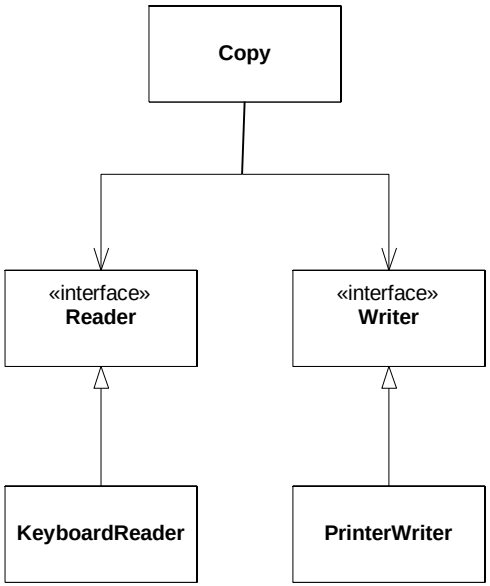
# *...este?*

*Este un mic program bazat pe o abstracție!*

- FILE este o abstracție
  - Ea reprezintă un anumit tip de flux de octeți
  - Are multe variante
- Are associate metode
  - Read, Write, getchar, putchar, etc
  - Metodele sunt legate *dinamic*

FILE este o clasă, implementată diferit.

# Reformulat în OO




```
interface Reader
{ char read(); }

interface Writer
{ void write(char c); }

public class Copy
{
    Copy(Reader r, Writer w)
    {
        itsReader = r;
        itsWriter = w;
    }
    public void copy()
    {
        int c;
        while( (c==itsReader.read()) != EOF )
            itsWriter.write(c);
    }
    private Reader itsReader;
    private Writer  itsWriter;
}
```

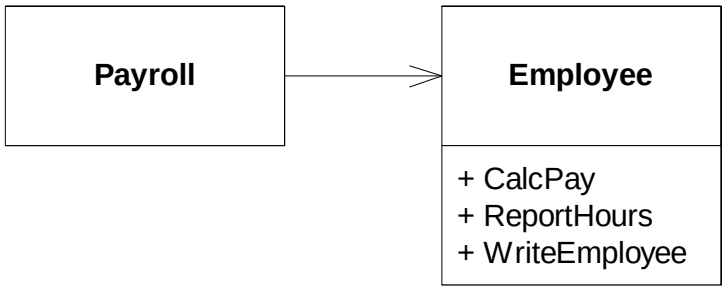
# Principiile de proiectare a claselor

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

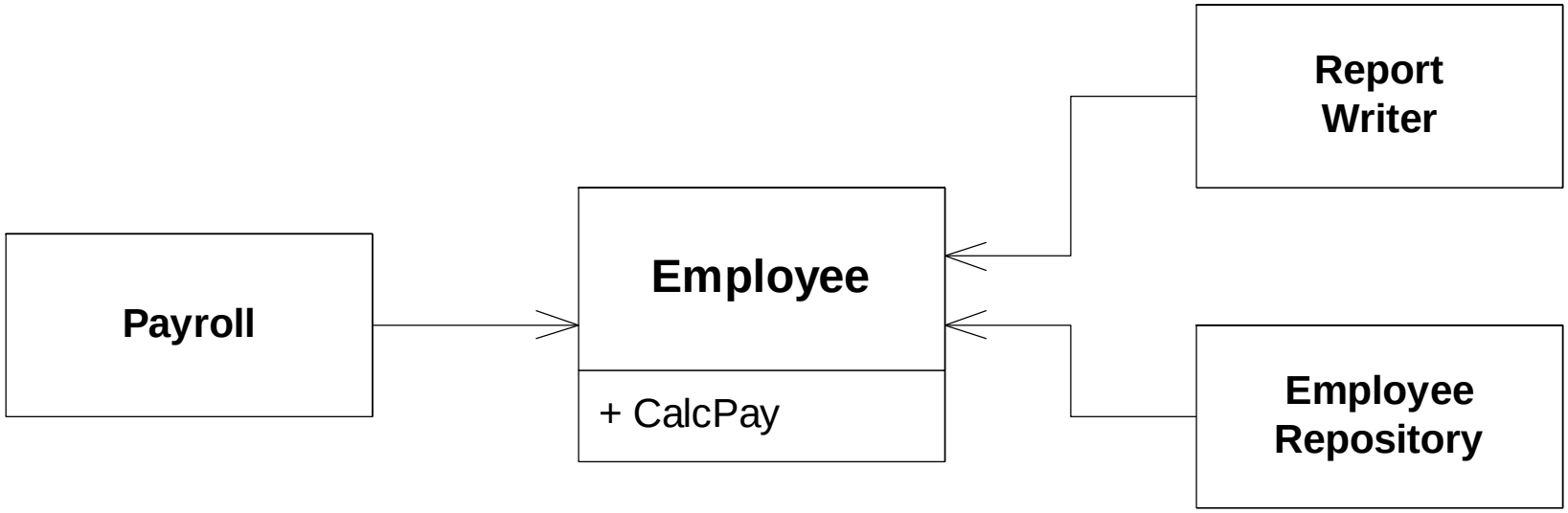
- SRP: Single Responsibility Principle
- OCP: Open/Closed Principle
- LSP: Liskov Substitution Principle
- ISP: Interface Segregation Principle
- DIP: Dependency Inversion Principle

# 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

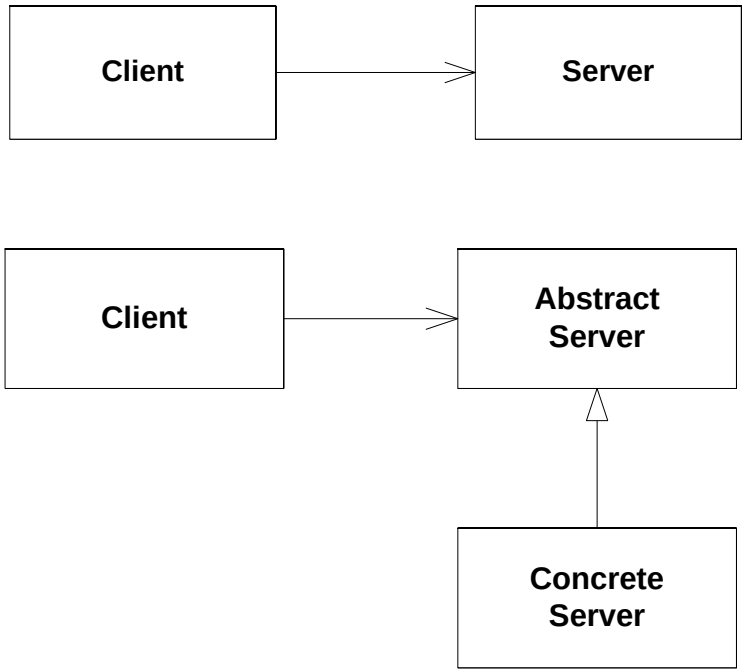
*“Modulele trebuie să fie deschise pentru extensie, dar închise pentru modificare”*
*-Bertrand Meyer*

- 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.



# Versiunea Procedurală (open)

***Shape.h***

```
enum ShapeType {circle, square};
struct Shape
     {enum ShapeType itsType;};
```

***Circle.h***

```
 struct Circle
 {
    enum ShapeType itsType;
    double itsRadius;
    Point itsCenter;
 };
 void DrawCircle(struct Circle*)
```

***Square.h***

```
struct Square
{
   enum ShapeType itsType;
   double itsSide;
   Point itsTopLeft;
};
void DrawSquare(struct Square*)
```

***DrawAllShapes.c***

```
#include <Shape.h>
#include <Circle.h>
#include <Square.h>

typedef struct Shape* ShapePtr;

void
DrawAllShapes(ShapePtr list[], int n)
{
   int i;
   for( i=0; i< n, i++ )
   {
       ShapePtr s = list[i];
       switch ( s->itsType )
       {
       case square:
            DrawSquare((struct Square*)s);
            break;
       case circle:
            DrawCircle((struct Circle*)s);
            break;
       }
   }
}
```

# Aspecte negative ale codului

*Se poate demonstra că funcționează. Nu este asta un lucru important ?*

- DrawAllShapes nu este ”close”.
  - *Switch/case* tinde să apară recurent.
  - Daca adaugăm o formă, adaugam un switch/case
  - Toate instrucțiunile *switch/case* trebuie identificate și editate.
  - *Instrucțiunile Switch/Case* sunt rareori ordonate
  - Când adăugăm la *enum*, trebuie reconstruit totul
- Software-ul este atât rigid cât și fragil

# O implementare ”close”

***Shape.h***

```
Class Shape
{
public:
   virtual void Draw() const =0;
};
```

***Square.h***

```
Class Square: public Shape
{
public:
   virtual void Draw() const;
};
```

***Circle.h***

```
Class Circle: public Shape
{
public:
   virtual void Draw() const;
};
```

***DrawAllShapes.cpp***

```
#include <Shape.h>

void
DrawAllShapes(Shape* list[],int n)
{
   for(int i=0; i< n; i++)
       list[i]->draw();
}
```

# 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.

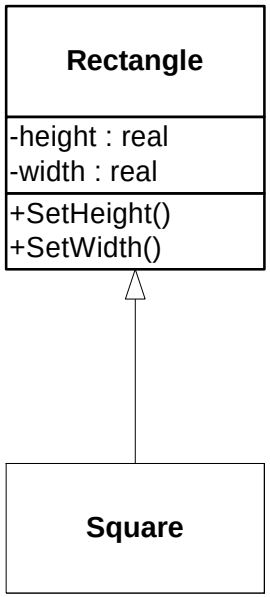
# Principiul Substituției Liskov

*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

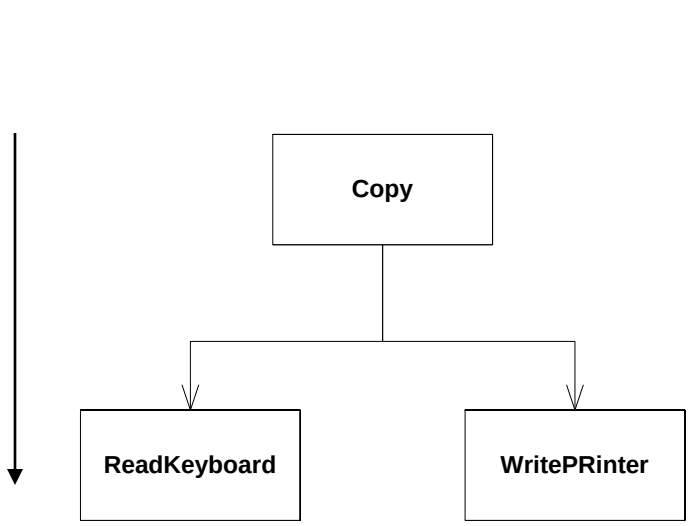
- Proiectarea prin Contract
  - Bertrand Meyer
  - Precondiții, postcondiții, invarianți
- Postcondițiile Dreptunghiului pentru setWidth()
  - width = newWidth
  - length = oldLength
- Pătratul nu poate necesita mai mulți clienți, nici nu poate promite mai puțini
  - Nu menține invariantă lungimea
 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

# LSP Ghidează Crearea Abstracțiilor

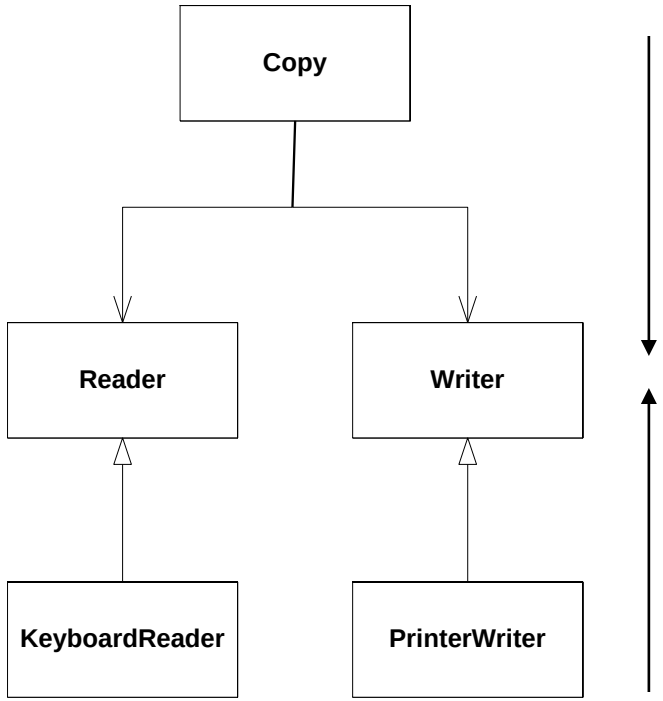
- Abstracțiile nu au o existentă izolată
- Abstracțiile nu se potrivesc întotdeauna cu așteptările lumii reale
- Violarea LSP este echivalent cu încălcarea OCP

# 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

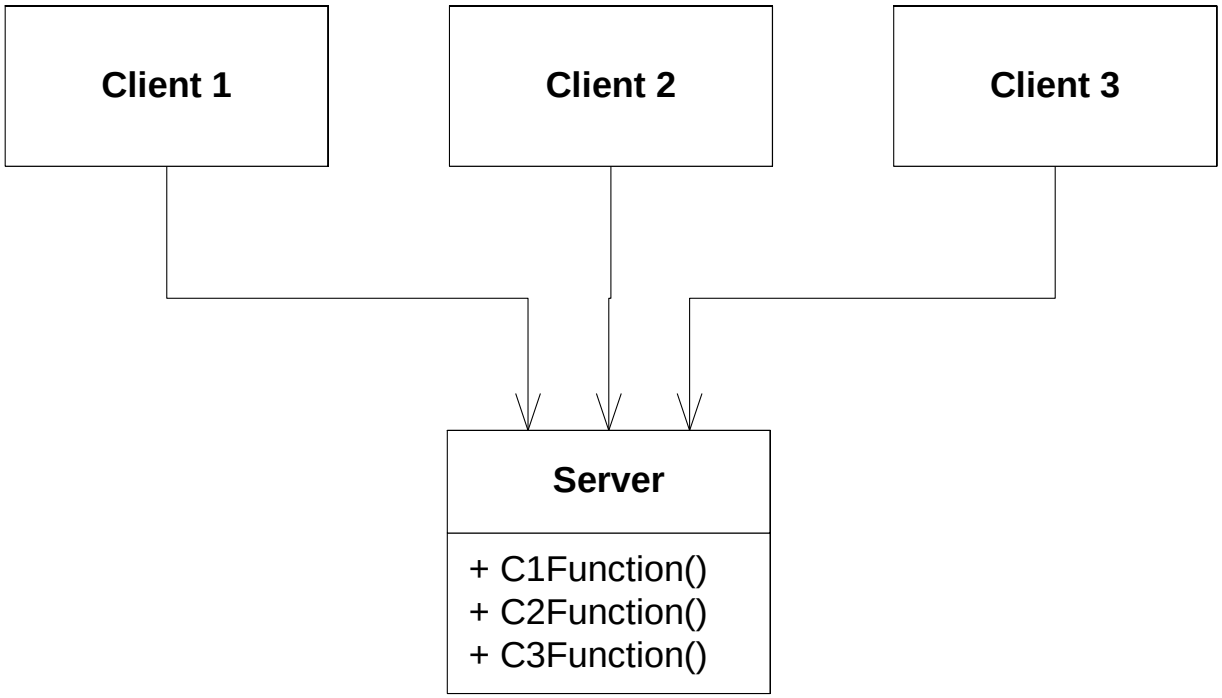
# Principiul Segregării Interfețelor

*Ajută la abordarea interfețelor "fat" sau necorespunzătoare*

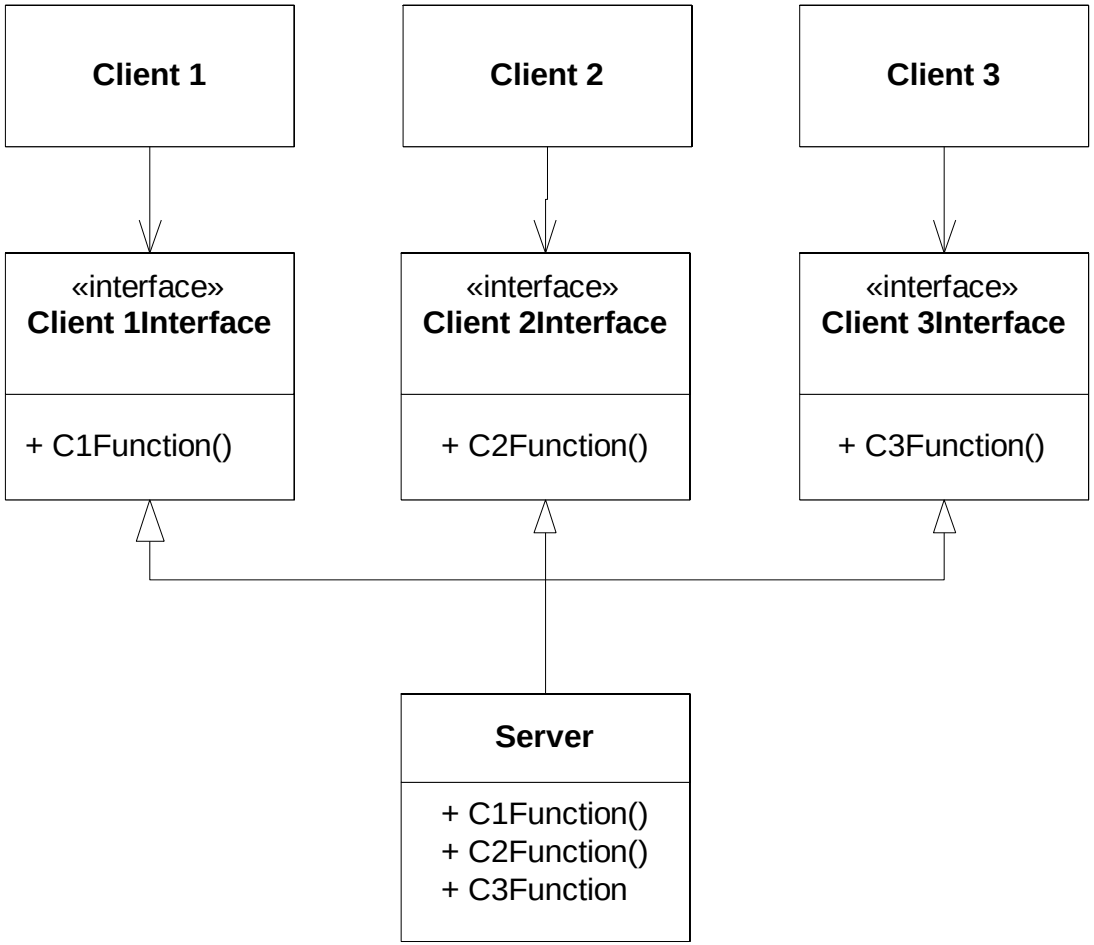
- Uneori metodele claselor includ diferite grupări.
- Aceste clase sunt folosite în scopuri diferite.
- Nu toți utilizatorii folosesc toate metodele.
- Această lipsă de coeziune poate cauza probleme grave de dependență
- Aceste probleme pot fi supuse reproiectării.

# 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