

Programare II

Limbajul C/C++

CURS 14



CUPRINS

SOLID

GRASP

Principii OO

❑ Definiție

❑ Un principiu de proiectare este un principiu sau o soluție de bază care poate fi aplicat pentru proiectarea sau scrierea de cod mai **ușor de întreținut, flexibil și extensibil**

❑ Principiile OO – SOLID

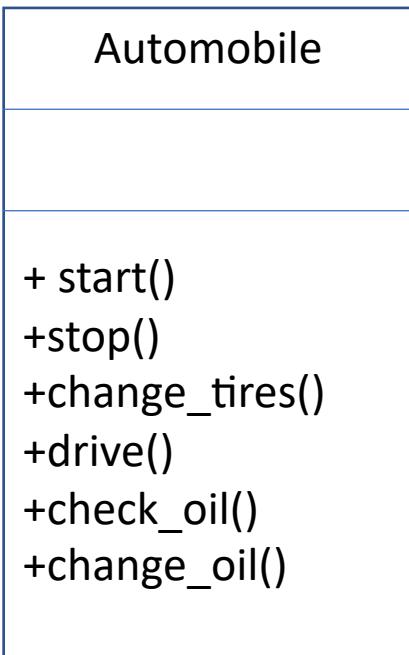
- ❑ **SRP** - Single-responsibility principle
- ❑ **OCP** - Open-closed principle
- ❑ **LSP** - Liskov substitution principle
- ❑ **ISG** - Interface segregation principle
- ❑ **DRY** - Dependency Inversion Principle

SRP- Single Responsibility Principle

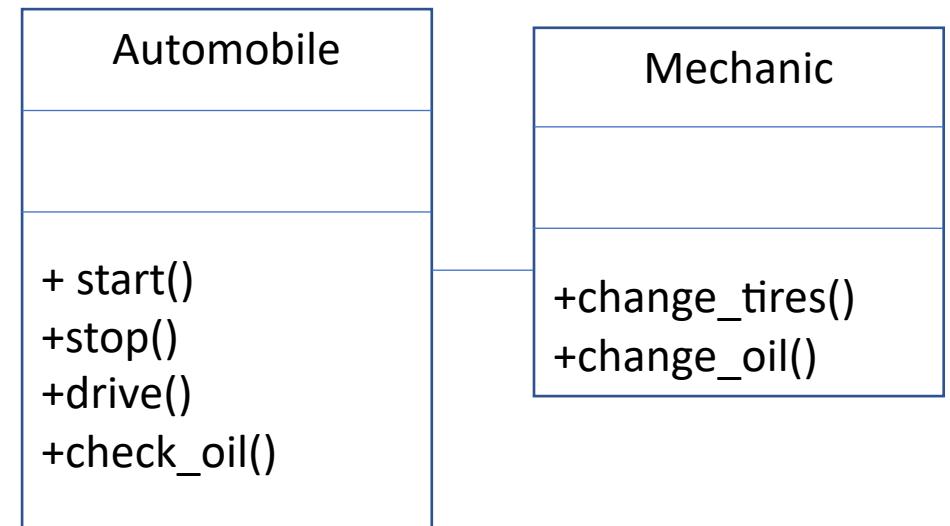
❑ SRP

- ❑ fiecare obiect ar trebui să aibă o **singură responsabilitate**, și toate serviciile ar trebui să se focuseze spre a retransmite acea responsabilitate
- ❑ Doar un singur motiv pentru a modifica ceva
- ❑ Codul este mai simplu și ușor de întreținut
- ❑ Exemplu
 - ❑ containerele și iteratorii (containerele gestionează obiecte, iteratorii traversează containările)
 - ❑ Cum să nu suportăm responsabilități multiple? Formând propoziții care se termină cu cuvântul **itself**

SRP- Single Responsibility Principle



The Automobile can start itself.
The Automobile can stop itself.
The Automobile can change tires itself.
The Automobile can drive itself.
The Automobile can check oil itself.
The Automobile can change oil itself.



OCP- Open-close Principle

- ❑ OCP
 - ❑ clasele ar trebui să fie deschise pentru extensii dar închise pentru modificare
 - ❑ Permite schimbări fără a modifica codul existent
 - ❑ Folosirea moștenirii pentru a extinde/schimba codul funcțional existent și a nu se atinge de codul care funcționează
 - ❑ Poate fi implementat și prin intermediul compoziției

OCP- Open-close Principle

```
class Shape {  
    int type;  
    void drawPolygon () { /*... */ }  
    void drawPoint () { /* ... */ }  
public:  
    void draw();  
};  
void Shape::draw() {  
    switch(type) {  
        case POLYGON: drawPolygon ();  
                      break;  
        case POINT: drawPoint ();  
                      break;  
    } }
```

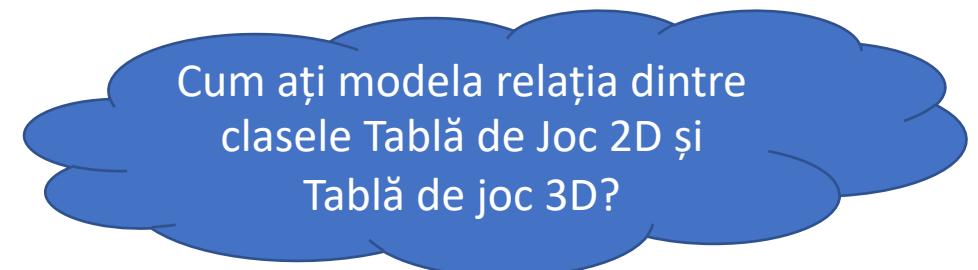
Ce se întâmplă dacă adaugăm un nou tip de figură?



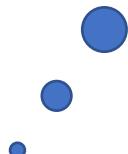
```
class Shape {  
public:  
    virtual void draw() = 0;  
};  
class Polygon : public Shape {  
public:  
    void draw();  
};  
class Point : public Shape {  
public:  
    void draw();  
};  
void Polygon::draw() { /* ... */ }  
void Point::draw() { /* ... */ }
```

LSP- Liskov Substitution Principle

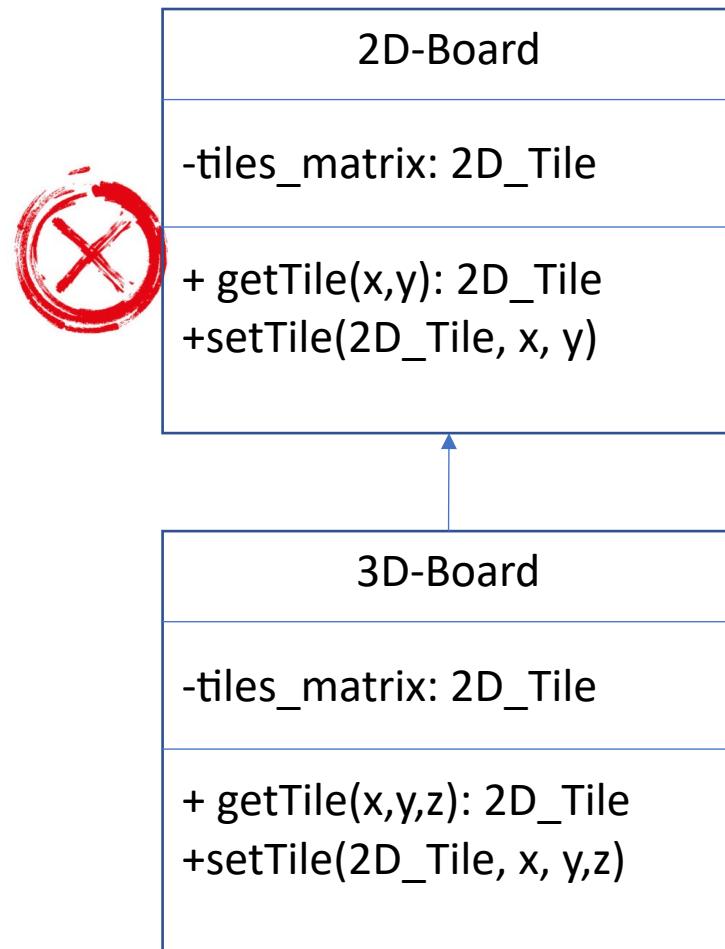
- ❑ LSP
 - ❑ Subtipuri trebuie să fie potrivite pentru clasele lor de bază
 - ❑ Ierarhii de clase bine proiectate
 - ❑ Subclasele trebuie să fie potrivite pentru clasele de bază fără a ne gândi că ceva nu este bine



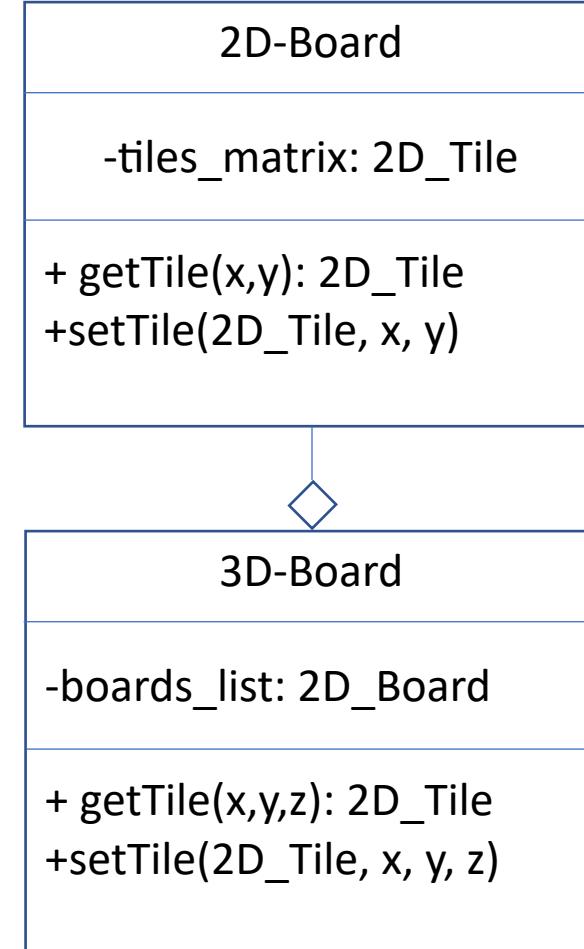
Cum ați modelat relația dintre clasele Tablă de Joc 2D și Tablă de joc 3D?



LSP- Liskov Substitution Principle



```
board = 3D-Board()  
board.getTile(4,5) // does not make sense of 3D board
```

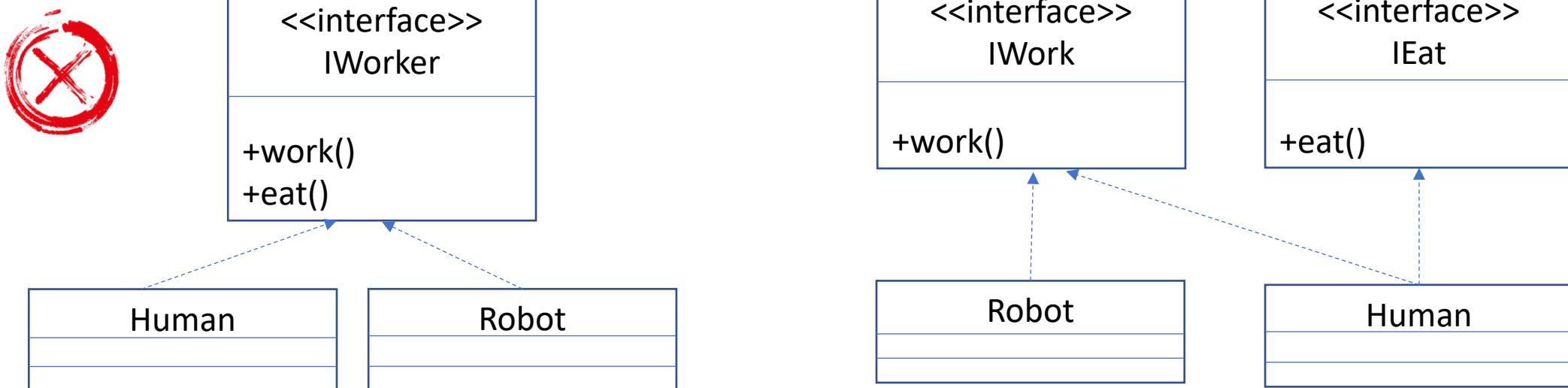


```
board = 3D-Board()  
board.getTile(1,4,5)
```

ISG- Interface Segregation Principle

❑ISP

- ❑ Un client nu ar trebui să fie obligat să implementeze o interfață pe care nu o folosește sau un client nu ar trebui să depindă de metode pe care nu le folosește.



ISG- Interface Segregation Principle



```
class ShapeInterface {  
public:  
    virtual double area() = 0;  
    virtual double volume() = 0;  
};  
  
class Square: ShapeInterface {  
public:  
    double area() { /*calcul arie */}  
    double volume() {/*nu are sens*/}  
};  
  
class Cuboid: ShapeInterface {  
public:  
    double area() { /*calcul arie cub*/}  
    double volume() {/*calcul volum cub*/}  
};
```



```
class ShapeInterface {  
public:  
    virtual double area() = 0;  
};  
  
class SolidShapeInterface {  
public:  
    virtual double volume() = 0;  
};  
  
class Square: ShapeInterface {  
public:  
    double area() { /*calcul arie */}  
};  
  
class Cuboid: ShapeInterface, SolidShapeInterface {  
public:  
    double area() { /*calcul arie cub*/}  
    double volume() {/*calcul volum cub*/}  
};
```

DRY- Dependency Inversion Principle

- ❑ Modulele la nivel înalt nu ar trebui să depindă de modulele de nivel scăzut.
 - ❑ Ambele ar trebui să depindă de abstractizări
- ❑ Abstractizările nu ar trebui să depindă de detalii.
 - ❑ Detaliile ar trebui să depindă de abstractizări
- ❑ Detaliile ar trebui să depindă de modul de funcționare.
 - ❑ Trebuie definite caracteristici generale și o referință spre abstractizarea pe care o implementează detaliul

DRY- Dependency Inversion Principle

```
class Worker{  
public:  
    virtual void work(){ cout << "... working"; }  
};  
  
class Manager{  
    Worker worker;  
public:  
    Manager( Worker w): worker(w) {}  
    void manage(){ this->worker.work(); }  
};  
  
class SuperWorker{  
public:  
    void work(){ cout << "... working much more"; }  
};
```

Poate Managerul să
gestioneze un muncitor
care este SuperWorker?

```
class IWorker{  
Public:  
    virtual void work(self) = 0;  
};  
  
class Worker: public IWorker{  
    virtual void work(){ cout << "... working"; }  
};  
  
class SuperWorker : public IWorker{  
    void work(){ cout << "... working much more "};  
};  
  
class Manager{  
    IWorker worker;  
public:  
    Manager( IWorker w): worker(w) {}  
    void manage(){ this->worker.work(); }  
};
```



GRASP

- ❑ GRASP
 - ❑ General
 - ❑ Responsibilities
 - ❑ Assignment
 - ❑ Software
 - ❑ Patterns (Principles)
- ❑ Descriu principiile fundamentale ale proiectării aplicațiilor orientate obiect și identificarea responsabilităților obiectelor

GRASP Patterns

- ❑ Pattern (șablon de proiectare)
 - ❑ o pereche **problemă/soluție** numită și **binecunoscută** care poate fi aplicată în contexte noi, cu indicații despre cum să fie aplicată în situații noi și discuții despre compromisuri, implementări, variații, etc.
- ❑ Un pattern este caracterizat prin
 - ❑ Un **nume**
 - ❑ O **problemă** pe care încercă să o rezolve
 - ❑ O **soluție**

Sabloane în inginerie

- ❑ Cum identifică și folosesc inginerii modele?
 - ❑ Inginerii utilizează documentații (**handbooks**) care descriu soluții la problemele cunoscute
 - ❑ Proiectanți de automobile nu proiectează mașini de **la zero** folosind legile fizicii, **refolosesc** modele standard cu rezultate de succes, învățând din experiență
- ❑ Ar trebui ca dezvoltatorii de software să folosească modele? De ce?
 - ❑ Dezvoltarea de software de la zero este costisitoare
 - ❑ Sabloane permit **reutilizarea** designului arhitecturii software

GRASP Patterns

1. **Information Expert**
 - atribuie responsabilitățile unei clase
2. **Creator**
 - Cunoaște detaliile de creare
3. **Low Coupling**
 - reducerea conectivității
4. **Controller**
 - Cazuri de utilizare sau clase de sistem
5. **High Cohesion**
 - Relațiile/responsabilități dintre/între obiecte
6. **Polymorphism**
 - Comportamentul depinde de tip
7. **Pure Fabrication**
 - Clase de bază
8. **Indirection**
 - Evitarea cuplării directe cu obiecte intermediare
9. **Protected Variations**
 - information hiding (ascunderea informației) - open/close

Information Expert Pattern

❑ Problema

- ❑ Care este un principiu general de atribuire a responsabilităților / funcționalității obiectelor?

❑ Soluția

- ❑ Atribuirea unei responsabilități expertului în informații, adică clasei care are informațiile necesare pentru a îndeplini responsabilitatea.

Information Expert Pattern

❑ Problema

- ❑ Care clasă este responsabilă de calcul notei pe care studentul o primește la curs?

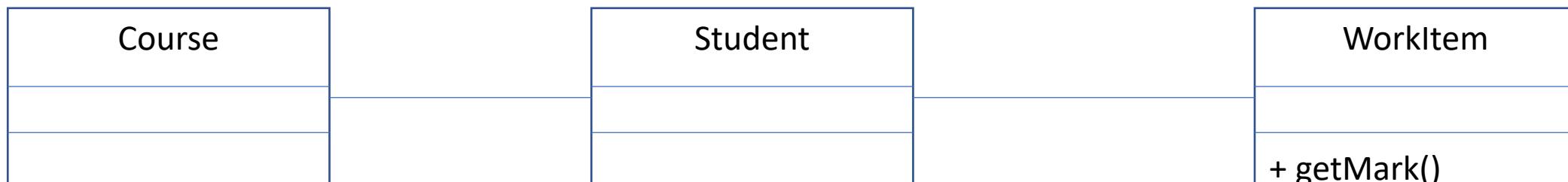
❑ Exemplu

❑ WorkItem?

- ❑ Clasa poate determina valoarea unei teme individuale, dar nu poate determina nota finală la o materie.

❑ Student?

- ❑ Clasei ar trebui să i se atribuie această responsabilitate, deoarece știe informații despre toate temele (dar nu înțelege cum este calculată nota).
- ❑ Clasa se bazează pe clasa WorkItem pentru a determina notele individuale.



Analogie cu lumea reală: pe cine întrebi despre X, întrebi expertul care știe despre X.

Information Expert Pattern

- ❑ Sistemul de notare poate fi modelat cu următoarele clase: **WorkItem**, **MarkingScheme**, **Student** și **Course**.
- ❑ Fie următoarele responsabilități:
 - ❑ Calculul mediei finale pentru un student,
 - ❑ Editarea unei teme,
 - ❑ Realizarea unui raport cu toate notele pentru o materie,
 - ❑ O listă cu toți studenții care participă la un curs,
- ❑ Utilizând Expert design pattern, se decide ce clasă este responsabilă pentru rezolvarea cerinței.
- ❑ Dacă nu există o astfel de clasă în sistem se recomandă introducerea unei noi clase care să preia responsabilitatea.

Creator pattern

□ Problema

- Cine creează o instanță a clasei A?

□ Soluție

- Atribuie clasei **B responsabilitatea de a crea** o instanță a clasei **A** dacă una din următoarele afirmații este adevărată
 - B conține un obiect sau o listă de obiecte de tipul A
 - B urmărește instanțe ale clasei A
 - B utilizează obiecte de tipul clasei A
 - B conține datele necesare clasei A pentru a fi instanțiată.

Creator pattern

- ❑ Sistemul de notare poate fi modelat cu următoarele clase: **WorkItem**, **MarkingScheme**, **Student** and **Course**.
- ❑ Problema
 - ❑ Cine este responsabil pentru crearea obiectelor de tip **MarkingScheme**?
 - ❑ Cine este responsabil pentru crearea obiectelor de tip **WorkItem**?

Low Coupling

❑ Problema

- ❑ Cum se poate menține o dependență scăzută între clase, un impact redus al modificărilor, reutilizarea sporită?

❑ Soluția

- ❑ Atribuirea unei responsabilități astfel încât cuplarea să fie redusă

❑ Coupling

- ❑ măsură a cât de puternic
 - ❑ este conectat un element la alte elemente
 - ❑ cunoaște un alte elemente
 - ❑ se bazează pe alte elemente

Low Coupling

Coupling

o măsură a cât de puternic:

- este conectat un element la alte elemente
- cunoaște un alte elemente
- se bazează pe alte elemente

Clasele care sunt cuplate strâns

- Sunt afectate de modificările din clasele relaționate
- Sunt mai greu de înțeles și întreținut
- Sunt mai greu de refolosit

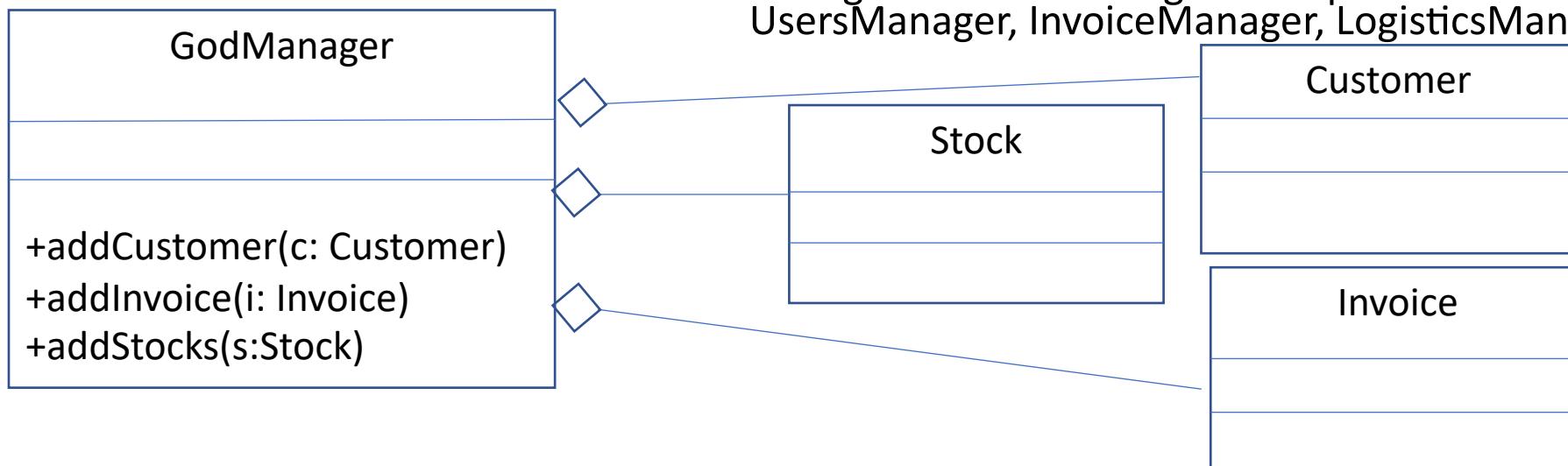
Cuplarea este necesară, deoarece clasele schimbă mesaje!

Problema este dacă abuzăm de cuplare sau este prea instabilă.

Low Coupling

❑ Manager include logică pentru lucrul cu

- Customers
- Invoices
- Logistics
- ...



❑ Simply for everything.

❑ "god objects" -> au prea multe responsabilități-> creează prea multe dependințe

❑ **Numărul total de dependințe din aplicație nu contează**, important este numărul de referințe **între obiecte**.

❑ O clasă ar trebui să comunice cu cât mai puține clase posibil,

❑ Adăugarea de clase de gestiune pentru fiecare entitate, ex. UsersManager, InvoiceManager, LogisticsManager și altelle.

Controller

□ Problemă

□ Care obiecte în spatele nivelului interfeței utilizator (User Interface - UI) primește și coordonează operațiile sistemului? (Cine este responsabil de gestionarea evenimentelor sistemului?)

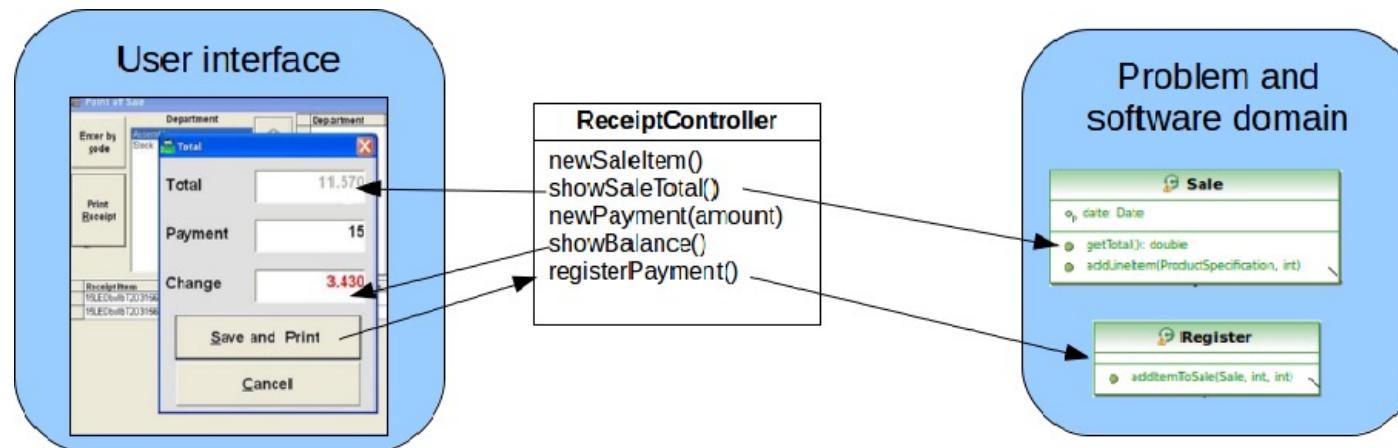
□ Soluție

□ Atribuirea responsabilității de primire și/sau tartare a evenimentelor sistemului în una din cele două situații:

- Obiect care reprezintă sistemul în general, dispozitiv sau subsistem (*façade controller*)
- Obiect care reprezintă un caz de utilizare când apare un eveniment în system (*<UseCase>Handler*)

Controller

- ❑ Clasele controller fac legătura între evenimentele sistemului și modelul software.
- ❑ Entity, Boundary, și Control Objects
 - ❑ **Entity** obiecte care sunt instanțe a claselor domeniului.
 - ❑ **Boundary** obiecte care reprezintă interacțiunea dintre actori și sistem
 - ❑ **Control** obiecte care se ocupă de realizarea cazurilor de utilizare.



High Cohesion

❑ Problema

❑ Cum să se mențină obiectele focusate, ușor de înțeles și gestionat?

❑ Soluția

❑ Atribuirea responsabilităților astfel încât coeziunea să rămână ridicată

❑ Cohesion

❑ o măsură a cât de puternic legate și conectate sunt responsabilitățile unui item (clăsă, subsistem etc.)

High Cohesion

❑ Gradul de coeziune

❑ Foarte scăzut

❑ O clasă este singura responsabilă pentru multe lucruri din domenii funcționale foarte diferite. Dacă majoritatea programelor sunt implementate într-o singură clasă, atunci acea clasă ar avea o coeziune foarte scăzută.

❑ Scăzut

❑ O clasă are responsabilitatea exclusivă pentru o sarcină complexă într-o zonă funcțională.

❑ Ridicat

❑ O clasă are responsabilități moderate într-o zonă funcțională și colaborează cu alte clase pentru a îndeplini sarcina.

❑ O analogie din lumea reală a coeziunii scăzute este o persoană care își asumă prea multe responsabilități fără legătură, în special cele care ar trebui delegate în mod corespunzător altora.

Polymorphism

□ Problemă

- Cum să se gestioneze elementele înrudite, dar diferite, în funcție de tipul de element?

□ Soluție

- Polimorfismul ghidează în a decide ce obiect este responsabil pentru manipularea elementelor diferite.

□ Beneficii

- Adăugarea de noi variatii este ușor de realizat.

Pure Fabrication

❑ Problemă

❑ Care obiect ar trebui să aibă responsabilitatea, atunci când nu dorim să încălcăm principiile de Coeziune Înaltă și Cuplare Scăzută, sau alte obiective, dar soluțiile oferite de Expert nu sunt adecvate.

❑ Soluție

❑ Atribuirea unei multimi extrem de coeziv de responsabilități unei clase artificiale sau conveniente care nu reprezintă un concept de domeniul problemei - ceva inventat, pentru a sprijini o coeziune ridicată, cuplare scăzută și reutilizare.

Pure Fabrication

❑ Pure Fabrication sugerează crearea unei noi clase pentru noile responsabilități

❑ Exemplu

❑ Stocarea cursurilor într-un format persistent

❑ *Clasa PersistentStorage este fabricată artificial*

❑ *Este rodul imaginatiei; nu are fundamente în modelul domeniului*

❑ Clasa Course rămâne: well-designed - high cohesion, low coupling

❑ Clasa PersistentStorage are ca scop să scrie/citească obiecte într-un/dintr-un sistem persistent (baza de date, fișier, ...)

PersistentStorage

+insert()
+update()
+delete()

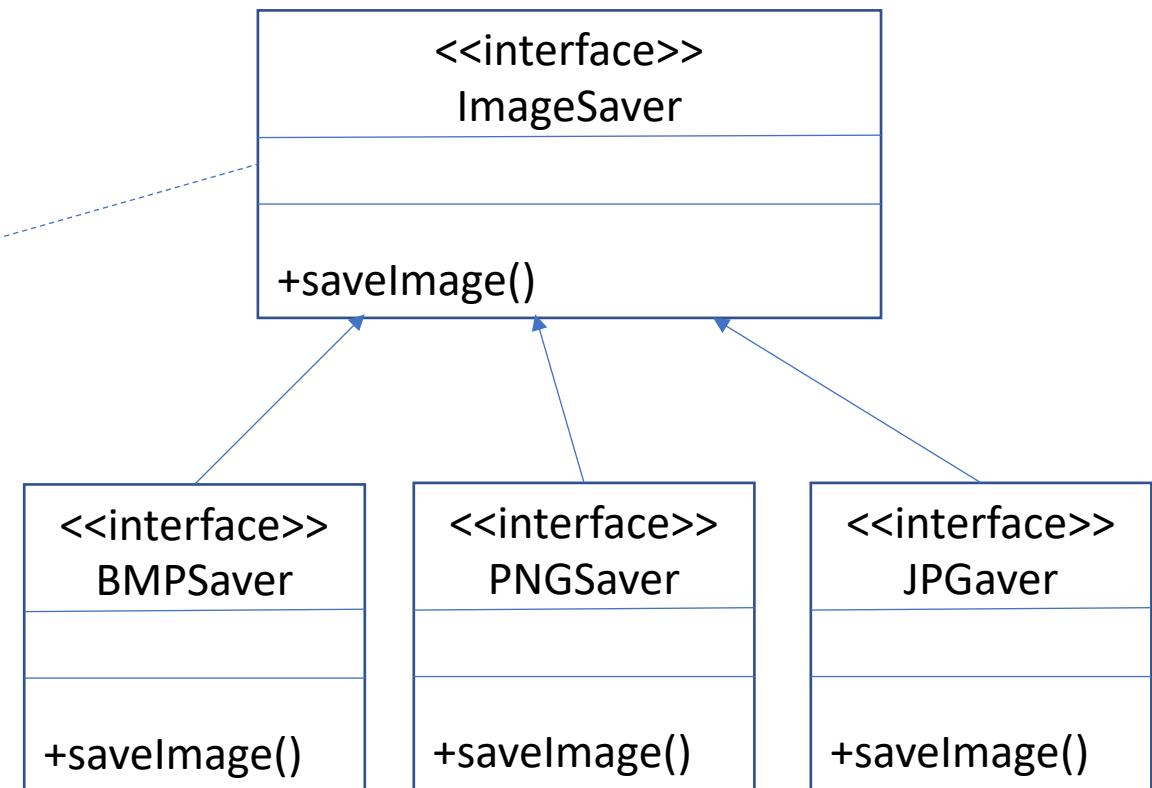
Pure Fabrication

Fară Pure Fabrication

Image
-bytesPerPixel: int
-channels:int
-columns:int
-rows:int
+saveBMPImage()
+savePNGImage
+saveJPEGImage()

Utilizând Pure Fabrication

Image
-bytesPerPixel: int
-channels:int
-columns:int
-rows:int
+saveImage(ImageSaver)



Indirection

□ Problemă

□ Cum putem evita o cuplare directă între două sau mai multe elemente?

□ Soluție

□ Indirectarea introduce o unitate intermediară pentru a comunica între celelalte unități, astfel încât celelalte unități să nu fie cuplate direct.

Indirection

□ Problemă

- Într-un sistem de vânzare, există mai multe moduri de calculare externe de taxe terțe care trebuie să fie acceptate
- Clasa de vânzare este responsabilă de calcularea taxelor
- Dorim să menținem sistemul independent de diferitele moduri de calculare externe de taxe

Protected Variation

□ Problemă

- Cum să se evite impactul variațiilor unor elemente asupra celorlalte elemente?



□ Soluție

- Realizarea unei interfațe bine definită, astfel încât să nu aibă niciun efect asupra altor unități.
- Realizarea unui sistem flexibil și protejat împotriva variațiilor.
- Realizarea unui design structurat.

□ Exemplu: polimorfism, încapsularea datelor, interfețe

Protected Variation

- ❑ Exemple
 - ❑ Incapsularea datelor, interfețe, polimorfism, indirectionare, și standarde.
 - ❑ Mașinile virtuale
 - ❑ Service lookup: clienții sunt protejați de variațiile în locația serviciilor, folosind interfața stabilă a serviciilor de căutare.
 - ❑ Principiul accesului uniform
 - ❑ ...

Concluzii

- ❑ GRASP oferă o serie de principii importante în proiectarea unei aplicații OO
- ❑ În același timp, GRASP lasă mâna liberă proiectantului crearea unui design bun este o artă!
- ❑ Urmarea principiilor GRASP—realizarea de diagrame UML și aplicarea de şabloane — este o bună practică pentru dezvoltatorii de aplicații OO care sunt încă neexperimentați

Bibliografie

- Larman, Craig. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*. Third edition, Prentice Hall, 2005
- Wirfs-Brock, Rebecca and McKean, Alan. *ObjectDesign: Roles, Responsibilities, and Collaborations*. Addison-Wesley Professional, 2002
- Evans, Eric. *Domain-DrivenDesign: Tackling Complexity in the Heartof Software*. Addison-Wesley Professional, 2003



ÎNTREBĂRI