# Ingineria Programării

Cursul 5 – 22,23 Martie 2022

adiftene@info.uaic.ro

# Cuprins

- Din Cursurile trecute…
- SOLID and Other Principles
- GRASP
  - Low coupling
  - High cohesion

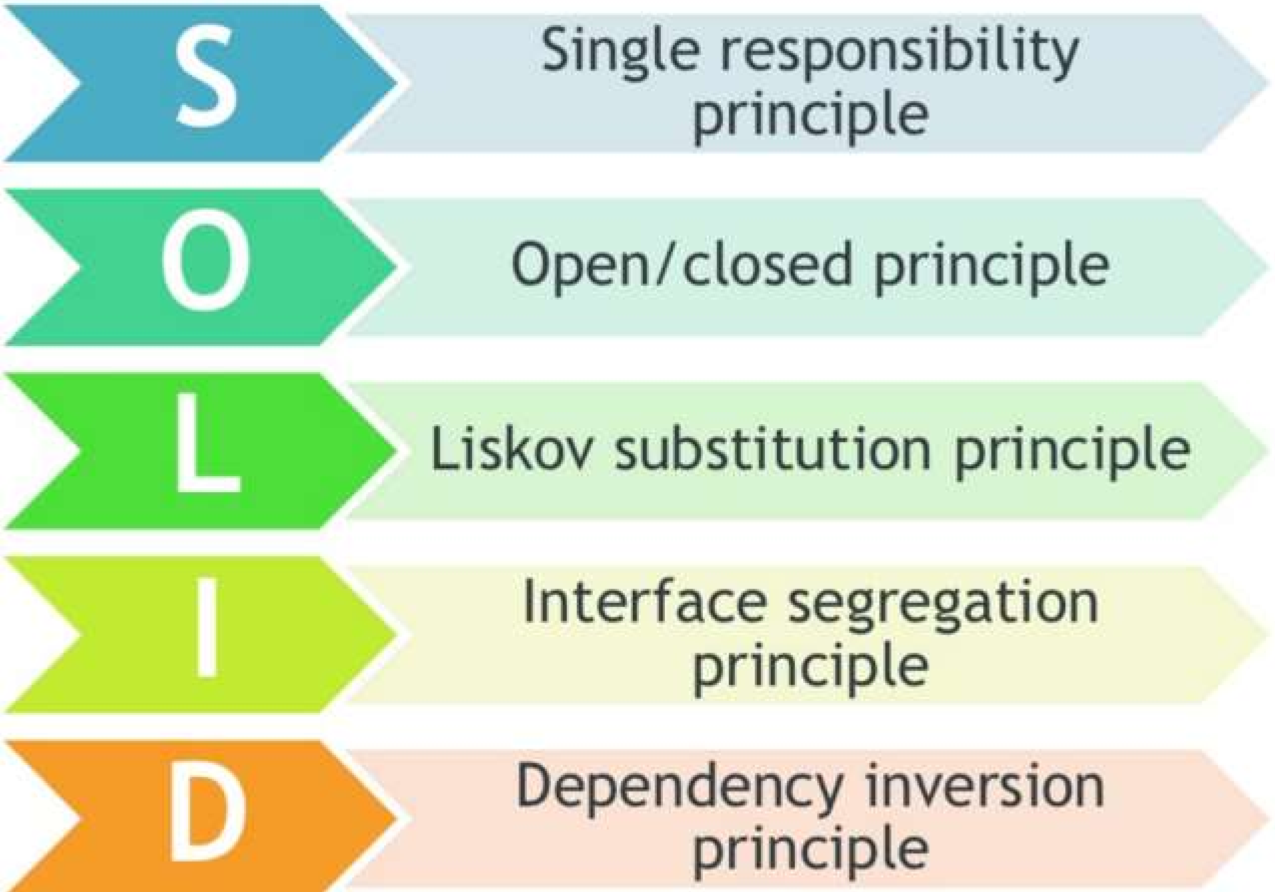
- De ce avem nevoie de modelare?
- Cum putem modela un proiect?
- SCRUM – roles, values, artifacts, events, rules

# SOLID and Other Principles

- SOLID Principles
  - SRP – Single Responsibility Principle
  - OCP – Open/Closed Principle
  - LSP – Liskov Substitution Principle
  - ISP – Interface Segregation Principle
  - DIP – Dependency Inversion Principle
- DRY – Don't Repeat Yourself
- YAGNI – You Aren't Gonna Need It
- KISS – Keep It Simple, Stupid

# SOLID

- SOLID was introduced by Robert C. Martin in the an article called the “Principles of Object Oriented Design” in the early 2000s



S Single responsibility principle

O Open/closed principle

L Liskov substitution principle

I Interface segregation principle

D Dependency inversion principle

# SOLID – SRP – Definitions

- “The Single Responsibility Principle states that every object should have a single responsibility, and that responsibility should be entirely encapsulated by the class.” – Wikipedia
- “There should never be more than one reason for a class to change.” – Robert Martin
- Low coupling & strong cohesion

# SOLID – SRP – Problems & Solutions

- Classic violations
  - Objects that can print/draw themselves
  - Objects that can save/restore themselves
- Classic solution
  - Separate printer & Separate saver
- Solution
  - Multiple small interfaces (ISP)
  - Many small classes
  - Distinct responsibilities
- Result
  - Flexible design
  - Lower coupling & Higher cohesion

# SOLID – SRP – Example

- Two responsabilities

```
interface Modem {
 public void dial(String pno);
 public void hangup();

 public void send(char c);
 public char recv();
}
```

- Separated interfaces

```
interface DataChannel {
 public void send(char c);
 public char recv();
}
```

```
interface Connection {
 public void dial(String phn);
 public char hangup();
}
```

# SOLID – Open/Closed Principle

- *Open chest surgery is not needed when putting on a coat*
- Bertrand Meyer originated the OCP term in his 1988 book, *Object Oriented Software Construction*

# SOLID – OCP – Definitions

- “The Open / Closed Principle states that software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.” – Wikipedia
- “All systems change during their life cycles. This must be borne in mind when developing systems expected to last longer than the first version.” – Ivar Jacobson
- **Open** to Extension – New behavior can be added in the future
- **Closed** to Modification – Changes to source or binary code are not required

# SOLID – OCP – How?

- Change behavior without changing code?!
  - Rely on abstractions, not implementations
  - Do not limit the variety of implementations
- In .NET – Interfaces, Abstract Classes
- In procedural code – Use parameters
- Approaches to achieve OCP
  - Parameters – Pass delegates / callbacks
  - Inheritance / Template Method pattern – Child types override behavior of a base class
  - Composition / Strategy pattern – Client code depends on abstraction, "Plug in" model

# SOLID – OCP – Problems & Solutions

- Classic violations
  - Each change requires re-testing (possible bugs)
  - Cascading changes through modules
  - Logic depends on conditional statements
- Classic solution
  - New classes (nothing depends on them yet)
  - New classes (no legacy coupling)
- When to apply OCP?
  - Experience tell you
- OCP add complexity to design (TANSTAAFL)
- No design can be closed against all changes

# SOLID – OCP – Example

```
// Open-Close Principle - Bad example
class GraphicEditor {
 public void drawShape(Shape s) {
  if (s.m_type==1)
        drawRectangle(s);
  else if (s.m_type==2)
        drawCircle(s);
 }
 public void drawCircle(Circle r)
{....}
 public void drawRectangle(Rectangle r)
{....}
}

class Shape {
 int m_type;
}

class Rectangle extends Shape {
 Rectangle() {super.m_type=1;}
}

class Circle extends Shape {
 Circle() {super.m_type=2;}
}
```

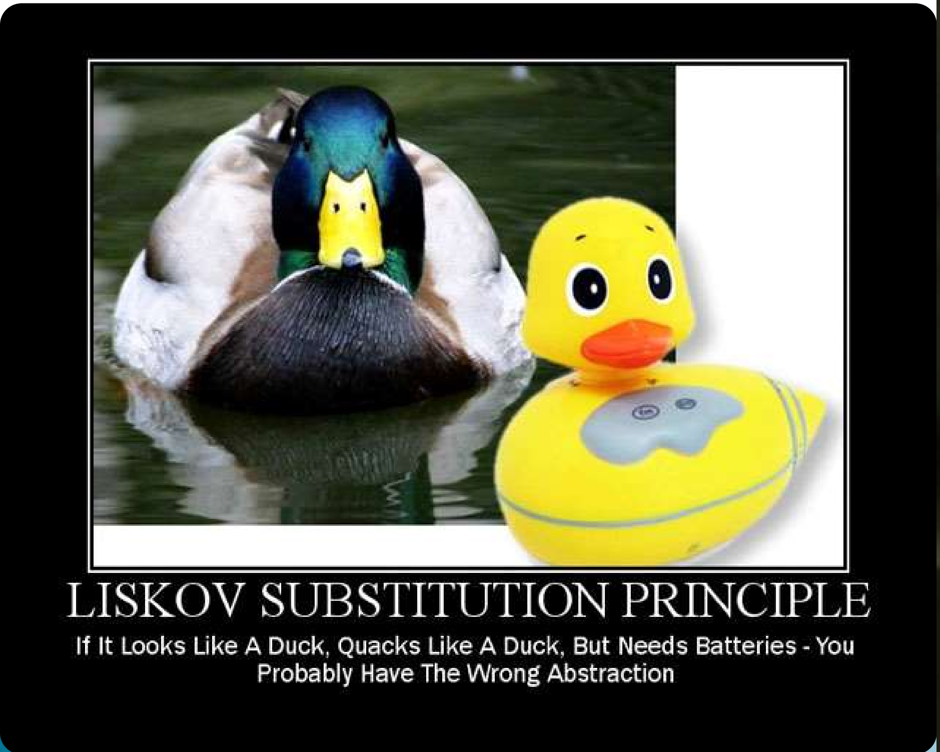
```
// Open-Close Principle - Good
example
class GraphicEditor {
 public void drawShape(Shape s) {
        s.draw();
 }
}

class Shape {
        abstract void draw();
}

class Rectangle extends Shape {
 public void draw() {
        // draw the rectangle
 }
}
```

# SOLID – Liskov Substitution

- If it looks like a duck, quacks like a duck, but needs batteries – you probably have the wrong abstraction
- Barbara Liskov described the principle in 1988


LISKOV SUBSTITUTION PRINCIPLE
If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You Probably Have The Wrong Abstraction

# SOLID – LSP – Definitions

- "The Liskov Substitution Principle states that Subtypes must be substitutable for their base types." – Agile Principles, Patterns, and Practices in C#
- Substitutability – child classes must not
  - Remove base class behavior
  - Violate base class invariants
- Normal OOP inheritance
  - IS-A relationship
- Liskov Substitution inheritance
  - IS-SUBSTITUTABLE-FOR

# SOLID – LSP – Problems & Solutions

- The problem
  - Polymorphism break Client code expectations
  - "Fixing" by adding if-then – nightmare (OCP)
- Classic violations
  - Type checking for different methods
  - Not implemented overridden methods
  - Virtual methods in constructor
- Solutions
  - “Tell, Don’t Ask” – Don’t ask for types and Tell the object what to do
  - Refactoring to base class– Common functionality and Introduce third class

# SOLID – LSP – Example (1)

```
// Violation of Liskov's Substitution Principle
 class Rectangle{
   int m_width;
   int m_height;

   public void setWidth(int width){
        m_width = width;
   }

   public void setHeight(int h){
        m_height = ht;
   }

   public int getWidth(){
        return m_width;
   }

   public int getHeight(){
        return m_height;
   }

   public int getArea(){
        return m_width * m_height;
   }
 }
```

```
class Square extends Rectangle {
  public void setWidth(int width){
        m_width = width;
        m_height = width;
  }

  public void setHeight(int height){
        m_width = height;
        m_height = height;
  }
 }
```

# SOLID – LSP – Example (2)

```
class LspTest
{
private static Rectangle getNewRectangle()
{
        // it can be an object returned by some factory ...
        return new Square();
}

public static void main (String args[])
{
        Rectangle r = LspTest.getNewRectangle();
        r.setWidth(5);
        r.setHeight(10);

// user knows that r it's a rectangle. It assumes that he's able to set the width
and height as for the base class

        System.out.println(r.getArea());
        // now he's surprised to see that the area is 100 instead of 50.
}
}
```

# SOLID – Interface Segregation

- You want me to plug this in. Where?

INTERFACE SEGREGATION PRINCIPLE
You Want Me To Plug This In, Where?

Interface Segregation Principle
If IRequireFood, I want to Eat(Food food) not,
LightCandelabra() or LayoutCutlery(CutleryLayout preferredLayout)

Not bad!
DRINK + EAT
EAT
DRINK

# SOLID – ISP – Definitions

- “The Interface Segregation Principle states that Clients should not be forced to depend on methods they do not use.” – Agile Principles, Patterns, and Practices in C#
- Prefer small, cohesive interfaces – Interface is the interface type + All public members of a class
- Divide "fat" interfaces into smaller ones
  - “fat” interfaces means classes with useless methods, increased coupling, reduced flexibility and maintainability

# SOLID – ISP – Problems & Solutions

- Classic violations
  - Unimplemented methods (also in LSP)
  - Use of only small portion of a class
- When to fix?
  - Once there is pain! Do not fix, if is not broken!
  - If the "fat" interface is yours, separate it to smaller ones
  - If the "fat" interface is not yours, use "Adapter" pattern
- Solutions
  - Small interfaces
  - Cohesive interfaces
  - Focused interfaces
  - Let the client define interfaces
  - Package interfaces with their implementation

# SOLID – ISP – Example

```
//Bad example (polluted interface)


interface Worker {

 void work();

 void eat();

}


ManWorker implements Worker {

        void work() {…};

        void eat() {30 min break;};

}


RobotWorker implements Worker {

        void work() {…};

        void eat() {//Not Appliciable
                for a RobotWorker};

}
```

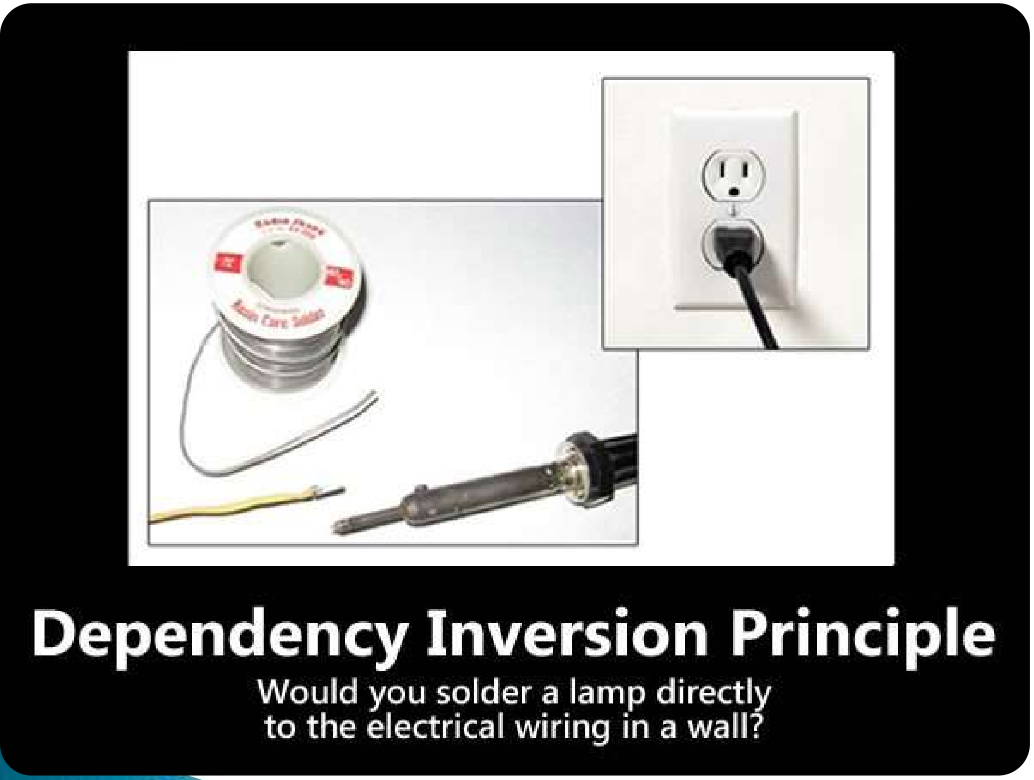
```
//Solution: split into two interfaces


interface Workable {
        public void work();
 }

 interface Feedable{
        public void eat();
 }
```

# SOLID – Dependency Inversion

- Would you solder a lamp directly to the electrical wiring in a wall?


Dependency Inversion Principle
Would you solder a lamp directly
to the electrical wiring in a wall?

SONY
Cyber-shot
Port doesn't define device

# SOLID – DIP – Definitions

- “High–level modules should not depend on low–level modules. Both should depend on abstractions.”

- “Abstractions should not depend on details. Details should depend on abstractions.” – Agile Principles, Patterns, and Practices in C#

# SOLID – DIP – Dependency

- Framework
- Third Party Libraries
- Database
- File System
- Email
- Web Services
- System Resources (Clock)
- Configuration
- The new Keyword
- Static methods
- Thread.Sleep
- Random

# SOLID – DIP – Problems & Solutions

- How it should be
  - Classes should declare what they need
  - Constructors should require dependencies
  - Dependencies should be abstractions and be shown
- How to do it
  - Dependency Injection
  - The Hollywood principle "Don't call us, we'll call you!"
- Classic violations
  - Using of the new keyword, static methods/properties
- How to fix?
  - Default constructor, main method/starting point
  - Inversion of Control container

# SOLID – DIP – Example

```
//DIP - bad example
public class EmployeeService {
        private EmployeeFinder emFinder //concrete class, not abstract. Can access a SQL DB for instance
        public Employee findEmployee(…) {
                emFinder.findEmployee(…)
        }
}

//DIP - fixed
public class EmployeeService {
        private IEmployeeFinder emFinder //depends on an abstraction, no an implementation
        public Employee findEmployee(…) {
                emFinder.findEmployee(…)
        }
}
//Now its possible to change the finder to be a XmlEmployeeFinder, DBEmployeeFinder, FlatFileEmployeeFinder,
MockEmployeeFinder….
```

# Other Principles

- Don't Repeat Yourself (DRY)
- You Ain't Gonna Need It (YAGNI)
- Keep It Simple, Stupid (KISS)

# OP – Don't Repeat Yourself

- Repetition is the root of all software evil

# OP – DRY – Definitions

- "Every piece of knowledge must have a single, unambiguous representation in the system.“ – The Pragmatic Programmer
- "Repetition in logic calls for abstraction. Repetition in process calls for automation.“ – 97 Things Every Programmer Should Know
- Variations include:
  - Once and Only Once
  - Duplication Is Evil (DIE)

# OP – DRY – Problems

- Magic Strings/Values
- Duplicate logic in multiple locations
- Repeated if-then logic
- Conditionals instead of polymorphism
- Repeated Execution Patterns
- Lots of duplicate, probably copy-pasted, code
- Only manual tests
- Static methods everywhere

# OP – You Ain't Gonna Need It

- Don’t waste resources on what you might need

# OP – YAGNI – Definitions

- "A programmer should not add functionality until deemed necessary.“ – Wikipedia

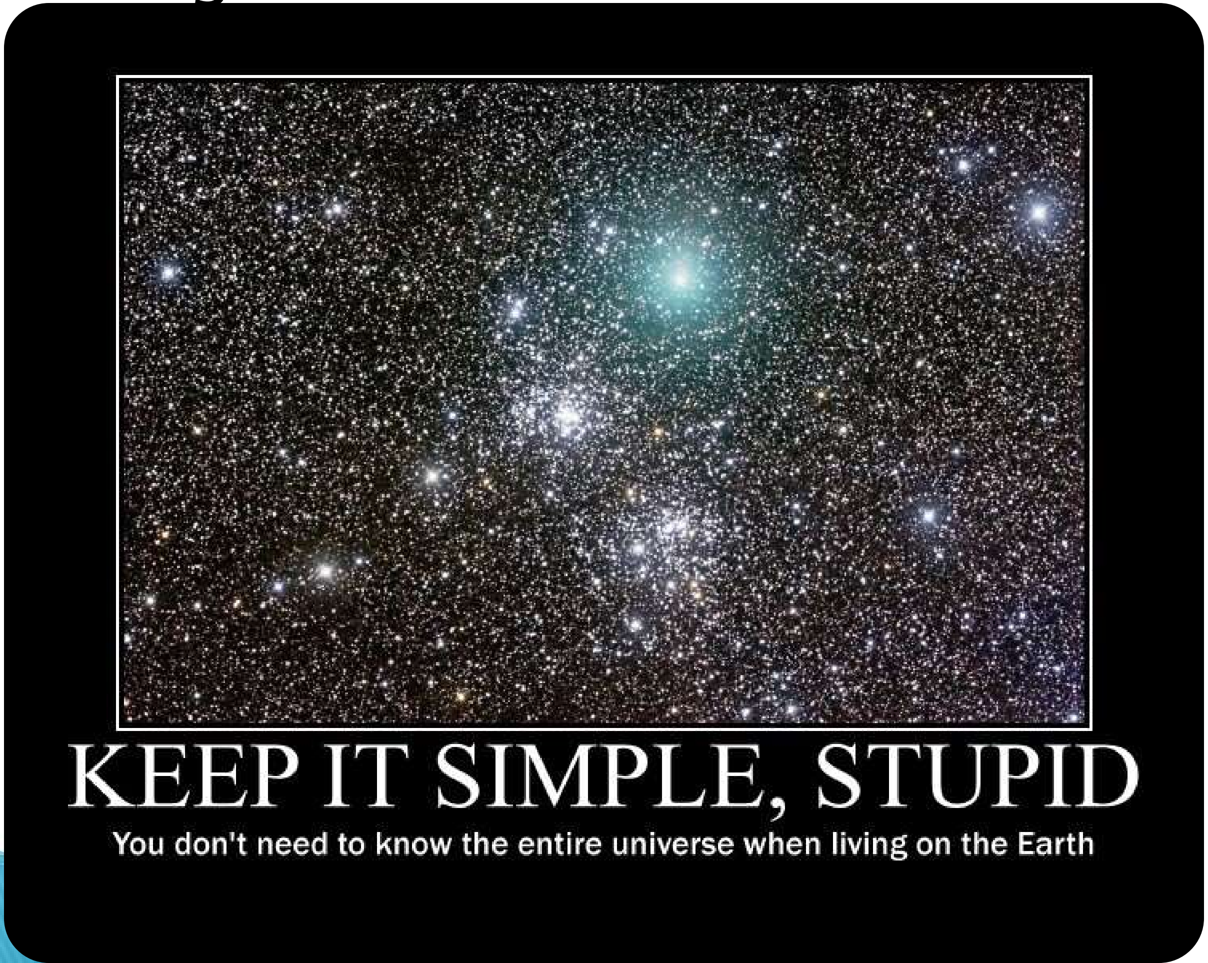
- "Always implement things when you actually need them, never when you just foresee that you need them.“ – Ron Jeffries, XP co-founder

# OP – YAGNI – Problems

- Time for adding, testing, improving
- Debugging, documented, supported
- Difficult for requirements
- Larger and complicate software
- May lead to adding even more features
- May be not know to clients

# OP – Keep It Simple, Stupid

- You don’t need to know the entire universe when living on the Earth


KEEP IT SIMPLE, STUPID
You don't need to know the entire universe when living on the Earth

# OP – KISS – Definitions

- "Most systems work best if they are kept simple.“ – U.S. Navy

- "Simplicity should be a key goal in design and unnecessary complexity should be avoided.“ – Wikipedia

# GRASP

- GRASP = General Responsibility Assignment Software Patterns (Principles)
- Descrise de Craig Larman în cartea *Applying UML and Patterns. An Introduction to Object Oriented Analysis and Design*
- **Ne ajută să alocăm responsabilități claselor și obiectelor în cel mai elegant mod posibil**
- Exemple de principii folosite în GRASP: *Information Expert* (sau Expert), *Creator*, *High Cohesion*, *Low Couplig*, *Controller* Polymorphism, Pure Fabrication, Indirection, Protected Variations

# Ce responsabilități?

- Să facă:
  - Să facă ceva el însuși, precum crearea unui obiect sau să facă un calcul
  - Inițializarea unei acțiuni în alte obiecte
  - Controlarea și coordonarea activităților altor obiecte

- Să cunoască:
  - Atributele private
  - Obiectele proprii
  - Lucrurile pe care le poate face sau le poate apela

# Pattern

- Traducere: șablon, model
- Este o soluție generală la o problemă comună
- Fiecare pattern are un nume sugestiv și ușor de reținut (ex. composite, observer, iterator, singleton, etc.)

# Information Expert 1

- **Problemă**: dat un anumit comportament (operație), cărei clase trebuie să-i fie atribuit?

- O alocare bună a operațiilor conduce la sisteme care sunt:
  - Ușor de înțeles
  - Mai ușor de extins
  - Refolosibile
  - Mai robuste

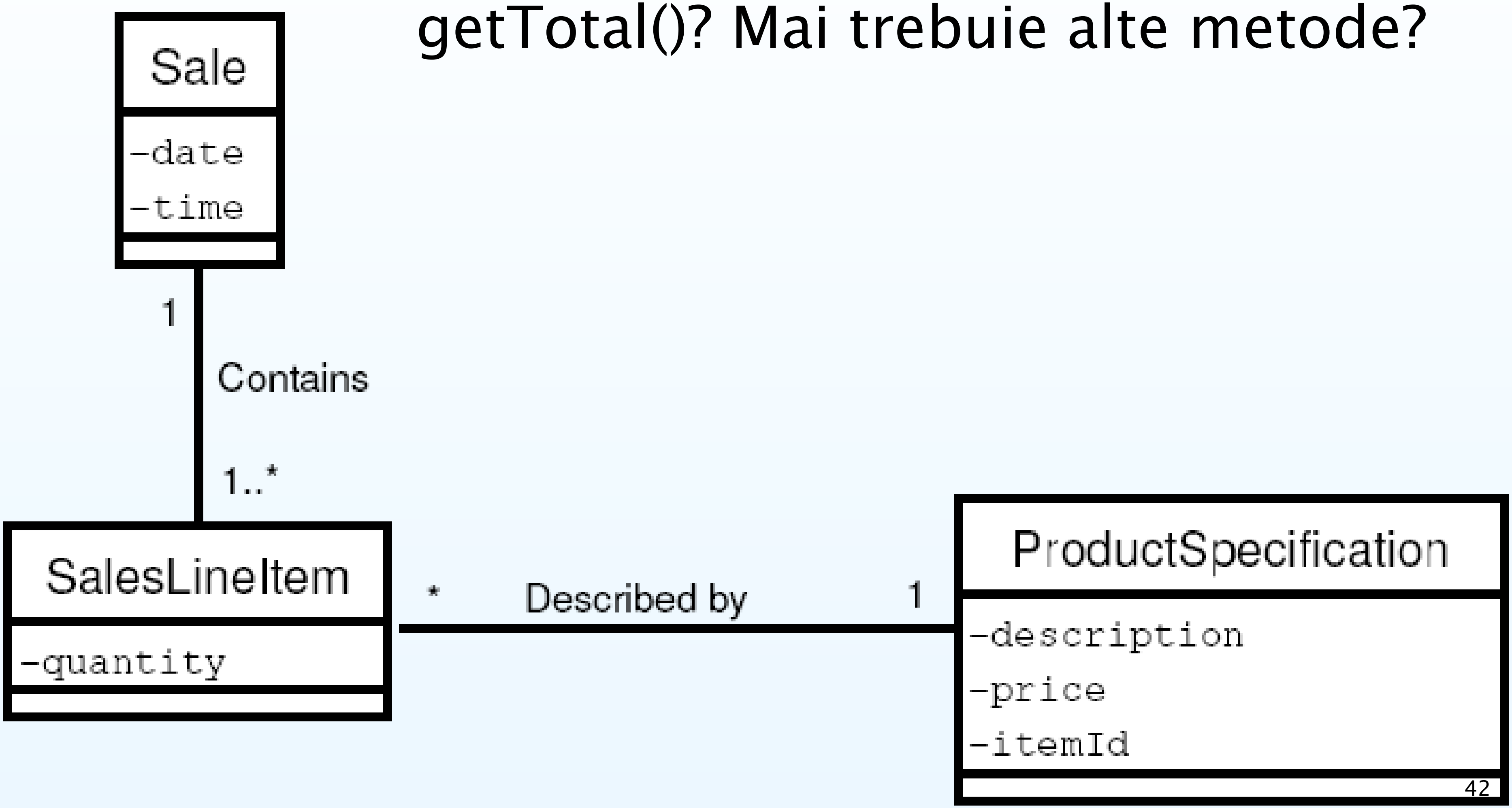
# Information Expert 2

- **Soluție**:
- asignez o responsabilitate clasei care are *informațiile necesare* pentru îndeplinirea acelei responsabilități

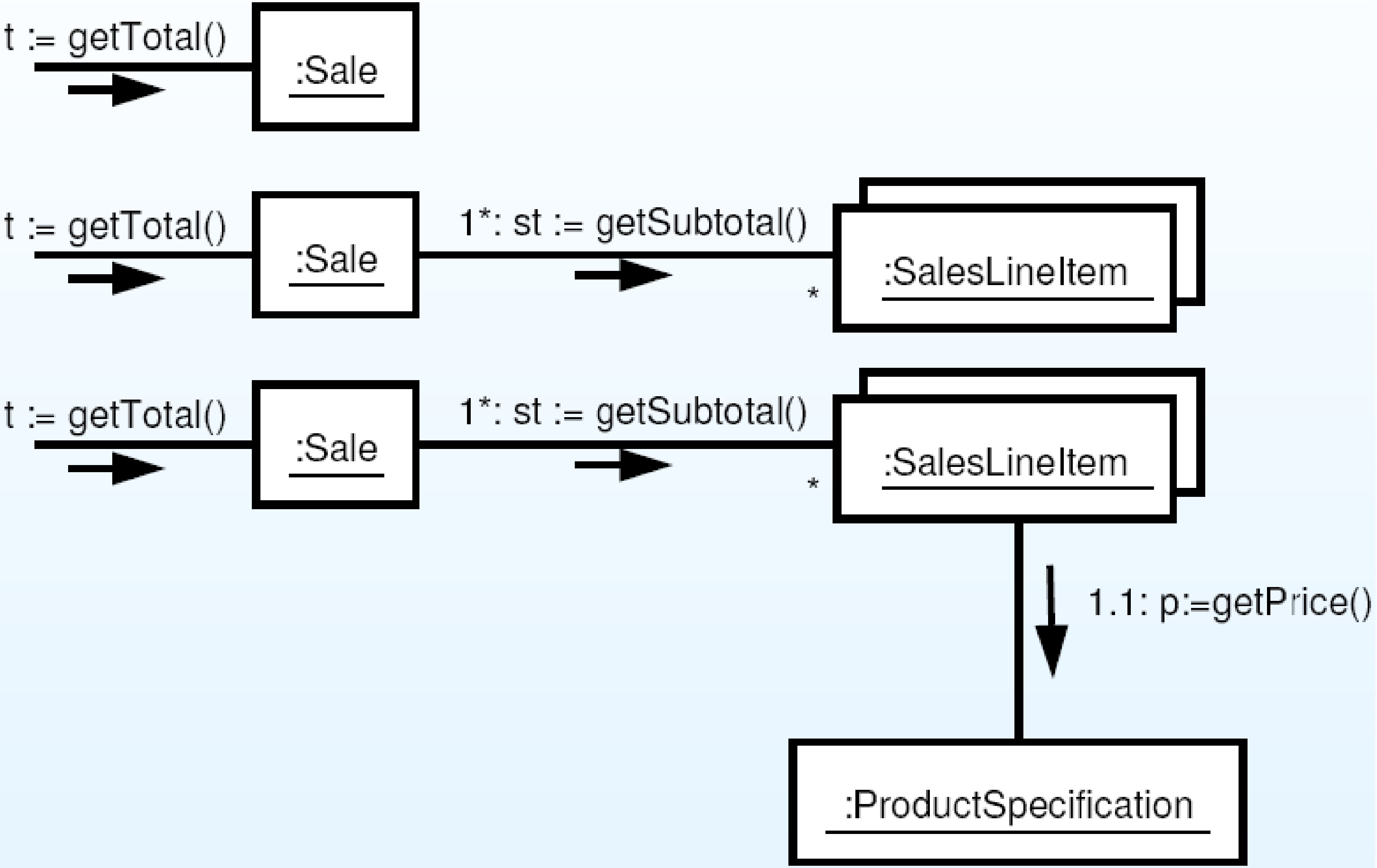
- **Recomandare**:
- începeți asignarea responsabilităților evidențiind clar care sunt responsabilitățile

# Exemplul 1

- Carei clase trebuie sa-i fie asignată metoda getTotal()? Mai trebuie alte metode?

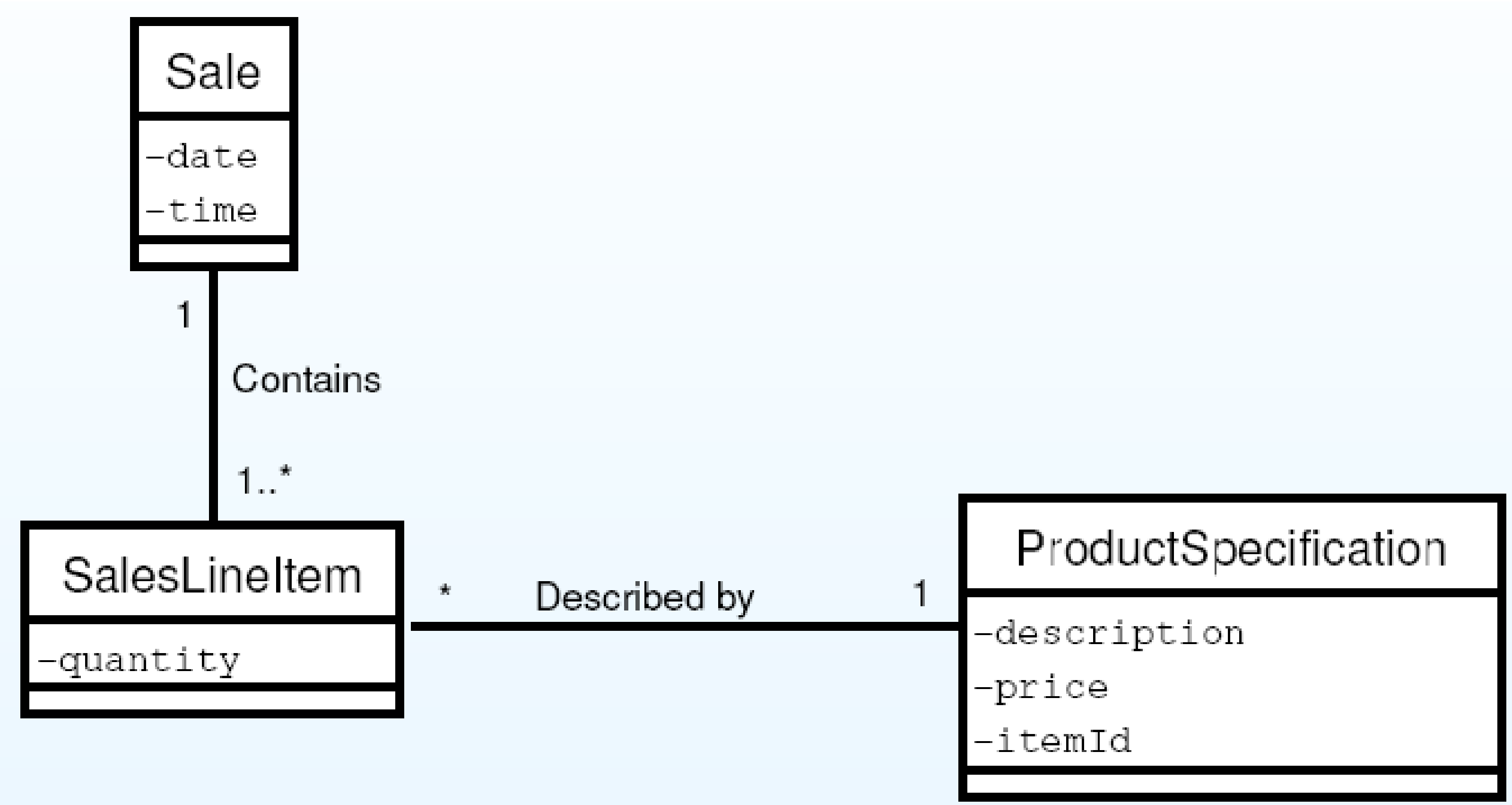


# Exemplul 2

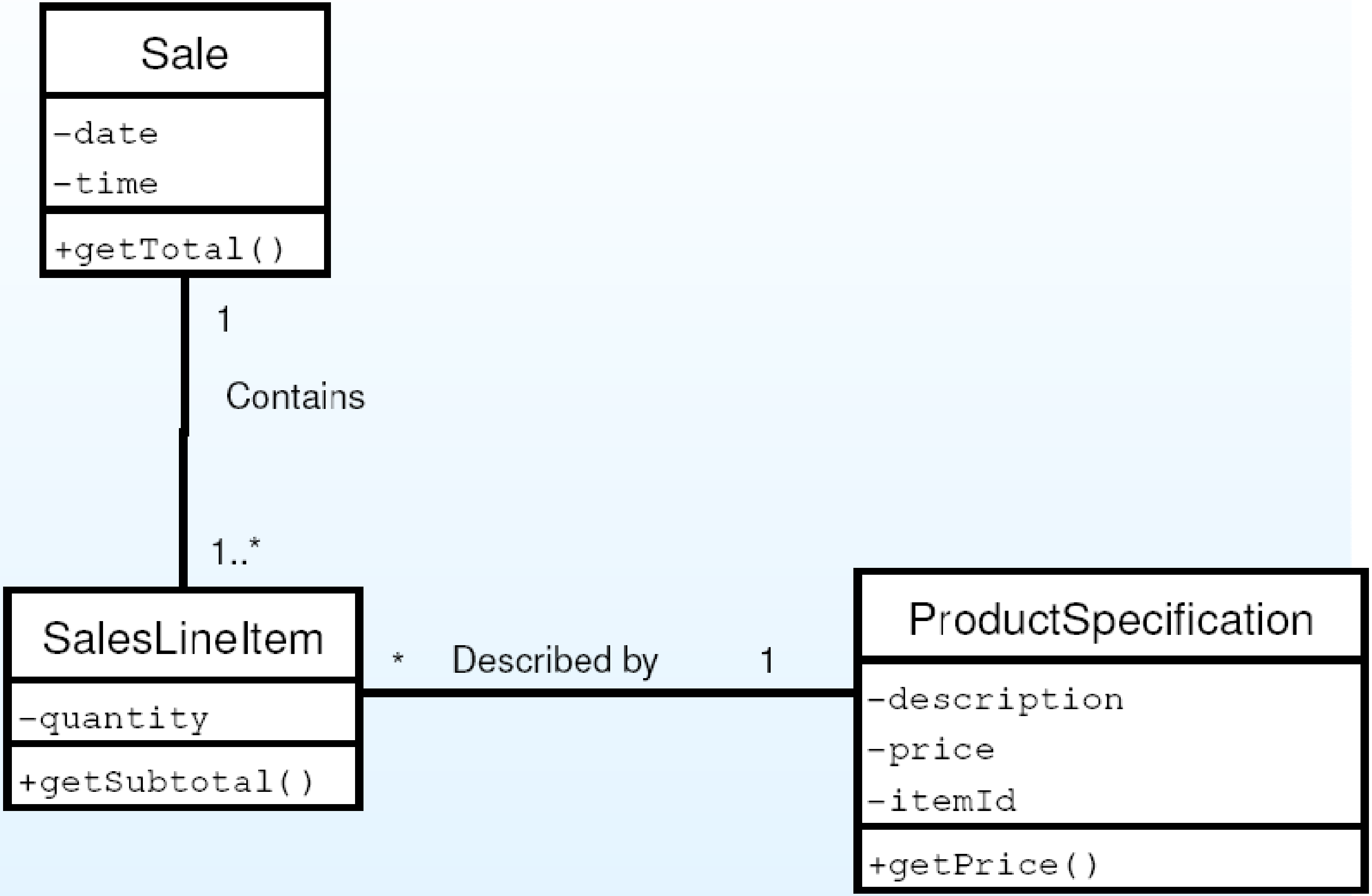

t := getTotal()
:Sale
t := getTotal()
:Sale
1*: st := getSubtotal()
*
:SalesLineItem
t := getTotal()
:Sale
1*: st := getSubtotal()
*
:SalesLineItem
1.1: p:=getPrice()
:ProductSpecification

# Soluție posibilă 1

| Clasă | Responsabilități |
|---|---|
| Sale | să cunoască valoarea totală a cumpărăturilor |
| SalesLineItem | să cunoască subtotalul pentru un produs |
| ProductSpecification | să cunoască prețul produsului |



# Soluţie posibilă 2

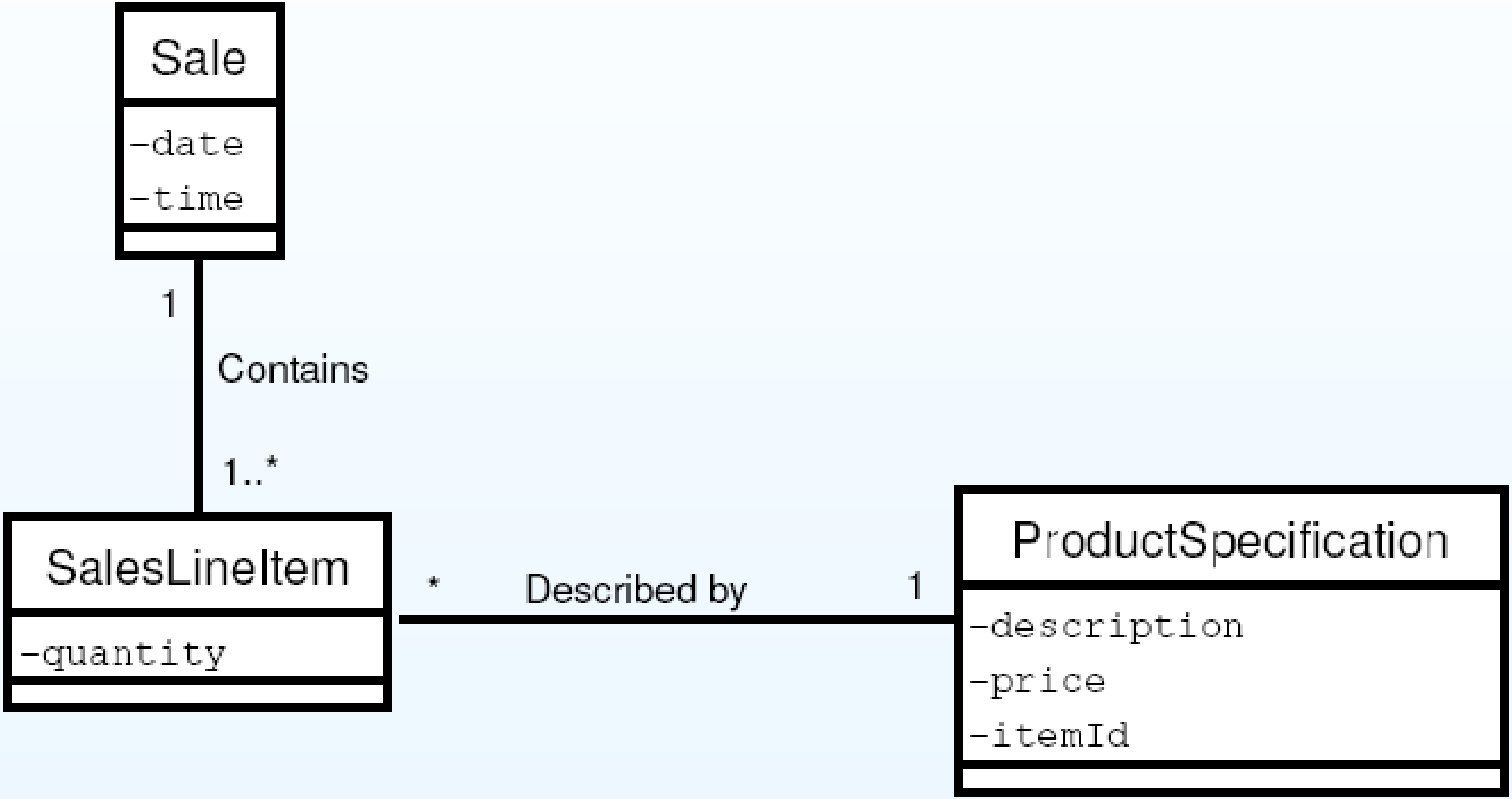


# Creator 1

- **Problemă**: cine trebie să fie responsabil cu crearea unei instanțe a unei clase?
- **Soluție**: Asignați clasei B responsabilitatea de a crea instanțe ale clasei A doar dacă cel puțin una dintre următoarele afirmații este adevărată:
  - B *agregă obiecte de tip A*
  - B *conține obiecte de tip A*
  - B *folosește obiecte de tip A*
  - B *are datele de inițializare care trebuie transmise la* instanțierea unui obiect de tip A (B este deci un Expert în ceea ce privește crearea obiectelor de tip A)
- *Factory pattern* este o variantă mai complexă

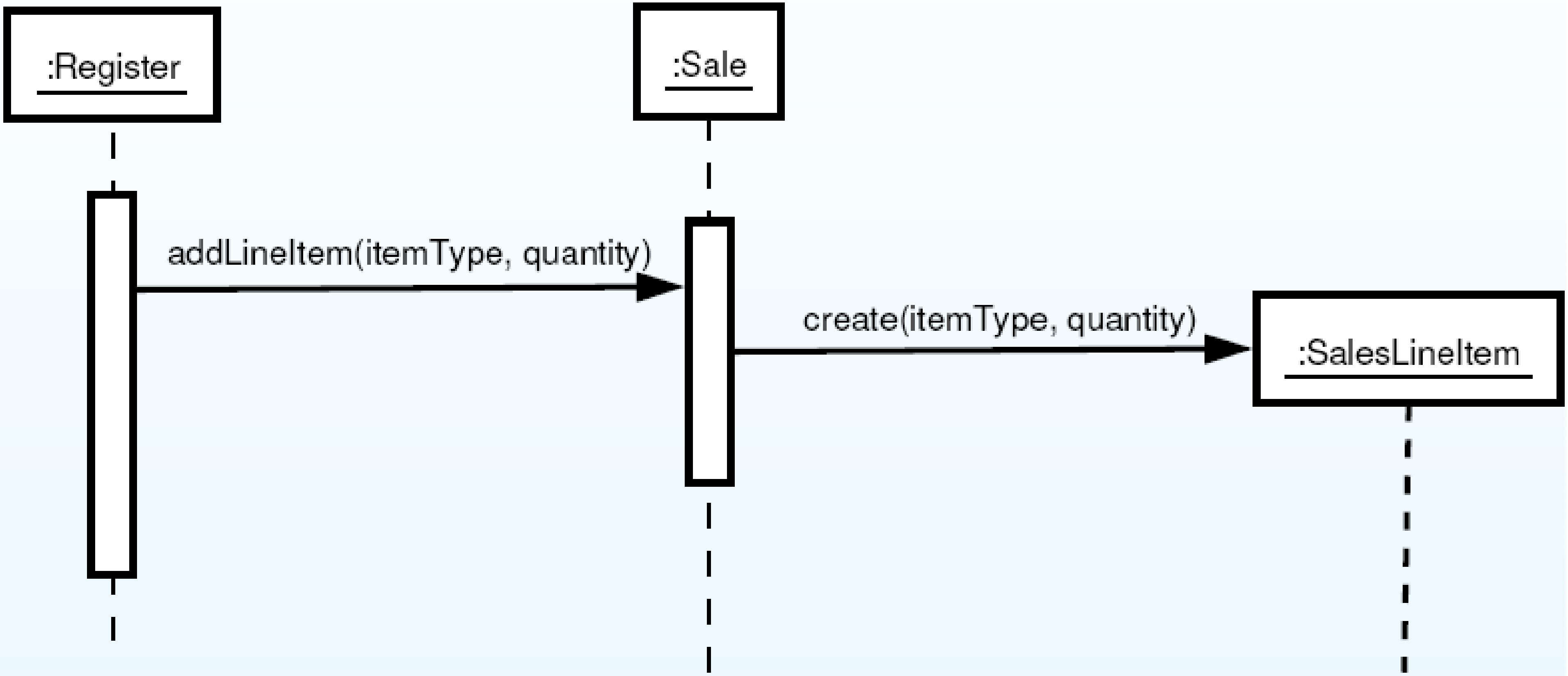
# Creator 2

- Cine este responsabil cu crearea unei instanțe a clasei SalesLineItem?



# Creator 3

- Deoarece Sale conține (agregă) instanțe de tip SalesLineItem, Sale este un bun candidat pentru a i se atribui responsabilitatea creării acestor instanțe



# Low coupling (cuplaj redus)

- Cuplajul este o măsură a gradului de dependență a unei clase de alte clase
- Tipuri de Dependență:
  - este conectată cu
  - are cunoștințe despre
  - se bazează pe
- **O clasă care are cuplaj mic (redus) nu depinde de “multe” alte clase; unde “multe” este dependent de contex**
- O clasă care are cuplaj mare depinde de multe alte clase

# Cuplaj 2

- Probleme cauzate de cuplaj:
  - schimbări în clasele relaționate forțează schimbări locale
  - clase **greu de înțeles** în izolare (scoase din context)
  - clase **greu de refolosit** deoarece folosirea lor presupune și prezența claselor de care depind

# Cuplaj 3

- Forme comune de cuplaj de la clasa A la clasa B sunt:
  - A are un atribut de tip B
  - O instanță a clasei A apelează un serviciu oferit de un obiect de tip B
  - A are o metodă care referențiază B (parametru, obiect local, obiect returnat)
  - A este subclasă (direct sau indirect) a lui B
  - B este o interfață, iar A implementează această interfață

# Legea lui Demeter

- *Don’t talk to strangers*

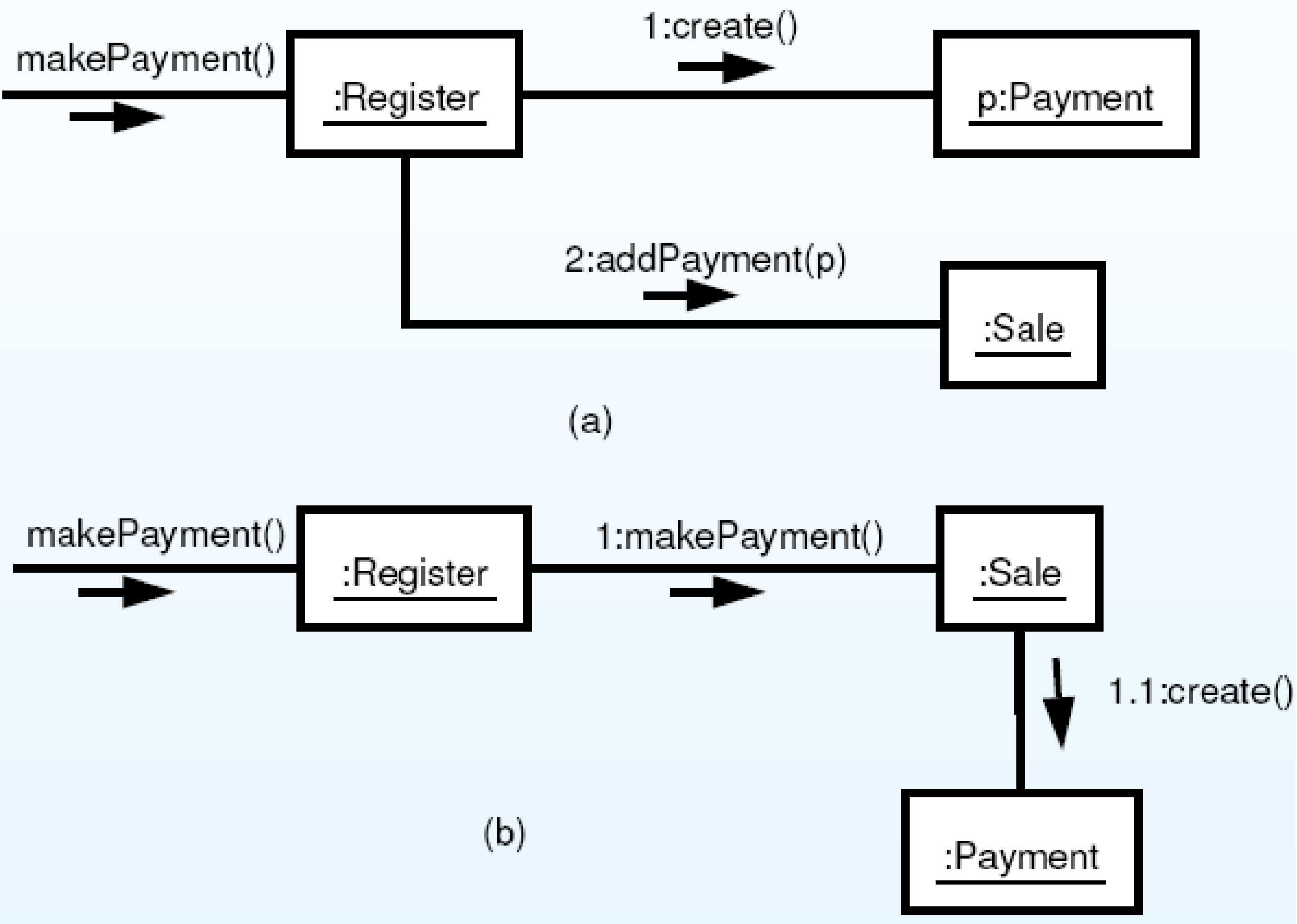
- Orice metodă a unui obiect trebuie să apeleze doar metode aparținând
  - lui însuși
  - oricărui parametru al metodei
  - oricărui obiect pe care l-a creat
  - oricăror obiecte pe care le conține

# Vizualizarea Cuplajelor

- Diagrama de clase
- Diagrama de colaborare

# Exemplul 1

- Exista legături între toate clasele
- Elimină cuplajul dintre Register și Payment



# High Cohesion

- **Coeziunea** este o măsură a cât de puternic sunt focalizate responsabilitățile unei clase

- O clasă ale cărei responsabilități sunt foarte strâns legate și care nu face foarte multe lucruri are o **coeziune mare**

- O clasă care face multe lucruri care nu sunt relaționate sau face prea multe lucruri are o **coeziune mică (slabă)**

# Coeziune

- Probleme cauzate de o slabă coeziune:
  - greu de înțeles
  - greu de refolosit
  - greu de menținut
  - delicate; astfel de clase sunt mereu supuse la schimbări

# Coeziune și Cuplaj

- Sunt principii vechi în design-ul software
- Promovează un design modular
- Modularitatea este proprietatea unui sistem care a fost descompus într-o mulțime de module coezive și slab cuplate

# Controller 1

- **Problemă**: Cine este responsabil cu tratarea unui eveniment generat de un actor?
- Aceste evenimente sunt asociate cu operații ale sistemului
- Un **Controller** este un obiect care nu ține de interfața grafică și care este responsabil cu recepționarea sau gestionarea unui eveniment
- Un Controller definește o metodă corespunzătoare operației sistemului

# Controller 2

- **Soluție**: asignează responsabilitatea pentru recepționarea sau gestionarea unui eveniment unei clase care reprezintă una dintre următoarele alegeri:
  - Reprezintă întregul sistem sau subsistem (fațadă controller)
  - Reprezintă un scenariu de utilizare în care apare evenimentul;

# Controller 3

- În mod normal, un controller ar trebui să delege altor obiecte munca care trebuie făcută;
- **Controller-ul coordonează sau controlează activitatea, însă nu face prea multe lucruri el însuși**
- O greșeală comună în design-ul unui controller este să i se atribuie prea multe responsabilități (fațade controller)

# Concluzii

- SOLID
- DRY, YAGNI, KISS
- GRASP

# Întrebări

- 1) Argumentați pentru folosirea SOLID.
- 2) Argumentați pentru folosirea diagramelor.
- 3) Veniți cu argumente pentru a nu folosi diagrame sau SOLID.
- 4) Cum putem folosi informațiile legate de coeziune și cuplaj? Când evaluăm un proiect. Când evaluăm un membru al echipei.

- Criticism: http://sourcemaking.com/design_patterns

# Bibliografie

- Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software* (GangOfFour)
- Ovidiu Gheorghieş, Curs 7 IP
- Adrian Iftene, Curs 9 TAIP: http://thor.info.uaic.ro/~adiftene/Scoala/2011/TAIP/Courses/TAIP09.pdf

# Bibliografie

- Craig Larman. *Applying UML and Patterns. An Introduction to Object Oriented Analysis and Design*

- Ovidiu Gheorghieș, Curs 6 IP

# Links (1)

- WebProjectManager: http://profs.info.uaic.ro/~adrianaa/uml/
- Diagrame de Stare și de Activitate: http://software.ucv.ro/~soimu_anca/itpm/Diagrame%20de%20Stare%20si%20Activitate.doc
- Deployment Diagram: http://en.wikipedia.org/wiki/Deployment_diagram http://www.agilemodeling.com/artifacts/deploymentDiagram.htm
- GRASP: http://en.wikipedia.org/wiki/GRASP_(Object_Oriented_Design)
- http://web.cs.wpi.edu/~gpollice/cs4233-a05/CourseNotes/maps/class4/GRASPpatterns.html
- Introduction to GRASP Patterns: http://faculty.inverhills.edu/dlevitt/CS%202000%20(FP)/GRASP%20Patterns.pdf

# Links (2)

- Gang-Of-Four: http://c2.com/cgi/wiki?GangOfFour, http://www.uml.org.cn/c%2B%2B/pdf/DesignPatterns.pdf
- Design Patterns Book: http://c2.com/cgi/wiki?DesignPatternsBook
- About Design Patterns: http://www.javacamp.org/designPattern/
- Design Patterns – Java companion: http://www.patterndepot.com/put/8/JavaPatterns.htm
- Java Design patterns: http://www.allapplabs.com/java_design_patterns/java_design_patterns.htm
- Overview of Design Patterns: http://www.mindspring.com/~mgrand/pattern_synopses.htm
- Gang of Four: http://en.wikipedia.org/wiki/Gang_of_four
- JUnit in Eclipse: http://www.vogella.de/articles/JUnit/article.html
- JUnit in NetBeans: http://netbeans.org/kb/docs/java/junit-intro.html

# Links (3)

- https://scotch.io/bar-talk/s-o-l-i-d-the-first-five-principles-of-object-oriented-design
- https://www.slideshare.net/enbohm/solid-design-principles-9016117
- https://siderite.blogspot.com/2017/02/solid-principles-plus-dry-yagni-kiss-final.html
- https://thefullstack.xyz/dry-yagni-kiss-tdd-soc-bdfu