Object-oriented design: GRASP patterns

J.Serrat

102759 Software Design
http://www.cvc.uab.es/shared/teach/a21291/web/

June 23, 2014

Index

1. Design & principles
2. GRASP
3. Expert
4. Creator
5. Low coupling
6. High cohesion
7. Controller
8. Polymorphism
9. Pure fabrication

What's software design ?

Design is

- Turning a specification for computer software into operational software.
- Links requirements to coding.
- One further step from the problem space to the computer-based solution

Object design

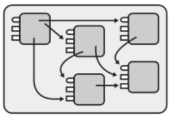
After identifying your requirements and creating a domain model, add methods to the software classes, and define the messaging between the objects to fulfil the requirements.

What's software design ?

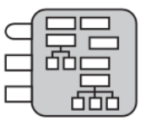
Design is needed at several different levels of detail in a system:

system

subsystems or packages: user interface, data storage, application-level classes, graphics ...



classes within packages, class relationships, interface of each class: public methods



attributes, private methods, inner classes ...

source code implementing methods

How to design object-oriented ?

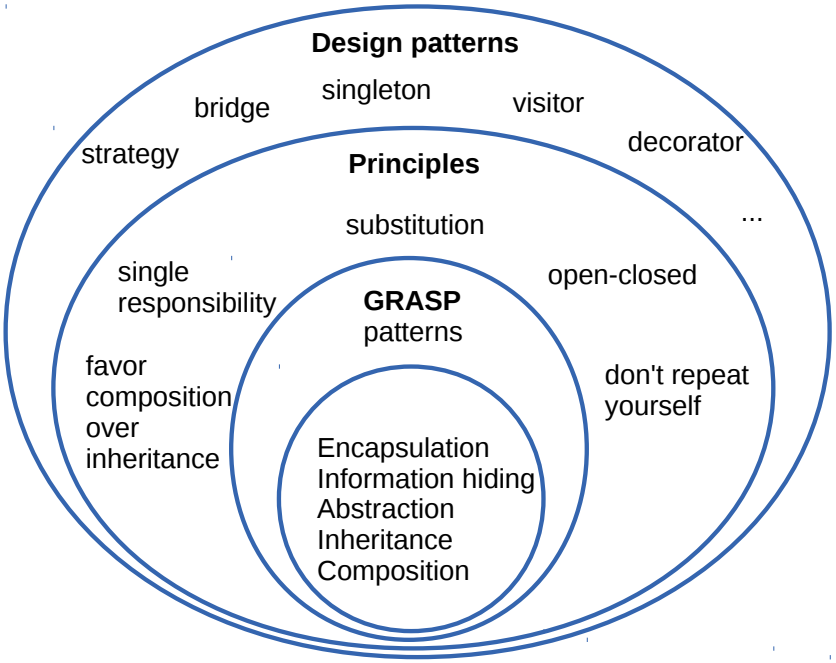
- Start from OO analysis, problem domain classes
- Add methods and define the messaging between them to fulfill the requirements
- New classes will appear, belonging to the software domain

Deciding what methods belong where, and how the objects should interact, is terribly important and anything but trivial.

Craig Larman

How to design object-oriented ?

There's no a methodology to get the best object-oriented design, but there are principles, patterns, heuristics.



Why follow principles and patterns ?

They are best practices found after decades of experience by many developers.

To build (more) change resistant designs.

Learn [to design objects] from the successes of others, not from their failures

Read *Symptoms of rotting design: rigidity, fragility, immobility, viscosity* in the article Design Principles and Design Patterns by Robert C. Martin, 2000 at `http://objectmentor.com/resources/articles/Principles_and_Patterns.pdf`.

Why follow principles and patterns ?

The design of many software applications begins as a vital image in the minds of its designers. At this stage it is clean, elegant, and compelling.

But then something begins to happen. The software starts to rot. At first it isnt so bad. An ugly wart here, a clumsy hack there. Yet, over time as the rotting continues, the ugly festering sores and boils accumulate until they dominate the design of the application. The program becomes a festering mass of code that the developers find increasingly hard to maintain.

Eventually, the sheer effort required to make even the simplest of changes to the application becomes so high that the engineers and front line managers cry for a redesign project.

Why follow principles and patterns ?

> *What kind of changes cause designs to rot? Changes that introduce new and unplanned for dependencies. Each of the four symptoms is caused by improper dependencies between the software modules. It is the dependency architecture that is degrading, and with it the ability of the software to be maintained.*
>
> *(...) the dependencies between modules in an application must be managed. This management consists of the creation of dependency firewalls. Across such firewalls, dependencies do not propagate.*
>
> *Object Oriented Design principles build such firewalls and manage module dependencies.*

—R.C. Martin, 2000

What's ahead

GRASP patterns :

- Expert
- Creator
- High Cohesion
- Low Coupling
- Controller
- Polymorphism
- Pure Fabrication

Principles :

- "Don't repeat yourself"
- Liskov substitution
- Single Responsibility
- Information hiding
- "Open for extension, closed for modification"
- "Program to an interface, not to an implementation"
- "Favor composition over inheritance"

Plus 23 design patterns.

GRASP patterns

Pattern

A named pair (problem, solution) plus advice on when/how to apply it and discussion of its trade-offs. In our case, solves an object–oriented generic design problem.

GRASP: General Responsibility Assignment Software Patterns

Principles of responsibility assignment, expressed as patterns.
Responsibility of an object/class is doing or knowing

- creating an object or doing a calculation
- initiating an action or controlling activities in other objects
- knowing about private, encapsulated data
- reference or contain other objects
- know how to get or calculate something

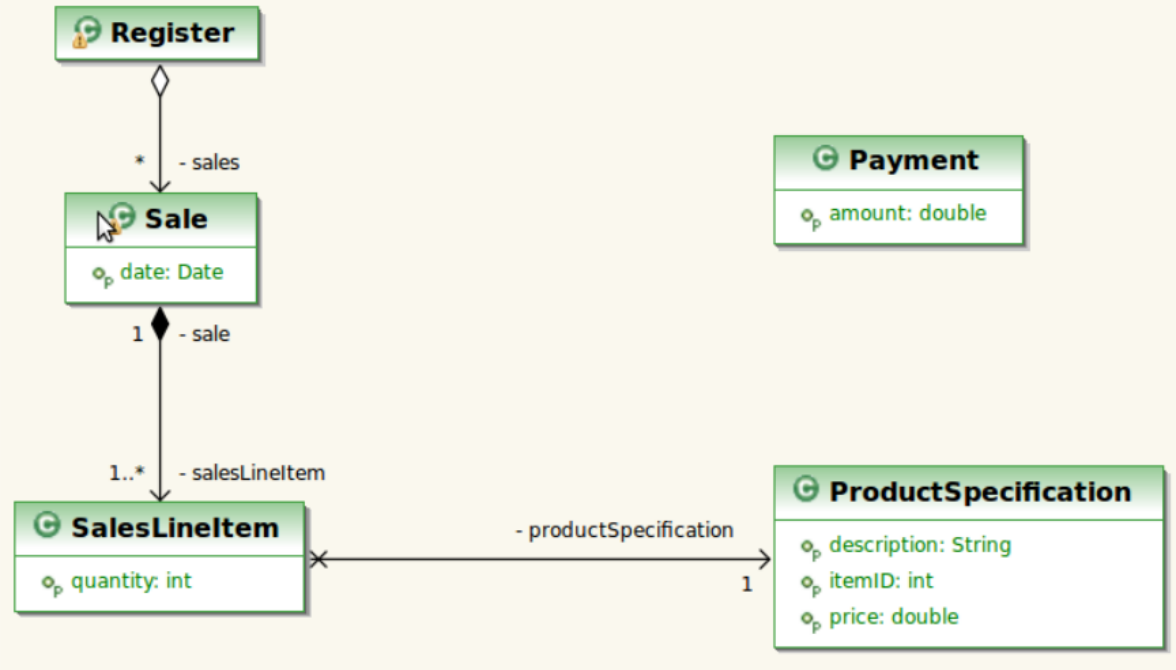
GRASP patterns: conducting example

Point of Sale :

- Application for a shop, restaurant, etc. that registers sales.
- Each sale is of one or more items of one or more product types, and happens at a certain date.
- A product has a specification including a description, unitary price and identifier.
- The application also registers payments (say, in cash) associated to sales.
- A payment is for a certain amount, equal or greater that the total of the sale.

GRASP patterns

Make a class design



Not really a design, because there are no methods! Designs must represent *how to do something*: register sales and payments.

GRASP patterns

Responsibilities are implemented through one or more methods, that may collaborate (send messages to = call methods of) other objects.

Recall : "*deciding what methods belong where, and how the objects should interact, is terribly important and anything but trivial*" .

GRASP patterns : Expert

Expert

Problem

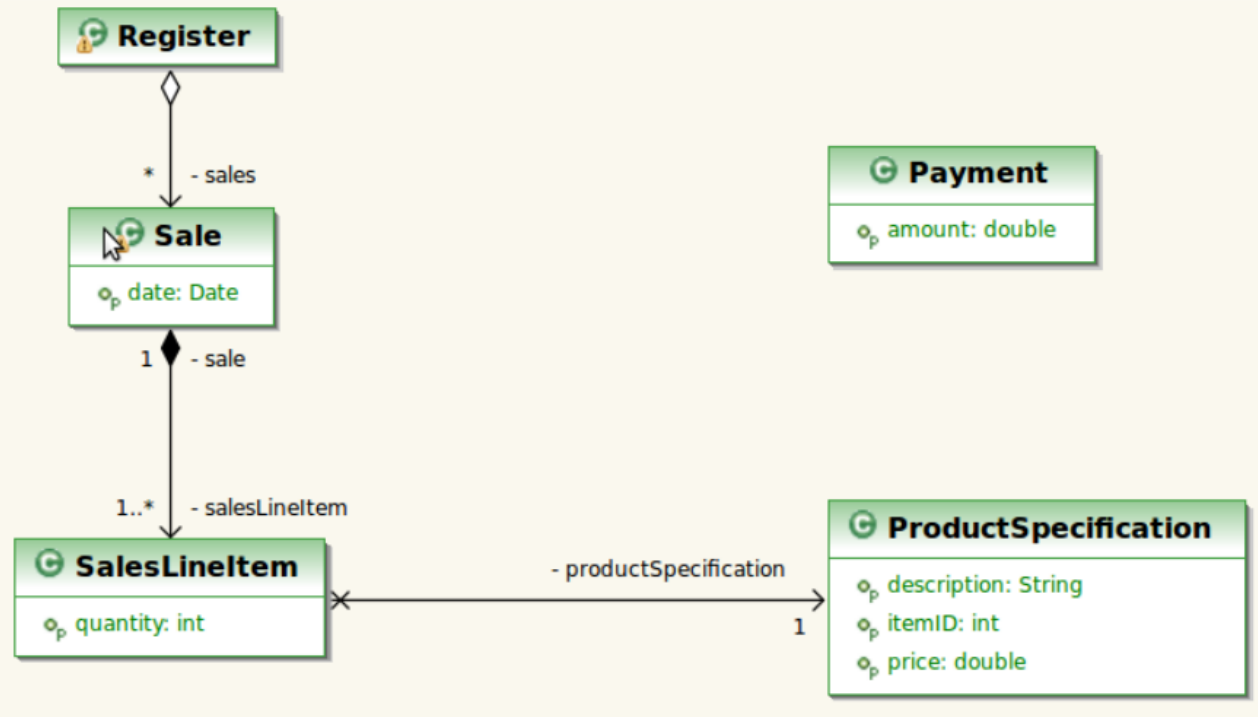
What is a general principle of assigning responsibilities to classes ?

Solution

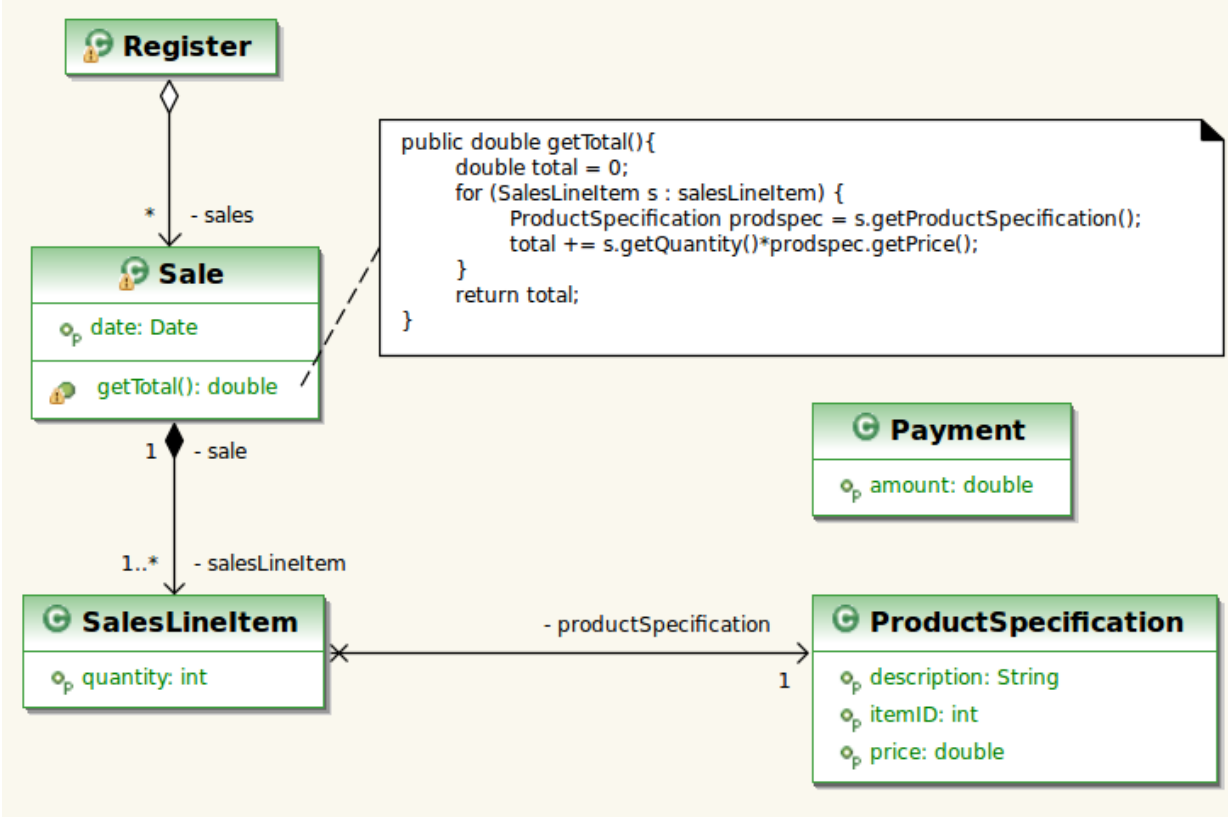
Assign responsibility to the information expert, the class that has the information necessary to fulfil the responsibility.

GRASP patterns : Expert

Who's responsible for knowing the grand total of a `Sale` ?



GRASP patterns : Expert



GRASP patterns : Expert

```java
public class Sale {
    //...
    public double getTotal(){
        double total = 0;
        for (SalesLineItem s : salesLineItem) {
            ProductSpecification prodspec
                = s.getProductSpecification();
            total += s.getQuantity()*prodspec.getPrice();
        }
        return total;
    }
}
```

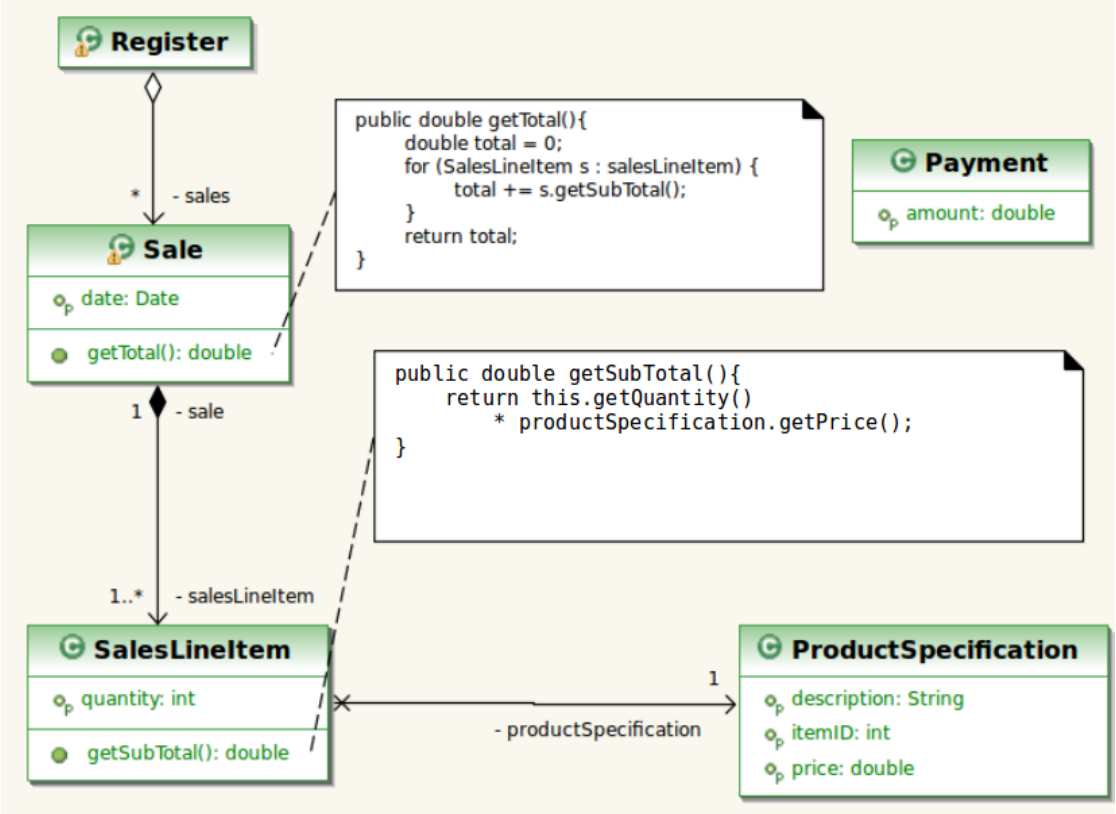
Is this right ?

GRASP patterns : Expert

```java
public class Sale {
    //...
    public double getTotal(){
        double total = 0;
        for (SalesLineItem s : salesLineItem) {
            total += s.getSubTotal();
        }
        return total;
    }
}

public class SalesLineItem {
    //...
    public double getSubTotal(){
        return this.getQuantity()
            * productSpecification.getPrice();
    }
}
```


GRASP patterns : Expert



GRASP patterns : Expert

Fulfilment of a responsibility may require information spread across different classes, each expert on its own data.

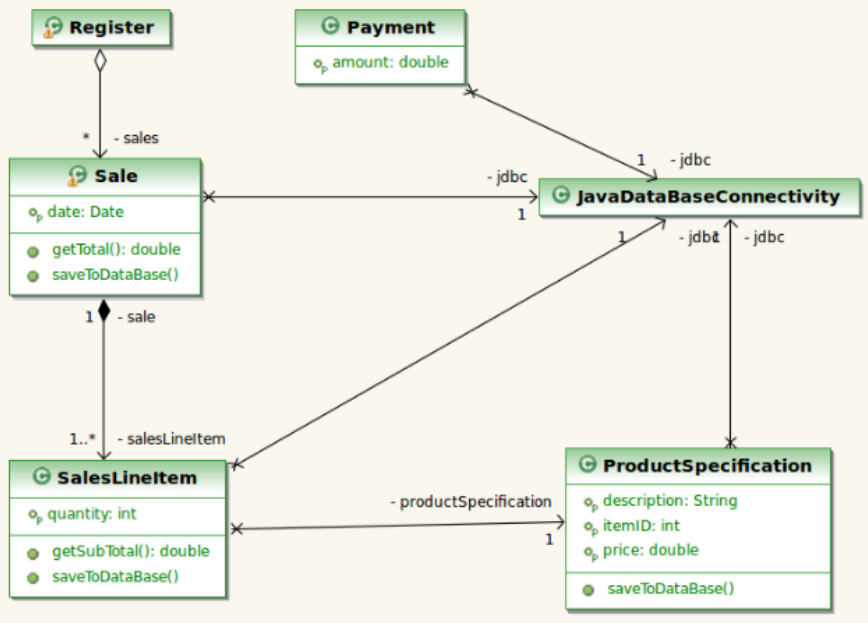
Real world analogy: workers in a business, bureaucracy, military. "Don't do anything you can push off to someone else".

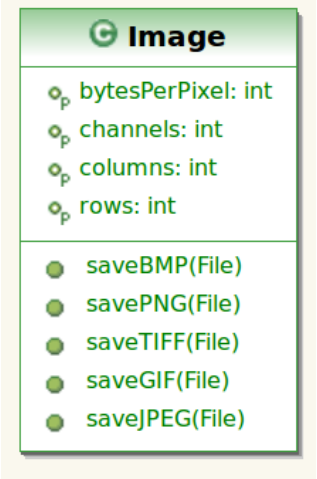
Benefits:

- Low coupling: `Sale` doesn't depend on `ProductSpecification`
- More cohesive classes
- Code easier to understand just by reading it

GRASP patterns : Expert

Not always convenient: problems of coupling and cohesion, separation of concerns.





GRASP patterns : Creator

Creator

Problem

Who should be responsible for creating an instance of some class ?

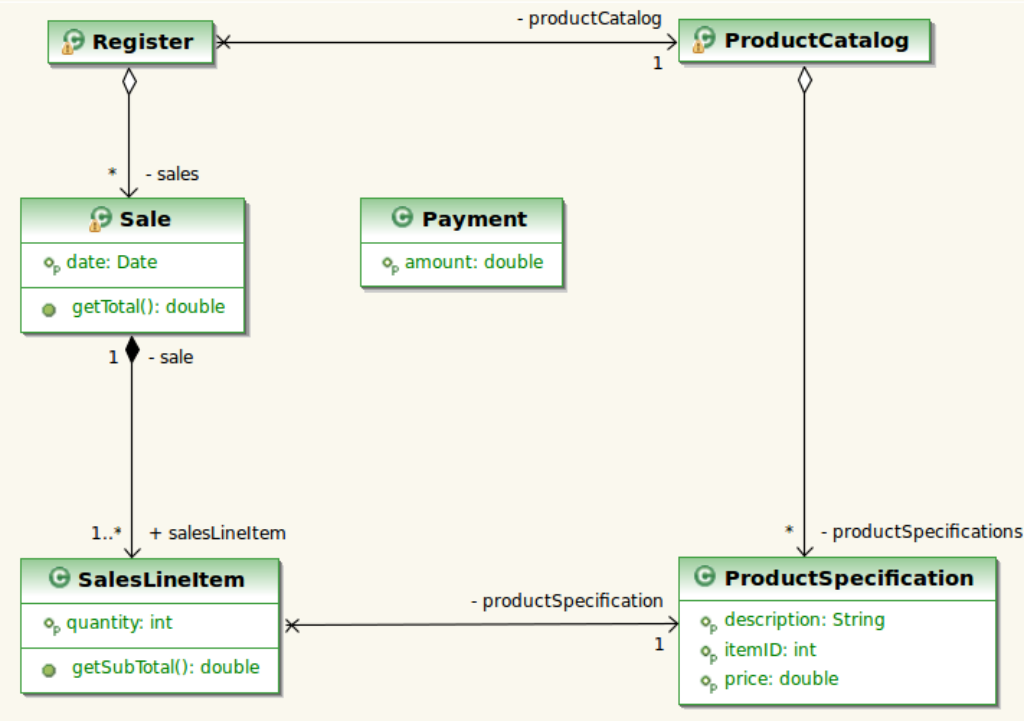
Solution

Assign B the responsibility of create instances of A if

- B aggregates or contains A objects
- B closely uses A objects
- B has the initializing data to be passed to the A constructor

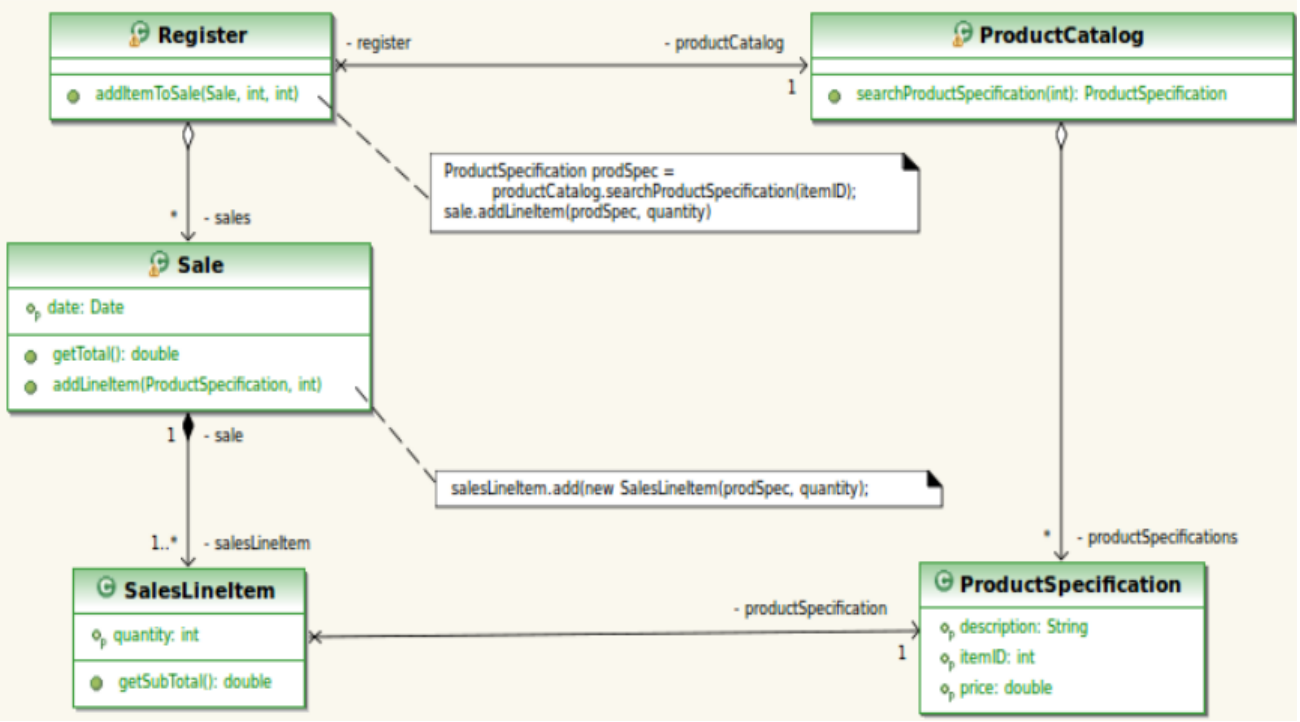
GRASP patterns : Creator

Who should be responsible for creating a `SalesLineItem` from an `itemID` and a `quantity` ?



GRASP patterns : Creator

Who should be responsible for creating a `SalesLineItem` from an `itemID` and a `quantity` ?



GRASP patterns : Creator

```
1  class Register{
2      ArrayList<Sale> sales = new ArrayList<Sale>();
3      //...
4      public void addItemToSale(Sale sale, int itemID,
5          int quantity) {
6          ProductSpecification prodSpec
7              = productcatalog.searchproductSpecification(itemID);
8          sale.addLineItem(prodSpec, quatity);
9      }
10 }
11
12 class Sale {
13     ArrayList<SalesLineItem> salesLineItem
14         = new ArrayList<SalesLineItem>();
15     //...
16     public void addLineItem(ProductSpecification prodSpec,
17         int quantity) {
18         salesLineItem.add(new SalesLineItem(prodSpec, quantity);
19     }
20 }
```

GRASP patterns : Creator

Benefits

- Low coupling: anyway, the created class A has to know (depend on) to the creator B

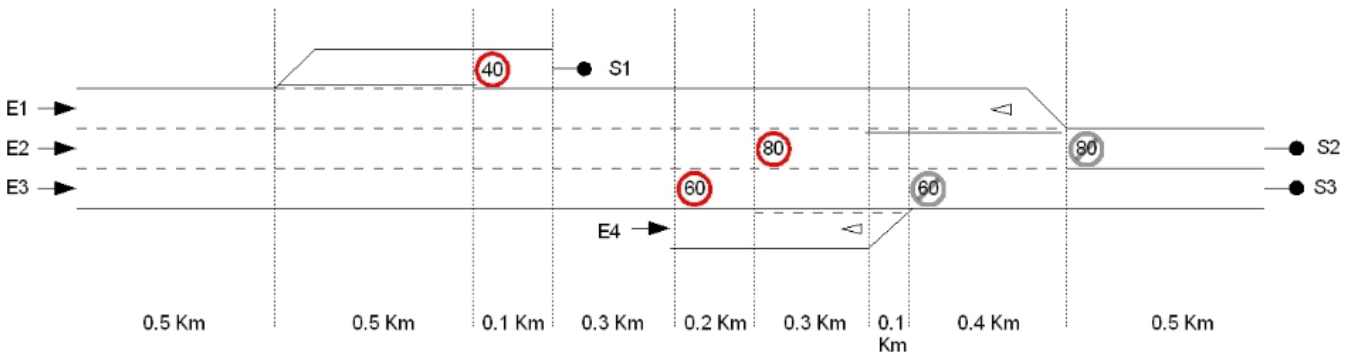
Exceptions

- When creation is more complex, like instantiating objects of different subclasses, or clone another object.

GRASP patterns : Creator

Traffic simulator application (old course assignment)

- roads with one or more lanes
- with entries, exits and different types of vehicles
- each type of vehicle differs in a range of possible values for length and maximum speed
- when a vehicle is created, these two parameters get a random value within their range

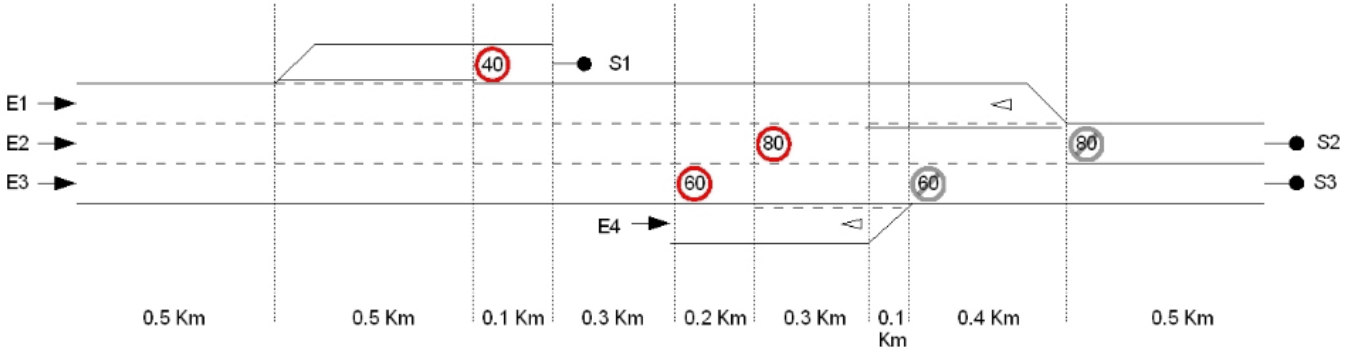


GRASP patterns : Creator

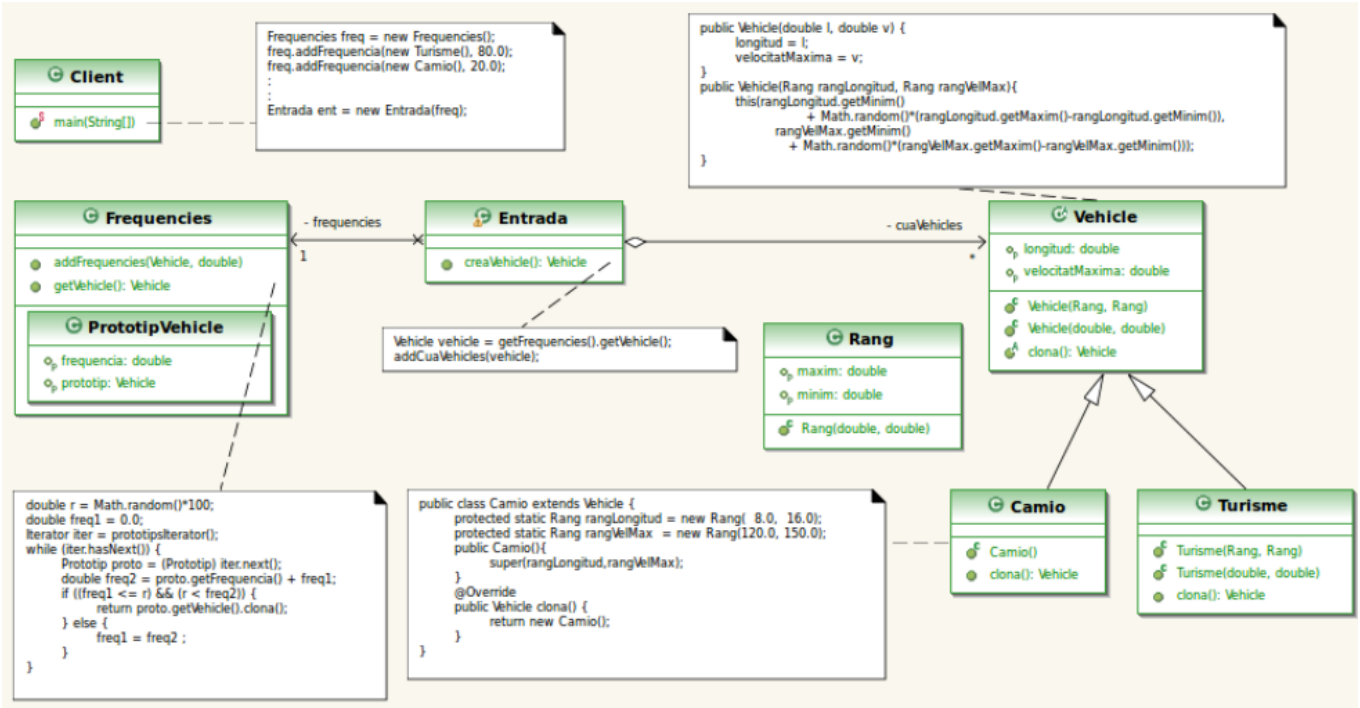
Entries E1, E2...throw vehicles to lanes, if traffic permits so.

Each entry must create each type of vehicle with its own programmed frequency, like

- E1 80% cars, 20% trucks
- E2 50% cars, 30% trucks, 20% long trucks
- ...



GRASP patterns : Creator



GRASP patterns : Creator

- Vehicles are created at `Frequencies.getVehicle()`, a factory method
- Creation means cloning a vehicle prototype from a list of prototypes.
- The prototype is chosen according to the lane frequencies table.

GRASP patterns : Creator

```
1  class Frequencies{
2      ArrayList<PrototipVehicle> prototips
3          = new ArrayList<PrototipVehicle>();
4      //...
5      public Vehicle getVehicle() {
6          double r = Math.random()*100.0;
7          double accumulatedFreq = 0.0;
8          iterator iter = prototips.Iterator();
9          while (iter.hasNext()) {
10             Prototip proto = (Prototip) iter.next();
11             double freq = proto.getFrequencia()
12                 + accumulatedFreq;
13             if ((r >= accumulatedFreq) && (r<freq)) {
14                 return proto.getVehicle().clona(); // no clone()!
15             } else {
16                 accumulatedFreq = freq;
17             }
18         }
19     }
```

GRASP patterns : Creator

- Cloning is easy in Java: implement `Cloneable` interface and call `clone()`:

```
1  public class Employee implements Cloneable{
2      private int empoyeeId;
3      private String employeeName;
4      private Department department;
5
6      public Employee(int id, String name, Department dept) {
7          this.empoyeeId = id;
8          this.employeeName = name;
9          this.department = dept;
10     }
11     @Override
12     protected Object clone() throws
       CloneNotSupportedException {
13         return super.clone();
14     }
15 }
```


GRASP patterns : Creator

- However, we want each "clone" of a prototype, to be different from it in its length and max. speed attributes.
- Therefore, we make a method `clona()` which calls the constructor to get a vehicle of the right type with random length and max. speed

```
1  public class Camio extends Vehicle {
2      protected static Rang rangLongitud = new Rang(8, 16);
3      protected static Rang rangVelMax = new Rang(120, 150);
4      public Camio(){
5          super(rangLongitud,rangVelMax);
6      }
7      @Override
8      public Vehicle clona() {
9          return new Camio();
10     }
11 }
```

(by the way, why static ?)

GRASP patterns : Low coupling

Coupling measures of how strongly an class is connected, depends, relies on or has knowledge of objects of other classes.

Classes with strong coupling

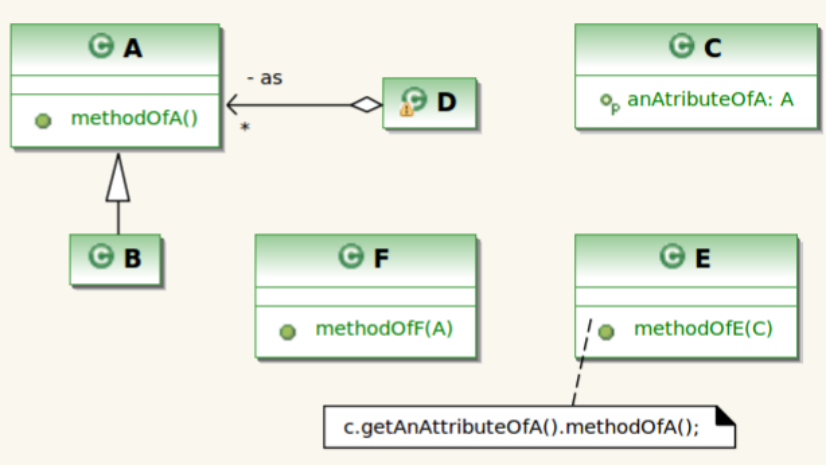
- suffer from changes in related classes
- are harder to understand and maintain
- are more difficult to reuse

But coupling is necessary if we want classes to exchange messages!

The problem is too much of it and/or too unstable classes.

GRASP patterns : Low coupling

Types of coupling



GRASP patterns : Low coupling

Problem

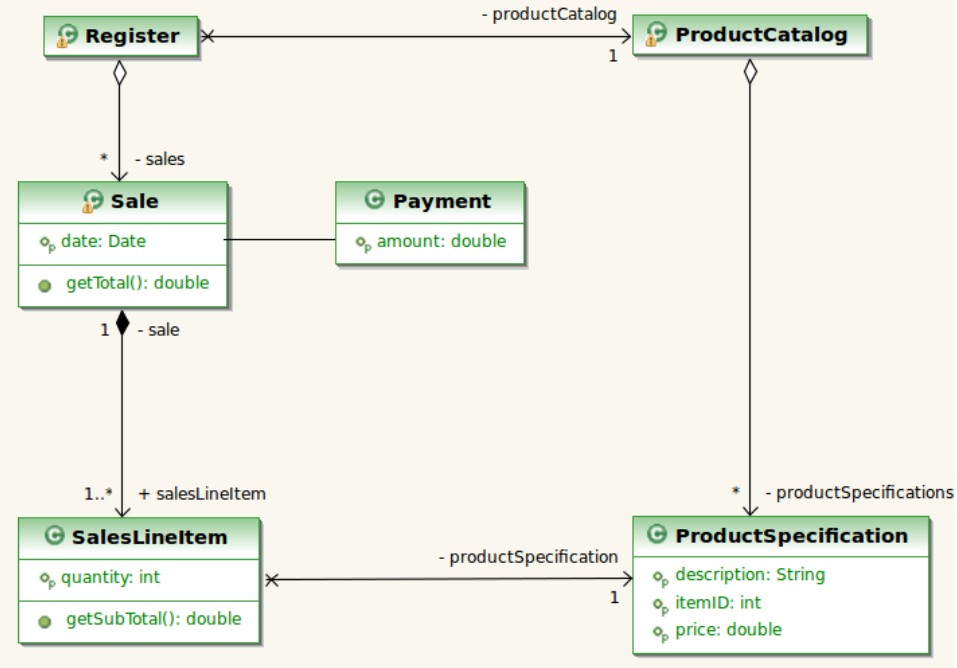
How to support low dependency, low change impact and increase reuse ?

Solution

Assign responsibilities so that coupling remains low. Try to avoid one class to have to know about many others.

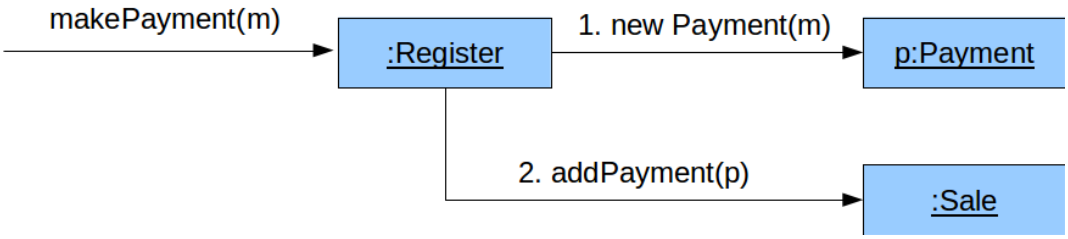
GRASP patterns : Low coupling

Who should be responsible for creating a Payment for a Sale from a handed amount ?



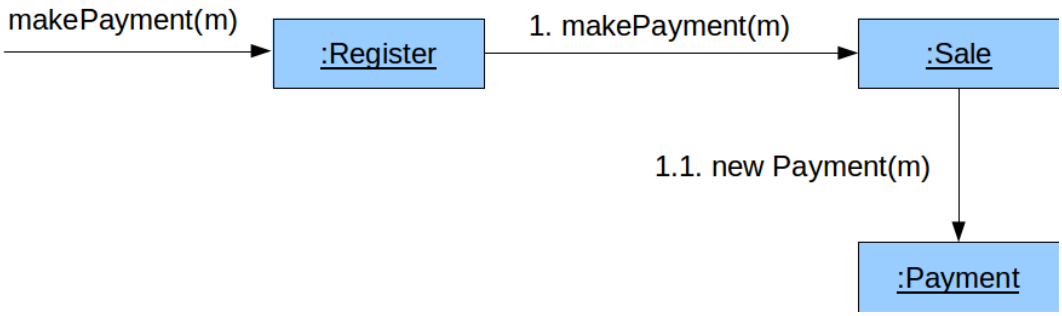
GRASP patterns : Low coupling

In the problem domain, `Register` knows about the handed amount `m`. According to Creator,



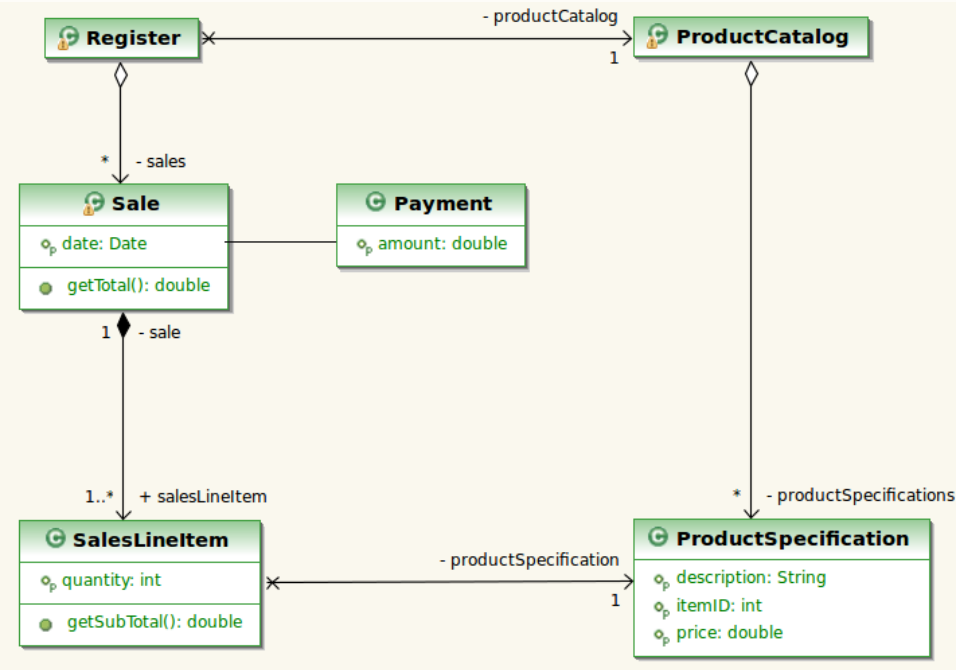
GRASP patterns : Low coupling

Delegate to `Sale` : `Register` is not coupled anymore to `Payment`. `Sales` is anyway in both cases (must be), because a `Sale` must know its payment.



GRASP patterns : Low coupling

Who should own the method `getBalance()` that computes payment amount - total of sale ?



GRASP patterns : Low coupling

Discussion

- extreme low coupling: few big, complex, non-cohesive classes that do everything, plus many objects acting as data holders
- it is not coupling per se the problem but coupling to unstable classes

GRASP patterns : High cohesion

Cohesion is a measure of how strongly related the responsibilities of a class are. A class with low cohesion does many unrelated things or too much work.

Problems: hard to

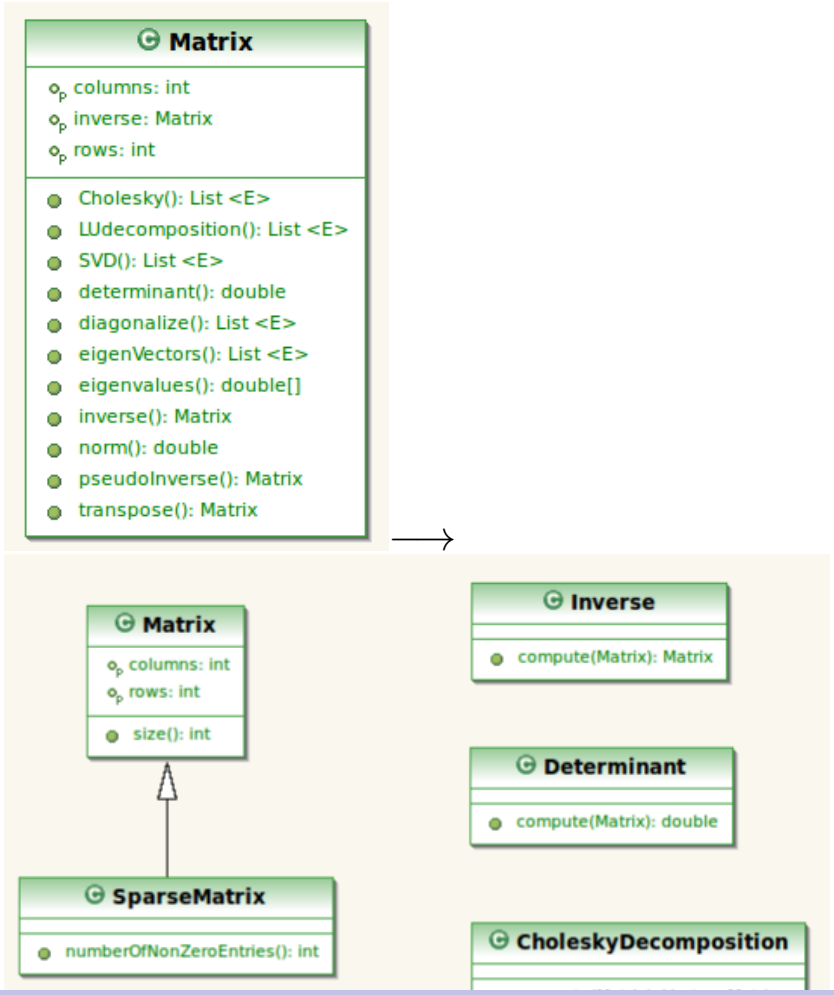
- understand
- maintain
- reuse

GRASP patterns : High cohesion

Problem
How to keep classes focused, understandable and manageable ?

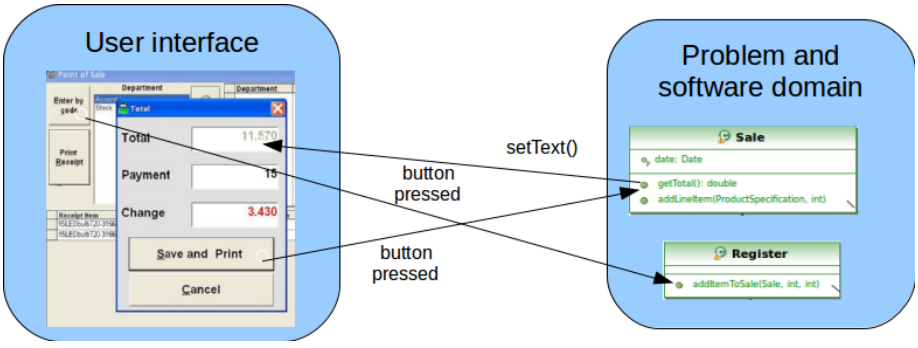
Solution Assign responsibilities so that cohesion remains high. Try to avoid classes to do too much or too different things.

GRASP patterns : High cohesion



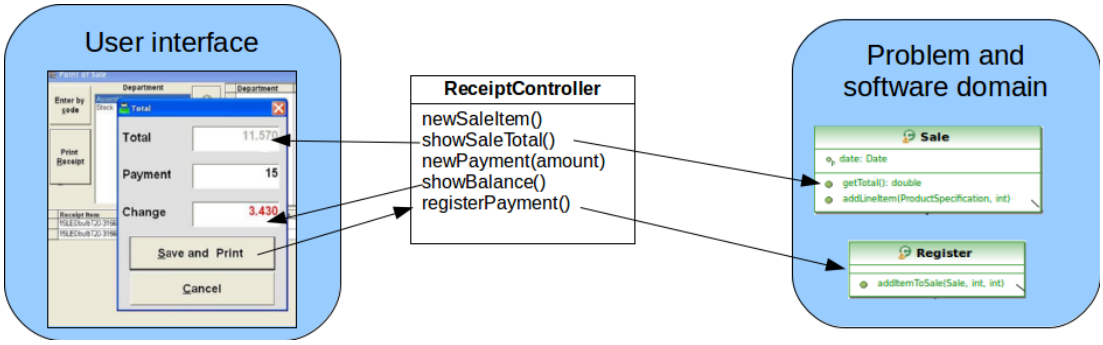
GRASP patterns : Controller

Problem Who should be responsible for handling an input event, which object/s beyond the UI layer receives interaction ?



GRASP patterns : Controller

Solution A controller class representing the whole system or a particular use case



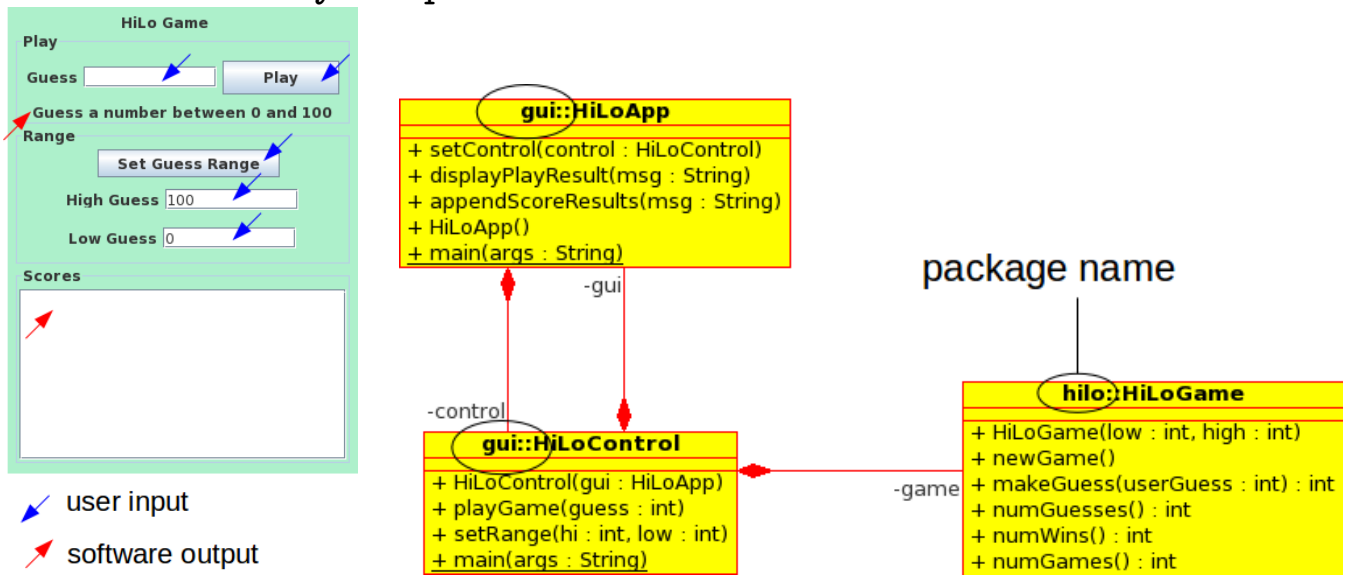
GRASP patterns : Controller

Benefits

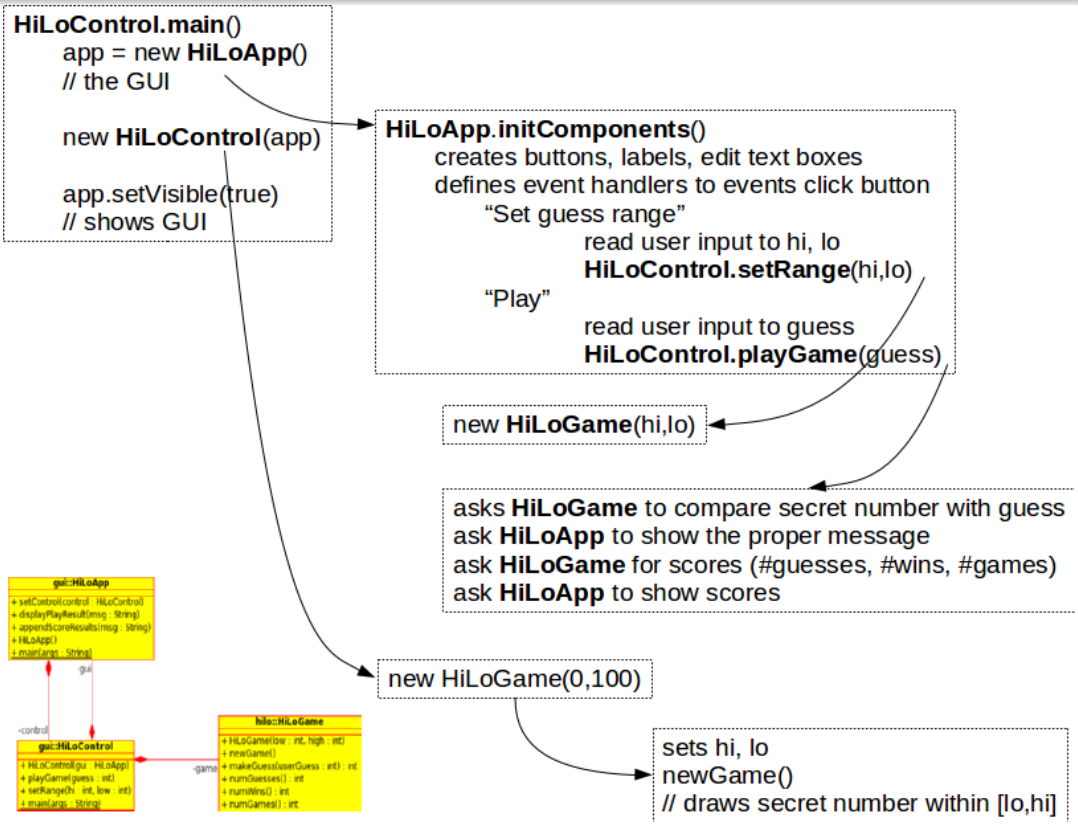
- either the UI classes or the problem/software domain classes can change without affecting the other side.
- controller is a simple class that mediates between the UI and problem domain classes, just forwards
  - event handling requests
  - output requests

GRASP patterns : Controller

HiLo game: read source code for Java + Swing at `GRASP_Rod_Byrne.pdf`



GRASP patterns : Controller

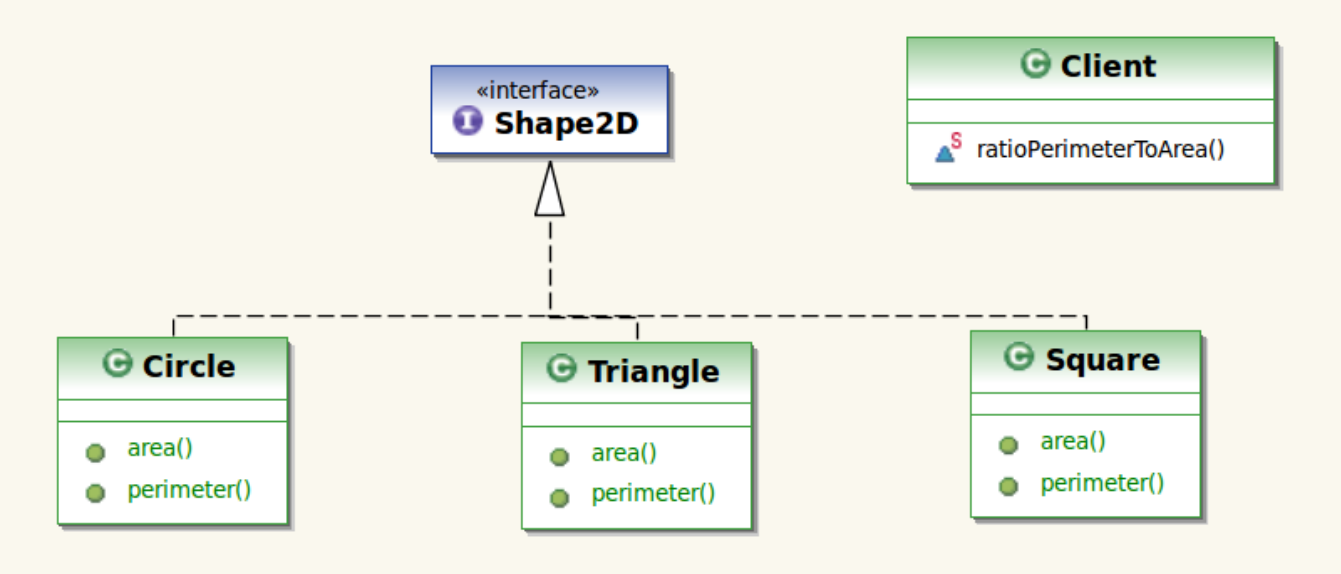


GRASP patterns : Polymorphism

Polymorphic methods: giving the same name to (different) services in different classes. Services are implemented by methods.

Problem
How to handle behavior based on type (i.e. class) but not with an if-then-else or switch statement involving the class name or a tag attribute ?

GRASP patterns : Polymorphism



GRASP patterns : Polymorphism

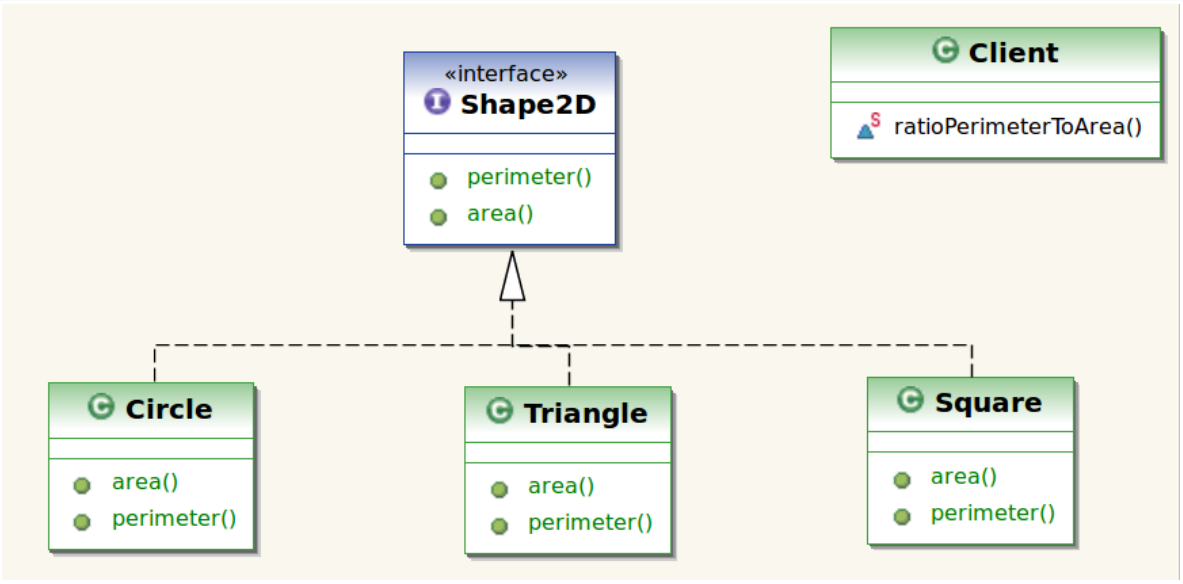
```
1
2  static double ratioPerimeterToArea(Shape2D s) {
3      double ratio = 0.0;
4      if (s instanceof Triangle) {
5      // or String name = s.getClass().getName();
6      //    if (name=="Triangle") {
7          Triangle t = (Triangle) s;
8          ratio = t.perimeter()/t.area();
9      } else if (s instance of Circle) {
10         Circle c = (Circle) s;
11         ratio = c.perimeter()/c.area();
12     } else if (s instanceof Square) {
13         Square sq = (Square) s;
14         ratio = sq.perimeter()/sq.area();
15     }
16     return ratio;
17 }
```


GRASP patterns : Polymorphism

Problems:

- new shapes and class name changes require the modification of this kind of code
- normally it appears at several places
- avoidable coupling between `Client` and shape subclasses

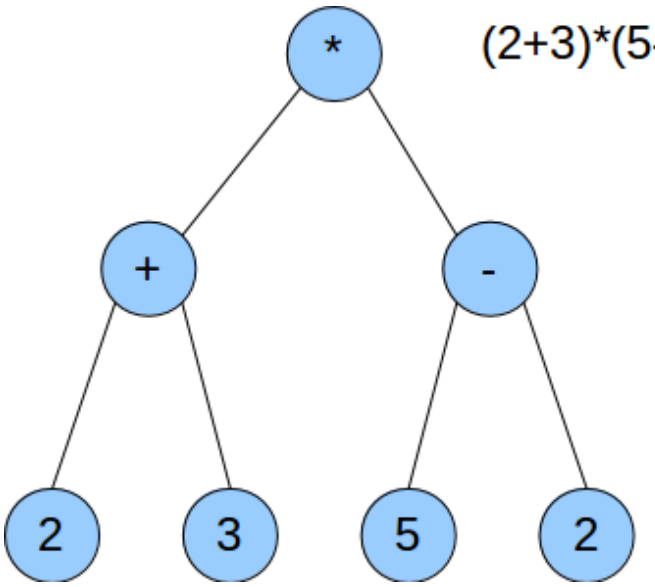
GRASP patterns : Polymorphism




```
1 static double ratioPerimeterToArea(Shape2D s) {
2     return s.perimeter()/s.area();
3 }
```


GRASP patterns : Polymorphism

Write a class `Expr` to support the evaluation of arithmetic expressions represented as trees:



GRASP patterns : Polymorphism

```java
public class NonPolyExpr {
    public static final int CONST_EXPR = 1;
    public static final int PLUS_EXPR = 2;
    public static final int TIMES_EXPR = 3;
    private int type;
    private double value;
    private NonPolyExpr left=null, right=null;

    public NonPolyExpr( int type, double value ) {
        this.type = type;
        this.value = value;
    }
    public NonPolyExpr( int type, NonPolyExpr left,
        NonPolyExpr right) {
        this.type = type;
        this.left = left;
        this.right = right;
    }
```


GRASP patterns : Polymorphism

```java
    public double evaluate() {
        switch( type ) {
        case CONST_EXPR:
            return value;
        case PLUS_EXPR:
            return left.evaluate() + right.evaluate();
        case TIMES_EXPR:
            return left.evaluate() * right.evaluate();
        }
        return java.lang.Double.NaN; // what is this?
    }
}

NonPolyExpr a = new NonPolyExpr(NonPolyExpr.CONST_EXPR, 3.0);
NonPolyExpr b = new NonPolyExpr(NonPolyExpr.CONST_EXPR, 4.0);
NonPolyExpr c = new NonPolyExpr(NonPolyExpr.PLUS_EXPR,a,b);
System.out.print( c.evaluate() );
```

GRASP patterns : Polymorphism

Problems:

- `evaluate()` is not cohesive as its polymorphic version
- `NonPolyExpr` cannot be easily extended: unary (-) and ternary operators (`x ? y : z`)
- coupling: clients must know about expression types `CONST_EXPR`, `PLUS_EXPR`, `TIMES_EXPR`
- it is legal to write non-sense code like

```
1 // a, b are NonPolyExpr objects
2 NonPolyExpr e = new NonPolyExpr(NonPolyExpr.CONST_EXPR,a,b);
3 System.out.print( e.evaluate() );
```

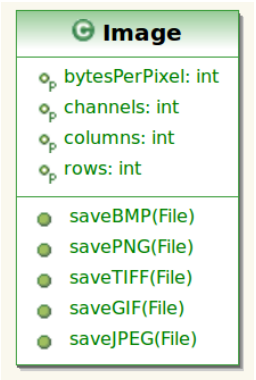
Exercise: write the polymorphic version.

GRASP patterns : Pure fabrication

Problem
What object should have a responsibility when no class of the problem domain may take it without violating high cohesion, low coupling ?

Not all responsibilities fit into domain classes, like persistence, network communications, user interaction etc.

Remember what was the problem here

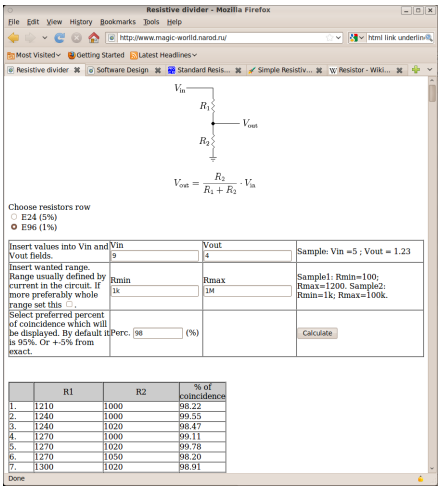


GRASP patterns : Pure fabrication

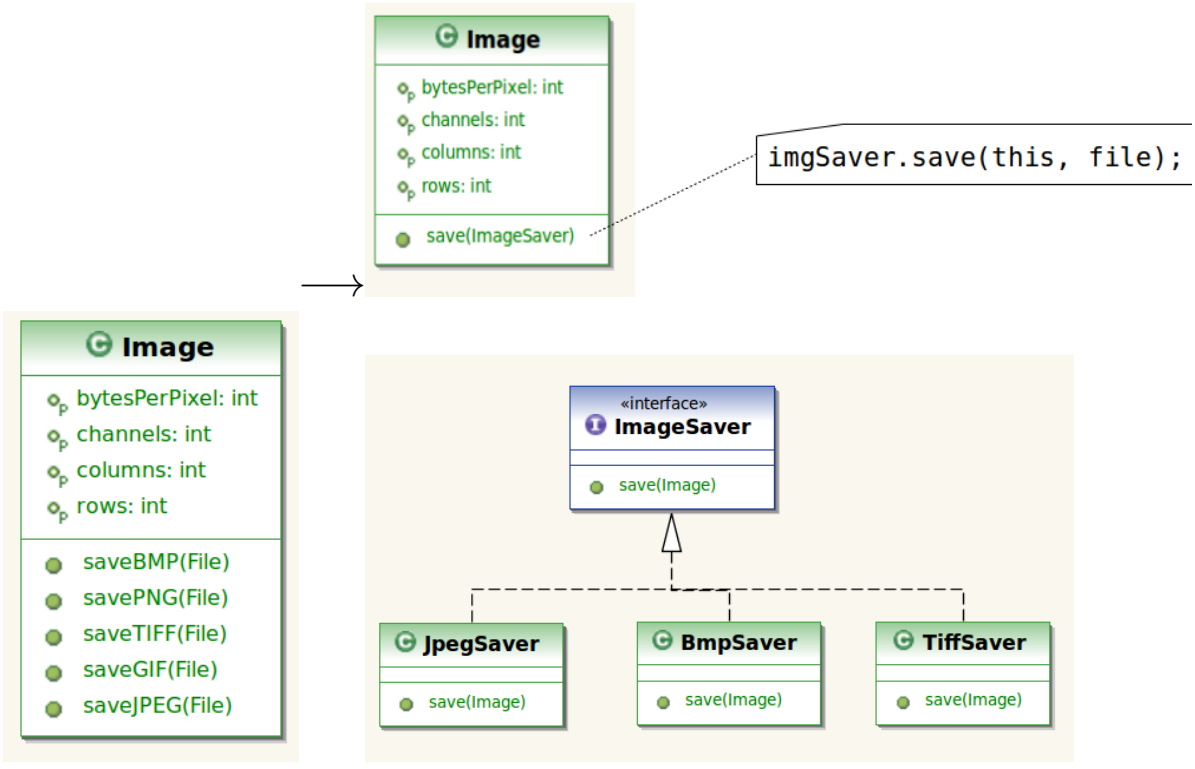
Solution

Assign a cohesive set of responsibilities to an artificial/convenience class not representing a concept of the problem domain.

Exercise voltage divider



GRASP patterns : Pure fabrication



Later we will call this the *strategy pattern*.

GRASP patterns : Summary

What should you know

- What's design, object design and how to do it
- What are responsibilities
- Problem + solution of Expert, Creator GRASP patterns
- What's cohesion and coupling
- What's a controller class
- Why `instanceof` is evil and how to avoid it
- What's a pure fabrication class and when to create them

Next: object-oriented design general principles