# GRASP Design Patterns

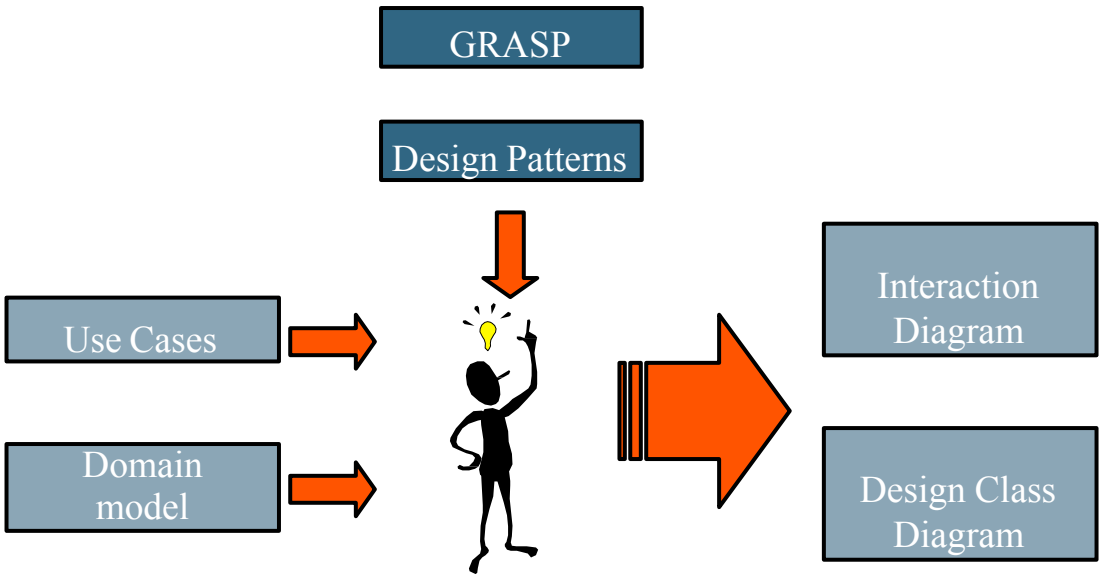
# Designing Objects with Responsibilities

Mona Demaidi

## Lecture learning outcome

- Learn about design patterns
- Learn how to apply GRASP patterns
- UML is just notation; now you need to learn how to make effective use of the notation
- UML modeling is an art, guided by principles

# Design: workflow




# Design patterns in architecture

- A *pattern* is a recurring solution to a standard problem, in a context.
- Christopher Alexander, professor of architecture…
  - *Why is what a prof of architecture says relevant to software?*
  - "A pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."

# Patterns in engineering

- *How do other engineers find and use patterns?*
  - Mature engineering disciplines have handbooks describing successful solutions to known problems
  - Automobile designers don't design cars from scratch using the laws of physics
  - Instead, they reuse standard designs with successful track records, learning from experience
  - *Should software engineers make use of patterns? Why?*
- Developing software from scratch is also expensive
  - Patterns support reuse of software architecture design

# Definitions and names

- Alexander: "A *pattern* is a recurring solution to a standard problem, in a context."
- Larman: "In OO design, a *pattern* is a named description of a problem and solution that can be applied in new contexts; ideally, a pattern advises us on how to apply the solution in varying circumstances and considers the forces and trade-offs."
- How is Larman's definition similar to Alexander's?
- How are these definitions significantly different?

# Naming Patterns—important!

- Patterns have suggestive names:
  - Arched Columns Pattern, Easy Toddler Dress Pattern, etc.
- Why is naming a pattern or principle helpful?
  - It supports chunking and incorporating that concept into our understanding and memory
  - It facilitates communication

Star and Plume Quilt

## Patterns/Principles aid Communication

***Fred: "Where do you think we should place the responsibility for creating a SalesLineItem? I think a Factory."***

***Wilma: "By Creator, I think Sale will be suitable."***

***Fred: "Oh, right - I agree."***

## Well-known Pattern Families

- GRASP:
  - General Responsibility Assignment Software Patterns
    - (or Principles)
  - 9 Patterns
- GoF: Design Patterns: Elements of Reusable Object-Oriented Software
  - GoF:
    - Erich Gamma
    - Richard Helm
    - Ralph Johnson
    - John Vlissides
  - 23 patterns

## Pattern Example

- Pattern name: Information Expert
- Problem: What is a basic principle by which to assign responsibilities to objects?
- Solution Assign a responsibility to (one of?) the class that has the information needed to fulfill it

## GRASP Patterns / Principles

- The GRASP patterns are a *learning aid* to
  - help one understand essential object design
  - apply design reasoning in a methodical, rational, explainable way.
- This approach to understanding and using design principles is based on patterns of assigning **responsibilities**

## GRASP - Responsibilities

- Responsibilities are related to the obligations of an object in terms of its behavior.
- Two types of responsibilities:
  - knowing
  - doing
- Doing responsibilities of an object include:
  - doing something itself, such as creating an object or doing a calculation
  - initiating action in other objects
  - controlling and coordinating activities in other objects
- Knowing responsibilities of an object include:
  - knowing about private encapsulated data
  - knowing about related objects
  - knowing about things it can derive or calculate

## GRASP

- Name chosen to suggest the importance of **grasp**ing fundamental principles to successfully design object-oriented software
- Acronym for **G**eneral **R**esponsibility **A**ssignment **S**oftware **P**atterns
- Describe fundamental principles of object design and responsibility
- General principles, may be overruled by others

## GRASP Patterns

- Low Coupling
  - Support low dependency and increased reuse
- High Cohesion
  - How to keep complexity manageable?
- Information Expert
  - Who, in the general case, is responsible?
- Creator
  - Who creates?
- Controller
  - Who handles a system event?
- Polymorphism
  - Who, when behavior varies by type?
- Pure Fabrication
  - Who, when you are desperate, and do not want to violate High Cohesion and Low Coupling?
- Indirection
  - Who, to avoid direct coupling?
- Protected variations
  - Who, to avoid knowing about the structure of indirect objects?

# Low Coupling Principle

Problem:

How to increase reuse and decrease the impact of change.

Solution:

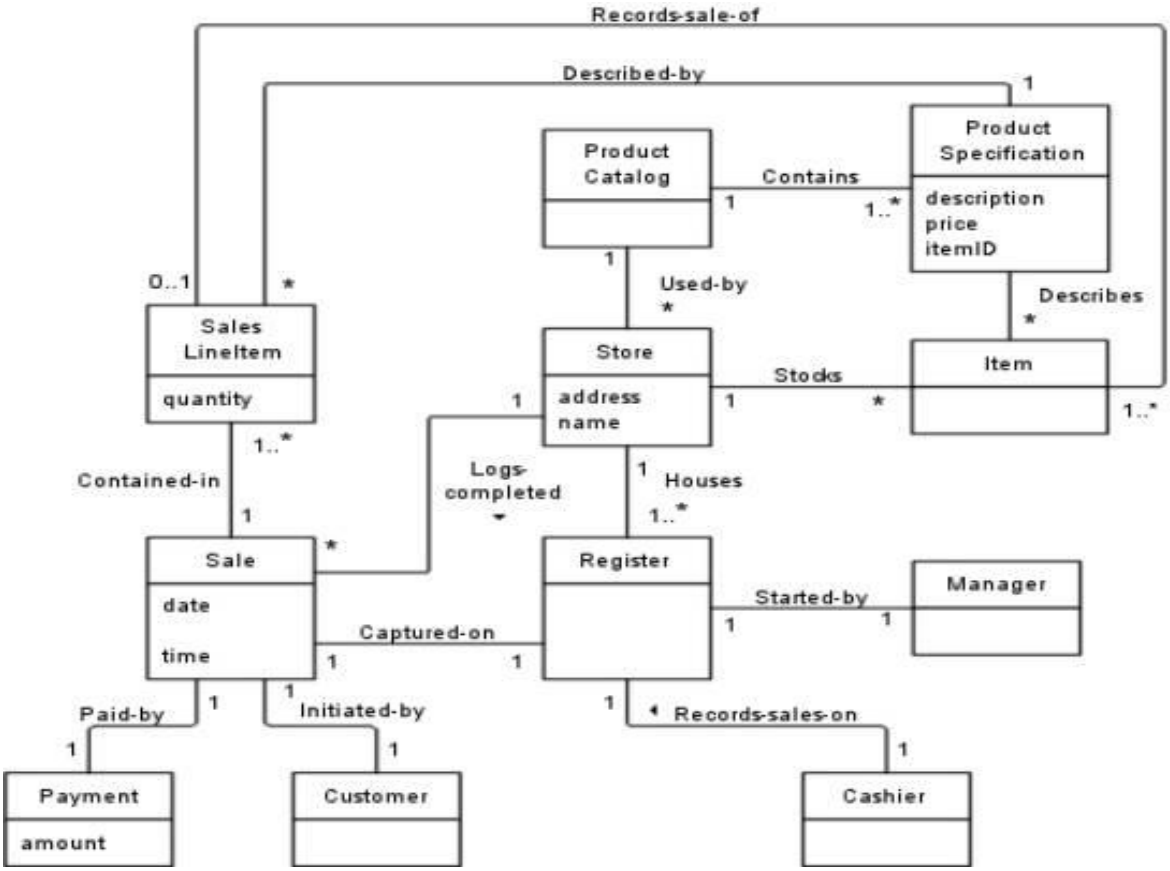
Assign responsibilities to minimize coupling.

Use this principle when evaluating alternatives

## Example – Point Of Sale system

- A computerised application used to record sales and handle payments.
- It is typically used in a retail store.
- It includes hardware components such as a computer and bar code scanner, and software to run the system.
- It interfaces to various service applications, such as a third-party tax calculator.

## Example – Point Of Sale system

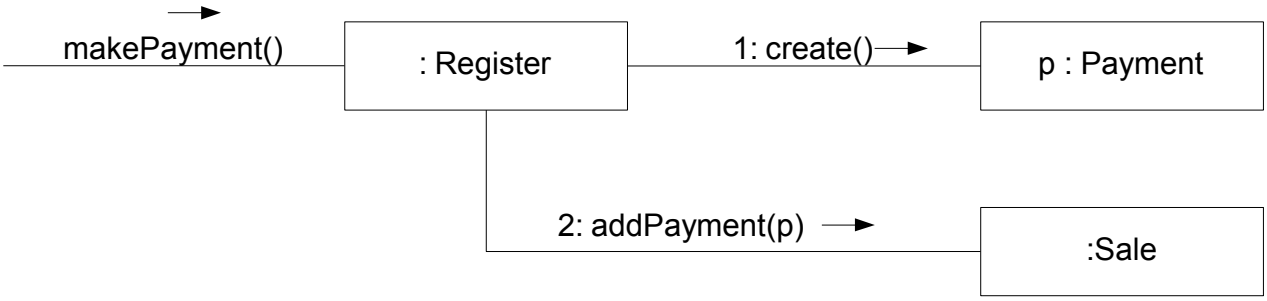



## Example – Point Of Sale system

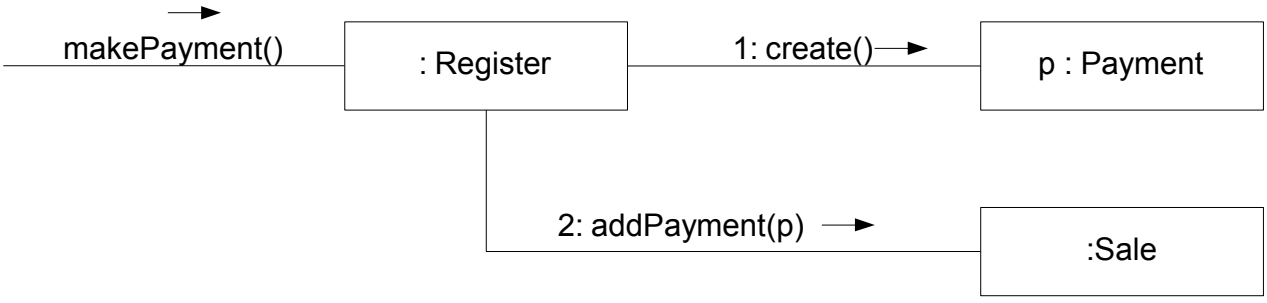
- Create a Payment and associate it with the Sale.

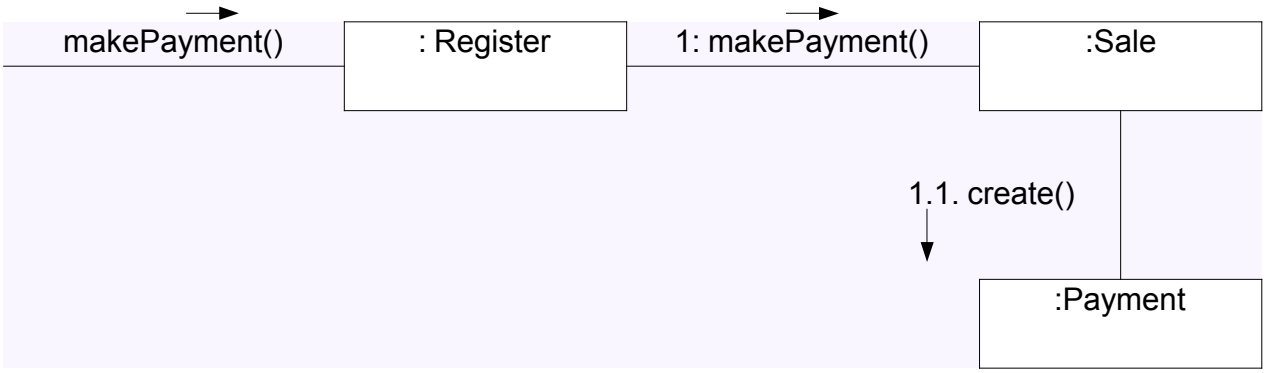
| Register | Sale | Payment |
|---|---|---|

# Example – Point Of Sale system

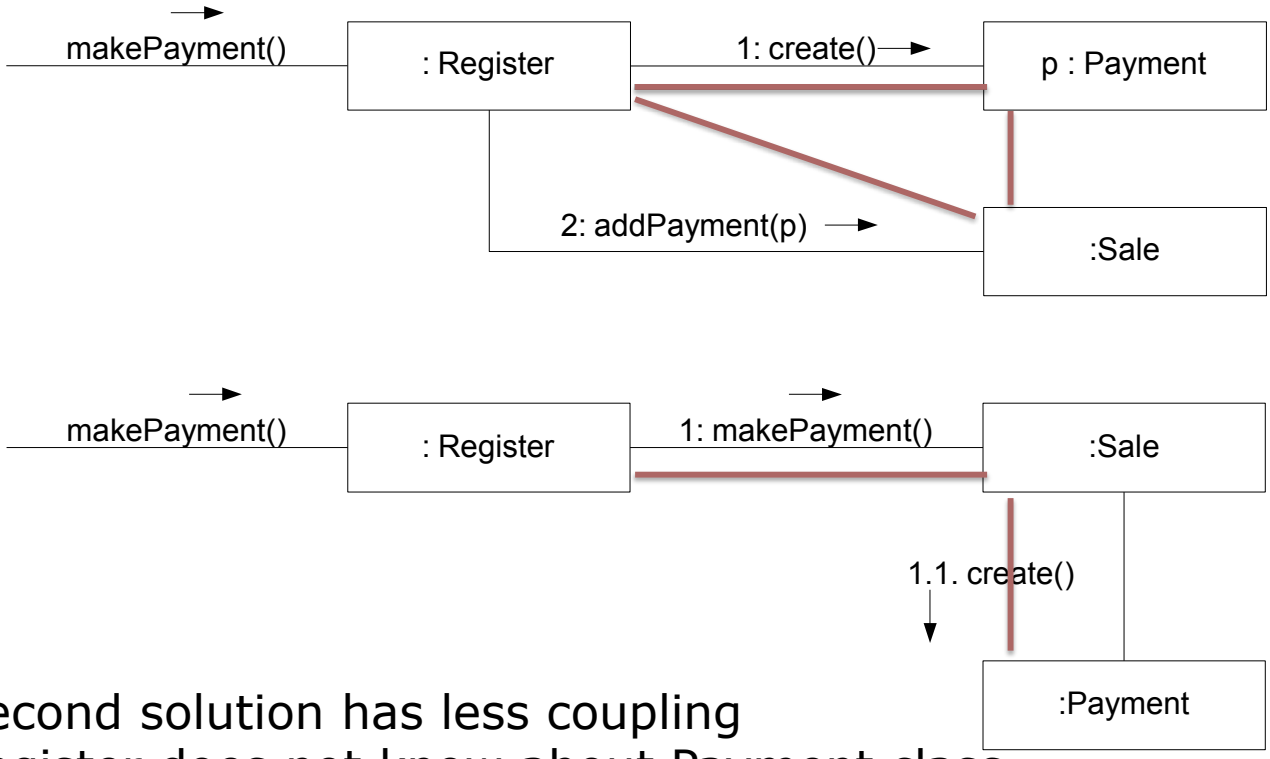



# Example – Point Of Sale system



# Coupling




Second solution has less coupling
Register does not know about Payment class

# Why High Coupling is undesirable

- Coupling is a measure of how strongly one element is connected to, has knowledge of, or relies on other elements.
- An element with low (or weak) coupling is not dependent on too many other elements (classes, subsystems, …)
  - "too many" is context-dependent
- A class with high (or strong) coupling relies on many other classes.
  - Changes in related classes force local changes.
  - Such classes are harder to understand in isolation.
  - They are harder to reuse because its use requires the additional presence of the classes on which it is dependent.

## Low Coupling

- Benefits of making classes independent of other classes
  - changes are localised
  - easier to understand code
  - easier to reuse code

## Common Forms of Coupling in OO Languages

- X has attribute of type Y
- X uses a service of Y
- X has method referencing Y (param, local variable)
- X inherits from Y (direct or indirect)
- X implements interface Y
- (X does not compile without Y)

## Low Coupling: Discussion

- Low Coupling is a principle to keep in mind during all design decisions
- It is an underlying goal to continually consider.
- It is an evaluative principle that a designer applies while evaluating all design decisions.
- Low Coupling supports the design of classes that are more independent
  - reduces the impact of change.
- Can't be considered in isolation from other patterns such as Expert and High Cohesion
- Needs to be included as one of several design principles that influence a choice in assigning a responsibility.

## Low Coupling: Discussion

- Subclassing produces a particularly problematic form of high coupling
  - Dependence on implementation details of superclass
- Learn to draw the line (experience)
  - do not pursue low coupling in the extreme
    - Bloated and complex active objects doing all the work
    - lots of passive objects that act as simple data repositories
  - OO Systems are built from connected collaborating objects
    - Coupling with standardized libraries is NOT a problem
    - Coupling with unstable elements IS a problem

## High Cohesion Principle

Problem:

How to keep complexity manageable.

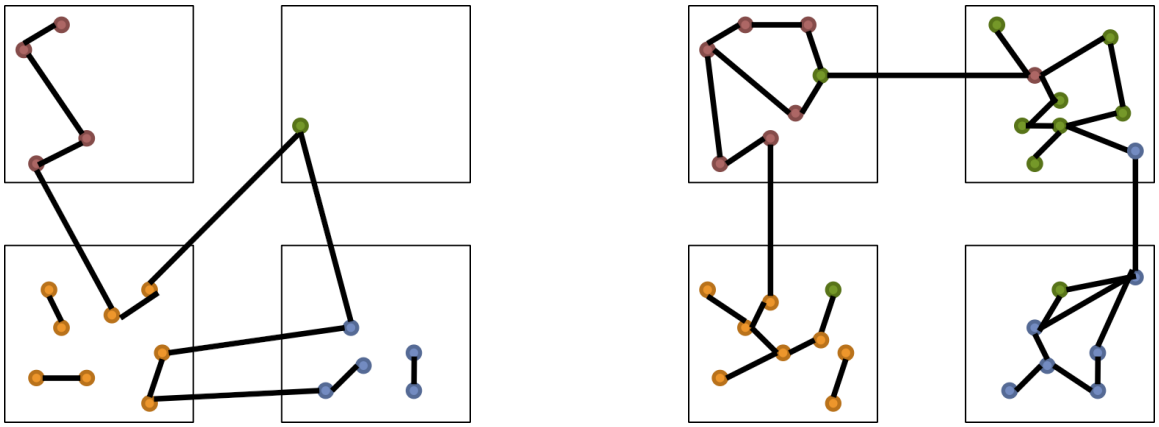
Solution:

Assign responsibilities so that cohesion remains high.

Cohesion is a measure of how strongly related and focused the responsibilities of an element are.

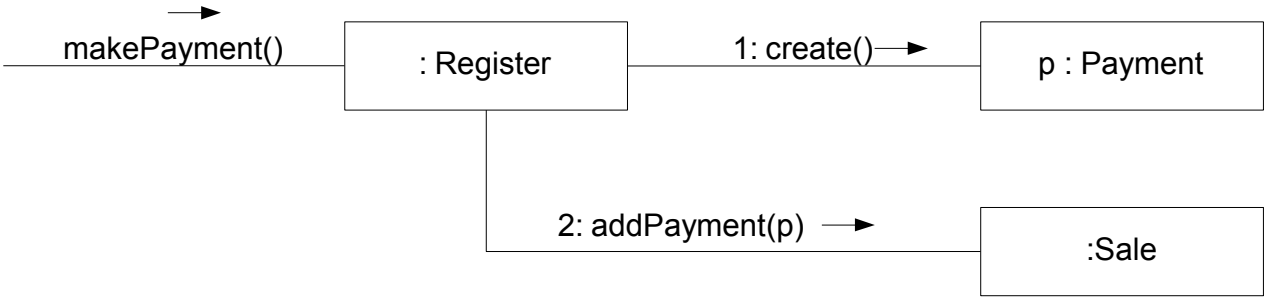
An element with highly related responsibilities, and which does not do a tremendous amount of work, has high cohesion

## High cohesion



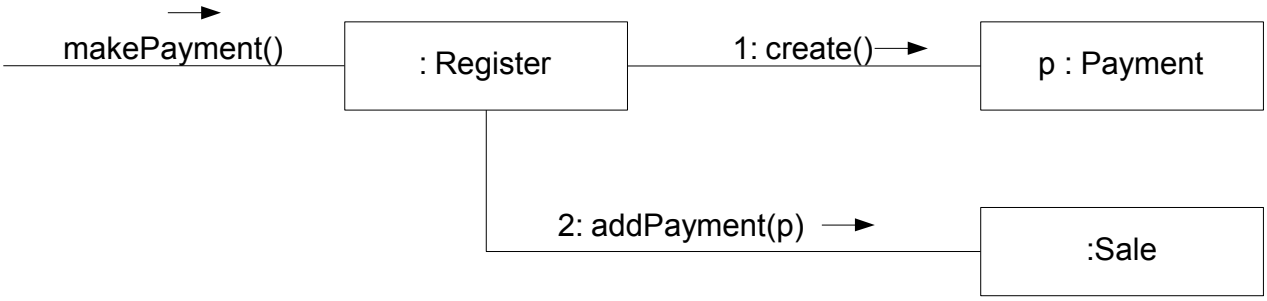
- Classes are easier to maintain
- Easier to understand
- Often support low coupling
- Supports reuse because of fine grained responsibility

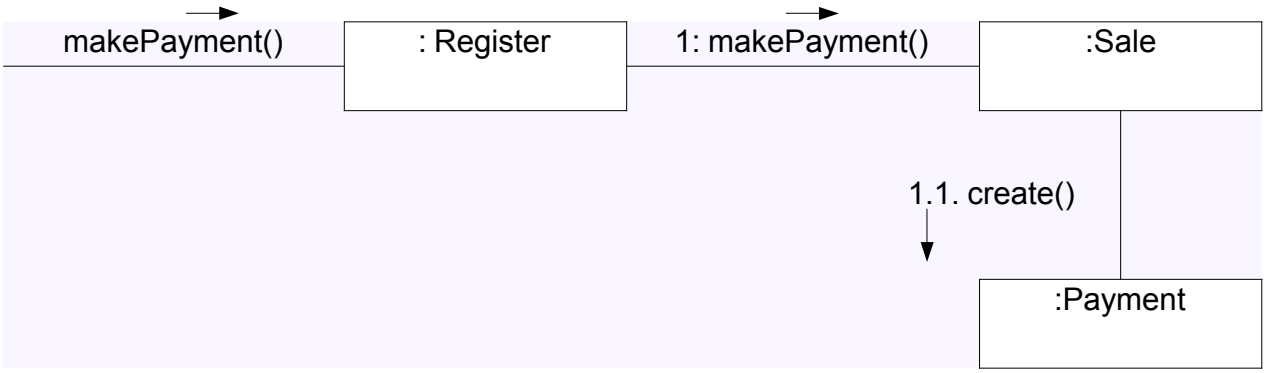
# Example




- Cohesion: Object should have strongly related operations or responsibilities
- Reduce fragmentation of responsibilities (complete set of responsibility)
- (except for cohension), looks OK if *makePayment* considered in isolation. To be considered in context => register cannot be responsible for all register-related tasks

# Example



## High Cohesion: Discussion

- Cohesion is a measure that shows how strong responsibilities of a class are coupled.

- Is an "evaluative" pattern:
  - use it to evaluate alternatives
  - aim for maximum cohesion (well-bounded behavior)

- Cohesie:
  - number of methods ↗ (bloated classes)
  - Understandability ↘
  - reuse ↘
  - Maintainability ↘

## High Cohesion: Discussion

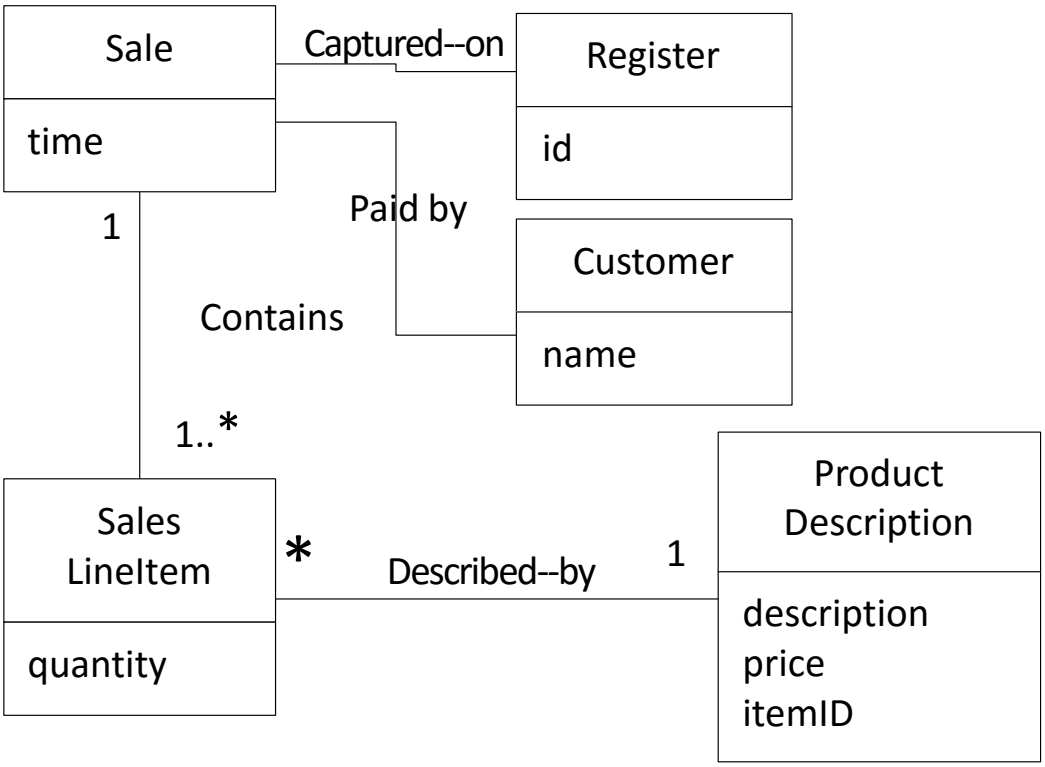
- Scenarios:
  - Very Low Cohesion: A Class is solely responsible for many things in very different functional areas
    - class RDB-RPC-Interface: handles Remote Procedure Calls as well as access to relational databases.
  - Low Cohesion: A class has sole responsibility for a complex task in one functional area.
    - class RDBInterface: completely responsible for accessing relational databases.
    - methods are coupled, but lots and very complex methods.
  - High Cohesion. A class has moderate responsibilities in one functional area and collaborates with other classes to fulfil tasks.
    - Class RDBInterface: partially responsible for interacting with relational databases

# High Cohesion: Discussion

- Advantages:
  - Classes are easier to maintain
  - Easier to understand
  - Often support low coupling
  - Supports reuse because of fine grained responsibility
- Rule of thumb: a class with high cohesion has a relatively small number of methods, with highly related functionality, and does not do too much work.

# Information Expert Principle

- Who should be responsible for **knowing** the grand total of a sale?



## Information Expert

- A very basic principle of responsibility assignment
- Assign a responsibility to the object that has the information necessary to fulfill it - the information expert
  - That which has the information, does the work"
  - Related to the principle of "low coupling"

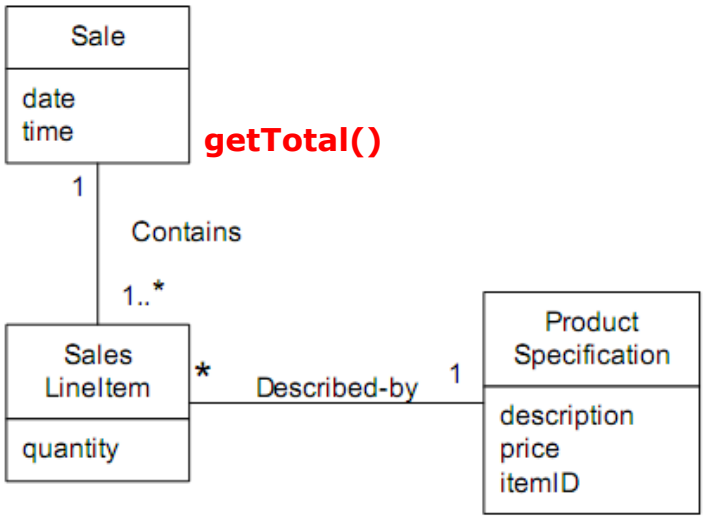
⇨ Localize work

## Information Expert

- Problem: What is a general principle of assigning responsibilities to objects?
- Solution: Assign a responsibility to the information expert, **the class that has the information necessary to fulfill the responsibility**
- Start assigning responsibilities by clearly stating responsibilities!
- Typically follows common intuition
- Design Classes (Software Classes) instead of Conceptual Classes

# Information Expert

- What information is needed to determine the grand total?
  - Line items and the sum of their subtotals
- *Sale* is the information expert for this responsibility.

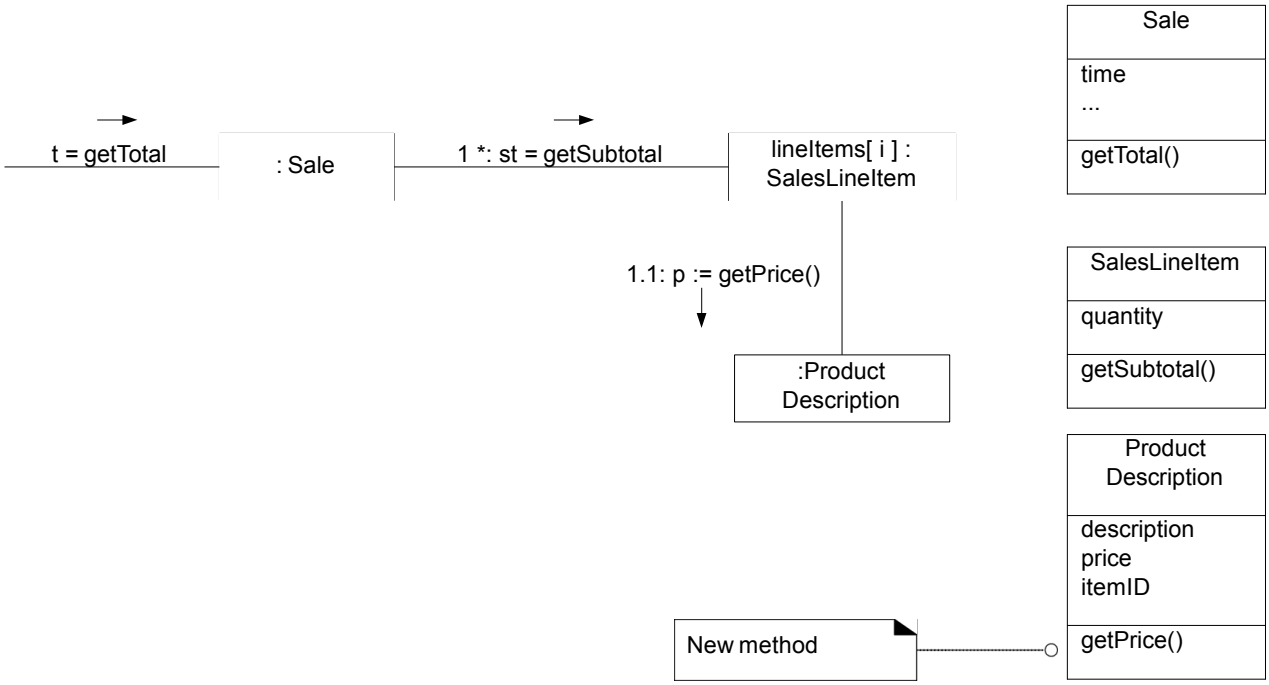



## Information Expert

- To fulfill the responsibility of knowing and answering the sale's total, three responsibilities were assigned to three design classes of objects

| Design Class | Responsibility |
|---|---|
| Sale | knows sale total |
| SalesLineItem | knows line item subtotal |
| ProductSpecification | knows product price |

# Information Expert




# Information Expert: Discussion

- Contraindication: Conflict with separation of concerns
  - Example: Who is responsible for saving a sale in the database?
  - Adding this responsibility to Sale would distribute database logic over many classes →→low cohesion

# Discussion

```
class Foo {
public:
  void SetValue(int);
  int GetValue();
private:
  int value;
};
```

```
Some other class …

Foo *foo = new Foo();
…
int v = foo->GetValue();  if (v >=
100)
  foo->SetValue(v * 0.8);  else
  foo->SetValue(v * 0.9);
…
```

What's wrong?

# Discussion

```
class Foo {
public:
  void SetValue(int);
  int GetValue();
private:
  int value;
};
```

```
Some other class …

Foo *foo = new Foo();
…
int v = foo->GetValue();  if (v >=
100)
  foo->SetValue(v * 0.8);  else
  foo->SetValue(v * 0.9);
…
```

What's wrong?

**Violates information expert**

# Discussion

```
class Foo {
public:
  void SetValue(int);  int
  GetValue();
  void Discount()
  {
     if (value > 100)
      value *= 0.8;
     else
       value *= 0.9
  }
private:
  int value;
};
```

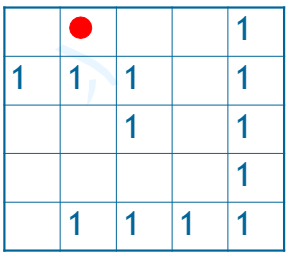
```
Some other class …

Foo *foo = new Foo();
…
foo->Discount();
…
```

Better!!!

# Discussion

```
class Map {
 public:
  …
  int GetCell(int x, int y)
  {
     return cell[x][y];
  }
   private:
  int cell[100][100]
};
```




```
class Hero
{
 …
 void Move()
 {
   if (map->GetCell(x,y) == 1)
     …
     else
     …
     }
   Map *map;
   int x, y;
 }
```

What's wrong?

# Discussion

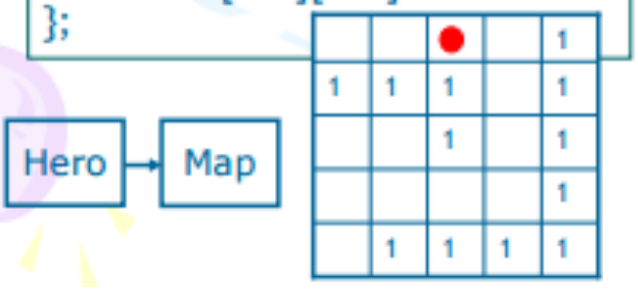
```
class Map {
 public:
  …
  int GetCell(int x, int y)
  {
          return cell[x][y];
  }
  bool isWalkable(int x, int y)
  {
    return cell [x][y]==1;
  }
 private:
  int cell[100][100]
};
```

```
class Hero
{
 …
 void Move()
 {
   if (map->IsWalkable(x,y))
     …
     else
     …
     }
    Map *map;
    int x, y;
  }
```

Better!!!

# Discussion- Low coupling

```
class Map {
public:
    …
    int GetCell(int x, int y)
    {
        return cell[x][y];
    }
    bool isWalkable(int x, int y)
    {
        return cell[x][y] == 1;
    }
private:
    int cell[100][100]
};
```



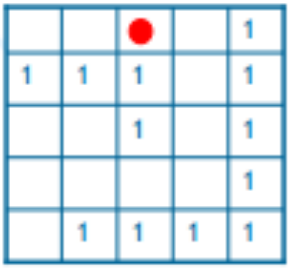

```
class Hero
{
   …
   void Move()
   {
      if (map->IsWalkable(x,y))
         …
      else
         …
   }
   Map *map;
   int x, y;
}
```

One-way dependency

## Discussion- Low coupling

```
class Map {
public:
    ...
    bool isWalkable(Hero *h)
    {
        return cell[h->GetX()]
                   [h->GetY()]
               == 1;
    }
private:
    int cell[100][100]
};
```

```
class Hero
{
  ...
  void Move()
  {
     if (map->IsWalkable(this))
        ...   Two-way dependency
     else
        ...
  }
  Map *map;
  int x, y;
}
```


## Discussion- Coupling and cohesion

Bad coupling and cohesion can result from assigning too many responsibilities to the same class or from assigning the same responsibility to more than one class. In NumbersStatsCommand, the responsibility to parsing the command line has been assigned to two classes.

This slide addresses the (high cohesion-low coupling) case study.

# Polymorphism

- Problem: How to handle behaviour based on type (i.e., class), but not with an if or switch statement? Polymorphism is also used to create pluggable software components.

  "When alternate behaviors are selected based on the type of an object, use polymorphic method call to select the behavior, rather than use if statements to test the type. ", Patterns in Java, Vol 2., p 69

- Solution: In Java, polymorphism is realized by overriding a method from a super class or implementing an interface. The implemented overridden methods are polymorphic in that a client class uses the same method, but the behaviour depends on the type of the object being referenced.

# Polymorphism- Discussion

- The Polymorphism GRASP pattern deals with how a general responsibility gets distributed to a set of classes or interfaces. For example, the responsibility of calculating grades depends on the type of marking scheme for a particular student.
- If polymorphism is not used, and instead the code tests the type of the object, then that section of code will grow as more types are added to the system. This section of code becomes more coupled (i.e., it knows about more types) and less cohesive (i.e., it is doing too much).
- A software component is pluggable if it presents itself as the known class or interface.

## Polymorphism- Example

- A possible set of two dimensional shapes include: circle, triangle, and square. Each of these shapes has an area and a perimeter. The calculation of area and perimeter depend on the type of shape.

- A possible shape interface is:

- shapes/Shape2D.java

- public interface Shape2D {
- public double area();
- public double perimeter();
- }

## Polymorphism- Example

- shapes/Circle.java
- import static java.lang.Math.PI;
- public class Circle implements Shape2D {
- private double radius;
- public Circle( double radius ) {
- this.radius = radius;
- }
- public double area() {
- return PI * radius * radius;
- }
- public double perimeter() {
- return PI * 2 * radius;
- }

- }

# Polymorphism- Example

```
shapes/Triangle.java
import static java.lang.Math.sqrt;
public class Triangle implements Shape2D {
    private double a, b, c;
    public Triangle( double a, double b, double c ) {
        this.a = a;
        this.b = b;
        this.c = c; }
    public double area() {
        // Heron's formula
        return sqrt( (a+b-c)*(a-b+c)*(-a+b+c)*(a+b+c) ) / 4.0; }
    public double perimeter() {
        return a+b+c;
    }
}
```

# Polymorphism- Example

Without polymorphism, the code would be:

```
Shape s = // Triangle, Circle, or Square
if ( s instanceof Triangle ) {
   Triangle t = (Triangle)s;
   System.out.println( t.area() );
else if ( s instanceof Circle ) {
   Circle c = (Triangle)c;
   System.out.println( c.area() );
}
else if ( s instanceof Square ) {
   Square sq = (Square)s;
   System.out.println( sq.area() );
}
```

What's wrong?

# Polymorphism- Example

Without polymorphism, the code would be:

```
Shape s = // Triangle, Circle, or Square
if ( s instanceof Triangle ) {
   Triangle t = (Triangle)s;
   System.out.println( t.area() );
else if ( s instanceof Circle ) {
   Circle c = (Triangle)c;
   System.out.println( c.area() );
}
else if ( s instanceof Square ) {
   Square sq = (Square)s;
   System.out.println( sq.area() );
}
```

What's wrong?

The code fragment is not as cohesive as the first. It also has more coupling, since it knows about Triangle, Circle, and Square. Additionally, any new shapes require the modification of the above code.

# Learning Outcomes: Patterns, Tradeoffs

**Identify criteria for the design of a software system and select patterns, create frameworks, and partition software to satisfy the inherent trade-offs.**

- Introduce Gang of Four Concepts
- Describe and use GoF Patterns
  - Adapter
  - Factory
  - Singleton
  - Strategy

# Gang of Four (GoF)

- Ralph Johnson, Richard Helm, Erich Gamma, and John Vlissides (left to right)

# Gang of Four Design Patterns

**Behavioral**

- Interpreter
- Template Method
- Chain of Responsibility
- Command
- Iterator
- Mediator
- Memento
- Observer
- State
- ✓ Strategy
- Visitor

**Creational**

- ✓ Factory Method
- Abstract Factory
- Builder
- Prototype
- ✓ Singleton

**Structural**

- ✓ Adapter
- Bridge
- Composite
- Decorator
- Façade
- Flyweight
- Proxy

# Design Pattern Categories

- Creational patterns:

–Deal with the process of object creation

–Class Scope : Defer some part of object creation to subclasses(Achieved through inheritance) (compile time)

–Object Scope : Defer object creation to another object(Achieved through association and interface)(run time)

## Design Pattern Categories

- Structural patterns:
  - –Deal with decoupling interface and implementation of classes
  - –Deal with how objects/classes can be combined
    - –Class Scope : use inheritance to compose classes
    - –Object Scope : describe ways to assemble objects

## Design Pattern Categories

*Most patterns are object patterns*

- Behavioural patterns:
  - –Deal with communication between objects and how the objects/classes distribute responsibility)
    - –Class Scope: use inheritance to describe algorithms and flow of control ( Template pattern: We want all the derived classes to implements the algorithm that can vary depending upon the behaviour of subclass.)
    - –Object Scope: describes how a group of objects cooperate to perform task that no single object can complete alone