Object Oriented Programming and Design

Software Engineering Project 2015

Outline:

- Object Oriented Programming
  - *Basics*

- Object Oriented Design
  - *OOD Principles*
  - *General Responsibility Assignment Software Patterns (GRASP)*

- Final Remarks

Object Oriented Programming

- OOP focuses on **Objects** that are defined by their..

States

Behavio
rs

Object Oriented basics

- Encapsulation
- Inheritance
- Abstraction

Encapsulation

- Encapsulation is the inclusion of all the recourses needed for a class to function – basically the methods and the data.

- It allows a class to change its internal implementation without affecting the overall functioning of the system.

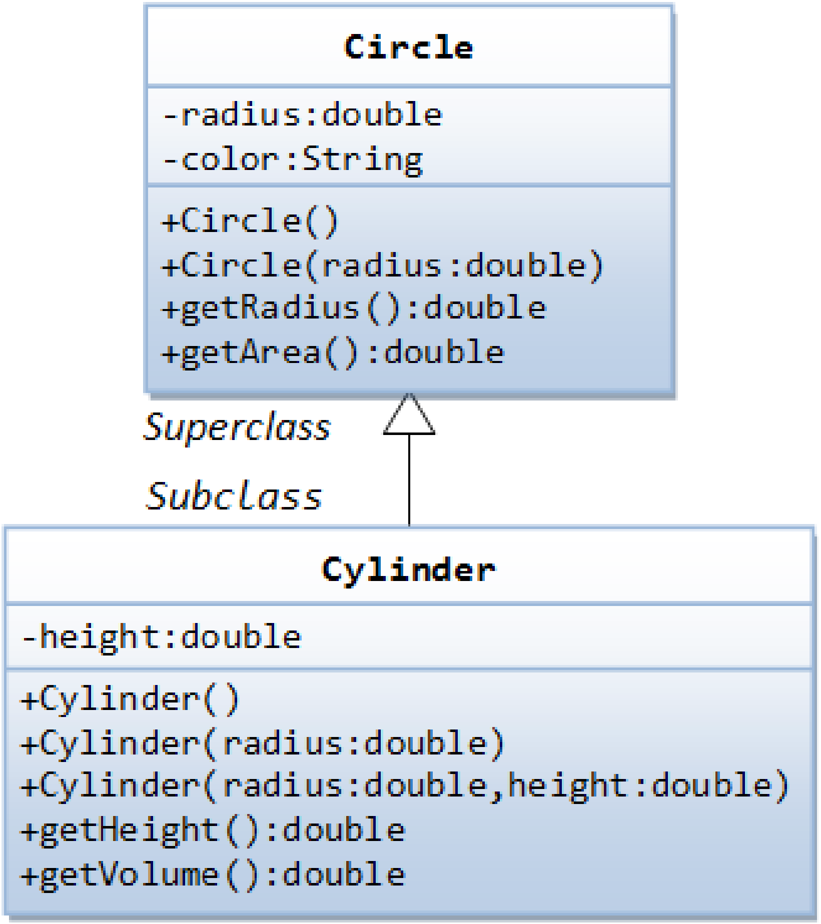
Encapsulation: Example

```
int myArray[] = {1, 21, 3, 8, 5, 13, 2, 34};
Arrays.asList(myArray).contains(8);   // return??
```

Inheritance

- It is the ability of a new class to be created from an existing class by extending it.
- Derived classes inherit properties and methods from the base class, allowing code reuse.

Inheritance: Example



Circle

-radius:double
-color:String

+Circle()
+Circle(radius:double)
+getRadius():double
+getArea():double

Superclass

Subclass

Cylinder

-height:double

+Cylinder()
+Cylinder(radius:double)
+Cylinder(radius:double,height:double)
+getHeight():double
+getVolume():double

Abstraction

- Abstraction is an emphasis on the idea and the properties of an object rather than the details.

- It places the emphasis on what an object is or does rather than how it is represented or how it works.

Abstraction: Example

- MobilePhoneA (Features:- Calling, SMS)
- MobilePhoneB (Features:- Calling, SMS, MP3)
- MobilePhoneC (Features:- Calling, SMS, MP3, Camera)

Abstraction: Example

- MobilePhoneA (Features:- **Calling**, **SMS**)
- MobilePhoneB (Features:- **Calling**, **SMS**, MP3)
- MobilePhoneC (Features:- **Calling**, **SMS**, MP3, Camera)

Abstraction: Example

```
abstract class MobilePhone
 {
        public void Calling();
        public void SendSMS();
 }
```

```
public class MobliePhoneA : MobilePhone {
 }

public class MobilePhoneB : MobilePhone {
        public void MP3();
 }

public class MobilePhoneC : MobilePhone {
        public void MP3();
        public void Camera();
}
```

Outline:

- Object Oriented Programming
  - *Basics*

- Object Oriented Design
  - *OOD Principles*
  - *General Responsibility Assignment Software Patterns (GRASP)*

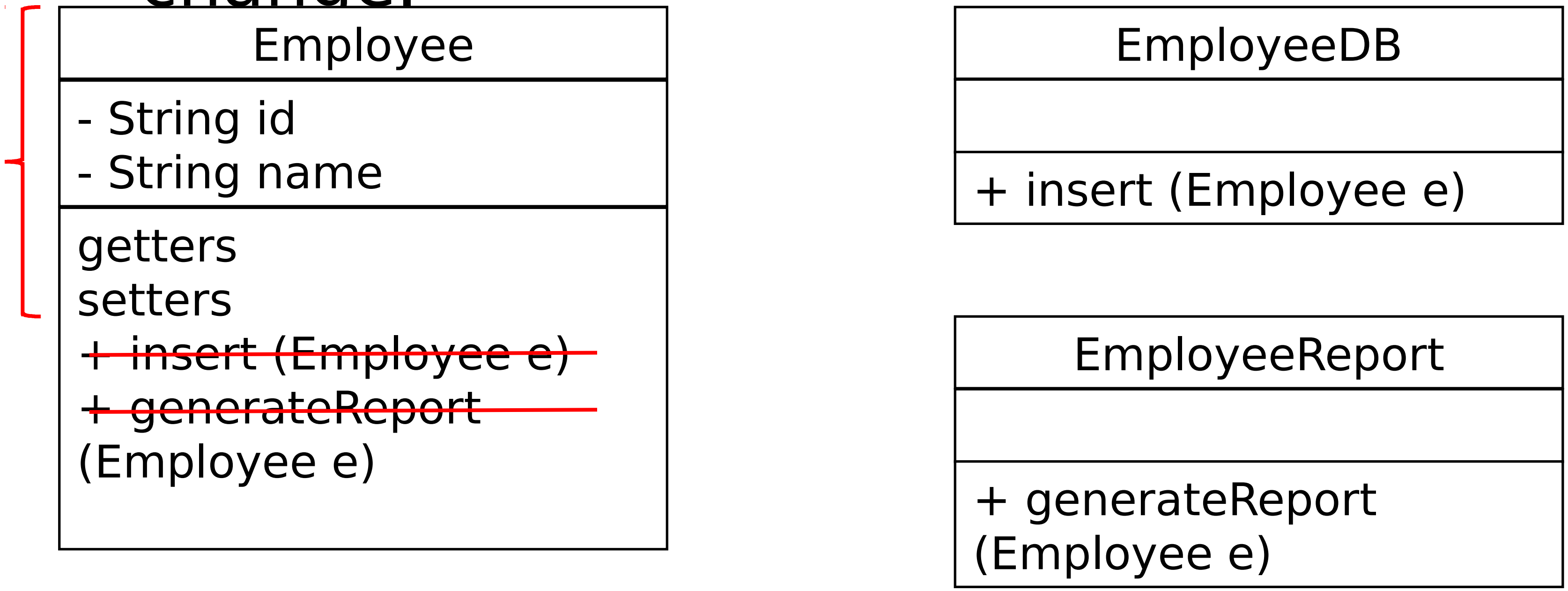
- Final Remarks

OOD Principles

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

Single Responsibility Principle (SRP)

- A class should have one reason to change.



Employee

- String id
- String name

getters
setters
~~+ insert (Employee e)~~
~~+ generateReport~~ (Employee e)

EmployeeDB

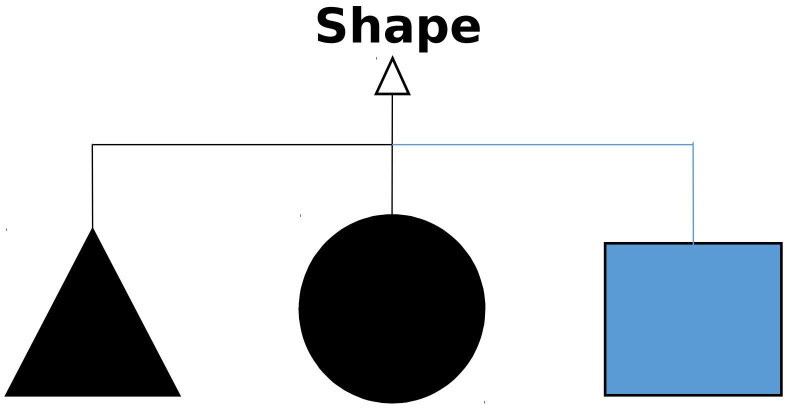
+ insert (Employee e)

EmployeeReport

+ generateReport (Employee e)

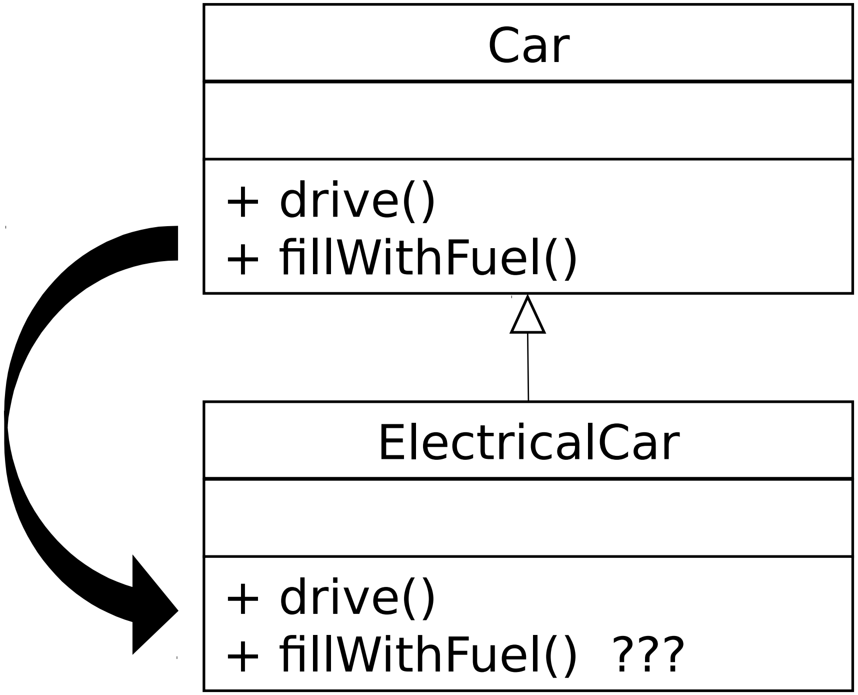
Open Closed Principle (OCP)

- Be able to extend a class behavior, without modifying it.



Liskov Substitution Principle (LSP)

- Derived classes must be substitutable for their base classes.



Car

+ drive()
+ fillWithFuel()

ElectricalCar

+ drive()
+ fillWithFuel() ???

Interface Segregation Principle (ISP)

- Make fine grained interfaces that are client specific.

Dependency Inversion Principle (DIP)

- Depend on abstractions, not on concretions

Object Oriented Design & GRASP:

- OOD is the process of planning a system of interacting objects for the purpose of solving a **software problem.**

- **G**eneral **R**esponsibility **A**ssignment **S**oftware **P**atterns (**GRASP)**, consist of guidelines for assigning responsibility to classes and objects in object oriented design.

GRASP:

1. Creator
2. Information Expert
3. High Cohesion
4. Low Coupling
5. Polymorphism
6. Pure Fabrication
7. Indirection
8. Law of Demeter
9. Controller

Patterns:

- *“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.”*
- *“A pattern is a recurring **solution** to a standard **problem**, in a **context**.”*

Christopher Alexander

Design Patterns and Engineering:

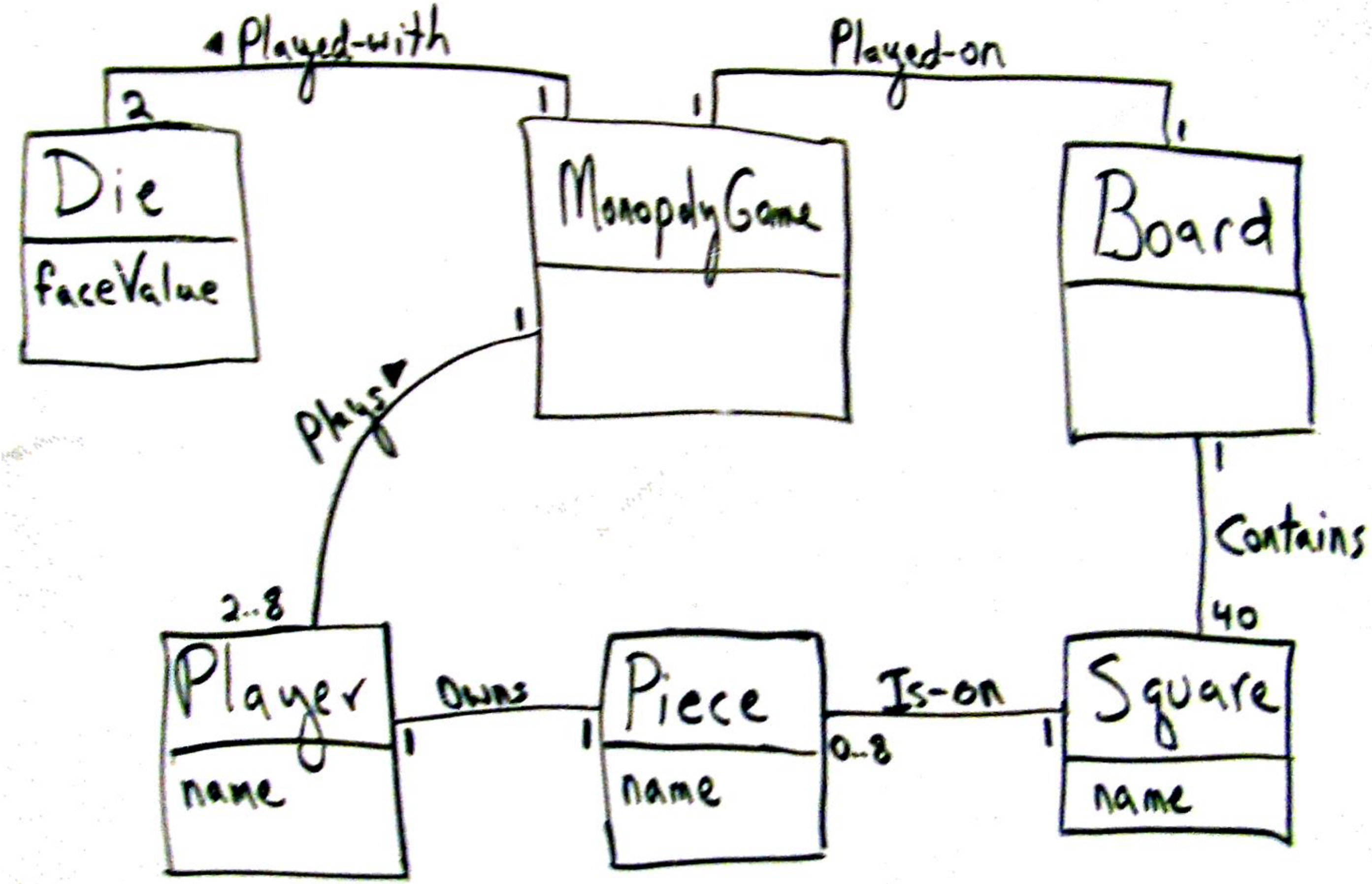
- Car manufacturer do not start from the laws of physics when they design a car.
- They have manuals that describe good solutions to known problems.
- They apply standard solutions that are known to work and learn from experience.
- So, patterns should be important to software engineering.

1. Creator

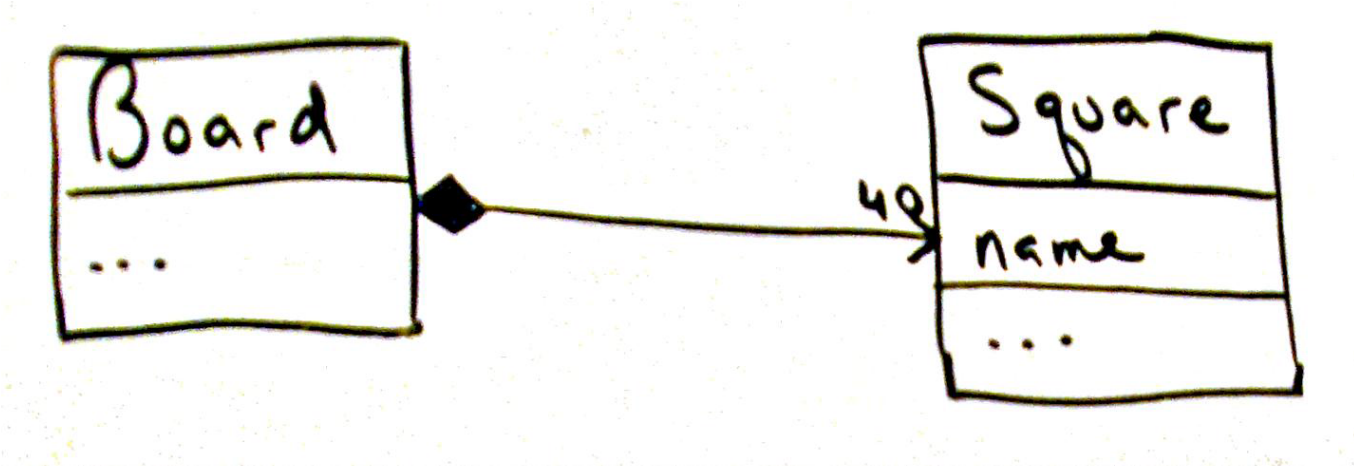
- Pattern to determine: who creates instances of class A

- B is responsible for it if:
  - B contains or aggregates A
  - B saves A
  - B uses A
  - B has data to initialize A.

Creator: Example



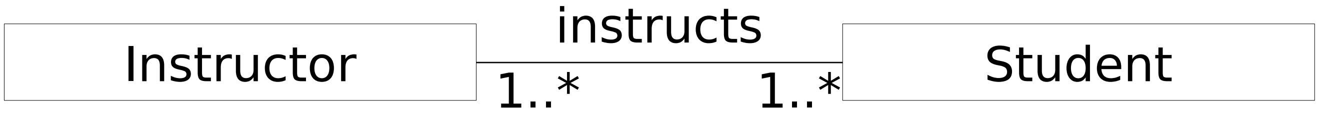
Creator: Example



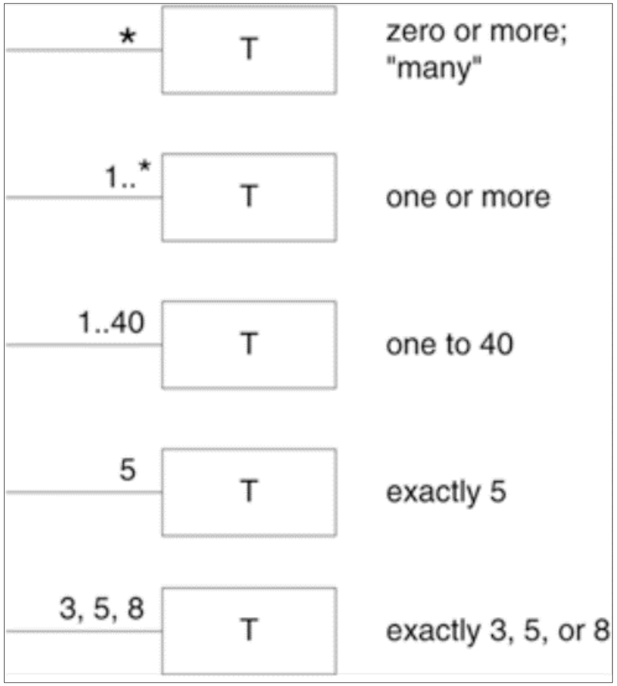
- We need to translate **associations** to **aggregations**, or **compositions**

Association:

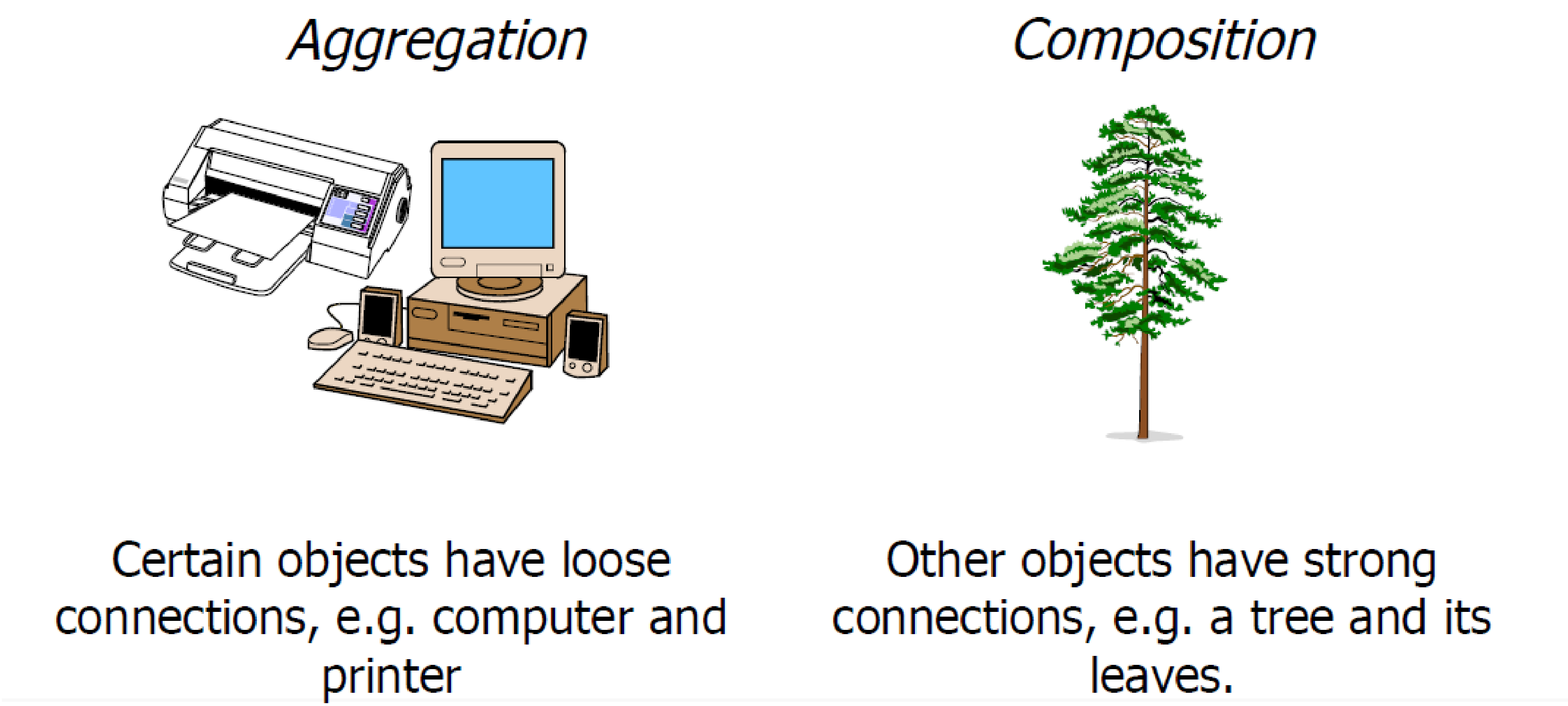
- It is a relationship between two classes.



Multiplicity:



Aggregation Vs. Composition



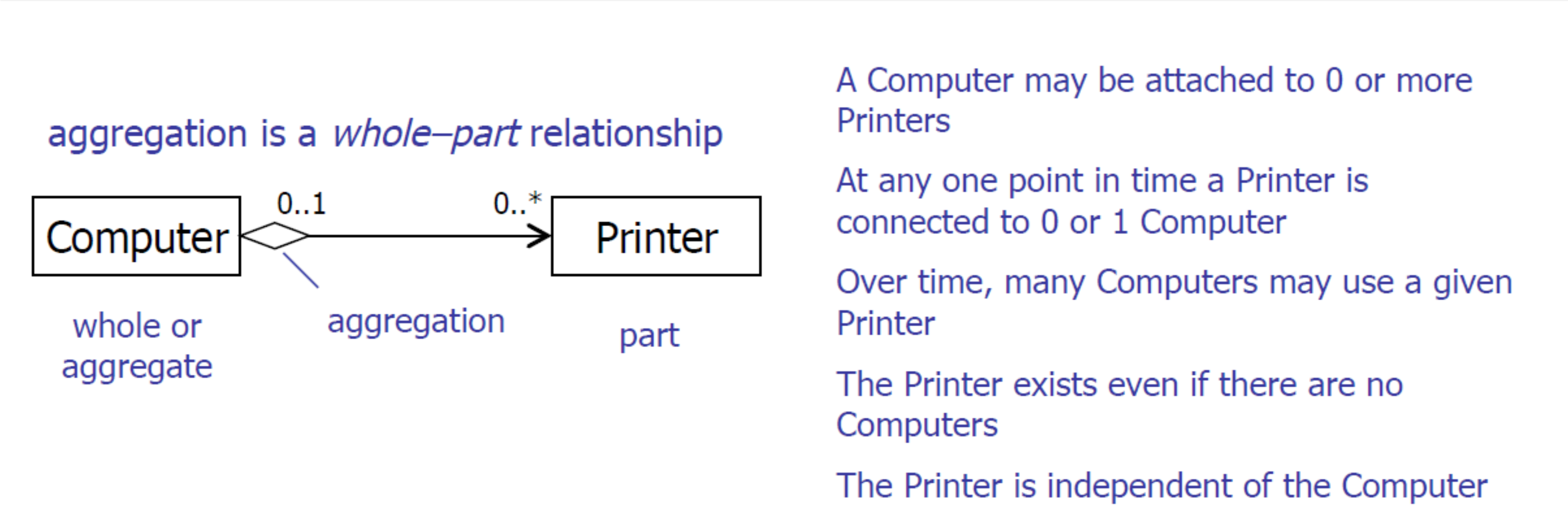
Aggregation

Composition

Certain objects have loose connections, e.g. computer and printer

Other objects have strong connections, e.g. a tree and its leaves.

Aggregation: Example



aggregation is a *whole–part* relationship

A Computer may be attached to 0 or more Printers

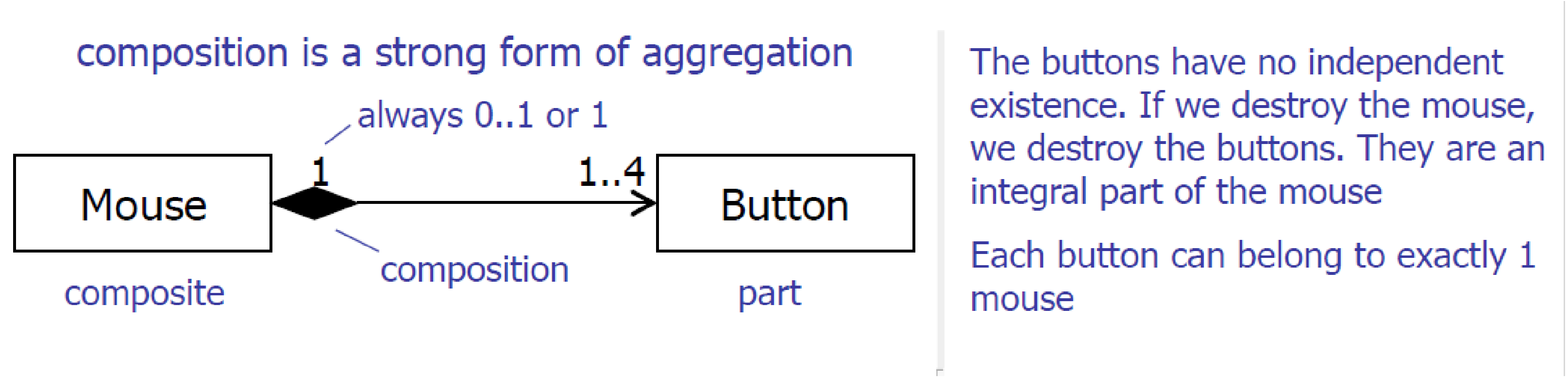
At any one point in time a Printer is connected to 0 or 1 Computer

Over time, many Computers may use a given Printer

The Printer exists even if there are no Computers

The Printer is independent of the Computer

Composition: Example

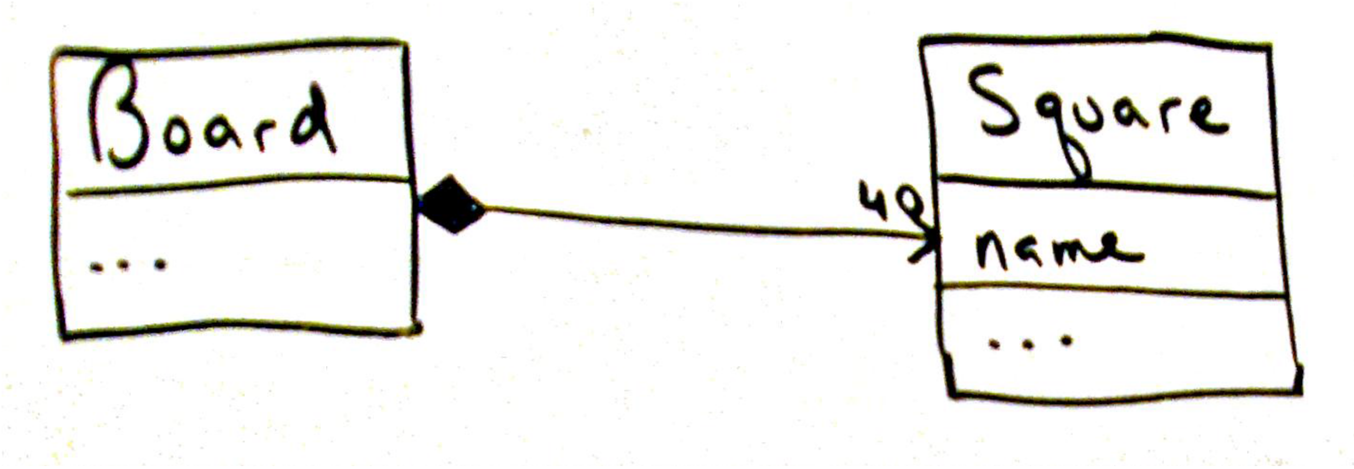


composition is a strong form of aggregation

The buttons have no independent existence. If we destroy the mouse, we destroy the buttons. They are an integral part of the mouse

Each button can belong to exactly 1 mouse

Composition: Example



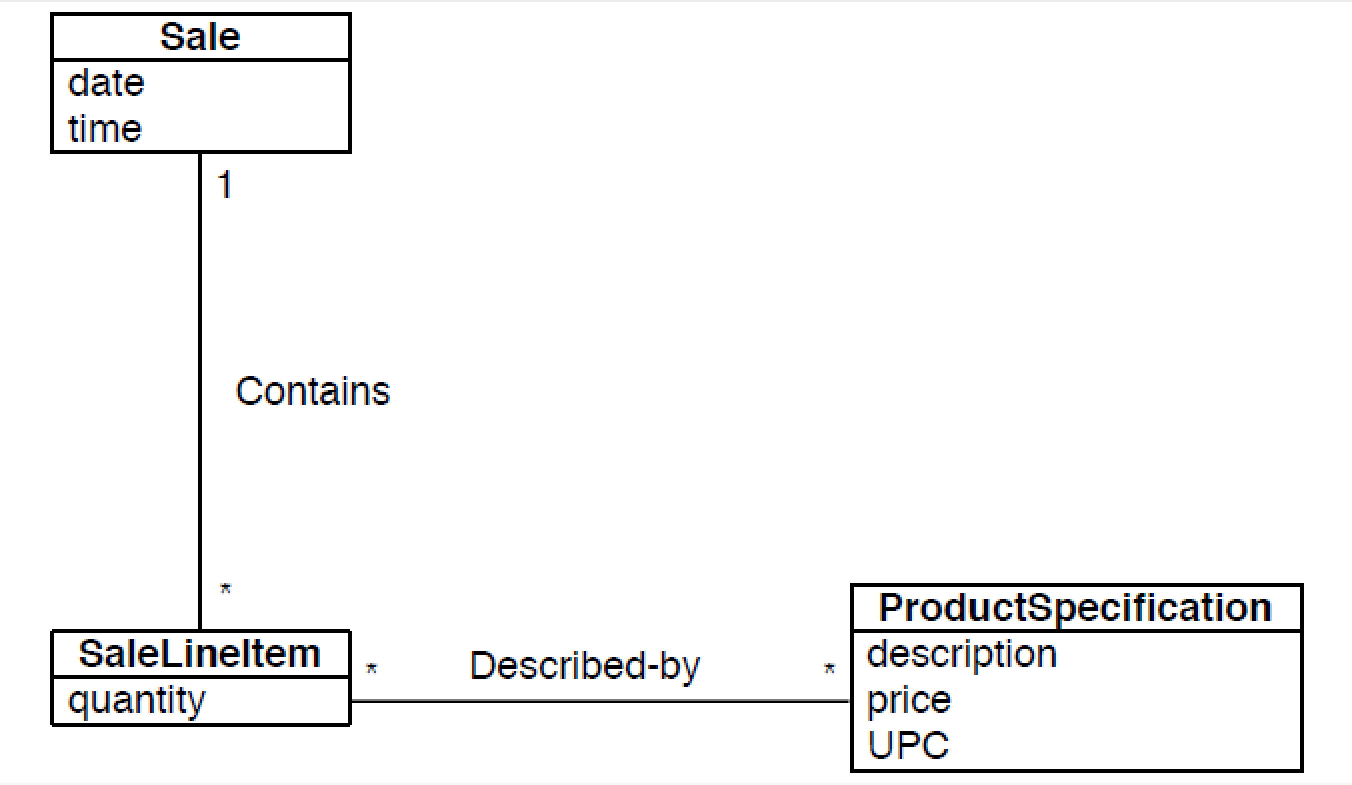
2. Information Expert

- How should responsibility be distributed among objects?

- The class/object with enough information should be responsible!

2. Information Expert: Example

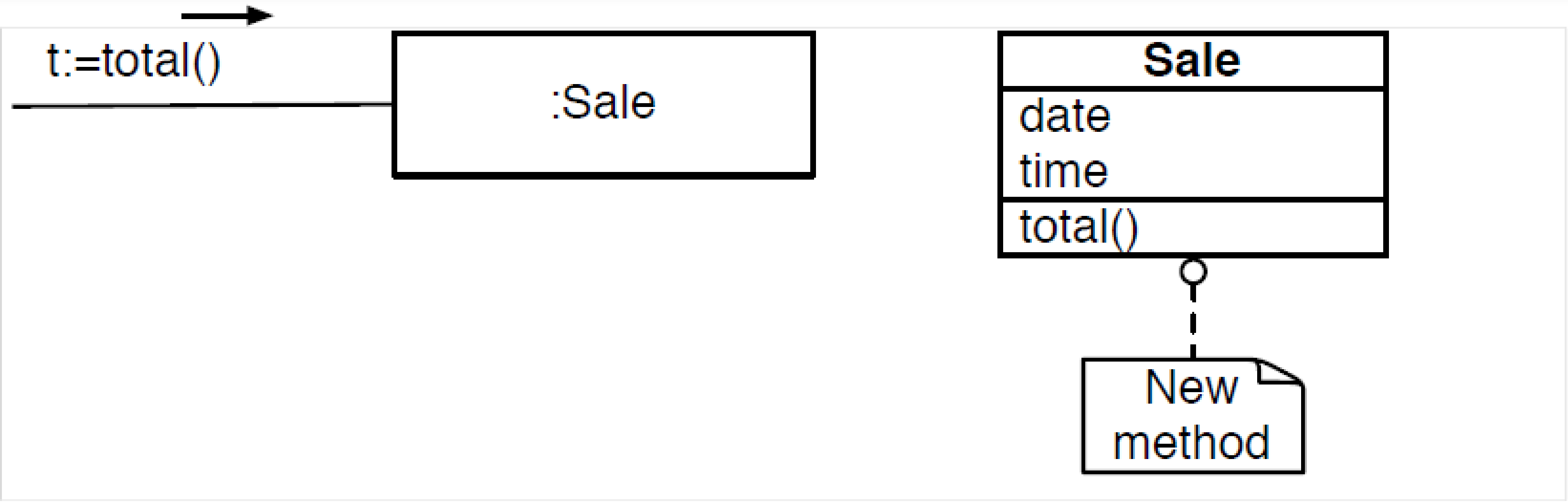
- Who is responsible for knowing the total of a Sale?



2. Information Expert: Example

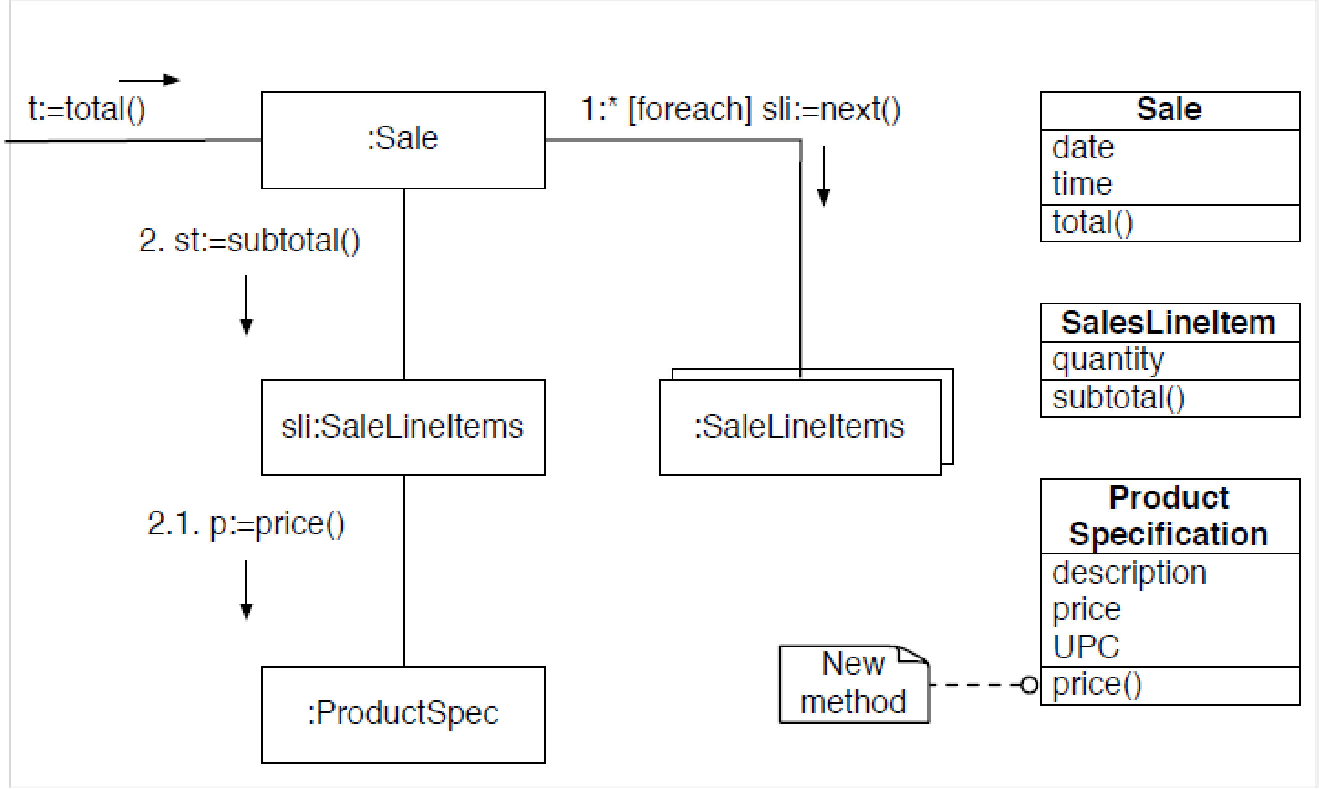
- Needs all instances of ***SalesLineItem*** and their sums

- ***Sale*** is Information Expert for the **total**



2. Information Expert: Example

- **Subtotal** is required for each ***SalesLineItem*** computed from price and quantity, so ***SalesLineItem*** is IE



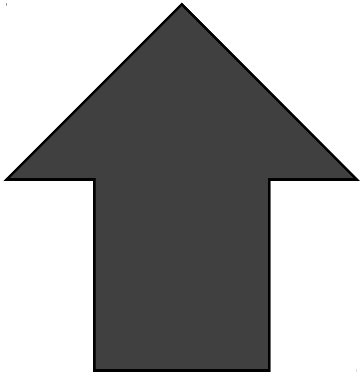
2. Information Expert: Example

Class	**Responsibility**
Sale	Total sum
SalesLineItem	Sum (subtotal)
ProductSpecification	Price

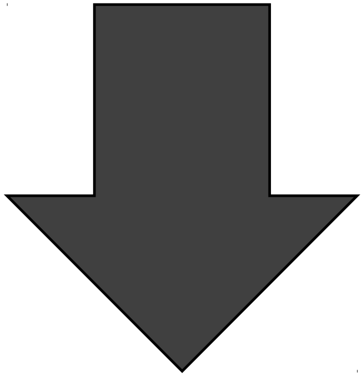
3.High Cohesion & 4.Low Coupling

"**Coupling**" describes the relationships between modules, and “**Cohesion**" describes the relationships within them.

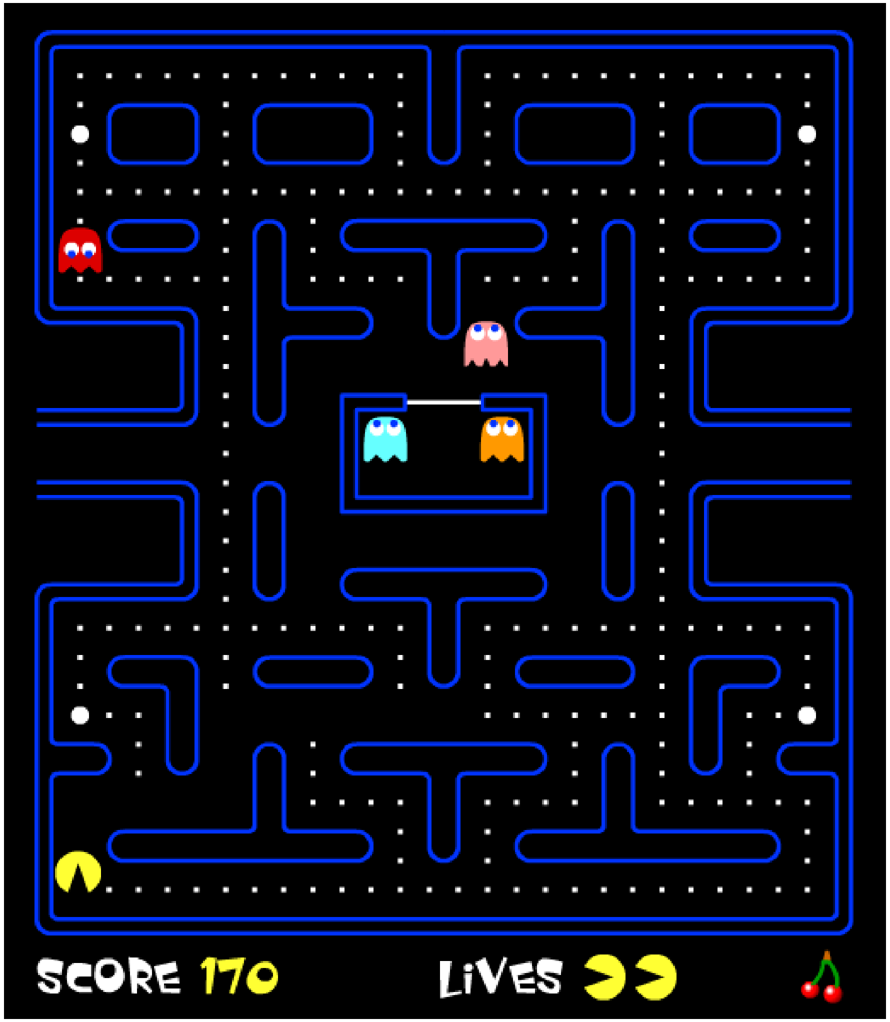
Cohesion



Coupling



Example: PAC-MAN



Example: Cohesion

```
public class Ghost {

public void move()
{}



public void
changeState() {
// change direction
// change color
//change speed
}



}
```

```
public class Ghost {

public void move() {}

public void changeDirection() {}
public void changeSpeed() {}
public void changeColor() {}

public void changeState() {
// call the 3 previous methods
}

}
```

Example: Coupling

```
public class PacMan{

…

If
(pacMan.eats(powerDot
)){
ghosts.changeState();
}

…

}
```

```
public class Game{

…

If (pacMan.eats(powerDot)){
ghosts.changeState();
}

…

}
```

Advantages

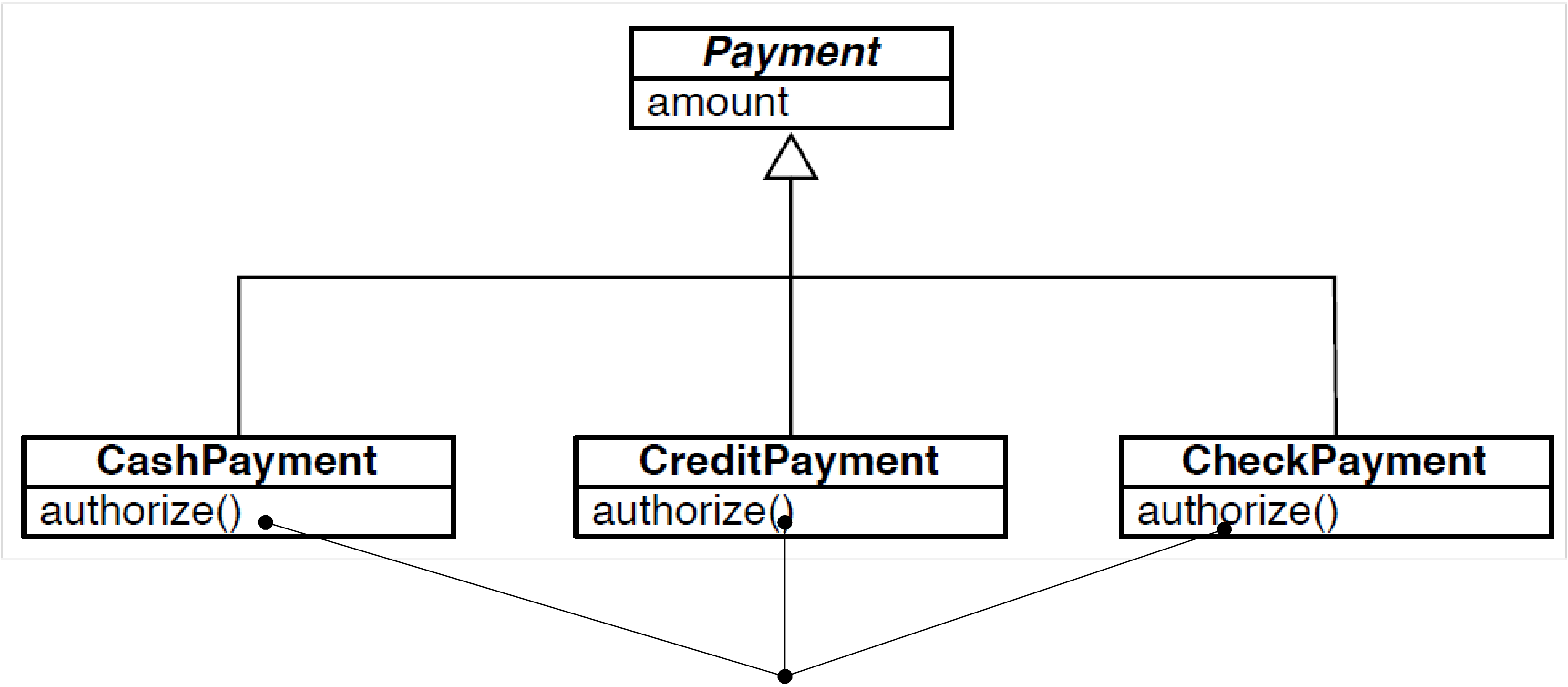
- High Cohesion:
  - complexity be reduced/managed.
  - Distribute responsibility to create cohesive classes/objects.

- Low Coupling:
  - minimize the effects of change and support reuse.
  - Distribute responsibility to minimize dependencies between classes.

5. Polymorphism

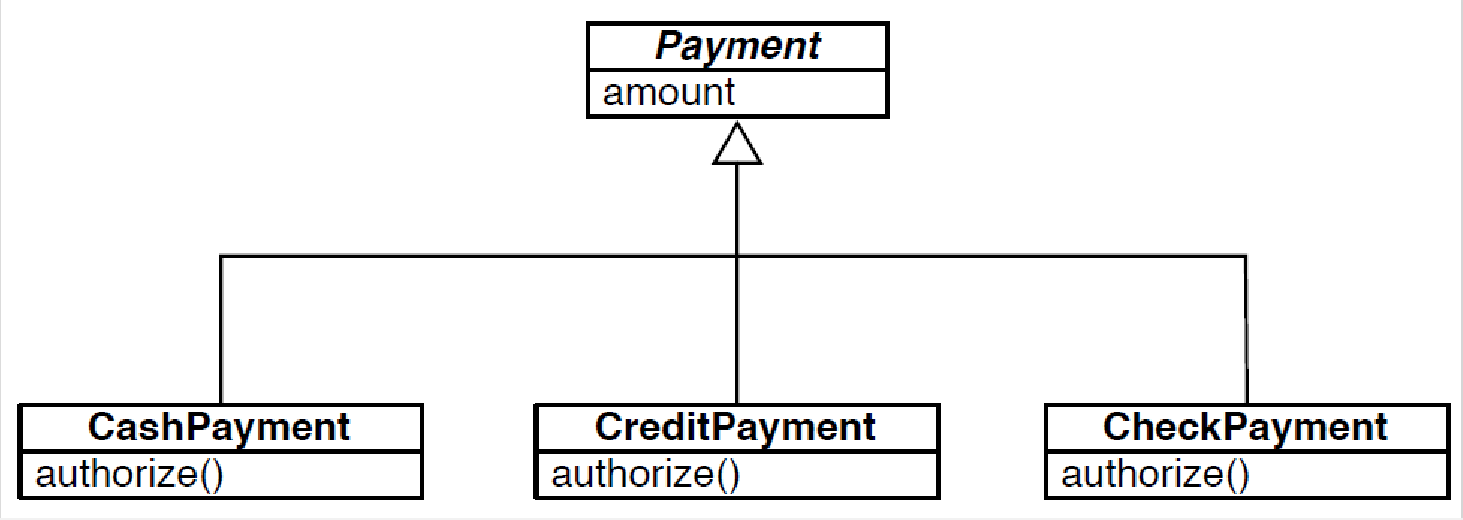
- How can alternatives be managed based on class (type).
- Move responsibility to sub classes (types) and call the correct implementation.

Polymorphism: Example



Each payment type will authorize it self

Polymorphism: Example



What about adding a new type of payment, **Debit Card**?

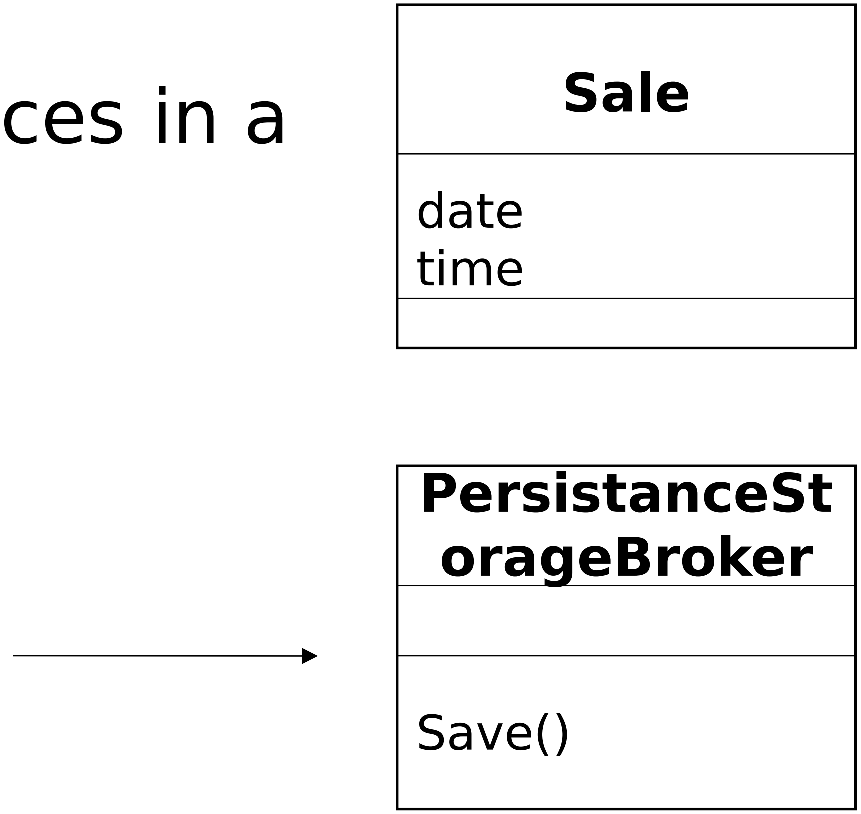
6. Pure Fabrication

- A **Pure Fabrication** is a class that does not represent a concept in the problem domain, specially made up to achieve low coupling, high cohesion, and the reuse potential thereof derived.

6. Pure Fabrication: Example

- Save **Sale** instances in a relational DB.

Pure Fabrication!



7. Indirection:

- The **Indirection** pattern supports low coupling between two elements by assigning the responsibility of mediation between them to an intermediate object.

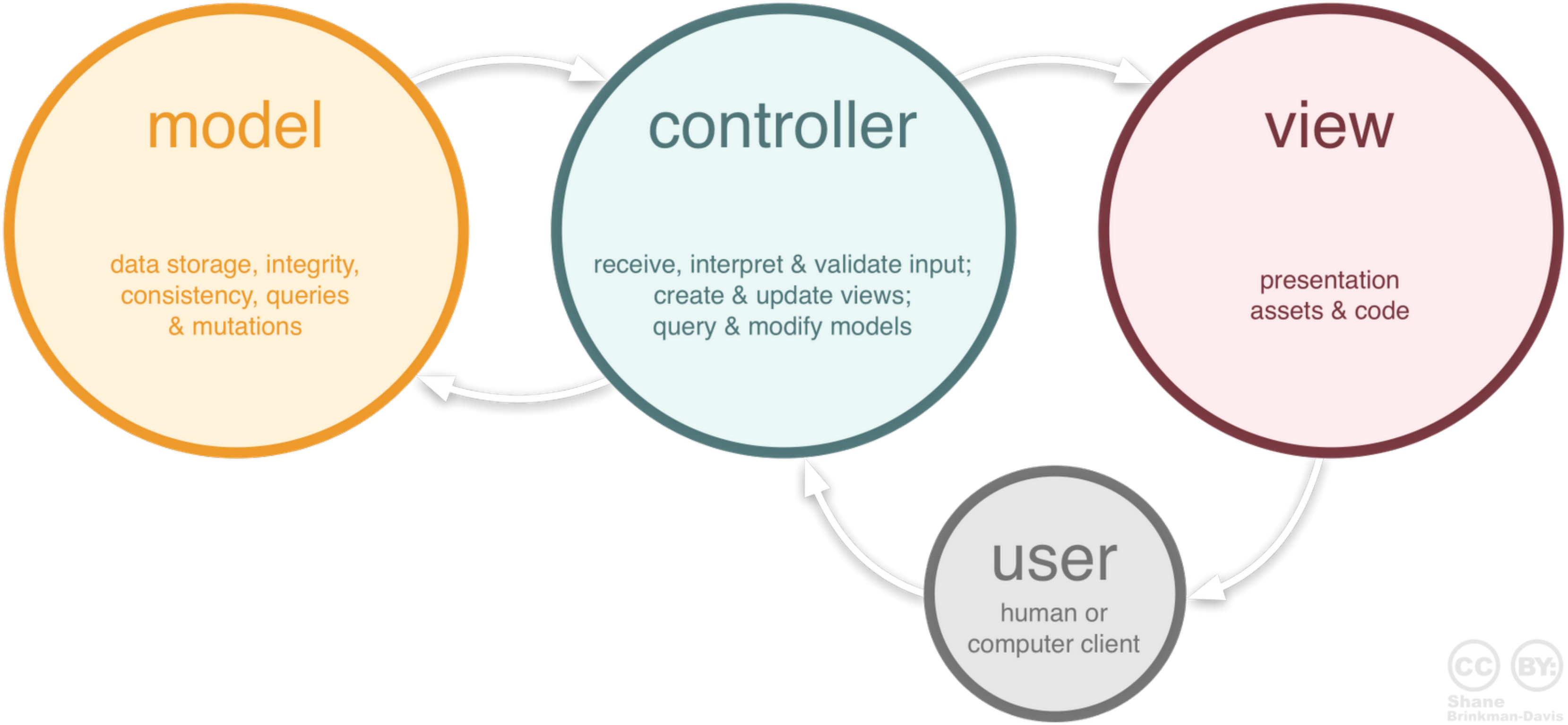
- Model View Controller (MVC)

MVC (Model-View-Controller)

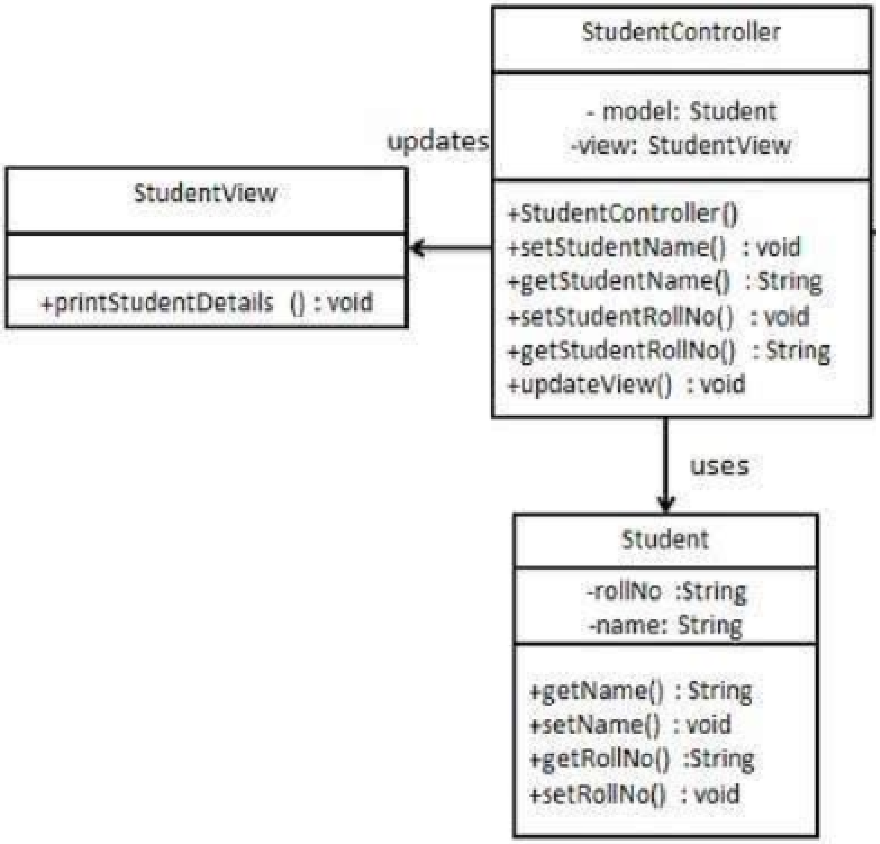
- MVC is a software architectural pattern.

- MVC is the concept of encapsulating some data together with its processing (**the model**) and isolate it from the manipulation (**the controller**) and presentation (**the view**) part that has to be done on a User Interface.

MVC


model
data storage, integrity, consistency, queries & mutations
controller
receive, interpret & validate input; create & update views; query & modify models
view
presentation assets & code
user
human or computer client
Shane Brinkman-Davis

MVC



StudentController

- model: Student
-view: StudentView

+StudentController()
+setStudentName() : void
+getStudentName() : String
+setStudentRollNo() : void
+getStudentRollNo() : String
+updateView() : void

StudentView

+printStudentDetails () : void

Student

-rollNo :String
-name: String

+getName() : String
+setName() : void
+getRollNo() :String
+setRollNo() : void

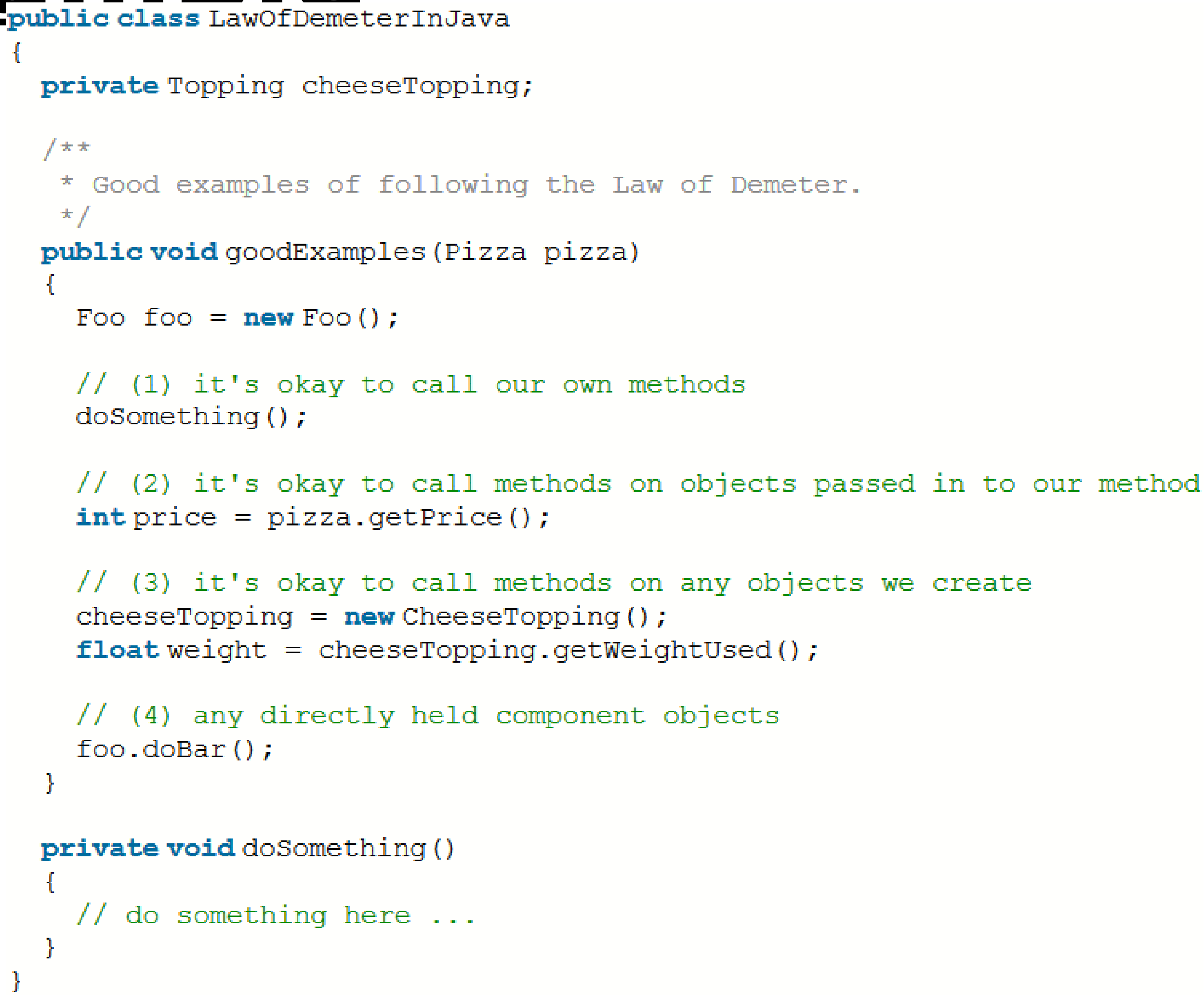
8. Law of Demeter

- “Do not talk to strangers, only talk to your immediate friends.”

- How to avoid knowledge of the structure (variable/operations) of indirect objects?

- Only model relevant associations
  - if two classes/objects have no reason to know of each other, they should not.

8. Law of Demeter: Example



```
public class LawOfDemeterInJava
{
  private Topping cheeseTopping;

  /**
   * Good examples of following the Law of Demeter.
   */
  public void goodExamples(Pizza pizza)
  {
    Foo foo = new Foo();

    // (1) it's okay to call our own methods
    doSomething();

    // (2) it's okay to call methods on objects passed in to our method
    int price = pizza.getPrice();

    // (3) it's okay to call methods on any objects we create
    cheeseTopping = new CheeseTopping();
    float weight = cheeseTopping.getWeightUsed();

    // (4) any directly held component objects
    foo.doBar();
  }

  private void doSomething()
  {
    // do something here ...
  }
}
```

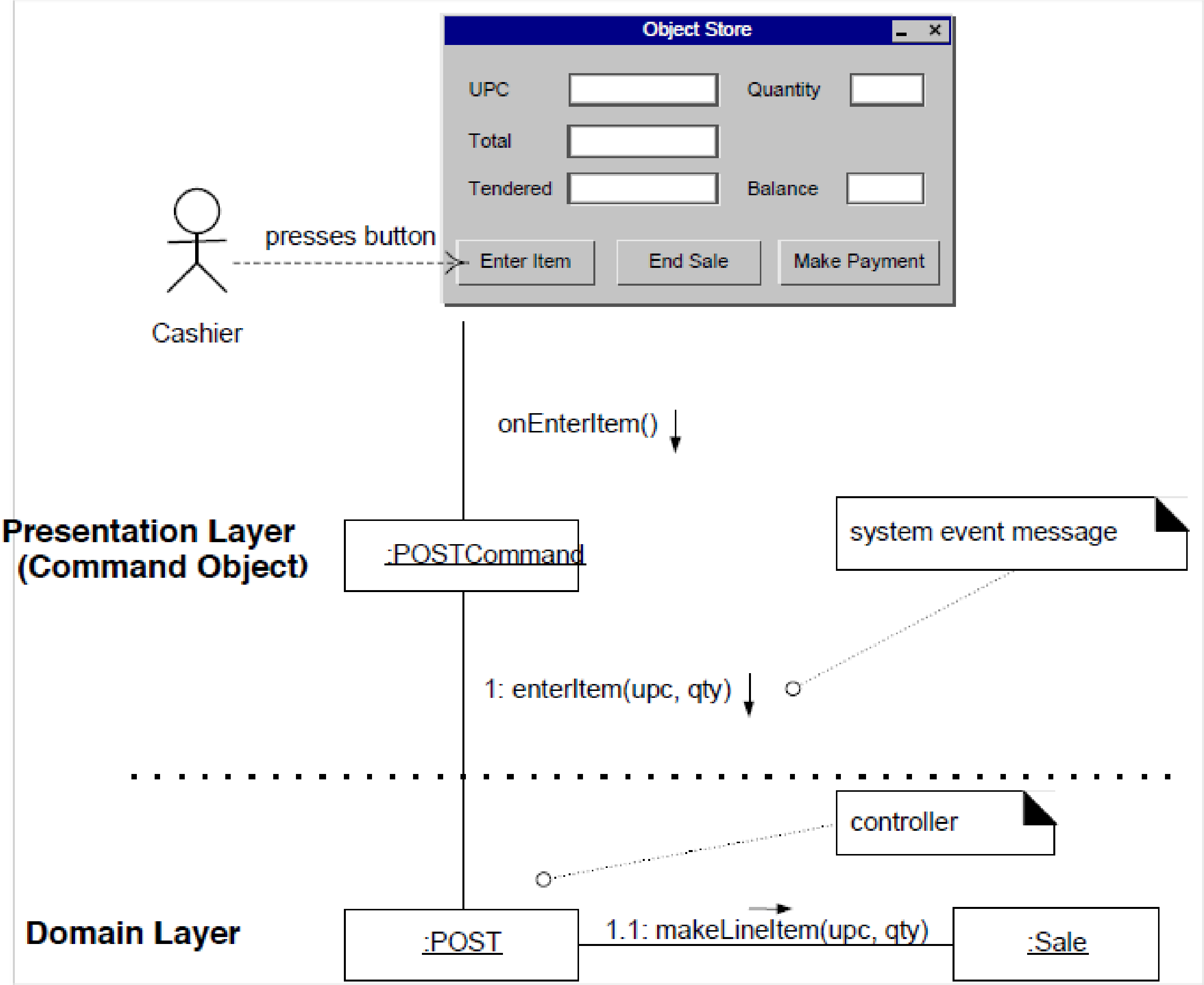
9. Controller

- Who handles a system or UI event?

- **Controller** receives inputs and responds to events.

- Distributes responsibility in two ways:
  - the entire system (façade controller)
  - the case where the event occurred (use case controller)

9. Controller: Example



Outline:

- Object Oriented Programming
  - *Basics*

- Object Oriented Design
  - *OOD Principles*
  - *General Responsibility Assignment Software Patterns (GRASP)*

- Final Remarks

Remarks

- Comment your code adequately.
- Structure your code in such a way to make the classes loosely coupled and highly cohesive.
- Separate the View from the Model/Business Logic.
- Use proper names for classes and methods.

Questions!?

Rodi Jolak
PhD Candidate of Software Engineering
Department of Computer Science and Engineering
Chalmers University of Technology & University of Gothenburg
SE-412 96 Gothenburg, Sweden

jolak@chalmers.se