# DESIGN PRINCIPLES

Lecture 2

# LET'S SWITCH TO MOODLE AND TAKE A QUIZ

It takes 4 minutes.

# CONTENT

General Responsibility ASsignment Principles (GRASP)

Package Design
- Cohesion Principles
- Coupling Principles

# REFERENCES

- Robert Martin
  http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

- Taylor, R., Medvidovic, N., Dashofy, E., Software Architecture: Foundations, Theory, and Practice, 2010, Wiley

- Craig Larman, Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development, 3rd Ed, Addison Wesley, 2004 – Chapters 17, 18.

# LAST TIME

OOP concepts +

**S**ingle Responsibility

**O**pen-Closed

**L**iskov Substitution

**I**nterface Segregation

**D**ependency Inversion

# CHALLENGE

Overloading and polymorphism

Compiler error?

Output?

```java
class Base{
    @Override
    public String toString() { return "This is base";}

}

class Sub extends Base{
    @Override
    public String toString() { return "This is subclass";}
}

class Host{
    public String print(Base b)
    {
        return b.toString() + " din metoda cu base";
    }
    public String print(Sub b)
    {
        return b.toString() + " din metoda cu sub";
    }
 }

public class Polytest {
    public static void main(String[] args){
        Base abase = new Base();
        Sub asub = new Sub();
        Host h = new Host();
        System.out.println(h.print(abase));
        System.out.println(h.print(asub));
    }
}
```

**Output - CMSC (run)**

```
run:
This is base din metoda cu base
This is subclass din metoda cu sub
```

# CONTINUED

Delete one method

Error?

Output?

```java
class Base{
    @Override
    public String toString() { return "This is base";}

}

class Sub extends Base{
    @Override
    public String toString() { return "This is subclass";}
}


class Host{
    public String print(Base b)
    {
        return b.toString() + " din metoda cu base";
    }
    /*public String print(Sub b)
    {
        return b.toString() + " din metoda cu sub";
    }*/
}

public class Polytest {
    public static void main(String[] args){
        Base abase = new Base();
        Sub asub = new Sub();
        Host h = new Host();
        System.out.println(h.print(abase));
        System.out.println(h.print(asub));
    }
}
```

**Output - CMSC (run)** ×

```
run:
This is base din metoda cu base
This is subclass din metoda cu base
```

# HOW TO FAIL WITH SOLID

**Single Responsibility Principle**: Each class has one (*very small*) specific responsibility. That can lead to thousands of classes.

**Open-Closed Principle**: Use *inheritance* to add every new feature.

**Interface Segregation Principle**: Everything is single-inheritance based

**Dependency Inversion Principle**: Everything is abstracted (data-driven) to the extreme.

=> Every single class is ultimately inherited out of the same family tree, sometimes *hundreds* of layers deep.

# GRASP

General Responsibility Assignment Software Patterns

OO system = objects sending messages to other objects to complete operations.

Issues:
- **Responsibilities** assigned to objects
- **Interaction** ways between objects

# RESPONSIBILITIES

**Knowing** responsibilities:

- knowing about private encapsulated data;

- knowing about related objects;

- knowing about things it can derive or calculate;

**Doing** responsibilities:

- doing something itself;

- initiating action in other objects;

- controlling and coordinating activities in other objects;

A responsibility is not the same as a method, but methods are implemented to fulfil responsibilities.

# GRASP: GENERAL PRINCIPLES IN ASSIGNING RESPONSIBILITIES

From Craig Larman's 9 principles:

- **Expert**

- **Creator**

- **Controller**

- **Low Coupling**

- **High Cohesion**

- Polymorphism

- Pure Fabrication

- Indirection

- **Don't Talk to Strangers (Law of Demeter)**

# CASE STUDY – POS SYSTEM

A Point-Of-Sale (POS) system is an application used (in part) 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 and inventory control.

# MODELS

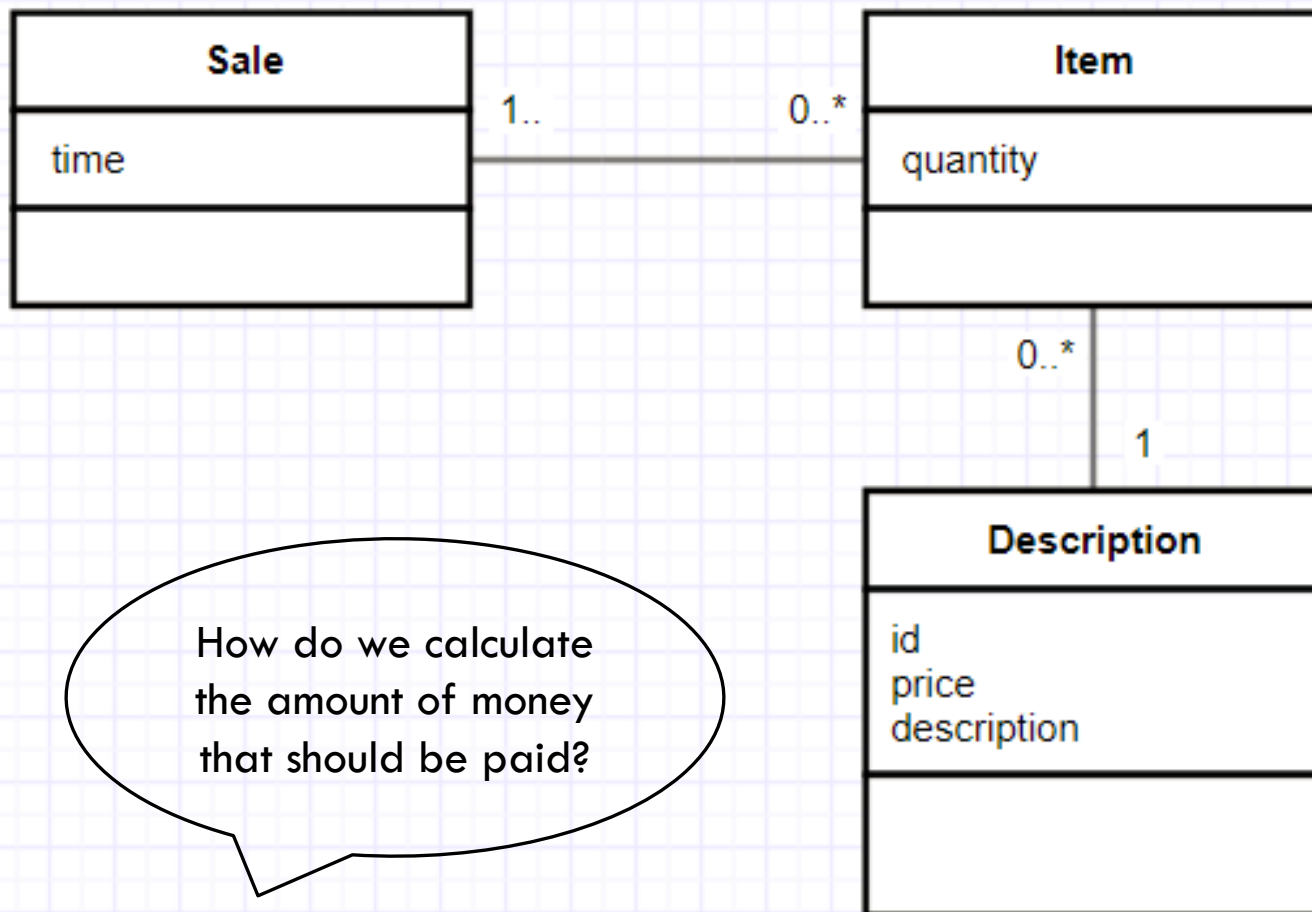Challenge: How do we get from the narrative requirements to an implementable model?

Domain Model

- Sale
- Item
- Payment
- Product
- Register

Design Model

- ?

# DESIGN MODEL

# INFORMATION EXPERT

**Problem:**   What is the most basic principle by which responsibilities are assigned in object-oriented design?

**Solution:**   Assign a responsibility to the information expert - the class that **has the necessary information** to fulfil the responsibility.

# INFORMATION EXPERT EXAMPLE

In the POS application, what class has the information needed to calculate the total amount of money?

SALE
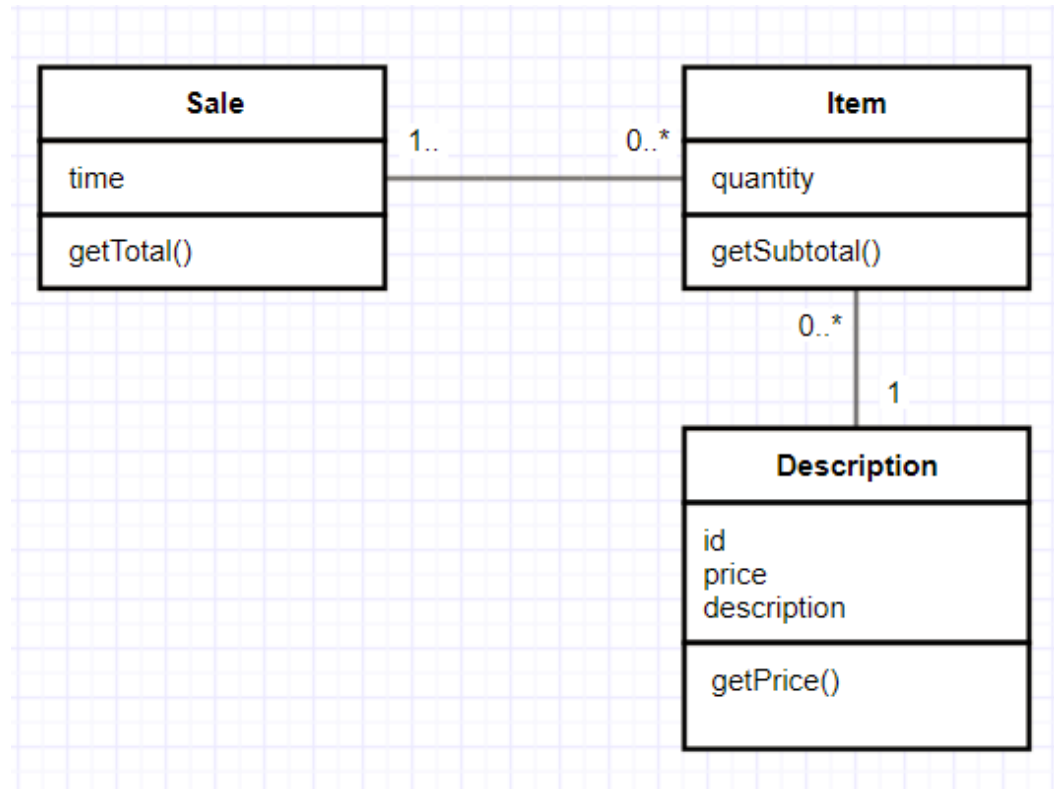
# CONCLUSION

**Discussion**

- Expert - most used principle in the assignment of responsibilities

- Information is spread across different objects => they need to interact

**Benefits**

- Information encapsulation is maintained since objects use their own information to fulfill tasks => supports low coupling

- Behavior is distributed across the classes that have the required information => more cohesive "lightweight" class definitions

# CREATOR

**Problem:** Who should be responsible for creating a new instance of some class?
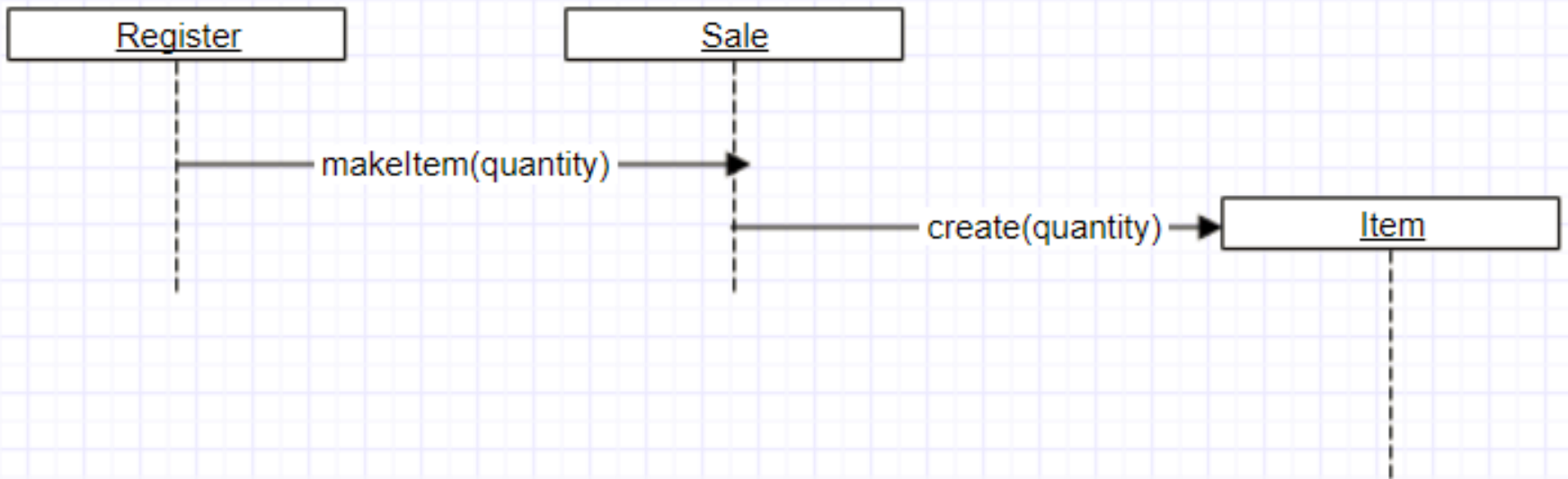
**Solution:** Assign class B the responsibility to create an instance of class A if one of the following is true:
- B contains A objects
- B closely uses A objects
- B has the initialising data that will be passed to A when it is created.

# CREATOR

**Example:** In the POS application who should be responsible for creating an Item instance?

SALE

# CONCLUSION

**Discussion:**

- Creator guides assigning responsibilities related to the creation of objects.
- Sometimes a creator is the class that has the initialising data that will be used during creation.

For example, who should be responsible to create a Payment instance ?

**Benefits:**

- Low Coupling is supported

**Issues:**

- Complex creation procedures
- Solution ?

# CONTROLLER

**Problem:** Who should be responsible for handling a system event?

- A system event is a high level event generated by an external actor.

- A Controller is a non-user interface object responsible for handling a system event.

# CONTROLLER

**Solution:** Assign the responsibility for handling a system event to a class representing one of the following choices:

- Facade controller
  - handles all the events
  - represents the overall "system"
- Role controller
  - handles the events associated to the role
  - represents a person in the real-world
- Use-case controller
  - handles the events associated to a use-case
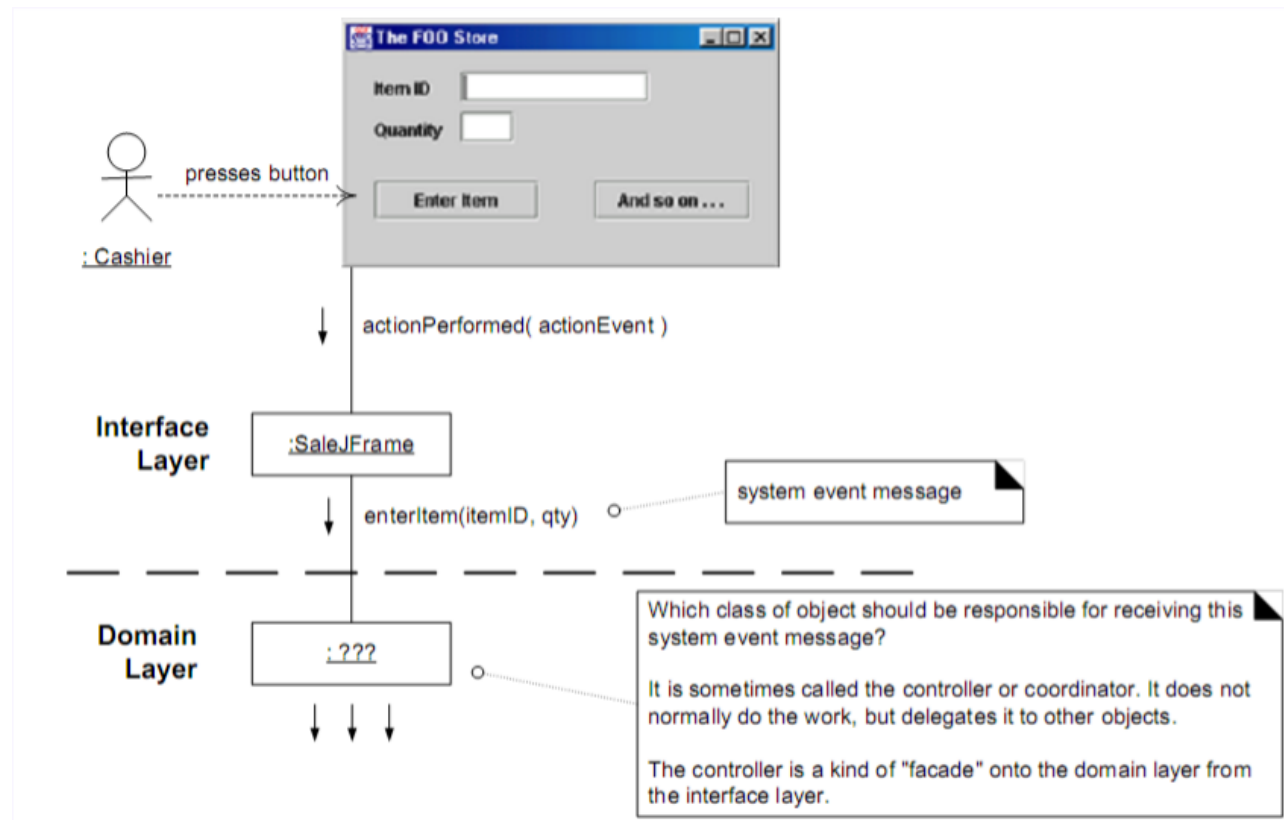  - represents an artificial handler

# CONTROLLER

**Example:** In the point of sale application the current system operations have been identified as:

enterItem()

endSale()

makePayment()



presses button

: Cashier

actionPerformed( actionEvent )

**Interface Layer**

:SaleJFrame

enterItem(itemID, qty)

system event message

**Domain Layer**

: ???

Which class of object should be responsible for receiving this system event message?

It is sometimes called the controller or coordinator. It does not normally do the work, but delegates it to other objects.

The controller is a kind of "facade" onto the domain layer from the interface layer.

# SOLUTION

By the Controller pattern, here are some choices:

- Register, POSSystem: represents the overall "system" (Façade controller)
- ProcessSaleSession, ProcessSaleHandler: represents a handler of all system events of a use case scenario (Use-case controller)
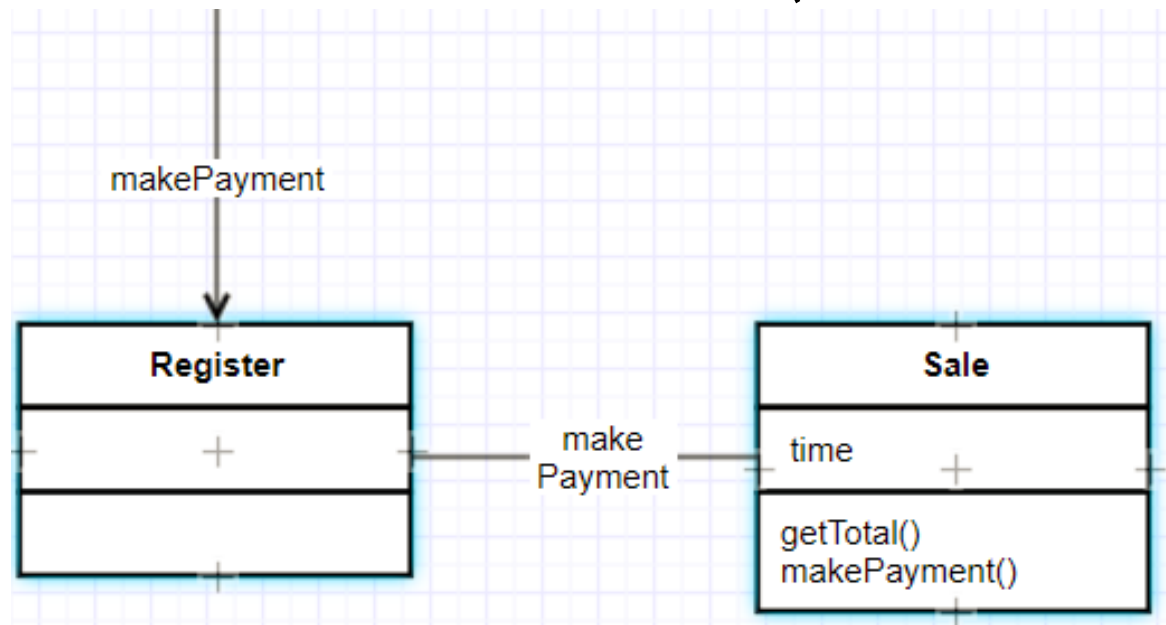
# DISCUSSION

- Usually **a controller delegates** to other objects the work that needs to be done; it coordinates or controls the activity.

- **Facade controllers** are suitable when there are not "too many" system events

- A **use case controller** is an alternative to consider when placing the responsibilities in a facade controller leads to designs with low cohesion or high coupling

- Controller class is called bloated if
  - The class is overloaded with too many responsibilities.
    - Solution – Add more controllers
  - Controller class performs many tasks itself.
    - Solution – controller class has to delegate things to others.

# CONCLUSIONS

**Benefits:**

Increased potential for **reusable** components:

- it ensures that business or domain processes are handled by the layer of domain objects rather than by the interface layer.
- the application is not bound to a particular interface.

Reason about the **state of the use case**:

- As all the system events belonging to a particular use case are assigned to a single class, it is easier to control the sequence of events that may be imposed by a use case (e.g. MakePayment cannot occur until EndSale has occurred).

# LOW COUPLING

How strongly are the objects connected to each other?

Coupling = object depending on other object.

When the independent element changes, it affects the dependent one.

Prefer low coupling – assign responsibilities so that coupling remains low.

# COUPLING

TypeX depends on TypeY if

- TypeX has an **attribute that refers** to a TypeY instance, or TypeY itself.

- TypeX has a **method which references** an instance of TypeY, or TypeY itself, by any means. (Typically include a parameter or local variable of type TypeY, or the returned object is an instance of TypeY.)

- TypeX is a direct or indirect **subclass** of TypeY.

- TypeY is an interface, and TypeX **implements** that interface.

# WHICH IS BETTER COUPLED?
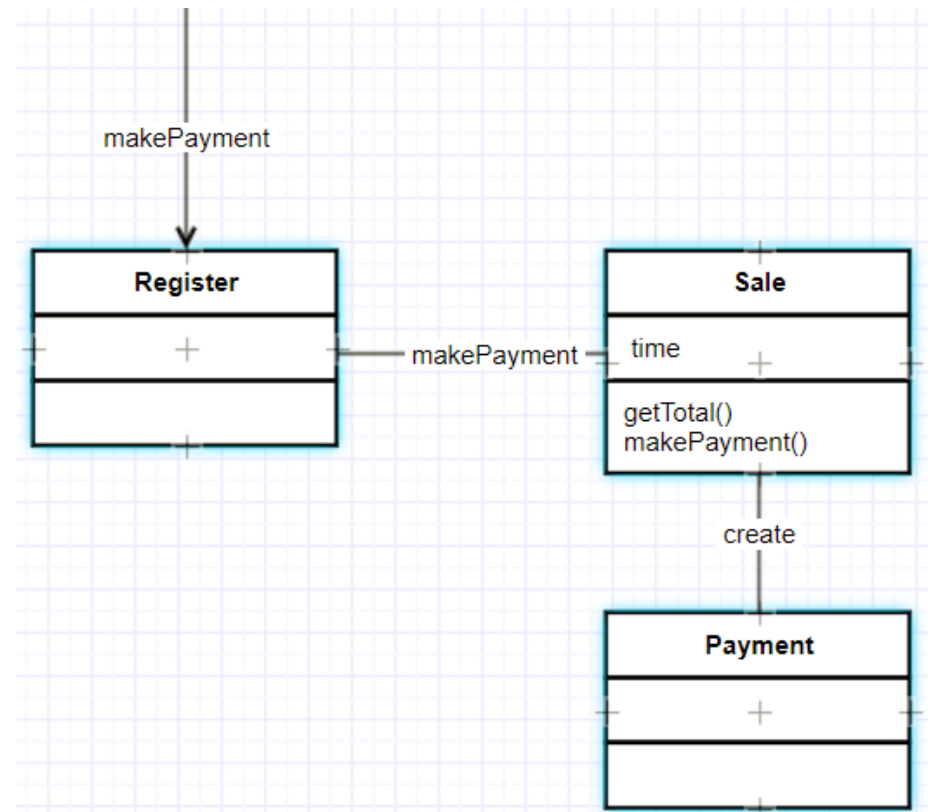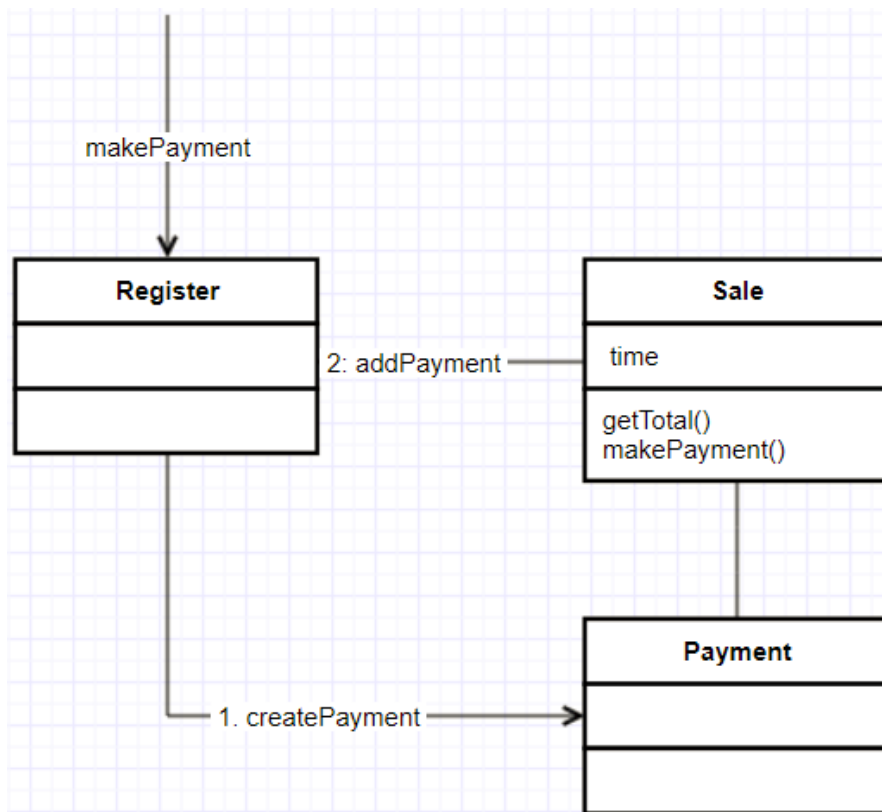
# MEASURING COUPLING

```
class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
    public double Salary { get; set; }
}
```

| Hierarchy | Class Coupling |
|---|---|
| ConsoleApplication6 ( | 0 |
| {} ConsoleApplication | 0 |
| Person | 0 |

```
class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
    public double Salary { get; set; }
}

class PersonStuff
{
    void DoSomething()
    {
        Person myPerson = new Person();
        myPerson.Age = 23;
        myPerson.Name = "Bubba";
        myPerson.Salary = 20.34;
    }
}
```

| Hierarchy | Class Coupling |
|---|---|
| ConsoleApplication6 (D | 1 |
| {} ConsoleApplication6 | 1 |
| Person | 0 |
| PersonStuff | 1 |
| DoSomething() | 1 |
| PersonStuff() | 0 |

```
class PersonStuff
{
    Person myPerson = new Person();

    void DoSomething()
    {
        myPerson.Age = 23;
        myPerson.Name = "Bubba";
        myPerson.Salary = 20.34;
    }
}
```

| Hierarchy | Class Coupling |
|---|---|
| ConsoleApplication6 (D | 1 |
| {} ConsoleApplication6 | 1 |
| Person | 0 |
| PersonStuff | 1 |
| DoSomething() | 1 |
| PersonStuff() | 1 |

[https://blogs.msdn.microsoft.com/zainnab/2011/05/25/code-metrics-class-coupling/]
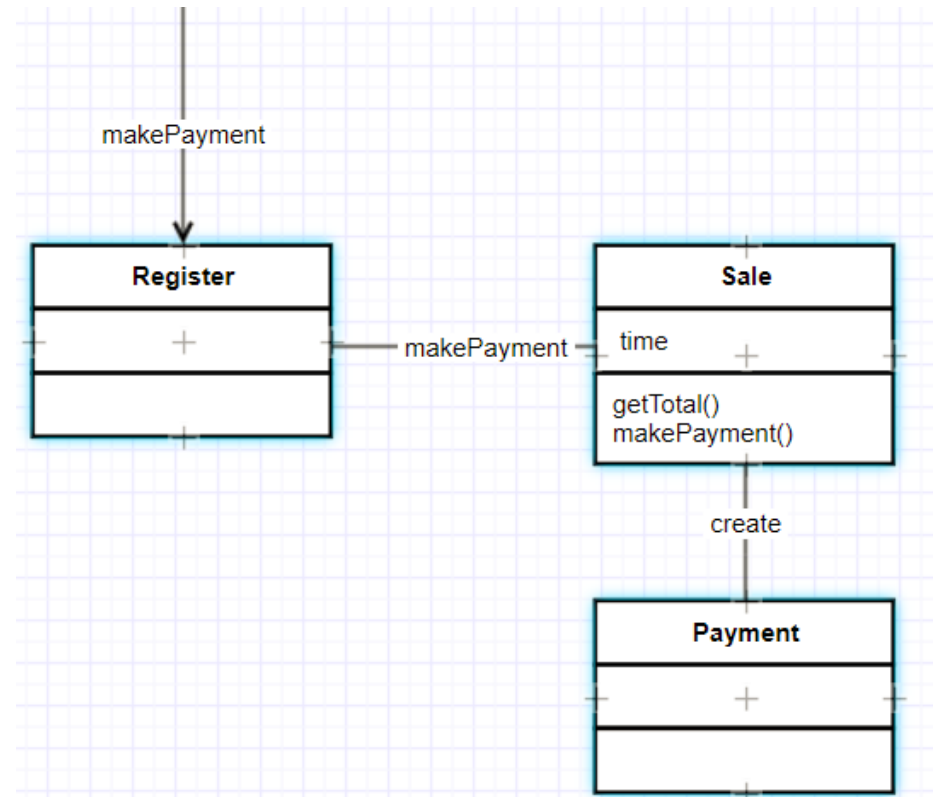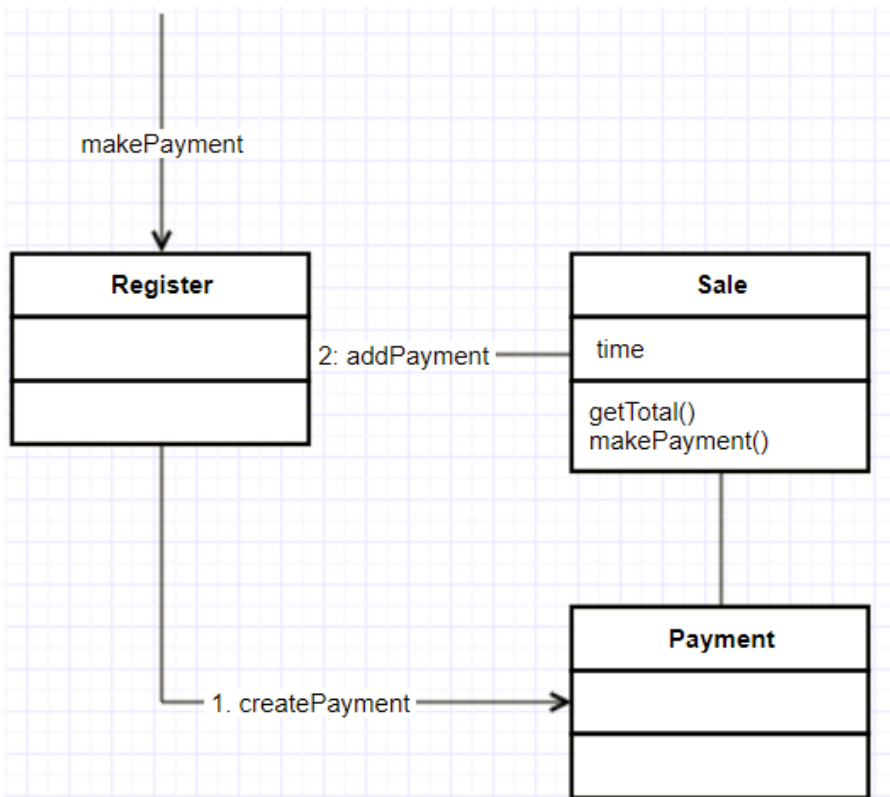
# HIGH COHESION

How are the operations of any element functionally related?

- Related responsibilities are placed into one manageable unit.
- Correlated to SRP!

**Benefits**

- Easily understandable and maintainable.
- Code reuse
- Low coupling

# COHESIVE?

# LACK OF COHESION (LCOM)

LCOM measures the **dissimilarity** of methods in a class by instance variable or attributes.

- *Functional cohesion* - the design unit (module) performs a single well-defined function or achieves a single goal.

- *Sequential cohesion* - the design unit performs more than one function, but these functions occur in an order prescribed by the specification, i.e. they are strongly related.

- *Communication cohesion* -  a design unit performs multiple functions, but all are targeted on the same data.

# LCOM CONT'D

- *Procedural cohesion* - a design unit performs multiple functions that are procedurally related. The code in each module represents a single piece of functionality defining a control sequence of activities.

- *Temporal cohesion* - a design unit performs more than one function, and they are related only by the fact that they must occur within the same time span (ex. a design that combines all data initialization into one unit and performs all initialization at the same time even though it may be defined and utilized in other design units).

- *Logical cohesion* - a design unit that performs a series of similar functions (ex. the Java class java.lang.Math)

# LCOM4

LCOM4 measures the number of *"connected components"* in a class.

**Definition.** A connected component is a set of related methods (and class-level variables).

Methods **a** and **b** are related if:
- they both access the same class-level variable, or
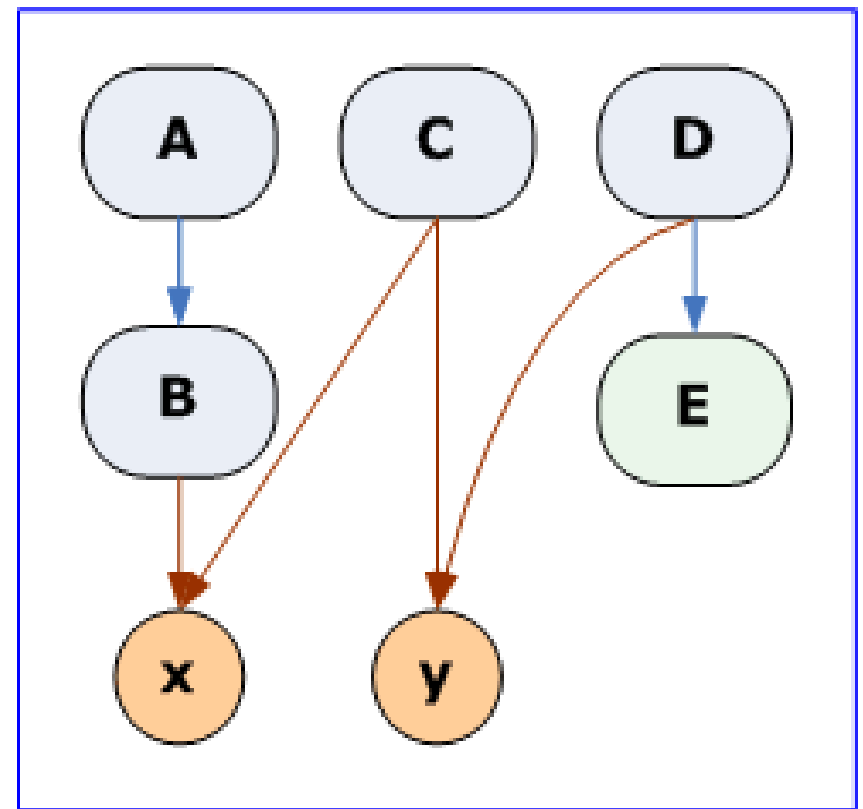- **a** calls **b**, or **b** calls **a**.

**There should be only one connected component in each class.**

If there are 2 or more components, the class should be split into so many smaller classes.

# EXAMPLE
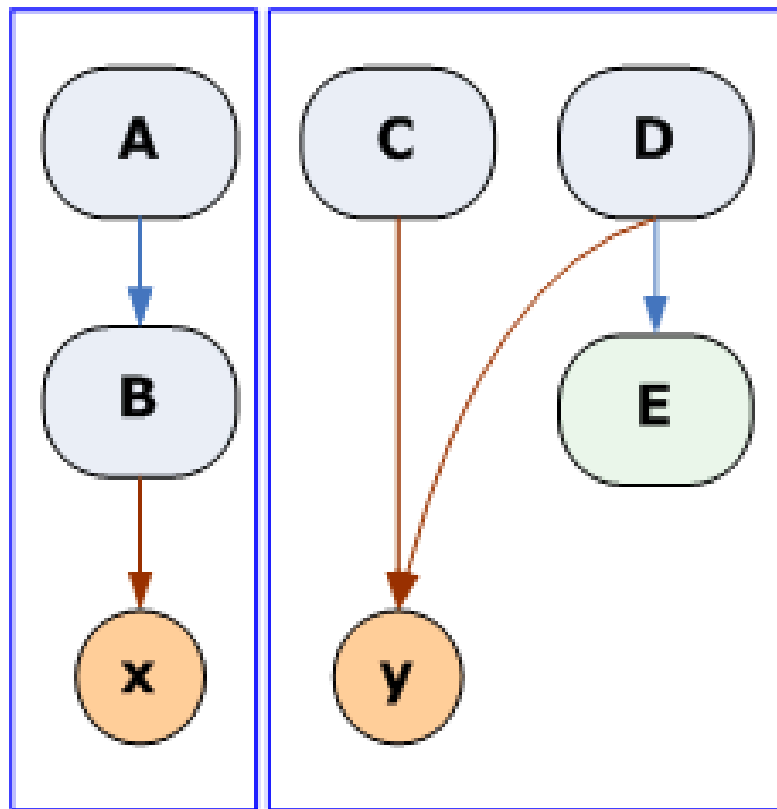


LCOM4 = 2

LCOM4 = 1

# LAW OF DEMETER

**Weak Form**

Inside of a method M of a class C, data can be accessed and messages can be sent to only the following objects:

- **this** and **super**
- **data members (attributes)** of class C
- **parameters** of the method M
- **objects created** within M
  - by calling directly a constructor
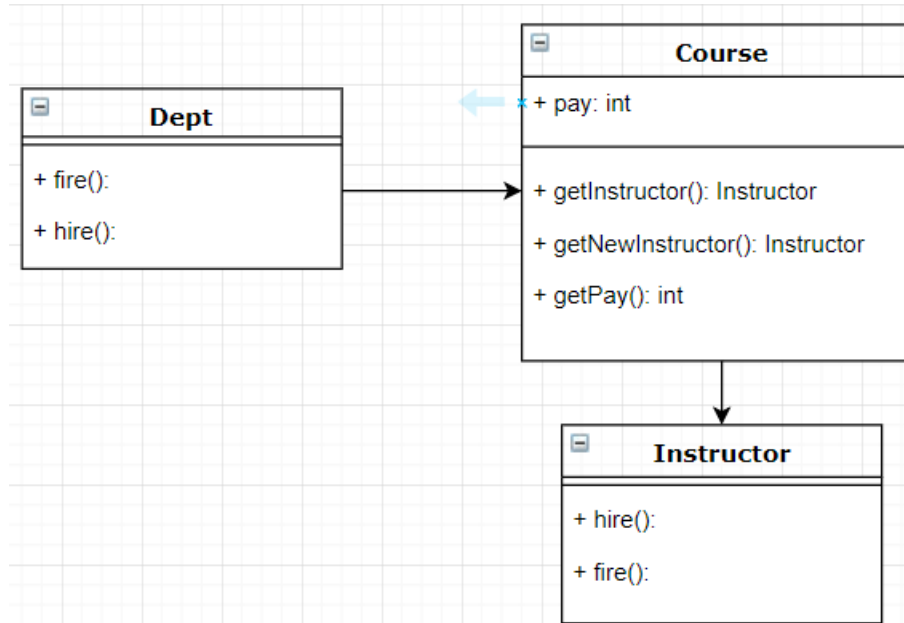  - by calling a method that creates the object
- **global variables**

**Strong Form**

In addition to the Weak Form, you are not allowed to access directly inherited members

# LOD EXAMPLE

```
class Demeter {
      private A a;

      public void example(B b)
      {
            C c;
            c = func();
            b.invert();                    passed parameters
            a = new A();
            a.setActive();                 attribute
            c.print();                     local variable
      }
}
```
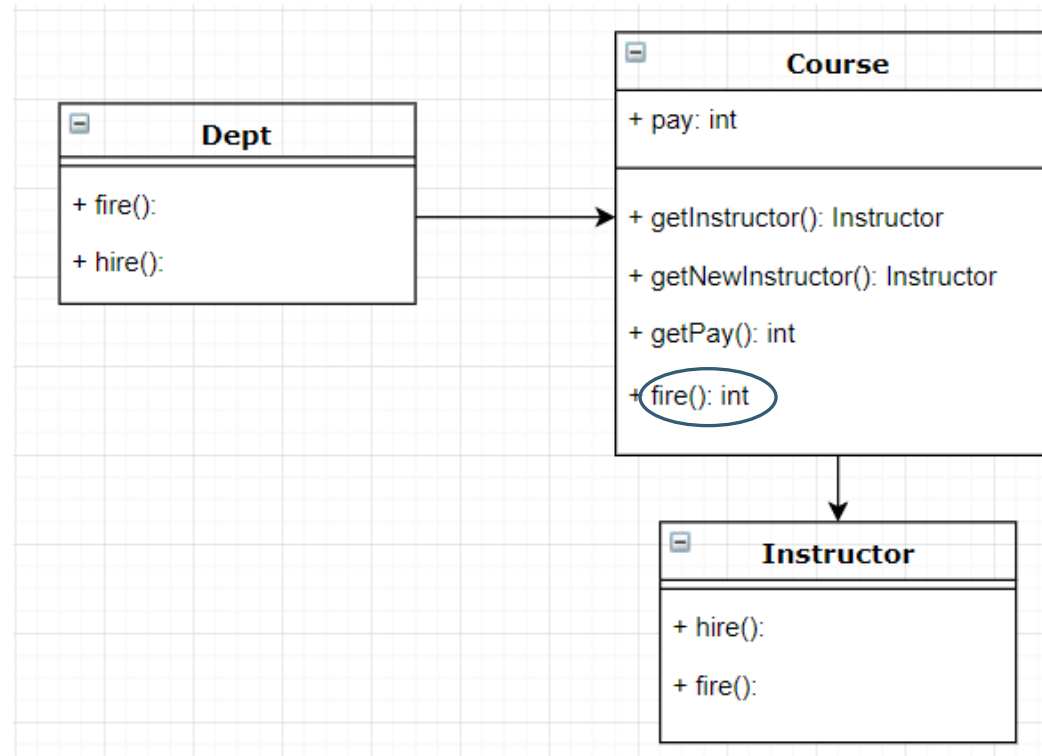
# LOD COUNTER EXAMPLE



```
class Course
{
    Instructor boring = new Instructor();
    int pay = 5;
    public Instructor getInstructor() { return boring; }
    public Instructor getNewInstructor() {return new Instructor(); }
    public int getPay() {return pay; }
}

class Dept {
 Course test = new Course();
 public void fire() { test.getInstructor().fire(); }
 public void hire() { test.getNewInstructor().hire(); }
 public int raisePay() { return test.getpay() + 10;}
}
```

# COUNTEREXAMPLE CORRECTED



```
class Course {
 Instructor boring = new Instructor();
 int pay = 5;
 public Instructor fire(){ boring.fire(); }
 public Instructor getNewInstructor() { return new Instructor();}
 public int getPay() { return pay ; }
}

class Dept {
 Course test = new Course();
 public void fire() {test.fire();}
     public void hire() {test.getNewInstructor().hire();}
     public int raisePay() { return  test.getpay() + 10;}
}
```

# LOD FOR CHILDREN

*You can play with yourself.*

*You can play with your own toys*

*You can play with toys that were given to you.*

*You can play with toys you've made yourself.*

# LOD BENEFITS

Coupling Control
- reduces data coupling

Information hiding
- prevents from retrieving subparts of an object

Information restriction
- restricts the use of methods that provide information

Few Interfaces
- restricts the classes that can be used in a method

Explicit Interfaces
- states explicitly which classes can be used in a method

# ACCEPTABLE LOD VIOLATIONS

If **optimization** requires violation
- Speed or memory restrictions

If module accessed is a **fully stabilized "Black Box"**
- No changes to interface can <u>reasonably</u> be expected due to extensive testing, usage, etc.

Otherwise, do not violate this law!!
- Long-term costs will be very prohibitive

# POS FINAL DESIGN
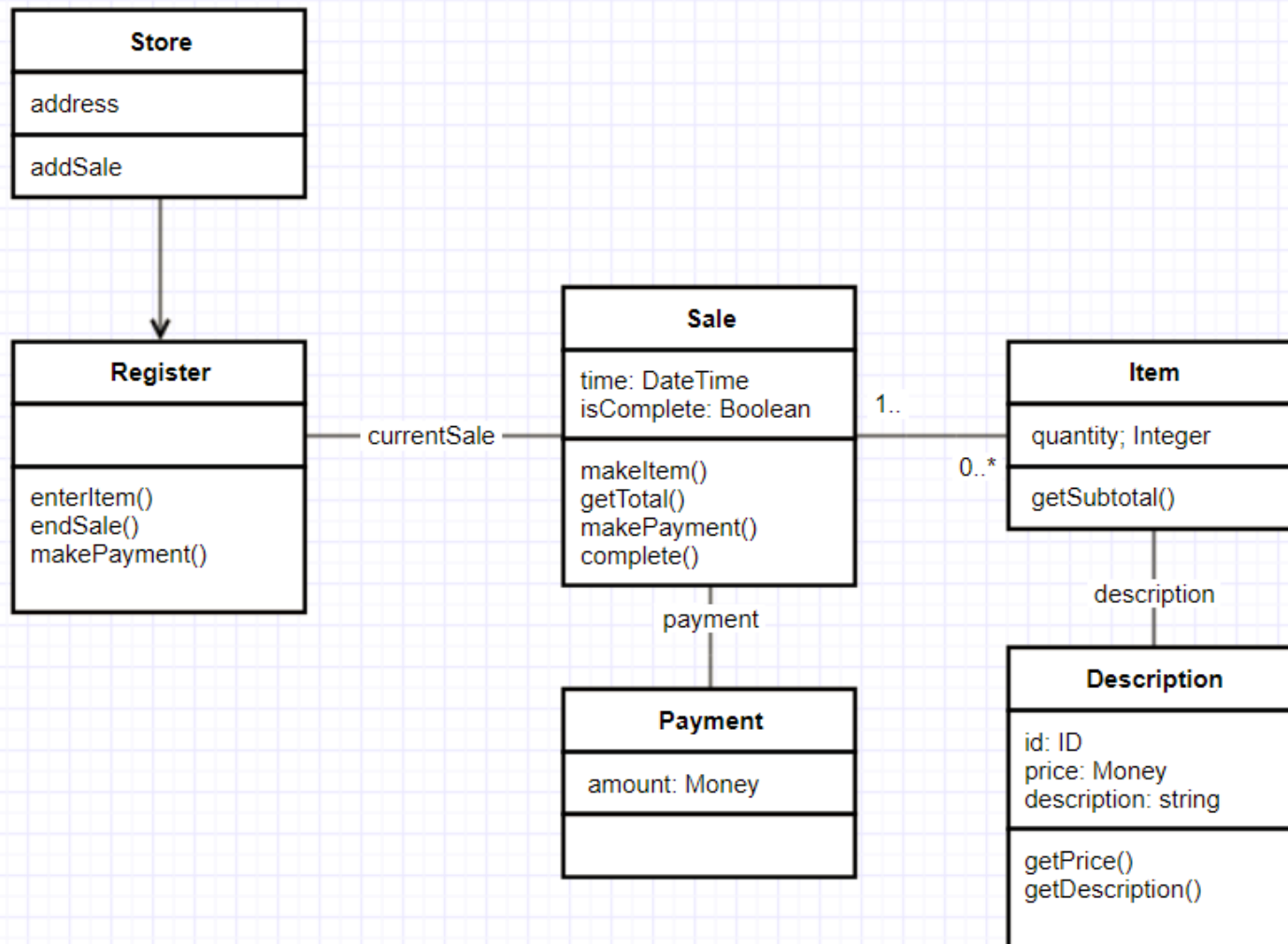
# HIGHER-LEVEL DESIGN

Dealing with *large-scale systems*
- team of developers, rather than an individual

Classes are a valuable but not sufficient mechanism
- too *fine-grained* for organizing a large scale design
- need mechanism that impose a higher level of order

**Packages**
- a logical grouping of declarations that can be imported in other programs
- containers for a group of classes (UML)
- reason at a higher-level of abstraction

# ISSUES OF HIGHER-LEVEL DESIGN

**Goal**

- *partition* the classes in an application according to some *criteria* and then *allocate* those partitions to packages

**Issues**

- What are the best partitioning criteria?
- What principles govern the design of packages?
  - *creation* and *dependencies* between packages

**Approach**

- Define principles that govern package design
  - the creation and interrelationship and use of packages

# PRINCIPLES OF OO HIGHER-LEVEL DESIGN

**Cohesion Principles**

- Reuse/Release Equivalency Principle (REP)
- Common Reuse Principle (CRP)
- Common Closure Principle (CCP)

**Coupling Principles**

- Acyclic Dependencies Principle (ADP)
- Stable Dependencies Principle (SDP)
- Stable Abstractions Principle (SAP)

# WHAT IS REALLY REUSABILITY ?

Does copy-paste mean reusability?

- Disadvantage: **You** own that copy!

  - you must change it, fix bugs.

  - eventually the code diverges

R.Martin's Definition:

- *I reuse code if, and only if, I never need to look at the source-code*

- treat reused code like a *product* $\Rightarrow$ don't have to maintain it

Clients (re-users) may decide to use a newer version of a component release

# REUSE/RELEASE EQUIVALENCY PRINCIPLE (REP)

**The granule of reuse is the granule of release.** *Only components that are released through a tracking system can be efficiently reused. [R. Martin]*

*Either all the classes in a package are reusable or none of it is! [R. Martin]*

# WHAT DOES THIS MEAN?

Reused code = product
- Released, named and maintained by the producer.


Programmer = client
- Doesn't have to maintain reused code
- Doesn't have to name reused code
- May choose to use an older/newer release

# THE COMMON REUSE PRINCIPLE

*All classes in a package [library] should be reused together. If you reuse one of the classes in the package, you reuse them all.* [R.Martin]

*If I depend on a package, I want to depend on every class in that package!* [R.Martin]

# WHAT DOES THIS MEAN?

Criteria for grouping classes in a package:
- Classes that tend to be **reused** together.

Packages have physical representations (shared libraries, DLLs, assembly)
- Changing just one class in the package

⇒ re-release the package

⇒ revalidate the application that uses the package.

# COMMON CLOSURE PRINCIPLE (CCP)

*The classes in a package should be closed against the same kinds of changes.*

*A change that affects a package affects all the classes in that package*

[R. Martin]

# WHAT DOES THIS MEAN?

Another criteria of grouping classes: **Maintainability**!

- Classes that tend to change together for the same reasons
- Classes highly dependent

**SRP at packages level!**

# REUSE VS. MAINTENANCE

REP and CRP makes life easier for **reuser**
- packages very small

CCP makes life easier for **maintainer**
- larger packages

**Packages are not fixed in stone**
- early in project focus on CCP
- later when architecture stabilizes: focus on REP and CRP

# ACYCLIC DEPENDENCIES PRINCIPLES (ADP)

*The dependency structure for released component must be a Directed Acyclic Graph (DAG).There can be no cycles.* [R. Martin]
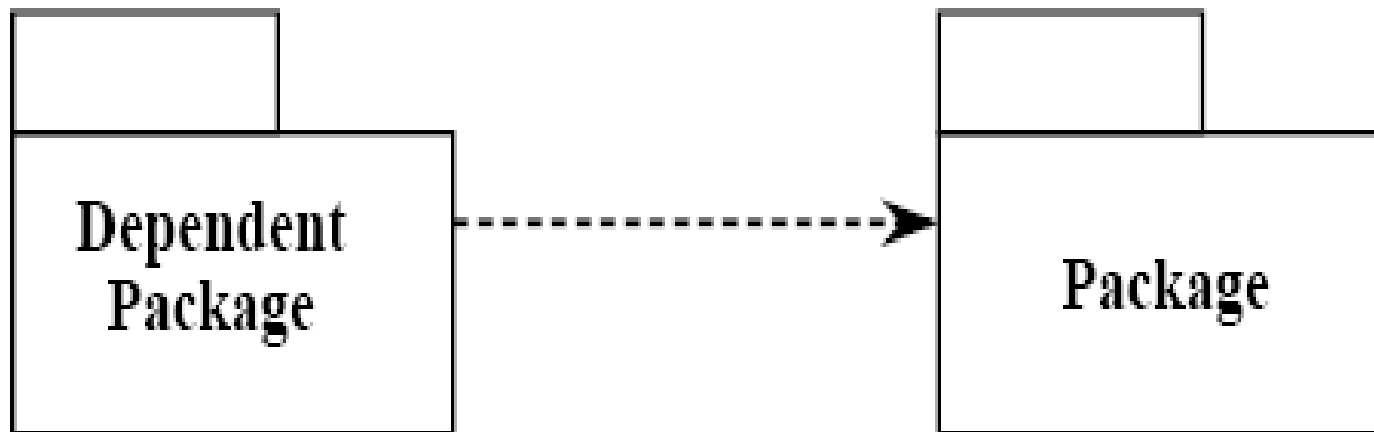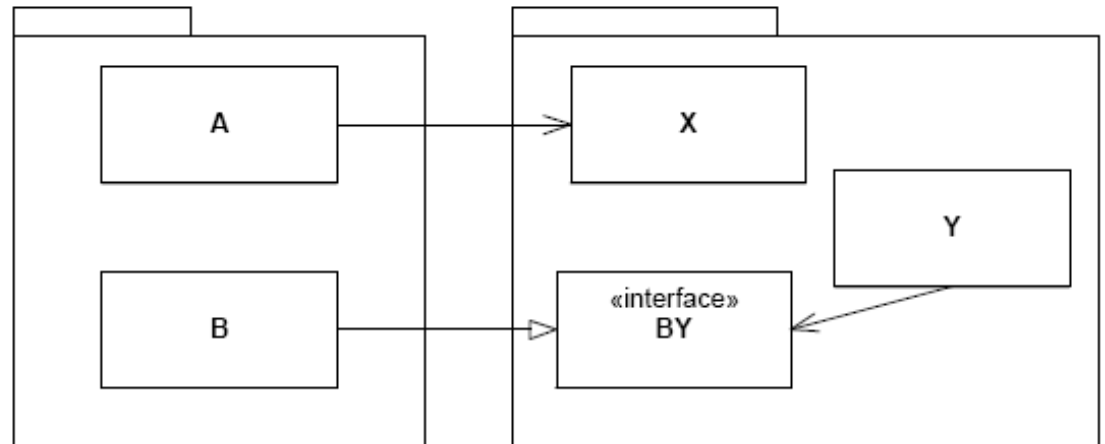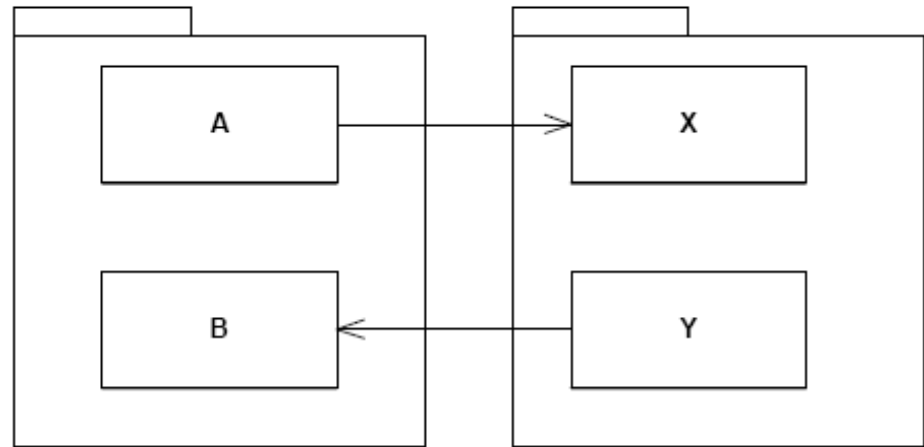
# DEPENDENCY GRAPHS
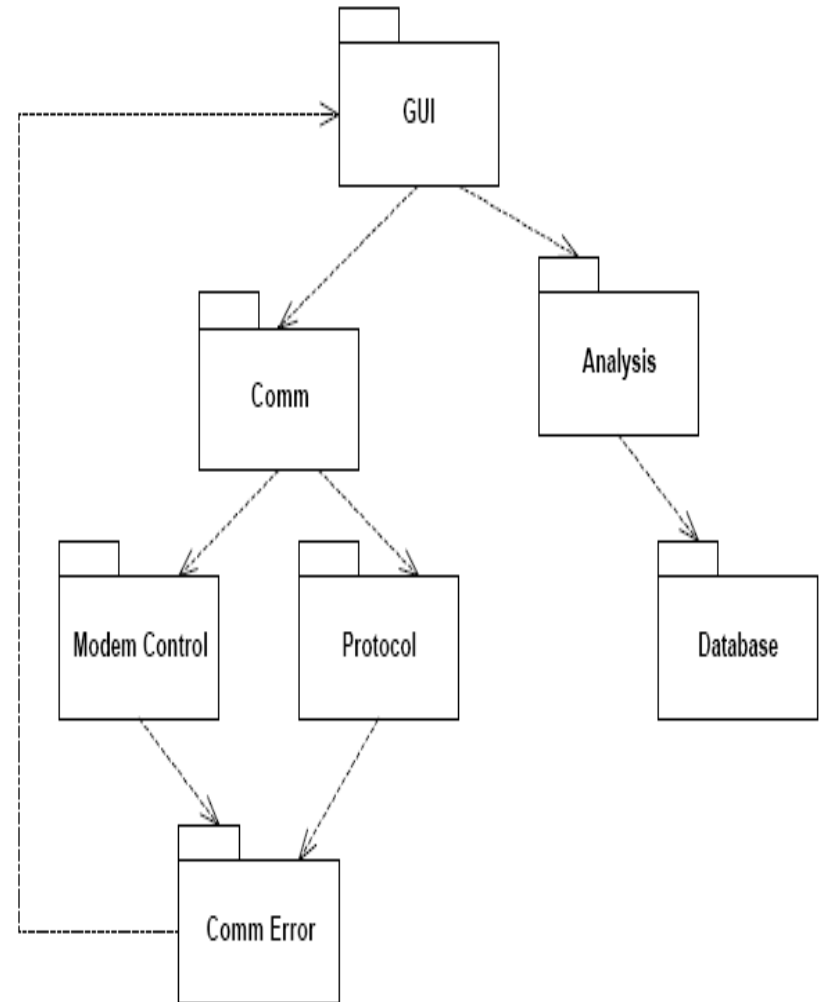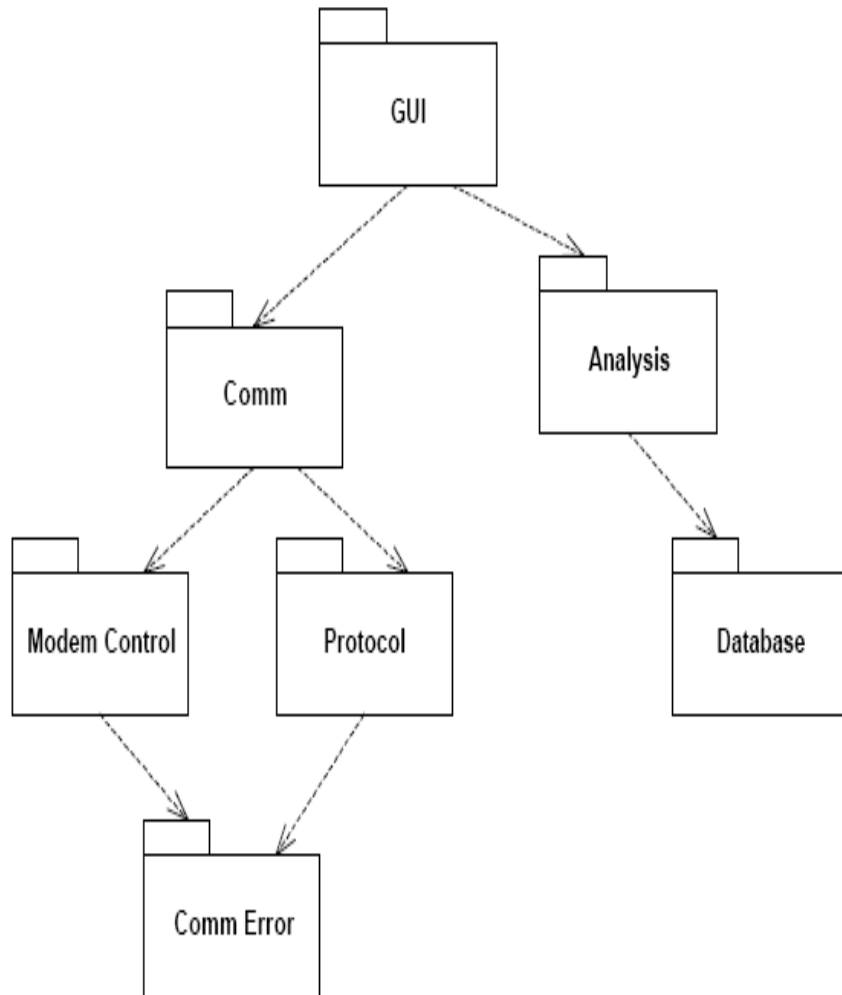
# BREAKING THE CYCLE
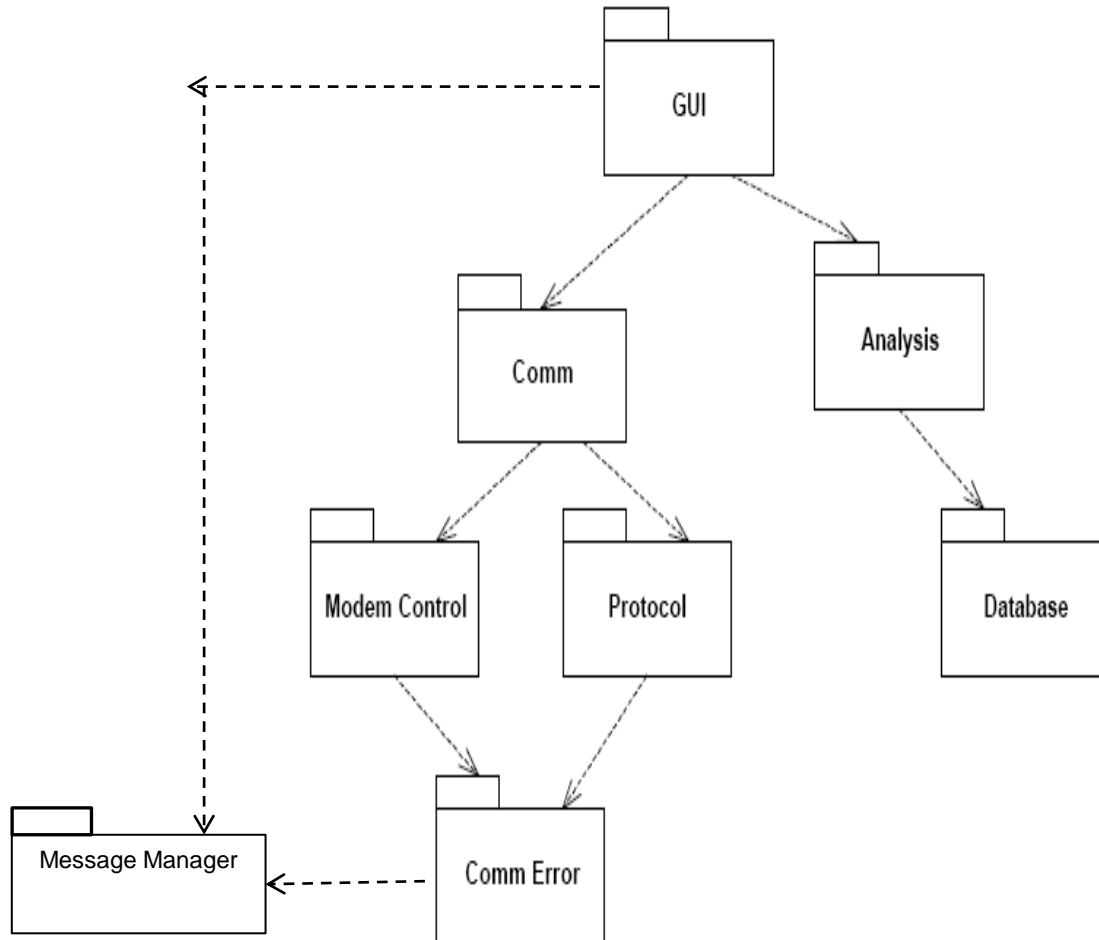


Take out of the GUI package the classes that Comm Error depend on
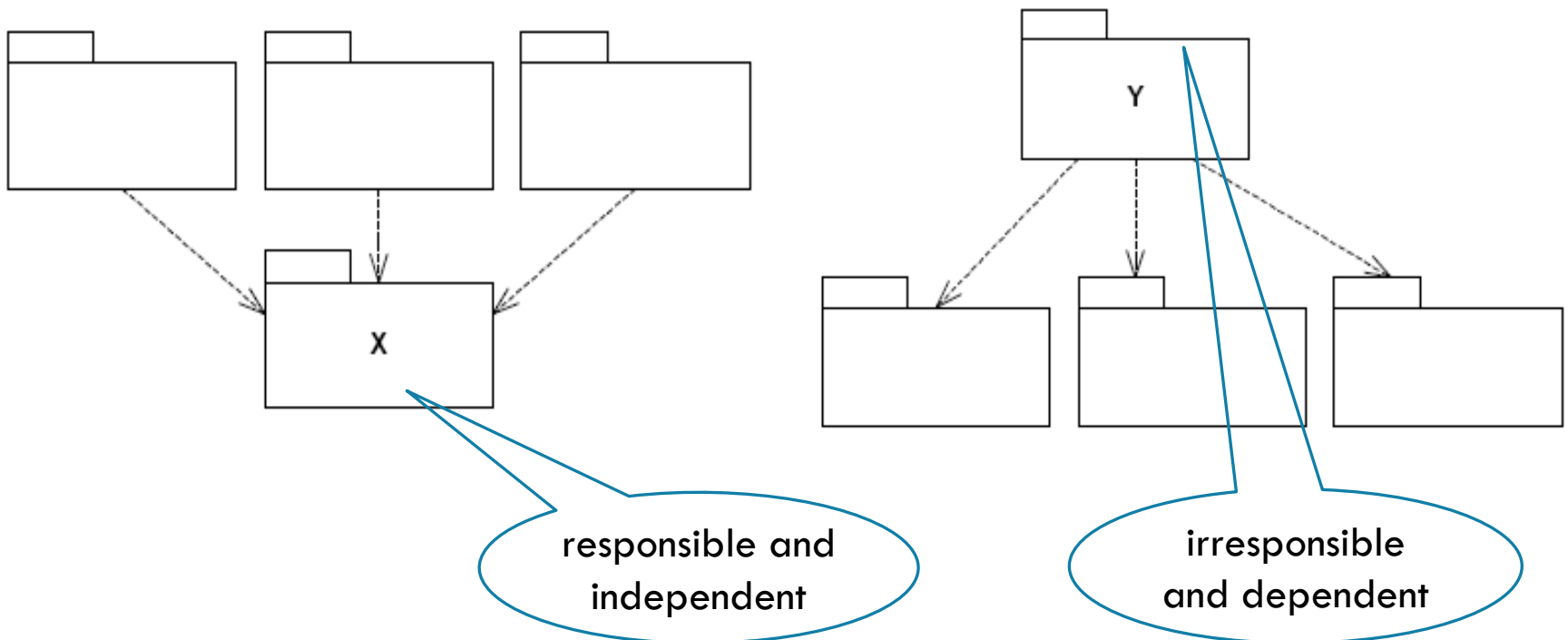
=> Add a new package (i.e. Message Manager)

# BREAKING THE CYCLE

DIP + ISP

# STABILITY

Stability is related to the amount of work in order to make a change.



responsible and independent

irresponsible and dependent

Driven by Responsibility and Independence

# STABILITY METRICS

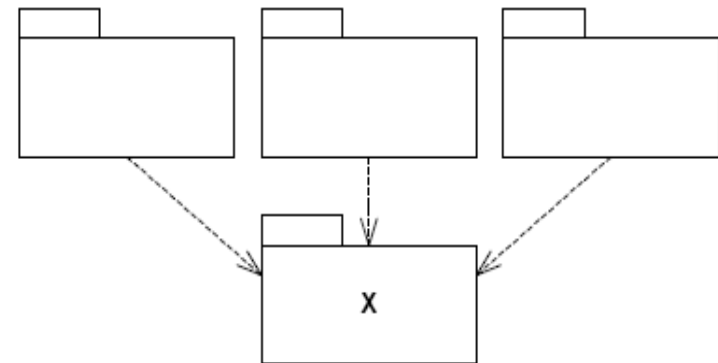***Ca*** – Afferent coupling (**incoming dependencies**)

- How responsible am I?

**Ce** – Efferent coupling (**outgoing dependencies**)

- How dependant am I?

Instability $I = Ce/(Ca+Ce)$

Example for X:

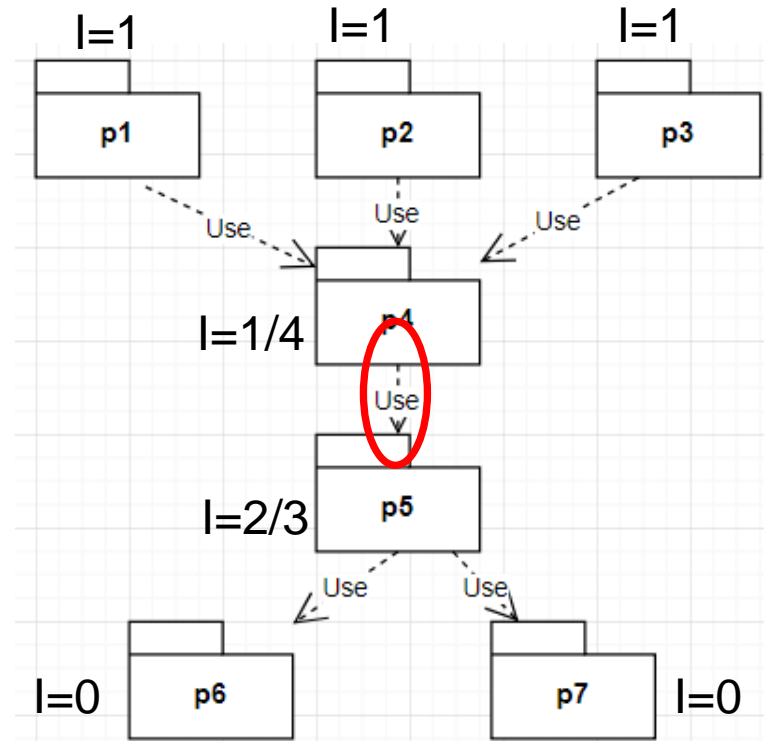$Ca = 3$, $Ce = 0$ => $I = 0$ (very st

# STABLE DEPENDENCY PRINCIPLE (SDP)

Depend in the direction of stability.

What does this mean?

- Depend upon packages whose I is lower than yours.

**Counter-example**

# WHERE TO PUT HIGH-LEVEL DESIGN?

High-level architecture and design decisions don't change often
- shouldn't be volatile $\Rightarrow$ place them in stable packages
- design becomes hard to change $\Rightarrow$ *inflexible design*

**How can a totally stable package (I = 0) be flexible enough to withstand change?**

Answer: ***The Open-Closed Principle***
- Classes that can be extended without modifying them $\Rightarrow$ Abstract Classes

# STABLE ABSTRACTIONS PRINCIPLE (SAP)

Stable packages should be abstract packages.

What does this mean?

- Stable packages should be depended upon
- Flexible packages should be dependent
- OCP => Stable packages should be highly abstract

# ABSTRACTNESS METRICS

$Nc$ = number of classes in the package

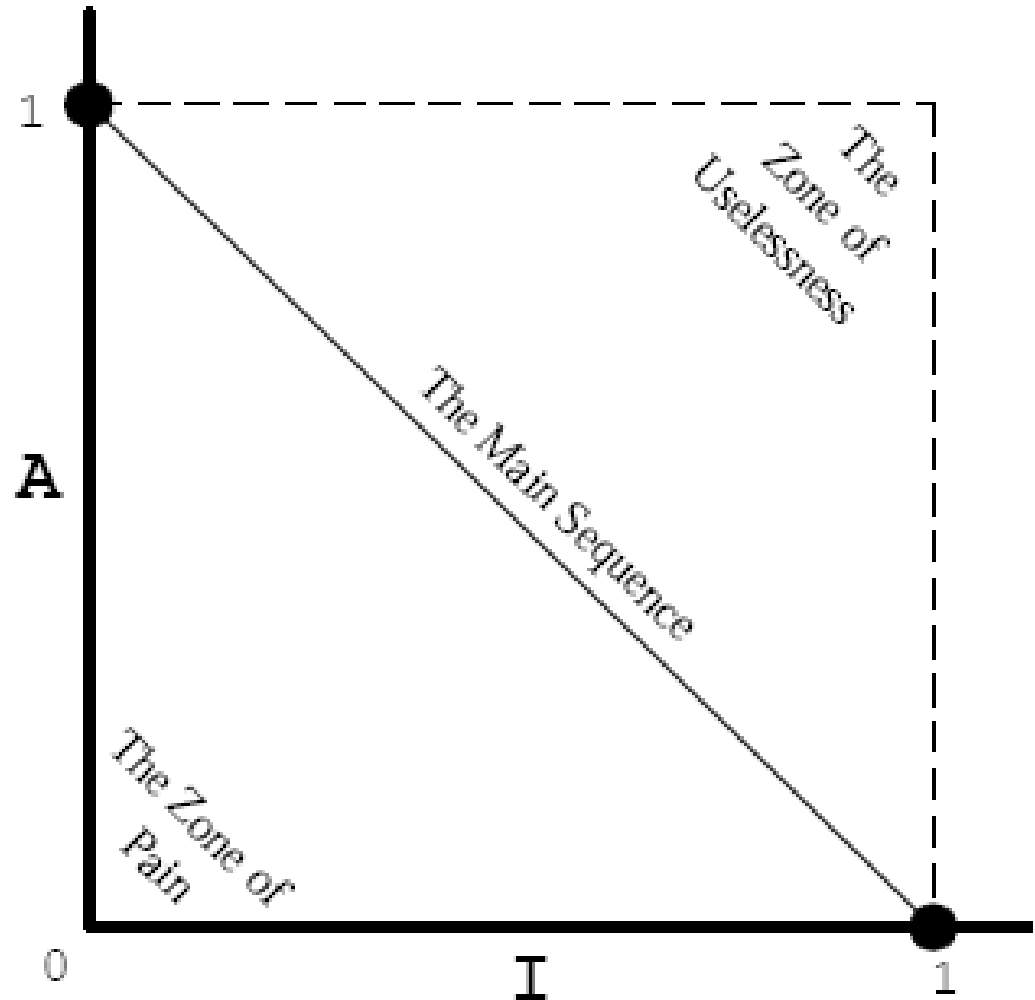$Na$ = number of abstract classes in the package

Abstractness $A = Na/Nc$

Example:

- *All classes are concrete Na = 0 => A = 0*

What about hybrid classes?

# THE MAIN SEQUENCE

*I* should increase as *A* decreases

# THE MAIN SEQUENCE

Zone of Pain

- highly stable and concrete $\Rightarrow$ rigid
- famous examples:
  - database-schemas (volatile and highly depended-upon)
  - concrete utility libraries (instable but non-volatile)

Zone of Uselessness

- instable and abstract $\Rightarrow$ useless
  - no one depends on those classes

Main Sequence

- maximizes the distance between the zones we want to avoid
- depicts the balance between abstractness and stability.

# WRAP-UP

Principles for good class design related to

- Assigning Responsibilities (GRASP)
- Package design
  - Coupling
  - Cohesion