

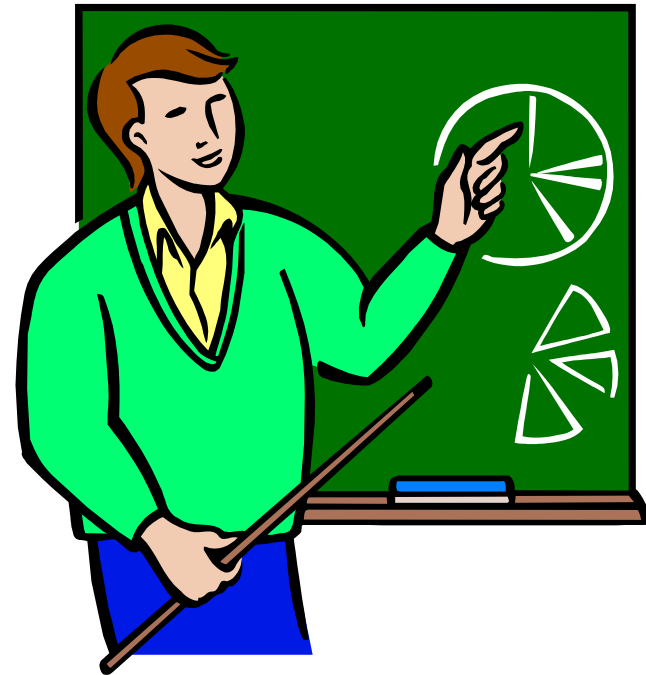
OO Design Principles and UML

By Pradeep LN

Primary OO Concepts

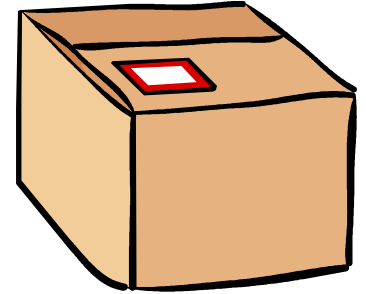
Primary Object Oriented Concepts

- Abstraction
- Encapsulation
- Inheritance
- Cohesion
- Coupling



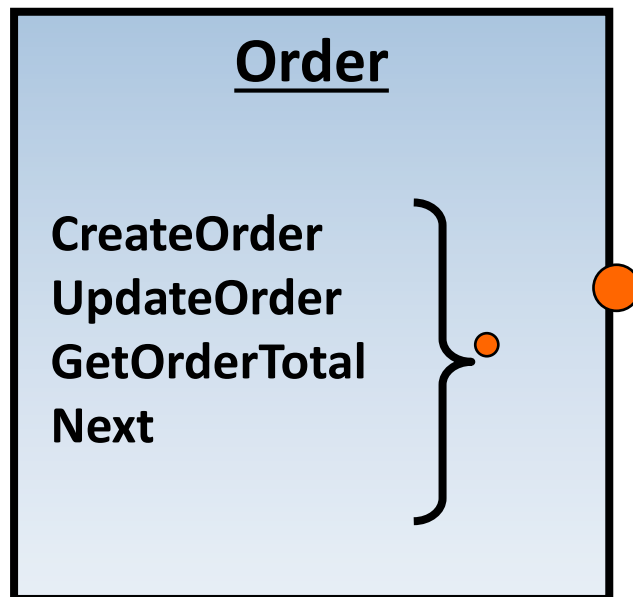
Abstraction

Public View of an Object



- *Abstraction* is used to minimize complexity for the class user
 - By allowing him focus on the essential characteristics
 - By hiding the details of implementation
- Simply put, abstraction is nothing but a process of ensuring that class users are not exposed to details which they do not need (or use).

Abstraction - Example

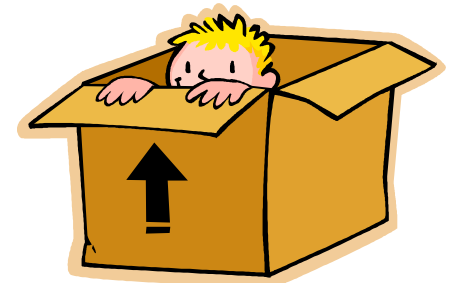
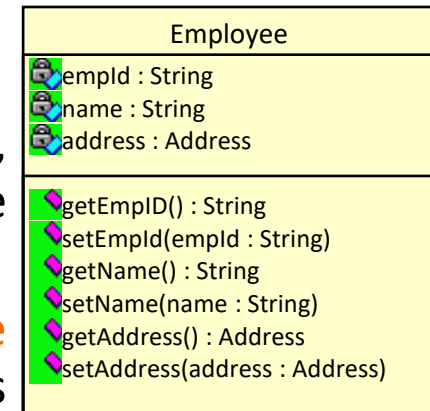


**“What should an
Order object do for
the class user?”**

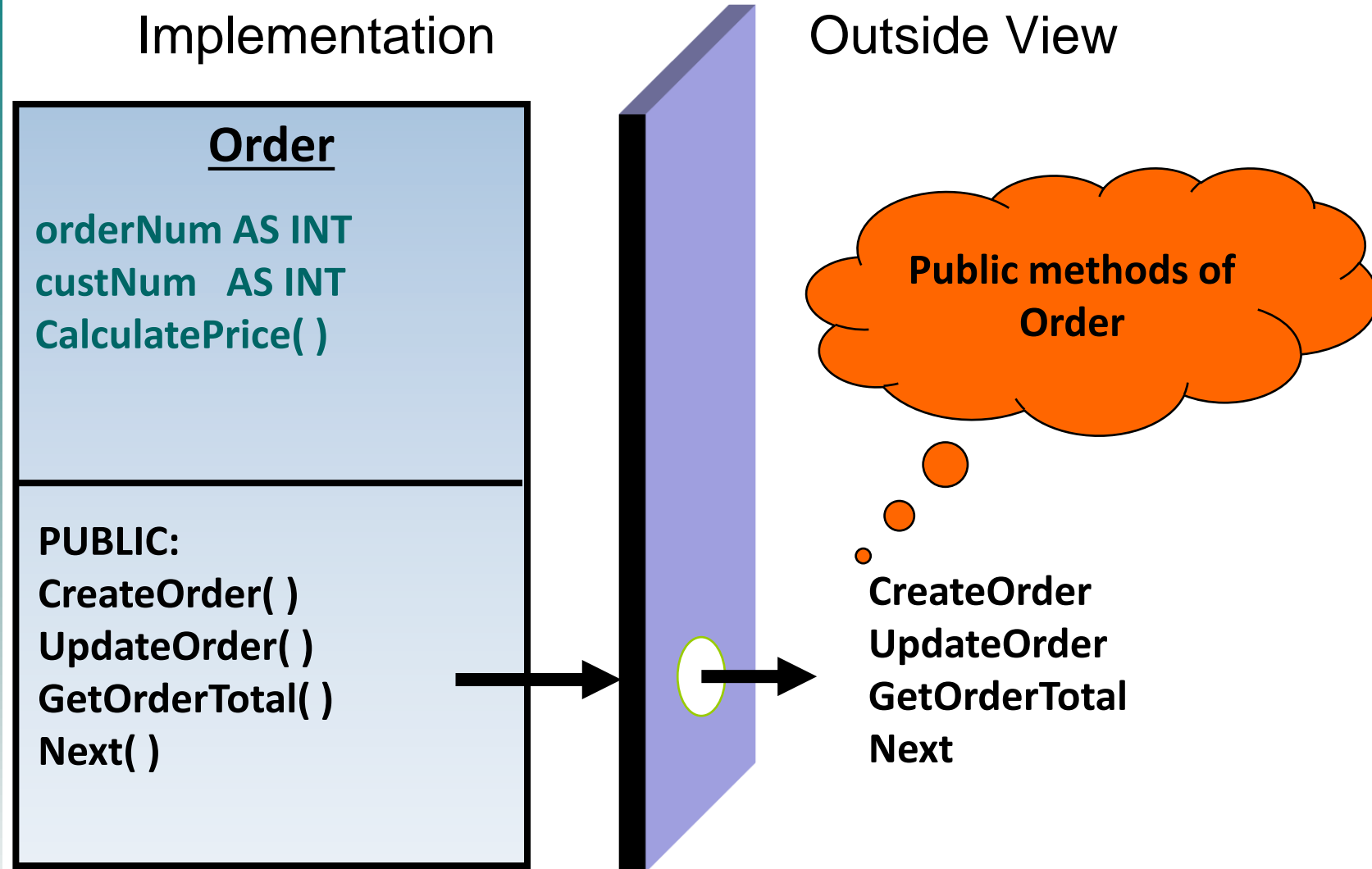
Encapsulation

Hide Implementation Details

- Encapsulation is
 - The grouping of related ideas into a single unit, which can thereafter be referred to by a single name.
 - The process of compartmentalizing ‘the elements of an abstraction’ that constitute its **structure** and **behavior**.
- *Encapsulation* hides implementation
 - Promotes modular software design – data and methods together
 - Data access always done through methods
 - Often called “information hiding”
- Provides two kinds of protection:
 - State cannot be changed directly from outside
 - Implementation can change without affecting users of the object



Encapsulation - Example



Cohesion

- Cohesion is a measure of how strongly-related and focused the responsibilities of a single class are.
- If the methods that serve the given class tend to be similar in many aspects the class is said to have high cohesion. In a highly-cohesive system, code readability and the likelihood of reuse is increased, while complexity is kept manageable.
- Cohesion is decreased if:
 - The responsibilities (methods) of a class have little in common.
 - Methods carry out many varied activities, often using coarsely-grained or unrelated sets of data.

Cohesion

- Disadvantages of low cohesion (or "weak cohesion") are:
 - Increased difficulty in understanding modules.
 - Increased difficulty in maintaining a system, because logical changes in the domain affect multiple modules, and because changes in one module require changes in related modules.
 - Increased difficulty in reusing a module because most applications won't need the random set of operations provided by a module.

Coupling

- Coupling can be "low" (also "loose" and "weak") or "high" (also "tight" and "strong").
- Low coupling refers to a relationship in which one module interacts with another module through a stable interface and does not need to be concerned with the other module's internal implementation.
- With low coupling, a change in one module will not require a change in the implementation of another module.
- Low coupling is often a sign of a well-structured computer system, and when combined with high cohesion, supports the general goals of high readability and maintainability.

Coupling

- Systems that do not exhibit low coupling might experience the following developmental difficulties:
 - Change in one module forces a ripple of changes in other modules.
 - Modules are difficult to understand in isolation.
 - Modules are difficult to reuse or test because dependent modules must be included.

UML INTRODUCTION

What is UML?

- Standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems, business modeling and other non-software systems.
- The UML represents a collection of best engineering practices that have proven successful in the modeling of large and complex systems.
- The UML is a very important part of developing object oriented software and the software development process.
- The UML uses mostly graphical notations to express the design of software projects.
- Using the UML helps project teams communicate, explore potential designs, and validate the architectural design of the software.

Overview of UML Diagrams

Structural

: element of spec. irrespective of time

- Class
- Component
- Deployment
- Object
- *Composite structure*
- *Package*

Behavioral

: behavioral features of a system / business process

- Activity
- State machine
- Use case
- *Interaction*

Interaction

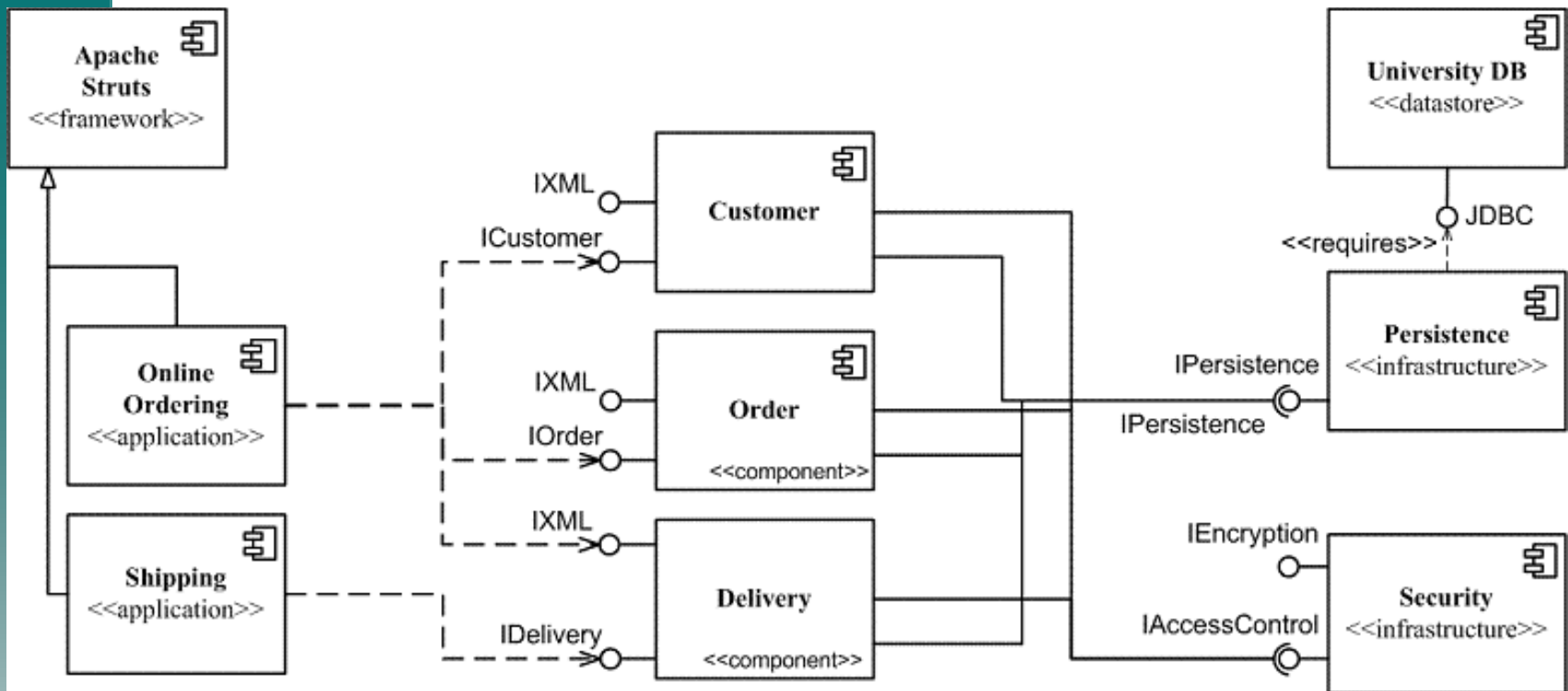
: emphasize object interaction

- Communication(collaberati on)
- Sequence
- *Interaction overview*
- *Timing*

Component diagram

UML component diagrams shows the dependencies among software components, including the classifiers that specify them (for example implementation classes) and the artifacts that implement them; such as source code files, binary code files, executable files, scripts and tables.

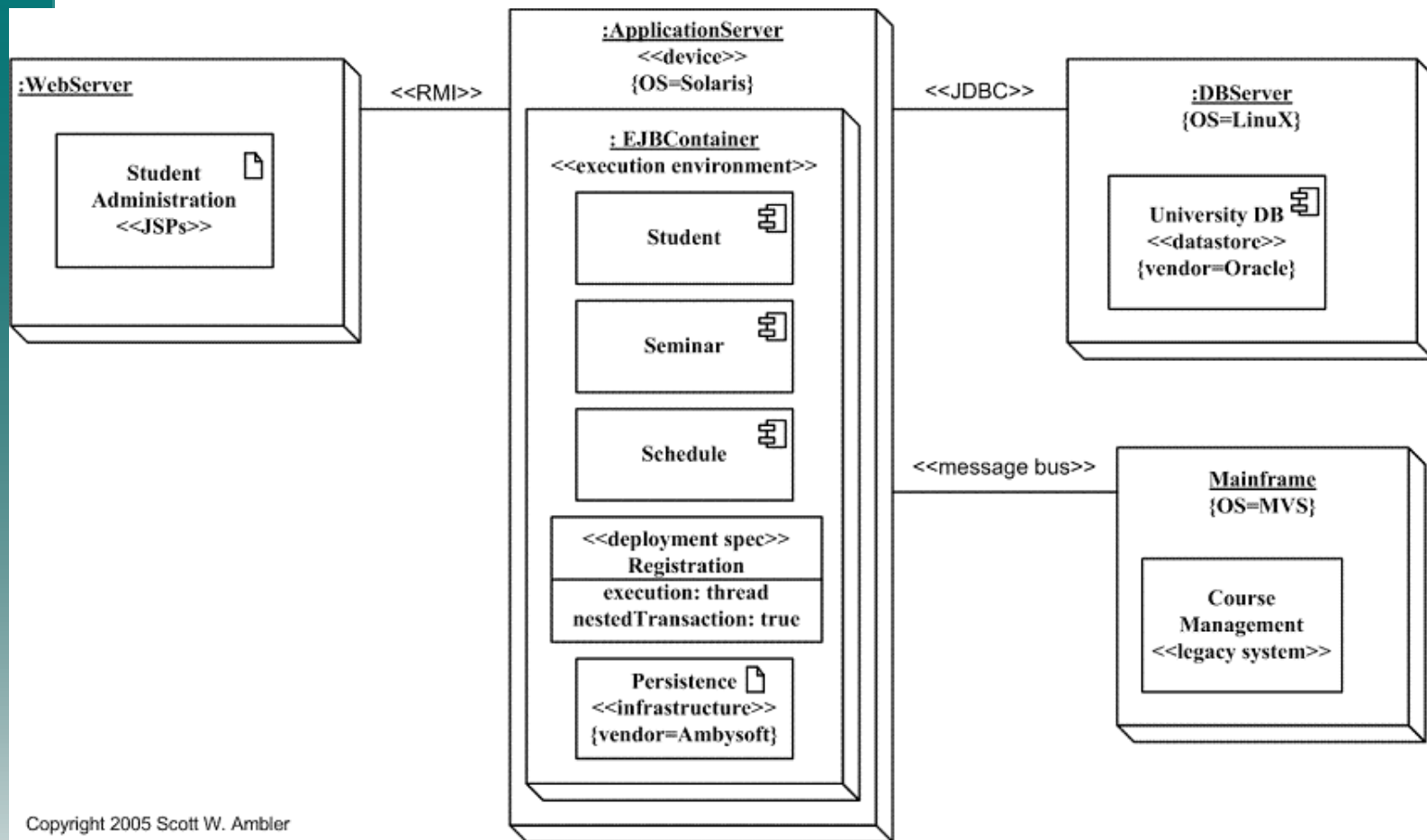
Component diagram



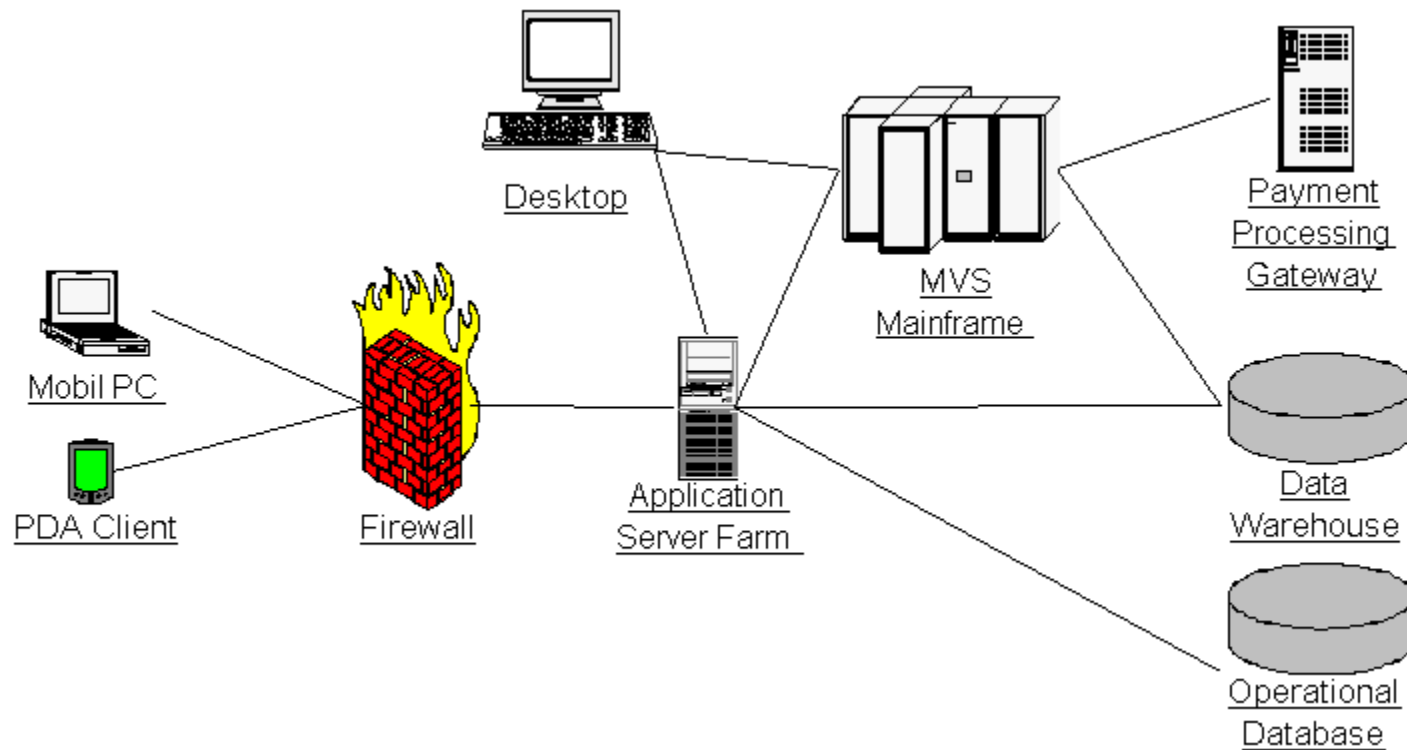
Deployment diagram

UML deployment diagram depicts a static view of the run-time configuration of hardware nodes and the software components that run on those nodes. Deployment diagrams show the hardware for your system, the software that is installed on that hardware, and the middleware used to connect the disparate machines to one another.

Deployment diagram



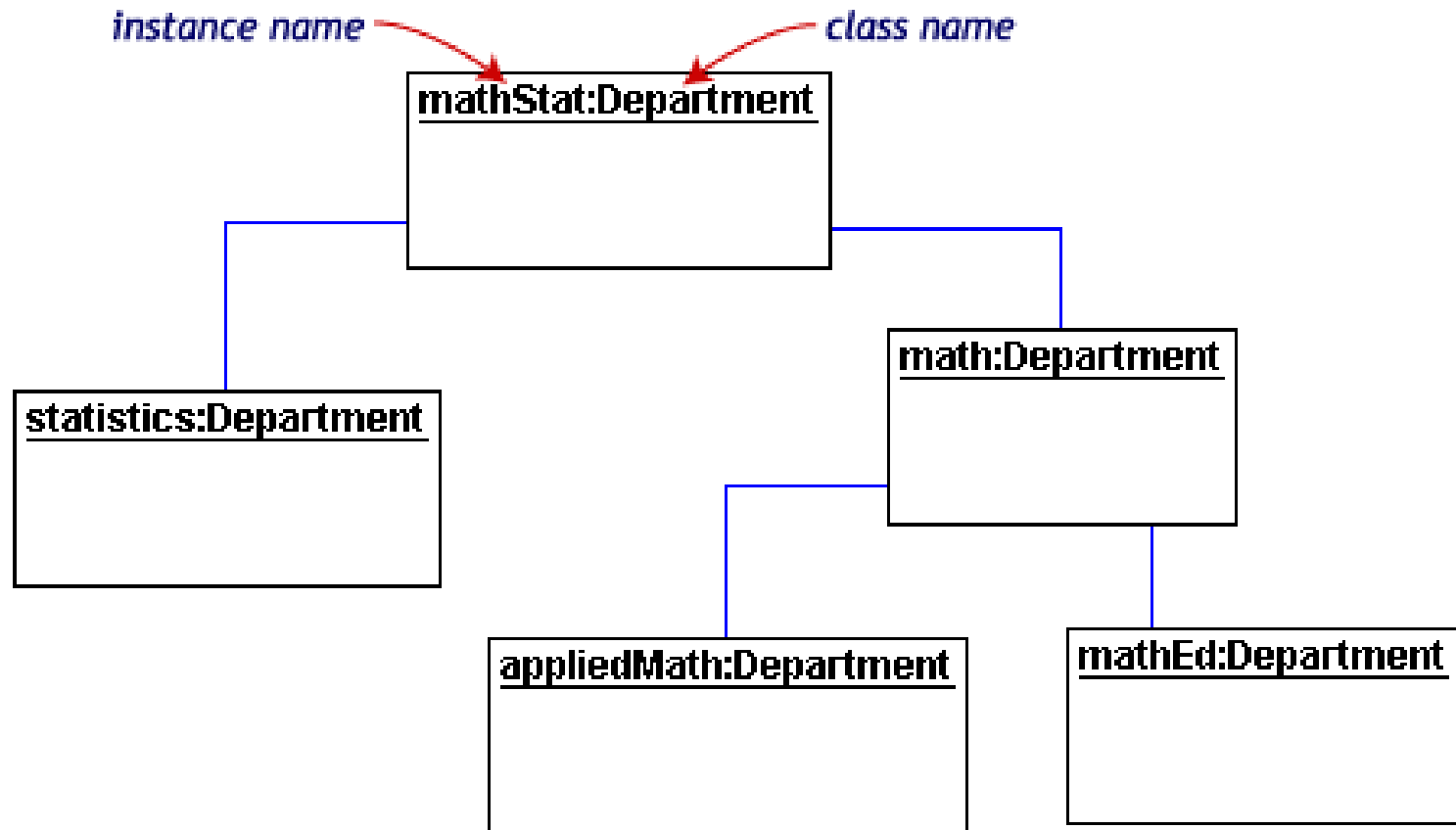
Deployment diagram



Object diagram

- UML 2 Object diagrams (instance diagrams), are useful for exploring real world examples of objects and the relationships between them. It shows instances instead of classes. They are useful for explaining small pieces with complicated relationships, especially recursive relationships.

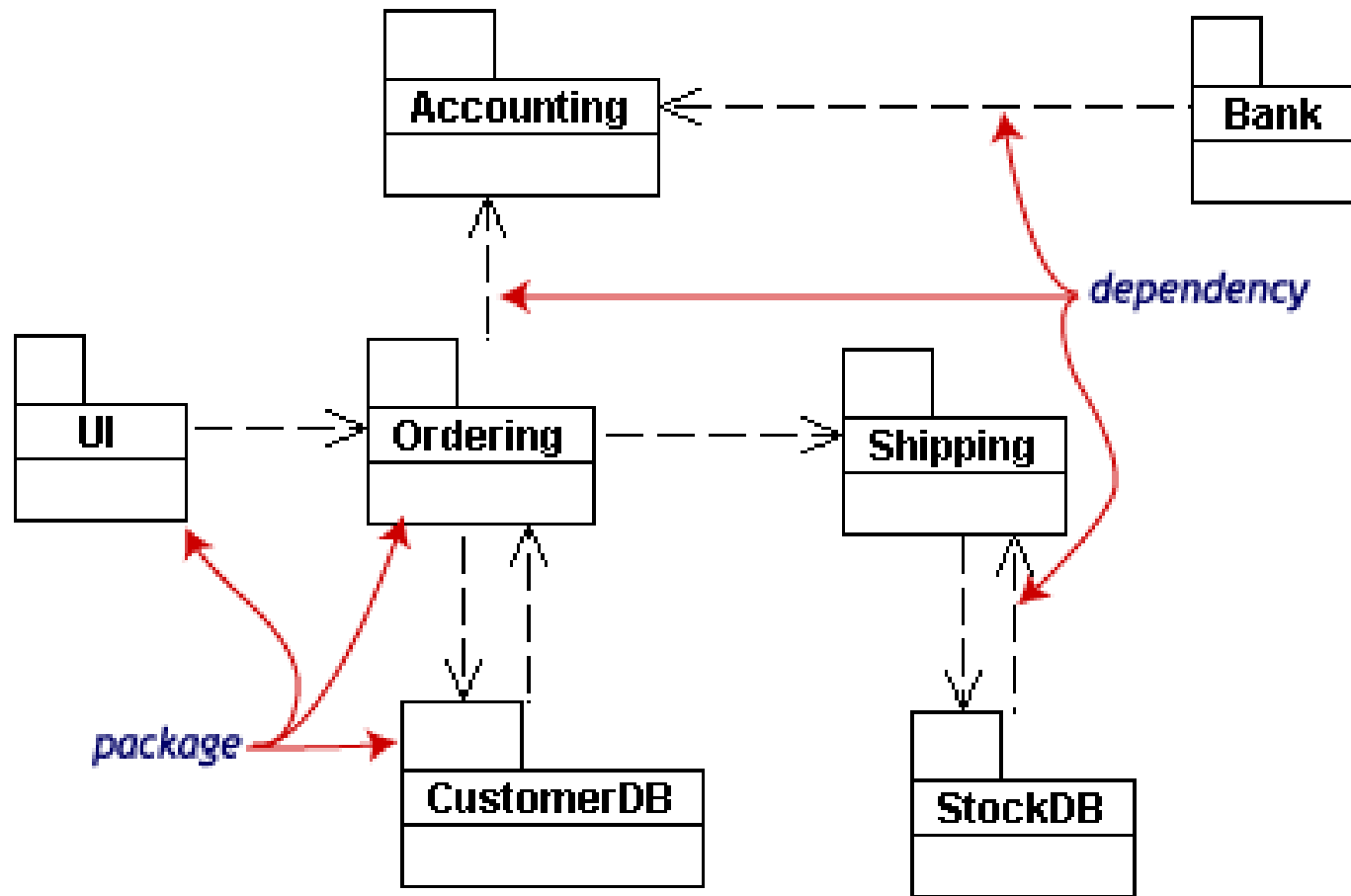
Object diagram



Package diagram

- UML 2 Package diagrams simplify complex class diagrams, it can group classes into **packages**. A package is a collection of logically related UML elements. Packages are depicted as file folders and can be used on any of the UML diagrams.

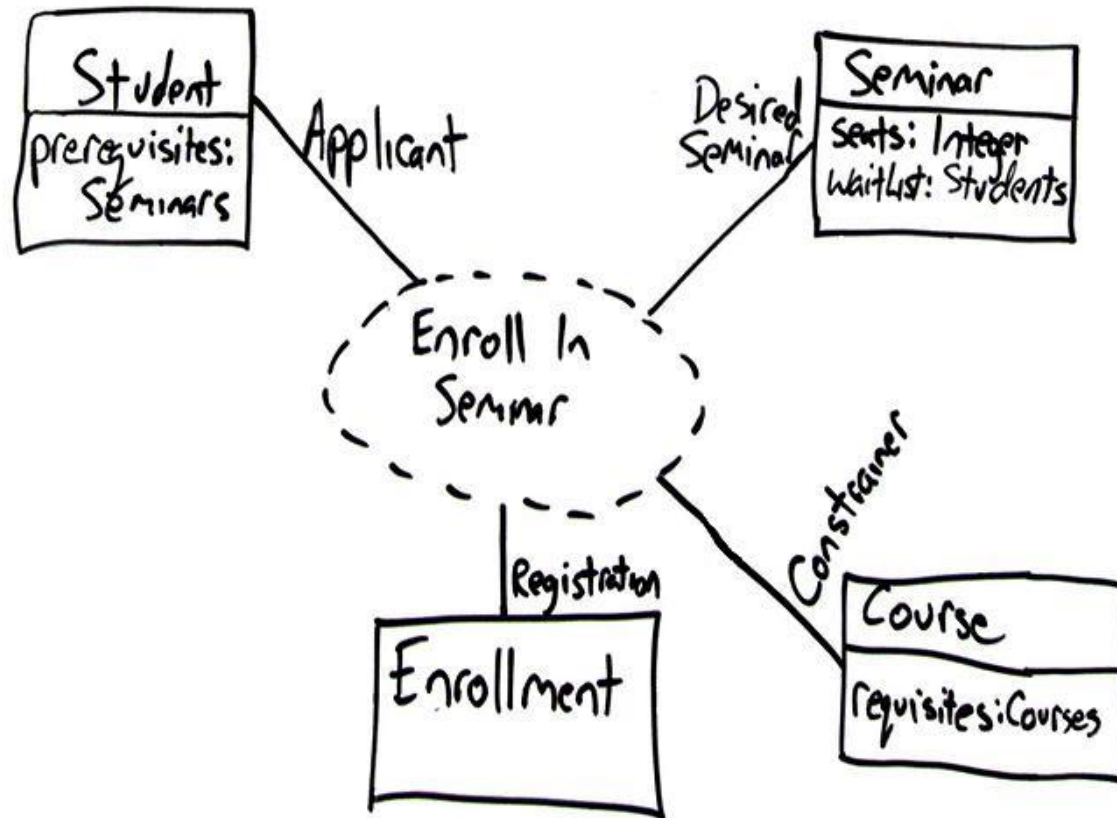
Package diagram



Composite structure diagram

- UML 2 Composite structure diagrams used to explore run-time instances of interconnected instances collaborating over communications links. It shows the internal structure (including parts and connectors) of a structured classifier or collaboration.

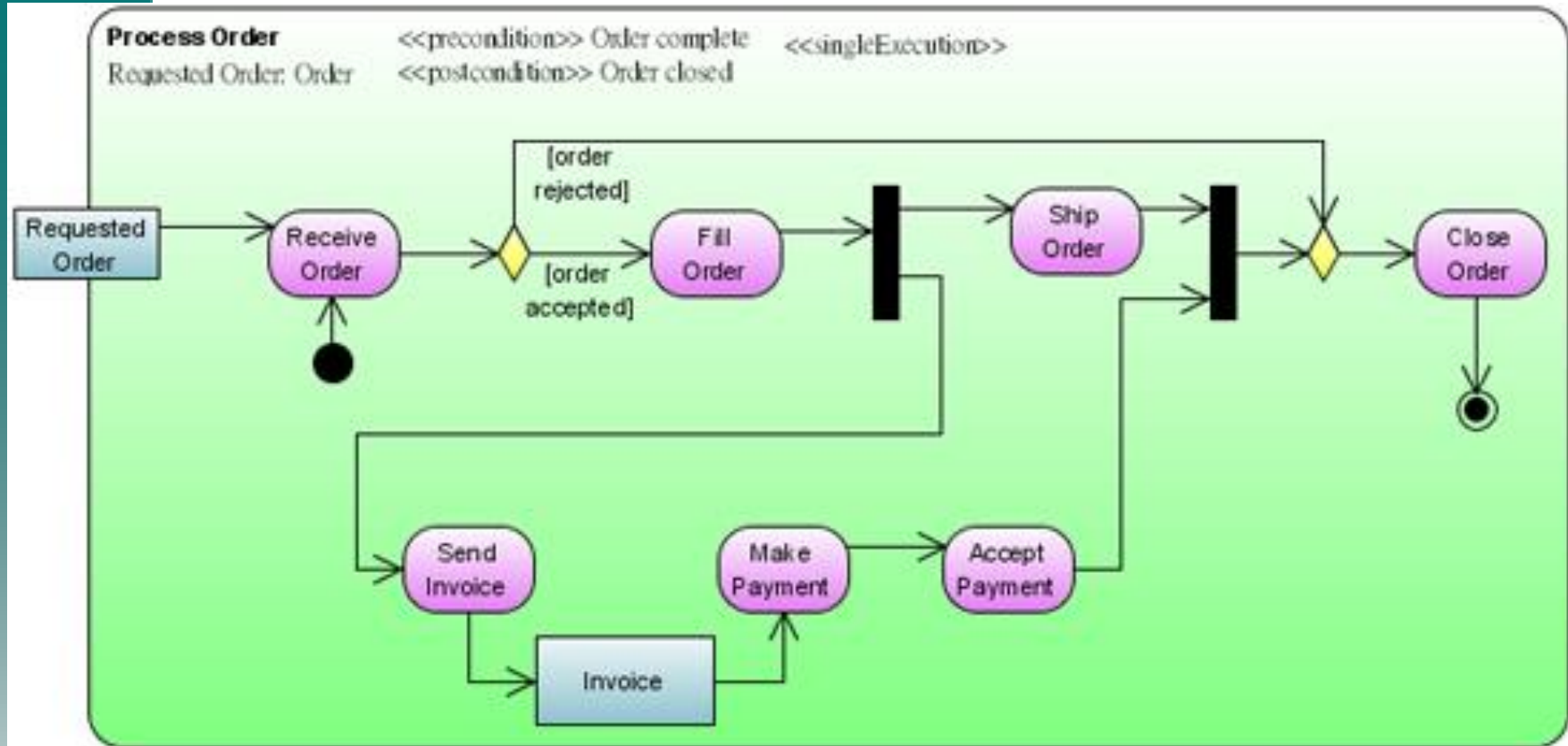
Composite structure diagram



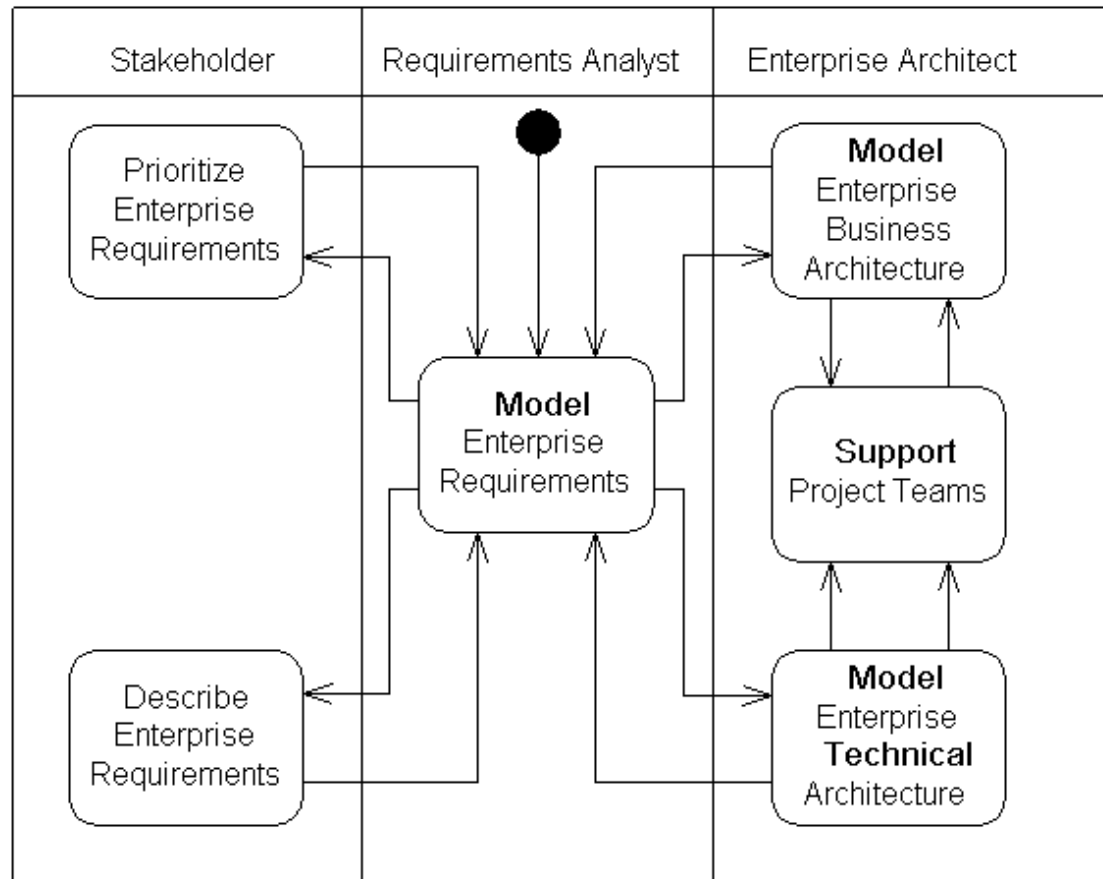
Activity diagram

- UML 2 Activity diagrams helps to describe the flow of control of the target system, such as the exploring complex business rules and operations, describing the use case also the business process. It is object-oriented equivalent of flow charts and data-flow diagrams (DFDs).

Activity diagram



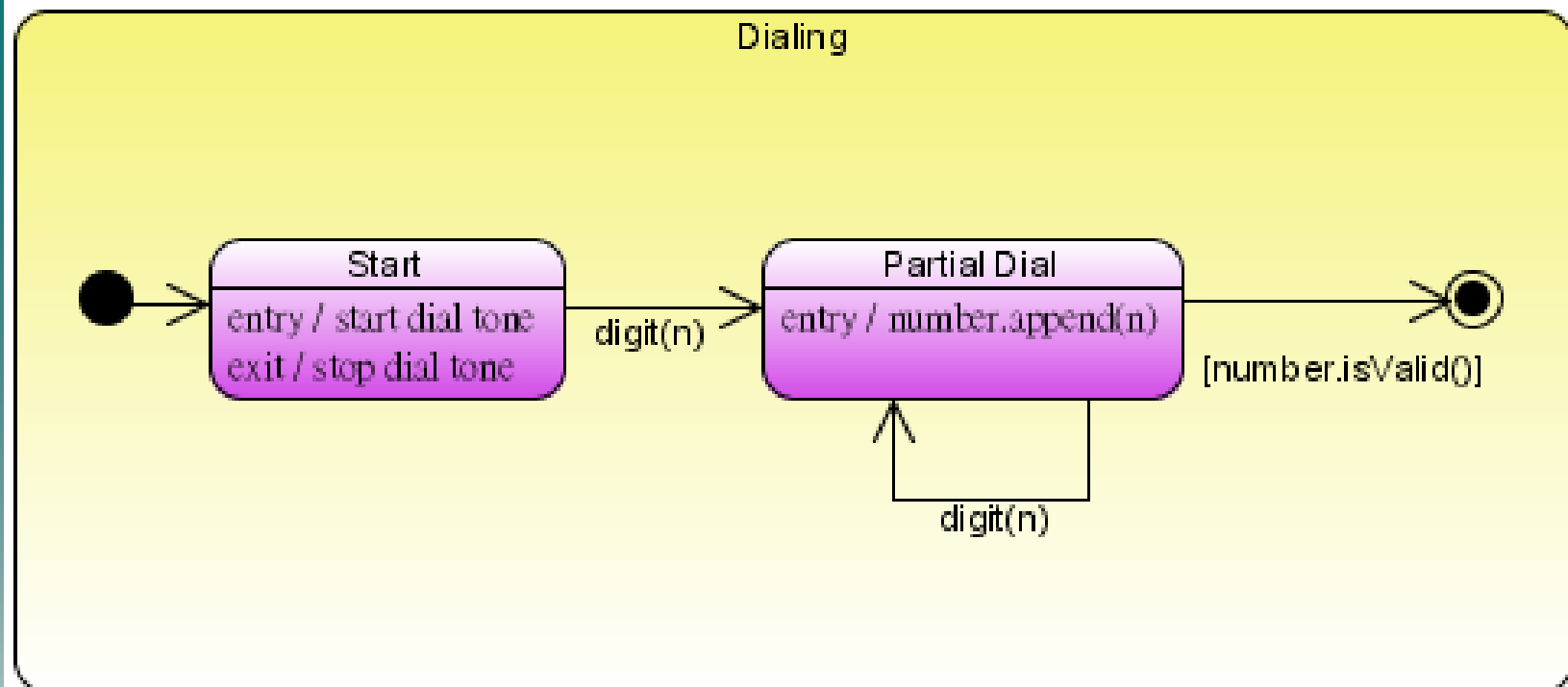
Activity diagram



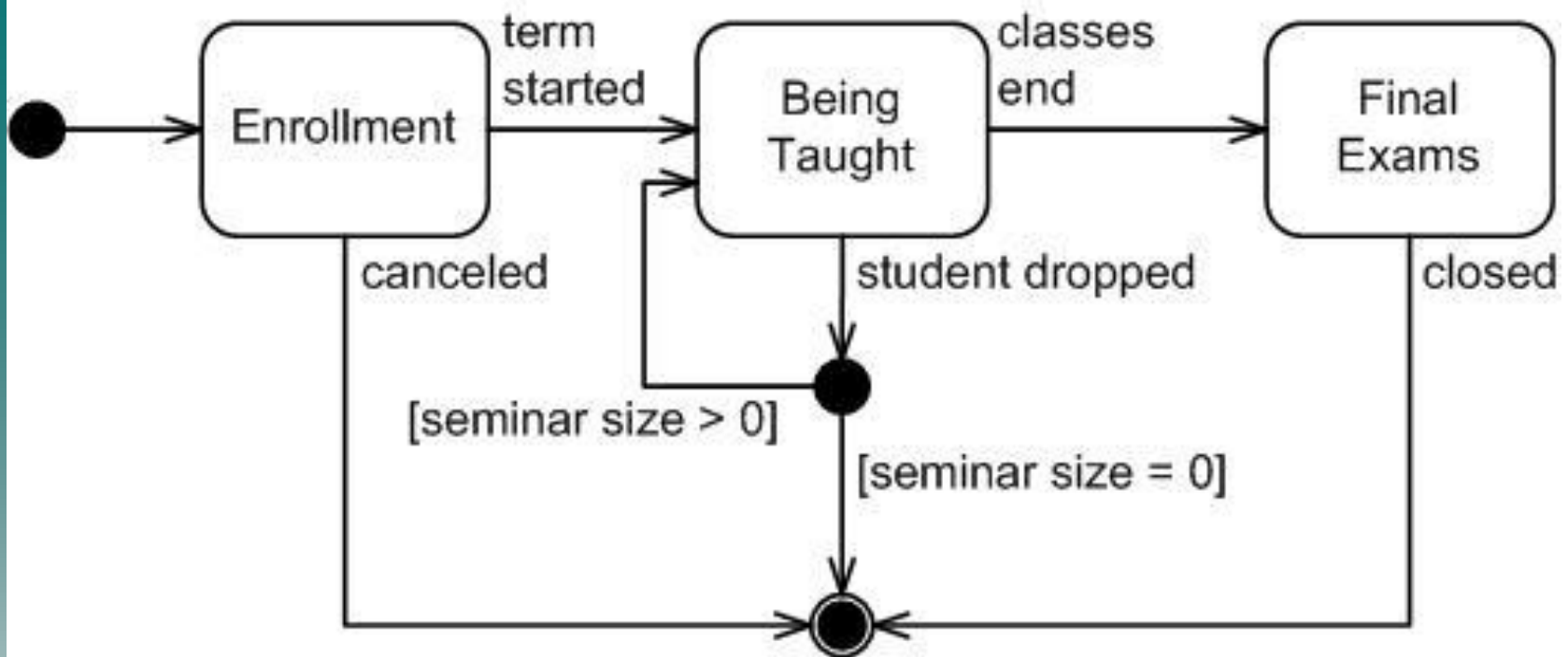
State machine diagram

- UML 2 State machine diagrams can show the different states of an entity also how an entity responds to various events by changing from one state to another. The history of an entity can best be modeled by a finite state diagram.

State machine diagram



State machine diagram

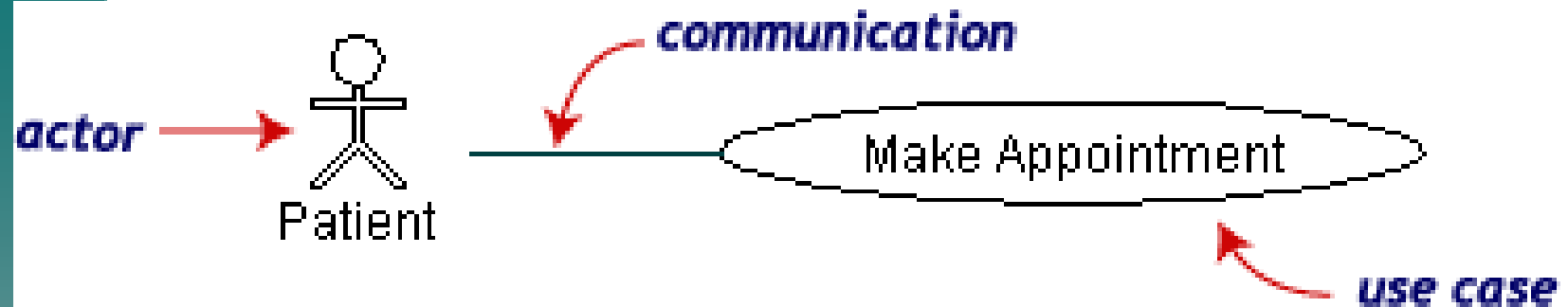


Use cases diagram

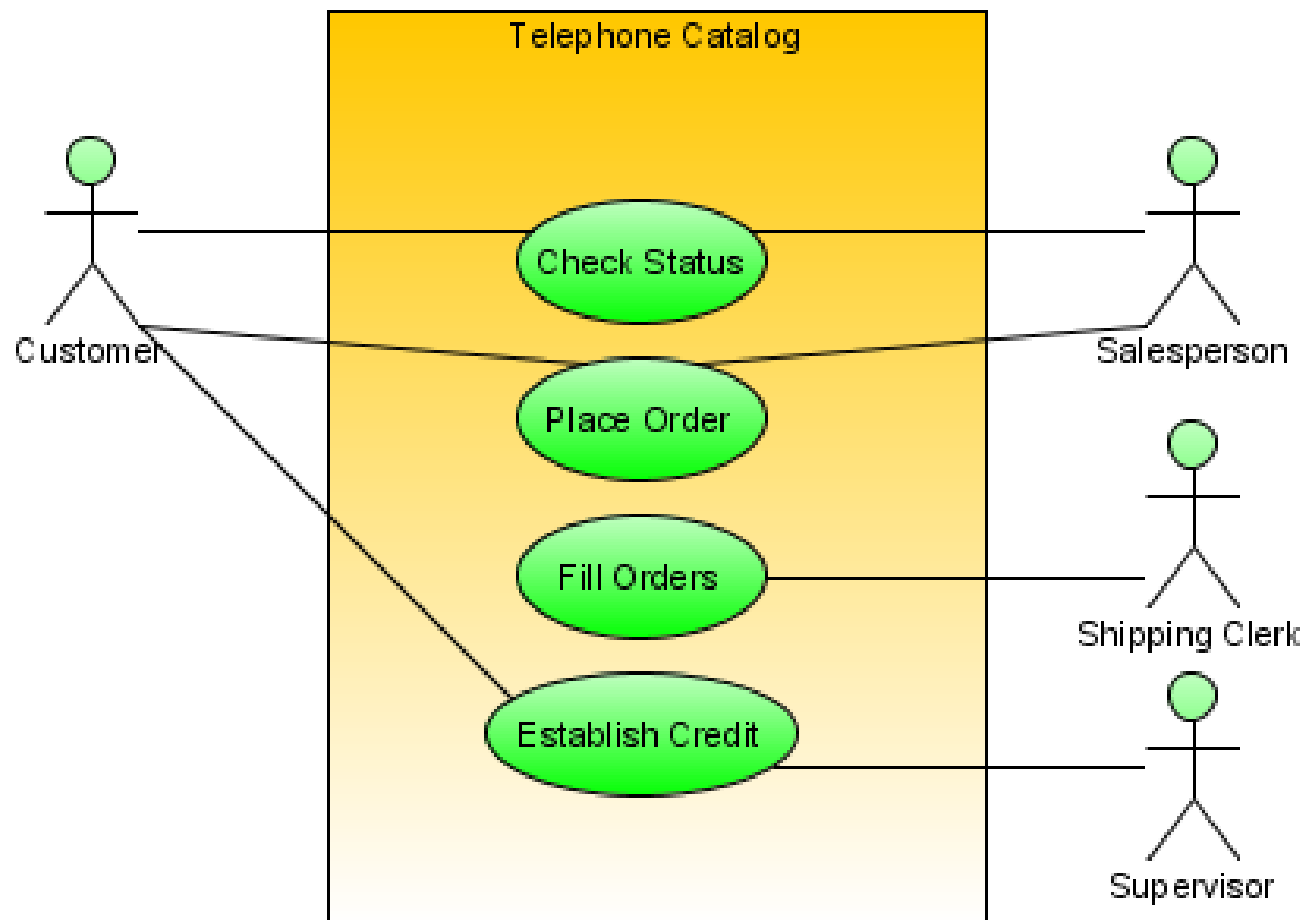
UML 2 Use cases diagrams describes the behavior of the target system from an external point of view. Use cases describe "the meat" of the actual requirements.

- **Use cases.** A use case describes a sequence of actions that provide something of measurable value to an actor and is drawn as a horizontal ellipse.
- **Actors.** An actor is a person, organization, or external system that plays a role in one or more interactions with your system. Actors are drawn as stick figures.
- **Associations.** Associations between actors and use cases are indicated by solid lines. An association exists whenever an actor is involved with an interaction described by a use case.

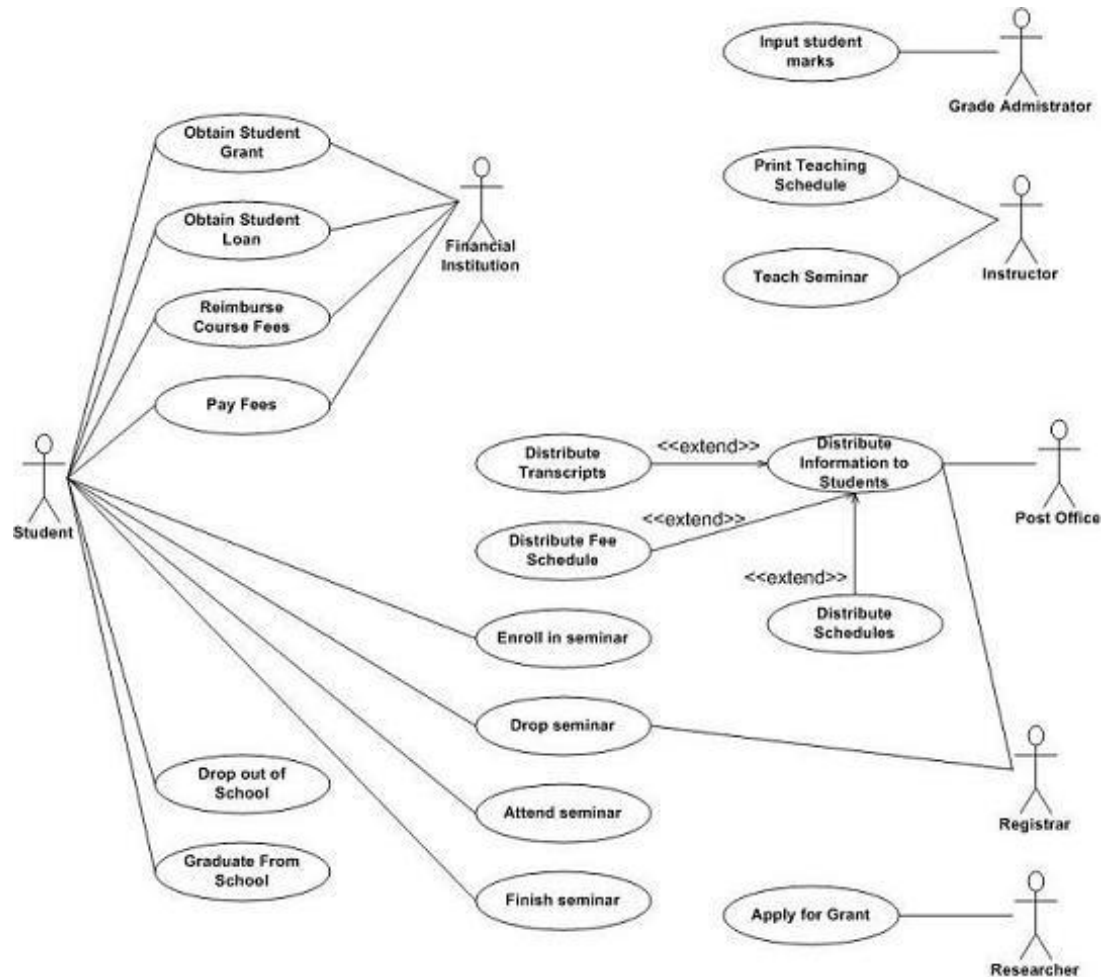
Use cases diagram



Use cases diagram



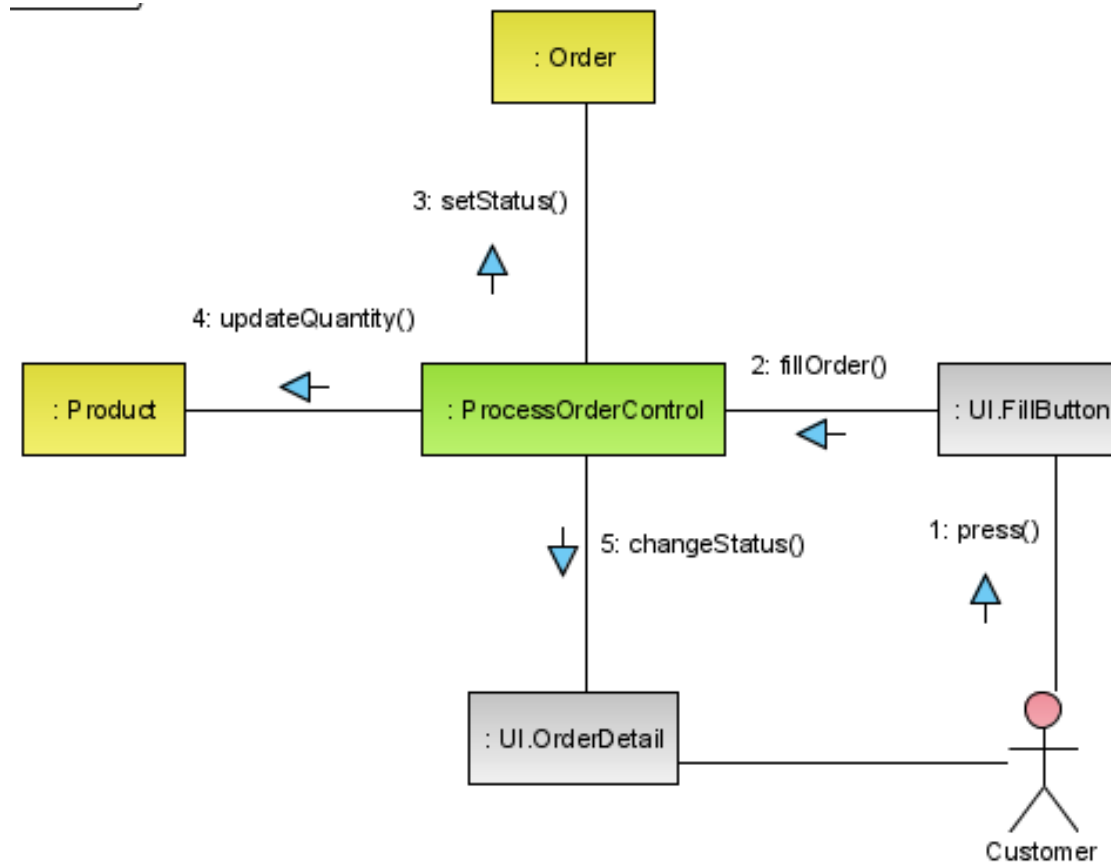
Use cases diagram



Communication diagram

- UML 2 Communication diagrams used to model the dynamic behavior of the use case. When compare to Sequence Diagram, the Communication Diagram is more focused on showing the collaboration of objects rather than the time sequence.

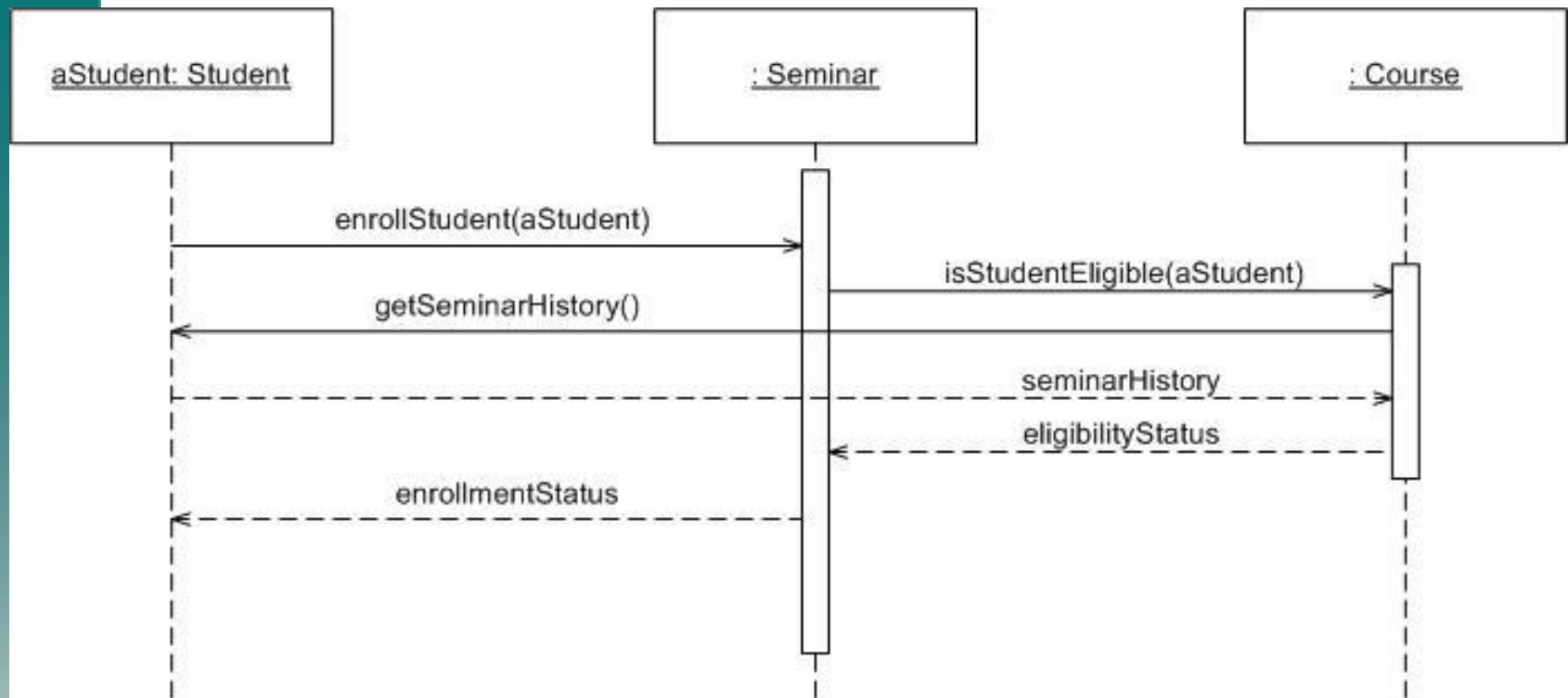
Communication diagram



Sequence diagram

- UML 2 Sequence diagrams models the collaboration of objects based on a time sequence. It shows how the objects interact with others in a particular scenario of a use case.

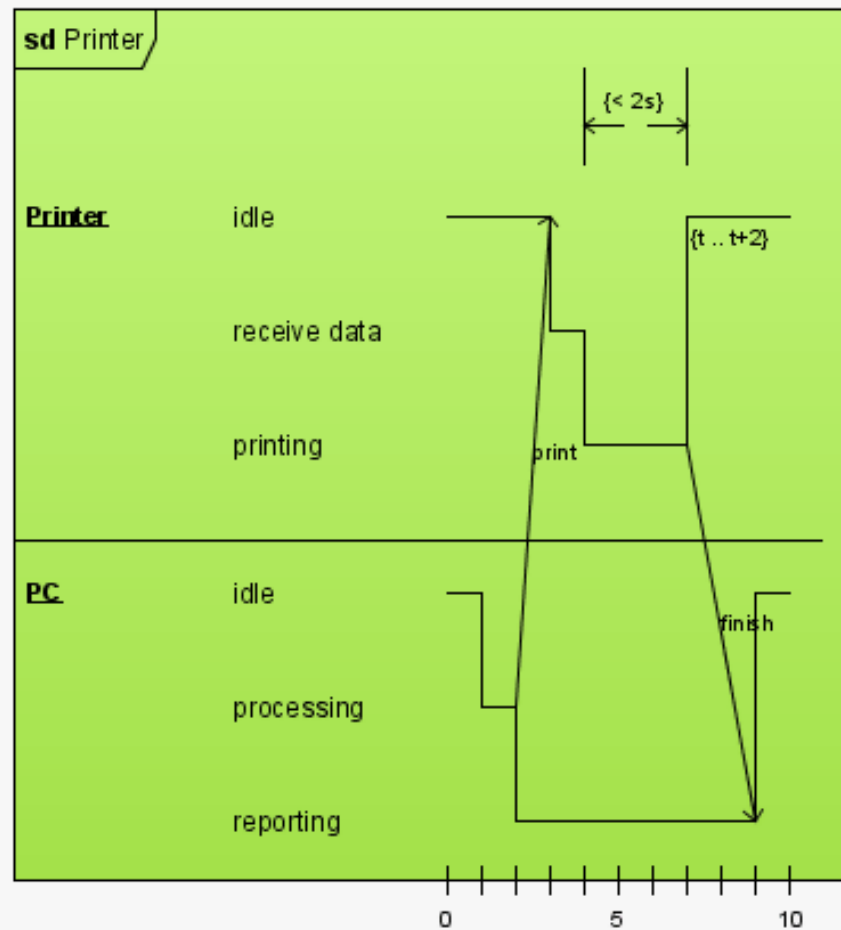
Sequence diagram



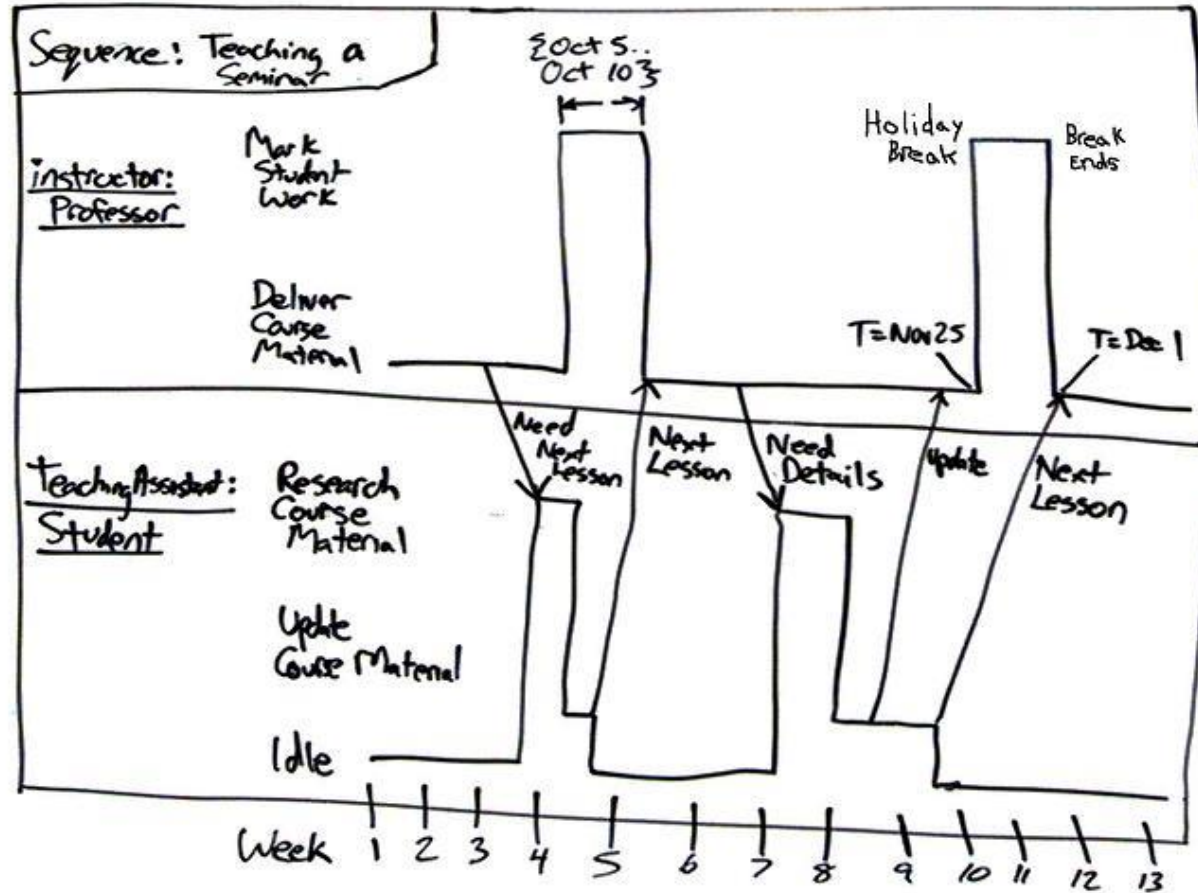
Timing diagram

- UML 2 Timing diagrams shows the behavior of the objects in a given period of time. Timing diagram is a special form of a sequence diagram. The differences between timing diagram and sequence diagram are the axes are reversed so that the time are increase from left to right and the lifelines are shown in separate compartments arranged vertically.

Timing diagram



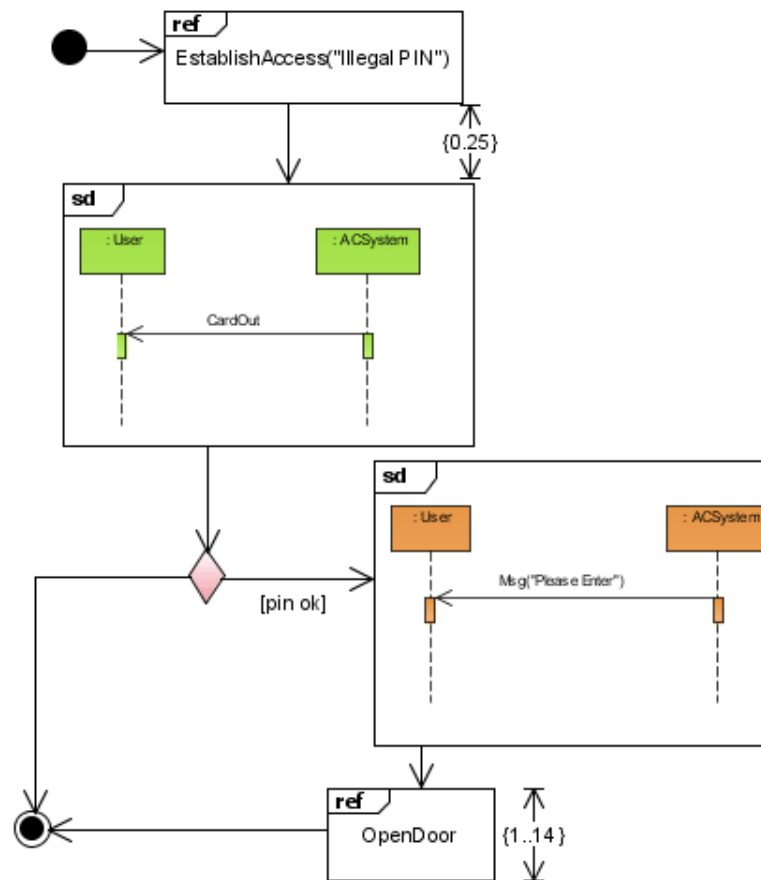
Timing diagram



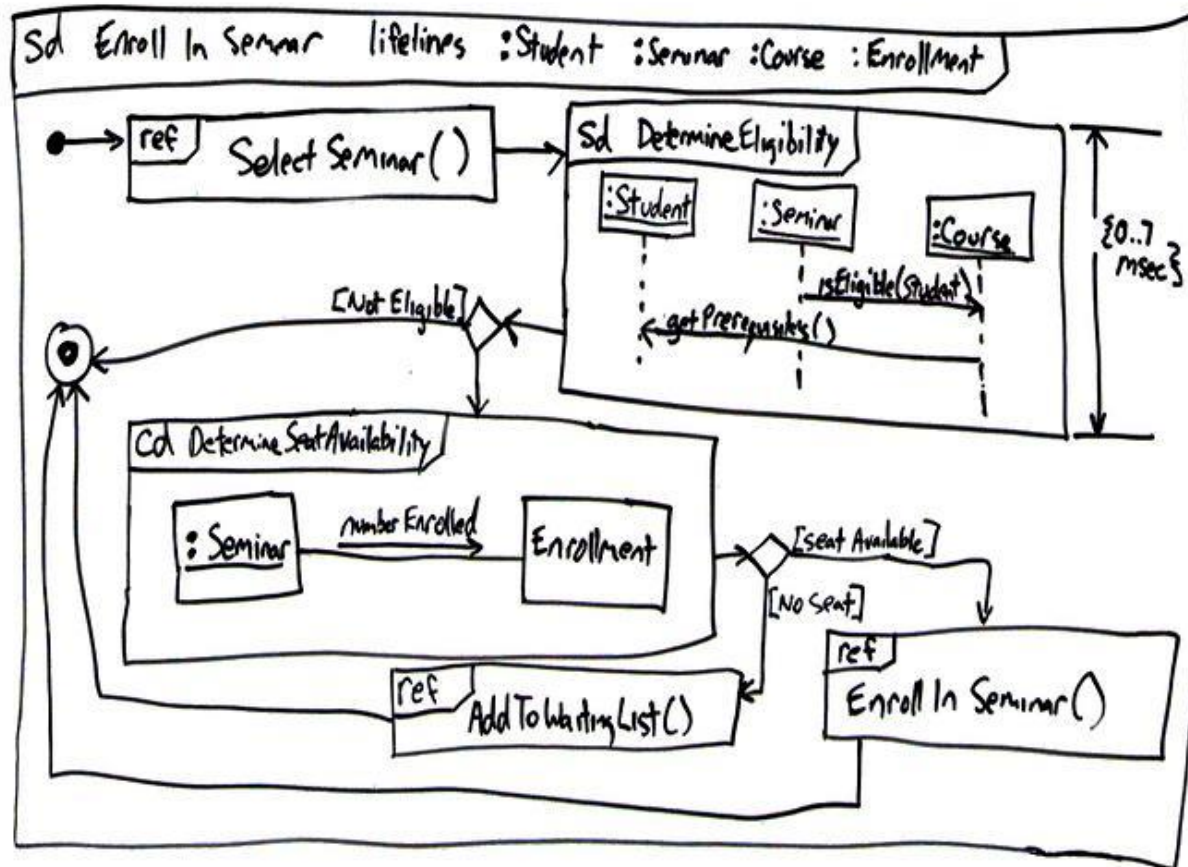
Interaction overview diagram

- UML 2 Interaction overview diagrams focuses on the overview of the flow of control of the interactions. It is a variant of the Activity Diagram where the nodes are the interactions or interaction occurrences. It describes the interactions where messages and lifelines are hidden.

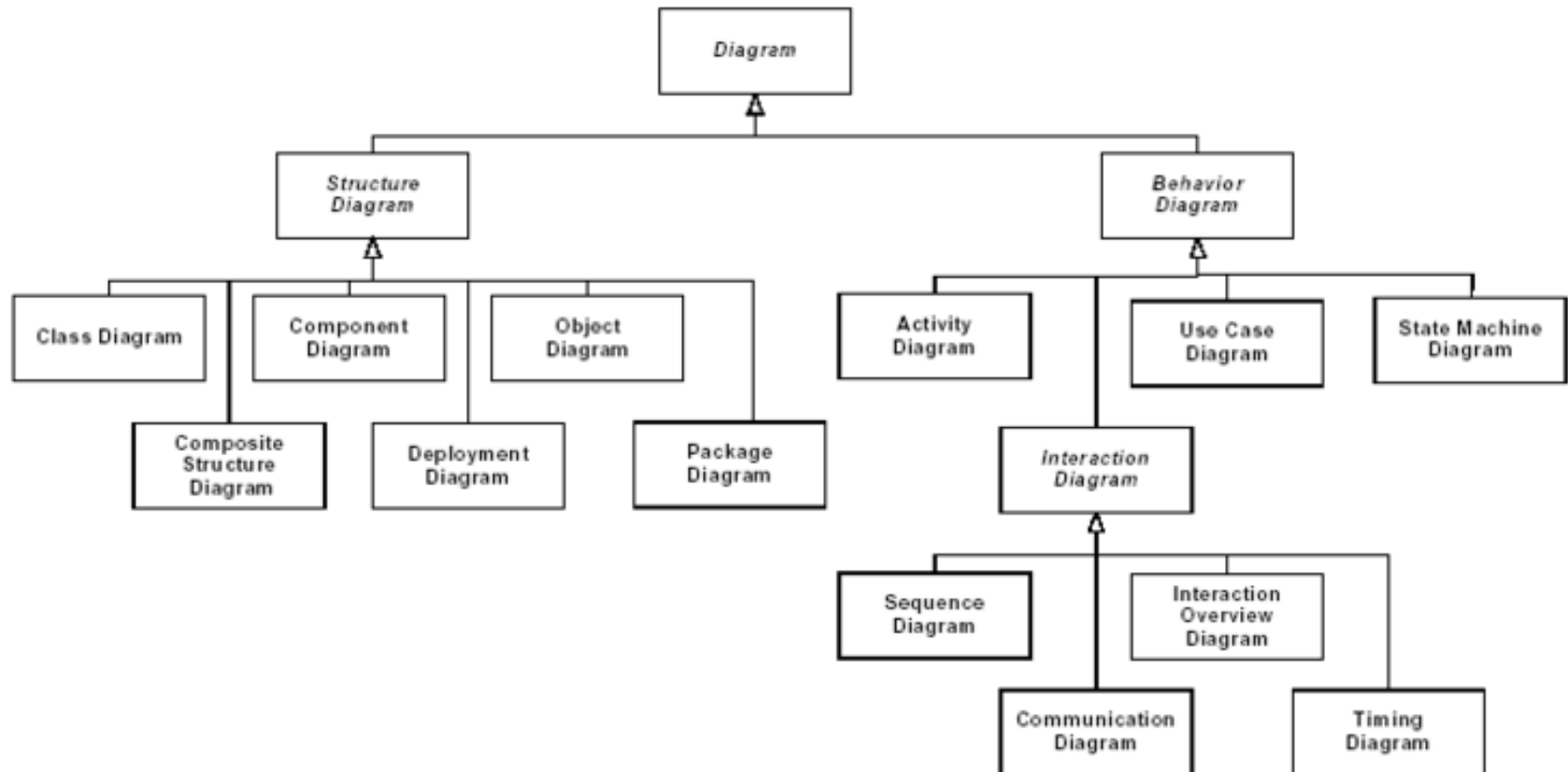
Interaction overview diagram



Interaction overview diagram



UML diagram hierarchy



References

- <http://www.agilemodeling.com/>
- <http://www.visual-paradigm.com/VPGallery/diagrams/index.html>
- <http://bdn.borland.com/article/0,1410,31863,00.html>
- http://en.wikipedia.org/wiki/Unified_Modeling_Language
- http://pigseye.kennesaw.edu/~dbraun/csis4650/A&D/UML_tutorial/index.htm

Design Fundamentals

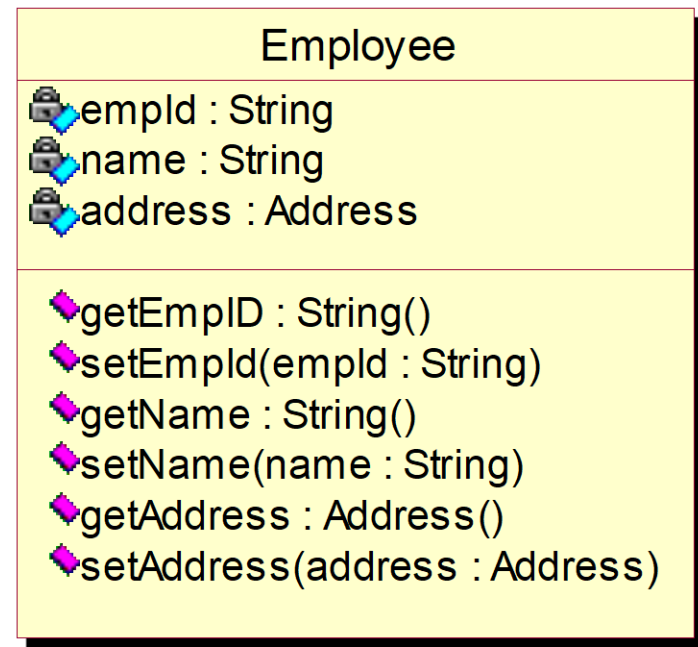
- Topics
 - A quick introduction to UML class diagrams
 - Relationships
 - Generalization
 - Realization
 - Association
 - Aggregation
 - Composition
 - Dependency

Class Diagram

- Every class has three compartments
 - Name of the class
 - Structure (Data members)
 - Behavior (Methods)

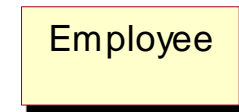
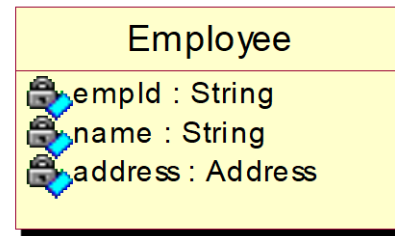
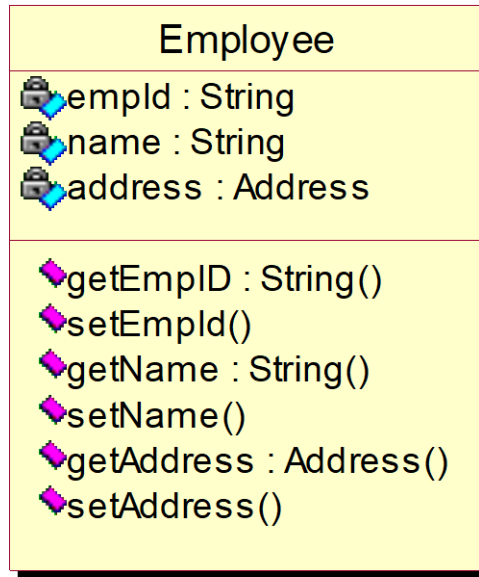
Structure →

Behavior →

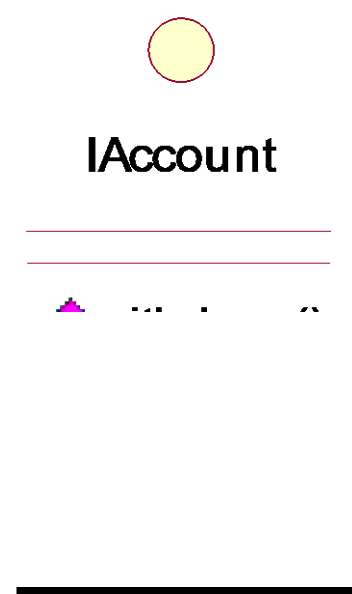
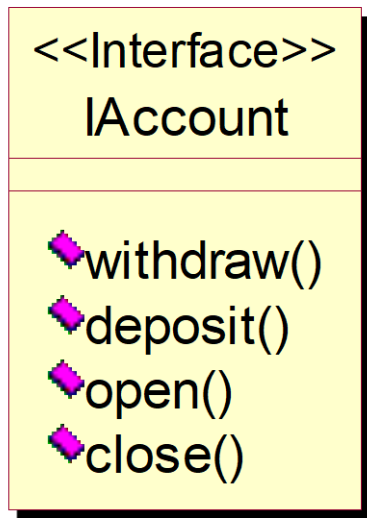


Class Diagram

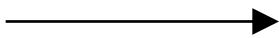
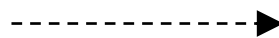

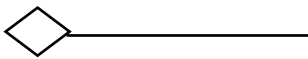
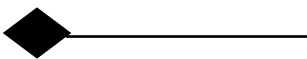
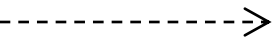
- Other representations



Interface



Relationships

- Six types of relationships in UML
 - Generalization 
 - Realization 
 - Association 
 - Aggregation 
 - Composite Aggregation or Composition 
 - Dependency 

Relationships

- Classification

<<Is-a>>

Generalization
Realization

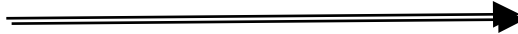
<<Has-a>>

Association
Aggregation
Composition

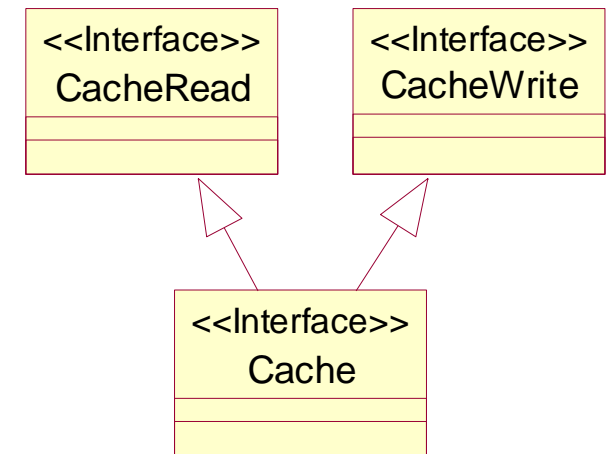
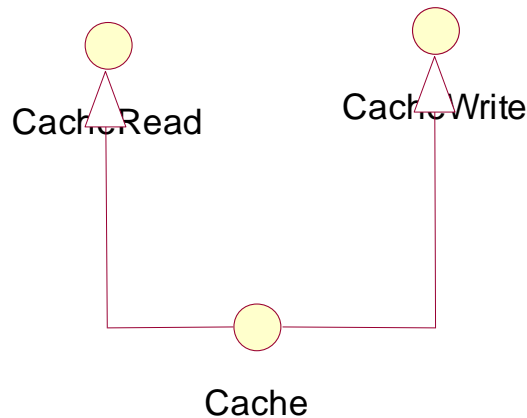
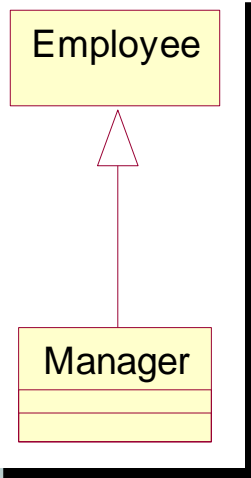
<<Uses>>

Dependency

Generalization

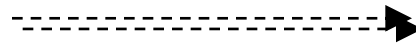
Generalization 

- Relationship between two classes or two interfaces
 - Avoid multiple inheritance between classes
 - Possible to have multiple inheritance between interfaces
 - Interface extends another interface



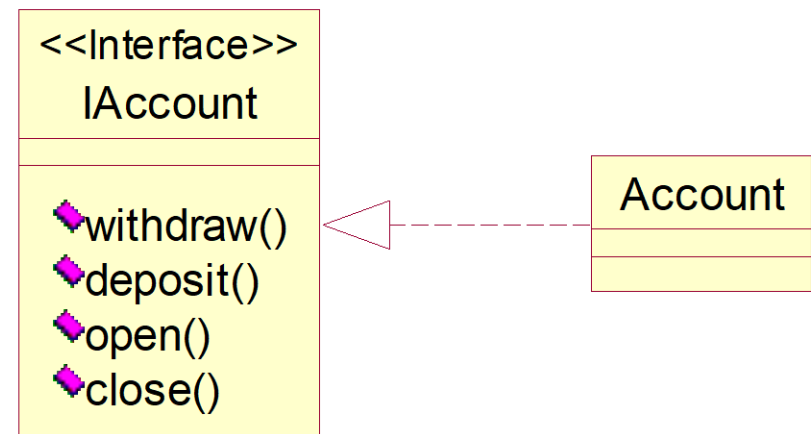
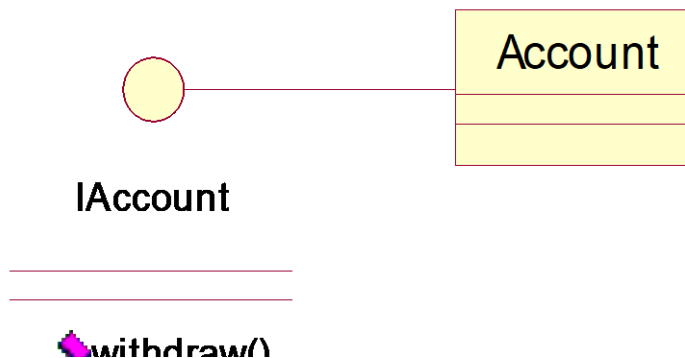
Realization

Realization



Relationship between a class and an interface

- A class can realize multiple interfaces
- Realizing an interface would require a class to provide an implementation for all the inherited method declarations



Inheritance

- Types of Inheritance
 - Interface Inheritance
 - Realization
 - Implementation Inheritance
 - Generalization

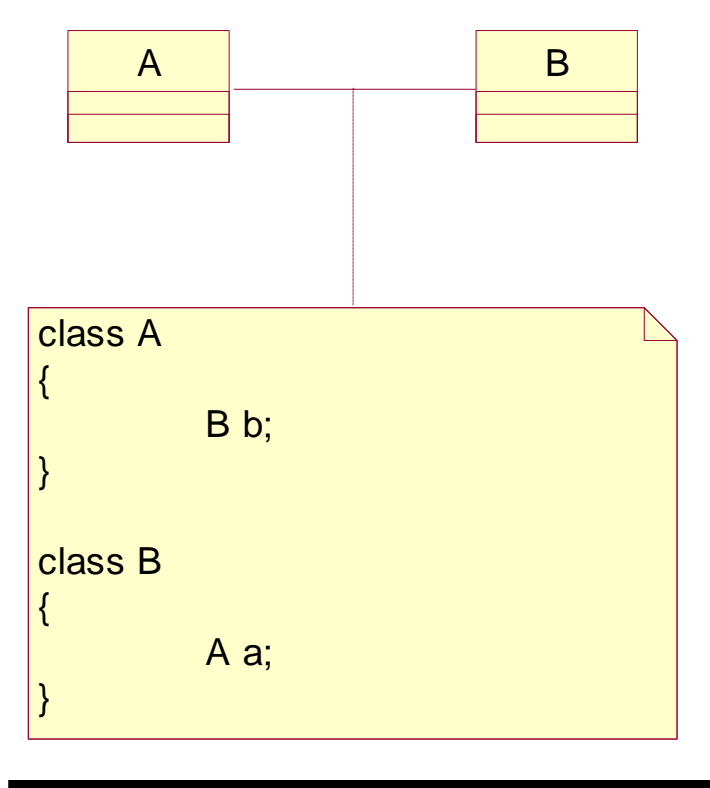
Association

Association _____

- ✦ 'Has-a' relationship
- ✦ Semantic relationship between two or more classifiers that involve connections among their instances
- ✦ The associations are qualified by
 - ✦ Role name
 - ✦ Navigability
 - ✦ Multiplicity

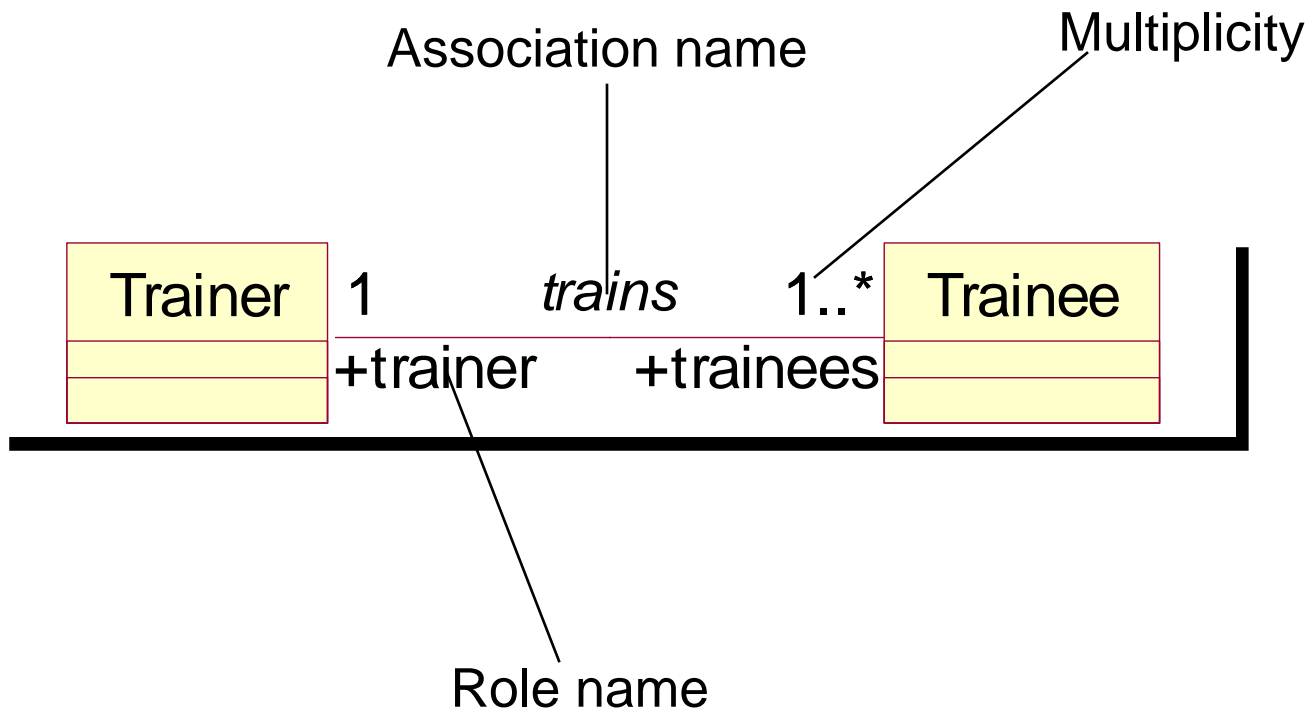
Association

- Multiplicity is by default 1
- Navigability is by default bi-directional



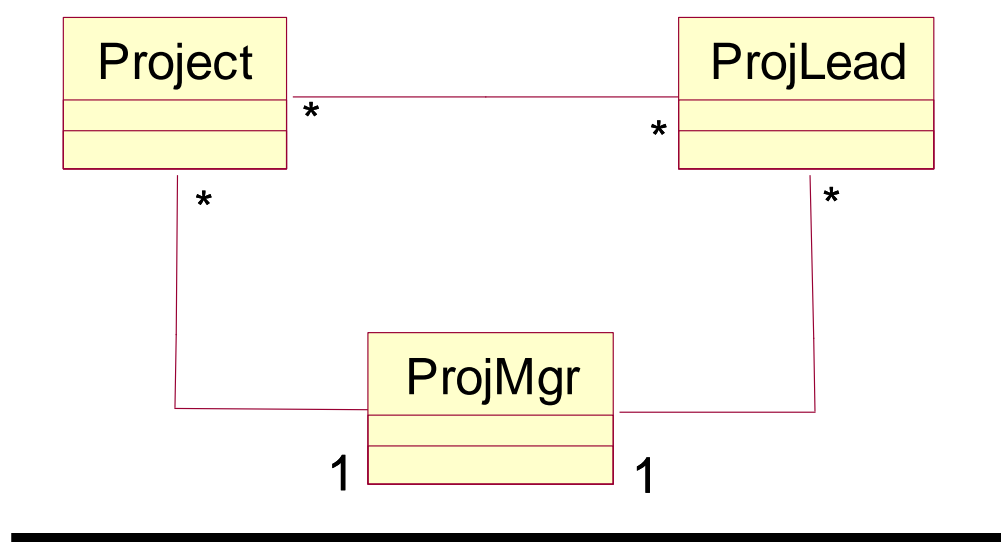
Association

✦ 'Has-a' relationship



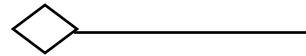
Association

✦ Examples



Aggregation

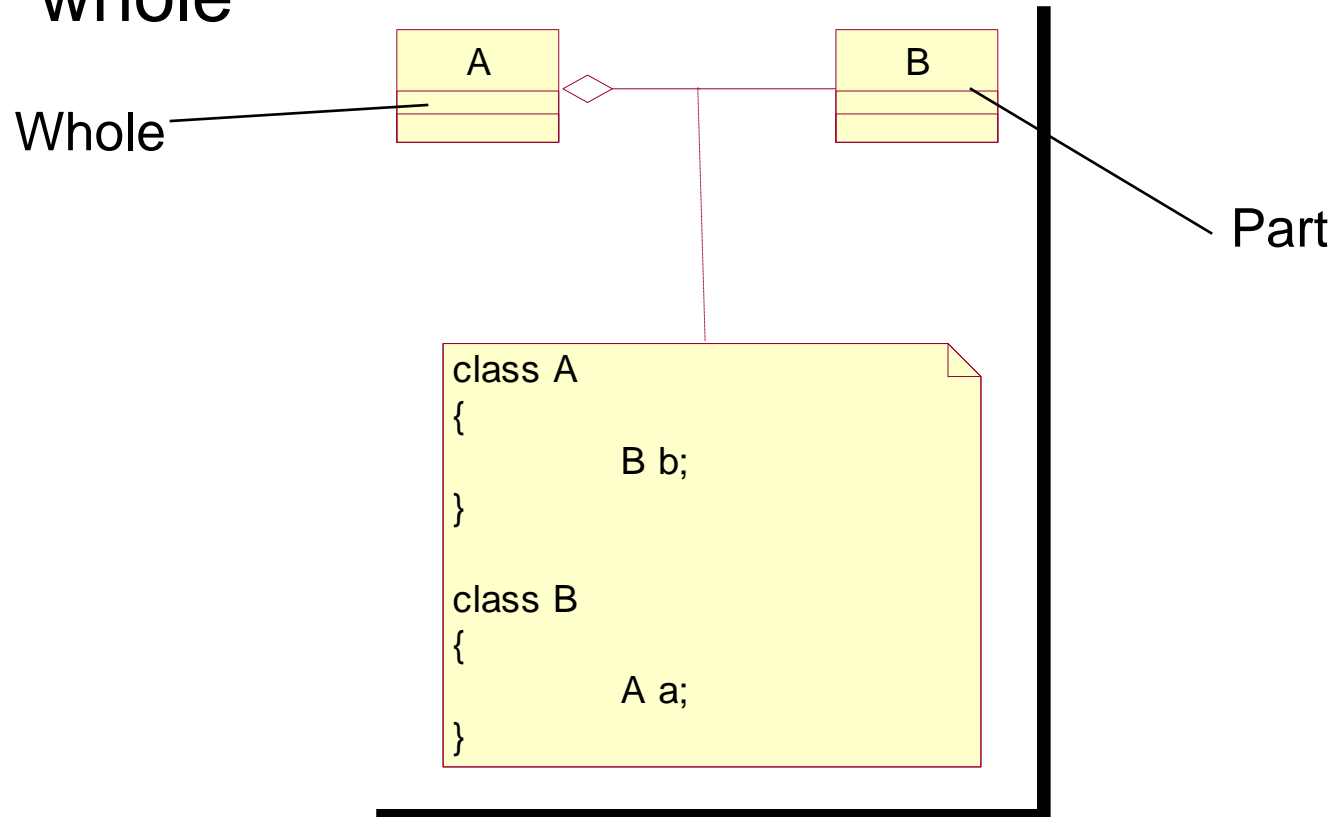
Aggregation



- ✦ 'Has-a' relationship
- ✦ Is a special (stronger) form of association which conveys a whole part meaning to the relationship
 - ✦ Also known as Aggregate Association
- ✦ Has multiplicity and navigability
 - Multiplicity is by default 1
 - Navigability is by default bi-directional

Aggregation

- ✦ The hollow diamond is placed towards the whole

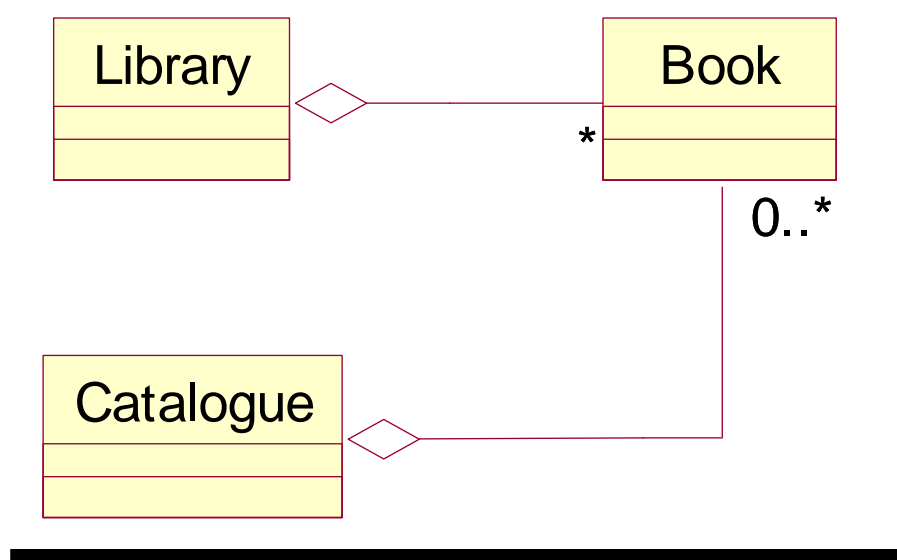


Aggregation

- ✦ So what's the difference between Association and Aggregation?
 - ✦ When it comes to code – NOTHING!
- ✦ Based on the context
 - ✦ Whole Part
 - ✦ Lack of independent use and existence
 - ✦ Scope of verb is constrained to only 'has' or 'contains'
- ✦ When in doubt, leave aggregation out!

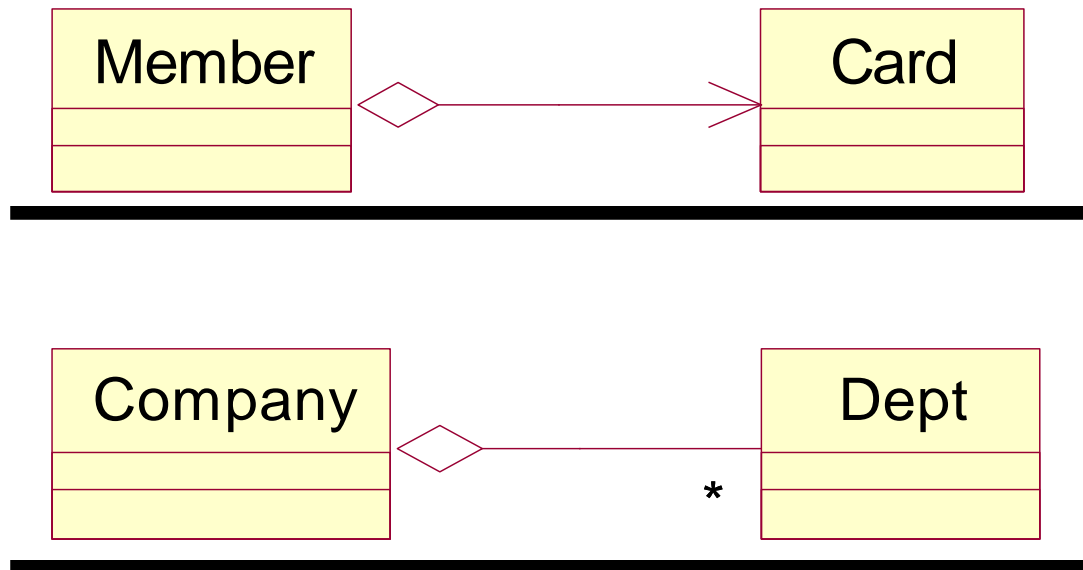
Aggregation

✦ The Whole – Part nature



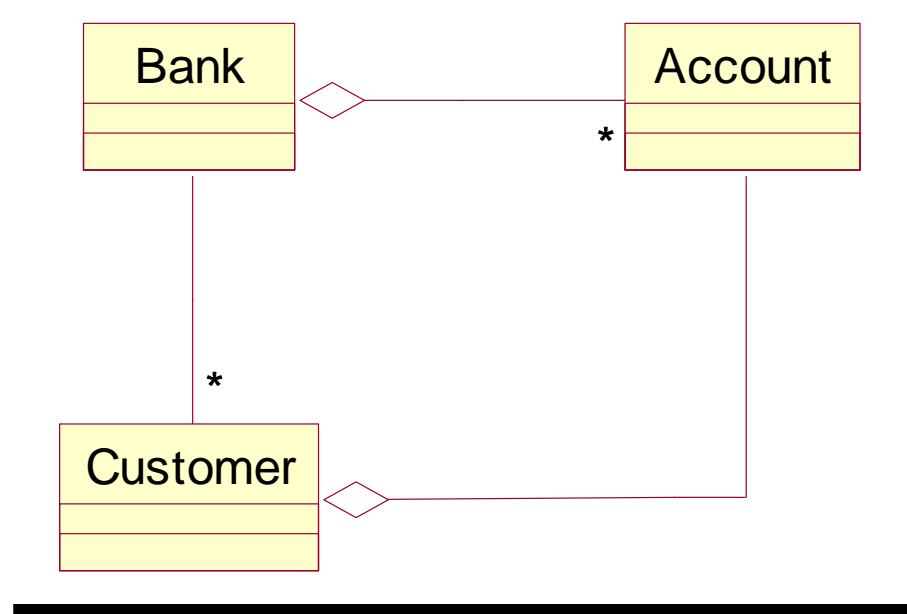
Aggregation

✦ Independent existence / use



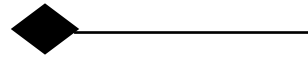
Aggregation

- ✦ A part can be shared between many wholes
 - ✦ Shared aggregation



Composite Aggregation

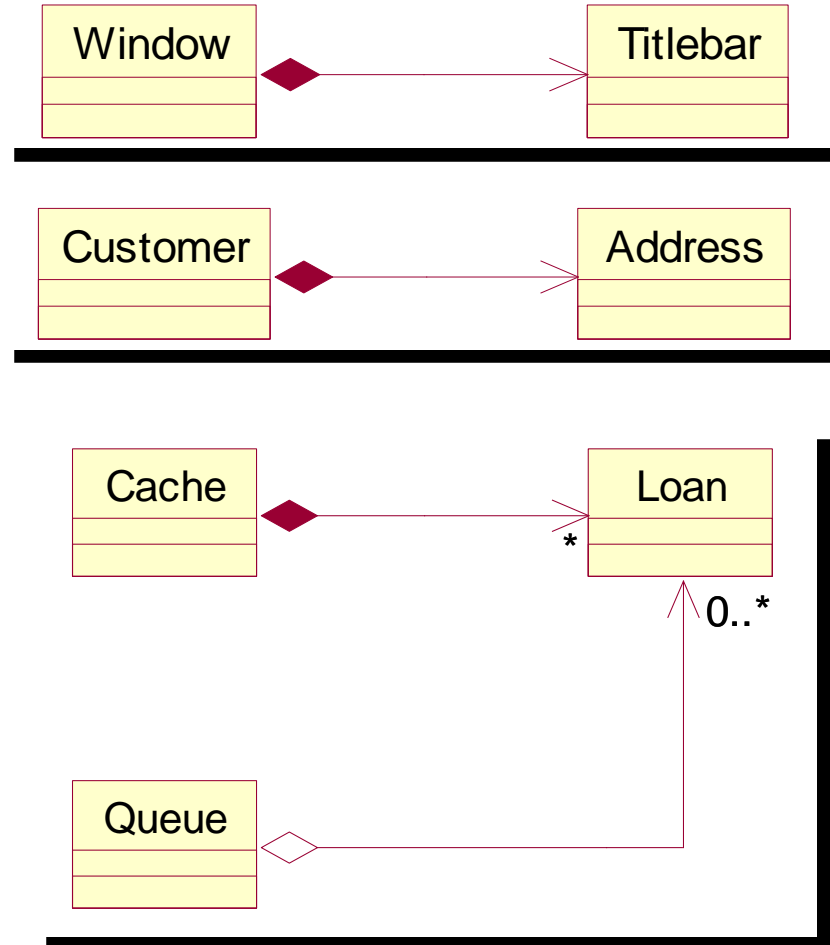
Composite Aggregation



- ✦ 'Has-a' relationship
- ✦ Is a stronger form of aggregation
 - ✦ Also known as Composition
- ✦ Explicit lifetime control
 - ✦ Part is created when whole is created
 - ✦ Part is destroyed when whole is destroyed
 - ✦ Exclusive ownership
- ✦ Has multiplicity and navigability
 - Multiplicity at the whole end should always be 1
 - Navigability is by default bi-directional

Composite Aggregation

Examples



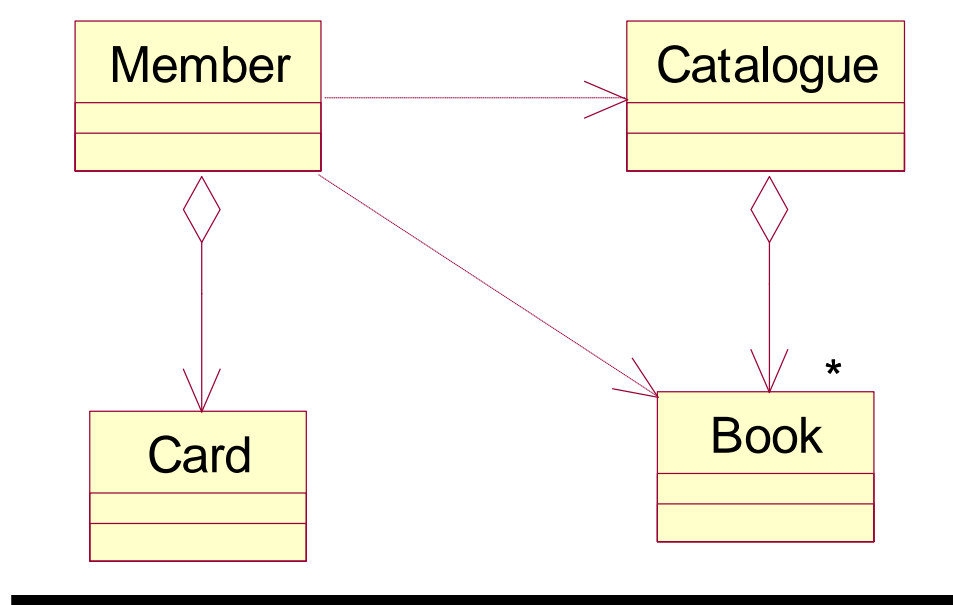
Dependency

Dependency ----->

- ✦ 'Uses' relationship
- ✦ Behavioral relationship
 - ✦ Loose coupling
 - ✦ Has no impact on class structure (data members)
- ✦ A class references another class only within its methods for the purpose of:
 - ✦ Invoking a static method
 - ✦ Local instantiation
 - ✦ Formal argument use
 - ✦ Return type

Dependency

✦ 'Uses' relationship



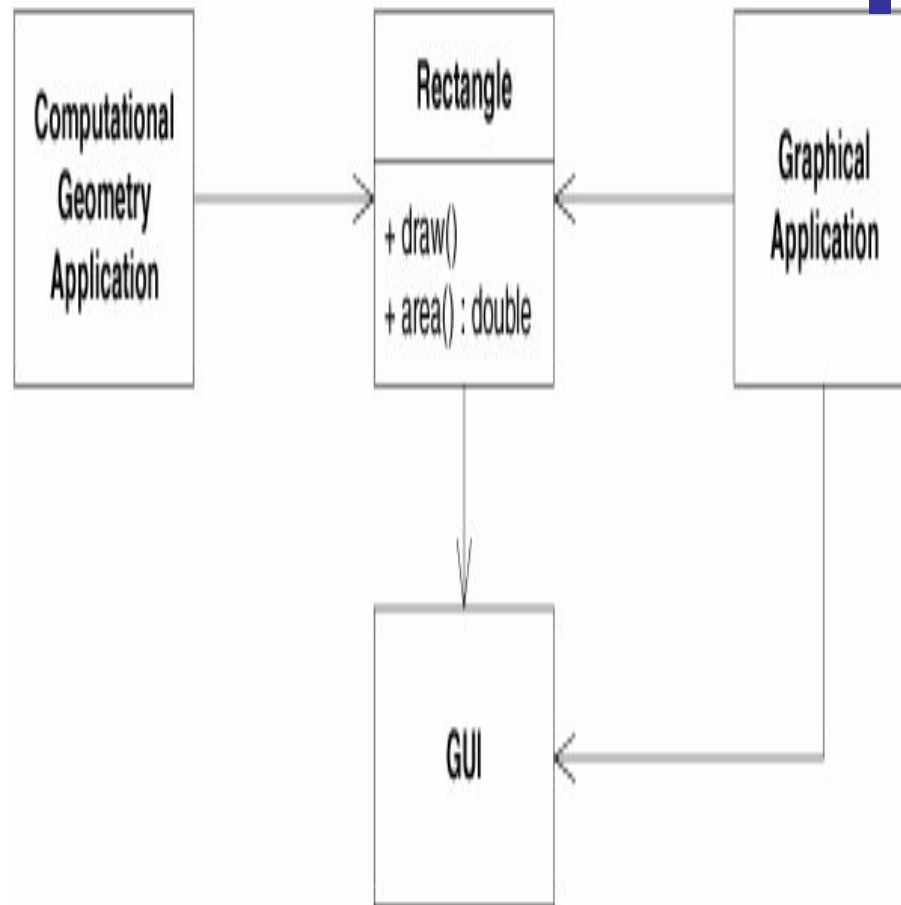
SOLID Design Principles

Single Responsibility Principle (SRP)

The Single-Responsibility Principle

- Every class should have a single responsibility, and that responsibility should be entirely encapsulated by the class.
 - All its services should be narrowly aligned with that responsibility
- *“A class should have only one reason to change”*
 - The reason is that each responsibility is an axis of change
 - If a class has more than one responsibility, the responsibilities become coupled. Changes to one responsibility may impair or inhibit the class's ability to meet the others

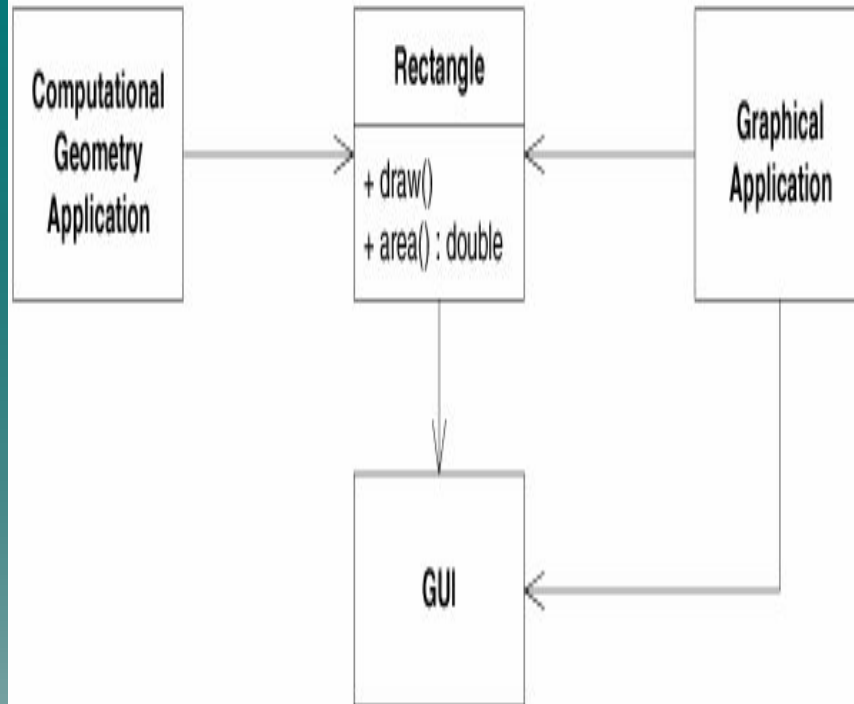
SRP - Example



■ This design violates SRP. The Rectangle class has two responsibilities.

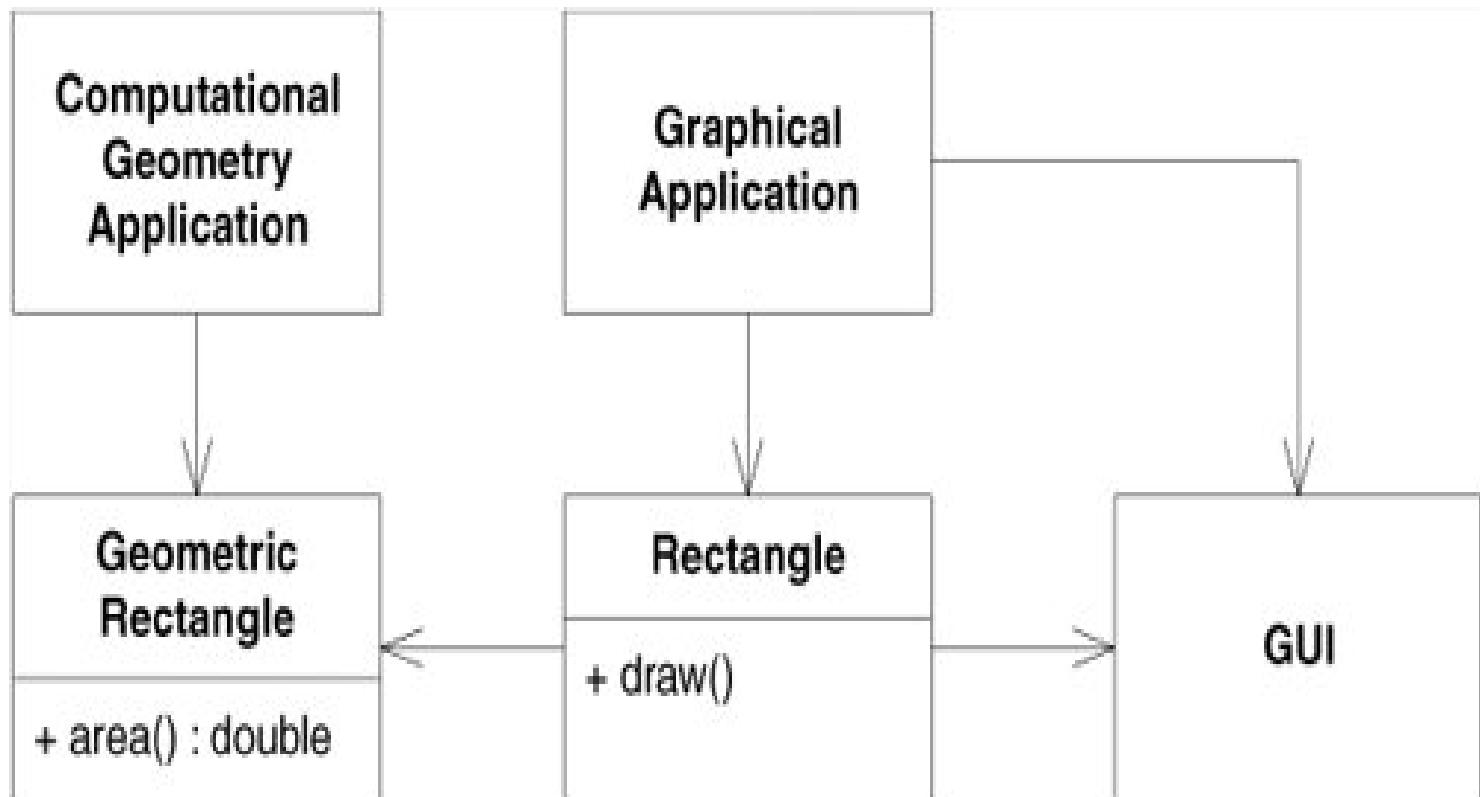
- The first responsibility is to provide a mathematical model of the geometry of a rectangle.
- The second responsibility is to render the rectangle on a GUI.

SRP - Example



- The violation of SRP causes several nasty problems.
- First, we must include GUI in the computational geometry application
- Second, if a change to the GraphicalApplication causes the Rectangle to change for some reason, that change may force us to rebuild, retest, and redeploy the ComputationalGeometryApplication.
- If we forget to do this, that application may break in unpredictable ways

Separated responsibilities



Defining a Responsibility

- In the context of the SRP, we define a responsibility to be a reason for change.
- If you can think of more than one motive for changing a class, that class has more than one responsibility.
- This is sometimes difficult to see.
 - We are accustomed to thinking of responsibility in groups

Example - 2

```
public interface Modem
{
    public void Dial(string pno);
    public void Hangup();
    public void Send(char c);
    public char Recv();
}
```

```
-----
public class ModemImpl
{
    public void Dial(string
        pno){}
    public void Hangup(){ }
    public void Send(char c){}
    public char Recv(){ }
}
```

- The four functions it declares are certainly functions belonging to a modem
- However, two responsibilities are being shown here.
- The first responsibility is connection management.
- The second is data communication.
- The dial and hangup functions manage the connection of the modem; the send and recv functions communicate data

Example - 2

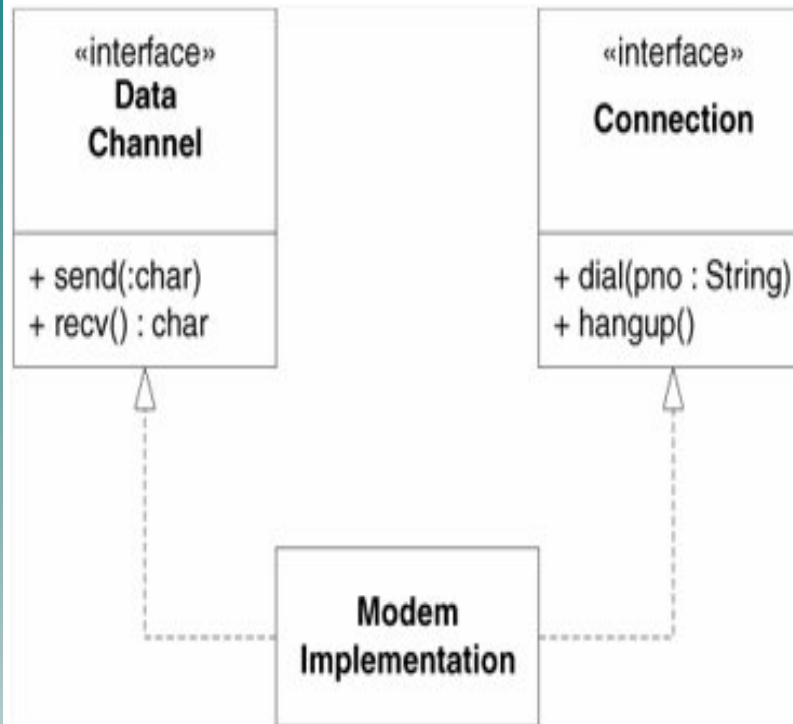
- Should these two responsibilities be separated?
- That depends on how the application is changing.
- **If** the application changes in ways that affect the signature of the connection functions, the classes that call send and read will have to be recompiled and redeployed more often than we like.
- In that case, the two responsibilities should be separated
- **If**, on the other hand, the application is not changing in ways that cause the two responsibilities to change at different times, there is no need to separate them.

Example - 2

An axis of change is
an axis of change
only if the changes
occur

- It is not wise to apply SRP or any other principle, for that matter if there is no symptom

Example - 2



- Note that, I kept both responsibilities coupled in the **ModemImplementation** class.
- This is not desirable, but it may be necessary.
- There are often reasons, having to do with the details of the hardware or operating system, that force us to couple things that we'd rather not couple.
- However, by separating their interfaces, we have decoupled the concepts as far as the rest of the application is concerned.
- however, note that all dependencies flow away from it. Nobody needs to depend on this class

Open/Closed Principle (OCP)

OCP - motivations

- How many times do you start writing a brand new application from nothing versus the number of times you start by adding new functionality to an existing codebase?
- Chances are good that you spend far more time adding new features to an existing codebase.

OCP - motivations

- Is it easier to write all new code or to make changes to existing code?
- It's usually far easier for me to write all new methods and classes than it is to break into old code and find the sections I need to change.
 - Modifying old code adds the risk of breaking existing functionality.
 - With new code you generally only have to test the new functionality.
 - When you modify old code you have to both test your changes and then perform a set of regression tests to make sure you didn't break any of the existing code.

The Open/Closed Principle (OCP)

Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification.

Description of OCP

- Modules that conform to OCP have two primary attributes.
 - They are open for extension. This means that the behavior of the module can be extended. As the requirements of the application change, we can extend the module with new behaviors that satisfy those changes. In other words, we are able to change what the module does.
 - They are closed for modification. Extending the behavior of a module does not result in changes to the source, or binary, code of the module.

Liskov Substitution Principle (LSP)

Liskov Substitution Principle

- The primary mechanisms behind the Open/Closed Principle are abstraction and polymorphism
- key mechanisms that supports abstraction and polymorphism is inheritance
- What are the design rules that govern this particular use of inheritance?
- What are the characteristics of the best inheritance hierarchies?
- What are the traps that will cause us to create hierarchies that do not conform to OCP? These are the questions addressed by the Liskov Substitution Principle (LSP).

The Liskov Substitution Principle

- ***Subtypes must be substitutable for their base types.***
- Barbara Liskov wrote this principle in 1988. She said:
 - What is wanted here is something like the following substitution property: If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T , the behavior of P is unchanged when o_1 is substituted for o_2 then S is a subtype of T .

The Interface Segregation Principle (ISP)

The Interface Segregation Principle (ISP)

- This principle deals with the disadvantages of "fat" interfaces.
- Classes whose interfaces are not cohesive have "fat" interfaces.
 - In other words, the interfaces of the class can be broken up into groups of methods. Each group serves a different set of clients.
 - ISP acknowledges that there are objects that require noncohesive interfaces; however, it suggests that clients should not know about them as a single class.

The Interface Segregation Principle

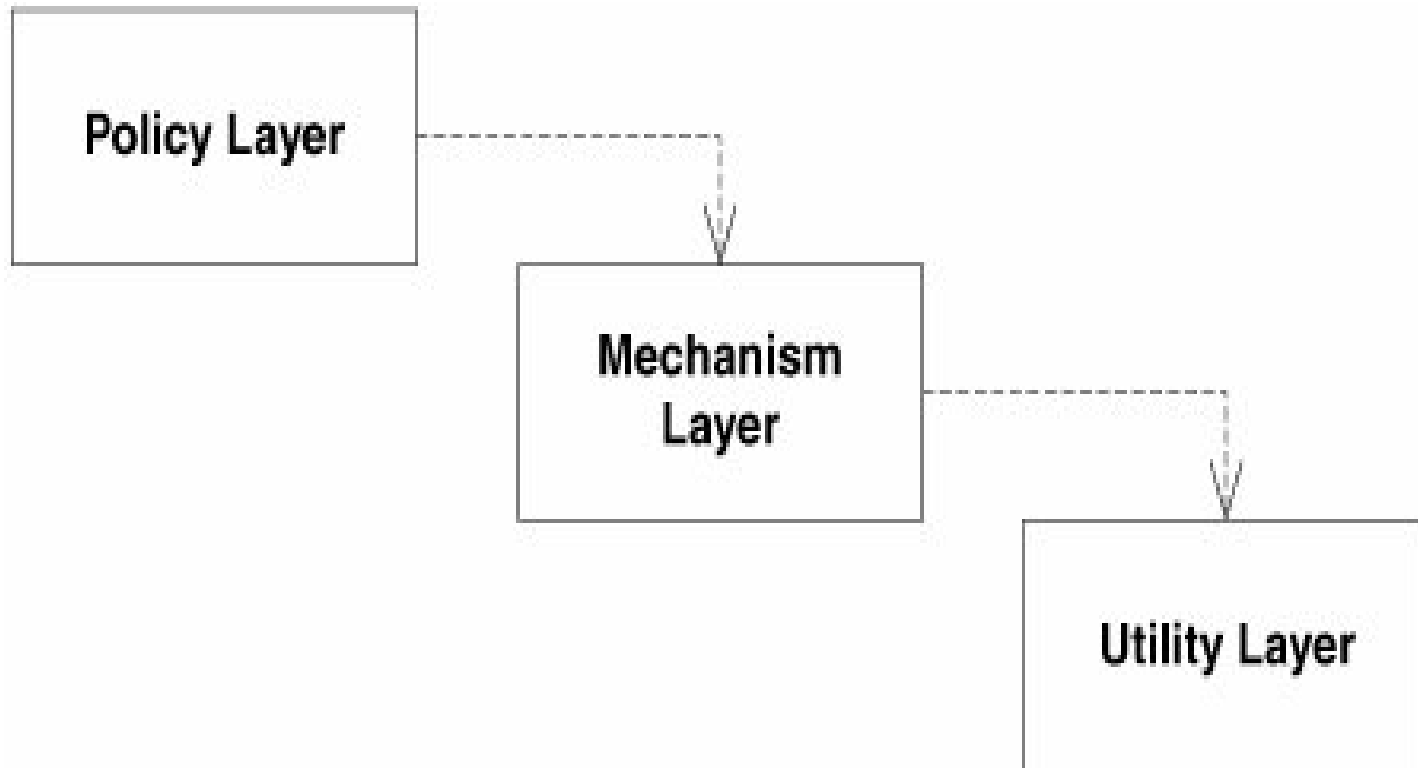
- ***Clients should not be forced to depend on methods they do not use.***
- When clients are forced to depend on methods they don't use, those clients are subject to changes to those methods.
 - This results in an inadvertent coupling between all the clients.
 - Said another way, when a client depends on a class that contains methods that the client does not use but that other clients do use, that client will be affected by the changes that those other clients force on the class.
- We would like to avoid such couplings where possible, and so we want to separate the interfaces.

The Dependency- Inversion Principle (DIP)

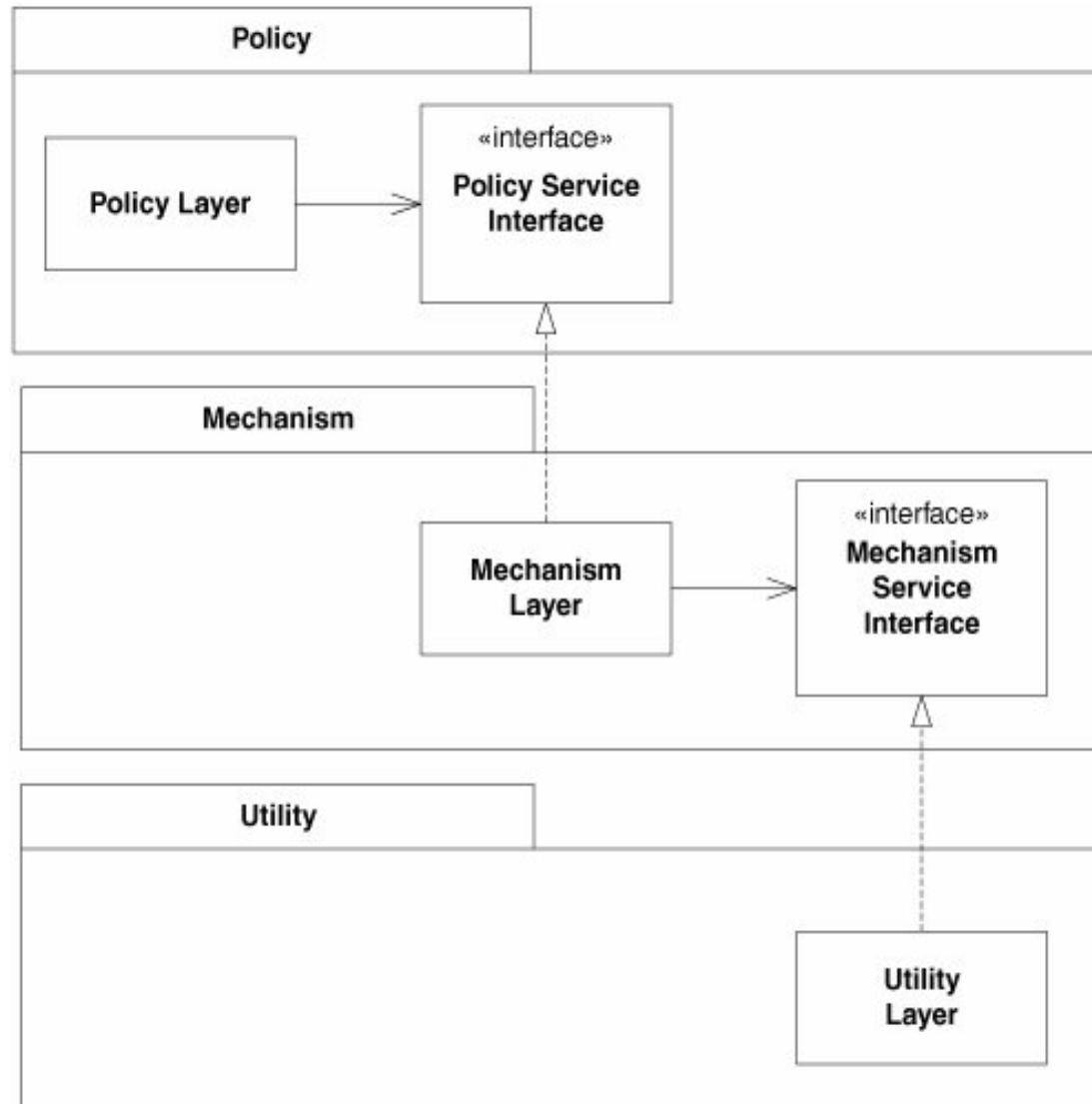
The Dependency-Inversion Principle

- **The Dependency-Inversion Principle**
 - *High-level modules should not depend on low-level modules. Both should depend on abstractions.*
 - *Abstractions should not depend upon details. Details should depend upon abstractions.*

In conventional architecture, higher-level components depend upon lower-level components as depicted in the following diagram:



Inverted layers



Ownership Inversion

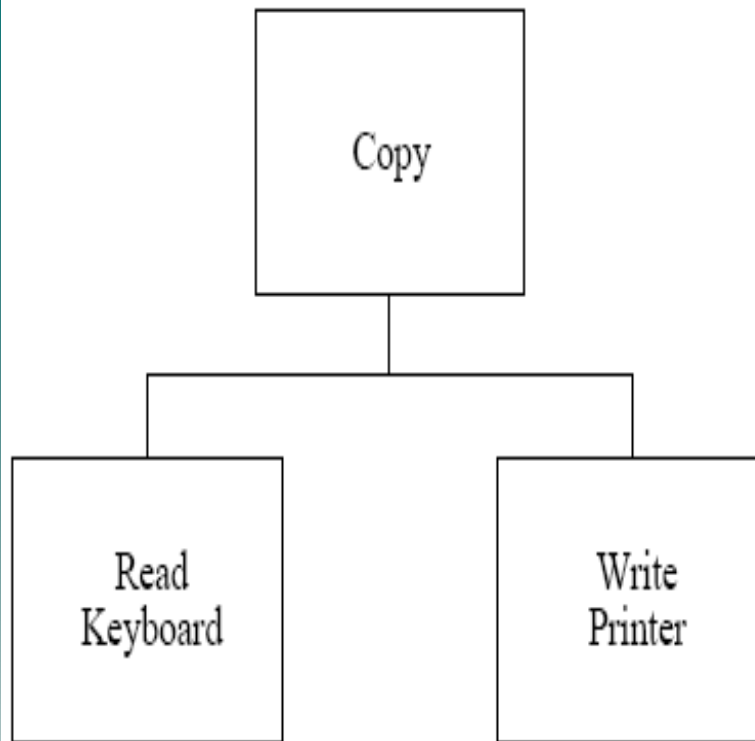
- Note that the inversion here is one of not only dependencies but also interface ownership. We often think of utility libraries as owning their own interfaces.
- But when DIP is applied, we find that the clients tend to own the abstract interfaces and that their servers derive from them.
- This is sometimes known as the Hollywood principle: "Don't call us; we'll call you."
- The lower-level modules provide the implementation for interfaces that are declared within, and called by, the upper-level modules.

Dependence on Abstractions

- A somewhat more naive, yet still very powerful, interpretation of DIP is the simple heuristic: "Depend on abstractions." Simply stated, this heuristic recommends that you should not depend on a concrete class and that rather, all relationships in a program should terminate on an abstract class or an interface.
 - No variable should hold a reference to a concrete class.
 - No class should derive from a concrete class.
 - No method should override an implemented method of any of its base classes.
- Certainly, this heuristic is usually violated at least once in every program. Somebody has to create the instances of the concrete classes, and whatever module does that will depend on them

An Example

Before



After

