OOAD using UML

Sathyanarayana Adiga

OOAD

- Proverb
  - “Owning a hammer doesn't make one an architect”

- OOAD
  - Art of finding objects
  - Assigning responsibilities to objects
  - Make them collaborate

The Software crisis

- Increasing cost, failure of Software systems
- Increasingly complex systems
- Legacy code
- Shorter development cycles
- Poorly defined requirements
- Poor risk management
- Insufficient testing

Risks in a Software project

- Functional risks
- Resource risks
- Architectural risks

Why OOP has not solved this crisis?

- Benefits of OOP
  - Real world Models
  - Easy to understand systems
  - Reusability through Inheritance
  - Stability through Encapsulation

- Just OOP is not enough
  - Developers start implementing too soon
  - Don't understand requirements
  - Lack of analysis
  - Lack of modeling

Introducing OOAD

- Facilitates Analysis and design before Implementation
- Requirements captured and thoroughly documented
- Detailed modeling based on requirements
- Analysts, designers, developers study other systems
- Abstraction for whole system
- Different solutions to a problem analyzed

Two phases of OOAD

- OOA
  - Models based on user requirements
  - Black-box approach
  - What the system will do and not HOW
  - Technologies not discussed
  - Analysis model – problem domain concepts

Two phases of OOAD

- OOD
  - Add details and design decisions to a model
  - White-box approach
  - Developer's perspective
  - New classes will be added
    - EG – Persistence, inter-process communication etc.
  - Complete solution to the problems discussed in analysis

Defining Successful software Systems

- User should be able to use it effectively
- Easily maintainable
- Scalable
- Portable between platforms
- Code should be reusable
- Delivered in time within budget
- Changes to one module should not affect others

Introducing Modeling

- Models usually represented through notations
- Notations include graphical symbols, connectors
- A notation portrays complex systems
- A good notation
  - Allows accurate description of systems
  - As simple as possible
  - Flexible to communicate new ideas

Introducing UML

- UML is robust
- Diagrams describe different views of the system
- Need proper process to use UML effectively
- Combination of process and notations
- Unified Modeling Language
  - OMT (James Rumbaugh)
  - Booch method (Grady Booch)
  - OOSC (Ivar Jacobson)
- 1997 UML 1.0 became OMG Standard

Aims of UML

- Standardize notations used in analysis and design
- Model system using OO concepts
- Accurately describe the artifacts
- Scalable
- Notations used by both humans and tools

Three components of UML

- Model elements
  - Classes, Objects, relationships
- Diagrams
  - Combination of model elements
- Views
  - Use-case view, Component view, Deployment view

Why use UML?

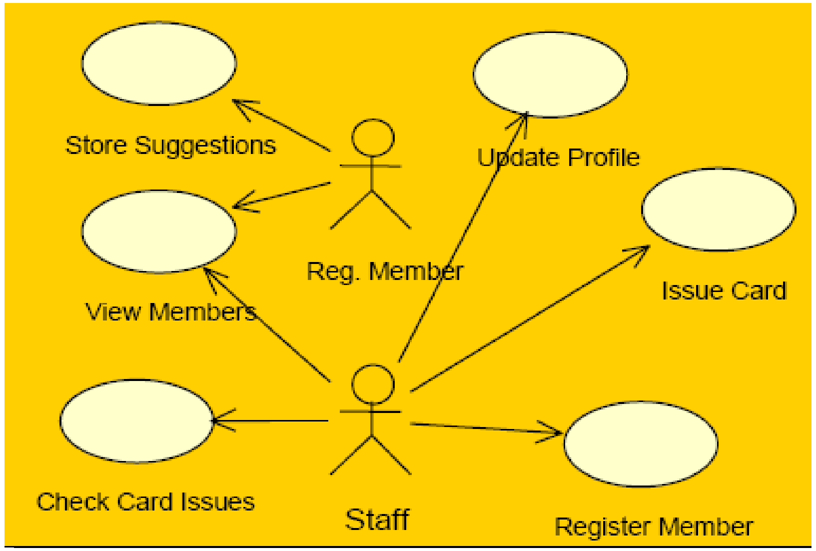
- Visual Modeling captures Business process
  - Use Case Analysis
  - Understand WHAT is to be built before trying to build the system
- Visual modeling is a communication tool
  - Common notation
- Visual Modeling manages complexity
  - Real world systems have hundreds, sometimes thousands of classes
  - Visual modeling defines packages
- Visual Modeling promotes reusability
  - Easily see what is available for reuse (Class, package, pattern … etc)

UML Diagrams

- Use case diagram
- Class diagram
- State chart diagram
- Activity diagram
- Sequence diagram
- Collaboration diagram
- Component diagram
- Deployment diagram

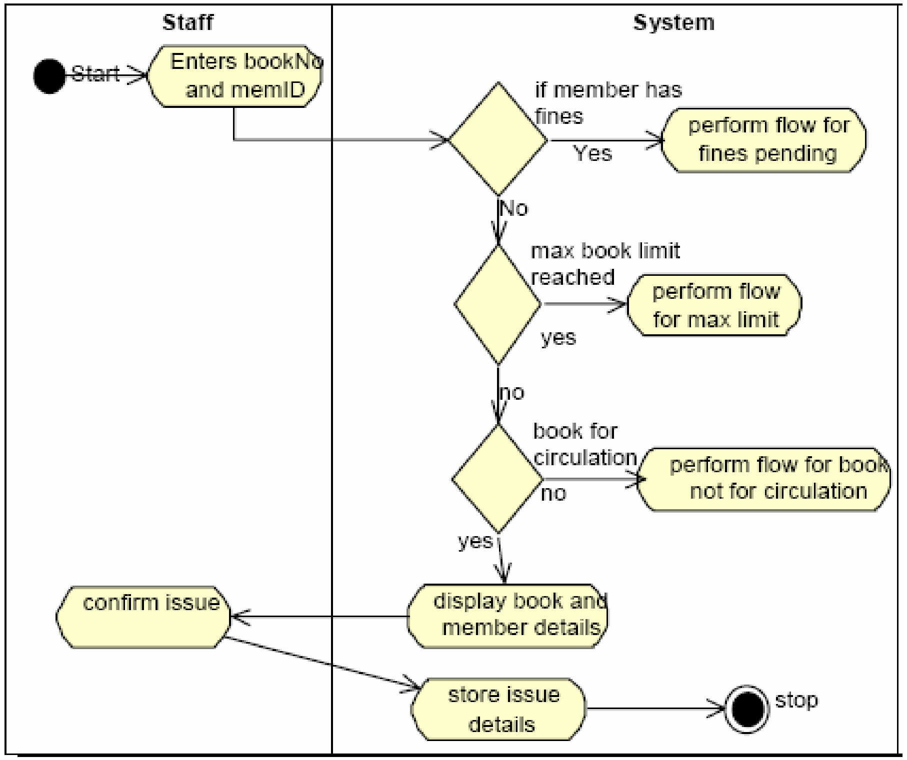
Use case diagram

- Use Case diagrams are created to visualize the interaction of your system with the outside world



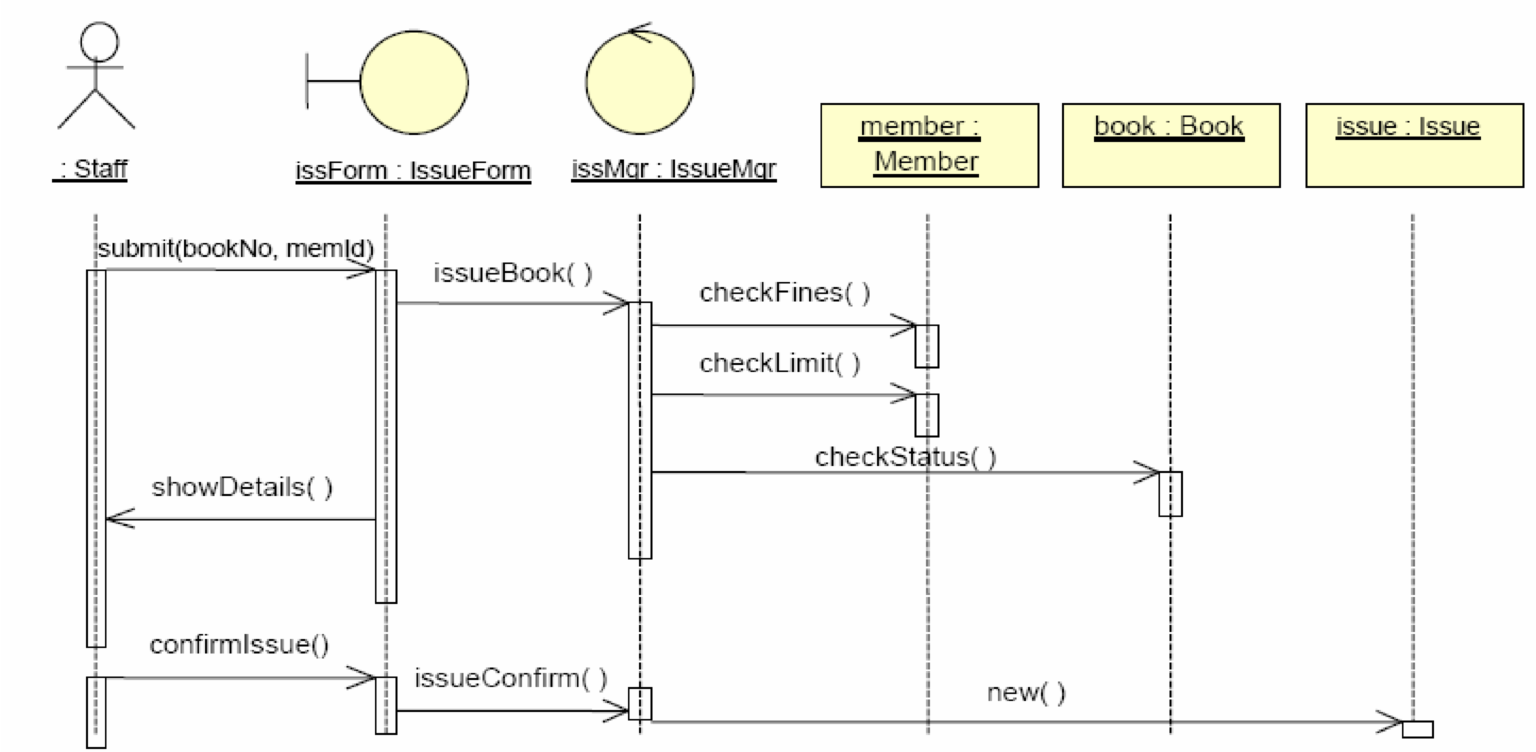
Activity diagram

- An activity diagram shows the flow of events within our system



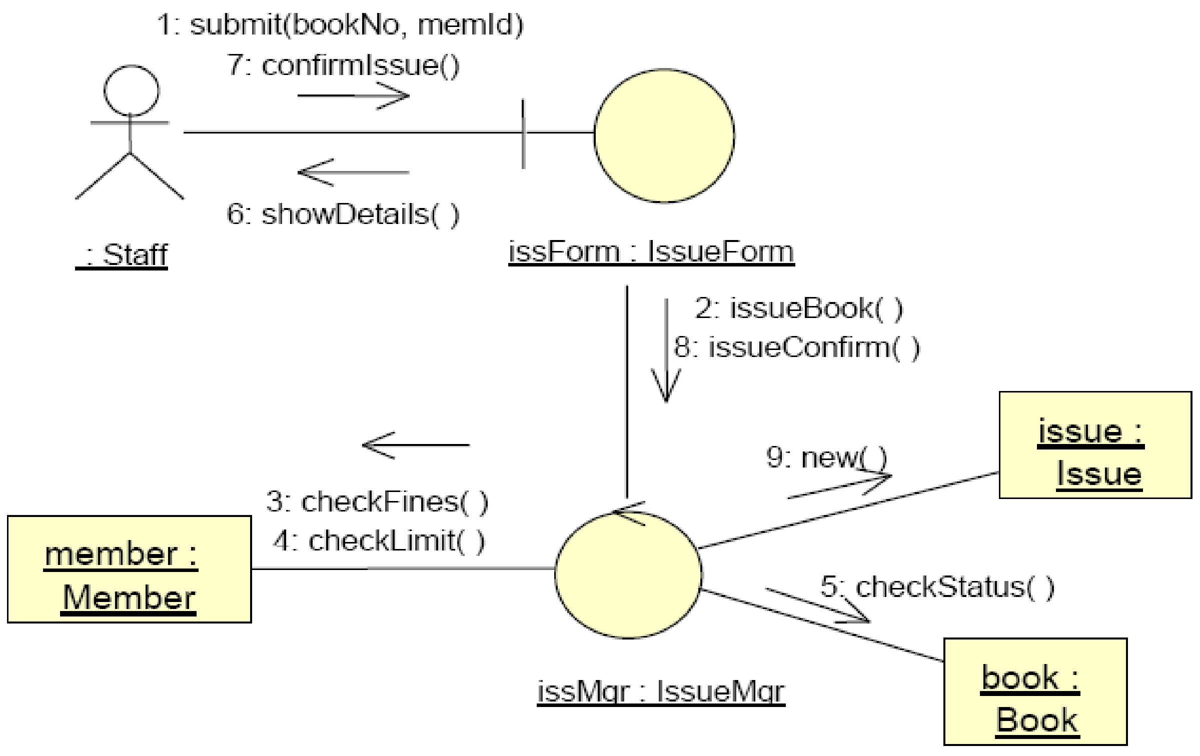
Sequence Diagram

- A sequence diagram shows step by step what must happen to accomplish a piece of functionality provided by the system



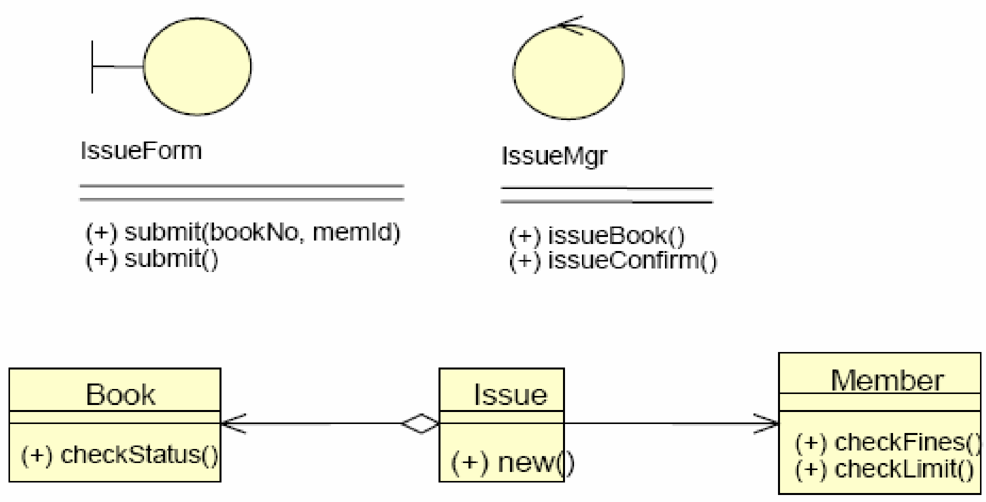
Collaboration Diagram

- A collaboration diagram displays object interactions organized around objects and their links to one another



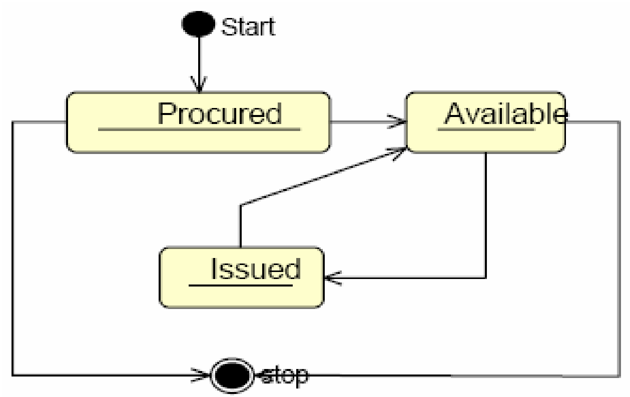
Class Diagram

- A class diagram shows the structure of your software



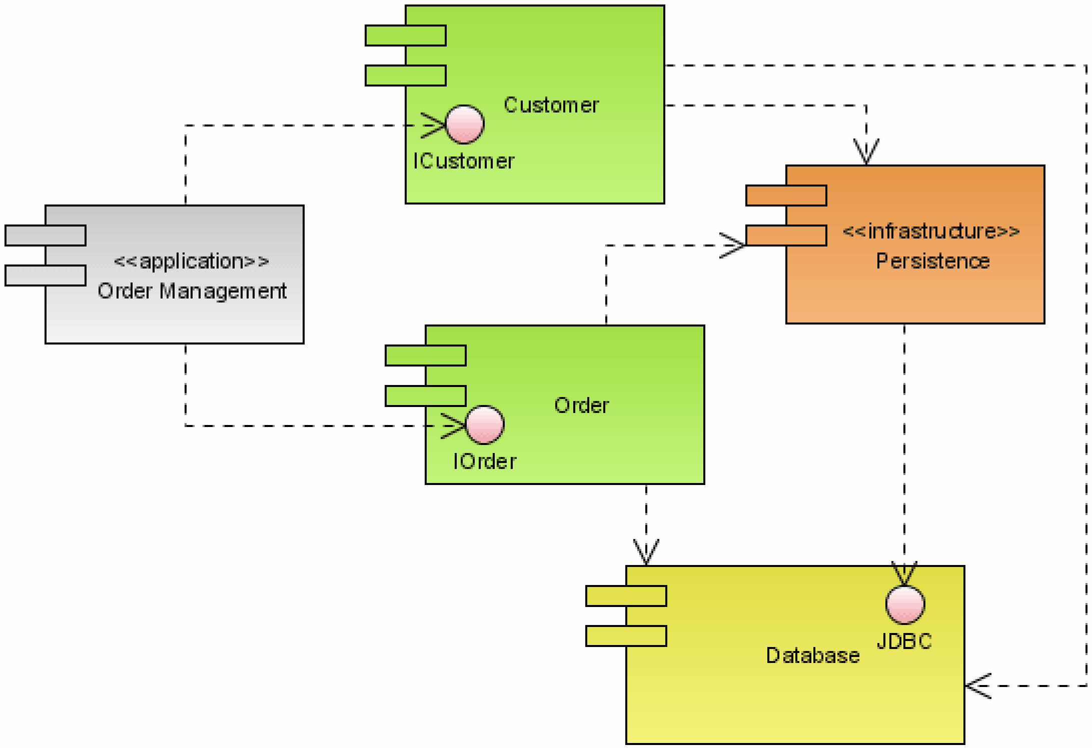
State Diagram

- A state diagram shows the lifecycle of a single class and its instance



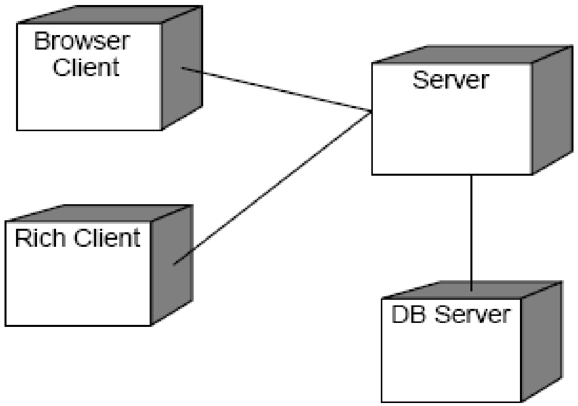
Component Diagram

- A component diagram illustrates the organization and dependencies among software components



Deployment Diagram

- A deployment diagram visualizes the distribution of components across the enterprise

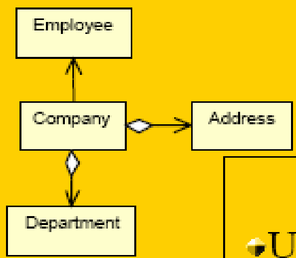


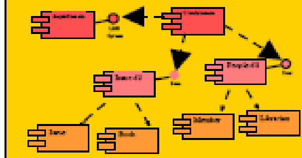
How do I construct a model

- Models are constructed using different views and diagrams to depict varying perspectives
- Views
  - A view is a perspective of the model that is meaningful to specific stakeholders
- Different views
  - Use case view
  - Logical view
  - Process view
  - Component view
  - Deployment view

4+1 Architectural view

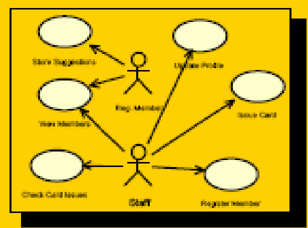
- Logical view
  - Analysts
  - Designers
- Structure



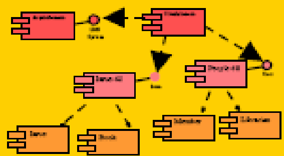


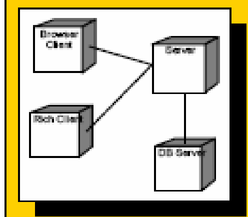
- Implementation view
  - Programmers
- Software management

- Use case view
  - End user
- Functionality



- Process view
  - System Integrators
- Performance
- Scalability
- Throughput





- Deployment view
  - System Engineers
- Topology
- Delivery, installation

Development Process

Why use a process?

- Guide through projects
  - using proper notations
  - Coordinating activities of the team
- Divides process into various phases
  - Provide guidelines to each phase
  - Easier to measure progress
- Guidelines for utilizing resources
  - People, equipments, Finances
- Guidelines for preparing and maintaining documents

Traditional Software Development process

- Contains five generic phases
  - Requirement capture
  - Analysis
  - Design
  - Implementation
  - Testing

Requirements Capture

- Finding Customer Needs
- Find out what the current problem is?
- Prepare detailed SRS document
- Use-case diagrams
- Initial Class diagram and Activity diagram

Analysis

- Analyzing information from requirements capture
- Modeling Real world entities
- Use-cases expanded and used to find classes
- Relate the classes
- Simple mock UI Screens to get user feedback
- No implementation and technology details
- High level class diagrams
- Activity, sequence, collaboration and state diagrams

Design

- Fleshing analysis model
- Detailed attributes and operations of each class
- Detailed SDS document
- Classes grouped into packages
- Domain classes got from analysis
- Technical classes
  - Persistence, interprocess communication, error and exception handling
- UI in terms of window layout, number of windows, event handling etc.
- Different outputs from the system studied
- Some classes mapped to DB Tables

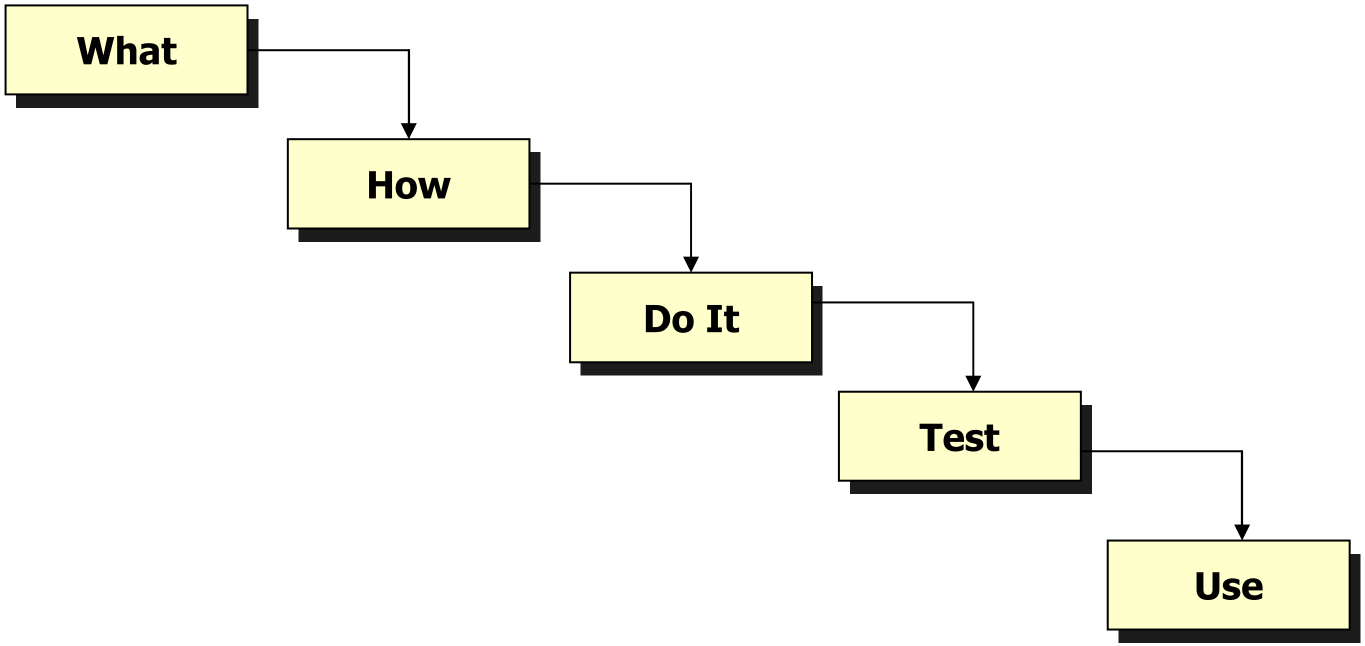
Implementation

- Coding will begin
- Reviewing code for reusability, quality etc.
- Code components integrated
- Whole system compiled, linked and debugged
- UML diagrams updated and corrected

Testing

- Eliminating errors
- Writing test cases and running them
- Unit tests, integration tests, system tests, regression tests etc.
- Automated using specialized tools

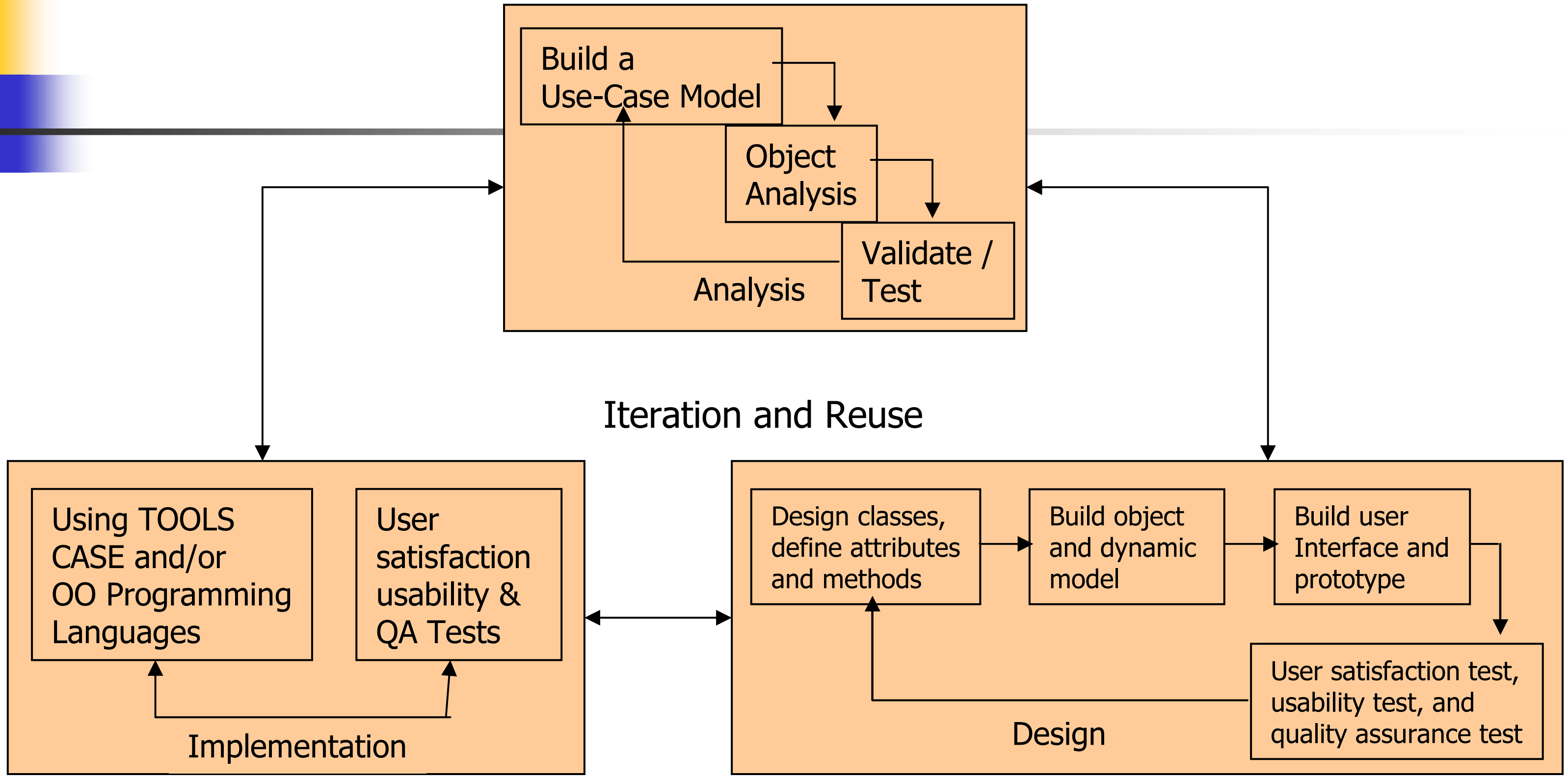
Waterfall Model



Waterfall Approach

- Simple and brings process to software development
- Limitations and Drawbacks
  - Testing occurs in the end – No early feedback
  - Potential risks discovered in the end
  - If potential defects encountered in testing, whole system might change (more time and money)
  - No balance of work in the team

Use Case driven 'iterative' approach



- Resolves major risks before making large investments
- Enables early user feedback
- Makes testing and integration continuous
- Focuses project short-term objective milestones
- Makes possible deployment of partial implementations

Rational Unified Process

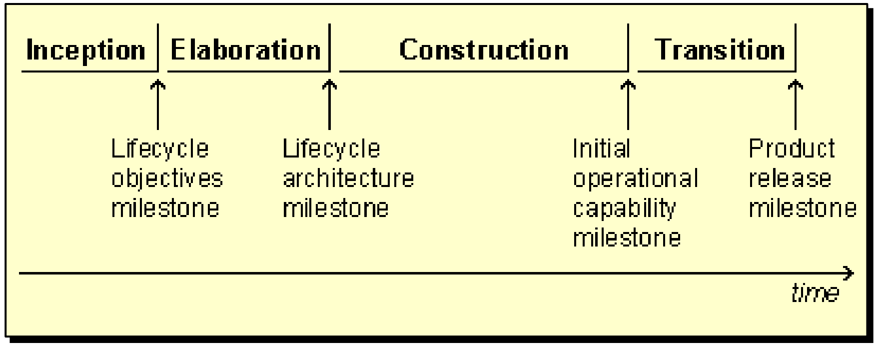
- **A small company is as good as its people, a large company is as good as its processes.**
- The RUP is a software engineering process that
  - enhances team productivity
  - delivers software best practices via guidelines, templates, and tool guidance for all critical software development activities
- Combines commonly accepted best practices
  - Iterative lifecycle
  - Risk-driven development
  - Well documented

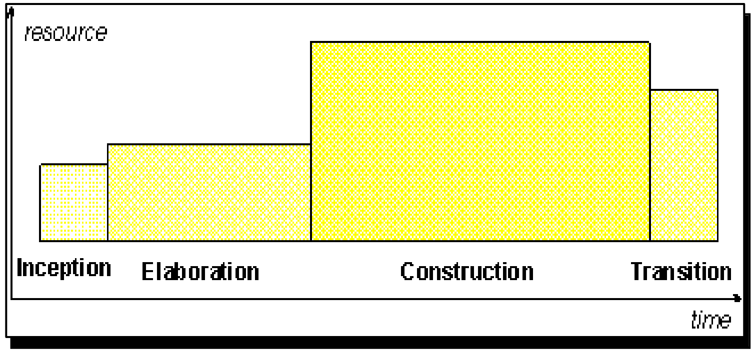
- RUP Best Practices
  - Develop iteratively
  - Manage Requirements
  - Use Component Architectures
  - Model Visually
  - Continuously Verify Quality
  - Control Changes

RUP phases

- RUP consists of 4 phases
  - Inception
  - Elaboration
  - Construction
  - Transition

- **Sample effort and schedule distribution**





	Inception	Elaboration	Construction	Transition
Effort	~5 %	20 %	65 %	10%
Schedule	10 %	30 %	50 %	10%

RUP Phases & Activities

- **Inception**
  - Approximate vision
  - Business case
  - Scope
  - Vague estimates
- **Elaboration**
  - Refined vision
  - Iterative implementation of core architecture
  - Resolution of high risks
  - Identification of most requirements
  - More realistic estimates
  - Function Points
  - Use case points
  - COCOMO
- **Construction**
  - Iterative implementation of the remaining lower risk and easier elements
  - Assessment of product releases against acceptance criteria for the vision
  - Preparation for deployment
- **Transition**
  - Beta tests
  - Deployment
  - User training

RUP Workflows (disciplines)

- In its simplest form, RUP consists of some fundamental workflows:
  - Business Engineering
    - Understanding the needs of the business
  - Requirements
    - Translating business need into the behaviors of an automated system
  - Analysis and Design
    - Translating requirements into a software architecture
  - Implementation
    - Creating software that fits within the architecture and has the required behaviors
  - Test
    - Ensuring that the required behaviors are correct, and that all required behaviors are present
  - Configuration and change management
    - Keeping track of all the different versions of all the work products
  - Project Management
    - Managing schedules and resources
  - Environment
    - Setting up and maintaining the development environment
  - Deployment
    - Everything needed to roll out the project

RUP Phases & Workflows (disciplines)

- Key characteristics of the RUP:
  - Iterative and Incremental
  - Use case-driven
  - Architecture-centric

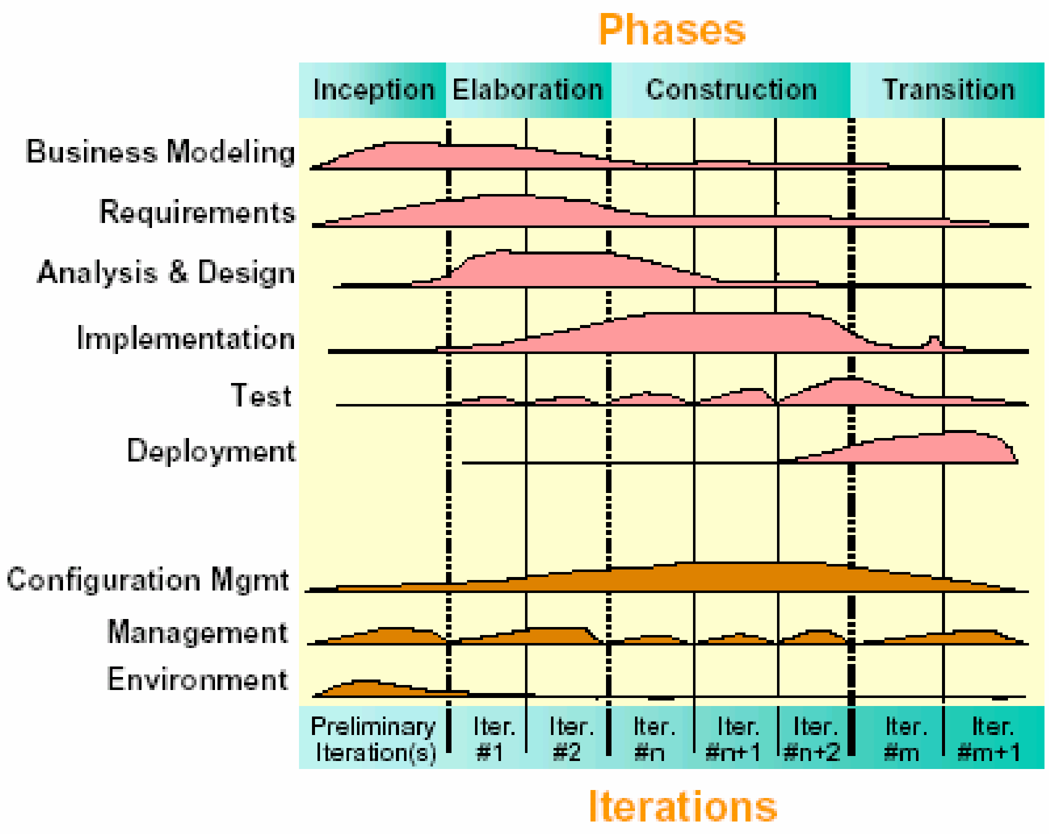
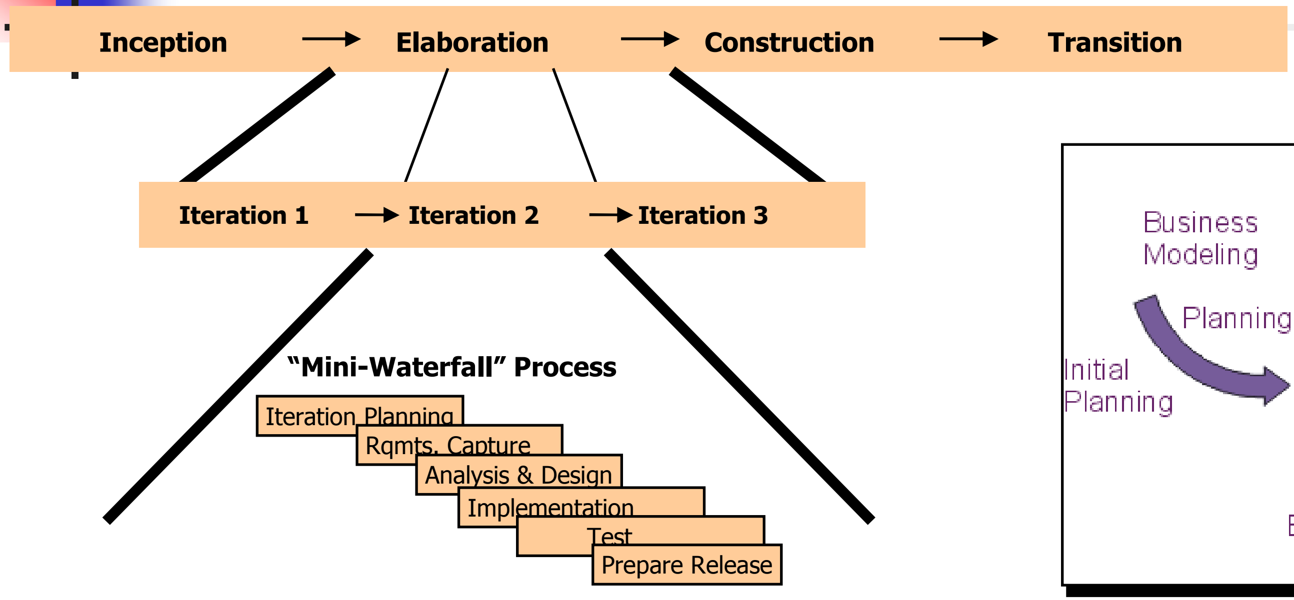


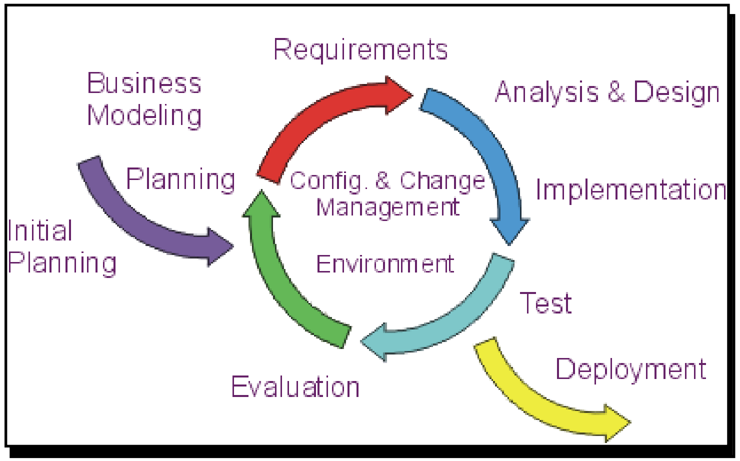
Figure 4-1
The Phases and Iterations of RUP.

- Inception
  - Not a requirements phase
  - Just a feasibility phase
- Elaboration
  - Not just the requirements or design phase
  - A phase where core architecture is iteratively implemented and high risks are mitigated

Iterative & incremental

- Each iteration is a mini-waterfall





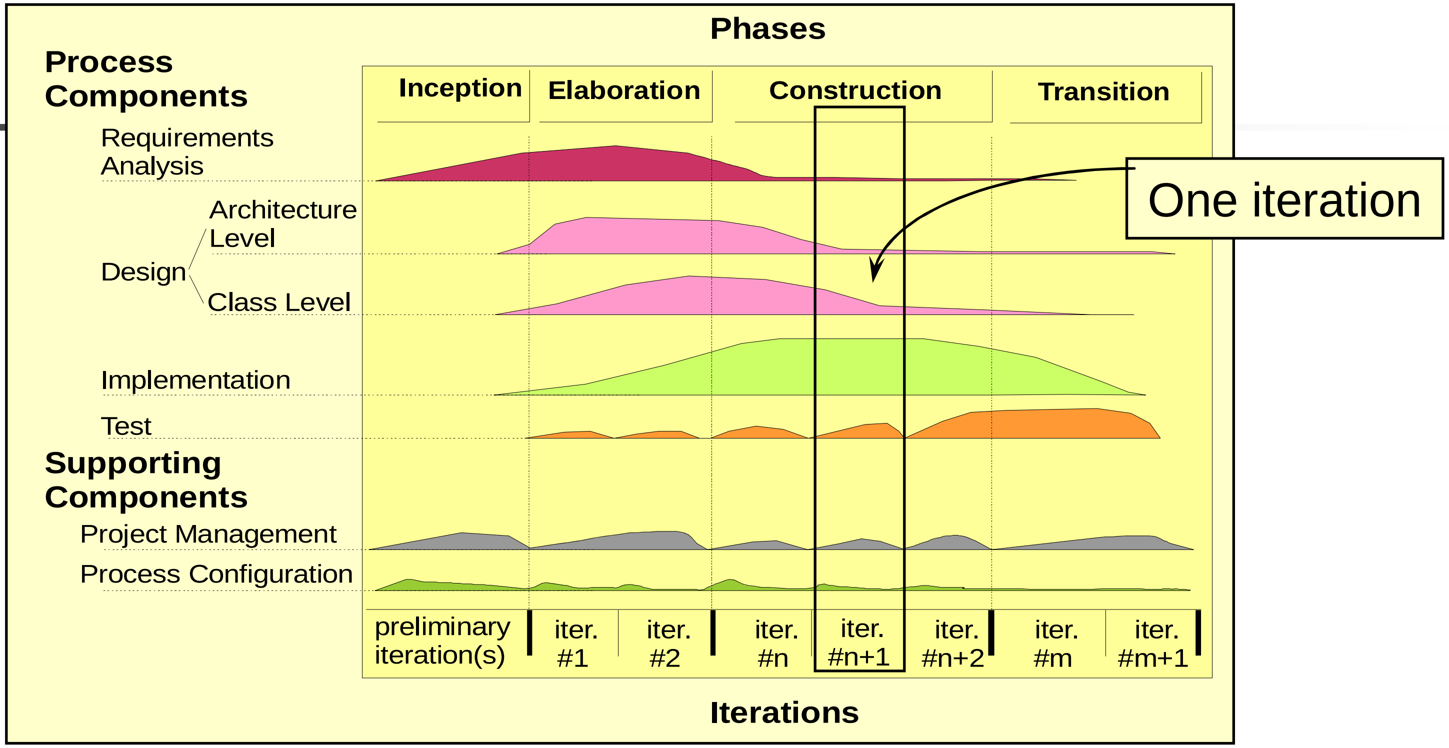
- Development is organized into series of short, fixed-length mini-projects called iterations
  - Typical iteration duration: 2 to 4 weeks
  - Each iteration includes
    - Requirements analysis
    - Design
    - Implementation
    - Testing

RUP Phases & iterations

- An **iteration** is a complete development loop ending in a **release** of an executable product, an increment of the final product under development.
- **Key features**
  - Multiple iterations
  - Cyclic feedback
  - Adaptation

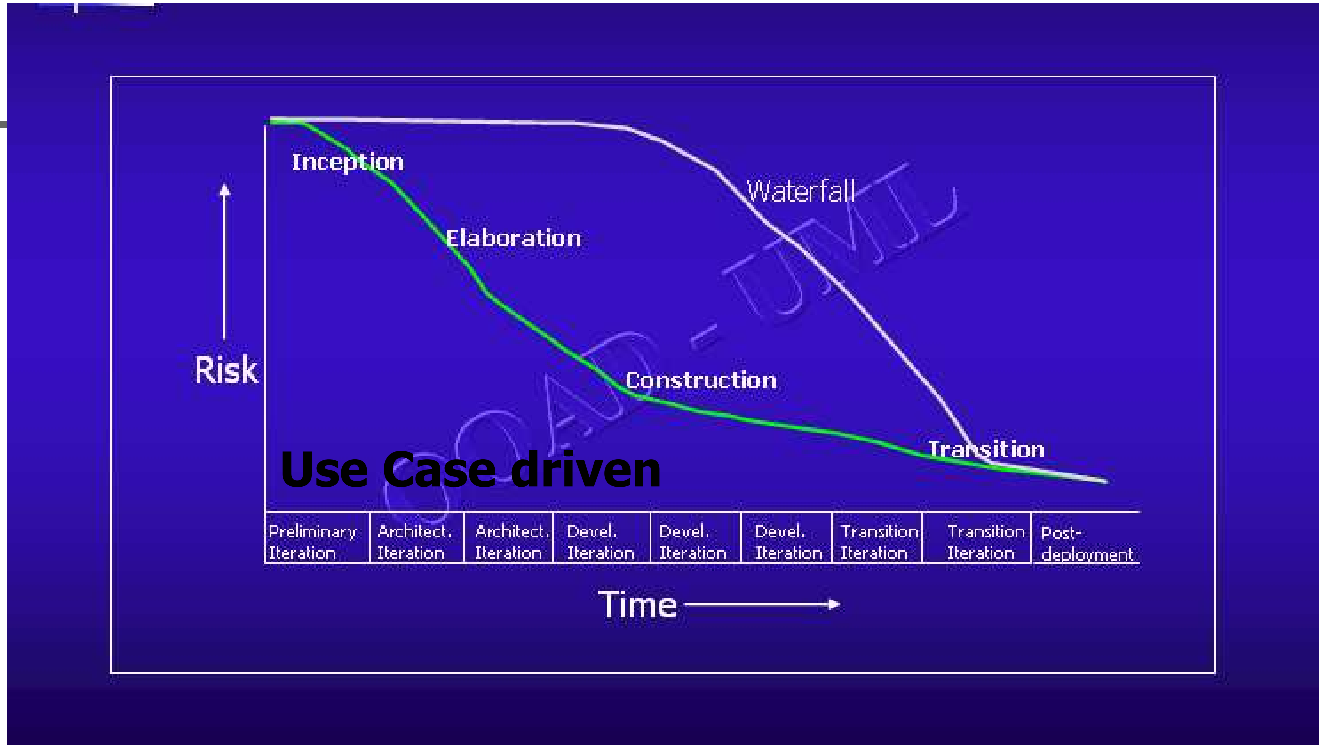
- Each Iteration begins with
  - Iteration Planning
    - Selected scenarios
    - Results of previous iterations
    - Up-to-date risk assessment
    - Libraries of models, code and tests
- Each iteration ends with
  - Release
    - Release description
    - Updated risk assessment

Iterative Development



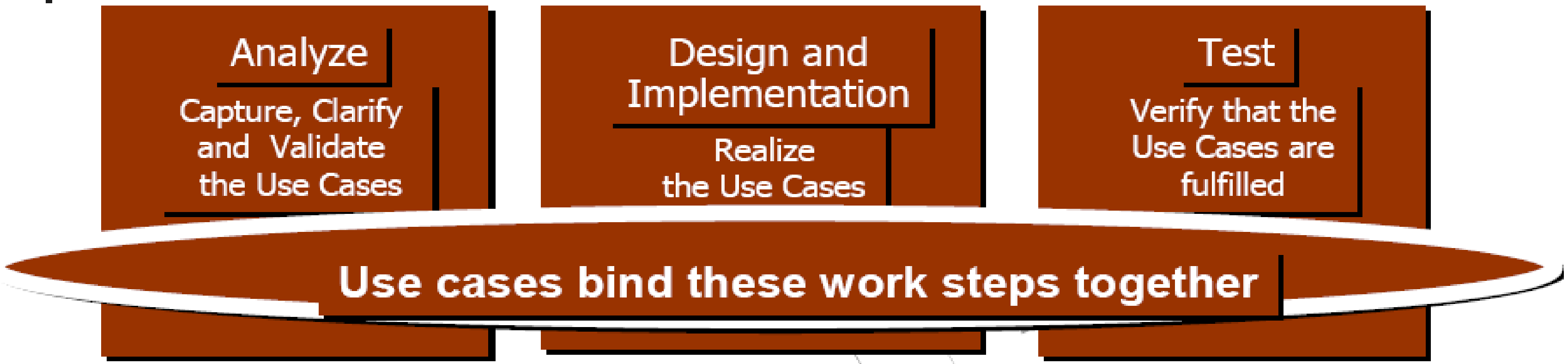
- No rush to code
- No long drawn design process
- Result of each iteration is an executable but incomplete system
  - But not an experimental or throw-away prototype
- How many iterations do I need?
  - On projects taking 18 months or less, 6 to 9 iterations are typical
- Are all iterations on a project the same length?
  - Usually Iteration length may vary by phase.
  - For example, elaboration iterations may be shorter than construction iterations

Risk Profile of an Iterative Development

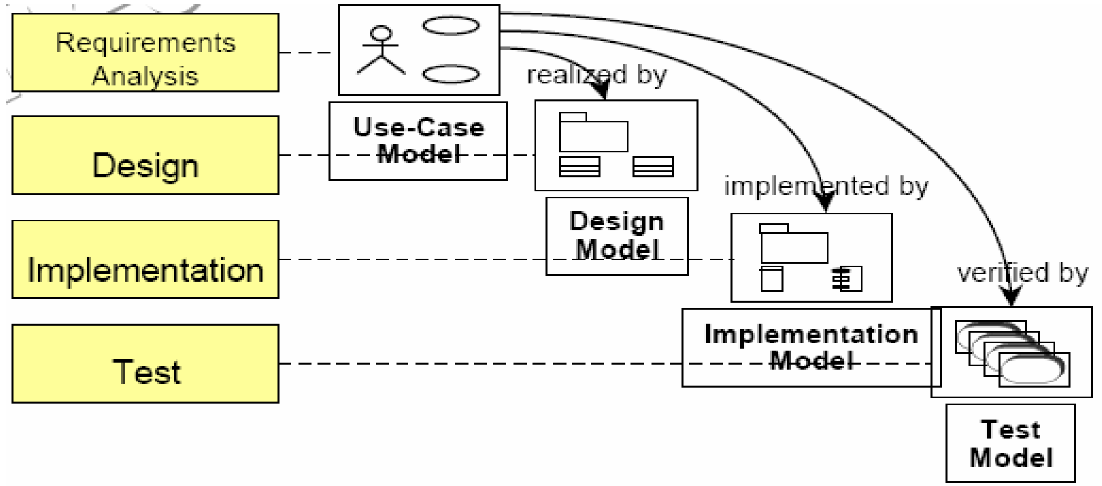


- Constant user-feedback
  - Key to successful development
  - Allows lesser speculation
- Cannot have the “90% done with 90% remaining” phenomenon
- Resolve and prove the risky and critical decisions early rather than late

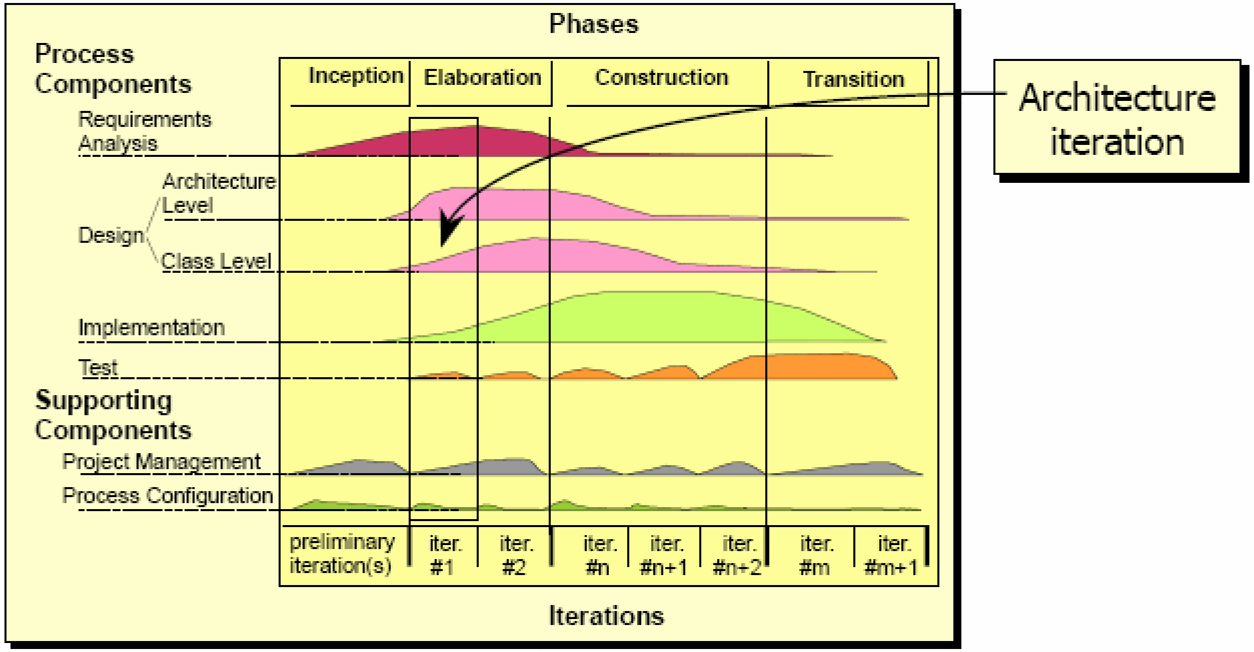
Use Case driven

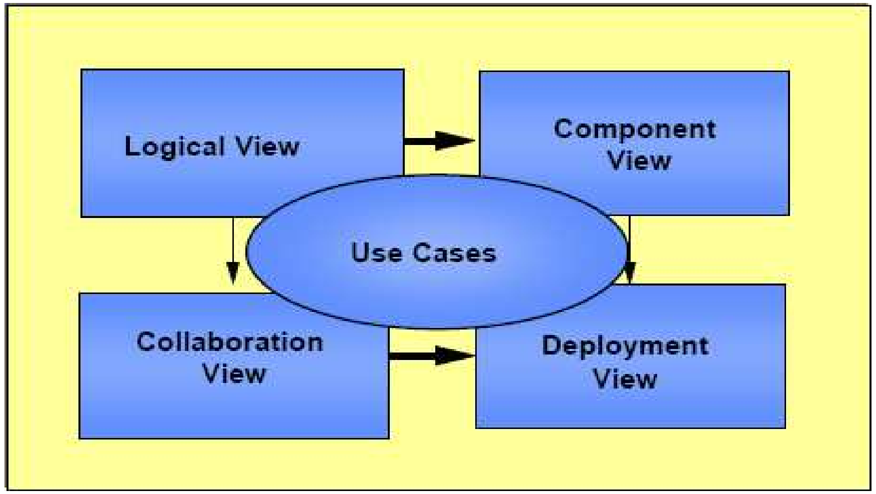


- Requirements analysis
  - Use cases are used to capture the requirements
- Design
  - Identify classes from the use-cases
- Implementation
  - Implement the use cases
- Test
  - Verify the use cases during test



RUP is architecture-centric





- Architecture is defined at the first iteration itself
- Architecture is multi-dimensional

You know you didn't understand the RUP when…

- You think in terms of a superimposing waterfall on to the UP
  - Inception = requirements
  - Elaboration = design
  - Construction = implementation
- You try to define most of the requirements before starting design or implementation
- A long time is spent doing requirements or design work before programming starts
- You believe that a suitable iteration length is four months long, rather than four weeks long
- You try to plan a project in detail from start to finish; you try to speculatively predict all the iterations, and what should happen in each one

Object orientation

Why OOP?

- Benefits of OOP
  - A more intuitive transition from business design models to software implementation models.
  - The ability to maintain and implement changes in the programs more efficiently and rapidly.
  - The ability to more effectively create software systems using a team process, allowing specialists to work on parts of the system.
  - The ability to reuse code components in other programs and purchase components written by third-party developers.
  - Better integration with loosely coupled distributed computing systems.

Why Classes?

- Supports Abstraction, Encapsulation, modularity, Inheritance, Polymorphism and Domain modeling

Key decisions creating Classes

- Simplicity
  - Follow Abstraction and SRP
  - Avoid God classes or all – encompassing
- Class Interfaces
  - Provide good abstraction for implementations
- Naming and Packaging
  - Good class name and should belong to right package

Objects

- An object represents an entity – physical, conceptual or software
- **Physical entity**
  - Employee, Customer, Television
- **Conceptual entity**
  - Sales, Tax Calculator
- **Software entity**
  - Linked List, Connection

Identifying Objects

- Example : university information system

Physical Objects

- The ***students*** who attend classes
- The ***professors*** who teach them
- The ***classrooms*** in which class meetings take place
- The ***furniture*** in these classrooms
- The ***buildings*** in which the classrooms are located
- The ***textbooks*** students use

Conceptual Objects

- The ***courses*** that students attend
- The ***departments*** that faculty work for
- The ***degrees*** that students receive

Conceptual Objects are hard to identify and need more effort.

Attributes

- Data element used to describe an object.

- Information about a student
  - The student's name
  - His or her student ID number
  - The student's birth date
  - His or her address

State

- Describes the current condition
- An object's attribute values, when taken collectively, define the **state**

Identity

- That property of an object which distinguishes it from all other objects
- Example: student ID

Behavior

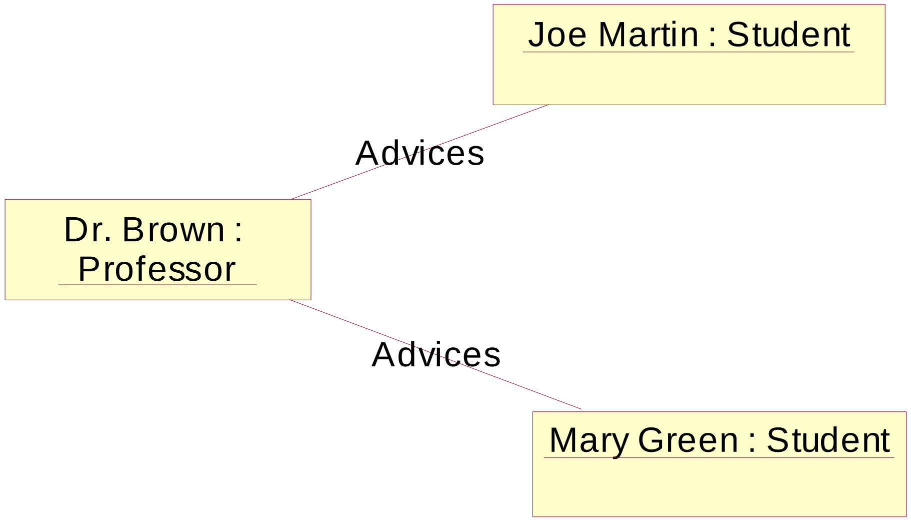
- Defines how the object respond to a specific message
- Example :
  - Knowing Student's name or ID
  - Telling a professor to teach a student

- Methods represent the behavior of an object

Object representation in UML

- When drawn objects appear as rectangles, with their names and types **underlined**
- objects that know about each other have lines drawn between them
  - This connection is known as an ***object reference***, or just, ***reference***
- Messages are sent across references

Object representation in UML



Class

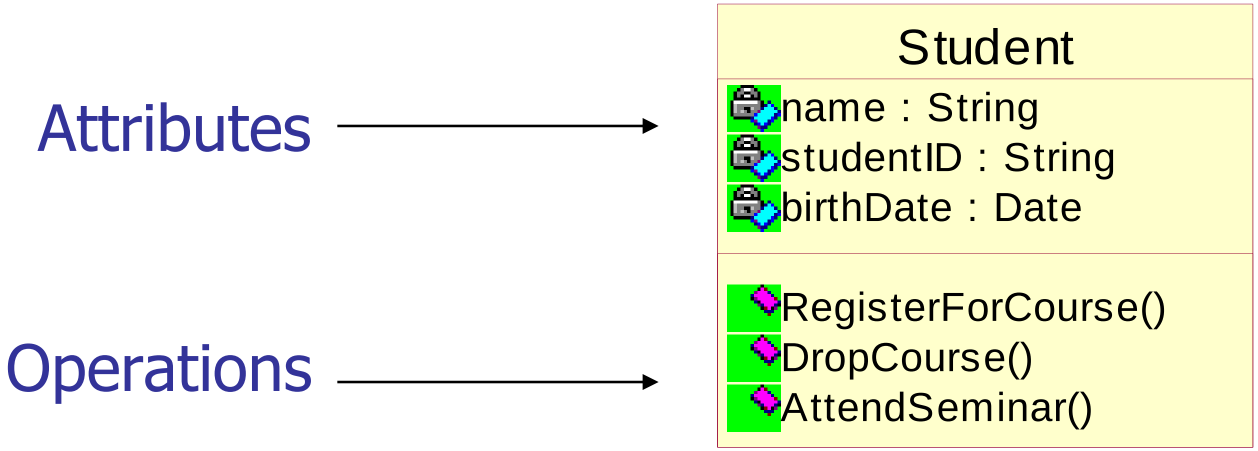
- Provides a template for object creation

- Each objects created from a class will have
  - Same attributes: may have different values
  - Same operations: may have differing results

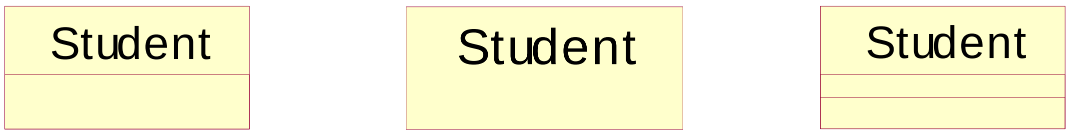
Class representation in UML

- classes appear as rectangles with three compartments
  - The first compartment contains its name (this name defines a type)
  - The second compartment contains the class's attributes (Structure)
  - The third compartment contains the class's methods (Behavior)

Class representation in UML



Other class representations



Class versus Object

- Students often have trouble with the difference between classes and objects

- For Example: ‘Apple’ is a class
  - A grocery store may have hundreds of instances of this class
  - Large apples, small apples, red apples, green apples, etc.

Object Instantiation

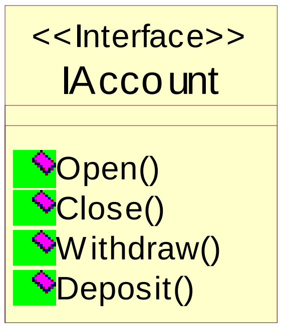
- An object is an instance of a particular class.

```
public class Student {
}
class MyClass {
    Student joe = new Student();
    Student mary = new Student();
}
```

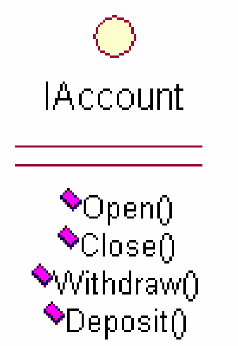
Interface

- An interface has only abstract behavior (key abstractions) and no structure
- Standard UML offers no different symbol for interface
  - Instead a stereotype is used to extend the class symbol to represent an interface

UML Representation



Rational Rose representation



Interface

```
public interface IAccount {

void Open();
void Close();
double Withdraw();
void Deposit(double amount);
}
```

Abstraction

- *Abstraction* is used to manage complexity
  - Focus on the essential characteristics
  - Eliminate the details
  - Find commonalities among objects
- Highlight the characteristics and behavior of something that is too complex to understand in its entirety
- Different people would build different abstractions for the same concept

Abstraction example

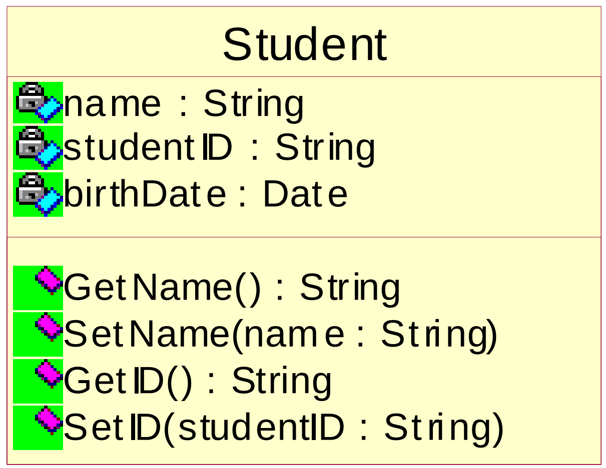
- Represent a car in rental bookings system we might abstract a model of a car which describes make, model, year, kms,…
  - Not model the detailed mechanical, aerodynamic properties of the car.
- A system to assist an automotive engineer to simulate the capabilities of a car, we might provide mechanical, aerodynamic properties of the car.

Encapsulation

- Mechanism of bundling together the state and behavior of an object into a single logical unit.

Encapsulation hides data and implementation

- Often called “information hiding”
- An Object grants access to the information through a proper interface.

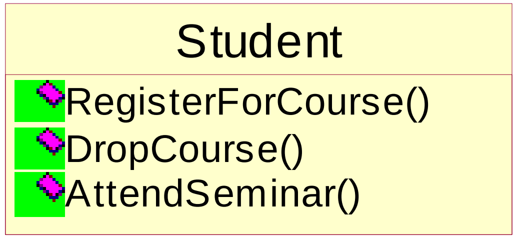


Advantages of Encapsulation

- State cannot be changed directly from outside hence limiting "ripple effects".
- Implementation can change without affecting users of the object
- Promotes modular software design – data and methods together

Identifying Abstraction and Encapsulation

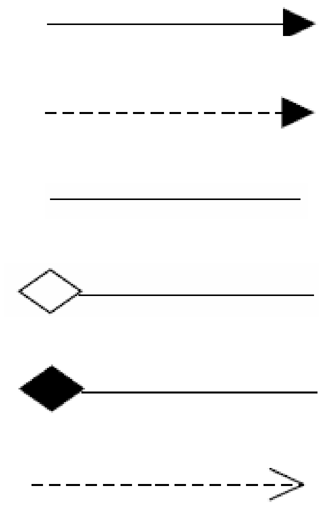
- **ABSTRACTIONS:**
  - What should the Student do?
  - What are the responsibilities of a Student?



- **ENCAPSULATION:**
  - What all should the Student contain (encapsulate) to meet its responsibilities?
  - What are all needed to provide an implementation for the ABSTRACTIONS?

Class Relationships

- Relationships between classes
  - Generalization
  - Realization
  - Association
  - Aggregation
  - Composition
  - Dependency



Class Relationships

- Classification

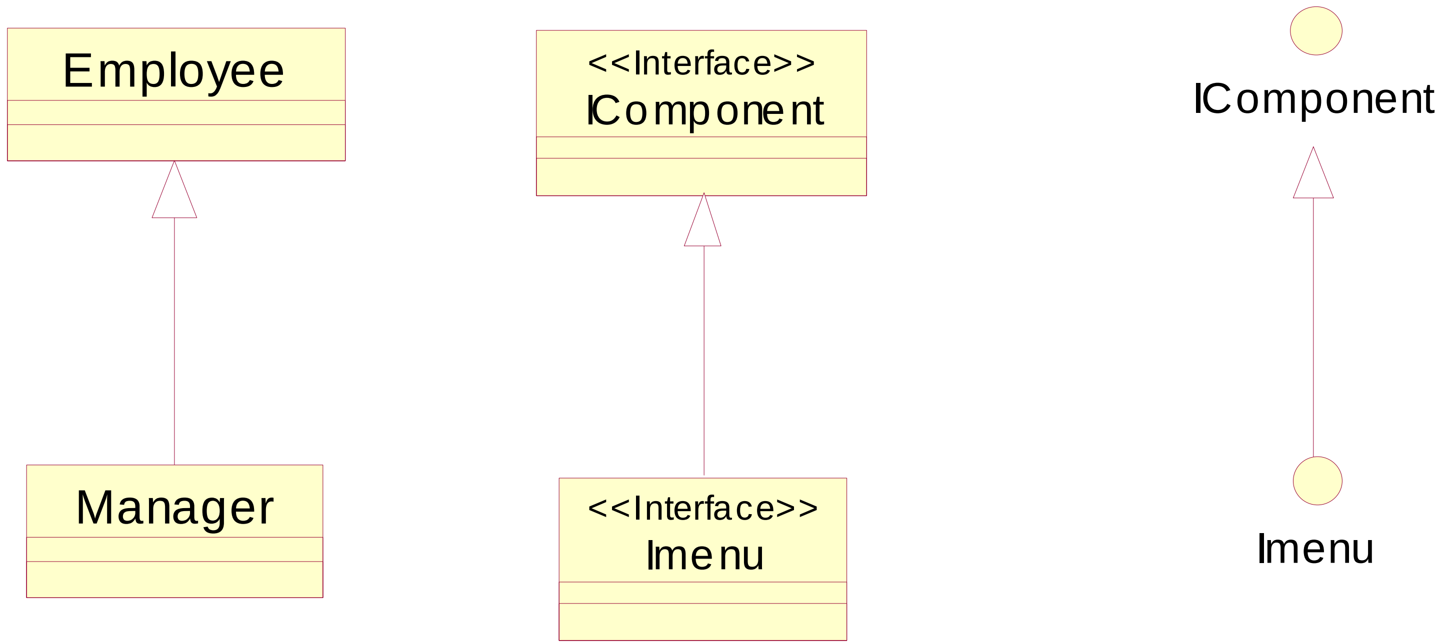
<<Is-a>>	<<Has-a>>	<<Uses>>
Generalization	Association	Dependency
Realization	Aggregation	
	Composition	

Generalization

- Relationship between two classes or two interfaces
- Benefits of generalization :
  - Reuse and extend code that has already been thoroughly tested
  - Can derive a new entity from an existing entity even if we don't own the source code for the latter!
  - Increased productivity, reduced maintenance and improved reliability

Generalization

Class Manager : Employee { }



Guidelines for Generalization

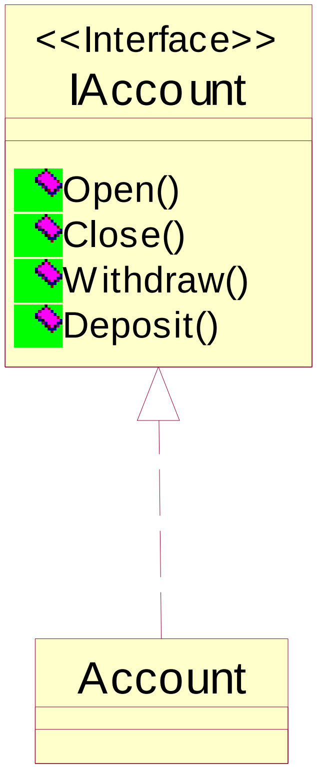
- Pass “IS-A” test
- Promote Polymorphism
- Minimize inheritance level

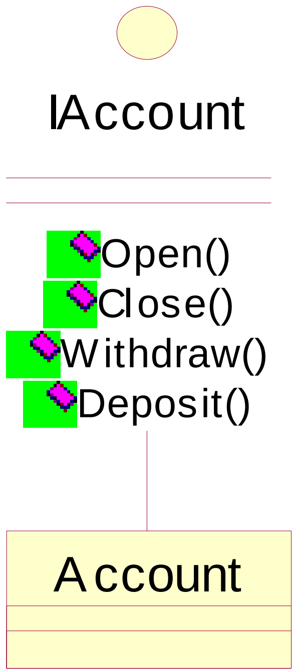
Realization

- Relationship between a class and an Interface.
- class provides implementation for all the inherited method declarations
- Inheritance by contract
  - Serves as a contract between the provider and the user
- No functionality is inherited, only the interface is inherited

Realization

Class Account : IAccount { }





Association

- 'Has-a' relationship
- Relationship between two or more classifiers that involve connections among their instances
- An object keeps a reference to another object and can call the object's, methods as it needs them

Association


```
Class Company {

    Customer joe = new Customer();
}
```

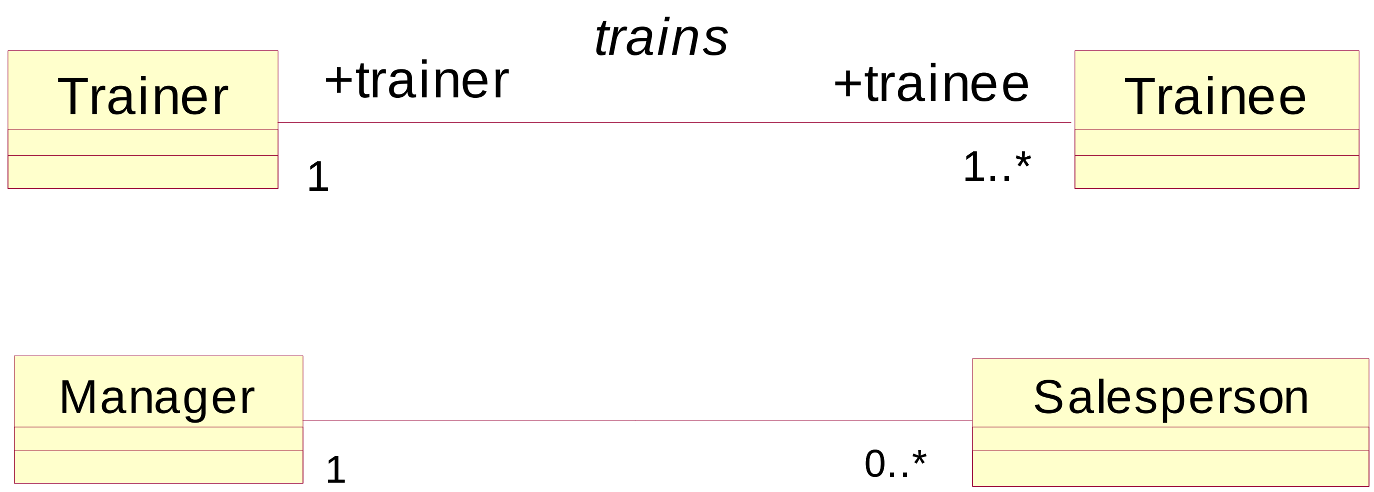
Association

The associations are qualified by

- Multiplicity
  - The number of instances with which a class is associated
  - Can be 1, 0..1, *, 1..*, 0..*, 2..*, 5..10, etc.
  - Multiplicity is by default 1
- Navigability
  - Can be unidirectional or bidirectional
  - Navigability is by default bi-directional
- Role name
  - The name of the instance in the relationship
  - Multiple associations based on different roles are possible

Association

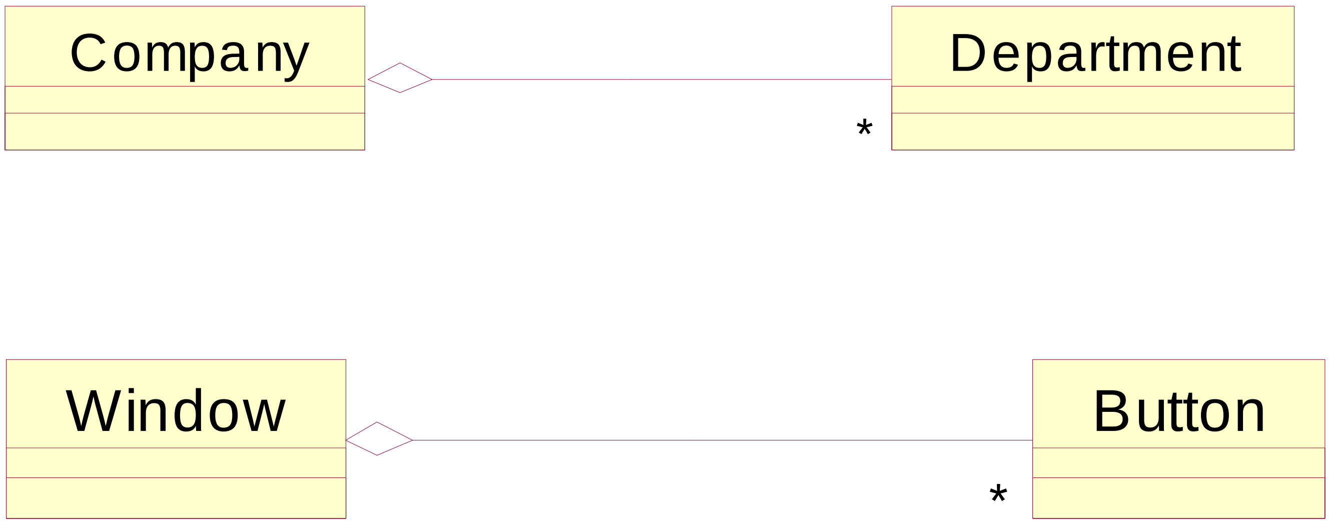
- Role name, navigability and multiplicity



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



Difference between Association and Aggregation

- When it comes to code nothing

- Association
  - Not a “whole – part” relationship
- Aggregation
  - kind of containment

- Leave Aggregation out

Composition

- ‘Has-a’ relationship
- Is a stronger form of aggregation
- A part cannot be shared between many wholes
  - Unshared aggregation
- Explicit lifetime control
- Exclusive ownership
- Has multiplicity and navigability
  - Multiplicity at the whole end should always be 1
  - Navigability is by default bi-directional

Composition

Difference between Aggregation and Composition

- Aggregation
  - Part can exist on its own
  - sometimes shared ownership
- Composition
  - part cannot exist (or is useless) on its own
  - Exclusive ownership

Difference between Association and Composition

- Composition
  - Strong form of containment
  - Only whole is visible

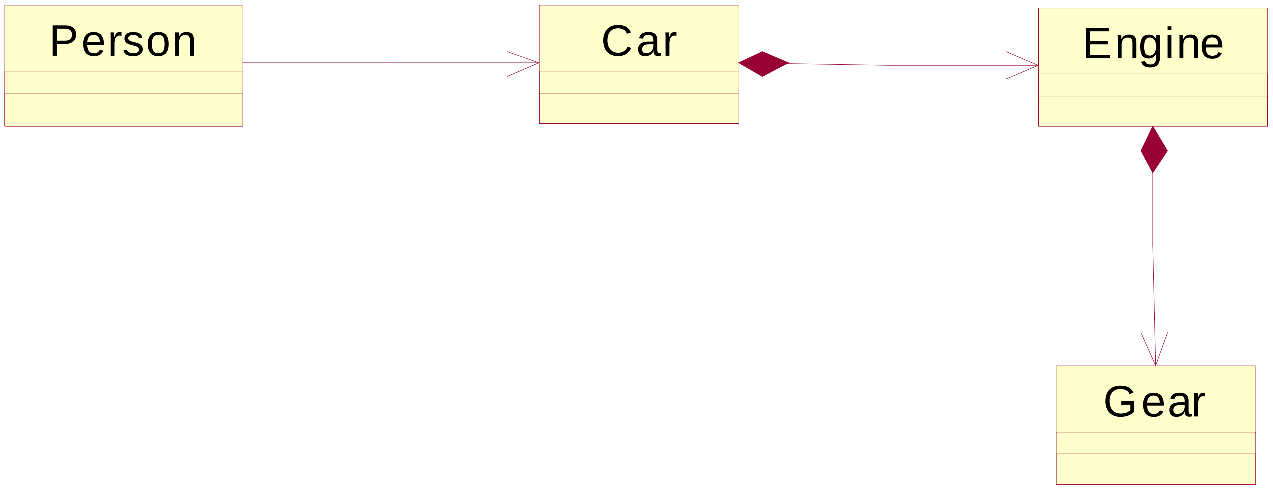
- Association
  - Interaction between objects (No containment)
  - More flexibility as classes can be swapped

Inheritance vs Composition

- Disadvantages of Inheritance
  - Changes made to the parent might affect the child classes (Strong Dependency)
  - If there is a problem in polymorphic tree you need to examine all classes
  - While designing a subclass, need to have knowledge of superclass (Opacity)
- Advantages of composition
  - Composition also promotes reusability
  - Loose-coupling (Less reliance on objects)

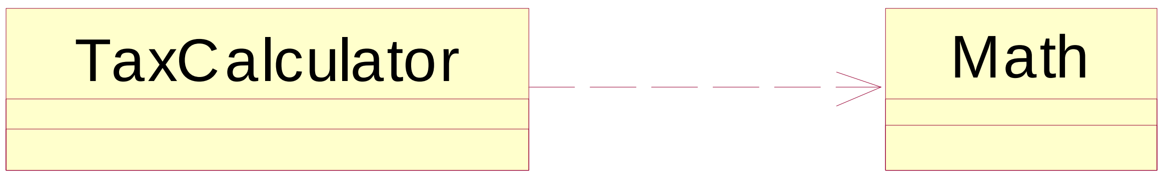
Navigability

- The arrow indicates direction of control



Dependency

- 'Uses' relationship
- The target element (client) is dependent on the source element (supplier).
- If the source element changes, the target element may require a change



Abstract Class

- A class whose main purpose is to define a common interface for its subclasses
- Defers some or all of its implementation to its subclasses
- Cannot be instantiated
- Only way C++ can come close to what an interface does in Java and C#.

Interface and Abstract class

- Deciding between Interface and an Abstract class
- Abstract Classes
  - Share common code between sub-classes
- Interfaces
  - Subclasses may not share common functionality
  - More polymorphism and more flexibility

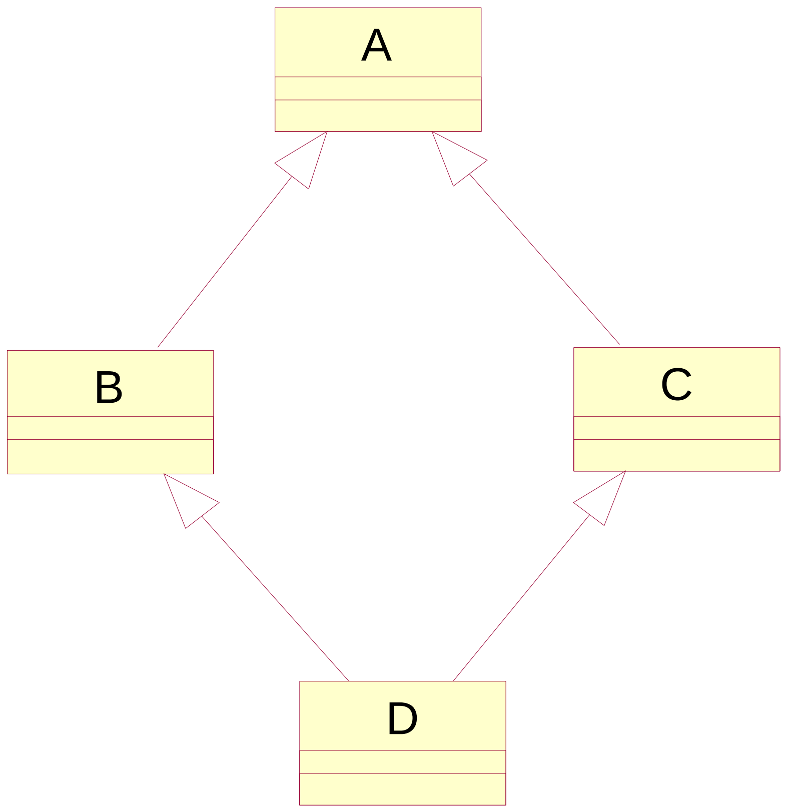
Multiple Inheritance

- The two types of inheritance are :
  - Implementation Inheritance (Generalization)
  - Interface Inheritance (Realization)
- Eliminate multiple inheritance between classes.
- A class can realize multiple interfaces.

Why not multiple implementation inheritance?

- Redundancy in implementation reuse
- Ambiguity in invoking methods of the Base classes (Diamond problem)

Diamond Problem

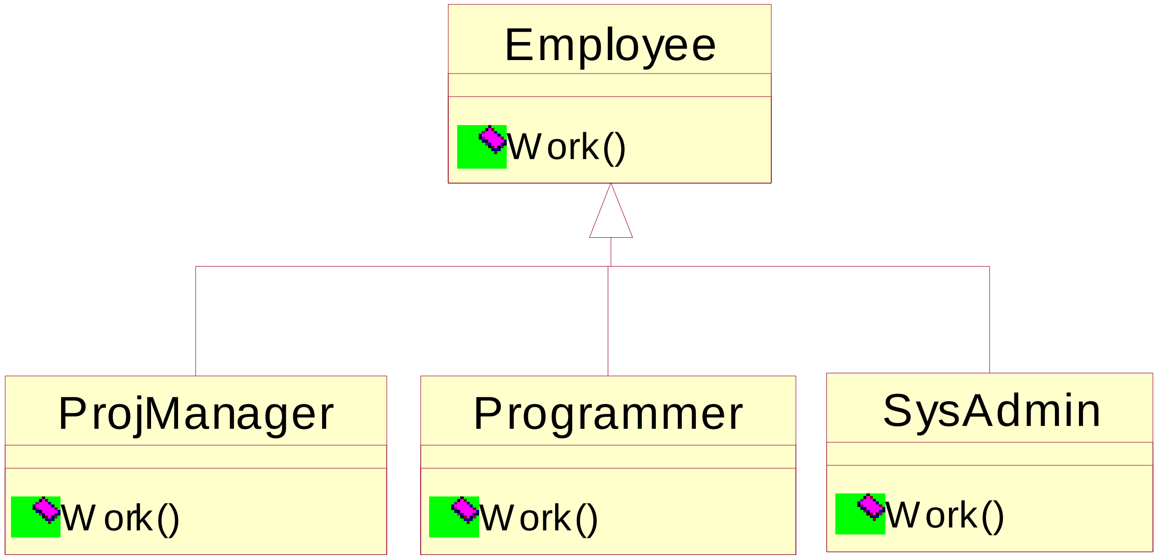


```
abstract class Animal  {
     abstract void Talk();
     }

class Frog : Animal  {
     void override Talk()
     {
              Print ("Croak, Croak");
      }
}

class Dinosaur : Animal  {
     void override Talk()
     {
               Print ("Make deafening noise");
     }
}

class Frogosaur : Frog, Dinosaur { }

//Client Code
Animal animal = new Frogosaur();
animal.Talk();
```

Polymorphism

- **Polymorphism** allows the same message to be responded to differently by different types of objects.
- Polymorphism is achieved through IS-A relationship.
- Polymorphism enables dynamic binding.
- Polymorphism Simplifies Code Maintenance.

Polymorphism

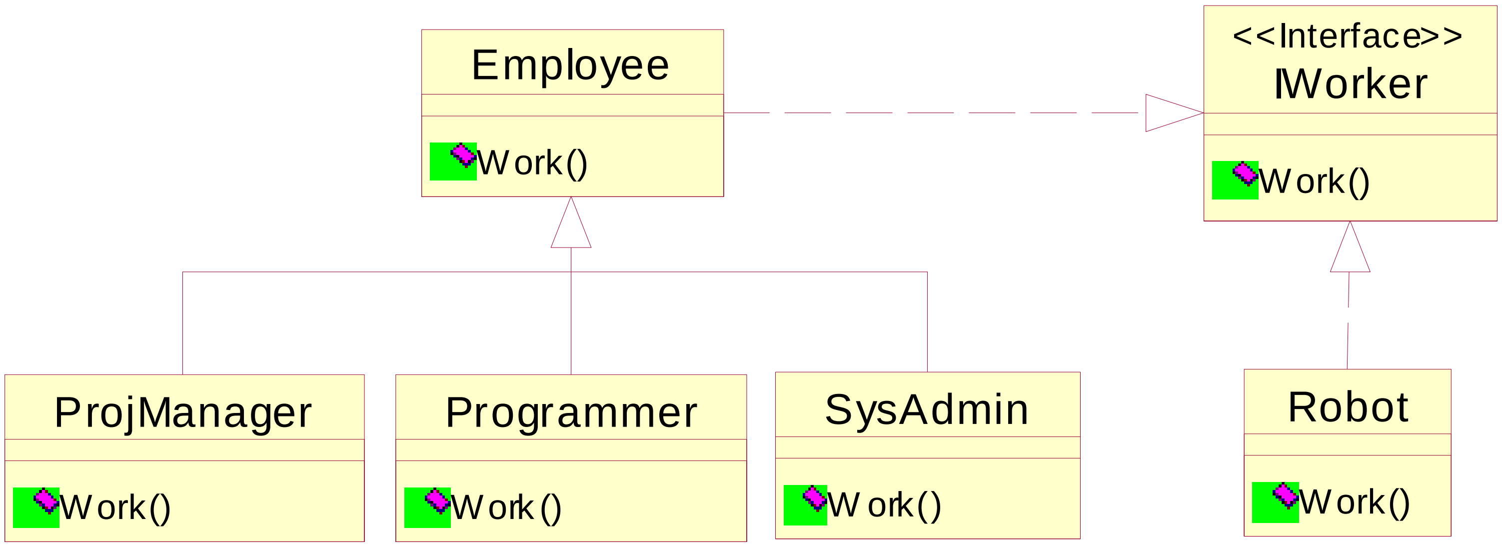


```
abstract class Employee  {
      abstract void Work();
      }

class ProjectManager : Employee  {
      void override Work()
      {
                Print ("Managing Project");
       }
}

class Programmer : Employee  {
      void override Work()
      {
                 Print ("Do some coding");
      }
}

class SystemAdmin : Employee {
      void override Work()
      {
                 Print ("Managing Systems");
      }
}

//Client Code
class Employer {
      static void MakeThemWork (Employee subject)
      {
                subject.Work();
      }
}
```

Dynamic Binding and Polymorphism

- Dynamic binding doesn’t commit an objects response to a message to a particular implementation until run-time
- Can write programs that expect a particular interface and know that any object with that interface will respond to the message
- Allows substitution at run time of objects which share the same interface

More Polymorphism through Interfaces



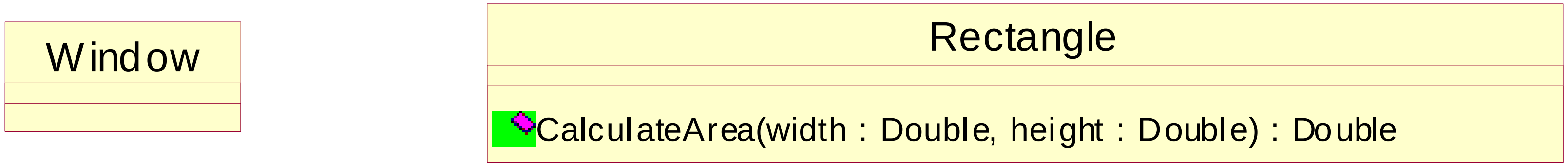

```
interface IWorker
{
   void Work();
}
```

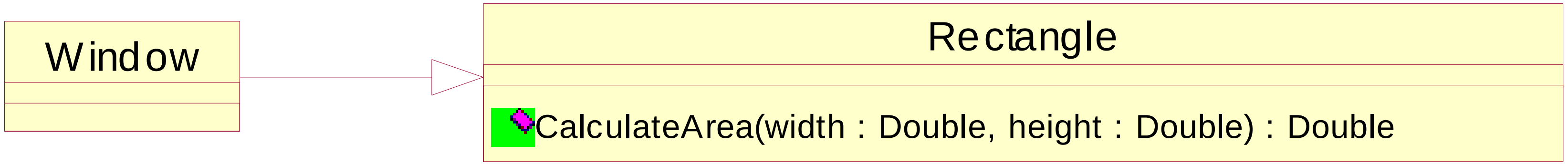
```
abstract class Employee : IWorker  {
      abstract void Work();
}

class ProjectManager : Employee  {
      void override Work()
      {
                Print (“Managing Project");
       }
}

class Programmer : Employee  {
      void override Work()
      {
                 Print (“Do some coding");
      }
}

class Robot : IWorker {
      void Work()
      {
                 Print (“Managing Systems");
      }
}

//Client Code
class Employer {
      static void MakeThemWork (IWorker subject)
      {
                subject.Work();
      }
}
```

Delegation

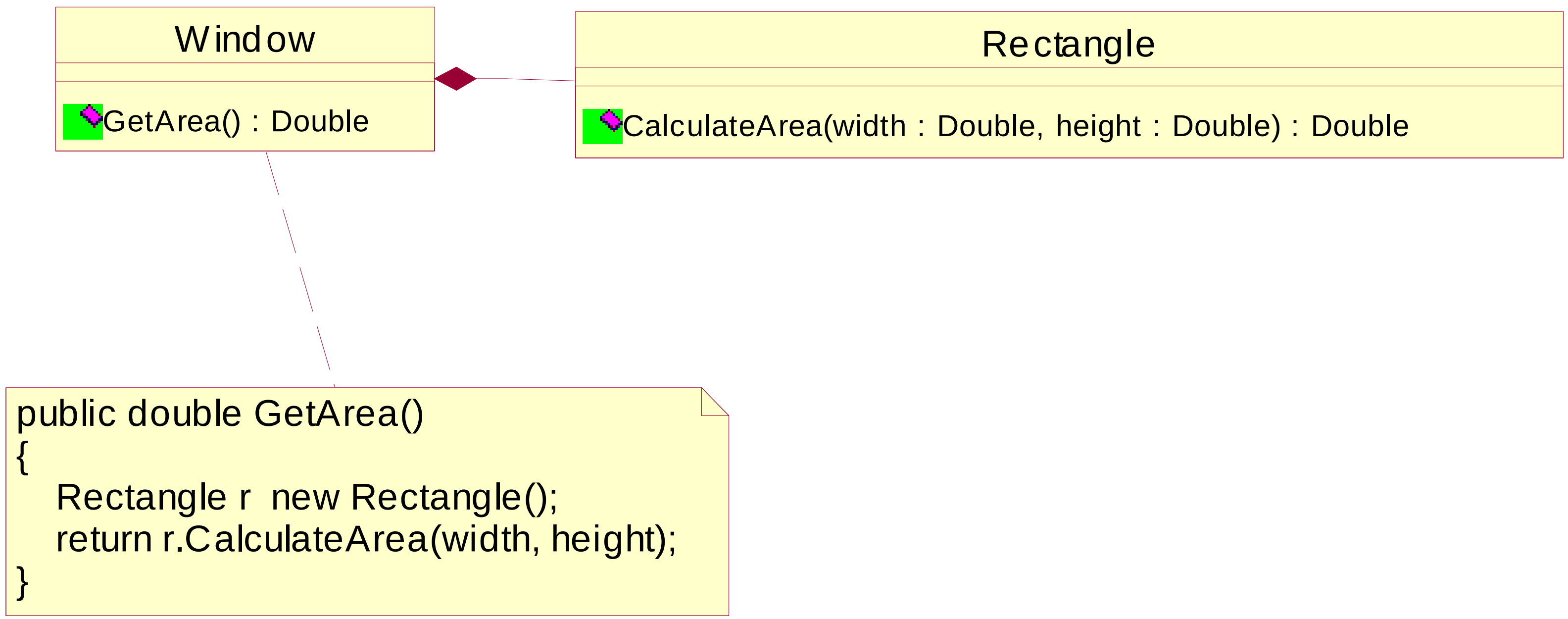
- A way for making composition as powerful for reuse as inheritance.
- Object forwards method calls to the contained object

Delegation


Window
Rectangle
CalculateArea(width : Double, height : Double) : Double


Window
Rectangle
CalculateArea(width : Double, height : Double) : Double

Delegation


Window
GetArea() : Double
Rectangle
CalculateArea(width : Double, height : Double) : Double
public double GetArea()
{
Rectangle r new Rectangle();
return r.CalculateArea(width, height);
}