EE3206
Java Programming and Applications

Lecture 4
Abstract Classes, Interfaces and OO Design Principles

Intended Learning Outcomes

- To understand abstract type
- To enforce a design protocol by using abstract classes or interface.
- To know the similarities and differences between an abstract class and an interface.

- To become familiar with the process of program development.
- To discover classes using CRC cards.
- To understand the impacts of coupling to a system.
- To learn the relationship types: association, aggregation, composition, realization and generalization.
- To understand design principles and guidelines.

Screw and Screwdriver

- There are many types of screws. How do you find a screwdriver that works?

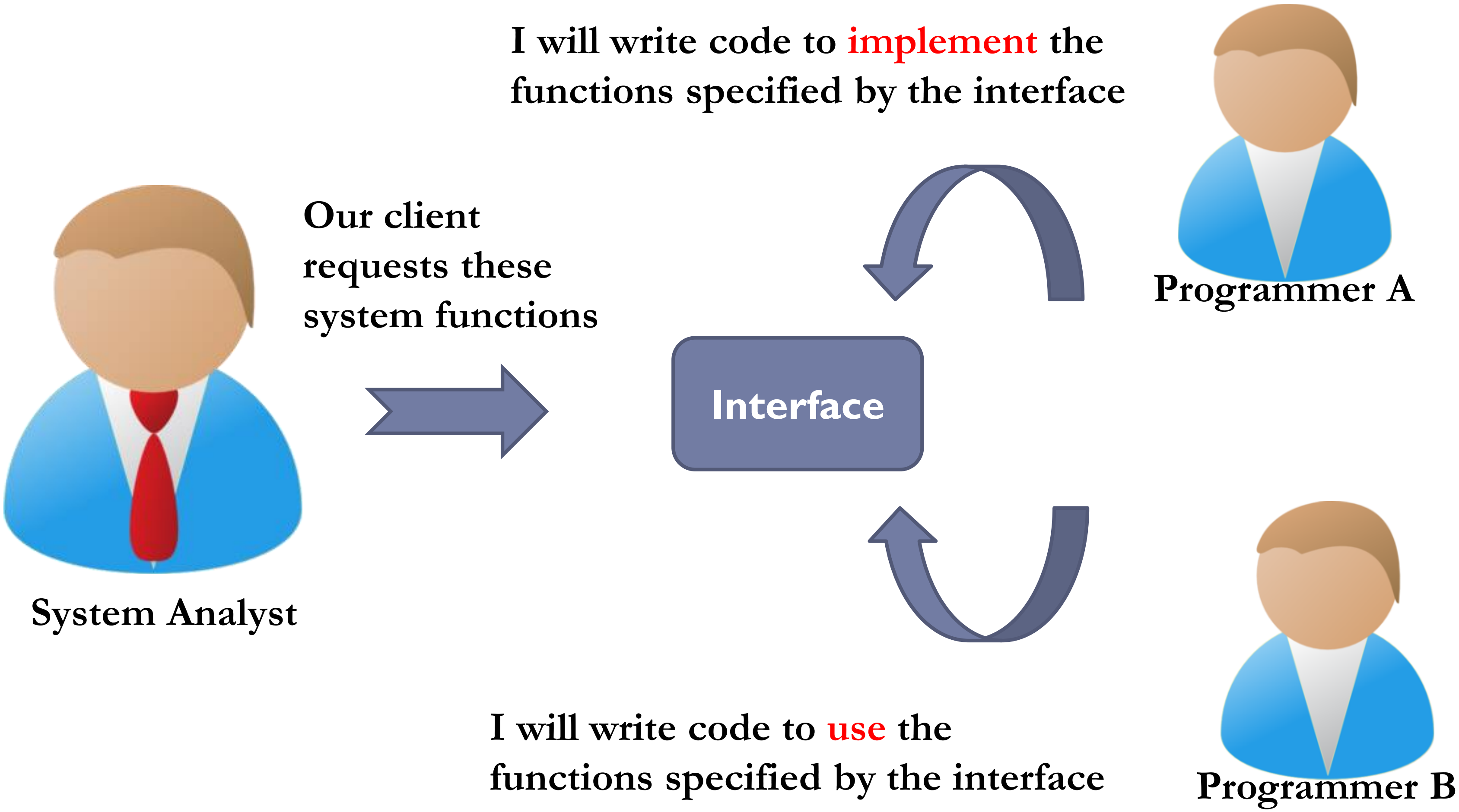
Interface

- We need an interface –
  - a specification that tells how the two bodies (screw and screwdriver) interact with each other
- In programming, when one component uses another component, an interface is used to tell what functions the latter should provide (dependency).
- In Java, an interface is a class-like construct that programmatically describes a set of methods that the conforming classes must implement.
- In reality, it is more or less like writing a contract to tell how the two parties work together.

Component A
e.g. client code calling functions from a library

Component B
e.g. a library providing functions to a client

Design by Interface
(in contrast to implementation)



Interface Syntax

- An interface is similar to a class but contains only constants and abstract methods. When you want to declare a method or a class where the *implementation is unknown* at the moment, then you can use the abstract modifier.
- Java uses the following syntax to declare an interface:

```
public interface InterfaceName {
  constant declarations;
  method signatures;
}
```

```
public interface Stack {
  public static final int MAX_SIZE = 100;
  public abstract int pop();
  public abstract void push(int e);
}
```

Interface is a Special Class

- An interface is treated like a special class in Java. Each interface is compiled into a separate bytecode file, just like a regular class.

- Because interfaces contain abstract methods which do not define method bodies, you cannot create instances from an interface using the **new** operator as usual, but you can use an interface to declare a variable.
  - Stack s1; // no instance, only reference var
  - Stack[ ] s2 = new Stack[10];

Properties of Interface

- Interface has
  - all data fields being constant (public static final)
  - all methods being abstract (public abstract)
- Therefore, these modifiers can be omitted.

```java
public interface T1 {
  public static final int K = 1;

  public abstract void p();
}
```

Equivalent

```java
public interface T1 {
  int K = 1;

  void p();
}
```

- A constant defined in an interface can be accessed like this:
  - InterfaceName.CONSTANT_NAME
  - e.g. Stack.MAX_SIZE

Replacing Dummy Implementation

It is always a good practice to replace those undefined or unclear implementations with an abstract method so as to avoid ambiguity.

Redundant implementation

```
class Fruit {
    public void color() {
        System.out.println("**undefined**");
    }
}
```

Replace

```
public interface Colorable {
    abstract public void color();
}
```

Using Interface

- You can provide the method bodies (method implementations) for the abstract methods in another class.

```
public interface Colorable {
    abstract public void color();
}
```

```
class Apple implements Colorable {
    @Override
    public void color() {
        System.out.println(“Red");
    }
}

class Banana implements Colorable {
    @Override
    public void color() {
        System.out.println(“Yellow");
    }
}
```

```java
public class FruitStore {

    public static Fruit[] getSomeFruits() {
        Fruit[] someFruits = {new Fruit(), new Apple(), new Banana()};
        return someFruits;
    }
}

class Fruit {
    public void color() {
        System.out.println("**undefined**");
    }
}

class Apple extends Fruit {

    @Override
    public void color() {
        System.out.println("Red");
    }

}

class Banana extends Fruit {
    @Override
    public void color() {
        System.out.println("Yellow");
    }
}
```

```java
public class InterfaceDemo {

    public static void main(String[] args) {
        Colorable[] fruits = {new Apple(), new Banana()};
        findEdibleFruitColor(fruits);
    }

    public static void findEdibleFruitColor(Colorable[] fruits) {
        for(int i=0; i<fruits.length; i++)
            if(fruits[i] instanceof Edible)
                fruits[i].color();
    }
}

class Apple implements Colorable, Edible {

    @Override
    public void color() {
        System.out.println("Red");
    }

    @Override
    public void eat() {
        System.out.println("I'm juicy");
    }

}

class Banana implements Colorable {

    @Override
    public void color() {
        System.out.println("Yellow");
    }
}
```

The abstract Modifier

- The modifier abstract can also be applied to a class – such class is called **abstract class**
  - A mix of ordinary class and interface
  - Usually contains both abstract and concrete (non-abstract) method
  - Usually be extended and the subclass overrides the abstract methods with a concrete method
  - Cannot be instantiated: you cannot create an object of an abstract class.

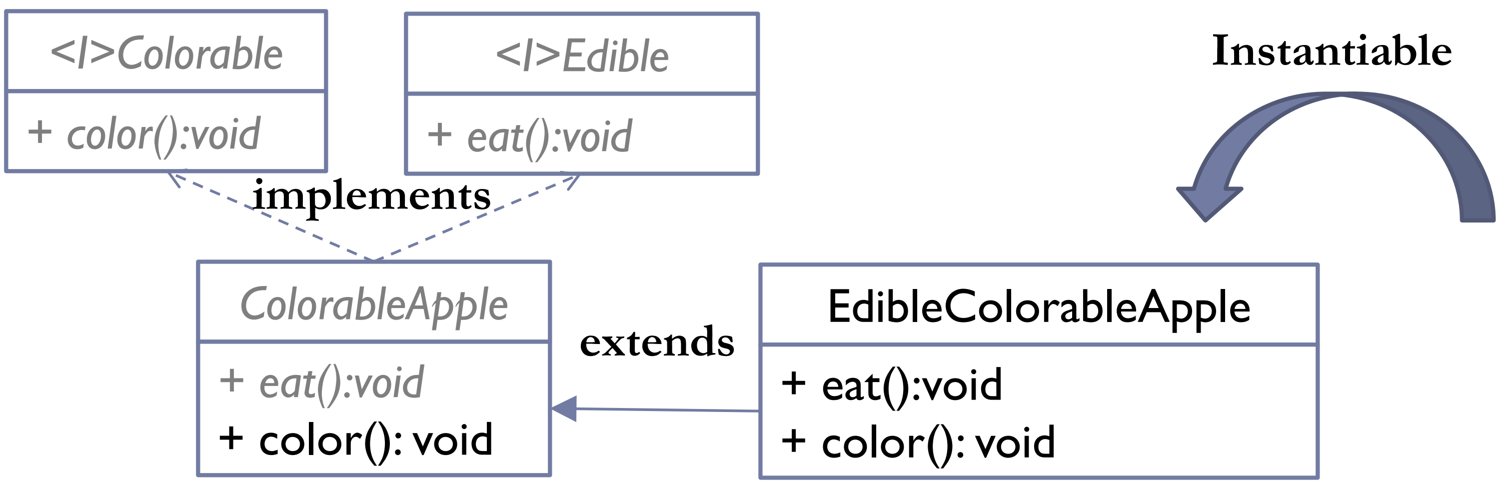
```
public abstract class Shape {
  protected char drawingChar = '*';
  public abstract void draw();
  public char getDrawingChar() {
        return drawingChar;
  }
}
```

Abstract Class

- It is fine to declare an abstract class that contains no abstract methods. But a class that contains abstract methods must be an abstract class (but not necessarily be all abstract methods).
  - If a subclass of an abstract superclass does not implement/override all the abstract methods, the subclass hence inherits abstract methods and must be declared abstract.
  - Similar to interface, an abstract class cannot be instantiated using the new operator, but can be used to declare reference variable.

Why Abstract Type?

- Abstract types are useful in that they can be used to define and enforce a protocol; a set of operations which all objects that implement the protocol must support.
- Abstract class is generally used as a base class for defining a new subclass. Its abstract methods force its subclasses to provide an implementation.
- The fact that Java language disallows instantiation of abstract types and force subtypes to implement all needed functionality further ensures program correctness.



Multiple Interfaces

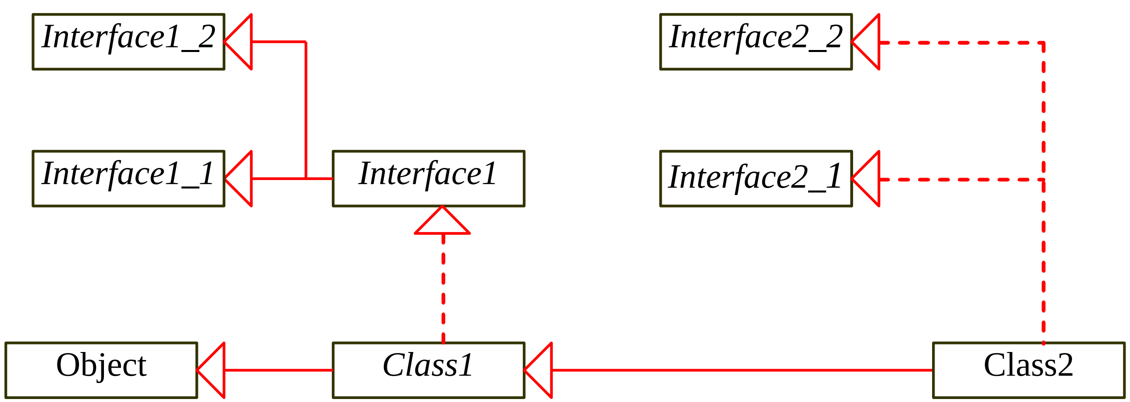
```java
interface InterfaceA {
    void methodA();
}

interface InterfaceB {
    void methodB();
}

class ClassC implements InterfaceA, InterfaceB {
    public void methodA() {
        // Implementation of methodA
    }

    public void methodB() {
        // Implementation of methodB
    }
}
```

Abstract Class

- **Overriding Methods**: A subclass can provide its own implementation of methods from the abstract class. This is known as method overriding. If a method is declared as abstract in the abstract class, the subclass is required to provide an implementation for it.
- **Accessing Variables**: A subclass can access and modify the variables of an abstract class if they are declared as protected or public. However, it's generally recommended to declare variables as private and provide public or protected getter and setter methods to access and modify them.
- **Calling Superclass Methods**: A subclass can also call the original method from the abstract class using the super keyword.

Multiple Inheritances

- All classes share a single root, the Object class, but there is no single root for interfaces.
- Like a class, an interface also defines a type. A variable of an interface type can reference any instance of the class that implements the interface. If a class implements an interface, this interface plays the same role as a superclass. You can use an interface as a data type and cast a variable of an interface type to its subclass, and vice versa.
- Suppose that c is an instance of Class2. c is also an instance of Object, Class1, Interface1, Interface1_1, Interface1_2, Interface2_1, and Interface2_2.



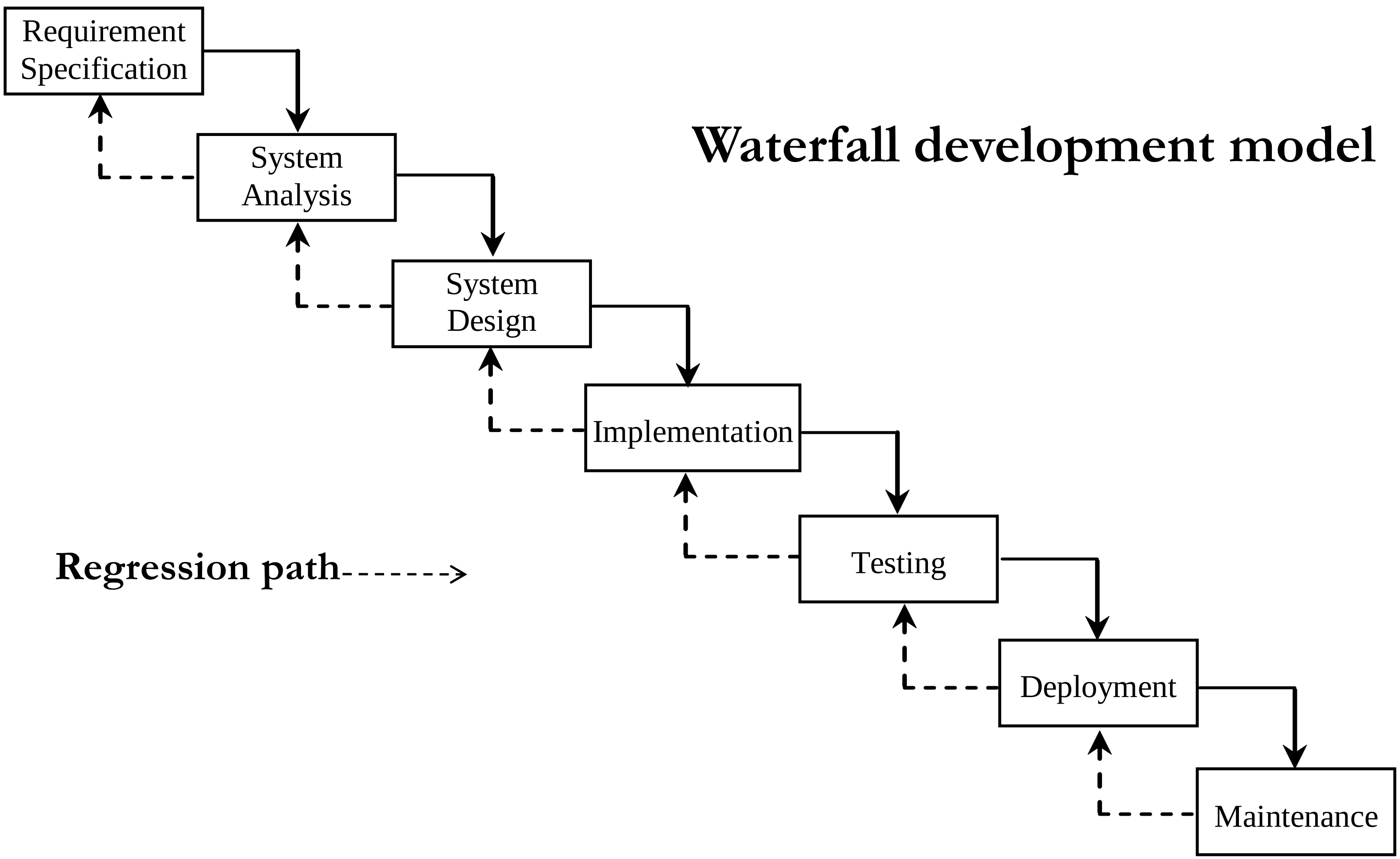
Interfaces vs. Abstract Classes

- In Java, both abstract classes and interfaces are used to create a contract for other classes to follow.
- In an interface, the data must be constants, but an abstract class can have all types of data.
- All methods of an interface are abstract, but an abstract class can have concrete methods.

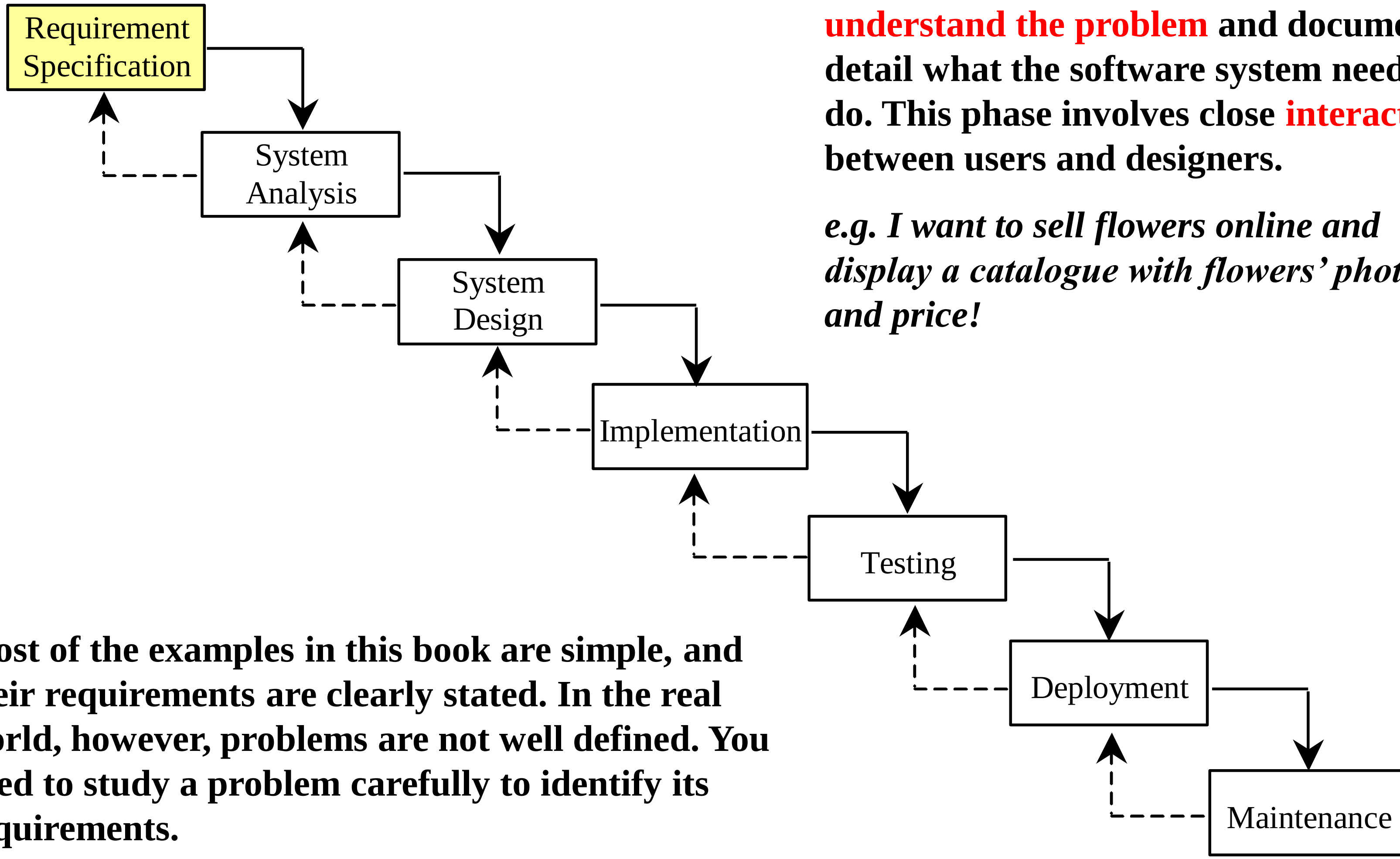
	Variables	**Methods**	**Constructors**	**Inheritance**
Abstract class	No restrictions	No restrictions.	Constructors are invoked by subclasses through constructor chaining. An abstract class cannot be instantiated using the new operator.	Single inheritance; extend one class only
Interface	All variables must be public static final	All methods must be public abstract instance methods	No constructors. An interface cannot be instantiated using the new operator.	Multiple inheritance; implement more than one interface

Object-Oriented Design Principles

Software Development Process



Requirement Specification

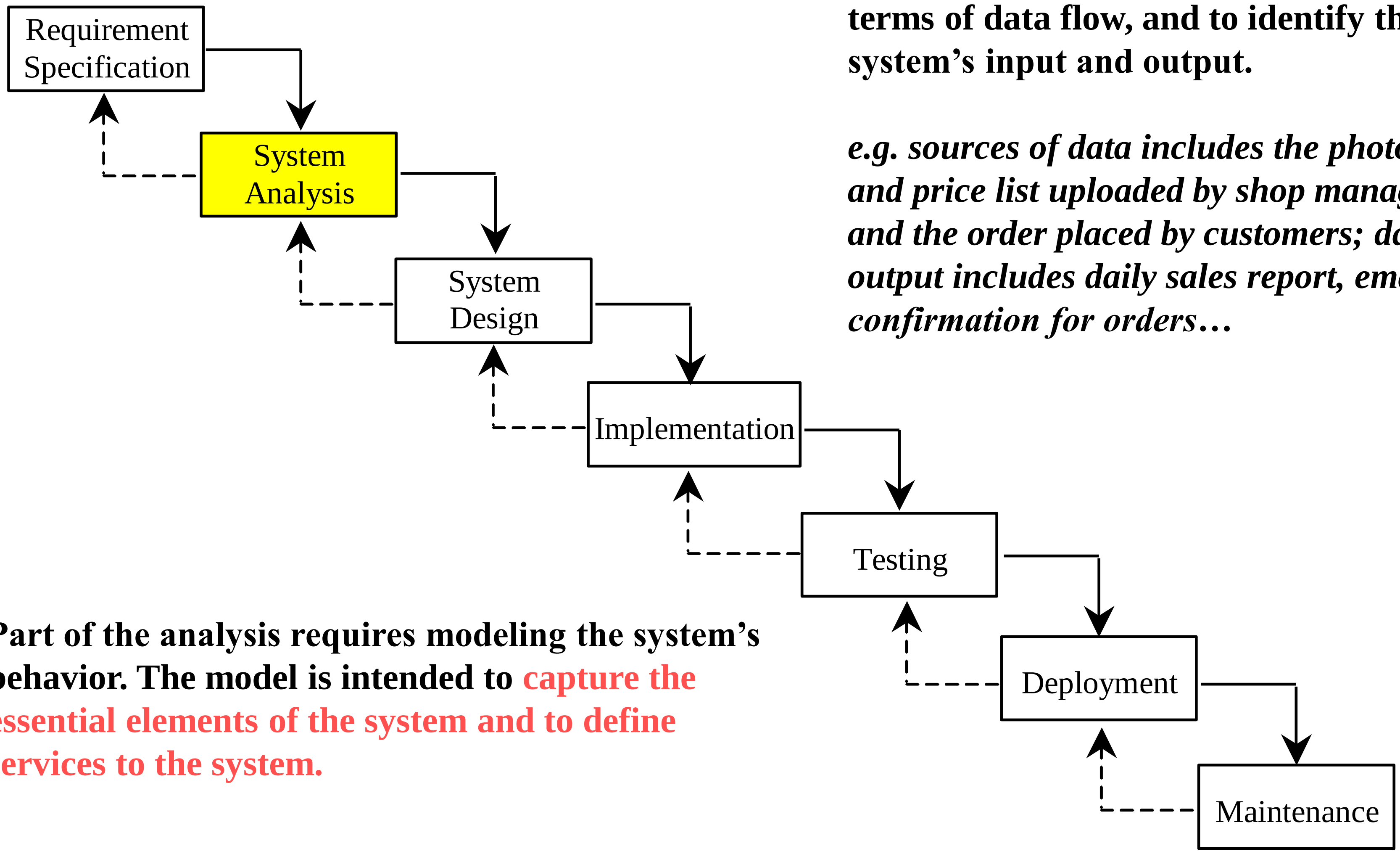


A formal process that seeks to understand the problem and document in detail what the software system needs to do. This phase involves close interaction between users and designers.

e.g. I want to sell flowers online and display a catalogue with flowers' photo and price!

Most of the examples in this book are simple, and their requirements are clearly stated. In the real world, however, problems are not well defined. You need to study a problem carefully to identify its requirements.

System Analysis

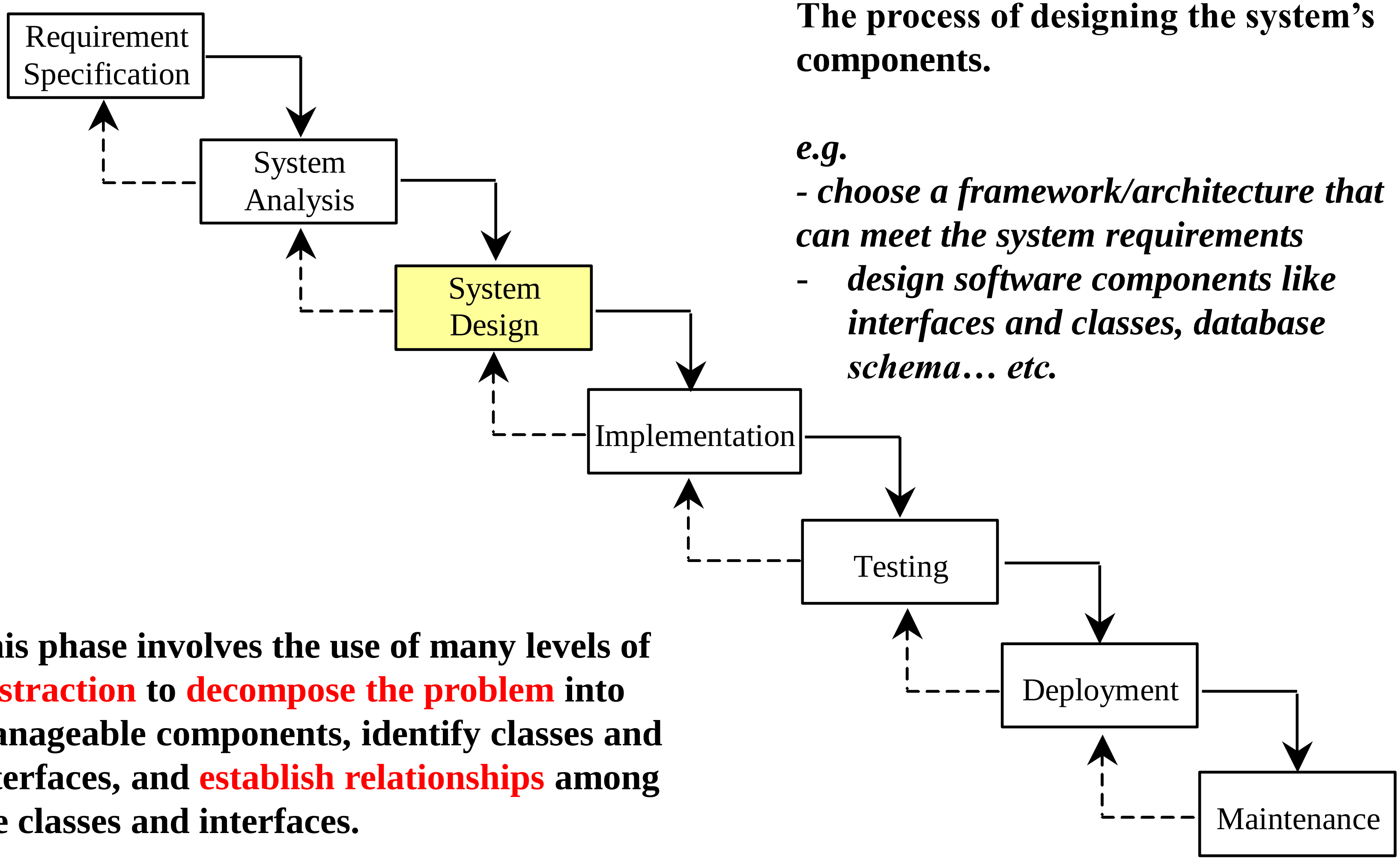


Seeks to analyze the business process in terms of data flow, and to identify the system's input and output.

e.g. sources of data includes the photos and price list uploaded by shop manager and the order placed by customers; data output includes daily sales report, email confirmation for orders…

Part of the analysis requires modeling the system's behavior. The model is intended to capture the essential elements of the system and to define services to the system.

System Design



The process of designing the system's components.

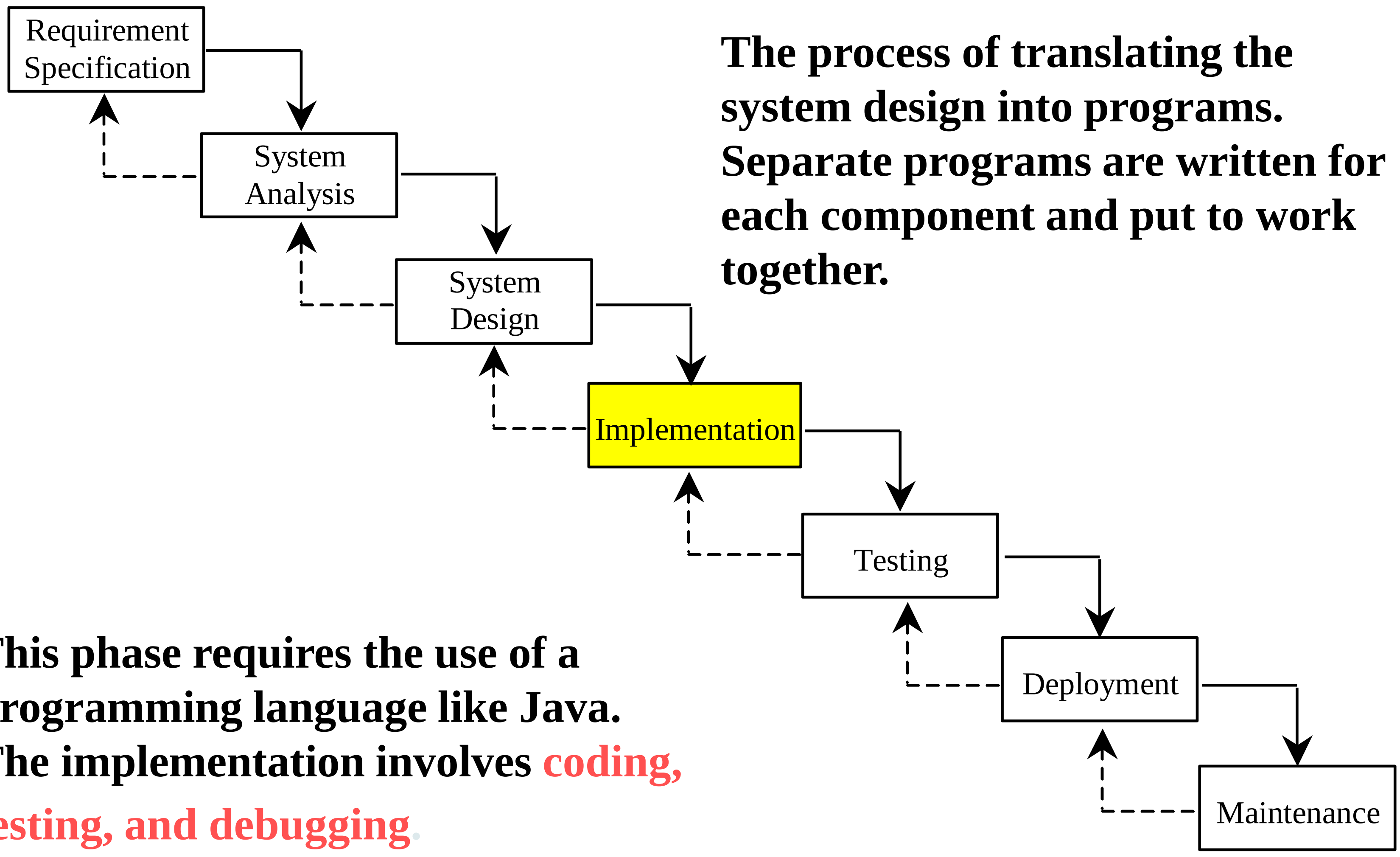
e.g.

- choose a framework/architecture that can meet the system requirements

- design software components like interfaces and classes, database schema… etc.

This phase involves the use of many levels of abstraction to decompose the problem into manageable components, identify classes and interfaces, and establish relationships among the classes and interfaces.

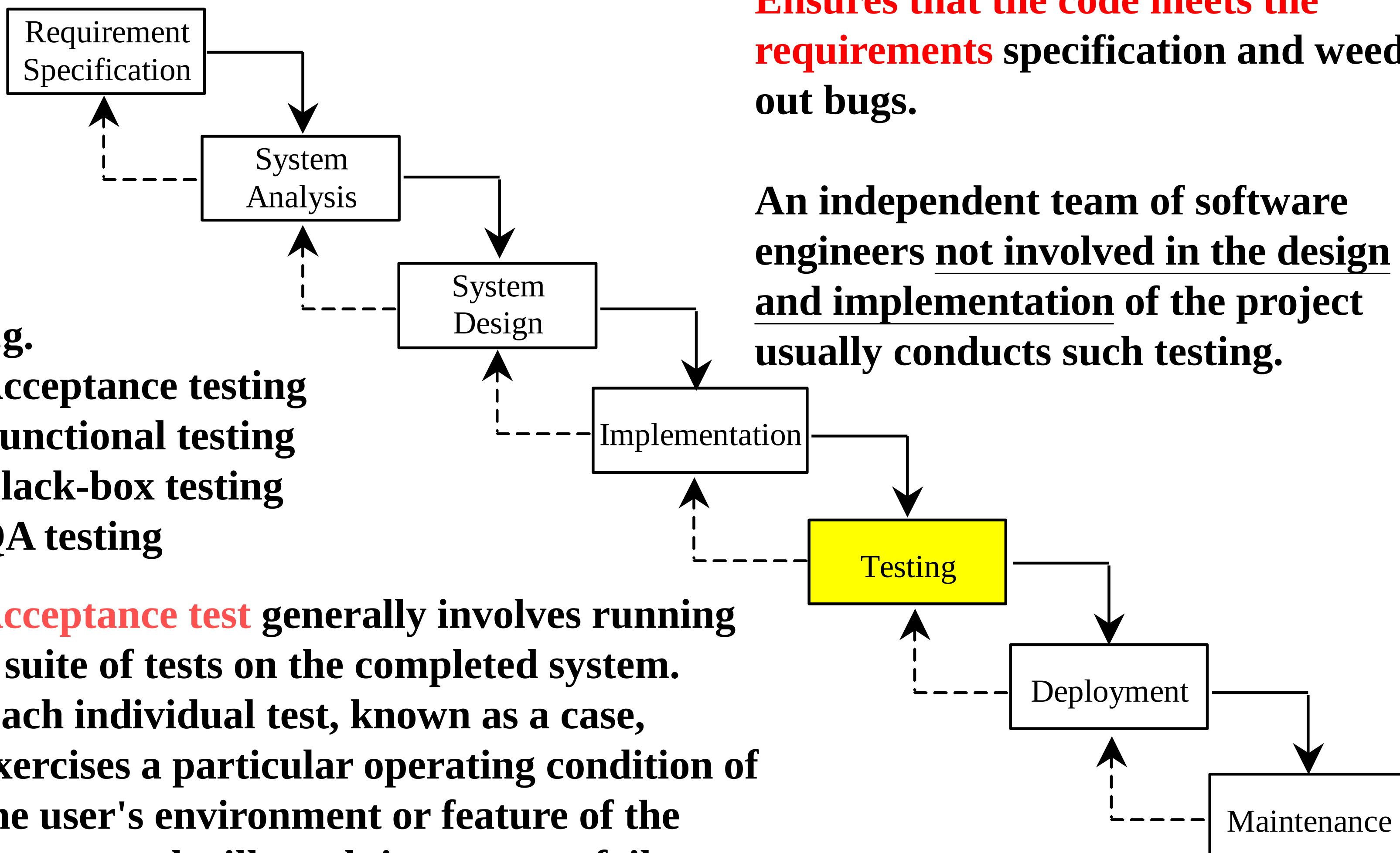
Implementation



The process of translating the system design into programs. Separate programs are written for each component and put to work together.

This phase requires the use of a programming language like Java. The implementation involves coding, testing, and debugging.

(System) Testing



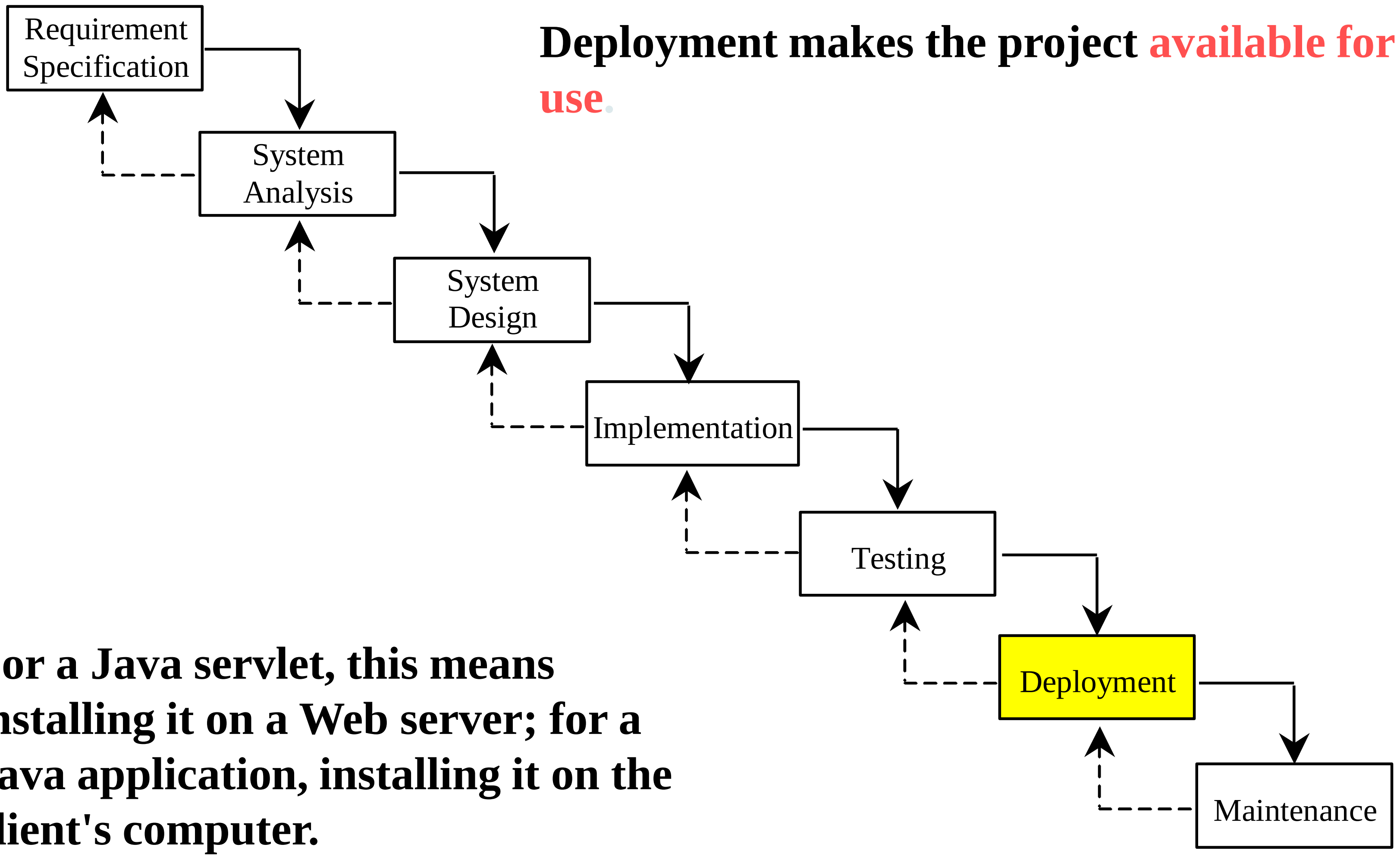
Ensures that the code meets the requirements specification and weeds out bugs.

An independent team of software engineers not involved in the design and implementation of the project usually conducts such testing.

e.g.
Acceptance testing
Functional testing
Black-box testing
QA testing

Acceptance test generally involves running a suite of tests on the completed system. Each individual test, known as a case, exercises a particular operating condition of the user's environment or feature of the system, and will result in a pass or fail expected outcome.

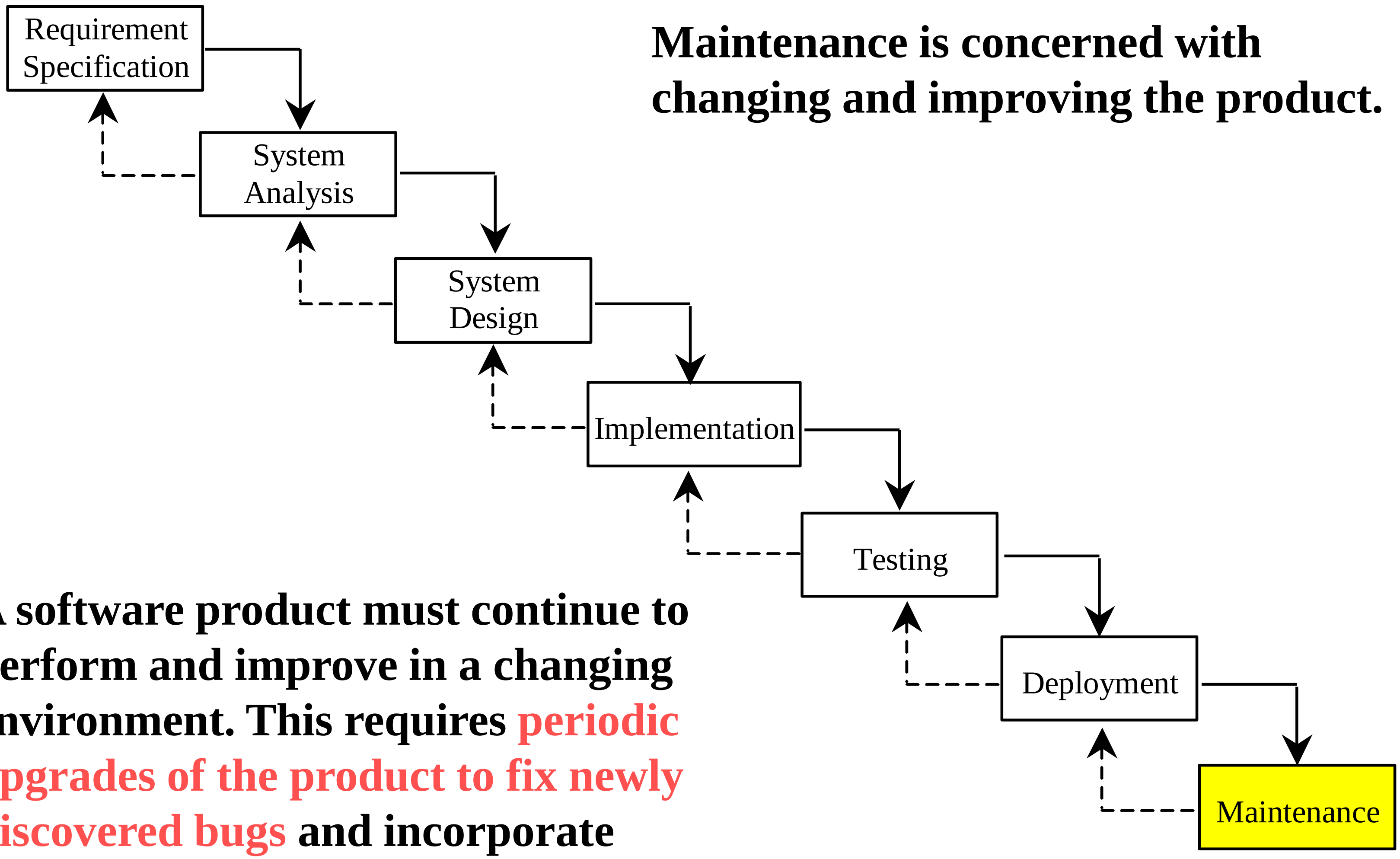
Deployment



Deployment makes the project available for use.

For a Java servlet, this means installing it on a Web server; for a Java application, installing it on the client's computer.

Maintenance

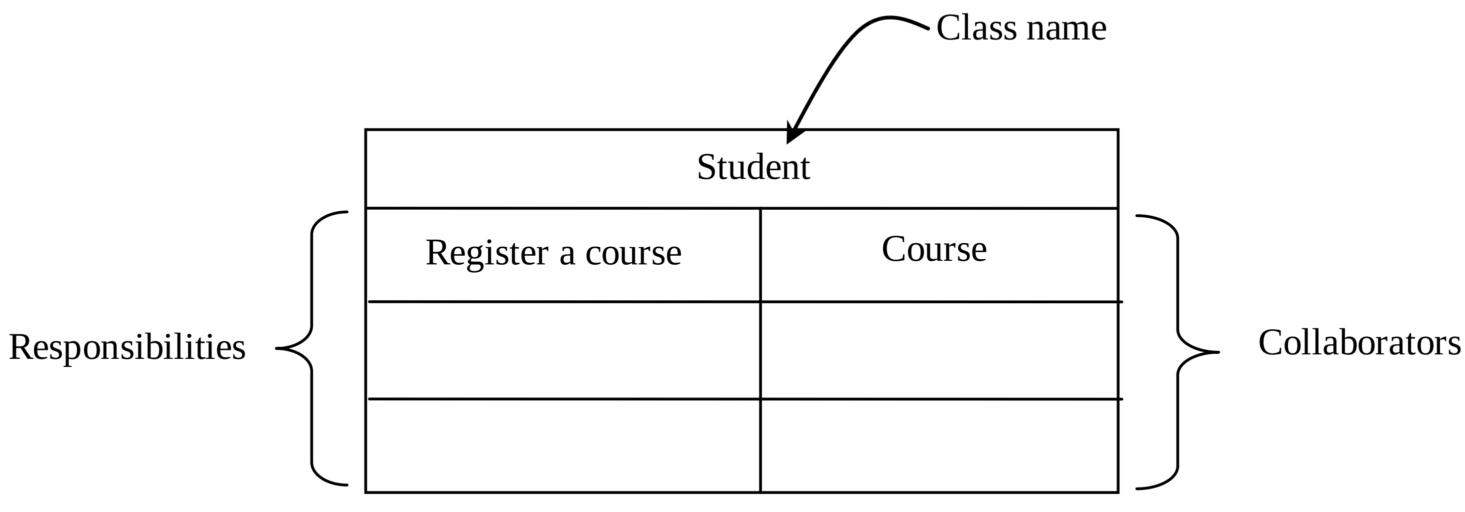


Maintenance is concerned with changing and improving the product.

A software product must continue to perform and improve in a changing environment. This requires periodic upgrades of the product to fix newly discovered bugs and incorporate changes.

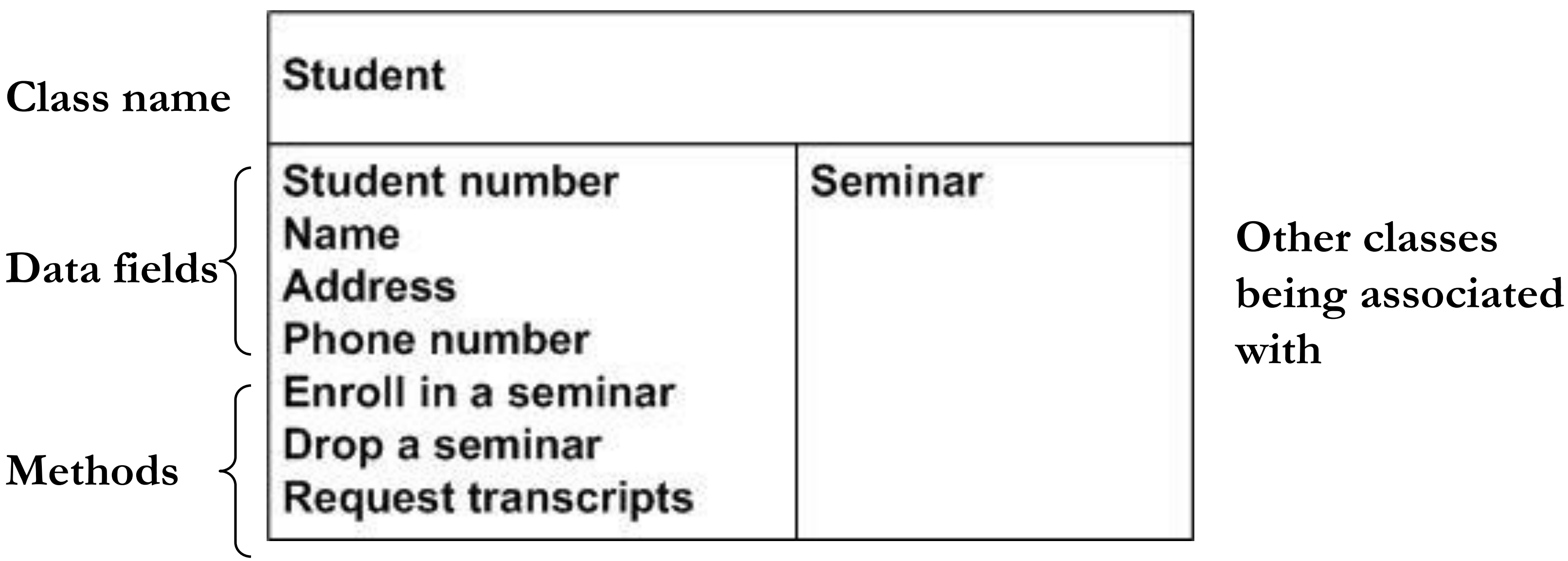
Discovering Classes Using CRC Card

- Class Responsibility Collaborator (CRC) cards are a popular brainstorming tool used in discovering classes. It uses an index card for each class as shown below.
- A class represents a collection of similar objects.
- A responsibility is something that a class knows or does.
- A collaborator is another class that a class interacts with to fulfill its responsibilities.



Using CRC to Capture a Student

- For example, students have names, addresses, and phone numbers. These are the things a student knows. Students also enroll in seminars, drop seminars, and request transcripts. These are the things a student does.
- Sometimes a class has a responsibility to fulfill, but not have enough information to do it. For example, to accomplish enrollment of seminars, a student needs to know if a seat is available in the seminar and, if so, he then requests to be added to the seminar.



Coupling

- Association, aggregation, composition, realization and generalization all describe the coupling between two classes.
- The difference is the degree of coupling.
- In general, lower degree of coupling implies higher stability of the system that a change in one module will not require a change in the implementation of another module.
- Low coupling is often a desirable system property.

`coupling increases`

$\longrightarrow$

association, aggregation, composition, realization, generalization

Association

- Association specifies objects of one class are connected to objects of another and there is a channel between them through which messages can be sent. .
  - Ability to send message to each other.
  - The state of the object changes when its associated object changes.
  - The association relationship is usually implemented using data fields. There is a strong connection between two classes.


```
public class Student {
  /** Data fields */
  private Course[] courseList;

  /** Constructors */
  /** Methods */
}
```

```
public class Course {
  /** Data fields */
  private Student[] classList;
  private Faculty faculty;

  /** Constructors */
  /** Methods */
}
```

```
public class Faculty {
  /** Data fields */
  private Course[] courseList;

  /** Constructors */
  /** Methods */
}
```

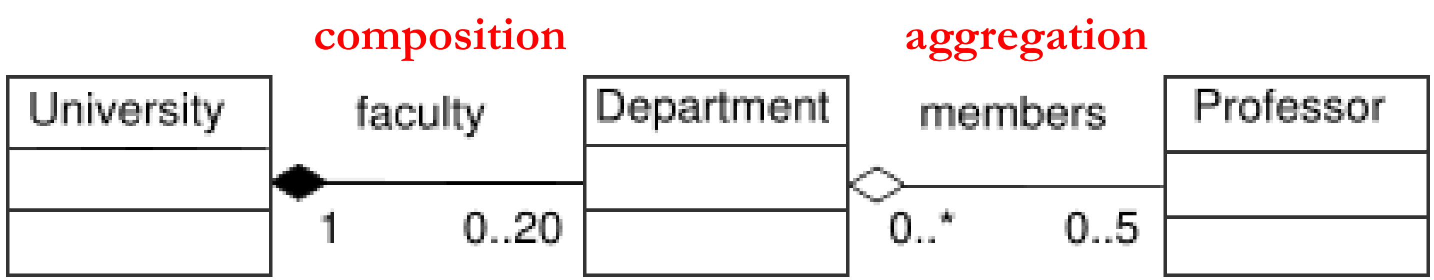
Association Properties

- The properties are optional in UML diagram
- Name
  - Describe the nature of the relationship
  - May give a direction to the name
- Role
  - Specify the role it plays in the relationship
- Multiplicity
  - State how many instances may be connected across the relationship

Digit	The exact number of elements
*	Zero to many
0..1	Zero or one
1..*	One to many
3..5	Three to five
0, 2..5, 9..*	Zero, two to five, and nine to many

Aggregation and Composition

- Aggregation and Composition represent a whole-part relationship (a.k.a. has-a relationship) between two classes.
  - The 'whole' contains the 'part', while the 'part' cannot has the 'whole'.
  - Composition is a stronger form of aggregation. It adds lifetime responsibility to aggregation that the part must be created and destroyed together with the whole.
  - An aggregation or composition relationship is usually represented as a data field in the 'owner' class. (same as association but different semantic)
- For example:
  - If the university closes, the departments will no longer exist, but the professors in those departments will continue to exist. Therefore, a University can be seen as a composition of departments, whereas departments have an aggregation of professors.



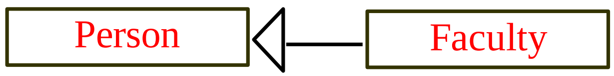
Inner Classes

- If Department is used in the University class only, it is usually declared as an inner class of University.
- There are three types of inner class:
  - Member inner class - declared within another class
  - Local inner class - declared within the body of a method
  - Anonymous inner class - declared within the body of a method without name
- The following example shows member inner class.

```
public class University {
  private Department[] depts;
  ...

  private class Department {
  ...
  }
}
```

Generalization

- Generalization models the inheritance relationship (is-a relationship) between two classes.
  - generalized class (superclass)
  - specialized class (subclass)



(A)

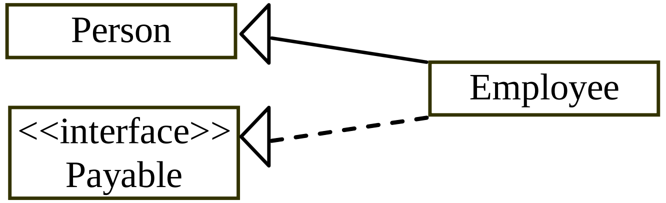
```
public class Faculty extends Person {
  /** Data fields */
  /** Constructors */
  /** Methods */
}
```

(B)

Realization

- Realization represents the is-a-kind-of relationship, which describes a class provides an implementation of a contract specified by an interface class.

```
public class Employee extends Person implements Payable {
  /** Data fields, Constructors, and */
  /** Methods */

  /** Implement the interface method */
  public void pay (int amount) {
    // ...
  }
}
```



Class Design in 4 Steps

- 1. Identify classes for the system.
  - Ordinary classes, abstract classes, interfaces
- 2. Describe attributes and methods in each class.
  - Using Modifiers public, protected, private and static
- 3. Establish relationships among classes.
  - Association, generalization, realization, … etc.
- 4. Create classes.

Class Design Principles

- Single Responsibility Principle (SRP)
  - A class should describe a single entity or a set of similar operations.
  - A single entity with too many responsibilities can be broken into several classes to separate responsibilities.
- Don't Repeat Yourself (DRY)
  - Classes should be designed for use and reuse in many different situations. One carefully designed piece of work could be useful in a wide range of applications and increases your productivity.
  - You should design a class that imposes no or minimal restrictions on what or when the user can do with it. That means users can incorporate classes in many different combinations, orders, and environments.
  - The class should provide a variety of ways for customization through properties and methods. This can also increase the chance of adoption of the class.

Using Visibility Modifiers

- Each class can present two contracts – one for the users of the class and one for the extenders of the class.
- Make the fields private and accessor/mutator methods public if they are intended for the users of the class.
  - A class should use the private modifier to hide its data from direct access by clients. You can use get methods and set methods to provide users with access to the private data, but only to private data you want the user to see or to modify.
  - A class should also hide methods not intended for client use.
- Make the fields or method protected if they are intended for extenders of the class.

Using Inheritance or Aggregation

- Sometimes, the choice between inheritance and aggregation is not obvious. For example, the relationship between the classes Circle and Cylinder can apparently be modeled with inheritance. But one could argue that a cylinder consists of circles; thus, you might use aggregation to define the Cylinder class as follows:

Aggregation

```
public class Cylinder {
  private Circle[] circles;

  /** Constructors */

  /** Methods */
}
```

Inheritance

```
public class Cylinder
   extends Circle {

  /** Constructors */

  /** Methods */
}
```

Using Inheritance or Aggregation

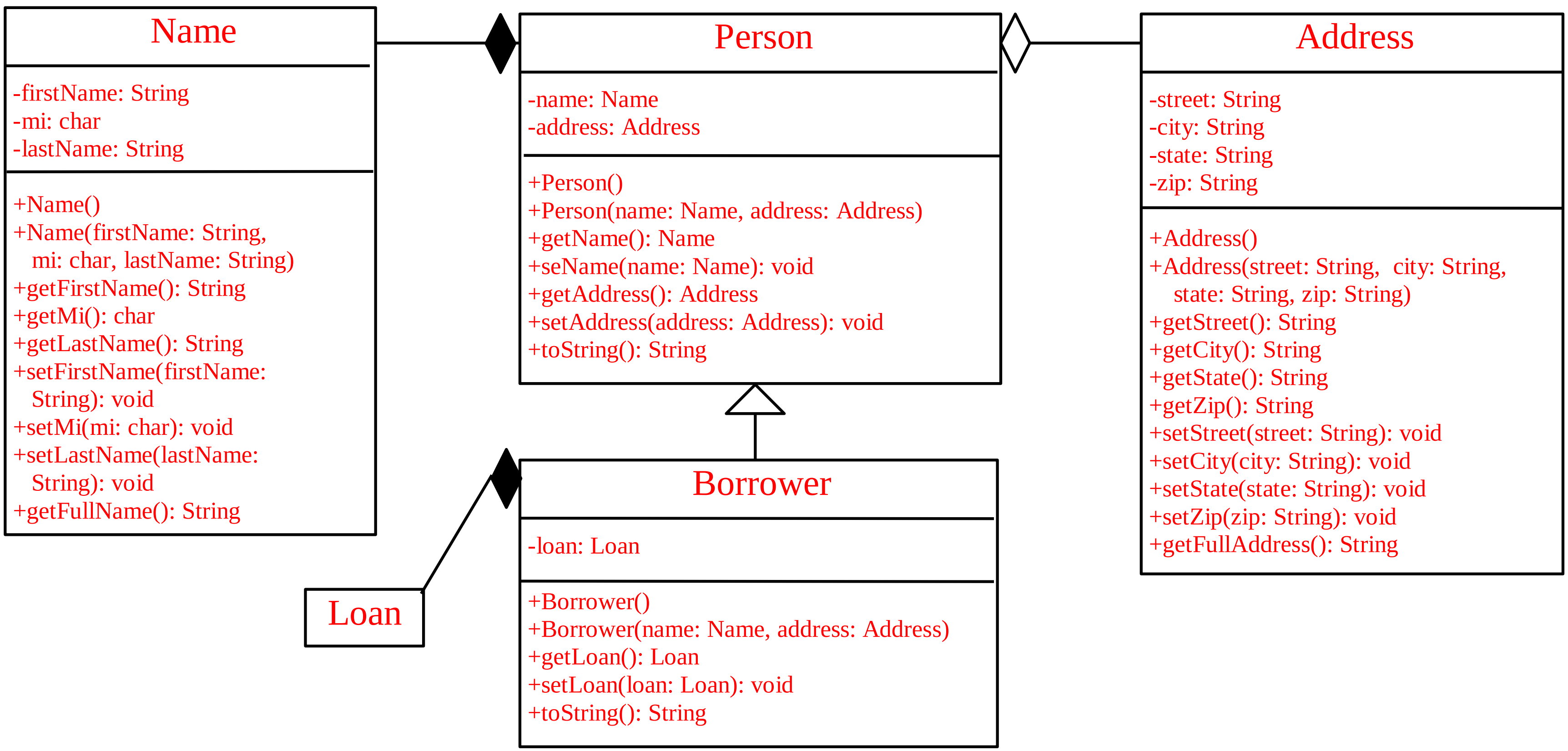
- Both designs are fine, but which one is preferred?

- If polymorphism is desirable, you should use the inheritance design. That is you may want to write:
  - Circle[] circles = {new Cylinder(), new Circle()};

- If you don't care about polymorphism, the aggregation design gives more flexibility because the classes are less dependent on the other when using aggregation than using inheritance.

Class Design Example - Borrowing Loans

- The following is a test program that uses the classes Name, Person, Address, Borrower, and Loan.



Applying OOD Principles in your Miniproject

- Feel free to choose topics you are interested in.
- Some possible topics
  - **Library Management System**: Create a system to manage books in a library, including features like book checkouts, returns, and inventory management.
  - **Student Information System**: Develop a system to manage student data, including grades, courses, and personal information.
  - **Restaurant Reservation System**: Create a system for making restaurant reservations, including table selection and time slot booking.

Miniproject

- Submission
  - Zip file: source code (make sure your code can be run without bugs), report (introduction, implementation details with CRC cards and UML, some demos/input,output)
  - Deadline: Friday 1159pm@Week 14

In-class practice

- Lab04: ex 1 and 2