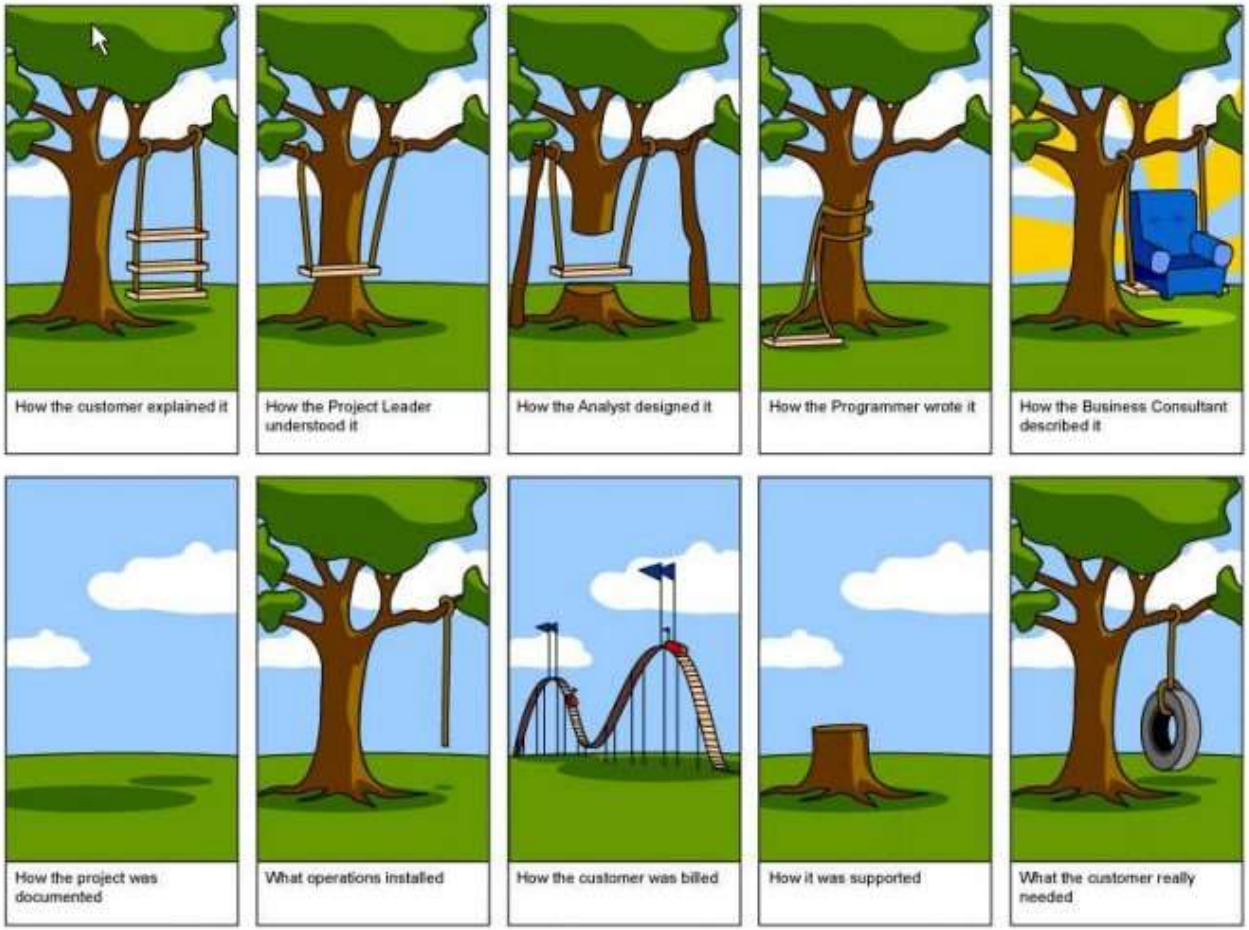
SOFTWARE DEVELOPMENT

Software Development

- Traditional software development involves several phases.
  Three major steps are:
- 1. Design
- 2. Coding
- 3. Testing and Debugging



Design

- It is in the design step that we decide how to divide the workings of our program into classes, when we decide how these classes will interact, what data each will store, and what actions each will perform.
- There are some rules of thumb that we can apply when determining how to define our classes:

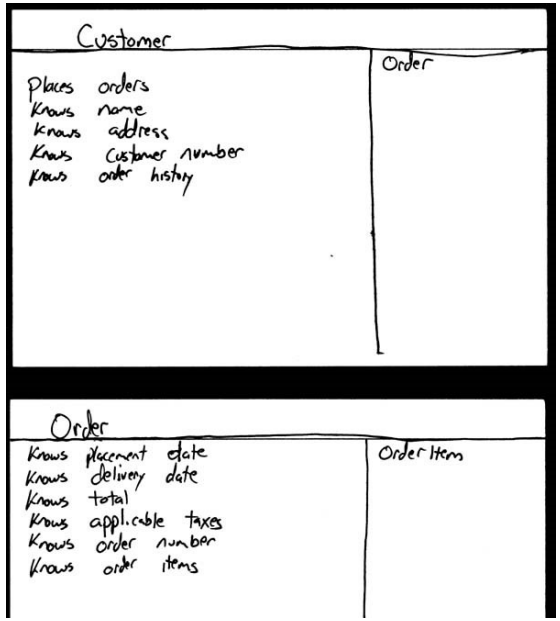
Design

- ***Responsibilities:***
  - *Divide the work into different actors, each with a different* responsibility. Try to describe responsibilities using action verbs. These actors will form the classes for the program.
- ***Independence:***
  - *Define the work for each class to be as independent from* other classes as possible. Subdivide responsibilities between classes so that each class has autonomy over some aspect of the program. Give data (as instance variables) to the class that has jurisdiction over the actions that require access to this data.
- ***Behaviors:***
  - *Define the behaviors for each class carefully and precisely, so* that the consequences of each action performed by a class will be well understood by other classes that interact with it. These behaviors will define the methods that this class performs, and the set of behaviors for a class form the *protocol by which other pieces of code will interact with objects from the* class.

Class-Responsibility-Collaborator (CRC) Cards

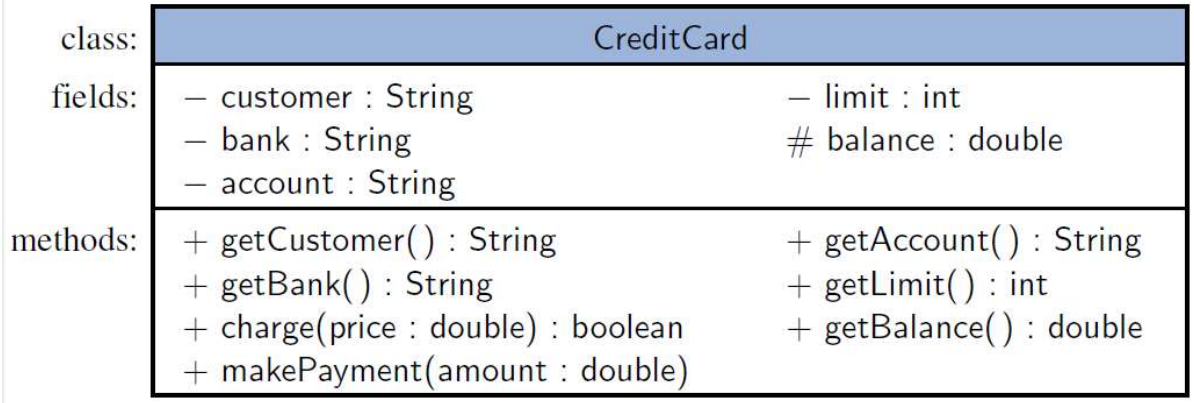
- A common tool for developing an initial high-level design for a project is the use of *CRC cards*.
  - Each card represent a component.

Class Name	
Responsibilities	Collaborators



UML Diagrams

- As the design takes form, a standard approach to explain and document the design is the use of UML (Unified Modeling Language) diagrams to express the organization of a program.
- One type of UML figure is known as a ***class diagram***.



A UML Class diagram for the CreditCard class

Pseudocode

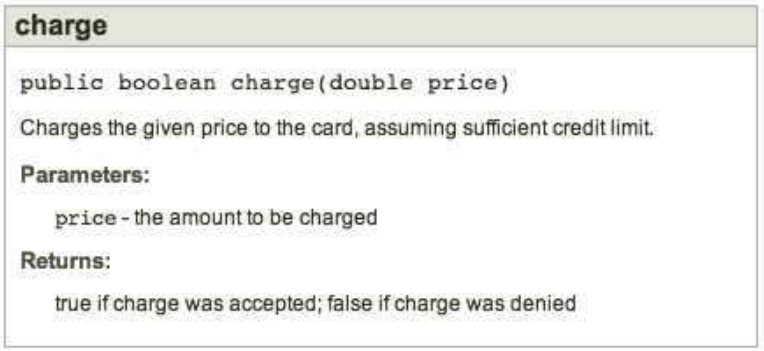
- As an intermediate step before the implementation of a design, programmers are often asked to describe algorithms in a way that is intended for human eyes only called ***pseudocode.***

Coding

- One of the key steps in implementing an object-oriented program is coding the descriptions of classes and their respective data and methods.
- In order to accelerate the development of this skill, we will discuss various ***design patterns for designing*** object-oriented programs.
- These patterns provide templates for defining classes and the interactions between these classes.

Documentation and Style

- Javadoc : In order to encourage good use of block comments and the automatic production of documentation, the Java programming environment comes with a documentation production program called ***javadoc. This program takes a collection of Java source*** files that have been commented using certain keywords, called ***tags, and it produces*** a series of HTML documents that describe the classes, methods, variables, and constants contained in these files.



Documentation rendered by javadoc for the CreditCard.charge method.

Javadoc

- Each javadoc comment is a block comment that starts with "/**" and ends with "*/", and each line between these two can begin with a single asterisk, "*", which is ignored.
- The block comment is assumed to start with a descriptive sentence, which is followed by special lines that begin with javadoc tags. A block comment that comes just before a class definition, instance variable declaration, or method definition is processed by javadoc into a comment about that class, variable, or method. The primary javadoc tags that we use are the following:
  - @author *text: Identifies each author (one per line) for a class.*
  - @throws *exceptionName description: Identifies an error condition that is* signaled by this method.
  - @param *parameterName description: Identifies a parameter accepted by this* method.
  - @return *description: Describes the return type and its range of values for a* method.

Javadoc

```
/**
* Constructs a new credit card instance.
* @param cust the name of the customer (e.g., ”John Bowman”)
* @param bk the name of the bank (e.g., ”California Savings”)
* @param acnt the account identifier (e.g., ”5391 0375 9387 5309”)
* @param lim the credit limit (measured in dollars)
* @param initialBal the initial balance (measured in dollars)
*/
public CreditCard(String cust, String bk, String acnt, int lim, double initialBal) {
      customer = cust;
      bank = bk;
      account = acnt;
      limit = lim;
      balance = initialBal;
}
```

Constructs a new credit card instance.
Parameters:

cust the name of the customer (e.g., ”John Bowman”)
bk the name of the bank (e.g., ”California Savings”)
acnt the account identifier (e.g., ”5391 0375 9387 5309”)
lim the credit limit (measured in dollars)
initialBal the initial balance (measured in dollars)

Readability and Programming Conventions

- Programs should be made easy to read and understand.
- Good programmers should therefore be mindful of **their coding style, and develop a style that communicates the important aspects of a program's design for both humans and computers.** Much has been written about good coding style, with some of the main principles being the following:
- Use meaningful names for identifiers.
  - The tradition in most Java circles is to **capitalize the first letter** of each word in an identifier, **except for the first word for a variable or method name**. By this convention, "Date," "Vector," "DeviceManager" would identify classes, and "isFull( )," "insertItem( )," "studentName," and "studentHeight" would respectively identify methods and variables
- Use named constants or enum types instead of literals. The tradition in Java is to **fully capitalize such constants.**

```
public static final int MIN_CREDITS = 12; // min credits per term
```

Readability and Programming Conventions

- Indent statement blocks.
  - Typically programmers indent each statement block by 4 spaces;
- Organize each class in the following order:
  - Constants
  - Instance variables
  - Constructors
  - Methods
- Use comments that add meaning to a program and explain ambiguous or confusing constructs. In-line comments are good for quick explanations and do not need to be sentences. Block comments are good for explaining the purpose of a method and complex code sections.

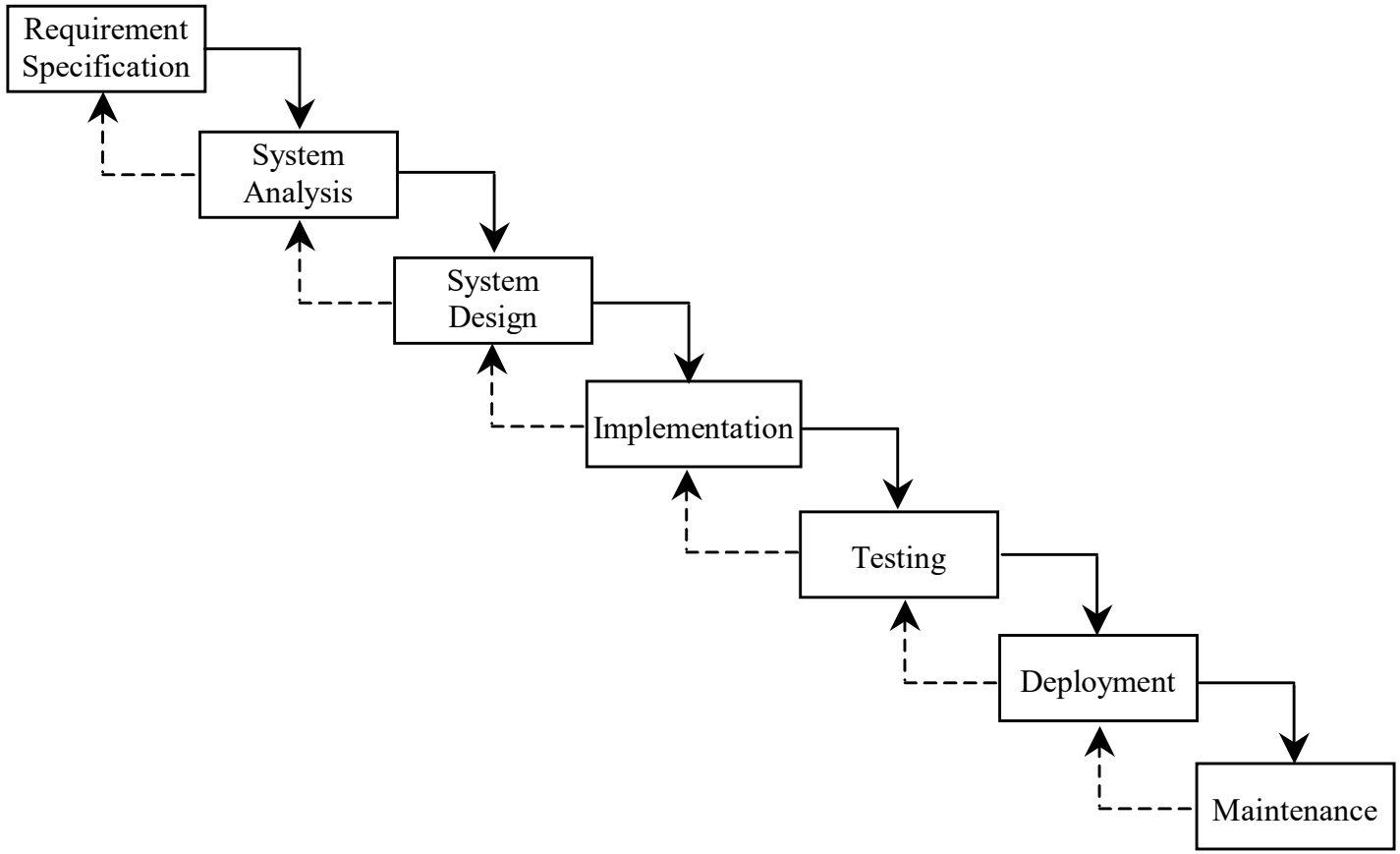
Testing

- A careful testing plan is an essential part of writing a program.
- Make sure that every method of a program is tested at least once (method coverage). Even better, each code statement in the program should be executed at least once (statement coverage).
- Programs often tend to fail on ***special cases of the input.***
  - The array has zero length (no elements).
  - The array has one element.
  - All the elements of the array are the same.
  - The array is already sorted.
  - The array is reverse sorted.

Debuging

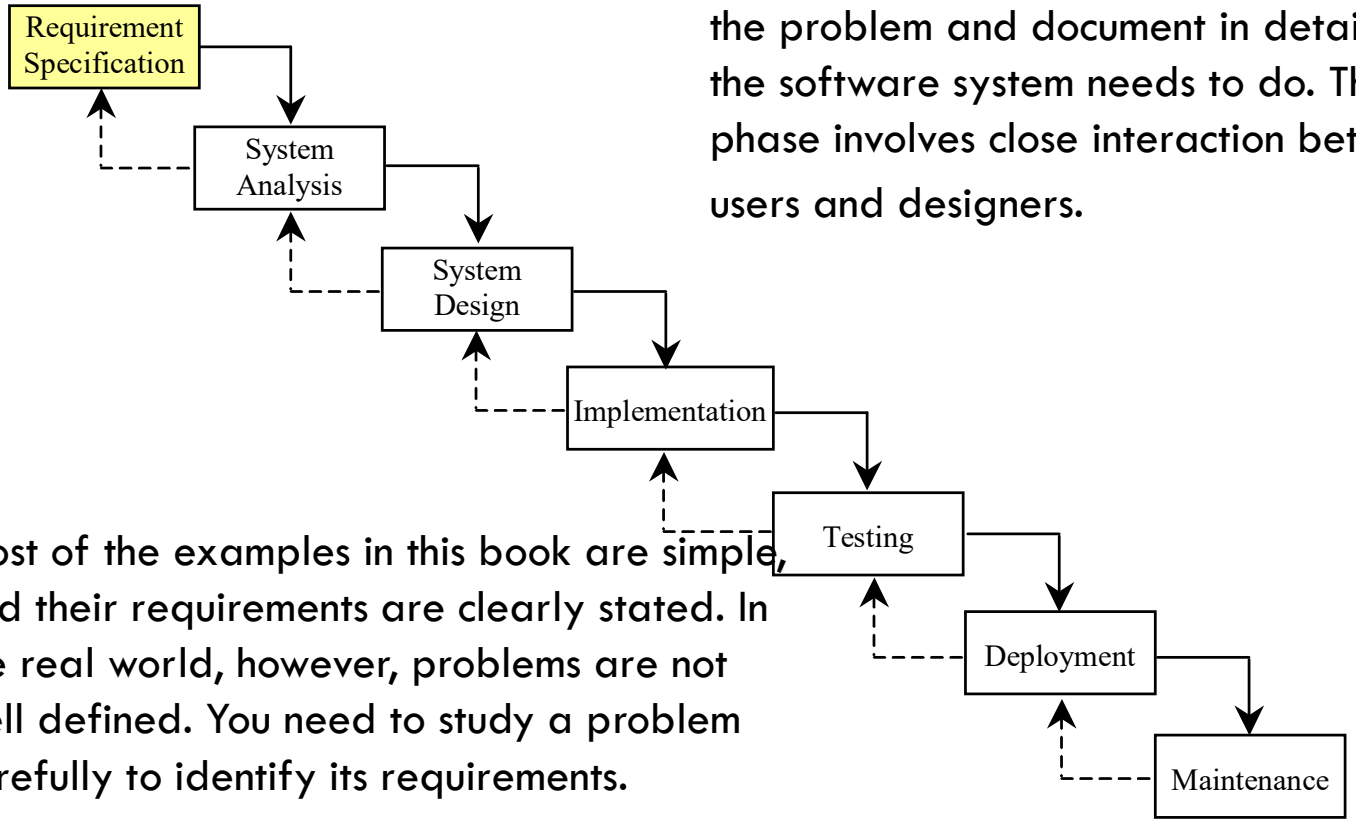
- The simplest debugging technique consists of using ***print statements to track the*** values of variables during the execution of the program. A problem with this approach is that eventually the print statements need to be removed or commented out, so they are not executed when the software is finally released.
- A ***debugger, which is a specialized*** environment for controlling and monitoring the execution of a program. The basic functionality provided by a debugger is the insertion of ***breakpoints within*** the code. When the program is executed within the debugger, it stops at each breakpoint. While the program is stopped, the current value of variables can be inspected.
- ***Conditional breakpoints, which are triggered only if a given expression is*** satisfied.
- The standard Java toolkit includes a basic debugger named jdb, which has a command-line interface. Most IDEs for Java programming provide advanced debugging environments with graphical user interfaces.

Software Development Process (Waterfall)



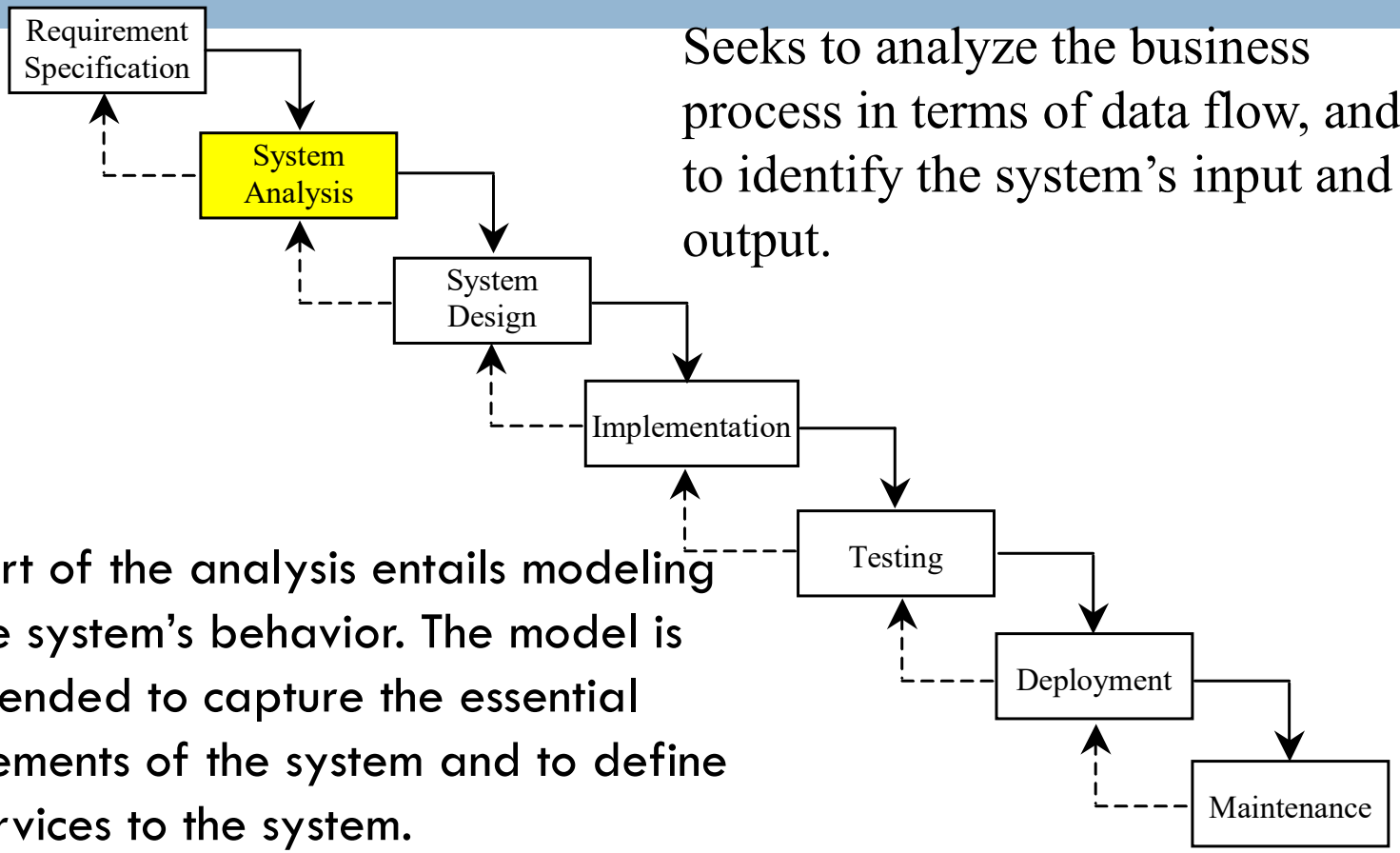
Requirement Specification

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.



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.

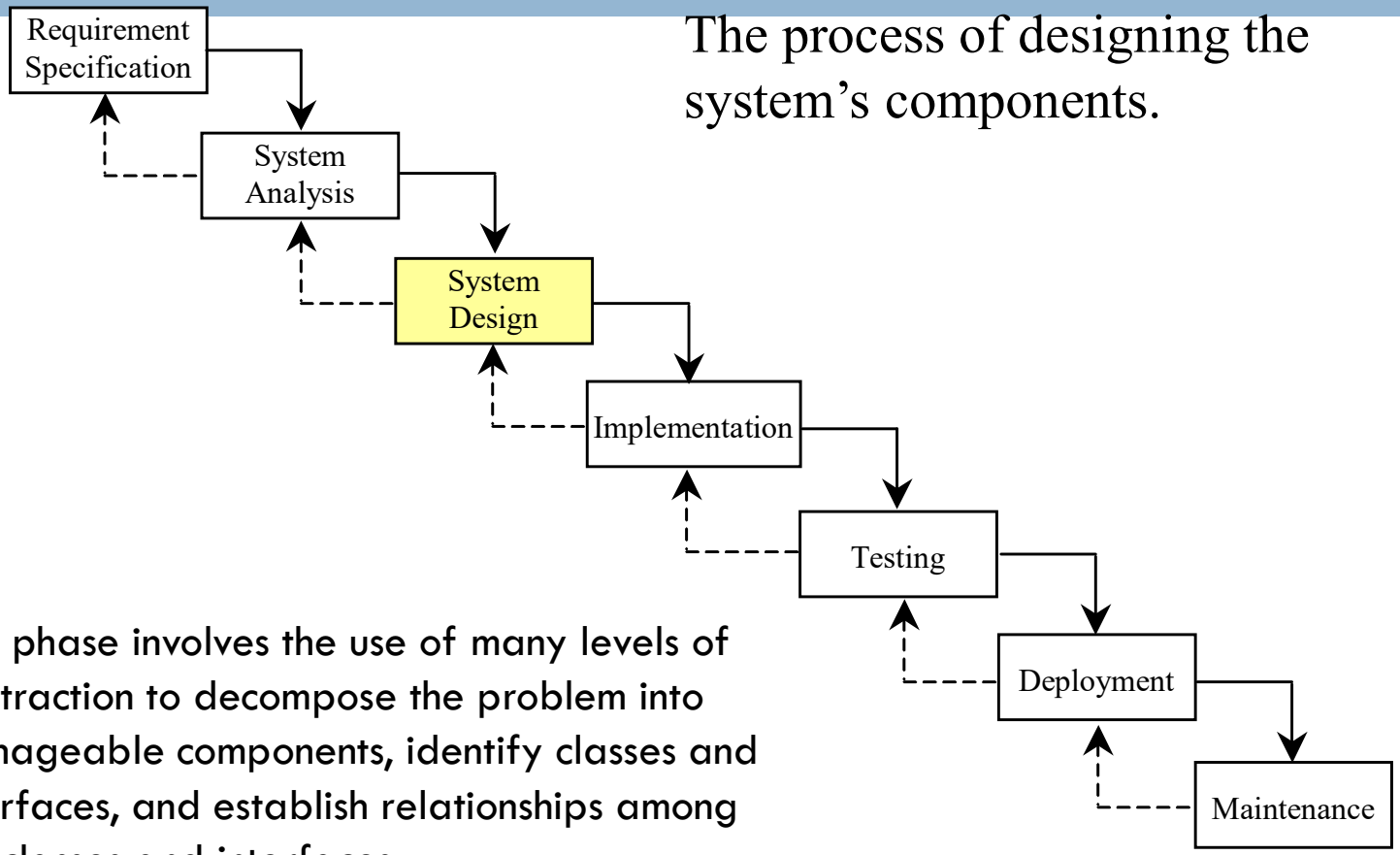
System Analysis



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

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

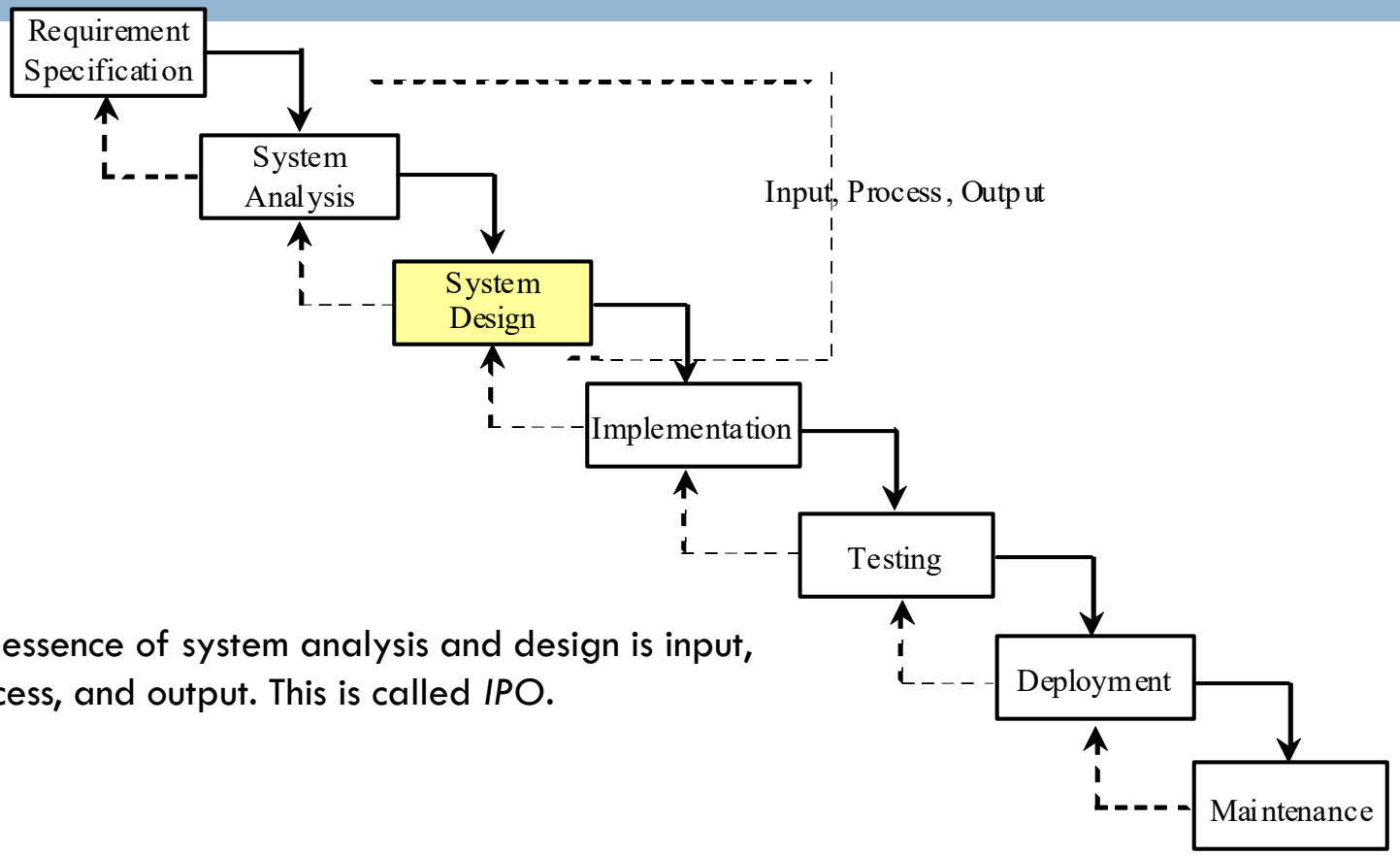
System Design



The process of designing the system's components.

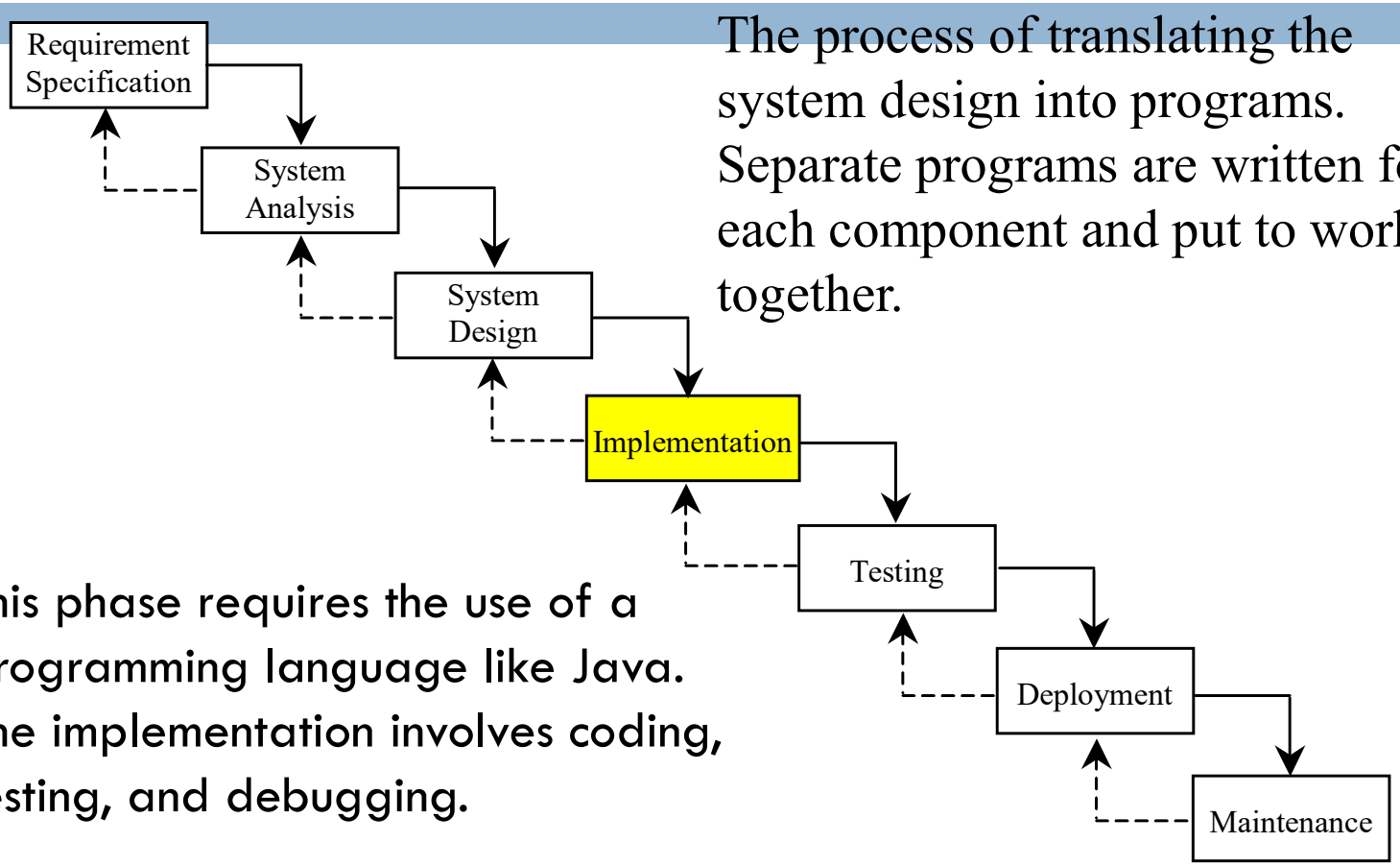
This phase involves the use of many levels of abstraction to decompose the problem into manageable components, identify classes and interfaces, and establish relationships among the classes and interfaces.

IPO



The essence of system analysis and design is input, process, and output. This is called *IPO*.

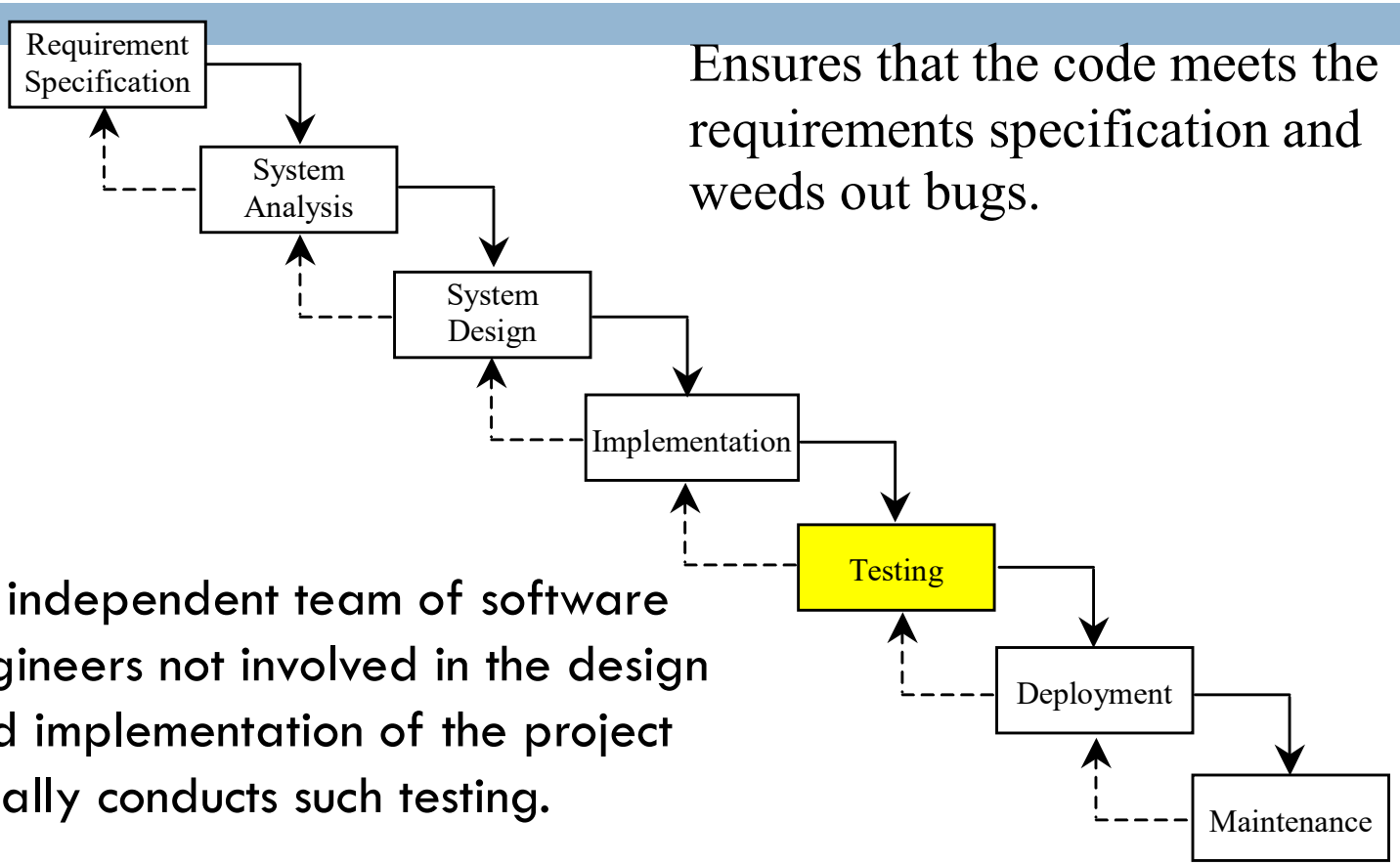
Implementation



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.

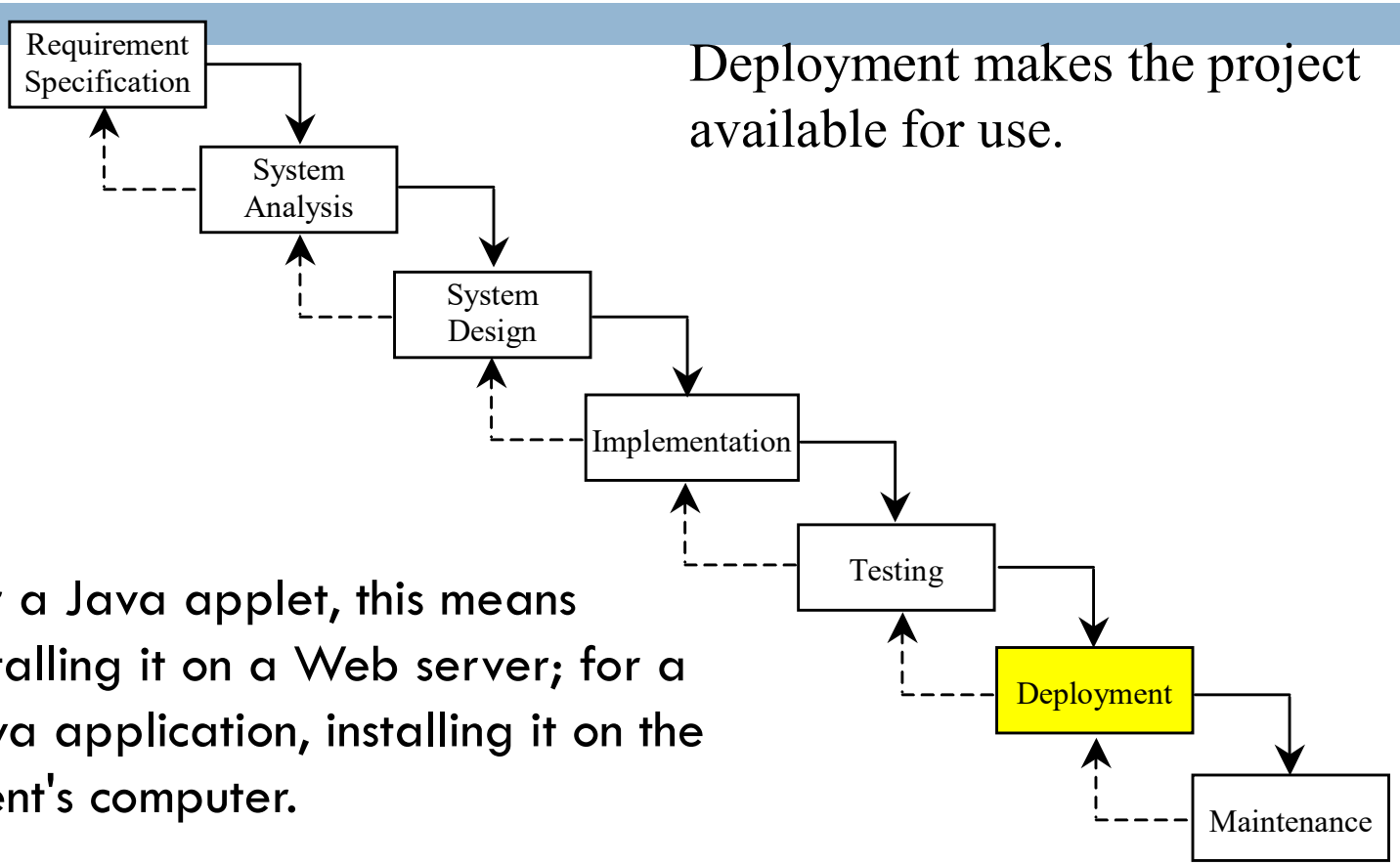
Testing



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.

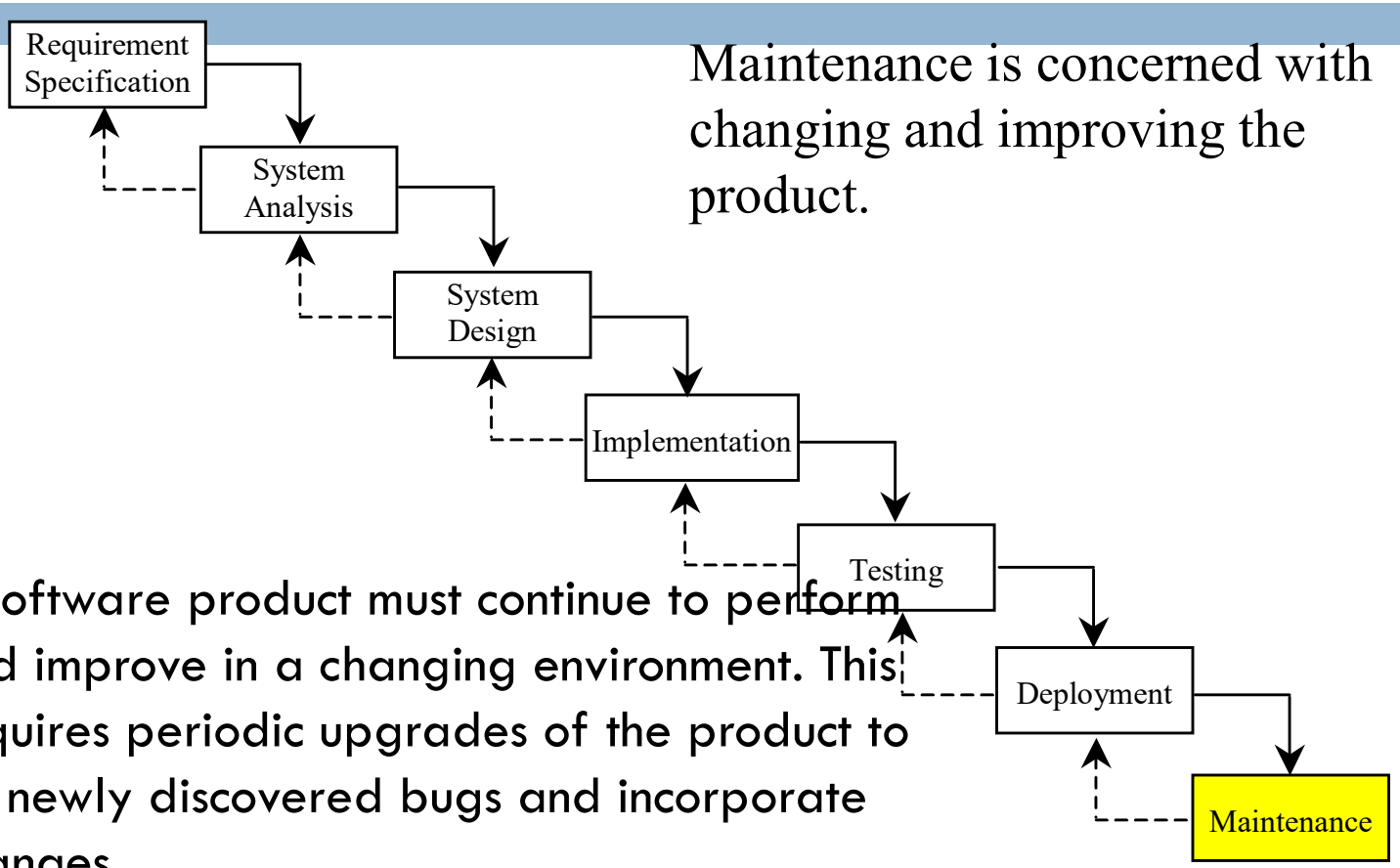
Deployment



Deployment makes the project available for use.

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

Maintenance



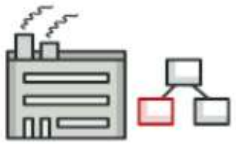
Maintenance is concerned with changing and improving the product.

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.

Patterns

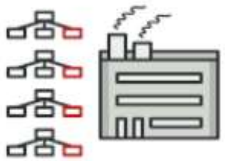
- Design patterns are typical solutions to common problems in software design. Each pattern is like a blueprint that you can customize to solve a particular design problem in your code.
- Patterns are a toolkit of solutions to common problems in software design. They define a common language that helps your team communicate more efficiently.
- They are typically categorized into 3 groups:
  - Creational Design Patterns
  - Structural Design Patterns

Creational Design Patterns



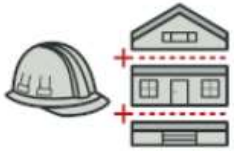
Factory Method

Provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.



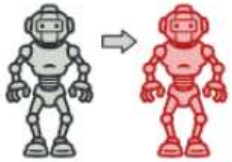
Abstract Factory

Lets you produce families of related objects without specifying their concrete classes.



Builder

Lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.



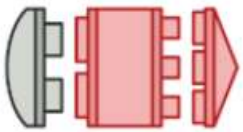
Prototype

Lets you copy existing objects without making your code dependent on their classes.

Singleton

Lets you ensure that a class has only one instance, while providing a global access point to this instance.

Structural Design Patterns

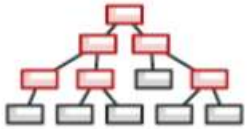


Adapter

Allows objects with incompatible interfaces to collaborate.

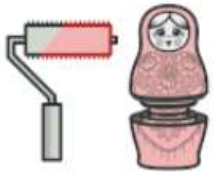
Bridge

Lets you split a large class or a set of closely related classes into two separate hierarchies—abstraction and implementation—which can be developed independently of each other.



Composite

Lets you compose objects into tree structures and then work with these structures as if they were individual objects.

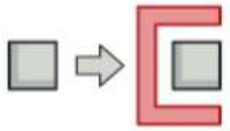


Decorator

Lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.

Facade

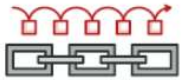
Provides a simplified interface to a library, a framework, or any other complex set of classes.



Proxy

Lets you provide a substitute or placeholder for another object. A proxy controls access to the original object, allowing you to perform something either before or after the request gets through to the original object.

Behavioral Design Patterns



Chain of Responsibility

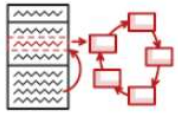
Lets you pass requests along a chain of handlers. Upon receiving a request, each handler decides either to process the request or to pass it to the next handler in the chain.

Command

Turns a request into a stand-alone object that contains all information about the request. This transformation lets you pass requests as a method arguments, delay or queue a request's execution, and support undoable operations.

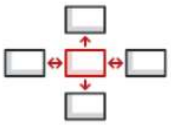
Iterator

Lets you traverse elements of a collection without exposing its underlying representation (list, stack, tree, etc.).



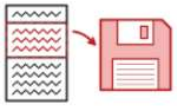
State

Lets an object alter its behavior when its internal state changes. It appears as if the object changed its class.



Mediator

Lets you reduce chaotic dependencies between objects. The pattern restricts direct communications between the objects and forces them to collaborate only via a mediator object.

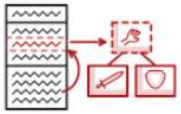


Memento

Lets you save and restore the previous state of an object without revealing the details of its implementation.

Observer

Lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing.



Strategy

Lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.

Example: Observer Pattern

- What Is the Observer Pattern?
  - Observer is a behavioral design pattern. It specifies communication between objects: **observable and observers. An observable is an object which notifies observers about the changes in its state.**
  - For example, a news agency can notify channels when it receives news. Receiving news is what changes the state of the news agency, and it causes the channels to be notified.

https://www.baeldung.com/java-observer-pattern

- Let's see how we can implement it ourselves.
- First, let's define the NewsAgency class:

```
public class NewsAgency {
       private String news;
       private List<Channel> channels = new ArrayList<>();
       public void addObserver(Channel channel) {
       this.channels.add(channel); }

       public void removeObserver(Channel channel) {
       this.channels.remove(channel); }

       public void setNews(String news) {
       this.news = news;
       for (Channel channel : this.channels)
         {
           channel.update(this.news);
         }
        }
     }
```

NewsAgency is an observable, and when news gets updated, the state of NewsAgency changes. When the change happens, NewsAgency notifies the observers about this fact by calling their update() method.

To be able to do that, the observable object needs to keep references to the observers, and in our case, it's the channels variable.

Let's now see how the observer, the Channel class, can look like. It should have the update() method which is invoked when the state of NewsAgency changes:

```
public class NewsChannel implements Channel {
        private String news;
        @Override
        public void update(Object news) {
        this.setNews((String) news);
        }
 }
```

The Channel interface has only one method:

```
public interface Channel {
        public void update(Object o);
}
```

Now, if we add multiple instances of NewsChannel to the list of observers, and change the state of NewsAgency, the instance of NewsChannel will be updated:

```
NewsAgency observable = new NewsAgency();
NewsChannel observer = new NewsChannel();
observable.addObserver(observer1);
observable.addObserver(observer2);
observable.addObserver(observer3);
observable.setNews("news");
```

https://www.baeldung.com/java-observer-pattern