3rd year Licence in computer science
Department of Computer Science
University of Biskra
Year 2024/2025
Credits: 5
Coefficient: 3

Software Engineering

Dr. Mohamed Lamine KERDOUDI

Email : **l.kerdoudi@univ-biskra.dz**

Software Engineering

Chapter 1: Introduction

1. Introduction
2. Definition and Objectives
3. Principles of Software Engineering
4. Expected Qualities of Software
5. Software Lifecycle
6. Software Lifecycle Models

Introduction

What is a computer program ?

- It is a **set of instructions** written in a programming language. It will be compiled or translated to the machine language to perform a specific task.

➔ **Is : Computer Program = Software ?**

What is a Software?

- It is not a computer program
- It is a set of computer programs **in addition to** the associated **documentation**, **configuration** files and **data.**

Examples : accounting software, loan management software, and hospital management system.

Writing a Program vs a Software

- For example, if you are asked to write a "Java program" that displays "He

Is this a good programming practice?

Hel

```
1  package se.course;
2
3  public class HelloWorld {
4
5      public static void main(String[] args) {
6
7          System.out.println("Hello World");
8      }
9
10 }
```

Is this a good programming practice?

- Putting **Business Logic** inside the main procedure
- No use of **Object-Oriented Design**
- Inclusion of **Text** inside the code
- No comments
- No Abstraction
- Not Modular
- …

```
3  public class HelloWorld {
4
5      public static void main(String[] args) {
6
7          System.out.println("Hello World");
8      }
9
10 }
```

Improved version of HelloWorld program

HelloWorld.java

```java
package se.course;

/***
 * This class allows to print a Hello World message by creating a class with a single print method
 * @author Mohamed Lamine Kerdoudi
 * @version 25-09-2023
 */
public class HelloWorld {


    public boolean print() {

        System.out.println("Hello World");
        return true;
    }

    /**
     * This is the main method
     * @param args are the command line arguments
     */
    public static void main(String[] args) {


        HelloWorld hello = new HelloWorld();
        hello.print();
    }

}
```

```java
import java.util.Scanner;
public class BadPracticeExample {
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    int result = 0;
    System.out.println("Enter first number: ");
    int a = sc.nextInt();
    System.out.println("Enter second number: ");
    int b = sc.nextInt();
    System.out.println("Enter the operation (+, -, *, /): ");
    String operation = sc.next();
    System.out.println("Result is: " + result);
  }
}
```

Is this a good programming practice?

- Poor Variable Naming : a and b are not descriptive
- Lack of Input Validation: invalid inputs or division by zero
- No Use of Functions
- Inefficient Conditional Logic: A switch-case is better
- …

```java
import java.util.Scanner;
public class ImprovedExample {
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    System.out.println("Enter first number: ");
    int firstNumber = sc.nextInt();
    System.out.println("Enter second number: ");
    int secondNumber = sc.nextInt();
    System.out.println("Enter the operation (+, -, *, /): ");
    String operation = sc.next();

    try {
        int result = performOperation(firstNumber, secondNumber,
    operation);
        System.out.println("Result is: " + result);

    } catch (IllegalArgumentException e) {
        System.out.println(e.getMessage());
    }
 }
```

```
    private static int performOperation(int a, int b, String operation) {

        switch (operation) {
            return a / b;

            default:
            throw new IllegalArgumentException("Error: Invalid operation");
        }
    }
}
```

Is this a good programming practice?

- No use of **Object-Oriented Design**
- No comments
- No Abstraction
- Not Modular
- …

Enhanced Example with OOP, Modularity, and Comments

```
import java.util.Scanner;
// Interface for Calculator Operations to ensure abstraction
and extensibility
interface Operation {
  int perform(int a, int b);
}
// Concrete classes for each operation, implementing the
Operation interface
class Addition implements Operation {
    @Override
    public int perform(int a, int b) {
     return a + b;
    }
}
class Subtraction implements Operation {
    @Override
    public int perform(int a, int b) {
     return a - b;
    }
}
```

```
class Multiplication implements Operation {
    @Override
    public int perform(int a, int b) {
      return a * b;
    }
}
class Division implements Operation {
    @Override
    public int perform(int a, int b) {
        if (b == 0) {
            throw new IllegalArgumentException("Error: Division
        by zero");
        }
        return a / b;
    }
}
```

```
// Calculator class that handles operations
class Calculator {

// Factory method to return the correct operation based on input
public Operation getOperation(String operator) {
    switch (operator) {
    case "+":
        return new Addition();
    case "-":
        return new Subtraction();
    case "*":
        return new Multiplication();
    case "/":
        return new Division();
    default:
        throw new IllegalArgumentException("Error: Invalid operation");
    }
}
// Method to perform the calculation
public int calculate(int a, int b, String operator) {
    Operation operation = getOperation(operator);
        return operation.perform(a, b);
    }
}
```

```java
public class CalculatorApp {
// Main method only handles user interaction and starts the program
 public static void main(String[] args) {
 Scanner scanner = new Scanner(System.in);
 // Getting user input
    System.out.println("Enter first number: ");
    int firstNumber = scanner.nextInt();
        System.out.println("Enter second number: ");
    int secondNumber = scanner.nextInt();
    System.out.println("Enter the operation (+, -, *, /): ");
    String operator = scanner.next();

 // Creating Calculator object to handle business logic
    Calculator calculator = new Calculator();

 // Performing the calculation and handling potential errors
    try {
        int result = calculator.calculate(firstNumber, secondNumber,
    operator);
        System.out.println("Result: " + result);
    } catch (IllegalArgumentException e) {
        System.out.println(e.getMessage());
    }
 }
}
```

Writing a Program vs Software

- **Is it easy to code and program ?**
  - How much you charge the customer for it ?
  - What happens if the customer would like to have some changes in the program?
  - Can you still maintain and work on the code after 10 years ?

Ad-hoc or Programming-centered approach

- You write a computer program in an unplanned way to solve a particular problem
- The main focus is made initially towards **starting** the **programming task,**
- It involves writing code or designing solutions **quickly** **without** considering **long-term maintainability, scalability, or standard practices.**
- **Lack of Abstraction or Generalization**: Solutions are typically not reusable or extendable.

- Characteristics of software in the modern era:
  - ***Scaling***
    - The complexity and size of problem are increasing dramatically
  - ***Change***
    - The **need (requirements)** of customers change fast

Developing a Software

- Software, depending on its **size** and **complexity**, can be developed by:
  - a single person,
  - a small team,
  - or a set of coordinated teams.
- Developing **large software** by large teams poses **significant problems**:

➔ **How to develop professional software?**

The famous "Chaos Report"

- In 1995, a study based on 8,380 software projects from 365 companies (conducted by the Standish Group) found the following results:
  - **Success**: **Only 16%** of software projects were completed on **time** and within **budget**.
  - **Issue (Challenged)**: **53%** completed but were late, over budget, or did not meet the initial specifications **(**reduced functionality offered),
  - **Failure**: **31%** were abandoned during development. Canceled before completion or were delivered but never used.

- **84% of projects failed! (Cancelled, Delivered Late, or run over their budget)**

❑ Main reasons for failures:

- Poor management and leadership
- Improper use of resources / Budget
- Lack of communication
- Lack of employee motivation
- Poor planning
- Incorrect budget estimation
- …

➔ Software development in a professional context must follow strict rules.

- In the NATO conference (1968) in Garmish, a new discipline was born:

Software Engineering

➔ It became clear that individual approaches to program development could not be applied to large and complex software systems

2. Definition and Objectives

"Software engineering is an **engineering discipline** that is concerned with **all aspects of software production** from the early stages of system specification through to maintaining the system after it has gone into use."

- ***Engineering discipline*** *:* Engineers make things work. They apply **theories**, **methods**, and **tools** where these are appropriate. However, they use them selectively and always try to discover solutions to problems. Solutions should **respect financial and deadline constraints.**
- ***All aspects of software production :*** technical and no technical such as : **software project management** and the **development of tools**, **methods**, and **theories**.

2. Definition and Objectives

❑ Objectives of Software Engineering (SE):

- SE was born in response to repeated failures of large software projects.
- The goal of SE is to develop software that meets the following criteria:
  - **meeting user needs**,
  - **staying within budget and time constraints**,
  - **maintaining quality according to the service contract**

Software process

- The systematic approach that is used in software engineering is sometimes called a **software process**.
  - A **software process** is a **sequence of activities** that leads to the production of a software product.

What are the fundamental activities in software engineering ?

- **To develop a Software for the “Hello World”**
  1. *Understand the problem*
  2. *Perform some design and planning*
  3. *Code and Develop the Software*
  4. *Validate and test the software*
  5. *Release, Install, Run or Deploy the software for the client*
  6. *Maintenance and Evolution*

- **To develop a Software for the “Hello World”**

1. ***Requirement Analysis*** *: Understand the problem*
2. ***Design :*** *Perform some design and planning*
3. ***Implementation :*** *Code and Develop the Software*
4. ***Testing :*** *Validate and test the software*
5. ***Integration :*** *Release, Install, Run or Deploy the software for the client*
6. ***Maintenance :*** *Maintenance and Evolution*

What are the fundamental activities in software engineering ?

- Four (04) fundamental activities are common to all software processes.

1. **Software specification**, where customers and engineers define the software that is to be produced and the constraints on its operation.
2. **Software development**, where the software is designed and programmed.
3. **Software validation**, where the software is checked to ensure that it is what the customer requires.
4. **Software evolution**, where the software is modified to reflect changing customer and market requirements.

Professional Software Development vs Amateur Software Development

- People in business write **spreadsheet programs** to simplify their jobs;
- **Scientists** and **engineers** write programs to process their experimental data;
- **Hobbyists** write programs for their own interest and enjoyment

Professional Software Development vs Amateur Software Development

- If you are writing a program for **yourself**,

  ➔ You don't have to worry about **writing program guides**, **documenting the program design**, and so on.

- If you are writing software that **other people will use** and other engineers **will change**,

  ➔ Then you usually have to provide **additional information** as well as the code of the program.

➔ Software engineering is intended to support professional software development rather than individual programming.

What Kinds of Software Products to develop ?

Two main kinds of Software Products :

1. Generic Software Products

- They are standalone systems that are produced and sold on the open market to any customer who is able to buy them.
- Standalone systems run on a personal computer or apps that run on a mobile device. They include all necessary functionality and may not need to be connected to a network.

- **Examples** : **mobile apps**, **website scripts**, **downloadable software**, **software for word processors**, **project management tools**.

2. *Customized Software*

- These are systems which are commissioned and developed for a particular customer

- **Examples** :
  - Control Systems for an Electronic devices:
    - **Temperature , Motor, and SmartLighting Systems, …**
  - Software Systems to support particular business process for a company
    - **Customer Relationship Management (CRM) Systems, Human Resource Management (HRM), Accounting and Financial Management Systems, …**

Kinds of Software Products

- **Generic products vs Custom products**

- **In generic products**, the organization that develops the software **controls** the software specification.
- This means that if they run into development problems, they can **rethink** what is to be developed.
- **For custom products**, the specification is developed and **controlled** by the organization that is **buying** the software. The software developers must work to that specification.

Kinds of Software Products

- **What if another customer wants to buy the same customized software ?**
  - This depends on how you agreed with the customer on the ownership
  - Are you selling a copy or full and exclusive ownership of the software ?

3. Principles of Software Engineering

- There **are no universal standard principles** and laws of software engineering that are accepted by all.
- Principles are **suggested** to serve as guidelines and recommendations to **improve software development productivity** and achieve **greater success**.

Davis, Alan M. "Fifteen principles of software engineering." *IEEE Software* 11.6 (1994): 94-96.

1. **Make Quality Number 1**: Prioritize high-quality standards throughout development.
2. **High Quality Software is Possible**: Achieving excellent software quality is feasible with the right practices.
3. **Give Products to Customers Early**: Early delivery allows for user feedback and improvements.
4. **Determine the Problem Before the Requirements**: Fully understand the problem before defining requirements.
5. **Evaluate Design Alternatives**: Consider multiple design options to find the best solution.
6. **Use an Appropriate Process Model**: Choose a development methodology that fits the project's needs.
7. **Use Different Languages for Different Phases**: Apply the best-suited languages and tools at each stage.
8. **Minimize Intellectual Distance**: Keep the solution close to the user's perspective for ease of use and maintenance. The software's structure should be as close as possible to the real-world structure.

Davis, Alan M. "Fifteen principles of software engineering." *IEEE Software* 11.6 (1994): 94-96.

9. **Put Technique Before Tools**: Before you use a tool, you should understand and be able to follow an appropriate software technique.
10. **Get it Right Before You Make it Faster**: Ensure correctness before optimizing for performance.
11. **Inspect Code**: Regularly review code to catch and correct issues early.
12. **Good Management is More Important than Technology**: Effective management is essential to project success.
13. **People are the Key to Success**: Skilled and motivated team members drive project success.
14. **Follow with Care**: Adopt new methods thoughtfully, not just because they're trendy (or popular).
15. **Take Responsibility**: Be accountable for the quality and success of the software. There are no excuses. If you develop a system, it is your responsibility to do it right. Do it right, or don't do it at all.

4. Expected qualities of software

- Often referred to as software quality attributes:

- **Usefulness (Correctness (Functionality)):** Match between user requirements and the functions offered by the software.

All functionalities are correctly implemented

- **Usability (Acceptability) *:*** Software must be acceptable to the type of users for which it is designed. This means that it must be understandable, easy to use.

➔ Factors such as : appropriete user interface and adequate documentation

Users should also accept and adopt the software product by continuing to use it

4. Expected qualities of software

- **Reliability :** It is specified as the probability that the system will operate without failure in a specified environment and for a specified duration.
  - **Example**: If you accept that one (**01**) transaction out of **1000** may fail, then you can specify the probability of failure is **0.001**

    ➔ This **does not mean** that there will be exactly one (**01**) failure every 1000 transactions.
- **Robustness** : The software operate even in exceptional (**unexpected**) circumstances: erroneous inputs, hardware failures, etc.

4. Expected qualities of software

- **Efficiency and Performance** : Efficient use of resources – memory, execution time, communication time.
  - The software can handle large amounts of data or traffic.
- **Maintainability** : ease of maintaining the software by other developers. The software should be written in such a way that it can evolve to meet new customer needs
  - Maintenance represents more than 50% of the cost of the software during its lifetime
  - Three types of maintenance :
    - ***Corrective*** : correct errors
    - ***Adaptive*** : adjust the software to changes in its environment
    - ***Perfective*** : improve the quality of the software (performance, usability, etc.)

4. Expected qualities of software

- **Reusability and Openness** : the ability of software to be reused, in whole or in part, in new software.
- **Security :** The software is protected against unauthorized access and protects data and functions from malicious attacks

- ***Portability*** : The software can be operate under different hardware and software environments.
- **……**

5. Software life cycle:

- The life of a software is composed of different stages
- The succession of these stages forms a :

  « Software Development Life Cycle (SDLC) »

  - It defines the a sequence of activities involved in the development of software from inception to retirement.
  - The term **software process** is almost synonymous with software life cycle.

6. Software Life Cycle Models

Definition

- A **software life cycle model** is an abstract representation of the software development process.

- A Software Process Model is a description of what activities/ tasks need to be performed in what sequence under what conditions, by whom, to achieve the desired results

6. Software Life Cycle Models

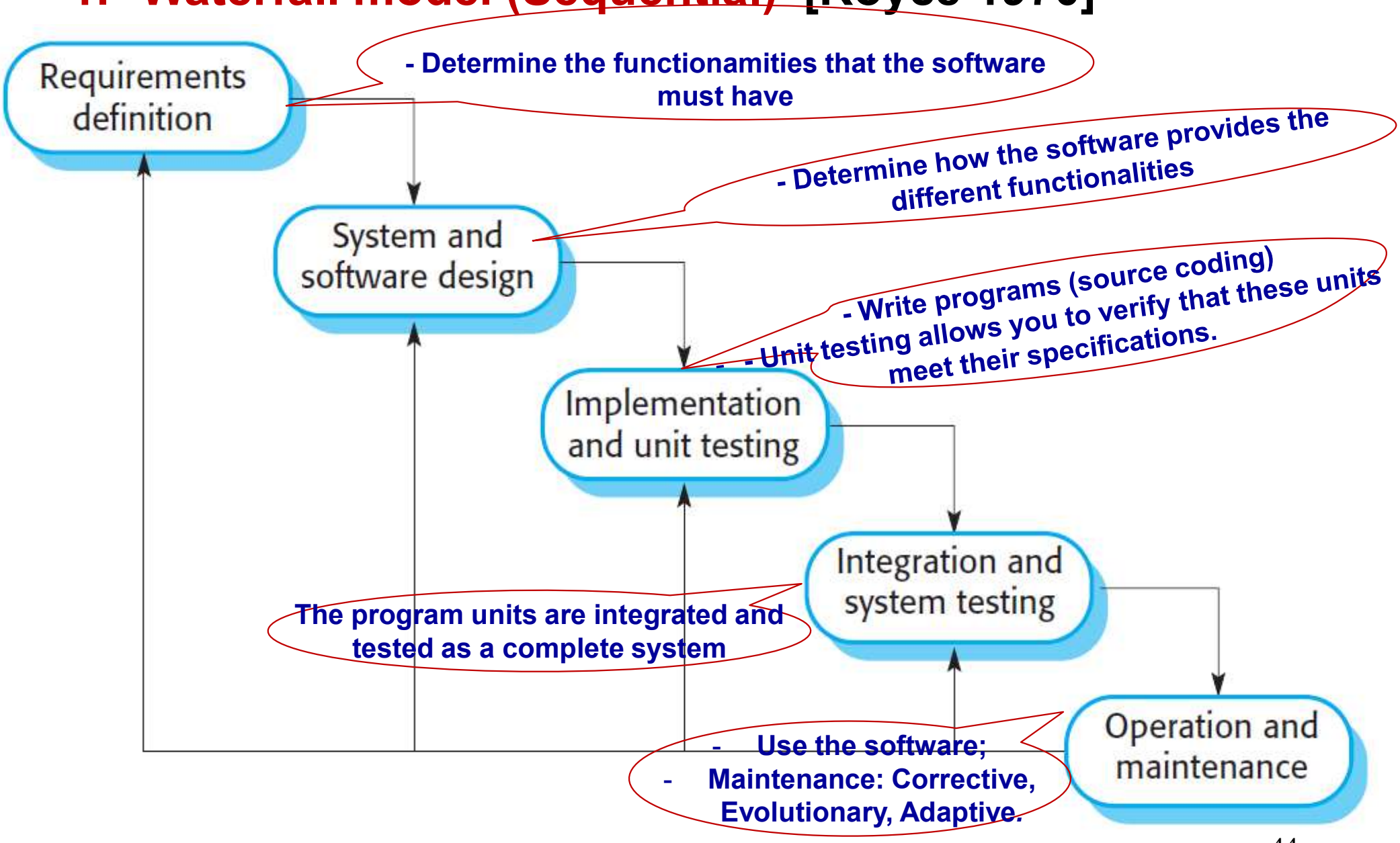
Very General Process Models

- They explain different approaches to software development

  - **Linear models**
    - **Waterfall model**
    - **V model**
    - **...**
  - **Nonlinear models**
    - **Prototyping**
    - **Incremental model**
    - **Spiral model**
    - **...**

6. Software Life Cycle Models

1. Waterfall model (Sequential) [Royce 1970]



1. Waterfall model (Sequential) [Royce 1970]

- ***Requirements analysis and definition:*** **The system's services**, **constraints**, and **goals** are established by consultation with system users. They are then defined in detail and serve as a **system specification**.
- **System and software design:** The systems design process allocates the requirements to either hardware or software systems. It establishes an **overall system architecture**.
  - **Software design** involves **identifying** and **describing** the fundamental software system **abstractions** and their **relationships**.
- **Implementation and unit testing** During this stage, the software design is **realized** as a set of programs or program units. **Unit testing** involves verifying that each unit **meets its specification.**

1. Waterfall model (Sequential) [Royce 1970]

- **Integration and system testing** The individual program units or programs are integrated and tested as a **complete system** to **ensure that the software requirements have been met**. After testing, the software system is delivered to the customer.
- **Operation and maintenance** The system is installed and put into practical use.
  - **Maintenance** involves **correcting errors** that were not discovered in earlier stages of the life cycle, **improving** the **implementation** of system units, and **enhancing** the **system's services** as **new requirements** are discovered.

1. Waterfall model (Sequential) [Royce 1970]

Example of Requirement Specification for Online Banking System:

Title: Online Banking System Requirements

Functional Requirements:

1.User Authentication:

1. Users should be able to create accounts with unique usernames and passwords
2. Users must log in securely with their credentials.

2.Account Management:

1. Users should be able to view their account balance, transaction history, and account details.
2. Users can transfer money between their own accounts.

3.Fund Transfer:

1. Users should be able to transfer money to other accounts within the bank.
2. The system should support international money transfers.

1. Waterfall model (Sequential) [Royce 1970]

Title: Online Banking System Requirements

Non-functional Requirements:

1.Security:

1. The system must use encryption for secure data transmission.
2. Account passwords must be stored securely.

2.Performance:

1. The system should respond to user requests within 2 seconds.
2. It must support at least 1000 concurrent users.

Caracteristics of Waterfall model

- The result of each phase is **one or more documents** that are **approved** (**“signed off”**).
- The following phase should not start until **the previous phase has finished**.
- The original model did not include the possibility of going back

❖ Disadvantages of the Waterfall model

❑ Inflexibility and Rigidity:

- Once a phase is completed, it is **difficult** and **costly** to go back and **make changes**. This lack of **flexibility** can be problematic if requirements **evolve** or if there are **errors discovered** in earlier phases.
    - If it is discovered that a requirement is **too expensive**, the requirements document should be **changed** to **remove that requirement**. This requires customer **approval** and **delays** the overall development process.
- As a result, both **customers** and **developers** may prematurely **freeze** the software specification so that no further changes are made to it.

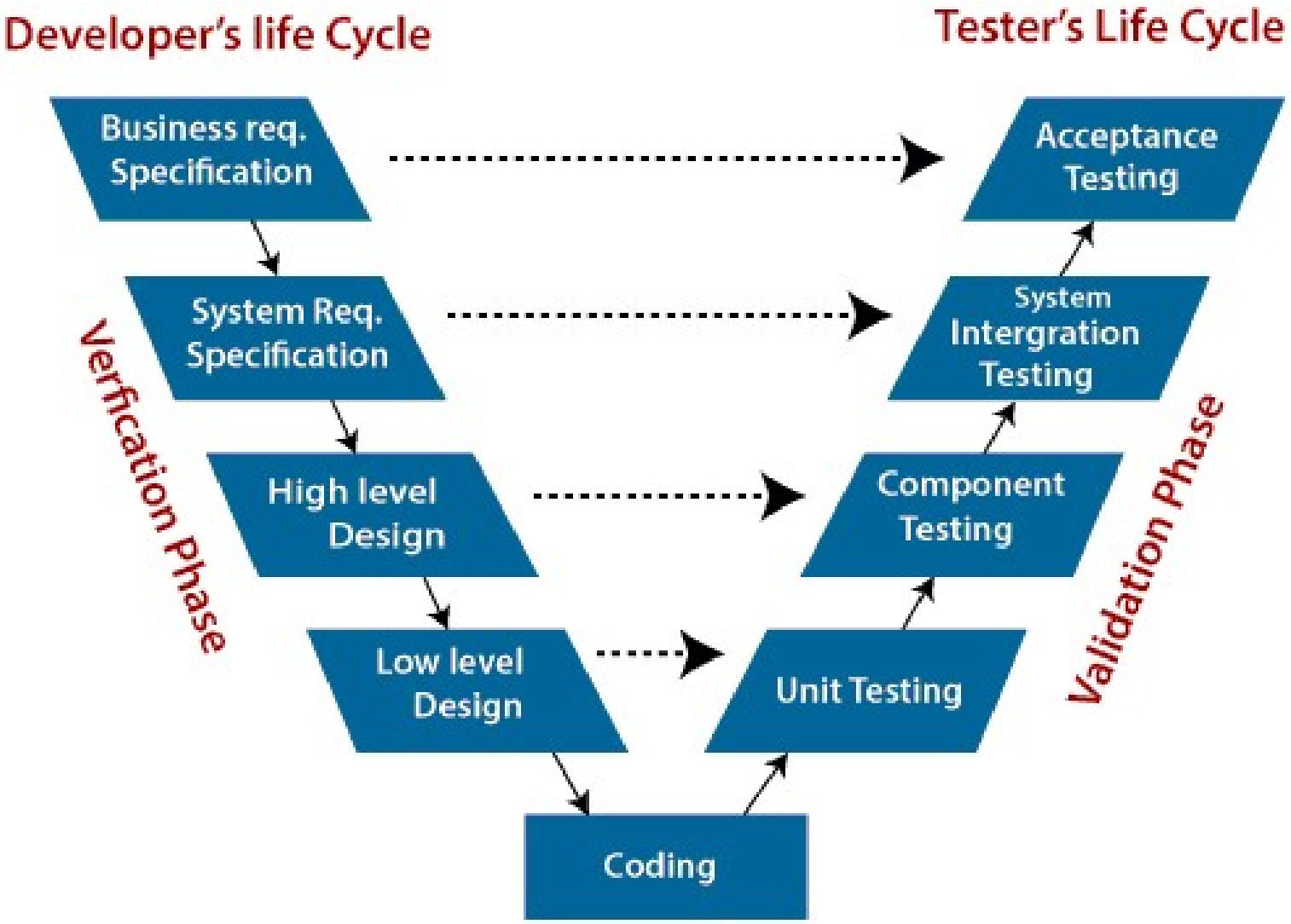
➔ This means that problems are **left for later resolution**, or **ignored**.

❑ The **working version** of the software will not be available until late in the development process.

- **In practice**, the phases **overlap** and **exchange** information with each other.
  - **Errors** and **omissions** in the programs and design may appear.
  - **New functionalities** can be identified.

➔ **Therefore, the system must evolve to remain useful.**

2. V Model (1980):



- **Expands** on the Waterfall model by emphasizing the **relationship** between **development phases** and corresponding **testing phases.**

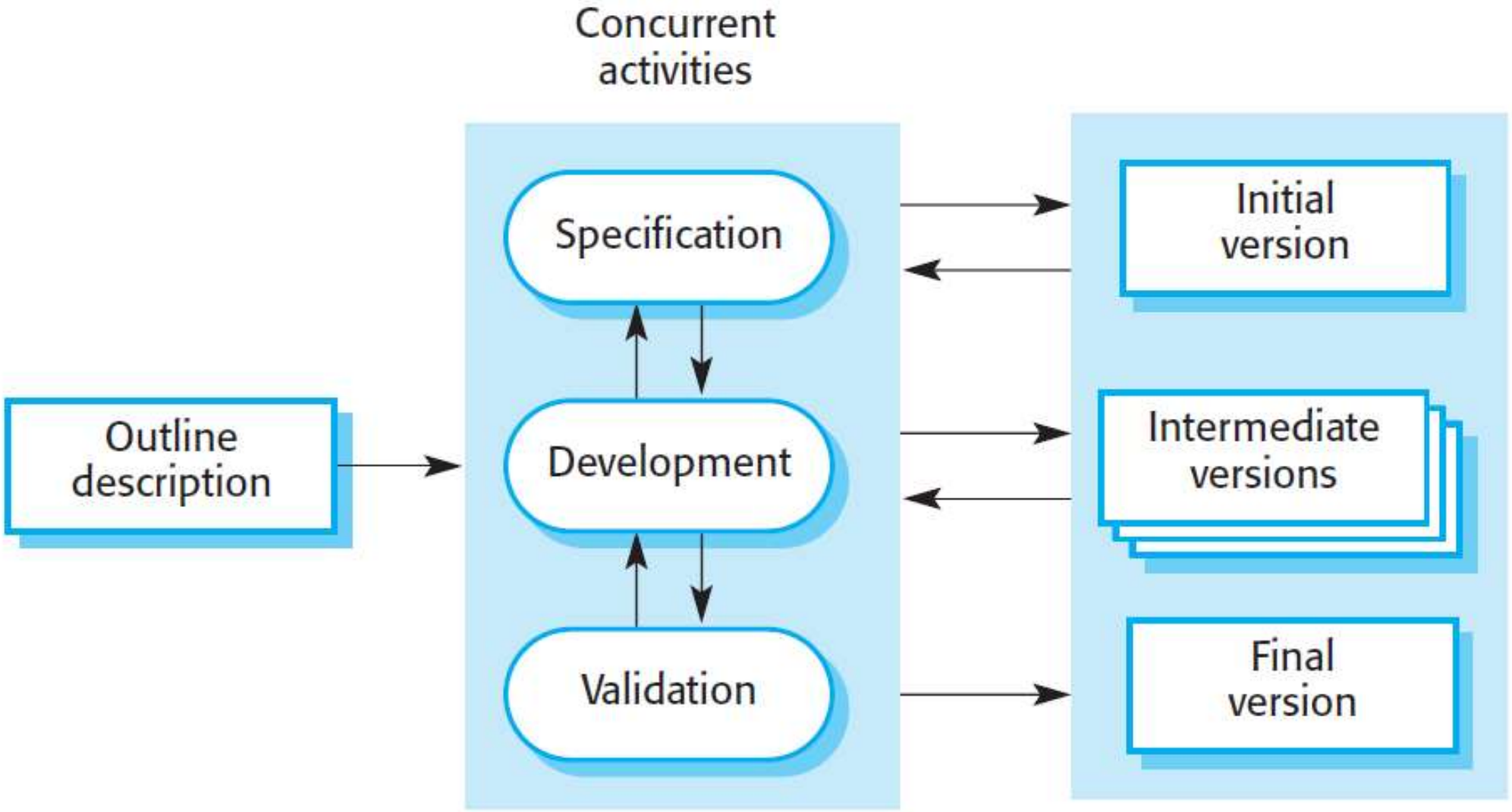
2. V Model (1980):

- Earlier detection of defects/problems
- The cost of changes is reduced
- It reflects how we solve problems:
  i. Progressing towards a solution gradually,
  ii. Going back when an error is made.

3. Incremental development

- It is based on the idea of developing an **initial implementation**, **getting feedback** from **users** and others, and **evolving** the software through **several versions** until the required system has been developed.
- It applies the activities of the waterfall model **iteratively**.
- The system is developed as a **series of versions** (**increments**).
- In each version, we add features to the previous version.
- The first increment is the **core of the system** (**contains the most important or necessary features**).

Incremental development



Incremental development

❑ Advantages over the waterfall model

- It is more **easier** to make **changes**.
- The **cost** of implementing **requirements changes** is **reduced**
- **Early delivery** and deployment of useful software to the customer **is possible**.
- Customers are able to use and **gain value** from the software earlier than is possible with a waterfall model.

❑ Disadvantages

- Difficulty in integrating increments.
- System structure tends to degrade as new increments are added.

4. Model by prototyping (1980s-1990s):

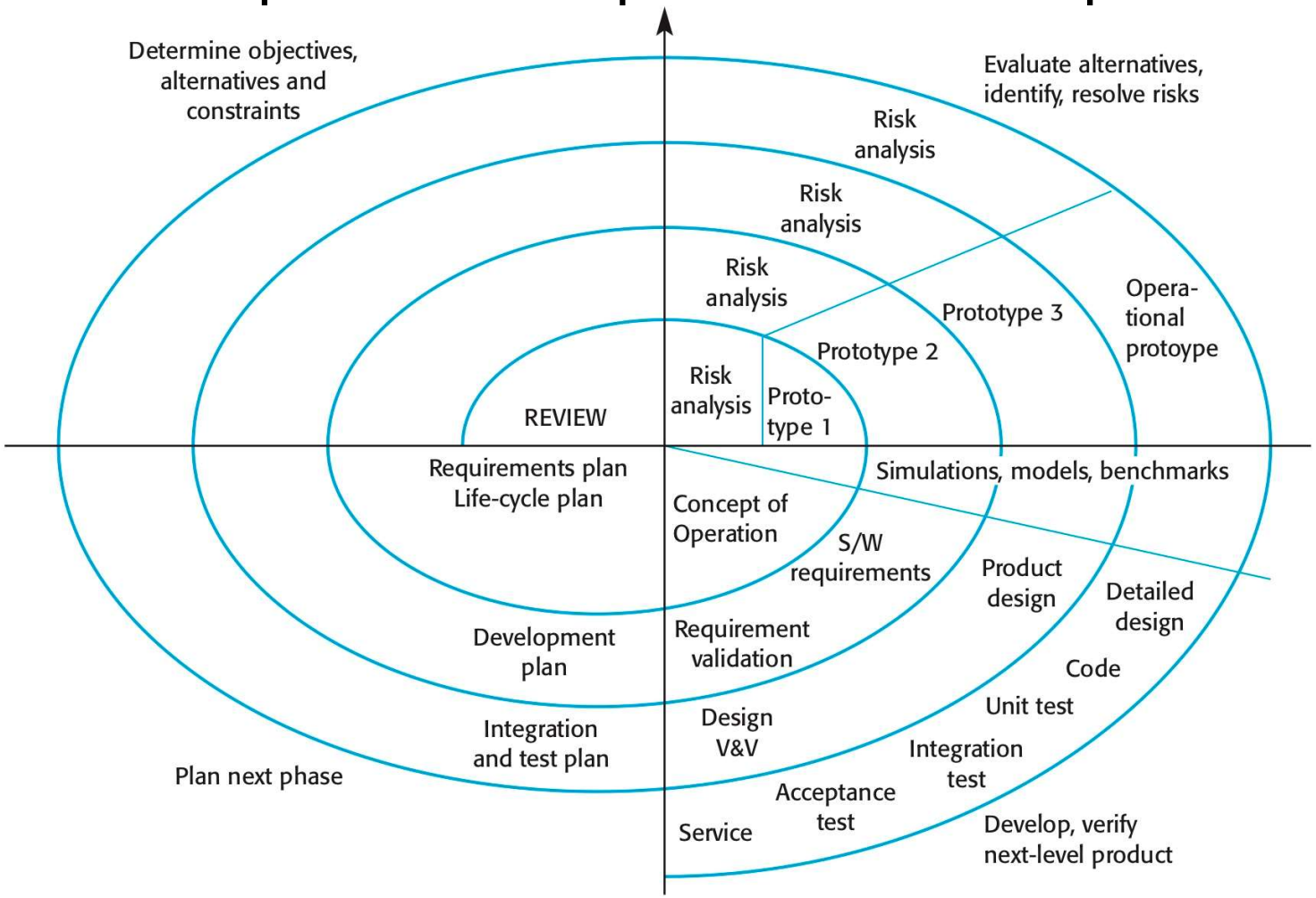
- Creating a prototype of the system (disposable) for the user to experiment with.
- The user provides **feedback** to the designers who modify the prototype.
- This is **repeated** until the **user is satisfied**

Advantages :

- Improved User Engagement
- The cost of accommodating new customer needs is reduced.
- Increased development costs

5. Boehm's spiral process model [1988]

- It combines elements of both iterative and waterfall models
- The software process is represented as a spiral



- Each loop represents a phase of the software process : requirements definition, system design and so on.

5. Boehm's spiral process model [1988]

- Each loop in the **spiral** is split into four sectors:

1. **Objective setting :** : identify:
   - **objectives** (performance, functionality, capacity to change, etc.),
   - **alternatives** (design A, design B, reuse, purchase, etc.),
   - **constraints** imposed to implement these alternatives (cost, planning, etc.).

   ➢ **Project risks are also identified.**

1. **Risk assessment and reduction:** For each of the identified project risks, a detailed analysis is carried out
2. **Development and validation:** a development model for the system is chosen
3. **Planning**: The project is reviewed and a decision is made to continue or not.

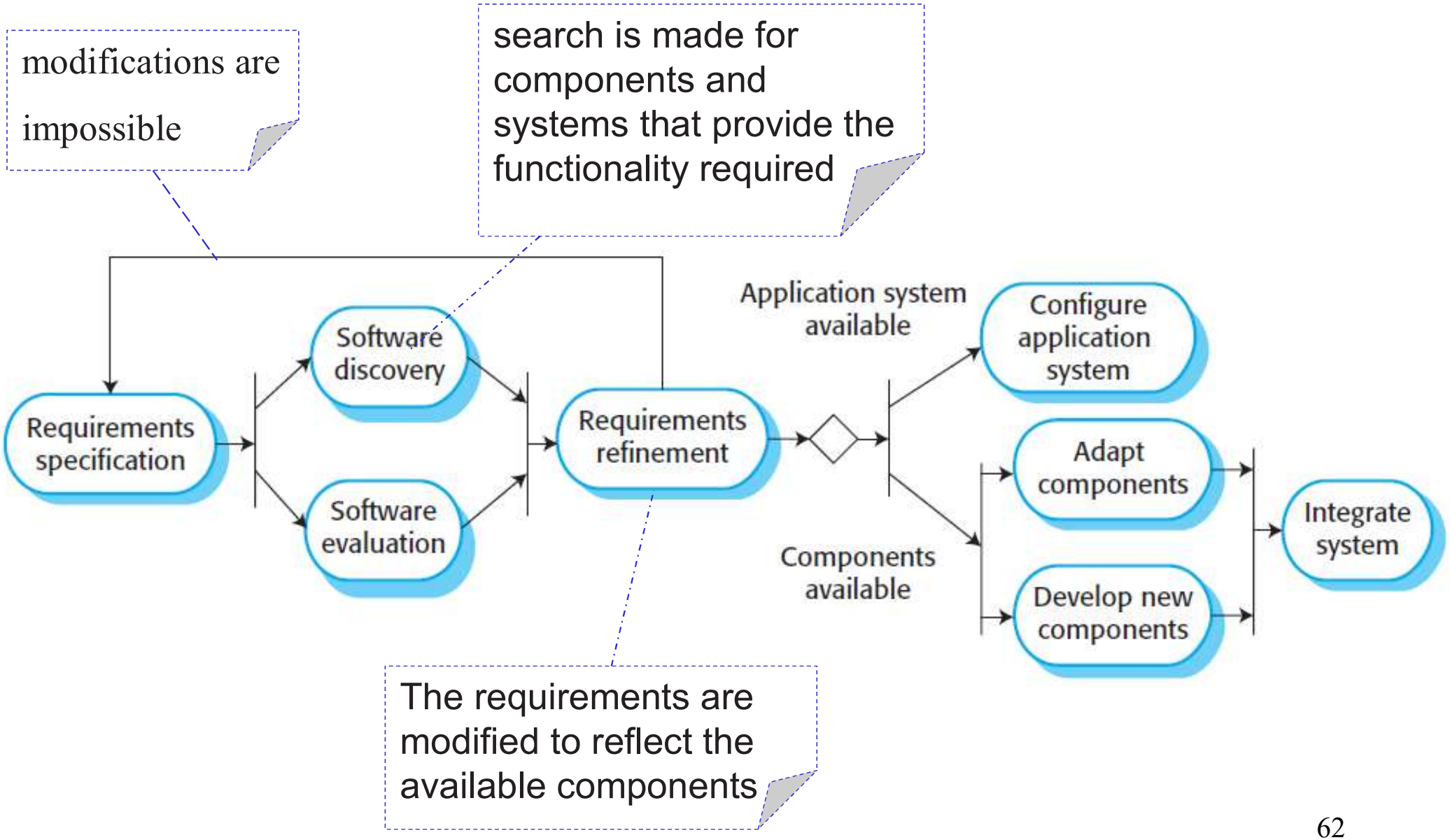
- Example of **major risks** in software development:

  - **Personnel failure**: hiring qualified staff, team spirit.
  - **Unrealistic schedule and budget**: detailed estimation of costs and schedules, reuse.
  - **Development of inappropriate user interfaces**: prototyping, scenarios, and user reviews.
  - **Performance issues**: simulations and modeling..

7. Integration and configuration- Development by reuse

- The development process aims to **reuse existing** software.
- This approach relies on the **availability** of **reusable components** or systems.
- The system development process focuses on **configuring** these components for use in a new setting and **integrating** them into a system rather than developing the system **from scratch**

7. Integration and configuration- Development by reuse



Development by reuse

- Software **reuse** has many **benefits**:
  - Reduce development time
  - Faster delivery of systems
  - Increase developer productivity
  - Reduce defect density
  - Improve software quality (if reusing something well done),
- ***But****, sometimes it is expensive to modify components for a new situation.*

Reuse techniques:

1. **Program Librairies**: **classes** and **fonctions** (Swing, SWT, String, Vector, ArrayList ….)
2. **CBSE**: **OSGi, EJB, CORBA, .Net, COM, COM…**
3. **Service-Oriented Systems**: The system is developed by composing services
4. **Design Patterns** : «it is a general, reusable **solution** to a commonly **occurring problem** in software design. »
5. **Legacy system wrapping**
6. **Model-driven engineering**
7. ….

7. Conclusion

- **There is no universal process model** that is right for all kinds of software development.

- It **depends** on the **customer** and **requirements**, the **environment** where the software will be used, and the **type of software** being developed.

Questions ?