Lecture 3

Software Development Lifecycle Models and Software Architecture

- The study of Software Lifecycle Models falls under the School of Computing and Mathematical Sciences and is taught by Dr Daniel Z. Hao and Dr Victoria R. Wright.

Intended Learning Outcomes (ILOs)

- The intended learning outcomes related to this topic include understanding the Software Development Cycle, understanding different Software Lifecycle Models (such as Waterfall, Agile, and Scrum), and understanding that designing software architecture is a part of the software development cycle.

Software Architecture

- Software architecture is defined by its **high-level structure**, its **main components & relationships**, and its role in **optimising performance, security, and maintainability**. Software architecture should be considered in all development phases.

Lifecycle of Software Development

- The Software Lifecycle Models represent a sequence of decisions, stages, or phases that determine the software built, which others will eventually use. When considering software development, **Stakeholders** and **Budget** are critical factors.

Stages in the Software Development Cycle:

- **Requirements Gathering:** The team collaborates with stakeholders to determine the goals and desired accomplishments of the software. Gathering **clear, detailed requirements is crucial** for guiding subsequent stages.
- **Design (Software Architecture):** This stage follows clear requirements. Architects and developers design the system structure, considering both functional and technical aspects. This often involves creating models,

diagrams, and selecting technologies to ensure the system meets the outlined requirements.

- **Implementation (Coding):** The phase where code is written and features are implemented.
- **Testing:** Involves debugging and feature verification. Testing includes unit testing, integration testing, and user acceptance testing.
- **Maintenance:** Includes updates, patches, and implementing new features. Maintenance may initiate a new life cycle for future updates.

Major Software Lifecycle Models

- The sources detail several major software lifecycle models, distinguished by their approach, key benefits, and invention approximate time:

Name	Key Benefits	First Invented Time (approximate)
Waterfall Model	Simple and easy to understand. Clear stages and structured.	1970s.
Agile Methodology	Flexible, adaptive, continuous feedback, and iterative development.	Early 2000s.
DevOps	Enhances collaboration, improves efficiency through automation.	Around 2009.
V-Model (Verification and Validation)	Clear and straightforward; each development stage has a corresponding testing phase.	1980s.
Spiral Model	Risk-driven approach, allows for incremental releases of the product.	1986 or 1988.
Scrum	Empirical feedback, self-organising teams, and customer satisfaction focus.	Early 1990s.
Lean Software Development	Eliminates waste, improves quality, delivers faster.	Early 2000s.
Extreme Programming (XP)	Enhances agility, customer satisfaction, and teamwork.	Late 1990s.
Others	Kanban and more.	

The Waterfall Model (Pure Model)

- The Waterfall model requires that **each phase of the development cycle must be completed before moving on to the next**.

  1. Origins and Characteristics

  - The model first appeared in the 1970 article, “Managing the development of large software systems,” by Dr. Winston W. Royce.
  - The term "Waterfall," although universally cited, does not appear in Royce's original article.
  - Royce actually suggested variants to mitigate the risks associated with the pure model.

  2. When the Pure Model Works Well

  The pure Waterfall model is effective when:

  - There is a **stable product definition**.
  - The **domain is well known**.
  - **Technologies are well understood**.

  3. Disadvantages and Failures

  - **Main disadvantage:** Not flexible.
  - **Problematic in setups where:** Requirements are partial and expected to change, developers are not domain experts, or technologies are new and evolving. This makes it less than ideal for most real-world projects.
  - **Core problems:** Doing everything correctly in a single sequence is unrealistic. The model fails because clients often do not know what they want, all alternatives cannot be mapped out in advance, the design cannot be fully decomposed into a sequence of decisions, and requirements and constraints constantly change.

  4. Case Study: FBI Virtual Case File (VCF) Failure

  - **Background:** This project in the early 2000s aimed to replace the FBI’s paper-based Automated Case Support system.
  - **Process:** The project was strictly managed using Waterfall phases: requirements → design → build → test → deploy.
  - **What Went Wrong:** Requirements changed rapidly after the events of 9/11, but the Waterfall structure could not adapt. Development took years, leading to an obsolete design by the time of delivery. The lack of

early testing or feedback meant major issues were only found at the end of the cycle. Delays were also worsened by vendor communication gaps and frequent leadership changes.

- **Outcome:** The project was **cancelled in 2005** before deployment, after spending over **$100 million**.
- **Lesson Learned:** In fast-changing environments, rigid Waterfall planning cannot keep up with evolving needs. Iterative and user-involved approaches (such as Agile) are more effective. The VCF project was replaced by the successful Sentinel project, which utilized Agile methods.

The Spiral Model

The Spiral model is an **incremental, risk-oriented lifecycle model**.

1. Origins and Philosophy

- Proposed in the 1988 article “A spiral model of software development and enhancement” by Boehm.
- It operates as a **risk-driven process** designed to support iterative development by deciding how to proceed based on reducing the risk of failure.

2. Motivation (Addressing Prior Model Flaws)

The Spiral model provides an alternative to:

- **Code-driven processes ("Code-and-fix"):** Where poor code and frustrated clients result from a lack of design.
- **Document-driven processes (Waterfall):** Where each step produces a new document, but the requirement for fully developed documents is unrealistic.

3. Process and Structure

- It follows a **Risk-Driven Approach** and is an **Iterative Process** involving multiple iterations.
- **Phases in each iteration:** Set Objective, Risk Analysis and Mitigation, Development and Testing, and Planning.

4. Advantages and Disadvantages

- **Main advantages:** Risk reduction, functionality can be added, and software can be produced early in the lifecycle.

- **Main disadvantages:** Requires specific expertise and estimations for risk analysis, is highly dependent on risk analysis outcomes, and is a complex process.

Agile Methodology

- Agile is often summarized by the philosophy: **‘ Users are always right… ’**. Agile development occurs in **iterative cycles (sprints)**.

  1. Advantages
     - Customer Satisfaction.
     - Flexibility and Adaptability.
     - Early Delivery/Deployment.
     - Collaboration and Team.
     - Continuous Improvement.

  2. Disadvantages
     - Cost/Overheads.
     - Less Predictability.
     - Team Dependency.
     - Customer Involvement.

Software Requirement Specifications (SRS) and Design

Core Requirements for Software Development

Before developing any software, the two most important things a developer needs to know are the **Stakeholder** and the **Budget**.

What is the Software Requirements Specification (SRS)?

The **Software Requirements Specification (SRS) document** outlines the functional and non-functional requirements of a software system.

- It serves as a **contract between stakeholders and developers**.
- An SRS helps stakeholders understand the system’s needs and provides developers with a guideline, thereby reducing the chances of rework.

- An SRS is a description of the system's functionalities, constraints, and behaviors.

Functional vs. Non-functional Requirements

Aspect	Non-functional Requirements	Functional Requirements
Objective	**How the product works**.	**What the product does**.
Focus	Focus on user expectations.	Focus on user requirements.
Documentation	As a quality attribute.	In use case.
End Result	Product properties.	Product features.
Testing	Performance, usability, security testing, etc. **Tested after functional testing**.	Component, API, UI testing, etc. **Tested before non-functional testing**.
Types	Operational, revisional, and transitional.	External interface, authentication, authorization levels, and business rules.

Example Scenario (E-commerce Payments):

- **Functional Requirement Example:** The system must allow users to select a payment method (credit card, PayPal, etc.); it should validate the payment information; and it must process payments, providing confirmation upon successful transaction.
- **Non-functional Requirement Example:** The system should handle up to 10,000 transactions per minute; maintain **99.99% availability**; and payment processing should not take more than 2 seconds on average.

The Impact of Non-functional Failure (Amazon S3 Outage): The failure of a non-functional requirement (availability) can compromise a system's functional goals. For instance, during the Amazon S3 service disruption in 2017 (caused by a typo):

- The **Functional Requirement** was that the system must store and retrieve data efficiently for millions of users.
- The **Failure of the Non-functional Requirement (Availability)** meant that the outage took down a significant portion of the service for several hours, causing data retrieval to fail, and consequently, websites and apps dependent on S3 were inaccessible, compromising the functional goal.

Representing Requirements

Requirements are typically represented using **Natural language** (plus tables and graphs) or **User stories**.

1. Natural Language (NL) Requirements

- The **overwhelming majority** of software requirement specifications (SRSs) in the industry are written in Natural Language (NL).
- Estimates suggest 90% of SRSs are written only in NL, and 72% are written in common NL.

Pros of NL Requirements	Cons of NL Requirements
All stakeholders can read and understand them.	Do not encourage order and structure.
Some stakeholders may help in writing them (Very common in Agile).	Difficult to analyse automatically.
	Prone to **multiple interpretations**.

2. Shortcomings of Natural Language

- **Ambiguity:** Defined as being open to more than one interpretation or inexactness.
- **Incompleteness:** Not having all the necessary or appropriate parts. Requirements should strive for completeness but are *always* incomplete to some extent (e.g., regarding environmental conditions or implementation details).
- **Precision:** The quality, condition, or fact of being exact and accurate. Over precision is not always best, and requirements may be imprecise on purpose.

◦ *Note:* Ambiguity should not be confused with incompleteness or precision.

3. Addressing NL Shortcomings

Solutions involve learning to write and detect ambiguity, and using **restricted NL**.

- A **controlled language** is a precisely defined subset of natural language for use in specific environments.
- Controlled NL increases readability and understandability by reducing inherent ambiguity through a **restricted grammar and a fixed vocabulary**.

4. User Stories

User stories are widely popular, commonly used in agile methods, and focus on the **user benefits**, rather than solely on the system characteristics.

- **Template:** User stories typically follow a format detailing: **Who?** (the user role), **What?** (the goal), and **Why?** (the benefit).
- **Example:** "As a developer, I want to change the status of a bug report from assigned to fixed, in order to let others know it has been fixed".
- **Pros:** They are simple, short, and effective for communication with stakeholders. They may also map directly to acceptance tests, enhancing **traceability**.
- **Cons:** They still rely on NL, making them prone to ambiguity, and they are not very flexible (though this inflexibility is sometimes considered an advantage).

Writing Good SRS and Common Challenges

To write a good SRS, one must **be clear and concise**, **avoid technical jargon** unless necessary, and ensure the requirements are **testable and traceable**. Including diagrams and use cases is also helpful.

Common Challenges in SRS include:

- Incomplete or changing requirements.
- Ambiguity leading to misinterpretation.
- Stakeholder conflicts.
- Difficulty in balancing functional and non-functional requirements.

Spring MVC: Views

1. Views and Front-end

- In Spring Boot, a **'view'** refers to JavaServer Pages (JSP) or other templates that users see. Views are responsible for **rendering data sent from controllers**.
- The terms "**View**" (rendering data from controller) and "**Front-end**" (user interaction) are related but are strictly speaking, different concepts.

2. Spring Boot

Spring Boot is a framework designed to simplify building web applications.

- It provides **embedded servers** (like Tomcat, Jetty).
- It offers powerful microservices tools.
- It helps speed up development by reducing boilerplate code (e.g., complex XML configurations).
- Though JSP is rarely used for view rendering today, Spring Boot supports alternatives like Thymeleaf, Velocity, and **RESTful API for Client-side rendering**.

3. JSP and JSTL

- JSP (JavaServer Pages) is still used for legacy systems and server-side views. JSP tags allow developers to directly use Java code (`<% %>`) within the page.
- **JSTL (JavaServer Pages Standard Tag Library)** provides a set of XML-like tags for common tasks, including iterating over collections (`<c:forEach>`), conditional logic (`<c:if>`, `<c:choose>`), and formatting dates and numbers.

Preparation Questions (Answers)

What fields does the `Goal` class have?

- In the provided project, the `Goal` class typically contains:

```
private int id;
private int minutes;
private List<Exercise> exercises;
```

What fields does the `Exercise` class have?

- `Exercise` usually contains:

```
private int id;
private int minutes;
private String activity;
```

How are `Goal` and `Exercise` related?

- A **Goal has a list of Exercises**.

- A **one-to-many relationship**.
- The `Goal` class contains:

```
private List<Exercise> exercises = new ArrayList<>();
```

and controllers add exercises to goals.

Exercise A — View for `/`

Create:

`src/main/webapp/WEB-INF/views/index.jsp`

index.jsp

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Home</title>
</head>
<body>

<p><a href="getGoals">List Goals</a></p>

<p><a href="addGoal?id=0&minutes=20&exerciseDesc=swimming">
    Add Goal
</a></p>

<p><a href="addExercise?goalId=0&id=0&minutes=20&activity=trekking">
    Add Exercise
</a></p>

</body>
</html>
```

Exercise B — View for `/addExercise?...`

Create:

`src/main/webapp/WEB-INF/views/addExercise.jsp`

This page receives an `exercise` model attribute using `@ModelAttribute`.

addExercise.jsp

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Exercise Added</title>
</head>
<body>

<p>Added ${exercise.activity} for ${exercise.minutes} min.</p>

</body>
</html>
```

✔️ Uses JSP EL (`${ ... }`) to print **activity** and **minutes**.

Exercise C — View for `/getGoals`

- Add dependency in `build.gradle`:

build.gradle JSTL dependency

```
implementation 'jakarta.servlet.jsp.jstl:jakarta.servlet.jsp.jstl-api:3.0.1'
```

- Now create the JSP file:

`src/main/webapp/WEB-INF/views/getGoals.jsp`

getGoals.jsp

```
<%@ taglib prefix="c" uri="jakarta.tags.core" %>

<!DOCTYPE html>
<html lang="en">
```

```
<head>
    <meta charset="UTF-8">
    <title>Goals</title>
</head>
<body>

<h2>List of All Exercises</h2>

<!-- Outer loop: all goals -->
<c:forEach var="goal" items="${goals}">

    <!-- Inner loop: exercises inside each goal -->
    <c:forEach var="ex" items="${goal.exercises}">
        <p>${ex.activity} for ${ex.minutes} min.</p>
    </c:forEach>

</c:forEach>

</body>
</html>
```

✔️ Two nested **forEach** loops.

✔️ Displays all exercises from all goals.

Exercise D — Improve Layout (Images, CSS, JavaScript)

- Place files in:

```
src/main/webapp/
```

Example File Structure:

```
src/main/webapp/
 ├── style.css
 ├── script.js
```

```
└── images/
    └── fitness.png
```

Add CSS and JS to `index.jsp`

```
<link rel="stylesheet" href="/style.css">
<script src="/script.js"></script>

<img src="/images/fitness.png" width="200">
```

Example `style.css`

```
body {
  font-family: Arial, sans-serif;
  background-color: #f2f2f2;
  padding: 20px;
}

a {
  font-size: 18px;
  color: blue;
}
```

Example `script.js`

```
console.log("Web App Loaded!");
```