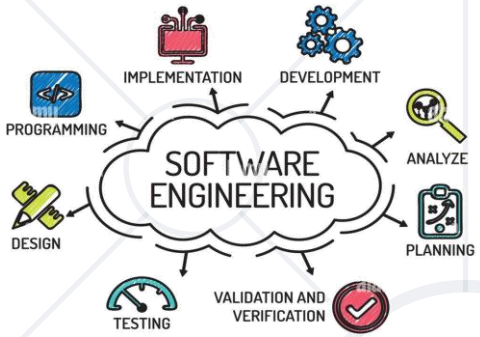
# Software Engineering Fundamentals

## Software Development Concepts, Phases and Practices




**SoftUni Team**

**Technical Trainers**

Software University

SoftUni

**Software University**

https://softuni.bg

# Have a Question?

Software University

# sli.do

# #Dev-Ops

# Table of Contents



# Software Development Lifecycle (SDLC)

Requirements → Design → Code → Test → Deploy

# Coding != Software Engineering

- Inexperienced developers consider "*coding == development*"
- Software engineering is **not just coding!**
  - In most projects **coding is only 20%-30%** of the project activities!
  - The **software development process** involves many more
  - The important decisions about the software are taken during the **requirements analysis** and **software design** phases
  - **QA**, **testing**, **integration**, **reviews**, **deployment**, **documentation**, and **management** are often disparaged
- **Software engineering** is about building software **professionally**, through a **manageable** and **predictable** process

# Software Engineering

- **Software engineering** is an engineering discipline aimed to provide knowledge, processes, practices, and tools for
  - Defining software **requirements**
  - Creating software **design**
  - Software **construction**
  - Software **testing** and **QA**
  - Software **deployment**
  - Software **maintenance** tasks
  - Managing the **development process**

# Software Development Lifecycle (SDLC)

- The **Software Development Lifecycle** (**SDLC**)
  - Analyzing and defining software **requirements**
  - Creating software **architecture** and **design**
  - Software **construction**
  - Software **testing** and **QA**
  - Software **deployment**
  - Software **maintenance**
- Methodologies manage the **development process**

# Software Requirements Analysis

Requirements, User Stories, Prototypes, Specification

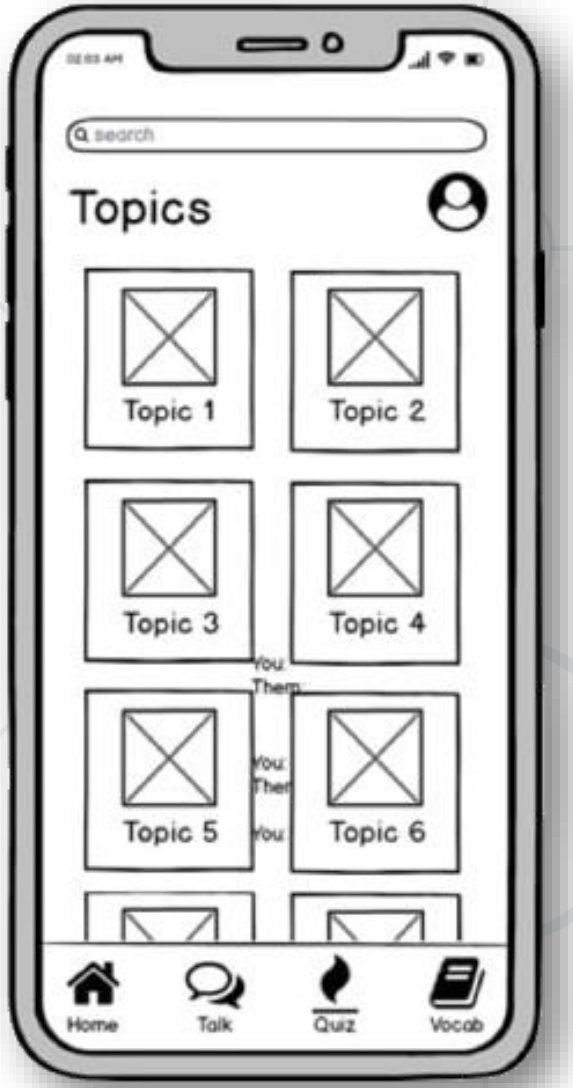
# Software Requirements

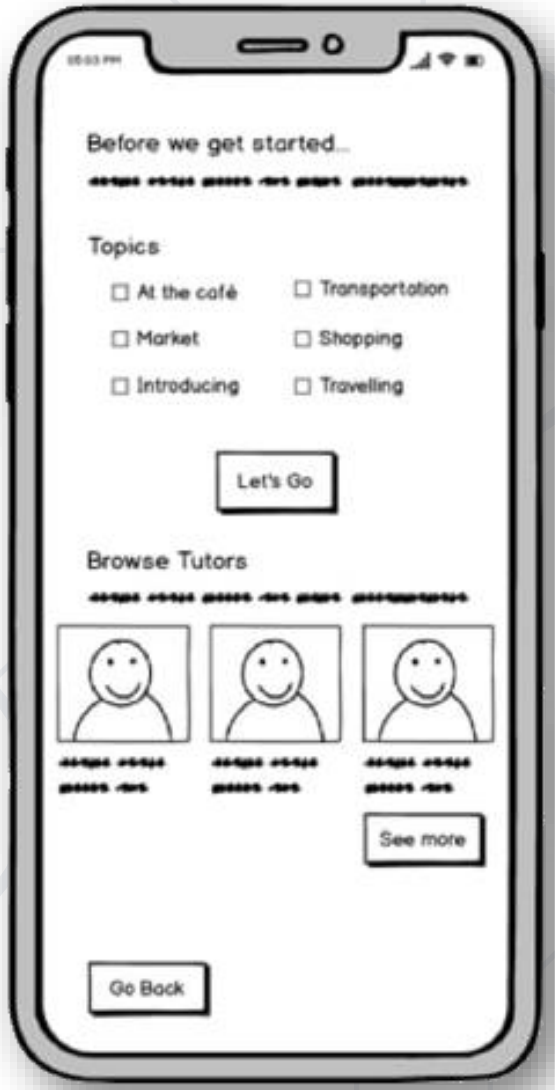
- **Software requirements** describe the functionality of the software
  - Answer the question "**what shall it do?**", not "**how shall it do it?**"
  - Define **constraints** on the **system**
- Two kinds of **requirements**
  - **Functional** requirements
    - Product functions, enabling users to achieve their tasks
  - **Non-functional** requirements
    - Reliability, efficiency, usability, maintainability, etc.

# Requirements Analysis

- **Requirements analysis** starts from an idea about the system
  - Customers often have a broad idea but may lack specifics
  - Requirements come roughly → adjusted during the development
  - Requirements change constantly!
- Analysis produces some requirements documentation
  - **User stories** / **UI prototype** / **Software Requirements Specification** (SRS) / informal system description
    - Should be clear, concise and unambiguous

# UI Prototype – Example





# SRS – Example



| | | | |
|---|---|---|---|
| 1.0 | 08/14/03 | M. Miranda/S. Roach | Final review changes before client's approval |
| 1.1 | 08/15/03 | M.Miranda/S.Roach | Corrections after validation by clients |
| 1.2 | | M. Miranda/S. Roach | Corrections by class, administrator functions |

TABLE OF CONTENTS







controls and cotton research. Data stored by the CDMS will be collected in the field by growers and researchers, taken from satellite images, derived from weather reports, and collected in the laboratory. Since the growers, researchers, and laboratory technicians may be geographically dispersed, the CDMS must allow users to enter and receive data remotely. Therefore, an internet-based system is envisioned.

The hierarchy of the data can be described in the following manner:

- Projects can have one-to-many seasons.
- Seasons can have one-to-many treatments.
- Treatments can have one-to-many fields.
- Field can have one-to-many plots.
- Plots can have one-to-many reps.
- Reps have many plants.

Figure 1 below is an example of the layout of a cotton field. The field is represented by the large rectangle. This field contains three plots. Plot 1 has 4 reps, each with 2 treatments. The treatments are irrigation/fertilizer. The numbers 4, 5, and 6 represent the number of irrigations. The high and low represent the amount of fertilizer used. The highlighted square would be referenced as "Field A, Plot 1, Rep 1, 4 High."

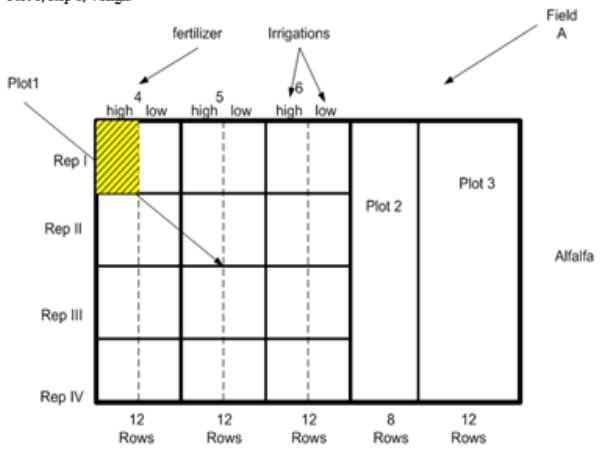



Figure 1 Field Layout

## 2.1 Product Perspective

CDMS will be a tool to manage information related to the prediction and analysis of cotton growth. The main uses of the system include the following.





- The system will act as a repository of insect data for cotton, including insect images, classifications, features, and population histories.
- The system will act as a repository for cotton growth data in the Upper Rio Grande Valley.
- The system will provide data to a DSS for advising growers.
- The system will provide electronic access to data for growers and researchers.

The CDMS will need to interact with both new and existing systems. Software tools such as GIS will provide access to geographic and satellite imaging data. The COTMAN software will generate Cotton yield predictions. Weather data will be obtained from the National Weather Service web site or university weather stations located throughout the state (http://www.weather.nmsu.edu). Insect classification and population data will be obtained from the image analysis software currently under development at NMSU.

## 2.2 Product Features

The use case diagram shown in Figure 2 describes the main features of the CDMS. Briefly, the system must interact with external programs such as COTMAN, and it must store data and provide query access to the data.

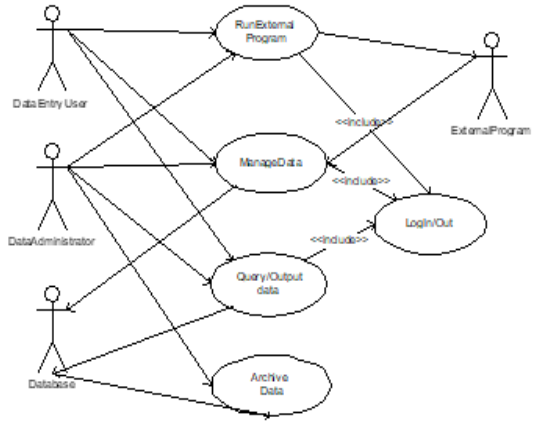


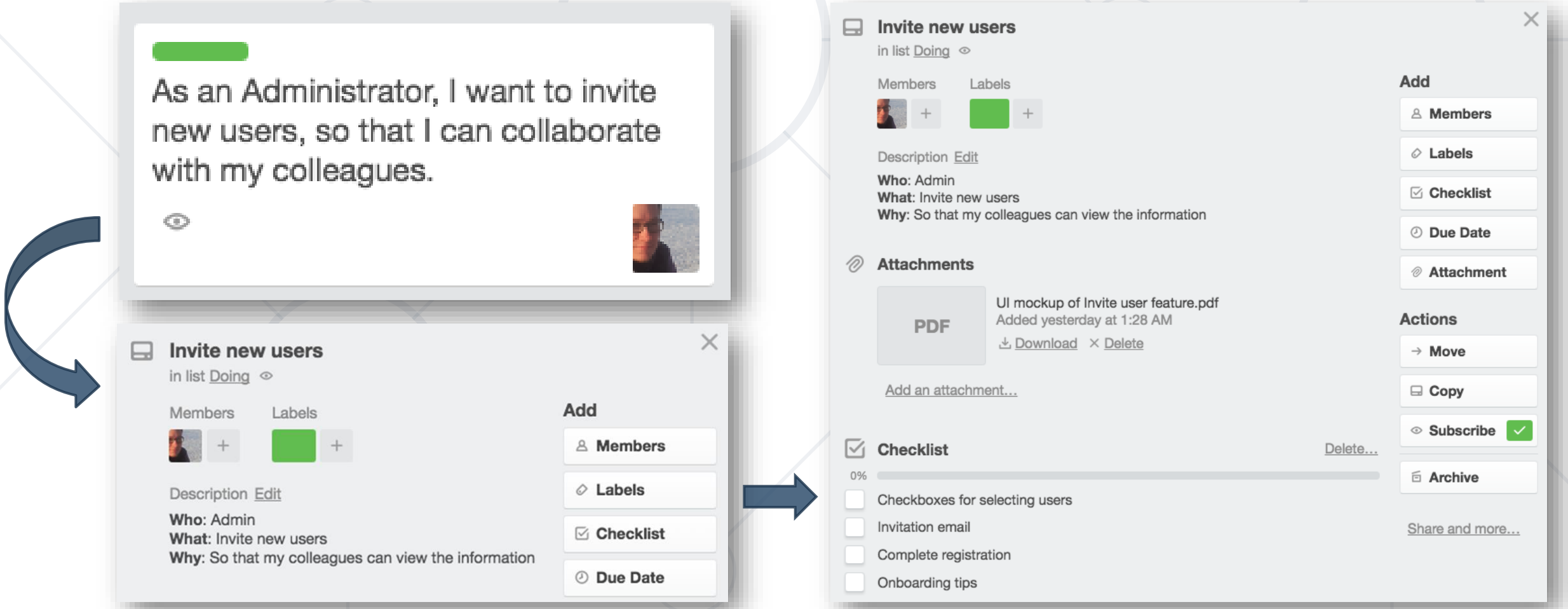

Figure 2 Use Case Diagram

### 2.2.1 Use Cases

Use cases are descriptions of interactions that users have with the system. They define the boundaries of the system as well as the sequence of operations needed to accomplish a task. The use case description contains descriptions of actors and use cases. An actor is any entity outside of the system

# User Story – Example

- A story starts from a simple idea (1-2 sentences)
- As the team discusses the story, they add more details



# Software Requirements Change Over Time

- It is always **hard** to **describe** and **document the requirements** in a comprehensive way
  - Good requirements save time and money, but are hard to reach
- **Requirements always change during the project!**
  - Good requirements **reduce the changes**
  - UI prototypes **significantly reduce changes**
  - Agile methodologies are **flexible to changes**
  - Use **user story tracking tools** to handle changing requirements

# Live Demo

User Stories, UI Prototypes and SRS

# Software Architecture and Design

# Software Architecture and Software Design



- **Software architecture and design**
  - Technical descriptions (e.g., diagrams) about how the system implements the requirements
- The **software (system) architecture** describes the **subsystems**, their **responsibilities** and **interactions**
  - High-level infrastructure of a software system
  - **Good** software architecture == **easier** maintenance and scalability

# Software Architecture and Software Design

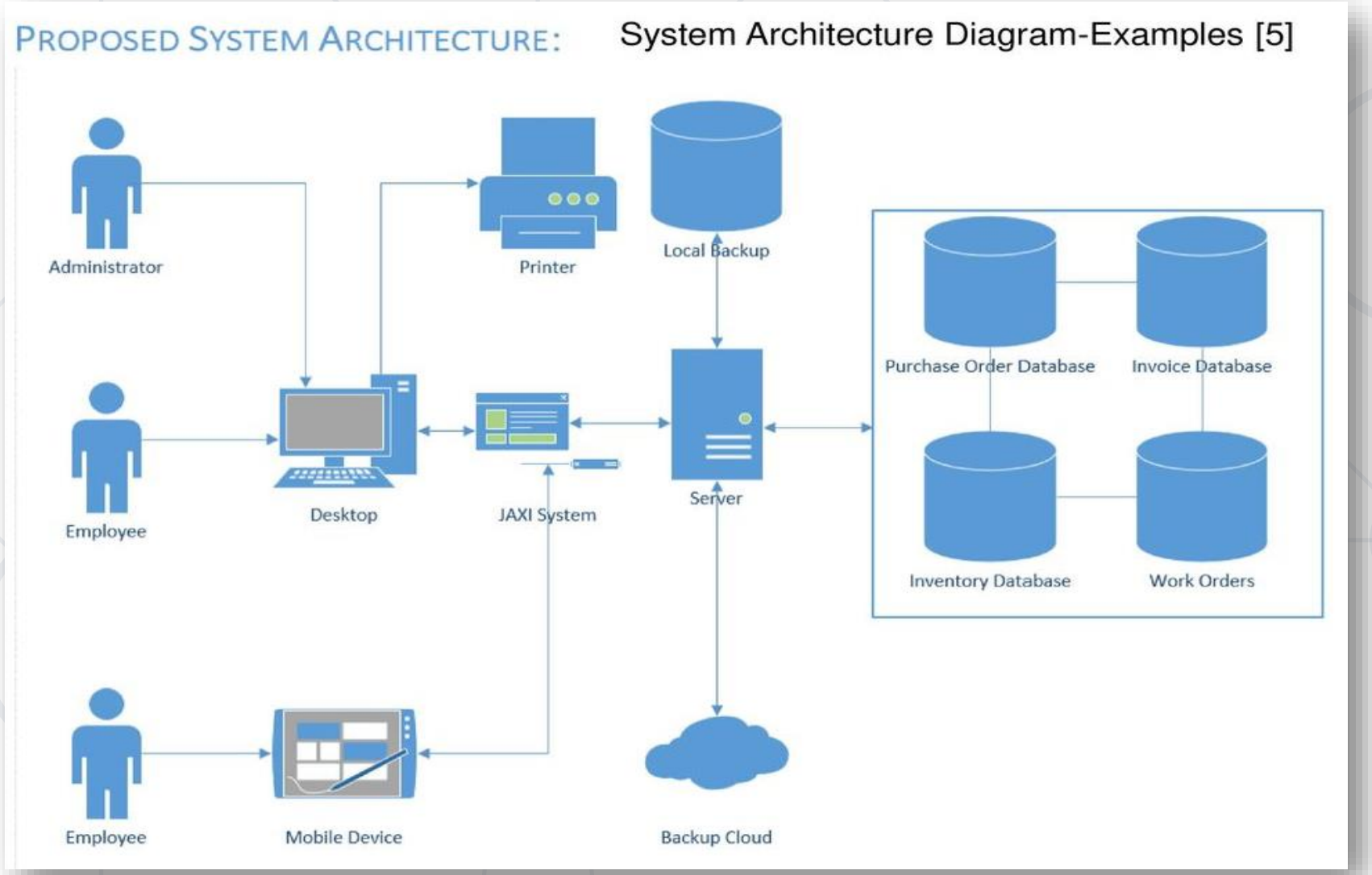


- The **software design** describes the system design at a lower level
  - Functions of individual modules, classes, etc.
- **Design patterns** in software design
  - **Proven** solutions to **recurring** problems
  - Fundamental in software engineering

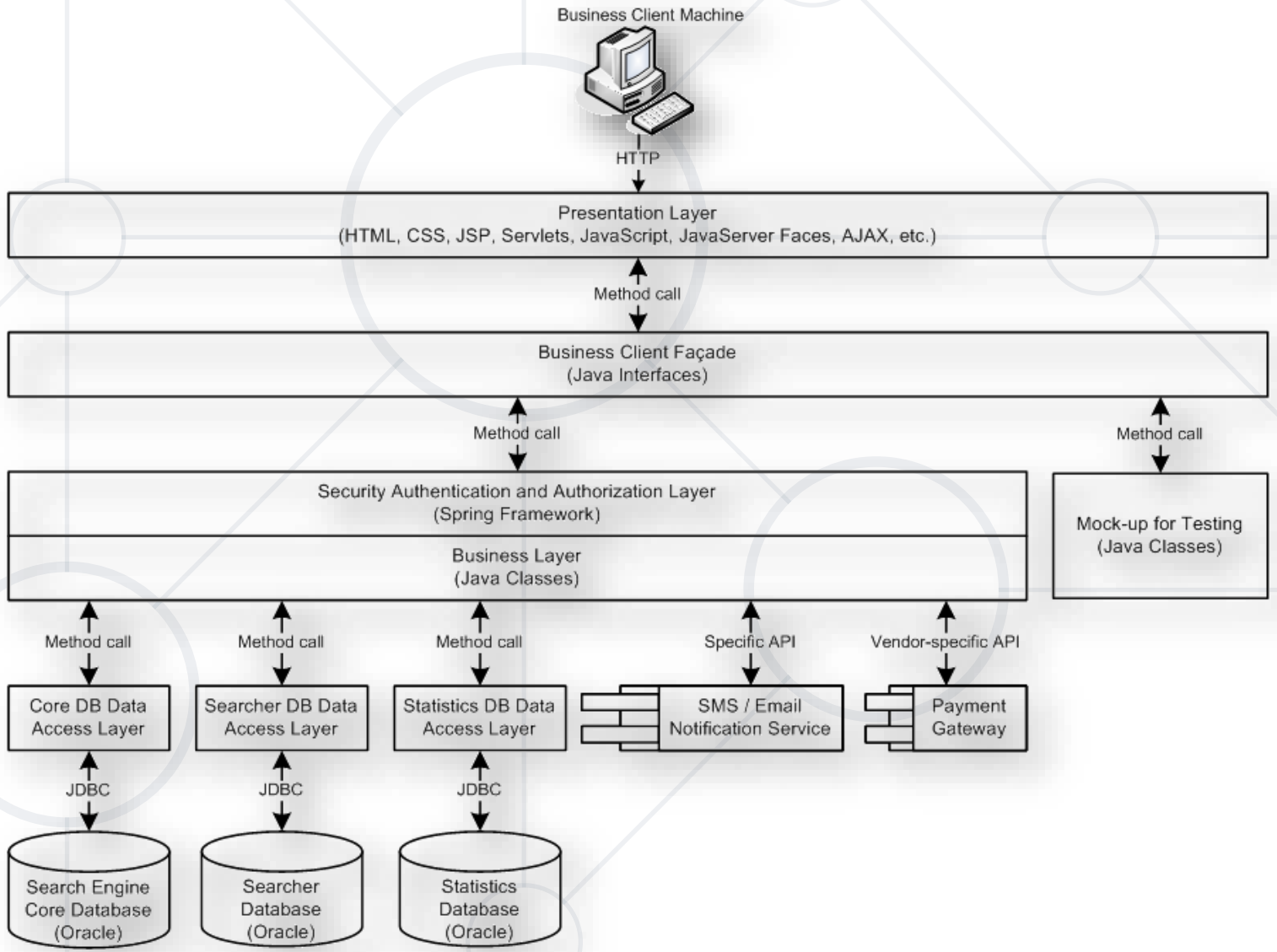
# System Architecture

- The **system architecture** is a conceptual model, describing
  - How the system will be **decomposed into modules** (subsystems)
  - Responsibilities of each **module**
  - **Interaction** between the **modules**
  - **Platforms** and **technologies**, communication **protocols**
- Can be **formal** or **informal** (typically)
  - Consists mostly of **diagrams** / **blueprints**

# System Architecture Diagram



# Software Architecture Diagram

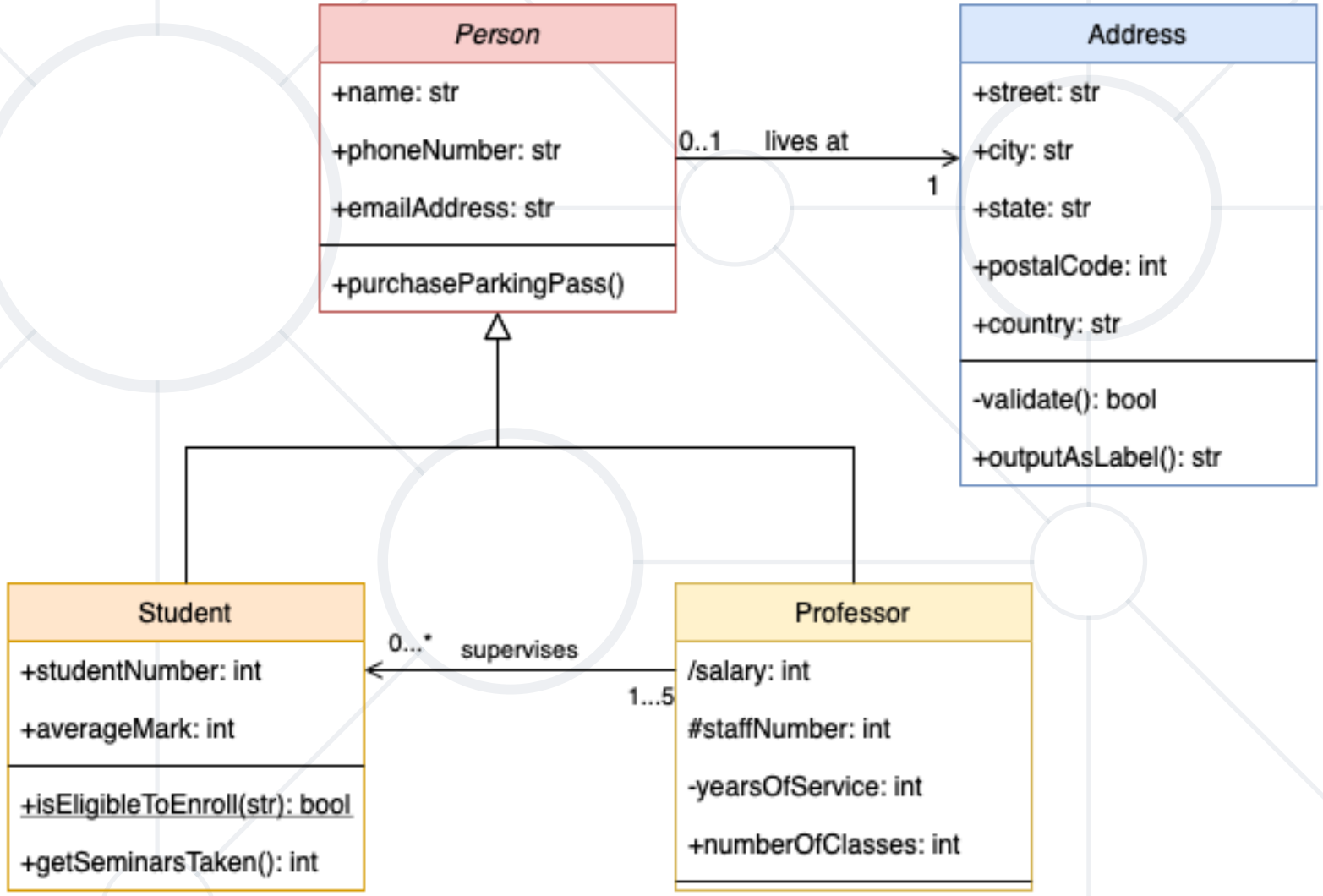


# Software Design

- **Detailed design**
  - Describes the internal module structure
  - Subcomponents, interfaces, process design, data design
- **Object-oriented design**
  - Describes the classes, their responsibilities, relationships, dependencies, and interactions (usually in UML)
- **Internal class design**
  - Methods, responsibilities, algorithms and interactions

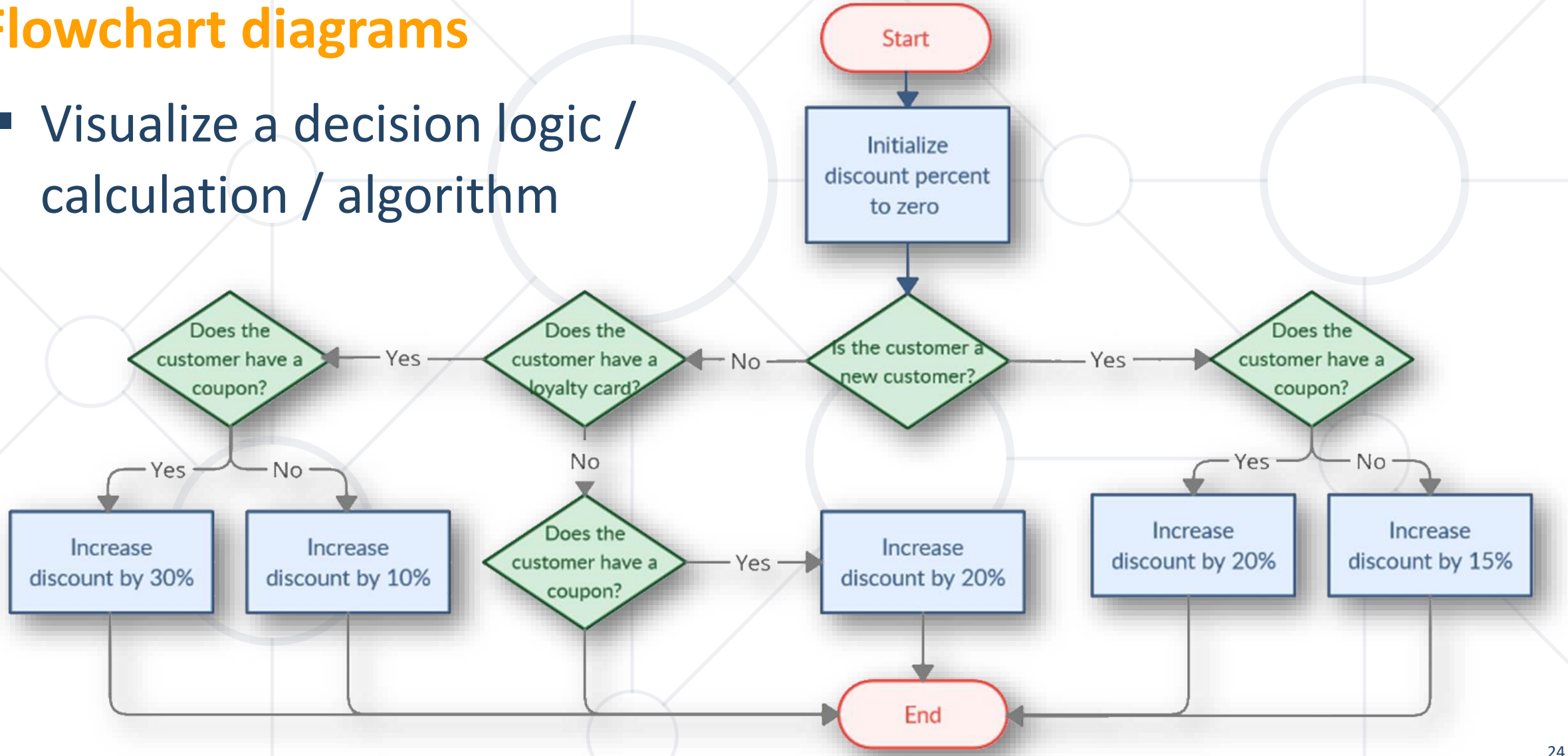
# UML Class Diagram

- **UML Class diagrams**
  - **UML** == **U**nified **M**odeling **L**anguage
  - Visualize classes and their relationships



# Flowchart Diagram

- **Flowchart diagrams**
  - Visualize a decision logic / calculation / algorithm



# Software Design Document (SDD)

- The **Software Design Document** (SDD)
  - Formal description of the architecture and design of the system
- **Essential** for software development process
  - Makes communication between collaborators easier
  - Often used for future references
  - Ensures the entire team has a clear understanding of the system's design
    - Beneficial when onboarding new team members

# Software Design Document (SDD)

- What's typically inside?
  - **Architectural design**
    - Modules and interactions (diagram)
  - For each module
    - **Process design** (diagrams)
    - **Data design** (E/R diagram)
    - **Object-oriented design** (class diagram)

# Live Demo

Software Design Document

# Software Construction

## Implementation, Unit Testing, Debugging, Integration

# Software Construction

- During the **construction phase** developers **build the software**
  - Sometimes it is called "**implementation phase**"
- Software construction **includes**
  - **Method design**
  - **Writing the source code**
  - **Testing and debugging**
  - **Writing the unit tests**
  - **Code reviews** and inspections
  - **Integration** of classes / modules

# Writing the Code

- **Coding**
  - The process of **writing** the programming code (the source code), **running** and **debugging** it
  - The code **adheres to established architecture and design**
  - Developers perform **method design** as part of **coding**
- The **source code** is the output of the **software construction process**
  - Written by developers
  - Can include **tests** (unit, integration, others)

# Testing the Code

- **Testing**
  - The process of checking whether the developed software **conforms** to the **requirements**
  - Aims to **identify defects** (bugs)
- **System testing** is done by the **QA engineers**
  - Unit / integration testing is done by **developers**
- **Developers test the code** after writing it
  - Run it at least once to see the results
  - **Unit testing** / **integration testing** work better
    - Automated tests are **repeatable**, executed many times

# Debugging

- **Debugging**
  - Finding the **source** of an already identified **bug** and **fixing** it
  - Performed by developers (constantly)
- **Steps** in debugging
  - **Find the defect** in the code
    - **Identify the source** of the problem
    - **Identify the exact place** in the code causing it
  - **Fix the defect** (modify the source code)
  - **Test** to check if the fix is working correctly
  - Write a **unit test** for the defect to avoid it reoccurring further

# Software Verification Activities

- **Inspections**: requirements / design / code inspections
  - Aim to find flaws early, e.g., in the requirements
  - Performed by an experienced dev / QA engineer
- **Code reviews**: developer reviews the code
  - Find flaws / bugs / improve quality
- **Static analysis**: software tools scan the source code for problems
  - Security, code quality, potential bugs, etc.
- **Dynamic analysis**: tools asses runtime code to find flaws
  - Memory issues, bad performance, arithmetic overflows, etc.

# Code Reviews

- **Code reviews**
  - **Assessments** of the source code and other assets
  - Developer A **reviews the code** written by Developer B
  - Goals
    - Identify **bugs**
    - Increase **code quality**
    - Apply **best practices**
    - Help developers learn the source code and from other developers

# Code Reviews

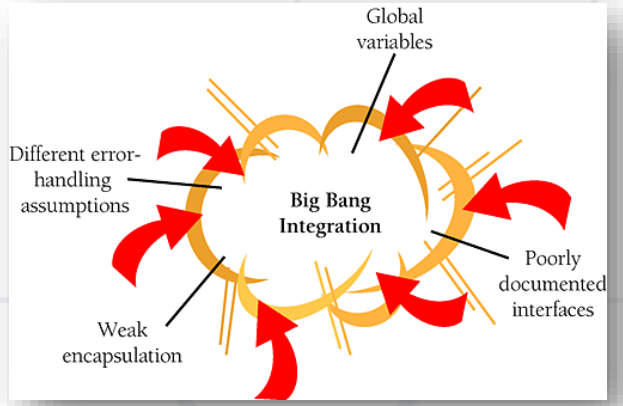
- Code reviews can be
  - **Formal** or **informal**
  - **Live** or **offline**
- Code review **tools** are used to perform code reviews online
  - Example: GitHub Pull Requests
  - Code under review gets improvement suggestions and comments
  - Code under review can finally be rejected or approved

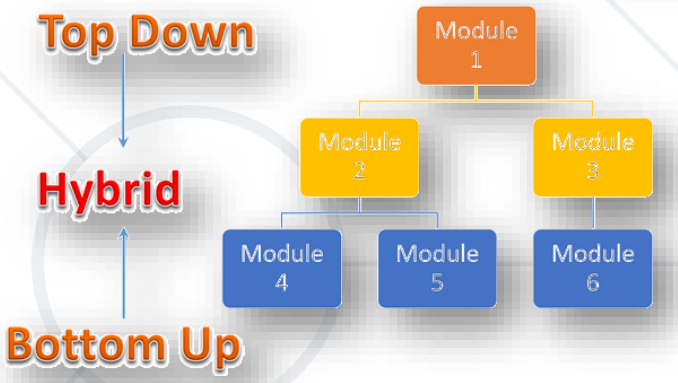
# Software Integration

- **Software integration**
  - **Combining** multiple software systems to work together seamlessly and efficiently
    - Enabling data exchange between them
  - **Enhances** overall productivity
    - Reduces redundancy
    - Improves efficiency
  - Compile, run and deploy separate modules as a single system
  - **Test to identify defects** (integration testing)

# Software Integration Strategies

- **Big bang**
  - All or almost all modules are integrated together at the same time
- **Top-down**
  - Integration of the higher-level layers first
- **Bottom-up**
  - Lower layers first
- **Continuous integration** (CI)
  - Integrate after each commit in the source control system



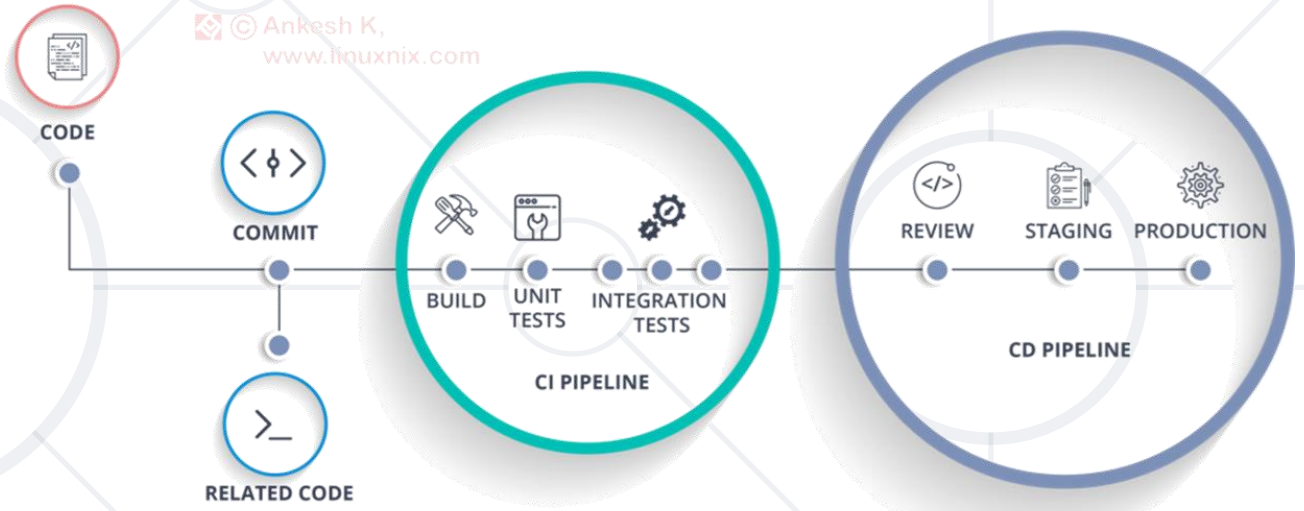




# Continuous Integration (CI)

- **Continuous integration** (CI)
  - **Integrating** the code from different developers frequently (several times a day)
  - **Automated building** and **testing** the software
    - Typically, **at Git push** in a certain branch
  - **Finding integration problems** and bugs early
    - Continuously maintain software quality
- CI is implemented by a **CI system** like Jenkins, GitHub Actions, TeamCity, Azure Pipelines

# CI/CD Pipeline

- **CI/CD pipeline**
  - Continuously **integrate** and **release** new features
- **Continuous integration (CI)**
  - Write code, test and **integrate** it in the product
- **Continuous delivery (CD)**
  - Continuously **release** new features
- Often **QA engineers** maintain and monitor the CI/CD pipeline



# Software Quality Assurance (QA)

Software Quality, Testing, QA Process

# Software Quality Assurance (QA)

- What is "**software quality assurance**" (**SQA**)?
  - Software quality assurance aims to
    - **Minimize software defects**
    - **Ensure** it behaves as **expected**
  - Defects are reported and tracked through a **bug tracking system**
  - Performed by the Quality Assurance engineers (**QA engineers**)
- Continuous integration and delivery (**CI/CD pipeline**)

# Quality Assurance (QA) Engineers

- **QA engineers** ensure the **software quality**
- Plan and execute **testing activities**
  - **Test** the software, its functionality, UX, etc.
  - Create **test plans**, design **test cases**, **execute tests**
  - Develop and execute **test automation** scripts
- **Report** and **track bugs** and their lifecycle
  - Perform **regression testing** when bugs are resolved
- Track the **development process** and its quality
  - Review the **requirements**, **design** and **code**
  - Build and monitor **CI/CD pipeline**, track QA **metrics**

# Bugs and Bug Tracking

- **Defects** (**bugs**) in software are problems in the source code / requirements / design, which cause incorrect behavior
- Once found (typically by the QA), bugs are tracked in a **bug trackers** / **issue tracking software**
  - Examples: Jira, GitHub Issues
- Bugs have a **lifecycle**
  - new → assigned / rejected → fixed → closed / reopened

# Test Plan, Test Scenarios and Test Cases

Software University

- **Test plan**
  - A guiding document outlining the testing approach, environments, schedule, and acceptance criteria to ensure software meets quality requirements
- **Test Scenarios**
  - High-level descriptions/stories representing the functionality or feature to be tested in the software
- **Test Cases**
  - Detailed, step-by-step instructions designed to validate specific conditions or functions of the software based on the associated test scenarios

# Manual and Automation Testing

- Most of the QA work is **software testing**
  - **Manual** testing
    - Fill forms
    - Click
    - Check the results
  - **Automated** testing
    - QA automation
    - Test cases as code

# Deployment and Maintenance

## Deployment Environments, Maintenance Process

# Software Deployment

- What is **software deployment**?
  - Getting software out of the hands of the developers into the hands of the users
  - Compile, package, install, configure and run the software into the customer environment / ship the app to the customer
    - Web app → deploy the new version to the Web servers
    - Mobile app → publish a new version to app stores
    - Desktop app → release a new version installer to the customers
- **Continuous deployment** (**CD**)
  - Automatically deploy the software after each commit → ensure the changes are deployable

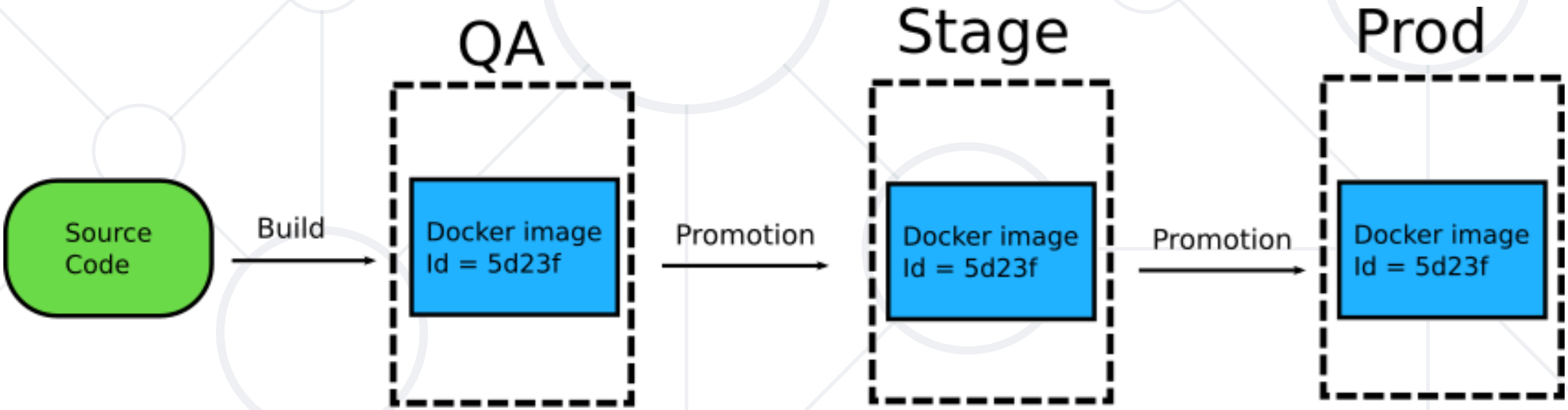
# Deployment Environments

- **Dev** environment
  - Used by developers
  - No client data
- **Test** environment
  - Used by QA engineers
  - No client data
- **Staging** environment
  - Used by QA engineers and/or clients for UAT
  - Limited production data
- **Production** environment
  - Used by clients
  - Full Production data

# Container-Based Deployment Environments

Software University

- Containers and clouds simplify creating, configuring and running the required environments



# Software Maintenance

- What is **software maintenance**?
  - The process of changing a system after it has been released
- **Reasons** for maintenance changes
  - Fixing bugs and patching security vulnerabilities
  - Changing business needs: new features and requirements
  - Adapting to new environments: hardware / platforms / software
- Typical **change process**
  - Analysis → requirements → issue backlog → prioritization
  - For each fix: design / re-engineering → code → QA → deploy

# Software Documentation

- **Documentation** makes software **maintainable**
  - Allow new people to join the development team over the time
- **Best practices** in software documentation
  - **Visualize** the development process (use a project board)
  - **Written requirements** (in a requirements tracking system)
  - **Design** and **architecture** documents (e.g., project Wiki)
  - **Code documentation** (comments and built-in docs in the code)
  - **Test management system** and **CI system** (with a dashboard)
  - User **documentation**, installation guide, quick start guide, etc.

# Summary

- **Software engineering** provides knowledge, processes, practices and tools for all the phases in software development lifecycle (SDLC)
- **Software requirements** describe the functionalities of the software
- **Software design and architecture** describe system structure – modules and interactions
- **Software constructions** involves coding, debugging, unit testing, code reviews and integration (including CI)
- The **QA engineering** process ensures the software works as intended, mostly through testing (manual and automated)
- **Software deployment** ships the software to customers and goes through different deployment environments

# Questions?

SoftUni

Software University | SoftUni Creative | SoftUni Digital | SoftUni Foundation | SoftUni Kids | СофтУни БУДИТЕЛ | Finance Academy

# SoftUni Diamond Partners

Software University

SmartIT

Coca-Cola HBC Bulgaria

SUPER HOSTING .BG

createX

Postbank

top EMPLOYER България Bulgaria 2024 FOR A BETTER WORLD OF WORK

DRAFT KINGS

THE CROWN IS YOURS

VIVACOM

INDEAVR

Serving the high achievers

AMBITIONED

PHAR VISION

# Trainings @ Software University (SoftUni)

- Software University – High-Quality Education, Profession and Job for Software Developers
  - softuni.bg, about.softuni.bg
- Software University Foundation
  - softuni.foundation
- Software University @ Facebook
  - facebook.com/SoftwareUniversity

# License

- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – https://about.softuni.bg/
- © Software University – https://softuni.bg