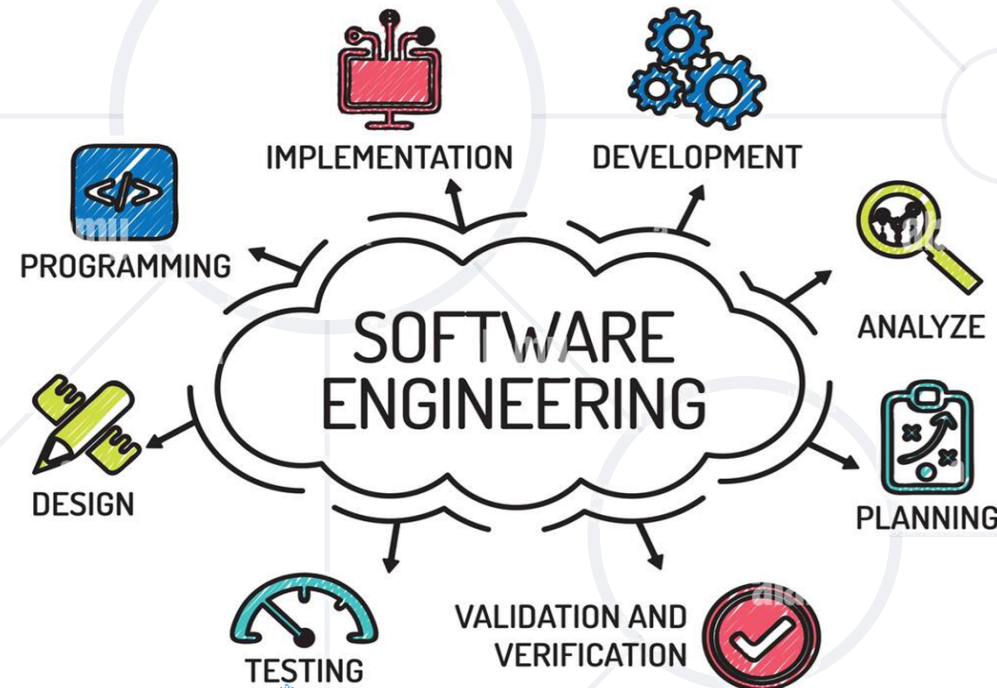


Software Engineering Fundamentals

Software Development Concepts, Phases and Practices



SoftUni Team
Technical Trainers



SoftUni



Software University

<https://softuni.bg>

Have a Question?

sli.do

#Dev-Ops

1. Software Development Lifecycle
2. Software Requirements
3. Software Architecture and Design
4. Software Construction
5. Software Quality Assurance
6. Deployment and Maintenance





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** 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** 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** 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

Software Requirements Specification v1.2

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

DOCUMENT CONTROL	II
APPROVAL	II
DOCUMENT CHANGE CONTROL	II
DISTRIBUTION LIST	II
CHANGE SUMMARY	II
1. INTRODUCTION	5
1.1 PURPOSE AND INTENDED AUDIENCE	5
1.2 SCOPE OF PRODUCT	5
1.3 DEFINITIONS, ACRONYMS, AND ABBREVIATIONS	5
1.3.1 Definitions	5
1.3.2 Acronyms	8
1.3.3 Abbreviations	9
1.3.4 Overview	9
1.4 REFERENCES	10
2. GENERAL DESCRIPTION	12
2.1 PRODUCT PERSPECTIVE	13
2.2 PRODUCT FEATURES	14
2.2.1 Use Cases	14
Use Case 1: Log In/Out	15
Use Case 2: Run External Program	17
Use Case 3: Manage Data	18
Use Case 4: Query/Output Data	22
Use Case 5: Archive Data	23
2.3 USER CHARACTERISTICS	24
2.4 GENERAL CONSTRAINTS	25
2.5 ASSUMPTIONS AND DEPENDENCIES	25
3. SPECIFIC REQUIREMENTS	26
3.1 EXTERNAL INTERFACE REQUIREMENTS	26
3.1.1 General Characteristics	26
3.1.2 Menu Options	26
3.1.3 Display Screen and Windows	26
3.1.3.1 Common Screens and Windows	27
3.1.3.2 Data Entry Users	27
3.1.3.3 Data Administrators	27
3.1.4 Hardware Interfaces	27
3.1.5 Software Interfaces	27
3.1.6 Communications Interfaces	28
3.2 BEHAVIORAL REQUIREMENTS	28
3.2.1 Same Class of User	28
3.2.1.1 Data Entry User	28
3.2.1.2 Data Administrator	29
3.2.2 Related Real-world Objects	29
3.2.2.1 External Programs	29
3.2.2.2 Fields	29
3.2.2.3 Season	29

Software Requirements Specification	CS4311 Fall 2003	Date	Page
		9/4/2002 5:55 PM	iii

Software Requirements Specification v1.2

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."

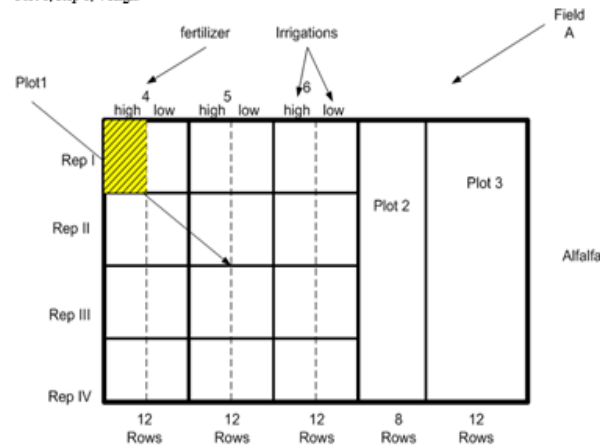


Figure 1 Field Layout

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.

Software Requirements Specification	CS4311 Fall 2003	Date	Page
		9/4/2002 5:55 PM	13

Software Requirements Specification v1.2

- 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.

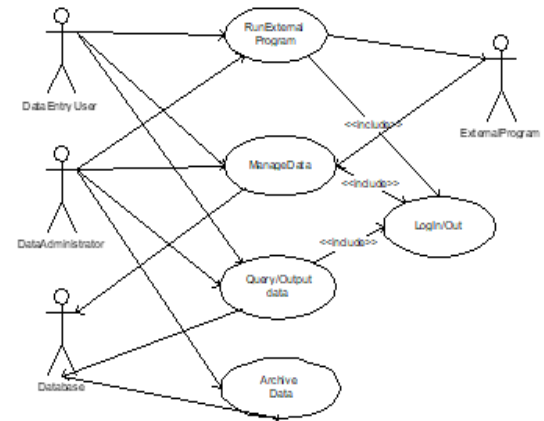


Figure 2 Use Case Diagram

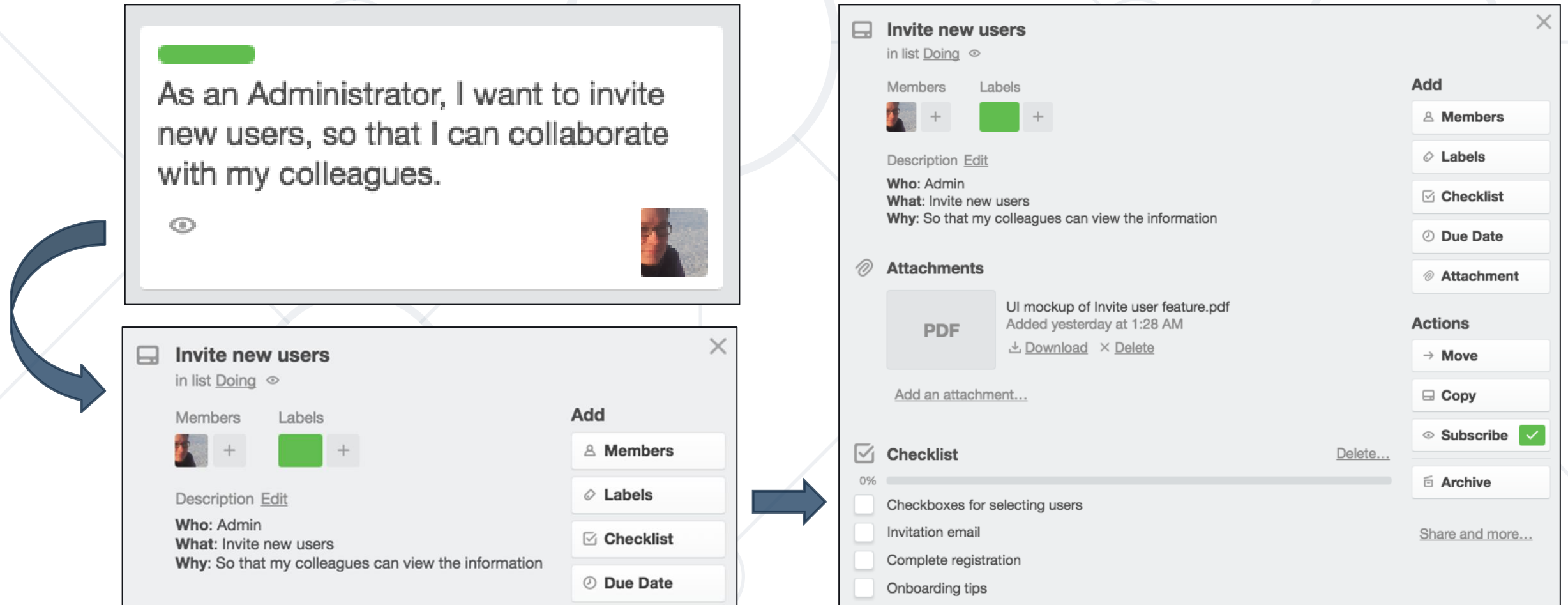
2.2.1 Use Cases

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

Software Requirements Specification	CS4311 Fall 2003	Date	Page
		9/4/2002 5:55 PM	14

User Story – Example

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



- 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



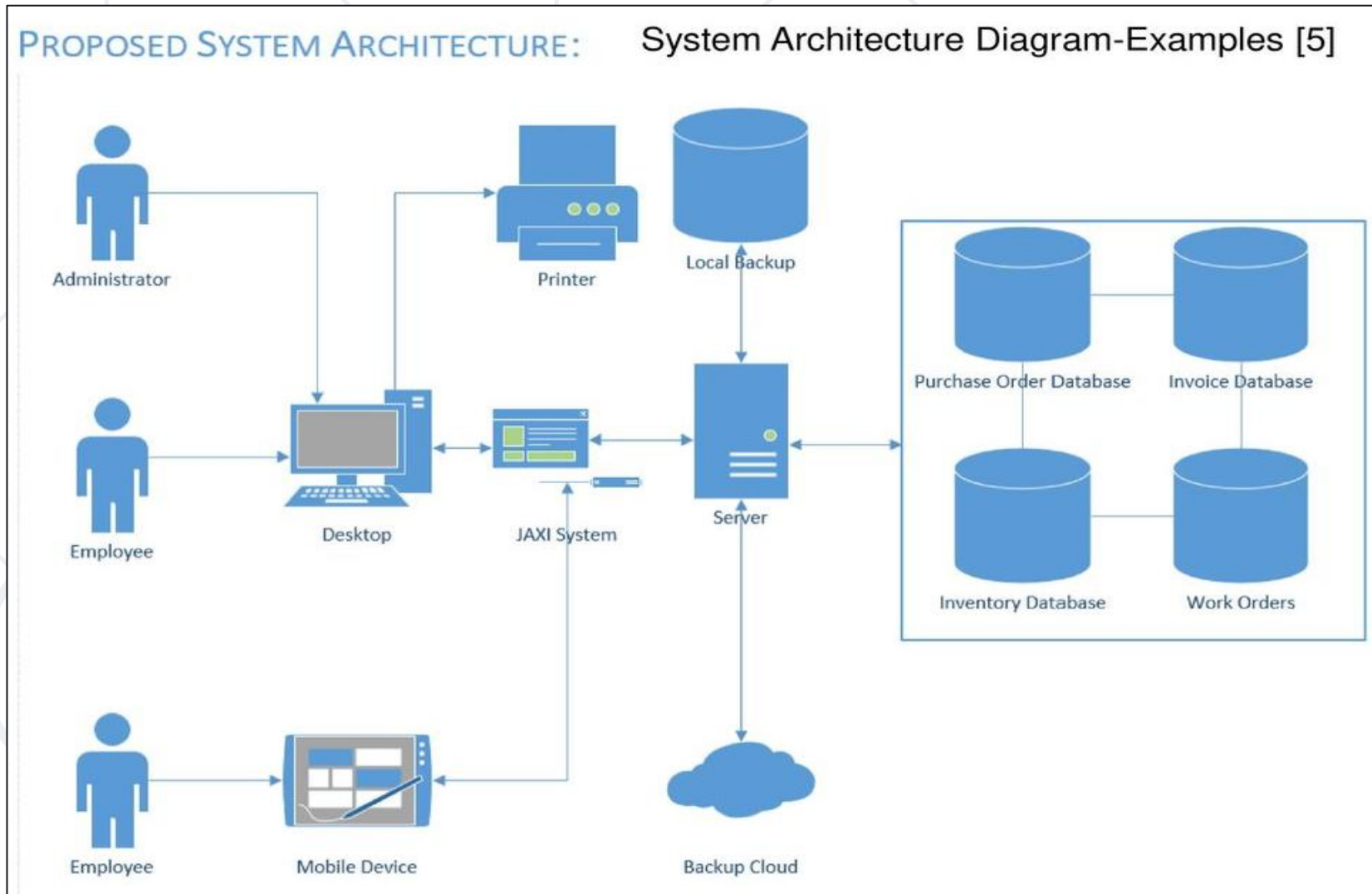
Software Architecture and Design

- **Software architecture and 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

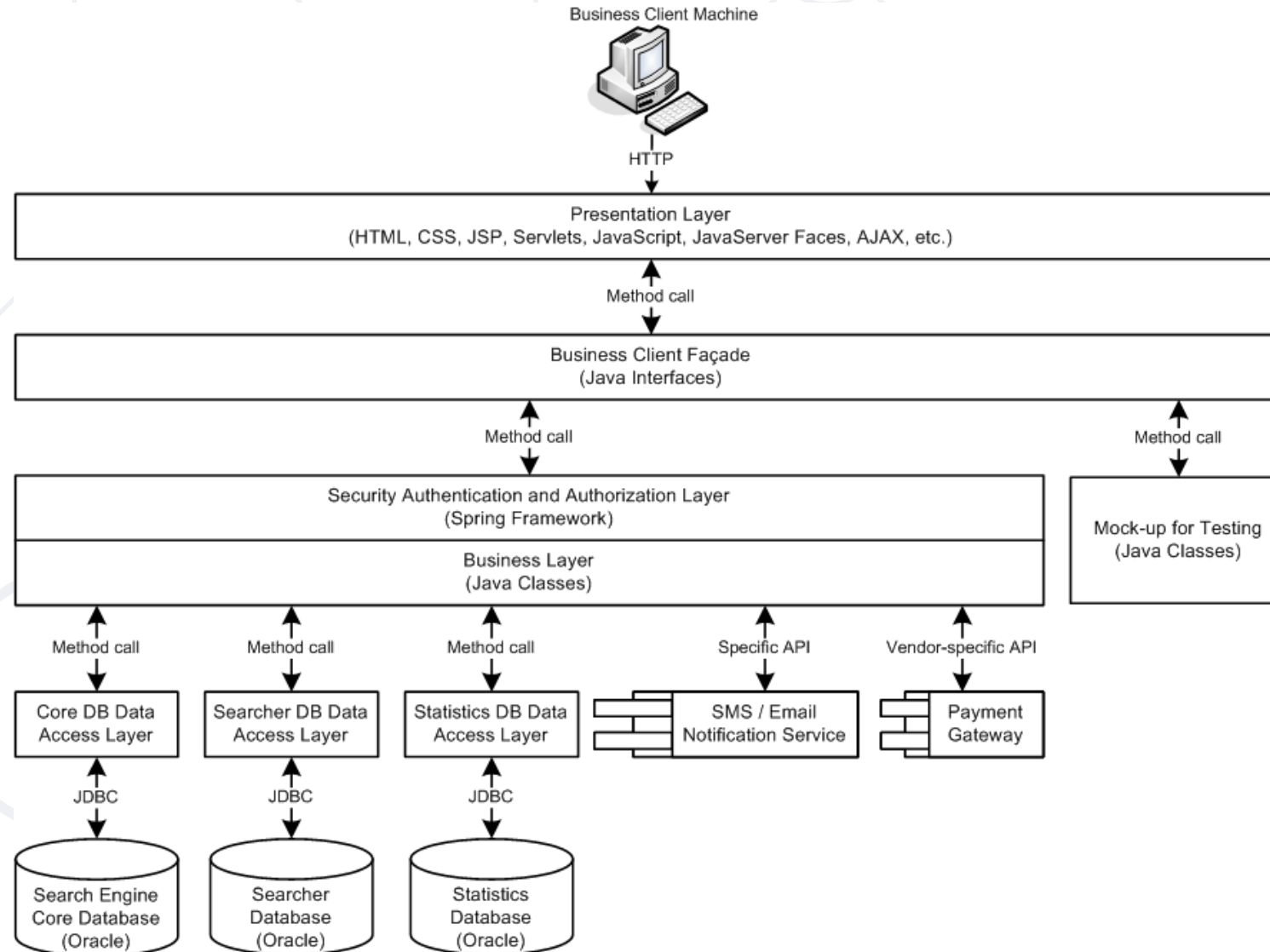
- The **software design** describes the system design at a lower-level
 - Functions of individual modules, classes, etc.
- **Design patterns** in software design
 - **Proven** solutions to **recurring** problems
 - Fundamental in software engineering

- 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

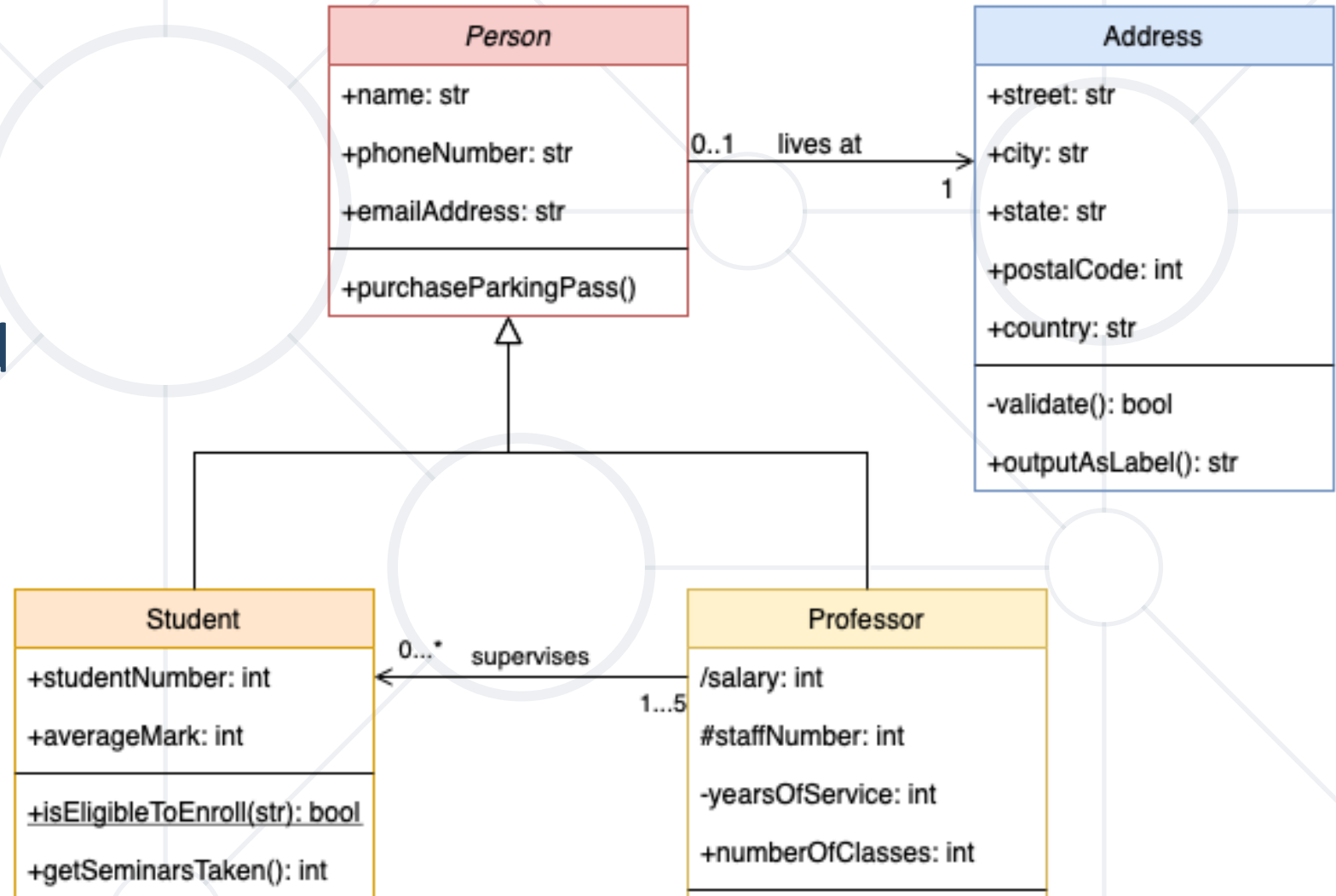


Software Architecture Diagram

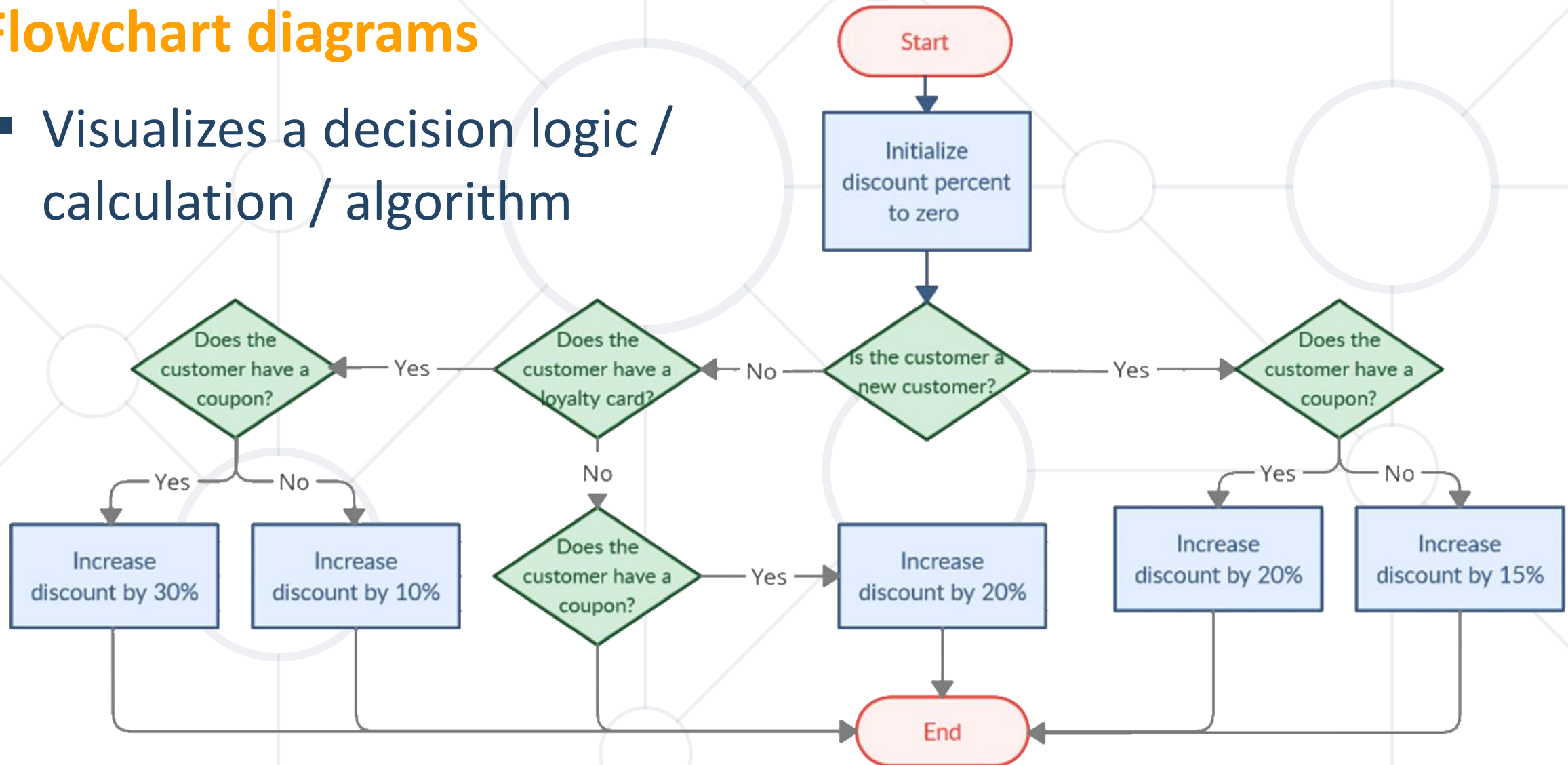


- **Detailed design**
 - Describes the internal module structure
 - Subcomponents, interfaces, process design, data design
- **Object-oriented design**
 - Describes the classes, their responsibilities, relationships, dependencies, and interactions (usually in UML)
- **Internal class design**
 - Methods,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

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



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



- 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

- 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

- 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

■ 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 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**
 - 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

- **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 assess runtime code to find flaws
 - Memory issues, bad performance, arithmetic overflows, etc.

- **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 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**
 - **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**

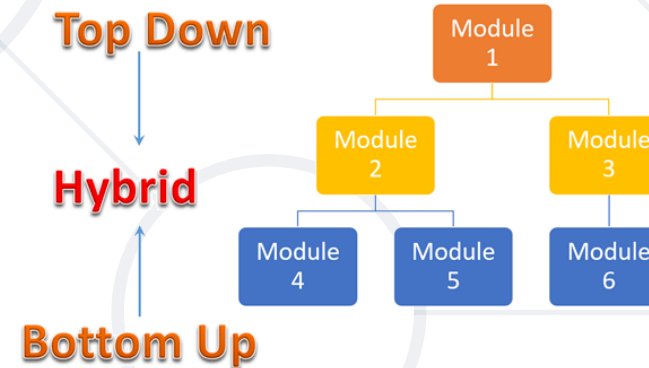
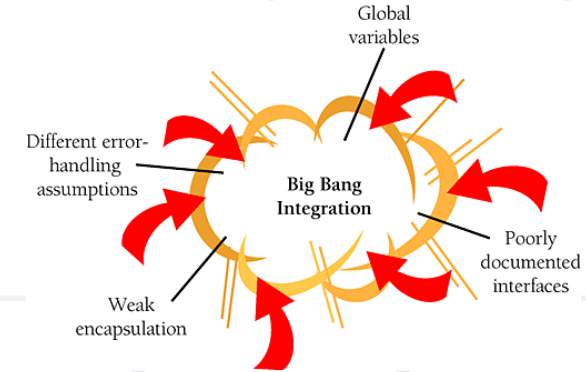
- Integration of the higher-level layers first

- **Bottom-up**

- Lower layers first

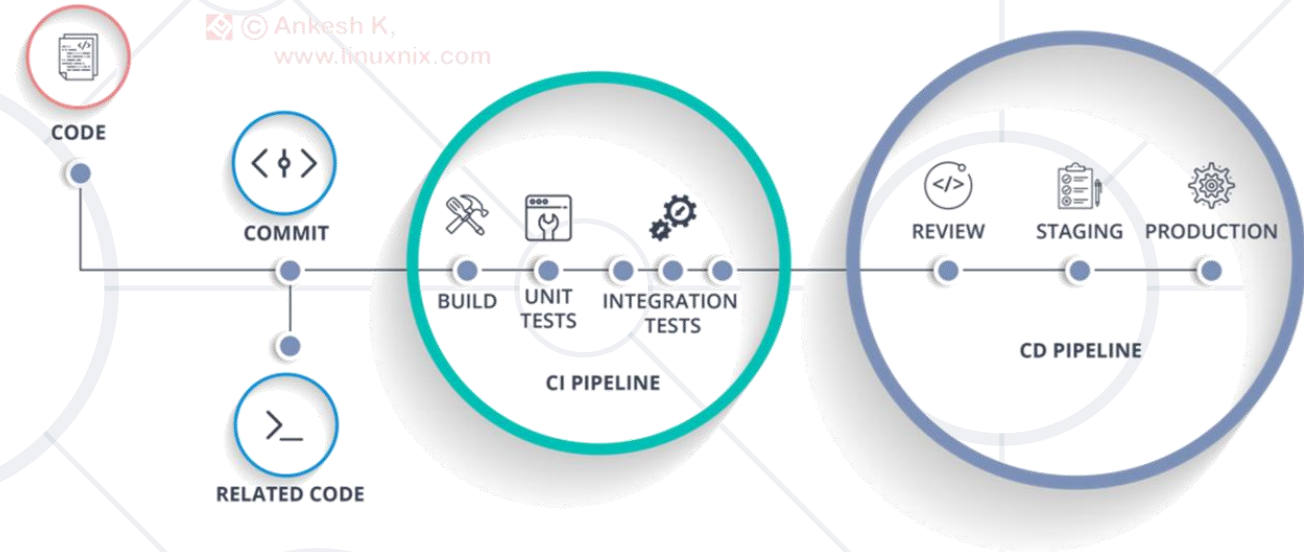
- **Continuous integration (CI)**

- Integrate after each commit in the source control system



- **Continuous integration** (CI)
 - **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**
 - 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

- 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**

- **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**

- 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

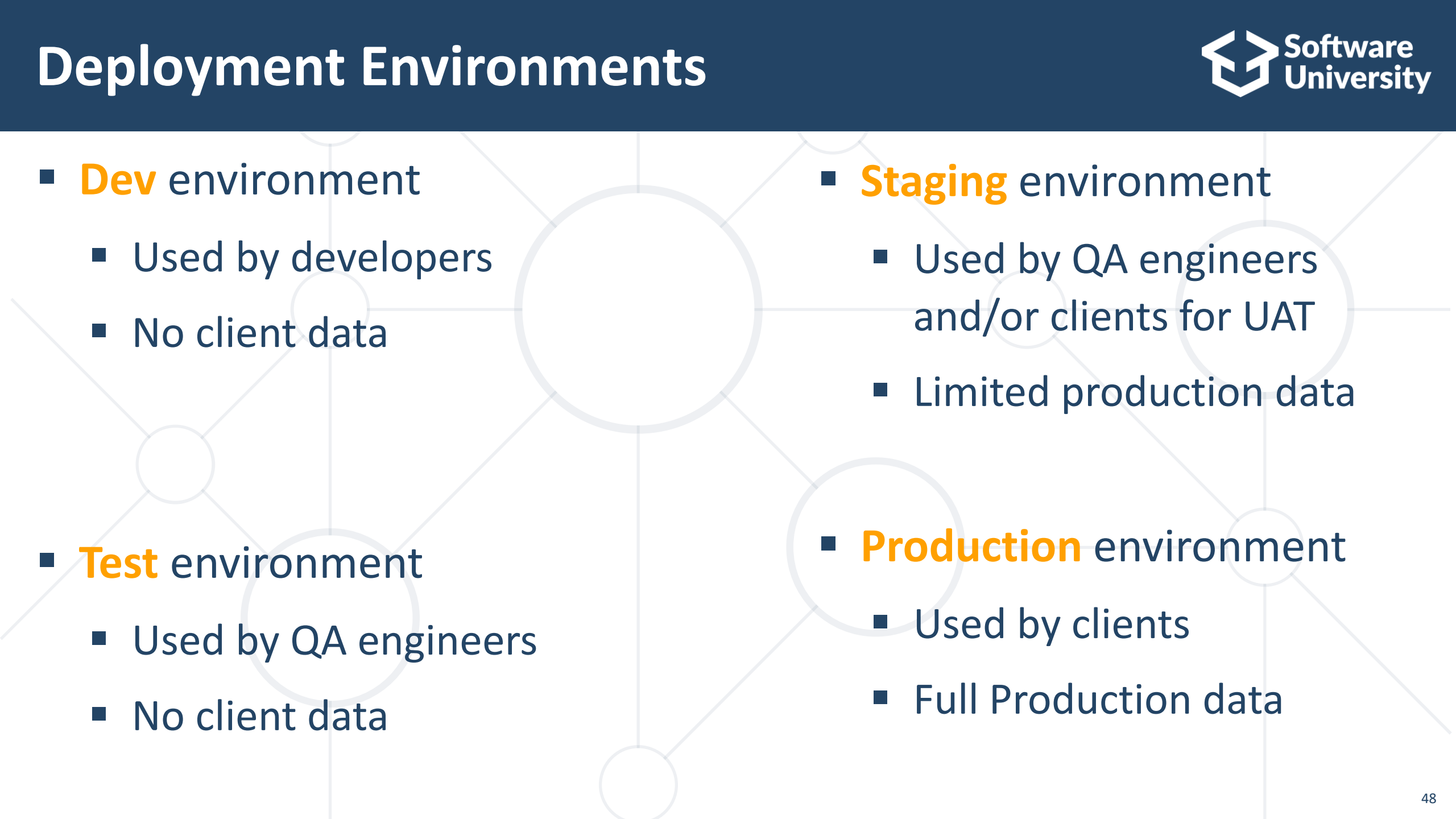
- Most of the QA work is **software testing**
 - **Manual** testing
 - Fill forms
 - Click
 - Check the results
 - **Automated** testing
 - QA automation
 - Test cases as code



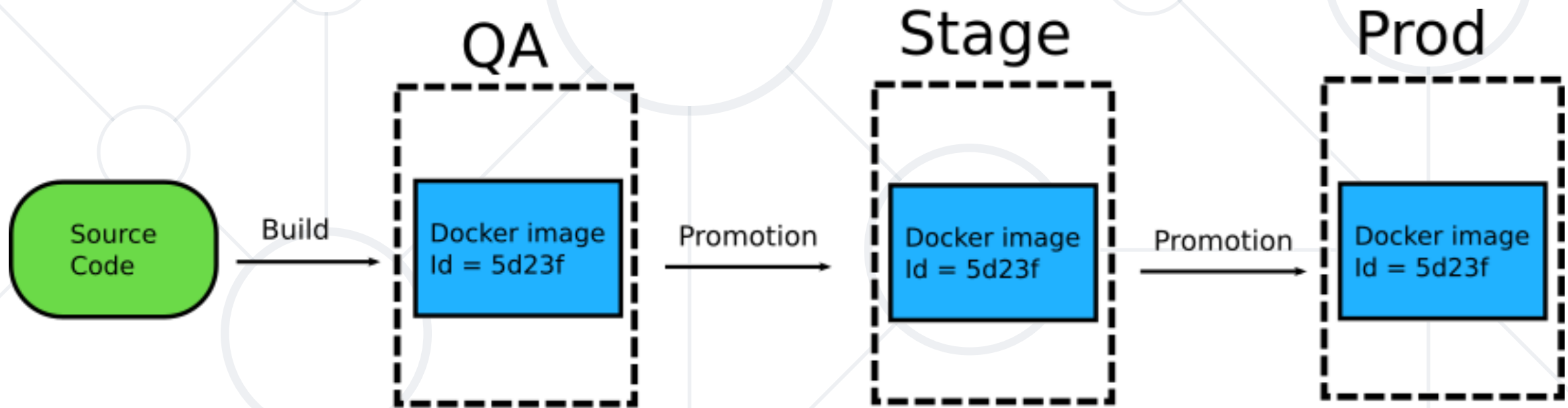
Deployment and Maintenance

Deployment Environments, Maintenance Process

- 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

- 
- **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

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



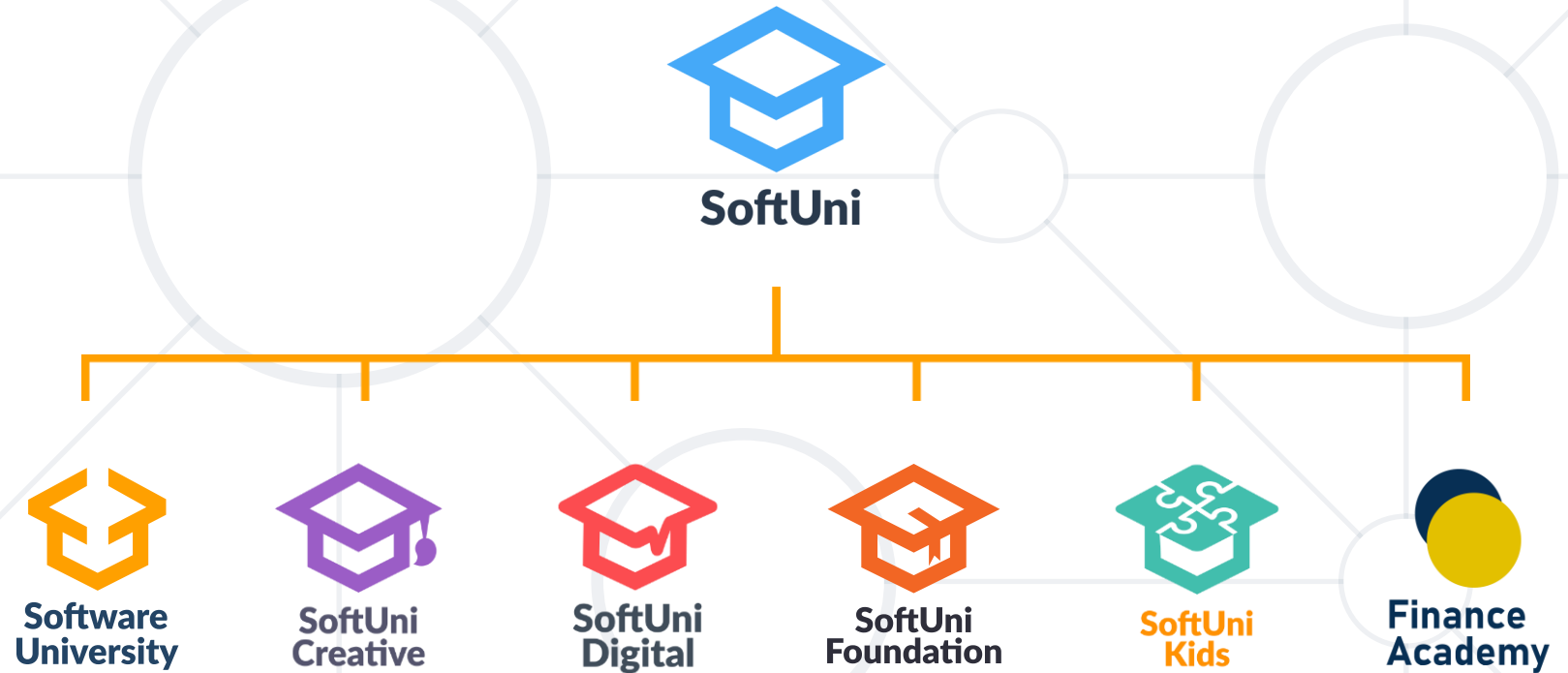
- What is **software maintenance**?
 - The process of changing a system after it has been released
- **Reasons** for maintenance changes
 - Fixing bugs and patching security vulnerabilities
 - Changing business needs: new features and requirements
 - Adapting to new environments: hardware / platforms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

- **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.

- **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 Diamond Partners

**SUPER
HOSTING
.BG**



**Coca-Cola HBC
Bulgaria**

 **Flutter**TM
International

INDEAVR
Serving the high achievers



AMBITIONED

 **DRAFT
KINGS**

 **SOFTWARE
GROUP**



BOSCH

 **Postbank**
Решения за твоето утре

 **PHAR
VISION**



SmartIT

DXC
TECHNOLOGY

createX

- 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>



- 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

- Software University Forums

- forum.softuni.bg



Software University

