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

1. Software Development Lifecycle
2. Software Development Phases
   - Requirements Analysis
   - Architecture and Design
   - Construction
   - Testing and QA
   - 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

- **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** tasks
- Methodologies manage the **development process**

Software Requirements Analysis

Requirements, User Stories, Prototypes, Specification

Software Requirements

- **Software requirements** describe the functionality of the software: what shall it do?
  - Answer the question "**what?**", not "**how?**"
  - 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 usually don't know what exactly they need!
  - 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

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

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

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

User Stories, UI Prototypes and SRS

Live Demo

Software Architecture and Design

Software Architecture and Software Design

- **Software architecture and design** == technical descriptions (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
- The **software design** describes the system design at a lower-level
  - Functions of individual modules, classes, etc.

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**

Example: System Architecture Diagram

Example: 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

Example: UML Class Diagram

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

Example: Flowchart Diagram

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

Software Design Document (SDD)

- The **Software Design Document** (SDD)
  - Formal description of the architecture and design of the system
- 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)

Software Design Document

Live Demo

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**
  - **Internal method design**
  - **Writing the source code**
  - **Testing and debugging**
  - **Writing the unit tests**
  - **Code reviews** of 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 **follows the architecture and design**
  - Developers perform **internal 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** checks whether the developed software **conforms** to the **requirements**
  - Aims to **identify defects** (bugs)
- **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
- **System testing** is done by the **QA engineers**
  - Unit / integration testing is done by **developers**

Code with Unit Tests – Example

- Example of simple C# class with **unit tests**

```
public class CircularQueue<T>
{
    private T[] elemen

    20 references | 8/8 passing
    public int Count {
    7 references | 6/6 passing
    public int Capacit
    11 references | 4/4 passing
    public int StartIn
    9 references | 4/4 passing
    public int EndInde

    private const int

    6 references | 5/5 passing
    public CircularQue

    36 references | 14/14 passing
    public void Enqueu

    1 reference
    private void Grow()...
```

```
[Test]
0 references
public void Test_Method_Enqueue_WithGrow
{
    queue.Enqueue(10);
    queue.Enqueue(20);
    queue.Enqueue(30);
    queue.Enqueue(40);
    queue.Enqueue(50);
    queue.Enqueue(60);
    queue.Enqueue(70);
    queue.Enqueue(80);
    queue.Enqueue(90);

    Assert.AreEqual(9, queue.Count, "Que
    Assert.AreEqual(16, queue.Capacity,
    Assert.AreEqual(0, queue.StartIndex,
    Assert.AreEqual(9, queue.EndIndex, "
}
```

CircularQueueTests (18)	85 ms
Test_Method_ToArray (3)	1 ms
Test_Method_ToArray([],[])	1 ms
Test_Method_ToArray([10, 20, 30, 40, 50, ...],[10, 20, 30, 40, 50, ...])	< 1 ms
Test_Method_ToArray([10, 20, 30, 40, 50],[10, 20, 30, 40, 50])	< 1 ms
Test_Method_ToString (3)	< 1 ms
Test_Method_ToString([],"[]")	< 1 ms
Test_Method_ToString([10, 20, 30, 40, 50, ...],"[10, 20, 30, 40, 50, 60, 70, 80, 90]")	< 1 ms
Test_Method_ToString([10, 20, 30, 40, 50],"[10, 20, 30, 40, 50]")	< 1 ms
Test_Constructor_Default	12 ms
Test_Constructor_WithGivenCapacity	< 1 ms
Test_Method_Dequeue	< 1 ms
Test_Method_Dequeue_Empty	4 ms
Test_Method_Enqueue	< 1 ms
Test_Method_Enqueue_WithGrow	< 1 ms
Test_Method_EnqueueDequeue_RangeCross	< 1 ms
Test_Method_Peek	< 1 ms
Test_Method_Peek_Empty	< 1 ms
Test_Method_ToArray_RangeCross	< 1 ms
Test_Multiple_Operatios	37 ms
Test_Performance_MillionItems	31 ms

Code with Integration Tests – Example

- Example of JavaScript MVC app with **integration tests**
  https://github.com/nakov/MVC-app-integration-tests-example-mocha

```javascript
const assert = require('assert');
const fetch = require('node-fetch');

suite('View Students page', function() {
  test('Page title', async function() {
    let res = await fetch("http://localhost:8888/students");
    let body = await res.text();
    assert.ok(body.includes("<h1>Registered Students</h1>"));
  });

  test('Students list', async function() {
    let res = await fetch("http://localhost:8888/students");
    let body = await res.text();
    assert.ok(body.includes("<ul><li>Steve (steve@gmail.com)</li><li>Tina (tina@yahoo.com)</li></ul>"));
  });
});
```

```
>npm test

> test
> mocha tests/*.js



  About page
    ✓ Page title (286ms)

  Add Students page
    ✓ Page title
    ✓ Students HTML form
    ✓ Add valid student
    ✓ Add invalid student

  Home page
    ✓ Page title
    ✓ Students count

  View Students page
    ✓ Page title
    ✓ Students list


  9 passing (562ms)
```

Debugging

- **Debugging** == find the **source** of an already identified **bug** and **fix** 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
- **Code reviews**: devs 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 analyze the code at runtime to find flaws
  - Memory issues, bad performance, arithmetic overflows, etc.

Static Analysis – Examples

- ESLint example: https://eslint.org/play
- CodeQL example: https://github.com/nakov/code-scanning-javascript-demo/security/code-scanning

Code Reviews

- **Code reviews** == **assessments** of the source code and other assets
  - Developer A **reviews the code**, written by his colleague Developer B
  - Goals: identify **bugs**, increase **code quality**, apply **best practices**, help developers learn the source code, learn 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

Git Pull Requests and The Code Review Process

Code Reviews in GitHub – Example

- Example from GitHub: https://github.com/angular/angular/pull/47310

github.com/angular/angular/pull/47310

Open **build(bazel): run aio a11y tests under bazel** #47310

kormide wants to merge 1 commit into `angular:aio-bazel-migration` from `aspect-build:aio-bazel-a11y`

kormide reviewed 3 days ago — View changes

aio/scripts/audit-web-app.mjs — Outdated

```
...  ...   @@ -57,7 +58,7 @@ async function _main(args) {
57   58      try {
58   59        console.log('');
59   60        const startTime = Date.now();
60        -   const browser = await puppeteer.launch();
     61   +   const browser = await puppeteer.launch({executablePath: path.resolve(process.env.CHROME_E
```

kormide 3 days ago — Contributor — Author

Would any of these flags be detrimental to the audit tests? They all still seem to pass..

kormide 3 days ago • edited — Contributor — Author

I took these flags from the ones we pass for karma and protractor tests, but since we weren't passing anything here originally I'll try to narrow this down to the minimal required set.

devversion 2 days ago — Member

SGTM. Usually in other Chrome integration Bazel sandbox tests we just have headless and no-sandbox (to avoid nested sandboxing)

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
- **QAs** maintain and monitor the CI/CD pipeline

Continuous Integration

A Real-World Live Demo

https://github.com/nakov/Eventures/actions

Software Quality Assurance (QA)

Software Quality, Testing, QA Process

Software Quality Assurance (QA)

- What is "**software quality assurance**" (**SQA**)?
  - Software quality assurance aims to **assure** that the **software** is **bug free** (behaves as expected)
  - Defects are reported and tracked through a **bug tracking system**
  - Performed by the Quality Assurance engineers (**QA engineers**)
- Most of the QA work is **software testing**
  - **Manual** testing (fill forms, click and check the results)
  - **Automated** testing (QA automation, test cases as code)
- 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

Issue Tracking Tools

Live Demo

https://github.com/twbs/bootstrap/issues , https://github.com/twbs/bootstrap/issues/31392 , https://daosio.atlassian.net/jira/software/c/projects/CART/issues

The Software Testing Process

- **Testing** checks whether the developed software **conforms** to the **requirements**
- Testing aims to **find & report defects** (bugs)
- Software testing process includes
  - **Test planning**: what, when, how to test?
  - **Test design**: test scenarios & test cases
  - Setup **test environment**: install, configure, prepare test data, …
  - **Test execution**: perform the tests
  - **Test reporting**: log the test results and bugs found

Test Scenario

- What is a "**test scenario**"?
  - Any **functionality** / **feature** / **user story** that can be tested
  - Also called **story under test** or **feature under test**

- **Test scenario 1**: Registration

- **Test scenario 2**: Enroll in a course

Test Scenarios and Test Cases

- One **test scenario** typically includes **several test cases**
- Example
  - User story: *Users should be able to login*
  - Test scenario: *Login with user + password*
  - Test cases
    - Login with **valid** user + pass → success
    - Login with **invalid** user + pass → error

Test Case

- Sequence of **steps** to check the **correct** behavior
- At **least two cases** to fully test certain scenario
  - A positive test
  - A negative test
- Test cases consist of
  - Title (+ optional description)
  - Steps to follow
  - Expected result

Test Case – Structure and Example

Title: *<test title>*

Description: *<short overview>*

Steps: *<the testing procedure>*

1. …
2. …
3. …

Expected results:

- …
- …
- …

Title: *Brew a small coffee*

Description: *start the coffee machine, fill water and ground coffee, and brew a cup of coffee.*

Steps:

1. Power on the machine.
2. Put ground coffee blend in the coffee outlet.
3. Fill the water container to its max level.
4. …

Expected results:

- The brew process should complete in < 50 seconds.
- The coffee cup should hold a hot small coffee (60 ml).
- …

Test Scenarios and Test Cases – Example

- Sample **test scenario**
  - Register to the platform
- **Test cases** for this scenario
  - Non-existing username → success
  - Duplicated username → error
  - Empty username or password → error
  - Too long username → error
  - Invalid characters in username / password → error

Test Plan, Test Scenarios and Test Cases

- The **test plan** describes how tests will be performed
  - List of QA and test activities to be performed to ensure **meeting the quality requirements** (more or less formal)
  - Features to be tested (scenarios), test cases, testing approach, testing environments, test schedule, acceptance criteria
- Test scenarios and test cases
  - **Test scenarios** – stories to be tested
  - **Test cases** – tests of single function
- **Formal test plans** are rarely used in modern projects

Test Plans and Test Cases

Live Demo

Manual and Automation Testing

- **Manual** testing
  - Type of software testing in which tests are **executed manually without** using any **automated tools**
  - A human performs the tests **step by step**, without test scripts
  - Tests are executed **one-by-one** in an individual manner
- **Automation** testing
  - Type of software testing in which tests are **executed automatically** via test **automation frameworks**
  - Testers utilize **tools** and **scripts** to automate testing efforts
  - Requires **coding** and test **maintenance**

Automated Testing

Live Demo

https://github.com/nakov/ , MVC-app-integration-tests-example-mocha , https://github.com/nakov/Eventures

Deployment and Maintenance

Deployment Environments, Maintenance Process

Software Deployment

- What is **software deployment**?
  - **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 server
  - Mobile app → publish a new version in the app store
  - 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

- Several **environments** for deployment
  - **Dev environment** → for developers
  - **Test environment** → for QAs
  - **Staging environment** → for QAs and clients
  - **Production environment** → for the end user

Container-Based Deployment Environments

- Containers, cloud, Docker, Kubernetes, etc. 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** describes system structure: **modules** and **interactions**
- **Software constructions** involves coding, debugging, unit testing, code reviews and integration (including continuous integration)
- The **QA engineering** process ensures the software works as intended (it is bug free), mostly through testing (manual and automated)
- **Software deployment** ships the software to customers: dev env → test environment → staging environment → production environment

Questions?

© SoftUni – https://about.softuni.bg. Copyrighted document. Unauthorized copy, reproduction or use is not permitted.

SoftUni Diamond Partners

Software University

SUPER HOSTING .BG

Coca-Cola HBC Bulgaria

POKERSTARS
POKER | CASINO | SPORTS
a Flutter International brand

INDEAVR
Serving the high achievers

AMBITIONED

DRAFT KINGS

SOFTWARE GROUP

createX

Postbank
Решения за твоето утре

BOSCH

DXC TECHNOLOGY

SmartIT

Educational Partners

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

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
- Software University Forums
  - forum.softuni.bg