HW5 Report: Quality Software Engineering Documentation

Student: Navleen Singh **Email:** singhn3@nku.edu **Course:** CSC 640 - Software Engineering **Semester:** Fall 2025

Executive Summary

This report documents my learning journey through six weeks of software engineering principles, practices, and technologies. The documentation demonstrates understanding of quality software development through comprehensive coverage of development tools, processes, programming paradigms, design principles, and modern web technologies.

Key Focus Areas: - Software development tools and version control - Agile methodologies and DevOps culture - Object-oriented programming and design principles - High-level programming languages evolution - Modern web development with TypeScript and React - Practical application through the TODO REST API project (HW4)

Week 1: Course Introduction & Tools Overview

Developer Tools: VSCode & Marp

Visual Studio Code (VSCode) serves as our primary integrated development environment, offering: - Lightweight yet feature-rich IDE with file explorer, text editor, version control integration, and terminal - Over 60,000+ extensions available (2025 stats) - Support for all major programming languages with IntelliSense - Most popular IDE among developers worldwide

Marp is our toolchain for creating presentations from Markdown: - Write content in Markdown format - Compile to presentation formats (PDF, PPT, HTML) - VSCode extension enables preview and easy export - Streamlines documentation workflow

These tools create an efficient development environment where VSCode handles all coding needs and Marp manages documentation and presentations.

Git & GitHub Basics

Git - Distributed version control system: - Tracks code changes locally over time - 93.87% developer adoption rate (2025, up from 87.1% in 2016) - Essential for managing code history and collaboration

GitHub - Cloud platform for Git repositories: - Provides remote storage and backup for code - Facilitates team collaboration and code sharing - Acts as a professional portfolio for projects - Central hub for accessing course materials and submitting assignments

Course Overview & Policies

The course emphasizes **high-quality software engineering practices** including: - Development processes from requirements to testing - Design principles and architectural patterns - Team collaboration and communication - Project management throughout the software lifecycle

Key Policies: - Attendance and participation are crucial - Timely submission of assignments required - Academic integrity strictly enforced - All code must be original work

Software as Storytelling

Writing code is analogous to storytelling: - Code should tell a clear, coherent story about the problem and solution - Well-structured code is readable by other developers and your future self - Documentation communicates design decisions and system purpose - Communication skills are as important as technical skills

Assignment Structure: - Early assignments focus on individual skills (tools, simple design) - Later assignments build toward team-based capstone projects - Incremental delivery reinforces lecture topics - Each assignment applies concepts in practice

---

Week 2: Software Process Overview

From Waterfall to Agile

Waterfall Model - Linear, phase-driven approach: - Sequential phases: requirements → design → implementation → testing → maintenance - Each phase must complete before the next begins - Changes are costly once a phase is finished - Testing occurs only after development is complete - Best suited for well-understood requirements unlikely to change

Agile Methods - Iterative, incremental approach: - Welcomes changing requirements even late in development - Work done in short cycles (sprints) producing working software frequently - Continuous testing and integration alongside development - 61% of organizations have used Agile for 5+ years (2025 stats) - Emphasizes adaptability and customer feedback - Delivers value faster and adjusts as needs evolve

Key Difference: Waterfall emphasizes upfront planning and heavy documentation; Agile emphasizes adaptability and working software.

Scrum Framework

Scrum - An Agile framework for iterative development: - Breaks projects into small increments completed in fixed-length sprints (1-4 weeks) - 87% of Agile organizations use Scrum (2025 stats) - Focuses on continuous improvement - Delivers potentially shippable software each sprint

Scrum Roles: - **Product Owner:** Manages backlog and priorities, represents customer interests - **Scrum Master:** Facilitates process, removes impediments, coaches team - **Development Team:** Self-organizing team that builds product increment

Scrum Events: - **Daily Stand-ups:** 15-minute daily team sync - **Sprint Planning:** Define sprint goals and tasks - **Sprint Review:** Demonstrate completed increment - **Sprint Retrospective:** Reflect and identify improvements

Scrum Artifacts: - **Product Backlog:** Prioritized list of all work items - **Sprint Backlog:** Tasks committed for current sprint - **Increment:** Deliverable software at sprint end

DevOps Culture

DevOps - Combining Development and Operations: - Bridges gap between software development and IT operations - Emphasizes collaboration between traditionally siloed teams - Culture of shared responsibility for outcomes - Automation of repetitive tasks

Continuous Integration & Delivery (CI/CD): - Automated building, testing, and deploying of code - Enables rapid, reliable releases - Code changes go live faster with fewer errors - 60% of businesses adopting GitOps for deployment automation (2025 Forrester) - AI/ML integration for predictive monitoring and anomaly detection

Key Principles: - Break down silos between development and operations - Automate manual and repetitive tasks - Continuous feedback loops - Monitor software in production - Iterate quickly without sacrificing reliability - Achieve faster delivery with improved stability and scalability

---

Week 3: Python, OOP, and UML Fundamentals

Python for Software Engineering

Python - High-level, interpreted, general-purpose programming language: - Known for readable syntax and code clarity - Design philosophy emphasizes simplicity - Supports multiple paradigms: procedural, object-oriented, functional - Dynamic typing (no variable type declarations) - Rich standard library - Cross-platform portability - Ideal for rapid prototyping and learning design principles

Object-Oriented Programming Basics

OOP Paradigm - Centered on objects that encapsulate data and behavior: - Organizes software as interacting objects - Promotes modularity and maintainability

Four Pillars of OOP:

1. **Abstraction** - Modeling real-world entities by focusing on relevant features while ignoring unnecessary details
2. **Encapsulation** - Hiding internal state and implementation behind public interfaces
3. **Inheritance** - Creating new classes by extending existing ones, enabling code reuse
4. **Polymorphism** - Treating objects of different subclasses through common interface

Benefits: - Improved code modularity - Enhanced maintainability - Better code reuse through inheritance and polymorphism - Complexity management through abstraction and encapsulation

Abstraction & Inheritance in Practice

Abstraction Example:

```python
class User:
    def __init__(self, name, email):
        self.name = name
        self.email = email

    def send_notification(self):
        # Abstract the complex email sending logic
        pass
```

Abstraction Concepts: - Identify essential attributes and model classes accordingly - Omit extraneous details not needed by the system - Focus on *what* an object does instead of *how* - Example: User class abstracts common properties (name, email) relevant to system

Inheritance Example:

```python
class Employee:
    def __init__(self, name, salary):
        self.name = name
        self.salary = salary

class Manager(Employee):  # Inherits from Employee
    def __init__(self, name, salary, department):
        super().__init__(name, salary)
        self.department = department
```

Inheritance Concepts: - Establishes "is-a" relationship between classes - Base class defines general attributes and methods - Derived class inherits and extends/overrides behavior - Promotes code reuse - **Caution:** Follow Liskov Substitution Principle - subclasses must truly satisfy parent relationship - Consider composition over inheritance when appropriate to avoid rigid hierarchies

UML Diagrams for Design

Unified Modeling Language (UML) - Standard visualization for system design: - Blueprints for understanding software structure and behavior - Communication tool without diving into code

Structure Diagrams: - **Class Diagrams:** Show classes, attributes, methods, and relationships (associations, inheritance, composition) - Design static architecture - Composition shown with filled diamond (strong "has-a" relationship)

Behavior Diagrams: - **Sequence Diagrams:** Illustrate interactions over time between objects, method call flows - **State Diagrams:** Show state transitions of objects - **Use Case Diagrams:** Depict functional requirements showing actors and interactions

Benefits: - Clear communication of designs - Common language for developer collaboration - Design clarification before coding

Software Requirements Fundamentals

Requirements - What software should do and how it should be:

Functional Requirements: - Define **what** system must accomplish - Features and behaviors - Example: "System shall send password reset email upon request"

Non-Functional Requirements: - Define **how** system performs - Quality attributes: reliability, usability, security, performance - Example: "Email sent within 5 seconds of request"

Importance: - Critical to project success - Foundation for design and testing - Wrong or incomplete requirements lead to software not meeting user needs

Gathering Techniques: - Stakeholder interviews - User stories - Use cases

Quality Criteria: - Unambiguous - Testable - Complete

Week 4: Advanced OOP, Design Principles, and Patterns

The APIEC Framework

Unified framework for object-oriented design principles:

- **A**bstraction - Focus on essential features, hide complexity
- **P**olymorphism - Many forms, uniform interfaces
- **I**nheritance - Code reuse through hierarchies
- **E**ncapsulation - Hide internal state, expose interfaces
- **C**omposition - Build complex objects from simpler ones

Key Principle: Favor composition over inheritance for flexibility and reduced coupling. APIEC principles work together to create maintainable, extensible software.

Encapsulation

Concept: - Keep object's internal data private - Expose only what's necessary through methods/interfaces - Protects integrity of object's state - Use getter/setter methods or properties instead of direct field access - Promotes modularity - Easier to change implementation without affecting other code

Example:

```python
class BankAccount:
    def __init__(self, balance):
        self.__balance = balance  # Private attribute

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount

    def get_balance(self):  # Controlled access
        return self.__balance
```

Dependency Injection

Concept: - Object's dependencies provided from outside rather than created internally - Pass required components via constructors or setters - Supports Inversion of Control principle - Control handed to external entity (framework, factory)

Benefits: - More testable code (can inject mocks in tests) - More flexible code (swap implementations without modifying class) - Reduced coupling

Example:

```python
class EmailService:
    def send(self, message):
        pass

class UserService:
    def __init__(self, email_service):  # DI via constructor
        self.email_service = email_service
```

Polymorphism

Concept: - "Many forms" - treating objects of different classes through uniform interface - Runtime dispatch to actual object's implementation - Improves extensibility - new subclasses introduced without changing base interface usage

Example:

```python
class Shape:
    def draw(self):
        pass

class Circle(Shape):
    def draw(self):
        print("Drawing circle")

class Square(Shape):
    def draw(self):
        print("Drawing square")

shapes = [Circle(), Square()]
for shape in shapes:
    shape.draw()  # Polymorphic behavior
```

Composition

Concept: - Class composed of one or more other classes - Strong "has-a" relationship - Alternative to inheritance for code reuse - Can change composed parts without affecting hierarchy - Runtime assembly of behaviors

Example:

```python
class Engine:
    def start(self):
        print("Engine starting")

class Wheels:
    def rotate(self):
        print("Wheels rotating")

class Car:  # Composition: Car HAS-A Engine and Wheels
    def __init__(self):
        self.engine = Engine()
        self.wheels = Wheels()

    def drive(self):
        self.engine.start()
        self.wheels.rotate()
```

Why Favor Composition: - More flexible than inheritance - Avoid rigid class hierarchies - Car is not a type of engine but has an engine - Combine objects with different responsibilities

Use Case Diagrams

Purpose: - UML diagram to model functional requirements - Shows actors (user roles) and use cases (system functions) - High-level overview of who uses system and what they can do

Components: - **Actors:** Users or external systems - **Use Cases:** Services/functions system provides (shown as ovals) - **Relationships:** Can include <> or <>

Example (Online Shopping): - **Actors:** Customer, Admin - **Use Cases:** Browse Products, Place Order, Manage Inventory

Benefits: - Communication tool between stakeholders and developers - Ensures system scope and interactions well-understood - Represents distinct functionality from actor's perspective

Software Testing

Definition: Process of evaluating software to detect differences between expected and actual output (finding defects).

Market Reality: - Testing market valued at $87.42B in 2024 - Projected to reach $512.3B by 2033 (CAGR 21.71%)

Testing Levels:

1. **Unit Testing** - Testing individual components/functions in isolation
2. **Integration Testing** - Checking modules/services work together properly
3. **System Testing** - Validating entire integrated system against requirements
4. **Acceptance Testing** - Ensuring system meets user needs and ready for deployment

Example (pytest):

```
def calculate_total(price, quantity):
    return price * quantity

def test_calculate_total():
    assert calculate_total(10, 5) == 50
    assert calculate_total(0, 10) == 0
    assert calculate_total(7.5, 2) == 15.0
```

Best Practices: - Write test cases for normal and edge conditions - Automated tests catch regressions quickly - Solid test suite enables confident refactoring - Test-Driven Development (TDD): Write tests before implementing code

SOLID Principles

Five foundational design principles for maintainable OOP code:

1. **Single Responsibility Principle (SRP)**
    - A class should have only one reason to change
    - Each class addresses one concern
    - Easier to debug and update
2. **Open/Closed Principle**
    - Software entities open for extension but closed for modification
    - Add new functionality without changing existing code
3. **Liskov Substitution Principle**
    - Objects of superclass replaceable with objects of subclasses
    - System should not break when using subclasses
4. **Interface Segregation Principle**
    - No client forced to depend on methods it doesn't use
    - Split large interfaces into smaller, specific ones
5. **Dependency Inversion Principle**
    - High-level modules should not depend on low-level modules
    - Both should depend on abstractions

SRP Example:

Violation:

```python
class User:
    def save_to_database(self):
        # Database logic
        pass

    def send_email(self):
        # Email logic
        pass
```

Following SRP:

```python
class User:
    def __init__(self, name, email):
        self.name = name
        self.email = email

class UserRepository:  # Handles DB operations
    def save(self, user):
        pass

class EmailService:    # Handles email operations
    def send(self, user):
        pass
```

Design Patterns

Definition: - General, reusable solution to commonly occurring problem - Template for solving problems, adapted to different situations - Not direct code, but conceptual guidance

Categories (Gang of Four): - **Creational:** Factory, Singleton, Builder - **Structural:** Adapter, Decorator, Facade - **Behavioral:** Observer, Strategy, Iterator

2025 Reality: - Many patterns now built into modern languages - Remain essential as conceptual tools and shared vocabulary - 23 Gang of Four patterns still relevant

Iterator Pattern Example:

```python
class BookCollection:
    def __init__(self):
        self.books = []

    def add_book(self, book):
        self.books.append(book)

    def __iter__(self):  # Iterator protocol
        return iter(self.books)

# Usage
collection = BookCollection()
for book in collection:  # Clean, standard iteration
    print(book)
```

Benefits: - Provides standard way to access collection elements sequentially - Doesn't expose internal structure - Improves interoperability and code clarity - Recognizable patterns communicate solution approach

---

Week 5: High-Level Programming with JavaScript

The Journey from Low-Level to High-Level

The Problem with Low-Level Languages: - C requires 500+ lines for simple web server - Manual memory management prone to errors - Complex string handling and buffer management - Takes weeks to build basic functionality

High-Level Languages Solution: - JavaScript: 20 lines for same web server - Automatic memory management (garbage collection) - Built-in data structures and APIs - Hours instead of weeks for development

Key Insight: High-level languages manage complexity so developers focus on business logic, not implementation details.

What is JavaScript?

JavaScript - High-level, interpreted programming language: - Created in 1995 by Brendan Eich at Netscape - Lingua franca of the web - Runs in browsers and on servers (Node.js)

Key Characteristics: - **Interpreted:** No compilation step required - **Dynamic typing:** Variables can hold any type - **First-class functions:** Functions are values - **Prototype-based:** Object inheritance through prototypes - **Event-driven:** Designed for asynchronous operations - **Multi-paradigm:** Supports procedural, OOP, and functional programming

JavaScript's Role in Web Development

The Web Technology Stack: - **HTML:** Structure and content - **CSS:** Styling and layout - **JavaScript:** Behavior and interactivity

JavaScript's Unique Position: - Only language running natively in all browsers - Both client-side (browser) and server-side (Node.js) - Powers interactive web applications - Enables Single Page Applications (SPAs)

Market Reality (2025): - 98% of websites use JavaScript - #1 most used programming language (GitHub) - Node.js powers millions of servers worldwide

JavaScript Basics: Variables and Types

Dynamic Typing:

```javascript
let message = "Hello";        // String
let count = 42;               // Number
let isActive = true;          // Boolean
let items = [1, 2, 3];        // Array
let user = {name: "Sarah"};   // Object
let nothing = null;           // Null
let notDefined;               // Undefined
```

Variable Declarations: - `var` - Function-scoped (old style, avoid) - `let` - Block-scoped, can be reassigned - `const` - Block-scoped, cannot be reassigned

Type Coercion:

```javascript
"5" + 3      // "53" (string concatenation)
"5" - 3      // 2 (numeric subtraction)
Boolean("")  // false
Boolean("hello") // true
```

Functions in JavaScript

Function Declarations:

```javascript
function greet(name) {
    return "Hello, " + name;
}

// Arrow function (ES6+)
const greet = (name) => "Hello, " + name;

// First-class functions
const operations = {
    add: (a, b) => a + b,
    multiply: (a, b) => a * b
};
```

Key Concepts: - Functions as first-class values - Can be passed as arguments - Can be returned from other functions - Enable functional programming patterns

Objects and Prototypes

Object Literals:

```
const user = {
    name: "Sarah",
    email: "sarah@example.com",
    greet: function() {
        return `Hello, ${this.name}`;
    }
};
```

Prototype-Based Inheritance:

```
function User(name, email) {
    this.name = name;
    this.email = email;
}

User.prototype.greet = function() {
    return `Hello, ${this.name}`;
};

const user = new User("Sarah", "sarah@example.com");
```

Asynchronous JavaScript

The Problem: JavaScript is single-threaded, but needs to handle I/O operations without blocking.

Solutions: - **Callbacks:** Functions passed to be called later - **Promises:** Objects representing eventual completion - **Async/Await:** Syntactic sugar for promises

Promise Example:

```
fetch('/api/users')
    .then(response => response.json())
    .then(data => console.log(data))
    .catch(error => console.error(error));
```

Async/Await Example:

```
async function getUsers() {
    try {
        const response = await fetch('/api/users');
        const data = await response.json();
        console.log(data);
    } catch (error) {
        console.error(error);
    }
}
```

Modern JavaScript (ES6+)

Key Features: - **Arrow functions:** Concise syntax - **Template literals:** String interpolation - **Destructuring:** Extract values from objects/arrays - **Spread operator:** Expand iterables - **Classes:** Syntactic sugar for prototypes - **Modules:** Import/export functionality

Example:

```
// Template literals
const name = "Sarah";
console.log(`Hello, ${name}!`);

// Destructuring
const {name, email} = user;

// Spread operator
const newArray = [...oldArray, newItem];

// Classes
class User {
    constructor(name) {
        this.name = name;
    }
}
```

Week 6: TypeScript & React

From JavaScript to TypeScript: The Type Safety Evolution

JavaScript's Growing Pains: - Fast development but runtime errors - No compile-time type checking - Refactoring risky in large codebases - Hard to maintain 5000+ line projects - 90% of production bugs are type-related

TypeScript's Solution: - Superset of JavaScript (all JS is valid TS) - Static type checking at compile time - Catches errors before code runs - Better IDE support (autocomplete, refactoring) - **90% fewer bugs** in production

Key Insight: TypeScript adds safety of statically-typed languages while keeping JavaScript's productivity.

What is TypeScript?

TypeScript - Strongly-typed language building on JavaScript: - Developed by Microsoft (2012) - Adds static type definitions to JavaScript

Key Characteristics: - **Transpiles to JavaScript:** TS code compiles to plain JS - **Gradual typing:** Adopt types incrementally - **Type inference:** Types often inferred automatically - **Modern features:** Includes latest ECMAScript features - **Tooling support:** Excellent IDE integration

Market Reality (2025): - Used by 78% of professional JavaScript developers - Powers major projects: VSCode, Slack, Airbnb - Default choice for new large-scale projects

TypeScript Basics: Type Annotations

Basic Types:

```
let username: string = "Sarah";
let age: number = 25;
let isActive: boolean = true;
let values: number[] = [1, 2, 3];
let tuple: [string, number] = ["Sarah", 25];

// Type inference
let message = "Hello";  // inferred as string
```

Compare to JavaScript:

```
// JavaScript - no type safety
let age = 25;
age = "twenty-five";  // No error, but likely a bug

// TypeScript - compile error
let age: number = 25;
age = "twenty-five";  // ERROR: Type 'string' not assignable to
        'number'
```

Functions with Types

Function Signatures:

```
function greet(name: string): string {
    return `Hello, ${name}`;
}

// Arrow function
const add = (a: number, b: number): number => a + b;
```

```typescript
// Optional parameters
function log(message: string, level?: string): void {
    console.log(`[${level || "INFO"}] ${message}`);
}

// Default parameters
function createUser(name: string, role: string = "user") {
    return {name, role};
}

// Rest parameters
function sum(...numbers: number[]): number {
    return numbers.reduce((acc, n) => acc + n, 0);
}
```

Interfaces: Defining Object Shapes

Interface Definition:

```typescript
interface User {
    id: number;
    name: string;
    email: string;
    age?: number;           // Optional property
    readonly created: Date; // Cannot be modified
}

function createUser(data: User): void {
    console.log(`Creating user: ${data.name}`);
    // data.created = new Date(); // ERROR: readonly property
}

const newUser: User = {
    id: 1,
    name: "Sarah",
    email: "sarah@example.com",
    created: new Date()
};
```

Interfaces vs Type Aliases

Type Aliases:

```typescript
type ID = number | string;
type Status = "pending" | "active" | "inactive";

type User = {
    id: ID;
    name: string;
    status: Status;
};
```

When to Use: - **Interfaces:** Object shapes, especially when extending - **Type aliases:** Unions, primitives, tuples

Extending Interfaces:

```
interface User {
    name: string;
    email: string;
}

interface Admin extends User {
    permissions: string[];
}
```

What is React?

React - JavaScript library for building user interfaces: - Developed by Facebook (2013) - Component-based architecture - Virtual DOM for efficient updates - Declarative programming model

Market Reality (2025): - Most popular frontend framework - Used by Facebook, Instagram, Netflix, Airbnb - 220k+ stars on GitHub - Rich ecosystem of tools and libraries

Key Concepts: - **Components:** Reusable UI building blocks - **Props:** Data passed to components - **State:** Component's internal data - **Virtual DOM:** Efficient rendering - **JSX:** JavaScript XML syntax

React Components

Function Components:

```
interface Props {
    name: string;
    age: number;
}

function UserProfile(props: Props) {
    return (
        <div>
            <h1>{props.name}</h1>
            <p>Age: {props.age}</p>
        </div>
    );
}

// Using destructuring
function UserProfile({name, age}: Props) {
    return (
        <div>
            <h1>{name}</h1>
            <p>Age: {age}</p>
        </div>
    );
}
```

React Hooks: useState

State Management:

```
import {useState} from 'react';

function Counter() {
    const [count, setCount] = useState(0);

    return (
        <div>
            <p>Count: {count}</p>
            <button onClick={() => setCount(count + 1)}>
                Increment
            </button>
        </div>
    );
}
```

Key Concepts: - `useState` hook for component state - Returns array: [value, setter function] - State changes trigger re-render - Functional updates for complex state

React Hooks: useEffect

Side Effects:

```
import {useEffect, useState} from 'react';

function UserData() {
    const [users, setUsers] = useState([]);

    useEffect(() => {
        // Runs after render
        fetch('/api/users')
            .then(res => res.json())
            .then(data => setUsers(data));

        // Cleanup function (optional)
        return () => {
            // Cleanup code
        };
    }, []); // Empty array = run once on mount

    return <div>{/* Render users */}</div>;
}
```

Dependency Array: - Empty `[]`: Run once on mount - `[dep]`: Run when dependency changes - No array: Run on every render

Component Composition

Building Complex UIs:

```
function TodoItem({todo, onDelete}: TodoItemProps) {
    return (
        <div>
            <span>{todo.title}</span>
            <button onClick={() => onDelete(todo.id)}>Delete</
button>
        </div>
    );
}

function TodoList() {
    const [todos, setTodos] = useState<Todo[]>([]);

    const handleDelete = (id: number) => {
        setTodos(todos.filter(t => t.id !== id));
    };

    return (
        <div>
            {todos.map(todo => (
                <TodoItem
                    key={todo.id}
                    todo={todo}
                    onDelete={handleDelete}
                />
            ))}
        </div>
    );
}
```

Benefits: - Reusable components - Clear data flow (props down, events up) - Maintainable code structure - Separation of concerns

---

Practical Application: HW4 TODO REST API

Throughout this course, I applied the principles learned to build a comprehensive TODO REST API project, demonstrating quality software engineering practices.

Project Overview

Dual Implementation: - PHP version with Nginx - Python version with FastAPI - Both implementations follow same API specification - Demonstrates language-agnostic design principles

Key Features: - 14 REST endpoints (Authentication, Lists, Tasks) - JWT authentication with secure password hashing - Token blacklisting for secure logout - SQL injection prevention (prepared statements/SQLAlchemy) - XSS protection (HTML escaping/Pydantic validation) - Comprehensive unit tests (39 PHP tests, Python pytest suite) - Bruno API collection for testing - Docker containerization

Application of Software Engineering Principles

Week 1 - Tools & Version Control: - Used Git for version control throughout development - GitHub for code hosting and documentation - VSCode as primary IDE - Marp for creating tutorial presentations

Week 2 - Agile Process: - Iterative development in phases - Continuous testing alongside development - Regular commits showing incremental progress - DevOps practices with Docker containerization

Week 3 - OOP & Design: - Object-oriented architecture in both implementations - Clear class responsibilities (User, TodoList, TodoItem) - UML diagrams documenting system structure - Requirements gathered and documented before coding

Week 4 - Design Principles: - **SOLID Principles:** - Single Responsibility: Separate classes for auth, lists, tasks - Dependency Injection: Database connections injected - Interface Segregation: Focused API endpoints - **Encapsulation:** Private methods and protected data - **Composition over Inheritance:** Services composed of focused components

Week 5 - High-Level Languages: - JavaScript for testing and Bruno collections - Demonstrated benefits of high-level languages - Rapid development with modern language features

Week 6 - TypeScript: - Type-safe API client interfaces defined - Documentation includes TypeScript examples - Demonstrates evolution from dynamic to static typing

Security Best Practices

Authentication & Authorization: - JWT tokens for stateless authentication - Argon2 password hashing (Python) / bcrypt (PHP) - Token blacklisting prevents reuse after logout - Bearer token authentication on protected endpoints

Input Validation: - Pydantic models in Python validate input types - PHP input sanitization and validation - Email format validation - SQL injection prevention with prepared statements

XSS Protection: - HTML special character escaping - Content-Type headers properly set - Input validation on all user data

Testing & Quality Assurance

PHP Implementation: - 39 PHPUnit tests covering all endpoints - Separate test database configuration - Authentication flow testing - Error case validation

Python Implementation: - Pytest test suite with comprehensive coverage - Fixture-based test setup - Async test support - Integration testing

API Testing: - Bruno collection with all 14 endpoints - Environment-based configuration - Authentication token management - Example requests and responses

Documentation Quality

Comprehensive README files: - Quick start guides - Setup instructions - Testing procedures - Troubleshooting tips

API Reference Documentation: - All endpoints documented - Request/response examples - Error codes and messages - Authentication requirements

Tutorial Presentations: - Marp slides explaining architecture - Code examples and best practices - Setup and deployment guides

---

Key Takeaways and Insights

Software Quality is Multi-Dimensional

Quality software requires attention to multiple aspects: - **Code Quality:** Following principles like SOLID, DRY, KISS - **Process Quality:** Using appropriate methodologies (Agile, Scrum) - **Documentation Quality:** Clear, comprehensive, up-to-date - **Testing Quality:** Comprehensive coverage, automated tests - **Security Quality:** Proactive threat mitigation

The Evolution of Programming Languages

High-level languages represent decades of accumulated wisdom: - Abstract away low-level complexity - Provide safer primitives (garbage collection, type safety) - Enable faster development - Trade-off: Performance for productivity (mostly worth it) - TypeScript shows evolution continues toward safer abstractions

Design Principles are Universal

Principles like SOLID, composition over inheritance, and dependency injection apply across languages and paradigms: - Same concepts in PHP, Python, JavaScript, TypeScript - Language syntax differs, but principles remain constant - Understanding principles > memorizing syntax

Testing is Not Optional

Comprehensive testing provides: - Confidence to refactor - Documentation of expected behavior - Regression prevention - Quality assurance - Faster debugging

Communication is Critical

Software engineering is as much about communication as coding: - Clear documentation enables collaboration - UML diagrams communicate design - Code comments explain why, not what - User stories capture requirements - API documentation serves as contract

DevOps Culture Matters

Breaking down silos between development and operations: - Faster, more reliable deployments - Shared responsibility for outcomes - Automation reduces errors - Continuous feedback improves quality - Infrastructure as code enables reproducibility

Security Must Be Proactive

Security cannot be an afterthought: - Built into design from the start - Defense in depth (multiple layers) - Input validation at all boundaries - Principle of least privilege - Regular security reviews

---

Reflection on Learning Outcomes

Technical Skills Developed

Programming Languages: - Python for backend development - PHP for web services - JavaScript for frontend and tooling - TypeScript for type-safe development

Frameworks and Tools: - FastAPI (Python web framework) - React (UI library) - Docker (containerization) - Git/GitHub (version control) - Bruno (API testing) - pytest/PHPUnit (testing frameworks)

Software Engineering Practices: - Agile methodology - Test-driven development - Continuous integration - API design - Database design - Security best practices

Conceptual Understanding

Design Principles: - SOLID principles in practice - APIEC framework application - Design patterns recognition and use - Composition over inheritance - Dependency injection

Software Processes: - Waterfall vs Agile tradeoffs - Scrum framework implementation - DevOps culture and practices - Continuous delivery pipelines

Quality Assurance: - Testing at multiple levels - Security threat modeling - Code review practices - Documentation standards

Professional Development

Communication: - Technical writing for documentation - Presentation creation with Marp - Code comments and explanations - API documentation

Problem Solving: - Breaking complex problems into manageable pieces - Choosing appropriate tools and technologies - Debugging systematic approach - Research and learning new technologies

Project Management: - Iterative development - Version control best practices - Task prioritization - Time management

---

Conclusion

This six-week journey through software engineering principles has provided comprehensive understanding of what makes software "quality." Quality is not just about working code—it encompasses design, testing, security, documentation, and maintainability.

The practical application of these principles in the TODO REST API project (HW4) demonstrated that theoretical knowledge translates into real-world value. By following established principles and best practices, I created secure, tested, well-documented software that would be maintainable in a professional setting.

Key insights from this experience:

1. **Quality is intentional** - It requires conscious effort and planning, not luck
2. **Principles are universal** - They apply across languages and paradigms
3. **Testing is investment** - Upfront cost pays dividends in maintenance phase
4. **Documentation is for humans** - Code is written once, read many times
5. **Security is foundational** - Cannot be bolted on after the fact
6. **Communication matters** - Software engineering is team sport
7. **Tools amplify skills** - Right tools make good practices easier
8. **Learning is continuous** - Technology evolves, principles endure

Moving forward, these principles will guide my approach to software development. Understanding not just *how* to code, but *why* certain practices matter, enables making better decisions and building better software.

The discipline of software engineering is about managing complexity, mitigating risk, and delivering value. Quality software emerges from the application of proven principles, supported by appropriate tools and processes, executed by skilled developers who communicate effectively.

Appendices

A. Project Links

GitHub Repository: - NKU-640 Course Repository

HW4 Project Documentation: - Project Overview - PHP Implementation README - Python Implementation README - PHP API Reference - Python API Reference

B. Weekly Presentation Links

- Week 1: Course Introduction & Tools Overview
- Week 2: Software Process Overview
- Week 3: Python, OOP, and UML Fundamentals
- Week 4: Advanced OOP, Design Principles, and Patterns
- Week 5: High-Level Programming with JavaScript
- Week 6: TypeScript & React

C. Technologies Used

Development Tools: - Visual Studio Code - Git & GitHub - Docker & Docker Compose - Postman/Bruno for API testing

Backend Technologies: - PHP 8.x with Nginx - Python 3.11+ with FastAPI - SQLite database - JWT for authentication - Argon2/bcrypt for password hashing

Frontend Technologies: - JavaScript (vanilla, for TODO app examples) - TypeScript (for type-safe React) - React 18+ with hooks - HTML5 & CSS3

Testing Frameworks: - PHPUnit (PHP testing) - pytest (Python testing)

Documentation Tools: - Marp for presentations - Markdown for documentation - MkDocs for documentation site

D. References and Resources

Books and Documentation: - Gang of Four: Design Patterns - Clean Code by Robert C. Martin - The Pragmatic Programmer - Python Official Documentation - TypeScript Official Documentation - React Official Documentation

Online Resources: - MDN Web Docs (JavaScript reference) - FastAPI Documentation - PHPUnit Documentation - Git Documentation

Course Materials: - NKU ASE GitHub Repository - Course lecture slides - Assignment specifications

---

Report Completed: December 2, 2025 **Total Pages:** Comprehensive documentation of 6 weeks of software engineering principles and practices

Signature: *Navleen Singh*