Methods in Software Engineering

1) Tests

→ Test Types:

Unit Tests

- Tests **expected functionality** of a module

  ➕ Fast to run, easy to debug
  ➖ Misses integration issues

System Tests

- Tests of **several modules** that work **together**

  ➕ Verifies that the entire system works
  ➖ Slower, harder to debug

→ Properties of Good Tests:

Specific

→ Each test should cover **one single aspect**

Isolated

→ Tests should **not depend** on each other (can run in **parallel**)

> **Use tools like Docker for isolation**

→ Regression

> ⚠ **Previously working features stop working after new changes**

→ Continuous Integration

- **Frequently merging** changes into a shared repository
- **Automated** tests & builds

→ Containers

> ✎ **Examples: Docker, Podman**

Benefits:

- **Reproducible**
- **Isolation**
- **Scalability**

2) Version Control Systems

> **Definition**
>
> - A system that lets you manage and keep track of changes over time
> - Version control is essential for collaboration

→ Git

Features

- `commit`
- `fork`
- `merge`
- `cherry-pick`
- `checkout`
- `stash`

Conventions

- Commit **often** with useful commit messages
- Naming convention examples:
  `feat`, `fix`, `chore`, ...

♨ **Document *why* the change is made, not just what was changed**

→ Local vs Centralized vs Decentralized

- **Local version control**
  → Tracks changes only on one machine
- **Centralized version control**
  → Uses a single central server that all users sync with
- **Decentralized version control**
  → Every user has a full copy of the repository

---

3) Software Maintenance

→ Challenges

- **Fixing bugs**
- **Adding new features**
- **Documentation needs to keep up**

→ Modern Development Infrastructure (Key Tools):

- **Distributed version control systems**
- **Ticket/Issue tracking**
- **Continuous integration**

→ Documentation

Challenge

- often viewed as **low priority**
- ensuring **consistency** and **quality**

Solutions

- **IDE tools like sphinx, TypeDoc …**
- **AI for automatic doc-generation**

→ Issue Tracking

- **Issue Tracking with tools like Jira**
- **Description: expected vs. actual behavior, steps to reproduce the issue, …**

---

4) Domain-Specific Languages

Definition

- A **language tailored** to a specific domain/problem
- Simpler and more concise than general-purpose languages (GPLs)

→ Advantages

- **improves productivity**

Example

- **SQL** is a DSL for querying databases. Instead of writing complex logic in a general-purpose language, you can simply write:

```sql
SELECT * FROM users WHERE age > 18;
```

This is **shorter, clearer**, and more maintainable for its domain.

→ Examples

- **SQL**
- **HTML**
- **Regex**
- **Custom rule-language for home automation**

→ Properties

- **Defines a valid expression (often as a grammar or UML)**
- **Needs function like** `evaluate(formula, validation): boolean`
- **Uses a valuation (e.g.** `{ A: true, B: false }` **) to evaluate logic formulas**

5) Refinement Types

Definition

A base type and a constraint → **more precise** than traditional types

→ Examples

Logical Expression

```
int where x: 2 <= x <=7
```

Scala

```scala
case class T(x: Int) {
    assert (2 <= x <= 7)
}
```

TypeScript

(personal favorite)

```typescript
type RefinementType = number & { __tag: 'between2And7' };

function isBetween2And7(x: number): x is RefinementType {
    return x >= 2 && x <= 7;
}

class T {
    public readonly value: RefinementType;

    constructor(x: number) {
        if (!isBetween2And7(x)) {
            throw new Error();
        }

        this.value = x as RefinementType;
    }
}
```

6) OOP: Object-Oriented Programming

→ Key Concepts

- **Encapsulation**
- **Inheritance**
- **Polymorphism**

- **Abstraction**

→ Composition vs Inheritance vs Association

Inheritance

- `is-a` relationship

```
class Animal:
    pass

class Dog(Animal):
    pass
```

Composition

- `has-a` relationship

```
class Engine:
    pass

class Car:
    __init__(self, engine: Engine):
        self.engine = engine
```

Association

- `uses-a` relationship

```
class Driver:
    def drive(self):
        print("driving")

class Car:
    def move(self, driver: Driver):
        driver.drive()
```

→ Components vs Data vs Algorithms

Concept	Description	Example
Component	A self-contained module (class/object) with data and behavior	`User`, `ShoppingCart`, `Engine`
Data	The state stored in a component	`name`, `balance`, `position`

Concept	Description	Example
Algorithm	The behavior or logic (methods)	`deposit()`, `move()`, `sort()`

→ Statement

OOP is outdated and should not be used anymore

OOP is **not outdated**, but like any paradigm, it has **strengths and weaknesses**:

➕ Great for **modeling real-world entities** using objects (e.g., `User`, `Account`)
➕ Supports **encapsulation**

➖ Can lead to **over-engineering**
➖ Makes **testing and reasoning** harder

7) Liskov Substitution Principle (LSP)

> **Definition**
>
> Subtypes must be **replaceable** for their base types **without breaking correctness**

→ Example

- If `A` is a base class and `B` is a subclass, then `B` must behave consistently with `A`
- Violations: changed input constraints, unexpected exceptions, etc.

➕ Practical Example:

In a **payment system**, if `CreditCardPayment` and `PayPalPayment` both implement a `PaymentMethod` interface, then code using `PaymentMethod` (like `checkout(payment)`) should work **correctly regardless of which subclass is passed** — ensuring reliable and flexible code reuse.

8) Invariants

> **Definition**
>
> - A condition that must **always hold** true for a class in a valid state
> - Checked before/after public method calls

→ Properties

- Example: `0 <= count <= len(items)`
- Use `assert` or validation logic inside constructors/methods
- cannot always be expressed as e.g. some invariants are too abstract or high-level
- can be expressed using `assert` statements, type systems and contracts

  ➕ They help catch bugs early

  ➖ Maintaining invariants can add complexity

9) Mutability

→ Mutable

- Object **can change** after creation → e.g. lists, dictionaries in Python

→ Immutable

- Object **cannot change** once created → e.g. tuples, strings in Python

→ Why it matters

- Immutable types are **safer**, better for testing and reasoning