Sri Lanka Institute of Information Technology

APPLICATION FRAMEWORKS

INTRODUCTION

LECTURE 01

Faculty of Computing
Department of Software Engineering
Module Code: SE3040

MODULE OUTLINE

01 Details

Module Name: Application Frameworks
Module Code: SE3040
Credit Points: 04

02 Assessment Criteria

Continuous Assessments	60%
Final Examination	40%

OUR TEAM

MS. KARTHIGA RAJENDRAN

Lecturer-In-Charge

FACULTY OF COMPUTING | SOFTWARE ENGINEERING

MR. EISHAN WEERASINGHE

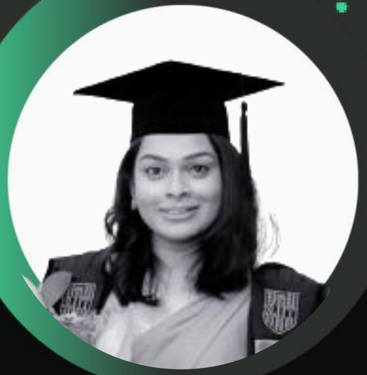
Co-Lecturer

FACULTY OF COMPUTING | SOFTWARE ENGINEERING

Ms. MADUSHA WEERASOORIYA

Assistant Lecturer

FACULTY OF COMPUTING | SOFTWARE ENGINEERING



MS. ANUDI WANIGARATHNE

Instructor

FACULTY OF COMPUTING | SOFTWARE ENGINEERING

OVERVIEW

- What is this course?
- Our objectives?
- What are we going to cover?
- How are we going to evaluate you?

WHAT IS THIS COURSE?

- This course will focus on application development using industry standards and industry leading frameworks.
- Mainly this course will build around Java and JavaScript languages.
- Course will also focus on industry practices and principles in software engineering.
- Popular JavaScript and Java frameworks as well as an introduction to popular NoSQL database will be delivered as well.

OBJECTIVES

- Learn industry practices and principles in software engineering and use them to develop individual skills.
- Discover new trends - two industry leading software development languages.

  (JavaScript and Java)
- Improve sufficient knowledge on JavaScript and Java full stack development.
- Learn NoSQL databases, MongoDB and how to use MongoDB in full stack

  development.
- Learn REST style web services.
- Learn industry leading frameworks in web application and web service development along with leading architecture and authentication mechanisms being followed in industry.

A student should be able to develop a full stack web application using JavaScripts and MongoDB while applying engineering principles and practices and should be able to replace the backend service code with a Java web service.

WHAT TECHNOLOGIES DO YOU KNOW?

SE3040 - Initial Data Collection

Weekend B1

WHAT ARE WE GOING TO COVER?

- JavaScript
- ReactJS
- NodeJS
- ExpressJS
- Java
- Spring Boot
- MongoDB
- Containerization

EVALUATION

Evaluation is based on applying the concepts and learnings in practical applications.

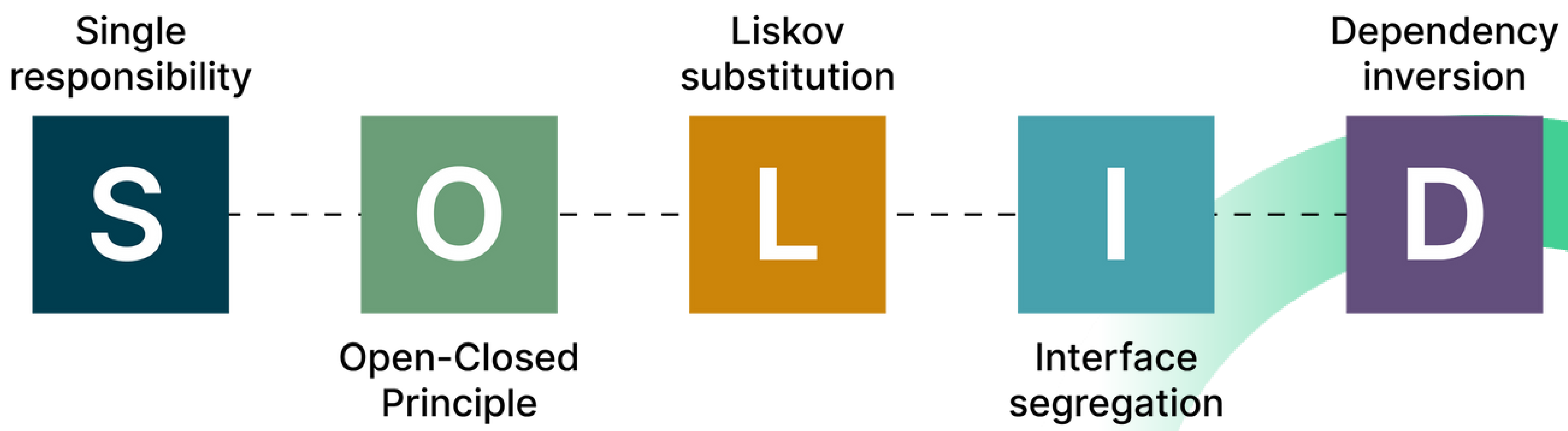
▪ Assignment 01	20%
▪ Assignment 02	20%
▪ Midterm examination	20%
▪ Final examination	40%

PRINCIPLES, GUIDELINES & PRACTICES

PRINCIPLES - S.O.L.I.D

The SOLID Principles are five principles of Object-Oriented class design. They are a set of rules and best practices to follow while designing a class structure.

- Single responsibility
- Open-close Principle
- Liskov substitution
- Interface segregation
- Dependency inversion



SINGLE RESPONSIBILITY

A class should have one and only one reason to change, meaning that a class should have only one job.

it don't mean your class should do only one thing

This is not only about a class its what we consider as a unit ex: function, module, API etc.

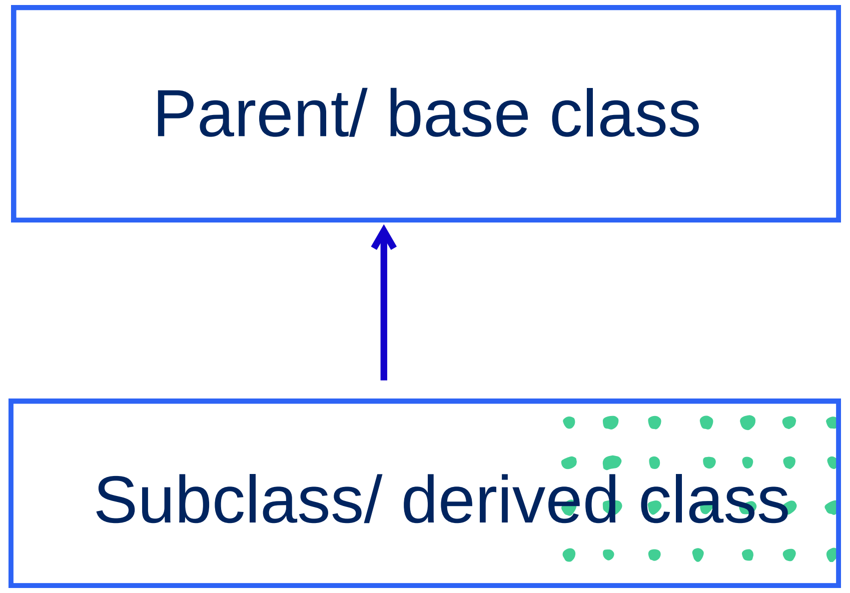
- Think about a class that calculate area or a circle and output the area as a HTML.
- What if in case a JSON output is required?

OPEN/CLOSE PRINCIPLE

"Objects or entities should be **open for extension**, but **closed for modification**"

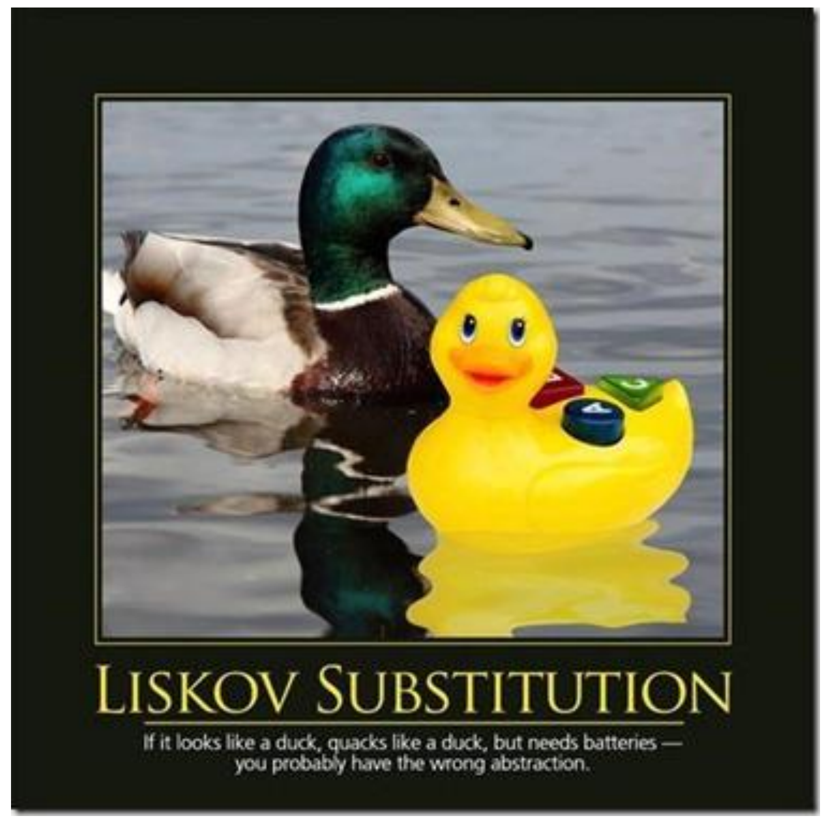
- Think about a class that calculate area or a circle and a square and output the area as a console value.
- What if in case this class need to calculate area of a triangle?
- Use abstraction to keep classes open for extension.
- When there is high possibility to change always depend on abstractions than concrete implementations.

LISKOV SUBSTITUTION



"Every subclass/derived class should be able to substitute their parent/base class"

- When extending a class, it should perform basic functionalities of the base class.
- Child class should not have unimplemented methods.
- Child class should not give different meaning to the methods exist in the base class after overriding them.

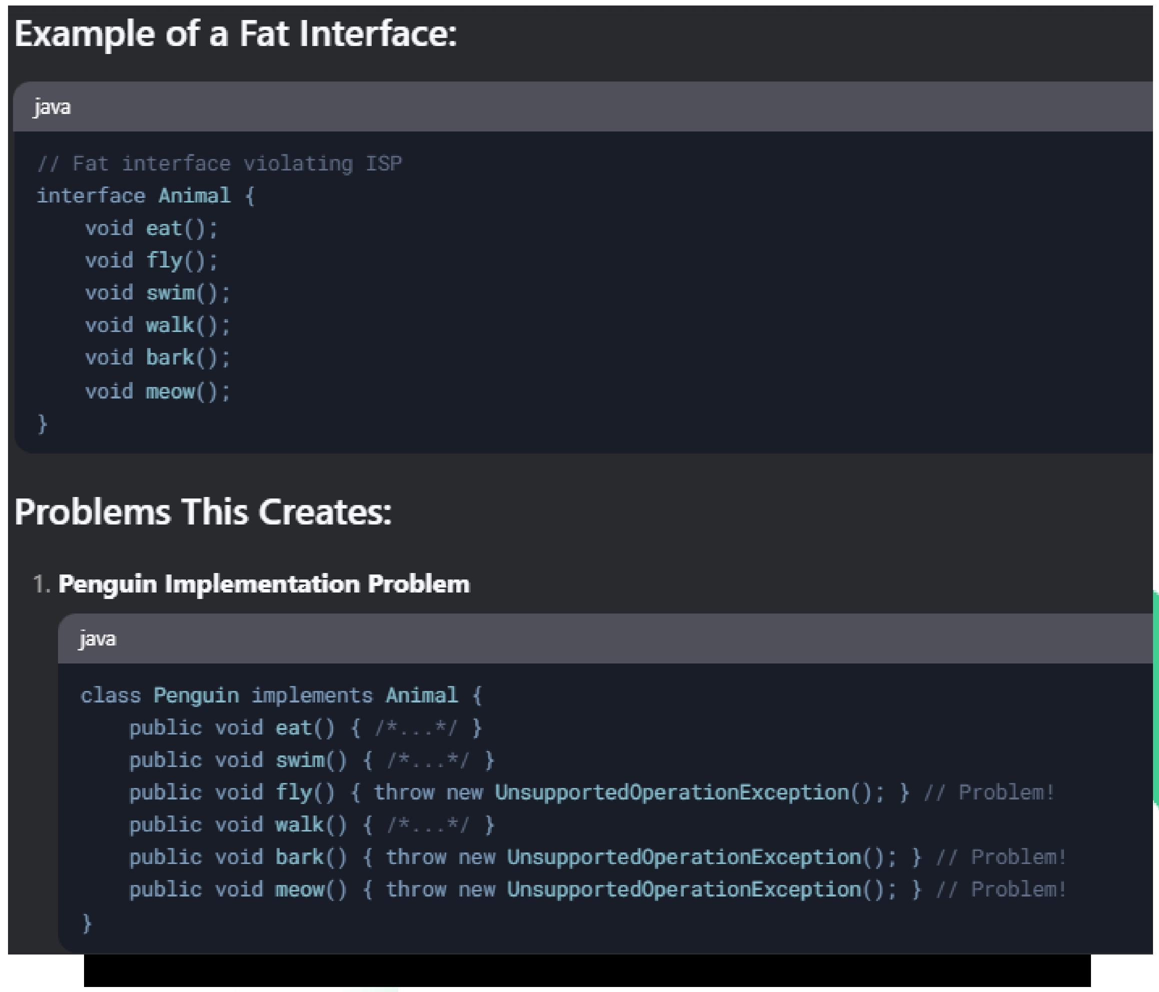

LISKOV SUBSTITUTION
If it looks like a duck, quacks like a duck, but needs batteries — you probably have the wrong abstraction.

INTERFACE SEGREGATION PRINCIPLE

use multiple inheritnce using interface

"Clients should not be forced to implement methods they do not use"

- Think of Shape interface with draw() and calculateArea() methods and a client who wants only the shape to be drawn.
- These are called fat interfaces.
- Group methods into different interfaces each serving different set of clients.



Example of a Fat Interface:

```java
// Fat interface violating ISP
interface Animal {
    void eat();
    void fly();
    void swim();
    void walk();
    void bark();
    void meow();
}
```

Problems This Creates:

1. **Penguin Implementation Problem**

```java
class Penguin implements Animal {
    public void eat() { /*...*/ }
    public void swim() { /*...*/ }
    public void fly() { throw new UnsupportedOperationException(); } // Problem!
    public void walk() { /*...*/ }
    public void bark() { throw new UnsupportedOperationException(); } // Problem!
    public void meow() { throw new UnsupportedOperationException(); } // Problem!
}
```

interface that contains too many methods

forcing implementing classes to provide implementations for methods they don't need or shouldn't have.

Violates Interface Segregation Principle (ISP)

DEPENDENCY INVERSION PRINCIPLE

Business logic related module

"Higher level modules **should not** depend on lower level modules, but they should depend on abstractions"

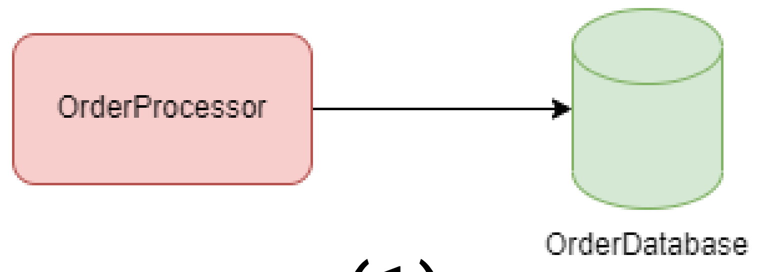
In the first diagram, the OrderProcessor class depends on the concrete implementation of the OrderDatabase class. This creates a **tight coupling** between the two classes.

Violation of DIP (Tight Coupling)

In the second diagram, we can see that OrderProcessor class depends on theabstraction (IOrderDatabase) rather than the concrete implementation (OrderDatabase). The OrderDatabase class implements the IOrderDatabase interface, which defines the operations required by the OrderProcessor class.

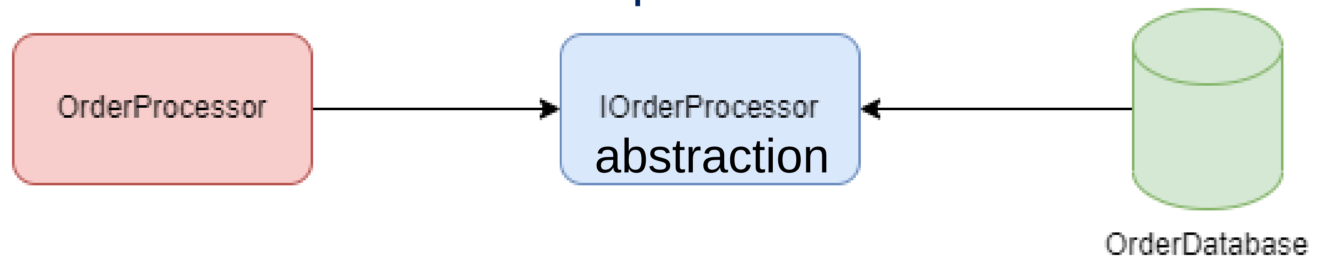
By using this abstraction, we **decouple** the two classes and make the code more flexible and maintainable.

OrderProcessor is tightly coupled to OrderDatabase
Hard to change database implementations



(1)

OrderProcessor depends on abstraction (IOrderDatabase)
Concrete OrderDatabase implements the interface



(2)

Follows the Dependency Inversion Principle

GUIDELINES

1. Think throughout the problem

- Before approaching the solution or even before starting to think about the solution, think through the problem.

- Make sure you understand the problem that you are going to resolve, completely.

- Make sure to clear any unclear part before designing the solution.

- Don’t be afraid to question there are no stupid questions.

2. Approaching the solution

In software engineering we try to find a technical solution for a business problem.

How can we achieve this solution in the best way possible?

Think throughout the problem

- Divide and conquer
- KISS
- Learn, especially from mistakes
- Always remember why software exists
- Remember that you are not the user

DIVIDE AND CONQUER

- Now divide the problem into smaller problems.
- Make it manageable and easily understandable.
- Try to find the perfect balance between priority and clarity (less complex, easily understandable).

KISS

- Keep it simple and stupid. Keep It Short and Simple
- හිතමතම
- Do not deliberately make your solution complex.
- Do not overthink or over engineer.

LEARN FROM THE MISTAKES

- Embrace the change.
- Always anticipate the changes as much as possible.
- Do not over engineer it, but keep the provisions to extend

REASON SOFTWARE EXISTS

- Keep in mind the bigger picture why this software exists.
- Loss of the bigger picture might cause following a wrong path.

YOU WON'T BE USING THE SOFTWARE

- End user is not technically capable as same as you are.
- Do not assume that user will understand.
- User friendliness and user experience matters.

THE NON-TECHNICAL COFOUNDER DISCOVERS HTML ... FOR THE FIRST TIME
LOOK MA... I'M A CODER!
<body>
<br/>
</body>
ENTREPRENEURFAIL.COM

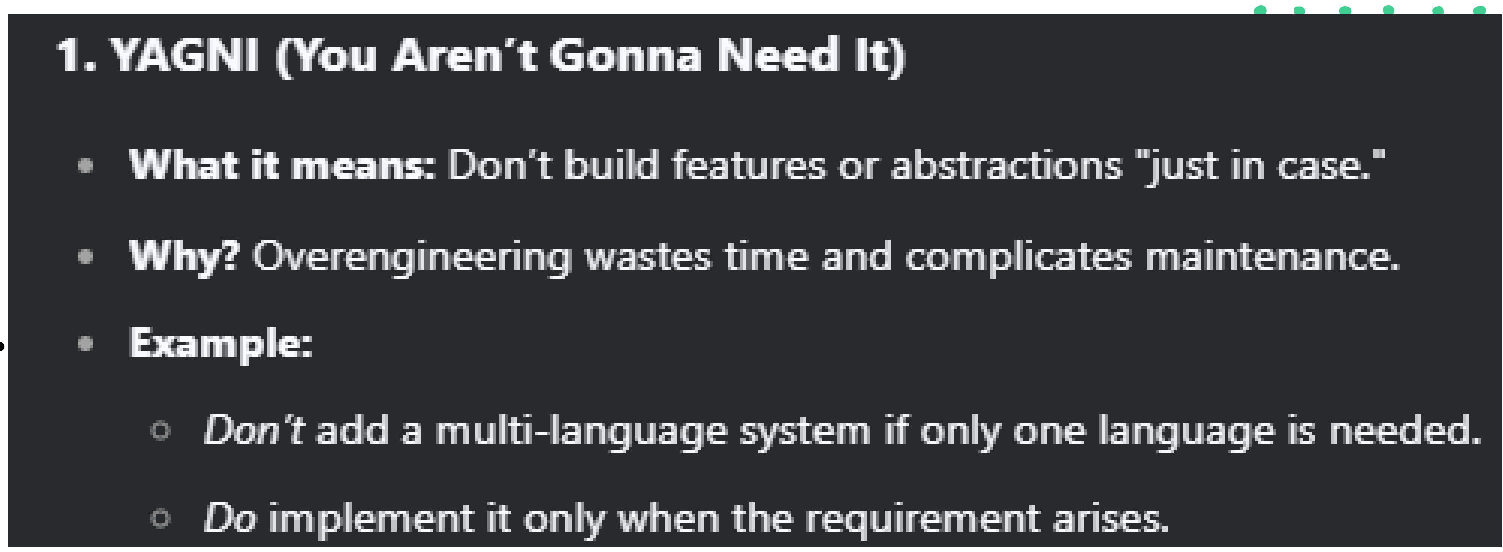
3. Implementing the solution

When we are implementing the designed solution there are some guidelines to keep in mind.

- YAGNI
- DRY
- Embrace abstraction
- DRITW
- Write code that does one thing well
- Debugging is harder than writing code
- Kaizen

YAGNI - YOU AIN'T GONNA NEED IT

- Don't write code that is no use in present but you are guessing will come in handy in the future.
- Future always change. This is a waste of time.
- Write the code you need for the moment- only that.

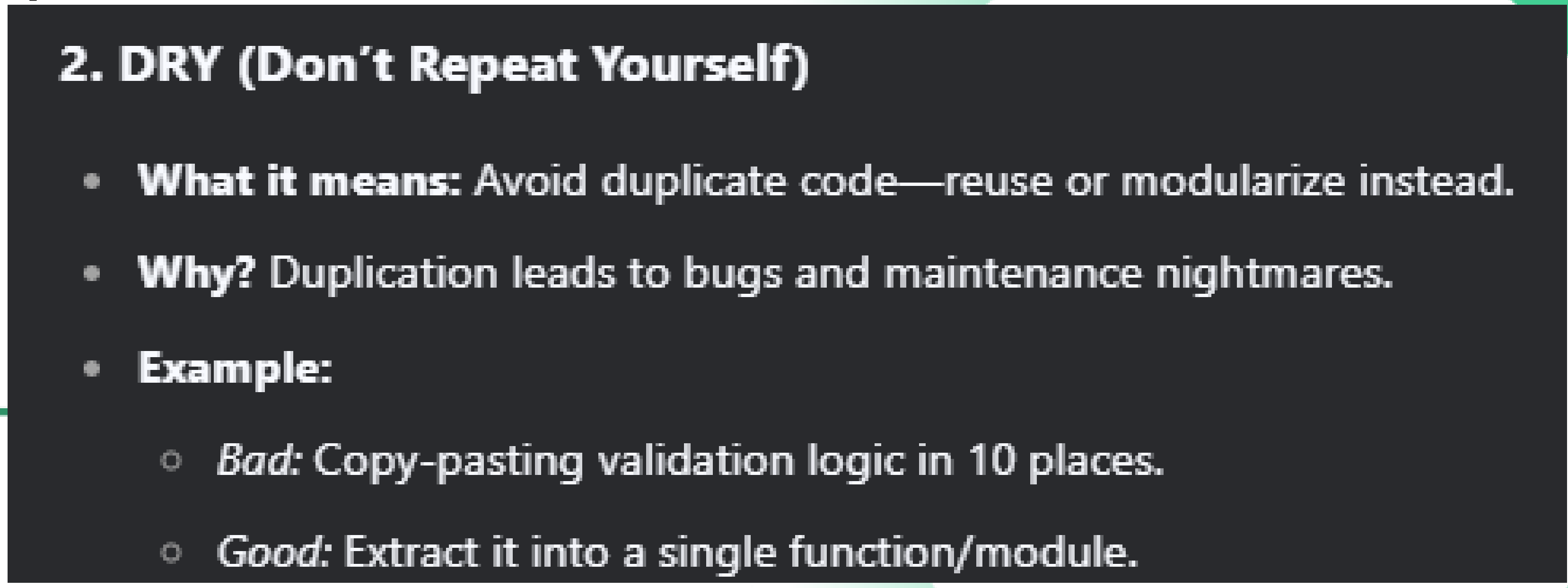


1. YAGNI (You Aren't Gonna Need It)

- **What it means:** Don't build features or abstractions "just in case."
- **Why?** Overengineering wastes time and complicates maintenance.
- **Example:**
  - *Don't* add a multi-language system if only one language is needed.
  - *Do* implement it only when the requirement arises.

DRY - DON'T REPEAT YOURSELF

- Always reuse code you wrote. Code as best as possible and
- keep it generalize and reusable.

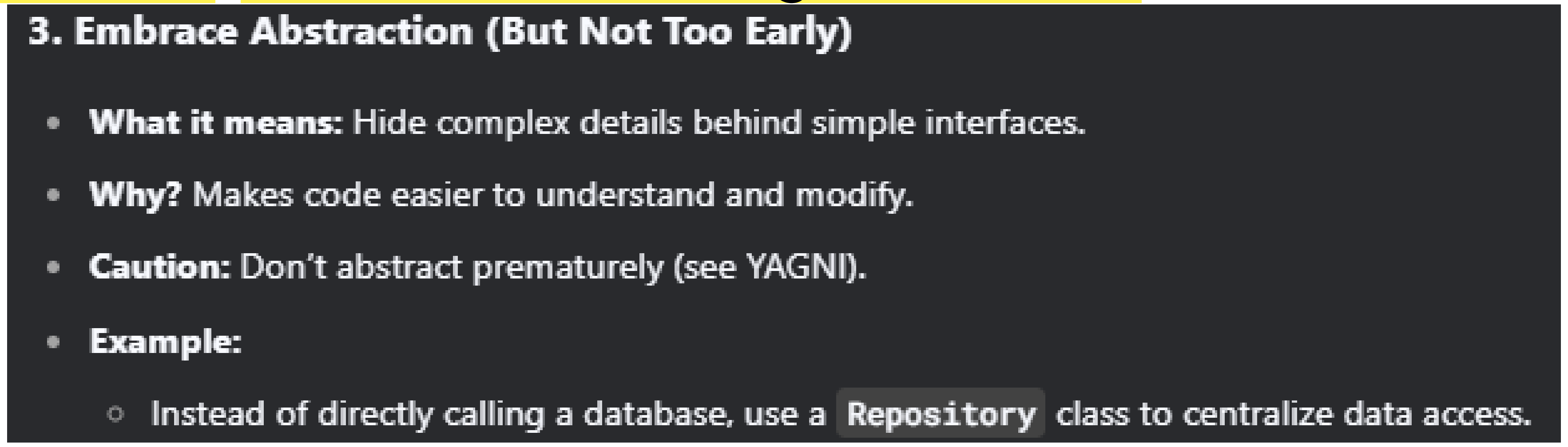


2. DRY (Don't Repeat Yourself)

- **What it means:** Avoid duplicate code—reuse or modularize instead.
- **Why?** Duplication leads to bugs and maintenance nightmares.
- **Example:**
  - *Bad:* Copy-pasting validation logic in 10 places.
  - *Good:* Extract it into a single function/module.

EMBRACE ABSTRACTION

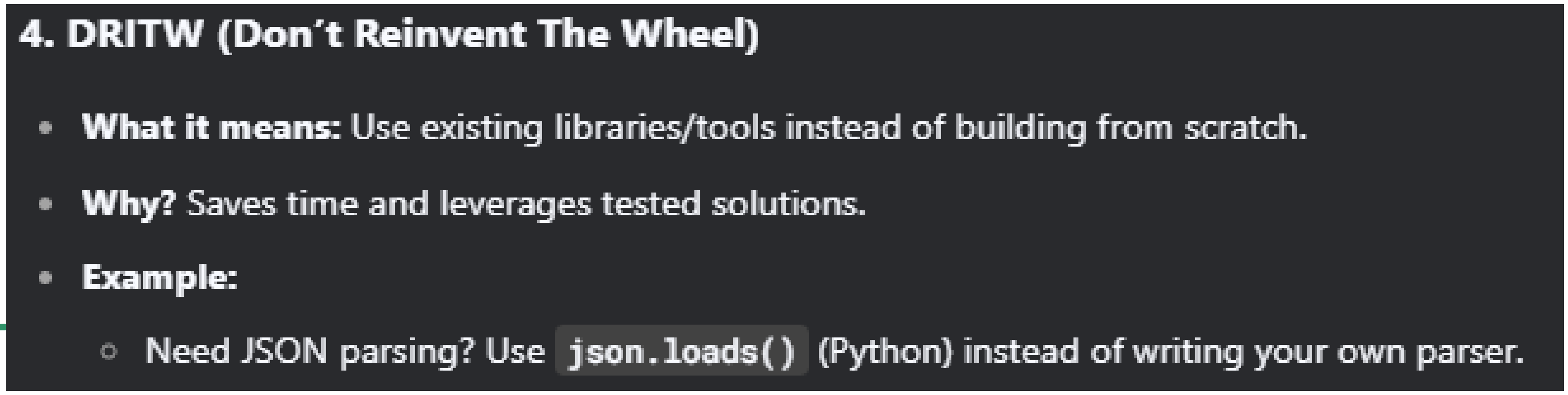
- Make sure your system functions properly without knowing the implementation details of every component part.
- User class should be able to authenticate user without knowing where to get username and password.


3. Embrace Abstraction (But Not Too Early)

- **What it means:** Hide complex details behind simple interfaces.
- **Why?** Makes code easier to understand and modify.
- **Caution:** Don't abstract prematurely (see YAGNI).
- **Example:**
  - Instead of directly calling a database, use a `Repository` class to centralize data access.

DRITW - DON'T REINVENT THE WHEEL

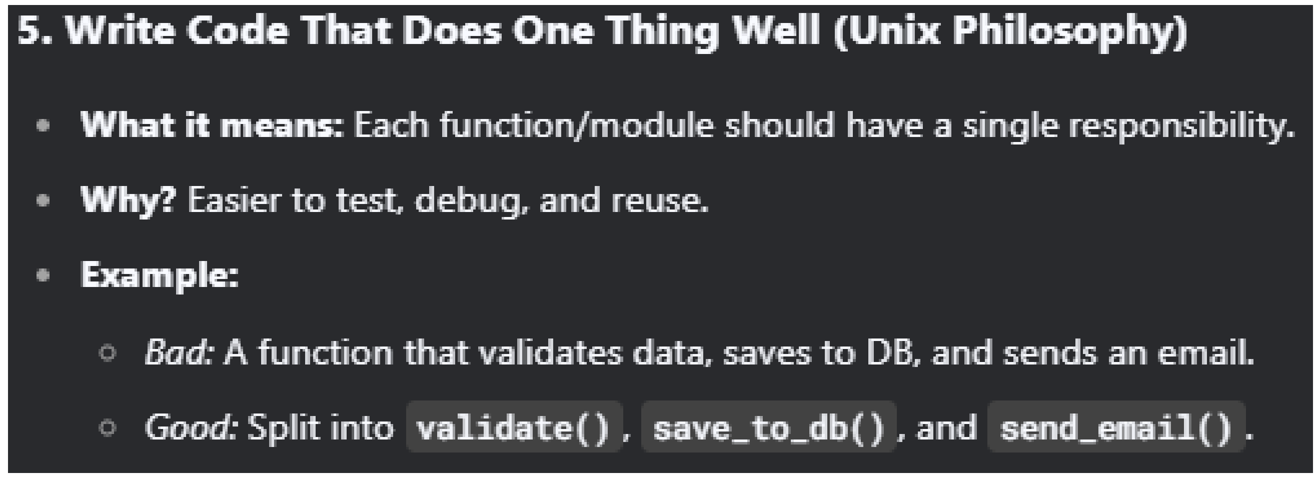
- Someone else might have already solved the same problem. Make use of that.


4. DRITW (Don't Reinvent The Wheel)

- **What it means:** Use existing libraries/tools instead of building from scratch.
- **Why?** Saves time and leverages tested solutions.
- **Example:**
  - Need JSON parsing? Use `json.loads()` (Python) instead of writing your own parser.

WRITE CODE THAT DOES ONE THING WELL

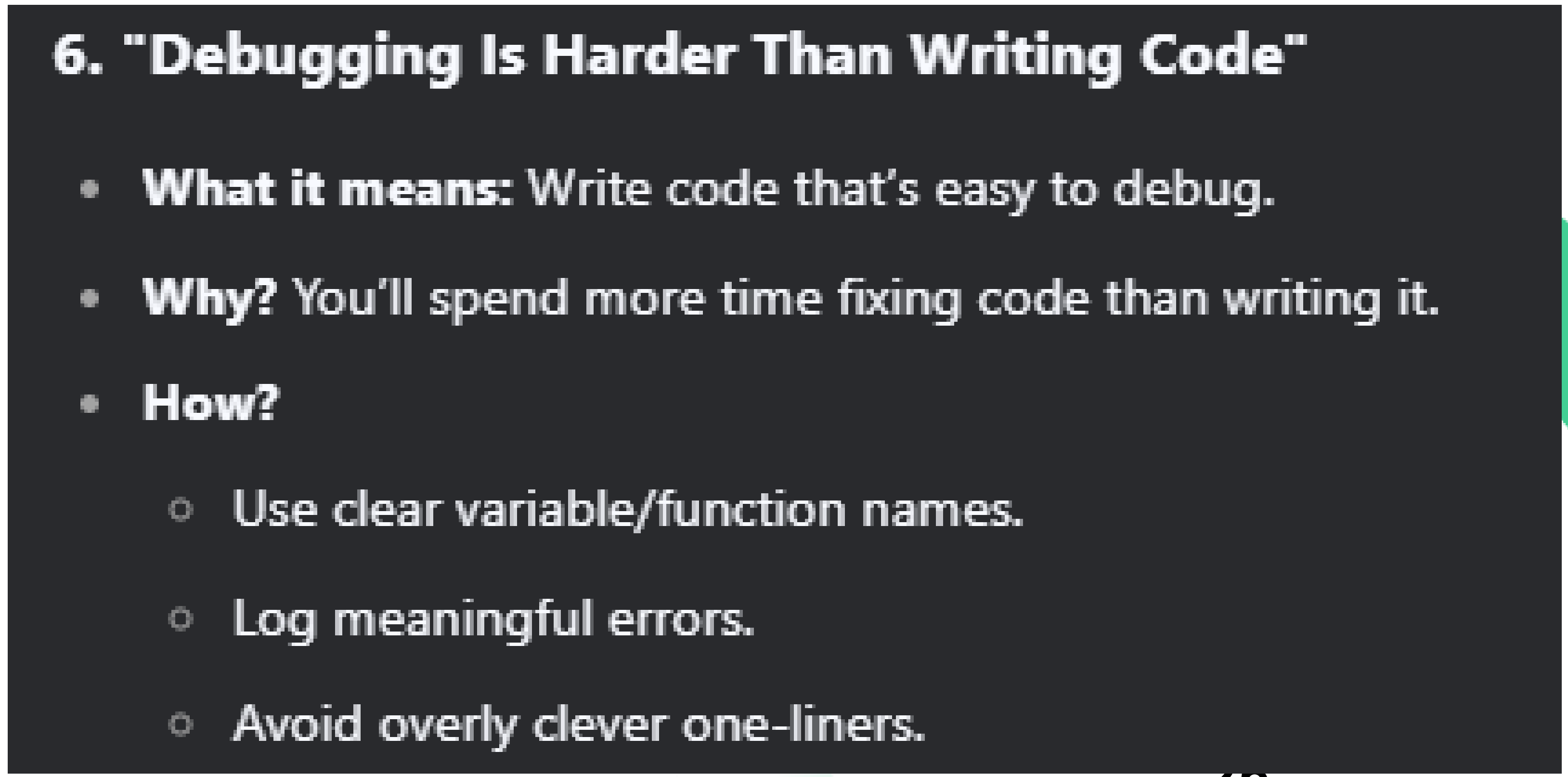
- Single piece of code that do one thing and that one thing really well.
- Do not try to write the magic code that does it all.


5. Write Code That Does One Thing Well (Unix Philosophy)

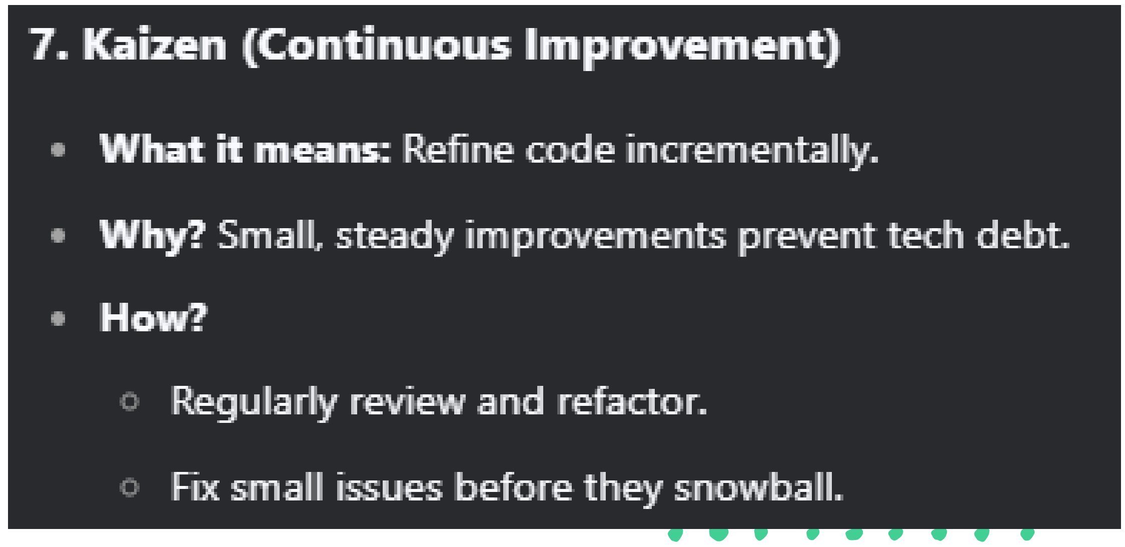
- **What it means:** Each function/module should have a single responsibility.
- **Why?** Easier to test, debug, and reuse.
- **Example:**
  - *Bad:* A function that validates data, saves to DB, and sends an email.
  - *Good:* Split into `validate()`, `save_to_db()`, and `send_email()`.

DEBUGGING IS HARDER THAN WRITING THE CODE

- Make it readable as possible.
- Readable code is better than compact code.


6. "Debugging Is Harder Than Writing Code"

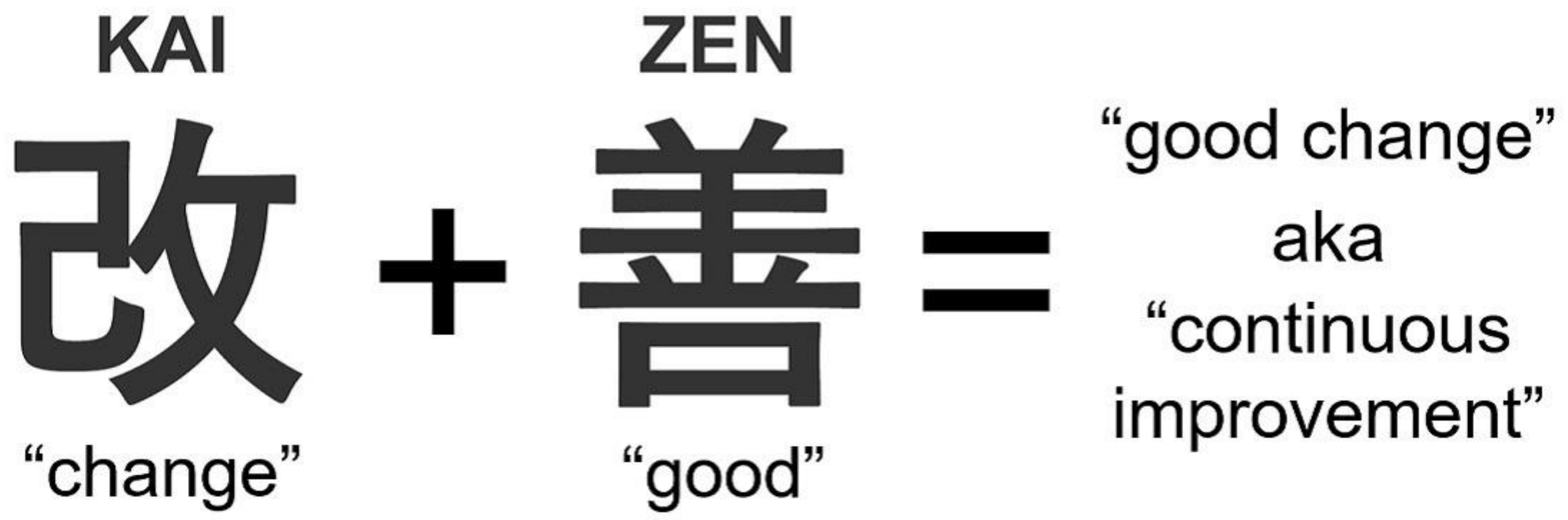
- **What it means:** Write code that's easy to debug.
- **Why?** You'll spend more time fixing code than writing it.
- **How?**
  - Use clear variable/function names.
  - Log meaningful errors.
  - Avoid overly clever one-liners.


7. Kaizen (Continuous Improvement)

- **What it means:** Refine code incrementally.
- **Why?** Small, steady improvements prevent tech debt.
- **How?**
  - Regularly review and refactor.
  - Fix small issues before they snowball.

KAIZEN - LEAVE IT BETTER THAN WHEN YOU FOUND IT

- Fix not just the bug but the code around it.
- Band-aid bugfix won't help if the real problem is a design problem.



PRACTICES

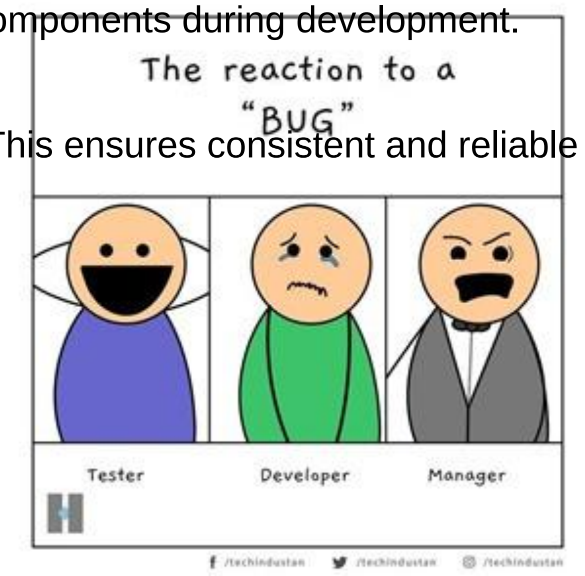
- Practices are norms or set of guidelines that we should follow when we are developing code.
- Introduced by coding guru after studying years of years experience.
- These practices are being considered industry wide as best practices for software engineering.
  - Unit testing
  - Code quality
  - Code review
  - Version controlling
  - Continuous integration

A good unit test has the following characteristics:

Fast Execution:
Unit tests are designed to run in milliseconds, allowing developers to quickly verify the behavior of individual components during development.

Independence:
Unit tests should be isolated and not depend on external systems like databases, file systems, or other tests. This ensures consistent and reliable results.

Small Scope:
Each test focuses on a specific function or method, making it easier to identify issues when a test fails.



UNIT TESTING

Testing individual components (functions, classes) in

- Small code that verifies parts of your main code.
- Unit could be a class, function, module, API etc.
- Unit test will verify the class, function.. Is working as expected and delivers the expected output.
- Allows developer to freely change or improve the code, make sure it didn't break anything by running the unit test.
- Unit testing will eventually make code testable which basically results an extensible code base.
- Verification mechanism for developers.
- Early identification of integration issues.

Catches bugs early.
Ensures code works as expected after changes.

Guidelines:
Follow AAA (Arrange-Act-Assert) pattern.
Use frameworks like JUnit (Java), pytest (Python), Jest (JS).
Aim for high coverage, but prioritize meaningful tests.

CODE QUALITY

Writing clean, readable, and maintainable code.

- Code to be maintainable code quality is vital.
- Code should readable and easily understandable.
- Code should adhere to engineering best practices as well as language and domain best practices.
- Frequently analyze quality of the code using tools.
- Identify potential erroneous scenarios.
- Improve the performance of the code.
- Code complexity, large methods and classes, meaningless identifiers, code duplication, large number of method parameters.

CODE REVIEW

- Best way to improve code quality.
- Objective is to improve the code not to criticize the developer.

indicate the faults of (someone or something)

- Improve the performance, find out the best way of resolving the problem; 4+ eyes on the code.

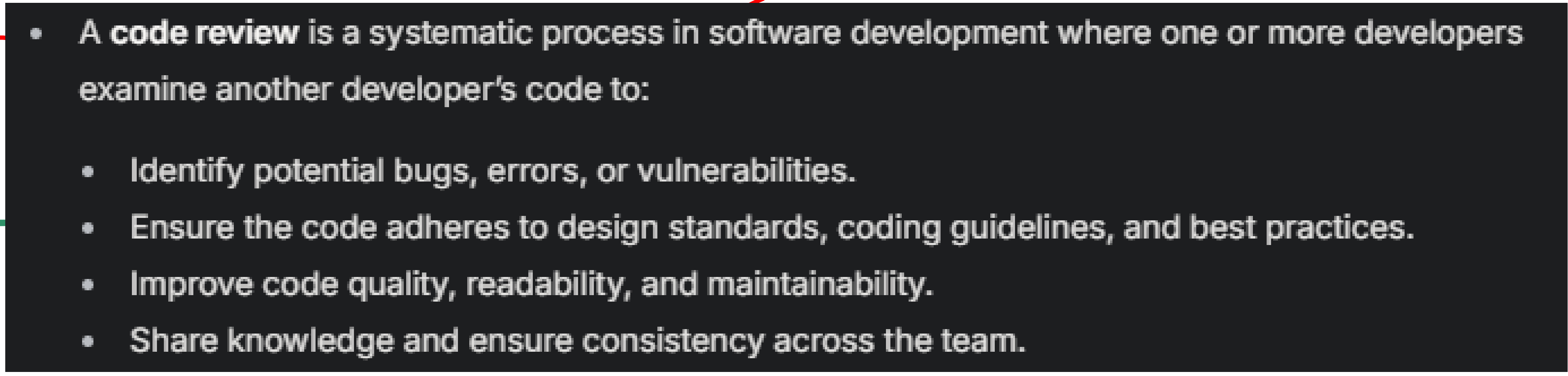
Line of code

- Review less than 400 LOC and rate should be 500 LOC per hour, do not review continuously more than hour.
- Peer reviews, lead reviews and pair programming are some methods of doing code reviews.

involve team members reviewing each other's code.

conducted by senior developers or project leads

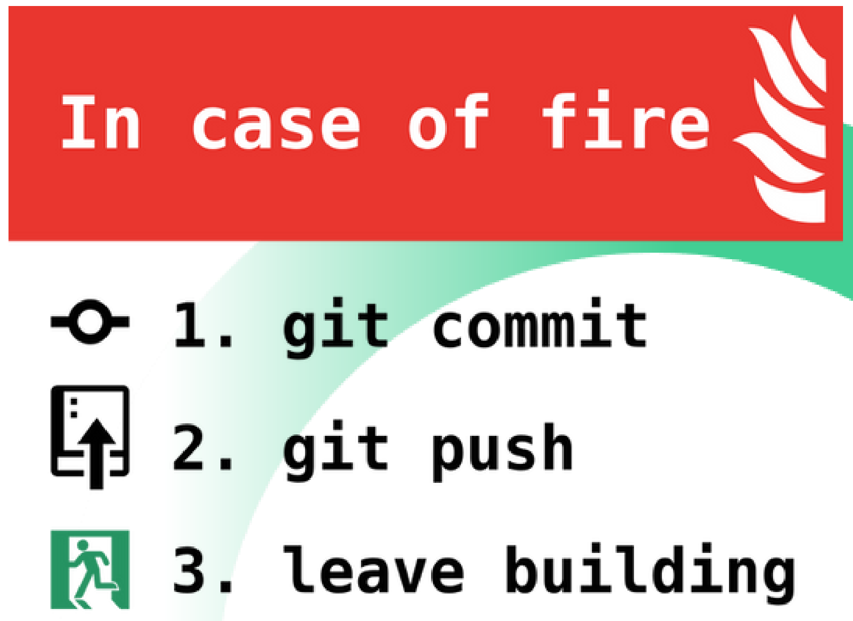
real-time collaborative technique where two developers work together on the same workstation. One acts as the "driver" (writing code), while the other is the "navigator" (reviewing and guiding).



- A **code review** is a systematic process in software development where one or more developers examine another developer's code to:
  - Identify potential bugs, errors, or vulnerabilities.
  - Ensure the code adheres to design standards, coding guidelines, and best practices.
  - Improve code quality, readability, and maintainability.
  - Share knowledge and ensure consistency across the team.

VERSION CONTROLLING

- Code should always be version controlled.
- Allow developers to change and improve the code freely without being afraid of breaking the code.
- Let multiple developers to collaborate on the same code base.
- Remove the single point of failure in code base.
- Use multiple branches tags for maintaining the code base.

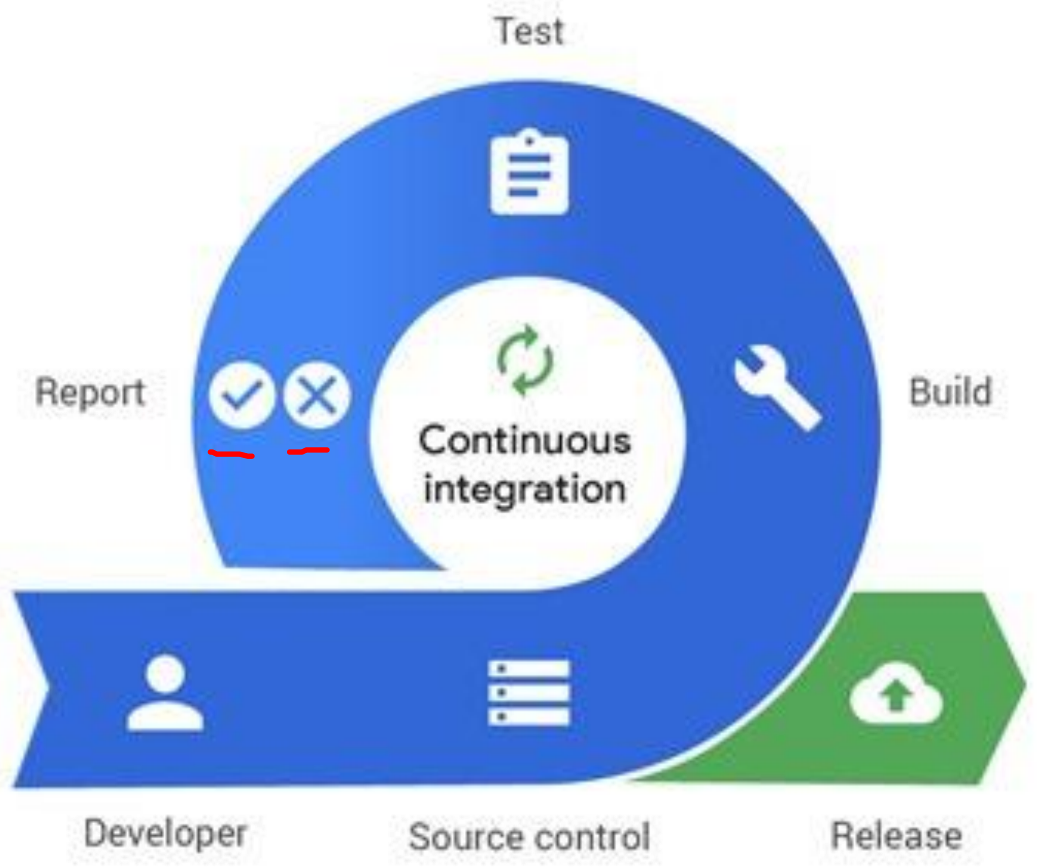

In case of fire

1. git commit
2. git push
3. leave building

CONTINUOUS INTEGRATION

Automatically building/testing code on every change.

- Continuous integration is a development practice.
- Developers need to check-in the code to a shared repository several times a day.
- Each checking is verified by an automated build.
- This allows developers to detect issues early and fix them without a delay.



Guidelines:
Run tests on every push (GitHub Actions, Jenkins).
Fail fast—reject broken builds.
Automate deployments (CD) after CI passes.

THAT'S ALL FOLKS !

ANY QUESTIONS ?