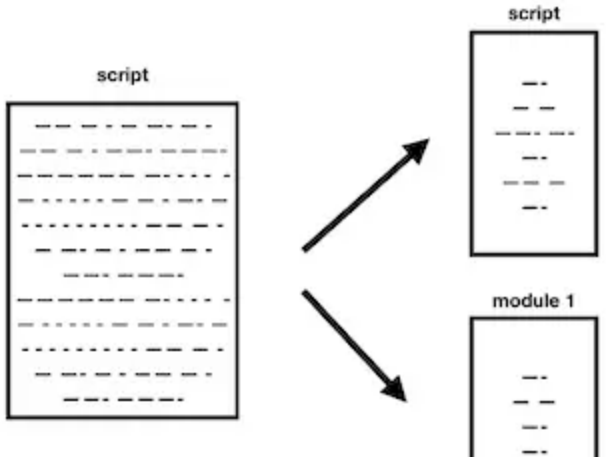
01. OOP and Software Design

Two Questions

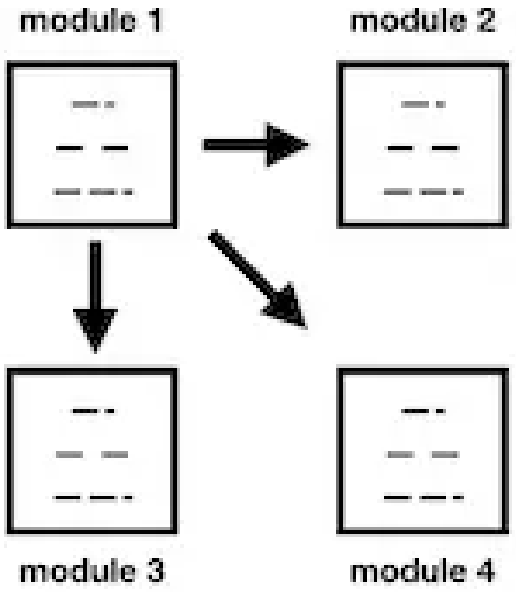
1. Why OOP?
2. Why Software Design?

Starting Small

- When we start programming, we start small.
- We write a small script to solve the given problem.

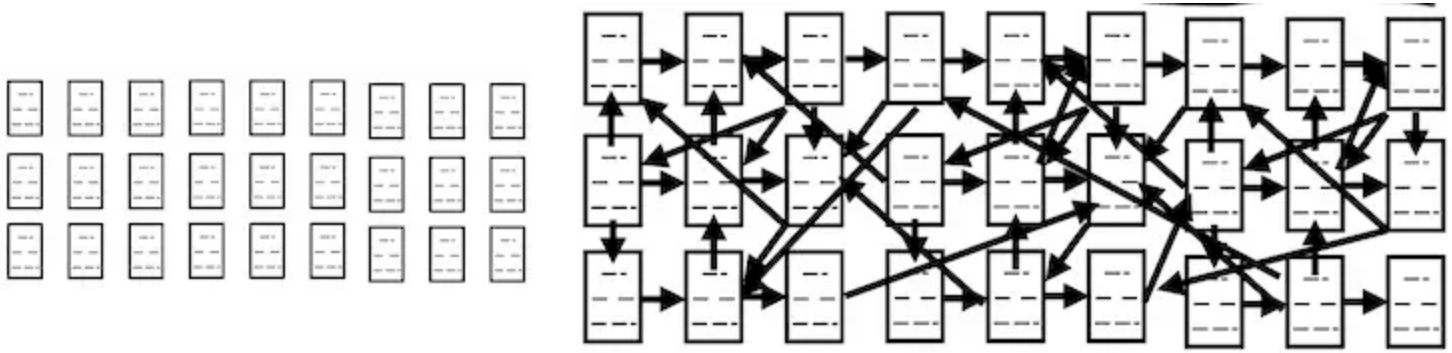


- However, the script becomes hard to read and manage.
- So we split the script into smaller parts.



Growing Complexity

- We add more and more modules.
- The couplings and dependencies among modules increase over time.



- Very quickly, we find that managing the modules becomes costly.
- We cannot respond to users' requests.
- We introduce more bugs when we fix bugs.
- We pay the price of **ad-hoc development** (development without design).

The Problems - Changes and Complexity

- Everything changes in programming because:
  - Clients ask for new features
  - We have new tools, change
  - We have library updates
  - and on...

Code always grows
Code always changes

- The changes add more and more complexity to programming.
- When we focus on making code, we can never manage the complexity of the changes.

Real-World Software is Inherently Complex

- Software is the most complex system that humans have ever experienced.
- **Managing complexity** is the core of successful software development.

Challenge: How do we manage the complexity?

OOP as a rescue

- OOP provides the tools and rules for managing complexity.
- OOP is the best way to manage complexity in commercial software.

However

- OOP may not be the best way to make any programs.
- Especially when we make small programs, OOP can be overkill.
- However, without OOP, it is not possible to build complex modern software.

We can Manage Complexity 1

- In the real world, we perceive anything as an object, not a combination of parts.
- We call this idea "abstraction."

Abstraction

- OOP's class/object is a smart way to manage modules and interfaces.
- We think of a car as a single entity, not as thousands of individual parts.

We can Manage Complexity 2

- OOP allows us to use the same method name for different objects.
- We call this idea polymorphism.

Polymorphism

- Imagine you ask to "speak":
  - Sarah (human student): She says, "Hello, my name is Sarah."
  - Rover (class pet dog): He barks "Woof! Woof!"
- They use the same command ("speak"), but each responds according to their nature.

We can Manage Complexity 3

- OOP's **inheritance** allows us to avoid making any objects from scratch.
- Instead, we can modify (override) or extend existing features.
- We call this idea **"inheritance."**

Inheritance

- All vehicles share common features (engine, wheels, steering, brakes).
  - Sports Car: Inherits base vehicle features + adds turbo engine, racing tires
  - Truck: Inherits base vehicle features + adds cargo bed, towing capacity

We can Manage Complexity 4

- OOP protects data from the outside world by providing **methods** that can access the data.
- We call this idea **"encapsulation."**

Encapsulation

- In the real world, when we take pills, pharmaceutical companies encapsulate the medicine to protect the pills until they reach our stomach.

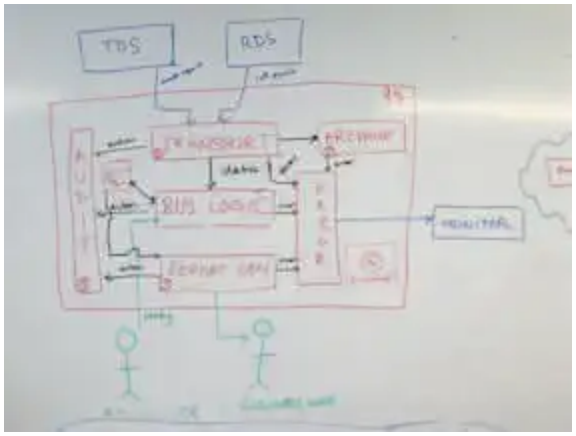
> *Challenge*: How do we manage the complexity?

- We can manage complexity using the OOP/APIE principles
  - In our daily lives, we use APIE principles to simplify our lives.
  - Likewise, we use APIE to manage complexity.

Challenge: How do we sense and prevent the complexity?

Software Design

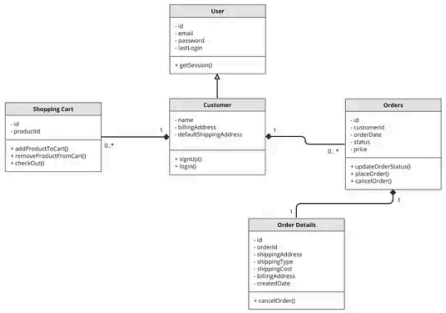
- Software design is about **modules** and **interfaces** to manage complexity.
- In other words, when we design software, we design modules and the interfaces of the modules.



- That is the reason why we use diagrams to describe software design.

UML as a Design Language

- Software engineers developed the unified diagram to express the modules and interfaces.
- It is called **UML** - Unified Modeling Language.



- The **three-section box** indicates a module.
- The **the methods/fields with the +/-/# sign** indicate interfaces.

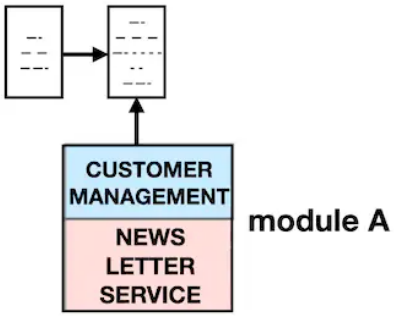
OOP as a Tool for Software Design

- **OOP classes** are modules.
- **OOP public functions** are interfaces. functions.
- OOP provides the building blocks for good software design.

Making modules and interfaces is not enough

- Clients ask us to build a newsletter service.
- We design (create modules with interfaces) to implement a feature.
- However, we can make *bad* design easily.

- We may create one module to manage customer **and** newsletter service because we need the two features.



- This is **bad**, as it violates the **SRP** (we will discuss it soon) principle.
- Mixing unrelated responsibilities in one module will add complexity that is hard to manage.

Real-World Example: Bridge Coupling

- A city renovates a bridge, and workers cut the bridge without knowing that the bridge is **coupled** with the power line.

- This simple cutting action blackouts the city.
- This (real) incident happened because the bridge designers violated the **"low coupling, high cohesion"** rule.
- We need software `design rules` and to follow them!

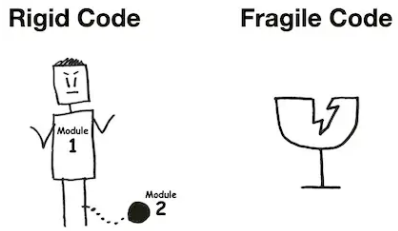
Change is Inevitable, But

- Change is inevitable in software development, but it is dangerous when we don't have a design and don't follow the rules.

- We **cannot manage complexity** with an ad-hoc design (just making modules and interfaces).
- When we violate software design rules, we cannot meet clients' requests due to the added complexity.

Managing Complexity from Changes

- We should avoid **rigid code**, which is hard to change.
- We also should avoid **fragile code**, which is easy to break.

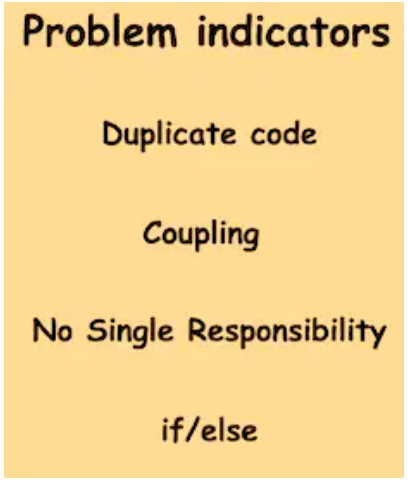


- To avoid rigid and fragile code, we need to follow **software design rules**.
- In other words, what rule should we apply to write code that adapts to change gracefully?

Code smell

- We can sense that something is wrong with the problem indicators.
- Software engineers call that **"code smell."**

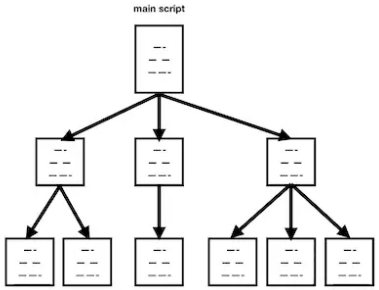
Problem indicators



- We should feel uncomfortable whenever we see these indicators (code smell) in the code.

- The problem indicators on the left side are well-known code smells.
- We already know the solutions:
  - Software Design (modules + Interfaces) and the APIE.
  - The APIE rules can guide us in making good modules and interfaces.

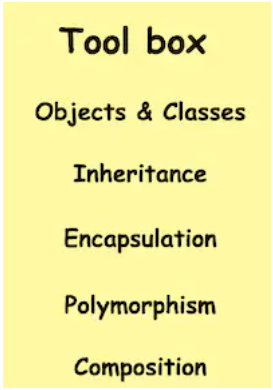
Making modules as the first step



- Organizing code into **modules** (and interfaces) is the first step to designing software.

- When code is organized, we can smell and refactor the code smell better.
- In other words, we can manage complexity better using **modularization**.
- However, we need rules to guide the modularization.

Solution toolbox

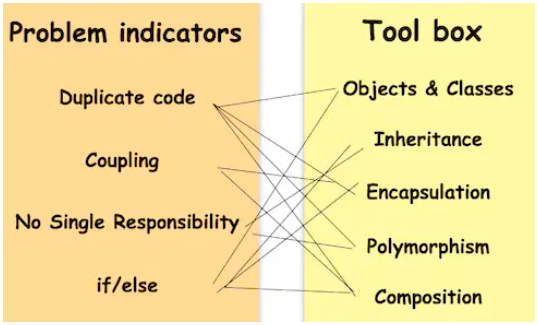


- We have a toolbox to address the issues when we use OOP/APIE.

APIE + C

- They are **Abstractions** in the form of objects and classes, **polymorphism**, **inheritance**, and **encapsulation** (in short, APIE).
- We have one more tool: **Composition**.

Tool-Problem Connection



The `Toolbox` from OOP and software design can address these `Problem Indicators` (code smell).

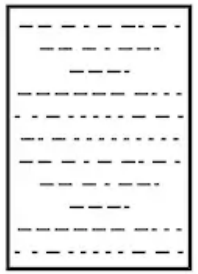
Problem Solver

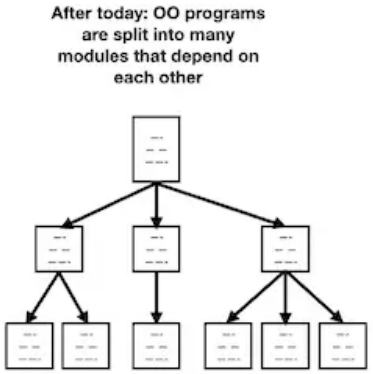
- Good problem solvers/software engineers can sense smelly code.
- They can come up with a better design to prevent code smells by applying software design rules.
- It's an art, and you can work with LLM: You design and AI code/test.

> *Challenge*: How do we sense and prevent the complexity?

- We identify a set of problem indicators we call `code smell`.
- Software Design removes the code smell guided by OOP/APIEC.

Coder to Designer





- We need to transform ourselves from coders to designers.
- It's important, as coders have no chance.

Topic Questions

- What is the challenge of developing software?