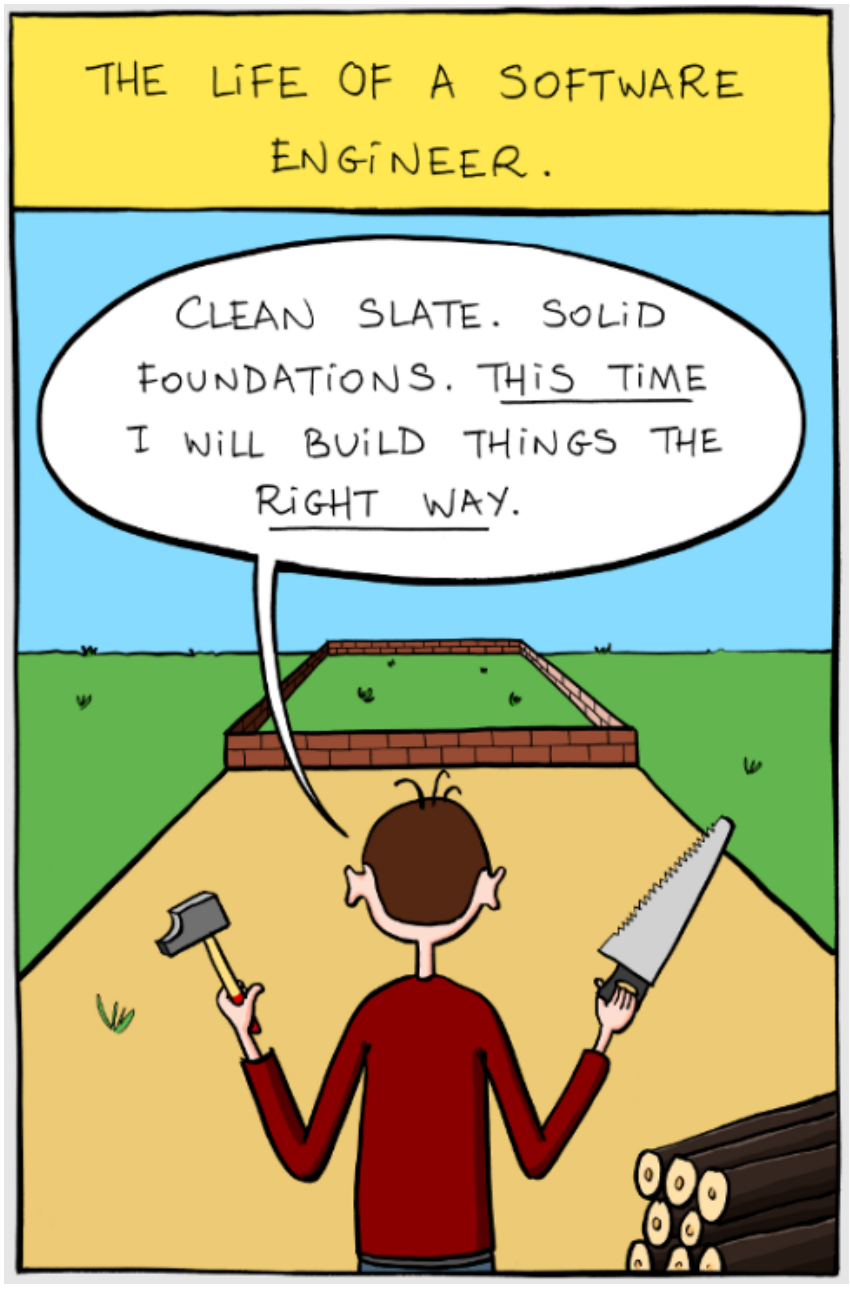
AMSS Lecture 3: Design Patterns (I)

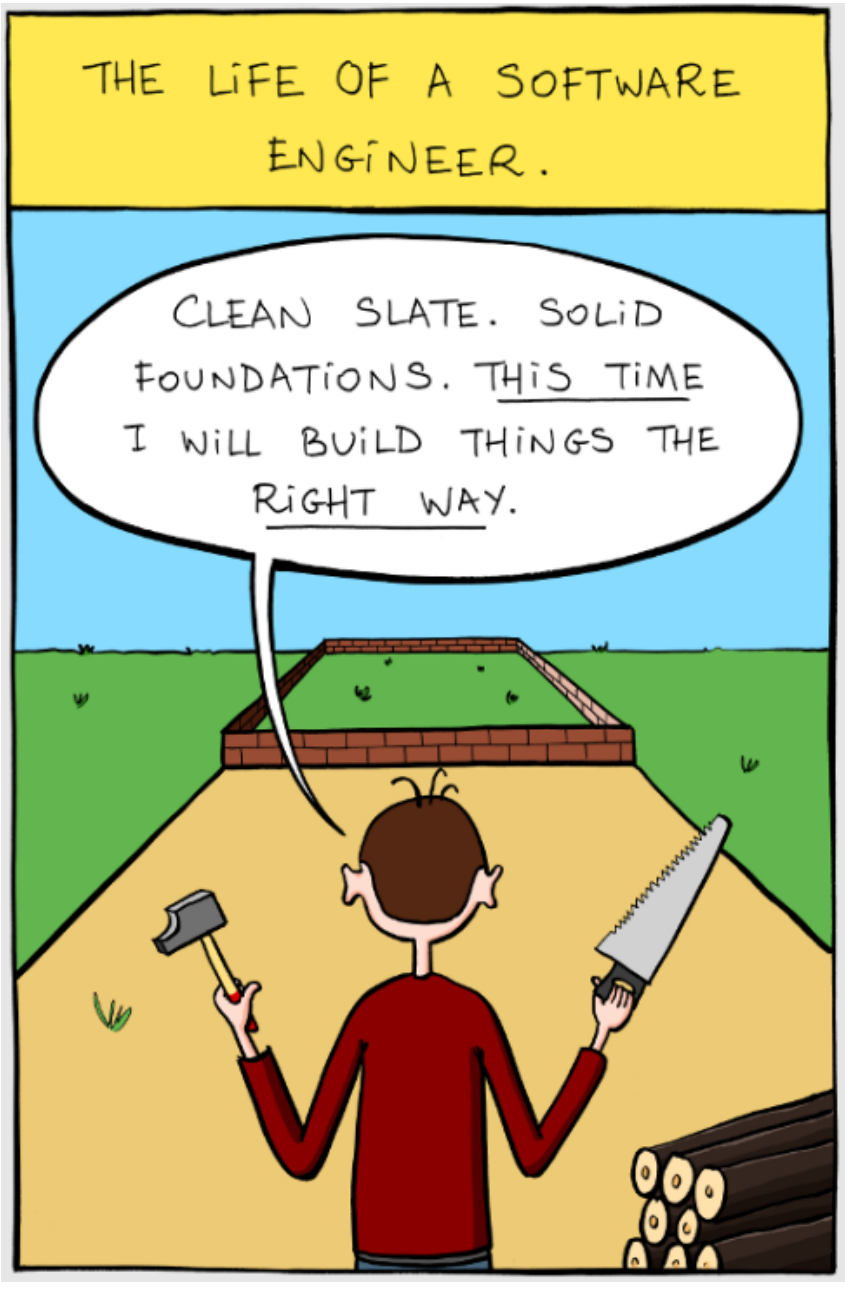
Virgil-Nicolae Șerbănuță

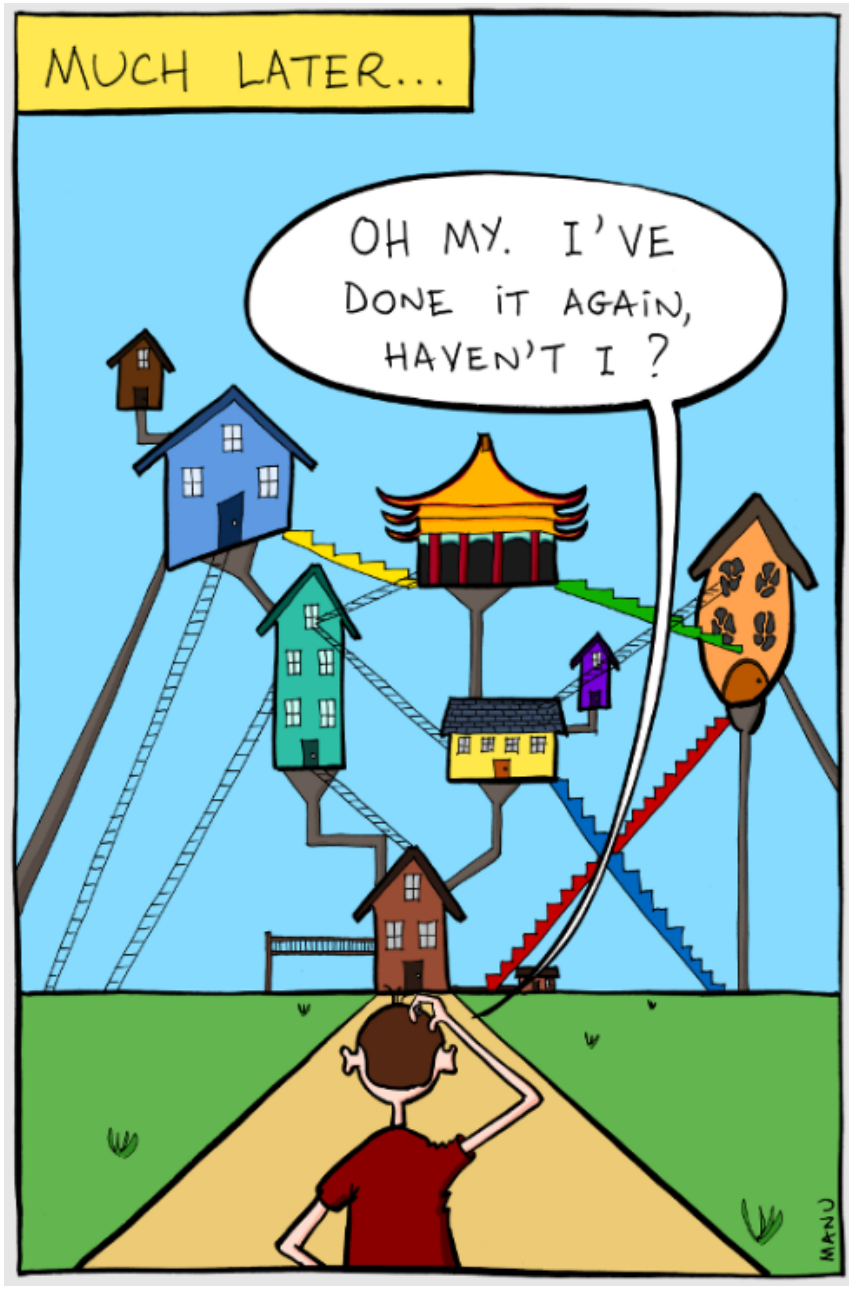
2025

Agenda

1. What are Design Patterns?
2. Classification of Patterns
3. **Iterator Pattern**
4. **Builder Pattern**
5. **Singleton Pattern**
6. Wrap-up


THE LIFE OF A SOFTWARE ENGINEER.
CLEAN SLATE. SOLID FOUNDATIONS. THIS TIME I WILL BUILD THINGS THE RIGHT WAY.


THE LIFE OF A SOFTWARE ENGINEER.
CLEAN SLATE. SOLID FOUNDATIONS. THIS TIME I WILL BUILD THINGS THE RIGHT WAY.


MUCH LATER...
OH MY. I'VE DONE IT AGAIN, HAVEN'T I ?
MANU

What Are Design Patterns?

Definition

Reusable solutions to common software design problems.

Origin

Popularized by the "Gang of Four" (Gamma, Helm, Johnson, Vlissides, 1994).

Purpose

- Provide shared vocabulary
- Improve code maintainability
- Promote reusability and clarity

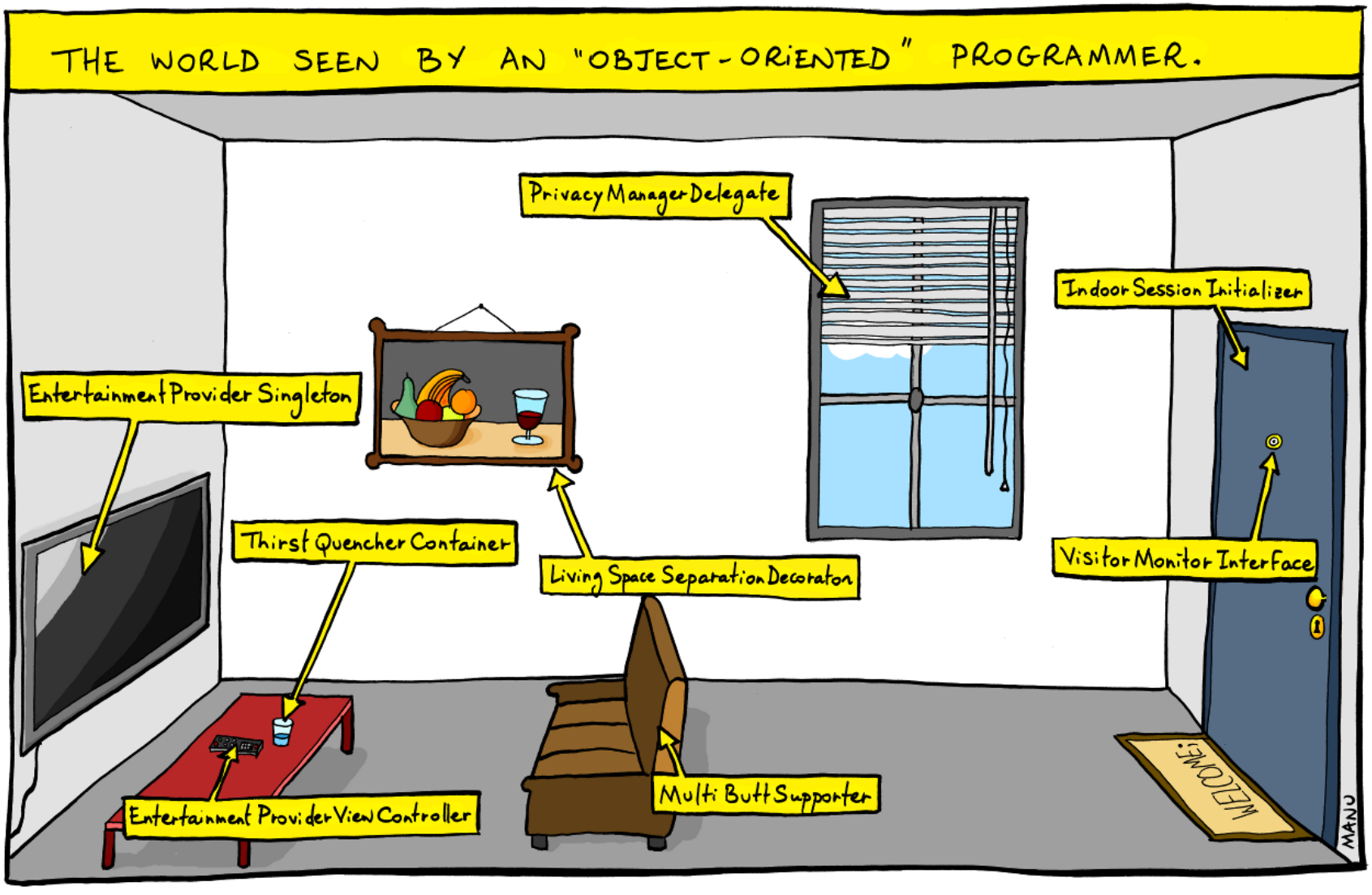
Example

Instead of reinventing how to traverse a collection, we apply the **Iterator** pattern.

Pattern Classification

Design patterns are typically grouped into three main categories:

Category	Description	Example Patterns
Creational	How objects are created	Singleton, Builder, Factory Method
Structural	How classes and objects are composed	Adapter, Bridge, Decorator
Behavioral	How objects interact and communicate	Iterator, Observer, State


THE WORLD SEEN BY AN "OBJECT-ORIENTED" PROGRAMMER.
PrivacyManagerDelegate
IndoorSessionInitializer
EntertainmentProviderSingleton
ThirstQuencherContainer
LivingSpaceSeparationDecoraton
VisitorMonitorInterface
EntertainmentProviderViewController
MultiButtSupporter
WELCOME
MANU

Iterator Pattern

Type

Behavioral pattern

Intent

Provide a way to access elements of a collection sequentially without exposing its internal structure.

Problem Solved

How to traverse a collection (e.g., list, tree, array) without knowing its implementation?

Solution

Define an `Iterator` interface with methods like `hasNext()` and `next()`.

Iterator Pattern code example

Source file

```
// Step 1: Create the Iterator interface
interface Iterator {
    boolean hasNext();
    Object next();
}

// Step 2: Create the Container interface
interface Container {
    Iterator getIterator();
}

// Step 3: Create a concrete class implementing Container
class NameRepository implements Container {
    private String[] names = {"Alice", "Bob", "Charlie", "I

    @Override
    public Iterator getIterator() {
```

Iterator Pattern Exercise

Exercise

Modify 3 lines in the example above to make the iterator a reverse iterator.

Question

How does this custom Iterator differ from Java's built-in `java.util.Iterator`, and when might you still implement your own?

Builder Pattern

Type

Creational pattern

Intent

Separate the construction of a complex object from its representation, so the same construction process can create different representations.

Problem Solved

How can we construct complex objects step by step while keeping the construction logic separate from the representation?

Solution

Use a *Builder* class to encapsulate object creation in multiple steps.

Builder Pattern (concrete example)

Task

Design a Computer class that represents a configurable computer system. The goal is to let users "build" a computer step-by-step, choosing which components to include.

A valid computer may include:

- CPU (e.g., "Intel i9", "AMD Ryzen 7")
- GPU (e.g., "NVIDIA RTX 4090", "AMD Radeon RX 7800")
- RAM (in GB)
- Storage (in GB)

You should be able to build objects fluently, like this:

```
Computer gamingPC = new Computer.Builder()
        .setCPU("Intel i9")
        .setGPU("NVIDIA RTX 4090")
        .setRAM(32)
        .setStorage(2000)
        .build();

System.out.println(gamingPC);
```

Builder Pattern (concrete example solution)

Source file

```
// Product class
class Computer {
    private String CPU;
    private String GPU;
    private int RAM;
    private int storage;

    // Private constructor - use Builder instead
    private Computer(Builder builder) {
        this.CPU = builder.CPU;
        this.GPU = builder.GPU;
        this.RAM = builder.RAM;
        this.storage = builder.storage;
    }

    @Override
    public String toString() {
```

Bulder Pattern exercise

Design a Pizza class that represents a customizable pizza order, using the Builder Pattern.

Your pizza should have:

- A size (e.g., Small, Medium, Large)
- A crust type (e.g., Thin, Thick, Stuffed)
- A list of toppings (e.g., Cheese, Pepperoni, Mushrooms)
- A flag for extra cheese

The goal is to make object creation flexible and readable, like this:

```
Pizza pizza = new Pizza.Builder()
        .setSize("Large")
        .setCrust("Stuffed")
        .addTopping("Pepperoni")
        .addTopping("Mushrooms")
        .setExtraCheese(true)
        .build();
```