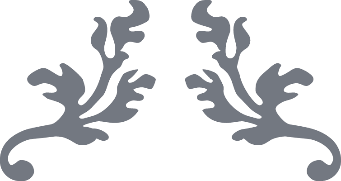


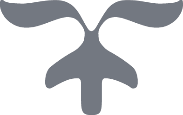
**DHA SUFFA UNIVERSITY**

Department of Computer Science



# Software Design & Architecture (SE-2403)

Spring 2022



# SOFTWARE DESIGN PRINCIPLES

## Features of Good Design

Before we proceed to the actual patterns, let’s discuss the process of designing software architecture: things to aim for and things you’d better avoid.

**Code reuse**

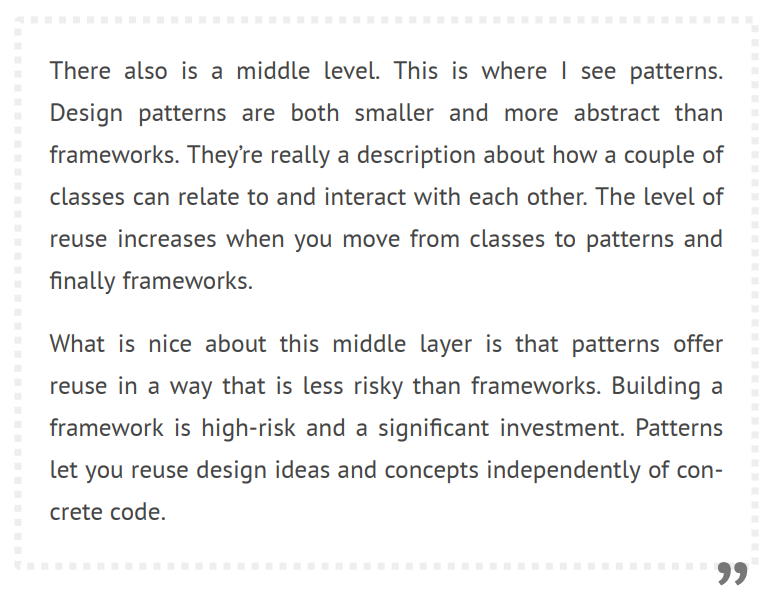
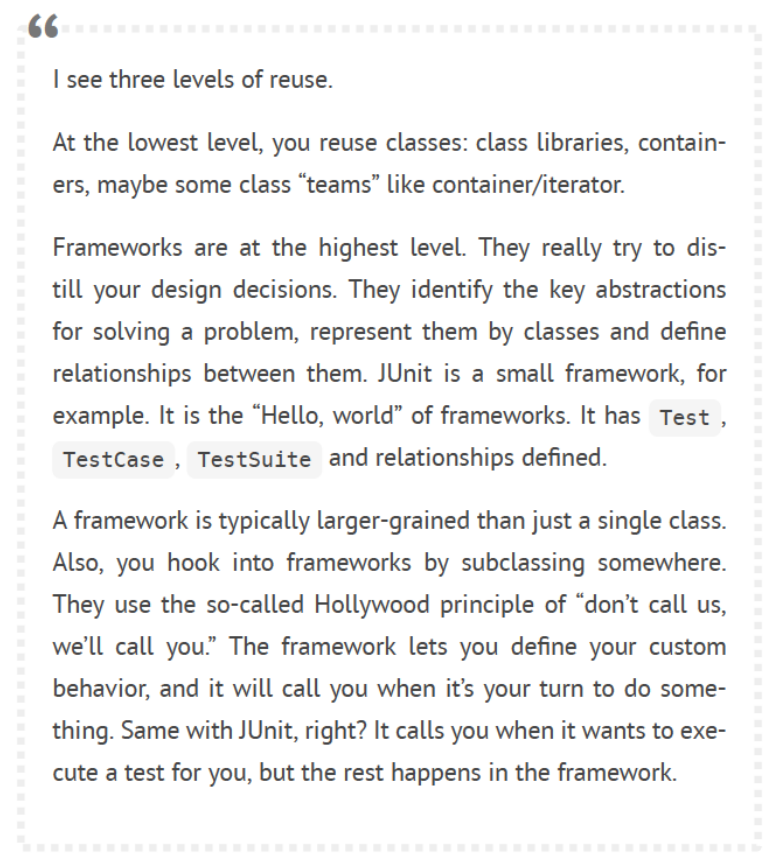
Cost and time are two of the most valuable metrics when developing any software product. Less time in development means entering the market earlier than competitors. Lower development costs mean more money is left for marketing and a broader reach to potential customers.

**Code reuse** is one of the most common ways to reduce development costs. The intent is pretty obvious: instead of developing something over and over from scratch, why don’t we reuse existing code in new projects?

The idea looks great on paper, but it turns out that making existing code work in a new context usually takes extra effort. Tight coupling between components, dependencies on concrete classes instead of interfaces, hardcoded operations—all of this reduces flexibility of the code and makes it harder to reuse it.

Using design patterns is one way to increase flexibility of soft- ware components and make them easier to reuse. However, this sometimes comes at the price of making the components more complicated.

Here's a piece of wisdom from **Erich Gamma**, one of the founding fathers of design patterns, about the role of design patterns in code reuse:



**Extensibility**

**Change** is the only constant thing in a programmer’s life.

* You released a video game for Windows, but now people ask for a macOS version.
* You created a GUI framework with square buttons, but several months later round buttons become a trend.
* You designed a brilliant e-commerce website architecture, but just a month later customers ask for a feature that would let them accept phone orders.

Each software developer has dozens of similar stories. There are several reasons why this happens.

First, we understand the problem better once we start to solve it. Often by the time you finish the first version of an app, you’re ready to rewrite it from scratch because now you understand many aspects of the problem much better. You have also grown professionally, and your own code now looks like crap.

Something beyond your control has changed. This is why so many dev teams pivot from their original ideas into something new. Everyone who relied on Flash in an online application has been reworking or migrating their code as browser after browser drops support for Flash.

The third reason is that the goalposts move. Your client was delighted with the current version of the application, but now sees eleven “little” changes he’d like so it can do other things he never mentioned in the original planning sessions. These aren’t frivolous changes: your excellent first version has shown him that even more is possible.

There’s a bright side: if someone asks you to change something in your app, that means someone still cares about it.

That’s why all seasoned developers try to provide for possible future changes when designing an application’s architecture.

## Design Principles

What is good software design? How would you measure it? What practices would you need to follow to achieve it? How can you make your architecture flexible, stable and easy to understand?

These are the great questions; but, unfortunately, the answers are different depending on the type of application you’re building. Nevertheless, there are several universal principles of software design that might help you answer these questions for your own project. Most of the design patterns are based on these principles.

##### Encapsulate What Varies

Identify the aspects of your application that vary and separate them from what stays the same.

The main goal of this principle is to minimize the effect caused by changes.

Imagine that your program is a ship, and changes are hideous mines that linger under water. Struck by the mine, the ship sinks.

Knowing this, you can divide the ship’s hull into independent compartments that can be safely sealed to limit damage to a single compartment. Now, if the ship hits a mine, the ship as a whole remains afloat.

In the same way, you can isolate the parts of the program that vary in independent modules, protecting the rest of the code from adverse effects. As a result, you spend less time getting the program back into working shape, implementing and testing the changes. The less time you spend making changes, the more time you have for implementing features.

Encapsulation on a method level

Say you’re making an e-commerce website. Somewhere in your code,

there’s a getOrderTotal method that calculates total for the order,

including taxes.

We can anticipate that tax-related code might need to change in the future. The tax rate depends on the country, state or even city where the customer resides, and the actual formula may change over time due to new laws or regulations. As a result, you’ll need to change the getOrderTotal method quite often.

But even the method’s name suggests that it doesn’t care about *how* the tax is calculated.

1

2

3

4

5

6

7

8

9

10

11

**method** getOrderTotal(order) **is**

total = 0

**foreach** item in order.lineItems

total += item.price \* item.quantity

**if** (order.country == "US")

total += total \* 0.07 // US sales tax

**else if** (order.country == "EU"):

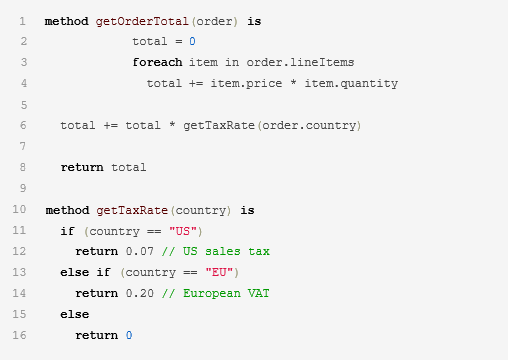
total += total \* 0.20 // European VAT

**return** total

*BEFORE: tax calculation code is mixed with the rest of the method’s code.*

You can extract the tax calculation logic into a separate method, hiding it from the original method.

Tax-related changes become isolated inside a single method. Moreover, if the tax calculation logic becomes too complicated, it’s now easier to move it to a separate class.



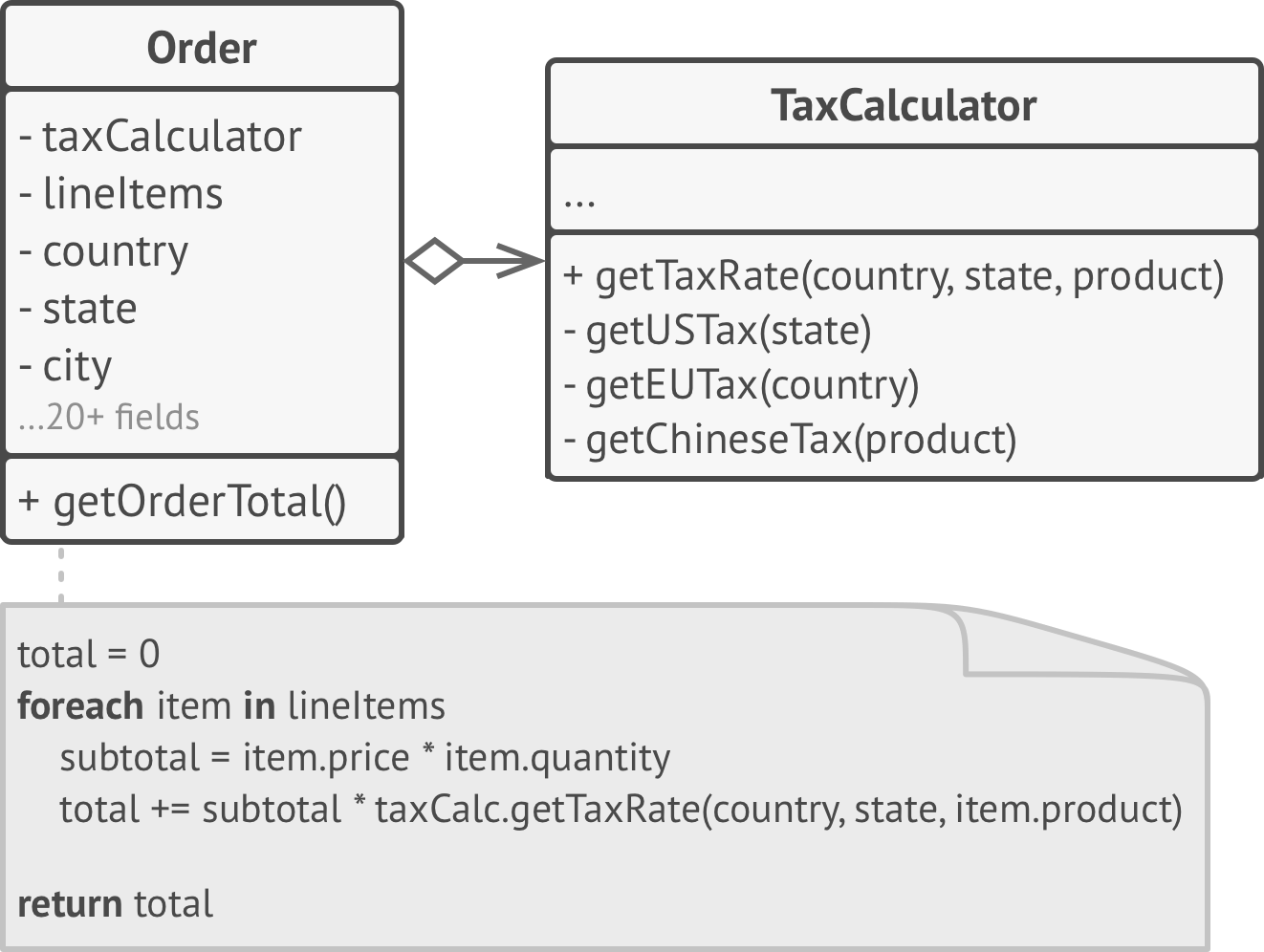
*AFTER: you can get the tax rate by calling a designated method*

**Encapsulation on a class level**

Over time you might add more and more responsibilities to a method which used to do a simple thing. These added behaviors often come with their own helper fields and methods that eventually blur the primary responsibility of the containing class. Extracting everything to a new class might make things much clearer and simpler.







*AFTER: tax calculation is hidden from the order class.*