Design – Solid Principles

<https://www.freecodecamp.org/news/solid-principles-for-programming-and-software-design/>

The Single Responsibility Principle states that each class should have only one reason to change.

**“A class should have only one reason to change”  
-Robert C. Martin, *Clean Code***

Checking the cohesion of a class allows you to detect if there is an SRP violation. If you have a highly cohesive class, it likely means that the design does not violate the SRP. If you are dealing with a low cohesive class, split it into several classes to achieve high cohesion.

A high cohesive class is close to the maximum cohesive class. You can say that almost every method uses almost every field.

The four methods should be located in different classes.

**BEFORE**

public class Order {

public Double calculateTax() {...}

public void applyPromotions() {...}

public Stock checkItemsAvailability() {...}

public String describeOrder() {...}

}

**AFTER**

public class ... {

public Double calculateTax() {...}

}

public class ... {

public void applyPromotions() {...}

}

public class ... {

public Stock checkItemsAvailability() {...}

}

public class ... {

public String describeOrder() {...}

}

Why is mixing responsibilities bad?

Conclusion ->

A class should be small, and it should have only one reason to change.

The source of any axis of change is a user or an actor.

High cohesion is a helpful additional method to identify violations of the SRP.  
Mixing responsibilities leads to a fragile and highly coupled system.

Open Closed Principle

changing one component does not require a modification to the other.

The Open-Closed Principle can help. When you apply this principle, you aim to create flexible code that allows for future changes that do not impact existing code

You should be able to change one feature without altering others.

**“Software entities should be open for extension and closed for modification.”**

**“What [the OCP] means is that you should strive to get your code into a position such that, when behavior changes in expected ways, you don't have to make sweeping changes to all the modules of the system. Ideally, you will be able to add the new behavior by adding new code and changing little or no old code.”**

**Conclusion**

Create an abstraction or use a design pattern to achieve the OCP

Apply TDD to maintain a balance between overdesign and the required abstraction for the OCP

Remember that you will never achieve the OCP completely; you must apply it strategically

**Tell, Don't Ask Principle**

Data encapsulated by a class should be the responsibility of its class instead of being passed to another class to process it.

When you apply this principle correctly, your programs can **achieve better modularity** and have **a clear separation of concerns.**

# Procedural Code vs Object-Oriented Code

**“Procedural code gets information, then makes decisions. Object-oriented code tells objects to do things.”**