SOLID

--------

Single Responsibility Principle (SRP)

* A class has responsibility over a single functionality
* There is only one single reason for a class to change
* responsibility to a separate class and methods that can be maintained and updated independently

Open/Closed Principle (OCP)

* being able to change the behavior of a method or class without having to modify the code.
* You can introduce new implementations by passing them as an argument without modifying the body of this method.
* Hence, it is now open for extension and closed for modification.
* This reduces the scope for introducing new bugs because it minimizes cascading changes required to parts of code that have already been implemented and tested.
* In other words, old code still works and is untouched.

Liskov Substitution Principle (LSP)

1. Every subclass or derived class should be substitutable for their base or parent class
2. Preconditions cannot be strengthened in a subtype

For example, all our Importer implementations have the precondition that the file being imported exists and is readable. As a result, the importFile method has validation code before any Importer is invoked

1. Postconditions cannot be weakened in a subtype

- Postconditions are things that have to be true after some code has run.

- So if the parent has some kind of side effect or returns some value, then the child must do so as well.

- For example, after importFile() has run, if the file in question is valid it must be in the list of documents returned by contents()

1. Invariants of the supertype must be preserved in a subtype

- An invariant is something that never changes

- we want to make sure that any invariants that are expected to be maintained by the parent class should also be maintained by the children

1. The History Rule

- the child class shouldn’t allow state changes that your parent disallowed.

- So, in our The Document Management System program we have an immutable Document class. In other words, once it has been instantiated you can’t remove, add, or alter any of the attributes.

- You shouldn’t subclass this Document class and create a mutable Document class.

- This is because any user of the parent class would expect certain behavior in response to calling methods on the Document class. If the child were mutable, it could violate callers’ expectations about what calling those methods does.

Interface Segregation Principle (ISP)

* Interface should not force classes to what they can’t do
* Large interfaces should be divided into small one
* It makes the case that no class should be forced to depend on methods it does not use because this introduces unnecessary coupling
* The ISP focuses on the user of an interface rather than its design.
* In other words, if an interface ends up very large, it may be that the user of that interface sees some behaviors it doesn’t care for, which causes unnecessary coupling.
* To provide a solution that meets the Interface Segregation Principle, we are encouraged to separate out concepts in smaller interface that can evolve separately. This idea essentially promotes higher cohesion.

Dependency Inversion Principle (DIP)

* Components should depend on abstractions not on concretions

**CHAPTER 2**

You’ll write a program to analyze bank statements in order to help people under‐ stand their finances better. This will help you to learn more about core object oriented design techniques such as Single Responsibility Principle (SRP), coupling, and cohesion.

KISS

* You can “Keep It Short and Simple” (KISS) and have the application code in one single class

few corner cases that are always good to think about when writing production-ready code

1. What if the file is empty?
2. What if parsing the amount fails because the data was corrupted?
3. What if a statement line has missing data?

code maintainability

* It should be simple to locate code responsible for a particular feature.
* It should be simple to understand what the code does.
* It should be simple to add or remove a new feature.
* It should provide good encapsulation. In other words, implementation details should be hidden from a user of your code so it is easier to understand and make changes.

So how do you apply SRP?

* It is clear that the main class has multiple responsibilities that can be broken down individually:

1. Reading input

2. Parsing the input in a given format

3. Processing the result

4. Reporting a summary of the result

Cohesion

* how strongly related responsibilities of a class or method are
* high cohesion, which means that the code is easier for others to locate, understand, and use
* class BankTransactionCSVParser is highly cohesive. In fact, it groups together two methods that are related to parsing CSV data.

Class-Level Cohesion

* Functional

1. group the methods functionally
2. This is generally a good way to achieve high cohesion because the methods are working together
3. The danger with functional cohesion is that it may be tempting to have a profusion of overly simplistic classes grouping only a single method.
4. Going down the road of overly simplistic classes adds unnecessary verbosity and complexity because there are many more classes to think about.

* Informational

1. work on the same data or domain object
2. Ex: CRUD operations
3. The downside of this approach is that this kind of cohesion can group multiple concerns together,
4. which introduces additional dependencies for a class that only uses and requires some of the operations.

* Utility

1. group different unrelated methods inside a class
2. This hap‐pens when it is not obvious where the methods belong so you end up with a utility class that is a bit like a jack of all trades.

* Logical

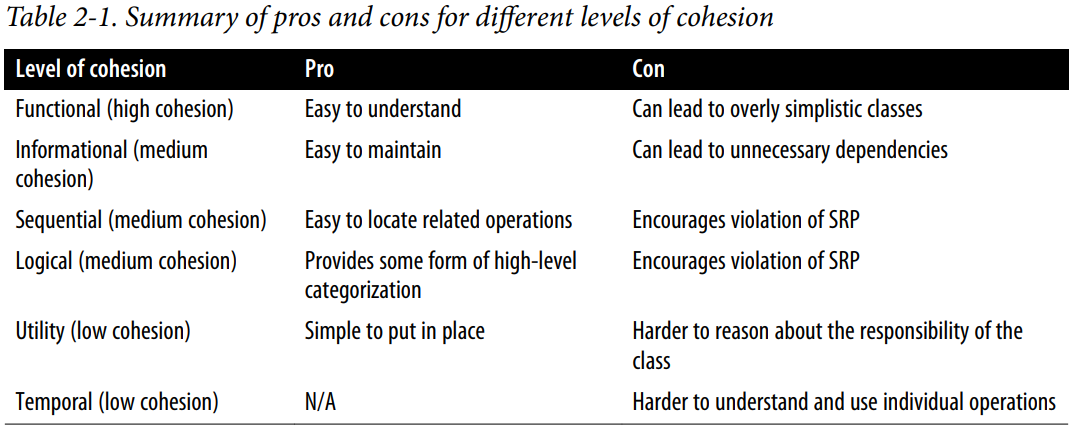
1. In fact, the methods are logically categorized to do “parsing.” Ex: CSV, JSON, and XML.
2. However, they are different by nature and each of the methods would be unrelated. Grouping them would also break the SRP

* Sequential

1. grouping the methods so that they follow a sequence of input to output
2. EX: read a file, parse it, process it, and save the information
3. AVOID -> breaking the SRP

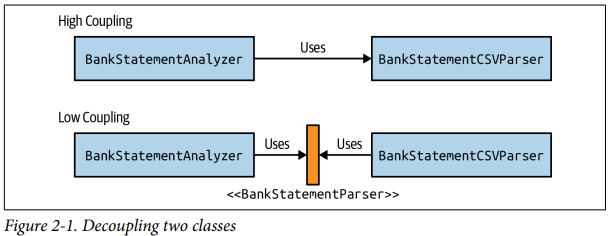
* Temporal

1. performs several operations that are only related in time
2. e.g., connecting and closing a database connection
3. The initialization and the other operations are unrelated, but they have to be called in a specific order in time



Coupling

* how dependent you are on other classes
* achieve low coupling
* To understand what coupling is, think about a clock. There is no need to know how a clock works to read the time, so you are not dependent on the clock internals. This means you could change the clock internals without affecting how to read the time. Those two concerns (interface and implementation) are decoupled from one another.



**CHAPTER 3**

In this chapter you learn how to extend the code from Chapter 2, adding more features, using the Strategy Design pattern, the Open/Closed Principle, and how to model failures using exceptions.

1. He’d like to also be able to search for specific transactions. For example, you should be able to return all the bank transactions in a given date range or for a specific category.

2. Mark would like to be able to generate a report of summary statistics for his search into different formats such as text and HTML.

Clearly this approach exhibits several downsides:

**findTransactionsGreaterThanEqual**

**findTransactionsInMonth**

**findTransactionsInMonthAndGreater**

1. Your code will become increasingly complicated as you have to combine multiple properties of a bank transaction.
2. The selection logic is coupled to the iteration logic, making it harder to separate them out.
3. You keep on duplicating code.

SOLUTION 🡪 **bankTransactionFilter** This is where the Open/Closed principle comes in. It promotes the idea of being able to change the behavior of a method or class without having to modify the code.

* Implicit vs Explicit API
* The return type **void** is not useful and is difficult to reason about. You don’t know what is returned
* Returning void makes it very hard to test the result with assertions
* Typically, errors due to business logic validation (e.g., wrong format or arithmetic) should be - **unchecked exceptions**, as they would add a lot of try/catch clutter in your code. In addition, system errors (e.g., disk ran out of space) should also be unchecked exceptions as **there’s nothing the client can do**

**VALIDATION LOGIC**

1. You don’t have to duplicate the validation logic when you need to reuse it.
2. You get confidence that different parts of your system validate the same way.
3. You can easily unit test this logic separately.
4. It follows the SRP, which leads to simpler maintenance and program comprehension.

**NOTIFICATION PATTERN (p.49)**

The Notification pattern aims to provide a solution for the situation in which you are using too many unchecked exceptions. The solution is to introduce a domain class to collect errors.

* Do not use exceptions for control flow eg., while, for loop
* Exception alternative 🡪 Null, Optional<T>, Try<T>

**TAKE AWAYS**

1. The Open/Closed Principle promotes the idea of being able to change the behavior of a method or class without having to modify the code.
2. The Open/Closed Principle reduces fragility of code by not changing existing code, promotes reusability of existing code, and promotes decoupling, which leads to better code maintenance.
3. God interfaces with many specific methods introduce complexity and coupling.
4. An interface that is too granular with single methods can introduce the opposite of cohesion.
5. You should not be worried about adding descriptive method names to help read‐ ability and comprehension of your API.
6. Returning void as a result of an operation makes it difficult to test its behavior.
7. Exceptions in Java contribute to documentation, type safety, and separation of concerns.
8. Use checked exceptions sparingly rather than the default as they can cause signif‐ icant clutter.
9. Overly specific exceptions can make software development unproductive.
10. The Notification Pattern introduces a domain class to collect errors.
11. Do not ignore an exception or catch the generic Exception as you will lose the benefits of diagnosing the root of the problem.

**CHAPTER 4**

In this chapter we help a successful doctor manage her patient records better. This introduces concepts such as inheritance within software design, the Liskov Substitution Principle, and tradeoffs between composition and inheritance. You will also learn how to write more reliable software with automated test code.

* but here we’re talking about the more practical use of strong typing in implementing your software. Types allow us to restrict the way in which data is used. For example, our Document class is immutable: once it has been created you can’t change, or mutate, any of its attributes.
* Software design is often as much about restricting functionality that is undesirable as it is about building things that you do want
* Test hygiene means to keep your test code clean and ensure that it is main‐ tained and improved along with your codebase under test.
* domain classes allow us to name a concept and restrict the possible behaviors and values of this concept in order to improve discoverability and reduce the scope for bugs.
* Behavior not implementation

**CHAPTER 5**

You’ll learn about building a core business rules engine—a way of defining business logic that is flexible and easy to maintain. This chapter introduces the topics of test-driven development, developing a Fluent API, and the Interface Segregation Principle.

**TDD**

1. Write a test that fails

2. Run all tests

3. Make the implementation work

4. Run all tests

**Mocking** is a technique that will allow you to verify that when the method run() is executed

**Builder pattern**

**TAKEAWAYS**

• The test-driven development philosophy starts with writing some tests that are going to let you guide the implementation of the code.

• Mocking allows you to write unit tests that assert that certain behaviors are triggered.

• Java supports local variable type inferences and switch expressions.

• The Builder pattern helps design a user-friendly API for instantiating complex objects.

• The Interface Segregation Principle helps promote high cohesion by reducing dependence on unnecessary methods. This is achieved by breaking up large interfaces into smaller cohesive interfaces so that users only see what they need.

**CHAPTER 6**

Twootr is a messaging platform that enables people to broadcast short messages to other users who follow them. This chapter builds out the core of a simple Twootr system. You’ll learn how to think outside-in—to go from requirements through to the core of your application. You’ll also learn how to use test doubles to isolate and test interactions from different components within your codebase.

1. How to take a big picture description and break it down into different architectural concerns
2. How to use test doubles to isolate and test interactions from different components within your codebase
3. How to think outside-in—to go from requirements through to the core of your application domain

* **pull-based** communication style the client makes a request to the server and quer‐ ies it for information
* **push-based** communication style. This could be referred to as a reactive or event-driven communication approach. In this model, streams of events are emitted by a publisher and many subscribers listen to them. So instead of each communication being 1 to 1, they are 1 to many. This is a really useful model for sys‐ tems where different components need to talk in terms of ongoing communication patterns of multiple events. For example, if you’re designing a stock market exchange then different companies want to see updated prices, or ticks, constantly rather than having to make a new request every time they want to see a new tick.
* **Hexagonal Architecture** (Alister Cockburn)
* Whenever you have a technology-specific concern that you want to **decouple from the core of your business logic**, you introduce a port. Events from the outside world arrive at and depart from your business logic core through a port. An adapter is the technology-specific implementation code that plugs into the port. For example, we may have a port for publishing and subscribing to UI events and a WebSocket adapter that talks to a web browser.
* A good principle to help you decide might be to think of anything that is critical to the business problem that you’re solving as living inside the core of the application and anything that is technology specific or involves communicating with the outside world as living outside the core application.

**TAKEAWAYS**

• You learned about bigger-picture architectural ideas like communication styles.

• You developed the ability to decouple domain logic from library and framework choices.

• You drove the development of code in this chapter with tests going outside-in.

• You applied object-oriented domain modeling skills to a larger project

**CHAPTER 7**

The final project-based chapter in the book extends the Twootr implementation from the previous chapter. It explains the Dependency Inversion Principle and introduces bigger picture architectural choices such as event-driven and hexagonal architectures. This chapter can help you extend your knowledge of automated testing by covering test doubles, such as stubs and mocks, and also functional programming techniques.

* What we want to design is something that enables us to harness the power of a data store’s querying capability without tying the business logic to the data store in question
* The **Execute Around pattern** is a common *functional design pattern*. You may encounter a situation where you have common initialization and cleanup code that you always want to do, but parameterize different business logic that runs within the **initialization** and **cleanup** code. Eg., open/close DB connection, open/close file

**TAKEAWAYS**

• You can now decouple data storage from business logic using the Repository pattern.

• You have seen implementations of two different types of repositories within this approach.

• You were introduced to the ideas of functional programming, including Java 8 Streams.

• You’ve seen how to structure a larger project with different packages.