FAKULTAS
ILMU
KOMPUTER

Module 04: Test-Driven Development & Refactoring

Advanced Programming

Ichlasul Affan
Editor: Teaching Team (Lecturer and Teaching Assistant)
Advanced Programming
Semester Genap 2024
Fasilkom UI

Author: *Ichlasul Affan*
Email: ichlasul.affan12@cs.ui.ac.id

Table of Contents

Learning Objectives

1. Students should be able to understand the motivation of Test-Driven Development.
2. Students should be able to make test suites thoroughly and able to cover edge cases.
3. Students should be able to understand Test Doubles (mocking, stubbing)
4. Students should be able to identify bugs or faults using tests.
5. Students should be able to identify code design flaws.
6. Students should be able to fix design flaws by using refactoring steps.

References

1. Martin, R. C (2007). *Clean Code: A Handbook of Agile Software Craftsmanship*. Pearson.
2. Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.
3. Beck, K. (2000). *Test-Driven Development: By Example*. Pearson Education, Inc.
4. Percival, H. J. W. (2017). *Test-Driven Development with Python: Obey the Testing Goat: Using Django, Selenium, and JavaScript*. O'Reilly Media.
5. Myers, J. G.; Badgett, T.; Sandler, C. (2012). *The Art of Software Testing (Third Edition)*. John Wiley & Sons, Inc.
6. Amman, P. and Offutt, J. (2017). *Introduction to Software Testing (Second Edition)*. Cambridge University Press.
7. Refactoring Guru: https://refactoring.guru
8. https://martinfowler.com/articles/practical-test-pyramid.html

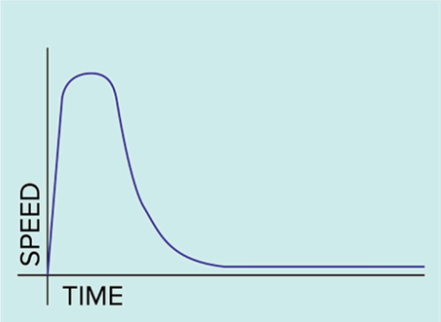
1. Test-Driven Development

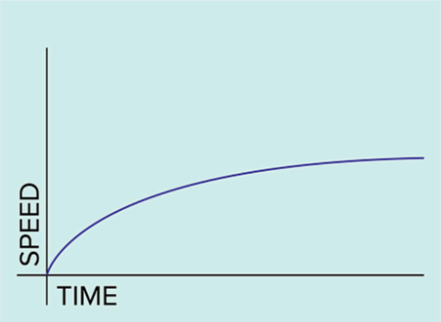
This submodule will explain about the motivation and workflow of Test-Driven Development. This submodule will also explain problems when doing Test-Driven Development and how to tackle them. By learning this submodule, students should be able to understand the motivation of Test-Driven Development.

Test-Driven Development

Motivation of Software Testing

Developing software is a **continuous work**, not just one time.
These graphs show performance speed over time.
Which one you prefer?





Let's contemplate on how you code, from the first time you know about programming, until today.

1. How fast do you code a simple problem (let's say, a to-do list web app)?
   _____
2. Look back at your "Platform-Based Programming" programming task (or group project) code. Try to read it. How many hours do you need to spend to understand how your code works? _____
3. Have you ever used a test before? Did the tests (if exist) help you understand your code?
   _____
4. If the lecturer asks you to add more features to your code, how long will it take for you to just "think" before starting coding? _____

Based on those questions, which graph describes your previous works? And which graph describes your dream? _____

Test-Driven Development (TDD)

In essence, we need to follow three simple steps repeatedly:

- **(Red)** Write a test for the next bit of functionality you want to add.
- **(Green)** Write the functional code until the test passes.
- **(Refactor)** Refactor both new and old code to make it well structured.

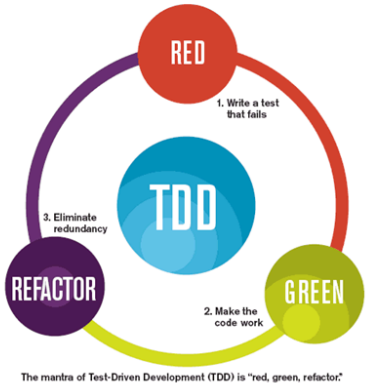


Image source:
https://www.nimblework.com/agile/test-driven-development-tdd/

In software development, there is a technique called Test-Driven Development (TDD) that aims to help developers build cleaner software, guided by software tests. TDD was developed by Kent Back in the late 1990s, as part of Extreme Programming (XP) methodology. XP methodology itself, exactly like its name: "extreme", combines 12 "extreme" practices for Agile software development, such as pair programming and TDD.

TDD itself is a continuous cycle that consists of three steps:

1. **RED**: Design and write a test for the next bit of functionality you want to add.
2. **GREEN**: Write the functional code until the test passes. For this step, the goal is only to pass tests.
3. **REFACTOR**: After all tests passed, refactor both new and old code to make it well structured, maintainable, and performant.

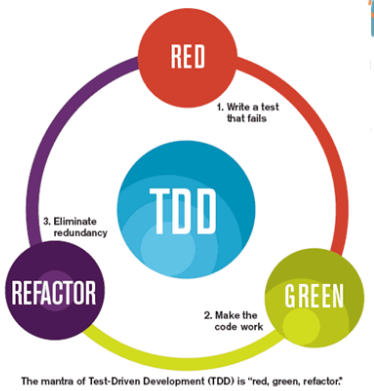
TDD has three laws that must be followed according to Martin (2007):

1. You may not write production code until you have written a failing unit test. **RED**
2. You may not write more of a unit test than is sufficient to fail, and not compiling is failing. Don't overthink at **RED**.
3. You may not write more production code than is sufficient to pass the currently failing test. Don't overthink at **GREEN**.

Test-Driven Development (TDD)

TDD seems scary at first, but we need to remember that:

By doing TDD, we <u>make tests first for **just one functionality at a time**</u>, then implement it, then we repeat the cycle for next functionality.
TDD **DOES NOT** mean making tests for the whole program first then implement it.



The mantra of Test-Driven Development (TDD) is "red, green, refactor."

Image source:
https://www.nimblework.com/agile/test-driven-development-tdd/

TDD seems scary at first. Some of you might ask, “How can we test something that did not even exist?”. Some of you that started to try TDD might also be stuck on thinking how the tests will be, and how many tests needed before continuing to the next step. If you ever encounter such questions or issues, remember that TDD **DOES NOT** require you to make tests for the whole program or module before implementing it. **Keep it small, write tests that you can think of, then implement, then repeat the cycle again**.

Some of you might also want to ask, “Do we need to refactor every time we successfully pass the GREEN step?” For that, refactoring depends on your needs and what you are currently thinking about. TDD cycle can be RED-GREEN-RED-GREEN-REFACTOR-repeat. Just remember that you must refactor your code as soon as you think the code starts to “smell”, or you feel like the structure becomes too bloated, or you have an idea to improve your code design. Sometimes, building tests over time might help you build the “clean code” instinct. :D

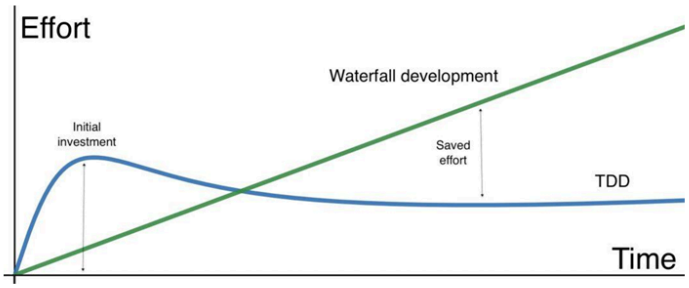
Before you move on to the next page, imagine yourself doing TDD, then explain to us what kind of advantages you will get if you start doing TDD?

The illustrations there explained enough about the contrast between implementing TDD vs disregarding tests. TDD feels super heavy at first, but it will really ease you in the long term

because most bugs are prevented by your own tests. Maintaining code without tests will be a lot more expensive than making tests in the first place.

Test-Driven Development (TDD)

- ✓ Developing software is a continues development.
- ✓ Regularly we add additional improvement to the software.
- ✓ The improvement should be done *fast forever*, every time we need it.



- Go Fast Forever
- Clean Code
- Refactoring
- Confidence
- Tests / TDD

TDD not only helps you during the development and maintenance process, but also helps you when you add more features. Tests that you build will help you do “regression testing” on existing features, making sure that old features still work after adding new features. Furthermore, automated testing will give us faster feedback whenever a fault happens. This is what we call “the improvement should be done **fast forever**, every time we need”. TDD, combined with clean code instinct, and good knowledge of refactoring techniques, will give you confidence on building your software with stable effort all the time.

Problems of TDD

How to do test-first on things we don't understand yet?
Sometimes, we are stuck on thinking how to test new and complex things, such as Authorization.

We can explore that "new thing" first, using a technique called **Spiking** and **De-spiking**.
Spiking = exploratory coding.

Of course, Test-Driven Development (TDD) is not a "silver bullet" solution for your programming workflow. Some things might hinder you from using TDD, such as testing new and complex things. How do we test things that we don't even understand yet?

TDD is not meant to be enforced all the time. Sometimes we need to branch out our work and do things without purely following TDD. Percival (2017) called this technique "Spiking" and "De-Spiking". "Spiking" here means exploratory coding. When you explore things, of course, you want to do all you want to gather every information possible. That's why we need to branch out our work before "spiking".

Spiking

1. Make a new branch specifically for “spiking”.
2. Do exploratory coding there. We don’t need to follow TDD there.
3. Gather as much information on how things work, until we can design how to test them.
4. Don’t forget to commit those “spike” codes in “spiking” branch, for documentation.
5. Don’t worry too much about ugly code, this is just exploratory coding.
6. Design tests based on the “spike” code.

Spiking example (Python, Django):
https://www.obeythetestinggoat.com/book/chapter_spiking_custom_auth.html#_exploratory_coding_aka_spiking

Before we spike, make a new branch in Git specifically for spiking. Don’t forget to always commit those spike codes in the “spiking” branch, and just don’t worry too much about anything. The end goal is you can design tests based on those knowledge.

De-spiking

1. Return to our usual work branch. Remove all “spike” code.
2. Implement tests that we have designed previously. (**RED**)
3. Implement the code to fulfil the tests. (**GREEN**)
4. Clean up, refactor, and optimize the code after passing tests. (**REFACTOR**)
5. Continue working on TDD flow as usual.

De-spiking example (Python, Django):
https://www.obeythetestinggoat.com/book/chapter_spiking_custom_auth.html#_de_spiking

To de-spike, just return to your main branch, remove “spiked” artefacts, and follow TDD workflow as usual. Use those design and implementation knowledge that you have obtained.

Problems of TDD

How to do test-first on a feature that depends on another service that does not exist yet?

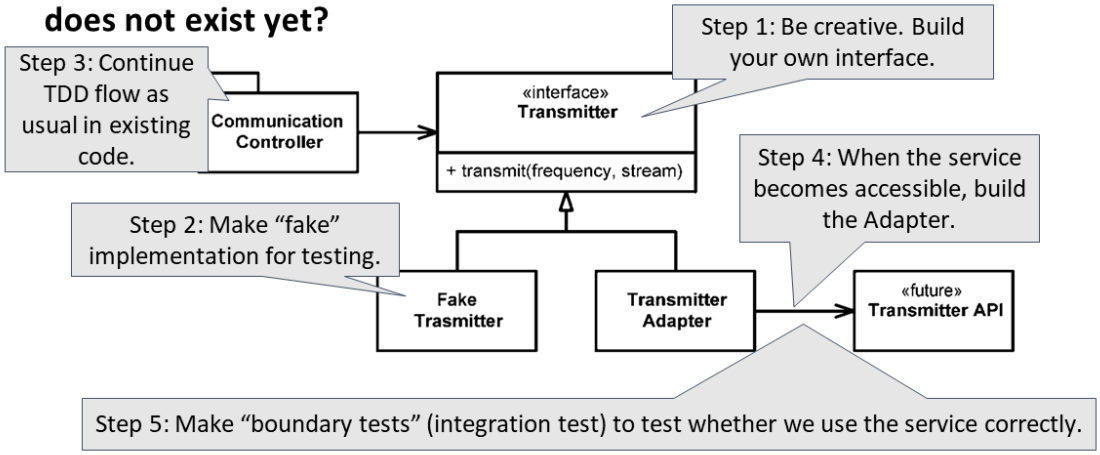
In a team-based software development, especially when adopting Microservices architecture, sometimes our service needs to depend to other services.

If the dependency is not implemented yet and there is no design available from other teams, it's okay to **define our own version of that dependency**.

You might also encounter another problem: "How to test a feature that depends on another service that does not exist yet?" This is often happening when you are doing teamwork with big teams, where each smaller team builds their own assigned service. To avoid bottlenecks and boredom on waiting for other teams to successfully build that service, why not define your own version? Be creative! You can make tests based on your imagination. Remember, TDD does not urge you to have good code at first, you can always refactor it later when you are ready.

Define A New and Unknown Dependency

How to do test-first on a feature that depends on another service that does not exist yet?



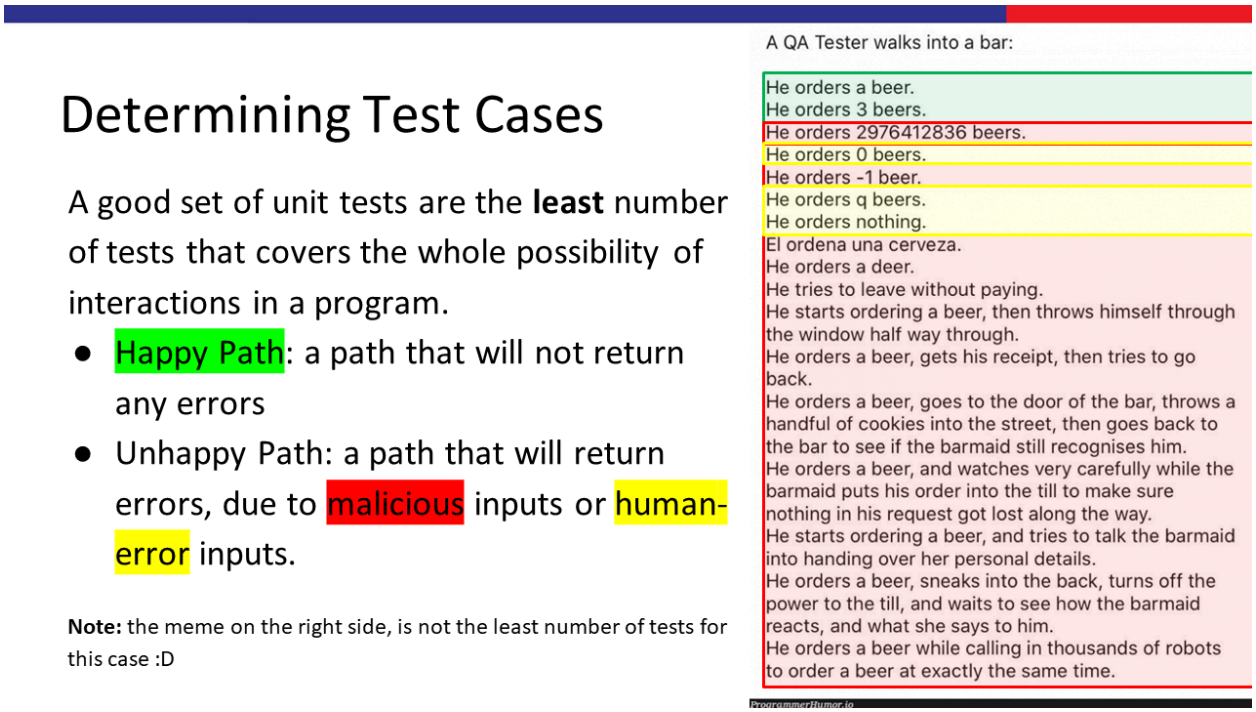
For this case, we can use Adapter Pattern (https://refactoring.guru/design-patterns/adapter) to make our own imagined interface and implementation. Focus on what we need from the unknown service. After we know the interface, we can build the "fake" implementation as the test double. Then we can continue TDD flow as usual in existing code.

When the service becomes accessible, build the Adapter pattern to adapt the new service to our previously made interface. If somehow there are things that we missed when we made the interface, we can refactor the interface and maybe change some tests. We can also make integration tests to check whether we use the new service correctly.

2. Deeper into Unit Tests

This submodule will explain deeper techniques on building unit tests. This includes how to determine test cases using test modelling techniques, and how to make Test Doubles. By learning this submodule, students should be able to make tests thoroughly and be able to cover edge cases. Students should also be able to understand Test Doubles (mocking, stubbing).

Determining Test Cases



The meme above is about a Quality Assurance (QA) tester that came to a bar (yeah, QA is truly a role in itself). If you think that testing should only be done with QAs, the answer is no. You as a developer must also know how to test your own code. That includes designing test cases for your own code.

There are two types of paths that will be taken by a user, either **Happy path**, or **Unhappy path**. The **happy path** is just the user doing what he/she should do, your program runs without error, and the user gets what he/she wants. Meanwhile, the **unhappy path** is either the user mistakenly input things in your program, or he/she just being malicious. Both of those paths must be covered by your tests. You need to make sure your program didn't exit or raise unexpected errors if an **unhappy path** ever happened.

Things to Remember When Designing Tests

From https://martinfowler.com/articles/practical-test-pyramid.html#WhatToTest:

- Unit tests should at least **test the public interface of the class**.
- Unit tests should ensure **non-trivial code paths are tested**, while making sure test codes **does not closely tied to the implementation**.
- Unit tests that are heavily tied to the implementation will hinder refactoring process.
- Do not reflect internal code structure within unit tests. **Test the observable behaviour only**: "if the input is x and y, will it return z?"
- **Never test private methods**. If testing private methods become necessary, think again about your code design.

Although we need to cover both paths, we need to make sure that our tests are not too exhaustive. When designing tests, make sure that your tests are not closely tied to the implementation. You can do this by focusing yourself to test the observable behaviour only, for example: "If I transfer money to account number 123456 with an amount of 10000 USD, will it return an invoice?". **Avoid reflecting internal code structure in your tests**. If your tests are too closely tied to the implementation, it will hinder your progress when refactoring. You would get busier fixing your tests than doing the actual refactoring process.

After following some TDD cycles, if you ever found yourself needing to test private methods, then check again your code design. Oftentimes, it's not about scoping mistakes, but more about design mistakes. **Private methods are not meant to be tested**. If your tests do not cover conditionals inside those private methods, it means another part of code that uses your private methods does not use those conditionals. The option is either to fix that private method, or fix other methods/classes that depend on the private method.

Determining Test Cases by Modelling

Sometimes, when designing tests, we miss some edge cases.
Sometimes, when designing tests, we also put too much "invalid" cases just to test our code's error handling.

There are many test modelling methods that can help you determine least amount of test cases that covers the whole function. One of them is **Boundary Value Analysis**.

Interested to learn more about test modelling methods? You can take **Software Quality Assurance** course after you pass Advanced Programming course or read these books:

- **Introduction of Software Testing (2nd edition)** by Paul Amman & Jeff Offutt.
- **The Art of Software Testing (3rd edition)** by Glenford J. Myers, Tom Badgett, and Corey Sandler.

Sometimes, when designing tests, we miss some edge cases. Sometimes, when designing tests, we also put too many "unhappy path" cases to test our code's error handling. Sometimes we make tests just for the sake of making tests, not really considering whether it is effective or is just too much.

Myers et al. (2012) and Amman and Offutt (2017) in their books, defined several modelling methods to determine test cases. One of the modelling methods is called **Boundary Value Analysis**. If you want to learn more about test design methods, you can read Myers' book (The Art of Software Testing) chapter 4, and Amman's book (Introduction of Software Testing) part 2.

Boundary Value Analysis (BVA)

BVA tests the boundary values of "valid" and "invalid" partitions of a function's input combinations. The edge values of a partition are more error-prone than values in the middle of a partition.

For each partition, test:

- Lowest value of the partition
- Lowest value of the partition + 1
- Middle partition
- Highest value of the partition – 1
- Highest value of the partition

Boundary Value Analysis (BVA) tests the boundary values of every partition of a function's input combinations. BVA only determines test cases based on input possibilities, without considering functionalities or output. BVA picks 5 values for each partition: lowest value, lowest value + 1, middle, highest value - 1, and highest value. This is due to edge values (near the lowest and near the highest) are more error-prone than any value in the middle.

Take example of the function that validates birth year: Birth year must be between 1900-2000. This can be achieved by using conditional: `1900 <= birth_year <= 2000`. But maybe we unconsciously misinterpret that requirement as "exclusive", such as `1900 < birth_year < 2000`. If that happens, the lowest value (1900) and the highest value (2000) will return "invalid". Meanwhile anywhere in the middle (for example: 1930) is just fine in both conditionals.

How do we partition an input function? We will use the "**equivalence partition**" method. We partition the input parameter based on whether the input is valid or invalid. In the birth year validation example, the "valid" partition will be between 1900 and 2000 (inclusive). There are also 2 "invalid" partitions: below 1900, and above 2000. Based on the example, how many tests will BVA suggest you? Write those tests here:

Boundary Value Analysis (BVA)

To test functions with multiple parameters, we can use Single Fault Assumption.
Single Fault Assumption = Holding all but one variable to the middle value and allowing the remaining variable to take the extreme value.

For example, the boundaries are: 1 <= month <= 12, 1900 <= year <= 2000
The tests will be:

month = 1, year = 1950	month = 6, year = 1900
month = 2, year = 1950	month = 6, year = 1901
month = 11, year = 1950	month = 6, year = 1999
month = 12, year = 1950	month = 6, year = 2000
month = 6, year = 1950	**Total tests: 4n+1 = 4*2 + 1 = 9 tests**

Now, how about "invalid" partition? How many tests we need?

How about functions with more than one parameter? For example, we want to expand our previous function to also check the month. So, there are now 2 parameters. In this case, we will use **Single Fault Assumption**. Single Fault Assumption will hold all but one variable to the middle value and allow only one variable to take the extreme value. This is based on the assumption that faults will only happen when validating a single parameter, hence the "Single Fault" name.

For example, our valid boundaries are: `1 <= month <= 12` and `1900 <= year <= 2000`. We can make 4n + 1 = (4 * 2) + 1 = 9 tests out of those boundaries. First, we make 4 tests that keep the `year = 1950`, and then `month` in one of {1, 2, 11, 12}. Second, we make 4 tests that keep `month = 6`, and then `year` in one of {1900, 1901, 1999, 2000}. Third, we make a test that both parameters have middle values (`month = 6`, `year = 1950`).

Based on that explanation, how many tests will BVA suggest, to cover both valid boundaries and invalid boundaries? Write those tests here:

Boundary Value Analysis (BVA)

Disadvantage of BVA: The analysis does not consider functionality of the test subject. The analysis only consider the input possibilities.

For example: Function to check type of a triangle based on three sides (a, b, c).
There are 4 possibilities of output: Not a triangle, Scalene (all sides are different), Isosceles (2 sides are the same), Equilateral (all sides are the same).

By only relying to Boundary Value Analysis, we don't have test case for Scalene possibility. This is because Single Fault Assumption only **varying the value of one parameter** and holds all other parameters to the same "middle" value.
To test Scalene case, all parameters must be different (for example: a=3, b=4, c=5).

Of course, due to its rudimentary nature of BVA, it has some disadvantages. You might miss one of the functionality requirements if you are solely following BVA suggestions. For example, a function that checks the type of a triangle based on the length of three sides: side a, side b, and side c. So, there are 3 parameters.

Due to Single Fault Assumption, the tests will only vary the value of one parameter at the time. The other two parameters will be the same, in the "middle" value. Meanwhile, there are 4 possibilities of "triangle type check" function output: Not a triangle, Scalene (all sides are different), isosceles (2 sides are the same), Equilateral (all sides are the same). To test Scalene validation, you need to input different numbers on each side (for example: a=3, b=4, c-5). There are no tests suggested by BVA that will cover such cases.

Well, then what should we do? **Don't rely on a single modelling method**. Be creative! If you know what the output possibilities are, you can do other modelling methods, or just define the tests yourself. Defining the tests yourself by adding missed cases, is called "Error Guessing" in Myers' book. If you want to use other kinds of modelling methods, you can read Amman's book at chapter 6 about Input Space Partitioning (ISP). ISP can partition input possibilities not only by interface (like we do in BVA), but also by functionality (the possibility of outcome, or classification by non-numeric means).

Coverage Metrics

Test Coverage Metrics

In IntelliJ IDEA, there are 4 kind of coverage metrics:

Element	Class, %	Method, %	Line, %	Branch, %
com.example.sibayar	91% (21/23)	100% (61/61)	100% (328/328)	81% (39/48)

- **Class Coverage**: Count of classes that have been executed at least once.
- **Method Coverage**: Count of methods that have been executed at least once.
- **Line Coverage**: Count of code lines that have been executed at least once. This is the most used coverage metrics.
- **Branch Coverage**: Count of conditional branches that have been executed at least once. This coverage counts every conditional branch, even when it is defined in a single line (for example: `value.isEmpty() ? 0 : value.get()`)

In week 2 we learned about CI/CD and DevOps, one of them is about automating code maintainability metrics (using SonarQube). One of the metrics is **code coverage**. In IntelliJ IDEA, there are 4 kinds of coverage metrics:

- Class Coverage: Count of classes that have been executed at least once.
- Method Coverage: Count of methods that have been executed at least once.
- Line Coverage: Count of lines of codes that have been executed at least once.
- Branch Coverage: Count of conditional branches that have been executed at least once. Remember that conditional branches can also be defined in a single line, for example: `value.isEmpty() ? 0 : value.get()`. This is what makes Branch Coverage more important than Line Coverage.

The important question is, **does 100% test coverage imply that a software has good tests?** Also, does 100% test coverage ensure that a software has no bugs? Explain your thoughts here:

You can highlight the answer but **be honest** and answer by yourself first.

Test Doubles

- **Dummy:** An object that is created only to fill parameters or as data container. Dummies only created inside a test suite.
- **Fake:** A concrete implementation that is only used for testing purposes, not for operational, such as `InMemoryDatabase`.
- **Stubs:** A fake implementation that is preprogrammed to give a predefined result when a certain kind of input is given. Stubs do not involve any real implementation.
- **Mocks:** A fake implementation (stub) with additional side effect and interaction monitoring. Mocks can record executions and interactions of the class or method.
- **Spies:** A real implementation with additional side effect and interaction monitoring. Unlike mocks, spies contain real implementation of the class or method.

Test Doubles are important techniques in software testing. Test Doubles allow you to isolate your unit tests by making fake implementations or to deeply test a class. There are five kinds of Test Doubles:

- **Dummy** is an object to store data or to fill parameters. Dummy is often defined in the test folder as a utility.
- **Fake** is a concrete implementation that is purposefully built for testing purposes, not for operational use. Fake is also often defined as a utility.
- **Stub** is a fake implementation that is preprogrammed to mimic the structure of a dependency class. Stub has no implementation inside whatsoever. Stub is created using a mock library. The mock library has functions to define input-output pairs for a stub. A stub works simply by returning the defined output when executed with a certain input.
- **Mock** is a stub that can also record interactions to the stub itself. Mocks can monitor side effects and execution count of a member method. Mock is also created using a mock library. The mock library has functions to verify whether a certain interaction ever happened at that time.
- **Spy** is a real implementation (unlike stub and mock), that can also monitor interactions just like mocks. Spy is also created using a mock library. The verification process is exactly the same as mock, but when a spy class/method is executed, it will execute normally just like the real class/method.

Here are the usage of Stubs, Mocks, and Spies respectively:

Stubs

Use Stubs to:

- provide a predetermined response from a collaborator
- take a predetermined action from a collaborator, like throwing an exception

Mocks

Use Mocks to:

- verify the contract between the code under test and a collaborator
- verify the the collaborator's method is called the correct number of times
- verify the collaborator's method is called with the correct parameters

Spies

Use Spies to:

- test collaborator’s legacy code that are too difficult or impossible to test with a Mock
- verify the collaborator's method is called the correct number of times
- verify the collaborator's method is called with the correct parameters
- provide a predetermined response from a collaborator
- take a predetermined action from a collaborator, like throwing an exception

Spies are particularly useful when we want to also understand the side effect of a legacy code or a dependency when the class/method is being executed.

Stubbing and Mocking using Mockito

- In many test frameworks, including Mockito, these concepts are combined to a single term: **mock**.
- The differences between stubbing and mocking are not really obvious programmatically.
- We still need to understand these concepts separately.

In Java, there is a mocking library called **Mockito** (https://site.mockito.org/). In Mockito, the definition of stub and mock is defined in a single term: **mock**. A mock in Mockito are stubs that can also do things that mocks can do. In Mockito, spy is defined separately as it involves real implementation.

Stubbing and Mocking in Spring Boot

```
@ExtendWith(MockitoExtension.class)
class PaymentServiceImplTest {
    @InjectMocks
    private PaymentServiceImpl service;
    @Mock
    private PaymentRepository paymentRepository;
    @Mock
    private MoneyAPI moneyAPI;

    ...
}
```

Mockito supports stub, mock, and spy for Dependency Injection technique in Spring Boot

Use:
`@InjectMocks` to mark a "bean" that becomes a test subject. Mockito will inject stubs/spies to it.
`@Mock` to mark a "bean" that will be stubbed.

Use `@ExtendWith(MockitoExtension.class)` above every test suite that uses mock, stub, or spy. Stubs are especially useful in Spring Boot when testing services, so that unit tests do not use production-like databases. We can do that by mocking a `Repository` (for example, in the slide `@Mock` annotation is added to `PaymentRepository`).

To make mocks useful, we need to inject them to a "test subject". Mark that test subject with `@InjectMocks`. For this example, we want to test `PaymentServiceImpl`, an implementation of `PaymentService`. You might wonder, why don't we use `PaymentService`? That's because in this test, we want to test the actual implementation, which is the `PaymentServiceImpl`. Maybe if there is another service that implements `PaymentService`, we also test that separately in a different test suite.

We also need to make sure that our unit tests are isolated from interactions with other libraries or external services, such as `MoneyAPI`. We can also mock the `MoneyAPI` here.

Spying in Spring Boot

```
@ExtendWith(MockitoExtension.class)
class PaymentServiceImplTest {
    @InjectMocks
    private PaymentServiceImpl service;
    @Spy
    private PaymentRepository paymentRepository;
    @Mock
    private MoneyAPI moneyAPI;

    ...
}
```

Use:
@Spy to mark a “bean” that will be spied.

Remember the difference of Spy vs Stub. Spy is a real object, but we can assert its interactions. Meanwhile, Stub is a fake (“double”) object.

To make a Spy, mark the dependency with `@Spy` annotation. Remember that Spy is a real implementation that is added/decorated by monitoring capabilities. It will run the logic of the original class. In this example, we want methods in the `PaymentServiceImpl` to also have side effects, so we allow `PaymentRepository` to be a real object. So, we add `@Spy` annotation to the `PaymentRepository` variable definition. Note that this is only possible when the `PaymentRepository` is a simple class without any Spring dependencies.

There is a catch (that is not added to the slide). You might wonder about “How to use real implementation [without any modification] together with mocks or spies, using Spring’s own `Autowired` dependency injection system?” If that’s the case, you can refer to this tutorial: Injecting Mockito Mocks in to Spring Beans | Baeldung. The tutorial guides you to use dependency injection profiles. Unlike `InjectMocks`, this is more tedious but more flexible. We can define our own mocks and spies there and let Spring’s `Autowired` inject them. Spring’s `Autowired` will refer to default Bean if there is no definition override in the test profile. Mockito’s `InjectMocks` don’t have any reference to the real implementation, so we must either Mock or Spy all dependencies.

Defining Stubs in Spring Boot

```
@Test
void testPaymentCallbackWithCancelledStatus() {
    doReturn(Optional.of(payment))
        .when(paymentRepository)
        .findByPaymentLinkId(any(Integer.class)));

    paymentCallbackRequest.setStatus("CANCELLED");

    Payment payment = service.paymentCallback(
        paymentCallbackRequest);
    assertEquals("PAYMENT_FAILED", payment.getStatus());
}
```

Use:
`doReturn(value)`
`.when(stubClass)`
`.stubMethod()`
to define execution signature and its return value.

`any(X.class)` to define that the parameter that will trigger the stub is any object of type `X`.

To make stubs usable, you need to add your own definition of input and output pair. To define that, use `doReturn(value).when(stubClass).stubMethod(parameters)` syntax. The syntax means that an execution of `stubClass.stubMethod(parameters)` will return `value`. Remember that the "value" must have the exact same type as the return value of `stubMethod`.

There is also a function called `any(AClass.class)`. This function means that you can input any object of `AClass` into the parameter, and it will return the defined value. In this example, if we execute `paymentRepository.findByPaymentLinkId(1)`, it will return `Optional.of(payment)`. The same `Optional` of `Payment` object will be returned even though I execute the method with any other `Integers`.

Defining Stubs in Spring Boot

```
@Test
void testPaymentCallbackWithInvalidPaymentLinkId() {
    doThrow(EntityNotFoundException.class)
        .when(paymentRepository)
        .findByPaymentLinkId(any(Integer.class)));

    assertThrows(PaymentError.class,
        () -> service.paymentCallback(
            paymentCallbackRequest)
    );
}
```

Use:
`doThrow(exception)`
`.when(mockClass)`
`.mockMethod()`
to define execution signature and exception that will be throwed.

This is to simulate exception handling in a test subject.

The syntax is a little bit different when we want to simulate throwing errors after executing the method with a certain input. Use `doThrow(Exc.class).when(stubClass).stubMethod(params)` syntax. For example, if we use any integer parameter to execute `paymentRepository.findByPaymentLinkId()`, it will throw an `EntityNotFoundException`.

Defining Mock/Spy Testing in Spring Boot

```
@Test
void testPayToUserSuccess() {
    doReturn(Optional.of(user))
        .when(userRepository)
        .findById(any(Integer.class)));
    doReturn(paymentLinkResponse)
        .when(moneyAPI)
        .getPaymentLink(any(Integer.class)));

    Payment result = service.payToUser(1,
paymentToUserRequest);
    verify(paymentRepository, times(1))
        .save(any(Payment.class));
    assertEquals(paymentLinkResponse.getLinkId(),
result.getPaymentLinkId());
}
```

Use:
`verify(mockClass, times(amount)).mockMethod()`
to verify whether the `mockMethod` is executed for a certain times.

In this case, we check that `paymentRepository.save` is executed once.

As the stub is also the same as mock in Mockito, we can verify (check) the interactions of the stubbed method. For that, use `verify(mockClass, times(amount)).mockMethod(params)` syntax. For example, we want to verify whether the `paymentRepository.save()` is executed using a `Payment` object as a parameter, at least once. The syntax will be `verify(paymentRepository, times(1)).save(any(Payment.class))`.

We can also check the range of execution amount (for example: executed between 3 to 8 times), by using `atLeast()` and `atMost()` functions. We need two lines of verification to do so. We can also define things such as `never()` or `once()`. For more info about mock verification syntaxes, visit this tutorial: [Mockito Verify Cookbook | Baeldung](#).

Notes on Test Doubles

- **Do not inject mocks** (or we call it "stub" in theory) **to spy objects**. Spy objects are real implementation. A clean test does not need such scenario to happen.
- **Only create stubs that the test need to use**. Mockito has a strict policy that raises `UnnecessaryStubbingException` when there is unused mock in a test case. This will also help you make a clean and fast unit tests.
- **Do not over-mock**. There are A LOT of Mockito features that you might want to explore. However, remember that overengineering mocks will make your tests heavily tied to the implementation.
- **Tests that heavily tied to the implementation** will make refactoring a lot more difficult. Use stubs/mocks/spies only if necessary, and keep it abstract enough so refactoring won't involve a lot of change in the test code.

These are the tips on best practice when making Test Doubles. We need to make sure by using mocks and spies does not make our test code too closely tied to the implementation. If you ever want to verify the mocks, just verify the execution times. Verifying deeper from that often will get your test code tied closer to the implementation, and it will make refactoring difficult as you also need to change the verification rules.

About the `UnnecessaryStubbingException`, this exception will be automatically raised when there is an **unused stubbing rule**. It is not just about unused stub class, but also about the unused rule (`doThrow()` or `doReturn()`). Make sure you created a stub rule that will certainly be executed.

3. Principles and Best Practices of Testing

This submodule will explain principles and best practices of testing. This submodule will give some tips and tricks on fulfilling such principles and best practices. By learning this submodule, students should be able to make cleaner tests using their previous knowledge given by previous submodules.

F.I.R.S.T. Principle

F.I.R.S.T. Principle on Testing

- **Fast:** The tests run as fast as possible so it can be run without interrupting your workflow.
- **Isolated/Independent:** A test case must not interfere, change the state of functions, or dependent to other test cases.
- **Repeatable:** Tests must be able to run repeatedly, with consistent result.
- **Self-Validating:** Tests must be self-validating (have strict and correct assertions to pass/fail).
- **Thorough/Timely:** Tests must cover all happy paths and unhappy paths. Cover all possibility of results and errors. Tests must be designed before coding.
  Note: You should make as few tests as possible (this correlates with **Fast** aspect), while trying to cover all the possibilities and conditions.

There are two slightly different definitions of F.I.R.S.T. Principle on Testing. Martin (2007) proposes Fast, Independent, Repeatable, Self-Validating, and Timely. Meanwhile, other sources (such as https://dzone.com/articles/first-principles-solid-rules-for-tests) defined the T as Thorough instead of Timely, due to “Thorough” being more suitable for broader types of testing.

F.I.R.S.T. Principle: Fast

Fast: The tests run as fast as possible so it can be run without interrupting your workflow.

- CI/CD pipeline management: Separate unit tests and functional tests.
- Avoid waiting for another subsystem/function when using unit tests. Use stubs to predetermine result of required functions or external library call.

There are many ways to make tests faster. Using test doubles can help you avoid bottlenecks of accessing other dependencies or subsystems, while isolating your unit tests. But remember that only use test doubles on unit tests, not functional tests. That's also why we need to separate unit tests, functional tests, and integration tests to different tasks so that they can be parallelized.

For functional tests, fine tune the delays between moving to a new page or submitting a form and checking the result page. You might need the delays, however, do not make it too long. CI Runners such as GitLab CI or GitHub Actions are fast enough, but you might also need to adjust for your own computer if you have a slower one.

F.I.R.S.T. Principle: Isolated

Isolated/Independent: A test case must not interfere, change the state of functions, or dependent to other test cases.

- Define `setUp` to set up dummy or mock objects before every test case, and `tearDown` to clean up objects and databases after testing.
- `setUp` and `tearDown` are meant to avoid duplications when isolating every test case in a test suite.
- When a function requires a function call to another function or class, use Test Double techniques.

Isolation is very important in unit tests. In functional tests and integration tests however, the isolation is kind of different. Isolation in integration tests means you need to build a separate testing environment for all the dependencies (including external services), so the testing process won't disturb the real environment. Isolation in functional tests can also be done in the same way, or we can stub the external services. In unit tests, we need to stub dependencies, not only the external services, as we want to focus on the logic in the "test subject" class.

In software tests, we can achieve isolation by setting up and tearing down environments or dependencies each time a test is executed. To avoid duplications in a test suite, we can define our own `setUp` and `tearDown` methods.

This is the example of `setUp` in a test suite. Annotate that with `@BeforeEach` so that JUnit will execute the `setUp` method before each test case.

setUp in a Test Suite

setUp can be used to avoid duplication of set up process for every test case in a test suite. For example, we want to set up `User` and `Payment` object as arguments for each test case:

```
@BeforeEach
void setUp() {
    user = User.builder().id(1).name("Bambang").accNumber("123456")
               .build();
    payment = Payment.builder().id(1).sender(user).amount(10000)
                    .build();
}
```

This is the example of `tearDown` in a test suite. Annotate that with `@AfterEach` so that JUnit will execute the `tearDown` method after each test case.

tearDown in a Test Suite

tearDown can be used to avoid duplication of tear down (clean up) process for every test case in a test suite. For example, we want to clear test folder to store invoices using `org.apache.commons.io.FileUtils`:

```
@AfterEach
void tearDown() {
    FileUtils.cleanDirectory("tmp/test_temp/invoices");
}
```

Not all test suites need “set up” and “tear down”. However, if it has dependencies or external side effects (for example: access to database, creating files, etc.), we must define “set up” and “tear down” to isolate our tests.

F.I.R.S.T. Principle: Repeatable

Repeatable: Tests must be able to run repeatedly, with consistent result.

- This principle correlated with Isolated/Independent principle.
- If the function involves calling another function that returns different result at different time, use Test Double techniques.

The “Repeatable” principle is also the reason why Test Double techniques are important for you to understand. Remember the reason why we want to isolate tests: we want to keep results consistent every time we run those tests. We can’t control how external services behave, that’s why we shield our tests from those uncertainties.

F.I.R.S.T. Principle: Self-Validating

Self-Validating: Tests must be self-validating (have strict and correct assertions to pass/fail).

- Make human-readable assertions. Add messages to every assertion.
- Only use assertions when making tests. Do not use “print” or manual exit process call.
- Make sure the assertions have as minimum scope as possible. If the test result needs range of result “tolerance” (for example, 2.0 to 2.99), design the allowed result range as tight as possible.
- Strive for “one test one assertion”. **Note:** make sure to consider **Fast** principle on this tip, do not make your tests slow because you are executing long-running function twice.

For faster feedback, automated tests must be self-validating. To achieve that, add assertions and make it human friendly by adding messages to each assertion. Do not use print, use only assertions. Make assertions as strict as possible. If you need some tolerance in the result (for example you are testing a Machine Learning model), keep the range as tight as possible.

We also need to make sure that a test case only tests one thing (or one scope). Keep it simple by making only one assertion per test whenever possible. But this is where “the art of balancing” is important, we do not want to make a lot of tests that run a single thing repeatedly.

F.I.R.S.T. Principle: Thorough/Timely

Thorough/Timely: Tests must cover all happy paths and unhappy paths. Cover all possibility of results and errors. Tests must be designed before coding.

- Design your tests. Understand the input-output possibilities of your functions.
- Cover all happy paths and unhappy paths, with the least number of tests.
- You can refer to Line Coverage and Branch Coverage metrics to help you know your tests are thorough enough, but do not refer to them as the only reference.
  Note: 100% Line/Branch Coverage **does not always** mean your tests are thorough enough. You might get 100% Line/Branch Coverage by making "dumb tests" or a code that is not covering the whole possibility (for example: no error handling).

Design your tests early, that's what TDD urges you to do. Understand input and output possibilities of your functions, then define happy and unhappy paths. Refer to coverage metrics such as Line Coverage and Branch Coverage to check whether you cover your program in tests or not.

But remember that 100% line/branch coverage **does not always mean** your tests are good and thorough enough (psst... this is the answer to the question in the "Coverage Metrics" subchapter). This is because coverage is measured by execution; coverage is not measured by assertions. So, we might just execute things and print, and make the line/branch coverage 100%, but that's not a good idea. **Always use assertions in every test case.** Be honest!

Evaluate Your Testing Objectives

There are no universal rule about what is the correct balance between unit tests, functional tests, and integration tests.
We must evaluate our tests based on these objectives:

- **Correctness**
- Clean and **maintainable code**
- **Productive workflow**

There is no universal rule about what the correct balance between unit tests, functional tests, and integration tests is. Again, it's the "art of balancing" (remember, programming is an art in itself). Percival (2017) proposes three main objectives that we need to constantly evaluate on our tests: Correctness, maintainability, and productive workflow. We must ask ourselves questions, whether our tests help us achieve those objectives or not. The answers may be unquantifiable, but at least we know where the problem is and start fine tuning as soon as possible.

Evaluate Your Tests: Correctness

- Do I have **enough functional tests** to reassure myself that my application really **works, from the point of view of the user**?
- Am I testing all the **edge cases thoroughly**?
- Do I have tests that check whether **all my components fit together** properly? Could some integrated tests do this, or are functional tests enough?

There are 3 self-reflecting questions about whether our tests help us go towards correctness. We must position ourselves from the user's point of view to answer these questions. Review again what users told you about your program, whether there are bugs that suddenly appear or not, whether the program is usable or not. Check again if you need more functional tests to help you get feedback faster. Check again if you need to cover more edge cases that you haven't imagined before. Check again if all the components fit together, and whether integration tests are needed or not. That way, we can at least fine tune our need for functional and integration tests.

Evaluate Your Tests: Maintainability

- Are my tests giving me the confidence to refactor my code, fearlessly and frequently?
- Are my tests helping me to drive out a good design? If I have a lot of integration tests but less unit tests, do I need to make more unit tests to get better feedback on my code design?

There are 2 self-reflecting questions about whether our tests help us go towards better maintainability. Check again whether your test code helps you refactor, or somehow you become too worried about breaking tests if you refactor. If you worry about breaking tests, then something is wrong about your test design. Check again whether you need to make more unit tests to check on the design for each class or function, or is it enough, or somehow you need more integration tests.

Evaluate Your Tests: Productive Workflow

- Are my feedback cycles as fast as I would like them? When do I get warned about bugs, and is there any practical way to make that happen sooner?
- Is there some way that I could write faster integration tests that would get me feedback quicker?
- Can I run a subset of the full test suite when I need to?
- Am I spending too much time waiting for tests to run, and thus less time in a productive flow state?

There are 4 self-reflecting questions about whether our tests help us go towards a more productive workflow. Check again on whether you really read tests or not; do tests help you find bugs or not. Check again if there is some way you can make integration tests faster. Look at your IDE and try to run only some tests out of a test suite, is it possible? If not, you need to check again the isolation and repeatability of your test code. Check again that if you run all the tests, it is still fast enough to make you not drowning in boredom :D

So, according to those self-reflective questions proposed by Percival (2017), what do you think of your own test-making approach on previous projects? Do making tests help you? If not, explain step by step on how you improve your test-making skill in future projects.

Self-Reflection A: Your approach on software testing

Unit Testing on Rust Programming Language

We will use Rust starting from Week 7. Here are references that can help you understand how to unit test and make test doubles on Rust:

- Unit testing: https://doc.rust-lang.org/beta/book/ch11-00-testing.html
- Test doubles: https://blog.logrocket.com/mocking-rust-mockall-alternatives/

This week, we learned about concepts of software testing. This can apply to any kind of programming language, but for other programming languages, you need to learn the libraries and syntaxes by yourself. *Ganbatte!* :D

For Rust though, as we will use Rust from Week 7, here are the references we can give you:

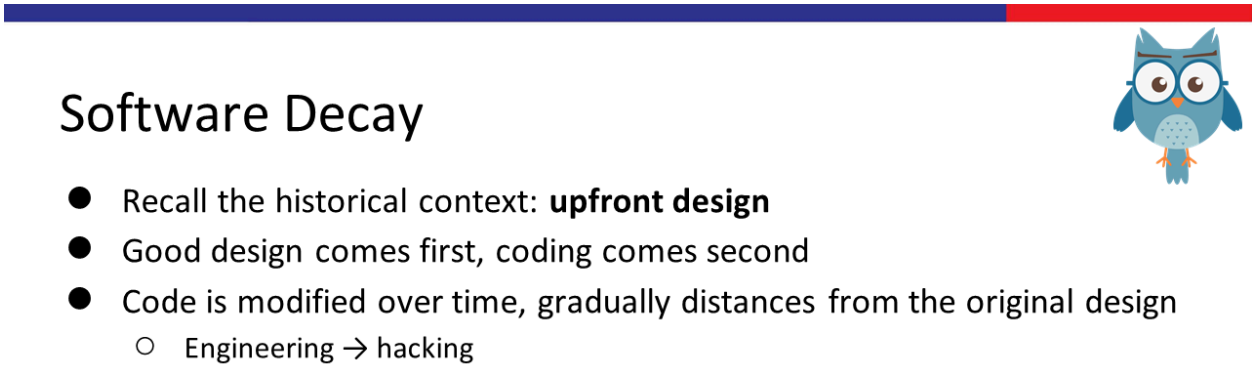
1. Unit testing: https://doc.rust-lang.org/beta/book/ch11-00-testing.html
2. Test doubles: https://blog.logrocket.com/mocking-rust-mockall-alternatives/

Hey, you might also want to tell us or other classmates on other links or reading sources that you might find helpful. Here we will give you a space to take note of those links. Feel free to use:

4. Introduction to Refactoring

This submodule will explain what to refactor and when to refactor. By learning this module, students should be able to understand the motivation of refactoring.

Motivation of Refactoring



In software development lifecycles (SDLC), “design” step is in front of the software development process. After we gather requirements from users, we analyse those requirements and then design the software (architectures, class design, etc.) first. The design will be used as our benchmark when coding the software. This is why it is called **upfront design**. Good design comes first, then we code based on it.

Unfortunately, as the software grows larger and more user requirements are incorporated, we will modify the code to suit those requirements. Oftentimes, when adjusting the code, we do not refer to the design that we did in the beginning of the project. The code will gradually distance from the original design over time. Sometimes, we need to fix the code back to the principle of good design.

Historical Context

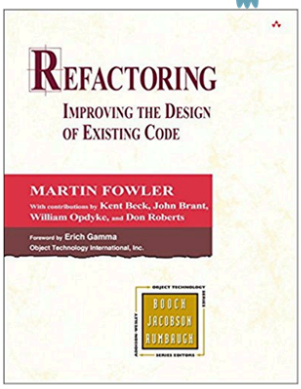
- OO paradigm was getting more popular in the industry (not just an academia language)
- Rise of **agility** in software development since early 90s
  - Traditional software development: Waterfall (upfront design) → design first, code later

In the historical context, as the Object-Oriented paradigm became more popular in the industry, it signifies our need to do **upfront design**. The rise of **agility** in software development since the 90s, makes the upfront design approach (we call it Waterfall method) less feasible. To meet the agility on understanding and implementing requirements, we can't design it upfront. In some cases (for example: startups), we even cannot understand the requirements upfront. We can only start with a simple idea that must be adjusted or improved over time according to market demand.

Definition of “Refactoring”

Refactoring

- *“The process of changing a software system in such a way that **it does not alter the external behaviour of the code yet improves the internal structure**.”*
- **Refactoring** (noun): *“a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour”*
- **Refactor** (verb): *“to restructure software by applying a series of refactorings without changing its observable behavior”*



The process of modifying our code to improve our software’s maintainability is called **Refactoring**. **Refactoring** is the process of changing a software system in such a way that **it does not alter (change) the external behaviour** of the code yet **improves the internal structure**. There are three keywords of the definition:

1. Changes made to the **internal structure** (in this case, the code).
2. It **does not change the behaviour** of the program. We need to make sure that our refactoring steps does not change the input output pair. We can still get the same output, by executing the refactored code with the same input as pre-refactored code.
3. The goal is to make our code **easier to understand** (readability) and **cheaper to modify** (maintainability).

Refactoring

- Given bad, even chaotic design, refactoring can rework it to well-designed code
- Each step in refactoring is **simple**
  - E.g. Push code up/down a hierarchy, move field to another class
- Small changes can radically improve the design

Refactoring

- Design is still an important stage in software development
- Refactoring lets software developers to improve the design during development process

Refactoring seems to have a huge role in software maintainability, but do not worry! Refactoring steps are actually very simple. For example, move a function to another class, or split a function to several separate functions. Those refactoring steps, although they are small, they often can radically improve the design if done appropriately. Sometimes, it can also improve program performance as well.

Design is still an important stage in software development, however agile we need to develop software. Refactoring does not work well if we do not have software design to refer to. Also, do not worry about imperfections in designing code, because we also can improve the design during the refactoring process.

Why Should You Refactor?

- Refactoring improves the design of software
- Refactoring makes software easier to understand
- Refactoring helps you find bugs
- Refactoring helps you program faster

When Should You Refactor?

Rule of Three

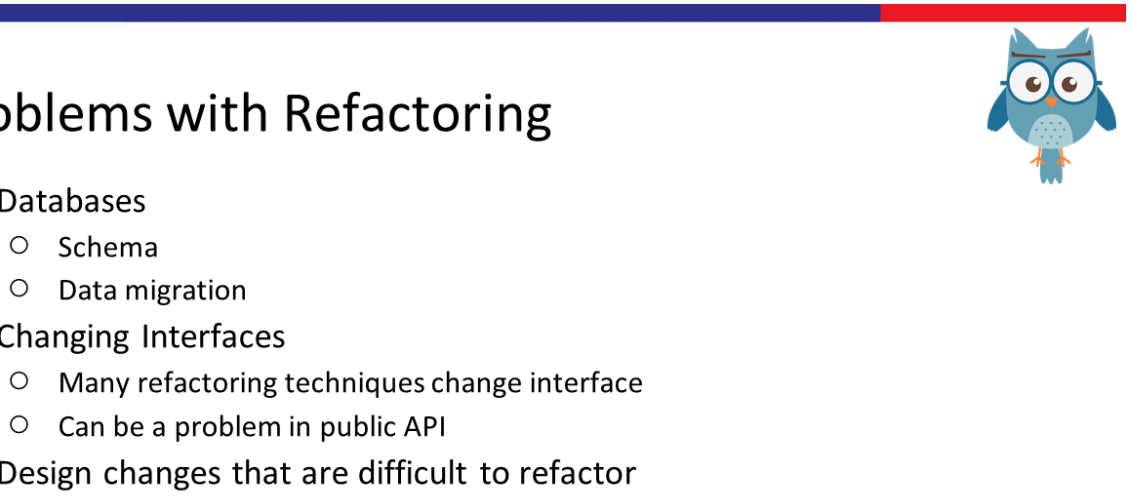
1. Refactor when you add function
2. Refactor when you need to fix a bug
3. Refactor as you do a code review

From previous explanations, we can conclude about why we refactor. There are 4 main keywords for this “why” question: **design improvement**, **readability**, **bug hunting**, and **performance**. Apart from three obvious reasons, refactoring can also be used to **hunt bugs**, because we often realise something wrong in our code when we refactor our code. Sometimes, bugs or design flaws are not always noticeable when designing or implementing the initial code, but it will come up more often when adding more functionalities to the code.

Then, when should we refactor our code? Fowler (1999) called these “rules” as the Rule of Three:

1. Refactor when we add new functionality to the code. Adding new functionality often means adding new complexity and dependency to another part of the code. By refactoring, we make sure that the new functionality won’t be a crumbled spaghetti in our code. We can also make sure that our code is extensible enough to add new functionalities in the future.
2. Refactor when we need to fix a bug. This is obvious 😊
3. Refactor as we do a code review. Let’s help our friends by reviewing their work and propose refactoring steps to improve their code’s readability. Review the code as soon as possible. Do not postpone the review until it’s too late.

Problems with Refactoring



There are problems that we often meet when we refactor. Oftentimes the problems are related to our program's dependencies. One of them is the database. When we refactor, and somehow, we need to change the database schema, make sure we also plan the data migration process. Make use of database migration and versioning tools such as Flyway (Java) to help on refactoring database schema.

Refactoring can also incorporate breaking changes, such as heavily changing interfaces. There are many techniques that require changing an interface. Again, we need to version our program (or our API) as reference that there are minor or major changes to the public interface.

Apart from versioning, we also need to thoroughly test our code after refactoring. That's why Test-Driven Development that we learned in previous submodules is very important. We can utilise unit tests to thoroughly check every functionality. But sometimes, radical changes when refactoring can also break unit tests as there are some dependency changes. If that ever happens, we also need to refactor our unit test code so that unit tests are focused on a single functionality. Make use of Test Doubles techniques, such as mocking and stubbing to isolate our unit tests from dependencies and make it usable even though we need major requirement changes.

5. Code Smells and Refactoring Steps

This submodule will explain how to identify code smells and what refactoring steps that can fix the code smells. By learning this module, students should be able to fix design flaws by using refactoring steps.

What are Code Smells?

Code Smells

- Symptoms that indicate whether a piece of code should be refactored.
- If the symptoms are ignored ⍰ difficult to maintain, more technical debts, less motivation.

Code smells are the symptoms that indicate whether a piece of code should be refactored. They often show up when we want to modify or add more functionalities to our code. If the symptoms are ignored and we choose to get along with the flow instead, the code will be increasingly difficult to maintain. Less maintainability means more technical debts to fix the code later, and it also means lesser motivation to push forward.

Fortunately, these code smells can now be detected early by using code quality check tools such as SonarQube or CodeClimate. Some tools, like SonarLint can also be used in your Integrated Development Environment (IDE). However, we might still need to check our code manually while we extend our code.

Categories of Code Smells

According to https://refactoring.guru/refactoring/smells:

- Bloaters
- OO Abusers
- Change Preventers
- Dispensables
- Couplers

There are 5 types of Code Smells:

- **Bloaters**: Parts of code, methods and classes that have increased to such huge proportions that they're hard to work with. Usually, these smells don't show up right away, rather they accumulate over time as the program evolves (and especially when nobody tries to eradicate them).
- **Object-Oriented Principle Abusers**: Parts of code that are resulted from incorrect or incomplete application of Object-Oriented Programming principles.
- **Change Preventers**: Parts of code that makes you need to do a lot of scattered changes at other parts of code, after modifying those parts.
- **Dispensables**: Useless, pointless, unneeded garbage part of code. Removing them will make your code cleaner, simpler, and easier to understand.
- **Couplers**: Parts of code that contribute to excessive coupling between classes.

Not all code smells and refactoring steps are explained here. You can learn more about each code smell in https://refactoring.guru/refactoring/smells. You can also learn more about each refactoring step in https://refactoring.guru/refactoring/techniques.

Bloaters

Long Method

Code Smells - Bloaters

- **Long Method**
  Common rule: A method's length should not exceed your monitor's height.

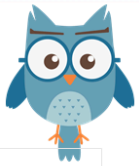
  Possible refactoring: **Extract Method**, **Replace Temp with Query**, **Introduce Parameter Object**, Preserve Whole Object, Replace Method with Method Object, **Decompose Conditional**

There is a best practice rule on functions that is common: Length of a function should not exceed your monitor's height. You might want to fix the limit to a certain number (for example: 25 lines). A long method needs more effort to read, as readers need to scroll the code. To avoid this, we can do some relevant refactoring steps. Remember that we might not need to do all those steps, only some of them:

- Use **Extract Method** to reduce the length of method body by extracting some of the logic out to a new method and replacing it with method call.
- You might need to use one of these refactoring steps if local variables and parameters interfere with the Extract Method process: **Replace Temp with Query**, Introduce Parameter Object, or Preserve Whole Object.
- If the method makes sense to be a separate object and you can't extract the method, do: Replace Method with Method Object.
- If the Long Method is caused by complex conditionals, you can use **Decompose Conditional** combined with the Extract Method.

Refactoring Step: Extract Method

Refactoring Step: Extract Method



Separate fragment(s) of a method to a new method.

```
// Before
void printOwing(double amount) {
  printBanner();

  // Print details
  System.out.println("Name: " + name);
  System.out.println("Amount: " + amount);
}
```

```
// After
void printOwing(double amount) {
  printBanner();
  printDetails(amount);
}

void printDetails(double amount) {
  System.out.println("Name: " + name);
  System.out.println("Amount: " + amount);
}
```

Extract Method means separating fragment(s) of a method to one (or several) new method(s). The main goal of this refactoring step is to reduce code duplication and isolate independent parts of the code. Make sure to give meaningful names to the methods created by this step.

Refactoring Step: Replace Temp with Query

Refactoring Step: Replace Temp with Query

Separate temporary or local variable expression to a reusable query method.
Can be used before **Extract Method** to make extracting a method easier.

```
// Before
double getPrice() {
  int basePrice = qty * itemPrice;
  double discountFactor;
  if (basePrice > 1000)
    discountFactor = 0.95;
  else
    discountFactor = 0.98;
  return basePrice * discountFactor;
}
```

```
// After
double getPrice() {
  int basePrice = basePrice();
  double discountFactor;
  if (basePrice > 1000) { ...
  // Omitted for brevity
  return basePrice * discountFactor;
}
private int basePrice() {
  return qty * itemPrice;
}
```

This refactoring step is often used as a groundwork for the Extract Method. This will also make more meaningful methods that encapsulate a calculation. This refactoring step can also help to reduce duplication if the calculation is used more than once (or used in more than one method).

Refactoring Step: Decompose Conditional

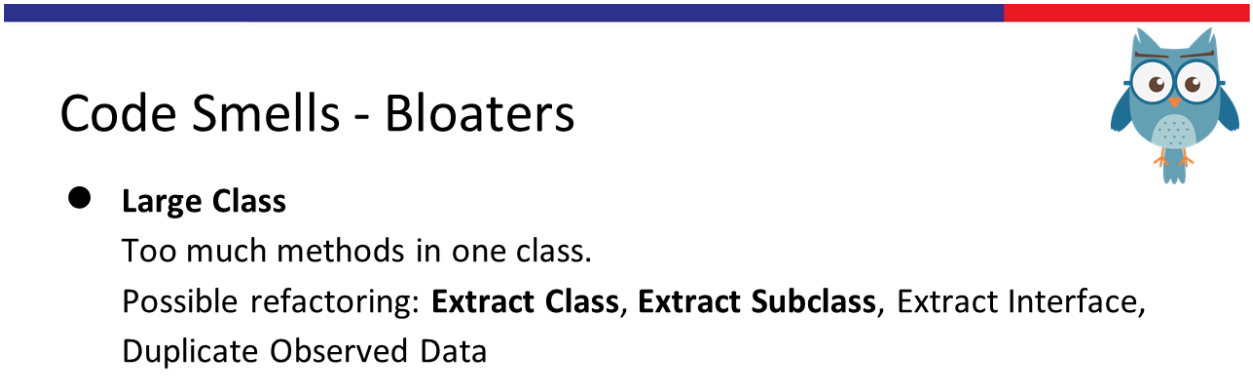
Extract complicated parts of the conditional into separate methods.

```
// Before
if (date.before(SUMMER_START) ||
date.after(SUMMER_END)) {
  charge = quantity * winterRate +
winterServiceCharge;
}
else {
  charge = quantity * summerRate;
}
```

```
// After
if (isSummer(date)) {
  charge = summerCharge(quantity);
}
else {
  charge = winterCharge(quantity);
}
```

The Decompose Conditional step is a variation of the “Extract Method” that focuses on simplifying conditionals. This step will extract conditional expressions and calculations inside a conditional to separate methods. This step will make the code more readable because it substitutes complex expressions into meaningfully named methods.

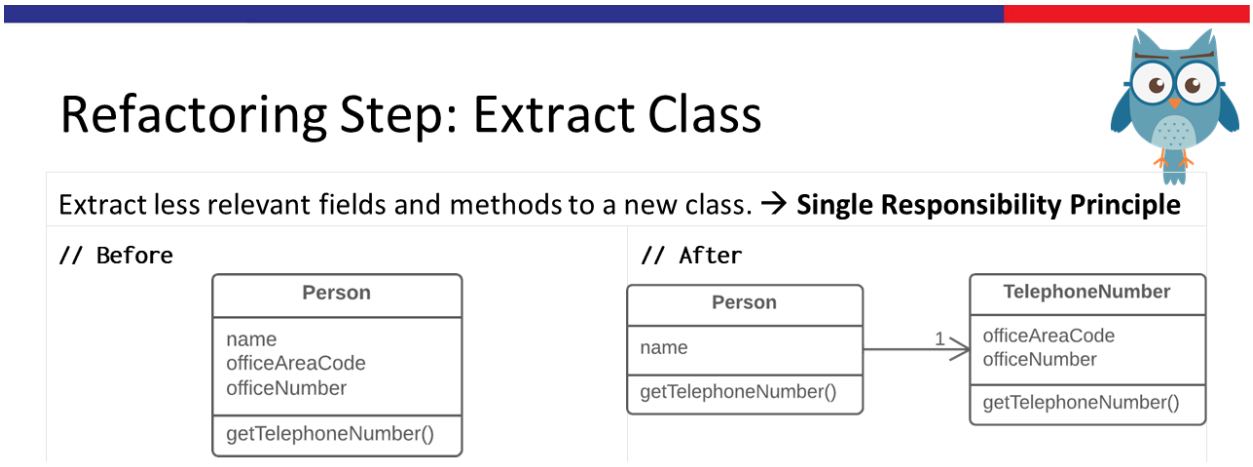
Large Class



Have you ever built a single class that contains **thousands** of lines of code? That is the simplest example of a Large Class. Large Classes are often caused by having too many methods inside a class. To refactor, you need to review:

- If there are any groups of methods that can be grouped into a different class, or if there are any irrelevant methods that should belong to a new class, use the **Extract Class** step.
- If there are methods that are only used in some of specific conditions or subtypes of this class, use the **Extract Subclass** step.
- If the class is responsible for graphical user interface, you may want to try to move some of the relevant data and behaviour into a separate domain object. To refactor this specific case, use the Duplicate Observed Data step.
- If your code involves interfaces, to finalise refactoring a Large Class, use the Extract Interface step.

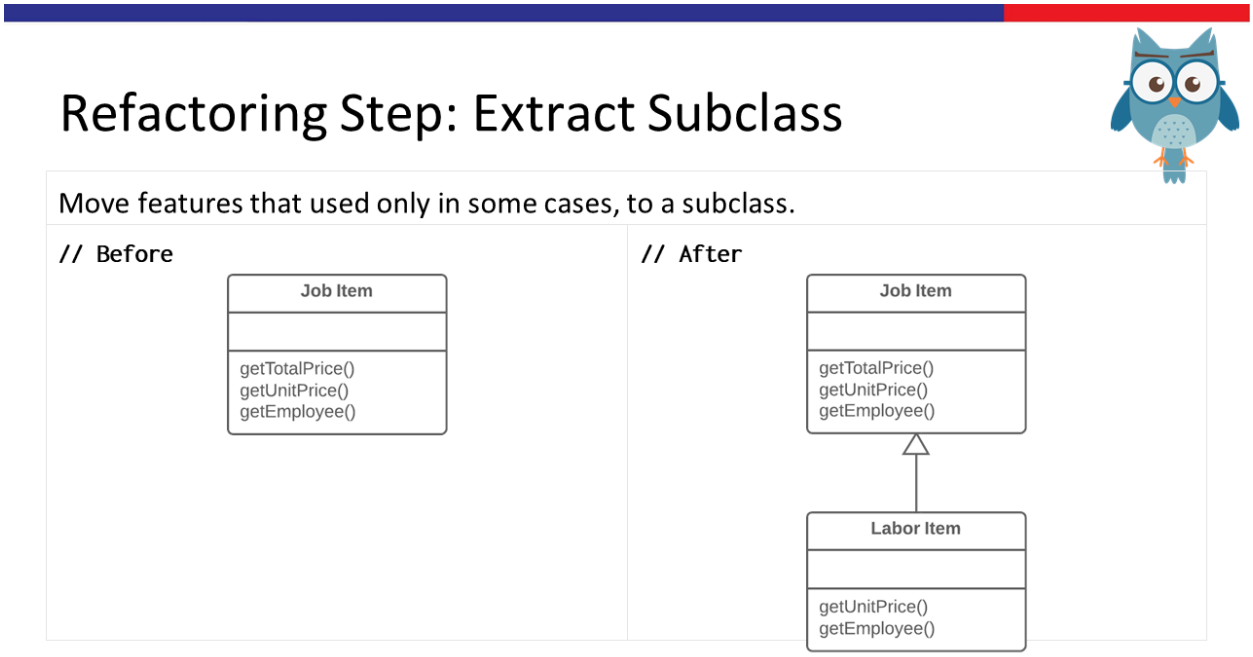
Refactoring Step: Extract Class



In Module 3 you have learned about S.O.L.I.D. Principle, one of them is Single Responsibility Principle. This step extracts less relevant fields and methods to a new class. If the original class needs to interact with that data, replace it with a method call. This refactoring step assures that our code fulfils Single Responsibility Principle if done correctly.

However, if we overdo it, the classes will be tightly coupled to each other. This is often because we extract things that still make sense if they belong to the original class. To revert those changes, we can do the opposite step, called **Inline Class** refactoring step.

Refactoring Step: Extract Subclass



This is the variation of the Extract Class step, but this step involves inheritance. This step is useful when fields/methods are only used in one (or some) subtype(s) of the original class, and the original class has no relevant subclasses yet. To do this step, we need to make a subclass, then extract those fields/methods to the new subclass.

If the subclass in which the fields/methods are moved into already exists, then it's called Push Down Field/Method step rather than Extract Subclass. If we need to create a superclass rather than a subclass to extract the fields/methods, then it is called the Extract Superclass step.

Long Parameter List

- **Long Parameter List**
  Too much parameters for one method.
  Possible refactoring: **Replace Parameter with Method Call**, Preserve Whole Object, **Introduce Parameter Object**

Have you ever found a method that needs a lot of arguments, that makes the method call very long? This is called Long Parameter List code smell. This code smell is often caused by our efforts to make our methods more independent of each other. Those arguments are needed to get information that can't be obtained due to reduced inter-class relationships. To refactor, consider some (or maybe all) of these cases:

- If some of the arguments are just results of method calls of another object, use **Replace Parameter with Method Call**. This object can be placed in the field of its own class or passed as a method parameter.
- Instead of passing a group of data received from another object as parameters, pass the object itself to the method, by using **Preserve Whole Object**.
- But if these parameters are coming from different sources, you can pass them as a single parameter object via **Introduce Parameter Object**.

Refactoring Step: Replace Parameter with Method Call

Refactoring Step: Replace Parameter with Method Call

If some parameters can be obtained with method call, move the call inside the method.

```
// Before
int basePrice = quantity * itemPrice;
double seasonDiscount = this.getSeasonalDiscount();
double fees = this.getFees();
double finalPrice = discountedPrice(basePrice, seasonDiscount, fees);

// After
int basePrice = quantity * itemPrice;
double finalPrice = discountedPrice(basePrice);
```

The example shows method calls that are required to obtain data for the parameter, moved into the internal implementation of the method. Now instead of asking for 3 parameters, it will only ask for 1 parameter. This step is suitable if the methods that are called are from the class itself

(`this`) or from a library which we don't need to obtain the object to call (for example: `static` methods). If the method call is from another object, Preserve Whole Object is more suitable.

Refactoring Step: Preserve Whole Object

Refactoring Step: Preserve Whole Object

Rather than passing multiple values from an object, pass the whole object instead.

```
// Before
int low = daysTempRange.getLow();
int high = daysTempRange.getHigh();
boolean withinPlan = plan.withinRange(low, high);

// After
boolean withinPlan = plan.withinRange(daysTempRange);
```

This is a variation of Replace Parameter with Method Call refactoring step. This variation is suitable if some of the data are obtained from method calls to another object. The step is the same as Replace Parameter with Method Call: moving method calls inside the internal implementation of the method, but the difference is that we will preserve the whole object as a parameter. For example, if the method needs 4 parameters which 3 of them are from one object, just accept that object as a parameter, so the method now only needs 2 parameters.

Refactoring Step: Introduce Parameter Object

Extract relevant parameters to a new parameter class and replace those parameters to an object type. We can also call this Data Transfer Object (DTO).

```
// Before
```

Customer
amountInvoicedIn (start : Date, end : Date) amountReceivedIn (start : Date, end : Date) amountOverdueIn (start : Date, end : Date)

```
// After
```

Customer
amountInvoicedIn (date : DateRange) amountReceivedIn (date : DateRange) amountOverdueIn (date : DateRange)

```
DateRange(date: Date, end: Date)
```

If there are parameters that are relevant to be grouped into an object, use this step. Create a new parameter object that contains those data as its fields, then use the Preserve Whole Object step to get the data. Data Transfer Object (DTO) is a good example of a “parameter object”.

Code Smells - Bloaters

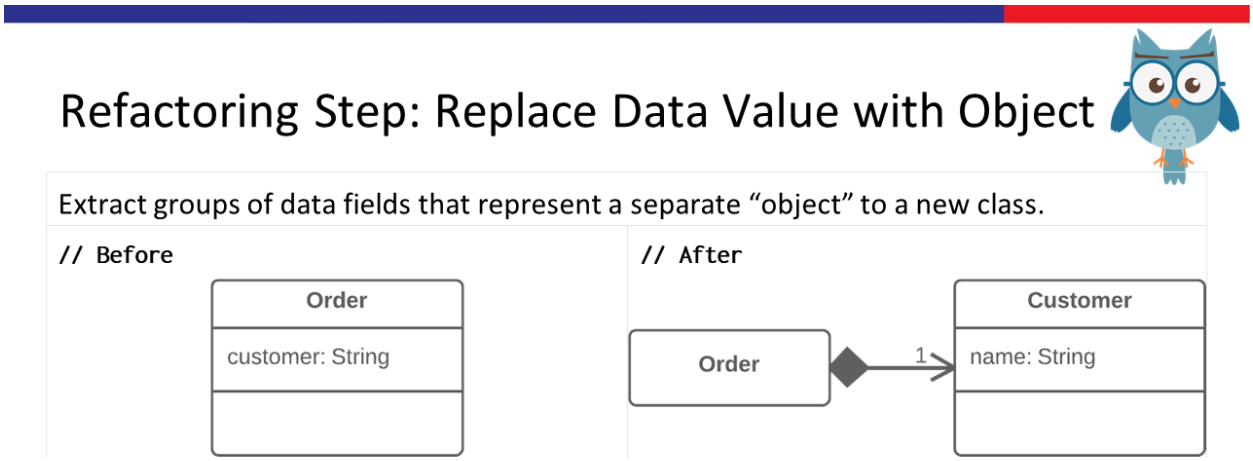
- **Primitive Obsession**
  Too much relying on primitive data types to define constants and information coding (for example: `int STATUS_SUCCESS = 1`).

  Possible refactoring:
  Replace Data Value with Object, **Replace Type Code with Class** (Enum), **Extract Class**, **Introduce Parameter Object**, Replace Array with Object, or choose one of: (Replace Type with Subclasses, Replace Type Code with State/Strategy)

This code smell happens when a code is relying too much on primitive data types to define constants and doing simple tasks such as information coding. Primitive constants often resemble a set of types or status. Using primitives instead of simple objects will increase duplications and reduce code flexibility. There are four ways to refactor this code smell according to the usage of primitives:

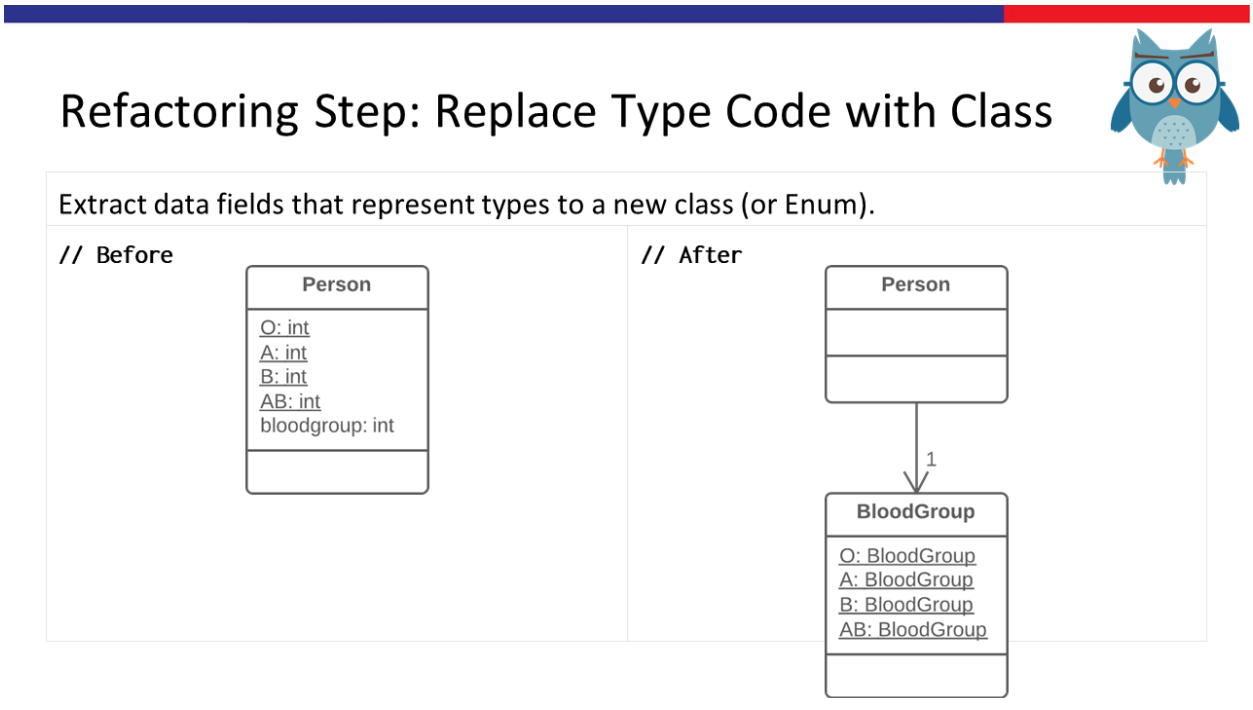
- If the class has a large variety of primitive fields, it may be possible to logically group some of them into their own class. Even better, move the behaviours associated with this data into the class too. For this task, try **Replace Data Value with Object** step.
- If the values of primitive fields are used in method parameters, use **Introduce Parameter Object** and/or Preserve Whole Object step.
- If complicated data is coded in variables, use one of: **Replace Type Code with Class**, Replace Type Code with Subclasses, or Replace Type Code with State/Strategy.
- If there are arrays among the variables, use Replace Array with Object.

Refactoring Step: Replace Data Value with Object



This refactoring step is a variation of **Extract Class** that is used for one specific case: move a group of data variables and its related methods to a new class. This will improve code organisation as variables inside of a class now becoming more relevant for each other.

Refactoring Step: Replace Type Code with Class



If the constants are simulating types or status, we can use a data class (or in Java, we call this as **Enum**). Extract those data fields to a new class. Implement methods that are helpful, such as converter from an Enum object to a primitive and vice versa, or a validator method to check whether a primitive is a correct type.

Code Smells - Bloaters

- **Data Clumps**
  Too much data variables in a class that are either irrelevant or can be combined, often caused by “copypasta programming”.

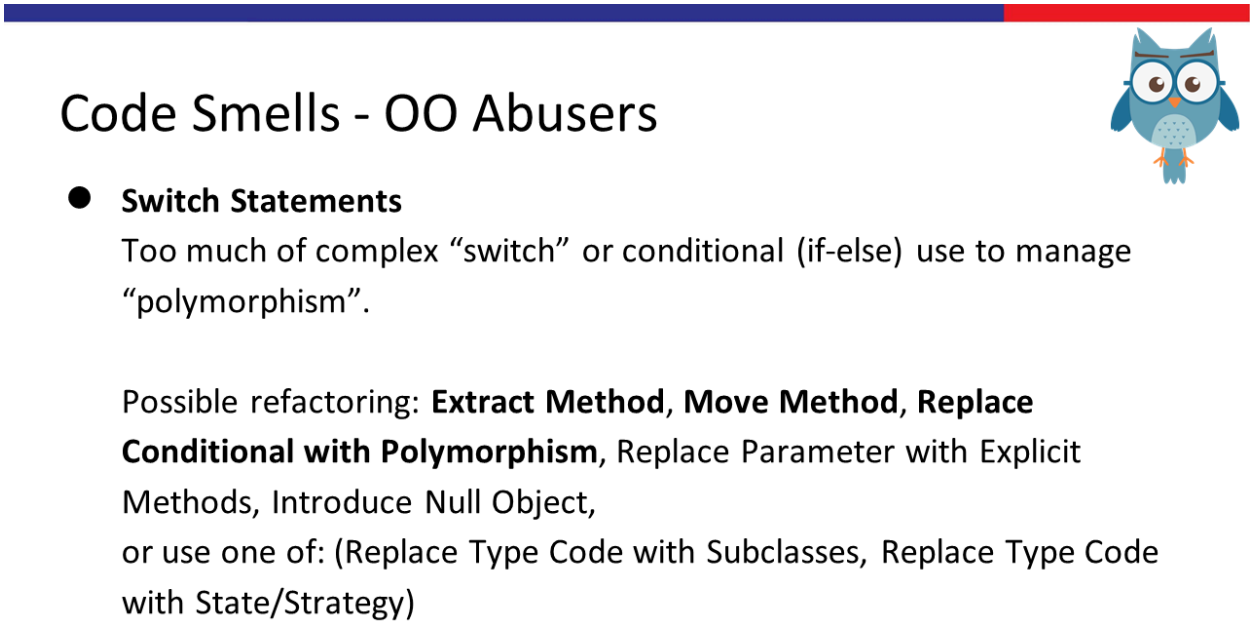
  Possible refactoring: **Extract Class**, Introduce Parameter Object, **Preserve Whole Object**

Sometimes different parts (we can call “clumps”) of the code contain identical groups of variables (such as parameters for connecting to a database). This is often caused by poor program structure, or maybe rushed, copypasta programming. These clumps should be turned into their own classes. Refactoring these data clumps often results in better code organisation and less code duplication. To refactor, consider these cases:

- If repeating data consists of fields of a class, use the **Extract Class** step to move them into their own class.
- If the same data clumps are passed to parameters of methods, use the Introduce Parameter Object step.
- If only some of the data passed to parameters of methods, use the **Preserve Whole Object** step.

Object-Oriented Principle Abusers

(Excessively Complex) Switch Statements



The common best practice of Object-Oriented programming is to have the least amount of `switch` statements possible. This is related to the Polymorphism aspect of Object-Oriented programming. To refactor this code smell, consider these refactoring steps:

- To isolate `switch` statements and put it into the right class, use **Extract Method** and Move Method.
- If a `switch` is based on type codes, use one of either **Replace Type Code with Subclasses** or **Replace Type Code with State/Strategy Pattern**. If type codes are related to a current state of the object, use State pattern. If type codes are related to current user strategy/choice, use Strategy pattern.
  (We will discuss about Design Patterns later at Module 7)
- If a `switch` is based on subtypes or variations that can be modelled using inheritance, use **Replace Conditional with Polymorphism**.
- If there aren’t too many conditions in the operator and they all call the same method but with different parameters, polymorphism will be unnecessary. You might want to use **Replace Parameters with Explicit Methods**.
- If one of the conditions is `null`, then **Introduce Null Object**.

Refactoring Step: Replace Conditional with Polymorphism

If each condition refers to a type/property: create subclasses matching the branches.

```
// Before
class Bird {
  // ...
  double getSpeed() {
    switch (type) {
      case EUROPEAN:
        return getBaseSpeed();
      case AFRICAN:
        return getBaseSpeed() - getLoadFactor() * numberOfCoconuts;
      case NORWEGIAN_BLUE:
        return (isNailed) ? 0 : getBaseSpeed(voltage);
    }
    throw new RuntimeException("Should be unreachable");
  }
}
```

This refactoring step is useful when the conditionals (switch-case or if-else if-else) are operating various tasks in a method that vary based on interface it implements, value of a field, or result of calling one of the methods. This signifies that the conditionals are representing possible subtypes or variations of that class.

Refactoring Step: Replace Conditional with Polymorphism

If each condition refers to a type/property: create subclasses matching the branches.

```
// After
abstract class Bird {
  // ...
  abstract double getSpeed();
}

class European extends Bird {
  double getSpeed() {
    return getBaseSpeed();
  }
}
```

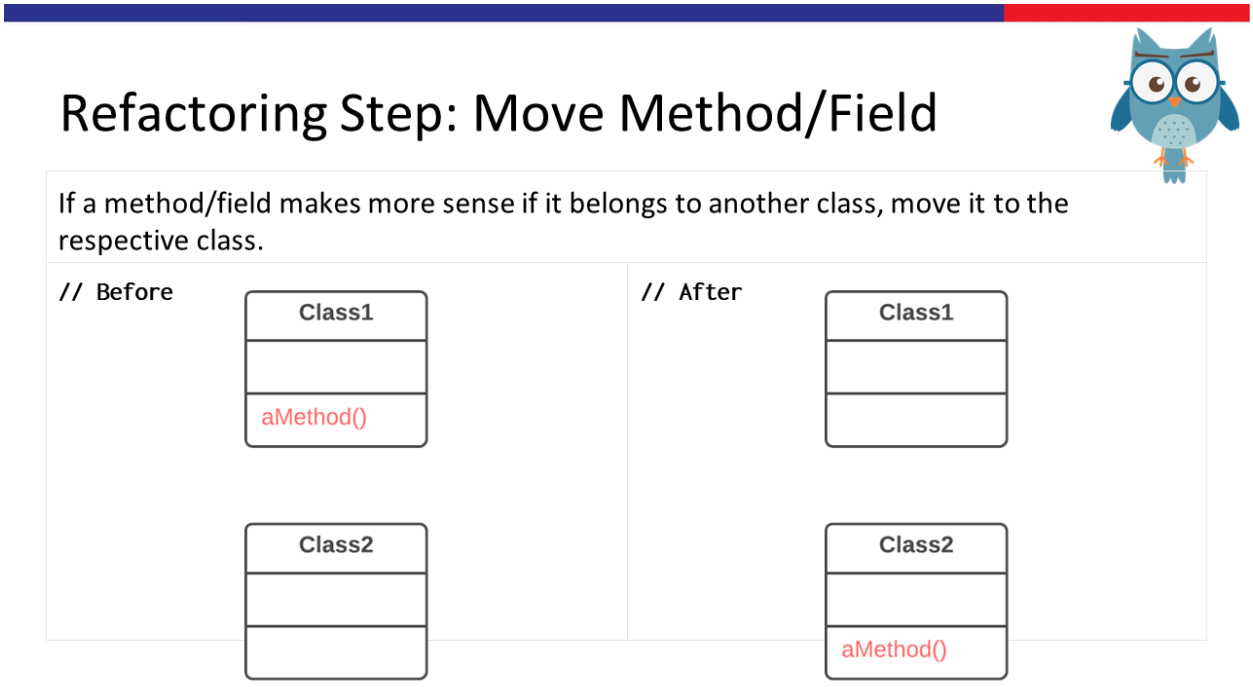
```
class African extends Bird {
  double getSpeed() {
    return getBaseSpeed() - getLoadFactor() *
numberOfCoconuts;
  }
}
class NorwegianBlue extends Bird {
  double getSpeed() {
    return (isNailed) ? 0 :
getBaseSpeed(voltage);
  }
}

// Somewhere in client code
speed = bird.getSpeed();
```

For better readability, split the conditionals using polymorphism. Create new subclasses according to each of the conditions. Then, set the original class as an abstract class. This way, readers of your code will be focused on each variant or subtype when analysing your code. This refactoring step also benefits you if these conditionals happen in many methods inside such a class, as this refactoring step will increase your code's adherence to Open-Closed principle.

This refactoring also adheres to the **Tell, Don't Ask** principle**.** Instead of asking an object about its state and then performing actions based on this, it's much easier to simply tell the object what it needs to do and let it decide for itself how to do that.

Refactoring Step: Move Method/Field



This refactoring step is simple, move a method or field that makes more sense if it belongs to another class. There are variants of this refactoring steps, such as: Push Down/Push Up Method/Field.

Refused Bequest

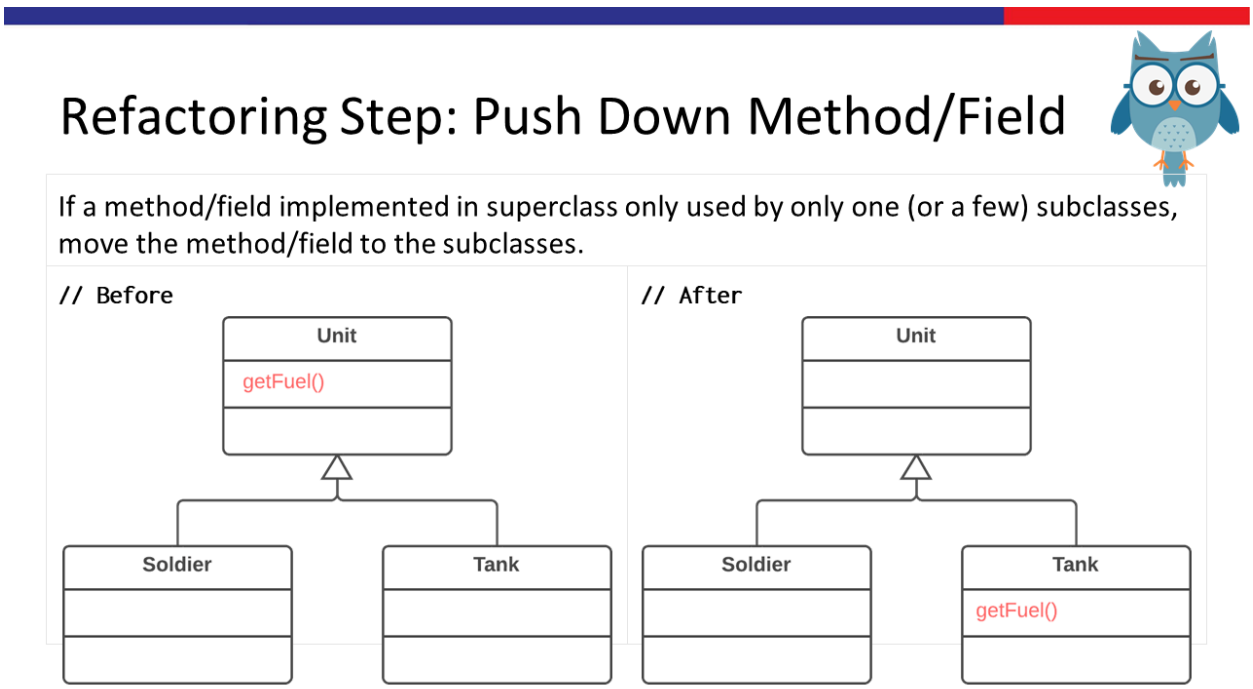
Code Smells - OO Abusers

- **Refused Bequest**
  Irrelevant class that becomes a subclass just because there are some similarities of the methods.

  Possible refactoring: **Extract Superclass**, Push Down Method, Push Down Field, **Replace Inheritance with Delegation**

This code smell happens because a class is forced to inherit another class just because there are some similarities of the methods/functionalities. This often happens because we overly strive for code reuse but didn't evaluate whether the inheritance structure makes sense or not. For example, someone put `Chair` as a subclass of `Animal` just because they have legs like other animals. To refactor this code smell, consider the following:

- If the inheritance is still appropriate: Extract all fields and methods needed by the subclass from the parent class, make a new superclass using the **Extract Superclass** step. Then rearrange methods/fields according to the new structure by using **Push Down/Push Up Method/Field**.
- If the inheritance does not make sense at all (for example: subclass and superclass have nothing in common), do **Replace Inheritance with Delegation** step.

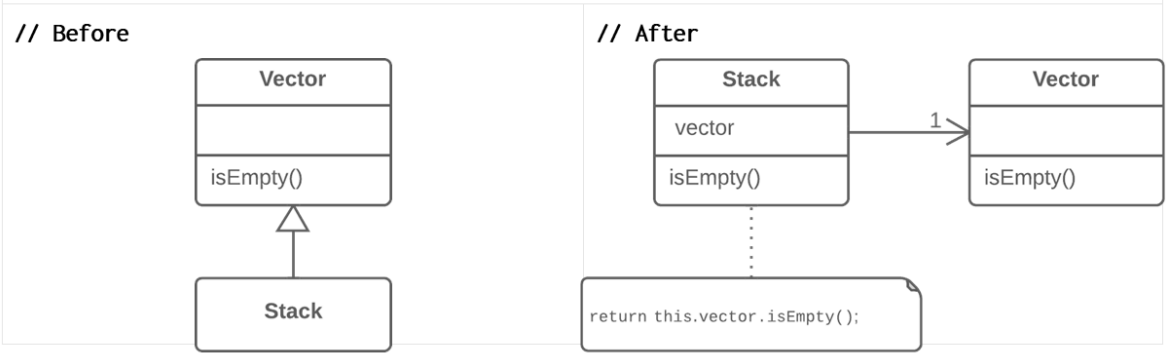


This is a variant of **Move Method/Field**, but this step will move methods/fields from a superclass to a subclass in an inheritance structure. This refactoring step is used when superclass fields/methods are only used by one (or a few) subclasses. This will make a subclass that does not need such a function won't inherit that unnecessary function.

The opposite of this step is Push Up Method/Field that will move methods/fields from a subclass to a superclass. This is used when a subclass fields/methods are deemed to be useful for its original class and all of its siblings, so they can be generalised upwards.

Refactoring Step: Replace Inheritance with Delegation

If a subclass uses only a portion of the methods of its superclass (or it's not possible to inherit superclass data): Create a field and put a superclass object in it, delegate methods to the superclass object, and get rid of inheritance.



This refactoring step can be used if the inheritance does not make sense due to the subclass only using a portion of the methods (or even nothing) of its superclass, or somehow the subclass is not able to inherit superclass data. Create a field to put the superclass object in it. Then, delegate existing methods to the superclass object. Then, remove the inheritance.

The opposite of this refactoring step is Replace Delegation with Inheritance that is useful when the delegated object actually makes sense to be modelled using inheritance.

Temporary Field

Code Smells - OO Abusers

- **Temporary Field**
  Instance variables of a class that are only used occasionally, often left empty when not used.

  Possible refactoring: **Extract Class**, **Introduce Null Object**

Temporary fields are often created because of a large algorithm that requires many inputs or data. These fields are often left empty and only used when the algorithm is active. To get rid of temporary fields, we can use **Extract Class** followed by Replace Method with Method Object to reduce the amount of temporary fields, and then **Introduce Null Object** to check whether a value exists in such temporary fields.

Refactoring Step: Introduce Null Object

Refactoring Step: Introduce Null Object

Avoid scattered `null` checks by making classes to represent `null` with default behaviour.

```
// Before
if (customer == null) {
  plan = BillingPlan.basic();
}
else {
  plan = customer.getPlan();
}
```

```
// After
class NullCustomer extends Customer {
  boolean isNull() {
    return true;
  }
  Plan getPlan() {
    return new NullPlan();
  }
  // Some other NULL functionality.
}
// Replace null values with Null-object.
customer = (order.customer != null) ?
    order.customer : new NullCustomer();
// Use Null-object as if it's normal subclass.
plan = customer.getPlan();
```

Null Objects are useful to give more context of an “empty” condition rather than using just `null`. By giving more context, we make our code more readable. Null Objects can also be operated just like any other object of its type (hence why it inherits the same superclass as its non-empty counterparts).

Alternative Classes with Different Interfaces

- **Alternative Classes with Different Interfaces**
  Two classes with identical functions but different names

  Possible refactoring: **Rename Method**, Move Method, **Extract Superclass**

Sometimes we forget that we have already implemented a functionality inside of our code. This often happens because of the lack of coordination when doing a team project. To streamline those alternative classes with same functions but different interfaces, use Rename Method, Move Method, Add Parameter, and Parameterize Method. If only some parts are duplicated, use Extract Superclass when you think inheritance is the correct way to model the differences. After that, continue to check which class can be removed.

Refactoring Step: Rename Method

There is no slide dedicated to explaining this refactoring step because it's too simple. This step will rename a method using a more meaningful name or an intended name.

Change Preventers

Divergent Change

This code smell happens when you find yourself having to change many unrelated methods when you make changes to a class. For example, when adding a new product type you must change the methods for finding, displaying, ordering, adding, and removing products. To solve this problem, use **Extract Class** and its variants (such as Extract Subclass or Extract Superclass) to figure out tight coupling and duplications that happen within a class.

Code Smells - Change Preventers

- **Parallel Inheritance Hierarchies**
  When you create a subclass of a class, you find yourself needing to also create a subclass for another class.

  Possible refactoring: De-duplicate parallel class hierarchies by **Move Method** and/or **Move Field**

If this code smell exists in your code, when you create a subclass for a class, you find yourself needing to create a subclass for another class. This happens due to our confusion that develops over time when adding more classes to the system. To fix this problem, we need to de-duplicate parallel class hierarchies in two steps. First, make instances of one hierarchy refer to instances of another hierarchy. Then, remove the hierarchy in the referred class by using **Move Method** and/or **Move Field**.

Code Smells - Change Preventers

- **Shotgun Surgery**
  To change something, you need to also change many methods in **many different classes**.

  Possible refactoring:
  Do **Move Method** and/or **Move Field**,
  then do **Inline Class** (the opposite of Extract Class) if the class now becomes almost empty.

The difference between this code smell and Divergent Changes is that this code smell will urge you to change many methods in many different classes if you want to change something. This often happened because of overthinking when solving the Divergent Changes problem. Instead of solving the problem, we instead split up single responsibility to several different classes that are tightly coupled to each other.

To solve the problem, we need to do two steps:

1. Use **Move Method and/or Move Field** to move existing class behaviours into a single class. If there's no class appropriate for this, create a new one.
2. If moving code to the same class leaves the original classes almost empty, get rid of those classes using **Inline Class**. This step will merge those classes into a single class.

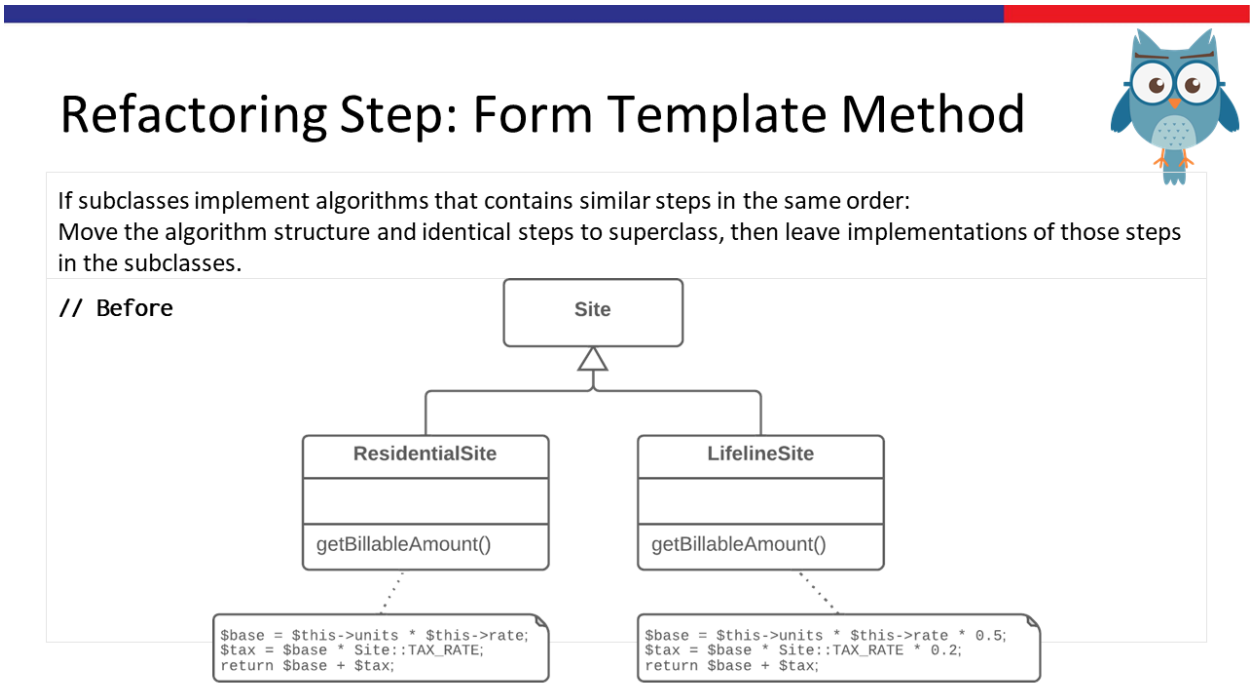
Dispensables

Duplicated Code

Duplicated code often happens when we rush the code. Duplicating things (doing “copypasta programming”) often makes ultra-fast coding possible, but it will hurt you in the end. To fix duplicated codes, consider these steps:

- If the duplicated code exists in more than one class of the same level, use the **Extract Method**. If the levels are different or there is no inheritance structure, do either **Extract Class** or **Extract Superclass**.
- If the duplicated code is similar but not completely identical (such as they have the same order but different implementation for each step), use **Form Template Method**.

Refactoring Step: Form Template Method



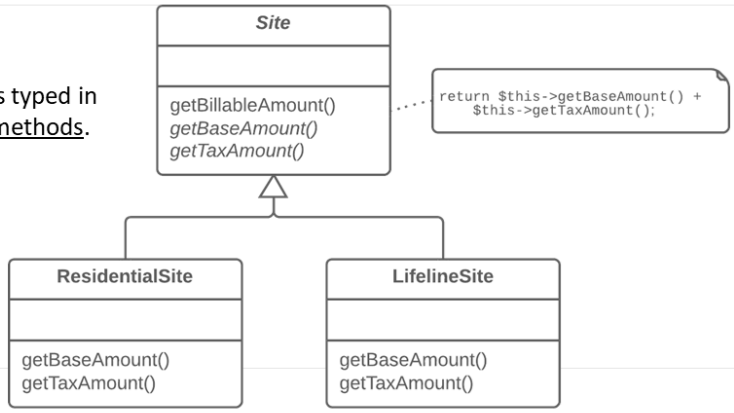
Template Method pattern is a design pattern that is useful to organise an algorithm that varies on implementations of each step but has the same steps and order of steps for every possible variation. This design pattern will create a “template” that defines the structure of the algorithm, then each step will be defined as an “abstract method” in the superclass. Implementation of each step will be defined by each subclass.

Refactoring Step: Form Template Method

If subclasses implement algorithms that contains similar steps in the same order:
Move the algorithm structure and identical steps to superclass, then leave implementations of those steps in the subclasses.

```
// After
```

Note: The methods typed in italic are abstract methods.



This is an example of the Template Method. The `getBaseAmount()` and `getTaxAmount()` are the steps of the `getBillableAmount()` algorithm. Each subclass will implement every “step”. Duplications are still possible if some of the steps are the same for some of the subclasses, but it will be more manageable.

If implementation of a step is the same for all of the subclasses, do the Push Up Method. Push the implementation to the superclass, then make the step method a regular method instead of an abstract method.

Lazy Class

- **Lazy Class**: A useless class.
  Possible refactoring: **Collapse Hierarchy** (the opposite of Extract Subclass/Superclass), **Inline Class**

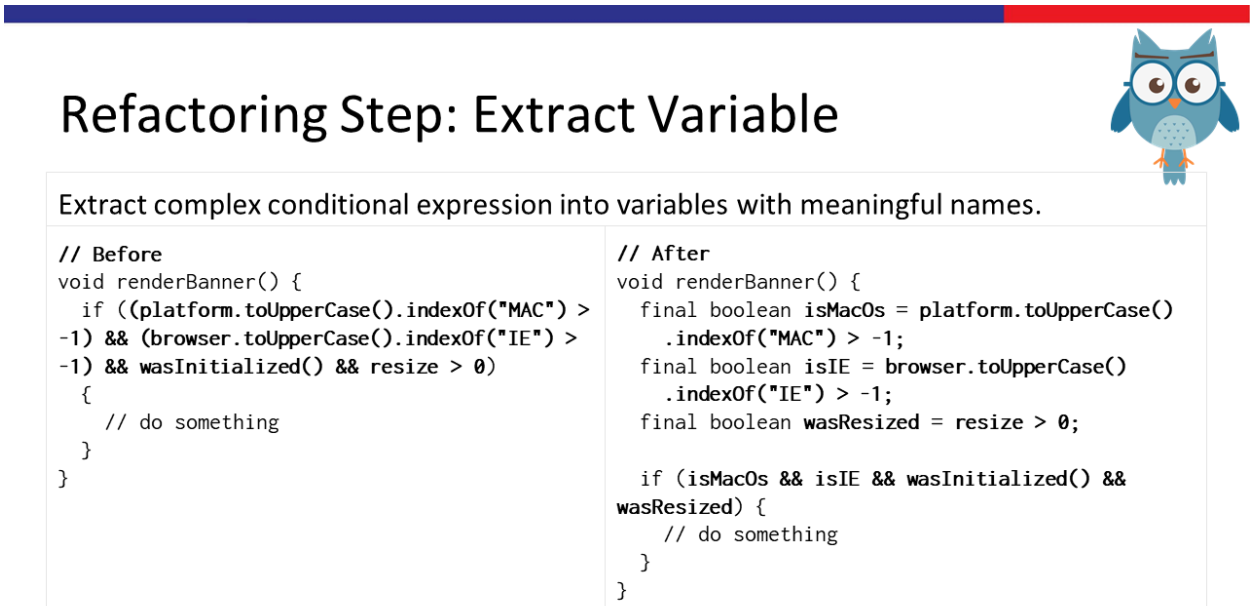
Lazy Class is a terminology for useless class. It does not have any uses in other classes. To fix that, use **Collapse Hierarchy** (the opposite of Extract Subclass/Superclass, which will remove the inheritance structure) and **Inline Class**.

Comments

- **Comments**: Too much explanatory comments.
  Possible refactoring: **Extract Variable**, Extract Method, **Rename Method**, **Introduce Assertion**

Comments sometimes are useful, but clean codes are codes that use the least number of comments. Instead of using explanatory comments, why not make your code explain itself using meaningful names. Use refactoring steps such as Rename Method, Extract Variable, Extract Method, and Introduce Assertion whenever appropriate.

Refactoring Step: Extract Variable



Extract Variable will extract complex expressions (actually, it's not only for conditionals, it can be for anything) to a variable with a meaningful name.

Refactoring Step: Introduce Assertion

Refactoring Step: Introduce Assertion

Replace explanatory comments with assertions instead (this isn't a JUnit test assertion).

```
// Before
double getExpenseLimit() {
  // Should have either expense limit or
  // a primary project.
  return (expenseLimit != NULL_EXPENSE) ?
    expenseLimit :
    primaryProject.getMemberExpenseLimit();
}
```

```
// After
double getExpenseLimit() {
  Assert.isTrue(expenseLimit != NULL_EXPENSE ||
primaryProject != null);

  return (expenseLimit != NULL_EXPENSE) ?
    expenseLimit:
    primaryProject.getMemberExpenseLimit();
}
```

Assertions are not only useful for unit testing purposes. It is also useful as a documentation and as a validation to check prerequisites of an input. Use assertions if you feel it is necessary instead of using comments. Comments are not binding, assertions are.

Speculative Generality

Code Smells - Dispensables

- **Speculative Generality**: The result of overthinking future-proofing. Possible refactoring: **Collapse Hierarchy** (the opposite of Extract Subclass/Superclass), **Inline Class**, Remove Parameter, Rename Method

This is why overthinking, although it's natural and humanly, it's not good for you. Too much speculation will result in more confusion as humans often can't predict the future (unless you are a certified fortune teller 😂).

To clean up your overthinking mess, use **Collapse Hierarchy**, **Inline Class**, Remove Parameter, and/or Rename Method. Choose which refactoring steps you need according to the current condition. Again, don't think too much about the future. Don't worry, your program's user (or the market demand) will tell you if you need to add a new thing in the future.

Data Class

- **Data Class**: A class that only contains fields and their getters/setters.
  Possible refactoring:
  - If all fields are public: **Encapsulate Field**, Encapsulate Collection
  - If fields are irrelevant (or more relevant for another class): **Move Method**, **Extract Method**
  - If fields are only used inside the class: Remove Setting Method, **Hide Method**

Sometimes, data classes are necessary. However, too much of it will confuse you, especially when the class can't operate the data it contains using its own methods. Remember that the true power of an object is that they can operate or behave according to their own data. To fix this code smell, use these steps:

- If all fields are public, use **Encapsulate Field** or Encapsulate Collection.
- If there are irrelevant fields (or maybe more relevant for another class), use the **Move Method** or **Extract Method**.
- If fields and behaviours are only used inside the class, use the **Remove Setter Method** and **Hide Method**.

Refactoring Step: Encapsulate Field/Hide Method

Refactoring Step: Encapsulate Field/Hide Method

Make a field/method private to encapsulate or to prevent outside access.

```
// Before
class Person {
  public String name;

  public UUID generateToken() {
    return UUID.randomUUID();
  }
}
```

```
// After
class Person {
  private String name;
  public String getName() {
    return name;
  }
  public void setName(String arg) {
    name = arg;
  }

  private UUID generateToken() {
    return UUID.randomUUID();
  }
}
```

This refactoring step basically fulfils the Encapsulation principle of Object-Oriented programming. Make those fields/methods private whenever possible, then use getters/setters.

Couplers

Feature Envy

Code Smells - Couplers

- **Feature Envy**
  A method that accesses the data of another object, more than its own data.

  Possible refactoring: **Move Method**, **Extract Method**

The keyword here is **envy**. A class that is envious of the data that belong to another class. This class has the operations but doesn't have the necessary data to do so. This condition often happens after fields are moved to another class, but the operations have not yet moved.

There are several conditions to consider when refactoring this code smell:

- If a method clearly should be moved to another place, use the **Move Method**.
- If only part of a method accesses the data of another object, use **Extract Method** to move the part in question.
- If a method uses functions from several other classes, first determine which class contains most of the data used. Then place the method in this class along with the other data. Alternatively, use the **Extract Method** to split the method into several parts that can be placed in different places in different classes.

By solving this code smell, we will reduce unnecessary duplications, and make code organisation better because data and their operations are now in a single place.

Code Smells - Couplers

- **Inappropriate Intimacy**
  Usage of another class's internal methods.
  Rule: A class should know other classes as little as possible.

  Possible refactoring:
  - If the class doesn't need those methods: **Move Method** and/or Move Field
  - To make relation between the classes is "official": **Extract Class**, Hide Delegate
  - If they are mutually interdependent: Change Bidirectional Association to Unidirectional
  - If this happens between a superclass and a subclass: Replace Delegation with Inheritance

This code smell means that there is a class that uses internal fields and methods of another class. This condition often happens due to incomplete refactoring that involves moving internal fields and methods to other classes. Remember that a class should know other classes as little as possible.

To fix this code smell, consider these conditions:

- The simplest solution is to use **Move Method** and/or Move Field to move parts of one class to the class in which those parts are used. This works only if the first class truly doesn't need these parts.
- Another solution is to use **Extract Class** and **Hide Delegate** on the class to make the code relations "official".
- If the classes are mutually interdependent, you should use Change Bidirectional Association to Unidirectional.
- If this "intimacy" is between a subclass and the superclass, consider Replace Delegation with Inheritance.

Message Chains

Code Smells - Couplers

- **Message Chains**
  If a method calls on many other method in other classes, making long message chains (such as: `A.methodA().methodB().methodC()...`).

  Possible refactoring:
  Hide Delegate, or
  Extract Method and/or **Move Method**

A message chain occurs when a client requests another object, that object requests yet another one, and so on. These chains mean that the client is dependent on navigation along the class structure. Any changes in these relationships require modifying the client.

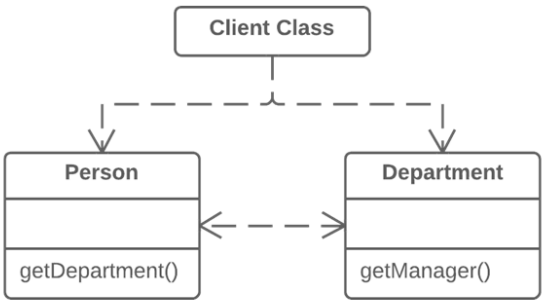
These are possible solutions to fix this code smell:

- To delete a message chain, use **Hide Delegate**.
- Sometimes it's better to think of why the end object is being used. We can use the **Extract Method** for that functionality, then move it to the beginning of the chain by using the **Move Method**.
- If those classes have identical intentions (for example: as a step to filter or process data), then we can redesign those classes using the Chain of Responsibility design pattern. (We will discuss about Design Patterns later at Module 7)

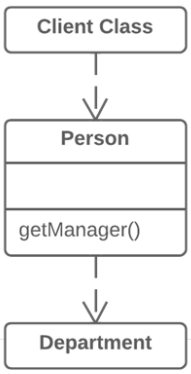
Refactoring Step: Hide Delegate

If the client needs to get object B from a field/method of object A, to call a method of object B: Create a new method in class A that delegate the call to object B.

// Before



// After



The problem that this refactoring will solve is that when a client gets object B from a field/method of object A, just to operate the method of object B. This will increase the coupling between the client and both class A and B. If one of the method definitions changes, client code will also need to be modified.

To solve it, we can make a method that delegates the call to object B. In the example shown by the slide, instead of executing `person.getDepartment().getManager()`, we only need to do `person.getManager()` in client code. The `getManager()` method will call the `getManager()` method of a `Department` object that is saved in the `department` field inside of `Person` object. The `Person` class acts as a middleman.

However, do not use this refactoring step too aggressively as it will result in too much middleman. This will cause code in which it's hard to see where the functionality is occurring, which in turn makes debugging more difficult.

Code Smells - Couplers

- **Middle Man**
  A class that only delegates a work to another class, without doing anything meaningful.

  Possible refactoring: **Remove Middle Man** (the opposite of Hide Delegate), Inline Method, Replace Delegation with Inheritance

This code smell is the opposite of **Message Chain** code smell. This code smell signifies that there is a class that only delegates work to other classes, without doing anything meaningful besides being just a middleman. We need to balance between delegation hiding and using delegates. To fix this problem, we can just do the opposite of Hide Delegate: **Remove Middle Man**. Then, use either Inline Method or Replace Delegation with Inheritance to clean up the code after refactoring.

How to Choose Refactoring Methods?

- Check again the design principles (**SOLID**) when choosing refactoring step and after done refactoring.
- For refactoring steps that involves **moving or extracting features, fields, or methods**, make sure no ambiguity that shows after refactoring. Also make sure that the code isn't becoming too tightly coupled on each other.
- **Involve peer reviewing** to get second opinion.
- **Don't overthink**. Don't over-refactoring, don't over-engineering.
- Consider **performance trade-offs**. **Avoid security trade-offs**.

For every kind of code smells, there are different refactoring methods that are possible to use. However, each refactoring method has its own use cases. Learn thoroughly, consider its trade-offs between benefits and drawbacks. Make sure that our code still adheres to **S.O.L.I.D.** principles as good as possible. Again, we can't make perfect code, and programs evolve over time, so don't overthink.

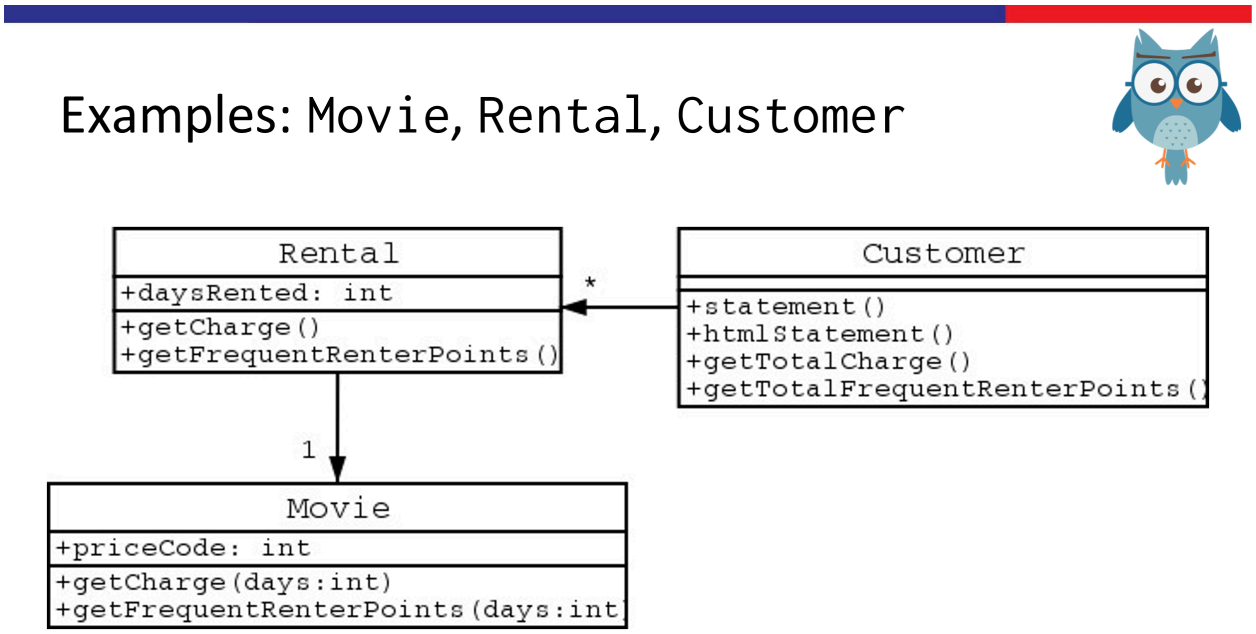
Overthinking when refactoring often results in worse code quality, as shown in some code smells such as Message Chains and Middle Man. Over-refactoring on one aspect will result in new code smells in another aspect. Sometimes, we don't know that we are overthinking. So, we need second opinions, and that's where peer review is important.

Refactoring often prioritises code readability and maintainability. Sometimes, refactoring will also affect performance and security aspects. Consider performance trade-offs, check your needs, whether you need more performance or more maintainability. However, regarding security, we need to prioritise security as our utmost priority. If there is a refactoring step that has security drawbacks, you need to avoid that and pick another refactoring alternative. You can utilise tools in your IDE or code security testing such as Static Analysis Software Testing (SAST) to help you determine security drawbacks.

6. Refactoring Example

This submodule will explain how to refactor after we identify code smells that exist in our code using an example. By learning this module, students should be able to fix design flaws by using refactoring steps.

Code Example: Movie Rental



Software design understanding is a must-have when we refactor our code. So, to illustrate a refactoring process, let's begin with an example. Suppose we are building a Movie Rental app that consists of three classes:

1. **`Movie`** is a data class that represents a movie. This class has two methods:
    a. **`getCharge(days:int)`** to check the rent price for that movie, and
    b. **`getFrequentRenterPoints(days:int)`** to check the "frequent renter" point a customer can get after renting that movie.
2. **`Customer`** represents the customer of the movie rental. This class has four methods:
    a. **`statement()`** generates a rent statement (we often call it "invoice") for that customer.
    b. **`htmlStatement()`** generates a human-friendly rent statement (we often call it "invoice") for that customer, using HTML.
    c. **`getTotalCharge()`** returns the total amount of money that a customer must pay to the movie rental.

d. **`getTotalFrequentRenterPoints()`** returns total "frequent renter" points that a customer will earn after finishing the rent transaction.

3. **`Rental`** represents a customer renting a movie. This class has two methods:
   a. **`getCharge()`** returns the amount of money to rent a movie for `daysRented` days.
   b. **`getFrequentRenterPoints()`** returns "frequent renter" points that a customer will earn after renting a movie for `daysRented` days.

NOTE: For more information about the Movie Rental example, here is the live code: https://gitlab.com/ichlaffterlalu/advprog-refactoring-example. This code is not exactly following the class diagram on the previous page, but the class diagram is the end goal.

Discussion: `Movie`, `Rental`, `Customer`

- What is your impressions about the design of this program, particularly on `statement()` in `Customer`?
- The program works, but ...
- Compiler doesn't care whether your code is ugly or clean, but (human) programmer does!

The One Constant in Software Development

Before you move up to the next page, let's think about what your impression regarding the design of the Movie Rental program is. There is one constant thing in software development: **CHANGE**. Now, let's imagine we want to change some rules in our movie rental. What kind of changes can you imagine?

After you know what changes are possible, now think about how current class design will embrace or prohibit those changes. For example, the `statement()` method in Customer, what will happen if we change the rent charging rules? How much will the code change? If you want to ask someone to apply that change for you, how easy will he/she understand your code? Remember, a compiler does not care about ugly code, but a human programmer does!

Change: HTML `Statement`

- Is it possible to reuse existing code to implement new feature?
  - How about duplicate most of behaviour from `statement()` and put it in `htmlStatement()`?
- What happens when the charging rules change?

Change: Movie Classification

- The users want to make changes to the way they classify movies, but they haven't yet decided on the change they are going to make
- Will affect both the way renters are charged for movies and the way the frequent renters points are calculated

These slides explain some of the possibilities of how our Movie Rental system can change. What happens if our Movie Rental expands its service, allowing users not only to rent, but also to buy the physical disc. Some parts of the invoice will be the same between "buying" and "renting", right? Also, do we need to change every `statement()` and `htmlStatement()` method if the template changes, or if the charging rules change?

Let's also consider movie classification. Previously in the design, we did not consider the classification of movies. A movie is a movie. But how about short movies? If we want to classify movies based on duration or based on how popular it is, we might want to change the way we calculate rent price and "frequent renter" points for each movie category. Of course, we need to update the software design to enable movie classifications.

Refactor!

- Before that: **provide tests!**
- Tests are essential to ensure correctness of the code before and after being refactored

Every change we make will incorporate a refactoring process. Refactoring is the essential part of adopting changes to our software. But before refactoring and adding new functionalities, remember to **provide tests** first. We want to make sure that previously available features do not break after we add those new features. Also, we want to make sure that behaviours of previously available functionalities do not change after we refactor them.

Discussion: `statement()`

- What could go wrong with a long method?
  - Hint: is it cohesive?
- Solution: decompose the method into smaller pieces
- Remember the end goal: `htmlStatement()` with less duplication of code
- Use this refactoring step: **Extract Method**

Look at this `statement()` code in `Customer` below:

```
01    public String statement() {
02        double totalAmount = 0;
03        int frequentRenterPoints = 0;
04        Enumeration<Rental> rentals = this.rentals.elements();
05        String result = "Rental Record for " + getName() + "\n";
06        while (rentals.hasMoreElements()) {
07            double thisAmount = 0;
08            Rental each = rentals.nextElement();
09
10            // Determine amounts for each line
11            switch (each.getMovie().getPriceCode()) {
12                case Movie.REGULAR:
13                    thisAmount += 2;
14                    if (each.getDaysRented() > 2) {
15                        thisAmount += (each.getDaysRented() - 2) * 1.5;
16                    }
17                    break;
18                case Movie.NEW_RELEASE:
19                    thisAmount += each.getDaysRented() * 3;
20                    break;
21                case Movie.CHILDREN:
22                    thisAmount += 1.5;
23                    if (each.getDaysRented() > 3) {
24                        thisAmount += (each.getDaysRented() - 3) * 1.5;
25                    }
26                    break;
27            }
28
29            // Add frequent renter points
30            frequentRenterPoints++;
31
32            // Add bonus for a two day new release rental
33            if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE)
34                && each.getDaysRented() > 1) {
35                frequentRenterPoints++;
36            }
37
38            // Show figures for this rental
```

```
39              result += "\t" + each.getMovie().getTitle() + "\t"
40                  + String.valueOf(thisAmount) + "\n";
41              totalAmount += thisAmount;
42          }
43
44          // Add footer lines
45          result += "Amount owed is " + String.valueOf(totalAmount) + "\n";
46          result += "You earned " + String.valueOf(frequentRenterPoints)
47              + " frequent renter points";
48
49          return result;
50      }
```

The code for this method is very long (50 lines), it even spans into two pages. You can also see that it is divided into several sections (or steps) using comments. This code is also not cohesive, there are calculations about the rent price amount, where `statement()` should only do printing statements. The end goal is that we want to make a HTML variant of the statement (called `htmlStatement()`) with less duplications.

Refactoring: Extract Method

- The switch-case is a good candidate to be separated to its own method
  - Anything else?
- See livecode

To fix this code, we will use a refactoring step called the **Extract Method**. We will of course move each step to its new methods, such as `getTotalAmount()` and `getTotalFrequentRenterPoints()`. This can be achieved by moving the `while` loop along with some of its contents to each method.

- `getTotalAmount()` has the "`Determine amounts for each line`" step and line 02 temporary variable,
- `getTotalFrequentRenterPoints()` has the "`Add frequent renter points`" and "`Add bonus for a two day new release rental`" steps and line 03 temporary variable,
- and then, in the `statement()` method, the while loop now only contains the "`Show figures for this rental`" part as it is relevant to the responsibility, which is printing a statement that shows figures for each rented movie.

We might also need to change the statement footer so it will call `getTotalAmount()` and `getTotalFrequentRenterPoints()` and use their return values. This way, we can make new `htmlStatement()` without duplication, as printing header, figures for each rented movie, and footer will be different between ordinary text statements and HTML statements.

Refactoring Step: Move Method

Discussion: `amountFor()`

- Should a `Customer` compute the charge amount?
  - Hint: which class should have been responsible in computing charge amount?
- In most cases, a method should be on the object whose data it uses
- Solution: Move Method to `Rental`
- Use this refactoring step: **Move Method**

And then after refactoring we realised that we have two new irrelevant methods in `Customer`: `getTotalAmount()` and `getTotalFrequentRenterPoints()`. It's irrelevant because a customer should not calculate those. The rental's cashier should be authorised to calculate total amount and total frequent renter points for a customer. So, we need to move those methods to `Rental` using the **Move Method** technique.

Refactoring: Move Method

- Move charge amount calculation into `Rental`
- Similarly, move frequent renter point calculation into `Rental` as well
  - We have to do Extract Method on frequent renter point calculation before we can move it to `Rental`
- See livecode

We need to extract the methods again before moving those methods, as we can't move `getTotalAmount()` and `getTotalFrequentRenterPoints()` straight away to `Rental`. If we just move those methods, there will be problems because they need a list of `Rental` objects.

- Extract the individual rent amount calculation in `getTotalAmount()` to `getAmount()`. In `getTotalAmount()`, call `each.getAmount()` to get the amount for each rented movie.
- Extract the individual frequent renter points in `getTotalFrequentRenterPoints()` to `getFrequentRenterPoints()`. In `getTotalFrequentRenterPoints()`, call `each.getFrequentRenterPoints()` to get the renter points for each rented movie.
- Then move `getAmount()` and `getFrequentRenterPoints()` to `Rental`.

Refactoring Step: Replace Temp with Query

Discussion: Temporary Variables

- Identify temporary variables used in `statement()`!
- Is it wrong to use them?
- Remember the end goal: `htmlStatement()` with less code duplication
  → maximise code reuse
- Use this refactoring step: **Replace Temp with Query**

Temporary variables can be a problem. They are only useful within their own method; therefore, they will encourage long and complex methods. It will also increase duplication and make changing ways to obtain those values difficult. For example, in line 04 in the previous `statement()` there is a temporary variable that gets an enumerator object from the list of `Rental` objects (`this.rentals`). What will happen if we want to change the source, not using `this.rentals`, but something else? Because of duplication, we need to change both temporary variables in `statement()` and `htmlStatement()`. That's why we need to replace those variables into a call to query methods.

Refactoring: Replace Temp with Query

- Temporary variables in `statement()`:
  - `totalAmount` → `getTotalAmount()`.
  - `frequentRentalPoints` → `getFrequentRenterPoints()`.
  - `renters` → `getRenters()`.

We will use **Replace Temp with Query** to fix this temporary variable problem. Create a new method to get the values of the temporary variable. Unfortunately, this will get you to some performance caveats (you will need to calculate things every time you call the query). You might still want to use temporary variables (**note:** don't make new instance variables) to avoid calculating those things multiple times in a single `statement()` call.

Self-Reflection B: Possible Further Refactoring

Possible Further Refactoring

- The switch-case block in `getCharge()`
- Frequent renter point calculation
- Introduce state pattern in computing price
  - Three refactoring: Replace Type Code with State/Strategy, Move Method, Replace Conditional with Polymorphism

Those are the possible further refactoring that you can do based on the live code. Is there any other code smells you can spot and what are your proposed refactoring steps to solve those code smell problems? Here we will give you some space to think:

The Next Question

- You can do refactoring
  - Modern IDEs support refactoring automation
- The next question is.. when?

Modern IDEs support refactoring automation. There are a lot of options in IntelliJ IDEA to help you extract or move things in or out of a class. Explore that by yourself. Also explore when you will need those features. Take notes here:

7. Tutorial & Exercise

We will use Spring Boot, a popular Java web framework, as an exercise. In this tutorial and exercise, there are several to-do checklists that you should achieve. The checklists are marked with **red font colour**.

Preparation

Before starting the development process, please install the following environments:

1. **Java 21**. If you have installed Java, please check the installed version before continuing. You can check the version by running java -version in Command Prompt/Terminal. Make sure the version is 21.X.X (for example, 21.0.2).
2. **Git**
3. IDE **IntelliJ** Ultimate
4. Make sure that you have at least done Tutorial on Week 1 as the dependencies of unit tests and functional tests are defined there.

Tutorial

Tutorial Preparation

1. Open your Module 3 Exercise project using IntelliJ or your favourite editor.
2. Create a new branch named **`order`** based on the latest version of your `main`/`master` branch.

[RED] Make tests for Order model

Before we make tests, we need to understand the requirements first. Order model consists of several fields:

- `id`: String (UUID)
- `products`: List<Product> (not null)
- `orderTime`: Long (UNIX timestamp)
- `author`: String (not null)
- `status`: String (Enum of “WAITING_PAYMENT”, “FAILED”, “CANCELLED”, “SUCCESS”)

The default value of status is “WAITING_PAYMENT”. When setting `status`, we must check that the parameter is only one of 4 strings in the Enum. Else, we reject the change by keeping previous `status`. We must also reject orders with empty `products`.

So, we need to build at least these tests:

- Unhappy: Test to create the order with empty Product.
- Happy: Test to create the order with no status defined.
- Happy: Test to create the order status of “SUCCESS”.

- Unhappy: Test to create the order with invalid status.
- Happy: Test to edit the order with one of correct status.
- Unhappy: Test to edit the order with invalid status.

Do these steps:

1. **Checkout the branch `order`.**
2. **Create `OrderTest` test suite class in `src/test/java/model`.**
3. Create `setUp` for the `OrderTest`.

```
class OrderTest {
    11 usages
    private List<Product> products;
    @BeforeEach
    void setUp() {
        this.products = new ArrayList<>();
        Product product1 = new Product();
        product1.setProductId("eb558e9f-1c39-460e-8860-71af6af63bd6");
        product1.setProductName("Sampo Cap Bambang");
        product1.setProductQuantity(2);
        Product product2 = new Product();
        product2.setProductId("a2c62328-4a37-4664-83c7-f32db8620155");
        product2.setProductName("Sabun Cap Usep");
        product2.setProductQuantity(1);
        this.products.add(product1);
        this.products.add(product2);
    }
```

4. Create unhappy path test: Test to create the order with empty `products`.

```
    @Test
    void testCreateOrderEmptyProduct() {
        this.products.clear();

        assertThrows(IllegalArgumentException.class, () -> {
            Order order = new Order( id: "13652556-012a-4c07-b546-54eb1396d79b",
                this.products, orderTime: 1708560000L, author: "Safira Sudrajat");
        });
    }
```

5. Create a happy path test: Test to create the order with no status defined.

```
@Test
void testCreateOrderDefaultStatus() {
    Order order = new Order( id: "13652556-012a-4c07-b546-54eb1396d79b",
        this.products, orderTime: 1708560000L, author: "Safira Sudrajat");

    assertSame(this.products, order.getProducts());
    assertEquals( expected: 2, order.getProducts().size());
    assertEquals( expected: "Sampo Cap Bambang", order.getProducts().get(0).getProductName());
    assertEquals( expected: "Sabun Cap Usep", order.getProducts().get(1).getProductName());

    assertEquals( expected: "13652556-012a-4c07-b546-54eb1396d79b", order.getId());
    assertEquals( expected: 1708560000L, order.getOrderTime());
    assertEquals( expected: "Safira Sudrajat", order.getAuthor());
    assertEquals( expected: "WAITING_PAYMENT", order.getStatus());
}
```

6. Create a happy path test: Test to create the order status of “SUCCESS”.

```
@Test
void testCreateOrderSuccessStatus() {
    Order order = new Order( id: "13652556-012a-4c07-b546-54eb1396d79b",
        this.products, orderTime: 1708560000L, author: "Safira Sudrajat", status: "SUCCESS");
    assertEquals( expected: "SUCCESS", order.getStatus());
}
```

7. Create an unhappy path test: Test to create the order with invalid status.

```
@Test
void testCreateOrderInvalidStatus() {
    assertThrows(IllegalArgumentException.class, () -> {
        Order order = new Order( id: "13652556-012a-4c07-b546-54eb1396d79b",
            this.products, orderTime: 1708560000L, author: "Safira Sudrajat", status: "MEOW");
    });
}
```

8. Create a happy path test: Test to edit the order with one of correct status.

```
@Test
void testSetStatusToCancelled() {
    Order order = new Order( id: "13652556-012a-4c07-b546-54eb1396d79b",
        this.products, orderTime: 1708560000L, author: "Safira Sudrajat");
    order.setStatus("CANCELLED");
    assertEquals( expected: "CANCELLED", order.getStatus());
}
```

9. Create an unhappy path test: Test to edit the order with invalid status.

```
    @Test
    void testSetStatusToInvalidStatus() {
        Order order = new Order( id: "13652556-012a-4c07-b546-54eb1396d79b",
            this.products, orderTime: 1708560000L, author: "Safira Sudrajat");
        assertThrows(IllegalArgumentException.class, () -> order.setStatus("MEOW"));
    }
}
```

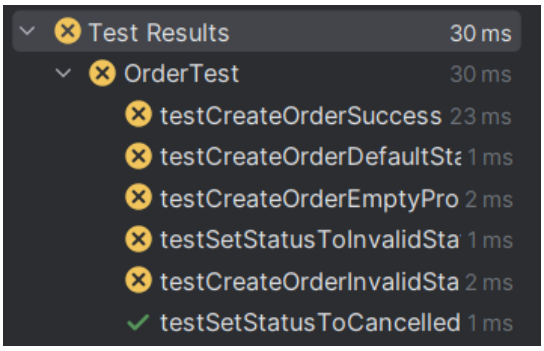
10. **Commit the changes to Git with this message [RED] Add tests for Order model.**
11. Create `Order` class in `src/main/java/model` for the skeleton to avoid compile errors.

```
@Builder
@Getter
public class Order {
    String id;
    List<Product> products;
    Long orderTime;
    String author;
    @Setter
    String status;

    4 usages
    public Order(String id, List<Product> products, Long orderTime, String author) {
    }

    2 usages
    public Order(String id, List<Product> products, Long orderTime, String author, String status) {
    }
}
```

12. **Run the test. You can see almost all the tests are failed (except Happy path test on `setStatus`).**



13. **Commit the changes to Git with this message [RED] Add Order model skeleton.**
14. **Push the changes to branch `order`.**

[GREEN] Implement `Order` model

1. **Go to `Order` model class in `src/main/java/model`.**
2. Implement the first constructor (the one that has no `status` parameter).

```
public Order(String id, List<Product> products, Long orderTime, String author) {
    this.id = id;
    this.orderTime = orderTime;
    this.author = author;
    this.status = "WAITING_PAYMENT";

    if (products.isEmpty()) {
        throw new IllegalArgumentException();
    } else {
        this.products = products;
    }
}
```

3. **Run the test, some of the tests should be passed.**
4. Implement the second constructor (the one that has `status` parameter).

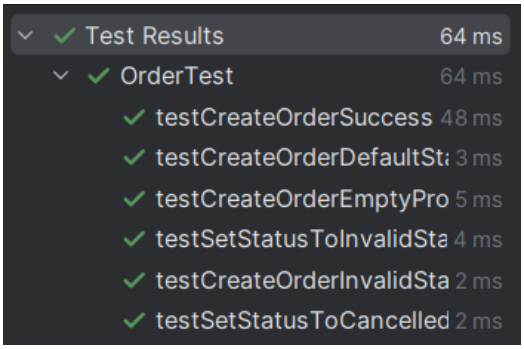
```
public Order(String id, List<Product> products, Long orderTime, String author, String status) {
    this(id, products, orderTime, author);

    String[] statusList = {"WAITING_PAYMENT", "FAILED", "SUCCESS", "CANCELLED"};
    if (Arrays.stream(statusList).noneMatch(item -> (item.equals(status)))) {
        throw new IllegalArgumentException();
    } else {
        this.status = status;
    }
}
```

5. **Run the test again, some of the tests should be passed.**
6. Delete the `@Setter` annotation in status variable, as we will implement our own `setStatus`.
7. Implement the `setStatus` method.

```
public void setStatus(String status) {
    String[] statusList = {"WAITING_PAYMENT", "FAILED", "SUCCESS", "CANCELLED"};
    if (Arrays.stream(statusList).noneMatch(item -> (item.equals(status)))) {
        throw new IllegalArgumentException();
    } else {
        this.status = status;
    }
}
```

8. **Run the test again, all the tests should be passed.**



9. **Commit the changes to Git with this message [GREEN] Implement Order model.**
10. **Push the changes to branch `order`.**

[REFACTOR] Implement OrderStatus enum

After implementing Order, we see too many hardcoded status strings scattered everywhere. If someday we want to change the language of this eShop, this condition will make the change difficult. So, we need to do a refactor step: **Replace Type Code with Class**. In this case, we will be using Java's Enum class type to list all possibilities of `Order` status in one place. Do these steps:

1. **Create `OrderStatus` enum in `src/main/java/enums`.** The enum also has `contains(String param)` static method that will be useful to validate strings.

```
@Getter
public enum OrderStatus {
    WAITING_PAYMENT( value: "WAITING_PAYMENT"),
    FAILED( value: "FAILED"),
    SUCCESS( value: "SUCCESS"),
    CANCELLED( value: "CANCELLED");

    private final String value;
    8 usages
    private OrderStatus(String value) {
        this.value = value;
    }

    1 usage
    public static boolean contains(String param) {
        for (OrderStatus orderStatus : OrderStatus.values()) {
            if (orderStatus.name().equals(param)) {
                return true;
            }
        }
        return false;
    }
}
```

2. **Commit the changes to Git with this message [REFACTOR] Add OrderStatus enum.**
3. Modify `Order` model class in `src/main/java/model`. We will refactor the `status` check in the constructor and `setStatus` method to use `OrderStatus.contains()`.

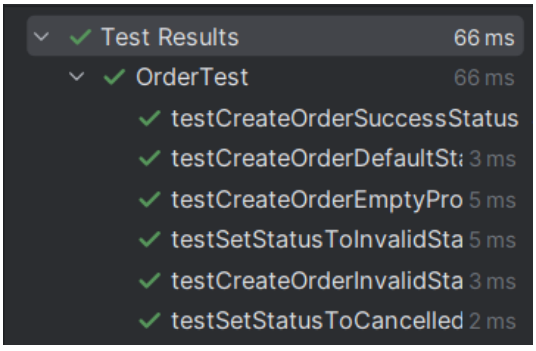
```
public Order(String id, List<Product> products, Long orderTime, String author) {
    this.id = id;
    this.orderTime = orderTime;
    this.author = author;
    this.status = OrderStatus.WAITING_PAYMENT.getValue();

    if (products.isEmpty()) {
        throw new IllegalArgumentException();
    } else {
        this.products = products;
    }
}

3 usages
public Order(String id, List<Product> products, Long orderTime, String author, String status) {
    this(id, products, orderTime, author);
    this.setStatus(status);
}

3 usages
public void setStatus(String status) {
    if (OrderStatus.contains(status)) {
        this.status = status;
    } else {
        throw new IllegalArgumentException();
    }
}
```

4. **Rerun the tests. Make sure the tests are still all passed.**



5. **Commit the changes to Git with this message [REFACTOR] Apply OrderStatus enum check to Order model.**

6. To make the refactor complete (this is optional, though), modify any hardcoded status string in `OrderTest` (the unit test for `Order` model) to `OrderStatus`. For example:

```
@Test
void testCreateOrderSuccessStatus() {
    Order order = new Order( id: "13652556-012a-4c07-b546-54eb1396d79b",
        this.products, orderTime: 1708560000L, author: "Safira Sudrajat",
        OrderStatus.SUCCESS.getValue());
    assertEquals(OrderStatus.SUCCESS.getValue(), order.getStatus());
}
```

7. **Rerun the tests again after modifying them. Make sure the tests are still all passed.**
8. **Commit the changes to Git with this message [REFACTOR] Apply OrderStatus enum to Order model tests.**
9. **Push the changes to branch `order`.**

[RED] Make tests for Order repository

Before we make tests, we need to understand the requirements first. For `OrderRepository`, we need to implement `save(Order order)` for create/update, `findById(String id)` to find order by order ID, and `findAllByAuthor(String author)` to find orders by author name. Author is case sensitive. To update an Order, we must find the Order object position in the list first.

So, we need to build at least these tests:

- Happy: Use `save` to add a new Order.
- Happy: Use `save` to update an Order.
- Happy: Use `findById` to find Order using a valid ID.
- Unhappy: Use `findById` to find Order using an ID that is never used.
- Happy: Use `findAllByAuthor` to find Order using a valid author name.
- Unhappy: Use `findAllByAuthor` to find Order using an all-lowercase name (this is to check case sensitivity of author).
  We did not test `findAllByAuthor` with a name that is never used or an empty string because this case covers the whole negative case for this method.

Do these steps:

1. **Create the `OrderRepositoryTest` test suite class in `src/test/java/repository`.**

2. Create setUp for the `OrderRepositoryTest`.

```
class OrderRepositoryTest {
    14 usages
    OrderRepository orderRepository;
    19 usages
    List<Order> orders;

    @BeforeEach
    void setUp() {
        orderRepository = new OrderRepository();

        List<Product> products = new ArrayList<>();
        Product product1 = new Product();
        product1.setProductId("eb558e9f-1c39-460e-8860-71af6af63bd6");
        product1.setProductName("Sampo Cap Bambang");
        product1.setProductQuantity(2);
        products.add(product1);

        orders = new ArrayList<>();
        Order order1 = new Order( id: "13652556-012a-4c07-b546-54eb1396d79b",
            products, orderTime: 1708560000L, author: "Safira Sudrajat");
        orders.add(order1);
        Order order2 = new Order( id: "7f9e15bb-4b15-42f4-aebc-c3af385fb078",
            products, orderTime: 1708570000L, author: "Safira Sudrajat");
        orders.add(order2);
        Order order3 = new Order( id: "e334ef40-9eff-4da8-9487-8ee697ecbf1e",
            products, orderTime: 1708570000L, author: "Bambang Sudrajat");
        orders.add(order3);
    }
```

3. Create a happy path test: Use save to add new Order.

```
    @Test
    void testSaveCreate() {
        Order order = orders.get(1);
        Order result = orderRepository.save(order);

        Order findResult = orderRepository.findById(orders.get(1).getId());
        assertEquals(order.getId(), result.getId());
        assertEquals(order.getId(), findResult.getId());
        assertEquals(order.getOrderTime(), findResult.getOrderTime());
        assertEquals(order.getAuthor(), findResult.getAuthor());
        assertEquals(order.getStatus(), findResult.getStatus());
    }
```

4. Create a happy path test: Use save to update an Order.

```
@Test
void testSaveUpdate() {
    Order order = orders.get(1);
    orderRepository.save(order);
    Order newOrder = new Order(order.getId(), order.getProducts(), order.getOrderTime(),
        order.getAuthor(), OrderStatus.SUCCESS.getValue());
    Order result = orderRepository.save(newOrder);

    Order findResult = orderRepository.findById(orders.get(1).getId());
    assertEquals(order.getId(), result.getId());
    assertEquals(order.getId(), findResult.getId());
    assertEquals(order.getOrderTime(), findResult.getOrderTime());
    assertEquals(order.getAuthor(), findResult.getAuthor());
    assertEquals(OrderStatus.SUCCESS.getValue(), findResult.getStatus());
}
```

5. Create a happy path test: Use `findById` to find Order using a valid ID.

```
@Test
void testFindByIdIfIdFound() {
    for (Order order : orders) {
        orderRepository.save(order);
    }

    Order findResult = orderRepository.findById(orders.get(1).getId());
    assertEquals(orders.get(1).getId(), findResult.getId());
    assertEquals(orders.get(1).getOrderTime(), findResult.getOrderTime());
    assertEquals(orders.get(1).getAuthor(), findResult.getAuthor());
    assertEquals(orders.get(1).getStatus(), findResult.getStatus());
}
```

6. Create an unhappy path test: Use `findById` to find Order using an ID that is never used.

```
@Test
void testFindByIdIfIdNotFound() {
    for (Order order : orders) {
        orderRepository.save(order);
    }

    Order findResult = orderRepository.findById("zczc");
    assertNull(findResult);
}
```

7. Create a happy path test: Use `findAllByAuthor` to find Order using a valid author name.

```
    @Test
    void testFindAllByAuthorIfAuthorCorrect() {
        for (Order order : orders) {
            orderRepository.save(order);
        }

        List<Order> orderList = orderRepository.findAllByAuthor(
            orders.get(1).getAuthor());
        assertEquals( expected: 2, orderList.size());
    }
```

8. Create an unhappy path test: Use `findAllByAuthor` to find Order using an all-lowercase name.

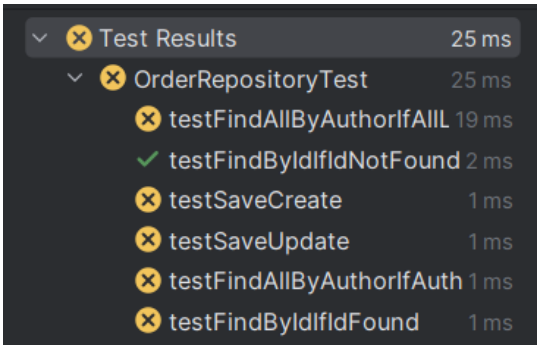
```
    @Test
    void testFindAllByAuthorIfAllLowercase() {
        orderRepository.save(orders.get(1));

        List<Order> orderList = orderRepository.findAllByAuthor(
            orders.get(1).getAuthor().toLowerCase());
        assertTrue(orderList.isEmpty());
    }
}
```

9. **Commit changes to Git with this message: [RED] Add tests for OrderRepository.**
10. Create `OrderRepository` class in `src/main/java/repository` for the skeleton to avoid compile errors.

```
@Repository
public class OrderRepository {
    no usages
    private List<Order> orderData = new ArrayList<>();
    16 usages
    public Order save(Order order) {return null;}
    13 usages
    public Order findById(String id) {return null;}
    5 usages
    public List<Order> findAllByAuthor(String author) {return null;}
}
```

11. **Run the test. You can see all the tests are failed (except `findById` with invalid ID test).**



12. **Commit the changes to Git with this message [RED] Add OrderRepository skeleton.**
13. **Push the changes to branch `order`.**

[GREEN] Implement Order repository

1. **Go to `OrderRepository` class in `src/main/java/repository`.**
2. Implement save method. Remember to check if an order with the same ID exists or not. If it exists, update. If it does not exist, save the Order object.

```
@Repository
public class OrderRepository {
    6 usages
    private List<Order> orderData = new ArrayList<>();

    7 usages
    public Order save(Order order) {
        int i = 0;
        for (Order savedOrder : orderData) {
            if (savedOrder.getId().equals(order.getId())) {
                orderData.remove(i);
                orderData.add(i, order);
                return order;
            }
            i += 1;
        }

        orderData.add(order);
        return order;
    }
```

3. **Run the test, some of the tests should be passed.**

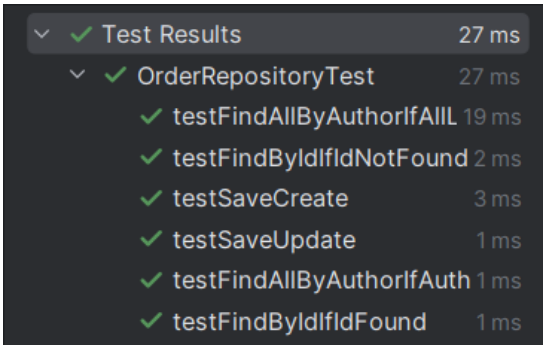
4. Implement `findById` method.

```
public Order findById(String id) {
    for (Order savedOrder : orderData) {
        if (savedOrder.getId().equals(id)) {
            return savedOrder;
        }
    }
    return null;
}
```

5. **Run the test again, some of the tests should be passed.**
6. Implement `findAllByAuthor` method.

```
public List<Order> findAllByAuthor(String author) {
    List<Order> result = new ArrayList<>();
    for (Order savedOrder : orderData) {
        if (savedOrder.getAuthor().equals(author)) {
            result.add(savedOrder);
        }
    }
    return result;
}
```

7. **Run the test, all the tests should be passed.**



8. **Commit the changes to Git with this message [GREEN] Implement OrderRepository class.**
9. **Push the changes to branch `order`.**

[RED] Make unit tests for Order service

Before we make tests, we need to understand the requirements first. For `OrderService`, we need to implement `createOrder(Order order)` to create new Order, `updateStatus(String orderId, String status)` to create new Order by update status, `findById(String orderId)` to find orders by ID, and `findAllByAuthor(String author)` to find orders by author name. Author is case sensitive. To update an Order, we must find it first by using the ID. If the Order is not found, raise `NoSuchElementException`.

So, we need to build at least these tests:

- Happy: Use `createOrder` to add a new Order.
- Unhappy: Use `createOrder` to add an already existing Order.
  The `createOrder` tests will make use of Mockito's mock verification.
- Happy: Use `updateStatus` to update status of an Order.
- Unhappy: Use `updateStatus` to update status of an Order using invalid status string.
- Unhappy: Use `updateStatus` with an ID that is never used.
- Happy: Use `findById` to find Order using a valid ID.
- Unhappy: Use `findById` to find Order using an ID that is never used.
- Happy: Use `findAllByAuthor` to find Order using a valid author name.
- Unhappy: Use `findAllByAuthor` to find Order using an all-lowercase name.

We will use the **mock** feature from Mockito, to mock `OrderRepository`. This is useful because we might want to migrate from using lists to a proper database someday.

Do these steps:

1. **Create the `OrderServiceImplTest` test suite class in `src/test/java/service`.**
2. Create `setUp` for the `OrderServiceImplTest`.

```
@ExtendWith(MockitoExtension.class)
class OrderServiceTest {
    9 usages
    @InjectMocks
    OrderServiceImpl orderService;
    15 usages
    @Mock
    OrderRepository orderRepository;
    11 usages
    List<Order> orders;

    @BeforeEach
    void setUp() {
        List<Product> products = new ArrayList<>();
        Product product1 = new Product();
        product1.setProductId("eb558e9f-1c39-460e-8860-71af6af63bd6");
        product1.setProductName("Sampo Cap Bambang");
        product1.setProductQuantity(2);
        products.add(product1);

        orders = new ArrayList<>();
        Order order1 = new Order( id: "13652556-012a-4c07-b546-54eb1396d79b",
            products, orderTime: 1708560000L, author: "Safira Sudrajat");
        orders.add(order1);
        Order order2 = new Order( id: "7f9e15bb-4b15-42f4-aebc-c3af385fb078",
            products, orderTime: 1708570000L, author: "Safira Sudrajat");
        orders.add(order2);
    }
```

3. Create a happy path test: Use `createOrder` to add a new Order.

```
@Test
void testCreateOrder() {
    Order order = orders.get(1);
    doReturn(order).when(orderRepository).save(order);

    Order result = orderService.createOrder(order);
    verify(orderRepository, times( wantedNumberOfInvocations: 1)).save(order);
    assertEquals(order.getId(), result.getId());
}
```

4. Create unhappy path test: Use `createOrder` to add an already existing Order.

```
@Test
void testCreateOrderIfAlreadyExists() {
    Order order = orders.get(1);
    doReturn(order).when(orderRepository).findById(order.getId());

    assertNull(orderService.createOrder(order));
    verify(orderRepository, times( wantedNumberOfInvocations: 0)).save(order);
}
```

5. Create a happy path test: Use `updateStatus` to update status of an Order.

```
@Test
void testUpdateStatus() {
    Order order = orders.get(1);
    Order newOrder = new Order(order.getId(), order.getProducts(), order.getOrderTime(),
        order.getAuthor(), OrderStatus.SUCCESS.getValue());
    doReturn(order).when(orderRepository).findById(order.getId());
    doReturn(newOrder).when(orderRepository).save(any(Order.class));

    Order result = orderService.updateStatus(order.getId(), OrderStatus.SUCCESS.getValue());

    assertEquals(order.getId(), result.getId());
    assertEquals(OrderStatus.SUCCESS.getValue(), result.getStatus());
    verify(orderRepository, times( wantedNumberOfInvocations: 1)).save(any(Order.class));
}
```

6. Create unhappy path test: Use `updateStatus` to update status of an Order using invalid status string.

```
@Test
void testUpdateStatusInvalidStatus() {
    Order order = orders.get(1);
    doReturn(order).when(orderRepository).findById(order.getId());

    assertThrows(IllegalArgumentException.class,
        () -> orderService.updateStatus(order.getId(),  status: "MEOW"));

    verify(orderRepository, times( wantedNumberOfInvocations: 0)).save(any(Order.class));
}
```

7. Create an unhappy path test: Use `updateStatus` with an ID that is never used.

```
@Test
void testUpdateStatusInvalidOrderId() {
    doReturn( toBeReturned: null).when(orderRepository).findById("zczc");

    assertThrows(NoSuchElementException.class,
        () -> orderService.updateStatus( orderId: "zczc", OrderStatus.SUCCESS.getValue()));

    verify(orderRepository, times( wantedNumberOfInvocations: 0)).save(any(Order.class));
}
```

8. Create a happy path test: Use `findById` to find Order using a valid ID.

```
@Test
void testFindByIdIfIdFound() {
    Order order = orders.get(1);
    doReturn(order).when(orderRepository).findById(order.getId());

    Order result = orderService.findById(order.getId());
    assertEquals(order.getId(), result.getId());
}
```

9. Create an unhappy path test: Use `findById` to find Order using an ID that is never used.

```
@Test
void testFindByIdIfIdNotFound() {
    doReturn( toBeReturned: null).when(orderRepository).findById("zczc");
    assertNull(orderService.findById( orderId: "zczc"));
}
```

10. Create a happy path test: Use `findAllByAuthor` to find Order using a valid author name.

```
@Test
void testFindAllByAuthorIfAuthorCorrect() {
    Order order = orders.get(1);
    doReturn(orders).when(orderRepository).findAllByAuthor(order.getAuthor());

    List<Order> results = orderService.findAllByAuthor(order.getAuthor());
    for (Order result : results) {
        assertEquals(order.getAuthor(), result.getAuthor());
    }
    assertEquals( expected: 2, results.size());
}
```

11. Create an unhappy path test: Use `findAllByAuthor` to find Order using an all-lowercase name.

```
    @Test
    void testFindAllByAuthorIfAllLowercase() {
        Order order = orders.get(1);
        doReturn(new ArrayList<Order>()).when(orderRepository)
            .findAllByAuthor(order.getAuthor().toLowerCase());

        List<Order> results = orderService.findAllByAuthor(
            order.getAuthor().toLowerCase());
        assertTrue(results.isEmpty());
    }
}
```

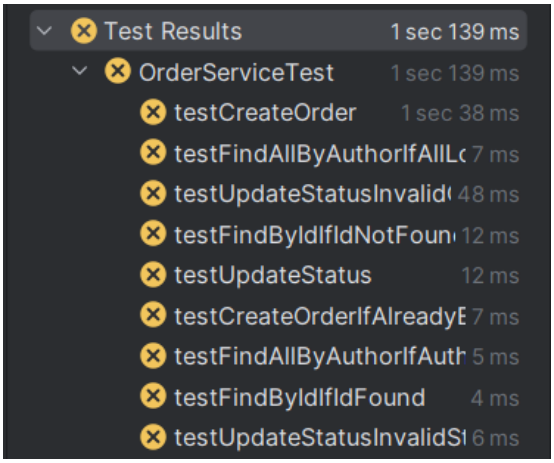
12. **Commit changes to Git with this message: [RED] Add tests for OrderServiceImpl.**
13. Create `OrderService` interface in `src/main/java/service` for the skeleton to avoid compile errors.

```
public interface OrderService {
    2 usages  1 implementation
    public Order createOrder(Order order);
    3 usages  1 implementation
    public Order updateStatus(String orderId, String status);
    2 usages  1 implementation
    public Order findById(String orderId);
    2 usages  1 implementation
    public List<Order> findAllByAuthor(String author);
}
```

14. Create `OrderServiceImpl` class in `src/main/java/service` for the skeleton to avoid compile errors.

```
@Service
public class OrderServiceImpl implements OrderService {
    @Autowired
    private OrderRepository orderRepository;
    2 usages
    @Override
    public Order createOrder(Order order) {return null;}
    3 usages
    @Override
    public Order updateStatus(String orderId, String status) {return null;}
    2 usages
    @Override
    public List<Order> findAllByAuthor(String author) {return null;}
    2 usages
    @Override
    public Order findById(String orderId) {return null;}
}
```

15. **Run the test. You can see all the tests have failed.**



16. **Commit the changes to Git with this message [RED] Add OrderServiceImpl skeleton.**
17. **Push the changes to branch `order`.**

[GREEN] Implement Order service to pass tests

1. **Go to `OrderService` class in `src/main/java/repository`.**
2. Implement `createOrder` method.

```
@Override
public Order createOrder(Order order) {
    if (orderRepository.findById(order.getId()) == null) {
        orderRepository.save(order);
        return order;
    }
    return null;
}
```

3. **Run the test, some of the tests should be passed.**
4. Implement `updateStatus` method.

```
@Override
public Order updateStatus(String orderId, String status) {
    Order order = orderRepository.findById(orderId);
    if (order != null) {
        Order newOrder = new Order(order.getId(), order.getProducts(),
            order.getOrderTime(), order.getAuthor(), status);
        orderRepository.save(newOrder);
        return newOrder;
    } else {
        throw new NoSuchElementException();
    }
}
```

5. **Run the test again, some of the tests should be passed.**

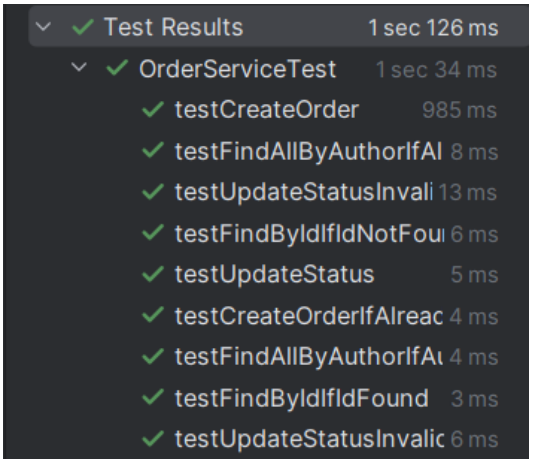
6. Implement `findById` method.

```java
@Override
public Order findById(String orderId) {
    return orderRepository.findById(orderId);
}
```

7. **Run the test again, some of the tests should be passed.**
8. Implement `findAllByAuthor` method.

```java
@Override
public List<Order> findAllByAuthor(String author) {
    return orderRepository.findAllByAuthor(author);
}
```

9. **Run the test, all the tests should be passed.**



10. **Commit the changes to Git with this message [GREEN] Implement OrderRepository class.**
11. **Push the changes to branch `order`. Do not merge the `order` branch to `main`/`master` yet.**

Reflection

You have followed the Test-Driven Development workflow in the Exercise. Now answer these questions:

1. Reflect based on Percival (2017) proposed self-reflective questions (in “Principles and Best Practice of Testing” submodule, chapter “Evaluating Your Testing Objectives”), whether this TDD flow is useful enough for you or not. If not, explain things that you need to do next time you make more tests.
2. You have created unit tests in Tutorial. Now reflect whether your tests have successfully followed F.I.R.S.T. principle or not. If not, explain things that you need to do the next time you create more tests.

Please write your reflection inside the repository's README.md file.

Exercise: Implement a New Feature

Exercise Preparation

1. Do this exercise by continuing your work on the **order** branch.
2. Carefully read Payment feature and its sub-features descriptions below.
3. In this eShop project, we want to give users freedom to choose which payment method they prefer to complete the order. You will implement two sub-features:
    a. Every student must implement Payment by Voucher Code.
    b. If your student number (NPM) is odd (for example: 1606895601), implement Cash on Delivery.
    c. If your student number (NPM) is even (for example: 1606895606), implement Payment by Bank Transfer.

Payment Feature Description

Payment's model has these attributes:

- `id`: String
- `method`: String to save a sub-feature name.
- `status`: String
- `paymentData`: `Map<String, String>` to save payment sub-feature data.

Payment's service has these functions:

- `public Payment addPayment(Order order, String method, Map<String, String> paymentData)`
  This method will create a new payment object for the current order. This method will also automatically save the new payment object to `PaymentRepository`.
- `public Payment setStatus(Payment payment, String status)`
  This method will set status for the current payment.
    a. If the payment status is set to "SUCCESS", then the status of the Order object that is related to the Payment object will also be "SUCCESS".
    b. If the payment status is set to "REJECTED", then the status of the Order object that is related to the Payment object will be "FAILED".
- `public Payment getPayment(String paymentId)`
  This method will get the `Payment` object by its `paymentId`.
- `public Payment getAllPayments()`
  This method will return all `Payment` objects.

Payment by Voucher Code Sub-feature Description

Cash on Delivery sub-feature will fill the Map<String, String> paymentData parameter when creating new Payment using this key-value pairs:

1. "`voucherCode`": the voucher code

The payment status will automatically be "SUCCESS" if the voucher code follows these rules:

1. The voucher code must be 16 characters long, and
2. The voucher code must be started with "ESHOP", and
3. The voucher code must contain 8 numerical characters.

Valid voucher code example is: "ESHOP1234ABC5678". If the voucher code is invalid, the payment status will automatically be "REJECTED".

Cash On Delivery Sub-feature Description

Cash on Delivery sub-feature will fill the Map<String, String> paymentData parameter when creating new Payment using this key-value pairs:

1. "`address`": the address of delivery.
2. "`deliveryFee`": the delivery fee that will be added.

The payment status will automatically be "REJECTED" if one of those information is empty (empty string or null).

Payment by Bank Transfer Sub-feature Description

Bank Transfer sub-feature will fill the Map<String, String> paymentData parameter when creating new Payment using this key-value pairs:

1. "`bankName`": the bank name.
2. "`referenceCode`": the reference code shown in the transfer invoice.

The payment status will automatically be "REJECTED" if one of those information is empty (empty string or null).

Exercise Checklist

1. Continue your work on the ***order*** branch.
2. Follow TDD workflow rules and implement:
    a. Payment model class
    b. Payment repository class

c. Payment service class
d. Payment by voucher sub-feature code
e. Cash on Delivery sub-feature code, or Payment by Bank Transfer sub-feature code (depends on your student number)

3. Commit history must contain **minimum 3 commits <u>for each class</u>** and commits must follow TDD cycle (RED [making tests], GREEN [implementation], REFACTOR). Add prefix on the commit message to signify each phase (for example: "[RED] Add tests for addPayment function").
4. Create pull request/merge request from **`order`** branch to **`main`**/**`master`** branch, but **do not merge it**. Keep the pull request/merge request open for Bonus 2.

Bonus 1: TDD to Create Controllers and UI for Order and Payment

Order's Controller

OrderController has these endpoints:

- `GET /order/create`
  To show create order form.
- `GET /order/history`
  To show history (input name) form. The user will be asked to input their name.
- `POST /order/history`
  To show all orders made by someone (author = inputted name).

You must also add two endpoints to Order's controller (you must complete the Tutorial before doing this):

- `GET /order/pay/[orderId]`
  This endpoint will show the payment order page.
- `POST /order/pay/[orderId]`
  This endpoint will accept a request to pay an order, and it will return a page with payment ID.

Payment's Controller

PaymentController has these endpoints:

- `GET /payment/detail`
  To show the payment detail form.
- `GET /payment/detail/[paymentId]`
  To show the details of a payment based on its ID.
- `GET /payment/admin/list`
  To show all payments.
- `GET /payment/admin/detail/[paymentId]`
  To show details of a payment based on its ID and options to reject/accept payment.
- `POST /payment/admin/set-status/[paymentId]`
  To set the status of a payment based on its ID and options to reject/accept payment.

You must also add two endpoints to OrderController:

- `GET /order/pay/[orderId]`
  This endpoint will show the payment order page.
- `POST /order/pay/[orderId]`
  This endpoint will accept a request to pay an order, and it will return a page with payment ID.

Bonus 1 Checklist

To fulfil the bonus points for Bonus 1, you must do the following:

1. Implement Thymeleaf templates for `Order` feature. Commit it with [CHORES] tag in the commit message.
2. Implement functional tests for `Order` feature. Commit it with [RED] tag in the commit message.
3. Implement `OrderController` in `src/main/java/controller`. Test it until the functional tests are passed. Then, commit it with the [GREEN] tag in the commit message.
4. Implement functional tests for Payment features. Commit it with [RED] tag in the commit message.
5. Implement `PaymentController` in `src/main/java/controller`. Test it until the functional tests are passed. Then, commit it with the [GREEN] tag in the commit message.
6. Implement Thymeleaf templates for `Payment` feature. Commit it with [CHORES] tag in the commit message.
7. Implement Thymeleaf templates for `/order/pay` pages. Commit it with [CHORES] tag in the commit message.
8. Push them to the **`order`** branch. **Do not merge it by yourself**.
9. Make sure that the **line coverage and branch coverage is 100%**.

Bonus 2: Refactor Other's Code

In this bonus exercise, we want to simulate a teamwork scenario. You are instructed to review and contribute to refactor the code made by one of your group project teammates. Your code will also be reviewed and refactored by one of your teammates.

Preparation

1. Look at the group project members list made by the Teaching Assistant team. **You will review the code made by a teammate below you** (for example: if you are no. 3, review teammate no. 4's code). If you are the last person in your group, review teammate no. 1.

No	Name	Reviews Code Owned By
1	Affan	Bambang (2)
2	Bambang	Cynthia (3)
3	Cynthia	Dimas (4)
4	Dimas	Edi (5)
5	Edi	Affan (1)

2. **Add the teammate that will review your code to your repository as a member.**
    a. In GitHub: Settings > Collaborators > Manage Access > Add people.
    b. In GitLab: Manage > Members > Invite Members. Use the "Developer" role.
3. **Add the teammate that will review your code to your pull request/merge request as a Reviewer.**

Contributing to Your Teammate's Code

To fulfil the bonus points for Bonus 2, you must do the following:

1. Look at your teammate's code repository. Find the pull request/merge request that will merge the **`order`** branch to the `main`/`master` branch. **Do not merge them, else you will make your teammate angry!**
2. Review your friend's code and add comments if you find any code smells. The comment must have these points:
    a. What is the code smell and how it will affect code maintainability.
    b. Refactoring steps that you suggest.
3. Create a new branch named **`refactor/[NPM]`** based on the latest version of your teammate's **`payment`** branch. For example, **`refactor/1606895606`**.
4. **Refactor your teammate's code, commit those changes in the `refactor/[NPM]` branch.**
5. **Test the code after refactoring, make sure other features don't break.**
6. **Make a new pull request/merge request from your refactoring branch to `order` branch.**

Bonus Reflection (mandatory to get points for Bonus 2)

1. Explain what you think about your partner's code? Are there any aspects that are still lacking from your partner's code?
2. What did you do to contribute to your partner's code?
3. What code smells did you find on your partner's code?
4. What refactoring steps did you suggest and execute to fix those smells?

Please write your reflection inside the repository's README.md file.

Grading Scheme

Scale

All components will be scored on scale 0-4. Grade 4 is the highest grade, equivalent to A. Score 0 is for not submitting.

Components

- 40% - Commits (Tutorial)
- 30% - Commits (Exercise)
- 20% - Correctness - Exercise
- 10% - Reflection (5% for Reflection 1, 5% for Reflection 2)
- 10% - Commits - Bonus 1 (max. Score 2 if line & branch coverage < 100%)
- 10% - Correctness - Bonus 2 (Score 0 if no trace of commits & comments in MR)

Rubrics

	Score 4	Score 3	Score 2	Score 1
Commits (Tutorial)	100% of the commits **in the PR/MR** are correct according to the tutorial.	>= 75% of the commits **in the PR/MR** are correct.	>=50% of the commits **in the PR/MR** are correct.	<50% of the commits are correct (**or >50% but no PR/MR**).
Commits (Exercise & **Bonus 1**)	100% of the commits **in the PR/MR** are following the TDD flow.	>=75% of the commits **in the PR/MR** are following the TDD flow.	>=50% of the commits **in the PR/MR** are following the TDD flow.	<50% of the commits are following the TDD flow.
Correctness (Exercise 1)	Implemented the program correctly and tests can be run successfully.	Tests are all passed but the program still has some errors.	Program works but tests are partially done (<70% coverage).	Incorrect program OR no test at all.
Correctness (**Bonus 2**)	Teammate's code runs successfully and no noticeable code smells/bugs.	Teammate's code runs successfully but still has noticeable code smells/bugs.	Teammate's code runs successfully but there are some errors in the test.	Teammate's code failed to run after refactoring.
Reflections	The description is sound, and the analysis is comprehensive.	The description is sound.	The description is not sound although still related.	The description is not sound. It is not related to the topics.