SOFTWARE ENGINEERING FOR TESTERS

Powered by OpenValue and Mauro Palsgraaf

16 juli 2020

INTRODUCTION

How much coding skill do you need for automation?

WHO ARE YOU?

WHO AM I?

AGENDA

DAY 1: THEORY + EXERCISES

- Introduction
- Object Oriented programming
- Introduction to clean code
- Unit testing
- Refactoring exercise
- SOLID principles
- Patterns

DAY 2: BUILDING AN APPLICATION

MODULE 0

INTRODUCTION


Download from
Dreamstime.com
This watermarked comp image is for previewing purposes only.

WHY IS CLEAN CODE IMPORTANT?

WHY IS CLEAN CODE IMPORTANT?

- > *"There will be code"*

  - Robert C. Martin

WHY IS CLEAN CODE IMPORTANT?

- > *"There will be code"*

  - Robert C. Martin
- More readers than authors.

WHY IS CLEAN CODE IMPORTANT?

- > *"There will be code"*

  - Robert C. Martin
- More readers than authors.
- Understandability affects the cost of change

WHY IS CLEAN CODE IMPORTANT?

- > *"There will be code"*

  - Robert C. Martin
- More readers than authors.
- Understandability affects the cost of change
- Working in a clean environment affects your state of mind.

UNDERSTANDABILITY

"As they added more and more features, the code got worse and worse until they simply could not manage it any longer. It was the bad code that brought the company down."

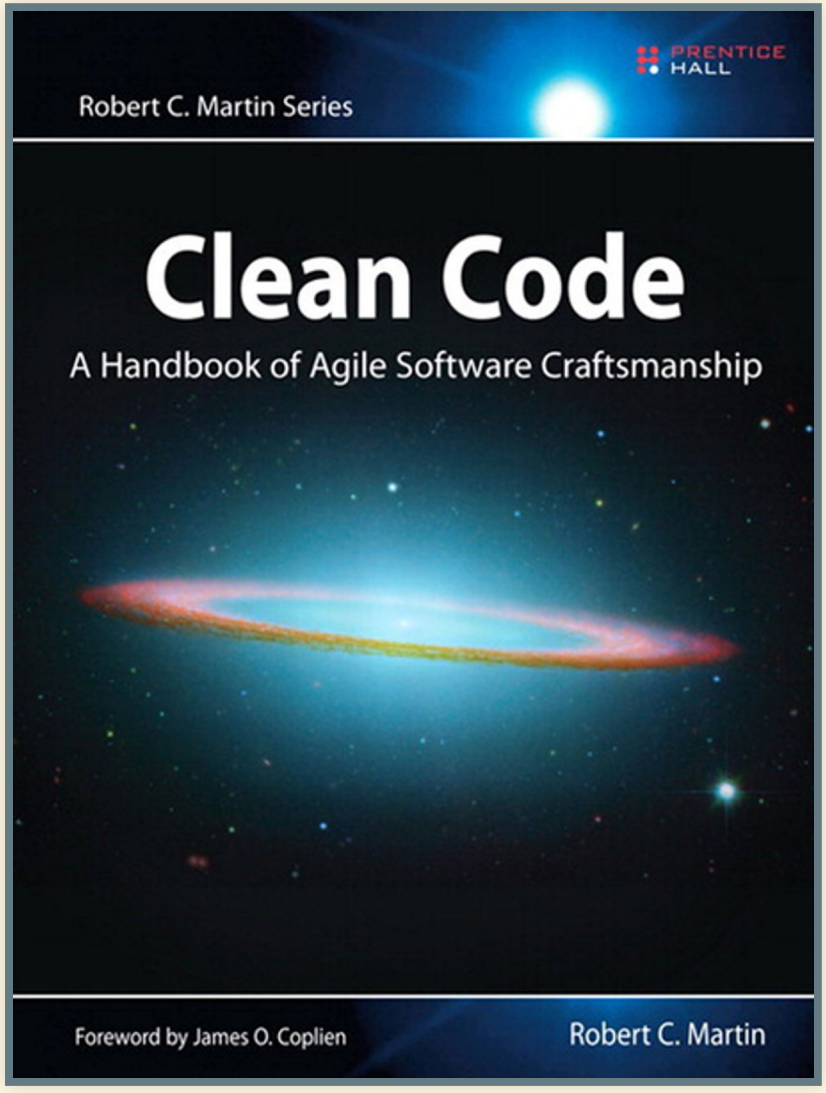
"Clean code - Uncle bob"

PREDICTABILITY

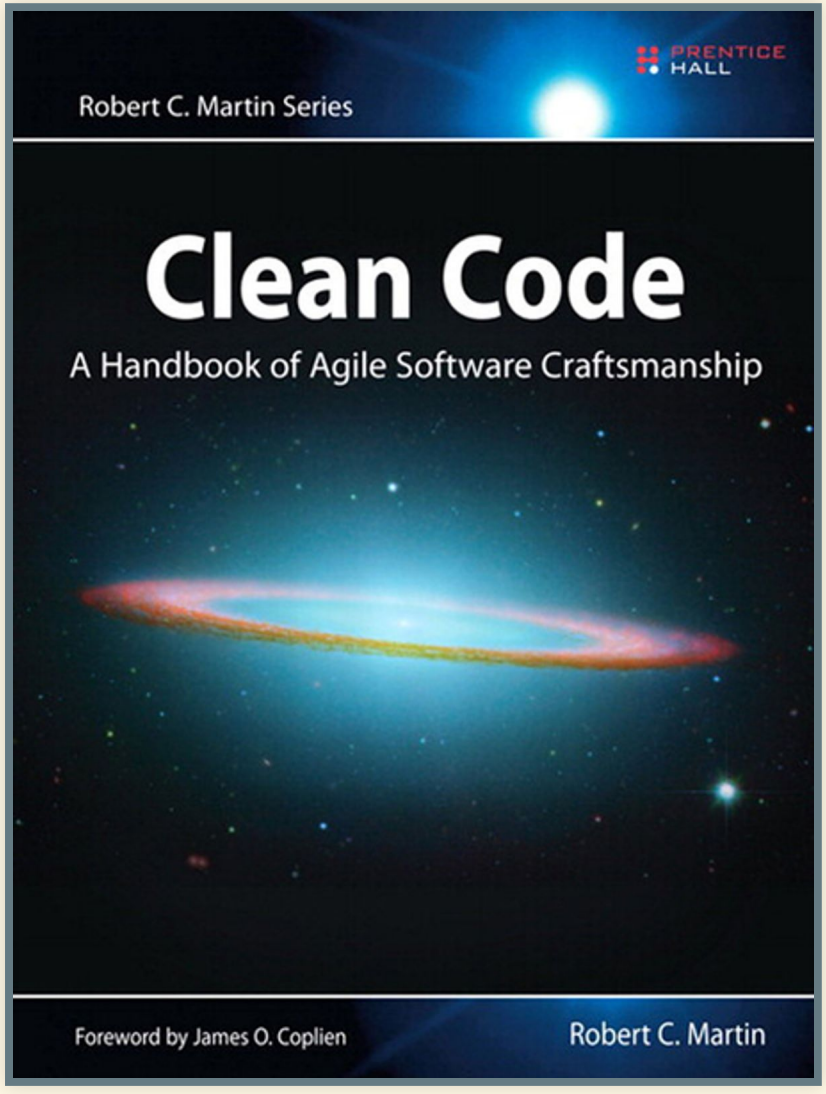
For a business, it can be crucial to know the pace in which we can develop.

If the code is a mess, untested, not automated, the cost of development is unpredictable.

THE 'BIBLE'


PRENTICE HALL
Robert C. Martin Series
Clean Code
A Handbook of Agile Software Craftsmanship
Foreword by James O. Coplien
Robert C. Martin

THE 'BIBLE'



POINTS OF INTEREST

1. Readability
2. Isolation of responsibility
3. Structure/organisation
4. No duplication
5. Less code

MODULE 1

OBJECT ORIENTED PROGRAMMING

TOPICS

TOPICS

- What is Object-oriented programming?

TOPICS

- What is Object-oriented programming?
- Characteristics

TOPICS

- What is Object-oriented programming?
- Characteristics
- Classes

TOPICS

- What is Object-oriented programming?
- Characteristics
- Classes
- Encapsulation

TOPICS

- What is Object-oriented programming?
- Characteristics
- Classes
- Encapsulation
- Access modifiers

TOPICS

- What is Object-oriented programming?
- Characteristics
- Classes
- Encapsulation
- Access modifiers
- Inheritance

TOPICS

- What is Object-oriented programming?
- Characteristics
- Classes
- Encapsulation
- Access modifiers
- Inheritance
- Polymorphism

TOPICS

- What is Object-oriented programming?
- Characteristics
- Classes
- Encapsulation
- Access modifiers
- Inheritance
- Polymorphism
- Interface

WHAT IS OBJECT-ORIËNTED PROGRAMMING?

WHAT IS OBJECT-ORIËNTED PROGRAMMING?

Object-oriented programming (OOP) is a programming paradigm based on the concept of "objects", which can contain data, in the form of fields (often known as attributes or properties), and code, in the form of procedures (often known as methods).

CHARACTERISTICS

CHARACTERISTICS

- Encapsulation

CHARACTERISTICS

- Encapsulation
- Inheritance

CHARACTERISTICS

- Encapsulation
- Inheritance
- Polymorphism

CHARACTERISTICS

- Encapsulation
- Inheritance
- Polymorphism
- Abstraction

SHORT INTRO IN JAVA

```java
public class Customer {
    private String firstName;
    private String lastName;

    public String getFirstName() {
        return this.firstName;
    }

    public String getLastName() {
        return this.lastName;
    }
}
```

```
// if no constructor is provided in a class, it will have an empt
Customer customer = new Customer();
```

```java
public class Customer {
    private String firstName;
    private String lastName;

    public Customer(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    // getters and setters
}
```

ACCESS MODIFIERS

- Public
- Protected
- Private
- Package private (no keyword)

ENCAPSULATION

In object-oriented programming (OOP), encapsulation refers to the bundling of data with the methods that operate on that data, or the restricting of direct access to some of an object's components. Encapsulation is used to hide the values or state of a structured data object inside a class, preventing unauthorized parties' direct access to them.

ENCAPSULATION

```
class Account {
    private int account_number;
    private int account_balance;

    public void show Data() {
        //code to show data
    }

    public void deposit(int a) {
        if (a < 0) {
            //show error
        } else
            account_balance = account_balance + a;
    }
}
```

ENCAPSULATION

ENCAPSULATION

- The fields of a class can be made read-only or write-only.

ENCAPSULATION

- The fields of a class can be made read-only or write-only.
- A class can have total control over what is stored in its fields.

INHERITANCE

INHERITANCE

Super Class: The class whose features are inherited is known as super class(or a base class or a parent class).

INHERITANCE

Sub Class: The class that inherits the other class is known as sub class(or a derived class, extended class, or child class). The subclass can add its own fields and methods in addition to the superclass fields and methods.

INHERITANCE

Reusability: As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

INHERITANCE

REUSABILITY

INHERITANCE

```
class Bicycle
{
    // the Bicycle class has two fields
    public int gear;
    public int speed;

    // the Bicycle class has one constructor
    public Bicycle(int gear, int speed)
    {
        this.gear = gear;
        this.speed = speed;
    }

    // the Bicycle class has three methods
    public void applyBrake(int decrement)
    {
```

INHERITANCE

```
class MountainBike extends Bicycle
{

    // the MountainBike subclass adds one more field
    public int seatHeight;

    // the MountainBike subclass has one constructor
    public MountainBike(int gear,int speed,
                        int startHeight)
    {
        // invoking base-class(Bicycle) constructor
        super(gear, speed);
        seatHeight = startHeight;
    }

    // the MountainBike subclass adds one more method
```

INHERITANCE - DIAMOND PROBLEM

```
public abstract class A {
    public abstract void doSomething();
}
```

INHERITANCE - DIAMOND PROBLEM

```
public class B extends A {
    public void doSomething() { ... }
}

public class C extends A {
    public void doSomething() { ... }
}
```

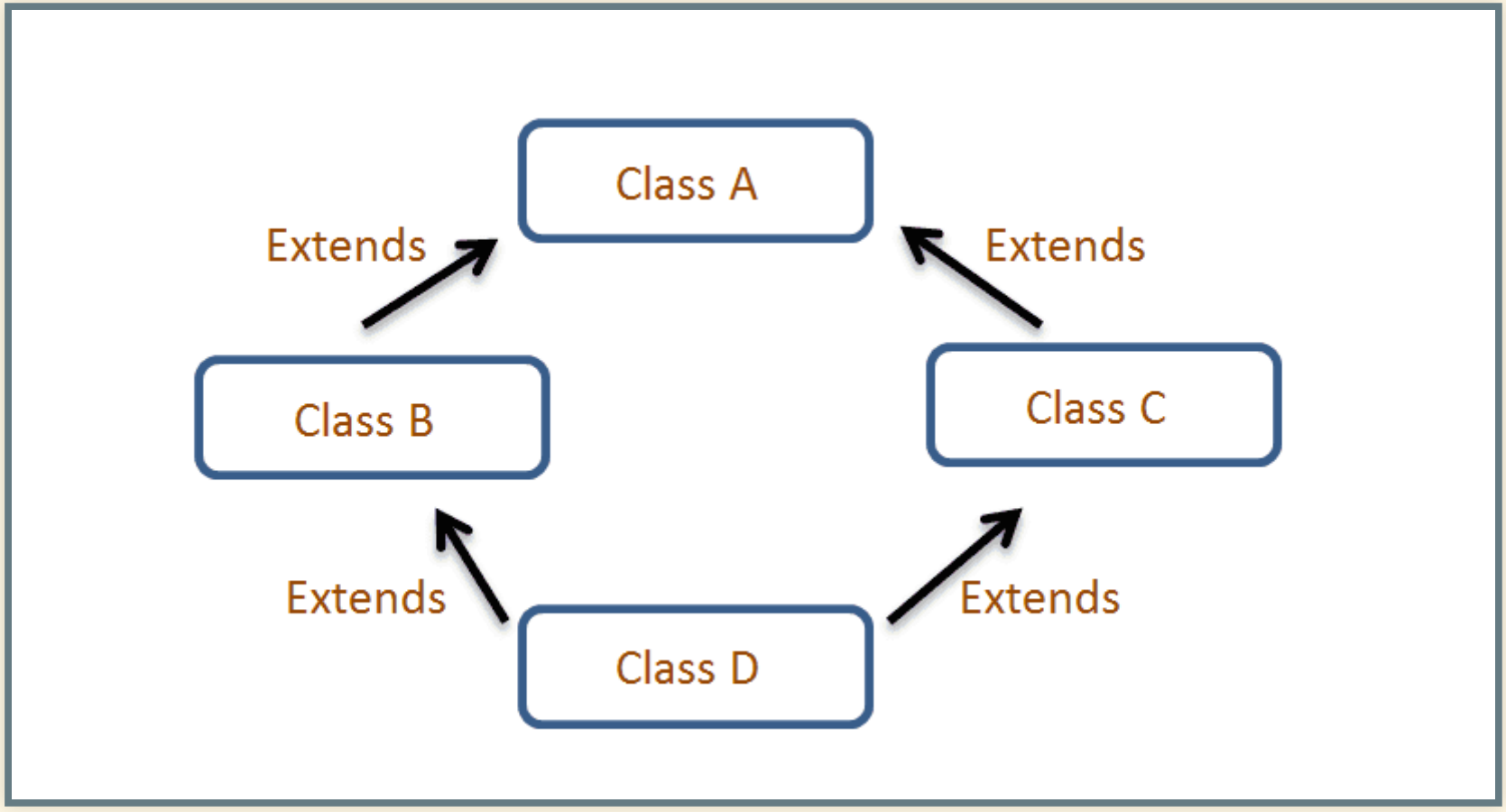
INHERITANCE - DIAMOND PROBLEM

```
public class D extends B, C { }
```

INHERITANCE - DIAMOND PROBLEM

```
D d = new D();
// What should this do?
// Call doSomething from B or C?
d.doSomething();
```

INHERITANCE - DIAMOND PROBLEM



RUNTIME POLYMORPHISM

Polymorphism is the capability of a method to do different things based on the object that it is acting upon

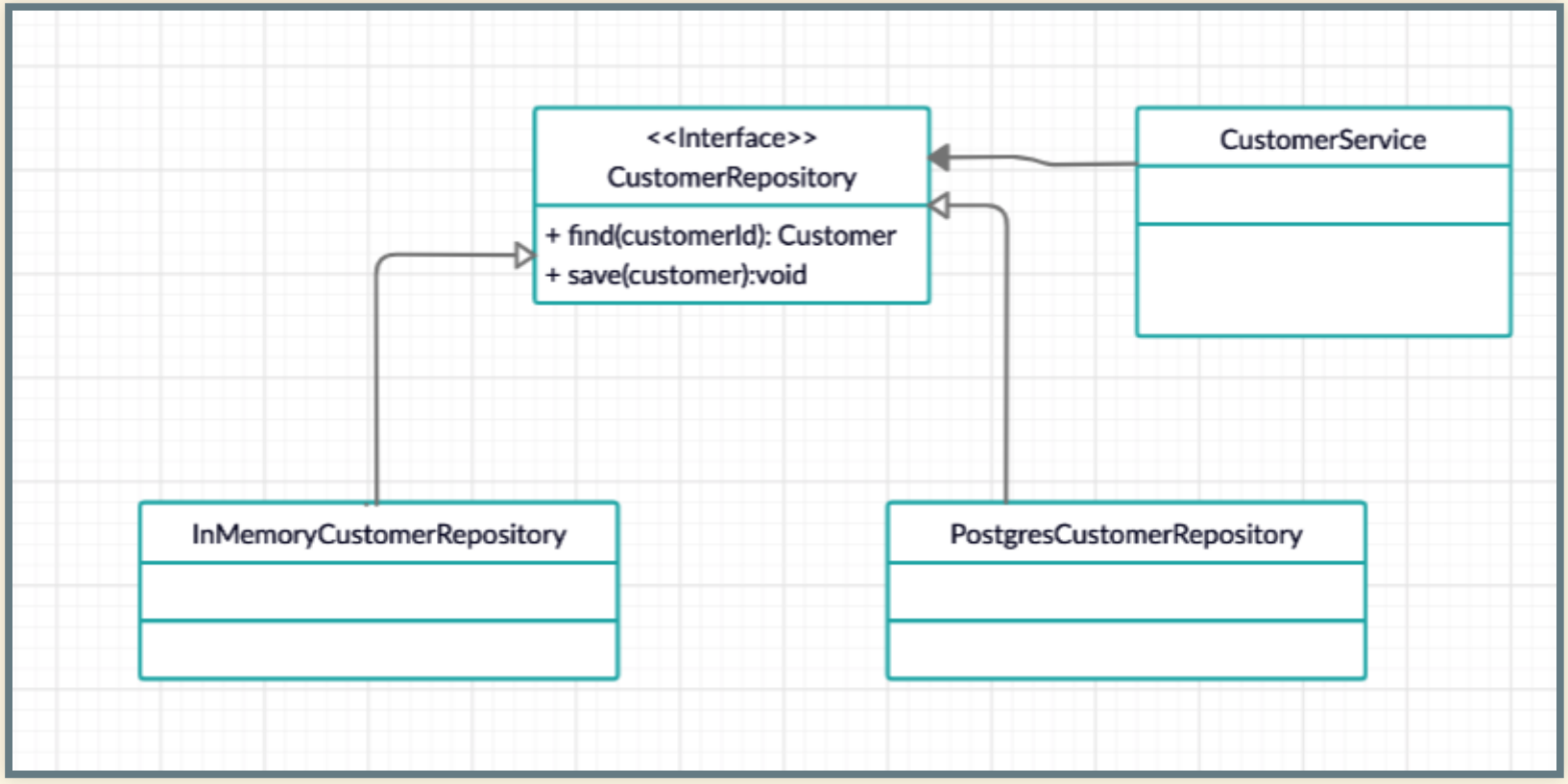
COMPILETIME POLYMORPHISM

Method overloading is a form of compile time polymorphism.

INTERFACES

An interface defines a collection of abstract methods. Classes can implement interfaces, thereby inheriting the abstract methods of the interface.

INTERFACES


<<Interface>>
CustomerRepository
+ find(customerId): Customer
+ save(customer):void
CustomerService
InMemoryCustomerRepository
PostgresCustomerRepository

INTERFACES

```
public interface CustomerRepository {
    Customer find(int customerId);
    void save(Customer customer);
}
```

INTERFACES

```
public class InMemoryCustomerRepository implements CustomerReposi
    public Customer find(int customerId) { ... }
    public void save(Customer customer) { ... }
}
```

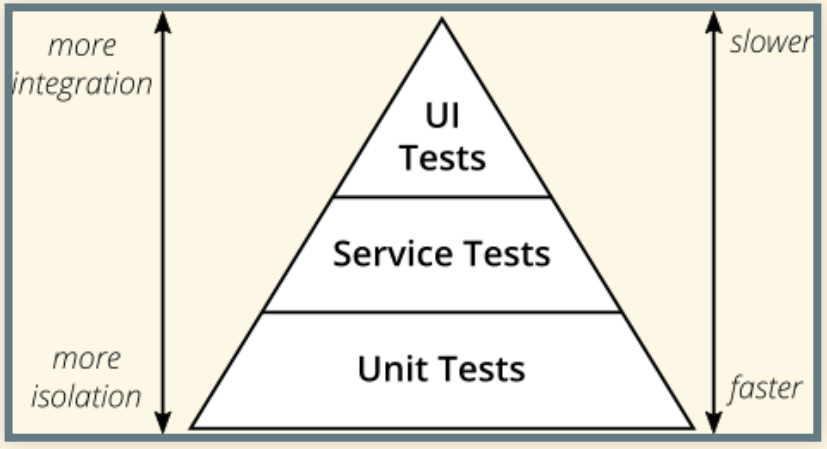
INTERFACES

```
public class PostgresCustomerRepository implements CustomerReposi
    public Customer find(int customerId) { ... }
    public void save(Customer customer) { ... }
}
```

MODULE 2

UNIT TESTING

THE TESTING PYRAMID



UNIT TESTING

Unit testing is a level of software testing where individual units/ components of a software are tested.

BENEFITS OF UNIT TESTS

- Increases confidence in changing / maintaining code
- Code is more reusable
- Code is more reliable
- Increases development speed

CLEAN TEST SHOULD FOLLOW F.I.R.S.T?

CLEAN TEST SHOULD FOLLOW F.I.R.S.T?

- Fast

CLEAN TEST SHOULD FOLLOW F.I.R.S.T?

- Fast
- Independent

CLEAN TEST SHOULD FOLLOW F.I.R.S.T?

- Fast
- Independent
- Repeatable

CLEAN TEST SHOULD FOLLOW F.I.R.S.T?

- **F**ast
- **I**ndependent
- **R**epeatable
- **S**elf-validation

CLEAN TEST SHOULD FOLLOW F.I.R.S.T?

- **F**ast
- **I**ndependent
- **R**epeatable
- **S**elf-validation
- **T**imely

ARRANGE ACT ASSERT (AAA)

UNIT TESTING IN JAVA

```
public class Calculator {
    public int sum(int left, int right) {
        return left + right;
    }
}
```

UNIT TESTING IN JAVA

```
public class CalculatorTest {
    @Test
    public void shouldSumTheInputs() {
        Calculator calculator = new Calculator();

        int result = calculator.sum(2, 3);
        assertEquals(5, result);
    }
}
```

UNIT TESTING IN JAVA

```
public class CalculatorTest {
    private Calculator calculator;

    @Before
    public void setUp() {
        this.calculator = new Calculator();
    }

    @Test
    public void shouldSumTheInputs() {
        int result = calculator.sum(2, 3);
        assertEquals(5, result);
    }
}
```

UNIT TESTING IN JAVA

```
@Test(expected=ArithmeticException.class)
```

METHODS WITH SIDE EFFECTS

WHAT IS A MOCK?

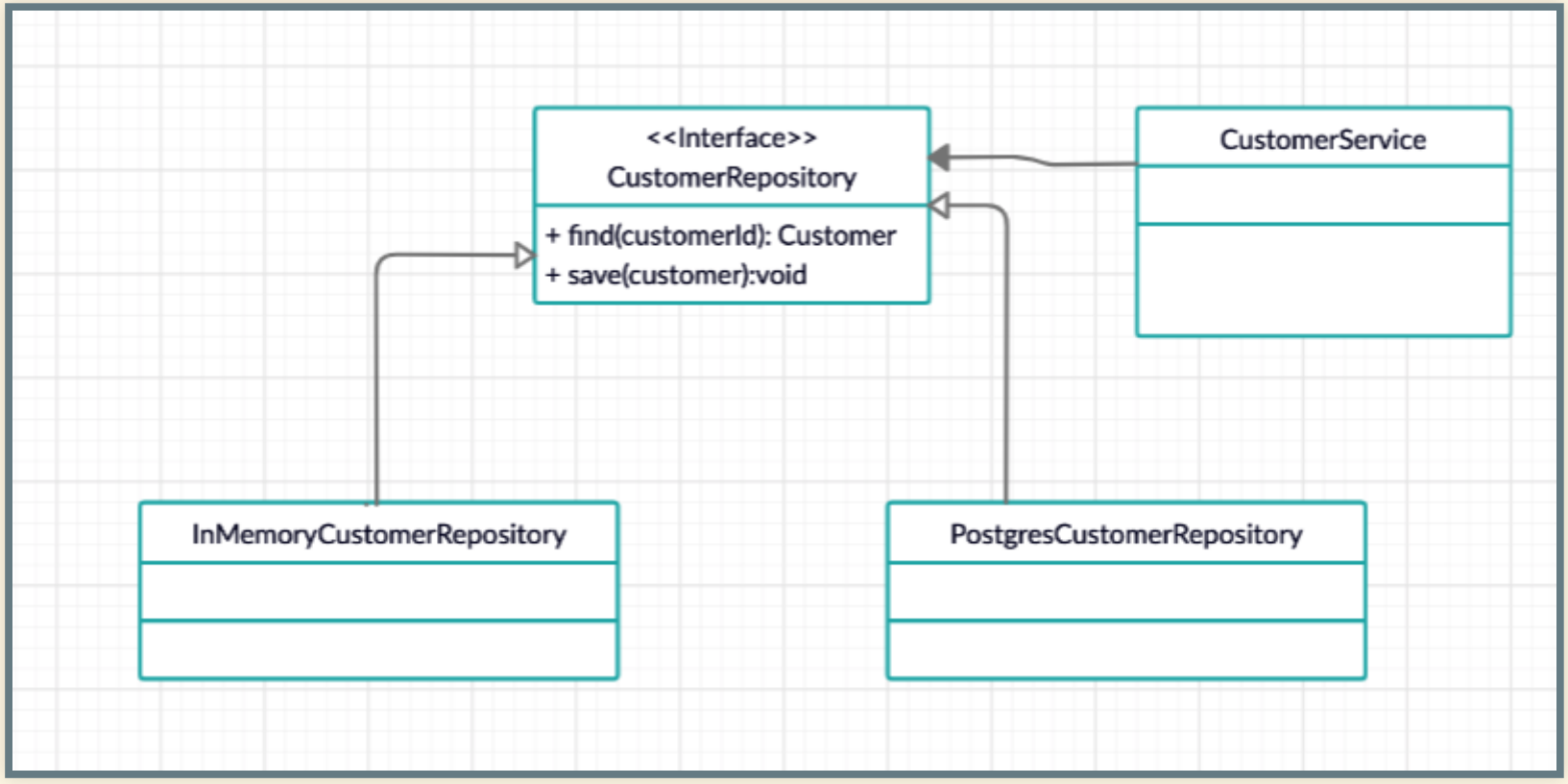
Mocks are pre-programmed with expectations which form a specification of the calls they are expected to receive. They can throw an exception if they receive a call they don't expect and are checked during verification to ensure they got all the calls they were expecting.

MOCKING TO THE RESCUE: MOCKITO

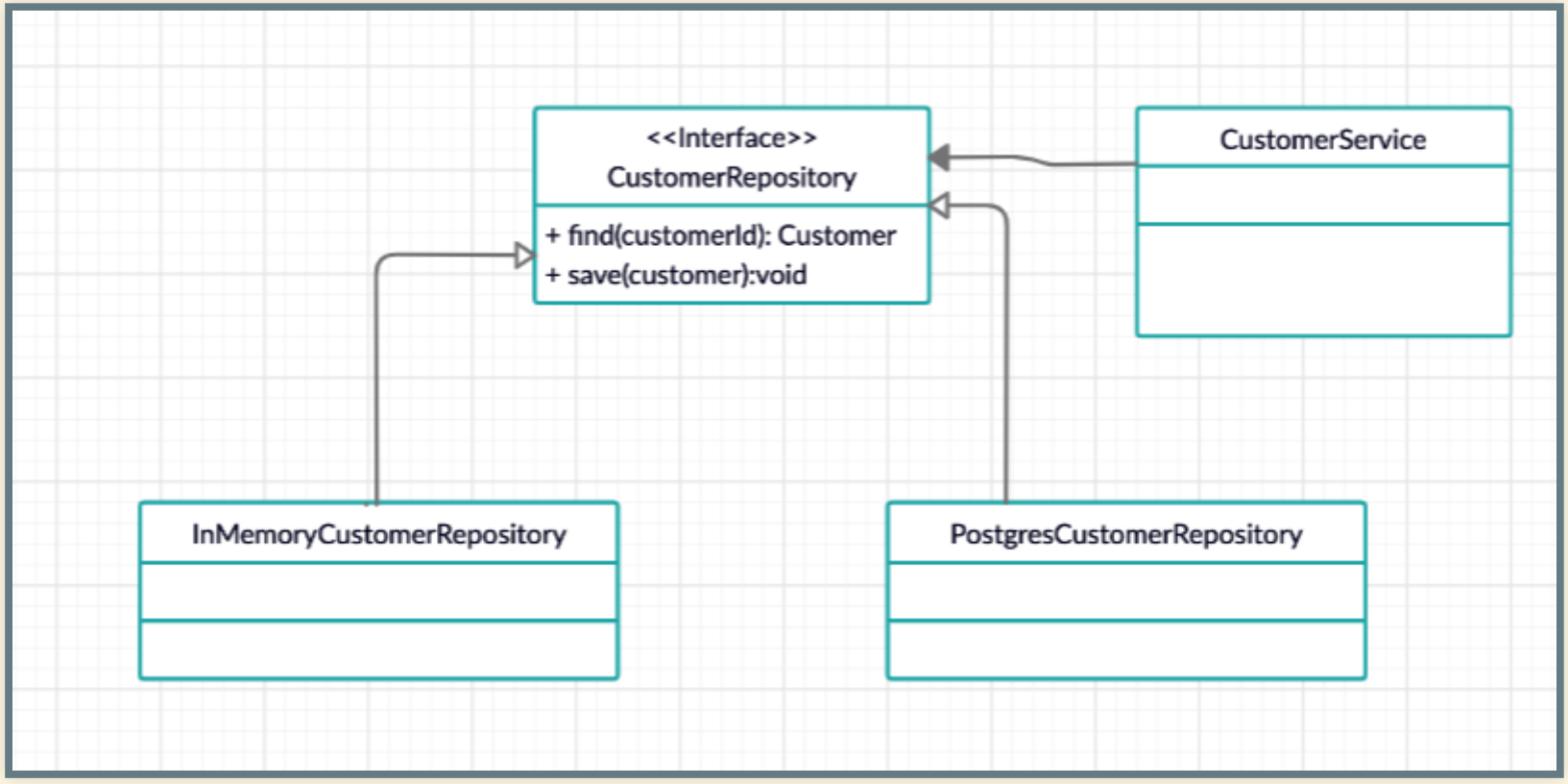
"Mockito is a mocking framework for Java. Mockito allows convenient creation of substitutes of real objects for testing purposes."

https://github.com/mockito/mockito/wiki/FAQ

MOCKING


<<Interface>>
CustomerRepository
+ find(customerId): Customer
+ save(customer):void
CustomerService
InMemoryCustomerRepository
PostgresCustomerRepository

MOCKING


<<Interface>>
CustomerRepository
+ find(customerId): Customer
+ save(customer):void
CustomerService
InMemoryCustomerRepository
PostgresCustomerRepository

MOCKING

```
public class UnemploymentServiceImpl implements UnemploymentServi

    private JobService jobService;

    public UnemploymentServiceImpl(JobService jobService) {
        this.jobService = jobService;
    }

    @Override
    public boolean personIsEntitledToUnemploymentSupport(Person p
        Optional<JobPosition> optional = jobService.findCurrentJo

        return !optional.isPresent();
    }
}
```

MOCKING

```
public class UnemploymentServiceImplTest {

    @Mock
    private JobService jobService;

    @InjectMocks
    private UnemploymentServiceImpl unemploymentService;

    @Test
    public void givenReturnIsOfTypeOptional_whenMocked_thenValueI
}
```

MOCKING

```
public class UnemploymentServiceImplTest {
    private JobService jobService;
    private UnemploymentServiceImpl unemploymentService;

    @Before
    public void setUp() {
        this.jobService = Mockito.mock(JobService.class);
        this.unemploymentService = new UnemploymentServiceImpl(th
    }

    @Test
    public void givenReturnIsOfTypeOptional_whenMocked_thenValueI
}
```

MOCKING

```
1  public class UnemploymentServiceImplTest {
2
3      @Test
4      public void testIsEntitledIfPersonHasJobPosition() {
5
6          Person person = new Person();
7
8          when(
9              jobService.findCurrentJobPosition(any(Person.class
10         ).thenReturn(Optional.empty());
11
12         assertTrue(unemploymentService.personIsEntitledToUnemp
13
14         verify(customerRepository, times(1)).getCustomer(1);
15     }
16 }
```

MOCKING

```
1  public class UnemploymentServiceImplTest {
2
3      @Test
4      public void testIsEntitledIfPersonHasJobPosition() {
5
6          Person person = new Person();
7
8          when(
9              jobService.findCurrentJobPosition(any(Person.class
10         ).thenReturn(Optional.empty());
11
12         assertTrue(unemploymentService.personIsEntitledToUnemp
13
14         verify(customerRepository, times(1)).getCustomer(1);
15     }
16 }
```

MOCKING

```
1  public class UnemploymentServiceImplTest {
2
3      @Test
4      public void testIsEntitledIfPersonHasJobPosition() {
5
6          Person person = new Person();
7
8          when(
9              jobService.findCurrentJobPosition(any(Person.class
10         ).thenReturn(Optional.empty());
11
12         assertTrue(unemploymentService.personIsEntitledToUnemp
13
14         verify(customerRepository, times(1)).getCustomer(1);
15     }
16 }
```

MOCKING

```
1  public class UnemploymentServiceImplTest {
2
3      @Test
4      public void testIsEntitledIfPersonHasJobPosition() {
5
6          Person person = new Person();
7
8          when(
9              jobService.findCurrentJobPosition(any(Person.class
10         ).thenReturn(Optional.empty());
11
12         assertTrue(unemploymentService.personIsEntitledToUnemp
13
14         verify(customerRepository, times(1)).getCustomer(1);
15     }
16 }
```

MOCKING

```
1  public class UnemploymentServiceImplTest {
2
3      @Test
4      public void testIsEntitledIfPersonHasJobPosition() {
5
6          Person person = new Person();
7
8          when(
9              jobService.findCurrentJobPosition(any(Person.class
10         ).thenReturn(Optional.empty());
11
12         assertTrue(unemploymentService.personIsEntitledToUnemp
13
14         verify(customerRepository, times(1)).getCustomer(1);
15     }
16 }
```

MODULE 2

INTRODUCTION TO CLEAN CODE

TOPICS

- Meaningful names
- Functions
- Comments
- Formatting
- Objects & data structures
- Error handling
- Classes

MEANINGSFUL NAMES

USE MEANINGFUL, READABLE VARIABLE NAMES

```
int d;
// elapsed time in days
int ds;
int dsm;
int faid;
```

USE MEANINGFUL, READABLE VARIABLE NAMES

```
int elapsedTimeInDays;
int daysSinceCreation;
int daysSinceModification;
int fileAgeInDays;
```

USE INTENTION REVEALING NAMES

```
public List<int[]> getThem() {
    List<int[]> list1 = new ArrayList<int[]>();
    for (int[] x : theList)
        if (x[0] == 4)
            list1.add(x);
    return list1;
}
```

USE INTENTION REVEALING NAMES

```
public List<int[]> getFlaggedCells() {
    List<int[]> flaggedCells = new ArrayList<int[]>();
    for (int[] cell : gameBoard)
        if (cell[STATUS_VALUE] == FLAGGED)
            flaggedCells.add(cell);
    return flaggedCells;
}
```

SEARCHABLE NAMES

```
int d;
int e;
int x;
int z;
```

FAVOR READABILITY OVER BREVITY

CanScrollHorizontally is better than ***ScrollableX.***

DON'T BE CUTE

Funny names remain understandable only as long as you and the people who share the joke stay on the project.

CLASSES

"Classes and objects should have noun or noun phrase names like Customer, WikiPage, Account, and AddressParser. Avoid words like Manager, Processor, Data, or Info in the name of a class. A class name should not be a verb."

Clean code - Uncle Bob

METHODS

Methods should have verb (phrase) naming. Accessors, mutators and predicates should start with get, set or is.

SMART CONSTRUCTORS

Use smart constructors when overloading the constructor.

```
Complex fulcrumPoint = Complex.FromRealNumber(23.0);

Complex fulcrumPoint = new Complex(23.0);
```

FUNCTIONS

- Small
- Do one thing
- Have explanatory names

NOT TO MANY ARGUMENTS

If your function has to many arguments, they most like are connected and should be wrapped in an object

NOT TO MANY ARGUMENTS

```
public void save(String name, String age, String jobTitle) { ...
```

NOT TO MANY ARGUMENTS

```
public void save(Employee employee) { ... }
```

COMMAND QUERY SEPARATION

```
public boolean set(String attribute, String value);
```

```
public boolean set(String attribute, String value);

if (set("username", "unclebob"))
```

```
if (attributeExists("username")) {
    setAttribute("username", "unclebob");
}
```

COMMENTS

COMMENTS ARE LAST RESORT

COMMENTS ARE LAST RESORT

```
// Check to see if the employee is eligible for full benefits
if ((employee.flags == HOURLY_FLAG) && (employee.age > 65))
```

COMMENTS ARE LAST RESORT

```
// Check to see if the employee is eligible for full benefits
if ((employee.flags == HOURLY_FLAG) && (employee.age > 65))
```

```
if (employee.isEligibleForFullBenefits())
```

FUNCTION / VARIABLES > COMMENTS

FUNCTION / VARIABLES > COMMENTS

```
// does the module from the global list  depend on the
// subsystem we are part of?
if (smodule.getDependSubsystems().contains(subSysMod.getSubSystem
```

FUNCTION / VARIABLES > COMMENTS

```
// does the module from the global list  depend on the
// subsystem we are part of?
if (smodule.getDependSubsystems().contains(subSysMod.getSubSystem
```

```
ArrayList moduleDependees = smodule.getDependSubsystems();
String ourSubSystem = subSysMod.getSubSystem();
if (moduleDependees.contains(ourSubSystem))
```

COMMENTS WILL LIE

REDUNDANT / NOISY COMMENTS

REDUNDANT / NOISY COMMENTS

```java
/** The name. */
private String name;

/** The version. */
private String version;

/** The licenceName. */
private String licenceName;

/** The info. */
private String info;
```

DON'T RETURN NULL

DON'T RETURN NULL

```
public void registerItem(Item item) {
    if (item != null) {
        ItemRegistry registry = peristentStore.getItemRegistry();
        if (registry != null) {
            Item existing = registry.getItem(item.getID());
            if (existing.getBillingPeriod().hasRetailOwner()) {
                existing.register(item); }
            }
        }
    }
}
```

DON'T RETURN NULL

DON'T RETURN NULL

```
List employees = getEmployees();
if (employees != null) {
    for(Employee e : employees) {
        totalPay += e.getPay();
    }
}
```

DON'T RETURN NULL

```
List employees = getEmployees();
if (employees != null) {
    for(Employee e : employees) {
        totalPay += e.getPay();
    }
}
```

```
// Get employees
public List getEmployees() {
    if (no employees) {
        return Collections.emptyList();
    }
}

List employees = getEmployees();
for(Employee e : employees) {
    totalPay += e.getPay();
}
```

USE OPTIONAL AS TYPESAFE NULL ALTERNATIVE

USE OPTIONAL AS TYPESAFE NULL ALTERNATIVE

```
    public void registerItem(Item item) {
        Optional<itemregistry> optionalRegistry = peristentStore.
        optionalRegistry
            .flatMap(registry -> registry.getItem(item.getID()))
            .flatMap(existing -> {
                if (existing.getBillingPeriod().hasRetailOwner())
                    return existing.register(item);
                }
            });
    }
</itemregistry>
```

DON'T PASS NULL

GILDED ROSE REFACTORING KATA

GILDED ROSE REFACTORING KATA

- Go to exercises/CleanCode

GILDED ROSE REFACTORING KATA

- Go to exercises/CleanCode
- Run mvn clean install to see if you get the project to run (1 test fail)

GILDED ROSE REFACTORING KATA

- Go to exercises/CleanCode
- Run mvn clean install to see if you get the project to run (1 test fail)
- Write unit tests to ensure the application works as you would expect

GILDED ROSE REFACTORING KATA

- Go to exercises/CleanCode
- Run mvn clean install to see if you get the project to run (1 test fail)
- Write unit tests to ensure the application works as you would expect
- Start refactoring by applying the clean code rules (and more)

MODULE 3

SOLID PRINCIPLES

THE PRINCIPLES

THE PRINCIPLES

- **S**ingle responsibility

THE PRINCIPLES

- **S**ingle responsibility
- **O**pen-closed

THE PRINCIPLES

- **S**ingle responsibility
- **O**pen-closed
- **L**iskov substitution principle

THE PRINCIPLES

- **S**ingle responsibility
- **O**pen-closed
- **L**iskov substitution principle
- **I**nterface segregation

THE PRINCIPLES

- **S**ingle responsibility
- **O**pen-closed
- **L**iskov substitution principle
- **I**nterface segregation
- **D**ependency inversion principle

SINGLE RESPONSIBILITY PRINCIPLE

The single-responsibility principle (SRP) is a computer-programming principle that states that every module or class should have responsibility over a single part of the functionality provided by the software, and that responsibility should be entirely encapsulated by the class, module or function.

CLASSICAL EXAMPLE

CLASSICAL EXAMPLE

```
class Book {
    public String getTitle() {
        return "A Great Book";
    }

    public String getAuthor() {
        return "John Doe";
    }

    public Page turnPage() {
        // pointer to next page
    }

    public void printCurrentPage() {
        echo "current page content";
    }
```

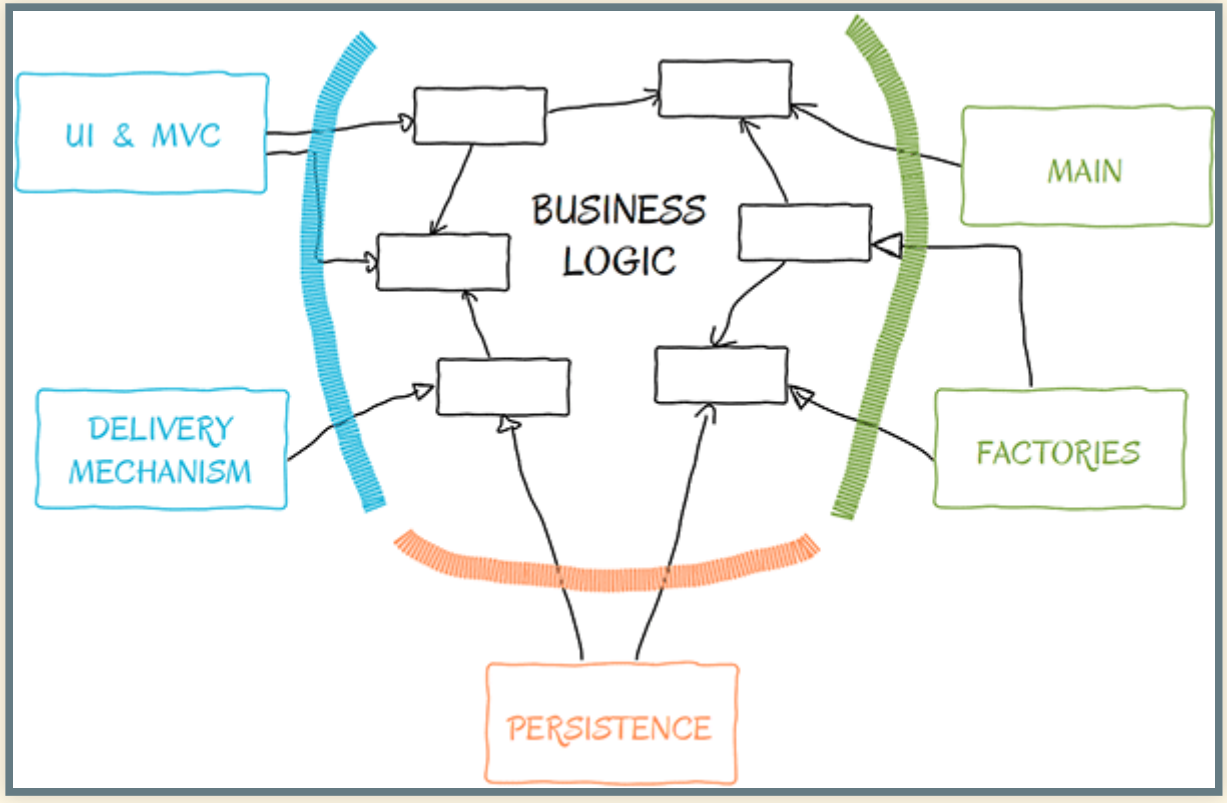
SEPERATING THE CONCERNS

```
interface Printer {
    public void printPage(Page page);
}

class PlainTextPrinter implements Printer {
    public void printPage($page) {
        echo $page;
    }
}

class HtmlPrinter implements Printer {
    public void printPage(Page page) {
        echo <div style="single-page">' . $page . '</div>';
    }
}
```

STRUCTURING AN APPLICATION



OPEN-CLOSED PRINCIPLE

In object-oriented programming, the open/closed principle states "software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification"; that is, such an entity can allow its behaviour to be extended without modifying its source code.

OPEN-CLOSED PRINCIPLE

OPEN-CLOSED PRINCIPLE

```
public interface CalculatorOperation {}
```

OPEN-CLOSED PRINCIPLE

OPEN-CLOSED PRINCIPLE

```
public class Addition implements CalculatorOperation {
    private double left;
    private double right;
    private double result = 0.0;

    public Addition(double left, double right) {
        this.left = left;
        this.right = right;
    }

    // getters and setters
}
```

OPEN-CLOSED PRINCIPLE

OPEN-CLOSED PRINCIPLE

```
public class Subtraction implements CalculatorOperation {
    private double left;
    private double right;
    private double result = 0.0;

    public Subtraction(double left, double right) {
        this.left = left;
        this.right = right;
    }

    // getters and setters
}
```

OPEN-CLOSED PRINCIPLE

```
public class Calculator {

    public void calculate(CalculatorOperation operation) {
        if (operation instanceof Addition) {
            Addition addition = (Addition) operation;
            addition.setResult(addition.getLeft() + addition.getR
        } else if (operation instanceof Subtraction) {
            Subtraction subtraction = (Subtraction) operation;
            subtraction.setResult(subtraction.getLeft() - subtrac
        }
    }
}
```

OPEN-CLOSED PRINCIPLE

```
public interface CalculatorOperation {
    int perform();
}
```

OPEN-CLOSED PRINCIPLE

```
public class Addition implements CalculatorOperation {
    private double left;
    private double right;

    // constructor

    @Override
    public int perform() {
        return left + right;
    }
}
```

OPEN-CLOSED PRINCIPLE

```
public class Multiplication implements CalculatorOperation {
    private double left;
    private double right;

    // constructor

    @Override
    public int perform() {
        return left * right;
    }
}
```

OPEN-CLOSED PRINCIPLE

```
public void calculate(CalculatorOperation operation) {
    operation.perform();
}
```

LISKOV SUBSTITUTION PRINCIPLE

"in a computer program, if S is a subtype of T, then objects of type T may be replaced with objects of type S (i.e., objects of type S may substitute objects of type T) without altering any of the desirable properties of that program (correctness, task performed, etc.)"

LISKOV SUBSTITUTION PRINCIPLE

- No new exceptions should be thrown in derived class.
- Preconditions cannot be strengthened in a subtype.
- Postconditions cannot be weakened in a subtype.
- Invariants of the supertype must be preserved in a subtype.
- Comply with the history constraint

LISKOV SUBSTITUTION PRINCIPLE

LISKOV SUBSTITUTION PRINCIPLE

```
class Person
{
    void walkNorth(int meters) {...}
    void walkEast(int meters) {...}
}
```

LISKOV SUBSTITUTION PRINCIPLE

LISKOV SUBSTITUTION PRINCIPLE

```
class Prisoner extends Person
{
    void walkNorth(int meters) {...}
    void walkEast(int meters) {...}
}
```

LISKOV SUBSTITUTION PRINCIPLE

The implementation of the derived classes may not be known for the code consumer

If you need to add some restriction in an overridden method and that restriction doesn't exist in baseline implementation, you violate Liskov Substitution Principle.

LISKOV SUBSTITUTION PRINCIPLE

LISKOV SUBSTITUTION PRINCIPLE

```
    public interface CustomerRepository {
        Optional<customer> find(int customerId);
        void save(Customer customer);
    }
</customer>
```

LISKOV SUBSTITUTION PRINCIPLE

LISKOV SUBSTITUTION PRINCIPLE

```
public class PostgresCustomerRepository {
    public Optional find(int customerId) { ... }

    public void save(Customer customer) {
        if (this.find(customer.id).isPresent()) {
            throw new RuntimeException("customer already exis
        }

        // save customer in database
    }
}
```

LISKOV SUBSTITUTION PRINCIPLE

Due to liskov substitution violations, the code that uses your type will have to have explicit knowledge of the internal workings of derived types to treat them differently. This tightly couples code and just generally makes the implementation harder to use consistently.

INTERFACE SEGREGATION PRINCIPLE

In the field of software engineering, the interface-segregation principle (ISP) states that no client should be forced to depend on methods it does not use.[1] ISP splits interfaces that are very large into smaller and more specific ones so that clients will only have to know about the methods that are of interest to them.

INTERFACE SEGREGATION PRINCIPLE

INTERFACE SEGREGATION PRINCIPLE

```
interface OrderService {
    void orderBurger(int quantity);
    void orderFries(int fries);
    void orderCombo(int quantity, int fries);
}
```

INTERFACE SEGREGATION PRINCIPLE

INTERFACE SEGREGATION PRINCIPLE

```
class BurgerOrderService implements OrderService {
    @Override
    public void orderBurger(int quantity) {
        System.out.println("Received order of "+quantity+" burger
    }

    @Override
    public void orderFries(int fries) {
        throw new UnsupportedOperationException("No fries in burg
    }

    @Override
    public void orderCombo(int quantity, int fries) {
        throw new UnsupportedOperationException("No combo in burg
    }
}
```

INTERFACE SEGREGATION PRINCIPLE

INTERFACE SEGREGATION PRINCIPLE

```
interface BurgerOrderService {
    void orderBurger(int quantity);
}

interface FriesOrderService {
    void orderFries(int fries);
}
```

DEPENDENCY INVERSION PRINCIPLE

In object-oriented design, the dependency inversion principle is a specific form of decoupling software modules. When following this principle, the conventional dependency relationships established from high-level, policy-setting modules to low-level, dependency modules are reversed, thus rendering high-level modules independent of the low-level module implementation details. The principle states:

DEPENDENCY INVERSION PRINCIPLE

DEPENDENCY INVERSION PRINCIPLE

- High-level modules should not depend on low-level modules. Both should depend on abstractions (e.g. interfaces).

DEPENDENCY INVERSION PRINCIPLE

- High-level modules should not depend on low-level modules. Both should depend on abstractions (e.g. interfaces).
- Abstractions should not depend on details. Details (concrete implementations) should depend on abstractions.

DEPENDENCY INVERSION PRINCIPLE

DEPENDENCY INVERSION PRINCIPLE

```
public class CoffeeMachine {
    public void makeCoffee() {
        System.out.println("Making coffee");
    }
}
```

DEPENDENCY INVERSION PRINCIPLE

DEPENDENCY INVERSION PRINCIPLE

```
public class CoffeeApp {
    private CoffeeMachine coffeeMachine;
    public CoffeeApp(CoffeeMachine coffeeMachine) {
        this.coffeeMachine = coffeeMachine;
    }

    public void run() { ... }
}
```

DEPENDENCY INVERSION PRINCIPLE

DEPENDENCY INVERSION PRINCIPLE

```
public interface CoffeeMachine {
    void makeCoffee();
}
```

DEPENDENCY INVERSION PRINCIPLE

DEPENDENCY INVERSION PRINCIPLE

```
public class FilterCoffeeMachine implements CoffeeMachine {
    public void makeCoffee() {
        System.out.println("Making delicious filter coffee");
    }
}
```

DEPENDENCY INVERSION PRINCIPLE

DEPENDENCY INVERSION PRINCIPLE

```
public class CoffeeApp {
    private CoffeeMachine coffeeMachine;
    public CoffeeApp(CoffeeMachine coffeeMachine) {
        this.coffeeMachine = coffeeMachine;
    }

    public void run() { ... }
}
```

MODULE 4

SOFTWARE PATTERNS

SOFTWARE PATTERNS

SOFTWARE PATTERNS

- Adapter

SOFTWARE PATTERNS

- Adapter
- Decorator

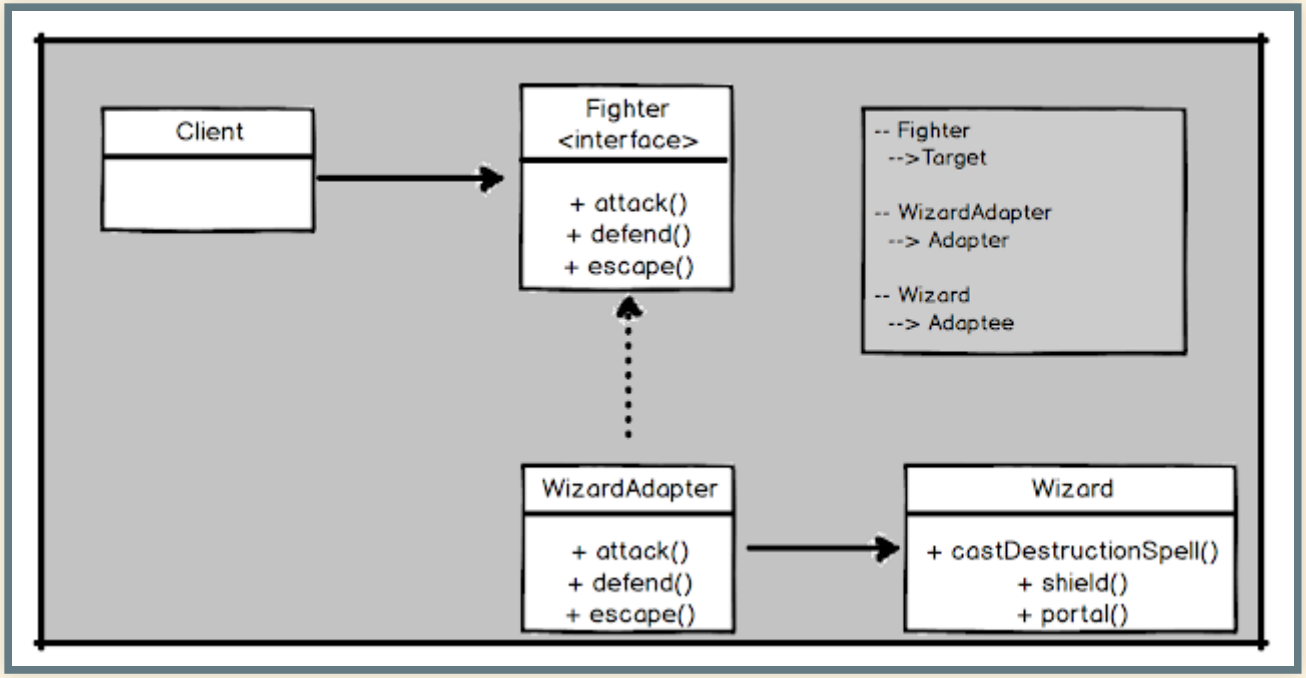
SOFTWARE PATTERNS

- Adapter
- Decorator
- Factory

ADAPTER PATTERN

"Adapter is a structural design pattern that allows objects with incompatible interfaces to collaborate." - https://refactoring.guru/design-patterns/adapter

ADAPTER PATTERN



ADAPTER PATTERN

```
public interface Fighter {
    void attack();
    void defend();
    void escape();
}
```

ADAPTER PATTERN

```
public class Wizard {
    public void castDestructionSpell() {
        System.out.println("destruction spell");
    }
    public void shield() {
        System.out.println("shield");
    }
    public void portal() {
        System.out.println("portal");
    }
}
```

ADAPTER PATTERN

```
public class WizardAdapter implements Fighter {
    private final Wizard wizard;

    public WizardAdapter(Wizard wizard) {
        this.wizard = wizard;
    }
    public void attack() { wizard.castDestructionSpell(); }
    public void defend() { wizard.shield(); }
    public void escape() { wizard.portal(); }
}
```

ADAPTER PATTERN

```
public void doSomething(Fighter fighter) { ... }
```

```
Wizard wizard = new Wizard();
doSomething(new WizardAdapter(wizard));
```

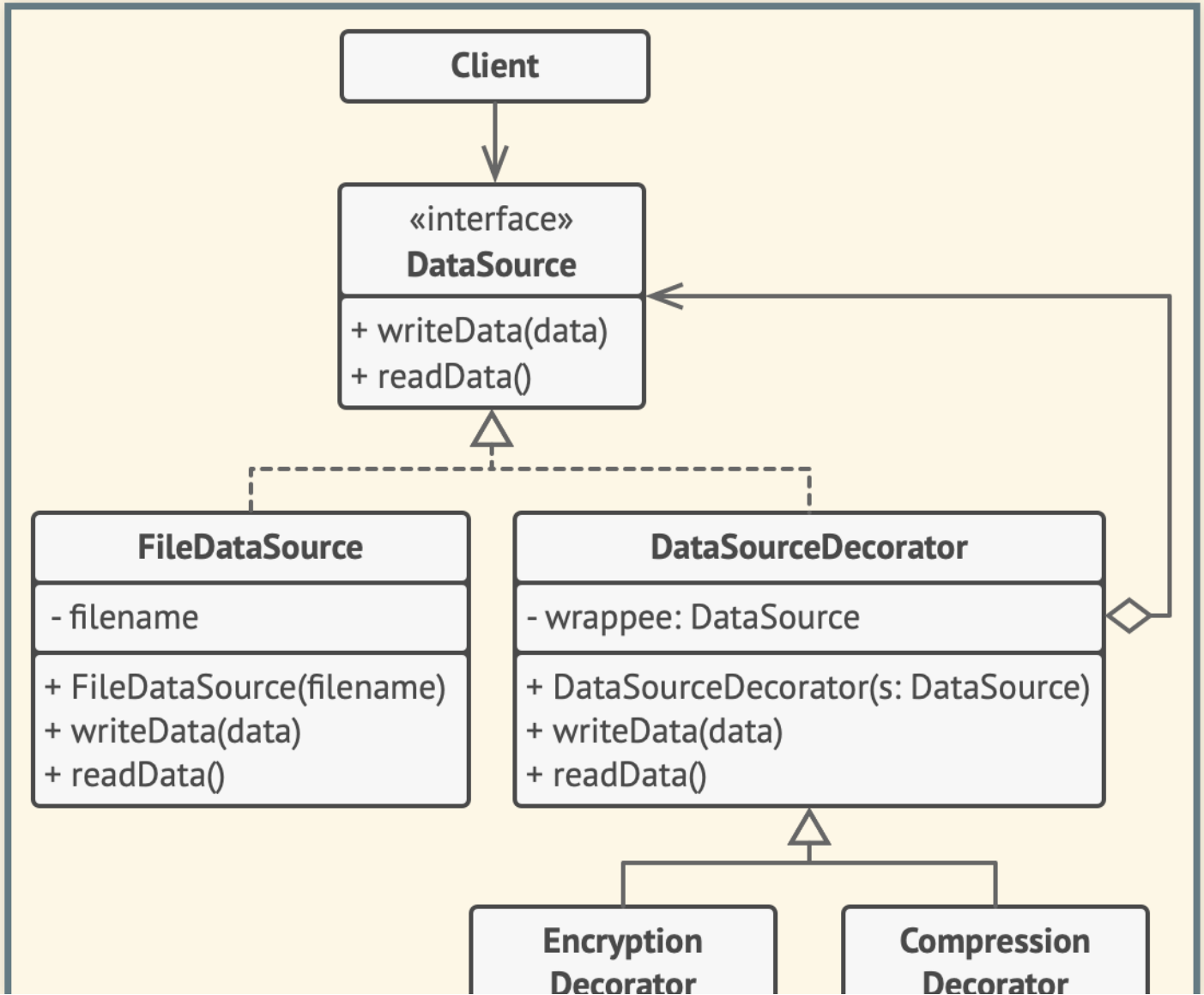
DECORATOR PATTERN

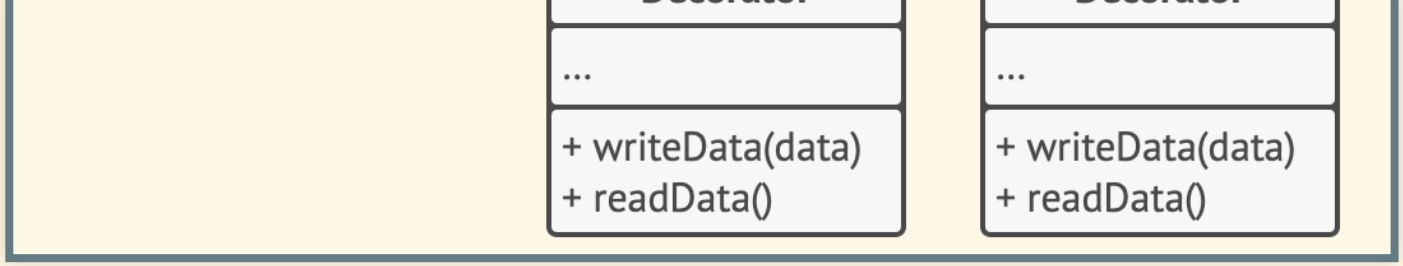
Decorator is a structural design pattern that lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.

DECORATOR PATTERN

- When you want to attach extra behavior / state to objects at runtime without breaking the code that uses it
- When it's not possible to extends an objects behavior using inheritance

DECORATOR PATTERN




...
+ writeData(data)
+ readData()
...
+ writeData(data)
+ readData()

DECORATOR PATTERN

```
public interface DataSource {
    void writeData(String data);

    String readData();
}
```

DECORATOR PATTERN

```
public class FileDataSource implements DataSource {
    private String name;

    public FileDataSource(String name) {
        this.name = name;
    }

    @Override
    public void writeData(String data) { ... }

    @Override
    public String readData() { ... }
}
```

DECORATOR PATTERN

```
public class DataSourceDecorator implements DataSource {
    private DataSource wrappee;

    DataSourceDecorator(DataSource source) {
        this.wrappee = source;
    }

    @Override
    public void writeData(String data) {
        wrappee.writeData(data);
    }

    @Override
    public String readData() {
        return wrappee.readData();
    }
```

DECORATOR PATTERN

```
public class EncryptionDecorator extends DataSourceDecorator {

    public EncryptionDecorator(DataSource source) {
        super(source);
    }

    @Override
    public void writeData(String data) {
        super.writeData(encode(data));
    }

    @Override
    public String readData() {
        return decode(super.readData());
    }
```

DECORATOR PATTERN

```
public class CompressionDecorator extends DataSourceDecorator {
    public CompressionDecorator(DataSource source) {
        super(source);
    }

    @Override
    public void writeData(String data) {
        super.writeData(compress(data));
    }

    @Override
    public String readData() {
        return decompress(super.readData());
    }

    private String compress(String stringData) {
```

DECORATOR PATTERN

```
DataSourceDecorator datasource = new CompressionDecorator(
    new EncryptionDecorator(
        new FileDataSource("out/OutputDemo.txt")
    )
);

String uncompressedDecryptedData = dataSource.readData();
```

FACTORY PATTERN

FACTORY PATTERN

"Factory Method is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created." - "https://refactoring.guru/design-patterns/factory-method"

FACTORY PATTERN

```
public interface Button {
    void render();
    void onClick();
}
```

FACTORY PATTERN

```
public class WebButton implements Button {
    public void render() {
        System.out.println("I'm a web button");
    }
    public void onClick() {
        System.out.println("You just clicked a web button");
    }
}
```

FACTORY PATTERN

```
public class WindowsButton implements Button {
    public void render() {
        System.out.println("I'm a windows button");
    }
    public void onClick() {
        System.out.println("You just clicked a windows button");
    }
}
```

FACTORY PATTERN

```
public abstract class Dialog {
    abstract Button createButton();

    void renderDialog() {
        System.out.println("Rendering dialog");
        System.out.println("Rendering button: ");
        createButton().render();
    }
}
```

FACTORY PATTERN

```
public class WindowsDialog extends Dialog {
    Button createButton() {
        return new WindowsButton();
    }
}
```

FACTORY PATTERN

```
public class WebDialog extends Dialog {
    Button createButton() {
        return new WebButton();
    }
}
```

FACTORY PATTERN

```
public static void main(String[] args) {
    Dialog dialog = new WebDialog();

    dialog.renderDialog();
}
```