Design Patterns cheat sheet

Contents

Unified Modelling Language

➔ General purpose way to visualize the design of a software system.

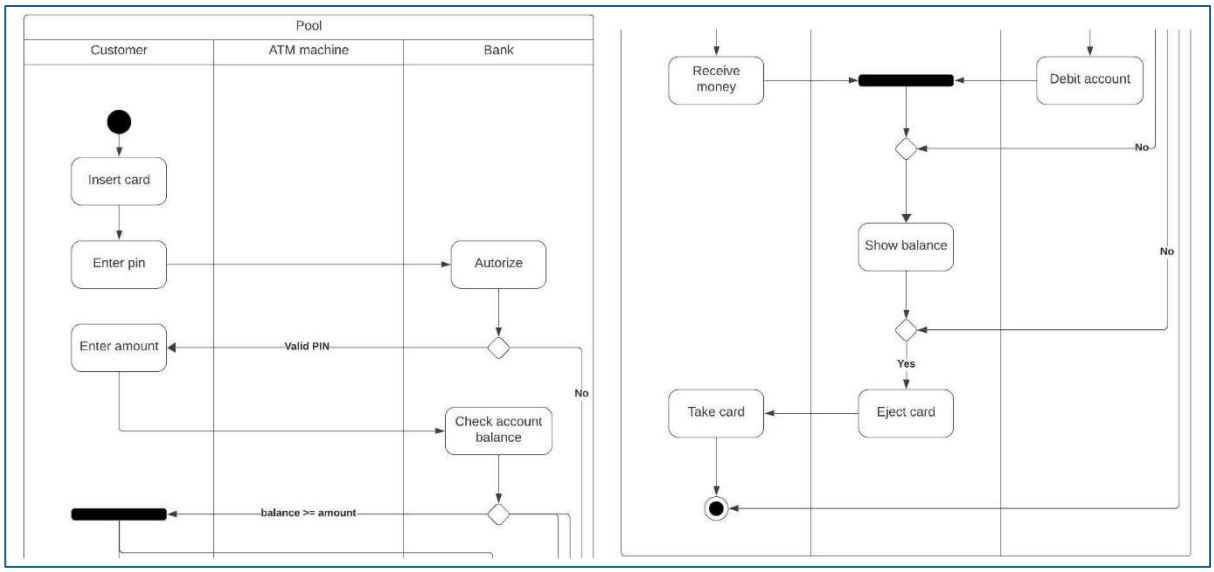
Use Case

➔ Textual (specifications) and visual (diagram) description of an ACTOR interacting with a system.

➔ FURPS

Activity Diagrams

➔ Business logic, building upon a USE CASE



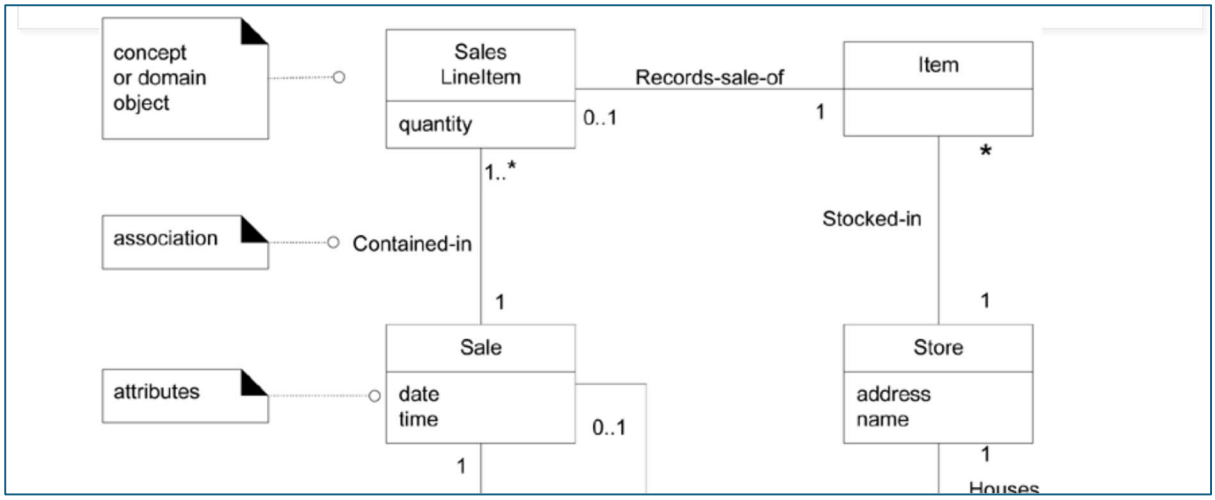
Domain Models

➔ NOT CLASS DIAGRAM but CONCEPTUAL presentation

➔ Find Classes

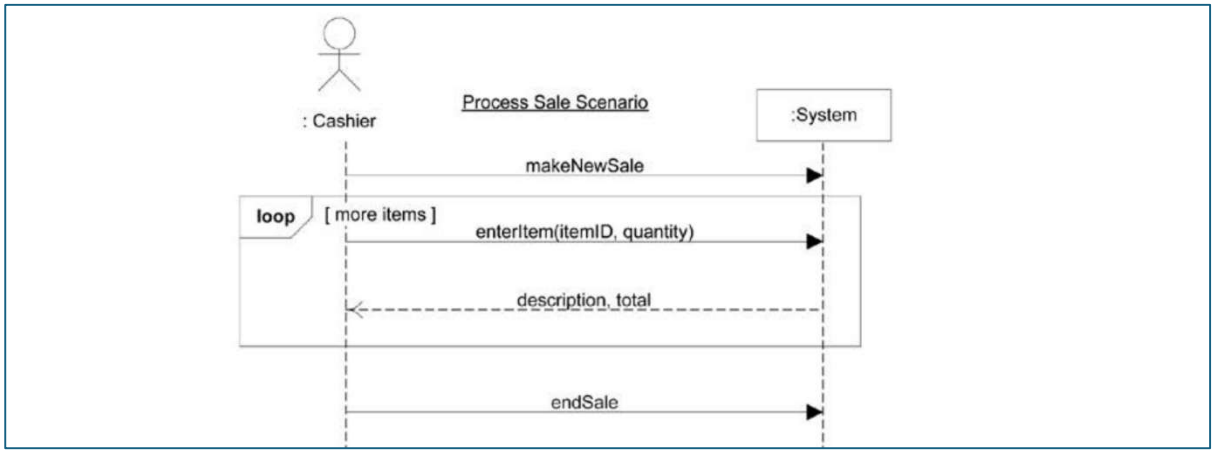
+ Draw in UML diagram

+ Add attributes and associations.



System Sequence Diagrams

- ➔ USE CASE EVENTS
  - External ACTOR input
  - Internal timing events
  - Exceptions
- ➔ Main or alternative, complex scenarios



STATIC

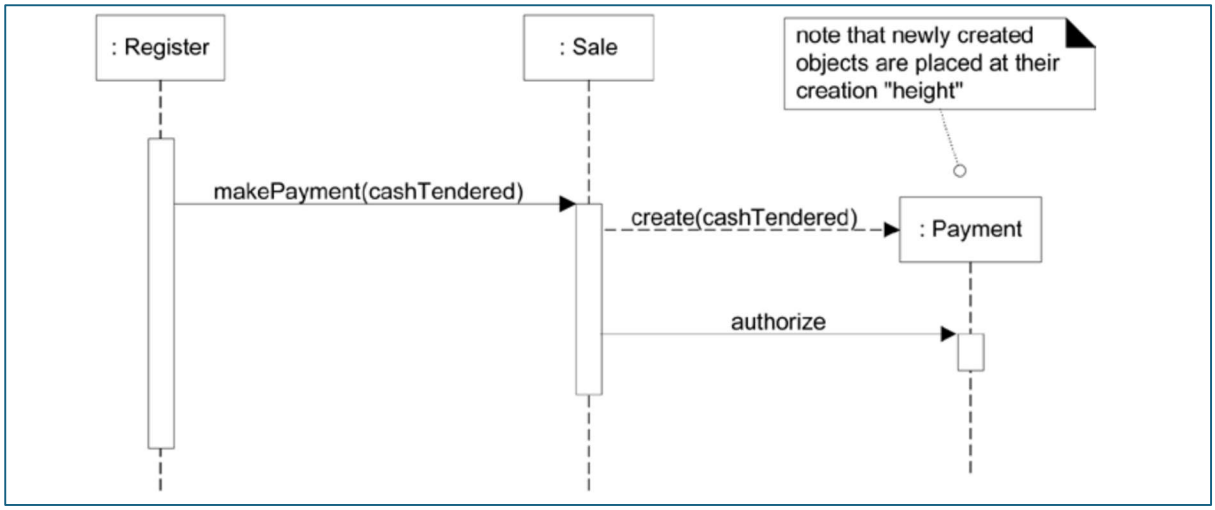
- ➔ CLASS DIAGRAM

DYNAMIC

- ➔ SEQUENCE DIAGRAM
- ➔ COMMUNICATION DIAGRAM

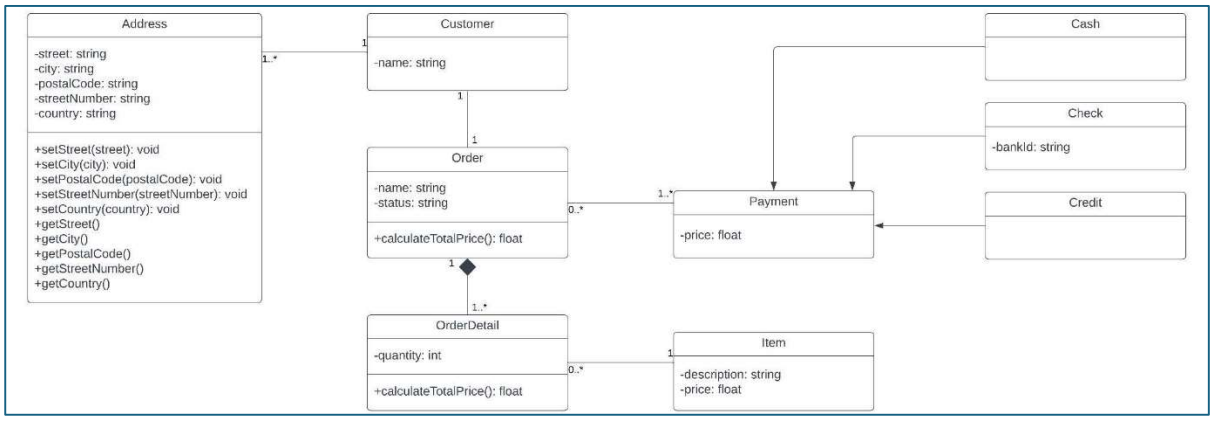
Interaction Sequence Diagram

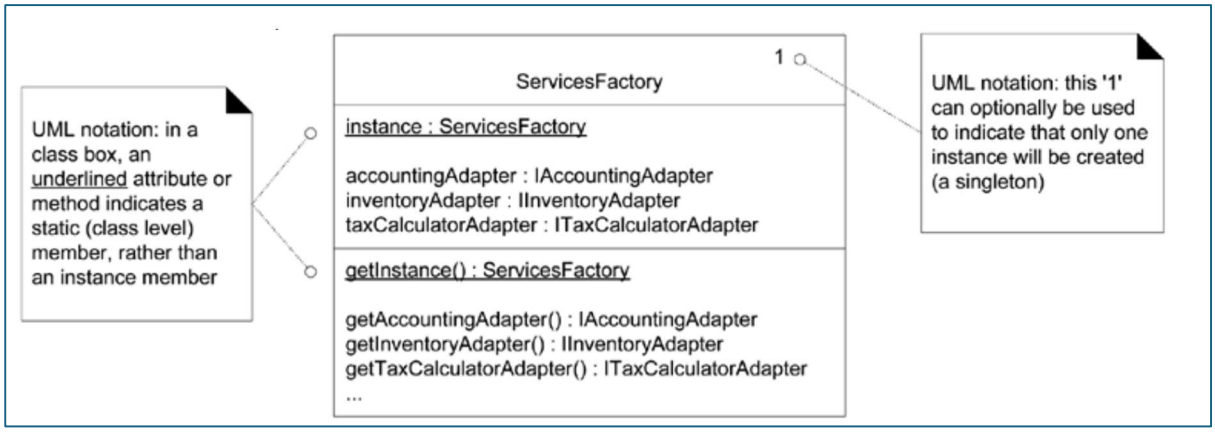
- ➔ Presentation OBJECT interactions
- ➔ `Return = message (parameter:parameterType):returnType`



Class Diagram

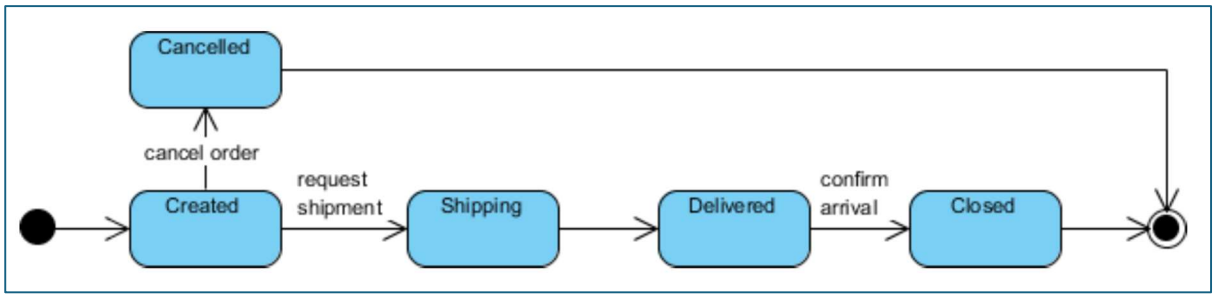
- ➔ Static presentation of CLASSES, INTERFACES, and their associations.
- ➔ + PUBLIC - PRIVATE # PROTECTED ~ PACKAGE



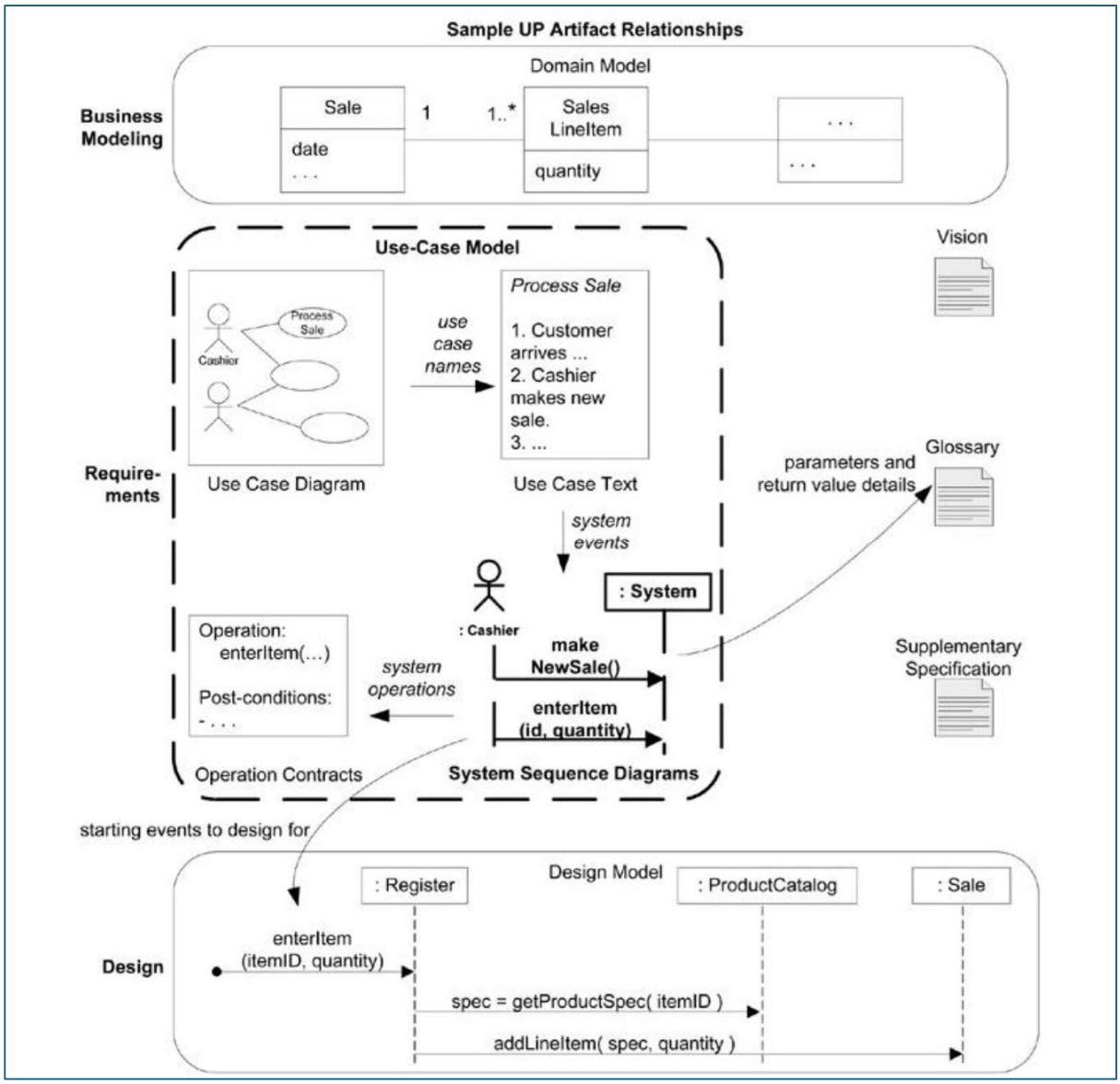


State Machine Diagram

➔ Object EVENT STATUS and BEHAVIOUR



Summary



Why Design Patterns

- ➔ POWERFULL, EFFICIENT, and creates DISCUSSION ON A DESIGN LEVEL
- ➔ GRASP General Responsibility Assignment Software Patterns
- ➔ GoF Design Pattern
- ➔ RDD Responsibility Driven Design

Relationship between RDD, GRASP and UML?

- ➔ RESPONSIBILITIES of objects during coding/modelling.
- ➔ UML diagram helps us think about the RESPONSIBILITIES.
- ➔ GRASP helps us assign RESPONSIBILITIES to OBJECTS.
- ➔ GoF PATTERNS are more advanced patterns.

GRASP Patterns

1. **CREATOR**
    i. Assign the responsibility to the class that has the information needed to create an instance of another class.
2. **INFORMATION EXPERT**
    i. Entrust the responsibility to the class with the necessary information to fulfil it most effectively.
3. **LOW COUPELING**
    i. Minimize dependencies between classes, promoting a design where changes in one class have minimal impact on others.
4. **CONTROLLER**
    i. Assign the responsibility for handling system events and coordinating activities to a controller class.
5. **HIGH COHESION**
    i. Group responsibilities to create classes that are focused and perform a specific, well-defined set of tasks.
6. **POLYMORPHISM**
    i. Design classes so that they can be used interchangeably through a common interface, allowing flexibility and extensibility.
7. **PURE FABRICATION**
    i. Introduce a class solely to achieve design goals, even if it doesn't represent a concept in the problem domain.
8. **INDIRECTION**
    i. Introduce an intermediary to decouple components and reduce direct dependencies.
9. **PROTECTED VARIATIONS**
    i. Shield elements from variations in other elements by encapsulating the volatile concepts and providing stable interfaces.

Design Patterns

1. **STRATEGY**
    i. Define a family of algorithms, encapsulate each one, and make them interchangeable, allowing the client to choose and switch between algorithms at runtime.
2. **OBSERVER**
    i. Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
3. **SINGELETON**
    i. Ensure a class has only one instance and provide a global point of access to it.
4. **COMMAND**
    i. Encapsulate a request as an object, allowing clients to parameterize clients with queues, requests, and operations, and supporting operations such as undo and redo.
5. **TEMPLATE METHOD**
    i. Define the skeleton of an algorithm in the superclass but let subclasses override specific steps of the algorithm without changing its structure.
6. **COMPOSITE**
    i. Compose objects into tree structures to represent part-whole hierarchies, allowing clients to treat individual objects and compositions of objects uniformly in a unified manner.
        a. *COMPONENT*
            i. Declare the interface for objects in the composition, allowing both leaves and composites to be treated uniformly.
        b. *LEAF*
            i. Represent the building blocks for compositions and implement the Component interface.
        c. *COMPOSITE*
            i. Compose objects into tree structures to represent part-whole hierarchies, allowing clients to treat individual objects and compositions of objects uniformly.
        d. *CLIENT*
            i. Manipulate objects in the composition through the Component interface.
7. **ADAPTER**
    i. Convert the interface of a class into another interface clients expect, allowing incompatible interfaces to work together.
8. **FAÇADE**
    i. Provide a unified interface to a set of interfaces in a subsystem, making it easier to use and reducing dependencies.
9. **DECORATOR**
    i. Attach additional responsibilities to an object dynamically, providing a flexible alternative to subclassing for extending functionality.

Information Expert

Let's consider an example in the context of an online shopping system where we want to calculate the total price of an order:

```
Class: Order
Responsibility: Represent an order placed by a customer.

Class: Product
Responsibility: Store information about a product, including its name,
price, and quantity available.

Class: ShoppingCart
Responsibility: Maintain a collection of Product objects representing
items added to the shopping cart.

Class: OrderProcessor
Responsibility: Calculate the total price for an order based on the
information stored in the ShoppingCart.
```

- The ***Information Expert*** principle would suggest assigning the responsibility of calculating the total order price to the ***OrderProcessor*** class because it has access to the necessary information stored in the ***ShoppingCart*** (which contains ***Product*** objects).
- Each ***Product*** object within the ***ShoppingCart*** class stores information about a specific item, including its price.

By adhering to the Information Expert pattern, we ensure that the class responsible for calculating the total order price is the one with direct access to the relevant information, promoting a design that is more modular and maintains a clear separation of concerns.

High Cohesion

Consider a multimedia editing software application, and let's look at the design of classes responsible for handling different types of multimedia content:

```
Class: ImageEditor
Responsibilities: Crop image, adjust brightness, apply filters.

Class: AudioEditor
Responsibilities: Trim audio, adjust volume, apply audio effects.

Class: VideoEditor
Responsibilities: Cut video, add transitions, adjust playback speed.

Class: MultimediaProject
Responsibilities: Manage a collection of multimedia content, coordinate
editing operations.

Class: FileIOHandler
Responsibilities: Read and write multimedia files, handle file formats.

Class: UserInterface
Responsibilities: Handle user input, display output, interact with
editors.

Class: MultimediaLibrary
Responsibilities: Store and organize multimedia assets for easy retrieval.

Class: ProjectExporter
Responsibilities: Export a multimedia project to a specific file format.
```

- The ***High Cohesion*** principle suggests that each class should have a focused and well-defined set of tasks. For instance, the ***ImageEditor*** class is focused on image-related operations, the ***AudioEditor*** on audio-related tasks, and so on.
- Each class has a clear and specific responsibility, promoting a high level of cohesion within each class. For instance, the ***FileIOHandler*** class is responsible for file input/output operations, ensuring that each class has a distinct role.

By organizing classes in this way, the design achieves high cohesion, making each class focused on a specific aspect of multimedia editing, which contributes to better maintainability and readability of the codebase.

Creator Pattern

Consider a scenario where you have a system for managing a library, and you want to create instances of a Book class:

```
Class: Book
Responsibility: Represent a book, store information such as title, author, and publication date.

Class: Library
Responsibility: Manage a collection of books.

Class: BookFactory
Responsibility: Create instances of the Book class based on provided information.
```

- The ***Creator principle*** suggests that the responsibility of creating instances of the ***Book*** class should be assigned to the class that has the necessary information. In this case, the ***BookFactory*** class is responsible for creating instances of the ***Book*** class.
- The ***BookFactory*** class would have methods like ***createBook*** that take parameters such as title, author, and publication date, and use this information to instantiate and return a new ***Book*** object.

By adhering to the Creator pattern, you ensure that the responsibility of creating instances of a class is delegated to a specialized class (BookFactory), promoting a more modular and maintainable design.

Low Coupling

Consider a billing system where you want to minimize dependencies between classes:

```
Class: Invoice
Responsibility: Generate invoices for customer orders.

Class: Customer
Responsibility: Store customer information.

Class: Product
Responsibility: Store information about products.

Class: BillingService
Responsibility: Calculate the total amount for an order and generate an
invoice.

Class: InventoryManager
Responsibility: Manage product inventory.
```

- The ***Low Coupling*** principle suggests minimizing dependencies between classes. For instance, the ***BillingService*** class calculates the total amount for an order and generates an invoice, but it does not directly depend on the internal details of the ***Customer*** or ***Product*** classes.
- If there were high coupling, changes in the ***Customer*** or ***Product*** classes might necessitate modifications in the ***BillingService*** class. However, adhering to low coupling means that changes in one class (e.g., modifications to the ***Customer*** class) have minimal impact on other classes (e.g., the ***BillingService*** class).

By minimizing dependencies, changes in one part of the system are less likely to cause ripple effects throughout the entire codebase, making the system more flexible and easier to maintain.

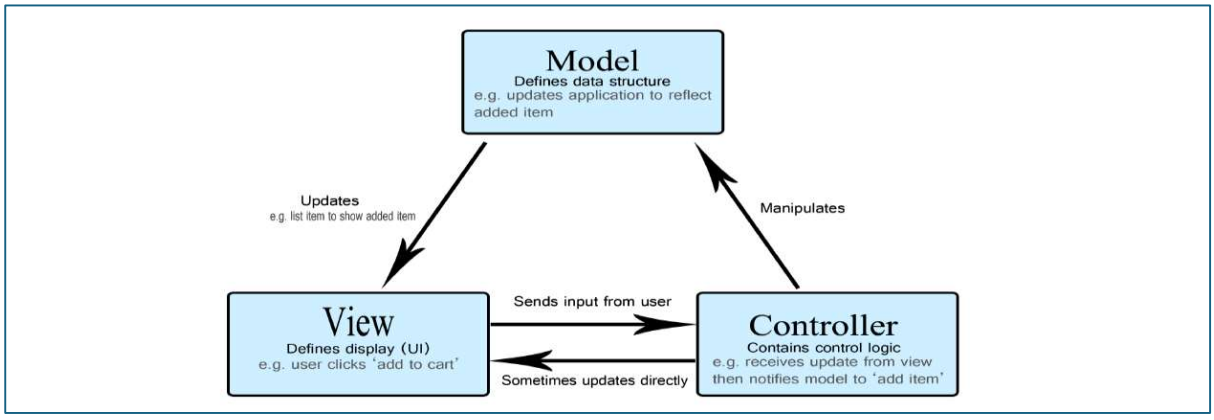
Controller

Consider a web application where you want to handle user interactions and coordinate activities:

```
Class: UserController
Responsibility: Handle user-related events, such as user registration and login.

Class: OrderController
Responsibility: Manage the processing of customer orders.

Class: ProductController
Responsibility: Handle product-related events, such as adding new products or updating existing ones.

Class: ShoppingCartController
Responsibility: Coordinate activities related to the shopping cart, such as adding or removing items.

Class: FrontendDispatcher
Responsibility: Route incoming requests to the appropriate controller.

Class: DatabaseManager
Responsibility: Manage interactions with the database.
```

- The ***Controller*** pattern suggests assigning the responsibility for handling system events and coordinating activities to specific controller classes. For instance, the ***UserController*** class is responsible for handling user-related events.
- Each controller class is focused on a specific aspect of the system and coordinates activities related to that aspect.
- The ***FrontendDispatcher*** class routes incoming requests to the appropriate controller, ensuring that each controller is responsible for a distinct set of functionalities.

By using the Controller pattern, you achieve a modular and organized design where different aspects of the system are handled by dedicated controller classes, promoting a separation of concerns and making the system easier to understand and maintain.



Indirection

Consider a scenario where you want to reduce direct dependencies between components in a messaging system:

```
Class: MessageSender
Responsibility: Send messages to a message queue.

Class: MessageQueue
Responsibility: Manage the queue of messages.

Class: MessageProcessor
Responsibility: Process messages from the queue.

Class: EmailService
Responsibility: Send email notifications.

Class: NotificationHandler
Responsibility: Handle notifications, coordinating between
MessageProcessor and EmailService.
```

- The ***Indirection*** principle suggests introducing an intermediary to decouple components. Here, the ***NotificationHandler*** class serves as an intermediary between the ***MessageProcessor*** and ***EmailService***.
- Instead of having the ***MessageProcessor*** class directly communicate with the ***EmailService***, the ***NotificationHandler*** class is introduced to coordinate and manage the flow of notifications.
- This indirection reduces direct dependencies between the ***MessageProcessor*** and ***EmailService***, making the system more flexible and allowing changes to one component without affecting the other.

By introducing an intermediary, you promote a design that minimizes direct dependencies, making the system more modular and adaptable to changes in individual components.

Polymorphism

Consider a scenario where you want to implement different payment methods in an e-commerce system:

```
Interface: PaymentMethod
Methods: processPayment(), refundPayment()

Class: CreditCardPayment implements PaymentMethod
Responsibility: Implement payment processing and refund for credit card payments.

Class: PayPalPayment implements PaymentMethod
Responsibility: Implement payment processing and refund for PayPal payments.

Class: PaymentProcessor
Responsibility: Process payments using a provided PaymentMethod.
```

- The ***Polymorphism*** principle suggests designing classes so that they can be used interchangeably through a common interface (***PaymentMethod***).
- Both ***CreditCardPayment*** and ***PayPalPayment*** classes implement the ***PaymentMethod*** interface, providing a common set of methods (***processPayment()*** and ***refundPayment()***).
- The ***PaymentProcessor*** class can then accept any class implementing the ***PaymentMethod*** interface, allowing flexibility and extensibility. For example, you can easily add new payment methods without modifying the ***PaymentProcessor*** class.

By utilizing polymorphism, you create a system where different payment methods can be seamlessly integrated, and the client code (in this case, the ***PaymentProcessor***) can work with any class adhering to the common interface, promoting flexibility and ease of extension.

Protected Variations

Consider a scenario where you want to shield a payment processing system from changes in external payment gateways:

```
Interface: PaymentGateway
Methods: processPayment(), refundPayment()

Class: VisaPaymentGateway implements PaymentGateway
Responsibility: Implement payment processing and refund for Visa payments.

Class: MasterCardPaymentGateway implements PaymentGateway
Responsibility: Implement payment processing and refund for MasterCard
payments.

Class: PaymentProcessor
Responsibility: Process payments using a provided PaymentGateway.
```

- The ***Protected Variations*** principle suggests shielding elements from variations in other elements by encapsulating the volatile concepts (payment gateways) and providing stable interfaces (the ***PaymentGateway*** interface).
- Both ***VisaPaymentGateway*** and ***MasterCardPaymentGateway*** classes implement the ***PaymentGateway*** interface, which defines common methods (***processPayment()*** and ***refundPayment()***).
- The ***PaymentProcessor*** class depends on the ***PaymentGateway*** interface, not on specific implementations. This shields the payment processing system from variations in individual payment gateways.

By applying the Protected Variations principle, you create a design that protects against changes in external elements (payment gateways) by providing stable interfaces. This encapsulation allows for easier adaptation to new payment gateways or changes in existing ones without affecting the core payment processing logic.

Pure Fabrication

Consider a scenario where you want to implement logging functionality in a system:

```
Class: Logger
Responsibility: Provide logging capabilities for the application.

Class: OrderProcessor
Responsibility: Process customer orders and utilize the Logger class for
logging.
```

- The ***Pure Fabrication*** principle suggests introducing a class solely to achieve design goals. In this case, the ***Logger*** class is introduced to handle logging functionality even though it may not represent a direct concept in the problem domain.
- The ***Logger*** class is a pure fabrication because it's not a real-world entity like a customer or an order, but it is introduced to achieve a specific design goal (separating concerns related to logging).
- The ***OrderProcessor*** class can utilize the ***Logger*** class to log events without directly incorporating logging functionality into its own responsibilities, promoting a more modular design.

By introducing a pure fabrication like the ***Logger*** class, you enhance the separation of concerns and maintainability, ensuring that classes focus on their core responsibilities while achieving specific design goals like logging through dedicated components.

Strategy

Consider a sorting algorithm strategy:

```
Interface: SortingStrategy
Methods: performSort(List<int> data)

Class: BubbleSort implements SortingStrategy
Responsibility: Implement the bubble sort algorithm.

Class: QuickSort implements SortingStrategy
Responsibility: Implement the quicksort algorithm.

Class: SorterContext
Responsibility: Allow the client to choose and switch between sorting
algorithms at runtime using a SortingStrategy.
```

- The ***Strategy pattern*** is applied by defining a family of sorting algorithms encapsulated in the ***SortingStrategy*** interface.
- The ***BubbleSort*** and ***QuickSort*** classes encapsulate specific sorting algorithms.
- The ***SorterContext*** class allows the client to choose and switch between algorithms at runtime by utilizing a specific sorting strategy.

This pattern enables flexibility by allowing the client to select the desired sorting strategy dynamically without modifying the client's code.

Observer

Consider a weather monitoring system:

```
Interface: Observer
Methods: update(float temperature, float humidity, float pressure)

Class: WeatherStation
Responsibility: Track and update weather conditions.

Class: Display
Responsibility: Display weather information.

Class: SmartphoneApp
Responsibility: Display weather updates on a smartphone app.

Class: WeatherDataSubject
Responsibility: Maintain weather data and notify observers of changes.
```

- The ***Observer pattern*** is applied by defining an ***Observer*** interface with an ***update*** method.
- ***Display*** and ***SmartphoneApp*** classes act as observers that display weather information.
- ***WeatherDataSubject*** is the subject that maintains weather data and notifies observers (like ***Display*** and ***SmartphoneApp***) when weather conditions change.

When the ***WeatherDataSubject*** changes its state (e.g., new weather data), all dependent observers are automatically notified and updated, ensuring that the displayed information is always synchronized with the latest weather conditions.

Singleton

Consider a configuration manager in a software application:

```
Class: ConfigurationManager
Responsibility: Manage application configuration settings.
```

- The ***Singleton*** pattern is applied by ensuring that there is only one instance of the ***ConfigurationManager*** class.
- The ***ConfigurationManager*** class provides a global point of access to its instance.

This pattern ensures that there is a single, globally accessible instance of the ***ConfigurationManager***, preventing multiple instances with potentially conflicting configurations and providing a centralized point for managing application settings.

Command

Consider a remote control for a smart home system:

```
Interface: Command
Methods: execute()

Class: LightOnCommand implements Command
Responsibility: Encapsulate the command to turn on the lights.

Class: LightOffCommand implements Command
Responsibility: Encapsulate the command to turn off the lights.

Class: RemoteControl
Responsibility: Maintain a list of commands and execute them when
requested.
```

- The ***Command*** pattern is applied by defining a ***Command*** interface with an ***execute*** method.
- ***LightOnCommand*** and ***LightOffCommand*** classes encapsulate specific commands to turn the lights on and off.
- The ***RemoteControl*** class maintains a list of commands and can execute them, allowing clients to parameterize the remote control with various commands.

This pattern enables the decoupling of the sender (remote control) and the receiver (light), supporting operations like undo and redo by storing a history of executed commands.

Template Method

Consider a template method for preparing a hot beverage:

```
Abstract Class: HotBeverageTemplate
Methods: boilWater(), brew(), pourInCup(), addCondiments(),
prepareBeverage()

Class: Tea extends HotBeverageTemplate
Responsibility: Implement specific steps for brewing tea.

Class: Coffee extends HotBeverageTemplate
Responsibility: Implement specific steps for brewing coffee.
```

- The ***Template*** Method pattern is applied with the ***HotBeverageTemplate*** abstract class defining a template method ***prepareBeverage***.
- The subclasses ***Tea*** and ***Coffee*** override specific steps (***brew***, ***addCondiments***) without changing the overall structure of the algorithm.
- The template method provides a skeleton for the algorithm (preparing a hot beverage) but allows subclasses to customize certain steps.

This pattern promotes code reuse by providing a common structure for related algorithms while allowing flexibility for subclasses to implement their variations.

Composite

Consider a graphic design application with shapes:

```
Interface: ShapeComponent
Methods: draw()

Class: Circle implements ShapeComponent
Responsibility: Represent a circle shape.

Class: Square implements ShapeComponent
Responsibility: Represent a square shape.

Class: CompositeShape implements ShapeComponent
Responsibility: Compose multiple shapes into a single composite shape.

Class: DrawingClient
Responsibility: Manipulate individual shapes and composite shapes
uniformly.
```

- The ***Composite*** pattern is applied by composing objects (***Circle***, ***Square***, ***CompositeShape***) into a tree structure representing part-whole hierarchies.
- The ***ShapeComponent*** interface (a) declares the common interface for all shapes, allowing both leaves and composites to be treated uniformly.
- ***Circle*** and ***Square*** classes (b) represent the building blocks (leaves) for compositions and implement the ***ShapeComponent*** interface.
- ***CompositeShape*** class (c) composes objects into a tree structure, allowing clients to treat individual objects and compositions of objects uniformly.
- The ***DrawingClient*** class (d) manipulates objects in the composition through the ***ShapeComponent*** interface, treating both individual shapes and composite shapes uniformly.

Adapter

Consider a scenario where you want to use a European electrical appliance (with a European plug) in a country with different power outlets (using a different type of plug):

```
Interface: EuropeanPlug
Method: plugIntoEuropeanOutlet()

Class: EuropeanDevice implements EuropeanPlug
Responsibility: Implement the European plug interface.

Class: USPlugAdapter implements EuropeanPlug
Responsibility: Adapt the European plug to fit into a US power outlet.
```

- The ***Adapter*** pattern is applied with the ***USPlugAdapter*** class, which converts the interface of the European plug (***EuropeanPlug***) into another interface (***USPlugAdapter***) that clients in the US expect.
- The ***EuropeanDevice*** class represents a European electrical appliance with a European plug.
- The ***USPlugAdapter*** class allows the European device to work with a US power outlet by adapting the European plug to the US interface.

This pattern enables the use of incompatible interfaces, promoting interoperability between systems with different specifications.

Façade

Consider a multimedia system with various components:

```
Class: AudioPlayer
Responsibility: Manage audio playback.

Class: VideoPlayer
Responsibility: Manage video playback.

Class: Display
Responsibility: Handle display configurations.

Class: MultimediaFacade
Responsibility: Provide a unified interface to control audio, video, and
display components.
```

- The ***Façade*** pattern is applied by the ***MultimediaFacade*** class, which provides a unified interface to a set of interfaces in the multimedia subsystem.
- The ***AudioPlayer***, ***VideoPlayer***, and ***Display*** classes represent individual components of the multimedia system.
- The ***MultimediaFacade*** simplifies the usage of the multimedia subsystem by offering a single interface that clients can interact with, reducing dependencies and making it easier to control audio, video, and display components.

This pattern hides the complexities of the subsystem behind a simplified interface, making it more convenient for clients to interact with the system without dealing with the intricacies of each individual component.

Decorator

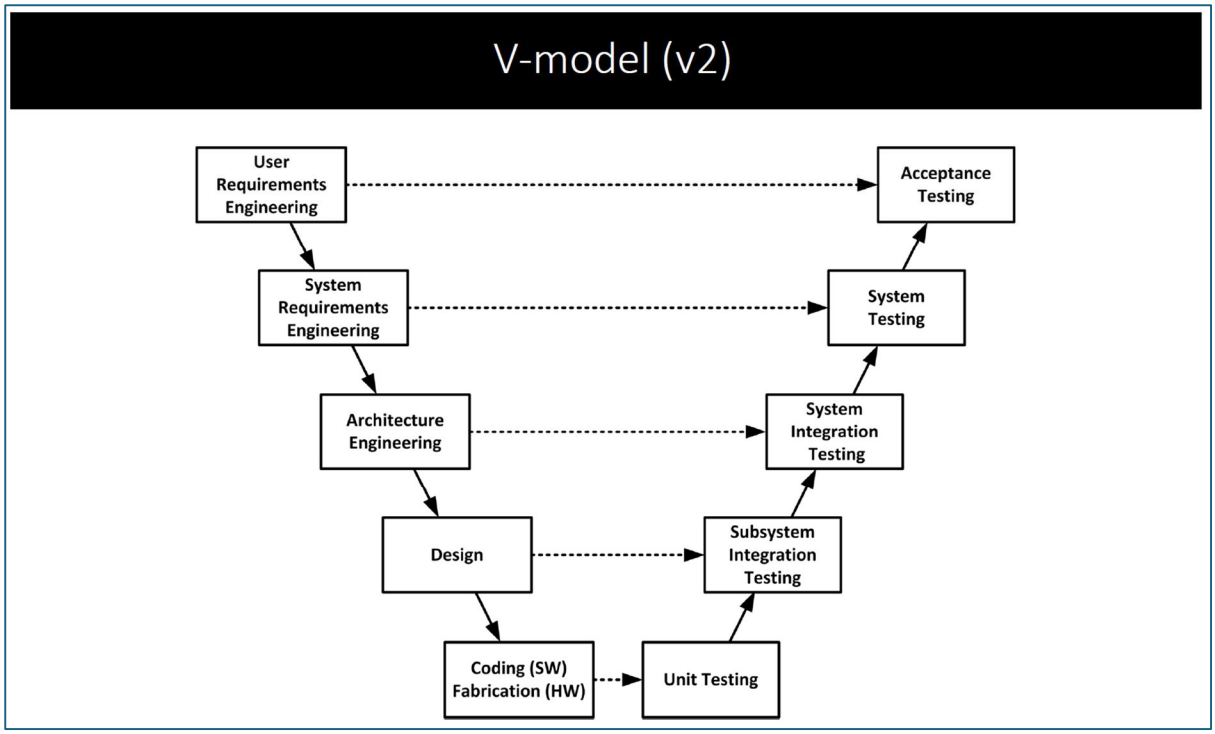
Consider a text formatting system:

```
Interface: TextComponent
Methods: format()

Class: ConcreteText implements TextComponent
Responsibility: Implement basic text formatting.

Class: BoldDecorator implements TextComponent
Responsibility: Attach additional responsibility for bold formatting.

Class: ItalicDecorator implements TextComponent
Responsibility: Attach additional responsibility for italic formatting.
```

- The ***Decorator*** pattern is applied by the ***BoldDecorator*** and ***ItalicDecorator*** classes, which attach additional responsibilities to a ***ConcreteText*** object dynamically.
- The ***TextComponent*** interface declares the common interface for text components.
- The ***ConcreteText*** class implements basic text formatting.
- The ***BoldDecorator*** and ***ItalicDecorator*** classes extend the functionality of the ***ConcreteText*** class dynamically by attaching additional formatting responsibilities.

This pattern provides a flexible alternative to subclassing, allowing new features to be added to an object dynamically at runtime without altering its structure.

Testing



MODULE TESTING (Component, Unit, or Program Testing):

- Testing individual units or modules of a software application.
- Tested in isolation to ensure the functionality of each component.
- Utilizes a unit test framework.
- Purpose: Verify the correctness of each module, catch and fix bugs early, and ensure modules meet specifications.

INTEGRATION TESTING:

- Testing the combination and interaction of multiple modules or components.
- Aimed at identifying issues in the interfaces between modules.
- Ensures that integrated components work together as expected.
- Types: Top-down, Bottom-up, and Incremental integration testing.

SYSTEM TESTING:

- Testing the entire software system as a whole.
- Focus on validating the system's compliance with specified requirements.
- Includes functional and non-functional testing.
- Tests system behaviour in different environments and usage scenarios.

ACCEPTANCE TESTING:

- Evaluates whether the software meets customer or business requirements.
- Validates that the system is ready for deployment and use in a real-world environment.
- Types: User Acceptance Testing (UAT) involves end-users, and Alpha/Beta testing may be conducted externally.
- Final stage before the software is released.

Classification of testing techniques

Blackbox Tests (Specification-Based)

- Focus is on testing functionality without knowledge of internal implementation details.
- Test cases are derived from the specifications of the software.
- Goal is to test the software based on input and expected output.
- Examples include functional tests, non-functional tests, and acceptance tests.

Equivalence Partitioning

- ➔ Equivalence Partitioning is a testing technique that divides the input space into classes to reduce the number of test cases while maintaining coverage.
- ➔ Test one or a few values from each class.
- ➔ Focus on different behaviour classes, ensuring coverage without testing every individual value.
- ➔ Benefits:
    - Reduces test cases, saving time and resources.
    - Systematic approach to cover diverse input scenarios.

Boundary Value Analysis

- ➔ Testing technique that evaluates the behaviour of a system at the boundaries of input ranges.
- ➔ Efficient way to catch errors that may occur at the boundaries.
- ➔ Useful for uncovering issues related to off-by-one errors, rounding, or other boundary-specific behaviours.
- ➔ Examples:
    - Full local disk/network disk
    - Nearly full local disk/network disk
    - Local disk/network disk with write protection
    - Damaged local disk/network disk
    - Unformatted local disk/network disk
    - Remove the local disk/network disk after the file is opened
    - Timeout while waiting for the local disk/network disk to come back online
    - Keyboard or mouse I/O while saving to the local disk/network disk
    - Other interrupts while writing to the local disk/network disk
    - Power outage during writing to the local disk
    - Local power outage during writing to the network disk
    - Network power outage during writing to the network disk

Whitebox Tests (Structure-Based):

- Operates based on the internal structure of the software.
- Testers possess knowledge of the internal logic, structures, and implementation details.
- Aim is to validate internal logic, check code validity, and test paths.
- Examples include statement coverage, branch coverage, and path coverage.

Code Coverage

➔ Code coverage measures the percentage of code that has been executed during testing.

Statement Coverage:

- Measures which statements in the source code have been executed at least once during testing.
- Indicates the extent to which each line of code has been tested.

Decision Coverage:

- Ensures that each decision point in the code has been exercised.
- Decision points are locations where the program's control flow can take different paths.

Loop Coverage:

- Aims to test the different possible scenarios within loops.
- Ensures that loops are executed with varying conditions, including zero iterations, one iteration, and multiple iterations.

Path Coverage:

- The most comprehensive form of coverage.
- Ensures that every possible path through the code, including all decision points and loops, has been tested.
- Provides a thorough assessment of the code's logical flow.

Experience-Based Tests

- Based on the knowledge, intuition, and experience of the testers.
- Test cases are developed based on the experience and insights of the testing team.
- Goal is to identify scenarios that may not be covered by other methods.
- Examples include exploratory testing and ad-hoc testing.

Error Guessing

➔ A testing technique where testers use their experience and intuition to identify potential errors or defects in the software.

Exploratory Testing

➔ An experience-based testing approach where testers simultaneously design and execute tests based on their knowledge, intuition, and understanding of the system.

SMART Testcases

Specific:

- The test case should have a clear and specific objective. It should focus on a particular functionality, feature, or aspect of the software.

Measurable:

- There should be criteria to measure the success or failure of the test case. It should provide a quantifiable result that indicates whether the expected outcome has been achieved.

Achievable (or Attainable):

- The test case should be realistic and achievable within the given constraints, such as time, resources, and available information.

Relevant (or Realistic):

- The test case should be relevant to the overall testing goals and objectives. It should address meaningful aspects of the software that contribute to its quality.

Time-bound:

- There should be a clear timeframe or deadline associated with the execution of the test case. This helps in managing testing schedules and ensures timely completion.

Mocking

➔ Mocking is a technique that creates simulated objects to mimic the behaviour of real objects or components. It helps isolate code being tested by replacing actual dependencies with controlled substitutes.

➔ EasyMock is a Java library simplifying the creation and use of mock objects in unit testing. It enables developers to set expectations, simulate behaviour's, and verify interactions easily.

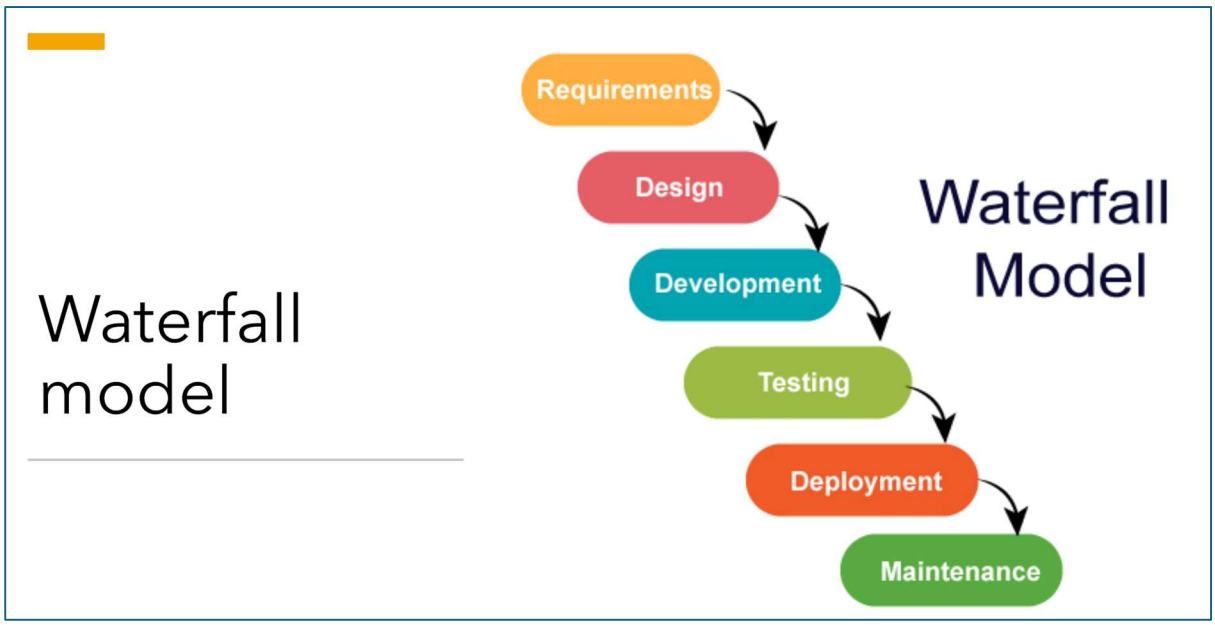
```
import static org.easymock.EasyMock.*;

// Create a mock object for an interface
MyInterface mockObject = createMock(MyInterface.class);

// Set expectations for method calls
expect(mockObject.someMethod()).andReturn("mockedResult");
expect(mockObject.anotherMethod(anyInt())).andReturn(42);

// Start the replay mode
replay(mockObject);

// Invoke the code being tested

// Verify that expected interactions occurred
verify(mockObject);
```

Software Development Methodologies

Waterfall Model

➔ The Waterfall Model is a linear and sequential software development approach organized into distinct phases, with progress flowing in one direction—like a waterfall.

The Waterfall Model's advantages lie in its **clear structure** and **documentation**, making it suitable for **projects with stable and well-defined requirements**. However, its **rigidity** and **late visibility** can be disadvantages, and it's best applied to **smaller projects** with **minimal anticipated changes**.



Agile

➔ Approach: Iterative, flexible, and collaborative.
➔ Advantages: Flexibility, customer satisfaction, early issue detection.
➔ Disadvantages: Uncertainty in the final product, resource intensity.
➔ Use When: Dynamic requirements, continuous customer involvement, small to medium-sized teams.

Agile is an adaptable, customer-centric approach suitable for projects with changing requirements and a need for ongoing collaboration.

1. Prioritized list of features. (Product Backlog)
2. Sprint planning
3. Teams collaborate on tasks within the sprint.
4. Deliver a shippable product increment.
5. Demonstrate work to stakeholders, gather feedback.
6. Reflect, identify improvements, adjust processes.
7. Repeat

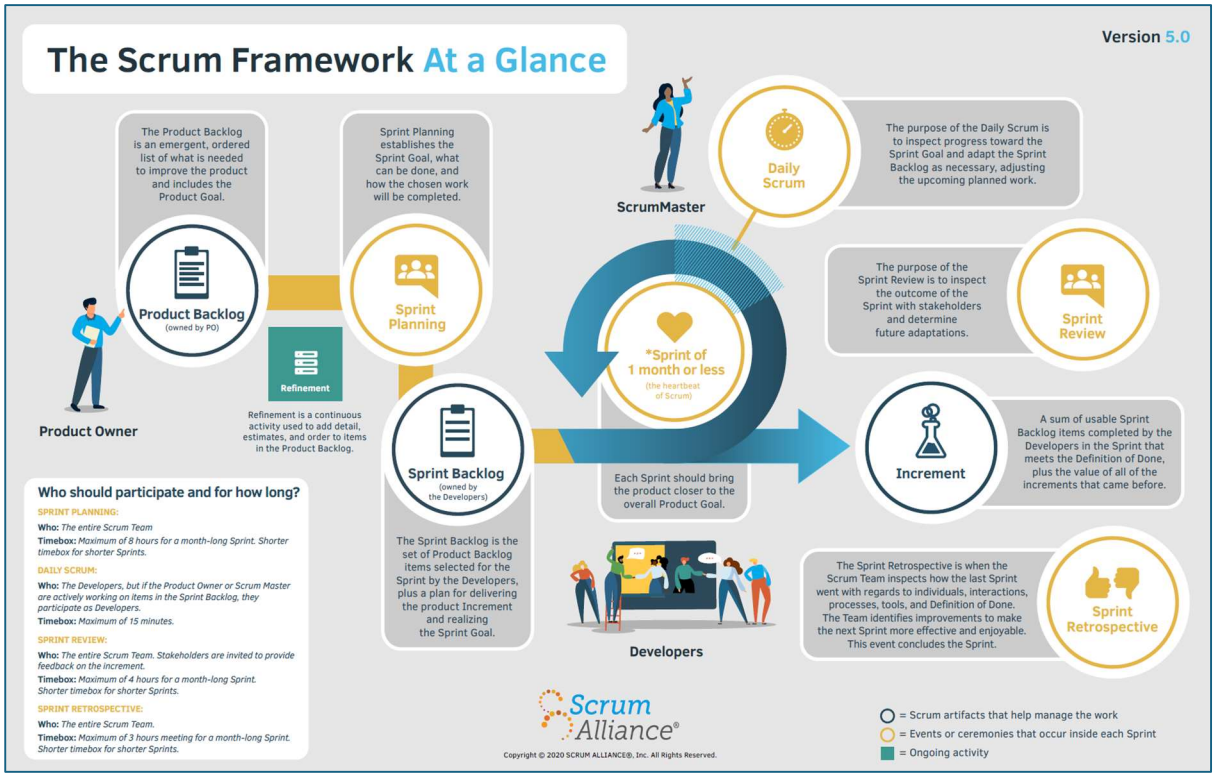
SCRUM

ROLES

- ➔ SCRUM master
- ➔ Development team
- ➔ Product owner

EVENTS

- ➔ Sprint
- ➔ Sprint Planning
- ➔ Daily Scrum
- ➔ Sprint Review
- ➔ Sprint Retrospective



7 Lean Development Principles

1. **Eliminate waste**
    i. Minimize any activity or resource that does not add value to the customer or the end product.
2. **Build Quality In**
    i. Integrate quality control measures throughout the development process rather than inspecting for defects afterward.
3. **Create Knowledge**
    i. Foster a culture of continuous learning and improvement, encouraging teams to acquire and share knowledge.
4. **Defer Commitment**
    i. Delay decision-making until the last responsible moment to accommodate evolving requirements and better-informed choices.
5. **Deliver Fast**
    i. Aim for quick and incremental delivery of valuable products or features to respond promptly to customer needs.
6. **Respect People**
    i. Value and empower individuals, recognizing their contributions and fostering a collaborative and respectful environment.
7. **Optimize the whole.**
    i. Focus on optimizing the entire value stream and the overall system rather than individual components or processes.

Minimum Viable Product

➔ Strategy where a basic version of a product is released with essential features to gather feedback and learn from early users, allowing for iterative improvements.

Example:

➔ With no money to build a business, the founders of Airbnb used their own apartment to validate their idea to create a market offering short-term, peer-to-peer rental housing online. They created a minimalist website, published photos and other details about their property, and found several paying guests almost immediately.

DevOps best practices

Continuous Integration:

- Automate the integration of code changes frequently, ensuring early detection of issues and maintaining a consistent codebase.

Continuous Delivery:

- Automate the entire software release process to enable the reliable and frequent delivery of updates to production.

Microservices:

- Architect applications as small, independent services to enhance scalability, maintainability, and deployment flexibility.

Infrastructure as Code:

- Manage and provision infrastructure through code to enhance automation, consistency, and version control.

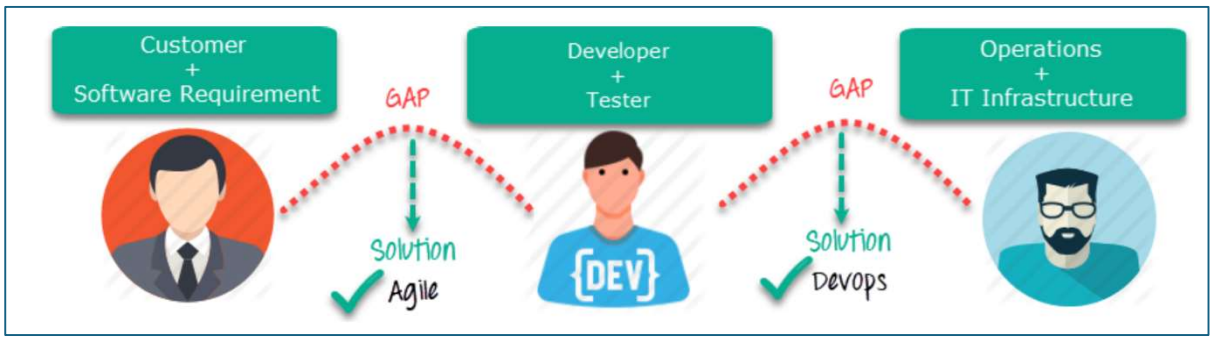
Monitoring and Logging:

- Implement robust monitoring and logging to gain insights into system performance, detect issues, and facilitate troubleshooting.

Communication and Collaboration:

- Foster open communication and collaboration among development, operations, and other stakeholders to streamline workflows and improve efficiency.

In the context of DevOps, these best practices promote agility, automation, and collaboration throughout the software development lifecycle.



Version Control

- ➔ History and Auditability:
  - Git maintains a complete history of changes, allowing you to track who made each change, when, and why. This audit trail enhances accountability and traceability.
- ➔ Collaboration and Teamwork:
  - Enables collaborative development by providing a centralized repository where team members can contribute, share, and merge code changes seamlessly.
- ➔ Parallel Development:
  - Facilitates concurrent work on different branches, allowing multiple developers to work on features or fixes simultaneously without conflicts.
- ➔ Branching and Merging:
  - Supports branching for parallel development efforts and easy merging of changes, aiding in the management of feature development, bug fixes, and releases.
- ➔ Code Stability:
  - Version control helps maintain a stable codebase by allowing you to revert to a previous state in case of issues, ensuring a reliable and working version is always available.
- ➔ Code Review:
  - Enables systematic code reviews, where team members can review proposed changes before merging them into the main codebase, promoting code quality and best practices.
- ➔ Continuous Integration and Delivery:
  - Integrates seamlessly with continuous integration systems, automating the process of building, testing, and deploying code changes, leading to faster and more reliable releases.
- ➔ Traceability and Documentation:
  - Provides a detailed history of changes, offering insights into the evolution of the codebase and helping with documentation and knowledge transfer.
- ➔ Risk Mitigation:
  - Reduces the risk of data loss or errors by maintaining a backup of the codebase and making it easier to identify and resolve issues.
- ➔ Facilitates Experimentation:
  - Encourages experimentation and innovation by providing a safe environment for trying out new ideas or features without affecting the main codebase.

Maven Build Tool

- ➔ Maven is a powerful build automation and project management tool that simplifies the build process, manages dependencies, and provides a consistent structure for Java projects, promoting best practices and efficient project development.