# CS3219 Cheatsheet

## Software Engineering Principles

### Principle of Modularity

- Shorter development time
- Better flexibility
- Better comprehensibility
- Decomposition of a big chunk into smaller well defined interfaces
- Way of managing the complexity

## Software Design Patterns

- Design Pattern: A solution to a problem in a context
- Tried and tested solutions to common problems in software design

## Creational

- Provide ways to instantiate single objects or groups of related objects.

### Builder Pattern

- Separate construction of a complex object from its representation so that the same construction process can create different representations
- Enforces a step-by-step process to construct a complex object as a finished product.
- Finished products can have different representations
- Good when designing classes where there are few compulsory parameters with optional / identical arguments for some

### Prototype pattern

- Create an object by cloning another object
- Do not create a new object for each client requesting it
- Create a single prototype and make copies of it for each of the client

## Structural Patterns

- Provide a manner to define relations between classes or objects

### Adapter Pattern

- Convert the interface of a class into another interface that the client expects
- Let classes work together that couldn't otherwise due to incompatible interfaces.
- Used when an incompatible module needs to be integrated with an existing module wihtout any source code modifications.

### Facade Pattern

- Provide a simple unified interface to a set of interfaces in a subsystem.
- Delegates client requests to appropriate subsystem classes.
- Implement subsystem functionality. Subsystems are used by the facade but not the other way around.

# Behavioural Patterns

- Define manners of communication between classes and objects

## Observer Pattern

- Let objects observe the behaviour of other objects so they can stay in sync
- A change to one object requires notifying others.
- Participants
    - Interface of abstract class defining the operations for attaching and de-attaching observers to the subject
    - Concrete subject class: Maintain the state of the object and when a change in the state occurs it notifies the attached observers
    - Observer: Interface of abstract class defining the operations to be used to notify this object

## Mediator Pattern

- Define an object that encapsulates how a set of objects interact to reduce coupling
- Delegates the routing of requests, messages, data that they exchange with other objects to the mediator.
- Allows loose coupling between sets of objects by handling the interactions between the objects
- Participants
    - Mediator Base: Abstract class that declares methods for communicating with Colleague objects
    - Concrete Mediator: Implements Mediator. Maintains and coordinates Colleague objects. Holds references to the colleague that it serves
    - Colleague Base: An abstract class for the colleague objects. Defines a single protected field that holds a reference to a mediator
    - Concrete Colleague: The concrete colleagues communicate with each other via the mediator. Send and receive methods are used to send messages to and receive messages from the mediator.

# Memento Pattern

- Without violating encapsulation, allow client to capture an object's state and restore
- Facilitates the current state of an object to be stored without breaking the rules of encapsulation
- The originating object can be modified as required but can be restored to the saved state at any time.
- Participants
    - Caretaker: Originator class to save and restore internal state
    - Originator: Defines operation for saving its current state and restoring to a previous state.
    - Memento: Stores an originator's internal state. Only originator that create the memento can access it

# State Pattern

- Allow an object to alter its behaviour when internal state changes

- Domain objects often have a concept of state
- Behaviour of domain object depends on its state
- Client requests mau change the state forcing the object to behave differently after being invoked
- Participants
    - Context
        - Provides an interface to client to perform some action
        - Delegates state specific requests to the Concrete State subclass that defines the current state
    - State: An interface that encapsulates the behaviour associated with a particular state of the Context
    - Concrete State: Concrete class that implements a behaviour associated with a state of the context.

## Strategy Pattern

- Represent a behaviour that parameterizes an algorithm for behaviour or performance
- Define family of algorithms, encapsulate each one, and make them interchangable.
- Let algorithm vary independently from the clients that use it.
- Participants
    - Strategy: An inteface common to all supported algorithm-specific classes.
    - Concrete Strategy: Implements the algorithm using the strategy interface.
    - Context
        - Provides the interface to client for encrypting data.
        - Maintains a reference to a strategy object and its instantiated and initialized by clients with concrete strategy object.

## Other Patterns

## Data Transfer Object

- Batch up multiple remote calls by encapsulating data that needs to travel from one application / subsystem to another
- Good for situations where there are multiple remote calls in a single call

## Software Architectures

### N tier Architecture

- Each layer is independent during development
- Each layer has a distinct and specific responsibility
- Example: Computer Networks

### Pipe and filter

- Data Enters the system and then flows through the each component 1 at a time until the data ends up at a data sink
- A series of transformations on successive pieces of input data
- Each component takes a set of inputs and returns a set of outputs

- Connector -> Transmit output of one pipe into input of another pipe
- Filter -> Transform the input and return to output

### Shared repository

- Maintains all data in a central repository shared by all functional components of the data-driven application
- Let availability, quality and state of the data trigger and coordinate the control flow of the application
- Components
    - A central data structure that represents the current state
    - A collection of independent components that operate on the central data store
- Connectors
    - Interactions between the repository and the other components
    - Varies between systems
- EG: Interpreter with Shared Symbol Table, IDEs

### Implicit Invocation / Event Driven Architecture

- An event announcement implicitly causes the invocation of procedures in other modules
- Components announce/ broadcast 1 or more events
- Other components register interest in evnet by accociating a function to that event
- When event is announced, it invoke all procedures which are registered
- Components are loosely coupled
- Components
    - Interface provides both a collection of procedures and a set of events
- Connectors
    - Procedure call
    - Bindings between events and procedures
- EG: Pub Sub Systems

### Hexagonal Architecture / Ports and Adapters

- 3 Blocks of code
    - User interface (UI)
    - Application Core / Business Logic
    - Infrastructure
- Adapters
    - Located outside the application core
    - Primary / Driving Adapters: Tell the application to do something
    - Secondary / Driven Adapters: Told by the application to do something
- Ports
    - An entry point to the application core
    - A specification of how to use the application core
    - Located within the application core
- Application Layer
    - Orgainising the application core
    - Several different processes that can be triggered in the application core by one or more user interface

    - Contains the interfaces + Ports
- Domain Layer
    - Contains the data and logic to manipulate data
    - Independent of the business logic that trigger it
- Domain Service
    - Handles logic that span multiple domain model objects
    - Receives a set of entities and performs business logic on them
- Domain Model
    - Contains the business objects that represents something in the domain
    - When an entity changes, a Domain Event is triggered and it carries the changed property values.
    - Represents a view of the problem domain
- Flow of control is inwards
- Dependencies go inwards

## Command Query Responsibility Segregation

- Separate commands from queries
- A command model and a query model
    - Each model can have theyr own operations an representations of data
- Commands
    - Operations that change the application state and return no data
    - Have side effects
- Queries
    - Operations that return data
    - No side effects

## Domain Driven Design

- Does not dictate any specific architectural style
- Only requires the model to be isolated from the technical complexities so that it can focus on domain logic concerns
- User Interface Layer
    - Interaction with external systems
    - Gateway to the effects that a human, an application or a message will have on the domain
    - Requests will be accpeted from this layer and the response will be shaped in this layer and displayed to the user.
- Application layer
    - It is the layer where business process flows are handled
    - The capabilities of the application can be observed in this layer
    - Direct clients of the domain model
    - Does not process business logic themselves
- Domain Layer
    - Core of the application
    - Where all the business logic is implemented
    - Kept away from dependencies as much as possible
    - Third Party libraries should not be added as much as possible
    - Focus on simulating the business processes

    - Kept agnostic from the infrastructure code.
- Infrastracture layer
    - Layer that accesses external services such as database, messaging systems and email services
    - External Services integration, repositories and persistence frameworks are implemented here
    - Dependent on the domain layer for domain objects and the repository contracts

## Microservices

- An independent, standalone capability designed as an executable that communicates with other microservices
- Standard lightweight inter-process communication
- Organized around business capabilities
- Loosely coupled
    - Highly maintainable and testable
- Owned by a small team
- Independently deployable
- Loose coupling
    - Limits the different types of calls from one service to another
    - Related behaviour to be together and unrelated behaviour elsewhere

## Monolith

- Simple to develop: Many development tools
- Simple to deploy: Simple deploy the WAR file
- Simple to scale: Just run multiple copies
- Single large code base
    - Hard for new developers to learn
- Frequent deployments gets difficult
- Scaling the application can be difficult
    - Each copy will need to access all the data

## Model View Controller

- MVC
    - Model: Data
    - View: Presentation
    - Controller: Logic
- Benefits
    - Separation of Concerns
    - Facilitates extensibility

## Model View Adapter

- All communication between model and view must flow through a controller / adapter
- Controller becomes a communication hub
- Accepting change notification from model objects and UI Events

## Model View Presenter

- Model: Represents business entities or domain model
- View: Lightweight. Only contains UI Elements
    - Passive View: View doesn't know model
    - Active View: Data binding or simple code in View
- Presenter: Presents user actions to the backend system. Presents to user after getting response

## Presentation Model

## Model View View-Model

- A structural design pattern that separates the object into three distinct types
- View: Similar to View in Presentation Model
- View Model: Equal to Presentation Model
- Model: Business logic layer of the application

## Web MVC

- Same as MVC but controller also handles the initial HTTP request

## Flux

- Action: Raised by View when user interacts with UI
- Dispatcher: Holds context to data store and propagates the action from View to store.
- Stores: Registered with dispatcher and respond to it
- Views: Respond to the change event and make appropriate changes