DARC

TODO:
- Familiarize with design principles and patterns
- Program design patterns
- Discuss about design patterns
- Learn about software architecture

Inheritance design patter - Example objects MallardDuck and RedHeadDuck are implementing the methods of a super class Duck.
Inheritance cons - code is duplicated among classes, Runtime behaviours are difficult, Hard to gain knowledge of all behaviours. Changes can unintentionally affect other classes that implement the super class.

Design principle: take what varies and 'encapsulate' it so it won't affect the rest of your code. The result? Fewer unintended consequences from code changes and more flexibility in your system.
Design principle: Program to an interface not an implementation. **Program to an interface** really means **program to a supertype**.
Design principle: Favour composition over inheritance; HAS-A can be better than IS-A. An example - each duck has a FlyBehaviour and a QuackBehaviour to which it delegates flying and quacking. When you put two classes together like this, you are using composition. Instead of inheriting their behaviour, the ducks get their behaviour by being composed with the right behaviour object. **Composition lets you change behaviour at runtime.**

The Strategy pattern defines a family of algorithms, Incapsulates each one and makes them interchangeable. Strategy let the algorithm vary independently from clients that use it.

State pattern allows an object to alter its behaviour when its internal state changes. The object will appear to change its class.
Strategy pattern - interchangeable algorithms

Strategy pattern - subclasses decide how to implement steps in an algorithm.
State pattern - change of internal state

The state and strategy patterns have the same class diagram but they differ in intent.
The Strategy patter typically configures Context classes with a behaviour or algorithm.
The state pattern allows a Context to change its behaviour as the state of the Context changes.
Using the State Pattern will typically result in a greater number of classes in your design.

Content - a class that can have a number of internal states. State interface defines a common interface for all concrete states.

Loose coupling - when two objects are loosely coupled, they can interact, bur they typically have very little knowledge of each other. The observer pattern is a great example of loose coupling.
Design principle - strive for loosely coupled designs between object that interact.

Observer pattern
Publisher - the **Subject**
Subscribers - **Observers**
Observers subscribe to a subject object with handles some important data and when the data in Subject gets changed, they get notified in some way. If an object is not a 'subscriber' - observer, it

does not get notified about the changes. *The Observer pattern defines one to many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically.*

The Singleton Pattern
- Ensures a class has only one instance and provides a global point of access to it.
- Ensures you have at most one instance of a class in your application.
- Provides a global access point to that instance.
- Using multiple class loaders, could result in in defeating the Singleton implementation and result in multiple instances.

Template Method Pattern
The Template Method defines the steps of an algorithm and allows subclasses to provide the implantation for one or more steps. Abstract class contains the template method and abstract versions of the operations used in the template method. There may be many ConcreteClasses, each implementing a full set of operations required by the template method. The concrete class implements the abstract operations which are called when the template method needs them.

To prevent subclasses from changing the algorithm, in the template method, declare the template as final.
The strategy and template method patterns both encapsulate algorithms, the first by composition and the other by inheritance.
The template method's abstract class may define concrete methods, abstract methods, and hooks.

The template method pattern defines the skeleton of an algorithm in a method, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

The Hollywood principle - don't call us, we'll call you. A low level component never call a high-level component directly - low level components may participate in computation, but the high-level components control when and how.

Dependency inversion principle - depend on abstractions. Do not depend upon concrete classes.

Factory Method Pattern
- Creator - has an abstract factoryMethod() that ask Creator subclasses must implement.
- ConcreteCreator - implements the factory method which is the method that actually produces products.
- ConcreteCreator is responsible for creating one or more concrete products. It is the only class that has the k owl edge of how to create these products.
- Product is an abstract class and Concrete product is it's implementation.

The Factory Method Pattern defines an interface for creating an object, but lets subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

Abstract Factory Pattern
The Abstract Factory pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes.

Quizz:
Hollywood principle
Loosely coupled designs
State pattern
Microservice architecture (loosely coupled architecture)
Event driven architecture : Mediator, Data replication engine, in - memory data, channels)
Space based architecture

Real-time production deployment
Layered architecture (4 layers)
Broker topology
Central even orchestration
Observer pattern
Factory method pattern
Template method
Strategy pattern
Abstract factory Pattern
DIP (Dependency inversion principle)
Least Knowledge principle
The Hollywood principle
Single responsibility principle