Design Rational
Make sure you only include:
The goals of the design
Explain the reason you implement/apply a principle
Advantages/disadvantages of your implementation

Example

Good design rationale

The diagram represents an object-oriented system for a football game that has five concrete classes implementing 3 different interfaces.

All five types of characters extended the abstract Character class. Since they share some common attributes and methods, it is logical to abstract these identities to avoid repetitions (DRY).

Coach has a dependency on Substitutable interface. In this way, we removed multiple dependencies for Coach class on multiple classes. When adding a new Character to the game, we do not need to fix the Coach class to substitute the new character (Open-Close Principle).

Defender and Attacker are implemented as interfaces. We make them interface to avoid multi-level inheritance from Character class. Also, classes such as Midfielder will need to implement both Defender and Attacker interfaces, which does not work if Defender and Attacker are classes. Our approach also made the system more extensible, a new set of operations could be easily added to the system by adding more interfaces since each of the current interfaces is separated by its own purpose (Interface Segregation Principle).

Dont repeat yourself (DRY)
Classes should be responsible for their own properties
Avoid excessive use of literals
Single Responsibility Principle
Open Closed Principle
Liskov Substitution Principle
Interface Segregation
Dependency Inversion Principle

Req 1:
The diagram represents an object-oriented system for the enemies system within a rogue-like adventure game. This includes 3 concrete classes implementing 1 abstract class.

The concrete enemies classes inherit the abstract class because many common attributes and methods are shared, hence the abstract class is made to avoid repetitions (DRY).

The enemies abstract class also has many dependencies and associations with other classes such as Behaviours, Application, Actor and AttackAction. By introducing this abstract class, we are able to reduce many dependencies that each concrete class may have as well as ensuring that if changes are to be made to those enemies classes such as balancing or reworking how the enemies are implemented, it will be easy to refactor and extend the system without modifying the rest of the code. This also allows new enemies to be added without adding multiple dependencies with those classes (Open-Close Principle).

Req 2:
The diagram represents an object-oriented system for the runes system within a rogue-like game. This includes multiple concrete classes working together.

The concrete class RunesManager is introduced as a way to manage all runes functionalities and interactions within the application. This includes updating the number of runes a Player may have after killing an enemy. The Runes interface is also implemented to help out with tracking the number of runes of each Actor. Its sole purpose is to update and return the number of runes a player may have (Single Responsibility Principle).

The Runes interface is also implemented to avoid having to change implementation of the Player and Enemy classes if RunesManager were to receive a change or update to the class which may include the methods on calculating how runes are calculated (Dependency Inversion Principle).