Design Rationale

Major Design Decisions:

Respawning the Player: When the player dies, we respawn them at the starting location in the Abandoned Village. To achieve this, we have modified the ‘InitialiseGameMap’ class to have a ‘reset’ method, which is called when the player dies. This method removes enemies, resets boss health, locks gates, and drops runes as required.

Basic requirements such as The maximum values of the player's attributes are maintained as they were before the player's death, The player keeps all items in their inventory and the player's wallet balance is reset to 0, and the corresponding amount of runes is dropped at the player's last location, and are implemented through the ‘ControlMapInitialise’ class.

The ’ContorlMapInitialise’ Class has a few static method such as ‘respawn()’ which the player class can access provided with valid arguments, and the ‘resetAll()’ method which player can call after they’ve become unconscious.

All spawned enemies (excluding bosses) are removed from the map when the player dies through the reset() method in ‘InitialiseGameMap’ class.

Advantages:

1. The static methods help reduce dependency for classes that require functionality for the game reset clockwork.

2. Code Reusability: The implementation leverages existing classes and methods, such as the `InitialiseGameMap` class, to reset the game state upon player death. This follows the DRY (Don't Repeat Yourself) principle.

3. Encapsulation: The encapsulation of functionality related to the player's attributes and wallet balance is maintained within the `Player` and `Runes` classes. This adheres to the SOLID principles, particularly the Single Responsibility Principle (SRP).

Disadvantages:

1. **Complexity:** The code for respawning the player and managing their attributes, inventory, and wallet balance adds complexity to the codebase. However, this complexity is necessary to implement the desired game behavior.

2. The usage of static methods and fields may lead to unintentional changes if not handled carefully.

Software Design Principles:

DRY (Don't Repeat Yourself): The design follows the DRY principle by reusing existing methods and classes (e.g., `InitialiseGameMap`) to reset the game state for each map. This promotes code reusability and maintainability.

SOLID Principles:
Single Responsibility Principle (SRP): The `Player` and `Runes` classes are responsible for managing attributes and wallet balance, respectively, adhering to the SRP. The InitialiseGameMap class is responsible for resetting the map and map only, others such as player attributes are handled differently in other class.

Open/Closed Principle (OCP): The design is open for extension, as adding new game features at restarting the game may be easily added into the InitialiseGameMap class, without affecting other class that is not responsible for game reset.

In conclusion, the design for respawning the player upon death adheres to important software design principles, ensuring player continuity and resource recovery while maintaining code reusability and encapsulation. Although it adds complexity, it aligns with good game design practices and provides an engaging player experience. Edge cases may require further consideration for robustness.