Advanced Programming Project

– Design Choices Overview –

Viki Lauvrys (s0230799) | Advanced Programming | 2024-2025

See **README.md** for game documentation

Model-view-controller pattern

*The entire source code related to the game is wrapped in the namespace **Doodle_Jump::**. It has two sub-namespaces: Doodle_Jump::Logic and Doodle_Jump::View.*

Doodle_Jump::Logic acts as the model, encapsulating the core game logic and mechanics. This namespace is designed to be a standalone library, which does not require SFML to compile. This ensures that the logic can be tested and compiled independently of the graphical interface. Adding SFML-includes to any file within this namespace will result in compilation errors as a safeguard to maintain this separation.

***Doodle_Jump*::View**, on the other hand, serves as both the view and the controller. It is responsible for rendering the game using SFML and handling user input. By maintaining this separation, the game logic remains decoupled from the graphical rendering, adhering to the principles of the Model-View-Controller (MVC) design pattern.

Observer Pattern

The Observer pattern is used to keep various parts of the game synchronized with the state of the World object. The World acts as the Subject, while classes such as PlayerRenderer and PlatformRenderer act as Observers.

The Observer interface is defined in the Doodle_Jump::Logic namespace. It declares a single method, update, which must be implemented by any concrete observer. This method will be called to notify the observer of changes in the subject's state.

Concrete observers such as PlayerRenderer and PlatformRenderer implement the Observer interface. These classes are responsible for updating the game's view in response to changes in the game state. PlayerRenderer updates the player's sprite based on the player's movements and interactions within the game world. PlatformRenderer: Updates the platform sprites to reflect their current positions and states within the game world.

The Subject, typically represented by the World class, maintains a list of observers and provides methods to attach, detach, and notify observers. When the state of the World changes, it calls the update method on each registered observer, ensuring that all observers are informed of the change.

Observer registration: Observers like PlatformRenderer register themselves with the World class using an attach method. This ensures that they will receive updates whenever the state of the World changes.

Notifications: When the World state changes, it notifies all registered observers by calling their update methods. The observers then update their views to reflect the new state. For example, the PlatformRenderer updates the positions of platform sprites based on the updated positions of platforms in the World.

The Observer design pattern in the Doodle_Jump project plays a vital role in maintaining a responsive and well-organized codebase. By ensuring that changes in the game state are efficiently communicated to various components, the pattern helps in creating a maintainable, scalable, and flexible game architecture. This design choice aligns with the project's goals of low coupling, high cohesion, and extensibility.

Abstract Factory

The Abstract Factory pattern is used to create different types of game entities such as players and platforms. The EntityModel class acts as the abstract factory, providing an interface for creating entities. Concrete factories such as PlayerCreator and PlatformCreator implement this interface to create specific types of entities.

The EntityModel class is the abstract factory that declares a method for creating entities. This method, FactoryMethod, is overridden by concrete factories to produce specific types of game entities.

Concrete factories, which are used to create the player and platforms, all extend the Entity-Model class. Each of these concrete factories overrides the FactoryMethod to create a specific type of entity. The PlayerCreator is responsible for creating player entities, actually only one every game. PlatformStaticCreator creates static platform entities. PlatformHorizontalCreator is tasked with creating horizontal platform entities, which move on the x-axis. PlatformVerticalCreator creates vertical platform entities, which move on the y-axis. And lastly, PlatformTemporaryCreator is responsible for generating temporary platform entities, which disappear when jumped on.

The createObject method in the EntityModel class calls the FactoryMethod to create an entity. This method is used by client code to create entities without needing to know the specific class of the entity being created. Thus nicely separating all operations.