Taller 5 - Johannes Almas: 202224983

Código fuente del proyecto: https://github.com/mtala3t/Super-Mario-Java-2D-Game.git

Super Mario

El proyecto elejido es una versión simple del juego clasico de Super Mario. El objetivo del proyecto es de lanzar un cliente que deja al usuario jugar este juego. La estructura general del proyecto tiene quatro partes – y paquetes - principales: “graphics”, “input”, “test” y “tilegame”.

“graphics”: Este paquete tiene como responsabilidad manejar las graphicas para que se puede ver el nivel y los caracteres etc.

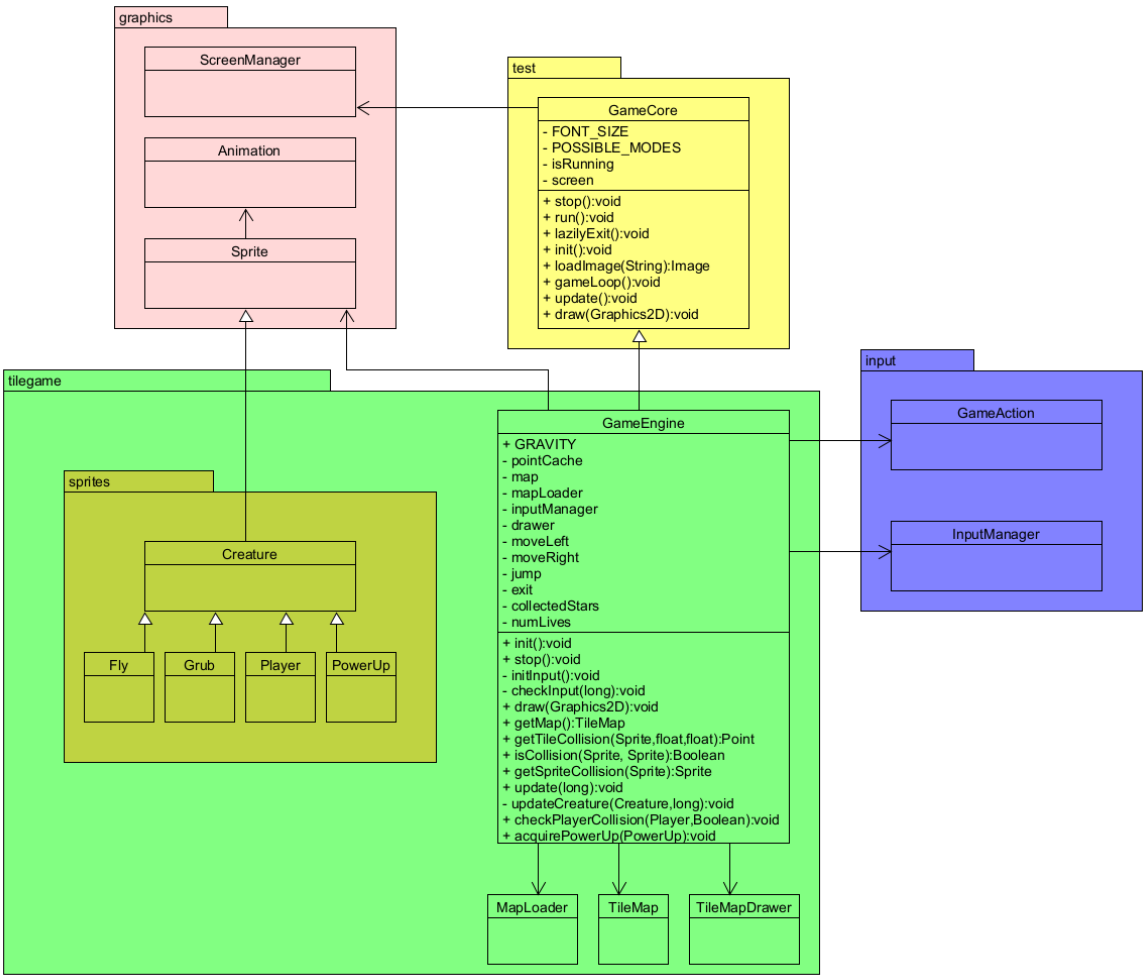
“input”: Este paquete tiene como responsabilidad manejar los inputs del usuario – es decir manejar el movimiento de Super Matio.

“test”: Este paquete tiene solo una clase, y es una de las dos clases más interesantes para este taller. La clase es una clase abstracta “GameCore” que maneja las cosas generales del juego (como iniciar, terminar y el manejo del tiempo) mientras deja a su subclase “GameEngine” determinar las cosas más únicas a este juego en particular.

“tilegame”: Este paquete maneja los elementos principales del juego, como el mapa, las creaturas y Super Mario. También tiene la clase “GameEngine” que herede de “GameCore” y agrega todas las informaciones del juego de Super Mario.

El reto de este proyecto es el manejo de muchas partes individuales: el carácter principal, los enemigos, el mapa etc.. y integrarlos visualmente con los inputs del usuario. Uno tiene que integrar las físicas de Mario, los movimientos de los enemigos y dibujar todo en tiempo real. Con tantas cosas que manejar, la clase “GameEngine” tiene un trabajo importantissimo.

Aquí se puede ver un diagrama de clases del proyecto con los paquetes diferentes mostrados en cajas coloradas.



Patrón Template Method

En este reto, vamos a mirar lo que es el patrón de diseño “Template Method” y cómo está implementado en este proyecto. El patrón “Template Method” es un patrón de diseño comportamental que define la estructura de un algoritmo en una clase base, pero permite que las subclases proporcionen implementaciones específicas de algunos pasos del algoritmo.

Tiene una clase abstracta que define el esqueleto del algoritmo y declara los métodos que deben ser implementados por las subclases. Esta clase tiene un “Template method” que define la secuencia de pasos del algoritmo. Este método utiliza otros métodos, algunos de los cuales se definen en la clase base y otros son métodos abstractos que se implementan en las subclases. Muchas veces, los métodos abstractos simplemente declarados sin nada más, y es el trabajo de las subclases llenarlos y darlos sentido. Las subclases disponen el contexto a la clase principal para que tenga sentido en el contexto del proyecto.

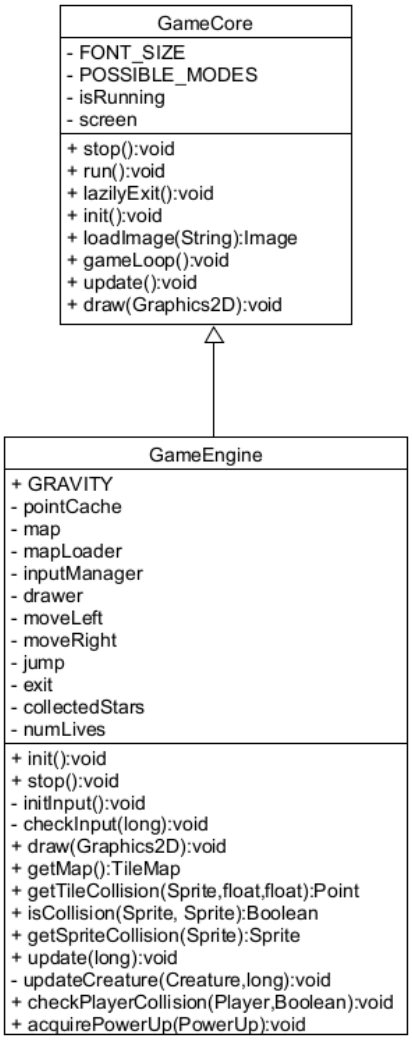
Este patrón se usa en muchas situaciones diferentes, entre ellos frameworks y bibliotecas, y plantillas de documentos. El patrón también está relacionado con la heredencia, porque para usarlo, necesitamos que una subclase herede el “Template method” y implementa los métodos que el “Template method” llama. Vamos a enfoquarnos en el uso des patrón en las clases

“GameCore” y “GameEngine”. También se usa mucha heredencia en el paquete “sprite”. La clase “Creature” herede de la clase “Sprite”, pero como “Sprite” no tiene un “Template Method”, no usa el patrón. De la misma manera, las quatro subclases de “Creature” hereden su comportamiento sin hereder ningun método “Template”.

Template Method en Super Mario

El uso del patrón “Template Method” en el que vamos a enfoquarnos es el entre las clases “GameCore” y “GameEngine”. Aquí está una representación de las dos clases.

Podemos ver que “GameEngine” herede de “GameCore”. “GameEngine” es la clase que maneja el juego, pero hay algunas cosas de cómo manejar un así juego que no son especificas para el juego de Super Mario. Estas cosas ya están implementadas en la clase “GameCore”. Usando el patrón “Template Method”, “GameEngine” puede hereder el funcionamiento de “GameCore” y agregar todas las cosas especificas para el juego de Super Mario. Tiene sentido usar el patrón en este caso para facilitar el trabajo de “GameEngine” y separar el trabajo de correr el juego del trabajo de manejar los elementos del juego mientras corre.



Las dos clases tienen algunos métodos en comun, como “init()”, “stop()”, “draw()” y “update()”.

Para empezar el juego, llamamos el “run()” de “GameEngine”. Este método llama “init()” y “gameloop()”. “run()” es el “Template method” de la superclase “GameCore”. “init()” y “gameloop()” ya estan implementados hasta tener una funcionalidad básica, pero “gameloop()” usa los dos métodos vacíos “update()” y “draw()”, dos operaciones que tiene que implementar “GameEngine”.

“draw()” y “update()” están implementados en “GameEngine” y por eso, tiene sentido el método “gameloop()” y por lo tanto el “Template method” “run()”.

“init()” es un ejemplo de una clase que tiene una implementación basica en la clase abstracta, pero dónde necesita modificaciones en la subclase para funcionar bien en nuestro contexto. Si fuera bien implementado el patrón, esta clase (en “GameCore”) llamaría a otra clase “initExtension()” – un método “hook” que deja al usuario extender la inicialización del juego si necesario. Si sabemos que “init()” sí o sí necesita una extensión, “initExtension()” sería no un “hook”, sino un “abstract operation”. En todo caso, “GameEngine” podría cambiar

“initExtension()” sin cambiar “init()”, que sería mejor selon los princípios “SOLID” y que tendría más sentido con el patrón “Template method”.

Ventajas, disventajas y conclusión

Ya hablamos un poquito de porqué tiene sentido usar el patron “Template Method” en este caso, pero ya vamos a hablar más de las ventajas y disventajas de hacerlo. Una ventaja es que las dos clases “GameCore” y “GameEngine” tienen trabajos distintos, entonces es una manera fácil de implementar los detalles en una clase y el funcionamiento general en otra. Esto hace la redación del código y la resolución de problemas más fácil porque uno puede saber, dependiendo del problema, en qué clase necesita trabajar.

Otra ventaja es que leer y entender el codigo es más fácil si cada clase tiene una responsabilidad especifica y definida, entonces separar el funcionamiento así en dos clases tiene mucho sentido.

Normalmente, la flexibilidad y usar el código varias veces es una gran ventaja de este patrón. Esta ventaja podemos por ejemplo ver en el paquete “sprites” del primer diagrama. Cada subclase es diferente, pero usa el mismo código de la clase abstracta. Cada clase también se puede diferenciar de las otras sobreescribiendo el código de la clase abstracta, haciendo las subclases más flexibles (Esta ventaja es también relevante en el caso de heredencia sin un “Template method”). Sin embargo, esto no es una ventaja tan grande en el ejemplo de “GameCore” y “GameEngine” ya que solo hay una subclase. El codigo de la clase abstracta no se usa tantas veces, y la flexibilidad sería la misma si fuera una sola clase.

Esto también es una de las desventajas del uso de este patrón en nuestro caso. Como solo hay una subclase, tener dos clases, y dos paquetes distintos puede complicar el entendimiento de la estructura del programa.

Una desventaja también puede ser que la clase abstracta tiene todo el poder, así que las subclases tienen que obeceder a la clase abstracta. Esto podria ser un problema en el paquete “sprites” si una de las subclases tuviera mucho que ver con las otras, pero que necesitara un funcionamiento que no dejara la clase abstracta. Esta desventaja, sin embargo, no es nada grave con “GameCore” y “GameEngine” como solo hay una subclase. Esto quiere decir que si hay un funcionamiento que no deja la clase abstracta, podemos editar la clase abstracta sin arriesgar que afecta otra subclase de manera mala. Normalmente, usando este patrón, es muy importante diseñar bien la clase abstracta para abstraer el esqueleto de un método, pero al mismo tiempo dar el poder necesario e las subclases.

En conclusión, en el caso de “GameCore” y “GameEngine”, no hay muchas desventajas, ni tantas ventajas. Casi sin desventajas, y con una ventaja bastante grande, sin embargo, yo diría que tiene mucho sentido usar el patron en este caso.

Si uno no quisiera usar el patron, podría hacer todo en una sola clase. La clase “GameEngine” sería más complicada, más pesada y menos legible, pero funcionaría. Si uno quisiera resolver el problema con la ventaja de separar los trabajos distintos (correr la aplicación y manejar lo demás) como hace el patrón, pero sin usarlo, podría usar varias clases, pero sin heredencia y por lo tanto sin el patrón. Si uno resolviera el problema así, sin embargo, arriesgaría complicar las cosas mucho más.