Assignment 2 - Group 8008 - SEM

Joost Verbraeken 4475208
Eric Cornelissen 4447859
Nick Winnubst 4145747
Cornel de Vroomen 4488628
Michael Tran 4499638

Exercise 1 - Design patterns

Exercise 1.1

Observer pattern:

For our game the player controls the doodle and menus with a mouse and a keyboard. There may be many objects that want to know whether a mouse button is pressed, but it would be inefficient and ugly to have each object each step check if the mouse button was pressed. Instead we use a single class, *InputManager*, to check each step if the user used the mouse or keyboard.

Every object that wants to know if the mouse button was pressed registers itself as a *MouseInputObserver* (in case of keyboard events as a *KeyInputObserver*) at the InputManager class and implements the method that InputManager will call if an event happened. This conforms with the programming principle “Tell, don’t ask”. When a mouse button was pressed the InputManager loops over all classes that registered themselves as a MouseInputObserver and calls the method they implemented specific to the mouse event on each of them. Of course classes should deregister themselves from InputManager if they don’t want to receive mouse events anymore.

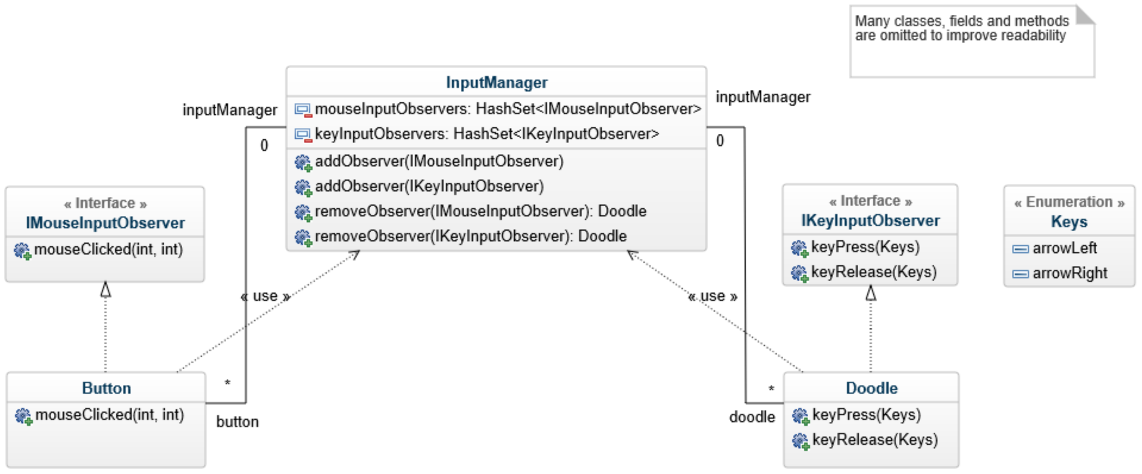
Factory pattern:

Our game uses a great many factories. It are so many it doesn’t make sense to elaborate on each of them, but we’ll have a look at a few. *ButtonFactory* creates buttons and has several methods like *createPlayButton*, *createMainMenuButton*, etc. When such a method is called it creates a new Button, which is an object that has no idea what it’s supposed to do by itself, and assign to this button a sprite and an action. In other words, the factory defines the button.

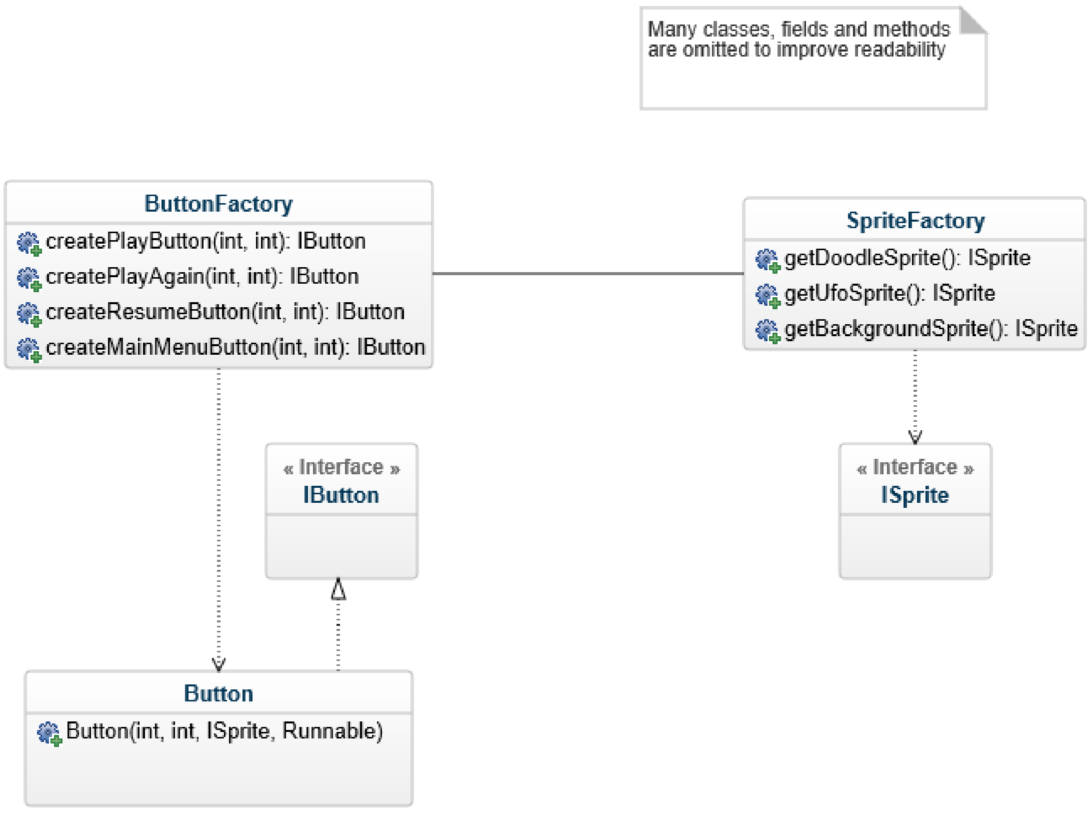
We want to use factories like this do this for several reasons. Firstly, the factory does not have to create a new object each time a createFoo method is invoked. For example, SpriteFactory uses a cache and will almost always return the cached sprite it that was created earlier already. Secondly, the factory can choose to create any subtype of its return type. Suppose we want to add multiple kinds of doodles, than we don’t have to change all calls to the method creating the doodle, but simply change the method in the factory and let that method return a subtype of Doodle.

Exercise 1.2:

Observer pattern:

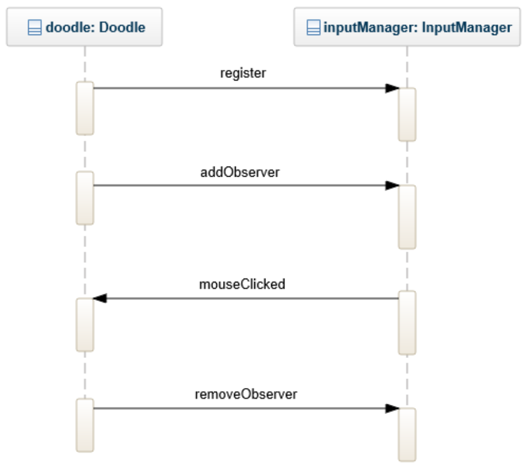

Many classes, fields and methods are omitted to improve readability
InputManager
mouseInputObservers: HashSet<IMouseInputObserver>
keyInputObservers: HashSet<IKeyInputObserver>
addObserver(IMouseInputObserver)
addObserver(IKeyInputObserver)
removeObserver(IMouseInputObserver): Doodle
removeObserver(IKeyInputObserver): Doodle
inputManager
0
inputManager
0
« Interface »
IMouseInputObserver
mouseClicked(int, int)
« Interface »
IKeyInputObserver
keyPress(Keys)
keyRelease(Keys)
« Enumeration »
Keys
arrowLeft
arrowRight
« use »
« use »
Button
mouseClicked(int, int)
*
button
Doodle
keyPress(Keys)
keyRelease(Keys)
*
doodle

Factory pattern:

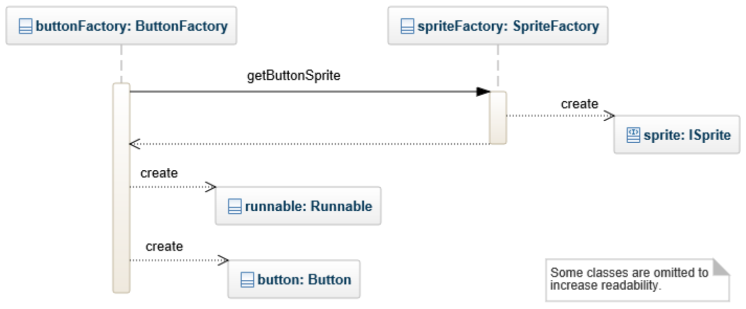

Many classes, fields and methods are omitted to improve readability
ButtonFactory
createPlayButton(int, int): IButton
createPlayAgain(int, int): IButton
createResumeButton(int, int): IButton
createMainMenuButton(int, int): IButton
SpriteFactory
getDoodleSprite(): ISprite
getUfoSprite(): ISprite
getBackgroundSprite(): ISprite
« Interface »
IButton
« Interface »
ISprite
Button
Button(int, int, ISprite, Runnable)

Exercise 1.3:

Observer:



Factory:


buttonFactory: ButtonFactory
spriteFactory: SpriteFactory
getButtonSprite
create
sprite: ISprite
create
runnable: Runnable
create
button: Button
Some classes are omitted to increase readability.

Exercise 2 - Your wish is my command

Power up & Power down requirements

For the game Doodle Jump, four categories of power ups and power downs can be identified using the MoSCoW method.

1. Must haves

- All power ups from the original game (trampoline, spring, spring-shoes, rocket, propeller, shield).
- A power up to decrease the size of the Doodle.
- A power up to increase the size of the Doodle.
- A user deployable power up that pushes enemies away.

2. Should haves

- A power up to increase shot speed.
- A temporary score multiplier.

- A power up that allows one hit with an enemy for free (does not stack).
- A power up/power down teleporting the Doodle a set distance up or down. Increasing or decreasing the score accordingly.
- A power up that adds an orbital to the Doodle which kills enemies on collision.

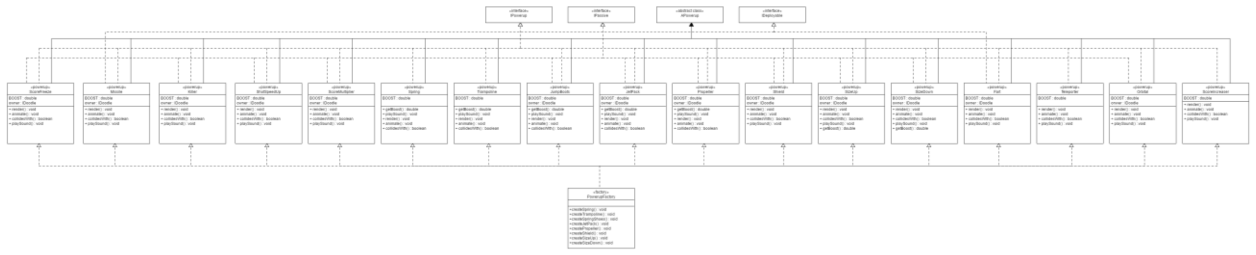
3. Could haves

- A usable target seeking missile.
- A temporary score freezer.
- Power ups that increase score by X amount.

4. Won’t haves

- Power up to slow down enemies.
- Power ups that change shot type.
- Power ups that change shot strength.
- user deployable powerdowns.

The following figure is the UML of the power ups. Because it’s hard to read you can open the full picture in UML/Powerups.png



Exercise 3 - 20-Time

For the 20-time sub-assignment, our group decided to try and implement new game modes, since doodle jump and our framework can easily support this.

We thought of a lot of possible modes to implement which can all be found in PossibleModes.txt.

Due to time constraints it is impossible to implement all of these. As such we have prioritized these possible game modes and concluded the top 6 contenders to be:

Space.

Underwater.

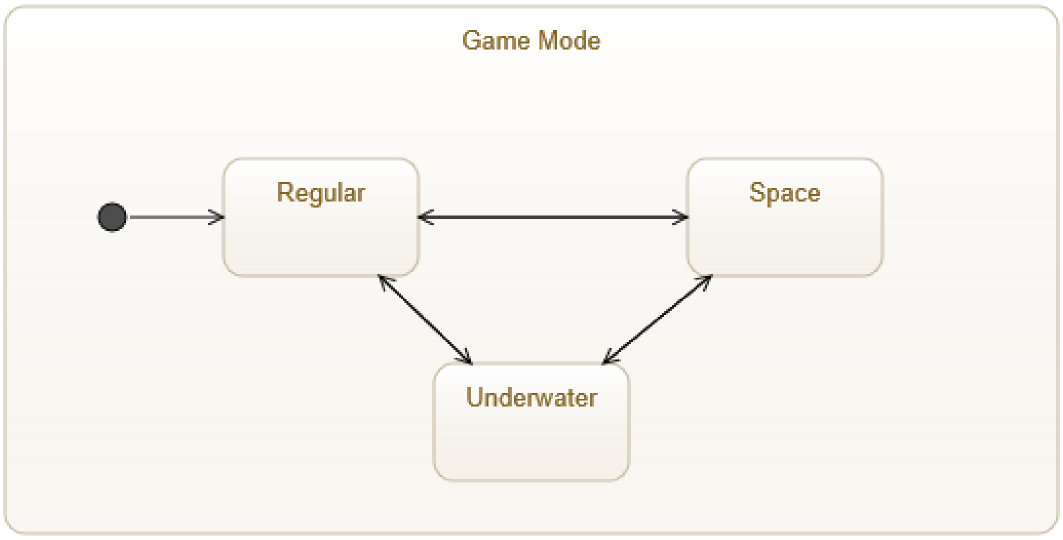
Invert.

Darkness.

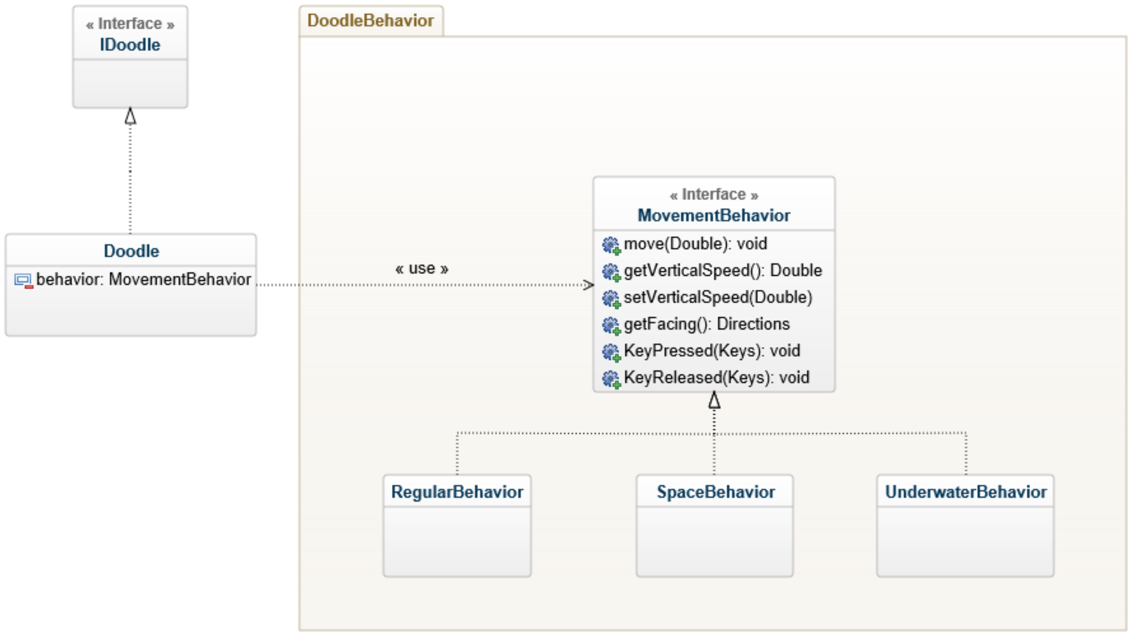
Story/Tutorial.

In the requirements for the game modes, one can find the specific demands these modes should meet. The currently implemented game modes (Space and Underwater) have been implemented according to two design patterns. To be more specific: A state machine pattern and a behavioral pattern.

As can be viewed in the following UML you can observe the switch between different states.



Furthermore, the game modes Space and Underwater are based on a different behavior in response to key input. This gives naturally rise to a behavioral pattern implemented as follows:



These two design patterns give rise to a strong increase in possible evolvements of the system. This includes the easy additions of even more game modes, and/or more type of doodle behaviors.