Assignment 3 - Group 8008 - SEM

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

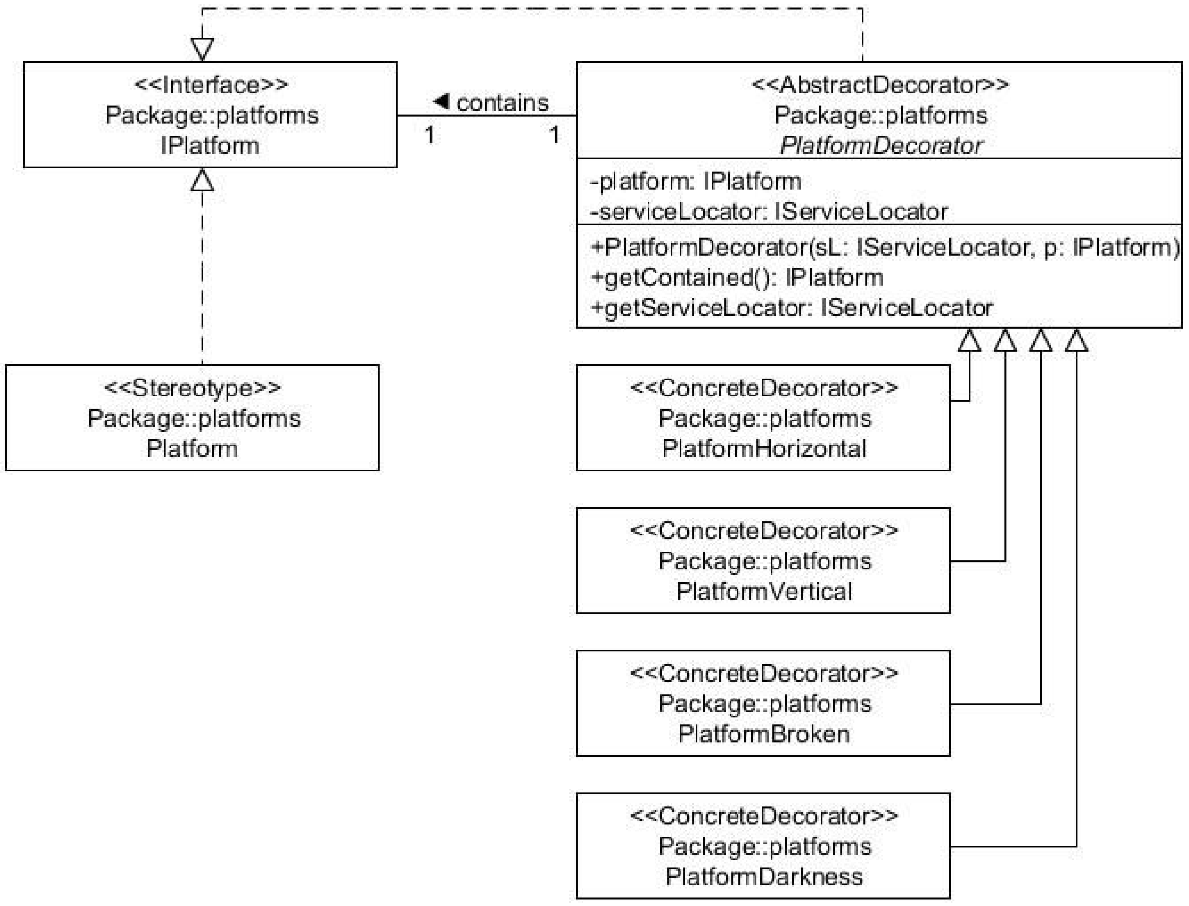
Exercise 1 – Design patterns

Exercise 1.1

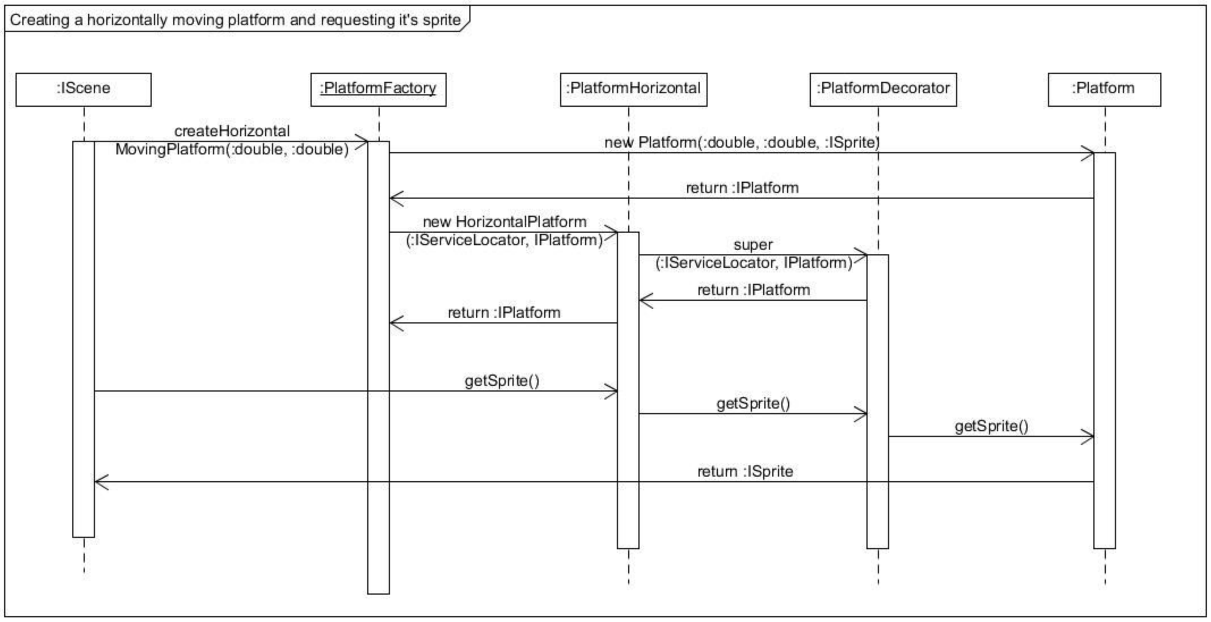
Singleton pattern
A very well known pattern is the Singleton pattern. This pattern guarantees that only a single object of a class is created. This is a very relevant pattern that can be used to efficiently manage your RAM. This is mainly due to the fact that instead of ending up with a large sum of largely unused objects, the same one is called over and over.
3 Singleton patterns have been implemented in the game. Firstly, a lazy/eager constructor in ServiceLocator. Secondly, a synchronized Getter in ButtonFactory. Thirdly and last, a double-locked Singleton pattern in the class Calc. These three classes are used throughout the game, but only need to be instantiated once. This makes these classes the perfect candidates for the singleton design pattern.

Decorator Pattern
The implementation of different types of platform is done using a decorator pattern. This pattern enables the easy combination and extension of different kinds of platforms. In example, a complex platform that moves horizontally, moves vertically, and breaks upon contact, is easily feasible.
The implementation of the decorator pattern is done using an abstract decorator which defines the global behavior of the so called condiments or decorations. This behavior is further specified in the concrete decorator classes. The easy overlapping of these objects is caused by encapsulating another platform (with or without condiments already encapsulating it), and adding its own functionality. Should no functionality be added, it will call for its encapsulated object, usually all the way to the base platform.

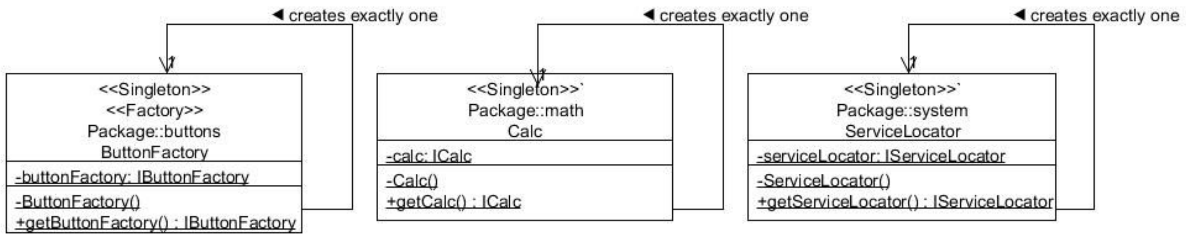
UML Class diagram decorator



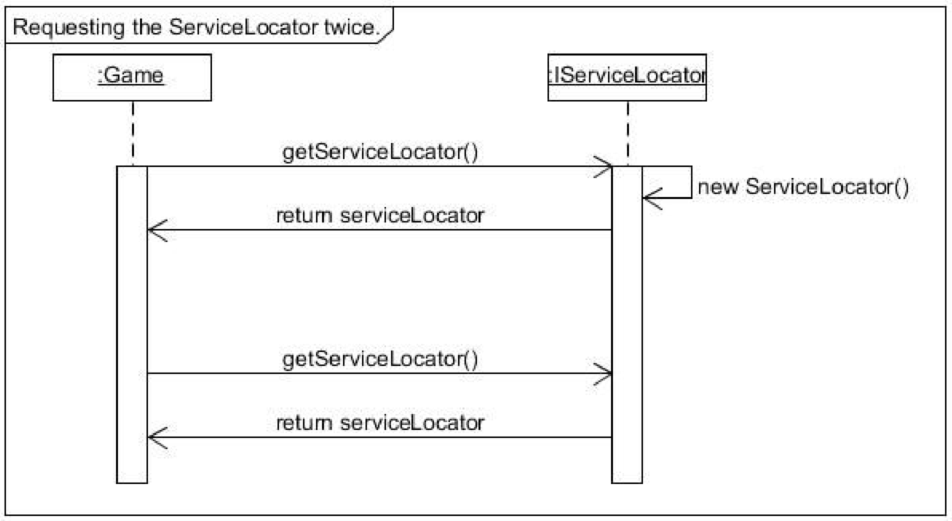
UML sequence diagram decorator


Creating a horizontally moving platform and requesting it's sprite
:IScene
:PlatformFactory
:PlatformHorizontal
:PlatformDecorator
:Platform
createHorizontal MovingPlatform(:double, :double)
new Platform(:double, :double, :ISprite)
return :IPlatform
new HorizontalPlatform (:IServiceLocator, IPlatform)
super (:IServiceLocator, IPlatform)
return :IPlatform
return :IPlatform
getSprite()
getSprite()
getSprite()
return :ISprite

Class diagram Singleton



Sequence diagram Singleton



Exercise 2 - Your wish is my command

Our task for this assignment from the TA was to implement a multiplayer mode into our game. At first we were a bit skeptical about the idea and we had a short discussion about it with our TA, but in the end we saw some possibilities and decided to implement it.

We decided to implemented arcade style. I.e. by moving the background automatically, indepenend of the Doodles in the screen while also allowing Doodle to move higher up the screen.
Another important thing we decided, especially after playing a bit with two players, that powerups should be disabled as every powerup will instantly win you the game. Also, because a Doodle can move higher up the screen, and so even of the screen, enemies should also be disabled.
Formally, we created these requirements:

Requirements

For the game Doodle Jump, four categories of multiplier requirements can be identified using the MoSCoW method.

1.1 Must haves

• It is possible to play with two or more people on one computer.
• Powerups and enemies are disabled when playing the game with two or more people.
• When playing with two or more players each player has its own Doodle that can be controlled independent of the other players/Doodles.
• When playing with two or more players the camera moves up automatically, independent of the any of the Doodles positions.
• When playing with two or more players the speed of the screen camera should increase over time (up until a certain limit).
• When all but one of the Doodles died, the Doodle that is still alives is the winner.

1.2 Should haves

• It is possible to specify the amount of players.
• The score is disable when playing with two or more players.
• When a Doodle is outside the screen, there is be an arrow at the top of the screen indicating the X position of that Doodle.

1.3 Could haves

• It is possible to play with two or more people over an LAN connection.

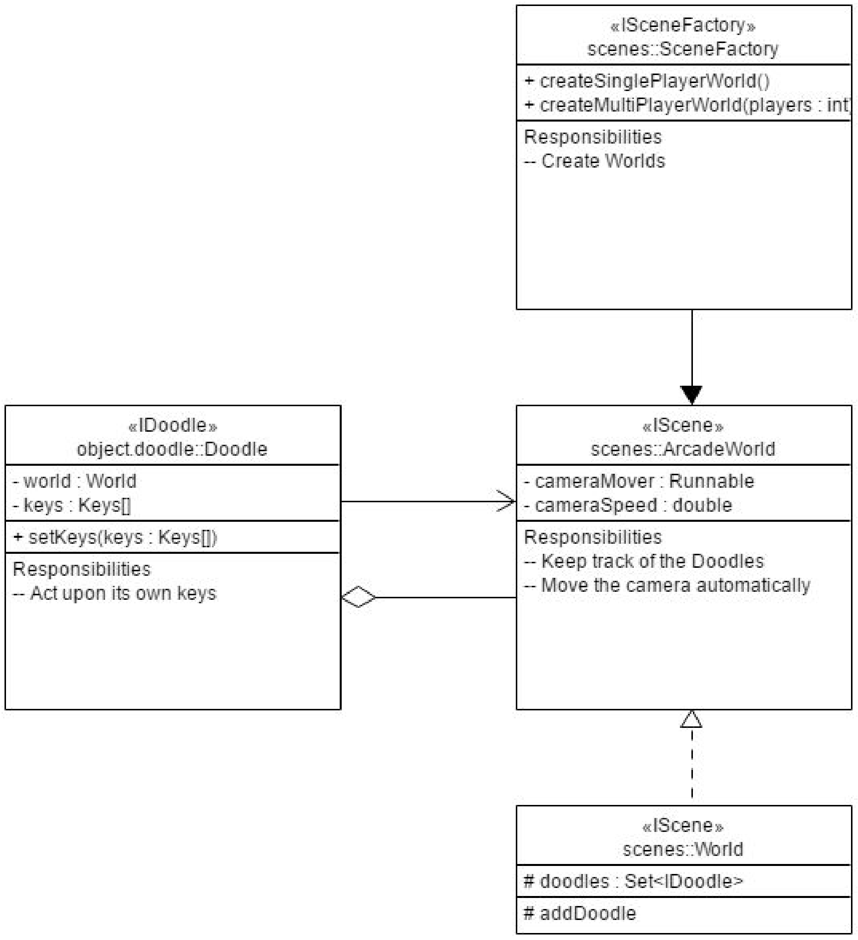
1.4 Won't Haves

• It is possible to play with two or more people over an internet connection.
• Powerups that are designed specifically for multiplayer.

1.5 Organizational Requirements

• A first working version of the game modes should be implemented by 14th of October.

After the requirements where created we created some UML to visualize how the multiplayer modes will be implemented
The UML described here only contains the new aspects of the system. From the requirements we can derive a few attributes that should be added to the Doodle, World and SceneFactory.



Exercise 3

Functional Requirements

For the implementation of upgradeable powerups in the game, a few things have to be done beforehand. Firstly, there should be a way of getting coins so that better powerups can be bought. This is done by completing missions. Secondly, because getting upgrades costs a lot of time, Within these function requirements, four categories can be identified using the MoSCoW model for prioritizing requirements:

Must Haves

- The player has a certain amount of coins
- The player has 3 missions each time
- If the player manages to complete one mission, a new mission will be created for the player for the next run
- Each missions has a certain reward in coins. The higher the difficulty, the higher the reward

Should Haves

- Items can be unlocked or upgraded by the player
- The missions the player has accomplished and the 3 missions he still has to do will be saved to the disk
- Each time the game is loaded, the 3 missions that must be done, the amount of coins and the quantity/quality of the items will be loaded from the disk if the file exists
- If the file containing the missions and coins does not exist, then the file and the 3 missions will be created and the amount of coins is set to 0 and the quantity/quality of the items is set to the minimum
- In the menu of the game a scene displaying a shop can be opened
- In the shop items can be bought with coins
- If an item is bought the amount of coins of the player is decreased with the price of the product
- Trampolines have 3 stages: Trampoline, Circus Cannon and Rocket Launcher
- Jetpacks have 3 stages: Jetpack, Chrome Plated Afterburner and Space Rocket

Could Haves

- Each day the game is started, the player is rewarded with a certain amount of coins
- If the player starts the game multiple days in a row, the rewards get higher
- The player has a rank, that is a level with a name
- The player has a certain amount of experience, initially 0 and loaded/saved from/to a file
- Each time the experience reaches a threshold associated with a rank, the rank goes up one level and the experience is set to 0

- Each unit the player jumps, results in an increase in experience
- Each time the player reaches a certain height, the experience increases with a certain amount
- The first time the player reaches certain heights, the experience increases with a certain amount
- Each rank is associated with a game mode that is unlocked when the player reaches the associated rank

Won't Haves

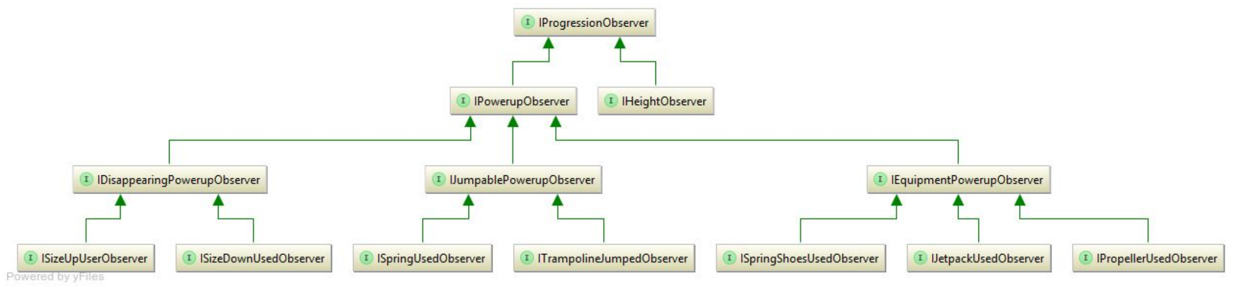
- An Iron Man suit that can be unlocked
- Support for mods
- Buying coins with real money
- A website with a ranking of players with the highest amount of money

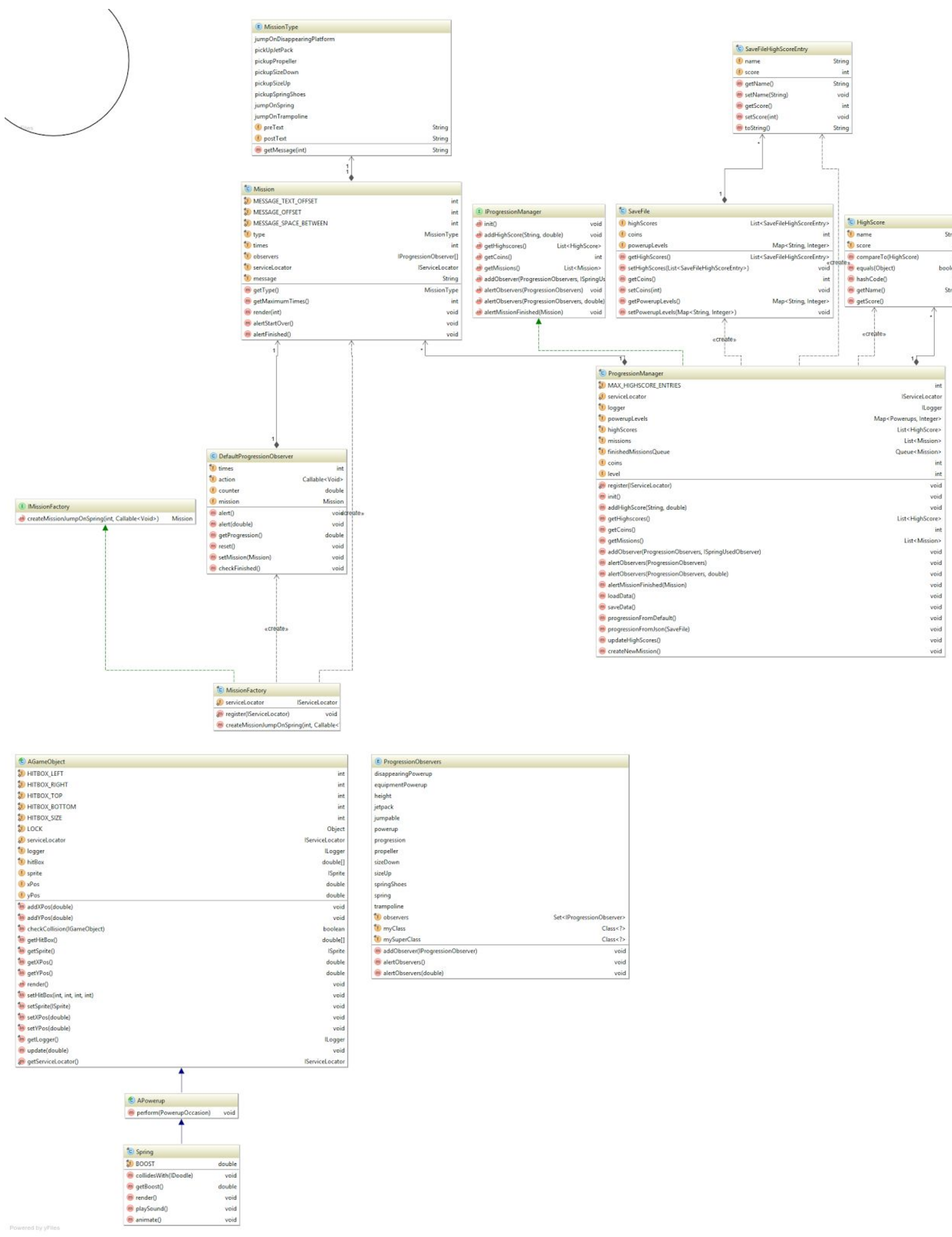
Non-functional Requirements

Besides the provided functionality and services, design constraints need to be included in the requirements specification as well.
- All new features will be implemented in Java 8
- All features will be implemented in the week of 10 October, 2016
- The implementation of the features shall have at least 75% of meaningful line test coverage (where meaningful means that the tests actually test the functionalities of the game and for example do not just execute the methods involved)
- No CheckStyle, PMD and FindBugs error will be present in the final version - Every class, interface, enum, method and field shall be documented using JavaDoc

UML

Both the UML's below can be found in the map UML in our Github repository. The first one is called progression.png and the second one is called progressionOverview.png.




MissionType
jumpOnDisappearingPlatform
pickUpJetPack
pickupPropeller
pickupSizeDown
pickupSizeUp
pickupSpringShoes
jumpOnSpring
jumpOnTrampoline
preText String
postText String
getMessage(int) String
1
1
Mission
MESSAGE_TEXT_OFFSET int
MESSAGE_OFFSET int
MESSAGE_SPACE_BETWEEN int
type MissionType
times int
observers IProgressionObserver[]
serviceLocator IServiceLocator
message String
getType() MissionType
getMaximumTimes() int
render(int) void
alertStartOver() void
alertFinished() void
IProgressionManager
init() void
addHighScore(String, double) void
getHighscores() List<HighScore>
getCoins() int
getMissions() List<Mission>
addObserver(ProgressionObservers, ISpringUs
alertObservers(ProgressionObservers) void
alertObservers(ProgressionObservers, double) void
alertMissionFinished(Mission) void
SaveFileHighScoreEntry
name String
score int
getName() String
setName(String) void
getScore() int
setScore(int) void
toString() String
SaveFile
highScores List<SaveFileHighScoreEntry>
coins int
powerupLevels Map<String, Integer>
getHighScores() List<SaveFileHighScoreEntry>
setHighScores(List<SaveFileHighScoreEntry>) void
getCoins() int
setCoins(int) void
getPowerupLevels() Map<String, Integer>
setPowerupLevels(Map<String, Integer>) void
HighScore
name String
score int
compareTo(HighScore) int
equals(Object) boolean
hashCode() int
getName() String
getScore() int
«create»
«create»
«create»
ProgressionManager
MAX_HIGHSCORE_ENTRIES int
serviceLocator IServiceLocator
logger ILogger
powerupLevels Map<Powerups, Integer>
highScores List<HighScore>
missions List<Mission>
finishedMissionsQueue Queue<Mission>
coins int
level int
register(IServiceLocator) void
init() void
addHighScore(String, double) void
getHighscores() List<HighScore>
getCoins() int
getMissions() List<Mission>
addObserver(ProgressionObservers, ISpringUsedObserver) void
alertObservers(ProgressionObservers) void
alertObservers(ProgressionObservers, double) void
alertMissionFinished(Mission) void
loadData() void
saveData() void
progressionFromDefault() void
progressionFromJson(SaveFile) void
updateHighScores() void
createNewMission() void
DefaultProgressionObserver
times int
action Callable<Void>
counter double
mission Mission
alert() void
alert(double) void
getProgression() double
reset() void
setMission(Mission) void
checkFinished() void
«create»
«create»
IMissionFactory
createMissionJumpOnSpring(int, Callable<Void>) Mission
MissionFactory
serviceLocator IServiceLocator
register(IServiceLocator) void
createMissionJumpOnSpring(int, Callable&lt;
AGameObject
HITBOX_LEFT int
HITBOX_RIGHT int
HITBOX_TOP int
HITBOX_BOTTOM int
HITBOX_SIZE int
LOCK Object
serviceLocator IServiceLocator
logger ILogger
hitBox double[]
sprite ISprite
xPos double
yPos double
addXPos(double) void
addYPos(double) void
checkCollision(IGameObject) boolean
getHitBox() double[]
getSprite() ISprite
getXPos() double
getYPos() double
render() void
setHitBox(int, int, int, int) void
setSprite(ISprite) void
setXPos(double) void
setYPos(double) void
getLogger() ILogger
update(double) void
getServiceLocator() IServiceLocator
ProgressionObservers
disappearingPowerup
equipmentPowerup
height
jetpack
jumpable
powerup
progression
propeller
sizeDown
sizeUp
springShoes
spring
trampoline
observers Set<IProgressionObserver>
myClass Class<?>
mySuperClass Class<?>
addObserver(IProgressionObserver) void
alertObservers() void
alertObservers(double) void
APowerup
perform(PowerupOccasion) void
Spring
BOOST double
collidesWith(IDoodle) void
getBoost() double
render() void
playSound() void
animate() void
Powered by yFiles

.