FIT2099 Assignment 3 – Work Breakdown Agreement

Group: Clayton Lab 19 Team 8

Members: Vedesh Appadoo, Shyam Prasad & Oskar Hosken

We choose creative mode, where requirements 4 and 5 will be new ideas by us rather than the provided requirements.

Requirement	Implementation	Testing & Reviewing	Dates to be completed
1: Lava Zone	Oskar	Everyone	22/05/2022
2: More Allies and Enemies	Shyam and Oskar	Everyone	22/05/2022
3: Magical Fountain	Vedesh	Everyone	22/05/2022
4 C1: - Treasure Room	Oskar	Everyone	22/05/2022
5 C2: - Random trees, enemies, mystery block	Shyam	Everyone	22/05/2022

We agree to further divide work between us where needed, due to certain tasks requiring more effort and time than others, that was not foreseen. Upon further work division, this WBA will be updated, and resigned once all parties agree to the specifications.

Team Members contribution:

Vedesh Appadoo (33%)

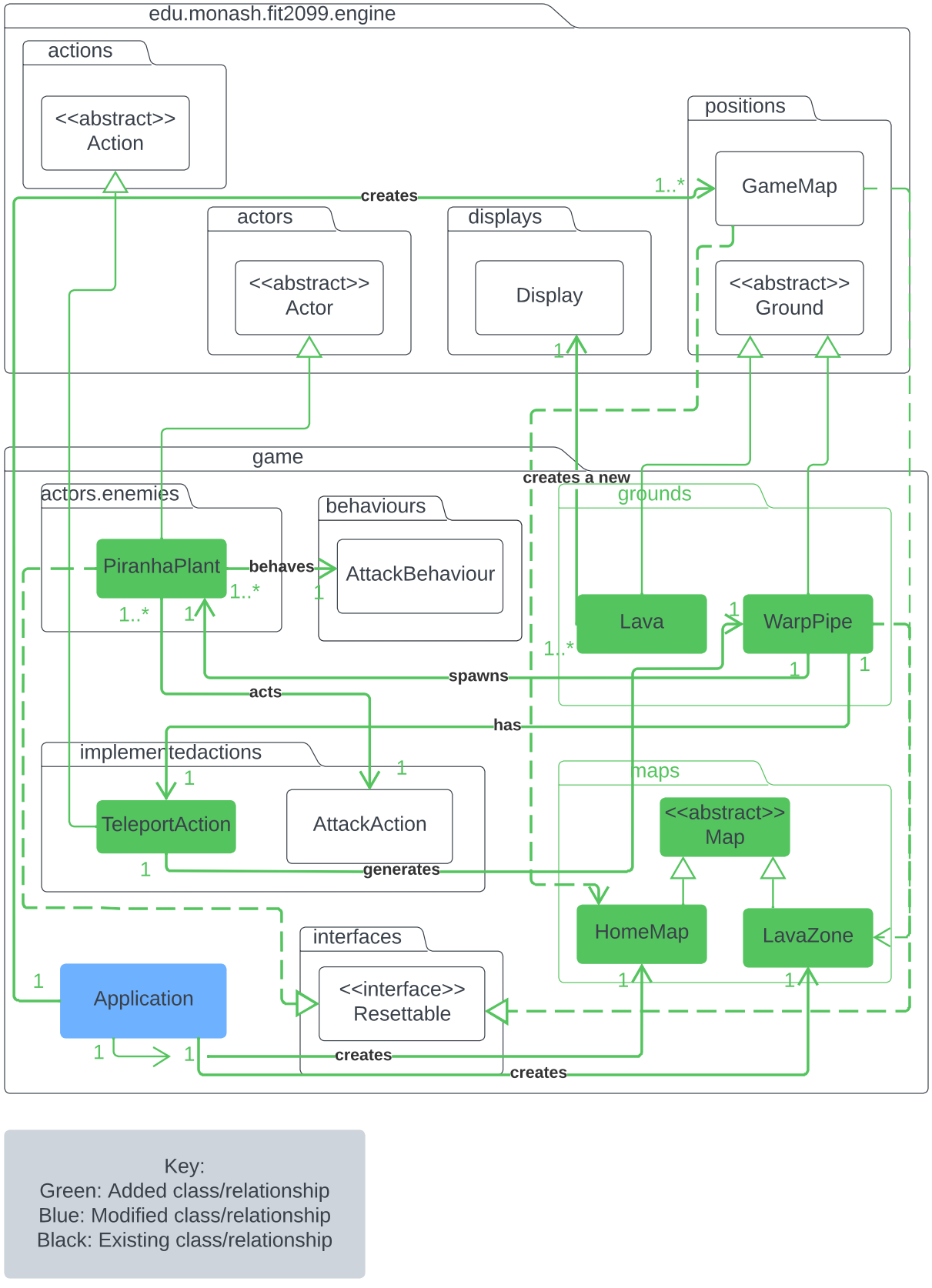
Shyam Prasad (33%)

Oskar Hosken (33 %)

I agree to this WBA – Shyam

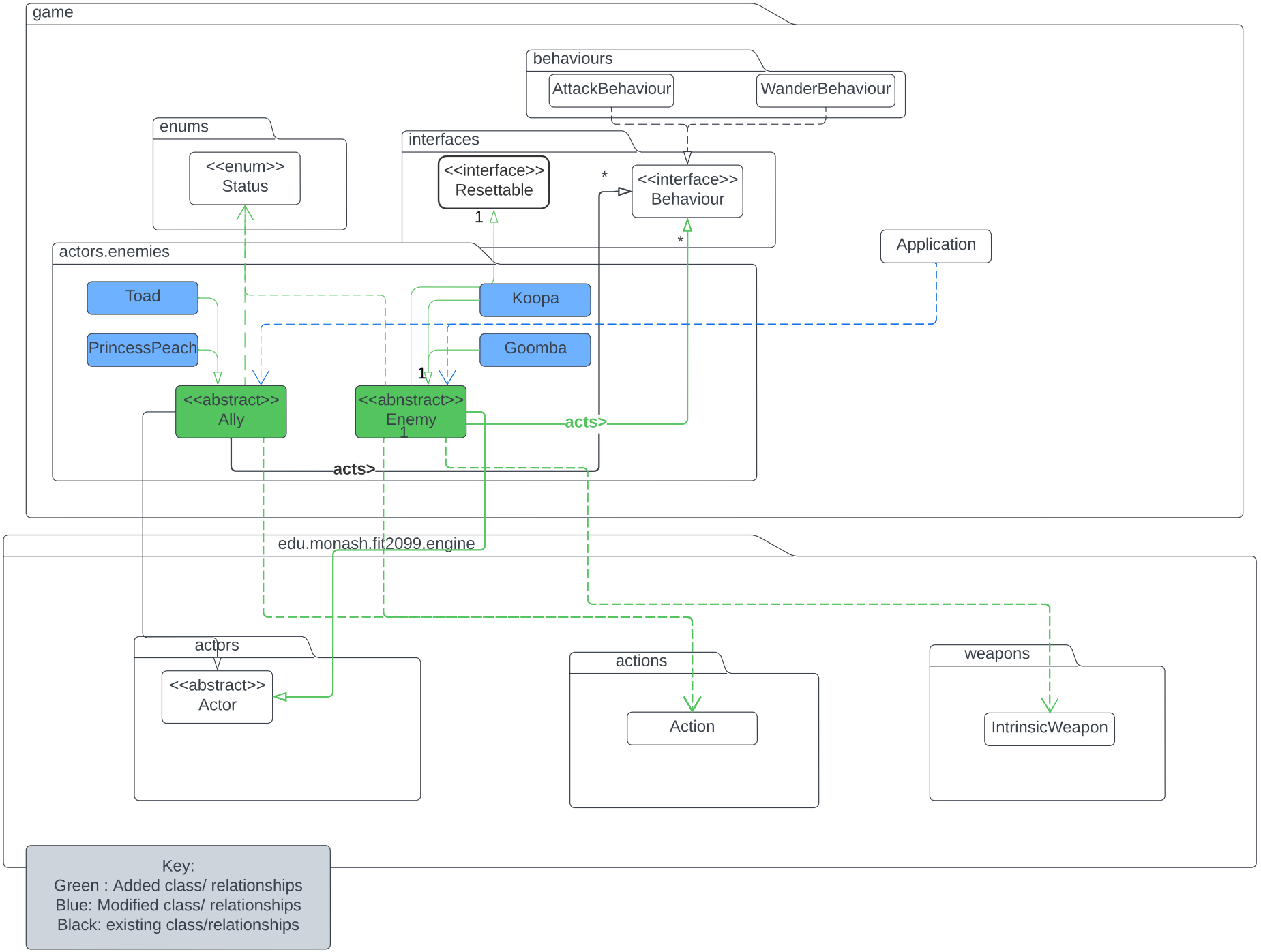
I agree to this WBA – Oskar

I agree to this WBA – Vedesh

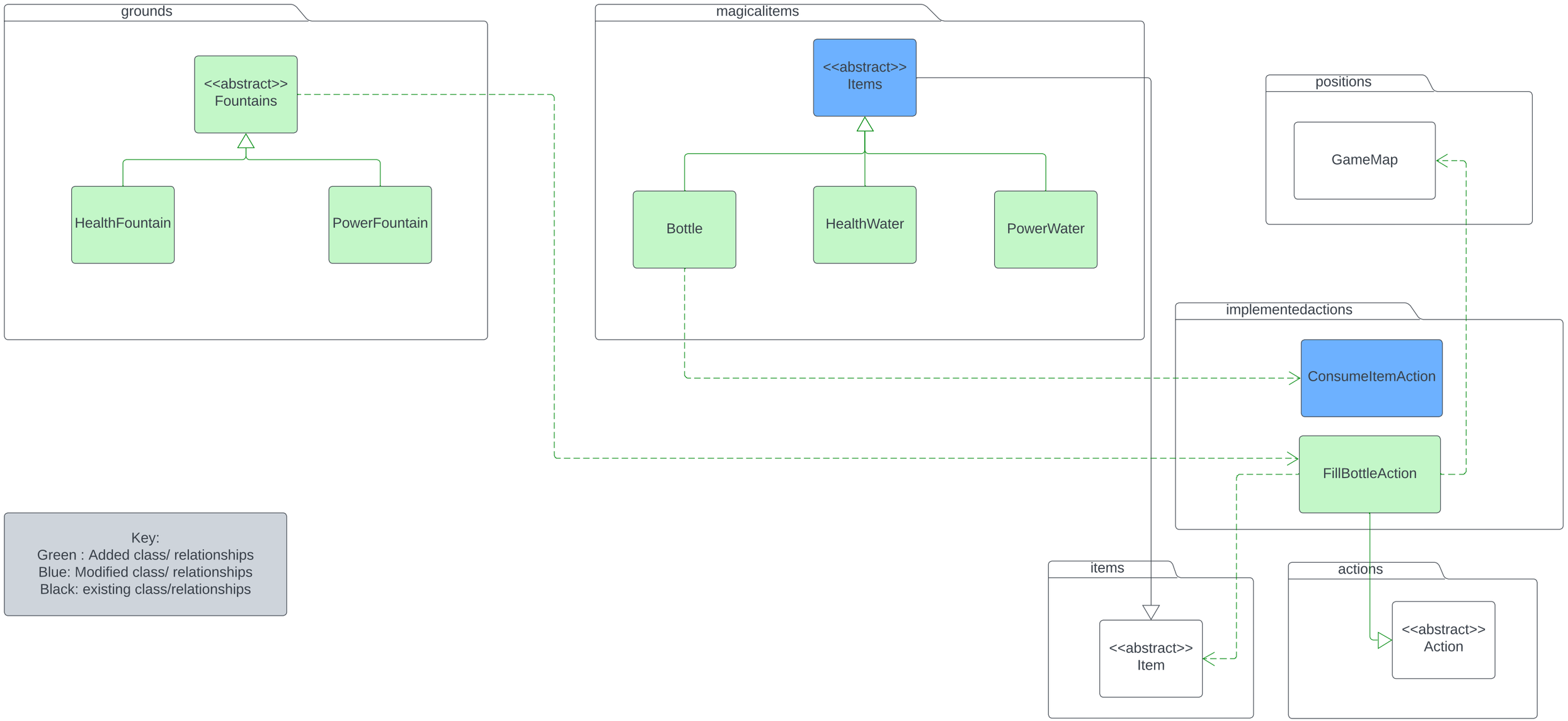


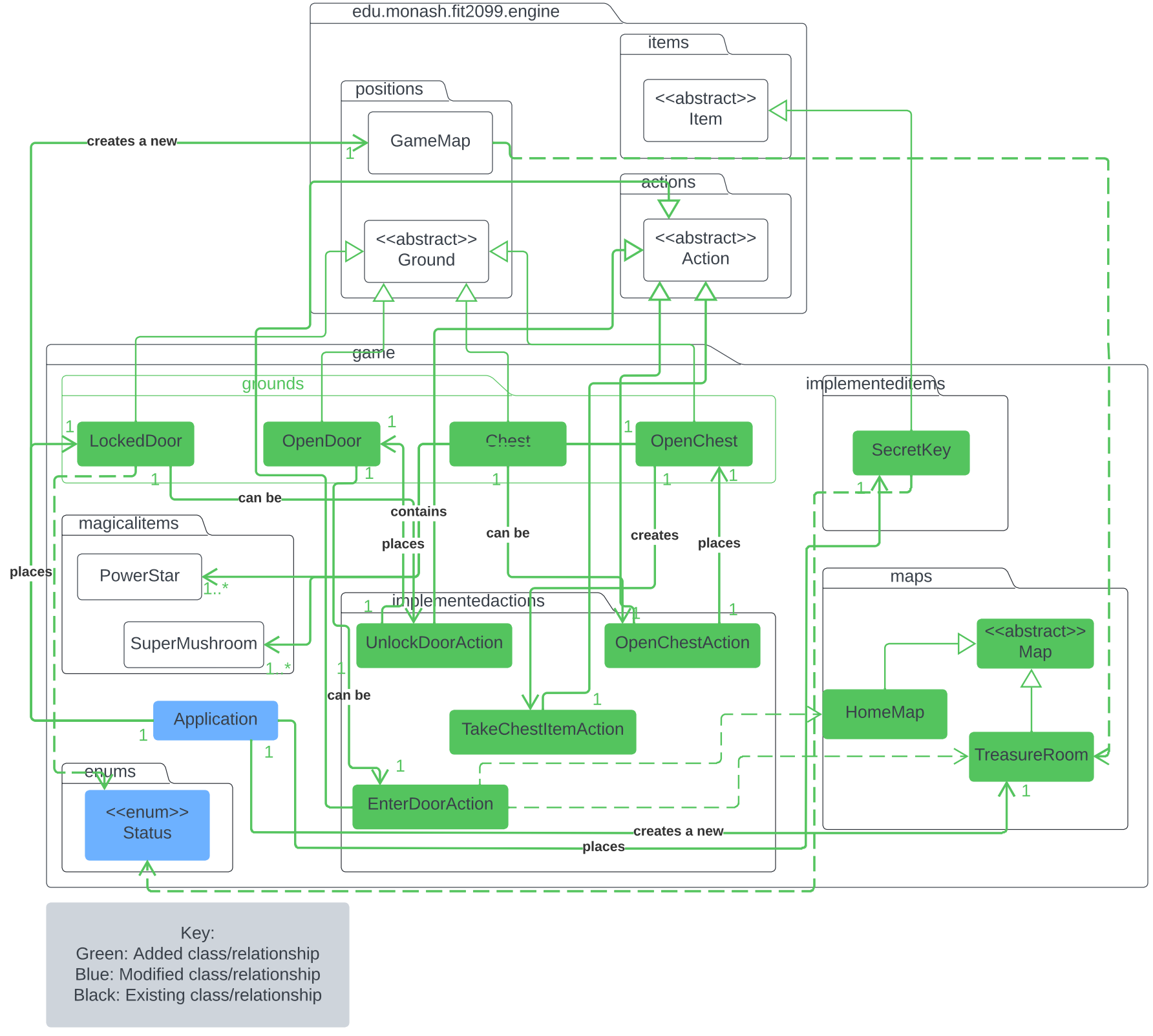
Key:
Green: Added class/relationship
Blue: Modified class/relationship
Black: Existing class/relationship

FIT2099 - Assignment 3 - Requirement 2 UML Class Diagram

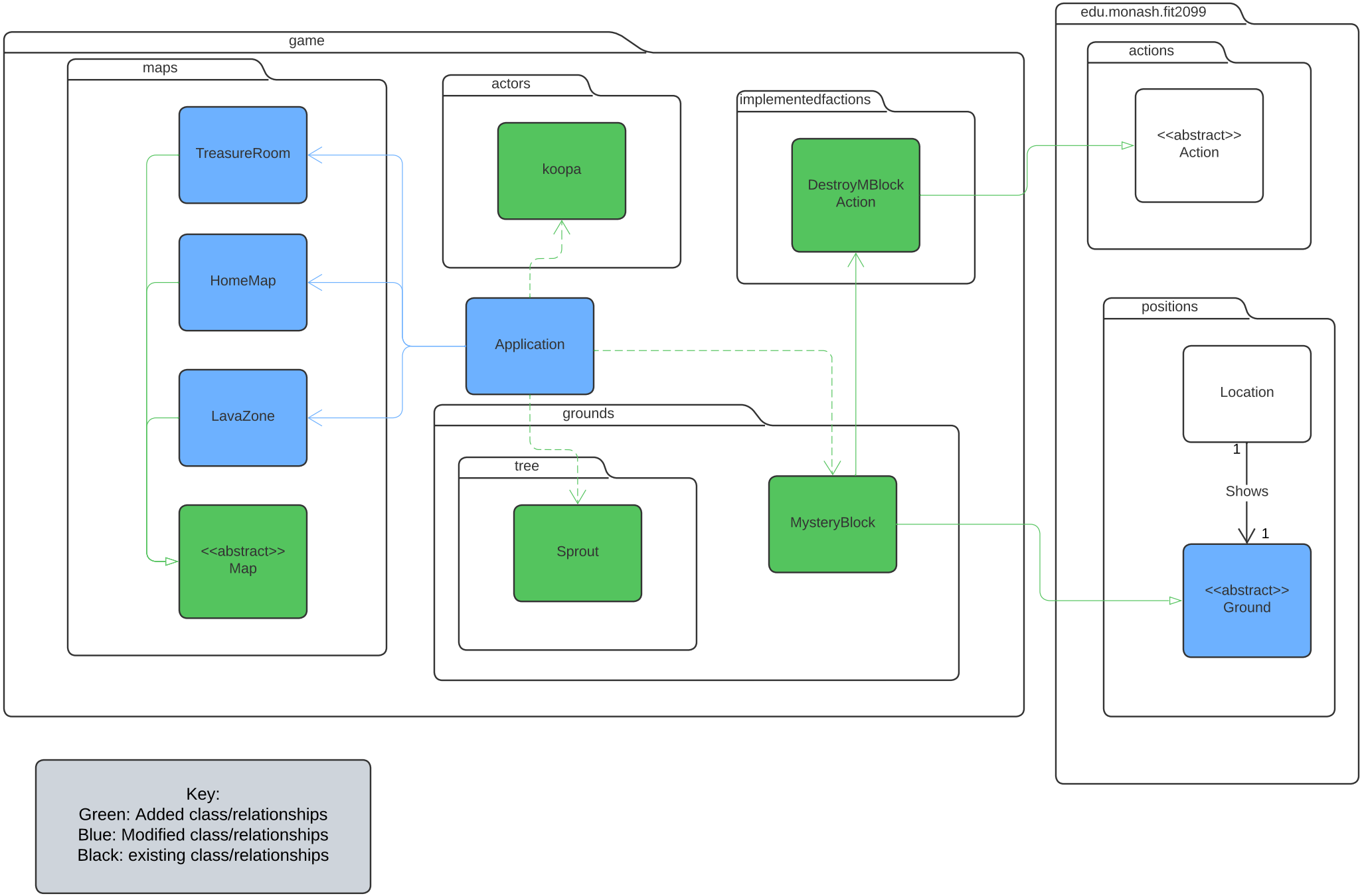


Requirement 3: Magical Fountains




edu.monash.fit2099.engine
items
<<abstract>>
Item
positions
GameMap
creates a new
1
actions
<<abstract>>
Ground
<<abstract>>
Action
game
grounds
implementeditems
LockedDoor
OpenDoor
Chest
OpenChest
SecretKey
1
can be
contains
places
can be
creates
places
magicalitems
PowerStar
1..*
SuperMushroom
1..*
places
implementedactions
UnlockDoorAction
OpenChestAction
maps
<<abstract>>
Map
HomeMap
TakeChestItemAction
TreasureRoom
Application
can be
enums
<<enum>>
Status
EnterDoorAction
creates a new
places
Key:
Green: Added class/relationship
Blue: Modified class/relationship
Black: Existing class/relationship

FIT2099 - Assignment 3 - Requirement 5 UML Class Diagram



Key:
Green: Added class/relationships
Blue: Modified class/relationships
Black: existing class/relationships

Task 1 Design Rationale

Oskar Hosken - 31450628

Creating a new abstract class called Map and inheriting all the maps (as a list of strings) from it:
In order to move between maps and have the name of each map show in the display of the game, I thought it was necessary to create an abstraction that allows all game maps to have a menu description per se. I have named the abstract method as menuName(). This allows for the future Teleport Action to display as “Mario teleports to ‘menuName()’”.
This also cleans up some of the code in the Application class and enhances the Single Responsibility Principle in our code as the Application class is no longer used to initialise the maps. We instead have a new abstraction to take care of that.

Creating LavaZone as a new map:
The LavaZone map is a smaller, harder map. I chose to make the map roughly half the size of the original as it saves some time for the player as they don’t have to move as far to get to Bowser for instance.
I added a lava lake in the centre with a bridge over the top purely for looks and randomly placed the Lava on the map. In the top left corner is a little arena for Bowser and Peach, with the bottom right dedicated to a safe zone for the player to get away from Bowser as he cannot walk on the Floor() ground type.
Additionally, the top left is a little room for the warp pipe.

Creating a WarpPipe ground:
Firstly, this extends the Ground class because it can be jumped onto and entered, it cannot be picked up or consumed.
The WarpPipe was difficult to implement. It takes the map it is on, the map it leads to, the location it is at and the location it leads to as parameters when being instantiated. It uses these parameters for the teleport action associated with it.
I used the tick() method to make sure that the Piranha Plants don’t spawn on the first round, but the second. This is simply done by using a counter and when it is incremented after the first turn, the Piranha Plant is placed on the Warp Pipe.
Next, if the Warp Pipe doesn’t have a Piranha plant on it, but has an actor on the location and the actor hasn’t been on that location before, we add a new teleport action that goes to the Lava Zone (if it is on the original map and vice versa). I then set a flag to true to indicate that we do not need to add another teleport action here if the actor decides to not go down the pipe.

Additionally, if the warp pipe doesn't contain any actors (piranha plant or player) we add a new jump action available for the player to use.
And finally, to reset the warp pipe we simply add the piranha plant back on top of the pipe. Note that I have created one instance of a piranha plant per warp pipe. This avoids us creating multiple instances of piranha plants for each warp pipe.

Creating a Teleport Action:
The teleport action is necessary for the player to go between maps and to enhance the experience of going down a warp pipe.
This action simply takes the same parameters as a warp pipe and uses the gameMap.moveActor method to move the actor between maps as if they are teleporting. If the player is teleporting to the Lava Zone, I create a warp pipe at the top left corner (where they teleport to) that teleports the player back to the original warp pipe they came from. This does create multiple instances on top of one another on the map but it was the only way I was able to make sure the player teleports back to the same warp pipe they came from.

Creating the Piranha Plant:
As for the Piranha Plant. It acts very similarly to every other enemy, except it cannot move.
I overwrote the getIntrinsicWeapon method, giving it 90 damage with the verb 'chomps'.
When it is reset, we simply increase its max HP by 50 and then set its HP to that number.
Each turn we check whether the player is nearby and if so, it will attack. Otherwise, it will do nothing and stay on the warp pipe.

Design Rationale for Requirement 2: More allies and enemies!

Explanation of system

This requirement involves the addition of numerous new allies and enemies classes, each with unique functionality.

Choice

There should be a new abstract class enemies, and another new abstract class allies created.

Justification

This will make code easily extensible for the future, as adding new types of enemies or allies will become much easier, therefore supporting OCP.

Choice

Making a BaseKoopa abstract class and having both the original Koopa and the new Flying Koopa extend it.

Justification

This enables both koopa classes to pull their common/shared functionality from the BaseKoopa class, therefore supporting LSP as the subclasses become substitutable with the superclass. This also supports ISP as no functions that will be inherited from the BaseKoopa superclass that aren't needed by the child Koopa class.

Design rationale for Requirement 3: Magical Fountains

Fountains, PowerFountain and HealthFountain classes

An abstract class Fountains was created and extended from Ground. This is because fountains has a lot of similar functionalities to Ground. Fountains was made into an abstract class so that new types of fountains can be created in the future, and it allows scalability. PowerFountain and HealthFountain both extend Fountains as they all share similar functionalities.

Bottle, HealthWater and PowerWater classes

A class Bottle that extends Item was created. A static variable/stack bottle was created; static is used because the variable/stack bottle can be accessed without instantiating the class Bottle, which prevents the use of 'instanceof' and casting. HealthWater and PowerWater both extend Item as they share similar methods and features.

FillBottleAction

FillBottleAction is just like any similar implemented action, and it extends Action. It uses the player's location and ground to check whether the player is on a PowerFountain or HealthFountain. It creates an appropriate instance of either a PowerWater or a HealthWater and adds it to the player's stack bottle.

Task 4 (Creative) Design Rationale

Oskar Hosken - 31450628

Creating a TreasureRoom Map:
For this creative task, a new mysterious room is needed to be added. I decided to make a new game map as this adds to the mystery of the task as the player cannot see the actual room on their map.
This treasure room is very small and simply contains a room full of floors, a chest, an open door and some coins scattered about the place.
The idea behind the implementation of this room was to give the player a small side mission that they can complete in order to get an advantage when it comes to fighting bowser or other enemies.

Adding a secret key item:
The secret key item further enhances the mysterious and difficult nature of the task. Adding an item to unlock the door is simply there to make it engaging and slightly more difficult as the player will need to find the key to unlock the locked door on the map.
I extended item as it is something that can be picked up and stored in an inventory, much like a real life key, so inheriting it from Item seemed necessary.
The secret key simply grants the player a capability that allows them to unlock the locked door. It doesn't have any other features.

Hiding the secret key until the player gets nearby:
This aspect of the game was implemented in the Player class. Hard-coding the coordinates of the secret key limited the complexity of the task and allowed me to access those coordinates in the Player class. It is made in this class because I couldn't access the Player's coordinates in the secret key class itself.
Besides, the way this works is that I created a new method called getDistance(). This method takes the coordinates of the key and the location of the player and calculates the distance between the two using Pythagoras' Theorem. If this distance is less than or equal to 5 blocks, the key shows up on the map. If the player is more than 5 blocks away, it disappears. This is simply done by using the addItem() and removeItem() methods.

Adding a hint to Toad's monologue:
I thought it would be fun to add a clue to this side mission into Toad's dialogue, because, without it, the player would only find the key by stumbling upon it accidentally. I thought it would be nice to add one clue, so that the player is aware that the key is hidden somewhere. This is done simply by adding in another line that Toad can say, and by adding a couple of

conditions to it so that Toad doesn't say that line when the player has the key or has picked it up before.

Adding various game-enhancing features like unlocking the door and opening the chest:

These features I added include a LockedDoor, OpenDoor, Chest and OpenChest. While all of these features are not 100% necessary for the game to function, I thought it would be more engaging for the player to be able to unlock the door and have it change look and the same with the chest.
The Locked Door is a child class of Ground as it cannot be picked up or consumed and is something that can be entered. Like the warp pipe.
It is on the map, locked until the player approaches it with the capability TREASURE_KEY. If they have the capability, an unlock door action appears for the player. This unlock door action allows the player to change the door from being locked ( ]), to being open ([ ). When the door is open, the player will have an available action called EnterDoor. This acts like the teleport action, except it is simpler and has a menu description of "Mario enters the door". I added this because it seems more realistic and it's fun to have some specific features and actions for this task.
When in the treasure room, the player can open the chest (=). Once opened, the player will have the opportunity to take items from it by using the TakeChestItemAction. This action differs from other similar ones like PickUpItemAction as it removes the item from the Chest's inventory, not the location. The Open chest action also changes the chest from closed (=) to open (-).
This is simply to add to the engagement of the game and to give the player a slight advantage when it comes to fights with the enemies.

Design Rationale for Requirement 5: Randomised Generation

Explanation of system

Randomly generates trees, enemies, and mystery blocks on world creation, rather than having them in fixed, hard-coded locations. This also includes creating the mystery blocks themselves as they were not an existing feature from Assignment 1/2 or Assignment 3 requirements. Mystery blocks can broken/opened by Mario using a Wrench, and will drop some rewards, such as items or coins.

Design choices ie. Choices made to follow SOLID principles

Single Responsibility Principle:

We will have a function for various objects of a specified type, mainly actors or grounds. These functions will be in the map class, as they will be used on only map objects, and are not needed at a higher or lower level. This adheres to the SRP as the functions which relate to maps are being added to the maps class, which maintains the single purpose of the maps class, which is to create and manage maps. Additionally, the mystery blocks will be made as a new type of ground, with information such as the items contained stored the new class. This also supports SRP as all of the information and functions needed for the mystery blocks to function will be in the one class, with no extra unrelated functionality in the class, and no mystery block specific functionality outside of it.

Open/Closed Principle:

As hinted at previously, the random generation function will be made such that they accept a parameter for the type of object (subclasses of actor or ground) to generate, so that they can be easily extended for new objects added in the future, therefore supporting the OCP. Additionally, the mystery blocks will extend the already existing ground class, and be implemented using already extensible code, therefore maintaining OCP. The blocks should also be able to give out various types of rewards, particularly items, which will allow any future items to be easily added to the 'loot pool'.

Liskov Substitution Principle:

LSP is followed as mystery blocks will be implemented by extending the already existing ground superclass, and therefore must include the necessary ground traits. Therefore, the mystery blocks must be replaceable by any other type of ground for the application to work.

Interface Segregation Principle:

This implementation should not require the addition of any new interfaces as there are not other similar classes to the mystery blocks that require similar functionality outside of the base ground functionality which is used by all ground subclasses in its entirety.

Dependency Inversion Principle:

The DIP is adhered to as the mystery blocks class will extend the ground superclass, which is also extended by other subclasses of ground that share some necessary functions. The mystery block specific functions will remain in only the mystery block class as there are no other classes that require this unique functionality.

Mario collecting and drinking health water from bottle - Sequence Diagram