FIT2099 Assignment 1 Design – Work Breakdown Agreement

Requirement	Implementation	Testing & Reviewing	Dates to be completed
1: Tree	Shyam	Vedesh & Oskar	08/04/2022
2: Jump	Oskar	Shyam & Vedesh	08/04/2022
3: Enemies	Vedesh	Shyam & Oskar	08/04/2022
4: Magical Items	Oskar	Shyam & Vedesh	08/04/2022
5: Trading (Toad)	Shyam	Vedesh & Oskar	08/04/2022
6: Toad speaking	Vedesh	Shyam & Oskar	08/04/2022
7: Reset	Shyam	Vedesh & Oskar	08/04/2022

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. The specifics of further work division will be noted in a separate document upon determination by all parties.

Team Members contribution:

Vedesh Appadoo (33%)

Appadoo (33%)

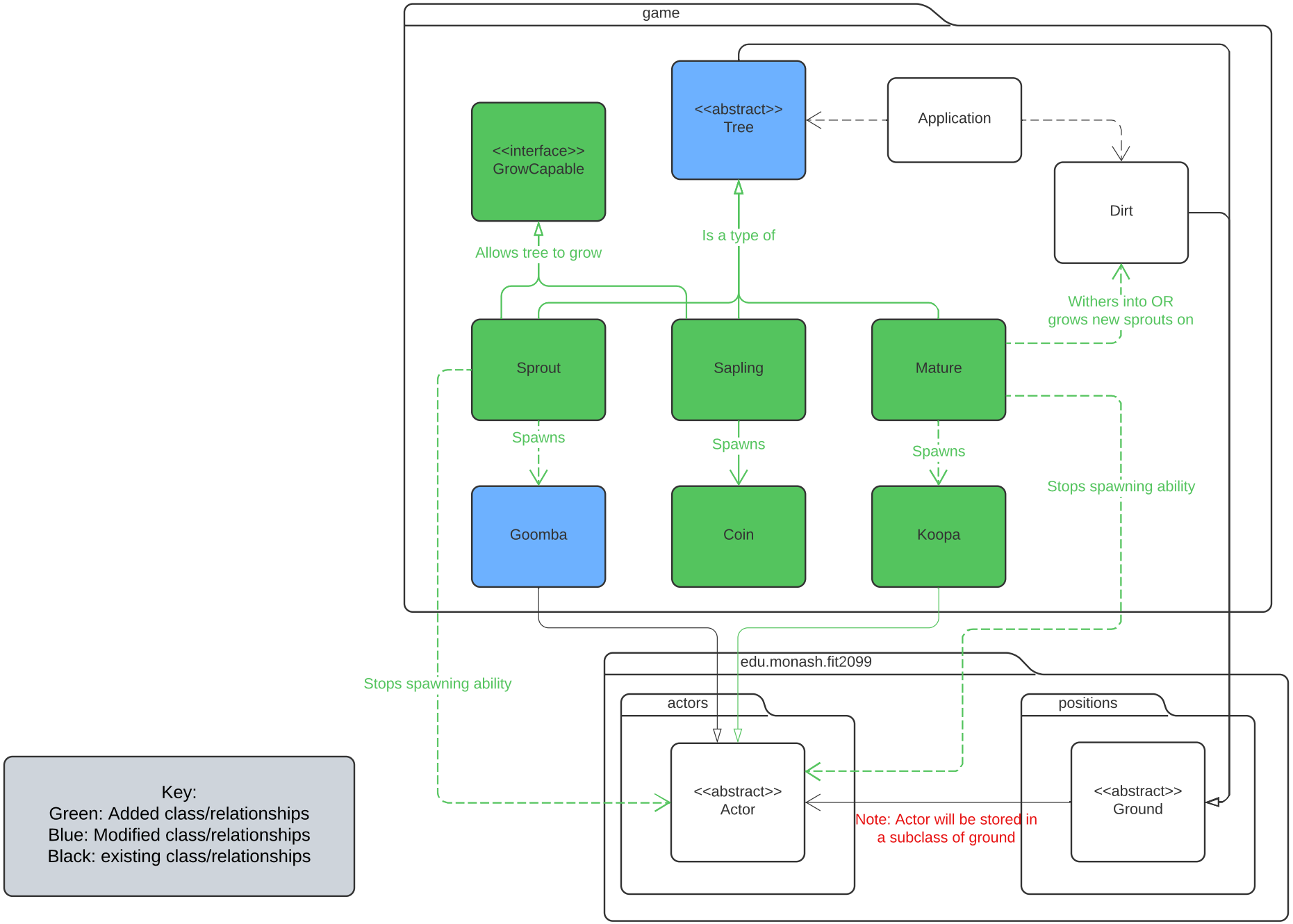
Oskar Hosken (33 %)

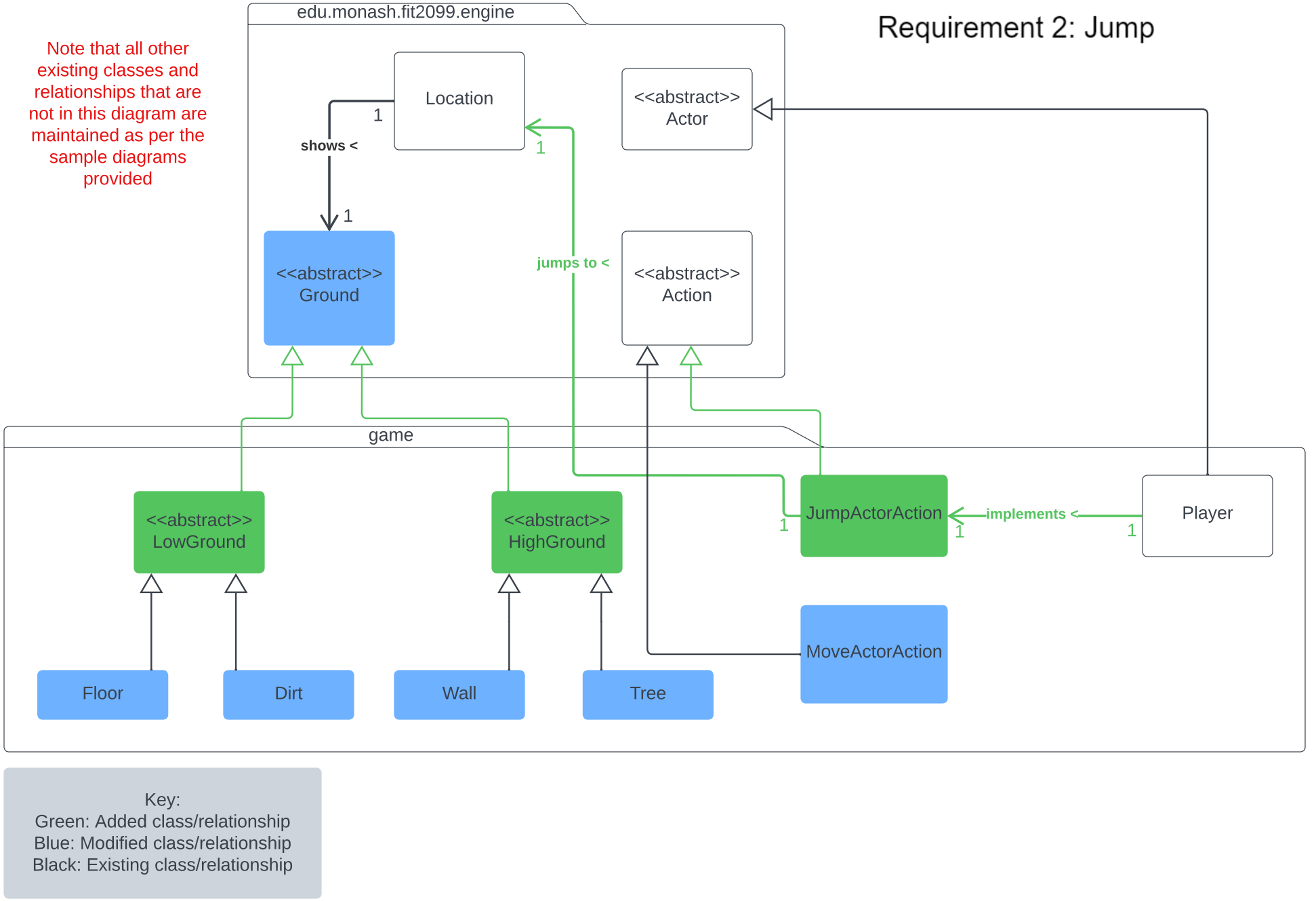
I agree to this WBA – Oskar

I agree to this WBA – Shyam

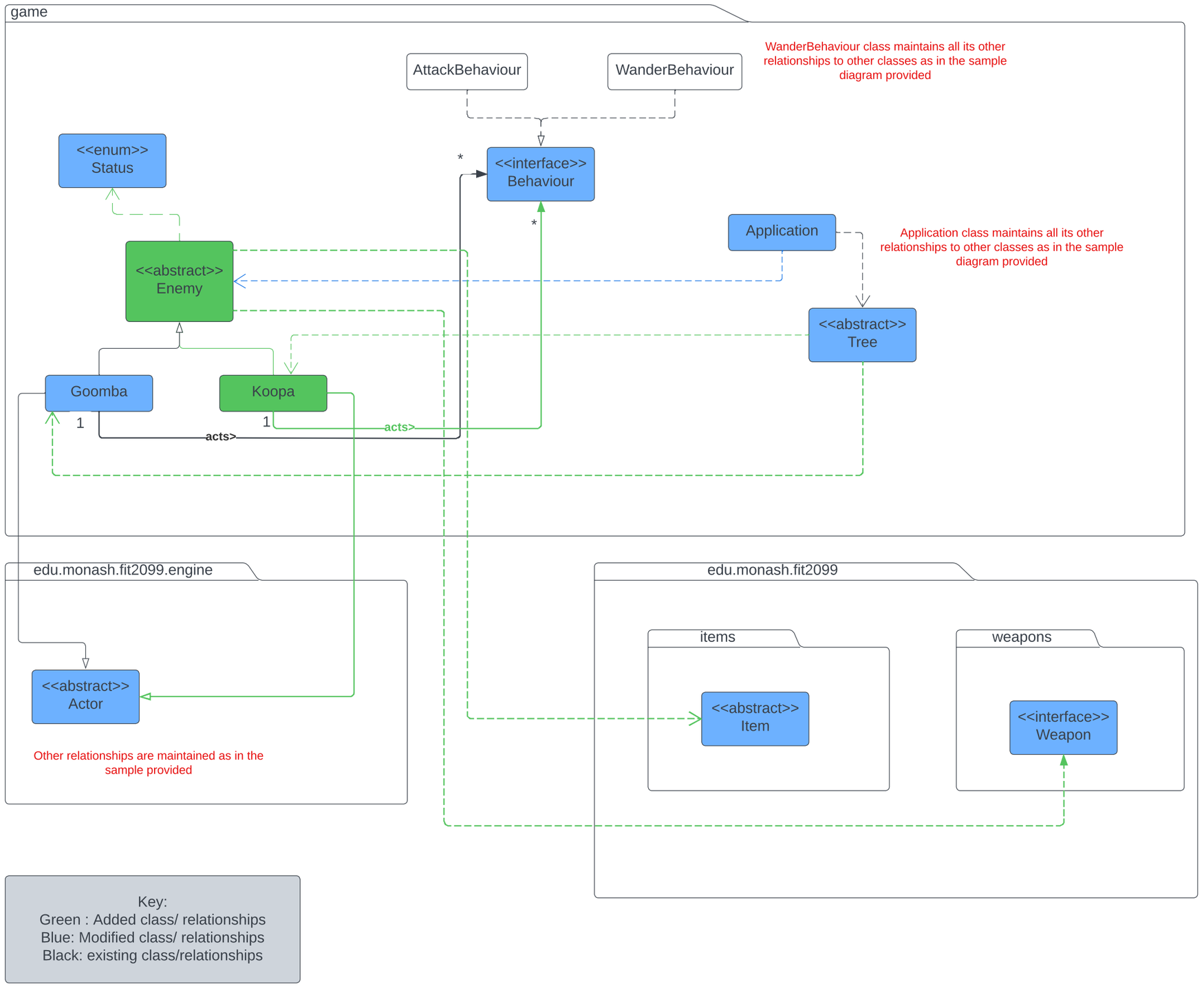
I agree to this WBA - Vedesh

FIT2099 - Assignment 1 - Requirement 1 UML Class Diagram

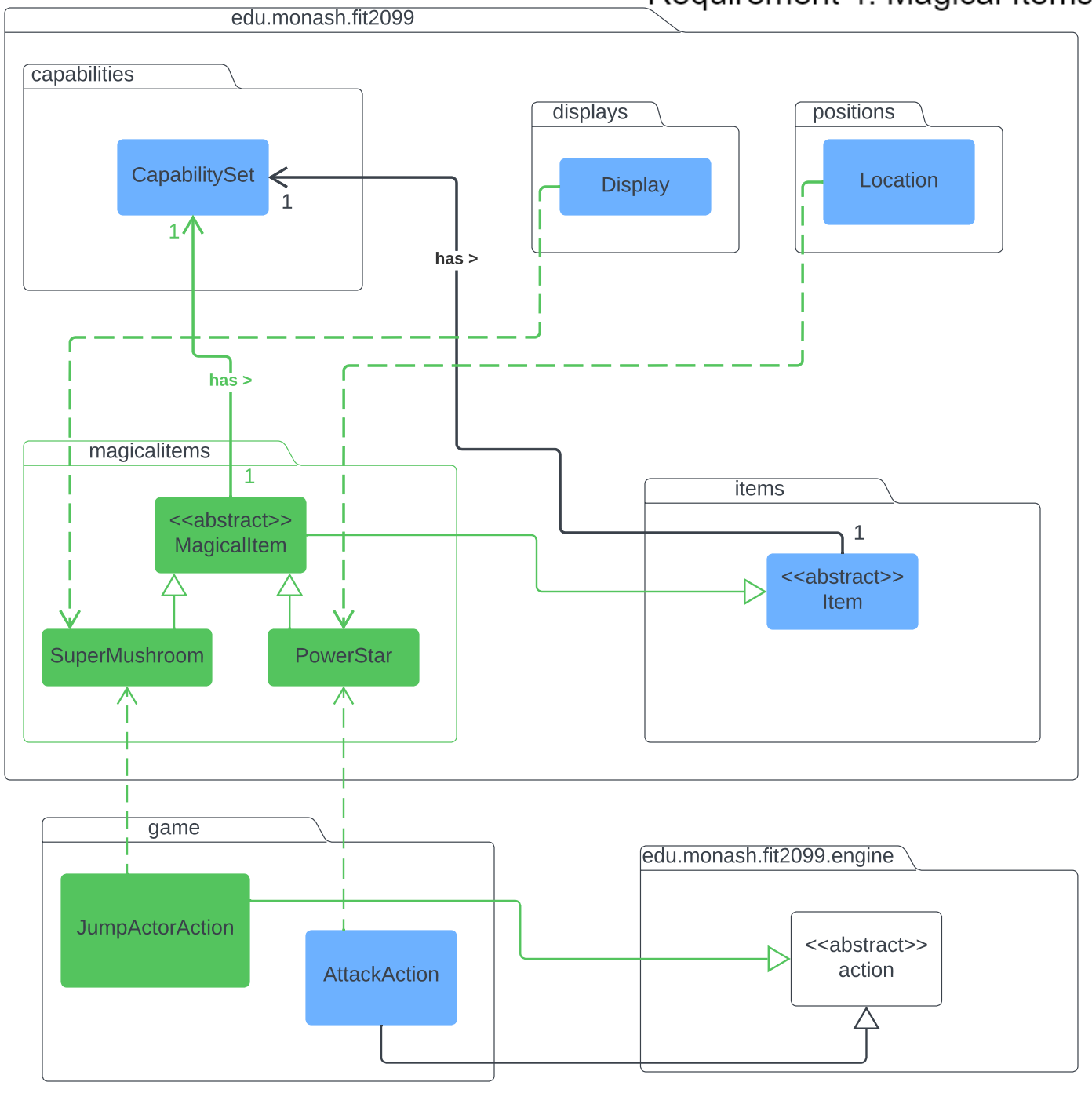



Requirement 2: Jump
Note that all other existing classes and relationships that are not in this diagram are maintained as per the sample diagrams provided
edu.monash.fit2099.engine
Location
<<abstract>>
Actor
1
shows <
1
1
<<abstract>>
Ground
jumps to <
<<abstract>>
Action
game
<<abstract>>
LowGround
<<abstract>>
HighGround
1
JumpActorAction
1
implements <
1
Player
MoveActorAction
Floor
Dirt
Wall
Tree
Key:
Green: Added class/relationship
Blue: Modified class/relationship
Black: Existing class/relationship

Requirement 3: Enemies



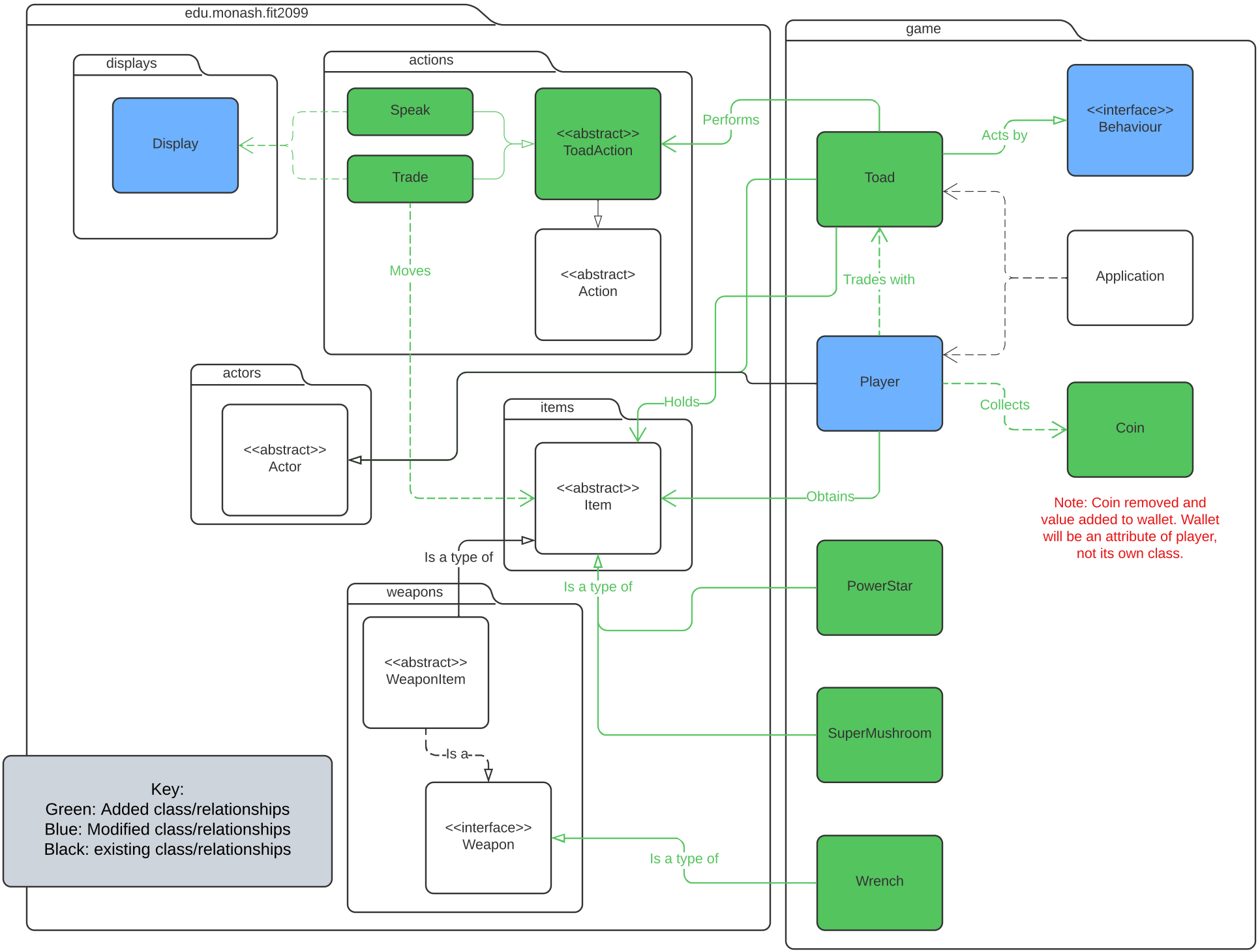
Requirement 4: Magical Items



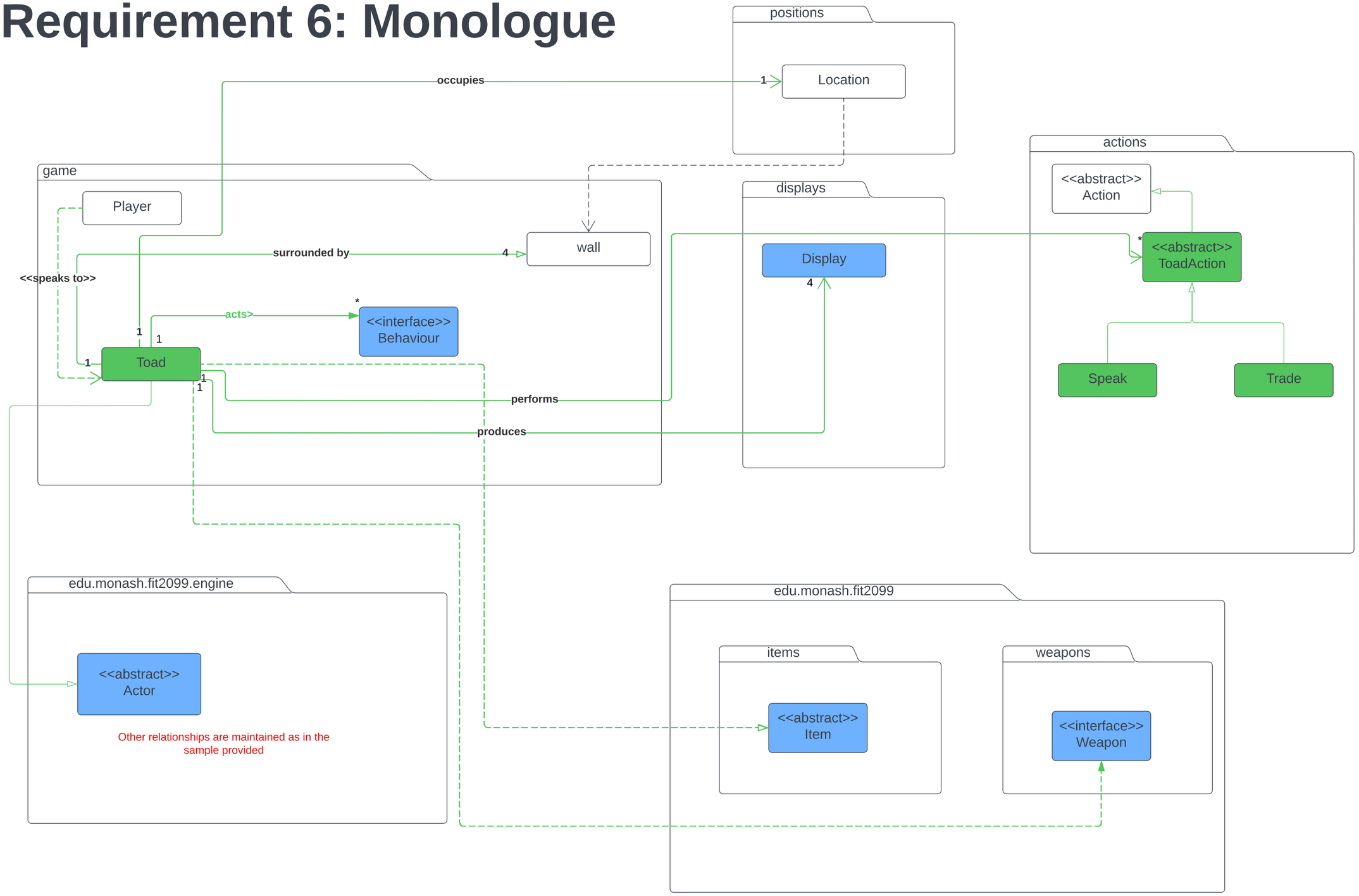
Key:
Green: Added class/relationship
Blue: Modified class/relationship
Black: Existing class/relationship

Note that all other existing classes and relationships that are not in this diagram are maintained as per the sample diagrams provided

FIT2099 - Assignment 1 - Requirement 5 UML Class Diagram

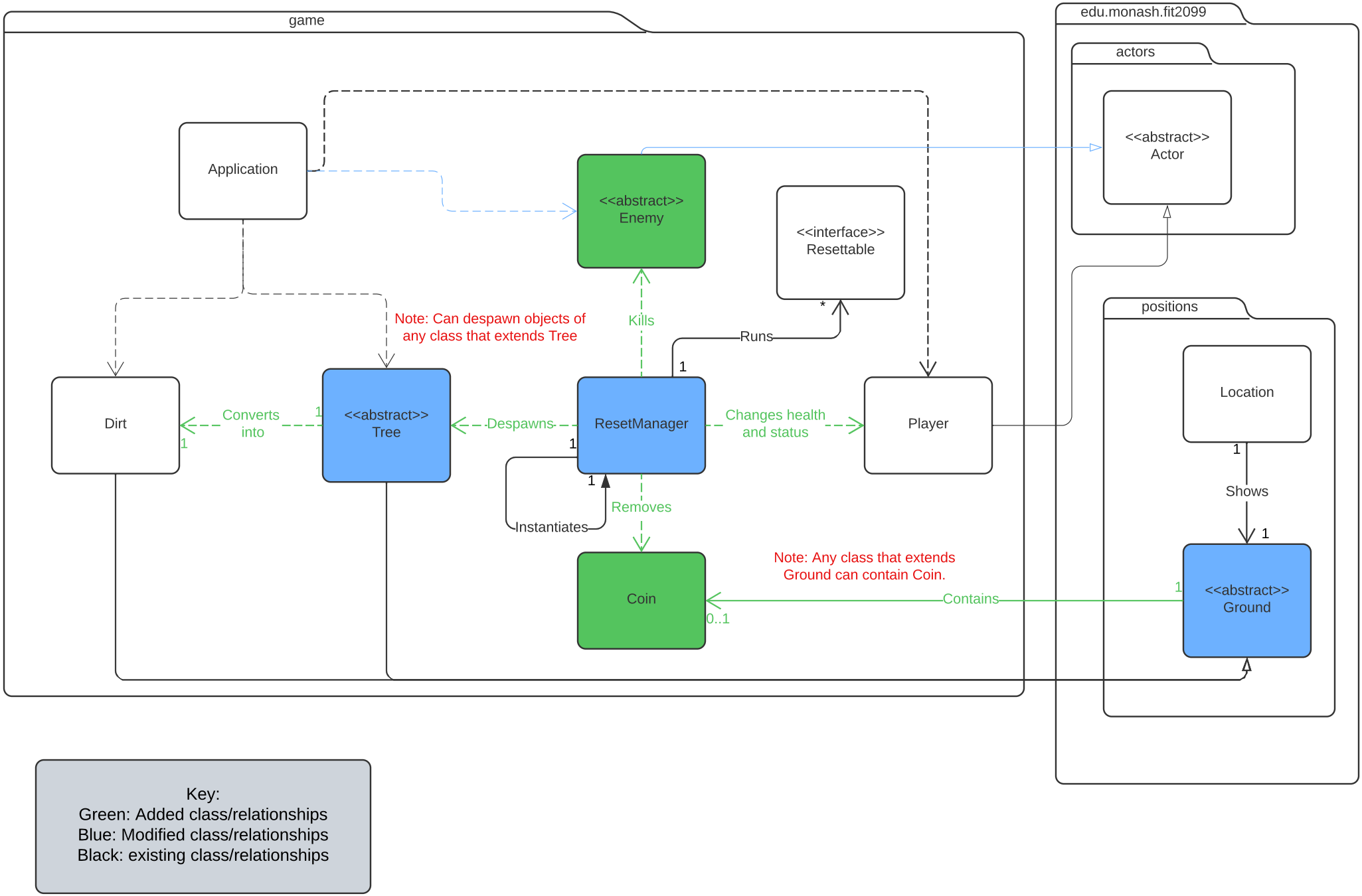


Requirement 6: Monologue



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

FIT2099 - Assignment 1 - Requirement 7 UML Class Diagram



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

Design Rationale

All changes made to the existing UML diagrams to get the submitted UML diagrams are highlighted and justified in this document. Each class has its own behaviours and serves different purposes. This implements Single Responsibility Principle (SRP). The pros are the code is simpler and shorter and the program is easier to maintain. However, repetition of code may occur, and this violates DRY principle.

In the assignment, we are inheriting attributes and methods from abstract class. By extending an abstract class, DRY principle is implemented. This means codes can be reused and can be shared between classes that have common traits. As a result, codes do not need to be re-written, resulting in a simpler and shorter codebase.

Design Rationale for Requirement 1: Let it Grow!

Explanation of system

This system controls the trees and their actions. We have 3 types of trees: sprouts, saplings, and matures. Their core action is growing: the sprout can grow into a sapling after 10 turns, and the saplings are able to grow into matures after 10 turns. The matures also spread a new sprout every 5 turns, but also has a 20% chance to wither and die each turn, turning into dirt. Additionally, the sprouts can spawn goombas at a 10% chance each turn, the saplings can drop coins at 10% each turn, and the matures are able to spawn koopas at 15% chance each turn.

- Making Tree an abstract class

  Justifications:

  - This enables the efficient creation of multiple different sub-classes of tree, namely the sprout, sapling and mature trees. Their common/shared attributes and functionality can be inherited from the base Tree class, which supports the Don’t Repeat Yourself (DRY) principle and reduces redundancy.

- Addition of the Sprout, Sapling and Mature sub-classes of Tree

  Justifications:

  - These new sub-types of trees will have different functionality that distinguish them from each other. Their shared attributes and functionality will be inherited from the parent class Tree.

- Using an interface GrowCapable to provide the functionality for certain trees to grow.

  Justifications:

  - This implements the DRY principle, as by using an interface we are reducing redundancy of defining the similar method for each class. Instead, they can each have their own implementation of the single method under the same function name. It also supports the Open-Closed Principle from the SOLID principles of object-oriented design, as by using this interface to create the grow function, we are opening the system for the extension of adding new types of trees in the future that may use this method but closing the existing types of trees for modification.

- Actors will be stored in locations

  Justifications:

  - We decided that we will track the actors' locations by moving the actor objects between different locations. This will include storing them in sub-types of locations, such as trees, dirt and floors. This will enable easy movement between locations, as well as helping provide the functionalities of stopping trees from spawning new actors when an actor is already standing on them.

Design Rationale for Requirement 2

- Abstract classes '**HighGround**' and '**LowGround**' inherited from abstract class 'Ground' added.
  Justifications:
    - This allows us to easily differentiate between which grounds can be jumped to or walked down to.
    - The HighGround objects will have slightly different characteristics than the LowGround objects and inheriting them respectively will make it easier for us to implement the 'Jump' action for our player.

- JumpActorAction class added.
  Justifications:
    - JumpActorAction is inherited from the abstract class action, as it is something the player can perform.
    - It does not associate with 'MoveActorAction' as we were unsure whether jumping counted as moving. And, moving can be forwards, backwards, left or right, whilst jumping is done vertically, moving up layers. Hence, making it its separate class was deemed necessary.
    - JumpActorAction alters the location of the player, similar to moving the player. They will be able to jump if they are next to the high ground and the location will change if the jump is successful. If the jump is unsuccessful, the location will not change.
    - I have associated this action with the Location class as it is the one that shows the current Ground that the player is on. If we associated it with HighGround or LowGround for instance, there wouldn't be anything that can show them.
    - I have also added an association between Player and JumpActorAction. This is because only the player can perform the JumpActorAction, not any enemies. Hence, I thought it was necessary to associate this action with the player specifically, not the Actor class.
    - Due to the JumpActorAction class being added, the MoveActorAction class needs to be modified to accommodate for both players and enemies moving to lower grounds. This modification will allow enemies to wander off high ground to lower ground and allow players to move from high ground to lower ground.

Design Rationale for requirement 3: Enemies

- An abstract Enemy class was created to encapsulate all types of enemies. Different relationships are implemented to fit the class Enemy in the system

  Justifications:
  - The abstract class Enemy was created as all types of enemies have similar behaviours and could potentially share methods. An abstract class would allow this and would prevent the developer from having to create the same methods multiple times.
  - The class Enemy is linked to the enum class Status to provide a status whether the enemy is alive or dead. The relationship is a dependency as enemy does not have an instance of Status as one of its attributes. By using enum, the code implements DRY principle. Since enum constants are final and static by default, they can be reused for every class. This helps us to reduce the workload of creating the same constants in every class that needs it.
  - Enemy class is linked to the abstract class Item via a dependency as the enemy instance's action changes depending on the items in the actor's inventory
  - Enemy class is linked to the interface weapon as Koopa instance need to know if it can be killed based on whether the actor (Mario) possess a weapon (a wrench)

- A new class Koopa was created, and several types of relationships are implemented for this class. (Includes changes made to abstract class Tree)

  Justifications:
  - This class was created to represent the other type of enemy called 'Koopa.' It shares similar methods and attributes to that of existing enemy type Goomba.
  - The class Koopa is also linked to the abstract Class 'Actor' as it is part of the game as an actor.
  - The class Koopa is linked to the interface Behaviour as it could potentially implement similar behaviour as other actors. By implementing an interface, related methods and attributes can be grouped together. An interface will ensure the child class implements these methods and attributes by overriding them. This would avoid careless mistakes like forgetting to write an important method for a certain class.
  - The abstract class Tree has a dependency to the class Koopa because it has a chance to create an instance of Koopa based on a specified probability and condition.

- The class Goomba has been modified to have additional relationships

  Justification:
  - The abstract class Tree has a dependency to the class Goomba because it has a chance to create an instance of Goomba based on a specified probability and condition.

- Application Class now has a dependency on the abstract Enemy class instead of having individual dependency on each type of enemies.

Design Rationale for Requirement 4

- Package '**magicalitems',** abstract class '**MagicalItem'**, and inherited classes '**SuperMushroom'** and **'PowerStar'** added. Alongside other relationships and modified classes.
  Justifications:
    - Magical items are similar to normal items but have different, special characteristics. This is why I thought it was necessary to inherit 'MagicalItem' from Item.
    - Furthermore, inherited classes SuperMushroom and PowerStar are added because they are 2 of the magical items specified that have their own special, unique powers and effects.
    - These are all maintained in their own package, 'magicalitems' because all of these classes are similar and this will allow us to easily manage and access the magical items.
    - MagicalItem has a one-to-one association with CapabilitySet. This is made under the assumption that the effects such as 'invincibility, instant kill and immunity' fall under capabilities. If not, a package and abstract class called 'Effect' can be added with those effects specified inherited from it. But I am under the interpretation that such effects fall under capabilities and hence the association between MagicalItem and CapabilitySet. Much like Item being associated with CapabilitySet.
    - Since this is the case, CapabilitySet will need to be modified slightly to accommodate for such effects granted by the magical items.
    - The Display class and Magical Item 'SuperMushroom' have a dependency relationship as the Display is changed when the actor consumes a SuperMushroom.
    - Similarly, Location and the Magical Item 'PowerStar' have a dependency relationship as whenever the actor walks on HighGround, the location will need to change it to dirt. The actor also does not need to jump to higher grounds which (due to assumption) the Location class can handle inside it.
    - JumpActorAction also has a dependency relationship with SuperMushroom as every jump with the SuperMushroom has a 100% success rate. This can be toggled in the JumpActorAction class.
    - As PowerStar grants the actor an 'instant kill' effect, a dependency has been added with the AttackAction class and the PowerStar class.

Design Rationale for Requirement 5: Trading

Explanation of system

This system enables the trading functionality, by which the player can interact with Toad, and trade coins for useful items. The system involves Toad's interactions, which can be to speak to the player, or trade with the player. It also shows that items can be held by Toad, and when traded for can be transferred to the player. (Note: items can also be found on the ground). It also shows that the player can collect coins while playing, and their value will be added to the player's wallet.

- Adding Toad as a new actor

  Justifications:

  - Toad is a friendly NPC actor who can be interacted with by Mario. He can speak to Mario, or trade with him.

- Giving Toad the interactions of speaking and trading

  Justifications:

  - Toad is given 2 actions, speaking with Mario or trading with Mario. These actions are their own classes, but are extended by the ToadAction class, which will include the shared attributes and functionality that all of Toad's actions need. This supports the Don't Repeat Yourself (DRY) principle as we are using inheritance and abstract classes to reduce the redundancy of code, as well as the Open-Closed Principle from SOLID principles, as future actions can easily be implemented by extending from the parent class ToadAction.

- The Trade function

  Justifications:

  - The trade function allows the player to trade coins to Toad, in exchange for a useful item. This is done by checking and subtracting an number of coins from Mario's wallet, and moving the item from Toad's inventory to Mario's or instantly using it.

- Movement and Storage of items

  Justifications:

  - Items can be stored in Toad's inventory, or Mario's inventory when traded for, or can be randomly found on the ground. Items can be moved between these 3 storages locations and can be used by the player when in Mario's inventory.

- Collecting and tracking of Coins

  Justifications:

  - Coin objects can be picked up from the ground or from sapling trees that spawned them. When picked up by Mario, the coin object will be removed, and the value of the coin will be added to Mario's wallet. The wallet will only be an attribute of Mario's player character, and so the coins are no longer necessary. We decided to implement the collection of coins this way as it is the most efficient since we do not have to store extra objects and does not lose any functionality as the coin objects are no longer needed after being collected.

Design Rationale for requirement 6: Monologue

- An abstract class ToadAction was created in order to regroup all the actions a toad can perform in a single class. Codes can be reused and shared. By using abstraction, methods that are static and necessary are created for all instances of the class.

  Justifications:
  - The class Speak allows the toad to speak different dialogues to the player and the class Trade allow a trade to be done.
  - These two above-mentioned classes have an inheritance to the abstract class ToadAction as both are relevant actions that any instances of Toad should/can perform.
  - The abstract class ToadAction has an inheritance to the abstract class Action because it can inherit and make use of several common actions that any actors can perform.

- A new class 'Toad' is created. The class contains all the attributes/methods that are to be used by instances of class Toad. Various types of relationships are implemented to fit the class in the system.

  Justifications:
  - The class Toad is also linked to the abstract Class 'Actor' as it is part of the game as an actor.
  - The class Toad is linked to the interface Behaviour as it could potentially implement similar behaviour as other actors. By implementing an interface, related methods and attributes can be grouped together. An interface will ensure the child class implements these methods and attributes by overriding them. This would avoid careless mistakes like forgetting to write an important method for a certain class.
  - The class player has a dependency to the class Toad as the player must speak to the Toad to initiate the conversation and the class Player does not have any attributes of type Toad, hence dependency.

- Toad class is linked to the abstract class Item via a dependency as the Toad instance's action changes depending on the items in the actor's inventory. (Whether the player has Power Star effect active in his inventory)
- Toad class is linked to the interface weapon as Toad instance need to what to say based on whether Mario has a wrench.
- Toad class has an association to the class Location as the former uses an instance of the class Location to determine a location appropriate to place the toad. The location class further has a dependency on the class Wall as the toad needs to be placed within walls.
- The class Toad has an association on the class Display as it needs to output whatever he is saying to the output window. As a result, it will need to have an instance of the class Display as its attributes in order to access the relevant methods to display the output.
- The class Toad has an association to the abstract class ToadAction. Using an abstract class means that DRY principle is implemented, and codes can be reused. This is done as both actions of a toad are similar and relevant to an instance of toad.

Design Rationale for Requirement 7: Reset Game

Explanation of system:

This system enables the game reset functionality, which the player can use once per playthrough, to clear an overwhelming map. Resetting the game will have the following effects:

- Trees have a 50% chance to be converted back to Dirt
- All enemies are killed.
- Reset player status (e.g., from Super Mushroom and Power Star)
- Heal player to maximum
- Remove all coins on the ground (Super Mushrooms and Power Stars may stay).

- ResetManager class

  Justifications:

  - This class is the core class that upon being called, resets the various parts of the game. This class will call methods that access data from across the system to reset them as needed.

- Resettable interface
  Justifications:
  - This class is used in order to track whether the game has already been reset or not. If the game has been reset before, it cannot be reset again.

- Removing trees
  Justifications:
    - When the reset is called, all trees have a 50% chance to be converted back to dirt. This includes any sub-class of the Tree abstract class, namely sprouts, saplings, or mature trees.

- Removing coins and killing enemies
  Justifications:
    - These are done with quite standard implementation. Any coins on the map when reset is called are removed. The player's coins, however, are maintained. Any enemies on the map are instantly killed/removed.

- Changing player health and status
  Justifications:
    - When reset is called, the player's current health should be reset to its maximum value. This may differ from the starting value, if the player has increased their maximum health through items such as the Super Mushroom. Any Status effects on the player at the time of reset, should also be cleared. This includes buffs from the Power Star, or Super Mushroom.

Class Descriptions for Requirement 1: Let It Grow!

- Class: <<abstract>> Tree

Description: The original tree class is modified to be abstract, and it will contain all of the common attributes and methods that all sub-classes of tree will need.

- Class: <<interface>> GrowCapable

Description: This interface will contain the method(s) that enable certain trees (sprouts and saplings to grow into larger trees).

- Class: Sprout

Description: The smallest variant of tree. 10% chance to spawn a Goomba each turn, unless an actor stands on it. Takes 10 turns to grow into a sapling tree.

- Class: Sapling

Description: The medium tree variant. 10% chance to drop a coin each turn. Takes 10 turns to grow into a Mature tree.

- Class: Mature

Description: The largest tree variant. 15% chance to spawn a Koopa each turn, unless an actor stands on it. 20% chance to wither and die, being turned into dirt each turn. Can grow a new sprout nearby every 5 turns.

Class Descriptions for Requirement 2:

- Class: LowGround (abstract)

This class is responsible for ensuring Floor and Dirt have similar characteristics that define them as 'low ground' for the game. This could include that they can be moved down to from higher grounds.

- Class: HighGround (abstract)

This class is responsible for ensuring Wall and Tree have similar characteristics that define them as 'high ground' for the game. This may mean that they can be jumped to and sit higher than the lower grounds.

- Class: JumpActorAction (abstract)

This class is responsible for the 'jump' movement that the player can do to move from low ground to high ground. it changes the location of the player and is a unique action to the player only. Enemies cannot access this action.

Class description for Requirement 3: Enemies

- <<abstract>> Enemy:

Description: The abstract enemy class has all the attributes and methods that are common for all types of enemies (i.e Goomba and Koopa)

- Koopa:

Description: The Class Koopa represents the enemy of type Koopa. It contains attributes/methods that are inherited from the class Enemy as well as methods/attributes specific to its class

Class description for Requirement 4:

- Class: MagicalItem (abstract)

This is the parent class for all magical items. It is inherited from Item and is responsible for defining methods and attributes that are shared by both magical items SuperMushroom and PowerStar. It also has a capability set.

- Class: SuperMushroom

This class is responsible for the item SuperMushroom that the actor can pick up and consume. This defines unique characteristics and dependencies with other classes that are affected by the super mushroom.

- Class: PowerStar

This class is responsible for the item PowerStar that the actor can pick up and consume. This defines unique characteristics and dependencies with other classes that are affected by the power star.

Class Descriptions for Requirement 5: Trading

- Class: Toad

Description: Toad is a friendly NPC actor that mario can interact with. This class will contain Toad's core information and functionality.

- Class: <<abstract>> ToadAction

Description: The ToadAction class will contain the various interactions Toad can have with Mario, namely speaking with him, or trading with him.

- Class: Trade

Description: The trade class will cause a text-based interaction with Mario, upon which mario can purchase items from Toad.

- Class: Coin

Description: To purchase items from Toad, Mario needs to collect and trade coins for the items. These coins are objects that can be found while playing the game, and when picked up by Mario the object will be removed, and the value of the coin will be added to Mario's wallet.

Class Description for Requirement 6: Monologue

- Class Toad:

Description: The class Toad represents an actor of type Toad. The class Toad contains all the attributes/methods specific to the class Toad and can inherit methods/attributes from the abstract class Actor to ensure all requirements of Toad Speaking is fulfilled.

- Class Speak:

Description: The class Speak deals with all the dialogues that the instance of class Toad needs to say depending on specific conditions.

- Trade:

Description: The class Trade deals with the trading that the user (Mario) does.

- <<abstract>> ToadAction

Description: The class contains all the attributes/methods that all instances of type Toad should have. The class ToadAction inherits from the abstract class Action as there are methods that every actor can do.

Class Descriptions for Requirement 7: Reset Game

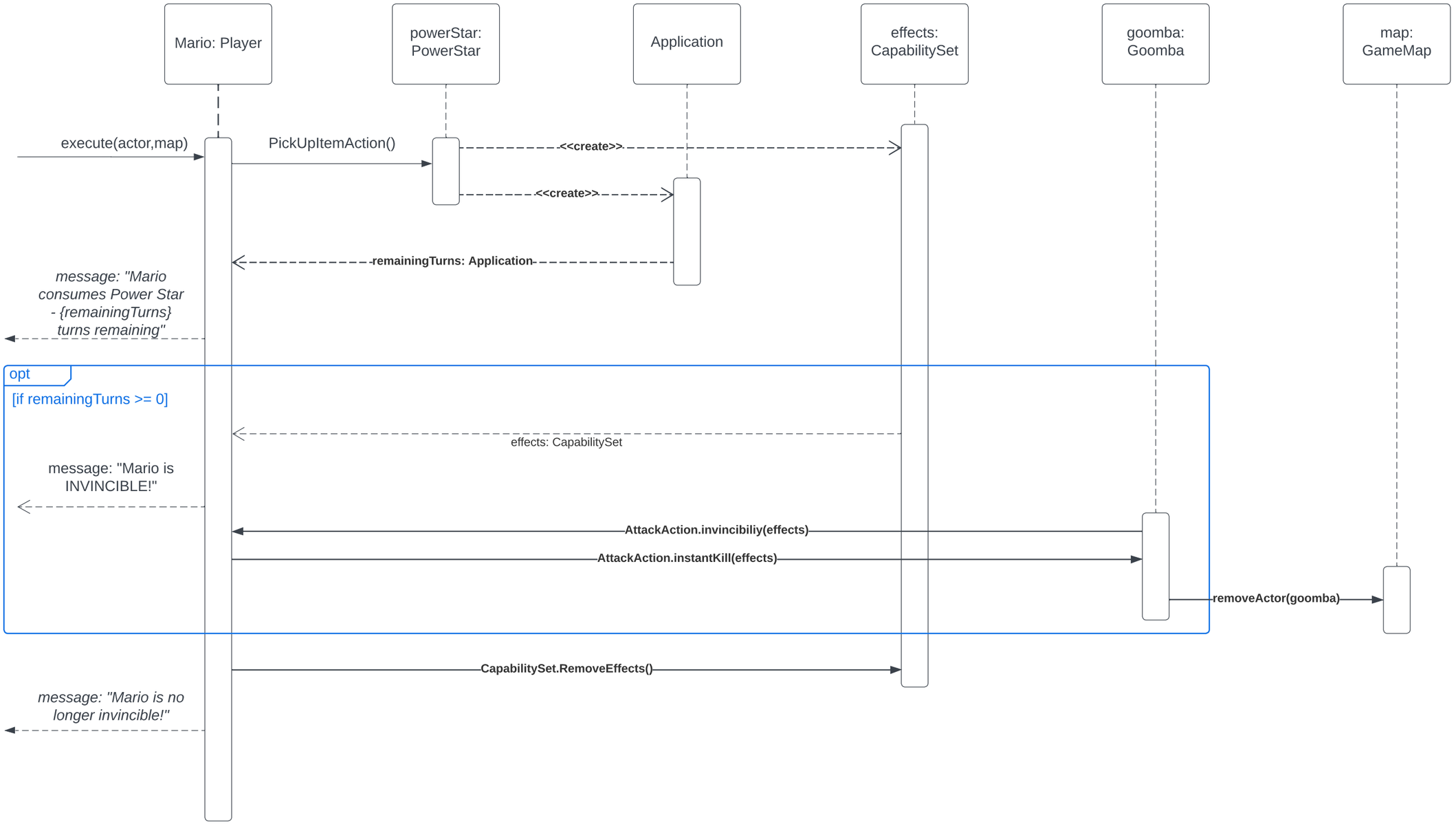
- Class: ResetManager

Description: This class handles the resetting of the game, which the player can do once per game. It will reset enemies, coins, player health and status, and 50% of trees.

- Class: <<interface>> Resettable

Description: This class will check whether or not the game can be reset, which is determined by whether the game has already been reset once in this playthrough.

Mario consuming Power Star and attacking Goomba Scenario - Sequence Diagram



Mario trading a Power Star with the toad- Sequence Diagram