Design Rationale

FIT2099

Zoe Low Pei Ee (31989985)

Chua Shin Herh (31902456)

Loh Zhun Guan (32245327)

Overview

This documentation includes the class diagrams, sequence diagrams and the design rationale of the extended system. It covers the explanation of the roles of the new classes as well as the interaction between the existing and extended classes. This documentation also discusses the design principles that the classes followed and utilized while implementing their functionalities.

Design Principles Utilized

[1] Avoid excessive use of literals

[2] Don’t Repeat Yourself

[3] Single Responsibility Principle

[4] Open-closed Principle

[5] Liskov Substitution Principle

[6] Interface Segregation Principle

[7] Dependency Inversion Principle

REQ 1 - Let it grow!

SpawnCapable

SpawnCapable is an interface which can be implemented by any object that has the ability to spawn something. An interface is used because we can adhere to the **Open-Closed Principle[4]**. For now, only the *Tree* class and its subclasses will implement this interface. If we want to add new functionalities or capabilities to these classes, we can just implement another interface so we don't have to alter anything in this interface because the methods in this interface can only be used by spawnable objects, which fulfills the **Single Responsibility Principle[3]** and **Interface Segregation Principle[6]**.

We are not implementing an abstract method in the *Tree* abstract class as we would like to ensure the extensibility of our system. We assume that trees might not be the only object that has the spawn capability hence we chose to implement an interface instead so we are able to maintain the flexibility of our code and it is also easier to keep track of the capabilities of the classes by implementing interfaces.

Tree

Tree is a class that extends the *Ground* class as it will inherit all the attributes and methods of the ground. It is used to represent the parent class of all the trees. *Tree* is an abstract class as we do not want to instantiate it. The *Tree* class is extended by several classes *Sprout*, *Sapling* and *Mature*. They all share a similar attribute which is the age where the tree will grow after a certain number of turns. This design is implemented based on the **Don't Repeat Yourself principle[2]** as the similar attributes are not repeated within each of the *Tree* subclasses which ensures the maintainability of our code.

Tree class implements the *SpawnCapable* interface as each of its subclasses will implement the drop() method to represent each of their unique spawning abilities. This makes it easier to maintain the code as we do not need to modify the parent class code so new or different functionality can just be added in the child class which adheres to the **Open-Closed principle[4]**.

Sprout

Sprout is a subclass of *Tree* because sprout is a type of tree in the game. As a subclass it will inherit from the parent class the constructors, methods with the same parameters and the return values in order to avoid violating the **Liskov Substitution Principle[5].** Thus, *Sprout* has to override the *drop()* method of the *SpawnCapable* interface as the *Tree* class implements the *SpawnCapable* interface. *Sprout* starts with the age 0 and in the *drop()*

method it will implement its functionality which has a 10% chance of spawning a *Goomba*.

Sapling

Sapling is a subclass of *Tree* because *Sapling* is a type of tree in the game. Similar to the *Sprout* class, it will inherit the parent class constructors, methods with the same parameters and the return values in order to avoid violating the **Liskov Substitution Principle[5].** Thus, *Sapling also* has to override the *drop()* method of the *SpawnCapable* interface. *Sapling* starts with the age 10 and in the *drop()* method it will implement its functionality which has a 10% chance of dropping a coin with a value of $20.

Mature

Mature is a subclass of *Tree* because *Mature* is a type of tree in the game. Similar to *Sprout* and *Sapling*, by adhering to the **Liskov Substitution Principle[5],** *Mature* starts with the age 20 and will override the *drop()* method where its has a 15% chance of spawning a *Koopa* every turn, spawn a new sprout in one of the surrounding fertile squares randomly every 5 turns and a 20% chance of withering and turning into dirt.

Wallet

Wallet is a class that represents the balance of the player. *Wallet* has a balance as a HashMap with *Actor* as key and Integer as a value. Balance was created as a hashmap because it would be easier to just map the players to their respective balance if there is more than one player in the game.

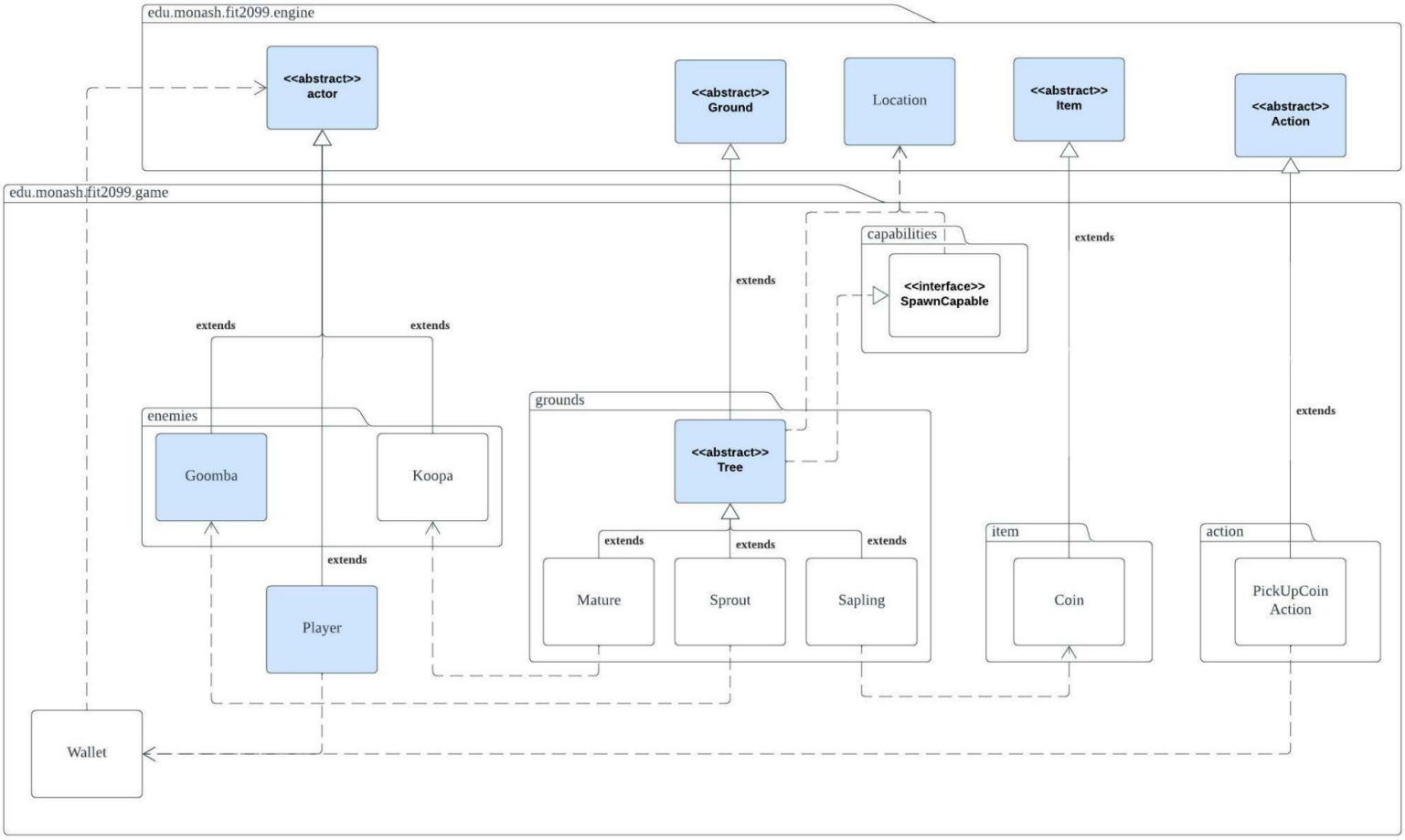
PickUpCoinAction

PickUpCoinAction is a subclass of *Action* as it is a type of action in the game. *PickUpCoinAction* is used to pick up the coins from the map and remove them. It adheres to the **Single-Responsibility Principle[3]** as the class is only used to handle the pick up coin action.

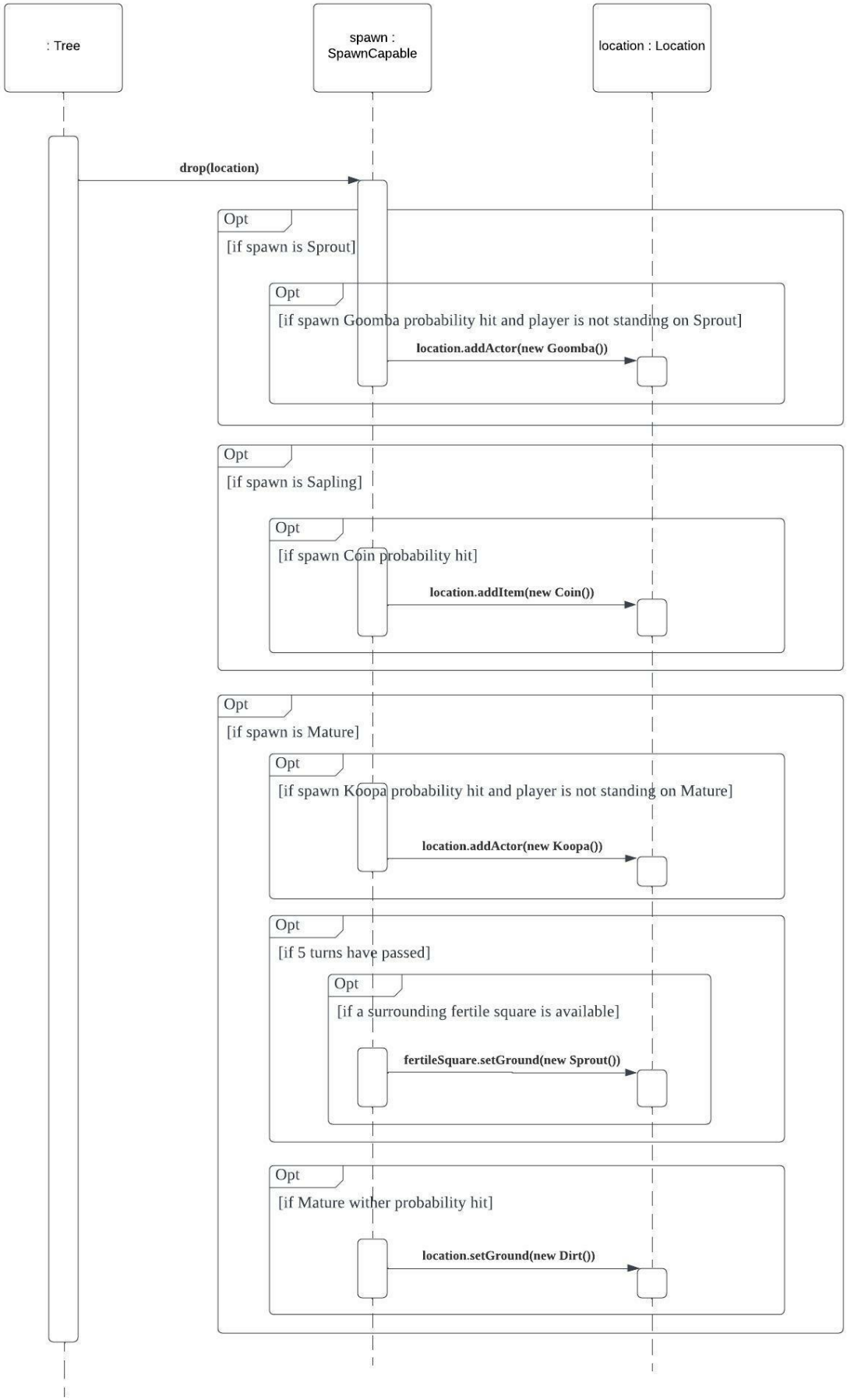
Coin

Coin is a class that represents the coin that is dropped from the *Sapling*. It creates a new action which will then be added to the allowable actions list that the player can perform when the player is on top of a coin.

Class Diagram



Sequence Diagram



REQ 2 - Jump Up, Super Star!

JumpCapable

JumpCapable is an interface which can be implemented by the high grounds. An interface is used because we can adhere to the **Open-Closed Principle[4]**. The high ground classes will implement this interface and if we want to add more features to these classes, we don't need to change anything inside the interface, as the methods in the interface are standard across all high grounds.

This interface also lets us adhere to the **Dependency Inversion Principle[7]**. Instead of directly having a dependency between *JumpAction* and the high ground classes, which can cause problems if we add more high ground classes in the future, we have this interface between *JumpAction* and the high ground classes, so changes in one class wouldn't affect the others.

This interface also adheres to the **Don't Repeat Yourself[2]** principle. We can have a default method in the interface which performs the jump for all the high ground classes rather than having the jump method in all the high ground classes, which reduces duplicate code.

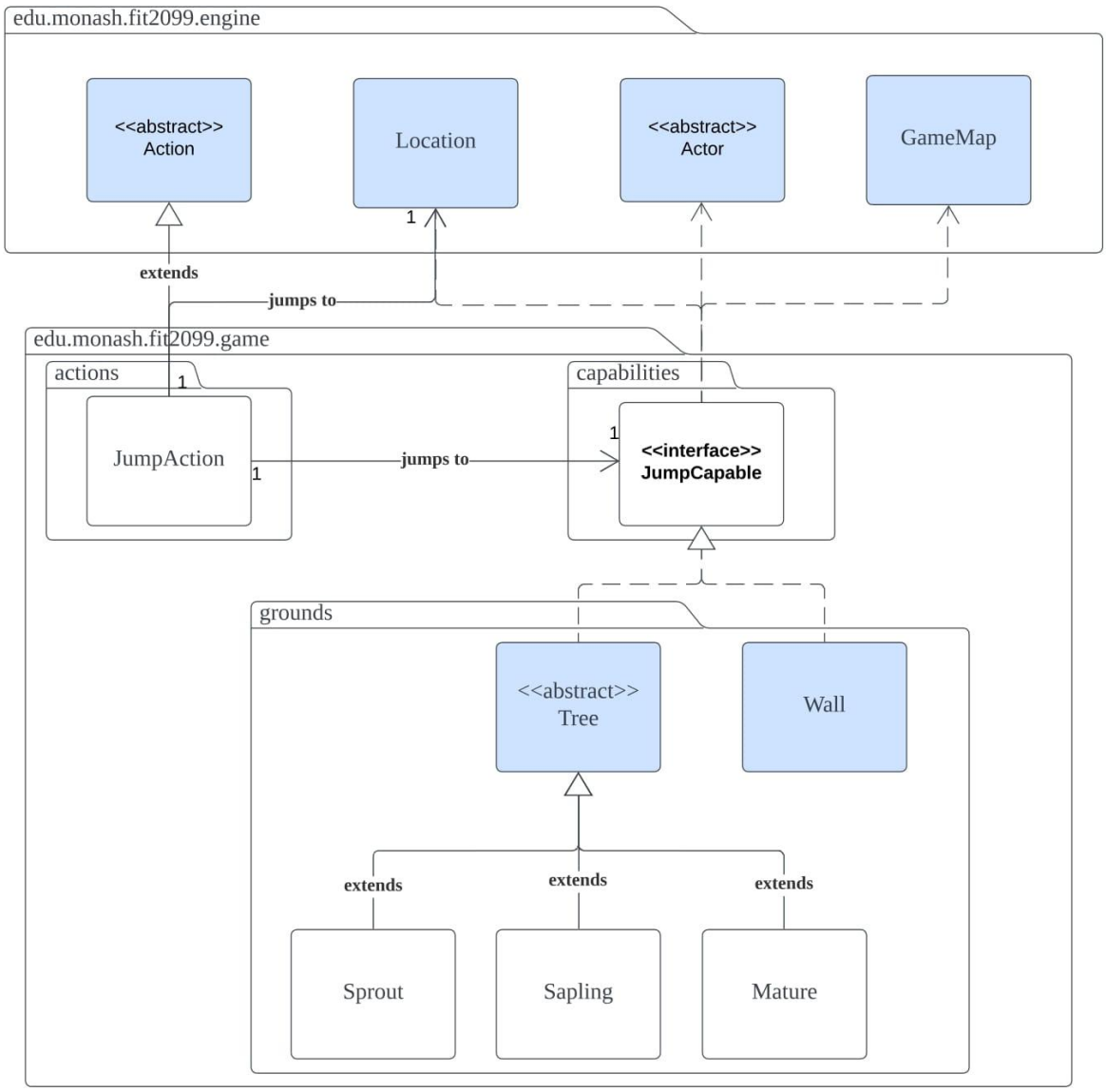
JumpAction

JumpAction is a class used by Actors to jump to a high ground. *JumpAction* extends the *Action* abstract class since jumping is an action, and to adhere to the **Single Responsibility Principle[3]** where this class only handles the jumping actions.

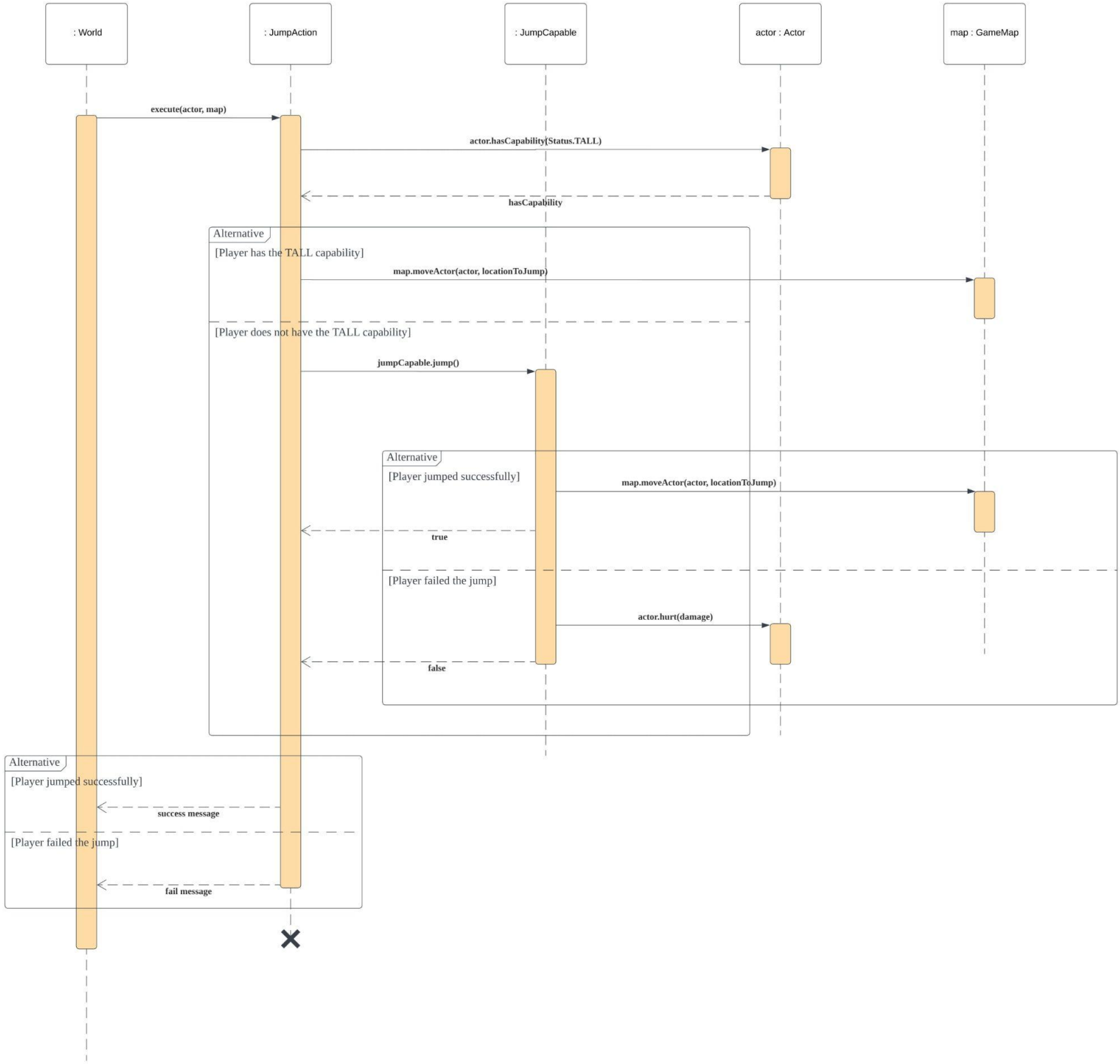
JumpAction has an association to the *Location* class because it needs the location of the high ground to jump to. *JumpAction* has an association to *JumpCapable* because the *JumpAction* can execute the jump method in the *JumpCapable* interface, and it needs a *JumpCapable* instance to know which *JumpCapable* object to execute the jump.

JumpAction also has a dependency on the *Status* enumeration which is not shown in the UML Class Diagram. This dependency is because when the player consumes a Super Mushroom, the player will jump to any high ground with a 100% success rate, and instead of using a magic string to determine whether the player has consumed a Super Mushroom, we can use the *Status*.TALL constant, which will follow the **Avoid Excessive Use of Literals[1]** principle.

Class Diagram



Sequence Diagram



REQ 3 - Enemies

Enemy

Enemy abstract class extends from *Actor* class as it will inherit the attributes and methods of an *Actor*. It represents the parent class of all enemies and we do not want to instantiate an *Enemy* object so we make it an abstract class. The *Enemy* class is extended by several classes which are *Goomba* and *Koopa*. They share similar attributes as all enemies are able to attack the player when the player stands in the enemy's surroundings and will follow the player once it is engaged in a fight. This design is implemented based on the **Don't Repeat Yourself principle[2]** as the similar attributes and methods would not be repeated within each *Enemy* subclass thus making maintenance of the code easier.

Enemy also stores a *behaviours* HashMap with its priority as key and behavior as value so that each enemy can have its own behavior. Hence if any enemy has a specific new behaviour then it is able to utilize the existing HashMap and just put the new behavior into it. This HashMap will be set as private access modifier and final to create a tighter control and avoid unnecessary or unintentional modification on the HashMap, so a separate *addBehaviour* method with a modifier of protected is created to allow only sub-classes that inherits from this Enemy class to be able to add the behaviours into the HashMap. *Enemy will* add a *FollowBehaviour* to the list of *behaviours* (attribute of the *Enemy* class) as it will follow the *Player* once they are engaged in a fight and also an *AttackBehaviour* to attack the player. This maintains the flexibility of the code as it does not need to modify the parent class if new behaviours or implementations are added, thus fulfilling the **Open-Closed principle[4]**. As each enemy has more than one behaviours by default and might have other different behaviours, so the multiplicity of Enemy to *behaviours* HashMap is one to many.

Goomba

Goomba is a subclass of the *Enemy* abstract class because *Goomba* is a type of enemy in the game. *Goomba* starts with 20HP and it has a 10% chance to be removed from the map. *Goomba* will overrides the *getIntrinsicWeapon* method and returns a new instance of the *IntrinsicWeapon* that attacks with a kick and 10 damage, the hit rate is the same as player so we can just overrides this method instead of creating duplicate methods. As a subclass, it will inherit and use the base class constructors, methods with the same signature and return value to avoid violating **Liskov Substitution Principle[5].**

As the behaviours are stored in the form of HashMap in the *Enemy* abstract class, the existing structure will not be changed whenever we add a behaviour which adheres to the **Open-Closed Principle[4]**. Next, the *Goomba* would implement a *SuicideAction* because the Goomba has the ability to suicide during the game thus it overrides the *playTurn* method from the *Actor* class to have a 10% chance of calling the *SuicideAction*. This

approach fulfills both the **Single Responsibility Principle[3]** and the **Don't Repeat Yourself Principle[2].**

Koopa

Koopa is a subclass of the *Enemy* abstract class because *Koopa* is a type of enemy in the game. *Koopa* starts with 100HP and will go into dormant state (D) when it is defeated by the player. *Koopa* will overrides the *getIntrinsicWeapon* method and returns a new instance of the *IntrinsicWeapon* that attacks with a punch and 30 damage, the hit rate is the same as player and *Goomba* so we don't have to extend *IntrinsicWeapon* class or create other methods to modify the hit rate value.

Koopa would override the *playTurn* method as it will go to a dormant state and stay on the ground when it is not conscious and will then be removed from the map and drop a Super Mushroom once its shell is destroyed by the player by a Wrench.

Koopa will execute the *playTurn* method of the parent class (Enemy abstract class) through the "super" keyword. The default implementation of *playTurn* is to execute the *WanderBehaviour*, *FollowBehaviour* and *AttackBehaviour* which are the default behaviour of the enemies based on the *Status* of the *Player* and *Enemy*. As the behaviours are stored in the form of HashMap in the *Enemy* abstract class, the existing structure will not be changed. Hence we won't need to repeat the same chunk of code which fulfills the **Don't Repeat Yourself principle[2]** as repeating code is avoided.

Wrench

Wrench will be extending from *WeaponItem* abstract class as it is a type of *Weapon* and also an *Item*. As a subclass, it will inherit and use the base class constructors and it can call the methods of the *WeaponItem* class when needed without implementing duplication methods which fulfills the **Don't Repeat Yourself principle[2].**

AttackKoopaAction

AttackKoopaAction will be an extension from *Action* class, in which the constructors and methods will have the same signatures. It stores the attributes of an *Actor* target (which is the one to be attacked -- *Koopa*) and a String of direction (to identify the direction of incoming attack). The random number attribute is also stored as a class level attribute, it is to generate the probability of successful attacks chances.

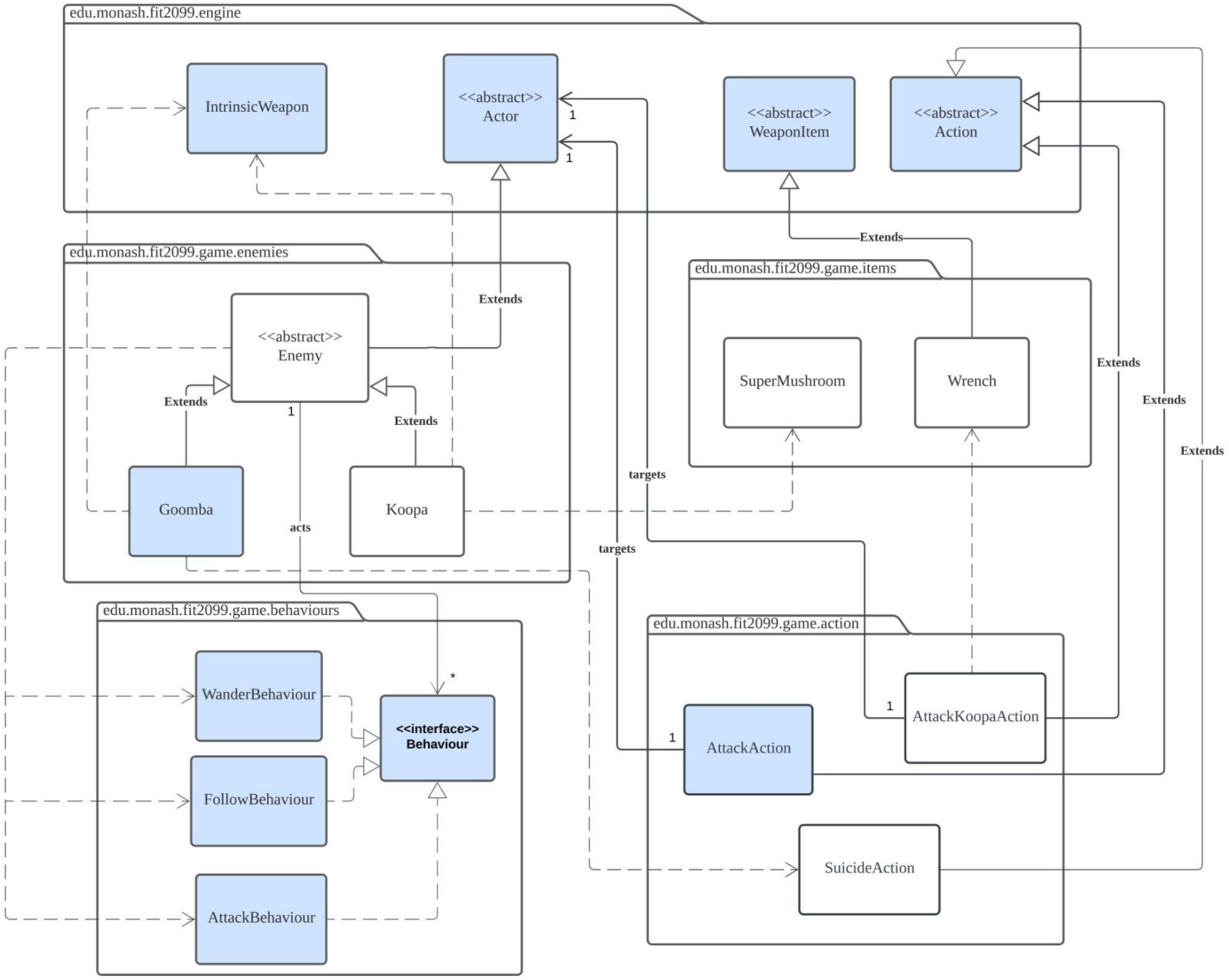
AttackKoopaAction fulfills the **Single-Responsibility principle[3]** as it will only be called while attacking *Koopa*. It will do nothing else such as the action of "checking if the target is *Koopa*" or "the *Koopa* attack the *Player*" are not within the class as its main responsibility is just to attack and defeat the *Koopa* (destroy its shell) but it will ensure that the weapon used by the *Player* is a *Wrench* as it is the only weapon that can destroy the shell.

AttackKoopaAction inherits *AttackAction* class and only overrides or add methods that will have different functioning logic so that we can avoid code duplication as both classes shared most of the logic which is to attack the target, hence adhering to the principle **Don't Repeat Yourself [2].** The main difference between two classes is the *execute* method, where the Player has to destroy *Koopa*'s shell with a *Wrench* which will then drop a *Super Mushroom* when the *Koopa* is defeated. As it is a special condition only for *Koopa*, hence the design decision made to create a new subclass is also to support different execution of attack separately rather than creating multiple check conditions and dependencies (i.e. Super Mushroom) in the base class.

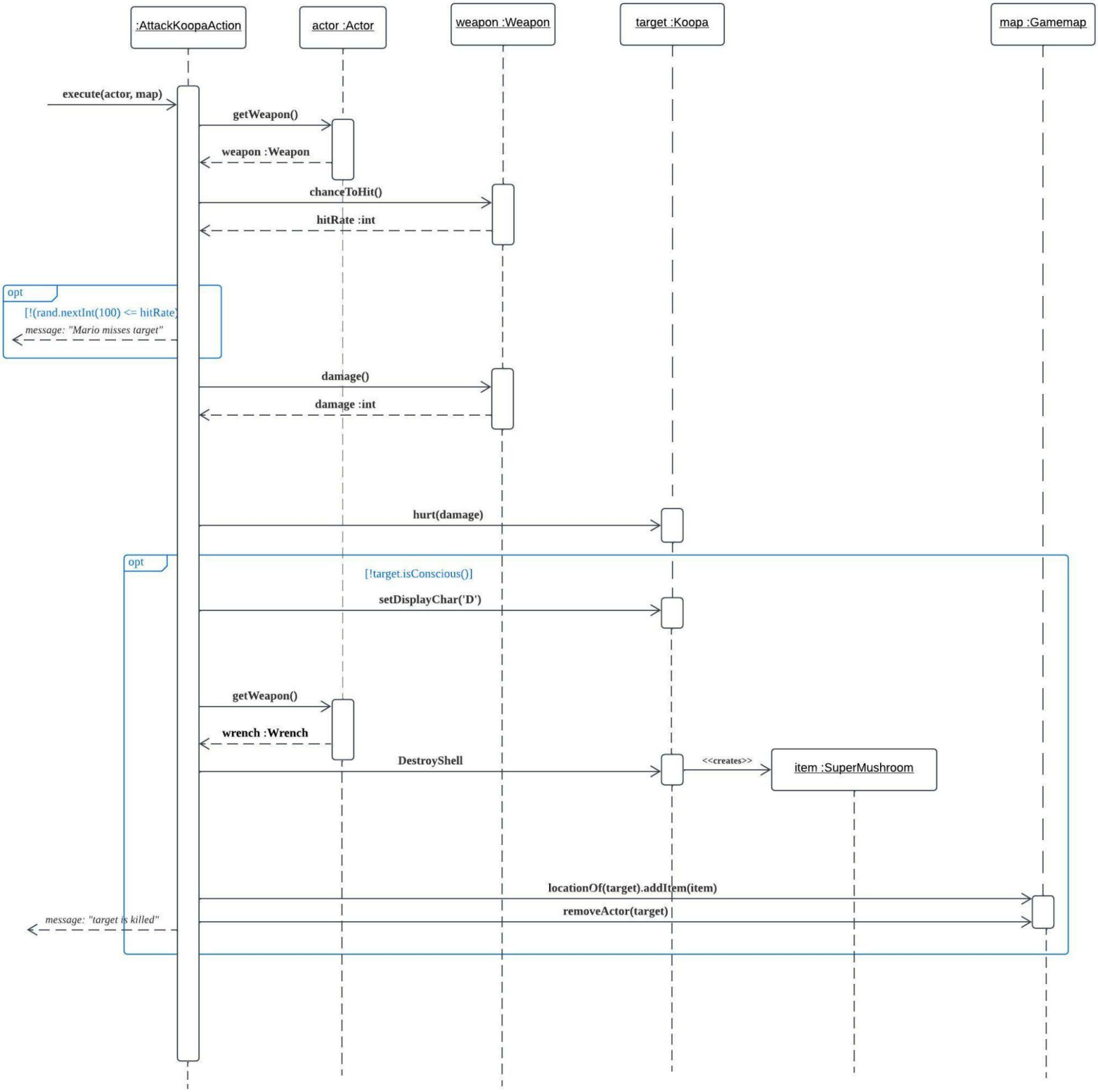
SuicideAction

SuicideAction will be extending from the *Action* abstract class. *SuicideAction* is used to remove *Goomba* from the map and its main implementation is just calling the *removeActor*() function and print out a String message indicating the *Goomba* has suicide which follows the **Single-Responsibility principle[3]**.

Class Diagram



Sequence Diagram



REQ 4 - Magical Items

MagicalItem

MagicalItem will be extending from *Item* abstract class as it is a type of *Item*. It represents the parent class of all magical items (PowerStar and SuperMushroom) and we do not want it to be instantiated by other classes. We make it as an abstract class to ensure we follow the **Dependency Inversion Principle[7]** where a concrete class should not depend on another concrete class and should depend on abstractions instead. A *MagicalItem* inherits the attributes and methods of the *Item* class. It has an abstract method *consumedBy()* which the subclasses have to provide the implementation of this method with this same signature. This design decision is made as each magical item will have different effects on the *Player* hence require different implementation of that method. Our design assumes that by default, *MagicalItem* is already on the same ground and the player is able to pick it up or drop it. As *MagicalItem* inherits *Item* abstract class, the extended class would only need to use the existing attributes and methods through the "super" keyword which fulfills the **Don't Repeat Yourself (DRY) principle[2]**.

SuperMushroom

SuperMushroom will be extending from *MagicalItem* abstract class. It will override the *consumedBy*() method of its parent class which executes the special features after the Actor consumes it. It will change the status of the actor to Status.TALL, which is already provided in the *Status enumeration*, hence we can **avoid excessive use of literals[1]**.

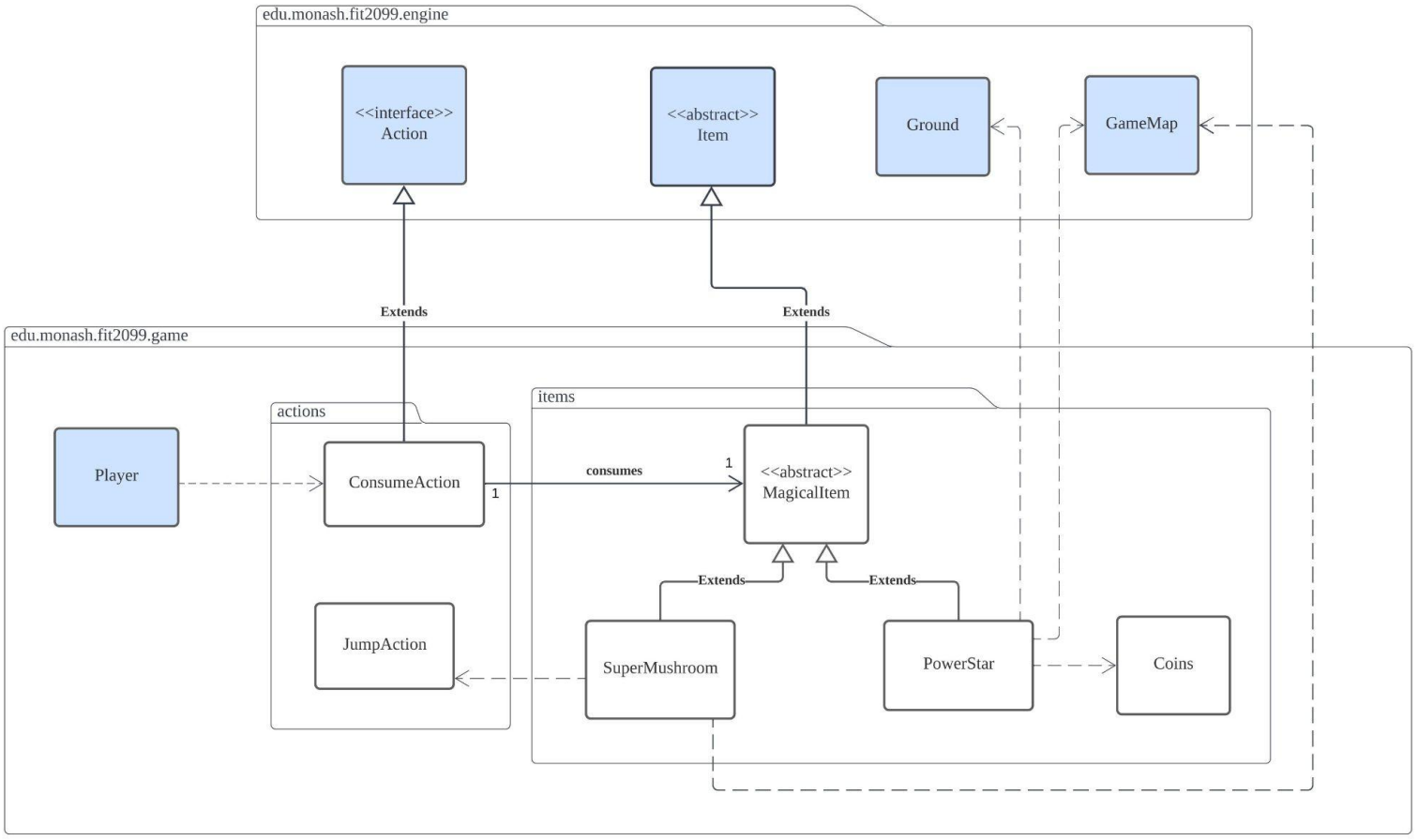
PowerStar

PowerStar will be extending from the *MagicalItem* abstract class. It will override the *consumedBy*() method of its parent class which executes the special features after the Actor consumes it. It will change the status of the actor to *Status*.INVINCIBLE, hence we can **avoid excessive use of literals[1]**. The difference between *PowerStar* and *SuperMushroom* is the dependency relationships of *Ground* class, *Dirt* class and *Coin* class as we will have to convert the higher grounds to dirt and drop a Coin after destroying the ground thus instantiating an instance of these classes. Also, *PowerStar* has a *tick*() method to keep track of the remaining turns of the effect as it will self-destruct after 10 turns.

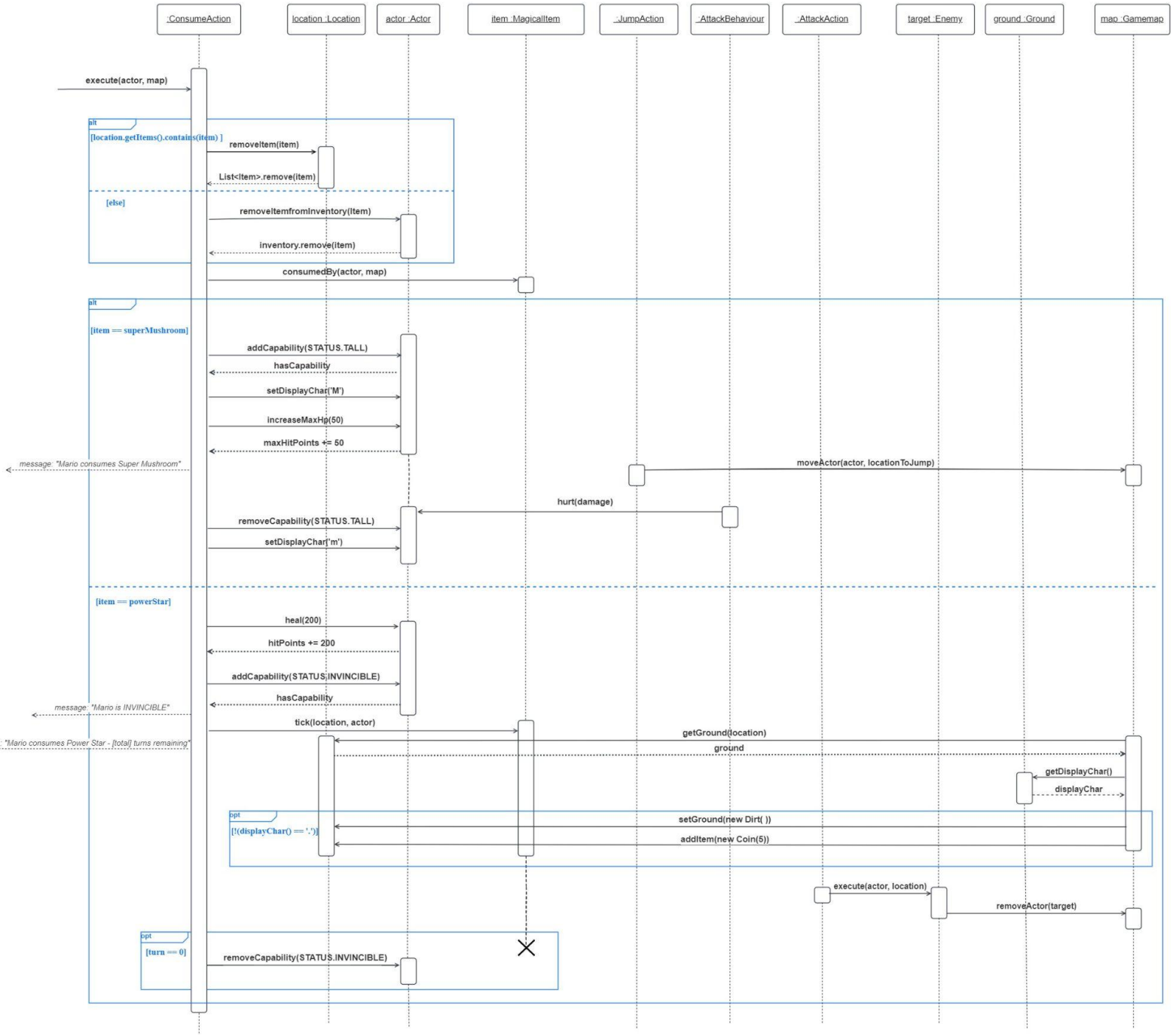
ConsumeAction

ConsumeAction will be a subclass of *Action* abstract class, it inherits to **minimize code duplication[2]**. *ConsumeAction* fulfills the **Single Responsibility principle[3]** because it will only remove the *MagicalItem* from the inventory or on the ground and consumes the *MagicalItem*. The *execute*() method is overridden, by checking whether the magical item is on the ground or in the Actor's inventory to remove it, followed by calling the consumedBy() method of the related subclass to consume the *MagicalItem*. The menu description is overwritten as well to output the message "[*Actor*] consumes [*MagicalItem*]".

Class Diagram



Sequence Diagram



REQ 5 - Trading

Buyable

Buyable is an interface which can be implemented by the buyable items. An interface is used because we can adhere to the **Open-Closed Principle[4]**. The buyable items will implement this interface and if we want to add more features to these classes, we don't need to change anything inside the interface, as the methods in the interface are standard across all buyable items.

This interface also lets us adhere to the **Dependency Inversion Principle[7]**. Instead of directly having a dependency between *BuyAction* and the buyable items, which can cause problems if we add more buyable items in the future, we have this interface between *BuyAction* and the buyable item classes, so changes in one class wouldn't affect the others.

This interface also adheres to the **Don't Repeat Yourself[2]** principle. We can have a default method in the interface which performs the buying for all the buyable items rather than having the buy method in all the buyable item classes, which reduces duplicate code.

Buyable has a dependency on the *Wallet* class as buying items would require access to the wallet balance of the player, and to deduct coins from the wallet's balance. *Buyable* has a dependency on the *Item* class because when the player buys something, the item will be added to the player's inventory, and adding an item to the player's inventory uses the item, but doesn't require an association to the item.

BuyAction

BuyAction is a class used by the player to buy items from *Toad*. *BuyAction* extends the *Action* abstract class because buying is an action, and to adhere to the **Single Responsibility Principle[3]** where this class only handles the buying actions.

BuyAction has an association to *Buyable* because the *BuyAction* can execute the buy method in the *Buyable* interface, and it needs a *Buyable* instance to know which *Buyable* object to execute the buy process.

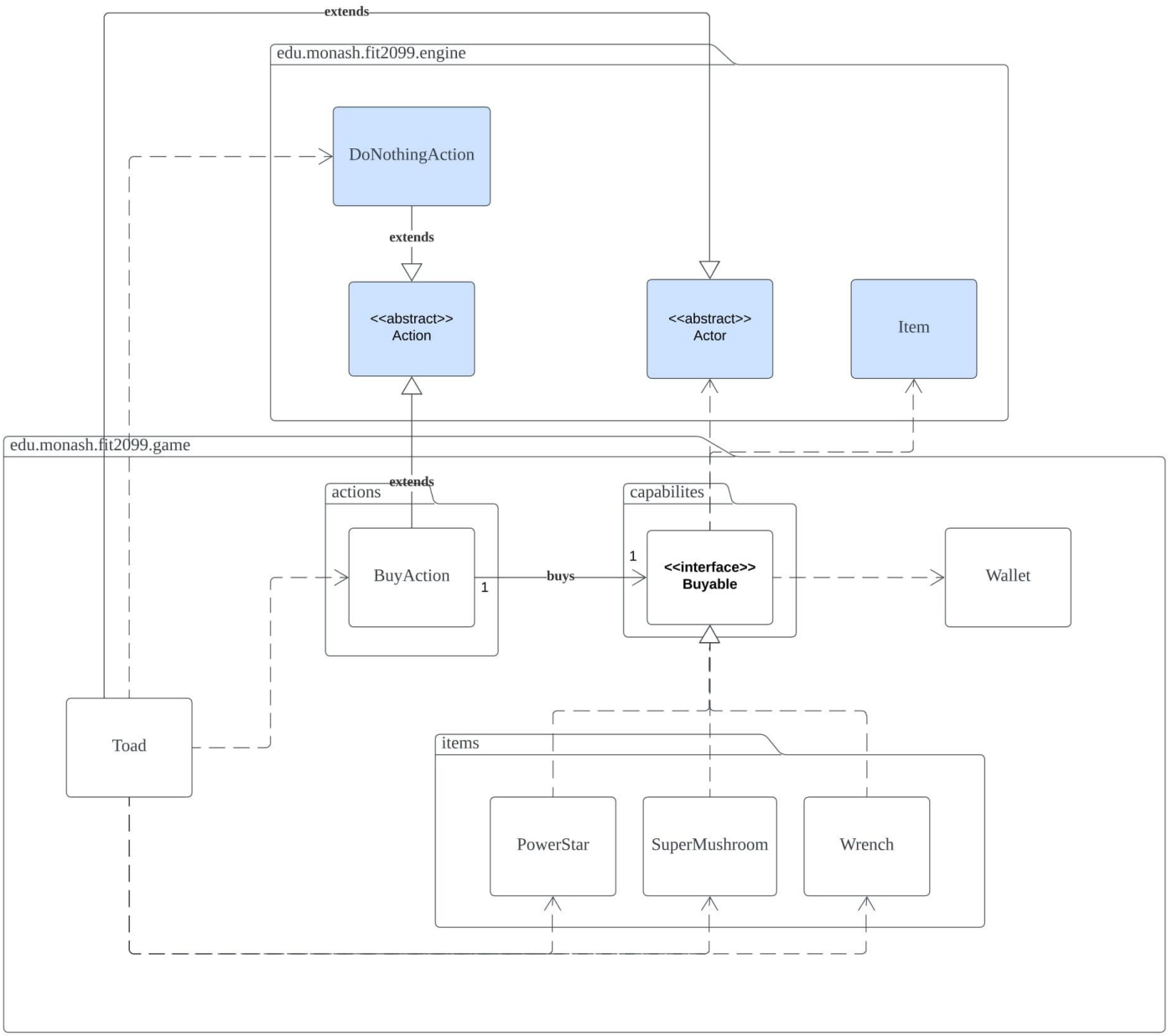
Toad

Toad is a class which will provide the player with items to buy. *Toad* extends the *Actor* class as *Toad* is an actor. *Toad* has dependencies to the classes *PowerStar*, *SuperMushroom* and *Wrench* because *Toad* uses new instances of these classes to sell

these items to the player. *Toad* has a dependency on *BuyAction* because buying items from *Toad* are part of Toad's allowable actions.

Toad has a dependency on *DoNothingAction* because *Toad* will not do anything on every turn.

Class Diagram

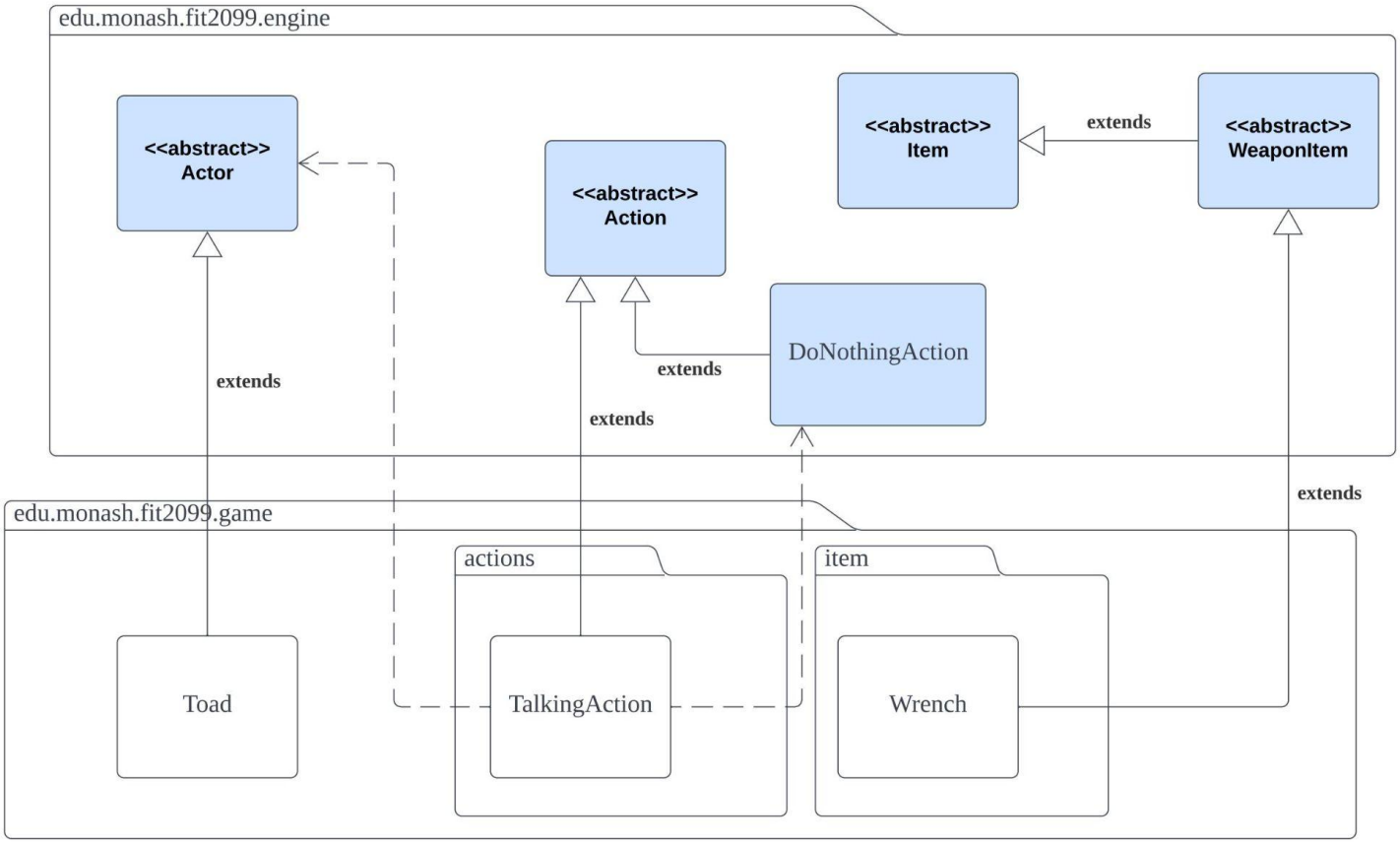


REQ 6 - Monologue

TalkingAction

TalkingAction is a subclass of *Action* used to represent the conversation between *Toad* and the player. *TalkingAction* adheres to the **Single Responsibility Principle[3]** as this class only handles the talking actions. *TalkingAction* also has a dependency on *Actor* as it needs to check if the actor, which is the player, has eaten a Power Star and has a wrench in the inventory to react with appropriate dialogue.

Class Diagram

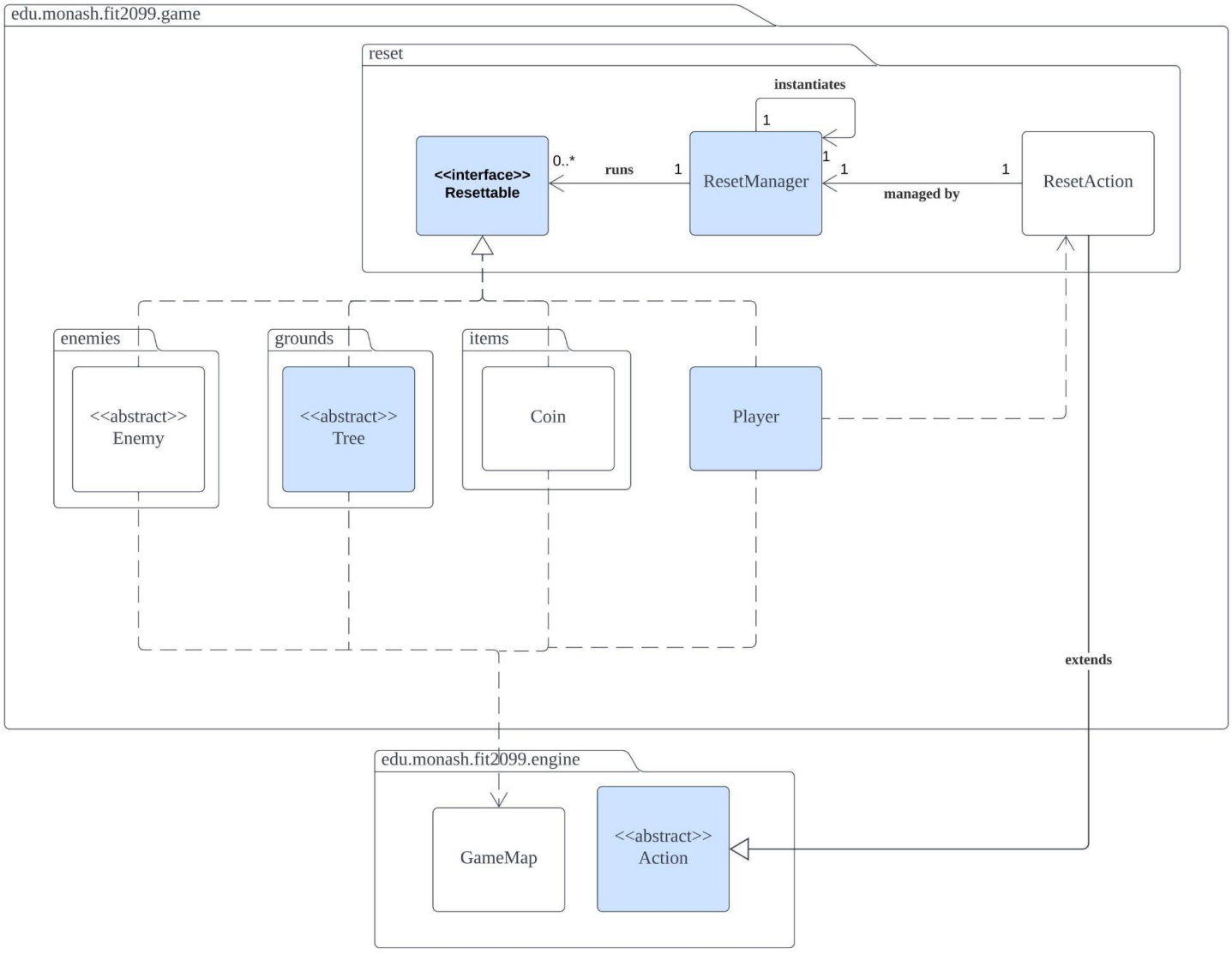


REQ 7 - Reset Game

ResetAction

ResetAction will be extended from the *Action* abstract class. In this class, we get the singleton *ResetManager* instance as the class attribute by calling the public static factory method *getInstance*() of the class *ResetManager* hence resulting in a one-to-one association relationship between *ResetAction* and *ResetManager*. The main responsibility of this class is to execute the *run*() method of the *ResetManager* to clean the map and reset the game, which adheres to the **Single Responsibility Principle[3]**. Furthermore, by doing this way it will avoid using "instanceof" all over the place, to ensure that we are not violating the **Liskov Substitution Principle[5]**. As the Player class will be the only class that can create *ResetAction* in the *playTurn*() method, hence it will have a dependency relationship with the *Player* class.

Class Diagram



Work Breakdown Agreement Assignment 1

Zoe Low Pei Ee	31989985	zlow0011@student.monash.edu
Chua Shin Herh	31902456	schu0064@student.monash.edu
Loh Zhun Guan	32245327	zloh0009@student.monash.edu

Task: Req 1: Create Trees UML Class diagram and implement CreateCoin UML Interaction diagram

Person In Charge: Loh Zhun Guan

Deadlines: 7th April 2022

Task: Req 2: Create Jump Action UML Class diagram and implement JumpAction UML Interaction diagram

Person In Charge: Chua Shin Herh

Deadlines: 7th April 2022

Task: Req 3: Create Enemies UML Class diagram and implement AttackKoopaAction UML Interaction diagram

Person In Charge: Zoe Low Pei Ee

Deadlines: 7th April 2022

Task: Req 4: Create Magical Item UML Class diagram and ConsumeAction UML Interaction diagram

Person In Charge: Zoe Low Pei Ee

Deadlines: 7th April 2022

Task: Req 5: Create Trading UML Class diagram

Person In Charge: Chua Shin Herh

Deadlines: 7th April 2022

Task: Req 6: Create Monologue UML Class diagram Interaction diagram

Person In Charge: Loh Zhun Guan

Deadlines: 7th April 2022

Task: Req 7: Reset Game UML class diagram Interaction diagram

Person In Charge: Collaborative

Deadlines: 7th April 2022

Task: Design rationale of each REQ

Person In Charge: Collaborative

Deadlines: 7th April 2022

Task: Review and testing

Person In Charge: Collaborative

Deadlines: 8th April 2022

1. Each task, when completed, is to be pushed onto the repository.

2. Work will be pushed to the git repository frequently to ensure each member gets the latest update of the work, push partially completed tasks with commit message “TO BE COMPLETED”.

3. Diagrams are required to be pushed after any changes to ensure consistency. Diagrams will be made through lucidchart.com.

4. All work to be done before the deadline, no work will be done later than the 10th of April.

If the member agrees to terms, please sign your name, date, with “I accept this WBA”.

- Zoe Low Pei Ee, 4th April 2022, I accept this WBA.
- Chua Shin Herh, 4th April 2022, I accept this WBA.
- Loh Zhun Guan, 5th April 2022, I accept this WBA.