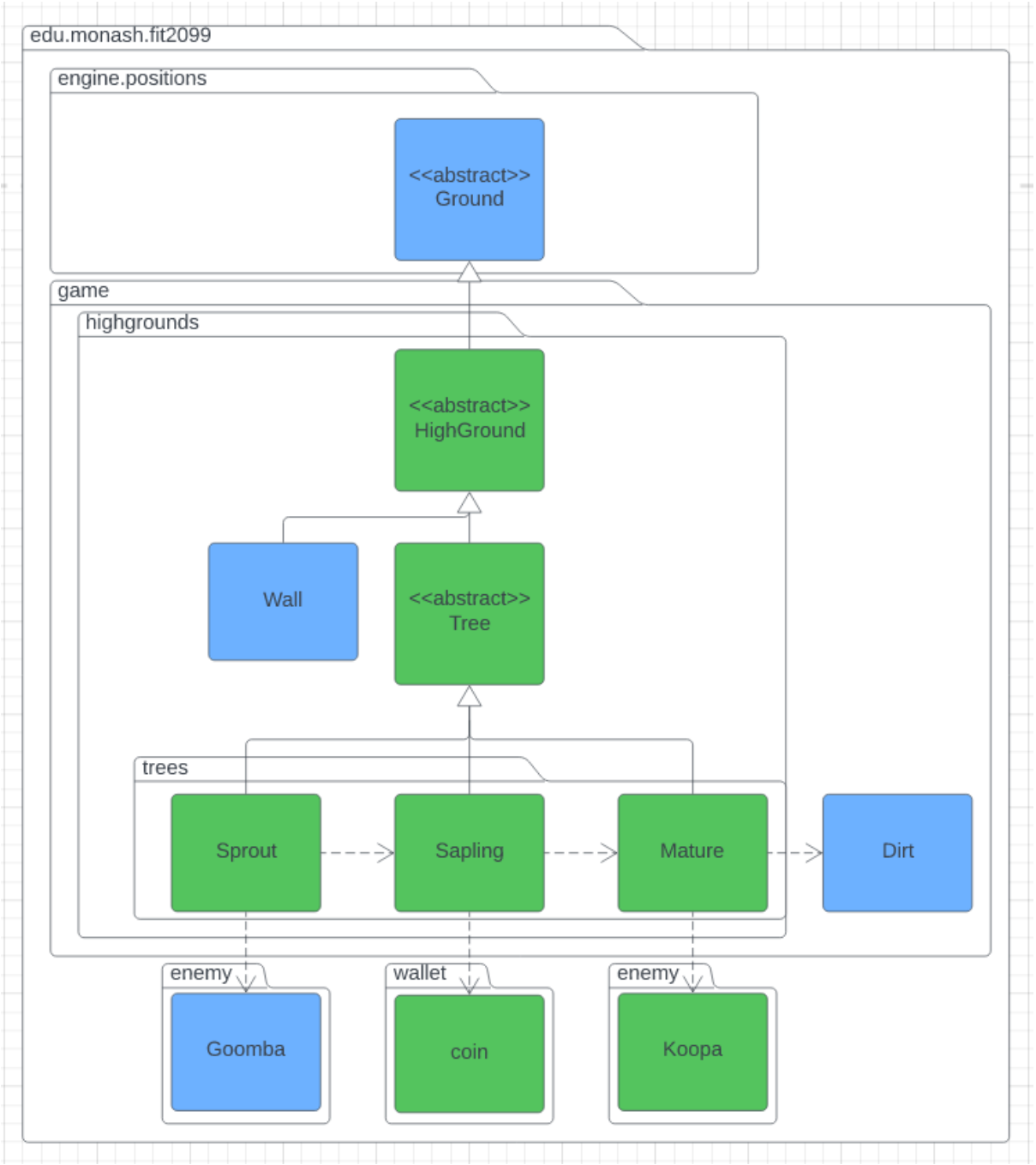
FIT2099 ASSIGNMENT 2

By Joel Atta, Jack Patton & Yepu Hou

Class diagram details:
Blue classes are classes that, as of starting that specific requirement, already existed in the project.
Conversely, green classes are classes that did not exist before starting that requirement and were added during it. Also, pre-existing classes that had their class header modified are green, such as when a non-abstract class that already existed is made abstract.
This means when a class is first created it will be in green, but if it is referenced in another UML diagram in a later requirement it will be in blue. Therefore, if you see a new class and don't understand its purpose, you can scroll up until you find the requirement in which that class is green, where it will be explained.

Req 1: Let it Grow!

Class Diagram:



Sequence Diagram:

Simple enough to not need a sequence diagram.

Design Rationale:

Part1:

We could maintain a single Tree class and then choose which type of tree an instance of Tree is based on some internal counter tracking the turns since the tree was created. However, it would be better if there was an abstract Tree class and then 3 different classes representing each life stage of a tree that extended the main abstract tree class. This follows the already present methodology wherein Ground is extended by Dirt and Floor classes.

As Ground is an abstract class, and interfaces cannot extend an abstract class, Tree must also be an abstract class and not an interface.

For Req1, this means you can open the Sprout class and see that it has a 10% chance to spawn a goomba, while in the Sapling class the goomba spawning method is not present. In other words, code not relevant to that specific type of tree is not present in that specific type's class. This means we don't have to check the tree's current life stage to determine if it should, for example, spawn or goomba or spawn a coin.

These 3 classes follow the Liskov Substitution Principle, and there's an example of why based on requirement 2. Let's say Mario has a jump function that can be passed whatever object Mario is attempting to jump onto; it is legal for you to pass any of the classes derived from the abstract class Tree, which all have a different value for the success chance of jumping/damage taken on a failed jump.

Also, I found the 'game' package to be a bit cluttered and had issues finding what I was looking for quickly, so I moved all the tree types to their package within 'game' to aid in the clarity

Tree	Abstract class containing shared attributes and methods of all trees.
Sprout	Extends Tree, youngest tree stage.
Sapling	Extends Tree, middle tree stage.
Mature	Extends Tree, oldest tree stage.

Part 2 Changes:

Sprouts create Goombas, and will eventually grow into a Sapling.

Saplings create Coins, and will eventually grow into a Mature.

Matures create Koopas, and will eventually die and create Dirt.

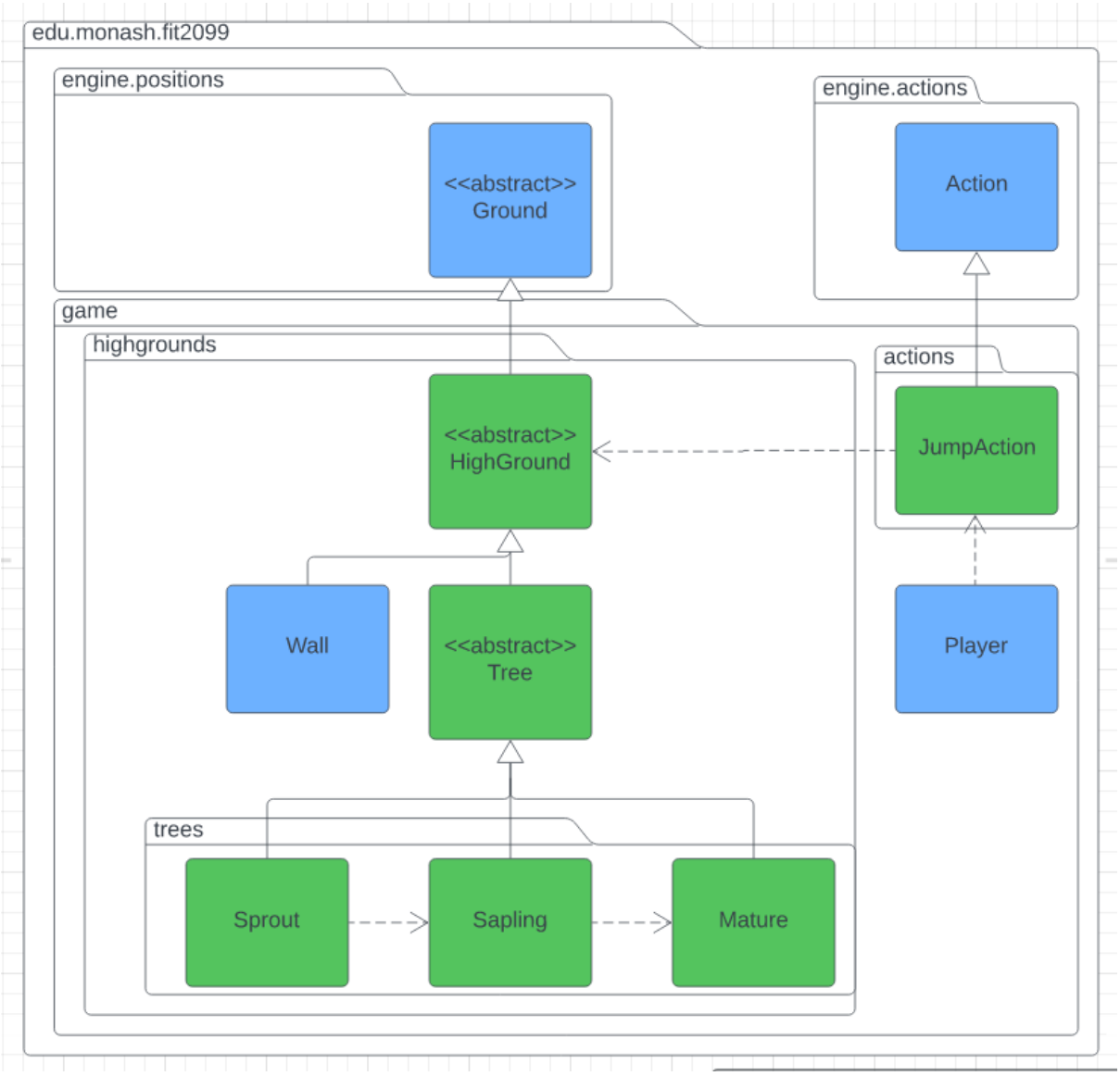
These are all now represented on the UML diagram.

Also, the dependencies between the Tree subtypes are now displayed, showing their growth.

Other than the addition of the abstract HighGround class (which is a consequence of req2 so it will be explained there), there aren't really any changes to the design, I simply included details that were described in the Part 1 rationale and should have been included in the original UML diagram. Most of the actual changes were made to accommodate req2, so they'll be explained there instead of here in req1.

Req 2: Jump Up, Super Star!

Class Diagram:



Sequence Diagram:

Simple enough to not need a sequence diagram. However, I will briefly describe the objects involved, to improve clarity.

Player stores an ArrayList of Actions that it can perform at any one time. Relevant jump actions will be added based on the Player instances' position in the world, e.g. if Mario is next to a tree and a wall there will be two JumpActions in his available action list for each of the high grounds he can attempt a jump to.

When the JumpAction is generated, the HighGround that is being jumped to has its chance of success and damage taken on failure values queried and stored, allowing the actual jump execution to be performed. If the Player beats the failure rate, they will successfully jump to on top of the HighGround. If they fail, they will not be moved and take damage equal to the HighGrounds damage on failure value.

It's important to note that if the Player is affected by a SuperMushroom they will always succeed on their jump attempts, but if they are affected by a PowerStar they will not be given a JumpAction when next to HighGround, as they instead treat it as normal ground while under the effects of the star and can simply walk onto HighGround without jumping. More details on this in the later requirements.

Design Rationale:

Part 1:

Going up requires a jump, going down doesn’t. Enemies cannot jump at all.

Therefore, we don’t need to implement a jump feature for the enemies. The specifications state that enemies are intended to not be able to jump, so we don’t need to worry about making it easy to add jump functionality to the enemies later down the line. (Not that it would be difficult to do, but it's good to keep in mind as it can make coding simpler if you don’t need to worry about checking what actor is jumping).

In the same way that AttackAction is a class that extends Action, JumpAction is also a class that extends action. AttackBehaviour is only used by enemies to determine how they automatically attack, and as enemies can’t jump we don’t need to implement a JumpBehaviour class. Any “behaviour” for the jump can be kept in the JumpAction class itself.

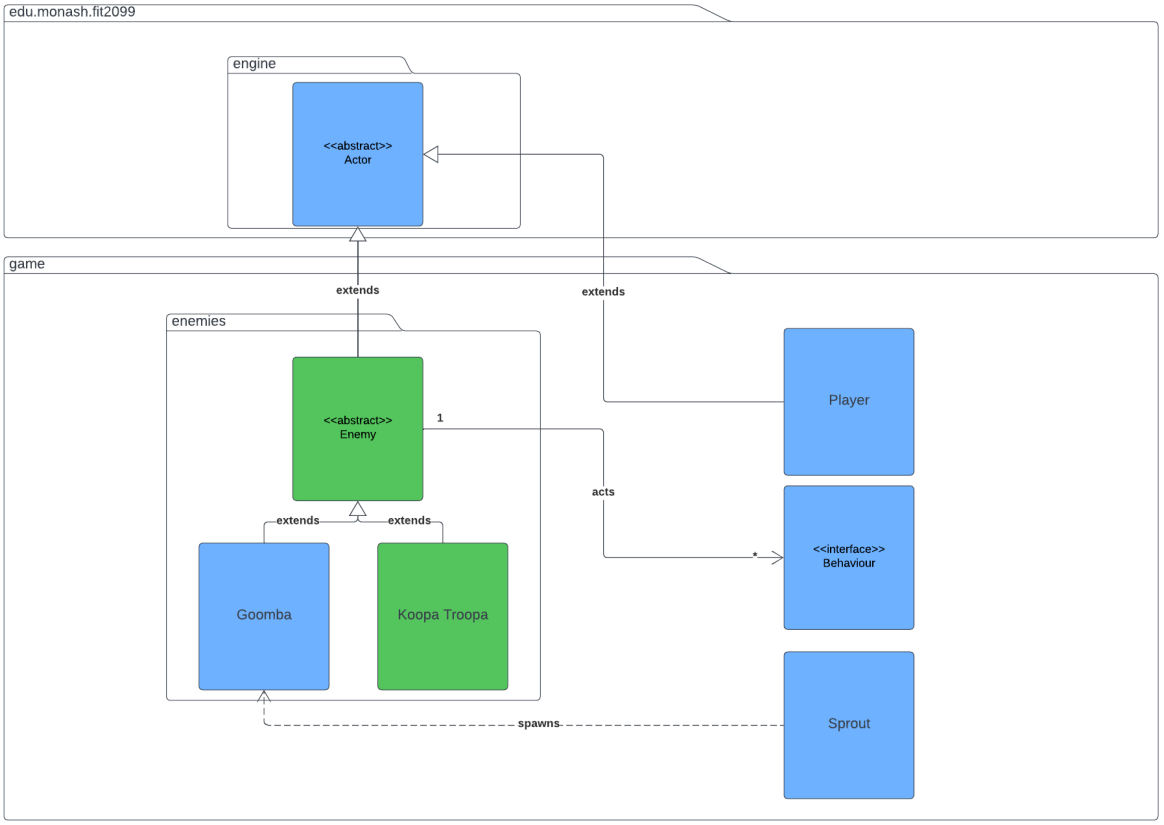
Action	Abstract class for actions an Actor can perform
Player	Adds an instance of JumpAction to its ArrayList of Actions for every jump the player can currently make.
JumpAction	Handles player jumping onto high ground action.

Part 2:

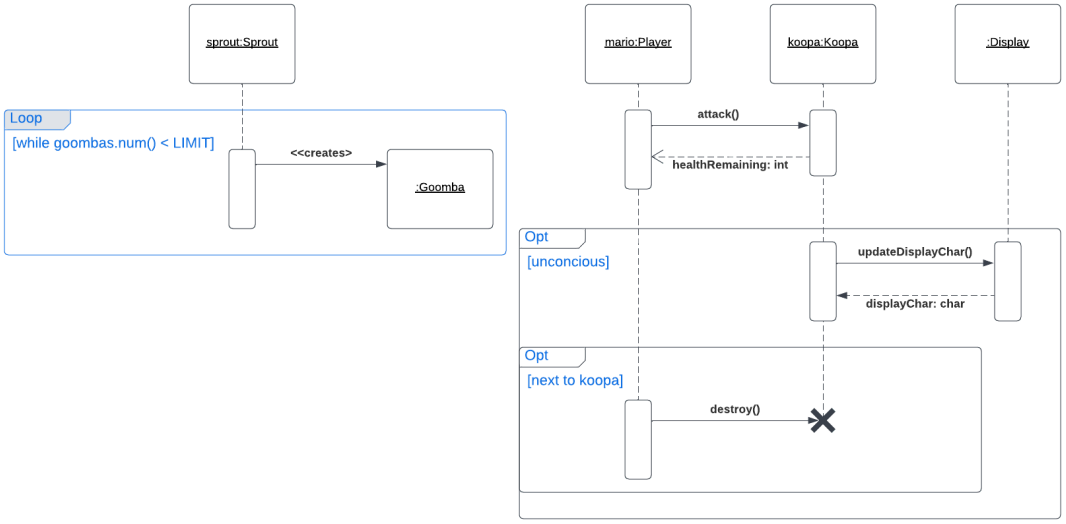
Note that even though Player extends the Actor class I didn't show this dependency in the class diagram as Actor doesn't have any new direct dependencies, and this requirement is focused on the JumpAction class. Further, Player is the only Actor that can use JumpAction and it could get confusing if Actor is included.
I noticed that there are multiple types of ground you could jump on, and they weren't just tree types. Due to this, I created a HighGround abstract class which Tree (and by extension the different Tree life stages) and Wall extends from. HighGround represents a Ground which Actors cannot by default walk on but allows Players to use their special JumpAction to attempt to jump onto it.
This makes it much easier to create new normal Ground (such as Dirt and Floor) or the further abstracted HighGround (such as Wall and Sapling).
For clarity, I also added a highgrounds package which includes the tree package, HighGround, and Wall. Any future high grounds will be stored here as well.
As JumpAction is an action, it follows the standards set by other actions e.g. AttackAction by extending the Action abstract class, allowing the JumpActions to be easily added to the player interface console.

Req 3: Enemies

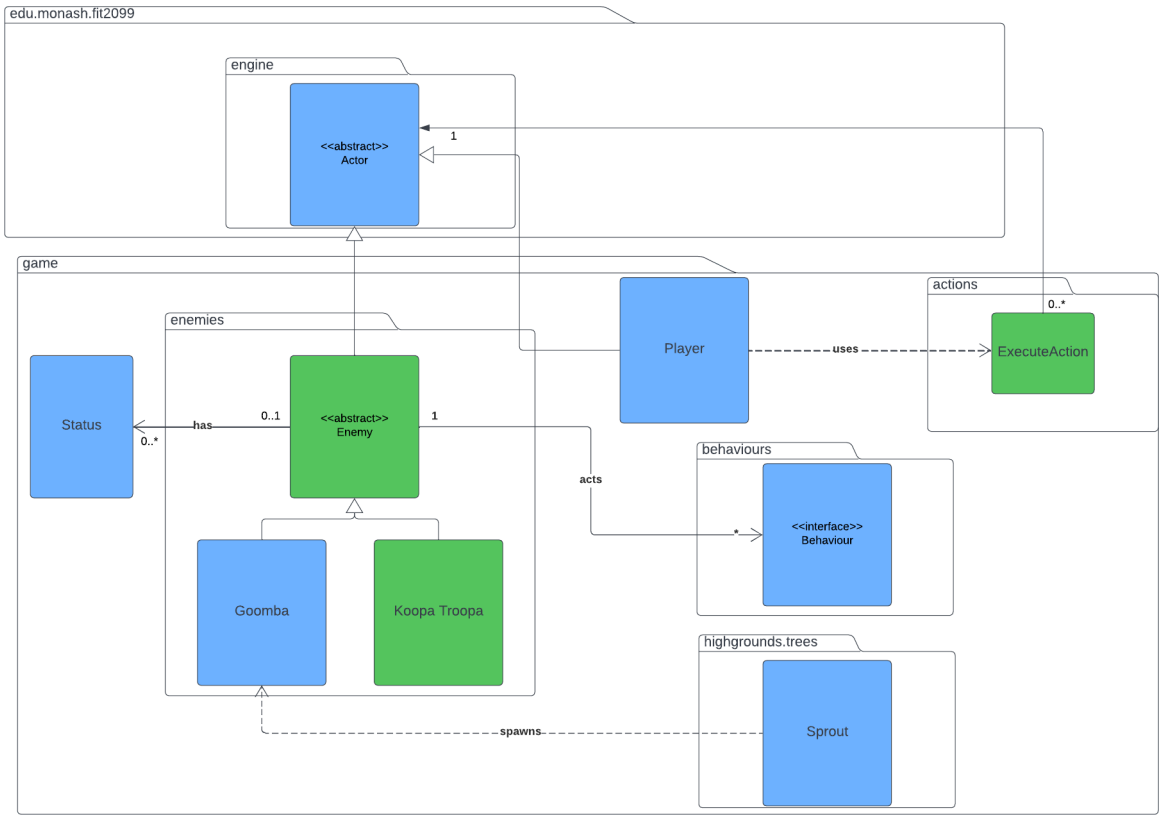
Class Diagram:



Sequence Diagram:



Updated class diagram (Assignment 2):



Design Rationale:

Enemy	Abstract class storing common attributes/methods to be extended by the many enemy types that may be in the game.
Koopa	A new enemy type within the Enemy abstract class. It will retreat to its shell when attacked by Mario and will only die if Mario hits it with a Wrench.
ExecuteAction	Used by the player to execute the Koopa if it is DORMANT and remove it from the map.

The starting point for my development of the class diagram for the Enemies requirement started by looking at the enemy that was already in the game, i.e. the Goomba. From this, I could see that the Goomba was inheriting from the *Actor* class within the engine package. As a result, I chose to make the new Koopa class also extend from the *Actor class* as the two enemies shared many attributes and

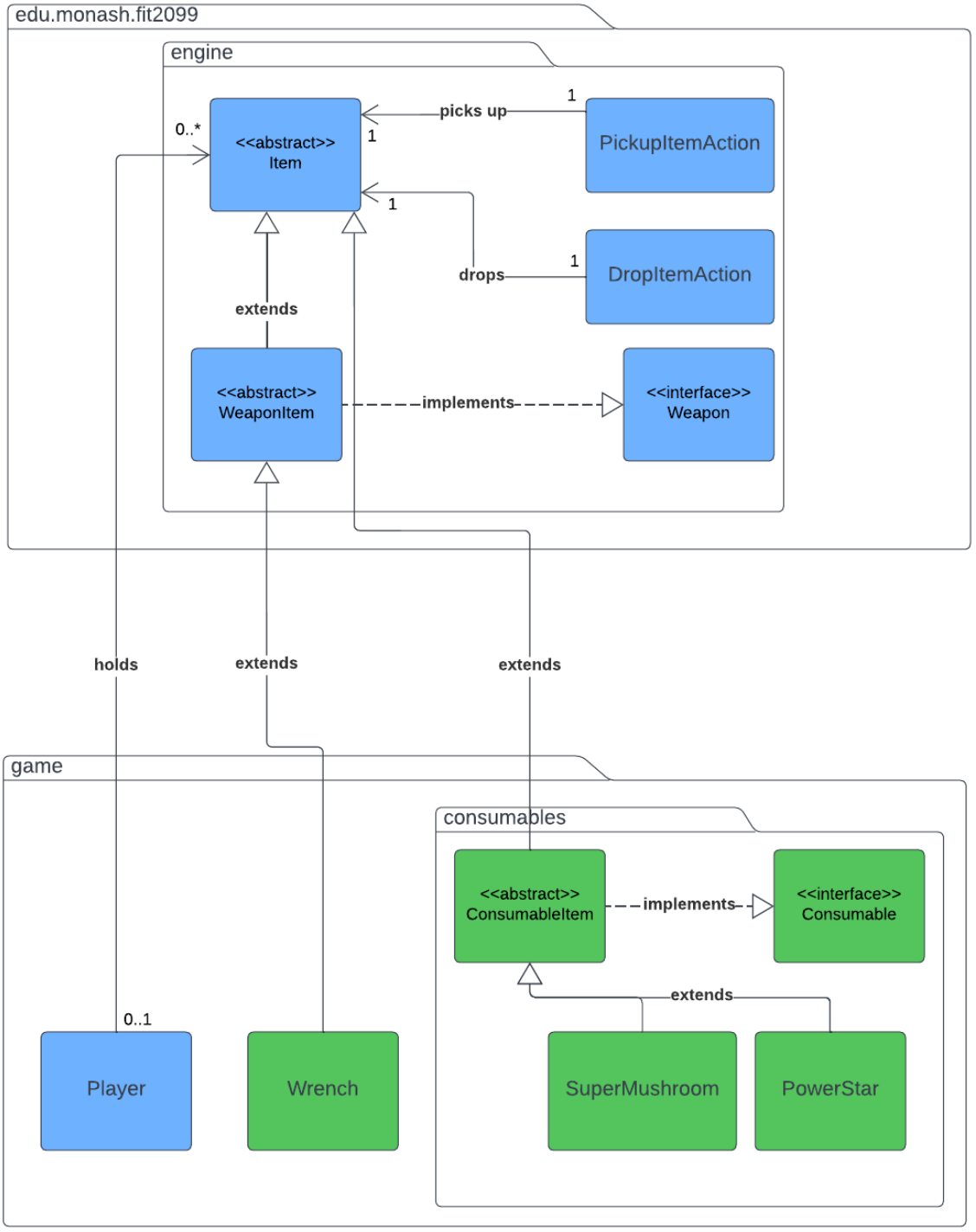
methods. This is in adherence to the Open-closed principle which makes use of abstraction to ensure that the *Actor* class remains unchanged when adding a new enemy class. However, the Player class also extends from *Actor* and has a vastly different role, and along with that different behaviours from the enemies. To combat this I added an extra layer of abstraction with the *Enemy* class as a means of further separating the two types of *Actor*. Whilst in many cases multi-layered abstraction can make debugging more difficult, in a case where there are enough shared attributes/methods between two children of a node, it warrants another class. This will also make it flexible if we need to implement more enemies in A3. Additionally, I also added the *Enemy*, Koopa and Goomba classes to an 'enemy' package. The sequence is demonstrative of the implementation details for requirement 3, where some Sprouts spawn a Goomba up until a point, whilst also showing the interaction between Mario and the Koopa from its unconscious state up until its death. Note that occasionally I have simplified a group of actions to a single method call for the sake of simplicity such as the 'attack()' method, which in the main program would likely be a series of calls between classes.

Assignment 2 Rationale Update:

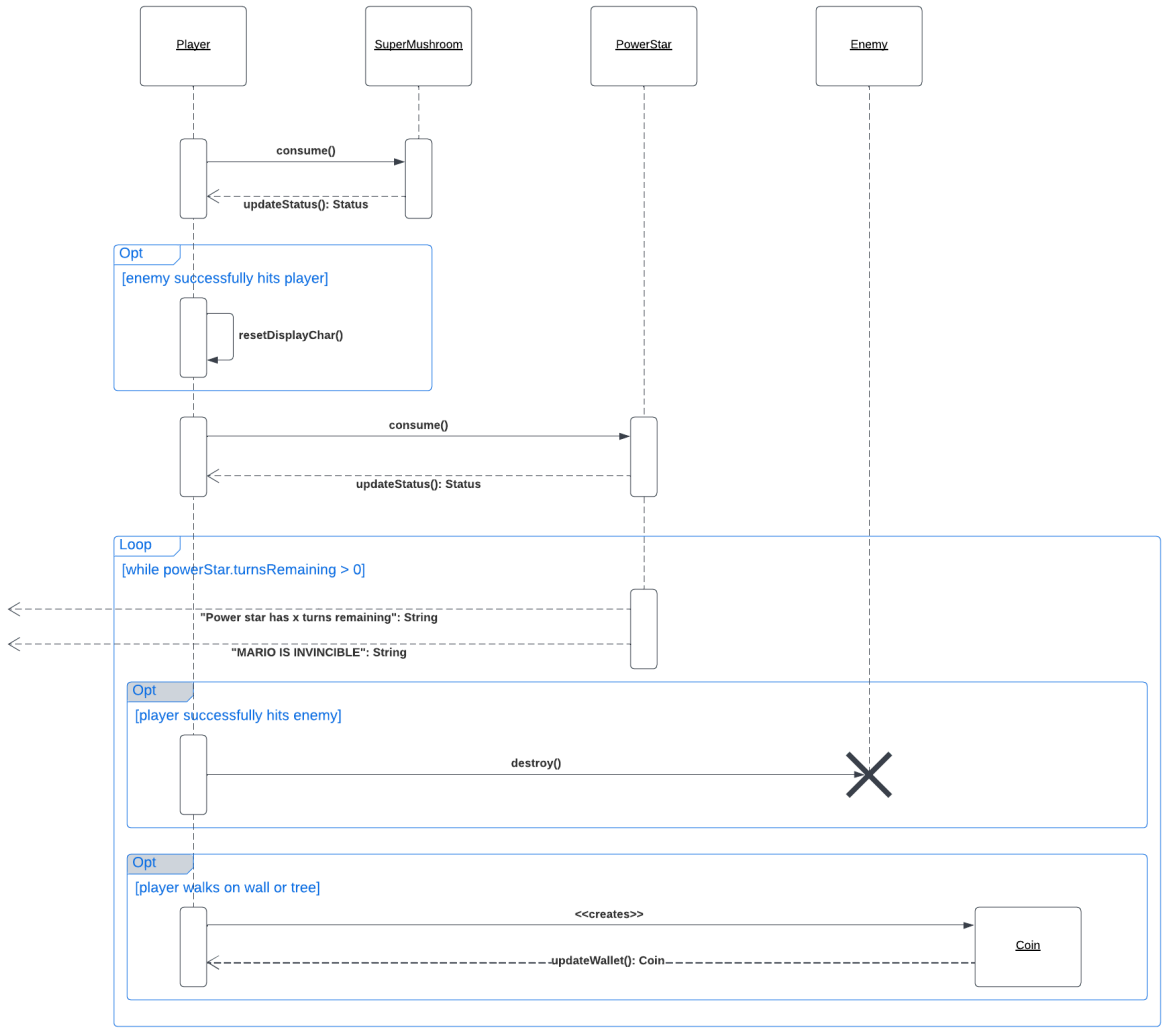
Not much has changed since the initial design of Assignment 1. I have made use of inheritance in the Enemy class to implement some common behaviours and attributes between the different subtypes of Enemy (Koopa & Goomba) for the reasons mentioned above. This will also allow for more enemy types if required to do so in Assignment 3 with minimal changes (Open-Closed). As for the modifications, more packages have been added to better support encapsulation and organisation within the program, shown in the updated UML class diagram with both the 'behaviours' and 'highgrounds.trees' packages being implemented. Additionally, whilst the Capabilities array that stores the list of 'Status' that an actor may take is within the Actor class, I felt that it was important to reference it in the context of Enemies specifically since it was instrumental in determining the state of an Enemy. Most importantly, it was used to determine whether or not an enemy (the Koopa) had the capability to go DORMANT, stopping it from being killed, taking further action, and changing its icon on update. Building upon this, the ExecuteAction was a new Action created to enable the player to break the shell of the Koopa (remove it from the map) if it is DORMANT and if they have a Wrench equipped. The execute action stores a reference to its target ie. the Koopa, yet this is in the form of an Actor reference. This allows for flexibility in case the ExecuteAction expands to include more Enemy types in future or even Mario.

Req 4: Magical Items

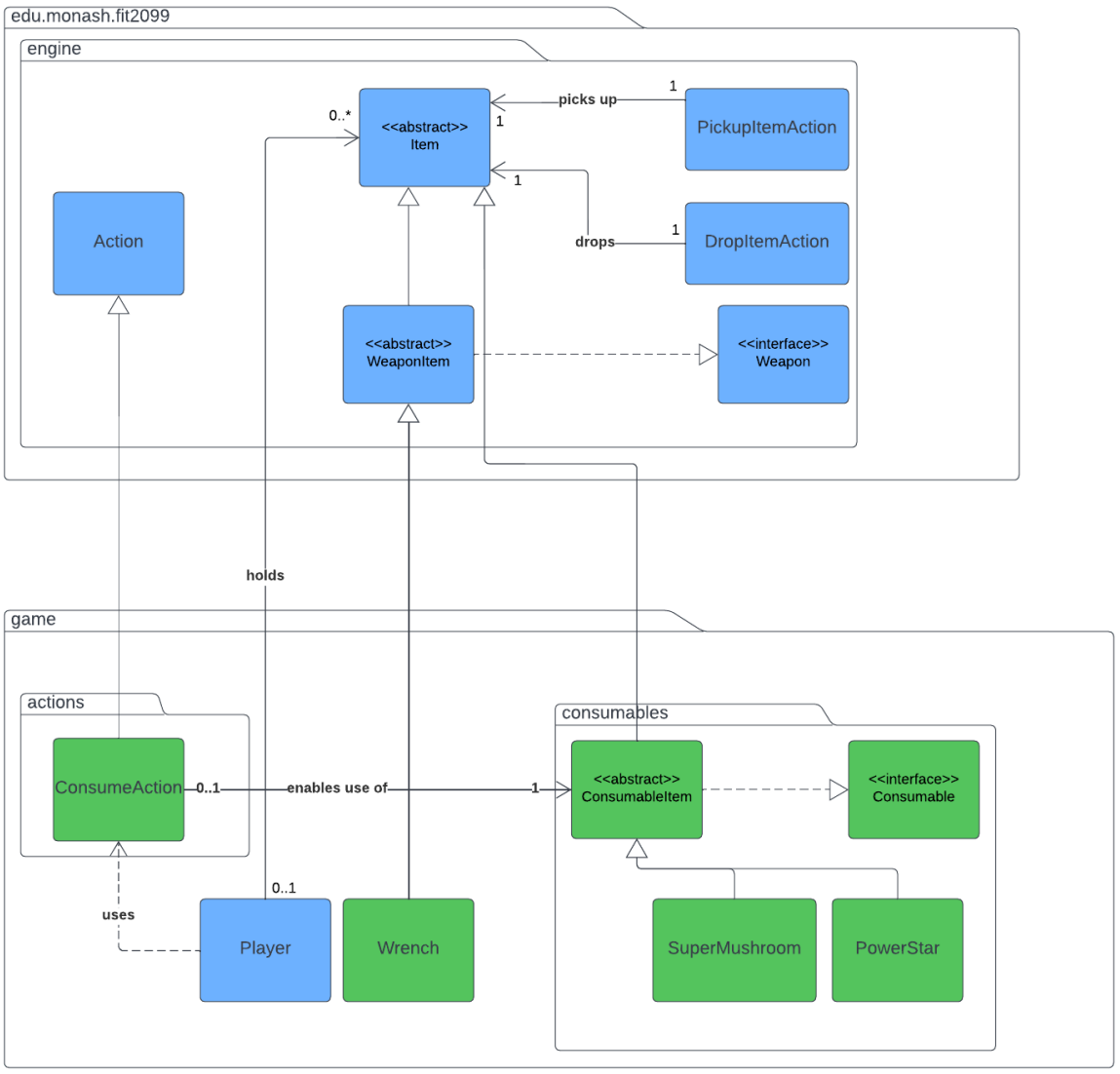
Class Diagram:



Sequence Diagram:



Updated class diagram (Assignment 2):



Design Rationale:

Consumable	A set of methods implemented by consumable items.
ConsumableItem	An Item that can be consumed (abstract).
Wrench	Used to break Koopa's shell (execute).
PowerStar	On consumption, makes Mario invincible, spawns coins on walls/trees and insta-kills enemies when hit. Resets after a set time.
SuperMushroom	On consumption, increases Mario's maximum health and changes his display char to be a capital 'M'. Resets when hit.

Initially, I started the class diagram for requirement 4 with the three items: Wrench, PowerStar and SuperMushroom all extending from the *Item* class. I quickly realised however that this is a bad design as it does not adhere to the Single Responsibility Principle. The items can be further subdivided into Weapons and Consumables. After I saw that within the engine there was a *WeaponItem* class that extended *Item* and implemented a Weapon interface, I sought to mirror this design by creating a *ConsumableItem* class that extends *Item* and implements *Consumable*. This gives a lot of flexibility when adding Items in future and adheres to the SRP, ISP and DIP. The sequence diagram shows the sequence of implementation details in requirement 4. Note that the updateStatus() method after the Player consumes() a *ConsumableItem* will change the Status enum for a player, which will then be checked in a tick and the corresponding actions/changes to attributes will take place. Additionally, the messages leading to no entity are Strings printed to the console.

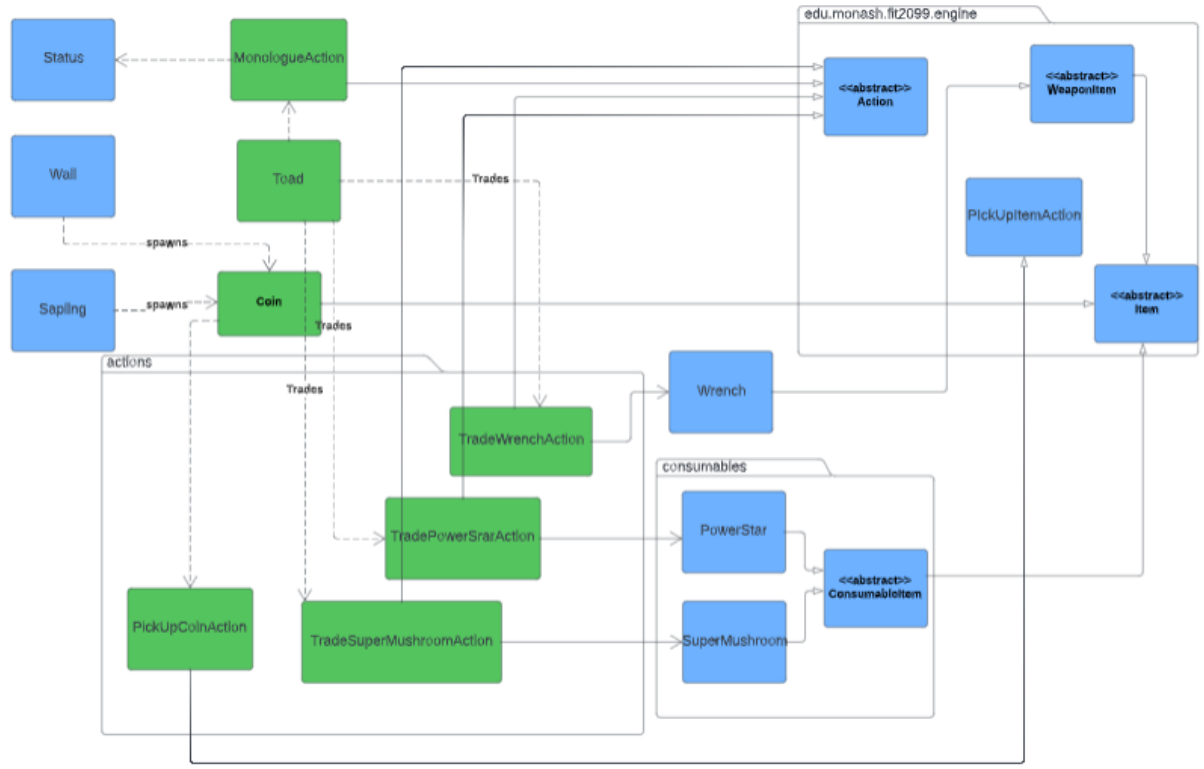
Assignment 2 Rationale Update:

ConsumeAction	Action used by the Player to consume items within their inventory.

Similar to requirement 3, I was mostly faithful to my original design plan when creating this class. There were, however, two key changes in order to allow for this to work more effectively. Firstly, the Player class is now a singleton. The benefit of this is that it ensures that there is only one player, and as a result, we can access that player at any point in our code, saving the need for an endless number of associations to get the player reference to our desired classes. Of course, this is a double edged sword and it can allow for data leaks and/or unwanted manipulation of the object. Additionally, it limits the game to only having one player. In this case we decided that the design pattern provided more benefits than cons, yet it could be implemented in a better way most definitely. The second major change was the use of the ConsumeAction, which stores a reference to a ConsumableItem and calls the relevant consume() function depending on the item (implemented from Consumable Interface). This adheres to the SRP as it does a better job of dividing up responsibilities between different classes. It also follows the Open-closed principle as it makes use of the Action class without directly modifying it. It fits in with the overall program design pattern of making anything that Mario can perform an Action that appears in the menu, and so it made sense to follow this pattern.

Req 5: Trading & Req 6: Monologue

Class Diagram:



Sequence Diagram (Req 5 & Req 6):

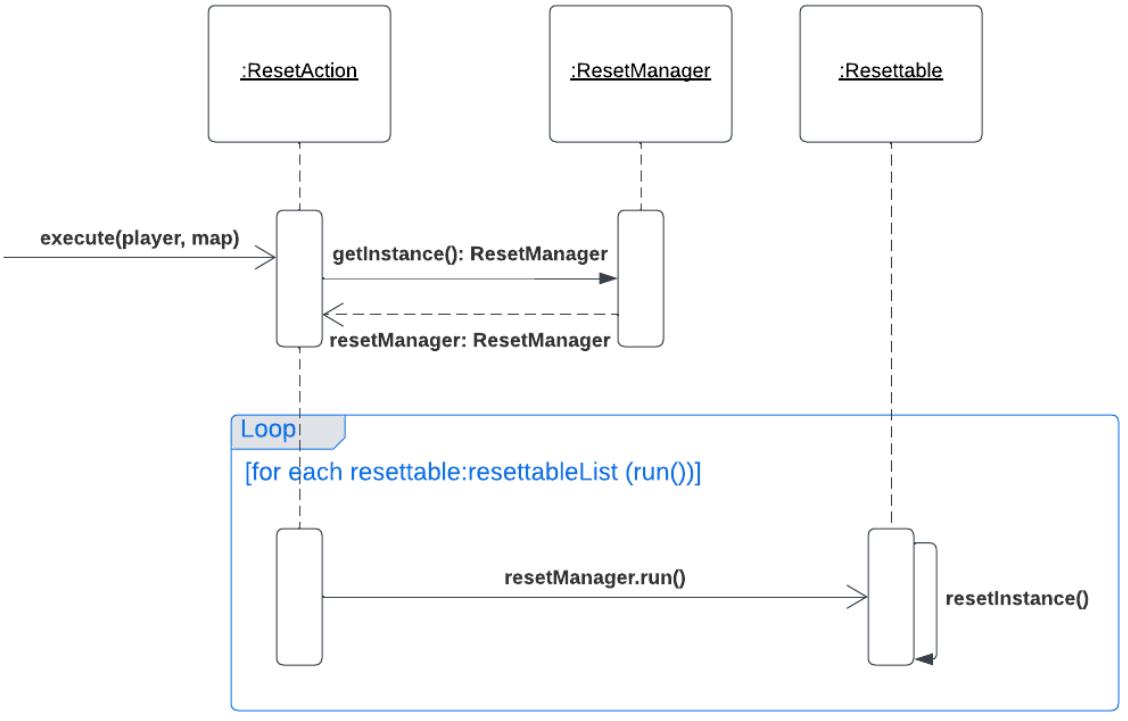
Design Rationale Update Assignment 2:

For this design, the coin is extended out to be an item which will be spawned by sapling and wall. The coin will then use the pickup coin action and the pickup coin action is extended to pick up item action which overrides the execute activities that will add coin.value to the wallet which is an attribute stored in the player. Toad has 4 actions which are monologue action, TradeSuperMushroomAction, TradeSuperStarAction, TradeWrenchAction. For monologue action, it is going to check the player's status to see whether the player has super mushroom empowered. This class is then extended from the action. For trade super mushroom, trade superstar, trade wrench actions the implementations are the same those three are extending from actions and when the trading happens it will check the wallet balance, reduce the wallet price according to the price of items and then add to player's inventory.
This design is more reasonable compared to the previous design where the coin is an abstract class that has different classes assigned to it. A simple attribute value can just replace it. All of the actions are extended to actions and items are extended to items attribute. Each class has their own responsibility which satisfies the single responsibility principle.

Req 7: Reset Game

Class Diagram:

Sequence Diagram:



Design Rationale:

Class ResetAction extends Action, adding a new action available to the player at any time which will reset most aspects of the game. Tree, Coin, Enemies, and Player all implement the Resettable interface, and all have their specific resets. When resetInstance is called, all Enemies and Coin instances are completely deleted. Note that they are deleted and not killed, as we don't want them to proc any on death effects such as dropping coins. When resetInstance is called on Tree's subclasses, trees are deleted and replaced with dirt 50% of the time on a per-instance basis. Finally, when resetInstance is called on the Player, their current power-up status is reset.

While each of the four classes (Tree, Coin, Enemies, and Player) implement their resetInstance method and could theoretically be called independently e.g. resetting Enemies but none of the other classes, this won't occur as the ResetManager, and by extension, the ResetAction will call resetInstance on all instances of all four classes.