Assignment 1. Work Breakdown Agreement

- [ ] Class Diagram
- [ ] Interaction Diagram
- [ ] Design rationale

- [ ] Work Breakdown Agreement

Class Diagram

- Tasks: REQ4, REQ5, REQ7
- Assignee: Chun
- Completed by: 4/4/2022
- Reviewer: James

* Completed by: 8/4/2022

- Tasks: REQ1
- Assignee: Matthew
- Completed by: 4/4/2022
- Reviewer: Chun

* Completed by: 8/4/2022

- Tasks: REQ2, REQ3, REQ6
- Assignee: James
- Completed by: 4/4/2022
- Reviewer: Matthew

* Completed by: 8/4/2022

Interaction Diagram

- Assignee: Matthew
- Completed by: 4/4/2022
- Reviewer: Chun, James

* Completed by: 8/4/2022

Design Rationale

- Assignee: Chun, James, Matthew
- Completed by: 4/4/2022
- Reviewer: Chun, James, Matthew

* Completed by: 8/4/2022

Signed

- Matthew Siegenthaler: I accept this WBA
- Chun Mok: I accept this WBA
- James Huynh: I accept this WBA

FIT2099 Assignment 1: Design Rationale

Lab 6 Team 3

REQ1: Trees

The tree stages were implemented as 3 subclasses that inherit from the abstract class **Tree**. Since all tree stages are expected to grow to a new stage after a set amount of turns, The number of turns required to grow should be stored as an instance variable as part of **Tree**. Once the requisite amount of time to grow has passed, a new object of the next tree stage may be returned. Once 5 turns has passed to grow for the Mature stage, the growth timer may be reset back to 5 to for it's to regrow a new sprout.

As sprouts require fertile land to grow, we have chosen to give **Ground** objects that are fertile the capability **FERTILE** in the enumeration **GroundTraits**. This is as future **Ground** tiles may also be fertile tiles, requiring a unified identifier for all such tiles. This allows us to follow the *"Don't repeat yourself"* design principle through use of public constants. **Ground** already features a **CapabilitySet** which we can use to store the enumeration value.

To check for adjacent **FERTILE Ground** for **Mature**, we will check the current location's **Exits** if they have the capability of **FERTILE**. This is as we know the surrounding exits are adjacent to the location. We can then randomly select one of the **FERTILE** locations and replace its **Ground** with **Sprout**.

Pros

- Safeguarding implementation of grow() and growTurns attribute.
- Able to identify all growth stages as a Tree class.
- Checking **Exit**s ensures **Location**s are adjacent to **Mature**

Cons

- Additional code required to implement abstract class.
- Requires **Mature** to handle setting ground of **FERTILE** at location.

REQ2: Jump

JumpActorAction extends **MoveActorAction**: We want **Player** to have an action to move to another location in the **GameMap**, the **MoveActorAction** has the methods to enable **Player** to have this ability. This will overide the execute() method to include the success rate check for jumps and if **Player** has the **TALL** status. It will also override the description() method of the action.

JumpManager is associated with **(interface) Jumpable**: We want **JumpManager** to store the **Jumpable** grounds.

(abstract) Tree and **Wall** implements **(interface) Jumpable**: We want only **Tree** and **Wall** grounds to be jumpable but not others, such as **Dirt**. Hence, we use interface. This will also add the **JumpActorAction** to the **ActionList** to allow **Actor**'s with the **CAN_JUMP** status the ability to jump to the **Jumpable** terrain.

Pros

- This follows the SOLID principle *"the Dependency Inversion Principle"* allowing for other objects (i.e. **JumpableManager**) to depend on the abstraction **Jumpable** rather than the **Ground** classes that are jumpable.

Cons

- Use of abstraction - slower time complexity and more use of resources (less efficient).

REQ3: Enemies

Goomba and **Koopa** both extends the abstract class **Enemy**, and **Player** extends the abstract class **Friendly**. As we know enemies have different methods compared to friendlies, e.g enemies are able to wander and follow the player but player can't. Indeed, both enemy and friendly extends actor. Enemies would have the **AttackAction** applied to its action list by default for the **Player** to attack it. This can be overriden for different behaviour such as **Koopa**

Once a **Koopa** is damaged enough, it will be given the capability **DORMANT** to signify it cannot be attacked or move. We can choose to add the **AttackAction** to the **Koopa** action list if it is not **DORMANT**, preventing other **Actor**s seeing the option when next to it.

AttackBehaviour, **WanderBehaviour**, **StationaryBehaviour** and **FollowBehaviour** all implement **(interface) Behaviour**, so can act these behaviours without the player's input. The **AttackBehaviour** should take in a target **Actor** upon construction and stored as an instance variable like **FollowBehaviour**. This would allow us to set enemies to attack the **Player** when nearby.

Player holds **Wrench** If **Player** is holding **Wrench** in the inventory, then **SmashShellAction** can be called to kill **Koopa**. This will drop a **SuperMushroom** on its location upon death.

The **Goomba**'s kick and **Koopa**'s punch will be implemented by overriding the **getIntrinsicWeapon()** method of **Actor**. Here we would specify the attack it performs by setting the verb and damage *(the hit rate is the default 50%)* of **IntrinsicWeapon**'s.

Pros

- Easier to understand and follow, e.g. an **Action** is used on the object being attacked
- Separating Actors into either enemies or friendlies (abstract classes) allows simple, clear divide between two Actors with different methods.
- IntrinsicWeapon best describes the type of attack for **Goomba** and **Koopa** and enables us to set the damage.

Cons

- More space and time complexity is required for abstraction.
- Cannot override the hit rate of **IntrinsicWeapon** for future enemy implementations as it is default to 50%.

REQ4: Magical Items

A new abstract class **MagicalItem**, which is inherited from the base class **Item**, has been added. **PowerStar** and **SuperMushroom** are implemented as inheriting **MagicalItem**. This is done to differentiate 2 types of **Item**: item that can be equipped as a weapon - **WeaponItem**; item that is not a weapon - **MagicalItem**. **MagicalItem** is consumable, which provides status to the **Actor**.

Eating a **SuperMushroom** will increase the **Player**'s max hp with **Actor**'s increaseMaxHP() method. This will also give the **Player** the status capability **TALL** to signify he has eaten the mushroom *(This is indicated with the icon M)*. Upon taking damage, if **Player** has **TALL**, it will remove the capability.

The **PowerStar** item will be given the **FADING** status. This counter for its duration would be stored as part of the **PowerStar** item, decreasing every tick until its removal.

High ground may check to see if the **Player** is standing on it and has the **INVINCIBLE** effect. If so, it will convert to dirt and drop a coin. Upon taking damage, we check the **Player** has the **INVINCIBLE** status to see if we are dealt damage.

The **AttackAction** may also check for the **INVINCIBLE** status of the actor to instantly kill a target when attacking after checking that **Player** successfully hits.

Pros

- More defined purpose of items
- Avoid the need of implementing 2 interfaces
- Future-proof for adding new features, like an inventory system

Cons

- Increased code sized
- Potentially harder to debug

REQ5: Trading

Two new classes, **Toad** and **Coin**, are added in this section. **Toad** serves as the item shop in this game, and **Coin** is the currency for buying items from **Toad**. **Coin** is inherited from **Item** as, like all items, coins can be picked up from the ground. However, it is not inherited from either **WeaponItem** or **MagicalItem**, as it can not be equiped as a weapon or consume to gain status. **Coin** should have 1 Integer attribute to represent its value and 1 static String attribute to represent its visual to be displayed. **Player** should also have an integer value tracking the amount of currency they have.

Pros

- No additional class to manage currency of only **Player**.

Cons

- Not as simple as each item having an Integer attribute representing its cost/player having an Integer attribute representing how much the player have.
- Requires other classes to implement class attribute to store total amount of currency. *(currently, this is exclusive to player)*.

REQ6: Monologue

GameMap and **Location** keep track of where **Toad** and **Player** are. If close enough, **SpeakAction** is available as an option: grabs monologue from **Toad**, checks if **Player** contains **Wrench**, **Status** of a PowerStar or else the **CapabilitySet** of the Actor e.g cannot be Goomba and decides what to say. Uses **Display** to print monologue string.

Player holds **(abstract) WeaponItem**, and is extended by **Wrench**. Use of **(abstract) WeaponItem** from engine, ease of implementation of new weapons in the future. **(enum) Status** able to contain the different statuses for **Player**

Pros

- Choice to utilise enumeration will avoid the excessive use of literals, and hence improve maintainability and extensibility of the code in the long-term.

Cons

- **Toad** isn't directly responsible for calling **SpeakAction**, but rather relies on **GameMap** and **Location**, e.g, **Player** must close enough in coordinates to **Toad** for **SpeakAction** to be available in console (could be a possible way for Toad to directly call **SpeakAction**).

REQ7: Reset Game

A **Resettable** interface and a **ResetManager** class are created. All classes that can be reset, like **Player** and **Enemy**, implments this interface. And **ResetManager** manages the reset process. This follows the *Dependency Inversion Principle* which depends on the *Resettable* interface. Classes that implement *Resettable* will only depend on it.

Pros

- Easier to debug this features as all the reset method are implemented in this class
- Able to specify which items are reset with the **Resettable** interface, following the *Interface Segregation Principle* to solely be responsible for what occurs to the item during a reset.
- ResetManager tracks *Resettable* instances allowing us to query it to get all resettable instances.

Cons

- Increased code size and complexity

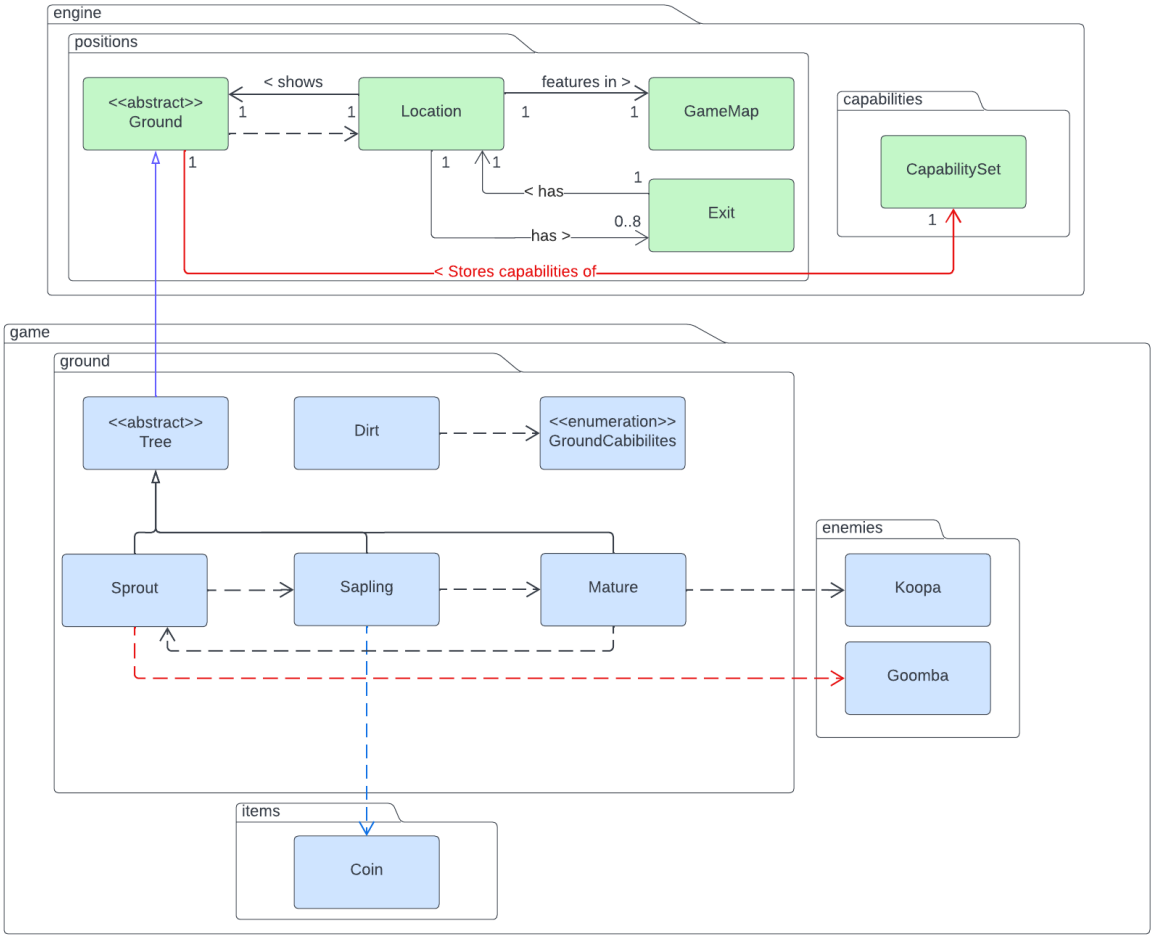
Assignment 1

Matthew Siegenthaler, James Huynh, Chun Kau Mok

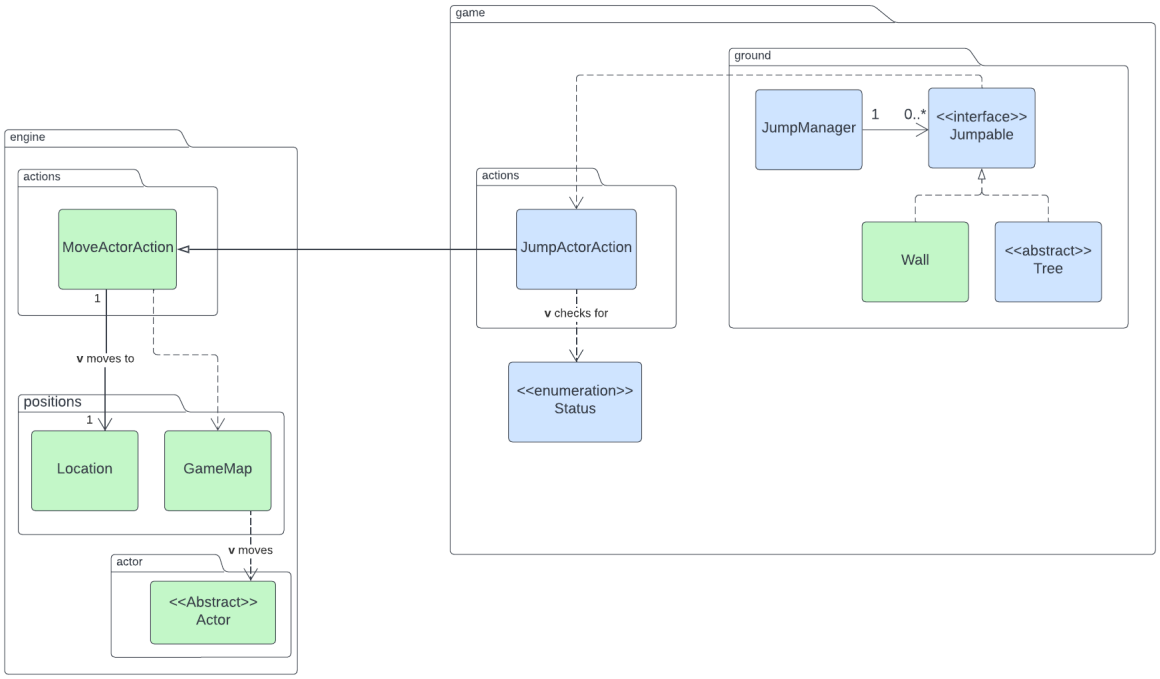
Contents

Class diagrams

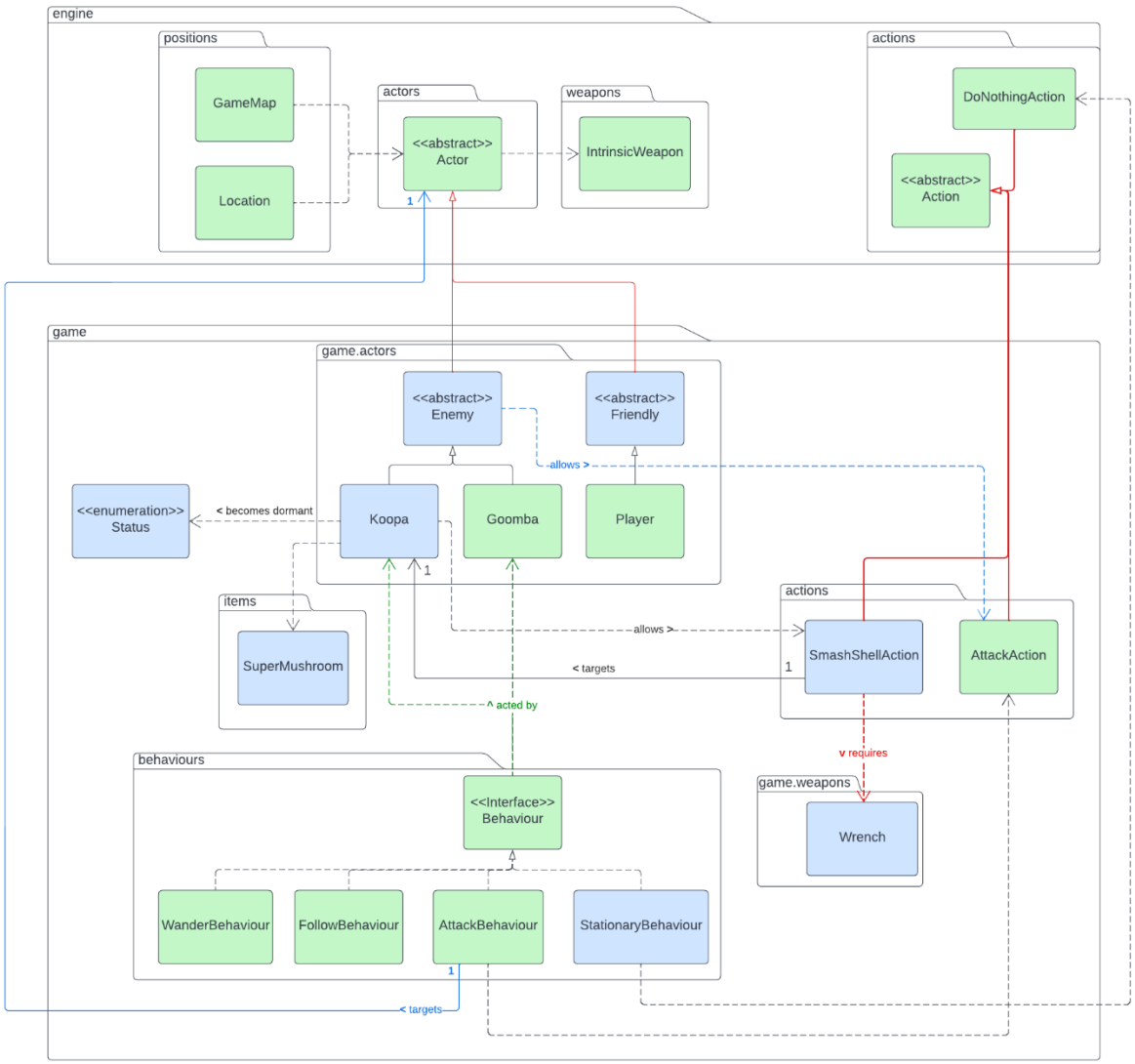
REQ1



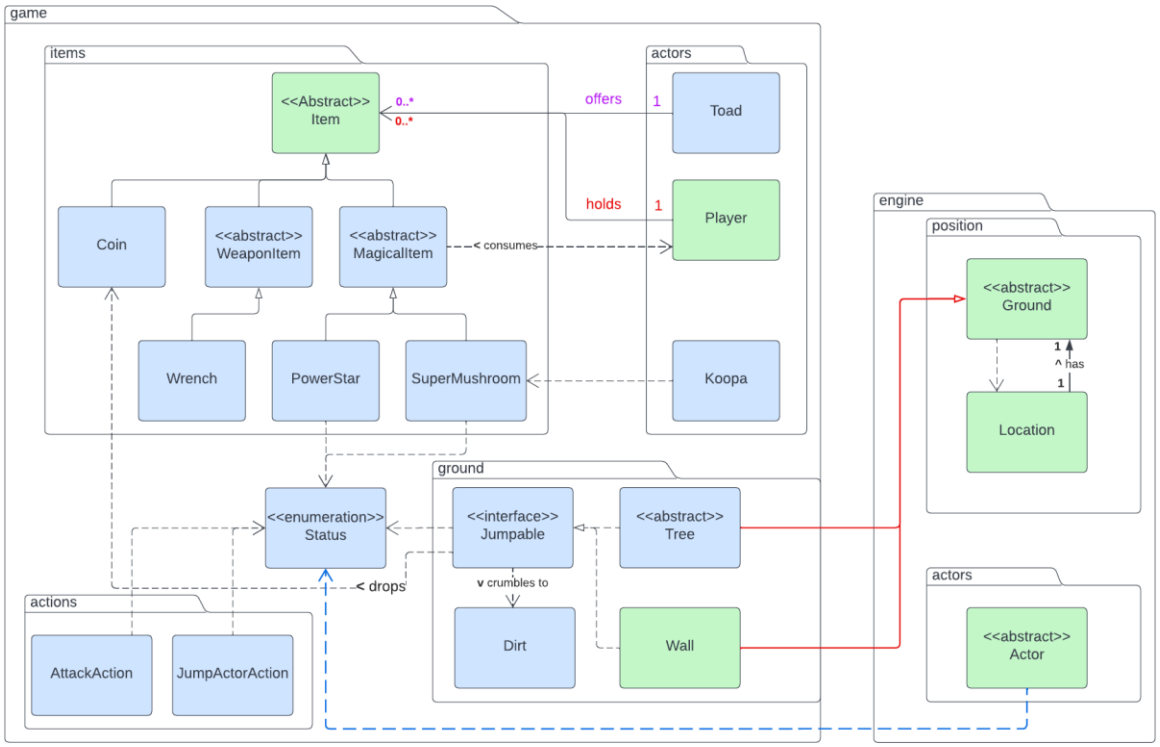
REQ2


game
ground
JumpManager
1
0..*
<<interface>>
Jumpable
Wall
<<abstract>>
Tree
actions
JumpActorAction
v checks for
<<enumeration>>
Status
engine
actions
MoveActorAction
1
v moves to
positions
1
Location
GameMap
v moves
actor
<<Abstract>>
Actor

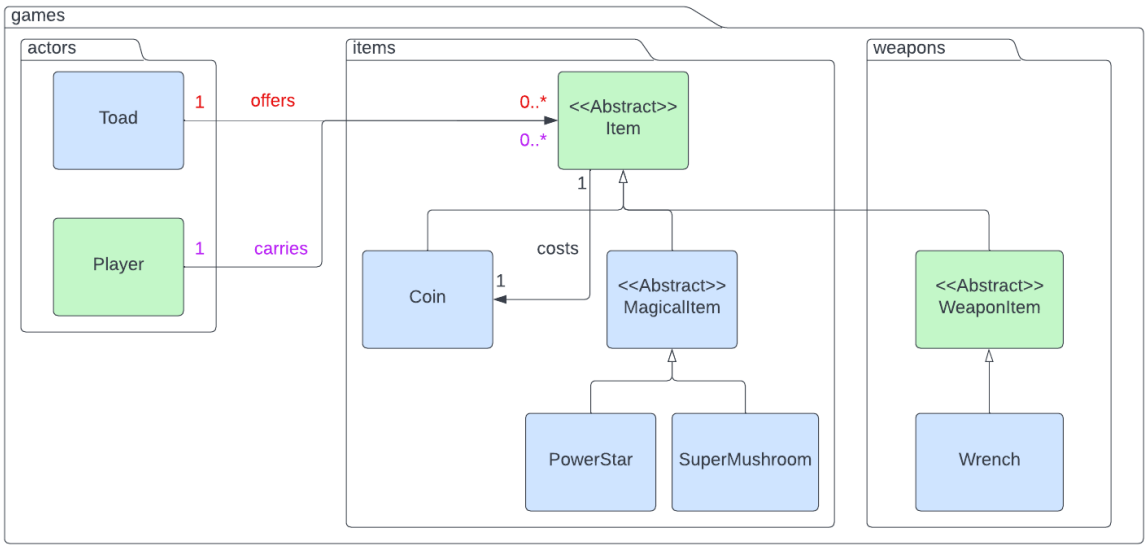
REQ3


engine
positions
GameMap
Location
actors
<<abstract>>
Actor
1
weapons
IntrinsicWeapon
actions
DoNothingAction
<<abstract>>
Action
game
game.actors
<<abstract>>
Enemy
<<abstract>>
Friendly
allows >
<<enumeration>>
Status
< becomes dormant
Koopa
Goomba
Player
1
items
SuperMushroom
actions
allows >
SmashShellAction
AttackAction
1
< targets
^ acted by
v requires
behaviours
<<Interface>>
Behaviour
game.weapons
Wrench
WanderBehaviour
FollowBehaviour
AttackBehaviour
StationaryBehaviour
1
< targets

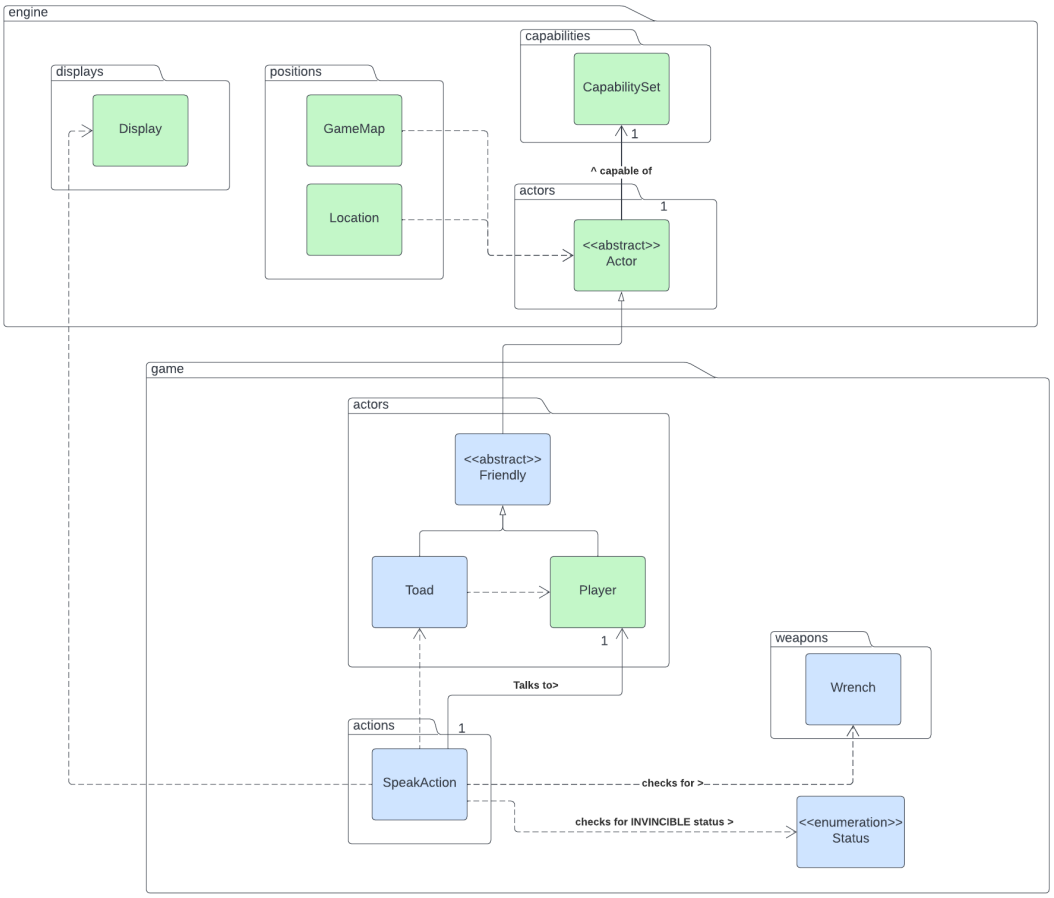
REQ4


game
items
actors
<<Abstract>>
Item
0..*
0..*
offers
1
Toad
holds
1
Player
Coin
<<abstract>>
WeaponItem
<<abstract>>
MagicalItem
consumes
Wrench
PowerStar
SuperMushroom
Koopa
engine
position
<<abstract>>
Ground
1
^ has
1
Location
ground
<<enumeration>>
Status
<<interface>>
Jumpable
<<abstract>>
Tree
drops
v crumbles to
Dirt
Wall
actors
<<abstract>>
Actor
actions
AttackAction
JumpActorAction

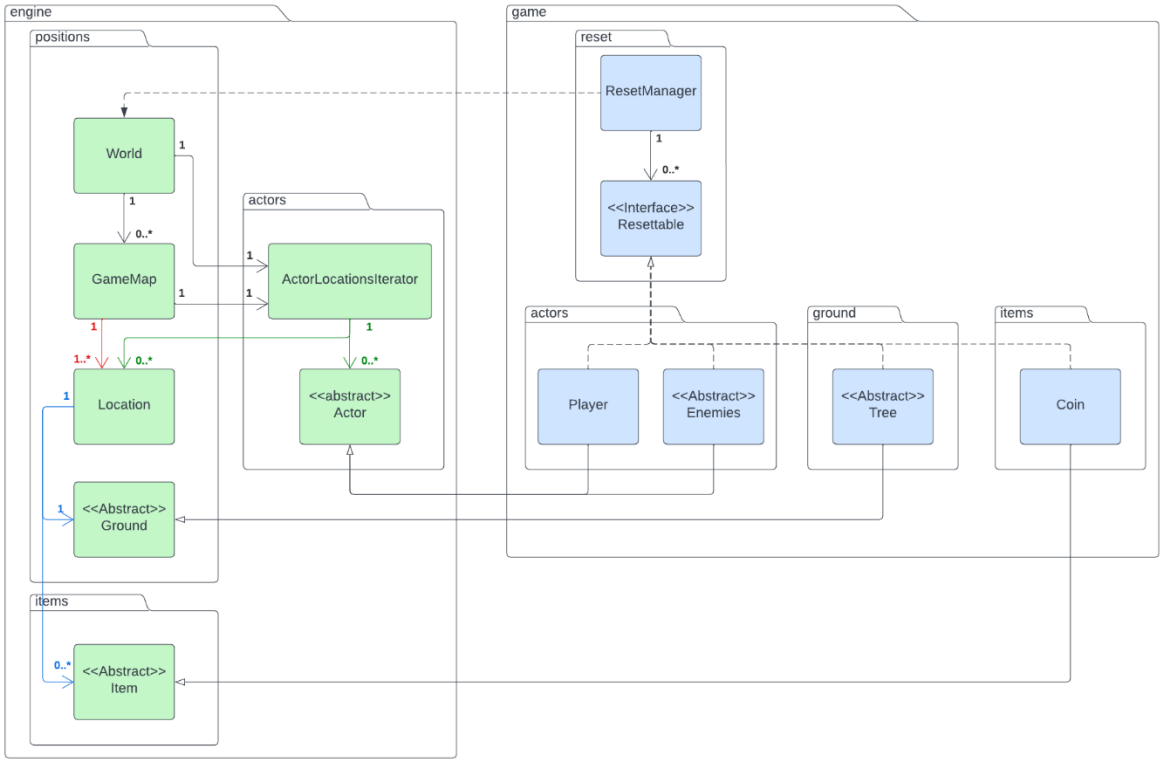
REQ5


games
actors
Toad
1
offers
Player
1
carries
items
0..*
0..*
<<Abstract>>
Item
1
costs
Coin
1
<<Abstract>>
MagicalItem
PowerStar
SuperMushroom
weapons
<<Abstract>>
WeaponItem
Wrench

REQ6

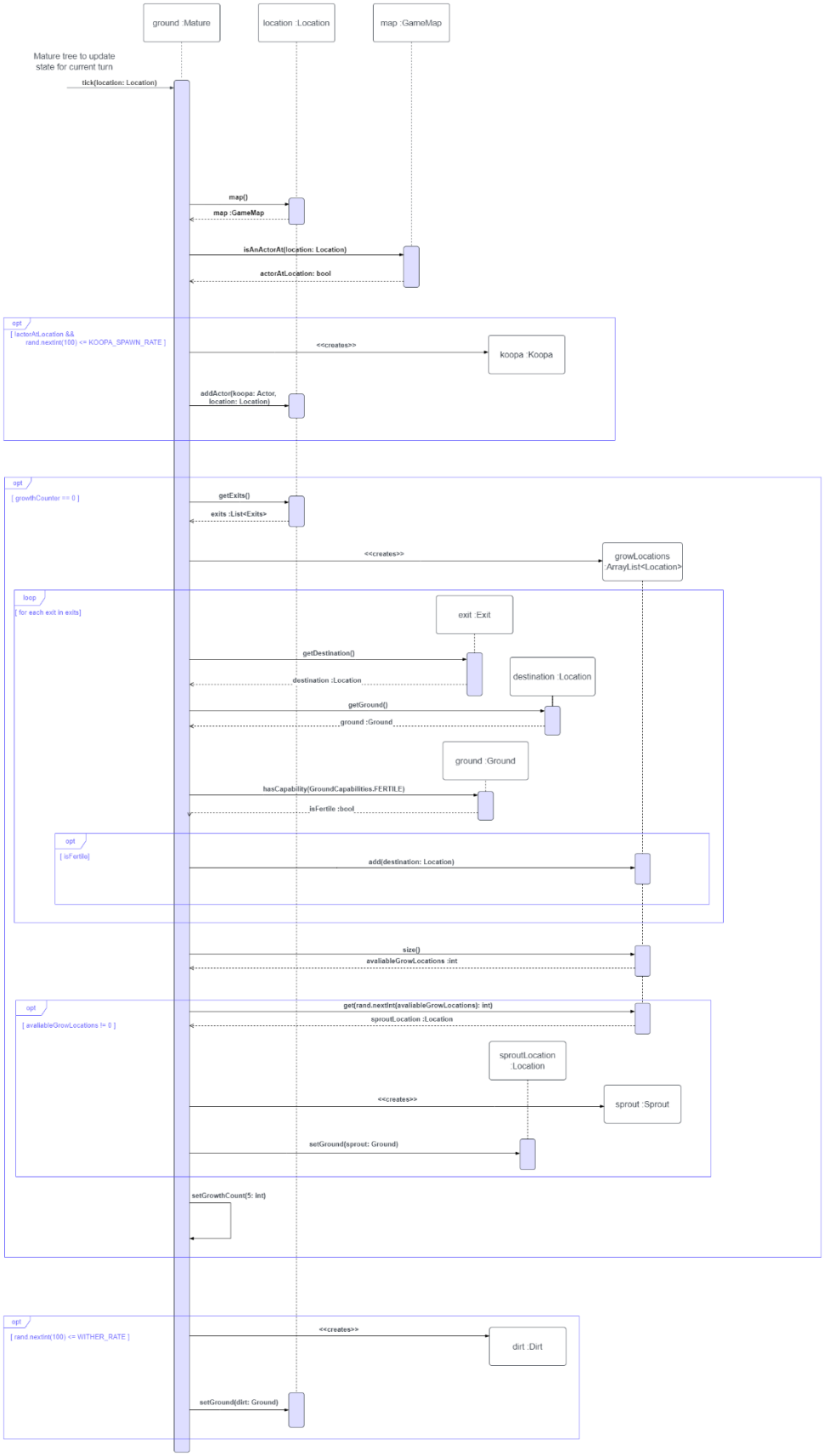

engine
displays
Display
positions
GameMap
Location
capabilities
CapabilitySet
1
^ capable of
actors
1
<<abstract>>
Actor
game
actors
<<abstract>>
Friendly
Toad
Player
1
weapons
Wrench
Talks to>
actions
1
SpeakAction
checks for >
checks for INVINCIBLE status >
<<enumeration>>
Status

REQ7

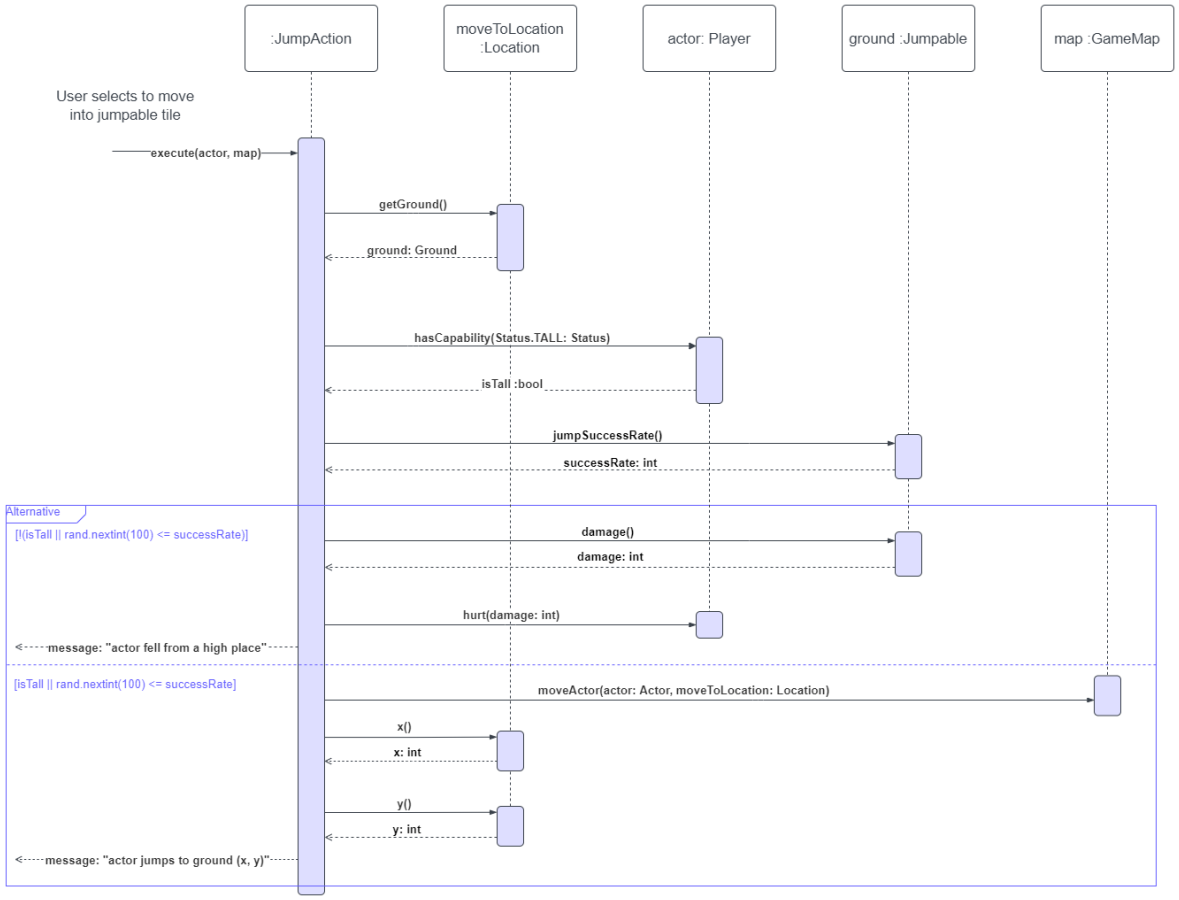

engine
positions
World
1
1
0..*
GameMap
1
1
1
1..*
0..*
Location
1
<<Abstract>>
Ground
1
actors
1
1
ActorLocationsIterator
1
0..*
<<abstract>>
Actor
items
0..*
<<Abstract>>
Item
game
reset
ResetManager
1
0..*
<<Interface>>
Resettable
actors
Player
<<Abstract>>
Enemies
ground
<<Abstract>>
Tree
items
Coin

Sequence diagrams

REQ1 Mature.tick()



REQ2 JumpActorAction.execute()



REQ5 trading