Requirement 4

Title: Crossing Rivers

Description: In your starting map, add a water ground(River ~) to create a lake. To cross the lake, Mario must swim. Mario may get wet as he is swimming across the lake. A wet Mario will not receive damage from fire grounds. This wet effect lasts for 20 turns or until Mario receives damage from a fire ground.

Implementation expectations:

- Create a bunch of water grounds together to form a lake.
- When Mario is next to a river ground, Mario should have a swim action to swim inside the lake. Enemies cannot swim.
- While Mario is swimming, Mario has a 20% chance to get wet status.

Explanation why it adheres to SOLID principles (WHY):

S Single Responsibility Principle (SRP)

O Open-closed Principle (OCP)

L Liskov Substitution Principle (LSP)

I Interface Segregation Principle (ISP)

D Dependency Inversion Principle (DIP)

- River is a a class on it's own, because it has it's own responsibilities(only allowing actors that can swim to enter). This is related to SRP.
- River is a ground, thus it extends from the abstract Ground class. For example, all grounds can tick, which allows River to give Mario wet status. Since the concrete class, River depends on the abstract Ground class, this is related to DIP.
- SwimAction extends from the action class. It has it's own responsibility, that is allow an actor to move across a River. This is related to SRP.
- By using the abstract action class, River can return a swimAction to an actor via the allowable actions method. This is related to DIP, because we use the abstract action class to implement our concrete swimAction. This prevents excessive use of instanceof (eg. if instance of swimAction, do something).
- Swimmable is an interface that allows actors to swim. It has only one responsibility and it is a small and specific interface, thus it complies with SRP and ISP.
- Since we have the swimmable inteface, any additional actors that want to swim in the future only need to implement this interface to be able to swim. This is in line with OCP (Do not need to modify River to allow other actors to enter)

Requirements	Features (HOW) / Your Approach / Answer
Must use at least two (2) classes from the engine package	~ River is a ground, so we are extending from the ground class.(Reusing engine) ~ swimAction extends from action class. (Reusing engine)
Must use/re-use at least one(1) existing feature (either from assignment 2 and/or fixed requirements from assignment 3)	~ Mario can get wet, which is a status. Wet status gives immunity from the damage from fire grounds. (using existing feature from ass3 req2)
Must use existing or create new abstractions (e.g., abstract or interface, apart from the engine code)	~ Swimmable Interface (New abstraction)
Must use existing or create new capabilities	~ Mario can get wet, which is a status. Wet status gives immunity from the damage from fire grounds. (New capability)

Requirement 5

Title: Bucket of Water

Description: In the middle of the lake present in the starting map (REQ 4), place a bucket (@) for the player to pick up and use. It can store water from the river and can only be used once before being required to be refilled from the lake again.

Implementation Expectations: - player can pick up bucket from a land in the middle of the lake

- player can store up to once instance of water
- player can preemptively pour water on themselves to acquire wet status (last 20 turns)
- player can choose to pour water on ground that's on fire to put the fire out

Explanation why it adheres to SOLID principles (WHY):

S Single Responsibility Principle (SRP)

O Open-closed Principle (OCP)

L Liskov Substitution Principle (LSP)

I Interface Segregation Principle (ISP)

D Dependency Inversion Principle (DIP)

- Bucket is a a class on it's own, because it has it's own responsibilities(storing water to be poured). Although water can be stored in the bottle, that water is meant to be drunk. This is related to SRP, since we do not combine the bottle and bucket classes.
- Bucket is an item, thus it extends from the abstract Item class. This is inline with DIP, by using abstractions, we can reuse our pickUpItem action, instead of having to code a pickUpBucket action.
- PourAction and SelfPourAction are seperate actions because they have their own responsibilities. This is related to SRP.
- PourAction and SelfPourAction both extend from abstract Action class. This layer of abstraction allows us to execute actions, without depending on the concrete implementation of the action. This is related to DIP.
- Both and bucket will implement Fillable interface. It is a small interface with a specific responsiblity, that is an item can be filled up. This is related to ISP.
- By using abstraction of the fillable interface, we can have a fillUp method inside the interface to allow bottle and bucket to be filled up respectively. The layer of abstraction ensures that fillUpAction doesn't need to know the concrete implementation of how bottle or bucket is filled up. Thus, this is related to DIP.

Requirements	Features (HOW) / Your Approach / Answer
Must use at least two (2) classes from the engine package	~ Bucket is an item, so we are using the Item class. (Reusing engine) ~ Fill, SelfPour and Pour action extend from action class, so we are using the Action class. (Reusing engine)
Must use/re-use at least one(1) existing feature (either from assignment 2 and/or fixed requirements from assignment 3)	~ Mario can get wet, which is a status. Wet status gives immunity from the damage from fire grounds.(using existing feature from ass3 req2)
Must use existing or create new abstractions (e.g., abstract or interface, apart from the engine code)	~ Fillable interface (new interface)
Must use existing or create new capabilities	~ Mario can get wet, which is a status. Wet status gives immunity from the damage from fire grounds. (Using existing capability from ass 3 req 4)