Design Rationale FIT2099 – A1

Giacomo Bonomi

Req 1.

Req 1 involved the implementation of the following classes:

- **MetalSheet**
- **Player**

And an additional package, Scraps, within the Items package to encapsulate the BigBolt and MetalSheet classes. This is due to the presence of the future variety of scraps with separate properties, special scrap.

Both the BigBolt and MetalSheet class extend Item. This is so they may access the getPickUpAction() and getDropAction() as defined in the abstract class.

Now, we may simply pass in a 'True' Boolean into the item constructor, ensuring SRP is maintained; the handling of picking up and dropping isn't handled in these classes. Additionally, the extension of the Item forces us to constrain to the rules for items defined in our World.Java class in engine.

This ensures that all followable implemented scrap classes may follow this structure and be handled (picked up/dropped) without revisiting the code implementation.

Req 2.

Req 2 involved the implementation of the following classes:

In game.

In game.items.fruit:

- **LargeFruit**
- **SmallFruit**

In grounds.flora:

- **Tree (abstract)**
- **MatureInheritree**
- **SaplingInheritree**

In game.items:

- **Consumables (interface)**
- **SpecialScrap(interface)**
- **Tradeable**

In game.actions:

- **ConsumeHealAction**

The fruits and consumables-

Both Large and Small fruit implement item. As said above, this permits the handling of these classes to be handled outside of this class. Additionally, this allows us to make use of the Item Abstract Class's healingValue method, further increasing SRP princicple by the routing of this handling logic.

For the option to be consumed to be considered, the @Override for both Fruit's allowableActions() method calls our newly made ConsumeHealAction class.

Consume heal action extends action for it's menuDescription and execute Action. The new execute method allows it to handle for any type of consumable as long as it is also an extension of Item.

The Trees-

I have encapsulated the Tree abstract Class and Mature/Sapling inheritree for greater readability in addition to their largely different implementation and functionality to other Ground-extending classes.

The Tree abstract class extends ground, being able to implement a variety of separate trees.

This allows for the establishment of common functionality and the forcing of our specific implementation for future tree classes through polymorphism. Both classes have relevant probability checks for spawning a fruit in their tick method, as well as a produceFruit() method to generate the new fruit and addFruitToMap() to add it. This separation has been made in case future trees may possible spawn different or more than one fruits.

Sapling inheritree matures after 5 ticks, simply creating a MatureInheriTree() where it originally was, using the setGround() method, replacing it.

Req 3.

Req 3 involved the implementation of the following classes:

In ground:

- **SpaceshipDoor**
- **HunstmanSpiderCrater**
- **Spawner (abstract)**
- **Spawnable (interface)**

In behaviours

- **Aggressive Behaviour**

In Actors

- **HunstmanSpider**
- **Players**

In Capabilities

- **CAN_OPEN_SPACESHIP_DOOR**

The Spider-

To prevent the entering of spider in the future, I implemented a SpaceShip door class that extends Ground.

This class overwrites the canActorEnter method in Ground, instead returning a Boolean if the actor has the required CAN_OPEN_DOOR enum to open this spaceship door, so far only given to the player.
I initially implemented this method into the Floor class that is inside the spaceship until later realising this would present issues with spider behaviour if there would ever be Floor outside of the spaceship.

To implement the attack of the spider, the creation of the AgressiveBehaviour() class was needed.

Then, the behaviour for aggression was place before wander behaviour in the constructor. Now, the spider will initially perform this Aggressive behaviour at the beginning of each 'tick'. Should this be impossible, the play turn method has been updated to handle the null error. The AggressiveBehaviour class performs a check for all surrounding locations, checking if nearby actors possess the HOSTILE_TO_ENEMY enum, and if so, performing an AttackAction. This aggressive behaviour allows for not only the maintenance of the SRP principle for HunstmanSpider, but allows for future implementations of monsters that attack in a 1 'tile' range to also utilise this class with no issues.

The Spawner-

Instead of simply making Crater spawn spiders, I have made it so that Spawner is an abstract class that future monster specific craters can implement. This allows for the addition of new craters for different monster types to be added, each being represented by its own unique crater. This follows Liskov's substitution principle.

Additionally, I have implemented a spawnable interface that returns wether an actor is spawnable, in case a crater should be expected to randomly spawn monsters. This would allow for it to simply draw upon a list of possible actors without the screening for eligibility. Additionally, it would allow for classes that may spawn from different sources to also implement this interface. This better follows DIP, as the specific logic for spawning is not handled in the abstraction rather simply being used as a guide.

Req 4.

Req 4 involved the implementation of the following classes:

- All Items.specialscrap Classes
- Game.actors.player

Special Scraps

Both fruits and Metal Pipe implement or extend the SpecialScrap Interface. This interface defines a method, hasTradeableProperty, that calls to check if the fruit has the IS_TRADEABLE enum created in the Tradeable enum in special scrap.
An obvious concern here is the presence of the Tradeable enum n the first place; it is assumed that by definition all special scrap are tradeable. However, in the case non-special scrap may be possibly traded (which does not appear entirely unfeasible), I believe this implementation allows the most flexibility and modification for these possible exemptions, as future classes may simply check for this capability upon attempting to be traded rather than having to specifically implement the Tradeable interface, following the Interface Segregation Principle.

The creation of this class helps with mainting the SRP principle, as well as the Open/Closed Principle; in this way future capabilities may be added to SpecialScrap for non-tradeable items without the alteration of the interface.

MetalPipe and Intrinsic Weapon

In order for metal pipe to be recognised as a potential action option for attacking, it must be in the actions list. This is done by initiating a new attack action in the override for WeaponItem's allowableActions in its class. This makes it so that it returns the list of actions available for another actor near it, and if that involves attacking, creating a new attack action using this weapon and adding it to the actions list.

I believe this method to be more optimal as it follows the open/closed principle; not modifying pre-existing code and simply extending it for it's functionality. I opted not to have a WeaponAttack abstract class as there might be specific requisited for future weapons that may be inconsistent with the large majority of others, for example different ranges or different attack conditions.

To implement Player's new intrinsic weapon specifications, the getInstrincWeapon override is simply hardcoded to these new values.