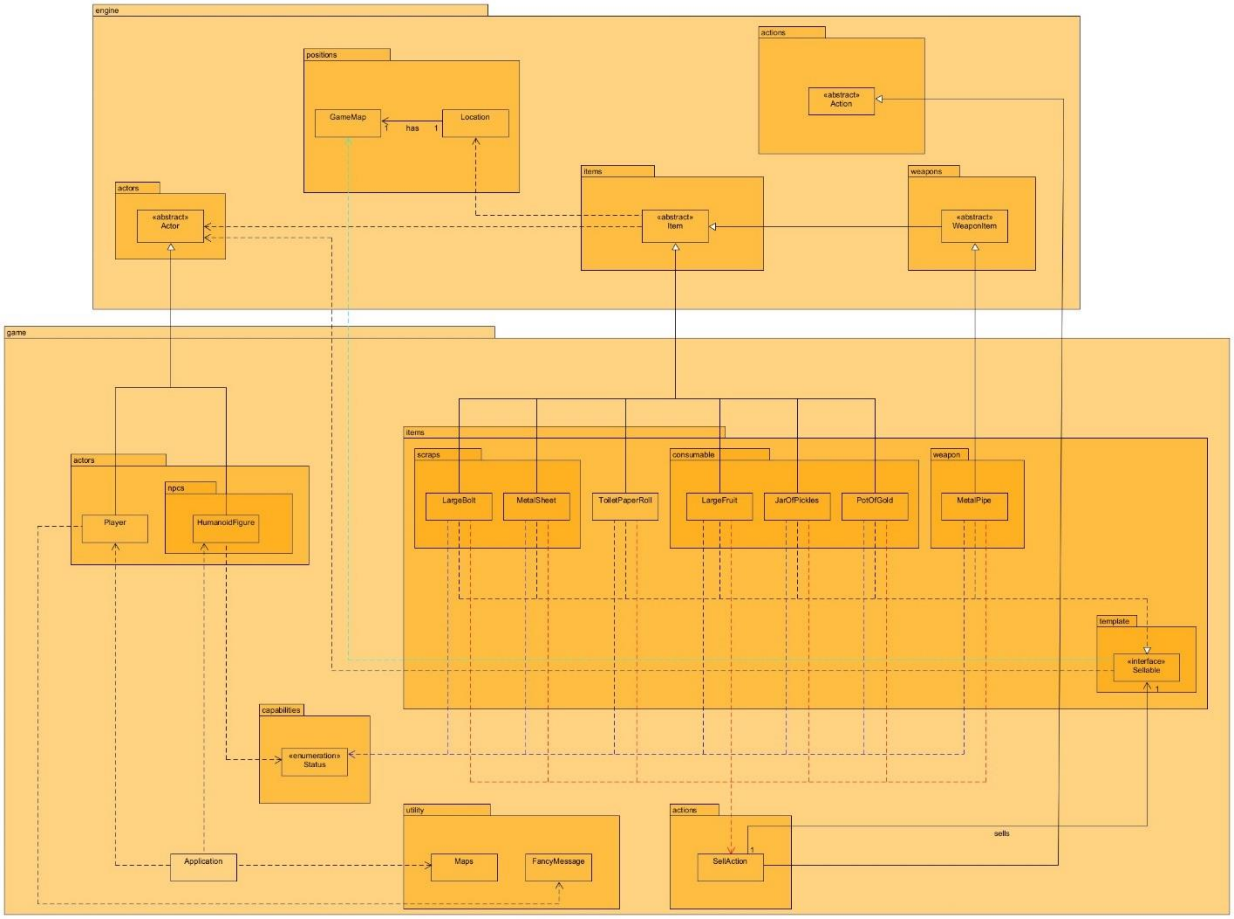
Assignment 3 Design Rationale

Requirement 2: The Static Factory

UML Diagram:



Design Goal:
The design goal for this assignment is to add the new class HumanoidFigure which represents an actor located at the spaceship's parking lot that the player can sells scraps to with different potential outcomes while adhering closely to relevant design principles and best practices.

Design Decision:
In implementing the HumanoidFigure class, the decision was made to have HumanoidFigure extend the abstract Actor class. This adheres to the DRY principle as by extending the Actor class, HumanoidFigure can use methods from the Actor class instead of rewriting it inside the HumanoidFigure, reducing code duplication. This also adheres to the Liskov Substitution Principle as the HumanoidFigure can be used whenever an instance of Actor is expected.

In order for items to be sold, a Sellable interface and SellAction class was created. The Sellable interface is implemented by all classes that are expected to be sold by the player. The Sellable interface contains a single method, sellItem.
The SellAction class extends the abstract Action class for the same reasons that HumanoidFigure extends the abstract Actor class. The SellAction class uses methods that the abstract Action class provides. By extending the Action class, SellAction can use its methods without duplicating it, adhering to the DRY principle. The SellAction class has an instance of Sellable as an attribute, and calls the Sellable's sellItem method within SellAction's execute method.

The advantage of having a Sellable interface that is implemented by all classes to be sold is that it helps our design adhere to the Open-Closed Principle. The SellAction class works with any class that implements the Sellable interface and hence has the sellItem method. This makes it easy to add new classes that can be sold by simply having them implement Sellable, without the need to modify the SellAction class. The Sellable interface also helps adhere to the Liskov Substitution principle, as any object of a class that implements Sellable could be used in SellAction. The Sellable interface only contains the single method sellItem, which means any classes that implement Sellable are not forced to have any other methods that it does not use, adhering to the Interface Segregation Principle. As opposed to having a single class that handles all selling logic, the Sellable interface helps our design adhere to the Single Responsibility Principle by having each Sellable handle its own selling logic in the sellItem method. By having the Sellable interface, our design adheres to the Dependency Inversion Principle as SellAction now depends on abstractions instead of concrete Sellable classes. This removes the coupling between SellAction and concrete Sellable classes, and makes the

design more flexible as SellAction can work with any new class that implements Sellable, making it easier to add new classes representing objects that are sold.

An alternative design would have been to have a “catch-all” God class that handled the selling logic of all items that could be sold. This class would have the Large Class code smell and violate the Single Responsibility Principle as this class would have to handle the selling logic of all items that can be sold. A God class would also violate the Open-Closed Principle as it would have to be modified whenever a new class that could be sold was added.

One disadvantage of having the Sellable interface is that some Sellables, such as LargeBolt and LargeFruit have identical sellItem implementations. This is code duplication, and a violation of the DRY principle.

An alternative design was hence considered where Sellable was an abstract class that extended the Item class. This would have the benefit of reducing code duplication, adhering to the DRY principle, as Sellables that had standard selling logic such as LargeBolt and LargeFruit could use the abstract Sellable class’s sellItem method. Sellables with more complex selling logic such as ToiletPaperRoll could then override the abstract class’s method and provide their own unique implementation.

However, this design has the disadvantage of being less flexible. If Sellable was an abstract class that extended Item, items such as MetalPipe would not be able to extend it as MetalPipe already extends the WeaponItem abstract class and Java does not support multiple inheritance.

In order for the player to be given the option to sell Sellables, the allowableActions method of each Sellable is overwritten to add a SellAction for the respective Sellable. In order to ensure that the player can only sell with actors meant to be traders, a new capability called TRADER was added to the Status enum class. This capability is added to HumanoidFigure inside HumanoidFigure’s constructor. The allowableActions method of each Sellable only adds a SellAction if the actor adjacent to the player has the TRADER capability. The TRADER capability is also used in the allowableActions method of weaponItems such as MetalPipe and DragonSlayerSword, where if the actor adjacent to the player has the TRADER capability, the AttackAction is not added. This prevents traders such as HumanoidFigure from being hurt by the player. The TRADER capability improves flexibility and extensibility as it allows our design to easily add new trader actors when the game is extended. The new actor class would simply need to have the TRADER capability added within its constructor for the player to use SellAction on them and sell items to them. A disadvantage to this design choice is that the current state of

the game has HumanoidFigure as the only trader. If there are no new traders to be added when the game is extended, the TRADER capability might be considered unnecessary and overengineering, which is indicative of the Speculative Generality code smell.
When it came to implementing the selling logic in the sellItem method of each Sellable, the classes LargeBolt, LargeFruit and MetalPipe had standard selling logic. Objects of the three classes could be sold for a specific price, with their sell price being added to the player's balance when sold. The classes MetalSheet, JarOfPickles, PotsOfGold, ToiletPaperRoll had more unique selling logic where given a certain special sell chance, an instance of the class could be sold for a discount (MetalSheet), double the price (JarOfPickles), taken without paying (PotOfGold) or have the player killed using the Actor class's unconscious method (ToiletPaperRoll). Instead of having magic numbers representing values such as the sell price within the sellItem methods of the Sellable classes, which would have Connascence of Meaning and Connascence of Value, constant variables were created. Each Sellable class contains the constant variables SELL_PRICE, and where applicable, SPECIAL_SELL_PRICE and SPECIAL_SELL_CHANCE. This converts Connascence of Meaning and Connascence of Value to the less strong Connascence of Name. Having less connascence in our design makes it easier to extend and reduces the chances of bugs.

Conclusion:
Overall, our chosen design provides a robust framework for implementing the HumanoidFigure and the player's ability to sell certain scraps to it. By carefully considering relevant factors, such as design principles, requirements, and the constraints of the game engine, we have developed a solution that is efficient, maintainable and extensible paving the way for future enhancements and extensions.