Requirement 2: The Static Factory

Design Goal:
The design goal for this assignment is to introduce the HumanoidFigure in the parking lot map and allow the player to sell a list of items to the factory with various outcomes. These design goals are done while adhering closely to relevant design principles and best practices.

Design Decision:
In implementing the HumanoidFigure class, the decision was made to extend from the abstract actor class in the engine and giving the figure the CAN_BUY_SELLABLE capability. This prevents repetition since the HumanoidFigure shares methods and attributes with the Actor class. This also reduces the time to debug since the bugs will lie in the HumanoidFigure class and not the abstract Actor class.

In implementing the list of items to sell, the decision was made for the sellable items to implement the Sellable interface. The sell method for the ToiletPaperRoll has a chance to return the unconscious method of the actor selling the item. To prevent connascence of meaning, magic numbers such as chances and selling prices are extracted as constants in the class with the name of the constant, to change it to connascence of name. Connascence of value is avoided by not passing in the selling price into the SellAction, and letting the sellable use its own selling price, since there will then be 2 values that are semantically linked.

The SellAction class extends the abstract Action class and is introduced to sell the sellable items. This reduces repetition since SellAction has similar functionality to the abstract Action class. It makes debugging easier since the Sellable interface has empty methods. It also makes it easier to extend since SellAction will not have to be modified when a new Sellable is added.

Alternative Design:
One alternative approach could involve a SellableItem abstract class extending from the abstract Item class. However, this will result in multiple level inheritance. This will also result in weapons not being able to extend from SellableItem, since weapons already extend from WeaponItem. This will also cause multiple dependencies between items to sell and SellAction

Analysis of Alternative Design:

The alternative design is not ideal because it violates various Design and SOLID principles:

1. **Open-Closed Principle:**
   - There will be multiple dependencies between SellAction and sellable items. When adding a new sellable, a dependency is needed from SellAction to the sellable and SellAction will have to be modified.

2. **Liskov Susbstitution Principle:**

   - Some methods of SellableItem will not be implemented if SellableItem inherits from WeaponItem, since not all sellable items are weapons. This makes polymorphism less robust and effective since the some of the methods are not implemented and different from the parent.

3. **Single Responsibility Principle:**
   - Multiple level inheritance causes less effective encapsulation, and the responsibility is shared between multiple classes, which are SellableItem and Item, making debugging more difficult and resulting in less effective encapsulation.

Final Design:
In our design, we closely adhere to DRY, Single Responsibility Principle, Open-Closed Principle, Liskov Substitution Principle and Dependency Inversion Principle:

1. **DRY:**
   - **Application:** SellAction inherits from the abstract Action class. Humanoid Figure extends abstract Actor class.
   - **Why:** The classes have common functionality since they have similar attributes and methods. Duplicated code is a code smell.
   - **Pros/Cons:** Best to abstract their identities to prevent repetition.

2. **Single Responsibility Principle:**
   - **Application:** Each Sellable handles its own selling method.
   - **Why:** instead of having a god class to handle Sellables, the classes are split into its own individual class. This will allow the Sellables to be responsible for a single part of its functionality, such as the outcome of selling the item.

Extracting the god class into an individual class addresses the code smell of divergent change, where the class is repeatedly changed.

- **Pros/Cons:** This makes debugging easier since the bugs will lie inside the class that implements sellable and not the Sellable interface since the methods are empty. This will debugging easier since the issue will be in the implemented classes. The class also does not have to be repeatedly changed, so it is easier to maintain. However, the selling methods may be similar between Sellables so it may not follow the DRY principle. An abstract SellableItem class can be used to solve this, but it is not done since the alternative violates several principles stated above.

3. **Open-Closed Principle:**
   - **Application:** Sellables implements Sellable interface
   - **Why:** SellAction does not have multiple associations with concrete Sellable classes
   - **Pros/Cons:** Ease of extensibility new Sellable, since it works for any class that implements it without modifying the SellAction class

4. **Liskov Substitution Principle:**
   - **Application:** Sellable items, WeaponItems and SellAction preserve the meaning of the methods from Item, WeaponItem and Action respectively.
   - **Why:** Ensures that polymorphism is robust.
   - **Pros/Cons:** Can be replaced with another class that implements Item and Action respectively without interfering with the behaviour of the program. Prevents switch statement with type conditions through polymorphism, which is a procedural programming code smell.

5. **Dependency Inversion Principle:**
   - **Application:** Sellables abstracted by implementing the Sellable interface
   - **Why:** Allow SellAction to depend on abstractions rather than concrete Sellable classes
   - **Pros/Cons:** This will make it simple to add an additional Sellable since SellAction does not need to depend on another concrete Sellable class, and Sellables can inherit from different parent classes.

Conclusion:

Overall, our chosen design provides a robust framework for achieving the assignment objective. By carefully considering relevant factors, such as design principles, requirements and constraints, we have developed a solution that is scalable and maintainable, paving the way for future enhancements and extensions.