Requirement 1: The Ship of Theseus

Design Goal:
The design goal for this assignment is to introduce the THESEUS, a portable teleporter that allows the intern to instantly move within a single map for free and can be purchased for 100 credits in the computer terminal. The other design goal is to add 2 new maps, which are the parking lot map and the new moon map and allowing the intern to travel between maps with the computer terminal. These design goals are done while adhering closely to relevant design principles and best practices.

Design Decision:
In implementing TravelAction class, the decision was made to extend from the abstract Action class in the engine. The MoveActorAction class from the engine was also used to implement the travelling between maps with the ComputerTerminal and within the same map with THESEUS. The rationale behind this is to prevent repetition by abstracting the identity with the abstract Action class and reusing the MoveActorAction class. This is also done to make debugging easier, since the problems will lie in the TravelAction class instead of the MoveActorAction class. This also makes adding a new Action easier, since we don't have to keep track of every Action class.

In implementing the THESEUS class, the decision was made to extend from the abstract item class in the engine and implement the purchasable interface introduced in the previous assignment. To prevent connascence of meaning, magic numbers such as default price are extracted as constants in the class with the name of the constant, to change it to connascence of name. Long methods which are a code smell, such as finding a location to teleport to, is extracted out as a separate method.

This reduces repetition since THESEUS shares similar attributes and methods to item. Furthermore, it also makes it easier to debug since the bugs will lie in the THESEUS class instead of the parent class. In addition, the PurchaseAction class will not need to depend on additional Purchasable concrete classes, making it easy to extend new purchasable items.

Alternative Design:
One alternative approach could involve extending from the MoveActorAction class. However, this will result in multiple level inheritance. This will also result in connascence of value, since the location to travel to will have to be both stored in MoveActorAction and TravelAction. The action methods of MoveActorAction will also have to be overridden.

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

1. **Do not Repeat Yourself (DRY) Principle:**
    - The same location for the actor to travel to are stored in both MoveActorAction and TravelAction, so this will cause repetition.

2. **Liskov Susbstitution Principle:**
    - All methods of the MoveActorAction class have been overridden by the TravelAction since they have different result and menu descriptions, so this makes polymorphism less robust and effective since the meaning of the methods are now different compared to the parent.

3. **Single Responsibility Principle:**
    - Multiple level inheritance causes less effective encapsulation, and the responsibility is shared between multiple classes, 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:** TravelAction inherits from the abstract Action class and reuses the MoveActorAction class and THESEUS inherits from the abstract Item class and reusing the generateRandomInt method from the RandomUtil class.
    - **Why:** Duplicated code is a code smell. The actions share common methods and the logic behind moving between locations is also very similar to the method in the MoveActorAction class. Similarly, THESEUS shares similar attributes and methods to Item, and the findTeleportLocation method to find a random location by picking random integers have similar logic.
    - **Pros/Cons:** Best to abstract their identities and reuse existing classes and methods to prevent repetition (DRY). However, there will be a dependency to the MoveActorAction class so it may not adhere to Reduce Dependency Principle (ReD).

2. **Single Responsibility Principle:**
    - **Application:** TravelAction and THESEUS class are individual classes
    - **Why:** instead of having a god class to handle everything, the classes are split into its own individual class. This will allow the classes to be responsible for a single part of its functionality. By extracting into an individual class, the class does not have to be repeatedly changed, preventing the code smell of divergent change.
    - **Pros/Cons:** This makes debugging easier since the bugs will lie inside the TravelAction class and not the Action class since the methods are implemented inside TravelAction. This also makes it easier to maintain, since the class does not have to be changed repeatedly. This will make adding additional items easier since there is no need to keep track of every item. However, this may lead to several similar attributes and methods to MoveActorAction, which may not abide with the DRY principle. The alternative would be to extend from MoveActorAction but it is not done since the alternative violates several principles stated above.

3. **Open-Closed Principle:**
    - **Application:** THESEUS implements Purchasable interface
    - **Why:** PurchaseAction does not have multiple associations with concrete Purchasable classes
    - **Pros/Cons:** Easy to extend new purchasables without modifying the PurchaseAction class.

4. **Liskov Substitution Principle:**
    - **Application:** THESEUS and TravelAction preserve the meaning of the methods from Item 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. The use of polymorphism instead of switch statements with type conditions avoids the procedural programming code smell.

5. **Dependency Inversion Principle:**
    - **Application:** THESEUS abstracted by implementing the Purchasable interface
    - **Why:** Allow PurchaseAction to depend on abstractions rather than concrete Purchasable classes

- **Pros/Cons:** This will make it simple to add an additional Purchasable since PurchaseAction does not need to depend on another concrete Purchasable class, and the Purchasable 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.