Assignment 3 Design Rationale
Requirement 4: Refactorio, Connascence's largest moon

Design Goal:
The design goal for this assignment is to model the various growth stages of an InheriTree class, with differing behaviours for growing or dropping fruits at each stage whilst simultaneously adhering closely to relevant design principles and best practices.

Design Decision:
In implementing the various stages of the Tree's growth (Sprout -> Sapling - > Young -> Mature), the decision was made to use two interfaces, one handling plant growth and the other with dropping fruit onto the map. Additionally, a new class called Utility was made that is able to return a valid exit given a location.

This decision was made as this design promotes loose coupling and high cohesion, as well as possessing a high degree of future modification/extension. This is done via:

- **New Utility File:** Trees that need to produce fruit can now utilize this new valid exit method. This avoids code redundancy where every fruit spawning tree would have to implement this identical code, but avoids forcing trees that don't generate fruits to implement this code in the same way an abstract class might.

- **Interfaces:** Using two separate interfaces 'Growable' and 'ProducesFruit' allows for classes adhering to these interfaces to implement relevant methods without forcing them to implement functionality they do not need, for example a tree that doesn't grow fruit being forced to return null for a produceFruit method they might not need. This helps reduce any forms of Connascence and makes the logic for adding fruit available to other future classes not implementing Tree (e.g. Shrub).

-

Alternative Design:
One alternative approach could involve utlising one singular tree class that uses flags/conditions to emulate the different stages and their behaviours.

For example:

```
public class Tree {
        private int stage; (0: Sprout, 1: Sapling, etc.)
        private boolean producesFruit;
        //variables for counter and period for each stage

        public void tick(Location location) {
                //for fruit generation
                if (stage == 0 && producesFruit){
                        //logic here
                }
                if (stage == 1 && producesFruit){
                        //logic here
                }
                // and so on

                //for changing stage
                if counter >= period for relevant stage
                stage ++
}
```

However, this tree class would be forced to handle various responsibilities about when to spawn fruits, it would be hard to implement additional behaviours or add/remove stages without heavy modifications to the current code, and multiple flags and conditions would reduce the code readability.

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

1. **Open/Closed Principle**
   Violation: Adding new stages for a tree, or a new behaviour for a new tree stage would require a large modification of the existing classes. This violates the idea of code being open to extension but closed to modification.

   Connascence of Position: High as additional stages/behaviours would affect the position of existing methods and logic.

2. **Single Responsibility Principle**

   Violation: Each class would have multiple responsibilities

   Connascence of Type: High since any change in the method signatures or data types would require changes across the entire class hierarchy.

3. **Interface Segregation Principle (ISP):**

   Violation: Fails to use interfaces to segregate different behaviours. In this example trees that don't drop trees are forced to have conditional checks if they do regardless.

   Connascence of Meaning: High, as the meaning of the classes would be overloaded with multiple, unrelated behaviours.

Final Design:

In our design, we closely adhere to [relevant design principles or best practices, e.g., SOLID, DRY, etc.

1. **Single Responsibility Principle**
   - **Application:** Each tree class is now responsible for its own singular stage of growth. It does not have to concern itself with the logic of the trees that came before or after it.
   - **Why:** This simplifies the maintenance and debugging of the class. Additionally, this reduces the connascence of Meaning by ensuring that each class has a single and clear responsibility.
   - **Pros/Cons:** Debugging, maintenance and overall connascence all improved. Might result in a greater number of classes.

2. **Open/Closed Principle**

   - **Application:** New stages of trees can be added without having to modify the previous stages or the interafaces.
   - **Why:** This makes the code highly extensible and scalable for future applications.
   - **Pros/Cons:** Allows for easy additions of different stages of tree types.

3. **Interface Segregation Principle**

   - **Application:** The interfaces Growable and ProducesFruit are separated in order to be responsible for different methods.
   - **Why:** This allows trees that don't need to implement methods (a tree that doesn't make fruit) to not have to implement irrelevant methods.
   - **Pros/Cons:** Clearly defined and separated interfaces and reduces Connascence of Type, requires multiple interfaces.
   -

Conclusion:

Overall, our chosen design provides a robust framework for representing a dynamically changing/growing inheritree class. By carefully considering the requirements for a Ground element, the separation of a valid exits method into the Utility class and taking into consideration avoidance of code redundancy or forced implementation, we have developed a solution that is easy to debug and maintain as well as being open to extend upon.