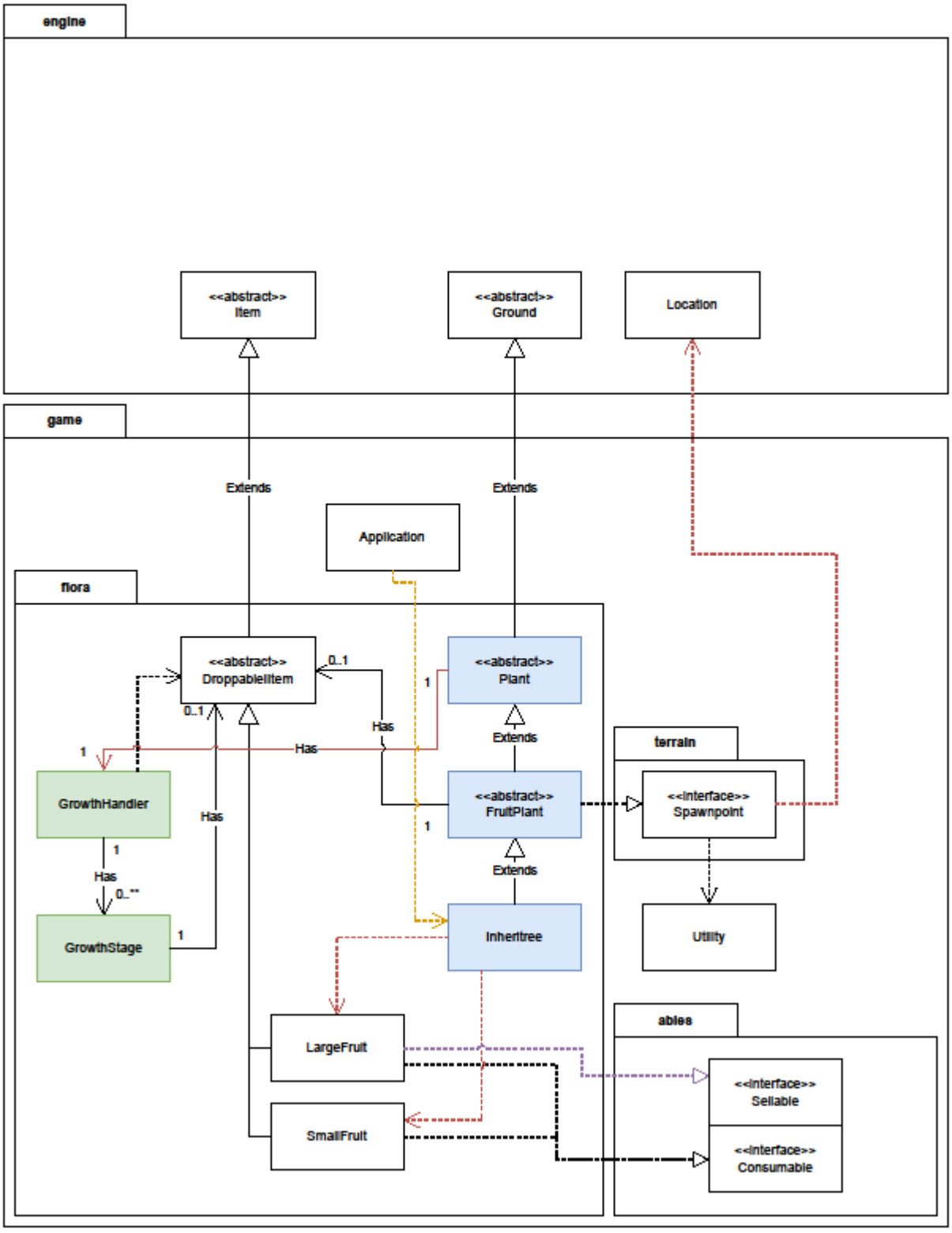
Assignment 3 Design Rationale

**Requirement 4: Refactorio, Connascence's largest moon**

**UML Diagram:**



**Design Goal:**
The design goal for this requirement is to refactor classes relating to plants to support variable growth stages while adhering closely to relevant design principles and best practices.

**Design Decision:**
In implementing this requirement with the reduction of connascences in mind, the decision was made to create two new classes. One to represent a growth stage and the other to handle the respective growth stages. For classes that represents a plant, we have two. Plant is to represent a plant that does not bear fruit (which is a DroppableItem) and FruitPlant is for a plant that does. Now each plant class has an attribute containing the class that handles growth. Each action to grow or add new growth stages is done through this class. The class that manages the growth stages is called GrowthHandler. To create a new growth stage, on a specific plant's side we simply call addGrowthStage that calls GrowthHandler's GrowthHandler's method that instantiates a GrowthStage object with the details of the duration, displayed character, and the fruit (if any) and adds it to the list of GrowthStages. Each time a plant class calls its tick method, the GrowthHandler will increment the tick count for a particular stage and return a Boolean based on whether the stage has changed. If the stage has changed, the plant class will update its attributes to ones returned by GrowthHandler. The reasoning behind this is to ensure that each class only has a single responsibility while being isolated from change from one another which leads it to being easy to maintain. Say you want to add more growth stages to a particular plant, just call addGrowthstage with the details of the new growth stage which makes it scalable. Lastly, in the design there is a Connascence of Execution here as the growth stages are dependent on the order the each GrowthStage is added.

**Alternative Design:**
One alternative approach could involve having each plant class handle their own growth and growth stages. However, say you want to change the way plants grow; you would have to modify each plant class accordingly which negatively impacts maintainability. Furthermore, the code would increase in complexity which leads to the classes being more complicated to add more growth stages. An example would be like below (note that the example below **only consists of 2 growth stages**):

```java
/**
 * The constructor of the Inheritree class.
 */
1 usage  Zhi Hao, Tong
public Inheritree() { super( name: "Inheritree",  displayChar: 't',new SmallFruit()); }

/**
 * Tries to spawn fruit and increment age every tick. If the age is 5, the displayed character becomes 'T'.
 * The type of fruit spawned becomes LargeFruit
 *z
 * @param location The location of the Ground
 */
 Zhi Hao, Tong *
@Override
public void tick(Location location) {
    System.out.println(this.getAge());
    if (this.getAge() == 5){
        this.setDisplayChar('T');
        this.setFruit(new LargeFruit());
    }

    this.spawnObject(location);
    this.incrementAge();

}
```

**Analysis of Alternative Design:**
The alternative design is not ideal compared to the suggested one because it violates various Design and SOLID principles:

1. **Single Responsibility Principle:**
    - The implementation above will result in a GOD class with too many responsibilities. This will be harder to maintain and add features to as more things need to be changed the larger it is scaled.

**Final Design:**
In our design for the implementation of this requirement, we closely adhere to DRY, and SOLID principles:

1. **DRY:**
    - **Application:** By having the GrowthHandler class evaluate the elapsed ticks for a GrowthStage and shifting to the next GrowthStage once its past the elapsed tick limit of the current one.
    - **Why:** This is to reduce repetitions of code such as if/else statements to check the tick limit. This property is evident when the number of GrowthStages is scaled to a larger number.
    - **Pros/Cons:** The pros are that it makes it easier to add a new GrowthStage without modifying much if of the code as we do that by adding an additional line in the plant which we want to add GrowthStages to. The cons are that new members of a team might have a tough time initially to figure out the inner workings of how adding GrowthStages work which means that there will be a bit of a learning curve.
2. **Single Responsibility Principle:**
    - **Application:** Having one class to represent the plant, another to handle growth stages, and another to represent growth stages
    - **Why:** This makes the code easier to maintain and understand due to the responsibilities being split into different classes. Now with reduced complexity, it is easier to look for the class and/or method that is associated with what a future developer would want to change.
    - **Pros/Cons:** Pros is that code is easier to maintain, understand, and modify due to the decreased complexity of each class. The cons are that adhering to this principle might put us at risk of over-abstraction. Therefore, there must be a balance between the number of responsibilities to split to adhere to the SRP and practical considerations.
3. **Liskov's Substitution Principle:**
    - **Application:** We can add different types of fruits for each GrowthStage as long as the "fruit" is either null or a subclass of DroppableItem.
    - **Why:** This provides flexibility to our system regarding the choice of fruit. This allows for polymorphism where each fruit might have a unique ability.
    - **Pros/Cons:** The pros are that this grants us the ability to have polymorphism when it comes to the fruit. This also makes our code more robust as we are able to handle all types of DroppableItems. The cons are that code might have increased complexity due to the need to carefully consider the implementation of each subclass as the project gets scaled.

4. **Dependency Inversion Principle:**
    - **Application:** We can add different types of fruits for each GrowthStage as long as the “fruit” is either null or a subclass of DroppableItem.
    - **Why:** Much like the principle above, this provides flexibility to our system regarding the choice of fruit. This allows for polymorphism where each fruit might have a unique ability.
    - **Pros/Cons:** Much like the principle above, the pros are that this grants us the ability to have polymorphism when it comes to the fruit. This also makes our code more robust as we are able to handle all types of DroppableItems. The cons are that code might have increased complexity due to the need to carefully consider the implementation of each subclass as the project gets scaled.

**Conclusion:**

Overall, our chosen design provides a robust framework for achieving the objectives of this specific requirement. By carefully considering the SOLID and DRY design principles along with the amount of connascence in the implementation, we have developed a solution that is efficient, scalable, and maintainable, paving the way for future enhancements, extensions, and optimizations.