REQ4

I have decided to use the Strategy design pattern to implement the trees and their behaviours. The Strategy pattern allows us to define a family of algorithms (behaviours), encapsulate each one, and make them interchangeable. This pattern is suitable for the trees as they can have multiple unique behaviours. This is done by having a PlantBehaviour interface and a Tree abstract class. The PlantBehaviour interface defines the contract for tree behaviours, with its concrete implementations stored within a TreeMap in the Tree subclasses by passing them in from the Application class. By using the Strategy pattern, we can easily add new behaviours to a tree without modifying the existing code. Additionally, the ItemSpawner interface follows the Factory Method design pattern, allowing us to create different types of items that can be created without having to modify any existing code.

Single Responsibility Principle (SRP):
Each class has a single responsibility. For example, the Tree class is responsible for managing the tree's behaviours, while the GrowBehaviour and DropBehaviour classes define specific behaviours of the tree. This makes the code easier to maintain and extend.

Open-Closed Principle (OCP):
This system is open for extension but closed for modification. This is evident in the PlantBehaviour interface, ItemSpawner interface, and the Tree class. New behaviours can be created by simply implementing the PlantBehaviour interface, which can then be passed into the Tree subclasses without modifying the existing code. Similarly, new item spawners can be created by implementing the ItemSpawner interface. Finally, new tree types can be added by inheriting the Tree class. This promotes code reusability and extensibility.

Liskov Substitution Principle (LSP):
Since the subclasses of the Tree class inherit from the Tree class, they can be used interchangeably with the Tree class. For example, a YoungInheritree can be used wherever a Tree is expected. This principle ensures that the code is flexible and extensible.

Interface Segregation Principle (ISP):
The ItemSpawner and PlantBehaviour interfaces were segregated into smaller, specific interfaces. This allows classes to implement only the methods they need, rather than implementing a large, general interface. This makes the code more maintainable and easier to understand.

Dependency Inversion Principle (DIP):
ItemSpawner instances are injected into DropBehaviour, decoupling the behavior from the specific item spawner implementation, while Tree instances are injected into GrowBehaviour, decoupling the growth behavior from the specific tree type. This promotes code maintainability and testability.

Connascence of Algorithm:
The tree subclasses doesn't need to know the specific algorithms used by the PlantBehaviour instances. The algorithm is encapsulated within the PlantBehaviour implementations, reducing the connascence of algorithm. Changes within the algorithm of a specific behaviour will not affect the tree subclasses.

Connascence of Position:
The use of a TreeMap in the Tree class to store behaviours reduces connascence of position. The order of behaviours is determined by their priority, not their position in the code.

God Class:
By breaking down the functionality into smaller, more manageable classes, the complexity of a potential God Class has been reduced. Each class now has a single, clear responsibility, adhering to the Single Responsibility Principle (SRP).

Feature Envy:
Without following the Strategy pattern, the Tree class would have to know the specific algorithms for each behaviour, leading to feature envy. By encapsulating the behaviours within their respective classes, the Tree class is no longer envious of the behaviours, reducing the risk of feature envy.

Shotgun Surgery:
By encapsulating related behaviours within their respective classes, changes to a particular behavior or functionality would be localized to that class, reducing the risk of Shotgun Surgery.

While this design has many benefits, there are a few potential drawbacks. Firstly, the use of multiple design patterns can make the codebase more complex and harder to understand for developers unfamiliar with these patterns. Secondly, if there are only a couple behaviours and they rarely change, there's no real reason to overcomplicate the codebase with new classes and interfaces that come along with the Strategy design pattern. Finally, the codebase might grow significantly as new behaviours, trees, and item spawners are added because they all need their own classes, making the system more complex to manage.