4/7/2020

Feature Implementation

Project Deliverable 4

Andrew Yue-Keung Leung, Brian Ning Yu Chim, David Fernandes, Patrick Angelo Del Rosario Ocampo, Sameed Sohani

LES PROGRAMMEURS

Table of Contents

Revamped Plant System – User Guide

As per deliverable 3, we have opted to revamp the code that deals with how plants behave in Minecraft. With the changes brought forth from our redesign, modders now have the abilities to:

- Define their own custom plant types with ease and not worry about any potential issues (such as null pointer exceptions).
- Define their own custom fertilizers (apart from Bone Meal) so that modders can extend the features of the plant mods they wish to create

Defining a Custom Plant Block

To define a custom Forge growable plant block, it must first implement the IForgeGrowable interface. With respect to the methods, there are three which must be implemented: **canGrow, canUseFertilizer, and grow**. **canGrow** is used to check whether a plant can grow in its current location; **canUseFertilizer** is used to check whether a plant can use the given fertilizer in its current location; **grow** is used to define how the plant will "grow" into the next stage.

Defining a Custom Plant Type

Upon defining the custom Forge plant block, it must now be given a type that determines which environments it can grow on. Performing this action is just as easy as calling a single method. In order to define a custom plant type, all a modder needs to do is call a static method "get(String plantTypeName)" under the PlantType class and pass in their custom plant type accordingly. Below is an example code snippet of how a custom plant block with a custom plant type would look like:

```
public static class CustomPlantExample extends FlowerBlock implements IPlantable {

    public static PlantType pType = PlantType.get("custom");

    public CustomPlantExample() {
        super(Effects.WEAKNESS,  effectDuration: 9, Block.Properties.create(Material.PLANTS).doesNotBlockMovement().
        this.setRegistryName(MODID, CUSTOM_PLANT_BLOCK);
    }
}
```

Defining a Custom Fertilizer

We can now move on to define a custom fertilizer (if a modder wishes) for their custom plant(s). With the inclusion of fertilizers in Forge, modders now have the ability to create fertilizer-like items that could either have less or more strength than bone meal in the game. In this sub feature, we define a new element called **potency.** This refers to the strength of a given fertilizer and can be characterized based on how much of a given fertilizer is required to grow a

plant. For example, a custom fertilizer with a potency of 2 means that this fertilizer is twice as strong as regular bone meal (**note:** we have made bone meal as the baseline for a fertilizer's potency, i.e. bone meal has been given a default potency of 1 - more on this in the details pertaining to our design).

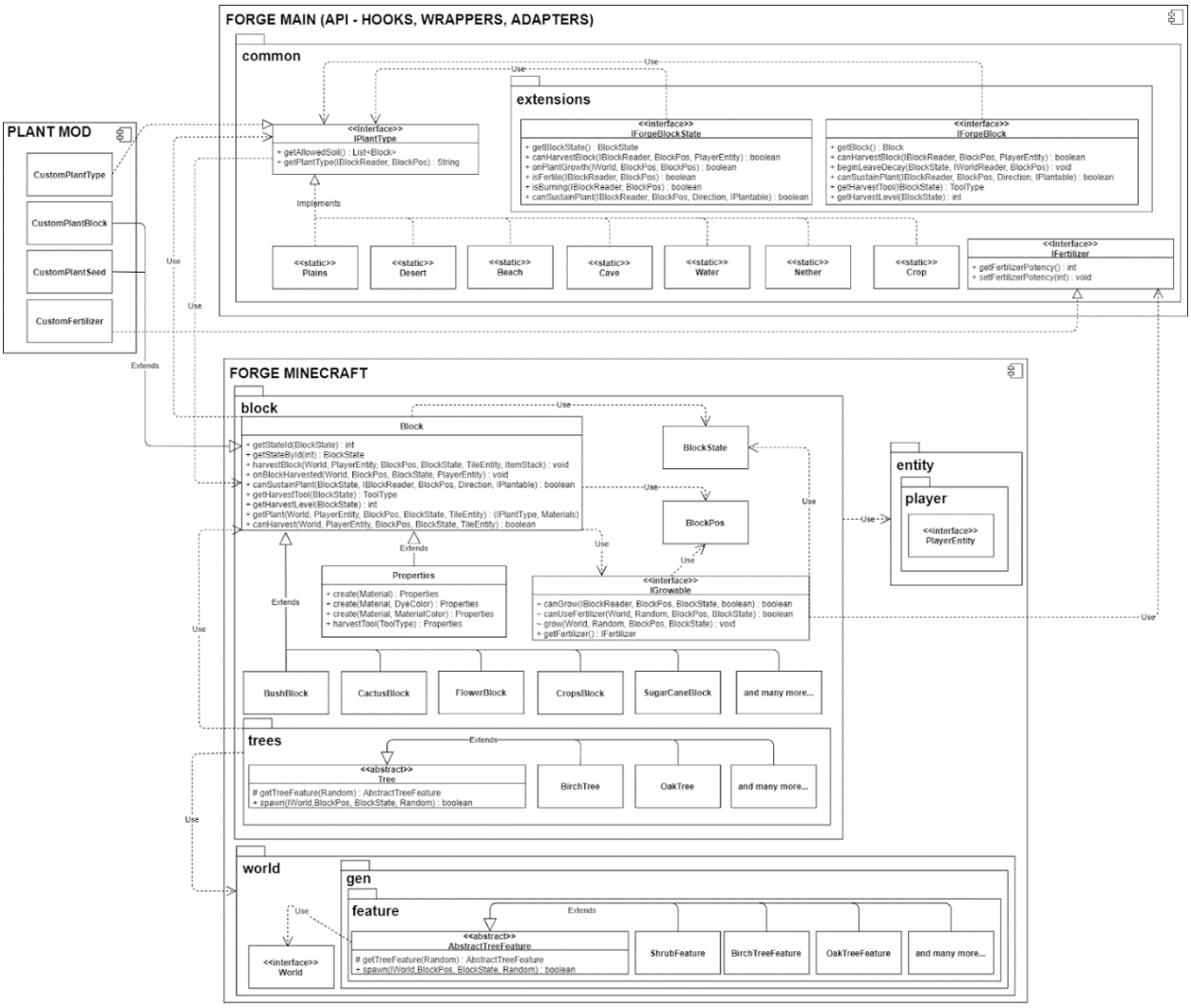
In order for a modder to create their own custom fertilizer, they must implement the IFertilizer interface we have established in the **Forge Main** component (located in **/src/main/java/net/minecraftforge/plants**). IFertilizer by default contains 2 methods that need to be implemented - getPotency and getValidPlantTypes. Setting a custom fertilizer's potency can be done by specifying the desired float in the return statement of getPotency. As for setting valid plant types, this is left to the modder's discretion when creating a custom fertilizer. To see an example of how exactly a custom fertilizer's class structure would look like, a modder can refer to the BoneMealItem (refer to **forge/src/main/java/net/minecraft/item/BoneMealItem.java** in the decompiled code, i.e. under the **projects** folder) class that we have redefined in order for it to also be recognized as a fertilizer.

Revamped Plant System – Design/Implementation Details

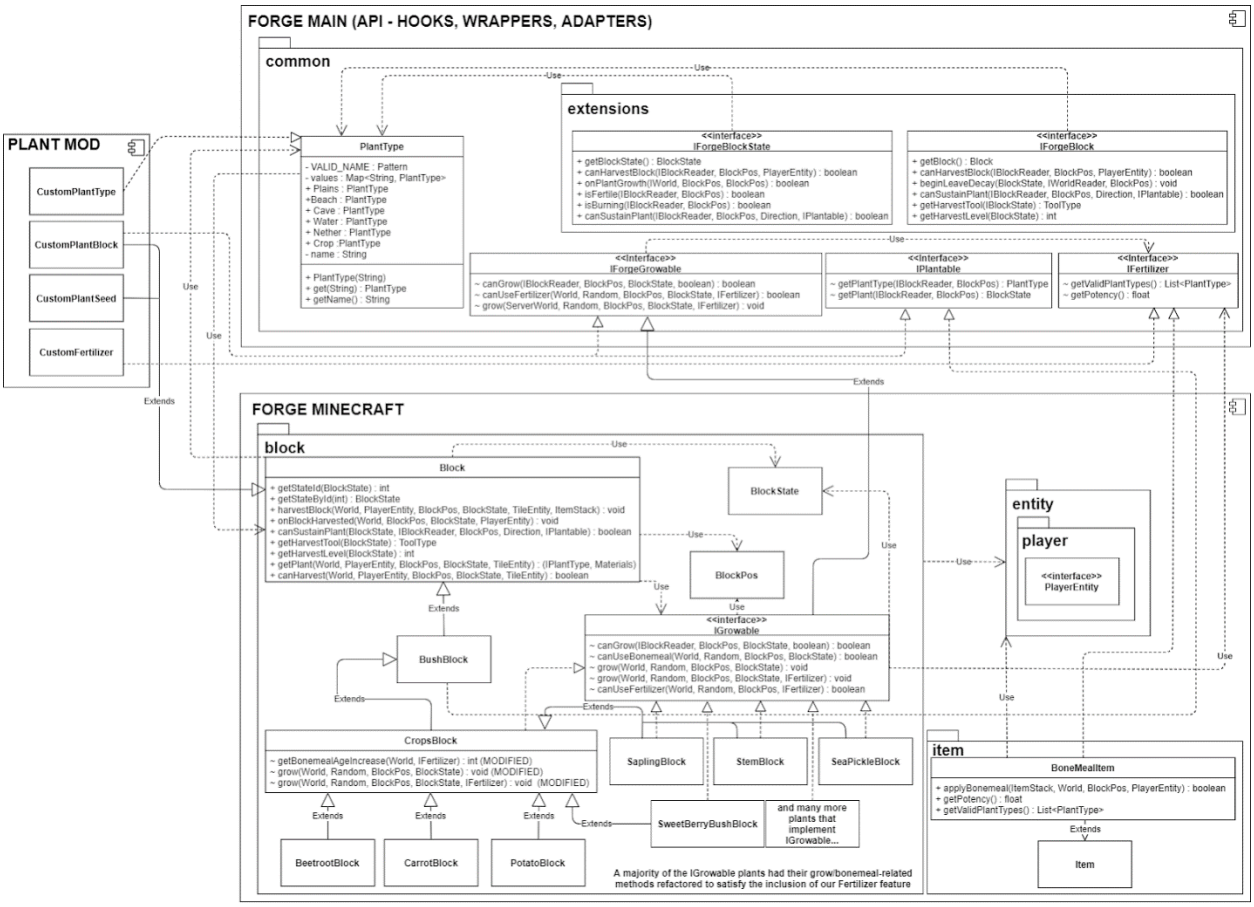
Issue / Relevant Links:

- https://github.com/MinecraftForge/MinecraftForge/issues/5371
  - Follow-up to our PR – https://github.com/MinecraftForge/MinecraftForge/pull/6573
- https://github.com/MinecraftForge/MinecraftForge/pull/5362

Changes to the Previous Design



*BEFORE (Deliverable 3) - Plant System Revamp (for full image, please refer to **plant-feature-revamp_uml_before.png**)*



*AFTER: Plant System Revamp (for full image, please refer to **plant-feature-revamp_uml.png**)*

Details on the design differences are discussed in further detail in **Custom Plant Types** and **Fertilizer**

Preface

As an aside to the implementation design, one thing to keep in mind with respect to our changes is that most are found within the patch files rather than directly to source code. These patch files are changes seen similar to a “git diff” file between the Minecraft Forge codebase and the vanilla Minecraft code. Due to the Forge team’s desire to maintain as small of a patch file[1] as possible with minimal line changes, you will often see our new methods written out with fully qualified names as well as on one line, in addition to sometimes insufficient spacing for clarity. Furthermore, due to the fact that we make our changes based on decompiled Minecraft code, there will be many “garbage” variables with names such as “p_22535_1_” which will remain and also be used. All our changes found in the patch files follow these guidelines to the best of our ability and have been done with the intention of supporting the Forge team’s requirements/style guide on contributing to the project.

[1]Recall from Deliverable 1 that any changes to the Minecraft code will require the generation of patch files. Refer to the commit history under the branch **1.15.x_plant-revamp_fertilizer-feature** and make note of .patch files

Custom Plant Types

With the previous implementation of plant types, modders were required to use the create() function inside of net.minecraftforge.common.PlantType in order to create new Plant Types. Unless the modder is familiar with the Forge codebase, they may be surprised and not understand why the method definition simply throws an exception. Un-intuitively designed, Forge uses ASM / Reflection in order to dynamically add new Plant Types which are declared in an enum class. With the goal of the Forge project being the allowance of modders to interact with and have freedom to modify and add game mechanics to suit their needs, it is beneficial for Plant Types to be intuitive such that modders can understand how to create new ones.

For the reason of clarity for modders and intuitive design, we thus modified the PlantType class such that it is no longer an enum but instead a regular final class with static variables and a HashMap which contains all the PlantTypes mapped to their canonical name. The primary method to be used in this class is the static **get**() method which returns the PlantType corresponding to the canonical name given as an argument. If this PlantType does not already exist, an entry is created in the HashMap and immediately returned. As a result of this change, a corresponding change was also needed to be made inside of the **canSustainPlant**() method found in net.minecraft.block.Block which previously had a switch statement based on the enum PlantTypes. With the change to the HashMap of PlantTypes, this has been switched based on the String equivalents instead.

As mentioned previously, the purpose of this redesign of the Plant Types is so that modders can understand at a glance how new Plant Types can be created and hence do not need to understand the inner workings of Minecraft Forge to understand how Plant Types work. In deliverable 3, we intended to replace the PlantType enum with an interface and consequently, all the existing PlantTypes would become classes and implement that interface. Rather than continue with that particular approach, we felt that simply placing all of the PlantTypes within a single class in a HashMap would be more appropriate. The reason for this is because we wanted the design to be similar to another class - net.minecraftforge.common.ToolType. ToolType is a class which solves a similar problem of providing predefined types of tools and allowing modders to add their own, similar to providing predefined types of plants and allowing modders to add their own.

Fertilizer

With respect to Fertilizer, this is a completely new feature our team has decided to introduce alongside the Forge Plant System Revamp. In the existing Minecraft / Forge codebase, there does not exist any ability for modders to accomplish tasks such as allowing blocks to be fertilized by different items, some of which may be working at different potencies (Note: As mentioned before, potency refers to a scalar number with respect to the effectiveness of a fertilizer given that bone meal is the baseline). This is primarily due to a design choice in the base game where “growable” plants (defined by blocks which implement the net.minecraft.block.IGrowable interface) only originally had methods such as canUseBonemeal()

(showing that bone meal was the only fertilizer) and **grow**(...), which did not have any argument where items which can be used to grow can be specified.

In terms of changes dealing with IFertilizer and ensuring that modders can now create new Fertilizers which can interact with different plants, many changes needed to be made to the Forge code. The primary change revolves around three classes: (new) net.minecraftforge.plants.IFertilizer, (new) net.minecraftforge.plants.IForgeGrowable, and (modified) net.minecraft.block.IGrowable.

IFertilizer is a new interface meant to be implemented by any items which will be considered "Fertilizers". In the vanilla Minecraft game, Bone Meal is currently the only item that is used as fertilizer (an item used to accelerate a plant's growth) and consequently, net.minecraft.item.BoneMealItem is the only class which now implements IFertilizer. All fertilizers should have a potency (represented as a float from positive real numbers) which reflects how effective the fertilizer is with respect to bone meal. As mentioned in the previous section, Bone Meal serves as our baseline with a potency = 1.0f. An important note on the potency is that it is directly related to how bone meal functions so as to not too heavily change base Minecraft game mechanics. For example, when bone meal is used on crops, it increases the crops age by a given factor whereas when used on tree saplings, bone meal provides a percentage chance to increase the tree sapling's growth stage. A fertilizer of potency 2 would hence provide double the effectiveness - it can increase the crops age by a given factor times 2, and it provides double the percentage chance to increase a tree sapling's growth stage. Further details on the effects on bone meal on a plant can be found on the wiki[2]. This potency value should be retrieved through the getPotency method. A difference in implementation from deliverable 3, we opted to not include the **setPotency**() method because we felt that it was inappropriate for this to be accessible to modders. Other mods should not have a simple way to overwrite potencies of fertilizer that other mods have introduced. Another method in IFertilizer is **getValidPlantTypes**, which returns a list of valid plant types on which the given fertilizer can be used.

IForgeGrowable is a new interface meant to be implemented by any new plants which can be grown. It provides similar functionality to the base vanilla game's IGrowable[3] interface, but also includes an extra argument in the **grow** method - an IFertilizer parameter that allows for modders to pass in their custom fertilizer. In addition, the **canUseBonemeal** has been renamed into **canUseFertilizer** for sake of intuition. **canUseFertilizer** (and **canUseBonemeal**) are boolean checks for if a plant can use a given fertilizer at a given instance.

[2] https://minecraft.gamepedia.com/Bone_Meal

[3] **canGrow** is used to check whether a plant can grow in its current location; **canUseFertilizer** is used to check whether a plant can use the given fertilizer in its current location; **grow** is used to grow the plant to its next stage

The base Minecraft's IGrowable interface has now been updated to extend the new IForgeGrowable interface. This was done so that base vanilla plants now have a wrapper around our IForgeGrowable interface. Recall that the goal of the Minecraft Forge project is to have minimal changes to the base Minecraft game. As a result, turning IGrowable into a wrapper for IForgeGrowable is more appropriate than simply replacing it completely with our new IForgeGrowable interface. IForgeGrowable was introduced rather than adding the functions into IGrowable because modders should only be able to add new growable plants if they define the fertilizers available to those plants, and also should not need to implement the now older methods which do not make reference to fertilizers. The default implementation for the IGrowable **grow** and **canUseFertilizer** functions is to simply call the previous implementation of the functions.

In terms of changes to the growable plants (those which implement net.minecraft.block.IGrowable), as a result of IGrowable now extending IForgeGrowable, it was necessary to delegate the work from the original methods which work based off of bone meal (either grow or canUseBonemeal in the case of SaplingBlock) to our new method which functions near exactly the same except adding the factor of fertilizer potency into the calculations.

One example provided could be inside of net.minecraft.block.StemBlock. The interaction that Bone Meal has with StemBlocks is that bone meal matures the planted stem's age by a random factor between 2 - 5. The corresponding method responsible for this interaction is **grow(ServerWorld, Random, BlockPos, BlockState).** In particular, the line responsible for calculating this age is now:

```
int i = Math.min(7, p_225535_4_.get(AGE)
+MathHelper.nextInt(p_225535_1_.rand,
      (int)fertilizer.getPotency() * 2, (int) fertilizer.getPotency() *
5));
```

Whereas the previous calculation simply returns a random number between 2 to 5, now we multiply those min and max numbers by a factor of the potency of fertilizer. Note that there is also the usage of Math.min to ensure that this calculation does not step out of bounds of the maximum age of a plant. (The minimum age is not of concern as users are supposed to use only positive real number for potency and a potency of 0 would simply cause the plant to not age). Similar changes were implemented in each of the growable plants found in Minecraft, with fertilizer interactions being based on bone meal interactions.

Previously, the process / logic of using fertilizer was the following:

1. Player attempts to use bone meal on the plant (BoneMealItem.applyBonemeal is called)
2. applyBonemeal checks whether the plant is growable
3. applyBonemeal checks whether the plant can use bone meal
4. applyBonemeal calls on the plant to grow with the appropriate logic
5. The plant grows

After our changes, the process / logic of using fertilizer has been changed to being the following (for the sake of continuity we will look at bone meal rather than a new fertilizer):

1. Player attempts to use some fertilizer on the plant (BoneMealItem.applyBonemeal is called)
2. applyBonemeal checks whether the plant is Forge Growable (has fertilizer methods)
3. applyBonemeal checks whether BoneMealItem (some fertilizer) can support the plant type of the corresponding plant
4. applyBonemeal checks whether the plant can use the corresponding fertilizer (bone meal)
5. applyBonemeal calls on the plant to grow with the given fertilizer
6. Fertilizer logic (potency) is applied
7. The plant grows

Acceptance Tests

Given the nature/current standards of the Minecraft Forge project, we did not create any unit tests for this project. As it was mentioned in previous deliverables, testing is done entirely with the creation of test mods, which serves as a customer acceptance test. This comes as a result of the fact that Minecraft players will be loading mods into their game with Forge. Ensuring that our test mods can be loaded into the game and call upon the modified components without any errors gives us the appropriate flags that we haven't broken anything. Through this way of testing, we are able to both verify and validate that our changes to the codebase are correct and yielding the appropriate results (i.e. seeing elements/objects/entities behave normally in Minecraft).

With regards to the acceptance testing, the same requirements still stand as per deliverable 3 - there are two sets of test suites that exist. The **regression suite** contains tests that ensure all of Minecraft's vanilla behaviour remains the same with our new changes. This will assure us that we have not broken anything in Forge's codebase. The **feature suite** contains tests to check that the new addition of specific system features work as they should with the Forge codebase**.** Overall, these test cases will have more of a focus towards testing the inclusion of our fertilizer feature and making sure both BoneMeal and any potential custom fertilizers interact correctly with in-game plants.

The following are regression (acceptance) test mods that a user can load into their game so that they can check that the vanilla game of Minecraft continues to work as expected:

1) **CropPlantTest** - Make an attempt to plant crop blocks and have them grow both naturally and through BoneMeal

- Refer to the following path → **/src/test/java/net/minecraftforge/debug/block/plants/CropPlantTest.java**
- The purpose of this test is to ensure that users are able to plant the vanilla game's crops as per usual and see that they grow into their appropriate vegetables. If the console does not log any exceptions as the user is trying to plant the given crops, then the test case is a pass.
- Test Details/Steps to Reproduce→
    a. For each crop plant in the game (Carrots, Wheat, Potatoes, Pumpkins, Melons, Beetroot, and Sugar cane), check that they can be grown appropriately with or without Bone Meal
        - The user adds a hoe, shovel, and a bucket of water into their inventory from the Creative inventory
        - The user finds a plot of land with grass and dirt, and digs out a 1 x 10 row of dirt
        - The user fills up the hole with water
        - The user fertilizes the dirt blocks adjacent to the water by equipping the hoe and right clicking said blocks
        - The user plants each of the seeds into the fertile land
            - Sugar cane would be an exception - this must be planted on a sand block adjacent to a water block
        - At this point, the user should expect to see fully grown crops.

2) **FlowerPlantTest** - Make an attempt to plant a select variety of flowers in their appropriate environment

- Refer to the following path → **/src/test/java/net/minecraftforge/debug/block/plants/FlowerPlantTest.java**
- The purpose of this test is to ensure that users are able to plant the vanilla game's plants as per usual and also see that they are planted in their correct environments (i.e., on dirt or grass blocks). If the console does not log any exceptions as the user is trying to place the plants on their respective blocks, then the test case is a pass.
- Test Details/Steps to Reproduce →
    a. For each of the Minecraft flowers, check that they can be planted on the respective correct soil
        - The user adds each type of flower into their inventory from the Creative inventory
        - The user places the appropriate flower on the appropriate land (i.e. on a grass / dirt block in an open plain)
        - The flower should plant with no problems

3) **TreePlantTest** - Make an attempt to plant a select variety of trees in their appropriate environment

- Refer to the following path → **/src/test/java/net/minecraftforge/debug/block/plants/TreePlantTest.java**
- The purpose of this test is to ensure that users are able to plant the vanilla game's saplings as per usual and ensure that they grow/flourish into their appropriate tree types in the correct environments (i.e. in a spot where there is grass/dirt and an enough growth space). If the console does not log any exceptions as the user is trying to place the plants on their respective blocks, then the test case is a pass.
- Test Details/Steps to Reproduce →
    a. For each of the Minecraft trees, check that they are able to be planted and can grow with bone meal.
        - The user adds a sapling for each tree type into their inventory (Birch, Oak, Spruce, Acacia, Jungle, and Dark Oak) from the Creative inventory
        - The user places the sapling on the appropriate land (i.e. on a grass/dirt block in an open plain)
        - The user adds bone meal into their inventory from the Creative inventory and equips it
        - The user proceeds to right click each of the planted saplings
        - At this point, the user should expect to see fully grown trees of each type

4) **OtherPlantTest/SeaPlantPlaceTest** - Make an attempt to plant other plants in the game in their appropriate environments

- Refer to the following path for planting Cactus → **/src/test/java/net/minecraftforge/debug/block/plants/OtherPlantTest.java**
- Refer to the following branch and path for planting SeaPickles/SeaGrass → **1.14.x_issue-6325_fix**
  /src/test/java/net/minecraftforge/debug/block/SeaPlantPlaceTest.java
- The purpose of this test is to ensure that users are able to plant the other kinds of plants available in the vanilla game (i.e., sea pickles, sea grass, cactus). As with the previous test cases, this is to also ensure that the plants are flourishing in their appropriate environments (underwater for sea plants, desert for cacti). If the console does not log any exceptions as the user is trying to place the plants on their respective blocks, then the test case is a pass.
- Test Details/Steps to Reproduce →
    a. For each of the other Minecraft plants, (Cactus, Sea Grass, Sea Pickle, Coral), check that they can be planted on the respective correct soil and that they grow properly
        - The user adds a cactus, seagrass, and sea pickle block into their inventory from the Creative inventory
            - The user will need to place the Cactus on some sand block in the open

- Seagrass will need to be placed on either dirt or sand underwater
- Sea Pickles can be placed anywhere
- Coral blocks can be placed anywhere

- At this point, these plant blocks should be present on the blocks they were placed in
  - For coral blocks → if they were placed on dry land, they should die in a few seconds and turn grey in colour; if they were placed underwater, they should retain their colour/vibrance
  - For sea pickles → an attempt should also be made to place multiple sea pickles on a single block → the light levels (i.e. brightness) should increase

*For this regression suite, we enforce the use of Bone Meal wherever applicable in order to ensure that our changes to the item (by making it a "Fertilizer" object and allowing for custom potencies) does not result in any different out-of-the-ordinary behaviour with growing plants in the game.

The following are feature (acceptance) test mods that a user can load into their game so that they can check that plant/fertilizer mods behave as they should. This is to ensure that modders are capable of creating their desired plant-based mods with our new additions.

1) **CustomPlantPlaceTest** - Make an attempt to plant a custom plant block on top of a vanilla plant block.

- Refer to the following path → **/src/test/java/net/minecraftforge/debug/block/plants/CustomPlantTest.java**
- The purpose of this test case is to ensure that custom plants can be created for the game and can also be planted on vanilla blocks in Minecraft. From deliverable 2, we had a similar test (**CustomPlantTypeTest.java**, located in branch **1.14.x_issue-6286_fix**) that made a check to ensure that custom plants can be planted on custom blocks.
- Test Details/Steps to Reproduce →
  - Create a mod that makes a custom plant of a custom plant type which can be planted on one vanilla block and one custom block.
    - Launch the game and enter "Creative Mode", i.e. sandbox mode
      1. The user adds the custom plant (labelled as "test_custom_plant" ingame), the vanilla block, and the custom block into their inventory.
      2. The user attempts to place their custom plant on the vanilla block and the custom block.
      3. At this point, the plant should place correctly.

2) **CustomFertilizerTest** - Make an attempt to use a custom fertilizer with twice the potency of bone meal on vanilla plants

- Refer to the following path → **/src/test/java/net/minecraftforge/debug/block/plants/CustomFertilizerTest.java**
- The purpose of this test case is to ensure that custom-made fertilizers can be used on vanilla plants on the game, given the inclusion of our new fertilizer feature.
- Test Details/Steps to Reproduce →
    - Create a mod that makes a custom fertilizer with two times potency (using one of this fertilizer should be equivalent of using two of a regular fertilizer like bone meal)
        - Launch the game and enter "Creative Mode", i.e., sandbox mode
            1. The user adds wheat seeds, the custom fertilizer (labelled as "test_custom_fert" ingame), dirt, and a hoe to their inventory.
            2. The user places the dirt on the ground and then right clicks it with the hoe, tilling it.
            3. The user plants the seeds and then right clicks on it with the fertilizer
            4. The process should be repeated until the user notices an instance where the fertilizer has caused the wheat to grow faster than just with bone meal (the user can also compare the behaviours of the custom fertilizer with regular bone meal)

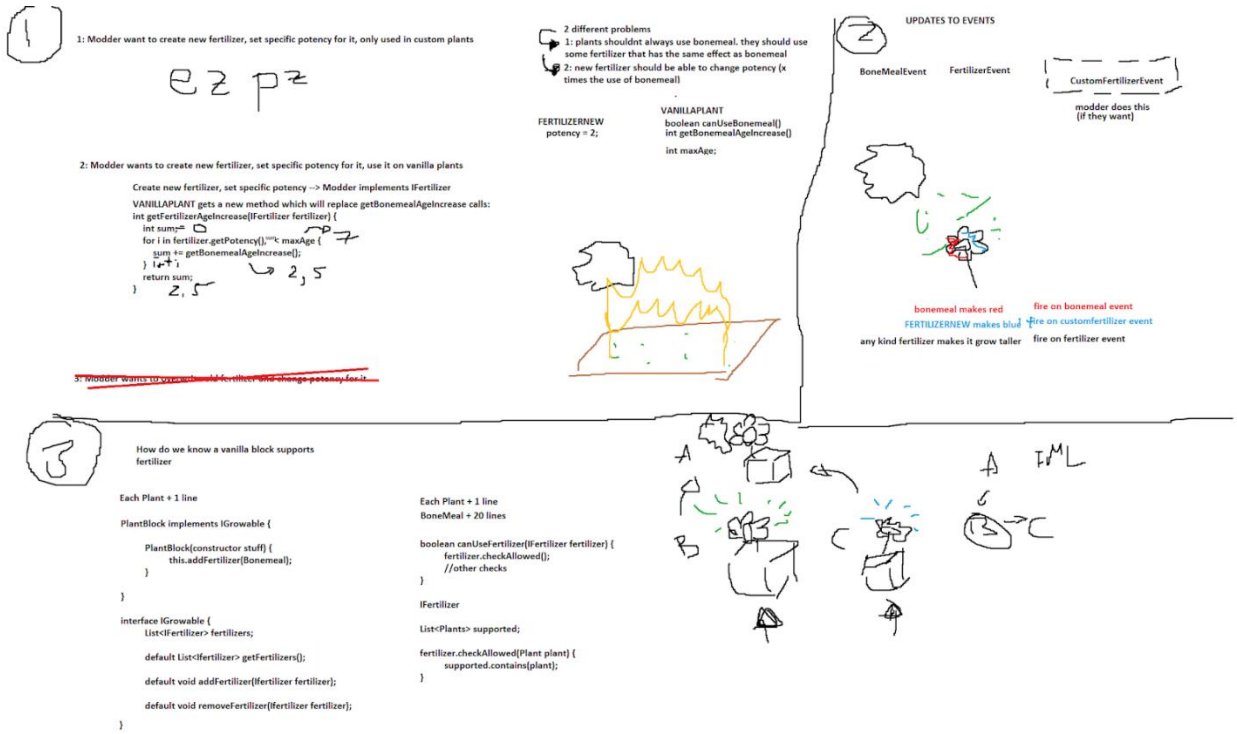
3) **NetherWartTest** - Make an attempt to use a custom fertilizer with potency equivalent to that of bone meal on Nether Warts

- Refer to the following path → **/src/test/java/net/minecraftforge/debug/block/plants/NetherWartTest.java**
- The purpose of this test case is to ensure that custom fertilizers can now be used on Nether Warts. Previously, the game did not allow for bone meal to be used on Nether Warts but with this plant revamp, this is now an option for modders.
- Test Details/Steps to Reproduce →
    - Create a mod that makes a custom fertilizer with potency equivalent to one bone meal (using one of this fertilizer should be equivalent of using two of a regular fertilizer like bone meal)
        - Launch the game and enter "Creative Mode", i.e. sandbox mode
            1. The user adds wheat seeds, the custom fertilizer (labelled as "test_custom_bone" ingame), soul sand, and a nether wart.
            2. The user places the soul sand anywhere on the ground
            3. The user plants the nether wart on top of the soul sand.

4. The user right clicks the nether wart with the custom fertilizer in their hand. The user should see the nether wart grow

Software Development Process

In the time that we've spent working on the latter half of this course project, we have continued to follow through with Kanban and using a board to visualize/track our progress on the deliverables. In addition, we have also **heavily** incorporated the practice of pair-programming for this particular deliverable. This came as a result from discussing how we would proceed to design/implement the plant system revamp we had in mind. The following is a graphic of the notes we had for a meeting:



*Plant System Revamp Meeting notes (for full image, please refer to **fertilizer-meeting-notes.png**)*

As we were planning out the design of our feature, we were writing code snippets and making comments/suggestions on each other's implementations. Given how our changes were going to result in the generation of many patch files and overall result in longer build/compile times over each of our machines, we decided to follow through with our own variation of pair programming. In this variant, one person would share their screen in a voice call while others would watch and make comments/suggestions. We thought that this would save us more time

since **one** person would be generating the patch files. If we split the development amongst all the members, more time would be spent trying to merge each person's changes and also generating patch files that accomodate for multiple people's changes. Overall, convenience and efficiency were our motivating factors for employing pair-programming-like practices.