2/26/2020

Work Planning

Project Deliverable 1

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

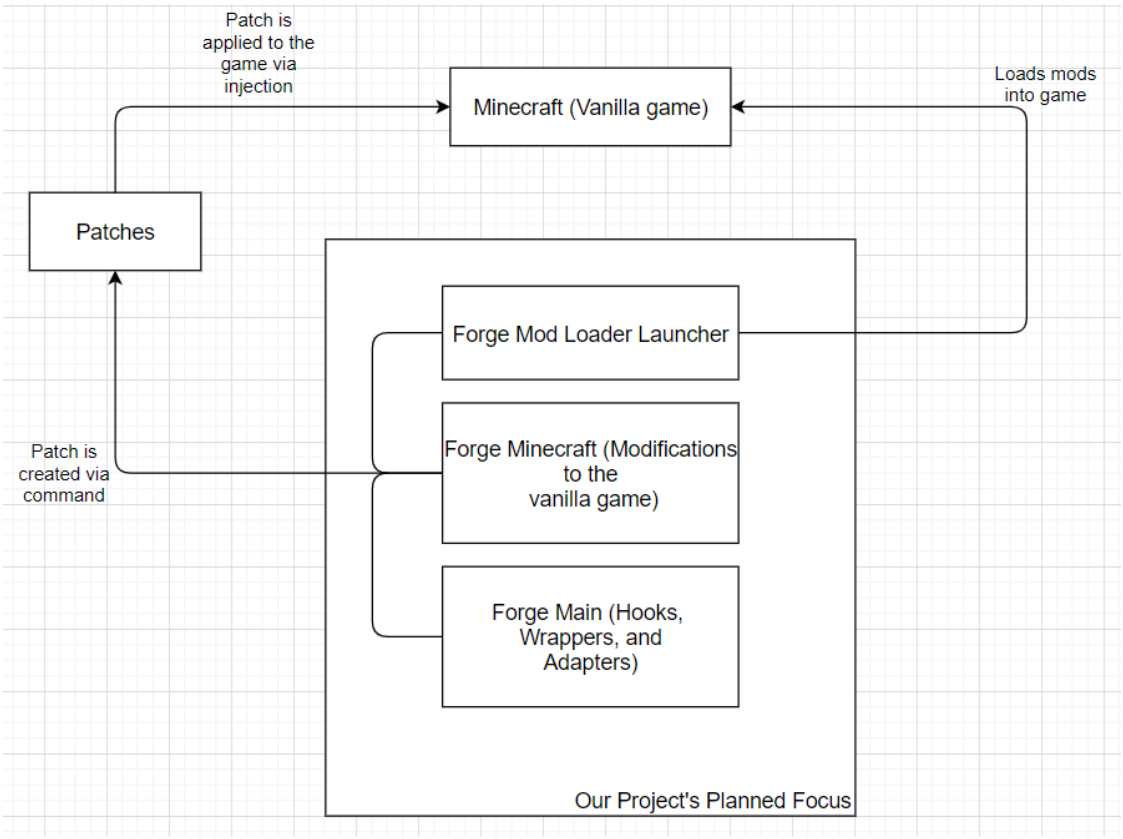
LES PROGRAMMEURS

Table of Contents

Architecture

Minecraft Forge (Forge) is an API allowing developers to make modifications (mods) to the base game of Minecraft. In simplest forms, the entirety of Forge as a project is an adapter pattern. It abstracts several game features and allows mods created by Minecraft modders to be cross compatible with each other as well as different versions of Minecraft. As an API, it also provides a platform for modders to use shared classes and cleanly implement similar features. Finally, Minecraft Forge is often used as a "testing ground" for Mojang, the developers of Minecraft, by way of implementing features which are then copied into the main game.

High-Level Architecture and Connectivity



In terms of architecture, the MinecraftForge project is primarily centered around five main components to work with the main Minecraft game. These components are located at the following:

Patches - root/patches
Minecraft (Vanilla game) - root/projects/clean
Forge (Modifications to the vanilla game) - root/projects/forge

Forge (Hooks, Wrappers, and Adapters) - root/src/main
Forge Mod Loader Launcher - root/src/fmllauncher

An important thing to note is that two of the five components, Minecraft (Vanilla game), and Forge (Modifications to the vanilla game), can be found in the code only after first building the source code.

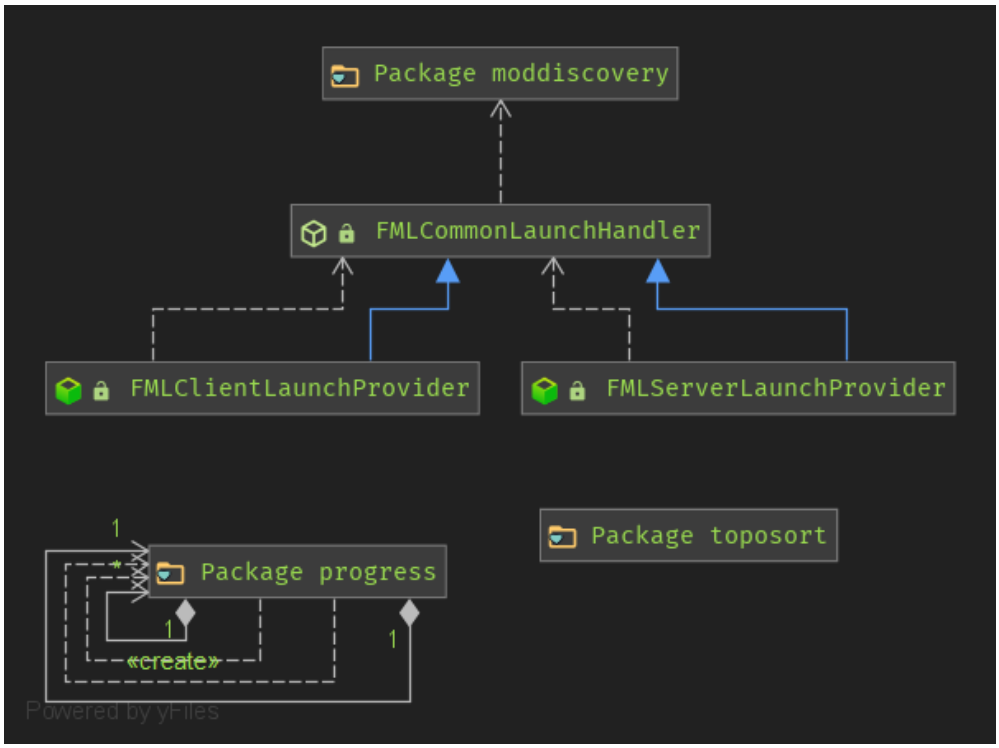
In terms of how Minecraft Forge interacts with the main Minecraft game, it does this through the concept of patches alongside the Forge Mod Loader Launcher (FMLL). These patches are generated from code differences between the Forge based code (discussed in the next paragraph) and the decompiled vanilla Minecraft code which is found in the root/projects/clean project. For the FMLL, this component is responsible for detecting installed Minecraft mods (of which Forge is considered one), and proceeds to load and inject the necessary code for the respective mod. As a result of this, when running Minecraft along with Forge, the FMLL injects the Forge code in place of the vanilla Minecraft code so that on startup of Minecraft, it will run with the modified Forge client and then load with the desired Minecraft mods.

With respect to the Forge code, this can be separated more finely into modifications in the base Minecraft game (found in root/projects/forge) and additional hooks, wrappers, and adapters (found in root/src/main) created by Forge developers to allow Minecraft modders developing with Forge to work with universal Minecraft Forge features (such as Energy networks) or non-exposed Vanilla Minecraft APIs, via the Minecraft Forge EventBus (an implementation of Google Guava's EventBus Observable/Observer).

Main Modules

As per the high-level diagram given above, our focus within this project will be directed towards the following ”sub-projects”: The **Forge Mod Loader Launcher**, **Forge Main**, and **Forge Minecraft**. We will be taking a quick high-level overview of each component and analyzing the architecture.

Forge Mod Loader Launcher – src/fmllauncher



Root-level overview of the FMLL component

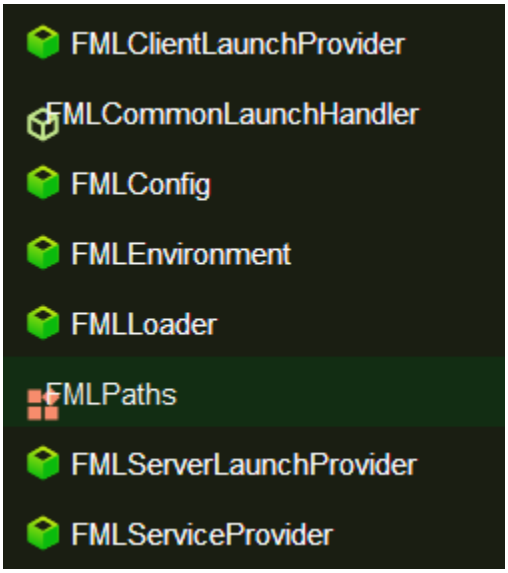
As one can tell from the name, the Forge Mod Loader Launcher is Forge’s launcher that is in charge of detecting all the mods present in a user’s directory and loading them into the game. The packages can be described as follows:

- **moddiscovery** → Detects all the mods to be loaded into the game. It traverses the directory that the users are supposed to place their mods in and parses / validates each JAR file to check if they can be loaded into the game.
- **progress** → Gives the client a visualization of the mod loading progress.

- **toposort** → Sorts the mods being loaded into the game so that everything is loaded in properly without any errors. This is because if mods are dependent on one another, it may break if one mod is loaded prior to a mod it is dependent on.

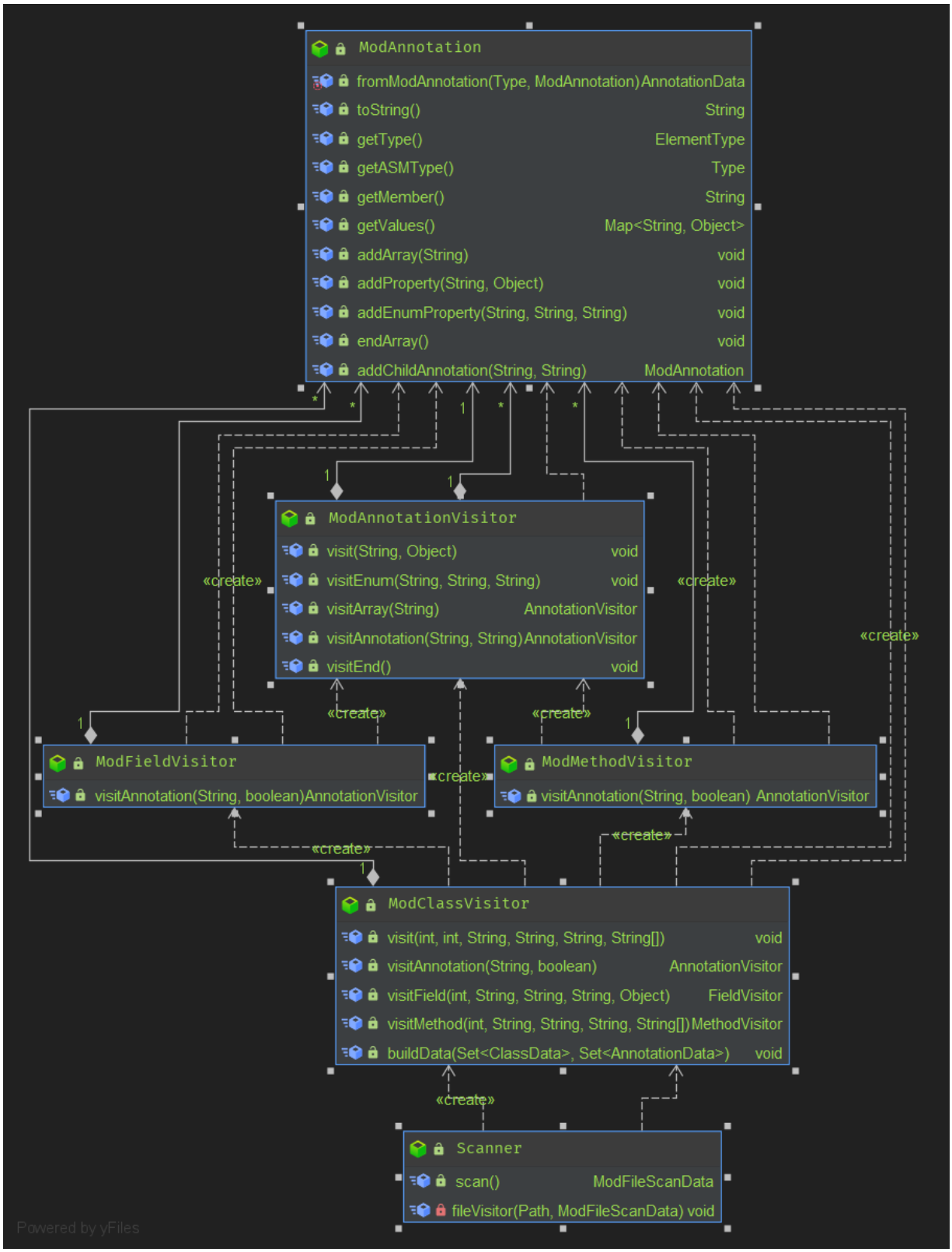
Due to the large number of files present in FMLL's root, many have been omitted from the diagram.

One thing to note about FMLL's project structure is the lack of packages used to organize all the files. As an improvement to the readability of the project, certain files can be grouped together. For example, the following files can be grouped into their own package:



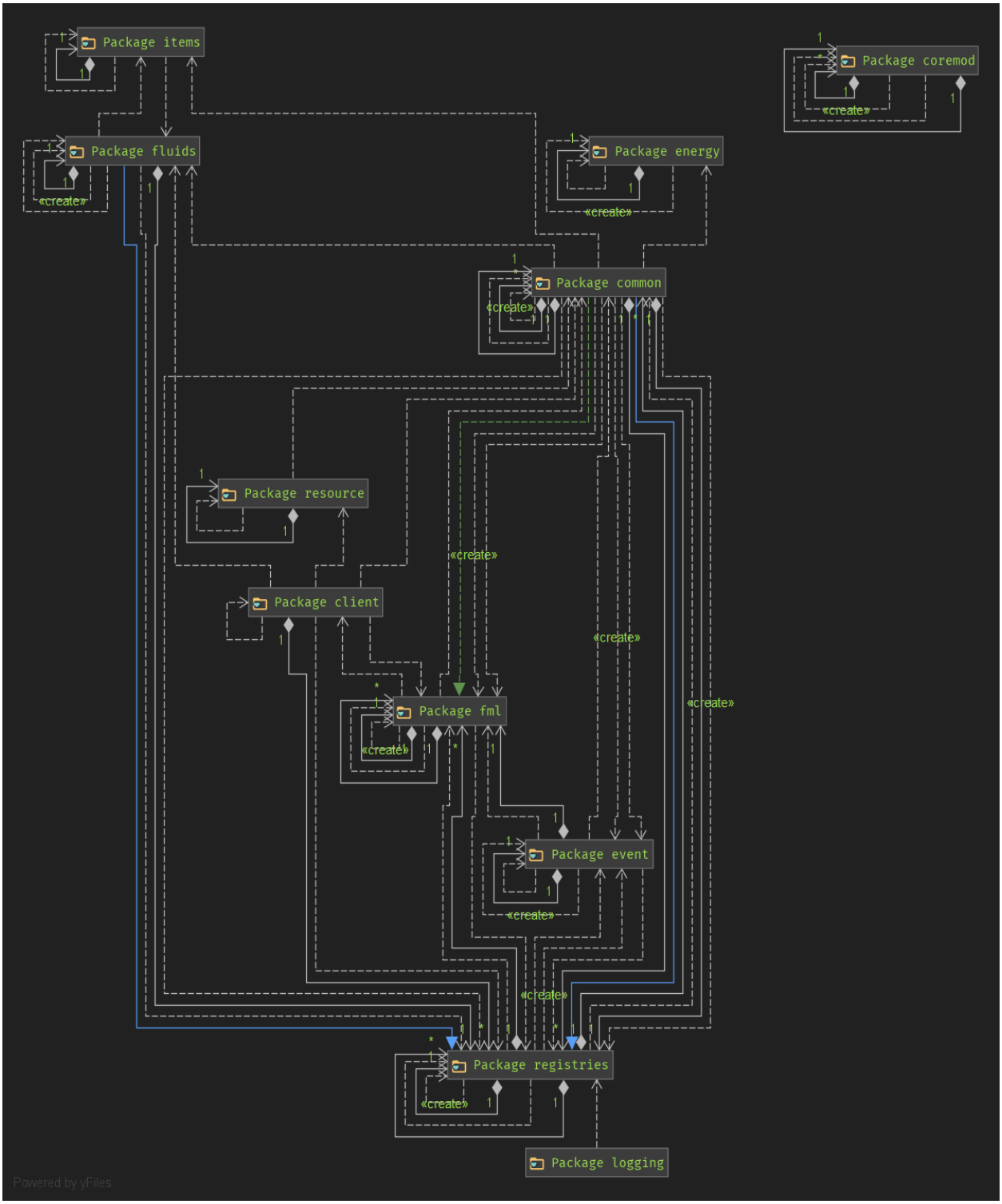
This would also apply to the other sub-projects mentioned as they have the same issue of having a lack of packaging and organizing with certain files.

In regards to notable patterns, the observer/observable and visitor patterns have been implemented with the mod annotation object. This allows for mods to be given the appropriate annotations during runtime and also keeps track of every mod being loaded **(see the next page).**



Mod Annotation following the Observer pattern in addition to the Visitor pattern

Forge Main (wrappers, adapters, and hooks) – src/main



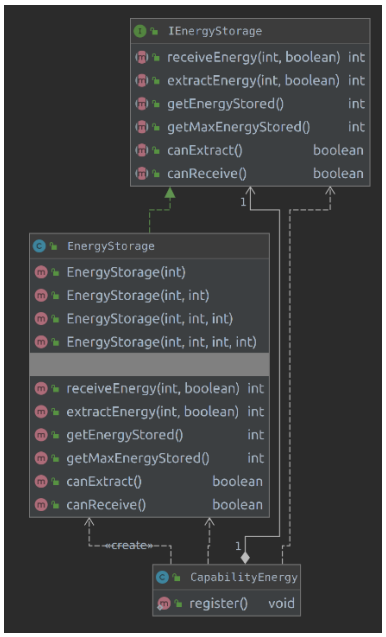
Root-level overview of the Forge Main component

Forge Main is essentially where all of Forge's main functionality resides. It contains all the code that allows for it to wrap around Minecraft and also inject the necessary changes to the game's code. The packages can be described as follows:

- **client** → Contains all the event handlers and GUI rendering for the client
- **common** → Contains interfaces and other in-game configurations for Minecraft
- **energy →** An API that enables modders to create their own energy implementation in the game
- **event** → contains events that modders often access from Minecraft (entity/block interactions, enchanting items, boss encounters, etc.)
- **fluids →** contains all the logic/configurations related to how custom fluids should work in the game (Minecraft itself doesn't implement fluid's generically, it hardcodes most values)
- **fml →** contains all of the mod loader's sided event handlers (server vs. client), hook management, and wrapper management

The rest of the packages follow a similar structure in regards to containing wrapper code, hooks, registry managers, and event handlers.

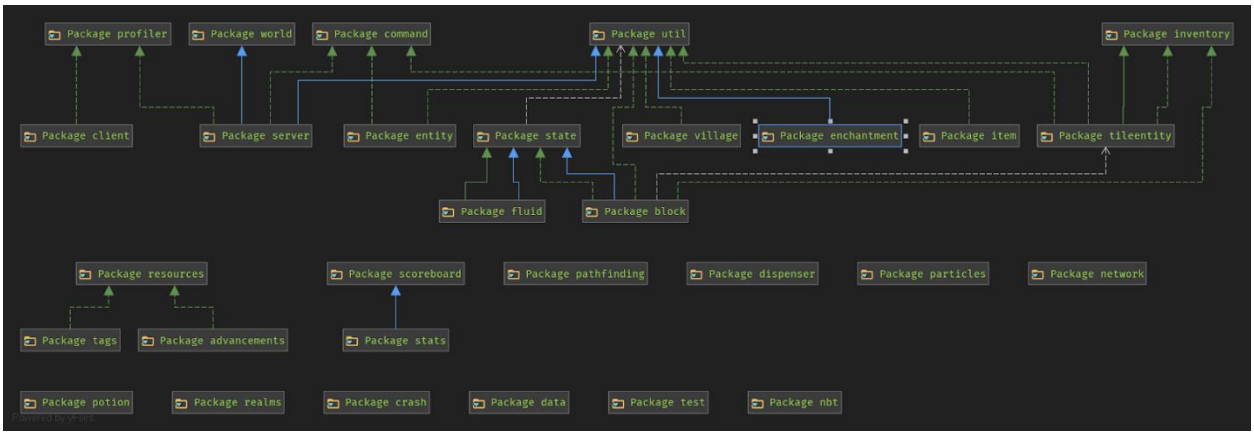
One thing to make note of in Forge Main's architecture is the interdependency of the packages mentioned. Given the size of Minecraft, there is much that must be managed when creating a wrapper that injects code into the base game. In addition, one interesting feature to make note of in this subproject is the energy API that was added to Forge in version 1.10.2:



Energy API class structure

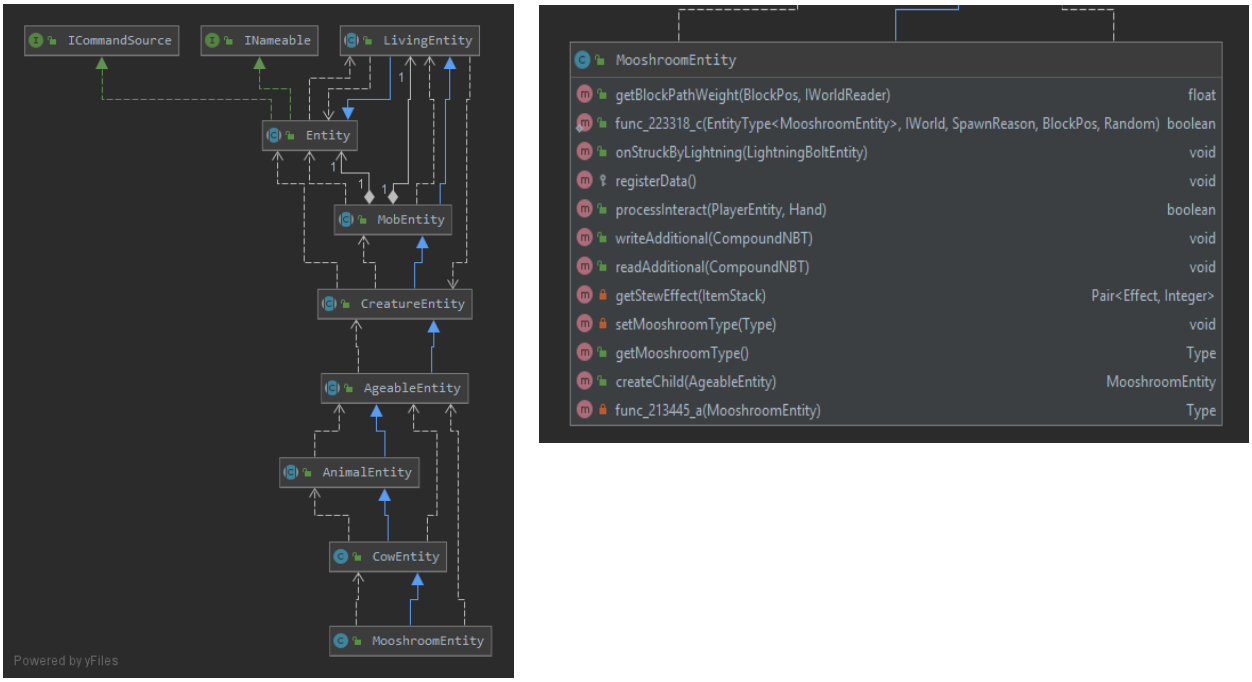
Forge's energy API is helpful for modders who wish to build an energy-like system and need some place to begin. The EnergyStorage class is a great starting point since it serves as a default implementation and may be used as is. Modders also have the option to customize by either extending this default implementation or simply just implementing the IEnergyStorage interface. This interface acts as the base energy implementation across all Minecraft mods since many popular mods build off of this.

Forge Minecraft (modifications to the Minecraft codebase) – projects/forge

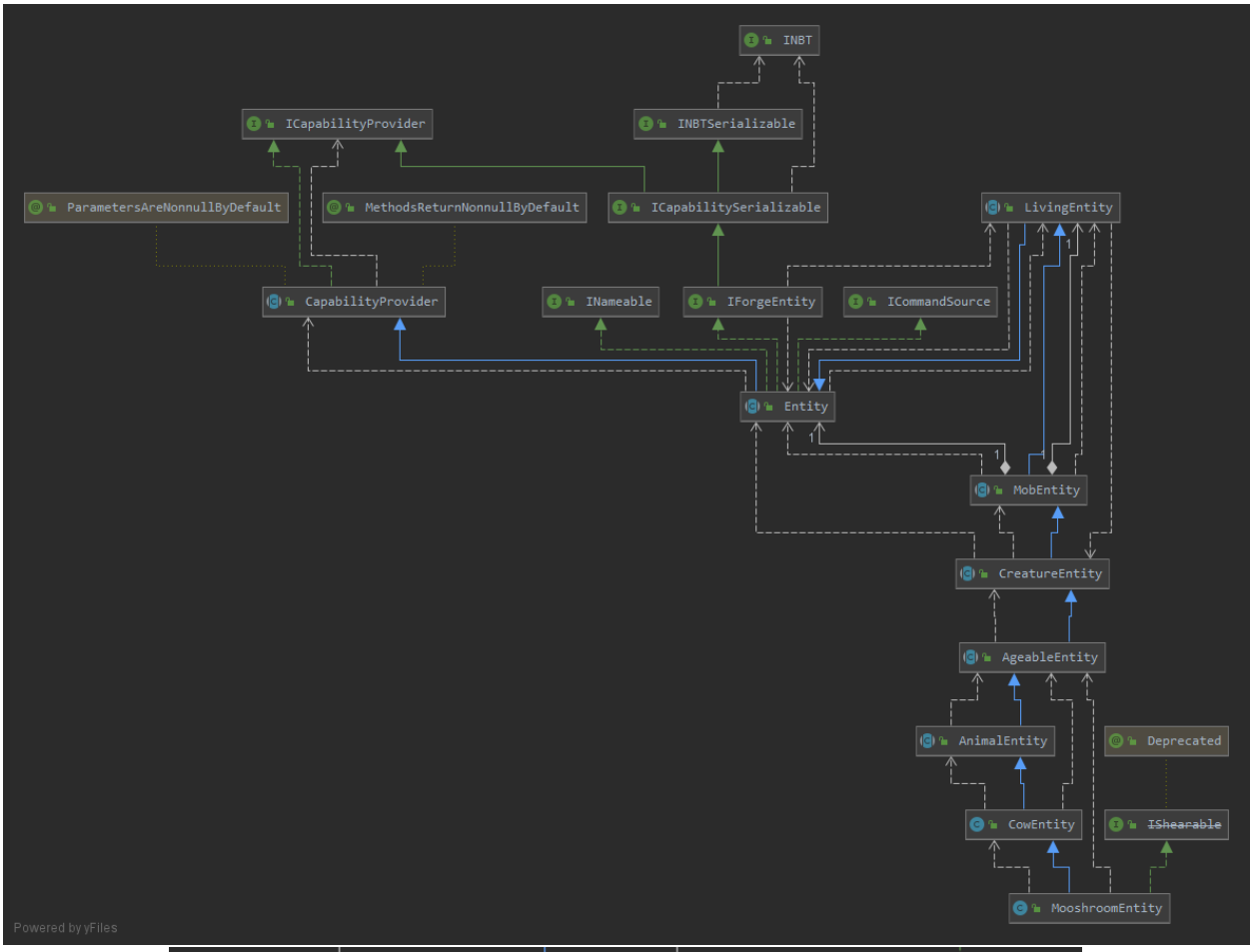


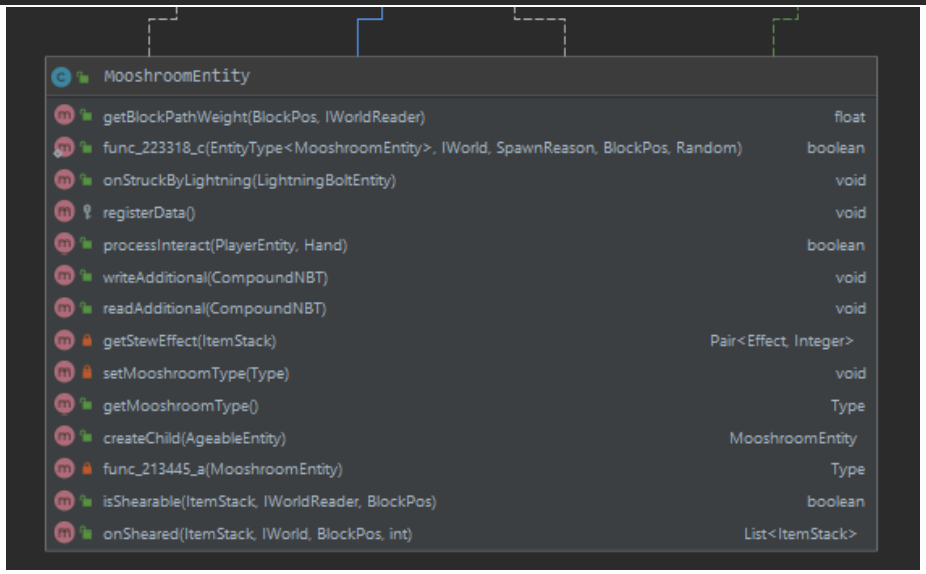
Forge Minecraft is the subproject of Forge which deals directly with modifications to the base code of Minecraft. Due to the amount of classes present, the above diagram only displays a diagram of how the packages are structured. From a lower level, many of the classes interact with one another in between packages.

For some of the classes within this subproject, there are little to no changes between Forge Minecraft and vanilla Minecraft. On the other hand, Forge Minecraft may redesign classes from the original implementation in order to add new features. An example of redesigning a base Minecraft class is Forge's reimplementation of rendering. The Forge team created a custom rendering engine on top of Minecraft's 1.10 engine, allowing more adaptability for modders to load custom models (such as OBJ files). One of the interesting aspects of the design in Forge is that the project chooses to keep the same file / package structure as the base Minecraft game (located in projects/clean). The only area that differs is the connection to wrappers that are found in Forge Main.



A – MooshroomEntity from projects/clean





B – MooshroomEntity from projects/forge

Other Modules

docs

- This folder contains all the documentation pertaining to the project (setting up your environment, what to do when you're encountering peculiar issues, etc.).

projects/clean

- This folder contains the code for Minecraft without any changes or modifications, i.e. the decompiled and de-obfuscated vanilla game. If one were to work on the Forge codebase, then no changes should be made to the code in this directory; gradle will run into issues during the build cycle if there are any.

mdk

- This folder contains the code required to develop mods with Forge. For this project, we are not too concerned with this as we are working on the Forge codebase and not developing any mods.

jsons

- This folder contains all the JSONs for configuring every specific setting of Minecraft Forge (since version 1.8).

patches

- This folder contains all the patch files required to generate the codebase. After making a change to Forge's codebase, you will need to run the appropriate tests on your code to make sure it's bug-free (for the most part) and after getting the green light, you will need to generate patch files and submit those in a pull request. Those files are then executed when generating a build and incorporate the changes you've made.

Software Development Process

There are several software development processes that one could consider for usage when contributing to an open source project of decent size. However, given the specificness of this course's project details, certain processes would not necessarily be viable. We will be taking a look at a few processes, weighing out the pros and cons of each as a means to determine their feasibility in contributing to Minecraft Forge.

Waterfall

Waterfall is a plan-driven model that builds on top of each subsequent process in the general workflow, i.e. each task is dependent on the output of the previous task. This model is generally comprised of the following phases:

1. Requirements definition → gathering details and specifications from the user
2. System and software design → using the established requirements to design a "blueprint" for the software
3. Implementation and unit testing → implementing the given design and testing each component
4. Integration and system testing → integrating the implemented system into a mock environment and running the appropriate tests to ensure it is functioning properly
5. Operation and maintenance → maintaining the software overtime

A benefit to following waterfall would be the fact that requirements are clearly defined at the beginning of the process, allowing for an easier time to design, implement, and test the product. In addition to the clear definition of requirements, more control over the project's development schedule can be established since everything is generally set in stone. Overall, this would work in our favor given the short amount of time we have for working on this project - managing time and the workloads would be easier.

However, on the occasions that requirements do change in the process, one would have to start from the beginning of the whole process. This can be quite time consuming and expensive since each phase would have to be revisited and redone in order to accommodate for the changes. Given the lack of time we have for this project, it would be infeasible to follow this

process since requirements are always changing with each frequent update to Minecraft and Minecraft Forge.

Extreme Programming

Extreme programming is a type of agile software development methodology that heavily emphasizes iterative development. Notable features include frequently releasing small builds, writing automated unit tests for functionalities prior to implementing them, and continuously refactoring to keep the code maintainable. Furthermore, other common practices of extreme programming include pair programming, continuous integration, and having an on-site representative of the end-users to readily provide requirements to the team.

We found that given the little amount of time we have to work on this project, planning incrementally and designing only what is necessary to meet requirements would be beneficial. Also, incorporating continuous integration and test-first development practices would help save us a lot of time in addition to giving us a good idea of how to implement something beforehand. All of these practices are parts of extreme programming which is what gave us a reason to consider it. However, with extreme programming comes frequent small releases, constant code refactoring, pair programming, and having a readily available end-user to consult - all of which is infeasible for us to adopt given the time constraints and limitations of this project.

Reuse-oriented Software Engineering

Reuse-oriented software engineering is a practice that involves integrating existing components (such as commercial off-the-shelf systems or frameworks) into some product or system (as opposed to the general practice of coding from scratch). This model can be broken down into the following stages:

1. Component analysis → components are chosen based on how well they can fit a given requirement
2. Requirement modification → requirements for a product or system are modified based on the specifications of the available components
3. System design with reuse → the system is designed based on the chosen components; new software is only developed if a component is unavailable
4. Development and integration → the components are integrated into the system

A benefit to following this practice is its frugality compared to other software development processes. Systems would already be present and free to use, so one can spend less time on development as a result. This would essentially lead to faster development and deployment, benefiting us greatly with the one-month time constraint for this project.

Despite the faster development this practice promotes, it brings a few issues to the table, such as mismatches between a product's functionality and the client's requirements. As more work is put into deriving compromises (i.e. modifying requirements) based on the availability of components, the potential for a product that doesn't meet the client's needs becomes more apparent. In regards to this project, our pull requests would have a higher chance of getting rejected due to the fact they might not appropriately address the given issues. In addition, Minecraft Forge consists of many modules that are interdependent with each other; it would be difficult to find a component that could be integrated into the codebase without causing a plethora of errors.

Kanban

Kanban is an agile software development methodology centered around the use of a Kanban board by the team. It helps with visualizing the team's work by having tasks written down on sticky notes and classifying them on the board under categories including "To do", "In progress", "Needs verification", "Under verification", "Completed", "Blocked", and more depending on the team's needs. Furthermore, Kanban limits the work in progress (WIP) of the team by requiring that the team set a maximum number of tasks per category on the board (this number may or may not be the same between categories). For example - if the "In progress" column were assigned a numerical value of 6 by the team, that means that there can only be at most 6 tasks under that category at any given point in time. This controls the workflow and ensures there isn't a plethora of "dangling" tasks under a given category. If a workflow does become full, any work needing to move into that category will be blocked, creating a visual image of what work should be finished first. As a result, this allows for people to be more focused towards working on current tasks and ensuring that they get completed with the utmost quality. The methodology features members of the team having daily stand-up meetings by the board and team members having the freedom to take out any task they wish.

We decided that this methodology suits our teams needs the best and will be making small adjustments to it as we see fit to even better suit our needs in addition to the project's limitations. This includes having our Kanban board online hosted on Trello since the members of our team will not be co-located while working on the project. The choice of Trello is simply because it is free, intuitive, and all members of the team have past experience working with it.

Another consequence of the team members not being co-located is that stand-up meetings will sometimes have to be done online and will take place 2-3 times a week as opposed to daily. This project is not the only responsibility we have during this term, and thus time must be managed appropriately. One last modification to Kanban we are making is that tasks will be assigned a priority based on how much their completion is relied upon by others. This is to ensure that the number of blocked tasks is kept at a minimum and to also boost productivity.