MINISTRY OF EDUCATION

TECHNICAL UNIVERSITY
OF CLUJ-NAPOCA, ROMANIA

FACULTY OF AUTOMATION AND COMPUTER SCIENCE

AN APPROACH TO GAME EXTENSION VIA MINECRAFT MOD DEVELOPING - ARMOR TINKERS

LICENSE THESIS

Graduate: **Tudor Florentin COROIAN**

Supervisor: **Conf.Dr.Ing. Lia-Anca HANGAN**

2022

Contents

Chapter 1. Introduction - Project Context

1.1. Project context

1.1.1. Problem statement

In 2009, Minecraft was created by Markus Persson, also known as Notch and was originally called Cave Game. It was an unusual game at the time, because it also involved gathering resources in order to build the structures wanted [1]. As the well-known game developed by *Mojang*, and purchased in 2014 by *Microsoft*, *Minecraft*, evolves rapidly, the expectations of players are harder to meet. With each update, there is only so much new content that can be added. As of the date of the latest release, *Minecraft 1.19 - The Wild Update* (June, 7th 2022), there is no rework or revamp done on the armor system and armor mechanics.

Some players argue that the current armor mechanics of the game are long due for an update. The armor system is the same as it was in the beginning days of *Minecraft*. There are four armor pieces (helmet, chestplate, leggings, and boots) which can be made from a single material (leather, chains, iron, gold, diamond or netherite). Even if they provide different attribute values for armor points, armor tougness or knockback resistance, the system is very rigid in terms of how the armor can be crafted.

With the apparition of the *Tinkers' Construct* mod, for *Minecraft 1.5.2*, players saw an improvement on the tool system. For the first time in the history of the game, tools were able to be crafted from more than one material. Depending on the materials used, the final tools would have different durabilities and different effects granted upon the player when using them. This spiked the curiosity for the players, wandering if such a change was viable for the armor system too.

Another complaint of the players was the limited resource pool for armor crafting. With the current mechanics, the progression is strictly linear: leather to chain to iron to gold to diamond to netherite. Later updates of the game tried to break this linearity, by implementing special mechanics for certain types of armor such as leather boots preventing the player from sinking into powdered snow or freezing, while golden helmets had the advantage of turning *Piglins* from hostile mobs towards the player to neutral mobs (neutral mobs do not attack the player, unless provoked). However, these attempts did not yield any significant results, as the progression was still strictly linear for most players.

Other mods tried to add their own types of armor, that intertwined with the existing ones. This was still not enough, because the player were still able to feel like they are progressing through the game while following the path that lead to the next better armor than the one the player was currently wearing.

As such, the problem of armor crafting, upgrading and available resources affect the players of the game and other eventual developers, the impact of which is limited

used of the available armor mechanics and an overall lack real, non-linear progression throughout the existing armor types.

1.1.2. Proposed solution

Given this problem, I propose the implementation of a *Minecraft* mod called, *Armor Tinkers*, which aims to improve the current armor mechanics while also providing new material for armor crafting. The mechanisms for this mod are inspired from the already mentioned *Tinkers' Construct* mod. However, the challenge in this case is to apply the same principles to armor items. It is worth mentioning at this point that tools and armor do not behave in the same way, thus the complexity of the problem is higher.

As such, the *Armor Tinkers* mod is evolving towards a full replacement of the existing armor mechanics, providing a better choice for players oriented around a PvE or PvP play styles. It should allow the player to craft armor out of different materials and pieces and provide them with different attribute values that are related to the type of material used. This would change the linearity of progression, because, even if the player uses the best materials to craft the perfect armor, it would be too expensive to keep finding that end-game material in order to repair the armor. Thus, the mod shall provide multiple armor types that offer the same best statistics, but this aspects should not be revealed to the player right away. Rather, they should have to play with the mod to figure out what parts of the armor can be made with cheaper materials, while also providing the same (or almost the same) attributes.

1.2. Motivation

The motivation for this project is that *Minecraft* is a very well-known game, that only increased in popularity since its release. The modding community of this game (i.e., the players who enjoy the mods and the developers who make them) aims to improve the game in meaningful ways. Some mods that were developed alongside *Minecraft* caught the attention of the game developers that work at *Mojang* and *Microsoft*. Thus some ideas implemented by people with technical knowledge served as inspiration for the actual developers of the game and made it into *Minecraft*.

1.3. Resources

The resources needed for this project are presented in the list bellow.

List 1.1: Resources needed

- working copy of the game: *Minecraft 1.18 - Caves & Cliffs Part II*, bought directly from the vendor (owned by the student)
- copy of the *Forge API* library, with the initial stubs (downloaded from their website)
- Intellij IDE (downloaded with studend credentials from the official website)
- AESPRITE - software used for drawing textures and pixel manipulation (downloaded for free for their official website)
- Blockbench - software used for drawing custom block models (downloaded for free for their official website)

Chapter 2. Project Objectives

2.1. Main objective

The main objective of the *Armor Tinkers* mod is to add new types of armor to the *Minecraft* game. This is because players of the game (including myself) want a new and improved armor system, that would allow for more variability in the way that it works. Something similar was done for the tools system by the mod *Armor Tinkers* and it became very famous and played by the community. *Armor Tinkers* should provide mechanics for crafting the new types of armor out of different materials (not just one material, as it is the case for the base game). The attributes of each piece of armor will change, depending on the materials used for crafting it. This mod is suited for players of the *Minecraft* game, who need a new and improved armor system. The *Armor Tinkers* (or *AT*) mod is a modification (hence, the name "mod") brought to the original game, through different means. It aims to improve the overall experience of the player and provide alternatives for the existing mechanics of the game. Unlike the current armor system implemented in the game, this product will provide more options in terms of armor and improve the general experience of the player.

2.2. Secondary objectives

Besides the main objective specified previously, the mod should also add new ores and trees (with natural generation) to the game, in order to provide the player with new materials for armor crafting. Additionally, the mod should provide values for the armor attributes within the limits that are considered standard for this game.

Based on the primary and secondary objectives, it can be shown that a successful solution needs to meet the criteria specified in the following list.

A successful solution would be:

- easy to integrate with the existing game
- easy to play with
- able to provide the player with new materials to craft the armor
- able to add new mechanics for armor crafting
- able to provide better armor at the cost of more effort from the player
- easy to maintain and update in the future

2.3. Requirements

The requirements presented in this section, together with the user needs that will be presented in Subsection 2.4.3, will generate the necessary features that this mod should have.

2.3.1. Functional requirements

FR-AT02 **Provide new materials** The mod should add to the game new material for the player to choose from. These materials should have attributes that are comparable to the ones already existing in the game. More details about the material types and tiers are presented in Subsection 4.2.6.

FR-AT03 **Provide new armor** This is related to the main objective of the mod. It should allow the player to obtain new types of armor, with different values for their attributes (e.g., defense points, armor toughness or knockback resistance). The armor should be wearable by the player and it should interract with the damage inflicted upon them.

FR-AT04 **Provide new crafting mechanics** This is also related to the main objective of the mod. It should provide the player with new types of crafting recipes, with the aim of making the game more immersive and add another challenge into the usual gameplay.

FR-AT01 **Continuity** The mod should provide means for the player to access any new items and blocks added. This can be done by means of crafting from materials already existing in the *Vanilla* game or by gathering the new materials that are generated automatically in the world.

2.3.2. Non-functional requirements

NFR-AT01 **Aesthetics and immersion** The mod should look like *Minecraft*. The player should not be able to notice big differences between the base game and the mod. In other words, the mod should use textures and colors that are already present in the base game.

NFR-AT02 **Response time** In the context of the new crafting mechanics, the mod should have comparable response times for actions such as opening a crafting GUI or performing operations on the tick.

NFR-AT03 **Compliability** The mod should comply to the existing game mechanics. In other word, high effort should be reward better than low effort, but in comparable limits to the *Vanilla* game. The mod should not feel too hard to play, but also it should not provide overpower the player.

2.4. Mod features

In order to determine the features that this mod provides for the player, it is necessary to first determine the stakeholders of the project, the users and their needs.

2.4.1. Stakeholders

STH01-PD **Project Developer (PD)** The student who is responsible for studying the domain and develop the application. Main stakeholder of the project. Interested in providing a scalable and maintainable solution for future updates. Focuses on providing the project analysis, on development and allocates the resources.

STH02-TA **Thesis Advisor (TA)** The teacher whose duty is to monitor the activity of the project developer. Important stakeholder for the project. Interested in obtaining satisfactory results. Helps allocating the resources on the project (mainly time management). Provides additional ideas and guidelines for the project developer.

STH03-LTEB **License Thesis Evaluation Board (LTEB)** The group of teachers who are interested in the results of the application, as well as the design and the implementation of the system. Important stakeholder for the project. Interested in the results of the application. Responsible for the evaluation of the system.

STH04-PYD **Player-devs (PyD)** The group of players with technical background that want to use the project as a starting point for their own mod. Marginal stakeholder of the project. Interested in the good implementation of the project. Interested in the documentation of the project. Interested in the scalability of the project.

2.4.2. Users

USER01 **Player** User that adds the mod to their own copy of the game. Progresses through the entire mod until reaching the desired set of attributes for the armor. The stakeholders for this user are STH04-PYD, STH01-PD and STD03-LTEB.

USER02 **Developer** User that extends the mod. Reads the documentation to understand the implementation of the project. The stakeholders for this user are STH04-PYD, STH01-PD and STD02-TA.

2.4.3. User needs

Table 2.1: Resource availability

ID	USRN-01
Priority	0
Current solution	Default resources are available. Additional resources (added by the mod) are not available.
Proposed solution	Additional resources (added by the mod) will be made available through means of crafting, mining, and gathering.

Table 2.2: Recipe availability

ID	USRN-02
Priority	0
Current solution	No current solution within the base game.
Proposed solution	Recipes for each object added by the mod will be displayed with the help of another existing mod (*JEI*).

Table 2.3: Crafting system

ID	USRN-03
Priority	1
Current solution	Default crafting system available (3x3 grid, furnace)
Proposed solution	Mod-specific crafting stations

Table 2.4: Identification and distinction

ID	USRN-04
Priority	4
Current solution	No current solution within the base game.
Proposed solution	Integration of the current mod with *TOP*.

Table 2.5: Status indicators

ID	USRN-05
Priority	3
Current solution	Default display of "Armor Points" and "Toughness Points".
Proposed solution	Default display of "Armor Points" and "Toughness Points" with the attributes for the new armor types integrated.

2.4.4. Product features

Based on the previously discovered user needs and the requirements presented previously, the mod will have the following features to accommodate each of them.

FEAT-AT01 **Resource distribution** The mod shall distribute any new resources in a sensible manner to the progression of the game and the current rarity-usage-mining level triplets. This will be done by adding new resources with attributes comparable to the ones for the base game materials. Additionally, any new resource added in the game that is not generated in the world shall have the means to be obtained from items that already exist in the *Vanilla* game. This feature covers the *Resource availability* need.

FEAT-AT02 **Tinkering tables** The mod shall add the following tinkering stations: *Pattern Station*, *Armor Part Station* and *Armor Forge* (two tiers). These stations will be used to craft the patterns necessary for obtaining armor parts (in the *Armor Part Station*), which will then be used to craft one of the four pieces of armor available (*helment, chestplate, leggings* or *boots*). This feature covers the *Crafting system* need.

FEAT-AT03 **Recipe display** The mod shall provide instructions for how to craft the added armor pieces. This will be done by integrating the mod with another utility mod, *Just Enough Items*. This feature covers the *Recipe availability* need. More details about the integration aspects will be presented in Section 4.9.

FEAT-AT04 **Block information display** The mod shall provide details about the blocks added to the game, in order for the player to distinguish between them easily.

Additionally, the mining level of each block shall be displayed. This will be done by integrating the mod with another utility mod, *The One Probe*. This feature covers the *Identification and distinction* need. More details about the integration aspects will be presented in Section 4.9.

FEAT-AT05 **Status attributes** The mod shall display the values of the attributes for the added armor (mainly the *Defense point*. This will be done by implementing formulas to determine the protections gained from each piece of armor, based on the type of materials used in their part. This feature covers the *Status indicator* need.

Figure 2.1: Relation between features with user needs and functional requirements

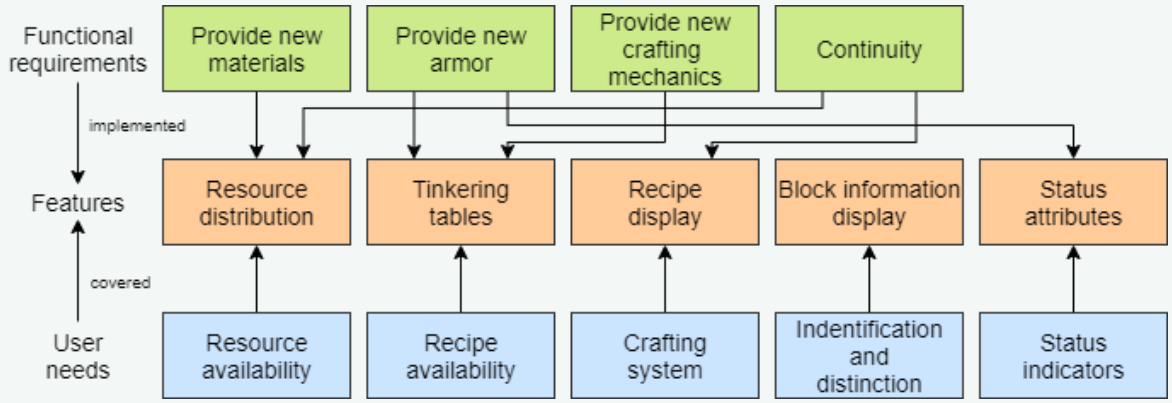

Figure 2.1: Relation between features, user needs and functional requirements

Chapter 3. Bibliographic Research

3.1. The game of *Minecraft* and its rules

Minecraft is a well-known game that focuses around exploration, breaking and placing blocks and gathering resources. Those resources can then be used to craft tools, which allows the player to access even better resources, establishing a natural progression. The game also has entities, which are dynamic objects that move based on an AI algorithm [2]. Even if the game has a progression tree, it is not mandatory for the player to follow it, as there are no quests that they have to complete. Rather, the game takes a much broader approach, allowing the players to play the game in any manner they want. The name *Minecraft*, according to Markus Persson (also knows as Notch), comes from the two most common actions in the game, *mining* and *crafting*. The name is also a subtle hint for Blizzard Inc., which release games such as *Warcraft* or *StarCraft* [3].

Remark 3.1: Minecraft Java Edition

Before moving forward, it should be specified that this paper presents aspects of the *Minecraft Java Edition* (the version for PC). Other version such as *Pocket Edition* for phones or *Bedrock Edition* for consoles exist, but the paper and the mod developed relate solely to the PC version of the game.

3.1.1. Minecraft timeline

To get a better understanding of how much the game has evolved over time and the amount of work put into the game, the list bellow presents the *Minecraft* timeline, from 2009 to the present day [4].

- Minecraft first release - May 10th, 2009
- Minecraft Indev release - November 10th, 2009
- Minecraft Infdev release - February 27th, 2010
- Minecraft Alpha Version release - June 30th, 2010
- Minecraft Beta Version release - December 20th, 2010
- Minecraft 1.0 Official release - November 18th, 2011
- Minecraft 1.1 *Adventure Update* - January 12th, 2012
- Minecraft 1.2 - March 1st, 2012
- Minecraft 1.3 - August 1st, 2012
- Minecraft 1.4 *Pretty Scary Update* - October 25th, 2012
- Minecraft 1.5 *Redstone Update* - March 13th, 2013
- Minecraft 1.6 *Horse Update* - July 1st, 2013
- Minecraft 1.7 *The Update that Changed the World* - October 25th, 2013
- Minecraft 1.8 *Bountiful Update* - September 2nd, 2014
- Minecraft 1.9 *Combat Update* - February 29th, 2016

- Minecraft 1.10 *Frostburn Update* - June 8th, 2016
- Minecraft 1.11 *Exploration Update* - November 14th, 2016
- Minecraft 1.12 *World of Color Update* - June 7th, 2017
- Minecraft 1.13 *Update Aquatic* - July 18th, 2018
- Minecraft 1.14 *Village & Pillage* - April 23rd, 2019
- Minecraft 1.15 *Buzzy bees* - December 10th, 2019
- Minecraft 1.16 *Nether Update* - June 23rd, 2020
- Minecraft 1.17 *Caves & Cliffs: Part I* - June 8th, 2021
- Minecraft 1.18 *Caves & Cliffs: Part II* - November 30th, 2021
- Minecraft 1.19 *The Wild Update* - June 7th, 2022

The *Armor Tinkers* mod presented in this paper was designed to work with *Minecraft* version 1.18, released in November 30th of 2021.

3.1.2. Minecraft as a sandbox game

In general, games are characterised by a set of rules that the player has to follow in order to complete the game. This is true for most games, but there are certain games available for the public that do not constrict the player to a rigid gameplay.

Minecraft is part of a category of games called *"sandbox games"*. This term (i.e., sandbox game) is defined on the Merriam-Webster website as "a video game or part of a video game in which the player is not constrained to achieving specific goals and has a large degree of freedom to explore, interact with, or modify the game environment". As such, *Minecraft* encourages a unrestricted play style, with no hard achievements or path that the player has to follow. This is similar in concept with a physical sandbox, in which kids can play in the park. The similarity is obvious right from the beginning of the game. There are no instructions for the player, rather they are spawned in an open world that can be explored freely. More on this topic will be described in the next section (i.e., *Game mechanics*). Thus, it can be safely assumed that *Minecraft* is a game with no rules.

3.1.3. Minecraft game modes

It was previously specified that *Minecraft* imposes no rules on its players. However, just as the physical sandboxes have some guidelines to how someone can play inside them, the game provides the player with four different options [5].

List 3.1: Game modes in Minecraft

- **Creative** - suitable for players that are focused on building
- **Survival** - the core game mode of *Minecraft*
- **Spectator** - suitable for assisting other players, with no interaction
- **Hardcore** - similar to **Survival**, but no respawning after death

It is worth noting at this point that a player can create multiple worlds, each with its own game mode, but can also switch the game mode within an already created world, via commands [2]. However, a rule born out of the player community is that switching the game mode within a *survival* world is considered cheating. There is no penalty implemented in the game for this, and the action is available to any player that knows the command and types it in the chat.

Creative game mode

In the *creative* game mode, the player in immune to all types of damage and has the ability to fly. Additionally, they have access to unlimited resources, via the *Creative Menu*. This game mode, as the name suggests, aims to ease the resource gathering part of the game and lets the player focus on building structures, limited only by the creativity of their mind.

Experienced players (i.e., Minecraft veterans) use this game mode as a testing ground. Usually, complicated buildings and contraptions are first designed in a separate, *creative* world, and then constructed in the *survival* world.

Lastly, the player is able to break any block in the world instantly, even without the required tool.

Survival game mode

The *survival* game mode is the main way in which players choose to play the game. All the mechanics (that will be described in the next section) are designed specifically for this game mode.

The player can take damage from a number of sources, including, but not limited to attacks, falling, starving, freezing, burning, drowning. In total, the player has 20 HP (i.e., health points), represented in the game by 10 red hearts, and 20 SP (i.e., saturation points), represented by 10 chicken drumsticks. Figure 3.1 shows both of these metrics and also the XP (i.e., experience) bar and the 9 slots in the hotbar. The hotbar holds items that need to be used by the player quickly. The one selected slot represents the item that the player is currently holding in their hands. The selection can be moved using the wheel of the mouse to any of the other 9 slots.

Figure 3.1: Minecraft player hotbar

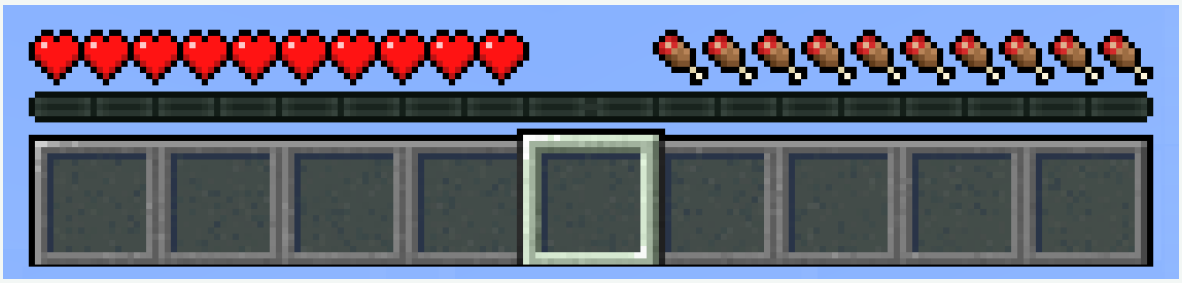

Figure 3.1: Minecraft player hotbar

Any damage received by the player is deduced from the maximum of 20 HP. Upon reaching 0 HP, the player dies and can choose to respawn at the selected location (i.e., world spawn, the place where they first spawned in the world) or near the last bed they slept in. Additionally, the player has limited inventory space (9 slots in the hotbar and 36 slots in the actual inventory), meaning that a player can only care a certain amount of items at any given time.

In the same manner, upon reaching 0 SP, the player starts taking *hunger* damage, 1 HP every second. However, saturation points can be restored by eating food items available in the game.

The player does not have the ability to fly by default (although it can be obtained with the use of an item called *Elytra*) and any block placed is deducted from the total ammount in the inventory. Additionally, any tool that has durability will be deducted 1 durability point upon its use. At 0 durability, the tool breaks and dissapears from the player's inventory.

The player is not able to instantly break blocks. They need to use the right tool, depending on the block type and the material of it. More details about this (and the *survival* game mode) will be presented in the next section.

Spectator game mode

In the *spectator* game mode, the player is not able to interact with any item or block in the world. However, they can pass through solid blocks and fly. This game mode is used for players that want to assist other players in their gameplay, without being able to directly use items or blocks.

The player in *spectator* mode is invisible to other players and is immune to all sources of damage.

Hardcore game mode

The *hardcore* game mode is very similar to the *survival* game mode, with the only difference being that, upon death, the player cannot respawn again, and the progress in that world is lost forever. Also, the difficulty level of the world is locked at *hard*. More about difficulty levels in *Minecraft* will be presented in the next subsection.

A comparison between the four available game modes is presented in Table 3.1, based on the actions that are available to the player in each of them.

Table 3.1: Comparison between game modes

Action	Creative	Survival	Spectator	Hardcore
Flight	YES	NO	YES	NO
Immune to damage	YES	NO	YES	NO
Instantly break blocks	YES	NO	-	NO
Unlimited resources	YES	NO	-	NO
Unlimited inventory space	YES	NO	-	NO
Unlimited durability	YES	NO	-	NO
Interaction with items	YES	YES	NO	YES
Interaction with world	YES	YES	NO	YES
Pass through solid blocks	NO	NO	YES	NO
Respawn after death	-	YES	-	NO

As a remark, it is worth mentioning that a dash in the table above means that the action is not applicable to that game mode. For example, in the *creative* game mode, the player is immune to all types of damage, and thus cannot die. Therefore, there is no way (nor need) to tell if the player will respawn after death.

3.1.4. Minecraft game difficulties

Often times, games come with difficulty options and means to change the game's difficulty, in order to adjust certain parameters to the player's abilities. In general, there are three difficulty classes that a player can choose from: *easy, normal*, and *hard* [5]. However, *Minecraft* adds another difficulty option called *peaceful*, which mainly influences hostile mobs spawning. More information about *mobs* will be presented in the next section.

List 3.2: Difficulty options in Minecraft

- **Peaceful** - no hostile mobs spawning
- **Easy** - hostile mobs spawn, dealing reduced damage
- **Normal** - hostile mobs spawn, dealing medium damage
- **Hard** - hostile mobs spawn, dealing a great amount of damage

Remark 3.2: Other difficulty factors

In addition to the four difficulty settings provided by the game, *Minecraft* has another two attributes that influence the difficulty. These are **Moon phase**, which affects slime spawning rate (both during the day and night), and **Regional difficulty**, which is a modifier that influences mob related attributes of the world More about these two topics can be found on the Minecraft Wiki website [5].

The difficulty of each world can be set when the world is first created and, just like the game mode, can be changed at any time via commands or by using the game menu.

Peaceful

In the *peaceful* difficulty, hostile mobs do not spawn anywhere, regardless of the light level or proximity to the player. The only hostile mobs that can spawn are *shulkers, hoglins*, and *zoglins*. No neutral mobs (such as *spiders, cave spiders, wolves, endermen* and so on) can deal damage to the player.

The player regains health over time quicker than in the other difficulties. It is still possible for the player to die, if they receive a lot of damage quickly (such as from fall damage). Additionally, TNT explosion deal no damage to the player, but they knock them back.

In this difficulty, the hunger bar never depletes.

Easy

In the *easy* difficulty, hostile mobs spawn, dealing less damage than in the normal difficulty. The hunger bar can be depleted and, upon reaching 0 SP, the player will take 1 HP damage every second, but the health bar cannot drop under 10 HP (or 5 hearts). Additionally, zombies cannot break doors or turn villagers into zombie villagers.

In this difficulty, cave spiders and bees cannot poison the player, and attacks from the wither will not give the player the wither effect. However, the player can still get the wither effect from wither skeletons.

Normal

In the *normal* difficulty, hostile mobs spawn, dealing standard damage. The hunger bar can be depleted and, upon reaching 0 SP, the player will take 1 HP damage every second, but the health bar cannot drop under 1 HP (or $\frac{1}{2}$ hearts). Additionally, zombies cannot break doors, but have a 50% chance to turn a villager into a zombie villager, upon killing it.

In this difficulty, vindicators are the only mobs (except for the mobs that can cause explosions) that are able to break doors.

Hard

In the *hard* difficulty, hostile mobs spawn, dealing more damage than the standard amount. The hunger bar can be depleted and, upon reaching 0 SP, the player will take 1 HP damage every second. When the health bar reached 0 HP (from hunger damage), the player dies, with the message "[PLAYER_NAME] HAS DIED OF STARVATION." Additionally, zombies can break doors and can turn any villager into a zombie villager.

In this difficulty, the items dropped by hostile mobs upon their death are more valuable.

A comparison between the four available game difficulties is presented in Table 3.2, based on the events that can take place in each of them.

Table 3.2: Comparison between game difficulties

Effect	Peaceful	Easy	Normal	Hard
Hostile mobs spawn	NO	YES	YES	YES
Deplete hunger	NO	YES	YES	YES
Regenerate quicker	YES	NO	NO	NO
Mobs can poison	NO	NO	YES	YES
Damage dealt by mobs	NO DMG	DECREASED	STANDARD	INCREASED
Starvation damage	NO	YES	YES	YES
Death by starvation	NO	NO	NO	YES
Mobs drop better loot	NO	NO	NO	YES

A very comprehensive table with how the amount of damage dealt by mobs varies, based on the game difficulty, as well as how the *Moon phase* and *Regional difficulty* influence certain effects can be found on the Minecraft Wiki website [5].

3.2. Minecraft as a platform for learning

Besides playing, *Minecraft* also has the potential of being a great platform for kids to learn any subject within the STEM fields of research. This idea was started by *Mojang Studios* with their release of *MinecraftEdu* for teachers and students. Since then, a lot of mods were created to promote a learning environment within *Minecraft*. A team of four people from the *University of Central Florida* developed a plugin for the game, that introduces the players to programming [6]. They implemented new blocks that act as instructions for a robot. The programming language developed by them is

called *CodeBlocks* and it controls a robot that navigates and interacts with the world within the game.

Another way in which *Minecraft* can be used in a education way is described in [7]. It describes how Minecraft, together with computational tools such as natural language processing and machine learning can be used to assess student learning in the engineering domain.

3.3. Minecraft mappings and Code obfuscation

Minecraft's code is very easy to decompile. In order to protect the game from reverse engineering attack that would expose core mechanics of the game, *Minecraft* ships with obfuscated code. Wroblewski presented an analysis on the security of code obfuscation together with a method for obfuscating machine code in [8], presented in 2002 at SERP. Obfuscation is the method of transforming machine code from one version to another that is equivalent to the original one (i.e., the execution of the machine code on the same input data set will result in the same output data set). On the other hand, software obfuscation transforms computer programs into new versions, which are semantically equivalent with the original ones, but much harder to understand. These definitions were given by Collberg, Thomborson and Low in their technical report, "A taxonomy of Obfuscating Transformations" [9].

As such, mappings were developed that aim to deobfuscate the code of the game. Some of the mappings that exist are made by the players in the community (such as *Parchment* or *Yarn*) and offer full deobfuscation of method names and parameters, while others (such as *MCP*) only offer deobfuscation for method name. Alternatively, *Mojang Studios* decided to release the official mappings to the public, in order to promote the mod developing branch to their game.

3.4. Methods of implementation for mods

Modding in *Minecraft* started in the early stages of the game and were mainly focused on server side mods. The reason was that multiplayer *Minecraft* was more popular than single-player and the code base was still hard to interract with on the client side. For servers, plugins were easily integratable. During the time of the Alpha release, *Minecraft* began to see the the first simple client mods. The first big change happened when Michael Stoyke (also known as *Searge*) first released the Mod Coder Pack (*MCP*). Despite being decompiled in an easy manner, *Minecraft*'s code was and still is obfuscated.

During the same year, another major development happened: the first mod loader (Risugami's Mod Loader) was released. This was a game changer, since prior to this, if the player wanted to play with two mods at once, they would have to manually merge the code from both mods. The mod loader created a framework where mods could simply hook into the its code to perform common tasks [10].

Minecraft players who had a technical background decided to put together libraries that would help other develop their own mods. As such, in the current day, there are two options for writing mods for the game of *Minecraft*. These options are *Forge API* and *Fabric API*.

3.4.1. Forge API

Along with the Beta release, many client mods began to appear again. However, even with the help of the newly developed mod loader, some of them still weren't compatible due to modifying too much of the base code. Thus, the creators of some popular mods decided to get together and develop their own framework for modding *Minecraft* and a new mode loader. This was the beginning of *Forge API* and *Forge Mod Loader*.

Forge API (known also as Minecraft Forge) is the most massive API and modloader used by mod developers to hook into Minecraft code and create mods. As opposed to the previous option described, Forge has numerous hooks (methods that access the game engine) that allow for the creation of mods with a high level of compatibility [11]. In the same manner, Forge API also allows the developers to perform bytecode manipulation, to further modify the game. Forge API was released together with Minecraft version 1.1 (January 12th, 2012), but it wasn't fully compatible with the game up until Minecraft version 1.3 (August 1st, 2012).

3.4.2. Fabric API

Fabric API and the Fabric modloader was released at the same time with the Minecraft version 1.14 – Village and Pillage (April 23, 2019). It is highly modularized, therefore, it allows players that play mods using only a certain part of the API to install only that part of the API, instead of the whole. The API and the modloader were created as a response to an allegedly slow development of Minecraft Forge (mentioned before as Forge API), around Minecraft version 1.13 – Update Aquatic (July 18, 2018). The developers of Fabric considered Forge to be exhaustively big, so they split the functionalities and discarded the unnecessary ones. Fabric API is another known method for players to develop their own mod and is also the latest one [12].

Unlike Forge API, which extends functionalities of *Minecraft*'s classes by registering the new ones to various buses, Forge API uses the *Mixin library* to inject code. This is why the API is relatively small, compared to Forge. Most of the functionalities that are more specific are implemented by third party APIs [13].

Table 3.3 presents a comparison between the two APIs. For this project, I chose to implement the *Armor Tinkers* mod using the *Forge API* because it is more robust and provides more functionalities, without the use of third party APIs.

Table 3.3: Comparison between Fabric and Forge

Feature	Forge API	Fabric API
Size	big	lightweight
Modding technique	class extension and overwriting	code injection
Has mod loader	YES	YES
Uses third party libraries?	NO	YES
Mappings	official or Parchment	official* or Yarn

3.5. Existing mods

There already exist mods that make an improvement to the armor system of *Minecraft* or add new resources to craft with or even new effects for the player, tools or armor. These mods will be presented in the following subsections.

3.5.1. Botania

Botania is a tech mod, with strong themes from the natural magic genre. The whole gameplay revolves around creating magical flowers that generate or consume mana, as well as devices and contraptions to automate repetitive tasks. Most importantly, it adds new sets of armor that far exceed the capabilities of the ones already existing in the game. It was designed to be played as a standalone game, but it work well in conjunction with other mods too.

It has no graphical interfaces. Rather everything that the player interacts with happens right inside the world. This might seem confusing at first, but it build up to a level of immersion that other mods cannot provide.

Although it adds new armor types, they follow the same principles from the base game, meaning that they are still crafted from a single material, with no room for experimenting. As it was mentioned before, this does not change the linearity of progression in *Minecraft*. It simply adds new armor types that are better (and more expensive to craft) than the ones that already exist.

The materials added by this mod do not spawn in the world naturally. Therefore, they need to be crafted with *Vanilla* resources, in crafting mechanics specific to the mod. Finally, the focus of the mod is not to improve upon the crafting or combat mechanics of the game, but rather to provide an immersive and unbounded way to engineer automation mechanisms for repetitive tasks that need to be carried out.

3.5.2. Mekanism

Mekanism is also a tech mod, but the theme is centered around a modern industrial look, rather than the natural magic genre. It adds new machinery that carries out processes for the player, jetpacks that allow the player to fly, a new armor set and also new materials. Everything within this mod is powered by electrical energy, which can be obtained from burning coal, heat generators, solar generators or even nuclear generators. It was designed to be played as a standalone game, but it work well in conjunction with other mods too.

Every new machinery added has a graphical interfaces, which makes the player feel like they are operating right inside the wiring of it. This aims to increase immersion into the game, but making the interfaces look industrial and technical. The mod doesn't have an actual goal. It just adds new functionalities to the game that aim to automate processes for the player.

Although it adds a new armor type, it is again made from the same material all over. The difference from other mods is that this armor type is more technological in nature. It can have scanners and sensors that interact with the world and it is also powered by electricity. This means that the armor itself does not take damage, rather the damaging value is subtracted from the current level of electricity stored in the armor.

The materials added by this mod spawn naturally, in the form of new ores and minerals. However, this materials are not used directly to craft the armor pieces. They

are used to build the machinery that will process the metals and transform them into circuits and batteries. This will in turn be used to craft each armor piece, together with armor plates. This process does not change the linearity of progression in *Minecraft*. It simply adds new materials between the existing ones and another armor type that is much more superior to the ones already present in the *Vanilla* game.

3.5.3. Apotheosis

Apotheosis is a mod that focuses around magic and improves many mechanics of the base game. It comes in 5 modules that work independently and can be enabled or disabled, based on a configuration file. The five modules are: enchanting module, spawner module, potion module, deadly module and the garden module. Each module modifies an aspect of the game.

The spawner module changes the way mob spawners work, by allowing the player to customize its values and even harvest it from the dungeons in which it spawns. The enchanting module adds new enchantments to the game and also new bookshelves type, which affect the kind of enchantments that the player can get from the enchanting table. The potion module adds new potion types to the game, which give the player different effects, based on the potion that was drank. The deadly module adds new mobs to the game with enhanced powers and abilities, that are much harder to combat than regular mobs. Finally, the garden module simply modifies the limit on how high sugar cane and cactus can grow (from 3 to 30).

This mod does not add any new armor type, nor does it add new materials to the game. However, it was mentioned here because it adds effects and enchantments to the armor that already exists in *Minecraft*, thus changing the armor system of the game and making its progression more non-linear.

3.5.4. Tinkers' Construct

Tinkers' Construct is a mod that adds modular tools that can be crafted from different materials and provide different values for their attributes. In *Vanilla Minecraft* tools, much in the same way as armor, can be made from only one material. This is limiting and imposes a linearity in the progression of the game.

It adds new materials in the form of ores and mob drops and also new crafting mechanics. It does not add any new armor types. This mod was mentioned here because it is the main inspiration for how to modularize armor, much in the same way that it does with the items. It serve as a creative starting point, such that the mechanics implemented by the mod described in this paper are cohesive and can be logically followed from starting the crafting process of an armor piece to completing it.

Depending on the material type used for the tool, this mod also adds effects to it, such as increased mining speed, natural silk touch, auto smelting, self repairing over time and many others, all based on the materials that were used in the crafting process.

Remark 3.3: Current solutions

All the mods presented in this section cover some of the aspects that *Armor Tinkers* aims to improve, but not all of them. As such, Table 3.4 shows the features that each mod has.

Table 3.4: Comparison between current solutions

Feature	Botania	Mekanism	Apotheosis	Tinkers' Construct
New materials	YES	YES	NO	YES
Generated materials	NO	YES	-	YES
Crafted materials	YES	YES	-	NO
Has GUIs	NO	YES	NO	YES
New recipe types	NO	YES	NO	YES
New blocks	YES	YES	YES	YES
New items	YES	YES	YES	YES
New block entities	YES	YES	NO	YES
New armor	YES	YES	NO	NO
Modular crafting	NO	NO	NO	YES
New effects	NO	NO	YES	NO

Chapter 4. Analysis and Theoretical Foundation

4.1. Game extension

The industry of game extension dates back to the first released games, when the players that had technical knowledge and skills started exploring the source code. This opened the door for them to add different new functionalities or change the appearance of the game. Thus, in the following years, both game developers and players worked together to provide a better experience for the fans of certain games.

According to G2A (*www.g2a.com*), a popular website for purchasing and discussing about games, there are two types of games, from the point of view of their modification capabilities.

List 4.1: Types of modifiable games

- Games that are modifiable straight out of the box, because of the backdoors that were left unlocked by the game developers on purpose
- Games that require additional intervention into the code, either through directly modifying certain parameters or functions, or by injecting new functionalities into the game, with the help of libraries

On the same website, there is an article presenting *13 Best Moddable Games*. Among the titles, we see famous ones *Grand Theft Auto V, The Elder Scrolls V: Skyrim, Civilization V, The Witcher 3, Wild Hunt*, and also, one of the most popular games since its release, *Minecraft*, which falls into the second category of modifiable games.

4.1.1. Modifying Minecraft

In the *Minecraft* context, extending the games means introducing resource packs, data packs or developing mods that inject their own code into the game.

Resource packs

One option to modify *Minecraft Vanilla* (i.e., the base game, unaltered) is to design new textures for the already existing items in the game. However, in later releases of the game, resource packs became more versatile. Today, a player can change the models of items and blocks in the game, the sounds of different actions or the particles generated by certain events. These are called *resource packs*, because they modify files in the *resource* directory of the game.

It is worth mentioning that this type of game extension requires almost no technical knowledge, since it mainly focuses on changing texture files (*.png* files) for items, blocks or entities that already exist in the game.

Remark 4.1: Resource pack limitation

Through this method of modding, the player CANNOT add new items, blocks or even functionalities into the game.

Data packs

Another option to modify *Minecraft Vanilla* is to alter recipes for existing items and blocks, or to change the behaviour of them when they are being interacted with. The result of this type of modification is called a *data pack*, because it modifies files in the *data* directory of the game.

This type of game extension requires some technical knowledge, since it focuses around changing configuration files (*.json* files) for items or blocks that already exist in the game. Additionally, players can add new commands to the game, thus adding new functionalities for items, blocks or entities that are already present in the game source code.

Data packs are the halfway point between resource packs and mods, since any data pack can contain a resource pack inside it.

Remark 4.2: Data pack limitation

Through this method of modding, the player CAN add new functionalities into the game, but CANNOT add new items, blocks or entities.

Mods and modpacks

The last and most versatile option for game extension is *mod developing*. In the *Minecraft* context, a *mod* refers solely to modifying the actual source code of the game, by code injection, with the help of different libraries. The result of this type of modification is a *mod*, a *.jar* file that contains new functionalities and methods for registering new items, blocks and entities into the game. Multiple *.jar* files that are configured to work together are called a *modpack*.

This type of game extension requires a lot of technical knowledge and skill, and also a deep understanding of the Java language. It focuses around creating new classes and methods, and registering them to the appropriate event busses, provided by the API.

Mods are the most versatile option for game extension in the *Minecraft* context, because, in general, mods contain both resource packs (for the textures of the new items and blocks added to the game) and data packs (since these are the main configuration files that specify item or block properties such as recipe, rotation when placed in the world, and so on).

Remark 4.3: No limitation on mods

Through this method of modding, the player CAN add new functionalities into the game, and also new items, blocks or entities.

A comparison between the three options for game extension in the *Minecraft* context is presented in Table 4.1.

Table 4.1: Comparison between game extension options

Capability	Resource pack	Data pack	Mod
Add/change textures	YES	YES	YES
Add/change sounds	YES	YES	YES
Add/change particles	YES	YES	YES
Add/change models	YES	YES	YES
Add/change functionalities	NO	YES	YES
Add items	NO	NO	YES
Add blocks	NO	NO	YES
Add entities	NO	NO	YES

As it can be seen in Table 4.1, designing a mod for *Minecraft* is the most versatile solution, as it allows both modification of existing items, blocks or entities and the implementation of new functionalities.

4.2. Minecraft mechanics

Since *Minecraft* is a relatively old game, many of its features have been designed and implemented over the course of several years. This means that the game has a lot of mechanics, some of which are beyond the scope of this paper. In this section, only the mechanics that are directly related to the mod's concept will be presented.

Any additional information can be found on the Minecraft Wiki website [5].

While playing *Minecraft*, the player can interact with three types of objects. Any other concept (such as *effects, particles, sounds*, and so on) are just the result of the player interacting with the previously mentioned objects.

List 4.2: Objects with which the player can interact

- **Items** - present in the inventory, can be used for different actions
- **Blocks** - present in the world, static, can be broken in order to drop the respective item associated with the block
- **Entities** - present in the world, dynamic, can be interacted with via left or right clicking

In addition to these three types of objects, *Minecraft* mechanics also include different types of actions that the player can perform, as well as the *health, hunger* and *armor* system.

All mechanics presented in the following subsections are related to *Minecraft* version 1.18 and will be discussed from the point of view of a *survival* world, with the *normal* difficulty set. Parallels for the other game modes and difficulties can be drawn easily, based on the information already presented in the previous section.

4.2.1. Items

According to the Minecraft Wiki website, an item is "an object that exists only within the player's inventory and hands, or displayed in item frames or armor stands". In the next chapter, this definition will be expanded to better explain what an item is from the technical point of view.

The following list presents the types of item that exist in *Minecraft*.

List 4.3: Types of items

- **Block items** - place blocks when used in the world
- **Entity items** - place entities when used in the world
- **Tools** - used to destroy blocks or to attack enemies
- **Armor** - when equipped, provides extra protection
- **Food** - restores hunger
- **Dyes** - can color sheeps, wool, concrete and other colorable blocks or entities
- **Spawn eggs** - used to spawn mobs
- **Saplings and seeds** - used to plant trees and crops (these are a subcategory of block items)
- **Flowers** - used for crafting dyes and placing flowers (these are a subcategory of block items)
- **Miscellaneous** - items that have use only as ingredients in different crafting recipes

Every item in the inventory is *stackable*. This means that like items can be stored in the same slot, up to a certain value. The values for which items can be stacked inside the inventory are: 1, 16, and 64. In general, tools and armor only stack to 1, *eggs, enderpearls* and some other items stack to 16, while most items stack to 64.

Figure 4.1 shows the three stack types that are used in *Minecraft*. The items are, in order, from left to right, a *stone pickaxe*, 16 *eggs*, 16 *enderpearls*, and 64 *bones*.

Figure 4.1: Minecraft stack types

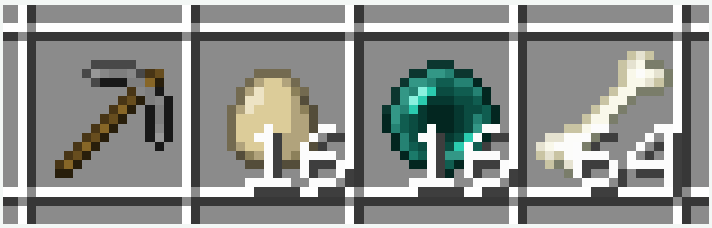

Figure 4.1: Minecraft stack types

Player inventory

While playing the game (i.e., the player is in a world), upon pressing the key 'E', the *inventory* is opened. Inside the inventory, there are 5 areas of interest. The purpose of the inventory is to allow the character to story items while exploring the world. Other storage options (such as *chests, shulker boxes* or *barrels*) are available, but their purpose is to store items in place. Figure 4.2 highlight the 5 areas of interest.

Figure 4.2: Minecraft inventory

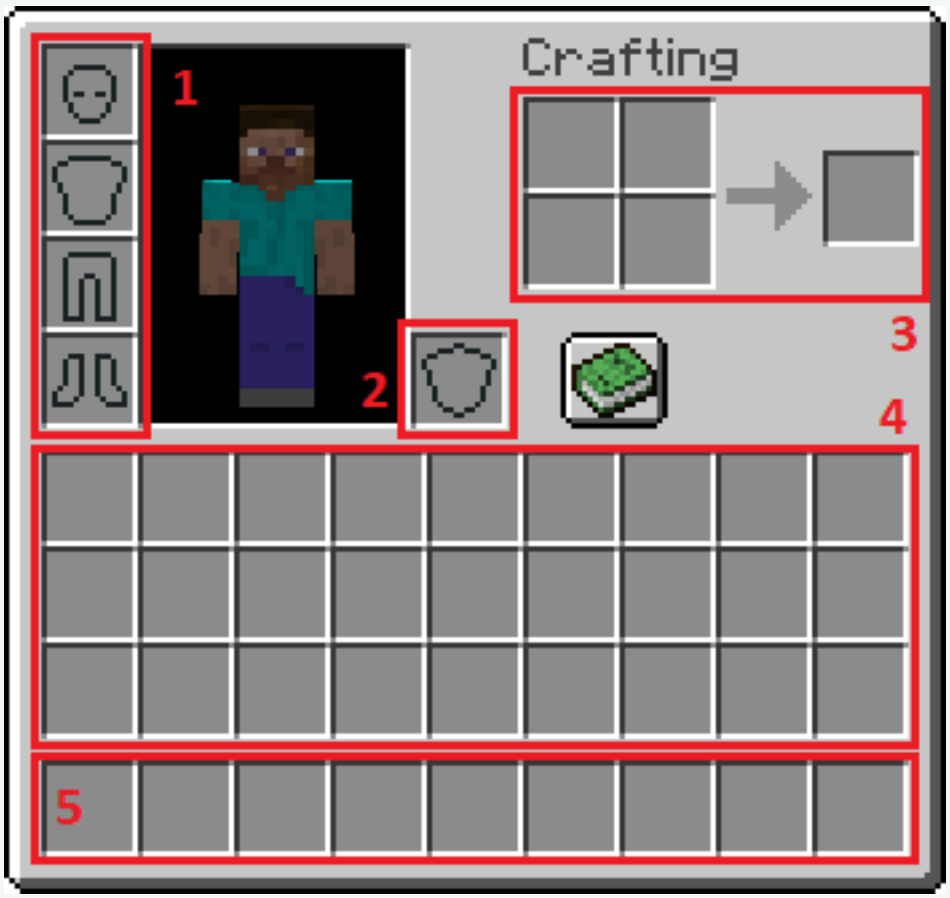

Figure 4.2: Minecraft inventory

Armor slots Armor slots are used to equip armor in order to benefit from the protection that it provides. There are 4 armor items, presented in order of their respective slots, from top to bottom: *helmet*, *chestplate*, *leggings*, and *boots*. The armor slots are marked on the figure with the number 1. Upon equipping the armor, the representation of the character Steve is updated to display the armor on themselves.

Shield slot The shield slot is a new addition to the game. It is also called the *off-hand* slot, since it can hold any item, not only a shield. Any item placed in this slot will be displayed as being held in the off-hand of the character. The shield slot is marked on the figure with the number 2.

Crafting grid (2x2) The crafting grid inside the inventory is composed of 4 slots arranged in a 2x2 array and a result slot, from where the result of the crafting recipe can be taken. While some recipes fit in this 2x2 grid, most of *Minecraft* crafting recipes need a 3x3 grid (which is provided by the *Crafting Table*. More about the crafting mechanics of the game will be presented in Subsection 4.2.4. The crafting slots are marked on the figure with the number 3.

Storage slots The storage slots are marked on the figure with the number 4. These 27 slots (together with the 9 slots from the hotbar) are used to store items inside the inventory.

Hotbar slots The hotbar slots are marked on the figure with the number 5. These 9 slots are used to quickly switch and access items in the primary hand. As mentioned before, using the mouse wheel, the player is able to select which item will be displayed and ready to use.

Tools and armor

In *Minecraft* there are special items such as *tools* or *armor* that have durability. The value of the maximum durability of an item depends on the material used when crafted. For example, a stone pickaxe will have less durability than an iron pickaxe, or a leather helmet will have less durability and provide less protection than an iron helmet. More about types of materials in *Minecraft* and how they influence the attributes of the items crafted with them will be presented in Subsection 4.2.6.

Durability is represented as green bar under the item. As the durability gets depleted, the color changes from green, to yellow, to orange and then to red. Sustaining damage from attacks takes 1 durability point from each equipped armor item and using a tool (on the correct block) will also deduct 1 durability point from the tool when the interaction is completed successfully.

Figure 4.3: Item durability

Figure 4.3: Item durability

Figure 4.3 shows, in order, from left to right: a *stone pickaxe*, an *iron pickaxe*, a *leather helmet*, and an *iron helmet*, with different values for durability. It can be noted that for the *iron pickaxe*, the durability bar is red, which tells the player that the durability of the item is getting closer to 0.

4.2.2. Blocks

According to the Minecraft Wiki website, a block is "the basic units of structure in Minecraft that makes up the game's world". Many of the blocks available in the game can be collected (with the proper tool and tier), placed anywhere in the world (with the exception that some blocks, such as *sand* or *gravel*, require a solid block underneath them), and can be used as resources in different recipes. The *Minecraft* world is arranged in a 3 dimensional grid, thus each block can occupy a position in this grid.

Blocks are said to measure $1m \times 1m \times 1m$, thus being cubes. However, some blocks such as *slabs*, are only $0.5m$ in height. Other blocks have different models (more details about models will be provided in Chapter 5), which is different from the standard cube. Some examples of such blocks are *fences* or *saplings*.

Blocks have properties based on which they can be categorized. These properties are presented in the listing bellow.

List 4.4: Types of blocks

- **solid** - The player cannot walk through them. (e.g.: dirt, iron blocks, oak logs)
- **non-solid** - The player can walk through them. (e.g.: flowers, torches, crops, corals)
- **transparent** - Light passes through them unaffected or partially dimmed. (e.g.: glass blocks, glass panes)
- **opaque** - Light cannot pass through them. (all solid blocks are opaque)
- **light-emitting** - Emit light when placed in the world. (e.g.: glowstone, torches, sea lanterns, sea cucumbers)
- **ignore gravity** - Can be placed in the world without a solid block underneath. (e.g.: cobblestone, stone, planks)
- **obey gravity** - Cannot be placed in the world without a solid block underneath. (e.g.: sand, gravel, anvils, dragon eggs, concrete powder)

A special type of block present in *Minecraft* is the *air* block. It cannot be obtained and, by convention, it occupies every free position in the world and every free slot in the inventory. Additionally, blocks have sounds and particles associated to them. Sounds are played when the player walks on the block or destroys it, while particles are shown when the block is destroyed.

Another special type of blocks are the *lava source block* and *water source block*. These blocks are non-solid, meaning that they let the player walk through them, but they impact the movement (i.e., mainly the speed).

Figure B.2 shows a house built in *Minecraft*. It displays different kinds of blocks, such as fences, fence gates, spruce stairs, spruce doors, spruce stripped logs, glass panes and so on.

Remark 4.4: How blocks are obtained

Some blocks generate naturally in the world (when the world is created and as the player explores it), while other blocks can only be obtained by crafting.

Ores

A special type of blocks that are generated in the world are *ore blocks*. When mined (destroyed) with the appropriate tool and tier, they drop *ore chunks*. These items are then smelted to obtain the respective metal. More details about how these ores generate in the world will be provided in Chapter 5.

Ores are found mainly underground and in caves. The eight ores that the player can find in the *Overworld* (one of the three dimensions existing in *Minecraft*) are, in increasing order of rarity: coal, copper, iron, gold, lapis lazuli, redstone, diamonds, and emeralds. Out of these eight ores, only iron, gold and diamonds are used in crafting tools and armor.

Based on their rarity, ores generate at different levels and in different quantities. Figure 4.4 shows the total amount of ores generated on a 128×128 sample section of the

world, based on the depth.

Figure 4.4: Ore generation

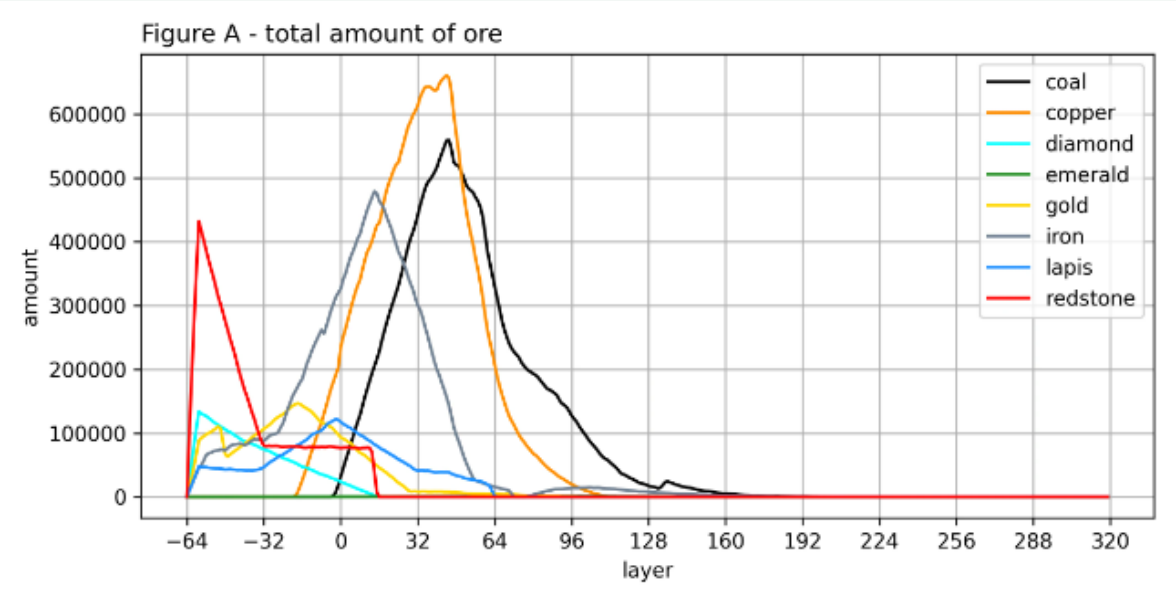

Figure 4.4: Ore generation

Trees

Trees are usually a group of blocks that are arranged in the shape of a tree. They contain two types of blocks: *logs* and *leaves*. The six types of trees that generate naturally in the world are: *oak, spruce, jungle, birch, dark oak*, and *acacia*. These have different shapes, based on the type of tree.

Some types of trees have giant variants. This means that instead of the tree trunk being a 1 by 1 tower, it is a 2 by 2. Trees that fall into this category are *spruce* and *jungle trees*. *Oak, birch*, and *acacia* have only the 1 by 1 variant, while *dark oak* has only the 2 by 2 variant.

When cutting down (destroying) the tree trunk, the logs of that specific type will be dropped. Leaves cannot spawn naturally without a tree trunk in their vicinity. As such, after cutting down all the logs of the tree, the leave will despawn naturally over a period of time and they will drop sticks and the sapling of that particular tree. However, *leaves* blocks that are placed by the player will not despawn, even if no tree trunk is found around them.

4.2.3. Entities

According to the Minecraft Wiki website, entities are objects that "encompass all dynamic, moving objects throughout the Minecraft world". As such, anything that is not static and moves in the world is considered to be an entity.

Remark 4.5: Items as entities

Items thrown on the ground from inventories are considered to be entities.

All entities have a series of properties that change, according to their interaction with the world and with other entities. These properties are presented in the list bellow.

List 4.5: Properties of entities

- **position** - a tuple of 3 floating point numbers in the 3 dimensional grid
- **rotation** - a floating point number between 0 and 1 that represents the deviation from the north direction
- **velocity** - a measure of how fast the block is moving (measured in blocks/tick)
- **terminal velocity** - applied only when falling
- **acceleration** - applied only when falling; different for each entity
- **health** - note that some entities may have infinite health

It is worth mentioning at this point that a tick is the 20^{th} part of a second, or, in other words, 20 ticks are equal to one second.

When falling, entities also experience drag forces. The final velocity of falling entities can be computed with the drag applied before (Equation 4.1) or after (Equation 4.2) the acceleration.

Equation 4.1: Drag applied before acceleration [5]

$$\text{finalVelocity} = \Big((\text{initialVelocity} - \text{acceleration}) \times (1 - \text{drag})^{\text{ticksPassed}}\Big) - \left(\text{acceleration} \times \frac{1 - (1 - \text{drag})^{\text{ticksPassed}}}{\text{drag}}\right)$$

Equation 4.2: Drag applied after acceleration [5]

$$\text{finalVelocity} = \Big(\text{initialVelocity} \times (1 - \text{drag})^{\text{ticksPassed}}\Big) - \left(\text{acceleration} \times \frac{1 - (1 - \text{drag})^{\text{ticksPassed}}}{\text{drag}} \times (1 - \text{drag})\right)$$

Note that *initialVelocity* and *finalVelocity* are measured in blocks/tick, acceleration is measured in blocks/tick^2, and drag is measured in 1/tick.

Mobs

The Minecraft Wiki website describes *mobs* as being "an AI-driven game entity resembling a living creature". The term *mob* is a shortened version of *mobile entity*, since all mobs have some sort of voluntary movement of their own. Also, on the website, it is specified that "all mobs can be attacked and hurt (from falling, attacked by a player or another mob, falling into the void, hit by an arrow, etc)". All types of mobs have a different AI, while only some of them drop certain items upon their death.

Remark 4.6: Mob spawning and despawning

Mobs spawn naturally in the world, based on their type and spawning conditions associated with it. They can also be spawned by the player, using the *spawn egg* items, mentioned in Subsection 4.2.1. Some mobs require the player to be within a certain amount of distance around them, otherwise they will despawn after a certain period of time.

The environment affects the mobs in the same way it affects the player. As such, they each have a certain type of AI system, with different behaviors and mechanics. Mobs usually wander around aimlessly. However, their advanced path-finding system allows them to navigate obstacles and to avoid walking off blocks high enough to cause fall damage (i.e., more than 4 blocks high).

Most mobs become aware of the player's presence up to 16 blocks away. Consequently, the player is able to hear the mobs that are as far as 16 blocks from them. Mobs cannot see or pass through solid blocks. They also can't see through transparent blocks. The 16 block distance is measured spherically around the player or mob.

As of *Minecraft* 1.19, there are four categories of mobs: passive, neutral, hostile, and bosses.

Passive mobs are harmless mobs that do not attempt to attack the player, even when provoked or attacked. Most of them can be bred or tamed. If they are attacked (either by the player or another mob), they will start running in random directions to avoid another attack. This behaviour stops after a certain amount of time. Figure B.3 shows all the passive mobs currently in the game.

Neutral mobs are passive by default toward the player. All of these mobs are provoked when the player (or another entity) attacks it first. Upon being attacked, these mobs will chase the attacker using the path-finding system. Figure B.4 shows all the neutral mobs currently in the game.

Hostile mobs are dangerous, aggressive mobs that always attack the player within their respective detection ranges. Some passive mobs such as chickens and skeleton horses can be hostile if ridden by a hostile mob on rare occasions. Figure B.5 shows all the hostile mobs currently in the game.

A subclass of hostile mobs are **boss mobs**. They are special hostile mobs that are distinctly more dangerous and tougher than other mobs. They do not spawn randomly, and are confronted intentionally. All of them also have a boss bar featuring their name and health. Boss mobs provide unique challenges but also equivalent rewards. Figure B.6 shows all the bosses currently in the game.

Remark 4.7: Mobs that hurt the player

In this paper, we are only interested in the neutral and hostile mobs, since they are the only ones that can deal damage to the player, thus being able to interact with the armor the player is wearing.

Block Entities

Another concept in *Minecraft* are the *block entities*. According to Minecraft Wiki, "a block entity (also known as tile entity) is extra data associated with a block, beyond

the finite set of block states associated with each block". Block entities should not be confused with entities.

Block entities allow the player to interact with the block to which it is attached. Most of the times, the block entity occupies the same physical space the the block that it is linked to. More details about block entities will be presented in Chapter 5.

Mechanics like crafting or inventory need a block entity in order for them to be implemented. Crafting is usually done within a GUI, but can also happen just by placing certain items inside of a block entity. Regardless of how it is achieved, in order to have access to these kind of mechanics, the corresponding block needs to have a block entity attached.

4.2.4. Actions in Minecraft

Minecraft allows the player to engage in certain actions throughout the gameplay. These actions are, as follows: crafting, smelting, brewing, enchanting, stone cutting, upgrading, cooking, exploring, farming, combat. Presenting all of them is beyond the scope of this paper. The focus will be shifted on the ones that are directly related to the specific of the mod.

Crafting

As mentioned previously, items and blocks can be crafted into other items and blocks, based on a *crafting recipe*. A *crafting recipe* is just like a regular, day to day recipe, that tells the player how to arrange certain items and blocks to obtain new ones. As such, a *crafting grid* is the space where the crafting takes place.

The first crafting grid that the player can access is the 2×2 one, inside their inventory. It was presented in Subsection 4.2.1. As Figure B.7 shows, to access the second crafting grid (i.e., the 3×3 one), the player needs to craft a *crafting table*. To do this, they need to fill the 4 slots inside their inventory's crafting area with any kind of wooden planks.

Note that this recipe only shows the usage of *birch planks*. The player can use any kind of wooden planks, even in combination with one another. Upon placing the final plank in the slots, the crafting table appears in the result slot. When the player takes it out of that slot, the 4 wooden planks are consumed. This is the crafting mechanic of the game.

Remark 4.8: Crafting procedure

This process is instantaneous. As soon as the crafting recipe is complete, the result will appear in the designated slot.

The 3×3 crafting grid is where almost all of the items are crafted. Some crafting recipes are *shaped*, which means that the position of each item or block in the crafting recipe is important, while other crafting recipes are *shapeless*. Generally, any crafting recipe that can be done in the 2×2 grid is shapeless.

Figure B.8 shows how the *furnace* block is crafted. The process is similar to the one presented previously (for the 2×2) grid.

It is worth noting that, upon opening the crafting table's interface (right click on the crafting table block placed in the world), the player's inventory (together with the 9

slots from the hotbar, not seen in the figure) is also attached to the crafting table's area. This is standard practice for all block entities that have an interface, since the player needs to be able to use the items in their inventory while operating inside the interface.

Combat

Combat is the action in which the player engages in a fight with either another player (i.e., Player vs. Player combat, or PvP), in case the gameplay takes place on a server or with an enemy (i.e., Player vs. Enemy combat, or PvE). The enemy can be any mob presented in Subsection 4.2.3, including the passive mobs (even if they will not attack back).

During combat, each participant uses a weapon to attack and (ideally) is protected by the armor equipped in the appropriate slots. Weapons and armor have different attributes, depending on the material they are made from. More details about how the material used while crafting impacts the weapon or armor piece will be presented in Subsection 4.2.6.

4.2.5. Health, hunger and armor

Health, hunger and armor, together with experience and effects are all part of a group called *player statistics* or *stats*, for short. Health and hunger mechanics have already been presented briefly.

The player has a total of 20 HP, represented visually by 10 red hearts. Each half of a heart represent 1 HP. Upon taking damage, a certain value is subtracted from the player's remaining HP. In case the HP value drops to 0, the player dies (unless other events prevent him from dying). The player can restore their health by eating food (so that the saturation bar is full) or by drinking healing potions (such as *regeneration* or *instant health*).

Remark 4.9: Saturation points restoration

Different kinds of food restore different amounts of saturation points.

On top of the health bar, the *armor points* (AP) are shown, if the player is wearing any piece of armor. The player can have a maximum of 20 AP, which is obtained by wearing a full set of diamond armor or a full set of netherite armor. Along with armor points, some types of armor also provide *armor toughness*, which is another factor taken into account when the player is attacked. Figure 4.5 shows how the armor bar looks with a full set of iron armor equipped.

Figure 4.5: Armor bar

Figure 4.5: Armor bar

There are certain formulae used when computing how much HP will be deducted from the player's health bar. These formulae are presented bellow. To be noted that w_d represents *weapon damage*.

Equation 4.3: Computation of total damage [5]

$$\mathsf{totalDamage} = \mathsf{damage} \times \left(1 - \frac{\max\left(\frac{\mathsf{defensePoints}}{5}, \mathsf{defensePoints} - \frac{4 \times \mathsf{damage}}{\mathsf{toughness}+8}\right)}{25}\right)$$

Equation 4.4: Computation of damage [5]

$$\mathsf{damage} = \mathsf{w_d} \times \left(1 - \frac{\min\left(20, \max\left(\frac{\mathsf{defense\ points}}{5}, \mathsf{defense\ points} - \frac{4 \times \mathsf{w_d}}{\mathsf{toughness}+8}\right)\right)}{25}\right)$$

From the second formula, we can easily see that the total amount of armor points taken into consideration is capped at 20. This means that, even through commands, the player will not be able to receive more than 4% damage reduction per armor point. This brings the total to 80% armor reduction, without any armor toughness. In the same time, it can be noted that a smaller percentage of incoming damage is reduced, the higher the incoming damage value. From here, it can be deduced that armor points protect the player more from weaker attacks.

The main difference between armor points and armor toughness is that armor points reduce the damage taken, whereas armor toughness increases the efficiency with which the damage is reduced. From here, it can be deduced that armor toughness increases the percentage of incoming damage that is reduced from stronger attacks.

Armor damage reduction can be easily understood from the graph in Figure B.9. Additionally, the received damage with armor is presented in Figure 4.6. From this, it can be deduced that, with a full netherite armor set, the player can sustain around 38 HP worth of damage at once, without dying.

Lastly, not all types of damage can be mitigated by armor points. Table 4.2 shows which types of damage is mitigated by the armor and which is not. The damage that is not mitigated also does not take any durability from the armor.

Figure 4.6: Received damage with armor

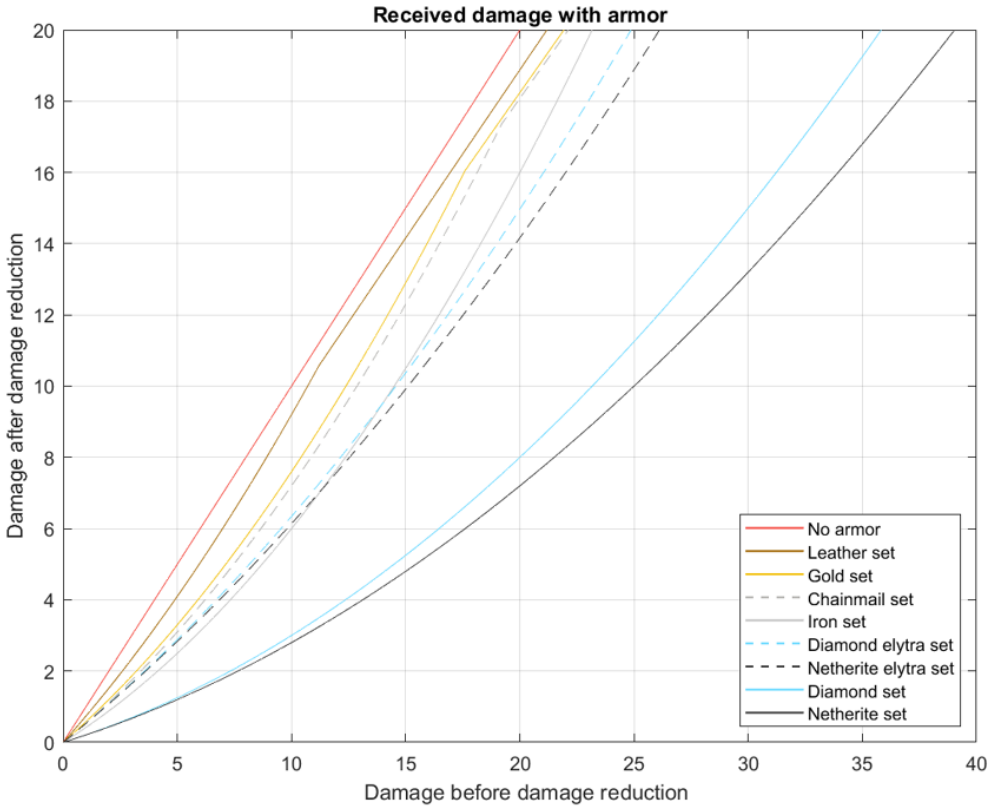

Figure 4.6: Received damage with armor

Table 4.2: Damage mitigation based on source

Mitigated	Not mitigated
Melee attacks from mobs and players	Ranged sonic attacks from wardens
Getting hit with a projectile	Fall damage
Getting hit with a fireball	Ongoing damage from being on fire
Laser beam of a guardian	Suffocating inside a block
Lightning	Drowning in water
Touching a damaging block	Starvation
Explosions	Colliding with a block while flying with elytra
Getting hit with a falling anvil	Touching a campfire block
Getting hit with a snowball	Falling into the void

4.2.6. Material types and tiers

In *Minecraft*, we distinguish between six material types for tools (wood, stone, gold, iron, diamond and netherite) and six material types for armor (leather, chainmail, gold, iron, diamond and netherite). For tools, each material has an associated tier. Tier

0 corresponds to wood, tier 1 to stone, tier 2 to iron and gold and tier 3 corresponds to diamond and netherite. With each tier, the tools are stronger and more durable.

The tier system works as follows:

List 4.6: Tiers in Minecraft

- any kind of stone needs to be mined with a T0 (i.e., tier 0) pickaxe or up
- coal, iron, copper and lapis lazuli ores need to be mined with a T1 pickaxe or up
- gold, redstone, diamond and emerald need to be mined with a T2 pickaxe or up
- obsidian and other blocks crafted from it need to be mined with a T3 pickaxe.

In case a certain block is mined with a tool of the wrong tier or type, that block will not drop the corresponding item upon destruction.

For armor, each material type provides the armor piece with certain values for the basic properties of the piece. The material type also influences the durability of the armor piece.

Remark 4.10: Armor crafting

Each of the four armor pieces (helmet, chestplate, leggings, and boots) can be crafted from only one material.

Table 4.3 shows how each material types influences the attributes of the final armor piece. The abbreviations stand for: durability multiplier (DM), slot protections (SP), enchantment value (EV), toughness (T), and knockback resistance (KR). Slot protections are presented in the following order: boots, leggings, chestplate, and helmet.

Table 4.3: Material types property values

Material	DM	SP	EV	T	KR
Leather	5	1, 2, 3, 1	15	0.0	0.0
Chain	15	1, 5, 5, 2	12	0.0	0.0
Iron	15	2, 5, 6, 2	9	0.0	0.0
Gold	7	1, 3, 5, 2	25	0.0	0.0
Diamond	33	3, 6, 8, 3	10	2.0	0.0
Netherite	37	3, 6, 8, 3	15	3.0	0.1

Armor toughness is computed per piece of armor. As such, a full diamond set will provide the player with 8 armor toughness, whereas a full netherite set will provide the player with 12 armor toughness. Knockback resistance is computed in percents, therefore, a full set of netherite armor will provide the player with 40% knockback reduction.

4.3. Communication in Minecraft

Minecraft splits the execution of methods on two different sides: *client* and *server*. This separation comes with the need of providing communication means between the two

sides. It should be specified that, in this context, we consider a *logical* client and *logical* server. It should not be confused with the server on which multiple players can join the same world.

4.3.1. Sides

In order to solve the ambiguity, the *Minecraft Forge* documentation distinguishes between two types of clients and two types of servers. This is presented in the list bellow.

List 4.7: Disambiguation between client and server

- **Physical client** - the entire program that runs when the player runs *Minecraft* (all threads, processes, and services)
- **Physical server** - the entire program that runs when the server instance of the game is launched (does not have a playable interface)
- **Logical client** - accepts input from the player, relays it to the server and also handles the rendering (the interface to the player)
- **Logical server** - runs the game logic (mob spawning, weather, *inventory updates*, and other game mechanics).

It is worth mentioning that the physical server can only host the logical server, but the physical client can host both logical server and logical client. In other words, Minecraft client hosts its own integrated server for singleplayer and LAN sessions.

Remark 4.11: Approaches to communication

The current mod presented in this paper targets both the logical server and logical client. As such, it can be played in singleplayer or multiplayer.

4.3.2. Communication

There is only one correct way to exchange data between the logical client and the logical server: by exchanging packets. These types of packets are sent between logical sides only, similar to the TCP protocol. All data sent over the network (except for VarInt and VarLong) is big-endian. Without compression, the format of the packet is presented in Table 4.4.

Table 4.4: Packet format

Field name	Field type
Length	VarInt
Packet ID	VarInt
Data	Byte Array

Based on the packet ID, the data section of the packet is partitioned in multiple fields. Table 4.5 shows the format of the *data* field for the *Spawn player* packet. This packet is sent by the server when a player comes into visible range, not when a player joins. It must be sent after the *Player info* packet.

Table 4.5: Packet format for *Spawn player*

Packet ID	State	Bound to	Field name	Field type
0x02	Play	Client	Entity ID	VarInt
			Player UUID	UUID
			X	Double
			Y	Double
			Z	Double
			Yaw	Angle
			Pitch	Angle

Mods can also add packets to transfer information between the two logical sides. This is useful when the mod adds a new mob or custom crafting recipes. More details about this will be presented in Chapter 5.

4.4. Creating a new mod - *Armor Tinkers*

As mentioned before, the mod described in this paper is called *Armor Tinkers*. It aims to add new items, blocks, and recipes into the game, while also integrating two other existing mods with itself. Among the items added, there are new materials and armor. In the same manner, some of the blocks added by this mod also have their own block entities attached to them.

Figure 4.7: Block diagram of the system

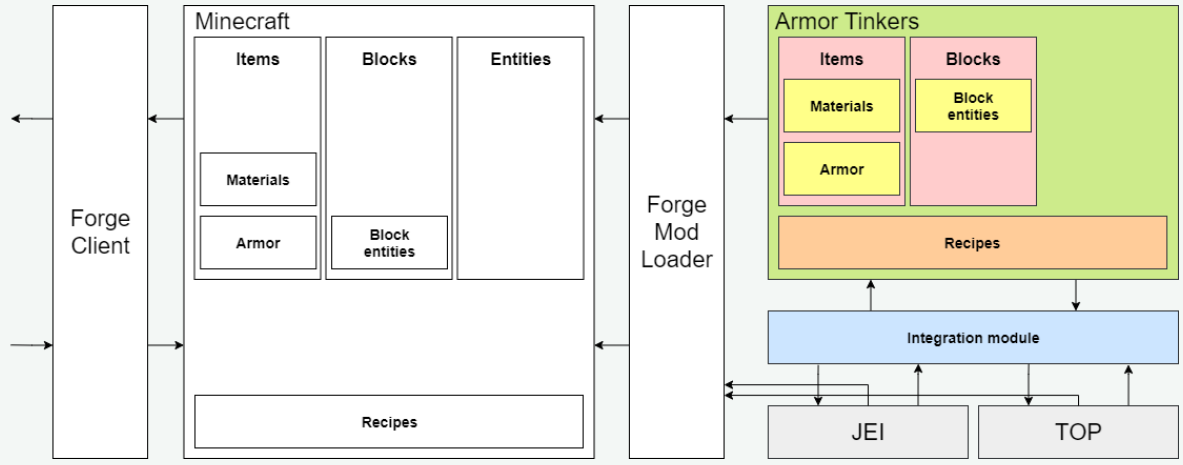

Figure 4.7: Block diagram of the system

In Figure 4.7, the colored components are the ones that will be implemented by me. They add new items (together with materials and armor), blocks (together with block entities), and recipes. The *Forge Client* will receive the input from the player and will display the result of the player's interaction with the system. The *Forge Mod Loader* is the one that injects the game's code with the code from the three mods added to the game. While both *JEI* (Just Enough Items) and *TOP* (The One Probe) can

communicate with *Minecraft* via the the *Forge Mod Loader*, an *Integration module* is necessary to provide communication channels between the mod developed by me and the two other mods mentioned previously.

4.5. Use cases

Starting from the functional requirements presented in Subsection 2.3.1, the paper will present next three use cases, with the aim of showcasing how these functional requirements will be fulfilled.

4.5.1. Crafting a *Pattern Station*

The following use case aims to fulfill the *Continuity* functional requirement (FR-AT01), by showcasing how a *Pattern Station* can be crafted from items available in the *Vanilla* game. The list below shows the preconditions for this use case and Figure 4.8 shows the diagram.

List 4.8: Preconditions

- player has a crafting table placed within reach
- player has 4 oak logs
- player has 3 oak planks and 2 planks of any other kind
- player has 2 stick

As it can be seen in the preconditions, all the items necessary to craft the *Pattern Station* are available in *Vanilla Minecraft*, thus fulfilling requirement FR-AT01. The player has to first craft a *Blank Wooden Pattern* and then use that pattern to craft the *Pattern Station*. In Figure 4.8, the letter 'P' stands for *player* and the letter 'S' stands for *system*.

Figure 4.8: Use case diagram for crafting a pattern station

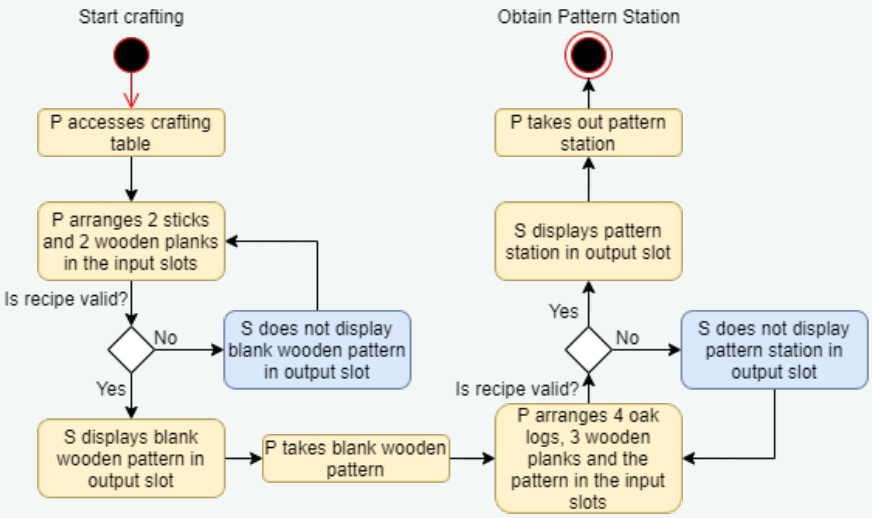

Figure 4.8: Use case diagram for crafting a pattern station

The main (happy) flow of the use case is highlighted in yellow, while the alternate flows are highlighted in blue. It can be seen in the figure that, in case the player does

not place the items in the crafting table in the appropriate configuration, the system will not place the resulting item in the output slot.

4.5.2. Obtaining *Magnesium*

The following use case aims to fulfill FR-AT02 (*Provide new materials* functional requirement), by showcasing how a *Magnesium* chunk can be obtained. The list bellow shows the preconditions for this use case and Figure 4.9 shows the diagram.

List 4.9: Preconditions

- player has a stone pickaxe (or higher tier)

Figure 4.9: Use case diagram for obtaining magnesium chunk

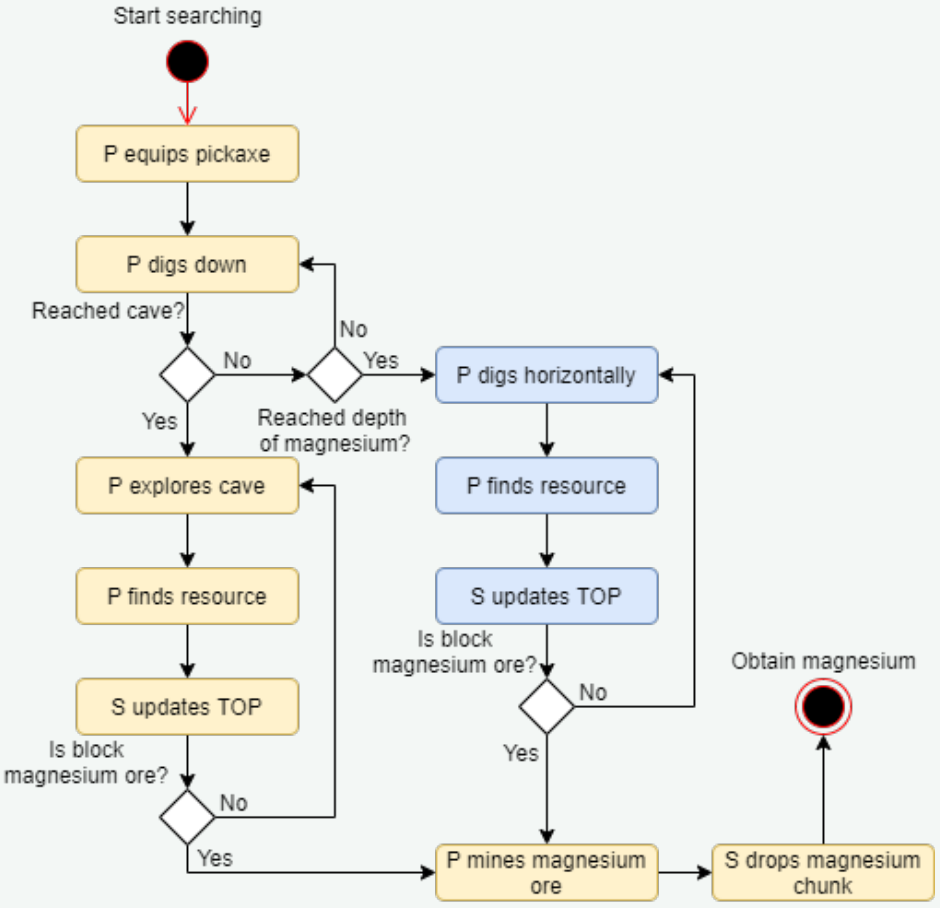

Figure 4.9: Use case diagram for obtaining magnesium chunk

The main (happy) flow of the use case is highlighted in yellow, while the alternate flow is highlighted in blue. The alternate flow is the result of *Minecraft*'s world generation strategy. Ores can generate in caves or completely surrounded by stone. All the new materials added by this mod can be found at different levels and with different rarities. The values for these attributes are tuned to match the ones for the ores that already exist in the *Vanilla* game. For example, *tin* and *magnesium* will generate at about the same

levels as *copper* and *iron*, while also providing the same amount of protections when used to craft an armor.

4.5.3. Crafting a *Chestplate*

The following use case aims to fulfill the FR-AT03 and FR-AT04 (*Provide new armor* and *Provide new crafting mechanics* functional requirements), by showcasing how a *chestplate* is crafted in this mod. The list bellow shows the preconditions for this use case and Figure 4.10 shows the diagram.

List 4.10: Preconditions

- player has an armor forge (of the appropriate tier for the materials selected) placed within reach
- 2 shoulder plates of any material
- 1 back plate of any material
- 1 front plate of any material
- 1 wrist band of any material
- 1 hide vest

The main (happy) flow of the use case is highlighted in yellow, while the alternate flow is highlighted in blue. There is a noticeable difference between this crafting process and the one presented in Figure 4.8, that is the system simulates the processing of resources, instead of displaying the result as soon as the recipe is validated. The result of this crafting process is a *chestplate*. In this manner, both FR-AT03 and FR-AT04 are satisfied.

Figure 4.10: Use case diagram for obtaining a chestplate

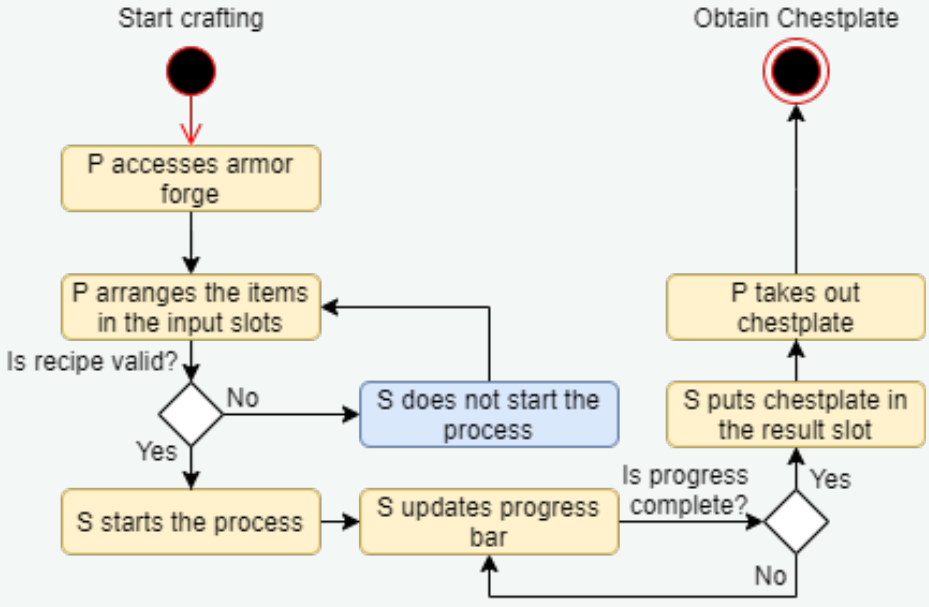

Figure 4.10: Use case diagram for obtaining a chestplate

4.6. Additions to the game

From the three use cases presented in Section 4.5, we can further analyze what are the items, blocks and block entities that need to be added to the game, as well as the recipes and their integration with *JEI*.

4.6.1. Items

As it was seen in Figure 4.7, the mod aims to add items to the game. Among the simple items that will be implemented, there are also *armor* items and *materials*. It should be noted that also any block added to the game will have a corresponding item, but these won't be presented as items. Instead, just the blocks will be mentioned. The list bellow presents some of the important items that the mod will add (other than armor and new materials).

List 4.11: Some important items

- **stone hammer** and **iron hammer** - will be used for armor part tinkering
- **resin** - will be used as a *fuel* item and in crafting recipes
- **blank wooden pattern** and **blank golden pattern** - will be used for pattern scribbing
- **hide cap** - will be used for crafting the armor
- **hide vest** - will be used for crafting the armor
- **hide pants** - will be used for crafting the armor
- **hide socks** - will be used for crafting the armor

New materials

Remark 4.12: Materials added by the mod

The new materials added by the mod are: zinc, lead, silver, tin, magnesium, brass, bronze, aluminium, vibranium and adamantium.

Along with the materials listed above, the mod will also add means to obtain them. As such, some of them will be available as ores that the player can mine, while the others will be obtained through crafting. At the same time, for all the materials listed above there are several types of items that should be added, such as: ingots, nuggets, blocks, ore blocks and ore chunks.

New armor

Given the fact that the name of the mod is *Armor Tinkers* and the main objective of the mod is to improve the existing armor system of *Minecraft* is is clear that it needs to add new types of armor as well. However, given the fact that the armor can be made out of multiple materials, the mod should add armor pieces for every combination of them.

For example, a helmet can be made out of two parts, a plate on the left side and a plate on the right side. There will be 6 types of soft materials (that already exist in the game, but are not used for armor crafting) and 14 types of hard materials (some of

which already exist in the game, while the others were mentioned when specifying what materials will be added).

Remark 4.13: Armor added to the game

After performing the computation for the example mentioned above, it can be seen that the mod will add a total of $6^2 + 14^2 = 232$ helmets. Similiarly, we can compute the number of chestplates, leggings and boots added, based on the numbers mentioned in the example.

4.6.2. Blocks

Besides the block related to the materials presented in the previous subsection (ore blocks and blocks of ingots), the mod should also add the blocks listed bellow.

List 4.12: Important blocks added

- **maple log**, **maple wood** and **maple planks** - will be used as part of the new maple tree added (it will drop the resin)
- **pattern station** - the station in which patterns are crafted
- **armor part station** - the station in which armor parts are crafted
- **armor forge** - should come in two tiers and is station in which the actual armor is crafted

Remark 4.14: Custom 3D models

The four stations mentioned in the previous list will have custom 3D models (i.e., they will NOT look like a regular cube).

4.6.3. Block entities

Each of the stations mentioned in the previous subsection (i.e., the pattern station, the armor part station and the two tiers of the armor forge) will have their own block entities designed and implemented, in order to allow for the implementation of the new crafting mechanics that will be presented in Section 4.8.

4.7. Automation for data generation

All of the new items and blocks that are planned to be added will need configuration files that describe their model, block state (for blocks only), their recipe (or recipes) and their loot table (for blocks only). Thus, it is not hard to observe that the total number of *.json* files needed will probably exceed 3000.

It is simply not feasible to write all those configuration files manually. Luckily, the *Forge API* provides the developer with implementation options for automatic configuration files generators, directly from Java code. Another alternative worth exploring is the use of python scripts to generate the *.json* files based on some specifications that will be deduced when the classes are implemented.

Remark 4.15: Data generation options

The two options for data generation are: built in functions of the *Forge API* library and Python [14] scripts that will be designed and implemented by me.

4.8. New crafting mechanics

As it was presented in Subsection 4.2.4, *crafting* is one of the fundamental actions in *Minecraft*. Through crafting, the player can obtain different items with which they can interact. As such, most mods add crafting recipes for the items and blocks they contain, in order to provide the player with a way of obtaining them. The strategy is to either use the available crafting recipe types, such as the ones used in the *crafting table* or the *furnace*, or to implement custom crafting recipe types.

4.8.1. Core concept

As stated before, blocks that have a block entity attached to them can provide the player with the means of accessing a new type of crafting strategy. All block entities go through an action called *ticking*, meaning that they perform a certain operation every tick. Through this capability, new crafting mechanics can be implemented.

Remark 4.16: New recipe types

For the mod being described in this paper, there exists the need for 3 new crafting recipe types. They are, as follows: *pattern scribbing*, *armor part tinkering*, and *armor forging*.

These three types of crafting mechanics follow a certain algorithm that is presented in Subsection 4.8.2. This algorithm is then adjusted to work with each type of recipe presented previously.

4.8.2. Crafting algorithm

To be noted that for the *crafting table*, the crafting process is instantaneous and the items are removed from the input slots only when the resulting item is taken out of the result slot. However, for the new recipe types, the crafting process is not instantaneous, meaning that from the moment the recipe is validated in the input slots to the moment the result appears in the output slot, there is a time delay. This aims to increase the game immersion. Additionally, items are removed from the input slots when the result appears in the output slot, not when the resulting item is removed.

Listing 4.1: Crafting algorithm

```
1: if recipeIdentified() then
2:     incrementProgress()
3:     if currentProgress > progressRequieredForItem then
4:         putItemInResultSlot()
5:         extractInputItems()
6:     end if
7: else
```

```
8:      resetProgress()
9:  end if
```

The algorithm in Listing 4.1 shows how the general process of crafting should take place when implementing the crafting method. This algorithm is called every tick, thus incrementing the *currentProgress*, when a recipe is found for the current configuration of items in the input slots. Recalling that a tick is 1/20 seconds, the total amount of time needed for an item to be crafted can be easily specified by assigning a number to the *progressRequiredByItem* variable. When *currentProgress* reaches *progressRequieredByItem*, the result is than placed in the output slot and the quantities of the input items are decreased. In case the items in the input slots are removed, the recipe becomes invalid, thus needing the *currentProgress* to be reset.

4.8.3. Client-Server communication

Given the fact that the actual crafting process is an action that happens on the *logical server*, another aspect that needs to be analyzed is the communication between the logical client and the logical server.

The interface that is displayed when the player interacts with a block entity is handled by the logical client. Any items placed in the input slots are sent to the logical server, to check if there is any valid recipe for that configuration of items. If there is, the response is sent back to the client and the crafting process can begin. After this, there should be an exchange between the client and the server with the progress of the process (in order for the client to update the GUI). The communication should end with the server returning the result item and the client displaying it in the output slot. Figure 4.11 shows how this communication should take place in the context of a crafting recipe.

Figure 4.11: Client-Server communication in crafting recipe

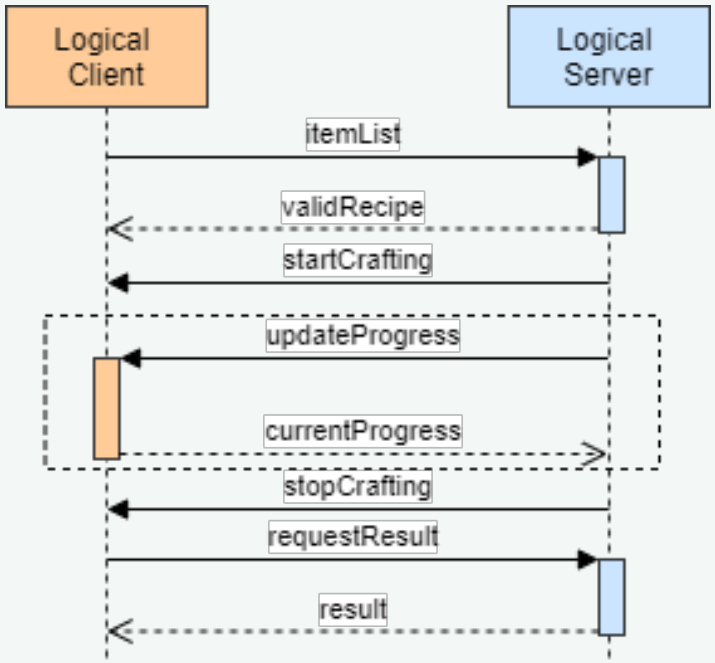

Figure 4.11: Client-Server communication in crafting recipe

4.9. Integrating with existing mods

Integration is a critical part of mod developing. In *Minecraft*, most players opt to play with multiple mods at once. This is how modpacks are born. A *modpack* is a collection of mods configured in such a way that they work together, without any conflicts.

Most mods work together (i.e., do not cause any crashes) straight out of the box. In this case, the process is as simple as putting the *.jar* files in the same folder and loading it all at once when the game is launched. Other times, two mods may conflict with each other, thus analysis is required to determine if the mods are compatible or not.

Even if two mods added to the same modpack do not conflict with each other, there might be the case of discontinuity. For example, suppose there are two mods that both add *magnesium* as a resource to the game. Each of them has their own model and texture for all items and blocks related to magnesium. Thus, in the world there will be two ores that look different, but provide the player with the same resource. This is usually avoided by using *tags* instead of item IDs.

Tags work by registering all items of the same type (from different mods) under the same *tag*. When that item is needed in a crafting recipe or when it is to be generated in the world, the tag name is used instead of the item's name.

The mod presented in this paper provides the player with out-of-the-box integration with *The One Probe* mod and *Just Enough Items* mod. Although the integration capabilities are provided, the two mods specified previously do not need to be in the modpack for it to run without crashing.

4.9.1. The One Probe

The One Probe mod (or TOP) is a *utility* mod. It does not add any new resources or mechanics to the game. Instead, it provides the player with a box drawn on the screen that describes the block the player is currently looking at. It is a successor to *WAILA* (short for *What Am I Looking At*).

As it can be seen in Figure B.10, the mod presents dynamically the name of the block the player is looking towards, the mod which adds it (in this case, *Minecraft*), the tool and tier required to break it and if it is currently harvestable (it can be broken correctly, as to drop the respective item).

Additional functionalities of this mod are showing the inventory contents of blocks that have block entities attached to them or other different attributes added by other mods.

The mod currently presented in this paper provides integration for *TOP*, in order for the player to be able to easily distinguish between the ores found in the caves that this mod adds.

4.9.2. Just Enough Items

Just Enough Items (or JEI) is also a *utility* mod. It provides a glossary for the player with all the items registered to the current game (both from *Vanilla* and from other mods). The purpose of this glossary is to allow the player to quickly check the recipes of various items and blocks, in order to craft them. Additionally, it also provides functionality for showing the player in which recipes a certain item is used.

Recipes of items are displayed by hovering over them and pressing the 'R' key (in the default configuration). In order for the player to check in which recipes a certain

item is used, they have to hover that specific item and press the 'U' key (in the default configuration).

Figure B.11 shows a *brewing* recipe for the *speed* potion. On the right side, the glossary of items can be seen. It is worth mentioning at this point that items in this glossary cannot be taken directly into the player's inventory. In other words, this mod does not encourage *cheating* by acquiring items without crafting them.

Additional functionalities of this mode are auto-filling recipes for the player, in case that they checked a recipe with an appropriate crafting interface opened previously, or quick saving items in order to check their recipes later (by pressing the 'A' key while hovering over the item). The glossary also has a search box.

The mod currently presented in this paper provides integration for *JEI* to allow the player to check the custom crafting recipes that it adds. This recipes are both for the already existing crafting table and also for the three new crafting recipe types.

Figure 4.12: Integration with *JEI* and *TOP*

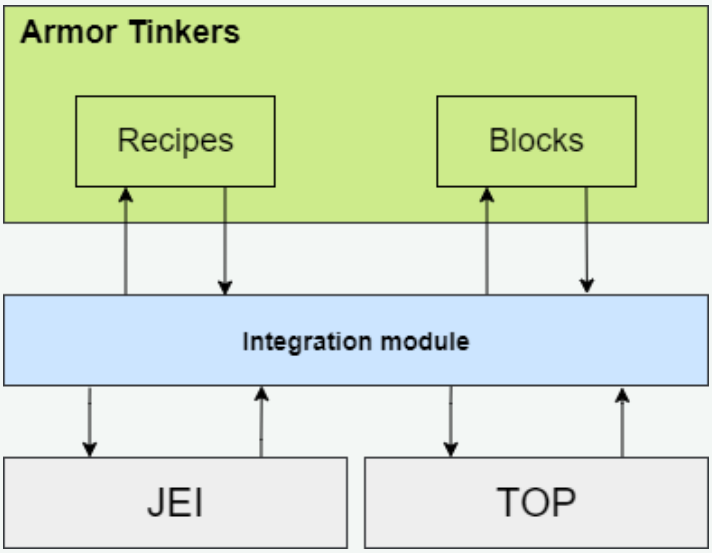

Figure 4.12: Integration with *JEI* and *TOP*

As it can be seen in Figure 4.12, *JEI* interacts with the recipes added by this mod, while *TOP* interacts with the blocks. This interaction will be handled by the *Integration module* marked on the figure.

Chapter 5. Detailed Design and Implementation

5.1. Chapter overview

This chapter will present the design of each important component of the mod as well as its implementation. Relevant code and images will mainly be listed in Appendix A and Appendix B.

5.1.1. General architecture

To reiterate, the name of the mod presented in this paper is *Armor Tinkers*. As such, by convention, the name of the main class should reflect this. *Forge Mod Loader* works by exposing event buses, registries and other classes to this main class and overriding the methods in the game's application. Figure 5.1 shows the block diagram for the whole project. Is is worth noting that it only presents the primary classes used to implement the project. Secondary classes and packages will be presented in their relevant sections.

Figure 5.1: *Armor Tinkers* block diagram with the main components

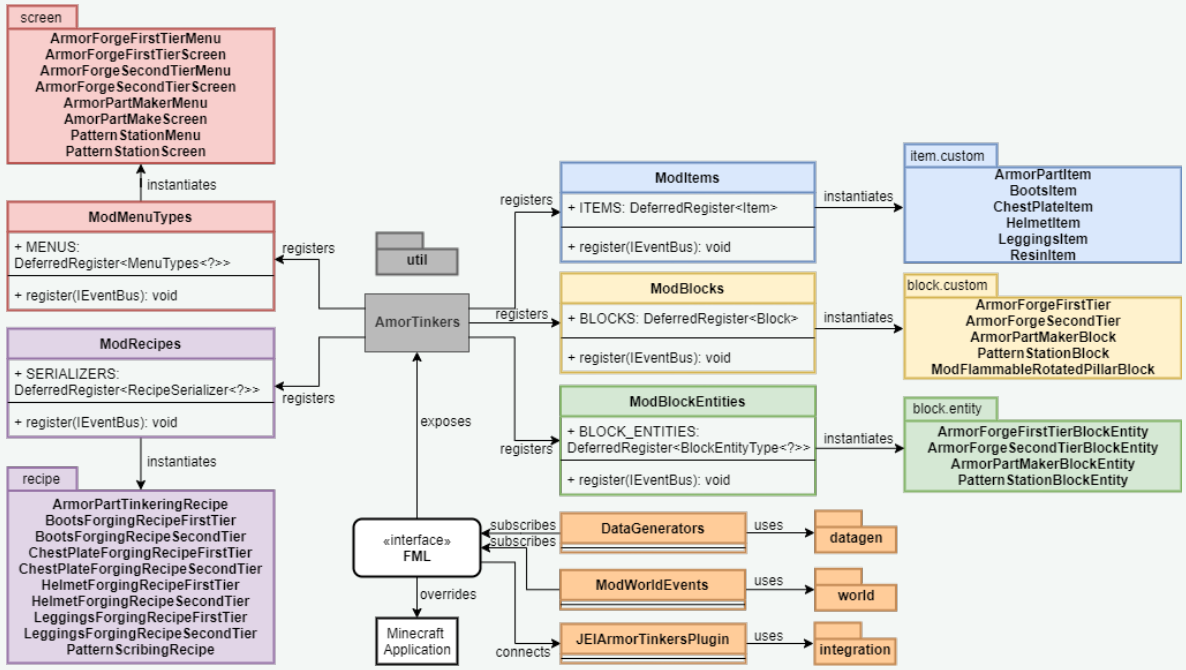

Figure 5.1: *Armor Tinkers* block diagram with the main components

As it can be seen in Figure 5.1, there are five registers that are accessed by the mod in order to register new *menu types* (in red), *recipes* (in purple), *items* (in blue),

blocks (in yellow), and *block entities* (in green). These registers then operate (instantiate) classes in their respective packages. Two event subscribers (*data generators* and *world events*, presented in orange) also access the application via the **Forge Mod Loader (FML)** interface. The plugin for integrating the mod with *JEI* is connected to the mod via the same interface. Lastly, the *util* package (in dark grey) contains classes that are used throughout the project.

5.1.2. Package description

All the packages of this project can be seen in Figure 5.1. Some of the classes in them have not been listed, but will be mentioned when the appropriate functionality is described.

List 5.1: Packages

- **block** package contains all the custom blocks added by the mod as well as all the block entities that are attached to some of them, the exception being *ModFlammableRotatedPillarBlock* class
- **datagen** package contains all the classes necessary to generate the any *assets* or *data* files automatically
- **integration** package contains all the classes used to integrate the mod with *JEI*
- **item** package contains all the custom items that are added by the mod, as well as the *ModCreativeModTab* class, that adds new tabs in the *creative mode* menu
- **recipe** package contains all the classes used to add new recipe types to the game
- **screen** package contains all the classes that are necessary to add new screens and menu types to the game, as well as *slot* classes that allow to implement special behaviours for the slots in these menus
- **util** package contains classes that have constants, enumerations or any other data that is static and final, as well as a class that adds new *tags* to the game
- **world** package contains all the classes necessary to generate new ore types into the game and the new tree type added by the mod

5.2. Resource package structure

Given the fact that this mod adds new features to the game, it is also mandatory to add textures and models for all the new items, blocks, crafting interfaces and so on. This should be done in order to have a good product that integrates well with the *Vanilla* game.

As such, another package that is separate from the source code is the *resources* package. It contains all the files necessary to specify the models for items and blocks or the images used to render the textures. This package has inside of it two folders, *assets* and *data*.

Remark 5.1: Namespaces

All of the files in the *assets* folder are under the same namespace, *armortinkers* (i.e., the namespace generated by the mod), while the files in the *data* folder can be under the following namespaces: *armortinkers*, *forge* or *minecraft*. The name space is used in conjunction with the id of blocks and items in order to uniquely identify them.

5.2.1. The *assets* folder

The *assets* folder for the mod presented in this paper is furthermore divided into 4 packages. They are listed in the list bellow, together with the types of files found inside them

List 5.2: Assets

- **blockstates** folder contains the block state specification for every block added in the game (i.e., the states in which the block can be found in the world and what model should be loaded for each state)
- **lang** folder contains a single file, namely *en_us.json* with the translation for each id added by the mod
- **models** folder contains all the models linked to items and blocks, which will then load the correct textures
- **textures** folder contains all the images (*.png* files) that are linked to every graphical representation of the features added by the mod

5.2.2. The *data* folder

As stated before, the data folder contains three different namespaces, which are used for specifying aspects like *recipes*, *loot tables*, *tags* and *advancements*. These are *.json* files.

List 5.3: Namespaces in the *data* folder

- **armortinkers** - any file in this folder is related to anything that the mod adds
- **forge** - any file in this folder is related to functionalities that the *Forge* API provides (used for configuration of other mods)
- **minecraft** -any file in this folder is related to functionalities currently existing in the *Vanilla* game (used to configure item, blocks, and so on, that already exist in the game)

Remark 5.2: Textures and models

Except for the models of armor that is worn on the player, any other texture file or model file was created by the author of this paper. This was done with the use of the software mentioned in Section 1.3. The files *hide_layer_1.png* and *hide_layer_2.png* were obtained from [15].

5.3. Design and implementation strategies

In order to keep the mode as close to the original *Minecraft* experience as possible, all textures were designed based on materials or colors that already exist in the game.

From the source code's perspective, all custom items, blocks, block entities, menu types and recipe types have been designed and implemented based on the classes already available in *Minecraft*. For example, the *HelmetItem* class extends the *Item* class by adding additional fields and overriding the methods available in that class. This can be seen throughout this chapter, as each custom class will be presented together with the class it extends. The reason for this approach is that the return type of methods or the type of the parameters are mostly specified as generics.

As it was previously mentioned, the namespace for this mode (and also the mod ID) is *armortinkers*. This mod ID is declared in the main class, as a constant and it's used any time a method requires the namespace in order to register or load a feature.

5.4. Registries

Figure 5.2: Armor Tinkers and Minecraft registries

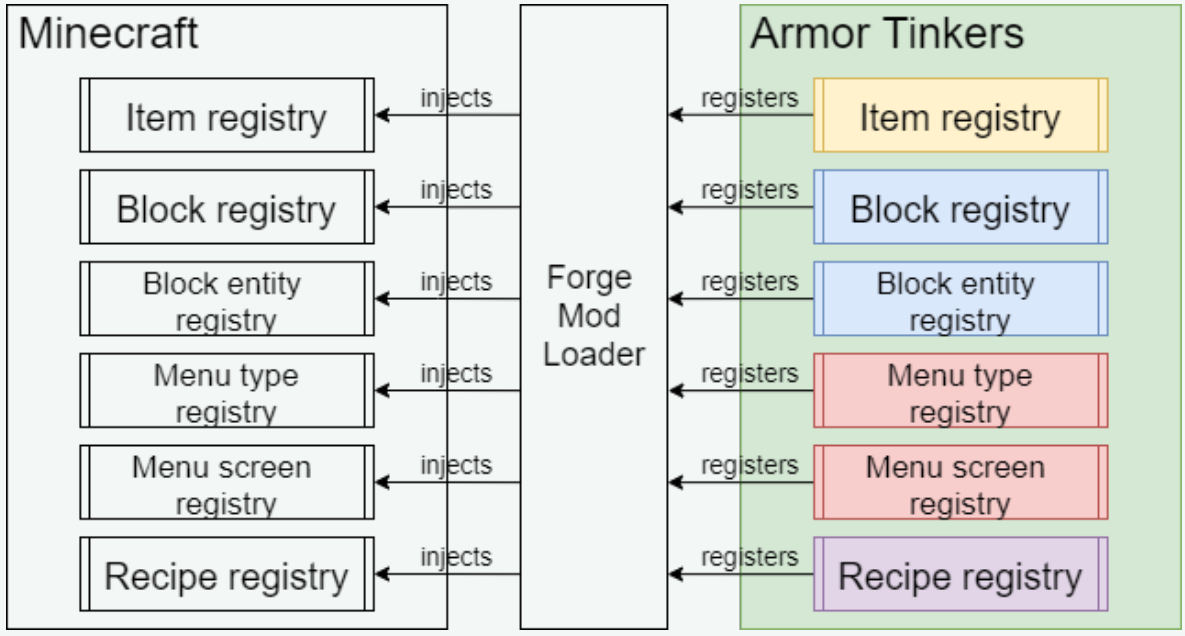

Figure 5.2: Armor Tinkers and Minecraft registries

Any new addition to the game needs to be registered in order for the *Forge Mod Loader* to load it at launch time. As such, *deferred registers* have been implemented that contain *registry objects* (which use generic types [16]). This section will present how simple items and blocks were added to the game. We distinguish between two types of registration: registering items to the deferred register and registering the whole register to the game, on the *event bus*. Figure 5.2 shows how *FML* registers all the mod registries and injects them into *Minecraft*. To be noted that the *Armor Tinkers* mod adds its own registries that are then injected into the existing registries of *Minecraft*.

Before moving forward, it needs to be specified that any ID allows only lower case letters, underscores and digits. Any other type of character will cause the game to crash. Additionally, all IDs are unique.

5.4.1. Registering an item

A simple item implies that the constructor of the *Item* class is called directly when adding a new entry in the register. To exemplify the whole process of how a simple item is added to the game, this subsection will present the design and implementation of the *Stone Hammer* item.

To begin with, Listing 5.1 presents how the item register is instantiated. The first parameter of the `create()` function is the *forge registry* that needs to be wrapped and the second one is the mod id (specified previously as `armortinkers`).

Listing 5.1: ModItems class

```
1 public static final DeferredRegister<Item> ITEMS =
2     DeferredRegister.create(ForgeRegistries.ITEMS,
3                             ArmorTinkers.MOD_ID);
```

After initializing the deferred register, any new item (be it simple or custom) is added to the game by adding it to the register. The deferred register behaves similar to a list, with the `add()` method being replaced by the `register()` method.

It can be seen in Listing 5.2 that the `register()` method takes in two parameters: the ID of the item being added and a supplier for the new entry, which should return a new instance every time is being called. The name of the item will automatically be prefixed with the mod ID.

Listing 5.2: ModItems class

```
1 public static final RegistryObject<Item> STONE_HAMMER =
2     ITEMS.register("stone_hammer",
3         () -> new Item(new Item.Properties()
4         .tab(ModCreativeModeTab.ARMOR_TINKERS_TAB)
5         .setNoRepair()
6         .durability(16)));
```

On line 3, the supplier gets a new instance of the `Item` class. The item properties are specified in a manner similar to the builder pattern [17]. Any attribute of the item that is not specified at this point will be given the default value in the `Item` class.

Remark 5.3: Declaration

Any object (item, block, block entity, recipe or menu) added to the game needs to be declared as both `static` and `final`. Thus, after declaration, no attributes of the object can be modified.

The attributes that need to be set for the *Stone Hammer* are the *creative mode tab* in which it will be displayed (in this case, it is a custom tab, added by the mod), the repair material (in this case, none) and the *durability* (in this case, 16). Another method

that is not specifically called here is the `stacksTo()` method. This is because when an item has a durability value, the `maxStackSize` is automatically set to 1.

Remark 5.4: `Item` vs `ItemStack`

The reason why any object added to *Minecraft* needs to be declared as both `static` and `final` is that it will be used as a template. What the player sees and uses in the inventory and world is the `ItemStack`, an instance of the `Item` class. It is similar to the *Singleton* design pattern [17]. Any modification brought to the `Item` will be visible on all the `ItemStacks` that instantiate it. In other words, within the game, there is only one *Stone Hammer* item and many *Stone Hammer* item stacks related to it.

If the code is compiled at this point, the item can already be seen in both the custom *creative menu tab* and in the index of items added by *JEI*. However, the name of the items is displayed as `armortinkers:stone_hammer` and the item has no texture (it appears as 2 black squares and 2 magenta squares). Additionally, the item cannot be obtained in a *survival* world, because it has no crafting recipe. To solve this issue, some additional files are needed in the *resources* package.

Item display name

The name of the item should be displayed as *Stone Hammer*. *Minecraft* convention has that within the name of any object in the game, nouns should always be capitalized. In other to set the proper name for this item, the line from Listing 5.3 should be added to the *en_us.json* file, in the *lang* folder.

It is worth mentioning at this point that this functionality for defining the name of the objects added to the game aims to allow the internationalization of the mod. For example, if the player sets the language of the game to *Romanian*, the *ro.json* file will be loaded instead. However, for this mod, only the *en_us.json* file is specified.

Listing 5.3: `en_us.json` file

```
1 "item.armortinkers.stone_hammer": "Stone Hammer"
```

Any `TranslatableComponent` created within the source code can be assigned here. This will be the case for the custom GUIs that will be added later.

Item model and Item texture

In order to add a texture for the item, a *model* file needs to be specified. The model file is placed in `resources/assets/armortinkers/models/item` and has to have the same name as the one specified for the item (in this case, `stone_hammer`). The model file is a regular *.json* file. The contents of this particular model file is presented in Listing 5.4.

Listing 5.4: `model/stone_hammer.json` file

```
1 {
2   "parent": "item/generated",
3   "textures": {
```

```
    "layer0": "armortinkers:item/stone_hammer"
  }
}
```

The `parent` field specifies how the item will look when being held in the players hand, while the `textures` field specifies the *.png* files that should be loaded for the texture. It should be noted that an item can have multiple layers for the texture, but most of the times it is just one.

The actual texture file is a *.png* file that is placed in the following destination, `resources/assets/armortinkers/textures/item`, and can have a different name from the one of the item, as long as it is correctly written in the model file.

Figure 5.3: Stone hammer texture

Figure 5.3: Stone hammer texture

Figure 5.3 presents the texture of the stone hammer. All textures for items and blocks are drawn as 16 pixels × 16 pixels images.

Item recipe

Finally, a recipe is needed for the *Stone Hammer*, in order to make it available to players in their *survival* world.

The *recipe* file is another *.json* file. This file is found in the following destination, `resources/data/armortinkers/recipes` and it does not need to have the same name as the item. When checking if the input configuration in a crafting table is a valid recipe, the whole folder is searched, thus not needing the file to have a special name.

Listing 5.5 show the structure of the recipe file. To be noted that this recipe is a *shaped crafting* recipe, meaning it can only be validated in a *Crafting Table*, with a 3×3 grid. The layout of the items is specified with keys, which are then assigned items. The result is specified under the `result` attribute.

Listing 5.5: `recipes/stone_hammer.json` **file**

```
{
  "type": "minecraft:crafting_shaped",
  "pattern": [
    " C ",
    " SC",
    "S  "
  ],
  "key": {
    "C": {
      "item": "minecraft:cobblestone"
```

```
    },
    "S": {
      "item": "minecraft:stick"
    }
  },
  "result": {
    "item": "armortinkers:stone_hammer"
  }
}
```

Figure B.1 shows how the recipe looks in game. This menu was generated by *JEI*, after pressing the key 'R' while hovering over the hammer. To be noted that since the default crafting system was used, there is no need to create any recipe classes in order to integrate the mod. The *JEI* mod provides functionalities for all basic recipe types found in *Minecraft Vanilla* by default.

Remark 5.5: Registering an item

This process of registering an item is repeated for all items added by the mod. The difference comes when adding custom items. In this case, instead of the `Item` class that was wrote for the supplier, we specify the name of the class that implements the custom item and add any additional parameters that might be needed in the constructor.

5.4.2. Registering a block

A simple block implies that the constructor of the *Block* class is called directly when adding a new entry in the register. To exemplify the whole process of how a simple block is added to the game, this subsection will present the design and implementation of the *Aluminium Block*.

To begin with, Listing 5.6 presents how the block register is instantiated. It is similar to how the item register was declared previously. The only difference is that the *block forge registry* is wrapped.

Listing 5.6: ModBlocks class

```
public static final DeferredRegister<Block> BLOCKS =
    DeferredRegister.create(ForgeRegistries.BLOCKS,
                            ArmorTinkers.MOD_ID);
```

After initializing the deferred register, any new block (be it simple or custom) is added to the game by adding it to the register. The deferred register behaves similar to a list, with the add() method being replaced by the register() method.

It can be seen in Listing 5.7 that the `register()` method is not called directly. This is because for blocks, the item associated with it must also be registered separately. As such, Listing A.1 show both the implementation of the `registerBlock()` method and the `registerBlockItem()` method.

Listing 5.7: ModBlocks class

```
public static final RegistryObject<Block> ALUMINIUM_BLOCK =
    registerBlock("aluminium_block",
                  () -> new Block(BlockBehaviour.Properties
                    .of(Material.METAL)
                    .strength(2f)
                    .requiresCorrectToolForDrops()),
                  ModCreativeModeTab.ARMOR_TINKERS_TAB);
```

Another notable difference between registering an item and registering a block is that the `Properties` of a block can be inherited from other blocks or materials. For example, a block has different attributes than an item, such as *sounds*, *particles*, *strength* and so on. To help with the development process, the library provides two methods: `Properties.of()` and `Properties.copy()`. The first one sets the block to inherit the properties from a certain material (in this case, *metal*) that was predefined by the game. The second one sets the block to copy the properties of another block that already exists in *Minecraft*.

Following the builder pattern, the values of any attributes can be changed, just like in the case for items. For the *Aluminium Block*, the *strength* of the block (i.e., the time required to break it with the correct tool of the lowest tier accepted) is set to `2.0f`, which is less than half as for the already existing *Iron Block*. Additionally, the *strength* is also related to how hard is for an explosion to break the block. To be noted that this method can take in two parameters, one for the *destroy time* and one for the *explosion resistance*.

Finally, the `requieresCorrectToolForDrops()` method ensures that the block will drop any items specified in its loot table only when destroyed with the appropriate tool of the right tier. In the same time, the *creative mode tab* is the same one as for the item registered in Subsection 5.4.1.

If the code is compiled at this point, the same scenario as for the item registration is encountered. However, along side with the block name, block texture and model, and block recipe, there are additional files that need to be created. These files are the *block state* file, the *loot table* file, and an *item model* file (in order to specify the model that the item related to the block should have. Additionally, the correct tool and tier for breaking the block needs to be specified. This is done with the use of tags.

Block name

To set the name of the block, the line presented in Listing 5.8 is added to the `en_us.json` file, similar to how the name for the item was set previously.

Listing 5.8: `en_us.json` file

```
"block.armortinkers.aluminium_block": "Aluminium Block"
```

Block states

As it was specified in Subsection 5.2.1, the *block state* describes which model should be loaded, depending on the state in which the block found in the world. The block state file is a *.json* file that needs to have the same name as the name of the block. Some of

the attributes that can be specified in this file are *facing*, *rotation*, *waterlogged*, *age* and multiple others. This file is located in `resources/assets/armortinkers/blockstates`.

Listing 5.9: `blockstates/aluminium_block.json` **file**

```
{
  "variants": {
    "": { "model": "armortinkers:block/aluminium_block"}
  }
}
```

As it can be seen in Listing 5.9, there is only one variant for the *Aluminium Block*, since its a cube that has the same texture for each of the six sides. Thus, it loads the same model, regardless of where and how it is placed in the world.

Block model and Block texture

The model file can have any name, since it is the block state file the one that specifies what model is loaded for each block. The location of the model files for blocks is `resources/assets/armortinkers/model/blocks`.

In Listing A.4 it can be seen that the *Aluminium Block* inherits the model from the `cube_all` specification. This, paired up with the `textures` attribute (which is set to `all`), means that the block will have the same texture for all of the faces of the cube. It should be noted that a *cube* block can have different textures for all of its sides. Additionally, as it will be seen in Section 5.7, we can specify model files that are not cubes.

The actual texture file is a *.png* file that is placed in the following destination, `resources/assets/armortinkers/textures/block`, and can have a different name from the one of the block, as long as it is correctly written in the model file. Figure 5.4 presents the texture of the aluminium block. To be noted that this texture is applied to all faces of the block.

Figure 5.4: Aluminium Block texture

Figure 5.4: Aluminium Block texture

Item model

As it can be seen in Listing 5.10, the block item model points to the block model. It will render the block item in the inventory in an isometric way.

Listing 5.10: `model/item/aluminium_block.json` **file**

```
{
  "parent": "armortinkers:block/aluminium_block"
```

```
3 }
```

Block recipe

The recipe file is located in the same folder as mentioned previously. Listing A.2 show the file, which is similar to the one for the *Stone Hammer*. Additionally Figure B.12 shows how the recipe looks in game, with the help of *JEI*, as it was previously presented.

Block loot table

The loot table for the *Aluminium Block* is presented in Listing A.3. The location of this file is `resources/data/armortinkers/loot_tables/blocks`. It is also a *.json* file that specifies what will be dropped from this block when it is broken with the correct tool and tier. It can be seen in the listing that it has only one pool to chose from, it will roll for the pool only one, with no bonus rolls. This means that the block will drop its block item variant any time it is mined correctly. Additionally, there is one condition in which the block item will be dropped, and that is if it survives an explosion. The way explosions work is that each block within the radius of explosion has a certain chance to drop its block item variant when an explosion occurs. This chance varies with the distance to the epicenter of the explosion and the strength of the block.

Set correct tool

In order to set the correct tool for drops, the *tag* files need to be created. As it was previously mentioned, *Minecraft* has a tag system. *Tags* are *.json* files in which the id of the item or block that needs to be under that tag is places.

To specify the correct tool and tier for the block, two tag files are needed. One file, `pickaxe.json`, tells the game that the block can be destroyed with a pickaxe, while the other file, `needs_stone_tool.json`, tells the game which tier of the previously mentioned tool is needed. The first file is located under the following path, `resources/-data/minecraft/tags/blocks/mineable` and is presented in Listing A.5, while the second file is located in `resources/data/minecraft/tags/blocks` and is presented in Listing A.6. To be noted that they both have the `replace` attribute set to `false`. This is because these tag files are under the *minecraft* name space, meaning they are already defined within the game. We do not want to replace those tag files, rather we want to append our own values to them. By creating this two files, the game now knows that the *Aluminium Block* can be mined with a pickaxe of T1 (stone) or higher.

Remark 5.6: Registering a block

This process of registering an block is repeated for all blocks added by the mod. The difference comes when adding custom blocks. In this case, instead of the Block class that was wrote for the supplier, we specify the name of the class that implements the custom block and add any additional parameters that might be needed in the constructor. Additionally, we might want some blocks to have a custom model. In this case, the model file will look different than the one already presented.

5.5. Creative mode tabs

Figure B.13 shows how the seven creative mode tabs all instantiate the class provided by *Minecraft*, `CreativeModeTab`. These tabs are used to group the items under the same menu when opening the creative mode inventory. The display name is a `TranslatableComponent`, thus the actual names of the tabs will be written in `en_us.json`.

The `makeIcon()` method is an abstract method, therefore, it must be implemented. Each object implements this method with anonymous classes. This method returns the item that will be displayed as icon for the respective tab.

5.6. Custom items

The items presented in this section have custom classes created, that extend [16] the `Item` class by overriding some of the methods or by adding new fields and functionalities.

5.6.1. Resin

Figure 5.5: `ResinItem` class

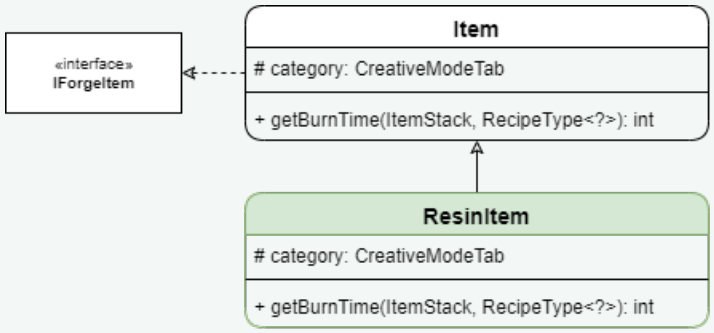

Figure 5.5: `ResinItem` class

Figure 5.5 shows how the `ResinItem` class extends the `Item` class, provided by Minecraft, which, in turn, implements [16] the `IForgeItem` interface, provided by Forge. It is worth mentioning that this figure shows only the fields and methods that are being overriden from the `Item` class, as presenting all of them would be beyond the scope of this paper.

As mentioned before, all items are registered in the same way and require the same configuration files. Therefore, those parts will be omitted from presenting the implementation of *Resin*.

The `ResinItem` class overrides a single method from the `Item` class, that being `getBurnTime()`. Listing 5.11 shows how this method is implemented. It returns the value 800 (coal returns 1600), which is the amount of time (in ticks) that this item will burn in a furnace if used as a fuel.

Listing 5.11: ModBlocks class

```
1 @Override
2 public int getBurnTime(ItemStack itemStack,
3                         @Nullable RecipeType<?> recipeType) {
```

```
4       return 800;
5   }
```

5.6.2. Armor parts

Figure 5.6 shows how the `ArmorPartItem` class extends the `Item` class, provided by Minecraft. It also has an instance of the `ColorRegisterHandler` class, in order to register the colors defined for items of this type. It is worth mentioning that this figure shows only the fields and methods that are being overriden from the `Item` class, as presenting all of them would be beyond the scope of this paper.

Figure 5.6: `ArmorPartItem` class

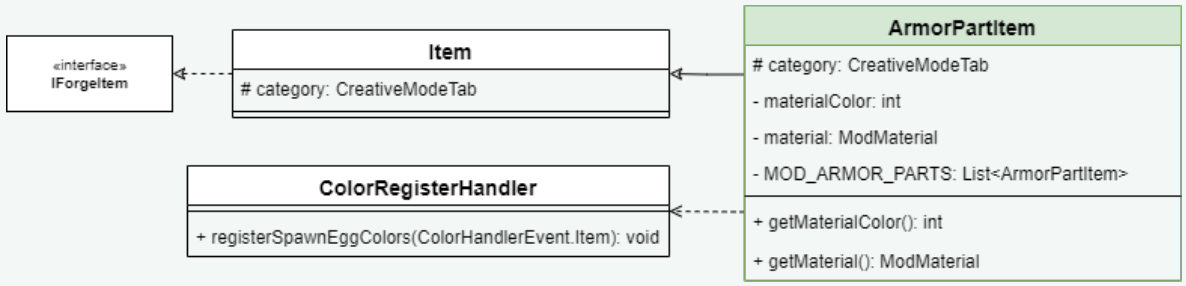

Figure 5.6: `ArmorPartItem` class

Since there are 20 new material types and 8 armor parts, it is not feasible to draw all the textures and to have 160 texture files that have the same shape, but different colors. Fortunately, *Minecraft* provides the ability to color the items dynamically, based on colors specified in implementation. For this, the color must be first registered. This is done with the help of the `ColorRegisterHandler` class. It registers the color of the armor part for every layer of the texture (in this case, there is only one layer). Thus, there are only 8 (gray-scale) texture files that get linked with the items and then colored.

Listing A.7 shows how the methods from the `ColorRegisterHandler` class are called only on the client side, because of the `value = Dist.CLIENT`. Additionally, the `getMaterialColor()` and `getMaterial()` methods return the private attributes from within the class.

The `ModMaterial` class is an enumeration of all the materials for which this mod provides armor parts. Listing 5.12 shows how the an *Armor Part Item* is instantiated and added to the items registry.

Listing 5.12: ModItems class

```
1 public static final RegistryObject<Item> WOOD_HEAD_PLATE
2     = ITEMS.register("wood_head_plate", () -> new ArmorPartItem(
3     new Item.Properties().
4         tab(ModCreativeModeTab.ARMOR_PART_TAB),
5     ModColors.WOOD, ModMaterial.WOOD));
```

Remark 5.7: Armor parts added in the game

The mod adds a total of eight armor parts, as follows: head plate, shoulder plate, front plate, back plate, wrist band, leg plate, tail plate, and boot plate.

5.6.3. Helmet, Chestplate, Leggings and Boots

The `HelmetItem`, `ChestPlateItem`, `LeggingsItem`, and `BootsItem` classes are all designed and implemented in a similar way. The only difference comes from their number of armor parts that make up the whole armor piece. Figure 5.7 shows how the *Helmet* item implement the `IForgeItem` class directly and extends the `ArmorItem` class. All methods in the `HelmetItem` class are presented in this figure, as well as where they were inherited from. Figures B.14, B.15, and B.16 present the UML diagrams for their respective classes, with less details than Figure 5.7.

Figure 5.7: `HelmetItem` class

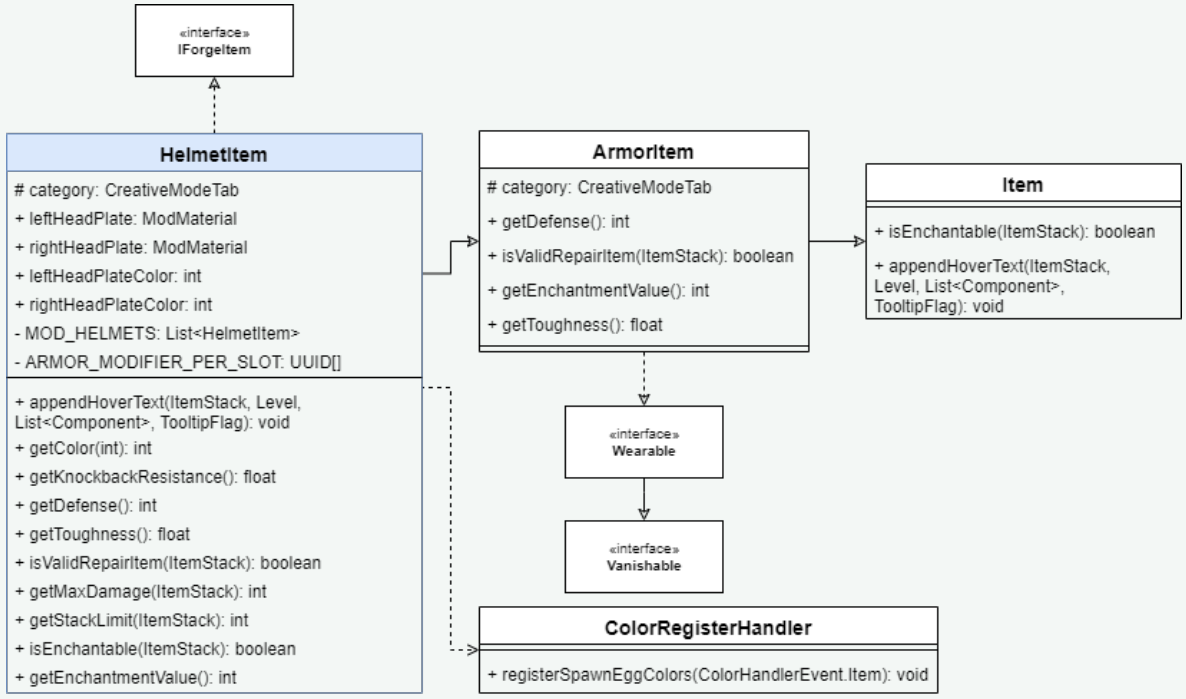

Figure 5.7: `HelmetItem` class

Remark 5.8: The concept of the mod

These four classes are the ones responsible for providing the player with the armor pieces they can wear. All the items derived from these classes are NOT enchantable, since they already have better attributes than the armor pieces currently implemented in the game. Additionally, they CANNOT be repaired, thus motivating the player to use the mod continuosly to craft new armor sets.

As it can be seen in the figures presented in this subsection, all of the four armor pieces are made out of different armor parts (that were discussed in Subsection 5.6.2),

which can then, in turn, be made out of different materials. The list bellow shows what are the parts used for each armor piece. With some computations, it can easily be seen that the player would be able to craft over 550.000 new armor pieces (computation based on the number of soft and hard materials and the number of armor parts used for each armor piece).

List 5.4: Armor pieces composition

- **helmet** - made out of a left head plate and a right head plate
- **chestplate** - made out of a left shoulder plate, a right shoulder plate, a front plate, a back plate and a wrist band
- **leggings** - made out of a left leg plate, a right leg plate and a tail plate
- **boots** - made out of a left boot plate and a right boot plate

In can be seen in Figure 5.7 (and Figures B.14, B.15,B.16, respectively) that all of the four classes also have a dependency on the `ColorRegisterHandler`. This is done for the same reason as presented in Subsection 5.6.2, only this time, the number of textures needed is of the order of hundreds of thousands. The difference is that, for these four classes, there are more layers to each texture.

In order to be able to color all of the armor parts independently, grey-scale textures for all the armor parts and their corresponding position in the armor piece were drawn. They are than layered on top of each other and together they form the whole item, with the advantage that each armor part can be colored independently. This is done by modifying the `getColor(int pTintIndex)` method to return a different color, based on the `pTintIndex` parameter. This parameter is actually the number associated to each layer of the texture.

Finally, each armor piece has four important attributes: durability, armor points, armor toughness and knockback resistance. These attributes are computed dynamically, based on the materials in the armor piece. Subsection 5.6.4 will present how each material type influences these four attributes.

List 5.5: General rule for computing the attributes

- **durability** - is the durability of the item for the armor slot (5 for head and feet and 6 for chest and legs) times the overall durability of the armor piece
- **defense** - sum of the defense points provided by each material
- **toughness** - sum of the toughness points provided by each material
- **knockback** - sum of the knockback points provided by each material

Since not all armor parts are equally big, weights have been attached to each armor part that makes up the whole armor piece. As such, the list bellow provides those weights. Additionally, armour toughness and knockback resistance is also different, depending if the armor part is a small one (head plate, shoulder plate, wrist bands, boot plate or leg plate) or a big one (tail plate, front plate or back plate).

List 5.6: Different weights for durability computation

- **helmet** - 50% left head plate and 50% right head plate
- **chestplate** - 20% left shoulder plate, 20% right shoulder plate, 30% front plate, 20% back plate and 10% wrist band
- **leggings** - 30% left leg plate, 30% right leg plate and 40% tail plate
- **boots** - 50% left boot plate and 50% right boot plate

5.6.4. Properties of new materials

Table 5.1 presents the properties of the new materials added (and of the materials already existing in *Minecraft*) in a similar manner to the table in Subsection 4.2.6. The abbreviations stand for: durability multiplier (DM), part protections (PP), toughness for a big part (TB), toughness for a small part (TS), and knockback resistance for a big part (KRB) and for a small part. Part protections are presented in the following order: head plate, front plate, back plate, shoulder plate, wrist band, leg plate, tail plate and boot plate.

Table 5.1: New material types property values

Material	DM	PP	TB, TS	KRB, KRS
Wood	2	1, 1, 1, 1, 1, 1, 1, 1	0.0, 0.0	0.0, 0.0
Cactus	1	1, 1, 1, 1, 1, 1, 1, 1	0.0, 0.0	0.0, 0.0
Stone	3	1, 2, 2, 1, 1, 1, 1, 1	0.0, 0.0	0.0, 0.0
Netherrack	3	1, 2, 2, 1, 1, 1, 1, 1	0.0, 0.0	0.0, 0.0
End Stone	3	1, 2, 2, 1, 1, 1, 1, 1	0.0, 0.0	0.0, 0.0
Flint	2	1, 1, 1, 1, 1, 1, 1, 1	0.0, 0.0	0.0, 0.0
Copper	4	1, 2, 2, 1, 1, 1, 2, 1	0.0, 0.0	0.0, 0.0
Iron	5	1, 2, 2, 1, 1, 2, 2, 1	0.0, 0.0	0.0, 0.0
Gold	4	1, 2, 2, 1, 1, 2, 1, 1	0.0, 0.0	0.05, 0.1
Obsidian	5	1, 1, 1, 1, 1, 1, 1, 1	1.0, 2.0	0.0, 0.0
Aluminium	4	1, 2, 2, 1, 1, 1, 2, 1	0.0, 0.0	0.0, 0.0
Tin	4	1, 2, 2, 1, 1, 1, 2, 1	0.0, 0.0	0.0, 0.0
Zinc	5	1, 2, 2, 1, 1, 2, 2, 1	0.0, 0.0	0.0, 0.0
Magnesium	5	1, 2, 2, 1, 1, 2, 2, 1	0.0, 0.0	0.0, 0.0
Silver	6	1, 2, 2, 1, 1, 2, 3, 1	0.0, 0.0	0.0, 0.0
Lead	6	1, 2, 2, 1, 1, 2, 3, 1	0.0, 0.0	0.0, 0.0
Brass	7	1, 2, 2, 1, 1, 2, 3, 1	0.5, 1.0	0.0, 0.0
Bronze	7	1, 2, 2, 1, 1, 2, 3, 1	0.5, 1.0	0.0, 0.0
Vibranium	9	1, 3, 3, 1, 1, 2, 3, 1	2.0, 3.0	0.1, 0.25
Adamantium	12	1, 3, 3, 1, 1, 2, 3, 1	3.0, 5.0	0.2, 0.3

Remark 5.9: Capped values and play incentive

It can be noted from the table that, in some cases, the player can obtain over 1.0 (100%) knockback resistance, as well as over 20 points of armor toughness. This is done on purpose, to provide incentive for the player to craft the armor out of different materials, instead of choosing the same (best) material for all the parts of the armor.

5.7. Custom blocks

The blocks presented in this section have custom classes created, that extend the `Block` class by overriding some of the methods or by adding new fields and functionalities. Some of the blocks (i.e., the new crafting stations) have custom 3D models that were designed by me in *Blockbench*.

5.7.1. Maple log

Figure 5.8 show how the class corresponding to the *Maple Log* was designed. It extends the *RotatedPillarBlock* class because, as part of a tree trunk, the *Maple Log* has the same texture on 4 of its sides and another texture on the remaining 2 sides. The methods overriden by this class are related to how a general log block should behave inside the game. The `getTollModifiedState()` method is called when the player right clicks the block with an axe, stripping it of its bark. This behaviour is similar to how the other logs that already exist in the game. Additionally, *Maple Planks* and *Maple Wood* have been added, together with the stripped variants for the log and the wood.

Figure 5.8: Maple log UML diagram

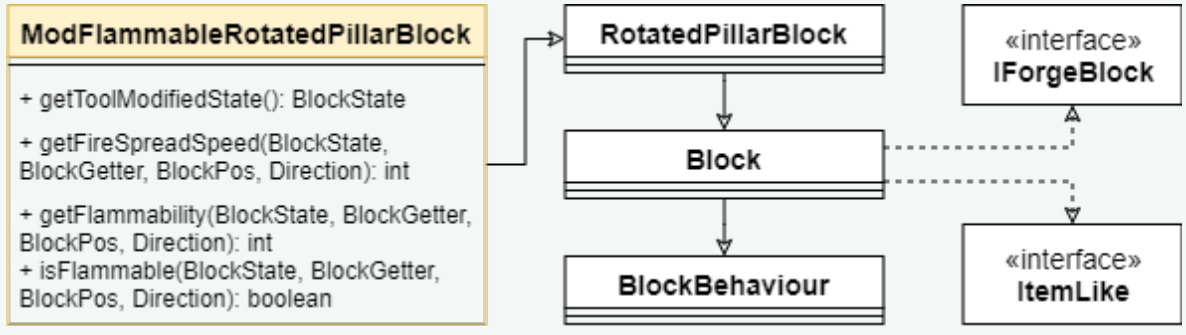

Figure 5.8: Maple log UML diagram

5.7.2. Pattern station

Figure 5.9 shows the UML diagram for all the four stations added to the game. They each have a custom 3D model that was designed in *Blockbench*. The resulting model files were placed inside the `model` folder, to be loaded when the game is launched.

It can be seen in the figure that each of the four stations extends the abstract class, `BaseEntityBlock`. In turn, this class extends the `Block` class and implements the

Figure 5.9: The four stations added

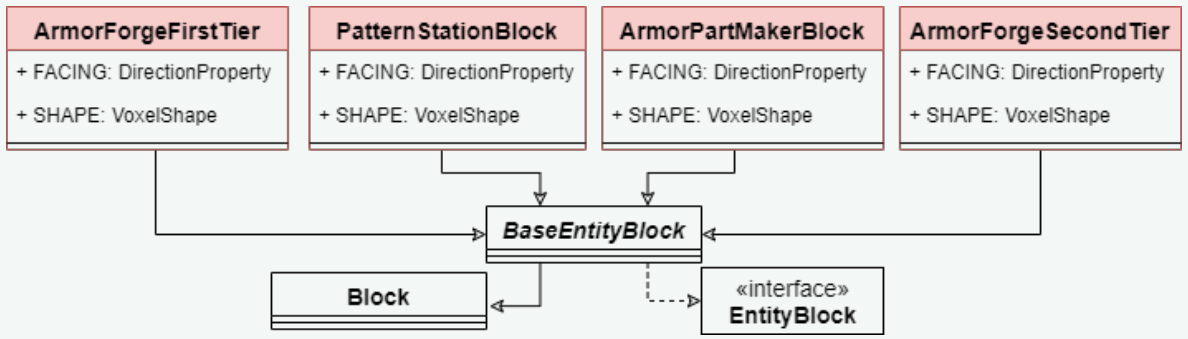

Figure 5.9: The four stations added

`EntityBlock` interface. The `SHAPE` attribute from each class specifies the hitbox of the block. This is necessary because all of these four blocks have custom models.

From the `BaseEntityBlock` class, each of the blocks inherits methods that will allow the developer to add a block entity to these stations. These methods are presented in the list bellow.

List 5.7: Methods related to block entities

- `newBlockEntity()` - creates a new block entity attached to this block (called when the block is placed in the world)
- `use()` - (in this case) opens the interface of the station via the `openGUI()` method from the `NetworkHooks` class
- `onRemove()` - is called when the block is destroyer (used to remove the block entity and to drop all the items that are inside of the container at the time of removal)
- `getTicker()` - used to return the `tick()` method from the block entity associated with the block

Remark 5.10: Rotation

Given the fact that these four blocks are not cubes, we need to also specify the `FACING` of the block (relative to the north direction). This way, the block will always be placed with the relative north of the block towards the player.

Figure 5.10 shows how the *Pattern Station* block model looks inside *Blockbench*.

5.7.3. Armor part station

The *Armor Part Station* is designed and implemented in a similar manner to the *Pattern Station* block. Figure 5.9 already presented how the class extends classes already existing in *Minecraft*.

Figure 5.11 shows how the *Armor Part Station* block model looks inside *Blockbench*.

Figure 5.10: Model for the pattern station

Figure 5.10: Model for the pattern station

Figure 5.11: Model for the armor part station

Figure 5.11: Model for the armor part station

5.7.4. Armor forge

The *Armor Forge* (tier 1 and tier 2) is designed and implemented in a similar manner to the *Pattern Station* block. Figure 5.9 already presented how the class extends classes already existing in *Minecraft*. The first tier of the armor forge is able to process the soft material, while the second tier of the armor forge is able to process only the hard materials.

Figure 5.12: Model for the armor forge tier 1 station

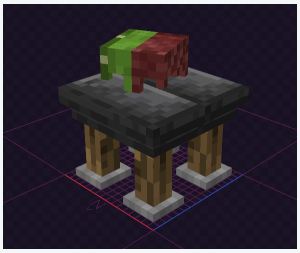

Figure 5.12: Model for the armor forge tier 1 station

Figure 5.12 shows how the *Armor Forge Tier 1* block model looks inside *Blockbench*.

Figure 5.13 shows how the *Armor Forge Tier 2* block model looks inside *Blockbench*.

Figure 5.13: Model for the armor forge tier 2 station

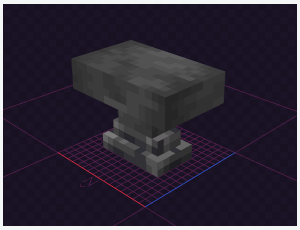

Figure 5.13: Model for the armor forge tier 2 station

Remark 5.11: Soft and hard materials

It is worth noting at this point that the soft materials of this mod are: wood, cactus, stone, netherrack, end stone and, flint, while the hard materials are: copper, iron, gold, obsidian, aluminium, tin, zinc, magnesium, silver, lead, brass, bronze, vibranium, and adamantium.

Remark 5.12: New crafting stations

These four new blocks added to the game are the new crafting stations that will provide the player custom recipes for crafting the armor added by the mod.

5.8. Custom block entities

As it was mentioned in Section 5.7, each of the four crafting stations has a block entity attached to it, in order to allow for the implementation of new crafting recipes. The UML diagram for all of these new block entities is presented in Figure 5.14.

It can be seen in the figure that each of the block entity classes extends the `BlockEntity` class, provided by *Minecraft*, and implements the `MenuProvider` interface. The implementation of this interface is not necessary, if the block entity does not have a GUI attached to it.

Remark 5.13: Motivation for block entities

Block entities are used in conjunction with blocks, when the block needs to be able to handle containers (and, in turn, items).

The main field of the block entity class is the `itemHandler` which, as the name suggests, is responsible for handling the items placed in it. Additionally, each entity has a `# data: ContainerData` field, which is used to store information about the current progress of the recipe (`- progress: int`) and the maximum progress of any recipe crafted in the particular block entity (`- maxProgress: int`). These values are expressed in ticks.

From the methods perspective, each block entity class has a method called `tick()`. This is the method in which the actual crafting algorithm is implemented. This aspect was already presented in Section 4.8.

Figure 5.14: Block entities UML diagram

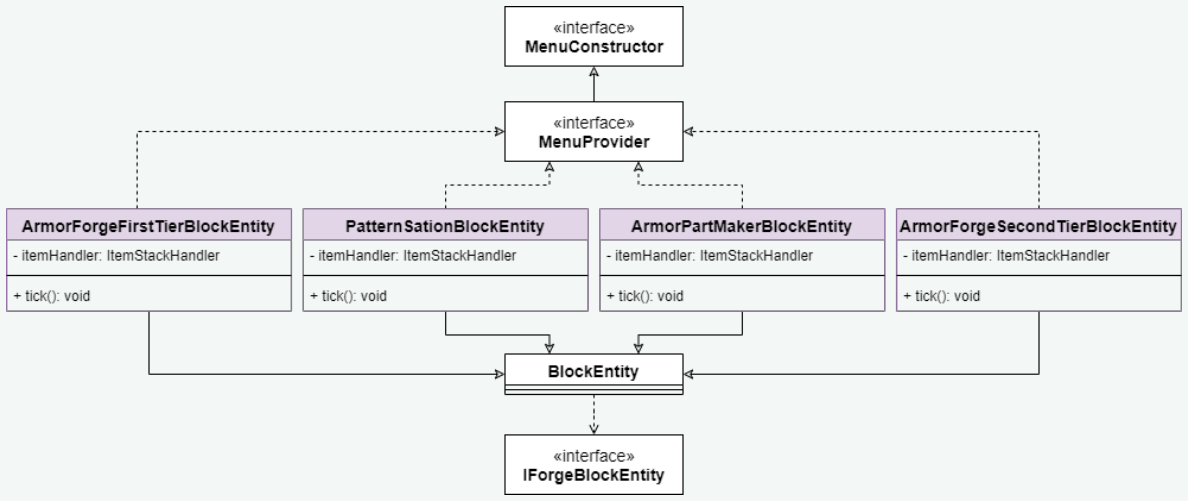

Figure 5.14: Block entities UML diagram

5.8.1. Pattern station

Listing 5.13 presents the implementation of the `tick()` method. It can be seen that it follows the crafting algorithm proposed in Subsection 4.8.2, with the additional `setChanged()` method that updates the GUI to show the current progress of the recipe being crafted.

Listing 5.13: `PatternStationBlockEntity.java` **file**

```
1  public static void tick(Level pLevel, BlockPos pPos,
2                          BlockState pState,
3                          PatternStationBlockEntity pBlockEntity){
4      if(hasRecipe(pBlockEntity)) {
5          pBlockEntity.progress++;
6          setChanged(pLevel, pPos, pState);
7
8          if (pBlockEntity.progress > pBlockEntity.maxProgress) {
9              craftItem(pBlockEntity);
10             extractIngredients(pBlockEntity);
11         }
12     } else {
13         pBlockEntity.resetProgress();
14         setChanged(pLevel, pPos, pState);
15     }
16 }
```

The implementation of the static methods `hasRecipe()`, `craftItem()` and also `extractIngredients()` are presented in Listings A.8, A.9, and A.10, respectively. Since they are almost similar between all the block entities, they will NOT be presented for the other three block entities.

5.8.2. Armor part station

Listing 5.14 presents the implementation of the `tick()` method. It can be seen that it follows the crafting algorithm proposed in Subsection 4.8.2, with the additional `setChanged()` method that updates the GUI to show the current progress of the recipe being crafted.

Listing 5.14: `ArmorPartMakerBlockEntity.java` **file**

```
public static void tick(Level pLevel, BlockPos pPos,
                        BlockState pState,
                        ArmorPartMakerBlockEntity pBlockEntity){
    if(hasRecipe(pBlockEntity)) {
        pBlockEntity.progress++;
        setChanged(pLevel, pPos, pState);

        if (pBlockEntity.progress > pBlockEntity.maxProgress) {
            craftItem(pBlockEntity);
            extractIngredients(pBlockEntity);
        }
    } else {
        pBlockEntity.resetProgress();
        setChanged(pLevel, pPos, pState);
    }
}
```

5.8.3. Armor forge

As stated before, there are two tiers of the armor forge. Their block entity implementation is similar in structure, since they both implement the same crafting recipe type, *Armor Forging*. The difference comes in what types of part are allowed in each of the block entity's inventory.

Unlike the two block entities presented previously, the `tick()` method is adapted from the the crafting algorithm and changed in a way that accommodates four different interfaces (one for each armor piece). Listing 5.15 shows the implementation of this method. It can be seen that it follows the same crafting algorithm, with the addition that it is repeated three more times.

Listing 5.15: `ArmorForgeFirstTierBlockEntity.java` **file**

```
public static void tick(Level pLevel, BlockPos pPos,
                        BlockState pState,
                        ArmorForgeFirstTierBlockEntity
                           pBlockEntity) {
    if(hasHelmetRecipe(pBlockEntity)) {
        pBlockEntity.progress++;
        setChanged(pLevel, pPos, pState);
        if (pBlockEntity.progress > pBlockEntity.maxProgress) {
            craftHelmet(pBlockEntity);
            extractIngredientsHelmet(pBlockEntity);
        }
```

```
11      } else if (hasChestplateRecipe(pBlockEntity)) {
12          pBlockEntity.progress++;
13          setChanged(pLevel, pPos, pState);
14          if (pBlockEntity.progress > pBlockEntity.maxProgress) {
15              craftChestplate(pBlockEntity);
16              extractIngredientsChestplate(pBlockEntity);
17          }
18      } else if (hasLeggingsRecipe(pBlockEntity)) {
19          pBlockEntity.progress++;
20          setChanged(pLevel, pPos, pState);
21          if (pBlockEntity.progress > pBlockEntity.maxProgress) {
22              craftLeggings(pBlockEntity);
23              extractIngredientsLeggings(pBlockEntity);
24          }
25      } else if (hasBootsRecipe(pBlockEntity)) {
26          pBlockEntity.progress++;
27          setChanged(pLevel, pPos, pState);
28          if (pBlockEntity.progress > pBlockEntity.maxProgress) {
29              craftBoots(pBlockEntity);
30              extractIngredientsBoots(pBlockEntity);
31          }
32      } else {
33          pBlockEntity.resetProgress();
34          setChanged(pLevel, pPos, pState);
35      }
36  }
```

The implementation of the `tick()` method for the second tier of the armor forge is almost the same and thus will not be presented in a listing.

5.9. Custom interfaces

In order to create an interface (or a new menu type), in addition to a block entity, two more classes are needed: a *screen* class and a *menu* class. The differences between these two types of classes are presented in Figure B.17[18]. In short, the screen class is only displayed on the client side and handles the display of the actual image used for the GUI, while the menu class is the one that synchronises the client and server when it comes to the contents of the item handlers. Thus, it can be said that the menu class handles the actual slots in which the items can be placed.

In the terms of slots that can be added to a menu, there is the option to extend the `SlotItemHandler` class as to provide different functionalities, such as only allowing certain items inside the slot or not allowing the player to input items in the slot (suitable for a result slot). Additionally, each menu class needs to add the 36 slots linked to the player's inventory as well. The manner in which slots are added is by specifying the coordinates of the image where they should be drawn.

We can see in Figure 5.15 that all the menu classes of this project extend the class called `AbstractContainerMenu`, while all the screen classes of this project extend the `AbstractContainerScreen` class. The later contains and instance of the first, thus imposing that any screen class contains an instance of the menu class. At the same time,

Figure 5.15: Menu and screen classes diagram

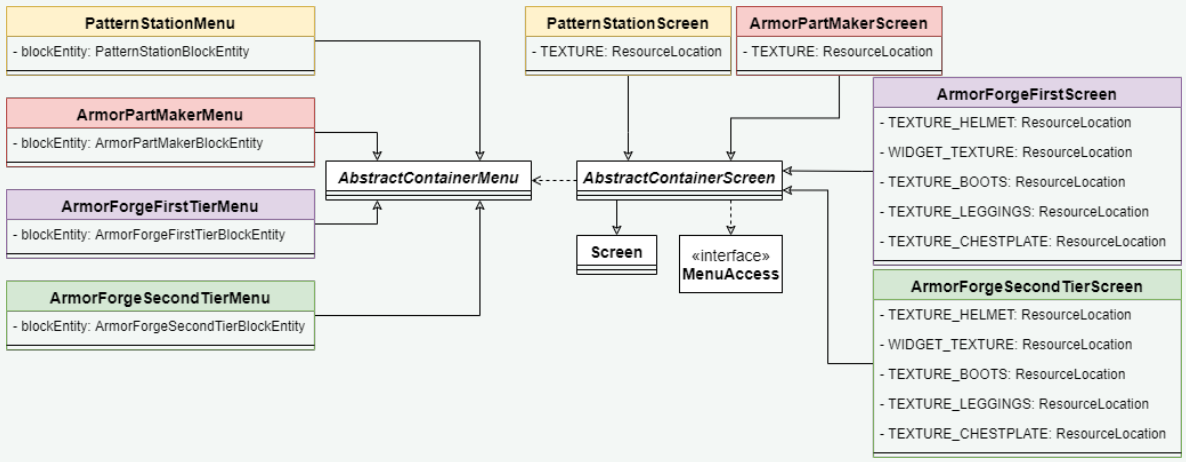

Figure 5.15: Menu and screen classes diagram

each menu class contains an instance of the block entity class for which it implements the menu.

All the screen classes have a field called `TEXTURE` which stores the location in the resource package where the actual image is found. For this mod, each texture related to the graphical interfaces of block entities (and later, for integration with *JEI*) are placed in `resources/assets/armortinkers/textures/gui`. The two tiers of the armor forge have four images that are loaded (one for each armor piece) and another image (`WIDGET_TEXTURE`) that stores the buttons with which the player can switch between the tabs of the interface. Images can have any sizes, but its a convention to draw them in a 256 by 256 square, even if the actual image does not cover the whole space.

Remark 5.14: Images used for the screen classes

Any text that needs to appear on an image from the screen class is added within the code. It does not to be drawn on the actual image.

Methods presented in Listing A.11 are used to add the slots for the inventory and the hotbar. They are exactly the same for each menu creation method and are called when the particular slots for each menu are added.

For each of the four stations, only the instruction that adds the specific slots will be presented, since it is the part that changes from station to station. Additionally, each texture file for the image that is displayed has the progress bar drawn in white, outside the actual texture (but inside the 256 by 256 square). The white progress bar is drawn over the grey progress bar proportional to how much time passed since the crafting process started.

Remark 5.15: Menu and screen registration

New menus and screes need to be registered the same way item or blocks do. This behaviour is covered by the `ModMenuTypes` class. Listing A.12 shows how the registry for menus is instantiated and how a menu is registered to it.

5.9.1. Pattern station menu and screen

Listing 5.16 shows the instruction in the menu creation method for the *Pattern Station* that handles the addition of slots in the menu. Besides the coordinates of the most left-upper corner, each slot also has an `index` associated with it, for easy referencing.

Listing 5.16: `PatternStationMenu.java` **file**

```
this.blockEntity.getCapability(CapabilityItemHandler.
    ITEM_HANDLER_CAPABILITY).ifPresent(handler -> {
    this.addSlot(new SlotItemHandler(handler, 0, 58, 21));
    this.addSlot(new ModBlankPatternSlot(handler, 1, 58, 43));
    this.addSlot(new ModResultSlot(handler, 2, 125, 32));
});
```

It can be seen in the listing above that the instruction adds one default slot (`SlotItemHandler`) which will be used to input the dye used to create the armor part pattern, a custom slot (`ModBlankPatternSlot`) that only accepts *wooden blank patterns* or *golden blank patterns*, and lastly, a custom slot (`ModResultSlot`) which is used to store the resulting item of the crafting recipe, without allowing the player to place items in it.

Listing 5.17: `PatternStationScreen.java` **file**

```
if (menu.isCrafting()) {
    blit(pPoseStack, x + 85, y + 33, 176, 0,
        menu.getScaledProgress(), 14);
}
```

Listing 5.17 shows the instruction in the `renderBg()` method of the screen class. This method shows how the progress bar is drawn on the image, based on the scale progress of the overall process. The scaled process is simple a ratio of the elapsed time over the total time, multiplied with the width of the proress bar. Additionally, Figure B.18 shows the texture for the interface (without any text components).

5.9.2. Armor part station menu and screen

Listing 5.18 shows the instruction in the menu creation method for the *Pattern Station* that handles the addition of slots in the menu. Besides the coordinates of the most left-upper corner, each slot also has an `index` associated with it, for easy referencing.

Listing 5.18: `ArmorPartMakerMenu.java` **file**

```
this.blockEntity.getCapability(CapabilityItemHandler.
    ITEM_HANDLER_CAPABILITY).ifPresent( handler -> {
    this.addSlot(new ModHammerSlot(handler, 0, 23, 32));
    this.addSlot(new SlotItemHandler(handler, 1, 58, 21));
    this.addSlot(new ModPatternSlot(handler, 2, 58, 43));
    this.addSlot(new ModResultSlot(handler, 3, 125, 32));
});
```

It can be seen in the listing above that the instruction implements a custom slot (`ModHammerSlot`), which will only allow the player to place a single *Stone Hammer* or

Iron Hammer, a default slot item `SlotItemHandler` for the material, a custom slot (`ModPatternSlot`) that only allows the player to place patterns that were crafted in the *Pattern Station*, and a custom result slot that was presented previously.

The instruction that shows what is the current progress status, with the help of the status bar will not be presented anymore, since it is similar, with the only difference being that the number values are changed, depending on the size of the bar. Additionally, Figure B.19 shows the texture for the interface (without any text components).

5.9.3. Armor forge menu and screen

As it was mentioned before, there are two tiers for the *Armor Forge*. From the point of view of their menu, they are very similar (the slots are placed in the same position). From the point of view of their corresponding screen classes, the textures used for the actual interfaces are different.

Both armor forges have four tabs, each selectable with one of the four buttons on the right side of the menu. These buttons change the background texture for the screen class and activate/deactivate the corresponding slots for the menu class.

Listing A.13 shows the instruction in the menu creation method for the *Armor Forge Tier 1* that handles the addition of slots in the menu. Besides the coordinates of the most left-upper corner, each slot also has an `index` associated with it, for easy referencing. The slots added are also saved as local variable in the class, as the screen class needs access to them in order to activate/deactivate them, based on the buttons pressed.

It can be seen in the listing that there are 16 `ModArmorForgeSlot` slots added. This class of slots implements the activation/deactivation methods. Inactive slots are not shown on the interface, thus they are inaccessible. In addition, there is the result slot with was presented previously, and four more results slots which are placed on top of the button. This was done in order to give the buttons a glint when hovered on (I chose the result slot type, because the player cannot place any items in it).

Figures B.20, B.21, B.22, and B.23, show the texture of each tab of the interface, in order, for the helmet tab, the chestplate tab, the leggings tab, and the boots tab respectively. These texture do not have any text components drawn onto them, as they will be drawn dynamically, from the code.

5.10. Custom recipe types

In order to add functionality to the menu types added in the game, new recipe types must be defined. We recall from Subsection 5.4.1 that recipes are specified in `.json` files. In order to create custom template for these recipe files, we need to implement recipe classes that will handle reading from these `.json` files.

Figure 5.16 shows how each recipe class simply implements the `Recipe` interface.

Each recipe class has a Boolean method called `matches()` which is used to validate the items in the input slots of each station. The items need to be in the right position for the recipe to be declared as valid. Additionally, each class also implement two network communication methods, that deal with the communication between the logical client and the logical server. This aspect was presented in Subsection 4.8.3.

It is worth mentioning that each recipe has a `Type` class and a `Serializer` class instantiated. The `Type` class is the one that handles how the `.json` files describes the

Figure 5.16: Recipe classes diagram

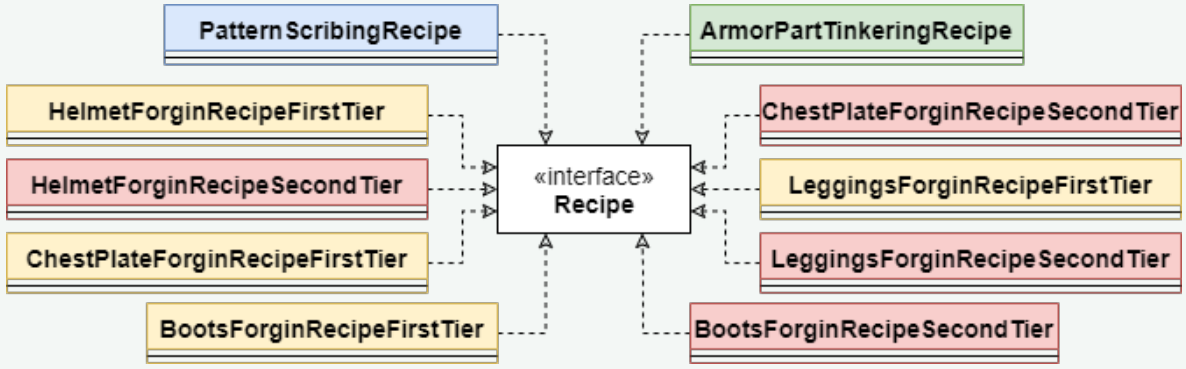

Figure 5.16: Recipe classes diagram

recipe, while the `Serializer` class extract the information from the `.json` file and handles the communication over the internal network.

The structure of the `.json` file is similar between all the new recipe types. A general template is presented in Listing 5.19. The `type` attribute represents the defined type for the new recipe, the `ingredients` attribute is a list of all the ingredients needed for the recipem abd the `output` attribute contains the result of the craft recipe. It is very important to note that for the ingredient list, the order in which the items are specified matters.

Listing 5.19: Template for the new recipe types

```
{
    "type": "",
    "ingredients": [],
    "output": {}
}
```

Remark 5.16: Recipe registration

New recipes need to be registered the same way item or blocks do. This behaviour is covered by the `ModRecipes` class. Listing A.14 shows how the registry for recipes is instantiated and how a recipe is registered to it.

5.10.1. Pattern scribing recipe

Listing 5.20 shows how the ID for the *pattern scribing* recipe type is defined. Being a `ResourceLocation`, it also requires the definition of the name space under which it can be found. The resulting string, `armortinkers:pattern_scribing`, is used when describing the type of the recipe inside the `.json` file.

Listing 5.20: `PatternScribingRecipe.Serializer` file

```
public static final ResourceLocation ID = new ResourceLocation(
    ArmorTinkers.MOD_ID, "pattern_scribing");
```

Listing 5.21 shows how the methods that handle the network communication are implemented. The `fromNetwork()` method reads the ingredients sent by the client side and validates the crafting recipe, while the `toNetwork()` method is used to send the resulting item back to the client side.

Listing 5.21: `PatternScribingRecipe.Serializer` file

```
@Override
public PatternScribingRecipe fromNetwork(ResourceLocation id,
   FriendlyByteBuf buf) {
    NonNullList<Ingredient> inputs = NonNullList.withSize(buf.
       readInt(), Ingredient.EMPTY);
    for (int i = 0; i < inputs.size(); i++) {
        inputs.set(i, Ingredient.fromNetwork(buf));
    }
    ItemStack output = buf.readItem();
    return new PatternScribingRecipe(id, output, inputs);
}
@Override
public void toNetwork(FriendlyByteBuf buf, PatternScribingRecipe
    recipe) {
    buf.writeInt(recipe.getIngredients().size());
    for (Ingredient ing : recipe.getIngredients()) {
        ing.toNetwork(buf);
    }
    buf.writeItemStack(recipe.getResultItem(), false);
}
```

Remark 5.17: Network communication methods

The two network communication methods are similar for all recipe classes, thus will NOT be presented again fpr the other classes.

5.10.2. Armor part tinkering

Listing 5.22 shows how the ID for the *armor part tinkering* recipe type is defined. Being a `ResourceLocation`, it also requires the definition of the name space under which it can be found. The resulting string, `armortinkers:armor_part_tinkering`, is used when describing the type of the recipe inside the `.json` file.

Listing 5.22: `ArmorPartTinkeringRecipe.Serializer` file

```
public static final ResourceLocation ID = new ResourceLocation(
   ArmorTinkers.MOD_ID, "armor_part_tinkering");
```

5.10.3. Armor forging

For the armor forging process, I implemented 8 new recipe types: helmet forging (tiers 1 and 2), chestplate forging (tiers 1 and 2), leggings forging (tiers 1 and 2), and boots forging (tiers 1 and 2). Listing 5.23 shows only how the ID for the *helmet forging tier 1* recipe type is defined. Being a `ResourceLocation`, it also requires the

definition of the name space under which it can be found. The resulting string, `armortinkers:helmet_forging_t1`, is used when describing the type of the recipe inside the `.json` file. For the rest of the recipe types, their ID can be easily inferred.

Listing 5.23: `HelmetForginRecipeFirstTier.Serializer` file

```
1 public static final ResourceLocation ID = new ResourceLocation(
    ArmorTinkers.MOD_ID, "helmet_forging_t1");
```

5.11. Custom world generation

Figure 5.17 shows how the classes related to world generation are designed and what properties they inherit from their parent classes or the interfaces they implement. As it can be seen, the class `ModWorldEvents` subscribes to an event bus provided by Forge. This class subscribes a single event to the bus, the `biomeLoadingEvent()` method. This method calls the generational methods for both tree generation and ore generation.

Figure 5.17: World generation classes diagram

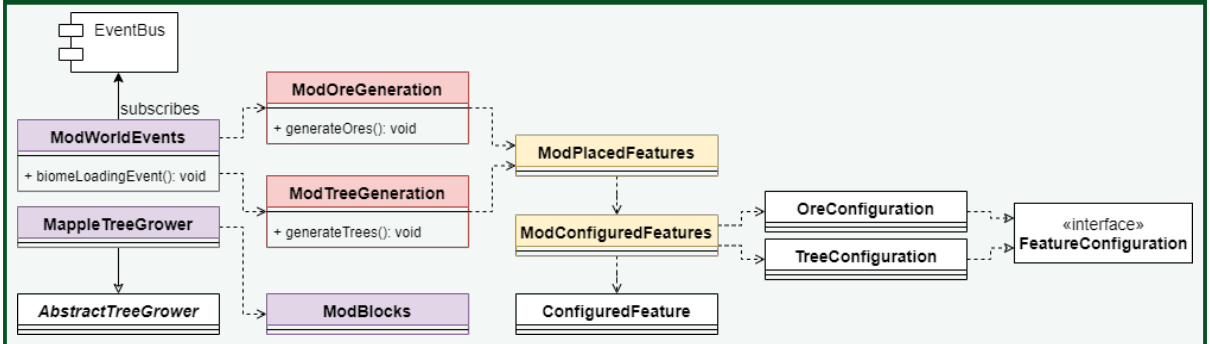

Figure 5.17: World generation classes diagram

It can also be seen in the figure that any placed feature uses a configured feature to generate ores or trees into the world. However, these features are different for ores and for trees. As such, they will be presented in separate sections.

5.11.1. Tree generation

This mod adds e new type of tree called *maple*. For the maple tree, the configured feature tells the game how an individual tree looks like. *Minecraft* already ships with a few configured features for this process. Since I want the maple tree to generate in the same biomes as the spruce tree, I chose the methods that build the maple tree in the same configuration as a spruce tree. However, the maple tree comes in the 1 by 1 configuration only. The leaves of the tree are also similar with the ones for the spruce tree, but they are colored differently.

The configured feature is used by the placed feature in order for the game to place the actual trees randomly, throughout the whole world. For the placed feature, I set the type of placement to a *tree* placement (available from Forge) and the number of trees spawned per chuck to 1, with a 10% chance to spawn another 2 trees in the same chunk. However, this tree is set to be a coniferous tree, thus spawning only in colder biomes.

5.11.2. Ore generation

This mod adds eight new ore types to the world, each with a stone and deep-slate variant. As it can be seen in Figure 5.17, the configured features for ore generation use *ore configurations*.

On one hand, ore configurations tell the game what kind of block are on the candidate list to be replaced by ores. For this mod, I chose to replace all types of stone (stone, andesite, granite, and diorite) with the stone variant of the ore, and any deep-slate with the deep-slate variant of the ore.

From these ore configurations, configured features are then implemented. In the case of ores, the configured features tell the game how will an ore vein generate. This mainly specifies how many blocks each vein can have (in general). I implemented two types of configured features, one with more ores per vein (`LARGE` variant) and another one with less ores per vein (`SMALL` variant).

All these configured features are then used to implement the placed features for ore generation. The placed features specify to the game how veins should be distributed throughout the world. Forge comes with two main methods for this process. One is a *triangle* type distribution and the other one is a *uniform* type distribution. Figure B.24 shows hos these types of distributions look like, but only for the ores already existing in the game.

For the mod presented in this paper, the triangle distribution type uses the configured features with more ores per vein, meaning that it is more likely to find bigger veins of the ore half way between the highest most point and lowest most point specified in the implementation. Likewise, the uniform distribution type uses the configured feature with less ore per vein, meaning that finding ore veins of the same size has the likelihood between the vertical coordinates specified in the method.

Table 5.2 shows a comparison between the values for the ores already existing in *Minecraft* and the values for the ores added by the mod. These values are related to the implementation that was specified in this subsection.

Remark 5.18: Comparison between Minecraft and Armor Tinkers

It is important to note that the upper half of the table shows the values for the ores already existing in *Minecraft*, while the lower half of the table show the values for the ores that are added by the *Armor Tinkers* mod. This was done in order to be able to compare the values and present to the reader how they are similar, in order to not affect the balance in the original game.

Table 5.2: Ore generation values

Ore	Type	Veins/chunk	Min height	Max height	Blocks/vein
Coal	uniform	30	136	256	10
	triangle	20	0	192	15
Iron	uniform	10	-64	72	6
	triangle	90	80	384	8
	triangle	10	-24	56	8
Copper	triangle	16	-16	122	10
Gold	uniform	1	-64	-48	4
	triangle	4	-64	32	4
Diamond	triangle	7	-80	80	4
	triangle*	9	-80	80	6
Aluminium	triangle	17	-32	80	10
	uniform	10	-16	112	9
Tin	triangle	17	-32	80	10
	uniform	10	-16	112	9
Lead	triangle	10	-40	40	7
	uniform	15	-64	40	5
Silver	triangle	10	-40	40	7
	uniform	15	-64	40	5
Zinc	triangle	10	-80	80	10
	uniform	30	32	64	8
Magnesium	triangle	10	-80	80	10
	uniform	30	-16	16	8

5.12. Integration with other mods

Integration with other mods is made possible by the authors of the mod with which the integration should be made. For some of the mods, the integration is as simple as adding the mod in the `build.gradle` file as a dependency, but for other mods, integration requires also additional classes.

5.12.1. *The One Probe* - TOP

Since *TOP*[19] interacts with only the blocks added to the game and since the *Armor Tinkers* mod register all the blocks to the same register as the ones already existing in *Minecraft*, integration with it is as simple as adding a dependency in the `build.gradle`

file. Listing 5.24 shows this dependency, that should be added under the `repositories` clause and another instruction that should be added under the `dependencies` clause.

Listing 5.24: TOP dependency

```
repositories{
    maven {
        // The One Probe - McJty
        url "https://cursemaven.com"
    }
}
dependencies{
    implementation fg.deobf("curse.maven:the-one-probe-
       245211:3671753")
}
```

5.12.2. *Just Enough Items* - JEI

JEI[20] interacts with recipes and since this mod adds new recipes to the game. Even if the recipes were added to the same registry as the *Vanilla* recipes, the *JEI* mod cannot properly show them, since there is no texture and no definition for them inside the specified (i.e., JEI) mod. As such, the author of the mod provides a couple of interfaces that should be implemented together with annotations that need to be added in order to integrate any mod with it.

Figure 5.18 shows how the `JEIArmorTinkersPlugin` implements the `IModPlugin` provided by the JEI mod this time (not Forge). This plugin class adds the all the recipe categories to JEI, while also registering the actual recipes (the ones written in `.json` files).

Figure 5.18: Integration classes diagram

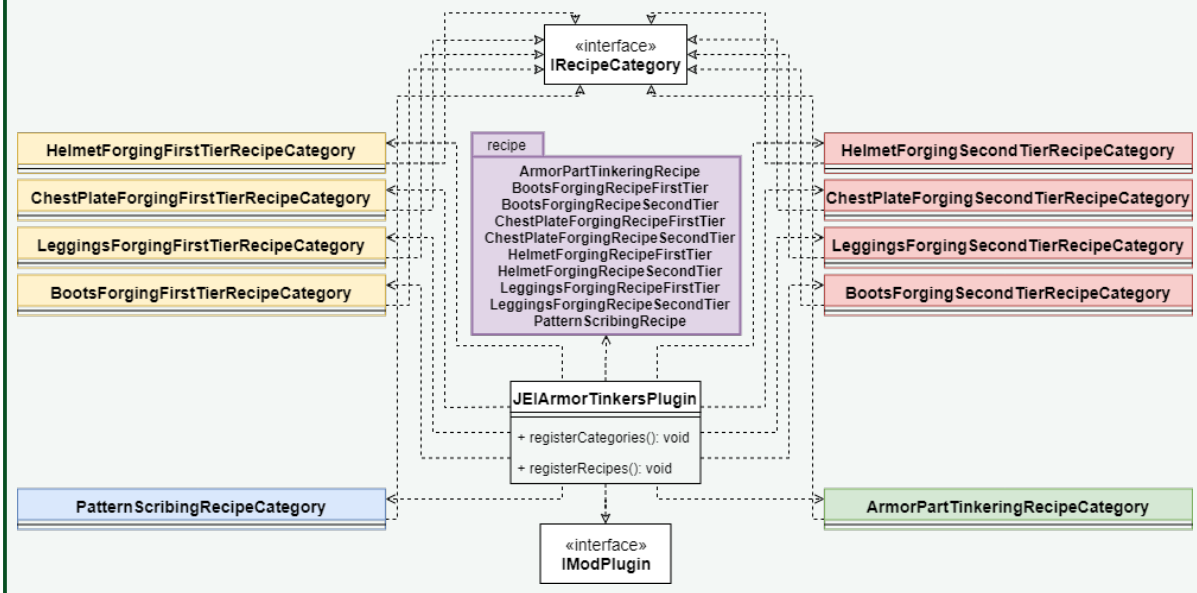

Figure 5.18: Integration classes diagram

Each recipe category implements the `IRecipeCategory` interface, provided by JEI. Both the plugin class and the recipe category classes use the recipe classes presented in Section 5.10 in order to get the recipe type (e.g., `armortinkers:pattern_scribing`). All the recipe category classes also have a background, which is a texture that looks similar to the one provided for the screen class of the block in which the recipe can be performed. Additionally, the slots are added in a similar manner to how they were added for the menu classes.

5.13. Automation for data generation

As the mod grows in size and it adds more items and blocks, it becomes unfeasible to write the configuration files by hand. For each item added, there are 3 `.json` files needed, while for each block added, there are 6 `.json` files needed. Without performing any computation, it can easily be seen how the number of files needed grows faster than the number of actual items or blocks added. As such, I implemented some automation mechanics for these repetitive tasks.

5.13.1. Data generators for `ArmorPartItem` item model and for recipes

In order to solve the problem of creating recipe files and also item and block models, Forge provides the developer with access to data generations classes used by the game. With the help of the library, I implemented the `DataGenerators` class, which subscribes to an event bus, with the method `gatherData()`. Figure 5.19 shows all the classes involved in the data generation process.

Figure 5.19: Data generator classes diagram

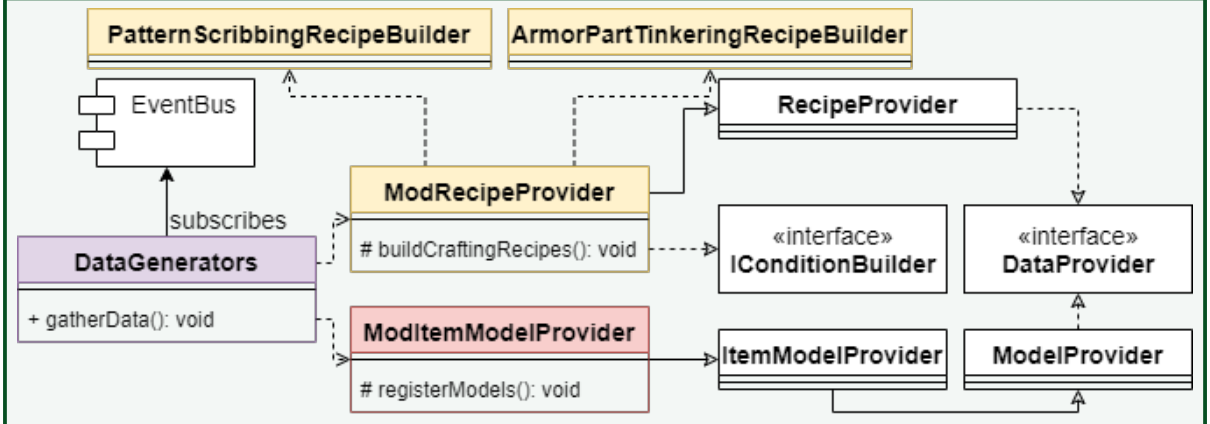

Figure 5.19: Data generator classes diagram

For this mod, recipe and item model data generators were designed and implemented. It can be seen in the picture that the `ModRecipeProvider` class uses the two recipe builder classes for pattern scribing and armor part tinkering. Listing 5.25 shows how two recipes (one for pattern scribing and one for armor part tinkering) are used to generate those recipes automatically for any item or block. It can easily be seen that the items on each recipes are simply added, in order, in the resulting file, while the result is listed last.

Listing 5.25: `ModRecipeProvider.java` file

```
new PatternScribingRecipeBuilder(Items.CYAN_DYE,
    ModItems.BLANK_GOLD_PATTERN.get(),
    ModItems.BOOT_PLATE_GOLD_PATTERN.get(), 1)
    .unlockedBy("has_blank_gold_pattern", inventoryTrigger(
       ItemPredicate.Builder.item()
        .of(ModItems.BLANK_GOLD_PATTERN.get()).build()))
    .save(pFinishedRecipeConsumer);
new ArmorPartTinkeringRecipeBuilder(ModItems.IRON_HAMMER.get(),
    ModItems.HEAD_PLATE_GOLD_PATTERN.get(),
    ModItems.MAGNESIUM_INGOT.get(),
    ModItems.MAGNESIUM_HEAD_PLATE.get(), 1)
    .unlockedBy("has_head_plate_gold_pattern", inventoryTrigger(
       ItemPredicate.Builder.item()
        .of(ModItems.HEAD_PLATE_GOLD_PATTERN.get()).build()))
    .save(pFinishedRecipeConsumer);
```

The generation of item models was primarily used for adding models for all the armor parts. Since all the armor parts point to the same 8 template models (which, in turn, point to the same 8 base, grayscale textures), the model creation for this items is implemented as a switch-case, based on which item is currently being processed.

Remark 5.19: Limitations of Java data generators

It can easily be seen that the developer still has to write java code, in order to generate the configuration files. As such, this automation strategy is still unsuited for generating the files necessary for all the armor pieces added by the mod.

5.13.2. Data generation for helmets, chestplates, leggings and boots

Since this mod adds hundreds of thousands of armor pieces (helmets, chestplates, leggings and boots), I decided to look for another solution in terms of automating the generation of files. As such, I implemented Python [14] scripts that create the necessary files automatically, without having to write any extra code. Listing A.15 shows an example of such a python script for automating the generation of helmet items made out of soft materials. Additionally, these scripts also generate the declarations that needed to be added in the `ModItems` class, thus making it easier to just copy and paste them.

Chapter 6. Testing and Validation

6.1. World generation

World generation is tested by creating a new world and checking how the ores and trees are spawning. In order to run this test case, I set up a new *Minecraft* survival world and then join it. It is worth specifying that the testing for the world generation is done subjectively, by checking if the new tree type and ores appear in the world at all.

Tree generation test For the tree generation test, I checked the seed of the world using the `/seed` command in chat. I then used the returned value on the Biome Finder [21] app, in order to find a *tundra* biome. This biome is one of the biomes that should spawn the *maple* tree.

EXPECTED RESULTS The maple tree should spawn in the tundra biome. The mapple tree should not spawn in the biomes where spruce would spawn.

OBSERVED RESULTS The maple tree spawns only in the colder biomes (as seen in Figures B.25 and B.26), and not in hotter biomes.

Ore generation test For the ore generation test, the process is quite similar. After joining the new world, I enter the game in spectator mode, using the command `/gamemode spectator`. I also give myself the *night vision* effect so I can see better in the dark. The reason I enter the spectator game mode is to be able to easily go through blocks in the search of ores.

EXPECTED RESULTS All ores should be found under ground, at levels lower than 80 (in general). Stone variants of the ores should appear only within stone clumps. The same should be true for the deep slate variants.

OBSERVED RESULTS All ores are found at levels lower than 80 (as it can be seen in Figures B.27 and B.28). Additionally, ores of the stone variant are found only in stone clumps, while ores of the deep slate variant are found only in deep slate clumps.

Remark 6.1: First test failed

The first test for the ore generation mechanism ended up generating too much tin (as it can be seen in Figures B.29 and B.30). The reason was that I accidentally registered the placed features of both aluminium and tin ores under the name of the placed feature for the tin ore. Forge does not throw any exceptions in this case, since the name is used to simply perform all the ore generation of the same type at the same time. Thus instead of generating aluminium ores, the game kept generating tin ores.

6.2. Integration

Integration is tested in a similar manner to world generation. A new world is created and joined.

TOP integration test The TOP integration can be tested very easily, by simply looking at a block added by the mode, inside the game.

EXPECTED RESULTS The tool tip in the corner of the screen should display the name of the block and the name of the mod, *Armor Tinkers*.

OBSERVED RESULTS The tool tip in the corner of the screen displays the name of the block (*Aluminium Block*, as seen in Figure B.31) and the name of the mod, *Armor Tinkers*.

JEI integration test The JEI integration can be tested by opening the inventory and pressing 'R' while hovering over any of the patterns, armor parts or armor pieces added by the mod.

EXPECTED RESULTS The menu should display an index with all the items and blocks in the game, including the ones added by my mod, *Armor Tinkers*. Upon pressing 'R' while hovering over any of the patterns, armor parts or armor pieces added by the mod, the recipe window (of the appropriate type) should open and the correct custom crafting recipe should be displayed.

OBSERVED RESULTS The menu shows the index with all the items and blocks in the game, including the ones added by my mod. The JEI mod displays the correct custom crafting recipe for each new recipe type added.

6.3. Testing automation

While picking the appropriate colors for all the armor parts added to the game, I realised that it takes too long to wait for the game to load each time I want to check a how a color code that I chose looks in the game. As such, I implemented a Python [14] script that automates this process and colors all the template textures in the same way *Minecraft* used (based on the same formula). This sped up the process, by not having to wait for the whole game to load. Instead, I ran the script and checked if the color looks good and is suitable as the color of the respective material. Listing A.16 shows how the script was implemented.

The formula that shows how these parts are colored is presented in Equation 6.1. The *multiplier* the one set for the color of the item, while the *oldPixel* variable is the value of the pixel from the template (gray-scale) image.

Equation 6.1: Coloring formula [22]

$$\text{newPixel} = \frac{\text{oldPixel} * \text{multiplier}}{255}$$

6.4. Crafting an armor piece from start to finish

Crafting a complete armor set is the best way to make sure that the mod is playable from start to finish and that no continuity errors appear.

It is worth mentioning beforehand that, for all the steps that are going to be presented in this section, the expected result matched the observed result, thus arriving to a correct and complete solution for the problem stated in the beginning of the document.

For all the other armor pieces (chestplate, leggings and boots) the testing process is similar. The only difference comes in the number of parts used for the respective armor piece. For chestplate, there are 5 armor parts, for leggings, there are 3 armor parts, and for boots, there are 2 armor parts.

Figure 6.1: Test case for crafting a helmet

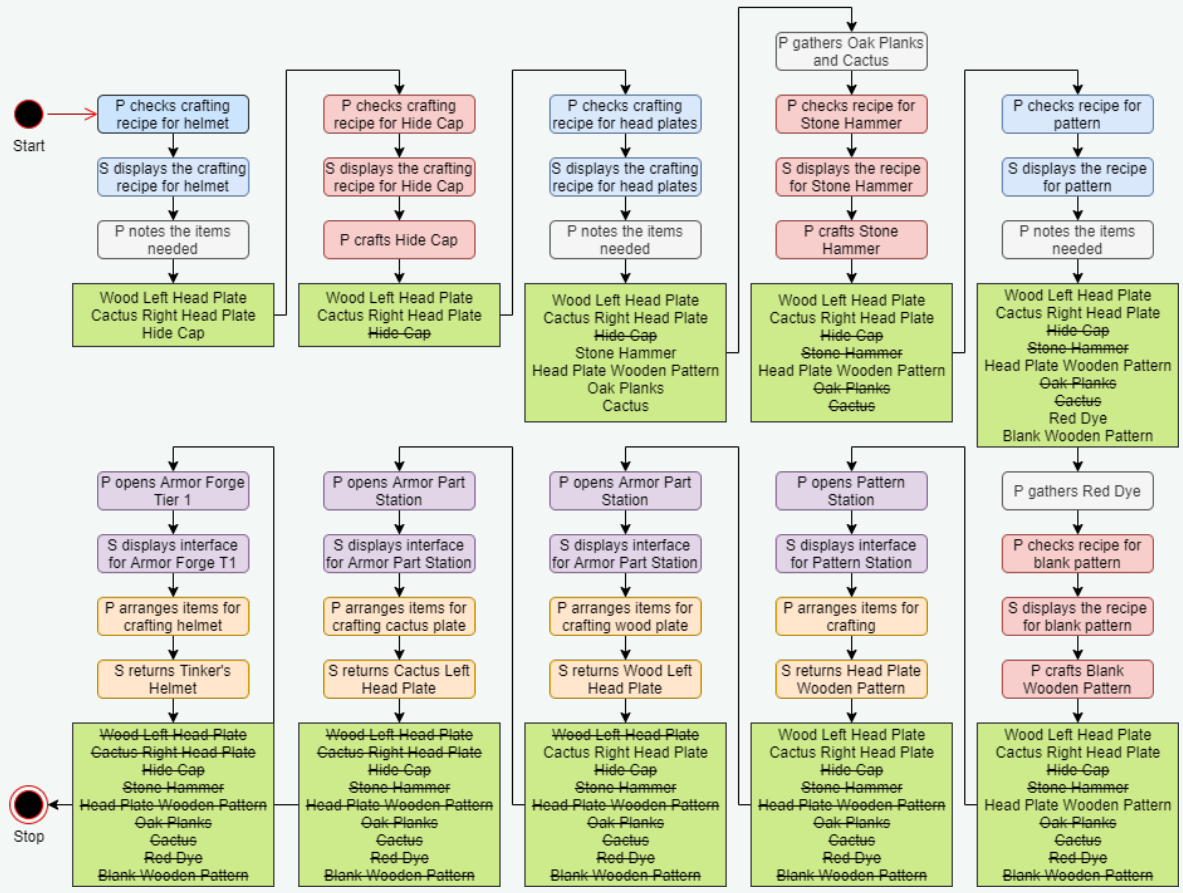

Figure 6.1: Test case for crafting a helmet

Figure 6.1 show the steps for crafting a helmet from start to finish. On this image, the green rectangles represents the list of items that the player needs to craft or gather, in order to obtain the final item (i.e., the helmet). Items are added to the list every time the player checks a new recipe type (added by the mod) and stricken off this list when the player successfully crafts the item. Each rounded rectangle is color coded to show what feature of the mod is being tested. Thus, the following color code was used.

List 6.1: Color codes for the test case

- **blue** was used for steps that test the integration of the mod with JEI
- **red** was used for steps that test the continuity of the mod (i.e., the ability to craft items of the mod from *Vanilla* items)
- **purple** was used for steps that test if the new menu types are displayed correctly
- **orange** was used for steps that test the new crafting recipe types
- **grey** was used for steps that do not test anything

Chapter 7. User's manual

This section presents how the user can install and play with the *Armor Tinkers* mod. The use of this mod is intuitive from within the game, since all recipes are provided inside the game, with the help of the JEI integration module.

7.1. Prerequisites

The user should already have the following software installed and any other necessary steps should already be taken, as described by the following list.

List 7.1: Prerequisites

- Operating system: Windows or macOS
- Minecraft version 1.18.2 already installed
- Minecraft online account (i.e., payed)
- Forge Mod Loader version 40.1.0
- Connection to internet, to download all the necessary dependencies

7.2. Installation steps

In order to install the *Armor Tinkers* mod, the user needs to add the `.jar` file inside the folder `mods`, created by Forge Mod Loader after installation. This scenario works only if the player wants to add this mod without any other mods present.

As it is usually the case with every mod created, if a user wants this mod to be part of a bigger modpack, special care should be taken while adding it. It can conflict with an unpredictable number of mods. For this situation, it is recommended that the mod is cloned directly from the github page and modifications are made inside the configuration files of the mod, in order to solve any conflicts. This is the usual process in which a modpack is put together. After modifying the code, a new `.jar` file should be created and placed inside the same `mods` folder.

After installing the mod, the user can open *Minecraft* and create a new world. Any resources added by the mod will be instantly loaded to the new world created and the player can start exploring the mechanics added by *Armor Tinkers*.

Chapter 8. Conclusions

To sum up, this project aimed to change the armor system and armor mechanics of the game, by adding armor pieces that can be crafted in a modular way, from multiple materials at a time. It also added new materials (such as magnesium, zinc and others) and provided functionalities to use materials that already existed in the game (such as wood, cactus or even iron and gold) to be used in the creation of the new armor types added. Lastly, in the development of this project, the need for automation was satisfied.

8.1. Contributions and achievements

The objectives and features of the mod presented in Chapter 2 have been accomplished successfully, as presented in the following paragraphs.

Adding new materials The *Armor Tinkers* mod adds the following materials: aluminium, tin, silver, lead, magnesium, zinc, brass, bronze, vibranium and adamantium. These materials are used directly in armor crafting. Additionally, it also adds resin, which is used both in crafting and as a fuel.

Providing usage for existing materials The *Armor Tinkers* mod provides similar usage in the new armor crafting mechanics for materials already existing in the game. This materials are: wood, cactus, stone, netherrack, end stone, obsidian, copper, iron and gold.

Generation of materials The *Armor Tinkers* mod implements generation mechanics for the following materials: aluminium, tin, silver, lead, magnesium, zinc, specified in the paper under the name of *Ore generation*, and also resin, specified in the paper under the name of *Tree generation*

Crafting recipes for new materials The *Armor Tinkers* mod adds standard crafting recipes for the materials that do not generate naturally in the world, such as: bronze, brass, adamantium and vibranium.

Continuity The *Armor Tinkers* mod provides continuity for all of the items and blocks it adds. This means that any item that is not generated naturally in the world, can be crafted out of items that already exist in *Minecraft*.

New block entities with new menus The *Armor Tinkers* mod adds four new block entities for the four stations available in the mod: pattern station, armor part tinkering station, armor forge tier 1 and armor forge tier 2. This entities also come with new menu types and screen types.

New crafting recipes for armor The *Armor Tinkers* mod provides for the player four new crafting recipe types, one for each crafting station presented previously: pattern scribing, armor part tinkering, armor forging tier 1 and armor forging tier 2.

Integration with JEI and TOP The *Armor Tinkers* mod comes with integration for TOP and JEI. TOP is used to help the player differentiate between the blocks added by the mod, while JEI is used to discover and perform the new crafting recipes added by the game.

New armor types The *Armor Tinkers* mod adds four new pieces of armor. In reality, there are more than four pieces, because each one of them is a combination of the materials that were added to the game.

Custom attribute values The *Armor Tinkers* mod provides custom attribute values for each armor piece added to the game, by implementing a set of individual attributes for each new material added. As such, based on the materials from which an armor piece is crafted, it will have different attribute values, computed dynamically.

Automation for data generation The project implemented two means for automatic data generation: one in Java, using the classes offered by the Forge API and another (more efficient) one in Python, based on the templates that are needed for each individual resource file type.

8.2. Analysis of the results

Table 8.1 shows a comparison between the mods presented in Section 3.5. It follows the same features as presented there, while also comparing them to the *Armor Tinkers* mod.

Table 8.1: Comparison between current solutions and Armor Tinkers

Feature	Bot	Mek	Apo	TC	AT
New materials	YES	YES	NO	YES	YES
Generated materials	NO	YES	-	YES	YES
Crafted materials	YES	YES	-	NO	YES
Has GUIs	NO	YES	NO	YES	YES
New recipe types	NO	YES	NO	YES	YES
New blocks	YES	YES	YES	YES	YES
New items	YES	YES	YES	YES	YES
New block entities	YES	YES	NO	YES	YES
New armor	YES	YES	NO	NO	YES
Modular crafting	NO	NO	NO	YES	YES
New effects	NO	NO	YES	NO	NO

8.3. Further development

There are multiple ways in which this mod can be developed further. They will be presented in the next paragraphs.

Custom effects The mod could add custom effects for each new material type it adds, similar to the system implemented by *Tinkers' Construct*. The effects will than transfer to the individual armor piece, based on the materials used in the crafting recipe.

Structure generation The mod could add new structures (similar to villager houses or to dungeons) that spawn naturally throughout the world and provide the player with either a challenge and some good loot or just with some medium level loot.

New entities The mod could add a new villager, such as a *black smith* or an *armourer* that can have custom trades and interract with the player.

Bibliography

[1] "History of minecraft," https://www.thescienceacademystemmagnet.org/2019/12/20/the-history-of-minecraft/. [Online]. Available: https://www.thescienceacademystemmagnet.org/2019/12/20/the-history-of-minecraft/

[2] A. Gupta and A. Gupta, *Minecraft Modding with Forge*, 1st ed., B. Forster and B. MacDonald, Eds. O'Reilly Media Inc., Apr. 2015. [Online]. Available: https://www.oreilly.com/library/view/minecraft-modding-with/9781491918883/

[3] D. Goldberg, L. Larsson, and M. Persson, *Minecraft: the unlikely tale of Markus "Notch" Persson and the game that changed everything*, 1st ed. Seven Stories Press, 2013. [Online]. Available: libgen.li/file.php?md5=b0d32cb70c392fc379d7e83897a5918f

[4] "Minecraft timeline," https://time.graphics/line/203698. [Online]. Available: https://time.graphics/line/203698

[5] "Minecraft wiki," https://minecraft.fandom.com/wiki. [Online]. Available: https://minecraft.fandom.com/wiki

[6] C. Zorn, C. A. Wingrave, E. Charbonneau, and J. J. LaViola Jr, "Exploring minecraft as a conduit for increasing interest in programming," in *Foundation of Digital Games*, 2013, pp. 352–359. [Online]. Available: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.381.4045&rep=rep1&type=pdf

[7] E. Shaw, M. T. La, R. Phillips, and E. B. Reilly, "Play minecraft! assessing secondary engineering education using game challenges within a participatory learning environment," in *2014 ASEE Annual Conference & Exposition*, 2014, pp. 24–34. [Online]. Available: https://peer.asee.org

[8] G. Woblewski, "General method of program code obfuscation," *unknown*, 2002. [Online]. Available: https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.19.8630

[9] C. Colleberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," University of Auckland, Department of Computer Science, Auckland, New Zealand, techreport, 1997. [Online]. Available: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.68.2651&rep=rep1&type=pdf

[10] P. Killarney, "Modding history," https://www.packt.com/brief-history-minecraft-modding/. [Online]. Available: https://www.packt.com/brief-history-minecraft-modding/

[11] "Forge documentation," https://docs.minecraftforge.net/en/1.18.x/. [Online]. Available: https://docs.minecraftforge.net/en/1.18.x/

[12] "Fabric documentation," https://github.com/hyperledger/fabric-docs-i18n. [Online]. Available: https://github.com/hyperledger/fabric-docs-i18n

[13] "Fabric wiki," https://fabricmc.net/wiki/tutorial:introduction. [Online]. Available: https://fabricmc.net/wiki/tutorial:introduction

[14] G. Van Rossum and F. L. Drake, *Python 3 reference manual*. CreateSpace, 2009. [Online]. Available: http://marvin.cs.uidaho.edu/Teaching/CS515/pythonReference.pdf

[15] "Armor textures when being worn," https://www.jehkoba.com/jehkobas-fantasy.php. [Online]. Available: https://www.jehkoba.com/jehkobas-fantasy.php

[16] K. Arnold, J. Gosling, and D. Holmes, *The Java programming language*, 4th ed. Addison Wesley Professional, 2005. [Online]. Available: https://www.acs.ase.ro/Media/Default/documents/java/ClaudiuVinte/books/ArnoldGoslingHolmes06.pdf

[17] E. Gamma, R. Helm, R. Johnson, R. E. Johnson, J. Vlissides *et al.*, *Design patterns: elements of reusable object-oriented software*. Pearson Deutschland GmbH, 1995. [Online]. Available: http://www.javier8a.com/itc/bd1/articulo.pdf

[18] J. Kaupen, "Modding tutorial lecture (75)," https://www.udemy.com/course/minecraft-modding-forge-118/learn. [Online]. Available: https://www.udemy.com/course/minecraft-modding-forge-118/learn/lecture/29961058

[19] McJty, "The one probe," https://github.com/McJtyMods/TheOneProbe/wiki. [Online]. Available: https://github.com/McJtyMods/TheOneProbe/wiki

[20] Mezz, "Just enough items," https://github.com/mezz/JustEnoughItems. [Online]. Available: https://github.com/mezz/JustEnoughItems

[21] "Biome finder," https://www.chunkbase.com/apps/biome-finder. [Online]. Available: https://www.chunkbase.com/apps/biome-finder

[22] "Forge item coloring," https://docs.minecraftforge.net/en/1.14.x/models/color/. [Online]. Available: https://docs.minecraftforge.net/en/1.14.x/models/color/

Appendix A. Relevant code sections

Listing A.1: ModBlocks class

```
// Register the block created
private static <T extends Block> RegistryObject<T>
registerBlock(String name,  Supplier<T> block,
              CreativeModeTab tab) {
    RegistryObject<T> toReturn = BLOCKS.register(name, block);
    registerBlockItem(name, toReturn, tab);
    return toReturn;
}

// Register the item associated with the block
private static <T extends Block> RegistryObject<Item>
registerBlockItem(String name, RegistryObject<T> block,
                  CreativeModeTab tab) {
    return ModItems.ITEMS
        .register(name, () -> new BlockItem(block.get(),
            new Item.Properties().tab(tab)));
}
```

Listing A.2: `model/item/aluminium_block.json` **file**

```
{
  "type": "minecraft:crafting_shaped",
  "pattern": [
    "###",
    "###",
    "###"
  ],
  "key": {
    "#": {
      "item": "armortinkers:aluminium_ingot"
    }
  },
  "result": {
    "item": "armortinkers:aluminium_block"
  }
}
```

Listing A.3: loot_tables/blocks/aluminium_block.json **file**

```
{
  "type": "minecraft:block",
  "pools": [
    {
      "rolls": 1.0,
      "bonus_rolls": 0.0,
      "entries": [
        {
          "type": "minecraft:item",
          "name": "armortinkers:aluminium_block"
        }
      ],
      "conditions": [
        {
          "condition": "minecraft:survives_explosion"
        }
      ]
    }
  ]
}
```

Listing A.4: model/block/aluminium_block.json **file**

```
{
  "parent": "block/cube_all",
  "textures": {
    "all" : "armortinkers:block/aluminium_block"
  }
}
```

Listing A.5: mineable/pickaxe.json **file**

```
{
    "replace": false,
    "values": [
        "armortinkers:aluminium_block"
    ]
}
```

Listing A.6: needs_stone_tool.json **file**

```
{
    "replace": false,
    "values": [
        "armortinkers:aluminium_block"
    ]
}
```

Listing A.7: ArmorPartItem.java file

```
public int getMaterialColor() {
    return materialColor;
}
public ModMaterial getMaterial() {
    return material;
}
@Mod.EventBusSubscriber(value = Dist.CLIENT,
                        modid = ArmorTinkers.MOD_ID,
                        bus = Mod.EventBusSubscriber.Bus.MOD)
private static class ColorRegisterHandler {
    @SubscribeEvent(priority = EventPriority.HIGHEST)
    public static void registerSpawnEggColors(ColorHandlerEvent.
       Item event) {
        MOD_ARMOR_PARTS.forEach(armorPartItem ->
                event.getItemColors().register(
                    (stack, layer) -> armorPartItem.
                       getMaterialColor(),
                    armorPartItem)
        );
    }
}
```

Listing A.8: PatternStationBlockEntity.java file

```
private static boolean hasRecipe(PatternStationBlockEntity
   entity) {
    Level level = entity.level;
    SimpleContainer inventory =
        new SimpleContainer(entity.itemHandler.getSlots());

    for (int i = 0; i < entity.itemHandler.getSlots(); i++) {
        inventory.setItem(i,
            entity.itemHandler.getStackInSlot(i));
    }

    // Do the items in slots match any recipe of
       PatternScribingRecipe type
    Optional<PatternScribingRecipe> match = level
            .getRecipeManager()
            .getRecipeFor(PatternScribingRecipe.Type.INSTANCE,
                          inventory, level);

    return match.isPresent() && canInsertItemIntoOutputSlot(
       inventory);
}
```

Listing A.9: PatternStationBlockEntity.java file

```
private static void craftItem(PatternStationBlockEntity entity){
    Level level = entity.level;
```

```
    SimpleContainer inventory =
        new SimpleContainer(entity.itemHandler.getSlots());
    for (int i = 0; i < entity.itemHandler.getSlots(); i++) {
        inventory.setItem(i, entity.itemHandler.getStackInSlot(i
           ));
    }

    Optional<PatternScribingRecipe> match = level
            .getRecipeManager()
            .getRecipeFor(PatternScribingRecipe.Type.INSTANCE,
                          inventory, level);

    if (match.isPresent()) {
        entity.itemHandler.setStackInSlot(2, new ItemStack(match
           .get().getResultItem().getItem(), 1));
    }
}
```

Listing A.10: `PatternStationBlockEntity.java` **file**

```
private static void extractIngredients(PatternStationBlockEntity
    entity){
    entity.itemHandler.extractItem(0, 1, false);
    entity.itemHandler.extractItem(1, 1, false);

    entity.resetProgress();
}
```

Listing A.11: Helper methods

```
private void addPlayerInventory(Inventory playerInventory) {
    for (int i = 0; i < 3; ++i) {
        for (int l = 0; l < 9; ++l) {
            this.addSlot(
                new Slot(playerInventory,
                    l + i * 9 + 9, 8 + l * 18, 86 + i * 18));
        }
    }
}

private void addPlayerHotbar(Inventory playerInventory) {
    for (int i = 0; i < 9; ++i) {
        this.addSlot(
            new Slot(playerInventory,
                i, 8 + i * 18, 144));
    }
}
```

Listing A.12: Menu registration

```
public static final DeferredRegister<MenuType<?>> MENUS =
```

```
    DeferredRegister.create(ForgeRegistries.CONTAINERS,
        ArmorTinkers.MOD_ID);

public static final RegistryObject<MenuType<ArmorPartMakerMenu>>
    ARMOR_PART_MAKER_MENU =
    registerMenuType(ArmorPartMakerMenu::new, "
        armor_part_maker_menu");
```

Listing A.13: ArmorForgeFirstTierMenu.java **class**

```
this.leftHeadPlateSlot = new ModArmorForgeSlot(handler,
    ModArmorForgeSlots.LEFT_HEAD_PLATE, 36, 32, true);
this.rightHeadPlateSlot = new ModArmorForgeSlot(handler,
    ModArmorForgeSlots.RIGHT_HEAD_PLATE, 58, 32, true);
this.hideCapSlot = new ModArmorForgeSlot(handler,
    ModArmorForgeSlots.HIDE_CAP, 80, 32, true);

this.leftShoulderPlateSlot = new ModArmorForgeSlot(handler,
    ModArmorForgeSlots.LEFT_SHOULDER_PLATE, 14, 21, false);
this.rightShoulderPlateSlot = new ModArmorForgeSlot(handler,
    ModArmorForgeSlots.RIGHT_SHOULDER_PLATE, 14, 43, false);
this.frontPlateSlot = new ModArmorForgeSlot(handler,
    ModArmorForgeSlots.FRONT_PLATE, 36, 32, false);
this.backPlateSlot = new ModArmorForgeSlot(handler,
    ModArmorForgeSlots.BACK_PLATE, 58, 32, false);
this.wristBandSlot = new ModArmorForgeSlot(handler,
    ModArmorForgeSlots.WRIST_BAND, 80, 54, false);
this.hideVestSlot = new ModArmorForgeSlot(handler,
    ModArmorForgeSlots.HIDE_VEST, 80, 32, false);

this.leftLegPlateSlot = new ModArmorForgeSlot(handler,
    ModArmorForgeSlots.LEFT_LEG_PLATE, 36, 21, false);
this.rightLegPlateSlot = new ModArmorForgeSlot(handler,
    ModArmorForgeSlots.RIGHT_LEG_PLATE, 36, 43, false);
this.tailPlateSlot = new ModArmorForgeSlot(handler,
    ModArmorForgeSlots.TAIL_PLATE, 80, 32, false);
this.hidePantsSlot = new ModArmorForgeSlot(handler,
    ModArmorForgeSlots.HIDE_PANTS, 58, 32, false);

this.leftBootPlateSlot = new ModArmorForgeSlot(handler,
    ModArmorForgeSlots.LEFT_BOOT_PLATE, 36, 32, false);
this.rightBootPlateSlot = new ModArmorForgeSlot(handler,
    ModArmorForgeSlots.RIGHT_BOOT_PLATE, 80, 32, false);
this.hideSocksSlot = new ModArmorForgeSlot(handler,
    ModArmorForgeSlots.HIDE_SOCKS, 58, 32, false);

this.addSlot((leftHeadPlateSlot));
this.addSlot((rightHeadPlateSlot));
this.addSlot((hideCapSlot));

this.addSlot((leftShoulderPlateSlot));
```

```
26 this.addSlot((rightShoulderPlateSlot));
27 this.addSlot((frontPlateSlot));
28 this.addSlot((backPlateSlot));
29 this.addSlot((wristBandSlot));
30 this.addSlot((hideVestSlot));
31
32 this.addSlot((leftLegPlateSlot));
33 this.addSlot((rightLegPlateSlot));
34 this.addSlot((tailPlateSlot));
35 this.addSlot((hidePantsSlot));
36
37 this.addSlot((leftBootPlateSlot));
38 this.addSlot((rightBootPlateSlot));
39 this.addSlot((hideSocksSlot));
40 this.addSlot(new ModResultSlot(handler, ModArmorForgeSlots.
      RESULT_SLOT, 126, 32));
41 this.addSlot(new ModResultSlot(handler, ModArmorForgeSlots.
      HELMET_BUTTON, 178, 0));
42 this.addSlot(new ModResultSlot(handler, ModArmorForgeSlots.
      CHESTPLATE_BUTTON, 178, 18));
43 this.addSlot(new ModResultSlot(handler, ModArmorForgeSlots.
      LEGGINGS_BUTTON, 178, 36));
44 this.addSlot(new ModResultSlot(handler, ModArmorForgeSlots.
      BOOTS_BUTTON, 178, 54));
```

Listing A.14: Recipe registration

```
1 public static final DeferredRegister<RecipeSerializer<?>>
      SERIALIZERS =
2     DeferredRegister.create(ForgeRegistries.RECIPE_SERIALIZERS,
          ArmorTinkers.MOD_ID);
3
4 public static final RegistryObject<RecipeSerializer<
      ArmorPartTinkeringRecipe>> ARMOR_PART_MAKER_SERIALIZER =
5     SERIALIZERS.register("armor_part_tinkering", () ->
          ArmorPartTinkeringRecipe.Serializer.INSTANCE);
```

Listing A.15: Automation for data generation

```
1 import sys
2
3 root_dir_models = "C:\\Users\\tudor\\Tudor\\010_MCM_MinecraftMod
      \\010_UsefulScripts\\models\\helmets\\"
4 root_dir_recipes = "C:\\Users\\tudor\\Tudor\\010
      _MCM_MinecraftMod\\010_UsefulScripts\\recipes\\helmets\\"
5 original_stdout = sys.stdout
6
7 materials_lower = ["wood", "cactus", "stone", "netherack", "
      end_stone", "flint"]
8 materials_upper = ["WOOD", "CACTUS", "STONE", "NETHERACK", "
      END_STONE", "FLINT"]
```

```
9
10 # Item registration
11 global_index = 1
12 with open("helmets_registration.txt", 'w') as f:
13     sys.stdout = f
14     for i in range(0, len(materials_upper)):
15         for j in range(0, len(materials_lower)):
16             print("public static RegistryObject<Item>
                   TINKERS_HELMET_%d = ITEMS.register(\"
                   tinkers_helmet_%d\","
17                 "\n() -> new HelmetItem(new Item.Properties().
                       tab(ModCreativeModeTab.HELMET_TAB),\
                       nModMaterial.%s, "
18                 "ModMaterial.%s));" % (global_index,
                       global_index, materials_upper[i],
                       materials_upper[j]))
19             global_index += 1
20
21 # Item lang
22 global_index = 1
23 with open("helmets_lang.txt", 'w') as f:
24     sys.stdout = f
25     for i in range(0, len(materials_upper)):
26         for j in range(0, len(materials_lower)):
27             print("\"item.armortinkers.tinkers_helmet_%d\": \"
                   Tinker's Helmet\"," % global_index)
28             global_index += 1
29
30 # Item model files
31 global_index = 1
32 for i in range(0, len(materials_upper)):
33     for j in range(0, len(materials_lower)):
34         file_name = root_dir_models + "tinkers_helmet_" + str(
               global_index) + ".json"
35         with open(file_name, 'w') as f:
36             sys.stdout = f
37             print("{\n\t\"parent\": \"armortinkers:item/
                   help_helmet\"\n}")
38         global_index += 1
39
40 # Recipe generation
41 global_index = 1
42 for i in range(0, len(materials_upper)):
43     for j in range(0, len(materials_lower)):
44         file_name = root_dir_recipes + "tinkers_helmet_" + str(
               global_index) + "_from_armor_forging_t1.json"
45         with open(file_name, 'w') as f:
46             sys.stdout = f
47             print("{\n\t\"type\": \"armortinkers:
                   helmet_forging_t1\",\n"
48                 "\t\"ingredients\": [\n"
```

```
                    "\t{\n\t\t\"item\": \"armortinkers:%
                       s_head_plate\"\n\t},\n"
                    "\t{\n\t\t\"item\": \"armortinkers:%
                       s_head_plate\"\n\t},\n"
                    "\t{\n\t\t\"item\": \"armortinkers:hide_cap\"\
                       n\t}\n\t],\n"
                    "\t\"output\": {\n"
                    "\t\t\"item\": \"armortinkers:tinkers_helmet_%
                       d\"\n\t}\n}"
                    % (materials_lower[i], materials_lower[j],
                       global_index))
        global_index += 1
```

Listing A.16: Automation for testing

```
from PIL import Image

MAGNESIUM = 0
WOOD = 1
CACTUS = 2
STONE = 3
NETHERACK = 4
ENDSTONE = 5
FLINT = 6
COPPER = 7
IRON = 8
GOLD = 9
OBSIDIAN = 10
ALUMINIUM = 11
TIN = 12
ZINC = 13
SILVER = 14
LEAD = 15
BRASS = 16
BRONZE = 17
VIBRANIUM = 18
ADAMANTIUM = 19
NO_COLOR = 20

root = "C:\\Users\\tudor\\Tudor\\010_MCM_MinecraftMod\\
   ArmorTinkersMod_1.18.2\\src\\main\\resources\\assets" \
       "\\armortinkers\\textures\\base\\"
images = ["head_plate_base.png",
          "front_plate_base.png",
          "back_plate_base.png",
          "shoulder_plate_base.png",
          "leg_plate_base.png",
          "boot_plate_base.png",
          "tail_plate_base.png",
          "wrist_band_base.png",
          ]
```

```
index = 0
colors = [0xce90c0,     # magnesium
          0x4c3d26,     # wood
          0x39581a,     # cactus
          0x686868,     # stone
          0x501b1b,     # netherack
          0xdee6a4,     # endstone
          0x2e2d2d,     # flint
          0xc15a36,     # copper
          0xd8d8d8,     # iron
          0xfad64a,     # gold
          0x271e3d,     # obsidian
          0xb4dceb,     # aluminium
          0xb5d9f4,     # tin
          0x9baea5,     # zinc
          0x8decfb,     # silver
          0xd2d5f7,     # lead
          0x9c8c00,     # brass
          0xee9d6e,     # bronze
          0x5f36d8,     # vibranium
          0x442a4d,     # adamantium
          0xFFFFFF,     # no color
          ]
materials = ["magnesium", "wood", "cactus", "stone", "netherack"
    , "endstone", "flint", "copper", "iron", "gold", "obsidian",
    "aluminium", "tin", "zinc", "silver", "lead", "brass", "
    bronze", "vibranium", "adamantium", "no_color"]

im = Image.open(root + images[index]).convert('RGB')

r, g, b = im.split()

rc = (colors[ALUMINIUM] & 0xFF0000) / 0x00FFFF
gc = (colors[ALUMINIUM] & 0x00FF00) / 0x0000FF
bc = (colors[ALUMINIUM] & 0x0000FF)

r = r.point(lambda i: i * rc / 255)
g = g.point(lambda i: i * gc / 255)
b = b.point(lambda i: i * bc / 255)

# Recombine back to RGB image
result = Image.merge('RGB', (r, g, b))

result.save("C:\\Users\\tudor\\Tudor\\010_MCM_MinecraftMod\\010
    _UsefulScripts\\images\\" + materials[ALUMINIUM] + "_" +
    images[index])

print("Decimal value: " + str(int(colors[ALUMINIUM])))
```

Appendix B. Figures

Figure B.1: Stone hammer recipe with JEI

Figure B.1: Stone hammer recipe with JEI

Figure B.2: Blocks in the Minecraft world

Figure B.2: Blocks in the Minecraft world

Figure B.3: Passive mobs

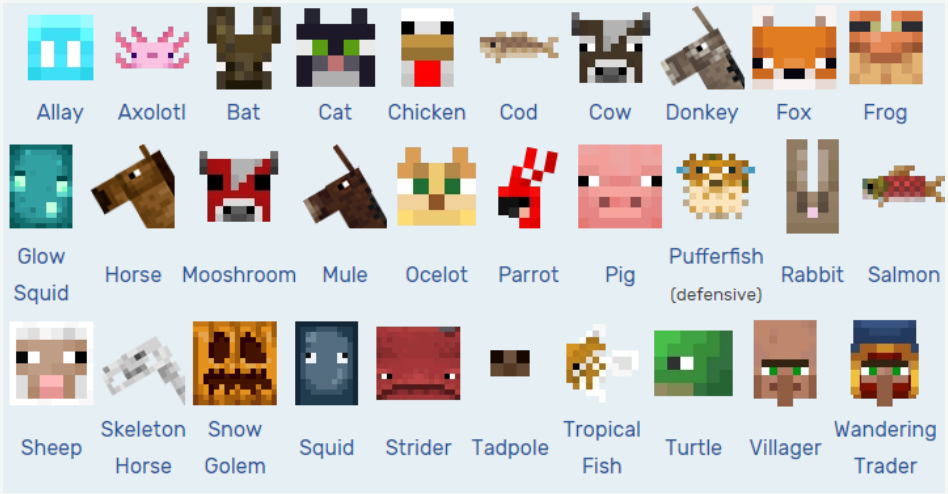

Figure B.3: Passive mobs

Figure B.4: Neutral mobs

Figure B.4: Neutral mobs

Figure B.5: Hostile mobs

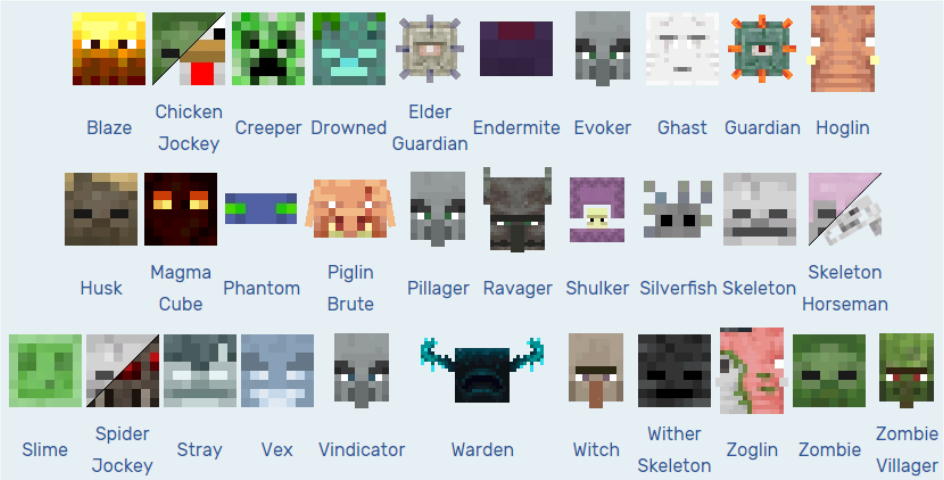

Figure B.5: Hostile mobs

Figure B.6: Bosses

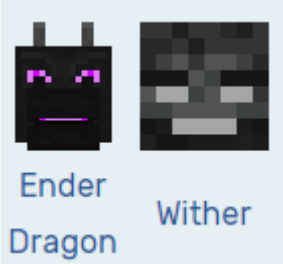

Figure B.6: Bosses

Figure B.7: Crafting table recipe

Figure B.7: Crafting table recipe

Figure B.8: Furnace recipe

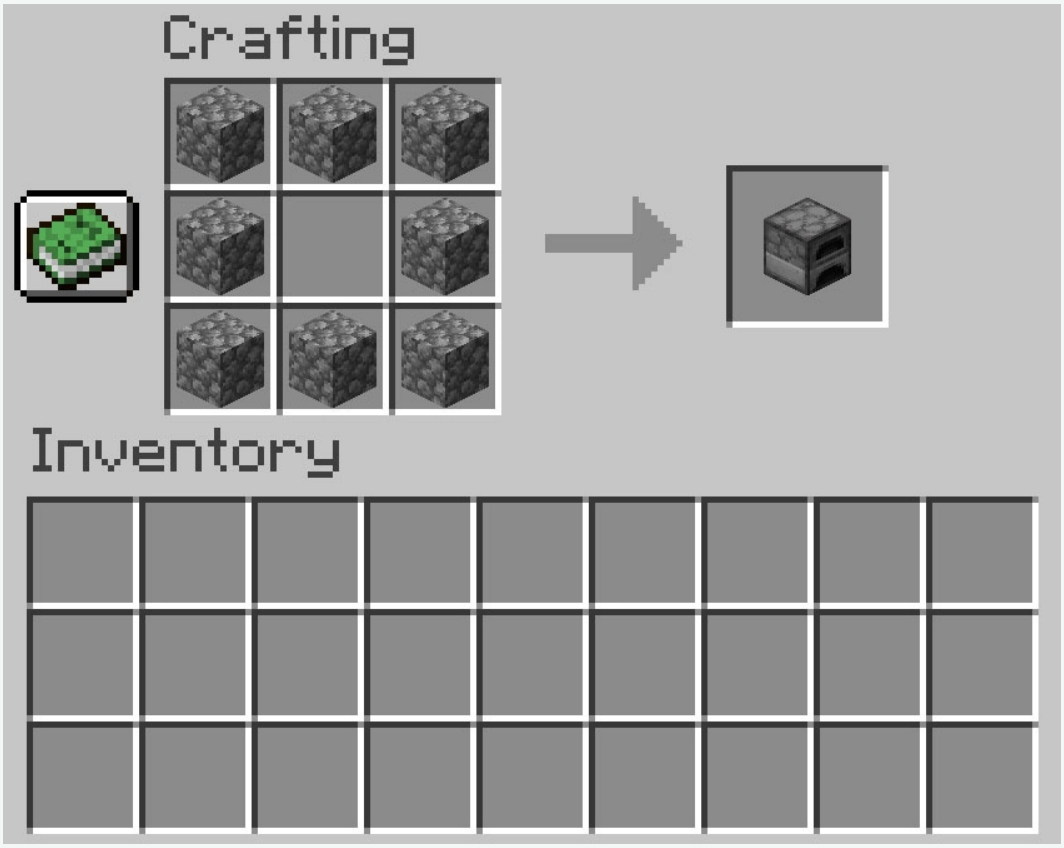

Figure B.8: Furnace recipe

Figure B.9: Armor damage reduction

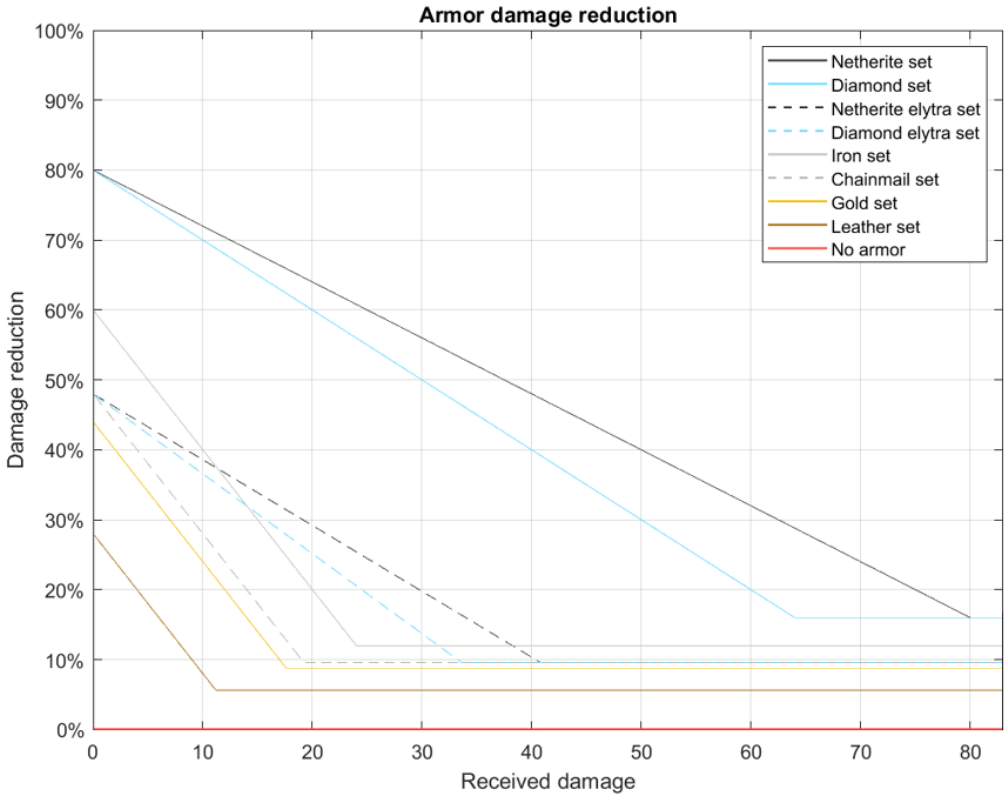

Figure B.9: Armor damage reduction

Figure B.10: *The One Probe* mod in game

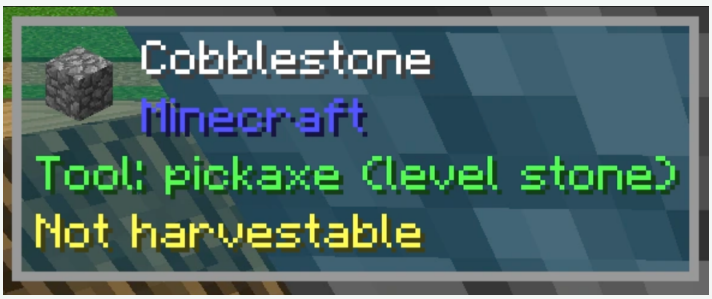

Figure B.10: *The One Probe* mod in game

Figure B.11: *Just Enough Items* mod in game

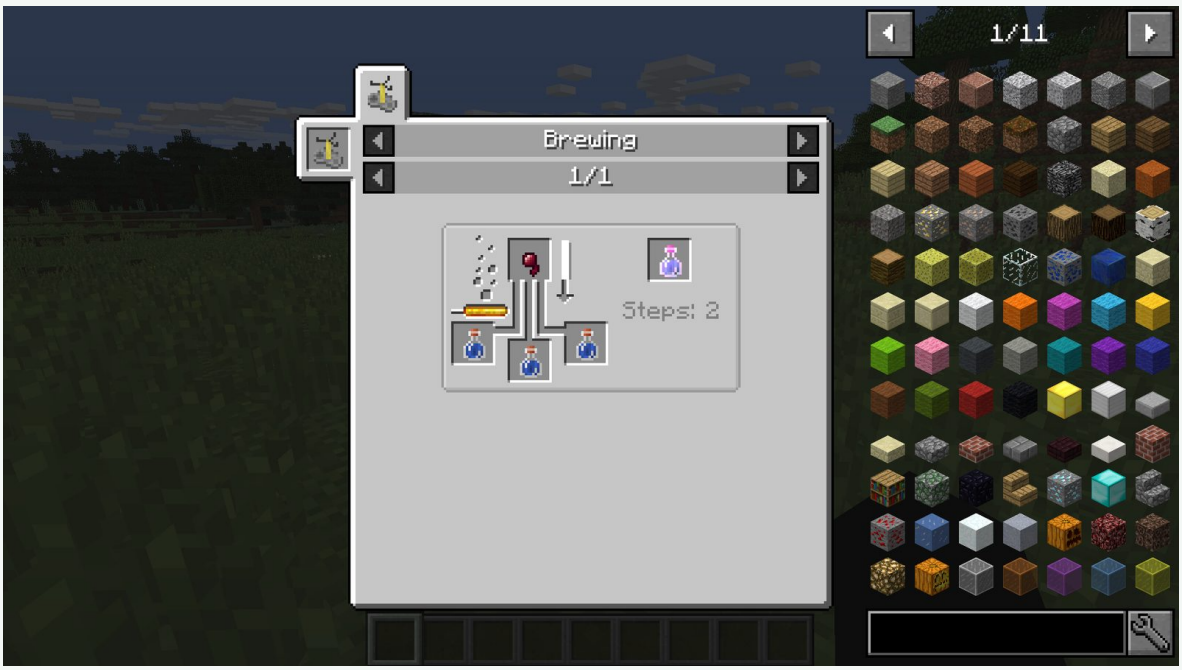

Figure B.11: *Just Enough Items* mod in game

Figure B.12: Aluminium Block recipe

Figure B.12: Aluminium Block recipe

Figure B.13: Tabs for creative mode

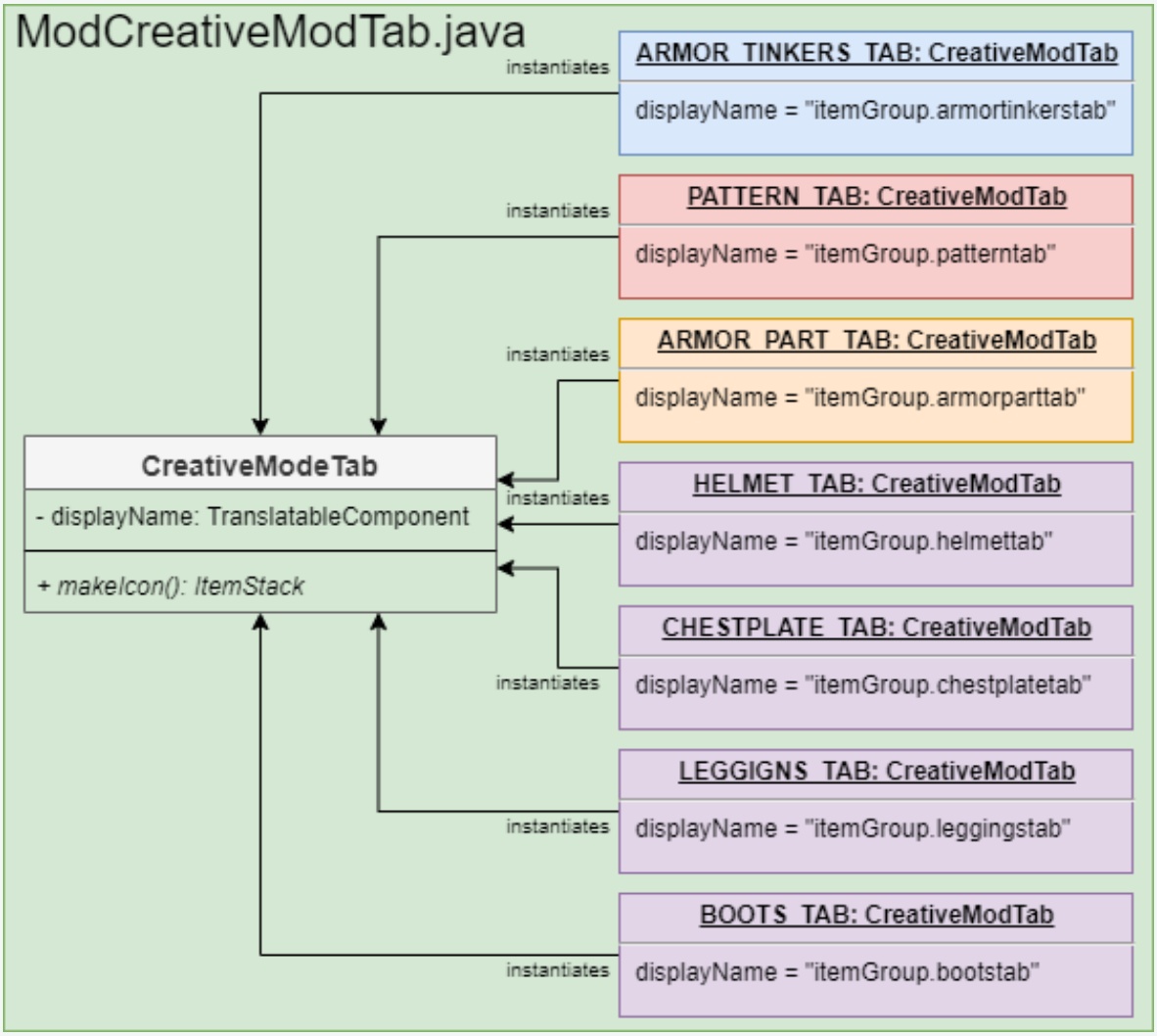

Figure B.13: Tabs for creative mode

Figure B.14: `ChestPlateItem` **class**

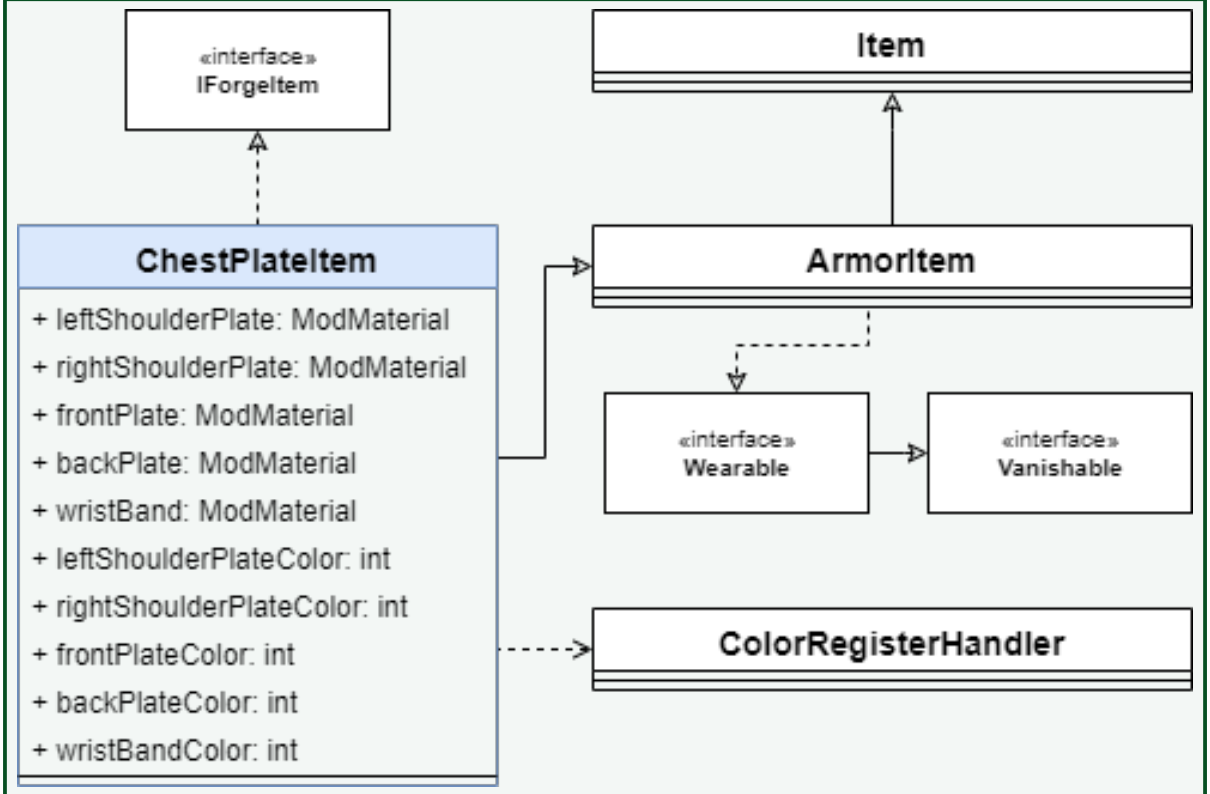

Figure B.14: `ChestPlateItem` class

Figure B.15: `LeggingsItem` **class**

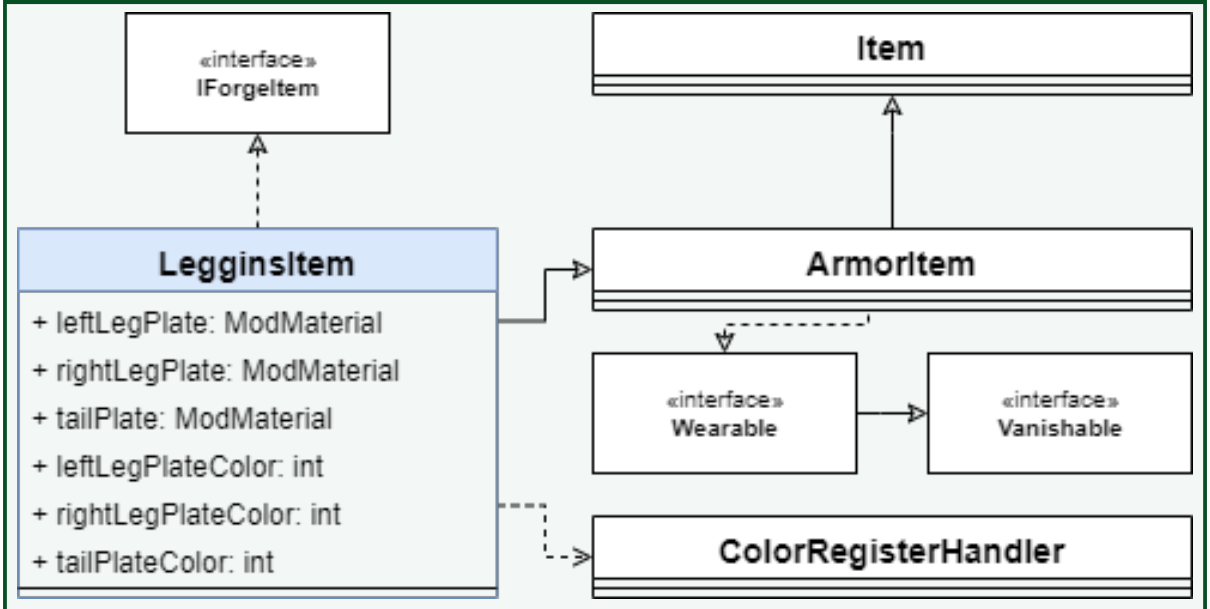

Figure B.15: `LeggingsItem` class

Figure B.16: `BootsItem` class

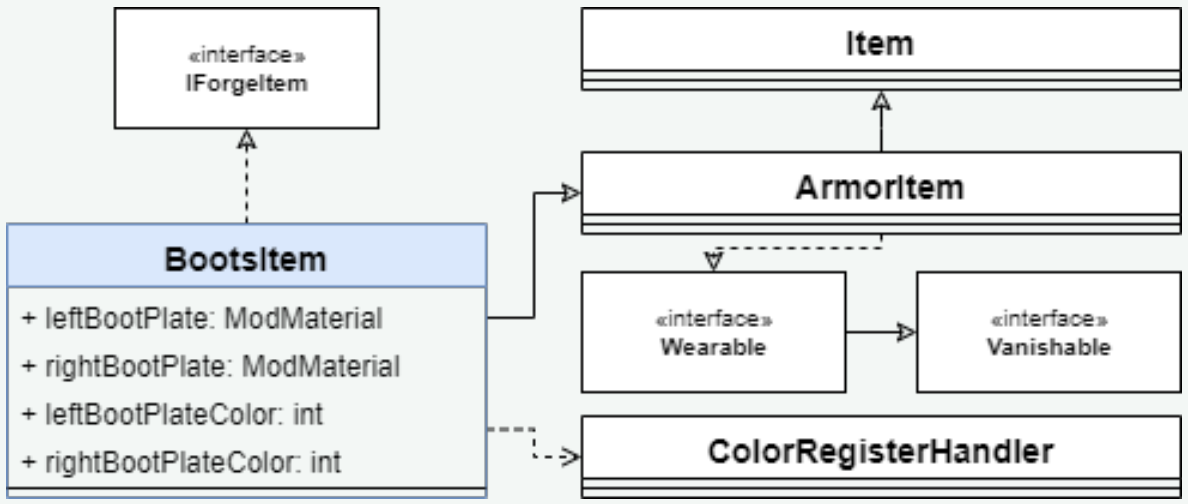

Figure B.16: `BootsItem` class

Figure B.17: Correlation between Block-BlockEntity-Menu-Screen classes

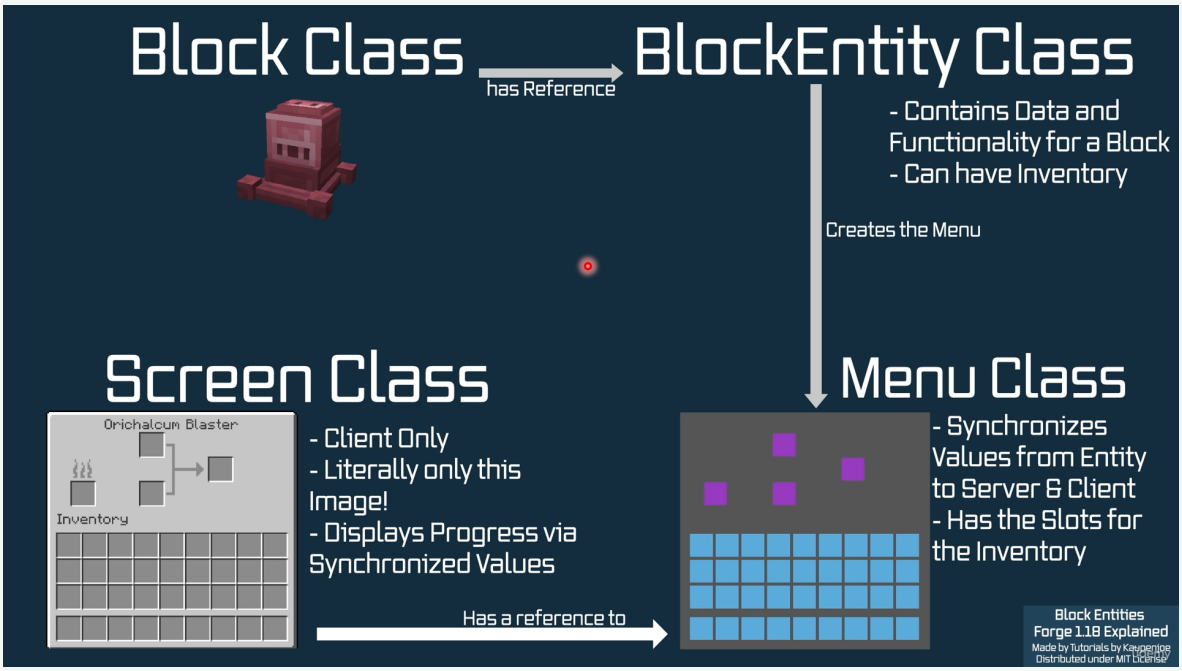

Figure B.17: Correlation between Block, BlockEntity, Menu and Screen classes

Figure B.18: Pattern station interface

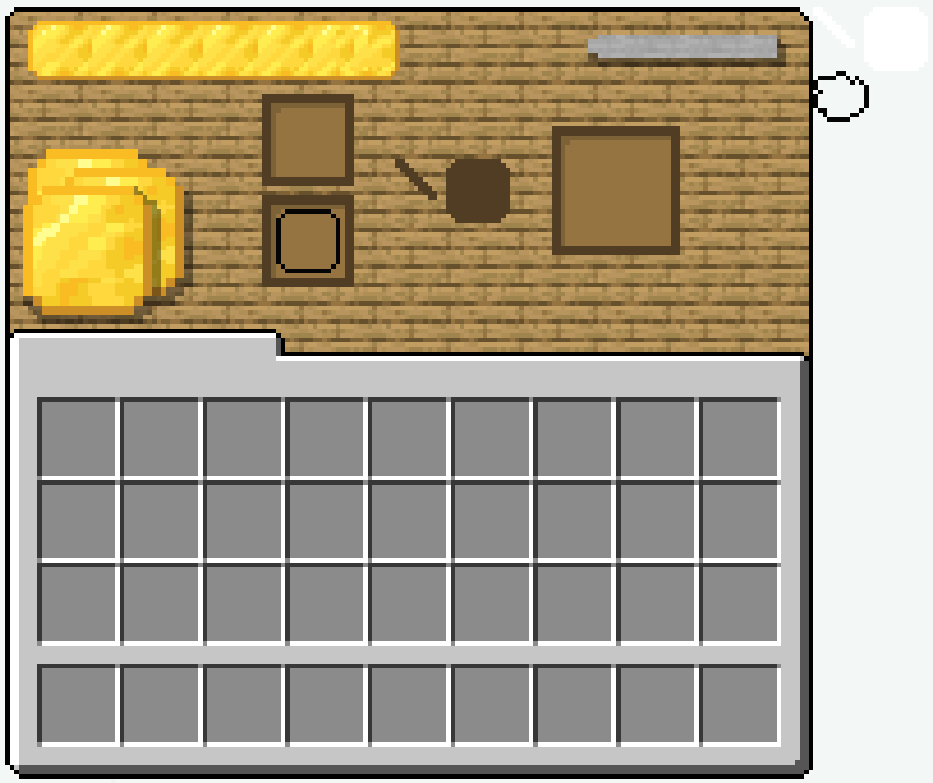

Figure B.18: Pattern station interface

Figure B.19: Armor part station interface

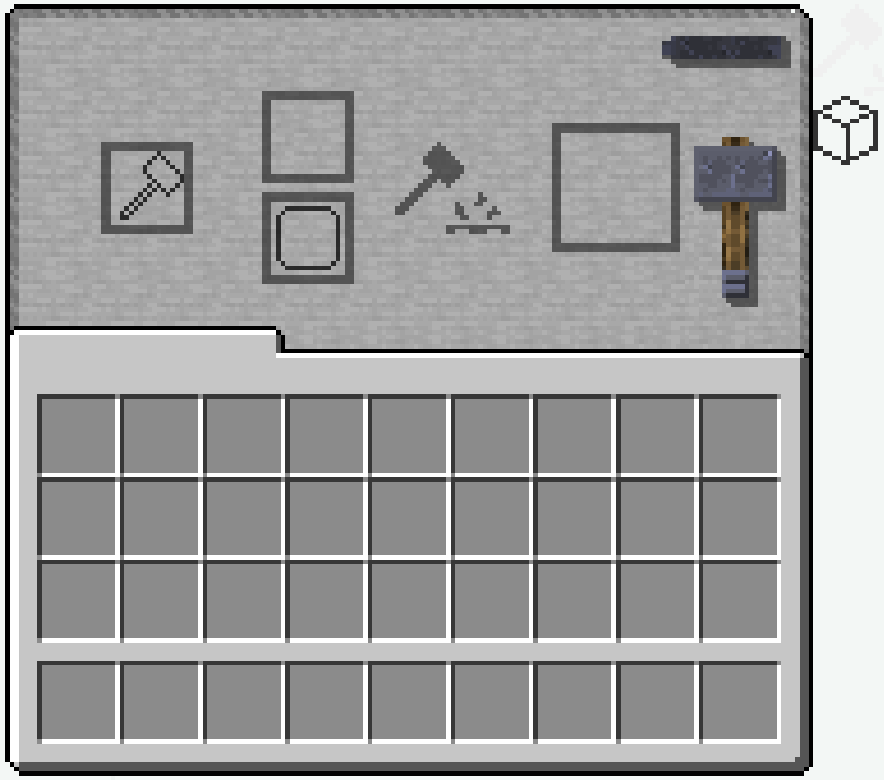

Figure B.19: Armor part station interface

Figure B.20: Armor forge (T1) helmet interface

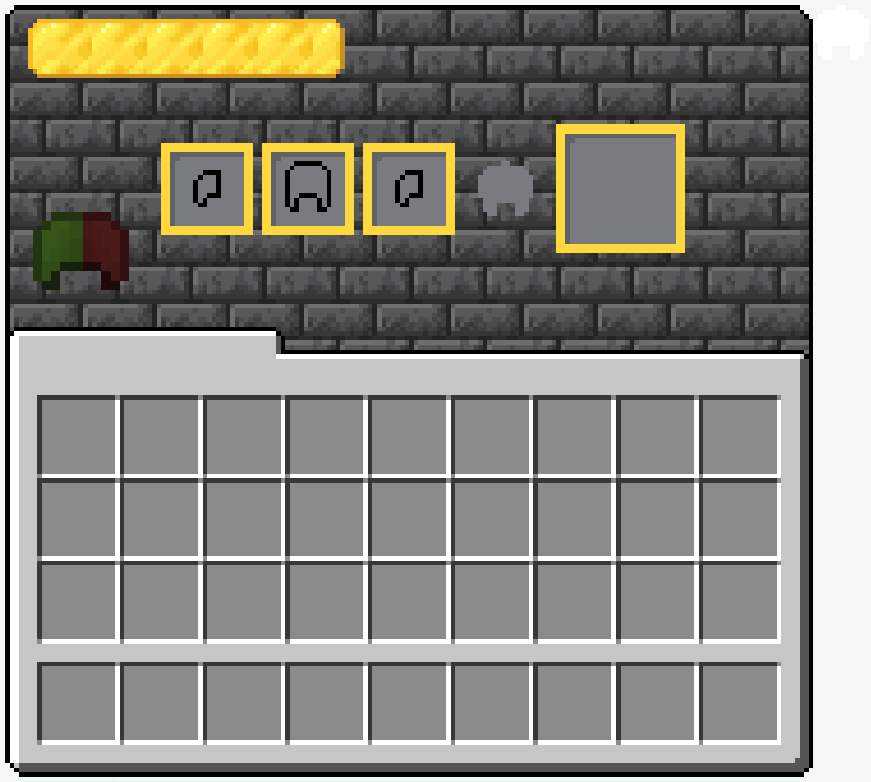

Figure B.20: Armor forge (T1) helmet interface

Figure B.21: Armor forge (T1) chestplate interface

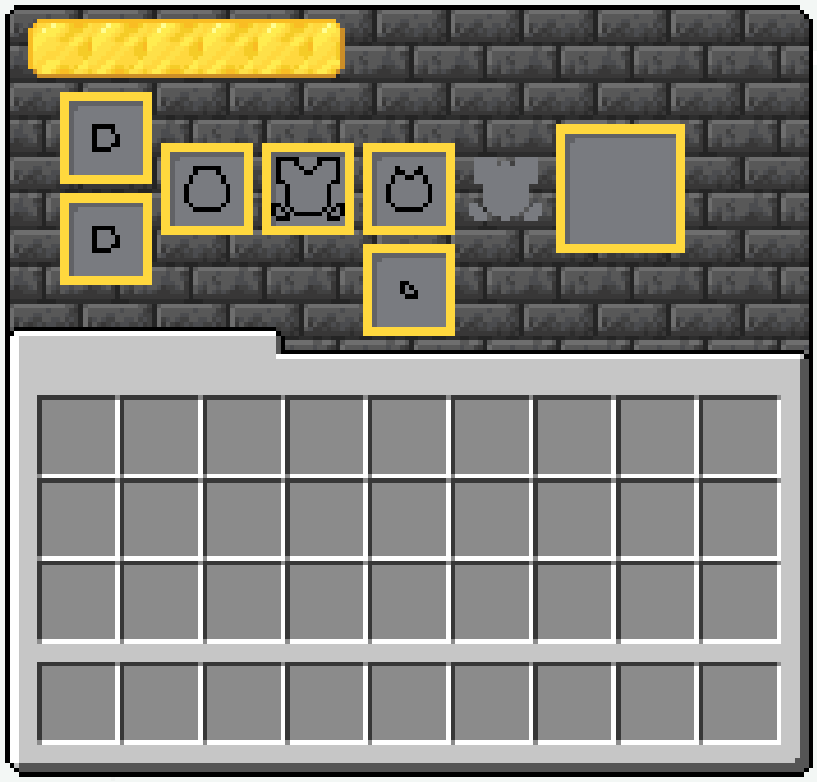

Figure B.21: Armor forge (T1) chestplate interface

Figure B.22: Armor forge (T1) leggings interface

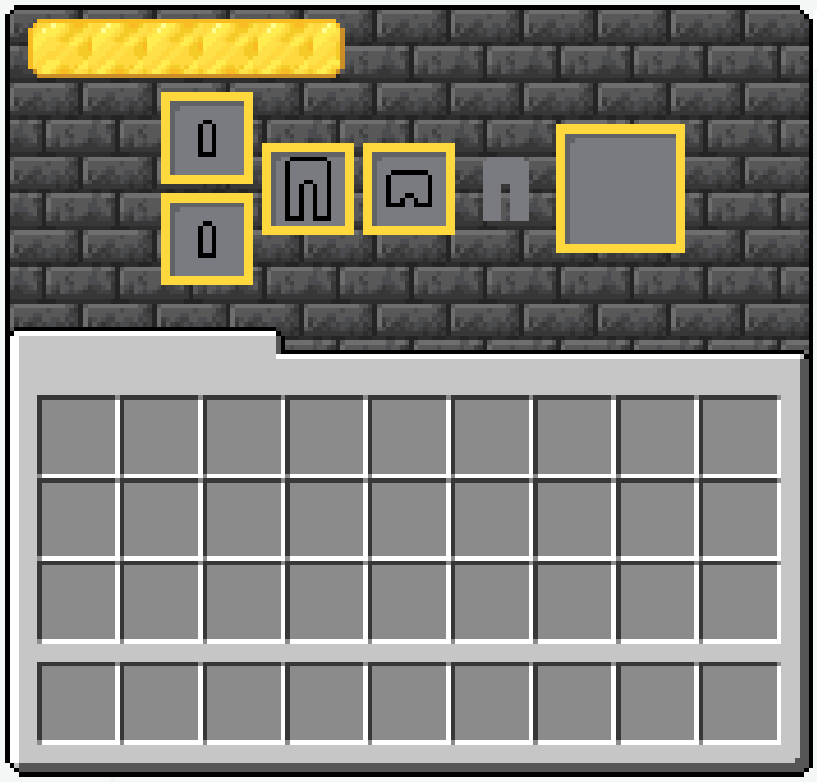

Figure B.22: Armor forge (T1) leggings interface

Figure B.23: Armor forge (T1) boots interface

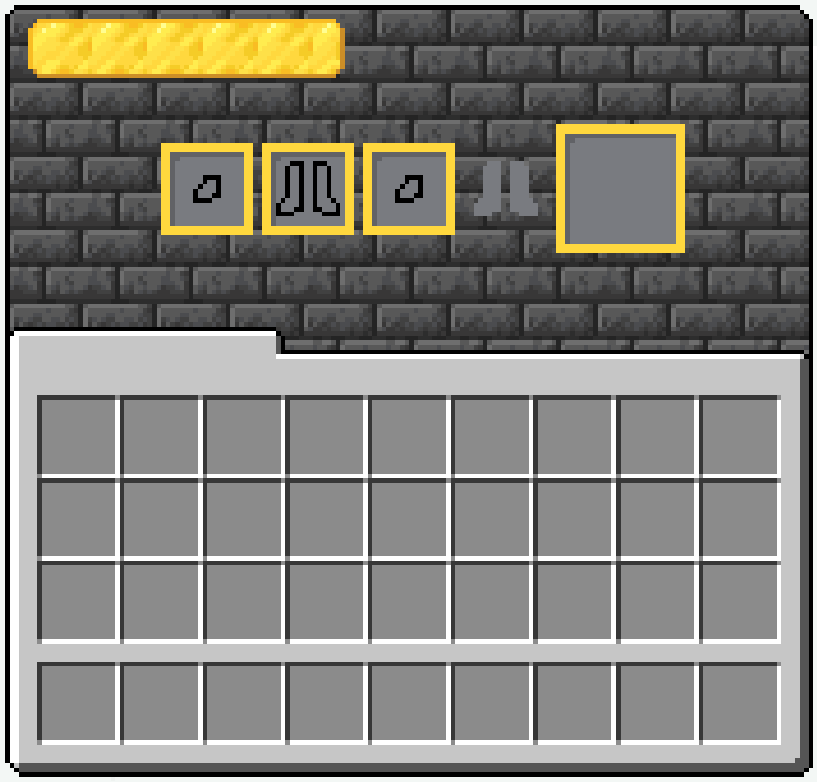

Figure B.23: Armor forge (T1) boots interface

Figure B.24: Ore distribution

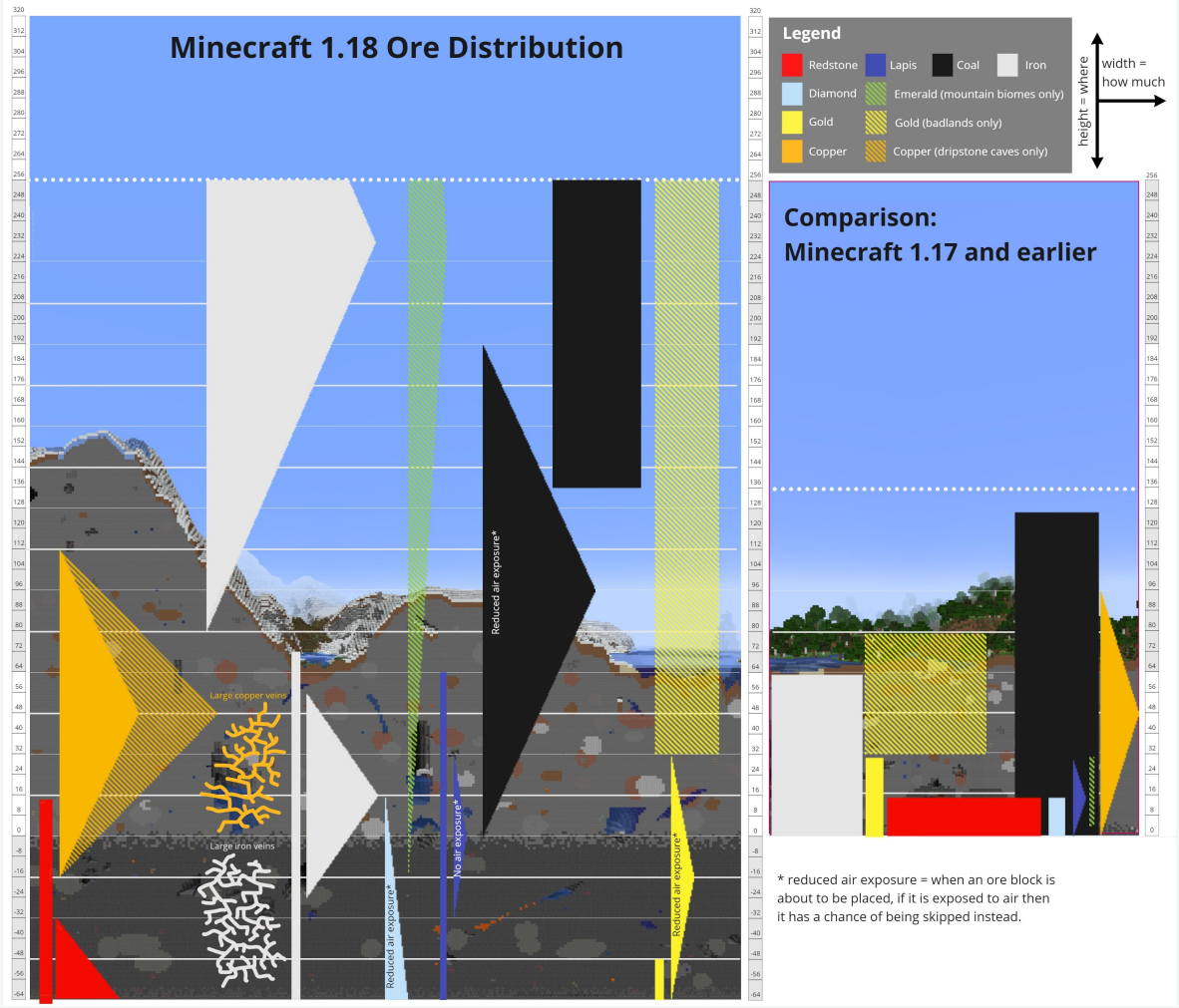

Figure B.24: Ore distribution

Figure B.25: Tree generation test

Figure B.25: Tree generation test

Figure B.26: Tree generation test

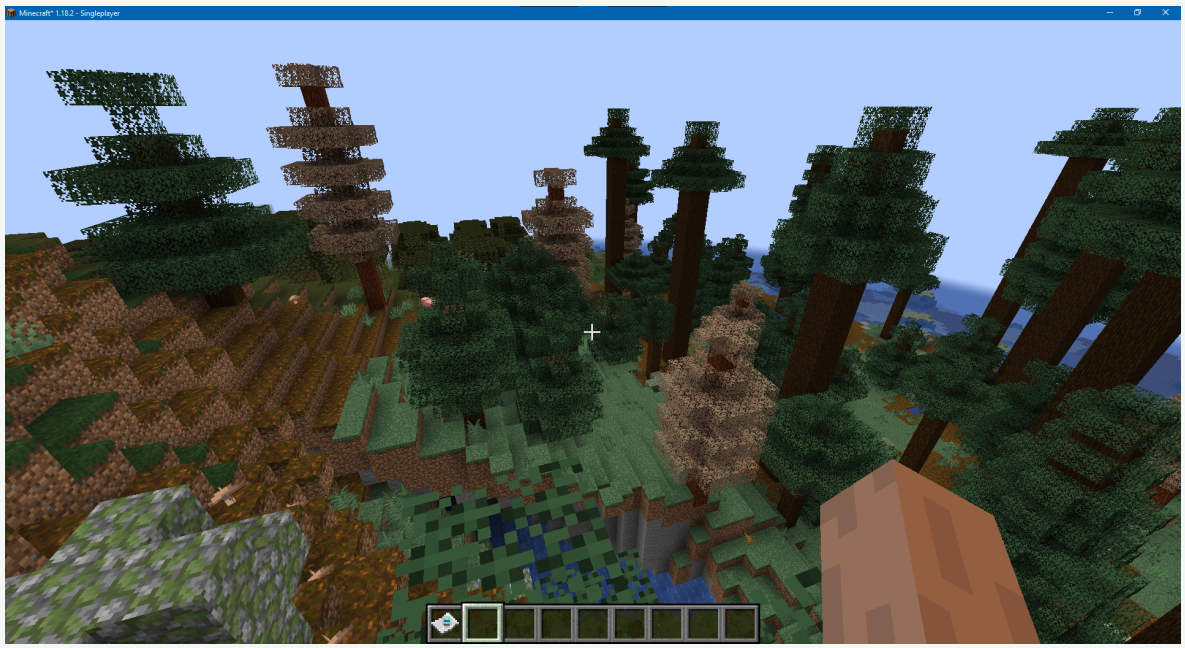

Figure B.26: Tree generation test

Figure B.27: Ore generation test

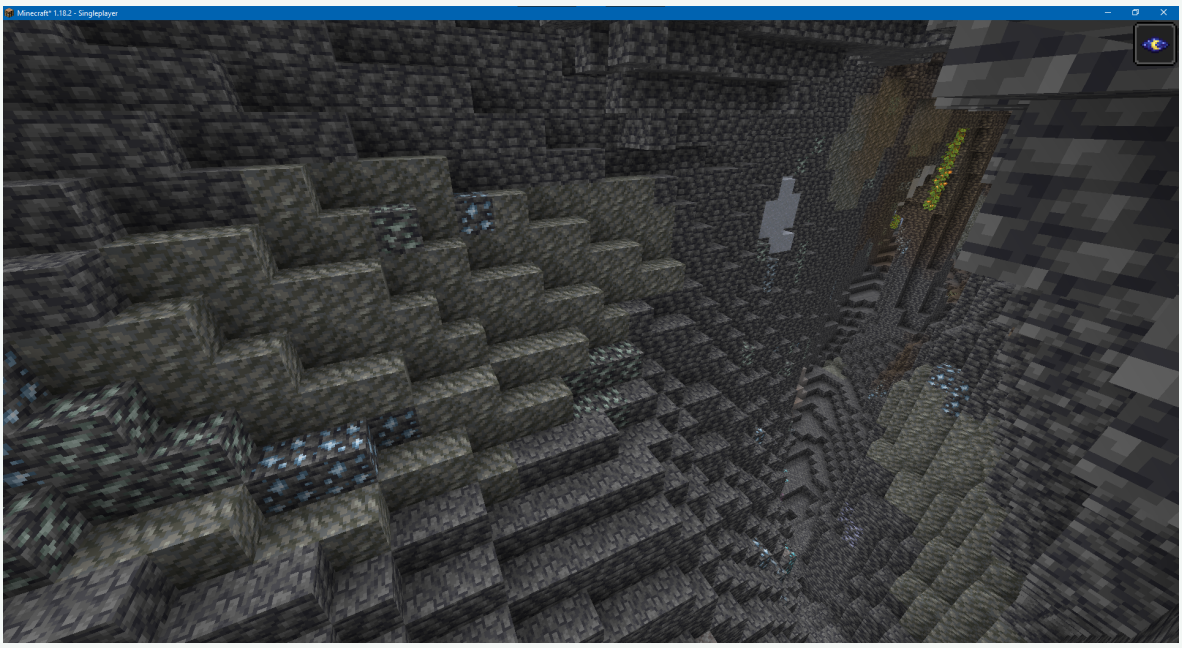

Figure B.27: Ore generation test

Figure B.28: Ore generation test

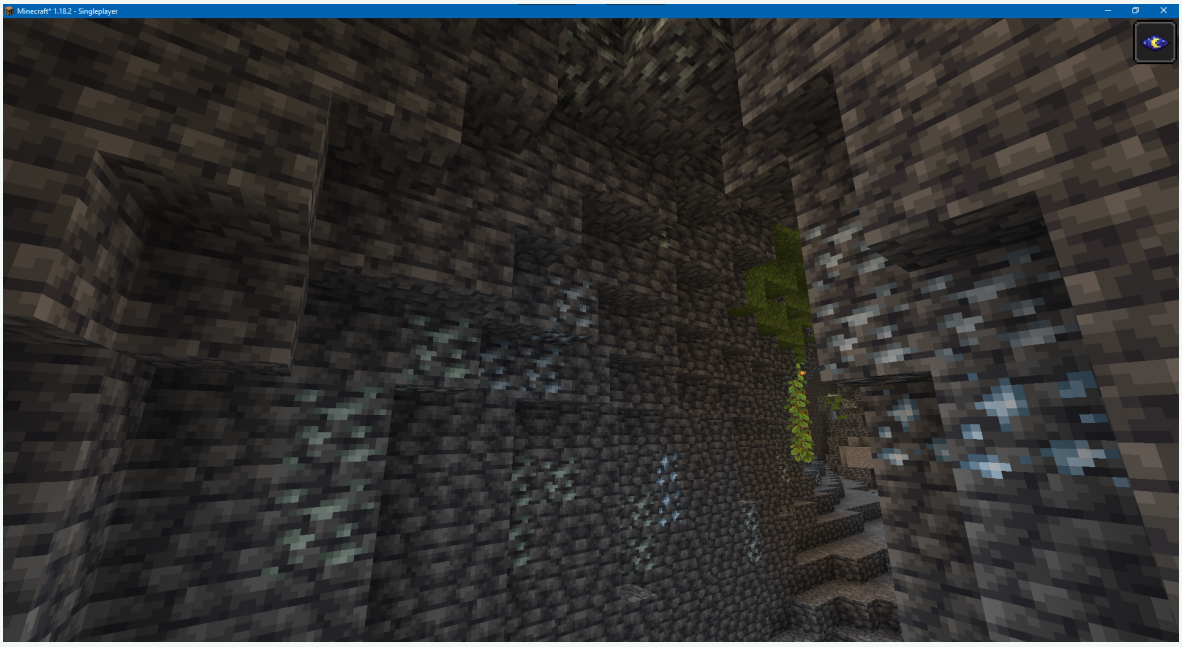

Figure B.28: Ore generation test

Figure B.29: Ore generation test

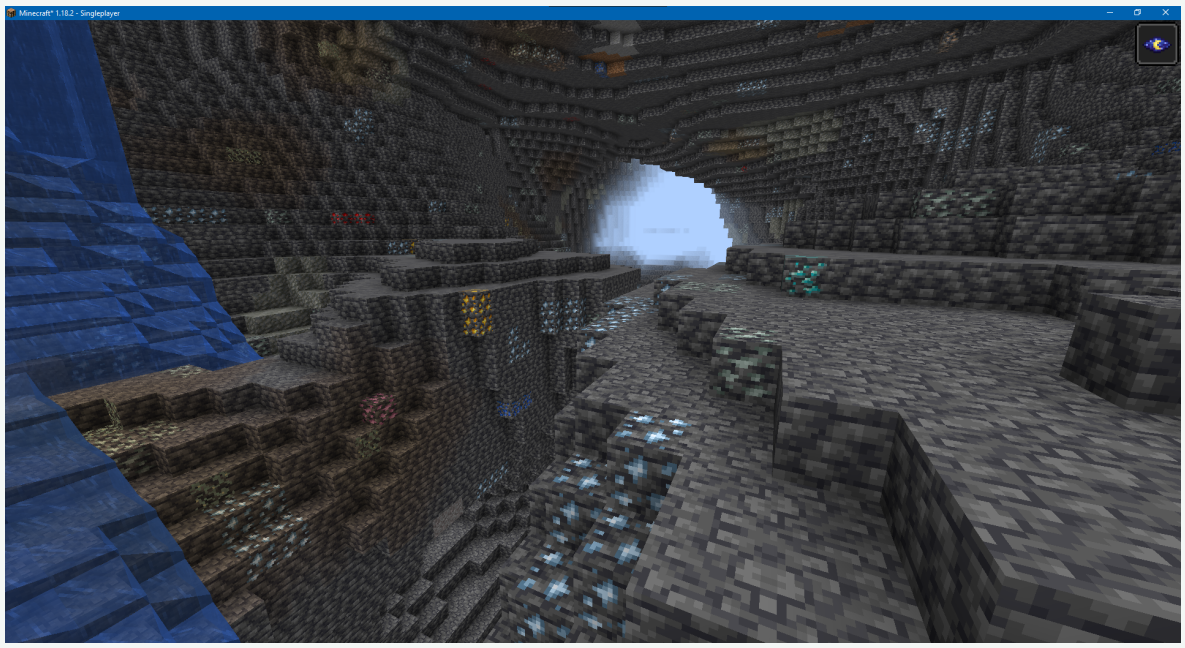

Figure B.29: Ore generation test

Figure B.30: Ore generation test

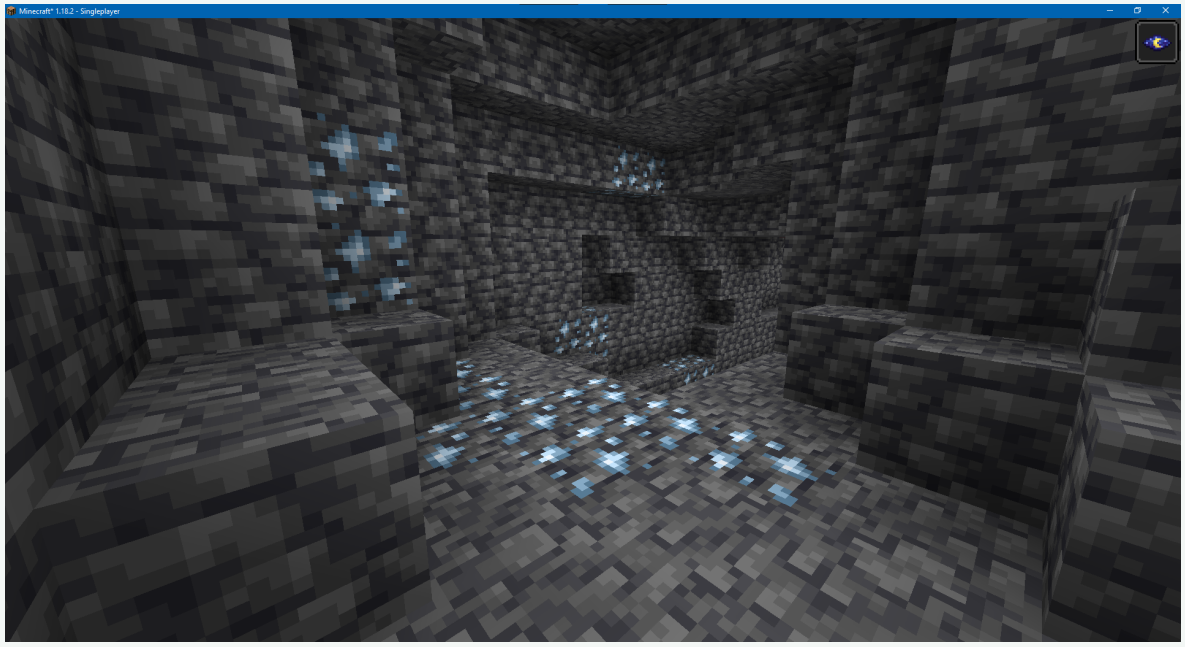

Figure B.30: Ore generation test

Figure B.31: TOP integration test

Figure B.31: TOP integration test