Unit XIV Assignment II

By Nathan Windisch

PIII: Designing A Bespoke Event Driven Application

The software that I shall be designing is a plugin for piece of server software called `Spigot` that is used to make the generation of add-ons to the video game `Minecraft`. The plugin in question is called **Chill**, a tool which allows Administrators to freeze possible cheaters to prevent them from causing any more damage to legitimate players. After this point they can a voice chat service such as Teamspeak3 or Discord so that they can communicate with the staff member that froze them, possibly share their screen and, after rigorous checks, may be unfrozen. If it turns out that the player was cheating, they will be punished accordingly. The input of the program is when the Administrator freezes a player, the output is when the user tries to perform an action that is prohibited while they are frozen, along with notifying staff members when a person who should not be frozen is attempted to be frozen, and the process is the blocking of these actions. There are a few different tools that I shall use within this program, and I shall list them and describe why they are important.

Selection

Selection is important to this program as without it, the Administrator cannot specify which user they would like to freeze, meaning that the program would be completely useless without it.

Loops

Loops are also very important for the "*" feature, which freezes all players on the server. This may be useful if there are some severe bugs that would affect the entire player base, so they are being frozen for their own safety. This feature request a loop to go through all the online players on the server and add them all to the ArrayList of frozen players.

Event Handlers

Event Handlers are used to prevent frozen players from moving, interacting with their environment, interacting with other players, typing commands or talking in chat, amongst other things. This is extremely useful and should be considered one of the core features of the system.

Debugging

Debugging is extremely useful from a development standpoint as it helps me check where errors are in the code if any arise. I can do this by outputting messages when every action is performed, but only showing the messages to staff members if the debugging Boolean is set to true. This should not be used in production to prevent spam towards the Administrator, as these messages will be sent in chat and could cause lots of spam such as a message being sent whenever a frozen player changes their position even by a centimetre.

Variable Declaration and Scope

Variable declaration is useful as it allows me to store the unique identifiers of players who are frozen, meaning that checks can be performed on them and cancel their actions if they are within the list. The scope of this variable will be within the main class due to the fact that it will need to be accessed from every class, so it being stored in the main class makes it in the most central position possible.

Constants

A constant that will be used is the ChatColor short hands. Instead of having to write out

```
player.sendMessage(ChatColor.GOLD + "[" + ChatColor.YELLOW + "Colossal" +
ChatColor.GOLD + "] " + ChatColor.DARK_AQUA + "You have been " + ChatColor.GREEN
+ "frozen" + ChatColor.DARK_AQUA + ".");
```

Which is an excessively long line of code which will need to be changed if the formatting of the colours ever needs to change, I can just write:

```
ChatColor pri = ChatColor.DARK_AQUA;
ChatColor sec = ChatColor.GREEN;


player.sendMessage(ChatColor.GOLD + "[" + ChatColor.YELLOW + "Colossal" +
ChatColor.GOLD + "] " + pri + "You have been " + sec + "frozen" + pri + ".");
```

Which is much shorter and more efficient. It also allows me to change the colour schema on the fly and it will be propagated across the entire program. Adding to this, I can add a `prefix` string to shorten this even more, changing it to:

```
string prefix = ChatColor.GOLD + "[" + ChatColor.YELLOW + "Colossal" +
ChatColor.GOLD + "] ";
ChatColor pri = ChatColor.DARK\_AQUA;
ChatColor sec = ChatColor.GREEN;


player.sendMessage(prefix + pri + "You have been " + sec + "frozen" + pri + ".");
```

As with the other constants, these variables will be accessible across the entire project.

Data Types

I shall be using many different data types, some of which are primitive and can be found within the Java language, and some of which are more advanced and are native to Spigot. I shall list them as follows:

String

Strings will be used to store text data such as the `prefix` and are generally used for communicating with the user.

Integers

Integers will be used within for loops to keep track of which player they are currently operating on, along with using constants for changing timings on the fly. MAX_INT is also used for setting effects to be applied to the player infinitely.

ArrayList

An ArrayList will be used to keep track of all of the users that have been frozen, to ensure that if they perform any action that is not allowed and if they are in the ArrayList then it will be cancelled.

UUID

Due to the fact that players can change their usernames it is important to use their Universal Unique IDentifier, or UUID, when adding players to the ArrayList due to the fact they could just change their name and be thawed.

Entity

An Entity is any dynamic, moving object within the game. This encompasses but is not limited to players, animals and monsters, along with projectiles and boats. These will need to be intercepted when preventing actions.

Player

The Player is a subset of an entity that is controlled by the user. Both the Administrator and the users are considered players, but the Administrators cannot be frozen due to a specific permission node that prevents it, unlike regular users.

HumanEntity

The HumanEntity is an interface of the Player, and NPCs, that is used by chest functions within Spigot.

Projectile

The Projectile is an entity that moves through the air within the game. There are a few subsets of it, such as Arrows, Potions, Enderpearls, Eggs and Snowballs. As a rule of thumb, if the player can throw it then it is a Projectile. These will need to be

Potion

A Potion is an item within Minecraft that gives the consumer a specific effect when applied. They can either be drunk or, in the case of this instance, thrown onto the ground. These 'splash' potions cause an AoE which applies the potion effect to all entities within the vicinity.

EnderPearl

An EnderPearl is a projectile which can be thrown by the Player. When it lands, it is deleted and the Player is teleported to that position. This will need to be cancelled if the user is frozen due to the fact that the frozen user should

not be able to move.

Inventory

An Inventory can be a large range of things, but in this case it will be both the menu that will be shown to the user when they are frozen, and also it will be the user's actual inventory to ensure that they cannot move anything around.

ItemStack

ItemStacks are all items that can be stored within an Inventory. These will be used when showing the menu to the player.

ItemMeta

ItemMeta is a subset of an ItemStack that is used for customizing the Item. It can change the name of the Item, as well as adding lore to the item.

ChatColor

ChatColor is the way that colours are used within Spigot, due to the fact that Minecraft only has 16 colours and 6 modifiers, such as obfuscated and bold. This will be used for all colour within the project, such as shown above when the user was informed that they were frozen.

Triggers Used

There will be two methods of triggering a user being frozen by an Administrator, the first one will be typing the command `/freeze <playerName>`. The second will be selecting their name from an Inventory based menu system. These methods will add the user to the frozen ArrayList and will display the message to the user that they have been frozen. They will also be given a Blindness effect so that they cannot see, and then when they perform any action that is not permitted then it will be blocked, due to the other events within the project.

Properties Assigned to Screen Components

There will not be that many on screen properties within this project due to the fact that this is a relatively small system, but I shall list them as follows:

Help Page

The help page will be shown when the user executes the command `chill`, `freezehelp`, `helpfreeze`, `sshelp` or `helpss`. When executed it will run the following code.

```
sender.sendMessage("/freeze <playerName> - Freezes/Unfreezes a player, used as a
toggle. chill.freeze");
sender.sendMessage("/frozen <playerName> - Checks if a player is frozen.
chill.frozen");
sender.sendMessage("/panic - Places you into panic mode, whereby you cannot be
unfrozen. chill.panic");
```

Command Reference

If the player runs the commands either with too many arguments, not enough arguments or invalid arguments, a help prompt will be displayed to the user telling them what the correct syntax is. The following is an example of that:

```
sender.sendMessage("/freeze <playerName>");
```

Confirmation Page

When performing a security critical command, such as freezing all players on the server, a confirmation command must be run. In this case, the command is `/freeze <*|all>` and will freeze all current players on the server. This may be due to a server error which may cause players to die and lose all of their items, so this ensures that they cannot be harmed. The command to confirm if `/freeze <*|all> confirm` and will ensure that the Administrator was serious about issuing the command as the only way to turn it off is to restart the server.

User Menu

When the user is frozen there will be a menu that they can interact with. The middle icon will be information about a general idea as to why they were frozen, and an IP address so that they can join TeamSpeak3, or a link so that they can join Discord, both of which are server-based VoIP services. The button on the left will be titled the **Admit** button and the user can press it if they want to admit to cheating. This will send a notification to the staff member and the user will get a shorter sentence due to the fact that they did not waste the Administrator's time. On the right side there will be a **Dispute** button which will allow the user to join a VoIP service of their choice (a list of VoIP services that they can choose will be displayed after this menu) so that they can talk to the Administrator that froze them. If it turns out that they were lying, their punishment may be extended.

Admin Menu

When the Administrator wishes to freeze a user, they can either issue the command `/freeze <playerName>` or they can issue the command `/freezegui` to open up a list of all of the players which are online. The Administrator can sort either by frozen or thawed, and if they click on a user's head within the menu then the selected user will be frozen.

Messages in Chat While Frozen

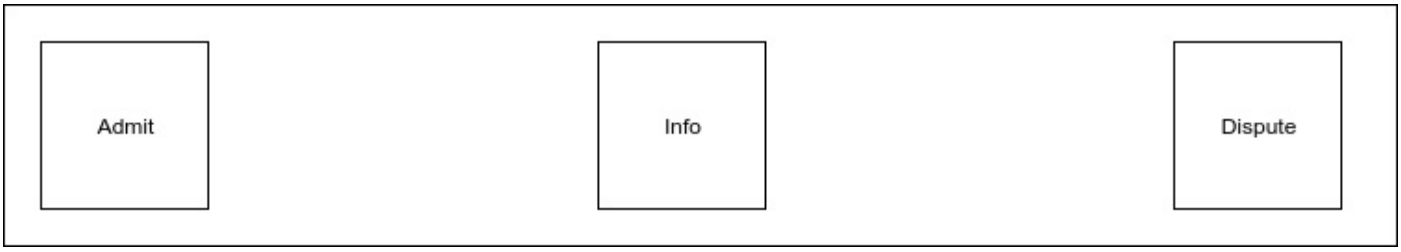
When the user is frozen, a message will pop up in their chat notifying them that they have been frozen. The message will also tell them to join a voice chat server such as TeamSpeak3 or Discord.

Messages in Chat when Action is Forbidden

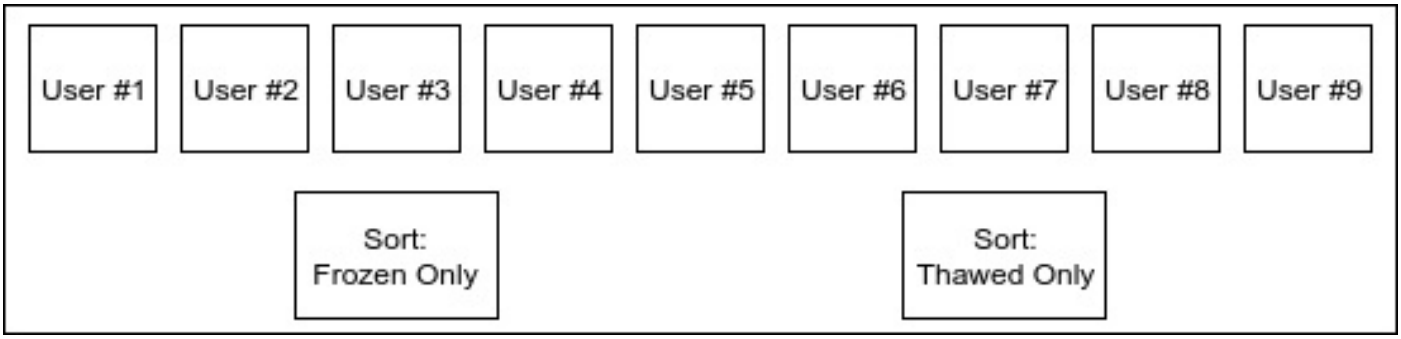
When a user performs an action which is prohibited while frozen, the same notification that was sent when they were first frozen will also be sent, but the action that they performed will also be within the message so that they know what they should not be doing.

Design Draft Sketches

#1



#2



#3

While this next design draft is not a sketch, is it the layout of how the command syntax will work.

```
freeze:
        <playerName>: Freezes/Thaws the specified user.
        <UUID>: Freezes/Thaws the UUID applied to the player.
        <all|*>: Asks the Administrator for a confirmation.
                <confirm>: Confirms the command and adds all players on the
server to the frozen ArrayList.
thaw:
        <playerName>: Thaws the specified user.
        <UUID>: Thaws the specified user.
chill: Shows information about the plugin.
panic: Adds the user that issued the command to the frozen ArrayList.
```

Test Plan

The following is a list of events that I will test, and the things that I will test them with.

Event	Expanded Explanation
onBlockBreak	Check if regular users can break blocks
	Check if frozen users can break blocks
onBlockPlace	Check if regular users can place blocks
	Check if frozen users can place blocks
onCommandPreprocess	Check if regular users can execute commands
	Check if frozen users can execute commands
onEnderpearlThrow	Check if regular users can throw enderpearls
	Check if frozen users can throw enderpearls
onInventoryClose	Check if regular users can close inventories
	Check if frozen users can close inventories
onPlayerDropItem	Check if regular users can drop items
	Check if frozen users can drop items
onPlayerGiveDamage	Check if regular users can give damage
	Check if frozen users can give damage
onPlayerMove	Check if regular users can move
	Check if frozen users can move
onPlayerPickupItem	Check if regular users can pickup items
	Check if frozen users can pickup items
onPlayerQuit	Check if regular users can notify staff on quit
	Check if frozen users can notify staff on quit
onPlayerTakeDamage	Check if regular users can take damage
	Check if frozen users can take damage

onPlayerTeleport	Check if regular users can teleport
	Check if frozen users can teleport

Pseudo Code

In the following segment I will show some pseudo code of various elements of my program.

A commented version of this can be found at natfan.github.io/pseudo

```
event : PlayerPlaceBlockEvent
        player event.getPlayer : Player
        if Main.frozen.contains(player.getUniqueID)
                event.setCancelled(true)
                player.sendMessage(frozenMessage)
```

```
event : CommandPreprocessEvent
        player event.getPlayer : Player
        if Main.frozen.contains(player.getUniqueID)
                event.setCancelled(true)
                player.sendMessage(frozenMessage)
```

```
event : onPlayerMove
        player event.getPlayer : Player
        if Main.frozen.contains(player.getUniqueID)
                event.setCancelled(true)
                player.sendMessage(frozenMessage)
```

```
event : CommandExecutor
        sender : CommandSender
        commandlabel : String
        if commandlabel.equalsIgnoreCase("freeze")
                if sender.hasPermission("chill.freeze")
                        if args.length == 0
                                sender.sendMessage(freezeSyntax)
                        if args[0].equalsIgnoreCase("all")
                                if !sender.hasPermission("chill.freeze.all")
                                        sender.sendMessage(permissionError)
                                if args[1].equalsIgnoreCase("confirm)
                                        for Player all : getOnlinePlayers()
                                                Main.frozen.add(all.getUniqueID)
                                sender.sendMessage(confirmPlease)
                        targetEntity args[1] : Entity
                        if targetEntity instanceof Player
                                target targetEntity : Player
                                if Main.frozen.contains(target.getUniqueID)
                                        removeEffects(target)
                                        Main.frozen.remove(target.getUniqueID)
                                else
                                        if
!target.hasPermission("chill.override")
                                                addEffects(target)

Main.frozen.add(target.getUniqueID)
                                        else
                                                sendMessage(player.getName() + "
might be trying to abuse!", "chill.notify")

        if commandlabel.equalsIgnoreCase("thaw")
```

```
                if sender.hasPermission("chill.freeze")
                        if args.length == 0
                                sender.sendMessage(freezeSyntax)
                        targetEntity args[1] : Entity
                        if targetEntity instanceof Player
                                target targetEntity : Player
                                if Main.frozen.contains(target.getUniqueID)
                                        removeEffects(target)
                                        Main.frozen.remove(target.getUniqueID)
                                else /* throw an error as the target isnt frozen
                                        sender.sendMessage(targetNotFrozen)

        if commandlabel.equalsIgnoreCase("panic")
                if sender.hasPermission("chill.panic")
                        if !Main.frozen.contains(target.getUniqueID)
                                addEffects(target)
                                Main.frozen.add(target.getUniqueID)
```

Flow Chart

The following is a flow chart that will outline the process of this system:

MII: Reasoning Behind Tools and Techniques

I shall now list some tools and techniques, give reasons as to why they are used and justify them.

Objects

Objects are useful within event driven programming due to the fact that it allows programmers to instantiate instances of their code, meaning that they can edit small bits of them on the fly and using them instead of the default values. This is highly useful as developers can then compare their edited values with the original ones in order to view the changes. This is a good thing as it lets developers follow the rules of object orientation, making their code more efficient and streamlined, allowing for easier documentation and better code continuity.

Triggers

Triggers are actions that get triggered when an event occurs. These are very useful due to the fact that it allows you to do some very interesting event driven things such as clearing cache when a user logs out of a system or assigning a discount when a customer uses a promotional code. This means that triggers make event driven programming better than traditional programming methods as the code for removing the cache on specific events only has to be written once as a function, then it can be applied to as many events as the developer wants.

Menus

Menus are a method of allowing users to interact with more advanced parts of a program, such as saving, loading and editing preferences. In many GUI programs, a list of menus can be accessed by pressing `ALT` which will show up headings such as

- File
- Edit
- View
- Print
- Help

All of these headers can be clicked on to access even more functions. For instance, in *File* alone there are usually the following:

- New
- Open
- Save
- Save as…
- Print
- Page Setup
- Preferences
- Quit

As you can see, this is a vast amount of options and functions to choose from. This allows the user to perform many more actions than if they did not have a menu, and the preferences settings allows the user to customize their experience as much as they want. Menus would be much harder to write in an imperative language as the developer would not be able to display the menu on button presses, such as clicking on `File -> Open` and also having the ability to press `C-o`. This means that menus are useful and should be used by developers that want to give users larger amount of functionality while still keeping a simple, clean user experience, or UX.

Debugging

Debugging is highly useful to developers as it allows them to step through code and see where errors are occurring. This allows developers to spend less time bug fixing and more time coding, meaning that their time is used more efficiently. Most Integrated Development Environments, or IDEs, have their own debugging tools built in, such as underlining or otherwise highlighting code that is syntactically incorrect. Debugging can also be found when executing the code in the form of StackTraces where the developer is shown the position and time of their code's failure, meaning that they can go out of their way to fix it.

Variables

Variables are arguably one of the most important concepts of traditional and event driven programming. They allow developers to store data within their programs with specific boundaries, such as being a whole number or one character in length. These pieces of data can be set to be able to be accessed from different places in the program, or they can be kept a secret. Variables can then be manipulated to produce desireable results. I shall show an example of this now.

```
Console.Write("0 ");
Console.Write("1 ");
Console.Write("2 ");
...
```

```
for (int i = 0; i >= 100; i++) {
        Console.Write(i + " ");
}
```

```
int i = 0;
while (i < 100) {
        Console.Write(i + " ");
        i++;
}
```