

ROSE-HULMAN INSTITUTE OF TECHNOLOGY

# Team Mohan

---

## Milestone 2

**Samuel Kim, Brian Collins, Michael Williamson, Kevin Geisler**

**10/10/2011**

## Contents

Executive Summary .....	3
Introduction .....	4
Problem Description.....	5
Client Background.....	5
Current System.....	5
Key Needs .....	6
Features .....	7
Use Cases.....	10
Use Case Identification.....	10
Use Case Feature Mapping .....	10
Use Case Description .....	11
Data Flow Diagrams .....	15
Context-Level Diagram .....	15
Run Player Commands Level 1 .....	16
Asserts a given game state .....	16
Create Mock Player .....	17
Player received text & Inventory State.....	18
Functionality Requirements.....	19
Server Functions.....	19
Player Command Functions.....	19
Entity State Functions.....	19
Player State Functions .....	19
World State Functions .....	20
Storyboard .....	21
References .....	22
Index.....	23
Glossary.....	24

## Executive Summary

Developers currently do not possess any means of testing plugins for Minecraft [1], which makes debugging plugins a tedious process.

This document will briefly summarize the problem with the current system, and how Liza intends to solve that need. The focus is outlining how *Liza* is intended to be used. This is described through several use cases that utilize several testing strategies. Some data flow diagrams (DFD) will give an overview of the steps Liza will take. A storyboard is also incorporated to illustrate a scenario that a plugin developer may run into, and how Liza will introduce a new solution to the problem

## Introduction

Minecraft [1] is a sandbox computer game where players can create and remove blocks in a simulated world. These blocks can be arranged in a nearly unlimited number of ways in order and is only limited by the user's imagination. Despite not being officially released, it has gained immense popularity with a user base exceeding 10 million people.

Modifying Minecraft [1] has become increasingly popular, as a number of changes can be made to suit each user's needs. There are currently two methods of modifying Minecraft [1]. Mods require a user to directly modify their game files in order to add or alter functionality in Minecraft [1]. For a server to effectively use a mod, the server requires each user to install that mod. On the other hand, plugins enable developers to make changes to Minecraft [1] without needing each user to directly modify the platform. Only the server needs to be modified.

A common way of introducing plugins to a Minecraft [1] server is to utilize a tool called Bukkit [2]. Bukkit wraps around the official server application and exposes an easy application programming interface (API) for developers to create plugins.

As recommended by Bukkit [2], many developers run Apache Maven [3] alongside the Bukkit server. Apache Maven [3] provides an effective solution for software project management. This includes being able to manage a project's build, as well as recording documentation.

Bukkit [2] currently does not possess any means of testing, which makes debugging plugins a tedious process. *Liza* intends to provide a unit testing framework for plugin developers to programmatically test their code.

The previous milestone introduced and described the need of *Liza*, and the features that will need to be implemented. This milestone discusses how *Liza* is intended to be used. These first two documents will tie into the third, where a preliminary prototype is designed, along with any supplementary specifications needed. The fourth milestone details possible test cases for our test framework. For our project, this will mean testing each process listed in the DFDs.

## Problem Description

### Client Background

Tim Ekl and Eric Stokes are software engineers from Rose-Hulman Institute of Technology. Both are active and prominent developers in the Minecraft [1] plugin community. As developers, they understand the importance of creating a strong test base before deploying code, and have become frustrated with the lack of a proper testing framework for Bukkit [2]. If a successful testing framework were available, they would be able to easily test their plugins and those of other developers. They also would like to see the testing framework used by other people in the plugin community, so that more plugins will be of higher quality at their release.

### Current System

Currently, there is no support for automated testing. Plugin developers load their code into a test server, and make sure functionality works manually. Doing thorough tests in this way, however, is extremely tedious. As such, some developers only run a few basic cases and therefore may miss some edge cases, or possibly conflict with functionality elsewhere. More daring developers may throw caution to the wind and load their plugins into active servers. This puts the stability of the server and its data at risk.

## Key Needs

The Problem of...	<ul style="list-style-type: none"> <li>- Bukkit [2] plugins go untested and therefore are unstable.</li> <li>- Importing plugins to test by trial and error is a lengthy process.</li> <li>- Testing code on a server with informative results is unimplemented</li> </ul>
Affects...	Plugin Developers Server Hosts Players
And results in...	<ul style="list-style-type: none"> <li>- Server crashes or plugin bugs occur.</li> <li>- Players have a frustrating experience on a server due to frequent crashes and lag.</li> <li>- Developers spend more time debugging and constantly exporting their code to test on a real server.</li> </ul>
Benefits of a solution	Reliable plugins Faster develop time Less server crashes Ease of mind

Developers of Minecraft [1] plugins tend to have problems testing their code. There is currently no way of testing their plugins in Minecraft [1] besides running it directly within the game's server. For larger plugins, identifying where issues occur becomes increasingly difficult. Plugins may crash servers if not tested thoroughly, which proves frustrating for both the players and the developers.

A possible solution to this is to create a new testing framework which extends the current Java 6 JUnit [4] testing. A user will be able to create a testing script which will be able to spawn a mock player. The mock player will then run the code written in the file and then listen for events or states which will be used to establish the results of the testing script.

## Features

ID	Feature	Priority	Effort	Risk
1	Create a mock player	High	High	High
2	Communicate with Bukkit [2]	High	Med	High
3	Emulate player control through mock player	High	High	High
4	Listen for events	High	Med	Low
5	Incorporate JUnit [4]	High	Med	Low
6	Assert Entity/Block attributes	High	Med	Low
7	Create/Remove Entity/Blocks/Items	High	Med	Med
8	Send mock events	Med	Med	Med
9	Enable/Disable other plugins	Low	Med	High
10	Can detect test interference (from other players/entities)	Low	High	Med
11	Display test results	None	Low	Low

- Create a mock player
  - Status: Approved
  - Priority: High
  - Effort: High
  - Risk: High
  - Stability: Medium
  - Reason: Most events in Minecraft [1] are player driven, so the testing framework should be able to create a mock player
- Communicate with *Bukkit* [2]
  - Status: Approved
  - Priority: High
  - Effort: Medium
  - Risk: High
  - Stability: High
  - Reason: *Bukkit* [2] provides a set of events to listen to, and being able to send/receive information with *Bukkit* [2] will prove invaluable for testing.
- Emulate player control through mock player
  - Status: Approved
  - Priority: High
  - Effort: High
  - Risk: High
  - Stability: Medium
  - Reason: A mock player needs to be able to do any action like a human player would

- Listen for events
  - Status: Approved
  - Priority: High
  - Effort: Medium
  - Risk: Low
  - Stability: Low
  - Reason: Event listening will be a major component in asserting correct behavior
- Incorporate JUnit [4]
  - Status: Approved
  - Priority: High
  - Effort: Medium
  - Risk: Low
  - Stability: High
  - Reason: As a Java based project, JUnit [4] provides an existing base for asserting code output
- Assert Entity/Block attributes:
  - Status: Approved
  - Priority: High
  - Effort: Medium
  - Risk: Low
  - Stability: High
  - Reason: This feature allows *Liza* verify that an element in the game is at a desired state
- Create/Remove Entity/Blocks/Items:
  - Status: Approved
  - Priority: High
  - Effort: Medium
  - Risk: Medium
  - Stability: High
  - Reason: This feature allows *Liza* create and remove elements in the game, so that the certain Entities, Blocks, or Items, can be tested.
- Send mock events
  - Status: Approved
  - Priority: Medium
  - Effort: Medium
  - Risk: Medium
  - Stability: Medium
  - Reason: This will allow the developer to simulate some events that may occur in the Minecraft [1] environment



- Enable/Disable other plugins
  - Status: Proposed
  - Priority: Low
  - Effort: Medium
  - Risk: High
  - Stability: Medium
  - Reason: Many servers operate using multiple plugins, which may interfere or conflict with the one being tested. The client believes this feature may be convenient, but may not be in the first release
- Detect test interference (from other players/entities)
  - Status: Proposed
  - Priority: Low
  - Effort: High
  - Risk: Medium
  - Stability: Medium
  - Reason: This will allow the testing framework to detect if the mock player has been affected in any unintended way by an outside entity. The client believes that this feature may be useful, but not necessary. The developer should be able to account for such interference independently.
- Display test results
  - Status: Unapproved
  - Priority: None
  - Effort: Low
  - Risk: Low
  - Stability: High
  - Reason: The client decided that printing test results is the responsibility of the plugin developer.

## Use Cases

### Use Case Identification

Use Case ID	Use Case Name
A	Developer runs a unit test
B	Developer writes test to maintain code
C	Developer writes test to debug code
D	Developer writes test for test-driven development

### Use Case Feature Mapping

Use Case ID	Feature IDs
A	1, 2, 8, 9, 10
B	3, 4, 5, 6, 7
C	3, 4, 5, 6, 7
D	3, 4, 5, 6, 7

## Use Case Description

### Use Case A

**Name:** Developer runs a unit test

**Brief Description:**

After a developer has written code for his or her unit test, the test needs to be run. A particular developer may choose to run the test in a number of different ways. For this use case, we will assume that the developer chooses to use Maven [3].

**Actors:** Developer

**Basic Flow:**

1. User starts building the project using Maven [3].
2. Maven [3] builds and compiles the project source file.
3. Maven [3] exports the compiled plugin
4. User then starts or restarts the current Bukkit [2] server
5. Maven [3] starts the unit test code. [Alternate Flow 1 Possible]
6. Unit test code creates mock player
7. Unit test performs the actions specified by user [Alternate Flow 2 Possible]
8. Unit test asserts a valid game state specified by user
9. Unit test removes mock player and completes operation.
10. Maven [3] displays test results specified by user

**Alternate Flow:**

1. During a test, the server becomes unavailable
  - a. The developer contacts the server administrator. If the server cannot return to operation, then the developer will wait until a later date, or try a different server. [End use case]
2. Mock player may be interrupted during a test
  - a. Developer may choose to abort the test or ignore the interference. [End use case or return to Basic Flow 7]

**Pre-conditions:**

- Developer has written a partially functional plugin
- Developer has written some test cases
- Server is available

**Post-conditions:**

- Unit test code has run successfully or server is unavailable

### *Use Case B*

**Name:** Developer writes test for regression testing

**Brief Description:**

The developer may use *Liza* to maintain a plugin that he or she has already written. This ensures that any current functionality remains working properly when adding future features.

**Actors:** Developers

**Basic Flow:**

1. User writes test code that covers all functionality of the current plugin
2. Developer runs the test case, and ensures that each test passes [See Use Case A][Alternate Flow 1 Possible]
3. Developer makes changes to the code
4. Developer runs the test again, and ensures that the previously written test cases still pass [See Use Case A][Alternate Flow 2 Possible]

**Alternate Flow:**

1. Before making changes, one or more test cases fail
  - a. This may be due to a bug in the code. [Go to Use Case C Basic Flow 3]
  - b. This may be due to an error in the test case. Developer makes amendments to the test case. [Return to Basic Flow 2]
2. After making changes, one or more test cases fail
  - a. The change in code has created a conflict in old code. [Go to Use Case C Basic Flow 3]

**Pre-conditions:**

- Developer has written a working plugin

**Post-conditions:**

- After the changes have been made, all original unit tests pass

### *Use Case C*

**Name:** Developer writes test to debug code

**Brief Description:**

A plugin developer may use *Liza* to recreate bugs and ensure that they are fixed after making changes.

**Actors:** Developers

**Basic Flow:**

1. Developer writes a test code that will recreate the conditions of the reported bug. The test should assert the desired effect, rather than the reported effect.
2. Developer runs code, and ensures that the bug occurs, and the test case fails. [See Use Case A] [Alternate Flow 1 Possible]
3. Developer makes changes to the code.
4. Developer runs the code again, and ensures that the test case now passes. [See Use Case A] [Alternate Flow 2 Possible]

**Alternate Flow:**

1. Before making changes, the test case(s) fails.
  - a. There may be an error in the test case. The developer makes corrections to the test case. [Return to Basic Flow 2]
  - b. There may be an error in the way the bug was reported. The developer will need more information to run the test. [End Use Case]
2. After making changes, the test case(s) fails.
  - a. The bug has not been fixed correctly. The developer makes corrections to the code. [Return to Basic Flow 4]

**Pre-conditions:**

- Developer has an existing plugin, and is at least partially functional

**Post-conditions:**

- After making the changes, all test cases pass

### *Use Case D*

**Name:** User writes test for test-driven development

**Brief description:**

When writing a new plugin, or adding a feature to an existing one. A developer may choose to write test cases before coding. This allows the code to be fully tested as it is developed.

**Actors:** Developer

**Basic Flow:**

1. Developer writes test code that will utilize the unimplemented features.
2. Developer runs the test code, and ensures that each fails. [See Use Case A][Alternate Flow 1 Possible]
3. Developer begins implementing the features
4. Developer runs the test, ensuring that the implemented features now pass [See Use Case A] [Alternate Flow 2 Possible]

**Alternate Flow:**

1. Before implementing a feature, one or more its test cases pass
  - a. There is an error in how the developer has written the test case. Developer revises the test case [Return to Basic Flow 2]
2. After implementing a feature, one or more test cases fails
  - a. There may be an error in how the test case was written. Developer revises the test case [Return to Basic Flow 4]
  - b. There may be an error in the implementation of the feature. [Go to Use Case C Basic Flow 3]

**Pre-conditions:**

- Planned plugin has features that are testable

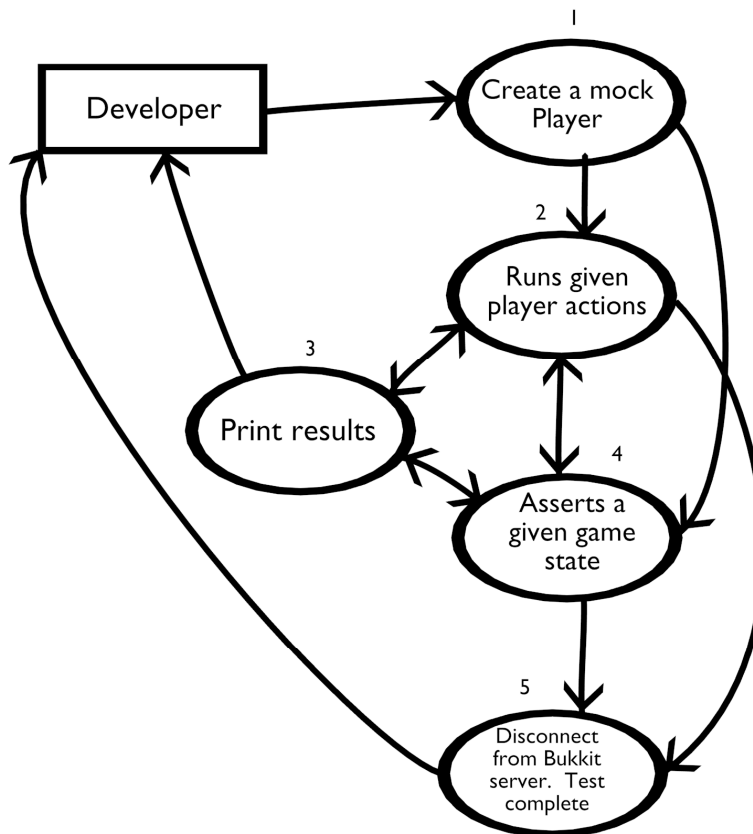
**Post-conditions:**

- All test cases pass

## Data Flow Diagrams

### Context-Level Diagram

Level 0 DFD



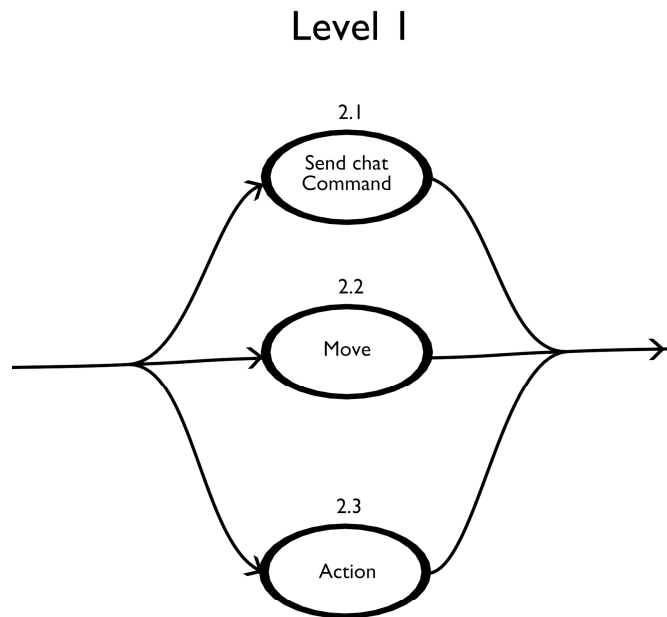
```
// This is an example on how a developer will write a unit test,
// developer will create a mock player (1)
MockPlayer myBot = new MockPlayer("localhost", portnumber);
myBot.connect();
```

```
// The developer can then run a series of player commands (2)
myBot.move(10, 0, 0);
myBot.action(LIZA_LEFT_CLICK, 5);
```

```
// The developer can assert given game states (4)
if (myBot.inventory.contains(Material.DIRT)) {
    assert false;
}
```

```
// The developer will disconnect the bot and the test
// will be complete (5)
myBot.disconnect();
```

## Run Player Commands Level 1



// The developer may send chat commands  
`myBot.sendChat("your text here");`  
`myBot.sendChat("/warp home");`

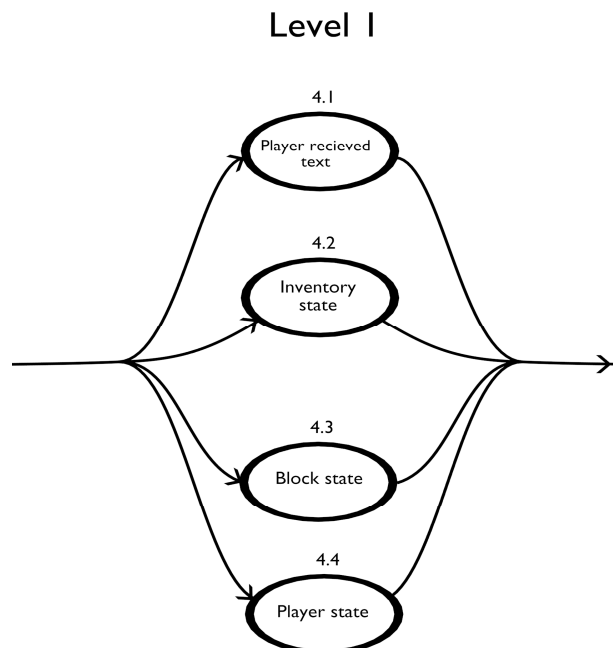
// The mock player is able to move in any direction

`myBot.move(x, y, z);`  
`myBot.teleport(x, y, z);`  
`myBot.face(x, y, z);`

// And the bot can perform a variety of other actions

`myBot.jump();`  
`myBot.sneak(true);`  
`myBot.sprint(false);`  
`myBot.action(LIZA_LEFT_CLICK, 5)`

## Asserts a given game state



// The developer may look for a string that has been sent through the chat  
`myBot.chatBuffer.search("string");`

// The inventory can be retrieved  
`Inventory i = myBot.getInventory();`  
`i.contains(Material.STONE);`

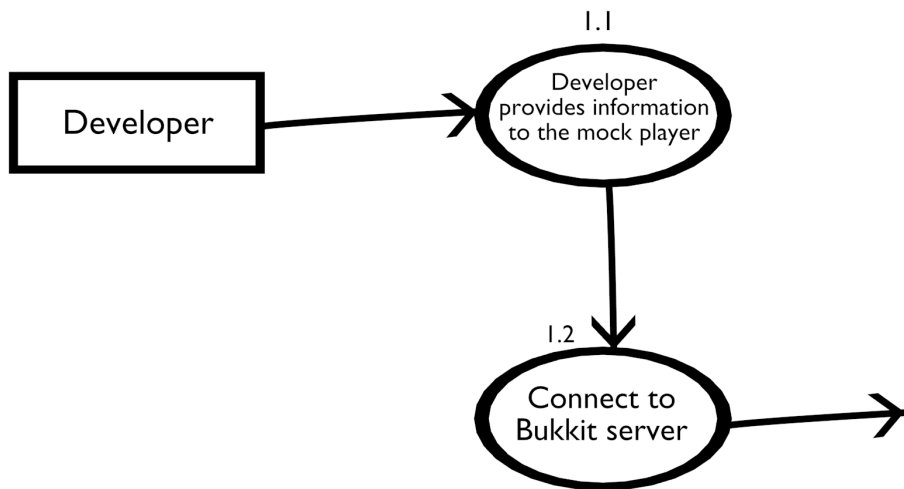
// Blocks located in the world can be retrieved.  
`Location l = myBot.location();`  
`Block b = myBot.getWorld().getBlock(l);`  
`b.getMaterial();`

// Data about the player can be checked  
`myBot.health();`  
`myBot.hunger();`  
`myBot.experience();`  
`myBot.armor();`  
`myBot.onFire();`



## Create Mock Player

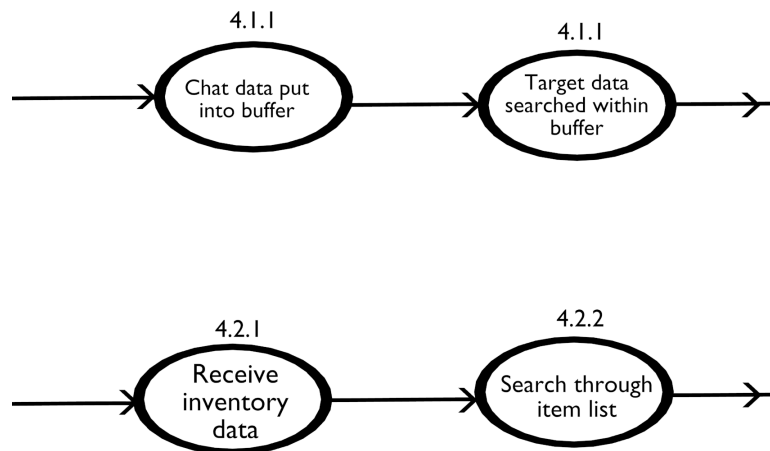
### Level I



As shown before, the developer will create a MockPlayer object, provided the server information where the test is to be run. The developer runs the connect method on the object to initiate connection with the server.

## Player received text & Inventory State

### Level 2



When the bot receives any messages through the in-game chat, it is placed into a buffer that the MockPlayer object possesses. The developer is free to search through the buffer for any string. For instance, a plugin may send a message to the player when performing a certain action. The developer can assert that this message was received through the buffer.

Bukkit [2] maintains the inventory of each player maintained as a list. This list can be retrieved, checked, and manipulated to fit the need of the developer.

## Functionality Requirements

- Requirement name [Mapped feature ID]

### Server Functions

- System must be able to create and remove mock players. [1]
- System must record chat dialogue in a buffer so that it can be referenced later. [2]
- System must be able to send chat commands through the mock player. [3]
- System must listen to Bukkit [2] events. [4]
- System could be able to enable or disable Bukkit [2] plugins. [9]

### Player Command Functions

- System must be able to move the mock player in a direction specified by the developer. [3]
- System must be able to teleport the mock player to a destination defined by the developer. [3]
- System must be able to allow the developer to have the mock player face a point in Minecraft [1] space. [3]
- System must be able to perform other movement actions (such as jumping, sneaking, swimming, sprinting, and flying) through the mock player. [3]
- System must be able to perform interactive actions (destroying blocks, placing blocks, block physics check, interacting with another entity, picks up an item, uses an item) through the mock player. [3]

### Entity State Functions

- System must be able to spawn and de-spawn other non-player entities on the server. [7]
- System must be able to read an entity's current health. [6]
- System must be able to read an entity's current location. [6]
- System must be able to retrieve a list of all entities in the Minecraft [1] world. [2]
- System must be able to retrieve a list of entities within a proximity of the mock player defined by the developer. [2]
- System must be able to remove all entities within a proximity of the mock player defined by the developer. [7]
- System must be able to read an entity's special conditions (being on fire, drowning, suffocating, poisoned, sprinting, sneaking, shooting, attacking, falling, sleeping, eating). [6]

### Player State Functions

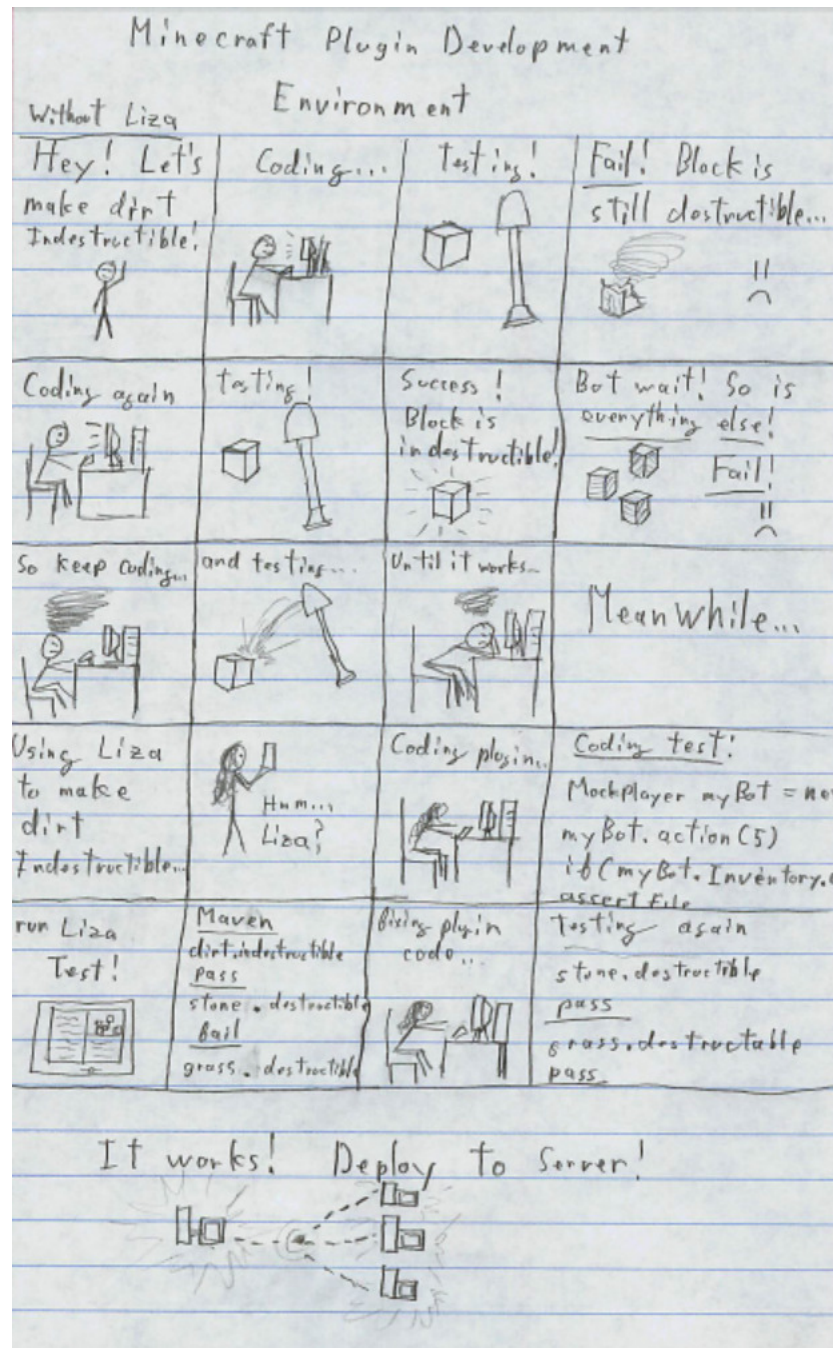
- System must be able to read a player's armor value. [6]
- System must be able to read a player's hunger meter. [6]
- System must be able to read a player's experience bar. [6]
- System must be able to read a player's inventory. [6]

- System must be able to read a player's equipped items. [6]

### **World State Functions**

- System must be able to create and remove world blocks on the server. [7]
- System must be able to retrieve a block at a location in Minecraft space. [2]
- System must be able to retrieve a list of blocks that a player is facing. [2]
- System must be able to retrieve properties of a block (such as material type, location, blast resistance, light level, opacity, powered, on fire, flammability). [6]
- System must be able to modify the properties of a block (as listed above). [2]
- System must be able to determine and adjust the time in the Minecraft [1] world. [2]
- System must be able to determine and adjust the weather in the Minecraft [1] world. [2]

## Storyboard



In this storyboard, a developer wishes to create a plugin that makes a block indestructible. He spends a lot of time testing his plugin, only to get frustrated when it does not work as intended. Meanwhile his friend is aware of the Liza testing framework. With this discovery, she can quickly write a test which discovers the plugin error. After a couple of thorough tests, she is able to make a working plugin that she is able to upload to a public server.

## References

- [1] Mojang AB. Minecraft. [Online]. <http://www.minecraft.net/>
- [2] Bukkit. [Online]. <http://bukkit.org/>
- [3] Brett Porter and Jason Zyl. Apache Maven Project. [Online]. <http://maven.apache.org/>
- [4] Oracle. Java. [Online]. <http://www.oracle.com/us/technologies/java/index.html>

## Index

application programming interface, 4  
Bukkit, 4, 5

*Minecraft*, 4, 5  
plugins, 4

## Glossary

**Blocks** – Blocks are stationary objects within the Minecraft [1] world which represents different materials.

**Bukkit** [2] – A wrapper for the *Minecraft* server that exposes a user-friendly API.

**Entity** – An Entity is an objects in the Minecraft [1] world which is not a block. Everything which can move around freely within the Minecraft [1] world is an entity.

**Maven** [3] – A software project management tool that builds and tests Java code.

**Minecraft** [1] – A sandbox computer game where players place and destroy blocks.

**Minecraft world** – The world consist of all the blocks and entities on a server

**Plugin** – A server-side modification to the game that alters the behavior of certain actions

**Sandbox game** – Refers to a style of game that involves an open world and no concrete directive