ROSE-HULMAN INSTITUTE OF TECHNOLOGY

Team Mohan

Milestone 4

Samuel Kim, Brian Collins, Michael Williamson, Kevin Geisler

10/28/2011

Contents

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 provide details on how Liza intends to solve them. This document will focus on the coding standards that will be followed throughout development. It will also address how change requests will be handled through a source control system. Lastly, we will introduce the how the test cases will be used to verify the integrity of the Liza system.

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 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.

In the previous milestone, there was a list of functional requirements and use cases that our system must perform. These will drive the test cases that are listed in this milestone.

Project Background

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...	- Bukkit [2] plugins go untested and therefore are unstable. - Importing plugins to test by trial and error is a lengthy process. -Testing code on a server with informative results is unimplemented
Affects...	Plugin Developers Server Hosts Players
And results in...	-Server crashes or plugin bugs occur. -Players have a frustrating experience on a server due to frequent crashes and lag. -Developers spend more time debugging and constantly exporting their code to test on a real server.
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* to 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* to 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 ran 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

Functionality Requirements

Category [Category ID]

1. Requirement name [Mapped feature ID]

Server Functions [SF#]

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

Player Command Functions [PCF#]

1. System must be able to move the mock player in a direction specified by the developer. [3]
2. System must be able to teleport the mock player to a destination defined by the developer. [3]
3. System must be able to allow the developer to have the mock player face a point in Minecraft [1] space. [3]
4. System must be able to perform other movement actions (such as jumping, sneaking, swimming, sprinting, and flying) through the mock player. [3]
5. 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 [ESF#]

1. System must be able to spawn and de-spawn other non-player entities on the server. [7]
2. System must be able to read an entity's current health. [6]
3. System must be able to read an entity's current location. [6]
4. System must be able to retrieve a list of all entities in the Minecraft [1] world. [2]
5. System must be able to retrieve a list of entities within a proximity of the mock player defined by the developer. [2]
6. System must be able to remove all entities within a proximity of the mock player defined by the developer. [7]
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 [PSF#]

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

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

World State Functions [WSF#]

1. System must be able to create and remove world blocks on the server. [7]
2. System must be able to retrieve a block at a location in Minecraft space. [2]
3. System must be able to retrieve a list of blocks that a player is facing. [2]
4. 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]
5. System must be able to modify the properties of a block (as listed above). [2]
6. System must be able to determine and adjust the time in the Minecraft [1] world. [2]
7. System must be able to determine and adjust the weather in the Minecraft [1] world. [2]

Coding Standards

Our coding standard will loosely follow those published by Sun Microsystems [5] for Java including, but not limited to, the following.

Indentation

- The unit of indentation shall be four spaces. Tabs shall not be used.

Line Length

- Lines longer than 80 characters should be avoided

Comments

- Document comments shall be included for each class and method. These should follow normal Javadoc convention.

White Space

- Blank lines should be present between methods, local variables and its first statement, and between logical sections inside a method

Naming Conventions

- Classes – Names will be nouns describing what the class represents. The first letter of each word should be capitalized. (example: MockPlayer)

- Methods – Names will be verbs describing what action is performed. The first letter should be lowercase, with each internal word beginning with a capital, also known as camel case. (example: isOnFire())

- Variables – Names should be short yet meaningful. One-character variable names should be avoided, except for temporary variables. (example: boolean onFire = true)

- Constants – Names should be all capitalized, with words separated by an underscore. (example: int FIRE_BLOCK = 51)

Other Formatting

- Opening brackets will be placed on the same line as the statement which require a bracket

Programming Practices

- Variables in classes should be kept private as much as possible. Typically, class fields will be modified and retrieved as an effect of a method.

Change Control

Our change control system will be handled through GitHub [6]. Through the “Issues” tab in the repository, one can file an issue, bug, or new feature. Each is given a title and a meaningful description. Labels may be added, which we plan on using to differentiate between bugs and proposed changes.

When a new change is added to this list, the team will meet and discuss the change. We will be concerned with the time needed to make the change, the availability for a team member to work on the change, and the feasibility of the change. In addition, we will seek the thoughts of the client about the change, if the request is from an external source.

If a change request is accepted, it is labeled as accepted and assigned to a team member (all handled through GitHub [6]). After the change has been made, the issue is then marked as closed, and any relevant documentation is updated. If the change request is rejected, it is labeled as rejected and the issue is closed.

All development work on Liza is handled through GitHub [6].

Test Cases

It is vital for any testing framework to work reliably. Because of this, we will be using Bukkit's [2] API to assert that the mock player's states are correct through a testing plugin. In general, we will be using Bukkit's [2] API (which is assumed to be correct) to check against Liza's values. For control's sake, the test plugin will be run on a world that is generated to be completely flat, with no obstacles or pitfalls, and natural entity spawning disabled. If a test case would require such things, the testing plugin will be able to spawn them in manually.

In the descriptions, the terms "testing plugin" (that will be used to test Liza) and "Bukkit" will be used interchangeably.

Test Case No.	Tested Functionality	Description and Expected Result
1	SF1	Bukkit [2] can retrieve a list of all connected players. The test will begin by retrieving this list. Liza will attempt to log in the mock player. The list is retrieved again. The mock player is expected to be added to this list.
2	SF2	Through Bukkit [2], a message can be sent directly to the mock player. Liza will search through the buffer and attempt to find this message.
3	SF3	Liza will attempt to send a command through the chat. Bukkit [2] has a callback function which retrieves any commands that are sent along with the sender. We can assert that the command was sent correctly and by the mock player.
4	SF4	Liza will send a command through the chat to the server. The testing plugin will read this command, and send out a predetermined event. It will be asserted that this event was received, and that all the values match.
5	SF5	The testing plugin will also come with an additional plugin which makes dirt indestructible. Liza will attempt to destroy this block. Bukkit [2] will assert that the dirt block still exists. Liza will attempt to disable this plugin, and destroy the block again. Bukkit [2] will assert that the block is destroyed, and will replace it. Liza will attempt to re-enable the plugin, and destroy the block. Bukkit [2] will once again assert that the block still exists.
6	PCF1 and PCF2	Bukkit [2] will retrieve the current location of the mock player. Liza will attempt to move the player in a specified direction or teleport to a location. The testing plugin will assert that the mock player is in the correct location. This test will be repeated multiple times for every direction.
7	PCF3	Bukkit [2] keeps track of the pitch and the yaw of a player. Given the player's location along with the target point, the expected viewing angles can be calculated. Bukkit [2] will retrieve the player's viewing angles and compare them to the expected values.

8	PCF4 (swimming)	Bukkit will encase the mock player in water. Then the testing plugin will assert swimming much like it does with regular movement.
9	PCF4 (sneaking and sprinting)	Liza will attempt to send the command to sneak. Bukkit can read the player's sneaking state. A similar process follows for sprinting.
10	PCF4 (flying)	Bukkit [2] will enable "creative mode" for the mock player. This will allow the mock player to fly. Bukkit [2] will retrieve the current location of the mock player. Liza will attempt to fly the player in a specified direction. The testing plugin will assert that the mock player is in the correct location. This test will be repeated multiple times for every direction.
11	PCF5 (place and destroy blocks)	Liza will attempt to place a block. The testing plugin will assert that the block exists. Liza will attempt to destroy that block. The testing plugin will assert that the block does not exist.
12	PCF5 (interacting with entities)	The testing plugin will spawn an entity at the mock player's location. Liza will proceed to attack this entity. This will trigger an event, which the testing plugin will retrieve.
13	PCF5 (picking up items)	The testing plugin will retrieve the mock player's inventory, and sets it so that a target item is not currently possessed by the mock player. The testing plugin will then proceed to drop that item onto the ground near the player. The mock player's inventory is once again retrieved, and the presence of the item is asserted.
14	PCF5 (using an item)	The testing plugin will give the mock player a specific item, such as a bow and arrow. Liza will attempt to use this item. Bukkit [2] will respond by throwing an event, which the testing plugin will receive.
15	ESF1	Bukkit [2] will retrieve a list of entities and their location on a server. Liza will attempt to add an entity (such as a cow) at a certain location. The testing plugin will assert that the list of entities has been modified and an entity is located at the determined place. Liza will then attempt to despawn the entity, and the testing plugin will assert that the list has been modified and that there is no entity at the specific location.
16	ESF2, ESF3	Liza will attempt to retrieve an entity's health and location. Bukkit [2] will retrieve the same entity's health and the testing plugin will assert that they are the same.
17	ESF4	Liza will attempt to retrieve a list of all entities in the Minecraft [1] world. Bukkit [2] will retrieve the list off of the server. The two lists will be compared and asserted to have the same values.
18	ESF5	Liza will attempt to retrieve a list of all entities in a predetermined radius from the mock player. Bukkit [2] will retrieve the list of all entities and their locations, and will calculate and find those that are within the same specified radius of the mock player. The two lists will be compared and asserted to have the same values.

19	ESF6	Liza will attempt to remove all entities within a predetermined radius from the mock player, while generating a list of removed entities. The testing plugin will retrieve the list of all entities on the server, and assert that none of the removed entities are present.
20	ESF7	The testing plugin will give a predetermined special condition (such as being on fire) to a specified entity. Liza will attempt to retrieve this information, and the testing plugin will assert that they are the same.
21	PSF1	Both Bukkit [2] and Liza will retrieve the list of equipped items on the mock player, and run calculations to find the armor value. The two are compared. Then the mock player will change its armor configuration by removing/equipping armor. Bukkit [2] and Liza will once again retrieve the equipped items and compare computed armor values.
22	PSF2	The mock player will wait until hunger is partially depleted. Bukkit [2] will retrieve the hunger value of the player, and assert that Liza's value matches. This hunger value will be altered, either by eating food or waiting longer, and Bukkit [2] and Liza compare values again.
23	PSF3	Before killing any entities, the testing plugin will assert that the mock player has zero experience. Then, after killing some entities, the testing plugin will retrieve the player's experience and assert that it matches Liza's value.
24	PSF4, PSF5	The testing plugin will clear the mock player's inventory. Liza will give the mock player a predetermined set of items. Bukkit [2] will read the item values in the player's inventory and assert that each item is present.
25	WSF1	Liza will attempt to create a block at a given location. The testing plugin will assert that the block exists. Removal is achieved by creating an air block at a location.
26	WSF2	Liza and the testing plugin will retrieve a block at a given location. The two are compared and asserted to be the same.
27	WSF3	Bukkit [2] is able to retrieve blocks that a player is facing. This value is compared to Liza's and asserted to be the same.
28	WSF4	Given a block, the value that the testing framework and Liza retrieves are compared and asserted to be the same.
29	WSF5	Given a block, Liza will attempt to set some material properties. The testing plugin will retrieve the block and read its properties, and assert that they have been set.
30	WSF6	Liza will attempt to set the time of day in the world. The testing plugin will assert that the time matches what was set.
31	WSF7	Liza will attempt to adjust the current weather at a location. Bukkit [2] will assert that it has been done.

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

[5] Oracle. Code Conventions for the Java Programming Language. [Online]. http://www.oracle.com/technetwork/java/codeconv-138413.html

[6] GitHub. [Online]. https://github.com/

[7] Eclipse. Eclipse. [Online]. www.eclipse.org

Index

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