INSTANCE-BASED CUCUMBER TESTING

COMP 4905 – Honours Project

Author: Matthew Walker

Supervisor: Dr. Jean-Pierre Corriveau

School of Computer Science

Winter Term, 2019

Carleton University

mattwalker@cmail.carleton.ca

Abstract

The focus of this project was to investigate the ability of the Cucumber framework to support user stories that are instance-based. More precisely, to determine whether it is possible to use the 'example table' mechanism of Cucumber in combination with a modeling and testing approach to allow user stories to refer to conceptual instances then mapped to actual instances of the implementation under test (IUT). This would make testing user stories with Cucumber more powerful than it currently is, as it would allow for monitoring any changes in state of the instances used in the backend of an IUT. These changes in state would demonstrate the behavior of the IUT at its core, exposing any and all inconsistencies, and/or further proving its functionality.

Table of Contents

List of Figures

1 Introduction

When testing software with the use of Cucumbers behavior driven approach, the users involved in developing test cases (business analysts, product owners) can provide helpful user stories for quality assurance engineers; however, the manner with which they are used can be lacking in complexity. This is due to the nature in which Cucumber's 'scenarios' are implemented when defining user stories to be used in the actual test code. When a user story is defined in a Cucumber Scenario, for example: Given I login into the library - When I withdraw the book 'x' - Then I have the book 'x', the use of an example table would be required. This example table would map actual names of books to be used in this scenario, with the scenario being ran repeatedly for each unique example in the table. This is making use of Cucumbers built-in parameterization of step methods, which supports passing primitive data expressions from the feature file to the step definition file. However, this means that the use of unique examples within code could only be used for the purpose of input into the program, to then produce an expected output that should pass the test.

This project investigated the possibility of using the unique inputs from Cucumbers example tables to access the actual instances of the objects in question, and in doing so grant the ability to track how the object changes throughout a scenario. This would provide a much more powerful use of testing to Cucumber, as tracking an object from the beginning to end of a user story could provide greater insight into how the state of an object behaves, further testing whether this behavior is correct.

2 Instance-Based Testing

2.1 Motivation

In Model-Based Testing, much work has been done on the generation of functional test cases. But few approaches tackle the executability of such test cases. And those that do, offer a solution in which tests and test cases are not directly traceable back to the actual behavior of an IUT (Implementation Under Test) [1]. This project was intended to find a way to demonstrate actual behavior with the use of accessing actual instances of the objects which exhibit the behavior being tested. This is far more reliable than the common case in which tests only simulate behaviors when dealing with individual features of a program. Transitioning from tests simulating parts of the program, to tests performed on the program while it is running, provides more in-depth assurances of the program's functionality, while also being more reliable.

2.1.1 Desire for Bindings

Binding examples of conceptual objects provided by a user to the actual instances of those objects could provide far greater testability than existing methods. A large draw back from current techniques is that they typically can only observe the state of the program at the user interface, or 'front-end' level. This context of testing can provide some insight into the state of the server, or 'back-end' level, through packets it sends to the user interface. However, these packets will only contain enough information to be relevant for the user interface to operate. There is certainly more information behind the scenes which would

be extremely useful for demonstrating that the program is functioning as intended, and that the internal logic hidden from the user is correct.

2.1.2 Stop Testing with Mock Objects

When the glue code corresponding to a Cucumber scenario is developed, any reference to a conceptual object would likely require the developer to somehow access an actual object, to simulate the scenario. The developer would need to simulate the conditions outlined in the scenario by creating the necessary objects as mock objects, input the variables needed for execution, and then execute the methods required to create the output which will be tested for correctness. This has many limitations. The primary drawback being that the developer is not testing the IUT at run-time. When the actual object is created by the program, and utilized at run-time, real conditions are created, which would allow testing of actual functionality, rather than what can be simulated. This way, there shouldn't be any surprises; simulated Scenarios could be incorrectly depicting the conditions that exist during run-time, resulting in inaccurate tests.

2.2 Methodology

Upon approaching how to accomplish binding the conceptual instances provided by an example table to the actual instances existing within back-end code, you must consider the conditions required for this to even be possible. Firstly, this can only work if the software being tested is running at full capacity during testing. The instances which need to be bound to, must exist in memory for them to be accessed, and furthermore can only be used in a scenario at run-time when you want to demonstrate a change in state as the

programs execution progresses. Secondly, at the same time the back-end processes are running, the front-end processes must also be running, and must receive the inputs that would be excepted from the user story defined in a scenario. This means that the user interface would need to be automated in some way and provided with the correct input to simulate a user story.

For the purpose of this project, a networked, turn-based computer game derived from the 1995 board game 'Quests of the Round Table', created by Gamewright, has been used. Being that this game is networked, it always requires a server to be running, and at least two clients must connect to that server for a game to begin.

2.2.1 Implementation Under Test

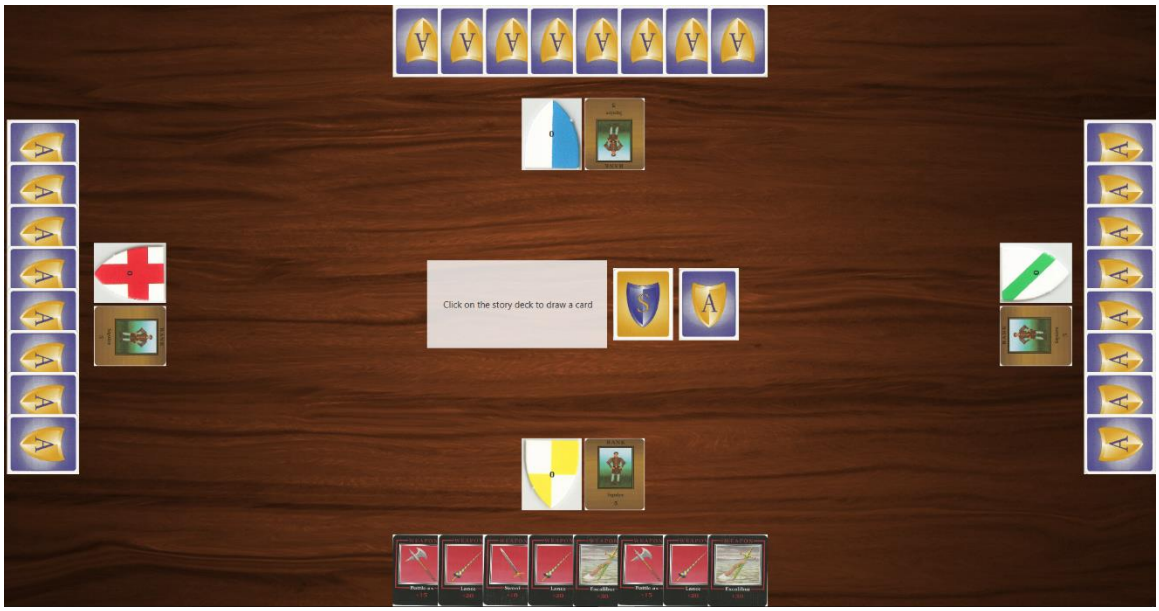

FIGURE 1: Client Window

The development of the networked, turn-based computer game 'Quests of the Round Table' was done so for the COMP 3004 course, taught by Dr. Jean-Pierre Corriveau. Students of the class were put into teams of four, with each team assigned the same goal

of turning the 1995 board game into a multiplayer computer game. The implementation used to serve as an IUT for this project was created by Ahmad Farhat, Mohamed Cheaito, Siraj Ahmadzai, and Abhinav Gurung, and was chosen out of the many versions made by other teams because it was considered the most complete. Being that it is a turn-based game, it operates in a synchronous fashion, meaning that all the clients (seen in Figure 1) connected to the same server must be in sync with each other, otherwise they could get out of turn, or display inconsistent information. In this case, object behavior can easily be predicted, and the state of information held within the actual instances of objects will also be predictable as it can be tested between turns, while there is a pause in the server's execution.

2.2.2 Parallel Cucumber Threads

To run a feature file, which contains the user stories/scenarios pertaining to a particular feature of the program, there needs to exist a corresponding step definitions file which contains the actual code (glue code) used to test these user stories, as well as a runner file, which ties the scenarios in the feature file to the step definitions, and executes the tests. The runner is executed on a single thread, which iterates through the scenarios in the feature file, testing them sequentially. Because of this, we know that each end of the program (server or client) will be executing on separate threads. As such, to test the game 'Quests of the Round Table', a minimum of three threads will be required for a game to run. One thread will be needed for the server to run on, and since the game requires a

minimum of two players, two clients must connect to the server, with each requiring their own thread.

However, this configuration does not support testing of the entire IUT. With feature files and step definitions defined to run and test the server and clients, each will have separate runners, making it impossible to run a single test on the entire program. To fix this, we must execute the runners simultaneously and in parallel with each other. Cucumber does apparently support running in parallel natively, as of version 4.0.0 [2], however an easier approach was found by using a plugin known as Maven Surefire. Surefire adds the ability to run multiple runner files in parallel with each other, thus allowing a single test suite to execute and test a server-client network, as exists within 'Quests of the Round Table'.

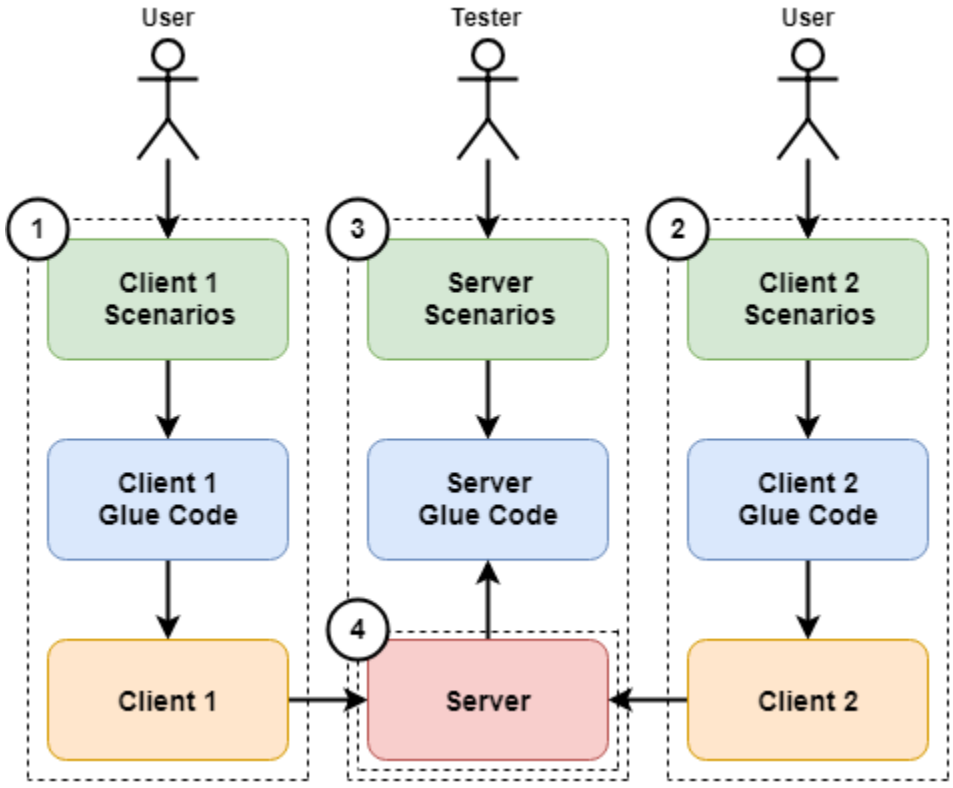

FIGURE 2: Parallel Threads

Figure 2 shows the task of each thread, and how each thread corresponds to other threads, with the arrows representing information being sent. Groups 1 and 2 depict the threads which automate the two clients connecting to the server. Each contains client scenarios defined in their Cucumber feature files, mapping to the step definitions defined in the glue code, which launches and automates their own client window. Group 3 depicts the thread which creates the server and can access actual object instances from the server, and group 4 depicts the thread spawned within group 3, tasked with running the server, which group 3 created. It is this nested thread tasked with running a server object created outside of itself that grants the ability to access objects that exist and are being used during run-time.

2.2.3 Dependency Injection

The server will run on its own thread, and when running in parallel, this works just fine when only testing client scenarios. However, when approaching the task of accessing object instances through server scenarios, the thread which is running the server cannot also run scenarios; it can only run the server. So, solving this means the addition of another thread, spawned within the glue code for the server scenarios, with the purpose of running the server. By doing this, we can create the server object in the glue code, create the new thread in the glue code, and then pass the server object into the new thread. The new thread's start method will start the server, providing the glue code access to a server object which is running, without impeding the glue codes execution. This method is a simple form of dependency injection, with the glue code depending on the

server object, then injecting it into the new thread to configure and run the server object externally once it's start method is triggered.

2.2.4 Binding Object Instances

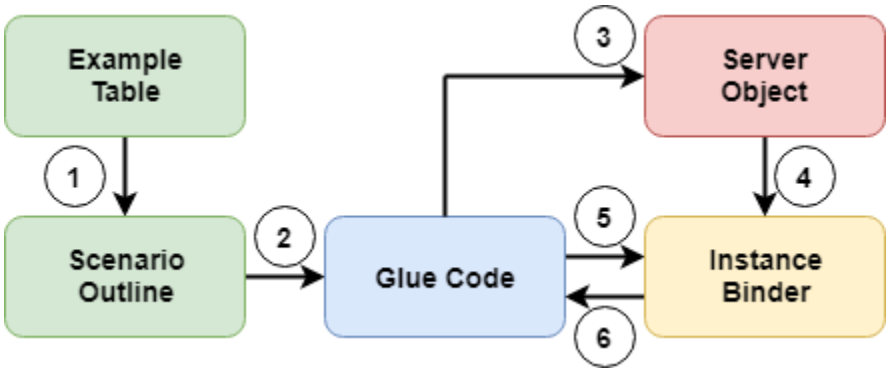

FIGURE 3: Turning Conceptual Instances into Actual Instances

Now that the glue code has access to a server object which is running on its own separate thread, object instances can be drawn out of it during runtime. Within the scenario outline defined in the feature files, these object instances would never be explicitly mentioned (they would be referred to conceptually). They will either be expressed as an incremental statement, such as player one or player two, or by the name of what the object represents, such as the name of a card in 'Quests of the Round Table'. When these conceptual instances are defined in an example table, the scenario that they correspond to will be ran repeatedly for each row of instances contained in the table, using the primitive values defined in each row. Arrow 1 in Figure 3 represents the transfer of the conceptual instances from the example table to the scenario outline. The scenario outline then has its steps mapped across to the glue code, passing the conceptual instance defined as a primitive value along with it, shown by arrow 2. Before the glue code can do anything with the conceptual instance, it always ensures that the server object is running,

as shown by arrow 3, and then passes that server object over to the instance binding class, shown by arrow 4. At this point, all the glue code needs to do to convert a conceptual instance into an actual one, is to define what the instance refers to as an object, call the instance binders 'bind' method by passing in the objects class type and any incremental value (if the instance exists in a list of some kind), and then the actual object instance will be returned, ready to be used for testing. Arrow 5 in figure 3 represents this process of the glue code passing primitive values provided by the example table to the instance binder, with arrow 6 representing the instance binder returning the actual object, that was fetched from the server object.

The instance binder can do this, because the server object is essentially the root of all objects. The instance binder contains an overloaded method called 'bind', which has generic return values, since the object type being returned could be anything. Each version of 'bind' takes at least the class type of the object being requested as a parameter, so that it knows what type it is returning, as well as what object it is getting from the server object; other overloaded version of 'bind' would take any index values required to access an object instance which is stored in a list of some kind. This is the key step where the test code actually accesses the server code, since these objects will need to be accessible from the server object through getters and/or public variables. The existence and relationship of objects in any IUT will be unique, and because of this, the developer will have code into the instance binders' methods how to get objects from the IUT being used in their case.

2.2.5 Writing Cucumber Scenarios

Being that the nature of accessing actual instances of objects at run-time means reaching into back-end code, Cucumber scenarios must be written for both the front and back end. When dealing with scenarios for the front-end, this is a standard case for use of Cucumber, since the front-end client window is the only thing that users will ever interact with, and as such, users of some kind would write their user stories/scenarios for use of front-end testing. Figure 2 shows how users would write the scenarios for each of the clients used in this case, however it also shows that a tester writes the scenarios for the server, rather than a user writing them. This difference is because users would never have scenarios describing use of the server, however they may be indirectly referencing the server in their client scenarios without realizing it. Because of this, a tester would have to write the server scenarios, but could refer to the client scenarios and how they tie into back-end processes; the user would be indirectly outlining the server scenarios for the tester. For example, if a user writes a client scenario in which they connect to the game, wait for their turn, and draw a card, the tester would then write a corresponding server scenario in which a player connects to the game, and upon it being that players turn, that player draw a card. With all threads running in parallel with each other, the server scenarios can simply follow along as the client scenarios progress through the game, testing the object instances as they are used in the game's logic to further progress the game as the clients continue to play.

2.3 Results

Once this method of binding conceptual instances to actual instances was functional, all that was left was to try using it. The nature of accessing and testing the state of objects that are currently being used by the IUT at run-time, provides testing opportunities that are new and more powerful that simulated ones. Once the automation of the IUT is fully functional, and the steps it will follow are all known, the change in state of the object instances should be completely predictable, making test assertions easier to write, and easier to understand if they fail. All relevant object instances that a user would refer to conceptually could be tested in a way which is even more influenced by what a user is expecting from the IUT. This even provides a more straight-forward method for developing tests, as the behavior expected by the user can be directly compared to the actual behavior exhibited by the objects used in the IUT.

2.3.1 Use of Object Binding

Each unique IUT will largely determine how object binding will be useful in tests. Being that the IUT used in this case is a synchronous networked game, object binding can be used to track the state of an object as it changes throughout a game's progression, as well as show any inconsistencies between information stored in the back-end vs. information displayed in the front-end. For example, testing a scenario that spans over the turn of a particular player would infer that the player object instance would be needed, as well as the instances of any cards used throughout the turn, to test that the state of these objects are changing in ways that are expected and correct. Furthermore, the client scenario that

is working in tandem with this server scenario would have visual representations of these instances and would be expecting these values shown in the interface to be the same as the actual values held by the objects that they represent.

2.3.2 Constraints

While having the ability to perform tests on object instances that are being use at run-time, the tester must be aware of constraints and limitations of these bindings. The most dangerous being a mistake where an object is being referenced before it has been created by the IUT. This is very possible if the scenario is incorrectly written, in the sense that it is assuming the game has progressed to a certain state, before it has, resulting in the reference of an object that will exist, but doesn't yet. It is up to the whom ever is writing the user stories to ensure that they are not missing any crucial steps involved in the scenario's progression. Another limitation of accessing object instances, is that the conceptual instance defined in an example table must contain all relevant information needed to understand what unique instance is being referenced. For example, if a scenario means to refer to the fifth card in a players hand, buy only defines the name of the card in the example table, then a problem could arise if the player has multiple cards of the same name in their hand, since there is not enough information to define which one of these multiple cards is the one needed to be returned by the instance binder. These constraints are largely avoidable when testing a synchronous system, as it is primarily determined by the quality in which the user stories are written, and the assurance that they contain all relevant information needed to perform their tests.

3 Conclusions

3.1 Synchronous vs. Asynchronous Testing

This project succeeded in finding a way in which conceptual objects instances can refer to actual object instances when testing a synchronous system. Since 'Quests of the Round Table' was used as the IUT, it operated synchronously as all members of its network remained in a constant rhythm with each other, and since it is a turn-based game, a defined order is understood by both the users and the testers, allowing thread safe accesses of object instances, while also keeping everything predictable. However, if this methodology for accessing object instances was used on an asynchronous system, there would be problems and inconsistencies. This is due to the nature of an asynchronous system being unpredictable, as clients are acting independently from each other, removing any ability of any one client knowing whether certain objects exist, or what state they are currently in. In addition to this, it is dangerous trying to access objects in memory when there is no way of knowing when other processes are accessing them. An asynchronous system would need to be far more robust, with explicit handlers for cases in which objects have changed unexpectedly, are in use by another client, or no longer exist.

3.2 Possible Alternative Approaches

Throughout development of this project, ideas for alternative ways of accessing of object instances at run-time have come up but were never fully investigated. The first of which being the use of an intermediate database in which references to all objects used by IUT

would be stored, and from which the example table used for a scenario outline would simply contain values used to query the database for the objects being referred to. The problem seen with this approach is that the IUT would either need to be heavily modified to support use of this database, by changing every area where an object is created so that it also immediately stores a reference to that object in the database, or the IUT would need to have been developed this way from the beginning.

A second idea for an alternative approach would be to modify Cucumber's built in anonymous object mapping ability [3]. This approach was investigated in the beginning of this project, although it was found that the custom object mapper is not in any way intended to map existing objects. However, it may be possible to modify the object mapper to stop creating new objects, and instead pass objects fetched from the server object, but it would need to have a reference to the server object, in the same way the object binder does. This approach would not be all that different from the method developed in this project, as it is just way to anonymously fetch object instances, similar to how the instance binder class already does in a generic way.

3.3 Remove Manual Dependency

The ultimate goal for further developing this method of testing, is to remove any and all manual accesses of the object instances which are being referred to in user stories. The project unfortunately does use manual accesses, as the instance binder class resorts to using getters and/or public variables to return the objects existing within the back-end code. If a method was developed without the use of manual accesses, it would work in

the same way that Cucumber already passes primitive values through the example table into parameterized step definitions defined in the glue code. Rather than these parameters simply existing as primitive values, they could instead exist as the actual objects being referred to in the scenario, immediately available to be tested for correctness. A method which would achieve this would still require that the IUT is operating at fully capacity during testing, meaning the use of parallel cucumber threads, and likely some form of automation. However, writing and testing user stories with this method would be extremely intuitive, and would create undisputable tests.

Reference

[1] Arnold, D., Corriveau, J-P., & Shi, W. (2010). Modeling and Validating Requirements using Executable Contracts and Scenarios. Retrieved April 2, 2019, from Carleton University School of Computer Science: http://people.scs.carleton.ca/~wei_shi/SERA10.pdf

[2] Korstanje, M.P. "Announcing Cucumber-JVM v4.0.0." Cucumber, Oct. 2018, cucumber.ghost.io/blog/announcing-cucumber-jvm-4-0-0/.

[3] Korstanje, M.P. "Announcing Cucumber-JVM v4.2.0." Cucumber, Oct. 2018, cucumber.ghost.io/blog/announcing-cucumber-jvm-4-2-0/.

Glossary

Phrase/Abbreviation	Definition
Actual Instance	The object which is being used in code
Back-end	Code running the server of a networked system
Conceptual Instance	The real-world equivalent of an object used in code
Feature File	Cucumber file containing scenarios pertaining to a single feature exhibited by the IUT
Front-end	Application running the client of a networked system
Glue Code	Java class containing code to run Cucumber scenarios
Instance Binder	Java class which returns actual object instances
IUT	Implementation Under Test
Mock Object	Object created to simulate behavior within an IUT
Primitive Values	Simple data types in java, such as int, boolean, string, float
Runner	Java file used to compile and execute Cucumber feature files with their corresponding glue code files
Run-time	Time in which the IUT is running at full capacity
Scenario	Cucumber user story, written in a Cucumber feature file
State	The current values assigned to variables in code
Step Definition	Java method defined glue code, which corresponds to a single unique step in a Cucumber scenario
Thread	An independent line of execution in a program
User Story	A short descriptive use of one feature exhibited by a program, written by a user