Mikko Niiranen

Model Testing Framework for Next Games Oy

NEXT GAMES

Bachelor of Business Administration

Autumn 2016

KAJAANIN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

KAJAANIN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

ABSTRACT

Author: Niiranen Mikko

Title of the Publication: Model Testing Framework for Next Games Oy

Degree Title: Bachelor of Business Administration

Keywords: testing, software, game development

This thesis was commissioned by Next Games Oy, a Finnish mobile gaming company located in Helsinki. The purpose of this thesis was to create a framework that would enable game developers to write automated tests and encourage to increase automated testing, leading to better product quality. Example uses for the test framework would be testing new features, security testing, migration testing and regression testing.

As a foundation for creating the testing framework the solutions for automated testing were studied. This included examining tools provided by Unity and extending those tools to allow simple game specific testing. With this knowledge, a set of classes were created that help developers by initializing the game model with determined data and creating a base for the tests. Different methods for the test initialization were tested and the best one was chosen with the priority of making the framework easy to use and minimizing the impact on compilation times. Methods for running the test both on developers' machines and as a part of continuous integration were investigated.

The Unity Editor Test Runner which implements the NUnit unit testing framework was chosen as the test platform due to being included in Unity and therefore being easy to integrate to Unity workflow. Tests were easy to execute inside Unity as well as a part of the continuous integration. The framework that was created successfully initializes the player model from given static and variable data of the game and player state. Because JSON format, which is relatively slow to deserialize, is used to store this data the best way to initialize game model was to deserialize the data when the test is run. This means that deserializing happens only for tests currently under development on the developer's computer instead of all the tests that are already made.

The framework was presented to all the programmers of Next Games. Additional hands-on workshop is planned in order to give better understanding on how to utilize the testing framework.

KAJAANIN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

TIIVISTELMÄ

Tekijä: Niiranen Mikko

Työn nimi: Testausjärjestelmän kehitys Next Games Oy:lle

Tutkintonimike: Tradenomi, tietojenkäsittely

Asiasanat: testaus, ohjelmistot, pelinkehitys

Tämä opinnäytetyö toteutettiin Next Games Oy:lle, joka on suomalainen mobiilipeliyritys Helsingissä. Tämän työn tarkoituksena oli luoda sovelluskehys, joka mahdollistaisi automaattisten testien kirjoittamisen pelinkehittäjille ja rohkaisisi lisäämään automatisoitua testausta, johtaen tuotteen korkeampaan laatuun. Esimerkki käyttötapauksia testi sovelluskehykselle ovat uusien ominaisuuksien testaaminen, turvallisuus testaaminen, migraatio testaaminen ja regressio testaaminen.

Testausjärjestelmän pohjaksi perehdyttiin automatisoidun testaamisen ratkaisuihin. Tämä sisälsi Unityn tarjoamien työkalujen arviointia ja niiden kehittämistä mahdollistamaan yksinkertaista peli kohtaista testaamista. Näiden tietojen perusteella toteutettiin joukko luokkia, jotka luotiin helpottamaan pelin mallin alustamista ennalta määritellyllä datalla testikäyttöön. Mallin alustamista kokeiltiin useilla eri tavoilla ja niistä valittiin paras huomioiden helppokäyttöisyyden ja mahdollisimman pieni vaikutus käännös aikoihin. Lisäksi mahdollisuudet testien suorittamiseen sekä kehittäjän omalla koneella, että osana jatkuvaa integraatiota selvitettiin.

Unity Editor Test Runner, joka hyödyntää NUnit yksikkötestaussovelluskehystä valittiin testien alustaksi koska se on sisällytetty Unityyn ja on näin helposti integroitavissa osaksi Unityn työnkulkua. Testit oli helppo suorittaa sekä Unityn editorissa, että osana jatkuvaa integraatiota. Toteutettu testausjärjestelmä alustaa pelaajan mallin annetusta staattisesta ja muuttuvasta pelin ja pelaajan tilasta. Koska tilan tallentamiseen käytetään JSON tiedostoformaattia, joka on suhteellisen hidas deserialisoida, paras tapa alustaa pelin malli oli deserialisoida data yksittäiselle testille ajonaikana. Tällä tavoin hidas datan deserialisatio tapahtuu vain kehityksen alla oleville testeille, eikä vie kehittäjän aikaa deserialisoimalla kaikkia testitilanteita.

Luotu testausjärjestelmä esiteltiin yrityksen ohjelmoijille. Tulevaisuudessa tullaan myös pitämään työpaja, jossa käydään tarkemmin läpi, kuinka luotua testausjärjestelmää käytetään.

CONTENTS

SYMBOLS

EXECUTION BASED TESTING	Testing activities where tests cases are generated and executed by a machine instead of a person, such as unit testing.
SERIALIZING / DESERIALIZING	Translating data to and from a format that can be stored and later reconstructed in the same or different environment.

1 INTRODUCTION

Software testing is increasing its importance in software development projects all the time. Detection of defects is substantially cheaper the earlier they are detected in the development cycle. Automated testing aims to discover defects as early as possible during the development. While using automated testing is common in developing business software, it is still relatively rare in the game industry. Automated testing is usually disregarded because games are too complex to test and require user input and skill which are hard to automate.

The goal of this thesis is to increase automated testing at Next Games by introducing an easy to use framework that makes it easy for developers to write relevant tests that are also run automatically.

The chapter 2 and 3 of this thesis is an overview of common software testing practices which give background for designing the test framework.

The empirical part of this thesis consists of designing, developing and deploying a model testing framework for Next Games. The empirical part is described in the chapter 4 of this thesis

2 SOFTWARE TESTING

This chapter focuses on software testing and the definitions related to software testing. This part does not cover the whole topic of software testing but rather focuses on parts that will be relevant during the empirical part of creating the model testing framework for Next Games.

2.1 Basic Definitions

This section describes common terms related to software testing. These terms are used throughout this thesis. Ilene Burnstein gives great definitions for the terms in the book “Practical Software Testing”. While other sources have both similar and different approaches to these definitions, Burnstein describes the terms in a very thorough and precise matter [1.]

2.1.1 Error

Errors are mistakes made by a developer such as a programmer. Causes for errors that programmers make can be divided into 5 categories: Education, Communication, Oversight, Transcription and Process. When an error occurs due to **education**, the programmer does not have the required knowledge to create a piece of code. An example of this could be a misconception about the precedence order of operators in a specific programming language [1.]

Communication, or more often the lack of it, can also cause errors. For example, two programmers might have agreed on an implementation of an interface but misunderstood which side of the interface is responsible for error checking. Then neither implements the error checking [1.]

Oversight causes errors when the programmer omits something. He might forget to initialize a variable or did not execute all the necessary method calls [1.]

Transcription is a type of error when the programmer knows how to implement code but makes a mistake when implementing. A simple example would be a misspelling of a variable name [1.]

Process introduces errors when the programmer does not have sufficient time to implement systems and must work in a hurry [1.]

2.1.2 Defect

Defect is an anomaly in the software that may cause incorrect behavior. Defects, commonly known as bugs, are result from errors. Defects may also be found on the designs or requirements of software. Developers are responsible for locating and fixing the defects in the software [1.]

2.1.3 Failure

Failure means that the software is not working according to its specifications or performance requirements. A failure can be, for example, unexpected output values, displaying wrong information to the user or an incorrect response from the device. Failure is caused by a defect in the software. Failures are observed by testers during development and by users when software has been deployed [1.]

Every defect does not necessarily manifest as a failure. Software can operate even for long periods of time with a defect without a failure. For the failure to occur the software must first cause a faulty statement to be executed. The statement must then produce a different result than the correct statement which causes an incorrect internal state for the software. The incorrect internal state must then propagate to output as failure. Only then the defect is observed as a failure in the behavior of the software [1.]

2.1.4 Test Case

Test case is an item that describes a set of test inputs, the execution conditions and expected output results. These are the minimum requirements for the test case but organizations may choose to include additional information in their test cases. The inputs are data items that are received from an external source such as hardware, software or human. The execution conditions define the conditions required to run a test. These can include, for example, a certain state of a database or the software or configuration of hardware. The expected output results are used by tester to determine if the test did pass or fail [1.]

2.1.5 Software Quality

Software quality can be inspected from two sides. How well the software meets its specifications and how well the software meets the needs and expectations of users or consumers. These sides can be inspected through quality attributes [1.]

The book “Practical Software Testing” by Ilene Burnstein describes the following software quality attributes: [1]

Correctness – the degree to which the system performs its intended function

Reliability – the degree to which the software is expected to perform its required function under stated conditions for stated period of time

Usability – the degree of effort needed to learn, operate, prepare input, and interpret output of the software

Integrity – relates to system’s ability to withstand intentional and accidental attacks

Portability – relates to the ability of the software to be transferred from one environment to other

Maintainability – the effort needed to make changes in the software

Interoperability – the effort needed to link or couple one system to another [1, 2.]

2.2 Continuous Integration

Continuous integration combines the making of the automated build and the integration process to run continuously. The process consists of build scripts, an integration server and build agents. The build scripts are scripts that execute the necessary chain of commands to produce a build from the source code [3.]

The integration server is responsible for coordinating the build process while build agents execute the builds. The build agents are responsible for triggering builds, checking out the source files, running the build scripts and finally deploying the results [3.]

3 LEVELS OF TESTING

Testing can be divided to different levels according to the size of the tested piece of software. These levels are unit testing, integration testing and system testing. Each level has different goals and challenges.

3.1 Unit Testing

This part describes what unit testing is and some related best practices. Unit testing is relevant to the empirical part of this thesis because the plan is to utilize a unit testing framework and some unit testing principles when creating the testing framework.

A unit test is a test that invokes a unit of work and checks for a specific result. The unit of work can span from a single method to multiple classes. The result of the unit of work can be separated to three different types [3.]

The first type is a public method returning a value. This is the simplest case to execute. Once test objects are initialized a method is called and the return value asserted against the specified value [3.]

The second is a noticeable change in the state or the behavior of the system after invocation of method. The change can be observed without interrogating the private state of the class. For example, a class can have a public property that is altered by a call to a public method of the class. The objects are initialized, the method called and then the property asserted against the expected result [3.]

The third result is a call to an external class. This can be a different class of your project or an external API that the tester does not have the source code for [3.]

The principal goal for unit testing is to ensure that each unit performs their desired tasks. Unit tests should be designed using both black box and white box testing techniques. These techniques are described in the chapters 3.4 and 3.5. Units should be reviewed and tested, preferably by someone else than the developer of

the unit. It is a good practice to save the test results as a part of the unit's history, therefore defects are easy to trace down to a specific unit when similar defects appear. Additionally, this allows everyone in the organization to analyze the history of the test results [1.]

Often unit testing is performed informally by the developer right after creating the unit. This sort of ad-hoc testing and reviewing may lead to defects not being recorded and do not become part of the history of the unit. To have proper practices in place, enough planning and resources should be allocated to testing of the units. Otherwise the missed defects will probably surface during integration testing, system testing or even during the operation of the unit. The fixing of the defect is always more expensive at these stages [1.]

3.2 Integration Testing

Any test that cannot be run independently, consistently and without external resources is no longer considered to be a unit test but rather an integration test. If, for example, a test uses a real filesystem or real system time it is considered an integration test, since running the same test can yield different results and is therefore no longer consistent [3.]

In integration testing the already tested units are integrated together one by one to exercise them together. Usually interfaces between units are a common place to detecting defects. Units should be integrated one by one to reduce the failure points which makes tracking down defects faster for the developers. Integration tests should be performed on units that have already been reviewed and tested [1.]

In object oriented systems a good way to do integration is the concept of object clusters. These clusters consist of classes that together achieve a small feature or functionality. An example of a class cluster is shown in the Image 1 where 4 classes together produce an output (Out Message) from the inputs (In Message) [1.]

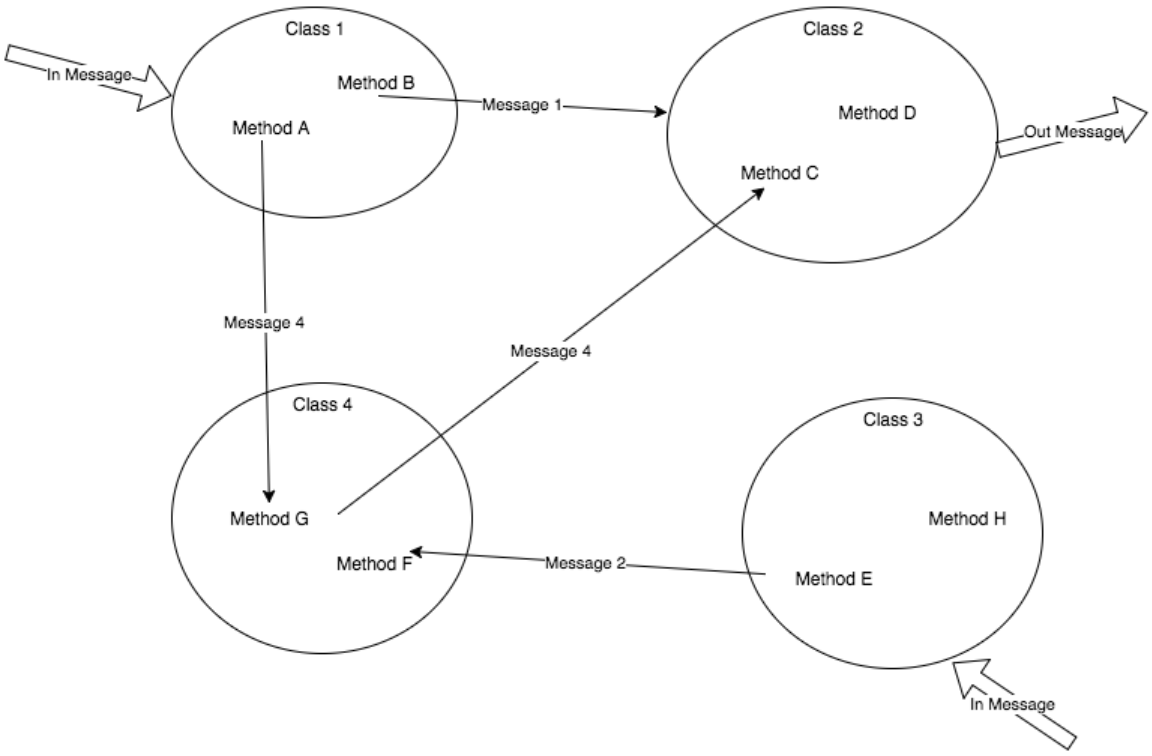

Image 1. Example of class cluster

When integrating classes using the cluster approach, the tester could choose clusters of classes that work together to support a simple function to be integrated first. Then these clusters are integrated with more complicated clusters until the whole system is complete [1.]

3.3 System Testing

When the software has been tested with integration tests, its important subsystems have been tested and the software is ready for system testing. System testing tests the software as a whole. The goal of system testing is to ensure that the software performs according to its requirements. During system testing both the functional and the quality requirements are verified. These include readability, usability, performance and security. System testing is a good chance to reveal external hardware and software interface defects, for example, softlocks, hardlocks and ineffective memory usage. System testing includes several types of tests, most of which have been tested at lower levels [1.]

3.4 Black Box Testing

In black box testing the tester tests a software component without the knowledge of its inner structure. The tester only knows how the software component behaves. The software component's size can range from a method to a complete software system. The specification of the components behavior is required to conduct black box testing. The specifications can come in many forms, for example, as an Input/Process/Output diagram or a well-defined input and output parameters. The tester determines the success of the test by running the test with the provided input parameters and checking them against specified output parameters. Black box testing is also often referred to as functional testing [1.]

An important part of black box testing is designing test cases in a way that they reveal as many defects as possible. Exhaustive testing with all possible input parameters is not feasible and therefore test inputs with the best chance of revealing defects should be chosen. An input domain is a set where the test inputs are selected from. The input domain can be divided into valid and invalid input domains. For example, if a method accepts integers in a range of 1-100, it would be the valid input domain for that method and its associated test cases. An invalid domain consists of any other input values such as negative integers. There are multiple ways of choosing the test inputs. Three different examples are covered here: random testing, equivalence class partitioning and boundary value analysis [1.]

3.4.1 Random testing

When using random testing, the tester randomly chooses a set of input variables for the test case. If a method, for example, has a valid input domain of integers between 1-100 the tester randomly chooses integers from that range such as, 4, 28 and 42. This is not very structured approach and leaves a few issues. The input set might not be adequate for checking if the method meets its specification. It is also hard to determine if having three inputs is the most efficient use of resources. The chosen inputs might not be the most likely to reveal defects. The test case

should probably also include inputs from the invalid domain. These issues can be addressed by using a more structured way of selecting inputs [1.]

Using random testing saves some time and trouble compared to structured methods. Many testing experts have said that randomly selected test inputs have very little chance of producing effective test cases. One use case where this kind of input selection is useful is randomly selecting inputs for stress testing [1.]

3.4.2 Equivalence Class Partitioning

In equivalence class partitioning the input domain is split to different partitions. These partitions, also called equivalence classes, represent a part of the input domain that is processed equivalently in the software component. Therefore, when one input from a selected class reveals a defect, all other tests based on that class are expected to reveal the same defect. Also, if an input from a selected class does not reveal a defect, other values from the same class are most likely to not reveal any additional defects. When the input domain is partitioned accurately the equivalence class partitioning brings multiple benefits for test case design [1.]

Equivalence class partitioning removes the need for testing all the inputs of the input domain which is not feasible. In addition, the tester is guided to selecting subsets of the input domain that are more likely to detect defects. Using this technique, the test case is able to cover a larger amount of input possibilities with a smaller subset of inputs [1.]

For equivalence class partitioning to be effective the input domain must be carefully divided to classes. The following is a set of guidelines that are useful when defining the classes [1.]

If an input condition for a software component is a value range such as 1-100, choose one class that includes all valid inputs and two classes, one outside of each end of the range. For example, -2, 10 and 103 [1.]

If an input condition is specified as number of values, choose one class for all the valid values, one invalid value below the valid values and one invalid value above

the valid values. For example, if the software component requires the input as a number of players and the valid inputs are 1 and 2, choose one value from the class below the valid numbers such as 0. Choose the second value from the valid inputs, such as 1. Lastly choose one value from the class above the valid inputs, such as 4 [1.]

If an input condition is specified as a set of values, choose one class to cover the valid set and one class with value outside the set. For example, if a gun module can have an ammo type of “energy”, “ballistic” or “missile”, choose one class that covers all the valid inputs. Then choose other class that covers all the other inputs [1.]

If an input condition is specified as a must have condition, choose one class to represent the must have condition and one invalid class that does not include the must have condition [1.]

Additionally, when determining the equivalence classes, it should always be considered if there is a reason to believe that software component handles values inside an equivalence class differently. If this is the case, that class should be partitioned to smaller classes [1.]

Another technique to compliment equivalence class partitioning is the boundary value analysis. Testers often find that many defects occur directly at or right next to an input value boundary. Therefore, it is most efficient to develop tests that focus on both the upper and the lower boundary and values just above and below them. For example, when creating a test for a software component that has an input condition of a range from 1-100, select input values according to the boundaries (1 and 100) and just below and above the boundaries (0 and 101) [1.]

3.5 White Box Testing

In comparison to black box testing, in white box testing the tester knows how the software component works. This approach focuses on the inner structure of the software component. This allows the tester to design tests that exercise the

software component in such ways that they can verify that all the logical and data elements in the software component are functioning properly [1.]

3.5.1 Test Adequacy Criteria

The extent of white box testing is usually defined by an adequacy criterion. This represent the criteria that is required for the software component to be considered adequately tested. Since white box testing focuses on logical and data elements, the test adequacy criteria are specified using these terms. The test adequacy criteria define how much of the logic and data is to be covered by the testing, this is also called the coverage criteria. For example, a software component might have 100% statement coverage criteria. This means that all the statements of the software component need to be tested to achieve the adequacy criteria [1.]

Even though 100% coverage is often required, sometimes full coverage of logic is not achieved. This might be due to multiple reasons. The software component may be very simple, small and not mission critical, in which case it is not economical to test all the statements. There might be a lack of resources, for example, not enough skilled testers or not enough time for testing [1.]

3.5.2 Covering Code Logic

Logic-based white box-based testing allows testers to exercise the specific logical elements and features depending on their mission criticalness and the available resources. The logic elements usually considered for the coverage are control statements, such as loops and branches. Programs can be broken down to three primes: sequential (e.g. assignment statements), decisions (e.g. if/then/else statements) and iterative (e.g. for-loops). Using these primes, programs can be described with control flow graphs. Image 2 is a control flow graph presentation of a simple pseudo-code snippet in Figure 1. The function “positive_sum_of_integers” calculates the sum of all positive integers stored in an integer array “a”. Input parameters are an array of integers “a” and the size of the

array as integer “num_of_entries”. The output value is the result of the calculation “sum” [1.]

```
1. positive_sum_of_integers(a, num_of_entries, sum)
2.    sum = 0
3.    i = 1
4.    while(i <= num_of_entries)
5.          if(a[i]>0)
6.                sum = sum + a[i]
            endif
7.          i = i + 1
      end while
8. end positive_sum_of_integers
```

Figure 1. Pseudo-code snippet with branch and a loop [1.]

The blocks in the Image 2 represent statements and predicates of the code. The numbers correspond to statements in the code of Figure 1. The lines represent the transfer of control with direction which depends on the result of the predicate in the block. Using the control flow graph, it is simple to determine a test case that will satisfy the code logic coverage. Such a test case would contain inputs of the array “a” with members of 1, -45 and 3, and “num_of_entries” with the value 3. The code is a simple case where it is feasible to achieve full logic coverage with only one test case. Because one input was an array it can include necessary cases for both positive and negative numbers satisfying the both paths of the if-statement. Similarly, the iteration of while loop can be satisfied by having multiple members in the array. The code does not include any checking for the validity of the inputs [1.]

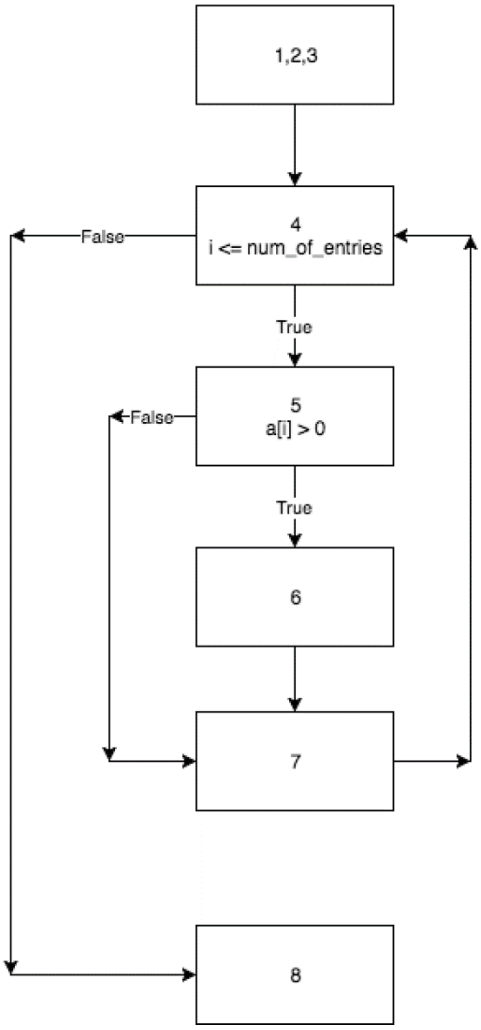

Image 2. Control flow graph describing pseudo code of Figure 1.

3.5.3 Regression Testing

In regression testing old tests are run after new modifications to ensure that all the existing features and functionalities still work according to the specifications. Regression testing can be done at any level of testing. For example, a unit test may pass multiple tests before one reveals a defect. After the unit is repaired, all the tests are run again to ensure that no other tests were broken while fixing the unit. Regression testing is important when new releases are created for the software. Users want the new capabilities while the existing features are also expected to work as usual [1.]

3.6 Model-View-Controller Architecture

Model-View-Controller or MVC for short is a software design pattern that aims to separate the logic that solves the original problem from the user interface. The interface of an application usually changes as time goes on even though the underlying logic may stay relatively similar. As an example, the core of a banking app that used to be run with command line interface would probably be very similar to a banking app that uses a graphical user interface [4.]

The model consists of classes which model the underlying problem. This part of the software will tend to be stable and long-lived as the problem itself [4.]

The views are interfaces to the model. They will consist of a set of classes that provide “windows” to the model, for example, a graphical user interface view or a command line interface view [4.]

The controller is an object that manipulates a view. The controller handles the input from the user and modifies the model. The controller and the view are often combined [4.]

4 CASE: MODEL TESTING FRAMEWORK FOR NEXT GAMES

4.1 Testing at Next Games

This part describes the current testing practices at Next Games. The testing can be roughly divided into three parts: execution based testing on client and server side and QA testing. These three types of testing focus on different areas of the product and vary greatly in quality and execution.

Execution based testing does not have strict rules on the backend side. There is a lot of old code that is not covered by tests. Unit and integration testing are encouraged and all new features should be accompanied by relevant tests. Unit and integration tests are run by continuous integration on a nightly basis. Static analysis is also used to generate reports on test coverage and code styling.

Execution based testing on the client side in The Walking Dead: No Man's Land is limited to a few tests that were written early in the project. These cover some limited functionality checks and include some safeguards, for example, assets and commands. Writing tests is optional and done extremely rarely. Tests are run by continuous integration during the build.

One way of improving the state of the client side execution based testing is to introduce the model testing framework that would enable writing model related tests effortlessly.

4.2 Testing with the Framework

The model testing framework aims to provide tools for writing tests. Since the model always requires external data, the GED and the Player Model, tests are not considered to be unit tests but rather system tests. The testing approach can include parts of both black and white box testing as the programmer writing the test will have the knowledge of the parts that they have created but not for the whole system.

There are some limitations with the testing framework. The tests are always run offline, which means that some features, such as groups and player versus player combat cannot be tested. While loading of missions is possible it would be hard to imitate player actions with commands due to the complexity of the combat system.

When programmers write tests for their new features, especially for the model commands, the chance of oversight and transcription errors is high. While writing the tests, the programmer goes through their code and the expected output which may reveal some of these errors. When executing model commands against the model during testing it is also easier to spot potential security issues which can be exploited with a modified client sending commands in an unusual way, or through ways that are usually blocked by the user interface.

4.3 The Walking Dead: No Man's Land

The Walking Dead: No Man's Land (TWDNML) is the official mobile game based on AMC's TV series. The title was launched on October 8th 2015 and has since been the number one free app in over 10 countries on the App Store [5]. In the game the player builds a camp where they train and upgrade survivors and their equipment before engaging walkers and human enemies in turn based combat.

4.4 Next Games Architecture

Current games, including TWDNML, at Next Games use a MVC like architecture. The state of the player and the game are stored in a model that is identical on both the server and the client. This creates a system that is hard to abuse. The client executes commands which modify the model. The model is allowed to modify itself and therefore there is no separate controller. The commands are first executed against the model on the client side and when the execution succeeds the command is sent over to the server which also executes the same command. When the player loads the game the model is always loaded from the server.

The data of the games consist of dynamic data, which changes according to player actions and the static data that is same for all players. The player model includes all the persistent player data. This includes for example, the currencies, buildings, survivors, completed missions and the combat state of the player. The player model is the dynamic data of the game. The game economy data (GED) stores all the static data related to the game. The GED includes the data for missions and all the values used for the game, such as upgrade costs, drop chances and events. The cached GED on the device is checked against the server version and updated if a new one is available when the game starts.

4.5 Tools

Unity has integrated NUnit test framework to the editor. This allows developers to write tests using the NUnit framework and run tests both inside the editor as well as through the command line. While NUnit is designed to run unit tests, it is also suitable for running other tests and offers versatile tools for creating different test cases [6.]

Next Games uses Bamboo as the continuous integration solution to create and deploy builds. Bamboo is a commercial continuous integration solution developed by Atlassian. The development builds are triggered when a change is detected in the source control. When the build finishes, it is deployed to HockeyApp for QA. When creating the builds, the Unity editor test runner is triggered to run all the tests.

4.6 Goal

The goal of the model testing framework is to enable developers to easily write tests against specific situations. This will be helpful when implementing new features, checking that the existing functionality is preserved intact, and creating test cases for bugs and using those tests to verify the fixes. The same functionality is also useful for testing the migration of players.

These tests will be implemented using a unit testing framework allowing anyone to execute and observe the results and enabling the tests to be effortlessly run during continuous integration.

4.7 Design

The model testing framework provides helper methods and classes that do the heavy lifting of loading a model from given player data. Therefore, developers can focus on the acting and asserting parts of testing. The test framework needs to take care of the following tasks:

1. Load the required GED and the player model
2. Execute the specified command or commands against the model
3. Assert all the required changes
4. Repeat the steps 1-3 for all necessary combinations of GEDs and player models

4.7.1 Creating the Foundation

The first task for creating the model testing framework was to create the helper methods for loading the player model and the GED from json files to the model manager. Most of this functionality already existed, but some refactoring was required to create suitable methods that would accept suitable arguments regarding the testing process. The goal was to use the same methods that the game uses when running to achieve the exact same behavior as running the game would have. Initializing the model manager includes the following steps:

1. Create a new model manager
2. Set the debugger for the model manager
3. Deserialize the GED

4. Load the GED into the model manager
5. Deserialize the player model
6. Load the player model into the model manager

The initialization is a relatively slow process, since it requires deserialization of the json files. When this is multiplied for multiple GED and player model combinations the time for running the test can grow rapidly. Performance needs to be considered from the usability point of view. When running tests on the CI, for example, on a nightly basis the execution time of the test is not greatly relevant, but when running the test manually on the development machine the performance should be adequate to not to slow down the workflow. The tests are loaded when the Unity editor test runner is opened.

After the model manager was initialized with correct data, a test could be written. The writing of tests uses basic test functionality of the NUnit. The next part was to generate test cases so that the same test could be easily ran for different player model and GED combinations. Each test requires a new model manager to be initialized so that any changes to the state of the model from previous tests are not included and the test is run on a clean slate.

An abstract class was created to handle the loading of the model and the GED. All the test classes inherit its functionality, providing them a simple way to execute tests on models. This class triggers the test case generation and takes care of initializing the model managers.

4.7.2 Test Case Generation

The NUnit provides a “TestCaseSource” attribute that can be used to generate multiple tests with the same functionality but with different arguments. The argument takes, for example, a method name that returns a list of test cases. When the tests are loaded (not executed) the method is called and the tests are generated. The first solution was to use the “TestCaseSource” method that initialized the model manager for each test. While the tests will run very quickly,

the loading time of the tests was significant and it would have hindered the workflow, as even running a single test would mean multiple minutes of waiting for loading tests that are not interesting at that moment.

The setup was changed so that instead of using the “TestCaseSource” attribute, each test loops through, and initializes, the model managers with all the combinations of player models and the GED. This removes the load time of tests since no additional tests are generated. The running time of tests increases, which is acceptable, but this makes it difficult to run tests with a specific player model and GED combination and instead forces running the test through all the data.

To combine the best of both previous methods, a wrapper class for the model manager was created. This class stores the model manager, file paths of source data and other information related to data such as the name of the GED and the player. This class initializes the model manager when required. Test cases are generated with the “TestCaseSource” attribute but the method returns list of the wrapper classes. When tests are loaded, only the file paths are saved. Therefore, the load time is fast. The test data combinations show up as single tests which can be run separately or in a group. Because deserializing the GED is the longest part in model manager initialization, and the GED usually remains same for a set of tests, GED caching was implemented to the wrapper class. Therefore, a new GED is deserialized only if it changes from the previous one.

4.7.3 Defining Data Sources for Tests

To allow programmers to easily and explicitly define which data to use with each test, a “TestCaseData” attribute was created. This attribute takes two string parameters (gedSource and playerSource) which define where the test case generation should look for the data. There are four different possibilities for each parameter: a specific file, a specific folder, the test fixture folder and the generic folder. The test cases are generated for each file in the target folder.

The “specific file” option uses only the given file and the “specific folder” option uses all the files in the specified folder. This is useful when there is a single incident

or a small group of incidents for the test in question. Examples include running the test for a player or a group of players with a very specific bug. This allows the test to be very specific and it can except certain things from the test data.

When the parameter is not specified or an empty string is passed, the test generation uses all files in the folder of the test case which is same for all the tests inside the test fixture. This is useful when creating tests, for example, for a new feature and all the tests use player data that has somewhat consistent specifics, such as a certain player level. These tests should be more general in nature and should not expect very specific things from the data.

If the “generic folder” option is used, the test data is read from a folder that all generic tests use. These tests should expect absolutely nothing from the data. This is useful when writing tests that should be possible with any player state such as load and migration tests.

4.8 User Interface

For testing to become a habit in the game teams, the tools need to be extremely easy to use and minimize the repetitive work related to test creation. To achieve this a Unity editor window was made for the model test framework, which allowed to automate the repetitive part of creating new tests and allowing programmers to dive straight into writing tests. The user interface is visible in Image 3.

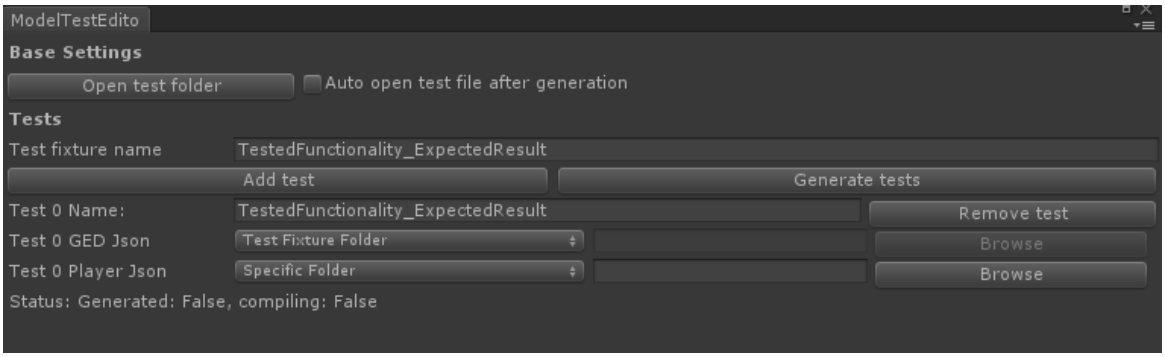

Image 3. The Unity UI for generating model test templates.

The UI includes four main parts: name of the test fixture, name of the test, source for the GED and the source for the player model. Dropdown menus are provided

for test source data, with additional text input for certain cases. One additional source for the GED is to use the offline GED. This is the version of the GED that is included in the build and is used when testing the game offline in the editor. When this is selected, the current GED is copied and used in the similar way as using a specific file as the test case source.

4.9 Running the Tests

In addition to using tests during the feature development to ensure that commands do what they are supposed to, they are meant to be run in the CI when the builds are created. The Unity editor test runner was already used to run tests during the build process, but the details of failed tests were very limited inside the Bamboo interface. The editor test runner produces an XML file that describes test results in more detail, but this data is not fully utilized by the Bamboo NUnit parser.

The solution for this was to create a simple XSL file to transform the XML file into a readable format. A python script is used to transform the XML file with the XSL to create a HTML document which is provided as an artifact for the build.

4.10 Deploying the Framework

The first step of deploying the framework was introducing the model testing framework to other programmers of the company. The framework was presented at a monthly companywide programmer breakfast. The basic functionality of the framework was introduced and an example use cases were showed in form of example tests and a demo presenting the framework in action. A follow up workshop with a more guided hands-on demonstration of the framework will be organized in the near future.

5 CONCLUSIONS

The goal of this thesis was to increase automated testing at Next Games, and eventually improve product quality, by implementing a model testing framework that would encourage automated testing. It is difficult to measure the effects of the framework on the product quality, especially since the framework is not yet widely adopted.

The framework itself fulfills the goals set for it. The creation of test is simple and choosing the data to run the test on is intuitive. The framework is efficient, flexible and fast. It answers to the specific need of defining a set of data (player model and GED) and running tests for that data during the continuous integration. The writing of the tests is simple and straightforward using the provided Unity interface.

There will be further attempt to increase the usage of the framework with workshops and additional education, but ultimately software testing requires a different culture than the current one at Next Games. While any culture shift is never easy or simple, providing the proper tools, in this case the model testing framework, is an important step in the direction adopting a new testing strategy.

When creating this thesis and the associated testing framework, I learned a great deal about methods of software testing and the quality improvements it brings. I believe that the game industry should try to adopt testing in larger scale as it would bring substantial benefits to the product quality which is extremely important in the highly competitive mobile marketspace.

SOURCES

1 Burnstein Ilene. Practical Software Testing. New York, NY, USA: Springer-Verlag New York, Inc.; 2003.

2 Karner Cem, James Bach, Pettichord Bret. Lessons Learned in Software Testing. New York: John Wiley & Sons.; 2002.

3 Osherove Roy. The Art of Unit Testing with examples in C#. Shelter Island, NY, USA: Manning Publications Co.; 2014.

4 John Deacon. Model-View-Controller (MVC) Architecture available at: http://www.battersea-locksmith.co.uk/briefings/MVC.pdf. Retrieved on 1/10/2016. 2015.

5 Saara Bergström. Next Games and AMC's The Walking Dead: No Man's Land Tops App Store Charts Around the World on Its Opening Weekend available at: http://www.nextgames.com/press-release/next-games-and-amcs-the-walking-dead-no-mans-land-tops-app-store-charts-around-the-world-on-its-opening-weekend/. Retrieved on 1/10/2016. 2015.

6 Unity Technologies. Editor Test Runner at: https://docs.unity3d.com/Manual/testing-editortestsrunner.html. Retrieved on 1/10/2016. 2015.