Check-In Testing Framework for iOS

Victor Andreoni, Alex Chen, Jason Whitehouse

December 19, 2014

A Major Qualifying Project Report:

submitted to the
Faculty of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Bachelor of Science

by

Victor Andreoni

Alex Chen

Jason Whitehouse

Date: December 2014

Approved:

Professor David Finkel, Advisor

Abstract

This project was conducted with Microsoft in Cambridge, Massachusetts. The purpose was to design and implement a system for check-in testing of iOS applications. We implemented a framework for managing and creating tests that are portable across iOS devices and versions. It also provides an interface for quickly running all tests locally and reporting results in an easy-to-read format. Running these tests allows developers to ensure that they are not introducing defects into the project before they integrate their changes.

Acknowledgements

We would like to thank Microsoft for sponsoring this project and supporting us through its completion. We would especially like to thank Aseem Kohli, Kris Reierson, and Sandy Gotlib for working closely with us and making this project possible. We would also like to thank Avery Yen for his help in defining our project's requirements, and Hugh Hallawell for finding all of the defects that we missed. Finally, we would like to thank Professor David Finkel for his guidance throughout the project.

Table of Contents

1 Introduction

Testing is one of the most important concepts in modern software development since it is one of the best ways to ensure that a functional product has been produced from the collective efforts of a development team. Equally as important is the method of integrating testing into the development process, which requires balancing the overhead of evaluating the correctness of code with the speed and agility needed to implement features in an acceptable timeframe. These concepts are especially significant large teams such as those at Microsoft, as the potential for error or inefficiency rises quickly with the expansion of product scope and team size. This translates into a need for testing tools that are specifically tailored to the functional needs and development practices of a team.

In the case of iOS development, there is no integrated process for thorough testing of an application. Although there exist a number of tools that are valuable for testing, such as Xcode, Instruments, and the iOS Simulator, there is no common interface to use all of these tools easily in tandem. Each of these resources has its own workflow for usage, set of input parameters, and method of test output, making the process for running all of them time-consuming and difficult. The closed nature of the iOS platform also limits what can be automated through iOS tools, making it necessary to implement a custom approach to orchestrating some kinds of tests.

This paper will describe our efforts to create an efficient local test system for the Intune iOS team designed to abstract the complexities of iOS testing into something that seamlessly integrates with their development process. We will first introduce the ideas and tools that were essential to developing this project. Then, we will discuss the various approaches we took to accomplishing our goal, as well as the drawbacks each of them had and what we learned from them. Finally, we will describe the steps we took to fully integrate our system with our sponsor team.

2 Background

Before discussing the specifics of our project, it is essential to introduce the concepts, tools, and context upon which it is based. We will discuss this background information in the following sections.

2.1 Project Goal

The goal of this project was to create a check-in tester for the Microsoft Intune team that would allow developers to easily validate and test their code before submitting their changes. The tester needed to be able to run from the command line, and have minimal or no configuration needed. The core functionality of this system needed to include capabilities for unit testing and component testing, as well as support anything else needed to accomplish these tasks. Additionally, the tester needed to sync and build the Intune projects prior to running any tests to ensure that the codebase was up-to-date and still able to be compiled.

We were also given a stretch goal of setting up and implementing a continuous integration server. This server had to run our tester on a pre-defined schedule, and would run more in-depth tests that developers would not necessarily need to run every time they made changes to the code. In addition, it was required to have the server be able to report the individuals that made changes to the project since the last run, archive each run, and send out email reports whenever a run failed.

2.2 Regression Testing

One issue in testing software is that, once a feature has been verified as complete or a bug has been fixed, it is entirely possible for that functionality to cease working at any point in the future. This is because of the unavoidable fact that a change to one part of a system may unintentionally change another part, even if there is no apparent connection between them (Savenkov). Regression Testing is an approach to solve this issue by uncovering new issues, or "regressions", introduced into existing functionality by new changes (Myers).

The most common approach to regression testing is to run all existing tests against the entire system whenever a change is made, usually with an automated testing system (daViega). If new tests are introduced along with functionality and bug fixes, running the existing body of tests will be able to verify their continuing correctness. Since it may not always be practical to run every test whenever a change is made, some systems will instead automatically re-run tests nightly or weekly and report failures (daViega). This may be combined with less complete regression testing along with individual changes to catch likely errors more quickly (Dustin, Rashka and Paul).

While existing methods for regression testing may not be able to catch every issue introduced with a change in the codebase, it is still an effective way to maintain the reliability of a system over the course of development. This is a cornerstone of our project, which will provide a system that our sponsors can use to more completely regression test their work. We will discuss the specific needs of Microsoft with respect to testing in Section 2.5.

2.3 Continuous Integration

As a final stretch goal, part of our project involves the implementation of continuous integration with our sponsor's build system. Continuous integration is a practice that involves building and applying quality assurance processes on a project in order to detect defects as quickly as possible (Meyer). Defects not only refer to flaws in the code logic, but also include errors in naming conventions, documentation, and software design (Berg). Catching and recognizing these defects early on in a project gives developers rapid feedback on the state of their project, improves code quality, and eliminates many of the risks of trying to integrate large portions of code at the end of a project.

One of the core practices of continuous integration is for developers to commit and integrate their code changes regularly (Meyer). This allows for the changes that a developer makes to more easily combine with the rest of the code base. Regular commits create a more continuous development process that will ultimately result in improved code quality. In addition, integrating code in small batches makes it easier to debug sudden failures in the code. Developers can easily pinpoint exactly which change or added lines of code could have caused a failure, as opposed to trying to track down a defect in the large overall project.

The key tool to continuous integration, and an important tool in software development in general, is the continuous integration server. The purpose of a continuous integration server is to check out all the changed committed code and run commands to build the project. In addition, tests might be run on the new build to ensure that the code still works as expected. There are a number of different continuous integration servers available, but a common one for large-scale deployment is Jenkins (Duvall, Matyas and Glover), which we will discuss more in Section 2.6.7.

2.4 Intune

Our sponsor team in Cambridge is part of Microsoft Intune, which, according to its web site, "helps organizations provide their employees with access to corporate applications, data, and resources from virtually anywhere on almost any device, while helping to keep corporate information secure." (Microsoft Corporation). Intune exists across several platforms and services, but our sponsors specifically work on application restrictions for iOS; the ability for apps to control user activity and data protection based on IT policies. An example of this is that a "managed" application might require a passcode to open, or it might prevent users from copying sensitive text from it into another, non-managed application.

At the time of our project, the most important task of our sponsors was creating an App Restrictions SDK that allowed application developers to integrate Intune policy handling into their own work. In addition, they were working on an 'AppWrapper' that took an existing iOS application and added policy handling into it (Kohli). The complex, enterprise nature of this project would be a driving force for many of our design decisions since it meant that we needed to provide the capability for testing scenarios involving multiple applications and device states.

2.5 Microsoft Strategy

Until recently, there were two primary kinds of software engineers in Microsoft, composing a ‘Distributed Engineering’ model: Software Development Engineer (SDE) and Software Development Engineer in Test (SDET). It was the responsibility of the SDE to write software that fit the requirements of the product, and the responsibility of the SDET to ensure that the software was fully functional and up to the standards of release. Testing for an SDET meant understanding how to fully evaluate a system, and writing automated test cases and testing frameworks to do so (Eliot).

Some parts of Microsoft have recently begun shifting towards a more agile approach to software development, where requirements are iterated on throughout the development process and releases are made more frequently. As part of this shift, testing was to be made a more integral part of the development process. This proved to be somewhat incompatible with distributed engineering since every agile shift meant that both the SDE and SDET would need to adjust and the separation of the two made testing a parallel process to development rather than an integral part of it (Kohli).

In order to address this several teams, including the Intune team, have moved to a “Unified Engineering” strategy in which the SDET role was removed entirely. In their absence, it would be the responsibility of every SDE to test their own code as it was written, both for its own functionality and how it contributed to the stability of the product as a whole (Kohli). This meant that SDEs would need a framework allowing them to easily write tests for their features, as well as run all tests to ensure the integrity of the product – a solution that our project would provide for our sponsors.

2.6 Tools

In this section, we describe the tools that we used in developing our testing framework. Most of the tools were used from the command line, since our system was primarily able to trigger them with shell commands.

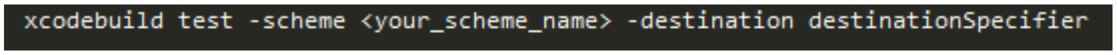
2.6.1 Python

Python is the language that we used to write our framework, and that developers would use to write tests for it. It was chosen because our framework needed to easily run on Mac computers, and Python comes pre-installed with Mac OS X (Using Python on a Macintosh). Python also supports different programming paradigms, including object-oriented programming, which allowed us to explore a number of different approaches to our design. Additionally, Python was picked over other options such as Bash and Perl because it includes extensive libraries (About Python) that we would otherwise have to recreate ourselves.

2.6.2 Xcode

Xcode is an integrated development environment (IDE) created by Apple for OS X and iOS development. It contains a number of software development tools that were useful and necessary for the successful implementation of our framework. Among these tools is the iOS Simulator, which allows developers to perform multiple tasks such as running, debugging, and testing applications without the need to have a physical iOS device present.

Another part of Xcode, and perhaps more important than the Xcode IDE, is the Xcode Command Line Tools Package. This package allows for command line development in OS X using the "xcodebuild" command. Using xcodebuild allows developers to compile, build, and unit test Xcode projects and workspaces from the command line. It also allows users to specify a specific device or target to operate on, as well as a specific scheme in the project to build (Building from the Command Line with Xcode FAQ). It also allows developers to unit test their project from the command line by specifying a build action of "test" when calling xcodebuild. Unit testing this way requires a project scheme to build and test, as well as a destination for the tests. Figure 1 below demonstrates the signature of the xcodebuild command when running unit tests.



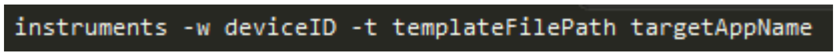
```
xcodebuild test -scheme <your_scheme_name> -destination destinationSpecifier
```

Figure 1: Running Unit Tests using xcodebuild

2.6.3 Instruments

Another developer tool included with Xcode is Instruments, which provides ways to track various types of information such as file access and memory usage from OS X and iOS code (Instruments Quick Start). For our project, the Automation tool within Instruments was key in being able to write component tests. The Automation instrument allows developers to automate user interface tests that simulate interaction by calling a JavaScript UI Automation API (Automating UI Testing). This tool also provides a mechanism for easing the process of creating JavaScript files for the automation by allowing developers to record actions on a targeted device, and having the UI Automation calls be auto-generated into a script that capture the performed actions.

Similar to Xcode, Instruments also supports running through the command line using the "instruments" command. When using this command, it is necessary to provide a unique device identifier, a template file, and the target application name. Figure 2 shows how this information should be passed into the instruments command in order to utilize it.



```
instruments -w deviceID -t templateFilePath targetAppName
```

Figure 2: Running the Instruments Command

If we wanted to run the Automation instrument using this command, the templateFilePath parameter would be the path to an automation instrument template that is generated using the Instruments UI tool. Other tools within Instruments can also be run using the above command by simply passing in their appropriate template as well.

2.6.4 iOS-deploy

iOS-deploy is an open-source project hosted on GitHub that allows for the installing, debugging, and uninstalling of iOS applications to a physical iOS device through the command line without the use of Xcode (Abdullah). Unfortunately, neither Xcode nor its command line tools provide a convenient mechanism for just installing an application to a physical device. There were possible ways of accomplishing this task by just using Xcode, but they did not provide the functionality that we desired. For example, when running unit tests with the command “xcodebuild test” as described in the Xcode section, the application being tested would be installed onto the targeted device if it was not previously there. However, it would then also run all included unit tests, which could end up being a lengthy process for just the desired effect of installing the application. Using iOS-deploy, we would be able to just install or uninstall applications to a device without the side effect of also running all of the unit tests associated with a project. The downside to using this tool is that it is a third party application that it is not maintained by Apple, and therefore is susceptible to becoming incompatible with future iOS versions. However, developers at Microsoft seem to have realized the need for iOS-deploy despite the possible downside, and maintain their own version of iOS-deploy.

2.6.5 Intune Build Systems

The Microsoft Intune projects have their own build script that compiles the source code and produces binaries. This script first sets up the developer's environment by running another script called "razzle" to set a number of environment variables that the Intune project is dependent on. The next step of the build script is to sync project files using the appropriate source control discussed in Section 2.6.6. If all of these steps succeed, it will then proceed with building the projects and then code signing the applications it created. All of these files are placed into a binaries folder in a standard location.

2.6.6 Source Control

There were two different source control systems that were used: SourceDepot and Team Foundation Server (TFS). SourceDepot is an internal Microsoft system that works on the command line, and was used by the Intune team for the first half of the term. Thus, any script that wanted to sync project files had to use SourceDepot commands. We used Git for our own development, and depended on our mentor to update the code in SourceDepot when we wanted to make it available to our sponsors.

In the second half of the project, the Intune project migrated to TFS, which uses Git as its source control system. This simplified the process of making updated versions of our system available to our sponsors, and changed our approach to continuous integration.

Since TFS is now the standard version control system for the Intune team, any future work on this system should not be dependent on SourceDepot.

2.6.7 Jenkins

Jenkins is a Java-based continuous integration server that automates a project's building process and provides feedback on broken builds or failed test cases. It has a simple web interface that is easy to learn, but is also flexible and adaptable with over 1000 plugins available to be installed (Kawaguchi). At the heart of the Jenkins build process are build jobs. In its essence, build jobs are thought of as a particular step in the build process, typically involving compiling source code and running unit tests. These Jenkins jobs can be configured to run on a customizable schedule, such as daily, monthly, or weekly, and can also be configured with source control to always sync the latest changes before building. Additionally, products from building can be archived and build results can be published for developers to see and generate trend reports with. Overall, Jenkins is able to assist with making sure that a project still properly functions after changes are made, without the need for developers to manually start a large project's build sequence and await the results.

3 Fundamental Approaches

The framework we designed needed to be flexible enough for developers to create tests of varying steps and complexity. However, since some of the developers using this framework were not accustomed to writing tests, we also did not want our design to be overly complicated in order to achieve the flexibility we wanted. To achieve this balance of flexibility and ease-of-use, we tried and discussed a number of different designs. In the following sections, we describe the designs that we considered and discuss what we learned from them, which led us to our final flat design of the Python API.

3.1 Single-Script

To introduce ourselves to the tools and commands we would be using, we began by creating a check-in script for a simple To-Do application we implemented while preparing for our project. This Python script was divided into three separate modules that encompassed what we believed to be the core functionality of our framework: syncing with source control, unit testing, and component testing. Although each module could be run individually, a Python wrapper was also created to easily combine the modules into a single script. Finally, for ease of use, a simple bash script was created to wrap the Python script, which allowed users to call the script from a terminal without having to invoke Python.

3.1.1 Design Details

Each of the modules contained a function to perform the corresponding action it represented. The functions took in different parameters based on the information it required in order execute. Since our To-Do application used Git for its source control, our syncing module required the name of the master branch to get the most recent code changes from, as well as the name of the branch the developer was working on in order to know where to merge the changes into. In order to run unit tests, the function needed a project scheme to build and test, the unique device identifier (UDID) of the device the tests should be run on, and the configuration of the project to build. These parameters, along with their associated option flags, were then passed into the "xcodebuild" command with the build option of "test". Finally, the component-testing module required the Xcode project name, the UDID of the device to test on, and the location of the JavaScript file that contained the automation instructions. These were then passed to the "instruments" command to run the test. All three modules also error-checked the output that resulted from running their commands. This was done simply by parsing any text in the standard error stream, and finding if any fatal errors occurred that prevented the command from executing successfully.

The Python wrapper combined the three modules into a single script, and also provided functions for selecting devices, schemes, and configurations to use for testing. This was convenient because if the user did not know the UDID of the device they wanted to test on, the wrapper would prompt with a list of available devices, and they could easily choose the one they wanted.

3.1.2 Final Thoughts

This check-in script was only designed to work with our To-Do application and not the Intune projects, but it provided a lot of insight into the commands that we would be using in later scripts. We were able to learn what commands and what information were necessary in order to unit test and component test. We were also able to discover how to list the available devices, as well as the available schemes and configurations associated with a project. By knowing all of this information and gaining experience in writing these commands in Python, we were able to focus more on the design of our subsequent approaches, rather than trying to figure out what commands we needed.

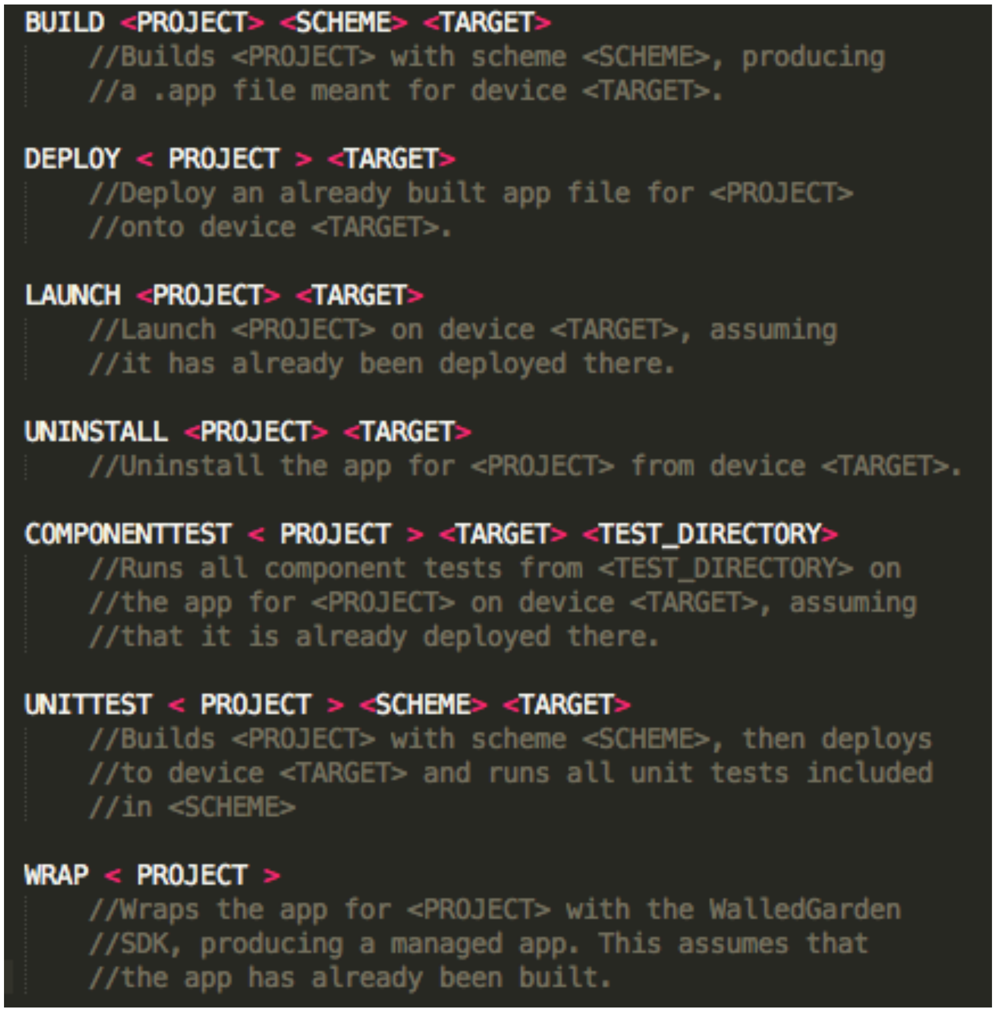
3.2 Build Language

It became clear from discussions with our sponsors that our project would not be limited to component and unit tests, but instead focus on tests involving multiple actions and applications. One example was a test that would install an application to reset all device security settings, and then install a second application that contained a component test to ensure that a passcode was prompted for.

In order to allow for such complex, multi-stage tests, we came to the idea of creating a build language that would allow developers to define any test that was required. This would be made up of several actions, which developers would be able to chain together into test cases. Doing this meant that our system did not have to be programmed to include every potential test, an impossible task for a project as large and involved as Intune, but instead simply provide the basic tools for developers to write their own tests. These tests would still be collected together and run by our command-line interface, preserving its value as a check-in gate.

3.2.1 Design Details

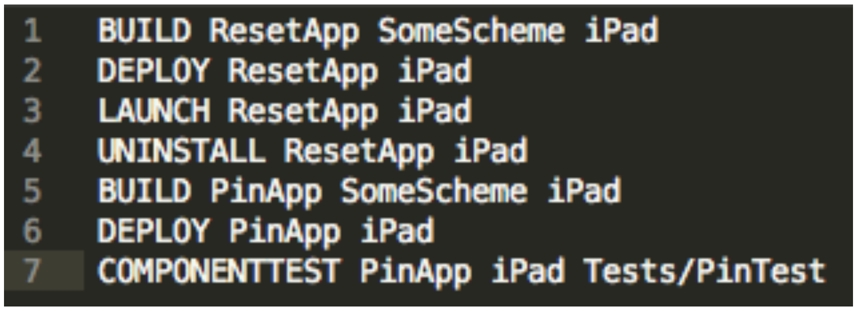
The actions that we originally intended to include are described in Figure 3. These actions build on the functionality of the single-script approach while also providing new features such as launching an application and wrapping an executable with the Intune App Restrictions SDK.



```
BUILD <PROJECT> <SCHEME> <TARGET>
    //Builds <PROJECT> with scheme <SCHEME>, producing
    //a .app file meant for device <TARGET>.

DEPLOY < PROJECT > <TARGET>
    //Deploy an already built app file for <PROJECT>
    //onto device <TARGET>.

LAUNCH <PROJECT> <TARGET>
    //Launch <PROJECT> on device <TARGET>, assuming
    //it has already been deployed there.

UNINSTALL <PROJECT> <TARGET>
    //Uninstall the app for <PROJECT> from device <TARGET>.

COMPONENTTEST < PROJECT > <TARGET> <TEST_DIRECTORY>
    //Runs all component tests from <TEST_DIRECTORY> on
    //the app for <PROJECT> on device <TARGET>, assuming
    //that it is already deployed there.

UNITTEST < PROJECT > <SCHEME> <TARGET>
    //Builds <PROJECT> with scheme <SCHEME>, then deploys
    //to device <TARGET> and runs all unit tests included
    //in <SCHEME>

WRAP < PROJECT >
    //Wraps the app for <PROJECT> with the WalledGarden
    //SDK, producing a managed app. This assumes that
    //the app has already been built.
```

Figure 3: Build Language Actions

Our system would crawl over a collection of these tests stored in some shared location in the Intune file system. Each test would be read in line-by-line, and their commands would be interpreted and executed in order. For example, the test described in the introduction to this section would be encoded as shown in Figure 4.



```
BUILD ResetApp SomeScheme iPad
DEPLOY ResetApp iPad
LAUNCH ResetApp iPad
UNINSTALL ResetApp iPad
BUILD PinApp SomeScheme iPad
DEPLOY PinApp iPad
COMPONENTTEST PinApp iPad Tests/PinTest
```

Figure 4: Pin Test (Build Language)

This test would first build, deploy, and launch the ResetApp to an iPad to reset the device's settings. It would then uninstall the ResetApp, build and deploy the PinApp, and run a component test on the PinApp to check whether a passcode was requested. This case would not have been supported by our single-script design, which demonstrated that our increased flexibility was a step in the right direction.

3.2.2 Design Drawbacks

Before we had made much progress towards implementing this solution, more discussion among ourselves and with our sponsors uncovered a fundamental drawback in this design. Because we expected tests to be encoded solely with our predefined actions, we were providing a language with a very weak syntax. It only executed sequential commands, with no support for non-action logic, assignment, or any control flow. We were providing more flexibility than the single-script approach, but we were still too restrictive to be confident that our system could handle all of Intune's testing needs. Some of this would have been possible to implement, but it would not have made sense to spend significant time implementing basic properties of a strong language when many such languages already exist. It would also make our code base much less maintainable since it would include not only the logic of the actions, but also the interpretation of our language's syntax.

3.2.3 Final Thoughts

This design was valuable because it led us to think about more flexible approaches that would enable our sponsors to define arbitrary test scenarios. It also forced us to define the actions that we needed to provide from our system, which would still be used in future iterations. Therefore, we consider this design to be an important stepping-stone. While it was too restrictive to be valuable on its own, it led to our decision to have test cases written in Python as described in the following section.

3.3 Python API

As discussed throughout this chapter, we considered several approaches for the implementation of our testing framework. Based on design meetings and discussions with members of the Microsoft team, we decided that implementing a testing framework API would allow us to meet the functionality requirements while achieving our goals of providing a flexible and maintainable structure. In the following sections we will discuss the two approaches that we considered for the API implementation.

3.3.1 Object-Oriented Design

The first approach that we considered for the implementation of the Python API was an object-oriented design. One of our main project goals was to implement a framework that was both easy to maintain and easy to extend, and we saw several advantages to achieve these goals in an object-oriented framework. However, after implementing the framework and discussing our design with our mentor, we determined that the lack of support for a strict object-oriented structure in Python made this approach unnecessary. Therefore, we decided to abandon this design in favor of a more linear one, described in Section 3.3.2. In this section, we will describe the object-oriented design that we initially implemented, its advantages and disadvantages, and the reasons that we had for discarding it.

3.3.1.1 Design Details

The object-oriented structure that we designed included the implementation of several design patterns and object-oriented practices. As shown in the UML class diagram in Figure 5, we divided our framework into actions that encapsulated the different features that our framework offered. This separation of the functionality of the framework lent itself to structuring the code of each action in a separate class, as shown in Figure 5. Each action was in charge of parsing and validating its arguments. In addition, each action was responsible for executing the expected behavior and reporting its success or failure. By taking this approach, actions were independent of each other, which made maintaining the framework an easier task.

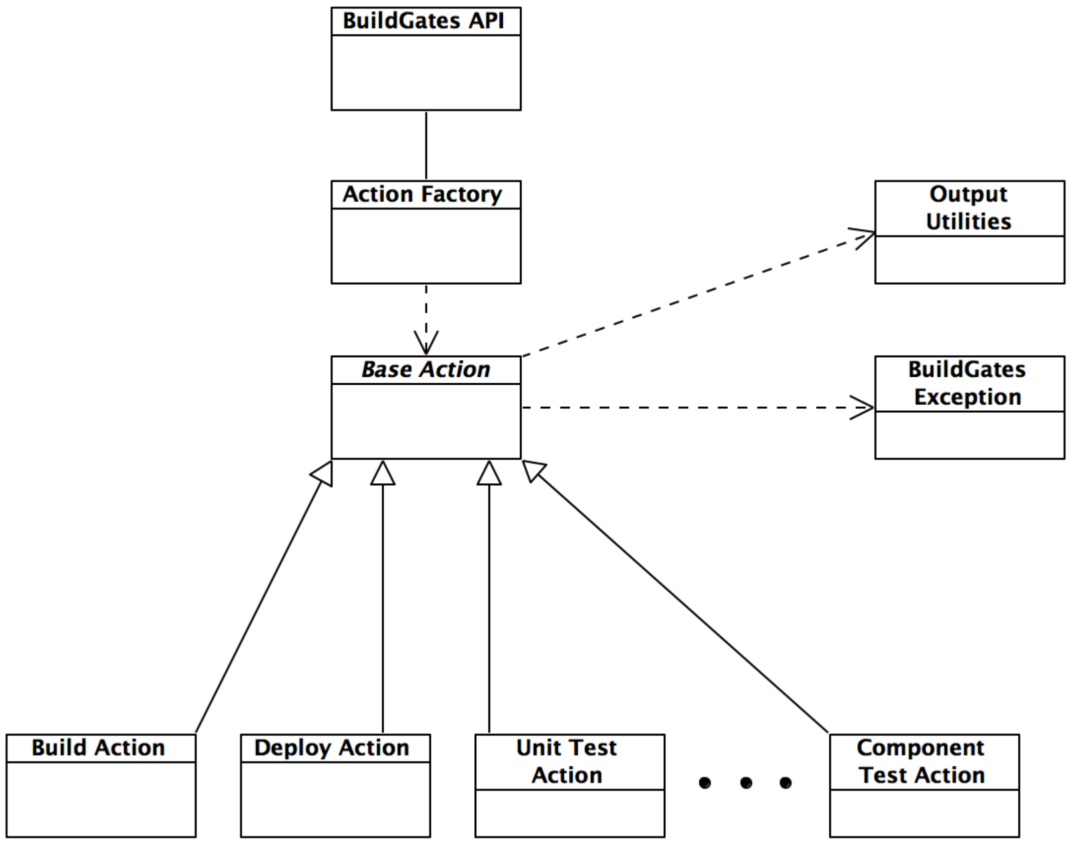
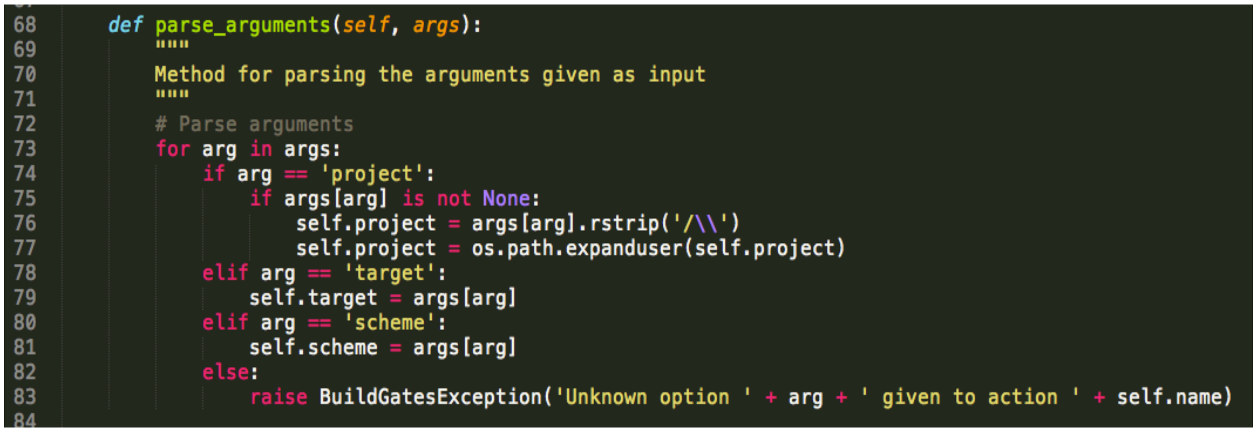


Figure 5: UML Class Diagram of Object-Oriented API Framework

As we continued to implement actions, it quickly became apparent that most actions shared some commonalities that could result in unnecessary code duplication. As a result, as shown in Figure 5, we implemented an abstract class, named "base action", to hold all the common methods and fields. For example, most actions depended on the unique device identifier (UDID) of a device in order to execute. By implementing a method in the base action class to get the UDID of connected devices, every action could reference this function without having to implement it itself. Aside from preventing code duplication, centralizing common functions also increased maintainability since bugs could be tracked and improvements could be implemented in a single place. The base action class also benefited from the implementation of template methods, where the base action would provide basic functionality to common functions that could then be overridden by specific actions. For example, the base action had a basic argument parsing function that would only parse for the most common arguments, shown in Figure 6. By overriding this method, specific actions could parse for additional arguments or prevent the parsing of other arguments that were not needed.



```python
def parse_arguments(self, args):
    """
    Method for parsing the arguments given as input
    """
    # Parse arguments
    for arg in args:
        if arg == 'project':
            if args[arg] is not None:
                self.project = args[arg].rstrip('/\\')
                self.project = os.path.expanduser(self.project)
        elif arg == 'target':
            self.target = args[arg]
        elif arg == 'scheme':
            self.scheme = args[arg]
        else:
            raise BuildGatesException('Unknown option ' + arg + ' given to action ' + self.name)
```

Figure 6: Basic Argument Parsing Function in Base Action Class

In order to connect the different actions of our framework, we implemented an action factory at the center of the design. This factory, as the name suggests, follows the factory pattern, which provided us with a centralized class in which to initialize the different actions that the framework supported. The action factory also provided a level of abstraction between each specific action and the public API interface. As a result, the public API could simply call the factory and ask for an instance of the desired action by specifying the action name with a string, as seen in Figure 7. In addition, the API provided a list of arguments to the factory when invoking the “get instance” method so that the factory could pass them to each action for parsing. In case of an error, the factory would catch an exception from the action constructor and raise it to the public API. In this design, the action factory acted as the principal entry point to the whole framework. This satisfied our original goal of abstraction since the public API could simply ask for an action and pass in an arbitrary list of arguments to the factory. The factory, in conjunction with the action constructors, would take care of input validation and object instantiation.

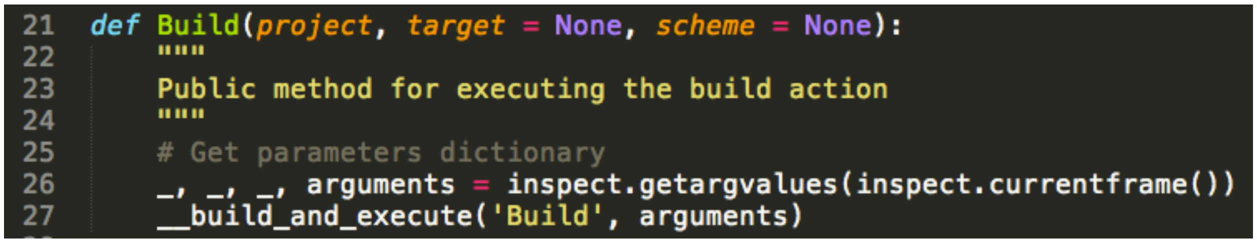
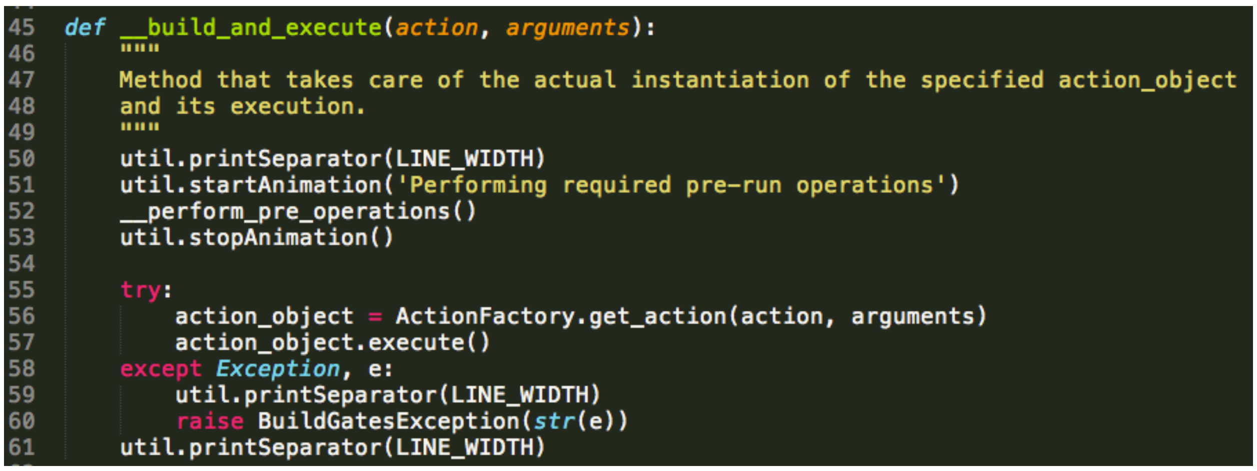

```python
def Build(project, target = None, scheme = None):
    """
    Public method for executing the build action
    """
    # Get parameters dictionary
    _, _, _, arguments = inspect.getargvalues(inspect.currentframe())
    __build_and_execute('Build', arguments)
```

Figure 7: Public API Method Calling Internal Function to Build and Execute the Specified Action

The public API itself consisted of a series of methods that allowed developers to execute the different actions that our framework supported. When calling a method, a list of arguments for a specific action was provided. The API would then construct a dictionary from this list of arguments and pass it along to a private method for internal execution. As shown in Figure 8, this method took care of any required pre-operations, such as checking for the proper setup of environment variables, called the factory to get an instance of the specified action object, and told the object to execute the action. These calls were surrounded in a try-catch block in order to catch any exceptions that might be thrown along the stack of internal calls.



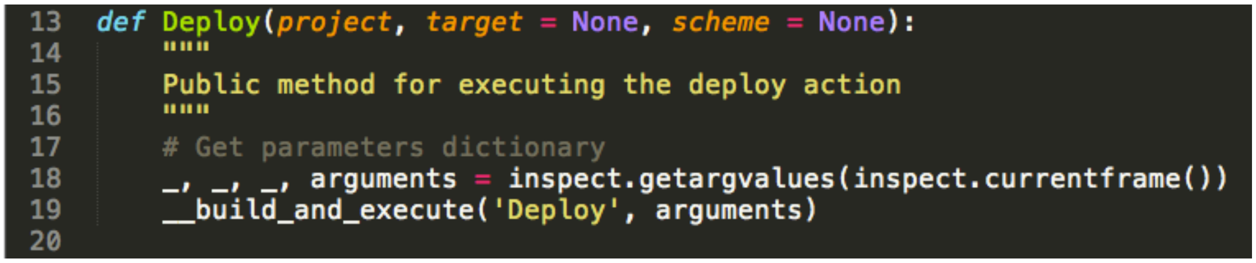
```python
def __build_and_execute(action, arguments):
    """
    Method that takes care of the actual instantiation of the specified action_object
    and its execution.
    """
    util.printSeparator(LINE_WIDTH)
    util.startAnimation('Performing required pre-run operations')
    __perform_pre_operations()
    util.stopAnimation()

    try:
        action_object = ActionFactory.get_action(action, arguments)
        action_object.execute()
    except Exception, e:
        util.printSeparator(LINE_WIDTH)
        raise BuildGatesException(str(e))
    util.printSeparator(LINE_WIDTH)
```

Figure 8: Public API Method to Invoke the Action Factory and Instantiate an Action of the Specified Type

3.3.1.2 Design Drawbacks

Overall, the object-oriented design satisfied many or our initial goals, especially those of maintainability and extensibility. However, there were some flaws to this design that, after discussion between the members of our group and members of the Microsoft team, resulted in us abandoning this idea and taking a different approach.

One of the main arguments against this design arose from the fact that the language we chose to use, Python, does not fit the object-oriented patterns as well as other languages. Our object-oriented design was, although not explicitly, based on the premise of programming against an interface. In our design, there was an inherit action interface that was implemented by the base action and extended by each individual action. Still, since Python does not enforce strong typing, many of the restrictions that justified the patterns we implemented in other languages were not there. For instance, since the factory could return an object of any type, there was no restriction that prevented us from having considerably different constructors. While some actions took in two parameters, others took in three or four. This resulted in the aforementioned obfuscation of parameters where we created a dictionary of arguments that each action had to parse, as shown in Figure 9. Although the public functions of the API did not need to concern with this, the process made the internal code hard to follow.



```python
def Deploy(project, target = None, scheme = None):
    """
    Public method for executing the deploy action
    """
    # Get parameters dictionary
    _, _, _, arguments = inspect.getargvalues(inspect.currentframe())
    __build_and_execute('Deploy', arguments)

```

Figure 9: Example of Parameter Obfuscation on the Deploy Action

This drawback was further emphasized by the implementation of a super constructor in the base action class. In strongly typed languages, such as Java, it is a good practice to have a super constructor perform basic common operations concerning initialization. However, due to the different arguments for each action, we found ourselves in a situation where some actions would have access to some fields that it never referenced during execution. Although we had implemented template methods that could be overridden to prevent this situation, we kept going back to trying to determine which fields were common enough to leave on the base class and which fields should be taken out. This situation continued to arise as we added more actions to the framework and realized that some fields became more relevant and others became less relevant.

In addition to these downfalls, discussions with our mentor and a testing engineer led to the conclusion that the very nature of a loosely typed language made most of the abstraction of our design unnecessary. Even though we were abstracting the public API from handling each individual action, there were no restrictions imposed by Python that would prevent developers from simply initializing a specific action from its class while circumventing the action factory and the other layers we had established. This issue became a more pressing one when we were adding new actions to the framework. Every new action had to implement a new method in the public API and a new class that extended the base action abstract class, and each API method needed to be aware of the signature of its corresponding action. This meant that the “base action” abstract class did not completely uncouple the API from the actions implementation. In addition, the action factory had to be modified in order to add the new action to the list of available actions.

Although these tradeoffs are common in object-oriented design and are usually worth the effort due to the abstraction benefits that they provide, the fact that developers could simply circumvent all of this due to the language that we were using made us question whether the effort was truly worth it. Based on these and other considerations, we decided to abandon the object-oriented design and opt for a more simple, single-layer structure, which will be described in Section 3.3.2.

3.3.1.3 Final Thoughts

The object-oriented framework design was a move in the right direction for our project. First, we addressed most of the issues described in the build language section by allowing developers to write their tests using a full-featured language such as Python. In addition, we started to pay more attention to maintainability and expandability, and set basic requirements that we would carry on to our final design. Finally, we concluded that this API approach would be suitable for our testing framework, and decided that we would stay focused on this approach throughout the rest of our project.

3.3.2 Flat Design

As discussed in Section 3.3.1, we determined that taking an API approach was suitable for the implementation of our testing framework. Therefore, in order to improve upon the drawbacks of our object-oriented design and maintain the API structure, we implemented a flat design that is easier to extend and maintain. In this section, we will describe the details of this design, its advantages and disadvantages, and our reasons for selecting it as the final design for our testing framework.

3.3.2.1 Design Details

The flat design is mostly a simplification of the object-oriented design discussed in Section 3.3.1. As shown in the UML class diagram in Figure 10, we maintained the division of the functionality of our framework into different classes representing actions. However, we removed the layers of abstraction that were the cause of most of the drawbacks of the object-oriented design.

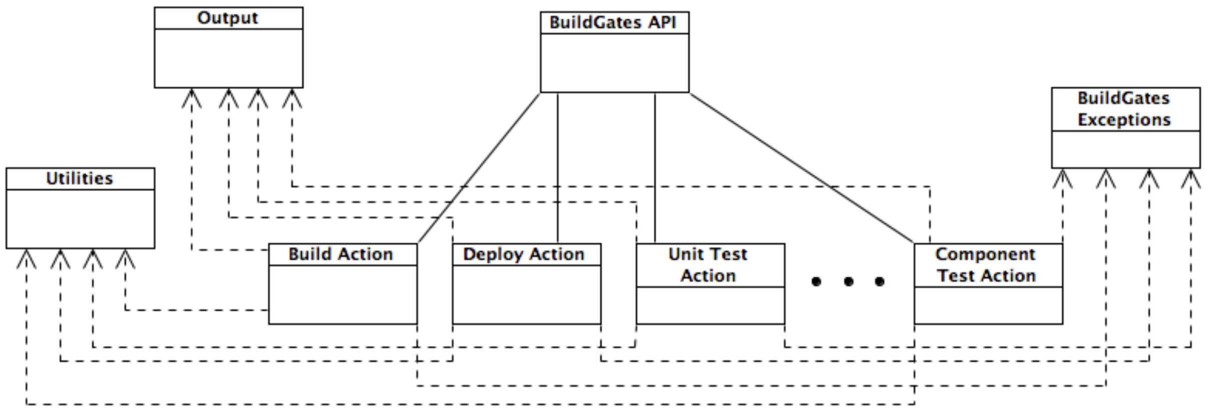
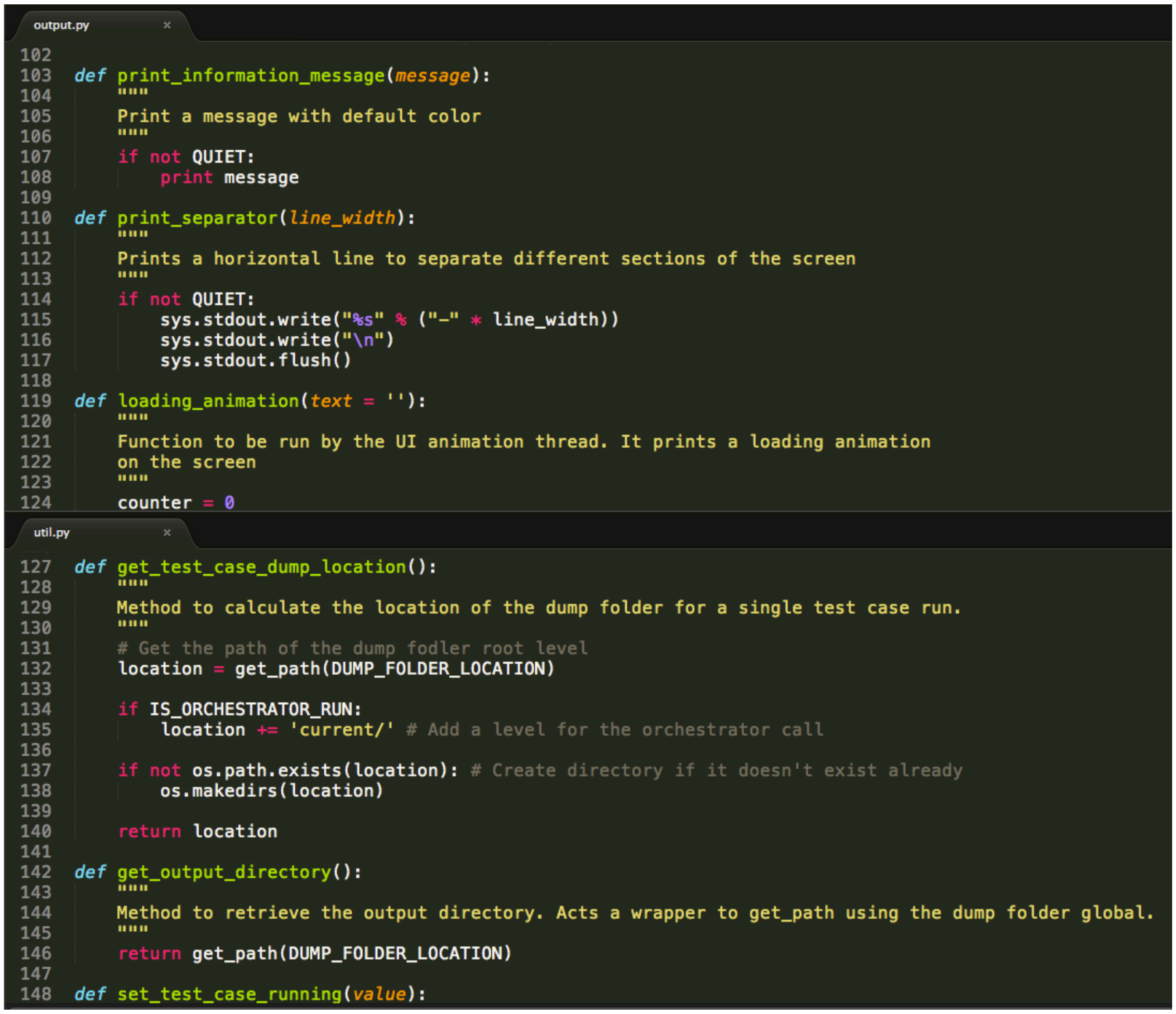


Figure 10: UML Class Diagram of Flat API Framework Design

One of the main changes in this version of the framework was the removal of the base action class. By removing this abstract class, each action became responsible for the parsing and instantiation of every field needed for execution. This shift of responsibility served as a solution to the issue of actions having unused fields, and simplified the addition of new fields. The common methods that were implemented in the abstract class were moved to the utilities class in order to avoid having to implement common functionality in each action and prevent code duplication.

Refactoring the utilities class made us realize that we were combining functional utilities and output utilities in the same file. For example, the utilities class had a method for printing the loading spinner and a method for getting a device UDID. Having functions with such distinct purposes in the same file made it hard to read and debug the utilities class. In addition, we were not achieving the encapsulation that we were aiming to provide in our framework. In order to avoid this discrepancy, we decided to refactor output methods into a new class, which we named output. This way, as shown in Figure 11, all methods concerning output were encapsulated in the output class, and all common methods related to actual framework functionality were encapsulated in the utilities class.



output.py

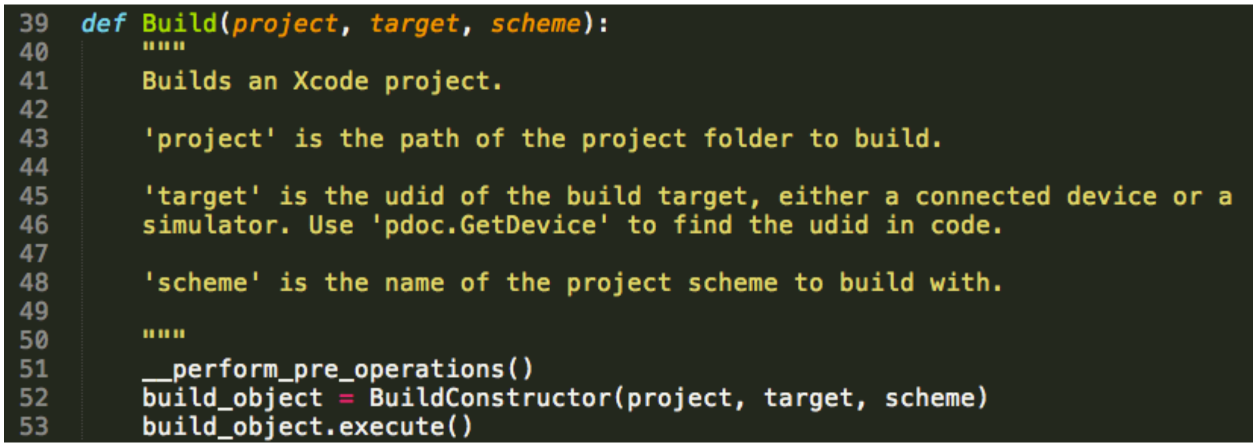
```
def print_information_message(message):
    """
    Print a message with default color
    """
    if not QUIET:
        print message

def print_separator(line_width):
    """
    Prints a horizontal line to separate different sections of the screen
    """
    if not QUIET:
        sys.stdout.write("%s" % ("-" * line_width))
        sys.stdout.write("\n")
        sys.stdout.flush()

def loading_animation(text = ''):
    """
    Function to be run by the UI animation thread. It prints a loading animation
    on the screen
    """
    counter = 0
```

util.py

```
def get_test_case_dump_location():
    """
    Method to calculate the location of the dump folder for a single test case run.
    """
    # Get the path of the dump fodler root level
    location = get_path(DUMP_FOLDER_LOCATION)

    if IS_ORCHESTRATOR_RUN:
        location += 'current/' # Add a level for the orchestrator call

    if not os.path.exists(location): # Create directory if it doesn't exist already
        os.makedirs(location)

    return location

def get_output_directory():
    """
    Method to retrieve the output directory. Acts a wrapper to get_path using the dump folder global.
    """
    return get_path(DUMP_FOLDER_LOCATION)

def set_test_case_running(value):
```

Figure 11: Output Methods Encapsulated in Output File (top) and Common Framework Methods Encapsulated in Utilities File (bottom).

Another major change in the flat design implementation was the removal of the action factory. As discussed in Section 3.3.1, the action factory caused parameter obfuscation between the public API and the framework actions. By removing the factory and linking the API to each individual action, we were able to remove the aforementioned obfuscation, which rendered our code structure easier to follow and understand. In addition, the code of the API functions became much simpler after the removal of the action factory. As show in Figure 12, the public API now initializes each action object by calling its constructor directly, without having to first inspect its arguments. Furthermore, the flat design does not require each API method to call a centralized method and check for exceptions, which improves performance by reducing runtime.



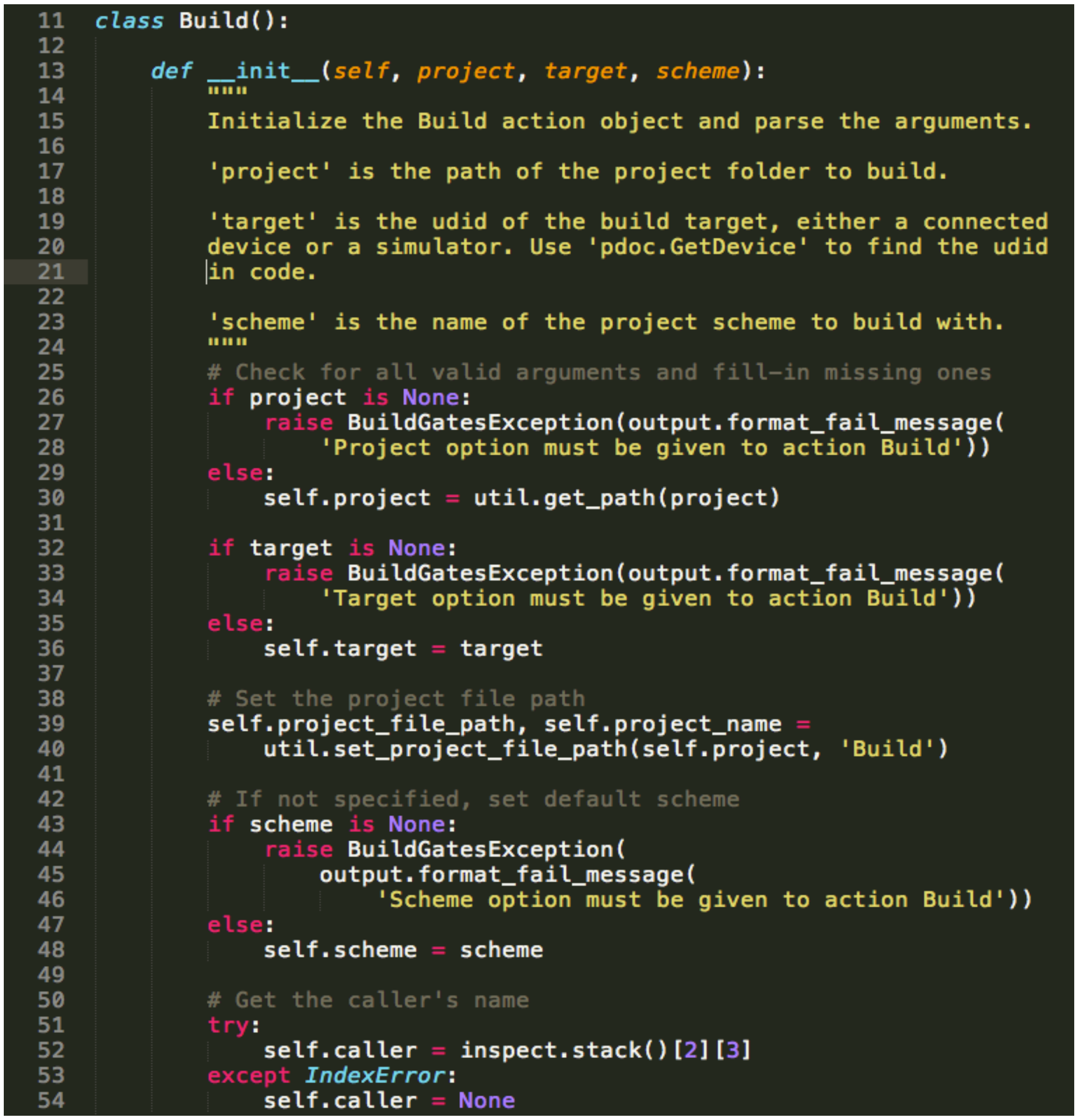
```python
def Build(project, target, scheme):
    """
    Builds an Xcode project.

    'project' is the path of the project folder to build.

    'target' is the udid of the build target, either a connected device or a
    simulator. Use 'pdoc.GetDevice' to find the udid in code.

    'scheme' is the name of the project scheme to build with.

    """
    __perform_pre_operations()
    build_object = BuildConstructor(project, target, scheme)
    build_object.execute()
```

Figure 12: Public API Method for Building a Project Calling Object Constructor Directly

3.3.2.2 Design Drawbacks

Through the implementation of the flat design, we were able to address the issues that we identified in Section 3.3.1. Still, there were some aspects that we had to sacrifice in order to address these issues.

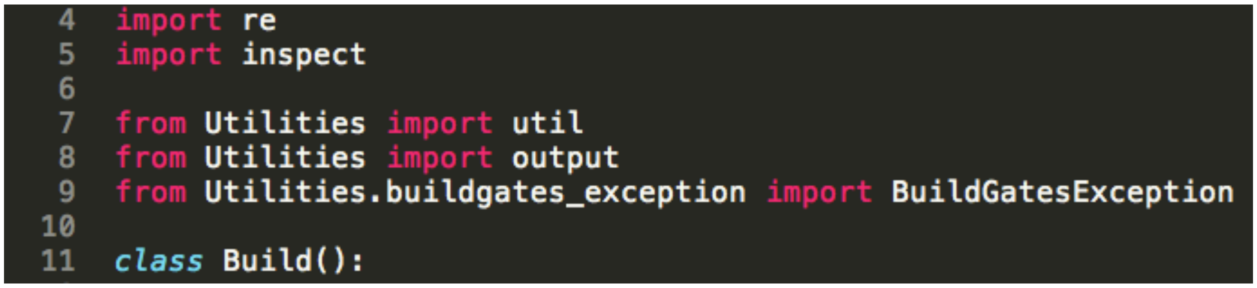
One of the major drawbacks from the flat design is the existence of some code duplication in the constructor of the framework actions. Since we removed the base action class, we no longer have a super constructor that takes care of argument parsing and variable initialization. As a result, as shown in Figure 13, every action must parse the arguments passed in. In addition, each action must also evaluate every variable that it needs for execution. Some of these variables, such as the “caller” variable shown in line 52 of Figure 13, are common to all actions, and could have been abstracted to the base action class.



```python
class Build():

    def __init__(self, project, target, scheme):
        """
        Initialize the Build action object and parse the arguments.

        'project' is the path of the project folder to build.

        'target' is the udid of the build target, either a connected
        device or a simulator. Use 'pdoc.GetDevice' to find the udid
        in code.

        'scheme' is the name of the project scheme to build with.
        """
        # Check for all valid arguments and fill-in missing ones
        if project is None:
            raise BuildGatesException(output.format_fail_message(
                'Project option must be given to action Build'))
        else:
            self.project = util.get_path(project)

        if target is None:
            raise BuildGatesException(output.format_fail_message(
                'Target option must be given to action Build'))
        else:
            self.target = target

        # Set the project file path
        self.project_file_path, self.project_name =
            util.set_project_file_path(self.project, 'Build')

        # If not specified, set default scheme
        if scheme is None:
            raise BuildGatesException(
                output.format_fail_message(
                    'Scheme option must be given to action Build'))
        else:
            self.scheme = scheme

        # Get the caller's name
        try:
            self.caller = inspect.stack()[2][3]
        except IndexError:
            self.caller = None
```

Figure 13: Constructor of Build Action Parsing Arguments and Initializing Variables

Another drawback of the flat design implementation is also related to the removal of the base action class. In the object-oriented design, the base action included all the required modules, and each action inherited the functions from it. Since the base class was removed, every action must now include the modules that it needs, as shown in Figure 14. This increases the number of individual dependencies, as shown in Figure 10, and makes the code harder to refactor since a change in one of the modules would require modifying every file that includes it.



```python
import re
import inspect

from Utilities import util
from Utilities import output
from Utilities.buildgates_exception import BuildGatesException

class Build():
```

Figure 14: Build Class Including all Required Modules Manually

3.3.2.3 Final Thoughts

Despite the drawbacks, the flat design framework fulfilled all of our original requirements. It is flexible and reliable, which makes it easy to maintain and extend. It also improves upon the drawbacks of the previous designs, which we discussed in previous sections. Based on these reasons and discussions with our sponsor, we decided to keep the flat design as our final implementation. In the next section, we will discuss how we incorporated this design to the rest of our project.

4 Integration

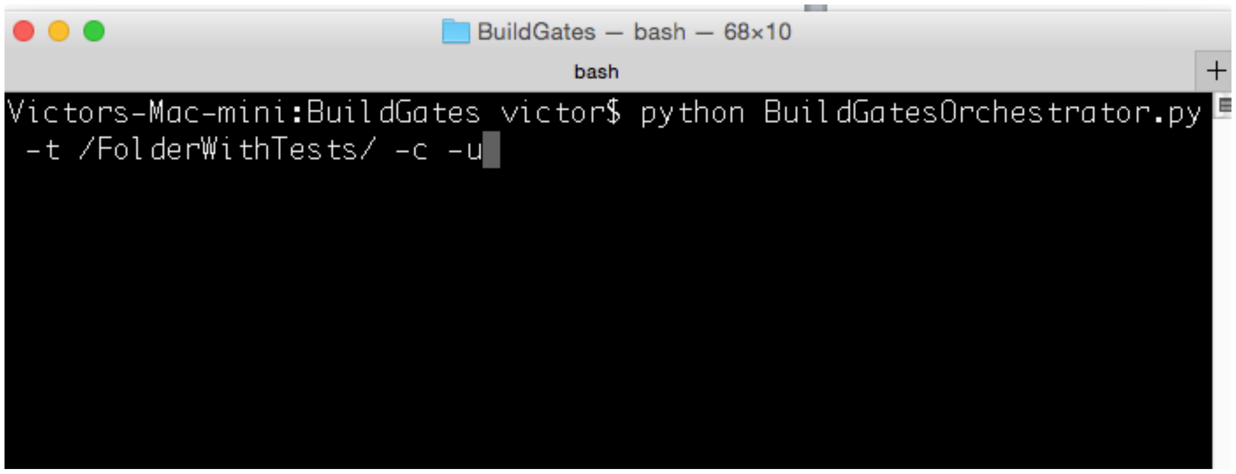
Once the framework had been designed, the more significant task of implementing it and integrating with the Intune team began. While we knew what the architecture of the system would look like, we still had to make it easy for developers to learn, use, and add tests to the system. In the following sections, we will explore the work that we undertook to make our testing framework into a valuable part of our sponsor's development process.

4.1 Testing Framework Implementation

One of the goals of creating the Python API was to implement an interface for developers to interact with the testing framework. Among the requirements for this interface was the ability to run all the tests in the codebase, run a subset of these tests, run only the tests that failed in the previous run, and trigger a source code build using the existing Intune build scripts. In order to encompass all of these features and allow for easy expandability, we decided to implement a Python script, called "BuildGatesOrchestrator", and a shell wrapper, called "BuildGates". By using this interface, developers can access the features of our testing framework and run test cases, which will be described in detail in Section 5.2.

The BuildGatesOrchestrator controls the selection and running of tests, as well as reporting their results. By using the Orchestrator, developers can specify which tests to run based on a series of filters. For example, they can select to run only component tests or only unit tests by passing in the "-c" or "-u" flags, as shown in Figure 15. In addition to filtering by the type of test, the orchestrator also supports filtering which test files or folders to run, as also shown in Figure 15. This allows for an increased level of filtering granularity.

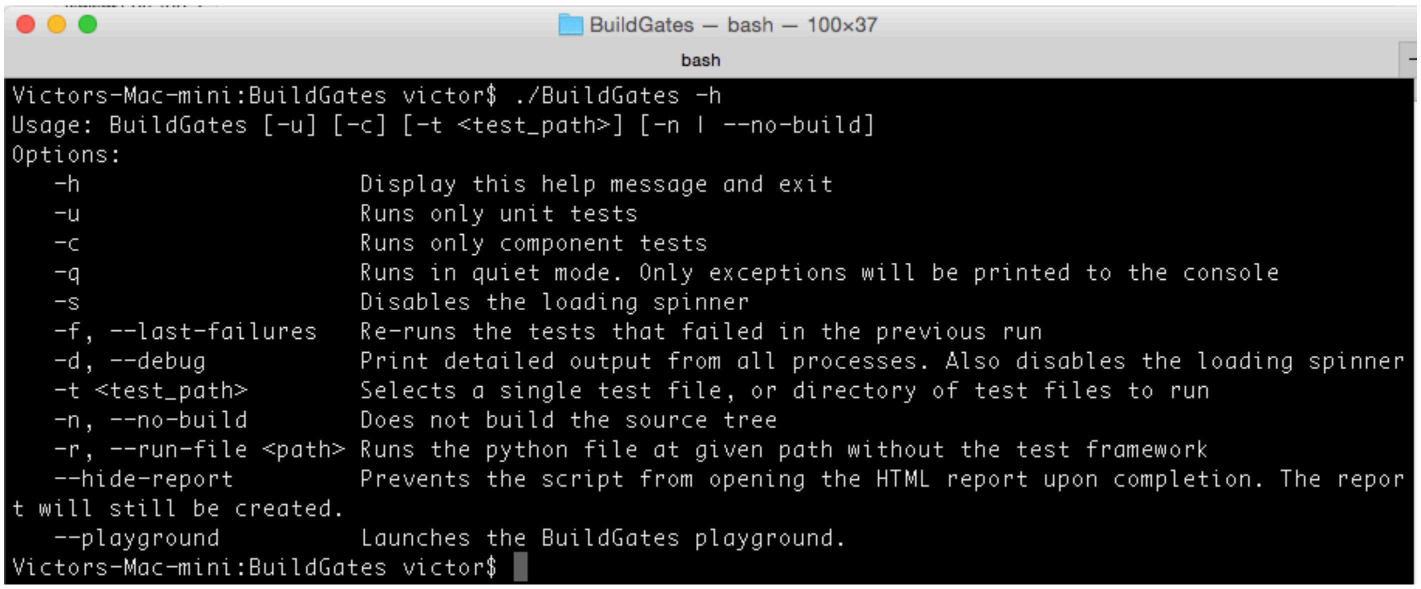
Finally, developers can select to run only those tests that failed in the previous run, making it easier to debug and prevent regression in the codebase.



```
Victors-Mac-mini:BuildGates victor$ python BuildGatesOrchestrator.py
 -t /FolderWithTests/ -c -u
```

Figure 15: Running the BuildGatesOrchestrator and Specifying Which Folder to Run and What Type of Tests to Execute

The BuildGates shell wrapper serves as the entry point to the BuildGatesOrchestrator. It allows developers to invoke the orchestrator without having to explicitly use the Python interpreter. In addition, the BuildGates script checks for proper environment variable initialization, basic argument parsing, and, if required, running the Intune build scripts. Figure 16 shows the different options available through the BuildGates shell script.



BuildGates — bash — 100×37

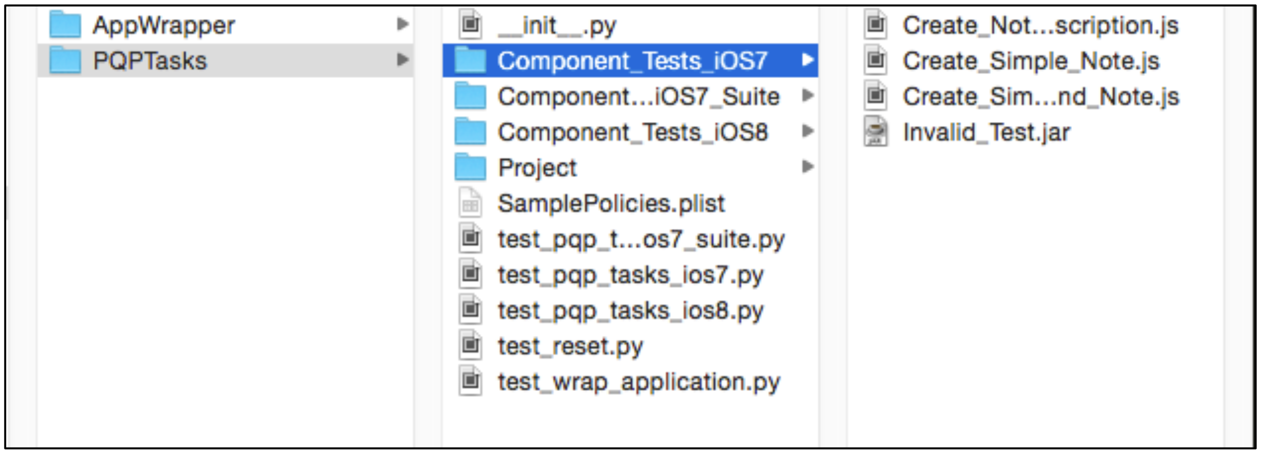
bash

```
Victors-Mac-mini:BuildGates victor$ ./BuildGates -h
Usage: BuildGates [-u] [-c] [-t <test_path>] [-n | --no-build]
Options:
   -h                        Display this help message and exit
   -u                        Runs only unit tests
   -c                        Runs only component tests
   -q                        Runs in quiet mode. Only exceptions will be printed to the console
   -s                        Disables the loading spinner
   -f, --last-failures       Re-runs the tests that failed in the previous run
   -d, --debug               Print detailed output from all processes. Also disables the loading spinner
   -t <test_path>            Selects a single test file, or directory of test files to run
   -n, --no-build            Does not build the source tree
   -r, --run-file <path>     Runs the python file at given path without the test framework
   --hide-report             Prevents the script from opening the HTML report upon completion. The repor
t will still be created.
   --playground              Launches the BuildGates playground.
Victors-Mac-mini:BuildGates victor$
```

Figure 16: Help Output of the BuildGates Script Showing the Different Options Available

4.2 Test Organization

For our system to be valuable, it was critical for us to provide an easy and intuitive way for developers to add and maintain tests. In order to do so, we organized each test case into a separate file, and stored all related files under a common folder, as shown in Figure 17. By having this structure, developers can add new tests by creating a file inside of the folder assigned to the project they are working on. If there is no folder for their project, they can create a new one and add their tests to it. Due to the implementation of the BuildGatesOrchestrator, developers do not have to specify that new tests were added; the orchestrator will detect and run them automatically. In the example shown in Figure 17, there are two project folders, AppWrapper and PQPTasks. All tests related to PQPTasks are placed inside the PQPTasks folder. If new tests were to be added to the PQPTasks project, the Python files would also be placed in the same location. If new tests for a new project were to be added, a new folder would be created at the same level as the AppWrapper and PQPTasks folders. In addition, all component tests related to the same project are placed in subfolders inside the project's folder. This is also shown in Figure 17, where there are three component test folders under the PQPTasks directory.

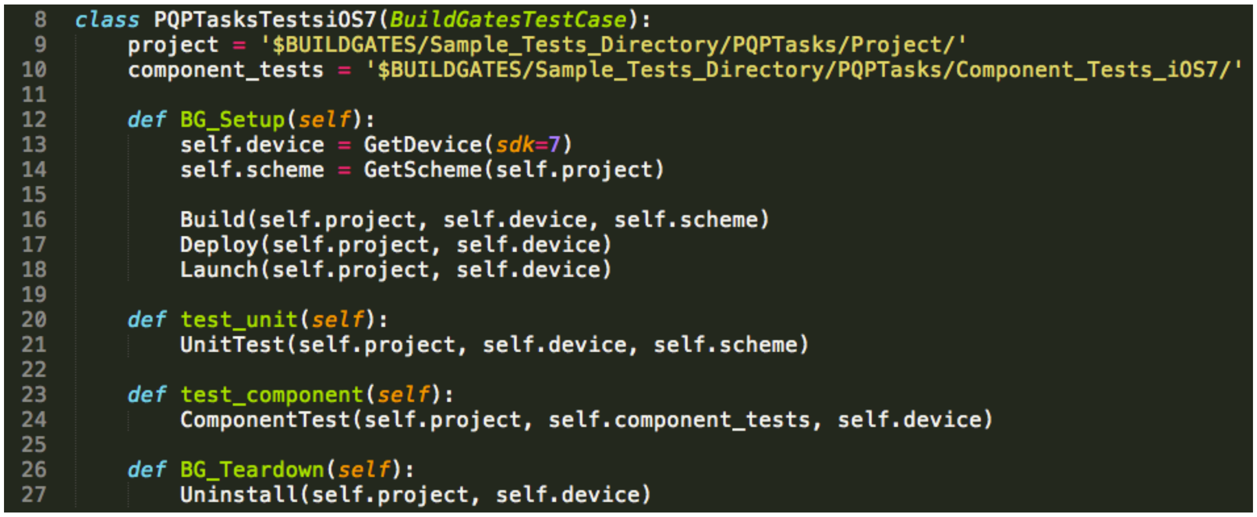

AppWrapper
PQPTasks

__init__.py
Component_Tests_iOS7
Component...iOS7_Suite
Component_Tests_iOS8
Project
SamplePolicies.plist
test_pqp_t...os7_suite.py
test_pqp_tasks_ios7.py
test_pqp_tasks_ios8.py
test_reset.py
test_wrap_application.py

Create_Not...scription.js
Create_Simple_Note.js
Create_Sim...nd_Note.js
Invalid_Test.jar

Figure 17: File System Organization of Tests

Inside each test case file, developers have access to four main functions: BG_Setup, test_unit, test_component, and BG_Teardown, as shown in Figure 18. The BG_Setup function allows developers to specify which actions must be executed before the test starts. These actions will be executed only once, and will affect all the tests within the same file. For example, this function can be used to build any required projects, select a device for testing, and deploy applications to the device. The BG_Teardown function, which serves as a complement to the BG_Setup function, is executed only once after all tests have run. This is intended to allow developers to clean up after their tests. For example, the BG_Teardown function can be used to uninstall applications that were installed during the test. Finally, as the names imply, the test_component and test_unit functions are used for running unit and component tests. Although developers can add any test functions that they want to their test files, test_unit and test_component are unique in that special flags can be passed to the orchestrator to run only these methods. For example, if developers want to run only the component tests of a file, they can pass the "-c" flag to the orchestrator. In this example, the orchestrator would look for the test_component function in the specified file(s) and execute them.



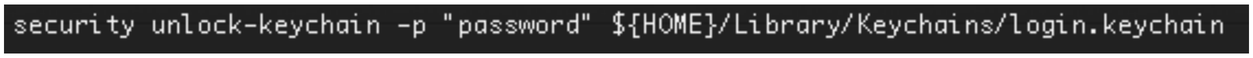
```python
class PQPTasksTestsiOS7(BuildGatesTestCase):
    project = '$BUILDGATES/Sample_Tests_Directory/PQPTasks/Project/'
    component_tests = '$BUILDGATES/Sample_Tests_Directory/PQPTasks/Component_Tests_iOS7/'

    def BG_Setup(self):
        self.device = GetDevice(sdk=7)
        self.scheme = GetScheme(self.project)

        Build(self.project, self.device, self.scheme)
        Deploy(self.project, self.device)
        Launch(self.project, self.device)

    def test_unit(self):
        UnitTest(self.project, self.device, self.scheme)

    def test_component(self):
        ComponentTest(self.project, self.component_tests, self.device)

    def BG_Teardown(self):
        Uninstall(self.project, self.device)
```

Figure 18: Test File Implementing BG_Setup, test_unit, test_component, and BG_Teardown

4.3 Implementing Continuous Integration

An additional step in integrating BuildGates with the Intune team was to set up a continuous integration server that would periodically run every test in the system. This server needed to pull all the code changes since the last run, build and test the new code using BuildGates, and automatically send out emails reporting the results of the tests and the names of the developers who made changes to the code since the last build. To accomplish this, we set up a Jenkins server that included a number of plugins to assist with the tasks that were required for every run.

This first involved setting up Jenkins on a Mac machine. The operating system of the machine running Jenkins was important because the majority of the actions in BuildGates utilize Xcode or other tools that are Mac dependent. When setting up a job on Jenkins, we wanted it to automatically pull the most recent code from the Intune Git repository. Although using Git as source control is not native to Jenkins, there is a Git plugin that was easily installable and offered the ability to report all of the commits made to the repository since the last build, as well as the individual who made each commit. Once the Jenkins job was properly set up with source control, actually running BuildGates was done by providing shell commands for the Jenkins server to execute. This script would essentially call BuildGates after some initial setup. However, one of the issues we faced when trying to use Jenkins to run BuildGates was that the process running the job did not seem to have the ability to access the keychain, which is where the Mac operating system stores passwords and identity information. Therefore, any time a project was built, it would fail because the code signing identity located in the keychain was unavailable and the project would not be properly code signed. To solve this, we needed to unlock the keychain by calling the command shown in Figure 19 before calling BuildGates.



```
security unlock-keychain -p "password" ${HOME}/Library/Keychains/login.keychain
```

Figure 19: Test File Implementing BG_Setup, test_unit, test_component, and BG_Teardown

The security command itself is a command line interface to the keychain. The actual command we wanted to run here was unlock-keychain, which needed the user password and the location of the keychain to unlock.

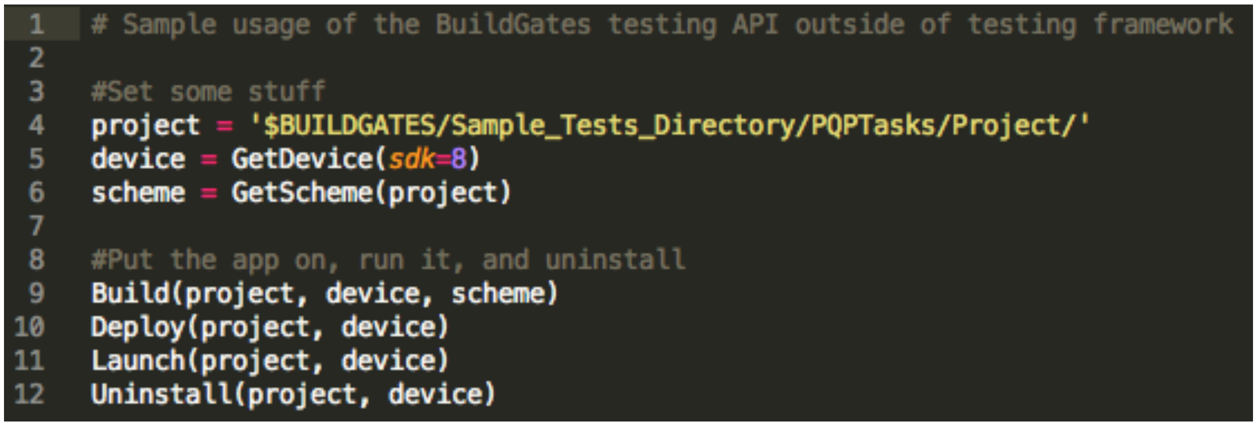
The final steps we took with Jenkins were reporting the output of BuildGates and emailing developers the status of each run. Since our framework already creates an HTML file containing the results of all the tests executed in a run, we found that the easiest way to report the output would be to make this HTML file viewable through Jenkins. This involved installing an HTML publisher plugin that would take the file we generated, link it to its associated build and give a URL that we could use to link to the HTML page. For emailing developers after each build, the default mailer plugin that comes with Jenkins did not give us enough functionality as it only sent email for every failed run instead of every run. In addition, the email's content was not customizable, which was a feature we were looking for. Therefore, an email extension plugin was also installed that allowed us to trigger email notifications for a variety of situations, not just failed runs. The new plugin also gave us the ability to control the content of the email that was sent out. With all of these steps taken, Jenkins turned out to be a very valuable way of automating periodic testing of the Intune system using BuildGates.

4.4 Onboarding and Documentation

With the system itself in a deliverable state and having established the process for running and writing tests, we were left with the question of how to help a new developer learn to use our testing framework. This onboarding process would have to include familiarizing developers with running the check-in tests, adding unit and component tests to a project, and creating multi-app tests with the BuildGates API. While not all members of the Intune team would need to know everything about the framework, it was still necessary to adequately explain all aspects of the system in order for it to be used effectively going forward.

The first step for us was to write comprehensive documentation, which covered every user-facing aspect of the system. This involved providing information on getting started with BuildGates, running tests, adding tests, and how to use Instruments for UI automation. This information was compiled in an OneNote notebook that was shared with the team, and as developers began to try the system, we worked with them to address gaps in the documentation.

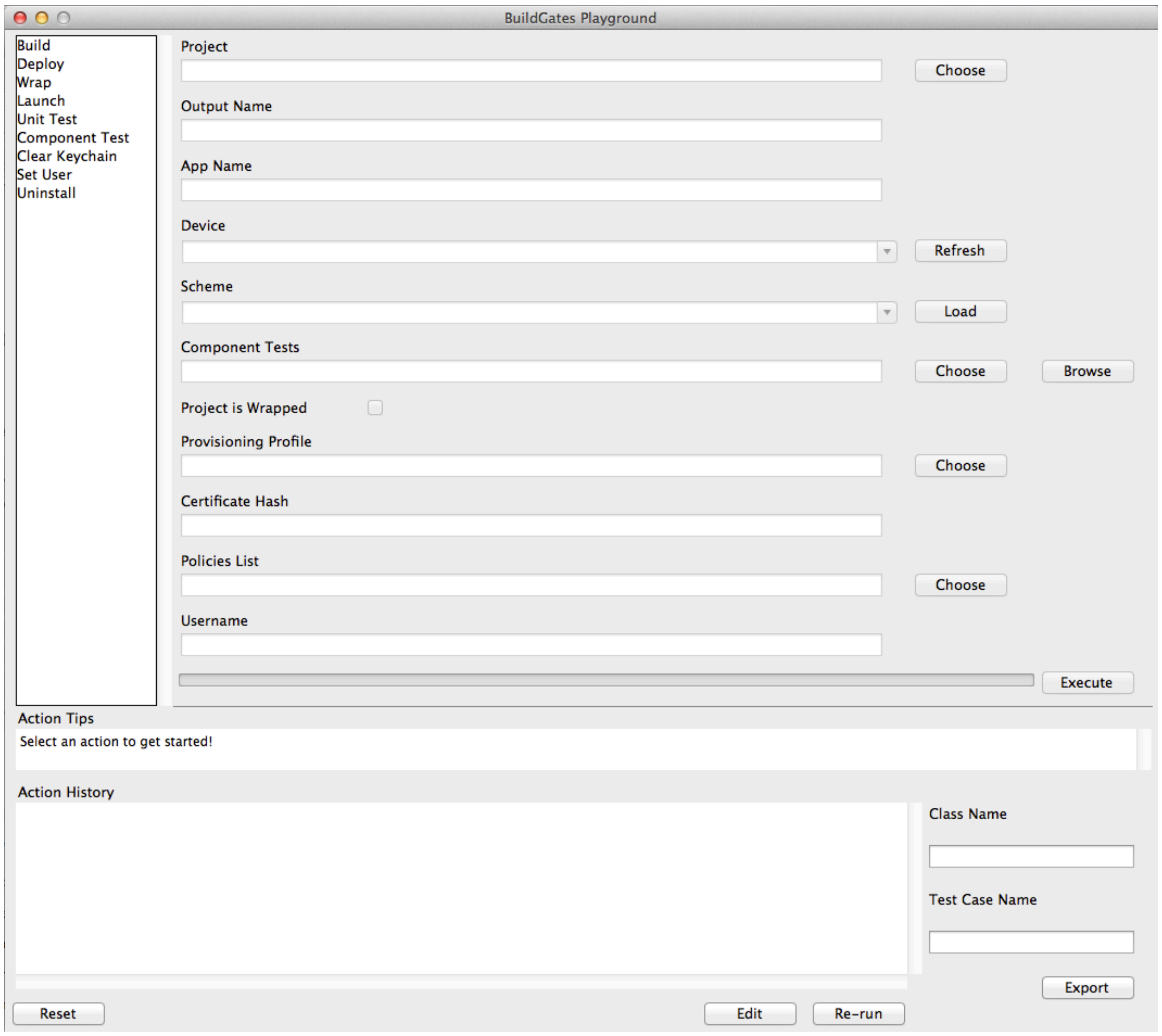
Along with documenting the system, we also introduced some additional features meant to help developers become familiar with the system. The first was a simple execution mode, which allows users to execute a single Python file without using the test framework. These files had none of the structural requirements of an actual framework test, so they were well suited for experimenting with the BuildGates API. An example of a simple script can be seen in Figure 20 below, showing the lack of syntactical overhead.



```
# Sample usage of the BuildGates testing API outside of testing framework

#Set some stuff
project = '$BUILDGATES/Sample_Tests_Directory/PQPTasks/Project/'
device = GetDevice(sdk=8)
scheme = GetScheme(project)

#Put the app on, run it, and uninstall
Build(project, device, scheme)
Deploy(project, device)
Launch(project, device)
Uninstall(project, device)
```

Figure 20 Example of a simple test script

The second feature, based on a suggestion by a member of the Intune team and shown in Figure 21, was a BuildGates Playground GUI that allowed users to try out API actions without writing any code. It also allows users to export a basic test case with the actions that they have run, bridging the gap between experimenting with the UI and writing a test. This GUI was not meant to be the primary approach for writing tests. Instead, the goal was to provide a way to let people become familiar with the system before getting caught up in the structural and syntax requirements of the testing framework.



Select an action to get started!

Figure 21 The BuildGates Playground GUI

The final, and most important, parts of our onboarding process were the Intune team members with whom we worked over the term. By the end of our project, there were several people who had become very familiar with the framework, each having written a number of tests that fully exercised the provided capability. These people will be an important resource for the team going forward since their hands-on knowledge will help the rest of the team learn how to best use the BuildGates system. Between these developers, our documentation, and the onboarding features included in BuildGates, we can be confident that our system will be used correctly after our project has ended.

5 Conclusion

This report outlines the background information, fundamental approaches, and integration steps that we used to create a check-in testing framework for the Microsoft Intune team. This framework provides developers with a fairly simple interface to interact with in order to test their code. It also plays a part in easing the transition to having all Intune developers begin testing their own code.

Our system, although written entirely in Python, utilizes a number of different tools to accomplish its goals. These tools include Xcode, Instruments, and iOS-deploy. The amount of flexibility our system required in order to support the numerous types of tests included caused us to explore several possible designs before settling on an API approach. Once our system was built, we needed to provide means for easily integrating it with the Intune team. This involved providing a simple to understand testing framework that developers could use to run our API, organizing test files in an understandable structure, and producing thorough documentation of how to use the system. Finally, we implemented a continuous integration server to run our system periodically on a larger test set. This allowed the system to detect if any bugs or failures were mistakenly added to the codebase, and quickly report them to the developers if required.

The need for having a simple to use testing framework showcases the importance of testing in software development, especially with large development teams that maintain even larger codebases. Catching failures and correcting them as quickly as possible allows a smoother development cycle and an easier to maintain codebase. We are confident that our

BuildGates system will provide all of these benefits to the Intune team and any other iOS development teams at Microsoft in the future.

6 Bibliography

Abdullah, Shazron. ios-deploy. 2014. 23 November 2014 <https://github.com/phonegap/ios-deploy>.

About Python. 23 Novemeber 2014 <http://legacy.python.org/about/>.

Automating UI Testing. 20 October 2014. 23 November 2014 <https://developer.apple.com/library/mac/documentation/DeveloperTools/Conceptual/InstrumentsUserGuide/UsingtheAutomationInstrument/UsingtheAutomationInstrument.html#//apple_ref/doc/uid/TP40004652-CH20-SW1 >.

Berg, Allan. Jenkins Continuous Integration Cookbook. Packt Publishing, 2012.

Building from the Command Line with Xcode FAQ. 21 May 2014. 23 November 2014 <https://developer.apple.com/library/ios/technotes/tn2339/_index.html >.

daViega, Nada. "Change Code WIthout Fear: Utilize a Regression Safety Net." Dr. Dobb's Journal (2008).

Dustin, Elfriede, Jeff Rashka and John Paul. Automated Software Testing -Introduction, Management, and Performance. Addison-Wesley, 1999.

Duvall, Paul M., Steve Matyas and Andrew Glover. Continuous Integration: Improving Software Quality and Reducing Risk. Addison-Wesley Professional, 2007.

Eliot, Seth. What is an SDET? 18 April 2010. Microsoft. 26 November 2014 <http://blogs.msdn.com/b/seliot/archive/2010/04/18/what-is-an-sdet.aspx>.

Instruments Quick Start. 20 October 2014. 24 November 2014 <https://developer.apple.com/library/mac/documentation/developertools/conceptual/instrumentsuserguide/InstrumentsQuickStart/InstrumentsQuickStart.html >.

Kawaguchi, Kohsuke. Plugins. 24 October 2014. 24 November 2014 <https://wiki.jenkins-ci.org/display/JENKINS/Plugins >.

Kohli, Aseem. Interview. 2 December 2014.

Meyer, M. "Continuous Integration and Its Tools." Software, IEEE (2014): 14,16.

Microsoft Corporation. Microsoft Intune. 26 November 2014 <http://www.microsoft.com/en-us/server-cloud/products/microsoft-intune/>.

Myers, Glenford. The Art of Software Testing. Wiley, 2004.

Savenkov, Roman. How to Become a Software Tester. Roman Savenkov Consulting, 2008.

Using Python on a Macintosh. 23 November 2014 <https://docs.python.org/2/using/mac.html>.