ComS 362 Design Project

Upgrading TestNG

Contributing Members:
Kendall Berner, William Fuhrmann, Christian Shinkle, Ben Trettin

Table of Contents

Introduction

TestNG is an open source project designed to be an efficient and complete unit testing software for java. TestNG contains features that allow the tests to be tailored to the software being tested. TestNG also supports running tests asynchronously. TestNG functions very similar to JUnit and other testing frameworks, and any familiarity with them will quickly translate into knowledge of how to write and run TestNG tests.

The feature we designed will be the ability to use a @Run_in annotation inside of test classes to specify environment specific variables. This will allow the same test class to be run in many different environments (development, integration lab, field, etc.). Using the @Run_in annotation will remove the need to rewrite test cases each time there is a new environment the tests need to be run in. Our proposed feature will capitalize on TestNG's annotation support to add the @Run_In feature with minimal refactoring required. The environment will be determined by command line input, then the @Run_in annotation will set the environment variables as specified in the annotation.

Technical Overview and Context

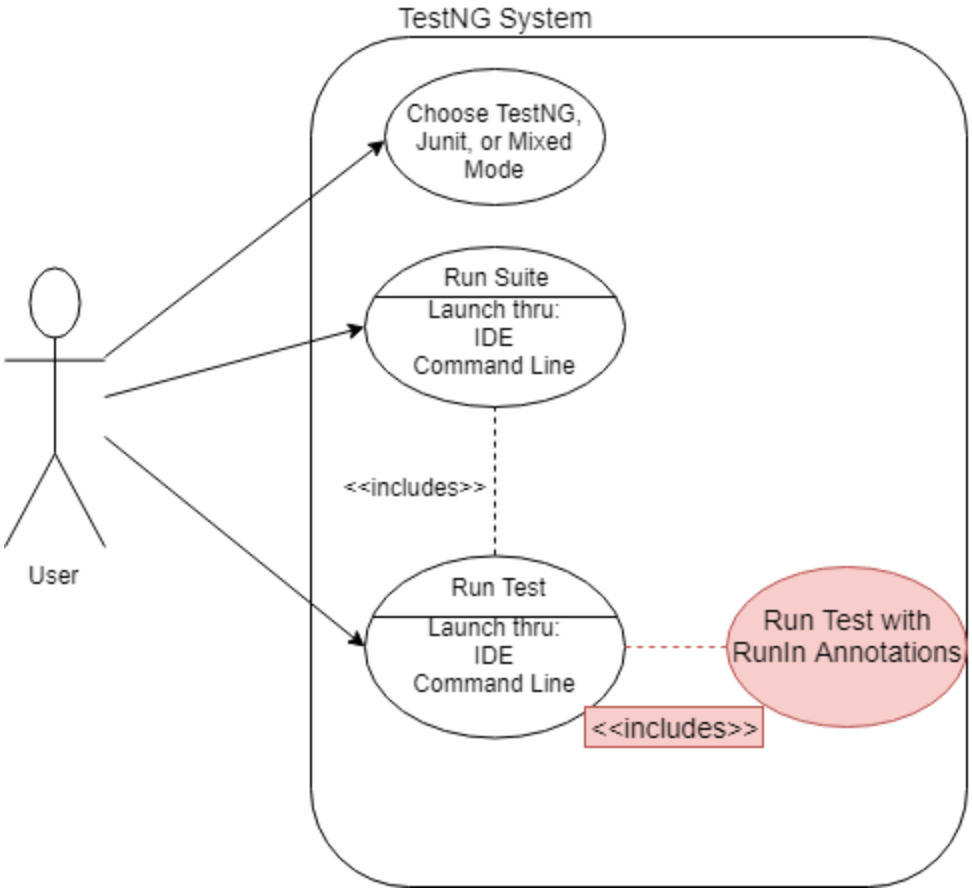
The next few pages will give a technical overview of TestNG as a whole. It will include a use case diagram, a domain diagram, and package diagram to give a fly-over of how the software is organized and how it interacts.

Use Case Diagram

As the Use Case diagram shows, TestNG has three main use case. The first is choosing which mode the framework will run in. TestNG supports running its own version of test, JUnit test for compatibility purposes, and a mixture of both for codes bases that may have existing tests in JUnit but want to transition into TestNG.

The second use case involves running a test suite. This allows developers to run several test files in one batch, which is one of the major advantages of TestNG over JUnit.

The third Use Case involves running test individually or file-by-file. This is useful for developers who are trying to work on bugs for a specific feature, but don't want to wait to run an entire suite of tests.



Detailed Use Case Scenarios

Use Case: Tester runs a test class
Trigger: Tester initiates test
Precondition: Test class is correctly compiled and configured.
Actor: Tester
Scenario:

1. The tester navigates to test he/she wishes to run.
2. The tester clicks the “run TesNG” in their IDE.
    a. Alternative: The testerenters “testng” command on the command line with the necessary options.
3. The test executes and the results are displayed in the IDE.
    a. Alternative: The results are printed in the command line.

Use Case: Tester runs a suite
Actor: Tester
Precondition: All tests in the suite are correctly compiled and configured

Trigger: Tester initiates test suite
Scenario: Tester runs test suite

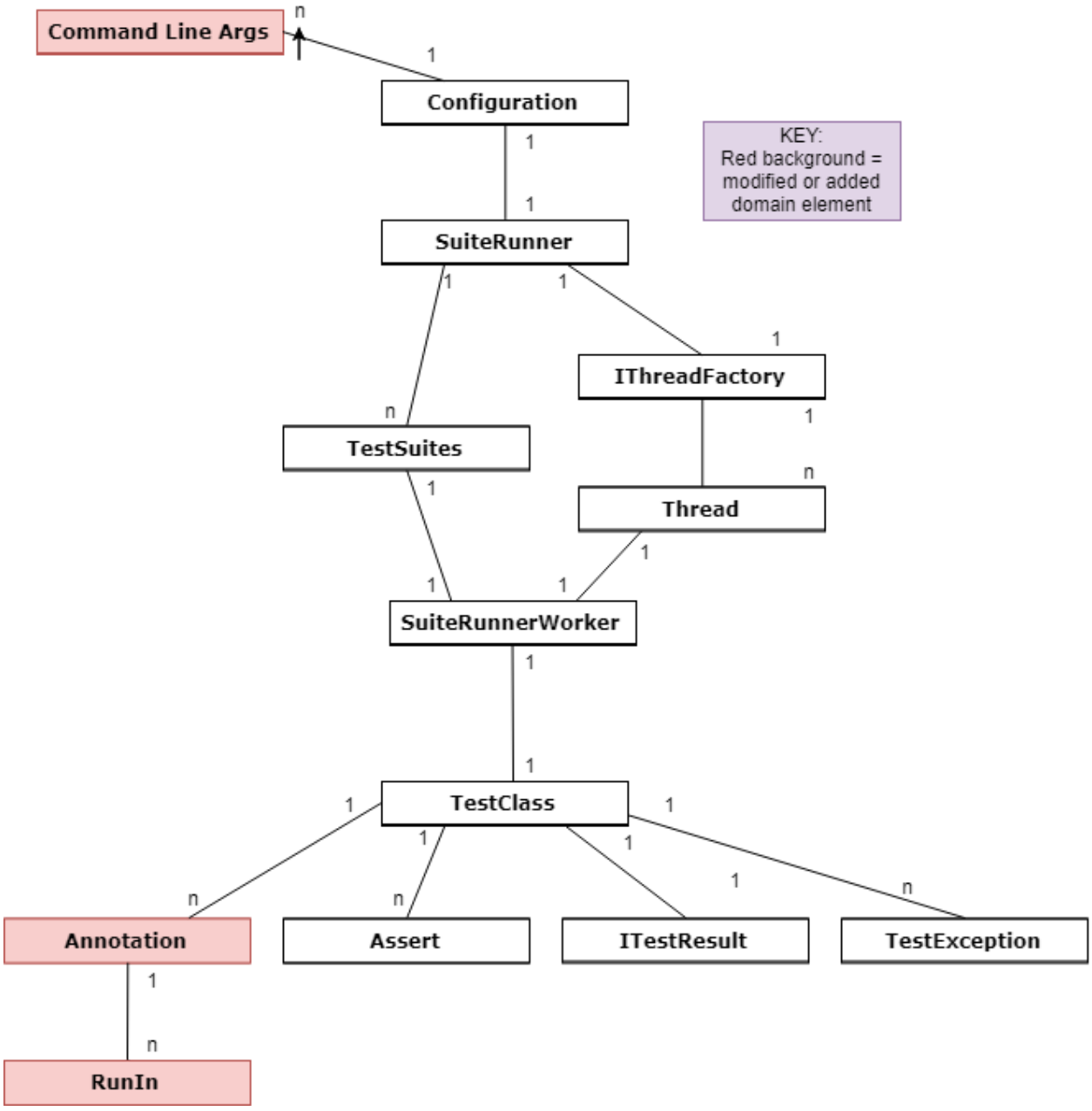
1. The tester navigates to the suite they want to run(either from IDE or Command Line)
2. The tester right clicks on the test suite and clicks the “run test suite” button
    a. Alternative: User uses command line and…….
3. The test executes the suite tests and returns the results
4. The results are displayed in the IDE
    a. Alternate: The results are printed in the command line.

Use Case: Tester configures specifications
Actor: Tester
Precondition: None other than having the necessary program installed
Trigger: Tester accesses the xml file or IDE menu
Scenario: Tester sets configuration mode

1. Tester opens menu on IDE and clicks on configure
    a. Alternate: User opens XML file pertaining to the test he/she wants to configure.
2. Tester sets the mode(Testng,Junit,Mixed Modes) for the test.

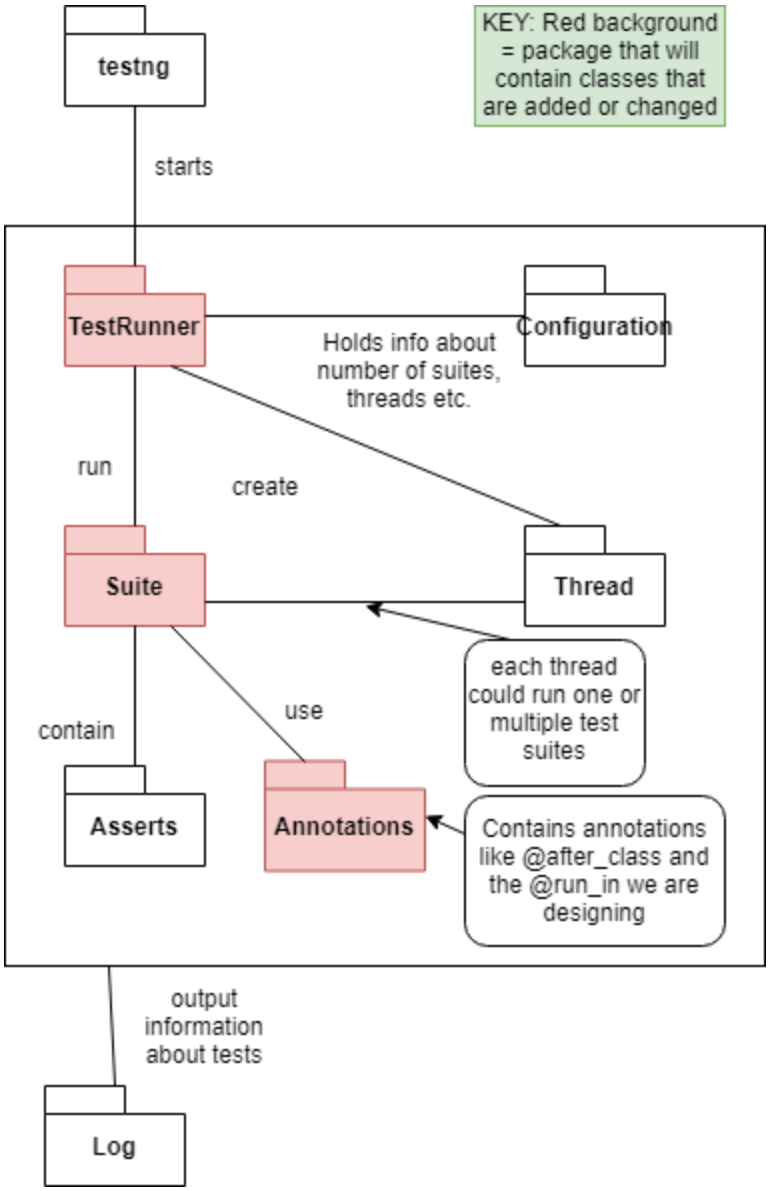
Domain Model

This domain model describes all domain elements used in running a simple TestNG test suite. It also includes the Run_in class that will be added to implement our feature. The Annotations interface has many other classes that implement it, but the only class needed to run a TestNG test suite that contains the @Run_in tag is the Run_in class.



Package Diagram with Updates

All of the packages in the box except the Annotations package are currently a part of the TestNG package. We split them up into subpackages in the following manner to make it easier to understand what is being done where. The Annotations package is a separate package in the current design and we will keep it that way, because we thought that is the most logical and clear way to do it.



Detailed Interface description:

The three most important interfaces we found that pertain to our new design feature were IAnnotation, Parameters, and ITestClass. IAnnotition is the parent interface for annotations like BeforeClass, AfterClass, BeforeSuite, and AfterSuite, etc. Our RunIn annotation will be an implementation of the IAnnotation interface.The Parameters interface describes how to pass parameters to the TestRunner class or to individual TestClass objects. Parameters is how we plan to store the new environment variable indicating the environment the tests are being run from. The ITestClass is the parent interface for TestClass objects. TestClass objects are the test sets that use annotations and will use our RunIn annotation once it is developed.

Development environment

The development environment mainly consists of Java tools and IDE's. For our design analysis, we performed all of our tests on Eclipse. The common Java testing tools, such as JUnit and Groovy, are compatible with TestNG's code base. The project uses a gradlew script to add and build dependencies from a maven repository. Many of the dependencies were written by the author of TestNG himself, so these dependencies couldn't be tracked down and make it all but necessary to use his personal gradlew scripts. Some of the dependencies include Jcommand, a tool used to create and manage annotations in Java, and JUnit, used as the testing suite for TestNG.

Code Characterization

Code Smell Report

To identify potential code smells and issues with the existing testng code, we as a team decided to use a couple of code analysis programs called pmd and cpd. The pmd test we used focuses only on design patterns and top-level code smells that might be occurring. Even after only accounting for design smells, the report still outputted 936 potential smells. Many of the smells that the program gave us were common smells that are more or less code convention and not design flaws; this includes empty method bodies, parameter reassignment, and variable assignments. The creator of testng may not be following these conventions, but they are not particularly concerning to us, because they will not have a real impact to us when we design our new feature. A couple of smells that may have an impact on us when we design our feature are calling overridable methods during object construction and some classes with private constructors are not final. Both of these smells could potentially cause us problems, because if our design feature also calls this overridable method, it may not allow us to call super() on the

superclass. Overall, the code smells appeared to be unalarming to us as we move towards implementing our new design feature.

The CPD report we used had a token minimum of 100 in order to really see the duplicated code that was more seriously needing fixed. After our CPD report, we found very few cases that indeed actually had duplicated code with more than 100 tokens. The 171 tokens that he is duplicating is the first main chunk of two assertEqual methods. The only difference between the two is the second method is an assertEquals Deep. What he could do in order to fix this issue is create a helper method that runs these 171 tokens and has both his two methods call that helper method at the beginning. The second and last duplicated code that the CPD program found was a little under 20 lines. This code appears to be in a similar boat as the first CPD, in that there is some duplicated code within two methods that could potentially be reduced down to a helper method. The one glaring issue with doing that with this code though, is that the two method are not in the same class or even in the same package. With this little of duplicated code, in may not be necessary to create a helper method or even viable with how the packages are set up.

Overall, the code analysis programs helped us as a team identify potential design flaws in the code that we need to be aware of moving forward. We have come to a conclusion that there are no major design flaws that we would need to address as a team to spend the time to refactor.

Tabular Summary

- Coverage: 68%
- Lines: 17,813
- Classes: 431
- Packages: 24
- Test Classes: 319
- Test Cases: 1641
- Cyclomatic Peak: 232 (Invoker)

From this tabular summary we can glean that TestNG has very good code coverage at 68%. The project is medium sized, and has an exceptional number of tests for its size. The cyclomatic peak is extremely high, which makes the Invoker class intimidating to mess with. Fortunately, our feature doesn't require any new interactions with it.

Detailed Design

Description of Changes

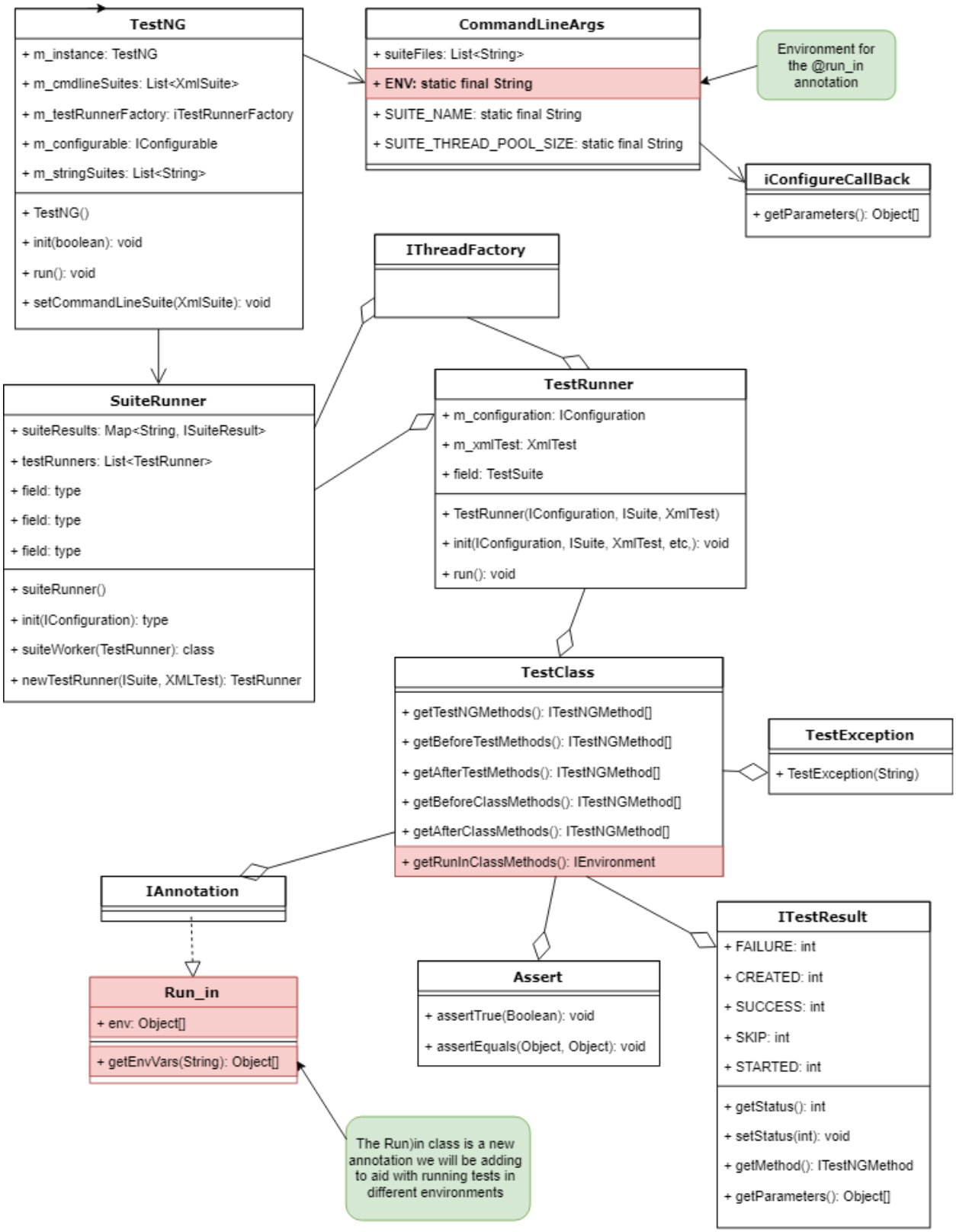
Our new feature we will add will be the ability to use a @Run_in tag on testng tests. Using this tag will allow different environments can run the same test and the tests can use different configurations to match the environment the tests are run in. Environment will be set using the command line.

The patterns that the current system use are Chain of Responsibility, and Singleton. The current system uses a chain of Responsibility to get the user input, then set up the test suite, run the tests, and finally output the results. The main method in TestNG.java calls the CommandLineArgs.java to get user input and set variables in the Parameters.java class. Then TestNG.java calls SuiteRunner to handle the running of the suites. Another pattern the system utilizes is the singleton pattern, the Parameters class instantiates a singleton object that holds all of the parameters needed by TestRunner, and TestClass. The TestNG class stores a TestNG object as a singleton. This is fine since there will only ever be one TestNG per time running the program. It is also good, because the TestNG singleton could be needed by some classes in the project. It is not accessed by any of the classes that we are dealing with for our change, but it is certainly good design practice to make it a singleton for future changes, or changes elsewhere in the project.

The classes we will need to update are Configuration.java, CommandLineArgs.java, Annotation.java, Parameters.java, and we will have to create a new class in the Annotations package, RunIn.java. In Parameters.java we will have to add a variable to store the default environment variable or the environment variable set by the user. In CommandLineArgs.java, we will have to add another optional parameter that will allow the user to specify the environment that the tests are being run in. We will have to modify Configuration.java to add the RunIn method to the list of possible TestNG methods.

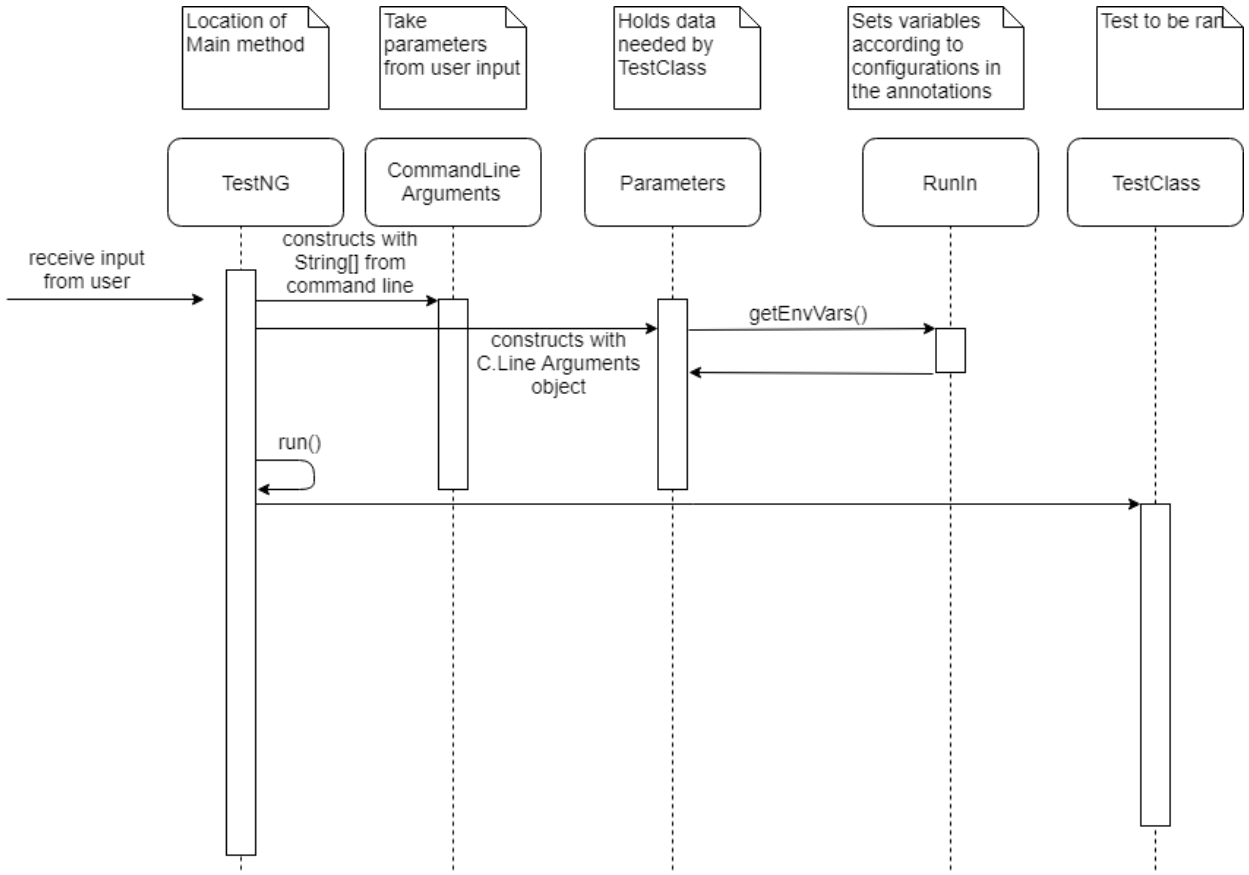
The new RunIn.java class will implement the Annotations.java interface. The After_Class and Before_Suite annotations also implement the Annotations interface. The RunIn.java class will hold environment dependent variables that are different between the different testing environment. An example of two different testing environments would be integration testing and routine developer tests. The routine developer tests are tests that might be run overnight on a development machine with simulated data from a file. The integration testing environment would be tests that are run in an environment with real sensor data. The RunIn annotation would allow a developer/test engineer to run the same tests on different sets of data.

Detailed Class Diagram with Changes



Sequence Diagram

The following diagram show the chronological order of interactions between classes that interact with RunIn. The only class that with directly class RunIn is Parameters. Parameters will then use this information and assign it to information used in the TestClass for its respective unit test.



Test Impact

Because the core logic of TestNG won't be changed due to this new feature, the testing suite we have now shouldn't need to be modified very much. The only new test that will need to be added will be to check environment variables were set correctly and can be retrieved correctly.

The test coverage for the annotation classes is very thorough with 88% of instructions covered and 82% of branches covered. This means if there were any bugs introduced by the new

code, it would be caught very quickly. The only downside would be in order to maintain this high level of coverage, many, thorough tests for RunIn may be required.

Level of Effort Estimates

	Counts	Function Points	Max Function Points
Input Classes	1	3	6
Input Attributes	5		
Output Classes	1	4	7
Output Attributes	5		
Interactive Classes	3	4	6
Interactive Attributes	8		
Internally Stored Classes	2	7	15
Internally Stored Attributes	14		
Externally Stored Classes	3	5	10
Externally Stored Attributes	12		
Total		23	44
Staff Months		0.590445132	1.492952768

Our proposed new feature is fairly lightweight in terms of interacting with the existing codebase. Following the guide by Robertson & Robertson and Capers Jones in determining level of effort for software projects, we found that our feature has only 23 function points, calculating out to a total of just .59 staff months, or a little over 2 weeks for 1 developer working full time. Applying a 20% uncertainty allowance gives us a range of .47 to .71 staff months. We recommend erring on the side of caution and assume that the task will take between .6 and .7 staff months.

Staffing Plan	1 Developer														
Time	15 Days														
	Week 1					Week 2					Week 3				
	M	T	W	R	F	M	T	W	R	F	M	T	W	R	F
Developer 1															

Staffing Plan	2 Developers														
Time	7 Days														
	Week 1					Week 2									
	M	T	W	R	F	M	T	W	R	F					
Developer 1															
Developer 2															

Staffing Plan	4 Developers														
Time	4 Days														
	Week 1														
	M	T	W	R	F										
Developer 1															
Developer 2															
Developer 3															
Developer 4															

The following plan should only be attempted if you have a team of four that works very well together, and an absolute need to get this feature done as soon as possible. For the earliest practical delivery of this feature, we would suggest assigning four developers to the task with a deadline of three days. This works out to about .60 staff months, which is dangerously close to the early end of our estimate. Any more developers would almost certainly slow down the project. Four is already pushing feasibility, so we really don't recommend this staffing plan. It is worth keeping in mind however, in case you begin the project with 1 or 2 developers and realize after a week that the feature is not as simple as first anticipated.

A minimal staff for this project would be a single developer. They would need two or three weeks to complete the project, which isn't so long that a lone worker would become overwhelmed. The codebase is large, but our feature doesn't require an extensive knowledge of all of it. A single developer would be able to learn enough about it to implement our new feature, and not have to worry about the overhead costs that come with teams working together.

The most reasonable staffing plan would be two developers working for 7 days. This plan affords the team the benefits of teamwork without the drawbacks of stuffing a lot of people into a small project to get it out the door faster. The developers will be able to help each other through roadblocks without getting in each other's way the rest of the time. If staff and time allows, this is our recommended strategy for implementing our feature.

Resources

Website with all of the testng documentation:
Testng.org

Website with all the source code:
github.com/cbeust/testng

Google forum for asking questions:
https://groups.google.com/forum/#!forum/testng-users

Testng book
https://www.amazon.com/Next-Generation-Java-Testing-Advanced/dp/0321503104/ref=pd_bbs_sr_1/104-7105897-1187923?ie=UTF8&s=books&qid=1193417110&sr=8-1

Process to Build TestNG

1. Clone into the repository with the command: git clone https://github.com/cbeust/testng.git
2. Checkout the most recent release. As of 4/20/18, that is build 6.14.3. Use the command: git checkout 67b8f040aae98088cab6331a78d66c9536f87b27
3. Run the command: gradlew tasks
4. Run the command: gradlew build jar
5. If you are working in Eclipse as we did, run the command: gradlew eclipse
6. Import the project into Eclipse
7. If you want code coverage, follow the following steps
    a. Run the command: gradlew jacocoTestReport
    b. Navigate to <TestNGDirectory>\build\reports\jacoco\test\html\index.html