Examining Usage Patterns of the FIT Acceptance Testing Framework

Kris Read, Grigori Melnik, Frank Maurer

Department of Computer Science, University of Calgary
Calgary, Alberta, Canada
{readk, melnik, maurer}@cpsc.ucalgary.ca

Abstract. Executable acceptance testing allows both to specify customers' expectations in the form of the tests and to compare those to actual results that the software produces. The results of an observational study identifying patterns in the use of the FIT acceptance testing framework are presented and the data on acceptance-test driven design is discussed.

1 Introduction

Acceptance testing is an important aspect of software development. Acceptance tests are high level tests of business operations and are not meant to test internals or technical elements of the code, but rather are used to ensure that software meets business goals. Acceptance tests can also be used as a measure of project progress. Several frameworks for acceptance testing have been proposed (including JAccept [5], Isis [6], and FIT [2]). The use of acceptance tests have been examined in recent studies from members of both academia [8, 7] and industry [5,10]. We are interested in determining the value of executable acceptance tests, both for quality assurance as well as to represent functional requirements in a test-first environment. To this end we have arranged several experiments and observational studies using tools such as FIT [2] and FitNesse [3] to work with executable acceptance tests. FIT is an acceptance testing framework which has been popularized by agile developers. FIT allows tests to be specified in tables, in multiple formats such as HTML, Excel, or on a wiki page. Although users can specify the test case tables, developers must later implement fixtures (lightweight classes calling business logic) that allow these tables to be executed. Suitability of acceptance tests for specifying functional requirements has been closely examined in our previous paper [4]. Our hypothesis that tests describing customer requirements can be easily understood and implemented by a developer who has little background on this framework was substantiated by the evidence gathered in our previous experiment. Over 90% of teams delivered functioning tests and from this data we were able to conclude that the learning curve for reading and implementing executable acceptance tests is not prohibitively steep.

In this paper we expand on our previous results and investigate the ways in which developers use executable acceptance tests. We seek to identify usage patterns and gather information that may lead us to better understand the strengths and weaknesses of acceptance tests when used for both quality control and requirements

representation. Further, examining and identifying patterns may allow us to provide recommendations on how acceptance tests can best be used in practice, as well as for future development of tools and related technologies. In this paper we report on results of observations in an academic setting. This exploratory study will allow us to refine hypotheses and polish the experimental design for future industrial studies.

2 Context of Study

Data was gathered from two different projects in two different educational institutions over four months. The natures of the two projects were somewhat different; one was an interactive game, and another a Web-based enterprise information system. The development of each project was performed in several two to three week long iterations. In each project, FIT was introduced as a mandatory requirement specification tool. In one project FIT was introduced immediately, and in the other FIT was introduced in the third iteration (half way through the semester). After FIT was introduced, developers were required to interpret the FIT-specified requirements supplied by the instructor. They then implemented the functionality to make all tests pass, and were asked to extend the existing suite of tests with additional scenarios.

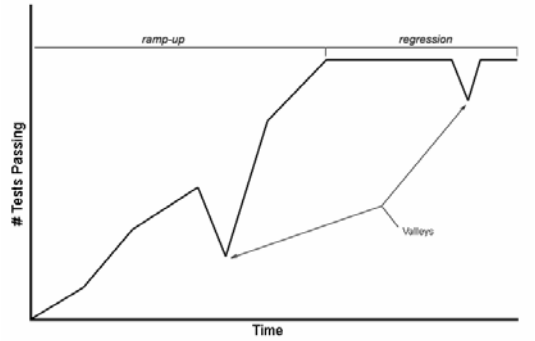

Fig. 1. Typical iteration life-cycle

The timeline of both projects can be split into two sections (see Figure 1). The first time period begins when students received their FIT tests, and ends when they implemented fixtures to make all tests pass. Henceforth this first time period will be called the "*ramp up*" period. Subjects may have used different strategies during ramp up in order to make all tests pass, including (but not limited to) implementing business logic with in the test fixtures themselves, delegating calls to business logic classes from test fixtures, or simply mocking the results within the fixture methods (Table 1). The second part of the timeline begins after the ramp up and runs until the end of the project. This additional testing, which begins after all tests are already passing, is the use of FIT for *regression testing*. By executing tests repeatedly, developers can stay alert for new bugs or problems which may become manifest as they make changes to the code. It is unknown what types of changes our subjects might make, but possibilities range from refactoring to adding new functionality.

Table 1. Samples of how a given fixture could be implemented

Example: In-fixture implementation

```
public class Division extends ColumnFixture {
  public double numerator, denominator;
  public double quotient() { return numerator/denominator;  }
}
```

Example: Delegate implementation

```
public class Division extends ColumnFixture {
  public double numerator, denominator;
  public double quotient() {
    DivisionTool dt = new DivistionTool();
    return dt.divide(numerator, denominator);
  }
}
```

Example: Mock implementation

```
public class Division extends ColumnFixture {
  public double numerator, denominator;
  public double quotient() { return 8; }
}
```

3 Subjects and Sampling

Students of computer science programs from the University of Calgary (UofC) and the Southern Alberta Institute of Technology (SAIT) participated in the study. All individuals were knowledgeable about programming, however, no individuals had any knowledge of FIT or FitNesse (based on a verbal poll). Senior undergraduate UofC students (20) who were enrolled in the *Web-Based Systems*[1] course and students from the Bachelor of Applied Information Systems program at SAIT (25) who enrolled the *Software Testing and Maintenance* course, took part in the study. In total, 10 teams with 4-6 members were formed.

4 Hypotheses

The following hypotheses were formulated prior to beginning our observations:

H_A: No common patterns of ramp up or regression would be found between teams working on different projects in different contexts.

H_B: Teams will be unable to identify and correct "bugs" in the test data or create new tests to overcome those bugs (with or without client involvement).

H_C: When no external motivation is offered, teams will not refactor fixtures to properly delegate operations to business logic classes.

H_D: When no additional motivation is given, students will not continue to the practice of executing their tests in regression mode (after the assignment deadline).

H_E: Students will not use both suites and individual tests to organize/run their tests.

[1] http://mase.cpsc.ucalgary.ca/seng513/F2004

5 Data Gathering

A variety of data gathering techniques were employed in order to verify hypotheses and to provide further insight into the usage of executable acceptance testing. Subjects used FitNesse for defining and executing their tests. FitNesse [3] is an open-source wiki-based tool to manage and run FIT tests. For the purposes of this study, we provided a binary of FitNesse that was modified to track and record a history of FIT test executions, both successful and unsuccessful. Specifically, we recorded:

- Timestamp;
- Fully-qualified test name (with test suite name if present);
- Team;
- Result: number right, number wrong, number ignored, number exceptions.

The test results are in the format produced by the FIT engine. *Number right* is the number of passed assertions, or more specifically the number of "green" table cells in the result. *Number wrong* is the number of failed assertions, which are those assertions whose output was different from the expected result. In FIT this is displayed in the output as "red" table cells. *Ignored* cells were for some reason skipped by the FIT engine (for example due to a formatting error). *Number exceptions* records exceptions that did not allow a proper pass or fail of an assertion. It should be noted that a single exception if not properly handled could halt the execution of subsequent assertions. In FIT exceptions are highlighted as "yellow" cells and recorded in an error log. We collected 25,119 different data points about FIT usage.

Additional information was gathered by inspecting the source code of the test fixtures. Code analysis was restricted to determining the type of fixture used, the non-commented lines of code in each fixture, the number of fields in each fixture, the number of methods in each fixture, and a subjective rating from 0 to 10 of the "fatness" of the fixture methods: 0 indicating that all business logic was delegated outside the fixture (desirable), and 10 indicating that all business logic was performed in the fixture method itself (see Table 1 for examples of fixture implementations).

Analysis of all raw data was performed subsequent to course evaluation by an impartial party with no knowledge of subject names (all source code was sanitized). Data analysis had no bearing or effect on the final grades.

6 Analysis

This section is presented in four parts, each corresponding to a pattern observed in the use of FIT. *Strategies of test fixture design* looks at how subjects construct FIT tables and fixtures; *Strategies for using test-suites vs. single tests* examines organization of FIT tests; *Development approaches* identifies subject actions during development; and *Robustness of test specification* analyzes how subjects deal with exceptional cases.

6.1 Strategies of Test Fixture Design

It is obvious that there are multitudes of ways to develop a *fixture* (a simple interpreter of the table) such that it satisfies the conditions specified in the table (test case). Moreover, there are different strategies that could be used to write the same fixture. One choice that needs to be made for each test case is what type of FIT fixture best suits the purpose. In particular, subjects were introduced to RowFixtures and ActionFixtures in advance, but other types were also used at discretion of the teams (see Table 2). Some tests involved a combination of more than one fixture type, and

Table 2. Common FIT fixtures used by subjects

Fixture Type	Description	Frequency of Use
RowFixture	Examines an order-independent set of values from a query.	12
ColumnFixture	Represents inputs and outputs in a series of rows and columns.	0
ActionFixture	Emulates a series of actions or events in a state-specific machine and checks to ensure the desired state is reached.	19
RowEntryFixture	Special case of ColumnFixture that provides a hook to add data to a dataset.	2
TableFixture	Base fixture type allowing users to create custom table formats.	30

subjects ended up developing means to communicate between these fixtures.

Another design decision made by teams was whether to develop "fat", "thin" or "mock" methods within their fixtures (Table 3). "Fat" methods implement all of the business logic to make the test pass. These methods are often very long and messy, and likely to be difficult to maintain. "Thin" methods delegate the responsibility of the logic to other classes and are often short, lightweight, and easier to maintain. Thin methods show a better grasp on concepts such as good design and refactoring, and facilitate code re-use. Finally, "mock" methods do not implement the business logic or functionality desired, but instead return the expected values explicitly. These methods are sometimes useful during the development process but should not be delivered in the final product. The degree to which teams implemented fat or thin fixtures was ranked on a subjective scale of 0 (entirely thin) to 10 (entirely fat).

The most significant observation that can be made from Table 3 is that the UofC teams by and large had a much higher fatness when compared to the SAIT teams. This could possibly be explained by commonalities between strategies used at each location. At UofC, teams implemented the test fixtures in advance of any other business logic code (more or less following Test-Driven Development philosophy [9]). Students may not have considered the code written for their fixtures as something which needed to be encapsulated for re-use. This code from the fixtures was further required elsewhere in their project design, but may have been "copy-and-pasted". No refactoring was done on the fixtures in these cases. This can in our opinion be explained by a lack of external motivation for refactoring (such as additional grade points or explicit requirements). Only one team at the UofC took it upon themselves to refactor code without any prompting. Conversely, at SAIT students had already implemented business logic in two previous iterations, and were applying FIT to existing code as it was under development. Therefore, the strategy for refactoring and maintaining code re-use was likely different for SAIT teams. In

summary, acceptance test driven development failed to produce reusable code in this context. Moreover, in general, teams seem to follow a consistent style of development – either tests are all fat or tests are all thin. There was only one exception in which a single team did refactor some tests but not all (see Table 2, UofC T2).

6.2 Strategies for Using Test Suites vs. Single Tests

Regression testing is undoubtedly a valuable practice. The more often tests are executed, the more likely problems are to be found. Executing tests in suites ensures

Table 3. Statistics on fixture fatness and size

Team	Fatness (subjective)		NCSS[2]	
	Min	Max	Min	Max
UofC T1	7	10	28	145
UofC T2	0	9	8	87
UofC T3	8	10	40	109
UofC T4	9	10	34	234
SAIT T1	0	1	7	57
SAIT T2	0	2	22	138
SAIT T3	0	0	24	57
SAIT T4	0	0	15	75
SAIT T5	1	2	45	91
SAIT T6	0	1	13	59

that all test cases are run, rather than just a single test case. This approach implicitly forces developers to do regression testing frequently. Also, running tests as a suite ensures that tests are compatible with each other – it is possible that a test passes on its own but will not pass in combination with others.

In this experiment data on the frequency of test suite vs. single test case executions was gathered. Teams used their own discretion to decide which approach to follow (suites or single tests or both). Several strategies were identified (see Table 4).

Table 4. Possible ramp up strategies

Strategy	Pros	Cons
(*) Exclusively using single tests	- fast execution - enforces baby steps development	- very high risk of breaking other code - lack of test organization
(**) Predominantly using single tests	- fast most of the time execution - occasional use of suites for regression testing	- moderate risk of breaking other code
(***) Relatively equal use of suites and single tests	- low risk of breaking other code - immediate feedback on the quality of the code base - good organization of tests	- slow execution when the suites are large

Exclusively using single tests may render faster execution; however, it does not ensure that other test cases are passing when the specified test passes. Also, it

[2] NCSS is Non-Comment Source Lines of Code, as computed by the JavaNCSS tool: http://www.kclee.de/clemens/java/javancss/

indicates that no test organization took place which may make it harder to manage the test base effectively in the future. Two teams (one from UofC and one from SAIT) followed this approach of single test execution (Table 5). Another two teams used both suites and single tests during the ramp up. A possible advantage of this strategy may be a more rapid feedback on the quality of the entire code base under test. Five out of nine teams followed the strategy of predominantly using single test, but occasionally using suites. This approach provides both organization and infrequent regression testing. Regression testing using suites would conceivably reduce the risk of breaking other code. However, the correlation analysis of our data finds no significant evidence that any one strategy produces fewer failures over the course of the ramp up. The ratio of peaks and valleys (in which failures occurred and then were repaired) over the cumulative test executions fell in the range of 1-8% for all teams. Moreover, even the number of test runs is not deterministic of strategy chosen.

Table 5. Frequency of test suites versus single test case executions during ramp up

Team	Suite Executions	Single Case Executions	Single/Suite Ratio
UofC T1 (***)	650	454	0.70
UofC T2 (***)	314	253	0.80
UofC T3 (**)	169	459	2.72
UofC T4 (*)	0	597	Exclusively Single Cases
SAIT T1 (**)	258	501	1.94
SAIT T2 (**)	314	735	2.40
SAIT T3 (**)	49	160	3.27
SAIT T4 (*)	8	472	59.00
SAIT T5 (**)	47	286	6.09
SAIT T6 (not included due to too few data points).	8	25	3.13

During the regression testing stage we also measured how often suites versus single test cases were executed (Table 6). For UofC teams, we saw a measured difference in how tests were executed after the ramp up. All teams now executed single test cases more than suites. Team 1 and Team 2 previously had executed suites more than single cases, but have moved increasingly away from executing full test suites. This may be due to troubleshooting a few problematic cases, or may be a result of increased deadline pressure. Team 3 vastly increased how often they were running test suites, from less than half the time to about three-quarters of executions being performed in suites. Team 4 who previously had not run any test suites at all, did begin to run tests in an organized suite during the regression period. For SAIT teams we see a radical difference in regression testing strategy: use single test case executions much more than test suites. In fact, the ratios of single cases to suites are so high as to make the UofC teams in retrospect appear to be using these two types of test execution equally. Obviously, even after getting tests to pass initially, SAIT subjects felt it necessary to individually execute far more individual tests than the UofC students. Besides increased deadline pressure, a slow development environment might have caused.

Table 6. Frequency of suites versus single test case executions during regression (post ramp up)

Team	Suite Executions	Single Case Executions	Single/Suite Ratio
UofC T1	540	653	1.21
UofC T2	789	1042	1.32
UofC T3	408	441	1.08
UofC T4	72	204	2.83
SAIT T1	250	4105	16.42
SAIT T2	150	3975	26.50
SAIT T3	78	1624	20.82
SAIT T4	81	2477	30.58
SAIT T5	16	795	49.69
SAIT T6	31	754	24.32

6.3 Development Approaches

The analysis of ramp up data demonstrates that all teams likely followed a similar development approach. Initially, no tests were passing. As tests are continued to be executed, more and more of the assertions pass. This exhibits the iterative nature of the development. We can infer from this pattern that features were being added incrementally to the system (Figure 2, left). Another approach could have included many assertions initially passing followed by many valleys during refactoring. That would illustrate a mock-up method in which values were faked to get an assertion to pass and then replaced at a later time (Figure 2, right).

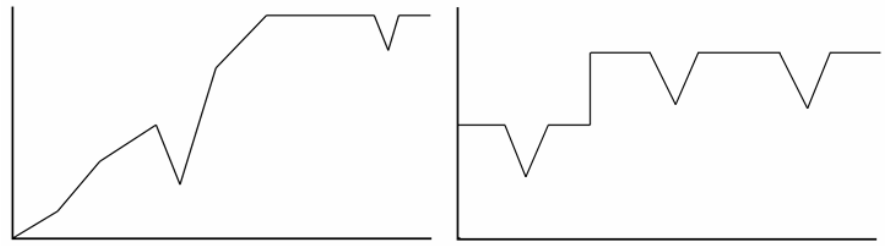

Fig. 2. A pattern of what incremental development might look like (left) versus what mocking and refactoring might look like (right)

Noticeably, there were very few peaks and valleys[3] during development (Table 7). A valley is measured when the number of passing assertions actually goes down from a number previously recorded. Such an event would indicate code has broken or an error has occurred. These results would indicate that in most cases as features and tests were added, they either worked right away or did not break previously passing tests. In our opinion, this is an indication that because the tests were specified upfront, they were driving the design of the project. Because subjects always had these tests in mind and were able to refer to them frequently, they were more quality conscious and developed code with the passing tests being the main criteria of success.

[3] The number of peaks equals the number of valleys. Henceforth we refer only to valleys.

Table 7. Ratio of valleys found versus total assertions executed

Team	"Valleys" vs. Executions in Ramp Up Phase	"Valleys" vs. Executions in Regression Phase
UofC T1	0.03	0.05
UofC T2	0.07	0.10
UofC T3	0.03	0.10
UofC T4	0.01	0.05
SAIT T1	0.06	0.12
SAIT T2	0.03	0.10
SAIT T3	0.04	0.09
SAIT T4	0.05	0.06
SAIT T5	0.05	0.09
SAIT T6	0.03	0.14

6.4 Robustness of the Test Specification

Several errors and omissions were left in the test suite specification delivered to subjects. Participants were able to discover all such errors during development and immediately requested additional information. For example, one team posted on the experience base the following question: *"The acceptance test listed … is not complete (there's a table entry for "enter" but no data associated with that action). Is this a leftover that was meant to be removed, or are we supposed to discover this and turn it into a full fledged test?"* In fact, this was a typo and we were easily able to clarify the requirement in question. Surprisingly, typos or omissions did not seem to affect subjects' ability to deliver working code. This demonstrates that even with errors in the test specification, FIT adequately describes the requirements and makes said errors immediately obvious to the reader.

7 Conclusion

Our observations lead us to the following conclusions. Our hypothesis that no common patterns of ramp up or regression would be found between teams working on different projects in different contexts was only partly substantiated. We did see several patterns exhibited, such as incremental addition of passing assertions and a common use of preferred FIT fixture types. However, we also saw some clear divisions between contexts, such as the relative "fatness" of the fixtures produced being widely disparate. The fixture types students used were limited to the most basic fixture type (TableFixture) and the two fixture types provided for them in examples. This may indicate that rather than seeing a pattern in what fixture types subjects chose, we may need to acknowledge that the learning curve for other fixture types discouraged their use. Subjects did catch all "bugs" or problems in the provided suite of acceptance tests, refuting our hypothesis and demonstrating the potential for implementing fixtures despite problems. Our third hypothesis, that teams would not refactor fixtures to properly delegate operations to business logic classes, was confirmed. In the majority of cases, when there was no motivation to do so students did not refactor their fixture code but instead had the fixtures themselves perform

business operations. Subjects were aware that this was bad practice but only one group took it upon themselves to "do it the right way". Sadly, the part of our subject pool that was doing test-first was most afflicted with "fat" fixtures, while those students who were writing tests for existing code managed by large to reuse that code. In all cases, students used both suites and individual test cases when executing their acceptance tests. However, we did see that each of the groups decided for themselves when to run suites more often than single cases and vice versa. It is possible that these differences were the result of strategic decisions on behalf of the group, but also possible that circumstance or level of experience influenced their decisions.

Our study demonstrated that subjects were able to interpret and implement FIT test specifications without major problems. Teams were able to deliver working code to make tests pass and even catch several bugs in the tests themselves. Given that the projects undertaken are similar to real world business applications, we suggest that lessons learned in this paper are likely to be applicable to an industrial setting. Professional developers are more experienced with design tools and testing concepts, and, therefore, would likely overcome minor challenges with as much success as our subjects (if not more).

References

1. Chau, T., Maurer, F. Tool Support for Inter-Team Learning in Agile Software Organizations. Proc. LSO 2004, Springer, LNCS, Vol. 3096: 98-109, 2004.
2. Cunnigham, W. FIT: Framework for Integrated Test. Online http://fit.c2.com. Last accessed on Jan 15, 2005.
3. Fitnesse. Online http://fitnesse.org. Last accessed on Jan 15, 2005.
4. Melnik, G., Read, K., Maurer, F. Suitability of FIT User Acceptance Tests for Specifying Functional Requirements: Developer Perspective. Proc. XP/Agile Universe 2004, LNCS, Vol. 3134, Springer Verlag: 60–72, 2004.
5. Miller, R., Collins, C. Acceptance Testing. Proc. XPUniverse 2001, July, 2001.
6. Mugridge, R., MacDonald, B., Roop, P. A Customer Test Generator for Web-Based Systems. Proc. XP2003, LNCS, Vol.2675, Springer Verlag: 189-197, 2003.
7. Mugridge, R., Tempero, E. Retrofitting an Acceptance Test Framework for Clarity. Proc. Agile Development Conference 2003, IEEE Press: 92-98, 2003.
8. Steinberg, D. Using Instructor Written Acceptance Tests Using the Fit Framework, Lecture Notes in Computer Science, LNCS, Vol. 2675, Springer Verlag: 378-385, 2003.
9. Test Driven Development. Online http://c2.com/cgi/wiki?TestDrivenDevelopment. Last accessed on Jan 15, 2005.
10. Watt, R., Leigh-Fellows, D. Acceptance Test Driven Planning, Proc.XP/Agile Universe 2004, LNCS, Vol. 3134, Springer Verlag: 43-49, 2004.