Simplifying the Description and Application of Tests

Daniel Brenner and Colin Atkinson

Software Engineering Group
University of Mannheim, 68131 Mannheim
{dbrenner, atkinson}@uni-mannheim.de

Abstract: This paper provides a brief introduction to test sheets, a new approach to test specification and execution that combines the expressiveness of programmatic test definition approaches such as JUnit with the readability of tabular approaches such as Fit using a spreadsheet metaphor.

1 Introduction

Testing has always been one of the key techniques used in software engineering to ensure that software components do what they are expected to do, but its importance has increased in recent years with the rise of test-driven development methods such as extreme programming [Be02]. As well as providing a prescriptive description of how to evaluate whether a component is fit for purpose, in principle tests are also an excellent way of specifying what service a component offers or what requirements a component should fulfill. However, they are rarely used for this purpose today, because the languages and notations used to describe tests are highly implementation-oriented and, thus, are meaningful only to developers. In fact, tests are usually written in the same programming language used to implement the software.

In the Java community, for example, tests are usually written in Java using JUnit. They are very detailed and, without the necessary knowledge about Java and JUnit, are unintelligible to people who are not experts in Java. To make tests useful as a specification technique that is accessible to customers a much more concise and high-level language for defining them is needed. One such approach is Fit [Mu05]. Fit was introduced to give customers the possibility to create tests. The tests are specified in a very high-level and abstract form and do not require any programming skills or any knowledge about the actual details of the software.

Fit has two major disadvantages, however. First, in contrast to JUnit, the user defined Fit test are not immediately executable. Developers need to translate the high-level form into an executable test. Second, Fit is rather limited in its range of functionality, and in particular is not well-suited to defining tests on software abstractions with state (i.e objects). To address this problem we have developed a new test definition approach which we refer to as test sheets. It combines the advantages of high-level approaches like Fit and programmatic approaches like JUnit. In this short paper we give a brief overview of the main ideas behind test sheets.

2 Simple Test Sheets

Figure 1 shows an example of a simple test sheet defining a test case for a simple component representing a bank account. This component exports three operations, deposit(), withdraw(), and getBalance(). The first two take a floating point value as input parameter but return no result, while the latter has no input parameter but returns a floating point value.

	BankAccountTest		A	B
1	BankAccount	create		
2	B1	deposit	32.33	
3	B1	withdraw	20.20	
4	B1	getBalance		12.13

Figure 1: Simple Test Sheet

A test sheet essentially consists of a sequence of method invocations: each row in the table representing a particular operation invocation. It also contains a so-called invocation line which divides the test sheet into two parts. The left-hand side (LHS) holds all the input values and the right-hand side (RHS) shows the expected output values. The number of labeled columns on the LHS is determined by the operation with the highest number of parameters. The number of labeled columns on the RHS is determined by the number of output values supported by the language used to implement the class. Since Java, for example, allows only one return value per method, there is only one column on the RHS. The first row (row 1) in Figure 1 shows the creation of a BankAccount. The first cell in that row identifies which entity is invoked (i.e. the BankAccount) and the next cell identifies which operation of the entity is executed (i.e. the constructor). Since this operation has no parameters the other cells before the invocation line are grayed out.

Test sheets look very much like spreadsheets in popular office suites such a Microsoft Office (Microsoft Excel) and OpenOffice (OpenOffice Calc). A spreadsheet is a matrix of labeled rows and columns. In a test sheet each labeled cell acts like a variable that holds a value. As in programming languages this value can be of a primitive data type or can point to an object. The advantage of the spreadsheet style is that it is possible to refer to each cell by its row and column identifier. This, in turn, makes it possible to reference the values stored in other cells. Thus, the literal value 12.13 in B4 could be replaced by "A2-A3" (see Figure 3), since this is the intended relationship between the values (and is how the value 12.13 is calculated). For example, in row 1 the output from the invocation of the BankAccount's create() operation is stored in the output column (B1) and this value (the pointer to the created instance) can be referenced in subsequent invocations using the label of this cell. Thus the label B1 appearing in the first column of all subsequent row (after row 1) refers to the instance of BankAccount generated by the execution of the first row.

The result columns of the RHS contain the value that the operation is expected to return based on the specified input parameters and the operations executed previously. If an operation delivers no result this column is simply left empty. The default comparison operator is "==". Thus, cell B4 in Figure 1 specifies 12.13 as the expected result [Br08]. This means that the actual result which the operation getBalance() delivers, which is stored in B4, should equal ("==") 12.13. Besides the default ("==") comparison operators "<=", "<", ">", ">=", and "<>" are also allowed. With these operators it is possible to specify almost all required comparisons. If the compared values are of primitive data types, the regular comparison operators will be used. If they are complex data types, the appropriate operations of the associated class will be used. In the case of Java this will be the equals() method. Each comparison can thus conceptually be mapped to an equivalent JUnit assertXYZ() method.

When a fully specified test sheet is applied to the software under test, a new related test sheet is created which displays the result of the test. The columns on the RHS of such "result" test sheets play a special role – they indicate whether the expected value was actually returned using the color coding popularized in JUnit. When the actual returned value matches the expected value the cell is colored green, and when it does not it is colored red and the actual returned value is shown alongside the expected value. **Fehler! Verweisquelle konnte nicht gefunden werden.** show an example of a result test sheet corresponding to the data test sheet in Figure 1.

	BankAccountTest		A	B
1	BankAccount	create		
2	B1	deposit	32.33	
3	B1	withdraw	20.20	
4	B1	getBalance		12.10 | 12.13

Figure 2 Result Test Sheet

3 Higher-Order Test Sheets

The test sheets we have considered so far only contained literal values or, in the case of references, only "internal" references that refer to values inside the same test sheet. This allows abstract relationships to be specified between operation input parameters and result values, but only for a given set of input values. With simple tests of the form just described, the overall test logic cannot be applied to more than one set of input values – that is, it cannot used as the basis for data-driven testing. In data-driven testing, sets of tests (i.e. input data as well as expected output data) are defined and each is executed using the same test logic. This has the effect of injecting the data in each into the each execution of the test logic. Such tests are also called parameterized tests [Ti05].

This can be realized using enhanced forms of test sheets – parameterized test sheets and higher-order test sheets. Parameterized test sheets are like simple data test sheets except that they contain at least one value that is injected into the test sheet when it is executed. Figure 3 is such a parameterized version of the simple bank account test sheet where two values, labeled ?A and ?B, are parameters inject from outside.

BankAccountTest			A	B
1	BankAccount	create		
2	B1	deposit	?A	
3	B1	withdraw	?B	
4	B1	getBalance		A2 - A3

Figure 3: Parameterized Test Sheet

A parameterized test sheet cannot be tested directly but only as a subject of a higher order test sheet. To do this, the parameterized test sheet is regarded as a component (e.g. BankAccountTest in this case) with a single operation, test(), which has an input parameters corresponding to each parameter of the test sheet (in this case ?A and ?B), defined in alphabetical order. In addition the test() operation returns a string which contains a comma separated list of the output cells which would be colored red in a particular execution of the test sheet on a particular set of input data.

BankAccountHOSheet			A	B	C
1	BankAccountTest	test	32.33	20.20	null
2	BankAccountTest	test	...	...	...

Figure 4: Higher Order Test Sheet

Figure 4 shows such a higher order test sheet holding the values. They are basically like ordinary test sheets except they are applied to the conceptual test() operation of the conceptual component derived from a parameterized test sheet. As before the first column contains the row number, followed by the software under test, the invoked operation, the operation's input parameters, and the expected output. There are two specialties here. The first one is that, instead of the name of a component, the name of the lower order test sheets is specified. Each test sheet only has the one operation, test(), which is called in each row with a different set of input value. The second is that the required value for the return string value is stated to be null (i.e. empty) since one usually desires each test (i.e. invocation of the parameterized test sheet) to have no discrepancies. This means that the error string returned from each invocation should be empty (i.e. should identify no red cells). This capability can now easily be combined with the automatic generation of test data to support data-driven testing.

4 Conclusions

Unlike other tabular test definition approaches like Fit, test sheets are intended to be directly executable. We implemented a first test sheet execution environment and we are currently evaluating it. The evaluation results are used to further improve the approach.

References

[Be02] Beck, K.: Test Driven Development: By Example. 2002.

[Mu05] Mugridge, R.; Cunningham, W.: FIT for Developing Software. Framework for Integrated Tests, Robert C. Martin, 2005.

[Br08] Brenner, D.; Atkinson, C.: Software Testing using Test Sheets, ISSTA. 2008 (submitted)

[Ti05] Tillmann, N.; Schulte, W.; Grieskamp, W.: Parameterized Unit Tests. 2005.