

A Collaborative Model for Developers and Testers using the Extreme Programming Methodology

Mark Foulkrod

SilverMark

5511 Capital Center Drive

Suite 510

Raleigh, NC 27606 US

+1 919 858 8300 ext.28

mfoulkrod@silvermark.com

Michael Silverstein

SilverMark, Inc.

5511 Capital Center Drive

Suite 510

Raleigh, NC 27606 US

+1 919 858 8300 ext.29

msilverstein@silvermark.com

ABSTRACT

One of the central tenets of Extreme Programming (XP) is relentless testing throughout the development cycle. In the XP methodology, QA personnel or ‘testers’ are responsible for interacting with the users to create functional tests while developers are responsible for unit testing of classes, subsystems and frameworks (components). This article discusses a model for collaboration between developers and testers where testers augment the developers’ unit testing activities, without adding additional process overhead

Keywords

Extreme Programming, QA, tester, developer, collaboration, test, tools

1 INTRODUCTION

There is no question that a large body of unit tests adds to overall code quality and developer confidence. In XP the “burden” of creating component tests has been placed squarely on the shoulders of developers because of the perception that only a developer can create unit tests. The advantage to this arrangement is that it forces developers to think about how their components are supposed to function, often leading to better code. The disadvantage is that developers may cast a less than critical eye on their code and miss writing robust tests. Writing tests, regardless of the ultimate benefit of having them, is a parallel development activity that detracts from the overall amount of time available for developers to write code.

In XP, the role of testers is to collaborate with users in order to derive and execute functional tests. While a very necessary task, this role often leaves testers less than 100% utilized during the early part of the development cycle while they wait for a user interface to become available. This is often exacerbated by XP’s *Model First* rule, which suggests that developers defer working on the graphical user interface (GUI) as long as possible.

Our goal is to bring testers into the process earlier rather than later, making them an integral part of end-to-end development. Doing so implies moving the responsibility

for creating unit and integration-level tests for some parts of the code from developers to testers. Enabling testers to participate in early component testing tasks:

- decreases the amount of time required to develop the full body of test cases that serve as the gateways for new product features
- increases the amount of time available for developers to create new functionality and refactor code
- makes better use of the QA team earlier in the development cycle
- engages testers in validating the implementation, as well as the user interface
- brings to bear an impartial set of eyes, experienced in the practice of testing, and more critical than the code owners.
- increases the overall efficiency of the team, leading to greater likelihood of project success.

Involving testers in component testing does NOT:

- alleviate developers of all unit testing responsibilities. It simply adds another set of helping hands.
- alleviate testers of their functional testing responsibility
- require that the testers become developers, although it does provide introductory exposure with code. We have seen several occasions of testers whose involvement with unit testing has led them down the development career path.

2 INHIBITORS TO COLLABORATION BETWEEN DEVELOPERS AND QA MEMBERS

Skills

Testers who perform late-cycle testing typically operate on systems from the user interface, an activity that does not normally require programming skills. As evidence of this, testers usually tend to automate tests by recording user

interface interactions, with the addition of light-weight scripting. If we assume that testers do not possess strong programming skills then this raises the question of how one can expect them to create tests that interact directly with code.

The following skills or abilities are required in order to participate in early testing of components:

- A tester must understand the nature of components as service providers. That is, they must understand that components provide services via calls to methods in objects that may or may not require parameters and may or may not return values. This implies the need to understand the mechanics of passing parameters in the form of objects and in the case of languages like Java, understanding concepts such as primitive types and operator overloading.
- A tester must understand the notion of object instantiation and the mechanics of creating object instances and holding them within a fixture.
- A tester must understand how to perform validation on:
 - simple and recursive object state. That is, a tester should be able to describe a strategy of extracting an object's field(s), including those of any referenced objects, as required.
 - returned values, ranging from primitive to complex objects.
 - exceptions that are expected to be thrown as a result calling methods under certain circumstances.
- A tester must understand how to test objects that are distributed on application servers, if that deployment scheme is used. In Java, objects that have been deployed on application servers, such as Enterprise Java Beans (EJBs) require extra effort to test because the test case is a remote client of the bean, and needs to interact with middleware in order to access it.

One possible answer is to train testers to have the minimal programming skills required to develop tests in the required programming language. For testing Java components, implementing a reasonable set of unit tests would require understanding all or most of type casting, primitive/reference type conversions, exception handling, file access, and other Java syntax and language specifics. Unfortunately training, especially to the level required to write component tests, incurs a very high cost

Rapid design evolution

XP is low on up-front design and high on rapid evolution through incremental functionality, refactoring and feedback through testing. Because code is constantly in flux, it

follows that tests need to change rapidly as well. Given this, one might conclude that communication and synchronization overhead between developers and testers as code evolves might add an undue burden on the team.

Test first

An ideal practice would be for testers to develop component tests and then pass them to developers, who would then create code that satisfies those tests. This is certainly a clean separation of labor, but ignores the practicalities of evolutionary design. That is, that the entire set of interfaces for each component would need to be known beforehand, which is not always the way things work out.

Poorly factored code

Poorly factored code helps no one, least of all test developers. Cumbersome interfaces, side effects, private implementations, large numbers of dependencies and high object coupling all contribute to overly complex and convoluted user stories, which make writing tests that much more difficult.

3 ENABLERS TO COLLABORATION BETWEEN DEVELOPERS AND QA MEMBERS

Test first

Despite the above, XP's prescription to create tests first substantiates the practice of involving testers as early as possible, provided that component interfaces and the tests for them are *designed* prior to code implementation. By design, we mean creating a reasonably representative set of user stories from which to implement tests, in much the same way that the code is created to satisfy those same stories. Given the same set of user stories for a component whose interface is clearly defined, it is conceivable that testers could create tests for components prior to their development, or at least in parallel.

Well factored code

If developers have followed good design principles, the interfaces to their components should be as simple as possible and easy to describe in terms of user stories. It follows that if the user stories are simple, the tests that reify them will be simple to implement.

Tools

Tools such as JUnit (Beck[2]) provide a framework for developers to create and execute component tests. JUnit requires programming skills and as such, is geared more for developers than testers according to the definition of testers here. In order to accommodate the tester skill set, we would require some mediating technology that makes it possible for testers to create component tests, without the need to acquire deep programming skills, while still supporting the preference by developers to create tests in the language of the component under test.

4 REQUIREMENTS FOR A COMPONENT TESTING TOOL THAT ENABLES QA AND DEVELOPMENT COLLABORATION

Because we are augmenting the XP process, our view is that we should not significantly change the way developers perform testing, but rather extend XP to include testers where appropriate. Any test tool should reflect the same viewpoint.

In order to promote developer and tester collaboration a test tool should:

- make it possible for testers to describe the services to be tested and what to validate, without having programming skills. In addition, the testing tool should provide a light-weight test definition representation that is rich enough to represent aspects of testing such as stimuli on objects under test, preconditions, postconditions, invariants, expected exceptions, and input data sources, without the need to write test code.
- allow developers to continue to develop simply. The testing tool should appeal to developers' preference for light-weight tools and being able to quickly create test cases as code, preferably in the same programming language that is used for the project. If developers are faced with learning a new scripting language, or creating test cases as anything other than code, it reduces their willingness to participate in continuous testing.
- have built-in test behavior that testers can leverage to create better tests faster. This is important because we do not want to require testers to become bogged down in implementing test-specific infrastructure.
- enable developers and testers to share their automated test assets, such that each group can reference, utilize, and expand upon the other's work. These automated test assets become the common currency of exchange between the two groups. There will be cases where testers create tests to be used by developers and where testers need to integrate more code-intense tests as provided by developers. A tool should promote back-and-forth sharing and reuse.

5 EXAMPLE COMPONENT TESTING TOOL

We at SilverMark have created a component testing tool called "Test Mentor" that embodies many if not all of the above requirements. Although it is a commercial tool, the principles outlined below are general enough to be applicable anywhere.

Test representation

We chose to provide a dual representation for tests. For developers, code is the representation of choice. For testers we developed a representation that enables testers to describe test cases without the need to write code.

Developer test representation

Tests are composed of *suites* at the highest level, which in turn are composed of test assets. Suites are physically represented as classes within a test hierarchy. Test assets are represented by methods.

Integration with JUnit

In order to support developers who are already comfortable with developing tests with JUnit we chose to integrate the JUnit and Test Mentor frameworks, rather than substantially changing the way developers work.

By changing the superclass of `junit.framework.TestCase` to inherit from Test Mentor's hierarchy, all JUnit tests become visible to Test Mentor without the need to change the tests themselves. Test Mentor also provides a means to register JUnit tests as Test Mentor assets available for reuse by testers.

Test representation for testers

The representation of a test for a tester takes one of two forms:

- An XML description of the complete test.
- A visual representation that testers edit through a specialized GUI. The visual representation is saved as Java code, although this is transparent to the tester.

These representations are robust enough to represent instantiation, interaction and validation as described above.

The representation for testers adds the notion of a *test step*. A test step is the unit of greatest granularity and is modeled after the notion of intelligent test artifacts (Silverstein[5]).

Test Mentor provides a number of step types, each providing its own specific behavior such as:

- Calling methods and constructors
- Setting/getting fields
- Iterating over test data
- Referencing test assets (asset reuse)
- Sequencing other test steps
- Running a piece of Java code (script)
- Recursively extracting object state
- Validation (assertion-like behavior)
- Accessing distributed objects such as EJBs or through distribution schemes such as RMI or CORBA.

Testers use the editing tools provided by Test Mentor to organize steps in a particular order and configure steps to operate on target objects.

Test steps are configured through common properties such

as:

- Name and description (documentation)
- Expected exception (class of exception expected to be thrown by executing the step)
- Iterations (number of times to execute the step)
- Performance criterion (maximum time step should take to execute)
- Precondition (step to execute as a precondition. This is similar to the setup method in JUnit).
- Postcondition (step to execute as a postcondition. This is similar to the teardown method in JUnit).
- Failure (step to execute if the owning step fails for any reason).
- Invariant (step to execute, which performs an assertion on some state that is assumed to be true under all circumstances)

Most of the above common properties are optional, which reduces the overhead of step configuration. Individual step types have their own configuration properties.

For example, a *method step* is a type of step that calls a method. A method step is configured by the method name, receiver (usually the object under test, known as a *test subject*) and any parameters. In this case the GUI provides lists of available methods in the class or provides a way for the tester to specify a method that does not yet exist in a class that may or may not yet exist. The following figure shows the part of the tool user interface for configuring a method step:

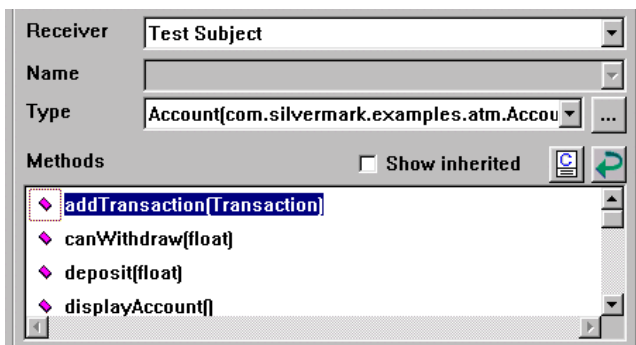


Figure 1 - Configuring a step to call a method

The execution framework takes care of the underlying details of assembling parameters and handling returned values.

To instantiate an object, Test Mentor provides a step similar to the above called a *constructor step*. Users configure constructor steps by selecting a constructor from a list and specifying the required parameters. When a

constructor step is executed, it returns an instance of the object in question.

In the case of remote objects, such as EJBs, access is provided by special steps called *EJB steps* that encapsulate the mechanics of remotely accessing an EJB on an application server. All the tester needs to know in order to test an EJB is the name that the bean has been registered under and its type (interface), as well as certain deployment-specific properties such as application server address and port. From the perspective of a tester, an EJB step performs exactly the same way as a constructor step (it returns an object) and can be used interchangeably. The tool hides the fact that what is actually returned is a stub for a remote object.

Sequences of steps are organized as tests in order to create, apply stimuli and validate the component under test. The following figure shows a very simple test asset (named “Test financialInstrument.Stock”) that creates an instance of a financialInstrument.Stock class, calls several methods and then validates the state of the object.



Figure 2 - A sequence of steps

Finally, test assets may be reused through a type of step called a shortcut step. These steps are configured by specifying the name of a target suite and asset within it. Because assets may be parameterized, and steps can be passed as parameters, there are few limits to the way tests can be composed and reused.

More detail about Test Mentor would be beyond the scope of this paper. To summarize, Test Mentor is an example of a tool that enables developers to create component tests as they always have – as code, and a visual representation that enables testers to define component tests without requiring programming skills with the end goal being collaboration and sharing between the two groups during early testing.

6 COLLABORATION RELATIONSHIP BETWEEN DEVELOPERS AND QA MEMBERS

With skills removed as an inhibitor to early tester involvement, we can then concentrate on the working relationship between developers and testers according to the following guidelines:

- Attach a tester to one or more working pairs of developers in order to collaborate on creating user stories for the components. The testers then use those user stories as input to creating unit tests.
- In creating user stories, the dialog between developers and testers continues to maintain the benefit to developers of forcing them to think about how to test their components just as current XP practices suggest, but under the critical guidance of a tester. This helps developers think like testers, and testers understand component implementation and risks.
- Given a unit test user story the tester is responsible for the mechanics of creating and validating the test, except for cases where a test might require elaborate scripting beyond the abilities of a tester or their tool's test representation scheme.
- Regardless of who creates a unit test, it is always the responsibility of a developer to ensure that their code passes all available tests prior to release into a new build. That is, test execution remains the responsibility of the developer.

7 PRACTICALITIES

We would not be foolish enough to suggest that testers should, or even could create the full body of component tests to validate a nontrivial application.

Tester involvement stands the greatest likelihood of success in areas of the system under development where interfaces are clearly defined ahead of time. Subsystems and frameworks that are provided for the benefit of other subsystems typically fall under this umbrella. They provide a well-defined set of services that can be described by user stories that are not likely to change a great deal from moment to moment. The key is to be able to describe the services provided by the components under test before hand so that developers and testers can create their assets relatively independently.

Tester involvement is less applicable where code interfaces are highly volatile, such as in private or helper classes. These classes tend to spring into existence and evolve rapidly as code is written or refactored, and as such possess interfaces that are difficult to anticipate. Developers will want the flexibility to create and alter code as needed without the communication overhead of notifying testers of changes and additions.

There will be some tests or parts of tests that are not practical for testers to write because they require more expressiveness than the tester or the test representation's capabilities can provide. In cases such as this it is reasonable to expect testers to pass these tests back to the developer. In some cases it is sufficient for a developer to provide one or more code-based utilities for testers to make use of. This is a truly collaborative relationship that benefits both parties.

Project scale

Developer and tester collaboration tends to become more practical as project scope and scale increases. Smaller projects tend to be more inwardly focused, both in terms of component interfaces, and developer perspective. Larger projects tend to be composed of groups of smaller teams that provide subsystems for the consumption of other teams on the project. These subsystems interfaces are perfect candidates for tester involvement.

8 CASE STUDY

Here we describe a particular implementation of the described XP process augmentation.

A company "C" has moved from a traditional development processes toward XP. After successful early product iterations time pressures and perhaps a reluctance to spend time creating tests bore down on the developers and they began to do less unit testing. The QA team, recognizing lower quality levels in the code they were receiving decided to step in to provide testing. The team adopted the following guidelines:

- Each day the development and QA team conducts a standup meeting in which testers and developers coordinate on which tests are going to be needed, in order to validate components. If it turns out that a tester is too loaded with work to be able to supply a test to a developer within the required time period, the developer takes that test development task onto him or herself.
- Once a component is identified and its interface is defined, the developers involved sit down with a tester and define a set of user stories. At this time they perform light design of the tests with the purpose of identifying which tests should be developed by the tester and which ones by developers. In many cases, this dialog identifies design decisions that hamper testability, which may then result in redesign or refactoring of the component in question.
- Developers create unit tests for lower-level and implementation-dependent component functionality. This is typically functionality that is not part of the component's public interface.
- For distributed components such as EJBs, developers are responsible for deploying the components on an application server and notifying the testers of how to access them.
- As the user interface becomes available, testers transition into a more traditional role of validating at that level.

Observed Results

- The quality of component tests, as shown by code

coverage profiling, has improved because of the dialog between testers and developers in designing those tests, and as a consequence, the quality of the code under test.

- Despite their general reluctance to create tests, the developers are perfectly happy to run tests created by testers, as a gateway to code release. An unanticipated but beneficial side-effect is that developers tend to be more inclined to build upon existing tests created by testers, than to create and organize tests completely from scratch.
- Testers have a higher utilization rate, (they are more consistently occupied) throughout the development process.

9 CONCLUSION

Testers can augment developer unit testing activities by acquiring either programming skills, or tools that present a simple test representation.

Tester participation in early testing is most practical where component interfaces are well defined prior to coding.

Allowing testers to wait until a user interface is available to test with may not be as efficient as possible. When testers help developers create unit tests, they speed product delivery, deepen the collaborative atmosphere between developers and testers, and increase the quality of the tests themselves.

ACKNOWLEDGEMENTS

The authors would like to thank the colleagues who provided comments on this paper and especially the clients who provided experience reports.

REFERENCES

1. Beck, K. *Extreme Programming Explained*, Addison-Wesley, 2000; ISBN 0201616416
2. K. Beck, E. Gamma, *Test Infected: Programmers Love Writing Tests*
<http://junit.sourceforge.net/doc/testinfected/testing.htm>
3. Binder, R. *Testing Object-Oriented Systems*, Addison-Wesley, 2000; ISBN 0201809389
4. Jeffries, R. *Model First*
<http://c2.com/cgi/wiki?ModelFirst>
5. Silverstein, M. *Automated testing of Object Oriented-Components Using Intelligent Test Artifacts*, Proceedings of the Thirteenth International Software & Internet Quality Week, May 30 – June 2, 2000