Test-Driven Global Software Development

Bikram Sengupta Vibha Sinha Satish Chandra
IBM India Research Laboratory,
Block 1, Indian Institute of Technology, New Delhi 110016, India

Sharath Sampath K. Guru Prasad
IBM Global Services India Pvt. Ltd.,
Embassy Golf Links Level-3, Bangalore 560071, India
{bsengupt,vibha.sinha,satishchandra,ssampath,guruprasad}@in.ibm.com

Abstract

In a global software development project, distributed teams need to have a consistent view of the system even in the face of frequently changing requirements. Thus how precisely requirements and changes therein are communicated to remote developers becomes a critical issue. In this position paper, we hypothesize that a test-driven methodology may help keep development across multiple sites consistent with changing requirements and with each other.

1 Introduction

According to the iterative model of software development, a project cycle commences with requirements gathering, followed by design, coding and testing; then the next cycle begins. An underlying assumption of this view is that once requirements are collected, they are generally stable through the rest of the cycle. However, the real-world scenario differs considerably. Business cycles are shrinking so rapidly these days that the boundaries between the phases are getting blurred. Very often, by the time developers begin coding, customer requirements are already changing. However, constantly having to change code to meet new requirements is only one half of the problem; in a large project spread across several development teams, a greater challenge lies in ensuring that even as requirements change, a consistent view of the system is maintained across all the teams. Consider for example, a major enhancement request that potentially affects the behavior of many modules; developers then need to clearly understand not only what changes to make in their own module, but also how the behavior of surrounding modules may change. In some cases, the interface agreements may need to be modified; in more subtle cases, there can be behavioral changes in those modules, without any externally visible syntactic changes. Unless there is a shared understanding about these changes, the system may easily slip into an inconsistent state.

In a single-site project, developers usually rely on extensive interactions to clarify doubts regarding requirements and their impact on the behavior of various modules. When the development environment becomes distributed however, face-to-face meetings, if any at all, are few and far in between. There are e-mails, tele-conferences etc. no doubt, but there is a practical limit to their efficacy when it comes to developing a common understanding of the imprecise, ever-changing textual documents that ususally convey requirements. The physical distance and the differences in time-zones make multi-site operation inherently disconnected in nature. Add to this the differences in cultures, languages etc., and it is easy to see why semantic information often do not get across uniformly to remote sites. The resulting discrepancies in understanding introduces delay [5] and may necessitate substantial re-work during integration. Hence there is a need to complement the usual forms of cross-site communication, with practical methodologies that can convey information (in requirements/interface agreements etc.) easily and precisely.

In this position paper, we propose a test-driven methodology to address some of the above challenges in global software development. The basic idea is to create test suites of the different modules prior to development, share them across all the sites and use them as a medium of communication between development teams. For example, changes in these test suites may be used to reflect changes in requirements or in module behavior. The shared understanding that would result from this should help preserve overall consistency.

2 Background

In this section, we (i) briefly discuss some existing approaches to precise behavioral descriptions and (ii) review the notion of test driven development (TDD). We then describe how TDD can be viewed as another precise, although incomplete, form of behavioral specification.

Precise Behavioral Specifications: The software industry has long felt the need for unambiguous specification techniques that developers can use. A number of formal notations have thus been proposed over the years to bring clarity and rigor to software development. These range from mathematical formulations given as algebraic axioms [8] to illustrate the behavior of class methods, to more accessible languages like Eiffel [2], Java Modeling Language (JML) [4], Object Constraint Language (OCL) [6] etc. that are based on the Design By Contract paradigm [1], and use method pre-conditions, post-conditions and class invariants to succintly represent behavior. However, the acceptance of these approaches in the industry has been low in general. Developers are usually unwilling to learn one language for implementation, and another for specification. The complexity associated with these methods may also serve to discourage users and moreover, their technical rigor is often considered an overkill. Finally, they usually do not scale up, and their application in industry-sized projects may be simply infeasible. This points to the need for lightweight but precise specification methodologies that may be used to convey semantic information to developers, and which may be easily integrated with their existing practices.

Test Driven Development: The idea behind Test Driven Development [7] or TDD is simple: before implementing a new functionality, first write executable unit test cases for it. Once you have written enough test cases to thoroughly check the new feature, write the actual code to pass these test cases. The test cases thus become a mini-specification of the functionality that was implemented. This then goes on iteratively as more and more functionalities are added. At the end, one not only has the complete implementation, but also an efficient regression test bed capturing all the new functionalities that were added, and which can be used to identify if subsequent changes break anything in the existing system.

Test Suites as Behavioral Specification: In effect, TDD is a novel approach of creating incomplete but precise specifications on-the-fly during development. Developers have an implicit understanding of what a program is supposed to do, and although they may not be able to specify this behavior formally (e.g. with JML like pre- and post conditions or algebraic axioms), their understanding reflects in the test suites they design. The test suites written prior to development may thus be looked upon as a lightweight specification that guides subsequent implementation. They are obviously incomplete in a formal sense with respect to the full specification, but have several practical advantages: creating test cases requires no new skills from developers, and the specification may be incrementally enriched by adding new test cases as needed. Finally, test suites are unambiguous; from the prespective of global development, this means that a test suite should be interpreted in the same way by different remote development teams.

3 Test-Driven Global Development

We now propose an approach whereby test suites are used as a knowledge sharing medium between remote sites in a distributed development environment. The idea is inspired by the TDD paradigm described above; just as unit test cases written prior to development specify the functionality to be implemented, we believe that early availability of module-wise test suites can serve as a precise documentation of requirements and of module behavior.

Early Test Suites In a typical software development endeavor, once requirements are formulated, some intermediate steps (e.g. use-case diagrams or scenarios) lead to a high level design (modules, interfaces etc.). In a multi-site environment, the modules (e.g. clusters of classes) are then distributed across the remote sites for implementation. The high-level design may be followed by a more detailed design at the remote sites, followed by implementation. Then testing begins, starting with unit testing, to class and module testing, to module integration testing, and finally system level testing.

To adopt TDD in the multi-site context, we propose a simple change to the process outlined above, by suggesting that the end-products of high-level design should not only be modules and interfaces, but also some module-wise test suites jointly created by the system architects and the testing team. These test suites can include unit test cases that illustrate both the normal and exceptional behavior of the public methods, as also functional test cases (e.g. sequences of method calls) that can capture the way a client may use a module. The test cases are associated with the interface, and as such become a first class entity in the design space. At the same time, the interface, usually only syntactic in nature, becomes enriched semantically.

These test cases need not necessarily be fully executable code, but should be precise enough to document the important details e.g. the various input events, corresponding outputs, error-conditions etc. in a proper format. Such artifacts arise naturally as part of testing activities, and executable test cases can subsequently be derived out of them. It may be noted here that, in practice, testing activities sometimes do start before development e.g. test plan documents are often created at the end of high-level design. These activities generally proceed in parallel to development, without contributing to the development effort as such, till the testing phase begins. We feel, however, that test cases may also be looked upon as detailed specification entities, and if these are created upfront and made available to developers, then we can fill a gap between higher level requirements and code. In a sense, test cases are the most precise form of requirements, and their usefulness to developers in deriving requirements understanding has been noted by several practitioners e.g. [3]. Developers work at the code and test case level, so higher level requirements make more sense to a developer if communicated through a medium he/she is familiar with. Hence we also propose mapping the higher-level requirements to these test cases, before coding starts.

Communication through Test Suites To keep distributed development teams in sync with requirements, we next propose that requirements cannot be updated without updates to the associated test cases. Thus for example:

- If a new requirement is added, create test cases for it, and map the requirement to the test cases
- If a requirement is deleted, remove the test cases that have become obsolete
- If a requirement is modified, modify related test cases to clearly reflect this change

These actions have to take place *before* any modification is made to the code. In essence, we change the usual traceability graph originating from the requirements and proceeding through code towards test cases, by having test cases precede source code files instead of following them. Thus, during impact analysis following a requirements change, the test suites have to be updated first. Then these modifications in the test suites guide the developers in changing the source code files.

Another advantage of early module-wise test suites may be in illustrating the behavior of a module, say M, to remote developers who need to use M's functionality. In a distributed environment, the development of the different modules proceeds simultaneously and till M becomes available, a remote developer who needs to use M would write a dummy functionality based on M's interface. Interfaces are primarily syntactic in nature, and are not a rich source of information for someone who wants to use the associated module. However, if we have interfaces enriched with test cases, then a developer, looking at the interface of a remote class, would get a much more clear idea of its behavior and how to use it. This may enable better simulation of M at a remote site, and thus smoothen subsequent integration testing. Moreover, when M's behavior needs to be modified in response to changing requirements, this may once again be conveyed to developers of related modules through changes in the interface test suite. Since test suites are much more precise than text documents, and since they may be made available through a central repository, the need for extensive cross-site communication should decrease, allowing the sites to operate in a relatively disconnected manner.

Our proposal does not, in any way, seek to reduce the importance of conventional post-development testing activities. These should be performed following well-established testing principles, as always. Rigorous testing would definitely require more test cases than can be made available in an early test suite, We believe, however, that if we use test cases only for post-development testing at each site, we make use of their power to validate an implementation, but do not utilize their expressive power. By creating some early test suites, we can not only use them during subsequent post-development testing, but also to convey precise semantic information during actual development.

4 Future Work

We are currently investigating what kind of tool support would be necessary to adopt some of the ideas described above in practice. For future work, we would like to define appropriate metrics to empirically determine the effectiveness of our method in improving multi-site software development.

References

[1] B.Meyer. Design by contract. *Advances in Object-Oriented Software Engineering*, 1991.

[2] B.Meyer. Eiffel: The language. 1991.

[3] E.M.Maximilien and L.Williams. Assessing test-driven development at IBM. *25th International Conference on Software Engineering*, pages 564–569, 2003.

[4] G.T.Leavens, A.L.Baker, and C.Ruby. Jml: A notation for detailed design. *Behaviroal Specifications of Businesses and Systems*, pages Chapter 12, 175 – 188, 1991.

[5] J.D.Herbsleb, A.Mockus, T.A.Finholt, and R.E.Grinter. Distance, dependencies and delay in a global collaboration. *ACM Conference on Computer Supported Collaborative Work*, pages 319–328, 2000.

[6] J.Warmer and A.Kleppe. The object constraint language, precise modeling with UML. 1999.

[7] K.Beck. Test driven development: By example. 2002.

[8] R.Doong and P.Frankl. The astoot approach to testing object-oriented programs. *ACM Transactions on Software Engineering and Methodologies*, 1994.