Bridging Gaps between Developers and Testers in Globally-distributed Software Development

Mark Grechanik
Accenture Labs & UIC
Chicago, IL 60601
drmark@uic.edu

James A. Jones
UC Irvine
Irvine, CA 92697-3440
jajones@ics.uci.edu

Alessandro Orso
Georgia Tech
Atlanta, GA 30332-0765
orso@cc.gatech.edu

André van der Hoek
UC Irvine
Irvine, CA 92697-3440
andre@ics.uci.edu

ABSTRACT

One of the main challenges in distributed development is ensuring effective communication and coordination among the distributed teams. In this context, little attention has been paid so far to coordination in software testing. In distributed development environments, testing is often performed by specialized teams that operate as independent quality assurance centers. The use of these centers can be advantageous from both an economic and a software quality perspective. These benefits, however, are offset by severe difficulties in coordination between testing and software development centers. Test centers operate as isolated silos and have little to no interactions with developers, which can result in multiple problems that lead to poor quality of software. Based on our preliminary investigation, we claim that we need to rethink the way testing is performed in distributed development environments. We then present a possible research agenda that would help address the identified issues and discuss the main challenges involved.

1. INTRODUCTION

Software development is increasingly distributed. In the last two decades, we have observed a clear shift from a purely centralized to a highly distributed software development approach. Many software companies have teams spread all over the globe that produce software through remote collaborations [1]. One of the main challenges in distributed development is ensuring effective communication and coordination among such teams. Failure to do so can result in incorrect assumptions about separately developed components and divergence between the actual and specified behaviors of these components. Researchers and practitioners alike recognize these problems and have started investigating solutions to them (*e.g.,* [12, 18, 22]). Most of the existing coordination methods, processes, and tools, however, tend to focus on requirements gathering, design, and coding activities only, while coordination in testing is mostly overlooked.

In distributed development environments, testing is often performed by *Software Test Centers (STC)*—teams specialized in testing and Quality Assurance (QA) that tend to operate in isolation, with little to no interactions with development teams. In some cases, STCs are completely external to an organization, which is a trend that has become increasingly prominent in the last decade. In such cases, STCs are independent entities that offer specialized testing and QA services to software development organizations. (It is worth noting that test outsourcing is today a $25B marketplace with a 20% yearly growth rate, which makes it the fastest growing segment of the application services market [6].)

While many reasons exist as to why organizations rely on STCs, two are most prevalent. The first reason is an economic one. Grouping testing expertise within one team enables companies to reduce testing costs through the use of workforces from geographic regions with lower labor cost. Moreover, the centralization of testing capabilities in specialized centers can help reduce the overall number of testers for organizations that use the services of these centers. In the case of external STCs in particular, testing outsourcing allows organizations to completely avoid harboring the cost of full-time testing staff and tools. The second reason for using STCs is that the use of specialized testers can establish additional confidence in a software system; in the absence of a close relationship between developers and testers, software is more likely to be tested in an unbiased, thorough way. When software is developed by a third-party consulting organization for a customer, the use of an external STC may even be requested by the customer, who may require QA to be performed by a separate, trusted STC company.

These benefits, however, are often offset by severe difficulties in effectively integrating STCs into the overall software development process, leading to critical shortfalls in how and how well a typical software system is tested. In an informal study of STCs, we have in fact observed the occurrence of a variety of problems caused by the lack of communication and coordination between STCs and development teams. For instance, developers often complain that the lack of insight into the testing results is causing inefficiencies in their development work. Another example is the common case of testers that misclassify features as faults and report them to developers, which also causes inefficiencies on both sides.

Combined, all these issues led to test practices that were supposed to work well on paper, but were not nearly as efficient and effective as expected in practice. In fact, in many cases it was questionable not only whether the anticipated economic and quality benefits expected from the use of STCs were indeed achieved, but even whether the final result was not actually to pay a higher cost for a lower quality software.

In this paper, we discuss the issues related to the use of STCs in distributed development environments in detail to motivate our claim that the way we perform testing in such environments today is highly inadequate. Using the results of our preliminary investi-

FoSER 2010, November 7–8, 2010, Santa Fe, New Mexico, USA.

gation, we show how the lack of coordination between the different parts of a distributed organization, and in particular between STCs and developers, is hindering the production of high quality software. It is important to notice, however, that the solution to these problems is not to eliminate the independence of development and verification teams, whose importance is universally recognized. We therefore propose a research agenda that can help preserve such independence while helping to address the shortcomings of existing approaches to testing in distributed development environments.

The first objective of the proposed agenda is to bridge the gap between software development and software testing teams that are geographically distributed through the use of suitable software engineering tools and practices. Another requirement for the research is to validate the newly defined tools and practices on real testbeds that are truly representative of real scenarios. The agenda also considers educational issues and the problem of better preparing today's students for operating in a globally distributed software engineering context by extending the software engineering curriculum. We believe that this research agenda has the potential to address coordination, communication, and cultural issues between developers and testers, and can ultimately improve the overall quality of the produced software.

2. RESEARCH ISSUES AND CHALLENGES

In this section, we describe the preliminary investigation that we performed and that supports our claims that there is a need for new approaches to testing for distributed development. We start by introducing the scenario for our investigation and illustrating a few relevant examples from this scenario. We then discuss our overall findings and the main issues that we have identified. These issues are the starting point for the definition of the research agenda proposed in the subsequent section of the paper.

2.1 Scenario

In our investigation, we target a scenario that involves a large insurance company whose name we cannot reveal for confidentiality reasons. (Moreover, knowing the name of the company would not change the nature or the validity of our results.) The scenario involves three different kinds of actors: testers, customers, and developers. *Testers* correspond to all of the members of an STC, which include test managers, test automation engineers, and novice testers who create and run integration, system, and acceptance tests (but no unit tests). *Customers* are individuals, groups, and companies that buy software products and perform acceptance testing of this product. Finally, *developers* include project participants who architect, design, and implement software, as well as perform unit testing.

The context for the scenario is the creation of a large-scale software product: a new enterprise claim system that contains more than 7MLOC and whose development took ten years and involved about 700 developers and an STC consisting of 300 testers. Development and testing teams are geographically separated and, as it is typical in these cases, the communication between the STC and the developers is limited.

One important aspect in scenarios of this type is that the different actors have different, and sometimes conflicting goals. Customers are interested in obtaining products with all the needed features, at a low price, and as early as possible. The goal of the testers is to find and report as many bugs as possible in the allotted time and within the allocated budget. Finally, the developers' goals are to produce and deliver the product expected by the customer with as few defects as possible, at the lowest possible cost, and on time.

Another important aspect for the scenario is the interaction between software developers and the STC, which is cyclical. During the implementation of a new release, usually little to no interaction takes place. However, once a release candidate is finalized and shipped to the STC, a period of intense interaction begins. Testing starts, and potential faults are reported to the developers, which in turn consider these reports and implement fixes when needed. At the same time, new revisions of the software are continuously shipped to the test center for re-testing, usually right until a release is made official and becomes available to customers.

In our scenario, and in many large-scale software developments in general, interaction between developers and the STC also depended on the stage of development. During the *rapid creation stage (RCS)*, which produces a few initial prototypes, the main work within the STC is to prepare the scaffolding and infrastructure needed to test the system at hand. Very few to no tests are developed in this stage because system requirements are typically highly unstable and integration and system tests would have to be changed constantly. The next stage is the *stabilizing functionality stage (SFS)*, whose main goal is to satisfy the set of use cases collected in the previous stage through prototyping. In this stage, the STC is increasingly used, as releases are more stable, and longer-lasting tests can be developed. Finally, the *Maintenance and Evolution Stage (MES)* is a stage in which little new functionality is added, and the software is mainly modified to fix existing bugs and implement small functionality extensions. In this stage, STCs are used extensively to perform regression testing and, eventually, acceptance testing.

2.2 Issues

The scenario we just presented demonstrated a number of challenges for performing testing in a distributed software development environment. In this section, we describe the main issues that we identified among testers, developers, and customers. Note that, although we describe these issues independently, in practice they are often intertwined—multiple issues occur simultaneously and further complicate the software development.

1. In RCS, although few/no test cases are produced, it is important that STCs are kept up to date on the evolution of the specs, which does not always happen. As a consequence, they may develop wrong infrastructure.
2. Developers are typically under pressure to deliver a working product and perform little to no unit testing, especially when the code must be delivered to an external testing organization by a due date. As a consequence, the code often contains many shallow bugs that waste the STC's resources and can prevent the discovery of more relevant or subtle bugs.
3. As developers continue modifying existing components, it is important to notify the STC in time about these modifications, so that testers can adjust existing tests accordingly. Unfortunately, this also rarely happens, mainly for scalability reasons. Consider a situation where several hundreds of developers modify existing code that affects test cases that are designed and used by a similar number of testers. Simply sending testers information on each low-level modification to the source code could easily result in millions of notification per day. Moreover, most of these notifications would go to testers who do not actually work on the code involved in the modification.
4. In RCS and early SFS, communication is almost entirely unidirectional from developers to testers: developers produce code and send it to testers for testing. However, in later phases of SFS, it is important to have a two-way communication channel between developers and testers. Not only should testers receive early messages about changes in source code that may break existing test cases, but also developers should be notified about the

(relevant) results of testing. A challenge we observed in this context is the difficulty in identifying the developer(s) who worked on the code where bugs were found, and to provide them with enough information to reproduce the failure without assembling a testbed.

5. During SFS and MES, when new progressive or corrective changes are made to the software, the cooperation between developers and testers becomes more active, cyclical, and needs to be more immediate. Test suites that require long-running times, manual set-up, or human interaction considerably slows down the process of bug finding and fixing.
6. The development organization (and often the STC as well) has little insight into how thoroughly a software system has been, is being, and should be tested. Moreover, with development schedules typically slipping, actual testing time is significantly squeezed. This limited time should be used as effectively as possible, whereas we observed that often set-up time accounts for as much as 40% of an STC effort. Classical questions of test coverage and test effectiveness arise in this case, but they must now be addressed in the context of two organizations that independently perform large amounts of parallel work, with intricate and changing dependencies crossing team/organizational boundaries.
7. Features added, modified, or removed by the development organization are often erroneously reported as faults by the STC. Tests must be up-to-date with the functionality of the system, which is often problematic in this context. First, little communication takes place between developers and STC as to what changed in the system. Second, because many test cases are kept as simple textual instructions for testers (*e.g.,* in a spreadsheet), updating them is a tedious and error-prone process that involves the translation and mapping of feature changes to these instructions (and that often simply does not happen).
8. Because development centers and STCs typically reside in different geographical locations, often many time zones away, implicit and explicit boundaries arise between them [7, 8]. Moreover, cultural differences exist, practices vary, and it is generally difficult to identify the right interlocutor on the other side of the center boundary. As a result, communication that should take place may incur costly delays or may not take place at all.

2.3 Limitations of Existing Testing Approaches

The issues that we have described are not all new, and some of them occur also in more traditional testing contexts (albeit to a smaller degree). It is therefore not surprising that a variety of testing strategies have emerged to try to address these issues. Unfortunately, these existing strategies often do not work in this context, as we now briefly discuss.

Source code sharing. There is a common belief that simply sharing the source code between developers and STCs could address most issues of communication and coordination. While this may be true in some cases, it is not a general solution. First, purely looking at the source code is not enough to understand changes, their effects, and the reasons behind them, especially for large and complex software. Second, in the case of external STCs, sharing source code is simply not an option due to trust issues and often contractual requirements between the customer and the development organization.

Use of continuous testing. A common strategy to address some of the issues we have highlighted is to use continuous integration [2, 9, 16, 23], which involves immediately integrating changes into the main system, continuously updating tests, and running all (old and new) test cases immediately to verify that changes did not break the code base. It is in general infeasible to implement this kind of approach in our context; every change that results in a regression error would require interactions between the developers and the STC, which would reintroduce many of the problems listed above.

Addition of more testers. Despite the adage "given enough eyeballs, all bugs are shallow," the simple addition of more testers is generally insufficient to find all bugs in a system. In particular, some bugs only manifest as failures after removal of other bugs, which means that the debugging process cannot be fully parallelized [11]. The prevalence of this phenomenon was confirmed during our discussions with test managers and engineers, in which they described many situations where shallow, relatively easy-to-find faults caused most of the failures observed early and prevented testing progress until fixes were made.

3. RESEARCH AGENDA

We argue that new approaches must be developed to suitably account for the issues in testing within distributed development environments that we discussed in Section 2.2. We discuss an initial set of relevant research directions that target such issues and can lead to the development of such approaches.

Traceability among different software artifacts. In order to determine what high-level requirements are tested, different software artifacts should be traced to one another. For instance, traceability techniques and methods allow stakeholders to partially address Issue 1 by connecting tests with source code, elements of models, and requirements, thereby maintaining continuous links that can be used to assess change propagation through software.

Precise detection of semantic interference. As we discussed in Issue 1, RCS is characterized by many developers updating code concurrently, thus potentially interfering with one another. Detecting semantic interference with a high degree of automation and precision is vital to resolving latent problems that are introduced by concurrent changes made to software artifacts that depend on one another [21].

Aggregation/distribution of change information for scalability. When hundreds of programmers make low-level changes to the source code, change information should be (1) clustered into higher-level change notifications and (2) sent only to the relevant testers (see Issue 3). For instance, *formal concept analysis* is a principled way of automatically deriving an ontology from a collection of events with properties that could be used to compactly describe code modifications. Also, publish-subscribe systems could be used for an effective distribution of change information [4].

Predicting bugs via mining of changes. As programmers commit their changes to the source code, the information provided earlier by testers in bug-tracking system could be used to identify and flag potentially problematic changes. Doing so would aid in addressing Issue 3.

Regression test selection and prioritization. Regression testing is typically an expensive activity, due to the cost of rerunning potentially large test suites every time the code is changed. Moreover, ensuring that the test suite is adequate for the changed code is challenging [10]. Test selection, augmentation, prioritization,

and minimization techniques that can work in a distributed development environment, especially in cases where the source code is not available to testers, can help addressing Issue 5.

Achieving test coverage faster. Test coverage is a commonly used software quality metric [24] because it is an objective, albeit not perfect [14, page 181], indicator of testing thoroughness. Specifically, achieving higher test coverage is correlated with the probability of detecting more defects [3, 15, 19, 20] and increasing software reliability [5, 17]. The faster testers reach a given coverage goal, the lower is the cost of testing [13], as testers can concentrate sooner on other aspects of testing (*e.g.,* performance and functional testing). Developing approaches for achieving higher test coverage faster would therefore improve the effectiveness of an STC and of the overall distributed development process and help to address Issue 6.

Test partitioning. Large-scale applications have millions of test input data that are used to create tests. Many of these input data will lead to the same behavior of the application, resulting in many hours of test time lost and reduced test effectiveness. It is important to have algorithms and techniques that determine how changes that are made by programmers may lead to effective partitioning of test data, thus addressing Issues 5 and 6.

Test repair and reuse. Test engineers routinely create test scripts to automate the testing process. These test scripts interact with applications by calling methods of their interfaces. The extra effort that test engineers put in writing such scripts is amortized over multiple test runs. Unfortunately, releasing new versions of applications with modified interfaces breaks the corresponding test scripts, thereby decreasing the benefits of test automation. Developing techniques and algorithms for automated test repair would help address Issue 7.

Coordination and awareness. There is a need for approaches that can raise awareness of past and ongoing activities/results among the different actors. This is especially important in this context, which is characterized by rapidly changing conditions that result from a large amount of parallel, highly interrelated work. For instance, we envision individual testers being provided with early warnings of code changes that impact their test cases, so they can appropriately prepare before a new revision is delivered. Change information could also be used to identify parts of the system that need extra attention. We believe awareness to be a key factor in addressing many of the issues in Section 2.2.

Economics of collaborative software development. Traditional software cost models are based on the assumption that everyone involved in a software project is driven to make it successful and agrees on the goals and methods to achieve success. However, different team participants view the ultimate success of the project differently based on their personal goals. This is especially true in contexts that involve actors from different organizations, as it is often the case in distributed development. We believe that new sophisticated economic models are required to analyze software projects as noncooperative games to (1) uncover hidden causes of failures of software projects and (2) suggest ways to fix them. A careful investigation of these economic factors of new software development models will be critical for the success of highly distributed development practices.

Field studies. It is of paramount importance that approaches, techniques, tools, and practices developed in the context of this research agenda are assessed and evaluated in realistic, when not real, scenarios. In this way, it will be possible to assess how the proposed solutions can affect and benefit software development practices, which would help evaluating them and promote their adoption.

Education. Companies involved in software development are increasingly engaged in a distributed engineering process aimed to acquire specific expertise, human resources, and more cost-effective services. The ability to engage in a such a technically and culturally complex process will be key to these companies' success. To enable such capability, it is crucial to develop an engineering workforce that has an international perspective, has experience collaborating with international partners, and understands the difficulties of distributed software engineering (and has the tools to overcome them). In particular, we need to involve students in truly distributed software engineering projects (*e.g.,* by engaging our international colleagues and developing joint courses with a project component). We believe that such engineering projects will provide all participants with insights on the problems of distributed development and testing, develop the technical skills required to tackle them, and ultimately provide them with a rich cultural and technical experience.

Acknowledgments

This work was supported in part by NSF awards CCF-0916605 and CCF-0725202 to Georgia Tech and NSF awards CCF-1017633 and CCF-0916139 to the University of Illinois at Chicago.

4. CONCLUSION

Through our observations in the field and our preliminary investigation, we have identified a broad set of issues that can be tackled by and benefit from research in several areas of software engineering. We believe that advances in addressing these issues can result in more efficient and effective approaches for distributed software development and testing, which will ultimately lead to better quality software and benefit all users of the software infrastructure.

5. REFERENCES

[1] W. Aspray, F. Mayades, and M. Vardi. *Globalization and Offshoring of Software*. ACM, 2006.

[2] Beck. *Test Driven Development: By Example*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[3] X. Cai and M. R. Lyu. The effect of code coverage on fault detection under different testing. In *Profiles,ICSE 2005 Workshop on Advances in Model-Based Software Testing (A-MOST*, pages 1–7, 2005.

[4] M. Caporuscio, A. Carzaniga, and A. L. Wolf. Design and evaluation of a support service for mobile, wireless publish/subscribe applications. *IEEE Transactions on Software Engineering*, 29(12):1059–1071, Dec. 2003.

[5] M.-H. Chen, M. R. Lyu, and W. E. Wong. An empirical study of the correlation between code coverage and reliability estimation. In *IN PROCEEDINGS OF THE THIRD IEEE INTERNATIONAL SOFTWARE METRICS SYMPOSIUM*, pages 133–141, 1996.

[6] Datamonitor. Application testing services: global market forecast model. Aug. 2007.

[7] J. A. Espinosa, N. Nan, and E. Carmel. Do Gradations of Time Zone Separation Make a Difference in Performance? A First Laboratory Study. In *Proc. of the International Conf. on Global Software Engineering*, pages 12–22, 2007.

[8] J. A. Espinosa and C. Pickering. The effect of time separation on coordination processes and outcomes: A case study. In *Proceedings of the Hawaii International Conference on System Sciences*, 2006.

[9] M. Fowler. Continuous integration. /url-http://martinfowler.com/articles/continuousIntegration.html, May 2006.

[10] M. J. Harrold and A. Orso. Retesting software during development and maintenance. In *Frontiers of Software Maintenance (FoSM 2008)*, pages 99–108, Beijing, China, October 2008.

[11] J. A. Jones, J. F. Bowring, and M. J. Harrold. Debugging in parallel. In *ISSTA '07: Proceedings of the 2007 International Symposium on Software Testing and Analysis*, pages 16–26, New York, NY, USA, 2007. ACM.

[12] J. A. Jones, M. Grechanik, and A. van der Hoek. Enabling and enhancing collaborations between software development organizations and independent test agencies. *Cooperative and Human Aspects on Software Engineering, ICSE Workshop on*, 0:56–59, 2009.

[13] C. Kaner. Software negligence & testing coverage. In *In Proceedings of STAR'96*, Jacksonville, FL, USA, 1996.

[14] C. Kaner, J. Bach, and B. Pettichord. *Lessons Learned in Software Testing*. John Wiley & Sons, Inc., New York, NY, USA, 2001.

[15] Y. W. Kim. Efficient use of code coverage in large-scale software development. In *CASCON '03: Proceedings of the 2003 conference of the Centre for Advanced Studies on Collaborative research*, pages 145–155. IBM Press, 2003.

[16] C. Larman and V. R. Basili. Iterative and incremental development: A brief history. *Computer*, 36:47–56, 2003.

[17] Y. K. Malaiya, M. N. Li, J. M. Bieman, S. Member, S. Member, and R. Karcich. Software reliability growth with test coverage. *IEEE Transactions on Reliability*, 51:420–426, 2002.

[18] A. Mockus and J. Herbsleb. Global software development. *IEEE Software*, 18:16–20, 2001.

[19] A. S. Namin and J. H. Andrews. The influence of size and coverage on test suite effectiveness. In *ISSTA '09: Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 57–68, New York, NY, USA, 2009. ACM.

[20] P. Piwowarski, M. Ohba, and J. Caruso. Coverage measurement experience during function test. In *ICSE '93: Proceedings of the 15th international conference on Software Engineering*, pages 287–301, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[21] R. A. Santelices, M. J. Harrold, and A. Orso. Precisely detecting runtime change interactions for evolving software. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST 2010)*, pages 429–438, Paris, France, April 2010.

[22] A. Sarma, A. V. der Hoek, and D. Redmiles. The coordination pyramid: A perspective on the state of the art in coordination technology. *Computer*, 99, 2010.

[23] L. Williams, E. M. Maximilien, and M. Vouk. Test-driven development as a defect-reduction practice. In *ISSRE '03: Proceedings of the 14th International Symposium on Software Reliability Engineering*, page 34, Washington, DC, USA, 2003. IEEE Computer Society.

[24] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, 1997.