EVALUATION OF TEST-DRIVEN DEVELOPMENT
An Industrial Case Study

Hans Wasmus
AEGEON NV, Service Center Pensioen, AEGEONplein 50
2591 TV Den Haag, The Netherlands

Hans-Gerhard Gross
Software Engineering Research Group (SERG)
Delft University of Technology
Mekelweg 4, 2628 CD Delft, The Netherlands

Keywords: Software development management, Software testing, Software requirements and specification.

Abstract: Test-driven development is a novel software development practice and part of the Extreme Programming paradigm. It is based on the principle that tests should be designed and written for a module iteratively, while the code of the module is devised. This is the opposite of what is usual in current software development methods in which testing is often an afterthought, rather than a primary driving factor of design.
Once applied systematically and continuously, test-driven development is supposed to incorporate requirements changes easier, lead to superior technical solutions in software, result in better and cleaner code, and motivate all stakeholders. We present a development project carried out in a company in which we put those claims to a test. We found that, indeed, some of the claims are valid, but we can also report difficulties with applying the test-driven development approach.

1 INTRODUCTION

Test Driven Development is one of the core practices of extreme programming (XP) and thus a part of the XP methodology (Beck, 2005). It addresses the problems of changing requirements that the "more traditional, heavy-weight, predictive" methods address only inadequately. TDD incorporates new requirements coming from the customer during the lifecycle, and it avoids long requirement elicitation and analysis phases (Beck, 2004; Geras et al., 2004; Janzen and Saiedian, 2005). The motto of TDD, is "clean code that works" (Beck, 2004). We should, therefore, not attempt to get a perfect model for a system, but rather use an iterative development approach to expand the project and its features along the way.

TDD is based on unit tests, and to some extent also on higher-level acceptance tests, as only specification items, evaluating pieces of code to be developed. The main difference between normal unit testing and TDD is the development of the test case before developing the functionality, instead of after. Tests guide the design and development in TDD, and "if you can't write a test for what you are about to code, then you shouldn't even be thinking about coding" (Chaplin, 2001) is one of the basic rules for TDD. In Extreme Programming (XP), this is prescribed as "testing everything that could possibly break" (Jeffries et al., 2000).

Many practitioners say that TDD has several shortcomings or disadvantages such as lack of design (Pancur et al., 2003), problems with applying unit tests, lack of documentation (van Deursen, 2001), reliance on refactoring, dependence on the skills of the programmer (George and Williams, 2004) and difficulties with scalability (Constantine, 2001), which are commonly put forward. However, other people believe that TDD has also numerous advantages over the more traditional, predictive development approaches. A non-exhaustive list of references on experiences with applying TDD is (Müller and Hagner, 2002), (George and Williams, 2003), (Maximilien and Williams, 2003), (Williams et al., 2003), (George and Williams, 2004), and (Erdogmus et al., 2005).

This paper presents an evaluation of the positive aspects that the TDD community is commonly putting forward. We performed this evaluation as part of an industry-scale application development project at EPCOS Inc. (EPCOS AG, 2007). This application, a

Wasmus H. and Gross H. (2007).
EVALUATION OF TEST-DRIVEN DEVELOPMENT - An Industrial Case Study.
In *Proceedings of the Second International Conference on Evaluation of Novel Approaches to Software Engineering* , pages 103-110
DOI: 10.5220/0002584401030110

project forecasting system, was meant as a pilot study to decide whether TDD could be introduced as new development paradigm across software development units at EPCOS.

This article is organized as follows. In Section 2, we list the claims that are commonly put forward by proponents of TDD in favor of their chosen design approach, and we describe how we evaluated these claims. In Section 3, we briefly describe the type of system developed in which we applied TDD as methodology. Section 4 discusses our experiences with applying TDD in the presented case study, and Section 5 summarizes our findings. Finally, Section 6 concludes the paper.

2 CLAIMS OF TDD COMMUNITY AND HOW THEY WERE EVALUATED

TDD can be used anywhere in any project. Proponents of TDD say that it is cost-effective to apply it anywhere in any software project. With minor adaptations, TDD will be suitable for testing databases, interfaces and any other software technologies. We talked to people involved to evaluate this usability claim, and we tried out several aspects for implementing our application. We assessed where and under which circumstances TDD may be used and what could be done in areas where we thought TDD was not so suitable.

TDD handles changes in requirements better than predictive approaches. TDD does not propagate upfront analysis and design that must be amended every time a requirement is changed or added. This should make the development process more suitable for changes, and the iterative approach, that results in working prototypes that can be extended, explicitly invites changes. Moreover, change requests emerging during implementation should have limited impact on the source codes. We assessed this by evaluating how much work needed to be done, considering the expected impact of the change. For each change request, we looked at when the request was issued, and whether this change request could have been detected earlier. Apart from looking at the change request itself, we looked at the impact of the change on the system, e.g., time required to implement the change, compared to the time it would have taken in a later stage of the project.

Developers will be able to look further with TDD. It is also argued, that programmers can come up with better solutions for their code. We assessed the entire life cycle of our project to evaluate this claim. We looked at which solutions were found to solve technical problems on an architectural level and on the code level, and how decisions were made in favor or against the solutions. An important question was "how would an early prototype affect the decisions of the stakeholders, the customers, in particular?"

The TDD process is more enjoyable for the stakeholders. The people involved in the case study were senior developers, new team members and the customers. It was difficult to make an assessment of the "joy" of the people working in the project. We talked to the stakeholders and looked at how well new people integrated in the project. A general observation was that developers appreciated the shortened requirements engineering and analysis cycle, so that they could immediately dive into development. TDD provides positive reinforcement: "a green bar is displayed which means that the test works and the code is ok." Tests are the replacement for specification and documentation (van Deursen, 2001) which takes a huge burden off the developers for producing such documents. TDD proponents argue that documentation is always out-of-date, anyway, and new team members rather start with looking at the code instead of going through outdated and bulky text documents.

Code quality will be higher. Except for delivering a better suited product, supporters of TDD say that the quality of the code will also improve. That is because we can achieve nearly 100% test coverage, during development, so that bugs will be found sooner, decreasing the necessity for workarounds and fixes in the final product. Unfortunately, we could not to draw definite conclusions about our code quality, but we got a general idea of the quality of the code coming out of our TDD approach. Secondly, we used code metrics in order get an idea of some properties, such as code complexity, dependability and size. Unfortunately, we could not to draw definite conclusions about our code quality, but we got a general idea of the quality of the code coming out of our TDD approach.

3 THE EPCOS CASE STUDY

We evaluated the claims put forward at an EPCOS software development unit, applying TDD in a web-

based project forecasting application. EPCOS is one of the largest manufacturers of passive components for electronic devices.

Project forecasting was, initially, not supported through a computer system. Originally, external sales representatives worked together with customers on the design of electronic devices which embed EPCOS' products. They told Marketing which types and volumes of components would be required, and hence, should be produced in the future. All information concerning product forecasting was gathered by Marketing through phone and email. Eventually, they sent out orders to the factories which assessed the requests, planned the production, and estimated the delivery dates. In order to streamline this information flow, the company decided to develop and set up a new web-based forecasting application.

The forecasting application has a typical 3-tier architecture with a client layer, a web interface and a database. The client is used by the sales representatives, with operations concerning project tracking and forecasting. The web service connects the client application database, and provides the business logic and authentication features. The application is used to forecast orders and calculate their budgets. Initially, it was done on a yearly basis, making forecasts inaccurate. The goal was to increase accuracy by increasing the frequency with which representatives update the database. Additionally, tracking of forecasts should be implemented, and access should be possible through a web interface. With the new system, the factories are now able to view and confirm the forecasts placed by the marketing department, and they can already start production before a concrete order is placed. This improves delivery time considerably.

The project was developed on typical PCs, connecting to Windows servers. The servers we used were Windows 2000 Servers running Microsoft .NET 2003 runtime libraries and IIS. The size of the project is, with some 3 person years, rather small. However, the system is typical in terms of size and complexity for this development unit, so that we can use the experiences gathered here, well for future projects. During the project, we applied an iterative development process. We had in total three iterations with a length of two and a half months each. Before this project was begun other developers had already worked on a prototype system for approximately 6 months. The first month of this project was used to gather requirements from the "customers" (from within the company), and understand their problems. The project was staffed with some 2 FTE programmer positions, with 1/2 FTE allocated to testing. The project team comprised two graduates and one experienced programmer/tester.

4 EVALUATION DURING THE CASE STUDY

TDD proposes to test every single atomic piece of implementation in order to facilitate changes. After devising a test, it should be realized and executed as soon as the code to be tested becomes available. Finally, the code is refactored according to the outcome of the test. Tests should be independent from each other, meaning that one code piece should only be assessed by one test, so that when this code piece is changed, only one test has to be adapted. If tests were not independent, we would have to amend all tests that are somehow linked to the tested code when a change imposes a code update. If that was not the case, we would have to maintain a trace table that links code pieces to corresponding test cases. It is, therefore, important to keep in mind that poorly organized tests can impede code changes considerably, which would be exactly the opposite of what we want from using TDD.

Usability claim. The first question to answer is if developers are able to use TDD anywhere in development project, and continue using it. Since there is not a predefined development process enforced, the focus here is on using the TDD as a practice. If it is too much effort to use TDD, developers will eventually let go off the practice which is especially the case in small projects, where mostly no strict standard processes are defined. In order to get an idea of usability of TDD over the lifetime of our project, every time a development phase finished, the changes were committed to the source repository. The commits in the repository have been monitored for six months throughout the development cycle.

Figure 1 and Figure 2 display the commits per task category performed in percent and in absolute numbers, respectively. Month 3 is not a very reliable data point, since it was the beginning of a new iteration within the project. The figures show that Test/Works/Refactored slightly go down making place for more bug fixes. The figures indicate TDD is still adopted by the developers, and more time is spent on fixing bugs. After each new test case written, there was a commit, and the the conclusion is primary based on looking at the time spent for writing tests. We could observe that the time spent on writing tests was still around 50According to the programmers, the charts give a good indication of the issue. During this cycle, the Test/Works/Refactored sequence was omitted selectively. This was due to the type of code, e.g., user interfaces are hard to test. Figure 3 confirms this experience in that the effort spent for test-

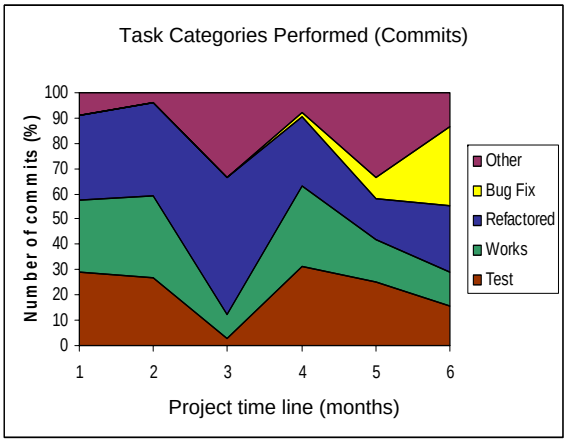

Figure 1: Task categories performed (commits) per month in percent.

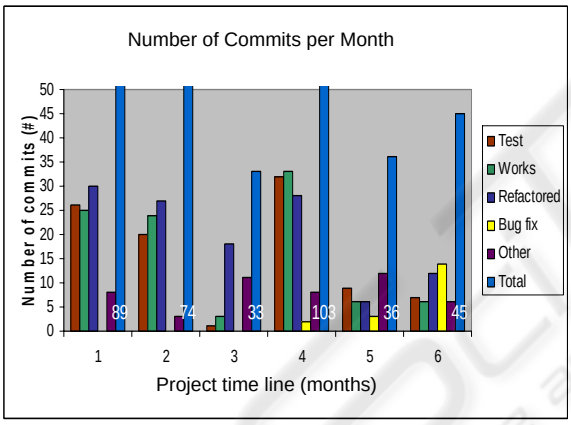

Figure 2: Task categories performed (commits) per month in absolute numbers.

ing went down from 18% to 6% compared to Figure 1. In comparison, the test commits only went down from 29% to 21%, resulting in a lower time per test implementation. This strengthens the idea that, later on in the project, fewer tests are made for those parts of an application that are more difficult to test. Another explanation is that the developers became more experienced with devising tests.

The other two categories, time spent on “fixing bugs” and “miscellaneous” activities, display an expected flow. Time spent for bug fixes is increasing towards the end of the life-cycle. The final months are largely impacted by the ”other” category which

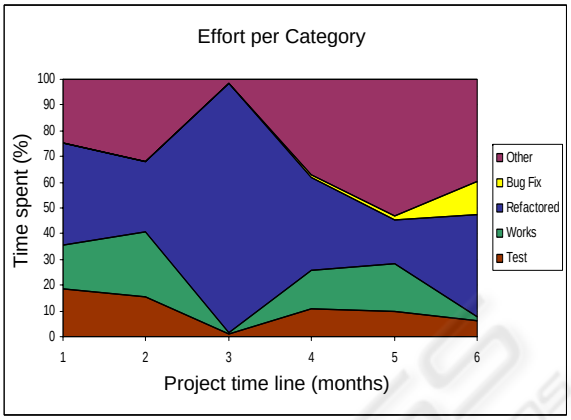

Figure 3: Task categories performed (commits) per month in effort.

includes adding/removal of projects, adding of comments and testing of possible technologies to embed. TDD was used continuously during the duration of the case study. To verify if the decrease of the time spent in Test/Work phases was caused by selective use of TDD by the programmers, the TDD usage was evaluated per application component.

In some cases TDD might not be suitable. We consider TDD to fail, when: (1) it is technically impossible/extremely difficult to test code using unit tests, or when (2) the time required creating a test is not in relation with manual verification of a feature, in respect to the complexity of the implemented feature, e.g. checking a dialog window. Inspecting a window on the screen is straightforward, simply by looking at the screen, but creating a unit test for a display is very difficult (access to the video memory, etc.).

Testing the database. The first step in the case study was the development of the database. According to TDD, we should first develop tests to assess features in the database, and we should test all features, e.g., test to add an unknown customer to a project. This way we can test all foreign and primary keys in the database, and, as a side-effect, generate a database design similar to what a design tool would do. Applying TDD for database design, we could take small steps and got results quickly, which increased programmer satisfaction. However, some problems appeared. Database adapters require making a connection with the database, writing an SQL statement and storing the result in a data set. Then, the result can be compared with the expected result or exceptions

can be analyzed. This is a typical unit test. For the database design, however, this was a time-consuming procedure, because the developers spent some 90% of their time creating test infrastructure for accessing the database, and only the rest of their time went into its design. This is largely caused by the lack of support of C# for testing databases. With a proper tool, the tests could have been created much easier. Therefore, it is advisable to create a simple test interface for the database. If possible, the database interface of the tested application could be used to setup the database connections. It is also helpful to wrap or modify the SqlException objects to throw more specific errors to be used for testing. In order to test security-related issues or error behavior, tests should send faulty information to the database, and to see whether the right SQL exception are generated.

Testing the web service. TDD is very suitable for testing general functions of web services. However, they can only be checked through executing a client connecting to the web service application. We cannot execute the web-service without a client. Using unit tests, we could assess the full range of inputs and outputs including behavior, and the developers found it easy to devise all sorts of different test cases for the web service. We could easily achieve nearly 100% test coverage, and changes were implemented quickly, and the tests amended. Web service development appeared to be an ideal candidate for TDD in our case study.

However, there are some problems concerning testing of security features. Web services use environment variables, such as identification of the client by IP number or network ID. Tests for setting IP numbers are difficult, because they require various networked computers as test setup. Another issue are static variables. Executing tests that change such variables, can damage the web-service leading to unexpected behavior. Operations related to authentication are another issue. Windows user accounts cannot be created and removed easily in a test case, and permissions are difficult to simulate.

A good example for such a problem was a flaw discovered with with multiple open service sessions. The web service opened a new database connection for each user. This worked well with our tests developed according to TDD, and with only one user on the system. However, when two users logged into the system the database connection was reopened and the first user could see data which was only intended for the second user. Our TDD tests were not powerful enough to discover this issue. It was only found through inspection.

From our experience, we believe TDD works well for developing web services, although, only checking function and behavior of the service are not sufficient. Dealing with security issues, such as the concurrent login of users, we have to take environmental issues into account. It is advisable to develop/adapt a small test suite that can be deployed on multiple clients for checking sessions and behavior under concurrent execution.

Testing the client. This is where TDD showed the greatest benefits. Specific technical task are easy to test and easily adapted when changes are requested. A success story in this respect was a request for faster system response. The design available at the time, loaded all the data from the web service when requested. With the accumulation of new projects, this became slower. The goal was to redesign parts of the application, but yet leave the original test cases intact. This can be compared with a maintenance activity and a regression test, albeit during system development. Our existing tests could instantly uncover problems with the new design that could be resolved quickly.

Using TDD for developing the logic of a client application is straight-forward in terms of building suitable test cases. However, the user interface is an entirely different story. It is daunting to build test cases mimicking human interaction with the system. It is very difficult to register, create and catch events in test cases, and it is difficult to check output on the screen. We found a solution by using mock objects based on DotNetMock. Mock objects are fake interface objects that act like real interface objects. The best way to use mock objects for interface testing is to split the interface components according to the Model-View-Controller design pattern (Gamma et al., 1995). This splits an interface into two classes: a controller and a view class. The view class displays the controls, passing on events. The controller class handles the input and output of the view class. The view class can be "faked" using a mock-up class. The application loads the real viewer into the controller, but the test loads a fake viewer (MockObject). In this way, the controller logic can be tested extensively without having to display the interface every time. In our case study, the GUI interfaces was changed a lot, so that having set up a usable testing environment turned out to be a major advantage.

5 SUMMARY OF THE EVALUATION

First claim. It is possible to apply TDD in a software project anywhere where requirements are implemented. With minor adaptations TDD will be suitable for testing databases, interfaces and other technologies.

This is in not true, in general. There are definitely problems with applying TDD everywhere in a project. Some approaches such as aspect oriented programming (AOP) and testing, increase the applicability of TDD, but do not diminish the problem of testing everything there is.

Second claim. TDD handles changes in requirements better than "traditional" development approaches.

Our development process was lead by requirement changes and what was supposed to be a forecasting application, originally, evolved into a project management application, eventually. During development, the stakeholders did not have a concrete idea of what should be in the end product and what shouldn't. New requirements were accommodated while the project evolved. A good example is the development of a feedback plug-in for Microsoft Outlook. Initially, feedback was not even on the requirements list. In the second project iteration, after working with the first release, a feedback panel was added as high priority requirement. Later, this lead to the idea of adding emails to a project and thus creating an e-mail plug-in. When several users believed the application had sufficient features to move from their Excel sheets towards the application, another new requirement was mentioned: an action list panel. Many of the requirements of the final product only appeared through using preliminary versions once people recognized the potential of an undeveloped product. After the first and second iteration, the requirements of the end users got more and more specific and changed several times. It cannot be denied that implementing changes is costly; however, it saved a lot of time implementing changes after the delivery of the application. With TDD, the design of the application is continuously optimized. When changes have to be implemented after the development of the system is completed, they are typically 'hacked' into the design. Eventually, this can turn into a maintenance-nightmare.

Third claim. Developers will be able to look further with TDD, thinking of better solutions while programming.

We found that developers are indeed able to look further with TDD, but only for low level programming problems. The main technologies to be used in the case study were decided in the period from before the implementation until the first weeks of the implementation. Also, the general design was determined in this period. During the implementation, several alternations were made to improve the design or to implement newly discovered technologies. An advantage of TDD is that programmers will know sooner whether the product will satisfy the end user. In this respect, TDD helps to ensure that the customers get the product they want.

Fourth claim. The TDD process is more enjoyable for the stakeholders.

Programmers appreciated the fact that they could quickly jump into developing code, and not having to write lengthy requirements documents, designs, and other technical documentation. However, making many tests and having to change those tests when requirements are added or changed can slow down productivity. Especially when simple changes make many tests fail, it can become frustrating for the developers, becoming reluctant to writing more tests. This would, in fact, mean a failure in the usage of TDD.

For new team members, the lack of a written technical documentation was not an obstacle. The tests provided necessary documentation and all architectural information was supplied by using small paper documents and discussions. Not having a large written documentation was not at all an inhibiting factor during the course of our project.

For a customer, the TDD approach is ideal, according to various interviews held among customers. Customers can get used to a new system, ask for migration tools and propose new requirements. Pricing models requiring payments for each change should be avoided, since this decreases the flexibility and does not impose the delivery of quality on the developers. Instead, payment can be based on usage (e.g. by using licenses).

Fifth claim. Except for delivering a better suited product, supporters of TDD say that the quality of the code will also improve with TDD. Due to creating tests first, nearly 100% test coverage of the code could be achieved at no extra cost (as for a traditional testing phase). There was a working product after each iteration, which was not full of bugs. Moreover, the product could also be shaped according to the customer's wishes, which means that programming and design

time is invested in the requirements most needed by the customer.

To get an estimate of the quality of the code, the final libraries were examined through code analysis tool, called NDepend (NDepend, 2007). It measures several metrics at assembly level and at type level. The metrics at assembly level can indicate how well the design of a project is in terms of maintainability. The type level metrics evaluate the program at a lower level and can detect possible complex classes, that cause trouble. When we compared the results with another project developed based on TDD, the NUnit project, experiences were similar. Compared to the CrystalDecisions library, which is part of the .NET framework, there are noticeable differences, especially for the cyclomatic complexity. By creating a control graph, cyclomatic complexity is an indicator for the complexity of a module (McCabe, 1976). In NUnit and in our case study, the highest complexity in a module is around 90, in the CrystalDecisions module this is more than 650. It must be noted that cyclomatic complexity is just an indicator of complexity; it is arguable whether cyclomatic complexity is an accurate complexity measure (Shepperd, 1988).

Finally, the code coverage of the test in both the client and the web service was 93.5%, which is close to the goal of TDD of 100% code coverage. These figures were produced by nCover (http://ncover.org) that issues the lines of code executed in the unit tests. We have to note that these figures can only count as an indication, not as evidence of code quality: a deeper investigation with more samples is required to be able to speak of evidence.

6 CONCLUSION

From the evaluation of the claims we can conclude that the use of TDD for developing applications has many advantages. Despite the problems caused by test dependencies, the construction of tests during the development process remained steady. The main problem with creating unit tests in our case study was creating tests for everything that should be tested. Sometimes the creation of many tests was simply too time consuming compared to manual verification. This was in particularly the case with testing the database and the user interface. In other cases, it was technically very difficult or even impossible to create a suitable test for a given situation. This is the case with the user interface, build-in features and special technologies, such as AOP techniques.

The biggest advantage of TDD is that a product of high quality can be developed by maintaining flexibility. The case study provided various indications for high quality source code. By creating tests first, we could refactor the design continuously avoiding changes that may have had a negative influence on the system design. This resulted in source code which was easier to maintain. By using iterations and active customer participation, we could identify problems faster, leading to lower effort for refactoring the code. The previous statements help the programmer to have a smooth development process, which adds to programmer's satisfaction. Work satisfaction can be an important ingredient for success. Customer participation also gives the customers the feeling they can shape the application to their needs and give a better understanding of the problems the programmers are facing and the other way around.

REFERENCES

Beck, K. (2004). *Test-Driven Development*. Addison-Wesley.

Beck, K. (2005). *Extreme Programming Explained*. Addison-Wesley.

Chaplin, D. (2001). *Test first programming*. TechZone.

Constantine, L. (2001). Methodological agility. *Software Development*, pages 67–69.

EPCOS AG (2007). Epcos company web site. http://www.epcos.com.

Erdogmus, H., Morisio, M., and Torchiano, M. (2005). On the effectiveness of the test-first approach to programming. *IEEE Transactions on Software Engineering*, 31(3):226–227.

Gamma, E., Helm, R. abd Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

George, B. and Williams, L. (2003). An initial investigation of test driven development in industry. In *Proc. of the 2003 ACM Symposium on Applied Computing*, pages 1135–1139, Melbourne, Florida, USA.

George, B. and Williams, L. (2004). A structured experiment of testdriven development. *Information and Software Technology*, 46(5):337–342.

Geras, A., Smith, M., and Miller, J. (2004). A prototype empirical evaluation of test driven development. In *Proceedings of the 10th International Symposium on Software Metrics*, pages 405–416.

Janzen, D. and Saiedian, H. (2005). Test-driven development concepts, taxonomy, and future direction. *Computer*, 38(9):43–50.

Jeffries, R., Anderson, A., and Hendrickson, C. (2000). *Extreme Programming Installed*. Addison-Wesley.

Maximilien, E. and Williams, L. (2003). Assessing test-driven development at ibm. In *Proc. of the 25th Intl Conference on Software Engineering*, pages 564–569, Portland, Oregon, USA.

McCabe, T. (1976). A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320.

Müller, M. and Hagner, O. (2002). Experiment about test-first programming. *IEE Proceedings Software*, 149(5):131–136.

NDepend (2007). .net code analyzer. www.ndepend.com.

Pancur, M., Ciglaric, M., M., T., and Vidmar, T. (2003). Towards empirical evaluation of test-driven development in a university environment. In *EUROCON 2003, Computer as a Tool, The IEEE Region 8*, volume 2, pages 83–86.

Shepperd, M. (1988). A critique of cyclomatic complexity as a software metric. *Software Engineering Journal*, 3(2):30–36.

van Deursen, A. (2001). Program comprehension risks and opportunities in extreme programming. In *Proceedings of the Eight Working Conference on Reverse Engineering (WCRE'01)*, pages 176–185.

Williams, L., Maximilien, E., and Vouk, M. (2003). Test-driven development as a defect-reduction practice. In *Proc. of the 14th Intl Symposium on Software Reliability Engineering*, pages 34–48, Washington, DC, USA.