Test Driven Development

Erfahrungsbericht

05/2006

Michael Albrecht, Fabian Gutsche, Manfred Wolff

Intro

Agilität ist oft die Antwort auf gängige Projektrisiken: überzogenes Projektbudget, moving targets bei Kundenanforderungen, starre Vorgehensmodelle. Ein Aspekt agiler Vorgehensweisen ist die testgetriebene Entwicklung (TDD test driven development). Nachdem die Autoren sich bereits bei Kent Beck und Erich Gamma infiziert hatten [BG98], wollten sie diese Vorgehensweise auch einmal in einem realen Projekt anwenden. Dieser Praxisbericht beschreibt TDD als Vorgehensmodell sowohl aus der Sicht von Entwicklern als auch von Projektmanagern. Dabei greifen wir auf das (fiktive) Blog eines Entwicklers zu.

TDD

Der folgende Praxisbericht erläutert einige Probleme und Fragestellungen, die bei testgetriebenen Projekten auftreten und erläutert, wie diese bei einem konkreten Softwareprojekt gelöst wurden. Das Projekt hat einen Umfang von ca. 300 BTs und es sind 5 Entwickler involviert. Es ist ein Java EE Projekt im Umfeld von EJB 3.0. Aufgrund der Komplexität des Themas ist sehr früh klar geworden, dass nur eine hohe Testabdeckung Erfolg beschert. Dieses war der Auslöser für eine „Test First“ Entwicklung. Nachdem die einschlägige Literatur studiert war, wurde schnell klar: TDD ist mehr als nur „Test First“.

Bremen, 09. Juni
`Deutschland - Costa Rica 4:2`

Kick Off Meeting
Das neue Projekt soll test driven entwickelt werden. Der Projektleiter ist zuversichtlich, aber unser Geschäftsführer ist wenig begeistert; das Projekt steht sofort unter Druck.

Zunächst erscheint es wenig Unterschied zu machen, ob eine Funktionalität implementiert wird und danach diese per Test auf Korrektheit prüfe, oder ob zunächst der Test spezifiziert wird, um dann die Funktionlität so zu programmieren, dass der Test erfüllt wird. Die Praxis zeigt: Bei der ersten Variante wird die Testabdeckung nicht besonders hoch sein. Zum guten Schluss gibt es immer einen Haufen von Ausreden, warum die Tests nicht mehr implementiert wurden: Zeitmangel, Budgetmangel, der Kunde hat auf Auslieferung gedrängt.
Tatsächlich bedeutet die zweite Variante nicht nur eine höhere Testabdeckung, sondern ein vollständiges Umdenken in der Softwareentwicklung. Während neuere Vorgehensmodelle wie RUP zwar die Abkehr vom Was-

serfallmodell versprachen, Projekte aber dennoch in der Praxis häufig so durchgeführt wurden, ist TDD Agilität pur und die definitive Abkehr von statischen Vorgehensmodellen. Dabei müssen nicht nur Entwickler und Projektmanager umdenken, sondern auch die Geschäftsführung. „Brauchen wir dann nicht doppelt so lange für die Entwicklung der Software“ war die erste Frage, die dem Projektleiter gestellt wurde. Tatsächlich sind auch in den Augen des Managements Tests eher „Zucker“, der bei Bedarf auch reduziert werden kann. Insofern erscheint dem Management eine solche Vorgehensweise erst einmal "nur"teuer. Aber auch das Projektmanagement musste erst einmal überzeugt werden. TDD ist, wie schon bemerkt, keine andere Reihenfolge der Entwicklung, sondern etwas völlig neues und avanciert so automatisch zu einem Risikofaktor im Projekt.

Strategie

Bremen, 12. Juni
`Australien - Japan 3:1`

Mein erstes Mal
Habe gerade Kent Beck fertig. Ich sitze vor dem ersten Test. Meine Aufgabe ist es, XML Daten zu verarbeiten. Wo und wie fange ich an?!

Einer der Hauptaugenmerke bei der strategischen Herangehensweise ist die Entscheidung, mit einer Top-down- oder einer Bottom-up-Strategie zu entwickeln.
Nur zur allgemeinen übereinkunft kurz die Definitionen bezogen auf TDD. Bei der Top-Down-Strategie wird zu Beginn ein Test erstellt, der die Gesamtfunktionalität der Anwendung testet. Die schnelle Erfüllung dieses Tests erfordert eine rudimentäre Implementierung. Der große Gesamttest wird nun unterteilt in (kleinere) Modultests. Diese werden durch Refactoring und Neu-Implementierung des bereits vorliegenden Lösungsansatzes zum Erfolg geführt. Nun wird so analog mit jedem einzelnen Modultest widerfahren, der dann zu noch kleineren funktionalen Tests führt. Es wird also die Funktionalität der Software aus Sicht des Test Designers zerlegt und schrittweise erschlossen.
Die Bottom-Up-Strategie bei TDD sieht vor, dass einzelne Module und Funktionen, die durch Entwurfsüberlegungen ermittelt wurden, mit Hilfe des Dreigestirns Test - Code - Refactoring umgesetzt werden. Hier steckt viel Arbeit und Kommunikationsbedarf im Refactoring. Es stellt sich nach jeder neuen Funktionalität sofort die Frage nach der Weiterverwendung und dem Einbau in bestehenden Quellcode.
Das soll nicht bedeuten, dass das Refactoring bei der Top-down-Strategie

leichter durchzuführen ist. Es gab hier häufig Probleme, die im Abschnitt über die Code Reviews noch angesprochen werden. In unserem durchgeführten TDD-Projekt wurde eine Mischung eingesetzt. Der erste Schritt war Top-Down angesetzt. Die komplette Applikation wurde vorneweg in mehrere Sub-Applikationen zerlegt. Diese einzelnen Module wurden dann den verantwortlichen Entwicklern mit dem Design Contract übergeben und zwar mit dem Auftrag sich nun mit Bottom-Up - Strategie auf die Lösung des Problems zuzubewegen. Auf diese Art und Weise besteht die Möglichkeit, risikoträchtige Implementierungen, die sogenannten „Hot Spots“ sofort anzugehen, und die vorgegebene Aufwandsschätzung zu überprüfen.

Tools

Bremen, 14. Juni
`Deutschland - Polen 1:0`

Prima Tools
Oh man, zum Glück gibt es Tools wie Maven, Eclipse und JUnit. Ohne die ginge gar nichts ...

Testgetriebene Entwicklung komplett ohne Tools durchzuführen, ist machbar, aber nicht wirklich empfehlenswert. Tools nehmen viele lästige oder auch fehlerträchtige Arbeiten automatisiert ab. Hierbei muss man nur an leichte Refactorings denken, wie z.B. eine Methode extrahieren oder eine Variable umbenennen. Schnell passieren kleinere Fehler, die im Nachhinein viel Zeit und Geld kosten. Aus diesem Grund sind entsprechende Tools entwickelt worden.

Wir setzten bei uns standardmäßig Maven [MAV06] als Projektverwaltungstool ein, weil es uns viele Kleinigkeiten rund um die Projektverwaltung abnimmt, angefangen beim Kompilieren des Sourcecodes bis hin zur Erstellung einer Projektseite, die alle wichtigen Informationen über das laufende Projekt enthält.
Gerade durch die testgetriebene Entwicklung und damit der hohen Testabdeckung der geschriebenen Software besitzen wir zu jeder Zeit die Gewissheit, dass der geschriebene Code funktioniert. Dadurch haben wir mittels Maven die Möglichkeit, sehr schnell neue Releases unserer Software an den Kunden zu bringen, ohne dass wir lange von Hand durchgeführte Tests machen müssen.
Vorteilhaft ist bei Maven die Funktionalität, dass keiner der Tasks (wie z.B. Kompilieren des Codes, Erstellen von einer Projektseite oder Erstellen einzelner Reports) ohne die Durchführung der Tests ausgeführt werden kann.

Schlagen während des Testens einige Tests fehl, so wird der eigentliche Task nicht ausgeführt. Der Fehler muss also zuerst behoben werden.
Des Weiteren zieht Maven uns schnell und zuverlässig alle projektabhängigen Bibliotheken in unser Projekt hinein. Dies ist eine große Zeitersparnis und wir können uns auf das Wesentliche, die Entwicklung, konzentrieren.

Auch eine Software wie Eclipse [ecl06] ist für uns bei der testgetriebenen Entwicklung nicht mehr wegzudenken. Da der grundlegende Gedanke bei TDD ja immer Red-Green-Refactor heißt, erleichtern Tools wie Eclipse den Entwicklungsprozess enorm. Nach dem Schreiben eines Tests, lassen wir uns von Eclipse die entsprechenden Klassen oder Methoden per Knopfdruck generieren. Sobald der Test wenigstens kompiliert und rot läuft, fängt man an den Test auszuimplementieren. Dies geschieht auf die verschiedensten Arten und Weisen, wie sie Kent Beck [Bec03] auch schon in seinem Buch beschrieben hat (z.B. Fake It und Obvious Implementation). Nachdem der Test kurze Zeit später grün läuft, kommt der wichtigste Schritt von TDD: das Refactoring. Hierbei sind Tools wie Eclipse als Hilfe unschlagbar. Methoden extrahieren oder Klassen auslagern, alles funktioniert ohne großen Aufwand und man muss sich keine Sorgen machen, dass etwas nach den automatisierten Refactorings nicht funktionieren wird. Das sich der Code genauso verhält wie vorher, können wir leicht überprüfen, in dem wir schnell alle automatisierten Tests durchlaufen lassen.

Damit kommen wir zu der wahrscheinlich wichtigsten Komponenten bei TDD. Um möglichst leicht automatisierte Tests schreiben zu können, ist ein Testframework notwendig. JUnit [JUn06] ist ein solches Framework, das wir mit Erfolg einsetzten. Schnell und einfach ist ein Test mit JUnit geschrieben. Mit einer ganzen Menge von assert-Methoden kann jeglicher Datentyp auf Gleichheit, Ungleichheit oder bei Objekten ebenfalls auf Null abgeprüft werden.
Seit Neuestem setzen wir neben JUnit auch TestNG [Tes06] als weiteres Testframework ein. TestNG funktioniert ähnlich wie JUnit, bietet aber ein paar Vorteile, wie z.B. Testmethoden mit verschiedenen Übergabeparametern in einer bestimmten Reihenfolge oder parallel in verschiedenen Threads durchlaufen zu lassen.
Mit einem Test wird jeweils ein kleiner Abschnitt des Sourcecodes getestet, ob dieser das erwartete Verhalten aufweist. Hierbei kommt es nicht selten vor, dass zunächst eine Methode A alleine durch einen Test geprüft wird. Nachfolgend wird die Methode A von einer Methode B aufgerufen und Methode A muss keine öffentliche Schnittstelle mehr darstellen. So muss der Test für Methode A mit in den Test von Methode B eingebunden werden, damit nur die Schnittstellen öffentlich bleiben, die wirklich als öffentliche Schnittstellen definiert bleiben müssen. Es wäre bei weitem kein gutes Design mehr, wenn Methoden eine andere Sichtbarkeit bekommen, nur damit

diese direkt getestet werden können. In der Regel ist dies aber kein Problem. Wir haben die Erfahrung gemacht, dass dadurch, dass wir die Tests zuerst schreiben, die Schnittstellen automatisch so gestaltet werden, dass alle Komponenten testbar bleiben und die entsprechend benötigte Sichtbarkeit nach dem Refactoring besitzen.
Die Erfahrung hat gezeigt, dass TDD für einen einzelnen Entwickler gut funktioniert. Arbeiten aber zwei oder mehrere Entwickler am gleichen Code, kann der Test nicht mehr als kleine Geschichte ohne Vorgedanken formuliert werden. Hier müssen die weiteren Entwickler sich den bestehenden Code ansehen und versuchen die entsprechenden vorhandenen Methoden zu verwenden, auch wenn diese den bestehenden Code ganz anders aufgezogen hätten. Ansonsten müsste jeder Entwickler das Rad wieder von neuem erfinden. Dies schränkt die Idee ein, komplett unbedarft und ohne sich zuvor Gedanken über die Struktur gemacht zu haben, einen Test zu schreiben. Falls sich der Entwickler keine Gedanken darüber macht, wie sein Test die bestehenden Methoden verwenden kann, wird hier mit Sicherheit doppelter Code entstehen.

Bei den Projekten, die wir durchgeführt haben, haben wir festgestellt, dass die Anwendung von Tools uns das Leben zum einen erheblich erleichtert, und wir dadurch Zeitersparnis haben. Zum anderen führt die Verwendung der Tools leichter zu sicherem Code. Diese Tools stellen auf keinen Fall eine Garantie dafür dar, dass am Ende des Tages auch wirklich sicherer Code erstellt wurde, jedoch erhöhen sie die Wahrscheinlichkeit enorm.

Code Reviews

Bremen, 16. Juni
`Mexiko - Angola 0:0`

Statusmeeting
Der Projektleiter setzt Code Reviews an. Versteht der überhaupt was ich gemacht habe. Bin ja mal gespannt, was da rauskommen soll. Die spinnen die Projektleiter.

Code Reviews sind ein unerläßliches Mittel des Projektmanagements sicherzustellen, dass qualitätsgesicherter Quellcode zum Einsatz kommt. Die erfolgreiche Ausführung der Tests stellt nur sicher, dass das Programm richtig läuft. Über die Qualität des Codes kann keine Aussage getroffen werden.
Umgekehrt sind Code Reviews, bei denen der diskutierte Quelltext nicht umgebaut werden kann, relativ ineffektiv. In unseren Code Reviews arbeiten wir mit Refactoring. Unsere Erfahrung ist es, dass man selten auf den

ersten Schlag gleich die beste Lösung implementiert, sondern vielmehr das mehrfache Arbeiten mit und am Code diesen verbessert. Deshalb empfiehlt es sich aus unserer Sicht, die Verbesserungen sozusagen an Ort und Stelle vorzunehmen.
Refactoring ohne Tests, insbesondere in der Phase des Code Reviews, gehen gar nicht. Metaphorisch entspricht das einer Motorrad-Jungfernfahrt im Porzellanladen.
Alle Regeln des Code Reviewers sind diskussionswürdig und so bleiben solche Auseinandersetzungen für gewöhnlich nicht aus.
Bei unserem Projekt wurde beispielsweise über keine Regel soviel diskutiert, wie über die „80-Zeichen-Regel“, auch bekannt unter dem Checkstyle - Namen LineLength - Check. Dabei treten deutlich Argumentationsmuster hervor, die auch aufzeigen, wie problematisch der Einsatz von Tools werden kann.
Argument 1: Warum eigentlich 80 Zeichen? Mit meinem neuen Bildschirm könnte ich auch 150 Zeichen Textbreite.
Das Problem an diesem Argument ist das besitzanzeigende Pronomen „meinem“. Jeder andere Kollege, der den Quellcode einmal begutachten möchte, um daraus zu lernen, wird wohl keinen ausgiebigen Blick darauf werfen können, ohne zu scrollen.
Bei den Regeln des Code Reviews darf man sich nicht auf einen egozentrischen Standpunkt stellen, sondern muss die Belange der anderen im Blickfeld haben.
Argument 2: Das sieht aus wie ein Fehler, aber der Quellcode kompiliert und läuft doch.
Konsequenterweise und aus Sicht des Code Reviewers ist es auch ein Fehler, wenn die Vorgabe eben 80 Zeichen sind und diese Vorgabe nicht erfüllt wird. In gewisser Weise kann dem Entwickler aber hier entgegengekommen werden. Ein andere Konfiguration ermöglicht den Meldungslevel der Checkstyle-Prüfung auf die Stufe Warnung zu setzen und damit ist der Quellcode auch nicht mehr als fehlerhaft gekennzeichnet.
Argument 3: Der Eclipse Code Formatter macht das nicht richtig, obwohl alles konfiguriert ist.
Das Tool-Argument zieht überhaupt nicht. Erstens sind Tools immer früher oder später unzureichend (TeX einmal ausgenommen ;-)). Zweitens haben wir es mit intelligenten Entwickler und nicht Tool-Bedienern zu tun. Drittens sind die meisten Open-Source-Tools entweder in neueren Versionen erhältlich oder ein Bugfix für das gefundene Problem steht im Netz zur Verfügung.
In unseren derzeitigen Java-Projekten arbeiten wir mit drei Tools:

Checkstyle [Che06] ist ein Tool, dass entwickelt wurde, um den geschriebenen Code auf richtige Formatierung zu überprüfen. Alle Formatierungsregeln können in einer XML-Datei ein oder ausgeschaltet bzw. umkonfiguriert werden.

Mittlerweile validiert Checkstyle nicht mehr nur die Formatierung des Codes, sondern kann auch auf doppelten Code in Klassen überprüfen. Des weiteren wird das Design von den erstellten Klassen analysiert.

PMD [PMD06] validiert den Code auf den angewendeten Programmierstil hin. Dabei werden bekannte Fehlerquellen und -situation gesucht und gefunden.

FindBugs, das wie alle drei Tools, ebenfalls in Java umgesetzt ist, findet gemäß vorher festgelegten Bug Patterns Fehler im Programmablauf. Dazu gehören beispielsweise Vergleiche von Zeichenketten mit den Operatoren == und !=, die meist unerwünscht sind, oder auch der Hinweis auf potentielle NullPointerExceptions.

Mit diesen drei Tools wird jedes unserer Code Reviews vorbereitet. Der Reviewer überprüft die Einhaltung der QS-Vorgaben und nimmt gegebenenfalls Korrekturen direkt vor.
Gegebenenfalls wird der Code durch den Reviewer auch umgebaut, um das Verständnis für den Quellcode zu erhöhen oder Verbesserungen vorzunehmen. Eben dieses Refactoring während des Code Reviews ist nur durch die Testgetriebene Entwicklung möglich. Nur die Sicherheit durch Refactoring - Maßnahmen nichts im korrekten Ablauf des Quellcodes verändert zu haben, weil die Tests immer noch laufen, gibt dem Reviewer alle Freiheit, die er braucht, den Quellcode auf höchstes Qualitätsniveau zu stellen.

Bremen, 19. Juni
`Togo - Schweiz 0:2`

Allmorgendliches Stehmeeting
Ich wusste gar nicht wie viele Kollegen meine Objekte mitbenutzen könnten. Hier entsteht Umbaubedarf. Hoffentlich läuft mein Code

Kurze Status-Meeting im Stehen sind für uns mittlerweile zum täglichen Geschäft geworden. Hier berichtet jeder kurz woran er gerade arbeitet und auf welche Probleme er evtl. gestoßen ist. Sobald allerdings größere Probleme diskutiert werden oder ein Problem nur eine kleine Teilmenge des Team betrifft, wird die Besprechung des längeren Themas auf einen anderen Zeitpunkt verlegt. Das Meeting soll eben nur ein kurzes Status-Meeting sein und nicht jedes Mal in eine lange Konferenz ausarten. Aus diesem Grund findet das Meeting ganz bewusst im Stehen statt. So können sich die Teammitglieder nicht gemütlich irgendwo in einen Stuhl niederlassen, wie es bei längeren Meetings üblich ist. Durch das Stehen wird jeder im Team erinnert, dass er sich kurz fassen soll und nur die wichtigsten Punkte ansprechen darf.
Jedes Teammitglied ist angehalten kurz zu berichten, wenn ein Refactoring durchgeführt wurde, so dass die anderen im Team ebenfalls über die

veränderte Klassenstruktur Bescheid wissen.

Refactoring

Wie bereits erwähnt ist beim Test-Driven-Development der letzte Schritt nachdem ein Test läuft immer das Refactoring. Wird kein Refactoring durchgeführt, warum auch immer, wird der Code sehr schnell schlecht und unstrukturiert. Regelmäßig muss jeder Entwickler prüfen, ob nicht irgendwo doppelter Code vorhanden ist und falls dem so ist, diesen entfernen. Mit Tools wie Eclipse geht das ziemlich leicht und schnell. Allerdings kann einem schon ein wenig mulmig werden, wenn man Refactoring durchführen soll, solange keine Tests vorhanden sind. Woher soll man denn wissen, ob noch alles läuft, sobald das Refactoring durchgeführt ist.
Während wir diesen Artikel geschrieben haben, führten wir in unserem Team ebenfalls ein großes Refactoring durch. Hierbei ging es darum mehrere Ebenen einer Generalisierung auf Generics aus Java 5.0 umzustellen. Da sich während des Refactorings herausstellte, dass noch ein bis zwei Klassen ausgelagert werden konnten, waren wir am Ende absolut nicht mehr sicher, ob der Code wirklich noch laufen würde. Durch die Einführung der Generics sah alles ein wenig anders aus. Wir führten voller Erwartungen unsere Tests aus und siehe da alles funktionierte.
Dies ist das schöne an TDD. Jeder hat durch die hohe Testabdeckung die Gewissheit, dass der Code sich noch so verhält wie er soll oder nicht. Wird ein Refactoring durchgeführt, können sofort alle Tests durchlaufen werden und man sieht, ob Probleme auftreten. Selbst wenn ein Fehler durch das Refactoring in den Tests aufgetreten ist, bedeutet dies nichts negatives, sondern vielmehr kann zielstrebig der Fehler behoben werden. Ohne jegliche Tests bedarf es manchmal langer Such- und Debugarbeit bis der Fehler gefunden wird.

Testabdeckung

Häufig stellt man sich die Frage, wie viel Code die Tests denn jetzt wirklich abdecken? Bei TDD ist der Code nur genau so gut wie die Tests. Eigentlich sollte der geschriebene Code 100% Testabdeckung haben. Warum? Na ja, die Philosophie bei TDD ist, dass nur „echter“ Code geschrieben werden darf, wenn auch ein Test hierfür vorhanden ist. Trotzdem ist man sich nicht wirklich sicher, ob die geschriebenen Tests den kompletten Code abdecken. Des weiteren wird keiner für Getter- und Setter-Methoden Test schreiben. Aus diesen Gründen wird nie eine Testabdeckung von 100% erreicht werden.
Damit man aber trotzdem weiß, wie hoch die Testabdeckung ist, gibt es Tools, die diese ermitteln. Wir wollten eigentlich das Tool JCoverage [JCo06]

dafür verwenden, da es uns in vergangenen Projekten hierbei gut unterstützt hat. Da unser aktuelles Projekt auf Java 5.0 basiert, traten allerdings Probleme mit JCoverage und Generics von Java 5.0 auf. Somit war es uns nicht möglich JCoverage einzusetzen. Es existiert jedoch TPTP [TPT06] von der Eclipse Foundation. Bei diesem Tool treten diese Probleme nicht auf, so dass es uns möglich war, dieses Tool zur Überprüfung der Testabdeckung zu verwenden. Ebenso wie JCoverage kann TPTP u.a. die Tests durchlaufen lassen und erstellt im Nachhinein Reports, die für jede Klasse anzeigen, wie hoch die Testabdeckung wirklich ist und welche Zeilen Code nicht von einem Test durchlaufen wurden. In diesen Codezeilen entdeckt man dann wahrscheinlich noch einen Fehler, nachdem man einen Test dafür geschrieben hat.

Fazit

Zunächst: Der ursprüngliche Ansatz, nämlich eine hohe Testabdeckung zu erreichen, ist erfüllt worden. Das Projekt ist zum jetzigen Zeitpunkt noch im Budget - damit haben sich auch die Managementbefürchtungen nicht bewahrheitet. Die Stehmeetings enden exact nach 15 Minuten, weil die Entwickler gelernt haben sich kurz zu fassen und den Fokus auf das Wesentliche zu lenken. Der größte „Aha“-Effekt ist es aktive Code Reviews vorzunehmen. Statt einer Liste von Defects, die dann vom Entwickler abgearbeitet werden müssen, werden Veränderungen während des Reviews vorgenommen und von Review zu Review wird der Code besser. TDD ist Agilität, die sich in allen Phasen der Softwareentwicklung wiederfindet. TDD ist ein Paradigma, welches sich auch in anderen Projekten durchsetzen sollte. Ob diesee Paradigma skalliert und auch in großen Projekten durchführbar ist, muss sich noch zeigen.

Bremen, 23. Juni
`Saudi-Arabien - Spanien 0:1`

Die Vorrunde ist beendet
Meine Befürchtungen wurden nicht bestätigt. Bin gespannt wie es weitergeht.

Literatur

[Bec03] Kend Beck. *Test-Driven Development by Example.* Addison-Wesley, 2003.

[BG98] Kent Beck and Erich Gamma. Test infected: Programmers love writing tests. 1998. `http://members.pingnet.ch/gamma/junit.htm`.

[Che06] Checkstyle. Homepage, 2006. `http://checkstyle.sourceforge.net`.

[ecl06] Eclipse. Homepage, 2006. `http://www.eclipse.org`.

[JCo06] JCoverage. Homepage, 2006. `http://www.jcoverage.com`.

[JUn06] JUnit. Homepage, 2006. `http://www.junit.org`.

[MAV06] Apache Maven Project. Homepage, 2006. `http://maven.apache.org`.

[PMD06] PMD. Homepage, 2006. `http://pmd.sourceforge.net`.

[Tes06] TestNG. Homepage, 2006. `http://testng.org`.

[TPT06] TPTP. Homepage, 2006. `http://www.eclipse.org/tptp`.