2
Medição e Acompanhamento

Para verificar a eficácia da aplicação da técnica de desenvolvimento dirigido por testes, foram usadas algumas métricas para determinar se houve melhoria ou degradação no processo de desenvolvimento de software após a adoção da técnica. Para garantir resultados confiáveis, as métricas foram balanceadas entre as seguintes características: tempo, escopo, custo e qualidade. Além disso, a abordagem *Goal Question Metric* (GQM) (Basili et al., 1994) foi utilizada para estabelecer quais os objetivos esperava-se atingir para depois definir quais as métricas seriam empregadas.

A abordagem GQM baseia-se na suposição de que, para uma organização fazer medições de maneira adequada, primeiro é preciso especificar quais são os seus objetivos. Depois, mapeá-los para os dados que definirão esses objetivos operacionalmente e, finalmente, oferecer um arcabouço para interpretar os dados de acordo com os objetivos estabelecidos (Basili et al., 1994). O resultado da aplicação da abordagem GQM é a especificação de um sistema de medição visando um conjunto de características e de regras para a interpretação dos dados que foram obtidos (Basili et al., 1994).

Para acompanhar o estado de um projeto de maneira significativa, por exemplo, é preciso identificar o esforço atual e o tempo gasto em cada tarefa e compará-los com o que foi planejado. É impraticável decidir se um produto já está estável o suficiente para ser distribuído sem que se constate qual é a velocidade com que a equipe está encontrando e corrigindo defeitos. Para quantificar se um novo processo de desenvolvimento está sendo vantajoso é preciso alguma medida da performance presente e uma linha de referência para futuras comparações. Por isso, métricas proporcionam um controle melhor sobre os projetos de software e informam mais sobre a forma como a organização está trabalhando (Wiegers, 1999).

Seguindo esse raciocínio, uma lista de objetivos e métricas foi elaborada para avaliar os resultados da aplicação da técnica de desenvolvimento dirigido por testes:

Objetivo: Aprimoramento

Pergunta: As estimativas são feitas com uma precisão aceitável?

Por quê? Essa métrica é importante para avaliar se os desenvolvedores estão conseguindo estimar seu trabalho com uma precisão aceitável. Além disso, permite verificar se a adição de testes está fazendo com que a implementação demore mais do que o esperado.

Métricas:

- Tempo estimado para completar uma tarefa.
- Tempo real para completar a tarefa.

Pergunta: Quantas novas funcionalidades indispensáveis são adicionadas após a fase de planejamento do projeto?

Por quê? Esta métrica identifica problemas na definição de requisitos. Também é importante saber quanto tempo passou até que essas novas funcionalidades fossem percebidas. Será que o uso de desenvolvimento dirigido por testes fez com que essa descoberta acontecesse antecipadamente?

Métricas:

- Quantidade de funcionalidades adicionadas na ferramenta de *issue tracking* durante a fase de execução do projeto.

Pergunta: Os módulos estão bem documentados?

Por quê? Essa métrica averigua se a qualidade da documentação está dentro do esperado.

Métricas:

- Quantidade de interfaces, classes e métodos que não estão documentados.
- Análise subjetiva da semântica da documentação.

Objetivo: Controle

Pergunta: Os projetos são cumpridos no prazo?

Por quê? Em um ambiente profissional real, todas as tarefas têm um prazo para serem cumpridas. Esta métrica permite averiguar se, mesmo com a aplicação

do desenvolvimento dirigido por testes, os prazos estão sendo cumpridos a contento.

Métricas:

- Tempo de realização de cada tarefa.
- Tempo definido para execução do projeto.

Pergunta: Todas as funcionalidades estão sendo atendidas?

Por quê? Essa métrica garante que não só um produto está sendo entregue, mas que este também está completo, de acordo com a análise de requisitos feita posteriormente.

Métricas:

- Funcionalidades planejadas para uma versão.
- Funcionalidades entregues na versão.

Pergunta: O código está sendo completamente testado?

Por quê? Esta métrica garante que testes estão sendo escritos. Isso não significa que um sistema está bem testado, mesmo que o relatório de cobertura indique que 100% do código das classes estão sendo testadas. Porém, se a quantidade de testes só se aplicar a uma fração do código, então é sinal de que existe um problema de teste insuficiente.

Métricas:

- Teste de cobertura dos módulos implementados.

Objetivo: Feedback

Pergunta: É mais demorado desenvolver escrevendo testes primeiro?

Por quê? Esta métrica é importante, pois responde a uma das primeiras perguntas que surgem quando se estuda *Test Driven Development*: "Será que vou demorar mais para programar usando esta técnica?".

Métricas:

- Tempo para implementação e depuração sem utilizar testes.
- Tempo para implementação e depuração utilizando testes.
- Análise subjetiva de tarefas com grau de dificuldade similar.

Pergunta: Quanto está custando realmente o desenvolvimento de software?

Por quê? Adicionar a tarefa de testar o código que é escrito com certeza aumenta o esforço para escrever código. Mas será que isso torna o desenvolvimento de software mais caro no final?

Métricas:

- Horas trabalhadas no projeto.

Pergunta: Qual a quantidade de erros encontrados durante o desenvolvimento e quanto depois da distribuição do produto?

Por quê? Esta métrica é importante porque mostra se produzir testes durante o desenvolvimento de software realmente diminui a quantidade de *bugs* que um sistema possui.

Métricas:

- Número de e-mails com notificações de erro.
- Quantidade de *bugs* adicionados na ferramenta de *bug tracking*.

Pergunta: É fácil ou difícil alterar os módulos que foram feitos utilizando testes?

Por quê? Uma das características do desenvolvimento dirigido por testes é o constante *refactoring* que deve ser feito. Se o custo de fazer *refactorings* for muito grande, então utilizar o desenvolvimento dirigido por testes será um problema.

Métricas:

- Tempo gasto para fazer correções.
- Quantidade de *refactorings* necessários.

Pergunta: Os programadores estão satisfeitos com o uso de desenvolvimento dirigido por testes? Por quê?

Por quê? O desempenho em uma tarefa está diretamente relacionado com a satisfação em realizá-la. Se depois do período de adaptação as pessoas continuarem descontentes com a forma como trabalham, a produção não será eficiente.

Métricas:

- Análise subjetiva das opiniões de cada membro da equipe.

Pergunta: Onde aparecem mais erros? Em módulos testados ou não testados?

Por quê? Este tipo de métrica vai demonstrar se realmente a quantidade de *bugs* nas partes testadas do sistema é menor do que a quantidade de erros em módulos feitos sem testes.

Métricas:

- Quantidade de *bugs* adicionados na ferramenta de *bug tracking* para os módulos específicos.

2.1.Ferramentas

Para que as métricas sejam coletadas de maneira eficiente, algumas ferramentas devem ser utilizadas para facilitar e automatizar a obtenção das medições especificadas. Para que as métricas sejam eficazes é preciso que todos na equipe colaborem adicionando as informações que não podem ser obtidas automaticamente. Este é o caso de informações como o tempo estimado e o realmente gasto para terminar uma determinada tarefa.

2.1.1.JIRA[1]

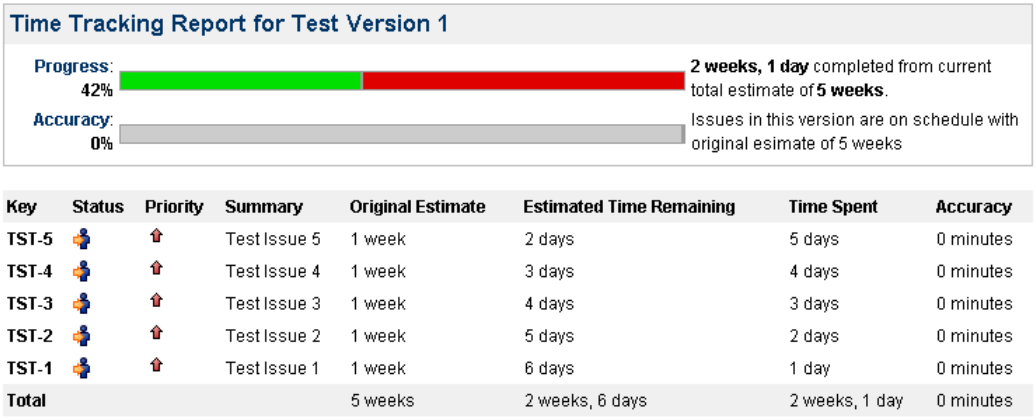
O JIRA é uma ferramenta para gerenciamento de projetos com rastreamento de *bugs* e necessidades. Esta ferramenta também permite estimar quanto tempo será gasto em cada tarefa e registrar quanto tempo foi efetivamente gasto. Esta foi utilizada para controlar diversas informações, como o planejamento de versões, as tarefas e o controle do trabalho feito pelos desenvolvedores, além da gerência dos *bugs* encontrados. JIRA é uma ferramenta comercial, mas possui uma versão gratuita para projetos *open source*. Outras ferramentas gratuitas também poderiam ser utilizadas como Trac[2], GForge[3] e Bugzilla[4].

[1] http://www.atlassian.com/software/jira/

[2] http://trac.edgewall.org/

[3] https://gforge.org/

[4] http://www.bugzilla.org/



Time Tracking Report for Test Version 1

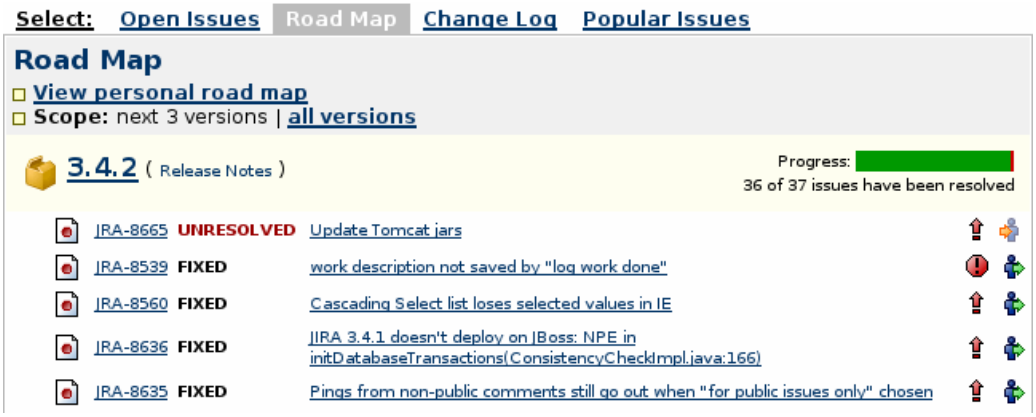
Progress: 42% — 2 weeks, 1 day completed from current total estimate of 5 weeks.

Accuracy: 0% — Issues in this version are on schedule with original esimate of 5 weeks

Key	Status	Priority	Summary	Original Estimate	Estimated Time Remaining	Time Spent	Accuracy
TST-5			Test Issue 5	1 week	2 days	5 days	0 minutes
TST-4			Test Issue 4	1 week	3 days	4 days	0 minutes
TST-3			Test Issue 3	1 week	4 days	3 days	0 minutes
TST-2			Test Issue 2	1 week	5 days	2 days	0 minutes
TST-1			Test Issue 1	1 week	6 days	1 day	0 minutes
Total				5 weeks	2 weeks, 6 days	2 weeks, 1 day	0 minutes

Figura 1: Relatório de planejamento e registro de trabalho.

A Figura 1 apresenta um relatório gerado automaticamente pelo JIRA com o registro de trabalho de cada desenvolvedor. Para que os resultados dessas métricas sejam válidos, é preciso que cada desenvolvedor estime o seu trabalho e registre o tempo que efetivamente consumiu para realizá-lo. A Figura 2 apresenta um resumo das novas funcionalidades e dos erros corrigidos (ou não corrigidos) de uma versão determinada. Para que esse relatório seja preciso, é obrigatório que todas as tarefas realizadas pelos desenvolvedores sejam cadastradas no JIRA.



Select: Open Issues | Road Map | Change Log | Popular Issues

Road Map

View personal road map

Scope: next 3 versions | all versions

3.4.2 (Release Notes)

Progress: 36 of 37 issues have been resolved

JRA-8665	UNRESOLVED	Update Tomcat jars
JRA-8539	FIXED	work description not saved by "log work done"
JRA-8560	FIXED	Cascading Select list loses selected values in IE
JRA-8636	FIXED	JIRA 3.4.1 doesn't deploy on JBoss: NPE in initDatabaseTransactions(ConsistencyCheckImpl.java:166)
JRA-8635	FIXED	Pings from non-public comments still go out when "for public issues only" chosen

Figura 2: Mapeamento de funcionalidades e erros corrigidos em uma versão.

Com relação às métricas de qualidade de software e relacionadas com o código fonte, foram utilizadas três ferramentas: JDepend, JavaNCSS e Cobertura. Essas ferramentas foram escolhidas por fornecerem uma variedade de métricas que podem ser obtidas baseadas apenas no código-fonte. Além disso, todas elas são facilmente integradas com a ferramenta Maven. Dessa forma, durante cada *build* feito pelo Maven, é possível gerar relatórios com as métricas oferecidas. Estes relatórios, em conjunto com o uso da prática de integração contínua, dão feedback constante sobre a qualidade do código em desenvolvimento.

2.1.2.Trac[5]

O Trac é um Wiki[6] aprimorado e um sistema para acompanhamento de pendências no desenvolvimento de software. Esta ferramenta permite relacionar informações que estão no banco de dados de *bugs* com o controle de revisões e o conteúdo do Wiki. Também serve como uma interface *web* sofisticada para o sistema de controle de versões Subversion. O Trac oferece vários relatórios sobre o projeto. Um deles é um esquema de linha do tempo que exibe em ordem os eventos que devem ser realizados e permite acompanhar o progresso de um projeto facilmente.

2.1.3.Bugzilla[7]

O Bugzilla é uma ferramenta para acompanhamento de *bugs* originalmente desenvolvida e usada pelo projeto Mozilla. Liberada como um software *open sorce* pela Netscape em 1998, foi uma das pioneiras nessa área. A noção de *bug* nesta ferramenta é bem genérica. O Bugzilla utiliza o termo *bug* não somente para acompanhar problemas em um programa, mas também para acompanhar a requisição de novas funcionalidades. Os *bugs* podem ser reportados por qualquer pessoa e são atribuídos para um desenvolvedor em particular resolver. Cada *bug* também pode conter notas do usuário e exemplos de como reproduzir o erro.

2.1.4.GForge[8]

O GForge é um software livre para gerenciamento de projetos e colaboração entre a equipe. Foi criado originalmente para o SourceForge, mas foi liberado por um de seus desenvolvedores para o uso aberto. Este possui ferramentas que ajudam a colaboração entre a equipe, como fóruns e listas de e-mail; ferramentas

[5] http://trac.edgewall.org/

[6] Chamado de Wiki por consenso, este é um software colaborativo que permite a edição coletiva de documentos usando um singelo sistema, sem que o conteúdo tenha que ser revisto antes da sua publicação.

[7] http://www.bugzilla.org/

[8] http://gforge.org/

para criar e controlar o acesso aos repositórios de controle de versão, como o CVS e o Subversion; e ferramentas para acompanhamento de *bugs*.

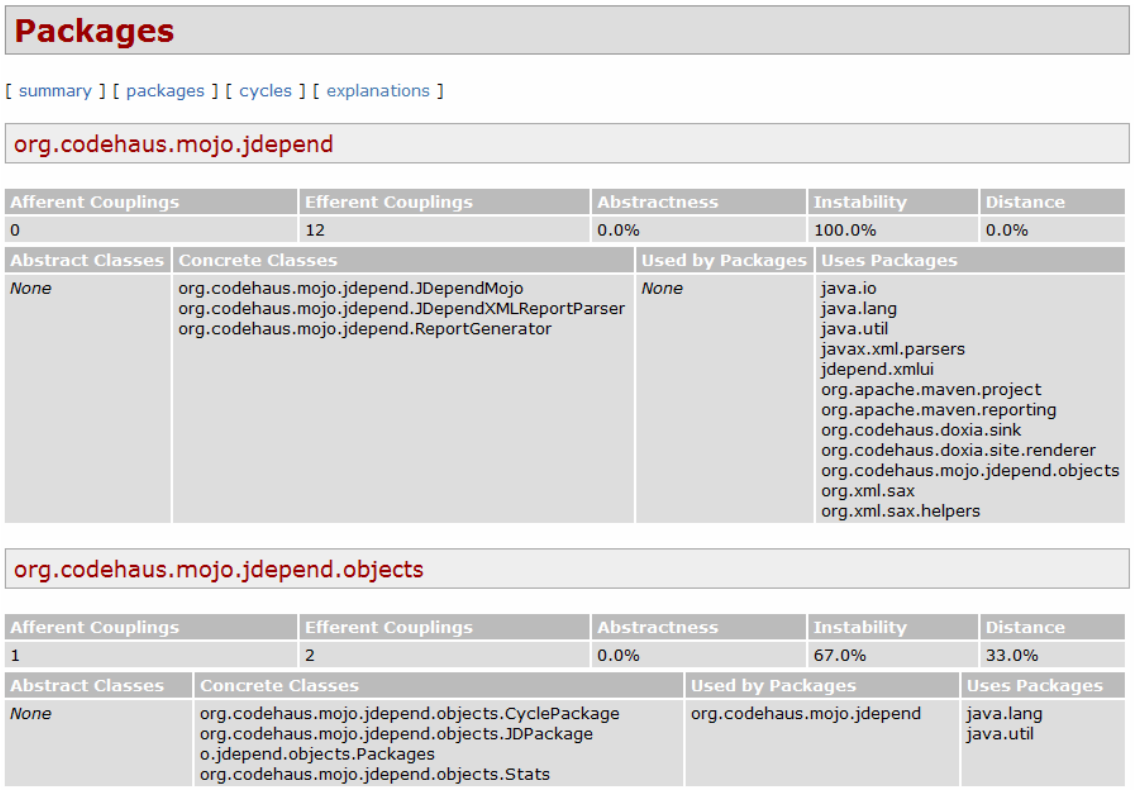
2.1.5. Confluence[9]

O Confluence é uma ferramenta comercial para gerenciamento de informação e colaboração, conhecida como Wiki. Esta ferramenta é gratuita para projetos *open source* e filantrópicos, mas é pago para organizações comerciais e acadêmicas. Uma das vantagens do Confluence é a sua extensibilidade por meio de *plug-ins* e o fato de o código-fonte ser disponibilizado para as pessoas que compram uma licença. Por ser do mesmo fabricante do JIRA, integra-se facilmente com esta outra ferramenta.

Uma ferramenta para colaboração e acompanhamento é fundamental para qualquer equipe de desenvolvimento. A forma como o Wiki funciona facilita a troca de informações entre gerentes de projeto, programadores e clientes. Os dados incluídos nesta ferramenta também podem servir como documentação de um projeto.

[9] http://www.atlassian.com/software/confluence/

2.1.6.JDepend[10]



Packages

[summary] [packages] [cycles] [explanations]

org.codehaus.mojo.jdepend

Afferent Couplings	Efferent Couplings	Abstractness	Instability	Distance
0	12	0.0%	100.0%	0.0%

Abstract Classes	Concrete Classes	Used by Packages	Uses Packages
None	org.codehaus.mojo.jdepend.JDependMojo org.codehaus.mojo.jdepend.JDependXMLReportParser org.codehaus.mojo.jdepend.ReportGenerator	*None*	java.io java.lang java.util javax.xml.parsers jdepend.xmlui org.apache.maven.project org.apache.maven.reporting org.codehaus.doxia.sink org.codehaus.doxia.site.renderer org.codehaus.mojo.jdepend.objects org.xml.sax org.xml.sax.helpers

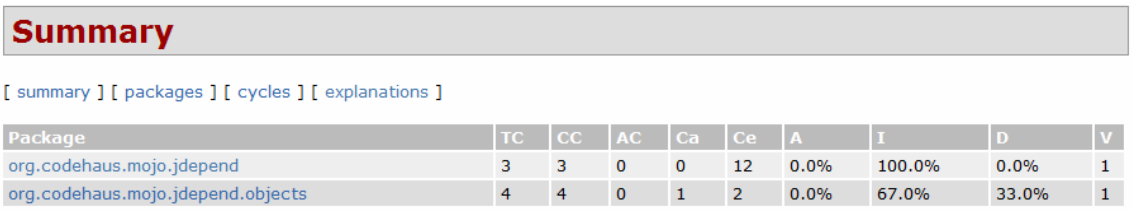
org.codehaus.mojo.jdepend.objects

Afferent Couplings	Efferent Couplings	Abstractness	Instability	Distance
1	2	0.0%	67.0%	33.0%

Abstract Classes	Concrete Classes	Used by Packages	Uses Packages
None	org.codehaus.mojo.jdepend.objects.CyclePackage org.codehaus.mojo.jdepend.objects.JDPackage o.jdepend.objects.Packages org.codehaus.mojo.jdepend.objects.Stats	org.codehaus.mojo.jdepend	java.lang java.util

Figura 3: Resultados obtidos pela análise do JDepend por pacotes.

O JDepend percorre todos os diretórios que contêm classes Java e gera métricas de qualidade do design para cada pacote Java. Esta ferramenta permite que a qualidade de um design seja medida automaticamente em termos de extensibilidade, reusabilidade e manutenibilidade de forma a gerenciar as dependências de pacotes efetivamente.



Summary

[summary] [packages] [cycles] [explanations]

Package	TC	CC	AC	Ca	Ce	A	I	D	V
org.codehaus.mojo.jdepend	3	3	0	0	12	0.0%	100.0%	0.0%	1
org.codehaus.mojo.jdepend.objects	4	4	0	1	2	0.0%	67.0%	33.0%	1

Figura 4: Resumo das métricas feitas pelo JDepend.

A Figura 3 representa um relatório gerado pelo JDepend com as métricas categorizadas por pacote. A Figura 4 corresponde a um relatório resumido com as métricas obtidas pelo JDepend. Os campos da tabela podem ser interpretados da seguinte forma:

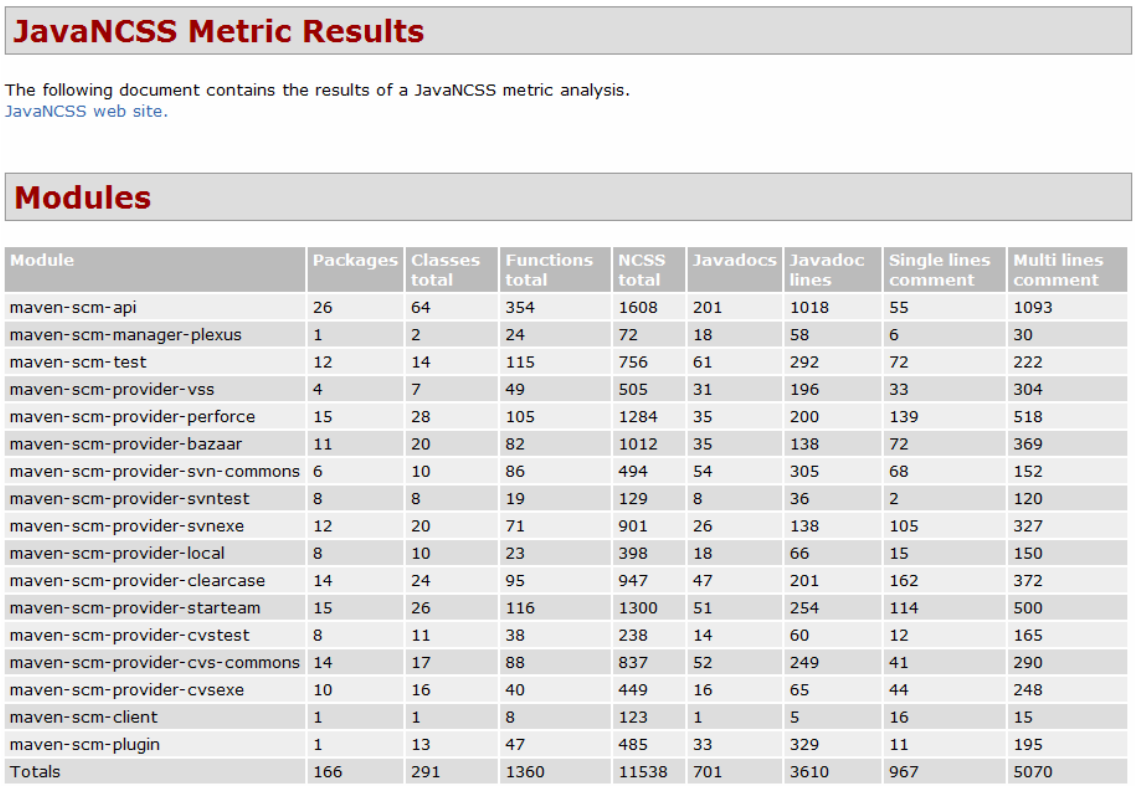
- ***TC***: número total de classes.

[10] http://clarkware.com/software/JDepend.html

- ***CC***: número de classes concretas.
- ***AC***: número de classes abstratas.
- ***Ca***: acoplamento aferente.
- ***Ce***: acoplamento eferente.
- ***A***: nível de abstração.
- ***I***: instabilidade.
- ***D***: distância da seqüência principal.
- ***V***: volatilidade.

Maiores informações sobre o significado de cada um desses valores pode ser encontrado no site do JDepend.

2.1.7.JavaNCSS[11]



JavaNCSS Metric Results

The following document contains the results of a JavaNCSS metric analysis.
JavaNCSS web site.

Modules

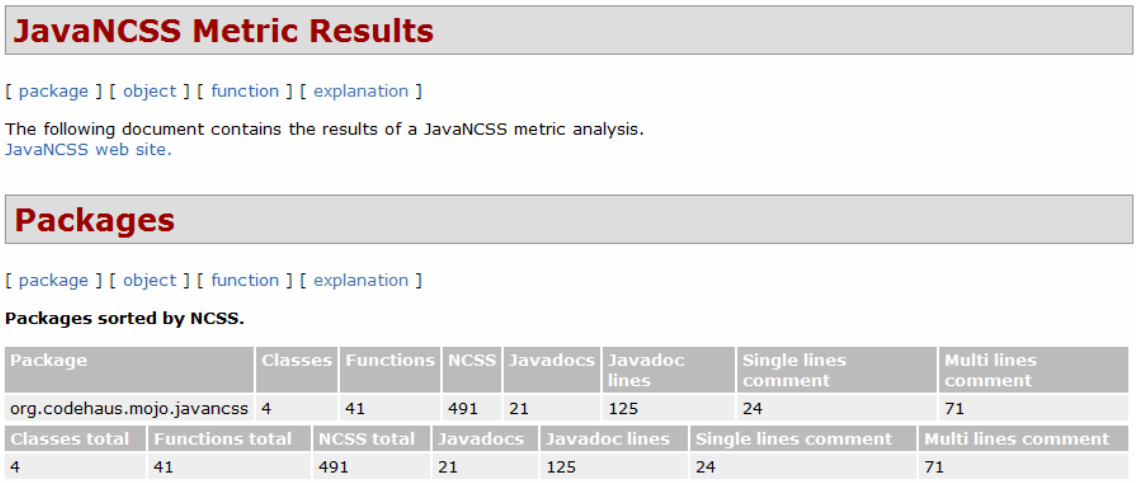
Module	Packages	Classes total	Functions total	NCSS total	Javadocs	Javadoc lines	Single lines comment	Multi lines comment
maven-scm-api	26	64	354	1608	201	1018	55	1093
maven-scm-manager-plexus	1	2	24	72	18	58	6	30
maven-scm-test	12	14	115	756	61	292	72	222
maven-scm-provider-vss	4	7	49	505	31	196	33	304
maven-scm-provider-perforce	15	28	105	1284	35	200	139	518
maven-scm-provider-bazaar	11	20	82	1012	35	138	72	369
maven-scm-provider-svn-commons	6	10	86	494	54	305	68	152
maven-scm-provider-svntest	8	8	19	129	8	36	2	120
maven-scm-provider-svnexe	12	20	71	901	26	138	105	327
maven-scm-provider-local	8	10	23	398	18	66	15	150
maven-scm-provider-clearcase	14	24	95	947	47	201	162	372
maven-scm-provider-starteam	15	26	116	1300	51	254	114	500
maven-scm-provider-cvstest	8	11	38	238	14	60	12	165
maven-scm-provider-cvs-commons	14	17	88	837	52	249	41	290
maven-scm-provider-cvsexe	10	16	40	449	16	65	44	248
maven-scm-client	1	1	8	123	1	5	16	15
maven-scm-plugin	1	13	47	485	33	329	11	195
Totals	166	291	1360	11538	701	3610	967	5070

Figura 5: Resultados obtidos pela análise do JavaNCSS por módulos.

JavaNCSS é uma ferramenta simples que mede dois padrões de métricas de código-fonte para a linguagem de programação Java: o número de instruções de código sem comentário (*Non Commenting Source Statements*) e a complexidade

[11] http://www.kclee.de/clemens/java/javancss/

ciclomática (métrica de McCabe[12]). As métricas são coletadas globalmente – para cada classe ou para cada função. Algumas métricas disponibilizadas por esta ferramenta são: contagem de pacotes, classes, funções e classes internas; quantidade de comentários Javadoc por classe e método; e médias desses valores são calculadas.



JavaNCSS Metric Results

[package] [object] [function] [explanation]

The following document contains the results of a JavaNCSS metric analysis.
JavaNCSS web site.

Packages

[package] [object] [function] [explanation]

Packages sorted by NCSS.

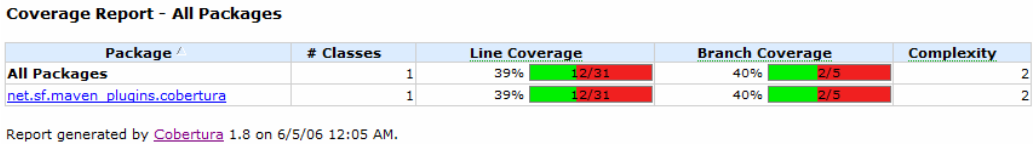
Package	Classes	Functions	NCSS	Javadocs	Javadoc lines	Single lines comment	Multi lines comment
org.codehaus.mojo.javancss	4	41	491	21	125	24	71

Classes total	Functions total	NCSS total	Javadocs	Javadoc lines	Single lines comment	Multi lines comment
4	41	491	21	125	24	71

Figura 6: Resultados obtidos pela análise do JavaNCSS por pacotes.

2.1.8.Cobertura[13]

Cobertura é uma ferramenta livre feita em Java que calcula a porcentagem de código que é percorrido pelos testes de unidade. Esta métrica pode ser usada para identificar que porção de um programa Java não está coberta por testes.



Coverage Report - All Packages

Package	# Classes	Line Coverage	Branch Coverage	Complexity
All Packages	1	39% 12/31	40% 2/5	2
net.sf.maven_plugins.cobertura	1	39% 12/31	40% 2/5	2

Report generated by Cobertura 1.8 on 6/5/06 12:05 AM.

Figura 7: Exemplo de relatório gerado pelo Cobertura.

2.1.9.Clover[14]

Testes de unidade podem determinar a qualidade de um código. Mas quem determina a qualidade dos testes que estão sendo escritos? O Clover é uma ferramenta altamente configurável para análise de cobertura do código-fonte. Esta

[12] http://www.sei.cmu.edu/str/descriptions/cyclomatic_body.html

[13] http://cobertura.sourceforge.net/

[14] http://www.cenqua.com/clover/

é capaz de descobrir seções do código que não estão sendo exercitadas adequadamente pelos testes de unidade. Ao contrário do Cobertura, o Clover é uma ferramenta paga.

2.2.Precauções

Para determinar com certeza se as mudanças estão melhorando ou degradando o processo de desenvolvimento de software, é preciso medir. Logo, dados precisam ser coletados. Mas quais dados? É importante ter um conjunto definido de campos na hora em que um defeito for reportado:

- Descrição administrativa do defeito (a data em que foi reportado, a pessoa que reportou, a descrição, a versão afetada, a data da correção).
- Descrição completa do problema.
- Passos para repetir o problema.
- Sugestões de alternativas para evitar o problema.
- Defeitos relacionados.
- Gravidade do problema (bloqueador, crítico, cosmético).
- Origem do defeito: requisitos, design, código ou testes.
- Horas gastas para achar o problema.
- Horas gastas para resolver o problema.

Só depois de obter alguns números será possível dizer se um projeto está melhorando ou piorando.

Inicialmente, a ferramenta de gerência de defeitos e pendências pode se tornar um gerenciador de tarefas. Ou seja, as pessoas podem começar a usar a ferramenta como um mecanismo de lembrete (agenda) ou envio de recados. Não é este o objetivo de uma ferramenta como o JIRA. Por isso, todos os envolvidos devem estar cientes do que é um *bug* e uma pendência e, mais importante, do que não é nenhum dos dois. O uso incorreto faz com que a ferramenta resolva problemas que não são sua responsabilidade e isso começa a criar um ambiente confuso, propenso a ser abandonado no futuro.

Ferramentas como o JIRA permitem que qualquer pessoa envolvida no projeto reporte problemas. Porém, alguns clientes não gostam de ter que manipular uma ferramenta para registrar *bugs* e pedir modificações. Eles preferem algum tipo de suporte, em que possam simplesmente reclamar (seja por telefone

ou e-mail). Outros clientes, no entanto, têm facilidade em usar ferramentas desse tipo. Por isso, antes de oferecer ao cliente a possibilidade de ele usar o gerenciador de *bugs* e pendências, é preciso conhecer o tipo de cliente com que se está lidando. Também é preciso esclarecer a ele o conceito de *bugs* e pendências.