

**Faculdade de Engenharia da Universidade do Porto**



**ESOF 2016/2017 - Mestrado Integrado em Engenharia Informática e Computação**

**Autores:**

Andreia Rodrigues - [up201404691@fe.up.pt](mailto:up201404691@fe.up.pt)

Eduardo Leite - [gei12068@fe.up.pt](mailto:gei12068@fe.up.pt)

Gonçalo Leão - [up201406036@fe.up.pt](mailto:up201406036@fe.up.pt)

Francisco Queirós - [up201404326@fe.up.pt](mailto:up201404326@fe.up.pt)

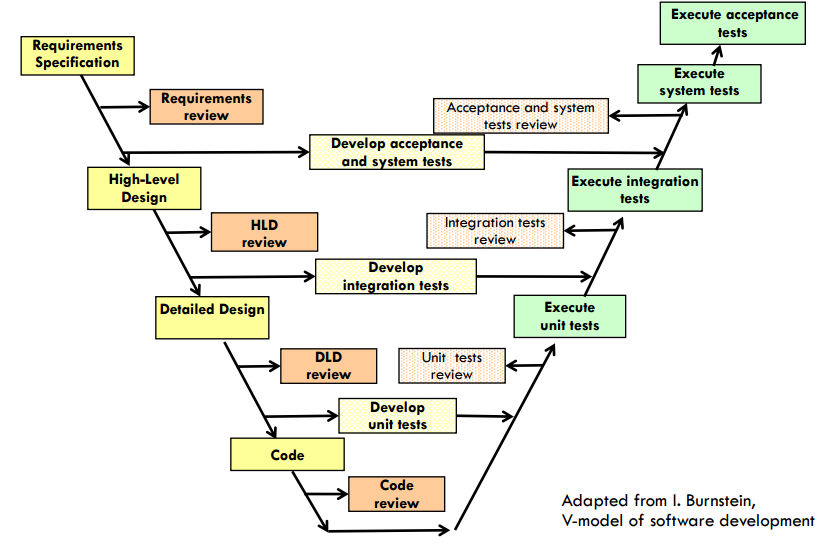
ESOF - Relatório 4

WorldEdit

Verificação e validação

A verificação e validação (V & V) de *software* é uma fase do processo de engenharia de *software* que tem início assim que os requisitos do sistema são elaborados e que ocorre em paralelo com as restantes etapas do processo. Os objetivos desta fase são mostrar que o programa respeita os requisitos que foram especificados e que cumpre todos os seus objetivos, antes que o produto final seja entregue ao cliente.

A figura abaixo apresenta o *V-Model* de desenvolvimento de *software*. A imagem tem como objetivo ilustrar que a verificação e validação do *software* ocorre em paralelo com todas as outras fases do seu desenvolvimento.



*Figura 1: V-Model de desenvolvimento de software  
(Fonte: slides das aulas teóricas)*

Existe uma diferença fundamental entre os conceitos de verificação e validação de um programa. Por um lado, verificação consiste em conferir que o *software* cumpre os seus requisitos funcionais e não-funcionais. Por outro lado, a validação é um processo mais abrangente pois consiste em confirmar que o programa faz precisamente o que cliente pretende que ele faça. A validação de *software* é essencial pois os requisitos levantados nem sempre traduzem as verdadeiras necessidades dos clientes e utilizadores do sistema.

A diferença entre verificação e validação de *software* pode ser explicada de forma sucinta citando Barry Boehm, que associa a cada uma uma questão:

* verificação - “Are we building the product right?”
* validação - “Are we building the right product?”

O seguimento deste relatório divide-se em três principais secções.

Para começar, vamos analisar o grau de testabilidade do *software*, que passará pela estudo de diferentes aspetos:

* controlabilidade do estado dos componentes sob teste.
* observabilidade dos resultados dos testes.
* isolabilidade dos componentes a testar.
* separação de funcionalidades.
* inteligibilidade dos componentes.
* heterogeneidade das tecnologias usadas.

De seguida, vamos apresentar algumas estatísticas relevantes a esta fase de verificação e validação do *plugin*, nomeadamente, o número de casos de teste e a percentagem de cobertura do código.

Por fim, será relatado o processo de identificação e correção de um *bug* do *software*.

Grau de testabilidade

O grau de testabilidade corresponde ao grau de dificuldade de testar um sistema (ou um dos seus componentes). Por outras palavras, a testabilidade de um componente indica o quão fácil é elaborar casos de teste para esse elemento e o quão provável é que esses testes revelem defeitos, caso estes existam.

Este grau depende de vários fatores, alguns dos quais serão apresentados de seguida para discutir o grau de testabilidade do WorldEdit.

Controlabilidade

A controlabilidade refere-se a quão fácil é controlar o estado do componente sob teste (Component Under Test - CUT).

No caso do WorldEdit a controlabilidade não é muito elevada, isto deve-se principalmente a dois factores:

* Os componentes estão muito interligados entre si (tal como foi visto no terceiro relatório), o que dificulta o controlo do estado de um componente a cada momento, visto que é preciso ter em conta a entrada de valores externos ao componente.
* O WorldEdit é um *plugin* do Minecraft, pelo que estas ligações estendem-se a código que não pertence ao plugin.

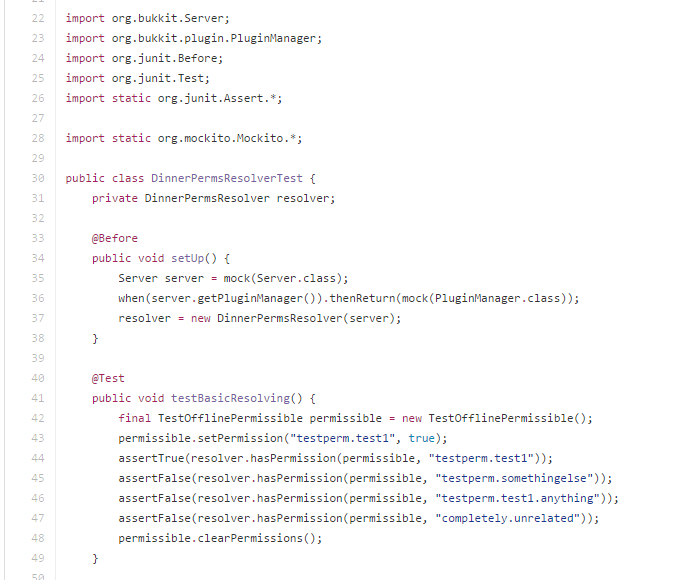
O projeto utiliza testes unitários para exercitar as suas classes e métodos. Estes testes são implementados usando classes de teste e a biblioteca JUnit.

Nestas classes, é possível distinguir vários tipos de métodos.

Os métodos com a notação @Test tratam-se das funções que vão realmente exercitar o código usando métodos do tipo *assert*.

Os métodos com a notação @Before são usados para inicializar objetos que depois vão ser usados nos métodos de teste dessas classes. Tratam-se de métodos interessantes para a controlabilidade pois evitam a repetição da especificação de alguns dos *inputs* necessários para testar um componente, o que permite que sejam alterados mais facilmente.

A figura abaixo apresenta um excerto de código de um dos ficheiros de teste que utiliza estes dois tipos de métodos, “Test” e “Before”.



*Figura 2: Excerto de código de um dos ficheiros de teste, DinnerPermsResolverTest.java*

Observabilidade

A observabilidade mede a facilidade de obter os resultados intermédios e finais dos testes efetuados.

Neste caso é possível distinguir dois casos:

* Nas funcionalidades que não interagem, ou interagem pouco, com o Minecraft, é mais fácil de observar se o comportamento é o esperado, pois apesar de às vezes o grau de interdependência ser elevado, ainda existe um grau de observabilidade aceitável.
* Nas funcionalidades que interagem directamente com o Minecraft, a tarefa torna-se mais complicada, pois às vezes os resultados só são observáveis no próprio minecraft. Como o WorldEdit tem como principal objectivo editar o mundo do Minecraft, esta condição é muito frequente, levando a uma observabilidade reduzida nestes casos.

Para poder observar os resultados da execução dos testes unitários, uma possibilidade é importar o projeto para Eclipse, compilar o código usando o Gradle e correr os testes usando o JUnit. Desta forma, é possível aceder a várias informações sobre os testes realizados como que testes passaram e, caso seja usado uma ferramenta de análise de cobertura (como o EclEmma), determinar que percentagem de código de cada *subpackage* é que os métodos de teste cobrem.

O WorldEdit também faz uso de uma ferramenta de *continuous integration* para testar a nova *codebase* com o fim de encontrar potenciais erros. Esta ferramenta, TravisCI, funciona ao nível dos *commits* feitos no repositório. Sempre que que é adicionado um commit, este passa por esta ferramenta que faz *build* do *source code* e depois testa o *output* com os testes com quais a ferramenta está configurada. Assim, o TravisCI fornece observabilidade a uma escala mais global do projeto.

Isolabilidade

A isolabilidade define quão fácil é exercitar o componente sob teste de forma isolada (sem que os resultados dos testes sejam enviesados por outros componentes).

No caso do WorldEdit, a isolabilidade é relativamente baixa, visto que os componentes estão bastante interligados.

Para compensar esta baixa isolabilidade, o WorldEdit faz uso da ferramenta Mockito em algumas das suas classes de teste. Esta ferramenta permite definir de forma fácil *mock objects*. Um *mock object* é um objeto com o mesmo *interface* que os objetos externos a serem usados no método de teste e que simula as suas funcionalidades de forma previsível. Assim, um *mock object* permite que, ao testar um método de uma classe A que faça uso de um método de uma classe B, seja possível criar um *mock* *object* para a classe B de forma a que eventuais erros na definição na classe B não interfiram nos resultados de teste sobre esse método da classe A.

Um exemplo do uso do Mockito no WorldEdit é visível na figura 2, onde são criados *mocks* para as classes Server e PluginManager, para que a definição destas classes não interfira nos resultados dos testes sobre a classe DinnerPermsResolver.

Separação de Funcionalidades

A separação de funcionalidades indica o quão bem estão definidas as responsabilidades do componente sob teste.

Na realização de projetos de dimensão elevada, é importante garantir que cada funcionalidade fique o mais confinada possível ao componente que lhe está associado, tentando torná-lo o mais modular possível. Se isto não acontecer, pode levar a um código confuso e mal estruturado, tornando mais difícil a sua compreensão e manutenção por parte dos diferentes contribuidores do projeto.

No caso do *plugin* WorldEdit, tal como apresentado no terceiro relatório, os componentes do projeto são bastante interdependentes. Mesmo assim, estes têm por norma uma funcionalidade bem definida, o que nos leva a dizer que existe um nível intermédio de separação de funcionalidades.

Inteligibilidade

A inteligibilidade indica o quão um componente está bem documentado e é auto-explicativo.

Trata-se de um parâmetro importante da testabilidade pois, quanto mais claro o código for, mais fácil é de o entender corretamente num menor espaço de tempo, sendo assim mais simples escrever testes unitários para o exercitar ou de o corrigir, caso tenha algum *bug*.

Ao nível dos *packages*, é apenas explicado o papel de cada um dos quatro pacotes principais do projeto, em COMPILING.md (<https://github.com/up201406036/WorldEdit/blob/master/COMPILING.md>). As funcionalidades dos sub-pacotes do projeto, nomeadamente os do *package* worldedit-core, não são claramente indicadas, o que dificultou a tarefa de análise mais profunda do código, por parte dos autores deste relatório, no momento da elaboração do terceiro relatório, relativo à *arquitetura* do *software*.

Ao nível das classes, o projeto encontra-se melhor documentado, dado que a maior parte das classes em worldedit-core usam notação Javadoc para explicar o seu objetivo e dos seus métodos públicos. As classes dos três restantes pacotes principais (worldedit-bukkit, worldedit-forge e worldedit-sponge) não apresentam esta documentação em Javadoc.

O ficheiro CONTRIBUTING.md (<https://github.com/sk89q/WorldEdit/blob/master/CONTRIBUTING.md>) especifica que, para que um *pull request* seja aceite, todos os métodos públicos que sejam acrescentados têm de ser corretamente documentados com Javadoc (usando as notações apropriadas como @param e @return).

Além disso, este ficheiro especifica um conjunto de regras para que o código escrito seja o mais homogéneo possível, o que facilita a sua legibilidade. Estas regras não são triviais pois, dado que o projeto é *open-source*, o código desenvolvido vem de dezenas de contribuidores diferentes, cada um tendo os seus hábitos e costumes de escrita de código. Exemplos de regras incluem:

* seguir as *Oracle coding conventions*,
* usar apenas espaços para indentação
* não exceder 120 caracteres em cada linha de código.

Assim, o inclusão do ficheiro CONTRIBUTING.md no repositório do projeto mostra que os contribuidores do projeto se preocupam que o código se mantenha compreensível.

Heterogeneidade

A heterogeneidade refere-se a quanto o uso de diferentes tecnologias requer o uso em paralelo de diferentes métodos de teste de código e ferramentas.

Uma forte heterogeneidade traz a desvantagem de adicionar pontos de falha no momento de teste do código pois as ferramentas usadas podem falhar. Assim, torna-se necessário que os métodos de teste desenvolvidos estejam preparados para lidar com este gênero de situações (lançando uma exceção especial ou executando um método de teste equivalente que não faça uso da ferramenta que falhou).

O projeto WorldEdit está inerentemente assente sobre a API do jogo Minecraft, dada a natureza do *software* que funciona como uma extensão do Minecraft (trata-se de um *plugin*). Isto cria uma grande heterogeneidade no projecto.

Além disso, o código apresenta a utilização das *frameworks* JUnit e Mockito para criar e executar código de teste e criar facilidades para imitar funcionalidades do código a testar, respetivamente.

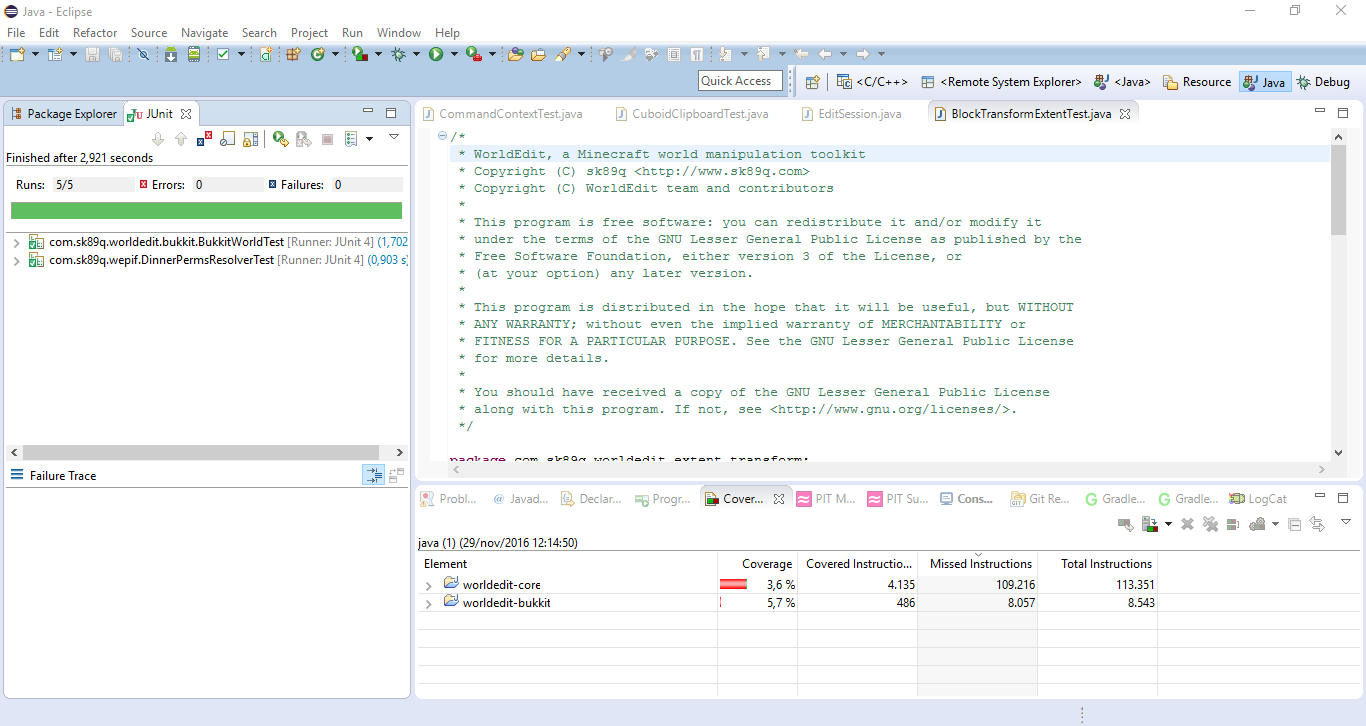
Dado que as ferramentas usadas pelo WorldEdit têm um bom grau de fiabilidade, os principais contribuidores do projeto consideram seguro assumir que as ferramentas usadas funcionarão como esperado.

Estatísticas de teste

A análise às estatísticas de teste obtidas a partir dos testes unitários permite perceber se estes foram bem concebidos (nomeadamente, se cobrem uma parte significativa do código, ou, pelo menos, os módulos mais utilizados, e se exercitam os diversos blocos de código com diversos *input*) , se permitem uma boa validação do código e a detecção de bugs, e a quantidade de código do projeto que foi testada, o que é bastante importante para o sucesso de um projeto.

As seguintes estatísticas dos testes unitários do WorldEdit foram obtidas usando as ferramentas JUnit e EclEmma, após termos carregado o projeto no Eclipse. As estatísticas referem-se à versão mais atual do repositório no dia 29 de fevereiro de 2016, por volta das 12 horas.

No repositório, existem dois diretórios de ficheiros de teste: um (<https://github.com/up201406036/WorldEdit/tree/master/worldedit-core/src/test/java/com/sk89q>) exercita classes do package “worldedit-core”, enquanto que o outro testa aspetos específicos ao código para a plataforma Bukkit (<https://github.com/up201406036/WorldEdit/tree/master/worldedit-bukkit/src/test/java/com/sk89q>).

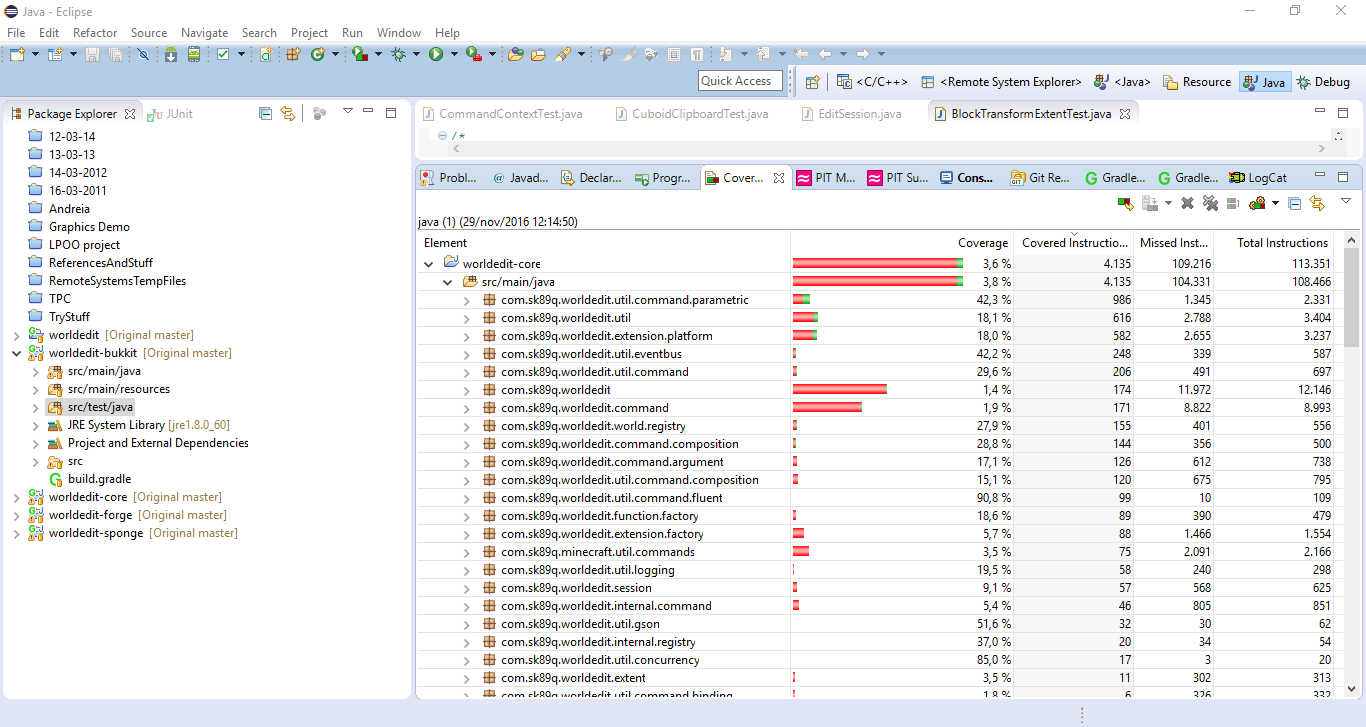


*Figura 3: Estatísticas produzidas pelo EclEmma usando ambos os diretórios de teste*

(<https://gyazo.com/a8e96766311f2fce1959b45070c9c4a6>)

Quanto ao diretório de testes específicos para a plataforma Bukkit, existem apenas 5 testes. Estes testes cobrem apenas uma percentagem pequena do código, tanto em relação ao “worldedit-core” (apenas 3,6%) como ao “worldedit-bukkit”(5,7%). Apesar da percentagem de código testado ser maior no caso do “worldedit-bukkit”, a quantidade de instruções deste pacote é muito mais reduzida que no “worldedit-core”.

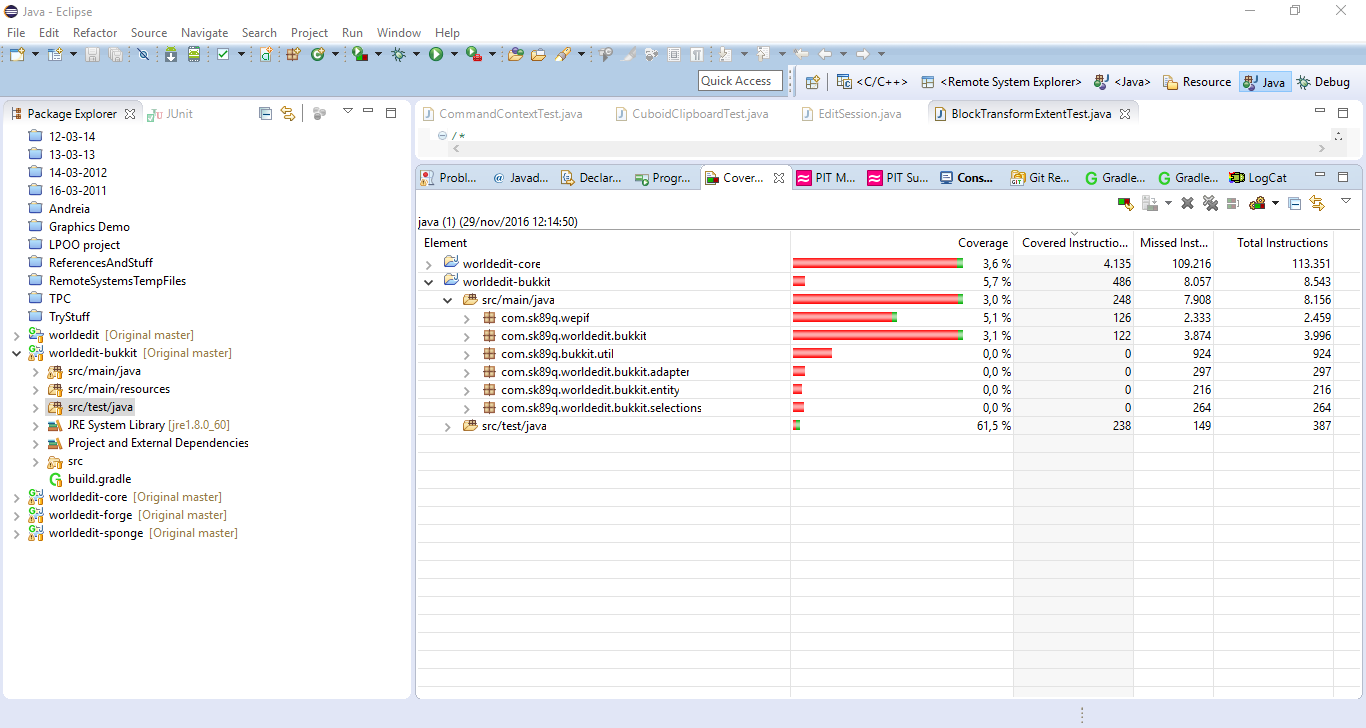
Nas seguintes imagens, podemos analisar os resultados dos testes mais em detalhe:



*Figura 4: Cobertura do testes para o “package” worldedit-core, em relação aos testes feitos especificamente para a plataforma Bukkit.*

(<https://gyazo.com/ebefb2f03c0b8ce28ec75d99a7b4ac2f>)

Em relação aos testes que exercitam aspetos específicos ao código para a plataforma Bukkit no Core do projeto, podemos observar as diferentes percentagens de cobertura distribuídas pelos pacotes cujo número total de instruções cobertas pelos testes é maior. Podemos observar que o pacote “main” do projeto tem uma cobertura de apenas 3,8%, o que não é de estranhar por apenas estarem a ser testados aspetos específicos da plataforma Bukkit.

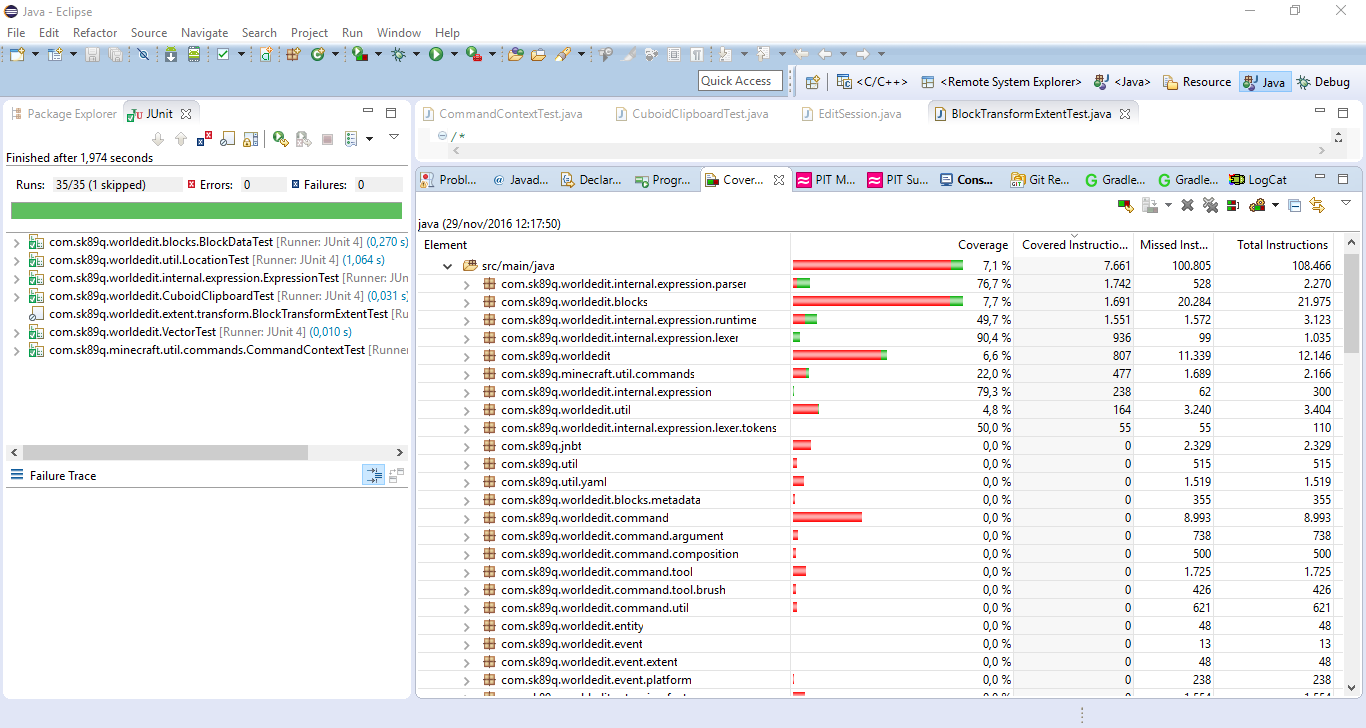


*Figura 5: Cobertura do testes para o “package” worldedit-bukkit, em relação aos testes feitos especificamente para a plataforma Bukkit.*

(<https://gyazo.com/0914486838cd5933a3aedb1e6032e011>)

Quanto ao mesmo tipo de testes mas para o pacote relativo à plataforma Bukkit, a cobertura é bastante menor do que seria esperado, sendo apenas de 5,7%. Esperava-se que esta percentagem fosse maior por este pacote ser inteiramente dedicado à plataforma sobre a qual os testes incidem.

Nas imagens seguintes podemos verificar a cobertura dos testes relativos ao código do pacote “core” do projeto.



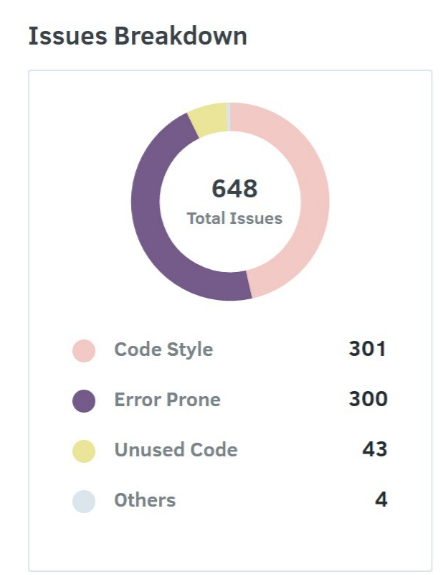
*Figura 6: Cobertura dos testes do worldedit-core para o “package” src*

(<https://gyazo.com/4a82679799a57bfe197b69ede3a0314b>)

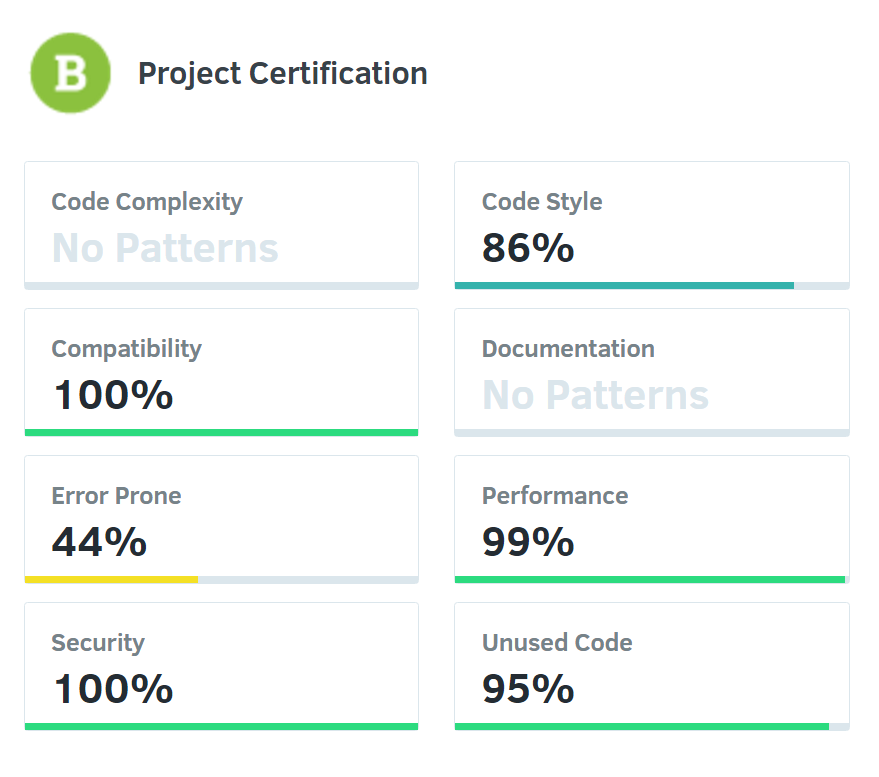
Em relação aos testes que exercitam o código das classes do package “worldedit-core”, existe um total de 35 testes que produzem uma cobertura de 7,1% no pacote “src” do projeto, um valor bastante mais baixo do que o esperado. Alguns dos pacotes internos ao “src” que têm maior número de instruções testadas são o “internal.expression.parser” e “blocks”.

Uma causa possível para esta percentagem ser tão baixa é que no ficheiro CONTRIBUTING.md, direcionado a toda a qualquer pessoa que queira contribuir para o projeto, é referido que a produção de testes unitários é opcional. Como a lista de diferentes contribuidores do projeto é bastante extensa, não existe um grande controlo sobre a percentagem de contribuidores que produz testes unitários e quem prefere não o fazer.

Podemos ainda observar, após correr o projeto na ferramenta Codacy, os issues do projeto, ou seja problemas, falhas, inconsistências ou conflitos no código.



<https://gyazo.com/c1217b4e1fcf3609a1f6238ba75b1df8>



<http://i.imgur.com/HQy3nvM.png>

*Figuras 7 e 8: Output the ferramenta de análise Codacy*

Desta forma podemos concluir que a cobertura geral dos testes em qualquer dos pacotes principais do projeto (referente a plataformas específicas ou ao pacote principal do projeto) é atualmente baixa. Deveria ser feito um esforço para aumentar a quantidade e talvez a qualidade dos testes.

Relato da correção de um bug

Casos particulares do *bug* que encontrámos foram referidos no *issue tracker* do projeto (<http://dev.enginehub.org/youtrack/issue/WORLDEDIT-3450>) (<http://dev.enginehub.org/youtrack/issue/WORLDEDIT-3362>) (<http://dev.enginehub.org/youtrack/issue/WORLDEDIT-2986>).

Num destes *issues*, um desenvolvedor do projeto declara o problema como não reproduzível. É possível que a versão de então (há cerca de um ano, 17 de Setembro de 2015) não tivesse este bug, mas é facto que na versão atual, este bug existe e é facilmente reproduzível. Com efeito, os autores deste relatório, após terem identificado as secções do código ligados à ocorrência do *bug* e de as terem analisado (análise estática), correram o Minecraft com o *plugin* do WorldEdit, usando a plataforma Bukkit para verificar em que condições o *bug* se manifestava (análise dinâmica).

O *bug* está presente no comando “center”. Este comando permite associar um tipo de bloco ao centro de uma região selecionada. No entanto, o comando nem sempre tinha o comportamento esperado.

Uma descrição mais completa deste bug e da forma como este foi corrigido está disponível no pull request (<https://github.com/sk89q/WorldEdit/pull/371>) elaborado pelos autores deste relatório.

O *bug* foi corrigido com sucesso, e após, efetuar algumas alterações à nossa correção, motivadas por sugestões por parte de alguns dos principais contribuidores do projeto, o *pull request* foi aceite pela comunidade do WorldEdit.

Conclusões e análise crítica

Em termos de testabilidade, apesar de haver uma forte interdependência entre módulos e uma fraca documentação de certas classes e sub-pacotes, o projeto tem várias características que facilitam a verificação e validação do sistema.

Por um lado, o projeto usa testes unitários e diversas ferramentas de suporte como JUnit, Mockito e TravisCI.

Por outro lado, o projeto apresenta várias regras para contribuir que impõem que o código seja bem documentado e que seja escrito de forma homogénea, o que contribui para uma melhor inteligibilidade.

Neste projeto, as estatísticas não são muito favoráveis visto que os testes unitários desenvolvidos têm uma cobertura muito fraca do código, como já discutido em secções anteriores.

No entanto, devido ao ambiente sem risco e não-profissional do projeto, é compreensível não haver tanta dedicação a esse nível, mesmo que não seja o mais correto em termos de boas práticas de desenvolvimento de *software*.

Bibliografia

Software Engineering, Ian Sommerville, 9th Edition, capítulo 8.

# Introduction to Software Testing, Paul Ammann, Jeff Offutt, 2nd Edition, capítulo 3.1. Disponível em: https://books.google.pt/books?id=bQtQDQAAQBAJ&pg=PA36&lpg=PA36&dq=software+observability&source=bl&ots=fxbM3-2WiQ&sig=CaObHiSrq32X0HDKHmna-Oalw6E&hl=en&sa=X&ved=0ahUKEwjfk\_Dy2cnQAhVH\_4MKHTWCAK8Q6AEIITAB#v=onepage&q=software%20observability&f=false

http://builds.enginehub.org/