

**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 5

WorldEdit

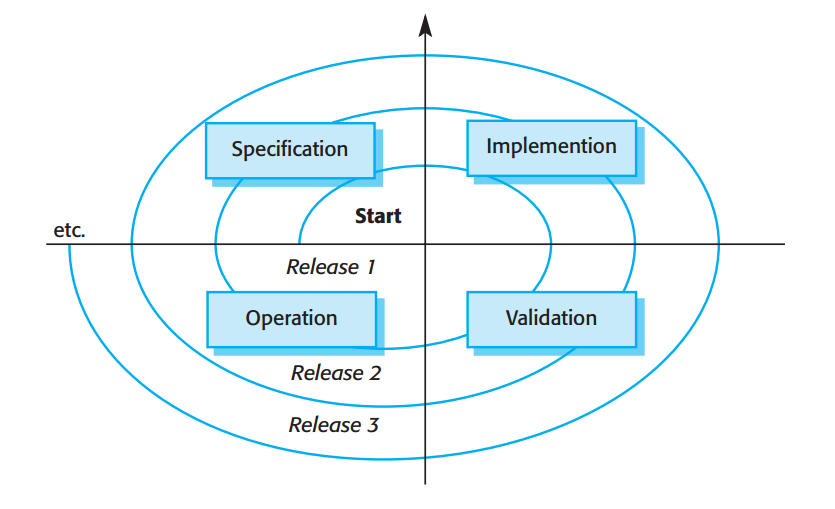
Evolução e manutenção de *software*

O desenvolvimento de software não acaba após a entrega deste. Para um projeto se manter atualizado face às mudanças no mercado e às expectativas dos utilizadores, é necessário implementar novas funcionalidades que respondam aos requisitos que lhe são propostos. Para além disto, a evolução e manutenção do software também é necessária para a correção de erros que surgem, o aumento ou modificações ao *hardware* que faz parte o sistema ou a melhoria da performance e fiabilidade do sistema, caso necessário.

As empresas investem grandes quantidades de dinheiro em *software* e para que o valor destes investimentos seja mantido, estes requerem alterações para que se mantenham atualizados face aos novos requisitos. Dessa forma, na maior parte das grandes empresas, uma maior parte do orçamento é dedicado a mudar e evoluir *software* existente do que a desenvolver novo *software*. A necessidade de evolução do sistema pode tornar-se óbvia mesmo antes do seu lançamento inicial, razão que leva às empresas a começarem o desenvolvimento da segunda versão ainda antes da versão atual ser lançada.

Durante a evolução do software em causa, há um fluxo constante de alterações dos requisitos propostos. Ora, à medida que o software é modificado para que responda a estes requisitos, a sua estrutura tende a degradar-se e fazer mudanças ao sistema torna-se cada vez mais caro, até chegar a um ponto do seu ciclo de vida em que fazer uma mudança significativa se torna cada vez menos rentável. A partir deste ponto de viragem, o *software* entre numa fase de *servicing*, onde continua a ser útil mas apenas são aplicadas mudanças pontuais (como a correção de *bugs*) para o manter operacional. O *software* pode passar por um segundo ponto de viragem em que entra numa fase denominada por *phase-out*, onde o sistema continua a ser usado mas já não sofre mais mudanças.

O desenvolvimento e evolução de software pode ser pensado como um processo integrado, iterativo que pode ser representado pelo modelo em espiral com o levantamento de requisitos, *design*, implementação e testes em curso durante toda a vida útil do sistema. O processo tem início com a primeira versão do sistema. Uma vez entregue, são propostas alterações e começa o desenvolvimento da segunda versão, e por aí em diante, com novos lançamentos e alterações necessárias que vão ser implementadas no seguinte.



*Figura 1: Representação do processo de desenvolvimento e evolução de software como um ciclo*

*(Fonte:* Software Engineering, Ian Sommerville, 9th Edition, capítulo 9)

<https://gyazo.com/e35ce73592d54a681e54f5ce83dc3415>

O seguimento deste relatório divide-se em duas principais seções.

Para começar, vamos analisar a manutenibilidade do *software*, usando as diretivas BMS.

De seguida, vamos apresentar o processo de evolução de uma *feature* do WorldEdit.

Manutenibilidade do *software*

A manutenibilidade de um *software* indica o quão fácil é aplicar-lhe mudanças para o tornar conforme a novos requisitos.

A manutenibilidade do WorldEdit foi analisada usando o serviço baseado em *web* Better Code Hub que verifica se o projeto segue as dez diretivas do “Building Maintainable Software: Ten Guidelines for Future-Proof Code” (diretivas BMS).

Das dez diretivas BMS, segundo o Better Code Hub, o WorldEdit respeita apenas três delas: “Keep Architecture Components Balanced”, “Keep your codebase small” e “Write Clean Code”.

Vamos agora, de forma sucinta, apresentar as dez diretivas BMS e explicar porque é que estas são ou não respeitadas pelo WorldEdit.

“Write Short Units of Code”

Esta diretiva defende que as unidades de código devem ocupar poucas linhas, para que sejam mais fáceis de perceber, reutilizar e testar. Uma unidade de código é o menor grupo de linhas de código que podem ser mantidas e executadas de forma independente. No caso do Java, uma unidade de código corresponde a um método ou construtor de uma classe.

Idealmente, uma unidade de código não deve ultrapassar as 15 linhas. Caso este limite seja excedido, a diretiva impõe que a unidade seja dividida em sub-unidades de código.

O WorldEdit não respeita esta diretiva, visto que uma grande parte das suas unidades de código ocupam mais que 15 linhas, e uma parte significativa destas unidades ultrapassa 60 linhas de código.

Em particular, a classe BlockData do pacote com.sk89q.worldedit.blocks (em worldedit-core), responsável por manipular os dados contidos nos blocos constituintes do mundo Minecraft, apresenta vários métodos que excedem, por uma grande margem, o limite de 15 linhas de código. A título de exemplo, o método “public static int flip(int type, int data, FlipDirection direction)” ocupa 320 linhas de código. Isto deve-se ao método incluir uma estrutura “switch … case” para lidar com os diferentes tipos de bloco (tochas, portas de madeira, etc).

“Write simple units of code”

A diretiva defende que deve ser limitado o uso de *branch points* em cada unidade de código para que estes sejam mais fáceis de ser modificados e testados. Um *branch point* é um ponto do programa que altera o fluxo normal de execução de código, de forma a que a próxima linha de código a ser executada não seja necessariamente a que sucede à anterior. Um *branch point* é criado usando estruturas de controle como *if/else*’s, *while*’s e *for*’s.

Em particular, a directiva impõe que o número de *branch points* de cada unidade de código deve ser no máximo 6. Caso este limite seja ultrapassado, tal como na diretiva anterior, a unidade de código deve ser decomposta em sub-unidades.

O WorldEdit não respeita esta diretiva. Tal como para o ponto anterior, a classe BlockData contém métodos que desrespeitam esta regra devido a um forte uso de estruturas “*switch...case*” com várias ramificações. Uma possível técnica de *refactoring* ao código que poderia resolver este problema seria o “Replace Conditional with Polymorphism”, que passaria por criar subclasses para BlockData para cada ramificação dos estruturas condicionais (“*switch...case*”).

“Write Code Once”

Esta diretiva defende que não deve haver código duplicado, para evitar que *bugs* tenham de ser corrigidos em mais que um sítio (o que é tanto ineficiente como mais sujeito a erros).

Caso exista código duplicado, várias técnicas de *refactoring* podem ser utilizadas como a extração do código repetido para um novo método (*Extract Method*), caso a repetição ocorre dentro de uma única classe, ou, a criação de uma superclasse com métodos que tenham o código duplicado (*Extract Superclass*), caso o código duplicado esteja presente em mais que uma classe e nenhuma das classes envolvidas tenha uma superclasse.

O WorldEdit não respeita este critério. Um exemplo de código duplicado são os *enums* BlockType e ItemType, do pacote com.sk89q.worldedit.blocks (em worldedit-core), que contêm ambas um mesmo conjunto de constantes de informação de cada tipo de bloco do Minecraft. Cada constante é um tuplo da forma <tipo de bloco>(<número de identificação, <nome do tipo de bloco>, <outros nomes que se referem ao tipo de bloco>).

“Keep Unit Interfaces Small”

Esta diretiva defende que devem haver poucos parâmetros de entrada em cada unidade de código, para que estes sejam mais fáceis de compreender e de reutilizar.

A regra define que o número máximo de parâmetros de entrada é 4. Caso o limite seja ultrapassado, podem-se unir parâmetros que se relacionem uns com os outros num único objeto de uma nova classe.

O WorldEdit não respeita esta regra. A título de exemplo, vários métodos da classe GenerationCommands do pacote com.sk89q.worldedit.command (em worldedit-core) têm uma quantidade elevada de argumentos, tais como generateBiome, que tem 10.

“Separate Concerns in Modules”

Esta diretiva defende que cada módulo do código não deve ter demasiadas responsabilidades. Trata-se de uma boa prática de programação pois reduz o impacto no programa como um todo de mudanças no código de um módulo em específico.

A regra defende que cada módulo não deve ter mais que 10 *incoming calls*. Uma *incoming call* para um dado módulo trata-se da chamada de um dos seus métodos.

O WorldEdit também não respeita este critérios visto que, por exemplo, a classe BlockType do pacote com.sk89q.worldedit.blocks tem 50 *incoming calls*, 11 das quais oriundas da classe AbstractPlayerActor do pacote com.sk89q.worldedit.extension.platform (em worldedit-core).

“Couple Architecture Components Loosely”

Esta diretiva defende que cada componente de alto nível deve fazer o mínimo possível de uso de outros componentes desse nível, para ser mais fácil manter os componentes isolados.

Em termos práticos, isto pode ser atingido reduzindo a quantidade de código de *interface*, ou seja, código utilizado para comunicar entre componentes, mantendo as mesmas funcionalidades e eficiência.

O WorldEdit não respeita esta directiva, pois como referido em relatórios anteriores, os componentes apresentam um elevado grau de interdependência.

“Keep Architecture Components Balanced”

Esta diretiva defende que sejam equilibrados o número e o tamanho relativo dos componentes do código (pacotes), para que seja mais fácil de encontrar secções específicas de código.

A regra defende que deve haver entre 2 e 12 componentes e que estes sejam de tamanho aproximadamente igual. Para este segundo critério, é usado uma métrica chamada *component size uniformity*, que deve ser mantida abaixo de 0.71 (o cálculo desta métrica não será apresentada, dada a sua complexidade).

O WorldEdit respeita este critério visto que tem 5 componentes principais e o seu *component size uniformity* é de 0.68. Os 5 componentes identificados correspondem a worldedit-core, worldedit-forge, worldedit-sponge e worldedit-bukkit, mais o pacote “contrib” (pacote de pequena dimensão que contém apenas código em Javascript para quatro *craftscripts*). Convém salientar que o Better Code Hub só analisou os pacotes principais, e não os seus sub-pacotes (nomeadamente os de worldedit-core), que, no entender dos autores deste relatório, seria mais interessante, visto que é natural que o pacote worldedit-core tenha maior volume que os restantes pacotes por conter o código comum a cada plataforma suportada pelo WorldEdit.

“Keep Your Codebase Small”

Esta diretiva defende que o volume total do código deve ser reduzido, para que mudanças estruturais ao código exijam menos esforço.

A regra defende que o volume do projeto deve ser inferior a 20 *Man-years*. Um *Man-year* mede a quantidade de trabalho levado a cabo por uma pessoa num ano. O Better Code Hub não especifica como é que a quantidade de *Man-Years* de um projeto é avaliado.

Caso a diretiva não seja respeitada, o código pode ser *refactored* para obter a mesma funcionalidade em menos linhas de código, ou podem ser usadas bibliotecas e *frameworks* externas para implementar certas funcionalidades mais elementares.

O WorldEdit respeita este critério dado que foi avaliado que o projeto tinha 67 *Man-months*, ou seja, cerca de 5,58 *Man-years*.

“Automate tests”

Esta diretiva defende que sejam automatizados os testes para tornar o desenvolvimento de código mais previsível e seguro.

Para projetos de grande escala (que têm mais de 10.000 linhas de código), como é o caso do WorldEdit (que tem 49.212 linhas, segundo o Better Code Hub), a regra exige que o número total de linhas de código de teste seja pelo menos 50% do número de linhas do restante código, e que o *assert density* deve ser de pelo 5% (o Better Code Hub não especifica como é que esta métrica é definida).

O WorldEdit não respeita esta diretiva, visto que o *test code* percentage é de apenas 2%, apesar do *assert density* ser de 17%. Estes valores devem-se ao facto que não foram criados muitas classes de teste para exercitar o código, visto que as regras para contribuir para o projeto não inpõem que sejam criadas tais classes (tal como foi visto no relatório anterior).

“Write Clean Code”

Esta diretiva defende que o código deve ser mantido limpo, ou seja, não deve conter *code smells*. Exemplos de *code smells* incluem *dead code* (uso de código que é executado mas cujo resultado não é usado) ou *duplicated code*.

O WorldEdit não respeita este princípio visto que o Better Code Hub detetou a presença de alguns *code smells*, como a presença de um bloco de código comentado em maze.js, do pacote contrib.

Processo de evolução de uma *feature* do WorldEdit

Identificação da *feature*

A *feature* implementada tem como funcionalidade a criação de um hemisfério de *blocks*. O comando para desenhar hemisférios tem como sintaxe //hemisphere <block> <radius>[,<radius>, <radius>], onde os símbolos <> indicam uma variável e os símbolos [] indicam um parâmetro opcional do comando. Os raios opcionais permitem criar formas elípticas. Também implementamos, através do uso de flags, a capacidade de inverter o hemisfério (sentido oposto) (-i) ou de, em vez de usar um sentido pré definido, usar a orientação da câmara do jogador como o sentido do hemisfério (-p). Nas funções //sphere e //hsphere (para desenhar esferas cheias e ocas, respetivamente) também adicionamos a possibilidade de usar uma flag -s juntamente com as flags já referidas para dar várias formas de chegar ao mesmo efeito e ser mais cômodo para o utilizador.

Esta ideia veio de um *thread* do fórum do projecto (http://forum.enginehub.org/threads/half-a-sphere-a-dome.168/). Dado que os requisitos da *feature* eram simples de entender e o *plugin* já era capaz de desenhar outras formas (nomeadamente esferas), os autores deste relatório acharam que esta *feature* seria interessante de implementar.

Embora haja outras formas de chegar ao efeito (criar uma esfera e manualmente tirar uma das metades, usar ferramentas avançadas de geração de volumes usando expressões matemáticas), uma ferramenta especializada é mais cómoda e fácil, especialmente para utilizadores sem as capacidades técnicas para usar as outras alternativas, o que é bastante típico da demografia do Minecraft.

Componentes que implementam a *feature* a desenvolver

A feature implementada pode ser decomposta em duas tarefas:

* Receber os dados do seu comando e verificar se este respeita a sintaxe que lhe está associada.
* Desenhar o hemisfério, segundo os parâmetros do utilizador.

Para determinarmos onde e o que alterar no projeto, decidimos usar como base uma funcionalidade já antes existente, o comando //sphere, visto que tinha um funcionamento bastante similar. Com a ajuda da funcionalidade “Find file” do Github, encontrámos a função que cria a interface entre a linha de comandos do Minecraft e o código do WorldEdit, e daí analisamos as funções invocadas.

**Componente de interpretação dos comandos**

Na classe GenerationCommands (worldedit-core/src/main/java/com/sk89q/worldedit/command/GenerationCommands.java), existem os métodos sphere e hsphere que criam uma esfera e uma esfera oca e estão associados aos comandos //sphere e //hsphere, respectivamente. Alteramos estas funções para suportarem hemisférios através de “flags” para acionar os mecanismos. A flag -s ativa o mecanismo de hemisfério. Com esta flag, podem-se usar as flags -p e -i, faladas anteriormente, caso contrário é mostrada uma mensagem de erro ao jogador.

Também, nessa mesma classe implementamos um método que serve como o recetor para o comando //hemisphere e que age como interface entre os comandos //hemisphere e //sphere (visto que se houver um método que entende todos os casos, reduz-se código duplicado).

**Componente de desenho de hemisférios**

Na classe EditSession (worldedit-core/src/main/java/com/sk89q/worldedit/EditSession.java), existe o método makeSphere que altera o mundo do Minecraft para adicionar uma esfera com os parâmetros que é lhe são dados.

Alteramos esta função para aceitar os novos parâmetros que criamos e depois usamos estes novos parâmetros para criar restrições nos blocos a gerar de forma a que crie um hemisfério como especificado.

Colocando apenas a configuração de hemisfério (//hemisphere ou outros comandos com a flag -s), limitamos o código a apenas gerar blocos cuja a diferença no y (vertical) seja positiva, ou seja, a gerar um hemisfério apontado para cima. Com a flag -i, ocorre um efeito semelhante mas com a diferença no y sendo negativa. Com apenas a flag -p, utilizamos uma funcionalidade já implementada pela equipa do WorldEdit para adquirir o vetor para onde o jogador está apontado e apenas geramos os blocos cujo o produto escalar do vetor que constitui a diferença entre o bloco e o centro e o vetor para onde o jogador está apontado seja maior ou igual a 0. Se misturarmos as flags -p e -i é o mesmo que a flag -p mas invertemos o vetor para onde o jogador está apontado.

Também tivemos de alterar outros métodos que dependem do método makeSphere, simplesmente para haver uma compatibilidade entre a quantidade e tipos dos argumentos usados para invocar a função, mantendo o comportamento destas funções igual.

É importante notar que nenhuma alteração nossa altera o funcionamento de casos de utilização já existentes.

*Pull request*

Após ter implementado e devidamente testado a nova funcionalidade, os autores deste relatório criaram um *pull request* (https://github.com/sk89q/WorldEdit/pull/374), para que acrescentar o nova código ao projeto.

Após efetuadas algumas mudanças ao código da *feature*, sugeridos por alguns dos principais contribuidores do WorldEdit (nomeadamente para permitir o desenho de hemisférios invertidos) é com grande satisfação que os autores deste relatório declaram que o *pull request* foi aceite pela comunidade!

Conclusões e análise crítica

Para concluir, o facto que o WorldEdit não respeita grande parte das diretivas BMS pode-se dever à natureza *open-source* do projeto. Com efeito, dado que há dezenas de colaboradores a contribuir para a evolução do programa, é mais difícil controlar a evolução do *software* e a sua manutenibilidade, nomeadamente no que toca ao aumento do volume de código e a código duplicado.

Relativamente à *feature* implementada, apesar da sua implementação ter exigido algum esforço (sobretudo para satisfazer os novos requisitos que os contribuidores iam sugerindo no *pull request*), as dificuldades foram ultrapassadas dado que se usou o código para a desenho de esferas como guia. Esta contribuição permitiu aos autores deste relatório compreender melhor como é possível contribuir para um projeto já com um tempo de existência e número de colaboradores considerável, mesmo que não se conheça todo o seu código em detalhe.

Bibliografia

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

Building Maintainable Software, Java Edition, Ten Guidelines for Future-Proof Code, Joost Visser, Sylvan Rigal, Rob van der Leek, Pascal van Eck, Gijs Wijnholds

Explicação da técnica “Replace Conditional with Polymorphism”: https://sourcemaking.com/refactoring/replace-conditional-with-polymorphism

http://builds.enginehub.org/