MC-Test: Uma Ferramenta para testes de cobertura e testes de mutação

Trabalho de Conclusão de Curso

Engenharia da Computação

Daniel de França Figueroa
Orientador: Prof. Gustavo Henrique Porto de Carvalho

Universidade de Pernambuco
Escola Politécnica de Pernambuco
Graduação em Engenharia de Computação

Daniel de França Figueroa

MC-Test: Uma Ferramenta para testes de cobertura e testes de mutação

Monografia apresentada como requisito parcial para obtenção do diploma de Bacharel em Engenharia de Computação pela Escola Politécnica de Pernambuco – Universidade de Pernambuco.

Recife, Dezembro de 2016.

Escola Politécnica de Pernambuco
Graduação em Engenharia de Computação
Projeto de Final de Curso

MONOGRAFIA DE FINAL DE CURSO

Avaliação Final (para o presidente da banca)*

No dia 30 de 12 de 2016, às 9:00 horas, reuniu-se para deliberar a defesa da monografia de conclusão de curso do discente **DANIEL DE FRANCA FIGUEROA**, orientado pelo professor **Gustavo Henrique Porto de Carvalho**, sob título **MC-Test: uma ferramenta para testes de cobertura e testes de mutação**, a banca composta pelos professores:

Joabe Bezerra de Jesus Júnior
Gustavo Henrique Porto de Carvalho

Após a apresentação da monografia e discussão entre os membros da Banca, a mesma foi considerada:

[X] Aprovada [ ] Aprovada com Restrições* [ ] Reprovada

e foi-lhe atribuída nota: 8,0 (oito)

*(Obrigatório o preenchimento do campo abaixo com comentários para o autor)

O discente terá 5 dias para entrega da versão final da monografia a contar da data deste documento.

JOABE BEZERRA DE JESUS JÚNIOR

GUSTAVO HENRIQUE PORTO DE CARVALHO

* Este documento deverá ser encadernado juntamente com a monografia em versão final.

Agradecimentos

Agradeço a todas as pessoas que me ajudaram no neste trabalho, Família, amigos e principalmente ao meu pai, que mesmo distante, me incentiva a continuar.

Agradeço também ao meu orientador Gustavo Carvalho, que teve muita paciência comigo, me incentivou até o último minuto e não desistiu de mim.

Resumo

Testes são muito importantes para a criação e a manutenção de um software. Podemos Pode-se dizer que uma campanha de teste é de qualidade se a mesma possuir uma alta capacidade de revelar defeitos. Duas métricas que podem, indiretamente, contribuir para esta avaliação são a análise de cobertura e testes de mutação. Na primeira, avalia-se, de acordo com diferentes perspectivas, o quanto do código foi exercitado pela campanha de testes. Na segunda, verifica-se a capacidade dos testes revelarem defeitos inseridos sistematicamente. Atualmente, existem ferramentas que permitem realizar estas duas análises, porém não de forma integrada e intuitiva. Portanto, o objetivo desse trabalho é a criação de uma ferramenta capaz de realizar análise de cobertura e testes de mutação em suítes de teste a fim de permitir a avaliação da qualidade desta suíte a partir destes dois critérios.

Abstract

Software testing is an important task for software development and maintenance. It is said that a testing campaign is of high quality if its capability of revealing faults is high. Two metrics can be considered to perform such an evaluation: coverage analysis and mutation testing. Based on different perspectives, the former analyses how much of the code has been exercised by the testing campaign. Differently, the latter evaluates the capability of revealing defects that are systematically created. Currently, there are tools to support these two techniques, however not in an intuitive and integrated way. Therefore, the goal of this work is to develop a software tool capable of performing both coverage analysis and mutation testing in order to enable quality evaluation of test suites based on these two criteria.

Sumário

Índice de Figuras

Índice de Tabelas

Capítulo 1

Introdução

Nos dias atuais, cada pessoa possui diferentes aparelhos tecnológicos, sejam eles celulares, televisores, computadores, entre outros, que ajudam na comunicação, facilitando tarefas, automatizando serviços, e produzindo conhecimento. A necessidade tecnológica da população só aumenta, pois os benefícios proporcionados por estes aparelhos são cada vez mais significativos, o que traz também a necessidade de tecnologias novas e melhores, aumentando assim a demanda por software de qualidade [9].

Em função da demanda e pela necessidade de uma pouca margem de tempo para chegar no mercado, existe uma necessidade para que o software seja desenvolvido de forma cada vez mais rápida. Esta rapidez pode implicar em um menor cuidado empregado durante o desenvolvimento, gerando, assim, produtos com mais falhas e menor qualidade [9].

Todo software minimamente complexo possui uma alta probabilidade de apresentar falhas. No entanto, o que se deseja evitar é a presença de falhas críticas, que afetem o bom funcionamento da tecnologia em questão. Por este motivo, normalmente, enquanto um software é desenvolvido, ele é testado pela própria equipe de desenvolvimento ou por uma equipe dedicada exclusivamente à criação e execução de testes [17].

Desta forma, a ideia de teste é fundamental para o desenvolvimento de software, em particular, para garantir uma boa qualidade do produto final, bem como manter esta durante a etapa de evolução e manutenção do mesmo. Testes são feitos para encontrar defeitos; ou seja, não são capazes de provar que uma implementação irá sempre funcionar como esperado [17].

Um desafio ao escrever ou gerar um conjunto (*suíte*) de testes é avaliar se esta suíte é adequada ao software em questão. Mais precisamente, diz-se que uma suíte de testes tem qualidade se ela possui uma chance alta de encontrar possíveis defeitos. Uma possível métrica neste sentido é avaliar o quanto do código de um software é exercitado pelos testes, essa métrica é também chamada de cobertura de testes [1].

Outra técnica que pode ser utilizada para avaliar a qualidade de uma suíte de testes é a geração de mutantes. Neste técnica, variações (mutantes) são geradas a partir do código original do programa; obtendo, assim, versões com possíveis defeitos [14]. Desta forma, pode-se avaliar a qualidade de uma suíte de testes a partir da quantidade de defeitos encontrados pelos testes nos mutantes gerados.

Apesar da existirem ferramentas que apoiam estas técnicas (avaliação de cobertura e análise de mutantes), tais como Code Cover[1] e MuJava [14], respectivamente, não existe uma ferramenta integrada e intuitiva que permita facilmente avaliar a qualidade de várias suítes de testes a partir destes critérios. Portanto, o problema de pesquisa deste trabalho é: como integrar de forma intuitiva e prática a avaliação de suítes de teste a partir de critérios de cobertura e análise de mutantes.

1.1 Objetivo do trabalho

Este trabalho tem como objetivo principal uma ferramenta de avaliação da qualidade de suítes de teste, considerando os conceitos de cobertura de testes e testes de mutação. O objetivo geral desdobra-se nos seguintes objetivos específicos:

[1] LUDEWIG, Jochen.CodeCover. <http://codecover.org/> Acesso em: 27 Set. 2016.

- Estudo dos conceitos de testes em geral;
- Estudo dos conceitos de cobertura de testes;
- Estudo dos conceitos de testes de mutação;
- Estudo de ferramentas existentes para análise de cobertura e testes de mutação;
- Planejamento do desenvolvimento da ferramenta;
- Desenvolvimento da ferramenta;
- Avaliação da ferramenta criada;

1.2 Resultados e impactos esperados

O principal resultado deste trabalho consiste no desenvolvimento de uma ferramenta de avaliação de suítes de testes considerando critérios de cobertura e quantidade de defeitos encontrados, estes gerados a partir de mutação.

Com esta ferramenta, espera-se apoiar a criação de suítes de testes de melhor qualidade, uma vez que será possível avaliar sistematicamente a qualidade de um conjunto de testes. Desta forma espera-se também melhorar a qualidade do produto (software) final.

1.3 Estrutura do trabalho

Este trabalho encontra-se estruturado da seguinte maneira: o Capitulo 2 apresenta uma visão geral sobre os assuntos teóricos necessários para o entendimento do que foi feito como objetivo deste trabalho. O Capítulo 3 apresenta a ferramenta idealizada como parte do objetivo deste projeto, mostrando como esta foi planejada e desenvolvida e comparando-a também com outras ferramentas similares. O Capitulo 4 trata da avaliação da ferramenta criada, ilustrando o uso da

mesma no contexto de um projeto fictício (ilustrativo), mas também considerando um projeto extraído do gitHub[2]. Por fim, o Capítulo 5 apresenta as conclusões deste trabalho e discute possíveis trabalhos futuros.

[2] GitHub. <https://github.com/> Acesso em: 01 Jan. 2016.

Capítulo 2
Referencial Teórico

Este capítulo apresenta a fundamentação teórica necessária para o entendimento desse trabalho. Inicialmente na Seção 2.1 é exibido um resumo do RUP, Na Seção 2.2 será mostrado os conceitos de testes. Na Seção 2.3 é explicado o conceitos de cobertura de testes e ferramentas que executam esse tipo de teste. A Seção 2.4, apresenta a abordagem de teste baseada em mutação junto a análise de ferramentas existentes para a execução desses testes.

2.1 RUP

O RUP (*Rational Unified Process*) é uma metodologia de desenvolvimento de software tradicional e bem definida, que propõe tornar mais eficiente e preditível a elaboração um software. O RUP fornece algumas práticas para cada etapa do desenvolvimento de software, algumas dessas práticas são [12]:

- Arquitetura baseada em componentes
- Desenvolvimento interativo e incremental
- Gerenciamento e controle das mudanças
- Gerenciamento de Requisitos
- Modelagem Visual
- Verificação continua da qualidade

As práticas citadas a cima são utilizadas no RUP focados em marcos (*milestones*), que são etapas no desenvolvimento que possuem valores agregador, estas etapas estão listadas abaixo [12]:

- Fase de concepção: fase com foco nos requisitos para o estabelecimento do escopo do sistema e criação de documentos relacionados ao sistema.

- Fase de Elaboração: fase com foco na modelagem do sistema a partir dos documentos gerados na fase de concepção.

- Fase de Construção: fase com foco no desenvolvimento do sistema podendo ou não ter testes inclusos.

- Fase de Transição: fase com foco na implantação do sistema e testes com acompanhamento da qualidade junto ao cliente.

2.2 Testes

Testes possuem entradas e saídas, e, até para softwares simples, a quantidade de combinações de entradas tende a ser muito grande. Tornando, desta forma, inviável testar todas estas combinações. Por outro lado, nem todas as combinações são relevantes, pois combinações diferentes podem exercitar o código de uma mesma forma e, assim, não aumenta as chances de encontrar defeitos [17].

Para a análise de um código é necessário fazer inúmeros testes, esses testes podem ser agrupados em *suite* de testes, sendo possível agrupar testes por características especificas. A execução de uma *suite* de testes é o mesmo que rodar cada teste individualmente.

2.3 Cobertura de testes

Cobertura de testes é uma técnica para verificar o quanto de um código o teste está exercitando; ou seja, quais comandos estão sendo testados, quais regiões do código o teste não cobriu, como também quais ramificações do código o teste está exercitando [11].

A avaliação de cobertura de uma suíte de testes pode ser feitas por várias abordagens diferentes, cada uma delas possui seus prós e contras. A seguir, são descritas as principais categorias de critérios de cobertura.

- Critérios baseados em grafos: É uma abordagem que avalia a cobertura de um código tratando-o com um grafo, onde cada nó é uma parte do código, tendo como resultado a avaliação da cobertura dos nós de um dado programa[11].

- Critérios baseados em lógica: avalia especificamente os predicados presentes no código; por exemplo, em condições de comandos *if* e *while*, entre outros. O intuito é ter testes que avaliem diferentes combinações de valores para as cláusulas que compõem estes predicados [11].

- Critérios baseados no particionamento de entradas: avalia o código a partir das entradas que são fornecidas para o mesmo. O objetivo é particionar os valores que podem ser passados como entradas em classes de equivalência (todos os valores de uma classe exercitam de forma similar o código). Desta forma, não se faz necessário considerar nos testes todos os valores, mas somente alguns representantes de cada uma das classes [11].

2.3.1. Resultados de cobertura de testes

Os resultados de uma avaliação de cobertura de testes fornecem à equipe de testes informações que permitem avaliar se é necessário aumentar a quantidade de testes feitos por uma suíte de testes. Considerando os critérios de cobertura antes apresentados, podem-se obter as seguintes informações:

- Quais nós do códigos foram acessados;

- Quais ramificações do código foram acessadas;

- Quais condições lógicas foi usada em cada transição do código;

- Quais iterações em loops foram executadas;
- Quais métodos foram avaliados e quais não foram.

2.3.2. Ferramentas de cobertura de testes

A seguir, serão mostradas algumas ferramentas que avaliam suítes de testes, verificando o nível de cobertura em relação aos seus códigos testados.

I. JaCoCo

O JaCoCo (*Java code coverage*) [16] é uma ferramenta de análise de cobertura que pode ser utilizada por via de *ant scripts* [22]. Seus resultados podem ser vistos nos relatórios gerados pela ferramenta em xml ou em csv, uma vez que não possui interface gráfica. O *ant script*, para o funcionamento do JaCoCo, é dividido em 3 etapas:

- Compilação: Usando a tag "*compile*", o script compila todos os códigos e seus testes para um local indicado no próprio script.
- Test: Usando a tag "test", e importando as bibliotecas de testes, o script executa os testes e armazena os resultados.
- Criação de relatórios: Usando a tag "*report*", o script armazena todos os resultados obtidos nos testes em relatórios gerados em xml e csv.

Os relatórios do JaCoCo mostram resultados por classe testada, validando a cobertura de: instruções, ramificações, linhas e métodos.

A execução do JaCoCo pode ser feita usando *ant scripts,* o processo é feito em 4 etapas são elas: Inicialização, compilação, execução de testes, geração de relatório. Abaixo segue um exemplo da etapa de compilação.

Figura 1. Exemplo *ant script* I

```
1 <target name="compile" depends="clean">
2      <javac srcdir="${source.dir}"    destdir="${destination.dir}"
debug="true" includeantruntime="false"
3            classpath="./lib/junit.jar"/>
4 </target>
```

O exemplo acima mostra a compilação de códigos com extensão .java que estão armazenados no diretório "*source.dir*". As classes geradas deverão ser armazenadas em "*destination.dir*". Tanto "*source.dir*" quanto "*destination.dir*" são endereços previamente inseridos como variáveis no script como visto abaixo.

Figura 2. Exemplo *ant script* II

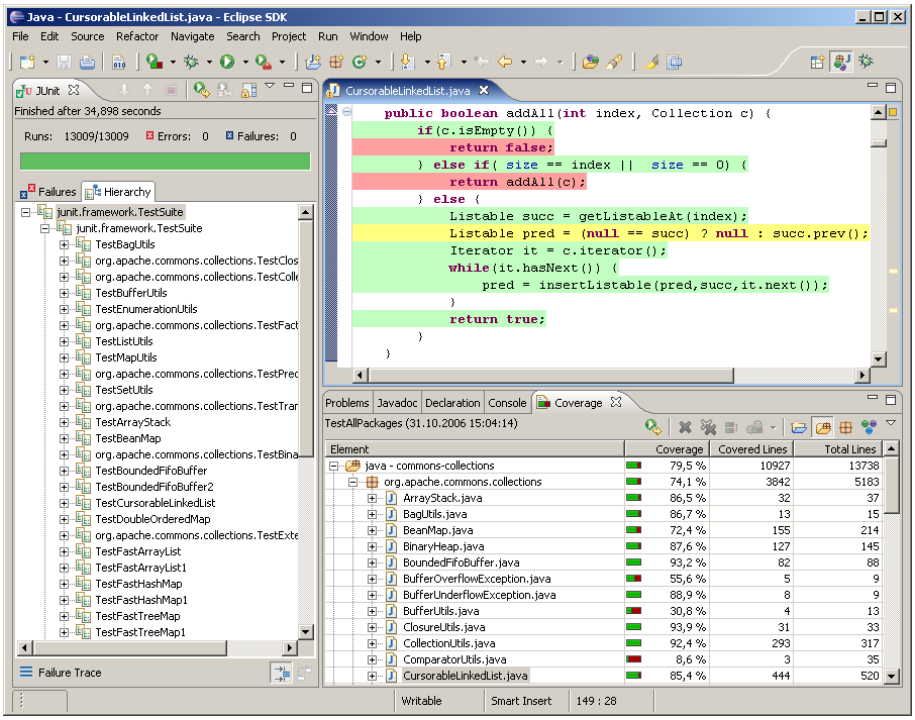
```
1 <property name="source.dir" location="./src"/>
2 <property name="destination.dir" location="./bin"/>
```

II. Eclemma

O EclEmma [15] é uma ferramenta de cobertura de teste baseada na biblioteca JaCoCo [16] (*Java code coverage),* feita para o Eclipse [8] com o objetivo de facilitar a visualização das informações de cobertura diretamente do ambiente do Eclipse, sem a necessidade de fazer qualquer alteração (instrumentação) no código original ou artificio extra.

Como pode-se ver na Figura 3, a ferramenta exibe graficamente quais linhas do código foram exercitadas pelos testes considerados, também mostra as ramificações que foram cobertas, além das condições que foram tomadas para seguir nessas ramificações [15]. Como é uma ferramenta para ser usada com o Eclipse, sua utilização é muito simples. Após instalado, é só abri-lo junto ao Eclipse e escolher quais testes deseja considerar para avaliar a cobertura.

Figura 3. Interface Gráfica da Ferramenta: Eclemma



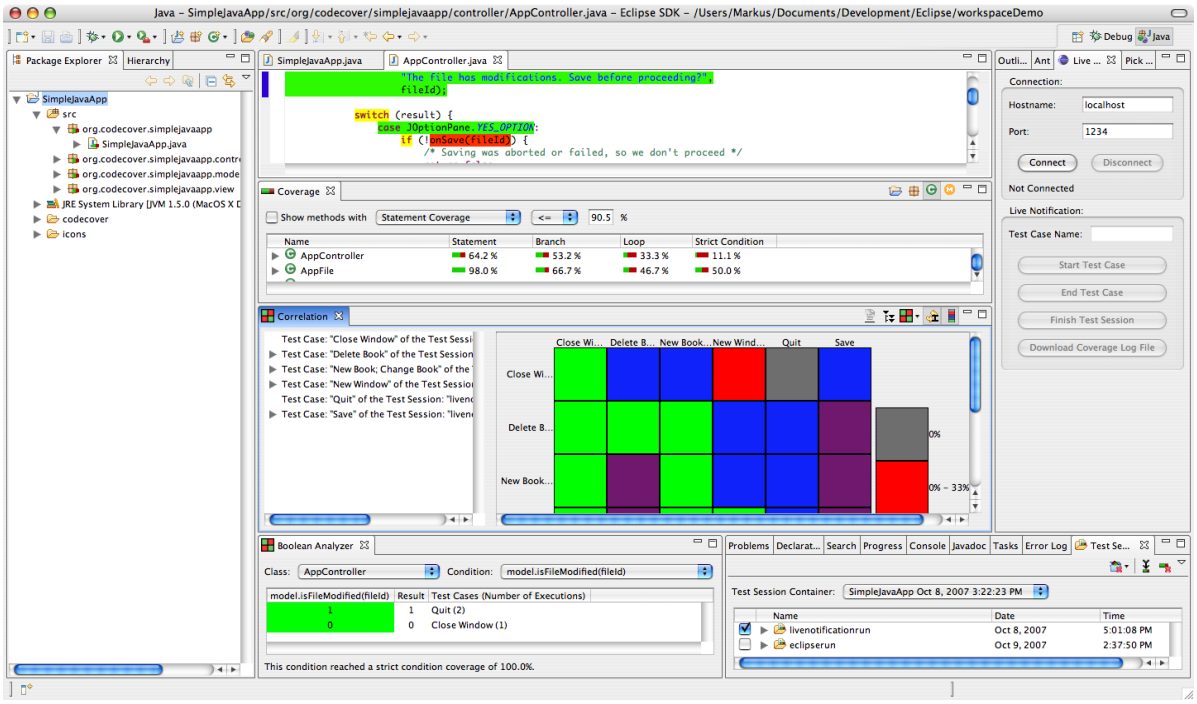
Fonte: <http://www.eclemma.org/>

III. CodeCover

CodeCover [21] é uma ferramenta com o mesmo objetivo que o EclEmma, porém com mais funcionalidades. Além de mostrar a cobertura de cada linha do código, também consegue fazer testes de cobertura em branches como mostrado na Figura 4. É uma ferramenta feita para Eclipse [8], também consegue ser utilizado por comandos de linha de código e *ant script [22], assim como o JaCoCo, e diferentemente do EclEmma, que não possui essas funcionalidades*.

Como uma ferramenta do Eclipse [8] o CodeCover funciona basicamente como o EclEmma, sem a necessidade de fazer alteração do código original. Em seu preparo é necessário alocação para pastas especificas, compilações do código fonte e testes antes de poder utilizar qualquer uma das ferramentas.

Figura 4. Interface Gráfica da Ferramenta: Code Cover



Fonte: < http://codecover.org/>

IV. Outras Ferramentas de Cobertura de Testes

Existem outras ferramentas para cobertura de testes, algumas de código aberto, outras proprietárias. Cada uma possui suas particularidades, seus pontos positivos e suas limitações. Na Tabela 1 resumem-se as principais características destas ferramentas. Esta tabela mostra quais ferramentas são gratuitas, quais aceitam testes no formato Junit [10], mostra também quais delas possuem ferramentas próprias e quais métodos de cobertura cada uma considera. Por fim, pode-se visualizar que nenhuma das ferramentas pesquisadas consegue rodar múltiplos testes simultaneamente para fins de comparação, como pode ser vistos na coluna destacada na Tabela 1.

Tabela 1. Tabela de ferramentas de cobertura de testes

	Atlassian [2]	Cobertura [6]	Eclemma [15]	JaCoCo [16]	Jcov [3]	Code Cover [21]	PIT [7]
Cobertura por linha	✗	✓	✓	✓	✗	✗	✓
Cobertura por método	✗	✗	✓	✓	✓	✓	✗
Cobertura por Ramificação	✓	✓	✓	✓	✓	✓	✗
Teste de Mutação	✗	✗	✗	✗	✗	✗	✓
Plug-in para o Eclipse	✓	✓	✓	✗	✓	✓	✗
Ferramenta própria	✓	✓	✗	✓	✓	✓	✓
Criação de relatórios	✓	✓	✓	✓	✓	✓	✓
Múltiplos testes	✗	✗	✗	✗	✗	✗	✗
Aceita testes JUnit	✓	✗	✓	✓	✗	✓	✓
Possui versão gratuita	✗	✓	✓	✓	✓	✓	✓

Fonte: adaptado de <https://confluence.atlassian.com/display/CLOVER/Comparison+of+code+coverage+tools>;. Acesso em 03 de Dezembro de 2016.

2.4 Testes de mutação

Teste de mutação é uma técnica que pode ser utilizada para guiar a escrita de testes, ou seja, como um critério de cobertura, como também para avaliar a qualidade de testes previamente escritos ou gerados. A técnica consiste em introduzir sistematicamente defeitos no código a ser testado para avaliar a capacidade dos testes em detectar tais defeitos. Desta forma, pode-se medir a qualidade de uma suíte de testes a partir da quantidade de defeitos em mutantes que esta suíte consegue identificar. Quantos mais defeitos forem encontrados, melhor a qualidade da suíte em avaliação [1].

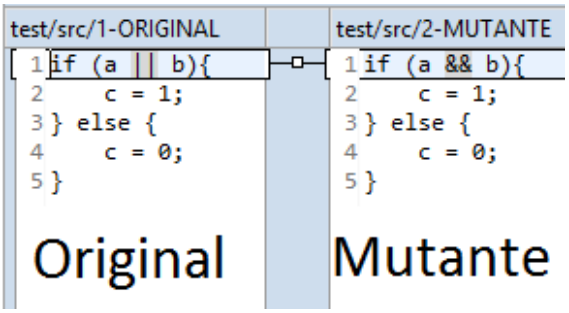
A técnica de teste de mutantes é complexa e custosa, pois tipicamente envolve a execução de uma grande quantidade de testes em uma grande quantidade de mutantes, não sendo, portanto, possível aplicá-la manualmente. Desta forma, faz-se necessária a automatização da mesma, realizando automaticamente as seguintes ações: criação de mutantes, execução do teste nos mutantes, classificação dos mutantes e avaliação dos resultados. Nas próximas seções, estas ações são detalhadas [1].

2.4.1. Criação de mutantes

Este é o primeiro passo da técnica de testes por mutação. É quando criamos mutantes a partir de um código fonte. Dependendo do tamanho deste código, milhares mutantes podem ser gerados.

Um mutante é uma versão de um código que possui uma única modificação. Esta mudança pode não ter impacto no comportamento do programa, mas também pode gerar alterar completamente o comportamento original. Em alguns casos, a modificação gerada pode produzir um código que não compila. Um exemplo de mutante pode ser visto na Figura 5: a condição, antes composta por uma disjunção (||), agora é composta por uma conjunção (&&) de cláusulas.

Figura 5. Exemplo de Mutante



Mutantes são criados baseados em regras. Estas regras são chamadas de operadores, que fornecem quais tipos de defeitos (mudanças) serão inseridos em cada mutante criado. Classicamente, existem 2 tipos de operadores de mutação: os operadores de métodos e os operadores de classes. Operadores ditam quais partes

da gramática do código vão ser alteradas, quais defeitos vão ser inseridos ou quais fluxos vão ser mudados.

Na criação de mutantes, para um mesmo operador, múltiplos mutantes podem ser gerados. Isto ocorre pois, apesar de cada operador estar relacionado a uma única mudança, estas podem ocorrer em vários pontos do código. Portanto, quanto maior o código, maior a probabilidade de existir vários mutantes por operador. Logo, a criação de mutantes pode ser um processo relativamente lento, caso seja escolhida a criação de mutantes usando múltiplos operadores, lembrando que cada operador pode gerar vários mutantes.

2.4.2. Execução dos testes nos mutantes

Após a criação dos mutantes, inicia-se a avaliação da qualidade dos testes previamente criados. Para tanto, cada teste é executado em cada mutante com o intuito de encontrar defeitos. Neste momento, é que se percebe a dificuldade de desempenho associada à esta técnica. Se houver muitos testes e muitos mutantes, esta execução pode demorar muito tempo.

2.4.3. Classificação dos mutantes

Após a criação de mutantes, e execução dos testes sobre estes, é necessário classificar os mutantes em um dos três grupos a seguir:

1) Mutantes mortos: mutantes que tiveram problemas na sua execução, isto é, que ao ter sido aplicada a suíte de testes, não passou em algum dos testes, indicando que a mudança no mutante foi reconhecida pelo testes que estão sendo executados.

2) Mutante sobrevivente: são mutantes que ao terem sido testados pela suíte de testes a ser avaliada, passam em todos os seus testes; ou seja, são mutantes nos quais erros não foram encontrados, apesar deles existirem.

3) Mutantes equivalentes: são mutantes sobreviventes que, apesar da modificação inserida pelo operador de mutação, possuem um comportamento equivalente ao código original. Desta forma, não podem ser mortos por nenhuma suíte de testes.

Em resumo, durante a execução dos testes, caso um erro seja encontrado em um mutante, diz-se que o mesmo foi morto. Os mutantes que não forem mortos após a execução de todos os testes são tidos como mutantes sobreviventes. A sobrevivência de um mutante pode ser decorrente de uma baixa qualidade da suíte de testes, que não foi capaz de detectar o erro associado ao mutante, como também em função de um mutante dito equivalente (apesar de ter uma diferença em relação ao código original, esta diferença não altera o comportamento do programa) [1].

Após a classificação dos mutantes em um destes três grupos, tem-se a principal métrica da técnica de testes de mutantes, que mostra quantos mutantes sobreviveram e morreram dado um código base e operadores de mutação [1].

2.4.4. Avaliação dos resultados

Esta é a parte do processo em que se avalia a suíte de testes, ou seja, se ela tem qualidade ou não. Portanto, após a criação de mutantes, execução dos testes e classificação dos mutantes, avaliam-se os resultados obtidos. Caso o número de mutantes mortos seja pequeno, isto é um indicativo de que a suíte de testes avaliada pode não ter uma boa qualidade, pois ela está deixando muitos mutantes sobreviventes (potencialmente, não equivalentes). Esta avaliação define a métrica conhecida como escore de mutação [1]:

Figura 6. Formula do *Mutation Score*

$$\textit{Mutation Score} = \frac{\text{Quantidade de mutantes mortos}}{\text{Quantidade de mutantes gerados - Quantidade de mutantes equivalentes}}$$

Portanto, o escore de mutação é a proporção de mutantes mortos por mutantes gerados (não equivalentes), gerando um valor entre 0 e 1. Quanto mais próximo do valor 1, melhor a qualidade da suítes de testes em avaliação

2.4.5. Ferramentas de testes de mutação

Nas próximas Seções, são descritas duas ferramentas de testes de mutação: MuJava (Seção I) e VisualMutator (Seção II).

I. MuJava

É um sistema de mutação para programas em Java, criado pela *Korea Advanced Institute of Science and Technology (KAIST)* e pela *George Mason University*[14]*.* Esta ferramenta gera mutantes tanto de métodos, quando mutantes de classes (que possuem alterações específicas para programas orientados a objetos). Cada mutante gerado é baseado no sistema de operadores de mutação, que ditam as regras que a geração de mutantes vai seguir. A Figura 7 mostra a tela principal desta ferramenta, onde é possível escolher quais operadores serão aplicados.

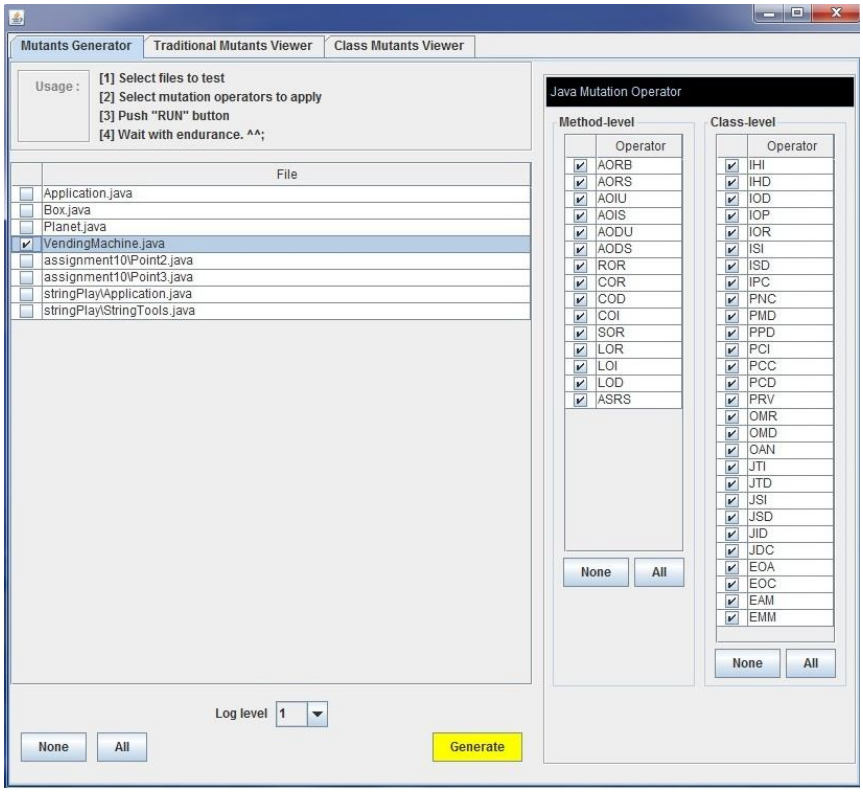
Alguns dos operadores de métodos suportados são:

- AOR (*Arithmetic Operator Replacement*): substitui um operador aritmético por outro por outro operador aritmético;
- ROR (*Relational Operator Replacement*): substitui um operador relacional, podendo trocar um predicado completo por "*True*" ou "*False*";
- COR (C*onditional Operator Replacement*): substitui um operador de condição (ramificação) por outro por outro operador de condição (ramificação).

Já alguns dos operadores de classes considerados são:

- AMC (*Access modifier change*): muda o acesso a variáveis e métodos, criando conflitos de acessibilidade;
- IOD (*Overriding method deletion*): deleta declarações de métodos sobrescritos, em uma subclasse.
- ISI (*super keyword insertion*): insere a palavra-chave "*super*" para que as referências de variáveis e métodos sejam referência de outras instancias de variáveis e métodos.

Figura 7. Interface Gráfica da Ferramenta: MuJava



Fonte: <https://cs.gmu.edu/~offutt/mujava/>

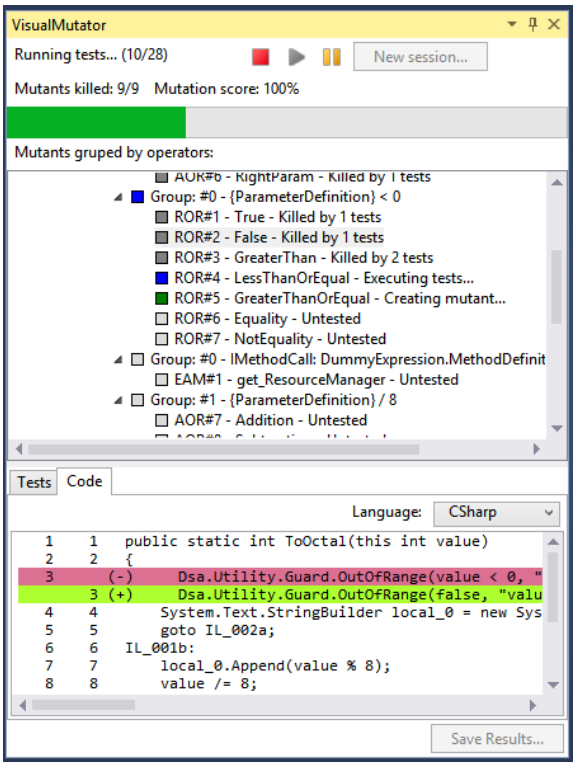
Após a geração de mutantes, a ferramenta executa os testes providos pelo usuário em cada um dos mutantes gerados, exibindo, em seguida, o escore de mutação da suíte de teste em avaliação. Para o correto funcionamento da ferramenta, é necessário posicionar o código original para pastas especificas antes de sua utilização. A ferramenta pode ser utilizada através da sua interface gráfica, mas

também por linha de comando, através do *MuScript*. A possibilidade de uso via linha de comando facilita a integração do MuJava com outras ferramentas [18].

II. VisualMutator

Esta é uma ferramenta integrada ao VisualStudio e, assim, gera mutantes no contexto de ambientes .NET. Assim como o MuJava, permite o usuário escolher quais operadores usar na geração de mutantes, possui uma fácil visualização das mudanças feitas em cada mutante, como pode ser visto na Figura 8. O escore de mutação também é calculado automaticamente.

Figura 8. Interface Gráfica da Ferramenta: Visual Mutator



Fonte: < https://visualmutator.github.io/web/>

Capítulo 3

MC-Test

Este capítulo apresenta a ferramenta (MC-Test) construída como parte deste trabalho. Na Seção 3.1, apresenta-se uma visão geral desta ferramenta. Em seguida, na Seção 3.2 está mostrado o modelo do processo de negócio aplicado, na Seção 3.3, mostra o diagrama de caso de uso de projeto do MC-Test, a Seção 3.4 mostra os protótipos de telas utilizado para idealizar a ferramenta, a Seção 3.5 apresenta o desenvolvimento do MC-Test, a Seção 3.6 mostra como usar a MC-Test. Finalmente a Seção 3.7 mostra exemplos da ferramenta em funcionamento em um estudo de casos.

3.1 Concepção da Ferramenta

É uma ferramenta criada como parte do objetivo central deste trabalho. O intuito desta é permitir de forma simples a avaliação da qualidade de suítes de testes, a partir da análise integrada de cobertura e testes de mutação.

A MC-Test possui esse nome por conta de que avalia a qualidade de uma suíte de testes a partir de dois tipos de análises, testes de mutação, daí vem o “M”, e análise de cobertura, “C”. A combinação destas duas técnicas é benéfica, pois uma complementa a outra. A análise de cobertura mostra o quanto do código foi exercitado por uma suíte de testes. No entanto, mesmo que um código tenha 100% de cobertura, os testes podem não ser de qualidade, em outras palavras, a suíte de testes pode ser trivial e repetitiva; ou seja, podem não ter a capacidade de encontrar uma quantidade relevante de defeitos.

Já os testes de mutação permitem avaliar quantos dos defeitos introduzidos sistematicamente forem detectados pela suíte de testes. Por outro lado, testes de mutação não garantem necessariamente que a suíte de teste está exercitando todo

o código existente. Basta que existam trechos do código que não foram modificados por nenhum operador de mutação. Desta forma, mesmo matando todos os mutantes, 100% de cobertura pode não ser alcançado.

A ferramenta MC-Test incorpora estas duas análises (testes de mutação e análise de cobertura) com o intuito de avaliar a qualidade de suítes de testes, criadas a partir de testes unitários escritos em JUnit[10], para códigos escritos na linguagem JAVA[19]. A seguir, destacam-se os diferenciais da ferramenta MC-Test.

- Múltiplas suítes de teste: a ferramenta possui a capacidade de avaliar múltiplas suítes de testes ao mesmo tempo, características que outras ferramentas não possuem, por exemplo, elas só executam testes de mutação considerando uma suíte de testes de cada vez.

- Cobertura e mutação: a ferramenta MC-Test faz 2 análises diferentes e retorna ao usuário a análise de cobertura junto com a análise dos testes de mutação.

- Automatização de tarefas: enquanto outras ferramentas necessitam que o usuário manualmente manipule arquivos e os posicione em locais específicos, a ferramenta MC-Test faz estas manipulações automaticamente para o usuário, facilitando assim o seu uso. Além disto, o uso da ferramenta é puramente a partir da sua interface gráfica, sem necessidade de manipulação via linha de comando.

3.2 Modelo do processo de negócio

O MC-Test tem o objetivo de automatizar tarefas de engenheiros de software e engenheiros de testes executando analises de cobertura e testes de mutação, a seguir estão as 6 etapas do processo do MC-Test:

1. Definição de parâmetros iniciais: etapa na qual o usuário indica qual código quer testar, quais testes usar no processo, quais operadores usar para o teste de mutação.

2. Inicialização dos testes: para a realização das etapas a seguir, são criados automaticamente *ant scripts*[22], que criam automaticamente diretórios, copiam os arquivos com os códigos e com as suítes de testes para estes diretórios, além de compilarem todos os arquivos que são necessários.

3. Análise de Cobertura: Também para a execução dos testes da análise de cobertura, é também utilizado um *ant script* para chamar a ferramenta JaCoCo, e então iniciar o teste de cobertura, gerando arquivos com os resultados.

4. Realização dos Testes de Mutação: Utilizando a biblioteca MuJava[13], o MC-Test gera os mutantes do código, e então, para cada suíte de testes, classifica eles em sobreviventes e mortos. Como a análise de equivalência é normalmente feita de forma manual, *a priori*, os mutantes não mortos são todos considerados como não-equivalentes; ou seja, uma análise pessimista.

5. Avaliação dos resultados: Após a conclusão de tanto os análise de cobertura quanto dos testes de mutação, o MC-Test concentra todas as informações geradas.

6. Exibição dos dados de saída: A ferramenta mostra em tela os resultados das análises para cada suíte de teste avaliada, separadamente. Dentre os dados exibidos, destacam-se: linhas cobertas, *branches* cobertas, percentual de cobertura, mutantes mortos e escore de mutação.

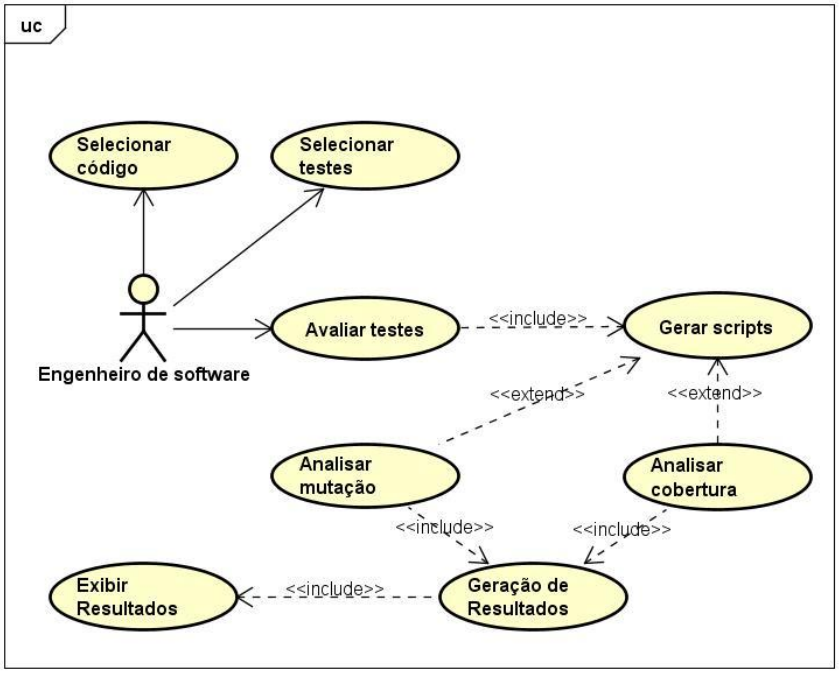
Inicialmente, levantou-se uma primeira versão das funcionalidades a partir da análise de ferramentas relacionadas, como as que foram discutidas no Capítulo 2.

Em seguida, Todas as informações adquiridas foram refletidas em diagramas de casos de uso, e juntadas num diagrama de caso de uso único, como pode ser visto mais detalhadamente na Seção 3.3. Após a distinção de cada caso de uso foi possível a criação de protótipos de telas que pode ser visto mais detalhadamente na Seção 3.4.

3.3 Diagramas de caso de uso

Para se obter uma visão das interações do sistema com seus usuários, foi utilizado o diagrama de Caso de Uso UML[20] e suas especificações. A Figura a seguir mostra o diagrama de Caso de Uso para a ferramenta MC-Test, onde mostra as ações do usuário junto ao sistema.

Figura 9. Diagrama de Caso de Uso

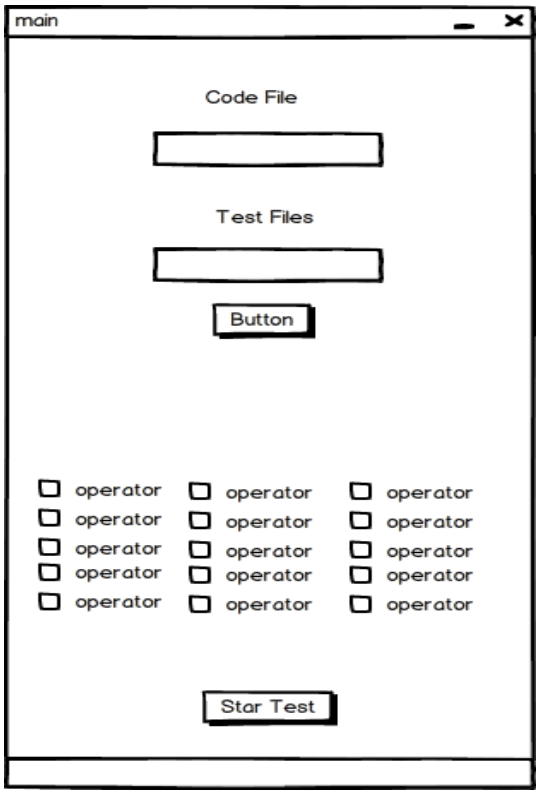


3.4 Protótipos de tela

Inicialmente, para auxiliar no levantamento das funcionalidades da ferramenta, foram criados protótipos de tela utilizado a ferramenta *Balsamiq Mockups* [3]. A escolha por esta ferramenta se deu em função de se ter conhecimento prévio da mesma.

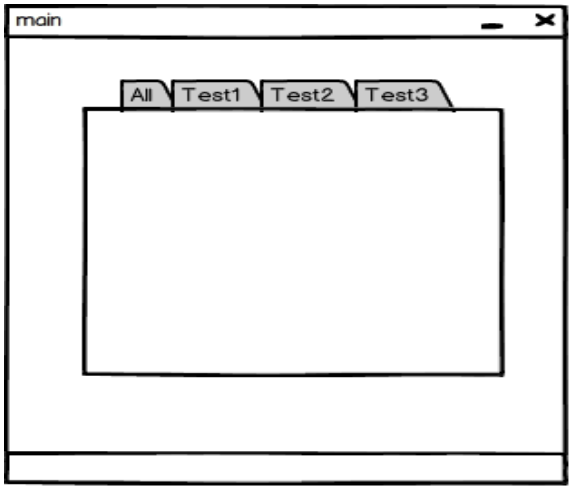
A Figura 10 mostra o protótipo de tela de entrada de dados, onde o usuário define os códigos que serão testados, as suítes de testes consideradas, e os operadores de mutação selecionados.

Figura 10. Protótipo de tela da tela de entrada de dados



A Figura 11 mostra o protótipo de tela de saída de dados, onde o usuário pode visualizar o resultado de cada teste tanto de teste de cobertura quanto de teste de mutação, de maneira que o usuário possa escolher qual suíte de teste deseja visualizar, podendo alternar entre as visualizações de cada suíte de teste.

Figura 11. Protótipo de tela da tela de resultados



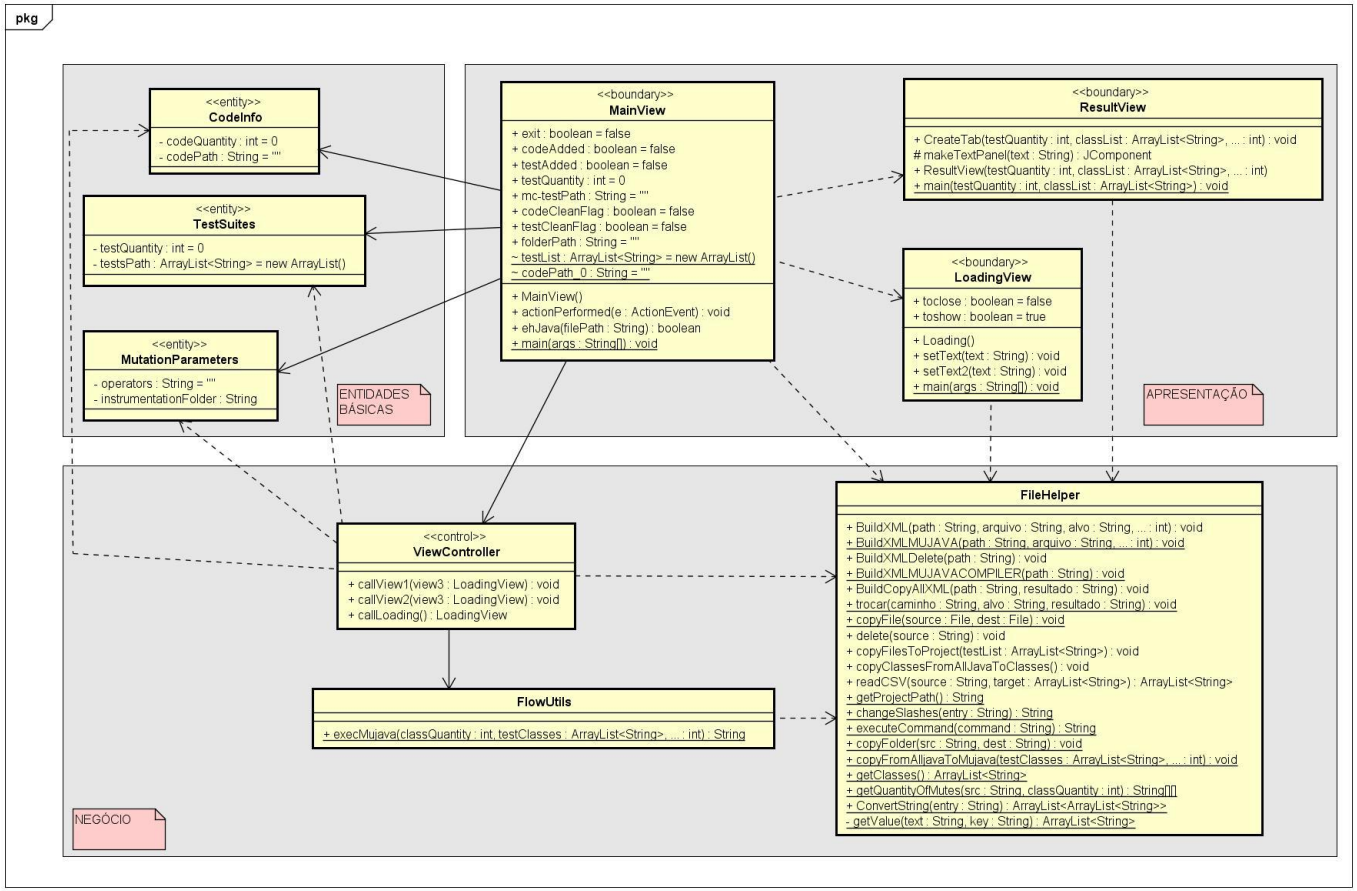
3.5 Construção da ferramenta

O processo de criação da MC-Test seguiu uma sequência de etapas com o intuito de garantir um bom planejamento e desenvolvimento da ferramenta. Escolheu-se a linguagem Java para desenvolver a ferramenta, considerando a facilidade de integração com o JaCoCo e o MuJava.

Utilizando o diagrama de casos de uso, que resume as principais funcionalidades da ferramenta e criando os diagramas de análise para cada funcionalidade, foi criado o diagrama classe para a fase de projeto, assim pode-se visualizar que classes precisam ser criadas, bem como seus atributos e relacionamentos.

O diagrama de classes foi criado usando a ferramenta Astah Profissional [5]. A escolha da ferramenta se deu pela sua facilidade de uso e por existir um conhecimento prévio da mesma.

Figura 12. MC-Test - Diagrama de classes



A Figura 12 mostra o diagrama de classes da ferramenta MC-Test. O diagrama encontra-se dividido em 3 partes, são elas:

- Entidades básicas: é o conjunto de entidades que possuem as informações necessárias para o funcionamento da ferramenta.
- Apresentação: nessa área encontra-se a interface gráfica com o usuário. Nela existem 3 classes, a MainView, que é a interface inicial para entrada de dados do usuário, a loadingView que mostra ao usuário as etapas até o termino do processamento, e finalmente a resultView que mostra ao usuário o resultado da ferramenta.
- Negócio: nessa camada tem-se uma classe de controle, que possui as regras e os fluxos necessários para a ferramenta funcionar, além de uma classe para auxílio na manipulação de arquivos e strings.

3.6 Fase de Transição do MC-Test

Nesta seção será mostrado o funcionamento da ferramenta na visão do usuário, desde sua instalação até a avaliação dos resultados gerados pelas análises realizadas. A ferramenta MC-Test pode ser encontrada e baixada no seguinte endereço: <https://github.com/kinatler/MC-Test>.

3.4.1. Fase de instalação do MC-Test

Antes de executar a ferramenta, é necessário configurar o *classpath* para referenciar a dependência de software de terceiros, como a utilização de *ant scripts* e do *MuJava*. Todas as bibliotecas dependentes já se encontram no diretório da própria ferramenta

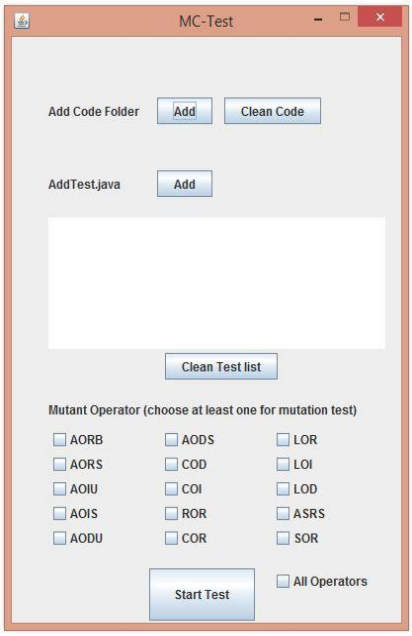
3.4.2. Fase de utilização do MC-Test

O funcionamento da ferramenta é feito a partir da seguinte sequência de passos pelo usuário:

1. Abrindo a ferramenta

Ao utilizar a ferramenta, a primeira tela da ferramenta visível para o usuário é a tela de entrada de dados como pode ser visto na Figura 13.

Figura 13. Tela de entrada de dados do MC-Test



2. Escolhendo a classe e testes a serem usados

Como pode ser visto na Figura 14, nas marcações 1 e 2, existem dois botões *add*, para indicar o código a ser testado, bem como as suítes de testes a serem consideradas, respectivamente. Ao apertar no botão 1, a ferramenta abrirá a tela de busca de arquivos (ver Figura 15) para que o usuário escolha a pasta em que o código a ser testado está. Ao apertar no botão 2, também irá aparecer a tela de busca de arquivos, como na Figura 15, mas desta vez o usuário irá procurar o arquivo que contém uma suíte de testes em JUNIT. O resultado desta ação é colocado na área de lista de testes a serem aplicados, como mostrado na Figura 16.

O usuário pode escolher quantas suítes de testes ele quiser. Caso o usuário tenha errado e queira refazer as escolhas do código ou de testes, pode- se apertar os botões 3 e/ou 4 para poder refazer as escolhas (ver Figura 14).

Figura 14. Entrada de dados

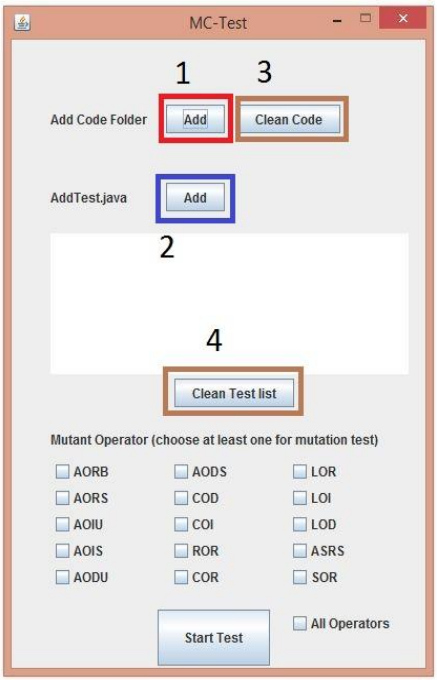


Figura 15. Escolha de arquivos

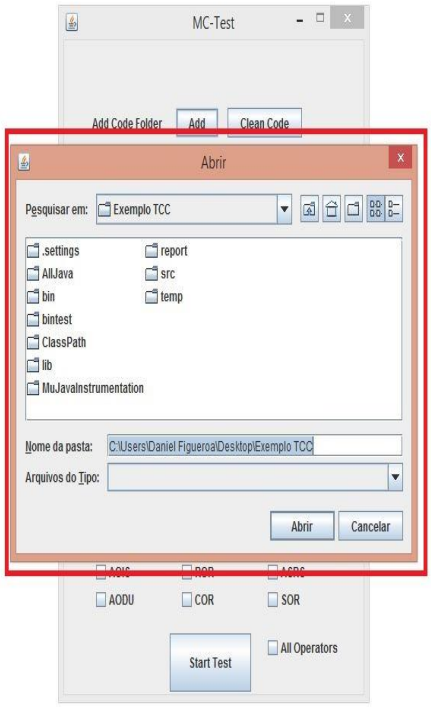
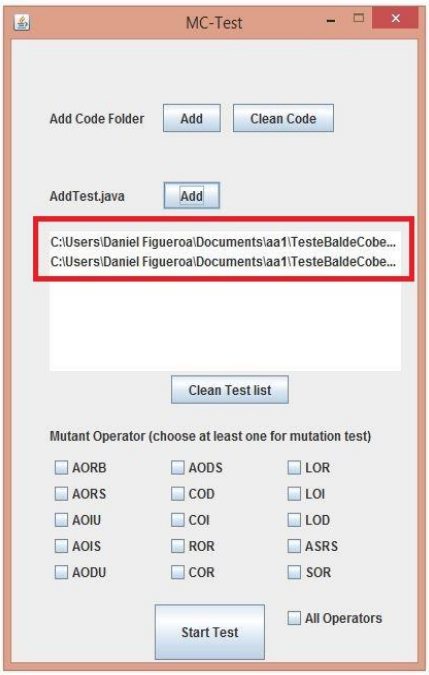


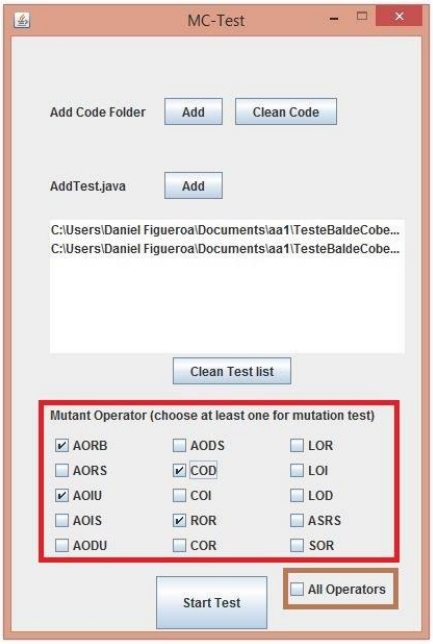
Figura 16. Validação da entrada de dados



3. Escolhendo os *operadores de mutação*

Para fazer os testes de mutação é necessário escolher quais operadores de mutação serão aplicados. Caso o usuário não marque nenhum, o teste de mutação não será considerado na análise de qualidade das suítes de testes. O usuário pode escolher quantos operadores quiser, e, caso queira selecionar todos ou desmarcar todos, o *checkbox* "*All Operators*" pode ser usado para esse propósito. Os operadores podem ser vistos na Figura 17.

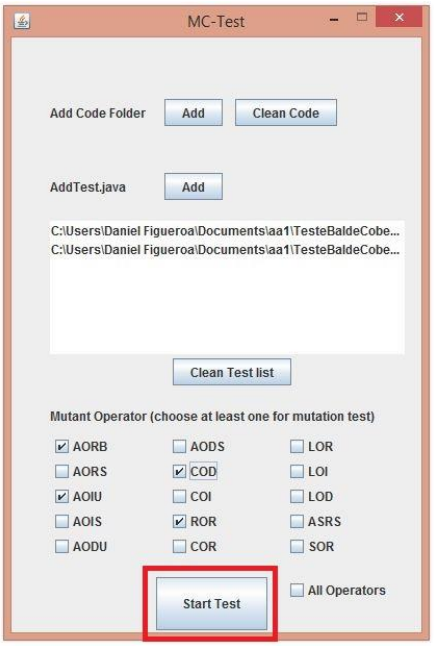
Figura 17. Escolha de operadores de mutantes



4. Inicializando o teste

Para iniciar a análise de cobertura e realizar os testes de mutação, aperta-se o botão "*Start Test*", mostrado na Figura 18. No entanto, para tanto, o usuário precisa ter escolhido previamente a pasta com código em Java do projeto, bem como ter escolhido as suítes de testes a serem avaliadas, como explicado nas etapas anteriores.

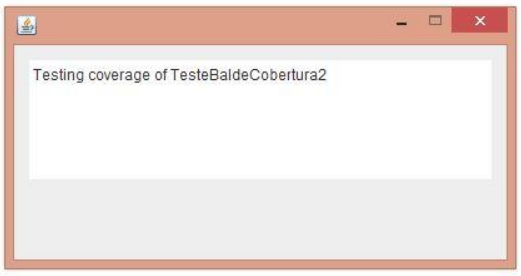
Figura 18. Inicializando teste



5. Esperando resultados

A ferramenta pode demorar um tempo para retornar a resposta, pois, dependendo do código que se queira testar, tanto a análise de cobertura quanto os testes de mutação podem levar muito tempo para serem realizados. Então, enquanto o usuário espera a conclusão desta etapa, a ferramenta mostra uma tela com as atividades que estão sendo realizadas (ver Figura 19).

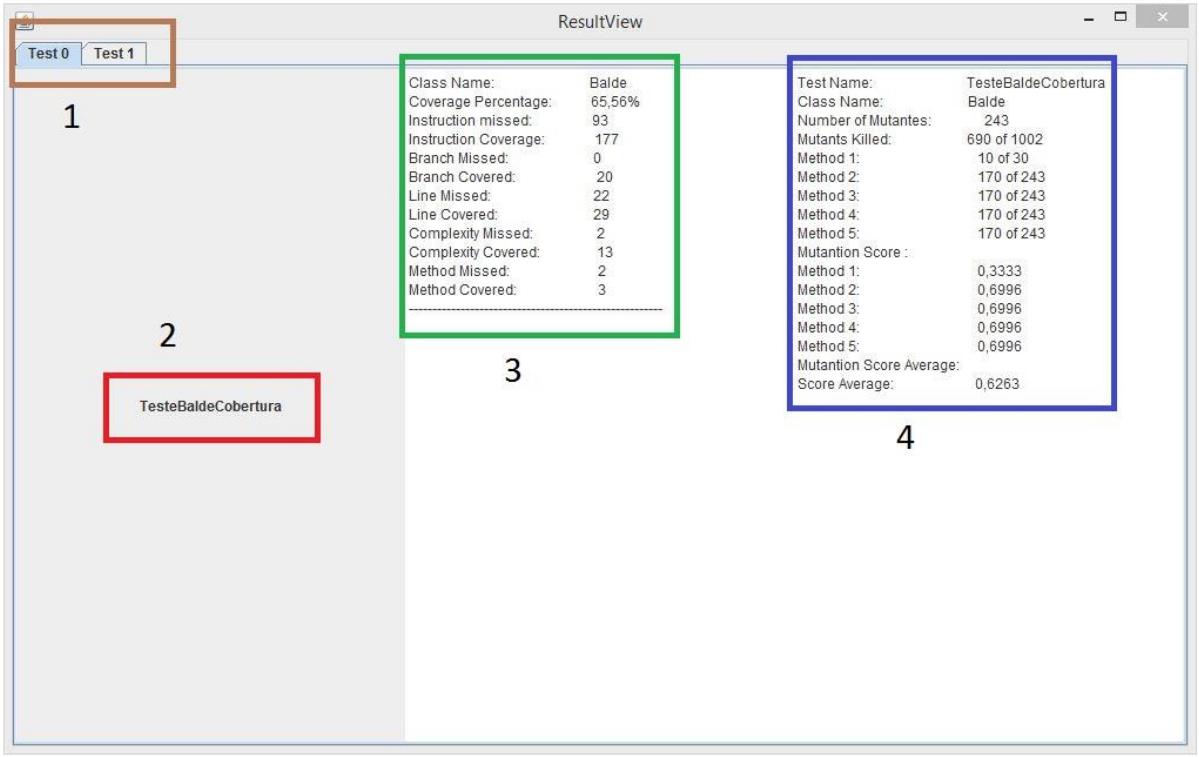
Figura 19. Tela de espera



6. Analisando resultados

Após a conclusão da etapa anterior, a ferramenta mostra a tela de resultados, mostrando o resultado das análises para cada suíte de testes. Desta forma, os resultados são agrupados em abas, como mostrado na Figura 20. Na área 1 é mostrado cada teste escolhido na tela inicial. Na área 2, mostra o nome daquele teste. Na área 3 são mostrados os resultados da análise de cobertura por suíte de testes selecionada na área 1.E por fim na área 4 estão os resultados dos testes de mutação da suíte de testes selecionada na área 1.

Figura 20. Resultados dos testes



7. Verificando arquivos criados

Uma vez que a ferramenta foi utilizada, e as análises foram feitas, o usuário pode encontrar logs com todas as informações exibidas na tela, além de informações extras, no diretório da ferramenta, na pasta "/*reports*". Espera-se que, com base nestas informações (e.g., percentual de cobertura, escore de mutação,

entre outras), o usuário possa decidir qual suíte de testes deve ser considerada para testar o seu projeto.

3.7 Estudo de Casos

Esta Capítulo apresenta alguns exemplos de utilização da ferramenta MC-Test. Na Seção 3.5.1, será considerado um exemplo simples: um problema de lógica envolvendo baldes. Na Seção 3.5.2, a ferramenta será utilizada para avaliar uma suíte de testes utilizada em um projeto disponível no gitHub.

3.5.1. Exemplo dos Baldes

Para um primeiro exemplo foi utilizado um código criado a partir de um problema de lógica envolvendo baldes. A classe Balde permite mover o líquido entre três baldes. Inicialmente, o terceiro balde, de capacidade 8 litros, está cheio. O segundo e primeiro baldes, com capacidade de 3 e 5 litros, respectivamente, encontram-se vazios. Nenhum dos baldes possui graduação de volume. O desafio é mover o líquido entre os baldes de forma a restar 4 litros no segundo e no terceiro balde. Este código possui originalmente uma suíte de testes criada em Junit. Esta suíte foi replicada e alterada para fazer com que sua cópia produzisse resultados piores que a primeira. O código e sua suíte de testes podem ser encontrados em anexo (ver Apêndice A).

Primeiramente, utilizou-se o JaCoCo[16] e o MuScript[18], de forma independente, para avaliar as duas suítes de testes. Em seguida, utilizou-se a ferramenta MC-Test para avaliar as mesmas suítes. Os resultados obtidos tanto na avaliação independente, como via MC-Test, foram iguais. Desta forma, validou-se a integração destas duas ferramentas via a ferramenta MC-Test. Os resultados obtidos foram os seguintes:

- Quantidade de classes testadas: 1
- Quantidade de suíte de testes testadas: 2

- Quantidade de linhas de códigos da classe testada: 92 linhas
- Quantidade de testes na suíte de teste 1: 9 testes
- Quantidade de testes na suíte de teste 2: 8 testes
- Quantidade de mutantes gerados: 243 mutantes
- Tempo para realização dos testes: 28 segundos
- Quantidade de operadores usados: 15 (todos)
- Resultado foi igual ao das ferramentas usadas separadamente: Sim

Para a primeira suíte de testes, obteve-se uma cobertura de 55,56% do código (considerando cobertura de linhas de código) e um escore de mutação de 0,6996. Já para a segunda suíte de testes, que foi piorada a partir da primeira, obteve-se uma cobertura de 55,19%, mas um score de mutação igual.

Com os resultados da ferramenta pode-se verificar uma perda na cobertura do segundo teste em relação ao primeiro, porém o escore de mutação se matem. Isso ocorre pois a segunda suíte de teste possui um teste a menos, como pode ser visto no Apêndice A. O fato do escore não mudar entre os dois testes não nos mostra realmente qual dos dois tem uma melhor qualidade e essa é a razão pelo qual ao ferramenta MC-Test faz essa dupla análise para melhor avaliar a qualidade dos testes.

3.5.2. Exemplo de uma calculadora simples

Para o segundo exemplo, foi utilizado um código proveniente do gitHub. O código[3] foi criado por Pierre-Henry Soria e Achintha Gunasekara possui sua própria suíte de testes. O código consiste em uma calculadora simples e todas as suas

[3] https://github.com/hczhcz/Java-Calculator-JUnit-Example

funcionalidades. O exemplo foi escolhido por possuir mais de uma classe e por possuir suíte de testes própria em JUnit [10].

Como no exemplo anterior, o código e seus testes foram avaliados utilizando tanto o JaCoCo e o MuScript de forma independente, como também através da ferramenta MC-Test. Os seguintes resultados foram obtidos:

- Quantidade de classes testadas: 3
- Quantidade de suíte de testes testadas: 1
- Quantidade de linhas de códigos da classe testada 1: 86 linhas
- Quantidade de linhas de códigos da classe testada 2: 10 linhas
- Quantidade de linhas de códigos da classe testada 3: 179 linhas
- Quantidade de testes em cada suíte de teste: 12 testes
- Quantidade de mutantes gerados: 93 mutantes
- Tempo para realização dos testes: 42 segundos
- Quantidade de operadores usados: 2
- Resultado foi igual ao das ferramentas usadas separadamente: Sim

A classe Calculador obteve uma cobertura de 75.46%, enquanto que as outras duas classes (a de interface gráfica e a SimpleJavaCalculator) não foram exercitadas pelos testes (0% de cobertura). O escore de mutação para a primeira classe foi de 0.6333, e como não tiveram testes para matar os mutantes gerados (421 mutantes e 13 mutantes) das outras classes, o escore de mutação dessas classes não foi calculado.

Capítulo 4
Conclusão e Trabalhos Futuros

A ferramenta desenvolvida e demonstrada neste trabalho (MC-Test) contribui para a avaliação da qualidade de suítes de testes, permitindo o usuário ter de forma simples várias informações sobre diferentes suítes de testes, a partir da análise de cobertura de código, bem como da técnica de testes de mutação. Uma das facilidades da ferramenta é permitir analisar suítes de testes a partir dessas duas técnicas sem exigir do usuário conhecimento no uso das ferramentas JaCoCo e MuJava, de forma independente.

O MC-Test foi a ferramenta criada como objetivo deste trabalho. Ela foi implementada em Java, com o objetivo de mostrar uma análise de cobertura de testes e a qualidade de uma dada suíte de testes. Para realizar esses tipos de análise a ferramenta utiliza a biblioteca JaCoCo para a análise da cobertura de testes, e a biblioteca MuJava para fazer testes de mutação para a avaliação de qualidade. O usuário ao utilizar o MC-Test estará em busca de avaliar suítes de testes em seus respectivos códigos.

4.1 Trabalhos Futuros

Apesar dos resultados obtidos, destacam-se os seguintes trabalhos futuros:

- Considerar dependências externas: Melhorar a ferramenta MC-Test para permitir o seu uso no contexto de códigos que possuem dependência de bibliotecas de terceiros.
- Melhorar a interface gráfica do usuário: exibir os resultados das análises não só de forma textual, mas também graficamente.

- Considerar outras linguagens: tornar o MC-Test capaz de avaliar testes em outras linguagens além de Java, e possibilitar a utilização de testes automáticos que não sejam testes JUnit.

- Implementar novos métodos de utilização do MC-Test: permitir o uso da ferramenta MC-Test via linha de comando ou como plug-in de outra ferramenta

- Experimentos com o MC-Test: fazer experimentações com a ferramenta focando em *benchmarks* e em projetos de compiladores.

- Estudos de usabilidade: fazer estudo de usabilidade para identificar oportunidades de melhoria na interface gráfica do MC-Test.

Referências

[1] AMMANN, PAUL; OFFUTT, JEFF. **Introduction to software testing**. [S.L.]: CAMBRIDGE UNIVERSITY PRESS, 2008.

[2] ATLASSIAN, **Atlassian Clover** <https://www.atlassian.com/software/clover>. Acesso em 20 de Dezembro de 2016.

[3] ATLASSIAN, **Jcov** <https://wiki.openjdk.java.net/display/CodeTools/jcov>. Acesso em 20 de Dezembro de 2016.

[4] BALSAMIQ STUDIOS, **Balsamiq Mockups**. Disponível em: <https://balsamiq.com/products/mockups/>. Acesso em 03 de Dezembro de 2016.

[5] CHANGE VISION, **Astah modeling**. Disponível em: <http://astah.net/>. Acesso em 03 de Dezembro de 2016.

[6] CHRISTOU, STEVE. **Cobertura** <http://cobertura.github.io/cobertura/>. Acesso em 20 de Dezembro de 2016.

[7] COLES, HENRY. **PITest** <http://pitest.org/>. Acesso em 20 de Dezembro de 2016.

[8] ECLIPSE FUNDATION, **Eclipse**. Disponível em: <https://eclipse.org/>. Acesso em 03 de Dezembro de 2016.

[9] JONES, Capers. **The technical and social history of software engineering**. 1 ed. [S.L.]: Addison-Wesley, 2013.

[10] JUNIT, **JUnit**. Disponível em: <http://junit.org/junit4/>. Disponível em 03 de Dezembro de 2016.

[11] KANER, CEM; FALK, JACK; NGUYEN, HUNG QUOC. **Testing Computer Software**. 2 ed.Wiley,1999.

[12] KRUCHTEN, PHILIPPE. **Introdução ao RUP: Rational Unified Process**. Ciência Moderna, 2003.

[13] MA1, Yu-Seung; OFFUTT, JEFF; KWON, YONG RAE. **MuJava: An Automated Class Mutation System.** <https://cs.gmu.edu/~offutt/mujava/>. Acesso em 20 de Dezembro de 2016.

[14] MA1, Yu-Seung; OFFUTT, JEFF; KWON, YONG RAE. **Software Testing, Verification and Reliability**, [S.L], jun. 2005.

[15] MOUNTAINMINDS GMBH, **Eclemma**. Disponível em: <http://www.eclemma.org/>. Acesso em 03 de Dezembro de 2016.

[16] MOUNTAINMINDS GMBH, **JaCoCo**. Disponível em: <http://www.eclemma.org/jacoco/>. Acesso em 03 de Dezembro de 2016.

[17] NETO, ARILO. Introdução a Teste de Software. **Revista Engenharia de Software**, [S.L], jun. 2010.

[18] OFFUTT, JEFF; DENG, LIN. **MuScript**. Disponível em: <https://cs.gmu.edu/~offutt/mujava/muscript/>. Acesso em 03 de Dezembro de 2016.

[19] ORACLE, **O que é Java?.** Disponível em: <https://www.java.com/pt_BR/about/whatis_java.jsp>. Acesso em 03 de Dezembro de 2016.

[20] RUMBAUGH, JAMES. JACOBSON, IVAR. BOOCH GRADY. **THE UNIFIED MODELING LANGUAGE REFERENCE MANUAL**. 2 ed. Addison-Wesley, 2004

[21] SCHMIDBERGER,RAINER. **CodeCover** <http://codecover.org/>. Acesso em 20 de Dezembro de 2016.

[22] The Apache Software Foundation,**Ant Script**. Disponível em: <https://ant.apache.org/>. Acesso em 20 de Dezembro de 2016.

[23] WYSOCKI, ROBERT K. **Effective Software Project Management**. 1 ed. Wiley, 2006.

Apêndice A

Exemplo dos Baldes

A1. Código do exemplo

```
public class Balde {
	private int maximo[];
	private int volume[];
	private int volFinalB2, volFinalB3;

	public Balde(int maxB1, int maxB2, int maxB3, int volFinalB2, int volFinalB3) {
		this.maximo = new int[3];
		this.maximo[0] = maxB1;
		this.maximo[1] = maxB2;
		this.maximo[2] = maxB3;
		this.volume = new int[3];
		this.volume[0] = 0;
		this.volume[1] = 0;
		this.volume[2] = maximo[2];
		this.volFinalB2 = volFinalB2;
		this.volFinalB3 = volFinalB3;
	}

	public void mover(int origem, int destino) throws Exception {
		if (origem == destino) {
			throw new Exception("Origem precisa ser diferente de destino!");
		} else {
			if (origem < 0 || origem >= 3 || destino < 0 || destino >= 3) {
```

```
                    throw new Exception("Origem e destino precisam estar entre 0..2!");
                } else {
                    if (volume[origem] == 0) {
                        throw new Exception("O balde origem está vazio!");
                    } else {
                        if (volume[destino] == maximo[destino]) {
                            throw new Exception("O destino já está cheio!");
                        } else {
                            if (volume[destino] + volume[origem] > maximo[destino]) {
                                int temp = (maximo[destino] - volume[destino]);
                                volume[origem] = volume[origem] – temp;
                                volume[destino] = maximo[destino];
                            } else {
                                volume[destino] = volume[destino]+volume[origem];
                                volume[origem] = 0;
                            }
                            if (volume[1] == volFinalB2 && volume[2] == volFinalB3) {
                                System.out.println("Problema resolvido!");
                            }
                        }
                    }
                }
            }
        }

        public int[] getVolume() {
            return volume;
        }
```

```
	public static void imprimirVolume(int[] volume) {
		System.out.println("[" + volume[0] + " " + volume[1] + " " + volume[2] + "]");
	}

	public static void main(String[] args) {
		Balde b = new Balde(3, 5, 8, 4, 4);
		Balde.imprimirVolume(b.getVolume());
		try {
			b.mover(2,1);
			Balde.imprimirVolume(b.getVolume());
			b.mover(1,0);
			Balde.imprimirVolume(b.getVolume());
			b.mover(0,2);
			Balde.imprimirVolume(b.getVolume());
			b.mover(1,0);
			Balde.imprimirVolume(b.getVolume());
			b.mover(2,1);
			Balde.imprimirVolume(b.getVolume());
			b.mover(1,0);
			Balde.imprimirVolume(b.getVolume());
			b.mover(0,2);
			Balde.imprimirVolume(b.getVolume());
		} catch (Exception e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}
	}
}
```

A2. Teste Junit 1:

```
import static org.junit.Assert.*;
import org.junit.Before;
import org.junit.Test;



public class TesteBaldeCobertura {
	private Balde b = null;

	@Before
	public void setUp() {
		b = new Balde(3, 5, 8, 4, 4);
	}

	@Test
	public void testBalde_getVolume() {
		int[] vol = b.getVolume();
		assertEquals("Balde1 inicializado incorretamente!", 0, vol[0]);
		assertEquals("Balde2 inicializado incorretamente!", 0, vol[1]);
		assertEquals("Balde3 inicializado incorretamente!", 8, vol[2]);
	}

	@Test
		public void testBalde_E1() {
			try {
				b.mover(0,0);
			} catch (Exception e) {
				assertEquals("Mensagem de erro diferente!",
				"Origem precisa ser diferente de destino!",
```

```
                    e.getMessage());
                    return;
            }
    }

    @Test
    public void testBalde_E2_1caso_true() {
        try {
            b.mover(-1,0);
        } catch (Exception e) {
            assertEquals("Mensagem de erro diferente!",
            "Origem e destino precisam estar entre 0..2!",
            e.getMessage());
            return;
        }
    }

    @Test
    public void testBalde_E2_2caso_true() {
        try {
            b.mover(3,0);
        } catch (Exception e) {
            assertEquals("Mensagem de erro diferente!",
            "Origem e destino precisam estar entre 0..2!",
            e.getMessage());
            return;
        }
    }
```

```
@Test
public void testBalde_E2_3caso_true() {
	try {
		b.mover(0,-1);
	} catch (Exception e) {
		assertEquals("Mensagem de erro diferente!",
		"Origem e destino precisam estar entre 0..2!",
		e.getMessage());
		return;
	}
}

@Test
public void testBalde_E2_4caso_true() {
	try {
		b.mover(0,3);
	} catch (Exception e) {
		assertEquals("Mensagem de erro diferente!",
		"Origem e destino precisam estar entre 0..2!",
		e.getMessage());
		return;
	}
}

@Test
public void testBalde_E3() {
	try {
		b.mover(0, 1);
	}  catch (Exception e) {
```

```
            assertEquals("Mensagem de erro diferente!",
            "O balde origem está vazio!",
            e.getMessage());
            return;
        }
    }

    @Test
    public void testBalde_E4() {
        try {
            b.mover(2, 1);
            b.mover(2, 1);
        }  catch (Exception e) {
            assertEquals("Mensagem de erro diferente!",
            "O destino já está cheio!",
            e.getMessage());
            return;
        }
    }

    @Test
    public void testBalde_Solucao() {
        try {
            b.mover(2,1);
            b.mover(1,0);
            b.mover(0,2);
            b.mover(1,0);
            b.mover(2,1);
            b.mover(1,0);
```

```
            b.mover(0,2);
        } catch (Exception e) {
            fail("Não deveria gerar uma exceção neste teste!");
        }
    }
```

A3. Teste Junit 2:

```
import static org.junit.Assert.*;
import org.junit.Before;
import org.junit.Test;


public class TesteBaldeCobertura {
    private Balde b = null;

    @Before
    public void setUp() {
        b = new Balde(3, 5, 8, 4, 4);
    }

    @Test
    public void testBalde_getVolume() {
        int[] vol = b.getVolume();
        assertEquals("Balde1 inicializado incorretamente!", 0, vol[0]);
        assertEquals("Balde2 inicializado incorretamente!", 0, vol[1]);
        assertEquals("Balde3 inicializado incorretamente!", 8, vol[2]);
    }

    @Test
```

```
	public void testBalde_E1() {
		try {
			b.mover(0,0);
		} catch (Exception e) {
			assertEquals("Mensagem de erro diferente!",
			"Origem precisa ser diferente de destino!",
			e.getMessage());
			return;
		}
}

@Test
public void testBalde_E2_1caso_true() {
	try {
		b.mover(-1,0);
	} catch (Exception e) {
		assertEquals("Mensagem de erro diferente!",
		"Origem e destino precisam estar entre 0..2!",
		e.getMessage());
		return;
	}
}

@Test
public void testBalde_E2_2caso_true() {
	try {
		b.mover(3,0);
	} catch (Exception e) {
		assertEquals("Mensagem de erro diferente!",
```

```
            "Origem e destino precisam estar entre 0..2!",
            e.getMessage());
            return;
        }
    }

    @Test
    public void testBalde_E2_3caso_true() {
        try {
            b.mover(0,-1);
        } catch (Exception e) {
            assertEquals("Mensagem de erro diferente!",
            "Origem e destino precisam estar entre 0..2!",
            e.getMessage());
            return;
        }
    }

    @Test
    public void testBalde_E2_4caso_true() {
        try {
            b.mover(0,3);
        } catch (Exception e) {
            assertEquals("Mensagem de erro diferente!",
            "Origem e destino precisam estar entre 0..2!",
            e.getMessage());
            return;
        }
    }
```

```
@Test
public void testBalde_E3() {
	try {
		b.mover(0, 1);
	}  catch (Exception e) {
		assertEquals("Mensagem de erro diferente!",
		"O balde origem está vazio!",
		e.getMessage());
		return;
	}
}

@Test
public void testBalde_E4() {
	try {
		b.mover(2, 1);
		b.mover(2, 1);
	}  catch (Exception e) {
		assertEquals("Mensagem de erro diferente!",
		"O destino já está cheio!",
		e.getMessage());
		return;
	}
}
```