Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Coordenação de Pós-Graduação em Informática

# Geração Automática de Testes com Objetos *Mock* Baseados em Interações

Sabrina de Figueirêdo Souto

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande - Campus I como parte dos requisitos necessários para obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: Ciência da Computação
Linha de Pesquisa: Engenharia de Software

Dalton Serey Guerrero
(Orientador)

Campina Grande, Paraíba, Brasil

**FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA CENTRAL DA UFCG**

S728g Souto, Sabrina de Figueirêdo

Geração automática de testes com objetos *Mock* baseados em interações / Sabrina de Figueirêdo Souto. — Campina Grande, 2010.

133 f. : il. col.

Dissertação (Mestrado em Ciência da Computação) – Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática.

Referências.

Orientador: Prof. Dr. Dalton Serey Guerrero.

1. Teste de Software. 2. Teste de Unidade. 3. Objetos *Mock*. 4. Análise Estática. 5. Análise Dinâmica. I. Título.

CDU – 004.415.532(043)

"GERAÇÃO AUTOMÁTICA DE TESTES COM OBJETOS MOCK BASEADOS EM INTERAÇÕES"

SABRINA DE FIGUEIRÊDO SOUTO

DISSERTAÇÃO APROVADA EM 31.08.2010

DALTON DARIO SEREY GUERRERO, D.Sc
Orientador(a)

JORGE CESAR ABRANTES DE FIGUEIREDO, D.Sc
Examinador(a)

ROBERTA DE SOUZA COELHO, Drª
Examinador(a)

CAMPINA GRANDE - PB

# Resumo

O objetivo de um teste de unidade é verificar o funcionamento de um software e procurar defeitos na menor parte testável, de forma isolada do restante do sistema. Porém, este isolamento nem sempre é possível devido às interações entre a a classe sob teste - CUT (*Class Under Test*) e seus colaboradores, fazendo-se necessário o uso de objetos *mock* para simular os colaboradores, em um mecanismo onde a CUT não sabe se ela está interagindo com colaboradores reais ou objetos *mock*. Desta forma, objetos *mock* podem ser de grande utilidade na escrita de testes de unidade, uma vez que isolam a CUT, tornando o teste mais eficaz e eficiente. Entretanto, escrever testes com objetos *mock* pode ser uma tarefa custosa, tediosa e repetitiva, mesmo com o suporte de *frameworks*, como o EasyMock [34] por exemplo, devido às atividades necessárias ao seu funcionamento. Dada a necessidade de se utilizar objetos *mock* nos testes de unidade e seus benefícios, e considerando a dificuldade envolvida no desenvolvimento e manutenção deste tipo de código, propomos e avaliamos uma técnica para automatizar o desenvolvimento de testes com objetos *mock*, gerando-os automaticamente para um determinado cenário de teste. Esta técnica tem por base a identificação das interações entre a CUT e seus colaboradores em um dado cenário de teste, através da análise estática e dinâmica de código. Após essas análises, a técnica provê a geração de código de teste com objetos *mock* para os colaboradores e interações identificadas. O código resultante consiste em uma nova versão do código do cenário de teste inicial, onde os colaboradores da CUT são substituídos por objetos *mock*, bem como suas interações. Considerando o baixo custo de se gerar automaticamente o código para os objetos *mock*, não se faz necessária a reutilização deste código, o qual poderá ser gerado novamente a cada modificação. Para avaliar nossa técnica realizamos um estudo experimental comparativo entre a forma manual e a automática de se desenvolver testes com objetos *mock*, através da observação do esforço empregado em tais atividades e o nível de cobertura de interações do código resultante, que tenta assegurar que o código gerado possui pelo menos a mesma precisão do código inicial. Os resultados da avaliação evidenciam que nossa técnica foi capaz de reduzir o esforço empregado no desenvolvimento de testes com objetos *mock*, resultando em um código melhor e mais preciso comparado ao manual.

# Abstract

The main purpose of unit tests is to verify the correct work of software and search for defects in the smallest part of the system, in isolated way. However, this isolation not always is possible due to the relationship among the CUT (Class Under Test) and its collaborators, needing the use of mock objects to simulate the collaborators behavior, by a mechanism that the CUT has no science if it is interacting with real collaborators or mock objects. As a result, mock objects can be very useful to write unit tests because they isolate the CUT, improving both efficiency and effectiveness of unit testing. On the other hand, writing tests that use mocks can be a tedious, repetitive, and costly task, even with the support of frameworks, due to the necessary procedures for it working. Given the high importance in the use of mock objects and its benefits, and considering the mentioned drawbacks, we propose a technique that automates the development of tests with mock objects, by automatically generating them for a given testing scenario. This technique is based on automatically identifying interactions among the CUT and its collaborators in a given testing scenario, via both static analysis and dynamic analysis of the code. After these analyses, the technique generates code of test with mock objects for those collaborators and the identified interactions, resulting on a new version of the initial testing scenario code, where the collaborators are replaced by mock objects as well its interactions. Considering the low cost in automatically generate test code with mock objects, it is not necessary maintain this code, which can be generated for each change in the testing scenario. In order to evaluate our technique, we performed a comparative study, by means of experimentation, between the manual and automatic way of develop tests with mock objects, by observing the spent effort (development time and size of produced code) on those procedures and the interaction coverage level of the resulting code, which tries to assure that the generated code contains in minimum the same quality of the input code. The evaluation results demonstrate that our technique was able to reduce the spent effort on developing tests with mock objects, resulting on a better and more precise code when compared to the one manually developed.

# Agradecimentos

Primeiramente a Deus, por me sustentar e me dar muito mais do que mereço, a Ele toda honra e toda glória.

A minha família, marido e filho, por me amarem o suficiente para tolerar minhas ausências, para encorajar e aplaudir esta conquista, dedico esta conquista a vocês.

Aos meus pais, a vóvó Lála e a Preta, que abdicaram um pouco de suas vidas para se dedicarem a mim. Tudo que eu posso fazer é agradecer e oferecer esta conquista a vocês.

A meus avôs, tios, primos e irmão pelo apoio e torcida. Em especial pela minha cunhada Vânia, pela força e incentivo desde o início do mestrado.

Ao meu orientador, Dalton Serey, por me guiar e apoiar nesta difícil caminhada.

A todos que me ajudaram a concretizar idéias e implementar o trabalho: Pryscilla Dora (minha co-orientadora "extra-oficial" e amiga), Alan Farias, Marcelo Iury, Alexandro, Lesandro, Ayla, Lívia, Amanda, Camila, Isabella, todos os que participaram dos experimentos, e todos que contribuíram de alguma forma para o sucesso desse trabalho.

A HP pelo apoio financeiro.

Aos professores e funcionários da COPIN e do DSC.

# Conteúdo

# Lista de Figuras

# Lista de Tabelas

# Lista de Códigos Fonte

# Capítulo 1

# Introdução

O termo "Qualidade de Software" tem sido introduzido ao longo de todo o processo de desenvolvimento através das atividades de V&V - Verificação e Validação, com o objetivo de minimizar a ocorrência de defeitos e riscos associados. Dentre as técnicas de verificação e validação, a atividade de teste é uma das mais utilizadas, constituindo-se em um dos elementos para fornecer evidências da confiabilidade do software em complemento a outras atividades.

Testar um software significa verificar, através de uma execução controlada, se o seu comportamento corresponde ao especificado. O objetivo principal desta tarefa é encontrar falhas, as quais podem ser originadas por diversos motivos, dentre eles podemos citar: requisitos impossíveis de ser implementados devido às limitações de hardware ou software; defeito em algum algoritmo; código complexo e difícil de manter; defeitos na base de dados; problemas de comunicação em sistemas distribuídos; mudança de tecnologia, muitas interações no sistema, etc.

Ainda no contexto de encontrar falhas, um motivo bastante relevante para suas ocorrências é a série de transformações pelas quais os requisitos passam até chegar o estágio de codificação, que são haja desvios no código em relação ao que foi especificado nos requisitos, contribuindo para a ocorência de falhas. Desta forma, se faz necessária a realização de testes em diferentes níveis em paralelo ao desenvolvimento de software, ou seja, para cada atividade do desenvolvimento há uma atividade de teste correspondente, com o propósito de evitar que defeitos sejam introduzidos ao longo do processo de desenvolvimento. Para dar suporte a essa abordagem, se utiliza o modelo V [23; 24;

30], ilustrado na Figura 1.1, que considera quatro níveis de teste: teste de unidade, teste de integração, teste de sistema e teste de aceitação, onde cada um tem um objetivo específico.


Figura 1.1: Níveis de teste de acordo com a fase de desenvolvimento. Fonte: [23].

O objetivo do teste de unidade é verificar o funcionamento do software e procurar defeitos na menor parte testável (em métodos, classes, módulos ou componentes) separadamente, ou seja, isolado do restante do sistema. Com os módulos, classes ou componentes testados separadamente, ao serem acoplados deve-se verificar se eles interagem corretamente, correspondendo ao nível do teste de integração. Neste nível, os testes não são focados na funcionalidade dos componentes, mas se sua comunicação está conforme especificado. Já os testes de sistema acontecem depois dos testes de integração e têm como foco o funcionamento do sistema como um todo para validar a execução das funções requeridas. O ultimo nível corresponde aos testes de aceitação, cuja finalidade é validar se o software está em condições de ser implantado em produção, com base em critérios de aceitação do cliente.

O código de um teste de unidade consiste em sequências de invocações a métodos que exploram/exercitam algum aspecto de comportamento da CUT (*Class Under Test*), que tem como objetivo explorar as menores partes testáveis e procurar falhas de funcionamento dentro de uma pequena parte do sistema funcionando independentemente do todo. Porém, quanto mais complexos são os testes unitários, maior a dificuldade em se isolar a unidade, devido ao grau de dependência com outras unidades envolvidas, resultando em um teste que também testa essas outras unidades ou dependências, chamadas de colaboradores, deixando de focar apenas na unidade sob teste. Para resolver este problema, podemos utilizar objetos *mock*, que ajudam a promover o isolamento do objeto sendo testado em relação aos demais.

O conceito de objetos *mock* foi originalmente apresentado por Tim Mackinnon [42] para dar suporte aos testadores/desenvolvedores na escrita de testes com objetos isolados, apoiado por Freeman [32] , muito utilizado em Extreme Programming [18]. Um objeto *mock* consiste em um pedaço de código executável que simula o comportamento de outros objetos em um cenário de teste específico. Em um teste de unidade, *mocks* são usados para simular objetos colaboradores, onde a CUT não sabe se ela está interagindo com colaboradores reais ou objetos *mock*. Desta forma, objetos *mock* podem ser de grande utilidade na escrita de testes de unidade, uma vez que isolam a CUT. Ainda segundo Freeman [42], objetos *mock* tornam o teste mais eficaz e eficiente. Primeiro, porque testes baseados em *mocks* provêem informação mais precisa sobre falhas e defeitos, já que isolam a unidade, fazendo com que o teste foque apenas nela. Segundo, porque testes baseados em *mocks* utilizam simulações simplificadas dos colaboradores reais, assim, o teste tende a executar mais rapidamente, especialmente quando os colaboradores possuem acesso a banco de dados, interagem com sistemas externos ou se utilizam de recursos da Internet, por exemplo.

## 1.1 Problema

Apesar das vantagens decorrentes do uso de objetos *mock*, escrever testes com objetos *mock* pode ser uma tarefa custosa e tediosa, mesmo com auxílio de *frameworks*, como EasyMock [34] por exemplo, devido as atividades necessárias ao seu funcionamento. Dado que um teste baseado em *mocks* consiste em escrever um script de expectativas. Para tanto, se faz necessário que o testador/programador tenha um profundo conhecimento do sistema como um todo para identificar as interações que envolvem a CUT e os colaboradores em um dado cenário de teste, o que inclui saber não só a sequência de chamadas a serem feitas, mas também os dados que são recebidos como parâmetro e retornados em cada chamada, o que é uma tarefa bastante suscetível a erros. Além disso, código *mock* se torna obsoleto rapidamente, pois eles precisam ser reconfigurados a cada mudança no cenário de teste.

Outro aspecto bastante associado ao uso objetos *mock* é a prática de desenvolvimento guiado por testes TDD (*Test Driven Development*), resultando em vários benefícios para o sistema, cujo principal seria a melhoria do design do código [31]. Porém, segundo Kerievsky [32] o uso de *mocks* com TDD na vida prática de uma empresa que adota esta abordagem

resulta em um código que tende a ficar ilegível, ocasionando perda de tempo com a manutenção dos *mocks* na fase de refatoramento do TDD.

Pelas razões supracitadas, utilizar objetos *mock* é uma tarefa árdua, e como conseqüência, na prática, testadores/programadores têm que fazer uma análise custo-benefício, resultando na utilização de objetos *mock* apenas em parte dos testes, ou seja, onde há um maior risco associado. Como exemplo, temos o projeto OurBackup Home [46] desenvolvido no LSD (Laboratório de Sistemas Distribuídos) [7], onde 53,4% dos testes de unidade automáticos utilizavam objetos *mock*, desenvolvidos manualmente com o suporte de um *framework*, correspondendo a uma grande parte dos testes.

## 1.2 Solução Proposta

Dada a elevada necessidade de se utilizar objetos *mock* nos testes de unidade e seus benefícios, e considerando a dificuldade envolvida no desenvolvimento e manutenção deste tipo de código, uma solução seria automatizar a escrita desses objetos *mock*, gerando-os automaticamente para um determinado teste. É neste contexto que se situa o trabalho apresentado nesta dissertação. Mais precisamente, investigamos a hipótese de que

> *é possível reduzir o esforço envolvido no desenvolvimento e manutenção de testes com objetos mock, através de uma técnica de automação da escrita de código de teste com objetos mock com qualidade.*

Para isso, propomos, implementamos e avaliamos uma técnica que gera código de teste com objetos *mock*. Esta técnica tem por base a identificação das interações entre a CUT e seus colaboradores em um dado cenário de teste, através da análise estática e dinâmica de código. A análise dinâmica é realizada através da instrumentação do código de teste fornecido, utilizando a programação orientada a aspectos [39]. Após essas análises, a técnica provê a geração de código de teste com objetos *mock* para os colaboradores e interações identificadas. O código resultante consiste em uma nova versão do código do cenário de teste inicial, onde os colaboradores da CUT são substituídos por objetos *mock*, bem como as interações da CUT com os mesmos são transformadas em código *mock*. Considerando o baixo custo de se gerar automaticamente o código para os objetos *mock*, os programadores

não precisarão reutilizar este código, o qual poderá ser re-gerado a cada alteração no teste inicial.

É importante ressaltar que o uso do AutoMock é mais indicado para a abordagem *Test-Last*, onde os testes são produzidos depois do desenvolvimento. Porém, também pode ser utilizado com a abordagem *Test-First*, onde os testes são produzidos antes do desenvolvimento, no caso do TDD. Para tanto, as interfaces da CUT e dos colaboradores precisam existir para que os testes com objetos *mock* sejam gerados, que neste caso resulta em um teste incompleto, dado que as interações reais ainda não existem. Desta forma, quando o código das entidades envolvidas no teste tiver sido desenvolvido, podemos re-gerar o teste, que desta vez será completo. E independentemente da abordagem de teste utilizada, o modelo de faltas do AutoMock cobre as faltas da CUT, tanto a nível funcional quanto estrutural.

## 1.3 Resultados

Com o propósito de verificar a viabilidade da técnica proposta, desenvolvemos um protótipo de ferramenta que implementa esta técnica, chamado AutoMock, que é uma extensão do protótipo apresentado por Souto em [60; 61], desenvolvido no contexto do projeto AutoTest do LSD [7]. Já para avaliar a técnica proposta, seguimos duas abordagens complementares: uma análise quantitativa e uma análise qualitativa, utilizando um estudo experimental e uma pesquisa de opinião respectivamente. O objetivo foi verificar a aplicabilidade e a relevância da técnica proposta quando comparada a atual forma de desenvolvimento de testes com objetos *mock*.

A análise quantitava é feita sob duas perspectivas: a primeira está relacionada ao uso de objetos *mock* de forma manual e a segunda está relacionada à utilização de objetos *mock* de forma totalmente automática, através da técnica proposta e implementada. Resultando em um estudo experimental comparativo entre a forma manual e a automática de se desenvolver objetos *mock* para um dado teste, através da observação do esforço empregado em tais atividades e o nível de cobertura de interações do código resultante destas atividades, como parâmetro de qualidade.

Como resultado, temos evidências de que o esforço empregado para se desenvolver código *mock* de forma automática é menor que o esforço para se desenvolver o mesmo tipo de

código de forma manual. Da mesma forma, a análise realizada para a cobertura de interações indica que a abordagem automática possui uma maior cobertura de interações que a manual, o que se deve ao fato do AutoMock reproduzir fielmente todas as interações do teste inicial, enquando a abordagem manual depende da perceptividade do programador/testador em relação as interações que precisam ser "*mockadas*".

Durante a análise qualitativa, os resultados indicaram possíveis causas para os resultados obtidos na análise quantitativa. Assim sendo, temos que as possíveis causas para o esforço ser maior na abordagem manual, além de tratar-se de uma tarefa lenta e repetitiva são: a dificuldade em configurar objetos *mock*, detectar interações entre a CUT e os colaboradores, entender a sintaxe usada pelo *framework*, identificar o nível de isolamento do objeto no teste, além de que se trata de uma tarefa lenta e repetitiva. Dentre estas dificuldades, podemos apontar a dificuldade de se detectar as interações entre a CUT e os colaboradores como a principal causa para a cobertura de interações na abordagem manual ser menor que a da automática.

É importante clarificar que os resultados obtidos através das análises quantitativa e qualitativa possuem validade apenas no presente estudo, e não são generalizáveis para todos os tipos de sistemas que existem. Para aumentar o conhecimento sobre o esforço e a cobertura de interações em diferentes contextos, definindo a validade da experimentação, sugerem-se replicações do experimento em sistemas distintos, desenvolvendo testes com código *mock* para diferentes tipos de sistema.

Com a realização do experimento, podemos perceber que o AutoMock possui algumas limitações em relação à gerência de configuração, ao tipo de sistema suportado e à qualidade do teste gerado, que depende diretamente da qualidade do teste inicial, ou seja, se o teste recebido como entrada para a técnica estiver mal escrito, conseqüentemente, o código final gerado também estará mal escrito, visto que a geração de código se baseia no teste inicial.

Como resultados acadêmicos, obtivemos duas publicações: uma no LADC 2009 [61] e outra no LA-WASP 2009 [60].

## 1.4 Estrutura da Dissertação

O restante da dissertação está estruturada da seguinte forma:

- Capítulo 2: fornece embasamento teórico para compreensão da técnica apresentada neste trabalho, bem como para sua avaliação;
- Capítulo 3: descreve a técnica para geração automática de testes com objetos *mock* e apresentamos a ferramenta AutoMock, que implementa a técnica proposta;
- Capítulo 4: descreve o estudo que realizamos para avaliar a técnica e a análise dos resultados;
- Capítulo 5: apresenta os trabalhos relacionados;
- Capítulo 6: apresenta as contribuíções e os trabalhos futuros.

# Capítulo 2

# Fundamentação Teórica

Neste capítulo, abordamos conceitos relevantes que foram utilizados como base para realização deste trabalho. Inicialmente, apresentamos conceitos relacionados a testes de software, com ênfase em testes de unidade e testes de integração. Em seguida, discutimos sobre as diferenças e entre stubs e objetos *mock* e ponderamos sobre as vantagens e desvantagens de usar cada abordagem. Posteriormente, mostramos testes com objetos *mock* e testes baseados em interação. Estudamos também as técnicas de automação e geração de testes, bem como técnicas de análise estática e dinâmica de código, que nos serviram de base para propor a técnica de geração automática de testes com objetos *mock*. Por fim, apresentamos uma revisão sobre Engenharia de Software Experimental, que embasará os conceitos utilizados durante o estudo experimental, descrito no Capítulo 4.

## 2.1 Teste de Software

Testar um software significa verificar através de uma execução controlada se o seu comportamento corresponde ao especificado. O objetivo principal desta tarefa é revelar o número máximo de falhas dispondo do mínimo de esforço, ou seja, mostrar aos que desenvolvem se os resultados estão ou não de acordo com os padrões estabelecidos. O IEEE tem realizado vários esforços de padronização, entre eles para padronizar a terminologia utilizada no contexto de Engenharia de Software. O padrão IEEE número 610.12-1990 [4] diferencia os termos básicos utilizados nesta área. Todo o vocabulário, da área de testes, utilizado neste trabalho segue a terminologia IEEE, portanto tomamos como importante apresentar alguns

conceitos aqui:

- **Defeito** (*fault*) - passo, processo ou definição de dados incorreta, como por exemplo, uma instrução ou comando incorreto;
- **Engano** (*mistake*) - ação humana que produz um resultado incorreto, como por exemplo, uma ação incorreta tomada pelo programador;
- **Erro** (*error*) - diferença entre o valor obtido e o valor esperado, ou seja, qualquer estado intermediário incorreto ou resultado inesperado na execução do programa constitui um erro;
- **Falha** (*failure*) - produção de uma saída incorreta com relação à especificação. Neste texto, os termos engano, defeito e erro serão referenciados como erro (causa) e o termo falha (conseqüência) a um comportamento incorreto do programa.

### 2.1.1 Teste de Unidade

Teste de unidade, também conhecido como teste unitário. No caso do paradigma orientado a objetos, esta unidade pode ser uma classe ou até mesmo um método. O código de um teste de unidade consiste de seqüências de invocações a métodos que exploram/exercitam algum aspecto de comportamento da classe sob teste (*Class Under Test* - CUT), que tem como objetivo explorar a menor unidade do projeto, procurando falhas ocasionadas por defeitos de lógica e de implementação em cada unidade, separadamente. A Figura 2.1, ilustra a composição dos testes de unidade.

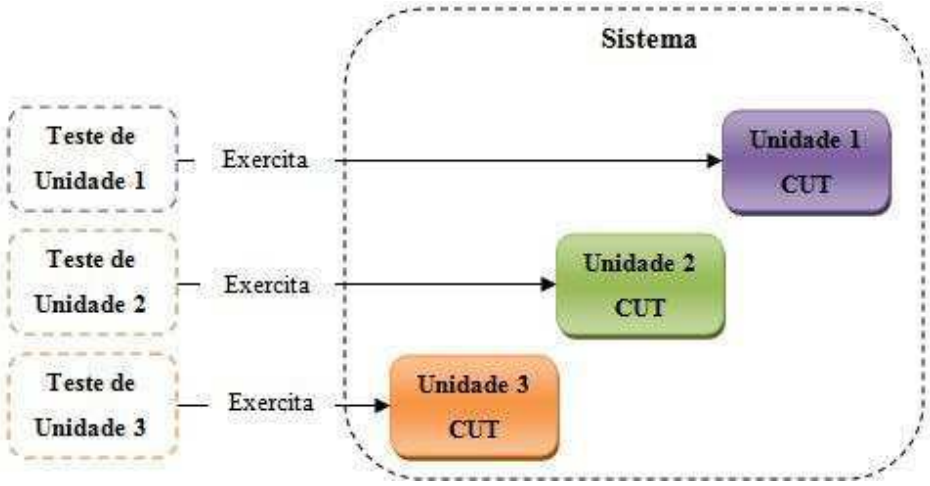



Figura 2.1: Visão geral de um teste de unidade. Adaptada de [44]

### 2.1.2 Teste de Integração

Conforme o padrão IEEE número 610.12-1990 [4], teste de integração é a fase do teste de software onde os módulos (ou classes) são combinados e testados de forma integrada e tem por objetivo testar a interface entre os módulos, como ilustrado na Figura 2.2. Esse teste sucede o teste de unidade, em que os módulos/classes são testados individualmente, e antecede o teste de sistema, em que o sistema completo (integrado) é testado num ambiente de produção.

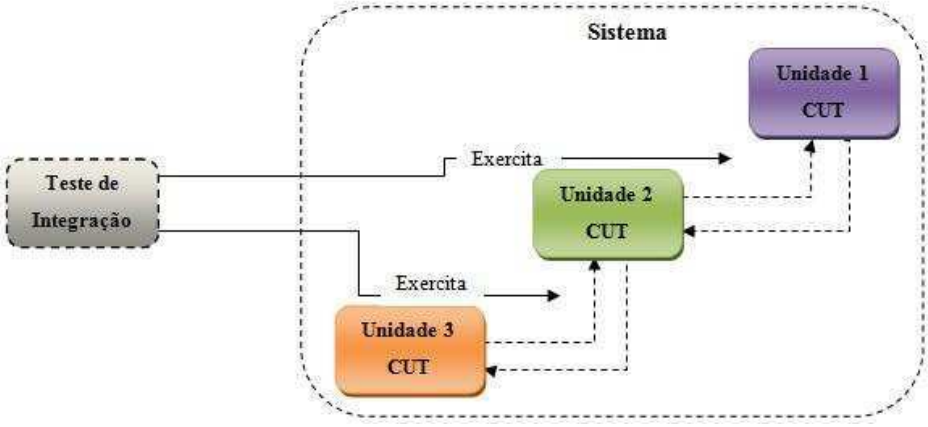



Figura 2.2: Visão geral de um teste de integração.

### 2.1.3 Objetos *Mock*

Segundo Mackinnon [42], o6bjetos *mock* são objetos que simulam o comportamento de um objeto real, um objeto *mock* possui expectativas de comportamento, as quais formam a especificação das chamadas que eles estão esperando receber para verificar este comportamento. Os objetos *mock* funcionam em conjunto com os testes de unidade da seguinte maneira: primeiro definimos um objeto *mock* que geralmente implementa a mesma interface do objeto que a unidade (o objeto sob teste) depende. Durante a codificação do teste, o estado do objeto *mock* é configurado, como também suas expectativas com os valores que a unidade espera, depois a unidade é exercitada, e segue a fase de verificação, onde o objeto *mock* compara os resultados obtidos com os esperados. O detalhe é que a unidade usa o objeto *mock* ao invés do objeto real, através de sua a interface ou classe, por reflexão. No entanto, não podemos utilizar objetos *mock* em todos os nossos testes, pois existem situações específicas onde se indica o uso dos mesmos, por exemplo, quando:

- O objeto real tem comportamento não determinístico;
- O objeto real é difícil de configurar;
- O objeto tem um comportamento difícil de acionar;
- O objeto real é lento;
- O objeto real é uma interface;
- O objeto real ainda não existe;
- O objeto real usa *call back*.

Dentre as vantagens de se utilizar objetos *mock*, destacamos:

- Não precisamos utilizar objetos colaboradores reais;
- Podemos definir exatamente o comportamento que esperamos do colaborador, no *mock*;
- A execução dos testes é rápida. Não tem dependências com qualquer base de dados ou com conexões de rede;

- O objeto *mock* utilizado pode ser reutilizado e alterado;
- Com objetos *mock* somente a unidade é testada, com isso podemos localizar com mais facilidade o defeito.

Porém, existem algumas desvantagens do uso de objetos *mock*:

- Acesso a colaboradores pode causar problemas, já que para alocar objetos, precisamos abrir acesso a métodos para permitir que os mesmos sejam testados;
- O uso de interfaces, pois às vezes o código não está preparado para ser testado, então temos que refatorar código e criar interfaces para que possamos construir os objetos *mock*;
- A implementação de *mocks* pode ser trabalhosa para inicializar sua configuração, dependendo do objeto que o *mock* está simulando;
- Asserções precisam ser mapeadas para possíveis objetos, chamadas a métodos ou estados finais do colaborador real. É necessário um grande conjunto de asserções robustas para se testar efetivamente, o que inclui o número de chamadas a métodos e o retorno do valor esperado.

### 2.1.4 Testes com Objetos *Mock*

Nos testes de unidade são testadas as menores unidades de software desenvolvidas. Sendo assim, o objetivo é encontrar falhas de funcionamento dentro de uma pequena parte do sistema funcionando independentemente do todo. Porém quanto mais complexos são os testes unitários, são mais difíceis de serem feitos isoladamente, devido ao grau de dependência com outras unidades envolvidas, resultando em um teste que também testa essas outras unidades ou dependências. Neste sentido, objetos *mock* ajudam a promover o isolamento do objeto sendo testado em relação aos demais. A Figura 2.3, fonte [22], compara um teste de unidade clássico, sem o uso de objetos *mock*, que usa diretamente suas dependências ou colaboradores com um teste de unidade que usa objetos *mock*, isolando a unidade sob teste.

Através do uso de objetos *mock*, a estrutura dos testes é simplificada, ajudando também a evitar a poluição do código fonte por causa da infra-estrutura de testes. Mackinnon [42]

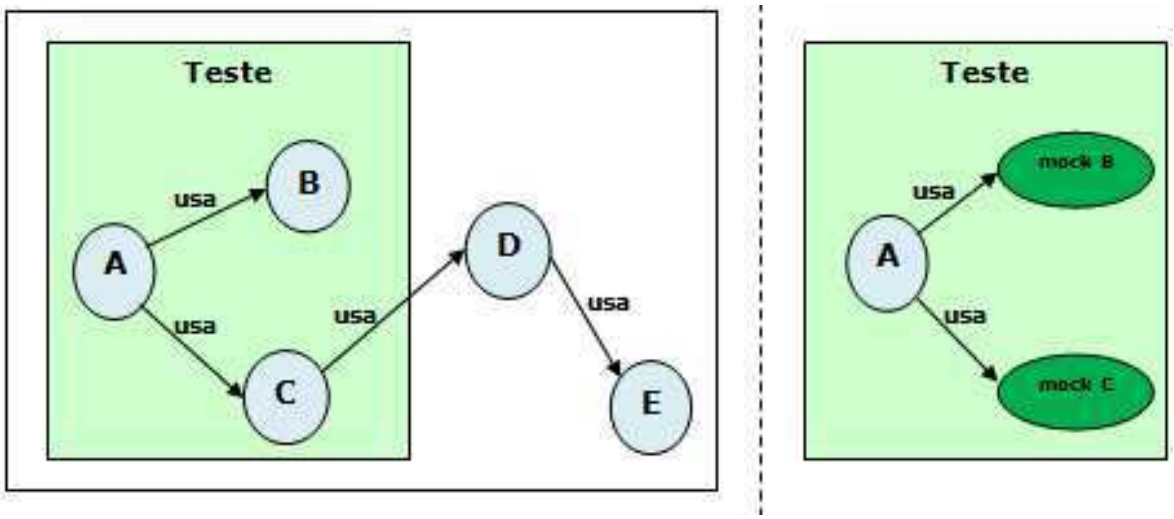



Figura 2.3: Teste de unidade com dependências x Teste de unidade sem dependências, através do uso de objetos *mock*

propõe a técnica de utilizar objetos *mock* para substituir objetos reais durante um teste de unidade. Estes objetos *mock* seriam passados para o interior do código em teste para que os testes sejam feitos internamente. A experiência do autor com o uso de *mocks* na construção de testes de unidade é que esta prática leva a testes mais robustos, melhorando também a estrutura do código desenvolvido e dos testes. Testes de unidade escritos com objetos *mock* têm um formato regular, que dão ao time de desenvolvimento um vocabulário comum. Um teste deve testar uma unidade por vez, de forma simples e clara. Isto pode ser difícil quando o cenário do teste é muito complexo de se construir e/ou finalizar, principalmente se o código fonte dificulta a construção deste cenário (algo não acessível a partir de uma classe de teste, por exemplo).

#### 2.1.4.1 *Stubs* x Objetos *Mock*

Testar a unidade isoladamente é um dos princípios mais importantes no teste de unidade, porém a CUT geralmente depende de outras classes, como a Figura 2.4 ilustra, fonte [22], onde A é nossa CUT:

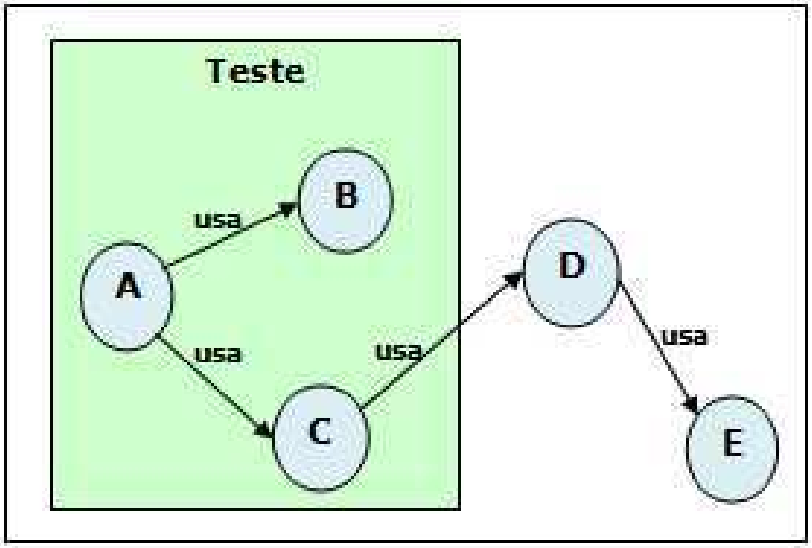



Figura 2.4: Representação de um teste que depende de outras classes.

Para testarmos isoladamente A, a CUT da Figura 2.4, precisamos quebrar suas dependências. Podemos fazer isto utilizando *stubs* ou objetos *mock*. Porém, um erro comum é interpretar *mocks* como sento *stubs*. Apesar de poderem ser usados no teste com a mesma finalidade, existem algumas diferenças significantes:

- Um *stub* consiste em uma classe falsa contendo métodos vazios que retornam algum comportamento padrão, baseado no estado do objeto, para a CUT. Porém stubs apresentam algumas dificuldades na prática, pois além deles serem programados manualmente, existe a necessidade de que certas operações sejam feitas no código de acordo com os cenários de teste, adicionando código que é desnecessário para o desenvolvimento do sistema, tornando desta forma o código difícil de manter, principalmente quando os estados do objeto mudam. Segundo Binder [19], manter *stubs* é um processo árduo e de alto custo, pois os mesmos não são reusados com facilidade devido ao alto grau de acoplamento com a CUT.

- Objetos *Mock*, técnica proposta por Mackinnon [42]. Estes promovem o isolamento da unidade sendo testada em relação às demais, através da substituição dos colaboradores reais por objetos que simulam seus comportamentos durante um teste de unidade. Tal simulação é feita com base na interface do objeto colaborador real, resultando em um objeto que "imita"ėste colaborador e, ao contrário de um *stub*, um objeto *mock*

precisa apenas ser configurado no contexto do teste de unidade, ou seja, o programador não precisará fazer alterações manualmente como se fazia com *stubs*, além disso os objetos *mock*, quando desenvolvidos com o suporte de um *framework*, são totalmente reusáveis, no sentido de que o *framework* permite limpar as configuraçõesde um objeto *mock*, atrevés do comando `reset(mock)` no EasyMock [34] por exemplo. A partir de então, esse objeto pode ser reconfigurado e reusado.

#### 2.1.4.2 *Frameworks*

Existem várias APIs no mercado para escrita de *mocks*, porém aqui iremos abordar as duas mais utilizadas: JMock [33] e EasyMock [34].

- **JMock**

  É um *framework* de código aberto, que provê uma API bastante simples e expressiva para construção de *mocks* a partir de interfaces. A API do JMock trabalha com a geração dinâmica de *mocks*, permitindo ao usuário definir e depois verificar todas as expectativas nos *mocks*. O ponto de entrada para o JMock é a classe "MockObject-TestCase", que estende "TestCase"do JUnit [6], dando o suporte para o uso de objetos *mock*. A classe "MockObjectTestCase"permite que as "expectations"de um *mock* seja definida de maneira simples. Objetos *mocks* são criados através de *mock* (*ClassInterface*), que retorna um objeto *mock* que implementa a interface passada. O JMock, segundo Mackinnon [33], é uma ferramenta facilmente extensível.

- **EasyMock**

  É um *framework* de código aberto, que provê uma forma simples de usar objetos *mock* a partir de interfaces fornecidas. EasyMock usa a técnica *record/replay* para configurar as expectativas. Para cada objeto que será um *mock*, deve-se criar um controlador e o objeto *mock*, o qual satisfaz a interface do objeto secundário e o controlador provê as características adicionais. Para indicar uma expectativa, realiza-se uma chamada ao método com os argumentos esperados no *mock*, segue a chamada do controlador, se você quiser o valor de retorno. Uma vez que as expectativas foram configuradas, você invoca o comando o *replay* no controlador, então o *mock* finaliza o *record*, e por fim ocorre a verificação das expectativas.

### 2.1.5 Teste Baseado em Interações

Existe o conceito de interação, teste de interação e teste baseado em interação. Sendo assim, se faz necessário diferenciar estes conceitos:

- **Interação**: é a forma como as classes ou componentes se comunicam em um sistema, que geralmente ocorre através da troca de mensagens, ou seja, uma classe ou componente envia e/ou recebe mensagem de outro, por sua interface, a qual provê métodos para esta finalidade.

- **Teste de Interação**: é um tipo de teste que visa verificar a correta colaboração dos componentes de um sistema, ou seja, como ele interage com os outros [43; 49];

- **Teste Baseado em Interações (*Interaction-Based Testing - IBT*)**: é um teste construído com base nas interações da CUT com seus colaboradores, estas interações especificam o comportamento da CUT. Desta forma, se escolhermos exercitar a CUT e suas interações, isso se configura um teste de unidade. Usamos objetos *mock* para testar se nosso objeto sob teste (CUT) interage com os outros objetos de forma correta, simulando o comportamento dos objetos colaboradores, e mantendo o foco apenas na CUT e suas interações com o resto do sistema. Desta forma, também podemos chamar o teste de unidade com objetos *mock* de teste baseado em interação, visto que ao usar objetos *mock* isolamos a unidade (CUT) através da substituição dos seus colaboradores por objetos *mock* [31; 52], ilustrado na Figura 2.5, focando apenas na CUT e suas interações com o resto do sistema.

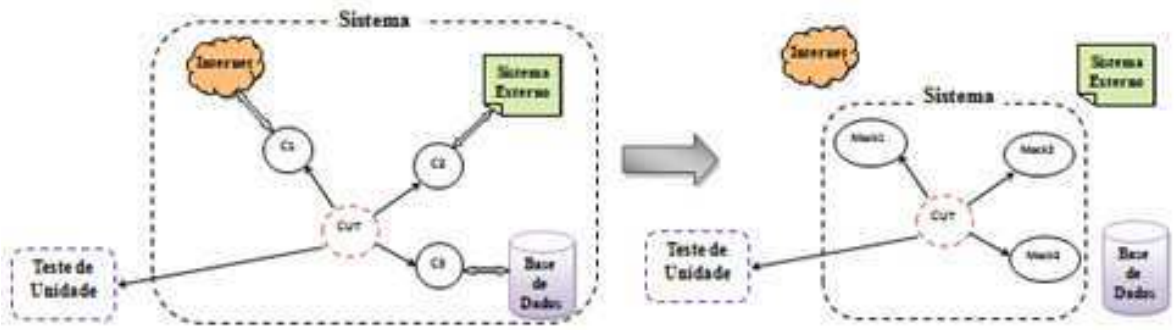



Figura 2.5: Um teste utilizando colaboradores reais e um teste utilizando objetos *mock* no lugar de colaboradores reais.

## 2.2 Automação de Testes

Segundo Kaner [38], o propósito da automação de testes pode ser resumidamente descrito como a aplicação de estratégias e ferramentas tendo em vista a redução do envolvimento humano em atividades manuais repetitivas. Tais atividades são características do teste manual, onde o testador executa o software manualmente com base nas especificações dos testes. Os testes realizados dessa forma, em geral, são cansativos e de difícil repetição. Já os testes automatizados, são aqueles feitos com auxilio de alguma ferramenta apropriada, ou ainda gerados automaticamente. Além disso, a execução de testes automatizados é mais rápida do que os testes manuais e menos suscetível a erros.

Existem várias abordagens para a automação de testes. Existem ainda técnicas de automação específicas para cada tipo de teste, por exemplo, para testes funcionais, testes unitários, teste de desempenho, dentre outros. No contexto da automação de teste, podemos ter também: a geração automática de testes, de casos de teste, e de dados de teste. No entanto, neste trabalho iremos focar apenas nas técnicas utilizadas para a geração automática de testes unitários. Dentre estas, podemos citar a Execução Randômica, a Execução Simbólica e a Captura e Reprodução.

- **Execução Randômica (*Random Execution*)**: Esta técnica analisa a CUT usando reflexão para recuperar os métodos e seus parâmetros. Suponha que uma classe C tenha um método com uma assinatura m(P1, P2,..., PN) que retorna R. Para testar este método, precisa-se saber como construir valores dos tipos C, P1, P2,..., PN, além de inferir um objeto do tipo R. A partir destas regras de inferência, sequencias de chamadas a método randômicas podem ser geradas automaticamente. Esta técnica é utilizada nas seguintes ferramentas [16; 25; 47; 66];

- **Execução Simbólica (*Simbolic Execution*)**: Esta técnica gera uma sequencia de chamadas a métodos com argumentos simbólicos, cada argumento deste representa um conjunto de possíveis valores concretos para os mesmos. Depois, explora todos os caminhos para cada método, analisando condições e restrições, que posteriormente são resolvidas, produzindo valores reais para os argumentos. Existem várias ferramentas que se utilizam desta técnica, dentre elas temos: [20; 63; 66];

- **Captura e Reprodução (*Capture and Replay*)**: Captura e reproduz: sequencias de métodos, argumentos, valores de retorno e exceções lançadas. Estas sequencias são observadas na execução ou teste de software. Resultando em dados que podem ser usados para gerar casos de teste e/ou objetos *mock*. Esta técnica é utilizada nas seguintes ferramentas: [27; 48; 55; 67].

## 2.3 Análise Estática de Código

Análise Estática de Código é qualquer tipo de teste ou exame que pode ser feito no código sem executá-lo [11]. Ainda segundo Sommerville [59] e o CTAL Syllabus [11], existem várias técnicas de análise estática: análise de fluxo de controle, análise de fluxo de dados, compilação e padronização de código, geração de métricas sobre o código. Pode-se ainda fazer a análise estática da arquitetura do sistema.

- **Análise de Fluxo de Controle**: fornece informações sobre os pontos de decisão lógica do sistema e sobre a complexidade de sua estrutura [11];

- **Análise de Fluxo de Dados**: testa os possíveis caminhos ou fluxos de execução, partindo da definição de uma variável até onde ela possivelmente será utilizada, formando um par que é chamado de definição-utilização (*definition-use*). Neste método, conjuntos de teste são gerados para se conseguir 100% de cobertura, para cada par [11];

- **Compilação e Padronização de Código**: verifica se o código obedece a normas de codificação, que abrange tanto aspectos arquiteturais quanto estruturas de programação, permitindo que o software seja mais manutenível e testável [11];

- **Geração de Métricas sobre o Código**: durante a análise estática podem ser geradas métricas de código que irão contribuir para um maior nível de manutenibilidade e confiança no código, essas métricas podem ser: complexidade ciclomática, tamanho, número de chamadas a funções, dentre outras [11].

- **Análise Estática de Arquitetura**: são análises feitas no código para verificar sua estrutura, como por exemplo: como os módulos e as classes se comunicam, tipos de retorno, parâmetros, chamadas indevidas, etc. [11].

Existem ferramentas que fazem análise estática automaticamente e são chamadas de verificador ou analisador estático de código. Os verificadores estáticos são ferramentas de software que varrem o código fonte ou mesmo o código objeto (no caso da linguagem Java, o *bytecode*). Cada versão tem suas próprias vantagens. Quando se trabalha diretamente sobre o código fonte do programa, suas verificações refletem exatamente o código escrito pelo programador, já que compiladores otimizam o código e o *bytecode* resultante pode não refletir o código fonte. Por outro lado, trabalhar em *bytecode* é consideravelmente mais rápido, o que é importante em projetos contendo dezenas de milhares de linhas de código [41].

Os verificadores estáticos de código também podem verificar regras de estilos de programação, erros ou ambos. Um verificador de regras de estilo examina o código para determinar se ele contém violações de regras de estilo de código. Por outro lado, um verificador de erro utiliza a análise estática para encontrar código que viola uma propriedade de correção específica e que pode causar um comportamento anormal do programa em tempo de execução, por exemplo [17].

## 2.4 Análise Dinâmica de Código

Existem defeitos que não são facilmente encontrados ou reproduzidos e que podem ter consequências significantes no esforço de teste e na produtividade do software. Tais defeitos podem ser causados por vazamentos de memória, uso incorreto de ponteiros e outros tipos de violação [38]. São típicos defeitos que não conseguimos encontrar na análise estática, mas apenas quando o sistema é executado, e às vezes somente sob condições específicas.

Neste contexto, surge a análise dinâmica com o propósito de capturar o comportamento do sistema para detectar pontos onde ele pode falhar, ou ainda para outros propósitos de programação, como verificar, em tempo de execução, a comunicação entre módulos e classes, complementando a análise estática. Este tipo de análise pode ser feita através de testes ou de mecanismos de instrumentação de código, dependerá do propósito. No nosso caso, utilizamos a instrumentação de código.

O conceito de instrumentação de código está contido num outro mais abrangente: o de Reflexão [15; 29], este é definido como sendo a capacidade de um programa de "olhar para si próprio"(i.e, saber como é constituído e estruturado) e ser capaz de modificar tanto a sua

estrutura como o seu comportamento em tempo de execução. A instrumentação de código é um mecanismo que permite aos programas reescrever o código de outros programas ou o seu próprio código, após a compilação e imediatamente antes ou durante a sua execução [21; 28].

Recentemente, a instrumentação de código tornou-se mais atrativa com o aparecimento da Programação Orientada aos Aspectos (POA) [39]. Existem muitas ferramentas que fazem instrumentação de código, porém a maioria cria dependência com o código nativo da linguagem. Já a instrumentação por POA permite a interceptação e instrumentação de elementos de classes, necessitando apenas o conhecimento da API das aplicações, sem causar dependência com o código nativo da linguagem. Assim, iremos introduzir alguns conceitos relacionados à POA na próxima seção.

### 2.4.1 Programação Orientada a Aspectos (POA)

O conceito foi criado por Gregor Kiczales e a sua equipe na Xerox PARC, a divisão de pesquisa da Xerox. Eles desenvolveram o AspectJ, a primeira e mais popular linguagem orientada a aspectos. Este tipo de programação (POA) [39] foi proposta com o objetivo de facilitar a modularização dos interesses transversais (crosscutting concerns) de um sistema. Interesses transversais são funcionalidades que precisam atingir várias partes do sistema de forma independente da implementação das suas funcionalidades. A POA complementa o uso da Programação Orientada a Objetos (POO) de maneira a contemplar os módulos de um sistema que atravessam toda a sua implementação.

Em POO cada interesse de um sistema é modularizado em objetos, que contém as informações referentes a este e concentra toda esta informação em um único local. Na POA há a inserção de um novo mecanismo para abstração e composição da informação de interesse, facilitando a modularização de interesses transversais. Este mecanismo é chamado de aspecto.

#### 2.4.1.1 AspectJ

Um aspecto é composto por um conjunto de definições que especificam as regras de combinação para interesses estáticos e dinâmicos. Um aspecto é a unidade central de AspectJ,

assim como uma classe é a unidade central em Java [58].

Em AspectJ, o aspecto é composto por blocos de construção que expressam os interesses transversais da implementação de um sistema, especificando as regras de combinação. Estes blocos são: ponto de junção (*joinpoint*), conjunto de junção (*pointcut*), adendo (*advice*) e declaração intertipos (*inter-type declaration*). Nas próximas seções cada um destes blocos é descrito para entendimento da construção dentro do aspecto.

#### 2.4.1.1.1 Ponto de Junção (*Join Point*)

Um ponto de junção é um ponto identificável na execução de um programa, pode ser a chamada de um método, a inicialização de um objeto ou a modificação de um membro deste. Em AspectJ, tudo remete a pontos de junção, pois estes são os lugares que as ações transversais acontecem. A seguir temos a lista de possíveis pontos de junção em AspectJ [35]:

- **Chamada de método**: É definido quando qualquer chamada de método é realizada por um objeto ou por um método estático, como o método main;
- **Chamada de construtor**: Quando um construtor é chamado durante a criação de um novo objeto;
- **Execução de método**: Este ponto é definido quando uma chamada de método é realizada por um objeto e o controle é transferido para o método invocado;
- **Execução de construtor**: Semelhante ao anterior, quando da entrega do controle para a execução do construtor. O ponto de junção é ativado antes do início da execução do construtor;
- **Acesso a campo**: É definido quando um atributo associado a um objeto é lido;
- **Modificação de campo**: Acontece quando um atributo associado a um objeto é modificado;
- **Lançamento de exceção**: O ponto de junção é definido quando o lançamento de uma exceção é executada;

- **Inicialização de classe**: É definido quando qualquer inicializador estático é executado para uma determinada classe. Se não existir nenhum campo estático então não terá este tipo de ponto de junção;

- **Inicialização de objeto**: É definido quando um inicializador dinâmico é executado para uma determinada classe. Este ponto de junção é definido depois da construção do objeto e antes de retornar o controle para o criador do objeto.

#### 2.4.1.1.2 Conjunto de Junção (*Pointcut*)

O conjunto de junção consiste na seleção dos pontos de junção e na coleta do contexto de execução destes pontos. Por exemplo, o conjunto de junção seleciona o ponto de junção, que é a chamada para um método, podendo capturar o contexto deste método, como o valor de campos do objeto que está executando o método invocado ou os argumentos do mesmo [40].

Os conjuntos de junção podem ser agrupados por pontos de junção através de operadores lógicos, tais como conjunção (&&), disjunção (||) e negação (!). O conjunto de junção pode ainda ser *anônimo* ou *nomeado*, com especificação de nível de acesso (publico, privado ou *default*). Um conjunto de junção nomeado pode ser definido no seguinte formato:

```
[especificador de acesso] pointcut <nome> ([argumentos]):
<definição do conjunto de junção>;
```

A definição para um conjunto de junção anônimo é de forma direta, já no ponto de atuação do mesmo. A definição de um conjunto de junção anônimo implica no uso direto de um adendo, que é a especificação de qual o momento certo de atuar sobre um determinado ponto(s) de junção(ões). O adendo será descrito em maiores detalhes na próxima seção. O conjunto de junção tem a seguinte forma:

```
<tipo de adendo>: <definição do conjunto de junção>
```

Para definir um conjunto de junção utiliza-se construtores de AspectJ nomeados de designadores. Um designador identifica o conjunto de junção por nome ou por uma expressão. Os principais designadores estão listados na Tabela 2.1 [35]:

Durante a especificação de pontos de junção, muitas vezes se faz necessária a especificação de um subconjunto total de um determinado tipo de elementos. AspectJ provê uma

| Designador | Características |
|---|---|
| `execution(Signature)` | Execução do método/construtor identificado pela assinaturas |
| `call(Signature)` | Invocação do método/construtor identificado pela assinatura |
| `get(FieldSignature)` | Acesso ao atributo identificado pela assinatura |
| `set(FieldSignature)` | Atribuição ao atributo identificado pela assinatura |
| `this(Type pattern)` | Objeto em execução é a instância do tipo |
| `target(Type pattern)` | Objeto de destino é a instância do tipo |
| `args(Type pattern)` | Os argumentos são instâncias do tipo |
| `within(Type pattern)` | Limita o escopo do conjunto de junção para determinados tipos |

Tabela 2.1: Listagem dos designadores em AspectJ

forma de defini-los sem ter que especificar cada um separadamente. A definição de *wildcard*, um caracter usado para representar um conjunto de combinações, é utilizada para generalizar alguns pontos de junção, capturando pontos de junção que compartilham das mesmas características.

Em AspectJ temos os seguintes *wildcards* [40]:

- qualquer seqüência de caracteres não contendo pontos;
- .. qualquer seqüência de caracteres, inclusive contendo pontos;
- + qualquer subclasse de uma classe.

#### 2.4.1.1.3 Adendo (*Advice*)

O adendo é semelhante a um método que provê uma forma de expressar a ação transversal nos pontos de junção capturados pelo conjunto de junção. Os três tipos de adendos são:

- Antes (*Before*) executa antes do ponto de junção;
- Depois (*After*) executa depois do ponto de junção;
- Durante (*Around*) executa durante a execução do ponto de junção.

O adendo é estruturado da seguinte forma:

```
<tipo do adendo>: <nome de um conjunto de junção> ||
<expressão de pontos de junção>
```

#### 2.4.1.1.4 Declaração Intertipos (*Inter-type Declaration*)

Além de identificação e modificação em aspectos dinâmicos, o AspectJ possibilita a definição e alteração de tipos estaticamente. A declaração intertipos descreve e pode modificar os interesses estáticos (*static crosscuting*).

AspectJ possibilita a modificação de forma externa em classes, interfaces e aspectos do sistema. Esta modificação aparentemente pode ser considerada inútil, mas se considerarmos a modificação de uma classe em conjunto com a ação de um aspecto, a declaração intertipos pode se tornar surpreendente [35]. No entanto, a modificação estática de um sistema não afeta diretamente no seu comportamento, não sendo utilizado para avaliar ou observar o controle do comportamento de um sistema desnecessário para uso neste trabalho.

### 2.4.1.2 Exemplo de uso do AspectJ

Após a identificação do comportamento transversal, são definidos os pontos de junção referentes a ele, definindo o novo comportamento que os mesmos devem executar. Em seguida, os aspectos que irão modificar o comportamento de sistema devem ser definidos, contendo os conjuntos de junção que devem capturar os pontos de junção previamente definidos. Finalmente, os adendos devem ser definidos para os respectivos pontos de junção, contendo as ações preteridas para cada um deles [40].

Neste trabalho, o propósito do uso de AspectJ é a possibilidade de realizar *logging* sem ser intrusivo, definindo de forma fácil a captura de eventos em vários pontos de um sistema em execução. A seguir entenderemos o mecanismo de realizar *logging* com AspectJ.

#### 2.4.1.2.1 *Logging e Tracing*

*Logging* é uma técnica que pode ser usada para proporcionar o entendimento do comportamento de um software. *Logging* é a captura e o armazenamento de informações sobre parte ou todos os processos de um sistema, fazendo a exibição de mensagens com o intuito de descrever as operações executadas em um sistema. Seguindo a mesma linha, o *tracing* ou rastreamento de operações adiciona à atividade de registro a capacidade de vincular uma determinada execução a um usuário do sistema [40; 64].

A forma tradicional que a técnica de *logging* é usada exige que cada entidade que pre-

cise logar informação tenha o controle e a ação sobre a atividade de guardar a informação requerida. Esta exigência implica na modificação de cada entidade de forma a possibilitar a gravação da informação. De maneira geral, podemos entender a técnica tradicional de *logging* ilustrada na Figura 2.6 [40].

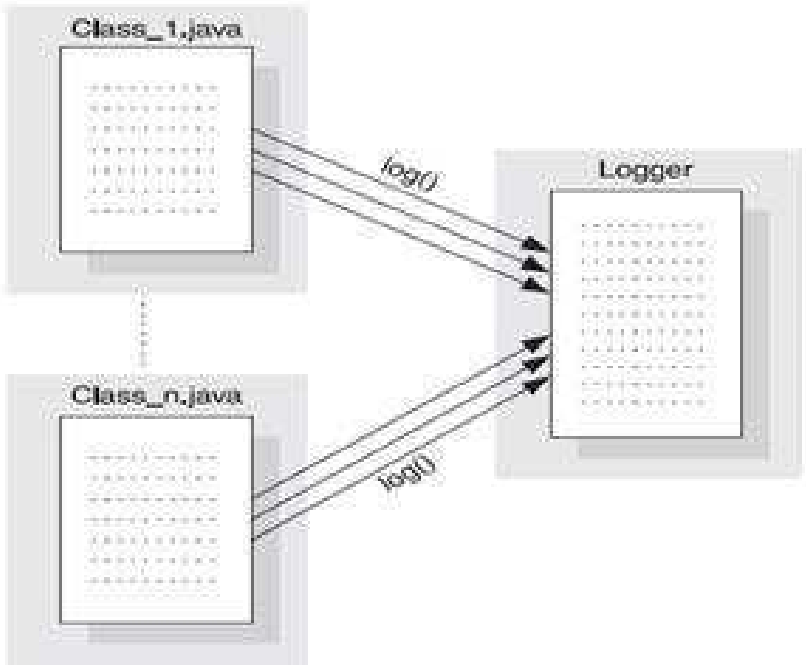



Figura 2.6: *Logging* de informação sem aspectos.

Como podemos ver, cada classe é a responsável por armazenar todas as informações referente a ela, propiciando mais uma responsabilidade para uma única entidade. Outra desvantagem é a necessidade de inserir código da técnica de *logging* em todas as entidades que precisam guardar informações, dificultando a manutenção do software com um interesse que atravessa grande parte ou todos os componentes de um software.

Uma forma de separar este interesse transversal para a técnica de *logging* é usando o conceito da orientação a aspectos [40]. Usando aspectos é possível inserir a técnica de *logging* sem modificar as classes das quais precisamos extrair informação. Para tanto, definiram-se aspectos que fazem a ligação entre o código em execução a ser logado e a classe que armazena todas as informações de *logging*. Podemos entender o mecanismo usando a orientação a aspectos observando na Figura 2.7.

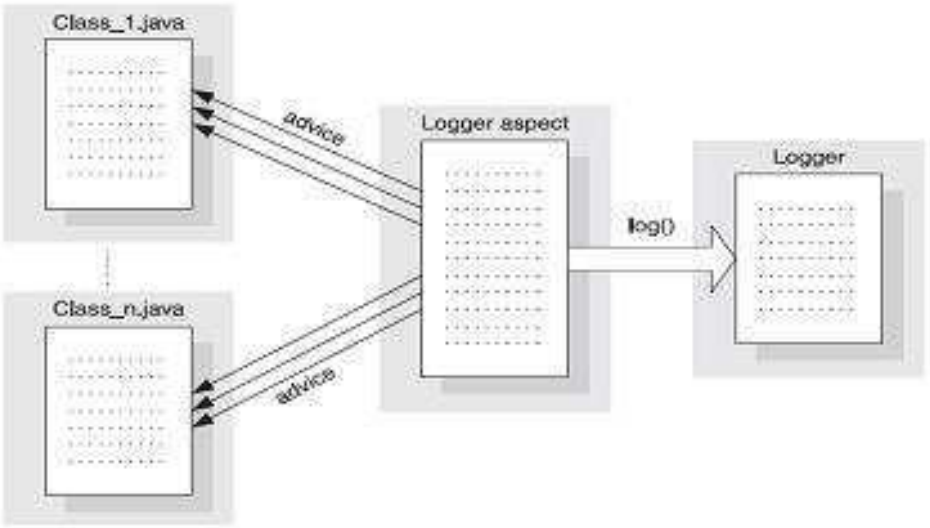



Figura 2.7: *Logging* de informação utilizando o conceito de aspectos

Usando o conceito de orientação a aspectos conseguimos separar o código do sistema implementado do interesse de armazenar as informações por ele executadas.

A realização do *logging* de informações é possível usando os aspectos. No exemplo de aspecto, no Código 2.1, podemos ver um exemplo de aspecto que pode ser utilizado para o armazenamento de informações durante a execução de uma aplicação.

Código Fonte 2.1: Código de um aspecto de *logging* das execuções de uma aplicação.

```
1  import org.aspectj.lang.*;
2  import logging.*;
3   public aspect LoggingAspect {
4    protected pointcut loggedOperations():
5      (execution(* *.*(..))
6                  || execution(*.new(..)));
7            before() : loggedOperations() {
8                    Signature sig = thisJoinPointStaticPart
9                                  .getSignature();
10                   System.out.println("Entering ["
11                           + sig.getDeclaringType().getName() + "."
12                           + sig.getName() + "]");
13                 }
14         }
```

## 2.5 Engenharia de Software Experimental

Na Engenharia de Software existe uma discussão sobre sua consideração como ciência ou engenharia. Esta questão se deve ao duplo caráter do software. Por um lado a Engenharia de Software considera questões técnicas, tais como linguagens de programação, sistemas operacionais, sintaxe e semântica. Por outro lado, ela possui questões humanas, sociais e psicológicas, características da engenharia e da produção [65].

Metodologias específicas são necessárias para ajudar a estabelecer uma base de engenharia e de ciência para a Engenharia de Software. Desta forma, existem quatro métodos relevantes de avaliação para pesquisas: científico, de engenharia, experimental e analítico [65]. O método científico se baseia na observação do mundo e na construção de um modelo baseado nestas observações. Já no método de engenharia, as técnicas atuais são analisadas, suas fraquezas são identificadas, inovações são propostas e comparadas com as técnicas que as precedem. No método experimental, um modelo para o mundo real é proposto e avaliado através de um conjunto de estudos experimentais. Por fim, o método analítico propõe uma teoria formal, onde resultados são derivados e comparados com observações do mundo real. Os métodos de engenharia e experimental são considerados derivações do método científico [65].

Segundo Basili [10], o método científico (e suas derivações) é tradicionalmente aplicado com sucesso em outras ciências, especialmente as sociais, onde raramente é possível estabelecer leis da natureza, como na física ou matemática. Como o fator humano é muito importante na construção e manutenção de software, a Engenharia de Software se aproxima destas ciências sociais. Sendo assim, este método é comumente aplicado para avaliar os benefícios providos por uma nova técnica, teoria ou método relacionado com software.

O tipo de experimento mais apropriado em uma situação vai depender, por exemplo, dos objetivos do estudo, das propriedades do processo de software usado durante a experimentação, ou dos resultados finais esperados. Ao mesmo tempo, a classificação dos experimentos está sempre relacionada aos conceitos das estratégias empíricas. Em síntese, Pfleeger [51] define as seguintes abordagens:

- **Análise das características**: é a mais simples e corresponde a uma abordagem subjetiva, utilizada para atribuir um valor e classificar os atributos de vários métodos,

visando decidir qual utilizar. Esta abordagem corresponde a um estudo em retrospectiva e é útil para estreitar o leque de opções a serem escolhidas, porém não avalia o comportamento em termos de causa e efeito;

- **Pesquisa de opinião (*survey*)**: é um estudo em retrospectiva que visa documentar as relações e os resultados de certa situação. Durante a realização da pesquisa, registram-se as informações sobre uma situação, comparando-as com informações semelhantes. Não ocorre à manipulação de variáveis neste estudo;

- **Estudo de Caso**: ao contrário das anteriores, esta abordagem define previamente o que se deseja investigar, identificando os principais fatores que possam afetar o resultado de uma atividade. Após sua execução, é realizada a documentação de suas entradas, restrições, recursos e saídas. Esta abordagem é utilizada principalmente para observar projetos ou atividades, sem muito controle sobre o objeto de estudo;

- **Experimento**: representa o tipo de estudo mais controlado, geralmente realizado em laboratórios. Nesta abordagem, os valores das variáveis independentes (entradas do processo de experimentação, também chamadas de fatores) são manipulados para se observar as mudanças nos valores das variáveis dependentes (saídas do processo de experimentação). As variáveis independentes (apresentam a causa que afeta o resultado do processo de experimentação, já as variáveis dependentes referem-se ao efeito que é causado pelos fatores do experimento, que é o resultado. Ao término da execução do experimento, os resultados são analisados, interpretados, apresentados e, por fim, empacotados. A Figura 2.8, obtida em [12], apresenta os relacionamentos entre os conceitos descritos acima.


Figura 2.8: Os conceitos de um experimento.

O processo de experimentação compreende a realização de diferentes atividades. O número e a complexidade destas atividades podem variar de acordo com as características do estudo. A literatura [36; 65] propõe os seguintes passos:

- **Definição**: é a primeira fase, que corresponde à definição clara das hipóteses e dos objetivos a partir do problema a ser resolvido.

- **Planejamento**: corresponde a fundamentação do experimento, onde seu contexto é explicitado em detalhes. Este passo compreende a formalização da hipótese nula (hipótese fundamental na qual o experimento objetiva rejeitar em favor das demais) e das alternativas (demais hipóteses que o experimento visa apoiar), determinação das variáveis independentes e dependentes, seleção dos participantes, preparação conceitual da experimentação e a consideração sobre a validade do experimento;

- **Execução**: este passo compreende a preparação, execução e validação dos dados. É durante esta etapa que ocorre a interação humana, sendo necessário preparar os participantes sob o ponto de vista moral e metodológico, evitando assim resultados errôneos ou desinteressantes.

- **Análise e Interpretação**: com a entrada de dados no passo anterior, é possível a análise e interpretação do experimento. Para tanto, ocorre à compreensão dos dados e a verificação das hipóteses utilizando estatística descritiva, possibilitando assim conclusões sobre a validade do experimento;

- **Empacotamento**: este passo compreende a documentação dos resultados e a estruturação do experimento, visando possibilitar sua replicação.

  A seguir, na Figura 2.9, temos a ilustração do processo de experimentação com todas as fases citadas anteriormente.


Figura 2.9: Processo de experimentação. Fonte: [12].

É importante esclarecer que um experimento não irá proporcionar uma resposta final para uma questão, porém uma resposta específica para uma determinada configuração. Nas próximas seções, serão detalhados os passos para condução do experimento.

### 2.5.1 Definição

A fase de definição corresponde à fundamentação dos objetivos do experimento e a definição de seu escopo, identificando os objetos e os grupos de estudo envolvidos. Que pode ser feita utilizando uma abordagem de [54] chamada *Goal Question Metric* (GQM), que corresponde a uma medição para melhoria de processo de software dirigida por objetivos. Esta abordagem parte da definição dos objetivos (nível conceitual) para o estabelecimento de questões (nível operacional), que tentam caracterizar o processo em termos de métricas (nível quantitativo).

### 2.5.2 Planejamento

Para que o processo de experimentação seja bem sucedido, é necessário um planejamento de todas as atividades envolvidas. Esta fase pode ser dividida em sete passos:

1. Seleção do contexto: com base em sua definição, este passo seleciona o ambiente no qual o experimento será executado;

2. Formulação das hipóteses: um experimento é normalmente formulado através de hipóteses, que representam a base para a análise estatística.

3. Seleção das variáveis: este passo compreende a definição das variáveis independentes e dependentes do experimento, isto é, as entradas e saídas do processo que será observado;

4. Seleção dos indivíduos: esta atividade se relacionada com a generalização dos resultados do experimento, com o objetivo de definir um conjunto representativo de indivíduos;

5. Projeto: determina a forma na qual o experimento será conduzido, incluindo seus objetos e seus participantes. A escolha do projeto correto é crucial para a validade das conclusões;

6. Instrumentação: representa a implementação prática, fornecendo os meios de como conduzir e monitorar o experimento;

7. Análise da Validade: realiza a avaliação dos resultados do experimento. Esta análise inclui a validade interna e externa, além da validade de sua construção e de sua conclusão.

#### 2.5.2.1 Seleção do Contexto

O contexto é composto pelas condições na qual o experimento será executado. A seleção adotada contempla as seguintes dimensões:

- **Processo**

*In-vitro ou In vivo*: o primeiro corresponde à experimentação controlada em um laboratório e o segundo em um projeto real;

- **Participantes**:

  *Alunos ou Profissionais*: define os indivíduos que farão parte do experimento;

- **Realidade**:

  *Problema desenvolvido em sala de aula ou Problema real*: define o tamanho do problema que será estudado;

- **Generalidade**:

  *Específico ou Geral*: apresenta o escopo da validade do experimento, em âmbito específico e contextualizado ou em todo o domínio da Engenharia de Software.

#### 2.5.2.2 Formulação das Hipóteses

A formulação das hipóteses corresponde à definição formal sobre o que se pretende com o experimento. Conforme já apresentado, a hipótese fundamental se chama Hipótese Nula $(H_0)$ e representa a não ocorrência da relação de causa e de efeito do experimento, ou seja, a não derivação de seus objetivos. Portanto, o objetivo do experimento é rejeitá-la em prol de uma ou mais hipóteses alternativas $(H_1, H_2, \ldots, H_n)$.

#### 2.5.2.3 Seleção das variáveis

A escolha das variáveis para o experimento não é uma tarefa trivial e geralmente requer certo conhecimento sobre o domínio do problema. É necessário que as variáveis independentes (entrada da experimentação) sejam controladas e que exerçam alguma influência sobre as variáveis dependentes (saída da experimentação).

Um experimento compreende observações sobre a alteração de uma ou mais variáveis independentes, também chamadas de fatores. Durante o estudo, é definido um fator e as demais variáveis independentes são fixadas. Isso é necessário para saber qual fator é responsável pelo fenômeno observado, também chamado de tratamento, que são aplicados para a combinação de indivíduos e objetos.

#### 2.5.2.4 Seleção dos Indivíduos

A seleção dos indivíduos é particularmente importante para a geração de resultados relevantes durante o experimento. Para tanto, é necessário escolher uma população representativa.

#### 2.5.2.5 Projeto do experimento

Para se obter conclusões significativas em um experimento, é necessário recuperar cuidadosamente os dados e aplicar métodos de análise estatística. O projeto do experimento corresponde à forma pela qual todas as atividades serão planejadas e conduzidas para a obtenção destas conclusões.

O experimento consiste em uma série de testes sobre tratamentos de variáveis independentes. Neste contexto, o objetivo da fase de projeto consiste em identificar o número de vezes que os testes são executados para que se torne visível os efeitos dos tratamentos sobre as variáveis.

#### 2.5.2.6 Instrumentação

O objetivo da instrumentação é proporcionar os meios para condução do experimento e sua análise. Sugerem-se as seguintes definições de instrumentos:

- **Objetos**: os objetos podem ser, por exemplo, documentos de especificação ou código fonte;
- **Guias**: são especialmente úteis para apoiar os participantes no experimento e incluem, por exemplo, descrição de processos, tutoriais e checklists;
- **Métricas**: são obtidas no experimento através da coleta de dados, normalmente através de entrevistas ou formulários preenchidos pelos participantes.

#### 2.5.2.7 Análise da Validade

Um ponto crítico durante o experimento é a análise de sua validade. Sugere-se que esta preocupação ocorra desde o seu planejamento, prevendo algumas questões sobre a avaliação do experimento. A literatura [36] sugere quatro tipos de validação dos resultados:

- **Validade interna:** define se o relacionamento observado entre o tratamento e o resultado é causal e não resultado de algum fator não previsto. A atenção principal é dada aos participantes durante a condução do experimento;

- **Validade externa:** sugere a generalização dos resultados obtidos durante o experimento em práticas industriais. É avaliada a representatividade dos participantes com relação ao público alvo;

- **Validade de construção**: avalia a relação de causa e efeito na qual o experimento é idealizado. A atenção desta validade se concentra em mapear a teoria, que motiva o experimento, nos indivíduos. Ao término, são observados os efeitos da experimentação;

- **Validade da conclusão**: esta validade corresponde à capacidade de chegar a uma conclusão correta a respeito dos tratamentos e dos resultados do experimento. Para tanto, é necessário escolher os testes estatísticos, os participantes e a confiabilidade das medidas e da implementação dos tratamentos.

### 2.5.3 Execução

A execução compreende a etapa operacional da experimentação, onde ocorre o envolvimento direto dos participantes. Ainda que o experimento seja perfeitamente planejado e os dados coletados sejam analisados com os métodos mais apropriados, o resultado será inválido se os indivíduos não se engajarem de maneira séria com o experimento.

A execução do experimento compreende três passos:

1. **Preparação**: corresponde a escolha dos participantes e dos materiais preparados para a experimentação;

2. **Execução**: é quando os participantes executam as tarefas do experimento, de acordo com os diferentes tratamentos previstos;

3. **Validação dos dados**: corresponde a validação dos dados coletados após a execução do experimento.

#### 2.5.3.1 Preparação

Existem alguns preparativos que devem ser cautelosamente analisados antes da execução do experimento. Dois pontos são avaliados nesta etapa: prover a informação aos participantes e preparar os materiais necessários ao experimento, tal como formulários e ferramentas.

#### 2.5.3.2 Execução

A execução de um experimento pode se dar de diferentes formas. Sua duração é variável e pode ocorrer em um período de tempo curto, no qual o responsável está presente em todos os detalhes da execução. Outra perspectiva é quando o tempo é longo, tornando inviável o responsável participar de cada detalhe da execução do experimento.

#### 2.5.3.3 Validação dos Dados

Após a coleta de dados, é necessário verificar se os dados são razoáveis e se foram coletados corretamente. Isso lida com aspectos específicos, como o entendimento dos participantes sobre os formulários e seu correto preenchimento, com a seriedade em que os participantes conduziram o experimento, com a remoção de dados a partir da análise, etc. É importante revisar se o experimento foi conduzido conforme o planejado e, para cada erro percebido, os dados devem ser considerados inválidos.

### 2.5.4 Análise e Interpretação

Após a coleta de dados experimentais gerados na fase anterior, é necessário extrair conclusões. Para se recuperar conclusões válidas, precisamos interpretar os dados experimentais.

A primeira observação sobre os dados brutos obtidos no experimento diz respeito à medida de suas escalas. As escalas determinam quais operações que podem ser executadas sobre os valores das variáveis. A literatura define os seguintes tipos de escalas:

- **Nominal**: representam diferentes valores que não possuem interpretação numérica nem ordenação;
- **Ordinal**: representam diferentes valores que podem ser ordenados, porém não possuem uma representação numérica;

- **Intervalar**: representam diferentes valores que podem ser ordenados e a distância entre os números possui a mesma interpretação;

- **Razão**: representam diferentes valores que podem ser ordenados e a distância e razão entre os mesmos pode ser interpretada.

Após a análise das medidas das escalas para as variáveis, é importante avaliar a apresentação e o processamento numérico dos dados obtidos através da estatística descritiva. Posteriormente, sugere-se a eliminação de distorções que comprometam a validade das conclusões. Por fim, é realizado o teste das hipóteses para extração das conclusões do experimento. A estatística descritiva lida com a apresentação e o processamento numérico de um conjunto de dados. Para este fim, sugere-se representar graficamente estes dados para identificação de alguns aspectos interessantes, como os indicadores de medidas. O objetivo da estatística descritiva é analisar a distribuição geral de um conjunto de dados. Esta etapa deve ser executada antes da validação das hipóteses para compreender a natureza dos dados e identificar os dados anormais ou os valores extremos (*outliers*).

#### 2.5.4.1 Teste de Hipótese

Um estudo experimental tem como objetivo colher dados para confirmar ou negar uma hipótese. Em geral, são definidas duas hipóteses, a hipótese nula $H_0$ e a alternativa $H_1$. Os testes de hipótese verificam se é possível rejeitar a hipótese nula, de acordo com um conjunto de dados observados e suas propriedades estatísticas. Com relação a este teste, existem dois tipos:

- **Paramétricos**: utilizam formas fechadas, derivadas de propriedades de distribuições de freqüências conhecidas e exigem que os dados possuam normalidade e homocedasticidade;

- **Não-Paramétricos**: devem ser usados quando os dados não atendem os requisitos de normalidade e de homocedasticidade. Esta abordagem utiliza rankings de valores observados como parâmetro ao invés dos valores propriamente ditos.

A escolha do teste de hipóteses adequado demanda a análise de dois requisitos: normalidade e da homocedasticidade. Neste contexto, apresentamos:

- **Normalidade**: os valores tendem a se concentrar próximos de uma média, e quanto maior a distância dessa média, menor a freqüência das observações;

- **Homocedasticidade**: implica em uma variância constante entre o conjunto de dados que serão testados.

Para verificar a normalidade, precisamos fazer um teste de normalidade. Neste contexto, existem dois tipos:

- **Teste de Kolmogorov-Smirnov (K-S)**

  Avalia se duas amostras têm distribuições semelhantes ou se uma amostra tem distribuição semelhante a uma distribuição clássica (como a distribuição normal, por exemplo). Este teste é utilizado para identificar normalidade em variáveis com pelo menos trinta valores.

- **Teste de Shapiro-Wilk**

  Calcula um valor W, que avalia se uma amostra xi segue a distribuição normal. Este teste é utilizado para identificar normalidade em variáveis com menos de cinquenta valores. É usado para pequenos conjuntos de dados, onde os valores extremos podem dificultar o uso do teste K-S.

Um conjunto de variáveis é homocedástico se as variáveis possuem variâncias similares. Para testar a homocedasticidade de um conjunto de dados, usamos o Teste de Levene.

- **Teste de Levene**

  Considere uma variável Y, com N valores divididos em K grupos, onde Ni é o número de valores no grupo i. O Teste de Levene aceita a hipótese de que as variâncias são homogêneas se o valor W for menor do que o valor da distribuição F.

Depois de avaliada a normalidade e a homocedasticidade, sugerem-se dois tipos de teste de acordo com os supostos paramétricos:

- **Teste T ou Student-T**: este teste é utilizado para comparar duas médias a partir de uma hipótese nula. Esta hipótese pressupõe que não existem diferenças significativas entre dois grupos. É sugerido usar em um projeto com um fator e dois tratamentos;

- **ANOVA**: técnica estatística cujo objetivo é testar a igualdade entre as médias de dois ou mais grupos. Supõe independência, normalidade e igualdade entre as variâncias dos grupos. É usada como uma extensão do Teste T, no caso se mais de dois tratamentos.

Caso algum dos princípios de normalidade ou homocedasticidade for violado, o teste se configura como não paramétrico, então se utiliza os seguintes testes:

- **Teste de Mann-Whitney**: alternativa não paramétrica para o Teste T. Este teste requer que as amostras sejam independentes, com dados contínuos e nas escalas ordinal, intervalar ou razão. Para a realização do teste, as observações das amostras são transformadas em rankings dentro do grupo e calcula-se o somatório dos rankings da menor amostra (T), e finalmente calcula-se Z que é comparado com uma tabela de valores.

- **Teste de Kruskal-Wallis**: alternativa não paramétrica para a análise de variância. Como grande parte dos testes não paramétricos, este se baseia na substituição dos valores por seus rankings no conjunto de todos os valores.

### 2.5.5 Empacotamento

Após a análise e interpretação, sugere-se o empacotamento do experimento devido à sua necessidade de replicação. Ao executar o experimento em contextos distintos, possibilita-se a aquisição de novos conhecimentos a respeito dos conceitos estudados. Para isso, é importante documentar os diversos aspectos relacionados com o experimento, entre eles:

- **Comunidade**: apresenta os pesquisadores e participantes relacionados com o experimento, além dos interessados que possam aproveitar os resultados obtidos;

- **Organização**: apresenta o planejamento, projeto e demais informações referentes à preparação, diretrizes, execução e instrumentação do experimento;

- **Artefatos**: apresenta os artefatos definidos durante a instrumentação do experimento;

- **Resultados**: incluem a descrição detalhada dos resultados recebidos, com os dados brutos, refinados (pela eliminação dos outliers) e analisados. Os dados analisados são utilizados para testar as hipóteses e fazer as conclusões sobre o experimento.

## 2.6 Considerações Finais

Os fundamentos teóricos apresentados neste capítulo mostram os conceitos usados no nosso trabalho, fornecendo subsídios para se entender como a técnica proposta foi desenvolvida, implementada e avaliada. No próximo capítulo iremos nos aprofundar na técnica proposta e sua implementação, fazendo uso dos conceitos abordados no presente capítulo. Por fim, no Capítulo 4 utilizaremos os conceitos aqui abordados sobre Engenharia de Software Experimental.

# Capítulo 3

# Técnica Para Geração Automática de Testes com Objetos *Mock*

Neste capítulo apresentamos a técnica de geração automática de testes com objetos *mock*, a qual se divide em três fases: Análise Estática, Análise Dinâmica e Geração de Código de Teste com Objetos *Mock*. A técnica proposta permite a geração automática de código de testes com objetos *mock* a partir de um código de teste inicial (uma classe de teste), que não contenha objetos *mock*.

Iniciamos este capítulo com uma visão geral da técnica. Em seguida, detalhamos as fases necessárias para produção de código de teste com objetos *mock*. Depois, apresentamos um exemplo da aplicação da técnica. E por fim, detalhamos a implementação da técnica.

## 3.1 Visão Geral

A técnica proposta identifica automaticamente interações entre objetos em um dado cenário de teste. Ela analisa pontos de execução, identificando todas as interações e fluxos de dados entre a CUT e seus colaboradores (ilustrados na Figura 2.5), resultando em um *log* contendo estas informações, a partir do qual podemos gerar código de teste com objetos *mock*. Este código consiste de um teste de unidade para a CUT, onde todas suas entidades colaboradoras, que inicialmente eram reais, se tornam objetos *mock*, simulando o comportamento destas. Desta forma, o isolamento da CUT é garantido (ilustrado na Figura 2.5).

Esta técnica é composta de três fases, como ilustrado na Figura 3.1:

- **Análise Estática**: identificação dos objetos que colaboram com a CUT no cenário do teste inicial;
- **Análise Dinâmica**: instrumentação e execução do teste inicial com a finalidade de capturar e gravar as interações existentes entre a CUT e os colaboradores identificados na fase anterior;
- **Geração de Código de Teste com Objetos *Mock***: as interações gravadas são usadas para gerar código de teste com objetos *mock* no teste inicial.

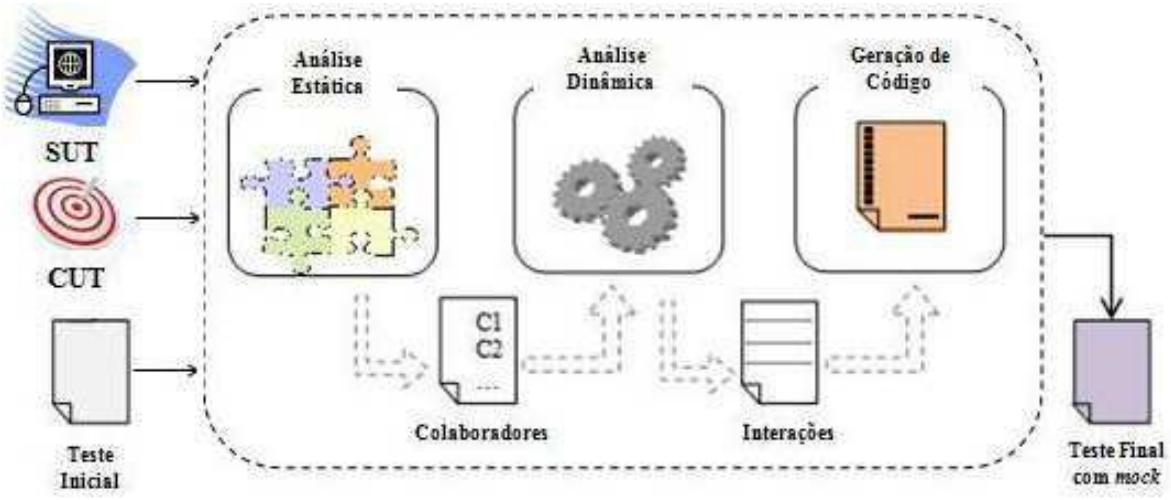



Figura 3.1: Fases da técnica proposta

## 3.2 Primeira Fase: Análise Estática

A primeira fase da técnica proposta consiste numa análise de dependência [19] em primeiro nível, que corresponde a maneira de identificar as dependências da CUT, ou seja, as entidades do sistema que interagem com a CUT durante o cenário do teste inicial, o qual queremos isolar. O que será feito através de uma análise estática no código do teste. Na Seção 2.3, apresentamos vários tipos de análise estática que podem ser realizadas no código. Dentre elas, optamos pela análise estática da arquitetura, que se trata de uma análise para verificar a estrutura do código. No caso da técnica, esta análise é feita através de uma varredura no código fonte em busca de entidades/classes que se comuniquem com a CUT no cenário do teste inicial, como ilustrado na Figura 3.2.


Figura 3.2: Funcionamento da Análise Estática

Esta análise no código consiste em ler cada linha do código fonte do teste e procurar entidades/classes que se comuniquem com a CUT durante o teste. Esta busca é feita com base em regras definidas que representam as estruturas e palavras-chave que indicam a criação de classes, como o "*new*", por exemplo, estruturas que indicam relacionamento, como um método, por exemplo. Estas regras são:

1. Se a linha de código não contiver a CUT e contiver a estrutura "*new*", então se trata da criação/declaração de uma entidade no teste, que pode ser um possível colaborador;

2. Se a linha de código contiver a CUT e se esta invocar algum método com parâmetro, então verificamos se esse parâmetro é alguma entidade do sistema (*System Under Test* - SUT). Se sim, então se trata de um relacionamento e essa entidade é um possível colaborador.

Desta forma, o nome e o tipo desta entidade são armazenados na lista de colaboradores. O Código 3.1 abaixo mostra o pseudocódigo dessas regras, as linhas 2 a 7 se referem a primeira regra, e as linhas 8 a 14 correspondem a segunda regra.

Código Fonte 3.1: Pseudocódigo das regras da Análise Estática.

```
1 PARA CADA linha de código em teste FAÇA
2   SE linhaDeCodigo.naoContem(cut)
```

```
3    && linhaDeCodigo.contem("new") ENTÃO
4            entidade.nome = linhaDeCodigo.getNomeEntidade()
5      entidade.tipo = linhaDeCodigo.getTipoEntidade()
6   FIM_SE
7   SENÃO SE linhaDeCodigo.contem(cut)
8             && cut.invocaMetodoComParametro() ENTÃO
9    SE tipoParametro.pertenceAoSistema() ENTÃO
10           entidade.nome = parâmetro.getNome()
11           entidade.tipo = parâmetro.getTipo
12          FIM_SE
13  FIM_SENÃO_SE
14  colaboradores.add(entidade)
15 FIM_PARA
```

## 3.3 Segunda Fase: Análise Dinâmica

A segunda fase da técnica proposta consiste em capturar e gravar todas as interações entre a CUT e seus colaboradores previamente identificados na fase 1, em tempo de execução, resultando em um *log* de interações. Para capturar e gravar estas interações, instrumentamos o código do teste inicial, por meio da programação orientada a aspectos. Posteriormente, executamos o teste instrumentado e obtemos um arquivo de *log*, contendo as interações entre a CUT e seus colaboradores. Todas estas atividades referentes à Análise Dinâmica podem ser ilustradas na Figura 3.3.

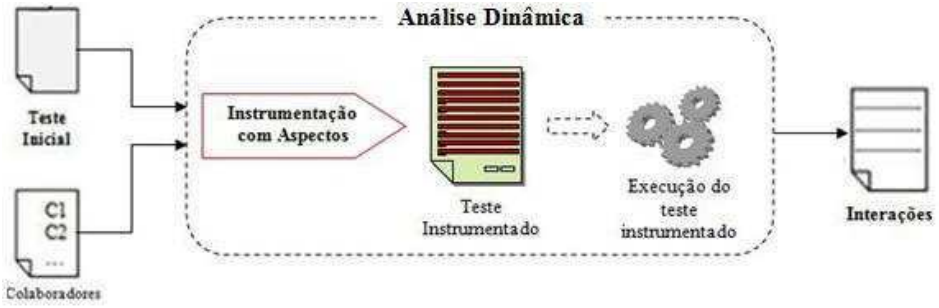



Figura 3.3: Funcionamento da Análise Dinâmica

Durante a Análise Dinâmica verificamos a comunicação entre a CUT e seus colaboradores, através de mecanismos de reflexão e instrumentação de código. Utilizamos

a programação orientada a aspectos como mecanismo de instrumentação, uma vez que ela pode capturar informações do código interferindo minimamente na execução do sistema monitorado, através da linguagem AspectJ [40], que implementa essa captura de informações em sistemas codificados em Java sem a necessidade de modificar o código fonte do mesmo.

Para realizar a captura das informações que precisamos, ou seja, das interações entre a CUT e seus colaboradores no teste inicial, primeiramente definimos como sendo uma interação: algum relacionamento entre a CUT e um colaborador durante a execução do teste inicial. Para detectar essas interações, definimos um ponto de corte genérico, o qual reúne todos os pontos de junção, que contém chamadas aos métodos do teste e pontos de execução destes métodos de teste, como ilustrado no Código 3.2

Código Fonte 3.2: Código referente ao ponto de corte.

```
1 pointcut testClass(): cflow(execution(* testCase.*(..)))
2                       && call(* *.*(..))
3                       && !(within(Automock))
```

No ponto de corte ilustrado observamos a necessidade de inserir uma instrução para, dentre as chamadas e execuções especificadas, não incluir o código da técnica implementada, a qual chamamos de Automock, e incluir apenas o código do SUT. Desta forma, a técnica é responsável por gerar um ponto de corte para cada caso de teste ("*testCase*") do teste inicial com base no ponto de corte genérico demonstrado acima, onde a execução de cada caso de teste e suas chamadas a métodos no SUT serão capturadas, através do mecanismos de reflexão do próprio aspecto, que chamamos de *thisJoinPoint* em AspectJ, a partir do qual podemos extrair informações sobre toda a execução dos pontos de junção e das chamadas a métodos capturadas.

Para se extrair as informações relativas à execução de cada caso de teste e suas chamadas a métodos utilizamos os métodos providos pelo *thisJoinPoint*, cujas funcionalidades estão descritas na Tabela 3.1.

| Métodos do *thisJoinPoint* | Funcionalidade |
|---|---|
| getSourceLocation() | Localiza a linha de código capturada |
| getSignature() | Retorna a assinatura da linha de código capturada |
| getTarget() | Retorna a referência atual do objeto da linha de código capturada |
| getArgs() | Retorna os argumentos dos métodos identificados nas linhas capturadas, incluindo seu tipo, nome e valor |

Tabela 3.1: Funcionalidades de alguns métodos providos pelo *thisJoinPoint*

A técnica também provê um aspecto, ilustrado no Código 3.3, para capturar as informações providas pelo *thisJoinPoint* antes da execução de cada caso de teste e suas chamadas a métodos.

Código Fonte 3.3: Código referente ao aspecto para captura de informações.

```
1 before(): testClass(){
2         thisJoinPoint.getSourceLocation();
3         thisJoinPoint.getSignature();
4         thisJoinPoint.getTarget();
5         thisJoinPoint.getArgs();
6 }
```

Após a geração do aspecto com os pontos de corte para cada caso de teste, estes aspectos precisam ser inseridos juntos ao código do SUT, que contém o teste inicial. Antes da execução do teste inicial, no SUT, é necessária a compilação do código fonte juntamente com os aspectos, possibilitando a construção do código fonte instrumentado com o *bytecode* dos aspectos inseridos a fim de capturar as informações em tempo de execução.

Depois do código do teste ser instrumentado, a técnica se encarrega de executá-lo. Durante esta execução o aspecto entra em ação e captura todas as informações providas pelo *thisJoinPoint*. Após esta captura, as informações são armazenadas em um arquivo de *log* XML, que são as interações.

## 3.4 Terceira Fase: Geração de Código de Teste com Objetos *Mock*

De posse do arquivo de *log*, que contém as interações, partimos para a terceira fase da técnica, a qual consiste em utilizar estas interações para gerar código *mock* para os colaboradores da CUT no contexto do teste inicial, como podemos observar na Figura 3.4. Objetos mock são gerados diretamente sem o uso de templates para geração, apenas o EasyMock para dar suporte.

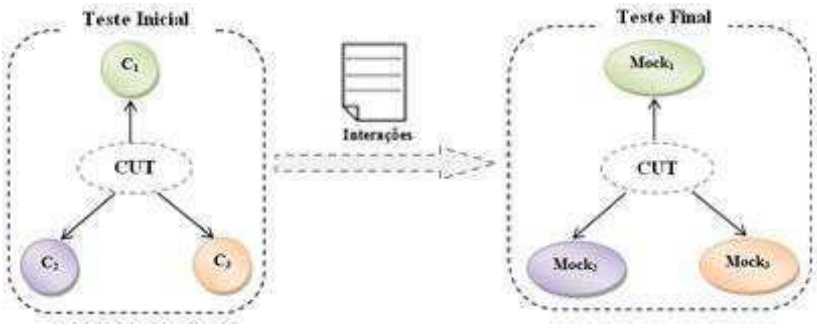



Figura 3.4: Geração de Código *Mock* para os Colaboradores através das interações gravadas no *log*.

As interações contidas no *log* correspondem aos fluxos de execução de cada linha de código do teste inicial. A partir desse *log*, a técnica cria um mapa de fluxos, que tem por chave o número da linha de código do teste inicial, e por valor, todas as linhas de código do SUT que foram exercitadas durante a execução desta linha de código do teste inicial, as quais chamamos linhas de código de subfluxo, conforme ilustrado na Tabela 3.2. A partir deste mapa, a técnica pode verificar quais linhas de cada subfluxo correspondem as interações entre a CUT e algum colaborador e cria códigos de objetos *mock* para cada interação.

| Linha de código do teste inicial | Subfluxo correspondente a linha |
|---|---|
| N° Linha 1 | **Linha 1 do subfluxo** |
| | **Linha 2 do subfluxo** |
| | **Linha 3 do subfluxo** |
| N° Linha 2 | **Linha 1 do subfluxo** |
| | **Linha 2 do subfluxo** |
| N° Linha 3 | **Linha 1 do subfluxo** |
| | **Linha 2 do subfluxo** |
| | **Linha 3 do subfluxo** |
| | **Linha 4 do subfluxo** |
| | **Linha 5 do subfluxo** |
| N° Linha n | ... |

Tabela 3.2: Mapa de fluxos de execução de cada linha de teste.

Para concretizar o mapa, ilustrado na Tabela 3.2, temos na Figura 3.5 um teste sem objetos *mock* que é transformado em um teste com objetos *mock*, através da técnica proposta. Desta forma, podemos observar que as linhas marcadas na cor azul correspondem as linhas de código do teste inicial e as linhas marcadas na cor amarelo correspondem as linhas do subfluxo correspondente a cada linha do teste inicial, que são transformadas em código de teste com objetos *mock* automaticamente.

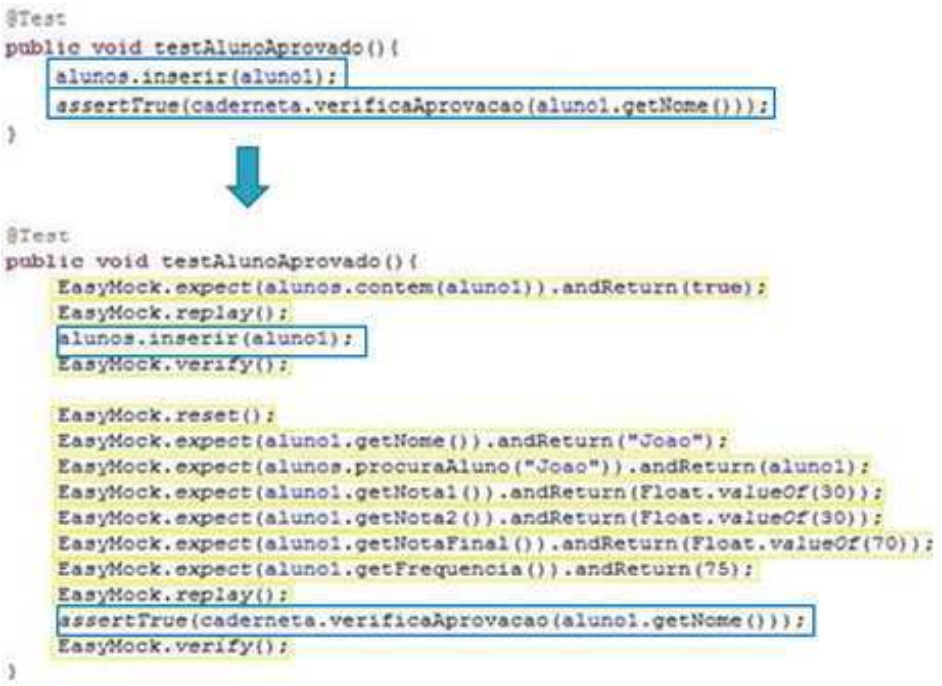



Figura 3.5: Geração de Código *Mock* para os Colaboradores através das interações gravadas no *log*.

Um teste com objetos *mock* funciona da seguinte forma: as interações existentes entre a CUT e seus colaboradores são transformadas em código de objetos *mock*, que refletem o comportamento destas interações; depois, este comportamento é gravado e executa-se a linha de código do teste inicial para poder verificar se esta linha reflete o comportamento dos objetos *mock*. Na verdade, o código referente às interações é o subfluxo de execução de cada linha de código do teste inicial que contenha algum colaborador. É desta forma que nosso algoritmo de geração funciona.

Este algoritmo, apresentado no Código 3.4, itera sobre todas as linhas de código do teste inicial (linha 1), verificando se cada uma contém algum colaborador (linha 2), caso positivo, procura esta linha no mapa de fluxos e verifica se existe algum subfluxo para ela (linha 3), se existir, o algoritmo passa a iterar sobre as linhas do subfluxo da linha do teste inicial (linha 4).

Código Fonte 3.4: Pseudocódigo para o algoritmo da geração de código.

```
1  PARA CADA linha de código do teste FAÇA
2   SE linhaDeCodigo.contem(colaborador) FAÇA
3    listaSubfluxo = mapaSubfluxos.recupera(linhaDeCodigo.numero)
4          PARA CADA linha de código do subfluxo FAÇA
5           SE linhaSubfluxo.naoContem(CUT)
6              && linhaSubfluxo.contem(colaborador)FAÇA
7            SE linhaSubfluxo.contem("new") ENTÃO
8                   linhaDeCodigoTesteFinal =
9                         "CriaMock(<nomeColaborador>.class)()"
10           FIM_SE
11           SENÃO
12            linhaDeCodigoTesteFinal +=
13                  "espera.(<colaborador.metodoInvocado>())
14                        .eRetorna(<valorDeRetorno>)"
15     FIM_SENÃO
16    FIM_SE
17   FIM_PARA
18   linhaDeCodigoTesteFinal += "grava()" //replay()
19   linhaDeCodigoTesteFinal += linhaDeCodigo
20   linhaDeCodigoTesteFinal += "verifica()" //verify()
21  FIM_SE
```

22 FIM_PARA

Ao iterar sobre as linhas do subfluxo da linha do teste inicial, o algoritmo procura por algum construtor ou alguma chamada a método envolvendo um colaborador (linhas 5 e 6). Se encontrar um construtor, a técnica provê uma estrutura de criação de objetos *mock* para este colaborador (linhas 7 a 10). Se for uma chamada a método, a técnica provê uma estrutura para expectativas de chamadas a métodos, como também para checagem de retornos destes métodos, e todas essas estruturas correspondem a novas linhas de código, as quais irão formar um teste final (linhas 11 a 15). Outros tipos de estrutura não são tratadas neste algoritmo, como: declaração ou importação de pacotes, ou ainda declaração de variáveis, por exemplo, mas são contemplados pela técnica em outros momentos.

Posteriormente, a técnica provê um comando para gravar o comportamento do código refletido por estas estruturas, que são linhas de código também, e adiciona ao teste final (linha 18), depois repete a linha de código do teste inicial, já dentro de um comando *assert* (linha 19), e adiciona um comando para verificar se o comportamento gravado corresponde ao da linha de código do teste inicial (linha 20). Como resultado, temos uma nova versão do teste inicial, desta vez com objetos *mock* ao invés dos colaboradores reais.

## 3.5 Exemplo de Aplicação da Técnica

Para exemplificar a aplicação da técnica de geração automática de testes com objetos *mock*, utilizaremos um sistema *ToyExample*, cujo diagrama está representado na Figura 3.6, que se trata de um sistema de notas de alunos. Dentre as muitas funcionalidades desse sistema, uma essencial seria a verificação da aprovação do aluno, que usaremos como exemplo. Neste sistema, o aluno faz parte de uma caderneta de notas de uma disciplina, onde ele tem duas notas e a nota final, além de sua freqüência. Com estes dados podemos calcular sua aprovação. Foram implementados alguns testes para validar este cálculo, com base nas seguintes regra de negócio:

1. Aluno reprovado por infrequência: freqüência inferior a 75;

2. Aluno reprovado por nota: freqüência igual ou superior a 75 e média inferior a 30;

3. Aluno aprovado por nota: freqüência igual ou superior a 75 e média igual ou superior a 70;

4. Aluno aprovado na final: freqüência igual ou superior a 75 e média final igual ou superior a 50;

5. Aluno reprovado na final: freqüência igual ou superior a 75 e média final inferior a 50.

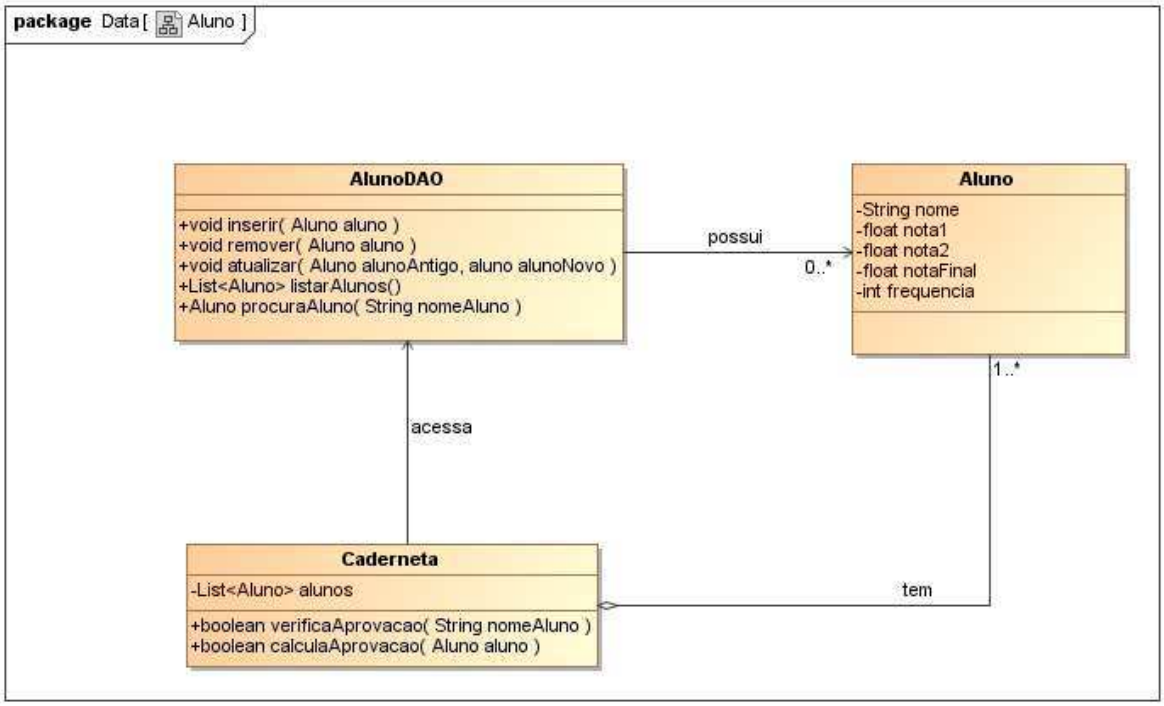



Figura 3.6: Diagrama de classes do *ToyExample*.

Como nosso objetivo é exemplificar a técnica, iremos tomar apenas um teste como exemplo, que checará a terceira regra de negócio, que verifica se um aluno foi aprovado na final. Para iniciar a aplicação da técnica, precisamos do SUT e da CUT, além do teste inicial, ilustrado na Figura 3.7, que são as entradas necessárias para o funcionamento da técnica. No contexto do *ToyExample*, essas entradas são:

- **SUT**: o sistema de notas de alunos;
- **CUT**: a classe Caderneta.java;
- **Teste**: a classe de teste TestAlunoAprovadoFinal.java.

A partir de então, podemos iniciar sua aplicação, que envolve as três fases: Análise Estática, Análise Dinâmica e Geração de Código de Teste com Objetos *Mock*.

```
1  package aluno;
2
3  import static org.junit.Assert.assertTrue;
4  import org.junit.*;
5
6  public class TestAlunoAprovadoNota {
7      private AlunoDAO alunos;
8      Aluno aluno1;
9      Caderneta caderneta;
10     @Before
11     public void setUp() throws Exception {
12         alunos = new AlunoDAO();
13         aluno1 = new Aluno("Marco", 70, 70, 0, 75);
14         caderneta = new Caderneta(alunos);
15     }
16     @Test
17     public void testAlunoAprovado(){
18         alunos.inserir(aluno1);
19         assertTrue(caderneta.verificaAprovacao(aluno1.getNome()));
20     }
```

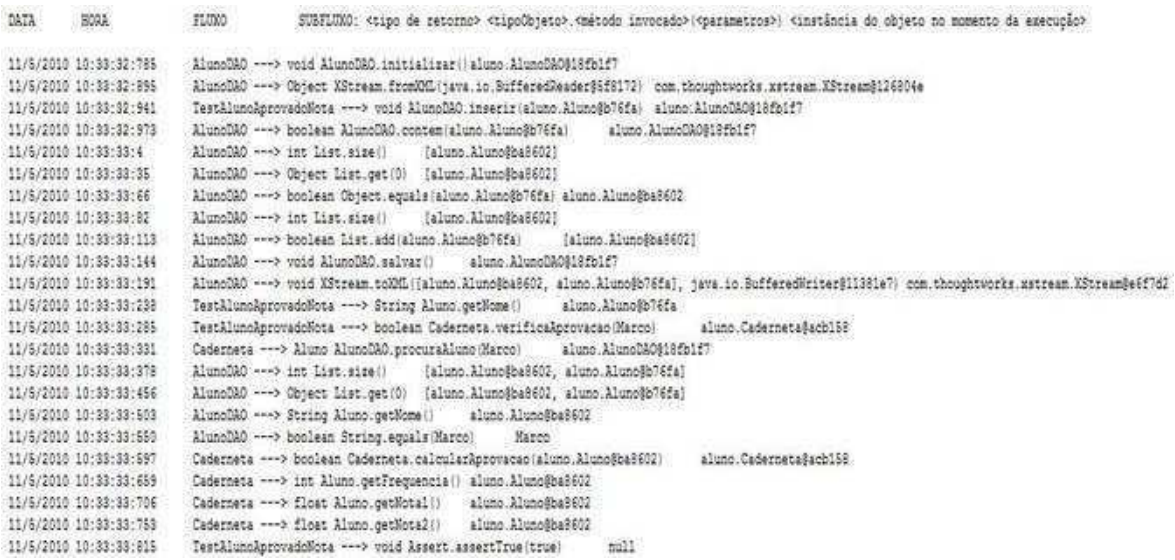
Figura 3.7: Código do teste inicial sem código *mock*.

### 3.5.1 Análise Estática

Durante esta fase, a técnica analisa o código do teste inicial, utilizando as regras explicadas na Seção 3.2 e no Pseudocódigo 3.1, em busca de colaboradores. Como resultado os colaboradores identificados foram as classes `Aluno.java` e `AlunoDAO.java`, cujas instâncias são `aluno1` e `alunos` respectivamente.

### 3.5.2 Análise Dinâmica

Ciente das entidades que interagem com a CUT, a técnica pode instrumentar o teste inicial para capturar as interações entre a CUT e os colaboradores, através da programação orientada a aspectos. Como a técnica provê um ponto de corte genérico, onde cada ponto de junção é um caso de teste, teremos apenas um ponto de junção, que é caso de teste do teste inicial, exibido na Figura 3.7, `testAlunoAprovado()`. Sendo assim, toda a execução e todas as chamadas deste caso de teste serão armazenadas em um arquivo de *log*, quando o teste instrumentado for executado. Este arquivo de *log* é exibido na Figura 3.8.



```
DATA       HORA          FLUXO       SUBFLUXO: <tipo de retorno> <tipoObjeto>.<método invocado>(<parametros>) <instância do objeto no momento da execução>

11/6/2010 10:33:32:785   AlunoDAO ---> void AlunoDAO.initializar() aluno.AlunoDAO@18fb1f7
11/6/2010 10:33:32:895   AlunoDAO ---> Object XStream.fromXML(java.io.BufferedReader@5f8172)  com.thoughtworks.xstream.XStream@126804e
11/6/2010 10:33:32:941   TestAlunoAprovadoNota ---> void AlunoDAO.inserir(aluno.Aluno@b76fa)  aluno:AlunoDAO@18fb1f7
11/6/2010 10:33:32:973   AlunoDAO ---> boolean AlunoDAO.contem(aluno.Aluno@b76fa)      aluno.AlunoDAO@18fb1f7
11/6/2010 10:33:33:4     AlunoDAO ---> int List.size()      [aluno.Aluno@ba8602]
11/6/2010 10:33:33:35    AlunoDAO ---> Object List.get(0)   [aluno.Aluno@ba8602]
11/6/2010 10:33:33:66    AlunoDAO ---> boolean Object.equals(aluno.Aluno@b76fa) aluno.Aluno@ba8602
11/6/2010 10:33:33:82    AlunoDAO ---> int List.size()      [aluno.Aluno@ba8602]
11/6/2010 10:33:33:113   AlunoDAO ---> boolean List.add(aluno.Aluno@b76fa)      [aluno.Aluno@ba8602]
11/6/2010 10:33:33:144   AlunoDAO ---> void AlunoDAO.salvar()      aluno.AlunoDAO@18fb1f7
11/6/2010 10:33:33:191   AlunoDAO ---> void XStream.toXML([aluno.Aluno@ba8602, aluno.Aluno@b76fa], java.io.BufferedWriter@11381e7) com.thoughtworks.xstream.XStream@e6f7d2
11/6/2010 10:33:33:238   TestAlunoAprovadoNota ---> String Aluno.getNome()      aluno.Aluno@b76fa
11/6/2010 10:33:33:285   TestAlunoAprovadoNota ---> boolean Caderneta.verificaAprovacao(Marco)      aluno.Caderneta@acb158
11/6/2010 10:33:33:331   Caderneta ---> Aluno AlunoDAO.procuraAluno(Marco)      aluno.AlunoDAO@18fb1f7
11/6/2010 10:33:33:378   AlunoDAO ---> int List.size()      [aluno.Aluno@ba8602, aluno.Aluno@b76fa]
11/6/2010 10:33:33:456   AlunoDAO ---> Object List.get(0)   [aluno.Aluno@ba8602, aluno.Aluno@b76fa]
11/6/2010 10:33:33:503   AlunoDAO ---> String Aluno.getNome()      aluno.Aluno@ba8602
11/6/2010 10:33:33:550   AlunoDAO ---> boolean String.equals(Marco)      Marco
11/6/2010 10:33:33:597   Caderneta ---> boolean Caderneta.calcularAprovacao(aluno.Aluno@ba8602)      aluno.Caderneta@acb158
11/6/2010 10:33:33:659   Caderneta ---> int Aluno.getFrequencia() aluno.Aluno@ba8602
11/6/2010 10:33:33:706   Caderneta ---> float Aluno.getNota1()    aluno.Aluno@ba8602
11/6/2010 10:33:33:753   Caderneta ---> float Aluno.getNota2()    aluno.Aluno@ba8602
11/6/2010 10:33:33:815   TestAlunoAprovadoNota ---> void Assert.assertTrue(true)      null
```

Figura 3.8: *Log* de execução do teste inicial, que contém as interações.

A partir deste *log*, foi possível mapear as interações entre a CUT e seus colaboradores, conforme a Tabela 3.3, onde observamos que as linhas de código do teste inicial possuem subfluxos de execução, que representam as interações entre a CUT e seus colaboradores.

| Linha de código do teste inicial | Subfluxo correspondente a linha |
|---|---|
| 18 | boolean AlunoDAO.contem(aluno.Aluno@b76fa) aluno.AlunoDAO@18fb1f7 |
| 19 | Aluno AlunoDAO.procuraAluno(Marco) aluno.AlunoDAO@18fb1f7 |
| | String Aluno.getNome()aluno.Aluno@ba8602 |
| | int Aluno.getFrequencia() aluno.Aluno@ba8602 |
| | float Aluno.getNota1()aluno.Aluno@ba8602 |
| | float Aluno.getNota2()aluno.Aluno@ba8602 |

Tabela 3.3: Mapa de fluxos de execução para cada colaborador.

### 3.5.3 Geração de Teste com Objetos *Mock*

A partir do arquivo de *log* que contém as interações entre a CUT e os colaboradores, a técnica utiliza o algoritmo de geração de código para transformar as interações contidas no *log* em código de teste com objetos *mock*. Mas, para isto, ela forma o mapa de subfluxos, já exibido na seção anterior, que agora é concretizado na Tabela 3.3, que contém o subfluxo correspondente ao *log* exibido na Figura 3.8.

A partir desse Tabela 3.3, podemos mapear às interações correspondentes as linhas de código do teste inicial, Figura 3.7:

- as linhas 19, 20, 22 e 23 do teste final, Figura 3.9, correspondem às interações da linha 18 do teste inicial, Figura 3.7;
- as linhas 24, 25, 26, 27, 28, 29 e 31 do teste final, Figura 3.9, correspondem às interações da linha 19 do teste inicial, Figura 3.7;
- as linhas 3, 4 e 5 correspondem às importações de pacotes necessárias para o código *mock*;
- as linhas 13 e 14 correspondem aos construtores dos objetos *mock* gerados a partir das linhas 12 e 13 do teste inicial, Figura 3.7;

Em resumo, todas as linhas selecionadas de amarelo, na Figura 3.9, correspondem as linhas de código geradas automaticamente pela técnica.

```
1  package aluno;
2  import static org.junit.Assert.assertTrue;
3  import org.easymock.*;
4  import static org.easymock.classextension.EasyMock.* ;
5  import org.junit.*;
6
7  public class TestAlunoAprovadoNotaMock {
8      private AlunoDAO alunos;
9      Aluno aluno1;
10     Caderneta caderneta;
11     @Before
12     public void setUp() throws Exception {
13         alunos = createMock(AlunoDAO.class);
14         aluno1 = createMock(Aluno.class);
15         caderneta = new Caderneta(alunos);
16     }
17     @Test
18     public void testAlunoAprovado(){
19         EasyMock.expect(alunos.contem(aluno1)).andReturn(true);
20         EasyMock.replay();
21         alunos.inserir(aluno1);
22         EasyMock.verify();
23         EasyMock.reset();
24         EasyMock.expect(alunos.procuraAluno("Marco")).andReturn(aluno1);
25         EasyMock.expect(aluno1.getNome()).andReturn("Marco");
26         EasyMock.expect(aluno1.getFrequencia()).andReturn(75);
27         EasyMock.expect(aluno1.getNota1()).andReturn(Float.valueOf(70));
28         EasyMock.expect(aluno1.getNota2()).andReturn(Float.valueOf(70));
29         EasyMock.replay();
30         assertTrue(caderneta.verificaAprovacao(aluno1.getNome()));
31         EasyMock.verify();
32     }
```

Figura 3.9: Código do teste final com código *mock* gerado automaticamente.

## 3.6 Detalhes de Implementação

Com o propósito de verificar a viabilidade da técnica proposta neste trabalho, desenvolvemos uma ferramenta protótipo, que implementa a técnica de geração automática de testes com objetos *mock*, a qual chamamos AutoMock. O AutoMock é uma extensão do protótipo apresentado por Souto em [33; 60] e foi desenvolvido no contexto do projeto AutoTest no LSD [7], onde utilizamos o OurProcess como processo de desenvolvimento, processo interno do mesmo laboratório.

A solução proposta é específica para sistemas orientados a objetos, mas o AutoMock é implementando para geração de testes com objetos *mock* para sistemas desenvolvidos em Java. Desta forma, consideramos as características próprias dessa linguagem para fazer a análise estática, a análise dinâmica e a geração de código.

Nosso protótipo possui quatro módulos: o *Static Analyzer*, que é responsável pela primeira fase; o *Dynamic Analyzer*, que é responsável pela segunda fase; o *Generator*, responsável pela terceira fase; por fim, o módulo Util, que é responsável por dar suporte aos demais. A Figura 3.10 apresenta a visão geral dos módulos, entradas e saídas do AutoMock.

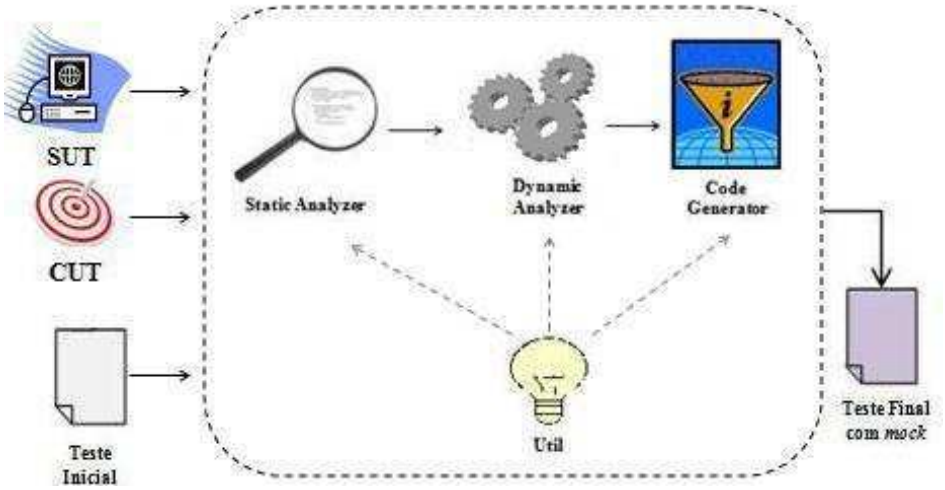



Figura 3.10: Visão geral do AutoMock.

De acordo com a Figura 3.10, o usuário é responsável por fornecer o SUT, a CUT e o teste inicial como entrada para o AutoMock, O SUT corresponde ao sistema que está sendo testado, a CUT é a classe alvo do teste (a que está sendo testada) e o teste inicial corresponde

a uma classe de teste do JUnit [6]. Essas informações são repassadas para o AutoMock em forma de parâmetros de entrada, através da ferramenta Ant [1], que automatiza o processo de execução do AutoMock. O processo de execução do AutoMock é dividido em três passos:

1. Análise estática através das entradas (SUT, CUT e teste). Como resultado, temos um arquivo contendo os colaboradores encontrados e um arquivo contendo todos os pacotes e classes necessários para combinação do aspecto, que é o *weaving*;

2. Análise Dinâmica: que é feita a partir da combinação (*weaving*) dos aspectos juntamente com o teste inicial, resultando em um teste instrumentado. A partir da execução desse teste, obtemos um arquivo de *log* de execuções;

3. Geração de código de teste com objetos *mock* para os colaboradores da CUT no teste inicial, a partir do *log* de execuções, resultando em uma nova versão do teste, que é o teste final.

O usuário interage com o AutoMock utilizando uma interface textual, através do Ant, passando os parâmetros necessários para sua execução. A partir daí, a ferramenta entra em execução, passando pelos quatro passos descritos anteriormente, resultando no teste final com código de objetos *mock*. Com todos esses passos para a execução da ferramenta, não seria possível executá-la com uma única linha de comando, por esse motivo usamos a ferramenta Ant, que automatiza tanto a execução de cada passo como a passagem de um passo para outro, através de *tasks*, que são código que especificam tarefas. Então, para cada passo temos uma *task* correspondente e, por fim, temos uma *task* maior que faz a integração entre as demais.

Para um maior detalhamento da implementação de cada módulo, vide Apêndice B.

## 3.7 Considerações Finais

Neste capítulo, apresentamos detalhes sobre a técnica proposta e suas fases: Análise Estática, Análise Dinâmica e Geração de Código de Teste com Objetos *Mock*. A primeira fase consiste em analisar o código do teste inicial em busca das entidades que colaboram com a CUT, tendo como resultado uma lista de colaboradores. A segunda fase é responsável por instrumentar o

código do teste inicial, com base nos colaboradores e na programação orientada a aspectos, e depois executá-lo para que os aspectos capturem todas as interações deste teste, resultando em um *log* de interações. Na terceira fase, a técnica mapeia, a partir do *log*, apenas as interações dos colaboradores, resultando em um mapa de subfluxos, a partir do qual um algoritmo de geração de código transforma estas interações em código de teste com objetos *mock*. Apresentamos também um exemplo de aplicação da técnica. E por fim, detalhamos como a técnica foi implementada, através de um protótipo da ferramenta, o AutoMock.

A técnica de geração automática de teste com objetos *mock* possui algumas limitações, como a gerência de configuração do protótipo em relação a sua execução, visto que temos várias fases durante a execução do mesmo, nos utilizamos da ferramenta Ant [1] para executá-la, o que não nos permite ter uma ferramenta independente. Temos ainda uma limitação em relação ao tipo de sistema que a ferramenta suporta, pois todos os experimentos foram realizados com sistemas de informação, dado que é um sistema típico para o uso de objetos *mock* nos testes, desta forma não sabemos como ela se comportará ao trabalhar com sistemas distribuídos, concorrentes, complexos, etc.

Outra limitação da técnica está no fato da qualidade do teste resultante depender da qualidade do teste inicial, ou seja, se o teste recebido como entrada para a técnica estiver mal escrito, conseqüentemente, o código final gerado também estará mal escrito, visto que a geração de código se baseia no teste inicial.

# Capítulo 4

# Avaliação Experimental

A técnica proposta apresenta duas perspectivas para sua avaliação quantitativa. A primeira está relacionada ao uso de objetos *mock* de forma manual. Já a segunda perspectiva está relacionada à utilização de objetos *mock* de forma totalmente automática, através da técnica proposta e implementada, já apresentada anteriormente. Resultando em um estudo comparativo entre a forma manual e a automática de se desenvolver objetos *mock* para um dado teste, através da observação do esforço (tempo e tamanho do código produzido) empregado em tais atividades e o nível de cobertura de interações do código resultante destas atividades.

Neste contexto, estudos experimentais são determinantes para uma boa avaliação, provendo uma disciplinada, sistemática, quantificada e controlada forma de avaliar atividades desenvolvidas tanto por seres humanos, na primeira perspectiva, quanto por sistemas, na segunda perspectiva. A técnica também é avaliada qualitativamente através de uma pesquisa se opinião.

Considerando as abordagens de avaliação existentes, vistas na Seção 2.5, e o objetivo deste trabalho, que é investigar a os ganhos advindos da técnica de automação de testes com objetos *mock* em detrimento da forma manual, através da manipulação das variáveis que determinem relações entre as duas estratégias (manual e automática), optamos pela utilização de um experimento para avaliação dessa proposta. A seguir, no tópico 5.1 detalharemos o estudo experimental realizado. Na seção 5.2 teremos uma avaliação qualitativa e, por fim, as considerações sobre a avaliação.

## 4.1 Estudo Experimental

Para se conduzir o experimento, se faz necessário um processo de experimentação que nos possibilite: analisar corretamente o objeto de estudo, controlar as variáveis e gerar conclusões significativas para o experimento. Na presente avaliação, faz-se necessário o controle das variáveis: esforço (tempo e tamanho do código produzido) e cobertura de interações, para as abordagens manual e automática de se desenvolver teste com código *mock*. Para conduzir o experimento, foram utilizadas como guia as propostas de Wohlin [65] e Travassos [36], de maneira complementar. A avaliação do experimento foi apoiada por Bisquerra [53] e Araújo [13].

O processo de experimentação compreende a realização de diferentes atividades. O número e a complexidade destas atividades podem variar de acordo com as características do estudo. A literatura propõe os seguintes passos: definição, planejamento, execução, análise e interpretação, e empacotamento. É importante lembrar que um experimento não irá proporcionar uma resposta final para uma questão, porém uma resposta específica para uma determinada configuração. Nas próximas seções, serão detalhados os passos para condução do experimento.

### 4.1.1 Definição

Utilizamos a abordagem GQM [54] para a definição do estudo, estabelecendo o objetivo global, e os objetivos de estudo e de medição, juntamente com suas questões e métricas.

O objetivo global é mensurar os ganhos em relação ao esforço (tempo e tamanho do código produzido) empregado na atividade de desenvolver testes com objetos *mock* de forma automática e a cobertura de interações do teste resultante desta abordagem, em comparação com a abordagem manual. Especificamente, nosso objetivo de estudo é:

*Comparar* o desenvolvimento de testes com objetos *mock* de forma manual e automática

*Com o propósito de* avaliar

*Com o foco* no esforço (tempo e tamanho do código produzido) e na cobertura de interações

*Sob o ponto de vista* do desenvolvedor de testes/sistemas

*No contexto de* desenvolver testes com objetos *mock* para três sistemas (*toy example*)

implementados para a realização do experimento.

Dessa forma, definimos os objetivos da medição, suas questões e respectivas métricas, em relação ao desenvolvimento de testes com objetos *mock*:

- **Objetivo 1**:Avaliar o esforço empregado na realização desta tarefa de forma manual e automática. O esforço é composto pelo tempo (medido em segundos) gasto nessas atividades e pelo tamanho do código produzido nessas atividades (medido em linhas de código);

- **Questão 1.1**:O tempo necessário para desenvolver testes com objetos *mock* de forma automática é igual ao tempo necessário para desenvolver testes com objetos *mock* de forma manual?

- **Questão 1.2**: O tamanho do código produzido durante o desenvolvimento de testes com objetos *mock* de forma automática é igual ao tamanho do código produzido no desenvolvimento de testes com objetos *mock* de forma manual?

- **Métrica 1.1**: O tempo, gasto em segundos, por cada participante para o desenvolvimento de testes com objetos *mock* de forma manual e automática, para um determinado teste. Sendo assim, definimos duas variáveis:

  **TMM** - Tempo para desenvolver Teste com Objetos *Mock* Manualmente;

  **TMA** - Tempo para desenvolver Teste com Objetos *Mock* Automaticamente.

  Desta forma, o tempo é dado por $T = TMM$, para a abordagem manual, e $T = TMA$, para a abordagem automática.

- **Métrica 1.2**: O tamanho do código produzido, medido em número de linhas de código de teste válidas (excluindo comentários e linhas em branco), para: testes com objetos *mock* desenvolvidos de forma manual (pelos participantes) e testes com objetos *mock* gerados automaticamente (utilizando a técnica proposta). Esse tamanho do código produzido é calculado a partir do teste inicial (sem objetos *mock*) para o teste final (com objetos *mock*), para cada abordagem (manual e automática). Sendo assim, definimos duas variáveis:

  **LOTCM** - Linhas de Código de Teste com Objetos *Mock* Manualmente;

**LOTCMA** - Linhas de Código de Teste com Objetos *Mock* Automaticamente.

Desta maneira, o tamanho do código produzido é dado por $Tam = LOTCM$, para a abordagem manual, e $Tam = LOTCMA$, para a abordagem automática.

- **Objetivo 2**: Avaliar a cobertura de interações do código resultante da produção de objetos *mock*, de forma manual e automática. É importante ressaltar que que para isso, se faz necessário que o teste sem objetos *mock* exista, como também a CUT e seus colaboradores, pelo menos suas interfaces;

- **Questão 2**: A cobertura de interações do código de teste com objetos *mock* gerado automaticamente é igual à cobertura do código de teste com objetos *mock* desenvolvido manualmente?

- **Métrica 2**: A cobertura de interações que o teste resultante (com objetos *mock*) deve ter em relação ao teste inicial (sem objetos *mock*), tanto utilizando a abordagem manual quanto a automática. Esta métrica tem por base um conjunto mínimo de interações entre a CUT e os colaboradores, no teste inicial, este conjunto deverá permanecer no teste final (produzido de forma manual e automática), para que se mantenha a conformidade entre o teste inicial e o final. Para medir esta cobertura, definimos as linhas de código que contém as interações entre a CUT e os colaboradores durante o teste inicial (que são as interações esperadas) e comparamos com o as interações do teste resultante (com código *mock*, de forma manual e automática), a fim de verificar se este contém o conjunto mínimo de interações definidas. Desta forma, tivemos que definir três variáveis:

  **IE** - Número de Interações Esperadas;

  **IR** - Número de Interações Reais (para testes com objetos *mock* manuais ou automáticos);

  **CI** - Cobertura de Interações.

  Sendo assim, o cálculo da cobertura se dá por: $CI = \frac{IR}{IE}$, onde o resultado é a percentagem de interações que o teste resultante contém em relação às interações esperadas.

### 4.1.2 Planejamento

#### 4.1.2.1 Seleção do Contexto

Para a condução do experimento, foi escolhido um contexto predominantemente não-industrial. Embora diversos autores argumentem sobre a necessidade de conduzir experimentos em ambientes realistas [57], semelhantes aos encontrados na indústria, essa abordagem demanda riscos e custos não previstos no escopo do mestrado. Dentro das dimensões apresentadas, caracterizam-se:

- **Processo**: foi utilizada à abordagem *In-vitro*, na qual o conjunto de participantes executou o experimento em um ambiente controlado;
- **Participantes**: na verdade, não pudemos especificar muitos critérios para seleção dos participantes, pois é muito difícil encontrar alguém disponível para executar um experimento voluntariamente. Desta forma, o critério de seleção foi ter alguma experiência com testes e objetos *mock*. Seguindo esse critério, todas as pessoas interessadas no experimento atendiam esses critérios, assim o experimento foi executado por alunos de graduação, pós-graduação do curso de Ciência da Computação da UFCG e alguns profissionais do mercado de trabalho de desenvolvimento e teste de software (egressos do curso);
- **Realidade**: o problema estudado se situa no contexto de três sistemas de informação implementados (vide documentação no Apêndice A) para a realização do experimento (*toy example*). O primeiro se dá no domínio de um pedido de compra, o segundo no domínio de autenticação de usuários e o terceiro, no domínio de alunos de uma disciplina, a descrição detalhada de cada sistema se encontra no Apêndice A. Cada sistema já possui testes de unidade, então o objetivo é produzir código *mock* para estes testes de forma manual (pelos participantes) e automática (através do protótipo que implementa a técnica já apresentada).
- **Generalidade**: o experimento é específico e com validade apenas no escopo do presente estudo.

#### 4.1.2.2 Formulação das Hipóteses

Conforme já discutido em capítulos anteriores, a atividade de desenvolver e manter código *mock* é uma tarefa repetitiva e custosa mesmo com o auxílio de APIs para a escrita deles [33; 42; 45], partindo desta premissa foram definidas três hipóteses informalmente: Ta

1. Indica-se que o esforço empregado para se desenvolver testes com objetos *mock* de forma automática é igual ao esforço empregado para desenvolver testes com objetos *mock* de forma manual. Esta hipótese mapeia as Questões 1.1 e 1.2, definidas anteriormente;

2. Sugere-se que a cobertura de interações do teste com objetos *mock* gerado automaticamente é igual à cobertura de interações de teste com objetos *mock* desenvolvido manualmente. Esta hipótese mapeia a Questão 2, definida anteriormente;

Com base na definição informal, viabiliza-se a formalização das hipóteses e a definição de suas medidas para avaliação.

1. **Hipótese Nula,** $H_0$: O tempo necessário para se desenvolver testes com objetos *mock* de forma automática é igual ao tempo necessário para se desenvolver testes com objetos *mock* de forma manual.

   **Medidas**: TMM e TMA.

   $H_0$: TMA = TMM

   **Hipótese Alternativa,** $H_1$: O tempo necessário para se desenvolver testes com objetos *mock* de forma automática é maior que o tempo necessário para se desenvolver testes com objetos *mock* de forma manual.

   $H_1$: TMA > TMM

   **Hipótese Alternativa,** $H_2$: O tempo necessário para se desenvolver testes com objetos *mock* de forma automática é menor que o tempo necessário para se desenvolver testes com objetos *mock* de forma manual.

   $H_2$: TMA < TMM

2. **Hipótese Nula,** $H_0$: O tamamho do código produzido nos testes com objetos *mock* desenvolvidos de forma automática é igual ao O tamamho do código produzido nos testes com objetos *mock* desenvolvidos de forma manual.

   Medidas: LOTCM e LOTCMA.

   $H_0$: LOTCMA = LOTCM

   **Hipótese Alternativa,** $H_1$: O tamamho do código produzido nos testes com objetos *mock* desenvolvidos de forma automática é maior que o tamamho do código produzido nos testes com objetos *mock* desenvolvidos de forma manual.

   $H_1$: LOTCMA > LOTCM

   **Hipótese Alternativa,** $H_2$: O tamamho do código produzido nos testes com objetos *mock* desenvolvidos de forma automática é menor que o tamamho do código produzido nos testes com objetos *mock* desenvolvidos de forma manual.

   $H_2$: LOTCMA < LOTCM

3. **Hipótese Nula,** $H_0$: A cobertura de interações do teste com objetos *mock* gerado automaticamente é igual à cobertura de interações do teste com objetos *mock* desenvolvido manualmente.

   Medidas: A cobertura de interações entre é medida através da razão entre as interações esperadas e as interações reais, resultando na percentagem de interações que o teste resultante (com objetos *mock* manuais ou automáticos) contém em relação às interações esperadas, onde:

   $CI_{Manual} = \frac{IR_{Manual}}{IE}$, representa a cobertura de interações de um teste com objetos *mock* manuais em relação a um teste sem objetos *mock*.

   $CI_{Automático} = \frac{IR_{Automática}}{IE}$, representa a cobertura de interações de um teste com objetos *mock* automáticos em relação a um teste sem objetos *mock*.

   $H_0$: $CI_{Automático} = CI_{Manual}$

   **Hipótese Alternativa,** $H_1$: A cobertura de interações do teste com objetos *mock* gerado automaticamente é maior que a cobertura de interações do teste com objetos *mock* desenvolvido manualmente.

$H_1$: $CI_{Automático} > CI_{Manual}$

**Hipótese Alternativa,** $H_2$: A cobertura de interações do teste com objetos *mock* gerado automaticamente é menor que a cobertura de interações do teste com objetos *mock* desenvolvido manualmente.

$H_2$: $CI_{Automático} < CI_{Manual}$

#### 4.1.2.3 Seleção das Variáveis

##### 4.1.2.3.1 Variáveis Independentes

Assumiram-se como variáveis independentes:

- Os participantes;
- Os testes para os quais serão desenvolvidos/gerados código *mock*;
- Sistemas *Toy*;

##### 4.1.2.3.2 Variáveis Dependentes

Assumiram-se como variáveis independentes:

- Tempo necessário para se desenvolver código *mock* de forma manual (TMM);
- Tempo necessário para se desenvolver código *mock* de forma automática (TMA);
- Tamanho do código produzido no desenvolvimento de código *mock* de forma manual (LOTCM);
- Tamanho do código produzido no desenvolvimento de código *mock* de forma automática (LOTCMA);
- Interações Esperadas (IE);
- Interações Reais (IR);
- Cobertura de Interações manuais ($CI_{Manual}$);
- Cobertura de Interações automáticas ($CI_{Automático}$).

#### 4.1.2.4 Seleção dos Indivíduos

A população definida para o experimento é formada por alunos do curso de graduação e pós-graduação do curso de Ciência da Computação da UFCG, e profissionais que estão no mercado de trabalho. Sendo um total de doze participantes, número máximo de pessoas que se disponibilizaram a fazer o experimento voluntariamente. Desta forma, conseguimos:

- Quatro alunos da graduação;
- Quatro alunos da pós-graduação;
- Quatro profissionais que estão no mercado de trabalho, atuando na área de desenvolvimento e teste de software.

Não foi utilizada uma amostragem probabilística para seleção dos indivíduos, mas uma amostragem não probabilística:

- **Amostragem por conveniência**: foram escolhidas as pessoas mais convenientes para o experimento, no sentido de ter alguma experiência com testes e objetos *mock*;
- **Amostragem por quota**: esta abordagem implica na escolha de indivíduos de diferentes populações, no caso, graduação, pós-graduação e mercado de trabalho. Pressupõe-se que a experiência das três populações com testes e objetos *mock* possa variar entre mais e menos experiente.

A atividade da maior parte destes participantes no experimento é desenvolver código *mock* para testes sem objetos *mock* de forma manual, utilizando o *framework* EasyMock [34], restantando um responsável pela abordagem automática, que é feita através da aplicação da técnica de geração automática de teste com objetos *mock* implementada no protótipo.

#### 4.1.2.5 Projeto do Experimento

Dentre os princípios genéricos para o projeto do experimento, caracterizamos:

- **Aleatoriedade**: a aleatoriedade foi utilizada para definir quais participantes iriam desenvolver testes com objetos *mock* manualmente;

- **Obstrução**: durante a experimentação, os participantes não possuem o mesmo nível de experiência acadêmica e profissional. Para minimizar o efeito da experiência sobre o experimento, os indivíduos foram selecionados utilizando o critério de quota e conveniência;

- **Balanceamento**: este princípio foi utilizado em nosso experimento para que cada teste fosse implementado pela mesma quantidade de participantes.

#### 4.1.2.6 Padrão para Tipo de Projeto

Para cada hipótese foram utilizadas as seguintes notações:

$\mu_{manual}$ Desenvolvimento de teste com objetos *mock* de forma manual;

$\mu_{automático}$ Desenvolvimento de teste com objetos *mock* de forma automática;

O tipo de projeto apresentado procura investigar se $\mu_{automático}$ possui o mesmo esforço (tempo e tamanho do código produzido) e cobertura de interações que $\mu_{manual}$. Para este fim, foi utilizada uma abordagem chamada **um fator com dois tratamentos**. O fator, neste experimento, consiste no que queremos observar (esforço e cobertura de interações) sob a perspectiva de dois tratamentos, que consistem na forma manual e automática de se desenvolver objetos *mock*.

A tabela 4.1 representa a forma pela qual o experimento foi executado: dez testes e doze participantes. Dos dez testes do experimento, T1 e T2 pertencem ao sistema *toy* de Pedido, T3 e T4 pertencem ao sistema *toy* de Autenticação, e T5, T6, T7, T8, T8, T9 e T10 pertencem ao sistema *toy* de notas de alunos. Esses três sistemas são relevantes no sentido de tratarem de domínios de problemas diferentes, bem como funcionalidade e estrutura de código diferentes, apesar de serem sistemas de informação, essas características impactam diretamente nos resultados do experimento, como veremos mais adiante. Dos doze participantes: dez produziram código *mock* para dois testes, manualmente; um produziu código *mock* para todos os testes, manualmente; e o ultimo produziu código *mock* para todos os testes de forma automática (utilizando o protótipo). Desta forma, temos que cada teste foi experimentado quatro vezes, por participantes diferentes, sendo três manuais e um automática.

| Teste/ Participante | $P_1$ | $P_2$ | $P_3$ | $P_4$ | $P_5$ | $P_6$ | $P_7$ | $P_8$ | $P_9$ | $P_{10}$ | $P_{11}$ | $P_{12}$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $T_1$ | X | | | | | X | | | | | X | X |
| $T_2$ | X | | | | | X | | | | | X | X |
| $T_3$ | | X | | | | | X | | | | X | X |
| $T_4$ | | X | | | | | X | | | | X | X |
| $T_5$ | | | X | | | | | X | | | X | X |
| $T_6$ | | | X | | | | | X | | | X | X |
| $T_7$ | | | | X | | | | | X | | X | X |
| $T_8$ | | | | X | | | | | X | | X | X |
| $T_9$ | | | | | X | | | | | X | X | X |
| $T_{10}$ | | | | | X | | | | | X | X | X |

Tabela 4.1: Forma como o experimento foi executado.

Para a realização dos testes das hipóteses, em um contexto de um fator e dois tratamentos, a literatura [12] sugere o teste de significância chamado Teste T para duas amostras independentes, caso seja realizado um teste paramétrico, ou Mann-Whitney, caso o teste seja não-paramétrico. A definição do teste a ser aplicado ocorrerá após a análise da normalidade (através do teste de Shapiro-Wilk) e variância dos dados obtidos pela execução do experimento (Teste de Levene).

#### 4.1.2.7 Instrumentação

No presente experimento, serão utilizados:

- **Objetos**: A modelagem UML de um diagrama de classes de um projeto de um sistema de informação, que contém os testes para os quais os participantes tiveram que desenvolver código *mock*, além do código do sistema. Uma descrição textual sobre o domínio do problema tratado neste sistema. Para o desenvolvimento de código *mock*, foi fornecido o apoio ferramental necessário: JUnit [6] para executar e desenvolver código de teste, EasyMock [34] para desenvolver o código *mock*, JDK 6.0 [5] para executar e desenvolver código Java; O sistema, juntamente com essas ferramentas, já se encontrava configurado na IDE Eclipse [26], sem precisar de nenhum esforço do participante neste sentido.

- **Guias**: Foi fornecido um documento sobre como proceder durante o experimento;

- **Métricas**: Os dados foram coletados de acordo com cada métrica observada tempo, tamanho do código produzido e cobertura de interações).

#### 4.1.2.8 Análise da Validade

- **Validade Interna**

  Para esta avaliação, foi adotada a interação da seleção, ou seja, os participantes que foram selecionados possuem um perfil apto aos tratamentos (manual e automático) do experimento, apresentando conhecimento prévio sobre testes e objetos *mock*. Contribuindo, desta forma, para os resultados do experimento. Além disso, para redução da influência dos fatores que não são interesse do nosso estudo e, portanto, para o aumento da validade interna do estudo, supõe-se aplicar um questionário para traçar o perfil de cada usuário, da viabilidade da solução proposta (técnica para geração automática de código *mock*) e do experimento.

- **Validade Externa**

  Como mencionado na seção de "Seleção dos Indivíduos", o estudo se propõe a utilizar alunos da graduação e pós-graduação do curso de Ciência da Computação da UFCG e profissionais que estão no mercado de trabalho, que tenham alguma experiência com testes e objetos *mock*. Assim, assume-se que eles são representativos para a população dos programadores com alguma experiência com testes e objetos *mock*, que são nosso público alvo.

- **Validade de Construção**

  Durante nosso experimento, foram avaliados os seguintes aspectos:

  **Explicação pré-operacional**: consiste na explicação operacional do experimento;

  **Adivinhação de hipóteses**: devido ao fato dos participantes serem humanos, é possível sua interação com o experimento, sugerindo novas hipóteses e exercitando a criatividade, durante a condução do experimento, mantendo sempre o foco no estudo planejado;

  **Expectativas do condutor do experimento**: ao se conduzir um experimento, o responsável pode exercer influências sobre as variáveis envolvidas e sobre o material

elaborado. Durante a presente proposta, todo o material utilizado será previamente avaliado por outro responsável.

- **Validade da Conclusão**

  Foram avaliadas as seguintes perspectivas:

  **Manipulação dos dados**: como os dados resultantes do experimento serão manipulados pelo pesquisador, é possível que os mesmos sofram algumas variações, tal como o coeficiente de significância para validação dos resultados;

  **Confiabilidade das medidas**: Em nossa proposta, as medidas foram objetivamente definidas;

  **Confiabilidade na implementação dos tratamentos**: consiste no risco em que diferentes participantes possam implementar de forma distinta os processos estabelecidos pelo experimento. Este risco será evitado em nosso estudo, visto que queremos que os participantes produzam código *mock* conforme as interações contidas no teste.

  **Configurações do ambiente do experimento**: consiste nas interferências externas do ambiente que podem influenciar os resultados durante a execução do experimento. O experimento será executado em um laboratório isolado, onde será proibida a interação externa como celulares, saídas, etc.;

  **Heterogeneidade aleatória dos participantes**: a escolha de diferentes participantes com diferentes experiências pode exercer um risco na variação dos resultados.

### 4.1.3 Execução

#### 4.1.3.1 Preparação

Para preparar a execução do experimento, atentou-se para:

- **Consenso com o experimento**: de acordo com Wohlin [65], se os participantes não concordam com os objetivos da pesquisa ou não tem conhecimento sobre o experimento, corre-se o risco de que sua participação não ocorra em encontro aos

objetivos. Durante a experimentação, a preparação dos participantes forneceu o embasamento necessário sobre o experimento, clarificando quais os objetivos e metas almejadas;

- **Resultados sensitivos**: é possível que o resultado obtido pelo experimento se influencie por questões pessoais, como a sensibilidade dos participantes por estarem sendo avaliados. Foi adotada uma postura de anonimato dos participantes em toda a descrição da experimentação.

Com relação à instrumentação, todas as variáveis e os recursos foram criteriosamente estabelecidos antes da execução do experimento. Foi apresentado um documento contextualizando os objetivos, a técnica, a motivação e o procedimento técnico para condução do experimento. Outro critério a ser considerado é a questão do anonimato, onde os nomes dos participantes não serão registrados.

#### 4.1.3.2 Execução

A presente proposta estruturou o experimento em um curto período de tempo (duas horas), no qual o responsável pela condução do experimento está presente em todos os detalhes da execução, ficando à disposição dos participantes para o esclarecimento das dúvidas que surgirem ao longo do processo. Ao final da execução dos experimentos, os participantes produziram os dados, cuja coleta foi responsabilidade do pesquisador.

Observamos que, devido à simplicidade envolvida no tratamento automático, ou seja, na execução do protótipo para se gerar código *mock* automaticamente, foi necessário apenas um participante para realizar esta tarefa. E, assim como no tratamento manual, a coleta dos dados foi responsabilidade do pesquisador. Em relação ao restante dos participantes, optamos por explicar e demonstrar apenas o uso da técnica, os quais puderam avaliar os ganhos advindos dela, através de uma análise qualitativa, que será descrita posteriormente.

Todo processo de execução do experimento, incluindo as observações e coleta de dados, durou em média dois meses, o que poderia ter ocorrido em menos tempo, caso o experimento não envolvesse aspectos humanos.

### 4.1.4 Análise e Interpretação

A primeira análise apresentada diz respeito à classificação das escalas das variáveis definidas no experimento, apresentada na Tabela 4.2. Com esta classificação, é possível determinar as operações que podem ser aplicadas sobre as variáveis.

| Variáveis | Nome | Escala |
|---|---|---|
| Dependentes | Esforço (tempo e tamanho do código produzido) | Razão |
| | Cobertura de Interações | Razão |
| Independentes | Participantes | Nominal |
| | Testes | Nominal |
| | Sistemas *Toy* | Nominal |

Tabela 4.2: Escala das variáveis.

#### 4.1.4.1 Análise Tabular e Gráfica

Para um melhor entendimento sobre os dados coletados, iremos apresentar uma visualização tabular e gráfica relativa a cada objeto de estudo: esforço (tempo e tamanho do código produzido) e cobertura de interações. Os dados referentes a cada um estão representados nas Tabelas 4.3, 4.4 e 4.5 e nas Figuras 4.1, 4.2 e 4.3. Para uma melhor interpretação desses dados, destacamos que os testes T1 e T2 se referem ao Sistema de Pedido *toy*, os testes T3 e T4 correspondem ao Sistema de Autenticação *toy* e, por fim, os testes T5, T6, T7, T8, T9 e T10 exercitam o Sistema de Notas de Alunos *toy*, todos esses três sistemas estão descitos em detalhes no Apêndice A.

##### 4.1.4.1.1 Esforço

###### 4.1.4.1.1.1 Tempo

Durante a execução do experimento, foram observados os tempos gastos para se desenvolver código *mock* de forma manual e automática. Para a abordagem manual, utilizamos à variável TMM e para a automática, utilizamos à variável TMA. A Tabela 4.3 contém os

dados coletados para cada uma dessas variáveis.

| Teste | TMM | | | TMA |
|---|---|---|---|---|
| T1 | 840s | 1410s | 1980s | 189s |
| T2 | 600s | 600s | 360s | 257s |
| T3 | 780s | 540s | 300s | 52s |
| T4 | 2220s | 1200s | 180s | 24s |
| T5 | 780s | 600s | 420s | 26s |
| T6 | 180s | 188s | 180s | 16s |
| T7 | 2940s | 1560s | 180s | 16s |
| T8 | 660s | 390s | 120s | 16s |
| T9 | 1800s | 1440s | 120s | 17s |
| T10 | 1560s | 760s | 60s | 15s |

Tabela 4.3: Tabulação dos dados obtidos após a execução do experimento para as variáveis TMM e TMA.

Como podemos observar, o experimento foi executado quatro vezes para cada teste, com participantes diferentes: três vezes, utilizando a abordagem manual; e uma vez, utilizando a abordagem automática. A partir destas observações, podemos visualizar graficamente, na Figura 4.1, a diferença entre cada abordagem e perceber o ganho de tempo em utilizar a abordagem automática em detrimento da manual.

Além das diferenças entre cada abordagem, podemos interpretar os dados de forma isolada. A partir dos dados da Tabela 4.3 observamos que:

- Comparando T1 e T2 em cada coluna, houve uma diferença significativa de tempo entre essas excuções. Isso se deve ao fato de que, apesar desses testes pertencerem ao mesmo sistema, eles são bastante diferentes do ponto de vista funcional e estrutural;

- Podemos ainda, fazer a mesma interpretação para T3 e T4. Onde os tempos de cada execução foram bem diferentes também devido a estrutura e funcionalidade de cada um, apesar de pertencerem ao mesmo sistema;

- Em relação as execuções dos testes T5, T6, T7, T8, T9 e T10, obtivemos certa variância entre os tempos observados, embora os testes pertencessem ao mesmo sistema, com exceção da ultima coluna de TMM, cujos testes foram produzidos por um único participante, daí podemos perceber que o tempo de cada execução vai diminuindo, devido ao conhecimento acumulado pelo participante em cada teste, fazendo com que

sua produtividade aumente. Outro comportamento parecido com esse, se dá na coluna TMA, onde o tempo de cada execução tende a diminuir, também devido ao conhecimendo do sistema por parte do AutoMock, na geração automática.

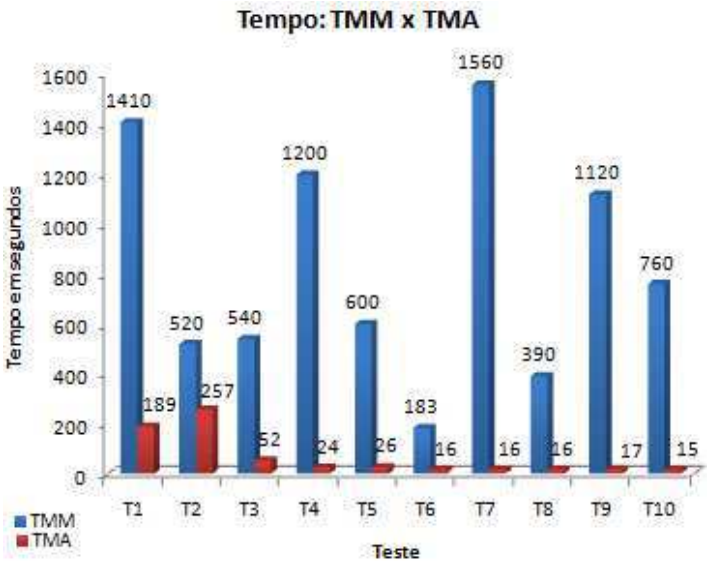



Figura 4.1: Gráfico de barras comparativo entre as médias de TMM e TMA.

#### 4.1.4.1.1.2 Tamanho do Código Produzido

Durante a execução do experimento, foi observado o tamanho do código produzido (medido em linhas de código) no desenvolvimento de código *mock* de forma manual e automática. Esse tamanho de código produzido foi obtido a partir do teste inicial (sem objetos *mock*, onde LOTC é o número de linhas deste teste), a partir do qual o participante produziu código *mock*.

Para a abordagem manual, temos a variável LOTCM, que representa as linhas de código do teste resultante, já com o código *mock*. Para a abordagem automática, temos a variável LOTCMA, que representa as linhas de código do teste resultante, já com o código *mock* gerado automaticamente. A Tabela 4.4 contém os dados coletados para cada uma dessas variáveis.

| Teste | LOTCM | | | LOTCMA |
|---|---|---|---|---|
| T1 | 57 | 120 | 84 | 54 |
| T2 | 54 | 117 | 78 | 50 |
| T3 | 18 | 28 | 38 | 26 |
| T4 | 39 | 35 | 31 | 29 |
| T5 | 22 | 31 | 40 | 21 |
| T6 | 23 | 27 | 31 | 21 |
| T7 | 34 | 36 | 37 | 21 |
| T8 | 33 | 35 | 38 | 21 |
| T9 | 38 | 31 | 36 | 21 |
| T10 | 33 | 30 | 34 | 20 |

Tabela 4.4: Tabulação dos dados obtidos após a execução do experimento para as variáveis LOTCM e LOTCMA.

Como podemos observar, o experimento foi executado quatro vezes para cada teste, com participantes diferentes: três vezes, utilizando a abordagem manual; e uma vez, utilizando a abordagem automática. A partir destas observações, podemos visualizar graficamente, na Figura 4.2 a seguir, a diferença entre cada abordagem e perceber a redução do tamanho do código produzido ao utilizar a abordagem automática em detrimento da manual.

Além das diferenças entre cada abordagem, podemos interpretar os dados de forma isolada. A partir dos dados da Tabela 4.4 observamos que:

- Comparando T1 e T2 em cada coluna, não houve uma diferença significativa entre o tamanho do código produzido nessas excuções. Isso se deve ao fato de que, apesar desses testes pertencerem ao mesmo sistema e serem diferentes do ponto de vista funcional e estrutural, a produção de código *mock* com o suporte de um *framework* (EasyMock) prover a reutilização dos objetos *mock* no mesmo teste, evitando a poluíção do código;

- Podemos ainda, fazer a mesma interpretação para T3 e T4. Onde valores de cada execução foram semelhantes também devido a reutilização dos objetos *mock* no mesmo teste, por conta do suporte oferecido pelo *framework*. Segue a mesma interpretação para os valores de T5 e T6, T7 e T8, T9 e T1. Esse efeito é destacado mais ainda na coluna da variável LOTCMA, onde temos o tamanho do código produzido

praticamente igual para os seis ultimos testes, o que acontece por conta da reutilização dos objetos *mock* pela automatização sistemática realizada pelo AutoMock, que leva em consideração o conhecimento acumulado da ferramenta.

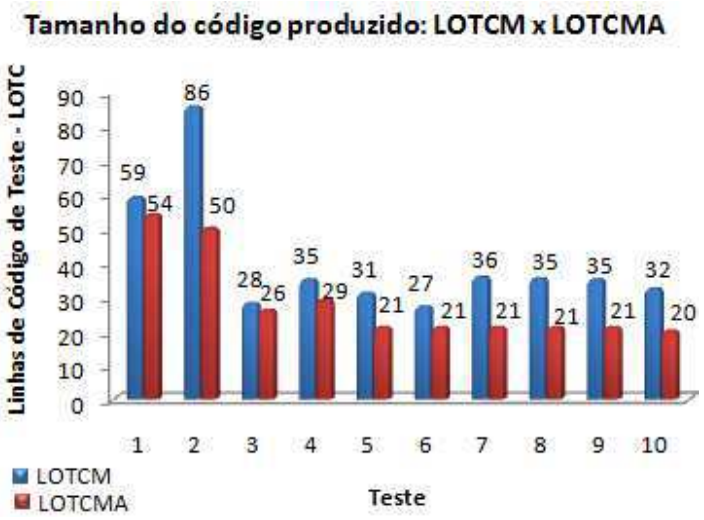



Figura 4.2: Gráfico de barras comparativo entre as médias de LOTCM e LOTCMA

#### 4.1.4.1.2 Cobertura de Interações

Durante a execução do experimento, foi observada a cobertura de interações que o teste resultante (com objetos *mock*) deve ter em relação ao teste inicial (sem objetos *mock*), tanto utilizando a abordagem manual quanto a automática. O cálculo da cobertura de interações tem por base um conjunto mínimo de interações que deverá permanecer no teste resultante (com objetos *mock*) para que se mantenha a conformidade entre o teste inicial e o final. Este conjunto mínimo é representado pelas interações esperadas IE, e as interações do teste resultante correspondem às interações reais IR. Desta forma, o cálculo da cobertura de interações se dar por: $CI = \frac{IR}{IE}$, onde o resultado é a percentagem de interações que o teste resultante contém em relação às interações esperadas. A tabela 4.5 contém os dados para a cobertura de interações resultante da abordagem manual ($CI_{Manual}$) quanto da automática ($CI_{Automático}$), onde podemos observar que a cobertura das interações são sempre mantidas com a abordagem automática, o que nem sempre acontece com a manual.

| Teste | $CI_{Manual}$ | | $CI_{Automático}$ | |
|---|---|---|---|---|
| T1 | 27% | 100% | 85% | 100% |
| T2 | 26% | 100% | 87% | 100% |
| T3 | 25% | 75% | 25% | 100% |
| T4 | 20% | 80% | 40% | 100% |
| T5 | 14% | 57% | 86% | 100% |
| T6 | 50% | 50% | 75% | 100% |
| T7 | 83% | 100% | 83% | 100% |
| T8 | 86% | 100% | 86% | 100% |
| T9 | 50% | 83% | 83% | 100% |
| T10 | 100% | 100% | 100% | 100% |

Tabela 4.5: Tabulação dos dados referentes à $CI_{Manual}$ e $CI_{Automático}$.

A partir da execução do experimento, calculamos a cobertura de interação dos testes resultantes, tanto da abordagem manual quanto da automática. Na Figura 4.3, observamos que a cobertura das interações são sempre mantidas com a abordagem automática, o que não acontece com a manual.

Além das diferenças entre cada abordagem, podemos interpretar os dados de forma isolada. A partir dos dados da Tabela 4.5 observamos que:

- Comparando T1 e T2 em cada coluna, como também T3 e T4, T5 e T6, T7 e T8, T9 e T10, não houve uma diferença significativa entre a percentagem de cobertura em cada execução. Em alguns casos houve a mesma percentagem de cobertura, sempre comparado de dois em dois, dado que cada participante produziu código *mock* para cada dois testes, exceto na ultima coluna de CIM (realizados pelo mesmo participante) e para os valores de CIA (abordagem automática). Sendo assim, podemos concluir que a cobertura de interações quase não variou em cada duas observações, e na maioria dos casos ele aumentou no segundo teste produzido pelo mesmo participante, demonstrando que o conhecimento do sistema pelo participante, o torna mais apto a discernir quais as interações deveriam ser *mockadas*.

- O comportamento observado nos valores de CIA, para todos os testes, se dar pelo fato das interações *mockadas* automaticamente seguirem o mapa de interações gerado pelo AutoMock, Tabela 3.2, que resulta na cobertura total das interações esperadas.

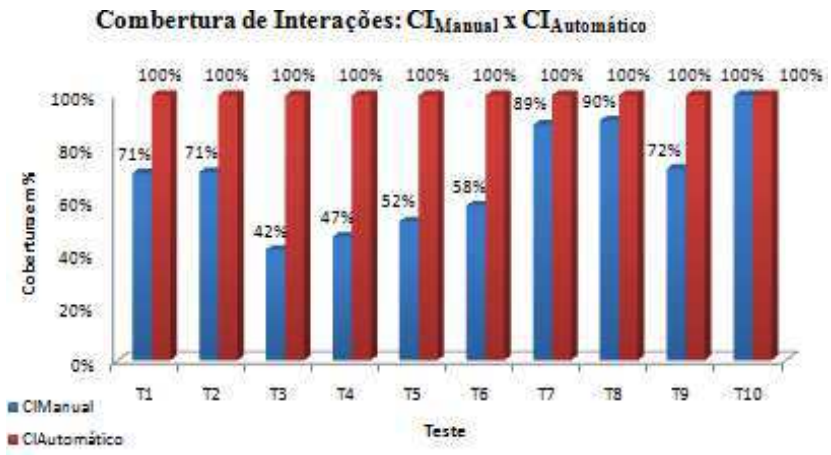



Figura 4.3: Gráfico de barras comparativo entre as médias de $CI_{Automático}$ e $CI_{Manual}$

#### 4.1.4.2 Estatística Descritiva

Conforme apresentado, as variáveis dependentes estão caracterizadas na escala razão, o que permite o calculo da normalidade e homocedasticidade, necessária para definir o tipo de teste de hipótese (paramétrico ou não-paramétrico). Conforme definido no projeto do experimento, o padrão para tipo de teste previsto é o Teste T para duas amostras independentes, caso o teste empregado seja paramétrico, ou Mann-Whitney, caso seja não paramétrico.

Nossa avaliação será executada para as hipóteses: esforço (tempo e tamanho do código produzido) e cobertura de interações. Para cada hipótese, os dados serão caracterizados, visualizando tendências centrais e dispersões. Posteriormente, sugere-se a análise de dados anormais ou incertos (*outliers*). Em seguida, realizaremos a avaliação estatística verificando o intervalo de confiança das médias para cada hipótese e, dependendo do resultado, realizaremos o teste de hipótese, considerando um nível de significância (*p-value*) de 5%. O *p-value* compreende o menor nível de significância com que se pode rejeitar a hipótese nula. Por fim, analisaremos os ganhos derivados da abordagem automática em relação à manual, para cada hipótese.

##### 4.1.4.2.1 Primeira Hipótese: Tempo

Uma análise inicial é eficiente para avaliar o comportamento das amostras. Para isto, utilizaremos o gráfico de dispersão *boxplot*, apresentado na Figura 4.4, para identificação dos *outliers*. Todas as análises estatísticas apresentadas neste experimento foram feitas utilizando

o pacote estatístico SPSS [9].

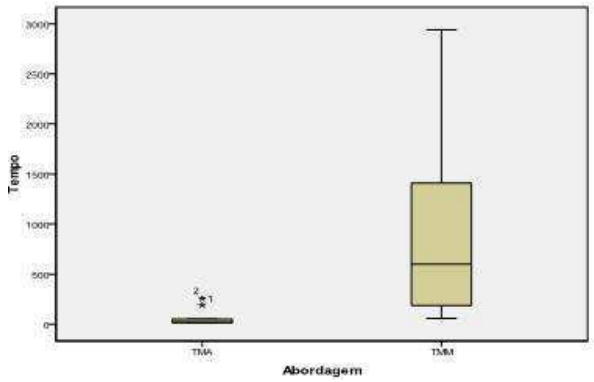



Figura 4.4: Gráfico de dispersão para a variável Tempo.

É importante verificar as origens de cada outlier, pois eles podem ser efetivamente observações válidas e que deveriam ser consideradas no universo de estudo. Conforme apresentado na Figura 4.4, a variável tempo possui dois *outliers* para as execuções *1 e *2, e um possível motivo para esta ocorrência é o fato de que essas execuções correspondem aos testes maiores e mais complexos que temos no experimento. Sendo assim, estas observações são válidas e devem ser consideradas no universo de estudo, ao invés de serem eliminadas.

Considerando todas as observações para a variável tempo, queremos verificar se a abordagem automática é mais rápida que a manual, através da observação do tempo médio de desenvolvimento de código *mock* com o uso de cada estratégia (TMM - para a estratégia manual e TMA - para a estratégia automática). Nesta análise, constatamos que, para um nível de confiança de 95%, TMA tem média de 62,8 e TMM tem média de 828,27.

Segundo Raj [37], quando os intervalos de confiança não se sobrepõem, é possível extrair relações de "maior que"òu "menor que", de acordo com as médias calculadas. Desta forma, a partir da observação dos intervalos de confiança, no Gráfico 4.5, e das médias calculadas, podemos perceber que estes intervalos não se sobrepõem. Conseqüentemente, podemos dizer que há evidências de que a abordagem automática (TMA) é mais rápida que a manual (TMM).

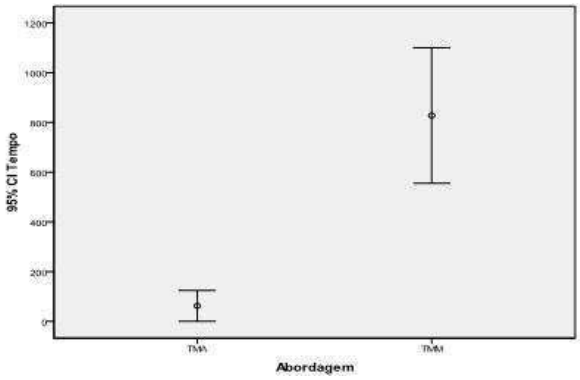



Figura 4.5: Gráfico dos intervalos de confiança de TMA e TMM.

Já que há evidências de que a abordagem automática (TMA) é mais rápida que a manual (TMM), fizemos a normalização dos dados em termos de TMA, a qual expressa o ganho de tempo em se utilizar a abordagem automática. Este ganho é medido para cada observação manual em relação à automática na Tabela 4.6, onde utilizamos a fórmula: Ganho de tempo $= \frac{(TMM-TMA)}{TMA}$.

Como resultado, os valores apontam um expressivo ganho de tempo com a utilização da abordagem automática, em média 3328% de ganho.

| **Teste** | **TMM** | | | **TMA** |
|---|---|---|---|---|
| $T_1$ | 840s | 1410s | 1980s | 189s |
| **Ganho** | **344%** | **646%** | **948%** | |
| $T_2$ | 600s | 600s | 360s | 257s |
| **Ganho** | **133%** | **133%** | **40%** | |
| $T_3$ | 780s | 540s | 300s | 52s |
| **Ganho** | **1400%** | **938%** | **477%** | |
| $T_4$ | 2220s | 1200s | 180s | 24s |
| **Ganho** | **9150%** | **4900%** | **650%** | |
| $T_5$ | 780s | 600s | 420s | 26s |
| **Ganho** | **2900%** | **2208%** | **1515%** | |
| $T_6$ | 180s | 188s | 180s | 16s |
| **Ganho** | **1025%** | **1075%** | **1025%** | |
| $T_7$ | 2940s | 1560s | 180s | 16s |
| **Ganho** | **18275%** | **9650%** | **1025%** | |
| $T_8$ | 660s | 390s | 120s | 16s |
| **Ganho** | **4025%** | **2338%** | **650%** | |
| $T_9$ | 1800s | 1440s | 120s | 17s |
| **Ganho** | **40488%** | **8371%** | **606%** | |
| $T_{10}$ | 1560s | 660s | 60s | 15s |
| **Ganho** | **10300%** | **4300%** | **300%** | |

Tabela 4.6: Observações do experimento para as variáveis TMM e TMA com seus respectivos ganhos.

#### 4.1.4.2.2 Segunda Hipótese: Quantidade de Código Produzido

Da mesma forma que na análise da primeira hipótese, utilizaremos o gráfico de dispersão *boxplot* para análise dos *outliers*, apresentado na Figura 4.6.

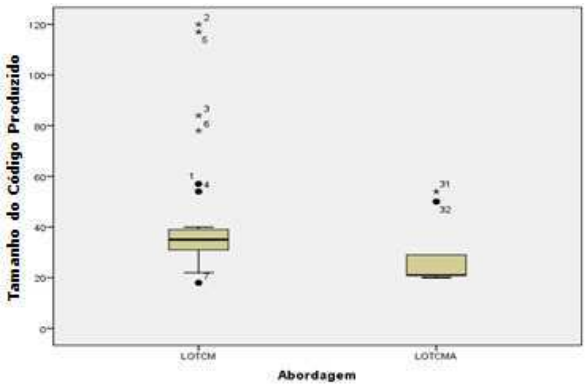



Figura 4.6: Gráfico de dispersão para a variável Quantidade de Código Produzido.

De acordo com a Figura 4.6, a variável tamanho do código produzido possui cinco *outliers*. Para a escolha de que observações serão excluídas da amostra, optou-se pela identificação numérica. A Tabela 4.7 apresenta algumas medidas estatísticas para essa variável, agrupadas por abordagem.

| Abordagem | Média | Desvio Padrão |
|---|---|---|
| LOTCM | 42,93 | 24,944 |
| LOTCMA | 28,40 | 12,791 |

Tabela 4.7: Média e desvio padrão para a variável Quantidade de Código Produzido.

Por critério de projeto, estabeleceu-se que os valores extremos que não atingirem a média com mais de dois desvios padrões, serão removidos da amostra. Neste contexto, apenas as observações *3, *6 e *31 foram eliminadas do conjunto de dados. Os demais *outliers* não foram considerados críticos para a validade das conclusões.

Eliminados os *outliers* críticos, vamos verificar se a abordagem automática requer menos código produzido que a manual, através da observação do tamanho médio do código *mock* desenvolvido com o uso de cada estratégia (LOTCM - para a estratégia manual e LOTCMA - para a estratégia automática).

Nesta análise, averiguamos que, para um nível de confiança de 95%, LOTCMA tem média de 25,56 e LOTCM tem média de 40,21.

Segundo Raj [37], quando os intervalos de confiança se sobrepõem e a média de um não está contida no intervalo de confiança do outro. Então, precisamos fazer o Teste T para extrair relações de "maior que"òu "menor que". A Gráfico 4.7 ilustra os intervalos de confiança para cada abordagem, a partir da qual podemos perceber que estes intervalos se sobrepõem, mas a média de um não está contida no intervalo de confiança do outro, então, precisamos fazer o Teste T para verificar se LOTCMA é menor que LOTCM, ou seja, se o tamanho do código produzido de forma automática é menor que na manual.

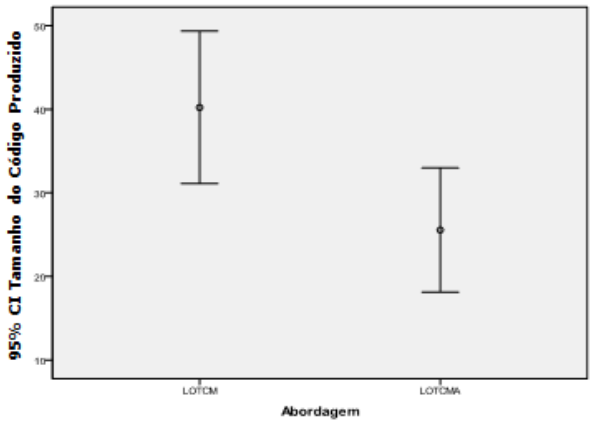



Figura 4.7: Gráfico dos intervalos de confiança de LOTCM e LOTCMA.

Para fazer o Teste T, precisamos verificar a normalidade e a homocedasticidade dos dados. Sendo assim, a próxima etapa consiste em identificar se os dados seguem uma distribuição normal. Para se avaliar a normalidade, é definida uma hipótese nula e uma hipótese alternativa, conforme:

- $H_0$: a distribuição é normal;
- $H_1$: a distribuição não é normal.

Existem duas formas para se avaliar a distribuição normal dos dados, que compreendem o Teste de Kolmogorov-Smirnov e o Teste de Shapiro-Wilk. O primeiro é utilizado para identificar a normalidade em variáveis com pelo menos 30 valores e o segundo em variáveis com menos de 50 valores. A Tabela 4.8 apresenta os testes de normalidades para a amostra utilizando o Teste de Shapiro-Wilk.

| Variável | Abordagem | Estatística | Grau de Liberdade | Significância |
|---|---|---|---|---|
| Tamanho do código produzido | LOTCM | 0,589 | 28 | 0,000 |
| | LOTCMA | 0,606 | 9 | 0,000 |

Tabela 4.8: Teste de normalidade Shapiro-Wilk para a variável Quantidade de Código Produzido.

Com base na Tabela 4.8, observa-se que a significância dos dados do teste de Shapiro-Wilk é inferior, em ambas as abordagens, ao nível de significância definido (0,05 ou 5%). Sendo assim, há indícios para rejeitar a hipótese nula e, consequentemente, não se pode aplicar um teste paramétrico para avaliação das hipóteses. Então, optou-se por aplicar o teste Mann-Whitney, para duas amostras independentes, por se tratar de uma alternativa não paramétrica para o Teste T.

O teste de Mann-Whitney para duas amostras independentes é utilizado para comprovar se as diferenças entre as médias observadas nos dois grupos independentes são estatisticamente significativas. Com base na declaração das hipóteses, sugere-se:

- $H_0$: Não há diferença entre as médias ($\mu_{LOTCM} = \mu_{LOTCMA}$)
- $H_1$: Há diferença entre as médias ($\mu_{LOTCM} \neq \mu_{LOTCMA}$)

O resultado do teste Mann-Whitney foi aplicado sobre as amostras e está apresentado na Tabela 4.9.

| Variável | U de Mann-Whitney | W de Wilcoxon | Z | Sig. Assimp. (bilateral) | Sig. Exata [2*(Sig.Unilateral)] |
|---|---|---|---|---|---|
| Tamanho do código produzido | 38,000 | 83,000 | -3,122 | 0,002 | 0,001[a] |

(a) Não corrigidos para os empates

Tabela 4.9: Teste de Mann-Whitney para a variável Tamamnho de Código Produzido.

Como o grau de significação associado (Sig. Assimpt.) é 0,002 e é menor que a significância assumida de 0,005, deve-se rejeitar $H_0$. Frente aos resultados apresentados para a variável precisão, existe diferença de média entre o tamamnho do código produzido, manual e o automático. Pela análise estatística dos dados, consegue-se recuperar duas informações:

1. A distribuição da variável tamanho do código produzido não é normal, o que implica na execução de testes não paramétricos;

2. Utilizando o teste Mann-Whitney, conseguiu-se verificar que existem diferenças entre as médias das duas amostras LOTC e LOTCM.

Utilizando o teste de Mann-Whitney, conseguiu-se apenas rejeitar a hipótese nula, porém não foi possível avaliar as hipóteses alternativas, pois não é possível extrair relações de "maior que"ċom o teste aplicado. Porém, sugere-se comparar a análise descritiva das médias da amostra conforme a Tabela 4.10.

| Abordagem | Média |
|---|---|
| LOTCM | 40,21 |
| LOTCMA | 25,56 |

Tabela 4.10: Média e desvio padrão para a variável Quantidade de Código Produzido.

Comparando as médias apresentadas, e com base nas médias das duas abordagens, observa-se que há evidências de que o tamanho do código *mock* produzido automaticamente é menor que o tamanho do mesmo tipo de código produzido manualmente. Desta forma, fizemos a normalização dos dados em termos de LOTCMA, a qual expressa a redução de código produzido com a utilização da abordagem automática. Então, medimos esta redução para cada observação manual em relação à automática na Tabela 4.11, onde usamos a fórmula: Redução do tamanho do código produzido $= \frac{(LOTCM - LOTCMA)}{LOTCMA}$.

Como resultado, os valores apontam uma redução significativa do tamanho do código produzido com a utilização da abordagem automática, em média 53%. Com a exceção de um caso onde a redução é negativa, -31%, significando que nesta observação específica não houve redução, ao contrário de todas as outras.

| Teste | | LOTCM | | LOTCMA |
|---|---|---|---|---|
| $T_1$ | 57 | 120 | 84 | 54 |
| **Redução** | **6%** | **122%** | **56%** | |
| $T_2$ | 108 | 117 | 78 | 50 |
| **Redução** | **116%** | **134%** | **56%** | |
| $T_3$ | 18 | 28 | 38 | 26 |
| **Redução** | **-31%** | **8%** | **46%** | |
| $T_4$ | 39 | 35 | 31 | 29 |
| **Redução** | **34%** | **21%** | **7%** | |
| $T_5$ | 22 | 31 | 40 | 21 |
| **Redução** | **5%** | **48%** | **90%** | |
| $T_6$ | 23 | 27 | 31 | 21 |
| **Redução** | **10%** | **29%** | **48%** | |
| $T_7$ | 34 | 36 | 37 | 21 |
| **Redução** | **62%** | **71%** | **76%** | |
| $T_8$ | 33 | 35 | 38 | 21 |
| **Redução** | **57%** | **67%** | **81%** | |
| $T_9$ | 38 | 31 | 36 | 21 |
| **Redução** | **81%** | **48%** | **71%** | |
| $T_{10}$ | 33 | 30 | 34 | 20 |
| **Redução** | **65%** | **50%** | **70%** | |

Tabela 4.11: Observações do experimento para as variáveis LOTCM e LOTCMA com suas respectivas reduções.

#### 4.1.4.2.3 Terceira Hipótese: Cobertura de Interações

Da mesma forma que na análise da primeira e da segunda hipótese, utilizaremos o gráfico de dispersão *boxplot* para análise de *outliers*, apresentado na Figura 4.8.

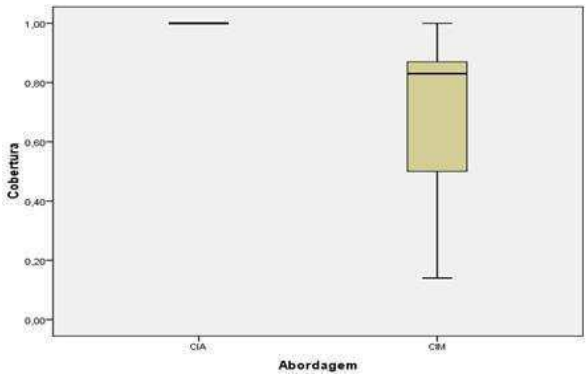



Figura 4.8: Gráfico de dispersão para a variável Cobertura de Interações.

Conforme apresentado na Figura 4.8, a variável Cobertura de Interações não possui *outliers*. Sendo assim, procedemos normalmente com todas as observações, considerando-as no universo de estudo.

Considerando todas as observações para a variável Cobertura de Interações, vamos verificar se a abordagem automática (CIA) possui uma cobertura de interações maior que a abordagem manual (CIM), através da observação da cobertura de interações média dos testes resultantes do uso de cada abordagem. Nesta análise, identificamos que, para um nível de confiança de 95%, CIA possui média de 1,0 e CIM possui média de 0,69.

O Gráfico 4.9 a seguir ilustra os intervalos de confiança para cada abordagem, a partir da qual podemos perceber que estes intervalos não se sobrepõem. Desta forma, podemos dizer que há evidências de que a abordagem automática (CIA) possui uma maior cobertura de interações que a manual (CIM), ou seja, o teste resultante da abordagem automática possui um maior tamanho de código que o teste resultante da abordagem manual, em relação às interações mínimas que o teste deve manter em relação ao original.

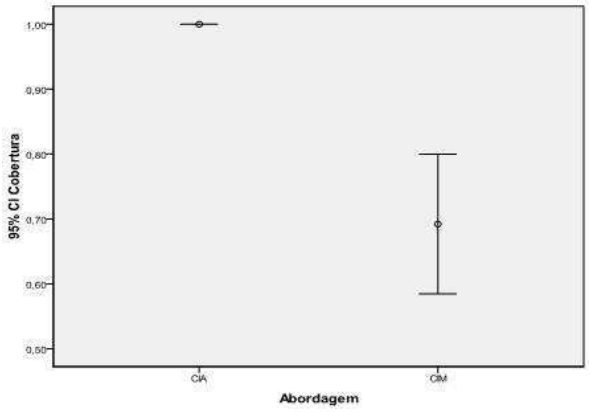



Figura 4.9: Gráfico dos intervalos de confiança de CIA e CIM.

Podemos, ainda, ilustrar o resultado com o diagrama de Venn, na Figura 4.10 a seguir, que é composto de três conjuntos:

- CI: Cobertura de Interações;
- CIM: Cobertura de Interações resultante da abordagem manual;
- CIA: Cobertura de Interações resultante da abordagem automática.

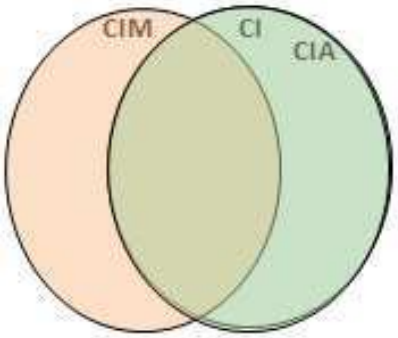



Figura 4.10: Diagrama de Venn para CI, CIM e CIA

Em CI temos todas as interações entre a CUT e seus colaboradores no teste inicial, que correspondem as interações esperadas, as quais deveriam ser *mockadas*. A partir desse diagrama, podemos observar que o teste resultante da abordagem manual (CIM) não contém todas as interações esperadas, apenas parte delas. Em contrapartida, apresenta outras interações diferentes, que exercitam outras interações não contempladas no teste inicial. Já o teste resultante da abordagem automática apresenta todas as interações do teste inicial.

De forma complementar a análise feita com os intervalos de confiança, analisamos cada observação manual em relação à automática, normalizando os dados em termos CIA, conforme a Tabela 4.12, e verificamos o ganho através da fórmula: Ganho $= \frac{(CIA-CIM)}{CIA}$.

Como resultado, os valores indicam ganho para a maioria das observações com a utilização da abordagem automática, temos algumas onde não houve ganho, mas em média obtivemos 31% de ganho.

| Teste | | CIM | | CIA |
|---|---|---|---|---|
| $T_1$ | 27% | 100% | 85% | 100% |
| **Ganho** | **73%** | **0%** | **15%** | |
| $T_2$ | 26% | 100% | 87% | 100% |
| **Ganho** | **74%** | **0%** | **13%** | |
| $T_3$ | 25% | 75% | 25% | 100% |
| **Ganho** | **75%** | **25%** | **75%** | |
| $T_4$ | 20% | 80% | 40% | 100% |
| **Ganho** | **80%** | **20%** | **60%** | |
| $T_5$ | 14% | 57% | 86% | 100% |
| **Ganho** | **86%** | **43%** | **14%** | |
| $T_6$ | 50% | 50% | 75% | 100% |
| **Ganho** | **50%** | **50%** | **25%** | |
| $T_7$ | 83% | 100% | 83% | 100% |
| **Ganho** | **17%** | **0%** | **17%** | |
| $T_8$ | 86% | 100% | 86% | 100% |
| **Ganho** | **14%** | **0%** | **14%** | |
| $T_9$ | 50% | 83% | 83% | 100% |
| **Ganho** | **50%** | **17%** | **17%** | |
| $T_{10}$ | 100% | 100% | 100% | 100% |
| **Ganho** | **0%** | **0%** | **0%** | |

Tabela 4.12: Observações do experimento para as variáveis CI e CIA com seus respectivos ganhos.

## 4.2 Avaliação Qualitativa

Conforme verificado na fundamentação teórica Seção 2.5, um experimento é responsável apenas por uma avaliação quantitativa. Para a análise qualitativa das duas abordagens de desenvolvimento de código *mock*, foi desenvolvida uma pesquisa de opinião integrada ao experimento. Ao término da execução, cada participante respondeu um questionário conforme Apêndice C.

Os objetivos almejados com a pesquisa foram: traçar o perfil de cada usuário, caracterizar o comportamento dos participantes durante o experimento em relação às atividades propostas e verificar a viabilidade da técnica desenvolvida, no ponto de vista dos participantes.

### 4.2.1 Perfil do Participante

Com relação ao perfil do participante, temos três tipos: de graduação, de pós-graduação e profissional. O perfil traçado no questionário diz respeito ao perfil em termos de conhecimento e experiência com testes, objetos *mock*, a linguagem de programação Java e tecnolo-

gias relacionadas. Neste contexto, observamos seis aspectos:

1. **Experiência na linguagem de programação Java**: 91% possui mais de um ano de experiência;

2. **Experiência com o uso da IDE Eclipse**: Todos possuem possui mais de um ano de experiência;

3. **Experiência com Testes de Unidade**: 91% possui mais de um ano de experiência;

4. **Experiência com JUnit**: 91% possui mais de um ano de experiência;

5. **Experiência com Objetos Mock**: 67% possui mais de um ano de experiência, 17% possui de seis meses a um ano de experiência e 17% possui menos de seis meses de experiência;

6. **Experiência com o framework EasyMock**: 67% possui mais de um ano de experiência, 17% possui de seis meses a um ano de experiência e 17% possui menos de seis meses de experiência.

### 4.2.2 Realização do Experimento

A respeito da condução do experimento, verificamos:

1. **O nível de dificuldade em se desenvolver código *mock* de forma manual**: 50% dos participantes achou o nível de dificuldade mediano, 25% achou fácil e o restante, difícil.

2. **As dificuldades apontadas para se desenvolver código *mock* de forma manual**:

   Configurar objetos *mock*;

   Detectar interações entre a CUT e os colaboradores;

   Entendimento da sintaxe usada pelo *framework* (EasyMock);

   Identificar o nível de isolamento do objeto no teste;

   Tarefa lenta e repetitiva. Dentre essas dificuldades, o problema de detectar as interações foi o mais citado.

3. **Necessidade de alguma mudança no sistema toy durante o experimento**: 58% dos participantes achou que foi necessário fazer alguma alteração no sistema *toy* para a realização do experimento, e 42% achou que não foi necessário fazer alterações. Alterações estruturais (passagem de parâmetros e retirar método `tearDown()` do teste).

### 4.2.3 Avaliação da Técnica para Geração Automática de Código *Mock*

A viabilidade da técnica desenvolvida foi avaliada pelos participantes, nos seguintes aspectos:

1. **Necessidade de automação da tarefa de desenvolver código *mock***: Todos os participantes concordam que o desenvolvimento de código *mock* é uma tarefa repetitiva e custosa. Os participantes também apontaram algumas possíveis causas:

   Dificuldade em configurar objetos *mock*;

   Dificuldade em detectar interações entre a CUT e os colaboradores;

   Entendimento da sintaxe usada pelo *framework* (EasyMock) O mais agravante, citado por um dos participantes, é que todas essas dificuldades tendem a aumentar dependendo tamanho e complexidade do sistema/teste.

2. **Possíveis vantagens da técnica**:

   Aumento da qualidade do teste;

   Redução do tempo de desenvolvimento de testes com código *mock*;

   Redução do tamanho do código de testes produzido.

3. **Possíveis desvantagens da técnica**:

   Geração de resíduos: código repetido e/ou inútil; Exemplo: `EasyMock.replay(); EasyMock.replay();`.

   A qualidade do código gerado depende diretamente da qualidade do teste inicial, visto que o código é gerado com base no teste inicial. Logo, se temos testes ruins ou mal escritos, isso será refletido no código gerado;

A técnica não contribui para a descoberta de problemas de *design* no código, dado que trabalha melhor com o *Test-Last* ao invés do *Test-First*, que encontra problemas de *design* logo no início do desenvolvimento.

Esta análise qualitativa reforça nossa motivação inicial para a geração automática de código *mock*, que é a redução do esforço (tempo e tamanho do código produzido) nesta atividade. Além de captar a necessidade dos desenvolvedores/testadores da área em relação a esse tipo de código. Podemos inferir então, que as possíveis causas para os resultados da análise quantitativa são as dificuldades apontadas durante a análise qualitativa.

## 4.3 Considerações Finais

Este capítulo apresentou uma avaliação quantitativa e qualitativa da proposta de geração automática de testes com objetos *mock*, utilizando um estudo experimental e uma pesquisa de opinião. O objetivo foi verificar a aplicabilidade e a relevância da técnica proposta quando comparada a atual forma de desenvolvimento de testes com objetos *mock*, que é feita de forma semi-manual, com o suporte de algum *framework* para a escrita de código *mock*.

No início da avaliação, pensamos em realizar um estudo de caso com o OurBackup Home, porém esse já tinha todos os testes necessários com código *mock*. Daí, partimos para a mais nova versão, o OurBackup Enterprise, que também não deu certo, pois a abordagem de teste não foi completada, não havendo testes para produção de código *mock*. Além disso, o projeto OurBackup encerrou e não tivemos mais suporte para utilizá-lo.

Diante da situação exposta, planejamos um experimento que se situa no contexto do desenvolvimento de testes com objetos *mock* para três sistemas de informação, conforme a documentação correspondente a cada um no Apêndice A. Esses tipos de sistema são típicos para o uso de objetos *mock* nos seus testes, onde variamos apenas as regras de negócio de cada um, de forma a se obter testes simples e complexos, como também pequenos e grandes. Se tivéssemos outros tipos de sistema (concorrente, distribuído, etc.), provavelmente teríamos resultados diferentes. Porém, esse não é o foco da solução proposta em sua concepção inicial. Desta forma, se faz necessário clarificar que os resultados obtidos através da experimentação não são generalizáveis para todos os tipos de sistemas que existem.

Os dados relativos ao esforço (tempo e tamanho do código produzido) e a cobertura

de interações foram avaliados nos seguintes passos: a análise de *outliers*, análise gráfica dos intervalos de confiança para cada abordagem (manual e automática) e dependendo do resultado, concluímos a avaliação ou passamos para a realização do teste de hipótese mais adequado, e finalmente fazemos uma análise complementar mostrando os possíveis ganhos em cada observação.

A análise realizada para o tempo compreende a análise de *outliers* e a análise gráfica dos intervalos de confiança, cujo resultado já indica que a abordagem automática (TMA) é mais rápida que a manual (TMM), em média 3328%. Da mesma forma, a análise realizada para a cobertura de interações, evidencia que a abordagem automática (CIA) possui uma maior cobertura de interações que a manual (CIM), em média 31% a mais, ou seja, o teste resultante da abordagem automática possui uma maior qualidade de código que o teste resultante da abordagem manual, em relação às interações mínimas que o teste deve manter em relação ao original.

O tamanho do código produzido apresentou um fenômeno distinto do tempo e da cobertura de interações, onde tivemos que aplicar um teste, visto que a análise de intervalos de confiança não foi suficiente. Desta forma, verificamos que o comportamento dos dados da amostra não apresentou uma distribuição normal impossibilitando a utilização do Teste T. O teste Mann-Whitney foi à escolha definida para avaliação das hipóteses. Este teste apenas informa se dois grupos independentes procedem da mesma população. O resultado foi que existem diferenças entre as médias das duas amostras LOTCM e LOTCMA. Neste contexto, não foi possível avaliar estatisticamente as hipóteses alternativas e se optou por comparar a análise descritiva das médias. Após comparar as médias, verificou-se que o tamanho do código *mock* produzido de forma automática é menor que o tamanho do código produzido no desenvolvimento do mesmo tipo de código de forma manual, resultando em uma média de 53% de redução de código produzido, que impacta diretamente o esforço.

Durante a análise qualitativa, os resultados indicaram possíveis causas para os resultados obtidos na análise quantitativa. Assim sendo, temos que as possíveis causas para o esforço ser maior na abordagem manual podem ser: a dificuldade em configurar objetos *mock*, detectar interações entre a CUT e os colaboradores, entender a sintaxe usada pelo *framework*, identificar o nível de isolamento do objeto no teste, além de que se trata de uma tarefa lenta e repetitiva. Todas essas causas variam de intensidade de acordo com o perfil de cada par-

ticipante (nível de conhecimento e experiência nos assuntos relacionados a tarefa de cada um, como abordado na primeira parte do questionário), como também com o conhecimento do mesmo a respeito do sistema, cujo código que será *mockado*. Dentre estas dificuldades, podemos apontar a dificuldade de se detectar as interações entre a CUT e os colaboradores como a principal causa para a cobertura de interações na abordagem manual ser menor que a na automática, visto que poucos participantes conseguiram cobrir todas as interações esperadas. Porém, não afeta o processo de desenvolvimento de código de teste com objetos *mock*.

Vale ainda ressaltar que a escolha dos testes iniciais afeta diretamente os resultados, dado que todo o código de teste com objetos *mock* é gerado a partir do teste inical. Inclusive, a qualidade do teste final depende diretamente da qualidade do teste inicial, pelo mesmo motivo.

Para aumentar o conhecimento sobre o esforço (tempo o tamanho do código produzido) e a cobertura de interações em diferentes contextos, definindo a validade da experimentação, sugere-se replicações do experimento em sistemas distintos e mais realistas, desenvolvendo testes com código *mock* para diferentes tipos de sistema. Sugere-se avaliar, por exemplo, o desenvolvimento de testes com objetos *mock* em sistemas mais complexos. Generalizando o experimento, possibilita-se a extração de novas informações sobre a proposta nas diferentes perspectivas dos diferentes tipos de sistema.

# Capítulo 5

# Trabalhos Relacionados

Várias técnicas têm sido propostas no sentido de automatizar as tarefas relacionadas a testes. Essas técnicas podem ser divididas em vários níveis, dependendo do tipo de teste. No nosso contexto de testes de unidade, cujo objetivo é testar cada unidade isoladamente, se faz necessário isolar a CUT, subtendendo-se que as entidades que interagem com ela funcionam como esperado. Mas, para se alcançar esse objetivo, precisamos de técnicas de provejam este isolamento, de preferência de forma automática. Neste contexto, as técnicas disponíveis hoje se apresentam em três níveis: de conceitos, de *frameworks* e de automação. Este capítulo tem como propósito apresentar algumas destas técnicas e realizar um estudo crítico e comparativo entre estas técnicas e o AutoMock.

Podemos contextualizar o AutoMock nos três níveis apresentados: para o conceitual, discorremos sobre as diferenças entre *mock*s e *stubs*; já para o nível de *frameworks*, discutimos sobre alguns *frameworks* que implementam o conceito de objetos *mock*; por fim, apresentamos algumas técnicas de automação de testes que utilizam o conceito de objetos *mock*. Posteriormente, comparamos nossa técnica nesses três níveis.

Segundo Fowler [31], um erro comum é interpretar *mocks* como sendo *stubs*. Apesar, de poderem ser utilizados no teste com o mesmo propósito, existem algumas diferenças significantes. De acordo com Meszaros [44], *stubs* provêem respostas padrões definidas estaticamente as chamadas feitas durante o teste. Já se são usados fora do contexto para o qual foram programados, eles tendem a responder de forma incorreta. Desta forma, *stubs* não são reusáveis e cada mudança no teste requer que de modifique o *stub*, de forma manual.

Por outro lado, objetos *mock* simulam o comportamento dos objetos (colaboradores) re-

ais e são dinamicamente pré-programados através de expectativas para responder de forma adequada as chamadas feitas durante o teste. Sendo assim, se são usados fora do contexto para o qual foram programados, eles explicitamente indicam uma falha no teste, oferecendo suporte para seu reuso. Através do seu mecanismo de definição e checagem de expectativas permitem: verificar quais métodos foram chamados, a quantidade de vezes, a ordem e os valores esperados, ou seja, objetos *mock* podem ser configurados para retornar valores específicos em cada chamada, podendo ser reiniciados sempre que mudar o contexto.

*Frameworks* nesta área suportam o conceito de objetos *mock* proposto por Mackinnon [42]. Porém, não automatizam por completo a escrita de código de teste com objetos *mock*. Dentre os vários *frameworks* que existem, podemos mencionar EasyMock, JMock, Mockito, DynaMock e GoogleMock [2; 3; 8; 33; 34]. Eles provêem suporte para criação e configuração de objetos *mock* dinamicamente, em comparação com *stubs*, isso evita a criação de uma classe de objetos *mock* para cada entidade que for "*mockada*", como acontecia com *stubs*. Ajudando, deste modo, a manter o código do teste de unidade isolado sem a necessidade de outras classes extra.

No entanto, precisa-se conseguir reproduzir o comportamento da entidade que será "*mockada*", o que exige um esforço da parte do testador/programador em identificar as interações da CUT com a mesma, para que se possa fazer uso do suporte provido pelo framework. Pois, apesar desse suporte em prover estruturas de definição e checagem de expectativas para o comportamento das entidades, os *frameworks* não dão suporte à identificação tanto de colaboradores quanto das interações entre a CUT e eles, que precisam ser "*mockados*".

Por outro lado, a técnica proposta neste trabalho automatiza todo esse processo de identificação de colaboradores e interações, e ainda gera o código de teste com objetos *mock* correspondente a cada colaborador e interação de forma automática, tomando por base um desses *frameworks*. O nosso protótipo de ferramenta, AutoMock, gera uma nova versão de um dado teste inicial contendo código de teste com objetos *mock* para todos os colaboradores e interações identificados pela técnica, utilizando o framework EasyMock para tradução dessas interações em código teste e *mock*.

Além dos *stubs*, objetos *mock* e *frameworks* relacionados discutidos anteriormente, existem vários outros trabalhos que compõem esta seção de trabalhos relacionados, que são as técnicas de automação de teste, mais especificamente as de geração de código de teste.

Dentre as várias técnicas de automação de teste já explicadas no Capítulo 2, existem técnicas derivadas e específicas para geração de teste unidade, umas que utilizam o conceito de objetos *mock*, outras que empregam técnica semelhante a nossa. Neste contexto discutimos sobre o trabalho de Pasternak [50], Saff [55; 56], Tillmann [62] e de Alshahwan [14].

Pasternak [50] propôs uma ferramenta, GenUTest, que gera teste de unidade (com JUnit [6]) automaticamente para um sistema (escrito em Java) que ainda não possui, seria o sistema sob teste - SUT (*System Under Test*). Sua técnica consiste em capturar e gravar as interações que ocorrem entre os objetos durante a execução do sistema para o qual serão gerados os testes. Para funcionar, o GenUTeste depende da versão completa do SUT para gerar testes de unidade para todos os objetos do sistema e para todas as interações exercitadas durante a execução do sistema, ainda constrói aspectos de *mock* (*mock aspect*, que na verdade são aspectos que intervêm na execução da CUT.

Embora a técnica empregada em GenUTest seja muito similar a nossa em termos de tecnologias utilizadas, nosso contexto e propósito são diferentes. Na nossa técnica, espera-se que o testador/programador forneça como entrada a CUT, um teste que a exercite e o SUT, e temos como saída uma nova versão do teste inicial acrescido de ódigo de teste com objetos *mock*. No AutoMock só o teste é exercitado, ao invés de todo o SUT. Desta forma, temos que as interações capturadas pelo AutoMock são mais significantes que as capturadas pelo GenUTest. Como evidência para esse fato, nos experimentos mencionados por Pasternak [50], onde ele gera testes de unidade para alguns sistemas, sua cobertura de testes sempre é menor que 20%. Outro fator relevante é que GenUTest não está disponível para que possamos realizar um estudo mais profundo. Além disso, esse *mock* aspect gerado por ele não é amplamente utilizado pelos testadores/programadores, o que dificulta a evolução do código gerado. Já o AutoMock faz uso de um framework de *mocks* conhecido, possibilitando a evolução do código gerado.

Tillmann e Schulte [62] implementaram um protótipo de ferramenta que gera objetos *mock*, utilizando uma combinação de execução concreta e simbólica de código em .NET. A execução simbólica explora todos os caminhos para cada método, analisando condições e restrições. Assim, são gerados objetos *mock* configurados com todos os comportamentos possíveis para um dado teste em um SUT. Sendo assim, a técnica gera objetos *mock* com comportamentos indesejáveis no contexto do teste em que está inserido. Já o AutoMock gera

objetos *mock* com o comportamento dos colaboradores reais do teste inicial, sem gerar outros comportamentos, além disso a ferramenta mencionada não se utiliza de nenhum framework conhecido para produzir estes objetos *mock*.

Alshahwan [14] também utiliza a execução simbólica para geração de casos de teste com objetos *mock*, sua técnica por coincidência também de chama "Automock", porém apresenta uma técnica bem diferente da nossa. Essa técnica produz casos de teste com objetos *mock* com base em pós-condições. Para isso, é necessário gerar valores de retorno para o método a ser "*mockado*", o que pode ser feito manualmente ou com o uso da técnica de Execução Simbólica. Dessa forma, recorremos ao mesmo problema de Tillmann e Schulte, a inserção de comportamentos indesejáveis no caso de teste, enquanto o AutoMock gera comportamentos precisos baseados em interações reais.

Saff [55; 56] propôs uma ferramenta para decompor um teste de sistema em testes de unidade menores com objetos *mock* para os colaboradores, utilizando a técnica de fatoração de testes (*"test factoring"*). A ferramenta recebe como entrada o SUT, o teste de sistema que se deseja fatorar e uma partição do programa que contém o código sob teste, que não é apenas uma classe, já que se trata de um teste de sistema. Tem como saída um conjunto de testes fatorados para o código sob teste. A técnica empregada consiste em instrumentar o *bytecode* do teste, incluindo todas as bibliotecas usadas pelo SUT, a fim de capturar e gravar as interações entre o código sob teste e o ambiente externo, no contexto do teste. Desta forma se consegue fatorar os testes e produzir código *mock* para o ambiente externo.

O AutoMock produz código de teste com objetos *mock* para o teste de uma classe, não para parte do sistema, não utiliza *bytecode* para instrumentação de código, esse tipo de estratégia interfere diretamente no código do SUT, enquanto a utilização da programação orientada a aspectos, utilizada pelo AutoMock, influencia minimamente no código. Em geral, apesar da técnica de Saff [55; 56] ser bastante semelhante a nossa, ela não tem o mesmo propósito, não se utiliza das mesmas técnicas nem tem o mesmo foco, pois seus resultados apontam diminuição de tempo de execução dos testes, e este benefício o AutoMock já herda só pelo fato de usa objetos *mock*, já que não está interagindo diretamente com as entidades reais do ambiente de execução.

Os objetivos do AutoMock são: reduzir esforço, composto pelo tempo de desenvolvimento de código *mock* e pelo tamanho do código produzido; verificar a qualidade do código

gerado. Com base nos resultados obtidos, analisados no Capítulo 4, há evidências de que estes objetivos foram cumpridos. Já não podemos fazer esse tipo de avaliação com as ferramentas comparadas aqui, pois elas não se encontram disponíveis para um estudo comparativo mais profundo. Entretanto, com as informações disponíveis podemos perceber que as técnicas mencionadas ou não resolvem o problema da geração de código *mock* completamente ou não resolvem de forma eficiente, enquanto o AutoMock resolve. Além disso, a técnica proposta neste trabalho é única pelo fato de ter propósitos diferentes e, conseqüentemente, uma avaliação diferente das demais citadas aqui.

# Capítulo 6

# Conclusão

As técnicas para geração de testes existentes hoje não resolvem o problema da geração de código *mock* completamente ou não resolvem de forma eficiente. Dado que às vezes geram um tipo de código em uma linguagem pouco conhecida, impossibilitando a manutenção do mesmo, e outras vezes gera apenas *templates* ou aspectos de *mock*. Ambas as situações continuam causando prejuízos de esforço (tempo e tamanho do código produzido) no desenvolvimento de testes com objetos *mock*.

Este trabalho teve como foco reduzir o esforço empenhado no desenvolvimento de testes com objetos *mock*. Então, para isso propomos uma técnica para automatizar a escrita desses objetos *mock*, gerando-os automaticamente para um determinado teste. Desta forma, validamos nosso trabalho de duas formas: verificando sua viabilidade, por meio da implementação dessa técnica, o protótipo de ferramenta AutoMock; e realizando uma avaliação da técnica, através de uma análise quantitativa e uma qualitativa.

A técnica proposta e implementada se divide em três fases: análise estática, análise dinâmica e geração de código de teste com objetos *mock*. Na primeira fase, Identificamos os objetos que colaboram com a CUT no cenário do teste inicial. Na segunda, instrumentamos e execução do teste inicial com a finalidade de capturar e gravar as interações existentes entre a CUT e os colaboradores identificados na fase anterior. Na última etapa, geramos de código de teste com objetos *mock* a partir das interações gravadas anteriormente.

A validação da técnica proposta foi realizada em duas etapas: implementação de um protótipo e realização de uma avaliação da técnica. Na primeira, avaliamos a viabilidade da técnica desenvolvendo um protótipo de ferramenta chamado AutoMock. O protótipo é

composto por quatro módulos: o *Static Analyzer*, que é responsável pela fase 1; o *Dynamic Analyzer*, que é responsável pela fase 2; o *Code Generator*, responsável pela fase 3; por fim, o módulo Util, que é responsável por dar suporte aos demais.

Na segunda etapa, realizamos uma avaliação da técnica, composta de uma análise quantitativa e qualitativa. Para se fazer a análise quantitativa, realizamos um estudo experimental para verificar as três hipóteses levantadas:

1. O *tempo* necessário para geração automática de testes com objetos *mock* é menor que o *tempo* necessário para se desenvolver testes com objetos *mock* de forma manual;

2. O *tamanho do código produzido* nos testes com objetos *mock* gerados automaticamente é menor que o *tamanho do código produzido* no desenvolvimento de testes com objetos *mock* de forma manual;

3. A *cobertura de interações* do teste com objetos *mock* gerados automaticamente é maior que, ou no mínimo igual, a *cobertura de interações* do teste com objetos *mock* desenvolvido manualmente.

A análise realizada para o *tempo* indica que a abordagem automática é mais rápida que a manual, em média 3328%. Da mesma forma, a análise realizada para a *cobertura de interações*, indica que a abordagem automática possui uma maior cobertura de interações em relação à manual, em média 31% a mais, ou seja, o teste resultante da abordagem automática possui uma maior qualidade de código que o teste resultante da abordagem manual, em relação às interações mínimas que o teste deve manter em relação ao original. Já para o *tamanho do código produzido*, não foi possível avaliar estatisticamente as hipóteses alternativas e se optou por comparar a análise descritiva das médias, onde se verificou que o tamanho do código produzido na abordagem automática é menor que o tamanho do código produzido de forma manual, resultando em uma média de 53% de redução de tamanho do código produzido.

Durante a análise qualitativa, os resultados indicaram possíveis causas para os resultados obtidos na análise quantitativa. Assim sendo, temos que as possíveis causas para o esforço ser maior na abordagem manual podem ser: a dificuldade em configurar objetos *mock*, detectar interações entre a CUT e os colaboradores, entender a sintaxe usada pelo *framework*, identificar o nível de isolamento do objeto no teste, além de que se trata de uma tarefa

lenta e repetitiva. Dentre estas dificuldades, podemos apontar a dificuldade de se detectar as interações entre a CUT e os colaboradores como a principal causa para a cobertura de interações na abordagem manual ser menor que a na automática.

## 6.1 Contribuições

Uma das principais contribuições deste trabalho consiste em disponibilizar uma técnica para geração automática de código de teste com objetos *mock*, implementada através de um protótipo de ferramenta, através de técnicas de análise estática e dinâmica de código.

O principal mérito deste trabalho é o enfoque na redução do esforço empregado no desenvolvimento de testes com objetos *mock*, que mesmo com o auxílio de *frameworks* [2; 3; 8; 33; 34] requer muito esforço para escrevê-los, reutilizá-los e mantê-los, devido às várias configurações necessárias conforme a alteração dos testes ou da CUT. Considerando o baixo custo de se gerar automaticamente o código para os objetos *mock*, os programadores não precisarão reutilizar este código, o qual poderá ser gerado novamente a toda modificação.

A técnica de geração automática de teste com objetos *mock* possui algumas limitações, como a gerência de configuração do protótipo em relação a sua execução, visto que temos várias fases durante a execução do mesmo, nos utilizamos da ferramenta Ant [1] para executá-la, o que não nos permite ter uma ferramenta independente. Temos ainda uma limitação em relação ao tipo de sistema que a ferramenta suporta, pois todos os experimentos foram realizados com sistemas de informação, desta forma não sabemos como ela se comportará ao trabalhar com sistemas distribuídos, concorrentes, complexos, etc.

Outra limitação da técnica está no fato de a qualidade do teste resultante depender da qualidade do teste inicial, ou seja, se o teste recebido como entrada para a técnica estiver mal escrito, conseqüentemente, o código final gerado também estará mal escrito, visto que a geração de código se baseia no teste inicial.

É importante clarificar que os resultados obtidos através das análises, quantitativa e qualitativa, possuem validade para o presente estudo e para sistemas de informação em geral e sistemas simples. Porém, não são generalizáveis para todos os tipos de sistemas que existem, fazendo-se necessárias mais avaliações e supostas melhorias na ferramenta.

## 6.2 Trabalhos Futuros

Com a conclusão deste trabalho, sugerimos alguns trabalhos para complementá-lo em termos de ferramenta e avaliação.

O primeiro passo seria partir de um protótipo para uma ferramenta de uso público. Para tanto, precisaríamos evoluir no código do protótipo para dar suporte a vários tipos de sistema e resolver os problemas de gerência de configuração citados anteriormente. Nesse sentido, pode-se começar por unificar a execução do AutoMock em um arquivo executável, e não através do Ant, como é feito hoje, por exemplo.

Para aumentar o conhecimento sobre o esforço e a cobertura de interações, e até outras métricas, em diferentes contextos, definindo a validade da experimentação, sugere-se replicações do experimento em sistemas distintos e mais realistas, desenvolvendo testes com código *mock* para diferentes tipos de sistema.

Outro importante trabalho a ser realizado consiste em um estudo de caso em um sistema real, no mercado, por exemplo, onde teríamos a oportunidade de implantar a técnica de forma a avaliá-la não só em relação ao código gerado, mas em relação a outros aspectos, como usabilidade, possibilidade de integração com outras ferramentas, escalabilidade, retorno econômico, ganho de tempo em projeto, etc.

Como a necessidade da técnica proposta nesse trabalho surgiu a partir das necessidades do OurBackup, inicialmente pretendíamos em realizar um estudo de caso nesse sistema, porém esse já tinha todos os testes necessários com código *mock*. Daí, partimos para a mais nova versão, o OurBackup Enterprise, que também não deu certo, pois a abordagem de teste não foi completada, não havendo testes para produção de código *mock*. Além disso, o projeto OurBackup encerrou e não tivemos mais suporte para utilizá-lo. Ainda tentamos contatos com outros grupos de pesquisa na UFCG para tentar conseguir algum sistema de informação, típico para o uso de objetos *mock*, para realizar esse estudo de caso, mas não encontramos. Desta forma, esse trabalho ficou para o futuro.

Por fim, pretendemos publicar esse trabalho em outras conferências para divulgar os resultados e contribuir de forma mais concreta para a evolução da pesquisa na área de automação de teste de software.

# Bibliografia


[1] Ant. Disponível em: http://ant.apache.org/. Acesso em: Outubro de 2009.

[2] DynaMock. Disponível em: http://www.mockobjects.com/DynaMock.html. Acesso em: Outubro de 2009.

[3] GoogleMock. Disponível em: http://code.google.com/p/googlemock/. Acesso em: Outubro de 2009.

[4] IEEE Standard Glossary of Software Engineering Terminology, Standard 610.12. IEEE Press, 1990.

[5] JDK. Disponível em: http://java.sun.com/javase/. Acesso em: Outubro de 2009.

[6] JUnit. Disponível em: http://www.junit.org/. Acesso em: Abril de 2010.

[7] LSD - Laboratorio de Sistemas Distribuidos. Disponível em: http://lsd.ufcg.edu.br. Acesso em: Fevereiro de 2010.

[8] Mockito. Disponível em: http://mockito.org/. Acesso em: Outubro de 2009.

[9] SPSS Statistical Analysis. Disponível em: http://www.spss.com.br/spss. Acesso em: Março de 2010.

[10] *Experimental Software Engineering Issues: Critical Assessment and Future Directions, International Workshop Dagstuhl Castle, Germany, September 14-18, 1992, Proceedings*, volume 706 of *Lecture Notes in Computer Science*. Springer, 1993.

[11] BSTQB Certificação em Teste Advanced Level Syllabus (CTAL), 2007. Disponível em: http://www.bstqb.org.br/uploads/docs/ctal_syllabus_2007br.pdf.

[12] Araújo M. A., Barros M., and Travassos G. H. Utilização de métodos estatísticos no planejamento e análise de estudos experimentais em engenharia de software. In *ESELAW*, 2009.

[13] Araújo M. A., Barros M., Travassos G. H., and Murta L. Métodos estatísticos aplicados em engenharia de software experimental. In *XXI SBBD - XX SBES*, 2006.

[14] Nadia Alshahwan, Yue Jia, Kiran Lakhotia, Gordon Fraser, David Shuler, and Paolo Tonella. Automock: Automated synthesis of a mock environment for test case generation. In *Practical Software Testing: Tool Automation and Human Factors*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Alemanha, 2010.

[15] J. Malenfant andand C. Dony P. Cointe. Behavioral Reflection in a Prototype-Based Language, 1992. Workshop on Reflection and Meta-Level Architectures (IMSA'92), Tokyo.

[16] J. H. Andrews, S. Haldar, Y. Lei, , and F.C.H Li. Tool support for randomized unit testing. In *1st International Workshop on Random Testing*, pages 36–45, 2006.

[17] Nathaniel Ayewah, David Hovemeyer, J. David Morgenthaler, John Penix, and William Pugh. Using static analysis to find bugs. *IEEE Software*, 25:22–29, 2008.

[18] Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley, Boston, 2004.

[19] Robert V. Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[20] M. Boshernitsan, R. Doong, and A. Savoia. From daikon to agitator: lessons and challenges in building a commercial tool for developer testing. In *International Symposium on Software Testing and Analysis*, pages 169–180, 2006.

[21] B. Cabral, P. Marques, and L. Silva. In *RAIL: Code Instrumentation for.NET*, 2005.

[22] P. Caroli. Using JMock in Test Driven Development. Disponível em: http://www.theserverside.com/news/1365050/Using-JMock-in-Test-Driven-Development. Acesso em Abril de 2010.

[23] Roberta Coelho. Mini-curso: Teste de software. apresentadora: Roberta coelho. (puc-rio), sbes 2005.

[24] Rick D. Craig and Stefan P. Jaskiel. *Systematic Software Testing*. Artech House, Inc., Norwood, MA, USA, 2002.

[25] Christoph Csallner and Yannis Smaragdakis. Jcrasher: an automatic robustness tester for java. *Software: Practice and Experience*, 34:2004, 2004.

[26] Eclipse. The Aspectj Project, 2008. Disponível em: http://www.eclipse.org/aspectj. Acesso em: Novembro de 2008.

[27] S. Elbaum, H.N. Chin, M.B. Dwyer, and J. Dokulil. Carving differential unit test cases from system test cases. In *14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 253–264, 2006.

[28] M. Factor, A. Schuster, and K. Shagin. In *Instrumentation of Standard Libraries in Object-Oriented Languages: The Twin Class Hierarchy Approach*, 2004.

[29] J. Ferber. In *Computational Reflection in Class Based Object-Oriented Languages*, 1989.

[30] Mark Fewster and Dorothy Graham. *Software test automation: effective use of test execution tools*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1999.

[31] Martin Fowler. Mocks Aren't Stubs, 2007. http://www.martinfowler.com/articles/mocksArentStubs.html. Acesso em: Fevereiro de 2009.

[32] Steve Freeman, Nat Pryce, and Joshua Kerievsky. Point/counterpoint. *IEEE Software*, 24(3):80–83, 2007.

[33] Steve Freeman, Nat Pryce, Tim Mackinnon, Joe Walnes, and Thoughtworks Uk. Mock roles, not objects. In *In OOPSLA 04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 236–246. ACM Press, 2004.

[34] Tammo Freese. Easymock, 2001. Disponível em: http://www.easymock.org. Acesso em: Outubro de 2008.

[35] J. D. Gradecki and N. Lesiecki. *AspectJ: Aspect-Oriented Programming in Java*. John Wiley & Sons, Inc., 2003.

[36] Travassos G. H., Gurov D., and Amaral E. A. G. G. Introdução a engenharia de software experimental, 2002. Relatório Técnico ES-590/02-Abril, Programa de Engenharia de Sistemas COPPE/UFRJ.

[37] Raj Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation and Modeling*. John Wiley & Sons, 1991.

[38] Cem Kaner, James Bach, and Bret Pettichord. *Lessons Learned in Software Testing*. John Wiley & Sons, Inc., New York, NY, USA, 2001.

[39] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP*. SpringerVerlag, 1997.

[40] Ramnivas Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications Co., Greenwich, CT, USA, 2003.

[41] Panagiotis Louridas. Static code analysis. *IEEE Software*, 23:58–61, 2006.

[42] Tim Mackinnon, Steve Freeman, and Philip Craig. Endo-testing: unit testing with mock objects. pages 287–301, 2001.

[43] John D. McGregor and David A. Sykes. *A practical guide to testing object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[44] Gerard Meszaros. *XUnit Test Patterns: Refactoring Test Code*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2006.

[45] I. Moore and M. Cooke. MockMaker, 2004. Disponível em: http://www.mockmaker.org. Acesso em: Outubro de 2008.

[46] M. I. S. Oliveira, W. Cirne, F. Brasileiro, and D. Guerrero. On the impact of the data redundancy strategy on the recoverability of friend-to-friend backup systems. In *26th Brazilian Symposium on Computer Networks and Distributed Systems*, 2008.

[47] C. Oriat. Jartege: a tool for random generation of unit tests for java classes. In *2nd International Workshop on Software Quality, (SOQUA)*, pages 242–256, 2005.

[48] A. Orso and B. Kennedy. Selective capture and replay of program executions. In *Workshop on Dynamic Analysis*, pages 1–7, 2005.

[49] Roy Osherov. *The Art of Unit Testing: with examples in .NET*. Manning Publications, 2009.

[50] B. Pasternak, S. S. Tyszberowicz, and A. Yehudai. Genutest: A Unit Test and Mock Aspect Generation Tool. In *Haifa Verification Conference*, pages 252–266, 2007.

[51] Shari Lawrence Pfleeger. *Software Engineering: Theory and Practice*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[52] Ben Pryor. State-based vs. Interaction-based Unit Testing, 2007. http://benpryor.com/blog/2007/01/16/state-based-vs-interaction-based-unit-testing/. Acesso em: Fevereiro de 2009.

[53] Bisquerra R., Sarriera J. C., and Martínez F. *Introdução a estatística. Enfoque informático com o pacote estatístico SPSS*. Porto Alegre: Artmed Editora, 2004.

[54] Van Solingen R. and Berghout E. *The Goal/Question/Metric Method: a Practical Guide for Quality Improvement of Software Development*. The McGraw-Hill Companies, UK, 1999.

[55] D. Saff, S. Artzi, J. Perkins, and M. Ernst. Automated test factoring for java. In *Conference of Automated Software Engineering (ASE)*, pages 114–123, 2005.

[56] D. Saff and M. Ernst. In *Automatic mock object creation for test factoring*, pages 49–51, 2004.

[57] Dag I. K. Sj"berg, Bente Anda, Erik Arisholm, Tore Dybå, Magne J"rgensen, Amela Karahasanovic, Espen F. Koren, and Marek Vokác. Conducting realistic experiments in software engineering. In *ISESE '02: Proceedings of the 2002 International Symposium on Empirical Software Engineering*, page 17, Washington, DC, USA, 2002. IEEE Computer Society.

[58] S. Soares and P. Borba. Aspectj - programação orientada a aspectos em java. In *VI Simpósio Brasileiro de Linguagens de Programação*, 2002.

[59] Ian Sommerville. *Engenharia de Software*. Addison Wesley, São Paulo, SP, 8 edition, 2007.

[60] S. F. Souto, R. Miceli, and D. Guerrero. Generating mock-based test automatically. In *9th Latin-American Workshop on Aspect Orientation Programming*, 2009.

[61] Sabrina Souto. Automock: Interaction-based mock code generation. *The Student Forum at Latin-American Symposium on Dependable Computing*, 2009.

[62] N. Tillmann and W. Schulte. Mock-object generation with behavior. In *Conference of Automated Software Engineering (ASE)*, pages 365–368, 2006.

[63] W. Visser, C.S. Pasareanu, and S. Khurshid. Test input generation with java pathfinder. In *International Symposium on Software Testing and Analysis*, pages 97–107, 2004.

[64] V. Winck, D.V. e Junior. *AspectJ - Programação Orientada a Aspectos com Java*. Novatec, 2006.

[65] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Bjöorn Regnell, and Anders Wesslén. *Experimentation in software engineering: an introduction*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[66] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2005.

[67] H. Yuan and T. Xie. Substra: a framework for automatic generation of integration tests. In *International Workshop on Automation of Software Test*, pages 64–70, 2006.

# Apêndice A

# Sistemas para realização do experimento (*toy examples*)

## A.1 Sistema de Notas de Alunos

Imagine que estivéssemos desenvolvendo um software para automatizar as operações relacionadas as notas de um aluno em uma disciplina. Entre as muitas funcionalidades desse sistema, uma essencial seria a verificação da aprovação do aluno, que usaremos como exemplo.

Suponha que um aluno faz parte de uma caderneta de notas de uma disciplina qualquer, onde ele tem duas notas e a nota final, além de sua freqüência. Com esses dados podemos calcular a aprovação. Nesse exemplo, implementamos alguns testes para validar esse calculo de aprovação/reprovação, que se resumem em cinco condições ou casos de teste:

1. **Aluno reprovado por infrequência**: Freqüência inferior a 75;
2. **Aluno reprovado por nota**: Freqüência igual ou superior a 75 e média inferior a 30;
3. **Aluno aprovado por nota**: Freqüência igual ou superior a 75 e média igual ou superior a 70;
4. **Aluno aprovado na final**: Freqüência igual ou superior a 75 e média final igual ou superior a 50;
5. **Aluno reprovado na final**: Freqüência igual ou superior a 75 e média final inferior a 50.

No contexto deste sistema de notas de alunos, o participande teve que desenvolver código de objetos *mock* para que medir o tempo gasto nesse desenvolvimento. Segue o diagrama de classes na Figura A.1 correspondente a esse sisteminha. No código real, temos a implementação dessas classes e de algumas classes de teste, cada uma com um caso de teste acima e mais uma tratando uma condição de exceção. Então, o participante, teve que desenvolver objetos *mock* para dois esses testes, ou seja, isolar a unidade de teste substituindo as dependências por objetos *mock*. No caso, nosso foco é a classe `Caderneta.java`.

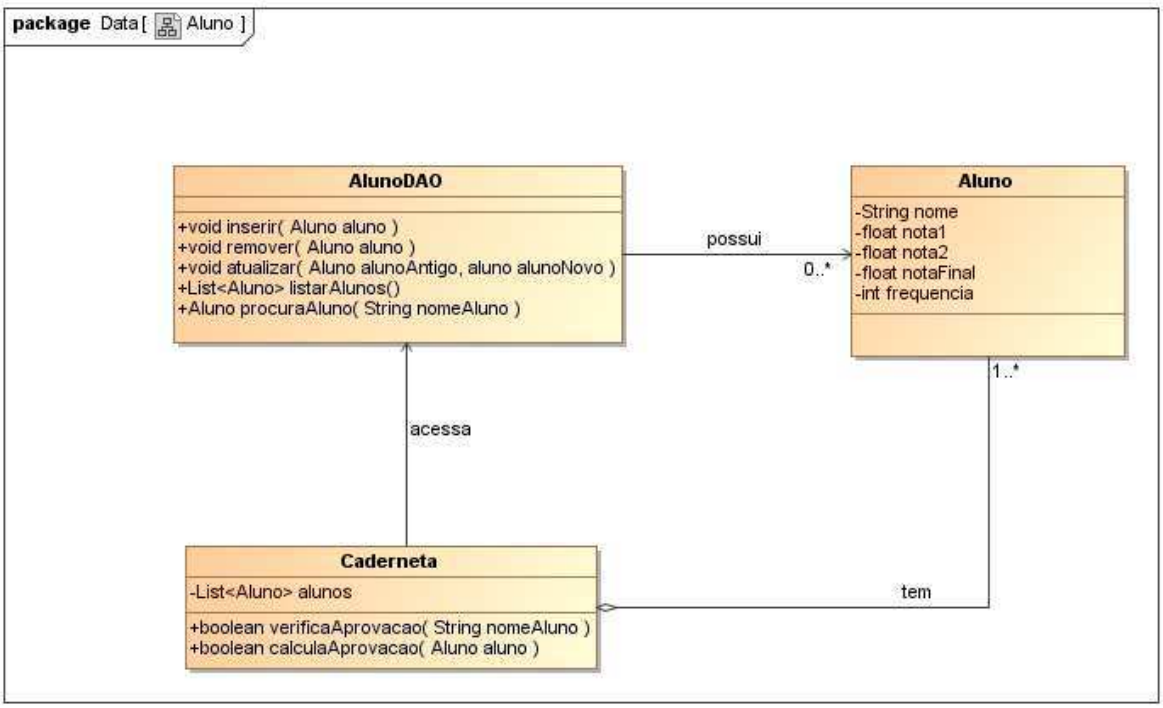



Figura A.1: Classes usadas para modelar o sistema de notas de alunos.

## A.2 Sistema de Autenticação

Imagine que estivéssemos desenvolvendo um software para automatizar as operações relacionadas à autenticação de um usuário em um determinado sistema. O sistema de autenticação tem como principal função verificar se um usuário está cadastrado no sistema, através se seu *login* (nome) e senha.

Nesse sistema, implementamos testes para validar se o *login* está realmente sendo feito de forma correta, ou seja, o usuário que está tentando fazer *login* num determinado sistema precisa estar cadastrado previamente no sistema de autenticação.

Segue um diagrama de classes do sistema de autenticação, representado pela Figura A.2, que é bastante simples. No código real, temos a implementação dessas classes e de alguns testes, o `LoginValidoTest.java` e o `LoginInvalidoTest.java` que são duas classes de teste prontas, onde o participante, irá desenvolver objetos *mock* para esses testes, ou seja, você terá que isolar a unidade de teste substituindo as dependências por objetos *mock*. No caso, nosso foco é a classe `Login.java`.

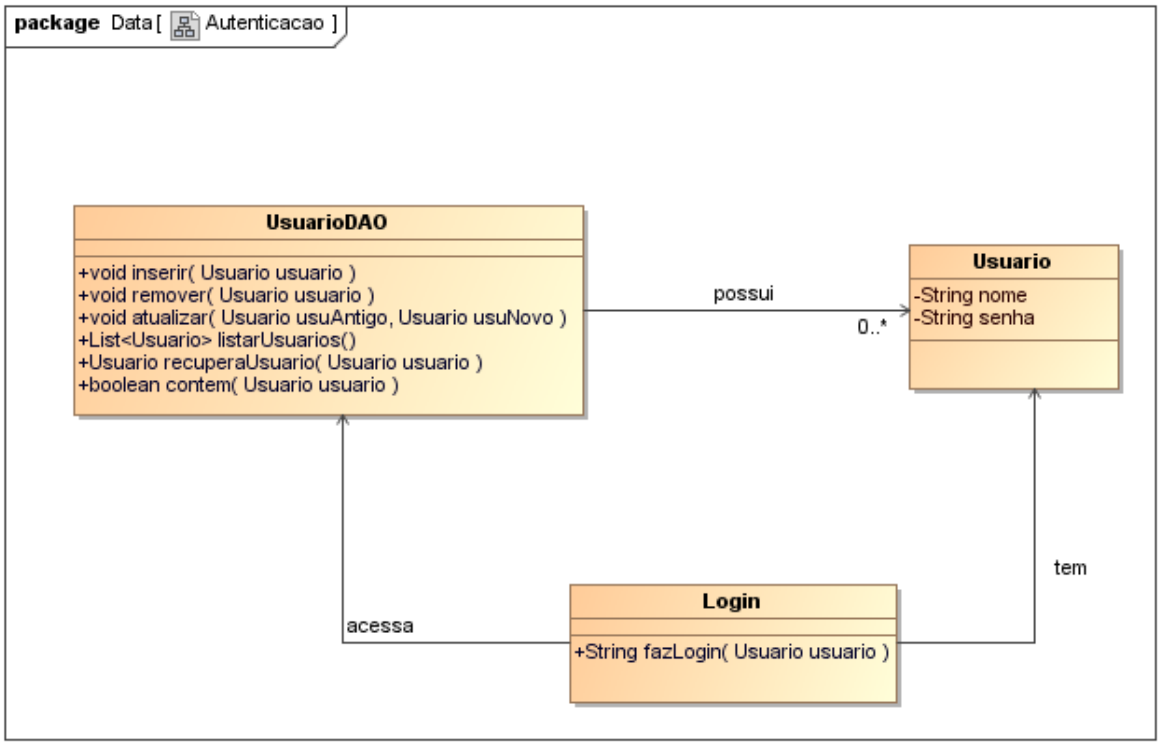



Figura A.2: Classes usadas para modelar o sistema de autenticação.

## A.3 Sistema de Pedido

Imagine que estivéssemos desenvolvendo um software para automatizar as operações relacionadas à geração de um pedido em um determinado estoque. Entre as muitas funcionalidades desse sistema, uma essencial seria a geração de uma nota de pedido, que usaremos como exemplo. Uma nota de pedido típica tem muitas características, mas iremos nos preocupar apenas com a formação desse pedido e o total a ser pago. Nesse exemplo, implementamos um teste para validar se o total do pedido está sendo calculado corretamente e outro para validar de uma linha de pedido está sendo adicionada com sucesso. A impressão de um pedido

(classe Pedido), como pode ser observada na Figura A.1, que é composta por linhas de pedido (classe `LinhaPedido`). Cada linha indica um produto, sua quantidade, e o preço total (quantidade de produtos da linha multiplicada pelo preço unitário do item). Um produto, por sua vez, possui um nome e um preço unitário. Na parte inferior da impressão do pedido é apresentado o valor total a ser pago, que é, como esperado, o somatório do valor total de cada linha do pedido.


Tabela A.1: Exemplo de uma nota de pedido.

No contexto deste sistema de pedido, o participante irá desenvolver código de objetos *mock* para que eu possa medir o tempo gasto nesse desenvolvimento. Segue na Figura A.3 o diagrama de classes correspondente a esse sisteminha. No código real, temos a implementação dessas classes e de alguns testes, dentre os quais temos o `AdicionaLinhaTeste.java` e o `TotalFinalTeste.java` que são duas classes de teste prontas, onde o participante, irá desenvolver objetos *mock* para esses testes, ou seja, você terá que isolar a unidade de teste substituindo as dependências por objetos *mock*. No caso, nosso foco é a classe `Pedido.java`.

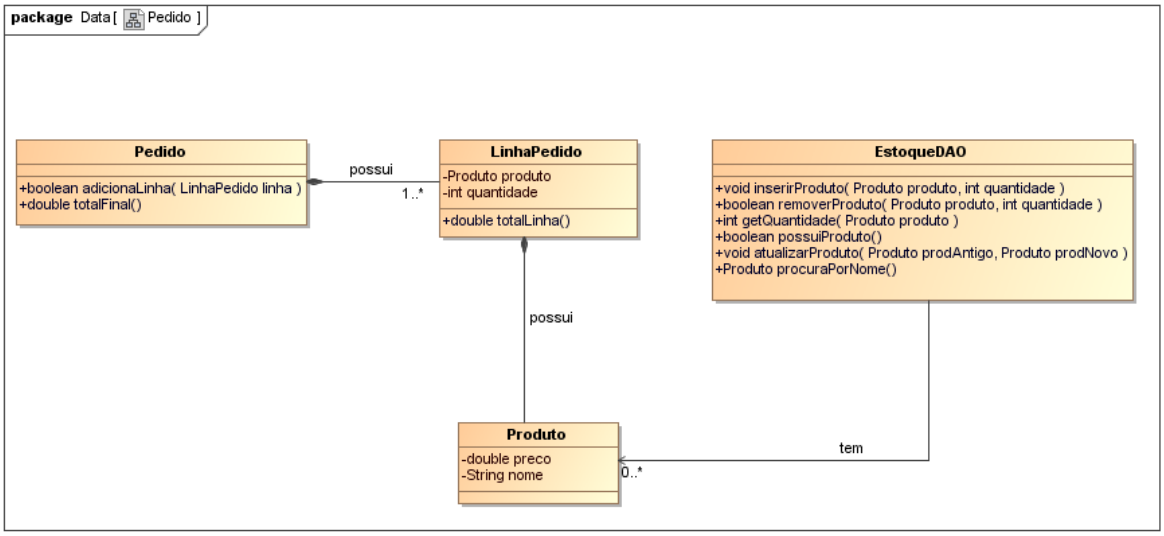



Figura A.3: Classes usadas para modelar um sistema de pedido.

# Apêndice B

# Detalhamento dos Módulos do AutoMock

## B.1 Primeira Fase: Análise Estática

O módulo responsável por esta fase é o *Static Analyzer*, cujo diagrama de classes está ilustrado na Figura B.1. O *Static Analyzer* tem por objetivos: identificar as entidades que colaboram com a CUT e a dependência estrutural entre da CUT e seus colaboradores. Esta identificação é realizada através de uma análise estática do código, que verifica os relacionamentos que envolvem a CUT e recupera apenas as entidades com as quais ela interage diretamente, que correspondem aos seus colaboradores.

A classe `IO` recebe os parâmetros de entrada da ferramenta, que são: SUT, CUT e teste, a partir de então passa para realizar a análise estática através da classe `IOHandler`, a qual implementa o algoritmo de análise estática exibido no Capítulo 3, no Código 3.1. Essa classe varre o código fonte do teste em busca das relações estruturais da CUT e, dentre elas, identifica entidades com as quais a CUT interage que são os colaboradores, representados pela classe `Collaborator`. Sendo assim, à medida que são identificados os colaboradores, eles são adicionados a uma estrutura de dados, representa pela classe `Collaborators`.

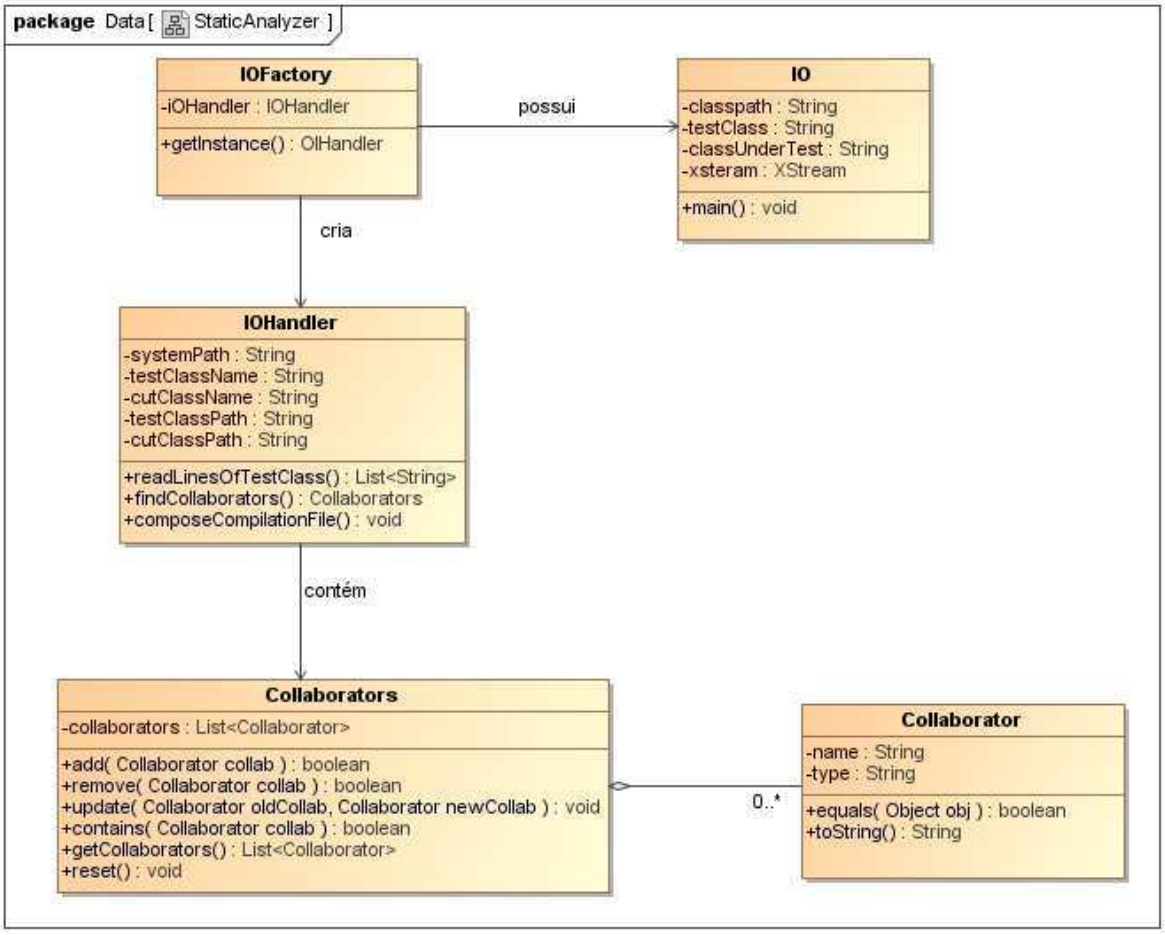



Figura B.1: Diagrama de classes do *StaticAnalyzer*.

## B.2 Segunda Fase: Análise Dinâmica

O módulo responsável por esta fase é o *Dynamic Analyzer*, apresentado na Figura B.2, que ilustra o diagrama de classes referente a este módulo. O *Dynamic Analyzer* tem como finalidade instrumentar o código do teste e executá-lo para capturar todos os *traces* de execução no contexto do teste e armazenar em um arquivo de *log*. Mas para tanto, se faz necessária a construção de um aspecto, `LogCollaborator`, que intercepta todas as execuções e chamadas a métodos do teste no contexto do SUT. Posteriormente, a ferramenta se encarrega de compilar o aspecto juntamente com o teste e executá-lo. Durante essa execução, o aspecto captura todas as informações dos objetos em execução através da entidade `ObjectRecord` e as armazena no arquivo de *log*.

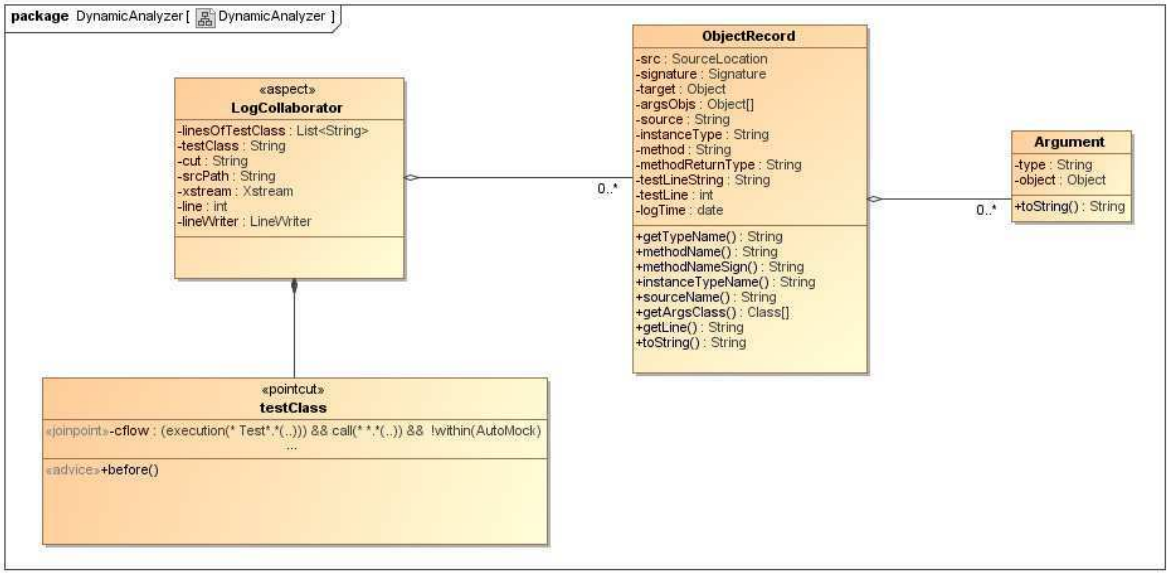



Figura B.2: Diagrama de classes do *DynamicAnalyzer*.

## B.3 Terceira Fase: Geração de Código de Teste com Objetos *Mock*

O módulo responsável por esta fase é o *Code Generator*, apresentado na Figura B.3, que ilustra o diagrama de classes referente a este módulo. O *Code Generator* tem por função gerar código de teste com objetos *mock* para os colaboradores e suas interações com a CUT, detectados nas fases anteriores. A classe que responsável por essa geração de código é a `CodeGenerator`, a qual se utiliza da classe `ObjectExecution`, que contém um mapa com todas as interações dos colaboradores, a partir do qual, o `CodeGenerator` pode traduzir essas interações em códigos de teste com objetos *mock*. Essa tradução é feita com base no JUnit [6] e o EasyMock [22], resultando em uma nova versão do teste inicial.

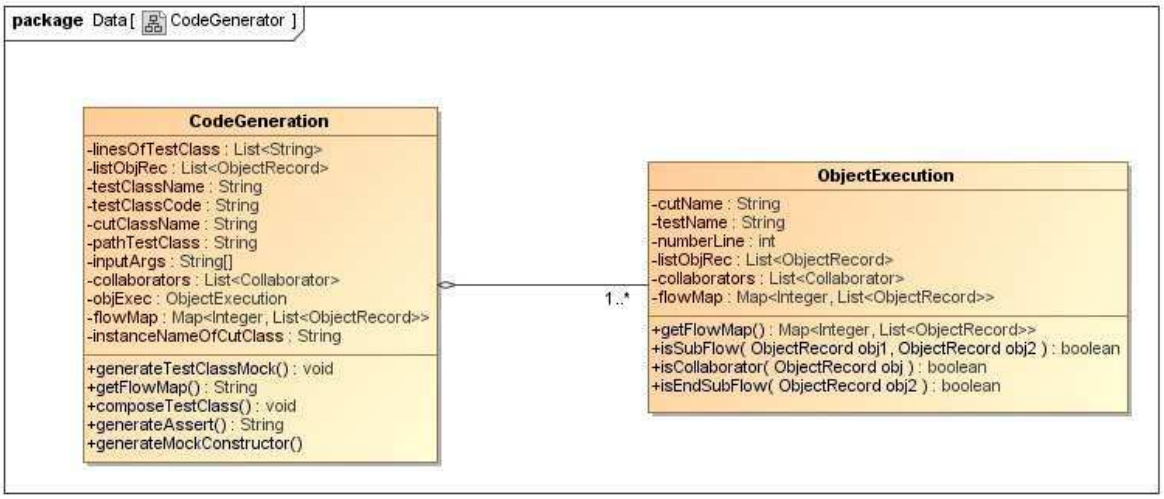



Figura B.3: Diagrama de classes do *CodeGenerator*.

Após discorrer sobre o funcionamento dos três módulos principais, apresentamos abaixo o diagrama de classes completo do AutoMock, exibido na Figura B.4, contendo as classes de todos os módulos integradas, inclusive as do módulo Util.

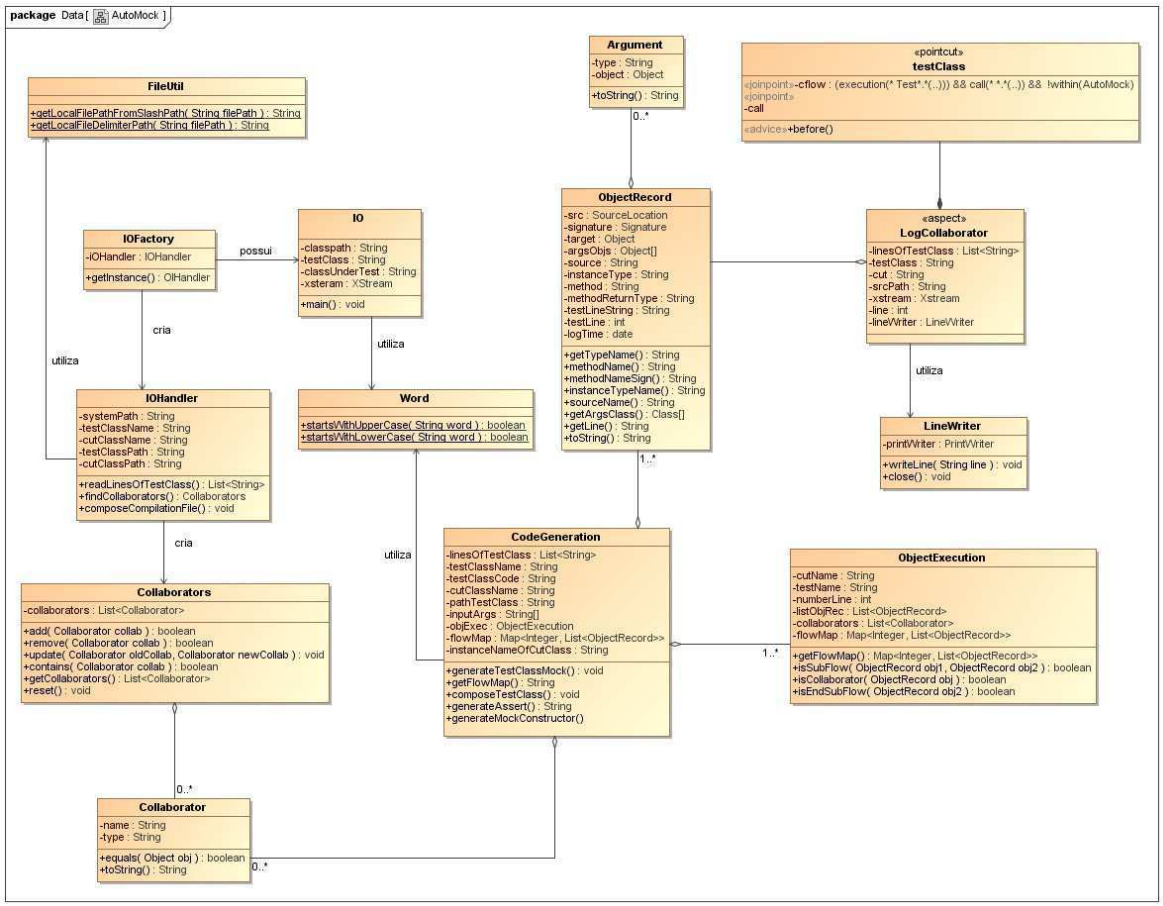



Figura B.4: Diagrama de Classes do AutoMock.

# Apêndice C

# Avaliação Qualitativa - Questionário

## C.1 Perfil do Participante

1. Quanto tempo de experiência você possui na linguagem Java?
   (a) Menos de 6 meses
   (b) Entre 6 meses e 1 ano
   (c) Mais de 1 ano
2. Quanto tempo de experiência você tem de uso com a IDE Eclipse?
   (a) Menos de 6 meses
   (b) Entre 6 meses e 1 ano
   (c) Mais de 1 ano
3. Quanto tempo de experiência possui em testes de unidade?
   (a) Menos de 6 meses
   (b) Entre 6 meses e 1 ano
   (c) Mais de 1 ano
4. Quanto tempo de experiência você possui com JUnit?
   (a) Menos de 6 meses

(b) Entre 6 meses e 1 ano

(c) Mais de 1 ano

5. Há aproximadamente quanto tempo você conhece o conceito de Objetos Mock?

   (a) Menos de 6 meses

   (b) Entre 6 meses e 1 ano

   (c) Mais de 1 ano

6. Quanto tempo você tem de experiência com o framework EasyMock?

   (a) Menos de 6 meses

   (b) Entre 6 meses e 1 ano

   (c) Mais de 1 ano

## C.2 Realização do Experimento

1. Qual o nível de dificuldade encontrado ao se desenvolver Objetos *Mock* para os testes do experimento?

   (a) Muito fácil

   (b) Fácil

   (c) Médio

   (d) Difícil

   (e) Muito difícil

2. Aponte, se possível, algumas das dificuldades encontradas para se desenvolver testes com Objeto *Mock*.

3. Foi necessário alterar o programa original para construir os testes com Objetos *Mock*?

   (a) Sim

   (b) Não

4. Se sim, quais alterações você teve que realizar e por quê?

## C.3 Avaliação da Técnica para Geração Automática de Código *Mock*

1. Você concorda que o desenvolvimento de código *mock* é uma tarefa repetitiva e custosa?

   (a) Sim

   (b) Não

2. Se sim, existe mais algum fator que contribua para que esta tarefa tenha tais características?

3. Apesar de existirem vários frameworks que auxiliam no desenvolvimento de código *mock*, esta tarefa ainda continua "manual". Sendo assim, você gostaria de uma ferramenta que automatizasse por completo esta tarefa, ou seja, que gerasse automaticamente código *mock* para um dado teste?

   (a) Sim

   (b) Não

4. Durante o mestrado desenvolvi uma técnica para automatizar o desenvolvimento de código *mock* para testes, sob a hipótese de reduzir o tempo, o esforço e produzir um código *mock* de qualidade. Você acha que esta técnica será útil no desenvolvimento de código *mock*?

   (a) Sim

   (b) Não

5. A respeito da técnica proposta, que vantagens e desvantagens você citaria?