

GUSTAVO DA MOTA RAMOS

Seleção entre estratégias de geração
automática de dados de teste por meio de
métricas estáticas de softwares orientados a
objetos

São Paulo

2018

GUSTAVO DA MOTA RAMOS

Seleção entre estratégias de geração automática de
dados de teste por meio de métricas estáticas de
softwares orientados a objetos

Versão original

Dissertação apresentada à Escola de Artes, Ciências e Humanidades da Universidade de São Paulo para obtenção do título de Mestre em Ciências pelo Programa de Pós-graduação em Sistemas de Informação.

Área de concentração: Metodologia e Técnicas da Computação

Versão corrigida contendo as alterações solicitadas pela comissão julgadora em xx de xxxxxxxxxxxxxxxx de xxxx. A versão original encontra-se em acervo reservado na Biblioteca da EACH-USP e na Biblioteca Digital de Teses e Dissertações da USP (BDTD), de acordo com a Resolução CoPGr 6018, de 13 de outubro de 2011.

Orientador: Prof. Dr. Marcelo Medeiros Eler

São Paulo

2018

Ficha catalográfica

Errata

Elemento opcional para versão corrigida, depois de depositada.

Dissertação de autoria de Gustavo da Mota Ramos, sob o título “**Seleção entre estratégias de geração automática de dados de teste por meio de métricas estáticas de softwares orientados a objetos**”, apresentada à Escola de Artes, Ciências e Humanidades da Universidade de São Paulo, para obtenção do título de Mestre em Ciências pelo Programa de Pós-graduação em Sistemas de Informação, na área de concentração Metodologia e Técnicas da Computação, aprovada em _____ de _____ de _____ pela comissão julgadora constituída pelos doutores:

Prof. Dr.

Presidente

Instituição:

Prof. Dr.

Instituição:

Prof. Dr.

Instituição:

Prof. Dr.

Instituição:

Aos meus pais, Nivaldo e Izildinha, que não mediram esforços para que eu chegasse até aqui. Eles são responsáveis pela maior herança da minha vida: meus estudos.

Lista de figuras

Figura 1 – Modelo de teste de software	18
Figura 2 – Hierarquia da classificação dos tipos de custo presentes em desenvolvi- mento de software	22
Figura 3 – Crescimento dos custos do tipo <i>nonconformance</i> ao longo do tempo de projeto	23
Figura 4 – Exemplo de problema do diamante	34
Figura 5 – Modelo de qualidade proposto por Srivastava e Kumar	37
Figura 6 – Distribuição de estado dos projetos após geração de dados de teste . .	45
Figura 7 – Quantidade de classes que atingiram cobertura de código superior ao outro algoritmo	46
Figura 8 – Quantidade de classes que consumiram menos tempo melhor em cada algoritmo	48
Figura 9 – Quantidade de classes que consumiram menos tempo melhor em cada algoritmo	49

Lista de algoritmos

Lista de tabelas

Tabela 1 – Distribuição das classes nas diferenças de cobertura entre os algoritmos	47
Tabela 2 – Correlação entre métricas CK e cobertura de código	50
Tabela 3 – Correlação entre métricas CK e o tempo de geração	50

Lista de abreviaturas e siglas

CUT	<i>Class under test</i>
SUT	<i>Software under test</i>

Sumário

1	Introdução	13
1.1	Problema de pesquisa	14
1.2	Objetivos	15
1.3	Justificativa	15
1.4	Método de pesquisa	16
1.5	Organização do documento	16
2	Aspectos conceituais e econômicos	17
2.1	Teste de software	17
2.1.1	Caso de teste	18
2.1.2	Mocks	18
2.1.3	Testes manuais	19
2.1.4	Testes automatizados	19
2.2	Teste caixa preta	20
2.3	Particionamento de equivalência	20
2.4	Análise do valor limite	21
2.5	Qualidade de software e aspectos econômicos	21
3	Geração automática de dados de teste	24
3.1	Algoritmos baseados em busca	25
3.2	Algoritmo baseado em busca aleatória	25
3.3	Algoritmos de busca baseados em evolução	27
3.3.1	Algoritmos genéticos	28
3.3.1.1	O algoritmo evolutivo $1 + (\lambda, \lambda)$	28
3.3.1.2	O algoritmo evolutivo $\mu + \lambda$	29
3.3.1.3	O algoritmo evolutivo MOSA	29
3.4	Algoritmo de busca randômica ou algoritmos evolutivos ?	29
3.5	EvoSuite	30
4	Métricas de Softwares orientados a objetos	32

4.1	Medidas diretas e indiretas	33
4.2	O conjunto de métricas CK	33
4.2.1	<i>Weighted methods per class (WMC)</i>	33
4.2.2	<i>Depth of inheritance tree (DIT)</i>	33
4.2.3	<i>Number of children of a class (NOC)</i>	35
4.2.4	<i>Coupling between object classes (CBO)</i>	35
4.2.5	<i>Response for a class (RFC)</i>	35
4.2.6	<i>Lack Of Cohesion Of Methods (LCOM)</i>	36
4.3	Modelo proposto por Srivastava e Kumar	36
4.4	Métricas extraídas com o SB4SE	37
5	Mineração de dados	39
5.1	Classificação	39
5.2	Classificador naive bayes	40
5.3	Weka	41
6	Resultados e Discussões	43
6.1	Materiais	43
6.2	Extrações de dados a análises primárias	44
6.3	Extrações de dados das métricas SB4SE	51
	Referências ¹	52

¹ De acordo com a Associação Brasileira de Normas Técnicas. NBR 6023.

1 Introdução

Produtos de software de diferentes tamanhos e complexidades são utilizados todos os dias em atividades profissionais ou de entretenimento. Em qualquer circunstância, entretanto, a falta de qualidade em produtos pode caracterizar uma situação preocupante para seus produtores (HILBURN; TOWHIDNEJAD, 2002) (BINDER, 1994), uma vez que cada vez mais os níveis aceitáveis de qualidade de um software estão aumentando, tanto no que se refere ao seu comportamento observável externamente quanto ao seu processo de desenvolvimento e estrutura interna (NORTHROP, 2006) (BASHIR; BANURI, 2008) (GRAHAM; VEENENDAAL; EVANS, 2008).

O teste de software (TAHIR; MACDONELL; BUCHAN, 2014) é a maneira mais popular de verificar se um software atende às especificações descritas e cumpre o papel desejado pelos interessados (SOMMERVILLE et al., 2008). Ela consiste na execução do programa sob teste para revelar seus defeitos. Para isso, casos de teste são gerados de maneira que satisfaçam a diversos critérios, em geral baseados na especificação e na implementação do software (PEZZÈ; YOUNG, 2008).

Entretanto, gerar casos de teste para testar o software nos mais diversos contextos e dados de entrada é uma das atividades mais custosas no ciclo de vida de software, requisitando muito tempo e esforço para o seu planejamento, execução e manutenção. (TAHIR; MACDONELL; BUCHAN, 2014). Consequentemente, pesquisadores e profissionais da área passaram então a desenvolver abordagens para gerar casos de teste automaticamente e assim reduzir o custo e a complexidade desta tarefa. Diversas técnicas já foram utilizadas com teste objetivo: geração de testes randômicos (PACHECO; ERNST, 2007); execução simbólica (CADAR; SEN, 2013); teste baseado em modelos (DICK; FAIVRE, 1993); e teste baseado em buscas (MCMINN, 2004; HARMAN; MANSOURI; ZHANG, 2012). Diversas ferramentas já foram desenvolvidas utilizando uma ou mais dessas técnicas, como, por exemplo, Randoop (PACHECO; ERNST, 2007), DART (GODEFROID; KLARLUND; SEN, 2005), CUTE (SEN; MARINOV; AGHA, 2005), JCut (SEN; AGHA, 2006), Klee (CADAR; DUNBAR; ENGLER, 2008), JavaPathFinder (VISSER; PăSăREANU; KHURSHID, 2004), PEX (TILLMANN; HALLEUX, 2008) e EvoSuite (FRASER; ARCURI, 2011).

Em particular, a técnica de teste baseado em busca tem sido amplamente utilizada para a geração de casos de teste. Neste contexto, destaca-se a ferramenta EvoSuite que é capaz de gerar conjuntos de casos de teste para programas escritos em Java automaticamente.

Ela utiliza um algoritmo genético para selecionar os melhores dados de teste e produzir os conjuntos de casos de teste que atingem altos níveis de cobertura estrutural e escore de mutação (FRASER; ARCURI, 2011). Esta ferramenta já foi extensivamente avaliada no que se refere à eficácia e escalabilidade (FRASER; ARCURI, 2013; ROJAS et al., 2017; FRASER et al., 2015; FRASER; ARCURI, 2014), e recentemente foi a ferramenta que obteve a maior pontuação na competição de ferramentas de geração de teste unitários promovida anualmente pelo Workshop on Search-Based Software Testing (FRASER et al., 2017).

Recentemente, novos algoritmos evolutivos foram adicionados à ferramenta EvoSuite como alternativa ao algoritmo genético padrão para a geração de casos de teste (CAMPOS et al., 2017): algoritmo genético monotônico, algoritmo genético de regime permanente (do inglês, steady state), $1 + (\lambda, \lambda)$, $\mu + \lambda$, MOSA (Many-Objective Sorting Algorithm) e DynaMOSA (dynamic MOSA); além da geração randômica já implementada anteriormente. Um estudo comparativo utilizando 346 classes selecionadas randomicamente de um conjunto de 117 projetos de código aberto mostrou que a escolha do algoritmo utilizado na geração de casos de teste influencia os resultados, e que diferentes configurações podem favorecer diferentes algoritmos.

1.1 Problema de pesquisa

O estudo realizado para comparar a eficácia dos algoritmos na geração de casos de teste implementados na ferramenta EvoSuite mostrou que a escolha do algoritmo importa. Entretanto, nenhuma análise mais criteriosa foi realizada para identificar em que situações ou para que tipo de classe um algoritmo é melhor do que outro. Portanto, as seguintes questões de pesquisa emergem neste contexto:

- Quais são as características das classes sob teste que podem influenciar a eficiência e a eficácia dos algoritmos evolutivos utilizados na geração de casos de teste?
- Existem padrões de características para os quais um algoritmo específico geralmente vai obter melhor resultado?

1.2 Objetivos

Neste contexto, o objetivo geral deste projeto de pesquisa é definir uma hiper-heurística para escolher o melhor algoritmo evolutivo na geração de casos de teste de acordo com o problema em mãos. Desta forma, ao invés de utilizar um único algoritmo para gerar todos os casos de teste de um projeto, os casos de teste de cada classe podem ser gerados por diferentes algoritmos dependendo das suas características.

Os objetivos específicos deste projeto de pesquisa são os seguintes:

1. Mapear sistematicamente as métricas de software mais utilizadas para estimar a testabilidade de um software.
2. Selecionar métricas de testabilidade que tem maior probabilidade de influenciar os resultados dos algoritmos na geração de casos de teste. Essa seleção será realizada utilizando-se a correlação entre as métricas e os resultados obtidos pelos algoritmos evolutivos em termos de cobertura de código dos casos de teste gerados por eles.
3. Utilizar um algoritmo de reconhecimento de padrões para analisar as métricas de testabilidade de uma classe e classificá-la de acordo com o algoritmo de geração de testes mais adequado.
4. Implementar uma hiper-heurística em uma ferramenta de geração de casos de teste para selecionar o melhor algoritmo evolutivo de acordo com a classe a ser testada.

1.3 Justificativa

Os diversos algoritmos e abordagens de geração de casos de teste encontrados no ambiente acadêmico e industrial possuem pontos fracos e fortes. Em especial, a técnica de teste baseado em busca tem sido amplamente utilizada para gerar casos de teste, e muitos dos alguns dos seus algoritmos conseguem convergir para resultados ótimos mais rapidamente do que os outros, mas não conseguem tratar de situações mais complexas, por exemplo. Portanto, utilizar o algoritmo que consegue obter os melhores resultados e convergir mais rapidamente para cada classe do projeto a ser testado irá aumentar consideravelmente a eficiência e a eficácia de ferramentas de geração de casos de teste como a EvoSuite, por exemplo. Além disso, os resultados obtidos podem encorajar a

implementação de algoritmos evolutivos que são utilizados em outras áreas mas que ainda não utilizados na geração de casos teste.

1.4 Método de pesquisa

Este projeto será, conduzido usando um método indutivo com objetivos de caráter exploratório, descritivo e experimental. Mais detalhes dos materiais e métodos de pesquisa utilizados neste projeto de pesquisa estão apresentados no Capítulo 5.

1.5 Organização do documento

Esta proposta de pesquisa está organizada da seguinte forma. No Capítulo 2 são apresentados os conceitos mais importantes que embasam este projeto.

2 Aspectos conceituais e econômicos

No presente capítulo será apresentado uma síntese dos conceitos base com o objetivo de fundamentação teórica dos capítulos seguintes e auxílio na estrutura do presente documento.

2.1 Teste de software

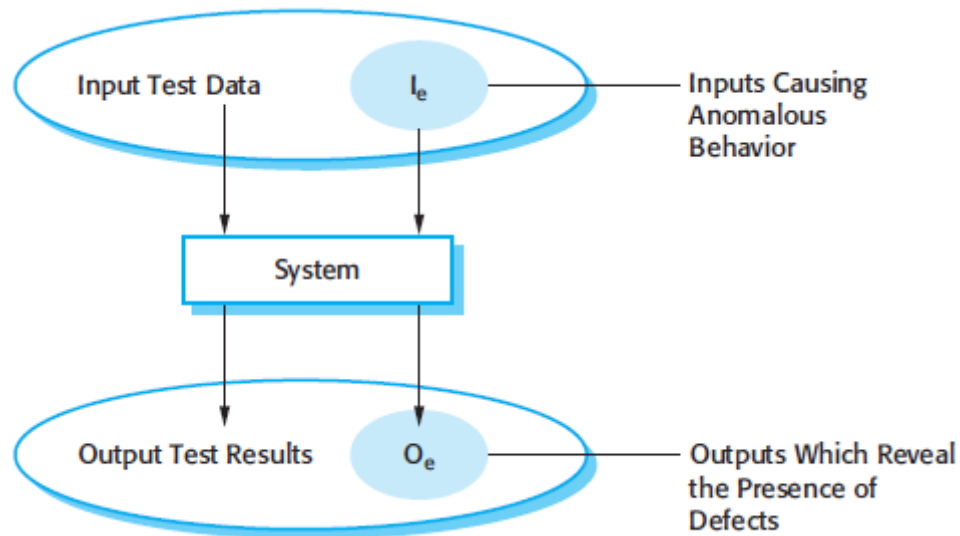
O teste de software é uma subárea da engenharia de software que compreende as atividades capazes de determinar se um software em análise contém erros (GOODENOUGH; GERHART, 1975) com objetivo encontrar falhas e não de provar a corretude de software, estes não são capazes de demonstrar se um software em análise está livre de defeitos ou que irá se comportar como esperado em todas as circunstâncias, ou seja, testes são responsáveis somente por evidenciar a presença de possíveis erros e não a falta destes (PFLEEGER; ATLEE, 2010). A partir destes é possível validar se o software cumpre o propósito de desenvolvimento e por fim eliminar possíveis defeitos e assim garantir maior qualidade de software (SOMMERVILLE, 2010).

De acordo com (PRESSMAN, 2009) e (DAVIS, 1995) são notórios os princípios de teste:

1. Todos os testes devem ser rastreáveis aos requisitos de usuários;
2. Testes devem ser planejados antes de sua execução;
3. O princípio de pareto deve ser aplicado ao teste de software;
4. Testes devem começar as menores unidades afim de atingir partes maiores de software;
5. Testes exaustivos não são possíveis.

O fluxo de testes é composto por modelar o requisito em formato de casos de teste, contendo os detalhes de execução, dados de entradas e estado esperado do SUT. Após a criação do caso de teste, este pode ser utilizado para execução, para executar um caso de teste, é necessário simular sua execução e validar, a partir dos resultados extraídos da execução, possíveis anomalias e erros, ou seja, testes de software são executados com dados artificiais (SOMMERVILLE, 2010). Com os dados de saída é possível verificar se o estado encontrado é o mesmo esperado, este modelo é demonstrado na figura 1.

Figura 1 – Modelo de teste de software



Fonte: (SOMMERVILLE, 2010)

2.1.1 Caso de teste

Um caso de teste é um conjunto de possíveis entradas, condições de execução e resultados esperados de acordo com seus objetivos, estes quando compilados compõe uma instância única de execução em termos de entrada, execução e resultados esperados, estas saídas quando analisadas em relação aos dados esperados após execução determinam se o estado encontrado pela aplicação é esperado (SINGH, 2014). Os casos de teste são projetados para cenários particulares de teste aos quais é desejado comparar estados e resultados coletados a partir do resultado de sua execução com requisitos funcionais anteriormente definidos (JACOB; PRASANNA, 2016).

2.1.2 Mocks

Ao construir testes automatizados é comum testar componentes de softwares que dependam de outros softwares, e desenvolvedores normalmente necessitam testar o software com todas suas dependências ou simulações (SPADINI et al., 2017). Ao testar todas as dependências juntamente, desenvolvedores passam a ter um comportamento similar ao ambiente de produção (SPADINI et al., 2017).

Apesar dos benefícios, algumas dependências como bancos de dados e chamadas à serviços online podem ser custosos para serem testados, nestes casos desenvolvedores podem usar mocks ([SPADINI et al., 2017](#)). Mocks são usados para substituir dependências reais em softwares ao simular suas características mais importantes, usualmente métodos em mocks retornam valores desejados de acordo com os valores de entrada ([SPADINI et al., 2017](#)).

2.1.3 Testes manuais

O teste manual é aquele executado através de um profissional de qualidade, o qual geralmente possui um caso de teste onde são descritos os passos de execução e por fim, o responsável toma decisões de acordo com a finalização da execução ([KAPROCKI; PEKOVIC; VELIKIC, 2015](#)).

Espera-se que os testes construídos para aplicações sejam, em geral, compostos pelos seguintes itens:

- Estado inicial do teste;
- Eventuais entradas para uso;
- Sequência de passos para execução;
- Uma ou mais saídas esperadas.

2.1.4 Testes automatizados

Os testes automatizados possuem uma estrutura comum aos testes manuais, porém o maior diferencial deste tipo de teste é delegar para a máquina a execução e possivelmente a validação dos resultados obtidos. Apesar de testes automatizados possuírem diversos papéis, como testes funcionais, testes de stress, é notório que estes possuam a capacidade de executar uma grande quantidade de casos de teste em um curto período de tempo, com maior precisão ([KAPROCKI; PEKOVIC; VELIKIC, 2015](#)).

2.2 Teste caixa preta

O teste caixa preta é uma categoria de testes funcionais a qual em sua execução não existe o conhecimento da estrutura interna do SUT (XU et al., 2016), o conceito nasceu da ideia da estrutura interna do SUT estar coberta por uma caixa preta de maneira que o profissional de testes não tenha conhecimento de sua estrutura interna, somente de seu funcionamento, desta maneira o profissional de qualidade concentra no que o software deve fazer e elimina o raciocínio baseado na maneira como é feito (JACOB; PRASANNA, 2016).

Geralmente ao efetuar um teste de caixa preta, o profissional de qualidade ou ferramentas de execução automatizada de scripts de teste poderão interagir com a interface de usuário de maneira que possam fornecer dados de entrada sem quaisquer noções da lógica de negócio por trás da interface utilizada (XU et al., 2016). Idealmente este tipo de teste é guiado por especificações formais dos requisitos elicitados ou modelos de dados de teste, porém na prática estas especificações nem sempre estão disponíveis (WALKINSHAW; FRASER, 2017).

2.3 Particionamento de equivalência

O particionamento de equivalência é um critério de teste o qual divide o domínio possível de entradas do SUT em classes de dados as quais os casos de teste são derivados (JACOB; PRASANNA, 2016). Neste critério todo o conjunto de entradas é particionado em subconjuntos ou classes, onde cada classe representa um conjunto de entradas similares para determinadas funcionalidades e especificações (JACOB; PRASANNA, 2016).

O conceito por trás deste critério é dividir as entradas de maneiras que os conjuntos façam parte de classes consideradas iguais, estes conjuntos são conhecidos como partições de equivalência ou classes de equivalência, desta maneira a quantidade de casos de teste é menor dado que é necessário testar uma única condição para cada partição ou classe de equivalência (JACOB; PRASANNA, 2016).

2.4 Análise do valor limite

A análise de valor limite é um critério similar ao critério de classe de equivalência no que diz respeito à estratégia de particionamento afim de diminuir a quantidade de casos de teste necessários para cobrir um requisito, o seu ponto principal é dividir o domínio de uma ou mais entradas de maneira que diferentes faixas de valor possam ser representadas por uma quantidade menor de entradas, estas entradas compostas pelos valores limites das partições.

Segundo (JACOB; PRASANNA, 2016), muitos dos desenvolvedores cometem erros ao implementar requisitos que dependem da utilização de valores limite para execução de asserções condicionais, desta maneira uma quantidade maior de defeitos ocorre nos valores limite de um domínio de entrada se comparado a defeitos que ocorrem por valores intermediários da mesma faixa.

O profissional de qualidade deve estar alinhado com os seguintes princípios para mapear uma análise de valor limite:

- Se a condição de uma entrada é compreendida entre valores m e n , os casos de teste devem ser projetados para os valores m , n assim como entradas que representem os valores abaixo e acima de m e n , desta maneira o caso de teste deverá ser projetado para o conjunto de entradas m, n, m ;

2.5 Qualidade de software e aspectos econômicos

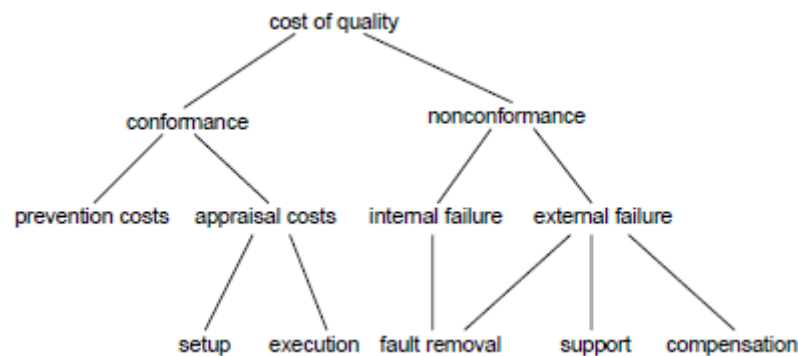
A Garantia de Qualidade ou *Quality Assurance* é considerada uma das partes mais caras do desenvolvimento de software (WAGNER; SEIFERT, 2005), o conceito de garantia de qualidade tornou-se mais evidente após a primeira conferência de engenharia de software com o propósito de estabelecer melhores princípios e estratégias econômicas para atingir um estado viável de software, esta conferência de em 1968 foi responsável inclusive a disciplina de teste e controle de qualidade em diversas fases de desenvolvimento de software desde então (REPASI, 2009).

De fato, os custos de garantia de qualidade são elevados (WAGNER; SEIFERT, 2005) (KOREL, 1990) já que consistem em todo e qualquer valor investido em atividades cujo propósito almeja qualidade de software (PRESSMAN, 2009), porém é notável que

fatores econômicos em melhoria de qualidade não são bem aceitos por todos os interessados no desenvolvimento de software e que inclusive exista uma certa confusão sobre o valor de negócio da qualidade de desenvolvimento de software (SLAUGHTER; HARTER; KRISHNAN, 1998) já que teste de software é uma atividade trabalhosa e cara (KOREL, 1990) mas estes investimentos estes são importantes e devem ser planejados visto que cada valor gasto em horas de trabalho e não investido em retrabalho pode ser usado para melhorias rápidas em produtos e processo existentes (SLAUGHTER; HARTER; KRISHNAN, 1998).

Os custos direcionados à qualidade de software são categorizados em *conformance* e *nonconformance* (SLAUGHTER; HARTER; KRISHNAN, 1998)(PRESSMAN, 2009) como demonstrado na figura a seguir:

Figura 2 – Hierarquia da classificação dos tipos de custo presentes em desenvolvimento de software



Fonte: (WAGNER; SEIFERT, 2005)

Os custos de *conformance* caracterizam os valores com o intuito de atingir maior qualidade de produto, este por sua vez é dividido em custo de prevenção e custo de avaliação. (WAGNER; SEIFERT, 2005)

Custos de prevenção são aqueles associados com o intuito de prevenir defeitos antes que possam ocorrer, geralmente são compreendidos por treinamentos, equipamentos, , revisões técnicas formais, atividades de planejamento de qualidade e reviews de produto (WAGNER; SEIFERT, 2005) (PRESSMAN, 2009).

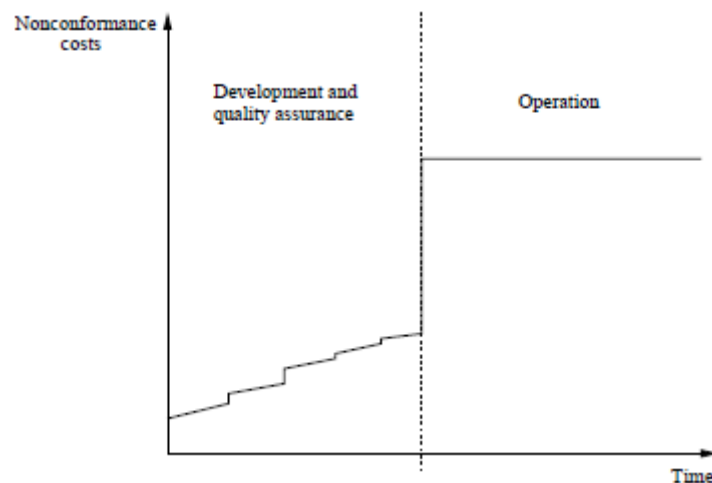
Custos de avaliação compreendem custos relacionados a medidas e extração de métricas, avaliação e auditoria de produtos.(WAGNER; SEIFERT, 2005), incluem atividades para ganhar conhecimento da condição do software em análise para cada início de ciclo, incluem: avaliações de processo e entre processos e manutenção (PRESSMAN, 2009).

Custos de não conformidade são os custos relacionados aos cenários em que não seguem o planejado e defeitos produzindo um erro e por fim levando à falha (WAGNER; SEIFERT, 2005), nesta categoria estão as subcategorias de custos de falha externa e custos de falha interna (PRESSMAN, 2009).

Os Custos de falha externa caracterizam os custos associados aos defeitos após liberação do produto de software para uso, são exemplos: Resolução de chamados, retorno e substituição de produtos e garantias (PRESSMAN, 2009) enquanto os custos de falha interna são os custos aplicados para remoção de falhas antes de liberação para uso.

O custo relativo para encontrar e reparar um defeito aumenta consideravelmente na linha do tempo entre as fases iniciais e finais do ciclo de desenvolvimento, sendo assim, custos do tipo *nonconformance* se tornam mais caros ao longo do tempo de projeto, como demonstrado a seguir (WAGNER; SEIFERT, 2005):

Figura 3 – Crescimento dos custos do tipo *nonconformance* ao longo do tempo de projeto



Fonte: (WAGNER; SEIFERT, 2005)

Apesar do custo relativo para encontrar e reparar um defeito aumenta consideravelmente na linha do tempo, o processo de desenvolvimento de um software flui entre os diversos ciclos, e por consequência, diferentes tipos de investimentos são feitos com o intuito de garantir maiores níveis de qualidade (PRESSMAN, 2009).

3 Geração automática de dados de teste

A manutenção de softwares é usualmente vista não somente como uma tarefa difícil mas cara visto que as atividades ligadas à esta somam por volta de 60% do investimento total de um software (SHAMSHIRI et al., 2018), concomitantemente os profissionais de desenvolvimento de software necessitam testar o SUT utilizando-se de casos de teste que revelem falhas, aumentando a cobertura de caminhos e minimizando de custo e tempo simultaneamente. (DAVE; AGRAWAL, 2015).

O alto custo presente em testes de mutação e geração de mutantes pode ser reduzido pela geração automática de dados de teste (DAVE; AGRAWAL, 2015), a geração automática de testes poupa o tempo investido por desenvolvedores na escrita de testes unitários (SHAMSHIRI et al., 2018).

A geração automática de dados de teste consiste em uma tarefa importante no processo de testes de software, ela possui o objetivo de melhorar a qualidade de um determinado software under test (SUT) com o intuito de aumentar a qualidade do software e diminuir o custo investido. (SHAMSHIRI et al.,). A geração automática de dados de teste possui também um papel importante no processo de desenvolvimento de software visto que esta pode afetar diretamente a qualidade final de um produto de software (DAVE; AGRAWAL, 2015).

Os testes gerados podem ser incluídos nos repositórios de código do SUT como qualquer outro artefato de software (SHAMSHIRI et al., 2018), porém conjuntos de testes volumosos podem levar mais tempo para serem executadas e precisam ser avaliados antes de serem incluídos em uma suíte de testes (SHAMSHIRI et al.,). Ferramentas como a EvoSuite têm se mostrado eficazes em atingir alta cobertura de código em diferentes tipos de softwares orientados a objetos. (CAMPOS et al., 2017)

A maneira mais fácil de avaliar conjuntos de testes automaticamente gerados é avaliar sua capacidade de expor falhas. (SHAMSHIRI et al., 2018), sendo assim, as avaliações de conjuntos de testes automaticamente gerados ocorre por meio de métricas como cobertura de linhas de código ou estimativas da habilidade de detecção de falhas, sendo um exemplo deste o score de mutação. (SHAMSHIRI et al., 2018)

3.1 Algoritmos baseados em busca

Algoritmos baseados em busca têm se mostrado efetivos na geração de dados de teste quando otimizados para cobertura de código de diferentes tipos de software (CAMPOS et al., 2017) também mostraram-se eficazes quando aplicada para geração de suítes de testes unitários otimizados para cobertura de código em classes de softwares orientados a objetos (CAMPOS et al., 2017)

Novas técnicas de busca foram desenvolvidas nos últimos anos (ROJAS; FRASER, 2017), algoritmos de busca estão entre as técnicas mais bem sucedidas para automaticamente gerar dados de teste e testes unitários. (ROJAS; FRASER, 2017)

Técnicas como buscas randômicas, buscas locais, simulated annealing e algoritmos genéticos são algumas das técnicas mais utilizadas. (ROJAS; FRASER, 2017)

Uma abordagem popular é o uso de algoritmos evolutivos no qual os indivíduos utilizados na população são testes os quais após gerados compõe suítes de testes completas, a construção destas suítes ocorre com objetivo de encontrar suítes de teste os quais atingem a cobertura de código máxima a partir de técnicas baseadas em evolução natural (CAMPOS et al., 2017)

3.2 Algoritmo baseado em busca aleatória

O algoritmo de busca randômica, ou random search, é um algoritmo o qual o mecanismo deve ser estabelecido para escolher os candidatos em um espaço X , o processo randômico pode ser definido como a função de densidade de probabilidade $P(x)$ a qual expressa a probabilidade de qualquer candidato presente em um espaço X de possíveis soluções (KARNOPP, 1963). Apesar de estudos na área, existe um número de técnicas padrão, técnicas como estas geralmente baseiam-se em um gerador randômico entre 0 e 1, o qual a saída pode ser utilizada para produzir uma aproximação da distribuição desejada. (KARNOPP, 1963)

O algoritmo de busca randômica é um algoritmo o qual não utiliza operações de crossover, mutações ou seleções, ao contrário dos algoritmos genéticos seu funcionamento baseia-se puramente em estratégias de substituição (CAMPOS et al., 2017). O algoritmo de busca randômica consiste em repetidamente gerar amostras de candidatos presentes em um espaço de busca de soluções possíveis para o problema a qual o candidato já presente

na solução parcial é substituído se a aptidão do indivíduo gerado supera a aptidão do indivíduo já presente na resolução do problema em análise. (CAMPOS et al., 2017)

O Algoritmo de Random testing, é uma variação dos algoritmo baseado em busca aleatória o qual utiliza random search como base para construir suítes de testes de maneira incremental (CAMPOS et al., 2017) , esta diferencia-se das outras por gerar sequencias aleatórias de casos de teste juntamente aos dados de entrada também aleatoriamente gerados (SHAMSHIRI et al.,), se um novo caso de teste gerado cobre um novo ramo do SUT o qual não foi coberto nas iterações anteriores, o novo caso de teste é adicionado ao conjunto (SHAMSHIRI et al.,), os casos de testes são tratados como amostras individuais, se uma amostra gerada aumenta a cobertura de código, a amostra recém gerada é então inserida na suíte de testes, caso contrário é descartada. (CAMPOS et al., 2017)

Segundo (KARNOPP, 1963) técnicas de busca randômicas possuem alguns vantagens de utilização como:

- Facilidade de desenvolvimento: Por sua facilidade de programação do algoritmo, bons resultados podem ser obtidos com pouco esforço de desenvolvimento e algoritmos generalistas para resoluções de problemas de busca.
- Baixo custo: Os valores randômicos utilizadas em buscas randômicas podem ser gerados com pouco custo computacional, muitos dos métodos necessitam meios de armazenamento e operações de comparação mutualmente simples.
- Eficiência: Em muitos casos, um algoritmo altamente de busca altamente randômico será muito mais eficiente se comparado a um algoritmo altamente determinístico, isto ocorre pois o tempo gasto em decidir a próxima estratégia de um algoritmo determinístico é investido em explorar e avaliar uma quantidade maior do universo de possíveis soluções.
- Flexibilidade: Algoritmos de busca randômicos possuem a flexibilidade de variar entre algoritmos puramente randômicos e altamente determinísticos de acordo com o problema em análise e o estágio de busca.
- Informação provida e utilizada: Algoritmos de busca randômicos podem coletar informações sobre a função de busca em execução e utilizar esta informação para guiar a busca, apesar destas informações serem do tipo local, podem ser utilizadas para otimização.

Em contraponto à facilidade de construção proporcionada pelo algoritmo de random search, possui a desvantagem do tamanho dos casos de testes gerados e consequentemente, o tempo de execução proporcionado pela execução e validação de cobertura destes. (SHAMSHIRI et al.,)

Em contraponto à facilidade de construção proporcionada pelo algoritmo, este possui uma a desvantagem do tamanho dos casos de testes gerados e consequentemente, o tempo de execução proporcionado (SHAMSHIRI et al.,). A técnica de random search também é dificultada por valores de entradas os quais por determinados tipos de entrada como constantes, valores específicos de strings os quais não são difíceis de serem gerados sem algoritmos de seeding. (SHAMSHIRI et al.,)

3.3 Algoritmos de busca baseados em evolução

Algoritmos evolutivos são inspirados na evolução natural, estes algoritmos têm se mostrado eficazes quando utilizados em diferentes tipos de problemas de otimização (CAMPOS et al., 2017).

A população é gradualmente otimizada usando operações geneticamente inspiradas como a operação de crossover, esta operação combina material genético de no mínimo dois indivíduos para assim gerar uma nova prole , e mutação, uma operação na qual troca altera material genético em um indivíduo a uma baixa probabilidade, para assim garantir diversificação genética (CAMPOS et al., 2017). No contexto de geração de suítes de teste, os indivíduos da população são caracterizados por conjuntos de casos de teste, também chamados de suítes de teste. (CAMPOS et al., 2017)

A seleção de indivíduos é guiada por uma função de aptidão, a qual pontua melhor os indivíduos mais adaptados para um problema de otimização (CAMPOS et al., 2017). A função de aptidão baseia-se em objetivos decobertura que vão da cobertura de código a cobertura de ramificações de código (CAMPOS et al., 2017), sendo assim indivíduos com valores altos de aptidão são mais adaptados ao problema em questão e mais prováveis de sobreviver a novas iterações e produzirem novos candidatos. (CAMPOS et al., 2017)

O aumento do número de objetivos de cobertura pode afetar a performance do algoritmo evolutivo (CAMPOS et al., 2017)

3.3.1 Algoritmos genéticos

Os algoritmos genéticos estão entre os algoritmos evolutivos mais usados em diversos problemas de otimização pois podem ser facilmente implementados e apresentam bons resultados (CAMPOS et al., 2017). Diversas variações do algoritmo genético padrão foram propostos com o intuito de melhorar sua efetividade (CAMPOS et al., 2017)

Uma das variações de algoritmo genético é sua versão monotônica do algoritmo genético padrão (Standard GA), a versão monotônica somente inclui, após operações de mutação e validação em cada iteração, somente o melhor conjunto da prole ou o melhor indivíduo da prole anterior (CAMPOS et al., 2017). Dentre as variações de algoritmos genéticos padrão, existe a variação Steady State GA, esta variação utiliza a mesma estratégia de substituição presente na variação monotônica, a prole descendente substitui os ascendentes da população atual imediatamente a ocorrência da fase de mutação. (CAMPOS et al., 2017)

3.3.1.1 O algoritmo evolutivo $1 + (\lambda, \lambda)$

A versão do algoritmo evolutivo $1 + (\lambda, \lambda)$, introduzida por Doerr, possui como primeiro passo a geração de uma população randômica de tamanho 1, em seguida, a operação de mutação é aplicada afim de criar λ versões diferentes com mutação do indivíduo em análise (CAMPOS et al., 2017). A alta probabilidade de mutação ocorre para incentivar que a exploração seja mais rápida no espaço de busca disponível. (CAMPOS et al., 2017)

No próximo passo, o operador de mutação é aplicado novamente com uma probabilidade maior de mutação, dada por K , onde K é usualmente maior que um, isto permite que, na maioria dos casos, mais de um gene seja mutado por cromossomo. Por fim, a operação de crossover acontece uniformemente aos ascendentes e aos melhores descendentes para criar uma prole de tamanho λ (CAMPOS et al., 2017). A operação de crossover unificada entre o melhor indivíduo com os λ mutantes e seu ascendente foi proposta para amenizar os defeitos causados pela mutação agressiva, após essa operação, toda a prole é avaliada e somente o melhor indivíduo é selecionado (CAMPOS et al., 2017)

Se o melhor descendente é melhor avaliado do que o seu ascendente, a população criada de tamanho um é então substituída pela melhor prole. (CAMPOS et al., 2017)

3.3.1.2 O algoritmo evolutivo $\mu + \lambda$

O algoritmo evolutivo $\mu + \lambda$ é um algoritmo baseado em mutações onde o número de ascendentes e descendentes são restringidos respectivamente a μ e λ (CAMPOS et al., 2017). Cada gene sofre mutação independente com probabilidade $1/n$, após a operação de mutação, a prole gerada é comparada com cada ascendente com o intuito de manter os melhores ascendentes presentes até então (CAMPOS et al., 2017).

Entre as diferentes versões do algoritmo, existem duas melhores configurações como $(1+\lambda)$ e $(1+1)$ os quais a população tem tamanho 1 e o número da prole é limitado a 1 no caso de parâmetros $(1+1)$ (CAMPOS et al., 2017).

3.3.1.3 O algoritmo evolutivo MOSA

O algoritmo MOSA, Many-Objective Sorting Algorithm, contrapõe os outros algoritmos por tratar cada objetivo de cobertura como um objetivo de otimização independente (CAMPOS et al., 2017). Este algoritmo é uma variação do algoritmo NSGA-II o qual melhor avalia os melhores testes gerados por cada objetivo de cobertura antes não coberto, independente da relação entre os testes e a população. (CAMPOS et al., 2017)

Para que seja possível trabalhar com o grande número de objetivos de cobertura, o qual ocorre por conta da combinação dos diversos objetivos de cobertura, a extensão do algoritmo DynaMOSA dinamicamente seleciona os critérios (CAMPOS et al., 2017). Tanto o algoritmo MOSA quanto o algoritmo DynaMOSA demonstraram atingir alta cobertura de alguns critérios selecionados quando comparados aos algoritmos genéticos tradicionais. (CAMPOS et al., 2017)

3.4 Algoritmo de busca randômica ou algoritmos evolutivos ?

Apesar de muitos dos aspectos destes algoritmos avaliados em detalhes, a influência de diferentes algoritmos em diferentes casos recebeu pouca atenção na literatura (CAMPOS et al., 2017). Vários algoritmos foram propostos porém nenhum dos algoritmos pode ser considerado melhor em todos os domínios de problemas, alguns algoritmos performam melhor do que outros de acordo com as características presentes no domínio do problema. (CAMPOS et al., 2017)

É teoricamente impossível desenvolver um algoritmo de geração de dados de teste o qual performa bem em todos os tipos de problemas disponíveis (CAMPOS et al., 2017). Uma abordagem comum em problemas de geração de dados de testes na engenharia de software é utilizar um algoritmo genético primariamente e somente depois refiná-lo ou compará-lo com outros algoritmos para assim verificar qual dos algoritmos e soluções disponíveis é apropriado para o problema em questão. (CAMPOS et al., 2017)

O estudo promovido por (CAMPOS et al., 2017) demonstra que algoritmos genéticos claramente performaram melhor do que algoritmos randômicos, em média os evolutionary algorithms possuem maior cobertura quando comparados a random search e random testing. (CAMPOS et al., 2017)

(CAMPOS et al., 2017) afirma a necessidade compreender as diferentes características de software as quais influenciam diferentes coberturas de teste nas diferentes técnicas para que assim seja possível desenvolver uma hyper-heurística a qual seleciona e adapta o algoritmo para cada problema de geração em específico.

3.5 EvoSuite

Existem várias ferramentas de geração de dados de testes com base em buscas disponíveis em uma gama de linguagens de programação, principalmente Java e .NET. Ferramentas as quais variam de técnicas de random search como Randoop, Jcrasher, JTEExpert, NightHawk, T3 ou Yeti-Test até ferramentas baseadas em técnicas evolutivas como EvoSuite, eToc ou Testful. (SHAMSHIRI et al.,)

A EvoSuite (EVOSUITE...,) é uma ferramenta baseada em técnicas de busca que também utiliza algoritmos genéticos na geração automática de suítes de testes para classes escritas em Java. (FRASER et al., 2017). Para uma determinada classe, referenciada na ferramenta como alvo, e seu classpath completo, o caminho o qual o ambiente de execução possa encontrar as classes compiladas como bytecode e respectivas dependências (CLASSPATH...,), a EvoSuite automaticamente produz um conjunto de testes unitários escritos para o framework de execução de testes Junit (JUNIT...,) com o intuito de maximizar a cobertura de código. (FRASER et al., 2017).

A efetividade da EvoSuite tem sido avaliada tanto como uma ferramenta open-source quanto uma ferramenta de apoio a testes nas áreas industriais se tratando de cobertura

de código, efetividade na busca de falhas e inclusive aumentando a produtividade de desenvolvimento (FRASER et al., 2017).

A ferramenta conseguiu o primeiro lugar nas segunda e quarta edições do unit testing tool competition além de conseguir o segundo lugar na terceira edição (FRASER et al., 2017), a configuração da EvoSuite para a competição em maioria manteve seus valores padrão, já que estes foram ajustados extensivamente (FRASER et al., 2017). Com um budget de 480 segundos, a ferramenta atingiu em média 66.5% de cobertura de caminhos, 0.9% superior se comparado ao ano anterior com um score de mutação de 50.7%, 9.7% super do que o ano anterior. (FRASER et al., 2017)

Além das últimas melhorias, diversas correções foram aplicadas desde a última edição do unit testing tool competition em 2017 (FRASER et al., 2017), no ano de 2017 a ferramenta passou a utilizar mocks em seus testes através da ferramenta Mockito (SITE. . . ,) (FRASER et al., 2017). Após atingir determinada porcentagem do tempo de busca disponível, a ferramenta considera o uso de mocks ao invés de classes reais, somente caminhos que não puderam ser cobertos por meio de mocks resultaram com o uso de mocks. (FRASER et al., 2017)

4 Métricas de Softwares orientados a objetos

Projeto e desenvolvimento de softwares orientados a objeto são conceitos populares no cenário atual de desenvolvimento de software (SRIVASTAVA; KUMAR, 2013), neste contexto é notável que classes e métodos são estruturas básicas neste gênero de software (KAN, 1995). O desenvolvimento de software orientado a objetos difere de outros paradigmas pois requer uma abordagem voltada para regras de negócio completamente diferente das tradicionais, dado na maneira como a decomposição funcional e fluxo de dados ocorrem (SRIVASTAVA; KUMAR, 2013).

A análise e projeto de softwares orientados a objeto focam nos objetos como estruturas primárias de computação (KAN, 1995) (SRIVASTAVA; KUMAR, 2013), nesta cada classe é composta por dados e operações realizadas, os quais são representações de uma única entidade real de negócio (SRIVASTAVA; KUMAR, 2013).

As operações realizadas são caracterizadas pelos métodos das classes, a quantidade de funcionalidade, representada por método, agregada ao um software orientado a objeto pode ser estimada com base na quantidade de classe, métodos e variantes destes, utilizados (KAN, 1995). Desta maneira, é comum relacionar métricas de softwares orientados a objeto a valores baseados em atributos de classes e métodos, sejam linhas de código, complexidade, entre outros (KAN, 1995).

As métricas de software tornaram-se elementos primordiais em diversos domínios de Engenharia de software, e principalmente no âmbito de garantia de qualidade, já que toda a informação reunida por métricas pode ser submetida à diversos tipos de análise e comparação com dados históricos com o objetivo de avaliar e garantir qualidade de software (SRIVASTAVA; KUMAR, 2013). Métricas de software podem ser utilizadas para prever atributos de qualidade em tempo de execução, inclusive para aplicações cujo requisito é ser uma aplicação de tempo real (SRIVASTAVA; KUMAR, 2013). O real valor de métricas de software vem da sua representação dos atributos de software externos, muitos destes com alto valor de negócio, como confiabilidade, manutenibilidade, reusabilidade, testabilidade e eficiência (SRIVASTAVA; KUMAR, 2013), os quais são requisitos de qualidade descritos pela ISO9126 (ZEISS et al., 2007).

4.1 Medidas diretas e indiretas

Uma medida direta de software é uma métrica a qual o cálculo de seu valor não depende nenhum outro atributo além do principal utilizado em seu cálculo (SRIVASTAVA; KUMAR, 2013). Exemplos claros de medidas em produtos são: LOC (*Lines of code*), velocidade de execução, tamanho de memória, quantidade de defeitos reportados (SRIVASTAVA; KUMAR, 2013).

Uma medida indireta de software é uma métrica a qual envolve a medida de um ou mais atributos de software (SRIVASTAVA; KUMAR, 2013). As medidas diretas são, em geral, mais fáceis de serem coletadas.

4.2 O conjunto de métricas CK

Em 1994 Chidamber e Kemerer proporam seis métricas de projeto e complexidade de softwares orientados a objeto, as quais, futuramente, tornaram-se o conjunto de métricas CK (KAN, 1995).

Chidamber e Kemerer aplicaram as seis métricas do conjunto (as quais serão definidas em seções posteriores) em estudos empíricos de duas empresas, uma usando C++ e outra Smalltalk (KAN, 1995), o sumário desta é demonstrado na tabela.

4.2.1 *Weighted methods per class (WMC)*

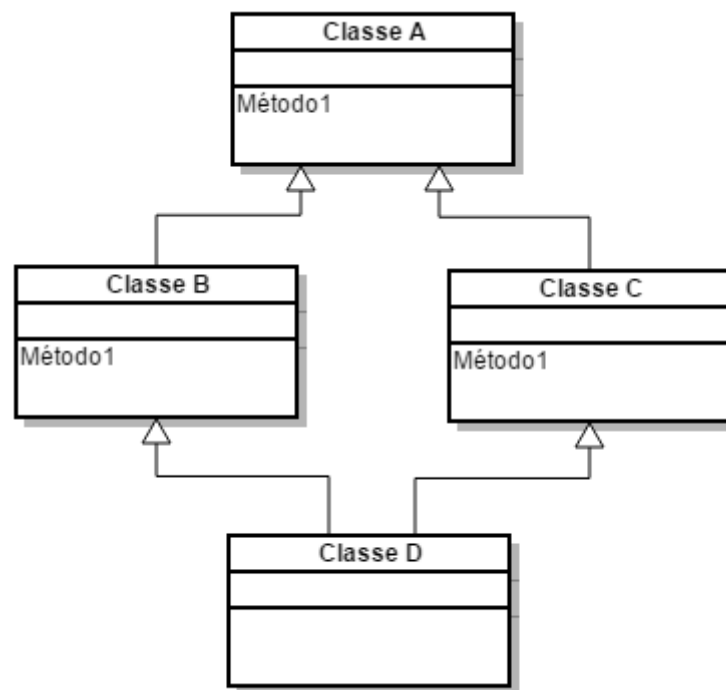
A WMC é uma métrica contabilizada pela soma das complexidades dos métodos, onde a complexidade é dada pelo *cyclomatic complexity number* de McCabe (CCN) (KAN, 1995) (WATSON; MCCABE; WALLACE, 1996). Medir o *cyclomatic complexity number* não é trivial de implementação dado que nem todos os métodos são acessíveis de acordo com a hierarquia dos objetos, isso é resultado por conta da herança aplicada no projeto do componente (KAN, 1995).

4.2.2 *Depth of inheritance tree (DIT)*

A DIT é uma medida que compreende o valor do comprimento máximo de um caminho em uma árvore de herança, este comprimento é dado pela distância entre o

nó em análise até o nó raiz (KAN, 1995). A definição da DIT baseia-se na premissa de que desenvolvedores lidam com linguagens de programação orientadas a objeto as quais permitam que uma classe possua no máximo uma classe pai (BRUNTINK; DEURSEN, 2004), linguagens alheias a este conceito são conhecidas por possuir herança múltipla, presente em linguagens como C++ e Python. Muitas linguagens orientadas a objeto não suportam herança múltipla por conta do problema do diamante, supondo a estrutura a seguir:

Figura 4 – Exemplo de problema do diamante



Fonte: Gustavo Ramos, 2017

Neste caso, um objeto instanciado do tipo D possui acesso para executar o **Método 1**, porém o compilador pode não saber a qual referência de **Método 1** executar já que o objeto do tipo D herda tanto de B e de C quanto de A, todos os quais possuem o **Método 1**. Este caso é conhecido por problema do diamante, por conta deste, linguagens como Java não permitem a herança múltipla. Dada a premissa de que a linguagem do software em análise não suporta herança múltipla, ao calcular o DIT, o número de ancestrais do nó c em análise, corresponde à profundidade de c na árvore de herança (BRUNTINK; DEURSEN, 2004). A fórmula que descreve o cálculo da DIT para um determinado nó c é dado a seguir:

$$DIT(c) = |Ancestrais(c)| \quad (1)$$

4.2.3 *Number of children of a class (NOC)*

O NOC é uma métrica de simples cálculo, caracterizada pelo número de sucessores imediatos(subclasses) na árvore de hierarquia, calculada com base em uma classe c em análise (KAN, 1995). Seu cálculo é dado pela fórmula a seguir:

$$NOC(c) = |Filhos(c)| \quad (2)$$

4.2.4 *Coupling between object classes (CBO)*

Uma classe A é acoplada a uma classe B se a classe A invoca uma método ou utiliza uma variável de uma instância de B (KAN, 1995), sendo assim, o CBO é dado pelo número de classes a qual uma classe c em análise está acoplada, ou seja, seu cálculo é dado por:

$$CBO(c) = |Quantidadedeclasesquerelaciona(c)| \quad (3)$$

4.2.5 *Response for a class (RFC)*

O RFC é caracterizado pelo número de métodos que podem ser executados na resposta de uma mensagem recebida por uma instância de uma classe (KAN, 1995), ou seja, o RFC é uma contagem do número de métodos de uma classe c em análise e o número de métodos de outras classes que são invocados pelos métodos de c (BRUNTINK; DEURSEN, 2004).

Quanto maior o número de métodos que podem ser invocados indiretamente através de uma chamada de um método, maior a complexidade de uma classe, basicamente o RFC captura o tamanho de conjunto de respostas de uma classe (KAN, 1995). O RFC é calculado pelo número de métodos locais mais o número de métodos chamados por métodos locais (KAN, 1995).

4.2.6 *Lack Of Cohesion Of Methods (LCOM)*

A independência pode ser medida por critérios qualitativos: coesão e acoplamento, a coesão é a medida relação funcional de um módulo ([PRESSMAN, 2009](#)), neste contexto, a coesão de uma classe é dada pela proximidade de métodos locais à instâncias de variáveis na classe, alta coesão indica uma boa subdivisão das classes ([KAN, 1995](#)).

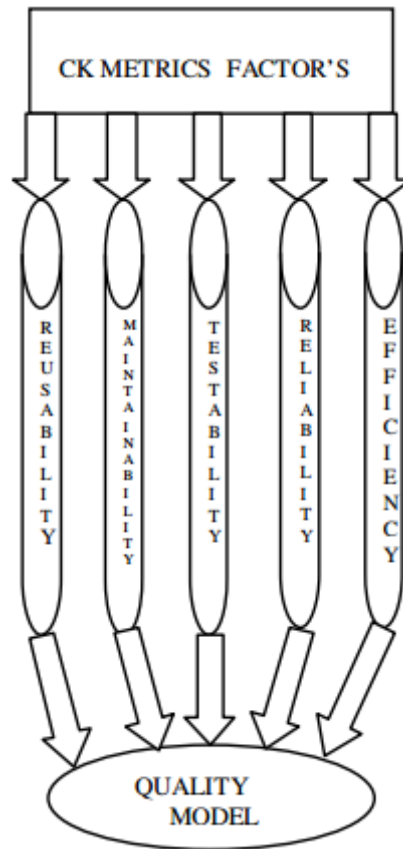
A métrica LCOM representa a dissimilaridade dos métodos em uma classe pelo uso de instâncias de variáveis, a alta de coesão aumenta a complexidade e por consequência, oportunidade de ocorrência de erros durante o processo de desenvolvimento ([KAN, 1995](#)).

4.3 Modelo proposto por Srivastava e Kumar

Srivastava e Kumar proporam em 2013 um modelo de métricas para validação de qualidade de software com a justificativa de que nenhuma das métricas tratavam acoplamento ou similaridades como relações transacionais, sendo assim, métricas que mapeiam estas características são passíveis de incorporação no modelo. No estudo em questão, diversos dados foram coletados de projetos orientados a objeto nas linguagens Java e C++ e neste constatado que diversas métricas são baseadas em ideias similares e representam informações redundantes, sendo assim, foi validado que um subconjunto de métricas pode ser utilizado para predição de falhas.

O modelo proposto por ([SRIVASTAVA; KUMAR, 2013](#)) atingiu um nível de acurácia superior a 80% na predição de falhas em classes.

Figura 5 – Modelo de qualidade proposto por Srivastava e Kumar



Fonte: (SRIVASTAVA; KUMAR, 2013)

É observado na literatura que métricas de softwares orientados a objetos aliadas aos modelos de classificação e predição existentes possuíam efetividade em encontrar relação entre identificação e classificação de falhas e qualidade de software (SURESH, 2016). O fato ocorre pois métricas de software são a melhor ferramenta para avaliar a qualidade de softwares orientados a objetos. (SURESH, 2016)

É observado que muitos conjuntos de métricas disponíveis provém informações de qualidade de softwares orientados a objetos. (SURESH, 2016)

4.4 Métricas extraídas com o SB4SE

Além de características que possam ser descritas por métricas C.K, é possível também utilizar métricas as quais descrevam características de métodos, já que métodos compõe as ações presentes dem classes. No trabalho desenvolvido por (ELER; ENDO; DURELLI, 2016), diversas métricas de métodos foram extraídas utilizando execução

simbólica, métricas como: Número de loops (NLp)número de loops aninhados (NNLp); quantidade de constantes (NCtt); número de variáveis (NVar); quantidade de variáveis do tipo inteiro (NInt); quantidade de variáveis do ponto flutuante (NFit); quantidade de referências nulas (NNull); Quantidade de variáveis do tipo string (NStr); número de objetos (NObj); Número de arrays (NArr).

5 Mineração de dados

Mineração de dados é o processo de extrair conhecimento valioso de um grande agrupamento de dados, desconhecido e de valor para o problema em questão (SETH; BANKA, 2016; SHARMA; SINGH; SINGH, 2015), processo este onde ocorre a análise de dados a partir de diferentes perspectivas, resumizando com o intuito de obter informações úteis para tomada de decisão (SHARMA; SINGH; SINGH, 2015). Ela é utilizada em diversos campos como robótica, decisões de marketing, decisões corporativas, inteligência de negócio e segurança (SETH; BANKA, 2016).

Na área de mineração de dados, distinguir diferentes classes a partir de um volume de dados e classificar a partir destes é uma técnica importante (SETH; BANKA, 2016), sendo assim, a classificação é uma subárea de mineração de dados importante, dado que ela é capaz de analisar dados e organizá-los em classes a partir de suas características (SETH; BANKA, 2016).

5.1 Classificação

A classificação é uma técnica vastamente utilizada no domínio de mineração de dados, onde escalabilidade e eficiência são requisitos imediatos em classificação de grandes conjuntos de dados (SHARMA; SINGH; SINGH, 2015). A classificação é uma atividade não supervisionada (SHARMA; SINGH; SINGH, 2015), a qual já existe o conhecimento desde o início quais são as classes disponíveis a serem utilizadas pelo algoritmo, o algoritmo então usa a informação de classes e características presentes em um dataset para treinamento e validação (SHARMA; SINGH; SINGH, 2015).

Um problema de classificação é conhecido estruturalmente por receber uma entrada de dados, conhecido por conjunto de dados de aprendizagem (dataset), o qual é caracterizado por uma série de características (features) de análise e respectivas classes (SETH; BANKA, 2016).

O principal objetivo de algoritmos de classificação é construir um modelo de classificação utilizando o conhecimento obtido a partir do conjunto de dados de aprendizagem e avaliar uma porcentagem destes a partir do conhecimento adquirido. (SETH; BANKA, 2016)

Diferentes algoritmos de classificação usam diferentes técnicas para encontrar relação entre os dados e as classes (SHARMA; SINGH; SINGH, 2015), estas relações estão sumarizadas em um modelo, o qual pode ser aplicado em novas instâncias para assim classifica-las. (SHARMA; SINGH; SINGH, 2015)

Um modelo pode também ser utilizado para validação, onde novas instâncias são submetidas a um modelo e avaliadas em seguida para verificar se foram classificadas corretamente, este processo é chamado de validação do modelo (SHARMA; SINGH; SINGH, 2015).

Dentre os classificadores mais famosos, o mais conhecido é o classificador bayseano. (SETH; BANKA, 2016), o classificador bayseano é um classificador construído com base no teorema de bayess. (SETH; BANKA, 2016)

Quando aplicado a grandes datasets, classificadores bayseanos provem alta acurácia em alta velocidade. (SETH; BANKA, 2016), a forma mais simples de classificador bayseano conhecido atualmente é o Classificador naive bayes. (SETH; BANKA, 2016)

5.2 Classificador naive bayes

O teorema bayseano deriva da estatística bayseana, o qual possui premissas fortes e ingênuas (SURESH, 2016). O classificador bayseano é uma abordagem probabilística simples o qual assume que todos os atributos de um tupla pertencem a uma dada classe com uma probabilidade independente de ser classificado em relação a outros atributos (SURESH, 2016).

O teorema bayseano é uma afirmação estatística que permite o cálculo de probabilidades condicionais, probabilidades condicionais são probabilidades que refletem a influência de um evento na probabilidade de outro evento (SHARMA; SINGH; SINGH, 2015) .

Os termos geralmente utilizados em estatística bayseana são: probabilidade anterior e posterior: A probabilidade anterior é o evento com sua probabilidade original, antes de obter qualquer informação adicional em contraponto, a probabilidade posterior é a probabilidade anterior seguida de alguma informação adicional (SHARMA; SINGH; SINGH, 2015). O teorema de bayes pode ser descrito como (SHARMA; SINGH; SINGH, 2015):

$$P(A|B) = \frac{P(B|A).P(A)}{P(B)} \quad (4)$$

Onde:

$P(A)$ é a probabilidade anterior de A

$P(B)$ é a probabilidade anterior de B

$P(A|B)$ é a probabilidade posterior de A dado B

$P(B|A)$ é a probabilidade posterior de B dado A

O classificador naive bayes funciona da seguinte maneira (SETH; BANKA, 2016):

1. Inicialmente é necessário um conjunto de treinamento C com k características e classes associadas, usando o valor de atributos presente em cada tupla do dataset, caracterizado por um vetor $\alpha=(\alpha_1, \alpha_2, \alpha_3, \dots, \alpha_k)$;
2. A partir de um conjunto um conjunto de M classes beta compostos por $\gamma_1, \gamma_2, \gamma_3, \dots, \gamma_m$; O papel do classificador é prever a partir de uma tupla α qual a classe γ com maior probabilidade a qual esta tupla pertence dado o que foi aprendido com o conjunto de treinamento, ou seja, o classificador prevê se uma tupla α_i pertence a uma classe γ_i a partir da seguinte fórmula:

$$P(\gamma_i|\alpha) > P(\gamma_j|\alpha), \text{ para todo } 1 \leq j \leq m, j \neq i \quad (5)$$

Consequentemente, $P(\gamma_i|\alpha)$ deve ser maximizado, a classe resultante com o maior valor de $P(\gamma_i|\alpha)$ é chamada de probabilidade posterior, como no teorema de bayes. Dado o fato de que para todas as classes $P(\alpha)$ é constante, maximizar $P(\alpha|\gamma_i) * P(\gamma_i)$ garante o propósito (SHARMA; SINGH; SINGH, 2015).

Um conjunto de treinamento com muitos atributos para o cálculo de $P(\alpha)$ pode ser computacionalmente custoso de ser calculado, sendo assim, o classificador reduz este custo assumindo que as classes são condicionalmente independentes entre si (SHARMA; SINGH; SINGH, 2015).

5.3 Weka

A ferramenta Weka (Waikato Environment for Knowledge Learning) é um software desenvolvido pela universidade de Waikato na nova zelândia (SUBBULAKSHMI;

DEEPA; MALATHI, 2012). A WEKA suporta diferentes tipos de atividades como pré processamento, classificação, clusterização, regressão, visualização e seleção de atributos (SUBBULAKSHMI; DEEPA; MALATHI, 2012). A premissa da aplicação é utilizar-se de um computador e o software para executar técnicas de aprendizado de máquina e mineração de dados e derivar informações úteis através de padrões (SUBBULAKSHMI; DEEPA; MALATHI, 2012).

A ferramenta possui em torno de duas décadas, com implementações dos algoritmos de classificação com interfaces gráficas de fácil uso e visualização além de algoritmos de validação (BOUCKAERT et al., 2008).

A Weka opera com dados unidimensionais, permitindo que pessoas de qualquer nível de conhecimento em mineração de dados possam identificar informações escondidas em seus conjuntos de treinamentos com uma interface simples e direta (SUBBULAKSHMI; DEEPA; MALATHI, 2012). Após carregar e treinar um modelo com um conjunto de dados, o classificador pode ser validado para o conjunto e as classes fornecidas (SUBBULAKSHMI; DEEPA; MALATHI, 2012).

6 Resultados e Discussões

Neste capítulo serão apresentados os resultados obtidos por meio das informações encontradas e dos testes aplicados durante o desenvolvimento da pesquisa. Para efeito de organização, este capítulo será dividido em seções que explicam desde os materiais e métodos utilizados até os resultados obtidos, dados gerados, assim como as conclusões extraídas e trabalhos futuros.

6.1 Materiais

Durante o processo de pesquisa, foi utilizado um conjunto de 110 projetos os quais são reconhecidos e utilizados por diversos pesquisadores da área. Os projetos em questão fazem parte do corpus SF110, o qual é caracterizado por um conjunto de 110 projetos open source retirados do site SourceForge¹ compostos por: 100 projetos aleatoriamente escolhidos somados a 10 dos projetos com mais download em junho de 2014. (SHAMSHIRI et al.,).

O corpus SF110 foi utilizado como objeto de análise pelo processo de pesquisa por conta de sua diversidade de características, isso ocorre pois os projetos que compõe o corpus resolvem problemas presentes em domínios variados. A partir do corpus, diversas ferramentas foram utilizadas para extrair informações e processá-las, o uso destas será detalhada na seção de métodos, para este processos as seguintes ferramentas foram utilizadas

- **EvoSuite²**: ferramenta utilizada para geração dos dados de teste nas duas técnicas apresentadas no capítulo 2 e cálculo da porcentagem de cobertura de código atingida;
- **CK³**: ferramenta disponível em repositório opensource para extração de métricas CK em projetos Java;
- **Weka⁴**: ferramenta de mineração de dados utilizada para classificação pela técnica citada no capítulo 5;

¹ <https://sourceforge.net/>

² <http://www.evosuite.org/>

³ <https://github.com/mauricioaniche/ck>

⁴ <https://www.cs.waikato.ac.nz/ml/weka/>

- **LibreOffice Calc**⁵: ferramenta utilizada para o cálculo das correlações entre os valores que serão apresentados.
- **Go**⁶: linguagem de programação utilizada para codificação de algumas rotinas de processamento dos dados.
- **Maven**⁷: ferramenta de compilação, gestão de dependências e empacotamento dos projetos java presente no SF110.

Nas seções posteriores será descrita a metodologia aplicada durante o processo de pesquisa e os resultados encontrados, o processo foi segmentado nas tarefas de extração e análise das métricas, geração dos dados de teste, associação dos resultados, mineração dos dados, teste da hipótese e análise dos resultados.

6.2 Extrações de dados a análises primárias

A primeira etapa do processo consistiu na extração dos dados para análises primárias das hipóteses apresentadas, este passo foi necessário pois a primeira análise estatística para validação foi a análise da correlação das métricas CK dos projetos e os dados de execução fornecidos da EvoSuite: Cobertura de código e tempo de execução. A análise da correlação é uma etapa de validação importante visto que valores de correlação com pouca ou nenhuma significância podem dar indícios da eficácia de seu uso na mineração de dados, esta etapa caracteriza como uma etapa eliminatória das etapas posteriores do processo.

Os dados necessários para a análise da correlação são as devidas quantidades de cobertura, tempo disponibilizados pela EvoSuite e métricas CK extraídas das classes do SF110. Por conta da velocidade de extrair métricas estáticas como as métricas CK, a primeira coleta ocorreu com a geração dos dados de teste pela EvoSuite, inicialmente com o algoritmo de busca genético padrão e em seguida com o algoritmo de busca randômica.

O SF110 possui em torno de $\langle X \rangle$ classes java, espera-se então que o conjunto de dados consolidado final para treinamento possua o mesmo número, porém alguns dos projetos apresentaram problemas ao serem compilados, outros apresentaram mensagens de erro ao iniciar a geração de dados de testes utilizando algoritmos genéticos por meio da EvoSuite e alguns projetos possuíram tempo de execução superior a 8 horas.

⁵ <https://www.libreoffice.org/discover/calc/>

⁶ <https://golang.org/>

⁷ <https://maven.apache.org/>

Figura 6 – Distribuição de estado dos projetos após geração de dados de teste

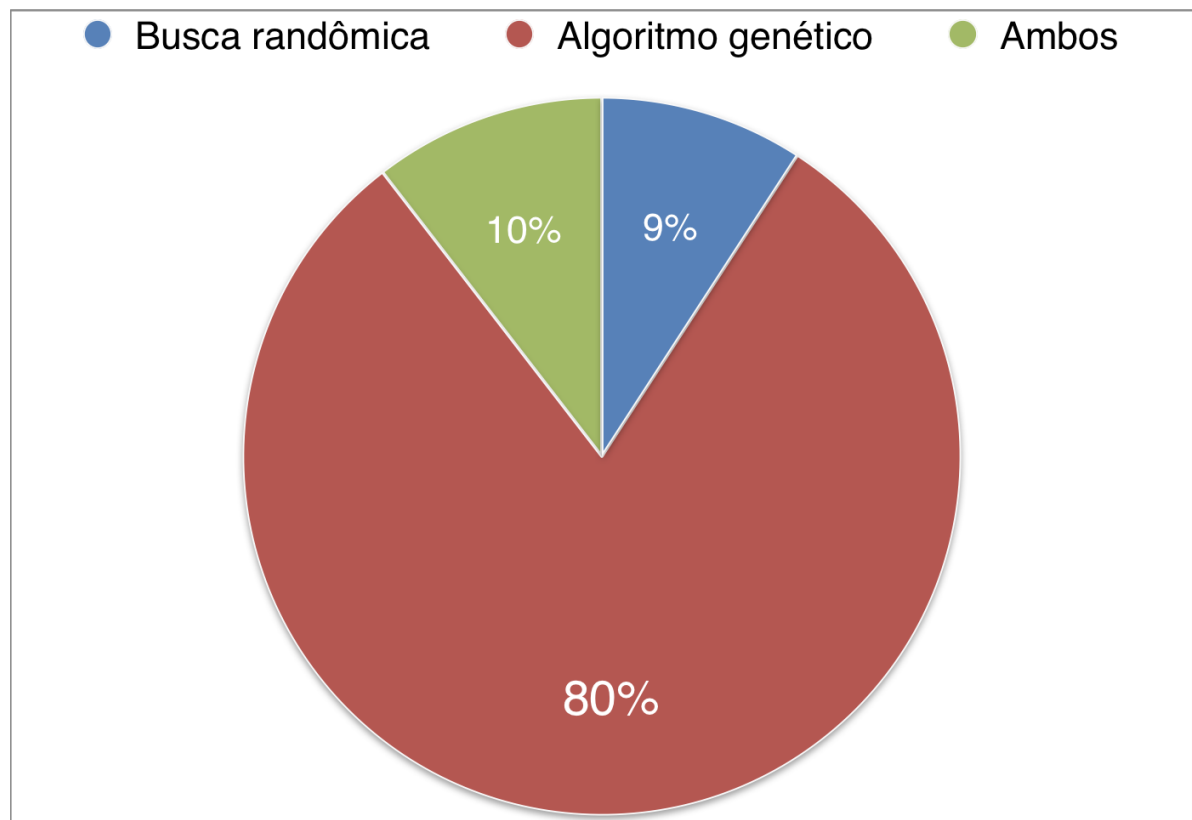


Fonte: Gustavo da Mota Ramos, 2018

Como demonstrado na figura 6, da quantidade total de projetos presentes no SF110, 75% tiveram seus dados de teste gerados corretamente, enquanto 8% apresentaram problemas ao serem compilados, 13% foram compilados corretamente porém apresentaram erros durante geração dos dados de teste e 4% dos projetos precisaram ter sua execução interrompida pois o processo já havia atingido 8 horas de duração para um único projeto.

Seguido à geração de dados de teste por meio de algoritmos genéticos, a geração de dados de teste por meio da técnica randômica ocorreu, porém esta ocorreu somente nos projetos com sucesso na geração anterior, isso ocorre por conta da comparação ser entre os dois algoritmos, então é necessário que ambos sejam gerados com base nos mesmos projetos.

Figura 7 – Quantidade de classes que atingiram cobertura de código superior ao outro algoritmo



Fonte: Gustavo da Mota Ramos, 2018

Após a coleta, o dataset contém a cobertura de código atingida até o tempo máximo registrado para cada classe em cada um dos algoritmos de geração descritos no capítulo [Capítulo 3](#), sendo assim foi possível verificar em qual dos algoritmos cada classe teve maior cobertura e quantificar, como demonstrado na figura 7.

Os resultados de cobertura de código obtidos divergem de alguns resultados apresentados por ([SHAMSHIRI et al.](#),), no experimento que consistiu em gerar os casos de testes a partir de 1000 classes randomicamente escolhidas do corpus SF110. O estudo em questão apontou que seus resultados sugerem que na prática existe pouca diferença quando os resultados de geração de casos de teste são comparados, concluindo que algoritmos randômicos tenham o mesmo nível de efetividade.

Esta diferença pode ser dada por dois fatores como volume de dados analisado e a evolução da ferramenta, o estudo citado utilizou em torno de 20% das classes utilizadas para análise na figura 7, o volume maior analisado pode representar uma quantidade maior de casos. Outro fator importante é a evolução da ferramenta EvoSuite ao longo de 3

anos anos, que através de melhorias constantes pode ter causado melhorias no algoritmo genético o suficiente para conseguir o aumento da cobertura de código demonstrado na diferença dos experimentos.

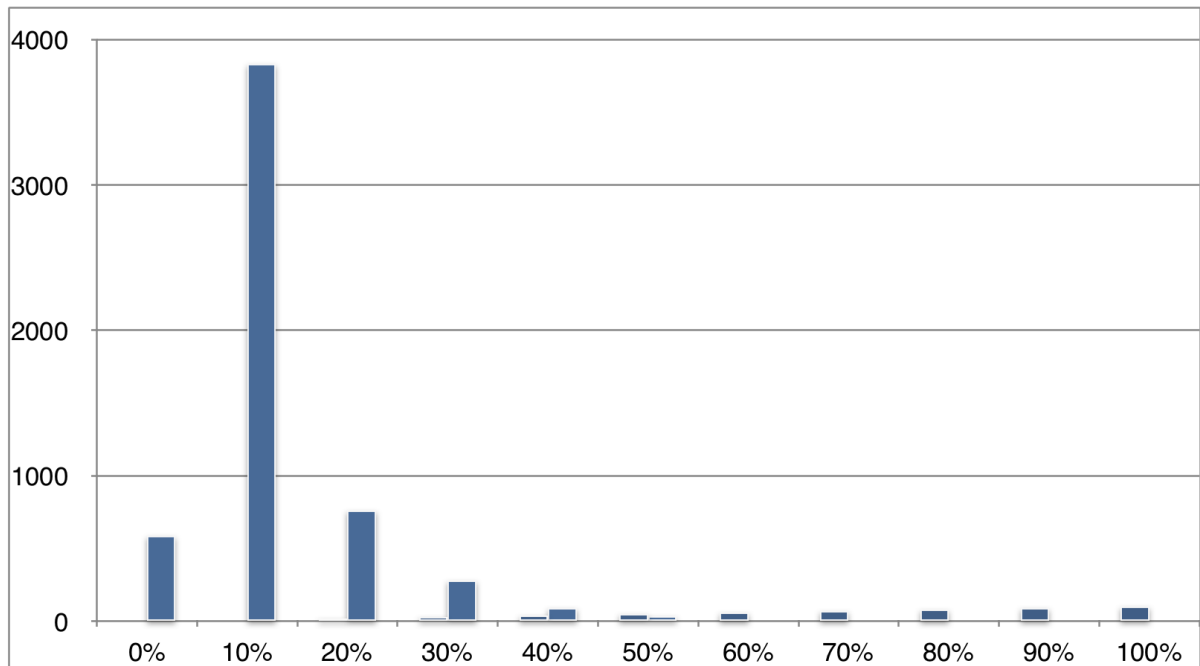
A partir dos dados de cobertura coletados em cada algoritmo, foi possível calcular a diferença entre eles em cada classe. As diferenças foram agrupadas e separadas em diferentes faixas como demonstrado na tabela a seguir:

Tabela 1 – Distribuição das classes nas diferenças de cobertura entre os algoritmos

Diferença de cobertura	Quantidade de classes
Sem diferença	585
10%	4232
20%	1075
30%	197
40%	74
50%	22
60%	1
70%	3
80%	0
90%	0
100%	2

O conteúdo da tabela 1 mostra a distribuição das diferenças de cobertura, é possível notar que maioria das classes, 85.7 % do total, possuem diferença de cobertura entre os algoritmos maior do que 0% e menor do que 20%. A diferença de cobertura entre 0% e 10% pode ser melhor vista na imagem 8.

Figura 8 – Quantidade de classes que consumiram menos tempo melhor em cada algoritmo

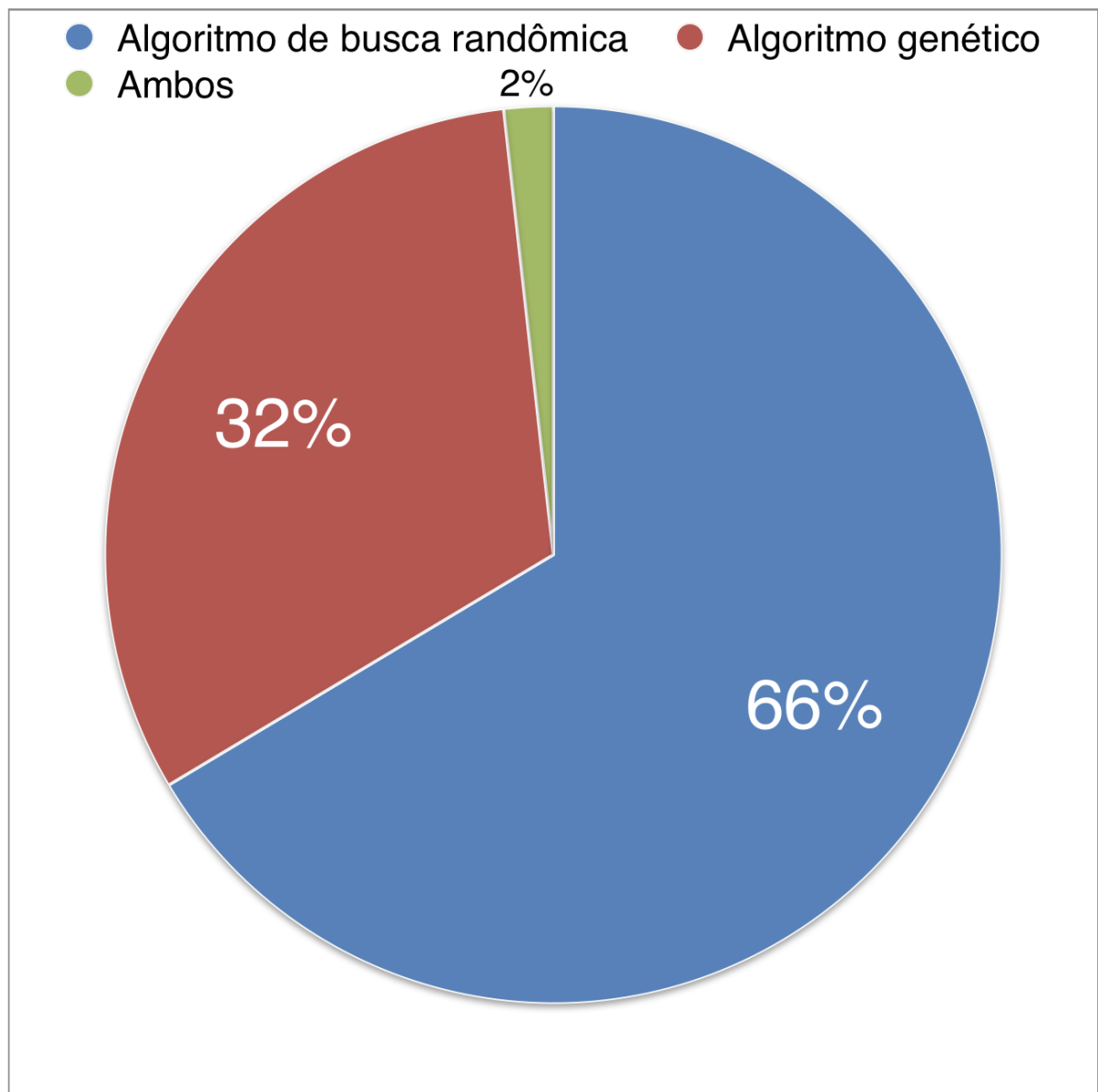


Fonte: Gustavo da Mota Ramos, 2018

A partir da imagem 8 é possível notar que a predominância da diferença de cobertura entre as técnicas situa-se entre 0% e 10% . Estes valores são importantes para compreender a magnitude da diferença coberta entre os algoritmos para que assim seja possível escolher o algoritmo mais rápido em casos que a diferença seja menor ao construir o conjunto de dados de aprendizado.

Um segundo ponto importante de análise é o tempo de geração utilizado em cada um dos algoritmos, em alguns dos projetos analisados, o tempo de geração total foi superior a 8 horas. o mesmo processo aplicado para análise da cobertura do código foi aplicado para o tempo de geração:

Figura 9 – Quantidade de classes que consumiram menos tempo melhor em cada algoritmo



Fonte: Gustavo da Mota Ramos, 2018

Como pode ser visto na imagem 9, uma quantidade considerável, totalizando 66% das classes foram geradas mais rapidamente utilizando o algoritmo de busca randômica, enquanto somente 32% das classes tiveram seus dados gerados mais rapidamente utilizando algoritmo genético e somente 2% tiveram seus dados gerados com a mesma quantidade de tempo.

Após a extração das métricas CK os dados foram consolidados por meio de executáveis escritos em Go, os arquivos de entrada consistem nos resultados gerados usando algoritmo genético, resultados gerados usando algoritmo randômico e as métricas CK. Os

3 arquivos diferem em suas estruturas, porém foram consolidados por meio de um conjunto de dados em comum pelos 3 arquivos: Todos possuem o caminho completo do pacote onde a classe encontra-se e o nome da classe, formando uma tupla única de união das três fontes de dados; O resultado final consistiu em um dataset de 5705 registros de classes contendo os dados consolidados, resultando assim no cálculo da correlação:

Tabela 2 – Correlação entre métricas CK e cobertura de código

Métrica	Algoritmo genético	Algoritmo randômico
CBO	-0.249850668595503	-0.232931214471127
WNC	-0.29807261754159	-0.276826266943643
DIT	-0.221073021178272	-0.229483879955976
NOC	0.000302617656848	0.001561246261131
RFC	-0.564049275058188	-0.552823956941685
LCOM	-0.040802466405277	-0.032051943296475
NOM	-0.109889786736834	-0.0827808149062
NOPM	-0.014135793315715	0.009596758553157
NOSM	-0.075230386707179	-0.071445190069526
NOF	-0.140224056429869	-0.120339077987188
NOPF	-0.004633572435886	-0.000497241219225
NOSF	-0.111343334357608	-0.109334015568744
NOSI	-0.282084913928777	-0.273109674212372
LOC	-0.287212833651501	-0.262779424522538

Tabela 3 – Correlação entre métricas CK e o tempo de geração

Métrica	Algoritmo genético	Algoritmo randômico
CBO	0.396318591856812	0.037898357322244
WNC	0.484002414944669	0.059460693763974
DIT	0.0040537265288	0.005185276134757
NOC	0.008046221302802	-0.00048066394787
RFC	0.491425142687552	0.073355181869394
LCOM	0.186475399697658	0.025181662291855
NOM	0.441803537967141	0.038466136152881
NOPM	0.38853663183873	0.027876177461111
NOSM	0.180912744473895	0.030067038091283
NOF	0.332907657997746	0.02963776551629
NOPF	0.059216533461886	0.002387027385852
NOSF	0.159268739166119	0.02786927036709
NOSI	0.283074420386565	0.042066832430523
LOC	0.476423297500777	0.052316363027409

6.3 Extrações de dados e análise das métricas SB4SE

Referências⁸

- BASHIR, M. F.; BANURI, S. H. K. Automated model based software test data generation system. In: *Emerging Technologies, 2008. ICET 2008. 4th International Conference on*. [S.l.: s.n.], 2008. p. 275–279. Citado na página 13.
- BINDER, R. V. Design for testability in object-oriented systems. *Commun. ACM*, ACM, New York, NY, USA, v. 37, n. 9, p. 87–101, set. 1994. ISSN 0001-0782. Disponível em: <<http://doi.acm.org/10.1145/182987.184077>>. Citado na página 13.
- BOUCKAERT, R. R. et al. Weka manual for version 3-6-0. 12 2008. Citado na página 42.
- BRUNTINK, M.; DEURSEN, A. van. Predicting class testability using object-oriented metrics. In: *Source Code Analysis and Manipulation, Fourth IEEE International Workshop on*. [S.l.: s.n.], 2004. p. 136–145. Citado 2 vezes nas páginas 34 e 35.
- CADAR, C.; DUNBAR, D.; ENGLER, D. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. Berkeley, CA, USA: USENIX Association, 2008. (OSDI'08), p. 209–224. Disponível em: <<http://dl.acm.org/citation.cfm?id=1855741.1855756>>. Citado na página 13.
- CADAR, C.; SEN, K. Symbolic execution for software testing: Three decades later. *Commun. ACM*, ACM, New York, NY, USA, v. 56, n. 2, p. 82–90, fev. 2013. ISSN 0001-0782. Disponível em: <<http://doi.acm.org/10.1145/2408776.2408795>>. Citado na página 13.
- CAMPOS, J. et al. An empirical evaluation of evolutionary algorithms for test suite generation. In: MENZIES, T.; PETKE, J. (Ed.). *Search Based Software Engineering*. Cham: Springer International Publishing, 2017. p. 33–48. ISBN 978-3-319-66299-2. Citado 8 vezes nas páginas 14, 24, 25, 26, 27, 28, 29 e 30.
- CLASSPATH, JAVA docs. <<https://docs.oracle.com/javase/8/docs/technotes/tools/windows/classpath.html>>. Acessado em: 2018-08-04. Citado na página 30.
- DAVE, M.; AGRAWAL, R. Search based techniques and mutation analysis in automatic test case generation: A survey. In: *2015 IEEE International Advance Computing Conference (IACC)*. [S.l.: s.n.], 2015. p. 795–799. Citado na página 24.
- DAVIS, M. J. Process and product: Dichotomy or duality? *SIGSOFT Softw. Eng. Notes*, ACM, New York, NY, USA, v. 20, n. 2, p. 17–18, abr. 1995. ISSN 0163-5948. Disponível em: <<http://doi.acm.org/10.1145/224155.565634>>. Citado na página 17.
- DICK, J.; FAIVRE, A. Automating the generation and sequencing of test cases from model-based specifications. In: WOODCOCK, J. C. P.; LARSEN, P. G. (Ed.). *FME '93: Industrial-Strength Formal Methods*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993. p. 268–284. Citado na página 13.
- ELER, M. M.; ENDO, A. T.; DURELLI, V. H. S. An empirical study to quantify the characteristics of java programs that may influence symbolic execution from a unit testing perspective. *Journal of Systems and Software*, v. 121, p. 281–297, 2016. Citado na página 37.

⁸ De acordo com a Associação Brasileira de Normas Técnicas. NBR 6023.

EVOSUITE.ORG. <<http://www.evosuite.org/>>. Acessado em: 2018-07-01. Citado na página 30.

FRASER, G.; ARCURI, A. Evosuite: Automatic test suite generation for object-oriented software. In: *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. New York, NY, USA: ACM, 2011. (ESEC/FSE '11), p. 416–419. ISBN 978-1-4503-0443-6. Disponível em: <<http://doi.acm.org/10.1145/2025113.2025179>>. Citado 2 vezes nas páginas 13 e 14.

FRASER, G.; ARCURI, A. Whole test suite generation. *IEEE Trans. Softw. Eng.*, IEEE Press, Piscataway, NJ, USA, v. 39, n. 2, p. 276–291, fev. 2013. ISSN 0098-5589. Disponível em: <<http://dx.doi.org/10.1109/TSE.2012.14>>. Citado na página 14.

FRASER, G.; ARCURI, A. A large-scale evaluation of automated unit test generation using evosuite. *ACM Trans. Softw. Eng. Methodol.*, ACM, New York, NY, USA, v. 24, n. 2, p. 8:1–8:42, dez. 2014. ISSN 1049-331X. Disponível em: <<http://doi.acm.org/10.1145/2685612>>. Citado na página 14.

FRASER, G. et al. Evosuite at the sbst 2017 tool competition. In: *Proceedings of the 10th International Workshop on Search-Based Software Testing*. Piscataway, NJ, USA: IEEE Press, 2017. (SBST '17), p. 39–41. ISBN 978-1-5386-2789-1. Disponível em: <<https://doi.org/10.1109/SBST.2017.6>>. Citado 3 vezes nas páginas 14, 30 e 31.

FRASER, G. et al. Does automated unit test generation really help software testers? a controlled empirical study. *ACM Trans. Softw. Eng. Methodol.*, ACM, New York, NY, USA, v. 24, n. 4, p. 23:1–23:49, set. 2015. ISSN 1049-331X. Disponível em: <<http://doi.acm.org/10.1145/2699688>>. Citado na página 14.

GODEFROID, P.; KLARLUND, N.; SEN, K. Dart: Directed automated random testing. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 40, n. 6, p. 213–223, jun. 2005. ISSN 0362-1340. Disponível em: <<http://doi.acm.org/10.1145/1064978.1065036>>. Citado na página 13.

GOODENOUGH, J. B.; GERHART, S. L. Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, SE-1, n. 2, p. 156–173, June 1975. ISSN 0098-5589. Citado na página 17.

GRAHAM, D.; VEENENDAAL, E. V.; EVANS, I. *Foundations of Software Testing: ISTQB Certification*. Cengage Learning, 2008. ISBN 9781844809899. Disponível em: <<https://books.google.ie/books?id=Ss62LSqCa1MC>>. Citado na página 13.

HARMAN, M.; MANSOURI, S. A.; ZHANG, Y. Search-based software engineering: Trends, techniques and applications. *ACM Comput. Surv.*, ACM, New York, NY, USA, v. 45, n. 1, p. 11:1–11:61, dez. 2012. ISSN 0360-0300. Disponível em: <<http://doi.acm.org/10.1145/2379776.2379787>>. Citado na página 13.

HILBURN, Y.; TOWHIDNEJAD, M. Software quality across the curriculum. In: IEEE. *Frontiers in Education, 2002. FIE 2002. 32nd Annual*. [S.l.], 2002. v. 3, p. S1G–18. Citado na página 13.

JACOB, P. M.; PRASANNA, M. A comparative analysis on black box testing strategies. In: *2016 International Conference on Information Science (ICIS)*. [S.l.: s.n.], 2016. p. 1–6. Citado 3 vezes nas páginas 18, 20 e 21.

JUNIT.ORG/JUNIT5/. <<https://junit.org/junit5/>>. Acessado em: 2018-08-04. Citado na página 30.

KAN, S. H. *Metrics and Models in Software Quality Engineering*. Reading, MA: Addison Wesley, 1995. Citado 5 vezes nas páginas 32, 33, 34, 35 e 36.

KAPROCKI, Z.; PEKOVIC, V.; VELIKIC, G. Combined testing approach: Increased efficiency of black box testing. In: *2015 IEEE 1st International Workshop on Consumer Electronics (CE WS)*. [S.l.: s.n.], 2015. p. 76–78. Citado na página 19.

KARNOPP, D. C. Random search techniques for optimization problems. *Automatica*, Pergamon Press, Inc., Tarrytown, NY, USA, v. 1, n. 2-3, p. 111–121, ago. 1963. ISSN 0005-1098. Disponível em: <[http://dx.doi.org/10.1016/0005-1098\(63\)90018-9](http://dx.doi.org/10.1016/0005-1098(63)90018-9)>. Citado 2 vezes nas páginas 25 e 26.

KOREL, B. Automated software test data generation. *IEEE Trans. Softw. Eng.*, IEEE Press, Piscataway, NJ, USA, v. 16, n. 8, p. 870–879, ago. 1990. ISSN 0098-5589. Disponível em: <<http://dx.doi.org/10.1109/32.57624>>. Citado 2 vezes nas páginas 21 e 22.

MCMINN, P. Search-based software test data generation: A survey: Research articles. *Softw. Test. Verif. Reliab.*, John Wiley and Sons Ltd., Chichester, UK, v. 14, n. 2, p. 105–156, jun. 2004. ISSN 0960-0833. Disponível em: <<http://dx.doi.org/10.1002/stvr.v14:2>>. Citado na página 13.

NORTHROP, L. M. Let's teach architecting high quality software. In: *19th Conference on Software Engineering Education and Training (CSEET 2006)*, 19-21 April 2006, Turtle Bay, Hawaii, USA. [s.n.], 2006. p. 5. Disponível em: <<http://dx.doi.org/10.1109/CSEET.2006.23>>. Citado na página 13.

PACHECO, C.; ERNST, M. D. Randoop: Feedback-directed random testing for java. In: *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA*. [S.l.: s.n.], 2007. p. 815–816. Citado na página 13.

PEZZÈ, M.; YOUNG, M. *Software testing and analysis: process, principles, and techniques*. Wiley, 2008. ISBN 9780471455936. Disponível em: <<https://books.google.com.br/books?id=mjEiAQAIAAJ>>. Citado na página 13.

PFLEEGER, S.; ATLEE, J. *Software Engineering: Theory and Practice*. Prentice Hall, 2010. ISBN 9780136061694. Disponível em: <<https://books.google.com.br/books?id=7zbSZ54JG1wC>>. Citado na página 17.

PRESSMAN, R. *Engenharia de Software - 7.ed.*. McGraw Hill Brasil, 2009. ISBN 9788580550443. Disponível em: <<https://books.google.com.br/books?id=y0rH9wuXe68C>>. Citado 5 vezes nas páginas 17, 21, 22, 23 e 36.

REPASI, T. Software testing - state of the art and current research challenges. In: *2009 5th International Symposium on Applied Computational Intelligence and Informatics*. [S.l.: s.n.], 2009. p. 47–50. Citado na página 21.

ROJAS, J. M.; FRASER, G. Is search-based unit test generation research stuck in a local optimum? In: *2017 IEEE ACM 10th International Workshop on Search-Based Software Testing (SBST)*. [S.l.: s.n.], 2017. p. 51–52. Citado na página 25.

ROJAS, J. M. et al. A detailed investigation of the effectiveness of whole test suite generation. *Empirical Software Engineering*, v. 22, n. 2, p. 852–893, Apr 2017. Disponível em: <<https://doi.org/10.1007/s10664-015-9424-2>>. Citado na página 14.

SEN, K.; AGHA, G. Cute and jcute: Concolic unit testing and explicit path model-checking tools. In: BALL, T.; JONES, R. B. (Ed.). *Computer Aided Verification*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006. p. 419–423. ISBN 978-3-540-37411-4. Citado na página 13.

SEN, K.; MARINOV, D.; AGHA, G. Cute: A concolic unit testing engine for c. *SIGSOFT Softw. Eng. Notes*, ACM, New York, NY, USA, v. 30, n. 5, p. 263–272, set. 2005. ISSN 0163-5948. Disponível em: <<http://doi.acm.org/10.1145/1095430.1081750>>. Citado na página 13.

SETH, H. R.; BANKA, H. Hardware implementation of naïve bayes classifier: A cost effective technique. In: *2016 3rd International Conference on Recent Advances in Information Technology (RAIT)*. [S.l.: s.n.], 2016. p. 264–267. Citado 3 vezes nas páginas 39, 40 e 41.

SHAMSHIRI, S. et al. How do automatically generated unit tests influence software maintenance? In: *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. [S.l.: s.n.], 2018. p. 250–261. Citado na página 24.

SHAMSHIRI, S. et al. Random or evolutionary search for object-oriented test suite generation? *Software Testing, Verification and Reliability*, v. 28, n. 4, p. e1660. E1660 stvr.1660. Disponível em: <<https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1660>>. Citado 6 vezes nas páginas 24, 26, 27, 30, 43 e 46.

SHARMA, P.; SINGH, D.; SINGH, A. Classification algorithms on a large continuous random dataset using rapid miner tool. In: *2015 2nd International Conference on Electronics and Communication Systems (ICECS)*. [S.l.: s.n.], 2015. p. 704–709. Citado 3 vezes nas páginas 39, 40 e 41.

SINGH, R. Test case generation for object-oriented systems: A review. In: *2014 Fourth International Conference on Communication Systems and Network Technologies*. [S.l.: s.n.], 2014. p. 981–989. Citado na página 18.

SITE.MOCKITO.ORG. <<http://site.mockito.org>>. Acessado em: 2018-08-04. Citado na página 31.

SLAUGHTER, S. A.; HARTER, D. E.; KRISHNAN, M. S. Evaluating the cost of software quality. *Commun. ACM*, ACM, New York, NY, USA, v. 41, n. 8, p. 67–73, ago. 1998. ISSN 0001-0782. Disponível em: <<http://doi.acm.org/10.1145/280324.280335>>. Citado na página 22.

SOMMERVILLE, I. *Software Engineering*. 9th. ed. USA: Addison-Wesley Publishing Company, 2010. ISBN 0137035152, 9780137035151. Citado 2 vezes nas páginas 17 e 18.

SOMMERVILLE, I. et al. *Engenharia de software*. ADDISON WESLEY BRA, 2008. ISBN 9788588639287. Disponível em: <<https://books.google.com.br/books?id=ifIYOgAACAAJ>>. Citado na página 13.

SPADINI, D. et al. To mock or not to mock? an empirical study on mocking practices. In: *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. [S.l.: s.n.], 2017. p. 402–412. Citado 2 vezes nas páginas 18 e 19.

SRIVASTAVA, S.; KUMAR, R. Indirect method to measure software quality using ck-oo suite. In: *2013 International Conference on Intelligent Systems and Signal Processing (ISSP)*. [S.l.: s.n.], 2013. p. 47–51. Citado 4 vezes nas páginas 32, 33, 36 e 37.

SUBBULAKSHMI, C. V.; DEEPA, S. N.; MALATHI, N. Comparative analysis of xlminer and weka for pattern classification. In: *2012 IEEE International Conference on Advanced Communication Control and Computing Technologies (ICACCCT)*. [S.l.: s.n.], 2012. p. 453–457. Citado na página 42.

SURESH, Y. Software quality assessment for open source software using logistic amp; naive bayes classifier. In: *2016 International Conference on Computation System and Information Technology for Sustainable Solutions (CSITSS)*. [S.l.: s.n.], 2016. p. 267–272. Citado 2 vezes nas páginas 37 e 40.

TAHIR, A.; MACDONELL, S. G.; BUCHAN, J. Understanding class-level testability through dynamic analysis. In: *Evaluation of Novel Approaches to Software Engineering (ENASE), 2014 International Conference on*. [S.l.: s.n.], 2014. p. 1–10. Citado na página 13.

TILLMANN, N.; HALLEUX, J. de. Pex–white box test generation for .net. In: BECKERT, B.; HÄHNLE, R. (Ed.). *Tests and Proofs*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. p. 134–153. ISBN 978-3-540-79124-9. Citado na página 13.

VISSER, W.; PĂSĂREANU, C. S.; KHURSHID, S. Test input generation with java pathfinder. *SIGSOFT Softw. Eng. Notes*, ACM, New York, NY, USA, v. 29, n. 4, p. 97–107, jul. 2004. ISSN 0163-5948. Disponível em: <<http://doi.acm.org/10.1145/1013886.1007526>>. Citado na página 13.

WAGNER, S.; SEIFERT, T. Software quality economics for defect-detection techniques using failure prediction. *SIGSOFT Softw. Eng. Notes*, ACM, New York, NY, USA, v. 30, n. 4, p. 1–6, maio 2005. ISSN 0163-5948. Disponível em: <<http://doi.acm.org/10.1145/1082983.1083296>>. Citado 3 vezes nas páginas 21, 22 e 23.

WALKINSHAW, N.; FRASER, G. Uncertainty-driven black-box test data generation. In: *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. [S.l.: s.n.], 2017. p. 253–263. Citado na página 20.

WATSON, A. H.; MCCABE, T. J.; WALLACE, D. R. Special publication 500-235, structured testing: A software testing methodology using the cyclomatic complexity metric. In: *U.S. Department of Commerce/National Institute of Standards and Technology*. [S.l.: s.n.], 1996. Citado na página 33.

XU, S. et al. A comparative study on black-box testing with open source applications. In: *2016 17th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*. [S.l.: s.n.], 2016. p. 527–532. Citado na página 20.

ZEISS, B. et al. Applying the iso 9126 quality model to test specifications - exemplified for ttcn-3 test specifications. In: *Software Engineering*. [S.l.: s.n.], 2007. Citado na página 32.