RAFAEL BALDIATI PARIZI

PRIORIZAÇÃO DE CLASSES PARA TESTES UNITÁRIOS BASEADA EM MULTICRITÉRIOS

Alegrete - RS

2010

RAFAEL BALDIATI PARIZI

PRIORIZAÇÃO DE CLASSES PARA TESTES UNITÁRIOS BASEADA EM MULTICRITÉRIOS

Monografia apresentada para obtenção do Grau de Bacharel em Ciência da Computação pela Universidade Federal do Pampa.

Orientador: Profa.Ms. Vanessa Gindri Vieira

Co-orientador: Prof.Dr. Eduardo Kessler Piveta

Alegrete - RS

2010

RAFAEL BALDIATI PARIZI

PRIORIZAÇÃO DE CLASSES PARA TESTES UNITÁRIOS BASEADA EM MULTICRITÉRIOS

Monografia apresentada ao Curso de Graduação em Ciência da Computação da Universidade Federal do Pampa, como requisito parcial para obtenção do Título de Bacharel em Ciência da Computação.

Área de concentração: Ciências Exatas e da Terra

Monografia defendida e aprovada em: 12 de Julho de 2010.

Banca examinadora:

Profa. Ms. Vanessa Gindri Vieira
Orientador
Ciência da Computação - Unipampa

Profa. Dra. Amanda Meincke Melo
Ciência da Computação - Unipampa

Prof. Ms. Marcelo Cezar Pinto
Ciência da Computação - Unipampa

Dedico esta monografia a meus familiares, colegas e professores, pelo apoio e incentivo durante o período de graduação.

AGRADECIMENTO

Meu agradecimento especial a Deus que me deu inspiração e coragem para superar todas as dificuldades durante o período de graduação. Aos meus familiares e amigos, que me incentivaram e me apoiaram em todos os momentos, em especial à minha mãe que me apoiou de forma incondicional nos momentos mais difíceis da minha vida.

Meu agradecimento especial à minha orientadora, Vanessa Gindri Vieira, e ao meu Co-Orientador, Eduardo Kessler Piveta, por terem acreditado no meu potencial e me ajudado com toda sua experiência e sabedoria, e também compreendido as dificuldades que tiveram de ser superadas neste período.

Finalmente, meu agradecimento aos meus colegas, ex-colegas e professores do curso, por terem dividido os momentos de estudos, os conhecimentos e a amizade durante a graduação.

RESUMO

A sociedade atual vem usando cada vez mais sistemas que auxiliem de alguma forma na realização das tarefas do dia-a-dia. Isso proporciona um crescimento no uso de software pelas pessoas, provocando um aumento na necessidade de estabelecer qualidade nos sistemas desenvolvidos. Essa qualidade pode ser obtida através da aplicação de boas técnicas de teste ao software, garantindo que os sistemas atendam as necessidades do cliente livres de erros e defeitos.

Estudos relacionados a testes de software estão em amplo crescimento no mundo acadêmico e profissional. Estes estudos buscam melhorias na etapa de testes com o objetivo de aumentar a eficácia e a garantia dos softwares desenvolvidos. Os testes em sistemas orientados a objetos estão entre os mais desenvolvidos e estudados pelo fato das linguagens de programação orientadas a objeto serem muito usadas pelos desenvolvedores de software.

A priorização de testes é fundamental para empresas que possuem pouco tempo para o desenvolvimento, dado que essa etapa pode ser a mais custosa no processo. Essa priorização pode ser baseada em multicritérios necessitando de um método de apoio a tomada de decisão para encontrar uma solução. Muitas ferramentas dão suporte à fase de teste, automatizando-a e tornando-a mais confiável.

Essa monografia apresenta conceitos relacionados a testes e análise de métricas de software, como extrai-las de um código, como avaliá-las através do método AHP (*Analytic Hierarchic Process*), uma proposta de processo de priorização de classes para testes de unidade de sistemas orientados a objetos, uma ferramenta para a realização de forma automatizada deste processo e um estudo de caso como forma de avaliação do trabalho. Dessa forma e, pelo contexto apresentado, este trabalho contém uma alternativa ainda não explorada para a priorização de classes de software com objetivo de guiar a fase de testes.

Palavras-chave: Classes; Métricas; Priorização; Testes de Software.

ABSTRACT

Contemporary society has increasingly using systems that help in some way in performing daily tasks. This use provides a growth in the use of software systems, increasing the need of high-quality software systems. This quality can be obtained by applying proven techniques to software testing, aiming to increase the confidence that the software systems meet the customer needs and do not present a high-rate of defects.

There are several studies related to software testing in academia and in industry. Such studies seek for improvements in benchmark testing with the goal of increasing the efficiency and the security of the developed software systems. Tests on object-oriented systems are among the most researched topics on software engineering because the very used programming languages are object-oriented.

The prioritization of tests is critical for companies who wants to optimize their development, given that this step can be one of the most expensives in the hole software development process. This prioritization can be based on a multi-criteria basis, using a decision method for finding a good solution. Several tools support the testing activities by automating them and making them more reliable.

This research presents concepts related to: testing and analysis of software metrics, how to compute them, how to evaluate them through the AHP (Analytic Hierarchic Process), a proposed process of prioritizing classes for unit testing object-oriented systems, a tool to automate this process, and a case study as a way of evaluation.

Key-words: Classes; Metrics; Prioritization; Software Testing.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 Introdução

Sistemas de software têm se tornando indispensáveis para mais pessoas a cada dia que passa. A competitividade imposta pelo mercado pode ser um dos fatores que tem elevado o uso de sistemas informatizados pelas empresas. O nível de concorrência que ocorre hoje no comércio exige que empresários lancem mão de novas tecnologias a fim de garantir seus clientes. Outro fator que aumenta o uso de software pela sociedade é a facilidade de comunicação com outras pessoas através do uso de videoconferências, *chats*, mensageiros eletrônicos, além da realização de tarefas a distância.

O aumento na taxa de uso de software, a competitividade existente entre as empresas desenvolvedoras, as exigências dos clientes por produtos qualificados, são fatores que fazem crescer a necessidade de estabelecer qualidade nos sistemas desenvolvidos (PRESSMAN, 2006). Essa garantia de qualidade é de responsabilidade da empresa que desenvolve o software, podendo garanti-la através do estabelecimento de modelos de desenvolvimento e políticas de garantia de qualidade que asseguram que o software desenvolvido atenderá de forma satisfatória ao propósito para o qual foi criado.

Os processos de desenvolvimento comumente utilizados como o UP (*Unified process*), RUP (*Rational Unified Process*), XP (*eXtreme Programming*), servem como guia para o processo de desenvolvimento de software (CARVALHO *et al.*, 2008), (BOENTE *et al.*, 2008). Geralmente eles definem etapas que podem variar de modelo para modelo, mas o objetivo principal é poder organizar o desenvolvimento de software em busca de maior qualidade, de forma que esta possa ser transferida para o software entregue ao cliente. Em processos de desenvolvimento com qualidade é mais fácil de garantir a qualidade do produto final (PRESSMAN, 2006).

Dentro do processo de desenvolvimento, as atividades de teste são aquelas que possuem maior ênfase na verificação da qualidade. Essas atividades são responsáveis por verificar se o sistema não possui erros validando-o para poder ser entregue ao cliente. Os testes são classificados em diversos tipos e podem sofrer alterações em sua aplicação dependendo do paradigma de programação que é usado no desenvolvimento (SOMMERVILLE, 2007). Ao ser feito o uso do

paradigma orientado a objetos para desenvolver uma aplicação, por exemplo, devem ser aplicados: teste de unidade, responsável por testar as classes que compõem essa aplicação; teste de integração, que testa se as classes funcionam corretamente quando combinadas; teste de sistema, que verifica se o sistema como um todo funciona corretamente e se atende aos requisitos, e por fim, o teste de validação, que é equivalente ao teste de sistema mas aplicado pelo usuário para verificar se o sistema está funcionando corretamente.

Sistemas precisam ser concebidos em um curto espaço de tempo, o que pode afetar o processo de desenvolvimento e consequentemente a qualidade do sistema criado. Esse problema é o responsável pelo estabelecimento de várias pesquisas por parte dos engenheiros de software com o intuito de buscar metodologias, criar ferramentas, novas abordagens de desenvolvimento, que possam melhorar e agilizar o processo de criação de software. Por exemplo, a XP é uma metodologia ágil de desenvolvimento que possui uma coleção de princípios e valores simples, focados em princípios como melhorar a interação entre os membros da equipe, a entrega de software funcional de forma iterativa e agregação de valor ao cliente.

Devido à implicação do imediatismo sofrido pelas empresas nas etapas do desenvolvimento, muitas delas são realizadas de forma superficial o que pode causar muitos problemas à empresa e ao produto. A etapa de teste é uma das etapas que mais sofre com o curto tempo de desenvolvimento. Nesse cenário os testes aplicados ao software são testes inadequados e insuficientes para a garantia de que o sistema não será entregue com erros ao cliente. Por esse motivo, estudos são realizados para encontrar alternativas de teste que possam automatizar essa que é uma das tarefas mais custosas no desenvolvimento de sistemas, chegando, em muitos casos, a 50% do custo total do software (TAHAT *et al.*, 2001).

Para automatizar a etapa de testes muitos trabalhos sugerem a priorização dos testes, a fim de aplicar primeiro os testes que são mais importantes (TAHAT *et al.*, 2001; LOPES e FERNANDES, 2007; OLIVEIRA *et al.*, 2007; BERNARDO e KON, 2008). Essa importância pode ser obtida através da análise de vários fatores e critérios, tornando-se mais um problema multicritério, pertencente a uma classe de problemas amplamente analisada atualmente. Em software esses critérios podem ser estabelecidos através das métricas de software que realizam a mensuração dos atributos de qualidade, complexidade e tamanho dos sistemas.

Esse trabalho visa propor uma abordagem para a priorização de classes para aplicação dos testes usando métodos de decisão multicritérios, que são métodos que auxiliam informando qual a melhor escolha dentre um conjunto de possíveis alternativas.

Os benefícios que são esperados com a realização deste trabalho têm relação com a melhoria no processo de desenvolvimento de software mais precisamente na fase de testes de software.

Também se espera que o trabalho possa contribuir para auxiliar os responsáveis pelos testes em focar os seus esforços em casos de teste com mais importância para o projeto em que estão participando. Através da ferramenta desenvolvida por este trabalho, inédita na literatura no que diz respeito à priorização de classes, espera-se automatizar e facilitar a execução do processo de priorização.

Este trabalho é composto dos seguintes capítulos: o Capítulo 2 apresenta uma revisão bibliográfica sobre priorização de testes, tema deste trabalho. O Capítulo 3 apresenta conceitos relacionados a testes de software voltados ao paradigma de orientação a objetos e também ferramentas que auxiliam o processo de teste. No Capítulo 4 são apresentadas as métricas de software e as ferramentas que dão o suporte para a extração dessas métricas. Já o Capítulo 5 compreende conceitos sobre métodos de apoio a tomada de decisão baseados em multicritérios, mais especificamente o método AHP. O Capítulo 6 aborda o processo de priorização de classes para testes de software, detalhando as possíveis etapas para a obtenção de uma ordem de classes, a descrição da ferramenta proposta para a execução do processo de priorização e um estudo de caso como forma de avaliação do trabalho. Na sequência, são apresentadas as considerações finais juntamente com os trabalhos futuros, as referências bibliográficas que deram suporte ao desenvolvimento deste trabalho e para finalizar, dois Apêndices, o primeiro contendo outro estudo de caso e o segundo uma descrição e exemplo de implementação do documento gerado pela ferramenta proposta.

2 *Revisão Bibliográfica*

Estudos na área de engenharia de software, mais especificamente na área de testes têm crescido devido a necessidade de encontrar alternativas para essa fase do processo de desenvolvimento, por ser uma das mais importantes e de maior custo, além de requerer grandes esforços (SOMMERVILLE, 2007).

Atualmente, pesquisas em relação à construção de ferramentas de automatização do processo de teste estão ganhando força e novos adeptos. Essas pesquisas são importantes, pois com o nível de complexidade e de tamanho que os sistemas possuem hoje é praticamente impossível realizar a aplicação de testes e garantir qualidade de forma manual (BERNARDO e KON, 2008).

Além dos estudos dirigidos à construção de ferramentas de automação de testes, outra frente que ganha força é a constituída pelas pesquisas relacionadas a priorização dos testes baseada em alguns critérios. A priorização dos testes visa estabelecer uma ordem para a aplicação dos testes com o objetivo de diminuir os esforços necessários para a realização dos mesmos. Essa priorização pode ser baseada em análise dos inúmeros critérios que cercam os componentes de software, por exemplo, as métricas de software.

Este capítulo tem por objetivo apresentar os estudos realizados com relação à priorização de testes de software.

2.1 Trabalhos relacionados com a priorização de teste

A UML (*Unified Modeling Language* - Linguagem de Modelagem Unificada) tem sido amplamente utilizada nas últimas décadas pelos engenheiros de software para a representação, através de diversos diagramas, da estrutura do projeto. Alguns trabalhos fazem uso da UML para a seleção e geração de casos de teste (LIMA *et al.*, 2007; KIM *et al.*, 1999), os quais realizam a geração de teste através da análise de diagramas de estado e máquinas de estado.

Os testes de integração são muito importantes em um software. Eles são responsáveis por

verificar se os componentes conseguem trabalhar de forma conjunta, integrada. Um estudo é apresentado por Lima e Travassos (2004) com o objetivo de ordenar classes para poder realizar testes de integração com um menor esforço. Nesse trabalho os autores tentam minimizar a construção de *stubs*, que são componentes que simulam classes que ainda não foram testadas, mas que são usadas por outras classes que estão sendo testadas. Os autores fazem uso de heurísticas para a ordenação das classes aplicadas em diagramas de classes UML, possibilitando a criação de uma ordem de prioridade.

Outra vertente de estudos de engenharia de software é a priorização dos testes de regressão. Esse tipo de teste é aplicado após mudanças realizadas no sistema, a fim de garantir que ele se comporta como pretendido e que as modificações não causaram nenhum impacto adverso na qualidade do software. O trabalho de Srivastava (2008) apresenta um novo algoritmo e sua eficácia para a priorização de casos de teste baseado na taxa de falhas por minuto, encontrada no sistema. Outro trabalho referenciado na priorização de testes de regressão é o de Rothermel *et. al.* (2001), no qual os autores apresentam um estudo realizado com diversas técnicas de obtenção de falhas com o objetivo de melhorar a fase de testes de regressão. Existem ainda, vários trabalhos relacionados com priorização de testes de regressão que podem ser encontrados facilmente na literatura (CRESPO *et al.*, 2004; ROTHERMEL *et al.*, 2001; SRIVASTAVA, 2008; SRIKANTH e WILLIAMS, 2005; SRIKANTH *et al.*, 2005).

Uma das linguagens de programação mais utilizadas atualmente é Java, que segue os princípios de orientação a objetos, proporcionando esforços em estudos para melhorias no desenvolvimento de software, consequentemente para a fase de testes, de sistemas que utilizam essa linguagem. Um estudo de antecipação de classes de testes é Bruntink(2004), cujo objetivo é testar primeiramente classes que possuem uma prioridade em relação a métricas de software orientadas a objeto. Essa prioridade dá-se através da avaliação das métricas com a ferramenta GQM/MEDEA (BRIAND *et al.*, 2002).

O próximo capítulo apresenta conceitos relacionados a testes de software, salientando os testes para sistemas orientados a objeto, além de metodologias de automação dos testes.

3 *Testes de Software*

As atividades de testes de software buscam incorporar confiabilidade e qualidade nos sistemas desenvolvidos, desta forma reduzindo custos tanto de manutenção como na correção de falhas do sistema.

Embora os testes de software não garantam que um aplicativo está livre de erros, eles auxiliam em mostrar que foram encontrados erros. A importância da realização de testes deve-se ao fato das outras atividades de garantia de qualidade de software serem insuficientes para a descoberta dos erros introduzidos ao longo do desenvolvimento do software. Apesar de ser impossível provar que um software está absolutamente correto por meio de testes, a sua utilização fornece evidências da conformidade com as funcionalidades especificadas.

Os testes também servem para apresentar aspectos de qualidade do software. Atividades de testes bem aplicadas e bem conduzidas aumentam o entendimento do sistema produzido, além disso, uma atividade de teste, conduzida de forma sistemática e criteriosa, auxilia no entendimento dos artefatos testados e evidencia as características mínimas do ponto de vista da qualidade do software.

Dependendo da empresa em que se está trabalhando ou do modelo em uso para o desenvolvimento de software, o processo de testes pode variar e ser aplicado de várias formas distintas, através de testes de validação que analisam o sistema para ver se está executando conforme o esperado ou testes de defeito, que são responsáveis pela exposição dos defeitos presentes (SOMMERVILLE, 2007).

Atualmente, o paradigma de programação orientado a objetos tem sido muito usado para o desenvolvimento de sistemas de software o que impulsiona o estudo de testes para este cenário. Durante os anos 80 aconteceu a consolidação do paradigma orientado a objetos para o desenvolvimento de software, proporcionando a realização de estudos visando estabelecer técnicas e critérios de teste voltados a esse paradigma (PRESSMAN, 2006).

Como a fase de testes é uma fase que demanda uma grande quantidade de tempo dentro do processo de desenvolvimento de software, faz-se necessária a utilização de políticas de

automação do processo para uma maior cobertura de testes e obtenção de melhorias no processo de teste.

A automação é vista como a principal maneira de se obter eficiência na fase de testes e, por esse motivo, várias medidas e soluções são apresentadas na literatura. A automação do teste consiste em repassar para o computador tarefas de teste de software que seriam realizadas manualmente, sendo feita geralmente por meio do uso de ferramentas de automação de teste. Podem ser consideradas para a automação as atividades de geração e de execução de casos de teste (FANTINATO *et al.*, 2007). Sem a utilização de ferramentas que dêem este suporte, a fase de testes estaria limitada a softwares de pequeno porte.

Neste capítulo serão apresentados conceitos relacionados a testes de software direcionados à programação orientada a objetos (POO) e os benefícios do uso de políticas de automação dos testes.

3.1 Testes de Software Orientados a Objetos

Os testes de software têm como objetivo encontrar a maior quantidade de erros possíveis com um esforço não muito elevado. Apesar do objetivo ser semelhante, a estratégia de testes em softwares desenvolvidos no paradigma de programação orientado a objetos deve ser diferente da estratégia de teste de sistemas baseados em linguagens estruturais. Algumas características de sistemas orientado a objetos como, por exemplo, encapsulamento, herança, polimorfismo e acoplamento dinâmico apresentam novos desafios aos engenheiros de software na fase de teste.

A possibilidade de representar objetos da realidade, faz com que a orientação a objetos seja um dos paradigmas de programação mais utilizado atualmente. O uso de classes, objetos, atributos, métodos, herança, polimorfismo e outros conceitos relevantes a orientação a objetos simplificam o desenvolvimento de software (BRITO e CARAPUÇA, 1994).

Nenhum programa, exceto os mais simplórios de todos, possuem uma sequência de execução pré-definida, podendo ainda ter várias tarefas executando em paralelo. Isso faz com que os testes necessitem de características que possam tratar de forma adequada os sistemas, firmando o objetivo de encontrar erros.

Para a elaboração de bons testes deve-se definir os casos de teste, que são o conjunto de entradas fornecidas ao teste e as saídas esperadas, além das condições necessárias para a execução dos testes (SOMMERVILLE, 2007).

Fundamentalmente, os testes orientados a objetos são divididos em quatro partes: testes

de unidade, testes de integração, teste de sistema e teste de validação. As subseções seguintes apresentam esses tipos de testes.

3.1.1 Teste de Unidade

Os testes unitários em programação orientada a objetos visam encontrar erros nas classes que constituem o sistema. A menor unidade testável de uma classe são os seus métodos (operações). Uma classe pode possuir vários métodos distintos, por isso, devem ser testados um por um verificando se executam de forma adequada. Um cuidado a ser tomado é com as heranças existentes. Se um método é herdado por várias subclasses, é necessário que sejam testados todos os níveis de herança, isso se o método for importante, caso contrário, pode ser que o método nem seja testado por não valer o esforço a ser empregado (PRESSMAN, 2006).

Um teste de unidade orientado a objetos é equivalente a um teste de unidade convencional, apenas diferindo no aspecto de que o teste convencional tem seu foco nos módulos enquanto o orientado a objetos é guiado pelas operações encapsuladas na classe e pelo estado de comportamento da classe (PRESSMAN, 2006).

Os testes de unidade devem ser amplos podendo testar um conjunto de entradas bem variado, verificando os métodos com vários dados diferentes tentando encontrar erros existentes. Esse conjunto de teste deve ser planejado com a idéia de reutilização dos casos de teste em novas baterias de verificação futuramente. Quanto mais testes de unidades forem realizados, mais confiável o software se torna, sendo recomendável o início dos testes no momento da codificação da classe (LOPES e FERNANDES, 2007).

3.1.2 Teste de Integração

O teste de integração é responsável por verificar se os componentes do software funcionam de maneira satisfatória quando combinados e colocados a trabalhar de forma conjunta. Muitas vezes podem aparecer erros na comunicação entre módulos, provenientes das interfaces dos módulos que não conseguem manter um canal de comunicação, um problema muitas vezes causado pela diferença no tratamento dos dados.

Existem duas estratégias diferentes para a integração de sistemas orientados a objetos (PRESSMAN, 2006):

a) **Teste baseado no caminho de execução (***thread-based testing***)**: Integra o conjunto de classes necessárias para responder uma entrada ou evento do sistema.

b) **Teste baseado no uso (***use-based testing***)**: A integração é iniciada com as classes que são independentes, isto é, que não usam nada ou muito pouco de outras classes.

Para testar classes que dependam de outras classes que ainda não foram testadas será necessário a criação de *stubs* que irão fornecer dados concisos para a classe em teste (LIMA e TRAVASSOS, 2004). Esses *stubs* simulam o comportamento de uma classe e poderão ser usados em estágios mais avançados do desenvolvimento para implementação real da classe.

3.1.3 Teste de Sistema

O objetivo da aplicação do teste de sistema é testar o sistema como um todo, verificando se os recursos que foram especificados na análise do sistema foram implementados de forma correta. Geralmente este tipo de teste pode ser realizado por analistas de sistemas que não participaram diretamente da implementação. Esses testes buscam testar funcionalidades, desempenho, usabilidade, entre outras características.

Com o teste de sistema, procura-se verificar se todos os elementos do sistema (por exemplo, hardware, bases de dados, interfaces com usuário) combinam adequadamente e se a função / desempenho global do sistema é alcançada (SOMMERVILLE, 2007).

3.1.4 Teste de Validação

Equivalentes ao teste de sistema com a diferença de que são realizados pelos usuários finais do software. Esses testes são realizados tendo como guia tutoriais gerados na especificação das funcionalidades do software, no início do processo de desenvolvimento. Com esses tutoriais em mãos os usuários finais podem testar se as funcionalidades desejadas foram implementadas no software e se estão funcionando de maneira correta. Com a aprovação do sistema nesta etapa de testes, ele estará pronto a ser entregue aos clientes (SOMMERVILLE, 2007).

Os testes de validação podem ser realizados no ambiente de desenvolvimento, com a supervisão dos desenvolvedores que podem colaborar dando dicas, mas também podem ser realizados no ambiente do cliente para verificar se está funcionando fora do espaço de desenvolvimento.

3.2 Automação de teste

A automação de teste é uma alternativa para proporcionar a entrega de produtos mais confiáveis em menor tempo (OLIVEIRA *et al.*, 2007). Significa agilizar o processo de teste

de software com o uso de ferramentas e metodologias a fim de melhorar os resultados e a garantia de qualidade. A principal vantagem desta abordagem é que a execução dos testes automatizados é bastante ágil, tornando possível repetir a realização dos testes a um baixo custo e a uma velocidade significativamente maior (OLIVEIRA *et al.*, 2007).

Uma das principais técnicas de automação de teste apresentada na literatura é *record & playback* (FANTINATO *et al.*, 2007). Com ela é possível gravar as ações executadas por um usuário sobre a interface gráfica de uma aplicação e converter estas ações em *scripts* de teste que podem ser executados o número de vezes que se desejar. Cada vez que o *script* é executado, as ações gravadas são repetidas, exatamente como na execução original. Para cada caso de teste é gravado um *script* de teste completo que inclui os dados de teste (dados de entrada e resultados esperados), o procedimento de teste (passo a passo que representa a lógica de execução) e as ações de teste sobre a aplicação (FANTINATO *et al.*, 2007). Uma característica importante desta abordagem é que o custo de montagem dos *scripts* é pequeno, pois ocorrerá a montagem uma única vez se não houver alterações no caso de teste. Um exemplo de ferramenta que segue esses princípios é a ferramenta *Selenium IDE*[1] para testes de navegação de aplicações Web (PARIZI e VIEIRA, 2009).

Como é necessário muito esforço para executar todo o conjunto de testes manuais, dificilmente, a cada correção de um erro, toda a bateria de testes será executada novamente como seria desejável. Muitas vezes, isso leva a não detecção de erros de regressão (erros em módulos do sistema que estavam funcionando corretamente e deixam de funcionar) (BRUNTINK, 2004). A tendência é este ciclo se repetir até que a manutenção do sistema se torne uma tarefa tão custosa que passa a valer a pena reconstruí-lo completamente (BERNARDO e KON, 2008).

Existem muitos *frameworks*, ferramentas de apoio a realização de atividades, que auxiliam na realização dos testes, dentre eles pode-se citar um dos mais importantes que é o JUnit[2]. Este é um *framework open-source* com suporte à criação de testes automatizados na linguagem de programação Java.

O JUnit facilita a criação de código para a automação de testes unitários com apresentação dos resultados. Com ele, pode ser verificado se cada método de uma classe funciona da forma esperada, exibindo possíveis erros ou falhas, podendo ser utilizado para a execução de baterias de testes (FANTINATO *et al.*, 2007).

No próximo capítulo são explanados conceitos de métricas de software, as suas classificações e ferramentas que auxiliam na extração dessas métricas.

[1] Selenium IDE: http://seleniumhq.org/projects/ide/
[2] Junit: www.junit.org

4 *Métricas de Software Orientado a Objetos*

Muitas empresas de desenvolvimento de software possuem equipes especializadas na medição de atributos de qualidade do processo de desenvolvimento de sistemas. Para tal finalidade, o uso de métricas pode colaborar de forma efetiva para obtenção de tais atributos de qualidade.

As medições e as métricas permitem um melhor entendimento do processo utilizado para desenvolver um produto, assim como uma melhor avaliação do projeto e do próprio produto. As medições podem permitir melhorias no processo, aumentando a produtividade e comparações entre projetos de desenvolvimento.

A busca pela qualidade de software através do estabelecimento de métricas permite a criação de um histórico, aumentando a qualidade e a eficiência em projetos futuros através da diminuição de erros.

Apesar de mudanças que estão ocorrendo através de novas ideias e modelos que são utilizados nos processos de desenvolvimento, há engenheiros de software não aceitam como válidas as métricas de software. Alguns membros da comunidade de software continuam a alegar que o software é não mensurável, ou que tentativas de mensurá-lo deveriam ser adiadas até que se entenda melhor o software e os atributos que deveriam ser usados para descrevê-lo. Desse modo, a criação de novos modelos e métodos de desenvolvimento está quebrando a política da não mensuração. Esse é o caso das metodologias ágeis de desenvolvimento que possuem uma coleção de conceitos e valores simples, focados em princípios como melhoria da interação entre os membros da equipe, a entrega de software funcional de forma iterativa e agregação de valor ao cliente, entre outros (ELIAS e WILDT, 2008).

Um conjunto de métricas para sistemas orientados a objetos, foco deste trabalho, é conhecido como conjunto de métricas CK (CHIDAMBER e KEMERER, 1994). Esse conjunto é composto por seis métricas voltadas aos sistemas orientado a objetos, como por exemplo, número de métodos por classe (*Weighted Methods Per Class* - WMC), que é a contagem do número de métodos em uma classe. Outro conjunto de métricas muito utilizado no contexto orientado a objetos é o conjunto de métricas MOOD (*Metrics for Object Oriented Design* - Métricas para

projetos orientado a objetos) (BRITO e CARAPUÇA, 1994). Existem também, métricas que são obtidas com a análise apenas do código fonte que não levam em conta as abstrações do conjunto CK e MOOD.

Nesse capítulo serão detalhados os conjuntos de métricas que podem ser extraídas de um código fonte, métricas do conjunto CK e MOOD, além da apresentação de algumas ferramentas que podem servir de apoio na obtenção destas métricas.

4.1 Métricas de Código Fonte

As métricas de código possuem fundamental importância para o acompanhamento e controle nas atividades de codificação e testes. Um grande benefício dessas métricas é a redução de retrabalho, um dos maiores desperdícios que acontece no processo de desenvolvimento de software.

Muitas métricas podem ser obtidas com a análise no código, entre elas:

4.1.1 Métricas relacionadas ao tamanho e quantidade

Essas métricas são valores que representam o tamanho das classes em relação ao código fonte. As métricas LOC (*Line Of Code* - linhas de código), KLOC (*Kilo Line Of Code* - mil linhas de código), e SLOC (*Source Code Line Of Code* - linhas de código fonte), sendo que esta última representa o número de linhas lógicas e físicas, não realizando a contagem de linhas em branco ou comentários existentes no código, são exemplos de métricas relacionadas ao tamanho. Já métricas do número de métodos de uma classe e do número de atributos são exemplos de métricas relacionadas à quantidade (PRESSMAN, 2006).

4.1.2 Métricas relacionadas à complexidade

Indicam o quanto um determinado módulo de software é legível (de fácil compreensão), ou então o quão complexo um sistema pode ficar com um número elevado de aninhamentos de laço e comandos de decisão.

4.1.3 Métricas de cobertura de código

Provê uma maior abrangência do que está sendo desenvolvido, tornando o desenvolvimento coeso. Várias ferramentas auxiliam no processo de análise de cobertura de código, algumas são citadas na Subseção 4.4 deste trabalho.

Dependendo da linguagem de programação adotada pelas equipes de desenvolvimento, outras métricas de código podem ser alcançadas. Algumas métricas são mais focadas em um determinado paradigma de programação, como, por exemplo, as métricas para sistemas que utilizam o paradigma orientado a objetos, do conjunto CK (CHIDAMBER e KEMERER, 1994), apresentadas logo a seguir.

4.2 Conjunto de Métricas CK

O conjunto de métricas CK, criado por Chidamber e Kemerer (1994), é um conjunto de medições de atributos de sistemas orientados a objetos. Apesar de esse estudo ter sido apresentado em 1994, ele ainda serve de referência para o assunto (BEN, 2006; COSTA, 2002; BESTEIRO *et al.*, 2009; PIVETA, 2009).

Chidamber e Kemerer propuseram seis métricas para análise das características das classes constituintes do sistema, por isso, são conhecidas como Métricas Orientadas a Classe, sendo de grande importância em orientação a objetos. Medidas e métricas para uma classe individual, para a hierarquia de classes e para as colaborações entre as classes são de grande valor para um engenheiro de software que precisa avaliar a qualidade do projeto (PRESSMAN, 2006).

As métricas que compõem o conjunto CK são: métodos ponderados por classe, profundidade da árvore de herança, número de filhos, acoplamento entre as classes, resposta de uma classe e, falta de coesão em métodos conforme descrito a seguir:

4.2.1 Métodos ponderados por classe (*Weighted Methods per Class - WMC*)

Representa o número de métodos contidos na classe. Quanto maior for o número de métodos de uma classe, a mesma tende a ter uma aplicação bem específica limitando a capacidade de reutilização.

4.2.2 Profundidade da árvore de herança (*Depth of the Inherance Tree - DIT*)

Esta métrica representa o comprimento máximo do nó até a raiz da árvore de hierarquia. Árvores mais profundas favorecem a reutilização de código com uso da herança. Em Java, uma classe, quando criada, inicialmente possui $DIT = 1$, pois herda da classe *System.Object* (ELIAS e WILDT, 2008).

4.2.3 Número de filhos (*Number Of Children - NOC*)

Esta métrica mensura as subclasses imediatamente subordinadas a uma classe na hierarquia, que são conhecidas como filhas. Um *NOC* elevado pode indicar que está ocorrendo uma grande utilização da classe pai e que ela precisa de mais testes pelo fato de estar sendo bastante utilizada. Classes do topo da árvore de herança tendem a possuir um *NOC* maior.

4.2.4 Acoplamento entre as classes de objetos (*Coupling Between Object class - CBO*)

Esta métrica é a contagem de outras classes acopladas na classe em análise. A métrica *CBO*, relaciona o nível de acoplamento entre objetos, sendo eles simplesmente acessados (*Efferent Coupling - EC*), herdados ou por receberem exceções de um nível inferior (ELIAS e WILDT, 2008).

4.2.5 Resposta de uma classe (*Response For a Class - RFC*)

O conjunto de respostas de uma classe é um conjunto de métodos que podem ser executados potencialmente em resposta a uma mensagem recebida por um objeto daquela classe.

4.2.6 Falta de coesão em métodos (*Lack of COhesion in Methods - LCOM*)

Relaciona a falta de coesão, indicando quanto os métodos de uma classe se relacionam, como por exemplo, utilizando atributos compartilhados de uma classe.

4.3 Conjunto de Métricas MOOD

As métricas MOOD avaliam os aspectos inerentes a orientação a objetos: herança, ocultação de informação, acoplamento, polimorfismo e reusabilidade. São baseadas em um grupo de funções formalmente definidas e na teoria dos conjuntos da matemática. Assim como as métricas que compõem o conjunto CK, as MOOD são independentes de linguagem de programação, apenas com a restrição de serem para sistemas orientados a objetos.

As métricas MOOD são calculadas através de uma razão, onde o numerador é o número de ocorrências encontradas no sistema para a característica avaliada e o denominador é o maior número possível de ocorrências daquela característica no sistema. Com isso, o valor de uma

métrica MOOD está contido no intervalo de 0 a 1, onde 0 representa a ausência total e 1 representa a máxima ocorrência possível daquele aspecto no sistema.

O conjunto de métricas MOOD apresenta: fator herança de método, fator herança de atributo, fator acoplamento, fator agrupamento, fator polimorfismo, fator ocultação de método, fator ocultação de atributo e, fator reuso, conforme apresentado a seguir:

4.3.1 Fator Herança de Método (*Method Inheritance Factor - MIF*)

Indica o percentual de herança que ocorre no sistema. Essa métrica é obtida através de cálculos realizados em dados retirados da hierarquia de herança do sistema. Esses dados podem ser o número de métodos herdados, o número de métodos que não foram herdados, o número de métodos redefinidos, aqueles métodos que são herdados e têm uma redefinição na classe, entre outros (PRESSMAN, 2006).

4.3.2 Fator Herança de Atributo (*Attribute Inheritance Factor - AIF*)

Indica o percentual de atributos que foram herdados no sistema. O cálculo para obtenção desta métrica é similar ao realizado para a obtenção de *MIF*.

4.3.3 Fator Acoplamento (*COupling Factor - COF*)

Para analisar o fator de acoplamento entre classes, Brito e Carapuça (1994) se basearam no modelo cliente/servidor. O conceito desse modelo é que uma classe X é servidora de Y se Y referencia pelo menos um membro de X, sendo esse membro um método ou um atributo. Essa métrica pode ser obtida com a razão entre o número total de conexões existentes e o maior número possível de conexões para o sistema.

4.3.4 Fator Agrupamento (*Clustering Factor - CLF*)

É a indicação do percentual de agrupamentos (*clustering*) existentes no sistema. Brito e Carapuça (BRITO e CARAPUÇA, 1994) se basearam no conceito de grafos para a extração deste valor, onde os nós do grafo representam as classes e as relações são representadas pelas conexões cliente/servidor existentes. *CLF* representa a proporção entre a quantidade de agrupamentos de classes e o total de classes do sistema.

4.3.5 Fator Polimorfismo (*Polymorphism Factor - PF*)

Representa o percentual de polimorfismo ocorrido no sistema. O cálculo de *PF* é baseado na redefinição de métodos, pois não existe a possibilidade de haver polimorfismo sem a ocorrência de redefinição de métodos. *PF* é a razão entre a quantidade de métodos redefinidos no sistema e o total de possibilidades de redefinição no software.

4.3.6 Fator Ocultação de Método (*Method Hiding Factor - MHF*)

Indica o percentual de métodos ocultos no software. Métodos ocultos são aqueles métodos privados da classe. Para o cálculo desta métrica é necessário o levantamento de valores básicos como: o número de métodos visíveis, aqueles que podem ser acessados por outras classes; a quantidade de métodos ocultos e também a quantidade de métodos definidos, que nada mais é que a soma dos métodos visíveis com os ocultos.

4.3.7 Fator Ocultação de Atributo (*Attribute Hiding Factor - AHF*)

Assim como *MIF* e *AIF* são similares, *AHF* é similar ao *MHF*. Essa métrica corresponde ao percentual de atributos ocultos no sistema e é obtido com a razão entre os atributos ocultos e todos os que foram definidos nas classes.

4.3.8 Fator Reuso (*Reuse Factor - RF*)

Indica o percentual de reuso ocorrido em um sistema. O cálculo de *RF* é dado pela soma de dois valores:

i) O reuso de classes de bibliotecas do sistema: razão entre o número de classes reutilizadas de bibliotecas de classes e o número total de classes do sistema.

ii) O reuso por herança de métodos: calcula-se o produto entre o *MIF* e a quantidade de classes não pertencentes a bibliotecas do sistema. O reuso por herança de métodos é obtido com a razão deste produto e o número total de classes do sistema.

4.4 Ferramentas de Extração de Métricas

O uso de ferramentas para apoiar a avaliação de software é muito importante, pois é um trabalho que, em muitas vezes, é tecnicamente e economicamente inviável de forma manual

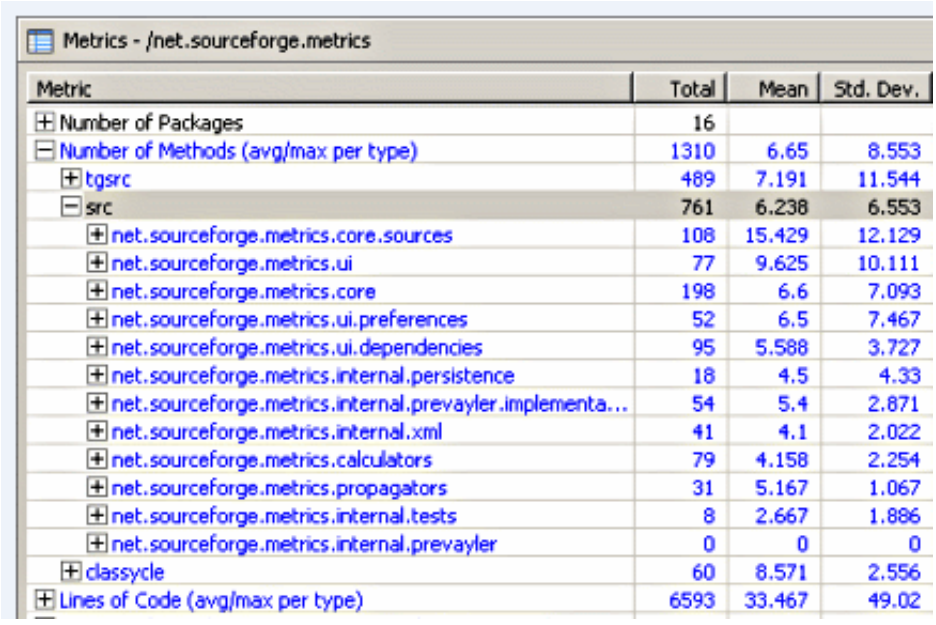
principalmente à medida que a complexidade do software aumenta. Algumas ferramentas que dão suporte a extração de métricas são JMetrics[3] e Metrics[4].

4.4.1 JMetrics

Esta ferramenta suporta a linguagem Java e tem por objetivo a extração de um conjunto de métricas de projeto, pacotes, métodos e atributos e também possibilita a extração de métricas a partir de bibliotecas de classes, gerando uma planilha com as métricas resultantes (VEIGA *et al.*, 1999).

4.4.2 Metrics

Ferramenta que realiza a medição de código fonte Java. É um *plug-in* que reúne um conjunto de métricas a serem usadas a partir da IDE (*Integrated Development Enterprise*) Eclipse, um ambiente de desenvolvimento de aplicações desktop, Web, entre outras, em várias linguagens de programação. A Figura 4.1 mostra algumas métricas extraídas de um projeto através da ferramenta Metrics:


Metrics - /net.sourceforge.metrics

Metric	Total	Mean	Std. Dev.
⊞ Number of Packages	16		
⊟ Number of Methods (avg/max per type)	1310	6.65	8.553
⊞ tgsrc	489	7.191	11.544
⊟ src	761	6.238	6.553
⊞ net.sourceforge.metrics.core.sources	108	15.429	12.129
⊞ net.sourceforge.metrics.ui	77	9.625	10.111
⊞ net.sourceforge.metrics.core	198	6.6	7.093
⊞ net.sourceforge.metrics.ui.preferences	52	6.5	7.467
⊞ net.sourceforge.metrics.ui.dependencies	95	5.588	3.727
⊞ net.sourceforge.metrics.internal.persistence	18	4.5	4.33
⊞ net.sourceforge.metrics.internal.prevayler.implementa...	54	5.4	2.871
⊞ net.sourceforge.metrics.internal.xml	41	4.1	2.022
⊞ net.sourceforge.metrics.calculators	79	4.158	2.254
⊞ net.sourceforge.metrics.propagators	31	5.167	1.067
⊞ net.sourceforge.metrics.internal.tests	8	2.667	1.886
⊞ net.sourceforge.metrics.internal.prevayler	0	0	0
⊞ classycle	60	8.571	2.556
⊞ Lines of Code (avg/max per type)	6593	33.467	49.02

FIGURA 4.1: Ferramenta de extração de métricas - Metrics

A tabela 4.1 mostra as métricas extraídas por esse *framework*.

[3]JMetrics: http://sourceforge.net/projects/jmetrics/
[4]Metrics: http://metrics.sourceforge.net/

Existem ainda, outras ferramentas que podem auxiliar na extração das métricas de código de software. Entre elas JavaNCSS3[5] e PMD[6] que extraem a complexidade ciclomática, métrica que indica o quanto o módulo do software é de fácil entendimento e, Krakatau Metrics[7] que calcula métricas de complexidade e outras relacionadas as linguagens C/C++ e Java (PRESSMAN, 2006).

A resolução de problemas, através da análise de vários critérios, é importante pois vários pontos de vista podem ser considerados na resolução de tal problema, várias análises podem ser realizadas e, além disso, os métodos que apoiam a resolução dessa classe de problemas, geralmente possibilitam a escolha da melhor solução do conjunto das factíveis.

No próximo capítulo são explanados alguns conceitos sobre métodos multicritérios de apoio a tomada de decisão além da apresentação do método AHP, suas fases e um exemplo de uso.

[5] JavaNCSS: http://www.kclee.de/clemens/java/javancss
[6] PMD: http://pmd.sf.net
[7] Krakatau Metrics: http://www.powersoftware.com/kepm/

TABELA 4.1:
Tabela com métricas extraídas pelo Metrics

Nome da métrica	**Descrição**
NORM - Número de métodos substituídos	Cálculo do número de métodos substituídos no escopo selecionado por classes ancestrais
NOF - Número de atributos	Representa o número de atributos da classe
NSC - Número de filhos	Número total de subclasses diretas de uma classe
NOC - Número de classes	Número de classes em cada pacote do sistema
MLOC - Linhas de código nos métodos	Contagem e soma das linhas de código dos métodos da classe, excluíndo linhas em branco e comentários
NOM - Número de métodos	Número de métodos referente ao escopo escolhido
NBD - Profundidade dos blocos aninhados	Valor da profundidade de blocos aninhados
DIT - Profundidade na árvore de herança	Comprimento do maior caminho a partir de uma classe até a classe base da hierarquia
NOP - Número de pacotes	Número total de pacotes no escopo selecionado
CA - Acoplamento Aferente	Número de classes localizadas fora de um pacote que dependem de classes de dentro do pacote
NOI - Número de interfaces	Número total de interfaces no escopo selecionado
VG - Complexidade ciclomática McCabe	Número de fluxos em um determinado trecho de código
TLOC - Total de linhas de código	Total de linhas de código no escopo selecionado. Não considera linhas em branco ou de comentário
RMI - Instabilidade	Grau de instabilidade no escopo selecionado. Varia no intervalo de [0,1]
PAR - Número de parâmetros	Número total de parâmetros no escopo escolhido
LCOM - Falta de coesão em uma classe	Valor de coesão de uma classe
CE - Acoplamento Eferente	Número total de classes de dentro de um pacote que dependem de classes de fora do pacote
NSM - Número de métodos estáticos	Número total de métodos estáticos no escopo selecionado
RMD - Distância normalizada	Distância perpendicular do componente até a sequência principal
RMA - Abstração	O número de classes abstratas (e interfaces), dividido pelo número total de tipos em um pacote
SIX - Índice de especialização	Média do indíce de especialização no escopo selecionado
WMC - Número de métodos em uma classe	Total de métodos pertencentes a uma classe
NSF - Número de atributos estáticos	Total de atributos estáticos no escopo selecionado

5 *Métodos Multicritérios de Apoio à Tomada de Decisão*

A sociedade atual enfrenta situações que exigem cada vez mais o uso de metodologias de apoio à tomada de decisão. Essas situações requerem o uso de instrumentos eficientes, eficazes e que possuam flexibilidade de tratar de problemas complexos de uma forma simples de modo a facilitar o entendimento do usuário.

A forma como as decisões são realizadas é tão importante quanto o que será decidido. Decidir pela escolha acadêmica de maneira muito rápida pode ser desastroso e demorar muito pode significar perder oportunidades. De qualquer forma, a escolha precisa ser feita. E uma abordagem sistêmica e compreensiva é necessária para a tomada de decisão (BESTEIRO *et al.*, 2009).

A tomada de decisão deve buscar uma opção que apresente o melhor desempenho, a melhor avaliação, ou o melhor acordo entre as expectativas do decisor, considerando a relação entre os elementos. Podemos, então, definir a decisão como um processo de análise e escolha entre várias alternativas disponíveis do curso de ação que a pessoa deverá seguir.

Esse contexto levou à formulação de vários métodos para a construção de modelos que possam auxiliar no processo de decisão, tais como: teoria dos jogos, programação linear, árvores de decisão. Atualmente, as decisões geralmente são baseadas na análise de vários critérios que estabelecem a natureza do problema (COSTA, 2002). Essas metodologias são conhecidas como Auxílio Multicritério à Decisão (AMD), Tomada de Decisão Multicritério (*Multiple Criteria Decision Making* - MCDM), Análise de Decisões com Múltiplos Critérios (ADMC). Dentre as metodologias pertencentes a AMD pode-se citar os Métodos da Família ELECTRE (*ELimination Et Choice TRadusàint la rEalitè*), o método PROMETHEE, o método Macbeth e Método de Análise Hierárquica (AHP), que são os mais conhecidos da literatura (COSTA, 2002).

Esse capítulo apresenta conceitos referentes ao método AHP, suas etapas e suas aplicações no auxílio à tomada de decisão em problemas multicritérios. A escolha do método AHP, como método a ser utilizado, é pelo motivo de ele atender de forma satisfatória às necessidades impos-

tas pelo trabalho. Não houve comparações entre métodos a fim de escolher o melhor, por não ser este o foco do trabalho, que é realizar uma priorização das classes a testar.

5.1 Analytical Hierarchy Process - AHP

O método AHP (SAATY, 1990) é um dos métodos AMD mais usado atualmente para a tomada de decisões em problemas com vários critérios. É eficaz porque identifica a melhor opção dentre as possíveis e ajuda na determinação de prioridades, considerando aspectos quantitativos e qualitativos (BEN, 2006).

O método AHP é muito utilizado na solução de problemas em que decisões devem ser tomadas baseadas na análise de vários critérios. Entre as aplicações deste método estão: seleção de funcionários para empresas, escolha de cursos em universidades, escolha de um automóvel, busca por oportunidades de refatoração de código (PIVETA, 2009), entre outros. No caso deste trabalho, a aplicação do AHP é na priorização de classes de um sistema orientado a objetos com o objetivo de priorizá-las para a realização de testes de unidade.

Para aplicar o AHP, primeiramente, deve-se definir o objetivo global e, assim, selecionar os critérios e alternativas para alcançar esse objetivo. Para isso é necessário que tanto os critérios quanto as alternativas possam ser estruturadas de forma hierárquica, sendo que o primeiro nível da hierarquia corresponde ao propósito geral do problema, o segundo, aos critérios e o terceiro, as alternativas como apresentado na Figura 5.1.

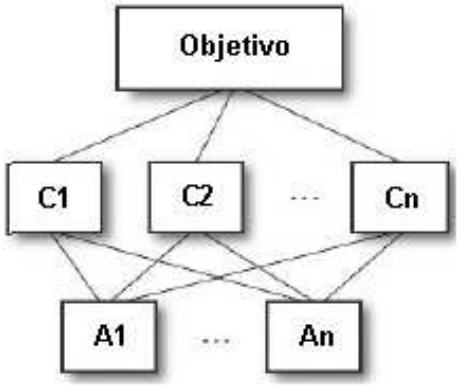


FIGURA 5.1: Estrutura Hierárquica proposta por Thomas L. Saaty

Na construção de um modelo de estabelecimento de prioridades fundamentado no uso de AHP, são desenvolvidas as seguintes etapas (COSTA, 2002):

i) Construção de hierarquia, identificando: foco principal, critérios, subcritérios (quando houver) e alternativas. Estes elementos formam a estrutura da hierarquia;

ii) Aquisição de dados ou coleta de julgamentos de valor emitidos por especialistas;

iii) Síntese dos dados obtidos dos julgamentos, calculando-se a prioridade de cada alternativa em relação ao foco principal; e,

iv) Análise da consistência do julgamento, identificando o quanto o sistema de classificação utilizado é consistente na classificação das alternativas viáveis.

Esse método está baseado em três princípios do pensamento analítico (COSTA, 2002): construção das hierarquias, definição das prioridades e análise da consistência, descritos a seguir:

5.1.1 Construção das hierarquias

No AHP, os problemas são estruturados de forma hierárquica a fim de buscar uma melhor compreensão e avaliação do mesmo. Na construção da hierarquia podem ser identificados os elementos chaves para a tomada de decisão, agrupando-os em conjuntos afins.

5.1.2 Definição das prioridades

Na etapa de estimação de prioridades são definidas as prioridades para os critérios e as alternativas. A relativa importância de um critério sobre os outros e cada alternativa sobre as demais é calculada realizando uma comparação de par a par, também chamada de julgamentos paritários. Os valores base para o estabelecimento das prioridades podem ser obtidos da tabela de prioridades definida por Thomas L. Saaty (1990), mostrada na Tabela 5.1.

TABELA 5.1:
Tabela de importância de Thomas L. Saaty

Grau	Importância
1	Mesma importância
2	Ligeiramente mais importante
3	Fracamente mais importante
4	Fraca a moderadamente mais importante
5	Moderadamente mais importante
6	Moderada a fortemente mais importante
7	Fortemente mais importante
8	Grandemente mais importante
9	Absolutamente mais importante

Com a definição das prioridades pode ser criada uma matriz quadrada com a relativa importância dos critérios sobre os demais. Considere, por exemplo, um conjunto de critérios $C = \{c_1, c_2, ..., c_n\}$ e uma matriz quadrada M representando a importância de um critério sobre outro $W = \{w_{11}, w_{12}, ..., w_{nn}\}$. A matriz de comparações é preenchida com a importância da comparação entre dois critérios na posição respectiva de relacionamento entre os critérios (linha e coluna) e $1/importancia$ na posição que corresponde à comparação inversa entre os dois critérios. Por exemplo, dados dois critérios C_1 e C_2, a comparação C_1/C_2 se dará nas posições de C_1 e C_2 na matriz, onde será inserido o valor definido como importância. Na posição C_2 e C_1, será inserido o valor $1/importancia$. Com isso, a matriz de comparações pareadas é construída da seguinte forma:

$$M = \begin{bmatrix} 1 & w_{12} & \cdots & w_{1n} \\ 1/w_{21} & 1 & \cdots & w_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 1/w_{n1} & 1/w_{n2} & \cdots & 1 \end{bmatrix} \tag{5.1}$$

Nesse ponto, vale ressaltar que a diagonal principal da matriz de comparações é igual a 1, pois um critério não pode ser comparado com ele mesmo.

Considere agora, por exemplo, 3 critérios c_1, c_2 e c_3 e as comparações pareadas com base na tabela 5.1, c_1 é fracamente mais importante do que c_3 ($c_1/c_3 = 3$), c_2 é ligeiramente mais importante do que c_1 ($c_2/c_1 = 2$) e c_2 é fraca a moderadamente mais importante do que c_3 ($c_2/c_3 = 4$). A partir destas comparações a matriz M é criada como segue:

$$M = \begin{bmatrix} 1 & 1/2 & 3 \\ 2 & 1 & 4 \\ 1/3 & 1/4 & 1 \end{bmatrix} \tag{5.2}$$

Através da matriz de comparações, calcula-se um vetor contendo os pesos relativos de todos os critérios da matriz, correspondendo a etapa de síntese. Esse vetor é calculado a partir de sucessivas elevações da matriz de prioridades ao quadrado onde a cada iteração a soma das linhas é calculada e normalizada, até que a diferença entre a soma de dois cálculos consecutivos não seja significativa, sendo analisadas quatro casas decimais à direita da vírgula. Geralmente são suficientes entre duas e quatro iterações para que o método encontre a solução.

Continuando o exemplo, um vetor pode ser extraído da matriz M com a sua elevação ao

quadrado, soma de suas linhas e por fim a normalização, como apresentado abaixo:

$$M = \begin{bmatrix} 1.00 & 0.50 & 3.00 \\ 2.00 & 1.00 & 4.00 \\ 0.33 & 0.25 & 1.00 \end{bmatrix} \tag{5.3}$$

$$M'(M^2) = \begin{bmatrix} 3.00 & 1.75 & 8.00 \\ 5.33 & 3.00 & 14.00 \\ 1.17 & 0.67 & 3.00 \end{bmatrix} \tag{5.4}$$

$$Linhas\ somadas\ M' = \begin{bmatrix} 12.7500 \\ 22.3332 \\ 4.8333 \end{bmatrix} \tag{5.5}$$

$$total\ soma\ vetor = 39.9165$$

$$vetor\ normalizado = \frac{12.7500}{39.9165}\ \frac{22.3332}{39.9165}\ \frac{4.8333}{39.9165} \tag{5.6}$$

$$vetornormalizado = \begin{bmatrix} 0.3194 \\ 0.5594 \\ 0.1210 \end{bmatrix} Totalnormalizado = 1.0000 \tag{5.7}$$

Após as iterações necessárias ao processo, o vetor obtido é o seguinte:

$$vetor\ normalizado = \begin{bmatrix} 0.3196 & 0.5584 & 0.1220 \end{bmatrix} \tag{5.8}$$

A realização de mais iterações não altera o vetor de forma significativa (fazendo uso de 4 casas decimais), tornando novas iterações desnecessárias ao processo.

Analisando os valores do vetor, obtém-se uma função matemática que expressa a importância dos critérios. A função matemática obtida no exemplo é a seguinte:

$$f(v_c) = v_{c1} \times 0.3196 + v_{c2} \times 0.5584 + v_{c3} \times 0.1220. \tag{5.9}$$

onde v_{c1}, v_{c2} e v_{c3} são valores quantitativos do problema.

5.1.3 Análise de consistência

É a última etapa do processo e possui a responsabilidade de verificar a consistência das comparações pareadas (par a par) presentes na matriz. É necessário que se realize essa verificação já que a matriz de pareamento pode inserir inconsistências no processo afetando outros julgamentos e podendo alterar a escolha a ser tomada.

O resultado desta etapa vai informar se o nível de inconsistência presente no processo é aceitável ou não. O processo é considerado aceitável se estiver com inconsistência abaixo dos 10%. O cálculo para obtenção do nível de inconsistência (*CR*) é dado pela equação abaixo:

$$CR = \frac{CI}{RCI} \tag{5.10}$$

CI é o primeiro grau de precisão da comparação em pares obtido através da análise de um valor λ_{max} e o número de critérios definidos para o problema *n*, representado pela equação abaixo:

$$CI = \frac{\lambda_{max} - n}{n-1} \tag{5.11}$$

O valor de λ_{max} é o significado da divisão da soma dos valores do vetor pelos valores do autovetor:

$$\lambda_{max} = 0.3196 \times \begin{bmatrix} 1 \\ 2 \\ \frac{1}{3} \end{bmatrix} + 0.5584 \times \begin{bmatrix} \frac{1}{2} \\ 1 \\ \frac{1}{4} \end{bmatrix} + 0.1220 \times \begin{bmatrix} 3 \\ 4 \\ 1 \end{bmatrix} \tag{5.12}$$

$$= \begin{bmatrix} 0.9646 \\ 1.6852 \\ 0.3680 \end{bmatrix} = \frac{\frac{0.9646}{0.3196} + \frac{1.6852}{0.5584} + \frac{0.3680}{0.1220}}{3} = 3.0177 \tag{5.13}$$

$$CI = \frac{\lambda_{max} - n}{n-1} = \frac{3.0177 - 3}{3-1} = 0.0088 \tag{5.14}$$

O Valor de *RCI* é um índice de consistência randômico fornecido por Thomas L. Saaty que varia de acordo com o número de critérios definidos *n*, mostrado na tabela 5.2:

TABELA 5.2:
Índices de *RCI* propostos por Saaty

n	< 3	3	4	5	6	7	8
RCI	0	0.58	0.9	1.12	1.24	1.32	1.41

Os dados do exemplo são consistentes, devido ao valor de *CR* obtido estar abaixo de 10%

como mostra a equação abaixo:

$$CR = \frac{CI}{RCI} = \frac{0.0088}{0.58} = 0.0157 = 1.50\% \tag{5.15}$$

Se a taxa de inconsistência obtida através da análise realizada pelo método AHP for inferior a 10%, o processo é considerado aceitável, caso contrário, não é aceito e as comparações precisam ser reavaliadas.

Uma possibilidade de avaliação da consistência das comparações pareadas caso o valor resultante da análise do AHP seja maior que 10%, pode ser através da busca de ciclos no grafo dirigido, que pode ser gerado através das comparações definidas.

Conforme as comparações são informadas, cria-se uma matriz auxiliar, contendo as adjacências dos critérios, que será 1 se houver comparação entre os critérios e 0 caso contrário, respeitando a direção estabelecida pela ordem da comparação. O grafo gerado conterá n vértices, sendo n o número de critérios definidos para o processo. Por exemplo, dado um conjunto de critérios $C = \{C_1, C_2, C_3\}$, o usuário deverá informar 3 comparações à ferramenta $\{(C_1/C_2$ ou $C_2/C_1)$, $(C_1/C_3$ ou $C_3/C_1)$, $(C_2/C_3$ ou $C_3/C_2)\}$. Através dessas comparações informadas pelo usuário, cria-se a matriz auxiliar contendo as adjacências das comparações, podendo ser usada posteriormente para a busca de ciclos. Continuando o exemplo, consideramos como definidas as seguintes comparações:

- Comparação 1: $C_1/C_2 \rightarrow$ Sendo o critério 1 mais importante que o critério 2;
- Comparação 2: $C_1/C_3 \rightarrow$ Sendo o critério 1 mais importante que o critério 3;
- Comparação 3: $C_3/C_2 \rightarrow$ Sendo o critério 3 mais importante que o critério 2;

Nesse caso, a matriz auxiliar das comparações gerada contendo as adjacências é a seguinte:

$$M = \begin{bmatrix} 0 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \tag{5.16}$$

Através dessa matriz pode ser gerado o grafo das comparações, mostrado pela figura 5.2.

A partir do grafo, através do algoritmo de busca em profundidade, pode-se dizer se foi ou não encontrado um ciclo no grafo. Caso seja encontrado um ciclo, o mesmo é removido e uma nova taxa de inconsistência é calculada. Esse processo deve ser realizado enquanto o valor da taxa de inconsistência não for inferior a 10%, pela possibilidade de existir mais de um ciclo e

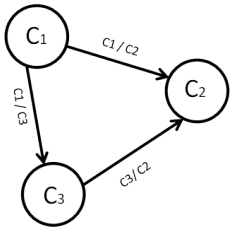


FIGURA 5.2: Grafo gerado a partir das comparações pareadas

de nem todos eles contribuírem de forma direta para que o índice de inconsistência seja maior que 10%.

O próximo capítulo apresenta o processo proposto para a priorização de classes, uma ferramenta desenvolvida para viabilizar a execução da priorização e um estudo de caso como forma de avaliação do processo e da ferramenta propostos.

6 *A Priorização de Classes para Testes Unitários*

Priorizar classes para testes é poder estabelecer uma ordem de aplicação de testes de unidade em classes participantes de um projeto de software. Essa priorização é de grande importância para empresas de desenvolvimento de software que possuem pouco prazo para a entrega do produto e sem tempo de testar todas as classes do sistema, o que não é muito indicado, ou mesmo para aqueles desenvolvedores que desejam testar primeiro as classes que sejam mais importantes, em relação aos critérios definidos, para depois testar o restante das classes.

Existem várias metodologias que podem ser usadas para o desenvolvimento de software. No passado, as empresas utilizavam o modelo clássico de desenvolvimento composto por etapas bem definidas e com o cronograma estático. Essas etapas eram o levantamento de requisitos, a análise dos requisitos, o projeto do sistema, a implementação, os testes e a manutenção (PRESSMAN, 2006). Esse processo seguia um cronograma elaborado no seu início e que causava uma variedade de problemas no decorrer do tempo. Caso algum imprevisto acontecesse em qualquer uma das etapas, o cronograma era alterado, afetando a realização das etapas subsequentes. Nesses casos, os testes eram afetados prejudicando a verificação da qualidade do produto a ser entregue para o cliente.

Um problema enfrentado pelas equipes de desenvolvimento do modelo clássico é ter que testar todas as classes que foram projetadas e implementadas. A solução para este problema foi a criação de metodologias ágeis para o desenvolvimento de software como o XP, *Scrum*, entre outros, que apresentam um conjunto de inovações ao processo de desenvolvimento de software.

As mudanças trazidas pelas metodologias ágeis em relação ao teste de software, é que o próprio implementador deve testar a sua classe (BERNARDO e KON, 2008). Mas, nesse caso, geralmente um implementador não implementa apenas uma classe do sistema, o que faz com que a priorização se torne importante.

A ideia da priorização das classes baseada em critérios é propor uma sequência de classes para que sejam aplicados os testes. Esse trabalho tem por objetivo propor uma sequência de classes priorizadas para a realização dos testes. Essa sequência de teste não é a ótima em

relação a heurísticas de ordenação de classes, é apenas uma proposta de rankeamento baseada em critérios definidos para o conjunto de classes.

A priorização de classes para testes pode ser aplicada nos processos de desenvolvimento de software que tem como base o modelo clássico, onde auxilia os testadores a focar seus esforços em classes mais importantes e testá-las primeiramente. Também pode ser aplicada em processos de desenvolvimento que usam metodologias ágeis, onde, auxilia os desenvolvedores, que neste modelo de desenvolvimento, testam suas próprias classes, indicando qual classe testar primeiro, pelo fato de que geralmente um implementador desenvolve mais de uma classe.

A priorização das classes pode ser realizada por qualquer integrante da equipe de desenvolvimento, desde que este possua conhecimentos do projeto e das classes que o constituem. Também há a necessidade que o executor da priorização tenha em mente os critérios que serão aplicados nas classes para a sua priorização.

Neste capítulo são apresentados o processo de priorização proposto e suas etapas, uma ferramenta desenvolvida para automatizar a execução da priorização e um estudo de caso como forma de avaliação do modelo e da ferramenta.

6.1 Processo de Priorização

O processo de priorização de classes descreve as etapas necessárias para que um conjunto de classes seja ordenado de forma prioritária baseado em critérios definidos previamente. Fazem parte deste processo a organização do conjunto de classes, estabelecimento dos critérios e suas respectivas importâncias, a análise destes critérios aplicados às classes, ordenação das classes segundo o grau de prioridade obtido a partir dos critérios e a criação de uma lista com uma sequência proposta para teste de classes. Esse processo é apresentado na Figura 6.1.

6.1.1 Etapas do Processo de Priorização de Classes

Organização do conjunto de classes

Esta etapa do processo tem por objetivo organizar as classes para que todas fiquem disponíveis no momento em que forem necessárias para aplicação das demais etapas. Se o responsável pela priorização desejar, pode criar um repositório de armazenamento das classes que serão analisadas para priorização.

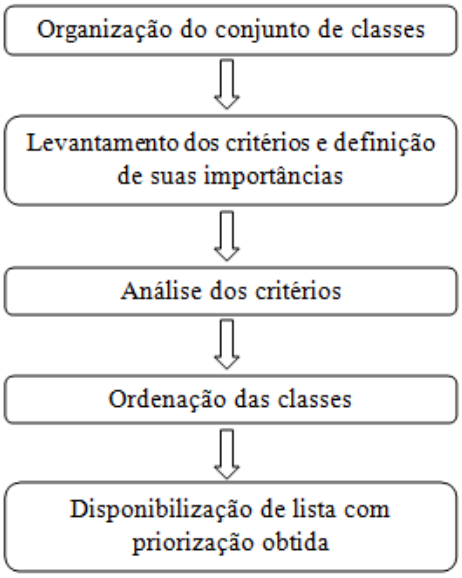


FIGURA 6.1: Etapas do processo de priorização

Levantamento dos critérios e definição de suas importâncias

Como apresentado no capítulo 5 deste trabalho, as métricas de software orientado a objetos podem ser usadas como critérios de avaliação das classes. Por exemplo, para uma determinada classe X um critério pode ser o número de linhas de código dessa classe, expresso pela métrica de código fonte (LOC). As importâncias dos critérios, definidas pelos responsáveis pela priorização, irá definir a ordem final das classes para teste. Por exemplo, os engenheiros que devem priorizar as classes de um projeto Y classificam as linhas de código (LOC) como de grande importância em relação ao nível de herança (DIT) baseados na tabela de Thomas L. Saaty mostrada na Tabela 5.1. Nesse ponto existe uma flexibilidade muito grande, pois podem ser estabelecidas diferentes importâncias para as métricas em projetos distintos.

Análise dos critérios

A análise dos critérios, representados pelas métricas de software, pode ser realizada com a aplicação do método AHP. Esse método fará comparações par a par dessas métricas e computará o grau de prioridade de cada classe em relação ao conjunto de classes.

Com a aplicação do método AHP obtém-se um valor para cada classe, o qual representa o grau de prioridade da classe em relação a todo conjunto de classes.

Ordenação das classes

A partir do grau de prioridade da classe perante o conjunto de classes, obtido com a aplicação da etapa anterior, realiza-se uma ordenação simples através do valor de prioridade, de forma decrescente, onde as classes prioritárias, classes que possuam maior grau de prioridade, sejam testadas primeiramente. Essa ordenação pode ser concebida com a aplicação de algum algoritmo de ordenação de conjuntos como, por exemplo, ordenação por seleção, por inserção, entre outros (KNUTH, 1997).

Disponibilização de lista com priorização obtida

A última etapa do processo de priorização é a disponibilização de uma lista que expressa a ordem em que as classes podem ser testadas. Essa ordem é apenas uma proposta baseada nas métricas e nas importâncias de cada métrica, definida pelos membros da equipe do projeto. Para projetos diferentes a ordem poderá sofrer alterações devido a novas classes, novos critérios, novas importâncias e, principalmente, a novos responsáveis pela priorização, dado que as importâncias são definidas por seres humanos que possuem ideias divergentes.

6.2 Ferramenta para o Processo de Priorização

Uma ferramenta foi desenvolvida baseada no modelo proposto por esse trabalho, provendo apoio computacional ao mesmo. A justificativa de desenvolver uma ferramenta é que em muitos casos o processo de priorização pode envolver vários critérios e várias classes, tornando-se desgastante ser realizado manualmente. Nomeada como TPTest (*Tool for Priorizating Test* - Ferramenta para a Priorização de Testes), a ferramenta visa guiar a execução da priorização de classes baseada em multicritérios, que podem ser métricas citadas no capítulo 4 deste trabalho. O esforço demandado a quem for realizar o processo de priorização será a informação dos critérios, das importâncias para as comparações pareadas e a importação do arquivo contendo as classes.

A ferramenta proposta por este trabalho pode ser usada em caso de aplicação de testes de regressão, onde ocorreram alterações na implementação do software, que por consequência, deve ser testado para avaliar se as funcionalidades presentes antes das alterações continuam funcionando corretamente. Também pode ser utilizada em situações que, os casos de teste não foram construídos antes, do uso da ferramenta.

Essa Seção apresenta uma visão geral da ferramenta, as atividades, um exemplo de utilização

e algumas limitações presentes na mesma.

6.2.1 Visão Geral da TPTest

A ferramenta TPTest foi desenvolvida com o uso da linguagem de programação orientada a objetos Java™, possuindo uma série de interfaces que possibilitam ao usuário participar ativamente do processo de priorização. Essas interfaces realizam o canal de comunicação entre o usuário e a ferramenta, facilitando a inserção dos dados necessários para o andamento da priorização de classes. A interface inicial da ferramenta TPTest é mostrada na Figura 6.2.

FIGURA 6.2: Interface inicial da ferramenta TPTest

Tendo em vista as etapas do processo de priorização apresentadas anteriormente, a ferramenta TPTest possui o seguinte fluxo padrão: (1) recebe como entrada as informações do projeto e as armazena; (2) recebe como entrada os critérios definidos e os armazena; (3) recebe como entrada os critérios definidos na etapa anterior juntamente com as importâncias para as comparações, compara os critérios de forma pareada e gera a função de prioridade; (4) recebe como entrada a função de prioridade juntamente com as classes e os critérios valorados quantitativamente e estabele o grau de prioridade para cada classe; (5) recebe as classes com o seu grau de prioridade e gera uma ordem das mesmas através de um método de ordenação; (6) recebe como entrada a ordem das classes e, juntamente com as informações do projeto definidas no início do fluxo, gera a documentação do processo de priorização que contém as informações do projeto, do processo e os resultados obtidos com a priorização de classes. A Figura 6.3 apre-

senta um diagrama contendo as atividades participantes deste fluxo e os artefatos de entrada e de saída relacionados a cada atividade.

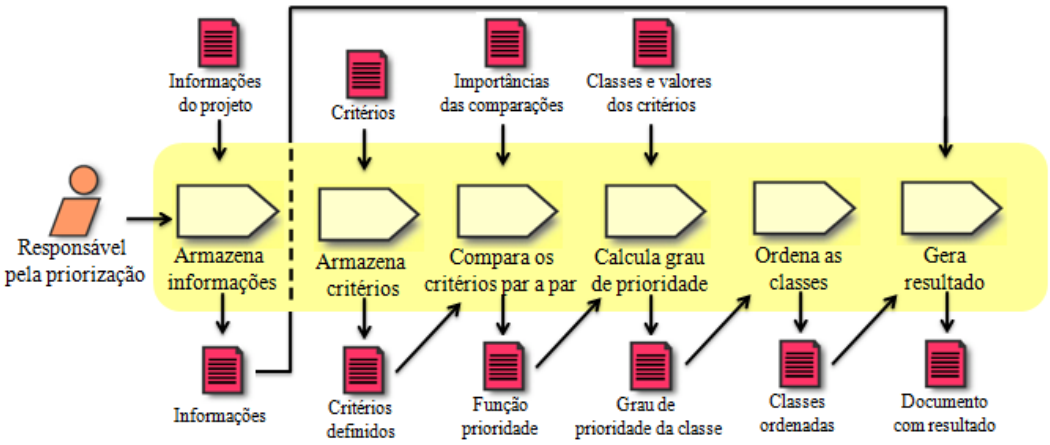


FIGURA 6.3: Diagrama de relacionamento das atividades da ferramenta TPTest

A figura 6.4 mostra a estrutura da ferramenta TPTest na visão de pacotes. A ferramenta possui cinco pacotes: *Beans*, *Core*, *Persistance*, *Print* e *TPTest*, descritos a seguir:

- *Beans*: contém classes que representam as entidades, como: ClasseBean, Criterio, ComparacaoPareadaBean;
- *Core*: contém as classes do núcleo da ferramenta, como: LeituraArquivoMetricas, RankingAHP, Resultado, ComparacaoPareada;
- *Persistance*: contém a classe ArmazenaDados, responsável pelo armazenamento das informações do processo de priorização;
- *Print*: contém a classe ImprimeDadosArquivo, responsável pela geração da documentação do processo de priorização;
- *TPTest*: que contém a classe de inicialização do sistema e as interfaces.

A Figura 6.5 mostra um diagrama com as principais classes da ferramenta. Nesse diagrama estão presentes classes como TPTest, classe principal que, associada as demais classes, realiza as atividades e funcionalidades da ferramenta. O diagrama foi desenvolvido com o apoio da ferramenta de diagramação UML *StarUML*[8].

[8]StarUML: http://staruml.sourceforge.net/

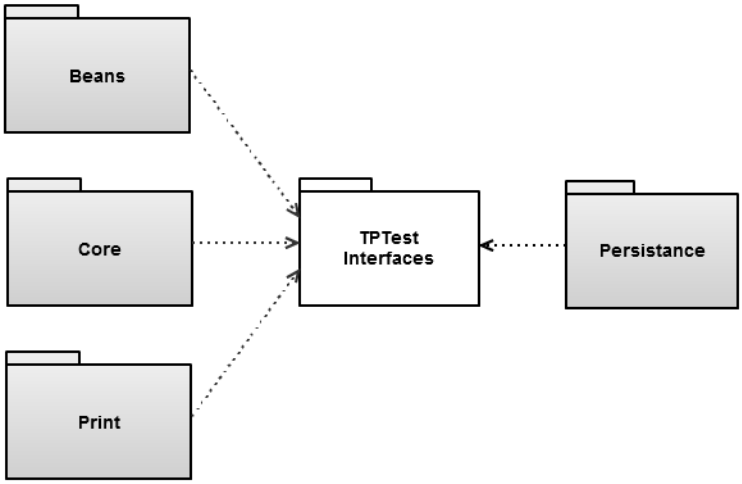


FIGURA 6.4: Estrutura da TPTest na visão de pacotes

6.2.2 Principais Atividades da TPTest

Para alcançar os objetivos, a ferramenta, é composta por uma série de atividades que realizam as tarefas necessárias para a execução do processo de priorização. As atividades são: *informação dos dados do projeto*, onde o usuário informa características do projeto em que a priorização é realizada, *definição dos critérios*, onde o usuário define um conjunto de critérios para a realização da priorização, *definição das importâncias das comparações pareadas*, onde o usuário define o quão importante um critério é sobre outro com base nos valores de Saaty, *importação das classes e valoração dos critérios*, atividade que tem por objetivo importar as classes que serão priorizadas juntamente com suas métricas e, *visualização dos resultados*, onde o usuário terá a sua disposição a ordem de classes para realização dos testes de unidade.

Cada uma destas atividades tem um objetivo específico e será analisada de forma isolada, para que se entenda como, de forma integrada, elas conseguem obter um rankeamento das classes para testar. A Figura 6.13 apresenta um diagrama de atividades da ferramenta desenvolvida.

Informação dos dados do projeto

Esta atividade é responsável por receber e armazenar dados referentes ao projeto de software no qual é aplicada a priorização das classes. Ela é importante pelo fato de os dados serem necessários para a geração da documentação que registra o processo de priorização realizado.

Os dados requeridos pela ferramenta são: o nome do sistema de software em desenvolvimento, uma descrição breve sobre o sistema e o nome do responsável pelo sistema em desenvolvimento. Os dados fornecidos pelo usuário são armazenados em disco de onde, serão lidos

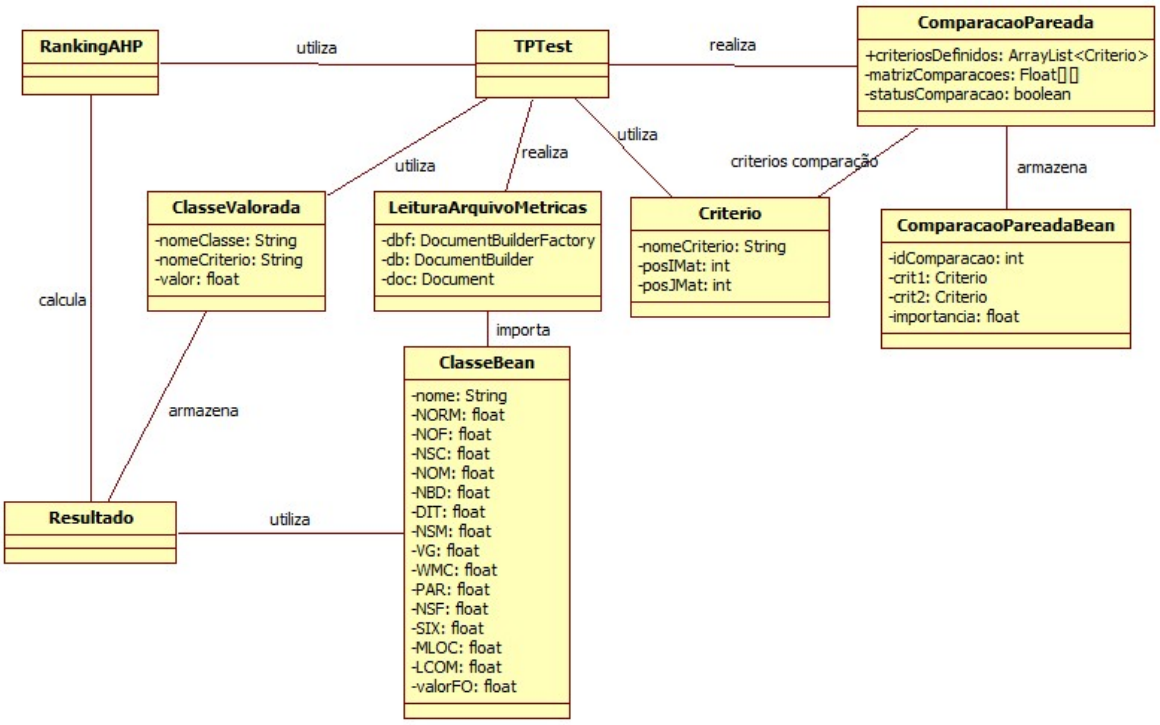


FIGURA 6.5: Diagrama contendo as principais classes da ferramenta

para a geração do documento com os resultados da priorização.

Definição dos critérios

Nesta atividade devem ser informados os critérios de avaliação das classes para que as mesmas possam ser priorizadas. Esses critérios podem ser as métricas de software apresentadas no capítulo 4, ou ainda podem ser qualquer outro critério de avaliação de classes disponível para a equipe que realiza a priorização.

As métricas que podem ser escolhidas pelo usuário são algumas das métricas extraídas pelo *framework Metrics*, um *plug-in* para o Eclipse IDE, que serve de apoio na realização do processo de priorização proposto por este trabalho, apresentado na seção 4.4.2.

A ferramenta TPTest fornece os nomes das métricas geradas pelo *Metrics* para a definição dos critérios, além da possibilidade de definição de critérios que não sejam métricas pertencentes aos conjuntos apresentados no Capítulo 4, como por exemplo, tempo de implementação, número de desenvolvedores envolvidos com a classe, número de testadores disponíveis para aquela classe, entre outros.

Definição das importâncias das comparações pareadas

Esta atividade é responsável por armazenar as importâncias para as comparações pareadas entre os critérios definidos anteriormente. O valor para as importâncias das comparações pode ser baseado na tabela 5.1 de Thomas Saaty. Nesse ponto, a TPTest baseia-se nos conceitos apresentados por Saaty na seção 5.1, gerando uma matriz quadrada de comparações pareadas. Essa matriz possuirá dimensão *nxn*, sendo *n* o número de critérios definidos. O número de importâncias requeridas pela ferramenta para as comparações será o necessário para preencher a parte triangular superior da matriz, sendo a parte triangular inferior representada por $1/elemento$ da triangular superior.

Através dos cálculos matemáticos realizados na matriz de comparações conforme apresentados na Seção 5.1.2, a ferramenta obtém uma função matemática que possibilita o cálculo do grau de prioridade de cada classe. Por exemplo, se fossem definidos 3 critérios (C_1, C_2, C_3) para a avaliação das classes, a função matemática obtida nessa atividade seria semelhante a função a seguir:

$$f = C_1 * Y_1 + C_2 * Y_2 + C_3 * Y_3 \tag{6.1}$$

onde C_1, C_2, C_3 seriam substituídos pelos valores quantitativos de cada classe, por exemplo, se C_1 fosse o número de métodos de uma classe, o valor na função matemática de C_1 para uma classe X seria o número de métodos da classe X, e Y_1, Y_2, Y_3 seriam os valores de avaliação obtidos através das comparações pareadas.

Finalmente nessa etapa, é realizada uma análise de consistência conforme apresentado na Subseção 5.1.3. Se as importâncias para as comparações, definidas por Saaty, forem consistentes a ferramenta estará habilitada a prosseguir para a próxima atividade, caso contrário, as importâncias deverão ser reinformadas. Na versão atual da TPTest apenas a análise de consistência realizada pelo método AHP é avaliada, não avaliando o grafo gerado pelas comparações e os possíveis ciclos existentes. Dessa forma, quando encontrada inconsistência nas comparações, o usuário é obrigado a informar todas as importâncias para as comparações, pelo fato de não ser avaliado em que comparação a inconsistência ocorre.

Importação das classes e valoração dos critérios

Nessa atividade, a ferramenta realiza a importação das classes através da leitura de um arquivo que deve ser informado pelo usuário. O arquivo contendo as classes pode ser obtido através da execução do *plugin Metrics* no projeto em que as classes se encontram. Juntamente com as classes, o arquivo conterá as métricas para cada classe ou para pacotes de classes,

métricas essas apresentadas na Tabela 4.1. Um exemplo de arquivo para importação das classes e suas métricas é apresentado pela Figura 6.6. Ele apresenta, nesse caso, as métricas da classe *"Layouter.java"*, que, por exemplo, para a métrica NORM possui um valor 30.

```
<?xml version="1.0" encoding="UTF-8" ?>
- <Metrics scope="jpplib-src-0.6" type="Project" date="2010-05-24">
  - <Metric id="NORM" description="Number of Overridden Methods">
    - <Values per="type" total="19" avg="0.655" stddev="1.294" max="4">
        <Value name="Layouter" source="Layouter.java" package="de.uka" value="30" />
      </Values>
    </Metric>
  + <Metric id="NOF" description="Number of Attributes">
  + <Metric id="NSC" description="Number of Children">
  + <Metric id="NOC" description="Number of Classes">
  + <Metric id="MLOC" description="Method Lines of Code">
  + <Metric id="NOM" description="Number of Methods">
  + <Metric id="NBD" description="Nested Block Depth" max="5">
  + <Metric id="DIT" description="Depth of Inheritance Tree">
  + <Metric id="NOP" description="Number of Packages">
  + <Metric id="CA" description="Afferent Coupling">
  + <Metric id="NOI" description="Number of Interfaces">
  + <Metric id="VG" description="McCabe Cyclomatic Complexity" max="10">
  + <Metric id="TLOC" description="Total Lines of Code">
  + <Metric id="RMI" description="Instability">
  + <Metric id="PAR" description="Number of Parameters" max="5">
  + <Metric id="LCOM" description="Lack of Cohesion of Methods">
  + <Metric id="CE" description="Efferent Coupling">
  + <Metric id="NSM" description="Number of Static Methods">
  + <Metric id="RMD" description="Normalized Distance">
  + <Metric id="RMA" description="Abstractness">
  + <Metric id="SIX" description="Specialization Index">
  + <Metric id="WMC" description="Weighted methods per Class">
  + <Metric id="NSF" description="Number of Static Attributes">
  </Metrics>
```

FIGURA 6.6: Formato do Arquivo de Importação das Classes

As métricas que a ferramenta importa do arquivo para cada classe, são métricas que tem relação direta com as classes, isto é, que indicam valores referentes a atributos das classes. As métricas NOC, NOP, CA, NOI, TLOC, RMI, CE, RMD e RMA não são importadas pela TPTest, pois são métricas que apresentam valores no nível de pacote e não de classes.

A ferramenta realiza a leitura dos dados presentes no arquivo através de um parser XML baseado na API (*Application Programming Interface* - Interface de Programação de Aplicativos) DOM (*Document Object Model*) (HORSTMANN e CORNELL, 2008). Com essa API, o documento XML é inteiramente armazenado em memória no formato de uma árvore de nodos, onde através da leitura de seus nós podem ser obtidas as informações desejadas.

Após realizada a importação, a TPTest verifica se todos os critérios definidos possuem valores para as classes lidas do arquivo. Se existirem critérios que não possuem valor, este deve

ser informado para que o grau de prioridade de uma classe em relação às demais seja obtido de forma correta. Um critério não terá um valor associado para cada classe se e somente se esse não for uma das métricas geradas pelo *Metrics*. Para as métricas conhecidas, a TPTest realiza a valoração automaticamente.

Visualização dos resultados

Nesse ponto, a ferramenta disponibiliza uma série de opções ao usuário quanto a forma de visualização do resultado do processo de priorização de classes. Caso o usuário desejar, poderá ordenar as classes de forma crescente ou decrescente com base no grau de importância da classe em relação as demais, ou ainda, ordenar as classes em ordem alfabética de A-Z ou Z-A. Outra opção da ferramenta para a visualização do resultado é através a apresentação das informações do processo de priorização. Essas informações vão desde os critérios definidos no início do processo até o resultado final com as classes ordenadas. O usuário poderá também salvar as etapas do processo de priorização realizadas para ter o controle das classes que foram priorizadas.

Na interface de visualização dos resultados, o usuário pode gerar um documento que serve como registro do processo de priorização. A TPTest fornece uma série de opções para a inserção ou não de determinadas informações do processo de priorização na documentação, ficando a critério do usuário utilizá-las ou não. Nesta documentação são adicionados os dados do projeto, definidos no início do processo. No APÊNDICE B é apresentado um exemplo de documentação gerada pela ferramenta TPTest.

6.2.3 Usando a Ferramenta TPTest

Seguindo o fluxo de execução, apresentado na Subseção 6.2.1, a seguir são apresentados alguns passos para a realização do processo de priorização através da ferramenta TPTest.

Incialmente o usuário deve *informar os dados do projeto* do qual as classes serão priorizadas. Se todos os dados forem informados de forma correta, o processo passa para a fase de definição dos critérios, caso contrário, os dados deverão ser informados novamente.

Para informar os dados do projeto o usuário pode acessar o menu *"Projeto"* e, então, a opção *"Configurar Informações Projeto"*, ou clicar no botão *"Configurar Informações Projeto"* na tela inicial. A Figura 6.7 mostra a interface de configuração dos dados do projeto cuja as classes se deseja priorizar.

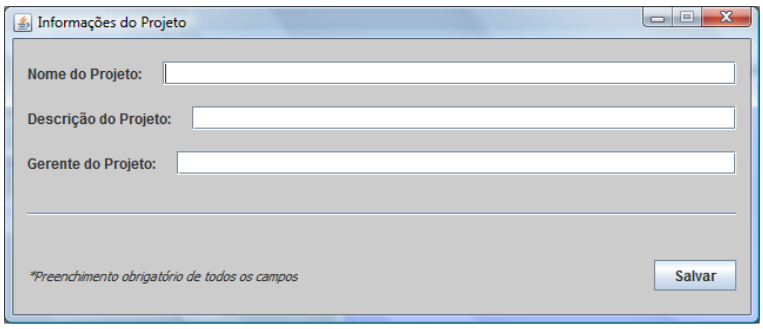


FIGURA 6.7: Interface de configuração dos dados do projeto

O próximo passo é a *definição dos critérios*, onde o usuário deve informar os critérios que deseja utilizar para realização da priorização de classes. Nesse ponto, tem como alternativas escolher métricas, definir um critério novo, que seria um critério qualquer que não uma métrica, ou excluir um critério definido. A Figura 6.8 apresenta a interface de definição de critérios da ferramenta TPTest.

Para avançar para a próxima etapa, o usuário deve clicar no botão *"Concluído"*. O painel *"Critérios Definidos"* apresenta os critérios que participarão do processo de priorização definidos pelo usuário.

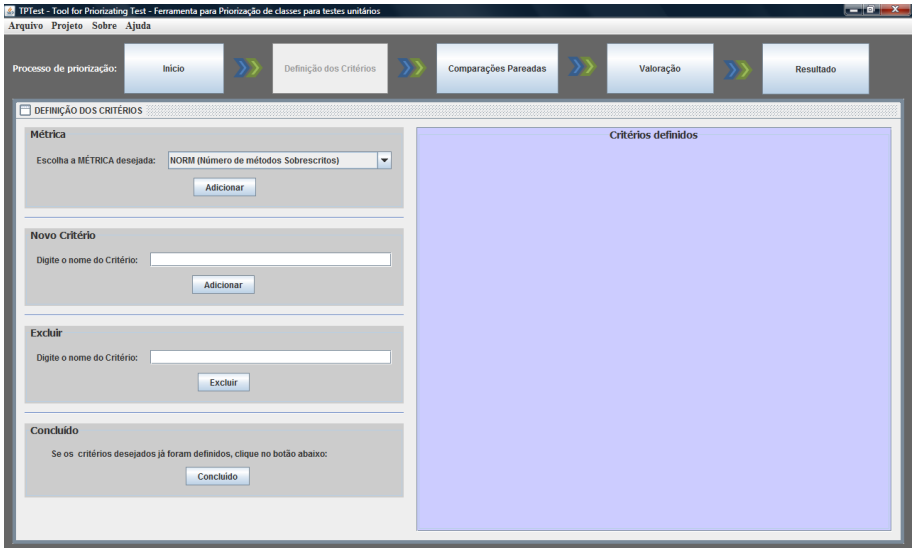


FIGURA 6.8: Interface de definição dos critérios do processo de priorização de classes

Após concluída a etapa de definição dos critérios, seguindo o processo proposto, a ferramenta requer o valor das *importâncias para as comparações pareadas* entre os critérios. O usuário poderá optar por adicionar as comparações ou editar as existentes, após terem sido adicionadas. Se as comparações apresentarem inconsistência deverão ser informadas novamente, caso contrário, a ferramenta irá para a etapa de importação de classes e valoração dos critérios. A Figura 6.9 apresenta a interface da TPTest para a definição das importâncias para

as comparações pareadas entre os critérios.

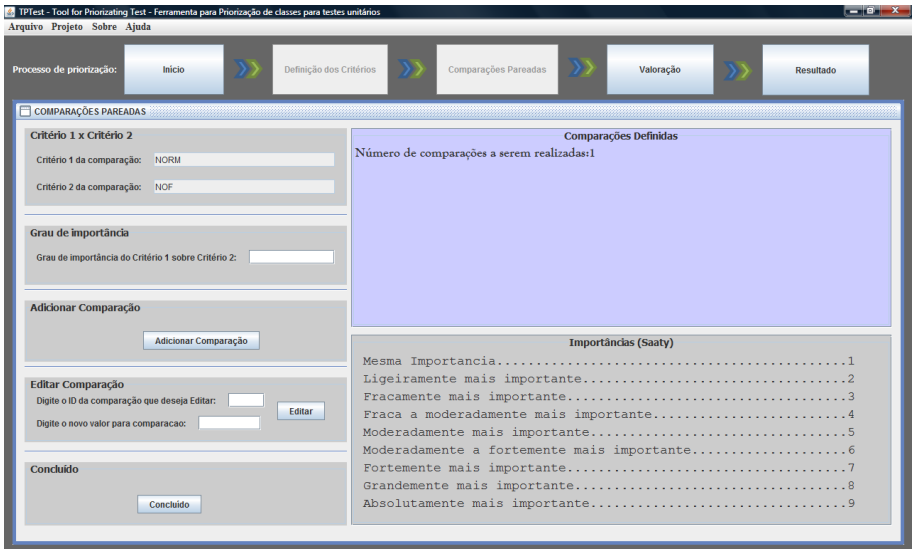


FIGURA 6.9: Interface de definição das importâncias para as comparações pareadas entre os critérios

O passo seguinte é o de *importação das classes*, onde o usuário deverá informar as classes que participarão do processo de priorização. Para importar as classes desejadas, basta informar o nome do arquivo com extensão XML *(Extensible Markup Language)* que contenha as classes e suas métricas e acionar o botão *"Importar Arquivo"*, presente na interface. A Figura 6.10 apresenta a interface de importação das classes. Neste passo também devem ser informados

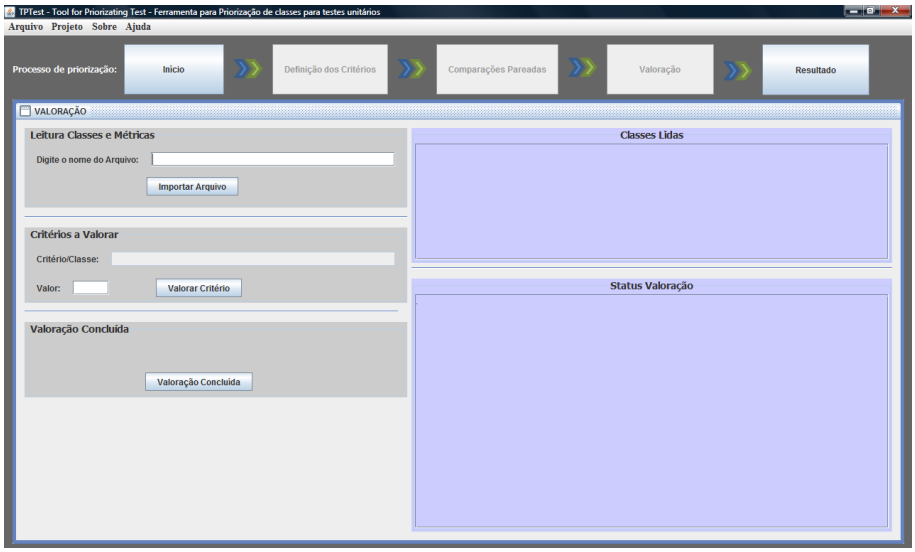


FIGURA 6.10: Interface de importação das classes e de suas métricas

valores quantitativos para os critérios definidos para o processo de priorização que não sejam métricas extraídas pelo *Metrics*.

Finalizada a importação das classes e valorados todos os critérios definidos, o usuário deve clicar no botão *"Valoração Concluída"* para ir para a próxima etapa do processo.

Finalmente, o usuário tem a sua disposição uma lista de classes ordenadas para a realização dos testes unitários. A Figura 6.11 apresenta a interface de visualização dos resultados.

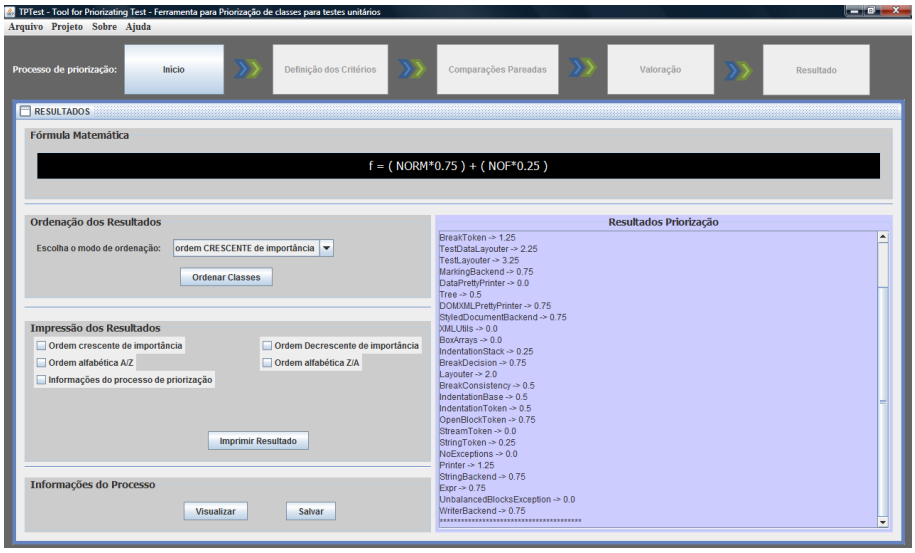


FIGURA 6.11: Interface de visualização dos resultados

O processo de priorização pode ser reiniciado a qualquer momento pelo usuário. Para tanto, basta clicar no botão *"Início"* que o processo reinicia, causando a perda dos dados do processo anterior. Caso o usuário necessite de ajuda, basta clicar no menu *"Ajuda"*, que a ferramenta apresenta um arquivo de ajuda no navegador padrão da máquina do usuário.

6.2.4 Limitações

Apesar de realizar as tarefas propostas pelo processo de priorização definido anteriormente, há algumas limitações presentes na ferramenta implementada. Essas limitações podem ser sanadas, em um projeto futuro que promova continuidade ao presente trabalho. Algumas limitações identificadas são as seguintes:

- A ferramenta TPTest não realiza o armazenamento do processo de priorização em uma base de dados. As informações do processo são armazenadas em arquivos localizados no disco rígido da máquina na qual a ferramenta está em execução. Isso ocorre pelo fato de ser uma implementação inicial onde o uso de arquivos para o armazenamento dos dados supriu as necessidades.

- Na etapa de importação das classes a ferramenta se limita a importar classes que estejam

presentes em arquivos que sigam o padrão apresentado pela Figura 6.6. Talvez em pesquisas futuras possa ser melhorada a importação de classes permitindo a leitura de classes presentes em novos formatos de arquivos.

- O arquivo contendo as classes a serem importadas pela ferramenta, informado pelo usuário, deve estar localizado no mesmo diretório em que a ferramenta se encontra ou ter todo o seu endereço informado. Essa é uma limitação presente na versão atual pelo fato da implementação do componente localizador de arquivos não ser o foco do trabalho.

- A versão atual da ferramenta não permite a escolha pelo usuário do local para armazenar o documento contendo o resultado do processo de priorização. Automaticamente, o documento é armazenado no mesmo diretório em que a ferramenta está armazenada.

- Na seção 5.1 foi apresentado que o método AHP é capaz de representar os problemas de forma hierárquica, através da definição do objetivo, dos critérios e das alternativas para alcançar o objetivo. A TPTest, em sua versão atual, se limita a analisar apenas os critérios, não percorrendo o nível de alternativas.

6.3 Estudo de Caso

O objetivo desta seção é apresentar um estudo piloto realizado com a finalidade de avaliar o processo e a ferramenta de priorização de classes para testes unitários. A priorização foi realizada em classes pertencentes a um software *open source*, encontrado no *sourceForge*[9], um portfólio centralizado de desenvolvedores de software para controlar e manter o desenvolvimento de *open sources*, o qual atua como um repositório de código fonte.

Esta Seção inicia com uma descrição do cenário em que o estudo piloto foi realizado. Em seguida é apresentado um relato do estudo de caso e, por fim, uma avaliação dos resultados obtidos é mostrada.

6.3.1 Cenário de Aplicação

O estudo de caso foi realizado com a avaliação das classes pertencentes ao software *JFreeChart*[10], uma biblioteca Java que auxilia os desenvolvedores na apresentação de gráficos em suas aplicações. É uma API consistente e bem documentada, suportando vários tipos de gráficos,

[9] sourceForge: http://www.sourceforge.net
[10] JFreeChart: http://www.jfree.org/jfreechart/

como gráficos de pizza (2D e 3D), gráficos de barras (horizontal e vertical, regular e empilhadas), gráficos de linhas, gráficos de dispersão, gráficos de séries temporais, entre outros.

JFreeChart possui licença LGPL (*GNU Lesser General Public License* - Licença Pública Geral Menor), licença que tem por objetivo regulamentar o uso de bibliotecas de código, podendo também ser aplicada na regulamentação de aplicações, permitindo a associação com software que não esteja sob a licença, como sistemas proprietários. É bastante portável podendo ser usada em aplicações desktop, *applets, servlets* e JSP (*Java Server Pages*).

6.3.2 Relato do Estudo de Caso

Com base nas atividades de execução da ferramenta TPTest, proposta por esse trabalho, apresentadas pelo diagrama de atividades na Figura 6.13, inicialmente foram definidas as informações do projeto de software das classes que se desejava priorizar para aplicar os testes, nesse caso, informações relacionadas a biblioteca *JFreeChart*. A Tabela 6.1 mostra os dados informados sobre o projeto.

TABELA 6.1:
Dados do projeto informados no processo de priorização

Campo	**Dado informado**
Nome do Projeto	JFreeChart
Descrição do Projeto	Biblioteca Java que auxilia os desenvolvedores na apresentação de gráficos em suas aplicações
Gerente do Projeto	Mr Gilbert - (responsável pela manutenção da biblioteca)

Após os dados do projeto terem sido informados, pôde-se então passar para a etapa de definição dos critérios. Nessa etapa, foram definidos os seguintes critérios: MLOC (Linhas de código nos métodos), WMC (Número de métodos em uma classe), NSC (Número de filhos) e, NSF (Número de atributos estáticos).

Com a etapa de definição de critérios concluída, avançando o processo, foram informadas as importâncias para as comparações pareadas entre critérios. Como foram definidos quatro critérios, a ferramenta criou uma matriz de comparações contendo quatro linhas e quatro colunas, necessitando a informação de seis graus de importância, dado que dois critérios só podem ser comparados pareadamente uma vez, como apresentado na Seção 6.2.2. As importâncias definidas para as comparações do experimento são apresentadas pela Tabela 6.2.

A partir destes graus de importâncias, definidos para as comparações pareadas, a TPTest

TABELA 6.2:
Comparações e suas importâncias

Critérios	Grau	Descrição
MLOC / WMC	1/2	WMC é ligeiramente mais importante do que MLOC
MLOC / NSC	1/2	NSC é ligeiramente mais importante do que MLOC
MLOC / NSF	3	MLOC é fracamente mais importantes do que NSF
WMC / NSC	1	WMC possui a mesma importância que NSC
WMC / NSF	5	WMC é moderadamente mais importante do que NSF
NSC / NSF	3	NSC é fracamente mais importantes do que NSF

criou a matriz de comparações (M):

$$M = \begin{bmatrix} 1 & 0.5 & 0.5 & 3 \\ 2 & 1 & 1 & 5 \\ 2 & 1 & 1 & 3 \\ 0.3333 & 0.2 & 0.3333 & 1 \end{bmatrix} \tag{6.2}$$

A matriz M (6.2) serviu, então, como entrada para o método AHP, que através de seus cálculos, retornou o vetor normalizado (V), vetor que permite a definição da função matemática para obtenção do grau de prioridade de cada classe. O vetor V obtido no estudo de caso, através da TPTest foi:

$$V = \begin{bmatrix} 0.1998 & 0.3789 & 0.3376 & 0.0836 \end{bmatrix} \tag{6.3}$$

Após a obtenção do vetor normalizado, como apresentado na Subseção 5.1.2, a ferramenta realizou a verificação do nível de consistência das comparações pareadas, a fim de garantir que as importâncias para as comparações foram definidas de forma consistente. Para o estudo de caso, o nível de consistência obtido foi de 1.8%, estando abaixo do limiar de 10% de consistência definido pelo método AHP.

A partir do vetor normalizado V6.3, a ferramenta TPTest gerou a função matemática para o cálculo do grau de prioridade. Cada critério de cada classe é multiplicado pelo seu correspondente na função matemática. A seguir é apresentada a função obtida no estudo de caso:

$$f(Cls) = (MLOC \times 0.1998) + (WMC \times 0.3789) + (NSC \times 0.3376) + (NSF \times 0.0836) \tag{6.4}$$

Seguindo no processo de priorização, com base na ferramenta, a próxima etapa foi a importação das classes a priorizar e a valoração dos critérios. O arquivo com extensão XML contendo as classes da biblioteca *JFreeChart*, extraído com o auxílio do *Metrics*, foi informado para a ferramenta, onde a mesma realizou a leitura das classes e de suas métricas através de um par-

ser XML. Como, para este estudo de caso, foram definidos como critérios apenas métricas que o *Metrics* extrai do código fonte, nenhum valor quantitativo para os critérios necessitou ser informado, todos os critérios de cada classe foram valorados automaticamente.

Com as classes importadas, a ferramenta calculou o grau de prioridade de cada classe em relação as demais, baseada na função matemática para o cálculo do grau de prioridade (6.4). Com isso, passou-se para a última etapa do processo de priorização, a visualização dos resultados. Nessa etapa a ferramenta apresentou os resultados e também gerou relatório contendo as informações do processo de priorização de classes para testes unitários realizado. Os resultados e o documento gerado com as informações são apresentados na subseção 6.3.3, a seguir.

6.3.3 Análise dos resultados

Ao final do processo de priorização de classes realizado pela ferramenta TPTest, para o estudo de caso, obteve-se a lista de classes em ordem para a aplicação dos testes unitários. Como apresentado na seção 6.1, a ordem das classes é obtida através da aplicação de uma ordenação baseada no grau de prioridade de cada classe. A Tabela 6.3 apresenta dados extraídos do processo de priorização e seus respectivos valores obtidos como resultado.

TABELA 6.3:
Dados e valores obtidos como resultado

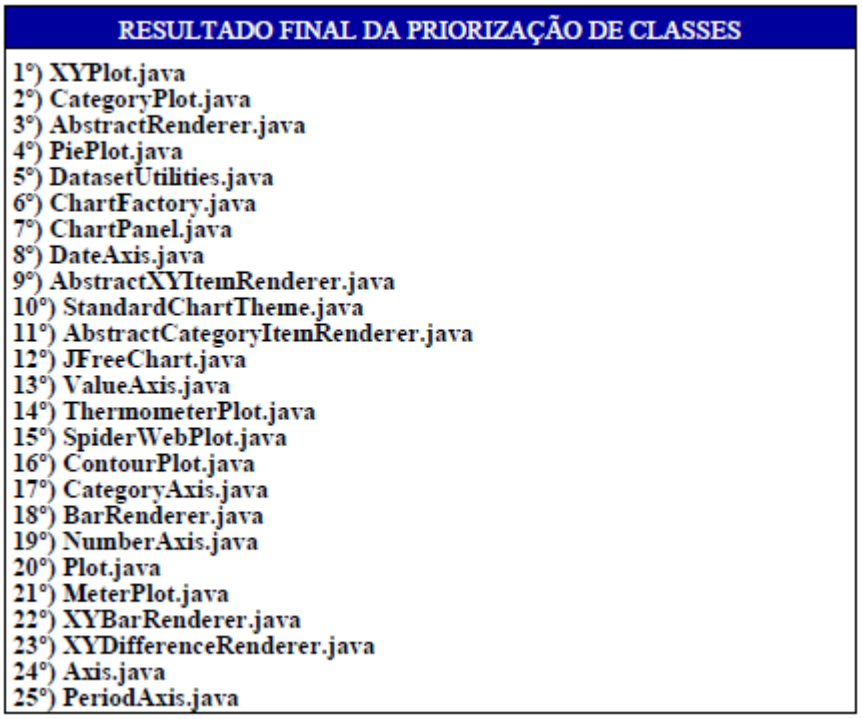
Dado	**Valor**
Número de critérios definidos	4
Número de importâncias definidas	6
Número de iterações do AHP	2
Número de classes importadas	870
Número de páginas de documentação	156

A Tabela 6.4 apresenta uma lista com 25 classes prioritárias para a aplicação dos testes, a posição de cada classe na ordem de classes em relação ao conjunto de classes e o respectivo grau de prioridade, além dos valores dos critérios que determinaram a posição da classe, com base na função matemática (6.4). Essas classes são uma amostra do conjunto de 870 classes importadas pela ferramenta e priorizadas. Essas classes estão localizadas no topo da listagem de classes ordenadas de forma decrescente em relação ao grau de prioridade. Vale salientar que esta listagem é baseada nos critérios e importâncias definidas para este estudo de caso em específico. Para novos processos de priorização, com novos critérios e importâncias, a listagem certamente sofreria alterações. O APÊNDICE A apresenta outro estudo de caso, onde as classes da própria TPTest foram priorizadas.

TABELA 6.4:
Lista de classes ordenadas baseada no grau de prioridade

Nome da Classe	Valor MLOC	Valor WMC	Valor NSC	Valor NSF	Grau de prioridade	Posição
XYPlot.java	2962	682	2	7	851.5198	1
CategoryPlot.java	2587	576	4	10	737.3514	2
AbstractRenderer.java	2057	396	4	11	563.3290	3
PiePlot.java	1703	310	2	15	459.6685	4
DatasetUtilities.java	1326	315	0	0	384.3074	5
ChartFactory.java	1545	156	0	1	367.8960	6
ChartPanel.java	1167	300	1	23	349.1156	7
DateAxis.java	1003	180	2	6	269.7903	8
AbstractXYItemRenderer.java	902	185	18	1	256.4886	9
StandardChartTheme.java	792	237	2	0	248.7297	10
AbstractCategoryItemRenderer.java	867	174	8	1	241.9508	11
JFreeChart.java	861	168	0	7	236.2792	12
ValueAxis.java	779	171	4	11	222.7171	13
ThermometerPlot.java	672	179	0	30	204.6082	14
SpiderWebPlot.java	693	152	0	13	197.1508	15
ContourPlot.java	630	176	0	3	192.8215	16
CategoryAxis.java	661	151	3	3	190.5547	17
BarRenderer.java	619	144	9	5	181.7033	18
NumberAxis.java	688	111	5	5	181.6341	19
Plot.java	545	161	13	12	175.2958	20
MeterPlot.java	607	137	0	10	174.0326	21
XYBarRenderer.java	587	139	5	3	171.8971	22
XYDifferenceRenderer.java	621	119	0	1	169.2562	23
Axis.java	577	136	2	16	168.8366	24
PeriodAxis.java	591	129	0	1	167.0515	25

Pelos dados apresentados na Tabela 6.4, percebe-se que tanto o processo de prorização quanto a ferramenta apresentaram resultados satisfatórios em relação aos objetivos propostos. Ao final do processo, o objetivo de gerar a lista ordenada com as classes prioritárias no topo da lista foi alcançado. A Figura 6.12 apresenta as 25 classes mais prioritárias do processo de priorização presentes na documentação gerada pela TPTest, as mesmas presentes na Tabela 6.4.


RESULTADO FINAL DA PRIORIZAÇÃO DE CLASSES

1°) XYPlot.java
2°) CategoryPlot.java
3°) AbstractRenderer.java
4°) PiePlot.java
5°) DatasetUtilities.java
6°) ChartFactory.java
7°) ChartPanel.java
8°) DateAxis.java
9°) AbstractXYItemRenderer.java
10°) StandardChartTheme.java
11°) AbstractCategoryItemRenderer.java
12°) JFreeChart.java
13°) ValueAxis.java
14°) ThermometerPlot.java
15°) SpiderWebPlot.java
16°) ContourPlot.java
17°) CategoryAxis.java
18°) BarRenderer.java
19°) NumberAxis.java
20°) Plot.java
21°) MeterPlot.java
22°) XYBarRenderer.java
23°) XYDifferenceRenderer.java
24°) Axis.java
25°) PeriodAxis.java

FIGURA 6.12: Amostra do resultado do estudo de caso presente na documentação

No próximo capítulo são apresentados os trabalhos futuros e as considerações finais do trabalho desenvolvido.

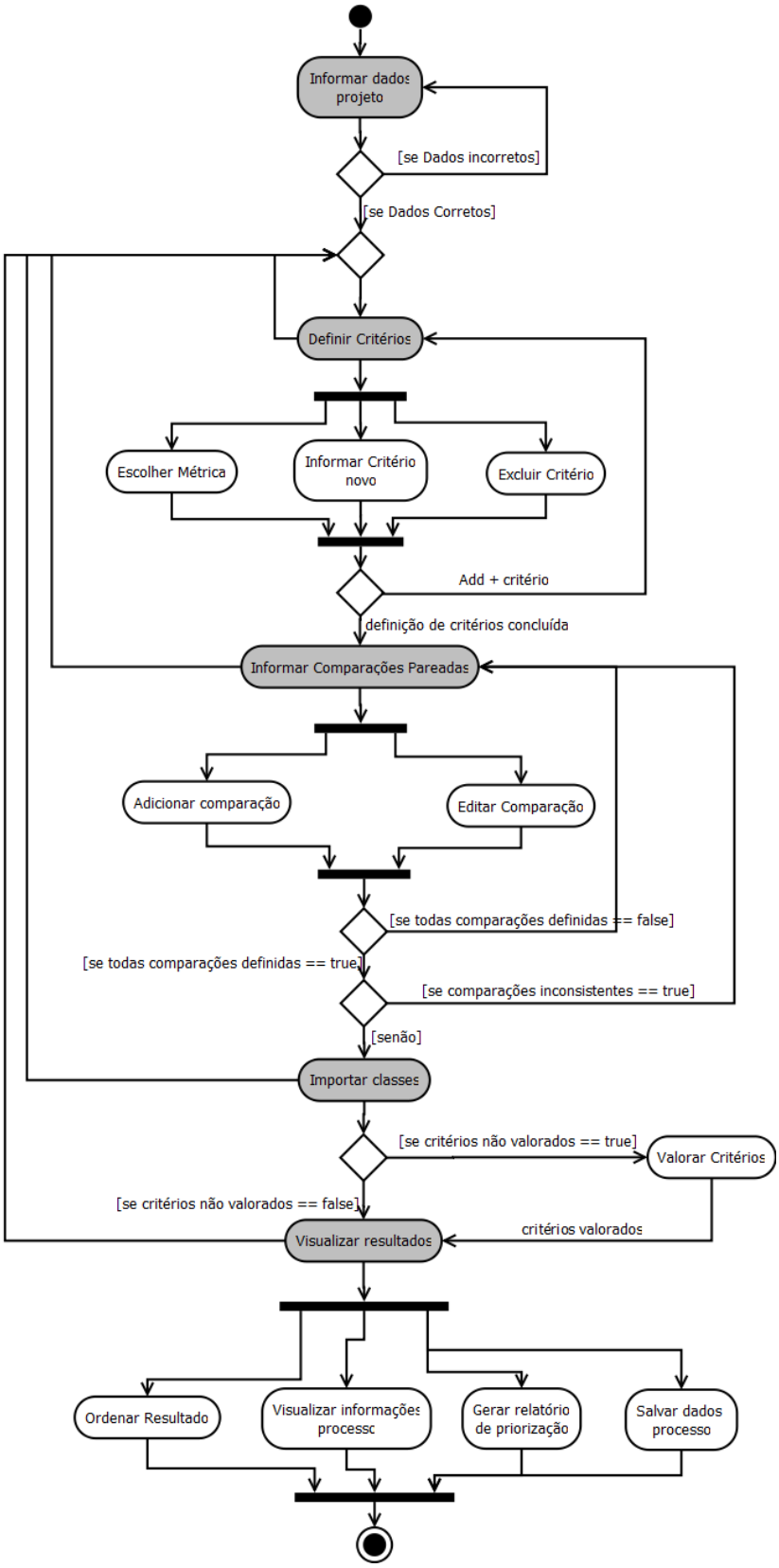


FIGURA 6.13: Diagrama de Atividades do processo de priorização de classes

7 *Considerações Finais*

Neste trabalho foi apresentada uma revisão bibliográfica com os trabalhos presentes na literatura que se relacionam com priorização de classes e testes de sofware, conceitos relativos a testes de software no paradigma de orientação a objetos, identificação de critérios de avaliação de qualidade para artefatos de software orientado a objetos, a apresentação de um método matemático para a análise par a par dessas métricas, a descrição de um processo de priorização de classes e a construção de uma ferramenta para a realização do processo de priorização de classes para testes unitários. Foram apresentados também um estudo de caso como forma de avaliação do processo de priorização e da ferramenta proposta.

Alguns critérios possíveis para a avaliação de artefatos de software são as métricas extraídas o próprio software. Essas métricas permitem medir o software em relação a vários critérios, como, por exemplo, o número de linhas de código-fonte, o número de métodos de uma classe, o nível na árvore de herança, entre outros. A extração dessas métricas, geralmente, não é uma tarefa simples e, em muitos casos, são usadas ferramentas que possibilitem essa extração.

Após a obtenção das métricas que realizam a mensuração do software que se deseja testar, deve acontecer uma análise dessas métricas. Alguns métodos matemáticos auxiliam nessa análise tornando critérios de avaliação, que em muitas vezes, são qualitativos e não quantitativos, em dados analíticos possíveis à comparação de forma semelhante. Um dos métodos de análise multicritério é o AHP, que realiza comparações par a par entre os critérios definidos. Aos critérios podem ser atribuídos graus de importância, auxiliando na tomada de decisão.

A priorização de testes é algo que tem sido cada vez mais empregado pelas empresas de desenvolvimento, pois auxiliam na automatização do processo de desenvolvimento e melhoram a fase de teste de software. Trabalhos na literatura focam seus estudos em priorização de testes de integração na tentativa da redução no número de *stubs* criados (simulação de classes ainda não testadas), mas a priorização de testes de unidade é muito importante também, pois como o tempo disponível para a execução dos testes pode ser limitado e curto, testar as classes mais importantes primeiro é o mais apropriado.

Mesmo apresentando resultados satisfatórios, tanto o processo quanto a ferramenta podem ser melhorados em uma série de pontos em trabalhos futuros que venham a ser realizados. Questões mais genéricas como características de acessibilidade podem ser inseridas na ferramenta, como garantia de que usuários com deficiência possam usar a ferramenta. Em um nível mais específico, fatores como a geração de um relatório contendo a ordem das classes em um formato XML para que possa servir como fonte de dados para outros sistemas, melhoria na fase de definição das importâncias informando ao usuário o ponto em que está acontecendo a inconsistência, caso ela exista, podem ser adicionadas ao trabalho. Outra melhoria possível de ser inserida, resultante de trabalhos futuros, é a construção de uma versão da TPTest que possa ser acoplada ao Eclipse IDE, pelo fato de as métricas serem extraídas através de um *plug-in* deste ambiente de desenvolvimento. Também, como alternativa para a melhoria do trabalho e do processo proposto, desenvolver um módulo de geração do arquivo XML que serve como base para a importação das classes, facilitando ao usuário a forma de inserir as classes no processo de priorização realizado com a TPTest. Por fim, outra possibilidade de trabalho futuro a ser realizado, é a validação da ferramenta realizada pelo usuário ou uma inspeção heurística de usar a TPTest, como forma de avaliar as metodologias propostas.

Na tentativa de contribuir com os estudos de melhoria do processo de desenvolvimento e priorização de testes, esse trabalho apresentou um processo de priorização de classes para testes de unidade em sistemas orientados a objetos baseada nas métricas que podem ser extraídas, apresentando um método matemático que pode quantificá-las analiticamente. Este trabalho realiza um estudo pouco explorado pela literatura, caracterizando-se como inovador na área de testes de software. Através do estudo de caso realizado com o apoio da ferramenta TPTest, percebe-se que as classes ordenadas em relação ao seu grau de importância obtido através da avaliação de múltiplos critérios garantem uma organização para a fase de teste de software, servindo de guia para os testadores realizarem a verificação de qualidade nos serviços a serem prestados pelo sistema em análise, sendo essas contribuições trazidas pelo trabalho.

Referências Bibliográficas

BEN, F. **Utilização do Método AHP em super Decisões de Investimento Ambiental**. *XXVI Encontro Nacional de Engenharia de Produção e XII International Conference on Industrial Engineering and Operations Management*, Fortaleza - CE, p. 254 – 254, 2006.

BERNARDO, P. C.; KON, F. **A Importância dos Testes Automatizados - Controle ágil, rápido e confiável de qualidade**. *Engenharia de Software Magazine*, p. 54 – 57, Julho 2008.

BESTEIRO, A. M. et al. **A Utilização do método AHP para traçar, como ferramenta para o auxilio a decisão de um candidato, a escolha de um curso de engenharia**. *VI Simpósio de Excelência em Gestão e Tecnologia - SEGeT*, Resende - RJ, 2009.

BOENTE, A. N. P.; OLIVEIRA, F. S. G. de; ALVES, J. C. N. **Obtenção da Qualidade de Software através do uso do RUP como Metodologia de Desenvolvimento**. *V Simpósio de Excelência em Gestão e Tecnologia*, Resende - RJ, 2008.

BRIAND, L. C.; MORASCA, S.; BASILI, V. R. **An Operational Process for Goal-Driven Definition of Measures**. *IEEE Transactions on Software Engineering*, p. 1106–1125, 2002.

BRITO, F.; CARAPUÇA, R. **Object - Oriented Software Engineering: Measuring and Controlling the Development Process**. *4th International Conference of Software Quality*, McLean - EUA, 1994.

BRUNTINK, M. **Predicting Class Testability using Object-Oriented Metrics**. In: *4th IEEE International Workshop on Source Code Analysis and Manipulation* SCAM 04. [S.l.]: Society Press, 2004. p. 136–145.

CARVALHO, S. C. et al. **Qualidade, Confiabilidade e Segurança nas Disciplinas do Processo Unificado**. *Simpósio Brasileiro de Engenharia de Software - SBES*, Campinas - SP, 2008.

CHIDAMBER, S.; KEMERER, C. **A Metrics Suite for Object Oriented Design**. *IEEE Transactions on Software Engineering*, IEEE Computer Society, Los Alamitos, CA, USA, p. 476–493, 1994.

COSTA, H. G. ***Introdução ao Método de Análise Hierárquica (Análise Multicritério no Auxílio À Decisão)***. Dissertação (Mestrado) — Universidade Federal Fluminense, Niterói, RJ , Brasil, 2002.

CRESPO, A. N. et al. **Uma Metodologia para Teste de Software no Contexto da Melhoria de Processo**. *III Simposio Brasileiro de Qualidade de Software - SBQS*, Brasília - DF, 2004.

ELIAS, G. da S.; WILDT, D. **Métricas para melhoria contínua de código: Um estudo de caso com Java**. *Seminário de Informática - RS (SEMINFO RS'2008)*, Torres-RS.

FANTINATO, M. et al. **AutoTest - Um Framework Reutilizável para a Automação de Teste Funcional de Software**. *Simpósio Brasileiro de Qualidade de Software - SBQS*, Porto de Galinhas - PE, 2007.

HORSTMANN, C. S.; CORNELL, G. ***Core Java***. 8. ed. São Paulo - SP: Pearson Prentice Hall, 2008.

KIM, Y. G. et al. **Test Cases Generation from UML State Diagrams**. In: *IN IEE PROCEEDINGS: SOFTWARE*. [S.l.: s.n.], 1999. p. 187–192.

KNUTH, D. E. ***The Art of Computer Programming***. 7. ed. [S.l.]: Adisson-Wesley, 1997.

LIMA, G. M. P. S.; TRAVASSOS, G. H. **Testes de Integração Aplicados a Software Orientado a Objetos: Heurísticas para Ordenação de Classes**. *III Simposio Brasileiro de Qualidade de Software - SBQS*, Brasília - DF, 2004.

LIMA, H. S.; ALVES, E. L. G.; RAMALHO, F. **Seleção e Geração Automática de Casos de Teste a partir de Diagramas de Máquina de Estados Comportamentais da UML 2**. *I Brazilian Workshop on Systematic and Automated Software Testing - SAST*, João Pessoa - PB, 2007.

LOPES, T. M. P.; FERNANDES, C. T. **Planejamento Integrado das Atividades de Codificação e Testes em Orientação a Objetos no Nível de Granularidade dos Métodos**. *X Workshop Iberoamericano de Ingenieria de Requisitos y Ambientes de Software - IDEAS*, Isla de Margarita - Venezuela, 2007.

OLIVEIRA, R. B. de; GóIS, F. N. B.; FARIAS, P. P. M. **Automação de Testes Funcionais: Uma Experiência do SERPRO**. *I Brazilian Workshop on Systematic and Automated Software Testing - SAST*, João Pessoa - PB, 2007.

PARIZI, R. B.; VIEIRA, V. G. **Teste de Navegação do Sistema de Emissão de Relatórios Acadêmicos com o uso de Selenium IDE**. *I Seminário Internacional Integrado de Ensino, Pesquisa e Extensão - SIC*, Uruguaiana - RS, 2009.

PIVETA, E. K. ***Improving the search for refactoring opportunities on object-oriented and aspect-oriented software***. Tese (Doutorado) — Universidade Federal do Rio Grande do Sul - UFRGS, Porto Alegre - RS, 2009. Disponível em: <http://hdl.handle.net/10183/15651>.

PRESSMAN, R. S. ***Engenharia de software***. 6. ed. São Paulo - SP: McGraw-Hill, 2006.

ROTHERMEL, G. et al. **Prioritizing Test Cases For Regression Testing**. *IEEE Transactions on Software Engineering*, IEEE Computer Society, Los Alamitos, CA, USA, v. 27, p. 929–948, 2001.

SAATY, T. L. **How to make a decision: The analytic hierarchy process**. *European Journal of Operational Research*, v. 48, n. 1, p. 9–26, September 1990. Disponível em: <http://ideas.repec.org/a/eee/ejores/v48y1990i1p9-26.html>.

SOMMERVILLE, I. ***Engenharia de Software***. 8. ed. São Paulo - SP: Pearson Prentice Hall, 2007.

SRIKANTH, H.; WILLIAMS, L. **On the economics of requirements-based test case prioritization**. *ACM SIGSOFT Software Engineering Notes*, p. 1–3, 2005.

SRIKANTH, H.; WILLIAMS, L.; OSBORNE, J. **System test case prioritization of new and regression test cases**. *International Symposium on Empirical Software Engineering (ISESE 2005)*, Noosa Heads - Australia, 2005.

SRIVASTAVA, P. R. **Test Case Priorization**. *Journal of Theoretical and Applied Information Technology*, p. 178 – 181, 2008.

TAHAT, L. H. et al. **Requirement-Based Automated Black-Box Test Generation**. *Computer Software and Applications Conference, Annual International*, IEEE Computer Society, Los Alamitos, CA, USA, 2001.

VEIGA, A. Barbosa da; FARNESE, R.; MELO, W. **JMetrics Java Metrics Extractor: An Overview**. Universidade de Brasília, Departamento de Ciência da Computação - Brasília - DF, 1999.

Apêndice

APÊNDICE A – Priorização das classes da ferramenta TPTest

Um segundo estudo de caso foi realizado como forma de avaliação do processo e da ferramenta de priorização de classes para testes unitários. Nesse estudo de caso, as próprias classes da ferramenta TPTest, proposta por este trabalho, foram priorizadas.

Através da execução das etapas do processo de priorização proposto, guiadas pela ferramenta TPTest, inicialmente foram definidas as informações correspondentes ao projeto a ser priorizado. A Tabela A.1 apresenta os dados informados para a ferramenta:

TABELA A.1:
Dados do projeto informados no processo de priorização

Campo	Dado informado
Nome do Projeto	TPTest
Descrição do Projeto	Projeto da própria ferramenta
Gerente do Projeto	Rafael Parizi

Posteriormente foram definidos os critérios de avaliação das classes. A Tabela A.2 mostra os critérios definidos para este estudo de caso.

TABELA A.2:
Critérios definidos

Critérios
NOM (Número de Métodos)
MLOC (Linhas de Código em Método)
DIT (Profundidade da Árvore de Herança)

Definidos os critérios, foram informadas as importâncias para as comparações entre esses critérios: NOM é fracamente mais importante do que MLOC (NOM/MLOC = 3); DIT é fraca a moderadamente mais importante do que NOM (NOM/DIT = 1/4); DIT é fracamente mais importante do que MLOC (MLOC/DIT = 1/3).

Através dos cálculos realizados pelo método AHP, a função matemática obtida para o cálculo do grau de prioridade de cada classe em relação as demais, foi a seguinte:

$$f = (NOM \times 0.3445716) + (MLOC \times 0.10856789) + (DIT \times 0.5468605)$$

A próxima etapa, importação das classes, foi realizada informando-se o arquivo XML que contém as classes e as métricas extraídas com auxílio do *framework Metrics* do projeto da ferramenta TPTest. Com as classes importadas, juntamente com a função matemática que calcula o grau de prioridade, a TPTest gerou a ordenação das classes. A Tabela A.3 apresenta alguns dados gerados como resultado desse estudo de caso.

TABELA A.3:
Dados e valores obtidos como resultado

Dado	**Valor**
Número de critérios definidos	3
Número de importâncias definidas	3
Número de iterações do AHP	2
Número de classes importadas	15
Número de páginas de documentação	6

A Tabela A.4 apresenta as classes em ordem decrescente segundo o valor de prioridade obtido no estudo de caso. A primeira coluna da Tabela apresenta o nome da classe, a segunda o valor da classe para o critério NOM, a terceira, o valor da classe para o critério MLOC, a quarta, o valor da classe para o critério DIT, e por fim, a quinta coluna apresenta o valor de prioridade obtida para cada classe.

TABELA A.4:
Lista de classes ordenadas baseada no grau de prioridade

Classe	**NOM**	**MLOC**	**DIT**	**Prioridade**
TPTest.java	37.0	2067.0	6.0	240.4401
ArmazenaDados.java	13.0	208.0	1.0	27.6084
ResultadoTemp.java	7.0	200.0	1.0	24.672
FrameConfiguraProjeto.java	4.0	168.0	6.0	22.8988
LeituraArquivoMetricas.java	6.0	167.0	1.0	20.7451
FrameSobre.java	3.0	136.0	6.0	19.0801
ImprimeDadosArquivo.java	1.0	158.0	1.0	18.0451
RankingAHP.java	5.0	135.0	1.0	16.9263
ClasseBean.java	33.0	46.0	1.0	16.9118
FrameVisualizaDadosProjeto.java	4.0	81.0	6.0	13.4534
ComparacaoPareadaTest.java	7.0	64.0	1.0	9.9072
ComparacaoPareadaBean.java	8.0	8.0	1.0	4.1719
ClasseValorada.java	6.0	6.0	1.0	3.2656
Criterio.java	6.0	6.0	1.0	3.2656
ProjetoBean.java	6.0	6.0	1.0	3.2656

APÊNDICE B – Documentação do Processo de Priorização de Classes

Conforme apresentado no Capítulo 6, o usuário pode gerar uma documentação do processo de priorização de classes através da ferramenta TPTest. Esse documento apresenta uma série de informações pertinentes à priorização realizada, tais como: os dados do projeto, as classes importadas, os critérios definidos, entre outros.

Para geração do documento é utilizada a biblioteca iText[11], uma biblioteca Java que permite a criação e manipulação de documentos PDF (*Portable Document Format*). A iText pode exportar o mesmo documento em vários formatos ou múltiplas instâncias do mesmo formato. Os dados podem ser gravados em um arquivo, ou por exemplo, a partir de um *servlet* para um navegador web. Primeiramente, cria-se um objeto da classe *"Document"* que representará o documento e após adiciona-se parágrafos e frases a esse documento. O algoritmo 1 apresenta um exemplo de código fonte utilizado para a construção de um documento PDF através da biblioteca iText.

ALGORITMO 1 Exemplo de código da biblioteca iText

```
public class ITextHelloWorld {
    public static void main(String args[]) {
        try {
            Document document = new Document();
            PdfWriter.getInstance(document, new FileOutputStream("HelloWorld.pdf"));
            document.open();
            document.add(new Paragraph("Hello World"));
            document.close();
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

A ferramenta TPTest possui uma classe responsável pela geração da documentação do processo de priorização. Essa classe utiliza os recursos fornecidos pela biblioteca iText para a criação do documento no formato PDF. O algoritmo 2 apresenta um trecho de código da classe "ImprimeDadosArquivo", classe que realiza a geração do arquivo de documentação:

O documento gerado pela ferramenta TPTest como resultado do processo de priorização é composto por uma página de apresentação, uma página com os dados do projeto e as classes importadas para a ferramenta e que foram priorizadas, além de páginas contendo informações do processo de priorização e o resultado contendo a ordem de classes para a aplicação de testes de unidade. As figuras B.1, B.2, B.3 e B.4 mostram um documento gerado pela TPTest.

[11] iText:http://itextpdf.com/

ALGORITMO 2 Trecho de código da classe ImprimeDadosArquivo.java

```
public class ImprimeDadosArquivo {
    public void imprimirArquivo(ArrayList <String>dados) {
        Document document = new Document();
        try {
            PdfWriter.getInstance(document, new FileOutputStream("TPTest.pdf"));
             document.open();
            document.addTitle("TPTest");
            Image jpg = Image.getInstance("header2.png");
            jpg.setAlignment(Image.ALIGN_CENTER);
            document.add(jpg);
            Phrase phrase1 = new Phrase("Priorização de Classes para Testes Unitários
baseada em Multicritérios", FontFactory.getFont(FontFactory.HELVETICA, 16,
Font.BOLD));
                                                ⋮
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

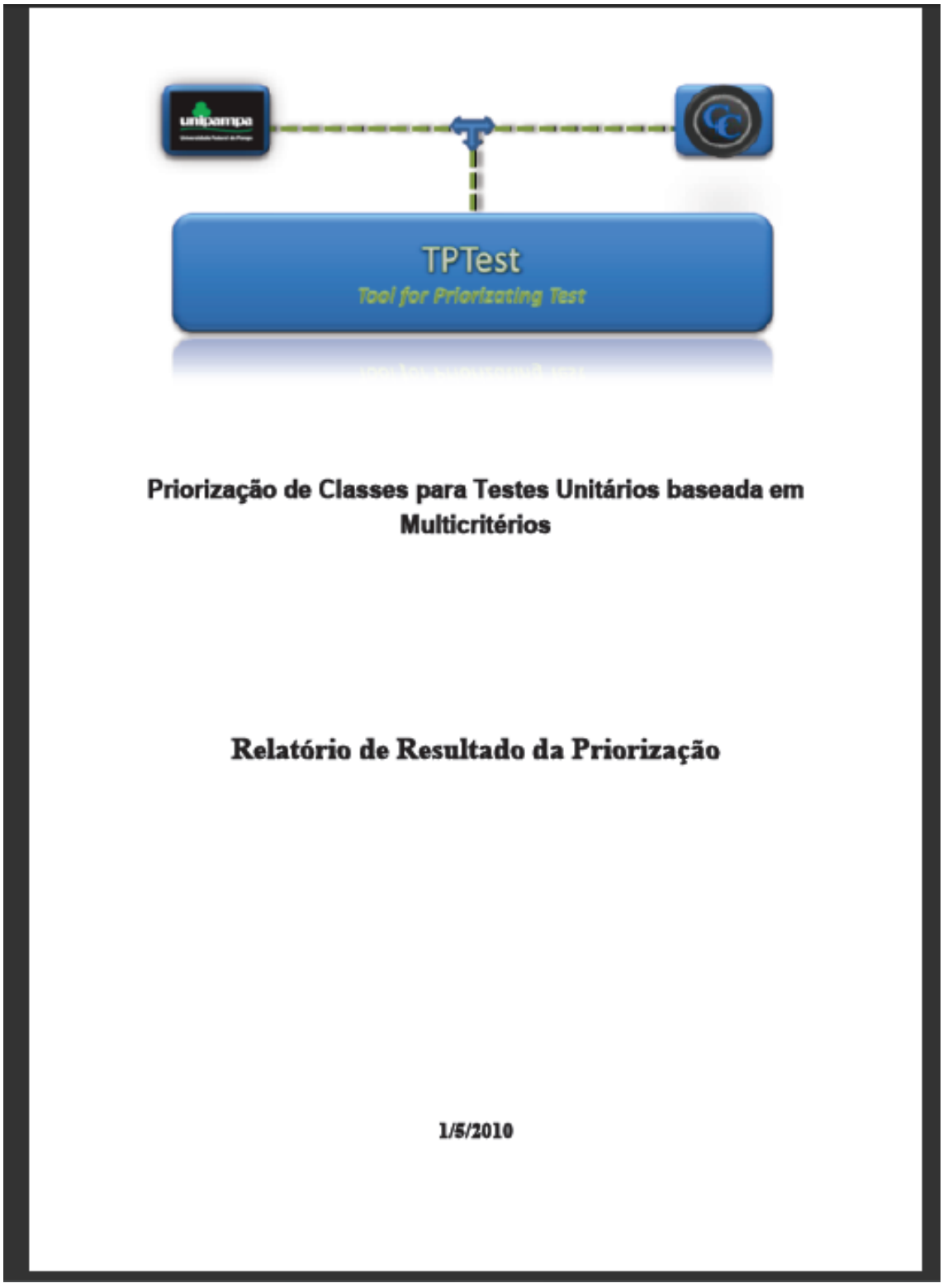
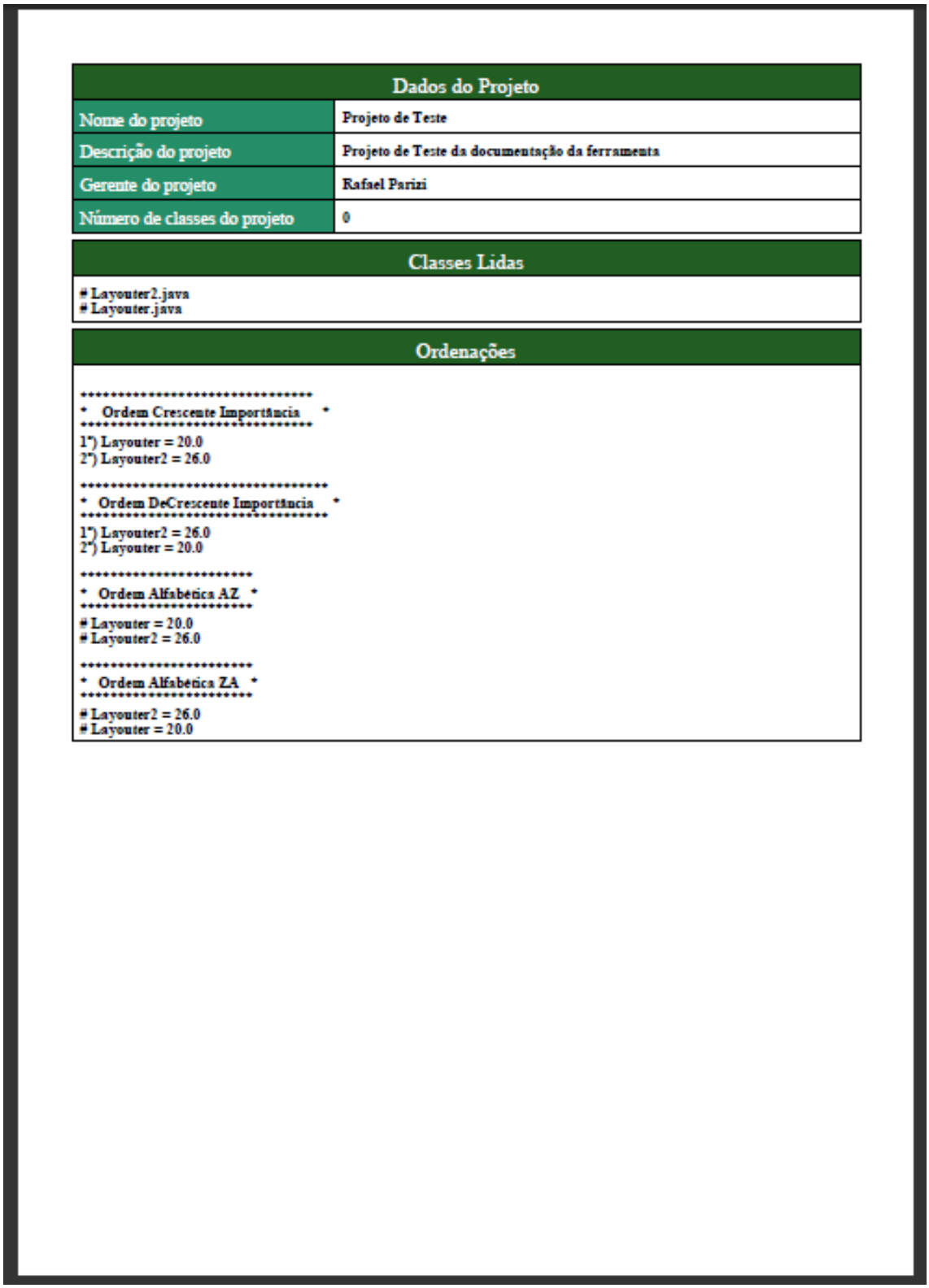


FIGURA B.1: Página contendo a identificação do processo de priorização



Dados do Projeto	
Nome do projeto	Projeto de Teste
Descrição do projeto	Projeto de Teste da documentação da ferramenta
Gerente do projeto	Rafael Parizi
Número de classes do projeto	0

Classes Lidas

Layouter2.java
Layouter.java

Ordenações

****
* Ordem Crescente Importância *
****
1°) Layouter = 20.0
2°) Layouter2 = 26.0

****
* Ordem DeCrescente Importância *
****
1°) Layouter2 = 26.0
2°) Layouter = 20.0

****
* Ordem Alfabetica AZ *
****
Layouter = 20.0
Layouter2 = 26.0

****
* Ordem Alfabetica ZA *
****
Layouter2 = 26.0
Layouter = 20.0

FIGURA B.2: Página contendo os dados do projeto que as classes estão sendo priorizadas

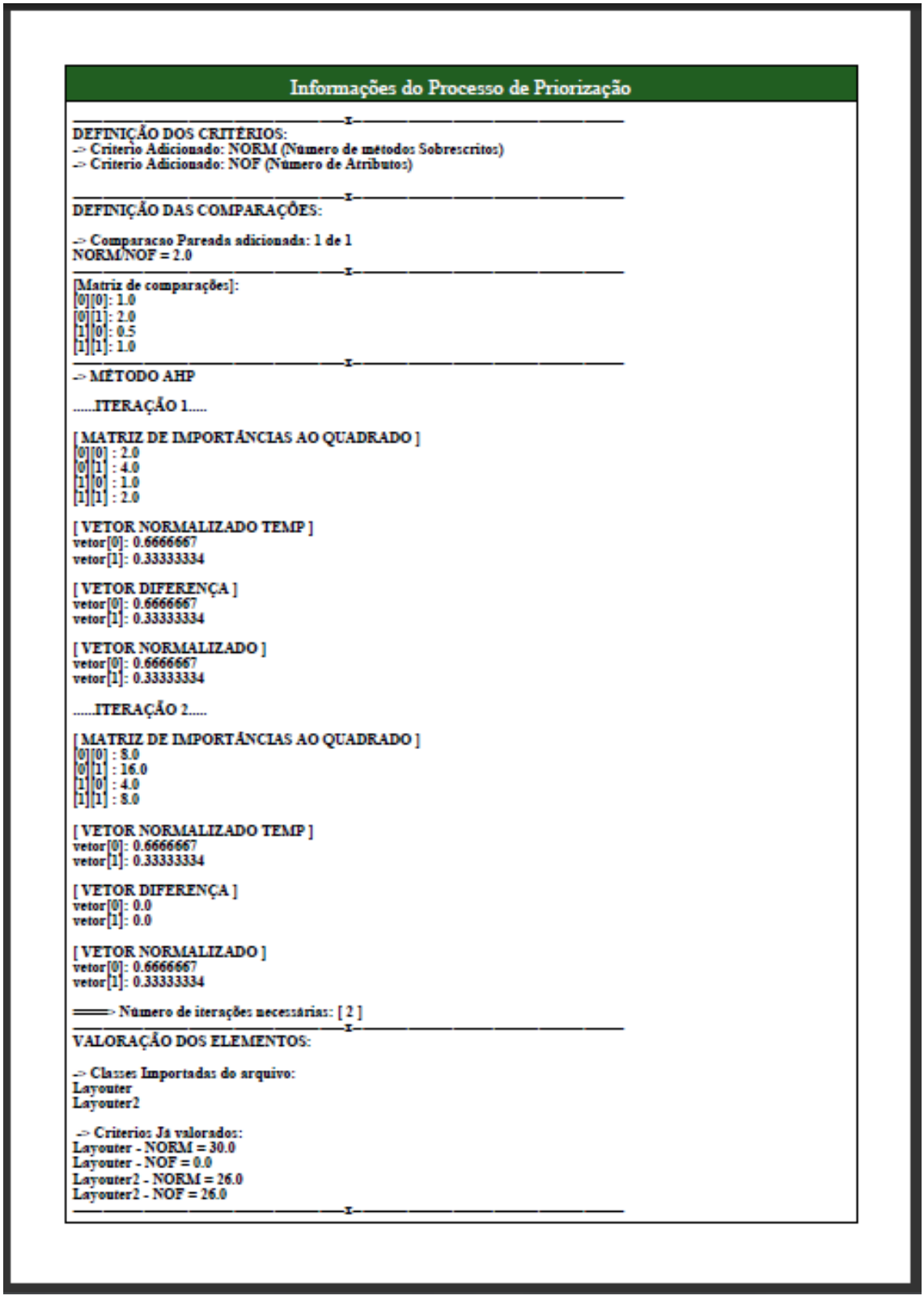


FIGURA B.3: Página contendo informações do processo de priorização

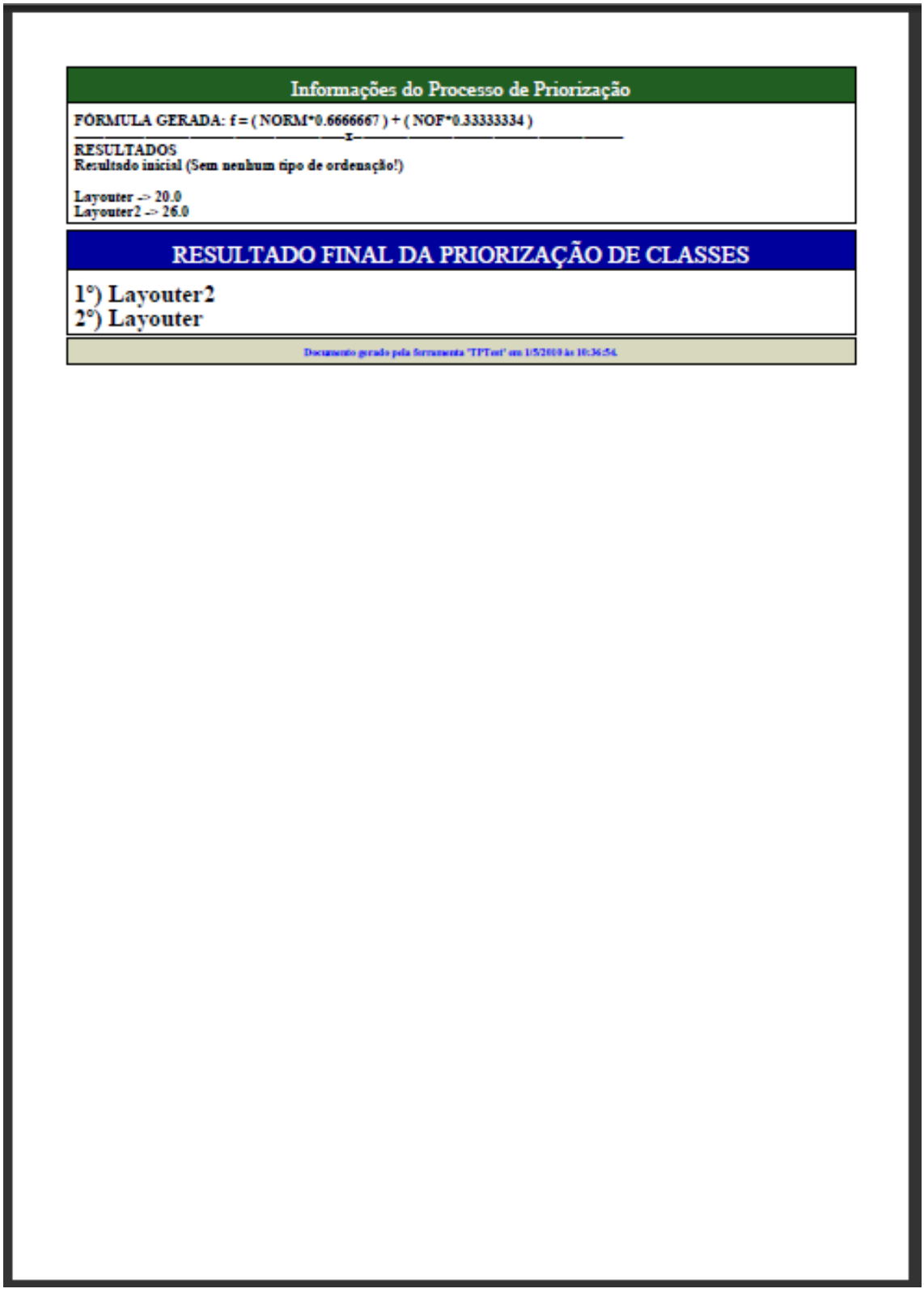


FIGURA B.4: Página contendo o resultado da priorização