CAMILA BASTOS

UMA ABORDAGEM PARA VISUALIZAÇÃO DA EVOLUÇÃO DE CÓDIGO MORTO EM SISTEMAS DE SOFTWARE ORIENTADOS A OBJETOS

LAVRAS-MG
2017

CAMILA BASTOS

UMA ABORDAGEM PARA VISUALIZAÇÃO DA EVOLUÇÃO DE CÓDIGO MORTO EM SISTEMAS DE SOFTWARE ORIENTADOS A OBJETOS

Dissertação apresentada à Universidade Federal de Lavras, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, área de concentração em Engenharia de Software, para a obtenção do título de Mestre.

Prof. Dr. Heitor Augustus Xavier Costa
Orientador

LAVRAS-MG
2017

Ficha catalográfica elaborada pelo Sistema de Geração de Ficha Catalográfica da Biblioteca Universitária da UFLA, com dados informados pelo(a) próprio(a) autor(a).

Bastos, Camila.
Uma abordagem para Visualização da Evolução de Código Morto em Sistemas de Software Orientados a Objetos / Camila Bastos. - 2017.
153 p.

Orientador(a): Heitor Augustus Xavier Costa.

Dissertação (mestrado acadêmico) - Universidade Federal de Lavras, 2017.
Bibliografia.

1. Código Morto. 2. Visualização de Software. 3. Evolução de Software. I. Xavier Costa, Heitor Augustus. II. Título.

CAMILA BASTOS

UMA ABORDAGEM PARA VISUALIZAÇÃO DA EVOLUÇÃO DE CÓDIGO MORTO EM SISTEMAS DE SOFTWARE ORIENTADOS A OBJETOS

DEAD CODE EVOLUTION VISUALIZATION APROACH IN OBJECT ORIENTED SOFTWARE

Dissertação apresentada à Universidade Federal de Lavras, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, área de concentração em Engenharia de Software, para a obtenção do título de Mestre.

APROVADA em 17 de fevereiro de 2017.
Dra. Cláudia Maria Lima Werner - UFRJ
Dr. Rodrigo Pereira dos Santos - UNIRIO
Dr. Paulo Afonso Parreira Júnior - UFLA
Dr. Rafael Serapilha Durelli - UFLA

Prof. Dr. Heitor Augustus Xavier Costa
Orientador

LAVRAS-MG
2017

AGRADECIMENTOS

Primeiramente, agradeço à Deus por ter me dado saúde e determinação para superar os desafios, por saber fazer das dificuldades oportunidades de aprendizado e pela força em sempre continuar lutando.

Aos meus pais, Sebastião e Dulcinéa, pelo amor, dedicação, incentivo, torcida e orações. Agradeço também pelo apoio e conselhos nos momentos de dificuldade. Aos meus avós, Luzia e João, pela torcida, palavras de carinho e orações.

Ao meu noivo Ênio, pelo companheirismo, amor, carinho, dedicação e apoio. Por sempre se fazer presente nos momentos difíceis. Por nunca me deixar sozinha. Por dar seu melhor para me ver feliz. Sem você tudo seria mais difícil.

À minha irmã Elaine e ao meu cunhado Paulo César, pela amizade, carinho, incentivo e torcida.

Ao professor Heitor Costa, pela orientação, dedicação e ensinamentos durante a realização deste trabalho.

Aos meus colegas de mestrado, especialmente Mariana, Danilo e João Antônio, pela agradável companhia diária e troca de ensinamentos.

Às pessoas que contribuíram com a avaliação deste trabalho, especialmente aos alunos que participaram do estudo experimental, aos revisores anônimos dos artigos e as bancas avaliadoras.

À Universidade Federal Lavras, especialmente aos funcionários do Departamento de Ciência da Computação, por conceder recursos necessários para realização deste trabalho.

À Coordenação de Aperfeiçoamento de Pessoal do Nível Superior - CAPES, pelo apoio financeiro.

Muito obrigada!

RESUMO

A evolução é fundamental para que os sistemas de software atendam as necessidades de seus usuários. A compreensão dessa evolução pode ser utilizada para encontrar a origem de problemas atuais ou para obter informações que possibilitam prever características futuras do software. No entanto, o aumento das informações, das funções, da poluição e da quantidade de código morto ao longo da evolução deixa o software mais complexo, dificultando essa compreensão. Desse modo, técnicas de visualização de software tem sido utilizadas para representar a evolução de determinados atributos do software, facilitando a compreensão de suas características evolutivas. Apesar dessas técnicas, ainda existem lacunas ao explorá-las para compreender fatores que contribuem com o aumento da complexidade ao longo da evolução, por exemplo, a presença de código morto. Com base nesses fatores, neste trabalho foi proposta DCEVizz, uma abordagem para identificar e visualizar código morto na evolução de sistemas de software orientados a objetos. A finalidade dessa abordagem é prover melhor percepção da existência de código morto nas versões do software, assim como suas características evolutivas. De modo geral, a análise e compreensão da evolução do código morto pode fornecer informações úteis para predizer o aumento desse código nas versões futuras, aumentar a segurança na sua eliminação e facilitar sua identificação nas versões. A abordagem foi implementada em um *plug-in* para a plataforma IDE Eclipse (DCEVizz Tool) e avaliada em duas fases. Na primeira fase, foi realizada uma comparação qualitativa do código morto identificado por DCEVizz Tool com o identificado pela ferramenta Understand. Essa comparação mostrou que a técnica de detecção utilizada em DCEVizz Tool identificou maior quantidade de código morto do que Understand em todos os sistemas analisados. Na segunda fase da avaliação, foi realizado um estudo experimental com um grupo de voluntários, que realizaram um conjunto de atividades sem e com o uso de DCEVizz Tool. Os resultados desse estudo indicaram que o uso de DCEVizz Tool aumenta a precisão e a eficiência na execução das atividades e que, na opinião dos participantes, a abordagem facilita a compreensão da evolução do código morto.

Palavras-Chave: Código morto. Evolução de Software. Visualização de Software.

ABSTRACT

The evolution is fundamental for the software to always meet the needs of their users. The comprehension of this evolution can be used to find the source of current problems or to obtain information that makes possible to predict future characteristics of the software. However, the increasing of information, functionality, pollution and the amount of dead code throughout evolution leaves the software more complex, making it difficult to comprehension. Thus, software visualization techniques have been used to represent the evolution of certain attributes of the software, facilitating the understanding of its evolutionary characteristics. Nevertheless, there are gaps in exploring these techniques to understand factors that contribute to increase the complexity throughout evolution, for example, the presence of dead code. Based on these factors, in this work the DCEVizz approach was proposed to identify and visualize dead code in the evolution of object oriented software. The purpose of this approach is to better understand the existence of dead code in software versions and their evolutionary characteristics. The analysis and understanding of dead code evolution can provide information to predict increases in this code, to enhance security in their elimination and to facilitate their identification in versions. The approach was implemented in a plug-in for Eclipse (DCEVizz Tool) and evaluated in two phases. In the first phase, a qualitative comparison of the dead code identified by DCEVizz Tool with the dead code identified by Understand tool was performed. This comparison showed that the detection technique used in DCEVizz Tool identified a greater amount of dead code than Understand in all the analyzed software. In the second phase of the evaluation, an experimental study was performed with a group of volunteers, who performed a set of activities using and not using the approach. The results of this study indicate that the use of DCEVizz increased precision and efficiency in the execution of the activities and facilitated the understanding of the evolution of the code.

Keywords. Dead Code. Software Evolution. Software Visualization.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE LISTAGENS

SUMÁRIO

1 INTRODUÇÃO

A evolução de sistemas de software ocorre por causa da execução de sucessivas atividades de manutenção, realizadas para correção de defeitos, para adição de novas funções ou para atualização de tecnologias (ATLEE; PFLEEGER, 2009; SOMMERVILLE, 2010; MAXIM; PRESSMAN, 2014). Essa evolução foi estudada na década de 70 por Lehman, resultando em um conjunto de leis que abordam algumas de suas causas e de suas consequências (BELADY; LEHMAN, 1985). Dentre essas leis, destaca-se a Lei da Mudança Contínua, definindo que sistemas de software devem evoluir constantemente para não se tornarem insatisfatórios.

A compreensão dessa evolução tem sido utilizada para encontrar a origem de problemas atuais ou para obter informações que possibilitam prever características futuras de sistemas de software (RATZINGER et al., 2007; D'AMBROS, 2008; CHAIKALIS; CHATZIGEORGIOU, 2015). Compreender a evolução demanda a análise do histórico desses sistemas, sendo considerada uma tarefa árdua por causa da quantidade significativa de dados a serem compreendidos. Além disso, existe tendência de aumento de trechos de código desnecessários ao logo da evolução (poluição de código), aumentando a complexidade e a dificuldade de compreensão (BURD; RANK, 2001; GOLD; MOHAN, 2003).

A relação entre evolução e complexidade foi abordada em uma das leis de Lehman, denominada Lei da Complexidade Crescente, afirmando que a evolução deixa os sistemas de software complexos, a menos que alguma ação seja realizada para diminuí-la. Além disso, a evolução causa "envelhecimento" do sistema de software, gerando versões com qualidade interna degradada e difíceis de serem compreendidas. Um dos fatores que contribui com o aumento da complexidade e da poluição é a presença de código morto, podendo ser definido como trechos de código desnecessários que não interferem na execução do sistema de software (BOOMSMA; GROSS, 2012).

Nesse contexto, técnicas de visualização de software têm sido utilizadas para representar a evolução de determinados atributos dos sistemas de software, facilitando a compreensão de suas características evolutivas (SCHOTS, 2011; CARNEIRO, 2013; NOVAIS et al., 2013a; MELANCON; RUFIANGE, 2014; FRANCESE et al., 2015). O uso dessas técnicas explora a capacidade de percepção humana em relação aos atributos visuais e apresenta visualmente apenas as informações de interesse. Além disso, essas técnicas facilitam a detecção de padrões que constam nas informações implícitas geradas ao longo dessa evolução.

1.1 Motivação

A poluição de sistemas de software é a principal causa do aumento da complexidade ao longo da sua evolução. O surgimento dessa poluição é impulsionado pela falta de familiaridade com o software pela maioria dos desenvolvedores envolvidos na manutenção, que implica em danos a estrutura original desses sistemas (LAYZELL; TJORTJIS, 2001). Além disso, a necessidade de adaptações para atender as exigências do mercado faz com que os desenvolvedores priorizem o cumprimento de prazos e de orçamentos, não se comprometendo com a qualidade interna e contribuindo com o aumento da poluição (ASH, 1994).

Código morto favorece o aumento da poluição, aumenta desnecessariamente o tamanho do código e dificulta a execução de testes, visto que podem ser testados trechos de código desnecessários. O código morto também prejudica identificar a implementação dos requisitos (rastreabilidade de requisitos) e prejudica a legibilidade, dificultando a compreensão (GUERROUAT; RICHTER, 2006; MARTINS et al., 2010). Sua presença é frequente em sistemas de software orientados a objetos, nos quais é enfatizado o comportamento dos objetos ao invés de detalhes de implementação (SRIVASTAVA, 1992). Além disso, código morto contribui para a manutenção ser considerada a etapa mais cara do ciclo de vida de sistemas de software, visto que aproximadamente metade do tempo utilizado nessa etapa é destinada à compreensão (DUCASSE; LANZA, 2005; SOMMERVILLE, 2010; SCANNIELLO, 2014).

O aumento da poluição e a presença de código morto impulsionou a definição de diferentes técnicas para visualizar e facilitar a compreensão da evolução de sistemas de software. A finalidade dessas técnicas é representar visualmente apenas a evolução de atributos de interesse, por exemplo, a evolução da arquitetura, do acoplamento, das linhas de código, dos atributos ou dos métodos (CASERTA; ZENDRA, 2011). Essa representação visual facilita a compreensão das características evolutivas desses atributos, visto que as demais informações desses sistemas (inclusive a poluição) são desconsideradas na visualização, fazendo com que o aumento da complexidade ao longo da evolução não interfira no processo de compreensão.

Apesar da existência de várias técnicas de visualização da evolução de sistemas de software, há lacunas para serem exploradas na compreensão de fatores que contribuem com o aumento da complexidade ao longo da evolução, por exemplo, a presença de código morto. A

visualização e a compreensão da evolução do código morto podem fornecer informações úteis para:

a) Facilitar a identificação de código morto nas versões de sistemas de software, bem como seu comportamento ao longo das versões;

b) Predizer aumento da quantidade de código morto nas versões futuras de sistemas de software, caso o tamanho desse código tenha aumentado continuamente ao longo das versões, fazendo com que medidas preventivas possam ser tomadas;

c) Aumentar a segurança na eliminação de código morto. Se determinado trecho de código é morto em diversas versões consecutivas, reforça o indicativo de sua inutilidade no sistema de software. Se o uso desse trecho de código oscila ao longo das versões, pode-se ter indícios que sua eliminação acarretará algum erro em futuras versões.

A compreensão dessas informações pode ser facilitada com a utilização de técnicas de visualização para representar características evolutivas do código morto. Desse modo, a motivação deste trabalho é auxiliar na legibilidade de versões de sistemas de software por meio da visualização da evolução de código morto, fornecendo informações que motivem a execução de medidas para reduzir a relação negativa que há entre a evolução e o aumento da poluição de sistemas de software ao longo das versões.

1.2 Objetivo

Neste trabalho, o objetivo é propor uma abordagem que provê melhor percepção das características evolutivas do código morto, utilizando técnicas de visualização de software. Mais especificamente, objetiva-se utilizar uma técnica de detecção de código morto para identificar métodos inacessíveis (mortos) em versões de sistemas de software orientados a objetos, analisar a evolução desses métodos e apresentar os resultados utilizando técnicas de visualização de software. Para atingir esse objetivo, foram definidos os seguintes objetivos específicos:

a) Investigar o estado da arte em relação as técnicas de detecção de código morto;

b) Selecionar/adaptar técnicas de detecção de código morto que possam ser utilizadas na abordagem proposta;

c) Investigar o estado da arte em relação as técnicas de visualização de software;

d) Selecionar/adaptar técnicas de visualização que possam representar a evolução do código morto;

e) Definir uma abordagem para identificar e visualizar código morto na evolução de sistemas de software orientados a objetos;

f) Desenvolver um apoio computacional (*plug-in* para a plataforma Eclipse IDE) que implemente a abordagem proposta;

g) Avaliar a abordagem proposta.

1.3 Estrutura do trabalho

O restante do trabalho está organizado da seguinte forma. O método de pesquisa utilizado na condução deste trabalho e uma visão geral dos principais resultados obtidos em cada etapa desse método são descritos no Capítulo 2. Alguns trabalhos relacionados com identificação de código morto e compreensão da evolução de sistemas de software estão resumidos no Capítulo 3. Referencial teórico sobre compreensão e visualização de software, além de técnicas de visualização e seus paradigmas, é apresentado no Capítulo 4. Breve referencial teórico sobre código morto e as principais técnicas identificadas em uma revisão sistemática de literatura são exibidos do Capítulo 5. Detalhes da abordagem proposta, das técnicas de visualização utilizadas e do *plug-in* desenvolvido são descritos no Capítulo 6. Avaliação qualitativa para verificar a capacidade de detecção de código morto da análise de acessibilidade, além de um estudo experimental para investigar os efeitos da abordagem proposta na compreensão da evolução desse código, é relatada no Capítulo 6. Algumas ameaças à validade são expostas no Capítulo 7. Conclusões, contribuições, limitações e trabalhos futuros são apresentadas no Capítulo 8.

2 MÉTODO DE PESQUISA

É importante classificar a pesquisa sob diferentes pontos de vista para melhor definição dos métodos utilizados neste trabalho (JUNG, 2009):

a) **Do ponto de vista da natureza**. Este trabalho consiste na realização de uma **pesquisa aplicada**, dirigida à geração de conhecimentos que possam ser aplicados na prática para contribuir com a manutenibilidade de sistemas de software orientados a objetos;

b) **Do ponto de vista da forma e da abordagem do problema**. O problema é abordado de maneira **qualitativa**, focando na identificação e na interpretação de fenômenos que ocorrem ao longo da evolução de um sistema de software e que prejudicam a sua manutenibilidade, como o surgimento de código morto;

c) **Do ponto de vista dos objetivos**. Foi realizada uma pesquisa **exploratória**, em que o conhecimento necessário para alcançar o objetivo foi obtido por meio de pesquisas bibliográficas;

d) **Do ponto de vista dos procedimentos técnicos**. A pesquisa pode ser classificada como uma **pesquisa bibliográfica** e um **quasi experimento**. Os problemas que a abordagem proposta visa amenizar foram fundamentados em pesquisas bibliográficas. Além disso, um dos procedimentos adotados para avaliação da abordagem pode ser caracterizado como um quasi experimento realizado com um conjunto de voluntários não identificados.

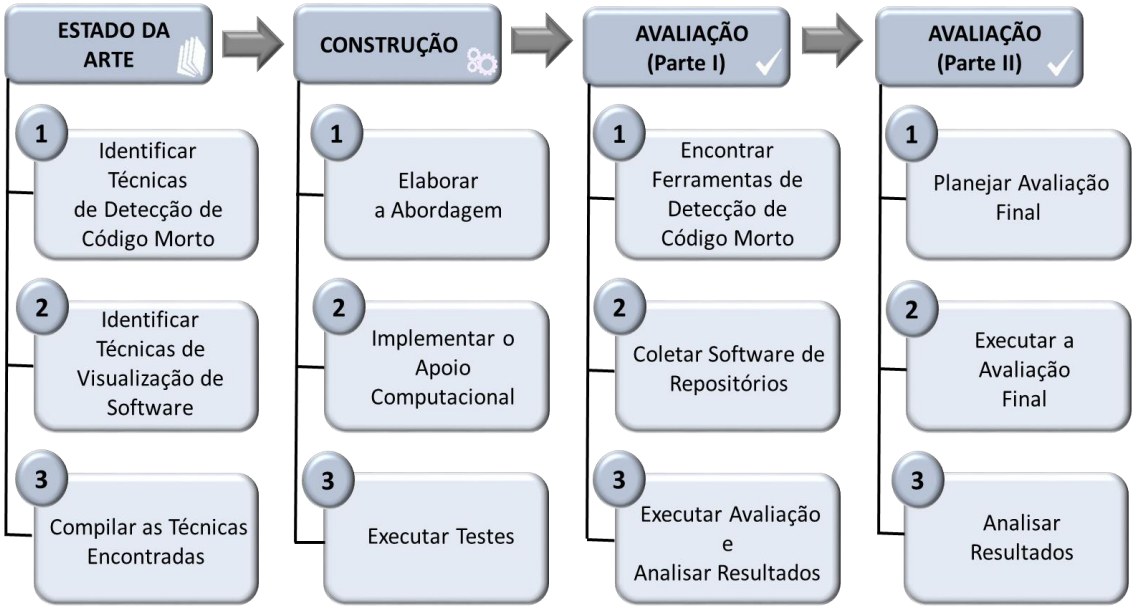
Com base nessa classificação, foi elaborado o método de pesquisa para a realização deste trabalho (FIGURA 2.1):

a) **Etapa 1 - Estado da Arte**. O objetivo foi encontrar estudos que abordam temas relacionados a código morto e a visualização de software. Os resultados obtidos são apresentados no Capítulo 4 e no Capítulo 5. Essa etapa é composta por três atividades:

   - **Atividade 1 - Identificar Técnicas de Detecção de Código Morto**. Breve revisão de literatura foi realizada para contextualizar o assunto e para encontrar técnicas de detecção de código morto. Com essa revisão, foi verificado que existem diferentes técnicas de detecção aplicadas em contextos distintos. Com isso, foi necessário realizar uma Revisão Sistemática de Literatura para capturar e sintetizar informações úteis relacionadas a código morto (BASTOS et al., 2016d);

- **Atividade 2 - Identificar Técnicas de Visualização de Software**. Foi realizada uma revisão de literatura para identificar técnicas de visualização de software. Foram encontrados estudos que sintetizaram as principais técnicas existentes, sendo utilizados como base para construção do referencial teórico sobre compreensão de sistemas de software, técnicas de visualização e suas principais características (KIENLE; MULLER, 2007; CASERTA, ZENDRA, 2011; NOVAIS et al., 2013b);

Figura 2.1 - Método de Pesquisa



Fonte: Do Autor (2017).

- **Atividade 3 - Compilar as Técnicas Encontradas**. As técnicas de detecção de código morto e as técnicas de visualização de software foram compiladas e sintetizadas de forma a contribuir para a elaboração da abordagem;

b) **Etapa 2 - Construção**. O objetivo foi elaborar a abordagem e automatizar sua execução por meio de um *plug-in* para a plataforma Eclipse IDE. Os resultados obtidos são apresentados no Capítulo 6. Essa etapa é composta por três atividades:

- **Atividade 1 - Elaborar a Abordagem**. As informações obtidas com as revisões de literatura foram utilizadas na definição da abordagem. O propósito da abordagem é identificar estaticamente métodos mortos em versões de sistemas de software e apresentar suas características evolutivas utilizando técnicas de visualização. De acordo com os resultados obtidos com a revisão sistemática de literatura, a técnica de detecção de código morto indicada nesse caso é a Análise de Acessibilidade. A evolução do código morto é apresentada

utilizando gráfico de linha (visualização quantitativa) e uma técnica definida com base nas técnicas TreeMap e Matriz de Evolução (visualização qualitativa);

- **Atividade 2 - Implementar o Apoio Computacional**. Foi desenvolvido um apoio computacional (*plug-in* para a plataforma Eclipse IDE) para automatizar a abordagem proposta. De modo geral, esse *plug-in* executa a técnica Análise de Acessibilidade em diferentes versões de sistemas de software, identificando os métodos mortos e analisando sua evolução. Uma característica visual é atribuída a cada método morto de acordo com suas características evolutivas. Posteriormente, essas características visuais são renderizadas nas técnicas de visualização e apresentadas ao engenheiro de software. Alguns recursos de interação foram implementados no *plug-in*, como mecanismos de busca, filtros, *tooltips* informativos e redirecionamento para o código do método morto;
- **Atividade 3 - Executar Testes**. Testes de unidade foram executados durante a implementação do *plug-in* para verificar a corretude dos módulos responsáveis pela identificação de código morto e pela geração das visualizações. Em seguida, foram realizados testes de integração para avaliar o funcionamento desses componentes quando executados em conjunto;

c) **Etapa 3 - Avaliação (Parte I)**. O objetivo foi investigar a capacidade de detecção de código morto da técnica Análise de Acessibilidade em relação a outras abordagens e ferramentas existentes. Os resultados obtidos são apresentados na Seção 7.2. Essa etapa é composta por três atividades:

- **Atividade 1 - Encontrar Ferramentas de Detecção de Código Morto**. Foi realizada uma busca na literatura por ferramentas que automatizam a detecção de métodos mortos e que possam ser utilizadas neste estudo. Foram encontradas algumas ferramentas, nas quais a maioria não eram funcionais ou possuíam características que as impediram de serem utilizadas. Dessa forma, foi utilizada a ferramenta Understand, uma ferramenta funcional que identifica métodos mortos em apenas uma versão de sistema de software Java;
- **Atividade 2 - Coletar Software de Repositórios**. Foram selecionados cinco sistemas de software Java para serem utilizados neste estudo. Esses sistemas foram selecionados por serem frequentemente utilizados na avaliação de

trabalhos que dependem da análise do código. Além disso, foram selecionadas as cinco versões mais recentes desses sistemas, as quais foram utilizadas em uma análise do comportamento da evolução do código morto identificado pelo *plug-in* desenvolvido e pela ferramenta Understand;

- **Atividade 3 - Executar Avaliação e Analisar Resultados.** A avaliação foi conduzida por meio de uma comparação qualitativa do código morto identificado pelo *plug-in* desenvolvido e pela ferramenta Understand, após a análise do código dos sistemas de software selecionados na etapa anterior. De modo geral, o *plug-in* identificou maior quantidade de código morto do que a ferramenta nos sistemas de software analisados. Foi possível perceber que algumas características adotadas na implementação das ferramentas (*plug-in* e Understand) causaram essa diferença. Além disso, foi possível perceber que ambas as ferramentas identificaram tendências evolutivas semelhantes para o código morto das cinco versões mais recentes dos sistemas de software;

d) **Etapa 4 - Avaliação (Parte II)**. O objetivo foi investigar possíveis benefícios da abordagem para identificar e compreender a evolução de código morto. Os resultados obtidos são apresentados na Seção 7.3. Essa etapa é composta por três atividades:

- **Atividade 1 - Planejar a Avaliação Final**. Foram elaborados formulários com questões para coleta de dados que permitem investigar os efeitos da abordagem para seus usuários. As questões foram definidas para verificar esses efeitos em dois focos distintos: i) identificação de código morto; e ii) compreensão da evolução de código morto. Além disso, foram solicitadas opiniões pessoais dos usuários a respeito de pontos positivos e pontos negativos da abordagem;
- **Atividade 2 - Executar a Avaliação Final**. O estudo foi conduzido em ambiente acadêmico, no qual um grupo de voluntários (alunos de graduação) executaram um conjunto de atividades sem e com o apoio do *plug-in* que implementa abordagem. A execução dessas atividades foi guiada pelos questionários elaborados na etapa anterior;
- **Atividade 3 - Analisar os Resultados**. Os resultados obtidos foram analisados de forma quantitativa e qualitativa. A análise quantitativa foi realizada por meio das variáveis precisão e eficiência, calculadas a partir das respostas dos formulários. Testes de hipóteses foram realizados para cada variável de acordo com o foco das questões (identificação ou compreensão da evolução de código

morto). De modo geral, o uso do *plug-in* resultou no aumento da média das variáveis em ambos os focos, mostrando-se útil na execução das atividades. A análise qualitativa mostrou que, na opinião dos participantes, o *plug-in* facilitou as atividades para identificar e compreender a evolução de códido morto.

3 TRABALHOS RELACIONADOS

Alguns trabalhos utilizaram a técnica Análise de Acessibilidade para identificar código morto e melhorar a qualidade dos sistemas de software. Outros trabalhos utilizaram técnicas de visualização para facilitar a compreensão da evolução desses sistemas. No entanto, não foram encontrados trabalhos para visualizar a evolução do código morto, conforme proposto neste estudo. Nas seções seguintes, é apresentada uma síntese desses trabalhos.

3.1 Uso de análise de acessibilidade para identificar código morto

A técnica Análise de Acessibilidade foi aplicada de diferentes formas para detecção de código morto. Um estudo pioneiro definiu um modelo de dados que permite analisar a acessibilidade de entidades de software desenvolvidas em C++ (CHEN et al., 1998). Esse modelo de dados deve satisfazer um critério de completude e abstrair as informações contidas em repositórios de código, mapeando as entidades, suas dependências e os relacionamentos existentes entre elas. O critério de completude é satisfeito se as dependências existentes no código forem mapeadas para o modelo de dados. A técnica Análise de Acessibilidade é utilizada nesse modelo de dados para detecção de código inacessível considerando declarações, funções ou arquivos. No primeiro passo, é identificado um conjunto de entidades alcançáveis a partir de entidades que iniciam a execução do sistema de software. No segundo passo, são identificadas as entidades do software. No terceiro passo, o conjunto de entidades inacessíveis é obtido pela diferença entre os conjuntos das entidades do software e das entidades alcançáveis.

Um estudo aplicou a técnica Análise de Acessibilidade para identificar especificamente métodos inacessíveis em sistemas de software desenvolvidos em Java ou em C++ (BACON et al., 2003). De forma similar ao estudo anterior, o código inacessível é detectado pela diferença entre o conjunto dos métodos do sistema de software e o conjunto de métodos acessíveis. Nesse estudo, também foram considerados como métodos acessíveis os responsáveis pela inicialização da execução do sistema. As informações para utilizar a técnica Análise de Acessibilidade são obtidas de forma estática diretamente no código do sistema de software.

A abordagem DUM (*Detecting Unreachable Methods*) analisa *bytecodes* Java e cria uma representação baseada em grafo do sistema de software, no qual os nós correspondem aos métodos e as arestas correspondem às chamadas existentes entre eles (ROMANO et al.,

2016). A técnica Análise de Acessibilidade é utilizada por meio da representação em grafo em três passos, conforme descrito nos estudos anteriores. Os métodos considerados como acessíveis são métodos `main()`, métodos que inicializam campos ou blocos e métodos relacionados com a serialização/desserialização de objetos, invocados via reflexão.

De modo geral, os estudos apresentados utilizaram a técnica Análise de Acessibilidade de forma semelhante, com diferença apenas na maneira na qual as informações do sistema de software são obtidas ou no tipo de código inacessível detectado (métodos, declarações ou arquivos). A principal limitação indicada nesses estudos é a dificuldade que abordagens que utilizam análise estática enfrentam ao lidar com algumas características desses sistemas, por exemplo, chamadas via reflexão ou polimorfismo. Além disso, nesses estudos, é preciso ter conhecimento prévio do sistema de software, visto que é necessário identificar um conjunto de métodos acessíveis antes de realizar a detecção de código morto propriamente dita.

Com base nesses fatores, a técnica Análise de Acessibilidade foi aplicada de modo diferente na abordagem proposta neste estudo, em que seu principal diferencial é a detecção direta dos métodos mortos, evitando a necessidade de análise prévia do sistema de software para identificar os métodos acessíveis.

3.2 Uso de visualização da evolução de software

A visualização de software tem sido utilizada para facilitar a compreensão de aspectos evolutivos de sistemas de software. A abordagem SourceMiner Evolution permite visualizar a evolução de sistemas de software por meio das estratégias diferenciais relativas, absolutas e temporal (NOVAIS et al., 2013a). Essa abordagem realiza mapeamentos de *features* (recursos disponíveis no sistema) utilizando heurísticas de expansão para identificar elementos de código que as implementam. São utilizadas cores para destacar elementos de código relacionadas a cada *feature*, assim como mudanças realizadas nesse código ao comparar duas versões. Essas cores são apresentadas por meio de quatro perspectivas, em que três delas apresentam o diferencial em relação a dependência, a estrutura e a herança existente entre duas versões, utilizando as técnicas Visão de Dependência, Treemap e Visão Polimétrica, respectivamente. Na quarta perspectiva, denominada Timeline Matrix, é apresentada uma visão sob demanda das outras perspectivas, mostrando detalhes da evolução do código referente a cada *feature*.

Uma abordagem foi proposta para visualizar a evolução da modularização e do acoplamento de sistema de software utilizando matrizes (MELANCON; RUFIANGE, 2014).

A técnica de visualização proposta, denominada AniMatrix, utiliza representação em matrizes para retratar a evolução de componentes desses sistemas que podem ser representadas por meio de grafos, por exemplo, chamadas entre métodos ou relação entre classes. Desse modo, a matriz de visualização contém dimensão `n × n`, sendo `n` a quantidade de nós do grafo correspondente. As arestas do grafo são representadas pelo preenchimento da interseção da linha e da coluna referentes aos vértices que as compõem. A variação das cores desse preenchimento é utilizada para especificar características evolutivas, por exemplo, a remoção de uma aresta entre duas versões é representada na cor vermelha, enquanto a adição é representada na cor verde.

Em outro trabalho, é apresentada SAMOA, uma abordagem para análise e visualização da evolução de aplicações para dispositivos móveis (LANZA; MINELLI, 2013). Essa abordagem propõe a análise de algumas medidas de software, apresentando-as em um catálogo de visões personalizados que auxiliam a compreensão da evolução estrutural desses aplicativos. A ferramenta utiliza três tipos de visualizações: i) **visualização instantânea**, apresenta uma versão específica do aplicativo; ii) **visualização da evolução**, apresenta a evolução dos aplicativos considerando suas versões; e iii) **visões de vários sistemas**, apresentam mais de um aplicativo simultaneamente. Além disso, a abordagem propõe mecanismos de interação para obtenção de detalhes das visualizações.

A abordagem PREViA foi proposta para facilitar a percepção das divergências entre a arquitetura planejada e a arquitetura emergente de sistemas de software, além de auxiliar a percepção da evolução da sua implementação e do seu projeto (*design*) (SCHOTS, 2011). De modo geral, essa abordagem compara versões do sistema de software considerando os modelos conceituais e o código dessas versões e utilizando medidas de similaridade de nome, de tipo, de valor e de relacionamentos. Os resultados obtidos com essas comparações são representados visualmente, em que as divergências entre dois modelos são representadas por meio de destaque de cores. Em relação ao foco de evolução, as cores são utilizadas para indicar elementos adicionados, removidos ou modificados entre duas versões.

De maneira geral, os estudos apresentados propuseram abordagens baseadas no uso de recursos visuais para compreender a evolução de sistema de software em relação a uma característica específica, por exemplo, estrutura, acoplamento e arquitetura. A definição dessas abordagens foi motivada pelo aumento da complexidade que decorre ao longo da evolução e a capacidade das técnicas de visualização em amenizar essa complexidade e facilitar a compreensão.

Assim como esses trabalhos, a abordagem proposta neste estudo tem como princípio facilitar a compreensão da evolução utilizando técnicas de visualização de software. Porém, seu principal diferencial é a análise do código morto, um dos fatores que contribui para aumento da poluição e, consequentemente, da complexidade ao longo da evolução. Dessa forma, a abordagem possibilita compreender a evolução da poluição de sistemas de software, permitindo que medidas sejam tomadas para reduzir a relação negativa entre a evolução e o aumento da complexidade de sistemas de software.

4 COMPREENSÃO E VISUALIZAÇÃO DE SOFTWARE

A compreensão é requisito fundamental nas fases do ciclo da vida de sistemas de software, principalmente na fase de manutenção, na qual grande parte do tempo utilizado é destinado ao entendimento das funções e das características do sistema de software (DUCASSE; LANZA, 2005). Os relacionamentos complexos entre as entidades desses sistemas, a escassez de documentação, a evolução e a poluição do código fazem com que essa compreensão não seja trivial. Desse modo, técnicas de visualização de software têm sido frequentemente utilizadas para representar o sistema de software e facilitar sua compreensão. Neste capítulo, são apresentados alguns conceitos relacionados à visualização de software e às principais técnicas de visualização existentes.

O restante do capítulo está organizado da seguinte forma. Conceitos relacionados à compreensão de sistemas de software são discutidos na Seção 4.2. Visão geral sobre visualização de software é apresentada na Seção 4.3. Algumas técnicas de visualização de software, seus paradigmas e técnicas para visualizar a evolução são relatados na Seção 4.4.

4.1 Compreensão de software

A compreensão consiste em obter pleno conhecimento da funcionalidade, da estrutura interna e do comportamento de sistemas de software (MAYRHAUSER; VANS, 1995). Essa compreensão é requisito principal para execução da maioria das atividades relacionadas ao desenvolvimento e à manutenção desses sistemas. Durante a fase de desenvolvimento, a compreensão conduz para o produto construído estar de acordo com o planejado. Na etapa de manutenção, a compreensão apoia a identificação de problemas, melhorias e evolução de sistemas de software (CARNEIRO, 2013). As teorias clássicas *bottom-up* e *top-down* descrevem como a compreensão pode ser obtida:

a) **Análise *bottom-up***. A compreensão geral de sistemas de software é obtida por meio da compreensão de segmentos menores (RAJLICH; WILDE, 2002). Esse modelo de compreensão aborda situações em que o programador não esteja familiarizado com o domínio do sistema, no qual o entendimento inicia-se com a leitura de informações de baixo nível (por exemplo, o código) e, posteriormente, são criadas abstrações de alto nível que facilitam a compreensão (O'BRIEN, 2003);

b) **Análise *top-down***. A compreensão é obtida pelo mapeamento do conhecimento previamente adquirido a respeito do domínio do sistema em estruturas de baixo nível, por exemplo, código. Usualmente, essa teoria utiliza avaliação de hipóteses definidas com base no conhecimento de alto nível do domínio. Essas hipóteses são aceitas ou rejeitadas conforme seu relacionamento com as informações extraídas a partir das estruturas de baixo nível (RAJLICH; WILDE, 2002).

Nesse contexto, o processo de compreensão de sistemas de software consome parte significativa do tempo destinado ao desenvolvimento e à manutenção, sendo considerado um processo árduo por causa das seguintes caraterísticas (CARNEIRO, 2013):

a) **Software é complexo**. A quantidade de informações vinculadas ao sistema de software, a complexidade dos relacionamentos entre os módulos, a poluição e a variedade de funções prejudicam a compreensão desses sistemas. Algumas técnicas de engenharia reversa foram propostas para amenizar esse problema, porém não são suficientes quando utilizadas de maneira isolada;

b) **Software é abstrato**. Sistemas de software não possuem uma forma física. Esse fato dificulta a compreensão desses sistemas, visto que o ser humano possui maior capacidade de absorver informações por intermédio da visão do que por qualquer outro sentido;

c) **Software evolui**. Apesar de necessária, a evolução está relacionada com o aumento da complexidade do sistema de software. À medida que esse sistema evolui, o tamanho do código, a quantidade de funções, de poluição e de informações tendem a aumentar, dificultando ainda mais o processo de compreensão. Outro fator que agrava essa complexidade é a documentação, que usualmente está desatualizada.

Além disso, um dos fatores que dificulta a compreensão é o código ser o principal artefato disponível para obtenção de informações (HENDRIX, 2002). A complexidade desse artefato e a quantidade de poluição usualmente presentes fazem com que seu entendimento seja prejudicado. Desse modo, a visualização de software é uma área que tem sido explorada para representar visualmente informações vinculadas aos sistemas de software, de modo a facilitar sua compreensão.

4.2 Visualização de software

A visualização de software pode ser definida como uma disciplina que faz uso de recursos visuais para representar a estrutura, o comportamento e a evolução de sistemas de software (CARPENDALE; GHANAM, 2008). Na visualização da **estrutura**, são representados aspectos estáticos desses sistemas, incluindo informações relacionadas ao código, à estrutura de dados e à organização modular. Na visualização do **comportamento**, são representadas informações reais obtidas durante a execução desses sistemas, tais como, chamadas de funções e comunicação entre objetos. Na visualização da **evolução**, são representadas características evolutivas de determinados atributos desses sistemas, destacando suas mudanças ao longo das versões. De modo geral, as visualizações são geradas por um processo descrito em três fases, sendo o resultado de uma fase utilizado como entrada para a próxima fase (DIEHL, 2007):

a) **Aquisição**. As informações são extraídas a partir de uma fonte de dados;

b) **Análise**. Síntese das informações para redução da quantidade de dados a serem apresentados, com ênfase nos aspectos mais importantes;

c) **Visualização**. As informações resultantes da fase Análise são mapeadas em componentes visuais, renderizadas e apresentadas ao usuário.

A maneira na qual a informação é representada visualmente (como as formas e as cores utilizadas) é conhecida como técnica de visualização ou metáfora visual (MALETIC et al., 2002]. A elaboração de uma técnica de visualização pode ser considerada uma atividade extremamente flexível, visto que não existe um modelo mental pré-definido para representar as informações abstratas de sistemas de software. Desse modo, o projetista da técnica possui total autonomia para utilizar quaisquer atributos visuais que achar necessário, tendo que disponibilizar uma legenda que descreva o significado desses atributos. Essa flexibilidade resultou na definição de uma diversidade de técnicas, nas quais muitas técnicas diferentes foram propostas para representar o mesmo atributo do sistema de software.

Um dos principais problemas enfrentados durante a definição de uma técnica de visualização é a escalabilidade. Esse problema ocorre quando a técnica é compreensível apenas se pouca quantidade de informações for representada. Para sistemas de software com grande quantidade de linhas de código e com relacionamentos complexos entre módulos, as visualizações tornam-se sobrecarregadas de informações, dificultando a interpretação e a

assimilação do conteúdo apresentado (PETRE; QUINCEY, 2006). Para contornar esse problema, as visualizações devem possuir recursos de interação que enfatizam apenas áreas de interesse do usuário. Os principais recursos de interação existentes são (KEIM, 2002):

a) ***Zoom***. Recurso responsável por ampliar ou reduzir um elemento visual de acordo com a interação do usuário. Além da alteração do tamanho, o detalhamento apresentado na visualização pode variar de acordo com esse recurso. Informações podem ser representadas de maneira detalhada à medida que o *zoom* aumenta e em menor detalhamento quando o *zoom* diminui (KEIM; KRIEGEL, 1996);

b) **Filtros**. Recurso responsável por enfatizar visualmente um conjunto específico de informações, podendo ser ativado pela seleção direta do subconjunto desejado (navegação) ou pela especificação de algumas propriedades desse subconjunto (buscas) (KEIM, 2002). Quando é possível saber com antecedência o que será filtrado, diz-se que a filtragem ocorreu por pré-processamento. Em contrapartida, a filtragem sob demanda ocorre em tempo real e, normalmente, é utilizada para tarefas específicas (SCHOTS, 2011).

Além dos recursos de interação, a expressividade e a eficácia são características desejáveis das técnicas de visualização (OUDSHOORN et al., 1996; MALETIC et al., 2002). A expressividade determina se a representação visual é capaz de representar as informações desejadas. A eficácia refere-se ao quanto a representação visual é capaz de expressar essas informações. Com o propósito de atender a essas características e facilitar a compreensão de sistemas de software, diversas técnicas de visualização foram propostas. Existem técnicas para visualizar variados tipos de informações, tais como, arquitetura, evolução, execução, processo de desenvolvimento e código (SHARAFI, 2011). Na próxima seção, são apresentados algumas dessas técnicas e os paradigmas visuais utilizados.

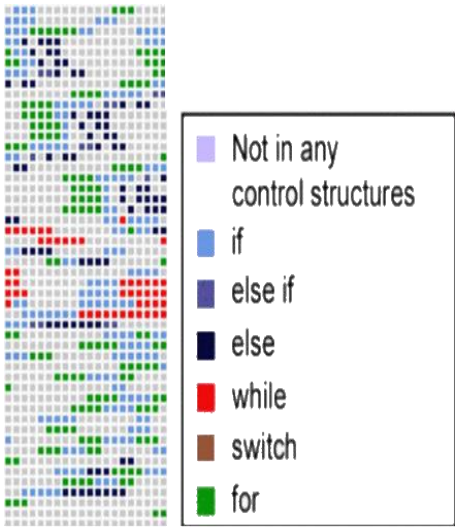
4.3 Técnicas de visualização de software

As técnicas de visualização viabilizam a representação visual dos dados e podem ser classificadas em quatro paradigmas (KEIM; KRIEGEL, 1996):

a) **Orientado a *Pixel***. As informações do sistema de software são mapeadas em *pixels* coloridos de acordo com um mapa de cores previamente definido. Um exemplo de técnica de visualização é SeeSoft (FIGURA 4.1), que pode ser utilizada para visualizar

instruções de controle do código (EICK et al., 1992). Cada instrução é representada por um *pixel* colorido, no qual a variação das cores é utilizada para representar tipos específicos de instrução. A compreensão dessa técnica depende de uma legenda que especifica a cor que representa cada tipo de instrução (CASERTA; ZENDRA, 2011);

Figura 4.1 - Exemplo de Visualização SeeSoft



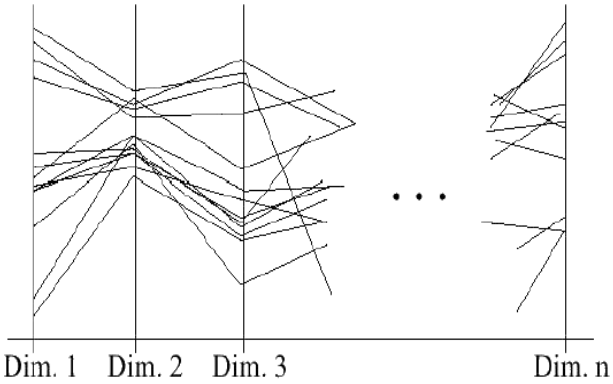
Fonte: Caserta; Zendra (2011).

b) **Geométrico**. As informações multidimensionais do sistema de software são representadas utilizando duas dimensões (KEIM; KRIEGEL, 1996). A busca de uma projeção geométrica representativa é conhecida como “busca de projeção” (*projection pursuit*). As técnicas desse paradigma representam simultaneamente várias dimensões, possibilitando identificar diferenças na distribuição dos dados ou correlação entre atributos (INSELBERG et al., 1987). Na Figura 4.2, é apresentada a técnica Coordenadas Paralelas, na qual um espaço de `n` dimensões é mapeado para um espaço bidimensional utilizando `n` eixos equidistantes e paralelos a um dos eixos de exibição (`x` ou `y`). Cada eixo representa uma dimensão mapeada linearmente em ordem crescente do conjunto de dados. Cada item de dado é apresentado como uma linha que intercepta cada eixo no ponto correspondente ao valor do atributo associado;

c) **Iconográfico**. As informações do sistema de software são representadas utilizando ícones representativos (KEIM; KRIEGEL, 1996). As técnicas iconográficas conseguem representar grande quantidade de informações, que variam de acordo com as configurações adotadas no ícone. A maior dificuldade desse paradigma é a definição de um ícone significativo que consiga representar adequadamente as informações desejadas. Na Figura 4.3, é apresentada uma técnica desse paradigma, na

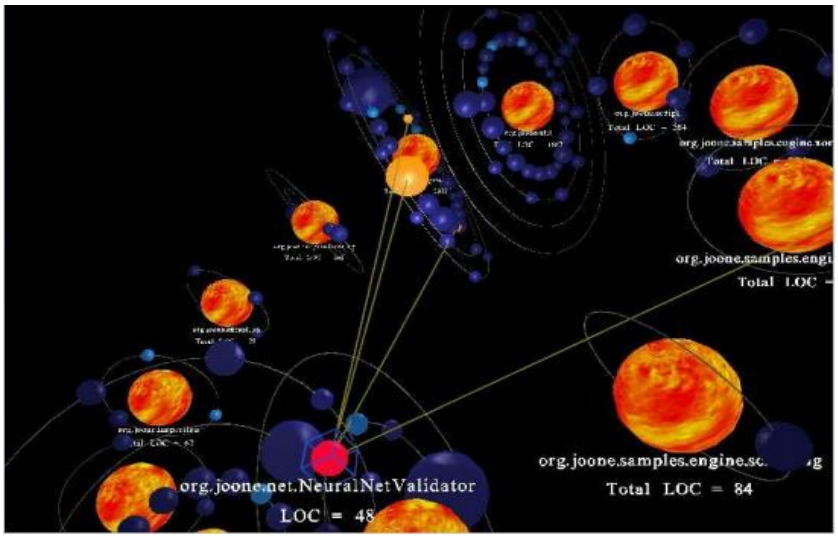
qual os componentes do sistema de software são representados por meio de ícones de um sistema solar. Os pacotes são representados pelo sol e as classes são representadas por planetas que orbitam o sol. Quanto maior a distância existente entre a órbita de uma classe e o sol, maior a profundidade dessa classe na hierarquia de herança (CASERTA; ZENDRA, 2011);

Figura 4.2 - Projeção com Coordenadas Paralelas



Fonte: Keim; Kriegel (1996).

Figura 4.3 - Técnica de Visualização SolarSystem

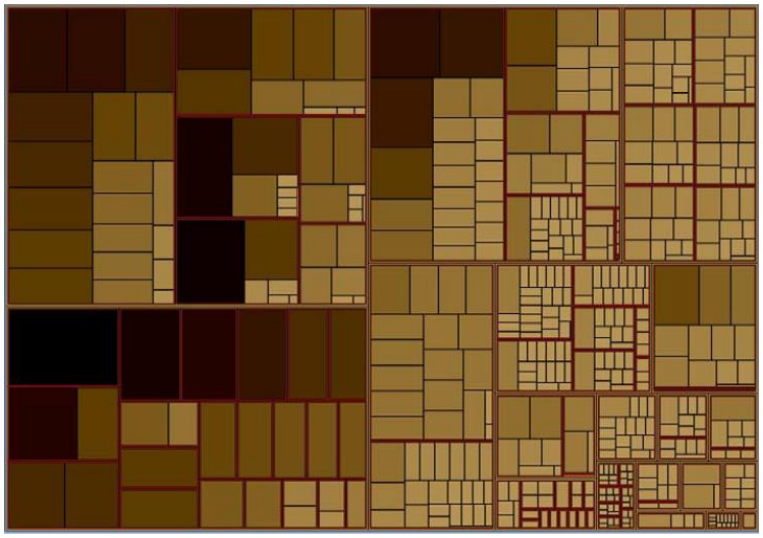


Fonte: Yang; Graham (2003).

d) Hierárquico. As técnicas são utilizadas para representar a hierarquia existente entre os componentes do sistema de software, como a hierarquia de herança entre classes ou a hierarquia estrutural entre pacotes, classes e métodos (KEIM; KRIEGEL, 1996). Na Figura 4.4, é apresentada a técnica hierárquica TreeMap, utilizada para representar a estrutura do sistema de software. As cores das bordas são utilizadas para distinguir os componentes representados, no qual bordas amarelas correspondem aos pacotes, bordas vermelhas correspondem às classes e bordas pretas correspondem aos

métodos. A cor de preenchimento e o tamanho dos retângulos mapeiam características específicas dos componentes. Por exemplo, as cores podem ser utilizadas para representar a visibilidade do método (público, privado e protegido), a largura do retângulo pode ser utilizada para representar a quantidade de parâmetros (quanto maior a largura, maior a quantidade de parâmetros) e a altura pode ser utilizada para representar a quantidade de linhas de código (quanto maior a altura, maior a quantidade de linhas de código).

Figura 4.4 - Exemplo de Técnica de Visualização TreeMap



Fonte: Carneiro (2013).

Além das técnicas de visualização definidas em cada paradigma, foram propostas técnicas para visualizar a evolução do software. De modo geral, a finalidade dessas técnicas é reduzir a dificuldade de compreensão da quantidade significativa de informações geradas ao longo dessa evolução (NOVAIS et al., 2013b). Algumas técnicas de visualização fornecem uma representação detalhada para cada versão do sistema de software, fazendo necessário o uso de recursos de interação para visualizar as características evolutivas dessas versões. Em contrapartida, outras técnicas exibem as características evolutivas das versões analisadas em apenas uma imagem (CASERTA; ZENDRA, 2011). Dessa forma, as técnicas de visualização da evolução são planejadas utilizando diferentes estratégias que definem como as informações são representadas visualmente (NOVAIS et al., 2013a). As duas estratégias são:

a) **Visualização Diferencial**. Representa visualmente as características evolutivas de duas versões do sistema de software, que podem ser selecionadas aleatoriamente dentre as demais versões. Essa estratégia pode ser especializada em:
   - **Diferencial Relativa**. Representa visualmente o crescimento ou a redução de alguma propriedade do sistema de software entre duas versões;

- **Diferencial Absoluta**. Utilizada quando a definição de crescimento ou de redução não se aplica. São representados visualmente eventos discretos que ocorreram entre duas versões, por exemplo, propriedades que surgiram ou desapareceram de uma versão para outra;

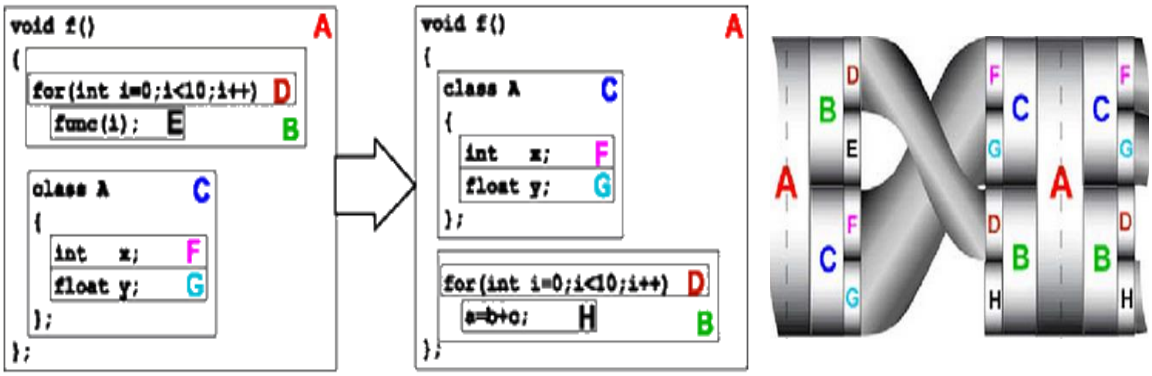
b) **Estratégias de Visualização Temporal**. Representa visualmente as características evolutivas das versões disponíveis para análise, auxiliando a compreensão de padrões e de mudanças ao longo da evolução. A visualização temporal pode ser especializada em:

- **Visão Geral**. É produzido um panorama da evolução do sistema de software ao longo do seu ciclo de vida. A principal dificuldade é a quantidade de informações que devem ser representadas, fazendo necessária a definição de recursos de interação;
- ***Snapshot***. Representa visualmente o estado do sistema de software em um momento específico da sua história. É necessário definir recursos de interação que permita a seleção do momento histórico a ser representado, além da navegabilidade nesse histórico;
- ***Snapshot* Acumulativo**. É um subtipo da visualização *Snapshot*, no qual a representação visual destaca as características evolutivas das versões analisadas até o momento do histórico selecionado.

Na Figura 4.5, é apresentada uma técnica para visualizar a evolução de linhas de código denominada CodeFlows. É apresentado na Figura 4.5A o código de duas versões do sistema de software, no qual a linha é identificada por uma letra. A representação visual de CodeFlows correspondente a esse código é representada na Figura 4.5B. Como pode ser observado, o cruzamento das linhas ilustrado em CodeFlows corresponde a alteração da disposição dos blocos de código B e C entre as duas versões.

A Matriz de Evolução (FIGURA 4.6) representa a evolução das classes do sistema de software, na qual as colunas da matriz correspondem às versões e as linhas correspondem às classes de cada versão (LANZA, 2001). A variação da largura e da altura dos componentes visuais que representam as classes pode ser utilizada para retratar suas características, como quantidade de atributos, de linhas de código ou de métodos. A capacidade de representação da informação dessa técnica pode aumentar com o uso de cores, que podem ilustrar outras especificidades dessas classes.

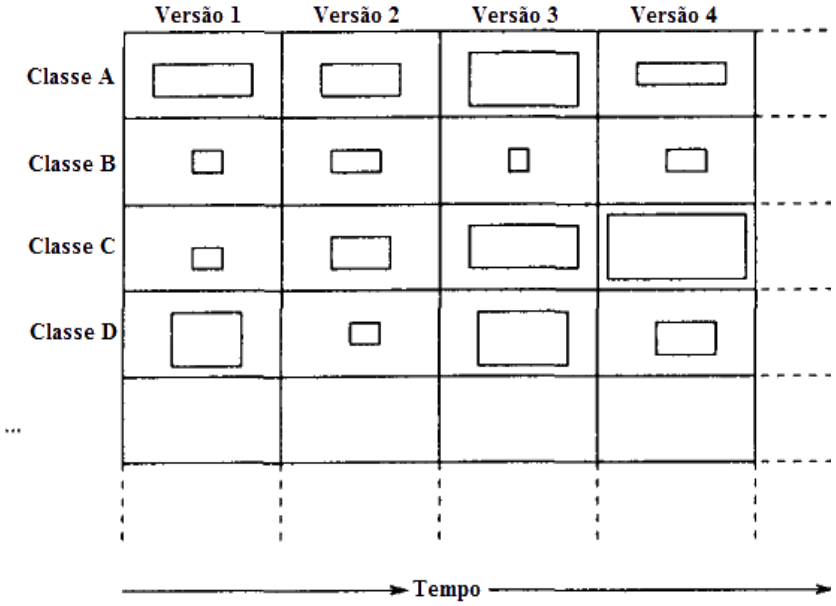
Figura 4.5 - Exemplo da Técnica Code Flows



(A) (B)

Fonte: Adaptado de Telea; Auber (2008).

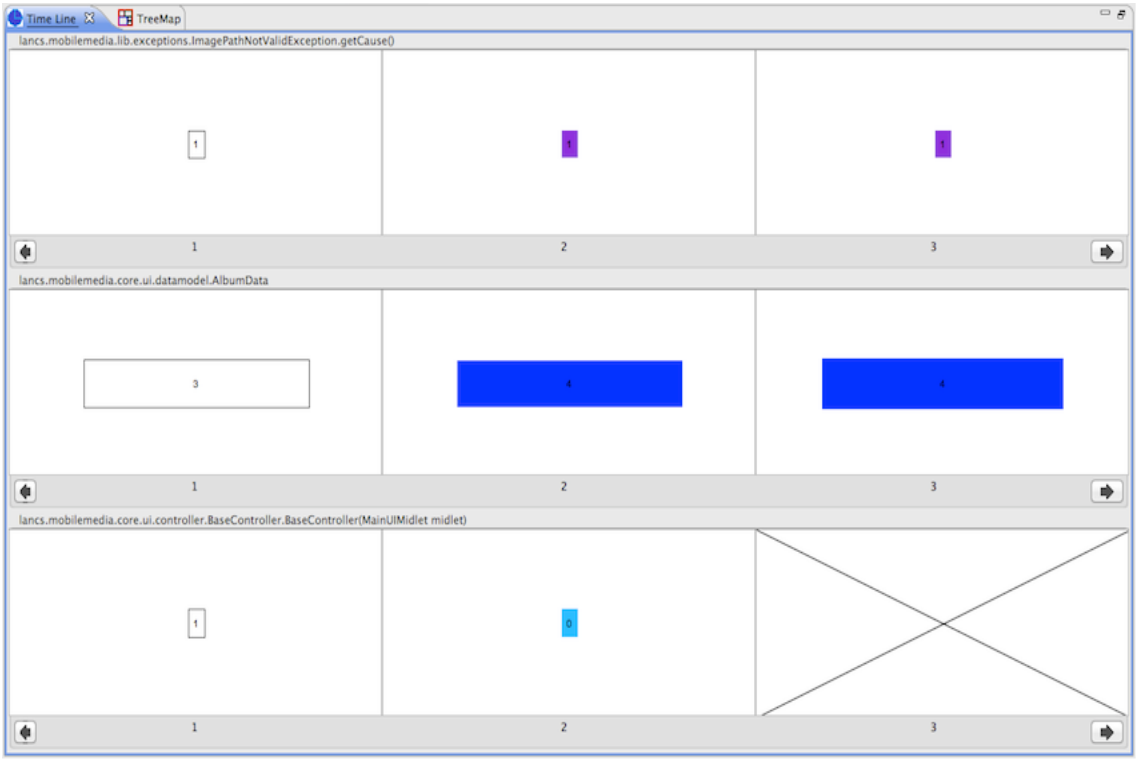
Figura 4.6 - Exemplo de Matriz de Evolução



Fonte: Lanza (2001).

A técnica TimeLine Matrix (FIGURA 4.7) é semelhante a Matriz de Evolução e representa simultaneamente a evolução de três componentes do sistema de software (NOVAIS et al., 2012). Nessa técnica, é utilizada uma matriz 3 x 3, sendo as linhas denominadas *timeline* de evolução e as colunas correspondem às versões do elemento do sistema de software em análise. Cada *timeline* pode representar uma informação distinta desse sistema, como pacotes, classes e métodos. A variação das cores é utilizada para representar especificidades desses componentes.

Figura 4.7 - Exemplo de TimeLine Matrix



Fonte: Novais *et al*. (2012).

4.4 Considerações finais

Neste capítulo, foram apresentados os principais conceitos sobre compreensão e visualização de software, além de alguns exemplos de técnicas e seus paradigmas. Ao longo do capítulo foi possível observar que a visualização de software é uma área flexível, no qual variadas técnicas foram propostas para visualizar diferentes informações do software. Além disso, existe a possibilidade de combinar técnicas de diferentes paradigmas e de diferentes propósitos, formando uma nova técnica para representar um novo tipo de informação.

Neste trabalho, a técnica hierárquica TreeMap foi adaptada para representar de maneira organizada o código morto. A técnica Matriz de Evolução foi adaptada e combinada com a técnica TreeMap, visando representar as características evolutivas desse código morto.

5 DETECÇÃO DE CÓDIGO MORTO - ESTADO DA ARTE

Código morto prejudica a manutenibilidade de sistemas de software por contribuir de forma significativa com a poluição do código e dificultar sua compreensão (SCANNIELLO, 2014). Identificar código morto é uma tarefa árdua, fazendo necessária a definição de técnicas que permitem automatizar essa identificação (ROMANO et al. 2016). Desse modo, foram propostas algumas técnicas de detecção de código morto ao longo dos anos, aplicadas em diferentes contextos relacionados a sistemas de software. Neste capítulo, são apresentados alguns conceitos básicos relacionados a código morto. Além disso, são relatados os resultados obtidos em uma Revisão Sistemática de Literatura (RSL) para identificar técnicas de detecção de código morto existentes e seu domínio de aplicação.

O restante do capítulo está organizado da seguinte forma. Algumas definições e alguns tipos de código morto existentes são apresentados na Seção 5.2. Descrição dos artigos e das técnicas de detecção de código morto encontradas na RSL é apresentada na Seção 5.3.

5.1 Conceitos básicos

O termo código morto está relacionado com trechos de código desnecessários/inoperantes, que podem ser eliminados sem alterar a funcionalidade do sistema de software (EDER et al., 2012). Grande parte dos trabalhos relacionados a Engenharia de Software consideram código morto como trechos de código inacessíveis e que não podem ser executados, por causa da impossibilidade de serem invocados durante o funcionamento desses sistemas. Na área Linguagem de Programação, código morto é considerado código desnecessário, pois são executados e produzem resultados que não interferem no funcionamento do sistema de software (KUMAR; SUNITHA, 2006). Existe ainda a relação de código morto com trechos de código não alcançáveis por qualquer fluxo de execução, como trechos de código localizados após instruções de desvio incondicional ou após instruções de retorno (MARTINS et al., 2010). Alguns tipos de código morto são:

a) **Variáveis Mortas**. Variáveis declaradas e inicializadas, mas não manipuladas ou não utilizadas em outros trechos do código. Outro procedimento que torna uma variável morta é a propagação de cópias, em que, dada uma atribuição $x \leftarrow y$, o uso de y será posteriormente substituído pelo uso de x, sem que instruções intermediárias alterem seu valor, fazendo com que ambas as variáveis sejam iguais e tornando y inútil (KUMAR; SUNITHA, 2006);

b) **Parâmetros Mortos**. Passagem de parâmetros não utilizados no corpo do método. Esse tipo de código dificulta a compreensão da funcionalidade desse método, principalmente se pertencer às bibliotecas/APIs públicas, as quais os desenvolvedores possuem pouca familiaridade com o código (FEHNKER; HUUCK, 2013);
c) **Valor de Retorno Morto**. O valor retornado por determinado método não é utilizado pelo seu chamador. Esse fato pode ocorrer por causa de chamadas realizadas de forma incorreta, que ignoram a funcionalidade do método (MARTINS et al., 2010);
d) **Métodos Mortos**. Métodos não acessíveis (não possuem chamadas) a partir de outros métodos acessíveis. Por causa de sua inacessibilidade, esses métodos não são executados durante o funcionamento do sistema de software (ROMANO et al. 2016).

Dentre os diferentes tipos de código morto, os métodos mortos podem ser considerados os mais prejudiciais para manutenibilidade de sistemas de software (ROMANO et al., 2016). Isso ocorre pelo fato de possuírem maior quantidade de linhas de código em relação aos outros tipos, contribuindo de forma mais significativa com o aumento desnecessário do tamanho do código, da poluição e da complexidade. Além disso, os métodos mortos não estão vinculados a requisitos de sistemas de software, prejudicando sua rastreabilidade. Outro problema causado por esse tipo de código morto é a existência de código não testado, podendo ocasionar falhas e anomalias durante o funcionamento do sistema de software caso o método torne-se acessível e ativo novamente (MARTIN, 2008).

As constantes modificações realizadas em sistemas de software são as principais causas do surgimento de métodos mortos. Essas modificações podem ocorrer para (i) substituição de métodos obsoletos, (ii) atualização de tecnologias, (iii) atualização de funções, (iv) troca de requisitos e (v) desativação de funções, fazendo com que métodos sejam adicionados/substituídos frequentemente, de forma a contribuir para a quantidade de código morto aumentar sem a percepção dos desenvolvedores (GOLD; MOHAN, 2003; SCANNIELLO, 2011). Esses fatores colaboram com o aumento da complexidade ao longo da evolução desses sistemas, conforme abordado na **Lei da Complexidade Crescente** (BELADY; LEHMAN, 1985).

A detecção de código morto pode ser realizada utilizando análise estática ou análise dinâmica do código. A precisão dos resultados obtidos com a análise dinâmica pode superar a precisão da análise estática (ROMANO et al., 2016). No entanto, a detecção de código morto com análise dinâmica depende da execução de vários cenários de uso, fazendo necessária a análise de “todas” as chamadas possíveis aos métodos do sistema de software em tempo de

execução. Dessa forma, a detecção de código morto com análise estática é considerada mais flexível por não depender da execução do sistema para ser efetuada.

Existem estudos que propuseram/aplicaram técnicas para detectar diferentes tipos de código morto em diversos contextos na área de computação. Dessa forma, foi necessário fazer uma busca sistemática na literatura para identificar e caracterizar as técnicas de detecção de código morto existentes, bem como identificar seu domínio de aplicação. Uma Revisão Sistemática de Literatura (RSL) foi realizada e os resultados obtidos foram analisados para identificar a técnica de detecção de código morto mais adequada para ser utilizada neste trabalho (BASTOS et al., 2016d).

5.2 Técnicas de detecção de código morto: RSL

A Revisão Sistemática de Literatura (RSL) foi utilizada por ser um processo metodológico cuidadosamente controlado por um protocolo de investigação formal e por fornecer consistência e robustez nos seus resultados. A RSL é dividida em três fases (BIOLCHINI et al., 2007):

a) **Planejamento**. É elaborado um protocolo que contém a descrição formal da metodologia utilizada na condução da RSL. De modo geral, neste protocolo, são definidas questões que motivam a execução da pesquisa, a *string* de busca, as estratégias para coleta das informações e os critérios de inclusão/exclusão dos estudos;
b) **Execução**. Envolve a identificação, a seleção e a avaliação dos estudos coletados de acordo com os critérios definidos no protocolo;
c) **Análise**. É realizada uma síntese dos resultados obtidos na fase anterior. Os estudos selecionados são analisados de modo a obter informações que auxiliam a responder as questões de pesquisa definidas no protocolo.

A metodologia utilizada na condução desta pesquisa é descrita no Apêndice A. Os artigos selecionados na RSL (estudos primários) foram analisados de forma qualitativa para responder a questão de pesquisa definida no protocolo. Esse tipo de análise é útil para sintetizar os resultados obtidos na RSL (LIU, 2014). A questão de pesquisa é:

`Quais são as técnicas de detecção de código morto existentes na literatura?`

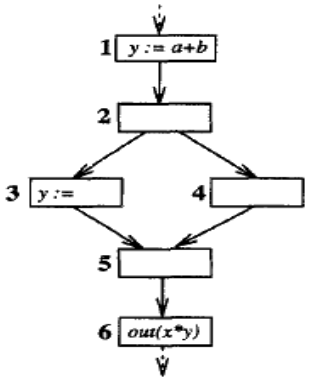
Foram encontrados artigos relacionados com técnicas de detecção de código morto aplicadas em diferentes contextos. A referência desses artigos e o código de identificação

definido neste trabalho estão listados no Apêndice B. A **análise de fluxo de dados** e a **análise de acessibilidade** foram as técnicas de detecção mais citadas, entretanto aplicadas de maneiras distintas.

5.2.1 Análise de fluxo de dados

De modo geral, a técnica Análise de Fluxo de Dados é indicada para detectar trechos de código que produzem resultados não utilizados durante a execução do sistema de software, por exemplo, atribuições em variáveis mortas e valores não utilizados retornados por métodos. Essa técnica também é indicada para detectar atribuições parcialmente mortas, que ocorrem quando o valor produzido pela instrução não é utilizado em determinado caminho de execução, porém utilizado em outros caminhos (KNOOP et al., 1994). Na Figura 5.1, o fluxo de execução dado pela sequência 1, 2, 3, 5 e 6 gera uma atribuição parcialmente morta, pois o valor atribuído a variável `y` no bloco 1 não é utilizado, sendo substituído pela atribuição efetuada no bloco 3. Entretanto, para o fluxo 1, 2, 4, 5 e 6 não acontece a substituição do valor de `y`.

Figura 5.1 - Exemplo de Código Parcialmente Morto



Fonte: Knoop *et al*. (1994).

Com a RSL, foram encontrados artigos que utilizam a técnica Análise de Fluxo de Dados para identificar código morto. Em [A31][1], *bytecodes* Java foram convertidos para a forma *Static-Single Assignment* (SSA) e representados por uma árvore. SSA é uma representação intermediária que fornece informações relacionadas ao fluxo de dados, simplificando a implementação de otimizações. Antes de iniciar a análise, os componentes de código representados pelos nós da árvore são definidos como mortos. Posteriormente, esses componentes devem estar de acordo com uma das três condições de vivacidade para serem

[1] Ao longo desse capítulo, esse código refere-se aos artigos obtidos com a RSL (APÊNDICE B).

classificados como vivos: i) declarações que afetam a saída do sistema de software; ii) instruções que produzem resultados utilizados por instruções classificadas como vivas; e iii) declarações condicionais que possuam outras instruções vivas dependentes. Após a análise dos nós da árvore, os nós que continuam marcados como mortos são removidos.

Além da representação em árvore, é comum ser utilizada a representação em grafos para realizar a análise dinâmica de fluxo de dados, como realizado em [A53]. Essa análise é feita utilizando a representação binária do código por meio de um grafo. Em [A37], o código parcialmente morto é eliminado utilizando análise de caminhos frequentemente executados. Basicamente, o primeiro passo do algoritmo é verificar quais instruções estão presentes nos caminhos mais executados. Para cada caminho, os predicados dessas instruções são analisados; se não forem utilizados ao longo desse caminho, a instrução é definida como morta. Em [A19], a técnica Análise de Fluxo de Dados é feita após a identificação de quais partes do código devem ou podem ser analisadas. Além disso, em [A33], essa técnica é utilizada sob demanda e na ordem inversa do grafo (*bottom-up*).

Em [A47] e [A41], o sistema de software é dividido em diversos blocos. A detecção de código morto é realizada com a técnica Análise de Fluxo de Dados em cada bloco. Em [A3], foi utilizada a representação SDDA (*Speculative Data Dependence Analysis*), desconsiderando as dependências de dados com baixa probabilidade de serem otimizadas. Em seguida, essa técnica foi utilizada e as atribuições consideradas parcialmente mortas foram movidas para ramos do grafo em que se tornam vivas. Outra abordagem baseada na análise de divisões de sistemas de software foi proposta em [A23]. O sistema de software é dividido em “fatias” de acordo com sua semântica. Cada fatia é representada por um subgrafo para a técnica Análise de Fluxo de Dados.

Em [A36], são apresentados algoritmos para identificar código parcialmente morto que consiste no fatiamento do código para reestruturação do fluxo de execução. Em [A20], a detecção de código morto é realizada em aplicações JavaScript. Os eventos do sistema de software são modelados com grafos que representam o fluxo de dados. Os nós do grafo não executados são considerados mortos. Com o fluxo de dados, é possível realizar análise estática. Em [A22], o algoritmo proposto divide o sistema de software em regiões analisadas separadamente e representadas por nós do grafo. Em [A34], além do grafo ser analisado de maneira *top-down*, como nas outras abordagens, é realizada a análise *bottom-up*. Em [A46], a detecção é realizada de maneira estática, entretanto o objetivo é a análise em aplicativos para Android.

Também foram encontrados estudos relacionados com detecção de código morto em sistemas de software concorrentes. Em [A29], a técnica Análise de Fluxo de Dados foi utilizada para detecção de variáveis "vivas" do tipo ponteiros em sistemas de software com *threads*. Em [A12], foi proposta uma adaptação da técnica Análise de Fluxo de Dados tradicional. Quando uma atribuição parcialmente morta é encontrada, ela é movimentada para ambos os ramos do grafo. Assim, a atribuição torna-se morta em um dos ramos e pode ser eliminada. O autor define esse procedimento em dois passos: i) *code sinking*[2]; e ii) eliminação de código morto. Considerando que otimizações realizadas em sistemas de software paralelos não podem prejudicar o desempenho, foi definida uma restrição cuja otimização em um componente paralelo é realizada apenas se for propagada para os outros componentes.

Outra abordagem para otimização de sistemas de software paralelos foi dada em [A27]. O algoritmo proposto inicia com a marcação das declarações desses sistemas como mortas. Esse conjunto de declarações é armazenado em uma lista utilizada para acompanhamento ao longo da execução do sistema de software. A cada instrução marcada como viva, a lista é atualizada. As regras para uma declaração ser considerada viva são equivalentes às apresentadas em [A31]. A mesma abordagem proposta em [A29] foi utilizada em [A21], entretanto para sistemas de software não paralelos, fazendo com que a definição da restrição seja desnecessária. Em [A42], antes do *code sinking*, é feita uma análise de disponibilidade para verificar onde essa operação pode ser feita e os impactos que podem acarretar. Em [A28], análise de dependência foi realizada para detecção de parâmetros não utilizados em procedimentos. Para uma dada variável `l`, é feito um cálculo para verificar as variáveis que dependem de `l` no procedimento. Caso não seja encontrada variável dependente, `l` é considerada uma variável morta.

Dois estudos propuseram a integração de métodos matemáticos com a técnica Análise de Fluxo de Dados. Em [A18], o grafo de controle de fluxo, determinado SPTerm, é construído utilizando expressões algébricas. A detecção de código morto é realizada com análise *top-down* e *bottom-up* no grafo. Em [A39], essa técnica foi utilizada de maneira tradicional, porém foram empregadas demonstrações matemáticas para provar a eficiência da técnica. Outra variação da técnica Análise de Fluxo de Dados foi apresentada em [A4], cuja detecção de código morto é realizada com auxílio de tabelas *Hash*, considerando o conceito de propagação de cópias de variáveis. No primeiro passo, é feita uma análise para identificação de cópias e as informações obtidas são armazenadas em tabelas *Hash*. No

[2] Movimentação do código para nós subjacentes no grafo de análise.

segundo passo, as informações contidas na tabela *Hash* são analisadas para detecção do código morto.

Em [A52], foi realizado um estudo para detecção de atribuições mortas utilizando análise de pilhas e *bytecodes* Java. A análise de código morto baseada em pilha não é simples, pois as instruções empilhadas podem não estar de maneira sequencial, dificultando a identificação de dependências entre essas instruções. A abordagem de eliminação de código morto proposta consiste em duas etapas: i) as instruções de armazenamento, de adição e de saltos condicionais são selecionadas; e ii) as instruções selecionadas são analisadas e eliminadas, se consideradas mortas.

5.2.2 Análise de acessibilidade

A técnica Análise de Acessibilidade é utilizada para verificar os componentes de software que não podem ser acessados e, consequentemente, nunca são executados. De modo geral, essa técnica é baseada na definição de dois conceitos apresentados em [A1] e [A35]:

a) **Conjunto de entidades alcançáveis**. Conjunto de entidades `R` alcançáveis a partir de um conjunto de entidades que inicializam a execução do software;
b) **Conjunto de entidades fonte**. Conjunto `S` das entidades presentes no software.

O conjunto de entidades mortas é dado pela diferença entre os cunjuntos `S` e `R` (`S – R`). Assim, as entidades não alcançáveis não são necessárias para a execução do sistema de software e são consideradas como código morto.

Foram encontrados estudos que aplicam a técnica Análise de Acessibilidade para detectar código morto, entretanto com adaptações de acordo com o domínio na qual é aplicada. Em [A2], essa técnica foi utilizada em trechos de código considerados vulneráveis quanto à ocorrência de problemas são analisados (dirigida por propósitos). Esse tipo de análise implica em um custo mais baixo e consumo de menos recursos do computador. Em [A11], essa técnica foi utilizada em sistemas de software escritos na linguagem de programação Esterel.

Em [A5], a técnica Análise de Acessibilidade é utilizada no processo de engenharia reversa, em que o código Java é transformado em redes de Petri. Em seguida, é realizada a busca por componentes de código inacessíveis utilizando a análise da rede. Essa técnica depende do nível de abstração utilizado no modelo, pois quanto maior a quantidade de detalhes colocado no diagrama, maior o seu tamanho e a sua complexidade. A técnica Análise

de Acessibilidade proposta em [A6] efetua o rastreamento para cada componente `l` do sistema de software. Se `l` for um componente acessível, a consulta retorna o caminho até `l`. Caso contrário, é retornado uma prova que `l` é inacessível e, consequentemente, código morto.

Em [A26] e [A48], a técnica Análise de Acessibilidade é utilizada em código representado na forma SSA. Em [A48], o grafo gerado é analisado e os métodos não alcançáveis são definidos como mortos. Em [A26], o procedimento de detecção de código morto é equivalente, mas a representação SSA é gerada de maneira diferente, em que as relações do grafo são calculadas em termos de equações matriciais simples.

Em [A38], é realizada a detecção de código morto em sistemas de software embarcados utilizando a técnica Análise de Acessibilidade. Inicialmente, os métodos do sistema de software analisado são definidos como acessíveis. Em seguida, o conteúdo de cada método é analisado e as chamadas para outros métodos encontradas são armazenadas. Métodos não chamados são considerados inacessíveis e definidos como código morto. A solução proposta para encontrar código inacessível em [A14] é baseada em Autômato Invariante de Erro. Esse conceito pode ser considerado uma abstração de um fragmento de código de entrada que contém somente declarações e fatos relevantes para a compreensão das causas de inconsistências. Com o Autômato Invariante de Erro, pode-se descrever os estados alcançáveis a partir de determinado local do sistema de software. Em [A15], é utilizada a técnica Análise de Acessibilidade simbólica, uma técnica formal que verifica a inacessibilidade de instruções de código de maneira exaustiva.

5.2.3 Demais técnicas

Além das técnicas Análise de Acessibilidade e Análise de Fluxo de Dados, foram encontradas outras maneiras para detecção de código morto. Em [A40], foi realizada a detecção de código morto utilizando anotações de código Java (comentários de código). No exemplo apresentado na Figura 5.2, o `return 2` é considerado código morto, pois, de acordo com a anotação referente ao método `withPre (int x)`, o valor da variável `x` não pode ser menor que `10`. Assim, o código dentro da instrução `if` pode ser inútil.

Alguns estudos aplicaram métodos matemáticos para detecção de código morto. Em [A45], é utilizado o estimador de Kaplan Meier para estimar a probabilidade de um método ser “sobrevivente” nas futuras versões do sistema de software em análise. Em [A43], é realizada uma análise do sistema de software utilizando relacionamento de seus componentes.

Esses relacionamentos foram representados por formulações matemáticas e utilizados para detecção de código morto. Em [A24], foi apresentada uma abordagem cuja eliminação de código morto é realizada em sistemas de software funcionais por meio da combinação de inferências.

Figura 5.2 - Exemplo de Anotação de Código

```
/*@ requires x > 10;
  @ ensures
  @   \result == 1;*/
int withPre(int x) {
 if (x < 10) {
   // not checked
   return 2;
 }
 return 1;
}
```

Fonte: Janota et al. (2007).

Dois estudos propuseram a detecção de código morto utilizando análise de esquemas XML (*eXtensible Markup Language*). Um esquema é um documento que fornece meios para definição da estrutura, do conteúdo e da semântica de documentos XML [W3C, 2001]. Em [A51], foi realizada a análise de documentos XQuery juntamente com o esquema que descreve seu conjunto de restrições. Para cada expressão XQuery, é efetuada uma análise em seu esquema para validar o significado da expressão de acordo com as restrições definidas. É comum que expressões XPath não estejam de acordo com a especificação no esquema. Nesse caso, as instruções XQuery dependentes dessas expressões são consideradas código morto. Em [A13], é apresentada uma ferramenta que detecta código morto por meio da verificação das expressões XPath com o esquema que as define. Para detectar inconsistências, a ferramenta converte as expressões XPath em um esquema e realiza a comparação com o esquema original. Expressões que estejam em desacordo são consideradas código morto.

Em [A8], foi apresentado um algoritmo para definição de variáveis mortas em laços de repetição. Basicamente, esse algoritmo faz uma varredura do código para encontrar instruções de atribuição. As variáveis que recebem o conteúdo de uma atribuição são adicionadas ao conjunto de variáveis declaradas. Em seguida, é realizada outra análise para encontrar as variáveis atribuídas. Essas variáveis são adicionadas ao conjunto das variáveis utilizadas. Se uma variável está no conjunto de variáveis declaradas e não está no conjunto de variáveis utilizadas, ela é considerada código morto.

Uma abordagem diferente foi proposta em [A44] para detecção de código morto em aplicações de dispositivos móveis. Como sistemas de software embarcados possuem recursos de memória limitados, os arquivos de código da aplicação ficam hospedados em um servidor. De acordo com as requisições de uso, os arquivos necessários são transferidos para o

dispositivo. Com isso, podem ser analisados quais blocos nunca foram transferidos para o dispositivo, sendo desnecessário para aplicação e classificados como código morto.

Existem estudos que optaram por não analisar diretamente o código para detecção de código morto, mas suas representações. Em [A10], para suprir a necessidade de análise de código em diversas linguagens, foi proposto efetuar análise sob código Assembler. A abordagem utilizada em [A25] é uma análise do código representado na forma *Concurrent Static Single Assignment* (CSSA) em sistemas de software paralelos, utilizando um algoritmo para detecção de propagação de cópias responsável pela geração de código morto.

Em [A30], uma técnica foi proposta para detecção de código morto em código PHP (*Hypertext Preprocessor*). A técnica proposta realiza análise sobre os arquivos do sistema de software e verifica a quantidade de vezes que esse arquivo foi utilizado em tempo de execução. Quanto menor for a quantidade obtida para um arquivo, maior será a chance de ser classificado como “morto”. Esse tipo de análise pode produzir resultados não exatos, visto que arquivos em desenvolvimento não serão executados e serão rotulados como “morto”. Para resolver esse problema, a técnica proposta realiza uma verificação do tempo de acesso, ou seja, arquivos não executados, mas que possuam tempo de acesso recente, não serão considerados código morto.

Utilizando algumas medidas de software, é possível prever a existência de código morto em sistemas de software desenvolvidos em Java. Em [A9], foi investigada a relação existente entre medidas de software e código morto. As medidas analisadas foram propostas por Chidamber e Kemerer (CK) e algumas medidas são de tamanho de código (tradicionais). Como resultado, obteve-se que as medidas *Weighted Methods per Class* (WMC) e *Response For a Class* (RFC) propostas por CK e as medidas de tamanho *Number of Message Sends* (NMS), *Number of Methods* (NM) e *Lines of Code* (LOC) são importantes para previsão de código morto.

Na eliminação de código morto realizada em [A50], foram utilizadas *liveness patterns* baseados em gramáticas e árvores regulares para análise de sistemas de software recursivos. *Liveness patterns* são construtores responsáveis por indicar quais partes do código devem ser “mortos” e quais devem ser “vivos”. Foi realizada análise em pontos desse sistema com base nas restrições construídas a partir da semântica e da linguagem de programação.

5.2.4 Trabalhos sem sugestões de técnicas

O objetivo das seleções primária e secundária na RSL foi selecionar artigos que contém resposta para a questão de pesquisa. Como essas seleções são baseadas apenas na leitura do resumo e da conclusão, não é possível garantir que o artigo possui uma resposta para a questão de pesquisa. Com isso, alguns artigos selecionados foram relacionados com código morto, entretanto não propuseram técnicas para detecção.

Em [A7], foi realizado um estudo para verificar a quantidade de código morto que poderia ser gerado com decisões erradas na etapa de projeto do software. Em [A17], foi apresentada uma ferramenta de suporte para análise dinâmica de código sob forma binária. Várias versões de um sistema de software foram analisadas utilizando comparação de classes e de métodos entre as versões. As informações adquiridas são apresentadas para o analista, não sendo realizada a detecção automática de código morto. Em [A32], foi apresentada uma ferramenta cujo objetivo é realizar detecção de escritas não utilizadas na memória, o que foge no contexto da RSL.

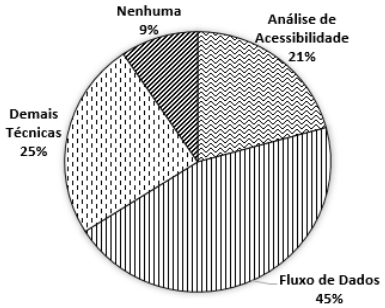
Em [A49], foi apresentada uma abordagem para realizar mutação da hierarquia de classes Java para capturar o comportamento de objetos. Além disso, o estudo propôs uma ferramenta que implementa essa abordagem e realiza análises estáticas adicionais, como detecção de código morto. As técnicas utilizadas para realização das otimizações não foram descritas. Em [A16], foi proposta uma ferramenta para geração de testes a partir da análise de acessibilidade dos métodos. A ferramenta pode ser utilizada para detecção de código morto. Os detalhes do modo em que a detecção é realizada não foram fornecidos.

5.2.5 Discussão dos resultados

As principais técnicas de detecção de código morto encontradas foram **Análise de Fluxo de Dados** e **Análise de Acessibilidade**. Além dessas, foram encontradas algumas técnicas referenciadas em menor quantidade, como a Análise de Anotações de Código, Estimador de Kaplan Meier, Análise de Arquivos XPath, com utilização da usabilidade de arquivos e com utilização de medidas de software. A técnica Análise de Fluxo de Dados foi a mais abordada, sendo referenciada em quase metade dos artigos selecionados (FIGURA 5.3). As técnicas menos referenciadas foram contabilizadas em “Demais Técnicas”. Alguns artigos não propuseram técnicas de detecção de código morto, representando 9% do total.

Os artigos selecionados na RSL consideraram diferentes granularidades de código morto. Existem trabalhos que o consideraram como componentes de código que nunca são executados. Outros trabalhos relacionaram código morto a componentes de código que produzem resultados não utilizados. Nesse contexto, ainda existe o conceito de código parcialmente morto, cujos componentes de software produzem resultados não utilizados dependendo do caminho da execução.

Figura 5.3 - Quantidade de Artigos por Técnica



Fonte: Do autor (2017).

Com as informações apresentadas na Tabela 5.1, pode-se observar que algumas técnicas foram mais utilizadas de acordo com a definição de código morto considerada. Dos 20 estudos que definiram código morto como componentes de código não executados, 11 estudos (55%) utilizaram a técnica Análise de Acessibilidade. A técnica Análise de Fluxo de Dados foi utilizada por 3 estudos (15%). Outras técnicas variadas (30%) foram propostas para detecção desse tipo de código.

Tabela 5.1 - Classificação dos Estudos por Definições

	Código não Executado	**Resultados não Utilizados**	**Parcialmente Mortos**
Análise de Acessibilidade	[A1] [A35] [A2] [A11] [A5] [A6] [A26] [A48] [A38] [A15] [A14]	-	-
	11 artigos		
Análise de Fluxo de Dados	[A23] [A46] [A20]	[A31] [A53] [A47] [A3] [A34] [A29] [A27] [A28] [A18] [A39] [A4] [A52]	[A37] [A33] [A41] [A36] [A22] [A12] [A42] [A21]
	3 artigos	**12 artigos**	**8 artigos**
Outras técnicas	[A40] [A45] [A44] [A10] [A30] [A9]	[A24] [A51] [A13] [A8] [A25] [A50]	-
	6 artigos	**6 artigos**	

Fonte: Do autor (2017).

Foram encontrados 18 estudos que definiram código morto como componentes de software que produzem resultados não utilizados. Dentre esses estudos, 12 (66,67%) utilizaram a técnica Análise de Fluxo de Dados para detecção. Técnicas diversas foram

propostas por 6 estudos (33,33%). Não foram encontrados estudos que consideraram essa definição e utilizaram a técnica Análise de Acessibilidade para detecção. A técnica de detecção aplicada nos 8 estudos que utilizaram o conceito de código parcialmente morto foi a Análise de Fluxo de Dados. Dois estudos não foram classificados: i) em [A43], não foi apresentada a definição de código morto utilizada, e ii) em [A19], foi utilizada a técnica Análise de Fluxo de Dados considerando duas definições de código morto.

De acordo com os resultados, para detecção de código não executado, a técnica mais adequada é a análise de acessibilidade. Para detecção de código parcialmente morto e de componentes de código que produzem resultados não utilizados, a análise de fluxo de dados pode ser a técnica mais indicada. Além disso, foram identificados alguns artigos que propuseram ferramentas para análise de código morto. Esse fato levou a elaboração de uma nova pergunta:

`Quais artigos desenvolveram ferramentas para aplicação das técnicas de detecção de código morto propostas?`

Na Tabela 5.2, são apresentados o código dos artigos que propuseram ferramentas e a técnica de detecção de código morto utilizada. Apenas 8 artigos (15%) propuseram ferramentas. Pode-se observar que metade dos artigos que propuseram ferramentas não definiram a técnica utilizada.

Tabela 5.2 - Estudos que Propuseram Ferramentas

Código de Identificação	Técnica Utilizada
[A6]	Análise de Acessibilidade
[A13]	Outras técnicas
[A16]	Não sugeriu técnicas
[A17]	Não sugeriu técnicas
[A19]	Análise de Fluxo de Dados
[A30]	Outras técnicas
[A32]	Não sugeriu técnicas
[A49]	Não sugeriu técnicas

Fonte: Do autor (2017).

5.3 Considerações finais

Neste capítulo, foram apresentados conceitos básicos a respeito de código morto. Também, foram descritas técnicas para identificação desse código morto, obtidas com a execução de uma RSL. Após a análise de 53 artigos selecionados com a seleção secundária,

foram identificas duas técnicas mais referenciadas: i) Análise de Fluxo de Dados e; ii) Análise de Acessibilidade.

De modo geral, a técnica Análise de Fluxo de Dados verifica o fluxo de execução do sistema de software para identificar código morto considerando variáveis não utilizadas, trechos de código que produzem resultados não utilizados e código parcialmente morto. Em contrapartida, a técnica Análise de Acessibilidade visa encontrar trechos de código inacessíveis (não executados) a partir de um ponto de inicialização do sistema de software. Além disso, foram encontradas técnicas menos citadas, tais como, Análise de Anotações de Código e Baseadas em Conceitos Matemáticos.

Existem poucos estudos cujo objetivo principal é analisar código morto. Apesar da quantidade de artigos selecionados na RSL, aproximadamente metade dos estudos tratava somente desse assunto. Os outros estudos propuseram técnicas de análise que poderiam ser utilizadas em otimizações diversas, inclusive detecção de código morto, indicando possível carência de pesquisas realizadas de maneira exclusiva nessa área. Além disso, não foram encontrados estudos que utilizaram técnicas de visualização de software juntamente com as técnicas de detecção de código morto.

Com a análise das datas de publicação dos artigos selecionados, foi possível notar que o tema “código morto” foi constantemente estudado nos últimos anos. Isso mostra que há lacunas na área que motivam os pesquisadores a estudarem código morto em sistemas de software.

6 *DEAD CODE EVOLUTION VISUALIZATION* (DCEVizz)

A abordagem DCEVizz (*Dead Code Evolution Visualization*) realiza a identificação de código morto considerando métodos em versões de sistemas de software orientados a objetos. As características evolutivas do código morto identificado são apresentadas por meio de técnicas de visualização de software, sob as perspectivas quantitativa e qualitativa (BASTOS et al., 2016a). Essa abordagem foi implementada em um apoio computacional (*plug-in* para a plataforma Eclipse IDE), denominado DCEVizz Tool. Neste capítulo, são apresentados as características e o funcionamento da abordagem DCEVizz e do *plug-in* desenvolvido.

O restante do capítulo está organizado da seguinte forma. Visão geral das técnicas utilizadas e do funcionamento de DCEVizz é exibida na Seção 6.2. Características do *plug-in* DCEVizz Tool são descritas na Seção 6.3. Exemplos de como a abordagem pode ser utilizada e das informações que podem ser obtidas a partir de suas visualizações são apresentados na Seção 6.4.

6.1 Visão geral da abordagem

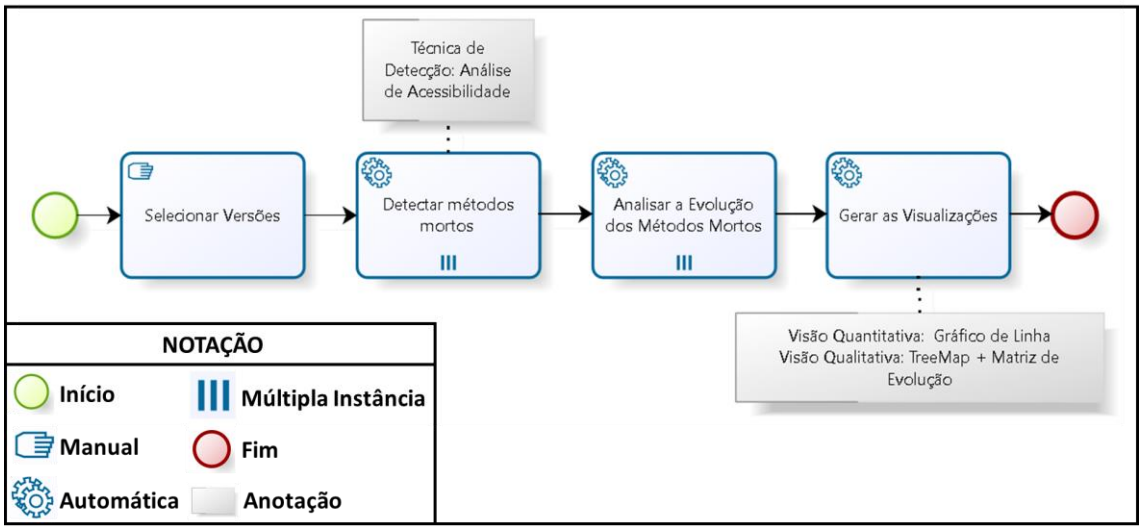
A abordagem DCEVizz consiste em detectar estaticamente código morto em versões de sistemas de software, identificar suas características evolutivas e apresentar os resultados utilizando técnicas de visualização de software. A granularidade de código morto considerada nessa abordagem é métodos, visto que a maior quantidade de linhas de código, a falta de vínculo com requisitos e a existência de código não testado fazem dessa granularidade o mais prejudicial para a manutenibilidade. Desse modo, a abordagem DCEVizz pode ser descrita em quatro etapas (FIGURA 6.1).

6.1.1 Selecionar versões

Informações históricas contidas implicitamente nas versões de sistemas de software são necessárias para compreender suas características evolutivas. Dessa forma, o primeiro passo da abordagem DCEVizz consiste em selecionar as versões para serem analisadas. Essa seleção é feita manualmente pelo engenheiro de software, que pode selecionar quantas versões forem necessárias. É importante que a análise e a identificação de código morto sejam

iniciadas na versão mais antiga e finalizadas na versão mais recente, para os resultados obtidos reflirem corretamente a evolução dos métodos mortos ao longo das versões.

Figura 6.1 - Visão Geral da Abordagem DCEVizz



Fonte: Do autor (2017).

6.1.2 Detectar métodos mortos

O objetivo é identificar estaticamente métodos mortos presentes nas versões do sistema de software. Esses métodos são caracterizados como mortos por serem inacessíveis e não serem invocados por outros métodos, fazendo com que nunca sejam executados. De acordo com os resultados obtidos com a RSL (Capítulo 5), a técnica de detecção indicada para esse tipo de código morto é **Análise de Acessibilidade**. Em geral, as abordagens que utilizaram essa técnica executaram os seguintes passos (CHEN et al., 1998; BACON et al., 2003; ROMANO et al. 2016):

a) Identificar os métodos de inicialização do sistema de software;
b) Identificar métodos acessíveis pelos métodos de inicialização;
c) Identificar os métodos inacessíveis, obtidos pela diferença entre todos os métodos do sistema de software com os métodos acessíveis.

Uma característica comum das abordagens apresentadas é a necessidade de conhecimento prévio do sistema de software. A utilização da técnica Análise de Acessibilidade nessas abordagens depende de uma investigação preliminar do código (se for automatizado) ou da familiaridade do mantenedor (se for manual) para os métodos de inicialização do sistema serem identificados antes de iniciar a detecção de métodos mortos

propriamente dita. Essa familiaridade não é trivial, visto que a maioria das pessoas envolvidas no processo de manutenção não possuem conhecimento do sistema (LAYZELL; TJORTJIS, 2001). Desse modo, a técnica Análise de Acessibilidade foi utilizada de forma diferente em DCEVizz, evitando a análise prévia do sistema de software. Para tanto, foram considerados dois tipos de métodos mortos:

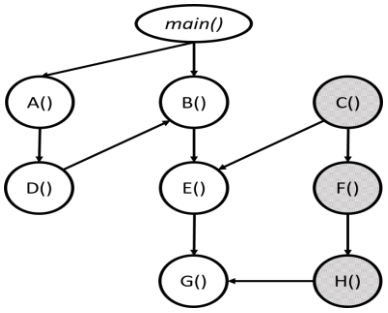
a) **Métodos Inacessíveis**. Métodos que não possuem chamadas a partir de outros métodos acessíveis;
b) **Métodos Acessíveis Apenas por Métodos Inacessíveis**. Métodos que possuem chamadas, porém todos os seus métodos chamadores são inacessíveis.

Dessa forma, a técnica análise de acessibilidade foi utilizada em duas fases, sendo que cada fase é responsável por identificar um tipo específico de método morto. O processo de detecção de métodos mortos da abordagem DCEVizz ocorre da seguinte forma:

a) **Abstração do Código Fonte**. As informações dos métodos de cada versão do sistema de software e as chamadas existentes entre eles são abstraídas em representações que permitem a execução da análise de acessibilidade. Representações baseadas em grafos ou em árvores são frequentemente utilizadas em abordagens que dependem de informações do código do sistema. Essas representações podem ser utilizadas na abordagem DCEVizz, na qual os nós correspondem aos métodos e as ligações entre os nós correspondem as chamadas existentes entre os métodos;
b) **Análise de Acessibilidade (Fase I)**. É realizada uma busca na representação do sistema de software para encontrar métodos inacessíveis. Esses métodos correspondem aos nós da representação que não possuem ligações ou arestas incidentes, sendo considerados inacessíveis por não possuírem chamadas de outros métodos do sistema. Na Figura 6.2, é apresentado um exemplo hipotético de representação em grafo dos métodos de um software (nós) e as chamadas existentes entre eles (arestas direcionadas). Nesse exemplo, após a execução da Análise de Acessibilidade (Fase I), o método `C()` é identificado como inacessível. Apesar do método `main()` não possuir chamadas, ele é desconsiderado por ser responsável pela inicialização da execução do sistema de software;
c) **Análise de Acessibilidade (Fase II)**. Após a identificação do conjunto de métodos inacessíveis, é realizada outra análise para identificar métodos chamados apenas por métodos inacessíveis pertencentes a esse conjunto, como o caso do método `F()` (FIGURA 6.2). Para cada método inacessível identificado, devem ser realizadas outras

análises, permitindo que sejam detectados métodos em níveis inferiores da representação, como o caso do método `H()` na Figura 6.2 Ainda nesse exemplo, apesar dos métodos `E()` e `G()` serem chamados por métodos inacessíveis (método `C()` e método `H()`, respectivamente), eles não são considerados mortos por possuírem chamadas de métodos acessíveis (método `B()` e método `E()`, respectivamente). Dessa forma, com Análise de Acessibilidade (Fase I) e Análise de Acessibilidade (Fase II), são identificados como mortos os métodos `C()`, `F()` e `H()`, destacados na cor cinza na Figura 6.2. Ambas as análises são realizadas em todas as versões do sistema de software;

Figura 6.2 - Exemplo de Representação dos Métodos do Software



Legenda: Métodos Mortos Destacados em Cinza.

Fonte: Do autor (2017).

d) **Armazenamento dos Resultados**. As informações de localização dos métodos identificados como inacessíveis devem ser armazenadas, sendo necessárias na etapa Análise da Evolução do Código Morto (etapa posterior). Essas informações consistem em versões, pacotes e classes nos quais esses métodos pertencem.

Vale ressaltar que a técnica Análise de Acessibilidade é executada utilizando as chamadas explícitas entre os métodos do software. Com isso, diferentes tipos de chamadas possíveis em linguagens orientadas a objetos podem não ser identificadas com essa técnica, por exemplo, chamadas por reflexão ou polimorfismo.

6.1.3 Analisar a evolução dos métodos mortos

De modo geral, a análise da evolução consiste em averiguar a existência de métodos mortos nas versões analisadas. Para cada método morto identificado em uma versão, é verificada a ocorrência de sua assinatura nas demais versões, atribuindo-o uma classificação. Na Figura 6.3, é apresentado um exemplo hipotético de duas versões de um sistema de software representadas por meio de grafos, sendo os métodos mortos destacados na cor cinza.

Considerando esse exemplo, na análise da **Segunda Versão**, pode-se atribuir uma das seguintes classificações aos métodos mortos:

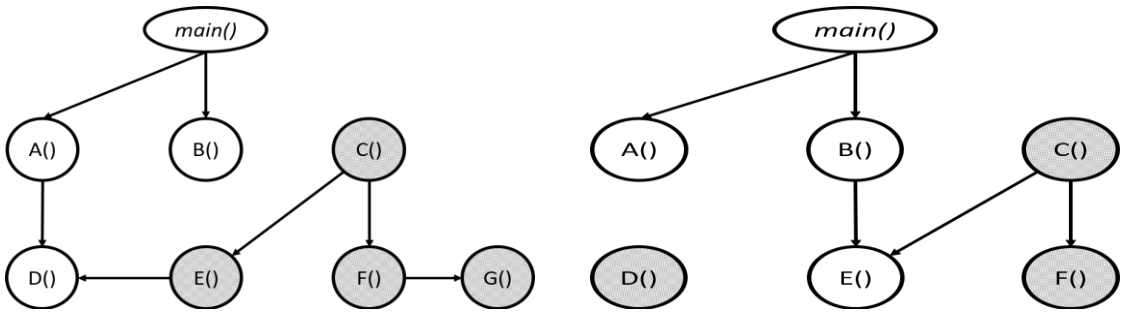
a) **Novo Método Morto**. Essa classificação é atribuída aos métodos mortos que não existiam na versão anterior, por exemplo, o método `D()`;

b) **Método Morto Propagado**. Essa classificação é atribuída aos métodos mortos que também eram mortos na versão anterior, por exemplo, o método `C()`;

c) **Método Morto Utilizado**. Essa classificação é atribuída aos métodos mortos na versão anterior e que tornaram-se acessíveis na versão em questão, por exemplo, o método `E()`;

d) **Método Morto Removido**. Essa classificação é atribuída aos métodos mortos na versão anterior e removidos na versão em questão, por exemplo o método `G()`.

Figura 6.3 - Evolução do Código Morto em Duas Versões (Software Hipotético)



Fonte: Do autor (2017).

Os métodos mortos da primeira versão analisada são classificados como "Novo Método Morto", visto que não é possível fazer comparações com versões anteriores. Desse modo, os métodos `C()`, `E()`, `F()` e `G()` ilustrados no grafo da **Primeira Versão** (FIGURA 6.3) recebem essa classificação. Todos os métodos mortos de todas as versões analisadas devem ser classificados. Os resultados obtidos são utilizados para renderizar as visualizações na etapa seguinte.

6.1.4 Gerar as visualizações

Os métodos mortos classificados durante a etapa Análisar a Evolução dos Métodos Mortos são representados por meio de duas perspectivas, utilizando técnicas de visualização de software: i) **quantitativa**, para perceber a variação da quantidade de código morto na evolução; e ii) **qualitativa**, para compreender detalhes do comportamento desse código em pacotes e classes específicos ao longo das versões. Quando utilizadas em conjunto, essas duas

perspectivas podem auxiliar na compreensão geral da evolução do sistema de software em relação a código morto.

A técnica de visualização quantitativa utilizada foi um gráfico de linha, no qual o eixo `y` corresponde a quantidade de código morto e o eixo `x` as versões analisadas. É apresentado na Figura 6.4 um protótipo dessa visualização, que exibe a evolução da quantidade de código morto de três versões de um sistema de software. Essa visualização facilita a percepção de tendência de aumento ou diminuição da quantidade desse código ao longo das versões do sistema.

Figura 6.4 - Protótipo da Visualização Quantitativa de DCEVizz

Fonte: Do autor (2017).

Na perspectiva qualitativa, são apresentados detalhes da evolução do código morto, utilizando uma técnica de visualização concebida com base nas técnicas TreeMap e Matriz de Evolução. Essas técnicas foram selecionadas considerando os fatores:

a) **Organização**. No contexto deste trabalho, a organização é um requisito importante da técnica de visualização, visto que auxilia na identificação da localização do método morto no sistema de software, facilitando a compreensão da sua evolução. Como sistemas de software orientados a objetos são estruturados hierarquicamente em pacotes, classes e métodos, técnicas de visualização hierárquicas (Seção 4.4) foram consideradas no momento da seleção. Com base nesses fatores, a técnica TreeMap foi selecionada por causa da sua capacidade em oferecer uma visão geral da estrutura das informações, utilizando cores para especificar características dos dados representados. A técnica de visualização proposta assemelha-se a essa pelas seguintes características:
   - **Cores das Bordas**. As cores das bordas dos retângulos são utilizadas para identificar a entidade de software sendo representada (versão, pacote, classe ou método);

- **Cores do Preenchimento**. As cores do preenchimento dos retângulos são utilizadas para representar características da entidade de software;
- **Agrupamento Hierárquico**. A hierarquia das entidades de software é representada utilizando agrupamento de retângulos em uma área de exibição plana.

Os fatores que diferenciam a técnica de visualização proposta da técnica TreeMap são:

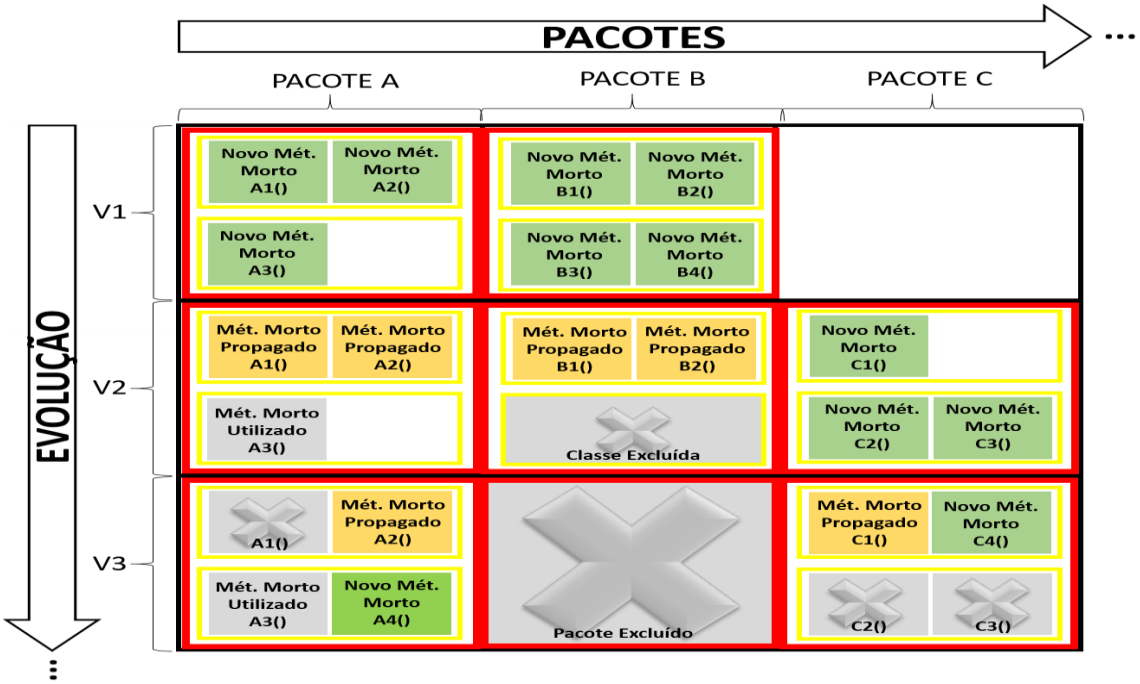
- **Uso de Espaço de Visualização Limitado**. Ao contrário da técnica TreeMap, a técnica proposta considera limitação de espaço apenas no sentido vertical para cada versão. No sentido horizontal, é utilizado o espaço necessário para representar os métodos mortos;
- **Tamanho de Retângulo Fixo**. A técnica TreeMap varia a proporção de tamanho dos retângulos de acordo com a limitação do espaço de visualização. Além disso, a variação da largura e da altura desses retângulos é utilizada para representar características da entidade do software correspondente, por exemplo, medidas de software. Esse recurso não foi utilizado na técnica proposta, visto que causaria dificuldade de compreensão da evolução dos métodos mortos;

b) **Evolução**. A seleção da técnica para representar a evolução foi realizada considerando técnicas que ofereçam uma visão geral da evolução de determinado atributo do sistema de software. Desse modo, a técnica Matriz de Evolução foi selecionada por apresentar a evolução das versões analisadas simultaneamente. Outro fator que levou a seleção dessa técnica foi sua capacidade de adaptação, podendo ser combinada com a técnica TreeMap.

Como ilustrado no protótipo apresentado na Figura 6.5, a técnica de visualização qualitativa possui a estrutura de uma matriz, em que as linhas correspondem as versões analisadas e as colunas correspondem aos pacotes dessas versões. No interior das linhas da matriz, são apresentados hierarquicamente os pacotes (bordas vermelhas), as classes (bordas amarelas) e os métodos mortos de cada versão analisada, conforme sugerido na técnica TreeMap. São utilizadas cores para caracterizar a evolução dos métodos mortos, que variam de acordo com a classificação obtida na etapa anterior. A cor verde foi utilizada para representar métodos mortos que não existiam na versão anterior (**Novo Método Morto**). A cor laranja foi utilizada para representar métodos mortos que existiam na versão anterior (**Método Morto Propagado**). A cor cinza foi utilizada para representar métodos mortos na

versão anterior e que se tornaram acessíveis (**Método Morto Utilizado**) na versão em questão. A cor cinza com símbolo “X” foi utilizada para representar métodos mortos na versão anterior e excluídos na versão em questão (**Método Morto Removido**).

Figura 6.5 - Protótipo da Visualização Qualitativa de DCEVizz



Fonte: Do autor (2017).

Não há limite quanto à quantidade de versões que podem ser analisadas, fazendo com que a matriz contenha quantas linhas forem necessárias. No exemplo da Figura 6.5, foram representadas três versões de um sistema de software (matriz com três linhas). A altura dessas linhas e dos retângulos que representam as classes e os pacotes é fixa. No entanto, a largura desses componentes varia de acordo com a quantidade de métodos mortos que devem ser representados em cada um. A altura e a largura dos retângulos que representam os métodos mortos são fixas. Ao contrário da técnica TreeMap, essa característica foi adotada na visualização proposta para não dificultar a compreensão da evolução desses métodos quando assumissem tamanhos diferentes em cada versão.

Para facilitar o acompanhamento da evolução, cada método morto é representado visualmente na mesma posição vertical em todas as versões analisadas. Pacotes e classes excluídos e que continham métodos mortos em algumas das versões analisadas anteriormente são representados com o símbolo “X”. Esse fato ocorre com `pacote B` na terceira versão (V3) e com a segunda classe presente no `pacote B` na segunda versão (V2). Métodos mortos classificados como Método Morto Removido também são representados com esse símbolo, como o caso do método `A1()` no `pacote A` em V3 e dos métodos `C2()` e `C3()`

no `pacote C` em V3. Versões que possuem pacotes ou classes que terão métodos mortos nas versões futuras são representadas com espaço em branco, como o caso do `pacote C` na primeira versão (V1).

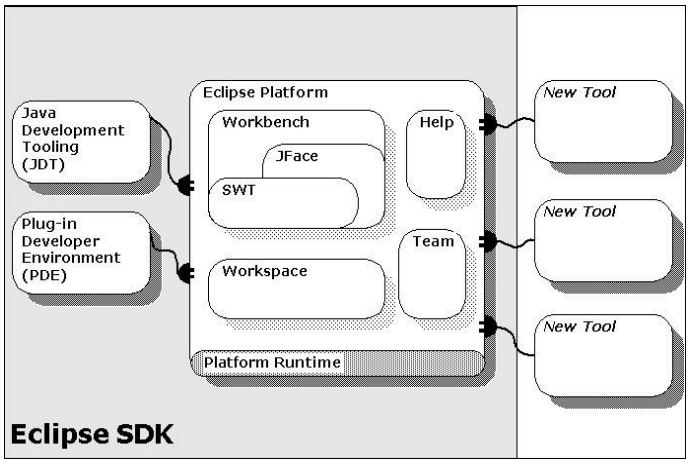
6.2 Apoio Computacional DCEVizz Tool

A abordagem DCEVizz foi implementada em um *plug-in* para a plataforma Eclipse IDE (DCEVizz Tool) para sistemas de software Java (BASTOS et al., 2016c). Nesta seção, são apresentadas as tecnologias e as características adotadas em DCEVizz Tool.

6.2.1 Tecnologias utilizadas

O *plug-in* DCEVizz Tool foi implementado na linguagem de programação Java (Java 8), utilizando o Eclipse SDK (*Software Development Kit*) na versão 4.4.2 (Luna). Essa IDE foi escolhida pela facilidade na adição de novas funções decorrente da sua estrutura baseada em *plug-ins* (MURPHY et al., 2006). Conforme apresentado na Figura 6.6, Eclipse SDK inclui uma plataforma base integrada com duas ferramentas úteis no desenvolvimento de *plug-ins* (JDT - *Java Development Tools* e PDE - *Plug-in Developer Environment*) (ECLIPSE, 2015).

Figura 6.6 - Eclipse SDK



Fonte: Eclipse (2015).

Com JDT, o acesso ao código é por meio de *Java Model* e/ou AST (*Abstract Syntax Tree*) (ECLIPSE, 2015). *Java Model* representa um sistema de software Java (projeto no Eclipse) utilizando um modelo definido pelos *plug-ins* classificados na categoria JDT *Core* e presentes no pacote `org.eclipse.jdt.core`. Esse modelo organiza as informações do

projeto em uma estrutura de árvore, cujos nós podem ser dos tipos apresentados na Tabela 6.1. Em contrapartida, AST representa código Java no formato de uma árvore sintática abstrata e oferece mais detalhes e recursos para manipulação do código do que *Java Model.*

Tabela 6.1. Componentes do *Java Model*

Elemento do Projeto	Representação no Java Model
Projeto Java	IJavaProject
Pastas src/bin ou bibliotecas externas	IPackageFragmentRoot
Pacotes	IPackageFragment
Arquivos de Código Java	ICompilationUnit
Tipos, Atributos e Métodos	IType/IField/IMethod

Fonte: Do autor (2017).

AST permite a criação, a leitura, a modificação e a exclusão de código de um sistema de software. Os *plug-ins* que permitem a manipulação de AST estão classificados na categoria JDT *Core* e estão localizados no pacote `org.eclipse.jdt.core.dom`. Conforme descrito na documentação da plataforma Eclipse IDE, por causa da maior robustez de AST, essa estrutura é computacionalmente mais custosa de ser gerada do que *Java Model*, além de que nem todas as aplicações necessitarem dos recursos de AST (ECLIPSE, 2015). Para executar a análise de acessibilidade em DCEVizz Tool, são necessárias apenas informações dos métodos do sistema de software, tais como, suas assinaturas e as chamadas existentes entre eles. Informações dessa granularidade são obtidas por *Java Model*, fazendo desnecessário o uso de AST.

PDE (*Plug-in Developer Environment*) é outro conjunto de *plug-ins* integrados à plataforma Eclipse IDE que fornece recursos para criação, desenvolvimento, teste e implantação de *plug-ins*. A perspectiva de *Plug-in Development* provida por PDE foi utilizada na implementação da abordagem, pois fornece um conjunto de visualizações que auxiliam na realização de algumas configurações, por exemplo, a criação do arquivo plugin.xml, definição de pontos de extensão e especificação de outros *plug-ins* necessários.

As representações visuais foram geradas utilizando recursos da biblioteca JFreeChart e da API (*Application Programming Interface*) *Swing*. JFreeChart foi utilizada para gerar a representação visual quantitativa, pois é uma biblioteca bem documentada e flexível e está disponível sem custos na internet (GILBERT, 2015). Essa biblioteca fornece recursos para geração de tipos variados de gráficos, tais como, gráfico de pizza, dispersão, histograma e linha, permitindo que as informações quantitativas sejam representadas de diferentes maneiras (GILBERT, 2009). API Java *Swing* foi utilizada na implementação da representação visual qualitativa. Essa API foi desenvolvida de modo a evoluir a funcionalidade de seu antecessor

(Java AWT - *Abstract Window Toolkit*) e foi escolhida por fornecer recursos extensíveis e configuráveis para construção de interfaces gráficas (SWING, 2015).

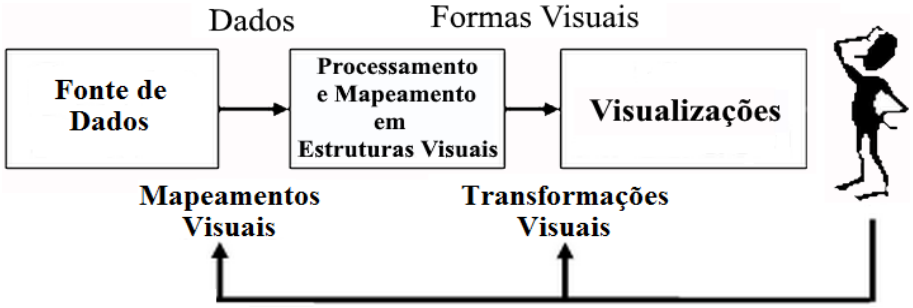
6.2.2 Desenvolvimento de DCEVizz Tool

O *plug-in* DCEVizz Tool foi implementado com base no modelo de referência de visualização da informação. Esse modelo é um padrão de arquitetura de software que divide o processo de geração da visualização em quatro etapas (CARD et al., 1999):

a) **Fonte de Dados**. Os dados a serem analisados e representados visualmente são coletados na fonte;
b) **Processamento de Dados**. Os dados são processados, filtrados e transformados em representações internas apropriadas para manipulação (*e.g.* estruturas de dados);
c) **Mapeamento em Estruturas Visuais**. As representações internas são mapeadas em estruturas de dados com as informações (cores, tamanhos e formas) necessárias para gerar as visões;
d) **Visualizações**. As estruturas de dados com os atributos visuais criados são utilizadas para criar as visões (renderização).

Em DCEVizz Tool, as etapas **Dados Processados** e **Estruturas Visuais** do modelo de referência foram condensadas em apenas uma etapa, denominada **Processamento e Mapeamento em Estruturas Visuais**, conforme apresentado na Figura 6.7.

Figura 6.7 - Modelo de Referência Utilizado em DCEVizz Tool



Fonte: Adaptado de Card *et al*. (1999).

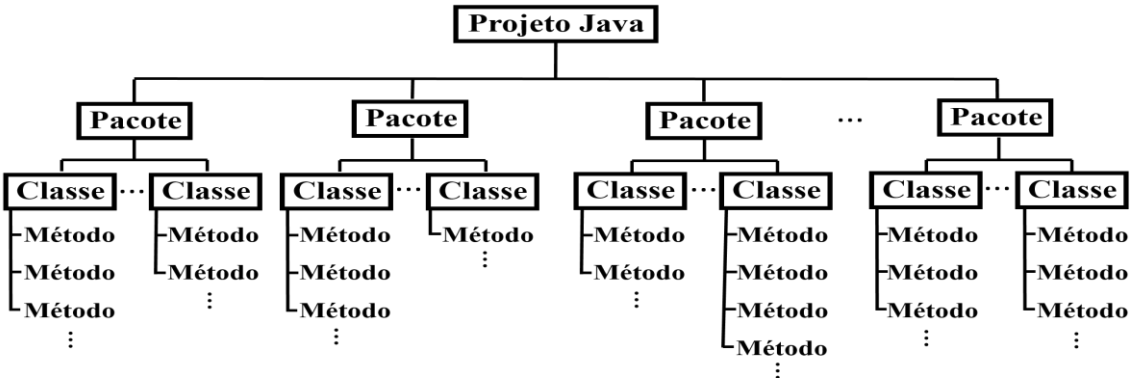
A adaptação nas etapas do modelo de referência permite que a identificação das características evolutivas e o mapeamento das características visuais sejam realizadas à medida que os métodos mortos são identificados. Como pode ser observado na Figura 6.7, além das etapas, o modelo sugere que os sistemas de visualização devem oferecer recursos de

interação ao usuário, que podem alterar o mapeamento ou a renderização dos componentes visuais.

6.2.2.1 Fonte de dados

Cada versão do sistema de software é analisada pelos *plug-ins* de JDT para sua decomposição em componentes de Java Model (TABELA 6.1). O resultado é a versão do sistema de software representada inteiramente em uma estrutura de árvore. Para cada versão, foi gerada uma árvore com as informações necessárias para realização das análises e para detecção de código morto. Uma representação simplificada da estrutura de *Java Model* para uma versão do sistema de software é apresentada na Figura 6.8, na qual o projeto Java (`IProject`) é decomposto em vários pacotes (`IPackage`), os pacotes são decompostos em classes (`ICompilationUnit`) e as classes são decompostas em métodos (`IMethod`).

Figura 6.8 - Representação Simplificada da Árvore *Java Model*



Fonte: Do autor (2017).

6.2.2.2 Processamento e mapeamento em estruturas visuais

A finalidade é identificar métodos mortos em versões do sistema de software, identificar suas características evolutivas, mapear as características visuais e armazenar em estruturas de dados (BASTOS et al., 2016c). No Algoritmo 1 (LISTAGEM 6.1), é apresentada uma visão geral dessa etapa, no qual a entrada são as estruturas de dados que representam as versões, obtidas na etapa Fonte de Dados.

Para cada versão analisada, o Algoritmo 1 inicia a execução da Análise de Acessibilidade - Fase I (linha 4) e da Análise de Acessibilidade - Fase II (linha 5). A saída desse algoritmo é as estruturas de dados `métodosMortos` e `métodosVivos`, cuja

representação simplificada pode ser visualizada na Figura 6.9(A) e Figura 6.9(B), respectivamente. A estrutura `métodosMortos` armazena uma coleção de métodos mortos (nome e informações de localização) para cada versão analisada, juntamente com uma lista de métodos chamados por eles e sua característica visual. A estrutura `métodosVivos` armazena uma coleção de métodos acessíveis para cada versão, juntamente com uma lista de métodos chamados por eles e uma lista de métodos que os chamam. Essas informações são necessárias para Análise de Acessibilidade - Fase II, evitando que novos acessos sejam efetuados na estrutura do *Java Model*.

Listagem 6.1 - Algoritmo 1 (Análise de Acessibilidade)

Algoritmo 1: Procesamento e Mapeamento em Estruturas Visuais

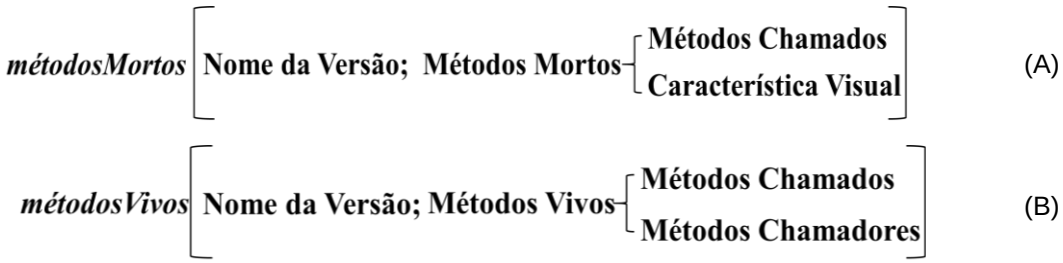
Entrada: *representacoes* (Conjunto de Estrutura de Dados *Java Model* para cada versão)

Resultado: *métodosVivos* (Estrutura de Dados com Métodos Acessíveis)
métodosMortos (Estrutura de Dados com Métodos Inacessíveis)

```
1  Início
2     declare métodosVivos, métodos Mortos          // estruturas de dados globais
3     para cada (versao ∈ representacoes) faça
4        executarAnaliseAcessibilidadeI (versao)    // Algoritmo 2
5        executarAnaliseAcessibilidadeII ( )        // Algortimo 3
6     fim para
7  Fim
```

Fonte: Do autor (2017).

Figura 6.9 - Estruturas de Saídas do Algoritmo 1



Fonte: Do autor (2017).

No Algoritmo 2 (LISTAGEM 6.2), é apresentada uma visão geral do funcionamento da Análise de Acessibilidade - Fase I. A entrada desse algoritmo é informações dos métodos da versão e das chamadas existentes entre eles, abstraídas em uma estrutura de dados de *Java Model*. A condição da linha 5 desconsidera alguns métodos da análise de acessibilidade. Métodos `main()` foram desconsiderados por serem responsáveis pela inicialização do sistema de software. Construtores e métodos abstratos de interface foram desconsiderados por não fazerem parte do escopo da proposta (analisar a acessibilidade de métodos concretos). Além disso, métodos relacionados aos eventos (*listeners*) foram desconsiderados por causa

das dificuldades em detectar suas chamadas utilizando análise estática (ROMANO et al., 2015).

Listagem 6.2 - Algoritmo 2 (Análise de Acessibilidade)

```
Algoritmo 2: Análise de Acessibilidade (Fase I)
     Entrada:     versao (Estrutura de Dados Java Model)
     Resultado:  preenchimento parcial das estruturas métodosVivos e métodosMortos
 1  Início
 2    para cada (pacote ∈ versao) faça
 3      para cada (classe ∈ pacote) faça
 4        para cada (metodo ∈ classe) faça
 5          se (metodo ≠ main | contrutor | listener | método de interface) então
 6            metodosChamadores ← obterChamadoresDoMetodo(metodo)
 7            metodosChamados ← obterMetodosChamados(metodo)
 8            se (metodosChamadores é vazio) então  // o método é inacessível (morto)
 9              adiciona metodo, metodosChamados em métodosMortos
10              fazerMapeamentoVisual(metodo)
11            senão                                  // o método é acessível (vivo)
12              adiciona metodo, metodosChamadores, metodosChamados em
                    métodosVivos
13            fim se
14          fim se
15        fim para
16      fim para
17    fim para
18  Fim
```

Fonte: Do autor (2017).

Como resultado, o Algoritmo 2 "preenche" as estruturas `métodosMortos` e `métodosVivos` com os métodos acessíveis e inacessíveis de cada versão, respectivamente. O mapeamento das características visuais é realizado na linha 10, variando de acordo com a classificação apresentada na Seção 6.2.3.

A Análise de Acessibilidade - Fase II visa encontrar métodos acessíveis apenas por métodos inacessíveis, não identificados pelo Algoritmo 2 por possuírem uma lista de métodos chamadores. Uma ideia geral do seu funcionamento é apresentada no Algoritmo 3 (LISTAGEM 6.3). Basicamente, esse algoritmo cria uma fila com os métodos inacessíveis (linha 2) e utiliza o conceito de busca em largura para percorrer as chamadas desencadeadas a partir de cada método pertencente a fila. Os métodos cujo conjunto dos chamadores estejam contidos na estrutura `métodosMortos` (condição da linha 9) também são considerados inacessíveis e inseridos no final da fila, permitindo que os métodos chamados por eles também sejam analisados. Essa busca é finalizada quando todos os métodos inacessíveis forem identificados.

Listagem 6.3 - Algoritmo 3 (Análise de Acessibilidade)

Algoritmo 3: Análise de Acessibilidade (Fase II)

Entrada: nenhuma

Resultado: *estrutura*: *métodosMortos* com todos os métodos mortos identificados no software

```
1  Início
2     fila ← criarFila(métodosMortos)
3     enquanto (fila não vazia) faça
4        raiz ← removerDaFila(fila)        // remove o primeiro método da fila
5        métodosChamados ← métodos chamados pela raiz em métodosMortos
6        para cada (metodo ∈ métodosChamados) faça
7           se (metodo ∉ métodosMortos) então
8              chamadores ← métodos que chamam metodo em métodosVivos
9              se (chamadores ⊂ métodosMortos) então
10                adiciona metodo em métodosMortos
11                remove metodo de métodosVivos
12                fazerMapeamentoVisual(metodo)
13                adicionaNaFila(metodo)
14             fim se
15          fim se
16       fim para
17    fim enquanto
18 Fim
```

Fonte: Do autor (2017).

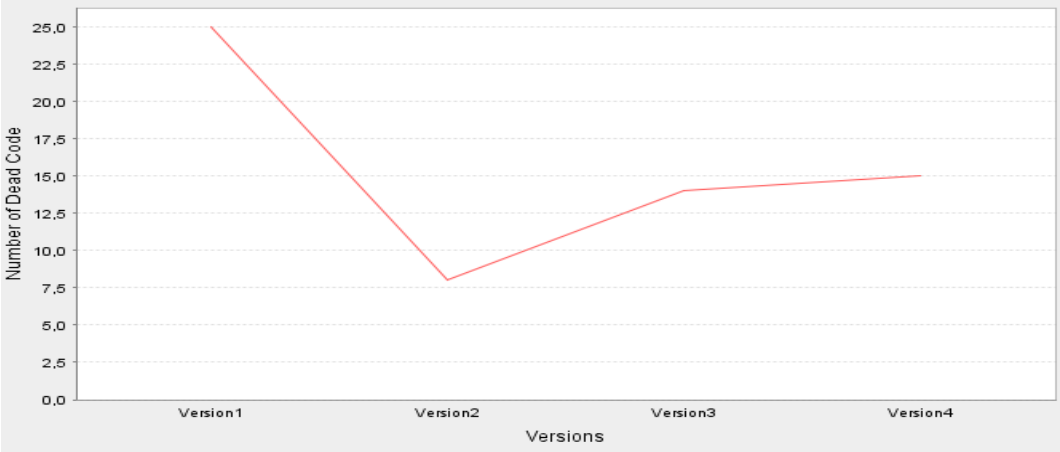
O mapeamento das características visuais dos métodos identificados como inacessíveis é realizado na linha 12. Como resultado, esse algoritmo finaliza o preenchimento das estruturas `métodosMortos` e `métodosVivos` com as informações necessárias para as visualizações serem renderizadas.

6.2.2.3 Visualizações

Representações visuais quantitativas e qualitativas foram implementadas e integradas ao *plug-in* DCEVizz Tool. Na Figura 6.10, é apresentada a representação visual quantitativa, no qual o eixo `X` contém as versões do sistema de software em análise e o eixo `Y` contém a quantidade de código morto presente em cada versão. Como pode ser observado, esse gráfico pode indicar que possível manutenção perfectiva foi realizada entre as versões 1 e 2, pois a quantidade de código morto diminuiu. Em contrapartida, essa quantidade aumentou continuamente a partir da Versão 2. Esse fato pode indicar que versões futuras desse sistema podem surgir com maior quantidade de código morto e que sua eliminação seria interessante para prevenir esse aumento.

A representação visual qualitativa foi concebida com base nas técnicas de visualização TreeMap e Matriz de Evolução. Nessa visualização, a evolução é mostrada pela organização das versões, dos pacotes, das classes e dos métodos em uma matriz, conforme apresentado na Figura 6.11. Nessa figura, são representadas duas versões de um sistema de software, sendo as versões representadas nas linhas (bordas pretas) e os pacotes (bordas vermelhas), as classes (bordas amarelas) e os métodos mortos (bordas pretas com espessura menor) representados nas colunas. A representação hierárquica[3] dos métodos mortos visa facilitar sua localização no sistema de software e foi baseada na técnica TreeMap.

Figura 6.10 - Representação Visual Quantitativa



Fonte: Do autor (2017).

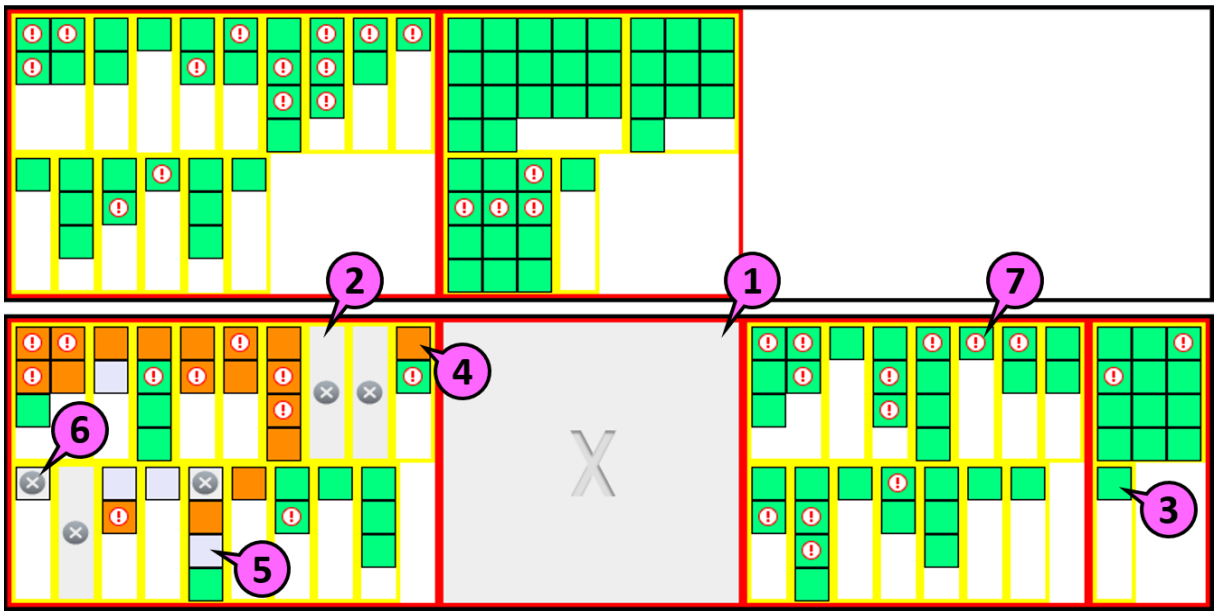
Para facilitar a compreensão da evolução, os componentes do software foram organizados de maneira posicional, ou seja, os mesmos pacotes, as mesmas classes e os mesmos métodos presentes em cada versão são colocados na mesma posição vertical. Nessa visualização, as diferenças relacionadas ao código morto são destacadas visualmente utilizando cores e símbolos no preenchimento dos componentes. A numeração presente na Figura 6.11 destaca as possíveis representações:

a) **Representação 1**. A cor cinza com o símbolo “ X ” significa que o pacote existiu em alguma versão anterior e foi removido na versão. No exemplo, o pacote existia na primeira versão e tinha métodos mortos, mas esse pacote foi excluído na segunda versão;

b) **Representação 2**. A cor cinza com o símbolo “⊗” significa que a classe existiu em alguma das versões anteriores e foi removida na versão. No exemplo, a classe existia na primeira versão e tinha métodos mortos, mas ela foi excluída na segunda versão;

[3] Representação visual dos métodos dentro de sua respectiva classe e pacote.

Figura 6.11 - Representação Visual Qualitativa



Fonte: Do autor (2017).

c) **Representação 3**. A cor verde significa que o método morto não existia na versão anterior. Essa cor é a única possível na primeira versão do sistema de software analisado;

d) **Representação 4**. A cor laranja significa que o método morto foi propagado de uma versão para outra. Ele existia na versão anterior e continua existindo na versão;

e) **Representação 5**. A cor cinza significa que o método era morto na versão anterior e deixou de ser morto na versão;

f) **Representação 6**. A cor cinza com o símbolo "⊗" significa que o método era morto na versão anterior e foi excluído na versão;

g) **Representação 7**. O símbolo "⊙" significa que o método é morto por ser chamado apenas por métodos mortos.

Pacotes e classes que não possuem métodos mortos não são apresentados na visualização. Na Representação 1, o pacote foi representado pois, em pelo menos uma das versões anteriores, ele existia e tinha classes com métodos mortos. O mesmo acontece na classe apresentada na Representação 2, apesar dela não existir na segunda versão, ela foi representada na visualização. Na Representação 6, a classe não possui método morto e apenas foi representada visualmente pelo fato de que na versão anterior ela tinha métodos mortos. Além disso, pode-se perceber que os métodos mortos do terceiro e quarto pacotes da segunda versão não existiam na primeira versão, sendo representados na cor verde. Caso a visualização

exceda o tamanho da *view* da plataforma Eclipse IDE, barras de rolagens vertical e horizontal aparecem, permitindo navegabilidade para a visualização.

6.2.3 Recursos de interação de DCEVizz Tool

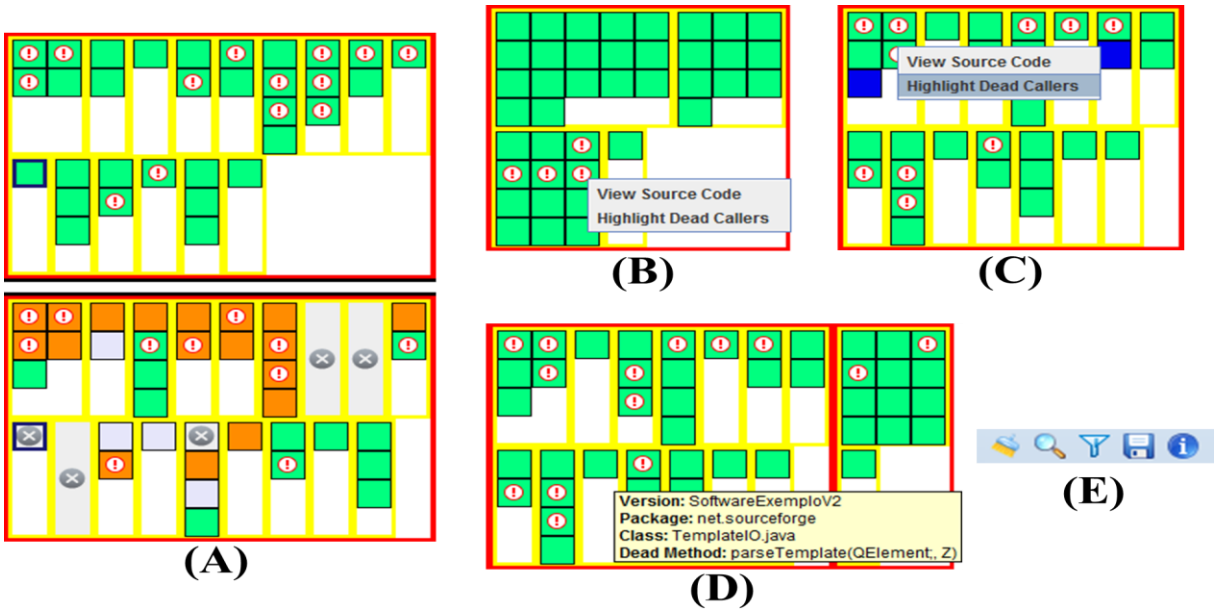
Foram implementados alguns recursos de interação com o *plug-in* DCEVizz Tool, apresentados na Figura 6.12:

a) **Figura 6.12A**. Ao clicar em cima de qualquer representação de um método morto, sua borda é destacada na cor azul nas versões analisadas, permitindo o acompanhamento de sua evolução;

b) **Figura 6.12B**. Ao clicar com o botão direito em cima da representação do método morto, aparece um *menu* com opção para visualizar o código. Uma segunda opção aparece nesse *menu* se o método morto selecionado for acessível apenas por métodos mortos, permitindo o destaque na cor azul dos métodos chamadores;

c) **Figura 6.12C**. Destaque na cor azul para os métodos mortos chamadores de um método morto;

Figura 6.12 - Recursos de Interação de DCEVizz Tool



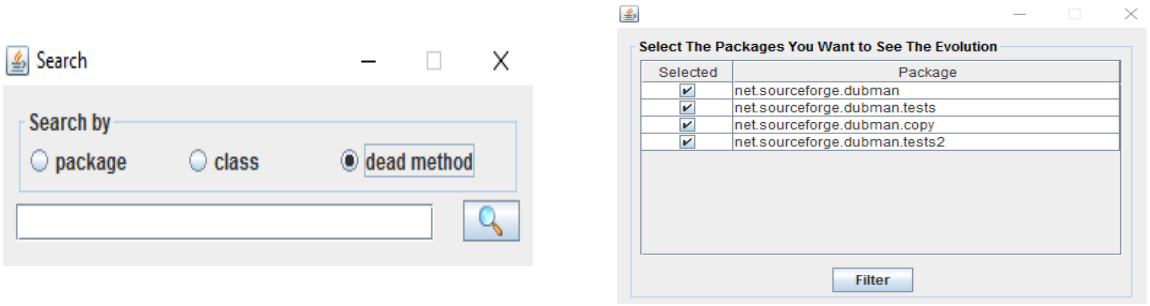
Fonte: Do autor (2017)

d) **Figura 6.12D**. Ao passar o *mouse* sobre a representação de pacotes, de classes ou de métodos, suas informações são apresentadas;

e) **Figura 6.12E**. São disponibilizados botões na *view* da plataforma Eclipse IDE. O botão "🧹" permite limpar qualquer destaque realizado na visualização. O botão "🔍"

permite buscar por pacote, classe ou método morto (FIGURA 6.13A). O botão “” permite filtrar pacotes, exibindo apenas os métodos mortos dos pacotes selecionados (FIGURA 6.13B). O botão “” permite salvar um arquivo de *log* no formato XML, contendo as informações do código morto identificado. Um exemplo parcial desse arquivo é apresentado na Figura 6.14. O botão “” contém uma legenda das representações visuais, permitindo a compreensão da visualização (FIGURA 6.15).

Figura 6.13 - *Menus* de Interação de DCEVizz Tool



(A) (B)

Fonte: Do autor (2017).

Figura 6.14 - Trecho do Arquivo de *Log* de DCEVizz Tool

```
<?xml version="1.0" encoding="UTF-8"?>
<resultsDCEVizz>
    <timeDeadCodeDetection startTime="09:52:45" endTime="09:54:12" totalTime="0:1:27"/>
    <timeGenerationViews startTime="09:54:12" endTime="09:54:13" totalTime="0:0:0"/>
    <version name="SoftwareExemploV1" numberOfPackages="12" numberOfClasses="27" numberOfMethods="346" numberTotalOfDeadMethods="71"
    numberOfDeadMethodsWithoutCallers="53" numberOfDeadMethodsWithCallers="18" numberOfConstructors="24" detectionTime="0:0:37">
        <package name="net" quantityDeadMethods="0">
        </package>
        <package name="net.sourceforge" quantityDeadMethods="0">
        </package>
        <package name="net.sourceforge.dubman" quantityDeadMethods="31">
            <class name="MetaTemplateEditor.java" quantityDeadMethods="1" >
                <method name="loadMeta" signature="(QFile;)V" line="516" />
            </class>
            <class name="BulkSettingsDialog.java" quantityDeadMethods="0" >
            </class>
            <class name="DubMan.java" quantityDeadMethods="3" >
                <method name="getCurrentTemplate" signature="()QJobTemplate;" line="777" />
                <method name="getCurrentFile" signature="()QFile;" line="781" />
                <method name="runMetaTemplate" signature="(QMetaTemplate;)V" line="1225" />
            </class>
            <class name="Job.java" quantityDeadMethods="1" >
                <method name="toString" signature="()QString;" line="438" />
            </class>
            <class name="JobsGenerator.java" quantityDeadMethods="2" >
                <method name="generateJobList" signature="(QMetaTemplate;)QList;" line="157" />
                <method name="saveEmpty" signature="(QFile;)V" line="696" />
            </class>
            <class name="HelpNavButtonPanel.java" quantityDeadMethods="0" >
            </class>
```

Fonte: Do Autor (2017).

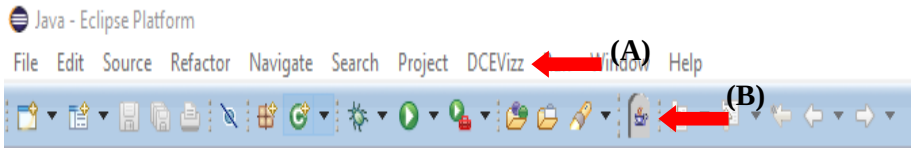
6.2.4 Execução de DCEVizz Tool

Uma apresentação prática da execução do *plug-in* DCEVizz Tool pode ser encontrada em <https://www.youtube.com/watch?v=Ax7Gyi6ebPw&feature=youtu.be>. Com o *plug-in* instalado na plataforma Eclipse IDE, são disponibilizados um botão na barra de ferramentas (FIGURA 6.16A) e uma opção na barra de menu (FIGURA 6.16B) que permite iniciar sua execução.

Figura 6.15 - Legenda das Representações Visuais de DCEVizz Tool

Fonte: Do autor (2017).

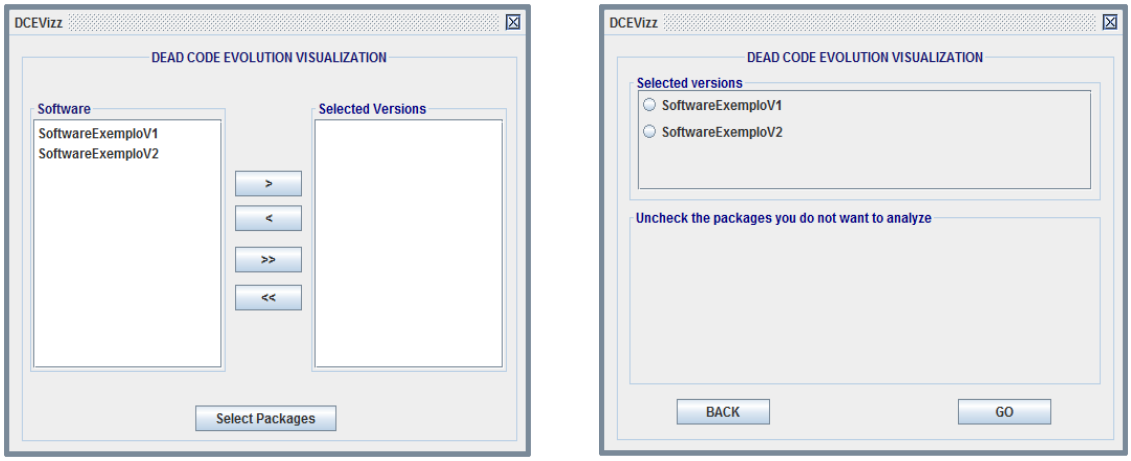
Figura 6.16 - Opções para Iniciar a Execução de DCEVizz Tool



Fonte: Do autor (2017).

As versões do sistema de software a serem analisadas devem estar no *workspace* e ser selecionadas da versão mais antiga para a mais recente, utilizando a interface do *plug-in* (FIGURA 6.17A). Após selecionar as versões, uma tela é apresentada na qual o usuário pode selecionar pacotes que não devem ser analisados (FIGURA 6.17B). Esse recurso permite que módulos em desenvolvimento não sejam considerados na análise, evitando que métodos mortos sejam indevidamente detectados. Além disso, o usuário pode detectar métodos mortos e analisar sua evolução apenas em pacotes de seu interesse.

Figura 6.17 - Telas Iniciais do *plug-in* DCEVizz Tool



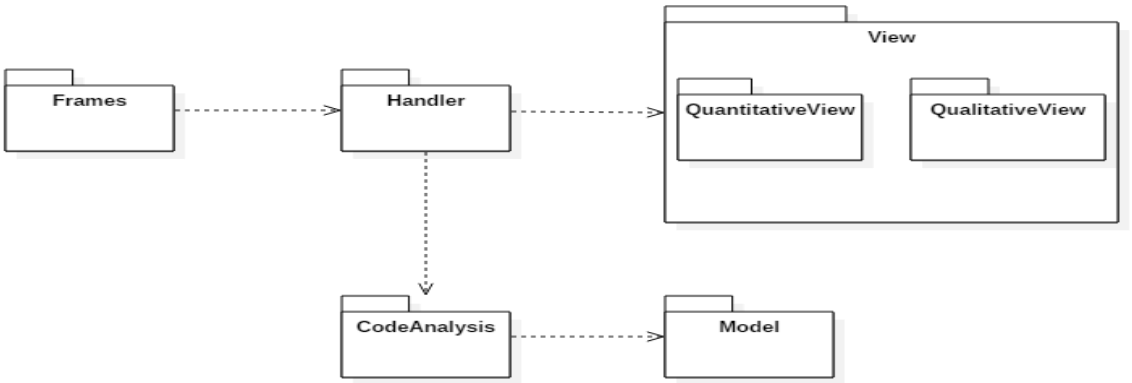
(A) (B)

Fonte: Do autor (2017).

6.2.5 Arquiteura de DCEVizz Tool

Na Figura 6.18, é apresentado o Diagrama de Pacotes que representa a arquitetura de DCEVizz Tool. O pacote `Frames` contém as interfaces de usuário responsáveis por iniciar a execução do *plug-in*. As classes no pacote `Handler` são responsáveis por receber as requisições do usuário e iniciar os componentes que analisam o código e geram as visualizações. As classes no pacote `CodeAnalysis` são responsáveis pela realização das atividades descritas nas etapas **Fonte de Dados** e **Processamento e Mapeamento em Estruturas Visuais**. O pacote `Model` contém as classes que armazenam as informações necessárias para renderizar as visões. O pacote `View` contém as classes responsáveis pela renderização das visualizações quantitativas e qualitativas (etapa **Visualizações**).

Figura 6.18 - Diagrama de Pacotes do *plug-in* DCEVizz Tool



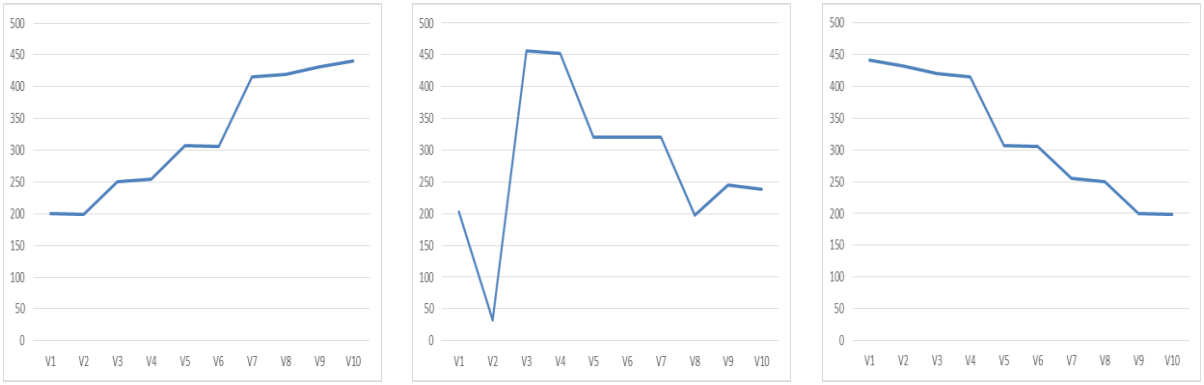
Fonte: Do autor (2017).

6.3 Exemplo de uso de DCEVizz

Nesta seção, são apresentados exemplos de informações que podem ser obtidas a partir das visualizações quantitativa e qualitativa da abordagem DCEVizz, além da utilidade dessas informações aos engenheiros de software. Na Figura 6.19, são apresentadas algumas visualizações quantitativas da abordagem DCEVizz para dez versões de um sistema de software. Como pode ser observado, cada figura representa uma tendência de evolução diferente em relação a quantidade de código morto:

a) **Figura 6.19A**. Existe tendência de aumento contínuo da quantidade de código morto ao longo das versões dos sistemas de software. Essa informação pode alertar os engenheiros de software sobre a necessidade de execução de medidas preventivas para eliminar esse código morto. A eliminação do código morto ao longo das versões diminui a poluição do código, aumenta a legibilidade e facilita a compreensão. Além disso, essa eliminação evita que o código morto seja propagado para as próximas versões, melhorando a qualidade interna e facilitando a manutenção;

Figura 6.19 - Protótipos de Visualização Quantitativas



(A) (B) (C)

Fonte: Do autor (2017)

b) **Figura 6.19B**. Existe oscilação da quantidade de código morto ao longo da evolução. Nesse caso, não é possível identificar algum padrão de evolução que possibilite a previsão do comportamento futuro desse código. Porém, a análise dessa oscilação pode ser útil para compreender outros fatores, por exemplo, quando alguma manutenção significativa foi feita no sistema. Como pode ser observado, houve queda expressiva da quantidade de código morto entre as versões V1 e V2. Exemplos de fatores que podem ocasionar essa queda são a eliminação de código morto ou a finalização de módulos que estavam em desenvolvimento em V1, desencadeando

chamadas para outros métodos e reduzindo a quantidade de métodos inacessíveis. Em contrapartida, o aumento súbito do código morto entre as versões V2 e V3 pode ocorrer por causa da troca de tecnologias ou da substituição de módulos obsoletos, fazendo com que os trechos de código substituídos se tornem mortos;

c) **Figura 6.19C**. Existe tendência de redução contínua da quantidade de código morto ao longo das versões. Essa informação pode indicar que atividades de manutenção estão sendo executadas para eliminar código morto, melhorando cada vez mais a legibilidade das versões. Também, pode indicar que a propagação de código morto entre as versões está sendo evitada pelos desenvolvedores. Com base nessas informações, os engenheiros de software podem verificar se possíveis medidas preventivas adotadas na empresa estão evitando aumento contínuo da quantidade desse código.

Apesar de mostrar o aumento, a redução ou a variação da quantidade de código morto ao longo da evolução, a visualização quantitativa não apresenta especificamente quais pacotes ou classes do sistema de software em que ocorreram essas mudanças. A visualização qualitativa complementa essa visualização oferecendo detalhes do código morto, como a classe e o pacote a que pertence e suas características evolutivas. Na Figura 6.20, é apresentado um exemplo de visualização qualitativa, gerada a partir da análise de três versões do sistema de software. A análise da visualização qualitativa pode ser útil para:

a) **Aumentar a segurança na eliminação de código morto**. Se determinado método é morto em várias versões analisadas, por exemplo, o método `A2()`, então reforça o indicativo de sua inutilidade e de que sua eliminação não fará falta ao sistema de software. Por outro lado, se o uso do método oscila ao longo da evolução, por exemplo, método `A3()`, deve-se ter cautela na sua eliminação, visto que poderá fazer falta nas futuras versões;

b) **Identificar pacotes e classes com tendência de se tornarem em desuso nas versões futuras**. Se a quantidade de métodos mortos aumenta gradativamente em um pacote, pode indicar que algumas de suas funções estão deixando de ser utilizadas ao longo das versões, por exemplo, o pacote `Pacote C`. Essa informação pode ser útil no momento da seleção de módulos do sistema de software para eliminar código morto, dando prioridade para aqueles com maior quantidade e com tendência de se tornarem obsoletos;

c) **Identificar pacotes e classes que tiveram eliminação constante de código morto ao longo da evolução**. Essa situação é ilustrada no pacote `Pacote B`, no qual métodos mortos foram eliminados ou voltaram a ser utilizados nas versões. Essa informação pode indicar que esse pacote tem sido alvo de atividades para combater a propagação de código morto nas versões, contribuindo para sua legibilidade;

d) **Identificar pacotes e classes estáveis em relação a código morto**. Por exemplo o pacote `Pacote D`.

Figura 6.20 - Protótipo de Visualização Qualitativa

Fonte: Do autor (2017)

Os argumentos utilizados no exemplo apresentado são possíveis conclusões que podem ser obtidas a partir da abordagem DCEVizz. A princípio, não faz parte do escopo da abordagem automatizar a análise das visualizações e das possíveis conclusões que podem ser obtidas. O propósito de DCEVizz é facilitar a identificação e a compreensão da evolução de código morto, servindo como alerta sobre possíveis problemas que podem ocorrer no sistema de software por causa da existência desse código. Desse modo, cabe ao engenheiro de software analisar e compreender as técnicas de visualização, a fim de obter suas próprias conclusões a respeito das informações apresentadas e tomar as medidas cabíveis.

6.4 Considerações finais

Neste capítulo, foi apresentada a abordagem DCEVizz, que detecta métodos mortos em versões de sistema de software e os apresenta utilizando técnicas de visualização, enfatizando suas características evolutivas. De modo geral, essa abordagem consiste na execução de quatro etapas: i) Seleção das Versões; ii) Detecção de Código Morto; iii) Análise da Evolução de Código Morto e; iv) Renderização das Visualizações. A detecção de código morto é feita estaticamente utilizando a técnica Análise de Acessibilidade. Além disso, foram definidas duas visualizações, nas quais é exibida a evolução do código morto sob as perspectivas quantitativa e qualitativa.

A abordagem DCEVizz foi implementada em um *plug-in* (DCEVizz Tool) para a plataforma Eclipse IDE concebido com base em uma adaptação do modelo de referência de visualização. Dessa forma, DCEVizz Tool implementa a abordagem proposta em três etapas: i) Fonte de Dados; ii) Processamento e Mapeamento em Estruturas Visuais e; iii) Visualizações.

7 AVALIAÇÃO DA ABORDAGEM DCEVizz

A abordagem DCEVizz foi avaliada em duas etapas. Na primeira etapa, a finalidade foi investigar a capacidade de detecção de código morto da técnica Análise de Acessibilidade utilizada na abordagem. Na segunda etapa, foi realizado um estudo experimental para investigar possíveis benefícios de DCEVizz para identificar e compreender a evolução do código morto. Neste capítulo, são apresentados os procedimentos adotados e os resultados obtidos em cada etapa da avaliação.

O restante do capítulo está organizado da seguinte forma. Detalhes e resultados da avaliação da técnica análise de acessibilidade são descritos na Seção 7.2. O estudo experimental realizado para avaliar a abordagem é apresentado na Seção 7.3. As ameaças à validade dos estudos são apresentadas na Seção 7.4.

7.1 Avaliação da análise de acessibilidade

A finalidade dessa avaliação foi verificar a capacidade de detecção de código morto da técnica Análise de Acessibilidade utilizada em DCEVizz, em relação a outras abordagens que detectam código morto. Neste estudo, o objetivo foi detalhado considerando os princípios da abordagem GQM (*Goal-Question-Metric*), sendo estabelecido da seguinte forma (BASILI et al., 1994):

Analisar	a capacidade de detecção de código morto da técnica Análise de Acessibilidade
Com o propósito de	avaliar
Com respeito ao	código morto identificado pelas ferramentas utilizadas na avaliação
Do ponto de vista de	engenheiros de software
No contexto de	manutenção de software

De modo geral, essa verificação foi realizada por meio da comparação dos resultados das ferramentas que automatizam a execução das abordagens. Para tanto, foi formulada a seguinte questão de investigação:

A implementação da técnica Análise de Acessibilidade utilizada em DCEVizz Tool é capaz de identificar métodos inacessíveis tanto quanto outra ferramenta de detecção de código morto?

Cinco sistemas de software desenvolvidos em Java foram analisados pelo *plug-in* DCEVizz Tool (que automatiza a abordagem DCEVizz) e pela ferramenta Understand[4].

7.1.1 Caractetização dos sistemas de software analisados

A avaliação foi conduzida utilizando a versão mais recente de cinco sistemas de software (TABELA 7.1), selecionados por: i) serem habitualmente utilizados em pesquisas sobre qualidade de software; ii) serem de código aberto; iii) possuírem pelo menos cinco versões disponíveis na internet; e iv) serem desenvolvidos em Java.

Tabela 7.1 - Características das Versões dos Sistemas de Software Analisados

Software	Versão	Descrição	Quantidade de Pacotes	Quantidade de Classes	Quantidade de Métodos
ArgoUML[5]	0.34	Ferramenta de modelagem para projeto, desenvolvimento e documentação de aplicações orientadas a objetos, utilizando a notação UML.	238	2.895	19.633
FreeMind[6]	1.0.1	Ferramenta para criação de mapas mentais, que permitem representar uma ideia ou um conjunto de ideias de forma visual.	88	1.015	7.676
JabRef[7]	3.2	Ferramenta para gerenciamento de referências bibliográficas utilizando o padrão BibTeX.	160	1.284	6.904
JEdit[8]	5.3.0	Ferramenta de edição de textos voltada para programadores, com uma arquitetura de plug-ins extensível.	66	1.082	8.119
LatexDraw[9]	3.3.3	Ferramenta de edição de desenhos gráficos para LaTeX.	81	906	7.199

Fonte: Do autor (2017).

7.1.2 Execução da avaliação

Existe carência de ferramentas funcionais fundamentadas em abordagens para detecção de métodos inacessíveis em sistemas de software Java. Algumas das ferramentas existentes possuem limitações que as impediram de serem utilizadas nessa avaliação. Na Tabela 7.2, são apresentados o nome e o endereço eletrônico dessas ferramentas, além da

[4] https://scitools.com
[5] www.argouml.org
[6] www.freemind.sourceforge.net
[7] www.jabref.org
[8] www.jedit.org
[9] www.latexdraw.sourgeforge.net

razão pela qual não foram utilizadas. Dentre essas ferramentas, a única que possui uma abordagem embasada na literatura é DUM-Tool (ROMANO et al., 2016). No entanto, essa ferramenta não foi utilizada por apresentar problemas de execução durante a análise de alguns sistemas de software.

Tabela 7.2 - Ferramentas Não Utilizadas na Avaliação

Ferramenta	Motivo por não ser utilizada
CodePro Analytics[10]	Apresentou incoerência significativa dos resultados obtidos em um projeto piloto.
DUM-Tool[11]	Não funcional. Necessita de conhecimento prévio do software[12].
JTombstone[13]	Necessita de conhecimento prévio do software.
PMD[14]	Analisa apenas métodos privados.
UCDetector[15]	Analisa apenas métodos públicos.

Fonte: Do autor (2017).

Apesar de não estar relacionada a uma abordagem, a ferramenta Understand foi utilizada nessa avaliação por ser funcional e consolidada no mercado. Essa ferramenta foi desenvolvida por SciTools (Scientific Toolworks Inc.), entidade que desenvolve sistemas para manutenção de software desde 1996. Essa ferramenta realiza análise estática do código de sistemas de software desenvolvidos em diversas linguagens de programação, permitindo a análise, a medição e a compreensão desses sistemas. Uma de suas funções é a detecção de código morto em sistemas de software Java. Assim como em DCEVizz Tool, essa ferramenta analisa os métodos do sistema de software independente de sua visibilidade. A avaliação foi executada analisando os sistemas de software apresentados na Tabela 7.1, utilizando a ferramenta Understand e o *plug-in* DCEVizz Tool.

7.1.3 Resultados e discussões

Na Tabela 7.3, é apresentada a quantidade de métodos mortos identificada por DCEVizz Tool e por Understand, assim como o percentual dessa quantidade em relação à quantidade total de métodos do sistema de software. Como pode ser observado, DCEVizz Tool foi capaz de identificar maior quantidade de métodos inacessíveis do que Understand nos sistemas de software analisados. Examinando essa quantidade em relação aos métodos

[10] https://developers.google.com/java-dev-tools/codepro
[11] http://www2.unibas.it/sromano/DUM-Tool.html
[12] O uso de ferramentas com essa característica depende de conhecimento prévio do sistema de software para indicar os métodos de inicialização (*main*). Considerando que no *plug-in* DCEVizz Tool não é necessário esse conhecimento prévio e que os pesquisadores não possuem domínio dos sistemas de software a serem analisados, essa ferramenta foi desconsiderada do estudo.
[13] https://sourceforge.net/projects/jtombstone/
[14] https://sourceforge.net/projects/pmd/
[15] http://www.ucdetector.org/

dos sistemas de software, em média, aproximadamente 20% dos métodos dos sistemas de software analisados são inacessíveis. Em contrapartida, Understand identificou, em média, aproximadamente 14% de métodos inacessíveis em relação aos métodos desses sistemas.

Tabela 7.3 - Quantidade de Métodos Mortos Identificada pelas Ferramentas

Software	Quantidade de Métodos Detectados com DCEVizz Tool	%	Quantidade de Métodos Detectados com Understand	%
ArgoUML	3.201	**16%**	1.860	**9%**
FreeMind	703	**9%**	495	**6%**
JabRef	1.432	**21%**	1.151	**17%**
JEdit	1.583	**19%**	987	**12%**
LatexDraw	2.615	**36%**	2.042	**28%**

Fonte: Do autor (2017).

Os resultados obtidos foram analisados de forma mais precisa considerando três casos. Seja $\mathbf{A_s}$ o conjunto de métodos identificados por DCEVizz Tool e $\mathbf{U_s}$ o conjunto de métodos identificados por Understand para um sistema de software `S`, os casos considerados neste estudo podem ser dados por:

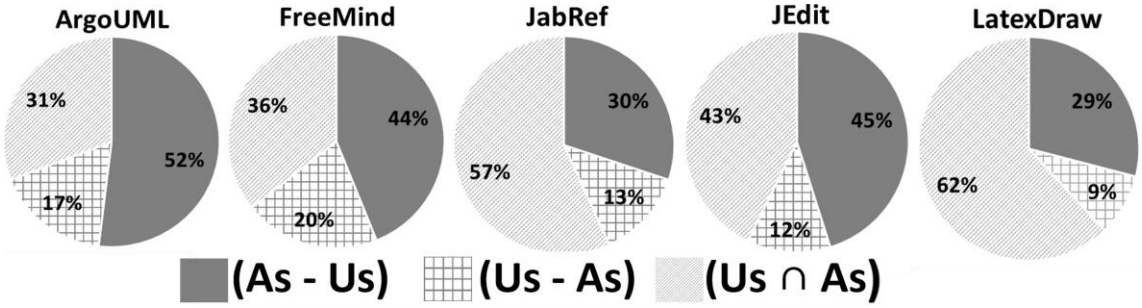
a) **Caso 1 ($\mathbf{U_s} \cap \mathbf{A_s}$)**. Métodos inacessíveis identificados por Understand e por DCEVizz Tool;
b) **Caso 2 ($\mathbf{U_s} - \mathbf{A_s}$)**. Métodos inacessíveis identificados por Understand e não identificados por DCEVizz Tool;
c) **Caso 3 ($\mathbf{A_s} - \mathbf{U_s}$)**. Métodos inacessíveis identificados por DCEVizz Tool e não identificados por Understand.

Na Figura 7.1, é apresentada proporção de métodos mortos pertencente a cada um dos três casos, considerando o total $\mathbf{T_s}$ de métodos mortos ($\mathbf{T_s} = \mathbf{A_s} \cup \mathbf{U_s}$). Após identificar os métodos mortos, ambas as ferramentas geram um arquivo de *log* com os resultados obtidos. Foi desenvolvido um *script* para comparar esses arquivos e identificar quais métodos mortos foram identificados pelas duas ferramentas, métodos mortos identificados apenas por DCEVizz Tool e métodos mortos identificados apenas por Understand.

Considerando, respectivamente, os sistemas de software ArgoUML, FreeMind, JabRef, JEdit e LatexDraw, essa operação mostrou que aproximadamente 64% (1.194 de 1.860 métodos), 64% (315 de 495 métodos), 81% (935 de 1.151 métodos), 78% (770 de 987 métodos) e 88% (1.787 de 2042 métodos) dos métodos inacessíveis identificados por Understand também foram identificados por DCEVizz Tool (Caso 1). Na Figura 7.1, esse percentual pode ser visualizado em relação à quantidade total $\mathbf{T_s}$. De modo geral, esses percentuais indicam que, em média, 75% dos métodos inacessíveis identificados por

Understand foram identificados por DCEVizz Tool. Em relação a **T_s**, aproximadamente 46% dos métodos inacessíveis foram identificados por Understand e por DCEVizz Tool.

Figura 7.1 - Percentual de Métodos Inacessíveis Identificados pelas Ferramentas



Fonte: Do autor (2017).

Os resultados do *script* foram analisados manualmente para identificar o tipo de código morto pertencente ao Caso 2 e ao Caso 3, a fim de identificar características adotadas nas ferramentas que podem ter causado divergências entre seus resultados. Na Tabela 7.4, é apresentada a quantidade de código morto pertencente ao Caso 2 em relação ao seu tipo. Como pode ser observado, construtores, métodos abstratos e métodos relacionados à interface gráfica (*listeners*) correspondem a maioria do código morto não identificado por DCEVizz Tool. Conforme discutido no Algoritmo 2 da Seção 6.3.2.2., código com essas características foram intencionalmente desconsiderados do processo de detecção. O propósito de DCEVizz tool é analisar apenas a acessibilidade de métodos; por esse motivo, tipos enumerados foram identificados apenas por Understand.

Tabela 7.4 - Tipo dos Métodos Mortos Pertencentes ao Caso 2 (Us - As)

Tipo	ArgoUML	FreeMind	JabRef	JEdit	LatexDraw
Construtores	347 (52,13%)	68 (37,77%)	45 (20,84%)	110 (50,69%)	95 (37,25%)
Métodos em Blocos Internos	65 (9,75%)	22 (12,23%)	60 (27,78%)	42 (19,35%)	-
Tipo Enumerado	10 (1,5%)	-	20 (9,25%)	7 (3,22%)	58 (22,75%)
Métodos Com Chamadores	4 (0,6%)	9 (5%)	44 (20,37%)	-	-
Métodos Abstratos	180 (27,02%)	31 (17,23%)	5 (2,32%)	26 (11,99%)	102 (40%)
Métodos de Interface Gráfica	51 (7,65%)	16 (8,89%)	37 (17,12%)	2 (0,93%)	-
Métodos em *Inner Classes*	9 (1,35%)	33 (18,33%)	2 (0,93%)	25 (11,52%)	-
Métodos Inacessíveis	-	1 (0,55%)	3 (1,39%)	5 (2,3%)	-
Total de Métodos do Caso 2	**666**	**180**	**216**	**217**	**255**

Fonte: Do autor (2017).

Além disso, pode ser observado que, em média, aproximadamente 5% dos métodos identificados apenas por Understand possuíam chamadores acessíveis (não eram mortos) e aproximadamente 1% desses métodos eram realmente mortos. Esses percentuais podem indicar uma limitação na detecção de Understand e de DCEVizz Tool. Métodos declarados em blocos internos e em classes internas (*inner classes*) não foram detectados pelo *plug-in* DCEVizz Tool. Esse fato não corresponde a uma restrição ou a uma limitação da abordagem, mas está relacionado a algum problema técnico que ocorreu na execução da ferramenta.

Na Tabela 7.5, são apresentados os tipos e a quantidade de métodos pertencentes ao Caso 3. Em média, aproximadamente 30% dos métodos são chamados apenas por métodos inacessíveis. Cerca de 3% dos métodos são chamados por métodos inacessíveis e sobrescrevem métodos de superclasses. Além disso, 7% dos métodos foram considerados inacessíveis por não possuírem chamadores e, aproximadamente, 59% dos métodos são inacessíveis e sobrescrevem métodos de superclasses. Alguns métodos (~1%) possuem algum método acessível em sua hierarquia de chamadas, podendo ser considerados vivos.

Tabela 7.5 - Tipos dos Métodos Pertencentes ao Caso 3 (As - Us)

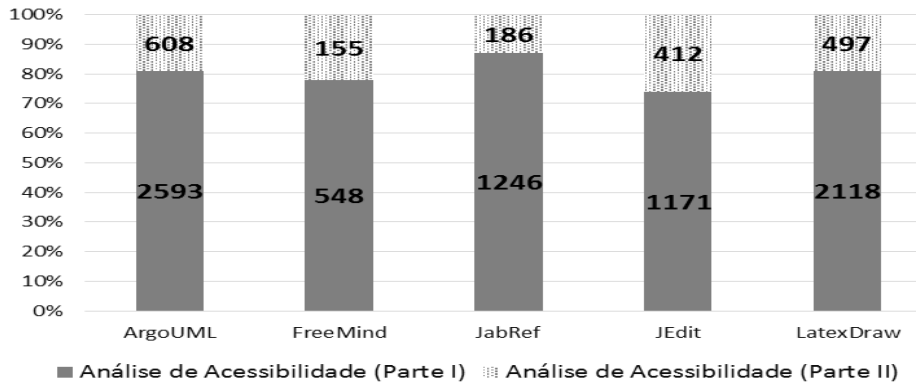
Tipo	ArgoUML	FreeMind	JabRef	JEdit	LatexDraw
Chamado por Métodos Inacessíveis	422 (21,03%)	116 (29,89%)	194 (39,04%)	366 (45,02%)	129 (15,57%)
Chamado por Métodos Inacessíveis (*Override*)	21 (1,04%)	23 (5,93%)	-	-	61 (7,37%)
Métodos “Vivos”	8 (0,39%)	23 (5,93%)	5 (1%)	-	-
Métodos sem Chamador	280 (13,96%)	16 (4,12%)	40 (8,04%)	16 (1,96%)	59 (7,13%)
Métodos sem Chamador (*Override*)	1.276 (63,58%)	210 (54,13%)	258 (51,92%)	431 (53,02%)	579 (69,93%)
Total de Métodos do Caso 3	**2.007**	**388**	**497**	**813**	**828**

Fonte: Do autor (2017).

Apesar da maioria dos métodos identificados por Understand (~75%) também ter sido identificado por DCEVizz Tool, algumas características adotadas na implementação ocasionaram divergências nos resultados obtidos. Ao contrário de Understand, DCEVizz Tool analisou os métodos dos sistemas de software sem considerar questões de herança (*e.g.* sobrescrita de métodos), fazendo com que métodos chamados via polimorfismo possam ser detectados como inacessíveis. Em contrapartida, Understand analisou construtores, métodos abstratos e métodos de interface gráfica. A análise de métodos relacionados à interface gráfica é uma limitação de abordagens que utilizam análise estática, visto que a captura de chamadas desses métodos depende da execução do sistema de software.

No que se refere a utilização da técnica Análise de Acessibilidade em duas fases, a Análise de Acessibilidade - Fase II aumentou a capacidade de detecção de métodos inacessíveis de DCEVizz Tool. Na Figura 7.2, são apresentados a quantidade de métodos mortos identificados em cada fase e o percentual (eixo `y`) em relação ao total de métodos identificados. Essa fase foi responsável por identificar, em média, aproximadamente, 20% do total de métodos inacessíveis para os cinco sistemas de software analisados.

Figura 7.2 - Percentual e Quantidade de Métodos Inacessíveis (Fase I e Fase II)



Fonte: Do autor (2017).

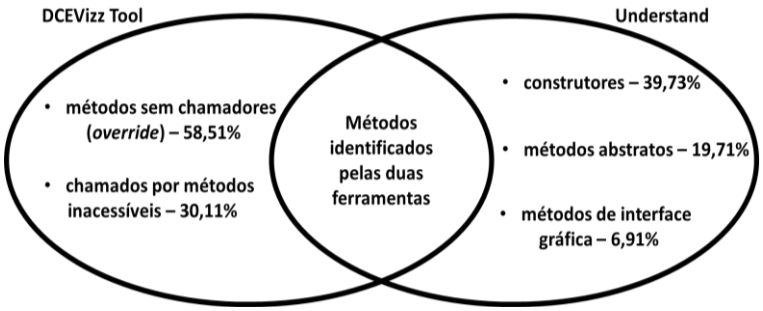
Além disso, considerando, respectivamente, os sistemas de software ArgoUML, FreeMind, JabRef, JEdit e LatexDraw, aproximadamente 34% (151 métodos de 443), 8% (11 métodos de 139), 0% (0 métodos de 194), 13% (48 métodos de 366) e 19% (94 métodos de 267) dos métodos identificados na Análise de Acessibilidade - Fase II também foram identificados por Understand. Com base nos resultados descritos, obteve-se a seguinte conclusão para a questão de investigação:

`A implementação da técnica Análise de Acessibilidade utilizada em DCEVizz Tool é capaz de identificar métodos inacessíveis tanto quanto outra ferramenta de detecção de código morto?`

Parcialmente. Apesar de parte significativa dos métodos mortos identificados por Undertsand também terem sido identificados por DCEVizz Tool, a resposta exata a essa pergunta depende de particularidades adotadas na implementação das ferramentas. Na Figura 7.3, é apresentada uma síntese das principais particularidades identificadas, assim como a porcentagem média em relação as demais, considerando os cinco sistemas de software analisados. Por exemplo, a maior parte dos métodos não identificados por DCEVizz Tool correspondem às restrições intencionalmente definidas na linha 5 do Algoritmo 2 (Seção 6.3.2.2), que exclui da análise de construtores, métodos abstratos e métodos de interface gráfica. Além disso, ao contrário de Understand, DCEVizz Tool analisou a acessibilidade desconsiderando sobrescritas de

métodos de superclasses, resultando em maior quantidade de métodos inacessíveis identificados.

Figura 7.3 - Particularidades Adotadas nas Ferramentas



Fonte: Do autor (2017).

7.1.4 Análise evolutiva do código morto

Tendo ciência das particularidades adotadas em cada ferramenta, foi realizada uma análise da evolução da quantidade de código morto identificada por DCEVizz Tool e por Understand para verificar se essas particularidades causam divergências nos resultados obtidos ao longo das versões. Para tanto, cinco versões mais recentes dos sistemas de software (Seção 7.2.1) foram analisadas pelas duas ferramentas. As versões desses sistemas foram analisadas de forma separada por Understand, visto que essa ferramenta não possibilita a análise simultânea das versões, como realizado por DCEVizz Tool.

Na Tabela 7.6, é apresentada a quantidade de pacotes, de classes e de métodos das versões dos sistemas de software analisados, a quantidade de métodos mortos identificados pelas ferramentas e o percentual de aumento/diminuição dessa quantidade em relação à versão anterior. Como pode ser observado, DCEVizz Tool identificou maior quantidade de métodos em relação Understand em todas as versões dos sistemas de software analisadas.

Com os dados apresentados, pode ser verificada a relação existente entre a evolução do sistema de software e o aumento da quantidade de métodos. Dentre os 25 sistemas de software analisados (5 versões de 5 sistemas de software), houve diminuição dos métodos em relação a versão anterior em apenas 3 sistemas de software (12%), evidenciando que a quantidade de informações tende a aumentar a medida que os sistemas de software evoluem.

Nos sistemas de software ArgoUML (versão 0.34) e JabRef (versão 2.1.1), houve diminuição da quantidade total de métodos e de classes em relação à versão anterior. Atividades de manutenção podem ter sido realizadas nessas versões, resultando na exclusão

de algumas de suas classes e, consequentemente, na eliminação de métodos mortos. Por outro lado, apesar da quantidade de classes ter aumentado no sistema de software JabRef (versão 3.0.0) em relação à versão anterior, houve pequena diminuição da quantidade total de métodos. Esse fato pode ter ocorrido por causa do versionamento (de 2.1.1 para 3.0.0), indicando que mudanças significativas podem ter ocorrido no código, inclusive eliminação de métodos mortos.

Tabela 7.6 - Resultados obtidos com a Análise Evolutiva

					(DCEVizz Tool)		(Understand)	
	Versão	Quantidade de Pacotes	Quantidade de Classes	Quantidade de Métodos	Quantidade de Métodos Mortos	% aumento/ redução	Quantidade de Métodos Mortos	% aumento/ redução
ArgoUML	0.26	178	2.425	17.819	2.867	-	1.752	-
	0.28	222	2.769	18.860	3.052	↑ 6,45%	1.994	↑ 13,81%
	0.30	230	3.082	19.932	3.190	↑ 4,52%	1.798	↓ 9,83%
	0.32	233	3.123	20.318	3.086	↓ 3,26%	1.780	↓ 1,00%
	0.34	238	2.895	19.633	3.182	↑ 3,11%	1.860	↑ 4,49%
FreeMind	0.8.0	62	679	6.103	849	-	537	-
	0.8.1	64	700	6.223	850	↑ 0,12%	545	↑ 1,49%
	0.9.0	77	828	6.298	534	↓ 37,18%	387	↓ 28,99%
	1.0.0	87	1.009	7.669	669	↑ 25,41%	494	↑ 27,65%
	1.0.1	88	1.015	7.676	670	↑ 2,77%	495	↑ 0,2%
JabReF	2.1.0	133	1.189	6.958	1.101	-	733	-
	2.1.1	115	1.086	6.438	898	↓ 18,44%	559	↓ 23,74%
	3.0.0	159	1.209	6.396	1.094	↑ 21,83%	797	↑ 42,58%
	3.1.0	163	1.277	6.859	1.372	↑ 25,41%	1.098	↑ 37,77%
	3.2.0	160	1.284	6.902	1.410	↑ 2,77%	1.153	↑ 5,01%
JEdit	4.52	57	1.028	7.551	1.344	-	813	-
	5.00	64	1.061	7.829	1.373	↑ 2,16%	835	↑ 2,71%
	5.10	67	1.074	7.981	1.501	↑ 9,32%	961	↑ 15,09%
	5.20	66	1.078	8.077	1.561	↑ 4,00%	984	↑ 2,39%
	5.30	66	1.082	8.119	1.563	↑ 0,13%	987	↑ 0,3%
LatexDraw	3.2.0	81	882	6.960	2.498	-	1.991	-
	3.3.0	81	895	7.102	2.567	↑ 2,76%	2.004	↑ 0,65%
	3.3.1	81	902	7.169	2.608	↑ 1,59%	2.037	↑ 1,64%
	3.3.2	81	900	7.193	2.610	↑ 0,07%	2.038	↑ 0,04%
	3.3.3	81	906	7.199	2.615	↑ 1,91%	2.042	↑ 0,19%

Fonte: Do autor (2017).

Além disso, os resultados de DCEVizz Tool mostraram que a maioria dos sistemas de software apresentou aumento da quantidade de métodos mortos quando a quantidade total de métodos aumentou em relação à versão anterior. Essa situação não aconteceu apenas nos sistemas de software ArgoUML (versão 0.32) e FreeMind (versão 0.9.0), em que a quantidade total de métodos aumentou, enquanto a quatidade de métodos mortos diminuiu. Houve divergências também nos sistemas de software ArgoUML (versão 0.34) e JabRef (versão 3.0.0), em que a quantidade total de métodos diminuiu e a quantitade de métodos mortos aumentou. Em JabRef (versão 2.1.1), a quantidade total de métodos e de métodos mortos

diminuiu em relação a versão anterior. De modo geral, essas divergências podem ter ocorrido por causa das atividades de manutenção realizadas entre as versões, que podem impactar no aumento/redução da quantidade total de métodos, assim como na quantidade de métodos mortos.

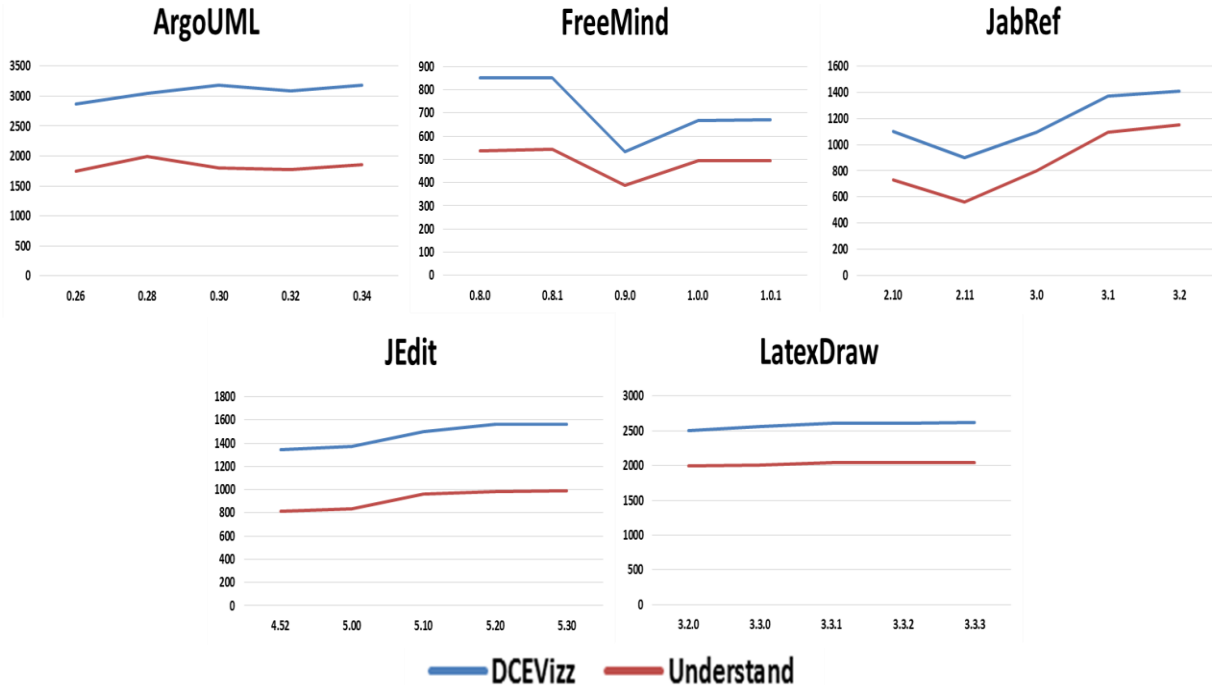
Ademais, pode-se observar que, apesar de Understand ter identificado menor quantidade de código morto, ambas as ferramentas identificaram tendência de evolução semelhante ao longo das versões, mostrando coerência entre os resultados obtidos pelas ferramentas. Mais especificamente, em 96% das versões analisadas, ambas ferramentas identificaram o mesmo padrão de aumento ou de diminuição da quantidade de código morto em relação a versão anterior. Esse padrão não ocorreu apenas no sistema de software ArgoUML (versão 0.30), em que DCEVizz identificou aumento de código morto em relação a versão 0.28, enquanto Understand identificou diminuição. A divergência na versão 0.30 de ArgoUML pode ter ocorrido por causa das especificidades adotadas na implementação das ferramentas e das caraterísticas dos seus métodos. Por exemplo, se a quantidade de métodos que sobrescrevem métodos aumentou nessa versão, então a quantidade de métodos identificados por DCEVizz Tool tende a aumentar, visto que o *plug-in* não faz tratamento desse tipo de método. Em contrapartida, possíveis tratamentos de Understand na análise de métodos desse tipo podem fazer com essa ferramenta identifique menor quantidade de métodos em relação à versão 0.28, causando divergências nos resultados das ferramentas.

Na Figura 7.4, é apresentada graficamente a quantidade de código morto identificada pelas ferramentas nas versões analisadas, em que o eixo `y` corresponde à quantidade de código morto e o eixo `x` corresponde às versões do sistema de software. Como pode ser observado, houve queda significativa da quantidade de código morto entre duas versões consecutivas dos sistemas de software FreeMind e JabRef. Esse fato pode ocorrer por causa da: i) execução de atividades de manutenção para eliminação de código morto; ii) módulos de software em desuso que voltaram a ser utilizados, reativando a chamada de alguns métodos; ou iii) finalização da implementação de algumas funções, fazendo com que métodos em desenvolvimento sejam chamados.

Além disso, pode-se perceber que houve tendência de aumento contínuo da quantidade de código morto ao longo das três versões mais recentes dos sistemas de software FreeMind e JabRef. No caso dos sistemas de software JEdit e LatexDraw, esse aumento foi contínuo em todas as versões analisadas. Essas informações podem ser úteis para alertar os mantenedores

de software a respeito da qualidade desses sistemas, indicando que as próximas versões possam surgir com maior quantidade de código morto.

Figura 7.4 - Quantidade de Código Morto Identificado pelas ferramentas



Fonte: Do autor (2017).

7.2 Avaliação geral da abordagem DCEVizz

De modo geral, a adoção de técnicas e de ferramentas de visualização de software tem como principal objetivo facilitar a percepção humana em relação a determinada característica de sistemas de software. A avaliação dessas técnicas e ferramentas depende da execução de alguns estudos empíricos, em particular, experimentos controlados envolvendo seres humanos (principais beneficiados com a área de visualização) (LUCCA; PENTA, 2006). Desse modo, a abordagem DCEVizz foi avaliada por meio de um estudo experimental, cuja finalidade foi investigar se as visualizações proveem melhor percepção da existência de código morto e de suas características evolutivas.

O estudo experimental consistiu na execução de um conjunto de tarefas por um grupo de voluntários, que responderam questões **não utilizando** e **utilizando** DCEVizz Tool. As respostas das questões foram analisadas quantitativamente utilizando as variáveis **precisão** e **eficiência**. Além disso, foi realizada uma análise qualitativa, em que foram consideradas as opiniões dos voluntários a respeito do *plug-in*. Os princípios da abordagem GQM foram utilizados para definir o objetivo do estudo:

Analisar	a abordagem DCEVizz
Com o propósito de	avaliar
Com respeito a	precisão, eficiência e facilidade, em comparação à execução das mesmas atividades sem o uso do apoio computacional da abordagem
Do ponto de vista de	engenheiros de software
No contexto de	manutenção de software

As variáveis precisão e eficiência são usualmente utilizadas em experimentos para avaliar abordagens relacionadas à visualização de software (CORNELISSEN et al., 2011; SCHOTS, 2011; NOVAIS et al., 2012; FRANCESE et al., 2014). De modo geral, essas variáveis permitem verificar se houve melhoria na exatidão e no tempo necessário para executar determinadas tarefas quando são utilizadas técnicas de visualização. Nesse estudo, a variável precisão foi utilizada para investigar se DCEVizz Tool auxilia a identificação e a compreensão da evolução do código morto, observando a corretude das respostas dadas pelos voluntários quando utilizam e não utilizam o *plug-in*. Essa variável é dada por:

$$\text{precisão} = \frac{\text{quantidade de respostas corretas dadas}}{\text{quantidade total de questões}}$$

A variável eficiência foi utilizada para observar se DCEVizz Tool diminui o tempo de execução das tarefas para identificar e compreender a evolução de código morto. Essa variável foi definida como:

$$\text{eficiência} = \frac{\text{quantidade de respostas dadas}}{\text{tempo utilizado}}$$

Os valores obtidos para cada variável foram analisados estatisticamente utilizando teste de hipóteses. Nas subseções seguintes, são descritos detalhes adotados no planejamento e na condução desse estudo.

7.2.1 Planejamento

Com base no estudo experimental executado em (SCHOTS, 2011), foram elaborados cinco formulários e um documento para contextualização (APÊNDICE C) e coleta das informações necessárias para avaliação de DCEVizz:

a) **Termo de Consentimento**. Informa aos participantes a finalidade do estudo, descrevendo detalhes adotados na avaliação. Além disso, obtém a autorização do

participante para uso e publicação de seus resultados em trabalhos científicos (APÊNDICE C1);

b) **Formulário de Caracterização do Participante**. São solicitadas informações a respeito da formação acadêmica e da experiência dos participantes em relação aos assuntos abordados na avaliação (APÊNDICE C2);
c) **Contexto**. Documento definido para imersão dos participantes no contexto da avaliação (APÊNDICE C3);
d) **Etapa 1. Sem o Uso de DCEVizz**. Formulário composto por 10 questões que abordam detecção de código morto e compreensão da sua evolução (APÊNDICE C4);
e) **Etapa 2. Com o Uso de DCEVizz**. Formulário composto por 13 questões, sendo que 10 questões são iguais as questões do formulário anterior. Foram definidas 3 questões adicionais que abordam uma análise geral do código morto, sendo inviável de serem respondidas sem o apoio de DCEVizz Tool (na Etapa 1) (APÊNDICE C5);
f) **Formulário de Opinião**. Foram solicitadas opiniões e sugestões de melhoria aos participantes a respeito de DCEVizz Tool (APÊNDICE C6).

As questões dos formulários da Etapa 1 e da Etapa 2 foram estruturadas em dois grupos: i) as questões de 1 a 5 abordam a identificação de código morto; e ii) as questões de 6 a 10 abordam aspectos evolutivos do código morto. De modo geral, essa estrutura permite investigar os efeitos da abordagem DCEVizz na identificação e na compreensão da evolução do código morto, considerando os valores das variáveis precisão e eficiência. Para cálculo da eficiência, foram solicitados os horários de início e de término em cada grupo de questões. Esse tempo não foi coletado para cada questão para evitar unidade de tempo inadequada, visto que existem questões que podem ser respondidas em menos de um minuto, comprometendo a exatidão dos resultados se utilizar segundos como medida. Além disso, o tempo utilizado por DCEVizz Tool para identificar código morto e gerar as visualizações não foi contabilizado no cálculo da eficiência na Etapa 2.

Ao contrário das demais questões, as questões adicionais (questões de 11 a 13), definidas apenas no formulário da Etapa 2, necessitam de análise geral e da identificação do código morto presente no sistema de software. Responder essas questões sem o uso de DCEVizz Tool aumentaria significativamente o esforço de execução do estudo. Nesse caso, os participantes teriam que analisar separadamente todos os pacotes, todas as classes e todos os métodos das versões do sistema de software em estudo, identificar os métodos mortos e fazer análise da sua evolução. Dessa forma, as questões foram respondidas apenas com o uso

de DCEVizz Tool, a fim de investigar se a abordagem possibilita uma visão geral do código morto do sistema de software, permitindo compreensão da sua evolução.

Além disso, foi solicitado ao participante descrever a dificuldade em executar as atividades definidas nos formulários. Para tanto, uma escala de Likert com cinco opções foi fornecida em cada questão: ( ) Muito fácil; ( ) Fácil; ( ) Mais ou menos fácil; ( ) Difícil; e ( ) Muito difícil. A finalidade dessa escala foi verificar **implicitamente** se, na opinião dos participantes, a abordagem DCEVizz aumentou, diminuiu ou não alterou a dificuldade em realizar as atividades. Em contrapartida, nas três questões adicionais definidas no formulário da Etapa 2, o objetivo da escala de Likert foi coletar a opinião do usuário sobre a dificuldade em obter a resposta caso a abordagem DCEVizz não fosse utilizada.

O gabarito dos formulários foi obtido utilizando o recurso *Call Hierarchy* da plataforma Eclipse IDE, que permite visualizar a hierarquia de chamadas de um método e verificar se ele é inacessível (morto). Dois pesquisadores que não possuem interesse direto nos resultados do estudo responderam os questionários para calibragem do experimento, ajustes das perguntas e revisão do gabarito. Os pesquisadores possuem experiência com desenvolvimento de sistemas de software Java na plataforma Eclipse IDE e foram selecionados por conveniência e disponibilidade de tempo.

Além dos formulários citados, foi definido um documento para imersão dos participantes em um cenário fictício, no qual eles deveriam assumir papel de engenheiros de software de uma empresa. Sua responsabilidade era preservar a legibilidade e a manutenibilidade das versões dos sistemas de software dessa empresa, por meio da identificação e da compreensão da evolução de código morto. Antes do início do estudo, foi ministrado um minicurso para contextualização dos participantes com assuntos correlatos a avaliação. Foi apresentado o tema Visualização de Software, conceitos relacionados a código morto, a abordagem DCEVizz, ao *plug-in* DCEVizz Tool, a plataforma Eclipse IDE e a sua funcionalidade *Call Hierarchy*.

A avaliação foi conduzida da seguinte forma. Antes de iniciar o estudo, os voluntários deveriam assinar o termo de consentimento e preencher o formulário de caracterização do participante. Em seguida, foi iniciada a primeira etapa da avaliação, no qual os participantes responderam o formulário da Etapa 1 sem o uso do *plug-in* DCEVizz Tool. Para tanto, o código de duas versões de um sistema de software foi analisado para identificar métodos mortos e compreender sua evolução ao longo das duas versões. Por causa da dificuldade em identificar código morto manualmente, os participantes poderiam responder esse formulário utilizando o recurso *Call Hierarchy* da plataforma Eclipse IDE.

Posteriormente, foi entregue aos mesmos participantes o formulário da Etapa 2, que deveria ser respondido com o uso de DCEVizz Tool. Para evitar que as respostas do formulário anterior fossem memorizadas e tivessem influência na segunda etapa, foram utilizadas duas versões de um sistema de software diferente, necessitando uma nova análise e compreensão por parte dos participantes. Vale ressaltar que as respostas do formulário dessa etapa deveriam ser obtidas **apenas** com a análise e a interação com as visualizações quantitativa e qualitativa geradas por DCEVizz Tool. Para evitar compartilhamento de respostas, foi solicitado que não houvesse comunicação entre os participantes durante a realização da avaliação.

Os sistemas de software utilizados na Etapa 1 e na Etapa 2 são, respectivamente, DubMan e PlayLister, selecionados por estarem disponíveis gratuitamente no repositório SourceForge[16]. A segunda versão desses sistemas utilizada em ambas as etapas é fictícia, sendo manualmente manipulada para induzir a existência das características evolutivas de código morto definidas em DCEVizz. Na Tabela 7.7, é apresentada a quantidade de pacotes, de classes e de métodos das versões dos sistemas de software. Como pode ser observado, houve variação dessa quantidade entre as versões dos dois sistemas, principalmente em relação às classes e aos métodos. Acredita-se que essa variação não favorece as etapas, visto que as questões dos formulários são pontuais e enfatizam métodos e pacotes específicos, fazendo com que a quantidade de métodos ou de classes seja irrelevante no esforço das atividades.

Tabela 7.7 - Caracterização dos Sistemas de Software Utilizados

Sistemas de Software		Quantidade de Pacotes	Quantidade de Classes	Quantidade de Métodos
DubMan	Versão 1	5	28	356
	Versão 2	5	40	549
PlayLister	Versão 1	5	43	506
	Versão 2	4	46	526

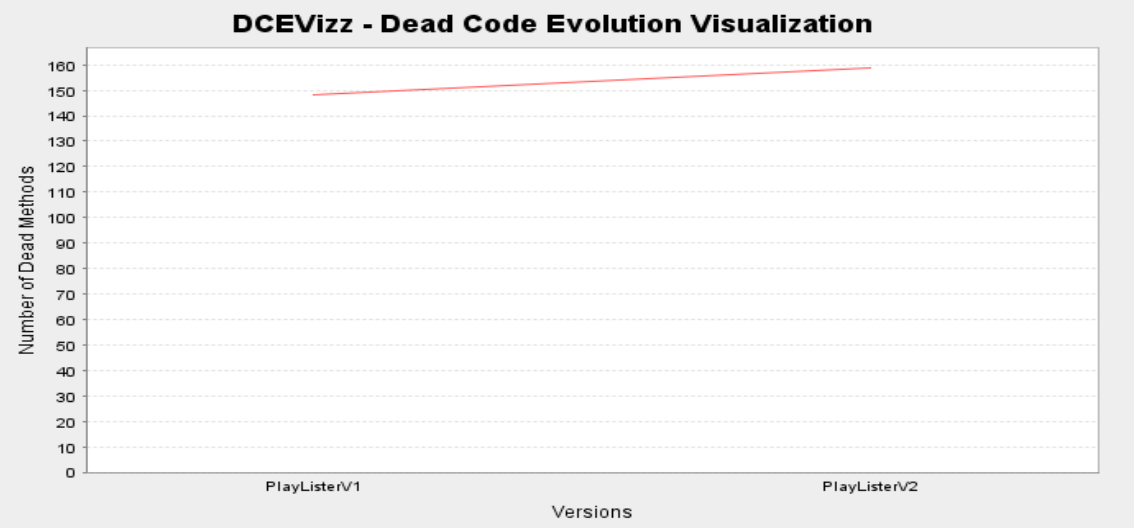
Fonte: Do autor (2017).

Por fim, na última etapa dessa avaliação, os participantes responderam o formulário de opinião, relatando seu ponto de vista a respeito de DCEVizz Tool, além de sugestões de melhoria. Após a execução do estudo, as respostas foram corrigidas utilizando o gabarito dos formulários. Para ser considerado um acerto, a resposta e a justificativa deveriam estar corretas. Desse modo, é possível verificar se o participante realmente compreendeu a tarefa, evitando que respostas dadas ao acaso sejam consideradas no estudo. Com base nos resultados obtidos, foi calculado os valores para as variáveis precisão e eficiência. As visualizações

[16] https://sourceforge.net/

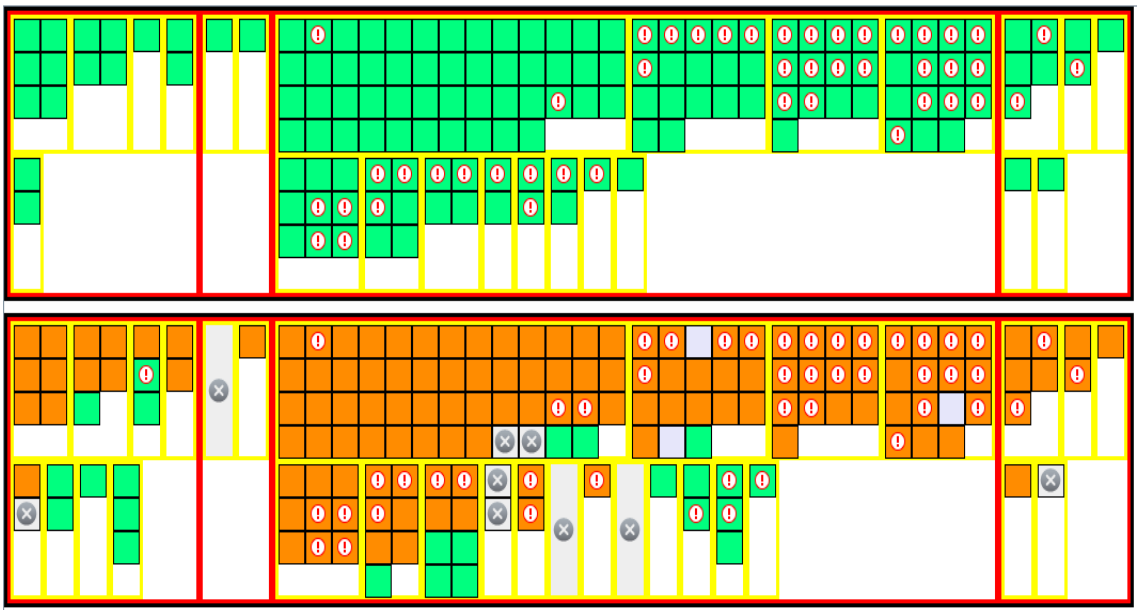
quantitativa e qualitativa geradas para o sistema de software PlayLister na Etapa 2 são apresentadas na Figura 7.5 e na Figura 7.6, respectivamente.

Figura 7.5. Visualização Quantitativa utilizada na Avaliação



Fonte: Do autor (2017).

Figura 7.6 - Visualização Qualitativa utilizada na Avaliação



Fonte: Do autor (2017).

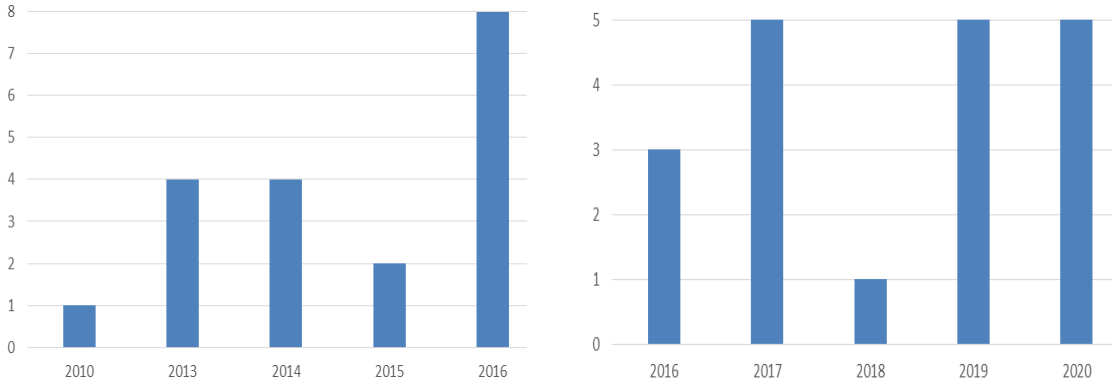
7.2.2 Execução

Os participantes do estudo são alunos de graduação de duas Instituições Federais de Ensino Superior (IFES). O estudo foi executado como parte de uma oficina de Visualização de Software ministrada pela pesquisadora responsável, no qual os interessados em participar

do evento deveriam efetuar sua inscrição. Desse modo, a seleção dos participantes pode ser considerada aleatória, visto que não houve influência dos pesquisadores na escolha dos participantes.

Os resultados obtidos nas duas IFES foram condensados antes de serem analisados, totalizando uma amostra única composta por 19 participantes. O conjunto de formulários respondidos pelos participantes recebeu aleatoriamente uma identificação que varia de P1 a P19, sendo utilizada como seu identificador único. O formulário de caracterização do participante aplicado no início do estudo foi utilizado para coletar algumas informações a respeito desses participantes. Na Figura 7.7A, é apresentado o ano de ingresso dos participantes, alunos dos cursos de Ciência da Computação e de Sistemas de Informação, no qual aproximadamente 40% iniciaram seus estudos no ano de 2016. Na Figura 7.7B, é apresentado o ano previsto pelos participantes para conclusão do curso.

Figura 7.7 - Ano de Ingresso/Término da Graduação dos Participantes



(A) (B)

Fonte: Do autor (2017).

Na Tabela 7.8 são apresentadas a experiência dos participantes com desenvolvimento de sistemas de software orientados a objetos e a linguagem de programação que tiveram contato. Como pode ser observado, 13 participantes tiveram contato com orientação a objetos em trabalhos de graduação. Dentre esses, 3 participantes também tiveram contato com esse paradigma em projetos no ambiente acadêmico. Apenas 1 participante desenvolveu software orientado a objetos em projetos pessoais e nenhum participante possui experiência na indústria. Além disso, 5 participantes afirmaram nunca ter tido contato com orientação a objetos.

Considerando os participantes que tiveram contato com orientação a objetos, 74% (14 participantes) relataram conhecer Java (utilizada neste estudo), 16% (3 participantes) relataram conhecer C# e 10% (2 participantes) relataram conhecer PHP. Os participantes

também informaram o tempo em que trabalharam em cada tipo de experiência; o participante que relatou ter experiência com orientação a objetos em projetos pessoais possui 32 meses de trabalho nessa atividade.

Tabela 7.8 - Experiência dos Participantes com Orientação a Objetos

Participante	Trabalho de Graduação	Projetos Pessoais	Projetos no Ambiente Acadêmico	Projetos na Indústria	Linguagem Utilizada
P1	✔				Java
P2	✔		✔		Java, PHP, C#
P3					
P4					
P5					
P6					
P7					
P8	✔				Java
P9	✔				Java
P10	✔				Java
P11	✔				Java
P12	✔				Java
P13	✔		✔		Java, C#
P14	✔		✔		Java, C#
P15	✔				Java
P16		✔	✔		Java
P17	✔				Java
P18	✔				Java
P19	✔				Java, PHP

Legenda: "✔" significa que o participante possui certa experiência no atributo em questão.

Fonte: Do autor (2017).

Na Tabela 7.9, é apresentada a experiência dos participantes em relação a alguns assuntos correlatos a abordagem DCEVizz. Os números correspondem ao grau de experiência em cada área, podendo assumir os seguintes valores: 0 = Nenhum; 1 = Estudado em aula ou em livro; 2 = Pratiquei em projetos em sala de aula; 3 = Utilizei em projetos pessoais; e 4 = Utilizei na indústria. Como pode ser observado, a maioria dos participantes teve pouco ou nenhum contato com alguns temas relacionados a abordagem DCEVizz.

Apesar da pouca experiência, os participantes tiveram contato prévio com assuntos relacionados a abordagem antes do início do experimento. Durante o minicurso, no qual foi realizado o estudo, foram apresentados conceitos introdutórios sobre a utilização de visualização durante a manutenção de sistemas de software, as consequências da presença de código morto na legibilidade do código e os efeitos da evolução na manutenibilidade de sistemas de software. Assim, os participantes ficaram familiarizados com o propósito da abordagem DCEVizz, tornando-os mais aptos a utilizá-la e avaliá-la.

7.2.3 Análise e discussão dos resultados

As respostas obtidas com os questionários foram analisadas quantitativa e qualitativamente. Na análise quantitativa, foram efetuados cálculos estatísticos e testes de hipóteses utilizando as variáveis precisão e eficiência. A análise qualitativa foi feita utilizando as respostas obtidas com o Formulário de Opinião, com as questões adicionais do formulário da Etapa 2 e com a escala de Likert definida nas questões do formulário da Etapa 1 e do formulário da Etapa 2.

Tabela 7.9 - Experiência dos Participantes em Assuntos Correlatos ao Estudo

Participante	Engenharia de Software	Manutenção de Software	Legibilidade de Código	Código Morto	Evolução de Software	Visualização de Software
P1	2	1	2	1	1	1
P2	2	4	4	1	4	2
P3	0	0	1	0	0	1
P4	0	0	2	0	0	0
P5	0	0	2	0	0	0
P6	0	0	0	0	0	0
P7	0	0	2	0	0	0
P8	2	1	1	1	0	1
P9	0	0	0	1	0	1
P10	1	1	1	0	0	0
P11	1	0	1	0	0	0
P12	1	1	1	0	1	1
P13	2	3	0	0	0	0
P14	2	1	2	0	0	0
P15	1	1	1	0	1	1
P16	1	2	3	1	2	2
P17	2	1	0	0	1	1
P18	0	2	3	1	0	2
P19	3	0	0	0	0	0

Fonte: Do autor (2017).

7.2.3.1 Análise quantitativa

Os valores das variáveis precisão e eficiência foram calculados para cada participante, de acordo com as fórmulas descritas no início da Seção 7.3. Na Tabela 7.10, são apresentados os resultados obtidos com esse cálculo, considerando as respostas do formulário da Etapa 1 e do formulário da Etapa 2, exceto as 3 questões adicionais.

Na Figura 7.8, é apresentada a Média e o Desvio Padrão geral das variáveis precisão e eficiência. O valor dessas variáveis foi calculado considerando as questões com foco na identificação e na compreensão da evolução do código morto. Como pode ser observado, o uso de DCEVizz Tool resultou em ligeiro aumento na Média das duas variáveis e no Desvio

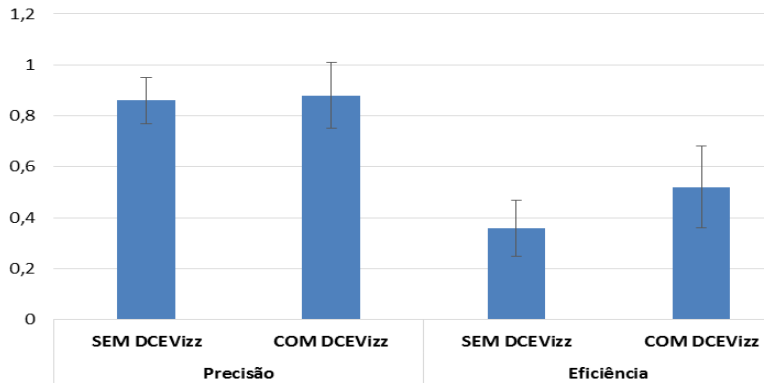
Padrão, indicando maior grau de dispersão dos dados em relação à Média. Esse fato pode ter ocorrido por causa da pouca familiaridade dos participantes com a ferramenta, sendo que cada um deles demanda tempo diferente para compreendê-la, resultando em variação mais significativa dos dados.

Tabela 7.10 - Valores das Variáveis Precisão e Eficiência (Todas as Questões)

Participante	Precisão		Eficiência	
	Sem DCEVizz	Com DCEVizz	Sem DCEVizz	Com DCEVizz
P1	0,7	1	0,27	0,32
P2	0,9	1	0,38	0,4
P3	0,8	0,8	0,26	0,34
P4	0,9	0,7	0,29	0,41
P5	1	0,7	0,27	0,35
P6	0,9	0,8	0,25	0,62
P7	0,9	0,9	0,24	0,23
P8	0,9	1	0,4	0,38
P9	0,9	1	0,62	0,52
P10	0,9	0,8	0,3	0,45
P11	0,8	0,6	0,41	0,66
P12	0,9	1	0,5	0,76
P13	0,9	1	0,52	0,71
P14	1	0,8	0,45	0,71
P15	0,9	1	0,5	0,66
P16	0,9	1	0,45	0,66
P17	0,8	1	0,33	0,44
P18	0,9	0,9	0,35	0,71
P19	0,6	0,8	0,21	0,71
Média	**0,86**	**0,88**	**0,36**	**0,52**
Desvio Padrão	**0,09**	**0,13**	**0,11**	**0,16**

Fonte: Do autor (2017).

Figura 7.8 - Média e Desvio Padrão das Variáveis Precisão e Eficiência



Fonte: Do autor (2017).

Para investigar se houve contribuição de DCEVizz Tool especificamente na identificação de código morto **e/ou** na compreensão de sua evolução, os valores das variáveis precisão e eficiência foram analisados separadamente para cada grupo de questões definidas nos formulários. Mais especificamente, o cálculo da precisão e da eficiência foi feito para as

questões de 1 a 5 (com foco na identificação de código morto) utilizando o formulário da Etapa 1 e o formulário da Etapa 2. Posteriormente, o mesmo cálculo foi realizado para as questões de 6 a 10, com foco na compreensão da evolução do código morto. Vale ressaltar que as questões adicionais não foram consideradas nessa análise.

Os dados utilizados nesse estudo são provenientes da avaliação de alguns participantes submetidos a duas situações diferentes, nas quais executaram determinadas atividades **sem** e **com** o uso de DCEVizz Tool. Dessa forma, foram obtidas duas amostras de valores das variáveis precisão e eficiência: i) sem o uso do *plug-in*; e ii) com o uso do *plug-in*. Como os elementos de cada amostra foram obtidos a partir de um mesmo participante, essas amostras podem ser caracterizadas dependentes ou pareadas (BARBETTA et al., 2010). A análise desse tipo de amostra permite verificar se houve alteração na Média quando cada participante foi submetido nas duas situações, permitindo verificar o efeito de DCEVizz Tool nas variáveis consideradas nesse estudo.

A análise estatística de duas amostras pareadas depende da distribuição dos dados. Quando os dados possuem distribuição normal, a análise estatística pode ser feita por meio do *Teste t Pareado*. Em contrapartida, se os dados não possuem distribuição normal, deve ser utilizado o *teste não paramétrico de Wilcoxon* (*teste de Wilcoxon*). Para execução de ambos os testes, é necessário definir uma hipótese nula (H_0) e uma hipótese alternativa (H_1), sendo H_0 a hipótese colocada à prova no teste, indicando igualdade a ser contestada e H_1 a suposição de que existe desigualdade entre as Médias de duas populações (contradizendo H_0).

A hipótese H_1 comumente representa o que se quer provar, correspondendo à própria hipótese de pesquisa formulada em termos de parâmetros (BARBETTA et al., 2010). Neste estudo, a hipótese H_1 foi definida para verificar se DCEVizz Tool aumentou os valores das variáveis precisão e eficiência. Dessa forma, espera-se que a diferença entre as Médias obtidas sem e com o uso do *plug-in* seja menor do que 0, caracterizando a hipótese H_1 como unilateral à esquerda. A aceitação/rejeição de H_0 depende dos valores das estatísticas calculadas em cada teste. Para o *Teste t Pareado* unilateral à esquerda, após o cálculo da estatística *T*, a regra de decisão pode ser dada da seguinte forma:

se $T_{observado} < -T_{tabelado}$[17] **então**
    rejeitar H_o
senão
    não rejeitar H_o

[17] A tabela de valores críticos para o *Teste t Pareado* pode ser obtida em: (BARBETTA et al., 2010), pg.379

Após o cálculo da estatística *W* do teste de Wilcoxon, a seguinte regra de decisão pode ser adotada:

se $W_{calculado} < W_{tabelado}$[18] **então**
rejeitar H_o
senão
não rejeitar H_o

Além dos valores das estatísticas calculadas nos testes, o *p-value* também pode ser utilizado para avaliação das hipóteses. O *p-value* pode ser interpretado como a medida do grau de concordância entre os dados e H_0. Quanto maior o *p-value*, maior o indicativo de aceitação de H_0. Dessa forma, se o *p-value* for menor do que o nível de significância adotado no estudo, existe indícios estatisticamente significantes de que a hipótese H_1 deve ser aceita. Neste estudo, o nível de significância (α) utilizado nas análises estatísticas foi α = 0,05 (95% de confiança).

Antes da aplicação dos testes estatísticos, foi verificada a existência de *outliers* e a normalidade dos dados para as variáveis precisão e eficiência. Os participantes identificados como *outliers* foram desconsiderados apenas dos testes da variável em questão. Essa análise foi feita utilizando gráficos *box plot* gerados pela ferramenta Action Stat[19]. A normalidade foi verificada com o *teste de Shapiro-Wilk*, considerado eficiente para análise de dados com diferentes distribuições e para variados tamanhos de amostras. Esse teste foi executado com auxílio da ferramenta Action Stat, fornecendo valores para a estatística *S* do *teste de Shapiro-Wilk* e *p-value*, utilizados como parâmetro para aceitação ou rejeição de H_0. No teste de normalidade, a hipótese nula (H_0) e a hipótese alternativa (H_1) pode ser definida da seguinte forma:

$$\begin{cases} H_0\text{: Os dados seguem uma distribuição normal} \\ H_1\text{: Os dados não seguem uma distribuição normal} \end{cases}$$

A regra de decisão adotada para aceitação ou rejeição de H_0 pode ser baseada no valor da estatística *S* ou no *p-value*. Caso a estatística *S* seja considerada, a seguinte regra de decisão deve ser utilizada:

[18] A tabela de valores críticos para o *teste de Wilcoxon* pode ser obtida em: [BARBETTA et al., 2010), pg.387
[19] http://www.portalaction.com.br/

se $S_{calculado} < S_{tabelado}$[20] **então**
    rejeitar H_o //os dados não possuem distribuição normal
Senão
    não rejeitar H_o //os dados possuem distribuição normal

Caso seja considerado o *p-value*, a regra de decisão a ser utilizada, considerando o nível de significância α = 0,05, é:

se $p - value \leq \alpha$ **então**
    rejeitar H_o //os dados não possuem distribuição normal
Senão
    não rejeitar H_o //os dados possuem distribuição normal

Com base nesses conceitos, são apresentadas nas subseções seguintes os valores obtidos para as variáveis precisão e eficiência em cada grupo de questões, além dos resultados das análises de *outliers*, dos testes de normalidade e dos testes utilizados para aceitação ou rejeição das hipóteses.

7.2.3.1.1 Análise com foco na identificação de código morto

Seja X_1, X_2, ..., X_n os valores amostrais obtidos para uma das variáveis (precisão ou eficiência) na primeira situação, em que os usuários realizaram as atividades sem o uso de DCEVizz Tool, e seja Y_1, Y_2, ..., Y_n os valores amostrais obtidos na segunda situação, quando os mesmos participantes realizaram as atividades utilizando o *plug-in*. Se o uso de DCEVizz Tool não tiver efeito sobre as variáveis, espera-se que, em Média, os valores observados nas duas situações sejam iguais. Em termos de Desvio Padrão, se não existir efeito significativo, as diferenças $X_1 - Y_1 = D_1$, $X_2 - Y_2 = D_2$, ..., $X_n - Y_n = D_n$ seriam, em média, iguais a zero. Na Tabela 7.11, são apresentados os valores X, Y e D, além da Média e do Desvio Padrão para as variáveis precisão e eficiência, calculadas a partir das respostas das questões de 1 a 5 dos formulários. Como pode ser observado, o uso de DCEVizz Tool resultou em ligeiro aumento dos valores da precisão e da eficiência para a maioria dos participantes, refletindo no aumento da Média para ambas as variáveis (FIGURA 7.9).

O crescimento desses valores pode indicar que o *plug-in* auxiliou na execução de atividades para identificar código morto. Além da Média, o Desvio Padrão aumentou com o uso do *plug-in*, provavelmente por causa da curva de aprendizado necessária para as técnicas

[20] A tabela de valores críticos para o teste de Shapiro-Wilk pode ser obtida em: http://www.portalaction.com.br/inferencia/64-teste-de-shapiro-wilk

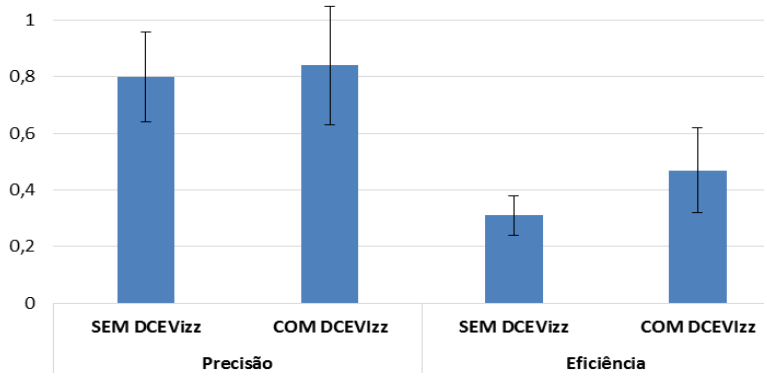
de visualização da abordagem DCEVizz, resultando em maior variação dos valores obtidos pelos participantes. Para verificar a existência de evidências estatísticas que comprovam os efeitos de DCEVizz Tool na identificação de código morto, a precisão e a eficiência foram analisadas separadamente por meio de testes de hipóteses. A análise de *outliers* e o teste de normalidade foram efetuados utilizando o conjunto `D` de diferenças obtido para cada variável.

Tabela 7.11 - Valores das Variáveis Precisão e Eficiência (Questões de 1 a 5)

Participante	Precisão			Eficiência		
	Sem DCEVizz	Com DCEVizz	Diferença	Sem DCEVizz	Com DCEVizz	Diferença
P1	0,6	1	-0,4	0,25	0,27	-0,02
P2	1	1	0	0,31	0,5	-0,19
P3	0,6	0,6	0	0,31	0,27	0,04
P4	0,8	0,4	0,4	0,31	0,41	-0,1
P5	1	0,4	0,6	0,2	0,31	-0,11
P6	0,8	0,8	0	0,21	0,62	-0,41
P7	0,8	0,8	0	0,23	0,14	0,09
P8	0,8	1	-0,2	0,33	0,41	-0,08
P9	0,8	1	-0,2	0,38	0,45	-0,07
P10	1	0,6	0,4	0,25	0,62	-0,37
P11	0,8	0,6	0,2	0,45	0,71	-0,26
P12	0,8	1	-0,2	0,38	0,55	-0,17
P13	0,8	1	-0,2	0,38	0,55	-0,17
P14	1	1	0	0,38	0,71	-0,33
P15	1	1	0	0,38	0,5	-0,12
P16	0,8	1	-0,2	0,38	0,55	-0,17
P17	0,6	1	-0,4	0,26	0,31	-0,05
P18	0,8	1	-0,2	0,35	0,62	-0,27
P19	0,4	0,8	-0,4	0,22	0,55	-0,33
Média	**0,8**	**0,84**	**-0,042**	**0,31**	**0,47**	**-0,16**
Desvio Padrão	**0,16**	**0,21**	**0,27**	**0,07**	**0,15**	**0,13**

Fonte: Do autor (2017).

Figura 7.9 - Média e Desvio Padrão da Precisão e Eficiência

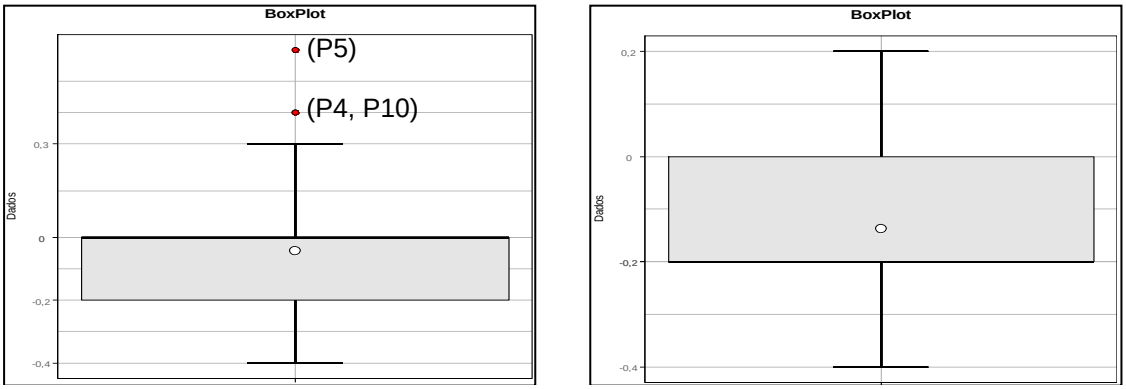


Fonte: Do autor (2017).

Análise da Variável Precisão. Conforme apresentado no gráfico *box plot* da Figura 7.10A, alguns participantes foram considerados *outliers* para a variável precisão, representados pelos pontos vermelhos. Os participantes *outliers* P4 e P10 obtiveram a diferença `D = 0,4` e P5

obteve $D = 0,6$. Na Figura 7.10B, é apresentado o gráfico *box plot* gerado após eliminar esses três participantes da amostra. Como pode ser observado, não houve outros *outliers* na iteração seguinte, resultando em uma amostra de tamanho $n = 16$.

Figura 7.10 - *Outliers* para Precisão (Identificação de Código Morto)



Fonte: Do autor (2017).

Com os *outliers* eliminados da amostra, foi realizado o teste de normalidade dos dados para definir se devem ser utilizados testes paramétricos ou não paramétricos na avaliação das hipóteses. Para $\alpha = 0,05$ e $n = 16$, o valor crítico obtido na tabela do *teste de Shapiro-Wilk* é $S_{tabelado} = 0,887$. Como pode ser observado na Figura 7.11, $S_{calculado} = 0,882 < S_{tabelado}$ e *p-value* $= 0,0427 < \alpha$, implicando que H_0 (normalidade dos dados) deve ser rejeitada. Portanto, a amostra para a variável precisão com foco na identificação de código morto não possui distribuição normal, fazendo necessário o uso do *teste de Wilcoxon*.

Figura 7.11 - *Teste de Shapiro-Wilk* para a Precisão (Identificação de Código Morto)

Testes de Normalidade		
Testes	*Estatísticas*	*P-valores*
Shapiro - Wilk	0,882684689	0,0427

Fonte: Do autor (2017).

Para execução do teste de Wilcoxon, além dos *outliers*, devem ser desconsiderados da amostra os participantes, cujo $D = 0$ (BARBETTA et al., 2010). Dessa forma, foram desconsiderados 6 participantes, resultando em uma amostra de tamanho $n = 10$. Para $\alpha = 0,05$ e $n = 10$, o valor crítico obtido na tabela de Wilcoxon é $W_{tabelado} = 11$. As seguintes hipóteses foram definidas para análise da variável precisão utilizando o teste de *Wilcoxon*:

H_0: DCEVizz Tool não altera a precisão em atividades para identificar código morto

H_1: DCEVizz Tool aumenta a precisão em atividades para identificar código morto

Na Figura 7.12, são apresentados os resultados obtidos com o cálculo das estatísticas do teste de Wilcoxon pela ferramenta Action Stat. Pela regra de decisão desse teste, como $W_{calculado} = 7 < W_{tabelado}$ e *p-value* $= 0,0182 < \alpha$, existem evidências estatisticamente significativas para rejeitar H_0. Logo, com a rejeição de H_0 e aceitação de H_1, pode-se concluir que DCEVizz Tool aumentou a precisão das atividades executadas para identificar código morto.

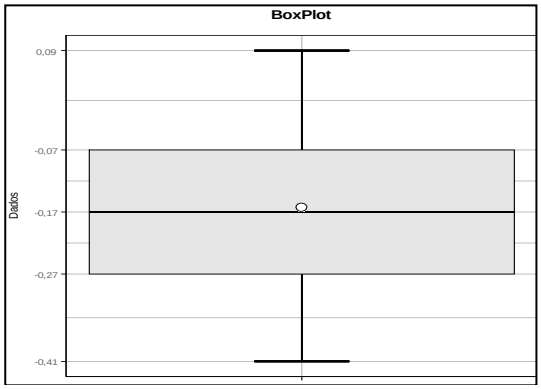
Figura 7.12 - *Teste de Wilcoxon* para a Precisão (Identificação de Código Morto)

Tabela da Estatística do Teste (Wilcoxon)	
Informações	*Valores*
Estatística	7
P-valor	0,0182
Hipótese Nula	0
Limite Inferior	-Inf
(Pseudo) Mediana	-0,200006552
Limite Superior	-0,199961987
Nível de Confiança	0,95

Fonte: Do autor (2017).

Análise da Variável Eficiência. Conforme apresentado na Figura 7.13, não foram identificados *outliers* para a variável eficiência com foco na identificação de código morto (ausência de pontos vermelhos no gráfico). Desse modo, a quantidade de participantes considerados na análise dessa variável é $n = 19$.

Figura 7.13 - *Outliers* para Eficiência (Identificação de Código Morto)



Fonte: Do autor (2017).

Na Figura 7.14, são apresentados os valores calculados para o *teste de Shapiro-Wilk*. O valor crítico obtido na tabela é $S_{tabelado} = 0,901$, considerando $\alpha = 0,05$ e $n = 10$.

Como $S_{calculado} = 0,975 > S_{tabelado}$ e *p-values* $= 0,876 > \alpha$, deve-se aceitar H_0. Portanto, os dados possuem distribuição normal, implicando no uso do *Teste t Pareado* para avaliação das hipóteses.

Figura 7.14 - *Teste de Shapiro-Wilk* Eficiência (Identificação de Código Morto)

Testes de Normalidade		
Testes	Estatísticas	P-valores
Shapiro - Wilk	0,975354097	0,876

Fonte: Do autor (2017).

Para a análise da variável eficiência no contexto de identificação de código morto, as seguintes hipóteses foram definidas:

$$\begin{cases} H_0: \text{DCEVizz Tool não altera a eficiência em atividades para identificar código morto} \\ H_1: \text{DCEVizz Tool aumenta a eficiência em atividades para identificar código morto} \end{cases}$$

Na Figura 7.15, são apresentadas as estatísticas calculadas com o *Teste t Pareado*. O valor crítico obtido na tabela é $T_{tabelado} = 1,734$, considerando $\alpha = 0,05$ e 18 graus de liberdade ($n - 1$). Como resultado, obteve-se que $T_{calculado} = -5,128 < -T_{tabelado} = -1,734$ e *p-value* $= 0,00000352 < \alpha$. Dessa forma, de acordo com a regra de decisão do *Teste t Pareado*, existem evidências estatísticas para rejeitar H_0. Logo, com a rejeição de H_0 e aceitação de H_1, pode-se concluir que DCEVizz Tool aumentou a eficiência das atividades executadas para identificar código morto.

Figura 7.15 - Teste *t* Pareado para Eficiência (Identificação de Código Morto)

Tabela da Estatística do Teste (Teste t - Pareado)	
Resultados	
Estatística T	-5,128126
Graus de Liberdade	18
P-valor	3,52E-05
Média da Amostra 1	0,3136842
Média da Amostra 2	0,4763158
Desvio Padrão das diferenças	0,1382366
Tamanho das Amostras	19
Hipótese Alternativa Menor que	0
Nível de Confiança	95%
Limite Inferior	-Inf
Limite Superior	-0,1076381

Fonte: Do autor (2017).

7.2.3.1.2 Análise com foco na evolução do código morto

Na Tabela 7.12, são apresentados os valores X_i (sem o uso de DCEVizz Tool), Y_i (com o uso de DCEVizz Tool) e D_i (dado por $X_i - Y_i$) para as variáveis precisão e eficiência, com base nas respostas do participante Pi para as questões de 6 a 10 (com foco na compreensão da evolução de código morto). Além disso, são apresentadas a Média e o Desvio Padrão, maiores para ambas as variáveis com o uso de DCEVizz Tool, indicando sua utilidade em atividades que demandam compreensão da evolução. Esses valores podem ser visualizados graficamente na Figura 7.16. Testes de hipóteses foram realizados para verificar a existência de evidências estatísticas que comprovam o efeito de DCEVizz Tool nas variáveis analisadas.

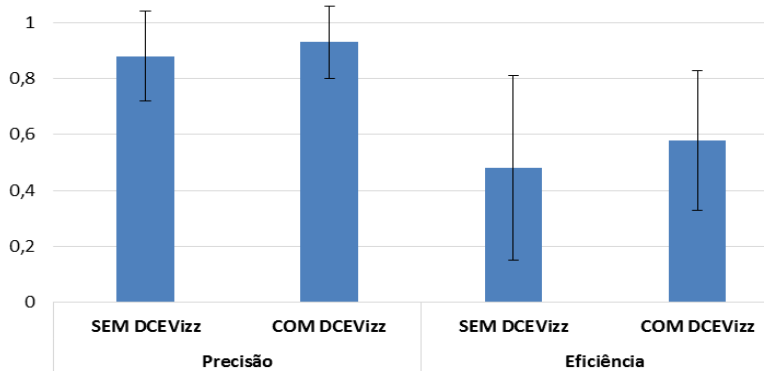
Tabela 7.12 - Valores das Variáveis Precisão e Eficiência (Questões de 6 a 10)

Participante	Precisão			Eficiência		
	Sem DCEVizz	Com DCEVizz	Diferença	Sem DCEVizz	Com DCEVizz	Diferença
P1	0,8	1	-0,2	0,29	0,38	-0,09
P2	0,8	1	-0,2	0,5	0,33	0,17
P3	1	1	0	0,21	0,45	-0,24
P4	1	1	0	0,27	0,41	-0,14
P5	1	1	0	0,38	0,41	-0,03
P6	1	0,8	0,2	0,31	0,62	-0,31
P7	1	1	0	0,25	0,62	-0,37
P8	1	1	0	0,5	0,35	0,15
P9	1	1	0	1,66	0,62	1,04
P10	0,8	1	-0,2	0,38	0,35	0,03
P11	0,8	0,6	0,2	0,38	0,62	-0,24
P12	1	1	0	0,71	1,25	-0,54
P13	1	1	0	0,83	1	-0,17
P14	1	0,6	0,4	0,55	0,71	-0,16
P15	1	1	0	0,71	1	-0,29
P16	0,8	1	-0,2	0,38	0,55	-0,17
P17	0,6	1	-0,4	0,26	0,31	-0,05
P18	0,8	1	-0,2	0,35	0,62	-0,27
P19	0,4	0,8	-0,4	0,22	0,55	-0,33
Média	**0,88**	**0,93**	**-0,05**	**0,48**	**0,58**	**-0,1**
Desvio Padrão	**0,16**	**0,13**	**0,19**	**0,33**	**0,25**	**0,32**

Fonte: Do autor (2017).

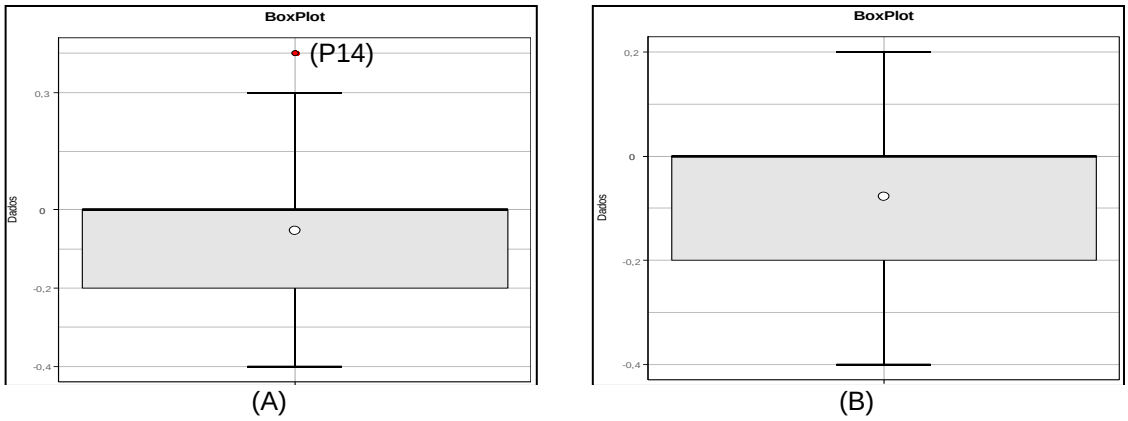
Análise da Variável Precisão. Na Figura 7.17, são apresentados os gráficos *box plot* para análise de *outliers* da variável precisão. Como pode ser observado na Figura 7.17A, o participante *outlier* é P14 ($D = 0,4$), removido das análises estatísticas. Na Figura 7.17B, é apresentado o gráfico *box plot* gerado após eliminar P14 da amostra, mostrando que na segunda iteração não há outros *outliers*. Dessa forma, o tamanho da amostra nessa análise é $n = 18$. Na Figura 7.18, são apresentadas as estatísticas obtidas para o *teste de Shapiro-Wilk*, executado para verificar a normalidade dos dados após a remoção do *outlier*.

Figura 7.16 - Média e Desvio Padrão da Precisão e Eficiência



Fonte: Do autor (2017).

Figura 7.17 - *Outliers* para Precisão (Compreensão da Evolução do Código Morto)



Fonte: Do autor (2017).

Para nível de significância $\alpha = 0,05$ e $n = 18$, o valor crítico obtido na tabela é $S_{tabelado} = 0,897$. Pela regra de decisão do *teste de Shapiro-Wilk*, como $S_{calculado} = 0,867 < S_{tabelado}$ e *p-value* $= 0,0159 < \alpha$, H_0 deve ser rejeitada. Portanto, os valores da variável precisão não possuem distribuição normal, implicando no uso do *teste de Wilcoxon* para avaliação das hipóteses. Para a análise da variável precisão no contexto de compreensão da evolução de código morto, as seguintes hipóteses foram definidas:

H_0: DCEVizz Tool não altera a precisão em atividades para compreender a evolução do código morto

H_1: DCEVizz Tool aumenta a precisão em atividades para compreender a evolução do código morto

Além dos *outliers*, antes da execução do *teste de Wilcoxon*, foram eliminados da amostra os participantes cujo $D = 0$. Dessa forma, foi eliminado 1 participante *outlier* e 9 participantes com $D = 0$, resultando em uma amostra de tamanho $n = 9$. O valor crítico

obtido na tabela para $\alpha = 0,05$ e $n = 9$ é $W_{tabelado} = 9$. Na Figura 7.19, são apresentadas as estatísticas do *teste de Wilcoxon* calculadas pela ferramenta Action Stat. Pela regra de decisão do teste, como o valor $W_{calculado} = 10,5 > W_{tabelado}$ e *p-value* $= 0,0796 > \alpha$, não há evidências estatisticamente significativas para rejeitar H_0. Logo, pode-se concluir que o uso de DCEVizz Tool não teve influência estatisticamente significante na precisão das atividades executadas para compreender a evolução do código morto.

Figura 7.18 - *Teste de Shapiro-Wilk* para Precisão (Compreensão da Evolução)

Testes de Normalidade		
Testes	**Estatísticas**	**P-valores**
Shapiro - Wilk	0,867065283	0,0159

Fonte: Do autor (2017).

Figura 7.19 - *Teste de Wilcoxon* para Precisão (Compreensão da Evolução)

Tabela da Estatística do Teste (Wilcoxon)	
Informações	**Valores**
Estatística	10,5
P-valor	0,0796
Hipótese Nula	0
Limite Inferior	-Inf
(Pseudo) Mediana	-0,199937493
Limite Superior	-1,70E-05
Nível de Confiança	0,95

Fonte: Do autor (2017).

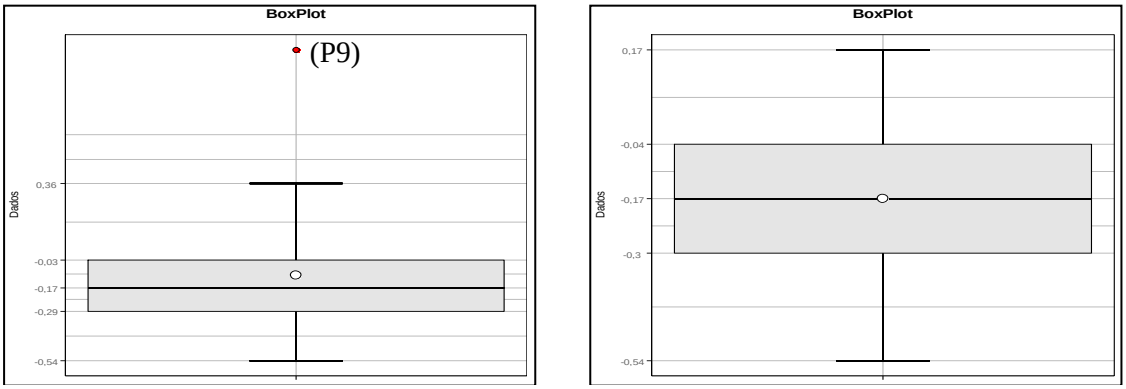
Apesar desse resultado, a Média da precisão obtida com o uso de DCEVizz Tool foi ligeiramente maior do que a Média obtida sem o *plug-in* (FIGURA 7.16). Esse fato pode indicar influência positiva do uso do *plug-in* nessa variável, visto que dentre os 19 participantes, apenas 16% (3 participantes) obtiveram piora da precisão com DCEVizz Tool, sendo que um deles ainda foi considerado *outlier* (P14). Além disso, 7 participantes (37%) obtiveram melhora e 9 participantes (47%) não sofreram alteração da precisão ao utilizar o *plug-in*.

Análise da Variável Eficiência. Na Figura 7.20, são apresentados os gráficos *box plot* para análise de *outliers* da variável eficiência. P9 foi identificado como *outlier* (FIGURA 7.20A), cujo $D = 1,04$. Na Figura 7.20B, é apresentado o gráfico *box plot* gerado após a eliminação desse participante. Como pode ser observado, não houve outros *outliers* na próxima iteração. Dessa forma, o tamanho da amostra utilizada na análise estatística é $n = 18$.

Na Figura 7.21, são apresentados os valores do *teste de Shapiro-Wilk*. O valor crítico obtido na tabela é $S_{tabelado} = 0,897$ para $\alpha = 0,05$ e $n = 18$. Como pode ser

observado, $S_{calculado} = 0,976 > S_{tabelado}$ e *p-value* $= 0,908 > \alpha$, resultando na aceitação de H_0. Portanto, os valores da variável eficiência possuem distribuição normal, implicando no uso do *Teste t Pareado* para avaliação das hipóteses.

Figura 7.20 - *Outliers* para Variável Eficiência (Compreensão da Evolução)



(A) (B)

Fonte: Do autor (2017).

Figura 7.21 - *Teste de Shapiro-Wilk* para Eficiência (Compreensão da Evolução)

Testes de Normalidade		
Testes	Estatísticas	P-valores
Shapiro - Wilk	0,976626964	0,9087

Fonte: Do autor (2017).

O valor crítico obtido na tabela é $T_{tabelado} = 1,740$, considerando $\alpha = 0,05$ e 17 graus de liberdade ($n - 1$). As hipóteses definidas para análise da variável eficiência no contexto de compreensão da evolução de código morto são dadas por:

H_0: DCEVizz Tool não altera a eficiência em atividades para compreender a evolução do código morto

H_1: DCEVizz Tool aumenta a eficiência em atividades para compreender a evolução do código morto

Na Figura 7.22, são apresentadas as estatísticas calculadas para o *Teste t Pareado*. Como resultado, obteve-se que $T_{calculado} = -3,976 < -T_{tabelado} = -1,740$ e *p-value* $= 0,000488 < \alpha$. Dessa forma, de acordo com a regra de decisão do *Teste t Pareado*, existem evidências estatísticas para rejeitar H_0. Logo, com a rejeição de H_0 e aceitação de H_1, pode-se concluir que DCEVizz Tool aumentou a eficiência das atividades executadas para compreender a evolução de código morto.

Figura 7.22 - *Teste t Pareado* para Eficiência (Compreensão da Evolução)

Tabela da Estatística do Teste (Teste t - Pareado) *Resultados*	
Estatística T	-3,976216
Graus de Liberdade	17
P-valor	0,000488124
Média da Amostra 1	0,4155556
Média da Amostra 2	0,585
Desvio Padrão das diferenças	0,180798
Tamanho das Amostras	18
Hipótese Alternativa Menor que	0
Nível de Confiança	95%
Limite Inferior	-Inf
Limite Superior	-0,09531199

Fonte: Do autor (2017).

7.2.3.2 Análise Qualitativa

A análise qualitativa foi feita com base nas respostas obtidas com o formulário de opinião, com as questões adicionais definidas no formulário da Etapa 2 e com a escala de Likert definida em cada questão dos formulários.

7.2.3.2.1 Análise do formulário de opinião

A finalidade do questionário de opinião foi coletar pontos positivos, pontos negativos e sugestões de melhoria para a abordagem DCEVizz. Foi solicitado aos participantes que descrevessem dificuldades encontradas durante a execução do estudo. Como resultado, 14 participantes (74%) relataram não ter tido dificuldades na realização das tarefas e 5 participates (26%) relataram que tiveram algumas dificuldades. A principal dificuldade relatada foi a falta de familiaridade com as representações visuais de DCEVizz Tool, o que deixou as atividades iniciais mais complexas de serem realizadas. Após o aprendizado da abordagem, a execução das tarefas tornou-se algo trivial.

Além disso, foi solicitado aos participantes que descrevessem o quanto o uso das visualizações de DCEVizz Tool auxiliou na execução das atividades. Como resultado, 15 participantes (79%) relataram que as visualizações facilitaram bastante e 4 participantes (21%) disseram que as visualizações facilitaram um pouco nessas atividades. Nenhum participante considerou que as visualizações não interferiram ou dificultaram a execução das atividades do estudo.

Para verificar de forma mais específica a compreensibilidade das visualizações, foi solicitado aos participantes que relatassem as dificuldades encontradas para interpretar a visualização quantitativa (gráfico de linha) e qualitativa (retângulos aninhados em uma matriz) de DCEVizz Tool. Como resultados, nenhum participante relatou ter tido dificuldade em compreender a visualização quantitativa. Em relação a visualização qualitativa, 13 participantes (68%) disseram não ter tido dificuldade e 6 participantes (32%) relataram alguma dificuldade no primeiro contato. Além disso, foi relatada certa dificuldade em compreender as cores e os símbolos utilizados na visualização.

Os participantes descreveram alguns pontos positivos de DCEVizz Tool, sintetizados nos itens seguintes:

a) Forma simples de apresentação das informações e facilidade para identificar código morto;
b) Redução do tempo para identificar o código morto e compreender sua evolução;
c) Visualizações intuitivas;
d) A visualização qualitativa é prática por representar o código morto e sua evolução em uma única imagem;
e) A abordagem é extremamente útil na execução de atividades para melhoria da qualidade do software;
f) Facilidade para perceber pacotes e classes com grande quantidade de métodos mortos.

Alguns participantes também descreveram pontos negativos, sintetizados nos itens seguintes:

a) Difícil interpretação à primeira vista;
b) Necessidade de conhecimento prévio para utilizar a abordagem;
c) Ausência de um manual/tutorial para facilitar o uso da abordagem.

Algumas sugestões de melhoria foram dadas pelos participantes, como mudança da tonalidade de algumas cores utilizadas na visualização qualitativa e disponibilização de um tutorial. Além disso, foram sugeridas algumas melhorias para DCEVizz Tool, como apresentar mensagens aos usuários quando determinado método não fosse encontrado no mecanismo de busca, inserir um recurso para exclusão automática dos métodos mortos e aumentar a velocidade de aparecimento dos *tooltips* com informações dos métodos. Alguns comentários adicionais foram feitos, como a utilidade de DCEVizz Tool para melhoria da qualidade do software. Outro participante enfatizou que, à princípio, as visualizações do *plug-in* dificultaram a execução das atividades, porém tornou-se fácil após o aprendizado. Por fim,

outro participante destacou que as visualizações podem não ser úteis para pessoas com deficiência visual.

De modo geral, pode-se perceber que alguns resultados esperados para a abordagem DCEVizz foram confirmados pelos participantes, como facilidade para identificar e compreender a evolução do código morto e a redução do tempo para executar essas atividades. Além disso, alguns aspectos negativos foram ressaltados, como a dificuldade inicial em compreender as visualizações.

Assim como qualquer outra ferramenta de visualização de software, DCEVizz Tool possui uma curva de aprendizado necessária para os usuários familiarizarem-se com suas representações visuais. Esse aprendizado é crucial na área de visualização, visto que há várias maneiras de representar visualmente a mesma informação por causa da característica abstrata do sistema de software. Acredita-se que, após o aprendizado, a compreensão das visualizações torna-se automática, facilitando a execução das atividades. Vale ressaltar que o tutorial solicitado por alguns participantes é um recurso disponível em DCEVizz Tool, que talvez pela falta de familiaridade, tenha passado despercebido por alguns participantes.

7.2.3.2.2 Análise das questões adicionais do Formulário da Etapa 2

As questões adicionais **Q11**, **Q12** e **Q13** do formulário da Etapa 2 foram definidas para investigar se DCEVizz tool auxilia a compreensão geral da evolução do código morto, fornecendo informações úteis aos engenheiros de software. Além disso, foi solicitada a opinião do participante sobre a dificuldade em responder essas questões sem o uso de DCEVizz Tool. Para tanto, ele deveria assinalar uma opção na escala de Likert referente a cada questão e justificar sua opinião. Essa justificativa foi solicitada para verificar se o participante compreendeu o propósito da questão e da abordagem, evitando que opiniões infundadas fossem consideradas na avaliação.

A finalidade da questão **Q11** foi verificar se a visualização quantitativa de DCEVizz possibilita a percepção de aumento/diminuição da quantidade de código morto ao longo das versões, permitindo a predição de características futuras do software. Essa questão foi definida como:

Q11. Observando a visualização quantitativa de DCEVizz Tool (gráfico de linha), a quantidade de métodos mortos aumentou ou diminuiu ao longo das versões?

A finalidade da questão **Q12** foi verificar a aplicabilidade da abordagem DCEVizz em um problema real, no qual deseja-se saber o pacote com maior quantidade de código morto. Essa questão foi definida como:

> **Q12.** A empresa deseja saber qual pacote possui maior quantidade de métodos mortos para que o mesmo seja priorizado na atividade de manutenção perfectiva. Com base na visualização qualitativa de DCEVizz Tool, qual pacote você indicaria para essa situação?

A finalidade da questão **Q13** foi verificar se a visualização qualitativa de DCEVizz possibilita uma percepção geral das características evolutivas do código morto e de suas mudanças ao longo das versões. Essa questão foi definida como:

> **Q13.** Observando a visualização qualitativa, qual pacote você acha que excluiu maior quantidade de métodos mortos da primeira para a segunda versão?

Na Tabela 7.13, é apresentada uma síntese dos resultados dessas questões adicionais. O valor presente na coluna "Dificuldade" corresponde à opção assinalada pelo participante na escala de Likert: 1 = Muito Fácil; 2 = Fácil; 3 = Mais ou Menos Fácil; 4 = Difícil; e 5 = Muito Difícil. A justificativa dada pela escolha da opção na escala de Likert mostrou que alguns participantes não compreenderam a questão, relatando a dificuldade em respondê-la com o uso de DCEVizz Tool. As opiniões desses participantes foram desconsideradas da análise, evitando que respostas infundadas tenham influência nos resultados do estudo. As opiniões desconsideradas foram representadas na tabela pelo símbolo "-". Como poder ser observado, houve 100% de acerto para as questões adicionais, indicando que DCEVizz Tool possibilitou aos participantes compreensão geral do código morto existente nas duas versões analisadas, além de suas características evolutivas. Considerando os 19 participantes, essas questões foram respondidas em tempo médio de aproximadamente 6 minutos, incluindo o tempo para justificativa.

Na Figura 7.23, é apresentada a opinião dos participantes sobre a dificuldade em responder as questões sem utilizar DCEVizz Tool. Como pode ser observado, a maioria dos participantes relatou ser "Difícil" ou "Muito Difícil" obter a compreensão de todo código morto dos sistemas de software sem o uso de DCEVizz Tool. De modo geral, eles alegaram que sem o *plug-in* seria necessário analisar manualmente todos os métodos das duas versões do sistema de software, anotar a quantidade de métodos mortos presente em cada classe e em

cada pacote e verificar a característica evolutiva de cada método, tornando-se um processo altamente custoso.

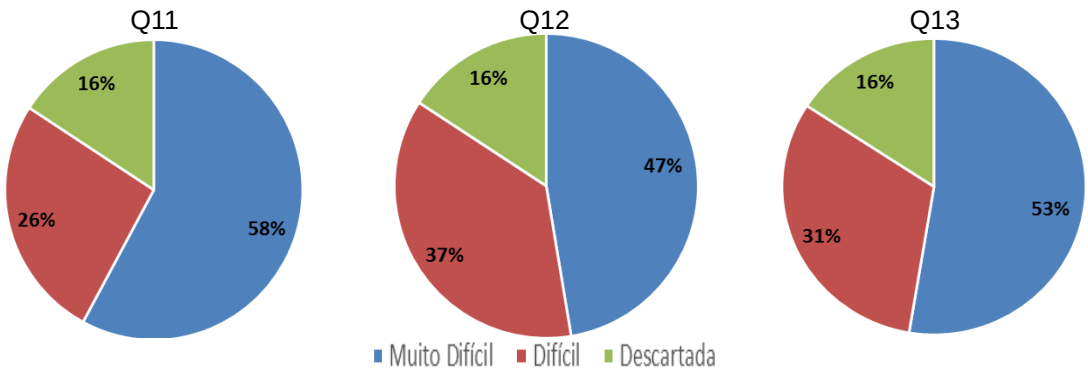
Tabela 7.13 - Resultados das Questões Adicionais (Formulário da Etapa 2)

Participante	Q11		Q12		Q13		Tempo (min)
	Acerto	Dificuldade	Acerto	Dificuldade	Acerto	Dificuldade	
P1	✔	5	✔	4	✔	5	6
P2	✔	4	✔	4	✔	4	11
P3	✔	-	✔	4	✔	4	5
P4	✔	5	✔	-	✔	-	8
P5	✔	-	✔	-	✔	-	8
P6	✔	4	✔	4	✔	4	7
P7	✔	5	✔	5	✔	5	6
P8	✔	4	✔	4	✔	5	11
P9	✔	5	✔	5	✔	5	8
P10	✔	5	✔	5	✔	5	5
P11	✔	4	✔	4	✔	4	5
P12	✔	5	✔	5	✔	5	5
P13	✔	5	✔	5	✔	5	5
P14	✔	5	✔	5	✔	5	3
P15	✔	5	✔	5	✔	4	5
P16	✔	4	✔	4	✔	4	5
P17	✔	-	✔	-	✔	-	7
P18	✔	5	✔	5	✔	5	4
P19	✔	5	✔	5	✔	5	7

Legenda: "✔" representa que o participante acertou a questão e "-" representa uma resposta desconsiderada.

Fonte: Do autor (2017).

Figura 7.23 - Dificuldade das Questões Adicionais (Formulário da Etapa 2)



Fonte: Do autor (2017).

7.2.3.2.3 Análise das escalas de Likert

Na Tabela 7.14, são apresentados os resultados obtidos com a escala de Likert definida em cada questão. Os números correspondem a opção assinalada pelo participante a respeito da dificuldade em realizar determinada atividade sem e com o uso de DCEVizz Tool, em que 1 =

Muito Fácil; 2 = Fácil; 3 = Mais ou Menos Fácil; 4 = Difícil; 5 = Muito Difícil. Como pode ser observado, 10 participantes (P1, P7, P10, P11, P12, P15, P16, P17, P18 e P19) não relataram aumento da dificuldade com o uso do *plug-in* para as questões do formulário. Dentre os outros 9 participantes, a maioria (P2, P6, P9, P13 e P14 - cinco participantes) relatou maior dificuldade com o uso do *plug-in* em apenas 1 questão e apenas P3, P4, P5 e P8 (quatro participantes) tiveram dificuldades em mais de uma questão. Apesar de alguns participantes terem indicado aumento da dificuldade com o uso de DCEVizz Tool, pode-se perceber que o *plug-in* facilitou ou manteve a dificuldade de execução da maioria das atividades dos formulários.

Tabela 7.14 - Resultados da Escala de Likert Definida para cada Questão

		P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15	P16	P17	P18	P19
Questão 1	SEM	3	1	3	1	2	4	3	1	2	2	2	3	2	3	3	2	3	2	3
	COM	3	2	3	3	2	3	3	2	3	1	1	3	3	3	3	2	2	2	2
Questão 2	SEM	2	2	3	2	2	3	3	1	3	2	1	3	2	3	3	2	3	3	3
	COM	2	2	3	3	2	4	3	2	2	1	1	3	2	2	3	2	2	2	2
Questão 3	SEM	2	2	4	2	2	3	3	2	2	2	2	3	2	3	3	2	3	2	3
	COM	1	2	4	3	3	3	3	3	2	1	1	2	1	3	2	2	2	1	1
Questão 4	SEM	2	3	4	2	2	3	3	2	4	3	3	4	3	3	4	3	4	2	4
	COM	1	2	5	3	3	3	3	2	1	1	1	2	2	2	3	2	3	1	1
Questão 5	SEM	4	2	5	3	3	3	4	3	4	4	5	4	5	3	5	2	4	4	4
	COM	1	2	4	2	3	3	3	2	2	1	2	1	1	2	3	2	2	1	1
Questão 6	SEM	3	2	3	3	2	3	3	2	2	2	2	4	4	4	4	2	3	3	3
	COM	1	2	2	2	2	3	3	2	2	1	1	2	2	1	3	1	2	1	1
Questão 7	SEM	3	2	4	3	2	3	3	2	3	2	2	4	4	2	4	2	4	3	3
	COM	2	2	3	2	2	3	3	2	2	1	1	1	1	3	1	1	2	1	1
Questão 8	SEM	4	2	4	3	2	3	3	3	4	2	2	5	5	4	4	2	4	3	4
	COM	1	2	4	2	3	3	1	2	3	1	1	1	2	1	2	1	2	1	2
Questão 9	SEM	3	2	3	4	2	4	4	2	2	2	2	4	4	3	4	2	4	3	4
	COM	1	2	5	2	2	3	3	2	2	1	2	2	1	2	2	1	2	1	1
Questão 10	SEM	3	2	5	3	2	3	4	3	3	3	2	5	4	4	4	2	4	4	2
	COM	1	2	5	2	2	3	3	2	2	1	2	1	2	2	1	1	2	1	1

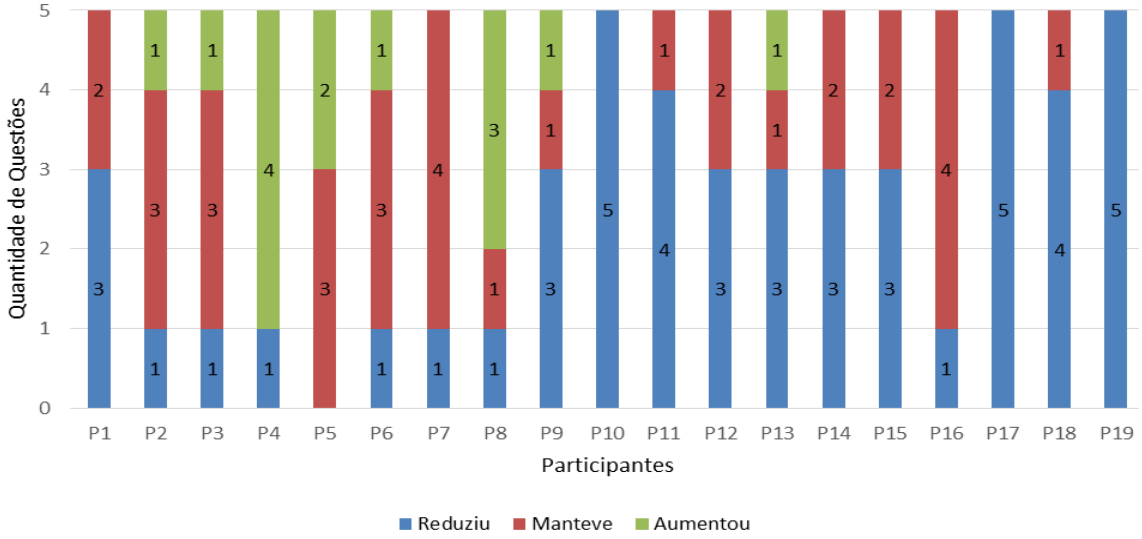
Fonte: Do autor (2017).

Para verificar os efeitos de DCEVizz Tool na facilidade de execução das atividades em cada foco, foram contabilizadas as questões cuja dificuldade foi reduzida, aumentada ou não alterada com o uso do *plug-in* nos focos em análise. Na Figura 7.24 e na Figura 7.25, são apresentados os resultados dessa contabilização para o foco de identificação e de compreensão da evolução do código morto, respectivamente. Conforme ilustrado, a maioria dos participantes (que relatou aumento da dificuldade com o uso de DCEVizz Tool) foi para as questões do foco de identificação de código morto. Em contrapartida, no foco de compreensão da evolução, apenas 3 participantes relataram aumento dessa dificuldade.

Esses resultados indicam que, na opinião dos participantes, DCEVizz Tool contribuiu de forma mais significativa na redução da dificuldade das questões que demandam

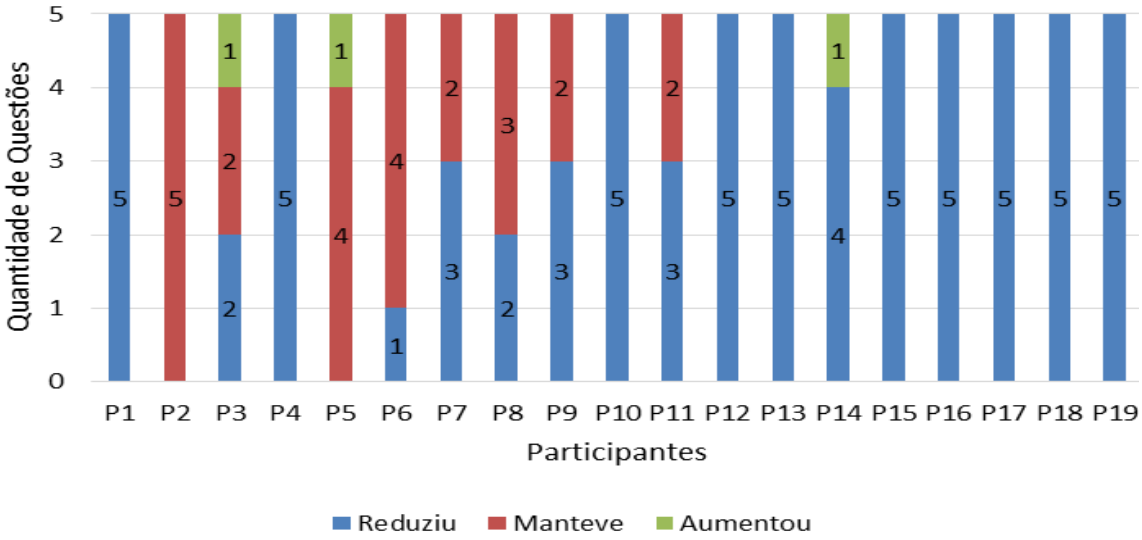
compreensão da evolução de código morto. As visualizações de DCEVizz pode ter contribuído para esses resultados, visto que o destaque visual das características evolutivas do código morto elimina a necessidade de analisar separadamente cada versão. Além disso, a maneira específica na qual as questões foram definidas nos formulários fez com que apenas pacotes, classes e métodos específicos fossem analisados. Dessa forma, foi eliminada a necessidade de análise de todo o código dos sistemas de software, reduzindo o esforço de execução das atividades sem o uso do *plug-in*. Acredita-se que, em cenários reais, o efeito de DCEVizz Tool na facilidade de execução das atividades seria mais significativo.

Figura 7.24 - Resultados da Escala de Likert (Questões de 1 a 5)



Fonte: Do autor (2017).

Figura 7.25 - Resultados da Escala de Likert (Questões de 6 a 10)



Fonte: Do autor (2017).

7.3 Ameaças à validade

É comum a existência de questões que possam impactar ou limitar a validade dos resultados obtidos ao longo dos estudos. Essas questões são denominadas ameaças à validade, que podem ser classificadas em quatro tipos principais (TRAVASSOS et al., 2002):

a) **Quanto à validade interna**. Uma ameaça a validade interna prejudica o conhecimento do grau em que os resultados da pesquisa refletem a realidade observada (relacionamento de causa e efeito). Neste estudo, as ameaças internas são:
   - Em relação à primeira fase da avaliação, possíveis falhas no funcionamento do *script* que compara o código morto identificado por DCEVizz Tool e por Understand podem afetar a validade das características identificadas nos resultados de cada ferramenta;
   - Em relação à segunda fase da avaliação, os valores calculados para a variável precisão dependem do tempo informado pelos participantes. Não é possível garantir que eles informaram esse tempo corretamente;
   - As perguntas definidas nos formulários da segunda etapa da avaliação são pontuais, ou seja, foi perguntado se um método específico é morto ou se uma classe específica possui métodos mortos. Essas questões foram definidas dessa forma para reduzir o esforço dos participantes na execução do estudo. Desse modo, não é possível garantir que, na prática, seriam obtidos resultados semelhantes, visto que na realidade seria necessária uma análise geral de código morto do sistema de software. Para contornar essa questão, seria necessário aplicar estudos mais elaborados, em que o participante teria tempo satisfatório para analisar inteiramente as versões do software;
   - Questões mal formuladas podem levar a diferentes interpretações, interferindo nos resultados obtidos no estudo experimental. Apesar de alguns pesquisadores terem revisado os formulários utilizados no estudo, não é possível garantir que os participantes tiveram a mesma interpretação das questões;
b) **Quanto à validade externa**. Uma ameaça à validade externa limita a capacidade de generalização dos resultados para contextos fora do ambiente avaliado:
   - Os participantes da segunda etapa da avaliação (alunos de graduação) podem não representar adequadamente a população de possíveis usuários reais da

abordagem, comprometendo a generalização dos resultados. A aplicação do estudo com diferentes tipos de profissionais pode amenizar essa ameaça;

- A quantidade limitada de questões definidas nos questionários pode não abordar todos os assuntos necessários para avaliação da abordagem;

c) **Quanto à validade de construção**. Uma ameaça à validade de construção está relacionada à validação do instrumento de pesquisa, definindo se o tratamento reflete adequadamente a causa e se os resultados refletem corretamente o efeito:

- Apesar da tentativa de evitar respostas dadas ao acaso considerando as justificativas das questões no momento da correção, não é possível garantir que todos participantes se comprometeram com o estudo e realizaram as atividades da maneira proposta;
- A alteração dos sistemas de software para condução do estudo sem e com o uso da abordagem pode não ter sido suficiente para evitar algum tipo de aprendizado entre as etapas;

d) **Quanto à validade de conclusão**. Uma ameaça à validade de conclusão afeta a habilidade de chegar a uma conclusão correta a respeito dos relacionamentos entre o tratamento e o resultado do experimento:

- Na primeira etapa da avaliação, a falta de informações a respeito da abordagem de detecção de código morto adotada em Understand pode dificultar a obtenção de informações conclusivas;
- A falta de testes estatísticos na primeira etapa da avaliação compromete a obtenção de resultados conclusivos a respeito do funcionamento da técnica Análise de Acessibilidade;
- Apesar da aplicação de testes estatísticos adequados para o tamanho da amostra, a quantidade limitada de participantes envolvidos na segunda etapa do estudo pode comprometer a obtenção de conclusões significativas. Para contornar essa questão, pode-se repetir o estudo experimental com maior quantidade de pessoas.

7.4 Considerações finais

Neste capítulo, foram apresentados os resultados obtidos com a avaliação da abordagem DCEVizz. Na primeira etapa da avaliação, foi verificada a capacidade de detecção de código morto da análise de acessibilidade implementada em DCEVizz Tool em relação a

ferramenta Understand. Os métodos mortos identificados por ambas as ferramentas foram comparados qualitativamente, a fim de verificar as características dos métodos identificados apenas por DCEVizz Tool ou apenas por Understand. Além disso, foi realizada análise da evolução do código morto utilizando as duas ferramentas e cinco versões mais recentes de cinco sistemas de software. Como resultado, pode-se perceber que ambas as ferramentas identificaram padrões evolutivos semelhantes, mostrando coerência no seu funcionamento.

A comparação dos resultados das ferramentas mostrou que os métodos identificados apenas por DCEVizz Tool eram realmente inacessíveis, sendo que a maioria deles possui a característica comum de sobrescreverem métodos de superclasses ou serem chamados apenas por métodos inacessíveis. Particularidades adotadas na implementação da técnica Análise de Acessibilidade levaram a esses resultados, por exemplo, o não tratamento de hierarquia de herança (métodos *override*, que podem ser chamados via polimorfismo) e a detecção de métodos chamados apenas por métodos inacessíveis. Além disso, essas particularidades fizeram com que alguns métodos mortos fossem identificados apenas por Understand, como os métodos relacionados a eventos de interface gráfica, construtores e tipos enumerados.

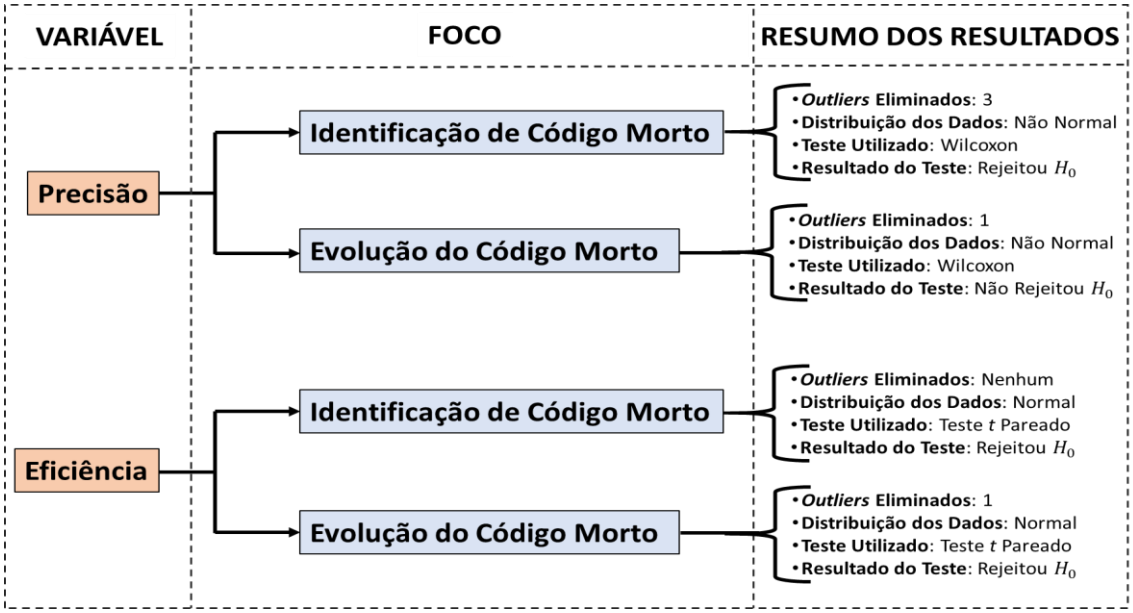
Na segunda etapa da avaliação, foram apresentados os resultados obtidos no estudo experimental executado para avaliar a abordagem DCEVizz. Esses resultados foram analisados de forma quantitativa e qualitativa. Na análise quantitativa, foram utilizados testes estatísticos para avaliação de hipóteses, no qual é afirmado, em H_0, que DCEVizz não influencia nas variáveis precisão/eficiência. Em H_1, essa afirmação é contestada, alegando que a abordagem aumenta os valores dessas variáveis. Na Figura 7.26, é apresentada uma síntese dos resultados e dos testes utilizados nesse estudo. Foram definidas uma hipótese nula e uma alternativa para cada variável em cada foco de análise, resultado em quatro H_0 e quatro H_1.

Como pode ser observado, as amostras da variável **precisão** não tiveram distribuição normal nos dois focos analisados, implicando no uso do *teste não paramétrico de Wilcoxon*. H_0 foi aceita para essa variável apenas no foco da evolução do código morto. Apesar da média com o uso de DCEVizz Tool ter sido maior, a eliminação do *outlier* e a quantidade de casos não influenciados pelo uso da abordagem fez com que não houvesse evidências estatísticas para rejeitar H_0. Em contrapartida, H_0 foi rejeitada no foco de identificação de código morto, evidenciando o efeito positivo da abordagem DCEVizz na variável precisão.

Em relação a variável **eficiência**, pode-se perceber que em ambos os focos a distribuição dos dados foi normal, implicando no uso do *Teste t Pareado* para avaliação das hipóteses. Além disso, H_0 foi rejeitada para os dois focos em análise, comprovando

estatisticamente que DCEVizz Tool aumentou a eficiência durante a execução de atividades para identificar e compreender a evolução do código morto.

Figura 7.26 - Síntese da Avaliação Quantitativa da Abordagem DCEVizz



Fonte: Do autor (2017).

Com a análise qualitativa, foi possível perceber que DCEVizz Tool facilitou a execução de algumas atividades. Essa conclusão foi obtida a partir das respostas do formulário de opinião e das escalas de Likert definidas para as questões. Além disso, o principal ponto negativo destacado por alguns participantes foi a dificuldade inicial em compreender a visualização qualitativa. Por fim, por meio das questões adicionais definidas no formulário da Etapa 2, foi possível perceber que a abordagem é capaz de prover uma visão geral da evolução do código morto. De acordo com a opinião dos participantes, seria muito difícil responder essas questões sem o uso de DCEVizz Tool.

8 CONSIDERAÇÕES FINAIS

Apesar de necessária, a evolução tem sido apontada como aspecto crítico em assegurar a manutebilidade dos sistemas de software. O aumento da quantidade de informações, das funções, da poluição e do código morto ao longo da evolução deixa esses sistemas mais complexos, dificultando sua compreensão. Desse modo, técnicas de visualização foram propostas para facilitar a compreensão da evolução de atributos específicos de sistema de software, por exemplo, arquitetura, acoplamento e modularização. No entanto, existe carência ao explorar a visualização para compreender atributos que contribuam com a degradação desses sistemas ao longo da evolução, estimulando a tomada de medidas que permitem melhorar a qualidade do software.

Com base nesses fatores, neste trabalho, foi proposta DCEVizz, uma abordagem que possibilita a visualização da evolução de métodos mortos (código morto) em sistemas de software orientados a objetos. Essa abordagem utiliza a técnica Análise de Acessibilidade para identificar métodos mortos, identifica suas características evolutivas e os apresenta utilizando técnicas de visualização. De modo geral, DCEVizz foi proposta com o objetivo de prover melhor percepção das caracerísitcas evolutivas do código morto, motivando a execução de medidas para reduzir a relação existente entre evolução e aumento da poluição de sistema de software.

8.1 Conclusões

A abordagem DCEVizz foi avaliada em duas fases utilizando o *plug-in* DCEVizz Tool. Na primeira fase, o objetivo foi verificar se a técnica Análise de Acessibilidade implementada em DCEVizz Tool possui boa capacidade de detecção de código morto, quando comparada com a técnica da ferramenta Understand. Na segunda fase, o objetivo foi verificar se a abordagem DCEVizz melhora a percepção da existência de código morto nas versões, assim como suas características evolutivas.

Foi realizada na primeira fase uma comparação qualitativa do código morto identificado pelas duas ferramentas. Com essa comparação, pode-se concluir que DCEVizz Tool foi capaz de identificar maior quantidade de código morto do que Understand nos sistemas de software analisados, mostrando boa capacidade de detecção da técnica Análise de Acessibilidade. Foi possível concluir também que as principais diferenças obtidas ocorreram por causa das características de implementação adotadas em cada ferramenta.

Na segunda fase, foi executado um estudo experimental com um grupo de voluntários, que realizaram um conjunto de atividades **sem** e **com** o uso de DCEVizz Tool. A análise quantitativa dos resultados mostrou que o uso da abordagem resultou no aumento da média das duas variáveis utilizadas na avaliação (precisão e eficiência), para os dois focos de análise (identificação e compreensão da evolução de código morto).

Os testes de hipóteses comprovaram os efeitos positivos de DCEVizz Tool nas duas variáveis, exceto para a precisão com foco na evolução do código morto. Apesar da média dessa variável com o uso do *plug-in* ter sido maior nesse foco, não houve evidências estatisticamente significantes para aceitar a hipótese alternativa. Além disso, a análise qualitativa dos resultados mostrou que, na opinião dos participantes, DCEVizz Tool facilitou a execução das atividades.

Com base nesses resultados, pode-se perceber que a utilização da técnica Análise de Acessibilidade mostrou boa capacidade de detecção de métodos mortos e que as técnicas de visualização implementadas em DCEVizz Tool facilitaram a execução das tarefas, afetando positivamente as variáveis precisão e eficiência. Dessa forma, foi possível concluir que existem indícios de que a abordagem DCEVizz provê melhor percepção da existência de código morto e das suas características evolutivas, facilitando sua compreensão.

8.2 Contribuições

As contribuições deste trabalho são:

a) Revisão Sistemática da Literatura para identificar técnicas de detecção de código morto e seu domínio de aplicação, assim como as técnicas indicadas para detectar os variados tipos de código morto;
b) Definição da abordagem DCEVizz, que identifica e representa visualmente a evolução do código morto em sistemas de software orientados a objetos, com a finalidade de prover melhor percepção das suas características evolutivas ao longo das versões;
c) Desenvolvimento do *plug-in* para a plataforma IDE Eclipse (DCEVizz Tool), que automatiza a execução da abordagem proposta;
d) Definição de uma técnica de visualização qualitativa baseada em técnicas existentes na literatura. Essa técnica representa visualmente a evolução do código morto presente em versões de sistemas de software;

e) Implementação, adaptação e avaliação qualitativa de uma técnica de detecção de código morto utilizada em diferentes estudos, destacando seu potencial para identificar código morto em relação a uma ferramenta amplamente utilizada no mercado.

8.3 Limitações

As principais limitações da abordagem DCEVizz são:

a) Utilização da técnica Análise de Acessibilidade na abordagem DCEVizz não leva em consideração questões de herança e de sobrescrita de métodos. A acessibilidade de um método é verificada independentemente de sua característica na hierarquia de herança. Dessa forma, chamadas de métodos realizadas via polimorfismo podem não ser identificadas pela técnica, classificando tais métodos como inacessíveis;
b) Utilização de análise estática para identificar código morto. Apesar de mais flexível que análise dinâmica, esse tipo de análise pode gerar alguns resultados inconsistentes. Por exemplo, métodos chamados durante a execução do sistema via reflexão podem ser detectados indevidamente como inacessíveis, por não possuírem chamadas explícitas de outros métodos. Além disso, a técnica Análise de Acessibilidade não identifica chamadas desencadeadas a partir de eventos gerados durante a execução do sistema de software;
c) A análise da evolução dos métodos mortos e da existência de determinados pacotes e classes ao longo das versões é inteiramente baseada em sua assinatura e em seu nome. A abordagem DCEVizz não analisa o conteúdo e não identifica renomeação desses componentes entre as versões. Dessa forma, se um método morto for renomeado de uma versão para outra, na visualização qualitativa constará que tal método foi removido da versão anterior e um método morto foi adicionado na próxima versão. O mesmo ocorre para pacotes e classes caso sejam renomeados;
d) As técnicas de visualização de DCEVizz podem ter problemas de escalabilidade caso sejam analisadas muitas versões ou exista grande quantidade de código morto no sistema de software. Na visualização quantitativa, o eixo x que contém as versões pode ficar “sobrecarregado” e comprometer a visibilidade dessas versões. Na visualização qualitativa, o limite da tela pode não ser suficiente para representar as informações, sendo necessário o uso das barras de rolagem. Para reduzir esse problema, foram desenvolvidos filtros que permitem a visualização apenas dos pacotes

de interesse, eliminando a necessidade das barras de rolagem horizontais. Entretanto, se muitas versões estiverem em análise, ainda haverá a necessidade da barra de rolagem vertical.

8.4 Perspectivas futuras

Algumas sugestões de trabalhos futuros interessantes que podem contribuir no refinamento e na avaliação da abordagem DCEVizz são:

a) Refinar a implementação da técnica Análise de Acessibilidade para tratamento de questões relacionadas a hierarquia de herança, permitindo identificar chamadas via polimorfismo e melhorar a precisão dos resultados;
b) Refinar os estudos em relação aos aspectos dinâmicos das linguagens orientadas a objetos, de forma a encontrar maneiras de identificar ou prever chamadas de métodos realizadas em tempo de execução;
c) Estender a análise da evolução do código morto realizada na primeira parte da avaliação deste estudo, executando DCEVizz Tool com maior quantidade de versões de sistemas de software. A análise detalhada dessas informações pode ser útil para identificar padrões evolutivos do código morto em comum nos diferentes sistemas de software, permitindo entender a origem desse problema;
d) Estender o estudo experimental realizado na segunda parte da avaliação, aumentando a quantidade de pessoas envolvidas, incluindo profissionais de mercado e professores;
e) Executar um estudo de observação em empresas que realizam versionamento de seus sistemas de software. O uso de DCEVizz nessas empresas permitiria identificar problemas reais durante a identificação de código morto, dificuldades de interpretação das técnicas de visualização e necessidade de definição de novas técnicas para a abordagem. Além disso, seria possível observar se DCEVizz é útil para prover melhor percepção das características evolutivas do código morto e a real utilidade dessas informações para as empresas.

8.5 Publicações

Os trabalhos publicados ao longo deste estudo são:

a) BASTOS, C; COSTA H. Uma abordagem para Visualização da Evolução de Código Morto em Sistemas de Software Orientados a Objetos. In: SIMPÓSIO BRASILEIRO DE QUALIDADE DE SOFTWARE, n°16, Outubro de 2016, Maceió. **Anais...** p. 33-38. 2016.

b) BASTOS, C; JÚNIOR, P. A. P; COSTA, H. Aplicação da Técnica Análise de Acessibilidade para Detecção de Métodos Mortos em Sistemas de Software Orientados a Objetos. In: SIMPÓSIO BRASILEIRO DE QUALIDADE DE SOFTWARE, n°16, Outubro de 2016, Maceió. **Anais...**. p. 79-93. 2016.

c) BASTOS, C; JÚNIOR, P. A. P; COSTA, H. DCEVizz: Uma ferramenta para Visualização de Código Morto na Evolução de Sistemas de Software Java. In: CONGRESSO BRASILEIRO DE SOFTWARE, n°7, Setembro de 2016, Maringá. **Anais...** p. 1-8. 2016.

d) BASTOS, C; JÚNIOR, P. A. P; COSTA, H. Técnicas para Detecção de Código Morto: Uma Revisão Sistemática de Literatura. In: SIMPÓSIO BRASILEIRO DE SISTEMAS DE INFORMAÇÃO, Maio de 2016, Florianópolis. **Anais...** p. 255-262. 2016.

e) CRUZ, A. P. S; BASTOS, C; JÚNIOR, P. A. P; COSTA, H. Software Visualization Tools and Techniques: A Systematic Review of the Literature. In. INTERNATIONAL CONFERENCE OF THE CHILEAN COMPUTER SCIENCE SOCIETY. **Anais...** p. 236-247. 2016.

REFERÊNCIAS

ASH, D. et al. Using Software Maintainability Models to Track Code Health. In: INTERNACIONAL CONFERENCE ON SOFTWARE MAINTENANCE, Setembro de 1994, Victoria, BC. **Proceedings...** IEEE, 1994. p. 154-160.

BACON, D. F.; LAFFRA, J. C.; SWEENEY, P. F.; TIP, F. **Removal of unreachable methods in object-oriented applications based on program interface analysis**. Patente Número US6654951 B1, 2003.

BARBETTA, P. A.; REIS, M. M.; BORNIA, A. C. **Estatística para Cursos de Engenharia e Informática**. 3. ed. 409p. Ed. Atlas, 2010.

BASILI, V.; CALDIERA, G.; ROMBACH, H. Goal question metric paradigm. **Encyclopedia of Software Engineering**, v. 1, p. 528-532, 1994.

BASTOS, C; COSTA H. Uma abordagem para Visualização da Evolução de Código Morto em Sistemas de Software Orientados a Objetos. In: SIMPÓSIO BRASILEIRO DE QUALIDADE DE SOFTWARE, n°16, Outubro de 2016, Maceió. **Anais...** p. 33-38. 2016. a.

BASTOS, C; JÚNIOR, P. A. P; COSTA, H. Aplicação da Técnica Análise de Acessibilidade para Detecção de Métodos Mortos em Sistemas de Software Orientados a Objetos. In: SIMPÓSIO BRASILEIRO DE QUALIDADE DE SOFTWARE, n°16, Outubro de 2016, Maceió. **Anais...**. p. 79-93. 2016. b.

BASTOS, C; JÚNIOR, P. A. P; COSTA, H. DCEVizz: Uma ferramenta para Visualização de Código Morto na Evolução de Sistemas de Software Java. In: CONGRESSO BRASILEIRO DE SOFTWARE, n°7, Setembro de 2016, Maringá. **Anais...** p. 1-8. 2016. c.

BASTOS, C; JÚNIOR, P. A. P; COSTA, H. Técnicas para Detecção de Código Morto: Uma Revisão Sistemática de Literatura. In: SIMPÓSIO BRASILEIRO DE SISTEMAS DE INFORMAÇÃO, Maio de 2016, Florianópolis. **Anais...** p. 255-262. 2016. d.

BIOLCHINI, J. C. A. et al. Scientific Research Ontology to Support Systematic Review in Software Engineering. **Advanced Engineering Informatics**. Abril de 2007. v.2, p. 133-151. 2007.

BOOMSMA, H.; GROSS, H.G. Dead Code Elimination for Web Systems Written in PHP: Lessons Learned from an Industry Case. In. INTERNACIONAL CONFERENCE ON SOFTWARE MAINTENANCE, Setembro de 2012, Trento, IT. **Proceedings...** IEEE, 2012, p. 511-515.

BURD, L.; RANK, S. Using Automated Source Code Analysis for Software Evolution. In. WORKSHOP ON SOURCE CODE ANALYSIS AND MANIPULATION, Setembro de 2001, **Proceedings...** IEEE, 2001, p. 204-210

CARD, S. K.; MACKINLAY, J. D.; SHNEIDERMAN, B. **Readings in information visualization**: using vision to think. 671p. Ed. Morgan Kaufmann, 1999.

CARNEIRO, G. F. **SourceMiner**: Um Ambiente Integrado Para Visualização Multi-Perspectiva de Software. 2013. 213p. Tese de Doutorado - Universidade Federal da Bahia, Bahia, 2013.

CARPENDALE, S.; GHANAM, Y. **A Survey Paper on Software Architecture Visualization**. Calgary, University of Calgary, 2008.

CASERTA, P.; ZENDRA, O. Visualization of the Static Aspects of Software: A Survey. In. IEEE TRANSACTIONS ON VISUALIZATION AND COMPUTER GRAPHICS, Dezembro de 2011. **Proceedings**... IEEE, 2012, p. 913-933.

CHAIKALIS, T.; CHATZIGEORGIOU, A. Forecasting Java Software Evolution Trends employing Network Models. In. IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, Julho de 2015. **Proceedings...** IEEE, 2015, v. 41, p. 582-602.

CHEN, Y.; EMDEN, R. G.; ELEFTHERIOS, K. A C++ Data Model Supporting Reachability Analysis and Dead Code Detection. In. IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, Janeiro de 1998. **Proceedings...** IEEE, 1998, v. 24, n. 9, p. 682-694.

CORNELISSEN, B.; ZAIDMAN, A.; van DEURSEN, A. A controlled experiment for program comprehension through trace visualization. In. . IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, Abril de 2011. **Proceedings...** IEEE, 2011, v. 37, p. 341-355.

D'AMBROS, M. Supporting software evolution analysis with historical dependencies and defect information. In. INTENATIONAL CONFERENCE ON SOFTWARE MAINTENANCE, Outubro de 2008, Beijing, China. **Proceedings...** IEEE, 2008, p. 412-415.

DI LUCCA, G. A.; DI PENTA, M. Experimental settings in program comprehension: Challenges and open issues. In. INTERNATIONAL CONFERENCE ON PROGRAM COMPREHENSION, n°14, Abril de 2006, Athens, Greece. **Proceedings...** IEEE, 2006, p. 229-234.

DIEHL, S. **Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software**. 187p. Ed. Springer Science & Business Media, 2007.

DUCASSE, S.; LANZA, M. The Class Blueprint: Visually Supporting the Understanding of Glasses. In. IEEE TRANSACTION ON SOFTWARE ENGINEERING, Fevereiro de 2005. **Proceedings...** IEEE, 2005, v.31, n.1, p. 75-90.

ECLIPSE, IDE. Disponível em: <http://www.eclipse.org>. Acesso em: Setembro 2015.

EDER, S. et al. How Much Does Unused Code Matter for Maintenance?. In. INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, Abril de 2012. **Proceedings...** IEEE, 2012, v.38, p. 1102-1111.

EICK, S. G.; STEFFEN, J. L.; SUMMER, E. E. Seesoft - A Tool for Visualizing Line Oriented Software Statistics. In. EEE TRANSACTION ON SOFTWARE ENGINEERING, Fevereiro de 1992. **Proceedings...** IEEE, 1992, v. 18, n. 11, p. 957-968

FEHNKER, A.; HUUCK, R. Model checking driven static analysis for the real world: designing and tuning large scale bug detection. **Innovations in Systems and Software Engineering**, Londres, v. 9. p. 45-56, 2013.

FRANCESE, R.; RISI, M.; SCANNIELLO, G. Enhancing Software Visualization with Information Retrieval. In. INTERNATION CONFERENCE ON INFORMATION VISUALIZATION, n°19, Julho de 2015, Barcelona. **Proceddings...** IEEE, 2015, p. 189-194.

FRANCESE, R. et al. Viewing object-oriented software with metricattitude: An empirical evaluation. In. INTERNATION CONFERENCE ON INFORMATION VISUALIZATION, n°18, Julho de 2014, Paris. **Proceddings...** IEEE, 2014, p. 59-64.

GILBERT, D. **JFreeChart**. Disponível em < http://www.jfree.org/>. Acesso em: Setembro 2015.

GILBERT, D. **The JFreeChart Class Library**. 159p. Object Refinery. 2009.

GOLD, N.; MOHAN, A. A Framework for Understanding Conceptual Changes in Evolving Source Code. In. INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE, Setembro de 2003, Amsterdam. **Proceddings...** IEEE, 2003, p. 431-439.

GUERROUAT, A.; RICHTER, H. A Combined Approach for Reachability Analysis. In. INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING ADVANCES, Novembro de 2006, Tahiti. **Proceedings...** IEEE, 2006, p. 23-23.

HENDRIX, D.; CROSS, J. H.; MAGHSOODLOO, S. The effectiveness of control structure diagrams in source code comprehension activities. In. IEEE TRANSACTION ON SOFTWARE ENGINEERING, Junho de 2002. **Proceedings...** IEEE, 2002, v. 28, p. 463-477. 2002.

INSELBERG, A.; REIF, M.; CHOMUT, T. Convexity Algorithms in Parallel Coordinates. **Journal of the ACM**, New York, v. 34, n. 4, p. 765-801. 1987.

JUNG, C. F. **Metodologia Aplicada a Projetos de Pesquisa: Sistemas de Informação & Ciência da Computação**. Ed. Taquara., 2009. Disponível em: < http://www.jung.pro.br/moodle/mod/resource/view.php?id=102 >. Acesso em: Setembro de 2015.

KEIM, D. Information Visualization and Visual Data Mining. In. IEEE TRANSACTION ON VISUALIZATION AND COMPUTER GRAPHICS, Abril de 2002. **Proceedings...** IEEE, 2002, v. 8, n. 1, p. 1-8.

KEIM, D.; KRIEGEL, H. P. Visualization Techniques for Mining Large Databases: A Comparison. In. IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING, Abril de 1996. **Proceedings...** IEEE, 1996, v. 8, n. 6, p. 923-938

KIENLE, H. M.; MULLER, H. Requirements of Software Visualization Tools: A Literature Survey. In. WORKING CONFERENCE ON SOFTWARE VISUALIZATION, Maio de 2007, Minneapolis. **Proceddings...** IEEE, 2007, p. 2-9.

KNOOP, J. Eliminating Partially Dead Code in Explicitly Parallel Programs. **Theoretical Computer Science,** v. 196, n. 1, p. 365-393.1998.

KNOOP, J.; RUTHING, O.; STEFFEN B. Partial Dead Code Elimination. In. CONFERENCE LANGUAGE ON PROGRAMMING LANGUAGE DESIGN AND IMPLEMENTATION, Junho de 1994, Orlando. **Proceedings...** ACM Sigplan Notices, 1994, v. 29, p. 147-158.

LANZA, M. The Evolution Matrix: Recovering Software Evolution Using Software Visualization Techniques. In. INTERNATIONAL WORKSHOP ON PRINCIPLES OF SOFTWARE EVOLUTION, Maio de 2001, Orlando. **Proceedings...** ACM, 2001, p. 37-42.

LEHMAN, M. M.; BELADY, L. **Program Evolution: Processes of Software Change**. 538p. Ed. Academic Press, 1985.

LIU, H. Discussion on the Statistical Analysis Method. In. INTERNATION JOINT CONFERENCE ON COMPUTATIONAL SCIENCES AND OPTIMIZATION, Julho de 2014, Beijing. **Proceedings...** p. 383-385, 2014.

MALETIC, J.; MARCUS, A.; COLLARD, M. A Task Oriented View of Software Visualization. In. INTERNATION WORKSHOP ON VISUALIZING SOFTWARE FOR UNDERSTANDING AND ANALYSIS, Junho de 2002, França. **Proceedings...** p. 32-40. 2002.

MARTIN, R. C. **Clean Code: A Handbook of Agile Software Craftsmanship**. 431p. Ed. Prentice Hall, 2008.

MARTINS, R. C.; PELLEGRINO, S. R. M.; SANTELLANO, J. Tratamento de "Dead Codes" em Software de uso Aeronáutico. In. BRAZILIAN CONFERENCE ON DYNAMICS, CONTROL AND THEIR APPLICATIONS, Junho de 2010, Serra Negra. **Anais...** p. 1-8, 2010.

MAYRHAUSER, A. V.; VANS, M. Program Comprehension During Software Maintenance and Evolution. **Computer**, v. 28, n. 8, p. 44-55. 1995.

MINELLI, R.; LANZA, M. SAMOA - Visual Software Analytics Platform for Mobile Applications. In. INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE, Janeiro de 2013, Eindhoven. **Proceedings...** IEEE, 2013, p. 476-479.

MURPHY, G. C.; KERSTEN, M.; FINDLATER, L. How are Java software developers using the Elipse IDE?. In. IEEE SOFTWARE, Abril de 2006. **Proceedings...** IEEE, 2006, v. 23, n. 4, p. 76-83.

NOVAIS, R. L. et al. Sourceminer Evolution: A Tool for Supporting Feature Evolution Comprehension. In. INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE, Janeiro de 2013, Eindhoven. **Proceedings...** IEEE, 2013. pp. 508-511.a.

NOVAIS, R. L. et al. Software Evolution Visualization: A Systematic Mapping Study. **Information and Software Technology**, p.1860-1883. 2013. b.

NOVAIS, R.; SIMÕES, P. R. M.; MENDONÇA, M. TimeLine Matrix: An on Demand View for Software Evolution Analysis. In. WORKSHOP BRASILEIRO DE VISUALIZAÇÃO DE SOFTWARE, Setembro de 2012, Natal. **Anais...** pp. 1-8. 2012.

O'BRIEN, M. P. **Software Comprehension**: A Review & Research Direction. 2003. Relatório Técnico - Department of Computer Science & Information Systems. University of Limerick, Ireland, 2003.

OUDSHOORN, M. J., WIDJAJA, H., ELLERSHAW, S. K. Aspects and Taxonomy of Program Visualisation. **Software Visualisation**, v. 7, p. 3-26. 1996.

PETRE, M., QUINCEY, E. A Gentle Overview of Software Visualisation. **PPIG News Letter**, p. 1-10. 2006.

PFLEEGER, S. L.; ATLEE, J. M. **Software Engineering**: Theory and Practice. 792p. Ed. Prentice Hall, 2009.

PRESSMAN, R.; MAXIM, B. **Software Engineering: A Practitioner's Approach**. 976p. Ed. McGraw-Hill, 2014.

RAJLICH, V.; WILDE, N. The Role of Concepts in Program Comprehension. In. INTERNATIONAL WORKSHOP ON PROGRAM COMPREHENSION, Junho de 2002, Paris. **Proceedings....** IEEE, 2002, p. 271-278.

RATZINGER, J.; Gall, H.; PINZGER, M. Quality assessment based on attribute series of software evolution. In. WORKING CONFERENCE ON REVERSE ENGINEERING, Outubro de 2007, Vancouver. **Proceedings...** IEEE, 2007, p. 80-89.

ROMANO, S.; SCANNIELLO, G. DUM-Tool. In. INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE AND EVOLUTION, Outubro de 2015, Alemanha. **Proceedings...** IEEE, 2015, p. 339-341.

ROMANO, S. et al. A graph-based approach to detect unreachable methods in Java software. In: ACM SYMPOSIUM ON APPLIED COMPUTING, Abril de 2016, Pisa. **Proceedings...** ACM, 2016, p. 1538-1541.

RUFIANGE, S.; MELANCON, G. AniMatrix: A Matrix-Based Visualization of Software Evolution. In. WORKING CONFERENCE ON SOFTWARE VISUALIZATION, Setembro de 2014, Canadá. **Proceedings...** IEEE, 2014, p. 137-146.

SCANNIELLO, G. An Investigation of Object-Oriented and Code-Size Metrics as Dead Code Predictors. In. CONFERENCE ON SOFTWARE ENGINEERING AND ADVANCED APPLICATIONS, Agosto de 2014, Verona. **Proceedings....** IEEE, 2014, p. 392-397.

SCANNIELLO, G. Source Code Survival with the Kaplan Meier. In. INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE, Setembro de 2015, Virigínia. **Proceedings...** IEEE, 2011, p. 524-527.

SCHOTS, M. **PREViA**: Uma Abordagem para a Visualização da Evolução de Modelos de Software. 2011. 202p. Dissertação de Mestrado - Universidade Federal do Rio de Janeiro, Rio de Janeiro, 2011.

SHARAFI, Z. A Systematic Analysis of Software Architecture Visualization Techniques. In. INTERNATION CONFERENCE ON PROGRAM COMPREHENSION, Junho de 2011, Canadá. **Proceedings...** IEEE, 2011, p. 254-257.

SOMMERVILLE, I. **Software Engineering**. 792p. Ed. Addison-Wesley. 2010.

SRIVASTAVA, A. Unreachable procedures in object-oriented programming. **ACM Letters on Programming Languages and Systems**, p. 355-364. 1992.

SUNITHA, K. V. N.; KUMAR, V. V. A New Technique for Copy Propagation and Dead Code Elimination Using Hash Based Value Numbering. In. INTERNATIONAL CONFERENCE ON ADVANCED COMPUTING AND COMMUNICATIONS, Outubro de 2006, Índia. **Proceedings...** IEEE, 2006, p. 601-604.

SWING. **Swing (Java™ Foundation Classes)**. Disponível em: <http://docs.oracle.com/javase/7/docs/technotes/guides/swing/>. Acesso em: Setembro 2015.

TELEA, A.; AUBER, D. Code Flows: Visualizing Structural Evolution of Source Code. **Computer Graphics Forum**, v. 27, p. 831-838. 2008.

TJORTJIS, C.; LAYZELL, P. Expert maintainers' strategies and needs when understanding software: a case study approach. In. ASIA-PACIFIC SOFTWARE ENGINEERING CONFERENCE, Agosto de 2001, Macao. **Proceedings...** IEEE, 2001, p. 281-287.

TRAVASSOS, G. H.; GUROV, D.; AMARAL, E. A. G. **Introdução à Engenharia de Software Experimental**. 2002. Relatório Técnico - PESC-COPPE, Universidade Federal do Rio de Janeiro, Rio de Janeiro, 2002.

W3C, **XML Schema**. Disponível em: <http://www.w3.org/XML/Schema>. Acesso em: Abril 2015.

YANG, H., GRAHAM, H. **Software Metrics and Visualisation**. 2003. Relatório de Projeto - Universidade of Auckland, Auckland, 2003.

APÊNDICE A - PROTOCOLO E ANÁLISES DA RSL

Neste Apêndice, há a descrição das características estabelecidas na etapa Planejamento e Execução da RSL, realizada para identificar técnicas de detecção de código morto. Além disso, são descritos resultados obtidos com as meta-análises.

O restante desse Apêndice está organizado da seguinte forma. O protocolo com a descrição dos repositórios e da *string* de busca utilizada é exibido na Seção A.1. Os passos executados pelos pesquisadores para coleta e seleção de artigos, assim como a quantidade de artigos obtidos em cada repositório, são descritos na Seção A.2. As meta-análises dos dados coletados a partir da RSL, que investigam aspectos importantes em relação a código morto, por exemplo, autores e conferências mais relevantes são apresentadas na Seção A.3.

A.1. Planejamento

Na fase **Planejamento**, é elaborado um protocolo que contém a metodologia utilizada na condução da RSL (BIOLCHINI et al., 2007). Foi elaborada uma questão central para ser utilizada na seleção dos trabalhos relevantes:

Quais são as técnicas de detecção de código morto existentes na literatura?

Além da definição da questão de pesquisa, foi elaborada uma *string* de busca que auxilia na padronização da pesquisa em diferentes repositórios de trabalhos científicos. Essa *string* é composta por palavras-chave e palavras que especificam o assunto procurado nos trabalhos:

("Dead Code" **OR** "Unnecessary Code" **OR** "Unused Code" **OR** "Dead Source Code" **OR** "Inaccessible Code" **OR** "Unapproachable Code" **OR** "Unreachable Code")

AND

(Detection **OR** Detecting **OR** Removing **OR** Remotion OR Elimination **OR** Eliminating)

AND

(Technique **OR** Algorithm **OR** Strategy)

A *string* foi utilizada na máquina de busca de alguns repositórios de trabalhos científicos para coletar trabalhos relacionados ao assunto investigado. Nomes e endereços eletrônicos dos repositórios utilizados são listados na Tabela A.1. Para serem selecionados, os artigos deveriam estar de acordo com os critérios de seleção definidos: i) possuir acesso livre ao seu conteúdo; e ii) ser publicado a partir do ano de 1990.

Tabela A.1 - Repositórios Utilizados

Nome do Repositório	Endereço Eletrônico
ACM Digital Library	http://www.acm.org
Enginnering Village	http://www.engineeringvillage.com
IEEE Xplore Digital Library	http://ieeexplore.ieee.org
Science Direct	www.sciencedirect.com
Scopus	http://www.scopus.com
Springer Link	http://link.springer.com

Fonte: Do autor (2017).

A.2. Execução

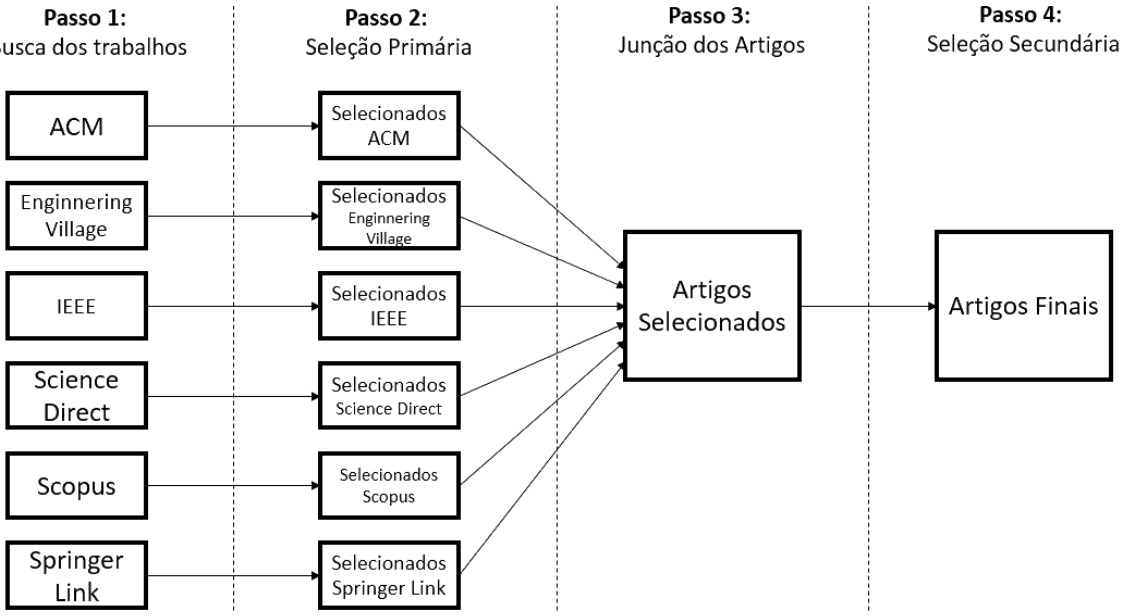
A fase **Execução** envolve a identificação, a seleção e a avaliação dos estudos de acordo com os critérios definidos no protocolo (BIOLCJINI et al., 2007). Essa fase foi conduzida com a execução dos seguintes passos (FIGURA A.1):

a) **Passo 1**: A string de busca foi utilizada nos repositórios para coleta de publicações, as quais foram analisadas para retirar as que não são artigos;
b) **Passo 2**: Em seguida, foi realizada a seleção primária com a leitura do título (title), do resumo (abstract) e das palavras-chave (keywords) para seleção de artigos relevantes;
c) **Passo 3**: Os artigos de cada repositório resultantes do Passo 2 foram reunidos para eliminar os artigos duplicados;
d) **Passo 4**: Foi realizada a seleção secundária. Os artigos resultantes do Passo 3 foram analisados com a leitura do resumo e das conclusões.

Por causa das diferenças existentes na ferramenta de busca dos repositórios, houve necessidade de utilizar diferentes filtros em cada uma delas (TABELA A.2). A busca no repositório da IEEE retornou artigos desde 1967, considerados irrelevantes neste estudo por causa do critério inicial anteriormente mencionado (artigos devem ter sido publicados a partir do ano de 1990); assim, foi utilizado o filtro de ano para limitar a busca. O mesmo tipo de filtro foi utilizado no repositório da ScienceDirect. Nos demais repositórios, não houve necessidade da filtragem por ano, visto que não foram retornados trabalhos publicados antes de 1990. No repositório da ACM, não foram utilizados filtros, pois não foi possível aplicá-los

juntamente com o recurso de busca avançada. Nas bases que retornam livros, foi aplicado o filtro de tipos de publicação. Além disso, foram realizadas filtragens por área, selecionando apenas a área Ciência da Computação.

Figura A.1 - Condução da Execução da RSL



Fonte: Do autor (2017).

Tabela A.2 - Filtros Utilizados nos Repositórios

Fontes	Filtros
ACM	Nenhum
Enginnering Village	Nenhum
IEEE	Ano (1990-2015) / Conference e Journals
Science Direct	Ano (1990-2015) / Journals - Área Ciência da Computação
Scopus	Nenhum
Springer	Artigos/Área da Ciência da Computação

Fonte: Do autor (2017).

A quantidade de artigos resultantes dos passos realizados é apresentada na Tabela A.3. Na segunda coluna, é apresentada a quantidade de artigos retornados em cada repositório após a realização do Passo 1. Na terceira coluna, é apresentada a quantidade de artigos após a seleção primária. Na quarta, na quinta e na sexta colunas, são apresentados os resultados obtidos com a seleção secundária. Os resultados finais são apresentados na última coluna, ou seja, a quantidade de artigos selecionados para análise (estudos primários). As referências dos artigos selecionados estão apresentadas no Apêndice B.

Foram encontrados 3.820 trabalhos, sendo 41,28% do repositório IEEE Xplore, 31,83% do repositório ACM, 2,07% no repositório Enginnering Village, 8,56% do repositório

Springer Link, 8,66% do repositório Science Direct e 7,59% no repositório Scopus. Após a Seleção Primária:

a) No repositório do ACM, 16 trabalhos foram selecionados (1,31% do total), sendo 3 irrelevantes (18,75%), 2 repetidos (12,50%) e 1 incompleto (6,25%) - **Resultou em 10 artigos (62,50%)**;
b) No repositório Enginnering Village, 17 trabalhos (20,25% do total) foram selecionados, sendo 4 irrelevantes (23,53%), 4 repetidos (23,53%) e 2 incompletos (11,76%) - **Resultou em 7 artigos (41,18%)**;
c) No repositório do IEEE Xplore, 44 trabalhos (2,79% do total) foram selecionados, sendo 20 irrelevantes (45,45%), 7 repetidos (15,91%) e 1 incompleto (2,27%) - **Resultou em 16 artigos (36,36%)**;
d) No repositório do Science Direct, 14 trabalhos (4,23% do total) foram selecionados, sendo 4 irrelevantes (28,57%), 2 repetidos (14,29%) e 0 incompleto (0%) - **Resultou em 8 artigos (57,14%)**;
e) No repositório do Scopus, 62 trabalhos (21,38% do total) foram selecionados, sendo 16 irrelevantes (25,81%), 25 repetidos (40,32%) e 10 incompleto (10,13%) - **Resultou em 11 artigos (17,74%)**;
f) No repositório do Springer Link, 3 trabalhos (0,92% do total) foram selecionados, sendo 2 irrelevantes (66,67%), 0 repetidos (0%) e 0 incompleto (0%) - **Resultou 1 em artigos (33,33%)**.

Tabela A.3 - Número de Artigos Obtidos com a RSL

Fontes	Quantidade Inicial de Resultados	Seleção Primária (Títulos, Resumos e Palavras-chave)	Seleção Secundária (Resumos e Conclusões)			
			Irrelevantes	Repetidos	Incompletos	Resultados dos Estudos Primários
ACM	1.216	16	3	2	1	10
Enginnering Village	79	17	4	4	2	7
IEEE Xplore	1.577	44	20	7	1	16
Science Direct	331	14	4	2	0	8
Scopus	290	62	16	25	10	11
Springer Link	327	3	2	0	0	1
Total	**3.820**	**156**	**49**	**39**	**14**	**53**

Fonte: Do autor (2017).

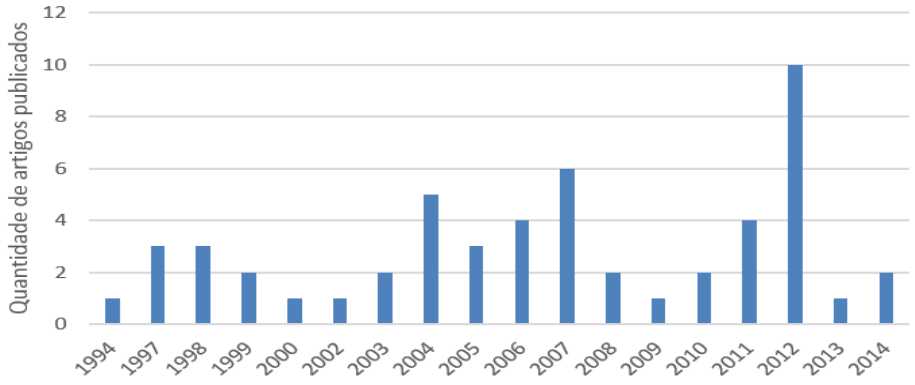
Ao final, **53 artigos** foram selecionados, sendo 10 artigos no ACM (18,87%), 7 artigos no Enginnering Village (13,21%), 16 artigos no IEEE Xplore (30,19%), 8 artigos no Science Direct (15,09%), 11 artigos no Scopus (20,75%) e 1 artigo no Springer Link (1,89%).

A.3. Meta-Análises

As meta-análises são úteis para investigar aspectos relevantes a respeito do assunto estudado. Algumas dessas análises foram realizadas para investigar quais são os anos em que mais artigos sobre código morto foram publicados, quais são os artigos mais citados, autores mais influentes na área e os eventos mais importantes.

Na análise de anos de publicação, o objetivo é apresentar a quantidade de artigos relacionados com o assunto detecção de código morto publicados ao longo dos anos (FIGURA A.2). Com isso, é possível visualizar se existe tendência de aumento ou de diminuição da quantidade de trabalhos publicados que abordam esse assunto. Como pode ser observado, a quantidade de artigos publicados não seguiu uma tendência em relação ao ano de publicação, pois houve oscilação na quantidade de artigos nos anos. O ano em que mais artigos foram publicados foi 2012, com 10 artigos. Além disso, nota-se que código morto é um assunto que está continuamente sendo estudado.

Figura A.2 - Quantidade de Publicações por Ano



Fonte: Do autor (2017).

Na análise de citações entre os artigos obtidos na RSL, foi verificado quais artigos selecionados que referenciam outros artigos também selecionados. Para isso, foi contabilizada a quantidade de vezes em que esses artigos foram citados (TABELA A.4). O artigo `Partial Dead Code Elimination` ([A21]) foi o mais citado com 11 citações, sendo, também, o mais antigo (ano de 1994).

Para identificação de possíveis autores influentes na área, foram analisados os autores dos artigos obtidos na RSL. Foram identificados 117 autores distintos que publicaram artigos relacionados com código morto, o que mostra razoável interesse dos pesquisadores sobre o assunto. Na Figura A.3, são apresentados os autores que publicaram mais de um artigo

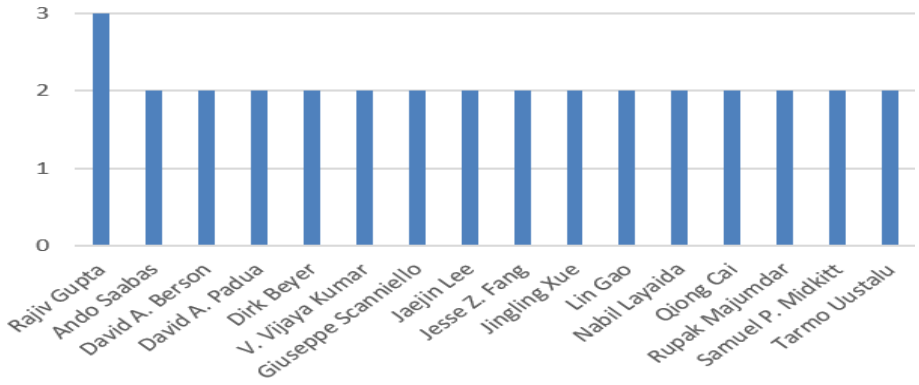
encontrado na RSL. O autor que mais contribuiu foi `R. Gupta` com 3 artigos. Apesar da quantidade significativa de pesquisadores encontrados, poucos publicaram mais de um artigo relacionado ao tema. Praticamente, metade dos estudos encontrados (51%) não tinham como objetivo principal a detecção de código morto, podendo ser uma justificativa para os resultados apresentados. Possivelmente, publicações futuras dos autores desses trabalhos estariam relacionadas com outras otimizações e não especificamente com detecção de código morto.

Tabela A.4 - Artigos Obtidos com a RSL

Título	Ano de Publicação	Código de Identificação	Quantidade de Citações
A C++ Data Model Supporting Reachability Analysis and Dead Code Detection	1998	[A1]	2
A New Technique for Copy Propagation And Dead Code Elimination Using Hash Based Value Numbering	2006	[A4]	1
Detecting Inconsistencies Via Universal Reachability Analysis	2012	[A35]	1
Eliminating Dead Code from XQuery Programs	2010	[A51]	1
Eliminating Dead Code on Recursive Data	2003	[A50]	2
Eliminating Partially Dead Code in Explicitly Parallel Programs	1998	[A12]	1
Generating Tests from Counterexamples	2004	[A16]	1
Partial Dead Code Elimination	1994	[A21]	11
Partial Dead Code Elimination Using Slicing Transformations	1997	[A36]	8
Path Profile Guided Partial Dead Code Elimination Using Predication	1997	[A37]	4
Practical Extraction Techniques for Java	2002	[A38]	1
Reachability Analysis for Annotated Code	2007	[A40]	1
Simple Relational Correctness Proofs for Static Analyses and Program Transformations	2004	[A43]	2
Source Code Survival with Kaplan Meier Estimator	2011	[A45]	1

Fonte: Do autor (2017).

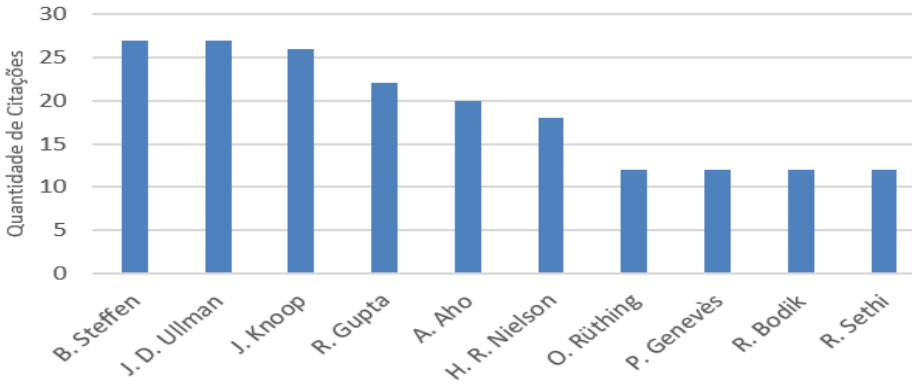
Figura A.3 - Análise dos Autores



Fonte: Do autor (2017).

Outra análise realizada para identificação de possíveis autores influentes na área foi verificar os citados nas referências dos artigos selecionados. Foram identificados 1.561 autores distintos. Os 10 autores mais citados estão apresentados na Figura A.4. Como pode ser observado, com 27 citações cada, os autores mais citados foram `B. Steffen` e `J. Ullman`, podendo ser considerados importantes na área.

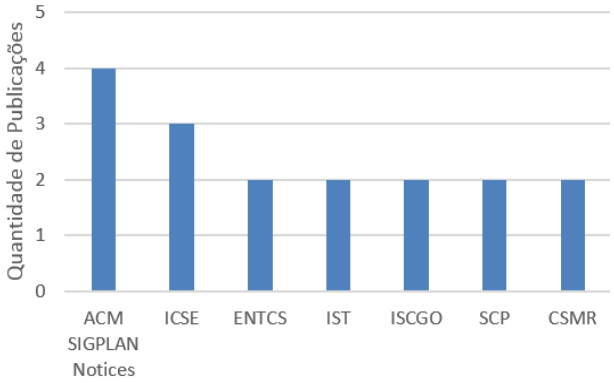
Figura A.4 - Autores mais Citados



Fonte: Do autor (2017).

Foram identificados 43 locais em que os artigos foram publicados. Os locais que tiveram mais de uma publicação são apresentados na Figura A.5. O local com mais publicações foi `ACM SIGPLAN Notices`, voltado para assuntos relacionados com compiladores e linguagens de programação. Isso ocorre porque parte dos trabalhos encontrados propõem a detecção de código morto com o objetivo de otimização de compiladores e, em menor quantidade, com propósito de facilitar as atividades de manutenção.

Figura A.5 - Locais de Publicação dos Artigos



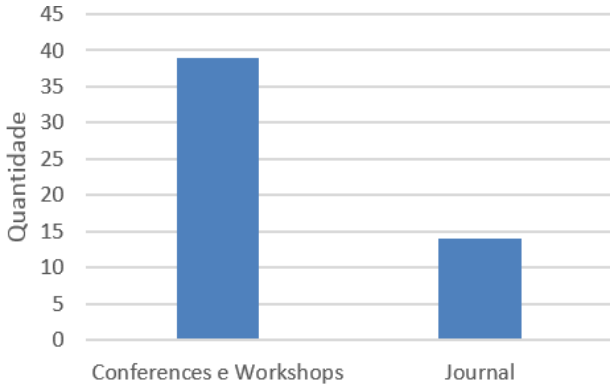
Fonte: Do autor (2017).

Em segundo lugar, com três publicações foi a `International Conference Software Engineering` (ICSE). Em seguida, com duas publicações foram `Eletronic Notes in Theorical Computer Science` (ENTCS),

Information and Software Technology (IST), International Symposium on Code Generation and Optimization (ISCGO), Science of Computer Programming (SCP) e Conference on Software Maintenance and Reengineering (CSMR).

Na Figura A.6, são apresentados os tipos de locais em que os artigos foram publicados. A maioria dos artigos foram publicados em Conferências e *Workshops* (73,58%) e, em menor quantidade, em *Journals* (26,42%).

Figura A.6 - Tipos de Veículos de Publicação



Fonte: Do autor (2017).

APÊNDICE B - ARTIGOS RESULTANTES DA RSL

Na Tabela B**.1**, são apresentadas as referências dos artigos resultantes da Revisão Sistemática da Literatura realizada.

Tabela B.1 - Artigos Resultantes da RSL

#	Referência
[A1]	Chen, Y.; Emden, R. G.; Eleftherios, K. A C++ Data Model Supporting Reachability Analysis and Dead Code Detection. In: IEEE Transactions on Software Engineering. v. 24, n. 9, p. 682-694. 1998.
[A2]	Guerrouat, A.; Harald R. A Combined Approach for Reachability Analysis. In: International Conference on Software Engineering Advances. p.23. 2006.
[A3]	Dai, X., Zhai, A., Hsu, W. C., Yew, P. C. A General Compiler Framework for Speculative Optimizations Using Data Speculative Code Motion. In: International Symposium on Code Generation and Optimization. pp. 280-290. 2005.
[A4]	Sunitha, K. V. N.; Kumar. V. V. A New Technique for Copy Propagation and Dead Code Elimination Using Hash Based Value Numbering. In: Advanced Computing and Communications. pp. 601-604. 2006.
[A5]	Fuhs, J.; James Cannady. An Automated Approach in Reverse Engineering Java Applications Using Petri Nets. In: SoutheastCon. pp. 90-96. 2004.
[A6]	Beyer, D., Henzinger, T., Jhala, R., Majumdar, R. An Eclipse Plug-in for Model Checking. In: IEEE International Workshop on Program Comprehension. pp. 251-255. 2004.
[A7]	Tempero, E. An Empirical Study of Unused Design Decisions in Open Source Java Software. In: Software Engineering Conference. pp. 33-40. 2008.
[A8]	Behera, C. K.; Pawan, K. An Improved Algorithm for Loop Dead Optimization. In: ACM SIGPLAN Notices. v. 41, n. 5, pp. 11-20. 2006.
[A9]	Scanniello, G. An Investigation of Object-Oriented and Code-Size Metrics as Dead Code Predictors. In: Conference on Software Engineering and Advanced Applications. pp. 392-397. 2014
[A10]	Davis, I. J., Godfrey, M. W., Holt, R. C., Mankovskii, S., Minchenko, N. Analyzing Assembler To Eliminate Dead Functions: An Industrial Experience. In: European Conference on Software Maintenance and Reengineering. pp. 467-470. 2012.
[A11]	Tardieu, O.; Stephen A. E. Approximate Reachability for Dead Code Elimination in Esterel. In: Automated Technology for Verification and Analysis. pp. 323-337. 2005.
[A12]	Knoop, J. Eliminating Partially Dead Code in Explicitly Parallel Programs. In: Theoretical Computer Science. v. 196, n. 1, p. 365-393. 1998.
[A13]	Geneves, P.; Layaïda, N. Equipping IDEs with XML-Path Reasoning Capabilities. In: ACM Transactions on Internet Technology. v. 13, n. 4, p. 13. 2013.
[A14]	Schäf, M.; Narbonne, D. S; Wies, T. Explaining Inconsistent Code. In: Joint Meeting on Foundations of Software Engineering. pp. 521-531. 2013.
[A15]	Chou, H; Chang, K.; Kuo, S. Facilitating Unreachable Code Diagnosis and Debugging. In: Design Automation Conference. pp. 485-490 .2011.
[A16]	Beyer, D., Chlipala, A. J., Henzinger, T. A., Jhala, R., Majumdar, R. Generating Tests from Counterexamples. In: International Conference on Software Engineering. 2004.
[A17]	Eder, S., Junker, M., Jürgens, E., Hauptmann, B., Vaas, R., Prommer, K. H. How Much Does Unused Code Matter for Maintenance?. In: International Conference on Software Engineering. pp. 1102-1111. 2012.

Tabela B-1 - Artigos Resultantes da RSL (cont.)

#	Referência
[A18]	Ogawa, M.; Zhenjiang H.; Sasano, I. Iterative-Free Program Analysis. In: International Conference on Functional Programming. p. 111-123. 2003.
[A19]	Stone, A.; Strout, M.; Behere, S. May/Must Analysis and the DFAGen Data-Flow Analysis Generator. In: Information and Software Technology. v. 51, n. 10, p. 1440-1453. 2009.
[A20]	Jensen, S. H., Madsen, M.; Moller, A. Modeling the HTML DOM and Browser API in Static Analysis of JavaScript Web Applications. In: European Conference on Foundations of Software Engineering. pp. 59-69. 2011.
[A21]	Knoop, J.; Rüthing, O.; Steffen, B. Partial Dead Code Elimination. In: Conference on Programming Language Design and Implementation. v. 29, n. 6, pp. 147-158. 1994.
[A22]	Xue, J.; Cai, Q.; Gao, L. Partial Dead Code Elimination on Predicated Code Regions. In: Software: Practice and Experience. v. 36, n. 15, p. 1655-1685. 2006.
[A23]	Barros, J. B., Da Cruz, D., Henriques, P. R., Pinto, J. S. Assertion-Based Slicing and Slice Graphs. In: Formal Aspects of Computing. v. 24, n. 2, p. 217-248. 2012.
[A24]	Damiani, F.; Giannini, P. Automatic Useless-Code Elimination for HOT Functional Programs. In: Journal of Functional programming. v. 10, n. 06, p. 509-559. 2000.
[A25]	Lee, J.; David A. P.; Samuel P. M. Basic Compiler Algorithms for Parallel Programs. In: ACM SIGPLAN Notices. pp.1-12.1999.
[A26]	Nguyen, Q. H.; Scholz, B. Computing SSA Form with Matrices. In: Electronic Notes in Theoretical Computer Science. v. 190, n. 1, p. 121-132 . 2007.
[A27]	Novillo, D. R. U.; Schaeffer, J. Concurrent SSA Form in the Presence of Mutual Exclusion. In: IEEE Parallel Processing. pp. 356-364. 1998.
[A28]	Wand, M.; Siveroni, I. Constraint Systems for Useless Variable Elimination. In: Symposium on Principles of Programming Languages. pp. 291-302. 1999.
[A29]	El-Zawawy, M. A. Dead Code Elimination Based Pointer Analysis for Multithreaded Programs. In: Journal of the Egyptian Mathematical Society. v. 20, n. 1, pp. 28-37. 2012.
[A30]	Boomsma, H.; Gross, H. G. Dead Code Elimination for Web Systems Written in PHP: Lessons Learned from an Industry Case. In: International Conference of Software Maintenance. pp. 511-515. 2012.
[A31]	Kim, K.; Kim, J.; Yoo, W. Dead Code Elimination in CTOC. In: International Conference on Software Engineering Research, Management & Applications. pp. 584-588. 2007.
[A32]	Chabbi, M.; Mellor-Crummey, J. Deadspy: A Tool to Pinpoint Program Inefficiencies. In: International Symposium on Code Generation and Optimization. pp. 124-134. 2012.
[A33]	Takimoto, M. Demand-driven Partial Dead Code Elimination. In: Information and Media Technologies. v. 5, n. 0, pp. 79-86. 2012.
[A34]	Fernandes, T, Desharnais, J. Describing Data Flow Analysis Techniques with Kleene Algebra. In: Science of Computer Programming. v. 65, n. 2, pp. 173-194. 2007.
[A35]	Tomb, A, Flanagan, C. Detecting Inconsistencies Via Universal Reachability Analysis. In: International Symposium on Software Testing and Analysis. pp. 287-297. 2012.
[A36]	Bodik, R.; Gupta, R. Partial Dead Code Elimination Using Slicing Transformations. In: ACM SIGPLAN Notices. pp. 159-170. 1997.
[A37]	Gupta, R.; Benson, D. A.; Fang, J. Z. Path Profile Guided Partial Dead Code Elimination Using Predication. In: International Conference on Parallel Architectures and Compilation Techniques. pp. 102-113. 1997.
[A38]	Tip, F., Sweeney, P. F., Laffra, C., Eisma, A., Streeter, D. Practical Extraction Techniques for Java. In: ACM Transactions on Programming Languages and Systems. v. 24, n. 6, p. 625-666. 2002.
[A39]	Saabas, A.; Uustalu, T. Program and Proof Optimizations with Type Systems. In: The Journal of Logic and Algebraic Programming. v. 77, n. 1, p. 131-154. 2008.
[A40]	Janota, M.; Grigore, R.; Moskal, M. Reachability Analysis for Annotated Code. In: Specification and Verification of Component-Based Systems. pp.23-30. 2007.
[A41]	Cai, Q., Gao, L. Xue, J. Region-Based Partial Dead Code Elimination on Predicated Code. In: Compiler Construction. Springer Berlin Heidelberg. pp. 150-166. 2004.
[A42]	Gupta, R.; Berson, D. A.; Fang, J. Z. Resource-Sensitive Profile-Directed Data Flow Analysis for Code Optimization. In: International Symposium on Microarchitecture. pp. 358-368. 1997.
[A43]	Benton, N. Simple Relational Correctness Proofs for Static Analyses and Program

	Transformations. In: ACM SIGPLAN Notices. v. 39, n.1, pp. 14-25. 2004.

Tabela B-1 - Artigos Resultantes da RSL (cont.)

#	Referência
[A44]	Wagner, G.; Gal, A.; Franz, M. “Slimming” a Java Virtual Machine by Way of Cold Code Removal and Optimistic Partial Program Loading. In: Science of Computer Programming. v. 76, n. 11, p. 1037-1053. 2011.
[A45]	Scanniello, G. Source Code Survival with the Kaplan Meier. In: International Conference of Software Maintenance. pp. 524-527. 2011.
[A46]	Payet, É; Spoto, F. Static Analysis of Android Programs. In: Information and Software Technology. v. 54, n. 11, p. 1192-120. 2012.
[A47]	Lokuciejewski, P.; Kelter, T.; Marwedel, P. Superblock-Based Source Code Optimizations for WCET Reduction. In: International Conference on Computer and Information Technology. pp. 1918-1925. 2010.
[A48]	Gutzmann, T.; Lundberg, J.; Lowe, W. Towards Path-Sensitive Points-to Analysis. In: International Conference on Source Code Analysis and Manipulation. p. 59-68. 2007.
[A49]	Su, L.; Lipasti, M. Dynamic Class Hierarchy Mutation. In: International Symposium on Code Generation and Optimization. pp. 98-110. 2006.
[A50]	Liu, Y. A.; Stoller, S. Eliminating Dead Code on Recursive Data. In: Static Analysis. Springer Berlin Heidelberg. pp. 211-231. 1999.
[A51]	Genevès, P.; Layaïda, N. Eliminating Dead-Code from XQuery Programs. In: International Conference on Software Engineering. pp. 305-306. 2010.
[A52]	Saabas, A.; Uustalu, T. Type Systems for Optimizing Stack-Based Code. In: Electronic Notes in Theoretical Computer Science. v. 190, n. 1, pp. 103-119. 2007.
[A53]	Jantz, M. J.; Kulkarni, P. A. Understand and Categorize Dynamically Dead Instructions for Contemporary Architectures. In: Workshop on Interaction between Compilers and Computer Architectures. pp. 25-32. 2012.

APÊNDICE C - FORMULÁRIOS DO ESTUDO EXPERIMENTAL

Neste Apêndice, são apresentados os formulários e os questionários utilizados no experimento realizado para avaliar a abordagem DCEVizz. Os espaços reservados para respostas foram omitidos para economia de espaço.

C1. Termo de Consentimento

O termo de consentimento foi o primeiro formulário entregue aos participantes, deixando-os cientes de que suas respostas e opiniões seriam utilizados na avaliação da abordagem.

Avaliação da Abordagem DCEVizz - Formulário de Consentimento

Você está sendo convidado(a) a participar, como voluntário(a), do estudo de viabilidade da abordagem DCEVizz (Dead Code Evolution Visualization), que visa apoiar a identificação e a visualização de código morto na evolução de sistemas de software orientados a objetos Java. Para tanto, deverão ser analisadas duas versões dos sistemas de software DubMan e PlayLister manualmente e utilizando a ferramenta DCEVizz Tool. A execução dessas tarefas será orientada por meio de questionários que serão submetidos aos participantes.

Desse modo, salientamos que:

- Sua participação não é obrigatória. A qualquer momento você poderá desistir de participar e retirar seu consentimento. Sua recusa, desistência ou retirada de consentimento não acarretará prejuízos;
- Os dados obtidos por meio desta pesquisa serão confidenciais e não serão divulgados em nível individual, visando assegurar o sigilo de sua participação;
- O pesquisador responsável se compromete a tornar públicos nos meios acadêmicos e científicos os resultados obtidos de forma consolidada, sem qualquer identificação dos participantes;
- Esta pesquisa não ocasiona nenhum risco ou custo financeiro ao participante;
- Não haverá nenhum tipo de recompensa aos participantes, a não ser a aquisição de conhecimento acerca do tema estudado;
- Os participantes são livres para solicitar quaisquer informações ao pesquisador responsável a qualquer momento durante a execução da pesquisa;

- Você deve comprometer-se a não comunicar seus resultados enquanto o estudo não for concluído;
- Você deve ter ciência de que suas respostas serão utilizadas para fins avaliativos de uma abordagem. Portanto, é fundamental que você se comprometa a realizar com responsabilidade as análises necessárias para responder o questionário.

Declaro que entendi os objetivos, riscos e benefícios de minha participação na pesquisa, que participo de livre e espontânea vontade, de forma a contribuir com o avanço da pesquisa em Visualização de Software.

Lavras, ___ de _____ de _____.

Assinatura do(a) participante: _____.

Assinatura da pesquisadora responsável: _____.

C2. Formulário de Caracterização do Participante

O formulário de caracterização foi o segundo a ser entregue aos participantes, com finalidade de coletar informações a respeito de sua formação profissional e de sua experiência nos assuntos relacionados a abordagem DCEVizz.

Avaliação da Abordagem DCEVizz - Questionário de Caracterização do Participante

Prezado(a),

Este questionário visa coletar informações que permitem caracterizar seu perfil e verificar seu grau de familiaridade em relação aos assuntos abordados nesta pesquisa.

1. Formação acadêmica

( ) Doutorado concluído
( ) Doutorado em andamento
( ) Mestrado concluído
( ) Mestrado em andamento
( ) Graduação concluída
( ) Graduação em andamento

Instituição acadêmica: _____

Ano de ingresso: _____ Ano de conclusão / Ano previsto de conclusão: _____

2. Experiência do Participante em Relação aos Assuntos Abordados na Pesquisa

a. Você possui alguma experiência com desenvolvimento de software Orientado a Objetos (OO)? (Assinale todas as alternativas que se aplicam)

( ) Meu contato com desenvolvimento OO se restringe aos trabalhos de graduação.

( ) Tenho experiência de ____anos e ____meses em desenvolvimento OO em projetos pessoais.

( ) Tenho experiência de ____anos e ____meses em desenvolvimento OO em projetos no ambiente acadêmico (exemplo: iniciação científica, projeto de pesquisa, trabalho de conclusão de curso).

( ) Tenho experiência de ___anos e ___ meses em desenvolvimento OO em projetos na indústria.

b. Seu contato com orientação a objetos foi utilizando quais linguagens de programação?

c. Considerando a escala apresentada na Tabela C.1, indique seu grau de experiência nas áreas de conhecimento listadas na Tabela C.2.

Tabela C.1 - Escala para Caracterização do Grau de Experiência

Número	Significado
0	Nenhum
1	Estudei em aula ou em livro
2	Pratiquei em projetos em sala de aula
3	Utilizei em projetos pessoais
4	Utilizei em projetos na indústria

Tabela C.2 - Áreas de Conhecimento

Área de Conhecimento	Significado				
Engenharia de Software	0	1	2	3	4
Manutenção de Software	0	1	2	3	4
Legibilidade de Código	0	1	2	3	4
Código Morto	0	1	2	3	4
Evolução de Software	0	1	2	3	4
Visualização de Software	0	1	2	3	4

Desde já, agradecemos sua disponibilidade e participação!

Camila Bastos

Heitor Costa

C3. Contextualização

Para imersão dos participantes no contexto da avaliação, foi estabelecida uma situação fictícia, em que uma empresa possui problemas com a manutenibilidade de seus sistemas de software.

Contextualização da Pesquisa

DCEVizz: Visão Geral. A abordagem DCEVizz (*Dead Code Evolution Visualization*) foi elaborada para auxiliar a compreender a evolução de código morto em software orientado a objetos. De modo geral, essa abordagem identifica código morto (métodos não chamados) em versões de software Java e, em seguida, apresenta esse código morto utilizando técnicas de visualização de software, dando ênfase em suas características evolutivas. Um *plug-in* para o Eclipse (DCEVizz Tool) foi implementado para automatizar a execução dessa abordagem.

Objetivo do Questionário. Este questionário tem como objetivo verificar os benefícios obtidos com a abordagem DCEVizz. Para tanto, foram definidas as seguintes metas:

1. Verificar a facilidade provida pela abordagem para detectar código morto;
2. Verificar a capacidade de representação da evolução do código morto das visualizações de DCEVizz.

Contextualização. Você foi contratado(a) como engenheiro de software de uma empresa que possui diversos sistemas de software Java em diferentes versões. É de fundamental importância que a manutenibilidade de todas as versões seja mantida, pois estão em uso por diferentes clientes e contribuem significativamente com os lucros da empresa, sendo alvo de constantes atividades de manutenção. Um dos problemas enfrentados pela empresa é a alta rotatividade de funcionários, fazendo com que os mantenedores tenham pouca familiaridade com o software, dificultando a etapa de manutenção. Desse modo, sua função é reduzir essa dificuldade, garantindo que as versões desses sistemas sejam de fácil compreensão e tenham boa legibilidade.

Tarefa. Sabendo que a presença de métodos mortos prejudica a legibilidade do software, sua tarefa inicial é identificar e analisar a evolução desses métodos em duas versões de dois sistemas de software (**DubMan** e **PlayLister**) da empresa. Essa tarefa será guiada por meio de um questionário e deverá ser realizada em duas etapas, sendo que na primeira etapa você deverá analisar as versões do software **DubMan** sem o apoio de DCEVizz Tool. Na segunda etapa, você deverá analisar as versões do software **PlayLister** utilizando o apoio de DCEVizz Tool.

Instruções Finais.

- Resolva as tarefas do questionário na ordem em que elas são apresentadas;
- Registre o horário de início e o horário de fim de cada atividade sempre que solicitado;
- Para responder o questionário **sem** o apoio de DCEVizz Tool, você poderá utilizar o recurso *Call Hierarchy* do Eclipse. **Lembre-se**: um método é considerado morto se ele não possui chamador, ou seja, o resultado do recurso *Call Hierarchy* do Eclipse é vazio.

C4. Etapa 1 - Sem o uso da abordagem DCEVizz

Este questionário foi aplicado na primeira etapa da avaliação da abordagem DCEVizz, no qual os participantes deveriam respondê-lo sem o uso da abordagem. As questões foram definidas com dois focos: i) detecção de código morto (questões de 1 a 5) e; ii) compreensão da evolução do código morto (questões de 6 a 10).

Etapa 1. SEM o uso de DCEVizz Tool

Prezado(a) Participante,

Esta é a primeira etapa para avaliação da abordagem DCEVizz. Você deverá responder as questões a seguir analisando o software **DubMan**, **SEM** o uso do apoio computacional DCEVizz Tool.

Questões

Escreva o horário de início desta atividade: _____

Q1. Para a versão **DubManV1**, o método **saveEmpty** da classe **JobsGenerator.java**, pertencente ao pacote **net.sourceforge.dubman** possui quantos chamadores diretos? Escreva o nome dele(s).

Obter a resposta dessa questão foi:

( ) Muito Fácil ( ) Fácil ( ) Mais ou Menos Fácil ( ) Difícil ( ) Muito Difícil

Q2. Para a versão **DubManV2**, quais são os métodos chamadores do método **getStatus** da classe **JobProcess.java** pertencente ao pacote **net.sourceforge.dubman.copy**?

Obter a resposta dessa questão foi:

( ) Muito Fácil ( ) Fácil ( ) Mais ou Menos Fácil ( ) Difícil ( ) Muito Difícil

Q3. Para a versão **DubManV1**, o método **getColumnName** da classe **JobTableModel.java** pertencente ao pacote **net.sourceforge.dubman** é morto? Justifique.

Obter a resposta dessa questão foi:

( ) Muito Fácil ( ) Fácil ( ) Mais ou Menos Fácil ( ) Difícil ( ) Muito Difícil

Q4. Para a versão **DubManV2**, os métodos chamadores do método **addJob** da classe **JobTemplate.java** pertencente ao pacote **net.sourceforge.dubman** são todos mortos? Justifique.

Obter a resposta dessa questão foi:

( ) Muito Fácil ( ) Fácil ( ) Mais ou Menos Fácil ( ) Difícil ( ) Muito Difícil

Q5. Para a versão **DubManV1**, a classe **JobEditDialog.java** pertencente ao **net.sourceforge.dubman** possui algum método morto (zero chamadas)? Se sim, escreva o nome de um deles.

Obter a resposta dessa questão foi:

( ) Muito Fácil ( ) Fácil ( ) Mais ou Menos Fácil ( ) Difícil ( ) Muito Difícil

Escreva o horário de finalização desta atividade: _____

Escreva o horário de início desta atividade: _____

Q6. O método morto **processRunning** da classe **ProcessManager.java** pertencente ao pacote **net.sourceforge.dubman** (**DubManV1**) também é morto na versão **DubManV2**? Justifique.

Obter a resposta dessa questão foi:

( ) Muito Fácil ( ) Fácil ( ) Mais ou Menos Fácil ( ) Difícil ( ) Muito Difícil

Q7. A classe **JobTableModel.java** pertencente ao pacote **net.sourceforge.dubman** possui métodos mortos em **DubManV1**. Essa mesma classe possui **pelo menos um** método morto em **DubManV2**? Se sim, escreva o nome do método morto.

Obter a resposta dessa questão foi:

( ) Muito Fácil ( ) Fácil ( ) Mais ou Menos Fácil ( ) Difícil ( ) Muito Difícil

Q8. Sabe-se que a classe **DubMan.java** do pacote **net.sourceforge.dubman**, na versão **DubManV1**, possui os métodos mortos **getCurrentTemplate** e **getCurrentFile**. Na versão **DubManV2**:

i. O método **getCurrentTemplate**:	ii. O método **getCurrentFile**:
( ) ainda é morto	( ) ainda é morto
( ) voltou a ser utilizado	( ) voltou a ser utilizado
( ) foi excluído	( ) foi excluído

Justifique. Obter a resposta dessa questão foi:

( ) Muito Fácil ( ) Fácil ( ) Mais ou Menos Fácil ( ) Difícil ( ) Muito Difícil

Q9. Sabe-se que, na versão **DubManV1**, as classes **Timer.java** e **CommandLogger.java** do pacote **net.sourceforge.dubman** possuem métodos mortos. Essas classes ainda existem na versão **DubManV2**? Justifique.

Obter a resposta dessa questão foi:

( ) Muito Fácil ( ) Fácil ( ) Mais ou Menos Fácil ( ) Difícil ( ) Muito Difícil

Q10. Em **DubManV1**, o método **doTestAllString** da classe **JobFileFilterTests.java** pertencente ao pacote **net.sourceforge.dubman.tests** é considerado morto por possuir um método chamador que também é morto. Esse método continua com essa característica em **DubManV2**? Justifique.

Obter a resposta dessa questão foi:

( ) Muito Fácil ( ) Fácil ( ) Mais ou Menos Fácil ( ) Difícil ( ) Muito Difícil

Escreva o horário de finalização desta atividade: _____

C5 Etapa 2 - Com o uso da abordagem DCEVizz

Este questionário foi aplicado na segunda etapa da avaliação da abordagem DCEVizz, no qual os participantes deveriam respondê-lo utilizando a abordagem. As questões são iguais as definidas no questionário da Etapa 1, porém, o software analisado é diferente. Além disso, foram acrescentadas três questões que necessitam de uma visão geral do código morto para serem respondidas, sendo inviáveis na primeira etapa.

Etapa 2. Com o uso de DCEVizz Tool

Prezado(a) Participante,

Esta é a segunda etapa para avaliação da abordagem DCEVizz. Você deverá responder as questões a seguir analisando o software **PlayLister**, **COM** o uso do apoio computacional DCEVizz Tool.

Questões

Escreva o horário de início desta atividade: _____

Q1. Para a versão **PlayListerV1**, o método **loadMusicMatchFile** da classe **Playlist.java**, pertencente ao pacote **helliker.id3** possui quantos chamadores diretos? Escreva o nome dele(s).

Obter a resposta dessa questão foi:

( ) Muito Fácil ( ) Fácil ( ) Mais ou Menos Fácil ( ) Difícil ( ) Muito Difícil

Q2. Para a versão **PlayListerV2**, quais são os métodos chamadores do método **getSuffix** da classe **FilterBySuffix.java** pertencente ao pacote **util**?

Obter a resposta dessa questão foi:

( ) Muito Fácil ( ) Fácil ( ) Mais ou Menos Fácil ( ) Difícil ( ) Muito Difícil

Q3. Para a versão **PlayListerV1**, o método **setEncodedBy** da classe **MP3File.java** pertencente ao pacote **helliker.id3** é morto? Justifique.

Obter a resposta dessa questão foi:

( ) Muito Fácil ( ) Fácil ( ) Mais ou Menos Fácil ( ) Difícil ( ) Muito Difícil

Q4. Para a versão **PlayListerV2**, os métodos chamadores do método **getYearAsInt** da classe **Song.java** pertencente ao pacote **base** são todos mortos? Justifique.

Obter a resposta dessa questão foi:

( ) Muito Fácil ( ) Fácil ( ) Mais ou Menos Fácil ( ) Difícil ( ) Muito Difícil

Q5. Para a versão **PlayListerV1**, a classe **DirFilter.java** pertencente ao pacote **util** possui algum método morto (zero chamadas)? Se sim, escreva o nome de um deles.

Obter a resposta dessa questão foi:

( ) Muito Fácil ( ) Fácil ( ) Mais ou Menos Fácil ( ) Difícil ( ) Muito Difícil

Escreva o horário de finalização desta atividade: _____

Escreva o horário de início desta atividade: _____

Q6. O método morto **getTaggingType** da classe **MP3File.java** pertencente ao pacote **helliker.id3** (**PlayListerV1**) também é morto na versão **PlayListerV2**? Justifique.

Obter a resposta dessa questão foi:

( ) Muito Fácil ( ) Fácil ( ) Mais ou Menos Fácil ( ) Difícil ( ) Muito Difícil

Q7. A classe **ID3v2p2Frames.java** pertencente ao pacote **helliker.id3** possui métodos mortos em **PlayListerV1**. Essa mesma classe possui **pelo menos um** método morto em **PlayListerV2**? Se sim, escreva o nome do método morto.

Obter a resposta dessa questão foi:

( ) Muito Fácil ( ) Fácil ( ) Mais ou Menos Fácil ( ) Difícil ( ) Muito Difícil

Q8. Sabe-se que a classe **ID3v2Frame.java** do pacote **helliker.id3**, na versão **PlayListerV1**, possui os métodos mortos **getGrouped** e **isEmpty**. Na versão **PlayListerV2**:

i. O método **getGrouped**:	ii. O método **isEmpty**:
( ) ainda é morto	( ) ainda é morto
( ) voltou a ser utilizado	( ) voltou a ser utilizado
( ) foi excluído	( ) foi excluído

Justifique. Obter a resposta dessa questão foi:

( ) Muito Fácil ( ) Fácil ( ) Mais ou Menos Fácil ( ) Difícil ( ) Muito Difícil

Q9. Sabe-se que, na versão **PlayListerV1**, as classes **PrefsScreen.java** e **DirSearcher.java** do pacote **fullGUI** possuem métodos mortos. Essas classes ainda existem na versão **PlayListerV2**? Justifique.

Obter a resposta dessa questão foi:

( ) Muito Fácil ( ) Fácil ( ) Mais ou Menos Fácil ( ) Difícil ( ) Muito Difícil

Q10. Em **PlayListerV1**, o método **getFrameLength** da classe **ID3v2Frame.java** pertencente ao pacote **helliker.id3** é considerado morto por possuir dois métodos chamadores que também são mortos. Esse método continua com essa característica na em **PlayListerV2**? Justifique.

Obter a resposta dessa questão foi:

( ) Muito Fácil ( ) Fácil ( ) Mais ou Menos Fácil ( ) Difícil ( ) Muito Difícil

Escreva o horário de finalização desta atividade: _____

Escreva o horário de início desta atividade: _____

Q11. Observando a visualização quantitativa de DCEVizz Tool (gráfico de linha), a quantidade de métodos mortos aumentou ou diminuiu ao longo das versões?

( ) aumentou ( ) diminuiu

Você acha que obter essa resposta sem o apoio de DCEVizz Tool seria:

( ) Muito Fácil ( ) Fácil ( ) Mais ou Menos Fácil ( ) Difícil ( ) Muito Difícil

Justifique._____

Q12. A empresa deseja saber qual pacote possui maior quantidade de métodos mortos para que o mesmo seja priorizado na atividade de manutenção perfectiva. Com base na visualização qualitativa de DCEVizz Tool, qual pacote você indicaria para essa situação em **PlayListerV1**? Justifique.

Você acha que obter essa resposta sem o apoio de DCEVizz Tool seria:

( ) Muito Fácil ( ) Fácil ( ) Mais ou Menos Fácil ( ) Difícil ( ) Muito Difícil

Justifique._____

Q13. Observando a visualização qualitativa, qual pacote você acha que excluiu maior quantidade de métodos mortos da primeira para a segunda versão? Justifique.

Resposta. _____

Você acha que obter essa resposta sem o apoio de DCEVizz Tool seria:

( ) Muito Fácil ( ) Fácil ( ) Mais ou Menos Fácil ( ) Difícil ( ) Muito Difícil

Justifique._____

Escreva o horário de finalização desta atividade: _____

C6. Questionário de Opinião

O questionário de opinião foi o último a ser aplicado na avaliação de DCEVizz, com intuito de coletar opiniões pessoais a respeito da abordagem e de seu apoio computacional.

Questionário de Opinião

Prezado(a) Participante,

A finalidade desta etapa é obter informações adicionais que possam contribuir para a avaliação da abordagem DCEVizz.

Questões

Q1. Você encontrou alguma dificuldade na realização das tarefas?

( ) SIM ( ) NÃO

Se **sim**, especifique-as:

Q2. O uso das visualizações da abordagem DCEVizz na execução das tarefas:

( ) facilitaram bastante

( ) facilitaram um pouco

( ) indiferente (não facilitou e nem dificultou em nada)

( ) dificultaram um pouco

( ) dificultaram bastante

Q3. Você sentiu alguma dificuldade em interpretar a visualização quantitativa (gráfico de linha) de DCEVizz?

( ) SIM ( ) NÃO

Se **sim**, especifique-as:

Q4. Você sentiu alguma dificuldade em interpretar a visualização qualitativa (retângulos aninhados) de DCEVizz?

( ) SIM ( ) NÃO

Se **sim**, especifique-as:

Q5. Na sua opinião, quais são os aspectos positivos da abordagem DCEVizz?

Q6. Na sua opinião, quais são os aspectos negativos da abordagem DCEVizz?

Q7. Você possui alguma sugestão de melhoria para a abordagem DCEVizz?

Se houver, deixe comentários adicionais a respeito de dificuldades, críticas e/ou sugestões relacionadas a abordagem DCEVizz.