Evolução do Ambiente SMartyModeling por meio de Testes Exploratórios

Leandro F. Silva[1], **Bruno Fernandes**[1], **Edson OliveiraJr**[1]

[1]Departamento de Informática
Universidade Estadual de Maringá (UEM)
Maringá, Brasil.

leandroflores7@gmail.com, bruno.fernandes@msn.com, edson@din.uem.br

Abstract. *Software Product Line (SPL) is a software reuse approach that has been consolidated over the past few years. However, the absence of tools that offer native support to the main activities in the life cycle of an SPL, motivated the development of the SMartyModeling environment. In order to identify problems when interacting with the environment, a series of exploratory tests were carried out. Therefore, this article presents the planning and execution of these exploratory tests, from the point of view of a tester who did not participate in the process of implementing the environment. The 12 defects resulting from these tests were discussed, describing the context in which they were identified and whether, to date, they have been corrected. Defects were also categorized by the features of the environment. The defects identified will be corrected in a new version of the environment. The result explains the importance of expanding to new tests that allow a safer and more reliable environment.*

Resumo. *Linha de Produto de Software (LPS) é uma abordagem de reuso de software que vem se consolidando no decorrer dos últimos anos. No entanto, a ausência de ferramentas que ofereçam suporte nativo às principais atividades no ciclo de vida de uma LPS, motivaram o desenvolvimento do ambiente SMartyModeling. Com o objetivo de identificar problemas ao interagir com o ambiente, foram realizados uma série de testes exploratórios. Portanto, este artigo apresenta o planejamento e a execução destes testes exploratórios, a partir do ponto de vista de um testador que não participou do processo de implementação do ambiente. Foram discutidos os 12 defeitos resultantes desses testes, descrevendo o contexto em que foram identificados e se, até o momento, foram corrigidos. Os defeitos foram também categorizados pelas funcionalidades do ambiente. Os defeitos identificados serão corrigidos em uma nova versão do ambiente. O resultado explica a importância de expandir para novos testes que permitam um ambiente mais seguro e confiável.*

1. Introdução

Softwares são constituídos de código-fonte e sua respectiva documentação, sendo o principal ativo de produtos. A preocupação com a competitividade no desenvolvimento de software se tornou uma questão primordial para empresas deste segmento, independentemente do porte e mercado alvo. Diante disso, os principais desafios passaram a ser lidar com o aumento de diversidade, demandas pela diminuição do tempo para entrega e desenvolvimento de software confiável [Sommerville 2011]. Técnicas voltadas ao reúso de

artefatos foram propostas, incluindo a Linha de Produto de Software (LPS), uma abordagem de desenvolvimento, que aplica o reúso de maneira sistemática em um domínio de atuação [Linden et al. 2007].

A ausência de ferramentas que ofereçam suporte às principais atividades relacionadas à LPS motivaram o desenvolvimento do ambiente SMartyModeling. O ambiente foi avaliado inicialmente em relação à modelagem de LPS. E como resultado, foi possível identificar limitações, que motivaram o desenvolvimento de uma nova versão com os defeitos identificados corrigidos e novas funcionalidades incluídas [Silva 2017]. Como disciplina, a Engenharia de Software (ES) se preocupa com todo o ciclo de produção, incluindo desde o levantamento de requisitos, especificação, desenvolvimento, validação e evolução de software. A validação de software tem a intenção de mostrar que um software atende suas especificações. E o teste de software é a principal técnica de validação [Sommerville 2011]. Nesse contexto, o planejamento e a execução de testes é fundamental para a evolução de um software, em especial a execução contínua de diferentes técnicas de testes e avaliações de usabilidade. A aplicação dos testes deve ser feita de maneira independente, incluindo a participação de membros que não participaram diretamente do processo de desenvolvimento, com o objetivo de reduzir o viés [Myers et al. 2011].

Portanto, este trabalho teve como objetivo realizar testes de software no SMartyModeling. Inicialmente, foi realizada uma análise mais minuciosa sobre a arquitetura do SMartyModeling e uma revisão sobre as principais técnicas de teste de software. Posteriormente, foram projetados e realizados testes exploratórios no ambiente, e, finalmente, analisados e discutidos os resultados obtidos. Este trabalho está organizado da seguinte maneira: a Seção 2 descreve os principais conceitos relacionados, a Seção 3 apresenta uma visão geral do ambiente testado neste estudo: o SMartyModeling, com a discussão dos resultados detalhadas na Seção 5, e as contribuições, limitações e trabalhos futuros na Seção 6.

2. Fundamentação Teórica

2.1. Teste de Software

O Teste de Software pode ser definido como um processo, ou conjunto de processos, projetado para garantir que o software execute o que foi planejado e que não realize nenhuma atividade não esperada [Myers et al. 2011]. O teste é destinado a mostrar que um programa realiza o que é proposto e para descobrir os defeitos do programa antes do uso. Os resultados do teste são voltados à procura de erros, anomalias ou informações sobre os atributos não funcionais do programa [Sommerville 2011].

Com objetivo de complementar o conceito de teste, é necessário apresentar a definição dos termos defeito, erro e falha no contexto de software. O defeito consiste em uma deficiência mecânica ou algorítmica que, se ativada, pode levar a uma falha. O erro corresponde à um item de informação ou estado de execução inconsistente. E, por fim, a falha é definida como um evento notável em que o sistema viola suas especificações. Assim, pode-se dizer que erro é a ação humana incorreta, defeito é a forma incorreta que o software foi construído e a falha é desvio percebido ao executar o software [ISTQB 2018].

No entanto, os testes não podem demonstrar se o software é completamente livre de defeitos em qualquer situação. É sempre possível que um novo teste eventualmente encontre mais problemas no sistema [Myers et al. 2011]. Segundo Dijkstra [Dijkstra 1972],

os testes mostram apenas a presença de erros, e não sua ausência. Apesar disso, o teste é parte de um amplo processo de verificação e validação do software, sendo fundamental para estabelecer a confiança de que o software está pronto para atender seu objetivo final [Sommerville 2011].

É praticamente impossível demonstrar a ausência de defeitos em um software, pelo fato de normalmente existir uma quantidade infinita de combinações possíveis para os dados de entrada [Dijkstra 1972]. Neste cenário, foram propostas estratégias econômicas para a execução dos testes. Duas técnicas clássicas de testes descritas foram [Myers et al. 2011]: **Teste de Caixa-Preta:** tem como objetivo testar o software sem qualquer preocupação com a estrutura interna do programa, por isso caixa-preta. Assim, a partir entrada de dados (válidas e inválidas), espera-se uma saída após o processamento. Como exemplo, temos o Teste de Sistema e o Teste de Aceite; e **Teste de Caixa-Branca:** tem como objetivo testar o software examinando a estrutura interna do programa, sendo orientado a lógica. Portanto, busca-se fazer com que cada instrução do programa seja executada pelo menos uma vez para um teste completo, o que se torna inviável. Como exemplo, temos o Teste de Unidade e o Teste de Integração.

Diante da inviabilidade lógica e das limitações para construção dos testes pelas técnicas apresentadas, a técnica de teste baseada na experiência foi proposta. Tal técnica é voltada à uma política mais informal e tem sua modelagem, execução, registro e avaliação feitos dinamicamente durante a execução dos testes. Como exemplo, o teste exploratório é utilizado, principalmente em situações que há especificações limitadas ou pressão de tempo significativa nos testes [ISTQB 2018].

2.2. Linha de Produto de Software e Gerenciamento de Variabilidades

Linha de Produto de Software (LPS) é uma abordagem de desenvolvimento centrado no reuso de forma sistemática em um domínio de atuação [Linden et al. 2007]. LPS é composta por um conjunto de sistemas de software que compartilham características comuns e gerenciáveis, que satisfazem as necessidades de um segmento particular ou de uma missão [Pohl et al. 2005]. A engenharia de LPS abrange todas as atividades envolvidas no planejamento, produção, e manutenção de produtos. Um *framework* é proposta por [Pohl et al. 2005] para a Engenharia de LPS a partir de dois processos: **Engenharia de Domínio:** processo responsável pela identificação e definição das similaridades e variabilidades da LPS; e **Engenharia de Aplicação:** processo responsável pela aplicação da LPS, em que os produtos são construídos reutilizando artefatos de domínio e resolvendo as variabilidades.

A Engenharia de Domínio é composta de cinco atividades: Gerenciamento do Produto, Engenharia de Requisitos do Domínio, Análise do Domínio, Implementação do Domínio e Teste do Domínio. Estes processos produzem uma plataforma incluindo as similaridades entre as aplicações e suas variabilidades que permitem a customização de produtos em massa. A partir da Engenharia de Domínio, são definidos os ativos base da LPS. A Engenharia de Aplicação é composta pelas seguintes atividades: Engenharia de Aplicação, Análise da Aplicação, Implementação da Aplicação e Testes da Aplicação, tendo como resultado a instanciação de produtos finais a partir dos ativos base [Pohl et al. 2005]. A Figura 1 apresenta a visão geral destes dois processos. Na parte superior da Figura 1, a sequência de atividades da Engenharia de Produto é detalhada,

resultando na identificação e definição dos ativos, que formam a base para as atividades da Engenharia de Aplicação, descritas na parte inferior da Figura 1.

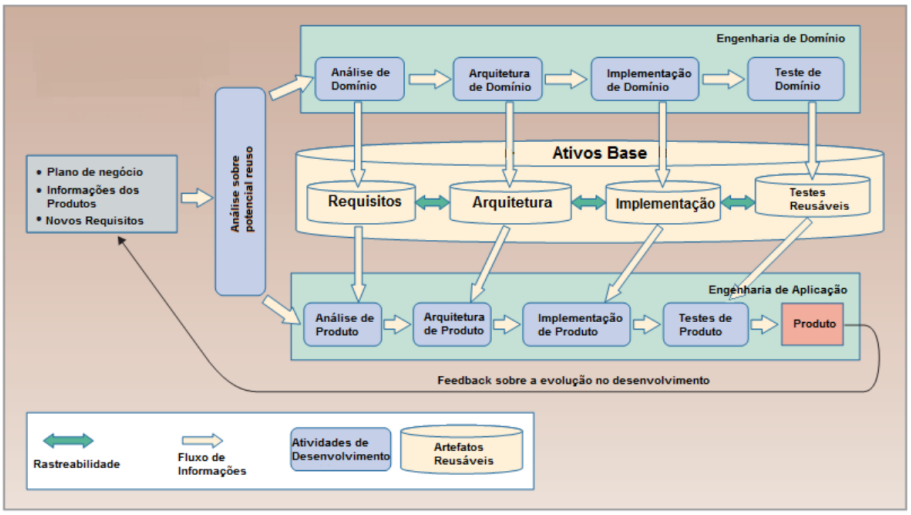

Figura 1. *Framework* de Engenharia de Linha de Produto de Software [Pohl et al. 2005].

Nos últimos anos, a abordagem de LPS vem definindo cada vez mais o Gerenciamento de Variabilidades (GV) como atividade fundamental para a instanciação de produtos em larga escala [Capilla et al. 2013]. Variabilidade de software é a capacidade de um sistema, ativo de software, ou ambiente de desenvolvimento ser configurado, customizado, ou alterado para uso em um domínio específico de uma forma pré-planejada. Em síntese, variabilidade é a forma como os membros de uma família de produtos podem se diferenciar entre si, ou seja, é o que difere os produtos gerados de uma mesma LPS [Capilla et al. 2013, Linden et al. 2007]. Para a abordagem de LPS, várias alternativas ao longo do tempo vêm sendo propostas para representar a variabilidade. No entanto, para a representação de uma variabilidade, existem três elementos fundamentais [OliveiraJr et al. 2010, Linden et al. 2007]:

- **Ponto de Variação:** descreve o ponto em que existe uma diferença nos produtos a serem instanciados, identificando os locais nos quais as variações são combinadas para ocorrer. São responsáveis por definir as diferenças da LPS;
- **Variante:** define os elementos para a resolução de um Ponto de Variação. Corresponde a uma alternativa de projeto para resolver uma determinada variabilidade. Define como uma variabilidade ou ponto de variação varia em uma LPS; e
- **Restrições entre Variantes:** estabelecem os relacionamentos entre uma ou mais Variantes com o objetivo de resolver seus respectivos Pontos de Variação.

Algumas abordagens para a representação de variabilidades em modelos UML são propostas na literatura, como PLUS [Gomaa 2006], Ziadi [Ziadi and Jezequel 2006], e SMarty [OliveiraJr et al. 2013]. A abordagem *Stereotype-based Management of Variability* (SMarty) é uma abordagem anotativa de GV baseada em estereótipos UML. A SMarty é composta por um perfil UML denominado *SMartyProfile*, que representa variabilidades por meio de estereótipos, e por um processo denominado *SMartyProcess*, que guia o usuário na identificação, representação e rastreamento de variabilidades em modelos de LPS [OliveiraJr et al. 2013]. A principal contribuição da SMarty é permitir o

gerenciamento de forma efetiva das variabilidades de uma LPS modeladas em UML. Os estereótipos definidos pela abordagem SMarty estão organizados em um perfil UML denominado *SMartyProfile* e em uma forma de representação de variabilidades por meio de estereótipos. O *SMartyProfile* é formado por um conjunto de estereótipos e meta-atributos para representar variabilidades em modelos UML de LPS [OliveiraJr et al. 2010].

3. O Ambiente SMartyModeling

O SMartyModeling é um ambiente para engenharia de LPSs baseadas em UML, nas quais as variabilidades são modeladas como estereótipos usando qualquer perfil UML compatível, adotando como padrão a abordagem SMarty. O ambiente oferece suporte à modelagem de diagramas de casos de uso, classes, componentes, sequência e atividades. As principais *features* do ambiente são: modelagem de diagramas, definição e restrição de variabilidades, rastreabilidade de elementos, e instanciação de diagramas por meio da resolução de variabilidades, configuração e exportação de produtos [Silva 2017].

A arquitetura do SMartyModeling foi instanciada a partir da VMTools-RA, uma Arquitetura de Referência para ferramentas de variabilidade de software [Allian 2016]. A VMTools-RA não especifica um estilo arquitetural, e a arquitetura do SMartyModeling foi instanciada considerando o padrão *Model-View-Controller* (MVC). O ambiente foi codificado na linguagem JavaSE para Desktop, com a arquitetura organizada em quatro pacotes: Modelo, Controle, Visão e Arquivo [Silva 2017]. A Figura 2 apresenta a interface principal do SMartyModeling, composta pelos seguintes componentes:

- **Painel Principal (Componente A):** contendo as principais operações do sistema: novo, abrir, salvar e fechar Projeto; desfazer, refazer; zoom+ e zoom-; exportar imagem, ajuda e sobre;
- **Painel de Operações (Componente B):** operações de acordo com o respectivo diagrama. Para o diagrama de casos de uso temos: arrastar e manipular elementos do diagrama; inserir novo ator; inserir novo caso e uso; inserir nova variabilidade; editar elemento, excluir elemento e inserir nova associação;
- **Painel do Projeto (Componente C):** organização do Projeto em uma ordem hierárquica, com a seguinte ordem: diagramas, variabilidades, produtos, instâncias, métricas e rastreabilidade;
- **Painel de Modelagem (Componente D):** modelagem dos diagramas; e
- **Painel de Informações (Componente E):** abas com as respectivas informações sobre o elemento selecionado.

No decorrer do desenvolvimento do ambiente, os testes ficaram restritos ao planejamento e execução de testes unitários envolvendo as principais classes de modelo e de controle, concentrando os esforços em cobrir as operações descritas de acordo com cada método. Os testes foram codificados utilizando o *framework* JUnit[1], voltado especificamente à automação de testes na linguagem de programação Java.

4. Testes Exploratórios no SMartyModeling

A partir da compreensão das principais funcionalidades do ambiente SMartyModeling, foram executados testes com o objetivo de analisar minuciosamente a ferramenta e si-

[1]Disponível em: `https://junit.org/junit4/javadoc/latest/index.html`.

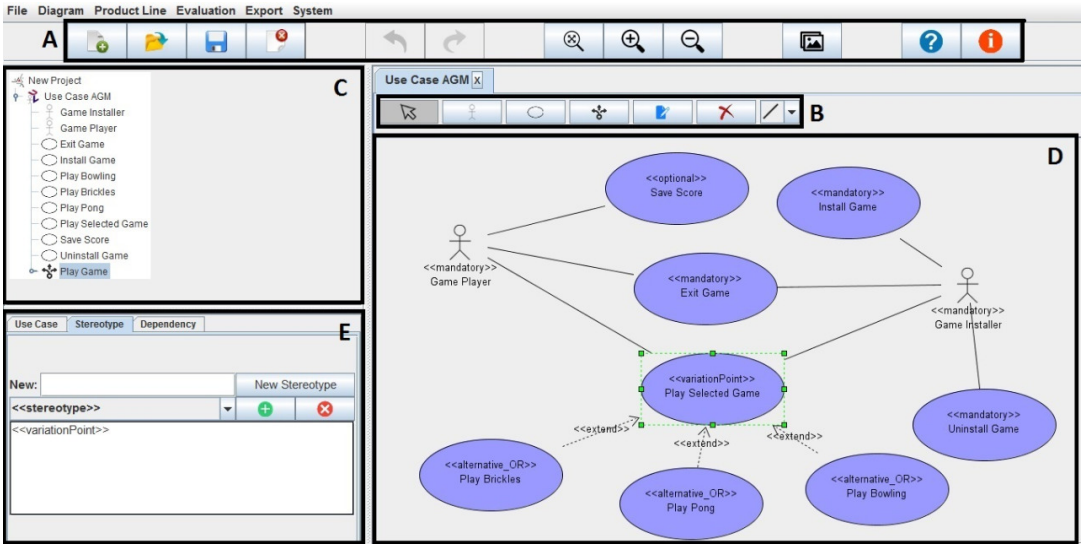

Figura 2. Interface do SMartyModeling.

multaneamente identificar possíveis defeitos[2] relacionados à falta de experiência no ambiente. Os testes foram executados por um usuário que não participou em nenhuma etapa do desenvolvimento da ferramenta. O usuário responsável pela realização do teste exploratório possui experiência em técnicas de testes aplicadas no mercado de trabalho, possuindo igualmente experiência em testes exploratórios, no entanto, não possui conhecimento avançado em LPS, e sobre as funcionalidades do ambiente.

A técnica de testes selecionada para o SMartyModeling foi o teste exploratório [Pfahl et al. 2014], principalmente em função do contexto de aplicação. Os testes exploratórios são especialmente úteis em ambientes em que a documentação é limitada e o responsável por examinar o software não possui conhecimento prévio e experiência sobre o software. A abordagem de testes exploratórios proporciona alguns benefícios que vão ao encontro das limitações mencionadas: contribuem para o aprendizado do funcionamento do sistema em teste, demanda um tempo aceitável de planejamento, além de revelar tipos de falhas que geralmente não são detectados em um plano ou caso de teste pré-estabelecido. Os testes exploratórios de maneira alguma podem ser utilizados como única estratégia de teste, no entanto podem ser utilizados como complemento à demais técnicas de teste com objetivo de obter uma melhoria na qualidade do software, e complementam eventuais falhas não identificadas nos testes unitários e avaliações de usabilidade inicialmente realizadas.

Primeiramente, para organização dos testes realizados, foi criado um quadro *Kanban* no Trello[3], organizando as atividades em listas, e possibilitando um controle mais efetivo sobre os testes realizados. Foi necessário preparar o ambiente para execução. Para isso, foram instalados o Apache NetBeans IDE 11.1[4], o OpenJDK[5] e o sistema de controle

[2]Todos os defeitos e correções estão disponíveis em `https://doi.org/10.5281/zenodo.3706642`.

[3]Trello é um aplicativo na web disponível em `https://trello.com` para gerenciamento de projetos utilizando conceitos de Kanban.

[4]Disponível em `https://netbeans.apache.org`

[5]Disponível em `https://openjdk.java.net`

de versão distribuído Git[6]. Foi realizado também o Clone do repositório da SMartyModeling no Git tendo assim acesso ao código-fonte. Com o ambiente de desenvolvimento preparado, foi possível executá-lo usando o Apache NetBeans IDE.

Portanto, com o ambiente SMartyModeling pronto, foram executados alguns testes exploratórios a fim de conhecer a ferramenta e encontrar defeitos relacionados a falta de experiência no ambiente para usuários iniciantes. As ocorrências de defeitos encontrados no teste foram categorizadas por funcionalidades, resultando em 12 defeitos identificados:

- **Defeito 1 - Redimensionar Tela:** ao redimensionar a janela o menu também é redimensionado e são criadas barras de rolagem horizontal e vertical no menu. Após a correção, foi feita a criação apenas da barra horizontal, sendo assim, possível o acesso a todos os botões mesmo com janela pequena;
- **Defeito 2 - Redimensionar Tela:** ao redimensionar a janela não está criando barra de rolagem, tornando alguns componentes inacessíveis. Após a correção, ao redimensionar a tela são criadas barras de rolagem horizontal e vertical;
- **Defeito 3 - Salvar Projeto:** a opção de salvar projeto não funciona após abrir o `Save Project` uma vez e fechar a opção sem salvar o projeto. Após ajustes realizados pelo desenvolvedor, o erro persistiu. Foi observado que o ambiente controla o salvamento do projeto pela ação do botão `Save Project`, porém apenas deveria identificar que foi salvo após clique na janela de salvamento e caso não ocorressem falhas. Foi repassado ao programador para que seja feita a correção;
- **Defeito 4 - Salvar Projeto:** não existe uma opção `Save As` para poder salvar o projeto com outro nome sem sobrescrever o atual. Após reportado ao programador, foi criada a opção `Save As`;
- **Defeito 5 - Abrir Projeto:** ao acionar `Open Project` ou `Close Project` não é aberta opção para salvar o projeto. O usuário pode acabar perdendo o trabalho feito;
- **Defeito 6 - Salvar Projeto:** ao salvar o projeto não está sendo salvo o arquivo com a extensão padrão SMTY, assim, ao abrir um projeto já salvo é necessário alterar a opção `Files of Type` para "All Files". Foi repassado ao programador para que seja feita a correção;
- **Defeito 7 - Abrir Projeto:** ao acionar a opção `Open Project` está sendo aberta uma tela com nome `Save`, inclusive com botão `Save` ao invés de `Open`. Após correção, o nome da janela e do botão de abertura de projetos está com grafia correta;
- **Defeito 8 - Atributos dos Elementos:** alguns caracteres são conflitantes com a estrutura XML e ao salvar um elemento com algum desses caracteres no nome ocorre falha ao abrir o projeto. Após a correção, este erro foi corrigido;
- **Defeito 9 - Desfazer Ação:** a opção `Undo` não está funcionando. Foi apagado um relacionamento entre "Activities" e não foi possível desfazer a remoção;
- **Defeito 10 - Manipulação de Elementos:** após apagar um relacionamento do tipo *flow* não está sendo possível incluir o mesmo novamente. Após correção, está sendo possível excluir um relacionamento e posteriormente incluir novamente;

[6]Disponível em `https://git-scm.com`

- **Defeito 11 - Painel de Projetos:** foi identificada a ocorrência de uma falha na visualização do painel do projeto ao alterar o campo `Variation Point` em uma Variabilidade. Ao modificar o ponto de variação, o SMartyModeling não estava mostrando esta alteração no painel mesmo ao salvar o projeto. Após a correção, ao selecionar uma nova opção para o ponto de variação já é alterado também no painel do projeto; e
- **Defeito 12 - Manipulação de Elementos:** para organização e legibilidade do diagrama, é possível mover as anotações de tipo de associação. Porém, ao clicar em qualquer um dos elementos do diagrama o texto da anotação volta ao local original.

A Figura 3 apresenta em detalhes o **Defeito 11**. Na tela A, a `Variation Point` está diferente do que o mostrado no painel de projeto. Após a correção (B), a `Variation Point` está condizendo com o mostrado no painel.

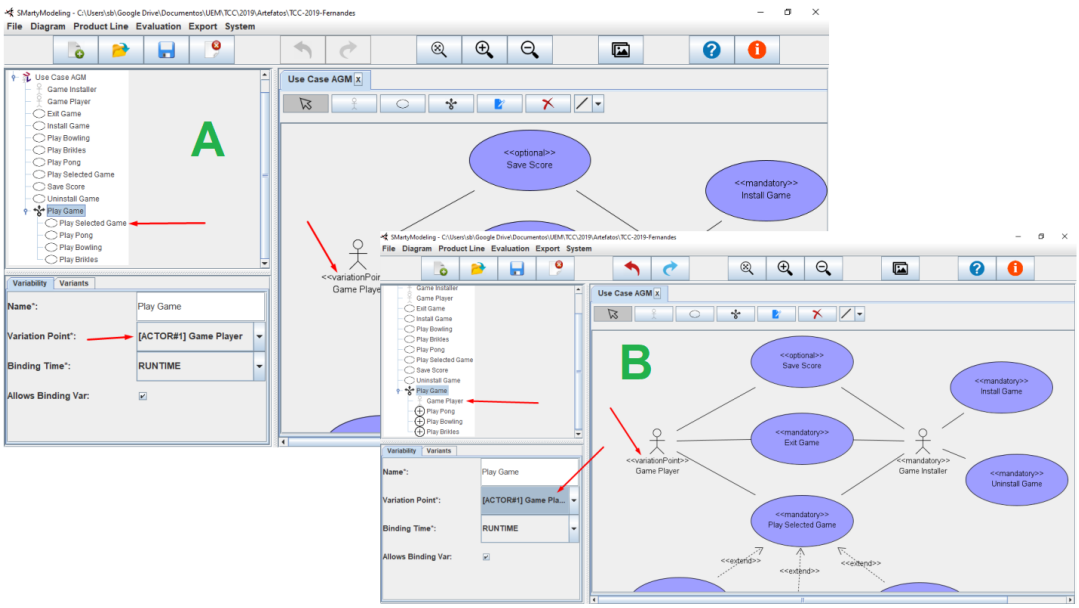

Figura 3. Detalhes do Defeito 11 (A) e após a correção (B).

5. Discussão dos Resultados

Como apresentado, os testes exploratórios resultaram em 12 defeitos no SMartyModeling, sendo categorizados de acordo com as funcionalidades relacionadas a "Salvar/Abrir Projeto", "Elementos dos Diagramas", "Dimensionamento da Janela", "Desfazer Ação" e "Painel de Projeto". A Tabela 1 apresenta os defeitos encontrados, classificando-os conforme a funcionalidade do ambiente.

Conforme a categorização dos defeitos, é possível observar que a considerável parte está concentrada nas funcionalidades implementadas no pacote responsável pela manipulação de arquivos. Portanto, como resultado inicial, podemos citar uma correção destas funcionalidades no ambiente, sendo realizada a correção de parte dos defeitos de maneira imediata, em especial, dos defeitos 1, 2, 4, 6, 7, 8, 10 e 11. Estas correções contribuíram significativamente para a melhoria e evolução da ferramenta.

Tabela 1. Funcionalidade por Identificador e Quantidade dos Defeitos.

Funcionalidade	Identificador dos Defeitos	Quantidade de Defeitos
Salvar/Abrir Projeto	3, 4, 5, 6 e 7	5
Elementos do Diagrama	8, 10 e 12	3
Redimensionar Tela	1 e 2	2
Desfazer Ação	9	1
Painel de Projeto	11	1

O teste exploratório ofereceu uma liberdade maior ao testador, de maneira que o testador interagiu com o SMartyModeling da maneira que considerou mais adequada e utilizou as funcionalidades que considerou importantes dentro do cenário de LPS. E mesmo diante dos defeitos identificados, o testador considerou o ambiente intuitivo, mesmo considerando um usuário com pouca familiaridade com modelagem LPS. O teste exploratório também permitiu, a partir de um ponto vista diferente do desenvolvedor, explorar as funcionalidades do ambiente em cenários alternativos, avaliar a capacidade de execução e encontrar pontos vulneráveis e suscetíveis a erros.

6. Conclusão

Este trabalho apresentou uma revisão sobre Teste de Software, LPS, descreveu em alto nível o ambiente SMartyModeling e descreveu o planejamento e execução de um teste exploratório realizado sobre o SMartyModeling, bem como os respectivos defeitos identificados. A principal contribuição deste trabalho é do ponto de vista prático, considerando a aplicação de técnicas de teste em um sistema real.

Como resultados obtidos deste trabalho, pode-se citar, principalmente, a identificação de falhas e a correção de defeitos no ambiente SMartyModeling. Estas correções contribuíram para a melhoria e evolução da ferramenta, com o objetivo principal de deixa-lá mais intuitiva para uso. O teste exploratório executado permitiu complementar os resultados dos testes unitários realizados durante o desenvolvimento e da avaliação inicial de usabilidade, detectando assim novos defeitos a partir do ponto de vista de um usuário sem experiência em utilizar o ambiente, porém familiarizado com a atividade de teste de software.

O teste exploratório realizado foi muito importante pelo fato de identificar uma série de defeitos que, assim que corrigidos, permitiram a correção e evolução do ambiente. Contudo, os resultados do teste exploratório reforçam a necessidade de realizar testes complementares, expandindo para as demais técnicas apresentadas na literatura e ampliando para cenários mais complexos.

Referências

Allian, A. P. (2016). VMTools-RA: a Reference Architecture for Software Variability Tools. Master's thesis, State University of Maringá. in portuguese.

Capilla, R., Bosch, J., and Kang, K. C. (2013). *Systems and Software Variability Management: Concepts, Tools and Experiences*. Springer.

Dijkstra, E. W. (1972). The humble programmer - ACM Turing Award Lecture. *Communications of the ACM*, 15(10):859–866.

Gomaa, H. (2006). *Designing Software Product Lines with UML 2.0: From Use Cases to Pattern-Based Software Architectures*.

ISTQB (2018). *Certified Tester Foundation Level Syllabus - International Software Testing Qualifications Board (ISTQB)*. URL: https://www.bstqb.org.br/.

Linden, F. J. V. D., Schmid, K., and Rommes, E. (2007). *Software product lines in action: The best industrial practice in product line engineering*, volume 20. Springer-Verlag New York, Inc.

Myers, G. J., Sandler, C., and Badgett, T. (2011). *The Art of Software Testing*. Wiley Publishing, 3rd edition.

OliveiraJr, E., Gimenes, I. M. S., Maldonado, J. C., Masiero, P. C., and Barroca, L. (2013). Systematic Evaluation of Software Product Line Architectures. *Journal of Universal Computer Science (JUCS)*, 19(1):25–52.

OliveiraJr, E., Maldonado, J. C., and Gimenes, I. M. S. (2010). Systematic Management of Variability in UML-based Software Product Lines. *Journal of Universal Computer Science (JUCS)*, pages 2374–2393.

Pfahl, D., Yin, H., Mäntylä, M. V., and Münch, J. (2014). How is exploratory testing used? a state-of-the-practice survey. In *ESEM*, pages 1–10, New York, NY, USA. ACM.

Pohl, K., Bockle, G., and van der Linden, F. (2005). *Software Product Line Engineering: Foundations, Principles, and Techniques*, volume 26. Springer-Verlag New York Inc., Secaucus, NJ, USA.

Silva, L. F. d. (2017). SMartyModeling: um Ambiente de Modelagem para Linha de Produto de Software com base no Eclipse Modeling Framework. Monografia (Bacharel em Informática), UEM (Universidade Estadual de Maringá), Maringá - PR, Brasil.

Sommerville, I. (2011). *Engenharia de Software*. Pearson Education do Brasil, São Paulo, 9 edition.

Ziadi, T. and Jezequel, J.-M. (2006). Software Product Line Engineering with the UML: Deriving Products. In *Software Product Lines*, pages 557–588. Springer, Berlin.