REVERSEJ: UMA FERRAMENTA PARA ENGENHARIA REVERSA BASEADA EM FEATURES

Lucas Veloso Schenatto[1]

Professor: Kleinner Silva Farias de Oliveira[2]

Resumo: Diagramas de classe e de sequência auxiliam na compreensão de sistemas, pois abstraem seu comportamento e arquitetura. Porém, manter tais diagramas atualizados a cada modificação feita no código exige muito esforço, além de ser uma atividade propensa a erros. A engenharia reversa estática permite gerar esses diagramas UML de maneira automatizada a partir da análise do código fonte, porém, ela não é capaz de abranger o comportamento do sistema quando o mesmo é orientado a objetos, devido à alta dinamicidade que ele assume durante a execução. A engenharia reversa em tempo de execução merece, portanto, ser mais aprofundada, de modo a gerar diagramas representando *features*. Este trabalho propõe a ReverseJ, uma ferramenta que gera os diagramas de classe e de sequência da UML para uma *feature*, a partir da análise de sua execução. Para isso faz-se uso de engenharia reversa em tempo de execução através da orientação a aspectos, para capturar o comportamento de um software na *feature* desejada. Tal ferramenta alcançou uma alta precisão na avaliação feita sobre os diagramas gerados.

Palavras-chave: engenharia reversa baseada em *feature*, programação orientada a aspectos, diagrama de classe, diagrama de sequência.

1 INTRODUÇÃO

Como ressaltado por Canfora e Di Penta (2007), lidar com a alta dinamicidade é um dos principais desafios da análise de programas atualmente. Muitas linguagens de programação amplamente utilizadas hoje permitem uma alta dinamicidade, o que constitui um mecanismo de desenvolvimento poderoso, mas torna a análise mais difícil. Um auxílio para a análise é a especificação do software, mas para uma boa compreensão do mesmo, a especificação apropriada é essencial. Porém, se os artefatos da especificação e design forem de baixa qualidade ou estiverem indisponíveis, a compreensão do funcionamento do software pode se tornar difícil, criando assim uma barreira para sua manutenção e aperfeiçoamento (MERDES; DORCH, 2006).

[1] Estudante de Análise e Desenvolvimento de Sistemas na Universidade do Vale do Rio dos Sinos, Porto Alegre, Brasil. E-mail: lucasvschenatto@hotmail.com.

[2] PhD e Professor no Programa Interdisciplinar de Pós-graduação em Computação Aplicada (PIPCA) na Universidade do Vale do Rio dos Sinos, São Leopoldo, Brasil. E-mail: kleinnerfarias@unisinos.br.

Tratando de artefatos de *design*, encontram-se a notação UML, que é uma alternativa amplamente utilizada para compreender melhor um sistema. Ela provê representações gráficas do mesmo, de modo que as principais informações são trazidas para um nível maior de abstração. Diagramas de classe, por exemplo, fazem parte do padrão UML, e contém informações preciosas da implementação do software. Essas informações fornecem a base tanto para a construção quanto para o entendimento do software, mas para isso, elas precisam estar disponíveis de forma clara e de fácil compreensão, além de estarem atualizadas.

Diagramas UML são feitos normalmente durante a fase de análise e *design* de um projeto, e tornam-se defasados quando há qualquer modificação posterior. Tais diagramas normalmente não são atualizados sempre que o código do sistema muda, assim deixando de representar o seu estado atual. Esses diagramas são de grande ajuda inclusive para quem faz a manutenção do sistema, possibilitando entender sua a arquitetura e localizar os lugares que requerem modificações. Não obstante, eles normalmente estão obsoletos e defasados em relação à concreta implementação do programa, quando sequer existem (DEMEYER; DUCASSE; NIERSTRASZ, 2000). A defasagem entre a implementação real e a documentada é ainda mais comum ao trabalhar com sistemas legados, pois ela tende a aumentar nesses cenários. Não obstante, para entender completamente um sistema legado orientado a objetos, é necessário ter informações atualizadas relativas à sua estrutura e comportamento.

Compreender o comportamento do sistema é especialmente importante em contextos onde ligações dinâmicas são amplamente usadas. Analisar o software em tempo de execução, ou seja, seu comportamento dinâmico, torna-se um complemento necessário para a análise estática (CANFORA; DI PENTA, 2007, p. 326-341). Todavia, quando não há nenhum modelo de *design* completo e consistente, precisamos recorrer à engenharia reversa para recuperar o máximo de informação possível, através das análises estática e dinâmica. Tratando-se da UML, os diagramas de classe e de sequência podem ser gerados com a engenharia reversa.

Diagramas produzidos automaticamente são similares aos produzidos manualmente, e mais precisos que aqueles normalmente produzidos (GUÉHÉNEUC, 2004, p. 28). Porém, artefatos de design feitos a partir da análise estática (baseada na leitura do código fonte) acabam não contemplando

características dinâmicas, como herança, herança múltipla, interfaces e polimorfismo, tão presentes nos sistemas utilizados atualmente. A análise de tais softwares torna-se complexa visto que, com o carregamento de classe em tempo de execução, não é possível determinar o conjunto de objetos para o qual uma referência aponta, de modo que a análise estática (baseada na leitura do código fonte) não consegue identificar essas informações.

Há ferramentas CASE[3] de UML que oferecem a capacidade de obter representações estáticas, através da engenharia reversa, a partir de um sistema orientado a objetos. Elas, no entanto, não são capazes de mapear um sistema orientado a objetos através da análise de *features,* obtida em tempo de execução. Esta é, portanto uma técnica que merece ser mais aprofundada. Analisar o comportamento é mais difícil do que analisar sua estrutura estática. Uma das principais razões é que, por causa das ligações dinâmicas, é difícil, e às vezes impossível saber, usando apenas o código fonte, o tipo dinâmico da referência de um objeto, e assim quais operações serão executadas. É necessário executar o sistema e monitorar sua execução, capturando informações significativas, para então sintetizar os diagramas desejados.

O objetivo desse trabalho é desenvolver uma ferramenta capaz de gerar os diagramas de classe e de sequência da UML que representem *features*, a partir da análise do fluxo de execução, na linguagem Java, utilizando orientação a aspectos através do AspectJ. Esses diagramas precisam conter uma representação mais detalhada do comportamento do software do que a obtida através da análise estática, e se aproximar do resultado esperado para a representação da *feature* analisada. É proposto que seja estudada a eficácia da ferramenta para gerar diagramas que representem as *features* analisadas, de modo que também amplie o conhecimento do uso de informações obtidas em tempo de execução para representar *features*.

Para ser avaliada, a ferramenta será utilizada na execução de *features* específicas de uma aplicação de calculadora (CALCULATOR, 2015) e de um sistema de folha de pagamento (PAYROLL, 2015) para produzir os respectivos diagramas de classe e de sequência dessas *features*. Os resultados da avaliação

[3] Astah,disponível em <http://astah.net/> e Enterprise Architect, disponível em <http://www.sparxsystems.com.au/products/ea/index.html>

apontam que a efetividade da ferramenta para dar suporte à compreensão das *features,* ao apresentar altas taxas de *precision*, *recall* e *f-measure*.

2 REFERENCIAL TEÓRICO

Este capítulo tem como objetivo apresentar a base teórica que será utilizada para a compreensão dos temas propostos por este trabalho. Para isso, a Seção 2.1 apresenta uma visão geral sobre engenharia reversa. A seção 2.2 trata sobre a programação orientada a aspectos. A seção 2.3 apresenta o AspectJ. Na seção 2.4 é abordado o tema de UML. A seção 2.5 descreve o SDMetrics, utilizado para comparar os diagramas. Por fim, a seção 2.6 apresenta as formulas utilizadas para a avaliação dos resultados.

2.1 Engenharia Reversa

Engenharia reversa de software é o oposto do processo tradicional de engenharia para desenvolvimento de software (NELSON, 1996). Muitas vezes desenvolvedores encontram-se em situações onde possuem o software, mas não a documentação do mesmo, o que leva ao uso de engenharia reversa. Para vários sistemas, os próprios proprietários não dispõem de documentação de suas especificações de projeto e arquitetura, pois foram feitos há muito tempo e se perderam, ou ficaram desatualizados. A única fonte de informação confiável nesses casos, quando disponível, é o próprio código-fonte. Diferentemente do mundo do hardware, a engenharia reversa de software não é tão utilizada na abstração de informações de um produto concorrente, mas sim daqueles que se possui a propriedade. Schach afirma (2010, p.527),

> Com muita frequência ao se fazer a manutenção de sistemas legados, a única documentação disponível para a manutenção pós-entrega é o próprio código-fonte, ou seja, o software que continua a ser usado atualmente, mas foi desenvolvido cerca de 15 ou 20 anos atrás, para não dizer antes disso. Sob tais circunstancias, fazer a manutenção do código pode ser extremamente difícil. Uma forma de lidar com esse problema é partir do código-fonte e tentar recriar os documentos de projeto ou até mesmo as especificações.

A engenharia reversa busca, portanto, recuperar as informações do projeto. Ela procura obter uma representação do sistema em um nível mais alto de

abstração, que facilite o entendimento de sua arquitetura e comportamento (PRESSMAN, 2010, p. 670). Essa representação do sistema, por sua vez, pode ser na forma gráfica, como o caso dos diagramas UML, muito usados no design e modelagem de sistemas.

Entre as abordagens usadas na análise do software, destacam-se a análise estática, e a análise dinâmica. A estática consiste na abstração de informações que podem ser encontradas no próprio código fonte ou código binário, sendo muito útil para obter a arquitetura estática do software (AHO et al, 1986). A dinâmica por sua vez foca na captura de informações geradas dinamicamente durante a execução do software, que podem variar entre execuções, e apontam o comportamento do mesmo (SYSTA, 1999). Como observado por Murphy, resultados muito distintos são obtidos de técnicas e ferramentas diferentes (MURPHY et al, 1998, p. 158-191).

Analisar sistemas enquanto estão executando nos ajuda a entender as interações entre os componentes do sistema, tipos de mensagens e protocolos utilizados e os recursos externos usados pelo sistema. (TILLEY, 1998). Dessa maneira é possível identificar as informações pertinentes a *features* específicas. Tais informações, que dão entendimento do sistema, são essenciais tanto para a manutenção, quanto para a reengenharia do mesmo. Ainda assim, é necessário contar com ferramentas adequadas para efetivamente acessar e interpretar informações dinâmicas do sistema, pois a quantidade, nível de detalhes e complexidade das estruturas dessas informações seriam esmagadoras (WALKER et al, 1998).

2.2 Programação orientada a aspectos

A programação orientada a aspectos (POA) é um novo paradigma de programação, que complementa e é usado em conjunto com a orientação a objetos. Através dela conseguimos modularizar interesses transversais, que na orientação a objetos estariam espalhados por toda a aplicação (KICZALES, 1996).

De acordo com Dijkstra (1974), um interesse é um requisito ou consideração específica de uma ou mais partes interessadas, que devem ser endereçados para satisfazer o objetivo geral do sistema. Interesses transversais são aqueles que estão espalhados por toda a aplicação, são funcionalidades dispersas em várias classes e módulos.

A POA permite modularizar interesses transversais e dispersos, separando-os por completo das classes e transformando-os em aspectos (unidades funcionais que contém *advices* e *pointcuts*, de maneira similar às classes encontradas na orientação a objetos, que contém campos e métodos) (WEBOPEDIA, 2014). Através disso, as classes originais são aliviadas do cargo de gerenciar funcionalidades que não constituem seu propósito principal.

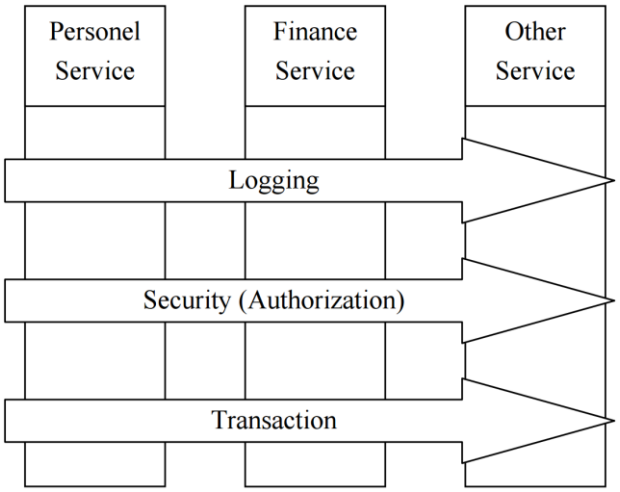
Na programação orientada a objetos (POO) não há como encapsular completamente um interesse transversal. O interesse principal (módulo) precisa referenciá-lo, e interagir com ele. Tendo um pouco do interesse transversal dentro de si, o módulo precisa se preocupar com ele. Numa situação ideal, porém, o módulo poderia ignorá-lo completamente (ROBINSON, 2007, p. 6).

Interesses transversais estão no coração da POA, e são a razão que levou à criação da mesma, pois a ideia básica de um aspecto é encapsular funcionalidades transversais fora do programa base, em módulos separados e bem definidos (RAJAN 2005, p. 59-68). Não é possível encapsulá-los usando apenas orientação a objetos, e isso causa problemas.

O código de um interesse transversal (que está espalhado e entrelaçado pelos módulos do sistema) pode parecer inofensivo, mas com o tempo tende a causar uma série de problemas. Esse código obscurece a funcionalidade central de um componente, dificultando o seu entendimento. Ele também é um obstáculo para a manutenção, pois qualquer mudança na interface de um interesse transversal pode implicar em atualizações em múltiplos lugares.

Além disso, o reuso é prejudicado, visto que estão entrelaçadas no código de um componente, inúmeras partes de um interesse transversal que pode não se aplicar ao propósito daquele que faz o reuso. A Figura 1 representa como interesses transversais se espalham pela aplicação.

Figura 1 – Exemplo de interesses transversais



Fonte JU et al. (2007).

É também difícil garantir que todo esse código espalhado esteja escrito consistentemente. Por estarem espalhados, os interesses transversais não têm um único responsável, e podem não ser implementados em todos os lugares em que deveriam. Cada programador precisa lidar com os interesses transversais enquanto tenta escrever a funcionalidade principal do componente, e todas as instancias do código espalhado precisam ser atualizadas se uma funcionalidade de interesse transversal é adicionada ou modificada.

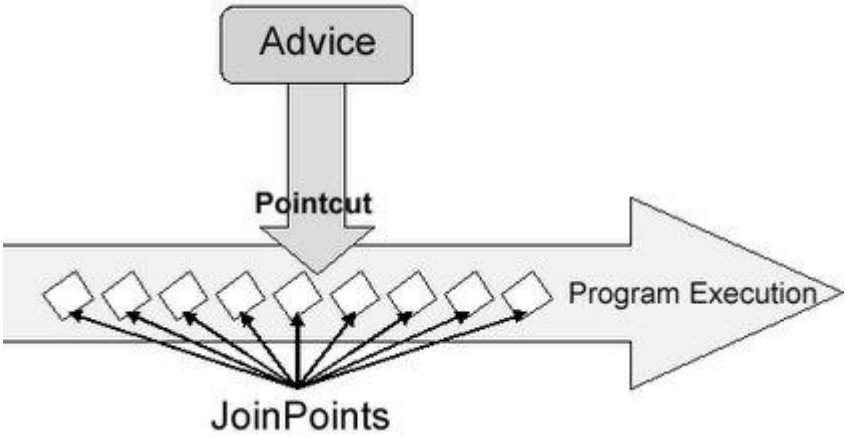
Ao usar a POA, os aspectos são injetados nos devidos lugares por um processo conhecido como *weaving*. Dependendo da implementação da linguagem de aspectos, o processo de *weaving* pode ocorrer em tempo de compilação, carregamento ou execução (KICZALES, 2005, p. 49-58). Kiczales (1996) ressalta que a programação orientada a aspectos permite aos programadores expressar cada um dos aspectos de interesse de um sistema de forma separada e natural, e então automaticamente combinar esses blocos separados em uma forma executável final usando uma ferramenta chamada *Aspect Weaver*.

Essencialmente, POA permite introduzir novas funcionalidades aos objetos sem que eles precisem ter qualquer conhecimento disso. É possível modularizar de maneira limpa tais funcionalidades, tornando fácil habilitar, manusear e desabilitar elas. Ela permite interceptar eventos durante a execução através de padrões escritos chamados *pointcuts*. Os eventos interceptados são chamados *joinpoints*. Sempre que um *pointcut* corresponde a um *joinpoint*, um *advice* (código extra) é executado. (AVGUSTINOV, 2007).

2.3 AspectJ

O AspectJ (GRADECKI, e LESIECKI, 2003), uma extensão de Java e *plugin* do Eclipse, é uma linguagem popular que implementa o conceito de orientação a aspectos (KICZALES et al., 2001). Ele implementa tais conceitos usando: aspectos, *advices*, e *joinpoints*, especificados através de *pointcuts*, efetuando o entrelaçamento em tempo de compilação (JOHNSON, 2005, p. 109). A Figura 2 ilustra os conceitos chave da POA em ação:

Figura 2 – Funcionamento da orientação a aspectos



Fonte: JAVATOUCH (2014).

O *advice* contém o interesse transversal que precisa ser aplicado. Os *join points* são os pontos dentro do fluxo de execução da aplicação que são candidatos a terem os advices aplicados. Os *pointcuts* definem em quais *joinpoints* o *advice* será aplicado. O conceito chave que tiramos disso é que os *poinitcuts* definem em quais *joinpoints* são aplicados os *advices*.

Um aspecto é semelhante a uma classe implementada no sistema, mas as principais funcionalidades que contém são os *advices* e *pointcuts* (STEIN et al, 2002, p. 109).

Traduzido do Inglês ele significa aviso ou informação, ou seja, ele informa à aplicação qual é o novo comportamento que o aspecto está introduzindo. O código de um *advice*, através dos *join points*, será executado antes, depois, ou no lugar de um comportamento, ou trecho de código existente, sendo esse entrelaçamento chamado *weaving* (WAND, 2004, p. 896-897).

Um *join point* é um ponto na execução da aplicação que pode sofrer a introdução de um *advice*. É importante ressaltar que essa introdução preserva o

código fonte intacto, pois ela acontece somente em tempo de compilação ou execução, e adiciona um novo comportamento no ponto especificado (WAND, 2004, p. 895).

Um *pointcut* é basicamente a regra que define quais são os pontos do sistema onde o *advice* do aspecto será introduzido. Nele pode-se: especificar nomes explícitos de classes e métodos e suas assinaturas, e inclusive usar de expressões regulares para identifica-los. Desse modo o aspecto é introduzido em todos os *joinpoints* desejados, e preserva o resto do sistema intacto (STOLZ e BODDEN, 2006, p. 115-116).

2.4 UML

A UML (*Unified Modeling Language*) é uma linguagem gráfica de modelagem reconhecida como padrão no mercado para modelar sistemas usando a orientação a objetos(RUMBAUGH et al., 1999). Como exemplo de sua utilização tem-se a fase de análise de um projeto, onde muitas informações são coletadas junto ao cliente para a obtenção dos requisitos do sistema. Essas informações fornecem a base para a construção do software, mas para isso, elas precisam estar disponíveis de forma clara e de fácil compreensão àqueles que irão ter de implementar essas especificações do sistema. Para tanto, na engenharia, são usados diagramas que representem o sistema a partir de diferentes perspectivas. Hoje, fazemos uso da UML para desenhar tais diagramas (LOBO, 2009, P. 18).

Em sua versão atual (UML 2.4.1), possui quatorze diagramas diferentes, sendo 7 de estrutura, 3 de comportamento e 4 de interação. Dentre eles, recebe destaque nesse trabalho os diagramas de classe e de sequência por fazerem parte do objeto do trabalho proposto.

2.4.1 Diagrama de classe

Um diagrama de classe é uma representação gráfica que mostra uma coleção de elementos declarativos, tais como classe, interfaces, tipos, assim como seus conteúdos e relacionamentos. (RUMBAUGH et al., 1999).

O diagrama de classe, como o nome já diz, mostra as classes do sistema, e pode conter diferentes níveis de detalhes de informações relacionadas a elas,

dependendo do propósito com que foi construído e de acordo com a fase em que o projeto se encontra. Para Fowler (2011, p. 52), o diagrama de classe representa o sistema, descrevendo os tipos de objetos e os tipos de relacionamentos estáticos que existem entre eles. Como ele serve de base para vários outros diagramas e é considerado como um dos principais, muitas vezes ele é utilizado como precursor no design do software. Um diagrama de classe pode conter as classes importantes de um sistema, seus atributos, métodos, e principalmente as associações entre as classes.

2.4.2 Diagrama de sequência

Um diagrama de sequência representa a sequência de ações que o sistema executa em um cenário específico dentro de um caso de uso. Para tanto, são representadas as mensagens trocadas entre os objetos do sistema, e entre o sistema e usuários externos. Alguns elementos utilizados para representar o diagrama de sequência são: atores, linha de vida, barra de ativação, mensagem (auto chamada, criação, chamada, deleção e retorno).

2.5 SDMetrics

O SDMetrics (SDMETRICS, 2015) é uma ferramenta para analisar qualitativamente arquivos UML através da mensuração de métricas pré-estabelecidas. Este conjunto de medidas cobrem as propriedades estruturais de elementos de *design* para todos os tipos de diagramas UML. Desta forma, a partir da leitura do arquivo UML é possível analisar a existência de possíveis problemas pertinentes ao modelo.

2.6 Medidas de avaliação

Como forma de mensurar a efetividade da ferramenta proposta, o estudo de avaliação dos resultados é baseado nas seguintes métricas obtidas da execução do software SDMetrics (SDMETRICS, 2015) sobre os diagramas UML gerados.

2.6.1 *Precision*

Esta métrica tem a finalidade de medir a precisão do experimento. Ela informa o percentual de elementos relevantes entre os que foram gerados. Isto é, entre os elementos que foram gerados em D_G, quais estão presentes no modelo esperado D_E. Porém, este não verifica se todos os elementos relevantes foram gerados. O valor resultante varia entre 0 e 1, e é calculado pela formula (FRAKES, 1992):

$$precision = \frac{| D_G \cap D_E |}{| D_G |}$$

2.6.2 *Recall*

A fórmula de *recall* analisa se todos os elementos relevantes foram gerados (FRAKES, 1992). Porém, esta não é uma verificação totalmente eficiente, pois, não são levados em conta os elementos gerados que não são relevantes. Seu resultado também está entre 0 e 1.

$$recall = \frac{| D_G \cap D_E |}{| D_E |}$$

2.6.3 *F-Measure*

Esta medida tem a finalidade de verificar a exatidão dos resultados gerados (FRAKES, 1992). Para isso, são levados em conta os resultados obtidos com as fórmulas de precisão e *recall*. Podendo ser interpretada como uma média ponderada entre estas duas.

$$F\text{-}Measure = \frac{2 \cdot precision \cdot recall}{precision + recall}$$

3 FERRAMENTA PROPOSTA

Esta seção apresenta a ferramenta ReverseJ. Para isso, a seção 3.1 apresenta o seu diagrama de componentes, a seção 3.2 descreve sua arquitetura

ferramenta. A seção 3.3 apresenta detalhes sobre como a ferramenta foi implementada. Por fim, a seção 3.4 traz mais detalhes sobre seu funcionamento.

O nome escolhido para a ferramenta (ReverseJ) é originado do conceito *reverse engineering* e do foco estabelecido em analisar software na linguagem Java. Ela foi desenvolvida no formato de *plugin* de desenvolvimento, para ser acoplada no projeto do software que se deseja analisar. O principal propósito é gerar uma representação gráfica da execução de uma *feature* do software, o que não é possível com a análise estática.

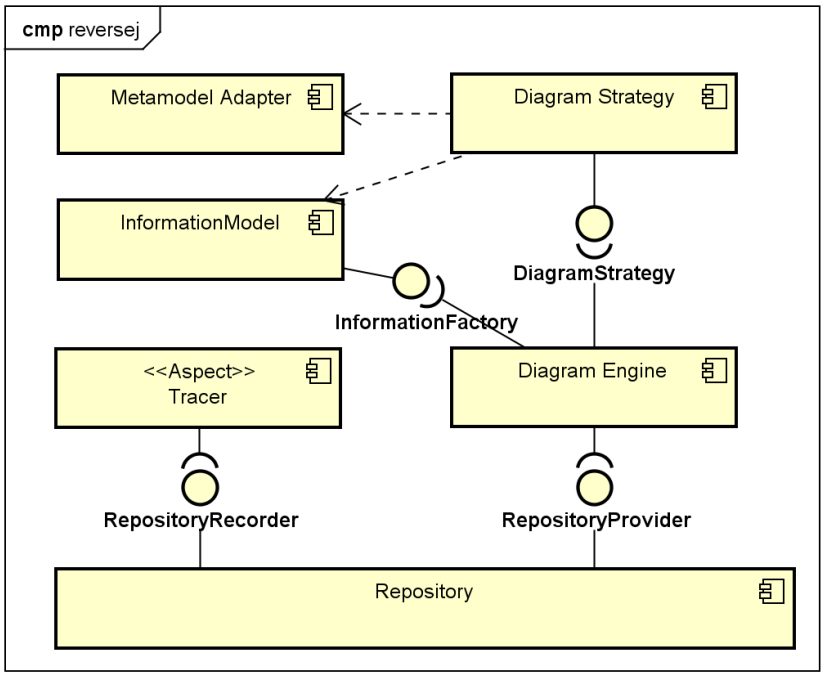
Os principais diferenciais da ferramenta proposta estão na utilização da orientação a aspectos, através do *plugin* AspectJ (ASPECTJ, 2015) e na análise de *feature* em tempo de execução. O uso da orientação a aspectos permite automatizar a análise de *features* da aplicação, sem que a aplicação tenha que se preocupar com essa análise. Essa instrumentalização feita sobre o código fonte através de aspectos preserva informações como os nomes das classes e métodos que seriam perdidas ao analisar código binário. Escolheu-se trabalhar em conjunto com a IDE Eclipse por suas extensões de modelagem que facilitam a criação e visualização de diagramas UML, e também pelo *plugin* do AspectJ que facilita seu uso.

A partir desta proposta, será apresenta a arquitetura de software utilizada no protótipo, mostrando como a ferramenta é constituída e os objetivos deste estudo. Serão também descritas suas características e atividades através da apresentação de seus diagramas de classe e de componentes, de modo a esclarecer seu funcionamento.

3.1 Diagrama de componentes

Partindo da ideia de construir uma ferramenta que, em suma, gere diagramas com escopo de *feature*, foram estipulados componentes os quais são responsáveis pela implementação das funcionalidades, e pela arquitetura da ferramenta. Essa forma de composição permite que os componentes sejam tratados de maneira independente, de modo a modularizar os elementos do projeto. Com isso, promovendo o reuso dos componentes produzidos e permitindo desenvolvimento e manutenções pontuais na ferramenta. Sendo assim, o diagrama contido na Figura 3 concentra-se em apresentar os componentes principais do processo.

Figura 3 – Diagrama de componentes



Fonte: elaborado pelo autor

O primeiro componente a atuar no processo é o *Tracer*. É ele que observa toda a execução do software sob análise e registra suas informações essenciais no *Repository*. Para tanto, ele conta com aspectos que atuam sobre todo o sistema sendo analisado.

Dentre os elementos da ferramenta, o *Repository* traz o conceito de repositório, e isola detalhes de persistência e leitura dos outros componentes. Além de armazenar os dados registrados pelo Tracer, ele desacopla o formato de entrada do formato de saída dos seus dados, de modo que um não depende do outro. Outra vantagem, é que fica transparente para a aplicação se eles são armazenados num banco de dados, ou sistema de arquivamento ou até mesmo em memória. Para o estudo de caso, os dados foram armazenados em memória.

O componente *Diagram Engine* possui conhecimento de como conectar todos os componentes necessários para gerar diagramas, porém não cuida dos detalhes de baixo nível de como fazê-lo. Essa informações ficam divididas nos componentes *Information Model, Diagram Strategy* e *Metamodel Adapter.* As políticas de alto nível desse componente para gerar diagramas estão completamente isoladas dos detalhes de baixo nível específicos para a criação dos mesmos.

Diagram Strategy contém as estratégias para gerar diagramas, e possui uma forte dependência com o *Information Model* justamente por depender de seu modelo

de informações para compor o diagrama. Outra forte dependência se dá com o *Metamodel Adapter* para se comunicar com o *framework* de criação de diagramas. Foram desenvolvidos nessa versão apenas os geradores de diagrama de classe e de sequência, mas a ferramenta suporta a implementação de quantos e quais geradores de modelos forem necessários, sem que a arquitetura seja alterada.

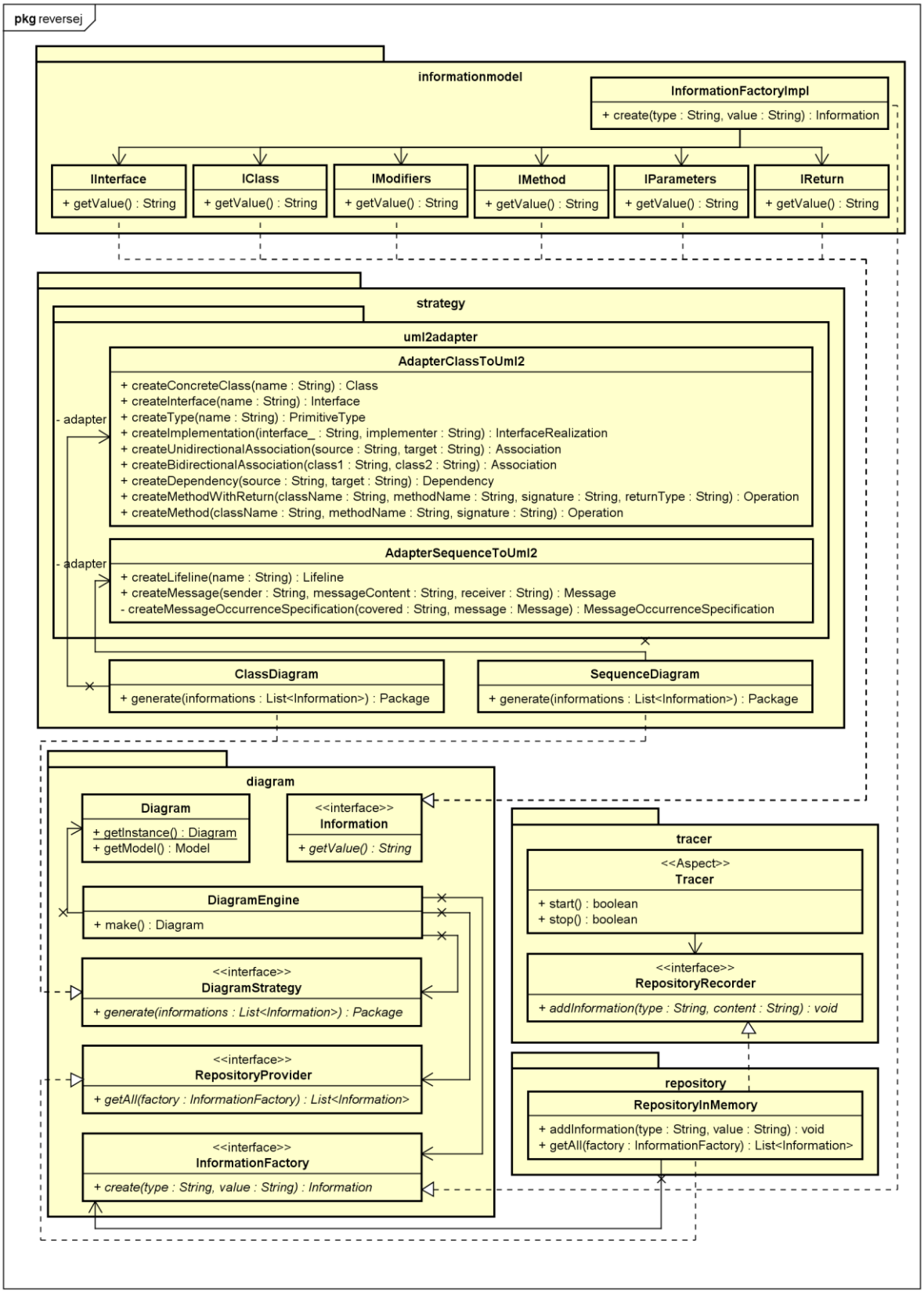
Information Model define o modelo das estruturas de dados com o qual as estratégias irão trabalhar. Informações provenientes do *Repository* tais como nomes de métodos, seus parâmetros e respectivas classes são estruturados de acordo com o definido nesse pacote.

Responsável pela comunicação direta com o framework de UML, o *MetaModelAdapter* estabelece uma ponte entre as estratégias de geração de diagramas do *Diagram Strategy* e o framework que vai de fato produzir tais diagrama. Foi escolhido o *framework* UML2 para agilizar o desenvolvimento, visto que o propósito não era a construção de diagramas UML em si, mas o conteúdo que deveria ser exibido nesse formato. Também é possível trabalhar com outros frameworks de modelagem, não ficando restrito ao UML2.

3.2 Visão lógica da arquitetura

O diagrama de classe exibe de forma mais detalhada as classes que formam o diagrama de componentes, isto é, como as classes estão estruturadas dentro da ferramenta, quais os métodos que elas implementam, assim como o modo que elas se relacionam. Na Figura 4, este diagrama é representado através de seis pacotes separados de acordo com suas funcionalidades. Cada um desses pacotes representa a composição interna de um componente. Para uma apresentação mais clara das funcionalidades das classes, seus atributos e métodos privados foram omitidos, bem como classes auxiliares irrelevantes para a compreensão da ferramenta.

Figura 4 – Diagrama de classe



Fonte: Elaborado pelo autor

3.3 Implementação

Como forma de exemplificar o modo que a ferramenta é executada e como as macro-atividades de registrar a execução e gerar os diagramas ocorrem, abaixo são exibidos alguns fragmentos do código fonte.

O trecho de código apresentado na Figura 5 foi extraído do aspecto *Tracer.aj*, que é o mecanismo utilizado para observar a execução do software sob análise. Sendo ele implementado utilizando a linguagem AspectJ, faz uso de *pointcuts* e *advices* para agir sobre o sistema sendo analisado. Isso acontece de maneira dinâmica, e sem que o sistema sob análise tenha qualquer conhecimento de sua atuação. Esses quatro *pointcuts* exibidos ilustram como ele age sobre todos os possíveis *joinpoints*, com exceção dos que precisam de imunidade, como é o caso do próprio código interno do aspecto. A partir dos *pointcuts*, os *advices* definidos registram todas as informações essenciais da execução do software.

Figura 5 – Principais *pointcuts* do *Tracer*

```
1.    pointcut methodCall():
2.          call(* *.*(..))&&immune();
3.
4.    pointcut methodExecution():
5.          execution(* *.*(..))&&immune();
6.
7.    pointcut constructorCall():
8.          call(*.new(..))&& immune();
9.
10.   pointcut constructorExecution():
11.         execution(*.new(..))&&immune();
```

Fonte: Elaborado pelo autor

O método da Figura 6 esclarece a abordagem para gerar todos os diagramas desejados a partir do mesmo conjunto de informações obtido pelo *Tracer*. Esse código pertence à classe *DiagramEngine.java*, a qual gera diagramas para todas as estratégias implementadas, ou seja, diferentes diagramas implementados.

Figura 6 – Gerador de diagramas

```
1.  public Diagram make() {
2.      for (DiagramStrategy diagram : strategies)
3.          diagram.generate(repository.getAll(factory));
4.      return Diagram.getInstance();
5.  }
```

Fonte: Elaborado pelo autor

Os dois métodos apresentados nas Figuras 7 e 8, por sua vez, são as implementações do método *generate*, chamado na linha três do trecho acima. Eles contêm as estratégias para gerar diagramas, sendo o primeiro pertencente à classe *ClassDiagram.java* e o segundo à classe *SequenceDiagram.java*. Eles possuem a responsabilidade de, a partir das informações fornecidas, gerar o diagrama de acordo com sua própria estratégia (diagrama de classe e diagrama de sequência respectivamente).

Figura 7 – Gerar diagrama de sequência

```
1.  @Override
2.  public void generate(List<Information> informations) {
3.      if(informations != null && !informations.isEmpty()){
4.          informations = addFooterAndHeader(informations);
5.          listLifelines(informations);
6.          arrangeMessages(informations);
7.          generateLifelines();
8.          generateMessages();
9.      }
10. }
```

Fonte: Elaborado pelo autor

Figura 8 – Gerar diagrama de classe

```
1.  @Override
2.  public void generate(List<Information> informations) {
3.      if(informations != null && !informations.isEmpty()){
4.          generateClasses(informations);
5.          generateInterfaces(informations);
6.          generateImplementations(informations);
7.          generateAssociations(informations);
8.          generateDependencies(informations);
9.          generateTypes(informations);
10.         generateMethods(informations);
11.     }
12. }
```

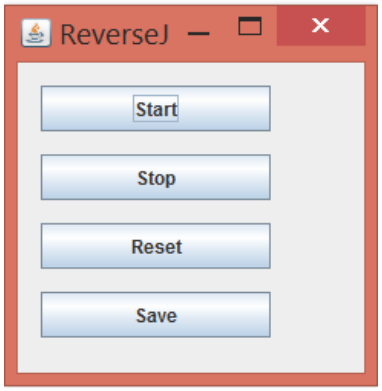
Fonte: Elaborado pelo autor

3.4 Funcionalidades

ReverseJ é capaz de produzir diagramas de classe e sequência, analisando a execução de uma *feature* do software, de forma a abranger exclusivamente a funcionalidade sendo analisada. O foco dessa análise é justamente informar os objetos específicos instanciados e chamados em tempo de execução, traçando o caminho que o sistema percorre em determinada *feature*. Cada diagrama, seja de classe ou sequência, corresponde ao comportamento e escopo do sistema para a *feature* executada, podendo conter informações distintas das obtidas com análise estática.

Para que o usuário tenha maior controle sobre a ferramenta, foi criada uma interface gráfica simples de controle, de modo que ele possa definir quando a ferramenta deve começar e parar a análise, resetar o processo, e salvar o diagrama gerado. Essa interface pode ser observada na Figura 9:

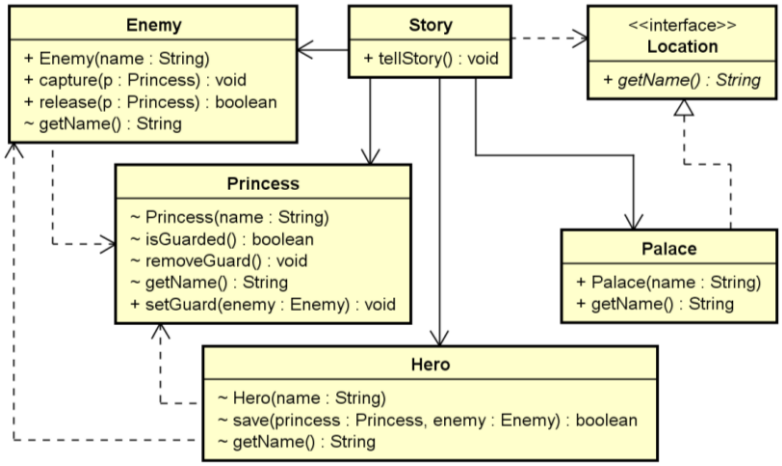
Figura 9 – Tela de controle



Fonte: Elaborado pelo autor

Após o usuário ter executado a análise e salvo o diagrama, ele pode ser exibido em ferramentas de exibição do formato *.uml*. Na Figura 10 é apresentado o diagrama de classe gerado como resultado da análise de uma pequena aplicação. Nele é possível identificar as classes e métodos utilizados na execução da *feature*.

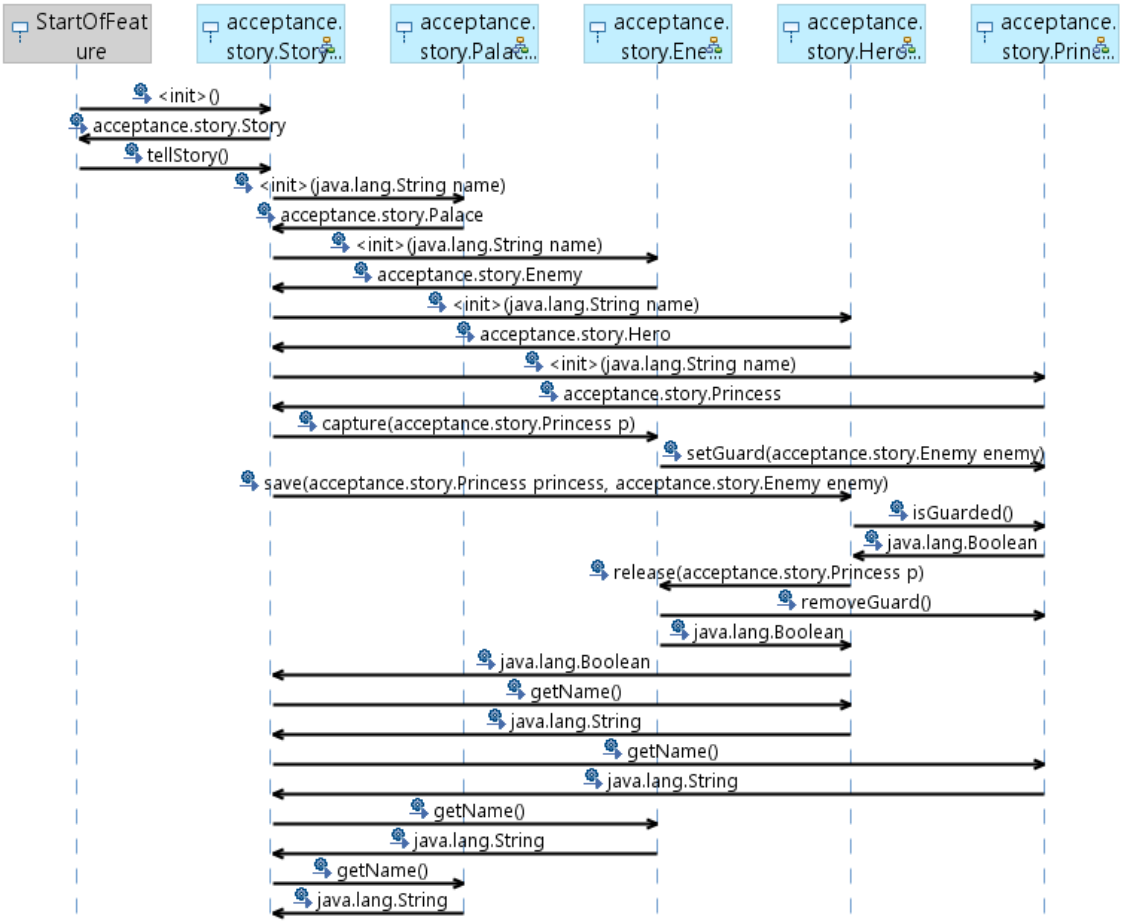
Figura 10 – Exemplo de diagrama de classe resultante



Fonte: Elaborado pelo autor

O diagrama de sequência gerado, como pode ser visto na Figura 11, evidencia as classes que compõem a *feature*, suas respectivas mensagens e a ordem em que eles ocorreram.

Figura 11 – Exemplo de diagrama de sequência resultante



Fonte: Elaborado pelo autor

4 AVALIAÇÃO DA FERRAMENTA

Essa seção tem como objetivo avaliar a ferramenta descrita na seção 3. Para isso, a seção 4.1 apresenta os procedimentos realizados na avaliação. A seção 4.2 apresenta o primeiro experimento, feito sobre um pequeno sistema de folha de pagamento, no qual é executada a *feature* de cadastrar usuário. A seção 4.3, por sua vez, apresenta o segundo experimento, realizado sobre um software de calculadora e utilizando a *feature* de soma.

4.1 Procedimentos de avaliação

Como forma de mensurar a efetividade da ferramenta ao produzir diagramas, este estudo fez uso de métricas coletadas através do software SDMetrics (SDMETRICS, 2015) a partir dos diagramas UML gerados. Sobre as métricas

geradas foram utilizadas três fórmulas para analisar a efetividade da ferramenta desenvolvida: *precision*, *recall* e *f-measure* (FRAKES, 1992).

Foram avaliados os diagramas gerados em dois experimentos, cada um relativo à execução de uma *feature* distinta, pertinentes a duas aplicações selecionadas, Payroll (PAYROLL, 2015) e Calculator (CALCULATOR, 2015). As características dessas aplicações são:

Tabela 1 – Caracteristicas das aplicações avaliadas

Aplicação	Linhas de código	Classes	Interfaces	Métodos
Payroll	1.775	41	7	214
Calculator	695	8	3	80

Fonte: Elaborado pelo autor

A primeira aplicação é referente ao experimento um, e trata-se de um sistema de folha de pagamento. A segunda aplicação é uma calculadora, sobre a qual foi efetuado o experimento dois.

Em cada experimento, foi aplicada a ferramenta ReverseJ para obter o diagrama gerado automaticamente $D_{G.}$ Para avalia-lo, o mesmo foi comparado com o diagrama esperado D_E, produzido manualmente. Essa comparação deu-se através das métricas coletadas dos mesmos, através do software SDMetrics (SDMETRICS, 2015), com o intuito de mensurar a eficácia do modelo gerado.

Optou-se, para este trabalho, por avaliar apenas o diagrama de classe. Isso porque o diagrama de sequência esperado deveria ser feito manualmente, e o grande volume de mensagens trocadas numa *feature* torna sua produção muito propensa a erros. Também, não foi realizada uma comparação com diagramas gerados por outras ferramentas com análise estática porque eles não representam *features*, inviabilizando a comparação.

Já em relação às métricas consideradas nesse estudo, elas são propriedades relevantes do diagrama de classe, e são apresentadas na Tabela 2.

Tabela 2 - Métricas

Métrica	Descrição
#Classes	Métrica referente ao número de *classes* presentes no diagrama
#OpsClasses	Métrica correspondente ao número total de operações referentes a todas as *classes*
#Interfaces	Métrica referente ao número de *interfaces* presentes no diagrama

#OpsInterfaces	Métrica correspondente ao número total de operações referentes a todas as *interfaces*
#Relações	Métrica que contabiliza o número de relações presentes no diagrama

Fonte: Elaborado pelo autor

Nos capítulos 5.2 e 5.3 são apresentados os experimentos realizados, bem como seus respectivos resultados.

4.2 Experimento 1

Para este experimento foi executada a *feature* de cadastrar funcionário comissionado, da aplicação Payroll. O resultado obtido da comparação entre D_E e D_G aponta uma grande similaridade entre ambos diagramas, com uma pequena divergência apenas nos números de classes e operações. Os resultados podem ser observados na Tabela 3.

Tabela 3 – Avaliação da ferramenta – 1º Experimento

Métrica	**D_E**	**D_G**	**$D_E \cap D_G$**	***Precision***	***Recall***	***F-Measure***
#Classes	12	13	12	1	0,92	0,96
#OpsClasses	28	29	28	1	0,96	0,98
#Interfaces	4	4	4	1	1	1
#OpsInterfaces	4	4	4	1	1	1
#Relações	12	12	12	1	1	1

Fonte: Elaborado pelo autor

Essa divergência foi ocasionada pelo fato de que em tempo de execução o Java pode processar operações implícitas irrelevantes à *feature*, mas registradas pela ferramenta. Nesse caso, a execução continha a inicialização de um objeto cuja classe não possui um construtor explícito mas, herda de outra classe através da instrução *extends*. Em tempo de compilação o Java criou um construtor implícito, o qual chamou o construtor da classe pai. Essa classe pai e seu construtor não foram incluídas em D_E, pois não agregam valor e não fazem parte da *feature*, embora tenham sido executados e registrados em D_G.

4.3 Experimento 2

Já para este experimento, foi executada a *feature* de soma, da aplicação Calculator. Nele, foram identificadas com exatidão todas as *classes* e *interfaces* que compõem a *feature* executada, bem como todas as relações presentes na *feature,* o que não poderiam ser identificados através da análise estática. Tais resultados ficam evidente nas fórmulas de *precision, recall* e *F-Measure*, apresentados na Tabela 4.

Tabela 4 – Avaliação da ferramenta – 2º Experimento

Métrica	**D_E**	**D_G**	**$D_E \cap D_G$**	***Precision***	***Recall***	***F-Measure***
#Classes	6	6	6	1	1	1
#OpsClasses	30	34	30	1	0,88	0,94
#Interfaces	2	2	2	1	1	1
#OpsInterfaces	3	3	3	1	1	1
#Relações	13	13	13	1	1	1

Fonte: Elaborado pelo autor

Há, porém, uma divergência no número de operações das classes. Ela foi ocasionada pela tela da aplicação, na qual foram configurados *ActionListeners* para os eventos de seus botões. Esses *listeners* foram configurados através de classes anônimas, e para acessar o conteúdo delas, o Java criou métodos em tempo de compilação para serem chamados no disparo dos eventos. A diferença de quatro operações corresponde exatamente aos quatro botões utilizados para executar a *feature*. Os *ActionListeners* foram configurados no bloco de código da classe da tela, mas os métodos de acesso deles foram gerados automaticamente, e não são relevantes para a *feature*, portanto, não foram considerados em D_E.

5. TRABALHOS RELACIONADOS

Esta seção tem como objetivo sumarizar os trabalhos encontrados na literatura que propõem uma abordagem similar ou relacionada ao trabalho proposto. A análise é apresentada nas próximas quatro seções. Por fim, na seção 3.5, é apresentada uma comparação entre os trabalhos relacionados e a ferramenta proposta.

5.1 Towards the Reverse Engineering of UML Sequence Diagrams for Distributed, Multithreaded Java software

O trabalho apresentado por Briand et al (2004) realiza a engenharia reversa de um sistema distribuído na linguagem JAVA, do qual se dispunha do código fonte, e tendo como produto diagramas de sequência. Partindo de uma abordagem muito similar a este trabalho, ele propõe que uma análise dinâmica seja feita sobre o software, para que assim possam ser identificadas informações que só aparecem em tempo de execução. Para isso, o autor decidiu fazer uso da orientação a aspectos através do AspectJ.

Briand apresentou a configuração dos aspectos para capturar a execução do programa, mas não apresentou a arquitetura da ferramenta. Ele também usou um sistema distribuído no caso de estudo que realizou, dando portanto muito foco na temporização das mensagens trocadas em cada componente do sistema.

5.2 Experiences with the Development of a Reverse Engineering Tool for UML Sequence Diagrams: A Case Study in Modern Java Development

Merdes e Dorch (2006) realizaram um estudo bibliográfico das principais técnicas e abordagens de engenharia reversa de software Java para a obtenção de diagramas de sequência UML. Segundo os autores, há duas principais divisões de métodos para a realização de engenharia reversa: métodos de tempo de execução (dinâmicos), e métodos em tempo de desenvolvimento (estáticos), podendo cada um deles ser baseados em código fonte ou em código binário. Ao analisar várias ferramentas de engenharia reversa, são mapeadas várias vantagens e desvantagens de cada uma delas. Além disso, os autores relatam a própria experiência ao desenvolver uma ferramenta de engenharia reversa.

Nesse trabalho eles mostram que Java é uma linguagem apropriada para desenvolver ferramentas de engenharia reversa em dois principais aspectos: o mecanismo de execução, baseado na máquina virtual, provê um suporte excelente para capturar coleções de dados; e as muitas funcionalidades avançadas de Java, com o rico acervo existente de bibliotecas facilitam o desenvolvimento de ferramentas como um todo. Segundo os autores, Java é uma tecnologia que possibilita diferentes técnicas de engenharia reversa, e é uma poderosa linguagem

para implementar tais ferramentas. Ela provê acesso para as informações necessárias durante a execução e é capaz de processar tais informações.

5.3 Dynamic Analysis For Reverse Engineering and Program Understanding

Nesse trabalho, Stroulia e Systä (2002) tratam do papel da engenharia reversa em compreender sistemas legados, sendo que muitos deles foram desenvolvidos orientados a objetos. Eles ressaltam a necessidade de compreender a arquitetura e comportamento dos mesmos, indicando a engenharia reversa como alternativa para essa questão.

Nesse artigo, há grande ênfase na importância da análise do comportamento dinâmico de sistemas, devido a sua ligação com a compreensão do processo e uso do mesmo. Os autores, além de prover uma visão geral das técnicas normalmente usadas de engenharia reversa, demonstram que a análise dinâmica precisa ser aprofundada, com o objetivo de suprir a necessidade de compreender sistemas orientados a objetos que carecem de documentação.

5.4 Understanding Web Applications through Dynamic Analysis

O trabalho apresentado por Antoniol et al (2004) apresenta uma ferramenta chamada WANDA que instrumentaliza aplicações web e combina informações estáticas e dinâmicas para recuperar a arquitetura real e, em geral, a documentação UML da aplicação propriamente dita. Além de implementar a ferramenta, ela foi testada em diversas aplicações WEB. Sua arquitetura tem sido concebida para permitir fácil customização e extensão.

5.5 Comparação dos trabalhos relacionados

Com o objetivo de facilitar a comparação dos trabalhos relacionados, foi desenvolvida a Tabela 5. Nela é possível a identificação de uma forma mais simples e compacta das características e tecnologias utilizadas em cada um dos trabalhos, possibilitando a identificação de características e lacunas relevantes que foram consideradas no trabalho desenvolvido.

Tabela 5 – Comparação dos trabalhos

	Towards the Reverse Engineering of UML Sequence Diagrams for Distributed, Multithreaded Java software	**Experiences with the Development of a Reverse Engineering Tool for UML Sequence Diagrams: A Case Study in Modern Java Development**	**Dynamic Analysis For Reverse Engineering and Program Understanding**	**Understanding Web Applications through Dynamic Analysis**	**ReverseJ**
Usa POA	Sim	Sim	Não	Não	Sim
Linguagem	Java	Java	Java	Não informado	Java
Usa AspectJ	Sim	Não	Não	Não	Sim
Tipo de análise	Dinâmica	Dinâmica	Dinâmica	Dinâmica e Estática	Dinâmica
Objeto da análise	Código binário	Código binário	Código fonte	Código fonte	Código fonte
Gera diagrama de classe	Não	Não	Não	Sim	Sim
Gera diagrama de sequência	Sim	Sim	Sim	Sim	Sim
Gera diagrama com escopo de *feature*	Sim	Sim	Sim	Sim	Sim

Fonte: Elaborado pelo autor.

De um modo geral, a maioria dos trabalhos relacionados fazem uso da análise dinâmica como meio de realizar a engenharia reversa. No entanto, mesmo os que geram diagramas em tempo de execução, o que se assemelha aos diagramas com escopo de *feature,* não o fazem usando o AspectJ. Além disso, os diagramas com escopo de *feature* que eles geram são apenas de sequência, não dispondo do equivalente para o diagrama de classe. A Ferramenta ReverseJ, por sua vez, gera diagramas de classes e de sequência em tempo de execução, tendo o AspectJ como mecanismo para a análise.

6 CONCLUSÃO

O objetivo deste trabalho foi desenvolver uma ferramenta capaz de gerar diagramas de classe e de sequência da UML representando uma *feature*, a partir da análise do fluxo de execução. A linguagem escolhida foi Java, e fez uso da orientação a aspectos através do AspectJ para capturar as informações da execução. Para desenvolver tal ferramenta, foi realizado um estudo sobre engenharia reversa, aspectos e UML. Após a implementação da ReverseJ, a mesma foi avaliada utilizando o SDMetrics para comparar os resultados obtidos com os esperados.

Ao longo desse trabalho, percebeu-se que a representação automática de *features* através de diagramas encontra boa metodologia na análise de tempo de execução. Diagramas de classe gerados em tempo de execução possuem informações muito próximas do real escopo das *features*, porém, tratando da linguagem Java, há informações que mesmo em tempo de execução divergem do que é esperado. Ainda assim, os resultados obtidos apontam uma ótima precisão nas informações obtidas, apresentando uma pequena margem de falha em *recall*. Com isso, foi identificado espaço para melhorias na ferramenta ReverseJ em relação ao tratamento das informações, e potencial para gerar outros diagramas que podem ser úteis na análise de *features.*

Desta forma, como trabalhos futuros, são apontadas possíveis melhorias no tratamento das informações obtidas em tempo de execução, de modo que levem em conta as peculiaridades implícitas do tempo de compilação e execução, além de identificar os atributos utilizados na *feature*, visto que tal funcionalidade não foi implementada nesse trabalho. Outro possível trabalho futuro é ampliar o uso das informações coletadas para gerar mais diagramas além dos diagramas de classe e de sequência, bem como fazer uma avaliação do diagrama de sequência. Também, pode ser de grande valor avaliar a qualidade dos diagramas gerados em comparação não somente com o resultado esperado, mas também com diagramas gerados por outras ferramentas.

REVERSEJ: A TOOL FOR REVERSE ENGINEERING FEATURES

Abstract: Class and sequence diagrams support understanding software, for they do abstract its behavior and architecture. However, keeping such diagrams updated on each code's change requires much effort, and it is likely to have errors. Static reverse engineering enables generating this UML diagrams automatically from source code analysis, but it cannot comprehend systems behavior when it is object oriented, because of its high dynamicity during execution. Reverse engineering at execution time is worth to studying more, in order to generate diagrams representing features. This work presents ReverseJ, a tool for generating class and sequence UML diagrams for features, from execution analysis. With this purpose, it reverse engineers in execution time thru aspect-oriented programming to capture behavior of a software in a specific feature. When evaluating this tool's generated diagrams, it has shown a high precision on the results.

Keywords: Reverse engineering of features, aspect oriented programming, class diagram, sequence diagram.

REFERÊNCIAS

AHO, A.; SETHI, R.; e ULLMAN, J. **Compilers: principles, Tools and Techniques.** Harlow, England: Addison-Wesley, 2nd edition, 1986.

ANTONIOL, G. ; DI PENTA, M. ; ZAZZARA, M. **Understanding Web applications through dynamic analysis**. Em: 12th IEEE International Workshop on Program Comprehension. IEEE. 2004, p. 120-129.

ASPECTJ. Disponível em: <https://eclipse.org/aspectj/>. Acesso em 15/06/2015.

ASTAH. Disponível em <http://astah.net/>. Acesso em 20/102015.

AVGUSTINOV, Pavel; HAJIYEV, Elnar; ONGKINGCO, Neil; DE MORE, Oege; SERENI, Damien; TIBBLE, Julian; VERBAERE, Mathieu. **Semantics of Static Pointcuts in AspectJ**. Em: Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. Nice, France, 2007, p. 11-23.

BOOCH, Grady. **Object-Oriented Analysis and Design with applications, 1st ed.** Benjamin Cummings, 1991.

BRIAND, L. C. ; LABICHE, Y. ; LEDUC, J. **Towards the Reverse Engineering of UML Sequence Diagrams for Distributed, Multithreaded Java software**. Em:

IEEE Transactions on Software Engineering, volume: 32, Issue: 9. Ottawa, Canada, 2006, p. 642-663.

CALCULATOR. Disponível em: < https://github.com/nando1/calulator>. Acesso em 15 de outubro de 2015.

CANFORA, Gerardo; DI PENTA, Massimiliano. **New Frontiers of Software Engineering**. Em: INTERNATIONAL CONFERENCE OF SOFTWARE ENGINNERING, Minneapolis MN, 2007, p. 326-341.

CHIKOFSKY, E.; CROSS J. **Reverse Engineering and Design Recovery: A Taxonomy**. Em: Software IEEE. 1994, p. 13-17.

DEMEYER, Serge; DUCASSE, Stéphane; NIERSTRASZ, Oscar. **Finding refactorings via change metrics**. Em: Doug Lea, editor, proceedings of 15th conference on Object-Oriented Programming Systems, Languages and Applications, Oct. 2000, ACM Press, p. 166–177.

DIJKSTRA, Edsger. **On The Role of Scientific Thought. 1974**. Disponível em: <http://www.cs.utexas.edu/users/EWD/ewd04xx/EWD447.PDF>. Acesso em: 10 novembro 2014.

ENTERPRISE ARCHITECT. Disponível em: <http://www.sparxsystems.com.au/products/ea/index.html>. Acesso em 20/10/2015.

FOWLER, Martin. **UML essencial** um breve guia para linguagem padrão. Porto Alegre: Bookman, 2011, p. 52.

FRAKES W. B.; MAEZA-YATES R. **Information Retrieval: Data Structures and Algorithms**. Prentice-Hall, 1992.

GRADECKI, Joseph D.; LESIECKI, Nicholas. **Mastering AspectJ** Aspect-Oriented Programming in Java TM. Editora: Wiley Publishing, Inc. Indianapolis, USA. 2003, p. 14-20.

GUÉHÉNEUC, Yann-Gaël. **A Reverse Engineering Tool for Precise Class Diagrams**. Em: Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research. Cascon, 2004, p. 28.

JACOBSON, I. **Object-Oriented Software Engineering – A Use Case Driven Approach.** Addison-Wesley, 1992.

JAVATOUCH. **Introduction to AOP**. Disponível em: <https://sites.google.com/site/javatouch/introductiontoaop>. Acesso em 25 outubro 2014.

JOHNSON, Rod. **J2EE development frameworks.** Em: Computer, volume 38, Issue 1. Janeiro 2005 pág. 107-110.

JU, Ke; Bo, Jiang. **Applying IoC and AOP to the Architecture of Reflective Middleware.** Em: Network and Parallel Computing Workshops, NPC Workshops, IFIP International Conference, 2007.

KICZALES Gregor. **Aspect-oriented programming.** Em: Journal ACM Computing Surveys (CSUR) – Special issue: position statements on strategic directions in computing research, volume 28, issue 4es, article 154, 1996.

KICZALES, Gregor; HILSDALE, Erik; HUGUNIN, Jim; KERSTEN, Mik; PALM, Jeffrey; GRISWOLD, William. **An overview of AspectJ**. Em J. Lindskov Knidsen, editor, Proceedings of ECOOP, volume 2072 of Lecture Notes in Computer Science. Springer-Verlag, p. 327-353, 2001.

KICZALES, Gregor; MEZINI, M. **Aspect-Oriented Programming and Modular Reasoning**. Em Proceedings of international conference of Software engineering. St. Louis USA, 2005, p. 49-58.

LOBO, Edson J. R. **Guia prático de engenharia de software**. Editora: Universo dos livros editora Ltda. 2009, p. 18.

MERDES, Matthias; DORCH, Dirk. **Experiences with the development of a reverse engineering tool for UML sequence diagrams:** a case study in modern Java development. Em: Proceedings of the 4th international symposium on Principles and practice of programming in Java, 2006, p. 125-134.

MURPHY, G.; NOTKIN, D.; GRISWOLD, W.; e LAN E. **An Empirical Study of Static Call Graph Extractors.** Em ACM Softw. Eng. Methodol., 7, 2, 1998. p. 158-191).

Nelson, Michael. **A survey of reverse engineering and program comprehension**. Disponível em: <http://arxiv.org/abs/cs/0503068>. Acesso em 12/07/2015.

PAYROLL. Disponível em: < https://github.com/grochon/payroll>. Acesso em 15 de outubro de 2015.

PRESSMAN, Roger S. **Engenharia de software**. Porto Alegre: ArtMed, 2010, p. 670.

RAJAN Hridesh; SULLIVAN Kevin. **Classpects: Unifying aspect and object-oriented language design.** Em: Proceedings of the 27th internacional conference on software engineering. St. Lois USA, 2005, p. 59-68.

ROBINSON, David. **Aspect-oriented programming with the e verification language:** a pragmatic guide for testbench developers. Editora: Elsevier, 2007, p. 6.

RUMBAUGH, J.; JACOBSON J.; e BOOCH G. **The Unified Modeling Language Reference Manual.** Reading USA: Addison-Wesley, 1999, capítulo 12.

RUMBAUGH, J.; BLAHA, M.; PREMERLANI, W.; EDDY, F.; e LORENSEN, W. **Object-Oriented Modeling and Design**. Englewood Cliffis USA: Prentice Hall, 1991.

SCHACH, Stephen R. **Engenharia de software 7**. Porto Alegre: ArtMed, 2010, p.527.

SDMETRICS. Disponível em: <http://www.sdmetrics.com/>. Acesso em 20 de outubro de 2015.

STEIN, Dominik; HANENBERG, Stefan; UNLAND, Rainer. **A UML-based Aspect-Oriented Design Notation for AspectJ**. Em Proceedings of the 1st international conference on Aspect-oriented software development, ACM. New York, USA: 2002, p. 106-122.

STOLZ, Volker; BODDEN, Eric. **Temporal Assertions using AspectJ**. Em: Eletronic Notes in Theoretical computer Science (ENTCS), Volume 144 Issue 4. Amsterdam, Netherlands, 2006, p. 109-124. Disponível em: <http://www.sciencedirect.com/science/article/pii/S1571066106003069>. Acesso em: 12 outubro 2015.

STROULIA, Eleni; SYSTÄ, Tarja. **Dynamic analysis for reverse engineering and program understanding**. Em: ACM SIGAPP Applied Computing Review, volume 10, Issue 1. New York USA: ACM, 2002, p. 8 - 17.

SYSTÄ, TARJA. **On the Relationships Between Static and Dynamic Models in Reverse Engineering Java Software.** Em Sixth Working Conference on Reverse Engineering. Atlanta, GA, 1999, p. 304-313.

TILLEY, Scott. **A reverse engineering environment framework.** Carnegie Mellon University, Software Engineering Institute. Pittsburgh, 1998.

WALKER, R.J.; MURPHY G.C.; FREEMAN-BENSON, B.; WRIGHT, D.; SWANSON, D.; ISAAK, J. **Visualizing Dynamic Softare System Information Through High-level Models.** Em Proceedings of OOPSLA '98. Vancouver, 1998, p. 271-283.

WAND, Mitchell; KICZALES, Gregor; DUTCHYN, Christopher. **A semantics for advice and dynamic join points in aspect-oriented programming.** Em: ACM Transactions on Programming Languages and Systems (TOPLAS), Volume 26, Issue 5, 2004, p. 890-891.

WEBOPEDIA. **Aspects.** Disponível em: <http://www.webopedia.com/TERM/A/aspects.html>. Acesso em: 31 outubro 2014.