UNIVERSIDADE DO VALE DO RIO DOS SINOS - UNISINOS
UNIDADE ACADÊMICA DE GRADUAÇÃO
CURSO DE SISTEMAS DE INFORMAÇÃO

LUCAS VELOSO SCHENATTO

ENGENHARIA REVERSA CONTÍNUA: AUTOMATIZANDO DIAGRAMAS UML COMO PARTE DA INTEGRAÇÃO CONTÍNUA ATRAVÉS DA UMLREV

Porto Alegre
2017

LUCAS VELOSO SCHENATTO

ENGENHARIA REVERSA CONTÍNUA: AUTOMATIZANDO DIAGRAMAS UML COMO PARTE DA INTEGRAÇÃO CONTÍNUA ATRAVÉS DA UMLREV

Trabalho de Conclusão de Curso apresentado como requisito parcial para obtenção do título de Bacharel em Sistemas de Informação, pelo Curso de Sistemas de Informação da Universidade do Vale do Rio dos Sinos - UNISINOS

Orientador: Prof. Kleinner Silva Farias de Oliveira

Porto Alegre

2017

ENGENHARIA REVERSA CONTÍNUA: AUTOMATIZANDO DIAGRAMAS UML COMO PARTE DA INTEGRAÇÃO CONTÍNUA ATRAVÉS DA UMLREV

Lucas Veloso Schenatto[1]

Orientador: Kleinner Silva Farias de Oliveira[2]

Resumo: A integração contínua está alcançando uma adoção cada vez maior no mercado devido aos benefícios em ter sempre um sistema funcionando a cada nova versão. Conforme novas versões do software são produzidas sua arquitetura muda, dificultando a manutenção da documentação técnica, que pode ser de grande valia para os próprios desenvolvedores. Mesmo com a versatilidade dos servidores de integração contínua, os mesmos não realizam de forma automática a confecção de diagramas. Este trabalho propõe uma abordagem para gerar diagramas UML de classes e sequência, e automaticamente atualizá-los a cada nova versão do software, realiza uma pesquisa com desenvolvedores de software, para avaliar sua percepção quanto aos benefícios da mesma e usa esta como hipótese de trabalho. Uma ferramenta capaz de produzir diagramas UML a partir da análise da execução de testes de integração foi desenvolvida, e inserida em um *pipeline* de integração contínua. Paralelamente foi realizado um questionário contendo 21 questões relacionadas à proposta do trabalho. Os resultados da pesquisa evidenciam a percepção de baixa efetividade da UML da maneira que é usada hoje. Também mostram uma percepção de que a UML seria mais amplamente utilizada no mercado com o uso da abordagem proposta, e que a mesma traz benefícios para os projetos de software.

Palavras-chave: Engenharia reversa. Integração contínua. Testes de integração. UML. Programação orientada a aspectos.

1 INTRODUÇÃO

A Integração contínua (CI), originalmente concebida como parte de um conjunto de práticas que forma a metodologia Extreme Programming (BECK, 2004), está alcançando uma adoção cada vez maior no mercado. Entre suas atribuições, os sistemas de integração contínua automatizam a compilação e testes do software (HILTON et al., 2017), trazendo benefícios sólidos para a indústria. Não obstante a ampla adoção e maturidade da integração contínua, ainda há amplo espaço para estudo e pesquisa.

[1] Estudante de Sistemas de Informação 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.

Como consequência natural da adoção da integração contínua, a arquitetura dos sistemas emerge ao longo de suas diversas versões. Mesmo que às vezes negligenciada, uma documentação técnica disponível e atualizada pode ajudar os desenvolvedores em suas atividades de desenvolvimento e manutenção. Em seu estudo, Dzidek (DZIDEK et al., 2008, p. 407-432) conclui que ter documentação disponível durante a manutenção do sistema reduz o tempo necessário para tarefas de manutenção em aproximadamente 20%. Ele conclui que a presença de documentação adicional no formato UML fornece aos desenvolvedores um melhor entendimento do sistema.

Apesar de seus benefícios, muitos percebem modelos UML como não efetivos (BECK, 2004). Confeccionar e manter diagramas UML são consequentemente vistos como de difícil aplicação em projetos de desenvolvimento com recursos e tempo apertados. Isso é ainda mais evidente no contexto de manutenção do software, que consome a maior parte dos recursos de desenvolvimento (PRESSMAN, 2010).

Essa percepção negativa se dá, em parte, pela grande quantidade de informações nos diagramas, o que dificulta sua compreensão. A fragmentação de diagramas UML em partes menores é uma alternativa vantajosa, por ser mais fácil de digerir (BRAUDE, 2017, p. 28). Tal abordagem, porém, não parece ser amplamente utilizada nem promovida por ferramentas CASE[3] de UML. As mesmas são capazes de produzir diagramas através da análise estática, no entanto, não são capazes de mapear um sistema orientado a objetos através da análise dinâmica, obtida em tempo de execução. Diagramas de análise estática e dinâmica possuem escopos diferentes, sendo os de análise dinâmica melhores representações do comportamento do software, e normalmente menores.

O objetivo deste trabalho é, portanto, propor uma abordagem prática para o desenvolvimento de software, a qual chamarei de engenharia reversa contínua, adereçando diretamente o problema de falta de documentação técnica em projetos que adotam a integração contínua. Através da engenharia reversa contínua, é provida uma documentação técnica de forma automatizada, e atualizada a cada nova versão do software. Essa documentação é no formato de diagramas UML de classes e de sequência, ambos orientados a *features*. A proposta inclui uma ferramenta capaz de produzir diagramas UML analisando a execução de testes de integração, e a insere

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

como uma etapa dentro do *pipeline* de integração contínua, permitindo assim sua atualização automatizada. Esses diagramas são produzidos através da engenharia reversa por análise dinâmica, que é capaz de obter informações de maior valor do que a análise estática. Paralelamente será realizada uma pesquisa com profissionais da área de desenvolvimento de software, para avaliar a percepção de valor e benefícios da proposta desse trabalho.

O artigo está dividido em seis seções. A seção 2 aborda o referencial teórico, tratando da engenharia reversa, orientação a aspectos, AspectJ, UML e integração contínua. Em seguida, na seção 3, são apresentados os trabalhos relacionados. Na seção 4, é apresentada a abordagem proposta de engenharia reversa contínua, que consiste na automatização da engenharia reversa junto com a integração contínua. Após isso, na seção 5, apresenta-se a pesquisa utilizada para mensurar a percepção de desenvolvedores de software quanto aos benefícios e valor da proposta desse trabalho. Por fim, a seção 6 apresenta as conclusões deste estudo, assim como sugestões para trabalhos futuros.

2 REFERENCIAL TEÓRICO

Esta seção tem como objetivo descrever a base teórica, necessária para compreender a proposta desse 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 (AOP). A seção 2.3 aborda o tema de UML. E, por fim, a seção 2.4 descreve o conceito de integração contínua (CI).

2.1 Engenharia Reversa

Engenharia reversa de software é o oposto do processo tradicional de engenharia para desenvolvimento de software (NELSON, 1996). Ela busca gerar modelos mais abstratos a partir de código-fonte, ou até mesmo binários. 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. A única fonte de informação confiável nesses casos, quando disponível, é o próprio código-fonte (SCHACH 2010, p.527).

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 sendo executados 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. Não obstante, 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).

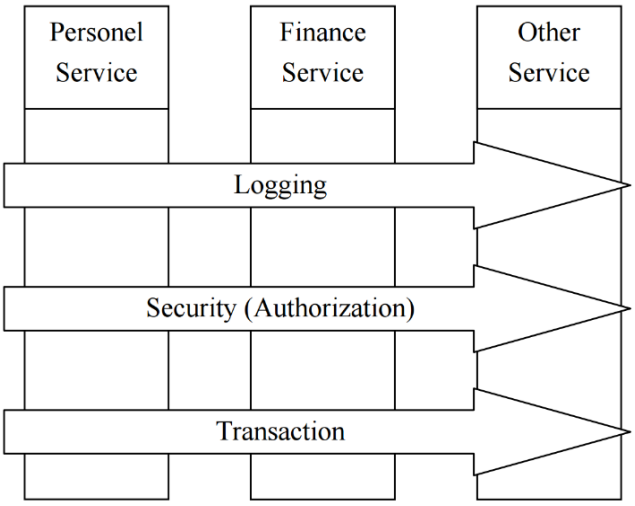
2.2 Programação orientada a aspectos

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.

Na programação orientada a objetos (POO), não há como modularizar 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 modularizar funcionalidades transversais fora do programa base, em módulos separados e bem definidos (RAJAN 2005, p. 59-68). A Figura 1 representa como interesses transversais se espalham pela aplicação.

Figura 1 - Exemplo de interesses transversais



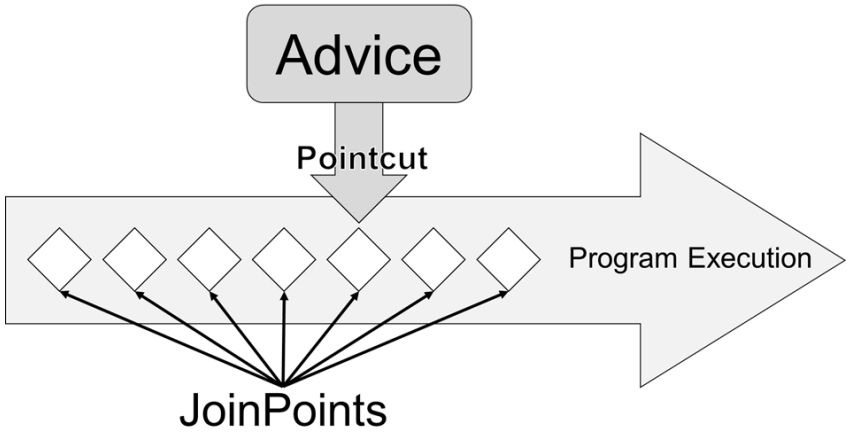
Fonte JU et al. (2007).

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). A Figura 2 ilustra os conceitos chave da POA em ação:

Figura 2 - Funcionamento da orientação a aspectos



Fonte: Elaborado pelo autor.

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). O conceito chave é que os *poinitcuts* definem em quais *joinpoints* são aplicados os *advices*.

O *advice* contém o interesse transversal que precisa ser aplicado. 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 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).

2.3 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. Para tanto, na engenharia, são usados diagramas que representem o sistema a partir de diferentes perspectivas. Hoje, a UML é utilizada para desenhar tais diagramas (LOBO, 2009, P. 18). Recebem destaque nesse trabalho os diagramas de classes e de sequência por fazerem parte do objeto do trabalho proposto.

Para Fowler (2011, p. 52), o diagrama de classes representa o sistema, descrevendo os tipos de objetos e os tipos de relacionamentos estáticos que existem entre eles. Um diagrama de classes pode conter as classes importantes de um sistema, seus atributos, métodos, e principalmente as associações entre as classes.

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).

2.4 Integração Contínua

A Integração Contínua é uma prática de DevOps (*development* e *operations*), em que os desenvolvedores, com frequência, juntam suas alterações de código em um repositório central. Para Humble e Farley, o objetivo da integração contínua é manter o software num estado funcional o tempo todo (HUMBLE e FARLEY, 2013, p. 55). Segundo Beck (1999), o processo de integração produz versões funcionais que crescem em funcionalidade a cada versão. A ideia motivadora da CI é que, quanto mais frequentemente um projeto puder ser integrado, melhor será (HILTON et al., 2017). Outros objetivos da integração contínua são encontrar e investigar bugs mais rapidamente, melhorar a qualidade do software e reduzir o tempo que leva para validar e lançar novas atualizações de software (DUVAL et al., 2007). Para que a integração

alcance tais objetivos, é necessário verificar o funcionamento do código após cada modificação. É, portanto, fundamental o desenvolvimento em conjunto com a realização de testes automatizados (HUMBLE e FARLEY, 2013, p. 60).

Como parte da integração contínua, temos o conceito *pipeline*. Ele quebra o processo de entrega de software em etapas. Cada etapa visa verificar a qualidade dos novos recursos de um ângulo diferente para validar a nova funcionalidade e impedir que os erros afetem seus usuários (PHILLIPS, 2014). Num processo de *pipeline* simples, o início é disparado quando o código é enviado por *commit* para um repositório hospedado em algum lugar como, por exemplo, GitHub. Em seguida, um sistema de integração contínua como, por exemplo, o Travis CI é notificado, e então compila o código e executa os testes unitários. Se o *pipeline* compilou corretamente, e os testes unitários não acusaram erros, os testes de integração são o próximo passo. Após a conclusão dos testes de integração, os pacotes podem ser implantados.

3 TRABALHOS RELACIONADOS

Esta seção tem como objetivo sumarizar os trabalhos encontrados que propõem uma abordagem relacionada com o trabalho proposto. O estado da arte sugere que há uma escassez de trabalhos adereçando a conciliação da automatização de diagramas UML com a integração contínua. Ainda assim, os trabalhos selecionados tratam de ferramentas de engenharia reversa capazes de gerar diagramas UML ou que apresentem diferentes abordagens para prover documentação na integração contínua.

3.1 Análise dos trabalhos selecionados

Esta subseção apresenta uma análise descritiva de cada um dos trabalhos selecionados com os critérios anteriormente mencionados. As subseções 3.1.1 a 3.1.6 apresentam cada uma um trabalho diferente.

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

O trabalho apresentado por Briand et al (2004) consiste na 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. Através dele, é possível inserir um interesse transversal (uma funcionalidade presente em todos ou vários módulos do sistema) sem que o código necessite saber dele.

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. Ele propõe como trabalhos futuros ao seu, que seja feito um estudo sobre a geração de diagramas de sequência formados a partir de vários diagramas de cenário, para abranger melhor o funcionamento de uma ferramenta.

3.1.2 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 softwares 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. 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.

3.1.3 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. Ao tratar da fase de manutenção dos mesmos, é apontada a migração desses sistemas legados para ambientes distribuídos, como a Internet. Eles ressaltam a necessidade de compreender a arquitetura e comportamento dos mesmos, indicando a engenharia reversa como alternativa para essa questão.

A orientação a objetos tem um grande impacto nos resultados esperados como produto de engenharia reversa. As técnicas tradicionais de análise estática de software mantêm seu valoroso papel na compreensão de programas, porém técnicas adicionais focando em análise de tempo de execução dos sistemas tornam-se cada vez mais importantes.

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 dos mesmos. 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.

3.1.4 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.

3.1.5 Incremental UML for agile development: embedding UML class models in source code Understanding and improving continuous integration

Nesse trabalho, Braude (2017, p. 27-31) descreve uma disciplina para embutir modelos de classes da UML junto ao código fonte de maneira incremental, para metodologias ágeis. Em sua abordagem tais modelos são manualmente construídos e incrementados representando apenas o necessário para as funcionalidades sendo desenvolvidas a cada iteração. Tal abordagem é descrita não como apenas mais um recurso, mas que mitiga a ilegibilidade de modelos de classes da UML complicados e volumosos. O trabalho de Braude propõe uma disciplina para manualmente confeccionar modelos fragmentados por funcionalidade. Pode-se, portanto, apontar uma grande diferença no método utilizado para alcançar o resultado final, visto que o trabalho de Braude propõe a confecção manual e incremental, e esse trabalho propõe a automação dos modelos, de maneira incremental.

3.1.6 ReverseJ

O ReverseJ (SCHENATTO, 2015) é uma ferramenta que gera, na linguagem Java, os diagramas de classe e de sequência da UML através da engenharia reversa. Visto que a mesma se dá a partir da análise da execução das *features* do software, é considerada dinâmica. Para tal, é feito uso da programação orientada a aspectos através do AspectJ, de forma que a análise é realizada sem que o código tenha qualquer conhecimento da análise em si. Uma execução do sistema analisada produz seus respectivos diagramas de classes e sequência. A avaliação qualitativa realizada aponta uma alta precisão da ferramenta ao comparar os diagramas produzidos por ela com os diagramas esperados como resultado.

3.2 Comparação dos trabalhos relacionados

Com o objetivo de facilitar a comparação dos trabalhos relacionados, foi desenvolvida a Tabela 1. Nela é possível ver de uma forma mais simples e compacta as características e tecnologias utilizadas em cada um dos trabalhos, permitindo a identificação de características e lacunas relevantes para o trabalho desenvolvido.

Foram comparados critérios quanto às tecnologias envolvidas, bem como aos métodos utilizados e aos artefatos produzidos.

Tabela 1 – Comparação dos trabalhos

	Briand et al (2004)	**Merdes e Dorch (2006)**	**Stroulia e Systä (2002)**	**Antoniol et al. (2004)**	**Braude (2017)**	**Schenatto (2015)**	**Schenatto (2017)**
Usa POA	Sim	Sim	Não	Não	Não	Sim	Sim
Linguagem	Java	Java	Java	Não informado	Não informado	Java	Java
Usa AspectJ	Sim	Não	Não	Não	Não	Sim	Sim
Tipo de análise	Dinâmica	Dinâmica	Dinâmica	Dinâmica e Estática	Manual	Dinâmica	Dinâmica
Objeto da análise	Código binário	Código binário	Código fonte	Código fonte	Código fonte	Código fonte	Código fonte
Gera diagrama de classe	Não	Não	Não	Sim	Sim	Sim	Sim
Gera diagrama de sequência	Sim	Sim	Sim	Sim	Não	Sim	Sim
Gera diagrama com escopo de *feature*	Sim	Sim	Sim	Sim	Sim	Sim	Sim
Suporta engenharia reversa contínua?	Não	Não	Não	Não	Não	Não	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. Os diagramas com escopo de *feature* que eles geram são apenas de sequência, não dispondo do equivalente para o diagrama de classe. Além disso, nenhum deles suporta a engenharia reversa contínua. A Ferramenta UMLRev, por sua vez, gera diagramas de classes e de sequência em tempo de execução, tendo o AspectJ como mecanismo para a análise e possui suporte para a engenharia reversa contínua.

4 ABORDAGEM PROPOSTA

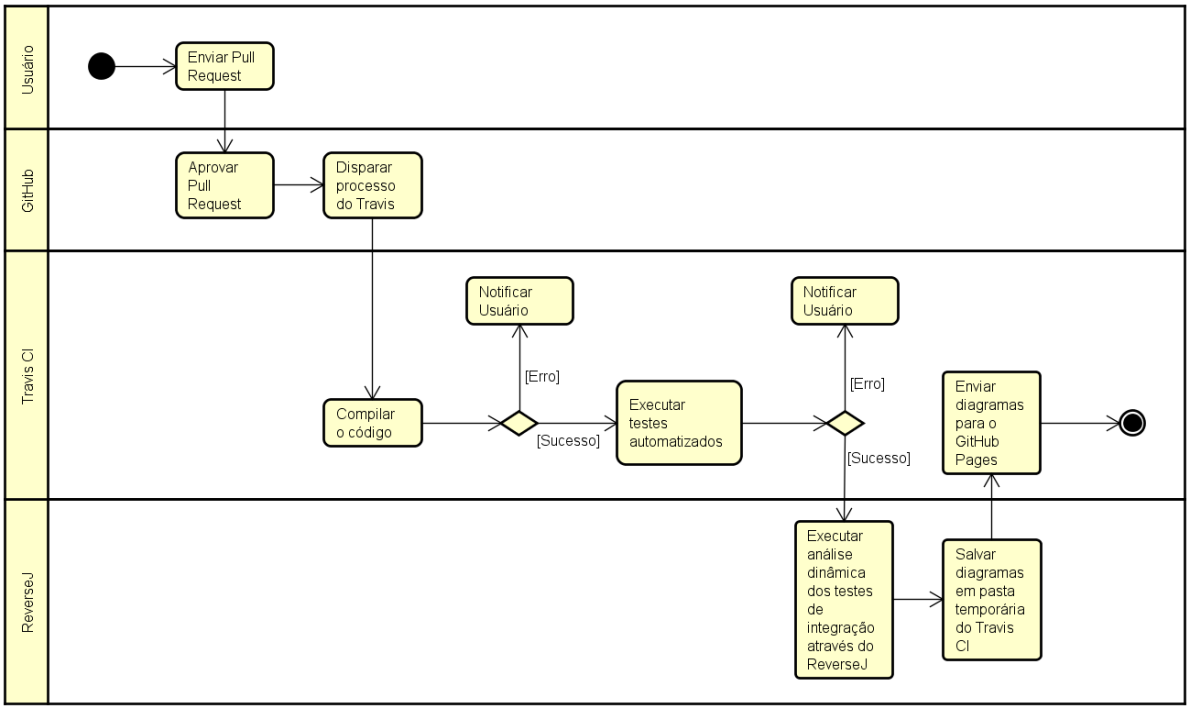
Esta seção apresenta a ferramenta utilizada para a engenharia reversa dinâmica, a técnica utilizada para automatizá-la e sua integração com ferramentas de integração contínua. Também são expostas as tecnologias utilizadas para alcançar tal objetivo.

4.1 Visão geral do processo

Esta seção apresenta o *pipeline* (ou processo) utilizado para entregar de maneira contínua a engenharia reversa do software, provendo uma documentação técnica automatizada e sempre atualizada. Utilizando a ferramenta de integração contínua Travis CI, é possível incluir etapas automatizadas, tais como *scripts* de configuração e testes de integração, ao *pipeline*.

Como pode ser observado na Figura 3, não foi necessário realizar nenhuma mudança no processo normal até o ponto em que o Travis CI (servidor de integração contínua) termina com sucesso a compilação do software e testes automatizados subsequentes. Nesse ponto, é acionada a execução da ferramenta UMLRev, no próprio servidor do Travis CI, que então analisa uma nova execução dos testes de integração (visto que os mesmos já foram executados na etapa anterior) e gera os respectivos diagramas de classes e sequência. Esses diagramas produzidos na execução da UMLRev são então armazenados em uma pasta local e temporária no próprio servidor do Travis CI. Por fim, os mesmos são enviados para o GitHub Pages via um *script* configurado para tal, cuja execução se dá no próprio servidor Travis CI.

Figura 3 – Fluxo do *pipeline*



Fonte: elaborado pelo autor.

4.2 Ferramenta para engenharia reversa dinâmica

A ferramenta utilizada é a UMLRev. Tal nome é originado do conceito *reverse engineering* e dos artefatos de UML produzidos pela mesma, logo UMLRev. Ela foi desenvolvida para funcionar de forma automatizada, possibilitando sua execução em servidores de integração contínua (CI). O principal propósito é gerar uma representação gráfica da execução de *features* do software, o que não é possível com a análise estática.

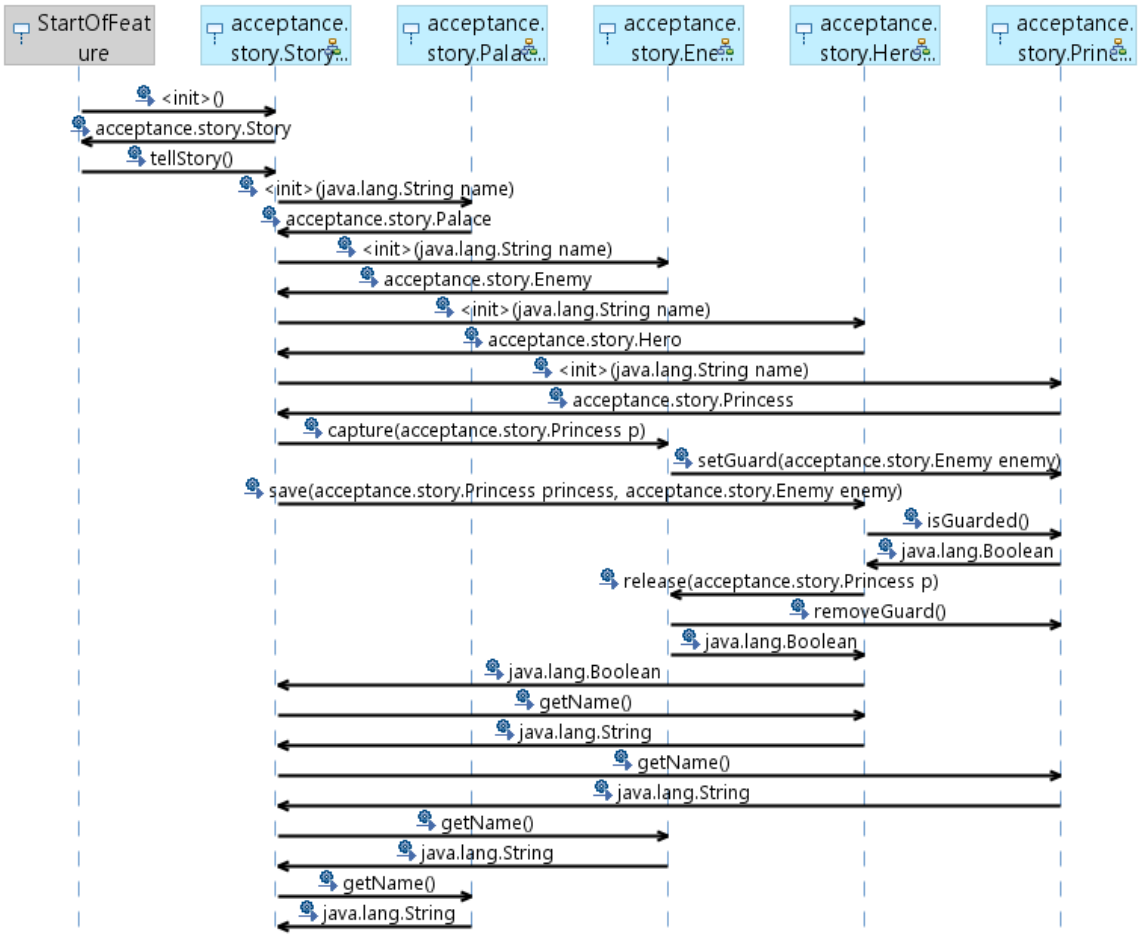
Os principais diferenciais da ferramenta estão na utilização da orientação a aspectos e na análise de *features* em tempo de execução. O uso da orientação a aspectos permite instrumentalizar a aplicação sem que ela tenha que se preocupar com essa análise. Essa instrumentalização, feita sobre o código fonte através de aspectos, preserva informações como os nomes das classes e métodos, as quais seriam perdidas na análise de código binário.

UMLRev é capaz de produzir diagramas de classes e de sequência, de forma a abranger exclusivamente a funcionalidade sendo analisada. Seu foco 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 classes ou sequência, corresponde ao comportamento e escopo do sistema para a *feature* sob análise, podendo conter informações distintas das obtidas com análise estática.

Na Figura 4 é apresentado um diagrama de sequência gerado como resultado da análise de uma pequena aplicação. Como pode ser visto, ele evidencia as classes que compõem a *feature*, suas respectivas mensagens e a ordem em que eles ocorreram.

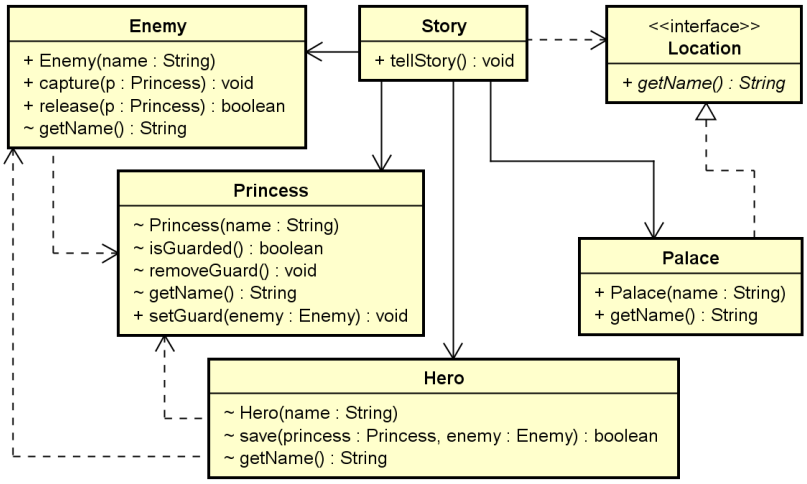
Figura 4 - Exemplo de diagrama de sequência



Fonte: Elaborado pelo autor.

O diagrama de classes gerado, como pode ser visto na Figura 5, permite identificar as classes e métodos utilizados na execução da *feature*.

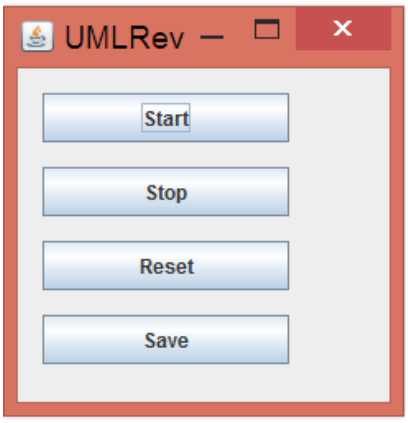
Figura 5 - Exemplo de diagrama de classes resultante



Fonte: Elaborado pelo autor.

A execução da UMLRev no servidor não necessita de interface gráfica para interação do usuário. Não obstante, uma interface gráfica foi criada para uso fora do servidor, permitindo analisar manualmente execuções da aplicação. Através dela é provido controle sobre quando a ferramenta deve começar e parar a análise, reiniciar o processo, e salvar os diagramas gerados. Essa interface pode ser observada na Figura 6:

Figura 6 - Interface gráfica



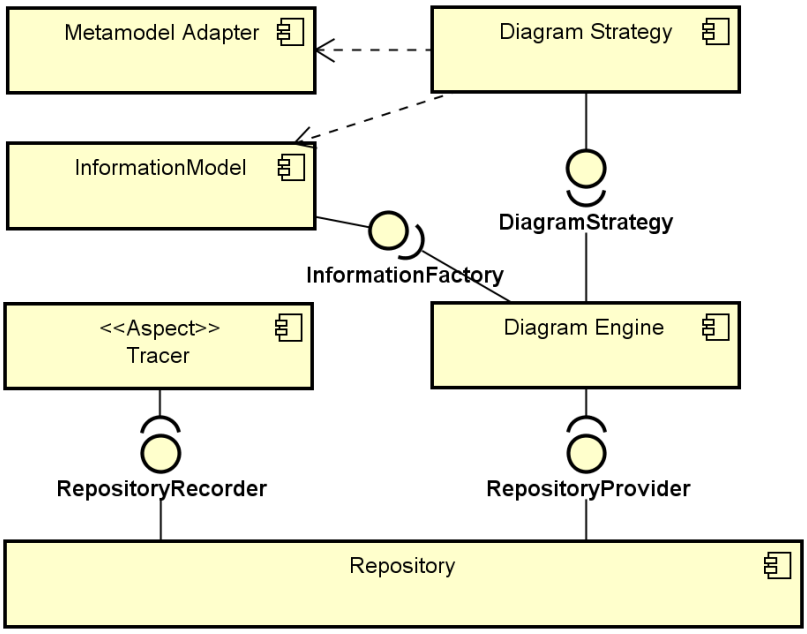
Fonte: Elaborado pelo autor.

4.3 Componentes da ferramenta

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 7 concentra-se em apresentar os componentes principais do processo.

Figura 7 – 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. Essas 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 classes e de sequência, mas a ferramenta suporta a implementação de quantos e quais geradores de modelos forem necessários, sem que a arquitetura seja alterada.

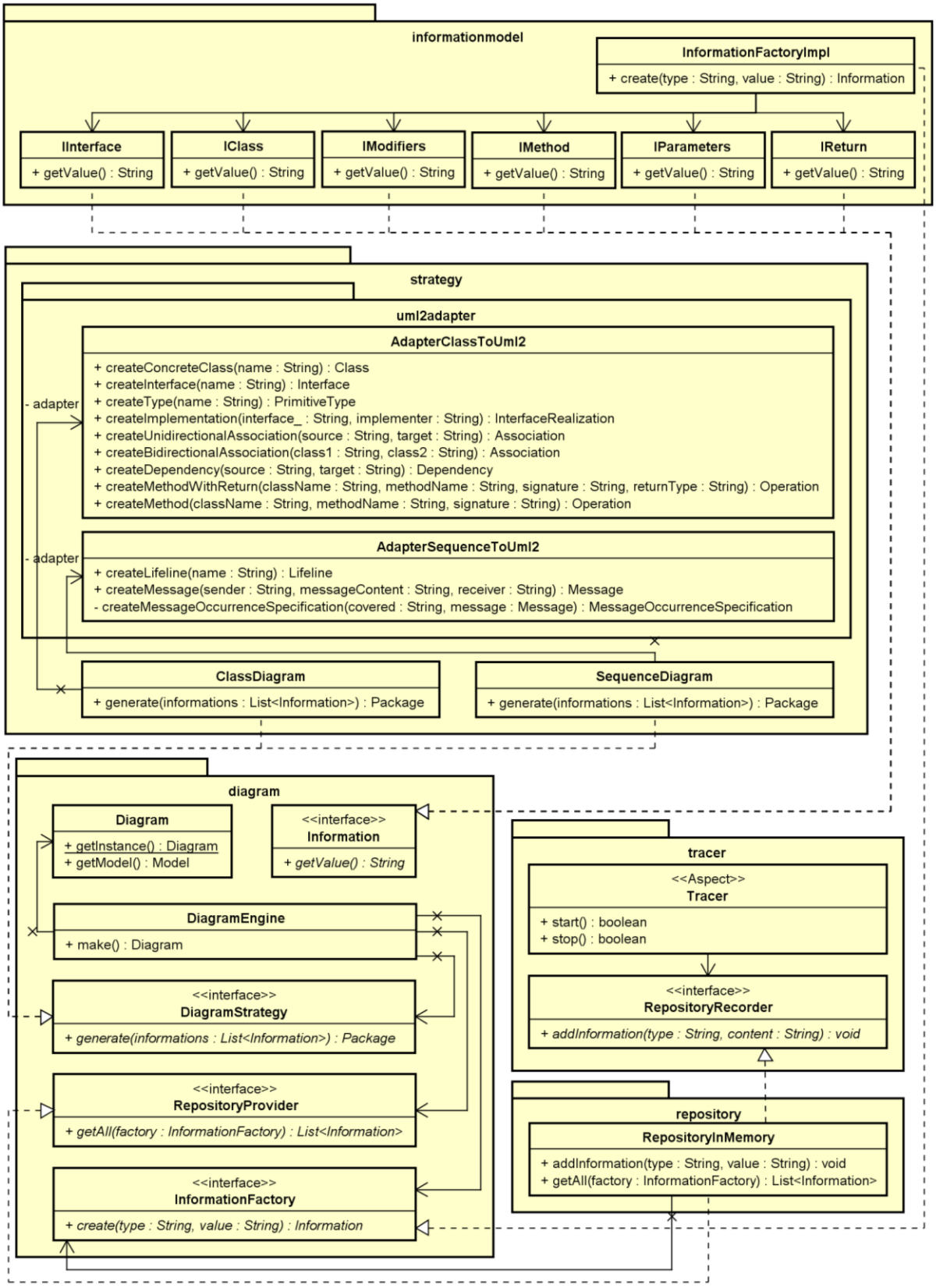
Information Model define o modelo das estruturas de dados com o qual as estratégias irão trabalhar. Informações provenientes do *Repository* tais como nomes de métodos, seus parâmetros e respectivas classes são estruturados de acordo com o definido nesse pacote.

Responsável pela comunicação direta com o framework de UML, o *Metamodel Adapter* estabelece uma ponte entre as estratégias de geração de diagramas do *Diagram Strategy* e o framework que vai de fato produzir tais diagramas. 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.

4.4 Visão lógica da arquitetura

O diagrama de classes 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 8, 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 8 – Diagrama de classes



Fonte: elaborado pelo autor.

4.5 Algoritmos

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, serão exibidos alguns fragmentos do código fonte.

O trecho de código apresentado na Figura 9 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 analisado. Isso acontece de maneira dinâmica, e sem que o sistema tenha qualquer conhecimento de sua atuação. Os 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 9 – Principais *pointcuts* do *Tracer*

```
1.  pointcut methodCall(): call(* *.*(..))&&immune();
2.  pointcut methodExecution(): execution(* *.*(..))&&immune();
3.  pointcut constructorCall(): call(*.new(..))&& immune();
4.  pointcut constructorExecution(): execution(*.new(..))&&immune();
```

Fonte: Elaborado pelo autor.

O método da Figura 10 esclarece a abordagem para gerar todos os diagramas desejados a partir do mesmo conjunto de informações obtido pelo *Tracer*. Este código pertence à classe *DiagramEngine.java*, a qual gera diagramas para todas as estratégias implementadas, ou seja, diferentes diagramas implementados.

Figura 10 – 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 na Figura 11 e na Figura 12, por sua vez, são as implementações do método *generate*, chamado na linha três da Figura 10. 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 classes e diagrama de sequência respectivamente).

Figura 11 – 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.

Figura 12 – 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.

4.6 Aspectos da implementação

Visto que a UMLRev analisa a execução do software para realizar a engenharia reversa, automatizar a análise de *features* envolve a escrita de *scripts* de execução separados por *features*. Com o intuito de simplificar e reduzir o trabalho necessário, viu-se grande utilidade numa das práticas utilizadas na integração contínua, que são os testes de integração. Tais testes possuem como uma de suas finalidades testar os requisitos funcionais, sendo inclusive desejável agrupá-los por áreas funcionais (HUMBLE e FARLEY, 2013, p.61). Se forem bem modularizados, cada um desses testes foca numa funcionalidade específica. Assim sendo, a ferramenta foi adaptada para utilizar a execução desses testes automatizados em sua análise.

O cerne da ferramenta está em sua capacidade de realizar a engenharia reversa através da análise dinâmica. A mesma usa a orientação a aspectos, através do AspectJ[4] para capturar informações da execução das *features* da aplicação sem que a mesma tenha que se preocupar com essa análise. A 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. O gerenciamento de dependências foi realizado através do Maven[5], devido à sua facilidade de configuração. Também, foi necessário que os diagramas produzidos sejam salvos em local acessível pela ferramenta de integração contínua, no próprio servidor.

Dentre as diversas opções para configurar o *pipeline* de integração contínua, três ferramentas foram utilizadas: Travis CI[6], GitHub[7] e GitHub Pages[8].

O Travis CI é um serviço distribuído de integração contínua, usado para construir e testar projetos de software hospedados no GitHub. Sua escolha deu-se por uma série de fatores, dentre eles: não necessitara instalação de um servidor dedicado; é de fácil configuração; é integrado com GitHub; e permite a execução de *scripts*, trazendo a versatilidade necessária.

[4] AspectJ, disponível em <https://eclipse.org/aspectj/>.
[5] Maven, disponível em <https://maven.apache.org>.
[6] Travis CI, disponível em <https://travis-ci.org/>.
[7] GitHub, disponível em <https://github.com/>.
[8] GitHub Pages, disponível em <https://pages.github.com/>.

O GitHub é um serviço Web de hospedagem de repositórios, com controle de versionamento git. Ele foi utilizado por sua integração com o Travis CI, e por facilitar o controle de *pull requests*, gatilho usado para disparar a execução do Travis CI.

GitHub Pages é um serviço integrado ao GitHub que permite transformar repositórios GitHub em websites, muito utilizado para documentação de projetos. Por contar com a estrutura do GitHub, possui todas as funcionalidades de um repositório GitHub normal. Por também se tratar de um repositório de documentação conectado com o repositório do projeto no próprio GitHub, ele foi escolhido como local para armazenar os diagramas produzidos pela ferramenta durante sua execução no Travis CI.

5 PESQUISA

Esta seção apresenta a pesquisa realizada de abordagem quantitativa, cuja técnica de coleta de dados foi baseada em questionário. Ela foi direcionada a avaliar a percepção dos benefícios da proposta apresentada nesse trabalho. A caracterização dos participantes pode ser encontrada na seção 5.1. O questionário elaborado é descrito na seção 5.2. Já os resultados e sua análise estão descritos na seção 5.3.

5.1 Participantes

Voluntários foram selecionados para participar nesse estudo através da seleção de conveniência. Um convite para participar no estudo foi enviado para uma lista de e-mails de estudantes atuais e antigos de mestrado da Unisinos, bem como contatos profissionais do pesquisador. Para participar no estudo, os participantes tinham que confirmar sua experiência com desenvolvimento de software. Cada participante foi contextualizado quanto aos temas e conceitos utilizados nesse trabalho, bem como o processo desenhado e abordagem proposta, antes de responder o questionário.

O questionário ficou aberto entre os dias 11 e 25 de outubro de 2017. Como resultado, foram coletados no total dados de 22 participantes. Os dados coletados das informações demográficas que caracterizam os participantes estão disponíveis no Apêndice A. Os participantes possuem idades entre 21 e 45 anos. A maioria possui formação em Ciência da Computação e Análise de Sistemas, sendo quase sua

totalidade com grau de escolaridade em Graduação e Mestrado e tempo de estudo de 5 anos ou mais. Com respeito à experiência de trabalho, todos possuem experiência com desenvolvimento de software, e há uma distribuição harmoniosa desde os que possuem menos de 2 anos até os que possuem mais de 8 anos. Grande parte deles trabalha atualmente em posições que realizam desenvolvimento de software ou diretamente relacionadas a ele, cuja experiência na posição atual também varia quase que uniformemente desde menos de 2 anos a mais de 8 anos.

5.2 Questionário

Para a pesquisa do trabalho proposto, foi realizado um questionário com 21 afirmações (ver Apêndice B), para as quais cada participante poderia responder indicando o quanto concorda, de acordo com a escala: Concordo Totalmente; Concordo Parcialmente; Neutro; Discordo Parcialmente; Discordo Totalmente.

O questionário da pesquisa foi dividido em cinco enfoques distintos: quanto à engenharia reversa contínua; quanto ao uso da UML no meio empresarial; quanto a diagramas de classes e seu uso associado a *features; q*uanto à automatização de diagramas; e quanto à utilidade de diagramas no dia-a-dia. Também foi fornecido no questionário um espaço para comentários sugestões, de preenchimento não obrigatório.

5.3 Resultados e Análise

Tendo detalhado o procedimento do estudo e seus participantes, prosseguirei agora para a apresentação dos resultados da pesquisa e sua análise, os quais serão descritos de acordo com os enfoques de questões introduzidos na seção 5.2. Uma apresentação completa dos números obtidos como resultado pode ser consultada no Apêndice C.

5.3.1 Quanto à engenharia reversa contínua

Somando os que concordam totalmente aos que concordam parcialmente, 73% dos participantes percebem a proposta como incentivadora da adoção de modelos

UML. Quando perguntados sobre a percepção de benefícios para a indústria, 86% deles concordam com a afirmação. Seguindo na mesma linha, 82% dos participantes concorda que tal proposta resolveria a falta de documentação técnica de um sistema.

5.3.2 Quanto ao uso da UML no meio empresarial

A maioria dos participantes concorda com a utilidade da UML no meio empresarial, e que sua manutenção é custosa e difícil. Vale ressaltar que 60% dos participantes se posicionaram de forma neutra ou discordando quanto ao custo de manter a documentação técnica atualizada, mas 68% concorda com a dificuldade em manter atualizados diagramas de classes, e 77% quanto a diagramas de sequência.

5.3.3 Quanto a diagramas de classes e seu uso associado a *features*

Parte da pesquisa realizada deu-se no intuito de identificar a percepção dos participantes quanto ao valor dos diagramas de classes convencionais em comparação com os gerados pela UMLRev. Respectivamente 72% e 86% dos participantes concordam parcialmente ou totalmente com as afirmações quanto à debilidade dos diagramas convencionais e quanto ao valor agregado pelos diagramas propostos.

5.3.4 Quanto à automatização de diagramas

Uma alta porcentagem dos participantes concorda que a automatização dos diagramas UML contribuiria com sua maior utilização na indústria. No entanto a percepção é diferente para cada metodologia ou etapa do projeto. 91% dos participantes concordam com a utilidade da automatização de diagramas em metodologias ágeis, sendo que 55% se manifestaram como concordando totalmente. Já para metodologias convencionais como *waterfall*, apesar de 82% concordarem com sua utilidade, apenas 23% concordam totalmente. Indo além, para a fase de manutenção, a percepção de utilidade cai drasticamente, com apenas 9% concordando totalmente e um total de 32% concordando. Tais números evidenciam o valor percebido durante a fase de projeto e em especial os de metodologias ágeis,

caracterizados por gerar várias mudanças na arquitetura ao longo do projeto. Não obstante pouco valor é percebido para a fase de manutenção.

5.3.5 Quanto à utilidade de diagramas no dia-a-dia

De forma semelhante aos resultados apresentados nos outros grupos de questões, houve grandes porcentagens dos participantes concordando totalmente ou parcialmente com esse grupo de questões, que trata da utilidade desses diagramas no uso prático em questões do dia-a-dia. Nesse ponto participantes concordaram com a utilidade para as situações de: entrar num projeto em andamento (72%), e realizar uma alteração na regra de negócio (82%). Pode-se ver também 23% se posicionando de forma neutra quanto à utilidade de diagramas ao entrar num projeto em andamento, o que pode ser indício de incerteza quanto a sua aplicação.

5.3.6 Sumarização dos comentários e sugestões

No total 5 participantes escreveram comentários ao responder a pesquisa, os quais estão sumarizados nesta seção. Os parágrafos a seguir não representam necessariamente a opinião do autor, mas sim sua sumarização dos comentários coletados.

Um problema que contribui para o não uso de UML em projetos de software é a falta de entendimento de como utilizar seus diagramas. Muitos profissionais de TI não sabem como aplicar a engenharia de software no dia a dia ou não possuem domínio do problema, o que ocasiona numa péssima modelagem do sistema.

A adoção de modelos UML seria aumentada de acordo com o valor que tais modelos agregam. Para isso não apenas a técnica influencia, mas também a qualidade do código, que precisa de bons identificadores (nomes de atributos, métodos, classes, pacotes) para produzir um bom diagrama.

No desenvolvimento de software, diagramas são complementares. Ainda assim, sem eles a curva de aprendizagem do sistema e base de dados é maior e penosa para o desenvolvedor.

A documentação técnica, mesmo que envolva diagramas UML, requer mais informações do que apenas os diagramas, tal como o registro de tomadas de decisões e seus porquês. A arquitetura do software é diretamente influenciada pelas decisões

tomadas ao longo do projeto, e ter conhecimento delas previne a tomada de decisões equivocadas no futuro, como em tempo de manutenção.

Quanto aos benefícios da engenharia reversa contínua, do ponto de vista da inovação traz grandes benefícios, mas do ponto de vista dos modelos UML depende da qualidade do código existente.

Ter disponíveis vários diagramas de classes separados por *features* traz mais valor do que um único diagrama mostrando todo o sistema. Porém, diagramas separados por “partes que façam sentido estar juntas” também agrega muito valor. Diagramas de sequência também agregam valor, pois definem um fluxo de ações necessárias para cumprir uma dada missão.

Diagramas gerados automaticamente deveriam passar por revisão de um analista, para garantir a aderência do diagrama à *feature* e incluir classes secundárias que possam ter impacto na mesma.

6 CONCLUSÃO

O objetivo deste trabalho foi apresentar a engenharia reversa contínua, uma proposta para o problema de falta de documentação técnica atualizada em projetos. Como resultado, a proposta entrega uma documentação no formato de diagramas UML de classes e de sequência, providos de forma automatizada e atualizados a cada alteração. Esse objetivo foi alcançado através da UMLRev, uma ferramenta de engenharia reversa que faz uso da análise dinâmica. A ferramenta foi então inserida como uma etapa dentro de um *pipeline* de integração contínua, permitindo assim a atualização automatizada dos diagramas.

Os maiores desafios foram o de integrar a ferramenta para execução em servidor de integração contínua, e o desafio de prover *scripts* de execução separados por *feature*, de modo a não adicionar uma sobrecarga de trabalho ao usuário. Viu-se oportunidade de aproveitar os próprios testes de integração como *scripts*, visto que já são uma das boas práticas da integração contínua, e que executam funcionalidades da aplicação. Em relação ao conjunto de ferramentas utilizadas com a UMLRev, encontram-se o servidor de integração contínua Travis CI; o GitHub, para o controle de versão; e GitHub Pages, para armazenar os diagramas produzidos.

Este artigo também buscou avaliar quantitativamente o valor percebido em sua proposta, por parte dos desenvolvedores de software, através de uma pesquisa

formada por um questionário. Ela evidencia que tanto em metodologias convencionais, quanto nas metodologias ágeis, a documentação técnica é percebida como útil e de alto valor para auxiliar na compreensão da aplicação. No entanto, mostra a percepção de baixa efetividade da UML da maneira que é usada hoje, e de que a UML seria mais amplamente utilizada no mercado com o uso da abordagem proposta. Destaca-se a percepção de utilidade e valor para tarefas do dia-a-dia, que exigem do desenvolvedor que esteja a par da arquitetura e design da aplicação, como correção de bugs e mudança nos requisitos. Também foi percebido grande benefício da proposta para a capacitação de novos recursos que chegam ao projeto.

Desta forma, conclui-se que a abordagem proposta pode trazer benefícios para a indústria de desenvolvimento de software, aumentando a adoção de modelos UML e provendo diagramas atualizados para os desenvolvedores de software. Como trabalhos futuros, é apontada uma pesquisa com mais participantes quanto à aceitação da proposta. Quanto à ferramenta UMLRev, é apontado como trabalhos futuros a confecção de outros diagramas da UML além do diagrama de classes e de sequência. Outro possível trabalho futuro é o de avaliar a qualidade dos diagramas gerados pela UMLRev em comparação com os gerados por outras ferramentas.

REFERÊNCIAS

AHO, Alfred V.; LAM, Monica S.; SETHI, Ravi; e ULLMAN, Jeffrey D. **Compilers: principles, Tools and Techniques.** Harlow, England: Addison-Wesley, 2nd edition, 1986.

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

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.

BECK, Kent. **Embracing Change with Extreme Programming**. Em: IEEE Computer Society Press, Volume 32 Issue 10. Los Alamitos, CA, 1999, p. 70-77.

BECK, Kent. **Extreme Programming Explained: Embrace Change**. Addison-Wesley Professional, 2004.

BRAUDE, Eric. **Incremental UML for agile development**: embedding UML class models in source code. Em Proceedings of the 3rd International Workshop on Rapid Continuous Software Engineering, Buenos Aires, 2017, p. 27-31.

BRIAND, Lionel C.; LABICHE, Yvan; LEDUC, Johanne. **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.

DIJKSTRA, Edsger. **On The Role of Scientific Thought. 1974**. Disponível em: <http://www.cs.utexas.edu/users/EWD/ewd04xx/EWD447.PDF>. Acesso em: 09/11/2017.

DUVAL, Paul M; MATYAS, Steve; GLOVER, Andrew. **Continuous Integration:** Improving Software Quality and Reducing Risk. Addilson Wesley, 2007.

DZIDEK, Wojciech James; ARISHOLM, Erik; BRIAND, Lionel C. **A Realistic Empirical Evaluation of the Costs and Benefits of UML in Software Maintenance**. Em: IEEE Transactions on Software Engineering, volume: 33, Nº: 3. Ottawa, Canada, 2008, p. 407-432.

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

HILTON, Michael; TIMOTHY, Tunnell; HUANG, Kai; MARINOV, Darko; DIG, Danny. **Usage, costs, and benefits of continuous integration in open-source projects**. Disponível em: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. Singapura, 2016, p. 427.

HUMBLE, Jez; FARLEY, David. **Entrega Contínua:** Como Entregar Software. Bookman Editora, 2014.

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; 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, Gail C.; NOTKIN, David; GRISWOLD, William G; e LAN, Erica S. **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 12/07/2017.

PHILLIPS, Andrew; SENS, Michiel; JONGE, Adriaan de; HOLSTEIJN, Mark Van. **The IT Manager's Guide to Continuous Delivery**. XebianLabs, 2014, p. 20. Disponível em: <https://xebialabs.com/resources/whitepapers/the-it-managers-guide-to-continuous-delivery/>. Acesso em: 11/11/2017

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

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, James R.; JACOBSON Ivar; e BOOCH Grady. **The Unified Modeling Language Reference Manual.** Reading USA: Addison-Wesley, 1999, capítulo 12.

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

SCHENATTO, Lucas V. **ReverseJ: Uma ferramenta para engenharia reversa baseada em features**. Trabalho de conclusão de curso (Tecnólogo em Análise e Desenvolvimento de Sistemas) – Curso de Análise e Desenvolvimento de Sistemas, Universidade do Vale do Rio dos Sinos (UNISINOS), Porto Alegre, 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: 12/10/2017.

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.

APÊNDICE A– CARACTERIZAÇÃO DOS PARTICIPANTES DA PESQUISA

Tabela 2 – Caracterização dos participantes da pesquisa

Característica	Resposta	Quantidade	Porcentagem
Idade	21-25 anos	7	31,8%
	26-30 anos	7	31,8%
	31-35 anos	3	13,6%
	36-40 anos	3	13,6%
	41-45 anos	2	9,1%
Formação acadêmica	Ciência da Computação	9	40,9%
	Engenharia da Computação	2	9,1%
	Sistemas de Informação	1	4,5%
	Análise de Sistemas	9	40,9%
	Sistemas para internet	1	4,5%
Grau de escolaridade	Graduação *	10	45,5%
	Mestrado *	11	50,0%
	Pós Graduação *	1	4,5%
Tempo de estudo	< 2 anos	1	4,5%
	2-4 anos	1	4,5%
	5-6 anos	8	36,4%
	7-8 anos	6	27,3%
	> 8 anos	6	27,3%
Experiência de trabalho	< 2 anos	5	22,7%
	2-4 anos	3	13,6%
	5-6 anos	1	4,5%
	7-8 anos	5	22,7%
	> 8 anos	8	36,4%
Posição atual	Estagiário	3	13,6%
	Programador	7	31,8%
	Analista	4	18,2%
	Arquiteto	4	18,2%
	Gerente	1	4,5%
	Pesquisador	1	4,5%
	Diretor de Tecnologia	1	4,5%

	Professor	1	4,5%
Experiência na posição atual	< 2 anos	5	22,7%
	2-4 anos	3	13,6%
	5-6 anos	6	27,3%
	7-8 anos	6	27,3%
	> 8 anos	2	9,1%

* Completo ou Incompleto.

Fonte: Elaborado pelo autor.

APÊNDICE B – QUESTIONÁRIO DA PESQUISA

Tabela 3 – Questionário da pesquisa

Enfoque	Número	Afirmação
Quanto à engenharia reversa contínua	1	Ter engenharia reversa orientada a *feature* associada a um sistema de integração contínua (Travis CI ou Jenkins) aumentaria a adoção de modelos UML.
	2	Engenharia reversa contínua traz benefícios claros para o dia a dia da indústria.
	3	Engenharia reversa contínua associada a um sistema resolveria a falta de documentação técnica do mesmo.
Quanto ao uso da UML no meio empresarial	4	Diagramas UML podem ajudar na compreensão do software.
	5	Diagramas UML são amplamente utilizados no meio empresarial.
	6	Diagrama de classes são uteis no meio empresarial.
	7	Diagramas de sequência são uteis no meio empresarial.
	8	É custoso manter uma documentação técnica atualizada.
	9	É difícil manter diagramas de classes sempre atualizados.
	10	É difícil manter diagramas de sequência sempre atualizados.
Quanto a diagramas de classes e seu uso associado a *features*	11	Um diagrama de classes com classes demais possui pouco valor.
	12	Vários diagramas de classes separados por funcionalidades agregam mais valor do que um único diagrama mostrando todo o sistema.
	13	Diagramas UML seriam mais utilizados na indústria se sua confecção fosse automatizada.
Quanto à automatização de diagramas	14	Diagramas UML associados a um sistema e confeccionados automaticamente seriam mais utilizados se filtrassem dados relevantes para cada *feature*.
	15	Ter diagramas atualizados automaticamente é útil em metodologias convencionais (ex. waterfall).

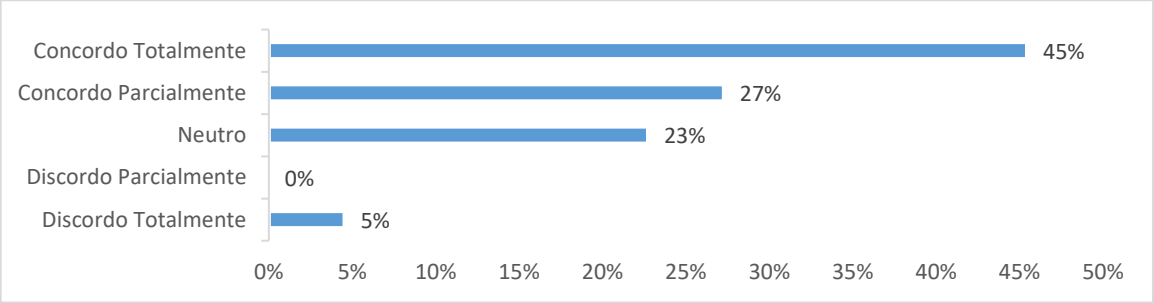
	16	Ter diagramas atualizados automaticamente é útil em metodologias ageis (ex: scrum).
	17	Ter diagramas atualizados automaticamente é útil na fase de manutenção.
Quanto à utilidade de diagramas no dia-a-dia	18	Ao entrar num projeto em andamento, eu gostaria de ter diagramas separados por *feature*.
	19	Ao começar a dar manutenção para um sistema, eu gostaria de ter diagramas separados por *feature*.
	20	Para codificar uma mudança na regra de negócio, eu gostaria de ter diagramas das *features* envolvidas.
	21	Para corrigir um bug, eu gostaria de ter diagramas da *feature* com problema.

Fonte: Elaborado pelo autor.

APÊNDICE C - RESULTADOS OBTIDOS NA PESQUISA

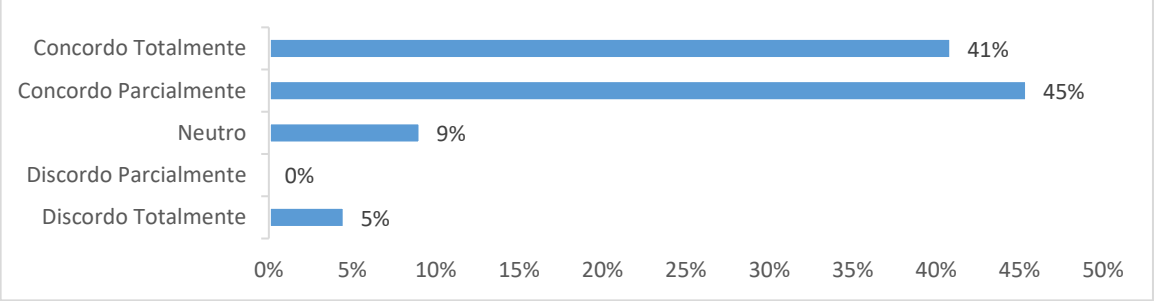
A seguir são apresentados os resultados da pesquisa, separados por questão.

Gráfico 1 - Questão 1 - Ter engenharia reversa orientada a feature associada a um sistema de integração contínua (Travis ou Jenkins) aumentaria a adoção de modelos UML.



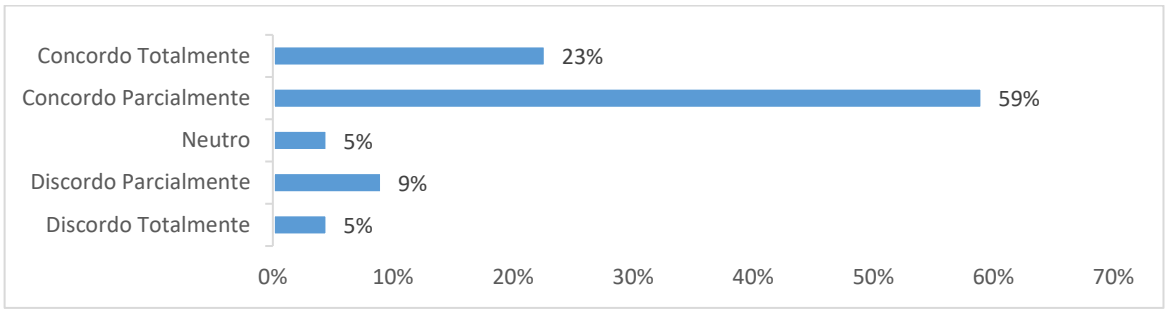
Fonte: Elaborado pelo autor.

Gráfico 2 - Questão 2 - Engenharia reversa contínua traz benefícios claros para o dia a dia da indústria.



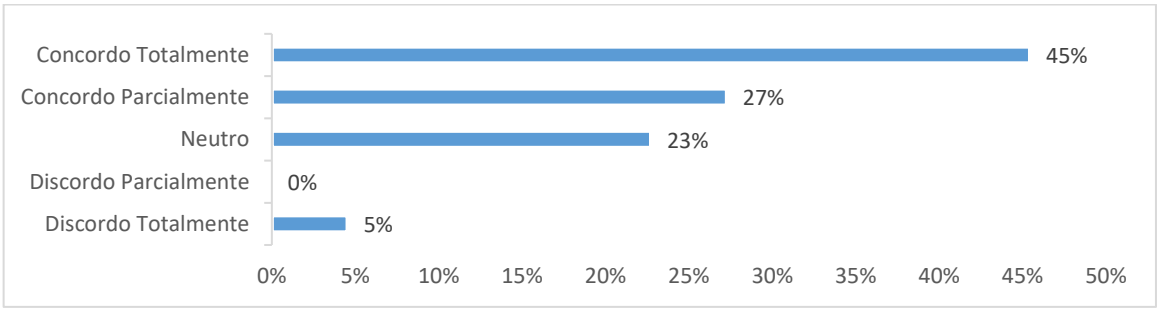
Fonte: Elaborado pelo autor.

Gráfico 3 - Questão 3 - Engenharia reversa contínua associada a um sistema resolveria a falta de documentação técnica do mesmo.



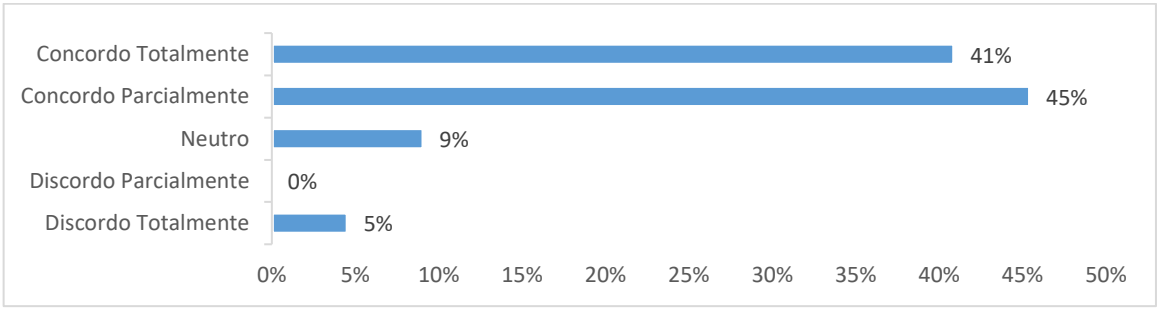
Fonte: Elaborado pelo autor.

Gráfico 4 - Questão 4 - Diagramas UML podem ajudar na compreensão do software.



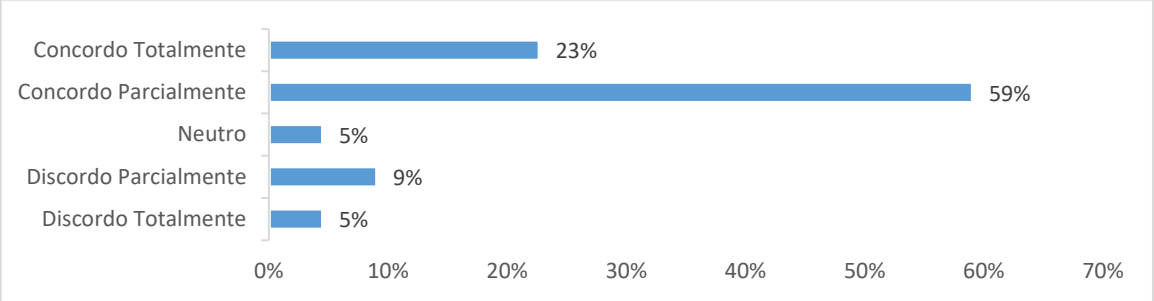
Fonte: Elaborado pelo autor.

Gráfico 5 - Questão 5 - Diagramas UML são amplamente utilizados no meio empresarial.



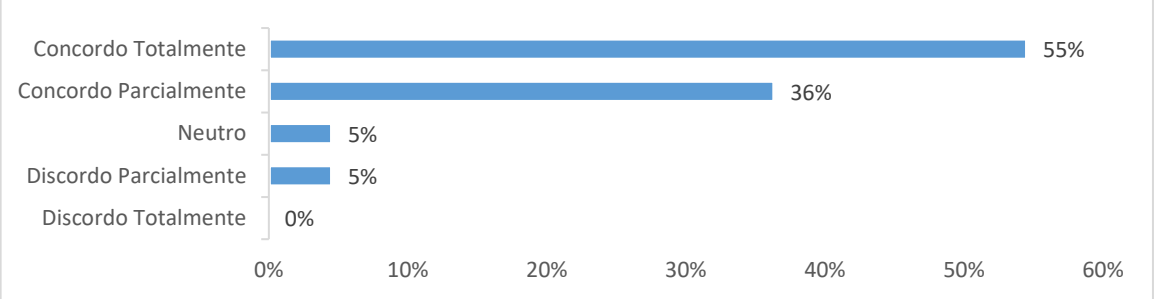
Fonte: Elaborado pelo autor.

Gráfico 6 - Questão 6 - Diagrama de classes são uteis no meio empresarial.



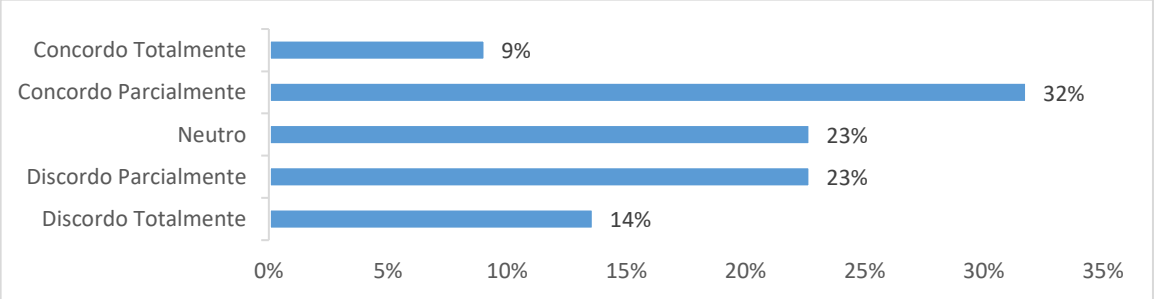
Fonte: Elaborado pelo autor.

Gráfico 7 - Questão 7 - Diagramas de sequência são uteis no meio empresarial.



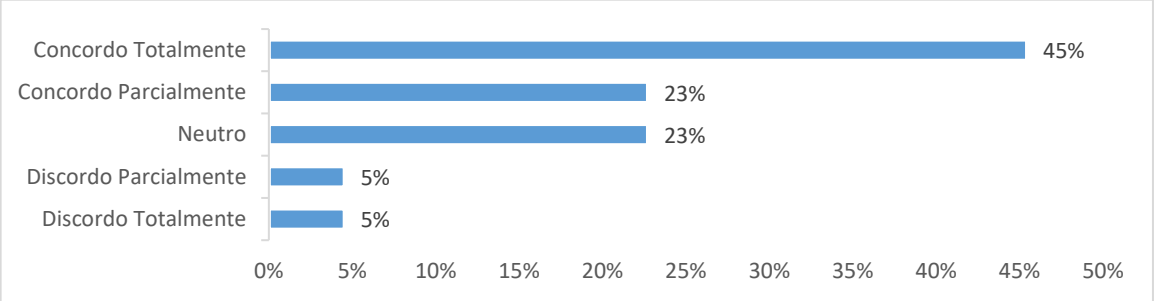
Fonte: Elaborado pelo autor.

Gráfico 8 - Questão 8 - É custoso manter uma documentação técnica atualizada.



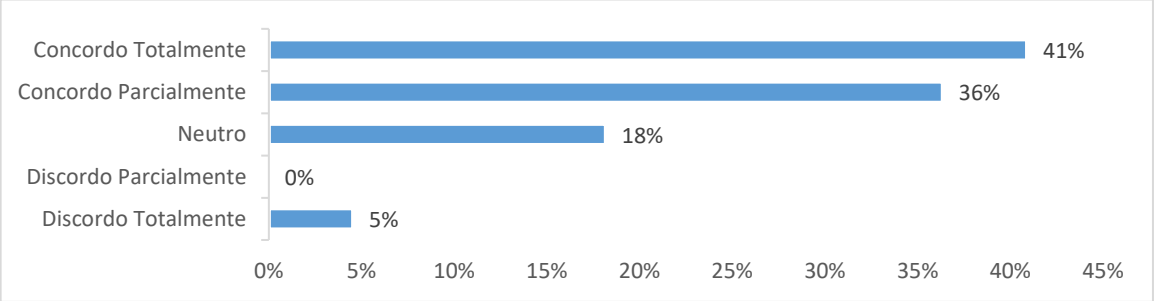
Fonte: Elaborado pelo autor.

Gráfico 9 - Questão 9 - É difícil manter diagramas de classes sempre atualizados.



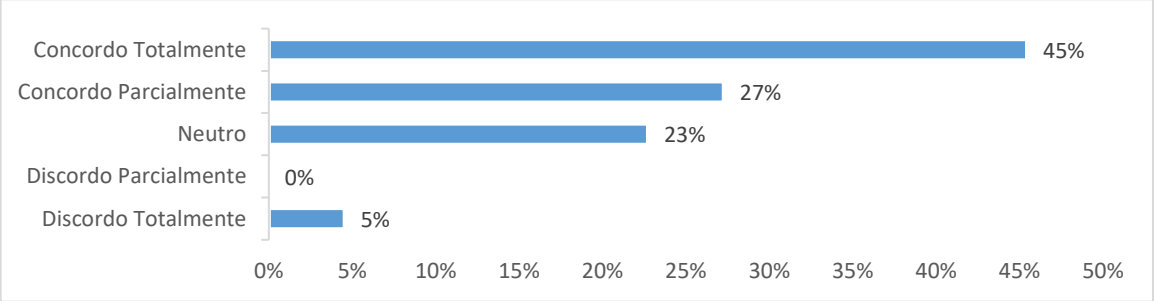
Fonte: Elaborado pelo autor.

Gráfico 10 - Questão 10 - É difícil manter diagramas de sequência sempre atualizados.



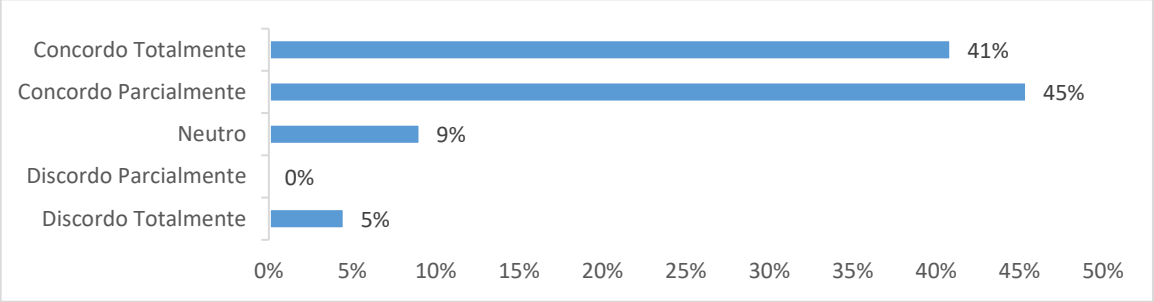
Fonte: Elaborado pelo autor.

Gráfico 11 - Questão 11 - Um diagrama de classes com classes demais possui pouco valor.



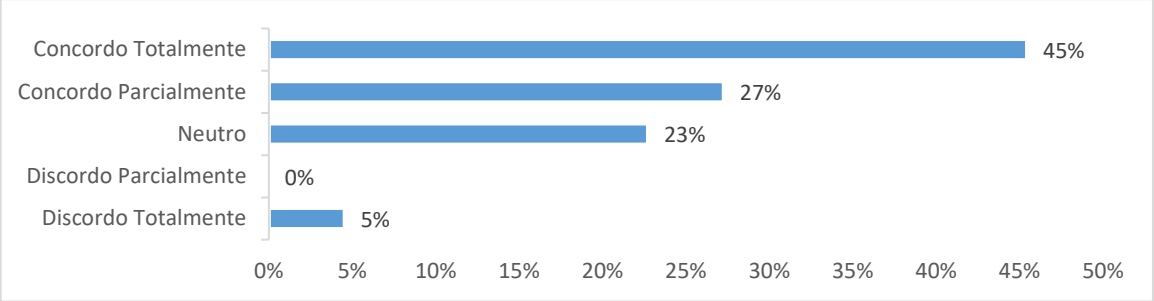
Fonte: Elaborado pelo autor.

Gráfico 12 - Questão 12 - Vários diagramas de classes separados por funcionalidades agregam mais valor do que um único diagrama mostrando todo o sistema.



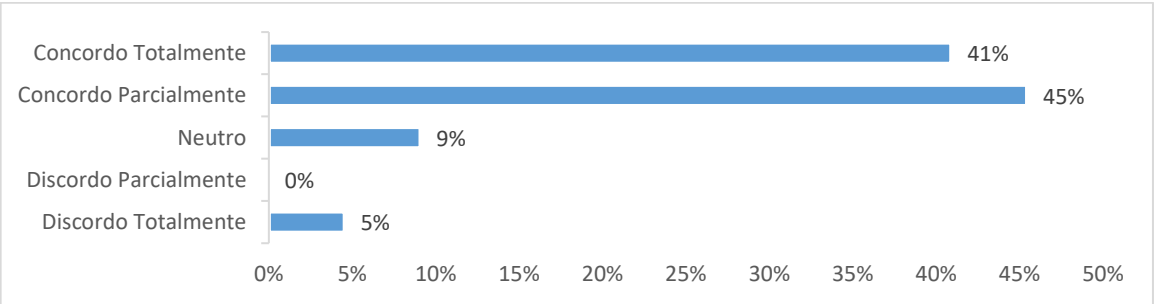
Fonte: Elaborado pelo autor.

Gráfico 13 - Questão 13 - Diagramas UML seriam mais utilizados na indústria se sua confecção fosse automatizada.



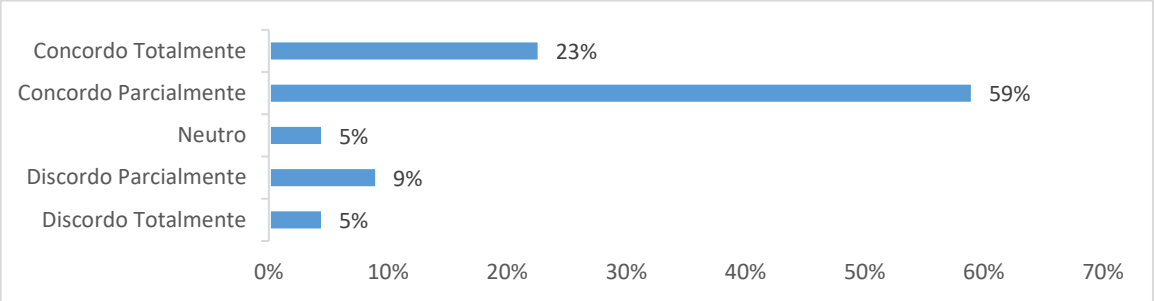
Fonte: Elaborado pelo autor.

Gráfico 14 - Questão 14 - Diagramas UML associados a um sistema e confeccionados automaticamente seriam mais utilizados se filtrassem dados relevantes para cada *feature*.



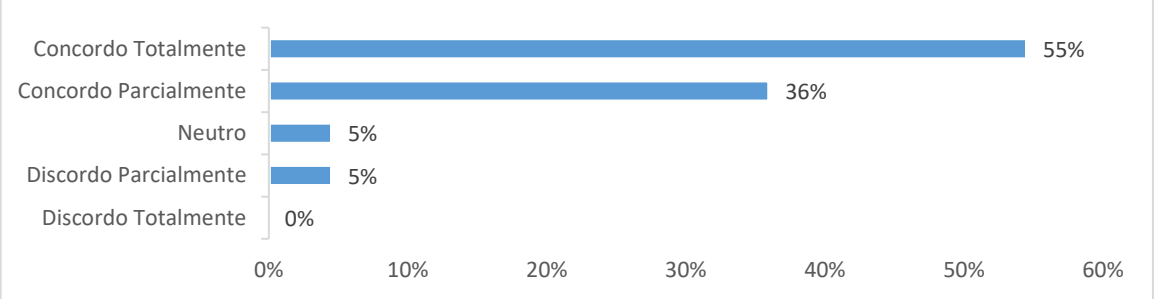
Fonte: Elaborado pelo autor.

Gráfico 15 - Questão 15 - Ter diagramas atualizados automaticamente é útil em metodologias convencionais (ex. *waterfall*).



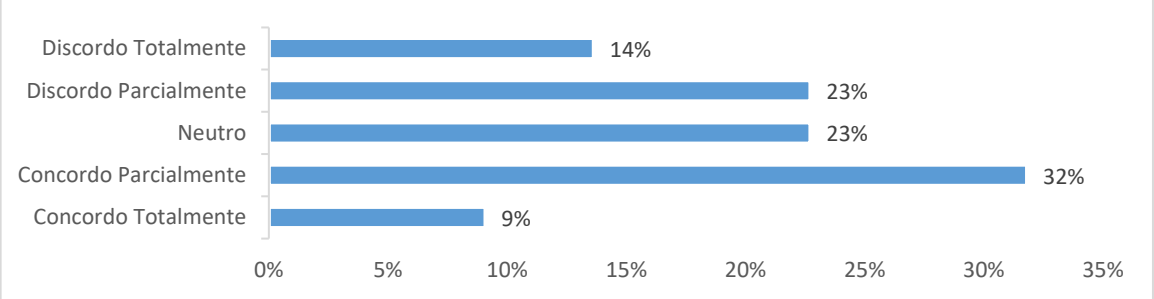
Fonte: Elaborado pelo autor.

Gráfico 16 - Questão 16 - Ter diagramas atualizados automaticamente é útil em metodologias ageis (ex: scrum).



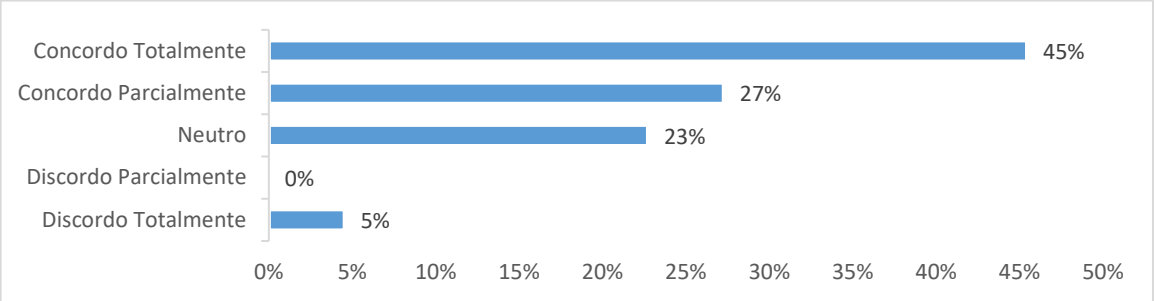
Fonte: Elaborado pelo autor.

Gráfico 17 - Questão 17 - Ter diagramas atualizados automaticamente é útil na fase de manutenção.



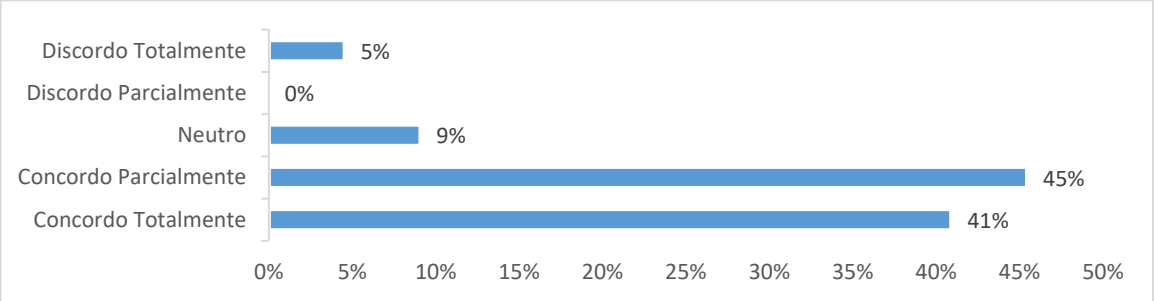
Fonte: Elaborado pelo autor.

Gráfico 18 - Questão 18 - Ao entrar num projeto em andamento, eu gostaria de ter diagramas separados por *feature*.



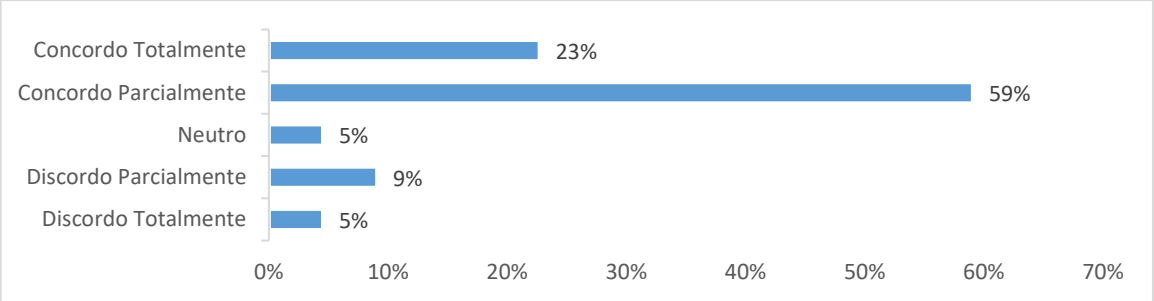
Fonte: Elaborado pelo autor.

Gráfico 19 - Questão 19 - Ao começar a dar manutenção para um sistema, eu gostaria de ter diagramas separados por *feature*.



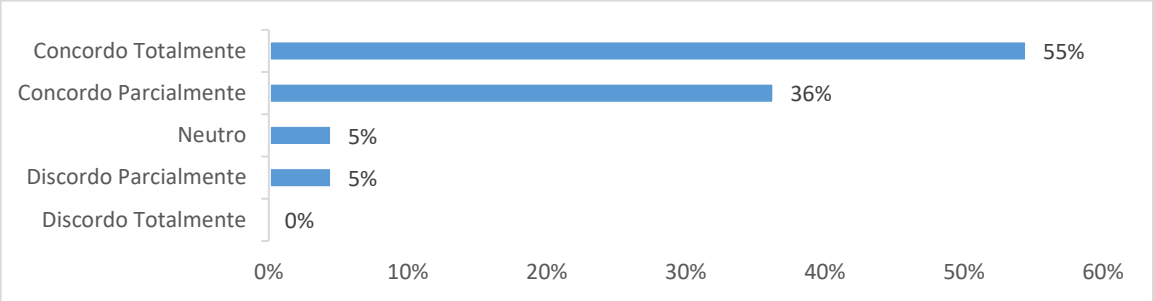
Fonte: Elaborado pelo autor.

Gráfico 20 - Questão 20 - Para codificar uma mudança na regra de negócio, eu gostaria de ter diagramas das *features* envolvidas.



Fonte: Elaborado pelo autor.

Gráfico 21 - Questão 21 - Para corrigir um bug, eu gostaria de ter diagramas da feature com problema.



Fonte: Elaborado pelo autor.