Universidade Federal de Juiz de Fora
Instituto de Ciências Exatas
Bacharelado em Sistemas de Informação

PickClass: Uma ferramenta para definição da ordem de integração de classes

Yan de Paiva Andrade Cunha

JUIZ DE FORA
JULHO, 2023

PickClass: Uma ferramenta para definição da ordem de integração de classes

Yan de Paiva Andrade Cunha

Universidade Federal de Juiz de Fora

Instituto de Ciências Exatas

Ciência da Computação

Bacharelado em Sistemas de Informação

Orientador: Marco Antônio Pereira Araújo

JUIZ DE FORA

JULHO, 2023

PickClass: Uma ferramenta para definição da ordem de integração de classes

Yan de Paiva Andrade Cunha

MONOGRAFIA SUBMETIDA AO CORPO DOCENTE DO INSTITUTO DE CIÊNCIAS EXATAS DA UNIVERSIDADE FEDERAL DE JUIZ DE FORA, COMO PARTE INTEGRANTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE BACHAREL EM SISTEMAS DE INFORMAÇÃO.

Aprovada por:

Marco Antônio Pereira Araújo
Doutor em Engenharia de Sistemas e Computação

Igor de Oliveira Knop
Doutor em Modelagem Computacional

Ronney Moreira de Castro
Doutor em Informática

JUIZ DE FORA
06 DE JULHO, 2023

Dedico este trabalho as minhas filhas,

Elize Cunha dos Reis e Laura Andrade Cunha.

.

Resumo

Observa-se que a orientação a objetos é o paradigma de desenvolvimento de software mais utilizado atualmente. Nesse contexto, uma etapa extremamente importante é a criação e elaboração de um plano de testes bem estruturado, assim como o desenvolvimento desses testes. O objetivo é assegurar a qualidade do produto a ser entregue e reduzir os custos de manutenção futuros, o que consequentemente aumentara a vida útil do sistema.

Portanto, o objetivo deste trabalho é fornecer insumos para auxiliar o desenvolvimento do plano de testes, com foco específico nos Testes de Integração. Uma das principais dificuldades encontradas nessa etapa é identificar a ordem de implementação das classes apresentadas no diagrama de classes, visando reduzir a complexidade durante o desenvolvimento dos testes. O acoplamento entre as classes no diagrama e o processo de seleção das classes prioritárias a serem implementadas, estão diretamente relacionados à complexidade dos testes.

Este trabalho propõe o desenvolvimento de um *framework* para automatizar o processo de identificação da ordem de prioridade das classes, a fim de facilitar a elaboração dos testes de integração. Com base nas dificuldades mencionadas anteriormente e na importância de reduzir a complexidade durante o desenvolvimento, essa proposta busca oferecer uma solução que contribua para uma abordagem mais eficiente e estruturada na etapa de testes de integração.

Palavras-chave: Monografia, Engenharia de Software, Classes, Diagrama de Classes, Ordem de prioridade, Componentes de apoio a Testes, Testes de Integração.

Abstract

It is observed that object-oriented programming is currently the most widely used software development paradigm. In this context, an extremely important step is the creation and elaboration of a well-structured test plan, as well as the development of these tests. The goal is to ensure the quality of the delivered product and reduce future maintenance costs, which will consequently increase the system's lifespan.

Therefore, the objective of this work is to provide inputs to assist in the development of the test plan, with a specific focus on Integration Testing. One of the main difficulties encountered in this step is identifying the implementation order of the classes presented in the class diagram, aiming to reduce complexity during test development. The coupling between the classes in the diagram and the process of selecting the priority classes to be implemented are directly related to the complexity of the tests.

This work proposes the development of a framework to automate the process of identifying the priority order of the classes, in order to facilitate the elaboration of integration tests. Based on the aforementioned difficulties and the importance of reducing complexity during development, this proposal seeks to offer a solution that contributes to a more efficient and structured approach in the integration testing phase.

Keywords: Monograph, Software Engineering, Classes, Class Diagram, Priority Order, Testing Support Components, Integration Testing.

Agradecimentos

Existem algumas pessoas que preciso agradecer por conseguir trilhar todo esse caminho e chegar até aqui; nada disso seria possível sem o esforço de todos vocês.

Lorraine Eduarda, minha amada esposa, que esteve ao meu lado durante todos os altos e baixos da minha graduação. Quando eu precisei de você, pude encontrar carinho, companheirismo, aconchego, amor e segurança. Quando duvidei de mim mesmo, você me desafiou e incentivou, sempre acreditando em mim. Jamais chegaria onde eu cheguei sem você ao meu lado. Obrigado.

Andreia, minha mãe, que incentivou o início da graduação e me mostrou, com base em exemplos, o significado de determinação e foco.

Meus queridos amigos Aline de Paula e Max Honório, quando nem eu acreditava mais, vocês me puxaram para cima e não me deixaram desistir. Muito obrigado.

Meu orientador, Marco Antônio, por toda a paciência didática e solicitude ao longo deste projeto. Admiro imensamente o seu trabalho.

"Aprendi que a coragem não é a ausência do medo, mas o triunfo sobre ele. O homem corajoso não é aquele que não sente medo, mas o que conquista esse medo".

Nelson Mandela

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Abreviações

FI	Fator de integração ou Fator de influência
FIT	Fator de integração tardio
UML	Unified Modeling Language
RNF	Requisito não funcional
RF	Requisito funcional
XML	Extensible Markup Languague
XMI	XML Metadata Interchange
ATCM	Automatic Test Case based on Models
AORD	Aspect and Oriented Relation Diagram
OCP	Open Closed Principle

1 Introdução

Este capítulo tem como objetivo apresentar o tema, delimitar o problema de pesquisa e justificar a relevância do estudo.

1.1 Apresentação do Tema

A evolução contínua da área de desenvolvimento de software tem impulsionado esforços no sentido de aprimorar a qualidade e eficiência dos produtos desenvolvidos. Nesse contexto, os testes de integração desempenham um papel fundamental ao permitir a verificação do funcionamento adequado de funcionalidades mais amplas e complexas do sistema. Além disso, a adoção de práticas sólidas de teste de software tem se mostrado crucial para evitar falhas e garantir a robustez do sistema em diferentes cenários de uso.

Compreender a importância dos testes de integração e contar com ferramentas que facilitem sua implementação e análise são aspectos essenciais para o sucesso do processo de desenvolvimento de software. Os testes de integração, também conhecidos como testes de serviços, ocupam um lugar central na estratégia de testes, situando-se em um nível intermediário da pirâmide de testes, conforme apresentado por Valente (2020). Enquanto os testes de unidade se concentram em verificar o comportamento de pequenas porções de código, como classes individuais, os testes de integração exercitam serviços completos, abrangendo funcionalidades de maior granularidade do sistema. Esses testes envolvem múltiplas classes, frequentemente de pacotes distintos, e dependem de sistemas reais, como bancos de dados e serviços remotos.

Diante da necessidade de testes de integração eficientes e de fácil compreensão, surgiu o desenvolvimento de uma extensão com o propósito de auxiliar na análise e entendimento dos conceitos relacionados a esses testes. Para alcançar esse objetivo, foi selecionada a ferramenta *StarUML*, uma escolha sólida devido à sua posição consolidada no mercado. Além disso, a versão gratuita do *StarUML* oferece recursos adicionais, como a exportação dos dados do diagrama no formato XMI. Essa funcionalidade é especial-

mente útil, pois permite desenvolver extensões que analisam os componentes do diagrama com precisão.

A extensão desenvolvida utiliza um diagrama de classes gerado utilizando o *StarUML*, fornecendo uma visão objetiva e ágil das funcionalidades em teste. Ao utilizar essa extensão, espera-se que a compreensão dos testes de integração seja facilitada, permitindo que desenvolvedores e testadores obtenham uma visão clara das interações entre as classes envolvidas e dos cenários de teste abordados. Isso contribuirá para a detecção precoce de possíveis problemas e para a garantia da qualidade do sistema como um todo.

O presente trabalho, visa explorar a concepção e implementação dessa extensão, bem como sua aplicação em um estudo de caso. Será realizada uma análise crítica dos resultados obtidos, levando em consideração os benefícios e limitações da extensão. Além disso, serão discutidas suas contribuições para a área de desenvolvimento de software e testes de integração, visando oferecer *insights* e diretrizes para futuras melhorias e aprimoramentos.

Portanto, ao fornecer uma abordagem mais clara e eficiente para a compreensão dos testes de integração, espera-se que a extensão aqui proposta se torne uma ferramenta valiosa para os profissionais envolvidos no desenvolvimento de software, contribuindo para a qualidade e eficiência dos processos de teste. Os resultados e conclusões deste trabalho têm o potencial de impulsionar avanços na área, incentivando a adoção de práticas aprimoradas e promovendo um desenvolvimento de software mais robusto e confiável.

1.2 Justificativa

O desenvolvimento de aplicações tornou-se uma tarefa cada vez mais complexa e rigorosa, exigindo a adoção de testes como parte fundamental para garantir a qualidade e o funcionamento adequado do software. No entanto, mesmo com sua importância reconhecida, os testes de integração ainda enfrentam desafios significativos.

Conforme apresentado pelos autores CARDOSO (2006), Ré (2009) e SOUSA et al. (2009), um dos principais obstáculos encontrados no processo de desenvolvimento de testes de integração é a ausência de um sistema automatizado que auxilie e esclareça o processo de construção dos testes de integração. A construção dos testes de integração

envolve a identificação da ordem de precedência das principais entidades e componentes do sistema que devem ser testados em conjunto, visando garantir a integração adequada entre eles.

A falta de um sistema específico para essa finalidade torna o processo de integração das classes mais complexo e suscetível a erros humanos. Além disso, a falta de automação nesse processo pode resultar em atrasos no desenvolvimento e em uma cobertura de testes insuficiente, comprometendo a qualidade do software final.

Diante dessa crescente necessidade de enfoque nos testes de integração e da ausência de um sistema disponível para automatizar e esclarecer a identificação da ordem de precedência das classes a serem integradas, justifica-se a elaboração deste deste trabalho. O principal objetivo deste trabalho é desenvolver um sistema capaz de apoiar os desenvolvedores neste processo.

Para fundamentar essa proposta, foi realizada uma revisão da literatura sobre o tema. Durante essa revisão, constatou-se que apenas um único sistema de suporte para a implementação de testes de integração foi encontrado na literatura. No entanto, esse sistema não está disponível para download atualmente, o que destaca a carência de ferramentas prontamente acessíveis para auxiliar os desenvolvedores nessa etapa crucial do processo de teste.

Dessa forma, torna-se evidente a importância desse trabalho, pois contribuirá para preencher uma lacuna na área de desenvolvimento de software, fornecendo uma ferramenta prática e automatizada para auxiliar no processo de integração das classes durante os testes de serviços. Espera-se que os resultados dessa pesquisa possam ser amplamente aplicados e beneficiar tanto os desenvolvedores em formação quanto as empresas que buscam aprimorar a qualidade de seus produtos por meio de testes eficientes.

1.3 Objetivo

O objetivo deste trabalho é desenvolver uma extensão que facilite a implantação de testes de integração em um sistema específico. A extensão será projetada para interpretar automaticamente as informações presentes em um diagrama de classes no *StarUML*, permitindo a identificação da melhor ordem para a implementação dos testes de integração.

Os resultados serão apresentados de forma clara e visual através de uma planilha, fornecendo aos desenvolvedores uma visualização precisa dos dados obtidos.

1.4 Organização do Trabalho

Este trabalho está organizado em 5 capítulos. O primeiro capítulo é a Introdução, que apresenta a justificativa e os objetivos do projeto. No segundo capítulo, é abordada a revisão da literatura que consiste no processo de seleção dos estudos preliminares essenciais para este trabalho. O terceiro capítulo discute a fundamentação teórica, onde são exploradas as teorias relevantes utilizadas para interpretar os resultados da pesquisa realizada. Em seguida, no quarto capítulo, o Desenvolvimento da Aplicação, seu requisitos funcionais e não funcionais, disponibilização da ferramenta, as tecnologias empregadas na criação da extensão e análises dos resultados alcançados. Por fim, o quinto capítulo consiste na Conclusão, onde são apresentadas as considerações finais do trabalho, bem como sugestões para trabalhos futuros.

2 Revisão de literatura

Este capítulo tem como objetivo apresentar a revisão de literatura realizada para fundamentar este trabalho. A revisão de literatura é uma etapa crucial no processo de pesquisa, pois permite situar o estudo em relação ao conhecimento existente sobre o tema.

Serão apresentados trabalhos relacionados que abordam a identificação da ordem de prioridade das classes no desenvolvimento de testes de integração, destacando as abordagens e os métodos utilizados.

2.1 Motivação

A maioria das pesquisas científicas começa com uma revisão de literatura realizada de forma *ad hoc*. No entanto, se essa revisão não for abrangente e imparcial, terá pouco valor científico. É por esse motivo que é importante considerar o uso de uma revisão sistemática, pois essa abordagem é uma maneira de identificar, avaliar e interpretar todas as pesquisas relevantes para uma pergunta de pesquisa específica (Kitchenham, 2004). Além disso, existem outras razões específicas que justificam o uso da revisão sistemática (Kitchenham, 2004):

- Resumir as evidências existentes sobre uma determinada teoria ou tecnologia;
- Identificar lacunas de pesquisa que precisam ser exploradas, permitindo a definição de áreas que requerem mais investigação;
- Fornecer uma base para novas atividades de pesquisa.

O objetivo desta revisão sistemática é identificar a disponibilidade ou falta de ferramentas relacionadas à determinação da ordem de precedência na integração de classes usando a abordagem *bottom-up*. Além disso, visa validar a necessidade de implementação de uma ferramenta e para fornecer de maneira prática e eficaz a sequência de implementação dos testes de integração.

2.2 Questão de pesquisa

Existem atualmente ferramentas disponíveis no mercado que auxiliam na identificação e na ordem de integração de classes, facilitando o entendimento conceitual e fornecendo orientações objetivas para a implementação dos testes de integração utilizando a abordagem *bottom-up*?

2.3 Aplicação

A Tabela 2.1 apresenta os critérios para a realização da revisão sistemática a partir da questão de pesquisa apresentada.

Tabela 2.1: Critérios da revisão

Critério	**Descrição**
Seleção de Fontes	Serão considerados como candidatos todos os artigos científicos ou ferramentas identificados através da *string* de busca definida nesta revisão de literatura. A pesquisa será realizada no Google Acadêmico, focando no contexto de desenvolvimento de software, engenharia de software, integração de classes e testes de integração.

Continua na próxima página

Tabela 2.1 – *Continuação da tabela*

Critério	**Descrição**
Palavras-chave	• Engenharia de Software; • Componentes de apoio a Testes; • Fator de Integração • FI • Fator de Influência • Fator de Integração Tardia • FIT • Testes de Integração • Software Engineering • Integration testing • *Stubs*
Idioma dos Estudos	Português.
Métodos de busca de fontes	As fontes serão acessadas via web, no contexto desta revisão não será considerada a busca manual.
Listagem de fontes	Google Acadêmico
Tipo dos Artigos	Tecnológico, Teórico, Estudos experimentais.
Critérios de Inclusão e Exclusão de Artigos	Os artigos selecionados devem estar disponíveis online e abordar o ensino de técnicas de integração de classes, testes de integração ou a aplicação prática desses testes.

2.4 Processo de Seleção dos Estudos Preliminares

Foi realizado um processo de refinamento durante a seleção da *string* de busca, que resultando na *string* apresentada a baixo.

```
(``testes de integração''  OR  ``teste de integração''
    AND ((``fator de integração" OR ``FI'')
        AND (``fator de integração tardia'' OR ``FIT'') ))
AND (``ferramentas'' OR ``software"
        OR ``extensão'' OR ``sistema''
        OR ``plugin'' OR ``aplicação")
AND (``engenharia de software'').
```

Após refinar a *string* de busca para obter um conjunto adequado de trabalhos para análise, foram selecionados artigos com base na leitura e verificação dos critérios de inclusão e exclusão estabelecidos. Esse processo foi repetido até alcançar um número adequado de artigos selecionados.

2.4.1 Estratégia de Extração de Informação

Para cada estudo selecionado no processo de seleção, foram extraídas as seguintes informações:

- Título do artigo;
- Autores;
- Fonte;
- Tipo de artigo;
- Trabalhos relacionados.

2.4.2 Sumarização de Resultados

Os resultados foram tabulados e foram realizadas análises para identificar os materiais que abordam as técnicas utilizadas na aplicação dos conceitos de integração de classes

e/ou testes de integração.

Foi utilizado os seguinte critérios para inclusão ou exclusão de artigos, resultando na Tabela 2.2:

- Inclusão de artigos: Foram incluídos artigos com o objetivo de analisar o desenvolvimento e o impacto dos testes de integração. Além disso, foram selecionados artigos que definem ou apresentam heurísticas para o desenvolvimento desses testes. Outro ponto importante para a inclusão desses artigos foi a constatação da ausência de ferramentas adequadas para auxiliar na análise e no desenvolvimento dos testes.

- Exclusão de artigos: Foram excluídos da seleção de artigos aqueles que não têm como foco principal os testes de integração. Também foram excluídos trabalhos que apenas fazem citações sobre o tema, mas não o abordam de maneira apropriada. Além disso, foram excluídos trabalhos que simplesmente reafirmam problemas já evidenciados anteriormente por outros trabalhos selecionados.

A inclusão ou exclusão desses artigos foi realizada com o intuito de manter a relevância e a coerência da revisão de literatura. Dessa forma, foram priorizados os trabalhos que realmente se concentram nos testes de integração, oferecendo análises, estudos empíricos, soluções propostas ou novas perspectivas sobre o assunto.

Tabela 2.2: Quantitativo de artigos selecionados

Base de busca: https://scholar.google.com.br/		
Período: 22/06/2023 à 26/06/2023		
Resultados Obtidos	**Resultados Descartados**	**Resultados Escolhidos**
41	33	**8**

Uma vez que o assunto testes de integração possui um grande campo de estudo. Foi necessário restringir o escopo da busca por meio de uma análise eliminatória dos resumos/abstracts dos trabalhos encontrados. Essa abordagem foi adotada para verificar

a disponibilidade de suporte ferramental gratuito ou pago, relacionado às heurísticas de criação para testes de integração ou integração de classes.

A análise foi realizada manualmente, resultando em 8 trabalhos selecionados. No entanto, nem todos os trabalhos escolhidos apresentam uma ferramenta para auxiliar a integração das classes. Entretanto, os artigos selecionados que não possuem tal ferramenta destacam a dificuldade enfrentada na execução do trabalho devido à sua ausência.

2.4.3 Artigos selecionados

Após a seleção dos artigos, foi realizada uma síntese de seu conteúdo com o objetivo de destacar sua relevância para esta revisão.

Testes de Integração Aplicados a Software Orientado a Objetos: Heurísticas para Ordenação de Classes

O trabalho aborda estratégias e heurísticas para a ordenação de classes em testes de integração de sistemas de controle de vendas. O objetivo é minimizar o número de *stubs* (módulos de simulação) necessários durante os testes. O trabalho discute abordagens baseadas em grafos e algoritmos genéticos, bem como heurísticas alternativas. As heurísticas propostas se baseiam em critérios de precedência, como herança, assinatura de métodos, agregação, navegabilidade, classes de associação e dependência. Foram realizados estudos de caso para analisar o uso das técnicas propostas. O trabalho também introduz os conceitos de fator de influência e fator de integração tardia para auxiliar na ordenação das classes. O autor também ressalta o esforço exacerbado para o cálculo manual de FI e FIT como um problema Lima e Travassos (2004).

FAROL:Uma ferramenta de Apoio à Aplicação de Heurísticas de Ordenação de Classes para Teste de Integração

FAROL é uma ferramenta de suporte automatizado para a aplicação de heurísticas de ordenação de classes no teste de integração de software orientado a objetos. O objetivo é identificar a ordem de integração das classes, visando reduzir o número de *stubs* necessários no processo de teste.

O trabalho também ressalta a importância da atividade de teste de integração das classes para descobrir defeitos na estrutura do software definida durante a fase de projeto. Porém, a identificação da ordem de integração e teste das classes pode ser desafiadora, especialmente devido às dependências entre os componentes.

O trabalho propõe um conjunto de heurísticas, desenvolvidas por Lima e Travassos, que formalizam os critérios de precedência entre as classes. Essas heurísticas são aplicadas diretamente nos diagramas de classe UML do projeto e utilizam a semântica UML para identificar a ordem de integração. Porém o trabalho não disponibiliza a ferramenta para utilização Neto, Lima e Travassos (2005).

Testes de Integração Aplicados a Software Orientado a Objetos: Heurísticas para Ordenação de Classes

O artigo aponta como uma questão crucial ao aplicar testes de integração em software orientado a objetos é decidir a ordem de integração das classes. O autor defende que as classes devem ser integradas uma de cada vez ou, em alguns casos, em pequenos grupos (*clusters*), uma vez que a abordagem de integração *big-bang* demonstra-se inadequada nessas situações. Conceitos como encapsulamento, herança e polimorfismo adicionam complexidade aos testes, exigindo que critérios sejam estabelecidos para quebrar as dependências entre as classes sem aumentar a complexidade (esforço) dos testes.

Este trabalho apresenta um conjunto de heurísticas aplicadas aos diagramas de classes UML, que permitem estabelecer uma ordem de prioridade para o teste de integração das classes que compõem o software, utilizando o número de stubs necessários como medida do esforço requerido Lima e Travassos (2004).

Um Método de Testes de Integração Para Sistemas Baseados em Componentes

Esta tese reafirma a importância dos testes de integração no processo de desenvolvimento de software. Além disso, sugere o uso da heurística dos cálculos de FI e FIT para determinar a ordem de implementação dos testes de integração.

Uma observação destacada no trabalho é a falta de ferramentas adequadas para auxiliar nos cálculos dessa heurística. A ausência desse suporte dificulta a aplicação efici-

ente da abordagem proposta para sistemas de alta complexidade. Também é evidenciado a tendência para soluções componentizadas.

Outro ponto enfatizado é o impacto significativo dos testes de integração durante todo o ciclo de desenvolvimento de software. Esses testes desempenham um papel crucial na garantia da qualidade do sistema, identificando problemas de integração entre os componentes e prevenindo possíveis falhas no sistema final Andrade (2017).

Teste de Integração para Sistemas Baseados em Componentes.

Como a conectividade entre os componentes é um ponto chave do Software Baseado em Componentes, a verificação da interação entre os componentes da aplicação torna-se essencial para obter um produto final de qualidade. Dessa forma, são realizados testes com a finalidade de garantir que a combinação entre os componentes do sistema produza um comportamento esperado. Esses testes são chamados de teste de integração. Apesar de que esforços consideráveis têm sido realizados, ainda existe uma grande carência por métodos e técnicas efetivas de teste de integração de componentes que cubra todas as etapas necessárias, a fim de se ter um processo de teste completo do ponto de vista de componentes. O objetivo deste trabalho é propor um método para testar a integração entre os componentes de um sistema.

Os artefatos de teste são gerados a partir de especificações em UML, que é uma linguagem de especificação largamente utilizada, facilitando o uso do método por grande parte da comunidade da engenharia de software. O método fornece potencial para automação, uma vez que disso depende o seu uso na prática. Um estudo de caso também é apresentado a fim demonstrar a aplicação do método Gouveia et al. (2004).

Construção automatizada de casos de teste usando engenharia dirigida por modelos

O surgimento das abordagens dirigidas por modelos fornece uma nova alternativa para o gerenciamento da complexidade do desenvolvimento de software, para criação de testes de software, para automação dos processos de testes e para fornecimento da ampla reutilização de modelos desenvolvidos durante a fase de análise dos requisitos e projeto de

software, reduzindo a possível injeção de erros e o tempo de desenvolvimento do software. No entanto, com a utilização das abordagens dirigidas por modelos, possíveis erros podem ser injetados na criação das regras de transformação para implementar um determinado sistema de software. Propõe-se neste trabalho metamodelos de testes, uma metodologia e um framework ATCM (Automatic Test Case based on Models) com a finalidade de gerar casos de teste a fim de testar o código-fonte gerado por uma abordagem dirigida por modelos.

Um protótipo do *framework* ATCM foi desenvolvido, fornecendo ferramentas que minimizam a injeção de erros durante a geração dos casos de teste, tornando esta tarefa menos dependente de pessoas e menos propensa a erros reduzindo o tempo de desenvolvimento e provendo maior qualidade e eficiência nos casos de teste gerados SOUSA et al. (2009).

Uma contribuição para a minimização do número de stubs no teste de integração de programas orientados a aspectos.

A programação orientada a aspectos é uma abordagem que utiliza conceitos da separação de interesses para modularizar o software de maneira mais adequada. Com o surgimento dessa abordagem vieram também novos desafios, dentre eles o teste de programas orientados a aspectos. Duas estratégias de ordenação de classes e aspectos para apoiar o teste de integração orientado a aspectos são propostas nesta tese. As estratégias de ordenação tem o objetivo de diminuir o custo da atividade de teste por meio da diminuição do número de *stubs* implementados durante o teste de integração. As estratégias utilizam um modelo de dependências aspectuais e um modelo que descreve dependências entre classes e aspectos denominado AORD *(Aspect and Oriented Relation Diagram)* também propostos em seu trabalho.

Tanto o modelo de dependências aspectuais como o AORD foram elaborados a partir da sintaxe e semântica da linguagem *AspectJ* [1] Para apoiar as estratégias de ordenação, idealmente aplicadas durante a fase de projeto, um processo de mapeamento de modelos de projeto que usam as notações UML e MATA [2] para o AORD é proposto neste

[1]Uma extensão da linguagem de programação Java que permite a programação orientada a aspectos
[2]Uma ferramenta para modelagem orientada a aspectos baseada em transformação de grafos.

trabalho. O processo de mapeamento é composto de regras que mostram como mapear dependências advindas da programação orientada a objetos e também da programação orientada a aspectos. Como uma forma de validação das estratégias de ordenação, do modelo de dependências aspectuais e do AORD, um estudo exploratório de caracterização com três sistemas implementados em *AspectJ* foi conduzido. Durante o estudo foram coletadas amostras de casos de implementação de *stubs* e *drivers* de teste.

Os casos de implementação foram analisados e classificados. A partir dessa análise e classificação, um catálogo de *stubs* e *drivers* de teste é apresentado Ré (2009).

Heurísticas para identificação da ordem de integração de classes em testes aplicados a software orientado a objetos

O objetivo principal desta tese é definir heurísticas e um processo para apoiar engenheiros de software na identificação da ordem de integração de classes em testes de software. A proposta é aplicar as heurísticas diretamente no diagrama de classes UML, de forma a obter uma ordem de integração com o menor esforço de teste possível.

Inicialmente, a pesquisa analisou estudos anteriores sobre testes de integração e estratégias existentes na literatura. Observou-se que uma proposta anterior utilizava a semântica da UML para identificar a ordem de integração, mas não abordava certas situações especiais, como dependências cíclicas, e não oferecia uma solução geral com menor esforço de teste. Foram desenvolvidos novos tratamentos para essas situações e aprimorado o processo de aplicação das heurísticas.

Para avaliar a efetividade das heurísticas e do processo, foram realizados estudos de caso acadêmicos e industriais, envolvendo estudantes de mestrado e doutorado e profissionais de uma organização naval. Os resultados obtidos permitiram avaliar as heurísticas e identificar a necessidade de automatizar o processo de aplicação.

Por fim, foi desenvolvida uma ferramenta chamada FAROL que também foi apontada nesta revisão de literatura anteriormente LIMA (2005).

2.5 Considerações da revisão de literatura

Após analisar os artigos, foi possível identificar a existência de uma ferramenta desenvolvida chamada FAROL (NETO; LIMA; TRAVASSOS, 2005). No entanto, a ferramenta não está disponibilizada para download. Dito isso, não foi possível identificar outra ferramenta de código aberto disponível que tenha como objetivo estabelecer uma ordem de precedência na integração de classes com abordagem *bottom-up*.

Foi possível também corroborar a eficácia da heurística utilizada no decorrer deste trabalho para cálculo de FI, FIT e identificação da ordem de precedência de integração das classes, com base em diagramas de classes. Outro ponto relevante é o impacto diretamente proporcional da quantidade de *stubs* na complexidade do desenvolvimento de testes de integração.

Não foram selecionados artigos em inglês, pois não foi encontrado nenhum artigo que faça alusão à implementação de um ferramental que objetive a análise de integração de classes ou testes de integração.

3 Fundamentação Teórica

Este capítulo apresenta os conceitos fundamentais para o entendimento do trabalho proposto. Serão apresentados conceitos de UML, testes de software e as heurísticas utilizadas. Além disso, serão apresentados todos os fundamentos teóricos necessários para o desenvolvimento da extensão proposta como resultado deste trabalho.

3.1 Diagramas *Unified Modeling Language* - UML

No atual cenário da engenharia de software, a necessidade de projetar sistemas de informação complexos de forma clara e eficiente tem se tornado cada vez mais importante. Nesse contexto, a *Unified Modeling Language* (UML) destaca-se como uma linguagem de modelagem gráfica que proporciona uma abordagem padronizada e visualmente compreensível para a análise e projeto de sistemas.

Segundo Valente (2020), é uma notação gráfica para modelagem de software. A linguagem define um conjunto de diagramas para documentar e ajudar no *design* de sistemas de software, particularmente sistemas orientados a objetos.

De acordo com a perspectiva de Fowler (2004), a UML pode ser utilizada de três formas distintas: como projeto, como linguagem de programação ou como esboço.

Projeto: Nessa abordagem, a UML é usada como uma ferramenta para projetar e modelar sistemas de software. Ela permite que os desenvolvedores criem representações visuais dos componentes do sistema, suas interações e suas estruturas. A UML é usada como um meio de capturar os requisitos, definir a arquitetura e planejar a implementação do sistema.

Linguagem de Programação: Embora a UML seja principalmente uma linguagem gráfica para modelagem, é possível estender seu uso como uma linguagem de programação. A UML pode ser usada para expressar lógica de negócios e comportamentos de sistema por meio de diagramas de atividades, estado ou sequência. No entanto, essa forma de uso é menos comum e não é o objetivo principal da UML.

Esboço: A terceira forma de uso da UML é como um meio de comunicação entre membros da equipe de desenvolvimento ou entre desenvolvedores e *stakeholders*. Nesse caso, a UML é usada para fazer esboços rápidos e informais de ideias, conceitos ou soluções potenciais. Esses esboços podem ser representados por diagramas simples e rápidos, que permitem uma comunicação efetiva entre as partes envolvidas.

3.1.1 Diagrama de classes

De acordo com Valente (2020) os diagramas de classes são os diagramas mais usados da UML. Oferecem representação gráfica para um conjunto de classes e além disso promovem informações sobre atributos, métodos e relacionamentos.

Atributos são características ou propriedades das classes que representam dados associados a essas classes. Eles descrevem os estados ou características que uma instância da classe pode ter. Segundo Booch, Rumbaugh e Jacobson (2005), os atributos podem ser representados na UML por meio de campos ou propriedades das classes. Eles podem ter tipos de dados específicos, como números inteiros, strings ou objetos de outras classes.

Métodos, também conhecidos como operações, representam o comportamento das classes. Eles descrevem as ações que as instâncias da classe podem realizar ou as operações que podem ser executadas nos objetos da classe. Segundo Fowler (2004), os métodos podem ser representados na UML por meio de caixas de texto ou listagens dentro das classes, indicando seus nomes, parâmetros e tipos de retorno.

Os relacionamentos, por sua vez, representam as associações entre as classes. Eles descrevem como as classes se relacionam umas com as outras e como as instâncias das classes podem interagir. Existem diferentes tipos de relacionamentos, como associação, agregação, composição e herança. Larman (2004) destaca que esses relacionamentos podem ser expressos na UML por meio de linhas e setas que conectam as classes, com multiplicidades e papéis indicando a natureza e a cardinalidade da relação.

Portanto, de acordo com as definições apresentadas por autores como Fowler (2004), Larman (2004) e Booch, Rumbaugh e Jacobson (2005), atributos são as características ou propriedades das classes, métodos são as operações ou comportamentos das classes, e os relacionamentos representam as associações entre as classes. Esses conceitos

são fundamentais na modelagem de sistemas utilizando a UML e fornecem uma representação clara e estruturada do sistema em desenvolvimento.

Relacionamentos

A seguir, são apresentadas as definições de cada relacionamento com base em referências bibliográficas relevantes.

- Associação: A associação é um relacionamento básico que descreve a conexão entre duas classes. Segundo Larman (2004), ela indica que uma classe usa ou se relaciona com a outra de alguma forma. A associação pode ter multiplicidade, representando quantos objetos de uma classe podem se relacionar com objetos da outra classe.
  - Segundo Valente (2020) Associação biderecional ocorre quando existe uma relação entre duas classes de forma simultânea.
  - De acordo com Fowler (2004) uma classe associativa, também conhecida como classe de associação, é uma classe que é introduzida para representar um relacionamento entre outras classes. Ela é criada quando um relacionamento direto entre duas classes não é suficiente para expressar todas as informações necessárias.

- Agregação: A agregação é um tipo de relacionamento que indica uma associação de todo-parte entre classes. De acordo com Booch, Rumbaugh e Jacobson (2005), ela representa uma relação em que uma classe é composta por outras classes, mas as partes podem existir independentemente do todo. Essas partes podem pertencer a vários agregados ao mesmo tempo.

- Composição: A composição é semelhante à agregação, porém com um grau maior de dependência entre as classes envolvidas. Segundo Larman (2004), a composição representa uma relação em que uma classe é composta por outras classes e é responsável por sua criação e destruição. As partes não podem existir sem o todo e são exclusivas desse todo.

- Generalização ou Herança: A herança é um relacionamento que representa a especialização de uma classe a partir de outra classe mais genérica, também conhecida como superclasse ou classe base. De acordo com Booch, Rumbaugh e Jacobson (2005), a herança permite que as classes herdem atributos e métodos da superclasse, além de adicionar novos atributos e métodos específicos.

3.2 *StarUML*

O StarUML é uma ferramenta de modelagem de sistemas mundialmente reconhecida e utilizada, conforme pode-se observar na seção *Customers* em seu site (StarUML web site, 2023) [3]. Esta ferramenta nos permite criar diversos diagramas UML de forma rápida e de fácil usabilidade, para auxiliar no desenvolvimento de software.

Por se tratar de uma ferramenta de alta popularidade, possui suporte ao desenvolvimento de extensões por terceiros, e um documento bem estruturada (StarUML documentation, 2023).

3.2.1 Extensão

O princípio de aberto/fechado (OCP: Open Closed Principle) definido por (MEYER, 1997) nos mostra que um sistema deve ser aberto a extensões e fechado para modificações, ou seja, caso alguém queira desenvolver uma nova funcionalidade para um sistema que esteja condizente com este princípio, deve desenvolver uma extensão totalmente isolada para integrar com a aplicação original.

Dito isto, extensão (*plugin* ou módulo) nada mais é do que uma funcionalidade desenvolvida em um momento posterior ao lançamento do software, a fim de possibilitar uma nova expêriencia para o usuário durante a utilização da ferramenta.

3.3 Testes

Estima-se que cerca de 40% dos custos do desenvolvimento de um software são destinados a testes, porém, este custo pode variar de acordo com o contexto e as necessidades

[3] https:// docs. staruml.io/

Sommerville (2018).

É possível observar que uma política de testes bem estabelecida possui um impacto direto no esforço dedicado à manutenção e evolução do software. Atualmente, esse esforço é distribuído da seguinte forma, na figura 3.1:

Figura 3.1: Distribuição do esforço de manutenção.

De acordo com (SOMMERVILLE, 2018), os testes desempenham um papel fundamental na detecção precoce de *bugs*. O que resulta em 24% do esforço destinado a reparos.

Conforme destacado por (FOWLER, 2020), para que um software seja passível de refatoração, os métodos a serem refatorados devem ser abrangidos por testes, garantindo o correto funcionamento do código após sua reestruturação. O autor também ressalta a importância dos testes no desenvolvimento de novas funcionalidades, a fim de evitar impactos nas funcionalidades existentes. Essa constatação leva a entender que 58% do esforço é direcionado para a adição ou modificação de funcionalidades.

Dito isto, os testes são classificados em três níveis, do mais básico para o mais complexo na Figura Figura 3.2:

- **Testes de unidade ou testes unitários**: Como o próprio nome já faz referência, os testes unitários, visam testar a menor parte possível do software. Normalmente é aplicado à pequenas funcionalidades dentro do escopo de uma única classe.

  Sommerville (2018) define testes de unidade da seguinte forma: O teste de unidade

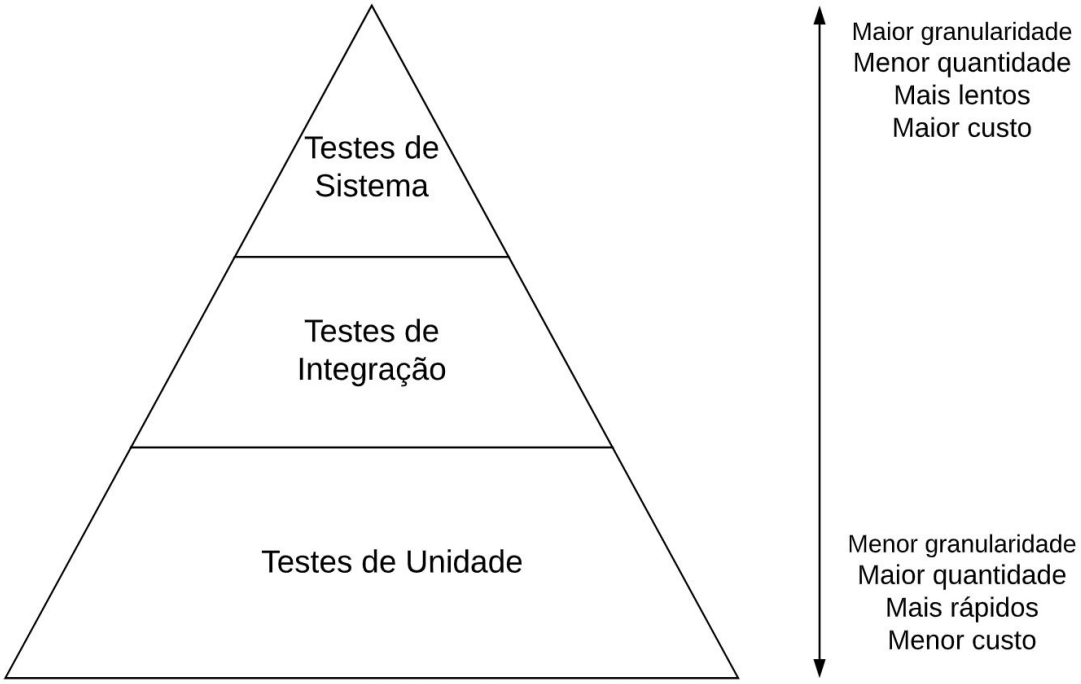

Figura 3.2: Piramide de testes

envolve a avaliação de componentes individuais de um programa, como métodos ou classes. Esses componentes podem ser considerados como as partes mais básicas do sistema. Os testes de unidade para esses componentes devem incluir a execução de chamadas para essas rotinas com diferentes conjuntos de parâmetros de entrada.

O autor ainda completa, ao realizar testes em classes, é importante criar testes que cubram todas as características do objeto em questão. Isso inclui testar todas as operações associadas ao objeto, definir e verificar os valores de todos os atributos relacionados e colocar o objeto em todos os estados possíveis. Essa abordagem abrange a simulação de todos os eventos que possam causar uma mudança de estado no objeto.

Quando os testes de unidade são desenvolvidos de acordo com os princípios mencionados anteriormente, o funcionamento correto das classes é assegurado. No entanto, essa garantia é dada de forma isolada, ou seja, só é garantido o funcionamento individual de cada classe, conforme Figura3.3.

- **Testes de integração, testes de serviços ou testes de componentes**: segundo Sommerville (2018), os testes de componentes têm como objetivo avaliar a interação

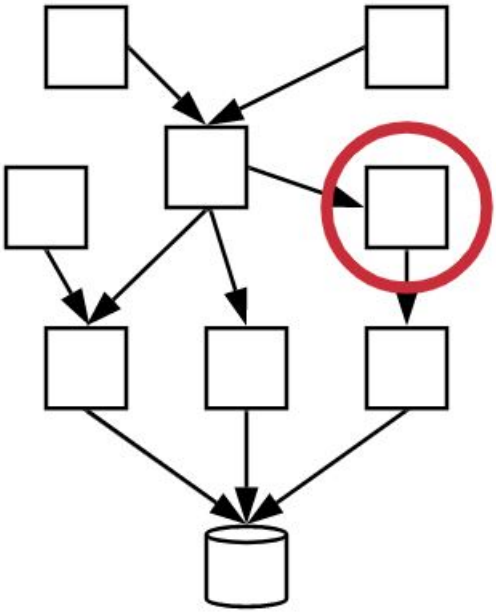

Figura 3.3: Testes unitários.

entre vários objetos, para este cenário podemos presumir que os testes de unidade de cada objeto já foram implementados e validados.

É necessário ajustar a perspectiva em relação aos testes, tanto no que diz respeito aos elementos a serem testados quanto à forma como são conduzidos. Agora o foco está na validação completa de um determinado serviço, o que requer uma comunicação efetiva entre os diferentes objetos que o compõem, conforme podemos observar na Figura 3.4.

Um dos maiores desafios enfrentados nessa fase dos testes é estabelecer a ordem correta de integração dos objetos para a realização dos testes de forma eficiente.

A literatura nos traz duas abordagens amplamente reconhecidas para auxiliar a resolução deste desafio, *top-down* e *bottom-up*.

Top-down, segundo Maxim e PRESSMAN (2021) a abordagem *top-dowm* dar-se-á pela implementação de 4 passos principais:

1. O módulo de controle principal é utilizado como um testador *(test driver)*, e todos os componentes diretamente subordinados ao módulo de controle principal substituem os pseudocontrolados *(stubs)*;

2. Dependendo da abordagem de integração selecionada, componentes pseudocon-

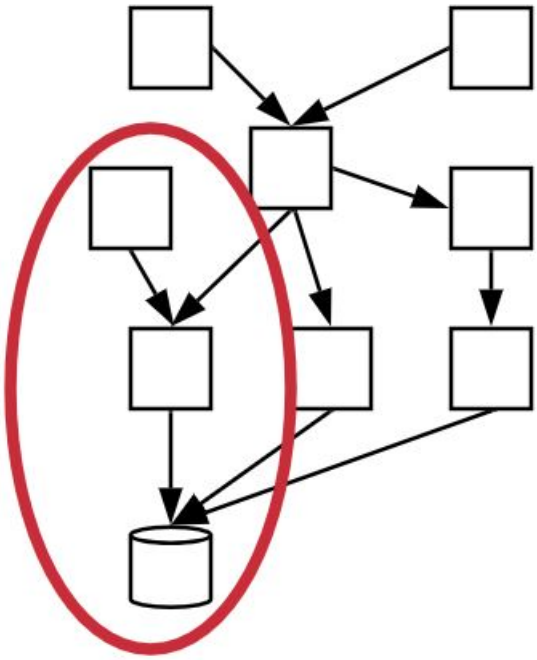

Figura 3.4: Testes de integração.

trolados *(stubs)* subordinados são substituídos, um de cada vez, pelos componentes reais;

3. Os testes são feitos à medida que cada componente é integrado;

4. Ao fim de cada conjunto de testes, outro pseudocontrolado é substituído pelo componente real;

O processo continua a partir do passo 2 até que toda a estrutura do programa esteja concluída.

Bottom-up, os autores também classificam a abordagem *bottom-up* pela implementação de 4 passos:

1. Componentes de baixo nível são combinados em agregados.

2. Um pseudocontrolador (um programa de controle para teste) é escrito para coordenar entrada e saída do caso de teste.

3. O módulo, como um conjunto de componentes, é testado.

4. Os pseudocontroladores (*stubs*) são removidos, e os agregados são combinados movendo-se para cima na estrutura do programa.

Conforme pode-se observar na Figura 3.5

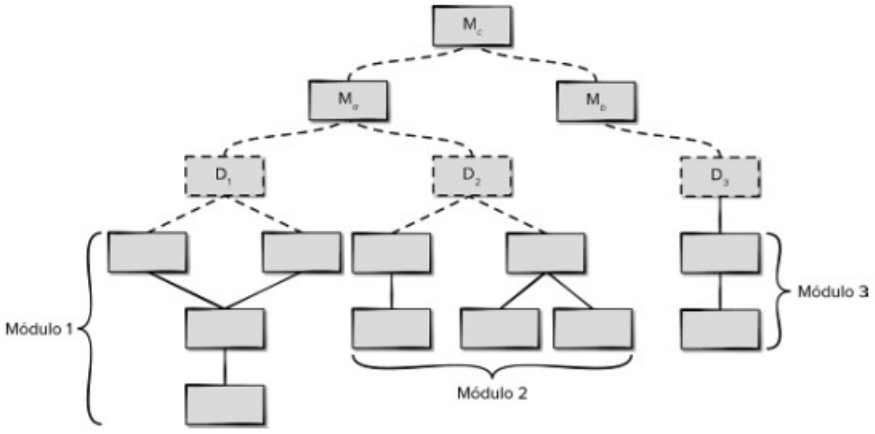

Figura 3.5: Testes de integração.

- **Testes de Sistema, testes ponta a ponta ou testes de validação**: É necessário ajustar perspectiva mais uma vez em relação aos testes. Desta vez, o foco está em garantir que o software atenda a todos os requisitos estabelecidos durante a fase de documentação e apresente um comportamento esperado com base nos casos de uso previamente definidos.

  Esse tipo de teste só faz sentido após o desenvolvimento e validação de todos os testes de integração da aplicação. Nesse momento, os testes de unidade já asseguraram o comportamento correto das classes e métodos, enquanto os testes de integração garantiram que os módulos estejam funcionando conforme o esperado, sem a introdução inadvertida de erros durante a integração dos mesmos. Conforme retratado na Figura 3.6.

  De acordo com Sommerville (2018), o teste de sistema tem como objetivo verificar a integração de todos os componentes do sistema, incluindo o uso de bibliotecas desenvolvidas por terceiros que foram incorporadas durante a fase de desenvolvimento. O autor enfatiza a importância desses testes na identificação de comportamentos emergentes, que só se tornam visíveis quando há interação entre todos os elementos do sistema.

  já para Maxim e PRESSMAN (2021)

  > “ Como todas as etapas de teste, a validação tenta descobrir erros, mas o foco está no nível de requisitos – em coisas que carão imediatamente

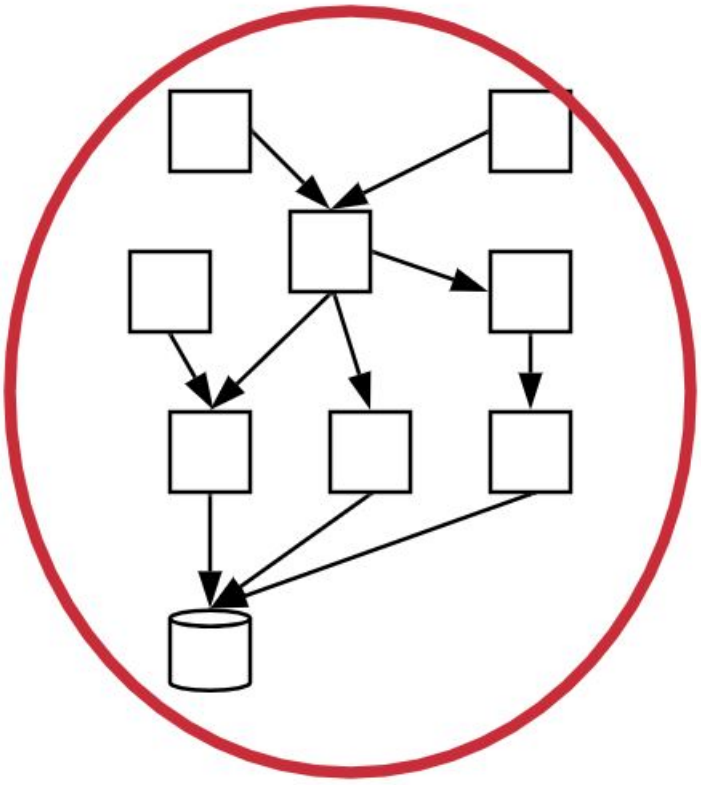

Figura 3.6: Testes de Sistema

aparentes para o usuário. O teste de validação começa quando termina o teste de integração, quando os componentes individuais já foram exercitados, o software está completamente montado como um pacote e os erros de interface já foram descobertos e corrigidos. No nível de validação ou de sistema, a distinção entre diferentes categorias de software desaparece. O teste focaliza ações visíveis ao usuário e saídas do sistema reconhecíveis pelo usuário."

3.4 Heurística para determinar a ordem de integração das classes

Para os propósitos deste trabalho, é necessário definir e utilizar uma heurística para identificar a ordem de precedência das classes com base em seus diagramas de classes, analisando seus relacionamentos.

LIMA (2005) define esta heurística conforme apresentado abaixo por meio de

tópicos.

- **Generalização ou Herança**: A superclasse concreta deve ser integrada *antes* das subclasses. Em outras palavras, só será possível integrar as subclasses *depois* da integração da superclasse.

- **Agregação**: As classes *parte* devem ser integradas *antes* da classe *todo*. Em outras palavras, só será possível integrar a classe todo *depois* da integração de todas as classes partes.

- **Composição**: As classes *parte* devem ser integradas *antes* da classe *todo*. Em outras palavras, só será possível integrar a classe todo *depois* da integração de todas as classes partes.

- **Associação direta**: No caso de uma associação direta, é aplicada a regra de navegabilidade, ou seja, a classe que se tornou atributo deve ser integrada *antes* da classe que consome esse atributo.

- **Associação Bidirecional**: Nesse caso, é analisado como duas associações diretas, ou seja, em uma associação bidirecional entre as classes A e B, a classe A deve ser integrada *antes* da classe B, e, por sua vez, a classe B deve ser integrada *antes* da classe A.

- **Classe associativa**: As classes que deram origem a classe associativa devem ser integradas *antes* da classe em questão.

Uma vez definida a regra de precedência para a integração das classes ou desenvolvimento dos testes de integração, observa-se a necessidade do surgimento dos *stubs*.

3.5 Stubs

Para Sommerville (2018) um *stub* é um pequeno programa ou módulo de código que substitui um componente real durante os testes. O objetivo principal de um *stub* é simular o comportamento e as interfaces do componente real de forma controlada. Dessa forma,

os *stubs* permitem que os testes sejam realizados mesmo quando os componentes reais ainda não foram desenvolvidos ou estão indisponíveis.

Além de reafirmar a definição apresentada anteriormente Pressman (2005), ainda completa apresentando o seu contexto de utilização, informando que os *stubs* são frequentemente utilizados em testes de integração, onde várias partes do sistema são combinadas para verificar a interação e a comunicação corretas entre elas. Um *stub* é criado para representar um componente que ainda não está pronto para ser utilizado no teste. Ele recebe as mesmas entradas e gera as mesmas saídas esperadas, mas sua implementação é simplificada e não possui a funcionalidade completa do componente real.

Dito isto, um *stub* é utilizado quando é necessário realizar o teste de integração de uma classe A, porém ela depende da ordem de precedência de outra classe B. Em outras palavras, o teste de integração da classe A só deveria ser implementado após o teste de integração da classe B. No entanto, de acordo com a heurística escolhida, é necessário implementar o teste de integração da classe A neste momento.

Nesta situação, é necessário criar um *stub* para simular o comportamento da classe B durante o teste de integração da classe A. O *stub* de B fornecerá as respostas esperadas e permitirá que o teste de integração da classe A seja realizado mesmo sem a implementação completa da classe B. Isso ajuda a cumprir as exigências das heurísticas e realizar o teste necessário no momento adequado.

Pressman (2005) alerta para dois pontos de atenção durante a utilização dos *Stubs*:

A presença de muitos *stubs* pode aumentar a complexidade dos testes, uma vez que cada *stub* precisa ser configurado corretamente para reproduzir o comportamento esperado do componente real. Além disso, é necessário garantir que os *stubs* estejam sincronizados adequadamente e em conformidade com as interfaces e comportamentos esperados.

O uso de um grande número de *stubs* também pode aumentar a carga de trabalho para a equipe de teste, uma vez que cada *stub* precisa ser desenvolvido, mantido e atualizado à medida que o sistema evolui. Isso pode resultar em um aumento do esforço e recursos necessários para realizar os testes de integração.

3.6 Heurísticas FI e FIT

A literatura apresenta várias heurísticas que auxiliam na determinação da ordem de precedência das classes durante o processo de integração. Neste trabalho, utilizaremos o cálculo do FI (Fator de Integração) e do FIT (Fator de Integração Tardia), conforme as definições a seguir:

O FI é um indicador que quantifica a interação entre as classes. Ele leva em consideração as dependências entre as classes, permitindo determinar a ordem de integração com base nos valores identificados.

Por sua vez, o FIT avalia a interação temporal entre as classes. Ele leva em conta as dependências temporais existentes entre as classes e pode ser usado para definir a ordem de integração com base nos valores identificados.

Ao utilizar esses cálculos, nossa abordagem considerará tanto as dependências funcionais quanto as temporais, proporcionando uma visão abrangente para determinar a ordem de precedência das classes durante o processo de integração. Isso contribuirá para um desenvolvimento de software mais eficiente e confiável, garantindo uma integração adequada entre as classes.

A tese LIMA (2005) define FI da seguinte forma:

> "O Fator de Influência (FI) é utilizado para quantificar a relação de precedência entre as classes, representando o número de classes que precisam ser testadas depois da classe em análise ser testada, ou seja, o número de classes sobre as quais a classe em análise tem precedência. Deve ser definido considerando apenas as precedências diretas da classe em análise."

A tese LIMA (2005) define FIT da seguinte forma:

> "O Fator de Integração Tardia (FIT) tem a intenção de capturar a idéia do tempo de integração das classes. O FIT indica o momento em que a classe deve ser considerada para integração e teste, em relação as demais. O FIT é calculado pela soma dos valores dos Fatores de Influência (FI) das classes com precedência direta sobre a classe em análise. Quanto maior o valor de FIT, mais tarde a classe deverá ser testada, pois maior será a dependência desta

classe em relação as demais."

3.7 Requisitos

Os requisitos de um sistema desempenham um papel fundamental no direcionamento do seu desenvolvimento, pois delimitam o escopo do projeto e especificam suas restrições e características. Além disso, eles definem os comportamentos esperados do sistema.

Esses requisitos podem ser classificados em dois grupos principais: requisitos funcionais e requisitos não funcionais.

Os requisitos não funcionais abrangem aspectos como desempenho, espaço, confiabilidade, robustez, usabilidade e portabilidade. Esses requisitos se referem às características que o sistema deve possuir, mas não estão diretamente relacionados às funcionalidades específicas.

Por outro lado, os requisitos funcionais englobam todas as funcionalidades que o sistema deve oferecer. Eles estão relacionados às ações que o sistema deve ser capaz de realizar e às expectativas de comportamento definidas para o software.

Essa classificação dos requisitos é amplamente utilizada na engenharia de software, conforme proposto por Valente (2020).

Ainda segundo Valente (2020) os requisitos não funcionais podem ser classificados conforme a Figura 3.7:

Além dos requisitos não funcionais mencionados anteriormente, é importante destacar que o autor Pressman (2005) também inclui a manutenibilidade como um requisito não funcional.

Pressman (2005) define a manutenibilidade como a facilidade de realizar modificações, correções e melhorias no sistema ao longo do tempo. Isso envolve aspectos como a modularidade do código, a legibilidade do código-fonte, a existência de documentação adequada, a facilidade de teste e depuração, entre outros fatores.

Garantir a manutenibilidade de um sistema é fundamental para facilitar sua evolução e aprimoramento contínuos, permitindo que seja adaptado às mudanças nas necessidades dos usuários e no ambiente. Portanto, esse requisito não funcional desempenha um papel crucial na sustentabilidade e longevidade de um sistema de software.

Requisito Não-Funcional	Métrica
Desempenho	Transações por segundo, tempo de resposta, latência, vazão (throughput)
Espaço	Uso de disco, RAM, cache
Confiabilidade	% de disponibilidade, tempo médio entre falhas (MTBF)
Robustez	Tempo para recuperar o sistema após uma falha (MTTR); probabilidade de perda de dados após uma falha
Usabilidade	Tempo de treinamento de usuários
Portabilidade	% de linhas de código portáveis

Figura 3.7: Requisitos não funcionais

4 Desenvolvimento da aplicação

Este capítulo tem como objetivo fornecer uma detalhada visão técnica sobre o desenvolvimento da aplicação, abrangendo seus requisitos funcionais e não funcionais, bem como informações sobre sua utilização.

4.1 Escopo do Sistema

Este trabalho tem como objetivo desenvolver uma extensão para o StarUML, cujo propósito é analisar os diagramas de classes de um módulo específico. A abordagem Bottom-up é utilizada para determinar a ordem de integração das classes presentes no diagrama. Como resultado, será gerado uma planilha eletrônica, no padrão XLSX, contendo todos os cálculos de FI, FIT e Stubs necessários, além da ordem de integração para as classes.

4.2 Requisitos

Os requisitos funcionais e não funcionais mínimos para o desenvolvimento desta aplicação são:

4.2.1 Requisitos Não Funcionais

Foram elicitados os seguintes requisitos não funcionais (RNF) para este trabalho:

1. RNF-01: A extensão deve ser acessível por pelo menos dois locais distintos no StarUML, visando facilitar sua utilização;
2. RNF-01.1: A extensão deve ser acessível pela barra superior de ferramentas presente no StarUML;
3. RNF-01.2: A extensão deve ser acessível pelo menu de contexto do diagrama no StarUML;
4. RNF-02: A extensão deve ser disponibilizada em um repositório aberto.

4.2.2 Requisitos Funcionais

Foram elicitados os seguintes requisitos funcionais (RF) para este trabalho:

1. RF-01: A extensão deve ser capaz de analisar um diagrama de classes contido em um módulo:
   - **Prioridade:** Normal
   - **Pré-condição:** Introduzir no StarUML um diagrama de classes contendo todas as suas respectivas classes e relacionamentos corretamente.
2. RF-02: A extensão deve ser capaz de identificar todas as classes presentes no módulo:
   - **Prioridade:** Normal
   - **Pré-condição:** RF-01.
3. RF-03: A extensão deve ser capaz de identificar todos os relacionamentos de associação bidirecionais presentes no diagrama:
   - **Prioridade:** Normal
   - **Pré-condição:** Não se aplica.
4. RF-04: A extensão deve ser capaz de identificar todos os relacionamentos de associação direta presentes no diagrama:
   - **Prioridade:** Normal
   - **Pré-condição:** Não se aplica.
5. RF-05: A extensão deve ser capaz de identificar todos os relacionamentos de generalização presentes no diagrama:
   - **Prioridade:** Normal
   - **Pré-condição:** Não se aplica.
6. RF-06: A extensão deve ser capaz de identificar todos os relacionamentos de agregação presentes no sistema:

- **Prioridade:** Normal
- **Pré-condição:** Não se aplica.

7. RF-07: A extensão deve ser capaz de identificar todos os relacionamentos de composição presentes no diagrama:
   - **Prioridade:** Normal
   - **Pré-condição:** Não se aplica.
8. RF-08: A extensão deve ser capaz de identificar todas as classes associativas presentes no diagrama:
   - **Prioridade:** Normal
   - **Pré-condição:** Não se aplica.
9. RF-09: A extensão deve ser capaz de calcular automaticamente o FI de todas as classes presentes no diagrama:
   - **Prioridade:** Muito Alta
   - **Pré-condição:** RF-02, RF-03, RF-04, RF-05, RF-06, RF-07 e RF-08.
10. RF-10: A extensão deve ser capaz de calcular automaticamente o FIT de todas as classes presentes no diagrama:
    - **Prioridade:** Alta
    - **Pré-condição:** RF-09.
11. RF-11: A extensão deve ser capaz de identificar a ordem de precedência de implementação dos testes de integração para as classes presentes no diagrama:
    - **Prioridade:** Alta
    - **Pré-condição:** RF-09, RF-10.
12. RF-12: A extensão deve ser capaz de recalcular o FIT de todas as classes após a identificação da precedência das classes em questão, e isso deve se repetir até não restar mais classes a serem analisadas:

- **Prioridade:** Muito Alta
- **Pré-condição:** RF-10, RF-11.

13. RF-13: O sistema deve ser capaz de identificar e informar a criação dos *stubs* para implementar o teste de integração da classe em questão quando necessário:
    - **Prioridade:** Normal
    - **Pré-condição:** RF-11.
14. RF-14: O sistema deve ser capaz de exportar todas as informações calculadas por meio de um documento com extensão XLSX:
    - **Prioridade:** Normal
    - **Pré-condição:** RF-02, RF-03, RF-04, RF-05, RF-06, RF-07, RF-08, RF-09, RF-10, RF-11, RF-12, RF-13.

4.3 Disponibilização da ferramenta

O plugin PickClass será disponibilizado em um repositório aberto no GitHub [4], permitindo assim o acesso à ferramenta. Neste repositório, está presente um arquivo README.md que contém todas as instruções necessárias para a instalação do plugin.

4.4 Detalhamento técnico do desenvolvimento

O desenvolvimento da extensão gira em torno de três funções principais e três tipos de objetos para manipular as informações.

4.4.1 Diagrama de classes da extensão

A extensão desenvolvida apresenta a seguinte Figura 4.1:

[4] https://github.com/YanPaivaAndrade/PickClassStarUml

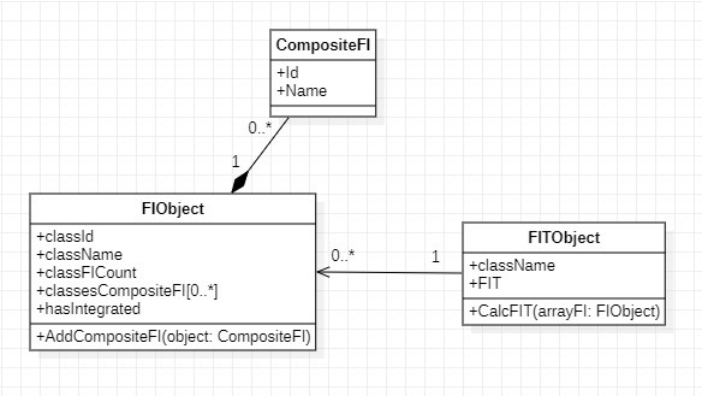

Figura 4.1: Diagrama de classes da extensão

4.4.2 Detalhamento da função generateFI:

A principal função responsável pelos cálculos das heurísticas utilizadas neste trabalho é a *generateFI*. Nesse sentido, é necessário realizar uma análise detalhada e aprofundada de seu conteúdo.

```
//classesForIntegration é um array de classes obtido através do diagrama.
//Com ele é possivel analisar todas as classes e relacionamentos prentesentes no
function generateFI(classesForIntegration) {
  let result = [];
  classesForIntegration.forEach(classe => {
    /*Este array funciona de forma análoga a um hash,
    visto que podemos ter N associações entre as mesmas classes,
    o que deve ser considerado apenas uma única vez em nosso contexto.*/
    let listAssociationsAnalyzed = [];
    classe.ownedElements.forEach(element => {
      let compositeFI = { id: '', name: '' };
      if (element instanceof type.UMLGeneralization) {
        /*
            O StarUML gera o relacionamento de generalização somente do lado
            da(s) classe(s) especializada(s), dito isto, elemento.source
            equivale a classe especializada e
            elemento.target equivale a classe generalizada.
        */
```

```
        if (element.target) {
          let specializedClass = element.source;
          compositeFI.id = specializedClass._id;
          compositeFI.name = specializedClass.name;

          let generalizationClass = element.target;
          let indexGeneralizationClass = _.findIndex(result, x =>
                              x.className === generalizationClass.name);

          if (hasCalculatedFIForClass(indexGeneralizationClass)) {
            result[indexGeneralizationClass].classFICount++;
            result[indexGeneralizationClass].classesCompositeFi.push(compositeFI);
          } else {
            let targetElementAssociation = createFIObject(generalizationClass._id,
            generalizationClass.name, compositeFI);

            result.push(targetElementAssociation);
          }
        }
      } else if (element instanceof type.UMLAssociation) {
        //Primeiro é necessário garantir a unicidade do relacionamento.
        let classOfRelationship = element.end2.reference;
        let indexAssociationsAnalyzed = _.findIndex(listAssociationsAnalyzed, x =>
        x === classOfRelationship.name);

        let unanalyzedAssociation = indexAssociationsAnalyzed === -1;
        if (unanalyzedAssociation) {
          listAssociationsAnalyzed.push(classOfRelationship.name);
          //Quando é uma associação do tipo todo-parte.
          let isTargetPartAssociation = element.end2.aggregation == "shared" ||
```

```
element.end2.aggregation == "composite";

//Quando é uma associação direta ou bidirecional.
let isDirectAssociationOrBidirectionalAssociation = element
                                                    .end2.aggregation == "none"
if (isDirectAssociationOrBidirectionalAssociation) {
  //Primeiro calcula-se a associação direta, classe A -> classe B
  let firstReference = element.end1.reference;
  compositeFI.id = firstReference._id;
  compositeFI.name = firstReference.name;

  let secondReference = element.end2.reference;
  let indexSecondReference = _.findIndex(result, x =>
  x.className === secondReference.name);

  if (hasCalculatedFIForClass(indexSecondReference)) {
    result[indexSecondReference].classFICount++;
    result[indexSecondReference].classesCompositeFi.push(compositeFI);
  } else {
    let secondElementAssociation = createFIObject(secondReference._id,
    secondReference.name, compositeFI);

    result.push(secondElementAssociation);
  }

  let isBidirectionalAssociation = element.end2
                                        .navigable == "unspecified";
  if (isBidirectionalAssociation) {
    /*Caso seja uma associação bidirecional, ainda é necessário tratar
       a associação inversa classe B -> classe A
```

```
        */
        let secondReference = element.end2.reference;
        let isCircularReference = secondReference.name
                                  === firstReference.name;
        if (!isCircularReference) {
          let unspecifiedConposite = { id: secondReference._id,
          name: secondReference.name };

          let indexUnspecified = _.findIndex(result, x =>
          x.className === firstReference.name);

          if (hasCalculatedFIForClass(indexUnspecified)) {
            result[indexUnspecified].classFICount++;
            result[indexUnspecified].classesCompositeFi.push(unspecifiedConpo
          } else {
            let firstElementAssociation = createFIObject(firstReference._id,
            firstReference.name, unspecifiedConposite);

            result.push(firstElementAssociation);
          }
        }
      }
    }
    else if (isTargetPartAssociation) {
      /*
          O StarUML adiciona o relacionamento do lado da classe parte,
          logo a classe todo é a referência do end2 do relacionamento.
      */
      let targetClass = element.end2.reference;
```

```
            compositeFI.id = targetClass._id;
            compositeFI.name = targetClass.name;

            let indexPartClass = _.findIndex(result, x =>
                                              x.className === classe.name);
            if (hasCalculatedFIForClass(indexPartClass)) {
              result[indexPartClass].classFICount++;
              result[indexPartClass].classesCompositeFi.push(compositeFI);
            } else {
              let classFI = createFIObject(classe._id,
                                classe.name, compositeFI);

              result.push(classFI);
            }
          }
        }
      }
      else if (element instanceof type.UMLAssociationClassLink) {
        compositeFI.id = classe._id;
        compositeFI.name = classe.name;

        let firstReference = element.associationSide.end1.reference;
        let secondReference = element.associationSide.end2.reference;

        let indexFirstReference = _.findIndex(result, x =>
                                    x.className === firstReference.name);
        let indexSecondReference = _.findIndex(result, x =>
                                    x.className === secondReference.name);

        if (hasCalculatedFIForClass(indexFirstReference)) {
```

```
          result[indexFirstReference].classFICount++;
          result[indexFirstReference].classesCompositeFi.push(compositeFI);
        } else {
          let firstElementAssociation = createFIObject(firstReference._id,
          firstReference.name, compositeFI);

          result.push(firstElementAssociation);
        }
        /*
        Quando se trata de uma auto associação, deve-se
        desconsiderar a segunda referência, visto que o
        objetivo é analisar a precedência de integração entre as classes.
        */
        let isSelfAssociation = indexSecondReference == indexFirstReference;
        if (!isSelfAssociation) {
          if (hasCalculatedFIForClass(indexSecondReference)) {
            result[indexSecondReference].classFICount++;
            result[indexSecondReference].classesCompositeFi.push(compositeFI);
          } else {
            let secondElementAssociation = createFIObject(secondReference._id, seco
            result.push(secondElementAssociation);
          }
        }
      }
    });
  });

  //Encontrando classes com FI = 0
  classesForIntegration.forEach(classe => {
    let indexClass = _.findIndex(result, x => x.className === classe.name);
```

```
    if (!hasCalculatedFIForClass(indexClass)) {
      let fi = {
        classId: classe._id,
        className: classe.name,
        classFICount: 0,
        classesCompositeFi: [],
        hasIntegrated: false
      };
      result.push(fi);
    }
  });
  return result;
}

function createFIObject(classId, className, compositeFI) {
  let object = {
    classId: classId,
    className: className,
    classFICount: 1,
    classesCompositeFi: [compositeFI],
    hasIntegrated: false
  };
  return object;
}

function hasCalculatedFIForClass(indexClass){
  return indexClass != null && indexClass >= 0;
}
```

4.4.3 Detalhamento das funções getFit e chooseClass:

- **getFit**: Esta função realiza o cálculo do FIT inicial para cada classe. Ela analisa o array de FIObject obtido a partir da função mencionada anteriormente e gera um somatório dos FI's das classes que serão integradas após a classe em análise.

- **chooseClass**: Esta função desempenha a responsabilidade de selecionar a classe a ser integrada em cada iteração. A seleção ocorre escolhendo a classe com o menor FIT entre todas as possibilidades. Em caso de empate, é analisado o maior FI entre as classes empatadas. Se o empate persistir, a seleção é feita em ordem alfabética.

  Além disso, essa função é responsável por atualizar o FIT das classes afetadas pela integração da classe selecionada. Após selecionar a classe 'X' para integração, o FI da classe selecionada é subtraído dos FITs de todas as classes que devem ser integradas antes dela.

4.5 Utilização

Os diagramas de classes apresentados como exemplo de utilização da ferramenta não apresentarão cardinalidade, propriedades ou métodos. Visto que estas informações não são aproveitadas nas heurísticas escolhidas.

Após inserir o diagrama de classes no StarUML, você pode realizar a analise de três formas diferentes, clicando com o botão direito do mouse, acessando a barra superior de ferramentas ou utilizando as teclas de atalho ctrl + w. Conforme Figura 4.2 e Figura 4.3:

Posteriormente, a extensão irá sugerir um local para salvar o arquivo e um nome. É importante observar que o nome sugerido é baseado no que está salvo no StarUML, conforme mostrado à direita na Figura 4.4.

Para o nosso exemplo o resultado gerado é 4.5:

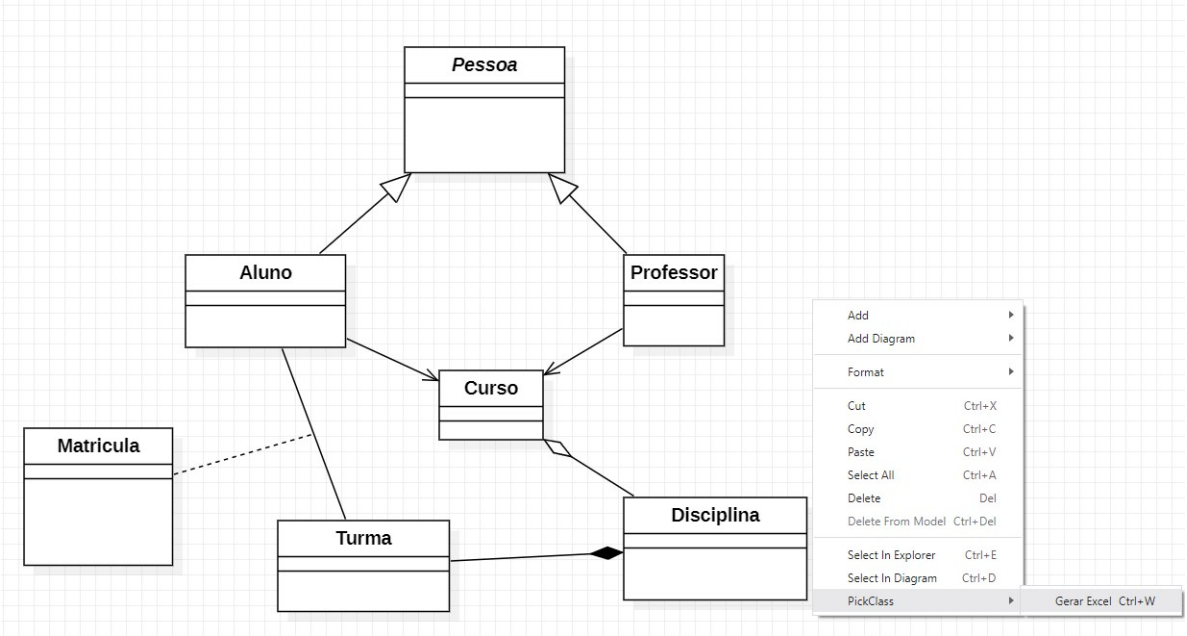

Figura 4.2: Utilização por menu de contexto

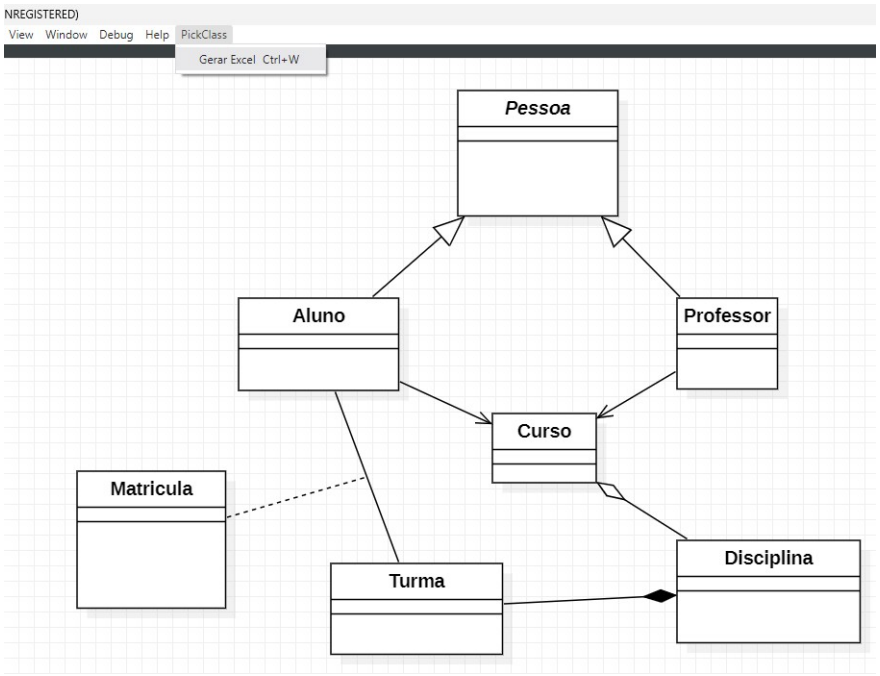

Figura 4.3: Utilização pela barra de ferramentas

4.6 Análise de resultados do FI:

O FI de uma determinada classe, é equivalente a quantidade de classes integradas depois da classe em análise.

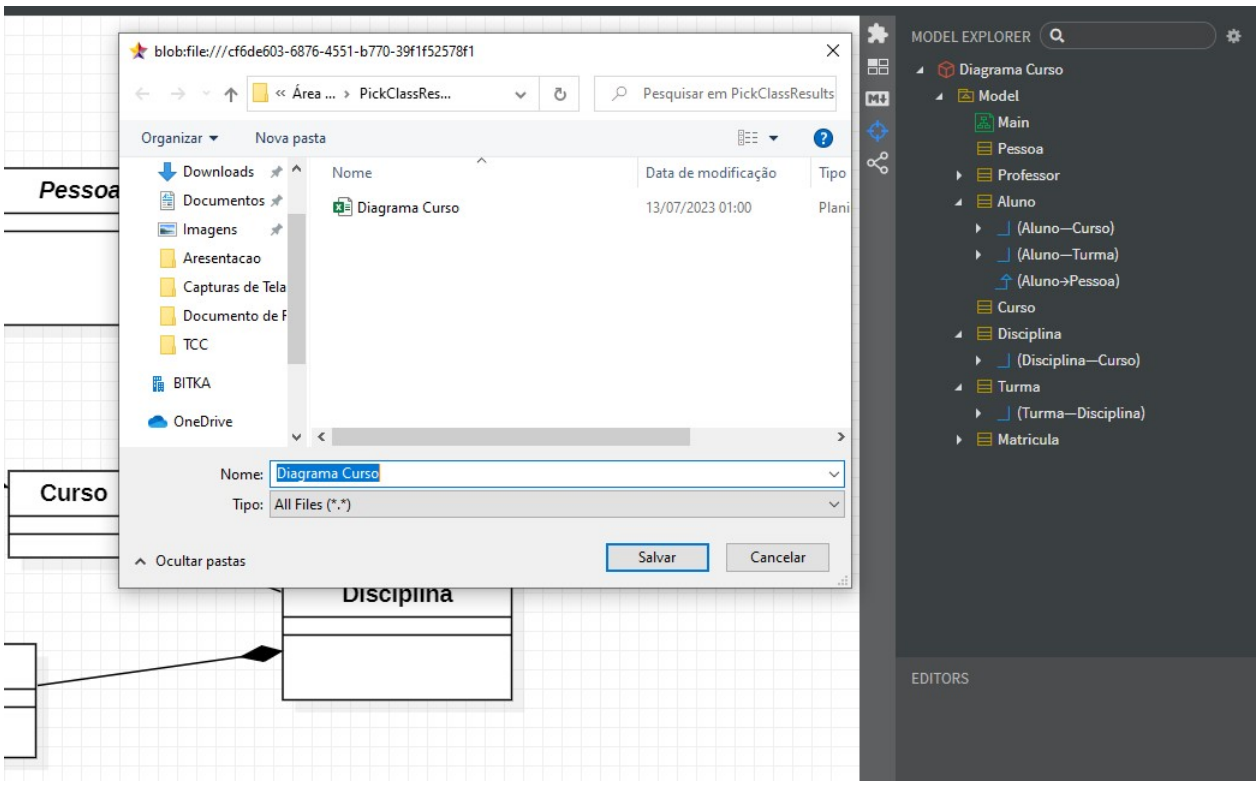

Figura 4.4: salvando arquivos

	FI	FIT1	FIT2	FIT3	FIT4	FIT5	FIT6	FIT7
Aluno	2	7	5	3	3	0	-	-
Curso	2	1	1	-	-	-	-	-
Disciplina	1	3	3	3	3	0	0	-
Matricula	0	5	5	5	5	2	0	0
Pessoa	2	0	-	-	-	-	-	-
Professor	0	4	2	0	-	-	-	-
Turma	3	2	2	2	2	-	-	-

FI = quantidade de classes integradas DEPOIS da classe em questao

FIT = somatório dos Fis das classes integradas ANTES da classe em questao

Ordem de integração

1	Pessoa	
2	Curso	Stub Disciplina
3	Professor	
4	Turma	Stub Aluno
5	Aluno	
6	Disciplina	
7	Matricula	

Figura 4.5: Planilha eletrônica com calculos de FI e FIT

4.6.1 Classe Aluno

Por se tratar de uma associação bidirecional, ao analisar a classe Aluno, ela possui precedência sobre a classe Turma. Além disso, ela também possui precedência sobre a classe Matrícula, uma vez que esta última é uma classe associativa conforme Figura 4.6. *Resultado: 2*

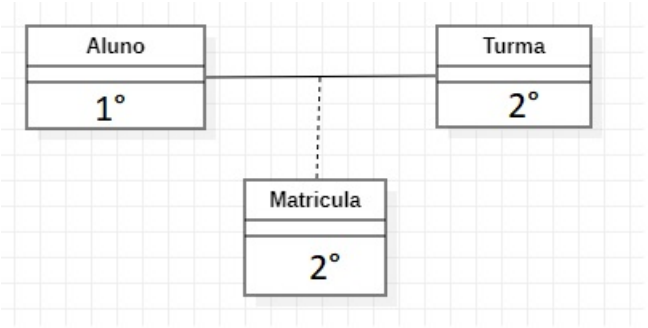

Figura 4.6: Análise de precedência da classe Aluno

4.6.2 Classe Curso

Por se tratar de uma associação direta, a classe Curso possui precedência sobre as classes Aluno e Professor conforme Figura 4.7 *Resultado: 2*

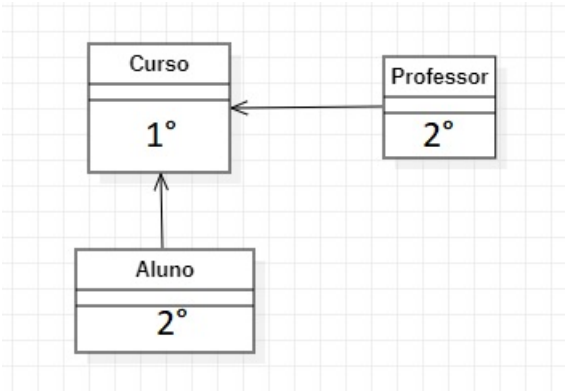

Figura 4.7: Análise de precedência da classe Curso

4.6.3 Classe Disciplina

Por se tratar de um relacionamento todo-parte, a classe Disciplina possui precedência sobre a classe Curso conforme Figura 4.8. *Resultado: 1*

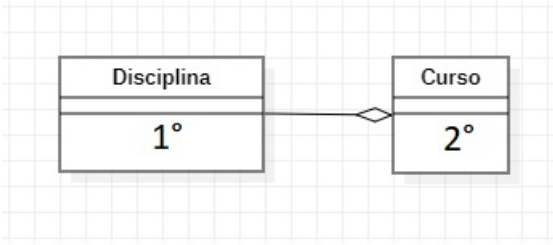

Figura 4.8: Análise de precedência da classe Disciplina

4.6.4 Classe Matricula

A classe Matrícula, por se tratar de uma classe associativa e não possuir relacionamentos, não possui precedência sobre nenhuma outra classe. *Resultado: 0.*

4.6.5 Classe Pessoa

Por se tratar de uma generalização, a classe Pessoa possui precedência sobre as classes Professor e Aluno, uma vez que essas classes são especializações da classe Pessoa conforme Figura 4.9. *Resultado: 2*

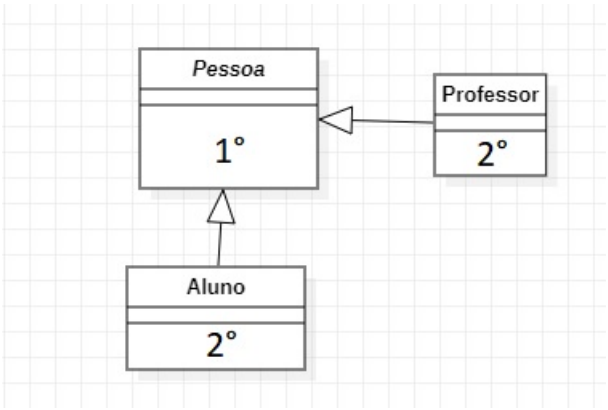

Figura 4.9: Análise de precedência da classe Pessoa

4.6.6 Classe Professor

A classe Professor não possui precedência sobre nenhuma outra classe, visto que seus únicos relacionamentos são de generalização, sendo ela uma especialização de outra classe. Além disso, como a classe Professor recebe atributos de outra classe por meio de associação direta, ela não possui precedência. *Resultado: 0*

4.6.7 Classe Turma

Por se tratar de uma associação bidirecional, ao analisar a classe Turma, ela possui precedência sobre a classe Aluno, ela também possui precedência sobre a classe Matrícula, uma vez que esta última é uma classe associativa, e também possui precedencia sobre a classe Disciplina, visto que é um relacionamento todo-parte e a classe Turma representa a parte deste relacionamento conforme Figura 4.10. *Resultado: 3*

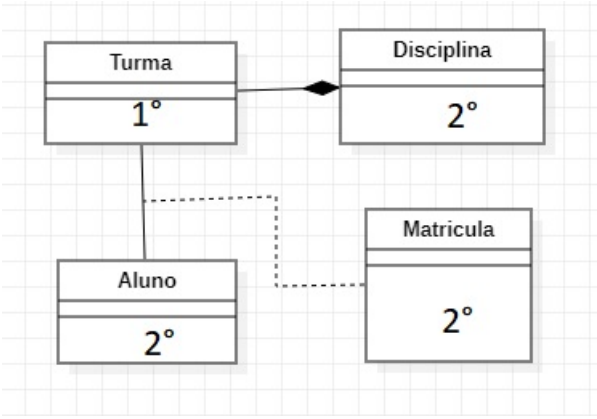

Figura 4.10: Análise de precedência da classe Turma

4.7 Análise de resultados do FIT:

O FIT de uma determinada classe, é equivalente ao somatório de todos os FI's das classes que devem ser integradas antes da classe em análise

Tabela 4.1: Análise do FIT

Classe em análise	**Classes integradas ANTES**	**Resultado**
Aluno	CursoFI = 2; PessoaFI = 2; TurmaFI = 3.	7
Curso	DisciplinaFI = 1	1
Disciplina	TurmaFI = 3	3

Matricula	AlunoFI = 2 TurmaFI = 3	5
Pessoa		**0**
Professor	CursoFI = 2 PessoaFI = 2	4
Turma	AlunoFI = 2	2

4.7.1 Escolha de classe:

Após analisar as tabelas de FI e FIT, é selecionada a classe com menor FIT e maior FI. Em outras palavras, é escolhida a classe que requer o menor número de classes integradas **ANTES** dela e , ao mesmo tempo, permite a implementação de mais classes em um cálculo futuro do FIT.

Seguindo o exemplo apresentado, a classe escolhida é *Pessoa*. Após escolher uma classe, é necessário recalcular o FIT.

Tabela 4.2: Análise do FIT2

Classe em análise	**Classes integradas ANTES**	**Resultado**
Aluno	CursoFI = 2; ~~PessoaFI = 2;~~ TurmaFI = 3.	~~7~~ 5
Curso	*DisciplinaFI = 1*	**1**
Disciplina	TurmaFI = 3	3
Matricula	AlunoFI = 2 TurmaFI = 3	5
~~Pessoa~~		~~0~~

Professor	CursoFI = 2 ~~PessoaFI = 2~~	4 2
Turma	AlunoFI = 2	2

A classe escolhida durante a segunda execução do FIT foi Curso, porém o mesmo não apresenta FIT igual a zero. Isso implica na necessidade de criar um *stub*, pois a classe Disciplina possui precedência em relação ao Curso.

Após a escolha de uma classe, o sistema substitui seu FIT por '-' sinalizando que esta classe acaba de ser integrada. E este processo se repete até restar somente uma classe.

4.8 Analisando auto-associação:

A extensão desenvolvida mostrou-se bastante robusta, sendo capaz de analisar inclusive diagramas com auto-associações, conforme demonstrado na Figura 4.11 e Figura 4.12:

Este segundo exemplo de diagrama de classes, apresenta duas peculiaridades para serem validadas, a primeira delas é a auto- associação, observa-se que a presença da auto associação, não gera um *loop* nos cálculos, conseguindo assim gerar os cálculos de forma precisa. Outro ponto relevante é a presença de mais de uma associação direta entre a classe Frete e Cidade por exemplo, esta situação não afeta os cálculos de FI e FIT, visto que o objetivo é definir a ordem de integração de classes, a quantidade de relacionamentos expressa no diagrama entre as mesmas tabelas devem ser desconsideradas, bastando contabilizar somente um relacionamento.

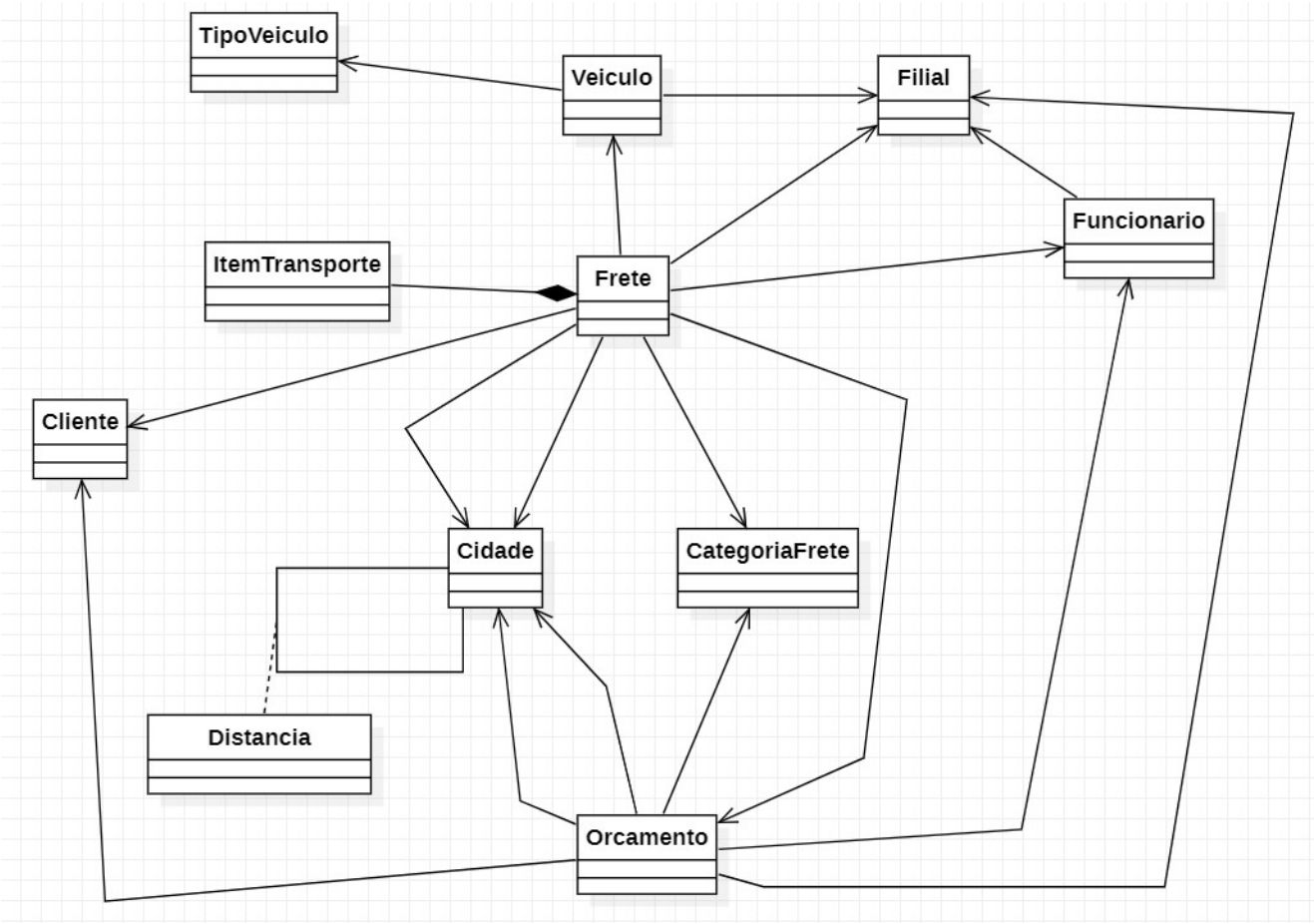

Figura 4.11: auto associação

	FI	FIT1	FIT2	FIT3	FIT4	FIT5	FIT6	FIT7	FIT8	FIT9	FIT10	FIT11
CategoriaFrete	2	0	0	-								
Cidade	4	4	4	4	4	4	4	4	4	-		
Cliente	2	0	0	0	-							
Distancia	0	4	4	4	4	4	4	4	4	0	0	-
Filial	4	0	-									
Frete	0	17	13	11	9	7	6	6	5	1	0	0
Funcionario	2	4	0	0	0	-						
ItemTransporte	1	0	0	0	0	0	-					
Orcamento	1	14	10	8	6	4	4	4	4	0	-	
TipoVeiculo	1	0	0	0	0	0	0	-				
Veiculo	1	5	1	1	1	1	1	0	-			

FI = quantidade de classes integradas DEPOIS da classe em questao

FIT = somatório dos Fis das classes integradas ANTES da classe em questao

Ordenacao	
1	Filial
2	CategoriaFrete
3	Cliente
4	Funcionario
5	ItemTransporte
6	TipoVeiculo
7	Veiculo
8	Cidade
9	Orcamento
10	Distancia
11	Frete

Figura 4.12: resultado auto associação

5 Considerações finais

A extensão desenvolvida abre algumas possibilidades para a equipe de desenvolvimento, pois uma vez que é possível identificar a ordem de integração das classes, com o menor número possível de *stubs*, passa a ser viável desenvolver os testes de integração em conjunto com o desenvolvimento das demais funcionalidades, permitindo assim a identificação precoce de possíveis erros.

Além de reduzir a complexidade de integração das classes, o arquivo xlsx se mostra bastante intuitivo, possibilitando também o estudo das regras de precedência utilizadas neste projeto.

Durante a revisão da literatura, foi evidenciado que a identificação da ordem de precedência das classes era um obstáculo comum em muitos trabalhos. Mostrando-se uma tarefa realizada de forma manual e desgastante, especialmente em sistemas maiores, onde a definição dessa ordem se torna mais complexa.

Também foi realizada uma avaliação exploratória da importância da ferramenta PickClass. Para isso, foi realizada uma pesquisa com o objetivo de coletar informações de graduandos em cursos relacionados ao tema, bem como desenvolvedores de software que estão exercendo a profissão atingindo aproximadamente 460 pessoas e obtendo um total de adesão de 79 respostas, estas respostas são classificadas conforme a Figura 5.1 e Figura 5.2.

Para essa pesquisa, foi disponibilizado o diagrama de classes apresentado na Figura 4.2 e foi solicitada a correta ordem de integração das classes. Apresentando o seguinte resultado, vide Figura 5.3. A diagonal principal destacada, representa a ordem correta de integração das classes. Curiosamente, dentre todas as respostas, somente uma pessoa conseguiu acertar a ordem completa de integração para as classes, evidenciando assim a importância da ferramenta tanto no contexto de aprendizagem quanto no ambiente profissional.

Como sugestão para futuras implementações e trabalhos subsequentes, gostaria de propor alguns tópicos com o objetivo de tornar o software mais robusto e fornecer uma

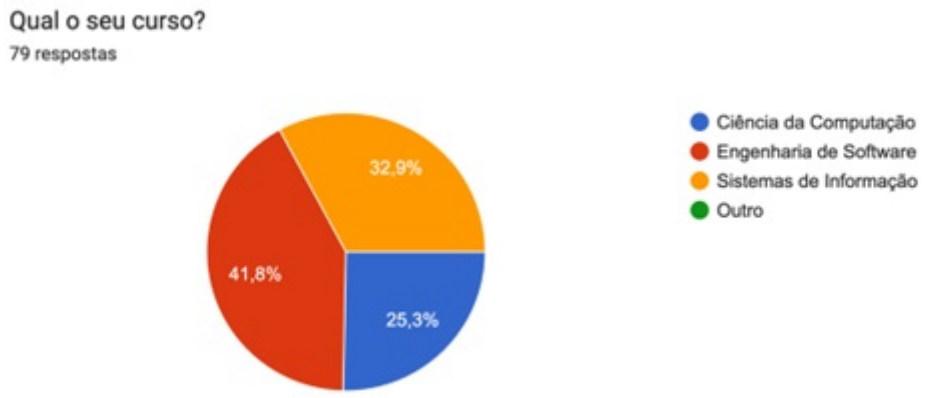

Figura 5.1: Distribuição do publico alvo

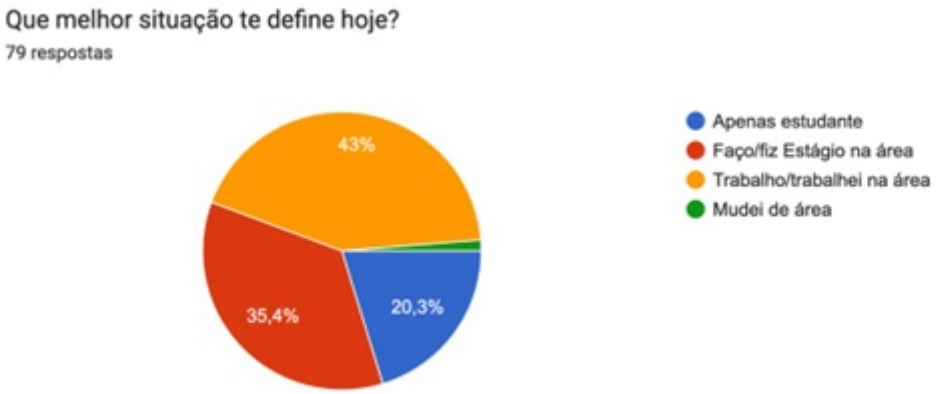

Figura 5.2: Situação Profissional do publico alvo

Classe	1 ª Classe	2 ª Classe	3ª Classe	4 ª Classe	5 ª Classe	6 ª Classe	7 ª Classe
Pessoa	77.2%	12.7%	-	-	5.1%	1.3%	3.8%
Curso	15.2%	17.7%	6.3%	43.0%	5.1%	3.8%	8.9%
Professor	-	16.5%	53.2%	11.4%	5.1%	10.1%	3.8%
Turma	-	1.3%	6.3%	16.5%	19.0%	45.6%	8.9%
Aluno	1.3%	45.6%	24.1%	11.4%	7.6%	6.3%	3.8%
Disciplina	-	3.8%	6.3%	12.7%	45.6%	15.2%	17.7%
Matricula	6.3%	2.5%	3.8%	5.1%	12.7%	17.7%	53.2%

Figura 5.3: Resultado da pesquisa

ampla variedade de ferramentas.

- Exportação de PDF: Nos grandes projetos, é comum incluir casos de uso e casos de teste na documentação. Para adicionar ainda mais valor ao projeto, uma funcionalidade interessante seria a capacidade de exportar em formato PDF as informações

sobre a ordem de integração das classes, bem como a justificativa para a mesma. Isso permitiria compor uma documentação completa e de qualidade, proporcionando uma visão abrangente do projeto e facilitando futuras implementações.

- Exportar JSON: Durante o desenvolvimento, foi observada a possibilidade de futuras integrações do plugin com outras aplicações. Dito isso, torna-se necessário criar uma estrutura mais intuitiva que possibilite uma análise semelhante à realizada na planilha eletrônica resultante deste desenvolvimento. Uma vez que essa estrutura seja definida corretamente, torna-se viável a exportação da mesma no formato JSON, permitindo assim integrações posteriores.

Bibliografia

ANDRADE, W. T. Utilização de técnicas de integração de software para aplicações web no contexto de ferramentas de acessibilidade. 2017.

BOOCH, G.; RUMBAUGH, J.; JACOBSON, I. *The Unified Modeling Language User Guide*. [S.l.]: Addison-Wesley Professional, 2005.

CARDOSO, J. A. *Um método de Testes de Integração para Sistemas Baseados em Componentes*. Tese (Doutorado) — Dissertação de Mestrado, Unicamp, Campinas, São Paulo, 2006.

FOWLER, M. *UML distilled: a brief guide to the standard object modeling language*. [S.l.]: Addison-Wesley Professional, 2004.

FOWLER, M. *Refatoração: Aperfeiçoando o design de códigos existentes*. [S.l.]: Novatec Editora, 2020.

GOUVEIA, C. C. et al. Teste de integração para sistemas baseados em componentes. Universidade Federal de Campina Grande, 2004.

LARMAN, C. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*. [S.l.]: Prentice Hall, 2004.

LIMA, G. *Heurísticas para Identificação da Ordem de Integração de Classes em Testes Aplicados a Software Orientado a Objetos*. Tese (Doutorado) — Tese de M. Sc., COPPE/UFRJ, Rio de Janeiro, Brasil, 2005.

LIMA, G. M. P. S.; TRAVASSOS, G. H. Testes de integração aplicados a software orientado a objetos: Heurísticas para ordenação de classes. In: SBC. *Anais do III Simpósio Brasileiro de Qualidade de Software*. [S.l.], 2004. p. 162–174.

MAXIM, B.; PRESSMAN, R. S. *Engenharia de software: uma abordagem profissional*. [S.l.]: Porto Alegre:[sn], 2021.

MEYER, B. *Object-oriented software construction*. [S.l.]: Prentice hall Englewood Cliffs, 1997. v. 2.

NETO, A. C. D.; LIMA, G.; TRAVASSOS, G. H. Farol: Uma ferramenta de apoio à aplicação de heurísticas de ordenação de classes para teste de integração. *19th SBES–12th Tools Session, Uberlândia, MG, Brazil*, p. 13–18, 2005.

PRESSMAN, R. S. *Software engineering: a practitioner's approach*. [S.l.]: Palgrave macmillan, 2005.

RÉ, R. *Uma contribuição para a minimização do número de stubs no teste de integração de programas orientados a aspectos*. Tese (Doutorado) — Universidade de São Paulo, 2009.

SOMMERVILLE, I. *Engenharia de software*. [S.l.: s.n.], 2018.

SOUSA, H. C. S. et al. Construção automatizada de casos de teste usando engenharia dirigida por modelos. Universidade Federal do Maranhão, 2009.

StarUML documentation. *documentação StarUML*. 2023. Disponível em: ⟨https://docs.staruml.io/⟩. [Acesso em : 24 jun. 2023].

StarUML web site. *site StarUML*. 2023. Disponível em: ⟨https://staruml.io/⟩. [Acesso em : 24 jun. 2023].

VALENTE, M. T. *Engenharia de Software Moderna*. [S.l.]: Independente, 2020.

Apêndice

6 Apêndice: Instalação do plugin

Faça o clone do repositório do plugin no seguinte local:

"C:/Users/USUARIO/AppData/Roaming/StarUML/extensions/user"

Isso permitirá que você baixe todos os arquivos necessários para o funcionamento do plugin.

Após realizar o clone, verifique se a pasta "PickClassStarUml" foi criada corretamente na pasta mencionada acima. E instale suas dependências.

Para utilizar os módulos de geração de arquivos XLSX e lodash do plugin, é necessário instalar o Node. A versão sugerida é a v18.14.0.

Além disso, é necessário instalar o Yarn, pois ele é a ferramenta responsável por gerenciar o versionamento das dependências do plugin. A versão indicada é a v1.22.19

Após instalar o Node e o Yarn, siga os passos abaixo:

1. Abra o terminal e navegue até a pasta raiz do seu projeto.

2. Execute o comando "yarn install". Isso irá baixar e instalar todas as dependências listadas no arquivo "package.json" do seu projeto.

Agora, abra o StarUML e vá até o menu "Tools" (Ferramentas) na barra de navegação superior.

Selecione a opção "Extension Manage..." (Gerenciar Extensões). Uma nova janela será aberta.6.1

6.0.1 Estrutura de pastas

- **PickClassStarUml**: é a pasta raiz do projeto, onde está presente toda a implementação da extensão.

- **PickClassStarUml/keymaps**: é onde está localizado o arquivo .json responsável por definir as teclas de atalho para utilização da extensão.

- **PickClassStarUml/menus**: há dois arquivos relevantes, PickClass.json e PickClass-cm.json.

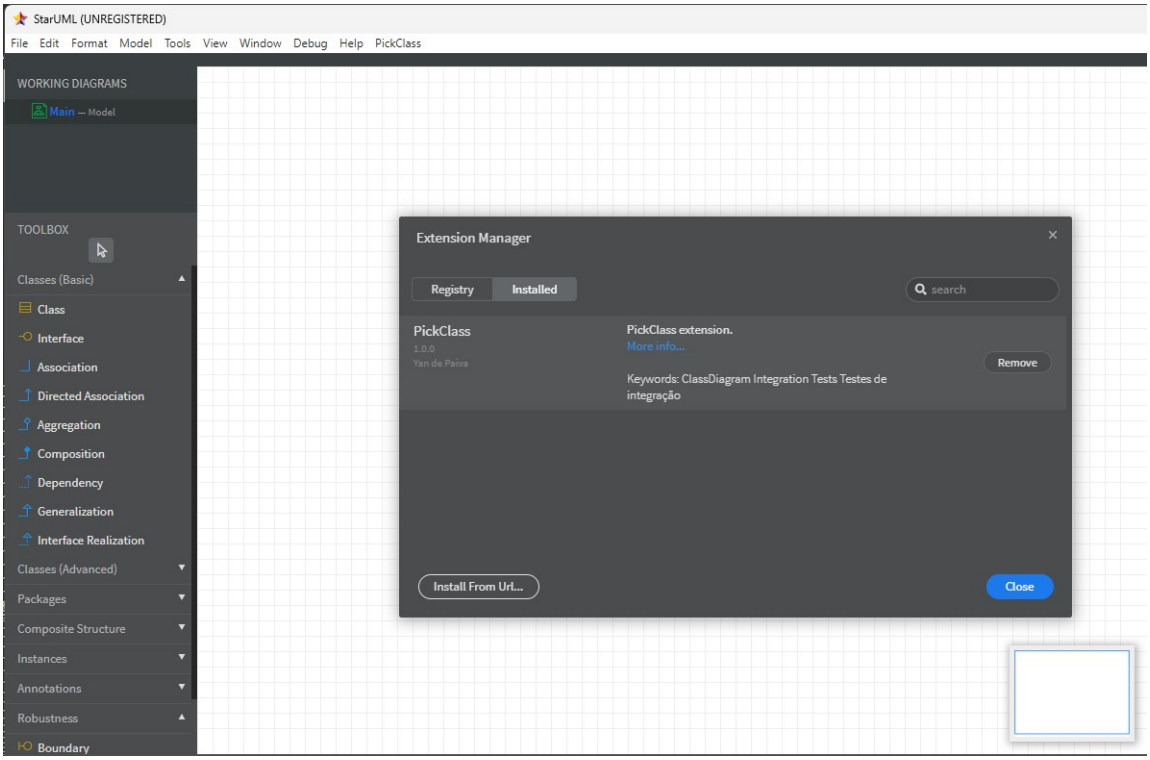

Figura 6.1: Instal plugin

O arquivo PickClass.json é responsável por definir a barra de ferramentas superior do StarUML.

Já o arquivo PickClass-cm.json é responsável pelo menu de contexto que é exibido ao clicar com o botão direito do mouse em um diagrama no StarUML.

6.0.2 Dependências

As dependências listadas abaixo estão definidas no arquivo “package.json” na pasta raiz do plugin, exceto pelo “yarn” e “node”, que foram mencionados durante o processo de instalação.

Para instalar essas dependências, basta executar o comando “yarn install”. Isso fará o download das dependências e gerará a pasta “node_modules” em seu projeto.

É importante ressaltar que a pasta “node_modules” não deve ser incluída no controle de versão do seu projeto. O Yarn, juntamente com o arquivo “package.json”, é responsável por garantir a instalação adequada dessas dependências.

- **Node**: v18.14.0;
- **yarn**: v1.22.19;

- **lodash**: 4.17.21;
- **fs**:0.0.1-security;
- **sheetjs**: 2.0.0;
- **xlsx**: 0.18.5.