UNISINOS

Programa Interdisciplinar de Pós-Graduação em

Computação Aplicada

Mestrado Acadêmico

Vanessa Lara Weber

Detecção de Inconsistências em Modelos UML: Uma Técnica Baseada em Métricas

São Leopoldo, 2017

Vanessa Lara Weber

DETECÇÃO DE INCONSISTÊNCIAS EM MODELOS UML:
UMA TÉCNICA BASEADA EM MÉTRICAS

Dissertação apresentada como requisito para a obtenção do título de Mestre, pelo Programa Interdisciplinar de Pós-Graduação em Computação Aplicada da Universidade do Vale do Rio dos Sinos – UNISINOS

Orientador: Dr. Kleinner Silva Farias de Oliveira

São Leopoldo

2017

W373d Weber, Vanessa Lara.
Detecção de inconsistências em modelos UML : uma técnica baseada em métricas / Vanessa Lara Weber. – 2017.
122 f. : il. ; 30 cm.

Dissertação (mestrado) – Universidade do Vale do Rio dos Sinos, Programa Interdisciplinar de Pós-Graduação em Computação Aplicada, 2017.
"Orientador: Dr. Kleinner Silva Farias de Oliveira."

1. Detecção de inconsistências. 2. Modelos UML multivisão. 3. Métricas de diagramas UML. 4. UML. 5. Catálogo de inconsistências. I. Título.

CDU 004

Dados Internacionais de Catalogação na Publicação (CIP)
(Bibliotecário: Flávio Nunes – CRB 10/1298)

Vanessa Lara Weber

Detecção de Inconsistências em Modelos UML:
Uma Técnica Baseada em Métricas

Dissertação apresentada à Universidade do Vale do Rio dos Sinos – Unisinos, como requisito parcial para obtenção do título de Mestre em Computação Aplicada.

Aprovado em 31 de março de 2017

BANCA EXAMINADORA

Dr. Jorge Luis Victória Barbosa – Universidade do Vale do Rio dos Sinos

Dr. Toacy Cavalcante de Oliveira - Universidade Federal do Rio de Janeiro

Prof. Dr. Kleinner Silva Farias de Oliveira (Orientador)

Visto e permitida a impressão
São Leopoldo,

Prof. Dr. Sandro José Rigo
Coordenador PPG em Computação Aplicada

Dedico esta dissertação ao meu orientador da graduação, Marcelo Ritzel, que me inspirou a iniciar o sonho do mestrado e ao meu orientador de mestrado, Kleinner Farias, que me apoiou, instruiu e dedicou-se a me auxiliar durante todo o andamento do curso.

AGRADECIMENTOS

Agradeço ao meu orientador, Kleinner Farias, que me apresentou os caminhos da pesquisa científica, sempre dedicado em acompanhar as atividades desenvolvidas no curso e motivando o desenvolvimento dos estudos, trabalhos e, sobretudo, artigos.

Agradeço também aos meus pais, Geolar e Márcia Weber, que sempre me apoiaram em toda a minha vida acadêmica e profissional, e, graças a eles, posso concluir este mestrado. E ao meu noivo, Jeferson Berwanger, que sempre me apoiou e incentivou, mesmo nos momentos mais difíceis.

Agradeço ao meu pequeno Grupo de Pesquisa, Lucian Gonçales e Vinicius Bischoff, que foram grandes companheiros de pesquisa e trabalhos, desde as atividades acadêmicas até a produção dos artigos e experimentos.

RESUMO

A *Unified Modeling Language* (UML) é uma linguagem padrão para modelagem de software orientado a objetos, sendo utilizada amplamente tanto na academia quanto na indústria para modelar aspectos estruturais e comportamentais de software. Essa abordagem de modelagem multivisão permite representar informações complementares do software em diagramas distintos (por exemplo, os diagramas de classes e de sequência), permitindo uma melhor representação e compreensão das decisões de projeto tomadas por parte dos desenvolvedores. O problema é que a literatura atual mostra que tais modelos complementares da UML possuem informações estruturais e comportamentais conflitantes, bem como as técnicas atuais têm se mostrado imprecisas para detectar tais inconsistências, incluindo as sintáticas e semânticas mais severas. Além disso, a detecção manual é caracterizada como uma atividade propensa a erros e que exige um alto esforço, principalmente tentando identificar inconsistências semânticas entre os elementos dos diagramas. A presença de inconsistências em modelos UML pode levar a interpretação incorreta de decisões de projetos, bem como reduzir a qualidade do software produzido. Por outro lado, métricas de projeto têm sido definidas e utilizadas nas últimas décadas, sendo apontadas como indicadores efetivos de possíveis problemas em modelos UML. Infelizmente, até o momento nenhum trabalho foi desenvolvido utilizando métricas para detecção de inconsistências em modelos UML. Sendo assim, este trabalho busca: (1) produzir conhecimento empírico sobre os impactos da utilização de modelos UML com inconsistências na qualidade e na produtividade (esforço) do desenvolvimento de software; (2) identificar e elaborar um catálogo de tipos de inconsistências em modelos UML; (3) propor uma técnica baseada em métricas para detecção das inconsistências identificadas no catálogo; e, por fim, (4) implementar uma ferramenta capaz de detectar inconsistências em modelos UML. Para isso, um mapeamento sistemático da literatura foi realizado com 31 estudos primários selecionados através de um cuidadoso processo de filtragem com o objetivo de responder a seis questões de pesquisa motivadoras para o entendimento do estado da arte sobre o tema proposto. Além disso, experimentos controlados foram projetados e executados com uma amostra de 15 pessoas, entre desenvolvedores e analistas de sistemas, para compreender os impactos da utilização da ferramenta proposta na Taxa de Má Interpretação, no Esforço e na Corretude de Interpretação dos modelos, assim como, identificar gravidades para cada tipo de inconsistência. Os resultados obtidos mostram que existem algumas limitações das técnicas de detecção de inconsistência existentes, ocasionando a falta de insights e conhecimentos práticos sobre os efeitos das inconsistências em questões de qualidade de software e do esforço a ser investido para detecção destas inconsistências. Além disso, foi possível identificar três graus de gravidade para determinados tipos de inconsistências e verificar o impacto da utilização da ferramenta, que utiliza as técnicas propostas, na detecção de inconsistências em modelos UML multivisão. Por fim, pode-se concluir que, mesmo não sendo possível comprovar estatisticamente a melhora nos graus de Taxa de Má Interpretação e Esforço, a utilização de uma técnica baseada em métricas possibilita o aumento da Corretude na interpretação de modelos UML miltivisão, melhorando assim a qualidade do software desenvolvido.

Palavras-Chave: Detecção de Inconsistências. Modelos UML Multivisão. Métricas de Diagramas UML. UML. Catálogo de Inconsistências.

ABSTRACT

Unified Modeling Language (UML) is a standard language for object-oriented software modeling, widely used in academia and industry to model structural and behavioral aspects of software. This multi-view modeling approach allows for the representation of complementary software information in distinct diagrams (for example, class and sequence diagrams), allowing for better representation and understanding of project decisions made by developers. The problem is that the current literature shows that such complementary UML models have conflicting structural and behavioral information, as well as current techniques have been inaccurate to detect such inconsistencies, including the more severe syntactic and semantics. In addition, manual detection is characterized as an error-prone activity that requires a lot of effort, mainly trying to identify semantic inconsistencies between the elements of the diagrams. The presence of inconsistencies in UML models can lead to incorrect interpretation of project decisions as well as reduce the quality of the software produced. On the other hand, project metrics have been defined and used in the last decades, being pointed out as effective indicators of possible problems in UML models. Unfortunately, to date no work has been developed using metrics to detect inconsistencies in UML models. Thus, this work seeks to: (1) produce empirical knowledge about the impacts of using UML models with inconsistencies in quality and productivity (effort) of software development; (2) identify and compile a catalog of inconsistency types in UML models; (3) propose a technique based on metrics to detect the inconsistencies identified in the catalog; And, finally, (4) implement a tool capable of detecting inconsistencies in UML models. For this, a systematic mapping of the literature was performed with 31 primary studies selected through a careful filtering process with the objective of answering six motivational research questions for the understanding of the state of the art on the proposed theme. In addition, controlled experiments were designed and executed with a sample of 15 people, between developers and system analysts, to understand the impacts of using the proposed tool on the Poor Interpretation Rate, the Effort and the Correctness of Interpretation of the models, as well as , Identify severities for each type of inconsistency. The results show that there are some limitations of existing inconsistency detection techniques, leading to lack of insight and practical knowledge about the effects of inconsistencies in software quality issues and the effort to be invested to detect these inconsistencies. In addition, it was possible to identify three degrees of severity for certain types of inconsistencies and to verify the impact of the use of the tool, which uses the proposed techniques, in the detection of inconsistencies in multi-view UML models. Finally, it is possible to conclude that, even though it is not possible to statistically prove the improvement in the degrees of Misinterpretation and Effort Rate, the use of a technique based on metrics makes it possible to increase correctness in the interpretation of UML models improving the quality of software.

Keywords: Inconsistency Detection. Multi-view UML models. Metrics UML diagrams. UML. Inconsistencies catalog.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS

min.	minutos

LISTA DE SIGLAS

A	Aspecto
CaSD	Classe abstrata instanciada no SD
CD	*Class Diagram* - Diagrama de Classes
Cm	Várias definições de Classes com mesmo nome
CnCD	Objeto sem Classe no CD
CnSD	Classe não instanciada no SD
Cor	Corretude na detecção de inconsistências
EcM	Mensagem sem Método
ED	Diagrama de Estado
ED	Mensagem na direção Errada
EfCt	Consolidação do esforço para detecção de inconsistências
Effort	Esforço para detecção de inconsistências
EI	Esforço para Interpretação
EnN	Mensagem sem Nome
GQM	*Goal, Question, Metric*
H	Hipótese
IBM	*International Business Machines*
ID	*Identity* - Identidade
OID	*Object Identity*
Om	Várias definições de Objetos com mesmo nome
PIPCA	Programa Interdisciplinar de Pós-Graduação em Computação Aplicada
Q	Questão
QP	Questão de pesquisa
RSA	*Rational Software Architect*
S	*Study* - Estudo
SD	*Sequence Diagram* - Diagrama de Sequência
SDMetrics	*Software Design Metrics*
TMI	Taxa de Má Interpretação
TRC	Taxa de Respostas Corretas
UC	*Use Case* - Caso de Uso
UML	*Unified Modeling Language*
UNISINOS	Universidade do Vale do Rio dos Sinos
XML	*eXtensible Markup Language*

SUMÁRIO

1 INTRODUÇÃO

A *Unified Modeling Language* (UML) tornou-se de fato um padrão para modelagem de software orientado a objetos (FERNANDÉZ-SÁEZA, 2013), sendo amplamente utilizada tanto na academia quando na indústria. Vários estudos realizados nos últimos anos demonstram os benefícios do uso da UML, bem como sua ampla aceitação. Exemplos destes estudos seriam (CHAUDRON, 2012), (FERNANDÉZ-SÁEZA, 2013) e (DZIDEK, 2008). Em (CHAUDRON, 2012) o autor destaca que a modelagem tornou-se uma atividade comum em projetos de desenvolvimento de software, porém, não há muitas evidências sobre a eficácia do uso de UML, fomentando assim, os debates sobre os pontos positivos e negativos modelagem de software na prática.

Em (FERNANDÉZ-SÁEZA, 2013), os autores destacam que o uso da UML permite melhorar a comunicação entre membros da equipe, bem como potencializar a compreensão de decisões de projetos por partes dos desenvolvedores ao longo do processo de desenvolvimento e manutenção. Em (DZIDEK, 2008), os autores reportam um experimento controlado, no qual puderam identificar que a UML ajudou a produzir código de melhor qualidade quando os desenvolvedores ainda não estavam familiarizados com o sistema, possibilitou identificar rapidamente partes relevantes do sistema que precisam ser compreendidas para implementação de mudanças e transmitiu informações difíceis de recuperar a partir do código, ajudando a evitar a deterioração da arquitetura do sistema.

Esses benefícios são, em parte, devido à capacidade da UML de modelar aspectos estruturais e comportamentais de software, daqui para frente tratada como *modelagem multivisão*. Essa abordagem de *modelagem multivisão* é viabilizada através da definição de 7 diagramas para representar aspectos estruturais e 8 diagramas para descrever aspectos comportamentais do sistema de software em desenvolvimento. A *modelagem multivisão* permite representar informações complementares do software em diagramas distintos (por exemplo, os diagramas de classes e de sequência), ao mesmo tempo também permite que tais diagramas sejam alterados em paralelo por diferentes times de desenvolvimento. Essa flexibilidade de representação e alteração permite uma melhor descrição das decisões de projeto tomadas por parte dos desenvolvedores, bem como, agiliza a elaboração ou manutenção dos diagramas de forma concorrente.

Porém, a manipulação de diagramas complementares em paralelo pode dar origem a informações conflitantes entre os diagramas. Em (LANGE, 2004), reporta que tais inconsistências em diagramas UML é algo comum em modelos da indústria, pois, ao oferecer nove tipos de diagramas, a UML permite um grande grau de liberdade que pode ocasionar conflitos entre diagramas, amplificando os problemas de falta de comunicação em projetos de desenvolvimento de software. Em (LANGE, 2006), Lange e Chaudron realizaram realizam um trabalho inicial para entender os efeitos destas inconsistências, elencando uma série de tipos de inconsistências que foram avaliadas.

Inconsistências em modelos UML (FARIAS, 2013), considerando a tendência de que Diagramas de Classe e Diagramas de Sequência têm de tornarem-se conflitantes, podem variar desde impactos sobre problemas estruturais, envolvendo a aplicação de padrões de projetos (KERIEVSKI, 2008) (LARMAN, 2007), até inconsistências sintáticas e semânticas, que acabam dificultando a interpretação e compreensão destes diagramas. Sendo assim, a presença de inconsistências acaba prejudicando a modularidade, a manutenção e a evolução dos modelos de software (FARIAS, 2013). Os impactos causados pelos efeitos da utilização

de diagramas com inconsistências variam desde problemas de manutenção, causados pela degradação arquitetural do software, até problemas de má interpretação oriundos da ambiguidade dos modelos (FARIAS, 2013).

A literatura mostra que as técnicas de detecção atuais têm se mostrado imprecisas para detectar inconsistências mais severas, incluindo as sintáticas e as semânticas. A maior parte das técnicas disponíveis na literatura realizam a detecção de inconsistências sintáticas (REDER, 2012) instantaneamente, tanto durante a própria criação dos diagramas, como na escrita do código fonte. No entanto, isso não ocorre de forma completa nos ambientes de modelagem, caracterizando abordagens parciais para detecção de inconsistências. As abordagens existentes para detecção de inconsistências em modelagens UML, que envolvem o desenvolvimento de aplicações que ainda não foram amplamente adotadas no meio acadêmico ou a indústria, podem ser classificadas entre completas e parciais. As abordagens completas visam definir uma semântica totalmente formal para a notação UML completa (READER, 2013). Sendo assim, desenvolve um significado bem definido para esse subconjunto com o objetivo de identificar inconsistências automaticamente neste projeto parcial (REDER, 2013).

A presença de inconsistências em modelos UML pode aumentar de esforço para a realização da interpretação dos diagramas por parte dos desenvolvedores, bem como acarretar em grandes consequências à qualidade do software produzido ao final do processo, alterando, inclusive, requisitos funcionais. Os custos altos para a criação e manutenção dos diagramas UML acabam não se justificando na maior parte das indústrias de software, pois, a qualidade dos diagramas não é avaliada e validada, ocasionando o aumento de custos e retrabalho, além de maior esforço para realização das atividades que envolvem a interpretação dos diagramas.

Apesar de importante para a qualidade do software desenvolvido a partir de modelos UML, o esforço a ser investido para identificar e remover inconsistências deve ser o mínimo possível, pois, se resultar em um aumento de esforço, os benefícios esperados, como o aumento de produtividade da equipe e qualidade no software desenvolvido, não serão compensados e a etapa de avaliação e validação dos diagramas UML pode ser dispensada (FARIAS, 2012).

1.1 Problemática

A falta de conhecimento sobre o impacto do uso de modelos UML com inconsistência na qualidade e na produtividade torna-se o principal motivador desta pesquisa. Busca-se compreender quais técnicas são utilizadas para detectar inconsistências de tal forma que o impacto das inconsistências seja minimizado. Além disso, é importante ressaltar que a detecção de inconsistências em modelos UML é um assunto bastante novo, que vem em consequência ao aumento da utilização dos diagramas UML. Com a identificação de poucos estudos sobre a detecção de inconsistências, pode-se afirmar que não há uma metodologia consolidada e reconhecida pela indústria para minimizar os impactos do uso de diagramas com inconsistências nem para detecção das inconsistências.

Além disso, os próprios impactos do uso de diagramas com inconsistências são de pouco estudo, não havendo muitos dados sobre o quanto cada tipo de inconsistência impacta de maneira diferente em um projeto de software. Ainda, a detecção das inconsistências realizada quando o projeto UML está concluído também acaba inviabilizando a execução de procedimentos de verificação e validação, conforme (REDER, 2012).

Igualmente, o momento de detecção das inconsistências também deve ser levado em consideração, uma vez que os maiores impactos se dão no momento de interpretação do diagrama UML pelo desenvolvedor e nos momentos de manutenção do código quando o projeto já tiver sido concluído. A grande maioria dos estudos apresenta a detecção de inconsistências apenas na passagem da fase de análise para o desenvolvimento, porém, a atualização dos diagramas, quando realizadas tarefas de manutenção não pouco tratadas.

Estas considerações levaram o estudo a concentrar-se na importância de investigar os tipos de inconsistências que são comumente identificadas através dos métodos encontrados, buscando definir novas técnicas capazes de detectar inconsistências a fim de minimizar os impactos de inconsistências de modelos UML na qualidade e produtividade do desenvolvimento de software.

1.2 Objetivos e questão de pesquisa

Este trabalho tem como objetivo geral propor uma técnica eficaz para a detecção de inconsistências em modelos UML que facilitem a detecção destas inconsistências, diminuindo o esforço necessário por parte dos desenvolvedores e analistas, minimizando assim, os impactos destas inconsistências na qualidade do software. Para apoiar este objetivo, os seguintes objetivos específicos foram elencados:

- **Produzir conhecimento empírico sobre os impactos da utilização de modelos UML com inconsistências na qualidade e na produtividade (esforço) do desenvolvimento de software.** Este objetivo motivou a pesquisa sobre a identificação dos diferentes tipos de inconsistências e seus impactos em diagramas UML, bem como, identificar como os diferentes tipos de inconsistências afetam no esforço necessário para a interpretação dos modelos multivisão, buscando entender e caracterizar como estes fatores afetam o esforço e a qualidade do software.
- **Identificar e evoluir técnicas para detecção de inconsistências sintáticas e semânticas em modelos UML multivisão.** Este objetivo procura identificar as diferentes técnicas de detecção já documentadas na literatura atual e evolui-las de forma a propor novas técnicas de detecção de inconsistências que atendam as deficiências da área. Deficiências estas, relativas à detecção de inconsistências sintáticas e semânticas em modelos multivisão, através de métricas. Ao se desenvolver essas técnicas, espera-se minimizar os impactos de inconsistências de modelos UML na qualidade e produtividade do desenvolvimento de software.
- **Implementar técnicas capazes de detectar inconsistências em modelos UML, considerando inconsistências semânticas e sintáticas, a partir de métricas e padrões de projetos de software.** Este objetivo visa validar as técnicas estudas e criadas neste trabalho, consolidando uma metodologia para a detecção de inconsistências em modelos UML através de métricas, que possa ser utilizada para minimizar os impactos de inconsistências de modelos UML na qualidade e produtividade do desenvolvimento de software, reduzindo o esforço e falhas na interpretação dos modelos.

Há ainda uma importante lacuna no conhecimento sobre o uso de UML e seu relacionamento com o impacto de modelos UML de má qualidade na qualidade e

produtividade do desenvolvimento de software (CHAUDRON, 2012). As consequências na qualidade e na produtividade do desenvolvimento de software devido a diagramas com inconsistências tornou-se, então, o principal motivador para a investigação dos métodos de verificação dos diagramas UML atuais, o que gerou a seguinte questão de pesquisa tratada deste trabalho:

Questão de Pesquisa: Como reduzir o impacto de inconsistências de modelos UML na qualidade e produtividade do desenvolvimento de software?

1.3 Método de pesquisa

Esta pesquisa busca definir um método de investigação que viabilize: (1) a investigação empírica do esforço que os desenvolvedores e analistas investem para compreender e detectar inconsistências em modelos UML; (2) o estudo dos fatores que envolvem a detecção de inconsistências em modelos UML e seus impactos na qualidade e produtividade do software; e (3) a construção de conhecimento necessário para a definição de técnicas capazes de facilitar a detecção de inconsistências em modelos UML multivisão. Para isto, este trabalho se utiliza das seguintes metodologias de pesquisa:

- Realização de uma ampla revisão da literatura empírica sobre detecção de inconsistências em diagramas UML, com o objetivo de identificar o estado-da-arte sobre este tema, através de publicações em fóruns, eventos, revistas e periódicos. Ainda, a compreensão das possíveis falhas no uso de UML e impactos na produtividade e na qualidade do código produzido, através da identificação de lacunas nas pesquisas atuais.
- Aplicação de estudos experimentais controlados para análise qualitativa e quantitativa da prática de detecção de inconsistências em modelos UML que buscam mensurar o esforço e capacidade de interpretação dos diagramas para detecção de inconsistências por parte dos desenvolvedores e analistas, com e sem uma técnica definida.
- Implementação de uma ferramenta com aplicação da técnica de detecção de inconsistências utilizando métricas para modelos UML multivisão.

1.4 Contribuições

Este trabalho busca trazer contribuições pertinentes a detecção de inconsistências em modelos UML multivisão, a partir da problemática, objetivos e questão de pesquisa apresentados nas Seções anteriores. Foram apresentadas as principais deficiências na literatura e no mercado sobre a detecção de inconsistências, então busca-se determinar um conjunto de contribuições que esta pesquisa visa produzir ao se concluir os objetivos propostos, tais como:

- **Mapeamento sistemático do estado-da-arte.** Através de um estudo de mapeamento sistemático (*Mapping Study*), analisando uma gama de trabalhos relacionados, busca-se identificar o estado-da-arte de detecção de inconsistências em modelos UML. Este estudo tem o objetivo de reunir, em um único trabalho, as

características das diferentes técnicas de detecção de inconsistências em modelos UML. A partir disto, pode-se construir uma base de conhecimento referente às técnicas para detecção de inconsistências mais comuns na literatura, e, dessa forma, se tornar um referencial na área de pesquisa desse conteúdo.

- **Conhecimento empírico sobre detecção de inconsistências em modelos UML.** Este trabalho busca construir um conhecimento empírico sobre detecção de inconsistências em modelos UML, não apenas para identificar o esforço que modelos de má qualidade causam no desenvolvimento de software, mas também identificar os impactos de tais modelos na qualidade do software desenvolvido, conforme os diferentes tipos de inconsistências possíveis em diagramas, considerando também o aspecto de multivisão.
- **Técnicas para detecção de inconsistências em modelos UML.** Com base no conhecimento construído a partir do mapeamento sistemático e dos estudos empíricos sobre detecção de inconsistências *em modelos UML multivisão*, busca-se a elaboração de uma técnica capaz de suprir as demandas levantadas a partir da análise da literatura atual, envolvendo também os aspectos de inconsistências sintáticas, semânticas e estruturais em modelos UML. Serão considerados também os impactos de cada um dos tipos de inconsistências possíveis nos diagramas de software, de forma a priorizar as atividades de detecção, permitindo que estas técnicas sejam eficazes e sirvam de referencial para o desenvolvimento de novas metodologias para detecção de inconsistências e sua viabilidade quanto a aplicação na indústria.
- **Desenvolvimento de uma aplicação utilizando as técnicas propostas para detecção de inconsistências em modelos UML multivisão.** A partir da elaboração de novas técnicas para detecção de inconsistências em modelos UML multivisão, busca-se desenvolver uma metodologia que, utilizando métricas para detecção de inconsistências nestes diagramas, seja capaz de facilitar o processo de detecção de inconsistências por parte dos analistas e desenvolvedores de software, de forma completa, envolvendo inconsistências sintáticas, semânticas e estruturais, minimizando assim, os impactos da utilização destes modelos na qualidade e na produtividade do desenvolvimento de software.

1.5 Organização

A organização deste trabalho é apresentada através da elaboração de seis capítulos. O Capítulo 2 apresenta os conceitos fundamentais para a detecção de inconsistências em modelos UML, tais como, os próprios modelos UML sobre a percepção multivisão, os tipos de inconsistências encontrados em modelos UML, e uma ferramenta que possibilita analisar as métricas de Diagramas UML e criar novas métricas conforme a necessidade do usuário. O Capítulo 3 apresenta o estado-da-arte sobre detecção de inconsistências através de uma comparação entre as diferentes técnicas utilizadas atualmente, identificando lacunas para possíveis oportunidades de pesquisa. O Capítulo 4 apresenta um Catálogo de Inconsistências, descreve a técnica para detecção de inconsistências em modelos UML através da utilização de métricas e apresentando a ferramenta implementada, bem como sua modelagem e funcionalidades. O Capítulo 5 apresenta como a avaliação da técnica proposta foi realizada e como foram identificadas as gravidades dos diferentes tipos de inconsistências. Por fim, o

Capitulo 6 apresenta as conclusões deste estudo, assim como, os resultados das avaliações realizadas, as limitações encontradas durante a execução do estudo e sugestões para trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Para a melhor compreensão da proposta desta dissertação, alguns conceitos e fundamentos devem ser desenvolvidos. Sendo assim, neste capítulo serão apresentados alguns conceitos sobre a elaboração de modelos UML, utilizando-se de modelos multivisão, através da aplicação e desenvolvimento de diagramas de Classe e Sequência, assim como, a notação de cada. Será apresentado também o conceito utilizado para o domínio de inconsistências em diagramas UML, assim como, o que representam para modelos multivisão. Por fim, apresenta o conceito base da ferramenta *SD Metrics*, que será utilizada como apoio à aplicação desenvolvida neste trabalho.

2.1 Análise e Projeto de Software

Atualmente é cada vez mais comum o uso do conceito de processos que envolvem a análise e projetos de software que tenham já em sua concepção a orientação a objetos (BEZERRA, 2007). O significado de análise é a identificação das necessidades da aplicação, a realização de um levantamento das alternativas de solução para a resolução de um problema. Quando a análise é aplicada em um projeto, pode-se apontar a definição, especificação, modelagem e construção dos artefatos que comporão o sistema, assim como a definição dos hardwares que serão utilizados, outros softwares a serem integrados e a arquitetura e modelagem do software, propriamente dito, a ser desenvolvido (KERIEVSKI, 2008).

A análise e desenvolvimento do projeto de software têm o objetivo de testar, avaliar e validar a estrutura arquitetural do software, assim como definir sua viabilidade técnica, antes do início do desenvolvimento através de sua codificação (LARMAN, 2007). Tudo isso é necessário para que se possa reduzir a complexidade do desenvolvimento, o desperdício de esforço e custo, o retrabalho e aumentar a previsibilidade do software concluído, com o objetivo de atender a todos os requisitos especificados.

Para isso, os modelos UML podem ser utilizados. Um modelo é uma abstração da realidade. Eles auxiliam na visualização do sistema como ele é ou como se espera que ele seja. Eles permitem especificar a estrutura ou o comportamento de um sistema, proporcionando, ao final, um guia para a construção e desenvolvimento do sistema, documentando as decisões tomadas para o desenvolvimento (BOOCH et al., 2008). As principais características que definem se o desenvolvimento de software se refere a um projeto com orientação a objetos é a sua modularidade e ser incremental, com direcionamento e possibilidade de reutilização de classes e métodos.

Para a modelagem destes sistemas, é comum a utilização de mais de um tipo de Diagrama, resultado em uma modelagem multivisão. Estudos apontam o uso deste tipo de modelagem na indústria, porém, a liberdade de utilização de diferentes diagramas, pode resultar em problemas de inconsistências entre os diagramas, como apontado em (FARIAS, 2013) e (FARIAS 2010).

2.1.1 Modelagem UML

A *Unified Modeling Language* (UML) é uma linguagem de modelagem de software que permite visualizar, especificar, construir e documentar artefatos de um sistema complexo de software (BOOCH et al., 2008). Ela utiliza-se de uma linguagem gráfica que possibilita a análise, especificação e construção de softwares utilizando-se de orientação a objetos.

A UML dispõe de diferentes tipos de artefatos e diagramas que possibilitam a análise e projeto do software, e estes diagramas permitem visões diferentes do software a ser desenvolvido. Por exemplo: um Diagrama de Casos de Uso permite que sejam especificados o que fazem e o que desejam os usuários do sistema e define quais objetos são necessários para cada caso de uso; um Diagrama de Classes especifica os objetos no mundo real que interagem com o sistema e suas associações; o Diagrama de Interação é utilizado para definir como colaboram entre si objetos dentro de um caso de uso; um Diagrama de Estados apresenta como serão implementados os controles de tempo real; e Diagramas de Pacotes, de Componentes e de Implantação, definem como será construído e codificado o sistema e seu banco de dados.

Através de uma visão de casos de uso, abrangem-se os casos de uso que descrevem o comportamento do sistema conforme usuários finais, analistas, até a equipe de testes, porém, não especifica como se dará a organização do sistema em si (BEZERRA, 2007). Para obter-se esta visão, utilizam-se os Diagramas de Casos de Uso, de Interação, de Estados e de Atividades. Já, na visão de projeto, apresentam-se as classes, interfaces e colaborações que formarão o vocabulário do sistema. Serve como suporte aos requisitos funcionais do sistema, ou seja, as funcionalidades a serem desenvolvidas (BOOCH et al., 2008). Para documentar esta visão, utilizam-se os Diagramas de Classes, de Objetos, de Interações, de Estados e de Atividades.

Para chegar-se a uma visão do processo do software que será desenvolvido, utilizam-se os mesmos diagramas da visão de projeto, pois, eles apresentam também o fluxo de controle entre as várias partes, a concorrência e a sincronização das mesmas, no que diz respeito à escalabilidade e desempenho do sistema (BEZERRA, 2007). A visão de implementação abrange os componentes e artefatos utilizados para construção e entrega do sistema, envolvendo o gerenciamento de configuração, ou seja, o versionamento dos artefatos necessários para a implementação do software. Para isto, utilizam-se os Diagramas de Componentes, de Interações, de Estados e de Atividades.

Por fim, a visão de implantação é a visão que abrange os nós da topologia de hardware sobre a qual o sistema roda, direcionando instalação dos componentes físicos em que o software atuará. Para obter-se esta visão utilizam-se os Diagramas de Implantação, de Interações, de Estados e de Atividades.

2.1.2 Principais conceitos

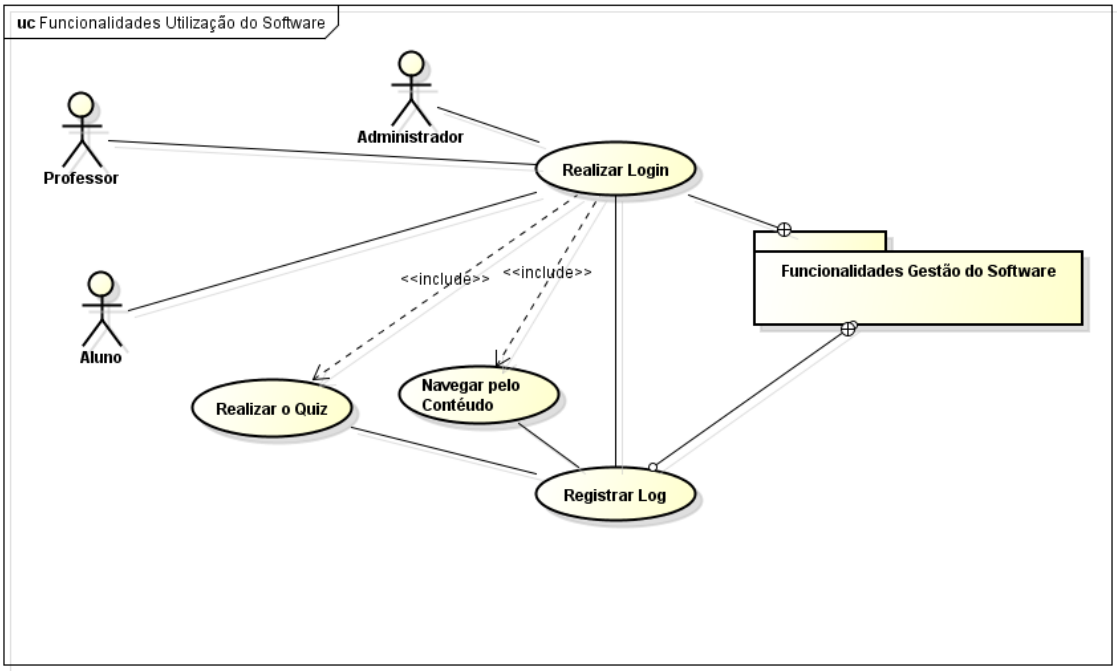
Com base nas diferentes visões para a análise e projeto do software, alguns conceitos são primordiais para a elaboração e manutenção dos diferentes diagramas que a UML disponibiliza. Abaixo seguem as definições dos principais conceitos utilizados quando se fala sobre Diagramas de Classe e de Sequência, que fazem parte do objeto de estudo desta dissertação (BEZERRA, 2007).

- **Classe**: É a representação de uma "coisa" do mundo real que possui características (atributos) e comportamentos (operações) que devem estar presentes na aplicação. É um agrupamento de objetos com características e comportamentos comuns.
- **Atributo**: Representa propriedade nomeada ou elemento de dados de uma classe. Pode referir-se à (1) visibilidade, que pode ser pública, quando é acessível por qualquer outra classe ou objeto; privada, quando é acessível somente pela própria classe; protegida, quando é acessível somente pela própria classe e pelas suas subclasses; ou de pacote, quando é acessível pelas classes do mesmo pacote; (2) ao tipo do dado, por exemplo, inteiro, *string*, *array*, *char*, *boolean*, entre outros; e (3) ao valor inicial, quando indicado o valor com o qual o atributo é iniciado.
- **Operação**: É uma função ou procedimento que pode ser aplicado a/ou por objetos de uma classe. Uma "mesma" operação pode ocorrer em diferentes classes, com comportamentos distintos, caracterizando polimorfismo.
- **Método**: É a implementação de uma operação para uma determinada classe. Ele é dado através da assinatura de uma operação é formada por um Nome, a Visibilidade, os Argumentos, e o Tipo de Retorno esperado.
- **Objeto**: Representa a ocorrência (instância) particular de uma classe, com valores para suas características (atributos) e que executam os comportamentos (métodos). Todo objeto possui uma identidade (*Object* ID ou OID).
- **Herança**: É um relacionamento entre uma classe (superclasse) e uma ou mais variações da classe (subclasses). Uma subclasse herda todos os atributos e operações da superclasse. Uma subclasse pode possuir atributos e operações próprias, não existentes na superclasse.
- **Polimorfismo**: Princípio pelo qual subclasses podem implementar uma mesma operação herdada da superclasse de forma diferente, produzindo resultados diferentes, mantendo a mesma assinatura para a operação.
- **Encapsulamento**: É a técnica para separar aspectos externos dos internos na implementação de um objeto.

2.1.3 Diagramas de Casos de Uso

Um Diagrama de Casos de Uso tem o objetivo de modelar as interações entre os atores e as funcionalidades do sistema, caracterizando assim a documentação dos Requisitos Funcionais do software. Pode-se observar, através da Figura 1, a implementação de um Diagrama de Casos de Uso no qual os atores esperam realizar operações de criação e manutenção de usuários e textos, envolvendo um banco de dados para armazenamento das informações (BOOCH et al., 2008).

Figura 1: Exemplo de um Diagrama de Casos de Uso



Fonte: Elaborado pela autora.

Pode-se observar uma série de elementos no diagrama anterior. Estes elementos são descritos conforme apresentados na Tabela 1, na qual podem ser identificados seu Nome, Descrição e Elemento Gráfico que os representam:

Tabela 1: Elementos do Diagrama de Casos de Uso

Nome do Elemento	Descrição do Elemento	Elemento Gráfico
Ator	Algo ou alguém que interage com o sistema	
Caso de Uso	Funcionalidade: Algo que o sistema deve fazer. É a especificação de um conjunto de ações executadas tipicamente por um sistema que entrega um resultado observável que é de valor para um ou mais atores ou outros interessados no sistema (definição da OMG). Foco no “que”, e não no “como”	

Fonte: Elaborado pela autora.

Para cada Caso de Uso, há um padrão de estrutura a ser seguindo, conforme apresentado na Tabela 2, na qual deve existir um nome único e auto explicativo, a definição dos ator(es) principal(is) e secundário(s), uma breve descrição, auto contida da funcionalidade, a alternativa referente ao caminho feliz (*happy path*), a qual define o resultado da aplicação da funcionalidade e os possíveis caminhos alternativos, que referem-se aos demais resultados e comportamentos caso o caminho feliz não se concretize (BOOCH et al., 2008).

Tabela 2: Estrutura de um Caso de Uso

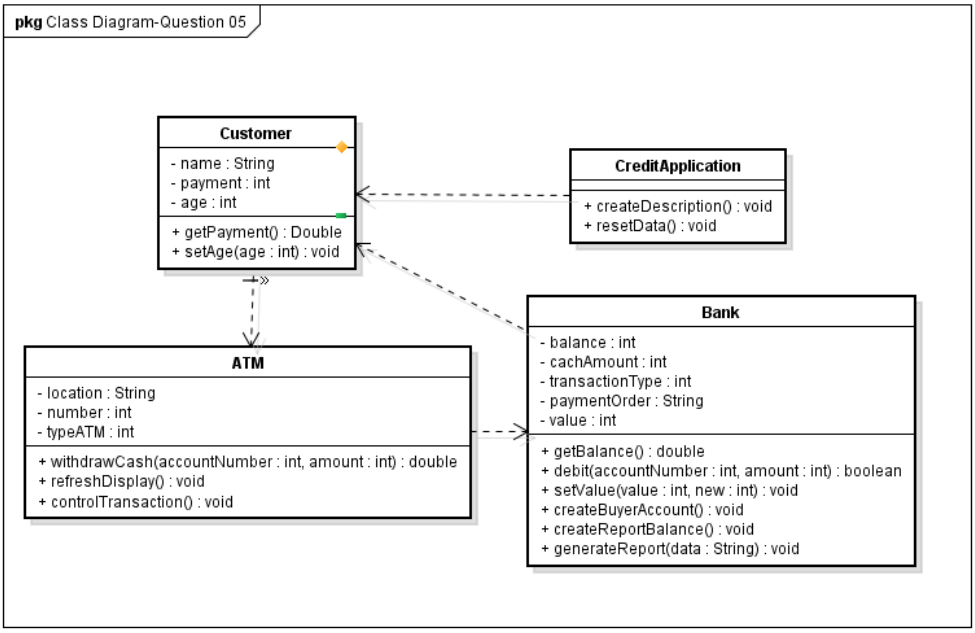
Caso de uso:	**Cadastrar novo usuário**
Resumo:	O usuário preenche o formulário de cadastro, informando seu e-mail, *user name*, senha, confirmação de senha e marcando estar de acordo com as regras de uso
Atores:	Usuário, Sistema de autenticação
Pré-condições:	Sistema aguardando criação de usuário
Descrição (fluxo):	1. Usuário informa todos os campos corretamente 2. Sistema cria cadastro do usuário 3. Sistema envia um e-mail de boas vindas ao usuário
Pós-condições:	Usuário criado com sucesso
Caminho Alternativo:	1. *User name* já existe 2. Sistema não cria novo usuário 3. Sistema informa que já existe um usuário com este *user name*

Fonte: Elaborado pela autora.

2.1.4 Diagrama de Classes

Os Diagramas de Classes têm o objetivo de demonstrar a estrutura estática do sistema a ser desenvolvido. Os elementos são representados por classes que contem seus respectivos atributos, métodos e relacionamentos. Pode-se observar, através da Figura 2, a elaboração de um Diagrama de Classes, com quatro classes relacionadas entre si. Cada uma delas apresenta seus métodos e atributos referentes, definindo o comportamento e relacionamento da estrutura do software que deverá ser desenvolvido (BEZERRA, 2007).

Figura 2: Diagrama de Classes



Fonte: FARIAS, 2012.

Os componentes do Diagrama de Classes permitem que se possa definir a estrutura das classes e os relacionamentos no momento em que se parte para o desenvolvimento e codificação do software. A maior vantagem é com relação aos diversos tipos de relacionamentos possíveis, o que deixa clara a estrutura do software, identificando as classes e métodos que podem ser reutilizados ou não (BOOCH et al., 2008).

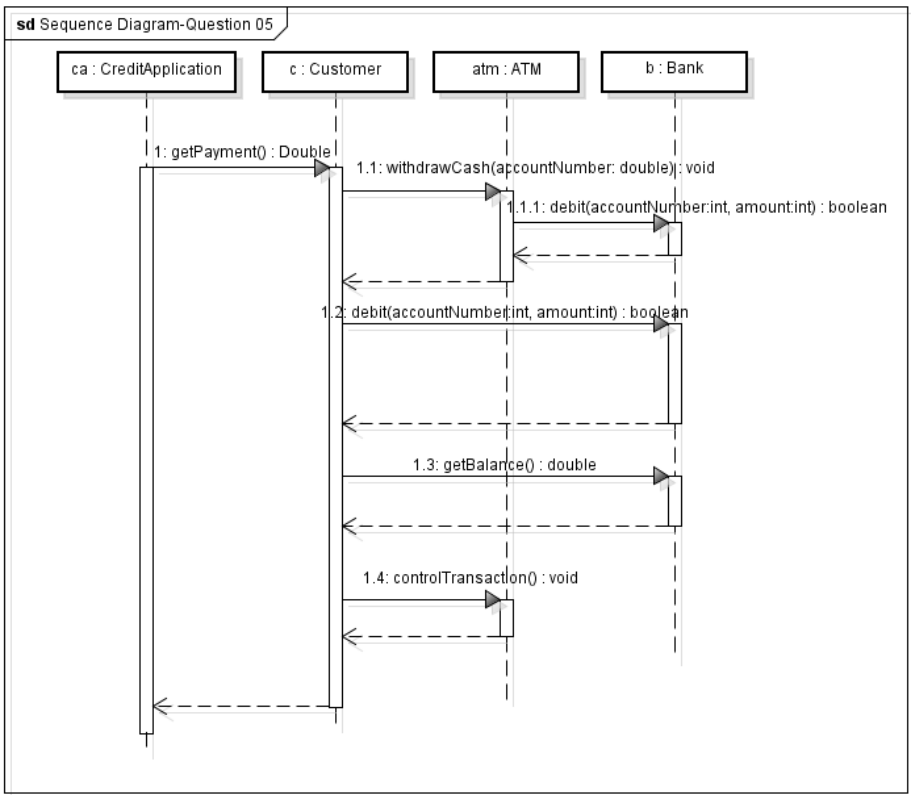
Para os principais tipos de relacionamentos, pode-se ainda, descrever suas funcionalidades e aplicabilidades, como:

- **Agregação**: A agregação é um caso da associação. A agregação indica que uma das classes do relacionamento é uma parte, ou está contida em outra classe.
- **Composição**: Relacionamento entre um elemento (o todo) e outros elementos (as partes) onde as partes só podem pertencer ao todo e são criadas e destruídas com ele.
- **Dependência**: São relacionamentos de utilização no qual uma mudança na especificação de um elemento pode alterar a especificação do elemento dependente
- **Realização**: Identifica classes responsáveis por executar funções para classes que representam interfaces
- **Restrição**: Tem por objetivo definir *constraints* iniciais nessa fase do projeto.
- **Estereótipos**: São uma maneira de destacar determinados componentes do diagrama, tornando explícito que eles executam alguma função diferente dos outros componentes do diagrama.
- **Herança múltipla**: É o conceito de herança de duas ou mais classes

2.1.5 Diagrama de Sequência

Diagramas de Sequência têm o objetivo de modelar os aspectos dinâmicos dos sistemas. Deixam claras as ligações entre os objetos com seus relacionamentos, especialmente, apresentando as mensagens que deverão ser trocadas entre eles (BOOCH et al., 2008). É dada ênfase na ordenação temporal das mensagens e na organização estrutural dos objetos que enviam e recebem as mensagens, assim como pode ser observado na Figura 3.

Figura 3: Diagrama de Sequência



Fonte: FARIAS, 2012.

A visão geral da interação é representada por uma coleção de diagramas de interação, onde cada um é um objeto da interação, a qual é composta de (1) um ponto de partida, (2) Casos de Uso (atores), (3) elementos, (4) atores (podem ser classes ou objetos), (5) Vínculos e (6) Mensagens. Um diagrama de interação é a realização de um caso de uso. É a descrição do comportamento do ponto de vista interno ao sistema, modelando cada um dos cenários do Caso de Uso (BEZERRA, 2007).

As classes de um Diagrama de Classes são representadas no formato de uma caixa, ou um retângulo, na qual nome_objeto[seletor] : nome_classe ref ocorrencia_interacao. Onde, (1) nome_objeto é o nome da instância da classe, este item é opcional, pois os objetos podem ser anônimos ou nomeados; (2) o seletor é utilizado quando se deseja referenciar um objeto dentro de uma coleção de objetos, como em uma lista ou um *array*, também opcional; (3) nome_classe é o nome da classe a qual está sendo instanciada; (4) ocorrencia_interacao indica que existe uma referência a outro diagrama de interação, que detalha como este objeto manipula as mensagens que recebe, também opcional (BOOCH et al., 2008).

As mensagens são os elementos centrais dos diagramas de interação, pois correspondem a uma solicitação de execução de uma operação em outro objeto, caracterizando um respectivo método no Diagrama de Classes. As mensagens podem se Síncronas ou Assíncronas, as quais significam que o remetente espera que o receptor execute a operação (processe a mensagem) antes de recomeçar o processamento, remetente fica bloqueado ou que o remetente não espera a resposta do receptor para continuar o processamento, respectivamente (BOOCH et al., 2008).

2.2 Inconsistências em Modelos UML

A utilização de modelos UML com inconsistências, que podem surgir a partir de modelos com Diagramas Conflitantes ou Não Correspondentes, pode incorrer em aumento de esforço para a realização da interpretação dos diagramas por parte dos desenvolvedores e analistas e acarretar em grandes consequências à qualidade do software produzido ao final do processo, alterando, inclusive, requisitos funcionais. Os custos altos para a criação e manutenção dos diagramas UML acabam não se justificando na maior parte das indústrias de software, pois, a qualidade dos diagramas não é avaliada e validada, ocasionando o aumento de custos e retrabalho, além de maior esforço para realização das atividades que envolvem a interpretação dos diagramas.

Inconsistências em modelos UML, principalmente envolvendo conflitos entre Diagramas de Classe e Sequência, por exemplo, podem variar desde problemas estruturais, envolvendo a aplicação de padrões de projetos (FARIAS, 2013), até inconsistências sintáticas e semânticas, que acabam dificultando a interpretação e compreensão destes diagramas. Inconsistências destes tipos acabam prejudicando a modularidade, a manutenção e a evolução dos modelos de software, afetando princípios básicos da analise e projeto de software orientado a objetos utilizando a UML. Estes impactos causados pelos efeitos da utilização de diagramas com inconsistências variam desde problemas de manutenção, causados pela degradação arquitetural do software, a problemas de má interpretação causada pela ambiguidade dos modelos (REDER, 2012).

Portanto, inconsistências podem ser definidas como:

- **Inconsistência estrutural:** são inconsistências que afetam a estrutura de um padrão de projeto de engenharia de software, não seguindo as boas práticas de modelagem para determinadas funcionalidades reconhecidas mundialmente, afetando a interpretação dos diagramas por parte de desenvolvedores e analistas que não se envolveram diretamente com a construção dos modelos multivisão (LARMAN, 2007).
- **Inconsistência sintática:** inconsistências nas regras e estrutura da linguagem UML, como o direcionamento errado de um evento em um diagrama de estado ou atributo sem tipo em um Diagrama de Classes. Normalmente alguns tipos desta inconsistência são detectados por grande parte dos ambientes de modelagem (REDER, 2012).
- **Inconsistência semântica:** são inconsistências no significado do diagrama, como um elemento presente em um Diagrama de Sequência sem o mesmo ter sido elencado em ao menos um Diagrama de Classes ou existir uma classe “venda” em um software para aluguéis, caracterizando uma inconsistência lógica, por exemplo. Este tipo de inconsistência dificilmente é identificado pelos softwares atuais no mercado (REDER, 2012).

As abordagens existentes para detecção de inconsistências em modelagens UML, como a implementação de ferramentas que acabaram não se consolidando no mercado nem em ambientes acadêmicos, podem ser classificadas em completas e parciais. As abordagens completas visam definir uma semântica totalmente formal para a notação UML completa (READER, 2013). Enquanto abordagens parciais selecionam um subconjunto da notação UML, normalmente relacionada com uma perspectiva como, for exemplo, a visão de processo. Sendo assim, desenvolve um significado bem definido para esse subconjunto com o

objetivo de identificar inconsistências automaticamente neste projeto parcial (READER, 2013).

Abordagens parciais podem ainda ser classificadas em dois grupos principais: um grupo com abordagem formal que se preocupa com mapeamento formal para as partes dos projetos UML, a fim de dar um significado para as construções de UML; e uma abordagem orientada a design, com ênfase na modelagem do UML, a fim de analisar propriedades do design (REDER, 2013). Métricas para inconsistência e incompletude podem ser bons indicadores de qualidade dos sistemas (REDER, 2012). A identificação de padrões de inconsistências a partir de métricas é importante e, se torna ainda mais relevante no contexto de desenvolvimento colaborativo, no qual o gerenciamento de modelos é imprescindível e a utilização de padrões de projetos de software é crucial (FARIAS, 2013).

Além disso, a composição de modelos UML que é uma prática comum na indústria, (BISCHOFF, 2016), tende a gerar muitas inconsistências no modelo final e algumas ferramentas utilizadas para a realização de composição de modelos disponíveis atualmente, tais como, *IBM RSA e Epsilon*, focam apenas em detectar as inconsistências sintáticas e raramente fazem alguma menção em relação à identificação de padrões de inconsistências semânticas e estruturais, bem como às suas propagações, co-ocorrências e inter-relacionamentos (FARIAS, 2012). Estudos apontam que o uso de métricas pode ajudar na detecção de inconsistências, como (FARIAS, 2008), (FARIAS, 2011), (FARIAS, 2012) e (FARIAS, 2013).

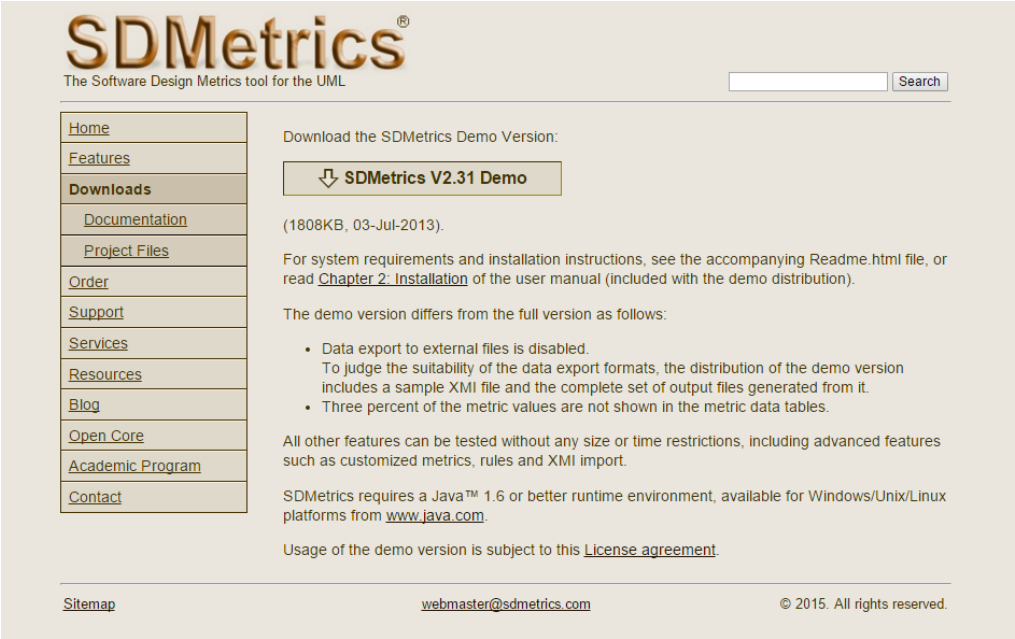
2.3 *Software Design Metrics* (SDMetrics)

Para auxiliar no desenvolvimento de uma aplicação que possibilite a detecção de inconsistências em modelos UML multivisão, a aplicação *Software Design Metrics* (*SD Metrics*) foi estudada. Com o slogan "*The Software Design Metrics tool for the UML*", o *SD Metrics* [1] é uma ferramenta que, através da leitura de arquivos no formato *eXtensible Markup Language (XML)* permite a identificação de diferentes métricas de design UML (*SD Metrics*, 2016). Sem o *SD Metrics*, o desenvolvedor ou analista de sistemas acaba tendo de utilizar apenas sua experiência em interpretação de Diagramas UML, enquanto que, através da utilização do *SD Metrics*, pode-se utilizar métricas e métodos que validam regras e apresentam um resultado.

Para a realização da instalação do software, é necessário acessar o site da aplicação e conforme Figura 4, pode-se realizar o download de uma versão Demo da ferramenta. Para a utilização da ferramenta completa, deve-se, conforme Figura 5, entrar em contato com o desenvolvedor da aplicação.

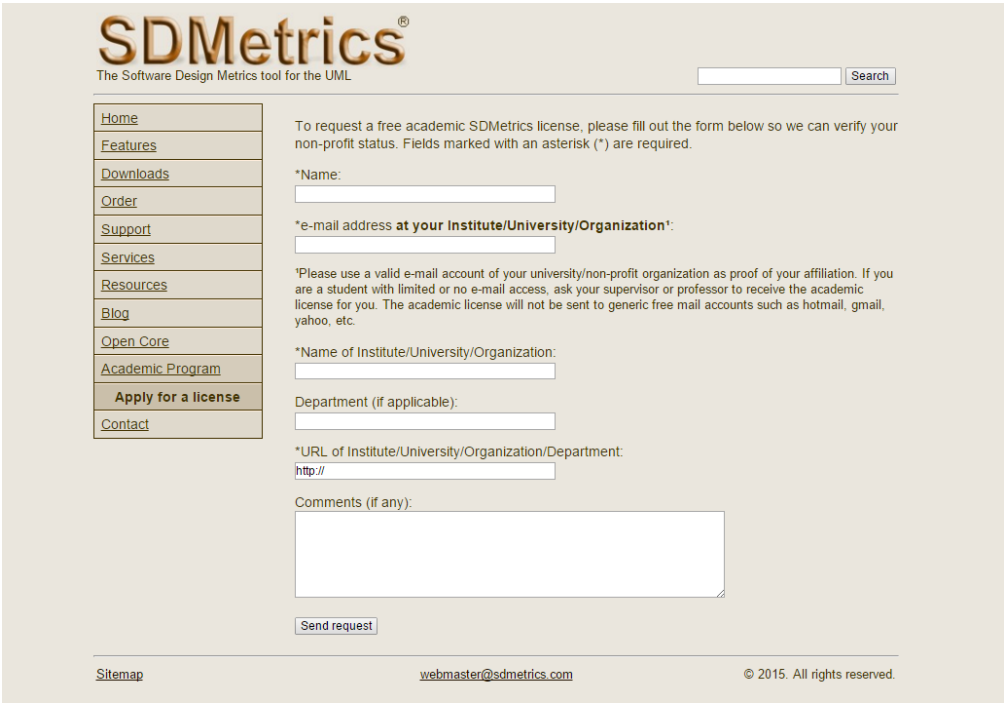
[1] http://www.sdmetrics.com/

Figura 4: SDMetrics Demo



Fonte: http://www.sdmetrics.com/

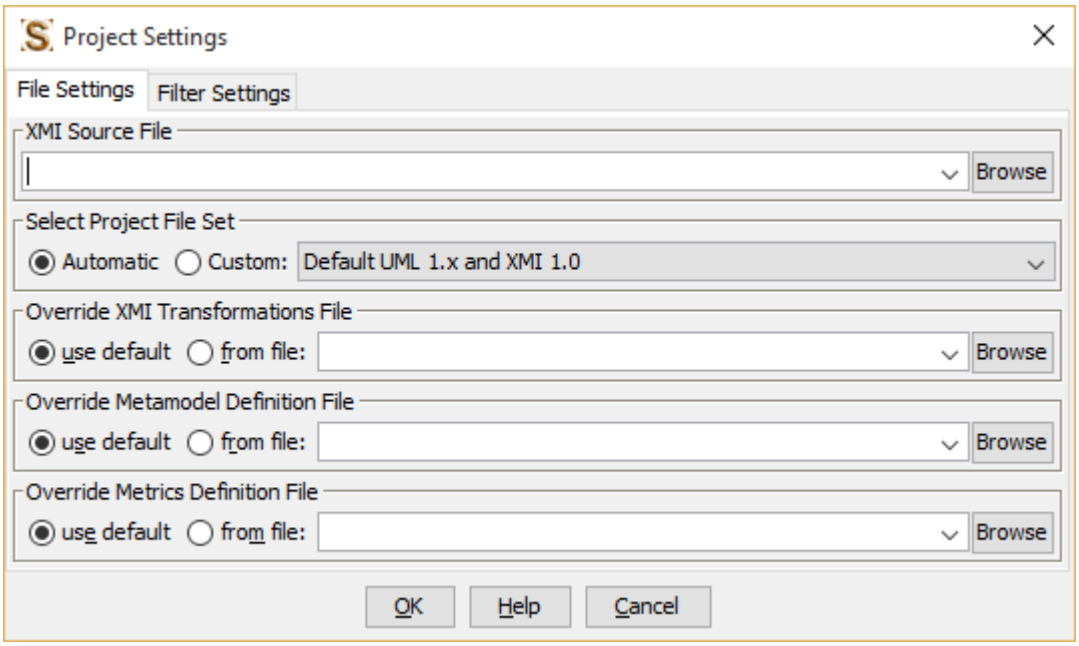
Figura 5: Solicitando SDMetrics full



Fonte: http://www.sdmetrics.com/

Uma vez instalada a versão completa do software, pode-se iniciar sua utilização, realizando a importação dos XML dos Diagramas que serão utilizados em experimentos e desenvolvimentos. Conforme apresentado na Figura 6, deve-se selecionar o arquivo XML do diagrama que será analisado.

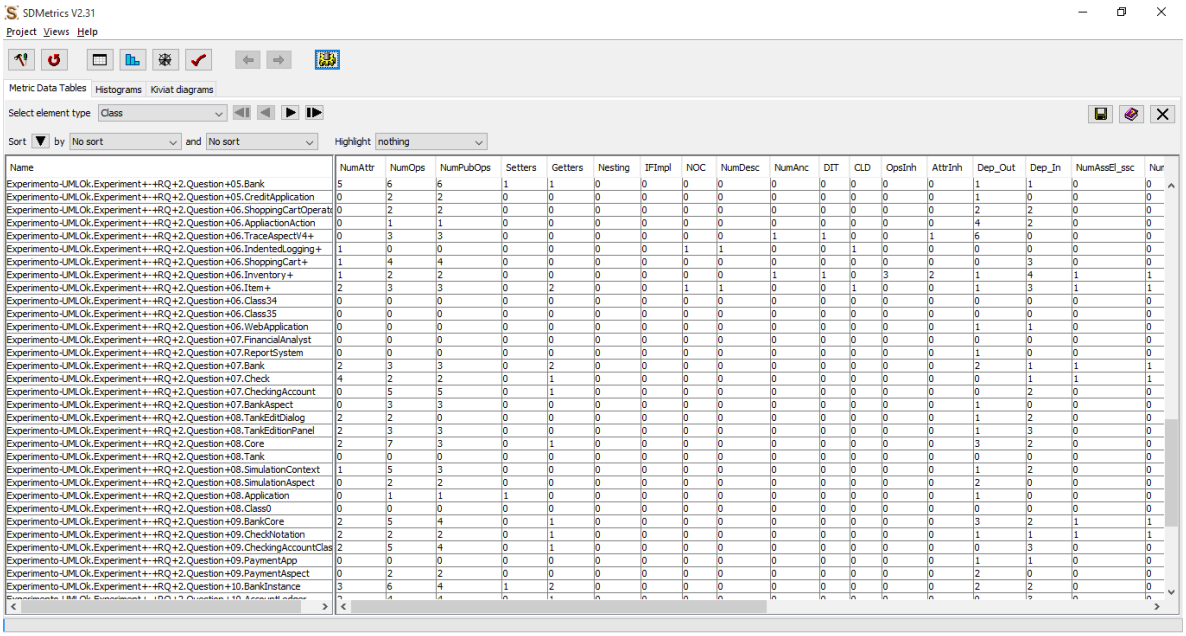
Figura 6: inclusão dos XML



Fonte: Ferramenta SDMetrics

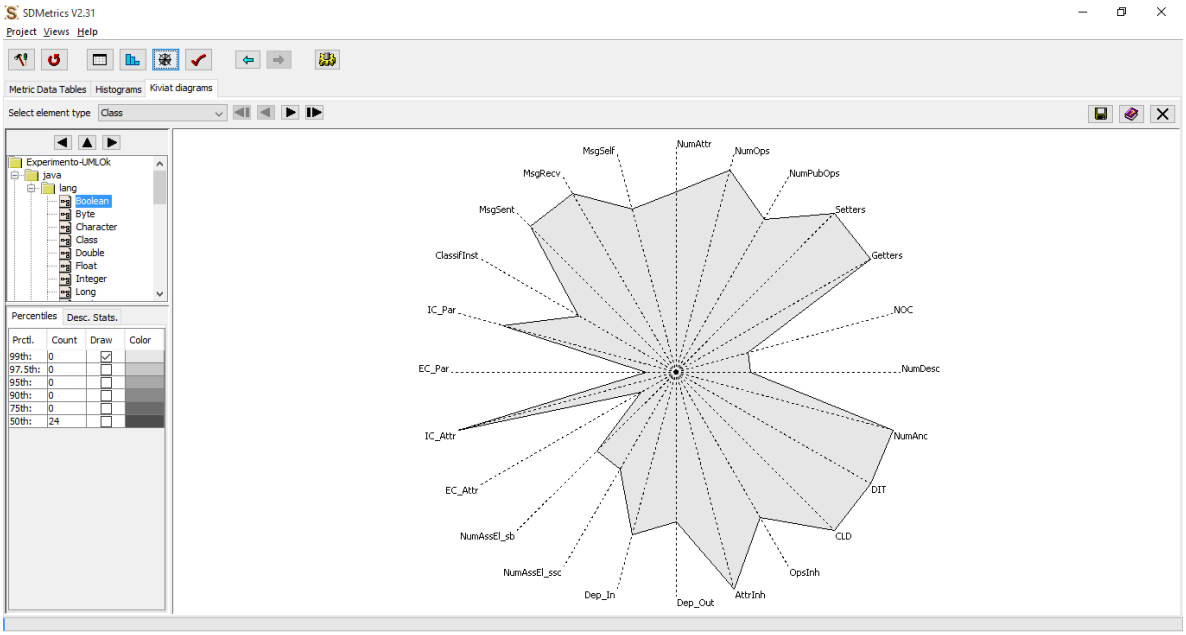
Depois de concluído o processamento dos XML, a aplicação apresenta as métricas de diferentes formatos para avaliação, conforme Figuras 7 e 8.

Figura 7: Visualização de Métricas em listagem



Fonte: Ferramenta SDMetrics

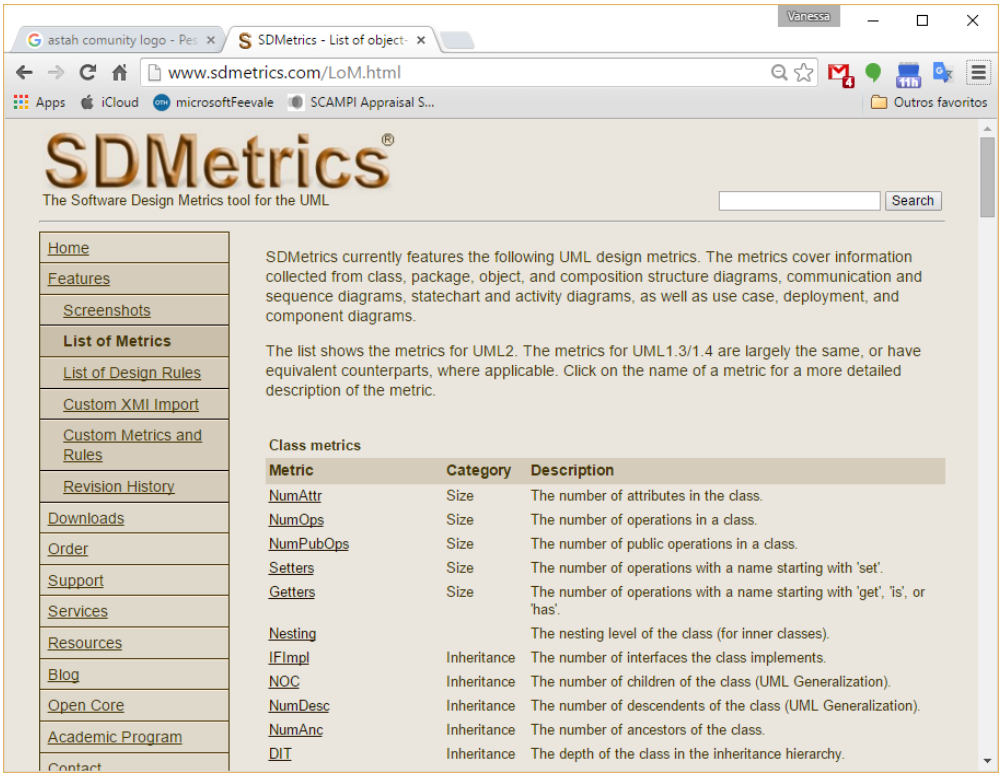
Figura 8: Visualização de Métricas através de Radar



Fonte: Ferramenta SDMetrics

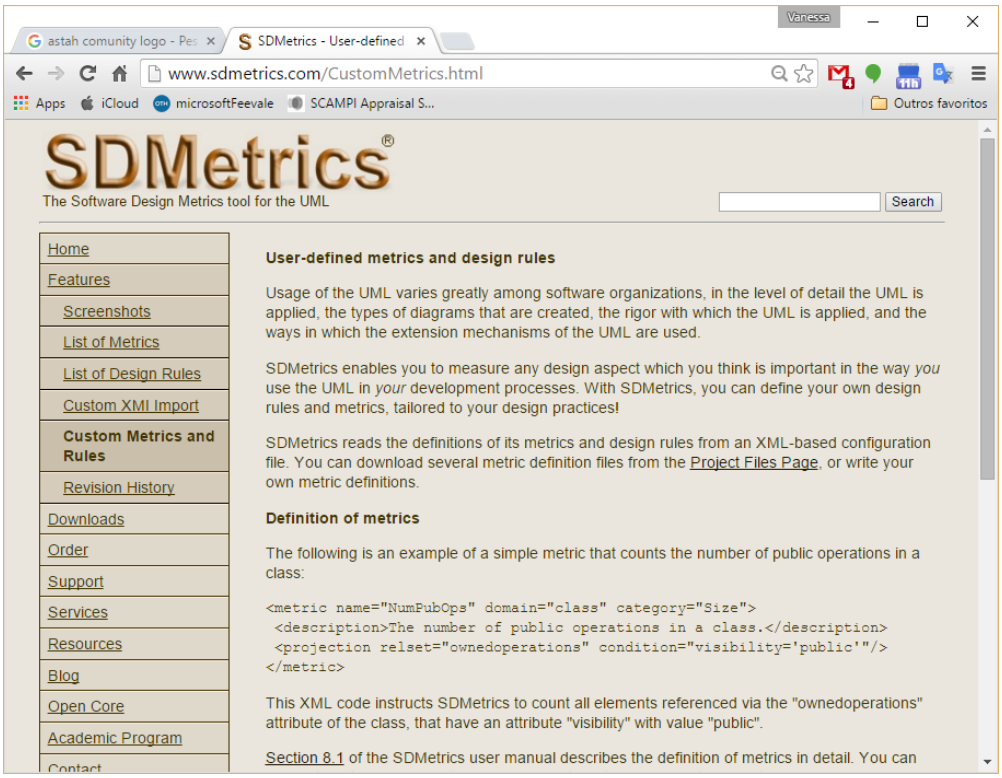
Esta ferramenta foi escolhida para fazer parte deste estudo, pois permite a configuração de diferentes métricas e regras para avaliação de diagramas UML. A Figura 9 apresenta um formato de visualização das métricas que o sistema reconhece como padrão, mas, conforme pode ser observado na Figura 10, as regras podem de definição de métricas podem ser utilizadas para configuração do software.

Figura 9: Lista de Métricas



Fonte: http://www.sdmetrics.com/

Figura 10: Definição de Novas Métricas



Fonte: http://www.sdmetrics.com/

A definição de novas métricas permite a utilização da ferramenta para verificações comparativas e completas sobre os arquivos XML dos Diagramas que podem ser analisados, oferecendo oportunidades de configuração conforme a necessidade do usuário (*SD Metrics*, 2016).

3 TRABALHOS RELACIONADOS

Neste capitulo são relatados uma série de estudos, incluindo artigos, mapeamentos sistemáticos e revisões sistemáticas que avaliam o estado da arte sob a perspectiva de identificação de inconsistências, tais como (SIMMONDS, 2005) (LAVANYA, 2005) (MALGOUYRES, 2006) (LOPEZ-HERREJON, 2010), e seus impactos (LANGE, 2006) (EGYED, 2007) (EGYED, 2008) (NUGROHO, 2013), que se relacionam com os objetivos de pesquisa deste trabalho (descritos na seção 1.2) e que contribuíram de alguma forma para a sua construção e desenvolvimento.

Encontra-se bem estabelecido na literatura atual, que a utilização de métricas para inconsistência e incompletude de diagramas UML podem ser bons indicadores de qualidade dos sistemas de software (LANGE, 2003). Assim, métricas para análise de modelos UML buscam quantificar atributos relacionados às regras de modelagem de software para representar a qualidade; isto é, minimizar as inconsistências e identificar seus impactos nos modelos de software, visando qualidade e produtividade. Através da elaboração de um estudo de mapeamento sistemático, pôde-se afirmar que há poucos estudos que abordam a identificação de inconsistências em diagramas através de métricas (LANGE, 2003) (LANGE, 2006) (NÖHRER, 2011) (THUNG, 2014) (CHAUDRON, 2014) e que realizam o relacionamento entre estas inconsistências e seus impactos (LANGE, 2006) se não identificadas e tratadas.

Os estudos que tratam sobre inconsistência em diagramas UML estão normalmente divididos entre abordagens sintáticas (CHAUDRON, 2014) (THUNG, 2014)(LIU, 2013) (REDER, 2012) e semânticas (LIU, 2000), não considerando os dois aspectos dos modelos. Além disso, a maior parte dos estudos avalia modelos parciais (CHAUDRON, 2014) (THUNG, 2014) (REDER, 2012), e não o projeto UML completamente. Há estudos que apresentam abordagens parciais que procuram selecionar um subconjunto de notação UML (EGYED, 2013), geralmente envolvendo apenas as regras formais da própria linguagem de modelagem, neste caso, a UML. Sendo assim, observa-se que existe uma grande tendência para o desenvolvimento de aplicações que detectam automaticamente as inconsistências nos modelos (KOTB, 2010) (KOCHAREKAR, 2010) (EGYED, 2011), representando a maior parte da literatura encontrada na área.

Porém, o desenvolvimento de abordagens orientadas a modelos que apresentam apenas uma série de regras de consistência definidas por meio de termos formais não compreendem todo o universo da integridade em um modelo UML. É necessária uma análise completa de inconsistências, envolvendo a análise semântica de modelos e as relações entre os diferentes diagramas de UML, validando também sua sintaxe. Essas relações devem também fazer sentido para o desenvolvedor (FARIAS, 2013).

Por fim, estudos sobre inconsistências em modelagem UML, desde sua detecção até seu impacto sobre a qualidade e a produtividade, são limitados e não fornecem uma conclusão geral do estado da arte sobre inconsistências. Mais especificamente, foram identificadas algumas lacunas importantes na literatura: (1) a maioria dos estudos apresentados envolve apenas a perspectiva de analistas de software e não dos engenheiros ou desenvolvedores de software; (2) não há relações claras entre a detecção e os impactos da modelagem UML com inconsistências; (3) há poucos estudos relativos à detecção de inconsistências na modelagem UML através de métricas; e, por fim, (4) não está consolidado um método automatizado para verificação de modelos UML antes de serem enviados para o desenvolvimento.

Como previamente mencionado, o impacto de inconsistências na qualidade e na produtividade tornou-se, portanto, o principal motivador para a investigação dos métodos de verificação dos diagramas UML atuais e, se estes podem minimizar tais impactos, tornando a UML mais atrativa à indústria.

Sendo assim, uma revisão da literatura sobre detecção de inconsistências em diagramas UML, com o objetivo de identificar o estado-da-arte através de publicações em fóruns, eventos, revistas e periódicos, foi um passo importante para o desenvolvimento deste trabalho. Buscou-se identificar métodos que minimizassem os impactos de inconsistências na qualidade e produtividade do desenvolvimento. Ainda, a compreensão das possíveis falhas no uso de UML e impactos na produtividade e na qualidade do código produzido, a identificação de lacunas e o relacionamento de quais áreas de pesquisa e quais os tipos de estudo ainda há escassez de publicações é relevante para estudos nesta área. Com este propósito em mente, decidiu-se realizar uma revisão da literatura relacionada a este tema, a fim de responder a principal questão de pesquisa elencada para o estudo de mapeamento sistemático realizado:

Como a literatura atual tem contribuído para a definição de diferentes técnicas que permitem a detecção de inconsistências em diagramas UML e como avalia quais os impactos destas inconsistências no desenvolvimento e na qualidade de software?

3.1 Metodologia do estudo de mapeamento sistemático

Foi seguida uma metodologia para a realização da revisão sistemática através de uma rigorosa abordagem sistemática, conforme descrita por (KITCHENHAM, 2007) e (PETERSEN, 2015). Sendo assim, foi realizado um estudo de mapeamento sistemático, devido à necessidade de adotar abordagens sistemáticas no sentido de avaliar e agregar os resultados da investigação, com o objetivo de fornecer um resumo equilibrado e objetivo de evidências de pesquisa.

É importante ressaltar que a detecção de inconsistências em modelos UML é um assunto bastante novo, que vem em consequência ao aumento da utilização dos diagramas UML. Com a identificação de poucos estudos sobre a detecção de inconsistências, pode-se afirmar que não há uma metodologia consolidada e reconhecida pela indústria para minimizar os impactos do uso de diagramas com inconsistências nem para detecção das inconsistências. Além disso, os próprios impactos do uso de diagramas com inconsistências são de pouco estudo, não havendo muitos dados sobre o quanto cada tipo de inconsistência impacta de maneira diferente em um projeto de software. Ainda, a detecção das inconsistências realizada quando o projeto UML está concluído também acaba inviabilizando a execução de procedimentos de verificação e validação, conforme (REDER, 2012).

Igualmente, o momento de detecção das inconsistências também foi levado em consideração, uma vez que os maiores impactos se dão no momento de interpretação do diagrama UML pelo desenvolvedor e nos momentos de manutenção do código quando o projeto já tiver sido concluído. A grande maioria dos estudos apresentou a detecção de inconsistências apenas na passagem da fase de análise para o desenvolvimento, porém, a atualização dos diagramas, quando realizadas tarefas de manutenção são pouco tratadas.

Estas considerações levaram o estudo a concentrar-se na importância de investigar os tipos de inconsistências que são comumente identificadas, tentando descobrir se os métodos atuais consideram todos os aspectos das inconsistências entre os diagramas UML, considerando inconsistências sintáticas e semânticas.

3.1.1 Planejamento do estudo

A primeira etapa do estudo realizado foi o planejamento, que definiu os critérios e etapas que fariam parte do processo controlado realizado para a coleta e análise dos artigos selecionados (PETERSEN, 2015). Inicialmente, foram definidas as questões de pesquisa que norteiam todas as outras atividades que foram realizadas. O objetivo de definir questões de pesquisa detalhadas para o estudo de mapeamento sistemático apresentadas na Tabela 3 é poder identificar quais os aspectos relevantes, juntamente com as motivações que impulsionaram a pesquisa.

Tabela 3: Questões de Pesquisa

Questões de pesquisa	Motivação
QP1: Quais diagramas são usados para identificar inconsistências?	Identificar os principais tipos de diagramas utilizados nos estudos para detecção inconsistência nos modelos UML, buscando verificar se há relacionamento entre os diversos diagramas de um modelo UML.
QP2: Quais são os tipos de inconsistências que existem nos diagramas?	Identificar quais os tipos de inconsistências detectados em cada tipo de diagramas apresentados nos estudos, se tratam inconsistências sintáticas e/ou semânticas.
QP3: Como as inconsistências são detectadas?	Identificar a metodologia utilizada atualmente para a detecção das inconsistências nos diagramas, seja de forma manual ou automática.
QP4: Quais os impactos das inconsistências?	Identificar se os estudos apresentam os impactos de modelos UML com inconsistências, quando em tempo de desenvolvimento ou manutenção.
QP5: Em que fase do projeto são detectados o impacto das inconsistências?	Verificar se os estudos consideram o ponto de vista do analista (ainda em tempo de análise do software), do engenheiro (quando na fase de desenvolvimento ou manutenção) ou do desenvolvedor de software (também entre as fases de desenvolvimento ou manutenção).
QP6: Inconsistências são identificadas e como são representadas através de métricas?	Investigar se os estudos utilizam abordagens de definição de métricas para detecção de inconsistência em modelos UML e como são definidas estas métricas.

Fonte: Elaborado pela autora.

Em seguida, foram definidos termos relevantes para identificação dos artigos, com o objetivo de formar uma chave de busca (*Search String*). Esta *Search String* foi utilizada para encontrar os artigos nas principais bibliotecas digitais disponíveis para utilização através da Biblioteca Virtual [2]da Universidade do Vale do Rio dos Sinos (UNISINOS).

A metodologia utilizada para definição da chave de busca seguiu as orientações de (FERNÁNDEZ-SÁEZA, 2013) para a realização de um estudo de mapeamento sistemático, revisões sistemáticas e demais pesquisas. Foram seguidas cinco etapas para definir as chaves de busca: (1) definição das principais palavras-chave; (2) identificação de palavras alternativas, sinônimos ou termos relacionados com palavras-chave principais; (3) verificação de que as principais palavras-chave estão realmente contidas em artigos da categoria de pesquisa; (4) utilização do relacionamento entre sinônimos associados, palavras alternativas ou termos relacionados com as principais palavras-chave utilizando o valor booleano "*OR*"; e (5) utilização do relacionamento entre as principais palavras-chave utilizando o valor booleano "*AND*".

[2] http://www.unisinos.br/biblioteca

Tabela 4: *Search Strings*

Palavra-chave	**Sinônimos e alternativas**
UML	*Unified Modelling Language OR Model*
Inconsistency	*Inconsistencies*
Maintenance	*Maintainability OR Modularity OR Reusability OR Analyzability OR Changeability OR Evolution OR Evaluability OR Modification OR Stability OR Testability OR Comprehensibility OR Comprehension OR Understandability OR Understanding OR Misinterpretation*
Empirical	*Experiment OR Survey OR Case study OR Action research*

Fonte: Elaborado pela autora.

Foram definidas, conforme apresentado na Tabela 4, as palavras-chaves "UML", "Inconsistência", "Manutenção" e "Empírica" como termos principais. Assim como, foram selecionadas palavras sinônimas e palavras alternativas para as palavras-chave definidas. Uma vez concluída a Tabela 4, várias combinações entre as chaves de busca foram desenvolvidas. No entanto, a *Search String* que retornou mais resultados relevantes para o estudo nos motores de busca selecionados foi:

((UML OR (Unified Modelling Language) OR Model) AND (Inconsistencies OR Inconsistency) AND (Maintenance OR Maintainability OR Modularity OR Reusability OR Analyzability OR Changeability OR Evolution OR Evaluability OR Modification OR Stability OR Testability OR Comprehensibility OR Comprehension OR Understandability OR Understanding OR Misinterpretation) AND (Empirical OR Experiment OR Survey OR Case study OR Action research))

A partir de então, a *Search String* foi aplicada nas bibliotecas digitais disponíveis na Biblioteca Virtual da UNISINOS e demais acessíveis, conforme Tabela 5.

Tabela 5: Resumo da Estratégia de Pesquisa

Bancos de dados pesquisados	• Science Direct • Library CiteSeerX • IEEE Xplore • ACM Digital Library • SpringerLink
Itens alvos	• *Journal papers* • *Workshop papers* • *Conference papers*
Pesquisa aplicada a	Resumo - quando não era possível, procurou-se no texto integral
Linguagem	Artigos escritos em Inglês
Período de publicação	De janeiro de 2000 a julho de 2016 (inclusive)

Fonte: Elaborado pela autora.

Uma vez definida a *Search String*, os artigos incluídos foram os que apresentaram qualquer tipo de estudo empírico sobre a detecção de inconsistências em modelos UML, que tivessem sido escritos em Inglês e que tinham data de publicação entre janeiro de 2000 e junho de 2016, com o objetivo de identificar literaturas atuais sobre o tema. Por fim, a busca foi limitada a estudos publicados em bibliotecas digitais e revistas, conferências e workshops.

Com o objetivo de refinar a busca entre os quase 1 milhão de artigos incluídos, foi necessária a definição de critérios de exclusão, tais como apresentados na Tabela 6.

Tabela 6: Resumo da Estratégia de Seleção

Critérios de Inclusão	Apenas inglês

	Data de publicação: a partir de janeiro de 2000 a julho de 2016 Obras publicadas e referenciadas Termos satisfazendo a cadeia de pesquisa
Critérios de exclusão para títulos e resumos	Estudos duplicados entre bancos de dados de pesquisa Artigos cuja apenas os resumos estavam disponíveis Artigos com baixa relevância e apenas referenciados
Critérios de exclusão para o texto completo	Artigos que tratam de extensões UML Artigos que são um resumo de uma oficina Artigos que não afetam diretamente a detecção de inconsistências

Fonte: Elaborado pela autora.

Por fim, foram definidos os dados que seriam coletados a partir dos artigos selecionados. O método utilizado para extração e tabulação dos dados envolveu a construção de uma planilha, com o objetivo de resumir o estado da arte sobre detecção de inconsistência em modelos UML e seus impactos na qualidade e produtividade de software, considerando: (1) dados implícitos aos critérios de inclusão e exclusão, como: a data de publicação, os fóruns de publicação, eventos ou jornais, e os motores de busca; (2) os dados básicos de publicações como autor e título; e, finalmente, (3) as informações relacionadas a cada uma das questões de pesquisa, tais como:

- **Tipos de Diagramas (QP1).** Tem-se o objetivo de identificar o conjunto de diagramas da notação UML apresentados, a partir dos estudos coletados. Eles foram contabilizados conforme os tipos de diagramas, incluindo Diagramas de Classe (CD), Diagramas de Sequência (SD), Diagramas de Estado (ED) e Diagramas de Caso de Uso (UC). Nenhum dos diagramas encontrados foi associado a uma versão específica da UML.
- **Tipos de Inconsistências (QP2).** Buscou-se selecionar as inconsistências em modelos UML, classificando-as como sintáticas, semânticas ou sintáticas e semânticas.
- **Métodos de Detecção de Inconsistências (QP3).** Identificaram-se os métodos de detecção de inconsistências em modelos UML apresentados nos artigos selecionados. Observou-se que a detecção de inconsistências pode ocorrer através de diversos métodos, tais como a configuração das regras de linguagem de modelação UML, a implementação de algoritmos de execução métricas ou através de métodos mistos, que envolvem duas ou mais técnicas de detecção.
- **Impactos das Inconsistências (QP4).** Os impactos das inconsistências nos modelos, diagramas ou projetos UML também foram considerados neste estudo, buscando identificar como elas afetam o desenvolvimento de software, das mais variadas formas. Deve-se levar em consideração também o custo benefício da correção de determinadas inconsistências porque alguns não afetam diretamente a produtividade qualidade e desenvolvimento é podem ser facilmente contornadas e/ou interpretadas pelo desenvolvedor, no entanto, há inconsistências graves, que são frequentemente combinações diversas falhas que afetam a interpretação dos diagramas, causando erros e retrabalho de testes e desenvolvimento, afetando diretamente a qualidade do software desenvolvido e produtividade da equipe envolvida, portanto, buscou-se identificar quais estudos consideram estes impactos.
- **Fase do Projeto na qual as Inconsistências são Detectadas (QP5).** Há interação de diferentes papéis no desenvolvimento de software, e estas também

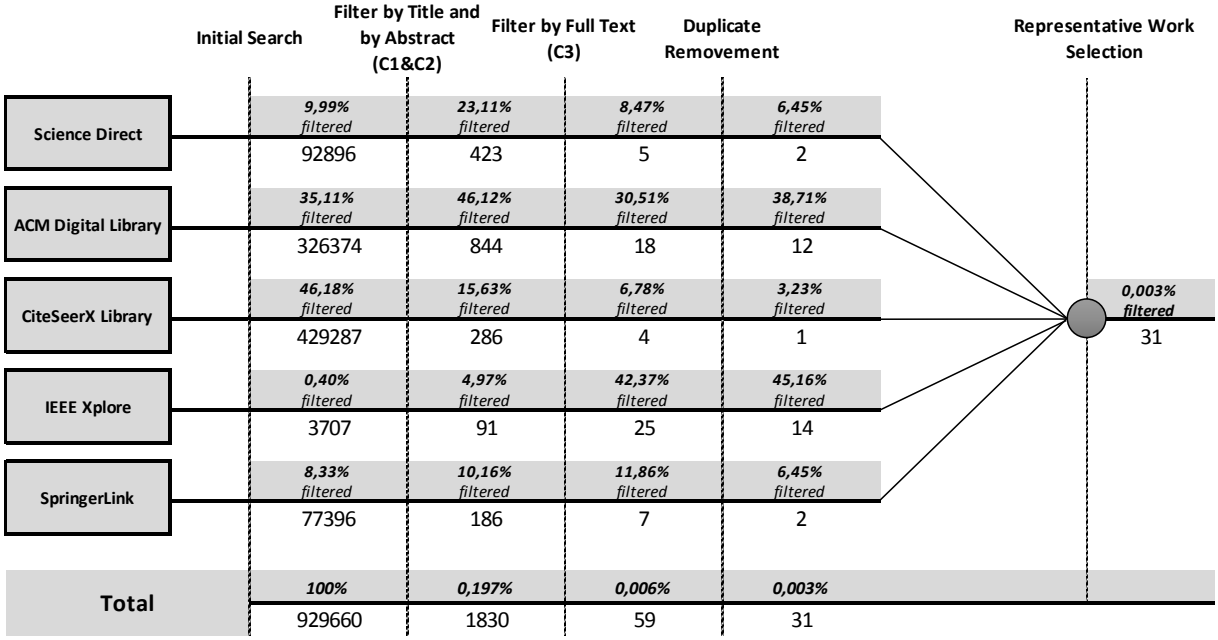
foram consideradas neste estudo. Assim, durante todo o ciclo de vida de um projeto de desenvolvimento podem ser encontradas inconsistências nos modelos e diagramas UML. Quando em tempo da análise de design e criação de diagramas, a detecção ocorre através da análise do próprio analista de software, já, ao passar para a fase de desenvolvimento, é um desenvolvedor o responsável pela detecção de inconsistências e contorná-las ou não, conforme sua percepção pessoal, e, finalmente, no pós-desenvolvimento, em tempo de manutenção do software, quando pode haver a aplicação da engenharia reversa para detectar inconsistências, essa tarefa é deixada para o Engenheiro de Software.

- **Uso de ou não de Métricas (QP6).** Esta é uma questão de pesquisa que visa proporcionar uma visão geral da direção dos estudos em curso para identificar inconsistências em modelos UML a partir de métricas. Ele procura compreender como a literatura descreve as métricas de inconsistências e as métricas que representam a qualidade de um diagrama, avaliar a qualidade presentes tanto sintática como a semântica de diagramas individuais como modelos consistem em mais de um tipo de diagrama, para projetar software completo.

3.2 Realização da seleção de artigos para o mapeamento sistemático

Na fase da execução do estudo, pode-se ver como os diferentes artigos foram selecionados com base em sua relevância. A Figura 11 ilustra o resultado obtido em cada passo ao longo do processo de filtragem, que é descrito como se segue.

Figura 11: Estudos obtidos em cada passo



Fonte: Elaborado pela autora.

Os passos utilizados para encontrar publicações, mostrados na Figura 11 mostram o que foi selecionado em cada passo.

• **Passo 1. Procura inicial:** exibe a quantidade de documentos eletrônicos que foram encontrados utilizando a cadeia de pesquisa.

• **Passo 2. Filtro por título e por resumo (C1 e C2):** mostra os resultados que foram encontrados após a aplicação das palavras-chave (UML e inconsistência) no filtro de pesquisa.

• **Passo 3. Filtro por texto completo (C3):** mostra o número de artigos que cumpriram com os requisitos estabelecidos em busca de questões e também aplicados os critérios de exclusão e inclusão.

• **Passo 4. Remoção de documentos duplicados:** todos os estudos repetidos, que foram encontrados em mais de um dos motores de busca foram removidos, com apenas os únicos estudos.

• **Etapa 5. Artigos representativos selecionados:** representa o número total de artigos selecionados para este estudo mapear sistematicamente.

Por fim, os 31 artigos selecionados com base na aplicação dos filtros e execução das etapas acima apresentadas, foram organizados e apresentados conforme pode ser visto no Apêndice A.

3.3 Síntese dos resultados encontrados

Uma vez aplicados os filtros e critérios de inclusão e exclusão, os resultados encontrados, através da consolidação dos dados dos artigos, que respondem à cada uma das questões de pesquisa apontadas na Tabela 3, é apresentado.

3.3.1 Tipos de Diagramas (QP1)

A característica principal é que o diagrama de classes UML tem sido o modelo de design mais adotado, sendo explorado em todos os estudos primários. Os diagramas de sequência e estado são os diagramas mais utilizados, isto é, 77% (24/31) e 29% (9/31), respectivamente. Apenas 3% dos estudos abordaram outros tipos de diagramas UML para detectar inconsistências ou avaliar o impacto de inconsistências na qualidade e produtividade do desenvolvimento de software. Analisando a detecção de inconsistências usando mais de um diagrama, pode-se observar que 29% (9/31) dos estudos primários consideram ambos os diagramas de classe, sequência e estado de UML (por exemplo, o estudo [S06], Apêndice A).

3.3.2 Tipos de Inconsistências (QP2)

Os dados mostram que os estudos primários visam a detecção de inconsistências sintáticas (96%, 30/31), enquanto as inconsistências semânticas não são tão abordadas. Somente 29% (9/31) das obras suportaram inconsistências sintáticas e semânticas. Ainda assim, a curiosidade é que o número de trabalhos explorando um tipo específico de

inconsistência é discrepante. Enquanto 75% (21/28) dos estudos só podem detectar inconsistências sintáticas, apenas um é capaz de detectar inconsistências semânticas, sem detectar um sintático, conforme o estudo [S01], Apêndice A. Esse resultado pode ser, em parte, explicado porque o significado dos modelos de projeto raramente é representado de forma formal. Por outro lado, as inconsistências sintáticas podem ser localizadas verificando se as regras de bem-formação da metaclasse definidas no metamodelo UML, conforme o estudo [S06], Apêndice A, foram desafiadas ou não. Se uma regra não é satisfeita, então o diagrama propriamente dito não pode ser considerado como significativo, resultando de inconsistências. Hoje, algumas estruturas de modelagem importantes (por exemplo, EMF (*Eclipse Modeling Framework*)) e trabalhos académicos (por exemplo, estudo [S13], Apêndice A) apresentam características para suportar a detecção de inconsistências sintáticas.

3.3.3 Métodos de Detecção de Inconsistências (QP3)

Considerando o nível de automação para detectar inconsistências nos modelos UML, observa-se que a maioria dos estudos (90%, 28/31) procura detectar inconsistências de forma automática. Para isso, geralmente definem um conjunto de regras e, em seguida, verificam se essas regras estão satisfeitas. Tipicamente, as inconsistências consideradas são a sintática; contudo, estes estudos ainda não foram aplicados a projetos de grande dimensão. Podemos também destacar um baixo número de estudos (6%, 2/31) que tem como objetivo detectar inconsistências de forma manual. Embora estas abordagens manuais permitam a detecção de inconsistências sintáticas e semânticas (por exemplo, os estudos [S05][S25], Apêndice A), a sua utilidade pode ser questionável em grandes projetos de software de grande escala, onde o tempo é apertado. Outras duas abordagens identificadas (isto é, conforme os estudos [S05][S07], Apêndice A) usam regras de modelagem e métricas para detectar inconsistências.

3.3.4 Impactos das Inconsistências (QP4)

Observou-se que poucos estudos (19%, 6/31) abordaram questões relacionadas com a compreensão do impacto das inconsistências nas práticas de desenvolvimento de software. Em geral, esses estudos mostram os efeitos de inconsistências na fase de desenvolvimento de software, discutem como eles afetam a produtividade do desenvolvedor e causam falhas no desenvolvimento de software.

3.3.5 Fase do Projeto na qual as Inconsistências são Detectadas (QP5)

A característica principal é que as inconsistências ocorrem geralmente em três pontos durante todos os projetos de desenvolvimento do software. A maioria dos estudos (84%, 26/31) indica que os analistas de software geralmente percebem inconsistências durante a fase de análise e concepção. Se os analistas não detectarem inconsistências, os desenvolvedores de software acabam tendo que detectar mais tarde. Uma pequena quantidade de estudos (10%, 3/31) relatou que inconsistências foram detectadas durante a fase de desenvolvimento. Além disso, apenas dois estudos [S15] e [S27], Apêndice A, (6%, 2/31) relataram que

inconsistências são detectadas inconsistências por engenharia de software durante a engenharia reversa de diagramas UML.

3.3.6 Uso ou não de Métricas (QP6)

Para resolver esta questão, os dados mostram que o uso de métricas não tem sido amplamente aplicado para detectar inconsistências. Apenas 16% (5/31) dos estudos primários adotaram métricas como um mecanismo para auxiliar na identificação de inconsistências em modelos de projeto multivisão.

3.4 Discussões e Direções Futuras

Alguns resultados adicionais foram possíveis de serem observados através da consolidação dos dados deste estudo de mapeamento sistemático. A Tabela 7 mostra que as conferências e revistas com maior número de estudos primários publicados. A maioria dos estudos primários (39%, 12/31) foi publicada em três conferências principais, ou seja, a *International Conference on Software Engineering (ICSE), Automated Software Engineering (ASE)*, e *Software Applied Computing (SAC),* enquanto a *IEEE Transactions on Software Engineering (TSE)* é a revista com o maior número (6%, 2/31) dos estudos publicados. Além disso, buscou-se descobrir como esses estudos foram publicados nas últimas décadas. A Figura 12 mostra o número de publicações realizadas de 2000 a 2016.

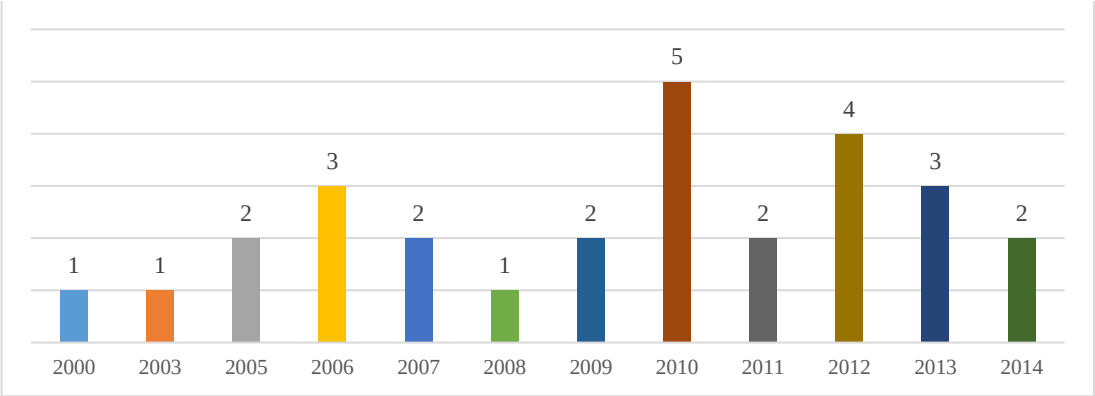
Tabela 7: Número de itens por evento / revista

Publicação	Montante aprovado	Percentual
ICSE	6	21%
ASE	4	14%
IEEE Trans. Software Eng	3	11%
SAC	2	7%
Outros	13	46%

Fonte: Elaborado pela autora.

Neste período, 2010 foi o ano com maior número de publicações (16%, 5/31), enquanto 2000, 2003 e 2008 apresentaram apenas uma publicação. Mais de metade dos artigos foram publicados (52%, 16/31) entre 2010 e 2016, representando um aumento em relação ao período 2003-2009. A Figura 12 também revela que não existe um padrão em termos do número de publicações. Ou seja, observando o número de artigos publicados por ano, não se pode conjeturar nenhuma tendência (aumento ou diminuição) de publicações nos próximos anos. Pode-se afirmar apenas que houve alguns períodos de aumento ou diminuição no número de publicações. A propósito, o que na verdade se pode observar é um pequeno número de publicações quando se considera o período 2010-2016.

Figura 12: Publicações por ano



Fonte: Elaborado pela autora.

Também foram identificados os autores que mais publicaram artigos neste campo de pesquisa. A Tabela 8 apresenta os autores que apresentam maior frequência de publicações encontradas como o principal autor nos estudos primários. Esses números podem servir para ilustrar quem tem, de fato, trabalhado em questões relativas à detecção de inconsistências em modelos de *design* multivisão. Destaca-se que Alexander Egyed e Alexander Reder são os autores que mais apareceram nos estudos primários, sendo responsáveis por produzir 26% (8/31) dos estudos. Em seguida, Christian Lange produziu três trabalhos importantes, nos quais o autor propõe uma classificação inteligível de inconsistências em modelos de *design* multivisão.

Tabela 8: Ranking quantidade de publicações por autor

Autor	Montante aprovado	Percentual
Alexander Egyed	4	13%
Alexander Reder	4	13%
C. Lange	3	10%
Lopez-Herrejon	2	6%
Outros	18	58%

Fonte: Elaborado pela autora.

Quanto à co-autoria, pode-se também indicar a presença de Alexander Egyed, aparecendo com co-autoria de 7 estudos selecionados, acrescentando participação em 11 artigos, demonstrando relevância de 35% na seleção do trabalho e dados coletados. Outro co-autor que vale a pena mencionar é Michel R. V. Chaudron, que participou em 16% dos estudos selecionados. Assim, estes dois autores têm contribuído fortemente para alavancar esta área de pesquisa, conforme apresentado na Tabela 9.

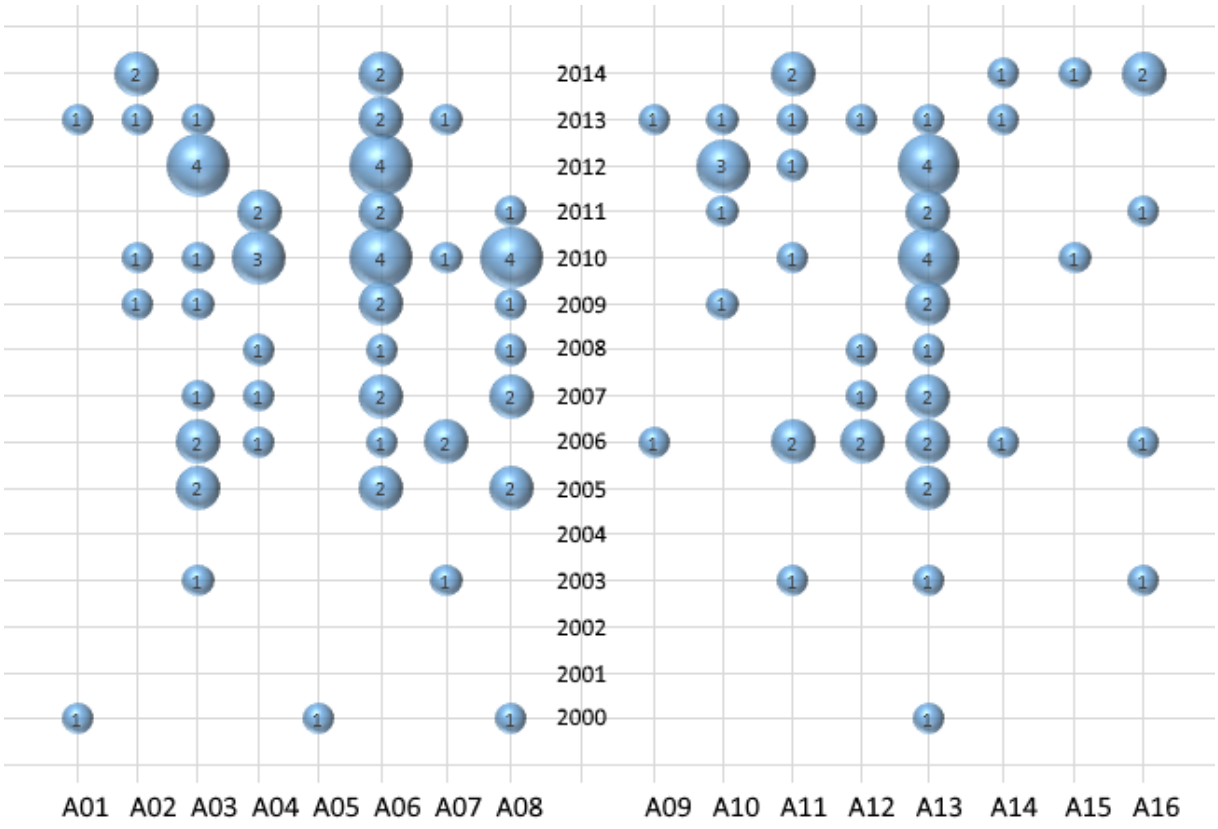
Tabela 9: Ranking quantidade de publicações por autor mais relevante

Autor	Montante aprovado	Percentual
Alexander Egyed	11	35%
M.R.V. Chaudron	5	16%
Xianhong Liu	2	6%
Outros	10	43%

Fonte: Elaborado pela autora.

Finalmente, a Figura 13 apresenta a distribuição dos estudos primários através de um intervalo (2000-2016). Este gráfico de bolhas representa pontos de dados como bolhas, e uma dimensão adicional dos dados é representada no tamanho das bolhas. O eixo-y do gráfico de bolhas ilustra a quantidade de estudos primários em um determinado ano, enquanto o eixo-x consiste em um conjunto de questões de pesquisa. Assim, este gráfico permitiu criar uma visão panorâmica sobre a literatura. Ou seja, este método apresenta um mapa e fornece uma visão geral de quantos trabalhos foram publicados considerando uma área específica. No total, 16 aspectos das questões de pesquisa (A1-A16) foram definidas para agrupar os estudos em função do ano publicado. Podemos observar que as os aspectos das questões A06 e A13 foram os mais explorados ao longo dos anos. Além disso, há um foco importante na proposição de técnicas de detecção de inconsistência, levando em conta tanto diagramas de classes quanto de sequência (A03).

Figura 13: Publicações por ano



A01 Avaliação de outros Diagramas
A02 Avaliação de Diagramas de Classe
A03 Avaliação de Diagramas de Classe e de Sequência
A04 Avaliação de Diagramas de Classe, de Sequência e de Estado
A05 Avaliação de Inconsistências Semânticas
A06 Avaliação de Inconsistências Sintáticas
A07 Avaliação de Inconsistências Semânticas e Sintáticas
A08 Detecção Automática de Inconsistências
A09 Detecção Manual de Inconsistências
A10 Detecção de Inconsistências através de Regras
A11 Detecção de Inconsistências através de Métodos Mistos
A12 Avaliação do Impacto de Inconsistências
A13 Detecção de Inconsistências por Analistas
A14 Detecção de Inconsistências por Desenvolvedores
A15 Detecção de Inconsistências por Engenheiros
A16 Utiliza Métricas para detecção de Inconsistências

Fonte: Elaborado pela autora.

Os resultados mostram que a detecção automática de inconsistências sintáticas através do desenvolvimento de aplicações (A08) utilizando configurações das regras de modelagem UML (A10) é mais adotada pela academia ao longo dos últimos anos. Pode-se também observar que a maioria dos estudos se concentra na detecção de inconsistências pelo analista de software, ainda na fase de analise do software, antes de seu desenvolvimento, não incidindo ambos os impactos que os desenvolvedores podem tirar o tempo de desenvolvimento. Além disso, há um grande foco de métodos de detecções de inconsistências utilizando a combinação de Diagrama de Classes e Diagrama de Sequência (A03), que podem ou não envolver o também o diagrama de estado (A04). No entanto, o Diagrama de Classes tem sido o foco das publicações mais recentes, estando sempre presente nos estudos selecionados para este estudo de mapeamento sistemático.

3.4.1 Avaliação geral

Pode-se observar que nem todos os trabalhos estudados relacionam diagramas UML multivisão, ou utilizam métricas para detecção de inconsistências, ou mesmo, consideram inconsistências sintáticas e semânticas. Ainda assim, alguns estudos compartilham técnicas e práticas afins, de modo que algumas pesquisas são claramente complementares. A Tabela 10, suportada pelo Apêndice A, apresenta um comparativo das características que se apresentam nos trabalhos estudados, baseando-se nas respostas às questões de pesquisa apresentadas na Tabela 3.

Tabela 10: Comparativo dos trabalhos relacionados

Avaliados	Tipo de Diagrama				Inconsistências		Tipo detecção			Analise de Impactos	Fase do Projeto			Utilização de Métricas
	Classe	Sequência	Estado	Outros	Semântica	Sintática	Automática	Manual	Mista		Análise	Desenvolvimento	Manutenção	
S01	+	-	-	-	+	-	+	-	-	-	+	-	-	-
S02	+	+	-	-	+	+	~	-	+	-	+	-	-	+
S03	+	+	-	-	-	+	+	-	-	-	+	-	-	-
S04	+	+	-	-	-	+	+	-	-	-	+	-	-	-
S05	+	+	-	-	+	+	-	+	-	+	-	+	-	-
S06	+	+	+	-	-	+	+	-	+	-	+	-	-	-
S07	+	+	-	-	+	+	~	-	+	+	+	-	-	+
S08	+	+	+	-	-	+	+	-	-	+	+	-	-	-
S09	+	+	-	-	-	+	+	-	-	-	+	-	-	-
S10	+	+	+	-	-	+	+	-	-	+	+	-	-	-
S11	+	+	-	-	-	+	~	-	+	-	+	-	-	-
S12	+	-	-	-	-	+	+	-	-	-	+	-	-	-
S13	+	+	-	-	-	+	+	-	-	-	+	-	-	-
S14	+	+	+	-	-	+	+	-	-	-	+	-	-	-
S15	+	+	+	-	+	+	~	-	+	-	-	-	+	-
S16	+	-	-	-	-	+	+	-	-	-	+	-	-	-
S17	+	+	+	-	-	+	+	-	-	-	+	-	-	-
S18	+	+	+	-	-	+	+	-	-	-	+	-	-	-
S19	+	+	+	-	-	+	~	-	+	-	+	-	-	+
S20	+	+	-	-	-	+	~	-	+	-	+	-	-	-
S21	+	+	-	-	-	+	~	-	+	-	+	-	-	-
S22	+	+	-	-	-	+	~	-	+	-	+	-	-	-
S23	+	+	-	-	-	+	~	-	+	-	+	-	-	-
S24	+	-	-	-	-	+	~	-	+	-	+	-	-	-
S25	+	+	-	-	+	+	-	+	-	+	-	+	-	-
S26	+	+	+	+	-	+	~	-	+	-	+	-	-	-
S27	+	-	-	-	-	+	~	-	+	-	-	-	+	+
S28	+	-	-	-	-	+	~	-	+	-	-	+	-	+
S29	+	+	-	-	-	+	+	-	-	-	+	-	-	-
S30	+	-	-	-	-	+	-	-	+	-	+	-	-	+
S31	+	-	-	-	-	+	-	-	-	-	+	-	-	-
+	31	23	8	1	7	30	14	2	14	6	26	3	2	6
~	0	0	0	0	0	0	14	0	2	0	0	0	0	0
-	0	8	23	30	24	1	3	29	15	25	5	28	29	25

Notas: (+) atente, (-) não atende, (~) atende parcialmente

Fonte: Elaborado pela autora.

Ao analisar o quadro comparativo dos trabalhos relacionados, nota-se que não há um estudo que represente uma visão completa do objetivo de pesquisa deste trabalho, abordando todos os itens avaliados. Verifica-se que alguns dos estudos são baseados no uso de métricas para identificar inconsistências, mas a maioria também apresenta um método automatizado para detecção de inconsistências sintáticas com base apenas nas regras formais da própria linguagem de modelagem UML. Isto pode ser explicado por dois motivos. Inicialmente, é mais simples de ser baseado diretamente na configuração das regras básicas de modelagem

UML na configuração de software para a detecção automática de inconsistências. Depois disso, o foco em inconsistências sintáticas também é mais simples porque não há necessidade de comparações semânticas entre diferentes diagramas UML.

3.4.2 Oportunidades de Pesquisa

Após análise dos estudos apresentados, vislumbram-se oportunidades de pesquisa para a área de detecção de inconsistências em modelos UML multivisão. Essas oportunidades levam em consideração as características e fragilidades identificadas nos trabalhos relacionados, bem como novas formas de aplicação das estratégias apresentadas, conforme segue:

- **Conhecimento empírico sobre detecção de inconsistências em modelos UML:** realizar experimentos sobre detecção de inconsistências em modelos UML multivisão, permitindo mensurar o esforço que analistas e desenvolvedores realizam para esta atividade. Identificar, também, as principais inconsistências com maior impacto na qualidade e produtividade do desenvolvimento de software, na qual os participantes identificam os modelos com inconsistências com o objetivo de identificar o esforço que modelos de má qualidade causam no desenvolvimento de software, mas também identificar os impactos de tais modelos na qualidade do software desenvolvido, conforme os diferentes tipos de inconsistências possíveis em diagramas.
- **Técnicas para detecção de inconsistências em modelos UML:** elaborar novas técnicas, capazes de suprir as demandas levantadas a partir da análise da literatura atual, envolvendo também os aspectos de inconsistências sintáticas, semânticas e estruturais em modelos UML, considerando também os impactos de cada um dos tipos de inconsistências possíveis nos diagramas de software, de forma a priorizar as atividades de detecção, permitindo que estas técnicas sejam eficazes e sirvam de referencial para o desenvolvimento de novas metodologias para detecção de inconsistências e sua viabilidade quanto à aplicação na indústria.
- **Implementação de uma aplicação que envolva as técnicas para detecção de inconsistências em modelos UML multivisão utilizando métricas:** desenvolver uma aplicação que, utilizando métricas para detecção de inconsistências em diagramas, seja capaz de facilitar o processo de detecção de inconsistências por parte dos analistas e desenvolvedores de software, de forma completa, envolvendo inconsistências sintáticas, semânticas e estruturais, minimizando assim, os impactos da utilização destes modelos na qualidade e na produtividade do desenvolvimento de software.

4 TÉCNICA

Considerando as oportunidades de pesquisa resultantes dos estudos realizados, foram definidas duas estratégias para atender à questão de pesquisa deste trabalho, referente à detecção de inconsistências em modelos UML a partir de métricas. Inicialmente, foi elaborado um catálogo de inconsistências, com o objetivo de identificar um conjunto de inconsistências sintáticas que podem ocorrer em modelos UML multivisão e como elas podem ser combinadas. A identificação destes tipos de inconsistências permitirá a definição de métricas para detecção de cada uma delas, o que resulta na segunda estratégia, que se refere a detecção de inconsistências a partir de métricas.

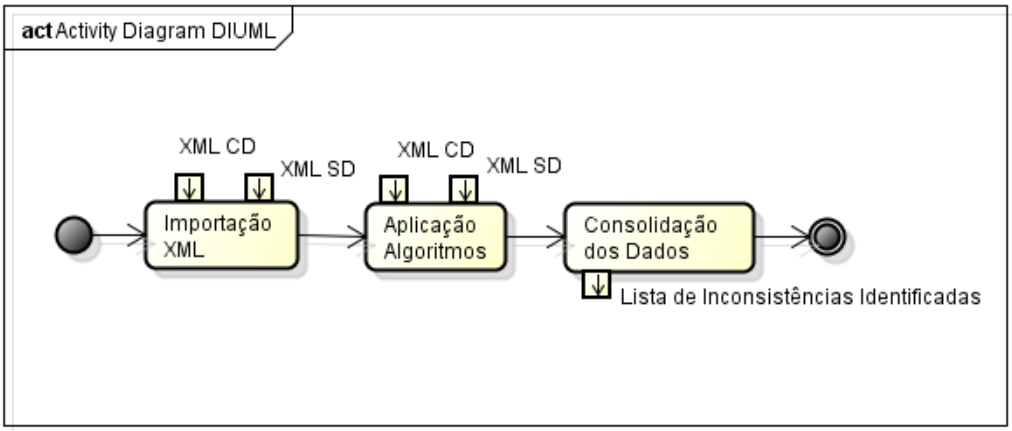
Sendo assim, neste capítulo serão apresentadas as duas estratégias. A seção 4.1 apresenta uma visão geral sobre as técnicas desenvolvidas para a implementação da aplicação, que é objeto de estudo deste trabalho, na seção 4.2 é realizada a apresentação do catálogo de padrões de inconsistências e na seção 4.3, são apresentados os algoritmos e métricas utilizados para a detecção de cada tipo de inconsistência. Por fim, a seção 4.4, apresenta a estrutura básica da aplicação que será desenvolvida para exemplificar a aplicação do modelo, apresentado os requisitos, e modelagem básica da aplicação que utilizará os algoritmos e métricas definidas para auxiliar desenvolvedores e analistas de sistemas na detecção de inconsistências em modelos UML.

4.1 Visão Geral

Para a aplicação da técnica baseada em métricas para detecção de inconsistências em modelos UML multivisão, optou-se pelo desenvolvimento de uma aplicação que se utilizasse das heurísticas apresentadas neste Capítulo, seção 4.3. Para a implementação destas heurísticas, inicialmente foi necessário o desenvolvimento de um Catálogo de Padrões de Inconsistências, também apresentado neste Capítulo, seção 4.2.

A Figura 14 demonstra como os algoritmos destas heurísticas e métricas definidas neste estudo estão aplicadas no desenvolvimento da aplicação. Inicialmente é necessária a importação dos arquivos XML referentes aos Diagramas de Classe e Sequência do modelo UML a ser analisado; em seguida é realizada a aplicação dos algoritmos baseados nas heurísticas de detecção de inconsistências, validando cada tipo de inconsistência levantada; e por fim, é realizada a consolidação e apresentação dos dados, através de uma lista de inconsistências detectadas nos modelos multivisão.

Figura 14: Estrutura Básica da Aplicação



Fonte: Elaborado pela autora.

Portanto, a utilização dos algoritmos e métricas aqui propostos servem como complementos para o desenvolvimento da aplicação que será desenvolvida para o processo de detecção de inconsistências em modelos UML multivisão.

4.2 Catálogo de padrões de inconsistências

Para a elaboração do Catálogo de Padrões de Inconsistências, foram elencados tipos de inconsistências que podem ocorrer em modelos UML que envolvam Diagramas de Classes e Diagramas de Sequência, assim como, a apresentação de algumas possíveis combinações de Inconsistências que também foram utilizadas para a validação desta Dissertação. Cada tipo de inconsistências apresentado no Catálogo de Padrões de Inconsistências conta com uma nomenclatura, uma abreviação e uma descrição, assim como proposto por Chaudron (2006).

Os tipos de inconsistências apresentados a seguir podem ser observados simplificadamente na Tabela 11. O catálogo de inconsistências em modelos UML foi desenvolvido com base no modelo proposto no estudo de Lange, (2006) e na exploração das regras de modelagem da linguagem UML.

Tabela 11: Relação de Inconsistências

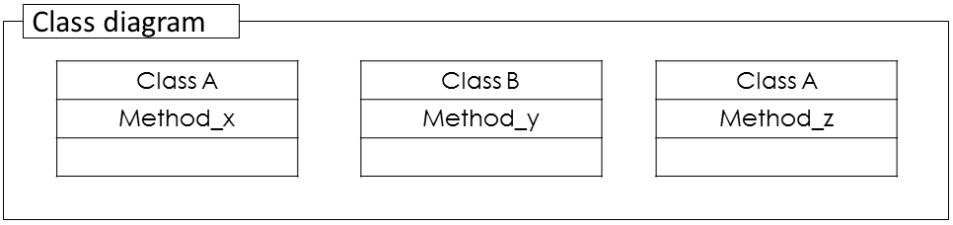
Sigla	Descrição
Cm	Várias definições de Classes com mesmo nome
Om	Várias definições de Objetos com mesmo nome
CnSD	Classe não instanciada no SD
CnCD	Objeto sem Classe no CD
ED	Mensagem na direção Errada
EnN	Mensagem sem Nome
EcM	Mensagem sem Método
CaSD	Classe abstrata instanciada no SD

Fonte: Elaborado pela autora.

4.2.1 Lista de inconsistências

Várias definições de classes com mesmo nome (Cm). Quando uma classe é instanciada mais de uma vez com o mesmo nome, no mesmo ou em diferentes diagramas, em um único modelo UML. A Figura 15 apresenta a definição da Classe A duas vezes, com diferentes métodos. Logo, pode-se identificar a ocorrência deste tipo de inconsistência. Este é um tipo de inconsistência que é restringido pelo próprio metamodelo da UML.

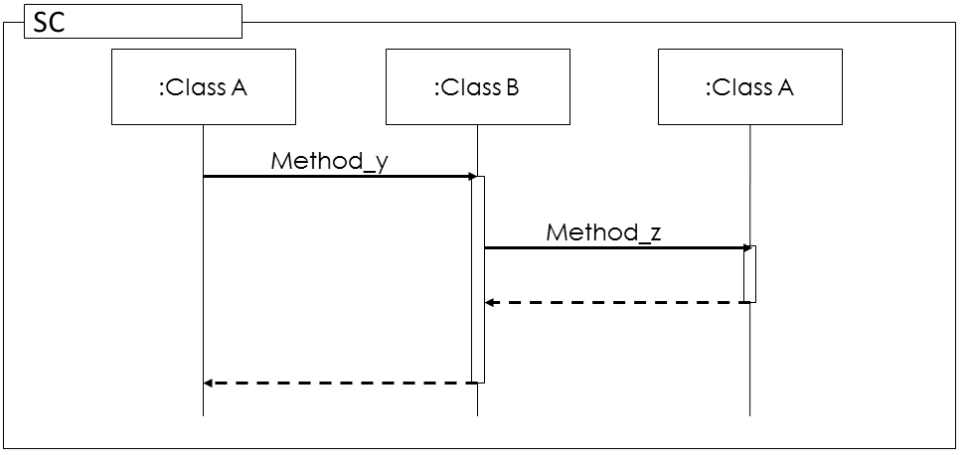
Figura 15: Tipo de inconsistência Cm



Fonte: Elaborado pela autora.

Várias definições de objetos com mesmo nome (Om). Esta inconsistência ocorre quando há a definição de mais de um objeto com mesmo nome no diagrama de sequência da UML. Assim como na inconsistência anterior, a *Om* acaba por não respeitar uma regra definida no metamodelo da UML (BOOCH et al., 2008) que define que em uma *namespace* não pode existir mais de um elemento com mesmo nome. A Figura 16 apresenta um exemplo desta inconsistência. Neste exemplo, tem-se dois objetos :*Class A* com o mesmo nome.

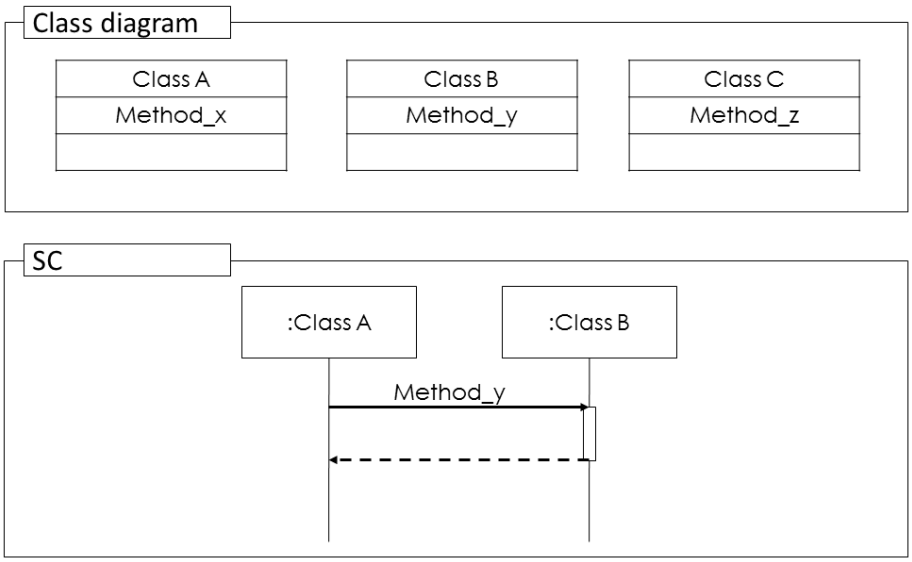
Figura 16: Tipo de inconsistência Om



Fonte: Elaborado pela autora.

Classe não instanciada em SD (CnSD). Esta inconsistência acontece quando não existir um objeto instanciado no Diagrama de Sequência para todas as classes presentes no Diagrama de Classes. Na Figura 17, por exemplo, a Classe C não aparece representada como um objeto no Diagrama de Sequência. Logo, pode-se afirmar que existe uma inconsistência do tipo**.**

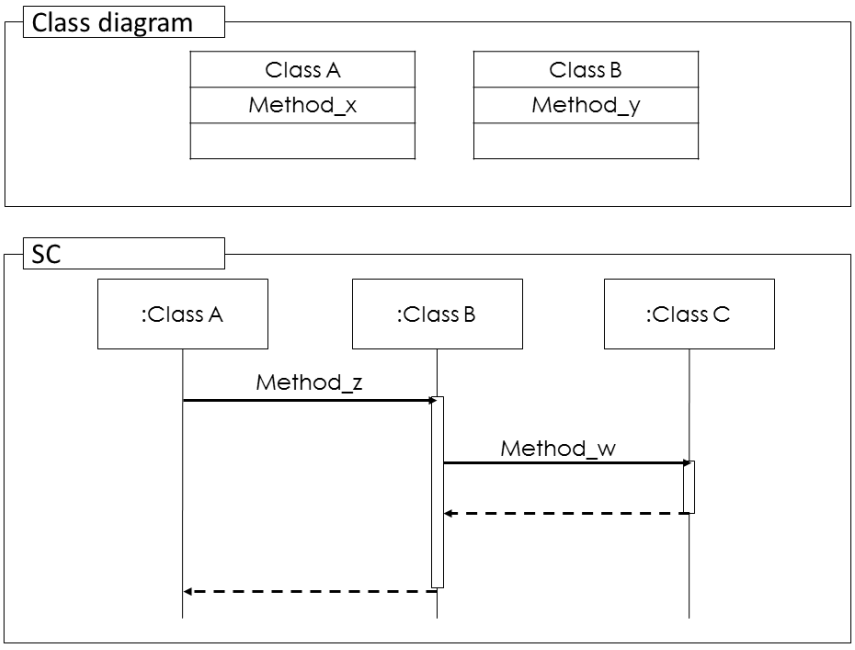
Figura 17: Tipo de inconsistência CnSD



Fonte: Elaborado pela autora.

Objeto sem Classe em CD (CnCD). Contrário à CnSD, esta inconsistência ocorre quando para um objeto instanciado no Diagrama de Sequência não exista uma classe correspondente no Diagrama de Classes. Como apresentado na Figura 18, na qual não existe uma classe C no Diagrama de Classes para o objeto C do diagrama de Sequência, resulta neste tipo de inconsistência.

Figura 18: Tipo de inconsistência CnCD

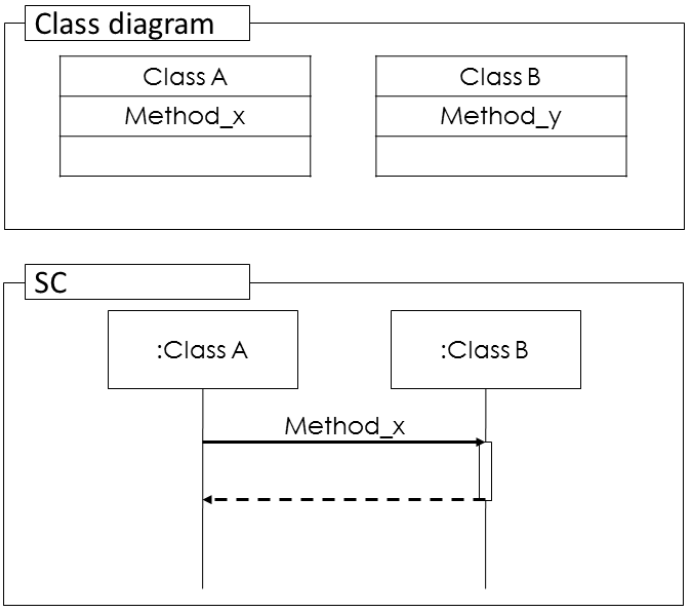


Fonte: Elaborado pela autora.

Objeto sem Classe em CD (CnCD). Um tipo simples de inconsistência que pode ser observado em grandes ou pequenos modelos é o tipo ED. Este tipo de inconsistência ocorre quando, no Diagrama se Sequência, um objeto chama uma mensagem que na realidade corresponde ao método da própria classe que originou o objeto. A Figura 19 exemplifica este

tipo de inconsistência quando, no Diagrama de Sequência o objeto A chama mensagem X do objeto B, porém, a mensagem X corresponde ao método X da própria Classe A do Diagrama de Classes. Este é um dos casos em que o nome da mensagem não faz correspondência ao respectivo método.

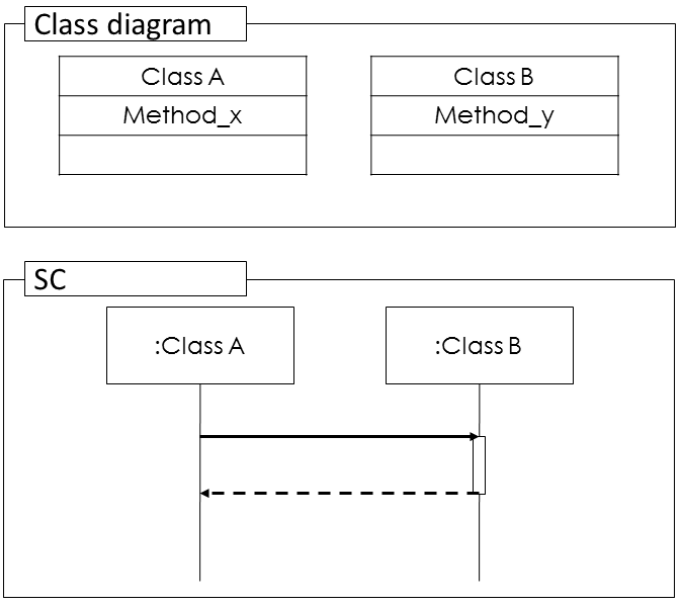
Figura 19: Tipo de inconsistência ED



Fonte: Elaborado pela autora.

Mensagem sem Nome (EnN). Quando se elabora um Diagrama de Sequência, deve-se atentar para que as mensagens trocadas entre os objetos referenciem-se aos métodos pertencentes às classes referentes no Diagrama de Classes. Quando as setas que representam as mensagens trocadas entre os objetos, assim como demonstrado na Figura 20, não apresentam um nome que descreve a mensagem referente ao método do Diagrama de Classes, ocorre o tipo de inconsistência conhecido como EnN.

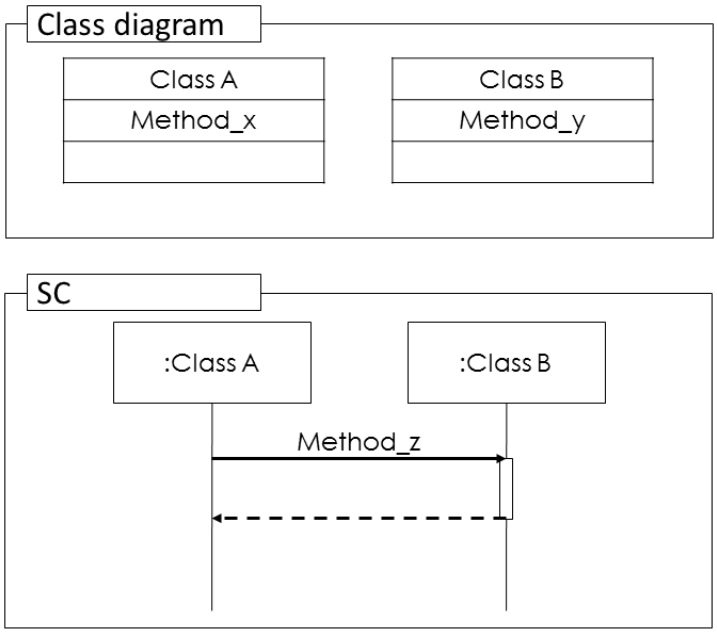
Figura 20: Tipo de inconsistência EnN



Fonte: Elaborado pela autora.

Mensagem sem Método (EcM). Este tipo de inconsistência trata dos casos em que o nome da mensagem no diagrama de sequência não faz correspondência ao respectivo método no diagrama de classes. Quando uma mensagem a partir de um objeto para outro deveria significar que o primeiro objeto chama um método que é fornecido pelo segundo objeto, ou seja, o nome da mensagem deveria corresponder ao nome do método que deveria ser chamado, mas não é possível realizar a correspondência entre a mensagem e o método, como apresentado na Figura 21 com a representação da mensagem Z, que não existe em nenhuma das classes do Diagrama de Classes.

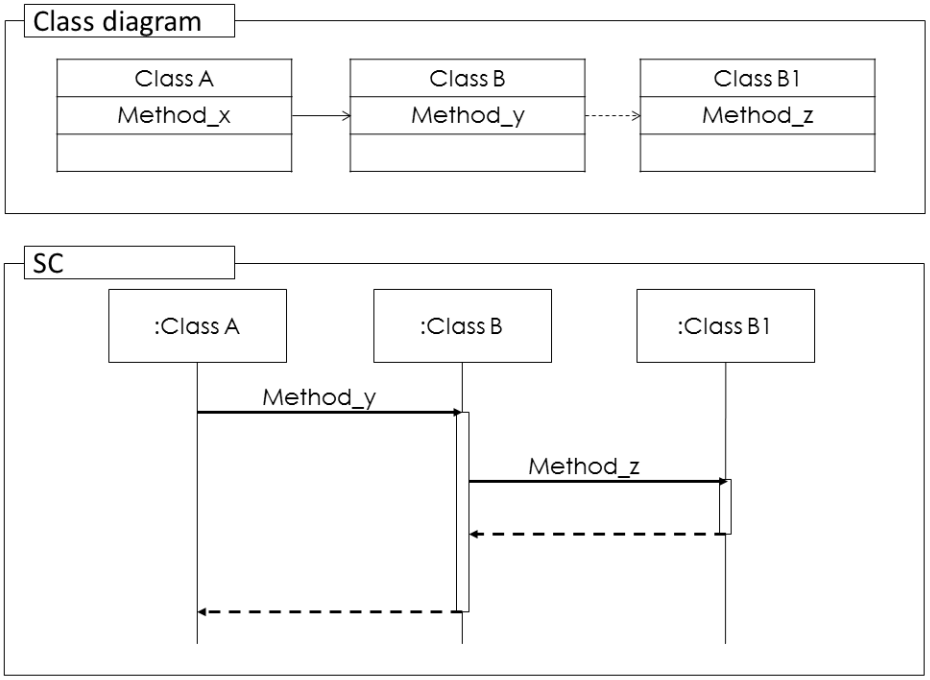
Figura 21: Tipo de inconsistência EcM



Fonte: Elaborado pela autora.

Classe abstrata instanciada no diagrama de sequência (CaSD). Um classe abstrata não pode ser instanciada, logo não é possível produzir objetos a partir delas. Dessa forma, as classes abstratas do Diagrama de Classes não podem estar presentes no Diagrama de Sequência, visto que isso significaria que as mesmas foram instanciadas. A Figura 22 apresenta um exemplo deste tipo de inconsistência. A presença de classes abstratas no Diagrama de Sequência é um tipo de inconsistência que pode afetar diretamente a intepretação do modelo, principalmente com relação a criação dos códigos para as classes do software, afetando também sua modularidade e reutilização.

Figura 22: Tipo de inconsistência CaSD

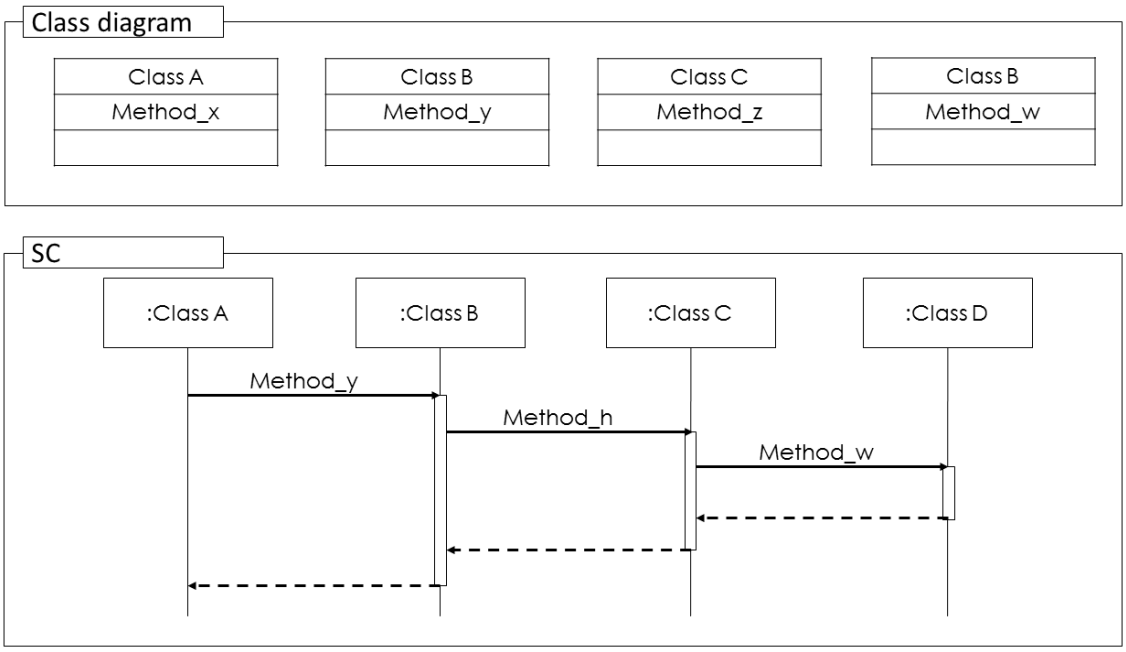


Fonte: Elaborado pela autora.

4.2.2 Inconsistências compostas

Normalmente as inconsistências, quando ocorrem em modelos UML, não aparecem de forma individual. A combinação de diferentes tipos de inconsistências é muito comum, pois ao ocorrer um tipo de inconsistência, outro tipo pode aparecer como consequência da inconsistência inicial. Sendo assim, a Figura 23 apresenta uma combinação dos tipos de inconsistências Cm, EcM e CnCD. Pode-se observar a inconsistência do tipo Cm na ocorrência da classe B duas vezes, com métodos diferentes, no Diagrama de Sequência. A inconsistência do tipo EcM aparece na implementação da mensagem H e W, que ao correspondem aos métodos das respectivas classes. Além disso, o tipo de inconsistência CnCD é representada através da implementação do objeto D no Diagrama de Sequências, mas que não apresenta Classe referente a ele, resultando na inconsistência previamente descrita referente a mensagem H.

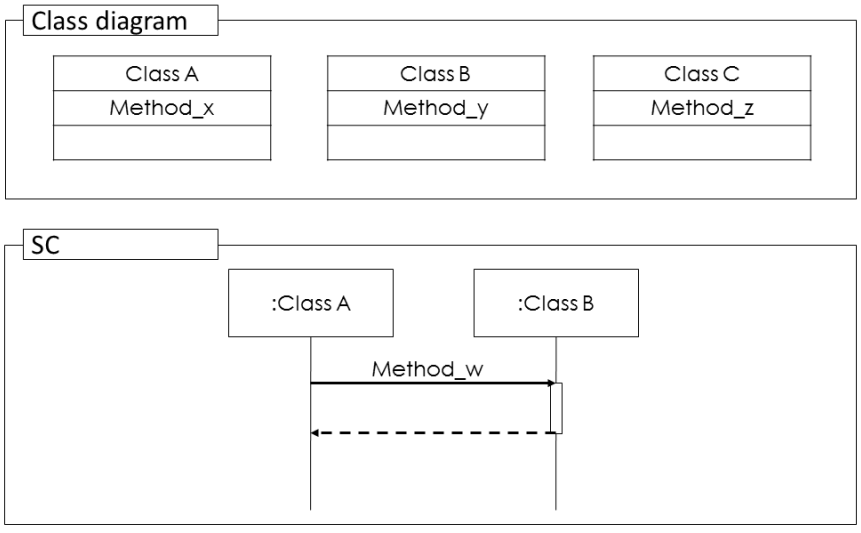
Figura 23: Cm, EcM e CnCD



Fonte: Elaborado pela autora.

As inconsistências apresentadas estão diretamente relacionadas entre si, pois, a partir da ocorrência de um tipo de inconsistência, os demais acabam ocorrendo como consequência. Outro tipo de combinação de inconsistências que se pode elencar é a combinação entre CnSD e EcM. Como se pode verificar na Figura 24, a Classe C não possui um Objeto respectivo no Diagrama de Sequência, caracterizando a inconsistência de CnSD. Além disso, foi implementada uma mensagem W, que sequer foi representada como um método de alguma classe no Diagrama de Classes.

Figura 24: CnSD e EcM

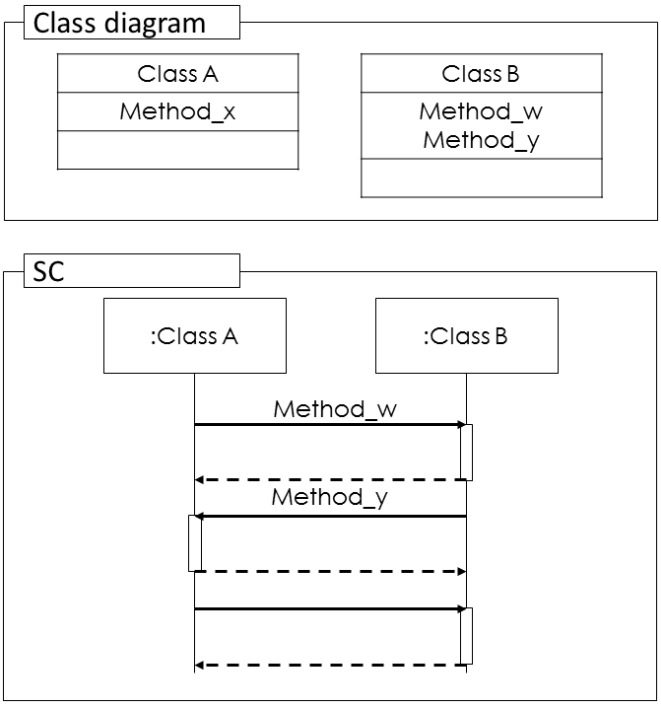


Fonte: Elaborado pela autora.

O resultado deste tipo de inconsistências é um modelo que apresenta classes não instanciadas e, consequentemente, métodos não aplicados como mensagens para concluir a intepretação do modelo UML.

Inconsistências teoricamente simples, como ED e EnN, também podem ocorrer em modelos UML e acabar passando despercebidas na interpretação dos diagramas. Sendo assim, na Figura 25, pode-se observar a ocorrência de ED quando o objeto B chama a mensagem Y do objeto A, sendo que a mensagem Y refere-se a um método da própria classe B. Além disso, a ocorrência de uma EnN é representada quando, na mensagem final, não há a definição do nome da mensagem.

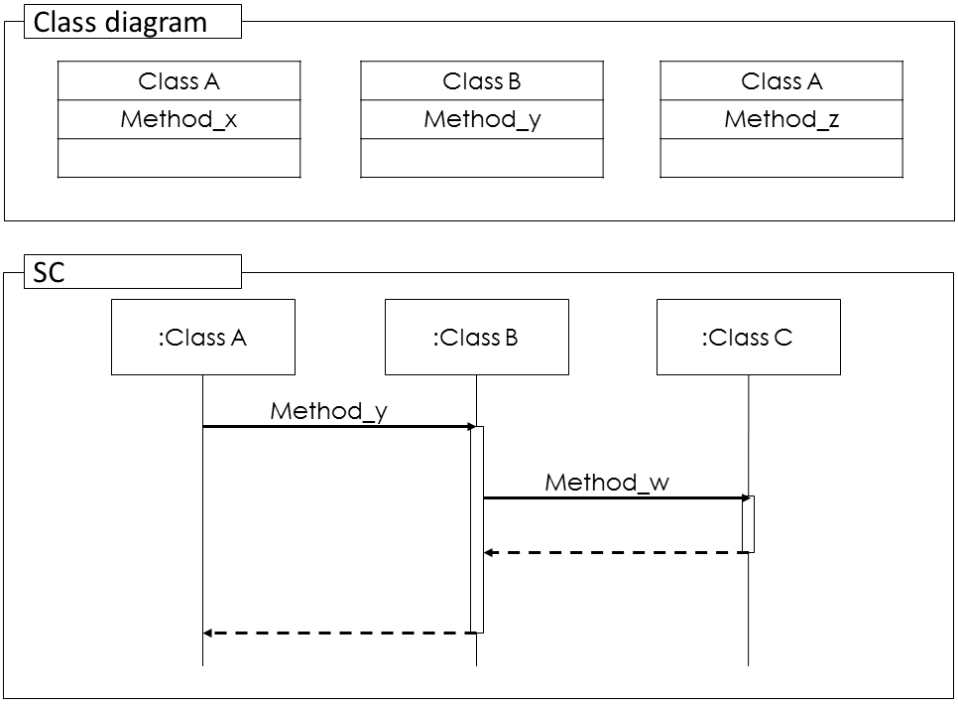
Figura 25: ED e EnN



Fonte: Elaborado pela autora.

Várias definições de classes com nomes iguais também podem ocorrem em composição com Mensagem sem Método, como apresentado na Figura 26. Tem-se a Classe A instanciada duas vezes no Diagrama de Classes, mas com métodos diferentes. Consequentemente, esta inconsistência ocasiona uma mensagem do Diagrama de Sequência sem um método correspondente no Diagrama de Classes, pois o método W não existe, assim como, a classe C também não existe.

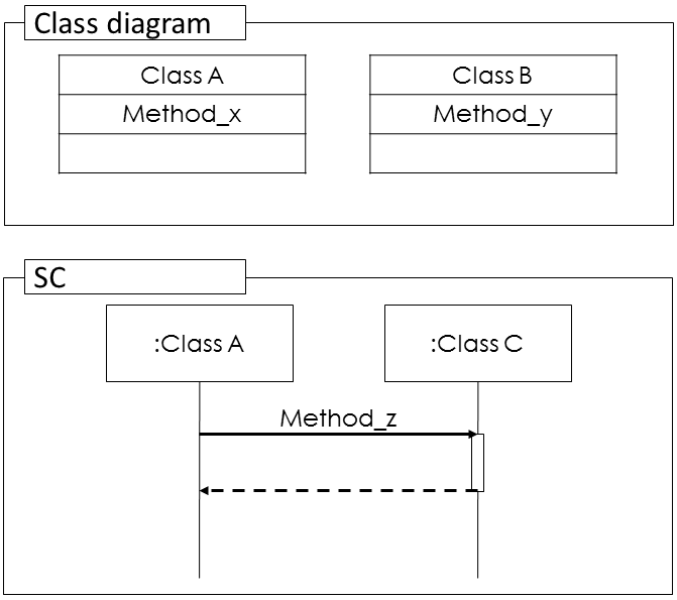
Figura 26: Cm e EcM



Fonte: Elaborado pela autora.

Assim como no exemplo anterior, na Figura 27 pode-se observar como um Objeto não tem nenhuma Classe em CD resulta em uma inconsistência também do tipo Mensagem sem Método. O simples fato de existirem uma classe A e uma B no Diagrama de Classes e um objeto A e um objeto B, resulta no fato de não existir um método Z para corresponder a esta mensagem ou ainda, demonstra a não implementação do método Y.

Figura 27: CnCD e EcM



Fonte: Elaborado pela autora.

Portanto, pode-se concluir que as inconsistências normalmente não ocorrem de modo individual, sendo que muitas vezes, uma inconsistência acaba sendo consequência da ocorrência de outra. Em grandes modelos, nos quais muitos diagramas são inter-relacionados, a detecção destas inconsistências acaba sendo muito onerosa, ou ainda, não eficaz.

4.3 Heurísticas para detecção de inconsistências

A utilização de heurísticas para detecção de inconsistências, a partir da detecção e utilização de métricas é objeto central deste estudo. Sendo assim, nesta seção são apresentados os algoritmos que serão utilizados no desenvolvimento da aplicação que implementará uma metodologia de detecção de inconsistências em modelos UML, considerando, inicialmente, Diagramas de Classes e Diagramas de Sequência, seguindo as inconsistências apresentadas no Catálogo de Inconsistências referente ao item 4.2 deste trabalho.

Optou-se por definição de heurísticas, sem considerar ontologias pra a detecção de inconsistências semânticas, pois, embora ontologias possam ser geralmente utilizadas na detecção de inconsistências semânticas, elas precisam ser atualizadas frequentemente, inviabilizando seu uso em ambientes reais de desenvolvimento de software. Em (FARIAS, 2008) e (FARIAS 2009), os autores mostram que ontologias podem ser utilizadas com o propósito de integração de modelos, visando evitar a produção de inconsistências. Por outro lado, eles também destacam a dificuldade de manter tais ontologias.

4.3.1 Cm: Várias definições de Classes com mesmo nome

A detecção de inconsistências do tipo **Cm** envolve o mapeamento do Diagrama de Classes, no qual, busca-se identificar quantas vezes cada classe é detectada no diagrama. Caso o contador de Classes encontre um valor diferente de 1 para cada instância de classe, este tipo de inconsistência está presente. O algoritmo utilizado para a detecção deste tipo de inconsistência é:

Algoritmo 1: Detecção de Inconsistências do Tipo Cm

Entrada: XML Diagrama de Classes

Saída: Lista de Inconsistência do Tipo Cm

```
1:    Para x=0 até x<=qtdeClasses
2:        qtdeClasses =count (intancia.nomeClasse)
3:        resultado ← qtdeOcorrencia
4:        Se resultado!=1
5:            inconsistencia=true
6:        Fim se
7:        x++
```

8: **Fim para**

Para a aplicação deste algoritmo, são necessárias as métricas Quantidade de Classes (qtdeClasses), Quantidade de Ocorrências (qtdeOcorrencia) e um contador para a métrica Nome da Classe (*intancia*.nomeClasse), pois, caso o resultado da Quantidade de Ocorrências de uma instância de classe seja diferente de 1, o tipo de inconsistência Cm é encontrado.

4.3.2 Om: Várias definições de Objetos com mesmo nome

Com relação à detecção de inconsistências do tipo **Om**, deve ser realizado o mapeamento do Diagrama de Sequência, buscando identificar quantas vezes cada objeto é instanciado. Caso o contador de Objetos apresente um valor diferente de 1 para cada instância de objeto, esse tipo de inconsistência é detectado. Para este tipo de detecção, é utilizado o seguinte algoritmo:

Algoritmo 2: Detecção de Inconsistências do Tipo Om

Entrada: XML Diagrama de Sequência

Saída: Lista de Inconsistência do Tipo Om

```
1:  Para x=0 até x<=qtdeObjetos
2:      qtdeOcorrencia=count (intancia.nomeObjeto)
3:      resultado ← qtdeOcorrencia
4:      Se resultado!=1
5:          inconsistencia=true
6:      Fim se
7:      x++
8:  Fim para
```

Para a aplicação deste algoritmo, são necessárias as métricas Quantidade de Objetos (qtdeObjetos), Quantidade de Ocorrências (qtdeOcorrencia) e um contador para a métrica Nome do Objeto (*intancia*.nomeObjeto), pois, caso o resultado da Quantidade de Ocorrências de uma instância de classe seja diferente de 1, o tipo de inconsistência Om é encontrado.

4.3.3 CnSD: Classe não instanciada no Diagrama de Sequência

Para detectar inconsistências do tipo **CnSD** é necessário o mapeamento do Diagrama de Classes e do Diagrama de Sequência, realizado uma comparação das classes e objetos, respectivamente. Mapeando o Diagrama de Classes e o de Sequência, busca-se identificar se para cada instância de Classe há um Objeto referente. Caso o contador comparativo entre as instâncias seja diferente de 1, esta inconsistência é identificada. Para a detecção deste tipo de inconsistência é utilizado o seguinte algoritmo:

Algoritmo 3: Detecção de Inconsistências do Tipo CnSD

Entrada: XML Diagrama de Classes e Diagrama de Sequência

Saída: Lista de Inconsistência do Tipo CnSD

```
1:    Para x=0 até x<=qtdeClasses
2:        Para i=0 até i<=qtdeObjetos
3:            Se intancia.nomeClasse=intancia.nomeObjeto
4:                qtdeOcorrencia= qtdeOcorrencia+1
5:            Fim se
6:            i++
7:        Fim Para
8:        resultado ← qtdeOcorrencia
9:        Se resultado!=1
10:           inconsistencia=true
11:       Fim se
12:       x++
13:   Fim para
```

Para a aplicação deste algoritmo, são necessárias as métricas Quantidade de Objetos (qtdeObjetos), Quantidade de Classes (qtdeClasses), Quantidade de Ocorrências (qtdeOcorrencia) e um contador para a comparação entre as métricas Nome do Objeto (*intancia*.nomeObjeto) e Nome da Classe (*intancia*.nomeClasse), pois, caso o resultado da Quantidade de Ocorrências da comparação entre uma instância de classe e uma instancia de objeto seja diferente de 1, o tipo de inconsistência CnSD é encontrado.

4.3.4 CnCD: Objeto sem Classe no Diagrama de Classe

Com relação à detecção do tipo de inconsistência **CnCD**, deve-se realizar o mapeamento do Diagrama de Classes e do Diagrama de Sequência. É necessário identificar se para cada objeto do diagrama de sequência há uma classe respectiva no Diagrama de Classes. Portando, se o contador de cada instância de objetos, relacionado ao contador de classe apresentar um resultado diferente de 1, esta inconsistência pode ser detectada. Sendo assim, o respectivo algoritmo para detecção deste tipo de inconsistência é:

Algoritmo 4: Detecção de Inconsistências do Tipo CnCD

Entrada: XML Diagrama de Classes e Diagrama de Sequência

Saída: Lista de Inconsistência do Tipo CnCD

```
1:    Para x=0 até x<=qtdeObjetos
2:        Para i=0 até i<=qtdeClasses
```

```
3:              Se intancia.nomeObjeto=intancia.nomeClasse
4:                  qtdeOcorrencia= qtdeOcorrencia+1
5:              Fim se
6:              i++
7:          Fim Para
8:          resultado ← qtdeOcorrencia
9:          Se resultado!=1
10:             inconsistencia=true
11:         Fim se
12:         x++
13:     Fim para
```

Para a aplicação deste algoritmo, são necessárias as métricas Quantidade de Objetos (qtdeObjetos), Quantidade de Classes (qtdeClasses), Quantidade de Ocorrências (qtdeOcorrencia) e um contador para a comparação entre as métricas Nome do Objeto (*intancia*.nomeObjeto) e Nome da Classe (*intancia*.nomeClasse), pois, caso o resultado da Quantidade de Ocorrências da comparação entre uma instância de classe e uma instancia de objeto seja diferente de 1, o tipo de inconsistência CnCD é encontrado.

4.3.5 ED: Mensagem na direção Errada

Para a detecção de inconsistências do tipo **ED**, deve ser realizado o mapeamento do Diagrama de Classes e do Diagrama de Sequência, com o objetivo de identificar quais métodos existem em cada classe e quais mensagens são trocadas entre quais objetos. Caso os pares de combinações referentes às instancias de mensagens e métodos não representem as respectivas combinações, resultando em 0, este tipo de inconsistência é detectada. Sendo assim, para a detecção desta inconsistência, utiliza-se o seguinte algoritmo:

Algoritmo 5: Detecção de Inconsistências do Tipo ED

Entrada: XML Diagrama de Classes e Diagrama de Sequência

Saída: Lista de Inconsistência do Tipo ED

```
1:  Para x=0 até x<=qtdeClasses
2:      Para i=0 até i<=qtdeObjetos
3:          var1[] = instancia.nomeClasse and instancia.nomeMetodo[]
4:          var2[] = instancia.nomeObjeto and instancia.nomeMensagem[]
5:          Para z=0 até z<=var1[z]
6:              Para w=0 até w<=var2[w]
7:                  Se var1 =! var2
```

```
8:                                  inconsistencia=true
9:                              Fim se
10:                             w++
11:                         Fim para
12:                         z++
13:                 Fim para
14:                 i++
15:         Fim Para
16:         x++
17:     Fim para
```

Para a aplicação deste algoritmo, são necessárias as métricas Quantidade de Objetos (qtdeObjetos), Quantidade de Classes (qtdeClasses), Quantidade de Ocorrências (qtdeOcorrencia) e dois contadores para a comparação entre as métricas Nome do Objeto (*intancia*.nomeObjeto) com Nome da Classe (*intancia*.nomeClasse) e Nome da Mensagem (*intancia*.nomeMensagem) com Nome da Método (*intancia*.nomeMetodo) pois, caso o resultado da comparação entre as variáveis contadores seja diferente, o tipo de inconsistência ED é encontrado.

4.3.6 EnN: Mensagem sem Nome

A detecção de inconsistências do tipo **EnN** está diretamente relacionada ao mapeamento do Diagrama de Sequência, no qual, deve ser identificado se para cada mensagem há um nome relacionado. Sendo assim, o contador de nome para cada mensagem deve ser igual a 1, caso contrario, este tipo de inconsistência está presente. Portanto, para a detecção deste tipo de inconsistência deve ser utilizado o seguinte algoritmo:

Algoritmo 6: Detecção de Inconsistências do Tipo EnN

Entrada: XML Diagrama de Sequência

Saída: Lista de Inconsistência do Tipo EnN

```
1:      Para x=0 até x<=qtdeMensagens
2:          qtdeOcorrencia=count (intancia.nomeMensagem)
3:          resultado ← qtdeOcorrencia
4:          Se resultado!=1
5:              inconsistencia=true
6:          Fim se
7:          x++
8:      Fim para
```

Para a aplicação deste algoritmo, é necessária a métrica Quantidade de Mensagens (qtdeMensagens), e um contador para a métrica Nome da Mensagem (*intancia*.nomeMensagem) pois, caso o resultado da comparação entre o contador seja diferente de 1, o tipo de inconsistência EnN é encontrado.

4.3.7 EcM: Mensagem sem Método

Para a detecção do tipo de inconsistência do tipo **EcM**, é necessário o mapeamento do Diagrama de Classes e do Diagrama de Sequência. Em seguida, deve-se verificar se para cada mensagem do diagrama de sequencia há um método no Diagrama de Classes. O contador de mensagens e métodos deve resultar em 1 para cada instância analisada, caso contrario, este tipo de inconsistência é detectado. Para a detecção deste tipo de inconsistência, pode-se utilizar o seguinte algoritmo:

Algoritmo 7: Detecção de Inconsistências do Tipo EcM

Entrada: XML Diagrama de Classes e Diagrama de Sequência

Saída: Lista de Inconsistência do Tipo EcM

```
1:  Para x=0 até x<=qtdeMensagens
2:      Para i=0 até i<=qtdeMetodos
3:          Se intancia.nomeMensagem=intancia.nomeMetodo
4:              qtdeOcorrencia= qtdeOcorrencia+1
5:          Fim se
6:          i++
7:      Fim Para
8:      resultado ← qtdeOcorrencia
9:      Se resultado!=1
10:         inconsistencia=true
11:     Fim se
12:     x++
13: Fim para
```

Para a aplicação deste algoritmo, são necessárias as métricas Quantidade de Mensagens (qtdeMensagens), Quantidade de Métodos (qtdeMetodos), Quantidade de Ocorrências (qtdeOcorrencia) e um contador para a comparação entre as métricas Nome da Mensagem (*intancia*.nomeMensagem) com Nome da Método (*intancia*.nomeMetodo) pois, caso o resultado da Quantidade de Ocorrências da comparação entre uma instância de classe e uma instancia de objeto seja diferente de 1, o tipo de inconsistência EcM é encontrado..

4.3.8 CaSD: Classe abstrata instanciada no SD

Por fim, para detectar o último tipo de inconsistência apresentado no Catálogo de Inconsciências, o tipo **CaSD**, também é necessário o mapeamento dos Diagramas de Classe e de Sequência, com o objetivo de verificar se as classes abstratas do Diagrama de Classes foram instanciadas no Diagrama de Sequência. Sendo assim, se para o contador de classes abstratas comparadas a objetos resultar em algum valor diferente de 0, este tipo de inconsistência está presente. Para isso, utiliza-se o seguinte algoritmo, que objetiva identificar este tipo de inconsistência:

Algoritmo 8: Detecção de Inconsistências do Tipo CaSD

Entrada: XML Diagrama de Classes e Diagrama de Sequência

Saída: Lista de Inconsistência do Tipo CaSD

```
1:  Para x=0 até x<=qtdeClasses
2:      Se tipoClasse=abstrata
3:          Se intancia.nomeClasse=intancia.nomeObjeto
4:              qtdeOcorrencia= qtdeOcorrencia+1
5:          Fim se
6:      Fim Se
7:      resultado ← qtdeOcorrencia
8:      Se resultado!=0
9:          inconsistencia=true
10:     Fim se
11:     x++
12: Fim para
```

Para a aplicação deste algoritmo, são necessárias as métricas Quantidade de Classes (qtdeClasses), Quantidade de Ocorrências (qtdeOcorrencia) e um contador para a comparação entre as métricas Nome do Objeto (*intancia*.nomeObjeto) com Nome da Classe (*intancia*.nomeClasse). Além disso, este algoritmo necessita coletar o atributo para verificar se a Classe é Abstrata (tipoClasse) pois, caso o resultado da comparação entre as variáveis contadores seja diferente, o tipo de inconsistência CaSD é encontrado.

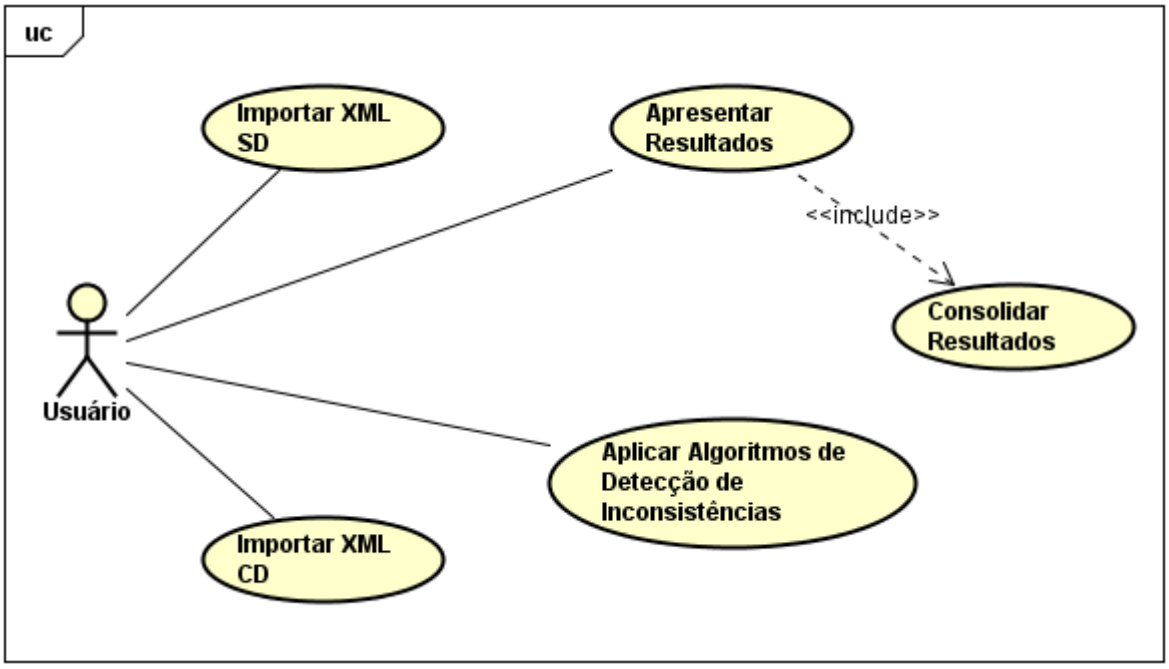
4.4 DIUML – uma Ferramenta para detecção de inconsistências

Para aplicar as métricas e heurísticas apresentadas anteriormente, nesta seção será apresentada a estrutura básica da ferramenta de detecção de inconsistências que foi desenvolvida, iniciando com a apresentação da modelagem básica do software, seguida pela descrição dos requisitos que serão aplicados, a engenharia da aplicação e seu cenário de aplicação.

4.4.1 Modelagem da ferramenta de detecção de inconsistências

Para o desenvolvimento da ferramenta de detecção de inconsistências foi identificado o Diagrama de Caso de Uso apresentado na Figura 28, de forma a definir os Atores que interagirão com o software e os Casos de Uso básicos da aplicação. O ator identificado trata-se do (1) Usuário, que tem a finalidade de dar entrada nos arquivos XMLs dos Diagramas de Classe e de Sequência.

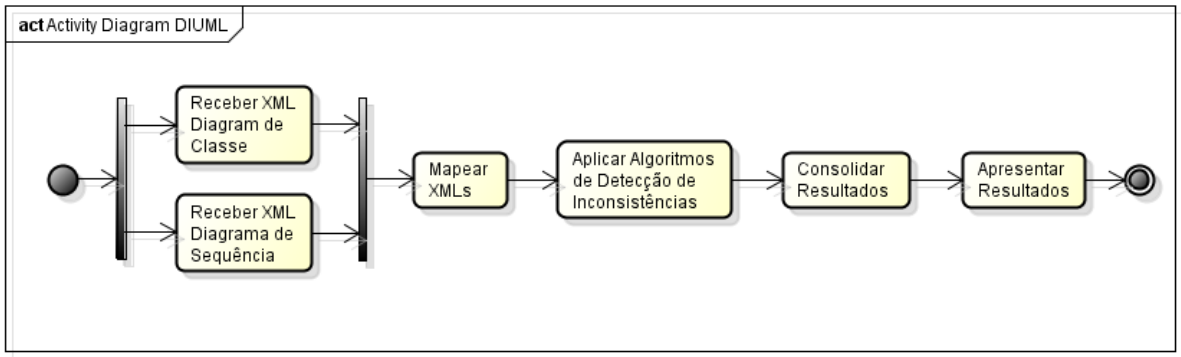
Figura 28: Diagrama de Casos de Uso da Ferramenta



Fonte: Elaborado pela autora.

Para facilitar o entendimento da ferramenta, na Figura 29 pode-se observar o mapeamento das atividades referentes às funcionalidades, através do Diagrama de Atividades, no qual, as atividades iniciais incluem o recebimento dos arquivos XML dos Diagramas de Classe e de Sequência. Estes são mapeados pela ferramenta, assim como na ferramenta *SD Metrics*, que serviu de apoio ao modelo desenvolvido. Em seguida são aplicados os Algoritmos apresentados na seção anterior. Por fim, os resultados são consolidados, em memória RAM e apresentados, através de uma interface que relaciona o elemento que apresenta a inconsistência e o tipo de inconsistência, ao usuário.

Figura 29: Diagrama de Atividades da Ferramenta



Fonte: Elaborado pela autora.

4.4.2 Descrição dos Casos de Uso da ferramenta de detecção de inconsistências

Para facilitar o entendimento da aplicação que foi desenvolvida, assim como modelar as interações entre os atores e as funcionalidades do sistema, nesta seção são refinadas as descrições dos Casos de Uso da aplicação, caracterizando assim a complementação da documentação dos Requisitos Funcionais do software, juntamente com o Diagrama de Atividades e o Diagrama de Casos de Uso, previamente apresentados. A descrição dos Casos de Uso da aplicação pode ser verificada conforme apresentado nas Tabelas 12, 13, 14, 15 e 16 a seguir:

Tabela 12: Caso de Uso 1 - Importar XML SD

Caso de uso:	**Importar XML SD**
Resumo:	O usuário realiza a importação do arquivo XML referente ao Diagrama de Sequência que será analisado
Atores:	Usuário
Pré-condições:	Arquivo XML disponível para importação Sistema aguardando importação
Descrição (fluxo):	4. O usuário seleciona a opção importar arquivo XML SD 5. O sistema apresenta a tela com os campos para a seleção do arquivo XML SD 6. O usuário seleciona a fonte do arquivo XML SD que deseja importar e aperta "Ok" 7. O sistema realiza a importação do arquivo e apresenta a mensagem "Arquivo importado com Sucesso!" 8. O sistema mapeia o arquivo XML
Pós-condições:	Sistema realiza a importação do arquivo e o mapeamento das métricas do diagrama
Caminho Alternativo:	Caso o usuário selecione um arquivo diferente de um XML, o sistema não realiza a importação e apresenta a mensagem "Arquivo selecionado não corresponde ao tipo de arquivo esperado. Tente novamente"

Fonte: Elaborado pela autora.

Tabela 13: Caso de Uso 2 - Importar XML CD

Caso de uso:	**Importar XML CD**
Resumo:	O usuário realiza a importação do arquivo XML referente ao Diagrama de Classes que será analisado
Atores:	Usuário
Pré-condições:	Arquivo XML disponível para importação Sistema aguardando importação
Descrição (fluxo):	1. O usuário seleciona a opção importar arquivo XML CD 2. O sistema apresenta a tela com os campos para a seleção do arquivo XML CD 3. O usuário seleciona a fonte do arquivo XML CD que deseja importar e aperta "Ok" 4. O sistema realiza a importação do arquivo e apresenta a mensagem "Arquivo importado com Sucesso!" 5. O sistema mapeia o arquivo XML
Pós-condições:	Sistema realiza a importação do arquivo e o mapeamento das métricas do diagrama
Caminho Alternativo:	Caso o usuário selecione um arquivo diferente de um XML, o sistema não realiza a importação e apresenta a mensagem "Arquivo selecionado não corresponde ao tipo de arquivo esperado. Tente novamente"

Fonte: Elaborado pela autora.

Tabela 14: Caso de Uso 4 - Aplicar Algoritmos de Detecção de Inconsistências

Caso de uso:	**Aplicar Algoritmos de Detecção de Inconsistências**
Resumo:	O sistema aplica aos arquivos XML a sequência de algoritmos de detecção de inconsistências
Atores:	Software
Pré-condições:	O sistema ter realizado a importação dos arquivos XML com sucesso.
Descrição (fluxo):	1. O sistema aplica o algoritmo 1 2. O sistema aplica o algoritmo 2 3. O sistema aplica o algoritmo 3 4. O sistema aplica o algoritmo 4 5. O sistema aplica o algoritmo 5 6. O sistema aplica o algoritmo 6 7. O sistema aplica o algoritmo 7 8. O sistema aplica o algoritmo 8
Pós-condições:	Sistema realiza a aplicação de todos os algoritmos de detecção de inconsistências
Caminho Alternativo:	Caso o sistema não consiga executar algum dos algoritmos, ele apresenta a mensagem "O sistema não conseguiu executar as operações, tente importar os arquivos XML novamente"

Fonte: Elaborado pela autora.

Tabela 15: Caso de Uso 6 - Consolidar Resultados

Caso de uso:	**Consolidar Resultados**
Resumo:	O sistema realiza a análise dos resultados encontrados e consolida para posterior apresentação
Atores:	Software
Pré-condições:	Arquivos XML mapeados corretamente.
Descrição (fluxo):	1. O sistema analisa as métricas mapeadas 2. O sistema analisa os resultados da aplicação dos Algoritmos 3. O sistema gera os resultados, apontando as inconsistências detectadas
Pós-condições:	Resultados sobre inconsistências detectadas consolidado
Caminho Alternativo:	Caso não seja possível consolidar os resultados, o sistema apresenta a mensagem "Não foi possível consolidar os dados, tente um novo arquivo XML"

Fonte: Elaborado pela autora.

Tabela 16: Caso de Uso 7 - Apresentar Resultados

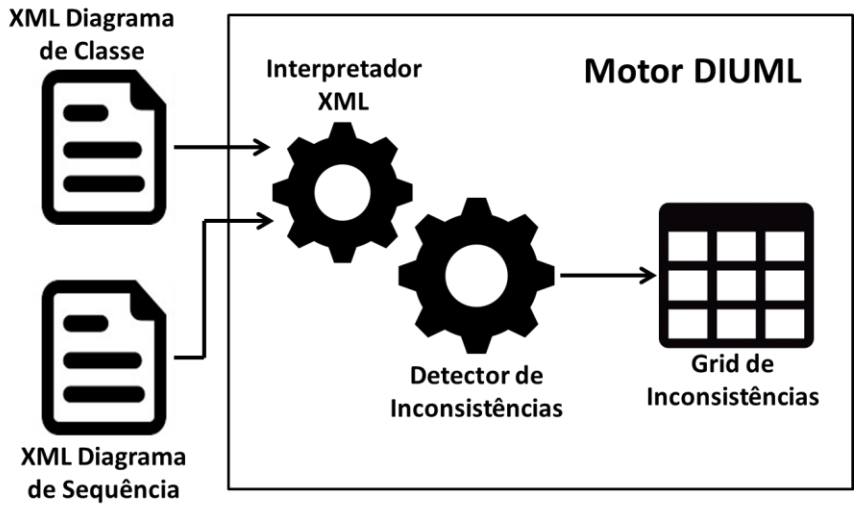
Caso de uso:	**Apresentar Resultados**
Resumo:	Os resultados encontrados sobre as inconsistências detectadas são apresentados aos usuários
Atores:	Usuário, Software
Pré-condições:	Resultados sobre inconsistências detectadas armazenadas em RAM
Descrição (fluxo):	1. O sistema consolida um formato de visualização apresentando quais métodos, classes, objetos ou mensagens que apresentam inconsistências e sua gravidade 2. O sistema apresenta ao usuário o resultado das inconsistências detectadas
Pós-condições:	Inconsistências detectadas apresentadas ao usuário
Caminho Alternativo:	Caso não seja possível apresentar os resultados, o sistema apresenta a mensagem "Resultados não podem ser apresentados, tente um novo arquivo XML"

Fonte: Elaborado pela autora.

4.4.3 Detalhes do Projeto e Implementação

A Figura 30 mostra uma representação do projeto Detecção de Inconsistências em UML (DIUML) e como seus principais elementos estão inter-relacionados. As duas entradas de dados para análise do DIUML referem-se aos arquivos XML referentes aos Diagramas de Classe e Sequência dos modelos que serão avaliados. Esses arquivos XML podem ser gerados a partir da própria ferramenta de modelagem UML, exportando os diagramas no formato XML.

Figura 30: Projeto da Ferramenta



Fonte: WEBER, 2016.

Uma vez que os arquivos XML são lidos e as variáveis necessárias são armazenadas, a ferramenta DIUML analisa cada tipo de inconsistência elencada no capitulo 4.2 deste trabalho, usando algoritmos que validam, através da lógica matemática, a presença de cada tipo de inconsistência em cada diagrama, de forma individual e conjunta, bem como apresentado no Código 3.

Após a validação da presença de inconsistência nos modelos UML, se houver alguma inconsistência detectada, a ferramenta DIUML organiza os dados encontrados em uma grade (Código 1).

Código 1. Lógica aplicada para exibir resultados.

```
1.  Type = diagramType == diagramType.Class? "Class":
2.  "Object" Name = class.Name
3.  Iconsistence = inconsistencyHelper.Obter Inconsistency
4.  (inconsistency? "NameInconsistencies")
5.  Severity = inconsistencyHelper.obterGravity
6.  (inconsistency? "High": "Medium": "Low")
```

O processo de detecção de inconsistências nos modelos UML de DIUML tem duas premissas. Primeiro, inclui a detecção de inconsistências em modelos UML multivisão através de métricas e algoritmos usando lógica matemática. Em segundo lugar, define que para cada tipo de inconsistências detectadas, uma gravidade alta, média e baixa (conforme definida na sessão 5.6) está relacionada, facilitando a tomada de decisão por parte dos desenvolvedores e analistas de sistemas sobre as ações necessárias para a correta implementação do código.

A detecção de inconsistências nos modelos UML com múltiplas visualizações usa métricas derivadas de variáveis armazenadas na memória RAM de dados de arquivos XML gerados a partir de ferramentas de modelagem UML. Para uma validação sintática e semântica da combinação de Diagramas de Classes e Sequência, a ferramenta DIUML armazena os nomes dos elementos (classe ou objeto), os métodos e mensagens, a relação entre os elementos e se as classes têm o atributo "abstract = true" na sua estrutura. A partir da

leitura e armazenamento dessas variáveis, conforme apontado no Código 2, é possível aplicar todos os algoritmos necessários para detectar inconsistências dos tipos apresentados anteriormente.

Código 2. Lendo lógica de variáveis de arquivo XML

```
1.  Var class = new Class ();
2.  Var object = new object ();
3.  Class.Name = ElementClasse.Attribute ("name"). Value;
4.  Class.Abstrata = bool.Parse (elementClasse.Attribute
5.  ("isAbstract") .Value);
6.  Class.Id = ClassClasse.Attribute ("xmi.id").Value;
7.  Var relationships = elementClasse.Descendants (ns +
8.  "ModelElement.clientDependency").FirstOrDefault ();
```

A ferramenta DIUML detecta os dois tipos de macros de inconsistências nos modelos multivisão, inconsistências sintáticas e semânticas. Para a detecção de inconsistências sintáticas, foram utilizadas métricas simples, como contadores e combinações emparelhadas. Para a detecção de inconsistências semânticas, foi necessário usar os valores dos elementos, como nomes, métodos, mensagens e IDs. Uma vez lidos os arquivos XML do Diagrama de Classe e Sequência, a ferramenta aplica, de forma linear, cada uma das validações de algoritmos para os diferentes tipos de inconsistências, como no exemplo mostrado no Código 3.

Código 3. Exemplo de método de detecção de inconsistência.

```
1.  Public void InconsistencyTypeCnCD () {
2.      Var inconsistent = Sequence.Classes.Where (c =>!
3.      Class.Classes.Any (s => s.Name == c.Name));
4.      Foreach (var inconsistent in inconsistent) {
5.         If (! Inconsistent.Insistencies.Any (i => i ==
6.          Inconsistency Type .CnCD)) {
7.             Inconsistent.Insistencies.Add
8.             (Inconsistency Type.CnCD); }}}
```

O algoritmo apresentado realiza a identificação de cada objeto no Diagrama de Sequência e verifica se há uma classe no Diagrama de Classes correspondente. Se o resultado for diferente de 1 para 1, este tipo de inconsistência estará presente.

Uma vez detectados os tipos de inconsistências nos Diagramas de Classe e Sequência, a ferramenta DIUML apresenta os resultados, incluindo uma análise da gravidade de cada tipo de inconsistência. Para a classificação de gravidade das inconsistências, foram realizados experimentos nos quais a qualidade e a produtividade dos desenvolvedores foram medidas na presença de diferentes conjuntos de Diagramas de Classes e Sequências. Na ferramenta DIUML, o código 4 mostra como a gravidade dos diferentes tipos de inconsistências foi implementada.

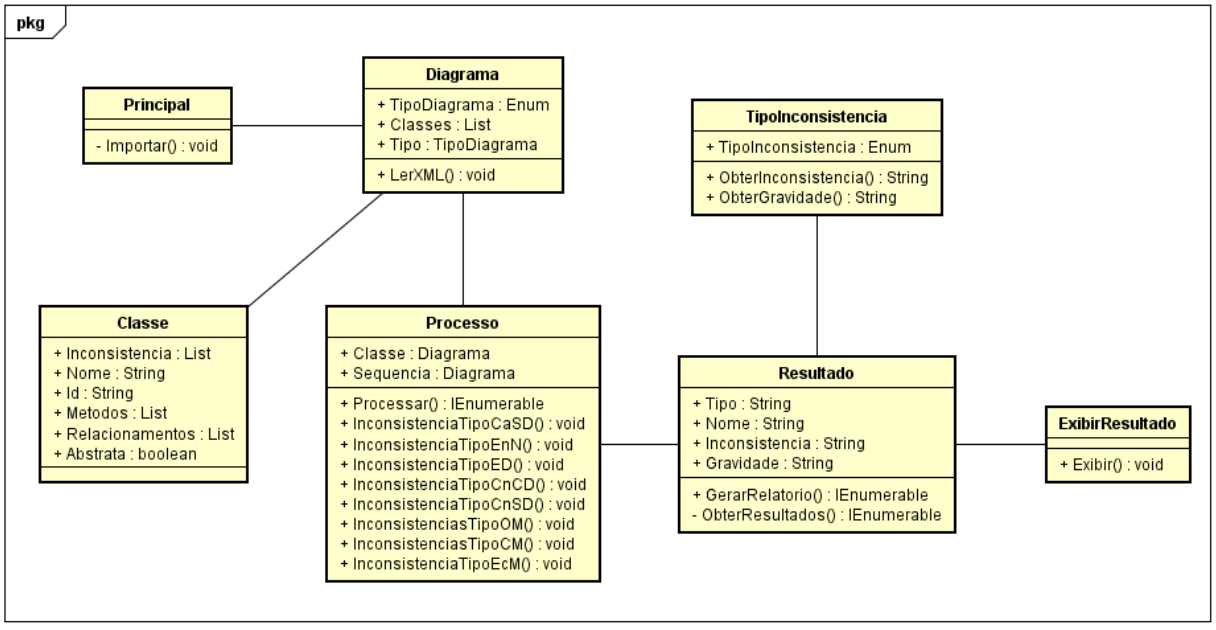
Código 4. Definição de gravidades para inconsistências.

```
1.  Case Type.Inconsistency.CnSD:
2.  Case Type.Inconsistency.EnN:
3.  Case Type.Inconsistency.CnCD:
4.       {Return "High";}
5.  Case Type.Inconsistency.EcM:
6.  Case Type.Inconsistency.ED:
7.  Case Type.Inconsistency.CaSD:
8.       {Return "Average";}
9.  Case Type.Inconsistency.Cm:
10. Case Type.Inconsistency.Om:
11.      {Return "Low";}
```

A definição da gravidade de cada tipo de inconsistência foi oriunda do resultado analisado do Esforço, Corretude e Taxa de Má Interpretação na interpretação de modelos UML multivisão, identificados através da validação experimental apresentada no capítulo 5, seção 5.6, a qual verifica o impacto de diferentes combinações de inconsistências, avaliando as de maior gravidade, resultando na classificação apresentada no Código 4.

Assim, a ferramenta implementa a leitura, validação e interpretação dos arquivos XML e aplica os oito tipos de algoritmos para detectar inconsistências, gerando assim o resultado final da detecção de inconsistências e classificando as gravidades, para permitir a interpretação dos dados por desenvolvedores e analistas de sistemas, para facilitar a tomada de decisão na codificação ou tempo de manutenção. Para exemplificar as variáveis utilizadas e detalhas a estrutura de classes presentes na ferramenta, a Figura 31 apresenta o Diagrama de Classes da ferramenta DIUML, identificando os métodos, atributos e relacionamentos de cada classe.

Figura 31: Diagrama de Classe da Ferramenta



Fonte: Elaborado pela autora.

4.4.4 Escopo e Cenário de Uso

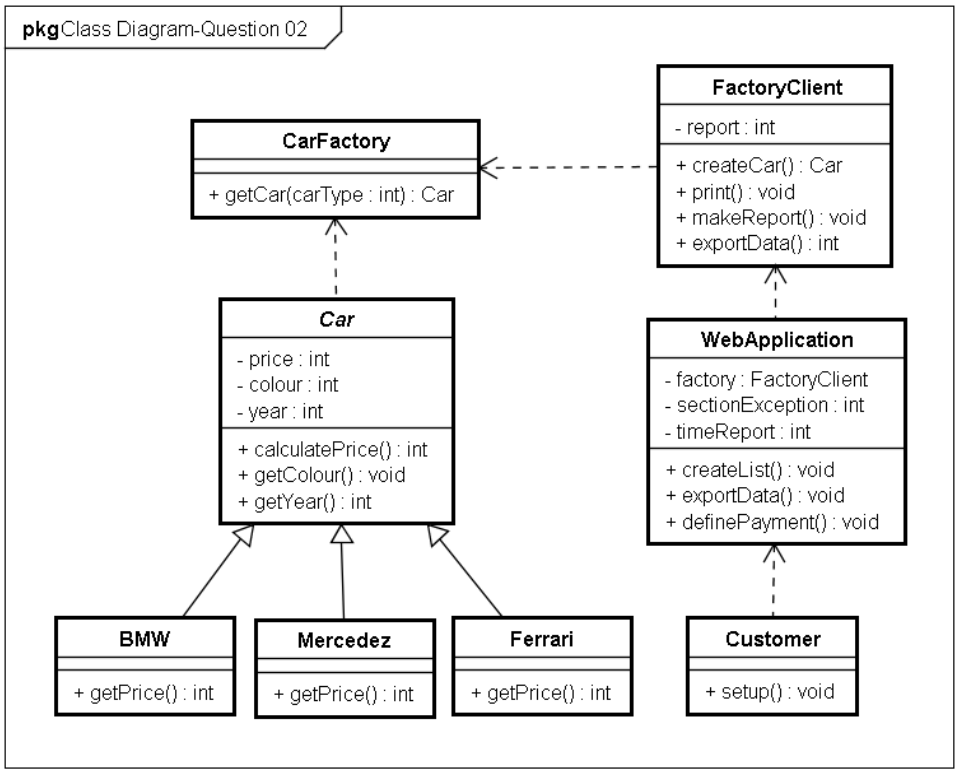
A ferramenta foi desenvolvida para implementar as técnicas de detecção de inconsistências em modelos UML multivisão baseadas em métricas, através de algoritmos de lógica matemática simples apresentadas na sessão 4.2. O objetivo disto, é facilitar a utilização das técnicas em ambientes de desenvolvimento e manutenção de softwares, possibilitando que Analistas de Sistemas e Desenvolvedores validem os Diagramas Desenvolvidos, a fim de evitar a codificação de inconsistências sintáticas e semânticas.

Para executar o DIUML, é necessário fornecer os Diagramas de Classes e Sequência no formato XML. Os arquivos de entrada não precisam estar em um local específico no computador porque a ferramenta permite a pesquisa dos arquivos usando o padrão Explorer. As entradas necessárias são os Diagramas de Classes e Sequência no formato XML, que podem ser gerados pela própria ferramenta de modelagem UML, como *Astah Pro (Astah Pro*, 2016) ou *Enterprise Architect (Enterprise Architect*, 2016).

A ferramenta não precisa ser instalada no computador do usuário e não precisa de configuração de banco de dados, pois é um arquivo executável. Isso facilita o uso em diferentes computadores e ambientes, como sistemas operacionais Windows e Linux. O DIUML não requer qualquer configuração ou alteração de parâmetros. O usuário simplesmente localiza e inicia o arquivo executável DIUML.exe. A própria ferramenta, quando executada, carrega consigo os códigos com os algoritmos de leitura dos arquivos XML e os algoritmos para detectar as inconsistências listadas na seção 4.2.

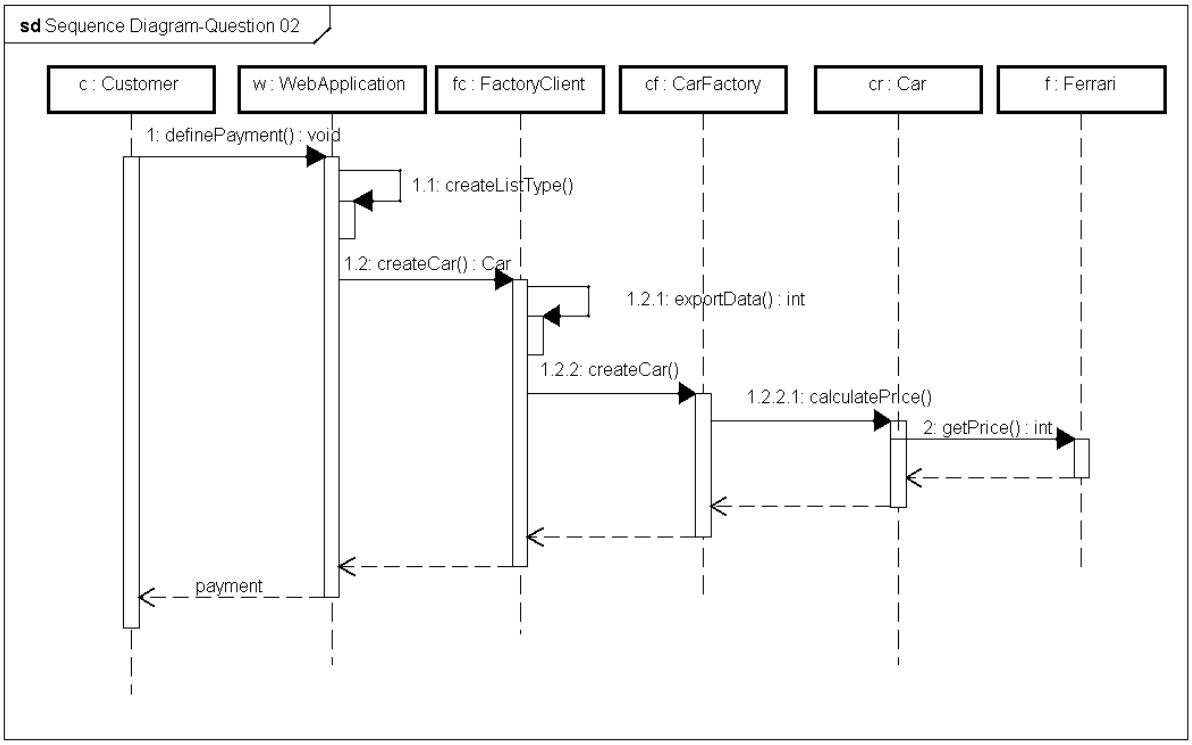
Para ilustrar o uso da ferramenta, os arquivos XML dos Diagramas de Classe e Sequência apresentados nas Figuras 32 e 33 serão lidos, analisados e as métricas serão coletadas:

Figura 32: Exemplo de Diagrama de Classe aplicado na Ferramenta



Fonte: WEBER, 2016.

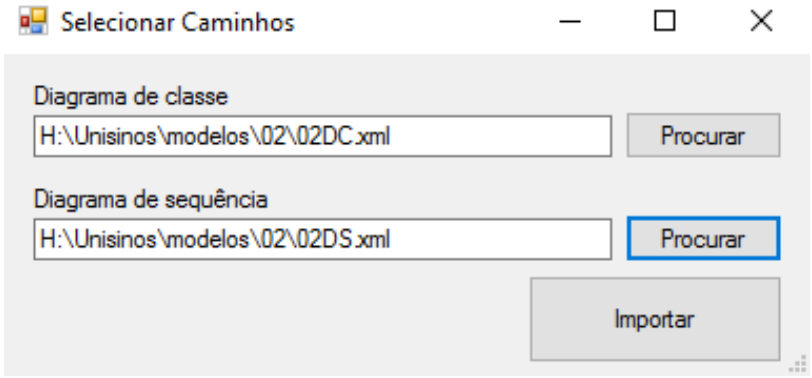
Figura 33: Exemplo de Diagrama de Sequência aplicado na Ferramenta



Fonte: WEBER, 2016.

Uma vez que iniciada, a ferramenta apresenta a tela da Figura 34, na qual o usuário deve informar os endereços do Diagrama de Classe e do Diagrama de Sequência que serão avaliados.

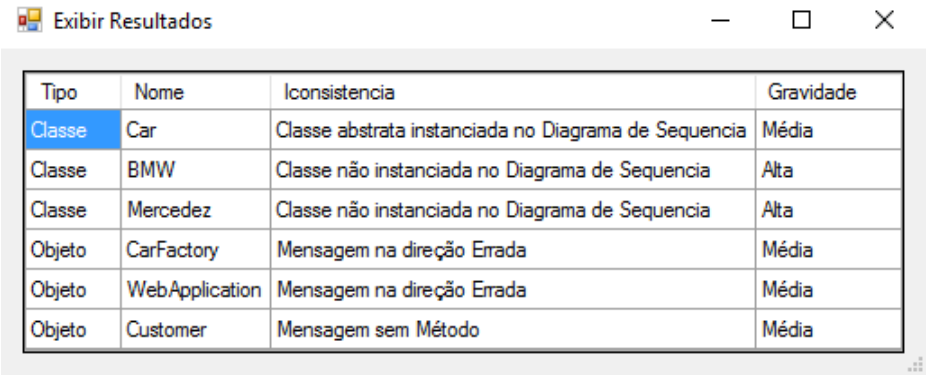
Figura 34: Tela para Seleção de Arquivos na Ferramenta



Fonte: Ferramenta DIUML.

Quando o usuário seleciona a opção "Importar", a ferramenta lê automaticamente os arquivos XML e armazena na memória local as variáveis necessárias para analisar os algoritmos de detecção de inconsistência. Uma vez completada a leitura e verificação das inconsistências bem-sucedidas, o sistema apresenta a tela referente à Figura 35, com uma grid informando os elementos nos quais foram encontradas inconsistências e quais inconsistências existem, bem como a gravidade de cada tipo de inconsistência.

Figura 35: Grid com o Resultado de Saída na Ferramenta



Fonte: Ferramenta DIUML.

5 AVALIAÇÃO

Neste Capítulo serão apresentados os experimentos realizados para investigar o impacto de inconsistências em modelos UML na qualidade e na produtividade dos desenvolvedores de software. Em particular, procura-se mensurar o esforço utilizado para interpretar diagramas UML com inconsistências, bem como entender se desenvolvedores detectam inconsistência ou não. A execução destes experimentos se justifica, uma vez que não foram identificadas, na literatura atual ou na indústria, ferramentas para a detecção de inconsistências em modelos UML, conforme apresentado nos Capítulos 2 e 3 deste trabalho. Além disso, estes experimentos permitirão avaliar a ferramenta proposta no Capitulo 4.

Sendo assim, foram realizados dois experimentos sobre detecção de inconsistências em modelos UML, referentes a análise de 10 modelos, cada um com um Diagrama de Classes e um de Sequência. O primeiro experimento realizado envolveu a aplicação de técnicas de detecção de inconsistências manuais por parte dos analistas de sistemas e desenvolvedores. O segundo experimento evolveu a utilização da ferramenta desenvolvida para detecção de inconsistências nos modelos UML por parte dos analistas de sistemas de desenvolvedores.

Este Capítulo é organizado da seguinte forma. Os objetivos e questões de pesquisa são apresentados na Seção 5.1. A formulação das hipóteses é apresenta na Seção 5.2 e na Seção 5.3 é apresentada a seleção dos participantes do estudo. O método da proposta de pesquisa é apresentado na Seção 5.4. A discussão sobre os resultados do experimento controlado é apresentada na Seção 5.5 e, por fim, a comparação entre o experimento utilizando métodos manuais para detecção de inconsistências em modelos UML e utilizando a ferramenta é apresentada na Seção 5.6.

5.1 Objetivo e Questões de Pesquisa

Para avaliar os efeitos dos impactos de inconsistências em modelos UML, foram definidas duas variáveis: o esforço para detecção de inconsistências em modelos UML e a corretude na detecção das inconsistências. Os efeitos investigados são analisados a partir da aplicação de cenários concretos envolvendo modelos UML com Diagramas de Classe e de Sequência para que os resultados empíricos possam ser gerados. Sendo assim, o objetivo deste estudo é indicado com base no modelo *Goal, Question, Metric* (GQM) (BASILI, 1994):

Analisar os impactos de inconsistências em modelos UML
com o proposito de identificar os problemas e esforço
relacionados à Qualidade de Código e Produtividade
a partir da perspectiva de desenvolvedores e analistas de software
no contexto de interpretação de modelos multivisão.

Portanto, para que se possam avaliar os impactos de inconsistências em modelos UML em esforço, corretude e taxa de má interpretação para intepretação dos diagramas, foram definidas as seguintes questões de pesquisa:

- **QP1**: A utilização da ferramenta DIUML influencia o esforço investido pelos desenvolvedores e analistas na detecção de inconsistências de modelos UML multivisão?
- **QP2**: A utilização da ferramenta DIUML afeta a eficiência dos desenvolvedores e analistas na detecção de inconsistências de modelos UML multivisão?
- **QP3**: A utilização da ferramenta DIUML provoca uma taxa de má interpretação diferente em comparação com técnicas manuais de detecção de inconsistências em modelos UML multivisão?

5.2 Formulação das Hipóteses

Hipótese 1 Espera-se que, para a análise e interpretação de modelos UML que envolvam Diagramas de Classe e de Sequência utilizando a ferramenta DIUML seja necessário menos esforço do que para a interpretação de modelos UML utilizando métodos manuais, mas de mesma configuração. Desenvolvedores e analistas acabam tendo de investir mais esforço para interpretar modelos com inconsistências de modo manual do que utilizando o apoio de uma ferramenta. Portanto, a primeira hipótese a ser analisada é de que modelos UML analisados através do apoio da ferramenta DIUML requerem menos esforço do que a interpretação de modo manual. Sendo assim, as hipóteses nula e alternativa podem ser apresentadas da seguinte forma:

> **Hipótese nula 1, $H_{1\text{-}0}$:** O esforço necessário para interpretação de modelos com Diagramas de Classe e de Sequência através do apoio da ferramenta DIUML é menor ou igual do que para a interpretação de modelos através do método manual.
> **$H_{1\text{-}0}$:** $\text{Esforço(SD, CD)}_{\text{DIUML}} \leq \text{Esforço(SD, CD)}_{\text{Manual}}$

> **Hipótese alternativa 1, $H_{1\text{-}1}$:** O esforço necessário para interpretação de modelos com Diagramas de Classe e de Sequência através do apoio da ferramenta DIUML é maior do que para a interpretação de modelos através do método manual.
> **$H_{1\text{-}1}$:** $\text{Esforço(SD, CD)}_{\text{DIUML}} > \text{Esforço(SD, CD)}_{\text{Manual}}$

O teste da primeira hipótese permite a avaliação se a utilização da ferramenta DIUML para interpretação de modelos UML representa algum impacto no Esforço para a interpretação dos diagramas.

Hipótese 2 Com o objetivo de avaliar também a corretude da interpretação realizada, foi elaborada uma segunda hipótese, a qual busca identificar se os analistas e desenvolvedores de software são capazes de afirmar se os Diagramas de Classe e de Sequência poderiam ser implementados no formato de um dos quatro códigos sugeridos ou se, devido a presença de inconsistências nos modelos, os mesmos não poderiam ser implementados em código, utilizando métodos de detecção de inconsistências manuais e, em seguida, utilizando a ferramenta DIUML. Portanto, as seguintes hipóteses nula e alternativa podem ser definidas:

> **Hipótese nula 2, $H_{2\text{-}0}$:** A corretude na interpretação de modelos com Diagramas de Classe e de Sequência com apoio da ferramenta DIUML é menor ou igual do que para a interpretação de modelos através do método manual.
> **$H_{2\text{-}0}$:** $\text{Corretude(SD, CD)}_{\text{DIUML}} \leq \text{Corretude (SD, CD)}_{\text{Manual}}$

Hipótese alternativa 2, $H_{2\text{-}1}$: A corretude na interpretação de modelos com Diagramas de Classe e de Sequência com apoio da ferramenta DIUML é maior do que para a interpretação de modelos através do método manual.
$H_{2\text{-}1}$: Corretude (SD, CD) $_{DIUML}$ > Corretude (SD, CD) $_{Manual}$

O teste da segunda hipótese permite a avaliação se a utilização da ferramenta DIUML para interpretação de modelos UML representa algum impacto na Corretude para a interpretação dos diagramas.

Hipótese 3 A terceira questão de pesquisa investiga se a Taxa de Má Interpretação (MisR) para a análise e interpretação de modelos UML que envolvam Diagramas de Classe e de Sequência é afetada quando a ferramenta de detecção de inconsistências DIUML é utilizada, em comparação com os métodos de detecção de inconsistências manuais. A utilização de uma ferramenta de apoio para a detecção de inconsistências tende a minimizar a Taxa de Má Interpretação dos Diagramas UML. Portanto, a terceira hipótese a ser analisada é de que modelos UML que são analisados através da ferramenta DIUML diminuem a Taxa de Má Interpretação. Sendo assim, as hipóteses nula e alternativa podem ser apresentadas da seguinte forma:

Hipótese nula 3, $H_{3\text{-}0}$: A taxa de má interpretação de modelos com Diagramas de Classe e de Sequência com apoio da ferramenta DIUML é menor ou igual do que para a interpretação de modelos através do método manual.
$H_{2\text{-}0}$: TaxaMaInterpretacao(SD, CD) $_{DIUML} \leq$ TaxaMaInterpretacao(SD, CD) $_{Manual}$

Hipótese alternativa 3, $H_{3\text{-}1}$: A taxa de má interpretação de modelos com Diagramas de Classe e de Sequência com apoio da ferramenta DIUML é maior do que para a interpretação de modelos através do método manual.
$H_{2\text{-}1}$: TaxaMaInterpretacao(SD, CD) $_{DIUML}$ > TaxaMaInterpretacao(SD, CD) $_{Manual}$

O teste da terceira hipótese permite a avaliação se a utilização da ferramenta DIUML para interpretação de modelos UML representa algum impacto na Taxa de Má Interpretação para a interpretação dos diagramas. Para uma visualização da formulação completa das hipóteses apresentadas, pode-se observar a Tabela 19.

Tabela 17: Hipóteses

Hipóteses Nulas	Hipóteses Alternativas
$H_{1\text{-}0}$: Effort(SD, CD) $_{DIUML}$ ≤ Effort (SD, CD)$_{Manual}$	**$H_{1\text{-}1}$:** Effort(SD, CD) $_{DIUML}$ > Effort(SD, CD) $_{Manual}$
$H_{2\text{-}0}$: Cor(SD, CD) $_{DIUML}$ ≤ Cor(SD, CD) $_{Manual}$	**$H_{2\text{-}1}$**: Cor(SD, CD) $_{DIUML}$ > Cor(SD, CD) $_{Manual}$
$H_{3\text{-}0}$: MisR(SD, CD) $_{DIUML}$ ≤ MisR(SD, CD) $_{Manual}$	**$H_{3\text{-}1}$**: MisR(SD, CD) $_{DIUML}$ > MisR(SD, CD) $_{Manual}$

Effort Esforço para detecção de inconsistências (QP1)
Cor Corretude na detecção de inconsistências (QP2)
MisR Taxa de Má Interpretação (QP3)
SD Diagrama de Sequência
CD Diagrama de Classes

Fonte: Elaborado pela autora.

5.3 Seleção dos Participantes e Contexto

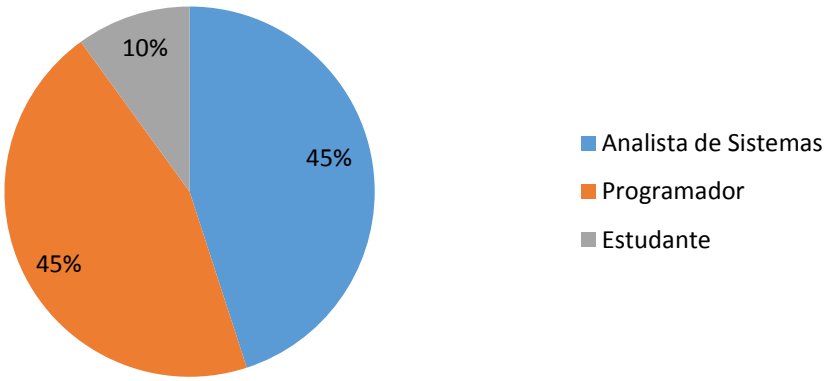
Os participantes do experimento não utilizaram nenhum tipo de auxilio ou ferramenta que facilitasse a detecção de inconsistências nos modelos UML para analisar os dez cenários de Diagramas de Classe e Sequência na primeira fase do experimento, com suas quatro opções de códigos para seleção e não estavam familiarizados com tais modelos. Na segunda fase os participantes do experimento utilizaram a ferramenta DIUML para auxilio na detecção de inconsistências nos modelos UML para analisar os dez cenários de Diagramas de Classe e Sequência. Destaca-se ainda, que os indivíduos não foram os responsáveis pela modelagem dos diagramas, apenas pela sua interpretação. Os modelos UML utilizados foram fragmentos de modelos complexos, utilizado na indústria, a partir de domínios de aplicação diferentes.

O experimento foi realizado com 15 participantes, dente eles estudantes, desenvolvedores e analistas de software. Os profissionais possuem mestrado e/ou bacharelado (ou equivalente) e possuem algum conhecimento sobre modelagem de software e grande conhecimento sobre programação. Os convidados foram selecionados para que se pudessem avaliar diferentes tipos de indivíduos, com diferentes origens e níveis de formação.

5.3.1 Perfil da amostra de pesquisa

O questionário foi aplicado em uma amostra de 15 pessoas, que efetivamente responderam as questões. Todos os que participaram foram homens, entre 22 (vinte e dois) e 52 (cinquenta e dois) anos, sendo a maioria deles, entre 25 (vinte e cinco) e 35 (trinta e cinco anos). As profissões atuais dos participantes correspondem a 45% de Programadores, 45% de Analistas de Sistemas e 10% são Estudantes, conforme a distribuição apresentada na Figura 36.

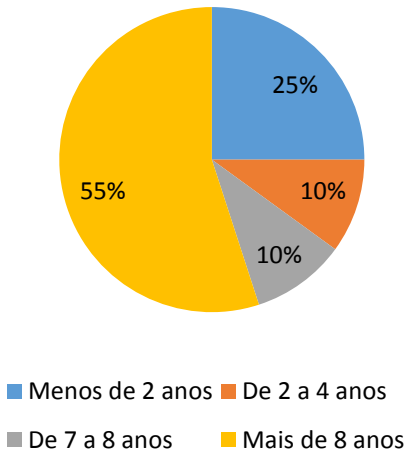
Figura 36: Gráfico de profissões da Amostra



Fonte: Elaborado pela autora.

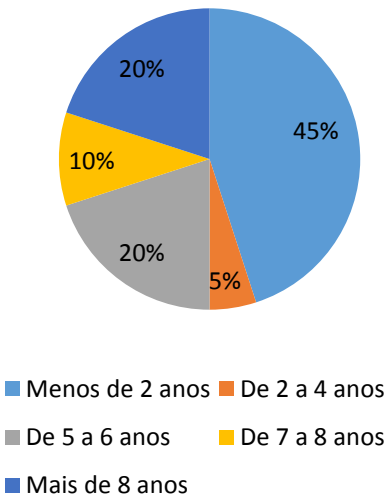
Com relação a experiência em desenvolvimento de software e análise de sistemas, observa-se que os participantes possuem mais experiência em desenvolvimento, sendo que o tempo de experiência superior a 8 (oito) anos representa 55% da amostra, enquanto 45% dos participantes, possuem menos de 2 (dois) anos de experiência em análise de sistemas, podendo ser verificado conforme as Figuras 37 e 38.

Figura 37: Gráfico de Experiência em Desenvolvimento



Fonte: Elaborado pela autora.

Figura 38: Gráfico de Experiência em Modelagem de Software



Fonte: Elaborado pela autora.

Conforme análise dos gráficos apresentados é possível concluir que a amostra é aceitável e corresponde a pessoas que possuem relação direta com a área de informática (seja relacionada a desenvolvimento ou modelagem de software) tanto nas suas formações acadêmicas quanto nas suas profissões.

5.4 Método da proposta de pesquisa

Para realização de um estudo experimental com o objetivo de compreender os impactos das falhas na UML, já elencadas, em qualidade e produtividade de código, foi elaborado um questionário, dividido em duas partes, sendo a primeira para identificação dos participantes e a segunda envolvendo dez questões de múltipla escolha, entre cinco opções, sendo somente uma correta.

Para elaboração das questões de múltipla escolha, foram desenvolvidos 10 diagramas de Classe e dez diagramas de Sequência correspondentes. Nestes diagramas, foram inseridos diferentes tipos de inconsistências, conforme pode ser observado na Tabela 18.

Tabela 18: Tipos de Inconsistências por Questão

Questão	Tipos de Inconsistências	Diagrama de Classes		Diagrama de Sequência	
		Classes	Métodos	Objetos	Mensagens
Q01	EnN e EcM	7	38	6	7
Q02	EcM e CnCD	8	22	6	7
Q03	ED	4	20	4	7
Q04	CnSD	4	18	3	4
Q05	Sem inconsistências (Questão Controle)	4	24	4	6
Q06	EcM e Cm	6	16	6	6
Q07	EnN e ED	4	16	4	6
Q08	EcM e CnSD	5	25	4	7
Q09	EcM, CnCD e Cm	4	18	6	6
Q10	EnN, EcM e CnCD	4	18	5	8

Fonte: Elaborado pela autora.

Além das questões com seus respectivos Diagramas de Classes e Sequência, os participantes receberam um questionário de levantamento de informações profissionais e acadêmicas, na qual puderam responder questões como tempo de experiência em análise e desenvolvimento de software e nível de formação acadêmica. A coleta destes dados permitiu a construção e análise do perfil e experiência dos participantes da amostra. A amostra definida para a aplicação do questionário envolveu estudantes de mestrado na área de computação e profissionais atuantes com desenvolvimento e análise de sistemas. Nenhum dos membros da amostra trabalhava apenas com análise de software.

Todos os participantes do experimento foram submetidos a dois tratamentos (detecção de inconsistências em modelos UML utilizando métodos manuais e detecção de inconsistências em modelos UML utilizando a ferramenta DIUML) para permitir a comparação dos pares de material experimental coletados. A primeira fase do questionário, sem apoio da ferramenta, foi realizada entre abril e maio de 2015 e a segunda fase do experimento, relacionada a realização do questionário com o apoio da ferramenta DIUML para detecção de inconsistências em modelos UML multivisão ocorreu novembro de 2016. Portanto, o método experimental deste estudo é, por definição, um método equilibrado, pois, em mais de um ano, as respostas do primeiro questionário tenderam a não influenciar no resultado do segundo, pois, como os participantes do experimento foram submetidos aos dois tratamentos, sempre foi presente a preocupação sobre as informações que os participantes poderiam ganhar com a primeira fase para realizar o experimento na segunda fase.

Além disto, para minimizar o possível "ganho de informação", algumas estratégias experimentais (KITCHENHAM et al., 2008) e (WOHLIN et al., 2000) foram seguidas. Primeiro, os modelos utilizados no estudo foram fragmentos de Diagramas de Classe e Diagramas de Sequência derivados de ambientes de produção reais, de contextos diferentes dos quais os participantes estavam familiarizados, para que os mesmos não tivessem nenhuma informação prévia e nenhum conhecimento acumulado sobre a semântica dos elementos do modelo. Em segundo lugar, cada questão do formulário terá composta por um Diagrama de Classe e um Diagrama de Sequência que representava diferentes funcionalidades do mesmo software. Por fim, cada par de Diagramas de Classe e Diagramas de Sequencia tinham diferentes tipos de inconsistências (conforme apresentado na Tabela 18), e os significados de seus elementos eram completamente diferentes. Portanto, pode-se supor que o desempenho

dos participantes do experimento controlado não foi influenciado pelos tratamentos das perguntas anteriores.

Para realizar o experimento, cada participante recebeu: (1) um Diagrama de Classes e um Diagrama de Sequência em cada questão (atividade) do experimento; e (2) cinco opções de respostas, da letra A a E, sendo, nas letras A a D, apresentadas opções de códigos para a implementação dos diagramas e a opção E, que deveria ser marcada caso o participante identificasse alguma inconsistência no modelo, não sendo possível a implementação dos diagramas através dos códigos disponíveis, conforme Apêndice B e Apêndice C. Para medir a o esforço necessário para cada uma das questões do experimento, cada participante registrou, junto à questão resolvida, o horário inicial e o horário final da resolução da questão, possibilitando assim a coleta e análise do intervalo de tempo gasto para realizar a detecção ou não de inconsistências, respondendo à Questão de Pesquisa 1. Assim, foi possível medir o esforço necessário para resolução das questões por parte dos participantes.

5.4.1 Variáveis e método de quantificação

A variável independente deste estudo é o método de detecção de inconsistências em modelos UML multivisão. É nominal e dois valores podem ser assumidos: detecção de inconsistências através de métodos manuais e detecção de inconsistências com apoio da ferramenta DIUML. Estas variáveis descrevem os tratamentos dados em cada questionário e, e seus impactos foram investigados nas variáveis dependentes Corretude, Esforço e Taxa de Má Interpretação, conforme descrito abaixo:

Esforço (Effort). A variável Esforço representa a média de tempo (minutos) gasto pelos participantes do experimento para a detecção das inconsistências nos modelos UML de cada questão (QP1). Os participantes do experimento detectam inconsistências quando eles explicitamente indicam que eles são incapazes de alcançar uma implementação adequada a partir dos diagramas apresentados em cada questão do formulário.

Correturde (Cor). A variável Corretude destina-se a medir o quanto os participantes do experimento conseguiram detectar as inconsistências presentes nos modelos UML (QP2). Representa a razão entre o número de participantes do experimento que detectaram as inconsistências em uma questão, dividida pelo número de participantes do experimento que respondem à pergunta sem identificar a presença de inconsistência.

Taxa de má interpretação (MisR). Esta variável representa o grau de variação das respostas (RQ3). Ou seja, mede a concentração das respostas sobre as quatro alternativas possíveis, uma vez que a quinta alternativa representa a detecção de inconsistência. Busca-se analisar se as inconsciências não detectadas afetam a interpretação dos diagramas pelos participantes do experimento, pois, (LANGE, 2006) uma inconsistência não detectada pode não ser necessariamente um problema, se todos os participantes do experimento tiverem a mesma interpretação. Por exemplo, se os 15 participantes tiverem a mesma resposta (como a alternativa "A") para uma pergunta, então as inconsistências nos diagramas não levaram a interpretações erradas (MisR = 1). Por outro lado, se as respostas dos participantes se espalham igualmente sobre as quatro alternativas propostas, as inconsistências apresentadas no modelo estão gerando interpretações errôneas (MisR = 0). Ou seja, a taxa de má interpretação é 0 se as respostas são distribuídas igualmente sobre todas as opções, e 1 se as respostas são concentradas apenas uma opção de resposta, conforme (LANGE, 2006), esta variável pode ser medida da seguinte forma.

$$MisR(k_0, \dots, k_{K-1}) = 1 - 2\frac{\sum_{0 \le i < K} k_i i}{N(K-1)}$$

Onde:

K: É o número de alternativas para uma pergunta

k_i: É o número de vezes que a alternativa i foi selecionada, onde $0 \le i < K$ e (para todo i: $0 \le i < K - 1: k_i \ge k_{i+1}$

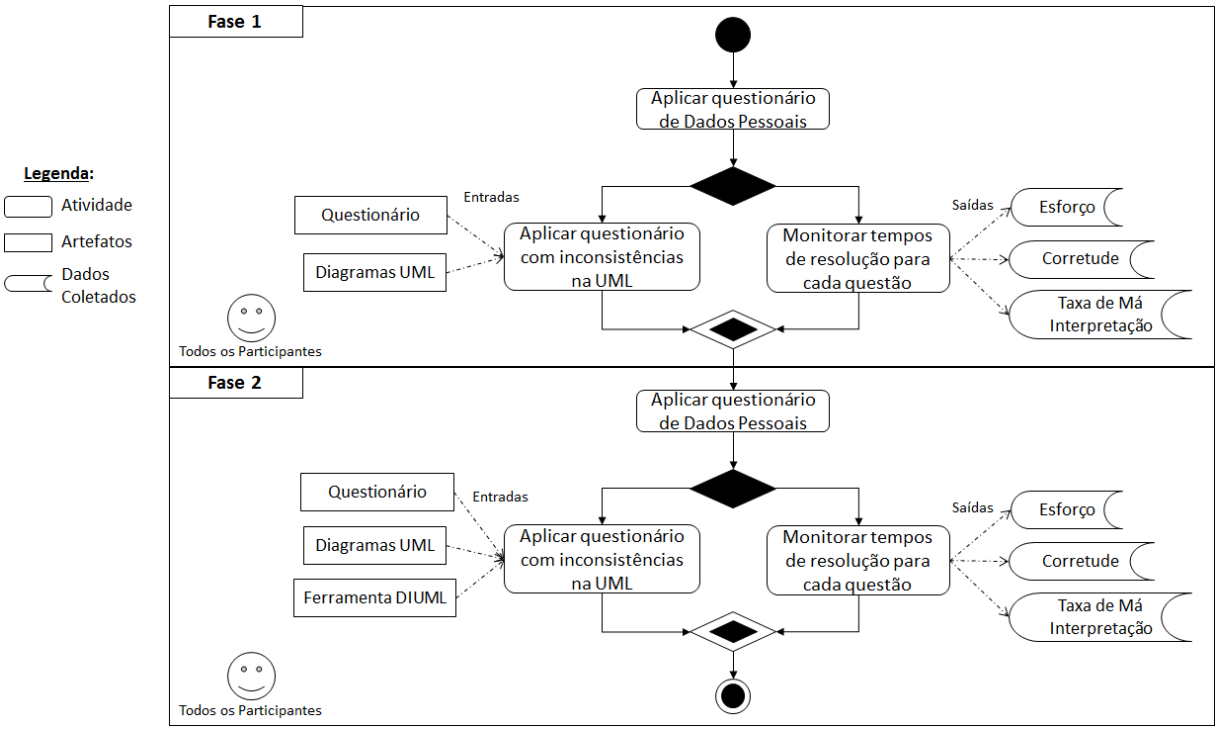
N: É a soma das respostas sobre todas as alternativas: $N = \sum_{0 \le i < K} k_i$

5.4.2 Operação

Fase de preparação. Para evitar resultados tendenciosos, os participantes do experimento não possuíam conhecimento das questões de pesquisa e hipóteses deste estudo. A motivação dos participantes foi a contribuição para um estudo de um colega do mestrado e colega de trabalho. Os participantes receberam o questionário na forma impressa ou por e-mail, no formado Word com campos editáveis.

Fase de execução. As duas fases da etapa de execução, conforme figura 39, deveriam ter ocorrido em uma sala do curso de Programa Interdisciplinar de Pós-Graduação em Computação Aplicada (PIPCA), da Universidade do Vale do Rio dos Sinos (Unisinos). Porém, devido às limitações de tempo e localização, os participantes puderam executar a experiência também em seu ambiente de trabalho, no entanto, o experimento foi cuidadosamente controlado. O tempo entre a realização de cada fase do experimento foi de 1 ano. Logo no início do experimento, os participantes preencheram um questionário para coletar seus Dados Pessoais, ou seja, sua formação acadêmica e experiência de trabalho. Em seguida, todos os participantes do experimento receberam as 10 perguntas do questionário e as folhas para respostas, conforme Apêndice B e Apêndice C. É importante ressaltar que os participantes ficaram sempre bem à vontade com relação ao tempo para execução das questões, mas também, foram rigorosamente supervisionados referente ao registro correto do tempo. Portanto, espera-se que o tempo tenha sido registrado corretamente. Estudos observacionais foram conduzidos para melhorar a compreensão de como as tarefas no experimento foram realizadas.

Figura 39: Fluxo de Execução do Experimento



Fonte: Elaborado pela autora.

5.5 Resultados do Experimento Controlado

Nesta sessão são apresentados os resultados do Experimento Controlado, descrevendo a análise e teste estatístico para cada uma das Questões de Pesquisa e Hipóteses levantadas.

5.5.1 QP1: Esforço para Detecção de Inconsistências em modelos UML multivisão

Estatísticas descritivas. A primeira questão de pesquisa investiga o esforço aplicado pelos desenvolvedores e analistas de sistemas para detectar inconsistências nos modelos UML multivisão, utilizando métodos manuais e o apoio da ferramenta DIUML. Os desenvolvedores e analistas de sistemas utilizam mais esforço para detectar inconsistências de modo manual do que com o apoio da ferramenta DIUML. A média de esforço para detecção de inconsistências utilizando a ferramenta DIUML é de 3,45 (minutos) enquanto que, utilizando métodos manuais é de 3,79 (minutos), como pode ser observado na tabela 19. Isso representa uma diminuição de 8,89% do esforço para detecção de inconsistências utilizando a ferramenta DIUL em comparação à métodos manuais. Esta diminuição do esforço oriundo do uso da ferramenta DIUML também é observado através da comparação das medianas. O esforço de detecção varia de 2,95 (minutos) com o uso da ferramenta para 3,33 (minutos) sem o uso da ferramenta, o que representa uma diminuição de 11,41% do esforço. Esse fenômeno confirmou a hipótese inicial de que o apoio de uma ferramenta baseada em métricas para a detecção de inconsistências em modelos UML multivisão minimiza o esforço para a interpretação dos Diagramas de Classe e Sequência.

Teste de hipóteses. Uma vez que os testes de normalidade de Shapiro-Wilk e Kolmogorov-Smirnov (LEVINE, 1999) indicam que os dados são normalmente distribuídos, o Paired t-test é aplicado para testar a H_1. O valor *t-statistic* encontrado é de 0,69 com o *p-value* de 0,50, conforme tabela 19. Este *p-value* (>0,05) indica que a primeira hipótese nula ($H_{1\text{-}0}$) não pode ser rejeitada. Isto sugere que a diferença média do esforço para detecção de inconsistência em modelos UML multivisão não é zero, porém, não há evidência (conforme o nível de significância 0,05) de que os desenvolvedores investem mais esforços para detectar inconsistências de modo manual do que utilizando a ferramenta DIUML. O esforço de detecção de inconsistências assume um valor negativo para a diferença média (-0,33), enquanto o *p-value*=0,50 não é menor que 0,05. Isto implica que o esforço de detecção de inconsistências utilizando métodos manuais não é maior, estatisticamente, do que utilizando a ferramenta DIUML como apoio. Além disso, o teste não paramétrico de Wilcoxon é aplicado para eliminar qualquer ameaça relacionada à validade estatística de conclusão. O valor do *p-value*=0,08 também confirmou a conclusão anterior. Portanto, não se pode rejeitar a hipótese nula $H_{1\text{-}0}$.

Tabela 19: Medidas para Estatísticas Descritivas e Testes Estatísticos

Variáveis	Tratamento	Mean	St Dev	Min	25th	Med	75th	Max	%diff	Wilcoxon	Paired t-test		
										p-value	t	p	MD
Esforço	Manual	3,79	1,55	2,17	2,43	3,33	4,79	7,20	8,89%	0,08	0,69	0,50	-0,33
	DIUML	3,45	1,21	2,33	2,62	2,95	3,88	6,53					
Corretude	Manual	0,39	0,16	0,13	0,25	0,37	0,55	0,60	32,21%	0,01	3,26	0,01	-0,13
	DIUML	0,52	0,09	0,40	0,47	0,47	0,62	0,67					
Má Interpretação	Manual	0,51	0,16	0,23	0,37	0,50	0,67	0,77	15,68%	0,14	1,37	0,20	-0,08
	DIUML	0,59	0,09	0,40	0,52	0,60	0,68	0,70					

*com 4 graus de variação, nível de significância de $\alpha = 0,05$, MD: mean difference, p: p-value, St Dev: standard deviation
Fonte: Elaborado pela autora.

5.5.2 QP2: Corretude na Detecção de Inconsistências em modelos UML multivisão

Estatísticas descritivas. A segunda questão de pesquisa investiga se os desenvolvedores e analista de sistemas detectam mais (ou menos) inconsistências nos modelos UML multivisão utilizando métodos manuais ou o apoio da ferramenta DIUML. Os desenvolvedores e analistas de sistemas detectaram mais inconsistências nos modelos utilizando a ferramenta DIUML como apoio do que os métodos manuais. A corretude superior do uso da ferramenta pode ser explicada comparando médias e medianas, conforme tabela 19. Os resultados apresentam uma melhora de 32,21% na corretude através do apoio da ferramenta DIUML do que a utilização apenas de métodos manuais, sendo a média de 0,39 com métodos manuais em comparação com a média de 0,52 com o uso da ferramenta DIUML. A diferença observada entre as medianas também favorece o uso da ferramenta. Isso inclui uma superioridade de 27,02% no número de casos em que os desenvolvedores e analistas relataram ser incapazes de fornecer uma implementação. Os resultados sugerem que a utilização da ferramenta DIUML permite aos desenvolvedores e analistas de sistemas identificar mais inconsistências do que a interpretação manual dos diagramas.

Teste de hipóteses. Uma vez que os testes de normalidade de Shapiro-Wilk e Kolmogorov-Smirnov (LEVINE, 1999) indicam que os dados são normalmente distribuídos, o Paired t-test é aplicado para testar a H_2. O valor *t-statistic* encontrado é de 3,26 com o *p-value* de 0,01,

conforme tabela 19. Este valor baixo para o *p-value* (<0,05) indica que a segunda hipótese nula (H_{2-0}) pode ser rejeitada. Isto implica que a diferença média da corretude na interpretação dos nos modelos UML multivisão utilizando ou não a ferramenta não é zero. Portanto, há fortes evidências (conforme o nível de significância 0,05) de que os desenvolvedores e analistas de sistemas detectem mais inconsistências em modelos utilizando a ferramenta DIUML do que métodos manuais. As diferenças médias entre os pares de detecção de inconsistências, utilizando ou não a ferramenta, indicam a direção na qual o resultado é significativo. Por exemplo, considerando a corretude variável para detecção de inconsistências utilizando ou não a ferramenta DIUML, a diferença média é negativa (-0,13). Isto implica que a corretude utilizando métodos manuais é estatisticamente menor do que utilizando a ferramenta DIUML. Além disso, o teste não paramétrico de Wilcoxon é aplicado para eliminar qualquer ameaça relacionada à validade estatística de conclusão. O valor baixo do *p-value* = 0,01 (<0,05) também confirmou a conclusão acima mencionada. Portanto, pode-se rejeitar a hipótese nula H_{2-0}.

5.5.3 QP3: Taxa de Má Interpretação na Detecção de Inconsistências em modelos UML multivisão

Estatísticas descritivas. A terceira questão de pesquisa investiga se a utilização da ferramenta DIUML como ferramenta de apoio na detecção de inconsistências em modelos UML multivisão conduz a uma taxa de má interpretação maior ou menor do que os métodos de detecção manuais. A Tabela 19 apresenta as estatísticas descritivas das medidas de taxas de má interpretação utilizando ou não a ferramenta DIUML. Considera-se que a variável MisR varia entre 0 e 1, e que MisR = 1 indica que os desenvolvedores e analista de sistemas não estão apresentando má interpretação e que, MisR = 0 indica que as respostas dos desenvolvedores e analistas de sistemas espalharam-se igualmente sobre as quatro alternativas diferentes, resultando em má interpretação. A utilização da ferramenta de apoio DIUML resulta em menos interpretação incorreta (maior valor de Misr) do que métodos de detecção de inconsistências manuais. A taxa de má interpretação é 15,68% menor utilizando métodos de detecção de inconsistências em modelos UML manuais. A média é de 0,51 para utilização de métodos manuais contra 0,59 utilizando a ferramenta DIUML. Esta tendência ascendente também é observada nas medianas: 0,50 utilizando métodos manuais de detecção de inconsistências e 0,60 utilizando a ferramenta DIUML, compreendendo um aumento de 16,66%. Os resultados sugerem que a utilização da ferramenta de apoio DIUML acarreta um impacto positivo na interpretação de modelos UML multivisão por parte dos desenvolvedores e analistas de sistemas, do que a utilização de métodos manuais. Esperava-se que, através da utilização da ferramenta DIUML para apoio na interpretação de modelos UML multivisão com inconsistências minimizaria o número de interpretações divergentes, e isso foi confirmado.

Teste de hipóteses. Para a análise dos resultados dos testes para a H_3, segue descrito. Como nas análises anteriores, o Paired t-test é aplicado para testar H_3, uma vez que as medidas assumem uma distribuição normal. A Tabela 19 demonstra os *p-value* em pares e as diferenças de média entre os pares para cada medida. Como a diferença média é negativa (-0,08) e o *p-value* é 0,20 é maior que 0,05, não se pode estimar que a evidência significativa de que o número de interpretações divergentes utilizando a ferramenta DIUML é estatisticamente maior do que utilizando os métodos de detecção de inconsistências manuais. Além disso, o teste não paramétrico de Wilcoxon foi aplicado para verificar esta conclusão, sendo seu *p-value*=0,14, assumindo um valor alto (<0,05). Portanto, como o *p-value* é maior que 0,05,

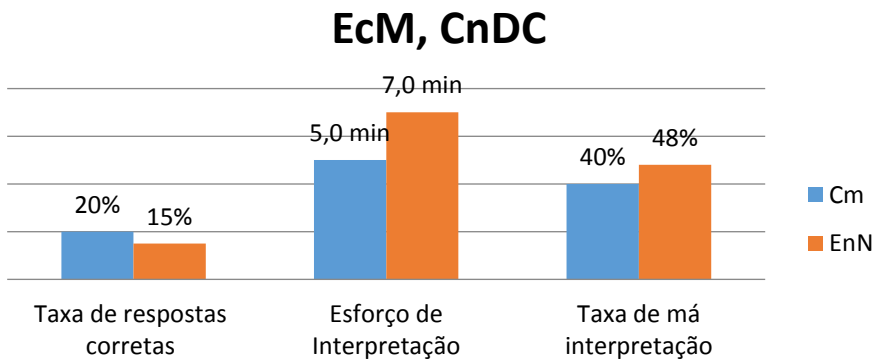
mesmo a diferença média sendo negativa, pode-se concluir que não há evidências estatísticas de que a taxa de má interpretação utilizando a ferramenta DIUML significativamente maior do utilizando métodos manuais para detecção de inconsistências em modelos UML multivisão. Portanto, não se pode rejeitar a hipótese nula $H_{3\text{-}0}$.

5.6 Discussão sobre a gravidade de cada tipo de Inconsistências

Com o objetivo de identificar os diferentes impactos oriundos de combinações diferentes de falhas na UML foram realizadas mais 4 (quatro) tipos de análises diferentes, fixando um ou dois tipos de inconsistências e comparando suas Corretude, Taxa de Má Interpretação e Esforço.

Para a primeira análise, foram fixadas as inconsistências do tipo EcM e CnDC variando com as inconsistências do tipo Cm e EnN. Como se pode observar, ambas as combinações resultaram em uma taxa de respostas corretas baixa, com 20% e 15% respectivamente. Além disso, ambas as combinações envolveram um esforço de interpretação elevado, com 5 e 7 minutos. Por fim, a taxa de má interpretação das questões também teve resultados baixos, ficando com 0,4 e 0,48, conforme apresentado na Figura 40.

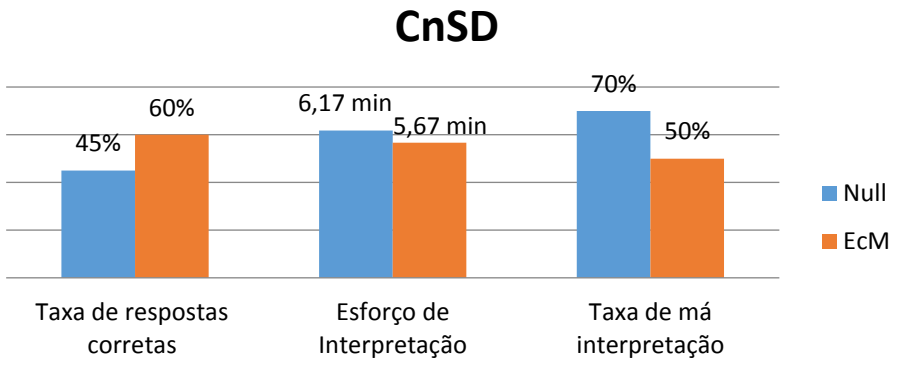
Figura 40: Gráfico da Primeira Análise Comparativa



Fonte: Elaborado pela autora.

Para a segunda análise de combinações, foi fixada a inconsistência do tipo CnDC variando com a aplicação de uma inconsistência do tipo EcM. Como se pode observar, a questão com aplicação do EcM junto ao CnDC apresentou maior taxa de respostas corretas do que a questão que apresentava apenas uma CnDC. Ambas as combinações envolveram um esforço de interpretação intermediário, variando pouco entre 6,17 e 5,67 minutos. Por fim, a taxa de má interpretação das questões apresentou bons resultados, pois a questão que obteve menor índice de respostas corretas também obteve a maior taxa de má interpretação, o que significa baixa má intepretação, conforme Figura 41.

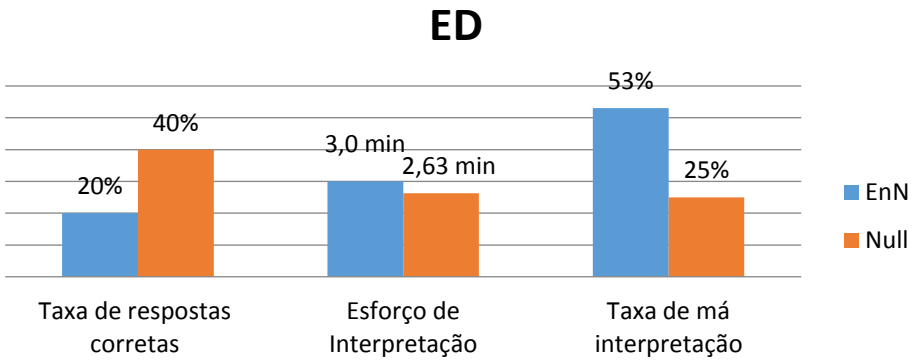
Figura 41: Gráfico da Segunda Análise Comparativa



Fonte: Elaborado pela autora.

Para a terceira análise realizada, foi fixado tipo de inconsistência ED variando com a aplicação de uma EnN. Diferente da questão anterior, neste caso a questão que apresentou apenas a falha ED apresentou maior taxa de respostas corretas, com 40%, o dobro da questão com a falha EnN. Ambas as combinações envolveram um esforço de interpretação baixo, variando pouco entre 3 e 2,63 minutos. Por fim, a taxa de má interpretação das questões apresentou relação direta a baixa taxa de acertos, apresentando 0,53 de má interpretação para a questão que combinava as falhas, conforme Figura 42.

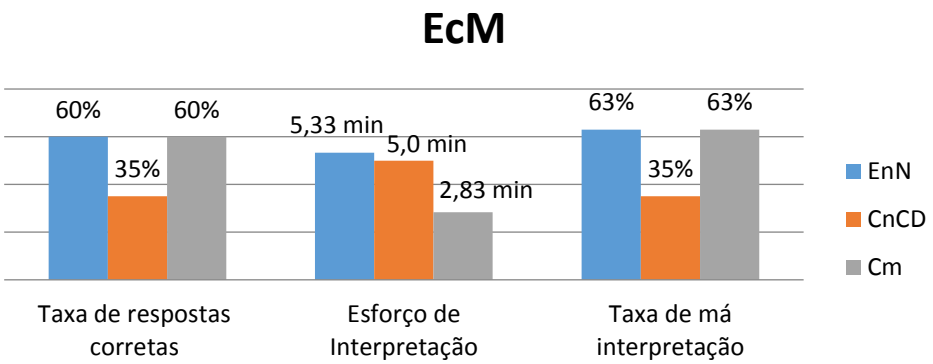
Figura 42: Gráfico da Terceira Análise Comparativa



Fonte: Elaborado pela autora.

Concluindo os experimentos de combinações de falhas, na análise final, apresentada na Figura 43, foi fixado o tipo de inconsistência EcM variando com a aplicação de do tipo EnN, ou CnCD ou ainda, uma inconsistência do tipo Cm. Como pode-se observar, as questões com aplicação do EnN e Cm apresentaram os mesmos índices de taxa de respostas corretas, 60%, enquanto a combinação com a inconsistência do tipo CnDC apresentou apenas 35%, demonstrando um maior impacto quando este tipo de combinação ocorrer. Com relação ao esforço, pode-se observar uma variação maior, com 5,33, 5 e 2,83 minutos em cada respectiva questão. Assim como a taxa de respostas corretas, a taxa de má interpretação apresentou resultados similares para EnN e Cm, 0,63, enquanto pode ser observada uma distribuição maior na combinação com CnCD, 0,35.

Figura 43: Gráfico da Quarta Análise Comparativa



Fonte: Elaborado pela autora.

Foi possível a verificação e validação de cada comparação de combinações de inconsistências identificadas e levantadas para o estudo, possibilitando coletar dados suficientes para uma analise da produtividade e qualidade do desenvolvimento de software quando são utilizados modelos UML com inconsistências nos Diagramas.

Este experimento foi realizado com o objetivo de comparar os impactos na qualidade e na produtividade de diagramas com diferentes tipos de inconsistências e sem inconsistências, identificando o esforço e a corretude das respostas, sendo o resultado apresentado na Tabela 20. A aplicação de um questionário, no âmbito de um estudo experimental possibilitou a análise e coleta dos dados necessários para verificação de cada tipo de inconsistência e uma comparação entre as combinações de inconsistências mais graves para a qualidade e produtividade do software, indicando em quais situações Analistas e Desenvolvedores devem dedicar-se mais para evitar a replicação destas inconsistências quando realizarem a interpretação destes diagramas.

Tabela 20: Gravidade por Tipo de Inconsistência

Gravidade	Inconsistência
Alta	CnSD
	EnN
	CnCD
Média	EcM
	ED
	CaSD
Baixa	Cm
	Om

Pode-se concluir que os casos com menor índice de acertos envolveram a combinação das inconsistências ED e EnN, EnN, EcM e CnDC e Cm, EcM e CnDC, indicando que falhas envolvendo Mensagens sem Nome, Objetos sem nenhuma Classe no Diagrama de Classes e Mensagem sem Método devem receber mais atenção no momento da criação e interpretação dos diagramas, principalmente por se tratarem de falhas que passam despercebidas aos usuários.

Muitos dos participantes não conseguiram identificar várias das inconsistências presentes nos diagramas UML, resultando em taxas de acerto, no geral, bem baixas, variando entre 15% e 50%, o que demonstra a dificuldade na detecção de inconsistências entre os Diagramas de Classe e de Sequência quando são interpretados por quem não os desenvolveu.

Por fim, as inconsistências que mais demandaram esforço, independente da taxa de acerto encontrada para elas, foram as que envolveram a combinação entre EnN, EcM e CnDC e CnCD e CnSD e EcM, novamente trazendo a tona a importância de manter a atenção em inconsistências dos tipos CnCD e EcM, que são inconsistências simples, mas que geram grande impacto em qualidade e produtividade do desenvolvimento de software.

6 CONCLUSÕES E TRABALHOS FUTUROS

Visto que a UML é uma linguagem de modelagem de software já consolidada na indústria e na academia, tornam-se necessários estudos que destacam o impacto negativo que as inconsistências em modelos e diagramas podem ter na produtividade e na qualidade de software (FARIAS, 2013), assim como, a consolidação de uma aplicação, que utilize métricas para a detecção de inconsistências nestes modelos. Sendo assim, foi realizado o estudo de mapeamento sistemático considerando o assunto detecção de inconsistências em modelos UML e seus impactos e efeitos sobre a produtividade e qualidade no desenvolvimento de software, para proporcionar uma compreensão aprofundada da literatura na área inconsistência nos modelos UML.

Foram identificados alguns poucos estudos que se baseiam no uso de métricas para identificar inconsistências, porém, a grande maioria dos estudos dispõe de um método automatizado para detectar inconsistências sintáticas baseadas em regras formais da própria linguagem de modelagem UML, o que pode levar ao questionamento sobre o porquê de uma aplicação destas não estar sendo utilizada largamente na indústria e na academia. Pôde-se concluir que este fato se deve a dois motivos. Primeiro, é mais simples a utilização das regras básicas de modelagem UML na configuração do software para a detecção automática de inconsistências, as quais, muitos softwares de modelagem já possuem configuradas. Segundo, referente ao foco em inconsistências sintáticas, é a característica mais simples a ser analisada na detecção de inconsistências, pois a análise semântica de inconsistências é muito relativa ao contexto dos modelos.

Através da identificação das lacunas de investigação na área, foi executado um experimento controlado, de forma que pudesse ser a avaliada a necessidade de uma metodologia baseada em métricas, consolidada em uma aplicação, para a detecção de inconsistências em modelos UML multivisão. Analistas de Software e Desenvolvedores foram questionados sobre quais cenários, dos dez apresentados, continham inconsistências e, se estivessem corretos, quais os respectivos códigos deveriam ser implementados. Ao final do experimento, através da análise da corretude e da produtividade dos participantes, pôde-se observar que o esforço para a identificação das inconsistências é extremamente oneroso, e que, mesmo o esforço sendo alto, o grau de corretude na identificação das inconsistências e relacionamento com o código correspondente é muito baixo. O resultado deste experimento demonstrou que a detecção manual de inconsistências em modelos UML é uma atividade difícil, que exige um grande nível de esforço e conhecimento, além de ser propensa a inconsistências sintáticas e semânticas.

Sendo assim, com base na técnica apresentada no Capitulo 4, foi implementada uma aplicação que, apoiando-se no uso de métricas, foi capaz de realizar a detecção de inconsistências sintáticas e semânticas, bem como, a identificação de boas práticas de padrões de software em modelos UML multivisão. O objetivo disto foi verificar se, de fato, a utilização de métricas pode auxiliar na detecção de inconsistências em modelos UML, de forma que possa tornar-se um padrão para qualquer metodologia de detecção de inconsistências em modelos UML. Para avaliar o desempenho da aplicação, foi realizada a segunda etapa do experimento controlado, no qual os participantes utilizaram a aplicação para detecção das inconsistências nos mesmos modelos, de forma foi possível, mesmo que não comprovado estatisticamente, aumentar um pouco a produtividade e, comprovado estatisticamente o aumento na corretude na interpretação dos participantes, comprovando a

melhora na qualidade do desenvolvimento do software utilizando as técnicas e métricas propostas.

Portanto, as principais contribuições deste estudo foram: (1) a identificação de diferentes níveis de gravidade por tipo de inconsistências presentes nos modelos UML multivisão e suas combinações para a qualidade e produtividade do desenvolvimento de software; (2) o desenvolvimento de heurísticas que, através da utilização de métricas de modelagem de software, são capazes de detectar diferentes tipos de inconsistências em modelos UML que utilizem Diagramas de Classe e Diagramas de Sequência; (3) a implementação de uma ferramenta capaz de executar cada uma das heurísticas apresentadas em forma de algoritmos computacionais; e (4) a validação estatística sobre os resultados da utilização de uma ferramenta de apoio à identificação e interpretação de inconsistências em modelos UML multivisão.

6.1 Limitações e Trabalhos Futuros

Foram identificadas as lacunas na literatura atual e exploradas oportunidades de pesquisa. Foi elaborado um catálogo de inconsistências possíveis em modelos UML, e identificadas métricas para detecção de inconsistências. Além disso, foi executado um experimento controlado para avaliar a qualidade e a produtividade da detecção de inconsistências em modelos UML de forma manual e através da utilização da ferramenta DIUML. Com a conclusão deste trabalho, puderam ser elencadas duas oportunidades para trabalhos futuros:

- **Permitir a configuração de novas Heurísticas:** trabalhar na implementação de uma funcionalidade na ferramenta que possibilite a importação de novas heurísticas para detecção de diferentes tipos de inconsistências não previstas neste estudo, assim como, na definição de um padrão para definição destas novas heurísticas.
- **Integrar com a Ferramenta MoCoTo:** integrar a aplicação desenvolvida como um componente à Ferramenta MoCoTo (FARIAS, 2015). Esta ferramenta propõe uma arquitetura flexível, baseada em componentes para ajudar na composição de modelos, considerando características relevantes relacionadas a questões de design. A integração da aplicação proposta como um componente ao MoCoTo permitirá a detecção de possíveis inconsistências resultantes da composição de modelos.

REFERÊNCIAS

AA, H. V. D.; LEOPOLDA, H.; REIJERS, H. A. **Comparing textual descriptions to process models The automatic detection of inconsistencies**. In Information Systems, 2016.

Astah Pro. Disponível em: http://www.astah.net/editions/professional. Acesso em 20/06/2016.

BASILI, V.; CALDIERA, G.; ROMBACH, H.. **The Goal Question Metric Paradigm**. Encyclopedia of Software Engineering. Wiley, New York, 1994.

BEZERRA, E. **Princípios de análise e projeto de sistemas com UML**. Rio de Janeiro, RJ: Campus, 2007.

BISCHOFF, V.; FARIAS, K.; GONÇALES, L.; WEBER, V. **Towards an Architecture for Integration of Feature Models.** International Journal of Computer Science and Software Engineering (IJCSSE), Volume 5, Issue 12, p. 265-272, December, 2016.

BOOCH, G.; RUMBAUGH, J.; JACOBSON, I. **UML: guia do usuário: o mais avançado tutorial sobre Unified Modeling Language (UML), elaborado pelos próprios criadores da linguagem**. Rio de Janeiro, RJ: Elsevier, 2005.

BUDGEN, D.; TURNER, M.; BRERETON, P.; KITCHENHAM, B. **Using Mapping Studies in Software Engineering.** Proceedings of PPIG, p. 195-204, Lancaster University, 2008.

CHANVILAI, S.; HONDA, K.; NAKAGAWA, H.; TAHARA, Y.; OHSUGA, A. **Goal-oriented approach to creating class diagrams with OCL constraints.** Symposium on Applied Computing, Riva del Garda (Trento), Italy, 2012.

CHAUDRON, M.; KATUMBA, B.; RAN, X. **Automated Prioritization of Metrics-Based Design Flaws in UML Class Diagrams**. EUROMICRO-SEAA, p. 369-376, 2014.

CHAUDRON, M.; HEIJSTEK, W.; NUGROHO, A. **How effective is UML modeling? An empirical perspective on costs and benefits**. Software System Model, cap. 11, p. 571–580, 2012.

CHAVEZ, H.; SHEN, W. **Finding inconsistency for UML-based composition at program level**. International Conference on Software Engineering, Vancouver, British Columbia, Canada, 2009.

CHONG, H., ZHANG, R.; QIN, Z. **Composite-based Conict Resolution in Merging Versions of UML Models**. In 17th IEEE/ACIS International Conference on Software Engineering (SNPD16), 2016.

DUBAUSKAITE, R.; VASILECAS, O. **The approach of ensuring consistency of UML model based on rules**. International Conference on Computer Systems and Technologies, Dublin, Ireland, 2010.

DZIDEK, W. J.; ARISHOLM, E.; BRIAND, L. C. **A Realistic Empirical Evaluation of the Costs and Benefits of UML in Software Maintenance**. IEEE Trans. Software Eng. 34(3), p. 407-432, 2008.

EGYED, A. **Automatically Detecting and Tracking Inconsistencies in Software Design Models**. In IEEE Transactions on Software Engineering, volume 37, p. 188-204, 2011.

EGYED, A. **Fixing Inconsistencies in UML Design Models**. ICSE, p. 292-301, 2007.

EGYED, A. **UML/Analyzer: A Tool for the Instant Consistency Checking of UML Models**. International Conference on Software Engineering, Minneapolis, Minnesota, USA, 2007.

EGYED, A.; LETIER, E.; FINKELSTEIN, A. **Generating and Evaluating Choices for Fixing Inconsistencies in UML Design Models**. 22nd International Conference on Automated Software Engineering (ASE), L'Aquila, Italy, 2008.

EGYED, A.; LETIER, E.; FINKELSTEIN, A. **Generating and Evaluating Choices for Fixing Inconsistencies in UML Design Models.** 22nd International Conference on Automated Software Engineering (ASE), L'Aquila, Italy, 2008.

EKANAYAKE, E. M. N. K.; KODITUWAKKU, S. R. **Consistency checking of UML class and sequence diagrams**. In 8th International Conference on Ubi-Media Computing (UMEDIA15), 98-103, 2015.

Enterprise Architect. Disponível em: http://www.sparxsystems.com/products/ea/. Acesso em 20/06/2016.

Epsilon. Disponível em: http://www.eclipse.org/gmt/epsilon/. Acesso em 15 de março de 2016.

FARIAS, K, **ComModeloS Composição de Modelos de Software: Métricas e Estudos Experimentais**. PIPCA, University of Vale do Rio dos Sinos - UNISINOS, MCT/CNPq N º 14/2013 – Universal, 2013.

FARIAS, K. **Empirical Evaluation of Effort on Composing Design Models**. PhD Thesis, Department of Informatics, Pontifical Catholic University of Rio de Janeiro (PUC-Rio), Rio de Janeiro, Brazil, 2012.

FARIAS, K., GARCIA A.; LUCENA, C. **Evaluating the Effects of Stability on Model Composition Effort: an Exploratory Study.** Proceedings of 8th Experimental Software Engineering Latin American Workshop (ESELAW'11), p. 77-86, Rio de Janeiro, Brazil, 2011.

FARIAS, K.; BREITMAN, K.; OLIVEIRA, T. **Ontology Aided Model Comparison**. 14th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'09), p. 78-83, Potsdam, Germany, 2009.

FARIAS, K.; GARCIA, A.; LUCENA, C. **Effects of stability on model composition effort: an exploratory study.** Software & Systems Modeling, 13(4), p. 1473-1494, 2014.

FARIAS, K.; GARCIA, A.; LUCENA, C. **Evaluating the Impact of Aspects on Inconsistency Detection Effort: A Controlled Experiment**. Proceedings of the 15th International Conference on Model-Driven Engineering Languages and Systems (MODELS'12), Vol. 7590, p. 219-234, Innsbruck, Austria, 2012.

FARIAS, K.; GARCIA, A.; WHITTLE, J.; CHAVEZ, C.; LUCENA, C. **Evaluating the Effort of Composing Design Models: A Controlled Experiment.** 15th International Conference on

Model-Driven Engineering Languages and Systems (MODELS'12), Vol. 7590, p. 676-691, Innsbruck, Austria, 2012.

FARIAS, K.; GARCIA, A.; WHITTLE, J.; LUCENA, C. **Analyzing the Effort of Composing Design Models of Large-Scale Software in Industrial Case Studies**. 16th International Conference on Model-Driven Engineering Languages and Systems (MODELS'13), p. 639-655, Miami, USA, September 2013.

FARIAS, K.; OLIVEIRA, T. **A Guidance for Model Composition**. 2nd International Conference on Software Engineering Advances (ICSEA'07), p. 27-33, French Riviera, France, 2007.

FARIAS, K.; OLIVEIRA, T. **Model Comparison: A Strategy-Based Approach**. 20th International Conference on Software Engineering and Knowledge Engineering (SEKE'08), Vol. 20, p. 912-917, San Francisco, USA, 2008.

FARIAS, K; GARCIA, A.; WHITTLE, J. **On the Quantitative Assessment of Class Model Compositions: An Exploratory Study.** Empirical Studies of Model-Driven Engineering (ESMDE'08) co-located MODELS'08, Vol. 1, p. 1-10, Toulouse, France, 2008.

FARIAS, K; GARCIA, A.; WHITTLE, J; CHAVEZ, C.; LUCENA, C. **Evaluating the Effort of Composing Design Models: A Controlled Experiment**. Journal on Software and Systems Modeling, p. 1-17, 2015.

FERNÁNDEZ-SÁEZA, A.; GENEROB, M.; CHAUDRON, M. **Empirical studies concerning the maintenance of UML diagrams and their use in the maintenance of code: A systematic mapping study**. Information and Software Technology, v. 55, Issue 7, p. 1119–1142, 2013.

FRANCE, R. **Model-driven development of complex software: a research roadmap**. Future of Software Engineering at ICSE'07, p. 37-54, 2007.

GENERO, M.; PIATTINI, M.; CHAUDRON, M. **Quality of UML models," Information & Software Technology**, p. 1629-1630, 2009.

IBM Rational Software Architect. Disponível em: http://www.ibm.com/developerworks/rational/products/rsa/. Acesso em 14 de março de 2016.

KERIEVSKY, J. **Refatoração para padrões**. Porto Alegre, RS: Bookman, 2008.

KIENZLE, J.; ABED, W.; KLEIN, J. **Aspect-oriented multi-view modeling.** International Conference on Aspect-Oriented Software Development, Charlottesville, VA, US, 2009.

KITCHENHAM, B.; AL-KHILIDAR, H; BABAR, M. A.; BERRY, M.; COX K.; KEUNG, J.; KURNIAWATI, F.; STAPLES, M.; ZHANG, H.; ZHU, L. **Evaluating Guidelines for Reporting Empirical Software Engineering Studies**. Empirical Software Engineering 13(1), 97–112, 2008.

KITCHENHAM, B.; BRERETON, P.; BUDGEN, D. **The educational value of mapping studies of software engineering literature**. 32nd Int. Conf. on Software Engineering, New York, NY, USA, 2010.

KITCHENHAM, B.; CHARTERS, S. **Guidelines for Performing Systematic Literature Reviews in Software Engineering.** Technical Report EBSE-2007-01, Keele University, 2007.

KOTB, Y. **Improving the UML consistency using Text Semantic Similarity approach**. International Conference on Computer Technology and Development, Cairo, Egypt, 2010.

LANGE, C. **Improving the quality of UML models in practice.** International Conference on Software Engineering, Shanghai, China, 2006.

LANGE, C., CHAUDRON, M. **An empirical assessment of completeness in UML design.** 8th Empirical Assessment in Software Engineering, p. 111–121, 2004.

LANGE, C.; CHAUDRON, M. **Effects of Defects in UML Models - An Experimental Investigation**. Proceedings of the 28th International Conference on Software engineering, p. 401-411, 2006.

LANGE,C.; CHAUDRON, M.; MUSKENS, J.; SOMERS, L.; DORTMANS, H. **An Empirical Investigation in Quantifying Inconsistency and Incompleteness of UML Designs**. 2nd workshop on Consistency Problems in UML-based software development, 2003.

LARMAN, C. **Utilizando UML e padrões: uma introdução à análise e ao projeto orientados a objetos e ao desenvolvimento iterativo**. 3ª Edição, Porto Alegre, RS: Bookman, 2007.

LAVANYA, K.; BALA, K.; MOHANTY, H.; SHYAMASUNDAR, R. **How Good is a UML Diagram? A Tool to Check It**. TENCON, Melbourne, Australia, 2005.

LEVINE, D.; RAMSEY, P.; SMIDT, R. **Applied Statistics for Engineers and Scientists**. Duxbury, 1999.

LIU, X. **Identification and Check of Inconsistencies between UML Diagrams**. International Conference on Computer Sciences and Applications, Wuhan, China, 2013.

LIU, X.; LILIUS, J. **Checking compositions of UML sequence diagrams for timing inconsistency**. Software Engineering Conference, Asia-Pacific, 2000.

LOPEZ-HERREJON, R.; EGYED, A. **Detecting Inconsistencies in Multi-view Models with Variability**. ECMFA, 2010.

LOPEZ-HERREJON, R.; EGYED, A. **Towards fixing inconsistencies in models with variability**. International Workshop on Variability Modeling of Software-Intensive Systems, Leipzig , Germany, 2012.

MALGOUYRES, H.; MOTET, G. **A UML model consistency verification approach based on meta-modeling formalization**. Symposium on Applied Computing, Dijon, France, 2006.

NOHRER, A.; REDER, A.; EGYED, A. **Positive effects of utilizing relationships between inconsistencies for more effective inconsistency resolution**. NIER track. Software Engineering (ICSE), 33rd International Conference, p. 864–867, 2011.

NUGROHO, A.; FLATON, B.; CHAUDRON, M. **Empirical Analysis of the Relation between Level of Detail in UML Models and Defect Density**. 11th International Conference on Model Driven Engineering Languages and Systems (MODELS), p. 600-614, 2008.

PETERSEN, K.; FELDT, R.; MUJTABA, S.; MATTSSON, M. **Systematic mapping studies in software engineering**. 2nd International Conference on Evaluation and Assessment in Software Engineering, p. 68-77, 2008.

PETERSEN, K.; VAKKALANKA, S.; KUZNIARZ, L. **Guidelines for conducting systematic mapping studies in software engineering: An update**. Information and Software Technology, vol. 64, p. 1–18, 2015.

REDER, A.; EGYED, A. **Computing repair trees for resolving inconsistencies in design models**. International Conference on Automated Software Engineering, Essen, Germany, 2012.

REDER, A.; EGYED, A. **Determining the Cause of a Design Model Inconsistency**. IEEE Transactions on Software Engineering, volume 39, p. 1531-1548, 2013.

REDER, A.; EGYED, A. **Incremental consistency checking for complex design rules and larger model changes**. MODELS'12 Proceedings of the 15th international conference on Model Driven Engineering Languages and Systems, p. 202-218, 2012.

REDER, A.; EGYED, A. **Model/analyzer: a tool for detecting, visualizing and fixing design errors in UML**. 25th International Conference on Automated Software Engineering (ASE), Antwerp, Belgium, 2010.

SATISH, S.; SHASHIKANT, S.; SAMBHE, V.; SHELKE, R.; KOCHAREKAR, G. **A minimum cardinality consistency-checking algorithm for UML class diagrams**. International Conference and Workshop on Emerging Trends in Technology, Mumbai, Maharashtra, India, 2010.

SDMetrics. Disponível em: http://www.sdmetrics.com/. Acesso em 20 de março de 2016.

SIMMONDS, J.; Bastarrica, M. **A tool for automatic UML model consistency checking**. International Conference on Automated Software Engineering, Long Beach, California, USA, 2005.

THUNG, F.; LO, D.; OSMAN, M.; CHAUDRON, M. **Condensing class diagrams by analyzing design and network metrics using optimistic classification**. ICPC, p. 110-121, 2014.

WEBER, V.; FARIAS, K.; GONÇALES, L.; BISCHOFF, V. **Detecting Inconsistencies in Multi-view UML Models.** International Journal of Computer Science and Software Engineering (IJCSSE), Volume 5, Issue 12, p. 260-264, December, 2016.

WOHLIN, C.; RUNESON, P.; HÖST, M.; OHLSSON, M.; REGNELL, B.; WESSLÉN, A. **Experimentation in Software Engineering: An Introduction, Kluwer Academic**. Publishers, Norwell, MA, USA, 2000.

WOHLIN, C.; RUNESON, P.; HÖST, M.; OHLSSON, M.; REGNELL, B.; WESSLÉN, A. **Experimentation in Software Engineering**. eBook Springer Heidelberg, New York, Dordercht, London, 2012.

APÊNDICE A - LISTA DOS ESTUDOS PRIMÁRIOS SELECIONADOS

ID Estudo	Ano de Publicação	Autores	Título do Estudo
S01	2000	Lilius, J.; Xuandong Liu	Checking compositions of UML sequence diagrams for timing inconsistency
S02	2003	C. Lange; M.R.V. Chaudron	An Empirical Investigation in Quantifying Inconsistency and Incompleteness of UML Designs
S03	2005	Jocelyn Simmonds; M. Cecilia Bastarrica	A tool for automatic UML model consistency checking
S04	2005	Bala, K.V.; Mohanty, H.; Shyamasundar, R.K; Lavanya, K.C	How Good is a UML Diagram? A Tool to Check It
S05	2006	C. Lange; M.R.V. Chaudron	Effects of Defects in UML Models - An Experimental Investigation
S06	2006	H. Malgouyres; G. Motet	A UML model consistency verification approach based on meta-modeling formalization
S07	2006	Christian F. J. Lange	Improving the quality of UML models in practice
S08	2007	Alexander Egyed	Fixing Inconsistencies in UML Design Models
S09	2007	Alexander Egyed	UML/Analyzer: A Tool for the Instant Consistency Checking of UML Models
S10	2008	Alexander Egyed	Generating and Evaluating Choices for Fixing Inconsistencies in UML Design Models
S11	2009	Jörg Kienzle; Wisam Al Abed ; Jacques Klein	Aspect-oriented *multi-view* modeling
S12	2009	Chavez, H.M. ; Wuwei Shen	Finding inconsistency for UML-based composition at program level
S13	2010	Alexander Reder; Alexander Egyed	Model/analyzer: a tool for detecting, visualizing and fixing design errors in UML
S14	2010	Roberto Erick Lopez-Herrejon; Alexander Egyed	Detecting Inconsistencies in *Multi-view* Models with Variability
S15	2010	Route Dubauskaite; Oleg Vasilecas	The approach of ensuring consistency of UML model based on rules
S16	2010	S. Salunkhe Satish; S. Radke Shashikant; V. K. Sambhe; Rahul B. Shelke; Ganesh Kocharekar	A minimum cardinality consistency-checking algorithm for UML class diagrams
S17	2010	Kotb, Y	Improving the UML consistency using Text Semantic Similarity approach
S18	2011	Alexander Egyed	Automatically Detecting and Tracking Inconsistencies in Software Design Models
S19	2011	Alexander Nöhrer; Alexander Egyed	Positive effects of utilizing relationships between inconsistencies for more effective inconsistency resolution
S20	2012	Alexander Reder; Alexander Egyed	Computing Repair Trees for Resolving Inconsistencies in Design Models

S21	2012	Alexander Reder; Alexander Egyed	Incremental consistency checking for complex design rules and larger model changes
S22	2012	Sombat Chanvilai; Kozo Honda; Hiroyuki Nakagawa; Yasuyuki Tahara; Akihiko Ohsuga	Goal-oriented approach to creating class diagrams with OCL constraints
S23	2012	Roberto E. Lopez-Herrejon Alexander Egyed	Towards fixing inconsistencies in models with variability
S24	2013	Alexander Reder; Alexander Egyed	Determining the Cause of a Design Model Inconsistency
S25	2013	Ariadi Nugroho; M.R.V. Chaudron	The impact of UML modeling on defect density and defect resolution time in a proprietary system
S26	2013	Xianhong Liu	Identification and Check of Inconsistencies between UML Diagrams
S27	2014	Ferdian Thung; M.R.V. Chaudron	Condensing class diagrams by analyzing design and network metrics using optimistic classification
S28	2014	M.R.V. Chaudron	Automated Prioritization of Metrics-Based Design Flaws in UML Class Diagrams
S29	2015	Ekanayake, E. M. N. K., & Kodituwakku, S. R.	Consistency checking of UML class and sequence diagrams.
S30	2016	Aa, H. V. D., Leopolda, H., & Reijers, H. A.	Comparing textual descriptions to process models The automatic detection of inconsistencies.
S31	2016	Chong, H., Zhang, R. &Qin, Z	Composite-based Conflict Resolution in Merging Versions of UML Models.

Fonte: Elaborado pela autora.

APÊNDICE B - QUESTIONÁRIO DO EXPERIMENTO 1ª FASE

Hora de Início:_____:_____ **Hora de Término:** _____:_____

QUESTIONÁRIO - ENGENHARIA DE SOFTWARE

Nome (opcional):			
Idade:		**Sexo:**	( ) Masculino ( ) Feminino
Profissão:		**Empresa em que Trabalha:**	
Cargo atual:		**Quanto tempo está neste cargo:**	

Escolaridade:		Técnico	**Formação acadêmica:**		Sistemas de Informação
		Graduação			Ciência da computação
		Mestrado			Engenharia da Computação
		Doutorado			Análise de Sistemas
		Outra. Qual?			Outro. Qual?

Por quanto você estudou/tem estudado em universidades?		Menos de 2 anos	**Seu cargo atual se encaixa em qual especialidade?**		Programador
		De 2 a 4 anos			Analista
		De 5 a 6 anos			Arquiteto
		De 7 a 8 anos			Gerente
		Mais de 8 anos			Outro: Qual?

Quanto tempo de experiência você tem em desenvolvimento de software?		Menos de 2 anos	**Quanto tempo de experiência você tem em modelagem de software?**		Menos de 2 anos
		De 2 a 4 anos			De 2 a 4 anos
		De 5 a 6 anos			De 5 a 8 anos
		De 7 a 8 anos			De 7 a 8 anos
		Mais de 8 anos			Mais de 8 anos

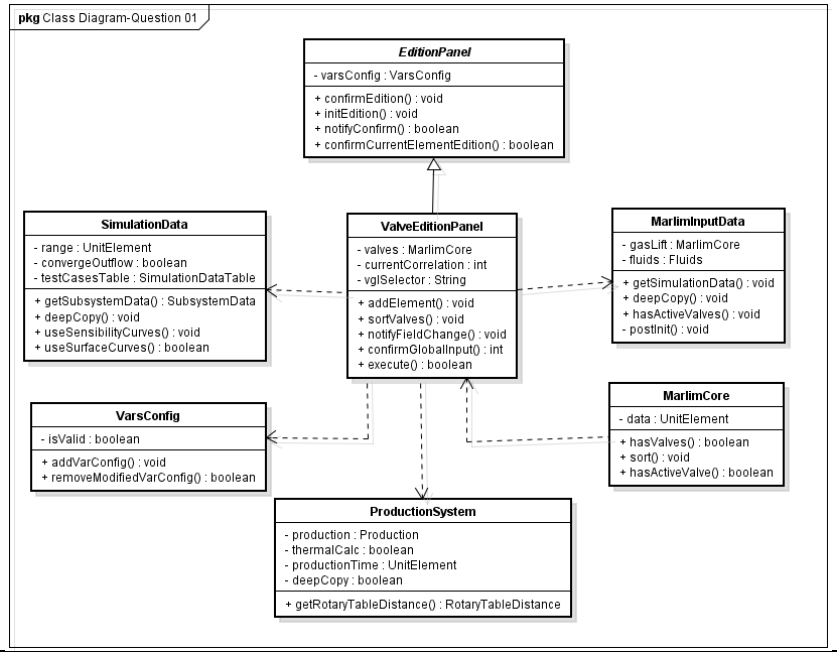
Este questionário tem o objetivo de avaliar os índices de identificação de alguns tipos de problemas relacionados à multimodelos UML para identificação de impactos na qualidade e produtividade de desenvolvimento. Desse modo, as respostas para as questões abaixo devem ser baseadas na experiência do participante. O questionário possui duas partes, uma para caracterizar o participante e a outra parte com as questões referente aos interesses da pesquisa. Os participantes não estão sendo avaliados e seus dados não serão divulgados. As questões pessoais acima servem somente para categorizar de acordo com a metodologia.

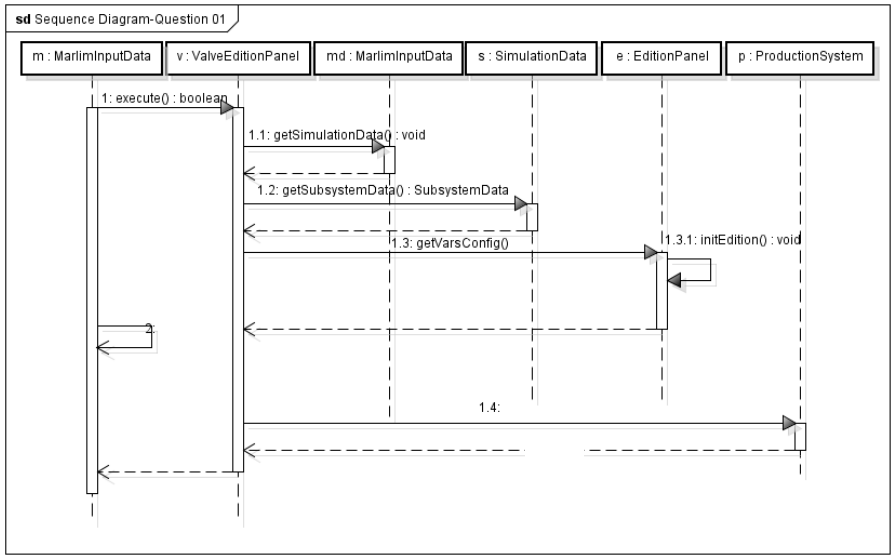
Obrigada pela participação!

Vanessa Weber – Mestranda em Computação Aplicada pela Unisinos.
Orientador Prof. Dr. Kleinner Farias

Hora de Início: _____:_____ **Hora de Término:** _____:_____

QUESTÃO 01: Suponha que você é um desenvolvedor do projeto XYZ, um software para simulação de extração de petróleo. Considerando o diagrama de classes e sequência acima, como você implementaria a classe ***ValveEditionPanel***?





A)
```
class ValveEditionPanel {
  boolean execute(){
    m.getSimulationData();
    s.getSubsystemData();
    e.getVarsConfig();
    md.hasActiveValves();
    p.getRotaryTableDistance()}}
```

B)
```
class ValveEditionPanel {
  boolean execute(){
    m.getSimulationData();
    s.getSubsystemData();
    e.getVarsConfig();
    md.hasActiveValves();
    p.getRotaryTableDistance()}}
```

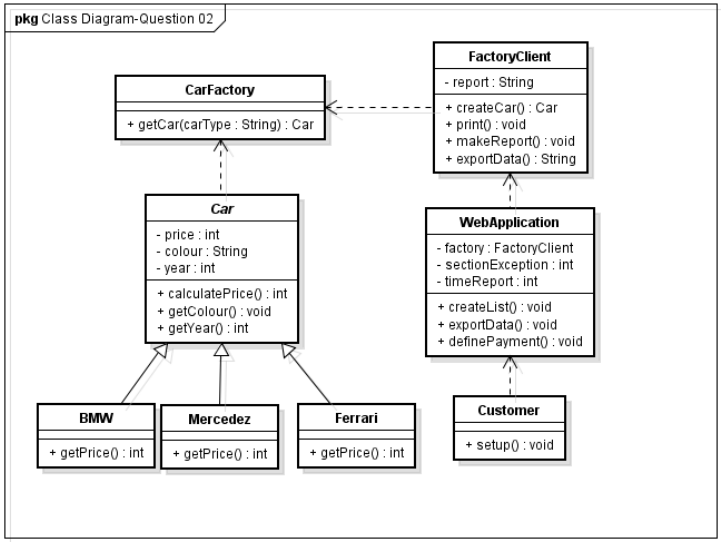
C)
```
class ValveEditionPanel {
  boolean execute(){
    m.getSimulationData();
    s.getSubsystemData();
    e.getVarsConfig();
    p.getRotaryTableDistance();}}
```

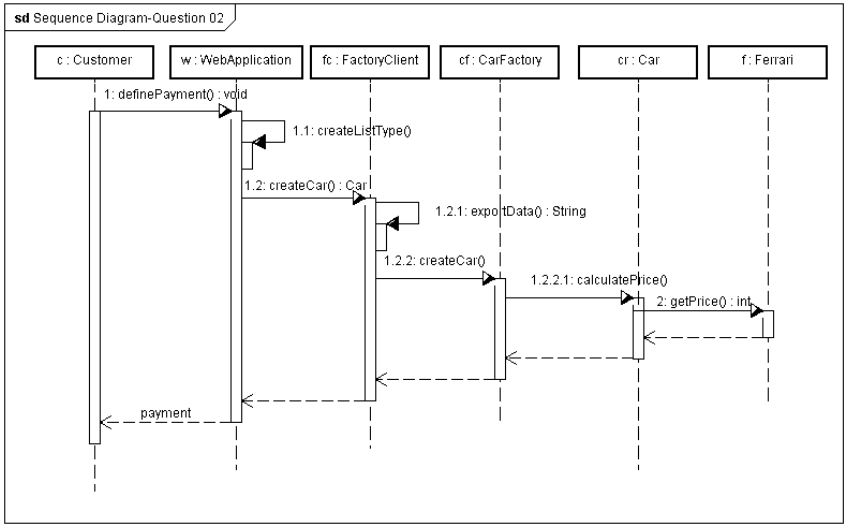
D)
```
class ValveEditionPanel {
  boolean execute(){
    m.getSimulationData();
    s.getSubsystemData();
    e.getVarsConfig();
    p.getRotaryTableDistance();}}
```

E) Nenhuma interpretação pode ser feita porque existe problema no modelo.

Hora de Início: _____:_____ **Hora de Término:** _____:_____

QUESTÃO 02: Suponha que você é um desenvolvedor de uma aplicação WEB para venda de carros. Considerando o diagrama de classe e sequência acima, como você implementaria a classe ***WebApplication***?





A)
```
class WebApplication {
   void definePayment(){
     w.createList();
     fc.createCar();
     // Do Something
     fc.print();}}
```

B)
```
class WebApplication {
   void definePayment(){
      w.createList();
      fc.createCar();
      // Do Something
      f.calculatePrice();}}
```

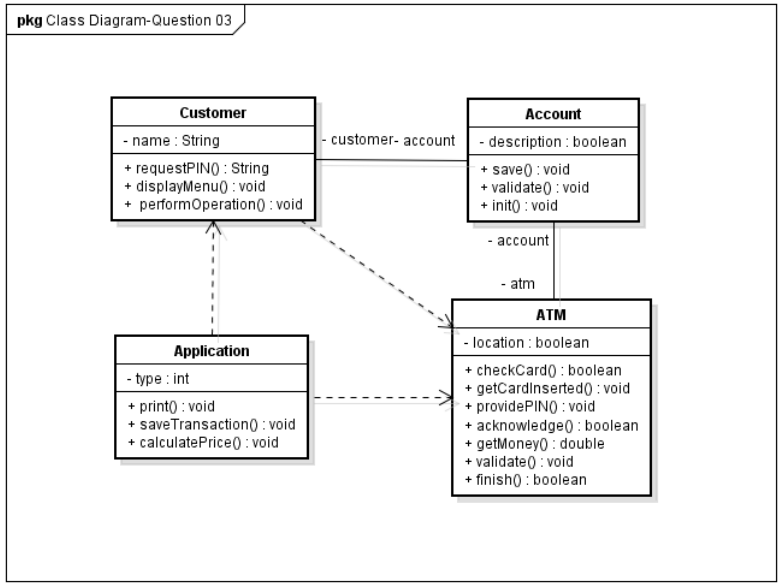
C)
```
class WebApplication {
   void definePayment(){
     w.createList();
     f.createCar();
     // Do Something}}
```

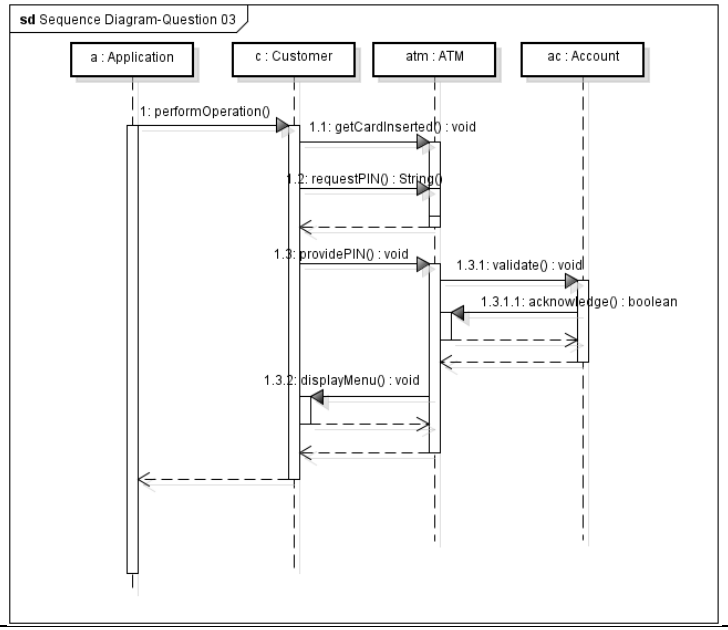
D)
```
class WebApplication {
   boolean definePayment (){
     w.createList();
     w.exportData();
     fc.createCar();
     // Do Something
     f.calculatePrice();}}
```

E) Nenhuma interpretação pode ser feita porque existe problema no modelo.

Hora de Início: _____:_____ **Hora de Término:** _____:_____

QUESTÃO 03: Suponha que você é um desenvolvedor de um projeto de uma instituição financeira. Considerando o diagrama de classe e sequência acima, como você implementaria a classe ***ATM***?





A)
```
class ATM {
     void getInsertedCard(){
        c.requestPIN()   }
     void providePIN(){
        ac.doOperation()
        //DoSomething
        c.displayMenu()  }}
```

B)
```
class ATM {
     void getInsertedCard(){
        c.requestPIN()   }
     void providePIN(){
        ac.validate()
        //DoSomething
        c.displayMenu()  }}
```

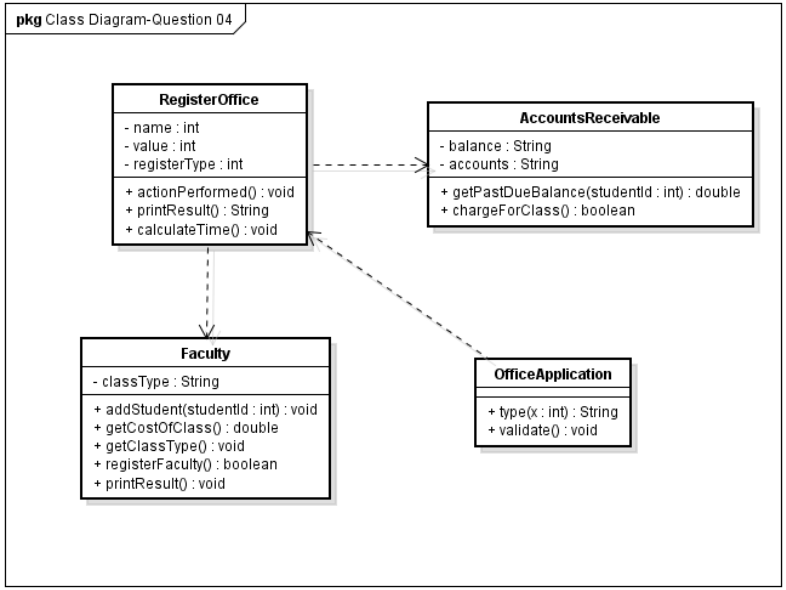
C)
```
class ATM {
     void getInsertedCard(){
        c.requestPIN()   }
     void providePIN(){
        //DoSomething
        c.displayMenu()  }}
```

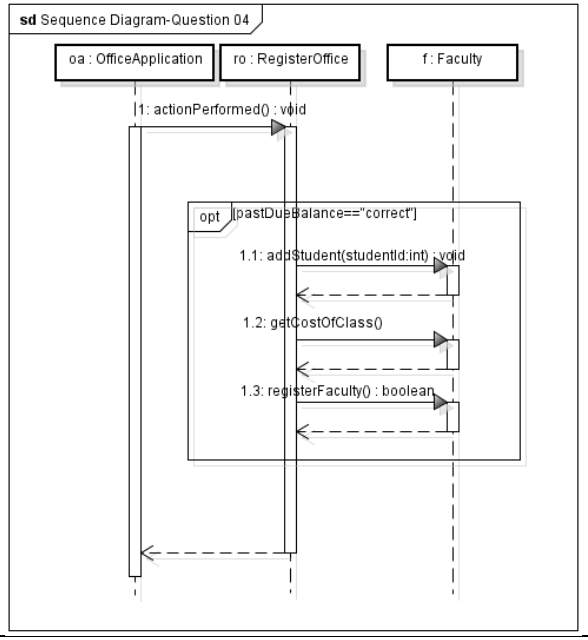
D)
```
class ATM {
     void getInsertedCard(){
        c.requestPIN()   }
     void providePIN(){
        ac.print()
        //DoSomething
        c.displayMenu()  }}
```

E) Nenhuma interpretação pode ser feita porque existe problema no modelo.

Hora de Início: _____:_____ **Hora de Término:** _____:_____

QUESTÃO 04: Suponha que você é um desenvolvedor de uma aplicação para uma faculdade. Considerando o diagrama de classe e sequência acima, como você implementaria a classe ***RegisterOffice***?





A)
```
class RegisterOffice {
    void actionPerformed(){
      ar.getPastDueBalance(student);
      f.addStudent(studentId);
      f.getCostOfClass();
      f.printResult();
      ar.chargeForClass();  } }
```

B)
```
class RegisterOffice {
    void actionPerformed(){
      ar.getPastDueBalance(student)
      if(pastDueBalance==”correct”){
         f.addStudent(studentId)
         f.getCostOfClass();
         f.registerFaculty();
         ro.chargeForClass();  }  } }
```

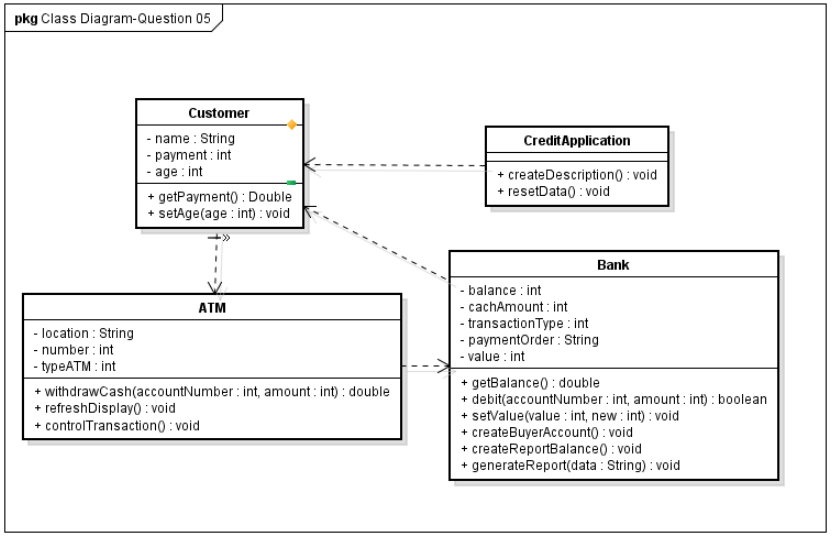
C)
```
class RegisterOffice {
    void actionPerformed(){
      ar.getPastDueBalance(student)
       if(pastDueBalance==”correct”){
          f.addStudent(studentId);
          f.getCostOfClass();
          f.registerFaculty();
          ar.chargeForClass();  }  } }
```

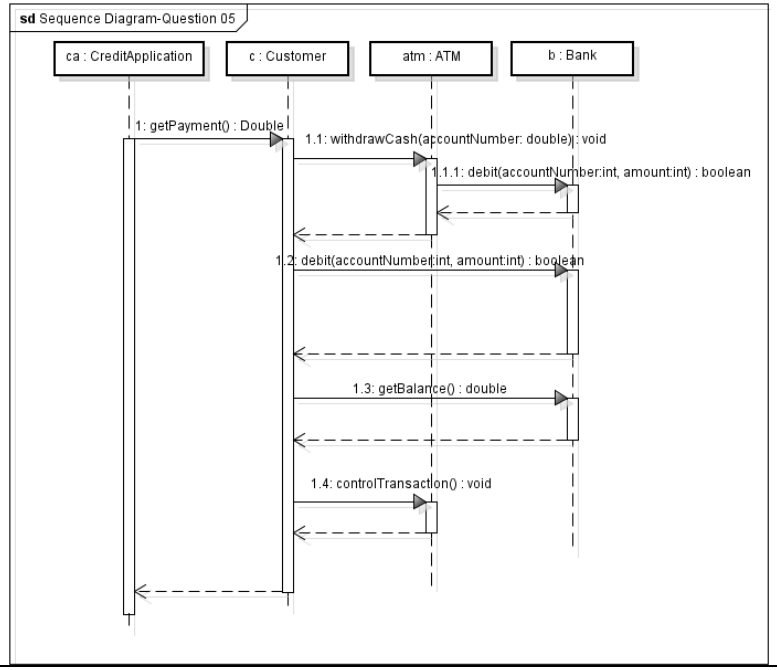
D)
```
class RegisterOffice {
    void actionPerformed(){
      ar.getPastDueBalance(student)  }
      if(pastDueBalance==”correct”){
          f.addStudent(studentId);
          f.getCostOfClass();
          f.registerFaculty();
          ar.chargeForClass();  }  } }
```

E) Nenhuma interpretação pode ser feita porque existe problema no modelo.

Hora de Início: _____:_____ **Hora de Término:** _____:_____

QUESTÃO 05: Suponha que você é um desenvolvedor de uma aplicação para uma instituição financeira. Considerando o diagrama de classe e sequência acima, como você implementaria a classe ***Customer***?





A)
```
class Customer {
    double getPayment(){
      atm.withdrawCash(accountNumber, amount);
      atm. refreshDisplay();
      atm.debit();
      b.getBalance();
      atm.controlTransaction();}
```

B)
```
class Customer {
    double getPayment(){
      atm.withdrawCash(accountNumber);
      b.debit(accountNumber, amount);
      b.getBalance();
      atm.controlTransaction();}}
```

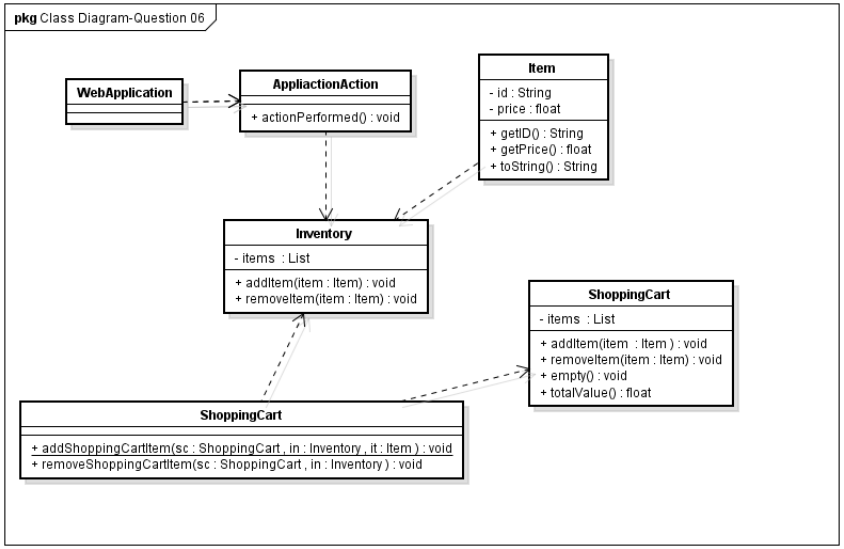
C)
```
class Customer {
    void getPayment(){
      atm.withdrawCash(accountNumber, amount);
      atm.debit();
      b.getBalance();
      atm.controlTransaction();}}
```

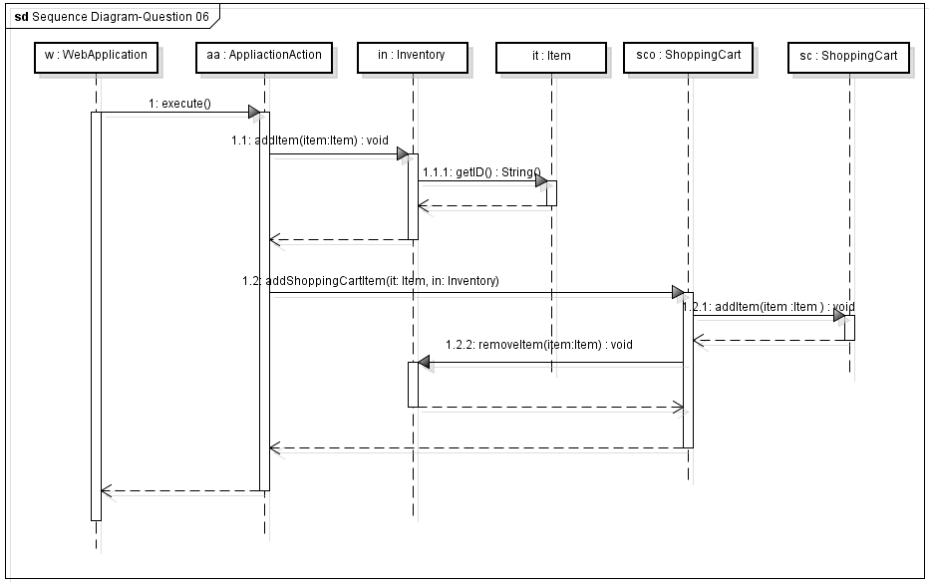
D)
```
Class Customer {
    void getPayment(){
      atm.withdrawCash(accountNumber, amount);
      atm.debit();
      atm.getBalance();
      b.controlTransaction();}}
```

E) Nenhuma interpretação pode ser feita porque existe problema no modelo.

Hora de Início: _____:_____ **Hora de Término:** _____:_____

QUESTÃO 06: Suponha que você é um desenvolvedor de uma aplicação para uma loja virtual. Considerando o diagrama de classe e sequência acima, como você implementaria a classe ***AplicationAction***?





A)
```
class ApplicationAction {
    execute(){
      in.addItem(item);
      it.getID();
      sco.addShoppingCartItem(item);
      sc.addItem(item);}
  }
```

B)
```
class ApplicationAction {
    execute(){
      in.addItem();
      it.getID();
      sco.addShoppingCartItem(item);
      sc.addItem(item);}
  }
```

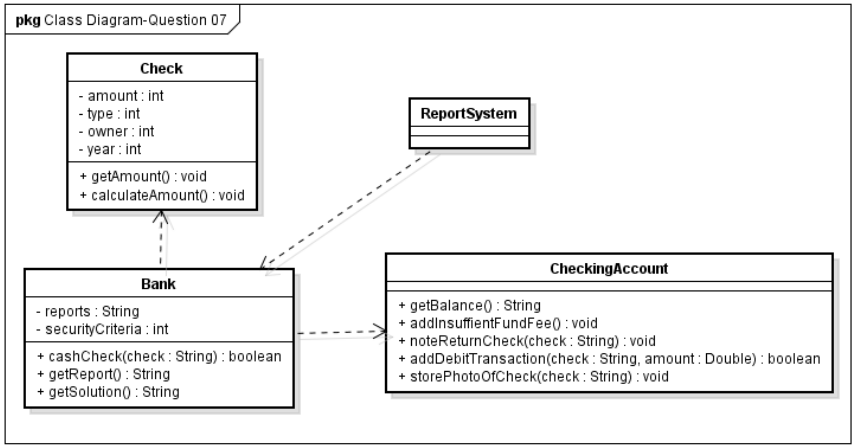
C)
```
class ApplicationAction {
    execute(){
      in.addItem(item);
      it.getID();
      sco.addShoppingCartItem(item);
      sco.addItem(item);}
  }
```

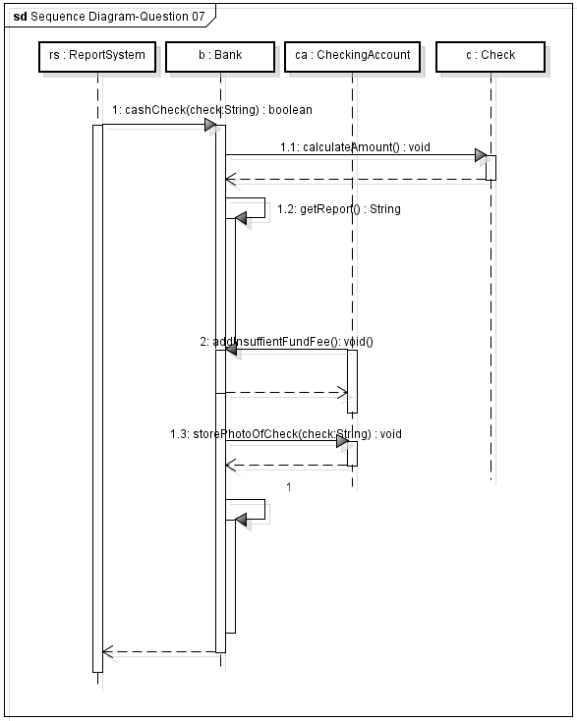
D)
```
class ApplicationAction {
    execute(){
      in.addItem(item);
      sco.addShoppingCartItem(item);
      sco.addItem(item);}
  }
```

E) Nenhuma interpretação pode ser feita porque existe problema no modelo.

Hora de Início: _____:_____ **Hora de Término:** _____:_____

QUESTÃO 07: Suponha que você é um desenvolvedor. Considerando o diagrama de classe e sequência acima, como você implementaria a classe ***Bank***?





A)
```
class Bank {
    boolean cashCheck (checkString){
      c.calculateAmount();
      b.getReport();
      ca. addInsuffientFundFee();
      ca. storePhotoOfCheck(check);}}
```

B)
```
class Bank {
    boolean cashCheck (checkString){
      c.calculateAmount();
      b.getReport();
      ca. storePhotoOfCheck(check);}}
```

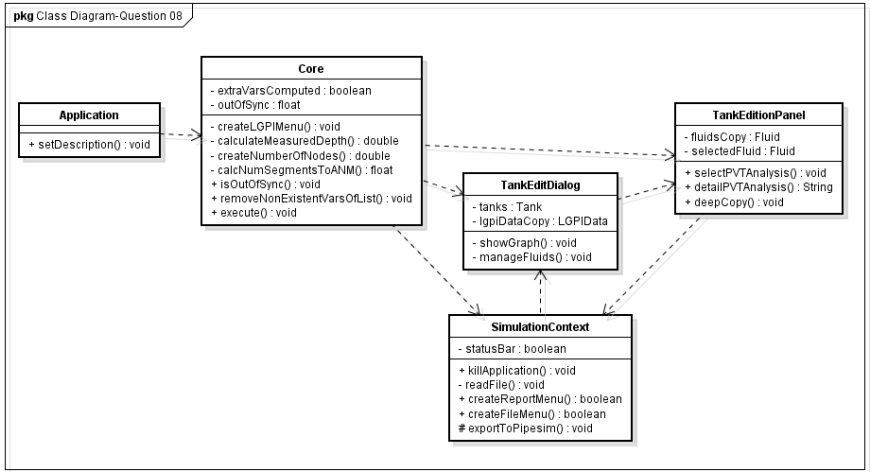
C)
```
class Bank {
    boolean cashCheck (checkString){
      c.calculateAmount();
      b.getReport(string);
      ca. addInsuffientFundFee();
      ca. storePhotoOfCheck(check);}}
```

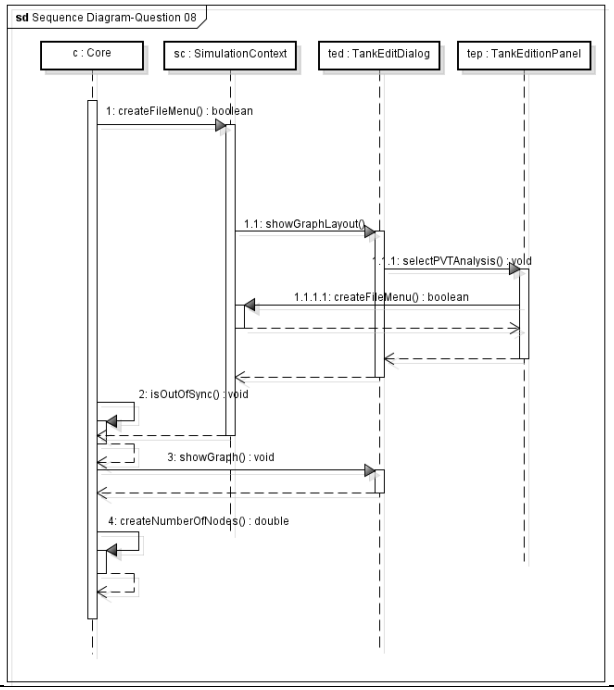
D)
```
class Bank {
    boolean cashCheck (checkString){
      c.calculateAmount();
      ca. addInsuffientFundFee();
      ca. storePhotoOfCheck(check);}}
```

E) Nenhuma interpretação pode ser feita porque existe problema no modelo.

Hora de Início: _____:_____ **Hora de Término:** _____:_____

QUESTÃO 08: Suponha que você é um desenvolvedor. Considerando o diagrama de classe e sequência acima, como você implementaria a classe ***SimulationContext***?





A)
```
class SimulationContext {
    boolean createFileMenu(){
      ted.showGraphLayout();
      tap.selectIPTVAnalysis();
      sc.createFileMenu();}}
```

B)
```
class SimulationContext {
    boolean createFileMenu(){
      ted.showGraphLayout();
      tap.selectIPTVAnalysis();
      sc.createFileMenu();
      c.showGraph();}}
```

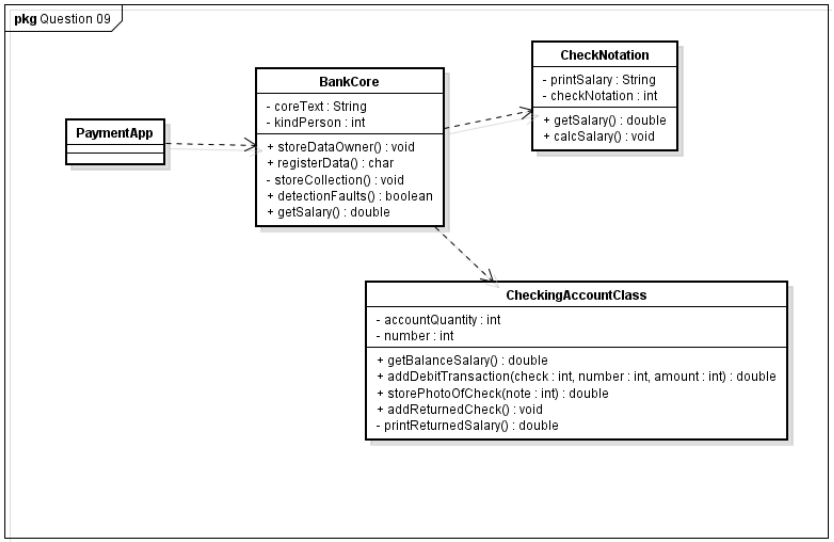
C)
```
class SimulationContext {
    boolean createFileMenu(){
      ted.showGraphLayout();
      tap.selectIPTVAnalysis();
      sc.createFileMenu();
      c.showGraph();
      c.createNumberOfNodes();}}
```

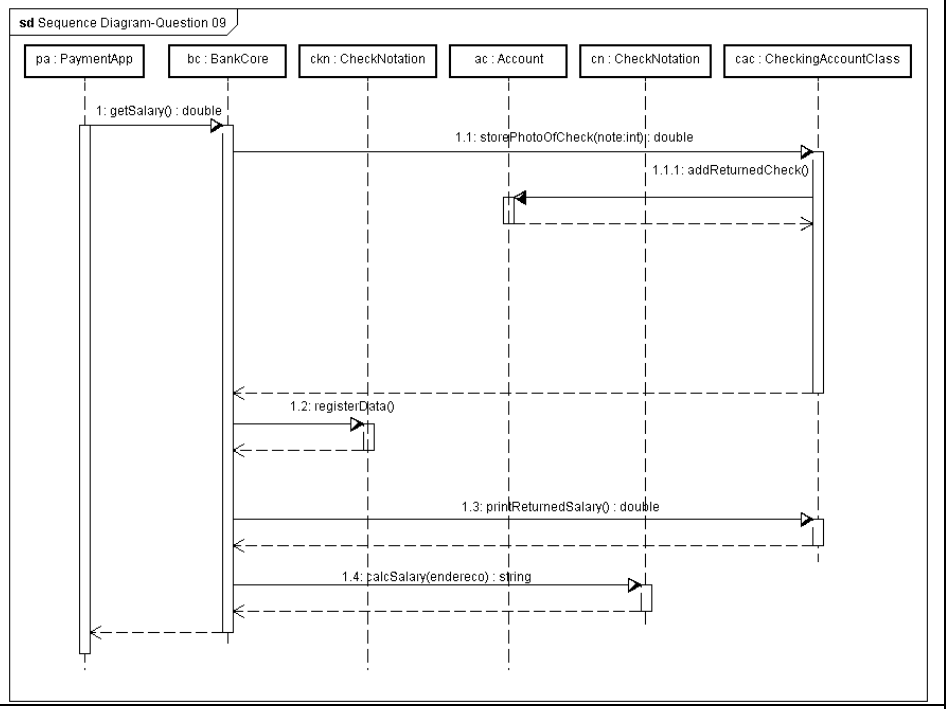
D)
```
class SimulationContext {
    boolean createFileMenu(){
      ted.showGraphLayout();
      tap.selectIPTVAnalysis();
      sc.createFileMenu();
      c.createNumberOfNodes();}}
```

E) Nenhuma interpretação pode ser feita porque existe problema no modelo.

Hora de Início: _____:_____ **Hora de Término:** _____:_____

QUESTÃO 09: Suponha que você é um desenvolvedor. Considerando o diagrama de classe e sequência acima, como você implementaria a classe ***BankCore***?





A)
```
class BankCore {
    double getSalary(){
      cac.storePhotoOfCheck(note);
      ckn.registerData();
      cac.printReturnedSalary();
      cn.calcSalary(endereco);}}
```

B)
```
class BankCore {
    double getSalary(){
      cac.storePhotoOfCheck(note);
      ac.addReturnedCheck();
      ckn.registerData();
      cac.printReturnedSalary();
      cn.calcSalary(endereco);}}
```

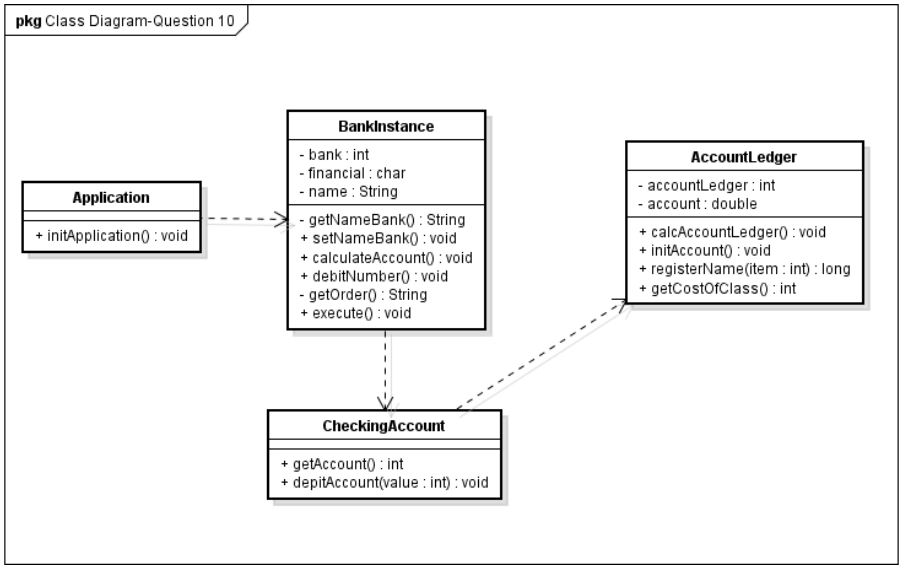
C)
```
class BankCore {
    double getSalary(){
      cac.storePhotoOfCheck(note);
      ac.addReturnedCheck();
      cac.printReturnedSalary();
      cn.calcSalary(endereco);}}
```

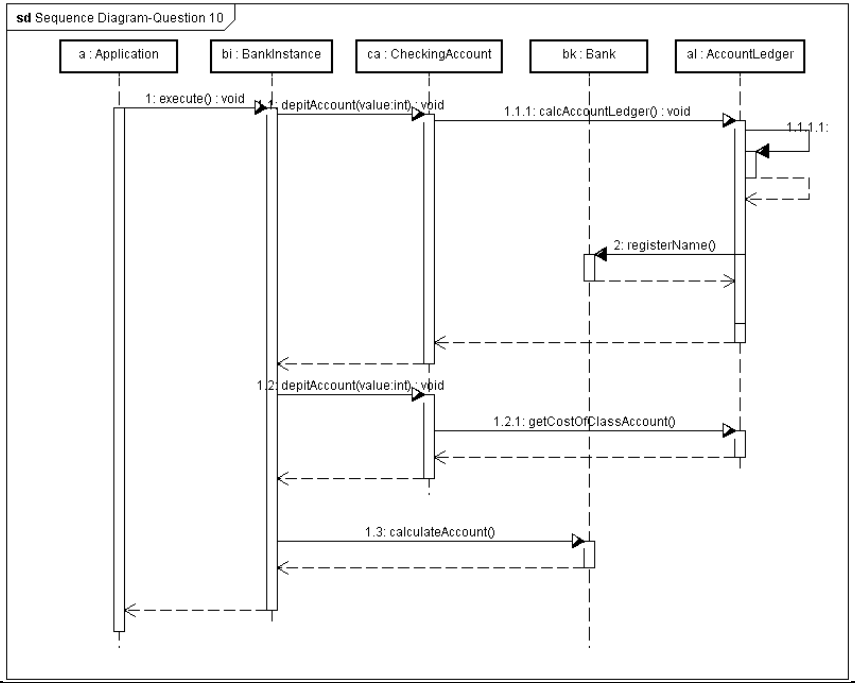
D)
```
class BankCore {
    get Salary(){
      cac.storePhotoOfCheck(note);
      ckn.registerData();
      cac.printReturnedSalary();
      cn.calcSalary(endereco);}}
```

E) Nenhuma interpretação pode ser feita porque existe problema no modelo.

Hora de Início: _____:_____ **Hora de Término:** _____:_____

QUESTÃO 10: Suponha que você é um desenvolvedor. Considerando o diagrama de classe e sequência acima, como você implementaria a classe ***BankInstance***?





A)
```
class BankInstance {
    void execute(){
      ca.depitAccount(value);
      at.registerName();
      ca.depitAccount(value);
      al.getCostClassAccount();
      bk.calculateAccount();}}
```

B)
```
class BankInstance {
    void execute(){
      ca.depitAccount(value);
      at.registerName();
      ca.depitAccount(value);
      bk.calculateAccount();}}
```

C)
```
class BankInstance {
    void execute(){
      ca.depitAccount(value);
      ca.depitAccount(value);
      al.getCostClassAccount();
      bk.calculateAccount();}}
```

D)
```
class BankInstance {
    execute(){
      ca.depitAccount(value);
      ca.depitAccount(value);
      bk.calculateAccount();}}
```

E) Nenhuma interpretação pode ser feita porque existe problema no modelo.

APÊNDICE C - QUESTIONÁRIO DO EXPERIMENTO 2ª FASE

Hora de Início:_____:_____ **Hora de Término:** _____:_____

QUESTIONÁRIO - ENGENHARIA DE SOFTWARE

Nome (opcional):			
Idade:		**Sexo:**	( ) Masculino ( ) Feminino
Profissão:		**Empresa em que Trabalha:**	
Cargo atual:		**Quanto tempo está neste cargo:**	

Escolaridade:		Técnico	**Formação acadêmica:**		Sistemas de Informação
		Graduação			Ciência da computação
		Mestrado			Engenharia da Computação
		Doutorado			Análise de Sistemas
		Outra. Qual?			Outro. Qual?

Por quanto você estudou/tem estudado em universidades?		Menos de 2 anos	**Seu cargo atual se encaixa em qual especialidade?**		Programador
		De 2 a 4 anos			Analista
		De 5 a 6 anos			Arquiteto
		De 7 a 8 anos			Gerente
		Mais de 8 anos			Outro: Qual?

Quanto tempo de experiência você tem em desenvolvimento de software?		Menos de 2 anos	**Quanto tempo de experiência você tem em modelagem de software?**		Menos de 2 anos
		De 2 a 4 anos			De 2 a 4 anos
		De 5 a 6 anos			De 5 a 8 anos
		De 7 a 8 anos			De 7 a 8 anos
		Mais de 8 anos			Mais de 8 anos

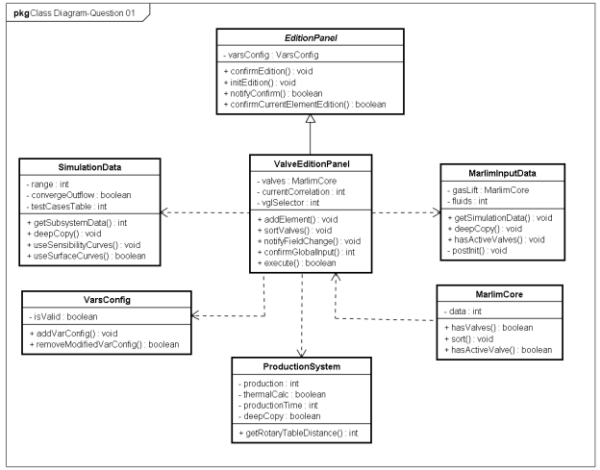
Este questionário tem o objetivo de avaliar os índices de identificação de alguns tipos de problemas relacionados à multimodelos UML para identificação de impactos na qualidade e produtividade de desenvolvimento. Desse modo, as respostas para as questões abaixo devem ser baseadas na experiência do participante. O questionário possui duas partes, uma para caracterizar o participante e a outra parte com as questões referente aos interesses da pesquisa. Os participantes não estão sendo avaliados e seus dados não serão divulgados. As questões pessoais acima servem somente para categorizar de acordo com a metodologia.

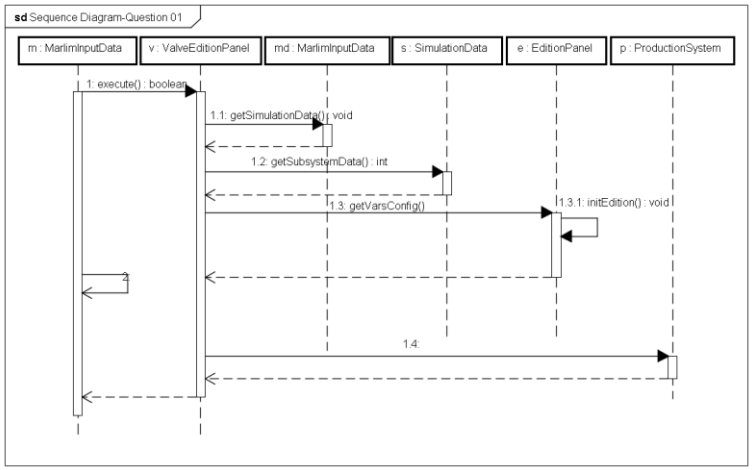
Obrigada pela participação!

Vanessa Weber – Mestranda em Computação Aplicada pela Unisinos.
Orientador Prof. Dr. Kleinner Farias

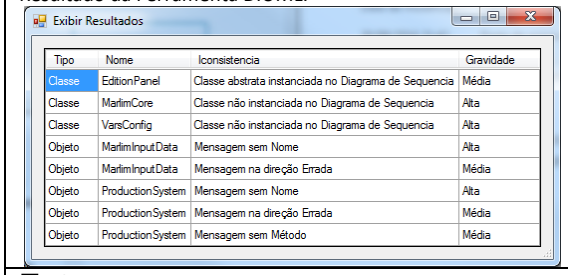
Hora de Início: hh : mm **Hora de Término:** hh : mm

QUESTÃO 01: Suponha que você é um desenvolvedor do projeto XYZ, um software para simulação de extração de petróleo. Considerando o diagrama de classes e sequência acima, como você implementaria a classe ***ValveEditionPanel***?





Resultado da Ferramenta DIUML:



Tipo	Nome	Iconsistencia	Gravidade
Classe	EditionPanel	Classe abstrata instanciada no Diagrama de Sequencia	Média
Classe	MarlimCore	Classe não instanciada no Diagrama de Sequencia	Alta
Classe	VarsConfig	Classe não instanciada no Diagrama de Sequencia	Alta
Objeto	MarlimInputData	Mensagem sem Nome	Alta
Objeto	MarlimInputData	Mensagem na direção Errada	Média
Objeto	ProductionSystem	Mensagem sem Nome	Alta
Objeto	ProductionSystem	Mensagem na direção Errada	Média
Objeto	ProductionSystem	Mensagem sem Método	Média

☐ **A)**
```
class ValveEditionPanel {
  boolean execute(){
    m.getSimulationData();
    s.getSubsystemData();
    e.getVarsConfig();
    md.hasActiveValves();
    p.getRotaryTableDistance()}}
```

☐ **B)**
```
class ValveEditionPanel {
  boolean execute(){
     m.getSimulationData();
     s.getSubsystemData();
     e.getVarsConfig();
     md.hasActiveValves();
     p.getRotaryTableDistance()}}
```

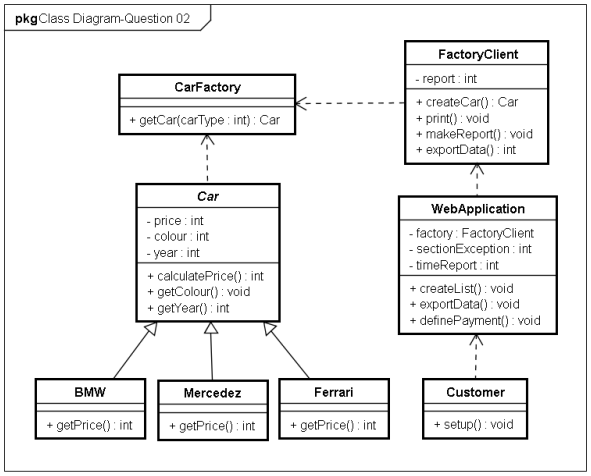
☐ **C)**
```
class ValveEditionPanel {
  boolean execute(){
     m.getSimulationData();
     s.getSubsystemData();
     e.getVarsConfig();
     p.getRotaryTableDistance();}}
```

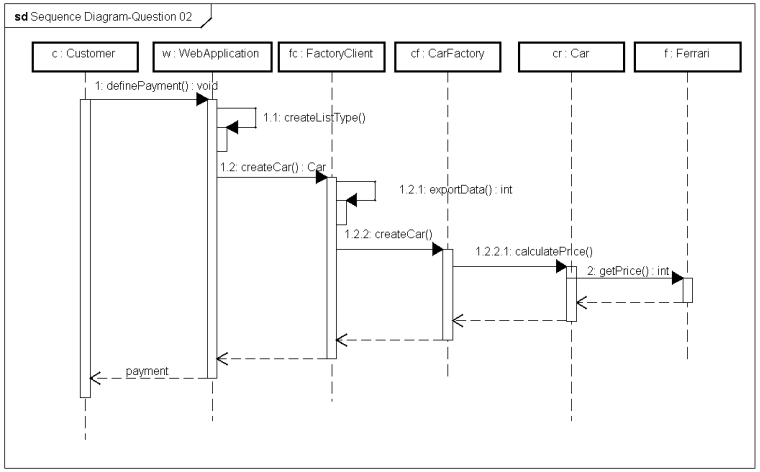
☐ **D)**
```
class ValveEditionPanel {
  boolean execute(){
     m.getSimulationData();
     s.getSubsystemData();
     e.getVarsConfig();
     p.getRotaryTableDistance();}}
```

☐ **E)** Nenhuma interpretação pode ser feita porque existe problema no modelo.

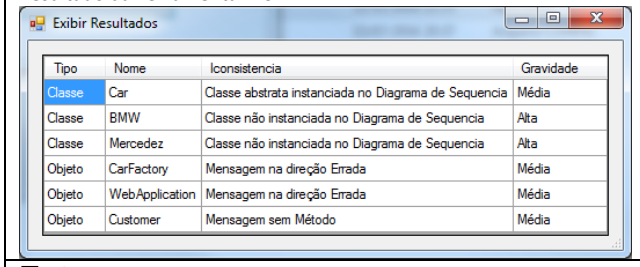
Hora de Início: hh : mm **Hora de Término:** hh : mm

QUESTÃO 02: Suponha que você é um desenvolvedor de uma aplicação WEB para venda de carros. Considerando o diagrama de classe e sequência acima, como você implementaria a classe ***WebApplication***?





Resultado da Ferramenta DIUML:



☐ **A)**
```
class WebApplication {
      void definePayment(){
        w.createList();
        fc.createCar();
        // Do Something
        fc.print();}}
```

☐ **B)**
```
class WebApplication {
      void definePayment(){
        w.createList();
        fc.createCar();
        // Do Something
        f.calculatePrice();}}
```

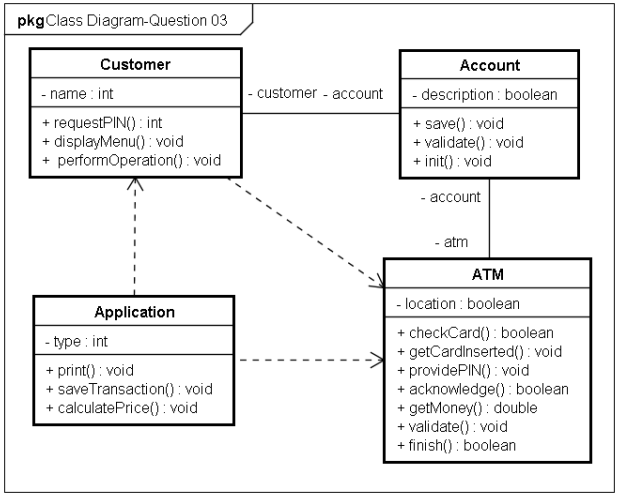
☐ **C)**
```
class WebApplication {
      void definePayment(){
        w.createList();
        f.createCar();
        // Do Something}}
```

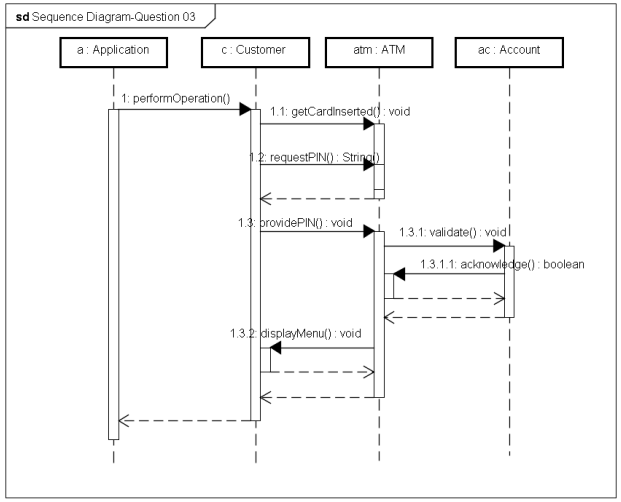
☐ **D)**
```
class WebApplication {
      boolean definePayment (){
        w.createList();
        w.exportData();
       fc.createCar();
       // Do Something
       f.calculatePrice();}}
```

☐ **E)** Nenhuma interpretação pode ser feita porque existe problema no modelo.

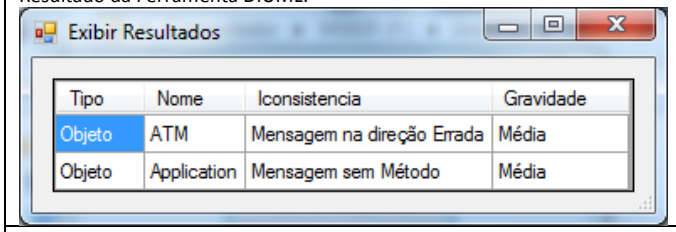
Hora de Início: hh : mm **Hora de Término:** hh : mm

QUESTÃO 03: Suponha que você é um desenvolvedor de um projeto de uma instituição financeira. Considerando o diagrama de classe e sequência acima, como você implementaria a classe ***ATM***?





Resultado da Ferramenta DIUML:



☐ **A)**
```
class ATM {
    void getInsertedCard(){
       c.requestPIN()   }
    void providePIN(){
       ac.doOperation()
       //DoSomething
       c.displayMenu() }}
```

☐ **B)**
```
class ATM {
    void getInsertedCard(){
       c.requestPIN()   }
    void providePIN(){
       ac.validate()
       //DoSomething
       c.displayMenu() }}
```

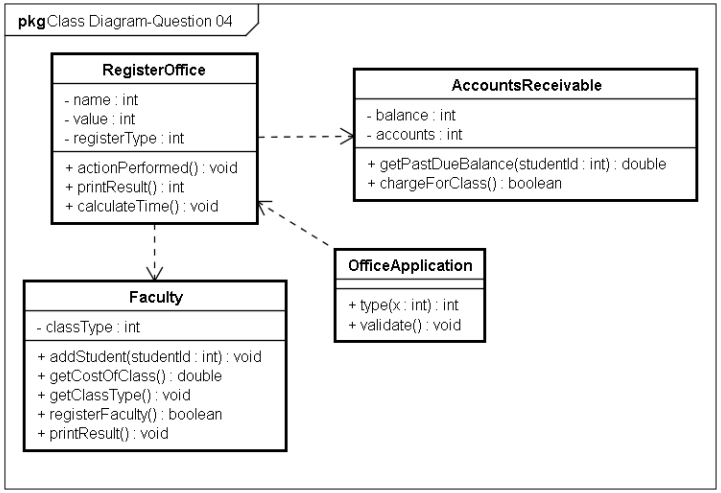
☐ **C)**
```
class ATM {
    void getInsertedCard(){
       c.requestPIN()   }
    void providePIN(){
       //DoSomething
       c.displayMenu() }}
```

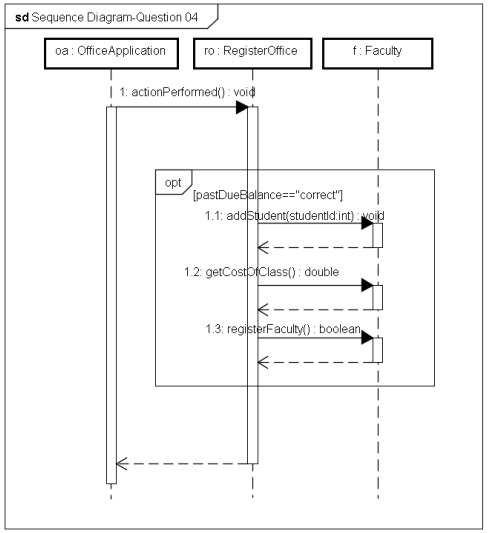
☐ **D)**
```
class ATM {
    void getInsertedCard(){
       c.requestPIN()   }
    void providePIN(){
       ac.print()
       //DoSomething
       c.displayMenu() }}
```

☐ **E)** Nenhuma interpretação pode ser feita porque existe problema no modelo.

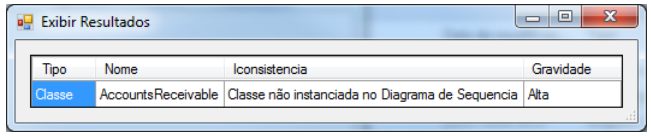
Hora de Início: hh : mm **Hora de Término:** hh : mm

QUESTÃO 04: Suponha que você é um desenvolvedor de uma aplicação para uma faculdade. Considerando o diagrama de classe e sequência acima, como você implementaria a classe ***RegisterOffice***?





Resultado da Ferramenta DIUML:



☐ **A)**
```
class RegisterOffice {
  void actionPerformed(){
    ar.getPastDueBalance(student);
    f.addStudent(studentId);
    f.getCostOfClass();
    f.printResult();
    ar.chargeForClass(); }}
```

☐ **B)**
```
class RegisterOffice {
  void actionPerformed(){
    ar.getPastDueBalance(student)
    if(pastDueBalance==”correct”){
       f.addStudent(studentId)
       f.getCostOfClass();
       f.registerFaculty();
       ro.chargeForClass(); }}}
```

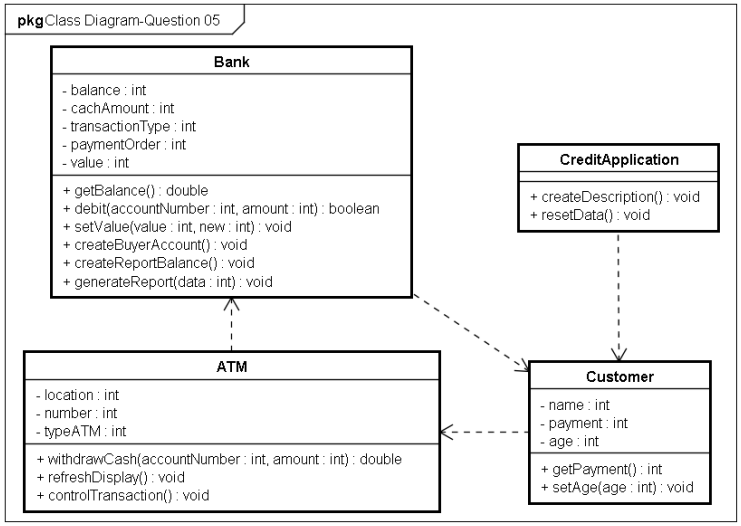
☐ **C)**
```
class RegisterOffice {
  void actionPerformed(){
    ar.getPastDueBalance(student)
     if(pastDueBalance==”correct”){
        f.addStudent(studentId);
        f.getCostOfClass();
        f.registerFaculty();
        ar.chargeForClass(); }}}
```

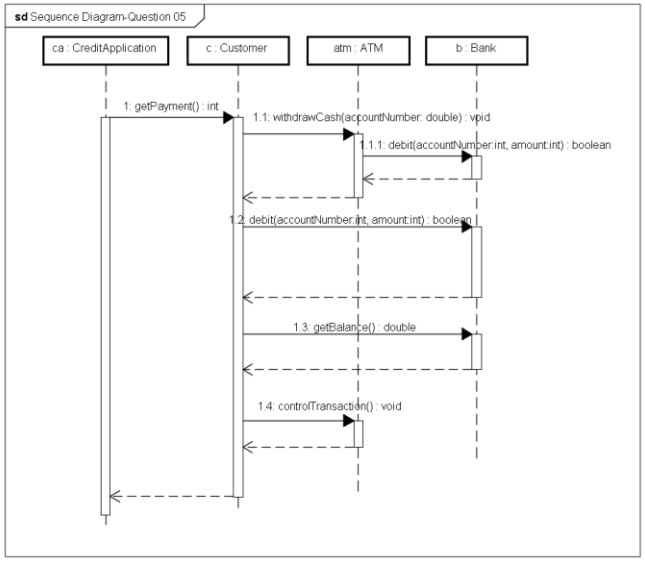
☐ **D)**
```
class RegisterOffice {
  void actionPerformed(){
    ar.getPastDueBalance(student) }
    if(pastDueBalance==”correct”){
        f.addStudent(studentId);
        f.getCostOfClass();
        f.registerFaculty();
        ar.chargeForClass(); }}}
```

☐ **E)** Nenhuma interpretação pode ser feita porque existe problema no modelo.

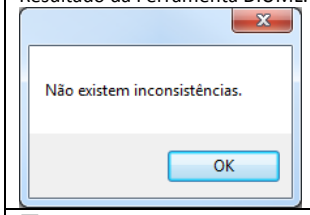
Hora de Início: hh : mm **Hora de Término:** hh : mm

QUESTÃO 05: Suponha que você é um desenvolvedor de uma aplicação para uma instituição financeira. Considerando o diagrama de classe e sequência acima, como você implementaria a classe ***Customer***?





Resultado da Ferramenta DIUML:



☐ **A)**
```
class Customer {
    double getPayment(){
      atm.withdrawCash(accountNumber, amount);
      atm. refreshDisplay();
      atm.debit();
      b.getBalance();
      atm.controlTransaction();}}
```

☐ **B)**
```
class Customer {
    double getPayment(){
      atm.withdrawCash(accountNumber);
      b.debit(accountNumber, amount);
      b.getBalance();
      atm.controlTransaction();}}
```

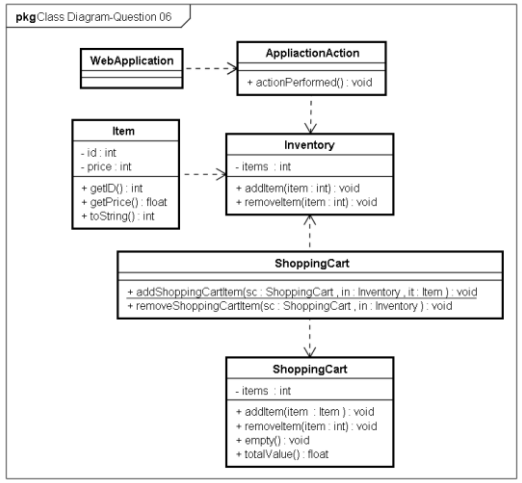
☐ **C)**
```
class Customer {
    void getPayment(){
      atm.withdrawCash(accountNumber, amount);
      atm.debit();
      b.getBalance();
      atm.controlTransaction();}}
```

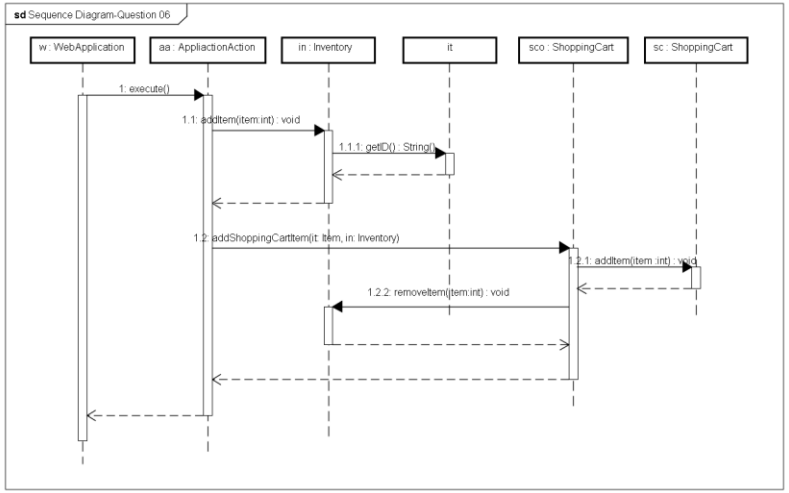
☐ **D)**
```
Class Customer {
    void getPayment(){
      atm.withdrawCash(accountNumber, amount);
      atm.debit();
      atm.getBalance();
      b.controlTransaction();}}
```

☐ **E)** Nenhuma interpretação pode ser feita porque existe problema no modelo.

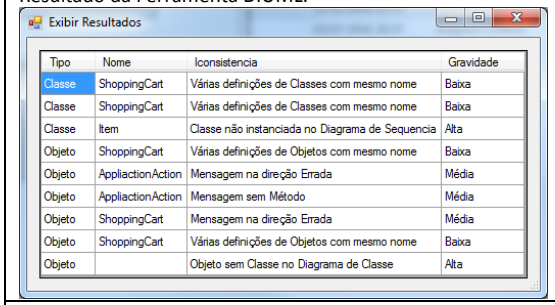
Hora de Início: hh : mm **Hora de Término:** hh : mm

QUESTÃO 06: Suponha que você é um desenvolvedor de uma aplicação para uma loja virtual. Considerando o diagrama de classe e sequência acima, como você implementaria a classe ***AplicationAction***?





Resultado da Ferramenta DIUML:



Tipo	Nome	Iconsistencia	Gravidade
Classe	ShoppingCart	Várias definições de Classes com mesmo nome	Baixa
Classe	ShoppingCart	Várias definições de Classes com mesmo nome	Baixa
Classe	Item	Classe não instanciada no Diagrama de Sequencia	Alta
Objeto	ShoppingCart	Várias definições de Objetos com mesmo nome	Baixa
Objeto	AppliactionAction	Mensagem na direção Errada	Média
Objeto	AppliactionAction	Mensagem sem Método	Média
Objeto	ShoppingCart	Mensagem na direção Errada	Média
Objeto	ShoppingCart	Várias definições de Objetos com mesmo nome	Baixa
Objeto		Objeto sem Classe no Diagrama de Classe	Alta

☐ **A)**
```
class ApplicationAction {
  execute(){
    in.addItem(item);
    it.getID();
    sco.addShoppingCartItem(item);
    sc.addItem(item);}}
```

☐ **B)**
```
class ApplicationAction {
  execute(){
    in.addItem();
    it.getID();
    sco.addShoppingCartItem(item);
    sc.addItem(item);}}
```

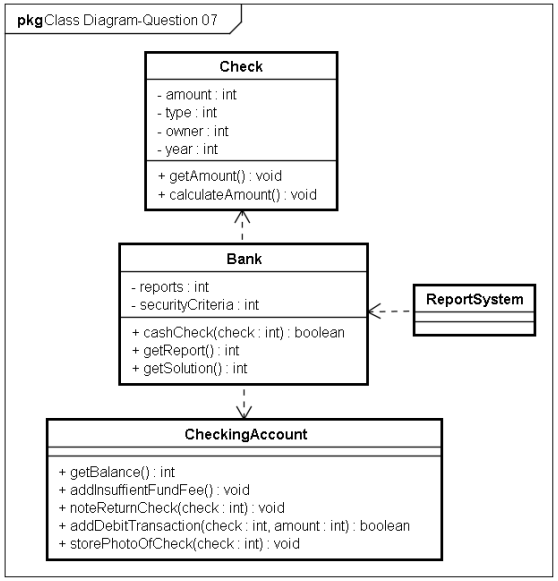
☐ **C)**
```
class ApplicationAction {
  execute(){
    in.addItem(item);
    it.getID();
    sco.addShoppingCartItem(item);
    sco.addItem(item);}}
```

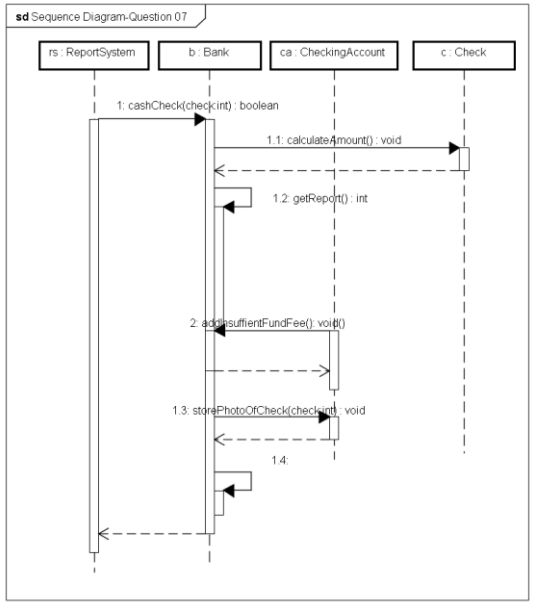
☐ **D)**
```
class ApplicationAction {
  execute(){
    in.addItem(item);
    sco.addShoppingCartItem(item);
    sco.addItem(item);}}
```

☐ **E)** Nenhuma interpretação pode ser feita porque existe problema no modelo.

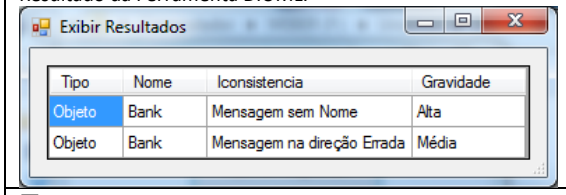
Hora de Início: hh : mm **Hora de Término:** hh : mm

QUESTÃO 07: Suponha que você é um desenvolvedor. Considerando o diagrama de classe e sequência acima, como você implementaria a classe ***Bank***?





Resultado da Ferramenta DIUML:



☐ **A)**
```
class Bank {
    boolean cashCheck (checkString){
      c.calculateAmount();
      b.getReport();
      ca. addInsuffientFundFee();
      ca. storePhotoOfCheck(check);}}
```

☐ **B)**
```
class Bank {
    boolean cashCheck (checkString){
      c.calculateAmount();
      b.getReport();
      ca. storePhotoOfCheck(check);}}
```

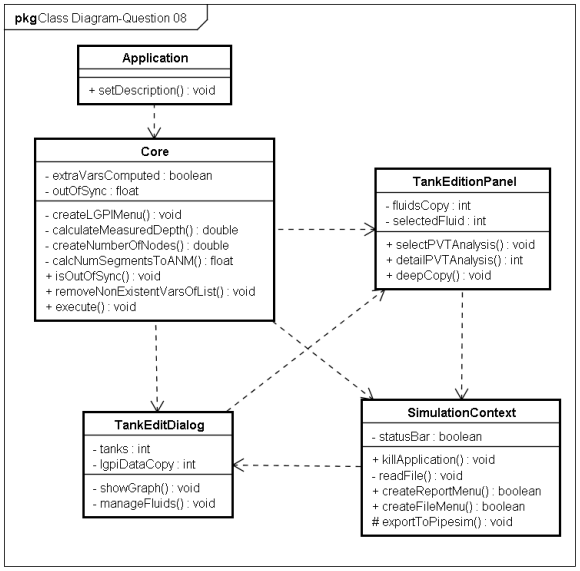
☐ **C)**
```
class Bank {
    boolean cashCheck (checkString){
      c.calculateAmount();
      b.getReport(string);
      ca. addInsuffientFundFee();
      ca. storePhotoOfCheck(check);}}
```

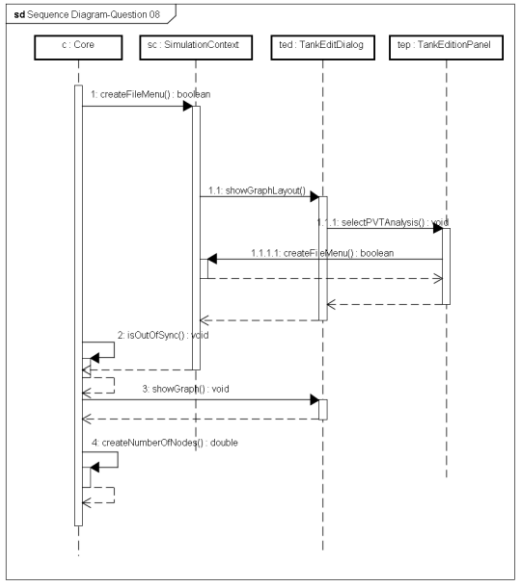
☐ **D)**
```
class Bank {
    boolean cashCheck (checkString){
      c.calculateAmount();
      ca. addInsuffientFundFee();
      ca. storePhotoOfCheck(check);}}
```

☐ **E)** Nenhuma interpretação pode ser feita porque existe problema no modelo.

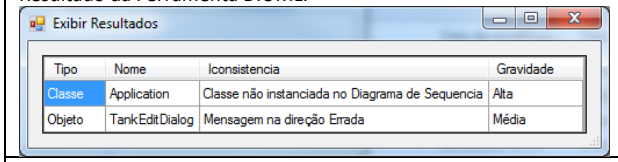
Hora de Início: hh : mm **Hora de Término:** hh : mm

QUESTÃO 08: Suponha que você é um desenvolvedor. Considerando o diagrama de classe e sequência acima, como você implementaria a classe ***SimulationContext***?





Resultado da Ferramenta DIUML:



☐ **A)**
```
class SimulationContext {
  boolean createFileMenu(){
    ted.showGraphLayout();
    tap.selectIPTVAnalysis();
    sc.createFileMenu();}}
```

☐ **B)**
```
class SimulationContext {
  boolean createFileMenu(){
    ted.showGraphLayout();
    tap.selectIPTVAnalysis();
    sc.createFileMenu();
    c.showGraph();}}
```

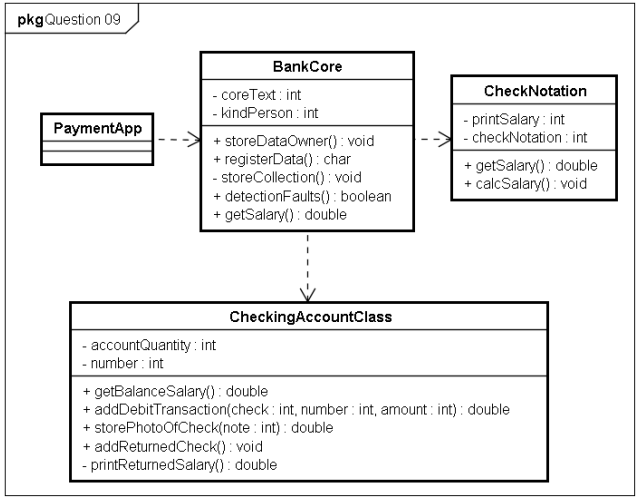
☐ **C)**
```
class SimulationContext {
  boolean createFileMenu(){
    ted.showGraphLayout();
    tap.selectIPTVAnalysis();
    sc.createFileMenu();
    c.showGraph();
    c.createNumberOfNodes();}}
```

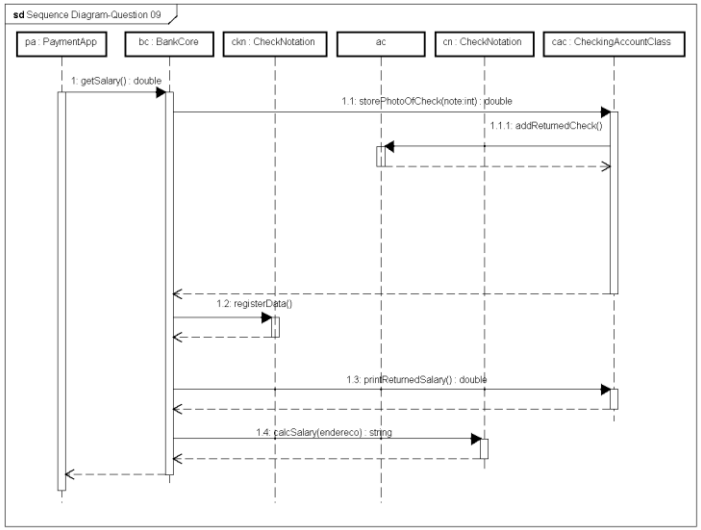
☐ **D)**
```
class SimulationContext {
  boolean createFileMenu(){
    ted.showGraphLayout();
    tap.selectIPTVAnalysis();
    sc.createFileMenu();
    c.createNumberOfNodes();}}
```

☐ **E)** Nenhuma interpretação pode ser feita porque existe problema no modelo.

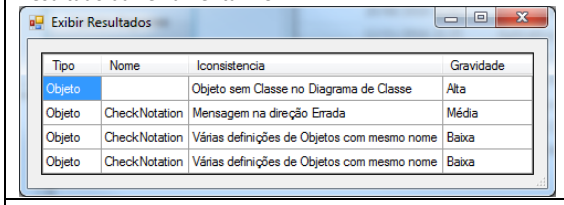
Hora de Início: hh : mm **Hora de Término:** hh : mm

QUESTÃO 09: Suponha que você é um desenvolvedor. Considerando o diagrama de classe e sequência acima, como você implementaria a classe ***BankCore***?





Resultado da Ferramenta DIUML:



☐ **A)**
```
class BankCore {
  double getSalary(){
    cac.storePhotoOfCheck(note);
    ckn.registerData();
    cac.printReturnedSalary();
    cn.calcSalary(endereco);}}
```

☐ **B)**
```
class BankCore {
  double getSalary(){
    cac.storePhotoOfCheck(note);
    ac.addReturnedCheck();
    ckn.registerData();
    cac.printReturnedSalary();
    cn.calcSalary(endereco);}}
```

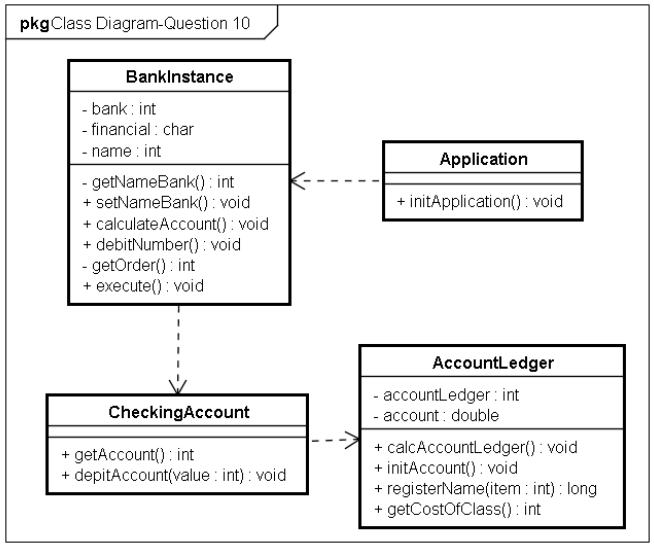
☐ **C)**
```
class BankCore {
  double getSalary(){
    cac.storePhotoOfCheck(note);
    ac.addReturnedCheck();
    cac.printReturnedSalary();
    cn.calcSalary(endereco);}}
```

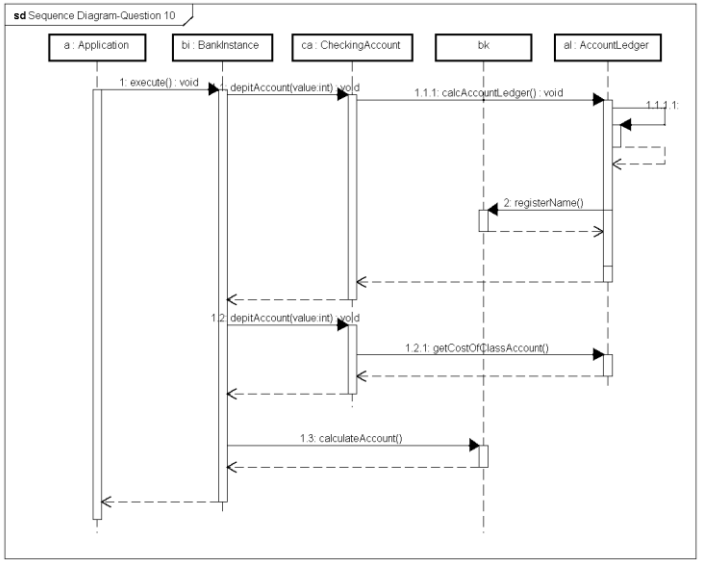
☐ **D)**
```
class BankCore {
  get Salary(){
    cac.storePhotoOfCheck(note);
    ckn.registerData();
    cac.printReturnedSalary();
    cn.calcSalary(endereco);}}
```

☐ **E)** Nenhuma interpretação pode ser feita porque existe problema no modelo.

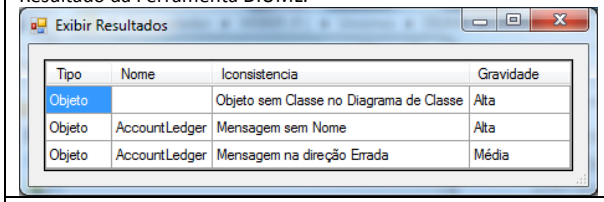
Hora de Início: hh : mm **Hora de Término:** hh : mm

QUESTÃO 10: Suponha que você é um desenvolvedor. Considerando o diagrama de classe e sequência acima, como você implementaria a classe ***BankInstance***?





Resultado da Ferramenta DIUML:



☐ **A)**
```
class BankInstance {
  void execute(){
    ca.depitAccount(value);
    at.registerName();
    ca.depitAccount(value);
    al.getCostClassAccount();
    bk.calculateAccount();}}
```

☐ **B)**
```
class BankInstance {
  void execute(){
    ca.depitAccount(value);
    at.registerName();
    ca.depitAccount(value);
    bk.calculateAccount();}}
```

☐ **C)**
```
class BankInstance {
  void execute(){
    ca.depitAccount(value);
    ca.depitAccount(value);
    al.getCostClassAccount();
    bk.calculateAccount();}}
```

☐ **D)**
```
class BankInstance {
  execute(){
    ca.depitAccount(value);
    ca.depitAccount(value);
    bk.calculateAccount();}}
```

☐ **E)** Nenhuma interpretação pode ser feita porque existe problema no modelo.

ANEXO A - ARTIGOS PUBLICADOS

BISCHOFF, V.; FARIAS, K.; GONÇALES, L.; WEBER, V. **Towards an Architecture for Integration of Feature Models.** International Journal of Computer Science and Software Engineering (IJCSSE), Volume 5, Issue 12, p. 265-272, December, 2016.

WEBER, V.; FARIAS, K.; GONÇALES, L.; BISCHOFF, V. **Detecting Inconsistencies in Multi-view UML Models.** International Journal of Computer Science and Software Engineering (IJCSSE), Volume 5, Issue 12, p. 260-264, December, 2016.