Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Coordenação de Pós-Graduação em Informática

Automação da Técnica de Inspeção Guiada para Conformidade entre Requisitos e Diagramas UML

Anne Caroline Oliveira Rocha

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande - Campus I como parte dos requisitos necessários para obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: Ciência da Computação
Linha de Pesquisa: Engenharia de Software

Patrícia Duarte de Lima Machado
Franklin Souza Ramalho
(Orientadores)

Campina Grande, Paraíba, Brasil

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA CENTRAL DA UFCG

R672a Rocha, Anne Caroline Oliveira.
Automação da técnica de inspeção guiada para conformidade entre requisitos e diagramas UML /Anne Caroline Oliveira Rocha. ─ Campina Grande, 2010.
138 f.: il.

Dissertação (Mestrado em Ciência da Computação) – Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática.
Orientadores: Profª. PhD. Patrícia Duarte de Lima Machado, Profº. PhD. Franklin Souza Ramalho.
Referências.

1. Testes de Software. 2. Inspeção de Software. 3. Semântica de Ações. 4. Inspeção Guiada – Automática. I. Título.

CDU 004.052.42 (043)

"AUTOMAÇÃO DA TÉCNICA DE INSPEÇÃO GUIADA PARA CONFORMIDADE ENTRE REQUISITOS E DIAGRAMAS UML"

ANNE CAROLINE OLIVEIRA ROCHA

DISSERTAÇÃO APROVADA EM 30.04.2010

PATRICIA DUARTE DE LIMA MACHADO, Ph.D
Orientador(a)

FRANKLIN DE SOUZA RAMALHO, Dr.
Orientador(a)

JORGE CESAR ABRANTES DE FIGUEIREDO, D.Sc
Examinador(a)

EDUARDO HENRIQUE DA SILVA ARANHA, Dr.
Examinador(a)

CAMPINA GRANDE - PB

Resumo

Em um processo de desenvolvimento de software, artefatos de uma etapa são utilizados como fonte para criação de novos artefatos para outras etapas. Então, defeitos podem ser inseridos durante a transição de uma etapa para outra: artefatos podem ficar inconsistentes, levando à construção de um software com defeitos. Neste contexto, uma técnica de inspeção pode ser utilizada para verificar esses artefatos, que são produzidos desde as primeiras etapas do desenvolvimento. Este trabalho apresenta uma forma de automação da técnica de inspeção guiada. Esta técnica tem o objetivo de verificar a conformidade entre artefatos de diferentes níveis de abstração (por exemplo, uma especificação de requisitos com relação a um diagrama de seqüência). Esta inspeção é realizada através de casos de teste, que representam cada cenário de caso de uso da especificação de requisitos. Como os passos de um caso de teste contêm o comportamento de um sistema, então a inspeção guiada permite encontrar defeitos semânticos para aquele sistema. Além disso, por ser uma técnica automática, é possível detectar também inconsistências entre as sintaxes dos artefatos de software. Para dar suporte à automação, serão utilizados conceitos de MDA (*Model Driven Architecture*) para transformação entre modelos e a ferramenta USE para simulação de modelos.

Palavras-chave: Testes de Software, Inspeção de Software, Semântica de Ações, Inspeção Guiada - Automática.

Abstract

In a software development process, artifacts from a stage are used as input to create new artifacts on another. The transition between different artifacts may not be precise; inconsistencies may occur. These inconsistent artifacts may produce software with defects. In this context, a software inspection technique is needed to validate these artifacts. This paper presents a method to automate a guided inspection technique, which evaluates the conformity between artifacts of distinct abstraction levels. The inspection uses test cases, that represent each use case scenario of the requirement specification. Since test case steps have the system behavior, so the guided inspection allows to detect semantic defects. Moreover, how it's an automated techinique it's also possible to detect inconsistences about the artefact syntaxes. As support for the automation, we are using MDA (Model Driven Architecture) to perform model-to-model transformations and the USE tool for model simulation.

Keywords: Software Testing, Software Inspection, Action Semantics, Automated Guided Inspection.

Agradecimentos

Primeiramente agradeço a Deus, por tantas bênçãos que me tem permitido vivenciar.

Quero agradecer a minha família. Em especial, agradeço a minha mãe, pelo seu amor incondicional e que apesar da distância, sempre me incentivou na minha carreira profissional. Agradeço ao meu pai e aos meus irmãos Karina e Vital pelo apoio e carinho.

Agradeço bastante ao meu esposo, Bruno Coitinho, por sempre estar ao meu lado, em todos os momentos bons e ruins, tornando meus dias mais especiais e sempre me ajudando a tomar as melhores decisões.

Agradeço aos pais de Bruno, Anete e Marcos, pelo apoio e carinho que sempre me dão quando preciso deles.

Sou muito grata à professora Patrícia Machado por sua orientação e apoio sempre que precisei. Agradeço também ao professor Franklin Ramalho, que não mediu esforços para me orientar e colaborar com o trabalho do início ao fim.

Agradeço às minhas amigas Maria de Lourdes, Camila Luna, Mara Caroline pela amizade sincera e por sempre me ajudarem quando necessitei de ajuda ou conselhos.

Agradeço aos meus amigos Denise Vaz e Rodrigo Souza, que com certeza sua amizade foi uma das melhores coisas que ganhei neste período de mestrado.

Agradeço também à equipe do GMF, pela colaboração durante as apresentações no GMF Café. Em especial, agradeço a Lilian por sua simpatia sempre que nos encontrávamos. Agradeço a Everton e a Neto pela colaboração com o trabalho e a Andreza pela amizade.

Agradeço também a Aninha por sempre ser tão atenciosa.

Ao Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq), pelo apoio financeiro.

Conteúdo

Lista de Símbolos

ATL - *Atlas Transformation Language*

CIM - *Computational Independent Model*

IG - *Inspeção Guiada*

IGA - *Inspeção Guiada Automática*

GQM - *Goal Question Metric*

LTS - *Labelled Transition System*

MBT - *Model-Based Testing*

MDA - *Model-Driven Architecture*

MDD - *Model-Driven Development*

MDT - *Model-Driven Testing*

OCL - *Object Constraint Language*

OMG - *Object Management Group*

OORT - *Oriented-Object Reading Technique*

PBR - *Perspective-Based Reading*

PIM - *Platform Independent Model*

PSM - *Platform Specific Model*

QVT - *Query/Views/Transformations*

UML2 - *Unified Modeling Language 2.0*

UMLAnt - *UML Animator and Tester*

UMLAUT - *Unified Modeling Language All pUrposes Transformer*

USE - *UML-based Specification Environment*

XMI - *XML Metadata Interchange*

Lista de Figuras

Lista de Tabelas

Lista de Códigos Fonte

Capítulo 1

Introdução

Engenharia de Software é uma área da informática que descreve os aspectos e as metodologias para a produção de um software desde as etapas iniciais, referentes à especificação e modelagem, até as etapas de construção, manutenção e evolução do software. Geralmente, um processo de desenvolvimento iterativo engloba as etapas de requisitos, análise, projeto, codificação, verificação e validação, produção e evolução, onde ao final de cada etapa é produzido um artefato do software [Som07].

Esses artefatos possuem diferentes níveis de abstração, ou seja, a especificação de requisitos é mais abstrata do que os modelos de projeto, os quais são mais abstratos que o código fonte do sistema. Os artefatos produzidos por uma etapa podem ser utilizados como fonte para as próximas etapas do desenvolvimento. Por exemplo, para criar os modelos de projeto, é necessário conhecer os requisitos do sistema, que podem estar contidos numa especificação de caso de uso, da etapa de análise. Entretanto, como a atividade de modelar um sistema a partir de requisitos descritos em linguagem natural é bastante criativa, podem ocorrer erros no momento de modelar o sistema, devido à informações ambíguas ou inconsistentes. Estes erros podem tornar os modelos de projeto e o código fonte, por exemplo, inconsistentes entre si, logo será produzido um sistema com defeitos, que não atende às necessidades do cliente.

Para minimizar os defeitos encontrados em artefatos de software existem as técnicas de verificação e validação (V & V). Com estas técnicas é possível verificar se o sistema faz o que deveria fazer e não faz o que não deveria fazer. Isto contribui para aumentar a qualidade do sistema. A verificação é realizada para comprovar se o sistema está de acordo

com os requisitos funcionais e não-funcionais especificados. A validação tem a finalidade de assegurar que o sistema atende às necessidades do usuário, já que nem sempre a especificação do sistema reflete os desejos do cliente [Som07].

Algumas das técnicas utilizadas para realizar a verificação e validação de software são os testes e a inspeção de software. Os testes de software examinam as saídas do sistema a fim de descobrir defeitos, nos quais o comportamento do sistema está incorreto, não é desejável ou não está de acordo com os requisitos do sistema [Som07]. Os testes geralmente são realizados após a etapa de codificação, num processo de desenvolvimento. Com isso, para executar os testes em um software é necessário que haja uma versão do código do sistema.

A técnica de inspeção de software é uma maneira de revisar artefatos de software em diferentes níveis de abstração, a fim de encontrar defeitos, omissões e anomalias [Som07]. A inspeção não demanda a execução do software, o que é uma vantagem com relação à técnica de testes. Pois, mantém a qualidade dos artefatos desde as primeiras etapas do desenvolvimento. No entanto, a técnica de testes tem um papel muito importante por permitir verificar se o código final do sistema está de acordo com os requisitos do sistema.

Durante as etapas de análise e projeto de um processo de desenvolvimento, o sistema precisa ser modelado a partir da especificação de requisitos, de forma que o desenvolvedor possa compreender melhor os requisitos do sistema. Pois, não é trivial codificar um sistema a partir de uma especificação descrita em linguagem natural. A linguagem natural pode gerar ambigüidades que serão refletidas no código fonte. Assim, para modelar um sistema orientado a objetos, geralmente, utiliza-se diagramas descritos na linguagem UML (*Unified Modeling Language*) [RJB98]. Com UML é possível descrever graficamente todo o sistema de forma menos abstrata que a especificação, através de diagramas estruturais e comportamentais. Os principais diagramas estruturais são o diagrama de classes e de componentes; os comportamentais são o diagrama de caso de uso, o diagrama de atividades e o diagrama de seqüência [Pen03].

Como é difícil automatizar o processo criativo de projetar um software, na prática, diagramas UML são criados de forma manual. Contudo, é possível que os diagramas UML não estejam em conformidade com a especificação, ou seja, eles podem não refletir, de forma completa, os requisitos. Por fim, requisitos e seus diagramas podem apresentar inconsistências devido ao uso de linguagem natural, principalmente em especificações de caso de uso.

Dado que isso aconteça, todo o processo de desenvolvimento estará comprometido, pois possivelmente haverá defeitos no código fonte.

Na literatura, encontra-se algumas técnicas de inspeção manual entre diagramas UML e a especificação de requisitos, como *Oriented-Object Reading Technique* (OORT) [TSCB99], *Perspective-Based Reading* (PBR) [SRB00] e Inspeção Guiada [MS01].

A técnica OORT realiza a inspeção manualmente através de uma leitura horizontal e vertical entre os artefatos de software. A leitura horizontal é realizada entre artefatos de um mesmo nível de abstração, por exemplo, entre um diagrama de classes e um diagrama de atividades. A leitura vertical é feita entre artefatos de níveis de abstração diferentes, por exemplo, entre a especificação de requisito e um diagrama de seqüência. Com isso, a técnica permite verificar se todos os artefatos de análise e projeto estão descrevendo o mesmo sistema. Além disso, a inspeção é feita com base apenas nas inconsistências encontradas nas sintaxes destes artefatos.

A técnica PBR realiza uma inspeção manual diferenciada em cada artefato de software de acordo com uma determinada perspectiva, por exemplo, na perspectiva do cliente, do analista, do projetista, do desenvolvedor, etc. Para cada perspectiva ou visão existe um conjunto de questões que orientam o inspetor a encontrar diferentes defeitos em um mesmo artefato. Existem modelos, na literatura, para estas questões utilizadas em PBR, mas elas podem ser adaptadas para um determinado projeto. Com PBR é possível verificar se todos os artefatos de projeto estão descrevendo o mesmo sistema para diferentes visões. Além disso, PBR permite detectar ambigüidades e inconsistências, tanto sintática quanto semanticamente, entre os diagramas UML e a especificação de requisitos. Uma vantagem de PBR é que um mesmo artefato será inspecionado várias vezes, o que aumentam as chances de se encontrar mais defeitos, mas pode aumentar o custo para o projeto.

Outra técnica para inspeção manual entre diagramas UML e a especificação de requisitos é a inspeção guiada, onde a inspeção é guiada por casos de teste, ou seja, os casos de teste são executados mentalmente pelo inspetor sobre os diagramas UML inspecionados. Esses casos de teste são definidos a partir da especificação de caso de uso, a fim de avaliar documentos produzidos a partir desta. Assim, torna-se possível uma investigação sistemática dos documentos com relação à ambigüidade, completude e consistência. Além disso, a inspeção guiada foca mais na semântica dos artefatos a serem inspecionados do que na sin-

taxe, diferentemente das outras técnicas de inspeção [MM99].

Técnicas de inspeção de software são bastante úteis para garantir a conformidade entre artefatos de diferentes níveis de abstração. No entanto, quando a técnica é realizada de forma manual pode haver alguns problemas, por exemplo, aumentar o tempo de execução e o custo para o projeto, requerer que o inspetor tenha conhecimento especializado, o inspetor pode não encontrar todos os defeitos devido à falta de atenção, entre outros fatores. Com isso, se houvesse uma técnica de inspeção automática seria mais vantajoso, pois os artefatos manipulados durante a inspeção seriam controlados e não dependeriam totalmente do inspetor. Então, aumentariam as chances de se encontrar mais defeitos nestes artefatos. Além disso, para que a técnica de inspeção seja completa é necessário que ela detecte defeitos tanto na sintaxe dos artefatos quanto na semântica dos requisitos. Desta forma, os artefatos inspecionados estarão em conformidade com os requisitos de forma completa, correta e consistente.

1.1 Objetivo do Trabalho

O objetivo deste trabalho é desenvolver uma técnica que viabilize a automação da técnica de inspeção guiada, para que seja possível realizar inspeção entre uma especificação de requisitos e diagramas UML de projeto. Assim, será possível analisar de forma automática a conformidade entre artefatos de software de diferentes níveis de abstração, mesmo quando estes artefatos estão descritos em diferentes linguagens, por exemplo, a especificação de requisitos em linguagem natural e diagramas de projeto na linguagem UML.

Além disso, o procedimento proposto tem a finalidade de encontrar defeitos semânticos e sintáticos nos diagramas UML, como: falta de requisitos; informações ambíguas ou divergentes entre o que foi especificado em cada artefato; informação desnecessária e falta de conformidade na sintaxe. O procedimento consiste em receber como entrada um conjunto de casos de teste abstratos (descritos em linguagem natural) e diagramas UML de classes e de seqüência. Os casos de teste abstratos são descritos em linguagem natural e deve existir um caso de teste para cada cenário de caso de uso do sistema. Os diagramas UML são descritos na linguagem *XML Metadata Interchange* (XMI) [Gro07c] de acordo com o padrão de UML. Após o processamento da inspeção, um relatório de defeitos é gerado automatica-

mente contendo as inconsistências encontradas entre os casos de teste e os diagramas UML inspecionados.

Para realizar a automação da inspeção guiada as seguintes etapas foram definidas:

1. Encontrar uma maneira de identificar a semântica de cada passo de execução dos casos de teste abstratos;
2. Definir uma linguagem controlada, passível de automação, para a semântica definida nos casos de teste;
3. Simular os diagramas UML de forma que os casos de teste possam ser executados sobre eles;
4. Verificar se os casos de teste estão de acordo com invariantes, pré e pós-condições;
5. Realizar a inspeção automática entre os casos de teste e os diagramas de seqüência;
6. Gerar um relatório de defeitos com as inconsistências encontradas durante a inspeção.

1.2 Relevância

Diagramas UML de projeto utilizados no processo de desenvolvimento podem estar inconsistentes com relação à especificação de requisitos e isto pode levar à construção de um software com defeitos. Com isso, este software não atenderá às necessidades do cliente e quanto mais tarde os defeitos forem encontrados nos artefatos maior será o custo para corrigi-los [Pre01].

Para melhorar a qualidade desses diagramas UML é necessário inserir uma técnica de inspeção dentro do processo de desenvolvimento para garantir a conformidade destes diagramas com a especificação de requisitos. Logo, este trabalho se torna relevante porque a técnica de inspeção automática proposta visa realizar inspeção utilizando a especificação de requisitos, mesmo ela estando descrita em linguagem natural. Além disso, devido à técnica de inspeção guiada utilizar casos de teste para guiar o inspetor, então é realizada uma inspeção semântica entre os artefatos e específica para cada sistema. Por ser uma técnica automática que realiza inspeção entre artefatos de software é possível encontrar inconsistências na sintaxe destes artefatos, além de detectar alguns defeitos que nem sempre são percebidos durante a inspeção manual.

Contudo, a maioria das técnicas automáticas de inspeção analisa os artefatos apenas do ponto de vista sintático, por exemplo, através de um conjunto de *checklists*. Os *checklists* são gerais e tentam abranger de forma abstrata todos os artefatos a serem inspecionados. Desta forma, eles nem sempre são completos e o inspetor geralmente não consegue encontrar problemas de ambigüidade ou inconsistência entre os artefatos, pois cada artefato é inspecionado separadamente.

1.3 Estrutura da Dissertação

As próximas partes deste documento estão estruturadas da seguinte forma:

Capítulo 2: Fundamentação Teórica Este capítulo apresenta uma descrição de conceitos básicos necessários para compreender melhor este trabalho. Os conceitos descritos estão relacionados à inspeção de software e às técnicas necessárias para a automação da inspeção guiada, como: inspeção de software, *Model-Driven Architecture* (MDA) [KWB03], Semântica de Ações de UML2 [Gro07b] e uma ferramenta para simulação de modelos UML.

Capítulo 3: Automação da Inspeção Guiada Este capítulo apresenta a descrição da solução encontrada para realizar a automação da inspeção guiada. Uma visão geral do processo de inspeção guiada é relatada, assim como a arquitetura da inspeção guiada automática e um exemplo ilustrando o funcionamento desta ferramenta de inspeção.

Capítulo 4: Trabalhos Relacionados Este capítulo apresenta os principais trabalhos relacionados à inspeção entre diagramas UML e a especificação de requisitos.

Capítulo 5: Avaliação Experimental Este capítulo apresenta a avaliação experimental comparativa entre a ferramenta de inspeção guiada e outras técnicas de inspeção manual que possuem o mesmo propósito.

Capítulo 6: Conclusão e Trabalhos Futuros Este capítulo apresenta a conclusão deste trabalho através dos resultados obtidos na avaliação e além de uma descrição das perspectivas de trabalhos futuros.

Capítulo 2

Fundamentação Teórica

Este capítulo tem como objetivo fornecer embasamento teórico sobre conceitos necessários para compreender melhor este trabalho. São apresentados conceitos sobre o processo de inspeção de software tradicional e sobre as técnicas utilizadas no processo de automação da inspeção guiada, como: inspeção de software, inspeção guiada manual, MDA, semântica de ações e simulação de modelos UML.

Estas técnicas foram combinadas para tornar viável a automação da inspeção guiada proposta neste trabalho. Um dos desafios para automação foi com relação a presença de artefatos descritos em linguagem natural, para isso semântica de ações foi bastante útil. Outro desafio foi com relação a realizar a inspeção entre artefatos de diferentes níveis de abstração. Para isso, a ferramenta de simulação de modelos UML foi necessária. Além disso, tivemos que encontrar uma maneira de realizar a inspeção guiada automaticamente, para isso utilizamos MDA.

2.1 Inspeção de Software

Inspeção de Software é uma maneira particular de revisar os artefatos de software, que tem por objetivo verificar a consistência de cada artefato durante as diferentes fases de desenvolvimento, garantindo uma maior qualidade nos artefatos [KT05]. O processo de inspeção de software tradicional [Fag76] constitui uma maneira prática de realizar a inspeção nos artefatos de software. Para realizar as atividades de inspeção é necessário que a equipe seja divida em 3 diferentes papéis, que são:

1. *Moderador:* é a pessoa chave para o sucesso da inspeção. Ele deve gerenciar a equipe de inspeção e oferecer liderança. Além disso, ele deve orientar os inspetores.

2. *Inspetor:* é responsável por inspecionar os artefatos de software durante todo o processo de desenvolvimento.

3. *Autor:* é responsável por produzir os artefatos de software, que pode ser um analista ou um programador, dependendo da etapa do processo de desenvolvimento.

De acordo com a Figura 2.1, o processo de inspeção é composto por 5 atividades principais, que são:

1. *Planejamento:* nesta etapa o moderador define a equipe de inspeção e também seleciona os documentos a serem inspecionados.

2. *Preparação individual:* os inspetores analisam individualmente cada artefato de software através de um checklist e produzem uma lista com cada defeito encontrado.

3. *Reunião de inspeção:* é feita uma reunião entre o moderador, o inspetor e o autor do artefato, a fim de avaliar se as divergências encontradas são defeitos ou falsos positivos.

4. *Re-trabalho:* o autor corrige os defeitos encontrados.

5. *Continuação:* o artefato de software corrigido é passado para o moderador, que decide se aquele artefato deve ou não ser novamente inspecionado.

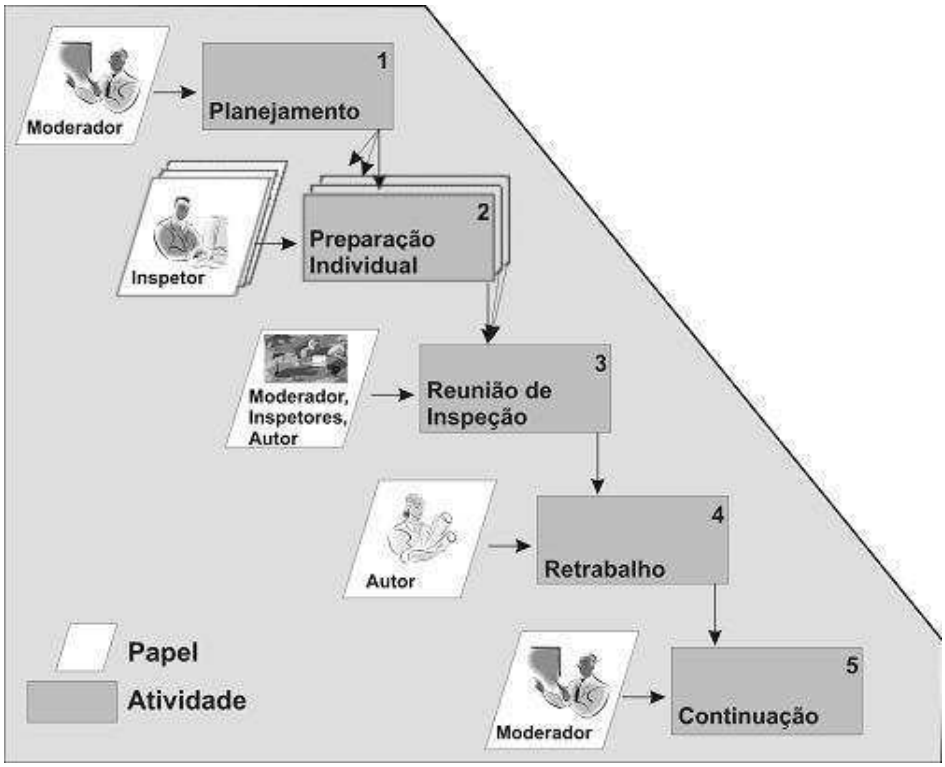

Figura 2.1: Processo de Inspeção de Software.

Baseado nesse processo de inspeção e em estudos experimentais, em [SJLY00] foi proposto uma reorganização do processo de inspeção tradicional. Este novo processo tem a finalidade de permitir que as equipes de inspeção estejam geograficamente distribuídas. O processo de inspeção definido por [SJLY00] consiste basicamente em substituir as atividades de Preparação e de Reunião pelas atividades de Detecção de defeitos, Coleção de defeitos e Discriminação de defeitos.

Na atividade de detecção de defeitos, cada inspetor tem que criar uma lista com as divergências encontradas no documento a ser inspecionado. Na atividade de coleção de defeitos, o moderador retira da lista as divergências repetidas e na atividade de discriminação de defeitos, o moderador, o autor do documento e o inspetor discute as divergências encontradas, as quais podem ser falsos positivos ou defeitos. Caso sejam defeitos, estes devem ser armazenados numa lista de defeitos, caso contrário, devem ser descartados.

Dentre as técnicas de inspeção de software mais comuns existem as técnicas fazem a inspeção *ad hoc*, ou seja, não seguem nenhum padrão para realizar a inspeção, mas apenas o conhecimento do inspetor. Há também as técnicas que utilizam um *checklist*, que é composto

por uma lista de questões que guiam o inspetor sobre o que deve e o que não deve estar nos artefatos.

2.2 Inspeção Guiada

A técnica de inspeção guiada tem o diferencial de realizar a inspeção em artefatos de software de forma a avaliar se os requisitos contidos na especificação estão presentes em todos os artefatos de design de forma consistente, completa e correta [MS01]. O critério de consistência avalia se o artefato inspecionado não possui ambigüidades nas informações do artefato. O critério de completude é mantido quando os artefatos representam 100% dos requisitos. O critério de corretude avalia se o artefato inspecionado está correto com relação aos requisitos. Esta é uma técnica de inspeção manual guiada por casos de teste.

Em testes de software, casos de teste são utilizados para exercitar o comportamento do sistema em diferentes cenários e testar se o sistema está de acordo com o comportamento esperado. Desta forma, a técnica de inspeção guiada utiliza casos de teste para exercitar o comportamento dos diagramas UML.

A inspeção guiada é realizada sobre diagramas UML de projeto criados por um analista de sistemas. Estes diagramas são feitos com base na especificação de requisitos. No entanto, nem sempre todos os requisitos estão presentes nos diagramas produzidos. Com isso, o inspetor prepara casos de teste baseados também na especificação de requisitos com a finalidade de realizar a inspeção guiada. As etapas para a elaboração da inspeção guiada manual podem ser visualizadas na Figura 2.2.

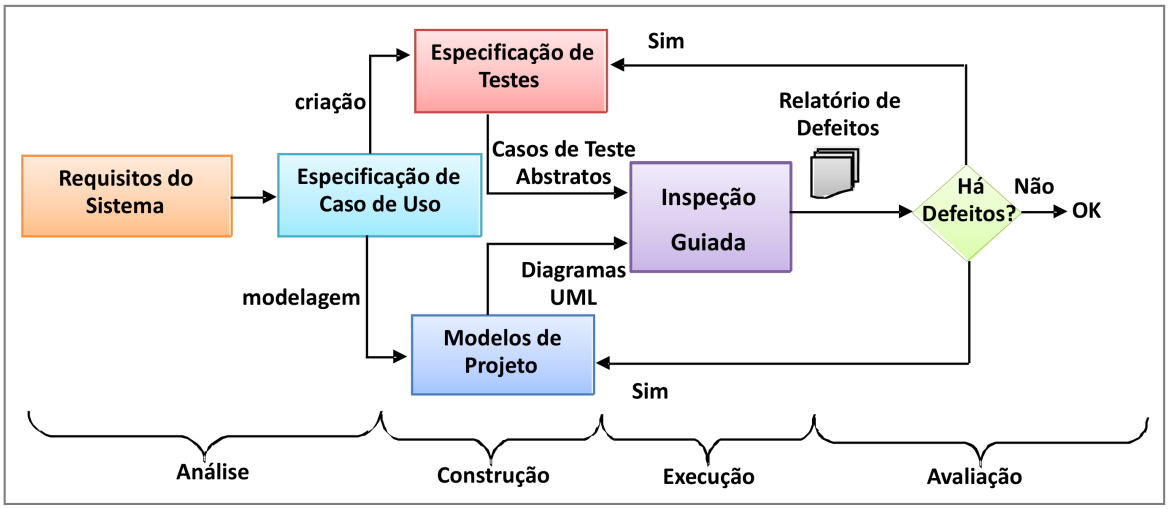

Figura 2.2: Etapas da Inspeção Guiada

- *Análise:* o inspetor deve analisar a especificação de caso de uso para classificar cada caso de uso de acordo com seu grau de criticalidade para o sistema, que é proporcional à freqüência daquele caso de uso.

- *Construção:* o inspetor deve construir uma especificação de testes, que é composta por um conjunto de casos de teste abstratos para cada cenário dos casos de uso. Estes casos de teste são usualmente descritos em linguagem natural. O analista deve criar os diagramas UML de projeto (diagrama de classes, de seqüência, de atividades etc) a serem inspecionados.

- *Execução:* a inspeção guiada em si é realizada nesta etapa, com isso a inspeção é realizada de forma que os casos de teste sejam executados mentalmente sobre cada diagrama UML, como, por exemplo, sobre um diagrama de seqüência, que represente o comportamento de um cenário de caso de uso.

- *Avaliação:* o inspetor deve avaliar se, ao final da execução dos casos de teste, se o comportamento presente nestes casos de teste estava presente nos diagramas inspecionados.

Exemplo de utilização da Inspeção Guiada

Este exemplo é sobre um "Sistema de Reserva de Hotel", o qual possui vários casos de uso, como: Fazer Reserva, Identificar Reserva, Cancelar Reserva, entre outros. A Figura 2.2 apresenta o fluxo de eventos do caso de uso "Cancelar Reserva", este caso de uso será utilizado para ilustrar a etapa de *Execução* da técnica de inspeção guiada.

Caso de Uso: Cancelar Reserva
Ator: Reservante
Objetivo: Cancelar uma reserva

Cenário Principal

1. Reservante solicita cancelamento de uma reserva.
2. Sistema identifica a reserva (Use Case Identificar Reserva).
3. Reservante confirma cancelamento.

Extensões

3. Reserva não identificada.
   a. Falha.
3. Reservante desiste do cancelamento.
   a. Falha.
3. A data da reserva é menor que a data atual.
   a. Falha.

Freqüência: Média
Criticalidade: Alta

Figura 2.3: Descrição do caso de uso Cancelar Reserva.

Foram criados 4 casos de teste para o caso de uso "Cancelar Reserva", que podem ser vistos na Figura 2.2:

Nº	Casos de Teste	Resultados Esperados
01	É solicitado o cancelamento de uma reserva. O sistema identifica a reserva.	O cancelamento é realizado.
02	É solicitado o cancelamento de uma reserva e a reserva não é identificada.	O cancelamento não é realizado.
03	É solicitado o cancelamento de uma reserva. A reserva é identificada e o hóspede desiste do cancelamento.	O cancelamento não é realizado.
04	É solicitado o cancelamento de uma reserva. **O sistema não consegue cancelar pelo fato de já ter passado do dia da reserva.**	O cancelamento não é realizado.

Figura 2.4: Casos de teste para o caso de uso Cancelar Reserva.

O analista de sistema criou alguns diagramas para representar a funcionalidade do caso de uso “Cancelar Reserva”.

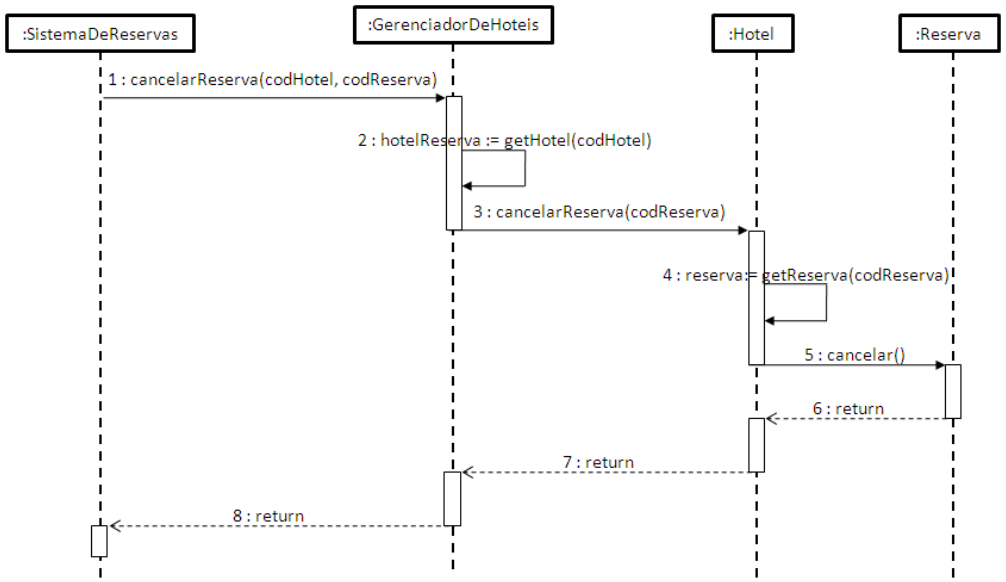

Figura 2.5: Diagrama de seqüência para o caso de uso Cancelar Reserva.

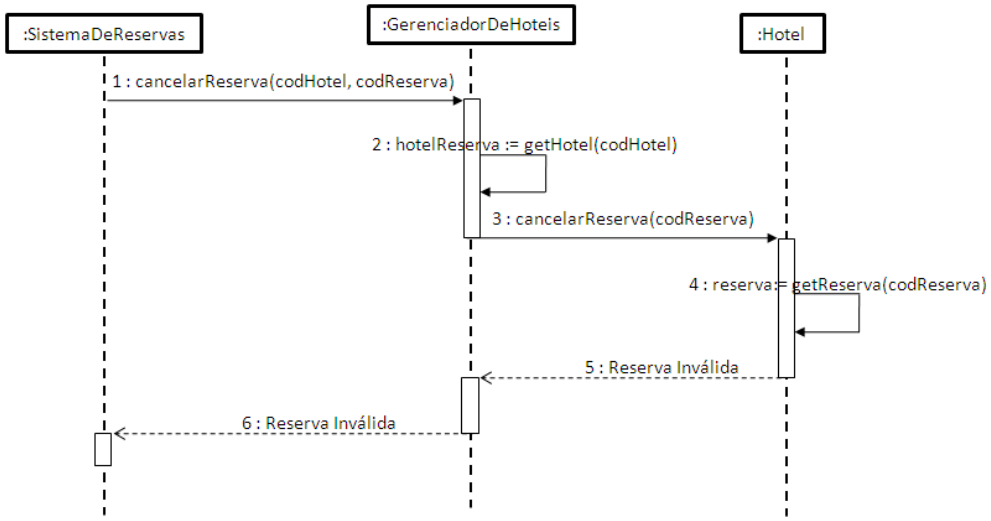

Figura 2.6: Diagrama de seqüência para o caso de uso Cancelar Reserva com reserva inválida.

Ao executar manualmente os casos de teste sobre os modelos, os inspetores perceberam que não é possível executar o caso de teste 04, pois não há um fluxo nos diagramas de seqüência do sistema (Figura 2.5 e Figura 2.6) que represente o fluxo alternativo: *cancelamento da reserva caso a data da reserva seja menor que a data atual*. Assim, de acordo

com os critérios de avaliação da inspeção guiada, temos que os modelos não atingiram o critério de *completude*, pois há funcionalidades que não foram descritas nos modelos. Com este exemplo podemos notar a importância da técnica de inspeção guiada para o processo de desenvolvimento.

2.3 *Model-Driven Architecture* (MDA)

Model-Driven Development (MDD) é um abordagem de desenvolvimento que foca na produção de software através de modelos [FS04]. Para dar apoio a esta abordagem, o *Object Management Group* (OMG) lançou o *framework Model Driven Architecture* (MDA) [KWB03]. Para realizar MDA é necessário definir meta-modelos para os modelos. Estes meta-modelos são modelos que descrevem outros modelos. A arquitetura de MDA como pode ser observada na Figura 2.7. Um processo de desenvolvimento de software usando MDA é dirigido pela atividade de modelar o sistema. Assim, o código é gerado automaticamente através de regras de transformação em diferentes níveis de abstração, partindo de um nível mais alto de abstração para um nível mais baixo. Os modelos descritos em MDA possuem geralmente 4 níveis de abstração, são eles:

1. *Computational Independent Model* (CIM): modelo de requisitos do sistema, que define o domínio, os serviços e as entidades envolvidas.

2. *Platform Independent Model* (PIM): modelo que é independente de qualquer linguagem de implementação, que define uma instância de uma meta-modelo.

3. *Platform Specific Model* (PSM): modelo adaptado para especificar o sistema com relação à implementação, dependente de uma linguagem específica.

4. Código: é a descrição do sistema em código fonte.

Geralmente, durante a produção de um software ocorrem várias mudanças nos requisitos do sistema. Entretanto, nem sempre estas novas mudanças são definidas nos modelos, pois o custo para elaborar e manter estes artefatos é alto. Normalmente, as mudanças são feitas apenas no código fonte. Neste caso, estes artefatos ficam desatualizados e não acompanham a evolução da codificação do sistema. Assim, utilizar MDA na produção de um software

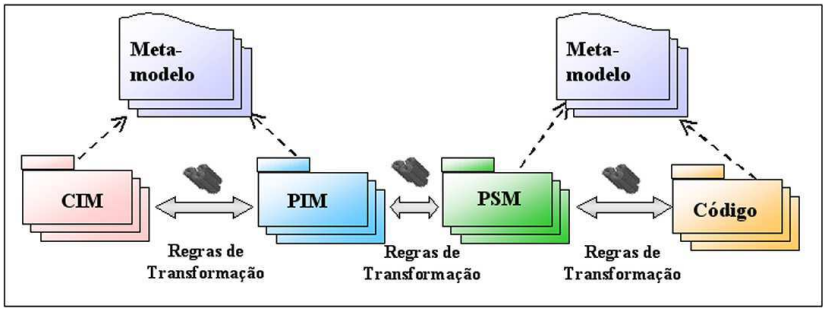

Figura 2.7: Arquitetura com Meta-modelos para MDA.

pode minimizar os custos de manutenção, pois qualquer mudança que haja na especificação, apenas é necessário realizar alterações nos modelos. Isto diminui o esforço com a etapa de codificação.

A filosofia de MDA pode ser utilizada para diferentes contextos, não apenas para a produção de sistemas. Pois, é possível definir meta-modelos para outros tipos de modelos, por exemplo, modelos de teste. Neste caso, os modelos são transformados em casos de teste, esta abordagem é conhecida por *Model-Driven Testing* (MDT) [BDG+07].

Para realizar as transformações de um modelo para outro pode utilizar qualquer linguagem de transformação, a exemplo das linguagens *Query Views Transformations* (QVT) [Gro07a], que é a linguagem padrão utilizada pelo OMG, e *Atlas Transformation Language* (ATL) [Pro09a], que tem sido bastante utilizada por ser bem documentada. Esta linguagem não é o padrão do OMG, mas tem um diferencial por possuir uma documentação completa, facilitando sua utilização pelos desenvolvedores. O Código Fonte 2.1 apresenta um modelo da linguagem ATL para transformação entre as linguagens *Python* e *Java*. Aquela regra ATL tem a finalidade de transformar um modelo descrito em *Python* para um modelo descrito em *Java*. Nas linhas 2 e 3 foram definidos os meta-modelos de saída (*Python*) e de entrada (*Java*), respectivamente. A linha 4 descreve o nome da regra para realizar a transformação das linguagens. A linha 6 apresenta uma instância do meta-modelo de entrada (*Python*) referente à classe “Class”. Na linha 8, a instância do meta-modelo de saída (*Java*) recebe no atributo “name” da classe “Class” o nome da classe do meta-modelo de entrada. Desta forma, ao transforma uma classe de *Python* em uma classe *Java*, elas terão o mesmo nome.

Código Fonte 2.1: Exemplo da Estrutura da Linguagem ATL.

```
1 Module Python2Java;
2 create OUT: Java
3 from IN: Python;
4 rule transformer {
5   from
6     py : Python!Class
7   to
8     jv : Java!Class (name <- py.name)
9 }
```

Neste trabalho proposto, a abordagem MDA foi bastante utilizada tanto para viabilizar a automação da técnica de inspeção guiada, pois permitiu baixar o nível de abstração dos artefatos utilizados na inspeção, como também foi utilizada para realizar a inspeção guiada automaticamente. Mais detalhes de como a automação foi realizada será apresentado no próximo capítulo.

2.4 Semântica de Ações

Uma ação representa uma unidade fundamental do comportamento de uma parte da especificação. Ela pode converter um conjunto de entradas em um conjunto de saídas modificando um estado do sistema. Semântica de ações [Gro07b] é um padrão, definido pelo *Object Management Group* (OMG), independente de plataforma, pois não possui uma linguagem concreta definida. Este padrão é um meta-modelo, que faz parte da especificação de UML2, o qual possui uma semântica bem definida com seus atributos, associações e restrições. Com isso, é possível descrever a semântica das ações de um sistema orientado a objetos, permitindo a geração automática de código em qualquer linguagem.

As ações podem ser do tipo criação ou destruição de objetos, leitura ou escrita de variáveis, chamada de eventos ou operações, associação entre objetos, entre outras. Na Figura 2.8 pode-se observar um trecho do meta-modelo para as ações *CreateObjectAction* e *DestroyObjectAction*. Estas ações, assim como as outras, herdam de um *Action*. Para a ação *CreateObjectAction*, há uma composição com a classe *OutputPin*, que é referente ao objeto a ser criado. No caso da ação *DestroyObjectAction*, a composição é feita com a classe *InputPin*, que é referente ao objeto alvo a ser removido.

Semântica de ações teve grande importância neste trabalho, pois permitiu transformar

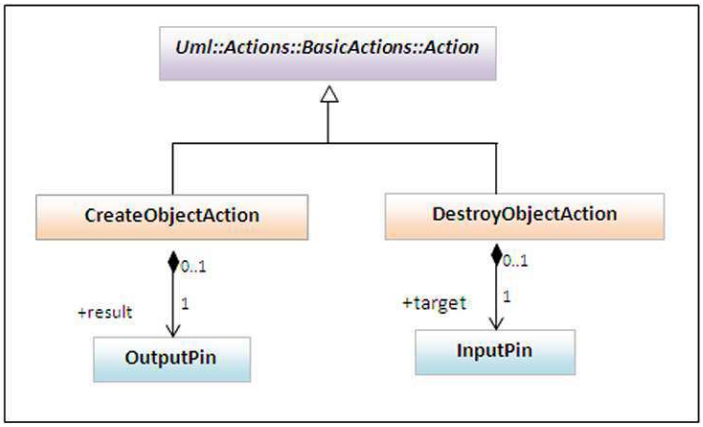

Figura 2.8: Exemplo do Meta-modelo de Semântica de Ações.

artefatos descritos em linguagem natural para um nível de abstração mais baixo. Pois, para cada ação presente nos artefatos, a semântica desta ação foi identificada com o padrão de semântica de ações de UML2.

2.5 Simulação de Modelos UML

As ferramentas que realizam simulação em modelos UML têm a finalidade de transformar um modelo UML (diagrama de classe, diagrama de seqüência, etc) em um modelo executável. Além disso, estas ferramentas permitem que os modelos sejam validados durante sua execução através de restrições (invariantes, pré e pós-condições) definidas em OCL *(Object Constraint Language)* [Gro05]. Na literatura existem algumas ferramentas para esta finalidade, como por exemplo, UMLAUT (*Unified Modeling Language All pUrposes Transformer*) [HJGP99], UMLAnt (*UML Animator and Tester*) [TKG+05] e USE (*UML-based Specification Environment*) [GBR07].

Dentre essas ferramentas, neste trabalho, a ferramenta USE foi utilizada para realizar a simulação dos modelos UML. Esta ferramenta foi escolhida por ser um software livre e por permitir a geração automática de diagramas de seqüência. Para executar os comandos na ferramenta USE e gerar um diagrama de seqüência existe uma linguagem específica, cuja sintaxe pode ser observada na Tabela 2.1. O usuário de USE deverá criar um script que contém todos os comandos a serem executados usando esta linguagem.

Além dessa linguagem para os comandos de USE, há também uma linguagem especí-

Sintaxe dos Comandos de USE	**Descrição**
!create object : ObjectType	Ação de criar um objeto.
!delete object	Ação de remover um objeto.
!set object.attribute := value	Ação de alterar o valor de uma variável.
? object.attribute	Ação de recuperar o valor de uma variável.
!insert (object1, object2) into Association	Ação de criar uma associação entre dois objetos.
!delete (object1, object2) from Association	Ação de remover a associação entre dois objetos.
!openter object operation()	Ação de chamada de execução de uma operação.
!opexit [return]	Ação de retorno de uma operação.

Tabela 2.1: Sintaxe da linguagem da ferramenta USE.

fica para descrever o diagrama de classes e as restrições OCL. A descrição do diagrama de classes, com todas as classes e associações, é essencial para execução dos comandos citados anteriormente. As restrições OCL ajudam na validação dos diagramas de seqüência gerados pela ferramenta após a execução dos comandos. Um exemplo dessa linguagem pode ser observado no Código Fonte 2.2.

Código Fonte 2.2: Exemplo da linguagem USE do diagrama de classes com extensão *.use*.

```
model ChessGame
enum PieceColor {White, Black}
class ChessGame
operations
  startNewMatch()
  requestsPlayersName(namePlayerWt: String, namePlayerBk: String)
  verifyValidName(playerName: String): Boolean
  addPlayer(namePlayer: String, points: Integer)
...
-- associations
composition HasPlayers between
  Match[0..*] role match
  Player[2] role player

association RecordOn between
  ChessGame[1] role chess
  Player[0..*] role player
...
-- invariantes
context Match
  -- Garantir que a partida será criada com 2 jogadores.
inv MatchHasNoMoreThan2Players:
   self.player->size() = 0 or self.player->size() = 2

-- pré e pós-condições
constraints
context ChessGame::startNewMatch()
  pre matchIsNotDefined: self.match->isEmpty()
  post playersRecordsOn: self.player->notEmpty()
  post matchStarted: self.match->notEmpty()
  post boardInitialized: self.board.isDefined()
  post piecesInitialized: self.board.piece->size() = 32
...
```

Para executar um script de comandos USE e gerar um diagrama de seqüência, é necessário digitar no console da ferramenta os comandos “open classDiagram.use” e “read script.cmd”, como pode ser visualizado na Figura 2.9. Então, o “script.cmd” é executado e então é gerado um diagrama de seqüência. Durante a execução, cada mensagem do diagrama de seqüência é avaliada de acordo com as restrições OCL pré-definidas para o diagrama de classes, no arquivo “classDiagram.use”, por exemplo. Isto garante uma maior qualidade dos diagramas de seqüência gerados, pois as restrições impedem que haja inconsistências entre os diagramas UML (de classe e de seqüência). Caso alguma invariante não seja satisfeita,

a ferramenta apresenta uma mensagem de alerta indicando o ponto do modelo que deverá ser corrigido. Caso alguma das pré ou pós-condições das operações não sejam satisfeitas, as mensagens do diagrama de seqüência são exibidas com uma cor vermelha durante a execução das operações.

Figura 2.9: Tela do console da ferramenta USE para execução de comandos.

A Figura 2.10 apresenta uma tela da ferramenta USE, onde do lado esquerdo são definidas as classes do sistema, as associações, as invariantes e as pré-/pós-condições. No centro da tela pode-se observar o diagrama de seqüência gerado após a interação entre os objetos das classes através da execução do script. Além disso, neste exemplo, é possível observar que há uma mensagem que não está de acordo com alguma restrição OCL, esta mensagem está em destaque na figura.

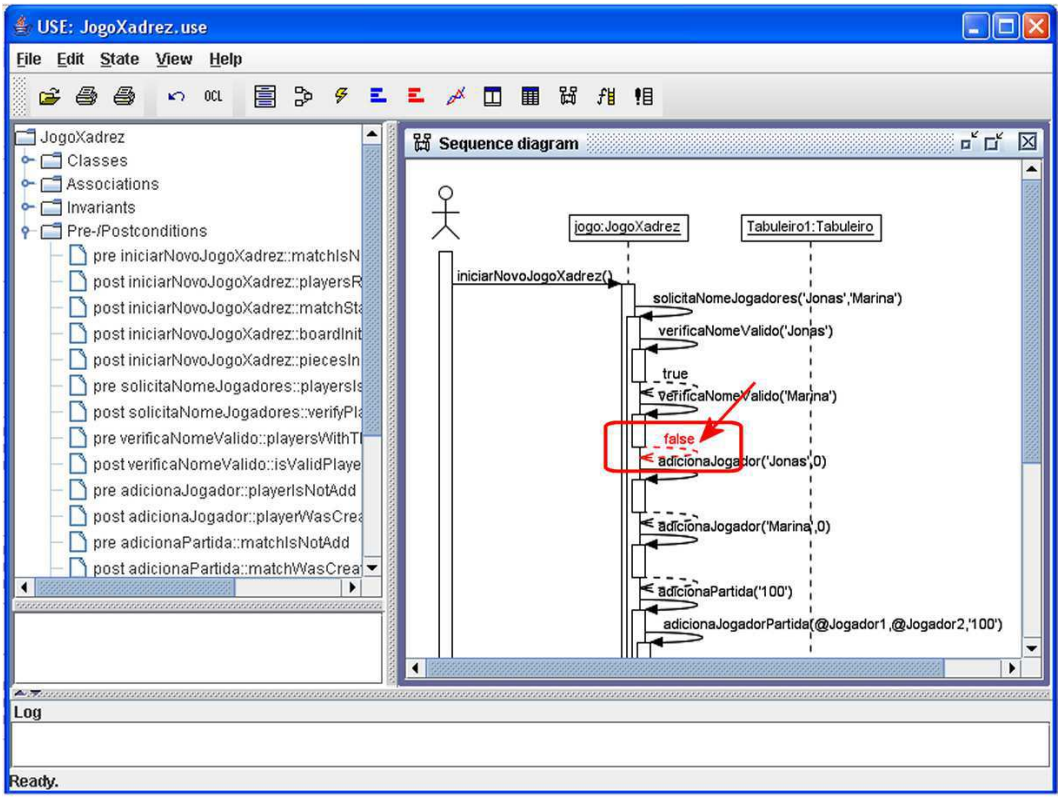

Figura 2.10: Tela da ferramenta USE

A ferramenta USE foi utilizada neste trabalho para criar instâncias de modelos estáticos, por exemplo, o diagrama de classes. Para então, avaliar a interação entre os objetos e validar a conformidade deste modelo com as restrições OCL pré-definidas de acordo com a especificação de requisitos.

2.6 Considerações Finais

Neste capítulo, foi apresentada a fundamentação teórica necessária para o entendimento deste trabalho. Foram descritos os principais conceitos sobre o processo de inspeção de software tradicional. Além disso, a abordagem MDA foi definida com seus quatro níveis de abstração, como também sua importância no desenvolvimento de software e na área de testes de software.

No caso de Semântica de Ações, é uma abordagem de suma importância para o trabalho, pois ela foi adaptada para recuperar a semântica dos casos de teste. Por fim, foram apresentadas algumas ferramentas que fazem simulação e validação de diagramas UML de projeto. Em nossa abordagem, os casos de teste foram transformados na linguagem de comandos de USE para que estes casos de teste pudessem ser executados na ferramenta USE. No próximo capítulo será apresentado como foi realizada a automação da técnica de inspeção guiada.

Capítulo 3

Automação da Inspeção Guiada

Este capítulo tem como objetivo apresentar a maneira como a técnica de inspeção guiada foi automatizada. Para automação foi necessário seguir uma seqüência de passos até conseguir realizar a inspeção entre os artefatos de projeto e a especificação de requisitos de forma automática.

3.1 A Técnica de Inspeção Guiada Automática

Este trabalho tem como propósito automatizar a etapa de *execução* da técnica de inspeção guiada manual, como pode ser observado na Figura 3.1. A técnica de inspeção guiada automática possui as mesmas entradas e saídas da inspeção guiada manual. As entradas são: (i) um conjunto de casos de teste abstratos, descritos em linguagem natural para cada caso de uso do sistema; (ii) diagramas de seqüência UML e o diagrama de classes UML ambos de projeto do sistema. Na inspeção guiada, a saída, após o processamento da inspeção, é um relatório de defeitos, o qual possui um conjunto de defeitos ou alertas encontrados nos diagramas de projeto inspecionados. Além disso, podem-se encontrar defeitos no diagrama de classes, caso haja alguma inconsistência entre este artefato e a especificação de requisitos durante os passos que antecedem a inspeção. Os defeitos encontrados pela técnica de inspeção guiada automática podem ser semânticos ou sintáticos. Os defeitos considerados semânticos são aqueles que identificam problemas comportamentais nos artefatos, ou seja, o comportamento presente em um artefato não está presente em outro. Os defeitos sintáticos são referentes a problemas de inconsistência entre a sintaxe de artefatos de um mesmo nível

de abstração, por exemplo, um método pode possuir dois atributos no diagrama de classes, mas possuir apenas um atributo no diagrama de seqüência. Na inspeção guiada manual geralmente são encontrados defeitos semânticos [MM99].

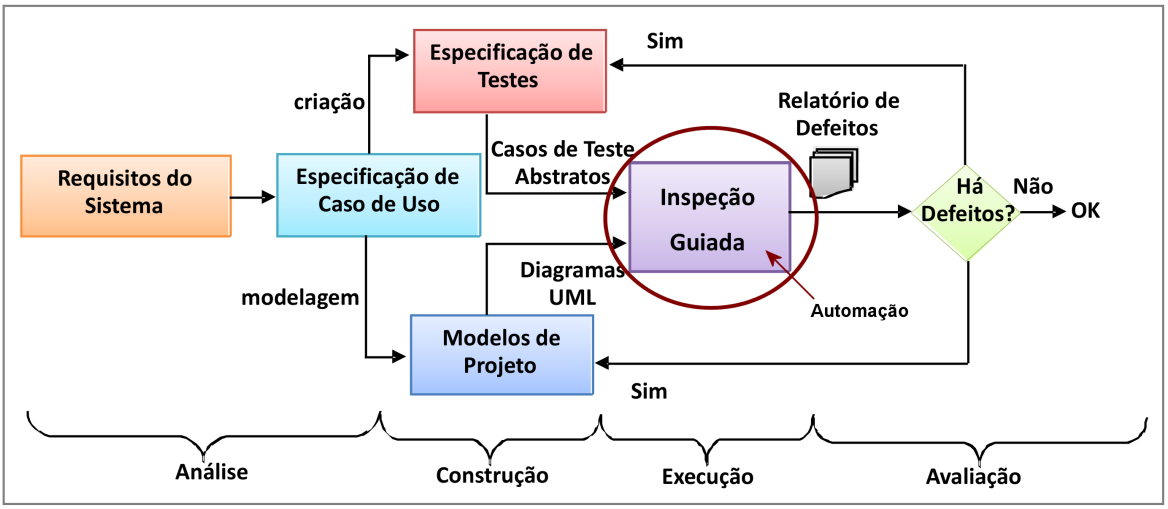

Figura 3.1: Etapas da Inspeção Guiada

Como a inspeção guiada manual é realizada com base em casos de teste, então para tornar possível a automação da técnica de inspeção guiada foi necessário transformar os casos de teste abstratos, ou seja, descritos em linguagem natural, em casos de teste executáveis. Com isso, estes casos de teste poderiam ser "executados" sobre os diagramas de projeto inspecionados.

Com os casos de teste formalizados em uma linguagem específica, é interessante utilizar alguma ferramenta que possa executar estes casos de teste. Então, existe a ferramenta USE [GBR07], descrita na Seção 2.5, que realiza simulação de modelos UML através da execução de comandos. Esta simulação é realizada com o objetivo de validar os modelos UML através da definição de restrições de integridade, descritas em OCL. As restrições são declaradas no diagrama de classes, que também é descrito na ferramenta. Após a execução dos comandos para realizar a simulação na ferramenta USE, ela gera automaticamente um diagrama de seqüência contendo a avaliação do modelo. Assim, caso alguma mensagem deste diagrama tenha infringindo alguma restrição OCL, então ela aparecerá com um destaque. Logo, será necessário fazer alterações nos modelos ou na ordem da execução dos comandos para que todas as restrições sejam satisfeitas. Desta forma, surgiu a idéia de converter cada passo dos casos de teste em um comando da ferramenta USE e então executar os casos de teste nesta

ferramenta.

Para converter os passos do caso de teste abstrato em um comando de USE, primeiro foi necessário identificar a semântica destes passos. Para isto, foi utilizado o conceito de Semântica de Ações UML2 [Gro07b]. Como este padrão não possui uma linguagem concreta, então algumas ações foram selecionadas e descritas em uma linguagem simples.

Assim, para que fosse possível converter o caso de teste já com semântica de ações para comandos de USE, foi necessário identificar a semântica de cada comando de USE. No entanto, não seria interessante que o inspetor que fosse realizar a inspeção guiada tivesse que aprender a sintaxe da ferramenta USE. Desta forma, foram realizadas transformações automáticas usando MDA. Com isso, foi definido um meta-modelo para os comandos de USE e com relação à semântica de ações foi utilizado apenas um subconjunto do meta-modelo definido para UML2.

Por fim, com o caso de teste transformado em comandos de USE, ele pôde ser executado nesta ferramenta. Como esta ferramenta gera diagramas de seqüência após a execução dos comandos, então na realidade os casos de teste que estavam em linguagem natural foram transformados em casos de teste no formato de diagrama de seqüência. Isto tornou a automação da inspeção guiada muito simples, pois foi possível realizar a inspeção nos diagramas de seqüência de projeto utilizando casos de teste que também estavam no mesmo nível de abstração.

A inspeção guiada em si foi automatizada utilizando a filosofia MDA, pois ela permite realizar transformações entre modelos. Definimos como modelos de entrada os diagramas de seqüência de caso de teste e os diagramas de seqüência de projeto, ambos descritos pelo meta-modelo de UML2. Então, estes modelos foram comparados a fim de verificar a conformidade entre eles (realizando a inspeção). Por fim, para o modelo de saída foi elaborado um meta-modelo para o relatório de defeitos, o qual contém todas as discrepâncias encontradas na inspeção.

Desta forma, foi definida uma seqüência de quatro passos para automatizar a técnica de inspeção guiada [RMR09], que são: (1) identificar a semântica de cada passo do caso de teste abstrato e anotar estes passos com a linguagem de semântica de ação estabelecida, (2) gerar automaticamente scripts dos casos de teste na linguagem da ferramenta USE através de transformações MDA sobre os casos de teste anotados com semântica de ações, (3) cadastrar

o diagrama de classes do sistema com suas restrições OCL na ferramenta USE e executar o script do caso de teste nesta ferramenta para ter como saída um caso de teste em formato de diagrama de seqüência e (4) realizar a inspeção guiada de forma automática utilizando regras de transformações MDA para comparar o diagrama de seqüência do caso de teste com o diagrama de seqüência de projeto e gerar um relatório de defeitos. A Figura 3.2 ilustra o fluxo de cada passo de acordo com a numeração. As caixas pontilhadas representam os artefatos que foram passados como entrada, que são: os casos de teste abstratos, o diagrama de classes e os diagramas de seqüência de projeto.

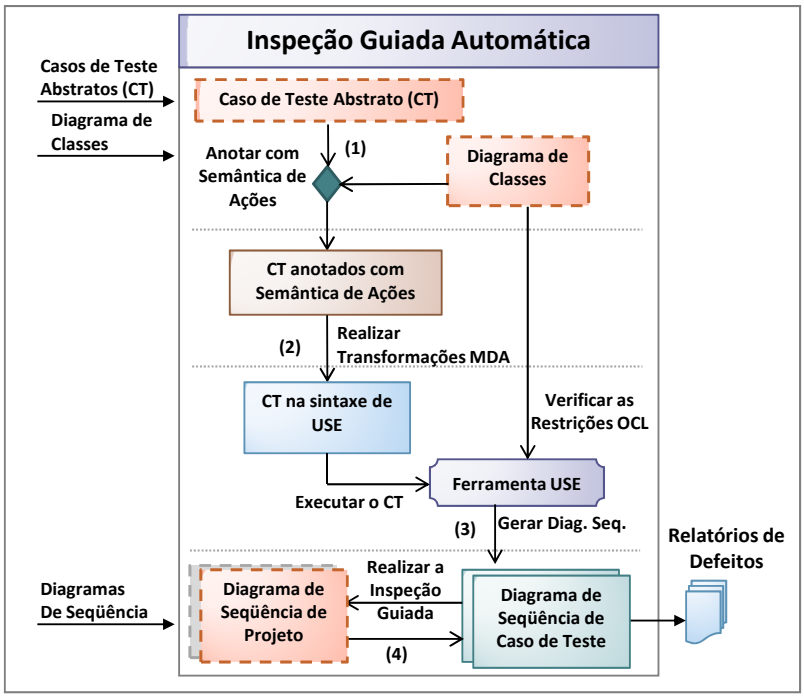

Figura 3.2: Atividades para realizar a automação da inspeção guiada.

Vale ressaltar que somente o passo 4 é referente à inspeção guiada dos artefatos, os outros passos são necessários apenas para se conseguir realizar a inspeção automaticamente. Este passo 4 é referente à inspeção entre cada diagrama de seqüência de caso de teste (gerados pela ferramenta USE no passo 3) e cada diagrama de seqüência de projeto, ambos diagramas estão descritos em XMI (*XML Metadata Interchange*). Para cada diagrama de seqüência de projeto, que representa um cenário de caso de uso do sistema, pode haver um ou mais casos de teste. Assim, a inspeção é realizada sobre cada diagrama de seqüência considerando a quantidade de casos de teste, ou seja, um mesmo diagrama de projeto pode ser inspecionado

várias vezes. As próximas subseções apresentam uma descrição detalhada sobre os passos 1, 2 e 3 para viabilizar a inspeção guiada automática e sobre o passo 4 referente à inspeção.

3.1.1 Passo 1: Anotando os Casos de teste com Semântica de Ações

De acordo com a técnica de inspeção guiada convencional, é necessário verificar se o comportamento de cada passo de um caso de teste está presente nos respectivos diagramas UML de projeto. Como os casos de teste abstratos estão descritos em linguagem natural, então o primeiro passo para a automação da inspeção guiada é identificar a semântica de cada passo de execução do sistema dos casos de teste abstratos, como foi dito anteriormente. Estes casos de teste foram anotados com Semântica de Ações de UML2 para serem transformados em casos de teste executáveis.

O procedimento utilizado para inserir a semântica de ações nos casos de teste é manual. Inicialmente, algumas ações da especificação de UML2 que poderiam ser necessárias para formalizar os casos de teste foram selecionadas e estão apresentadas na Tabela 3.1. A anotação dos casos de teste deve ser realizada com o auxílio do diagrama de classes de projeto, onde é possível identificar as classes e os métodos envolvidos por cada ação. Como se pode observar, na primeira linha há a descrição da semântica de ação <*CreateObjectAction*> referente à criação de um objeto, logo é necessário identificar o nome do objeto criado e o seu tipo. Por exemplo, a semântica de ação <*WriteVariableAction*> referente à atualização do valor de uma variável, em um caso de teste esta semântica seria descrita por <*WriteVariableAction*> *jogador pontos 10*.

Semântica de Ações	**Descrição da Semântica**
<CreateObjectAction> object ObjectType	Ação de criar um objeto.
<DestroyObjectAction> object	Ação de remover um objeto.
<WriteVariableAction> object variable value	Ação de alterar o valor de uma variável.
<ReadVariableAction> object variable	Ação de recuperar o valor de uma variável.
<CreateLinkObjectAction> object1 link object2	Ação de criar uma associação entre dois objetos.
<DestroyLinkAction> object1 link object2	Ação de remover a associação entre dois objetos.
<CallOperationAction> object operation parameters	Ação de chamada de execução de uma operação.
<ReplyAction> [return]	Ação de retorno de uma operação.

Tabela 3.1: Semântica de ações selecionadas com extensão.

3.1.2 Passo 2: Usando transformações MDA para gerar casos de teste executáveis

Com o intuito de executar os casos de teste sobre os modelos de projeto para realizar a inspeção guiada, foi necessário utilizar a ferramenta USE para execução destes casos de teste. Esta ferramenta possui uma sintaxe própria e após a execução dos casos de teste, gera automaticamente diagramas de seqüência da interação entre os objetos do sistema. Desta forma, os casos de teste abstratos, anotados com semântica de ações, tiveram que ser transformados em casos de teste na sintaxe da ferramenta USE. Isto permite realizar a inspeção entre artefatos de mesmo nível de abstração, ou seja, inspeção entre diagramas de seqüência de caso de teste (gerado por USE) e diagramas de projeto (fornecidos como entrada previamente), tornando a inspeção mais simples.

Para realizar tal geração automática, foi adotada uma abordagem MDA porque permite realizar transformações entre modelos. Nesse sentido, se fez necessário reusar parte do meta-modelo de semântica de ações e criar um meta-modelo para a sintaxe da ferramenta USE.

Então, para que essas transformações fossem possíveis, a semântica de cada comando da linguagem da ferramenta USE também foi identificada com o padrão de semântica de ações, como pode ser observado na Tabela 3.2. Nesta tabela, a coluna da direita apresenta os comandos da sintaxe de USE, por exemplo, o comando *!create* tem a mesma semântica da ação *CreateObjectAction*, assim como o comando *!set* tem a mesma semântica da ação *WriteVariableAction*. Assim, através de transformações MDA foi possível gerar casos de teste na sintaxe da ferramenta USE, que podem ser executados de forma automática na ferramenta USE.

As transformações MDA permitem transformar modelos abstratos em outros modelos através de regras de transformação sobre os meta-modelos e estas regras foram descritas em ATL (*ATLAS Transformation Language*), pois é bastante utilizada atualmente e é muito bem documentada. O procedimento para a transformação dos modelos foi composto por 4 etapas: (i) criação dos meta-modelos de entrada e de saída; (ii) criação da instância do modelo de entrada; (iii) definição das regras de transformação em ATL; (iv) geração do modelo de saída e (v) definição das regras em MOFScript para gerar os casos de teste na linguagem de USE. A arquitetura de MDA para transformação dos casos de teste com semântica de ações em

Semântica de Ações com Extensão	**Sintaxe dos Comandos de USE**
<CreateObjectAction> object ObjectType	!create object : ObjectType
<DestroyObjectAction> object	!delete object
<WriteVariableAction> object variable - value	!set object.attribute := value
<ReadVariableAction> object variable	? object.attribute
<CreateLinkObjectAction> object1 link object2	!insert (object1, object2) into Association
<DestroyLinkObjectAction> object1 link object2	!delete (object1, object2) from Association
<CallOperationAction> object operation parameters	!openter object operation()
<ReplyAction> [return]	!opexit [return]

Tabela 3.2: Relação entre a semântica de ações e a sintaxe de USE.

casos de teste em comandos da linguagem USE pode ser visualizada na Figura 3.3.

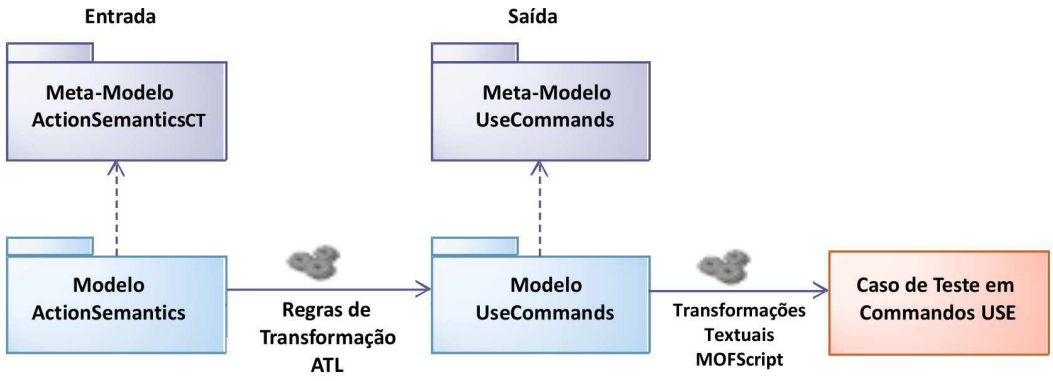

Figura 3.3: Arquitetura MDA para transformação de semântica de ações em comandos USE.

Na etapa de criação dos meta-modelos de entrada e saída, o meta-modelo de semântica de ações foi reusado como entrada, disponibilizado pelo OMG. No entanto, este meta-modelo foi estendido com o pacote *ActionSemanticsCT*, conforme ilustra a Figura 3.4, para adaptá-lo ao conceito de especificação de testes. Desta forma, no pacote *ActionSemanticsCT*, um caso de teste (*TestCaseAction*) é composto por um conjunto de comandos (*CommandsAction*), os quais são compostos por um conjunto de ações do tipo *Action* reusadas do pacote *BasicActions*.

Para realizar a transformação nos modelos foi necessário construir um meta-modelo de saída, que representa uma especificação de testes na linguagem de comandos de USE, pois não havia um meta-modelo para esta linguagem. O pacote *UseCommands* também contém a estrutura de um *suite* de teste que é composta por um conjunto de casos de teste, os

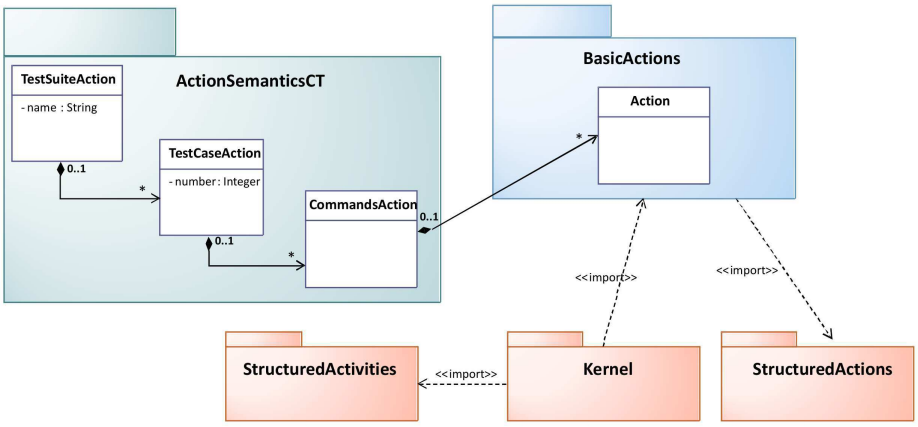

Figura 3.4: Extensão do meta-modelo de semântica de ações para casos de teste.

quais são compostos por um conjunto de comandos da linguagem USE. A hierarquia de comandos de USE é refletida no meta-modelo ilustrado na Figura 3.5. Por exemplo, a classe *CreateObjectCmd* do pacote *UseCommands* é equivalente à classe do *CreateObjectAction* do pacote *BasicAction*, outro exemplo é com relação à estrutura dos casos de teste, no caso do pacote *UseCommands* tem a classe *TestCase* e no pacote *ActionSemanticsCT* a classe equivalente a esta é *TestCaseAction*.

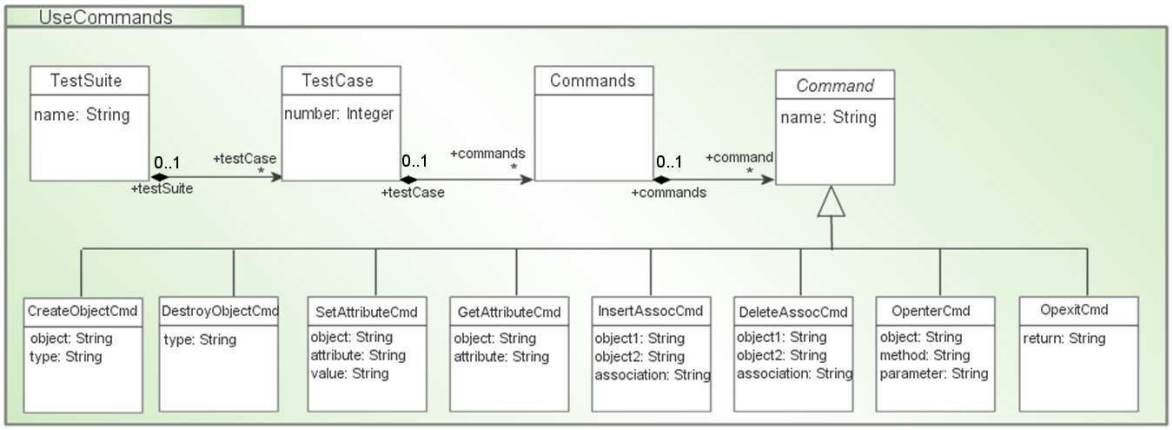

Figura 3.5: Meta-modelo para os comandos da linguagem USE.

O Código Fonte 3.1 ilustra parte das regras de transformação utilizadas. A linha 2 define os meta-modelos de entrada (*ActionSemanticsCT*) e saída (*UseCommands*). Na linha 3, tem-se a definição da regra *createTestSuite*, que cria uma *TestSuite* do meta-modelo *UseCommands* (linha 7) a partir de uma *TestSuiteAction* (linha 4) encontrada no meta-modelo de entrada. De acordo com o meta-modelo estendido de *ActionSemantics*, uma *Test-*

SuiteAction é composta por um conjunto de *TestCaseAction*, que por sua vez é composto por um conjunto de *Action*. Assim, para cada *Action* encontrada no modelo, identifica-se o seu tipo e então é feita uma transformação para o comando de USE equivalente. Por exemplo, na linha 17 identifica-se uma *Action* do tipo *CreateObjectAction* que será transformada (através de outra regra ATL chamada *createObjectUseCommand*) na classe *CreateObjectCmd* do meta-modelo de *UseCommands*.

Código Fonte 3.1: Regras ATL de transformação de semântica de ações para sintaxe USE.

```
1  module ActionSemantics2UseCommands;
2  create OUT : UseCommands from IN : ActionSemanticsCT;
3  rule createTestSuite {
4    from tsAs : ActionSemanticsCT!TestSuitAction
5    using { testCasesAc : ActionSemanticsCT!TestCaseAction = tsAs.testCase; }
6    to
7       tsUse : UseCommands!TestSuite(name <- tsAs.name, testCase <- tcUse),
8       tcUse : UseCommands!TestCase(),
9       cmdsUse : UseCommands!Commands()
10   do {
11      for (tcAc in testCaseAc) {
12        if (tcAc.oclIsTypeOf(ActionSemanticsCT!TestCaseAction)) {
13          self.testCaseUse(tcAc, tcUse);
14          for (cmdsAc in tcAc.commands) {
15            tcUse.commands <- tcUse.commands->including(cmdsUse);
16            for (action in cmdsAc.action) {
17              if(action.oclIsTypeOf(ActionSemanticsCT!CreateObjectAction)) {
18                self.createObjectUseCommand(action, cmdsUse);
19              }...
20   } ...
21 }
```

Para executar as regras de transformação em ATL é necessário que haja um modelo de entrada com valores de instâncias para as entidades presentes no meta-modelo de entrada estendido de semântica de ações. No Código Fonte 3.2, a linha 1 representa a entidade *TestSuiteAction* e tem como nome "Jogo de Xadrez". Na linha 2 encontra-se o número do caso de teste. Entre as linhas 3 e 8 há o conjunto de comandos *Action*, no caso do exemplo há apenas uma ação do tipo *CreateObjectAction*, cujo nome da classe é "JogoXadrez" (linha 5) e o nome do objeto é "jogo" (linha 6).

Código Fonte 3.2: Exemplo de modelo de entrada de semântica de ações.

```
1  <ActionSemanticsCT:TestSuiteAction xmi:version=''2.0'' xmlns:ActionSemantic=''
       ActionSemantics'' xmlns:BasicActions=''BasicActions'' name=''Jogo de Xadrez''>
```

```
2   <testCase number=''1''>
3     <commands>
4       <action xsi:type=''BasicActions:CreateObjectAction'' name=''CreateObjectAction''>
5         <owner xsi:type=''Kernel:Classifier'' name=''JogoXadrez''/>
6         <input name=''jogo'' qualifiedName=''jogo''/>
7       </action>
8     </commands>
9   </testCase>
10 </ActionSemanticsCT:TestSuiteAction>
```

O modelo de saída, apresentado no Código Fonte 3.3, é gerado a partir da execução das regras de transformação sobre o modelo de entrada e portanto são equivalentes com relação aos valores das instâncias. Por exemplo, o atributo *type* (linha 4) do comando *CreateObjectCmd* do modelo de saída possui o valor "JogoXadrez", o qual tem o mesmo nome da classe da ação *CreateObjectAction* do modelo de entrada na linha 5.

Código Fonte 3.3: Exemplo de modelo de saída de um caso de teste na linguagem USE.

```
1 <xmi:XMI xmi:version=''2.0'' xmlns:UseCommands=''UseCommands''>
2   <UseCommands:TestSuite name=''Jogo de Xadrez''>
3     <UseCommands:TestCase number=''1''>
4       <UseCommands:CreateObjectCmd name=''!create'' obj=''jogo'' type=''JogoXadrez''/>
5     </UseCommands:TestCase>
6   </UseCommands:TestSuite>
7 </xmi:XMI>
```

Após a execução das regras de transformação ATL, um modelo da linguagem de comandos de USE é gerado. No entanto, este modelo ainda não está na sintaxe concreta da linguagem USE, por isso a ferramenta *MOFScript* [Pro09b] foi utilizada para transformar o modelo em uma linguagem concreta, como se pode observar no Código Fonte 3.4.

Código Fonte 3.4: Exemplo do caso de teste na sintaxe de USE gerado através de transformações textuais.

```
1 -- Jogo de Xadrez
2 -- Caso de Teste #1
3 !create jogo : JogoXadrez
```

3.1.3 Passo 3: Gerando um diagrama de seqüência na ferramenta USE a partir da execução de um caso de teste

Neste passo, os casos de teste transformados no passo 2 podem ser executados na ferramenta USE. No entanto, para que estes casos de teste possam ser validados e garantir uma inspeção mais eficiente, pois como a inspeção guiada é feita com base nos casos de teste, se eles tiverem qualidade, a inspeção também terá. Para a validação dos casos de teste é necessário que na ferramenta USE tenham sido cadastradas algumas restrições OCL (invariantes, pré- e pós-condições) para as operações do diagrama de classes, um exemplo das restrições OCL poderá ser visualizado no Código Fonte 3.5. Este diagrama deve ser cadastrado conforme a sintaxe própria de USE em um arquivo com extensão ".use". As restrições geralmente definem quais objetos devem ter sido criados antes e após a execução de uma operação.

Nesse código, pode-se observar entre as linhas 2 e 4 uma parte da declaração da classe JogoXadrez, continuando entre as linhas 7 e 9 identifica-se um exemplo de associação entre as classes JogoXadrez e Jogador. Também é possível definir as restrições de pré- e pós-condições para cada método do diagrama de classes, isto pode ser visualizado entre as linhas 13 e 16, que possui um exemplo de restrições para o método "*iniciarNovoJogoXadrez*" da classe "JogoXadrez". Como pré-condição (linha 14) há uma restrição para que antes da execução deste método não haja nenhuma partida já iniciada. Após a execução deste mesmo método há duas pós-condições (linhas 15 e 16) para garantir que foram criados dois jogadores para aquele jogo e também uma nova partida. vale ressaltar que este código apresenta apenas um exemplo, pois na realidade é necessário descrever todas as classes e associações, assim como outras restrições OCL para cada método.

Código Fonte 3.5: Exemplo da linguagem USE para o diagrama de classes e restrições OCL.

```
1  model JogoXadrez
2  class JogoXadrez
3  operations
4    iniciarNovoJogoXadrez()
5  ...
6  -- associations
7  association Participam between
8    Jogador [0..2] role jogador
9    JogoXadrez[1] role jogo
10 ...
11 -- pré e pós-condições
```

```
12 constraints
13 context JogoXadrez::iniciarNovoJogoXadrez()
14   pre matchIsNotDefined: self.partida->isEmpty()
15   post playersAssoc: self.jogador->size() = 2
16   post matchStarted: self.partida->notEmpty()
17 ...
```

Assim, à medida que os casos de teste são executados, cada passo de execução é validado através das restrições OCL. Após a execução de todos os comandos do caso de teste, um diagrama de seqüência é gerado pela ferramenta USE, no qual cada mensagem representa a execução de cada passo do caso de teste.

Este passo 3 poderá ser executado quantas vezes forem necessárias até que todos os comandos do caso de teste estejam de acordo com as restrições OCL definidas no diagrama de classes da ferramenta USE.

É importante notar que um caso de teste pode ser inválido com relação às restrições OCL especificadas, ou seja, podem induzir a ações que seriam inválidas durante a verificação das restrições. Neste caso, há duas possibilidades: i) o caso de teste é inválido e precisa ser revisto, juntamente com o caso de uso correspondente (defeito no documento de requisitos); ii) as restrições OCL anotadas no diagrama de classes precisam ser revistas (defeito no diagrama de classes). Assim, tanto os casos de teste quanto o próprio modelo estrutural passam por uma revisão neste processo. Desta forma, serão levados para inspeção apenas os diagramas de caso de teste executados com sucesso.

3.1.4 Passo 4: Realizando inspeção no diagrama de seqüência de projeto

A inspeção propriamente dita foi automatizada utilizando a linguagem ATL para descrever as regras de transformação. Para realizar as transformações usando ATL foi necessário passar como entrada o meta-modelo de UML2 e como saída o meta-modelo do relatório de defeitos. As instâncias do meta-modelo de entrada são o modelo do diagrama de seqüência do caso de teste e o modelo do diagrama de seqüência de projeto.

Com ATL é possível realizar dois tipos de tranformações: as *called rules* e as *matched rules*. As *called rules* são código imperativos e foram utilizadas para comparar o diagrama de seqüência de caso de teste (TC) com o diagrama de seqüência de projeto (SD), ambos

modelos UML2 de entrada. Ao comparar as mensagens destes dois diagramas, a inspeção foi realizada. À medida que os defeitos ou alertas foram ocorrendo, então eles foram transformados em um relatório de defeitos. Esta transformação do modelo de entrada no modelo de saída é feita através das *matched rules*.

O meta-modelo do relatório de defeitos (saída) para a inspeção guiada é apresentado na Figura 3.6. Este meta-modelo faz um merge com o pacote *BasicIteractions* de UML2, pois as classes *Message* e *MessageEnd* deste pacote são utilizadas para descrever os defeitos. O relatório é composto por duas classes, uma para relatar os defeitos e outra os alertas.

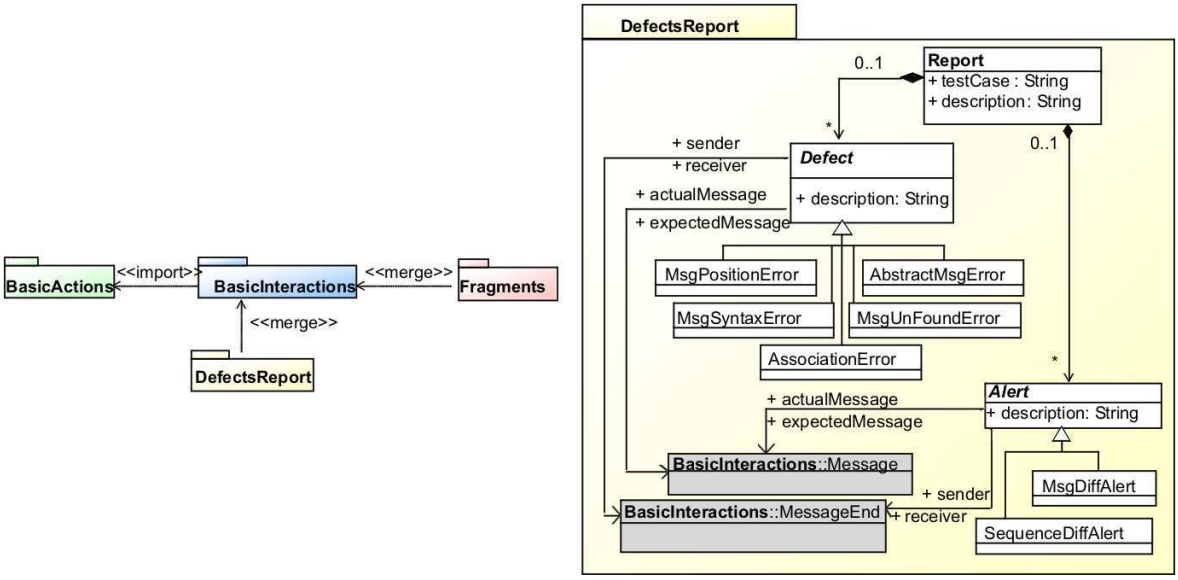

Figura 3.6: Meta-modelo do relatório de defeitos

Os defeitos apresentados no relatório de defeitos da Figura 3.6 são basicamente as mensagens particulares, que aparecem no diagrama de seqüência do caso de teste e não aparecem no diagrama de seqüência de projeto, portanto são consideradas como erro. Os tipos de defeitos que estão sendo tratados são:

- *MsgPositionError*: referente à uma mensagem, cujas linhas de vida de envio ou de recebimento são diferentes;
- *MsgSyntaxError*: referente à uma mensagem que possua a sintaxe diferente;
- *MsgUnfoundError*: referente às mensagens não encontradas;

- *AbstractMsgError*: referente à ocorrência da criação de um objeto, cuja classe deste objeto seja abstrata;

- *AssociationError*: referente à ocorrência de mensagens que não poderiam estar associadas no diagrama de seqüência, pois não há nenhuma associação entre os objetos no diagrama de classes.

Os alertas não são considerados um erro, pois são mensagens que aparecem no diagrama de seqüência de projeto e não aparecem no diagrama de caso de teste. Com relação à seqüência das mensagens estarem em ordens diferentes, não se pode afirmar que é um erro, portanto também é considerado como um alerta. Os tipos de alertas são:

- *MsgDiffAlert*: referente às mensagens diferentes encontradas no diagrama de seqüência de projeto;

- *SequenceDiffAlert*: referente às mensagens que estiverem em uma ordem diferente.

O Código Fonte 3.6 apresenta uma parte das regras ATL para a inspeção guiada. Nas linhas 2 e 3 encontra-se os meta-modelos de saída (*DefectsReport*) e de entrada (UML2), respectivamente. A linha 4 descreve o nome da regra principal para a inspeção. As linhas 6 e 7 apresentam as instâncias dos meta-modelos de entrada e na linha 9 a instância do meta-modelo de saída. Estas instâncias são passadas como entrada na chamada da regra da inspeção guiada (linha 11). Na regra *guidedInspection* é feito uma iteração sobre os dois diagramas de seqüência passados na entrada para verificar a conformidade entre suas mensagens.

Código Fonte 3.6: Regra ATL para inspeção guiada.

```
1  module GuidedInspection;
2  create OUT : DefectsReport
3  from TC : uml2, SD : uml2;
4  rule inspection{
5    from
6      tc : uml2!Model
7    using { sd : uml2!Model = ''''; }
8    to
9      rep:DefectsReport!Report(testCase <- 'TestCase description')
10   do {
11     self.guidedInspection(tc, sd, rep);
12   }
```

Para cada tipo de defeito encontrado foi definido um conjunto de instruções que definem uma regra, como por exemplos, a regra *reportMsgUnFoundError* apresentada no Código Fonte 3.7. Esta regra é referente a alguma mensagem que não foi encontrada no diagrama de seqüência de projeto inspecionado. Na linha 1 encontra-se a assinatura da regra com seus parâmetros de entrada. A linha 4 apresenta o nome da classe (*MsgUnFoundError*) do meta-modelo de relatório de defeitos que deve ser atualizada com seus respectivos atributos descritos entre as linhas 5 e 7. Na linha 9, o defeito é adicionado no modelo do relatório de defeitos, cujo modelo foi passado como parâmetro (definido como "rep") na regra *guidedInspection(tc, sd, rep)* da linha 13 do Código Fonte 3.6.

Código Fonte 3.7: Regra ATL o defeito do tipo *MsgUnFoundError*.

```
1  rule reportMsgUnFoundError(senderTC : String,
2      receiverTC : String, msgTC : String, rep : DefectsReport){
3    to
4      unFound : DefectsReport!MsgUnFoundError(
5        actualSender <- senderTC,
6        actualReceiver <- receiverTC,
7        expectedMessage <- msgTC)
8    do {
9      rep.defect <- rep.defect -> including(unFound);
10   }
11 }
```

Um exemplo do relatório de defeitos pode ser observado no Código Fonte 3.8. Este relatório de defeitos encontra-se descrito em XMI e possui um defeito e um alerta. No Código Fonte 3.8, a linha 3 apresenta o nome do caso de teste executado. Nas linhas 4 e 5 há uma descrição do defeito encontrado no diagrama de seqüência de projeto, o qual é do tipo *MsgUnFoundError* e possui os atributos *actualSender* e *actualReceiver*, que representam as linhas de vida da mensagem em questão e o atributo *expectedMessage*, que é referente ao nome da mensagem esperada. Há também a descrição de um alerta nas linhas 6 e 7. Este alerta é do tipo *MsgDiffAlert*, que indica a presença da mensagem *getPontos* (linha 7) que está presente no diagrama de seqüência inspecionado, mas não está presente no diagrama de seqüência do caso de teste.

Código Fonte 3.8: Exemplo do relatório de defeitos no formato XML.

```
1  <?xml version=''1.0''?>
2  <?xml-stylesheet type=''text/xsl'' href=''defectsreport.xsl''?>
3  <Report testCase=''TestCase01''>
```

```
4   <defect type=''MsgUnFoundError''
5       actualSender=''Usuario'' actualReceiver=''JogoXadrez''expectedMessage=''
            iniciarNovoJogoXadrez''/>
6   <alert type=''MsgDiffAlert''
7       actualMessage=''getPontos'' actualSender=''Jogador'' actualReceiver=''Jogador''/>
8 </Report>
```

O relatório de defeitos gerado é independente de plataforma e descrito em XMI, o que permite adaptá-lo a qualquer linguagem. Com o relatório de defeitos em XMI não é fácil identificar os defeitos, pois os dados ficam confusos. Com isso, foi necessário criar uma maneira de visualizar o relatório de defeitos. O relatório visual foi elaborado utilizando a linguagem de transformação *Extensible Stylesheet Language Transformation* (XSLT) [Cla99]. Desta forma, o relatório de defeitos em XMI pode ser facilmente visualizado em um navegador apenas adicionando a tag "*xml-stylesheet*" presente na linha 2 do Código Fonte 3.8, que contém o XSL (*defectsreport.xsl*) que descreve o estilo do relatório.

Na Figura 3.7 há um exemplo da interface gráfica do relatório de defeitos, onde o inspetor poderá observar quais os defeitos e alertas encontrados nos diagramas de seqüência inspecionados. Como se pode se notar, os valores presentes no XML também estão presentes neste relatório de defeitos. A linha 1 contém um defeito do tipo *MsgUnFoundError* e a linha 2 contém um alerta do tipo *MsgDiffAlert*. Cada tipo possui uma imagem associada, de acordo com a legenda, para facilitar na identificação do defeito ou alerta.

DEFECTS REPORT

TestCase	Test Case 01			**Description**	Iniciar um novo jogo de xadrez.				
	MsgUnFoundError		MsgSyntaxError		MsgPositionError		AbstractMsgError		MsgDiffAlert
Number	**Type**	**Description**		**Actual Sender**	**Actual Message**	**Actual Receiver**	**Expected Sender**	**Expected Message**	**Expected Receiver**
1		The expected message **iniciarNovoJogoXadrez** should be in the sequence diagram, but it was not found.		Usuario		JogoXadrez		iniciarNovoJogoXadrez	
2		The message **getPontos** was not found in the test case sequence diagram.		Jogador	getPontos	Jogador			

Figura 3.7: Exemplo da interface gráfica do relatório de defeitos.

3.2 Considerações Finais

Este capítulo apresentou a seqüência de passos necessários para realizar a técnica de inspeção guiada de forma automática. A inspeção automática acontece apenas no Passo 4, onde os diagramas seqüência de caso de teste gerados pela ferramenta USE, no Passo 3, são utilizados para inspecionar os diagramas de seqüência de projeto. Os Passos 1, 2 e 3 são importantes para transformar os casos de teste abstratos, descritos em linguagem natural, em casos de teste executáveis, o que torna possível a automação da inspeção guiada. A inspeção guiada pode trazer grandes benefícios para detecção de defeitos, tanto na especificação de caso de uso quanto nos artefatos sendo inspecionados. Em um processo de especificação/projeto manual, erros podem ser comumente cometidos. A automação torna o processo mais preciso e confiável, visto que a validação dos casos de teste é fundamental para garantir uma correta interpretação dos resultados. Próximo capítulo ilustra um exemplo de utilização da técnica de inspeção guiada automática.

Capítulo 4

Exemplo de Uso da Técnica de Inspeção Guiada Automática

Neste capítulo é apresentado um exemplo de como realizar a técnica de inspeção guiada automática. Este exemplo será ilustrado com um sistema de Jogo de Xadrez, descrito na próxima seção. Este sistema possui a especificação de casos de uso, o diagrama de classes e os diagramas de seqüência para cada caso de uso do sistema. O exemplo será elaborado através dos quatro passos para automatizar a inspeção guiada. O detalhamento de cada um destes passos poderá ser observado nas seções subseqüentes.

4.1 Especificação do Sistema Jogo de Xadrez

O sistema jogo de xadrez é composto por um tabuleiro, só pode ser jogado por 2 jogadores e possui regras para o movimento das peças. O objetivo do jogo é analisar e imaginar estratégias para conseguir atacar o rei adversário de forma que se aplique um xeque-mate, ou seja, que o rei não possa se livrar do ataque. Na Figura 4.1 pode-se observar o diagrama de casos de uso que representam as interações entre os atores e o sistema.

A descrição do caso de uso “Iniciar Jogo de Xadrez” pode ser observada na Tabela 4.1. Este caso de uso possui dependência com os casos de uso ”Criar Partida”, “Adicionar Jogadores” e “Exibir Tabuleiro”.

A Figura 4.2 apresenta o diagrama de classes para o sistema Jogo de Xadrez. A classe “JogoXadrez” inicia o jogo e permite cadastrar os jogadores em uma nova partida.

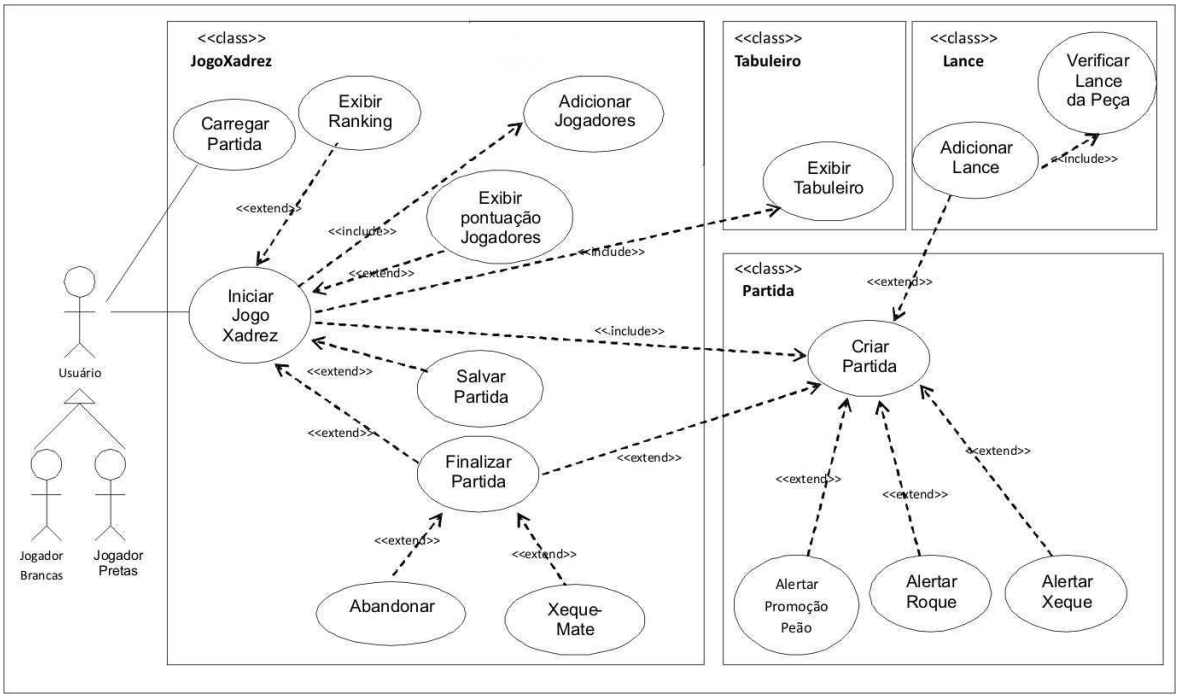

Figura 4.1: Diagrama de Caso de Uso do Sistema de Jogo de Xadrez.

Especificação de Caso de Uso
Caso de Uso: Iniciar Jogo de Xadrez **ID:** UC 001 **Ator:** Usuário **Descrição:**Iniciar uma nova partida do jogo de xadrez.
Fluxo Principal:
1. O usuário clica no botão “Novo” para iniciar um novo jogo.
include:: Adicionar Jogadores
2. O sistema solicita os nomes dos 2 jogadores. 3. O usuário informa o nome do jogador Brancas. 4. O usuário informa o nome do jogador Pretas. 5. O sistema verifica se os nomes dos jogadores são válidos. 6. O sistema adiciona os 2 jogadores no jogo.
include:: Criar Partida
7. O sistema cria uma partida. 8. O sistema adiciona os jogadores na partida.
include:: Exibir Tabuleiro
9. O sistema exibe o tabuleiro com as peças nas posições iniciais.

Tabela 4.1: Descrição do fluxo principal do caso de uso Iniciar Jogo de Xadrez.

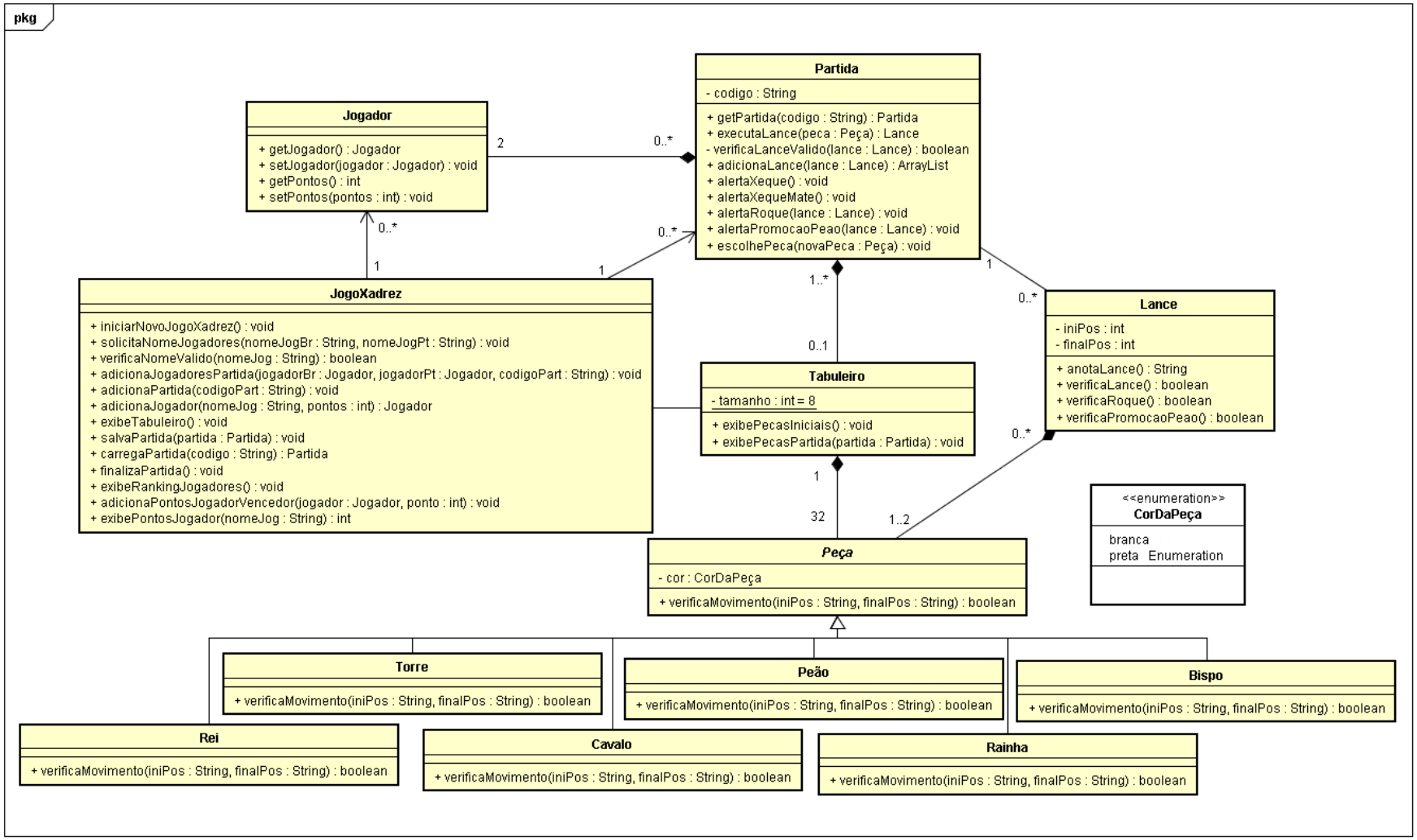

Figura 4.2: Diagrama de Classes para o Sistema de Jogo de Xadrez.

4.2 Passo 1: Anotar com Semântica de Ações os Casos de Teste

Para realizar o primeiro passo é necessário que o inspetor ou o engenheiro de testes elabore os casos de teste abstratos para cada cenário de caso de uso do sistema. O caso de uso *Iniciar Jogo de Xadrez* será utilizado para ilustrar a inspeção e está presente na Tabela 4.1 da seção anterior. Os casos de teste também são descritos em linguagem natural e devem representar todos os passos do caso de uso.

Para que possam ser executados, os casos de teste deverão ser anotados com semântica de ações. Esta anotação deverá se feita com base no diagrama de classes (Figura 4.2), pois é necessário identificar o relacionamento dos objetos e seus respectivos métodos.

Um exemplo da anotação pode ser visualizada na Tabela 4.2, onde há a descrição do caso de teste em linguagem natural e em semântica de ações.

No passo 1 do caso de teste, foi necessário criar duas linhas de semântica de ação, pois para realizar a chamada do método *iniciarNovoJogoXadrez* da classe *JogoXadrez* referente à semântica <*CallOperationAction*>. Antes é necessário criar um objeto do tipo *JogoXadrez*, para isso temos a semântica <*CreateObjetctAction*> *jogo JogoXadrez*. As outras anotações seguem este mesmo raciocínio.

É importante notar que cada passo do caso de teste pode possuir mais de uma semântica de ação equivalente. Além disso, quando o passo for uma ação do usuário não é necessário identificar uma semântica de ação. Pois, a ação do usuário pode ser passada como parâmetro em algum método de outro passo. Esta anotação provavelmente será modificada após a execução dos próximos passos da inspeção.

Neste passo, os casos de teste em linguagem natural são anotados com semântica de ações. Isto facilita na transformação de um caso de teste executável.

Caso de Teste	
Cenário: Iniciar Jogo de Xadrez **Número:** 001 **Dependências:** Adicionar Jogadores - Criar Partida - Exibir Tabuleiro **Descrição:** Iniciar uma nova partida do jogo de xadrez.	
1. O usuário clica no botão "Novo" para iniciar o jogo.	1. <CreateObjectAction> jogo JogoXadrez 1.1. <CallOperationAction> jogo iniciarNovoJogoXadrez()
2. O sistema solicita os nomes dos 2 jogadores.	2. <CallOperationAction> jogo solicitaNomeJogadores('Jonas','Maria')
3. O usuário informa os nomes dos jogadores.	—
4. O sistema verifica se os nomes dos jogadores são válidos.	4. <CallOperationAction> jogo verificaNomeValido('Jonas') 4.1. <CallOperationAction> jogo verificaNomeValido('Maria')
5. O sistema adiciona os 2 jogadores no jogo.	5. <CallOperationAction> jogo adicionaJogador('Jonas', 0) 5.1. <CreateObjectAction> jogBr Jogador 5.2. <CallOperationAction> jogo adicionaJogador('Maria', 0) 5.3. <CreateObjectAction> jogPt Jogador
6. O sistema cria uma partida.	6. <CallOperationAction> jogo adicionaPartida('01')
7. O sistema adiciona os jogadores na partida.	7. <CallOperationAction> jogo adicionaJogadoresPartida('Jonas', 'Maria', '01')
8. O sistema exibe o tabuleiro com as peças nas posições iniciais.	8. <CallOperationAction> jogo exibeTabuleiro() 8.1. <CallOperationAction> tabuleiro exibePecasIniciais()
Resultado Esperado: O jogo é iniciado com sucesso.	<ReplyAction>

Tabela 4.2: Exemplo de um caso de teste anotado com semântica de ações.

4.3 Passo 2: Gerar caso de teste na linguagem da ferramenta USE

Este passo é referente a criação do caso de teste na sintaxe da ferramenta USE com o intuito de executá-los nesta ferramenta para gerar um diagrama de seqüência. Para isso, é necessário formalizar os casos de teste anotados com semântica de ações para o formato XMI, conforme pode ser visualizado no Código Fonte 4.1. O primeiro passo do caso de teste pode ser observado entre as linhas 5 e 9, que é referente à ação de *1. <CreateObjectAction> jogo JogoXadrez*, na Tabela 4.2 da seção anterior. O mesmo acontece entre as linhas 10 e 14, que se refere à ação *1.1* do mesmo caso de teste.

Código Fonte 4.1: Caso de teste com semântica de ações no formato XMI.

```
1  <?xml version=''1.0'' encoding=''UTF-8''?>
2  <ActionSemantics:TestSuiteAction xmi:version=''2.0'' xmlns:ActionSemantics=''
       ActionSemantics'' xmlns:BasicActions=''BasicActions'' name=''Chess_TC01''>
3    <testCase number=''01''>
4      <commands>
5        <action xsi:type=''BasicActions:CreateObjectAction'' name=''CreateObjectAction'' >
6          <owner xsi:type=''Kernel:Classifier'' name=''JogoXadrez'' />
7          <input name=''jogo'' qualifiedName=''jogo''/>
8          <result name=''jogo'' qualifiedName=''jogo''/>
9        </action>
10       <action xsi:type=''BasicActions:CallOperationAction'' name=''CallOperationAction''>
11         <owner xsi:type=''Kernel:Operation'' name=''iniciarNovoJogoXadrez'' qualifiedName
               =''iniciarNovoJogoXadrez" lower=''0'' upper=''10''/>
12         <owner xsi:type=''Kernel:Parameter'' name=''()'' qualifiedName=''()'' />
13         <target name=''jogo'' qualifiedName=''jogo''/>
14       </action> ...
15     <commands>
16   </testCase> ...
```

Esse caso de teste em formato XMI é utilizado como modelo de entrada durante a transformação MDA para a linguagem de comandos da ferramenta USE. Para isso, o inspetor deve configurar a transformação definindo: (1) as regras ATL a serem executadas, (2) os metamodelos de entrada e saída, (3) o modelo de entrada (XMI do caso de teste com semântica de ações) e (4) o local onde o modelo de saída será gerado (XMI de comandos USE). Essa configuração pode ser visualizada na Figura 4.3 de acordo com as numerações indicadas no texto. Após a execução da transformação, o XMI com os comandos da ferramenta USE são transformados, via MofScript, em um script na linguagem de USE.

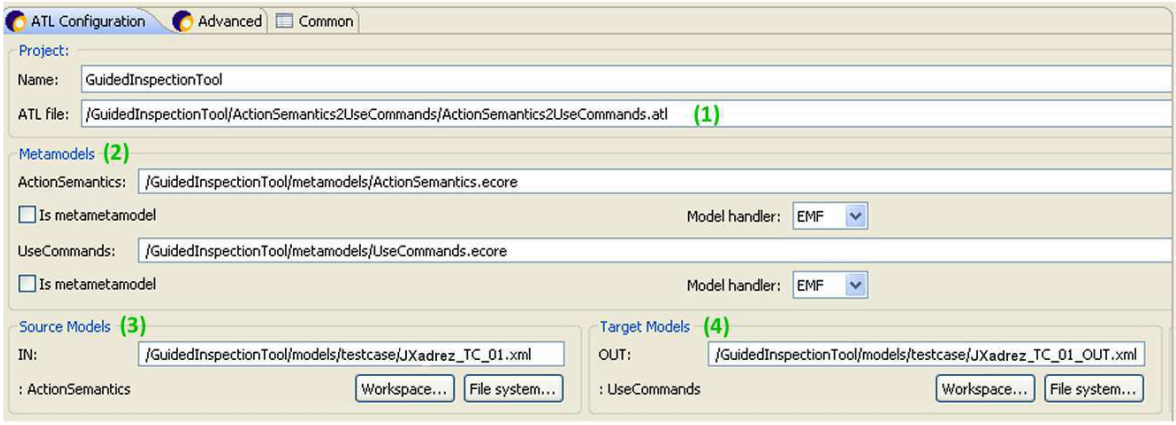

Figura 4.3: Configuração da inspeção guiada automática para gerar os comandos de USE.

Com a realização deste passo, o caso de teste que estava em XMI no formato de semântica de ações foi transformado em um script executável na ferramenta USE, que pode ser observado no Código Fonte 4.2.

Código Fonte 4.2: Caso de teste executável na linguagem da ferramenta USE.

```
1  --Caso de Teste: Chess-TC01
2  --Número: 01
3  !create jogo : JogoXadrez
4  !openter jogo iniciarNovoJogoXadrez()
5  !openter jogo solicitaNomeJogadores('Jonas','Maria')
6  !openter jogo verificaNomeValido('Jonas')
7  !openter jogo verificaNomeValido('Maria')
8  !openter jogo adicionaJogador('Jonas', 0)
9  !create jogBr : Jogador
10 !openter jogo adicionaJogador('Maria', 0)
11 !create jogPt : Jogador
12 !openter jogo adicionaPartida('01')
13 !openter jogo adicionaJogadoresPartida('Jonas', 'Maria', '01')
14 !openter jogo exibeTabuleiro()
15 !openter tabuleiro exibePecasIniciais()
```

4.4 Passo 3: Executar o caso de teste na ferramenta USE

Para executar o caso de teste na ferramenta USE é necessário cadastrar o diagrama de classes do sistema (Figura 4.2) nesta ferramenta, juntamente com todas as restrições OCL (invariantes, pré- e pós-condições), como foi exemplificado no Passo 3 do Capítulo 3. Estas restrições são importantes, pois elas são utilizadas para validar os casos de teste. Pode-se observar o diagrama de classes na ferramenta USE ilustrado na Figura 4.4.

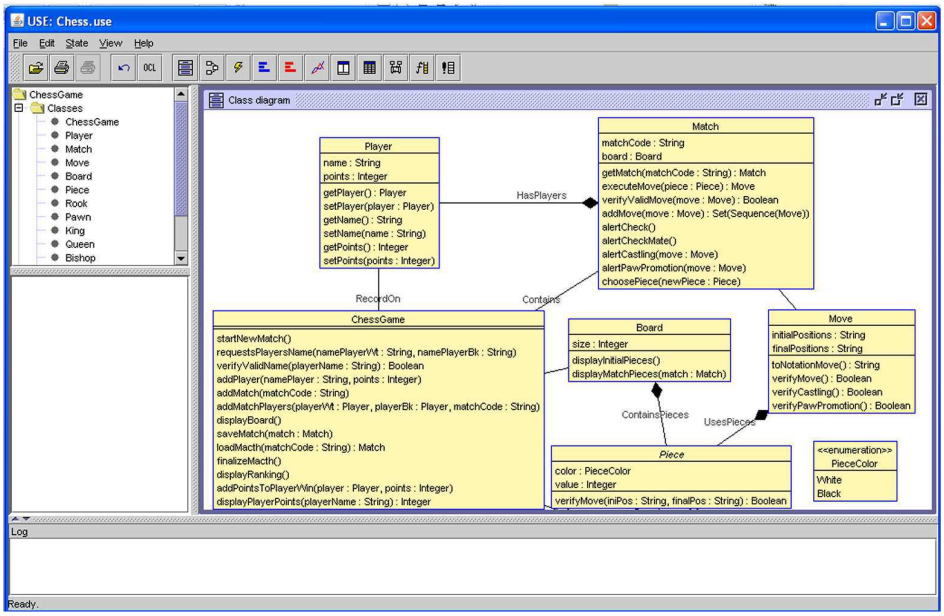

Figura 4.4: Diagrama de classes cadastrado na ferramenta USE.

A execução do caso de teste é acompanhada por uma tela no console. Então, após a execução de cada passo do caso de teste deve-se verificar se há alguma mensagem de erro. Os erros também podem ser visualizados no diagrama de seqüência gerado por USE, através das mensagens referenciadas pelas setas, como pode ser visualizado na Figura 4.5.

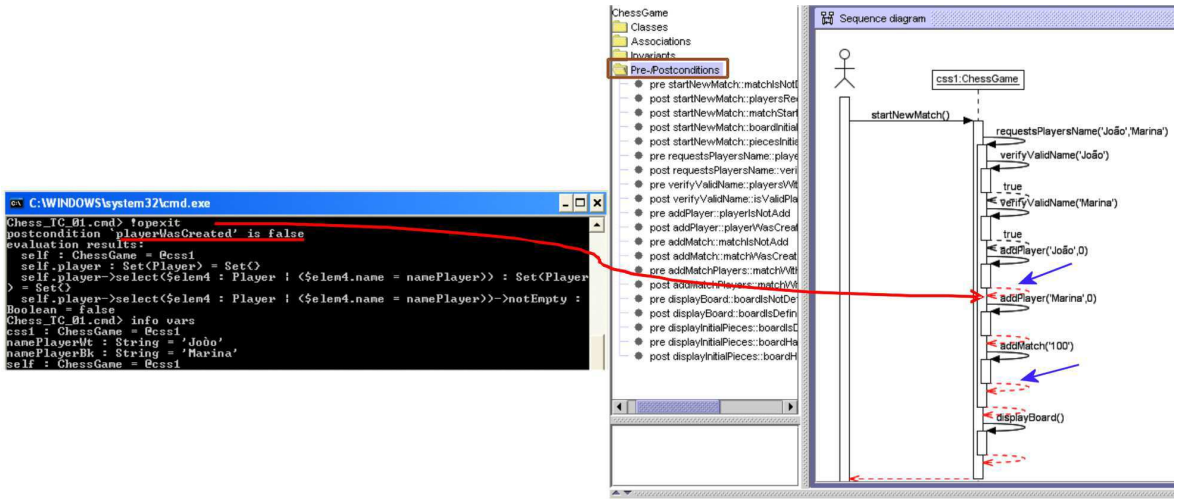

Figura 4.5: Console da ferramenta USE com alguns erros no caso de teste executado.

Se houver erros, então eles devem ser corrigidos. Em seguida, é necessário executar novamente o mesmo caso de teste até que ele esteja de acordo com todas as restrições OCL. Este procedimento deverá ser realizado para todos os casos de teste.

Os erros encontrados nos casos de teste podem refletir defeitos na especificação de caso de uso ou no diagrama de classes. Isso indica que estes artefatos não estão em conformidade entre si. Alguns defeitos encontrados na especificação de caso de uso são relacionados à dependência de criação entre objetos. No diagrama de classes, estas dependências podem ser visualizadas através da multiplicidade dos relacionamentos entre as classes. Caso a especificação de caso de uso seja corrigida, então os casos de teste também devem ser alterados.

Ao final deste passo um diagrama de seqüência para cada caso de teste é gerado. Além disso, este diagrama de seqüência estará em conformidade com a especificação de requisitos e o diagrama de classes.

4.5 Passo 4: Realizar a inspeção guiada automática

A inspeção guiada automática é realizada entre os diagramas de seqüência de caso de teste gerados pela ferramenta USE e os diagramas de seqüência de projeto fornecidos inicialmente no processo. Para isto, é necessário que ambos os diagramas estejam em formato XMI. Para a inspeção, o inspetor deve configurar a inspeção guiada com: (1) as regras ATL, (2) os metamodelos de entrada e saída, (3) os modelos de entrada (os dois diagramas de seqüência) e (4) definir o local onde o relatório de defeitos deverá ser gerado. Esta configuração pode ser visualizada na Figura 4.6 de acordo com as numerações do texto.

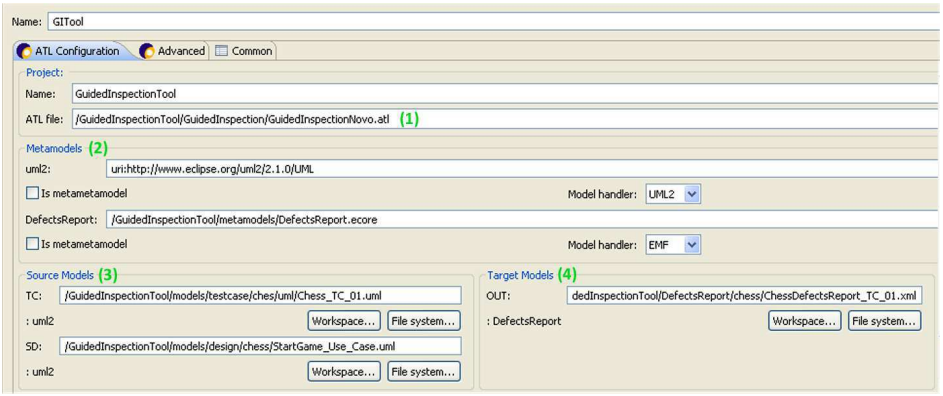

Figura 4.6: Exemplo da configuração da inspeção guiada automática.

A Figura 4.7 apresenta um exemplo de um diagrama de seqüência gerado automaticamente pela ferramenta USE. Esse diagrama é resultado da execução do caso de teste apresentado na Tabela 4.2 da seção 4.2 deste capítulo.

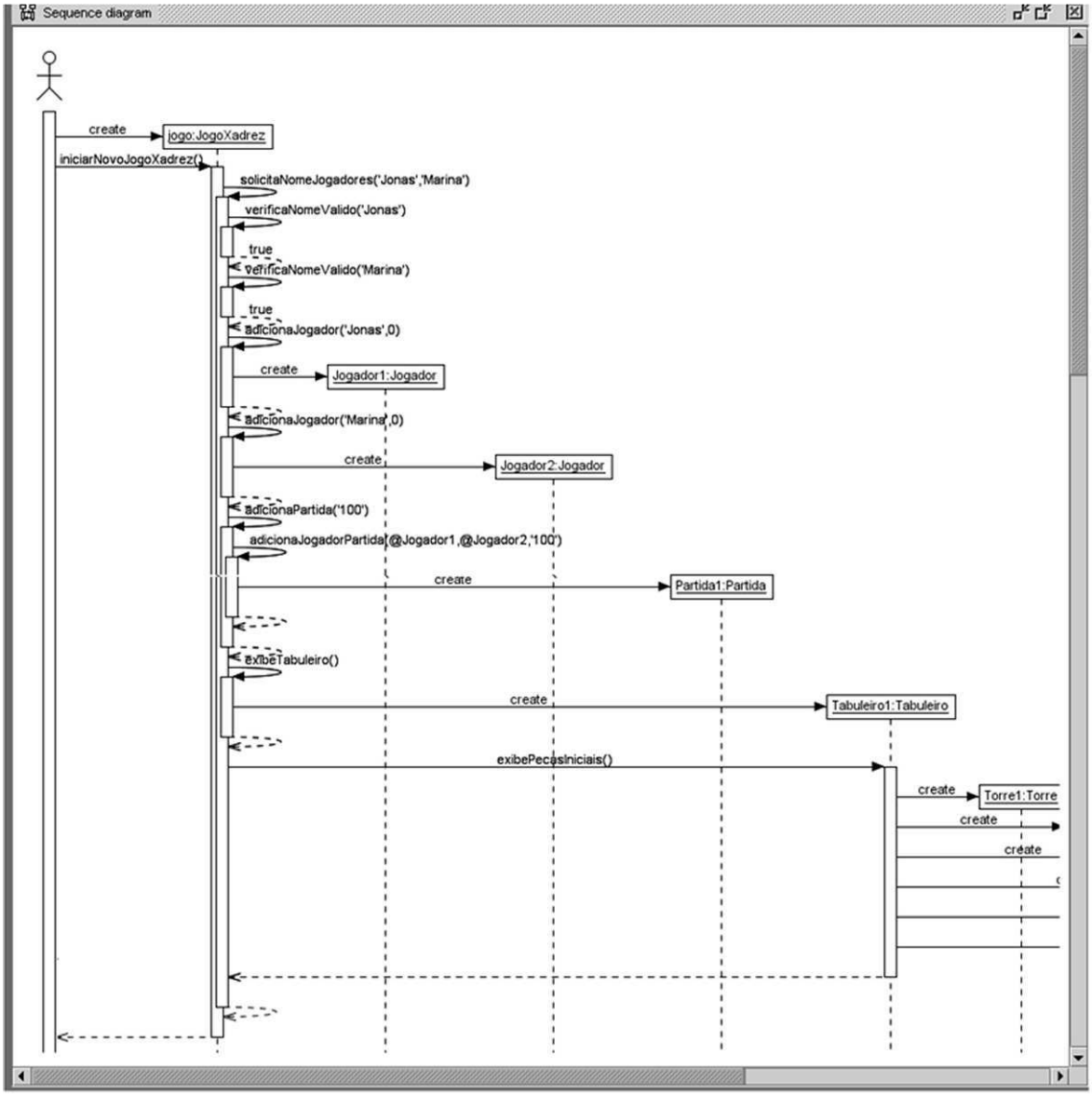

Figura 4.7: Diagrama de seqüência gerado pela ferramenta USE.

O diagrama de seqüência de projeto a ser inspecionado pode ser visualizado na Figura 4.8. Este diagrama foi criado a partir da especificação de caso de uso, independente da ferramenta USE.

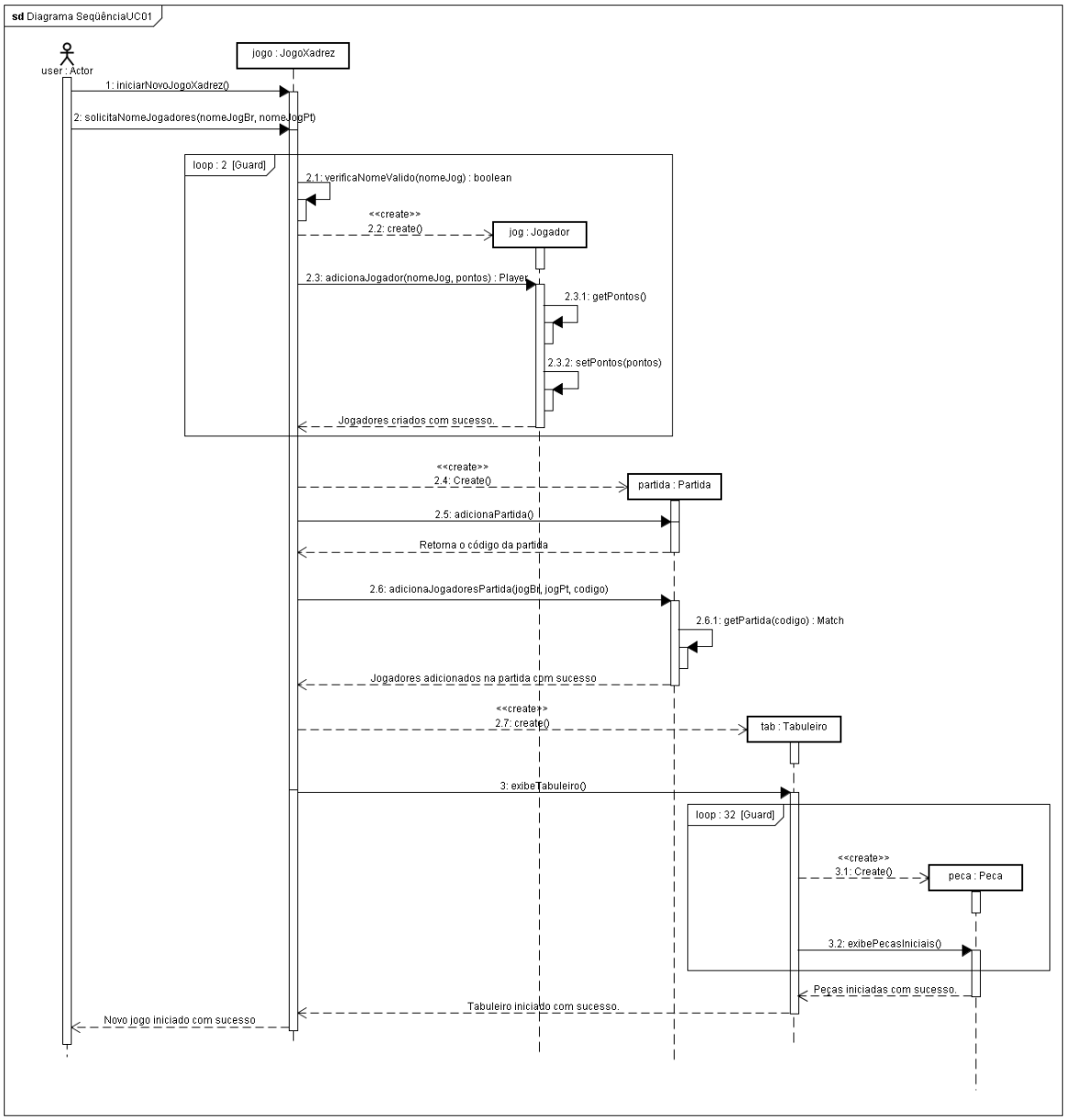

Figura 4.8: Diagrama de Seqüência para o caso de uso “Iniciar Jogo de Xadrez”.

A inspeção é realizada de forma automática, através das regras de inspeção descritas em ATL. Estas regras analisam a seqüência das mensagens presentes em cada diagrama para verificar se os dois diagramas (o de projeto e o gerado automaticamente através do USE) possuem o mesmo comportamento.

Neste exemplo, após a inspeção foram encontrados nove defeitos, sendo um alerta, que são apresentados na Tabela 4.3. Dentre os defeitos, cinco foram do tipo *MsgPositionError*, que indica a qual classe um determinado método deveria pertencer. Houve um defeito do

tipo *AbstractMsgError*, referente à criação de um objeto que pertence a uma classe abstrata e houve um defeito do tipo *MsgUnFoundError*, referente à falta de detalhe no diagrama de seqüência com relação à criação de cada instância de peça do tabuleiro. Um alerta foi do tipo *SequenceDiffAlert*, referente à regra de negócio, que só permite criar uma partida caso haja 2 jogadores para aquela partida, desta forma a ordem da mensagem deve ser alterada.

Número	Tipo	Descrição
1	*MsgPositionError*	O método "solicitaNomeJogadores(nomeJogBr, nomeJogPt)" deveria ser executado a partir da classe "JogoXadrez".
2	*MsgPositionError*	O método "adicionaJogador(nomeJogBr, pontos)" deveria ser executado a partir da classe "JogoXadrez".
3	*MsgPositionError*	O método "adicionaPartida(codPtd)" deveria ser executado a partir da classe "JogoXadrez".
4	*MsgPositionError*	O método "adicionaJogadoresPartida(nomeJogBr, nomeJogPt, codPtd)" deveria ser executado a partir da classe "JogoXadrez".
5	*SequenceDiffAlert*	Não deve ser possível criar uma partida sem antes ter adicionado os 2 jogadores no jogo.
6	*MsgPositionError*	O método "exibeTabuleiro()" deveria ser executado a partir da classe "JogoXadrez", pois não tem como exibir o tabuleiro antes de ele ser criado.
7	*MsgPositionError*	O método "exibePecasIniciais()" deveria ser executado a partir da classe "Tabuleiro", pois não tem como exibir as peças antes de elas terem sido criadas.
8	*AbstractMsgError*	As peças devem ser criadas por cada um de seu tipo concreto, pois não há como instanciar uma classe Abstrata, ex: "Peca".
9	*MsgUnFoundError*	Deveria haver o detalhe sobre a criação das peças. Ex.: 2 torres brancas, 2 torres pretas etc.

Tabela 4.3: Lista de defeitos encontrados.

4.6 Considerações Finais

Neste Capítulo, foi apresentado um exemplo de utilização da técnica de inspeção guiada automática, o qual foi ilustrado através de um sistema de jogo de xadrez. Basicamente, a inspeção guiada automática é composta por quatro passos, sendo que alguns dos passos requerem atividades manuais e outros já estão automatizados. O passo 1, por ser manual, requer mais tempo para realizá-lo e também é o mais difícil, pois necessita que o inspetor consiga identificar corretamente as semânticas de cada passo dos casos de teste. O passo 2 é muito simples, pois é automático. O passo 3 é bastante importante, pois ele faz a validação

dos casos de teste através das restrições OCL, garantindo que os diagramas de seqüência de caso de teste possuem qualidade suficiente para serem utilizados na inspeção. O passo 4 é bem simples de ser realizado, pois é automático. No próximo capítulo será apresentada a avaliação deste trabalho em comparação com outras técnicas de inspeção de software.

Capítulo 5

Trabalhos Relacionados

Este capítulo tem como objetivo relatar os principais trabalhos que focam em inspeção de software com especialidade na inspeção semântica entre a especificação de requisitos e os diagramas UML de projeto.

O estado da arte na área de inspeção de software é composto por técnicas manuais e automáticas com o propósito de inspecionar código, especificação de requisitos, modelos etc, sendo que estes artefatos representam o software em diferentes níveis de abstração. Outras técnicas focam apenas no processo de inspeção, ou seja, definem a metodologia que deve ser utilizada para garantir um processo de inspeção eficiente e integrado com o processo de desenvolvimento. Técnicas de inspeção são importantes porque melhoram a qualidade do software por analisar seus artefatos e detectar defeitos durante todo o processo de desenvolvimento, antes mesmo que o software seja produzido.

Em [Fag76] foi descrito o processo tradicional de inspeção de software, que é a base para outras técnicas de inspeção de software, tanto manuais quanto automáticas. Em 2000, [SJLY00] propôs uma reorganização para o processo tradicional de inspeção, visando adequar este processo com reuniões assíncronas, onde as equipes podem estar geograficamente distribuídas. Com esta reorganização foram incluídas mudanças para redução do custo e do tempo total para realização da inspeção. A principal mudança foi na realização da inspeção por vários inspetores em paralelo, aumentando a probabilidade de encontrar defeitos nos artefatos de software. Apesar de aumentar o custo com vários inspetores, há uma diminuição no tempo da inspeção e um aumento na qualidade do processo de inspeção. Isto traz mais benefícios e qualidade para o processo de desenvolvimento do software.

Uma visão geral de algumas técnicas de inspeção encontradas na literatura pode ser observada na Tabela 5.1. Estas técnicas são classificadas por suas (manual ou automática), em que artefato ela aplica a inspeção (análise, projeto ou código) e pelo tipo de inspeção realizada (sintática ou semântica). A coluna "Tipo" contém algumas células com "**", isto indica que a técnica é referente ao processo de inspeção e não à um técnica específica para inspecionar os artefatos, por isso não há nenhum dado.

Nosso trabalho propõe uma técnica de inspeção automática, que inspecione artefatos de análise e projeto de forma que sejam analisados os aspectos sintáticos e semânticos de cada artefato. De acordo com estas características, na tabela abaixo, as técnicas que mais se aproximam deste propósito são Inspeção Guiada e PBR.

Ano	**Autores**	**Técnica**	**Característica**	**Inspeção**	**Tipo**
1976	Fagan	Proc. Inspeção	Proc. Manual	Processo	**
2000	Sauer et al.	Proc. Inspeção	Proc. Manual	Processo	**
2004	Kalinowisk et al.	ISPIS	Proc. Automát.	Processo	**
1999	Travassos et al.	OORT	Insp. Manual	Análise/Proj.	Sintática
2000	Shull et al.	PBR	Insp. Manual	Análise/Proj.	Sint./Semânt.
2001	McGregor et al.	Inspeção Guiada	Insp. Manual	Análise/Proj.	Semântica
2004	Travassos et al.	PBR Tool	Proc. Automát.	Análise/Proj.	Sint./Semânt.

Tabela 5.1: Relação entre algumas técnicas de inspeção.

A ferramenta ISPIS é um *framework* que dá suporte a um processo de inspeção de software, permitindo o gerenciamento de cada etapa do processo [KT05]. Com este *framework* a inspeção pode ser realizada com equipes geograficamente distribuídas, pois ISPIS segue o novo processo de inspeção de software definido por [SJLY00]. A inspeção em si é realizada através da integração com ferramentas externas de inspeção, como *PBR Tool*. ISPIS tem como proposta aumentar a eficiência da inspeção de software, por utilizar uma metodologia experimental, levando em consideração o conhecimento a respeito de inspeções de software efetivamente avaliado. No entanto, em [KT05], não foi apresentado o nível de dificuldade para realizar a inspeção utilizando ISPIS, quais tipos de dados são manipulados ou como é feita a inspeção nos diferentes artefatos, pois a ferramenta ISPIS propõe apenas

automatizar o processo de inspeção de software.

A técnica OORT (*Object-Oriented Reading Technique*) tem o propósito de inspecionar manualmente diagramas UML de acordo com os diferentes níveis de abstração [TSCB99]. Quando a inspeção é realizada entre artefatos de um mesmo nível de abstração, denomina-se que foi realizada uma inspeção horizontal (por exemplo, comparar o diagrama de classes com o diagrama de seqüência analisando a consistência entre eles). No caso da inspeção ser realizada entre artefatos de diferentes níveis de abstração, denomina-se que foi realizada uma inspeção vertical (por exemplo, comparar o diagrama de classes com a especificação de requisitos verificando a completude dos requisitos). Desta forma, todos os artefatos são inspecionados entre si, seguindo as regras de inspeção para cada artefato. Com relação à inspeção entre a especificação de requisitos e os diagramas UML, onde está inserido este trabalho, a técnica OORT é incompleta, pois as regras de inspeção definidas para inspecionar estes artefatos detectam apenas defeitos sintáticos, ou seja, não verificam se a semântica de cada requisito foi representada em todos os diagramas UML.

As técnicas *Perspective-Base Reading* (PBR) [SRB00] e Inspeção Guiada [MS01] também realizam inspeção manual entre a especificação de requisitos e os diagramas UML e diferem da técnica OORT, porque conseguem detectar defeitos semânticos e sintáticos nos artefatos inspecionados. Nas próximas subseções, será apresentada uma descrição do funcionamento destas técnicas, além de suas vantagens e desvantagens.

5.1 *Perspective-Base Reading* (PBR)

Perspective-Base Reading (PBR) é uma técnica de inspeção de software para detectar defeitos em todos os artefatos de um processo de desenvolvimento. Os artefatos podem estar em linguagem natural, com isso a inspeção pode ser realizada desde as primeiras etapas do desenvolvimento. A técnica PBR tem a finalidade principal de ajudar os revisores a identificar o ponto certo do artefato a ser inspecionado e quais tipos de defeitos devem ser encontrados naquele artefato. Com isso, permite encontrar mais defeitos em menos tempo [CSM06].

Com PBR é possível determinar em quais perspectivas cada artefato deve ser inspecionado. As perspectivas podem ser diferenciadas pelos papéis dos *stakeholders* du-

rante o processo de desenvolvimento do software (por exemplo, o cliente, o analista, o desenvolvedor, o testador, o usuário, etc). Para cada perspectiva o inspetor é orientado com um conjunto de questões, que relacionam o artefato inspecionado com a visão de um determinado *stakeholder*, tornando os artefatos mais confiáveis.

Os tipos de defeitos que podem ser encontrados ao realizar a inspeção com a técnica PBR [SRB00] são:

- *Omissão:* qualquer requisito significativo, restrição de design, atributo, classe etc, que não tenha sido definido.
- *Fato incorreto:* um requisito que não pode acontecer sobre as condições especificadas para o sistema.
- *Ambiguidade:* múltiplas interpretações para uma mesma característica de um artefato.
- *Inconsistência:* dois ou mais requisitos que conflitam com outro.
- *Informação extra:* informação desnecessária ou que não está sendo utilizada, a qual pode confundir a leitura.

As principais vantagens desta técnica é que ela consegue realizar a inspeção utilizando o mesmo artefato de inspeção independente do projeto, variando apenas o tipo de inspeção de acordo com cada perspectiva. Com isso, o tempo para realizar a inspeção com PBR não é muito alto, já que o tempo de preparação é praticamente nulo.

As principais desvantagens da técnica PBR são com relação às questões que guiam o inspetor, pois às vezes elas são abstratas e gerais, pois se o inspetor não tiver muita experiência, ele pode descobrir menos defeitos do que os que realmente existem. Durante a fase de correção dos defeitos, não é fácil para o autor identificar qual parte do diagrama deverá ser corrigido, pois o relatório de defeitos de PBR às vezes é confuso.

Para resolver alguns destes problemas, em [ST04], foi desenvolvida uma ferramenta, chamada de *PBR Tool*, para gerenciar cada etapa da técnica PBR e cadastrar os defeitos em um relatório de defeitos. No entanto, esta ferramenta não realiza a inspeção automaticamente, pois a presença de um inspetor ainda é necessária para responder as questões definidas e cadastrar os defeitos encontrados. Com relação ao relatório de defeitos, como os defeitos

são descritos em uma lista descrita em linguagem natural, não há uma relação direta com os diagramas UML, o que pode ainda não facilitar na identificação e correção dos defeitos, pois depende da maneira como o inspetor descreve cada defeito.

5.2 Inspeção Guiada

O funcionamento da técnica de inspeção guiada manual foi descrito no Capítulo 2. A técnica de inspeção guiada tem a vantagem de realizar inspeção entre artefatos de diferentes níveis de abstração. A inspeção é realizada através da utilização de casos de teste, que compõe vários cenários de caso de uso. Como os casos de teste possuem uma seqüência de passos que representam o comportamento do sistema, então a inspeção guiada consegue identificar problemas semânticos nos artefatos de projeto inspecionados.

A principal desvantagem desta técnica é com relação à atividade criativa de relacionar os passos de cada caso de teste com o respectivo diagrama inspecionado e identificar se há alguma inconsistência. Outro ponto é com relação ao tempo para realizar a inspeção, pois como é necessário criar casos de teste para cada cenário de caso de uso. Vale ressaltar que ainda não existe nenhuma ferramenta para realizar a inspeção guiada de forma automática.

5.3 Considerações Finais

Este capítulo apresentou as principais técnicas de inspeção de software que se relacionam com o trabalho proposto. A maioria das ferramentas de inspeção encontradas na literatura focam apenas em automatizar o processo de inspeção de software. Além disso, as técnicas PBR e Inspeção Guiada, que realizam inspeção entre a especificação de requisitos e os diagramas de projeto foram descritas com suas vantagens e desvantagens. No próximo capítulo será apresentada uma avaliação experimental para avaliar a técnica proposta neste trabalho.

Capítulo 6

Avaliação Experimental

Este capítulo tem por objetivo apresentar um modelo de avaliação experimental para avaliar a técnica de inspeção guiada automática com relação às técnicas PBR e Inspeção Guiada Manual. Estas técnicas foram escolhidas porque realizam inspeção manual para verificar a conformidade semântica entre a especificação de requisitos e os diagramas de projeto do sistema.

A avaliação experimental foi elaborada com base na abordagem *Goal Question Metric* (GQM) [SB99], com isso o modelo de avaliação foi definido em termos de objetivos, questões e métricas. Por fim, foram realizados os estudos de caso e foi feita a coleta dos dados. Então, os dados foram analisados para se chegar aos resultados.

6.1 Avaliação Experimental para a Técnica de Inspeção Guiada Automática

O modelo de avaliação experimental utilizado para validar a técnica de inspeção guiada automática foi a abordagem GQM. O experimento foi executado com o objetivo de avaliar e comparar o comportamento da técnica de inspeção guiada automática com relação à técnica de inspeção guiada manual e à técnica *Perpective-Based Reading* (PBR). Estas técnicas realizam a inspeção em artefatos de software de forma a avaliar a conformidade entre os artefatos de projeto e a especificação de requisitos. A característica principal comum destas técnicas é que elas detectam defeitos semânticos em artefatos de software descritos em lin-

guagem natural e por isso foram escolhidas.

O mecanismo utilizado para realizar a avaliação foi por meio de estudos de casos, utilizados para aplicar cada uma das técnicas de inspeção avaliadas. Nas próximas subseções será apresentada a descrição das etapas para avaliar a técnica proposta.

6.1.1 Objetivos, Questões e Métricas

Foram definidos os objetivos, as questões e as métricas utilizados para identificar se o propósito deste trabalho foi alcançado. Cada objetivo visa avaliar a técnica de inspeção guiada automática com relação às técnicas de inspeção guiada manual e PBR. Os aspectos gerais avaliados foram: o tempo de inspeção, o número de defeitos encontrados, os tipos de defeitos, a complexidade dos defeitos e a eficiência de cada técnica. Para cada objetivo foram definidas questões, as quais são respondidas através das métricas capazes de identificar se os objetivos foram atingidos ou não.

1. Tempo de Inspeção

Verificar o tempo total para realizar a inspeção em cada estudo de caso. O tempo de realização da técnica PBR e inspeção guiada manual será composto pelo tempo de preparação, que corresponde ao tempo de preparar o ambiente para iniciar a inspeção, e pelo tempo de execução, que é o tempo para realizar a inspeção em todos os artefatos. No caso da técnica de inspeção guiada automática, o tempo total é composto por quatro partes, onde cada período de tempo corresponde ao tempo gasto em cada passo da técnica.

- *Questão 1.* Quanto tempo levou para realizar a inspeção sobre o experimento?

  Métrica 1.

  $$T = T_P + T_E \tag{6.1}$$

  sendo T o tempo total da realização da técnica de inspeção; T_P o tempo de organização dos artefatos e T_E o tempo de execução da inspeção.

Para o caso da técnica de inspeção guiada automática, o tempo de preparação é dividido em três etapas, as quais correspondem aos três passos para transformar os casos de teste descritos em linguagem natural para casos de teste executáveis. Estes passos levam um

tempo considerável para ser realizado, pois o primeiro passo, para anotar os casos de teste com semântica de ações, requer bastante experiência do inspetor.

2. Número de defeitos encontrados

Verificar o número total de defeitos encontrados nos modelos de projeto inspecionados. Os defeitos identificados pela inspeção que não forem defeitos reais, os falsos positivos, serão descartados. Para isso, após a inspeção o analista deverá analisar quais dos defeitos encontrados por cada uma das técnicas são realmente válidos.

- *Questão 2.* Quantos defeitos foram encontrados no diagrama de projeto inspecionado usando uma técnica de inspeção para o experimento?

  Métrica 2.

  $$ND = D_1 + D_2 + \ldots + D_i + \ldots + D_n \tag{6.2}$$

  sendo ND o número de defeitos válidos encontrados durante a inspeção e D_i um defeito válido encontrado, onde n é o número de defeitos encontrados.

3. Tipos de defeitos encontrados

Identificar os tipos de defeitos encontrados pelas técnicas de inspeção. Os tipos de defeitos encontrados pelas técnicas PBR e inspeção guiada manual são definidos como: Omissão, Fato incorreto, Ambigüidade, Inconsistência e Informação Extra. Para a técnica de inspeção guiada automática foram definidos tipos de defeitos equivalentes. No entanto, estes defeitos podem ser semanticamente comparados com os defeitos definidos para a técnica PBR. A Tabela 6.1 apresenta os tipos de defeitos encontrados pela inspeção guiada automática e suas respectivas equivalências com os defeitos de PBR.

- Questão 3. Quais os tipos de defeitos encontrados durante a inspeção?

  Métrica 3.

  $$D = (TD_1 : NTD_1; TD_2 : NTD_2; \ldots; TD_i : NTD_i; \ldots; TD_k : NTD_k) \tag{6.3}$$

  sendo D uma seqüência da quantidade de defeitos por tipo encontrada pela técnica de inspeção e TD_i o tipo de defeito encontrado; NTD_i é o número de defeitos encontrados daquele tipo, onde k é número de tipos de defeitos existentes.

Perspective-Based Reading	**Inspeção Guiada Automática**
Omissão	MsgUnFoundError
Fato incorreto	MsgSyntaxError, MsgPositionError
Ambiguidade	AssociationError, Ambigüidade
Inconsistência	AbstractMsgError, SequenceDiffAlert
Informação extra	MsgDiffAlert

Tabela 6.1: Equivalência entre os defeitos.

4. Grau de complexidade dos defeitos encontrados

Definir o grau de complexidade de cada defeito encontrado por cada técnica de inspeção. Neste caso, para cada tipo de defeito será determinado o seu grau de complexidade. Então, cada técnica será avaliada de acordo com a complexidade total dos defeitos válidos encontrados.

A complexidade dos tipos de defeitos, de acordo com [CSM06], pode ser visualizada na Tabela 6.2.

Inspeção Guiada Automática	**PBR**	**Complexidade**
MsgUnFoundError	Omissão	Alta
MsgSyntaxError, MsgPositionError	Fato incorreto	Alta
AssociationError, Ambiguidade	Ambiguidade	Média
AbstractMsgError, SequenceDiffAlert	Inconsistência	Baixa
MsgDiffAlert	Informação extra	Baixa

Tabela 6.2: Grau de complexidade de cada defeito.

5. Eficiência

Verificar a eficiência das técnicas de inspeção avaliadas. A eficiência foi medida através da razão entre a quantidade de defeitos encontrados e o tempo total para realizar a inspeção. Isto porque se uma técnica encontrar mais defeitos em um tempo menor, então ela será mais eficiente que outra técnica.

- Questão 5. Qual é a eficiência da técnica de inspeção?

  Métrica 5.

$$Efi = \frac{NDef}{T} \tag{6.4}$$

  sendo Efi a eficiência da técnica de inspeção; $NDef$ o número total de defeitos válidos encontrados; T é o tempo para realizar a inspeção.

6.1.2 Descrição dos Estudos de Caso

A avaliação foi realizada sobre estudos de casos com sistemas de diferentes tamanhos. O tamanho dos sistemas foi medido de acordo com o número de cenários de casos de uso na especificação de requisitos. Para realizar o estudo de caso o sistema deverá possuir os seguintes artefatos:

1. *A especificação de caso de uso:* para definir os cenários referentes a cada caso de uso do sistema. Esta especificação poderá ser descrita em linguagem natural.

2. *Casos de teste:* um conjunto de casos de teste para cada cenário dos casos de uso da especificação. Os casos de teste têm a finalidade de verificar se resultado obtido após a execução de uma seqüência de passos é equivalente ao resultado esperado descrito naquele caso de teste. Estes casos de teste também poderão ser descritos em linguagem natural.

3. *O diagrama de classes de projeto:* para definir a estrutura estática das classes do sistema.

4. *Diagramas de seqüência de projeto:* para definir o comportamento entre os objetos do sistema de forma que representem cada cenário dos casos de uso.

Além disso, a equipe deverá ser composta por participantes que possuam conhecimento de UML e inspeções de software. Estes participantes têm os papéis de analista, responsável por criar os modelos de projeto, e inspetores, que irão realizar a inspeção utilizando a técnica de PBR, a inspeção guiada manual e automática.

6.1.3 Coleta dos Dados

Para a coleta dos dados foram utilizados dois sistemas com tamanhos diferentes, possibilitando assim avaliar a eficiência de cada uma das técnicas de acordo com os diferentes aspectos. O tamanho dos sistemas foi medido pela quantidade de cenários de caso de uso, que podem ser considerados *pequenos* quando possuem de 1 a 10 cenários e *grandes* quando possuem mais de 11 cenários.

Foram realizados três estudos de caso de acordo com a Tabela 6.3. Para realizar uma avaliação completa, cada sistema foi submetido à inspeção utilizando cada uma das três técnicas citadas neste capítulo: a inspeção guiada automática, a inspeção guiada manual e PBR. A equipe que realizou os estudos de caso foi composta pelos analistas que criaram os artefatos a serem inspecionados e os inspetores com e sem experiência nas técnicas de inspeção.

Sistema	**Insp. Guiada Automática**	**I. Guiada Manual**	**PBR**
S1 - pequeno	Com experiência	Com experiência	Com experiência
S2 - grande	Com experiência	Com experiência	Com experiência
S2 - grande	Sem experiência	Sem experiência	Sem experiência

Tabela 6.3: Organização dos estudos de caso para a avaliação.

Durante a realização dos estudos de caso, os inspetores coletaram todas as informações necessárias para atender as métricas desta avaliação. Como por exemplo, o tempo para realização das técnicas, a quantidade de defeitos e os tipos de defeitos. Com estes dados foi possível inferir os resultados para avaliação de cada uma das técnicas de inspeção abordadas. Estes resultados poderão ser observados na próxima subseção.

6.1.4 Resultados

A seguir estão descritos os resultados da realização dos estudos de caso para avaliar a técnica de inspeção proposta com relação às outras técnicas manuais de inspeção. Além disso, será feita uma avaliação com relação ao tamanho do sistema e com relação à experiência dos inspetores nas respectivas técnicas.

Estudo de Caso 1: Sistema Quiz e Inspetores com Experiência

Este estudo de caso foi realizado sobre o sistema *Quiz*, que possui 7 cenários de casos de uso. A especificação deste sistema pode ser visualizada no Apêndice B.

As técnicas PBR, inspeção guiada automática (IGA) e inspeção guiada manual (IGM) foram realizada por inspetores com experiência nas técnicas de inspeção. Para este estudo de caso foram consideradas as métricas descritas neste capítulo.

1. Tempo de Inspeção

De acordo com a Tabela 6.4, o tempo para realizar a técnica IGA é composto por quatro etapas. Pode-se notar pelo resultado obtido em T_{AGI} que o tempo de preparação (T_1) da inspeção leva mais tempo que o tempo de execução.

A partir dos resultados para as métricas T_{GI} e T_{PBR}, pode-se notar que o tempo de execução é bem superior ao tempo de preparação. Pois, estas técnicas manuais requerem bastante atenção durante a execução da inspeção. Por outro lado, as técnicas automáticas requerem mais tempo para preparação, pois necessitam que os artefatos sejam adaptados para automação.

Questão 1. Quanto tempo levou para executar a técnica de inspeção?	
Técnica	**Métrica**
IGA	T_{AGI} = 115 + 3 + 2 + 15 = 125 min = 2,25 h
IGM	T_{GIM} = 10 + 63 = 73 min = 1,21 h
PBR	T_{PBR} = 20 + 68 = 88 min = 1,46 h

Tabela 6.4: Tempo total para realizar a inspeção sobre o sistema Quiz.

2. Número de defeitos encontrados

A Tabela 6.5 apresenta a quantidade de defeitos encontrados no sistema Quiz. Dentre os defeitos válidos encontrados durante a inspeção utilizando as técnicas de IGA, IGM e PBR, pode-se perceber pelas métricas acima que a IGA encontrou 60% de defeitos a mais que a técnica PBR. Já com a técnica IGM não foi possível identificar muitos defeitos.

Questão 2. Quantos defeitos foram encontrados nos diagramas de projeto inspecionados usando as técnicas de inspeção?	
Técnica	**Métrica**
IGA	$NDef_{AGI}$ = 16
IGM	$NDef_{GIM}$ = 5
PBR	$NDef_{PBR}$ = 10

Tabela 6.5: Quantidade de defeitos encontrados no sistema Quiz.

3. Tipos de defeitos encontrados

De acordo com a Tabela 6.6, pode-se perceber que a maioria dos defeitos encontrados são do tipo "Omissão" independente da técnica de inspeção. Além disso, a técnica IGM se mostra deficiente em detectar vários tipos de defeitos com relação às técnicas IGA e PBR.

Questão 3. Quais as variedades dos tipos de defeitos encontrados em cada técnica de inspeção?	
Técnica	**Métrica**
IGA	D_{AGI} = (Omissão : 10; Fato incorreto : 3; Ambigüidade : 1; Inconsistência : 0; Informação Extra : 2)
IGM	D_{GIM} = (Omissão : 4; Fato incorreto : 0; Ambigüidade : 0; Inconsistência : 1; Informação Extra : 0)
PBR	D_{PBR} = (Omissão : 5; Fato incorreto : 4; Ambigüidade : 0; Inconsistência : 0; Informação Extra : 1)

Tabela 6.6: Tipos de defeitos encontrados no sistema Quiz.

4. Grau de complexidade de todos os defeitos encontrados

A Figura 6.1 apresenta o valor absoluto e o percentual da complexidade dos defeitos encontrados pelas técnica IGA, IGM e PBR. Para este estudo de caso, as técnicas de inspeção guiada manual e automática apresentaram 80% dos defeitos com complexidade alta. No caso da técnica PBR, apresentou 90% de defeitos com complexidade alta que tem bastante relevância para o sistema.

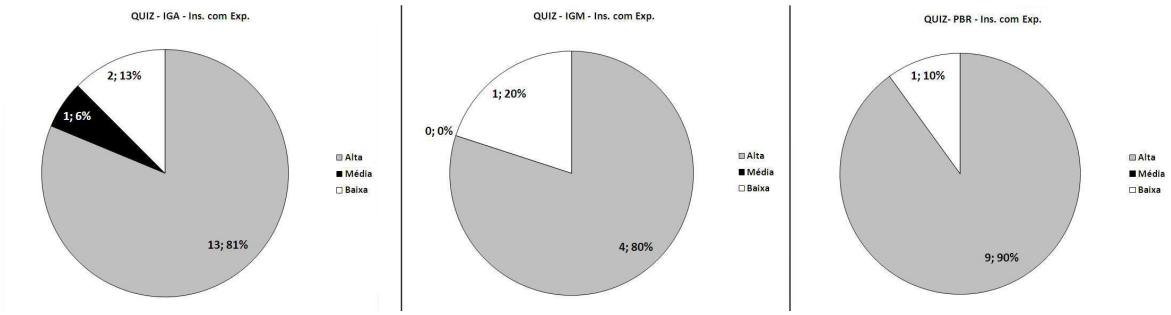

Figura 6.1: Grau de complexidade dos defeitos encontrados por cada técnica.

5. Eficiência

De acordo com a Tabela 6.7, as técnicas de IGA e PBR possuem uma eficiência equivalente para o sistema Quiz, apesar da técnica de IGA levar quase o dobro do tempo para realizar a inspeção, mas como IGA detectou mais defeitos que PBR, então a eficiência foi compensada. No entanto, a técnica IGM apresentou uma eficiência muito inferior às outras técnicas por detectar menos defeitos e possuir tempo equivalente à PBR.

Questão 5. Qual é a eficiência das técnicas de inspeção?	
Técnica	**Métrica**
IGA	$Efi_{AGI} = \frac{16}{2,25} = 7,11$
IGM	$Efi_{GIM} = \frac{5}{1,21} = 4,13$
PBR	$Efi_{PBR} = \frac{10}{1,46} = 6,85$

Tabela 6.7: Eficiência das técnicas de inspeção para o sistema Quiz.

Estudo de Caso 2: Sistema ATM e Inspetores com Experiência

Este estudo de caso foi realizado sobre o sistema *ATM*, que possui 18 cenários de casos de uso, ou seja, possui um tamanho maior que o sistema Quiz do estudo de caso anterior. A especificação deste sistema pode ser visualizada no Apêndice D.

Para este estudo de caso também foram utilizados inspetores com experiência nas técnicas PBR, inspeção guiada automática e manual. Para este estudo de caso também foram consideradas as mesmas métricas definidas para o primeiro estudo de caso.

1. Tempo de Inspeção

Apesar da experiência do inspetor na técnica de IGA, para realizar o passo 1 é necessário algum tempo para anotar os casos de teste com semântica de ações, pois esta é uma atividade criativa e não trivial, pois cada sistema é um novo desafio. Já as técnicas manuais de inspeção, como PBR, não requerem tanta criatividade, pois os artefatos utilizados na inspeção são os mesmos independentemente do sistema a ser inspecionado. Portanto, o tempo de preparação é bem mais rápido que o tempo de execução.

De acordo com a Tabela 6.8, se considerarmos apenas o tempo execução da inspeção(T_E), a técnica de IGA é realizada em um tempo mais de 10 vezes inferior às outras

técnicas manuais. Sendo que, a técnica IGA levou apenas 9 min para inspecionar os 18 cenários de caso de uso, ou seja, cerca de 30 segundos por cenário (considerando o tempo de abrir o sistema e executar), pois a execução de cada cenário leva cerca menos de 1 segundo para ser executado.

Questão 1. Quanto tempo levou para executar a técnica de inspeção?	
Técnica	**Métrica**
IGA	T_{AGI} = 300 + 9 + 9 + 9 = 327 min = 5,45 h
IGM	T_{GIM} = 20 + 93 = 133 min = 1,88 h
PBR	T_{PBR} = 16 + 100 = 1,93 h

Tabela 6.8: Tempo total para realizar a inspeção sobre o sistema ATM com inspetor experiente.

2. Número de defeitos encontrados

Com relação à métrica de número de defeitos válidos encontrados, pode-se notar na Tabela 6.9 que com a técnica de inspeção guiada automática foram encontrados cerca de 50% mais defeitos que as técnicas PBR e IGM apesar de ambos inspetores possuírem experiência nas técnicas.

Questão 2. Quantos defeitos foram encontrados nos diagramas de projeto inspecionados usando as técnicas de inspeção?	
Técnica	**Métrica**
IGA	$NDef_{AGI}$ = 35
IGM	$NDef_{GIM}$ = 17
PBR	$NDef_{PBR}$ = 14

Tabela 6.9: Quantidade de defeitos encontrados no sistema ATM com inspetor experiente.

3. Tipos de defeitos encontrados

De acordo com a Tabela 6.10, a variedade dos tipos de defeitos encontrados por IGA foi maior que a variedade dos tipos de defeitos encontrados pela técnica PBR. Há defeitos que não são facilmente visualizados durante a inspeção manual, como defeitos relativos à associação entre classes. Por exemplo, em um diagrama de classes saber se a multiplicidade e a direção dos relacionamentos estão em conformidade com a especificação. Os tipos de

defeitos de omissão e fato incorreto são os mais fáceis de serem identificados independentemente da técnica.

Questão 3. Quais as variedades dos tipos de defeitos encontrados em cada técnica de inspeção?	
Técnica	**Métrica**
IGA	D_{AGI} = (Omissão : 14; Fato incorreto : 14; Ambigüidade : 5; Inconsistência : 0; Informação Extra : 2)
IGM	D_{GIM} = (Omissão : 7; Fato incorreto : 5; Ambigüidade : 0; Inconsistência : 5; Informação Extra : 0)
PBR	D_{PBR} = (Omissão : 6; Fato incorreto : 7; Ambigüidade : 0; Inconsistência : 1; Informação Extra : 0)

Tabela 6.10: Tipos de defeitos encontrados no sistema ATM com inspetor experiente.

4. Grau de complexidade de todos os defeitos encontrados

A Figura 6.2 apresenta um gráfico da complexidade dos defeitos encontrados pelos inspetores utilizando as técnicas de inspeção. Para este estudo de caso, a técnica IGA apresentou 80% dos defeitos com complexidade alta e 14% com complexidade média, ou seja, os defeitos encontrados por esta técnica têm grande relevância para o sistema. A técnica IGM apresentou 70% dos defeitos com complexidade alta e 30% com complexidade baixa, no entanto não encontrou defeitos de complexidade média. No caso da técnica PBR, apresentou 93% de defeitos com complexidade alta e apenas 7% dos defeitos com complexidade baixa, ou seja, encontrou defeitos bastante relevantes.

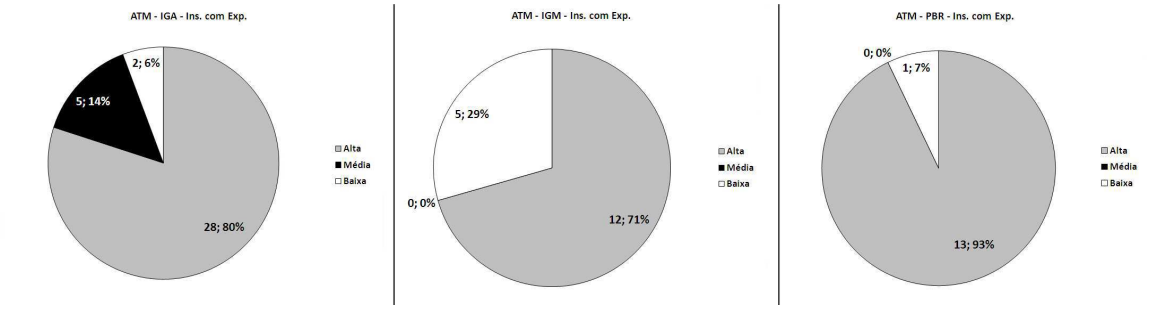

Figura 6.2: Grau de complexidade dos defeitos encontrados por cada técnica.

5. Eficiência

De acordo com a Tabela 6.11, a técnica PBR teve melhor resultado que IGA. Isto foi devido, principalmente, pelo tempo de realização da inspeção, onde com AGI foi gasto mais de 2 vezes o tempo para realizar a técnica PBR. Entretanto, como a técnica IGM obteve mais

defeitos que PBR e em menos tempo, pode-se dizer que esta técnica foi mais eficiente que as outras duas neste estudo de caso.

Questão 5. Qual é a eficiência das técnicas de inspeção?	
Técnica	**Métrica**
IGA	$Efi_{AGI} = \frac{(35)}{5,45} = 6,42$
IGM	$Efi_{GIM} = \frac{(17)}{1,88} = 9,04$
PBR	$Efi_{PBR} = \frac{(14)}{1,93} = 7,25$

Tabela 6.11: Eficiência das técnicas de inspeção para o sistema ATM com inspetor experiente.

Estudo de Caso 3: Sistema ATM e Inspetores sem Experiência

Este estudo de caso também foi realizado sobre o sistema *ATM*, que possui 18 cenários de casos de uso. A principal diferença deste estudo de caso com relação ao anterior é que desta vez os inspetores não possuem experiência nas técnicas apresentadas. Desta forma, temos o intuito de avaliar o desempenho de cada uma das técnicas quando não há tempo para realizar treinamento na equipe de inspeção.

1. Tempo de Inspeção

Como a técnica de inspeção guiada automática foi realizada por um inspetor sem experiência, então se pode perceber que o tempo para realizar o passo 1 (T_1) foi consideravelmente alto. Isto foi devido ao tempo para entender como a técnica de inspeção guiada automática funciona. Como os outros passos 2 ao 4 são completamente automáticos foi gasto em média um pouco mais de 1 minuto por cenário de caso de uso, pois este sistema possui 18 cenários a serem inspecionados.

De acordo com a Tabela 6.12, o tempo para realizar a técnica PBR foi relativamente baixo com relação ao número de cenários de caso de uso que havia. A principal vantagem desta técnica é o fato de não ser necessário muito tempo para preparar a inspeção, pois os artefatos da inspeção com PBR são genéricos para qualquer sistema. A técnica IGM também teve um tempo consideravelmente baixo, dado que o inspetor não tinha experiência.

Questão 1. Quanto tempo levou para executar a técnica de inspeção?	
Técnica	**Métrica**
IGA	T_{AGI} = 372 + 33 + 51 + 20 = 476 min = 7,93 h
IGM	T_{GIM} = 30 + 139 = 169 min = 2,82 h
PBR	T_{PBR} = 25 + 95 = 118 min = 1,97 h

Tabela 6.12: Tempo total para realizar a inspeção sobre o sistema ATM para inspetor sem experiência.

2. Número de defeitos encontrados

De acordo com a Tabela 6.13, pode-se notar que apesar da técnica de IGA ter sido realizada por um inspetor sem experiência, foram encontrados cerca de duas vezes mais defeitos que as técnicas PBR e IGA.

Questão 2. Quantos defeitos foram encontrados nos diagramas de projeto inspecionados usando as técnicas de inspeção?	
Técnica	**Métrica**
IGA	$NDef_{AGI}$ = 28
IGM	$NDef_{GIM}$ = 13
PBR	$NDef_{PBR}$ = 12

Tabela 6.13: Quantidade de defeitos encontrados no sistema ATM para inspetor sem experiência.

3. Tipos de defeitos encontrados

Segundo a Tabela 6.14, A variedade dos tipos de defeitos encontrados por PBR foi maior que a variedade dos tipos de defeitos da técnica de IGA. No entanto, esta técnica detectou mais defeitos com complexidade alta que PBR. Com relação á técnica IGA não foi possível identificar problemas de ambigüidade e informação extra.

Questão 3. Quais as variedades dos tipos de defeitos encontrados em cada técnica de inspeção?	
Técnica	**Métrica**
IGA	D_{AGI} = (Omissão : 19; Fato incorreto : 4; Ambigüidade : 4; Inconsistência : 0; Informação Extra : 1)
IGM	D_{GIM} = (Omissão : 5; Fato incorreto : 4; Ambigüidade : 0; Inconsistência : 4; Informação Extra : 0)
PBR	D_{PBR} = (Omissão : 4; Fato incorreto : 3; Ambigüidade : 1; Inconsistência : 3; Informação Extra : 1)

Tabela 6.14: Tipos de defeitos encontrados no sistema ATM para inspetor sem experiência.

4. Grau de complexidade de todos os defeitos encontrados

A Figura 6.3, apresenta um gráfico com o percentual da complexidade dos defeitos encontrados utilizando as técnicas de inspeção. Apesar da falta de experiência do inspetor, com a técnica IGA foi possível encontrar uma quantidade maior de defeitos relevantes para o sistema, pois com esta técnica foi possível encontrar 28 defeitos, sendo que 82% destes defeitos possuem complexidade alta e 14% complexidade média. No entanto, as técnicas manuais IGM e PBR tiveram comportamentos parecidos entre si, pois em ambas foram encontrados em média 65% dos defeitos com complexidade alta e cerca de 30% com complexidade baixa.

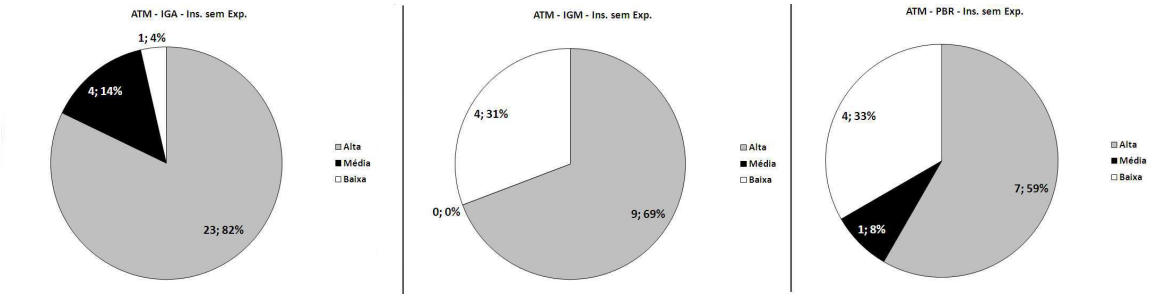

Figura 6.3: Grau de complexidade dos defeitos encontrados por cada técnica.

5. Eficiência

De acordo com a Tabela 6.15, a técnica PBR foi mais eficiente que IGA e GIM. Isto foi devido, principalmente, pelo tempo de realização das técnicas quando o inspetor não tem experiência em usar as técnicas. Em PBR foi gasto quatro vezes menos tempo que a técnica de IGA. A técnica de IGM também teve um bom tempo, mas com relação à quantidade de defeitos foi bem parecida com PBR.

Questão 5. Qual é a eficiência das técnicas de inspeção?	
Técnica	**Métrica**
IGA	$Efi_{AGI} = \frac{(28)}{7,93} = 3,53$
IGM	$Efi_{GIM} = \frac{(13)}{2,82} = 4,60$
PBR	$Efi_{PBR} = \frac{(12)}{1,97} = 6,09$

Tabela 6.15: Eficiência das técnicas de inspeção para o sistema ATM para inspetor sem experiência.

6.1.5 Análise dos Resultados

As técnicas de inspeção PBR, inspeção guiada automática e inspeção guiada manual foram analisadas com relação a duas perspectivas: o tamanho do sistema e a experiência dos inspetores. Isso foi possível porque a diferença entre estudo de caso 1 e o estudo de caso 2 é o tamanho do sistema, pois os inspetores possuíam experiência nas técnicas. Desta forma, foi possível analisar o comportamento de cada uma das técnicas de acordo com cada sistema.

Com relação à experiência dos inspetores, para os estudos de caso 2 e 3 utilizamos o mesmo sistema, mas variamos a experiência dos inspetores, ou seja, os estudos de caso 2 e 3 inspetores com experiência e sem experiência, respectivamente.

Tamanho do Sistema *versus* Técnicas de Inspeção

Analisar o desempenho das técnicas de inspeção à medida que os sistemas crescem é uma avaliação importante, pois tempo está diretamente ligado a custo. Com esta análise será possível verificar a relação entre custo e benefício para cada técnica, isto porque o tempo está relacionado com o custo e a quantidade de defeitos encontrados por cada técnica está relacionada aos benefícios que cada técnica pode trazer durante o desenvolvimento de um sistema.

A Figura 6.4 apresenta um gráfico com os resultados obtidos para o estudo de caso 1, que utilizou como base o sistema Quiz com apenas 7 cenários de caso de uso (representado pelas barras pretas) e os resultados obtidos no estudo de caso 2, que utilizou o sistema ATM com 18 cenários de caso de uso (representado pelas barras cinzas). Ambos os estudos de caso foram realizados por inspetores com experiência, por isso a experiência não foi levada em consideração nesta análise.

De acordo com esse gráfico pode-se inferir que independente do tamanho do sistema o tempo para realizar a técnica IGA foi superior às outras técnicas manuais (IGM e PBR). Estas apresentaram tempos bem semelhantes para os dois sistemas. Como já foi dito anteriormente, a técnica IGA demanda bastante tempo para realizar o passo 1 de preparação, por ser um passo manual. Além disso, pode-se perceber que o tempo para realizar a técnica IGA foi proporcional ao tamanho do sistema, pois teve um tempo bem superior para o sistema ATM com relação ao sistema Quiz.

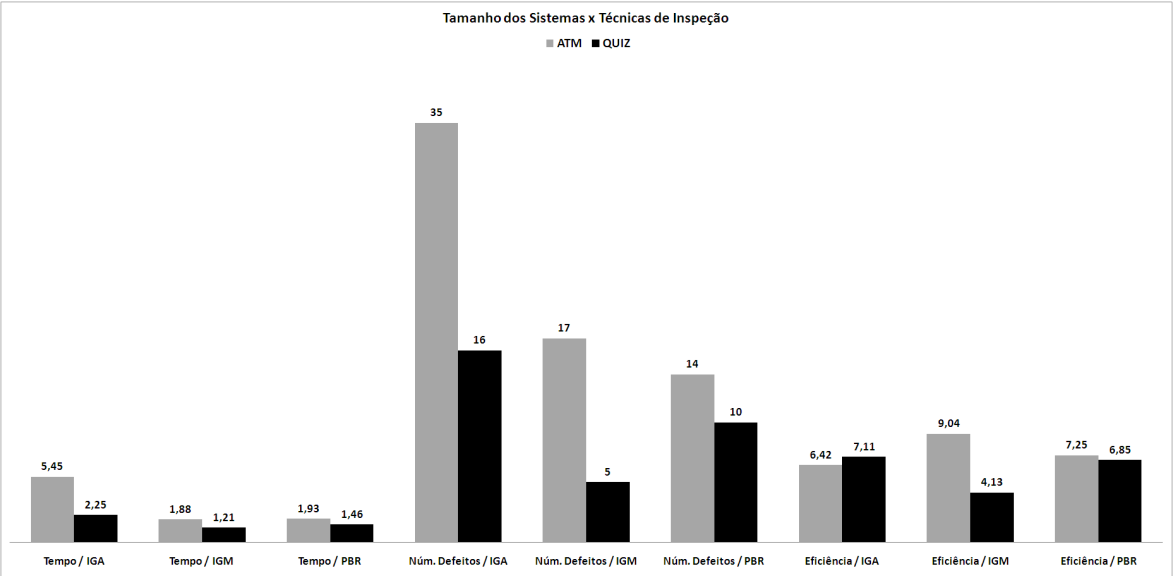

Figura 6.4: Gráfico com o resultado da avaliação das técnicas de inspeção para os sistema Quiz e ATM.

Sobre a quantidade de defeitos encontrados por cada técnica, pode-se notar que para os dois sistemas a técnica IGA encontrou mais que o dobro de defeitos das outras técnicas, independente do sistema. A eficiência das técnicas é inversamente proporcional ao tempo, logo a técnica IGA teve uma inferior às duas outras técnicas para o sistema maior (ATM). No entanto, pode-se notar que a eficiência da técnica PBR é semelhante à eficiência de IGA, no entanto por motivos distintos, pois PBR é beneficiada pelo tempo e IGA pela quantidade de defeitos. Portanto, se em um projeto o tempo tiver maior relevância do que a qualidade do sistema, PBR seria mais indicado.

Com relação aos defeitos encontrados no sistema Quiz, dentre os 10 defeitos encontrados utilizando a técnica PBR 2 defeitos não foram encontrados por IGA, estes defeitos são referentes a parte externa do *REF* do diagrama de seqüência de projeto. No entanto, a IGA conseguiu detectar 4 defeitos no diagrama de classes, referentes à ausência de métodos. Já com PBR estes defeitos não foram encontrados, pois as questões utilizadas para inspecionar são genéricas para qualquer projeto. Com relação a IGM, todos os 5 defeitos foram encontrados por IGA.

Experiência dos Inspetores *versus* Técnicas de Inspeção

A experiência do inspetor pode influenciar diretamente nos resultados de uma técnica de inspeção. Com este intuito foi realizado o estudo de caso 3 que utilizou o mesmo sistema do estudo de caso 2 (ATM) por possuir vários cenários de caso de uso. Para esta avaliação, podemos observar a Figura 6.5 que apresenta um resumo de todos os resultados obtidos por cada uma das técnicas de inspeção, quando realizadas por inspetores com e sem experiência. De acordo com este gráfico em resumo, pode-se observar que a experiência foi um fator relevante para encontrar mais defeitos e gastar menos tempo para realizar cada uma das técnicas de inspeção.

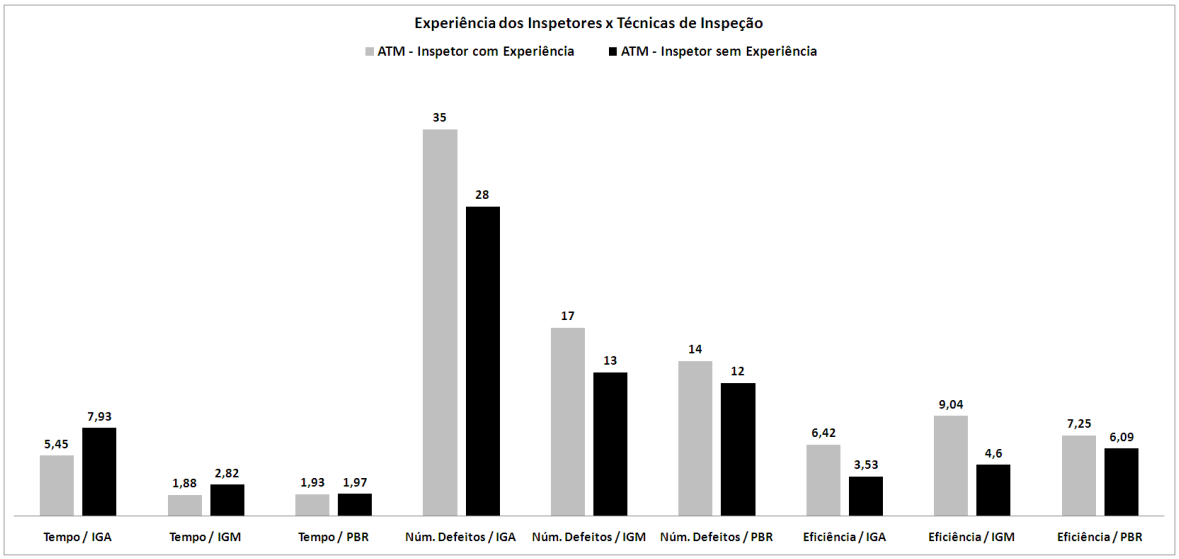

Figura 6.5: Gráfico com o resultado da avaliação das técnicas de inspeção para inspetores com e sem experiência.

Além disso, se compararmos a relação entre as técnicas e cada métrica, a técnica PBR apresentou praticamente nenhuma diferença com relação ao tempo e a quantidade defeitos encontrados pelos inspetores com e sem experiência. No caso da técnica IGA, pôde-se perceber que o inspetor com mais experiência na técnica conseguiu formalizar os casos de teste para execução com mais precisão do que o inspetor sem experiência, pois aquele conseguiu identificar mais defeitos que o outro. Além disso, realizou IGA em menos tempo que o inspetor sem experiência. A técnica IGM foi proporcionalmente semelhante à técnica IGA, no entanto com IGA foi possível identificar muito mais defeitos que com a técnica IGM. A técnica PBR permitiu identificar poucos defeitos com relação às outras duas técnicas. Mas,

como o tempo para realizar PBR foi inferior às outras técnicas, independente da experiência do inspetor, essa apresentou ser mais eficiente que a técnica IGA.

O grau de complexidade dos defeitos encontrados por cada técnica também é uma métrica importante para avaliar as técnicas de inspeção. Pois, quando um defeito tem maior relevância para o sistema ao ser corrigido, ele é considerado mais complexo que outro. A complexidade está relacionada aos tipos de defeitos encontrados, com base nisto, pode-se perceber na Figura 6.6 que independente da experiência do inspetor, com a técnica IGA é possível identificar defeitos praticamente os mesmos defeitos. Isto não acontece na técnica PBR, pois as questões que são utilizadas como base para a inspeção são gerais para qualquer sistema, dependendo bastante da experiência do inspetor em detectar ou não um determinado defeito. No caso da técnica de inspeção guiada manual, ela utiliza casos de teste que representam todo o comportamento esperado do sistema, estes casos de teste guiam o inspetor durante a inspeção, logo a experiência do inspetor não tão influente e os tipos de defeitos encontrados são bem semelhantes.

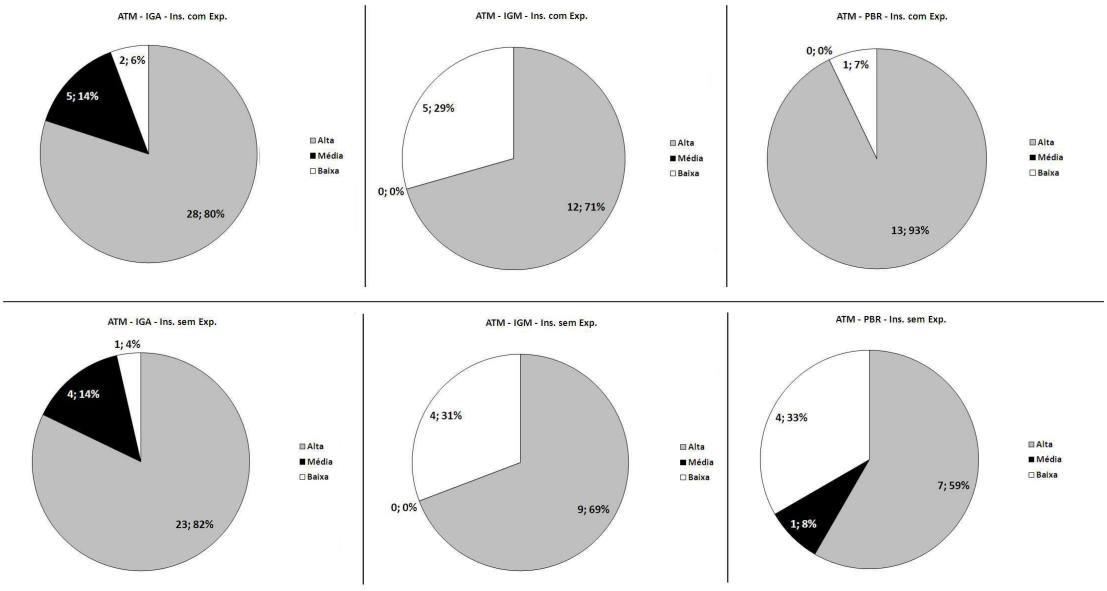

Figura 6.6: Gráfico com o grau de complexidade dos defeitos encontrados por inspetores com e sem experiência.

Se fizermos uma comparação com relação à quantidade de defeitos encontrados, pode-se notar que as técnicas de inspeção guiada, manual e automática, tiveram melhores resultados do que a técnica PBR. Isso é bastante relevante para IGA, pois se pode notar que mesmo

quando IGA é utilizada pela primeira vez por um inspetor, a técnica permite detectar mais defeitos que as técnicas IGM e PBR.

Dentre os 12 defeitos encontrados por PBR, apenas 3 defeitos não foram encontrados usando a técnica IGA, sendo um defeito referente à presença dos nomes das linhas de vida no diagrama de seqüência, outro defeito referente à inconsistência na própria especificação e outro defeito referente a uma mensagem a mais no diagrama de seqüência de projeto.

Além dessa análise, pode-se perceber também que uma técnica automática para inspeção de artefatos de software pode ser bem mais eficiente e talvez possuir melhor desempenho se considerarmos futuras modificações nos artefatos inspecionados. Pois, na técnica IGA foi gasto mais tempo para formalizar a especificação de requisitos em casos de teste executáveis do que para realizar a inspeção. Uma vez que a especificação já está formalizada, então quando houver modificações nos modelos de projeto, a inspeção guiada automática poderá ser realizada em um tempo muito pequeno. Enquanto, as técnicas manuais, tanto PBR quanto a IGM, quando há uma mudança nos artefatos, podem necessitar de tempos equivalentes para realizar a inspeção, pois seria preciso inspecionar todos os artefatos novamente.

6.1.6 Considerações Finais

Este capítulo apresentou a avaliação experimental da técnica IGA com relação às técnicas IGM e PBR. Fazendo uma análise geral com relação aos resultados obtidos pelos três estudos de caso pode-se notar que, independente da experiência do inspetor a técnica IGA permitiu encontrar uma quantidade de defeitos superior às outras duas técnicas. Pois, o segundo e o terceiro estudo de caso foram realizados por inspetores com experiência em IGA e outros sem experiência, respectivamente. Além disso, grande parte dos defeitos encontrados pelas técnicas PBR e IGM também foram encontrados através da IGA.

É possível perceber também que, no estudo de caso 3, o tempo para IGA apesar de ser bastante elevado para um inspetor sem experiência, mas houve uma redução quando foi utilizada por um inspetor com experiência. Mas, independente disso IGA precisa ser aprimorada, já que no estudo de caso 3, ela também teve um tempo pior que as outras técnicas. Logo, a automação da técnica de inspeção guiada é bastante relevante por encontrar defeitos nos artefatos de projeto de software, apesar do passo 1, referente à anotação de semântica de ações nos casos de teste para torná-los executáveis, necessitar ser otimizado.

Capítulo 7

Conclusão

O trabalho apresentado neste documento propõe uma maneira de automatizar a técnica de inspeção guiada utilizando técnicas de transformações MDA. A automação possui quatro passos que foram apresentados no Capítulo 3. O passo 1 é realizado manualmente, por isso é o que demanda mais tempo, o passo 2 é realizado automaticamente, o passo 3 é essencial para a inspeção guiada possa ser automatizada, pois facilmente a inspeção é realizada quando se tem artefatos de um mesmo nível de abstração (ambos diagramas de seqüência). Por fim, o passo 4 é realizado automaticamente e é referente à inspeção guiada propriamente dita.

Ao realizar a inspeção guiada automática, nesse passo 4, os fragmentos combinados dos diagramas de seqüência de projeto não são considerados. Pois, os diagramas de seqüência de caso de teste gerados por USE não possuem tais fragmentos. No entanto, caso um diagrama de projeto possua tais fragmentos ele poderá ser inspecionado normalmente, a diferença é que durante a inspeção não será considerada a lógica dos fragmentos. Mas, isso não influencia no resultado da inspeção, pois como há um caso de teste para cada cenário de caso de uso, que são equivalentes aos cenários formados pelos fragmentos combinados. Então, em algum momento da inspeção o que estiver dentro dos fragmentos será inspecionado corretamente.

A técnica de inspeção manual tem o propósito de realizar inspeção entre a especificação de requisitos e os diagramas UML de projeto, a fim de verificar a conformidade entre estes artefatos. O diferencial da técnica de inspeção guiada (manual ou automática) com relação às outras técnicas de inspeção é que ela utiliza casos de teste para identificar inconsistências semânticas entre os artefatos de software. A maioria das técnicas de inspeção realizam a inspeção nos artefatos individualmente e detectam defeitos sintáticos. O diferencial da

técnica de inspeção guiada automática com relação à inspeção guiada manual é que aquela consegue detectar defeitos tanto sintáticos quanto semânticos, enquanto esta, principalmente semânticos.

Apesar de a inspeção guiada automática ser realizada sobre os diagramas de seqüência de projeto, o diagrama de classes também passa por um processo de inspeção, pois ele é utilizado durante a anotação da semântica de ações nos casos de teste. Com isso, se houver alguma inconsistência à medida que os casos de teste são anotados com semântica de ações. Por exemplo, se estiver faltando algum método no diagrama de classes para realizar uma determinada ação do caso de teste, então isto será considerado como um defeito. Há casos em que são encontradas algumas inconsistências nos casos de teste e como eles refletem diretamente a especificação de requisitos, então são reportados defeitos para a especificação. Desta forma, a inspeção guiada automática se torna uma técnica completa, pois abrange várias formas de se detectar inconsistências nos artefatos de software.

Para avaliar a técnica proposta foram realizados três estudos de caso com características distintas, então foram definidas várias métricas para comparar o comportamento das técnicas de inspeção PBR, IGA e IGM.

O estudo de caso 1 foi realizado sobre um sistema com apenas 7 cenários de caso de uso, sendo que as técnicas de IGA, IGM e PBR foram realizadas por inspetores com experiência em cada uma dessas técnica. Para este estudo de caso, os resultados foram bem satisfatórios para a técnica IGA, pois com ela foi possível detectar 16 defeitos válidos, enquanto PBR teve 10 defeitos e IGM apenas 5 defeitos. Além disso, o grau de complexidade dos defeitos encontrados também foram fatores positivos para a IGA. No entanto, PBR teve melhor resultado com relação a eficiência, pois a inspeção foi realizada em um tempo bem inferior à técnica IGA.

Com relação ao estudo de caso 2 e 3, eles foram realizado sobre um sistema com 18 cenários de caso de uso. A principal diferença entre estes dois estudos de caso é que o segundo foi realizado por inspetores com experiência e o terceiro por inspetores sem experiência. A falta de experiência para realizar a IGA foi bem perceptível no tempo de preparação, que foi 4 vezes maior que o tempo para realizar PBR considerando outro inspetor sem experiência. No entanto, mesmo um inspetor com pouca experiência na técnica, com a IGA foi possível detectar 16 defeitos a mais que o outro usando PBR e 11 defeitos a mais

que IGM. Como o tempo para realizar a IGA foi bem elevado, então a eficiência foi mais satisfatória para a técnica de IGM, quando o inspetor tem experiência e para PBR para o inspetor sem experiência.

Apesar de nos três estudos de caso o tempo total para realizar IGA ter sido superior às outras técnicas, se compararmos apenas o tempo de execução da inspeção, a IGA tem um tempo inferior as outras duas técnicas. Este tempo poderá ser menor no caso de haver mudanças nos diagramas de projeto a serem inspecionados. Neste caso só será necessário gastar tempo com a execução da IGA, o que é irrelevante. Enquanto, as técnicas manuais, como PBR e IGM, para este mesmo caso será necessário realizar toda a inspeção novamente, o que levará um tempo equivalente ao tempo da execução inicial.

7.1 Limitações

Com a realização dos estudos de caso pôde-se perceber que a técnica IGA desenvolvida neste trabalho possui algumas limitações, como:

- Tempo gasto para realizar a anotação da semântica de ações nos casos de teste muito elevado com relação aos outros passos.
- A técnica depende da ferramenta USE para realizar a automação.
- Faltou realizar mais estudos de caso de forma a analisar a escalabilidade da técnica IGA para sistemas ainda maiores e mais complexos.
- A técnica poderia compor uma ferramenta a fim de haver um ambiente mais adequado para realizar a inspeção, pois no momento ela está bem fragmentada.
- Faltou realizar estudos de caso para verificar a eficiência da técnica IGA com relação à IGM e PBR quando os artefatos de projeto sofrem alterações.
- A técnica IGA não está considerando os fragmentos combinados dos diagramas de seqüência de projeto.

7.2 Trabalhos Futuros

Com a conclusão deste trabalho, tivemos a possibilidade de fazer uma análise do mesmo, a qual resultou no seguinte conjunto de propostas para sua continuidade:

- **Otimizar a anotação dos casos de teste com semântica de ações:** o passo 1 é o que leva mais tempo para ser realizado. Uma forma de melhorar o tempo seria criar uma interface, onde o inspetor pudesse selecionar a semântica de ações equivalente para cada passo do caso de teste.

- **Transformar o XMI de semântica de ações em um diagrama de seqüência:** se o XMI de semântica de ações fosse transformado automaticamente em um diagrama de seqüência de caso de teste, não seria necessário utilizar a ferramenta USE para gerar o diagrama de seqüência.

- **Submeter o trabalho a mais estudos de caso:** foram desenvolvidos três estudos de caso, no entanto poderia haver um experimento que simulasse alterações nos artefatos para verificar o comportamento das técnicas.

- **Aprimorar a técnica de automação da inspeção guiada:** apesar de termos uma versão que realiza a inspeção guiada que encontra defeitos relevantes, esta técnica poderia se transformar uma ferramenta integrada com um ambiente de modelagem de sistemas.

- **Verificar o comportamento da técnica de automação da inspeção guiada à medida que o sistema evolui:** analisar o comportamento da técnica de inspeção guiada automática em comparação com outras técnicas de inspeção manual à medida que o sistema inspecionado evolui, ou seja, em que haja atualizações nos artefatos a serem inspecionados.

- **Realizar a inspeção considerando os fragmentos combinados:** aprimorar as regras ATL que fazem a inspeção entre os diagramas de seqüência de caso de teste e o diagrama de seqüência de projeto a fim de levar em consideração os fragmentos combinados deste diagrama de projeto.

Bibliografia

[BDG+07] P. Baker, Z. R. Dai, J. Grabowski, Ø. Haugen, I. Schieferdecker, and C. Williams. *Model-Driven Testing: Using the UML Testing Profile*. Springer, 1 edition, 2007.

[Cla99] James Clark. Xsl transformations (xslt). http://www.w3.org/TR/xslt, 1999.

[CSM06] P. Costa, F. Shull, and W. Melo. Getting requirements right: The perspective-based reading technique and the rational unified process. *IBM*, 2006.

[Fag76] M.E. Fagan. Design and code inspections to reduce errors in program development. In *IBM Systems Journal*, pages 182–211, 1976.

[FS04] F. Fondement and R. Silaghi. Defining model driven engineering processes. In *Proceeding of the 3rd Workshop in Software Model Engineering*, Lisbon, Portugal, 2004.

[GBR07] M. Gogolla, F. Büttner, and M. Richters. Use: A uml-based specification environment for validating uml and ocl. *Sci. of Comp. Programming*, 69(1-3):27–34, 2007.

[Gro05] Object Management Group. Ocl 2.0 specification. Technical Report ptc/2005-06-06, OMG, 2005. http://www.omg.org/docs/ptc/05-06-06.pdf.

[Gro07a] Object Management Group. Meta object facility (mof) 2.0 query/view/transformation specification. Technical Report Final Adopted Specification ptc/07-07-07, OMG, 2007. http://www.omg.org/cgi-bin/doc?ptc/2007-07-07.

[Gro07b] Object Management Group. Uml superstructure, v2.1.1. Technical Report formal/07-02-05, OMG, 2007. http://www.omg.org/cgi-bin/doc?formal/07-02-05.

[Gro07c] Object Management Group. Xml metadata interchange (xmi), v2.1.1. Technical Report formal/2007-12-01, OMG, 2007. http://www.omg.org/docs/formal/07-12-01.pdf.

[HJGP99] W. M. Ho, J. M. Jquel, A. L. Guennec, and F. Pennaneac'h. Umlaut: An extendible uml transformation framework. In *ASE '99: Proceedings of the 14th IEEE international conference on Automated software engineering*, pages 275–278, Washington, DC, USA, 1999. IEEE Computer Society.

[KT05] M. Kalinowski and G. H. Travassos. Software technologies: The use of experimentation to introduce ispis, a software inspection framework, into the industry. In *Proceedings of ESELAW05*, 2005.

[KWB03] A. Kleppe, J. Warmer, and W. Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison Wesley, 2003.

[MM99] M. L. Major and J. D. McGregor. Using guided inspection to validate uml models. *SEW99*, 1999.

[MS01] J. D. McGregor and D. A. Sykes. *A Practical Guide to Testing Object-Oriented Software*. Object Technology Series. Addison-Wesley, 2001.

[Pen03] T. Pender. *UML Bible*. John Wiley & Sons, 2003.

[Pre01] R. Pressman. *Software Engineering: A practitioner approach*. McGraw-Hill, New York, 5 edition, 2001.

[Pro09a] AMMA Project. Atlas transformation language. http://www.sciences.univ-nantes.fr/lina/atl/, 2009.

[Pro09b] Eclipse Project. Mofscript user guide. http://www.eclipse.org/gmt/mofscript/doc/MOFScript-User-Guide.pdf, 2009.

[RJB98] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1 edition, 1998.

[RMR09] A. Rocha, P. Machado, and F. Ramalho. Automação da técnica de inspeção guiada usando mda e simulação de modelos. In *3rd Simpósio Brasileiro de Componentes, Arquiteturas e Reutilização de Software*, pages 182–194, 2009.

[SB99] R. Solingen and E. Berghout. *The Goal/Question/Metric Method: a pratical guide for quality improvement of software development.* The McGraw-Hill Companies, United Kingdom, 1999.

[SJLY00] C. Sauer, D. R. Jeffery, L. Land, and Philip Yetton. The effectiveness of software development technical reviews: A behaviorally motivated program of research. *IEEE Transactions on Software Enfineering*, 2000.

[Som07] I. Sommerville. *Software Engineering*. Addison-Wesley, United Kingdom, 8 edition, 2007.

[SRB00] F. Shull, I. Rus, and V. Basili. How perspective based reading can improve requirements inspections. *IEEE*, 2000.

[ST04] L. F. S. Silva and G. H. Travassos. Tool-supported unobtrusive evaluation of software engineering process conformance. In *ISESE '04: Proceedings of the 2004 International Symposium on Empirical Software Engineering*, pages 127–135, Washington, DC, USA, 2004. IEEE Computer Society.

[TKG+05] T. D. Trong, N. Kawane, S. Ghosh, R. France, and A. Andrews. A tool-supported approach to testing uml design models. In *ICECCS '05: Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems*, pages 519–528, Washington, DC, USA, 2005. IEEE Computer Society.

[TSCB99] G. H. Travassos, F. Shull, J. Carver, and V. R. Basili. Reading techniques for oo design inspections. *XIV Brazilian Symposium on Software Engineering*, 1999.

Apêndice A

Artefatos para Perspective-Based Reading - PBR

A.1 Questões na Perspectiva do Usuário

Assuma que você é um usuário do sistema. A preocupação do usuário é garantir que a especificação do sistema após a etapa de análise está completa, sem erros e que satisfaz os requisitos do usuário. Isto significa que ela não deve estar inconsistente entre os outros modelos de análise, tal como a especificação de requisitos, diagramas caso de uso e de seqüência.

Siga as tarefas definidas nos passos 1 ao 3. Para cada passo você deve localizar o documento correspondente e seguir as instruções, realizar as tarefas necessárias e responder cada uma das questões. Descreva as tarefas no formulário de comentários como também qualquer comentário sobre elas. Quando você detectar um defeito, marque-o no diagrama e preencha a informação necessária no formulário de registro de defeitos.

Passo 1 Localize o Diagrama de Seqüência

O diagrama de seqüência mostra a interação entre os objetos do sistema. Estas interações acontecem em seqüências específicas em um tempo apropriado. Verifique os diagramas de seqüência e responda as questões a seguir.

1.1. Cada objeto tem ao menos uma mensagem enviada ou recebida?

1.2. Os nomes de cada objeto e de cada mensagem estão definidos?

Passo 2 Localize os Diagramas de Seqüência e a Especificação de Requisitos

Os diagramas de seqüência devem estar em conformidade com os requisitos do sistema, definidos em uma especificação de requisitos. Faça uma lista dos objetos mencionados nos diagramas de seqüência. Depois responda as seguintes questões.

2.1. Todos os objetos do diagrama de seqüência estão relacionados ao domínio descrito na especificação de requisitos?

2.2. Há alguma inconsistência entre os diagramas de seqüência e a especificação de requisitos?

Passo 3 Localize os Diagramas de Seqüência e os Diagramas de Caso de Uso

Os diagramas de seqüência precisam satisfazer o comportamento do sistema descrito no diagrama de caso de uso. Escreva os nomes dos diagramas de seqüência, que correspondem a cada caso de uso do diagrama de caso de uso próximo a cada caso de uso. Depois disto, responda as seguintes questões.

3.1. Cada caso de uso do diagrama de caso de uso está implementado em algum diagrama de seqüência?

3.2. Todos os objetos e mensagens importantes entre os objetos estão presentes nos diagramas de seqüência?

A.2 Questões na Perspectiva do Analista

Assuma que você é o analista do sistema. A preocupação do analista é assegurar que as necessidades do analista estão completamente satisfeitas de acordo com os seguintes documentos: Especificação de Requisitos, Diagrama de Classes e Diagrama de Seqüência. Durante o processo de inspeção você irá precisar inspecionar aqueles documentos a fim de detectar defeitos nos diagramas de classes e de seqüência do ponto de vista do analista.

Realize as tarefas seguindo os passo 1 ao 3. Para cada passo você deve localizar os documentos correspondentes, seguindo as instruções e responder as questões dadas. Quando você detectar algum defeito, marque-o no diagrama e preencha as informações necessárias no formulário de registro de defeitos. Se você tiver mais algum comentário escreva-os no formulário de comentários.

Passo 1 Localize o Diagrama de Classes

Analise o diagrama de classes e responda as seguintes questões.

1.1. O nome de cada classe está definido?

1.2. A multiplicidade de todas as associações estão definidas?

Passo 2 Localize o Diagrama de Classes, Especificação de Requisitos e Diagramas de Caso de Uso

O diagrama de classes precisa apresentar todas as classes de objetos necessárias, seus atributos, métodos e associações entre as classes. Compare o diagrama com a especificação de requisitos e diagramas de caso de uso para assegurar que não há inconsistências entre eles. Faça uma lista dos objetos mencionados na especificação de requisitos. Depois responda as seguintes questões.

2.1. Todos os objetos listados na especificação de requisitos estão representados no diagrama de classes?

2.2. Há algum elemento redundante no diagrama de classes?

2.3. Todas as classes e associações estão definidas?

2.4. Todas as classes necessárias para criar os casos de uso do diagrama de caso de uso estão definidas no diagrama de classes?

Passo 3 Localize o Diagrama de Classes e os Diagramas de Seqüência

Os diagramas de seqüência apresentam a interação entre os objetos do sistema. Estas interações acontecem em certa seqüência no tempo apropriado. Faça uma lista de todos os objetos incluídos nos diagramas de seqüência. Compare a lista com o diagrama de classes. Esteja certo que todos os objetos dos diagramas de seqüência estão definidos no diagrama de classes. Analise as mensagens entre os objetos no diagrama de seqüência para ter certeza que elas estão definidas como métodos ou atributos da classe correspondente no diagrama de classes. Analise também a relação entre os objetos do diagrama de seqüência. Assegure-se que a relação entre dois objetos, que existem no diagrama de seqüência, existe também entre os mesmos objetos no diagrama de classes. Depois disso, responda as seguintes questões.

3.1. Todos os objetos dos diagramas de seqüência estão definidos no diagrama de classes?

3.2. Todas as mensagens entre os objetos no diagrama de seqüência correspondente estão definidas como métodos ou atributos na classe correspondente no diagrama de classes?

3.3. A relação entre dois objetos que existem no diagrama de seqüência existe também entre as mesmas classes de objeto no diagrama de classes?

3.4. Há algum elemento redundante ou faltando no diagrama de seqüência?

Apêndice B

Especificação do Sistema de Quiz

Sistema de perguntas e respostas (*Quiz*), que será utilizado por um empresa para identificar qual área da computação cada um de seus funcionários mais se identificam. Este sistema é composto por dois componentes: (i) Quiz, componente servidor e (ii) *Quiz Manager*, componente cliente.

A Figura B.1 apresenta o diagrama de classes do sistema Quiz. O contexto é responsável pela criação de instâncias dos componentes servidor e cliente e pela integração dos mesmos. O componente *QuestionGenerator* é encarregado por gerar as questões, que serão utilizadas pelo componente Quiz.

B.1 O componente Quiz

O componente Quiz atua como um servidor fornecendo todos os serviços para a execução de um quiz. Cada instância deste componente será um objeto independente com um tempo de vida que é iniciado no instante em que este objeto é inicializado. O tempo de vida de cada objeto pode terminar de duas formas:

- quando todas as questões forem respondidas;
- quando o usuário resolver, por algum motivo, abortar a execução.

A qualquer instante, durante a execução do quiz, as funcionalidades de *help* e de *pause* podem ser invocadas. O *help* tem o objetivo de fornecer uma ajuda ao usuário nas questões correntes. O *pause* pausa a execução do quiz por um determinado intervalo de tempo.

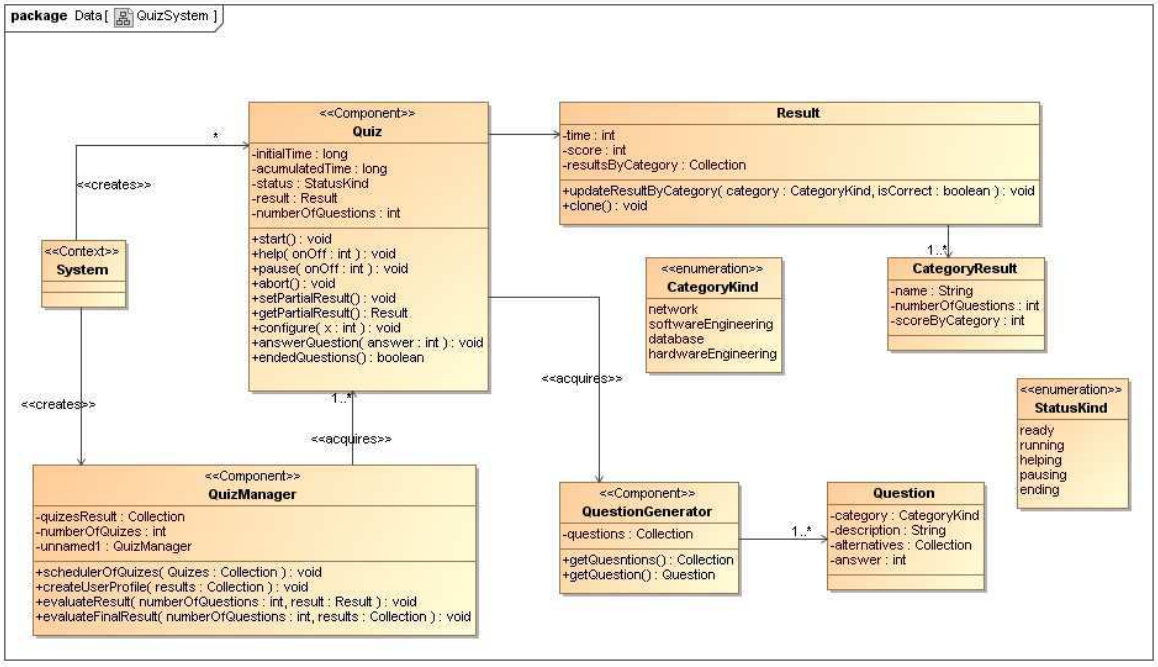

Figura B.1: Diagrama de classes do sistema Quiz.

Uma instância de um Quiz pode se encontrar em um dos seguintes estados: *Ready* (pronto), *Running* (em execução), *Pausing* (em pausa), *Helping* (em ajuda) e *Ending* (finalizando). O comportamento da instância do Quiz é ilustrado no diagrama de estados comportamentais da Figura B.2.

B.2 O componente *QuizManager*

O componente QuizManager atua como cliente, fazendo uso dos serviços fornecidos pelo componente servidor (Quiz) e utilizando os dados resultantes destes serviços para produzir outros resultados. Este componente tem as seguintes funcionalidades:

- escalonar cada quiz criado pelo contexto para o conjunto de usuários que irão respondê-los;
- fornecer uma avaliação parcial dos resultados no final da execução de cada quiz;
- fornecer uma avaliação final dos resultados de cada quiz quando todos forem finalizados;

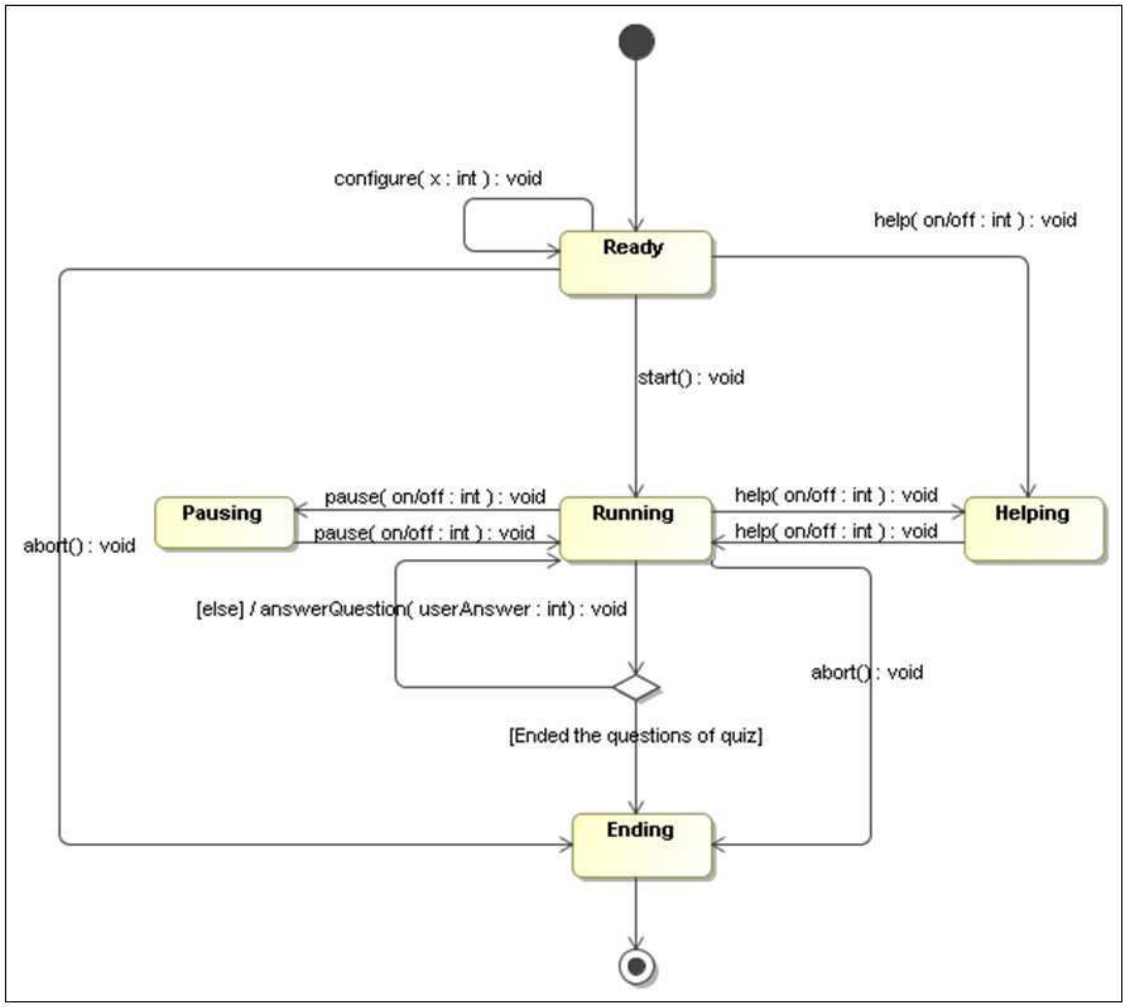

Figura B.2: Diagrama de estados do componente Quiz.

- criar um perfil para cada usuário, ou seja, enquanto o usuário está respondendo o quiz, paralelamente o *QuizManager* irá obter informações, como por exemplo, o tempo que o usuário demora para responder determinado tipo de questão, e no final fornecerá a informação de qual área ou departamento da empresa aquele funcionário tem mais afinidade.

B.3 Cenário de Execução do Sistema

A seguir serão apresentados os diagramas de seqüência que representam os cenários de execução do sistema Quiz. A Figura B.3 mostra um cenário onde um quiz é criado e enviado para ser respondido. Pode-se notar que o contexto da aplicação (a classe *System*) cria os

componentes cliente e servidor, QuizManager e Quiz, respectivamente.

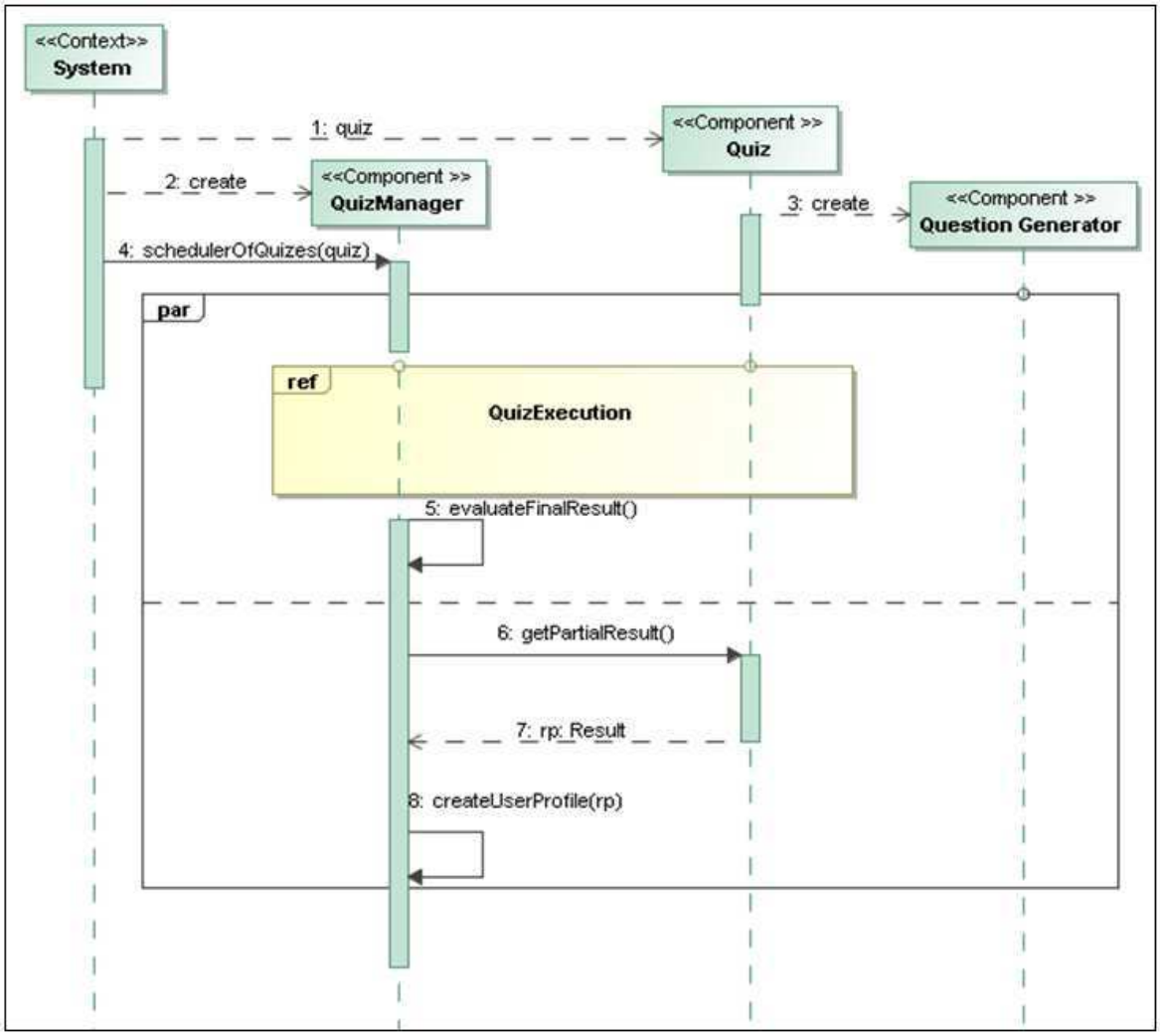

Figura B.3: Diagrama de seqüência do sistema de Gerenciamento de Quiz.

A Figura B.4 representa o cenário de "Start Execution", que realiza a execução de vários quiz. Durante a execução há a criação de um resultado, que será atualizado continuamente a cada resposta até a avaliação final de cada quiz.

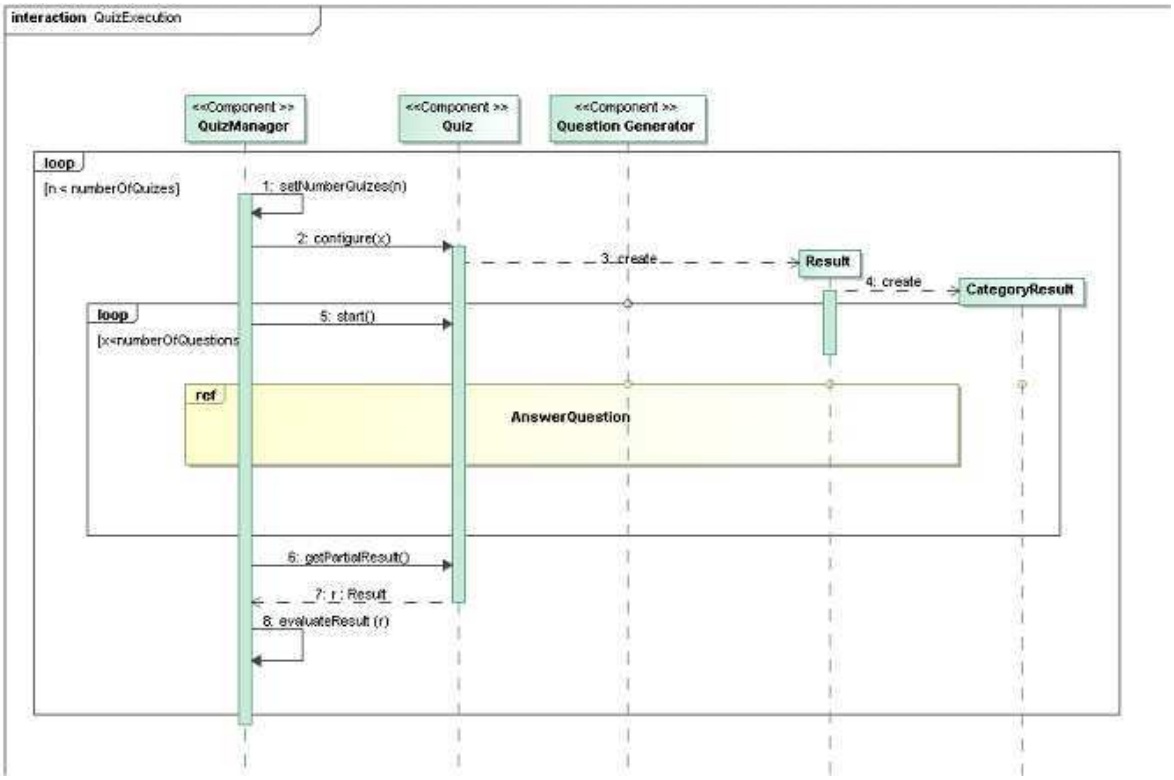

Figura B.4: Diagrama de seqüência do cenário “Start Execution”.

A Figura B.5 ilustra mais detalhadamente o comportamento do sistema relativo à exibição de uma nova questão do quiz e resposta do usuário.

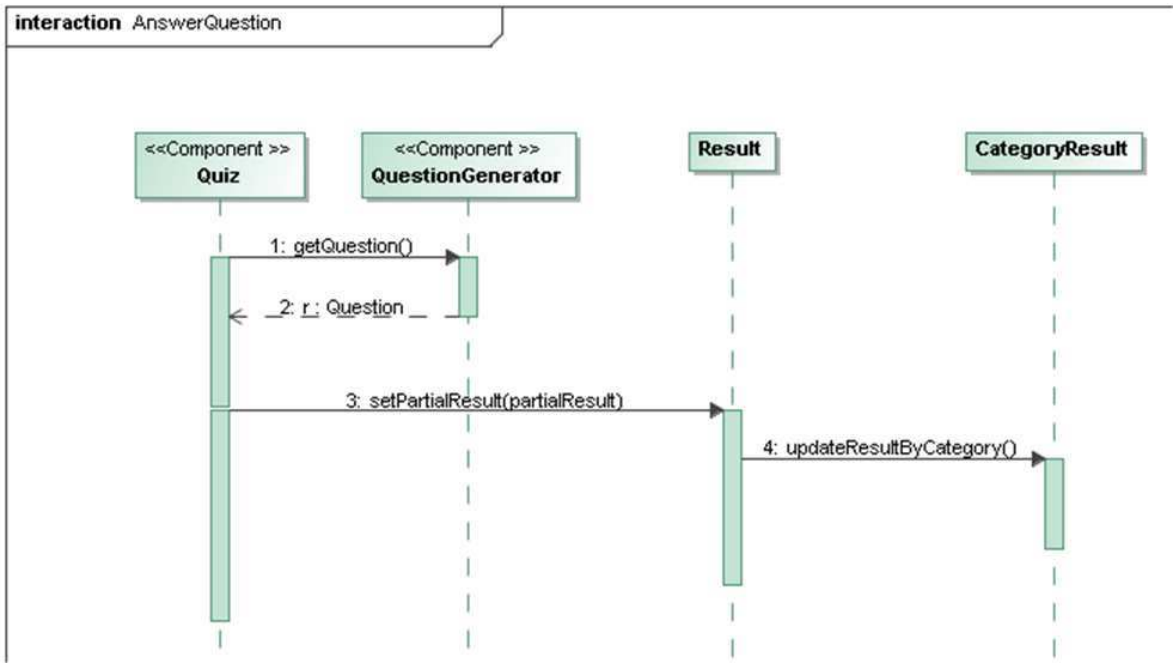

Figura B.5: Diagrama de seqüência do cenário “Answer Question”.

B.4 Cenários de Caso de Teste do Sistema Quiz

Para o sistema Quiz foram elaborados sete casos de teste que representam cada cenário de caso de uso do sistema. Estes cenários podem ser observados na Tabela B.1 abaixo.

Cenário: Quiz action “start” **Número:** 001	
Passos	**Resultado Esperado**
1. The user configures the number of questions.	The quiz is started successfully.
2. The user gets the Quiz starting.	
3. The questions are displayed.	
4. The accumulated time is started.	
Cenário: Quiz action “help” **Número:** 002	
Passos	**Resultado Esperado**
1. The quiz was started.	The quiz help is invocated
2. The user selects the Quiz help.	successfully.
3. The system shows a screen with the help.	
Cenário: Quiz action “pause” **Número:** 003	
Passos	**Resultado Esperado**
1. The quiz was started.	After the time is over,
2. The user selects the Quiz pause.	the quiz continues
3. The system pause the Quiz execution for a determined time.	his execution.
Cenário: Quiz action “abort” in the starting **Número:** 004	
Passos	**Resultado Esperado**
1. The quiz was started.	The quiz system is
2. The user selects the Quiz abort.	finished successfully.
Cenário: Quiz action “abort” during the execution **Número:** 005	
Passos	**Resultado Esperado**
1. The quiz was started.	The quiz system is finished successfully
2. The user answers some questions.	and the time and the user
3. The accumulated time is started and the user points are sated.	points are displayed.
4. The user selects the Quiz abort.	
Cenário: Quiz action “answer question” **Número:** 006	
Passos	**Resultado Esperado**
1. The quiz was started.	The system shows the
2. The questions are displayed.	partial result to
3. The user answers the question.	that question.
Cenário: Action Quiz “finish” **Número:** 007	
Passos	**Resultado Esperado**
1. The quiz was started.	The system is finished and the user final
2. The questions are displayed.	result is displayed with the
3. The user answers every the questions.	accumulated time and the points.

Tabela B.1: Casos de teste para cada cenário de caso de uso do sistema Quiz.

Apêndice C

Resultados para o Experimento do Sistema Quiz

Neste capítulo são apresentados todos os artefatos do sistema Quiz que compõem os resultados do Experimento 1 realizado através das técnicas de inspeção guiada automática, inspeção guiada manual e PBR.

C.1 Resultados da Inspeção Guiada Automática

Código Fonte C.1: Arquivo de USE com as restrições OCL para cada método do diagrama de classes do sistema Quiz.

```
1  constraints
2
3  context Quiz::configure(x : Integer)
4    pre statusReady: self.status = #ready
5    post numberQuestionsMoreThanZero: x > 0
6
7  context Quiz::start()
8    pre numberOfQuestionsMoreThanZero: self.numberOfQuestions > 0
9    pre statusQuizReady: self.status = #ready
10   pre initialTimeIsZero: self.initialTime = 0
11   post statusIsEnding: self.status = #ending
12
13 context Quiz::help(on_off : Integer)
14   pre statusQuizOkPre: if on_off = 1 then self.status = #running else self.status = #
       helping endif
```

```
15    post statusQuizOkPost: if on_off = 1 then self.status = #helping else self.status = #
          running endif
16
17  context Quiz::pause(on_off : Integer)
18    pre statusQuizOkPre: if on_off = 1 then self.status = #running else self.status = #
          pausing endif
19    post statusQuizOkPost: if on_off = 1 then self.status = #pausing else self.status = #
          running endif
20
21  context Quiz::abort()
22    pre statusRunning: self.status = #running
23    post statusEnding: self.status = #ending
24
25  context Quiz::getPartialResult() : Result
26    post resultOk: self.result.isDefined()
27    post resultScoreOk: self.result.score.isDefined()
28
29  context Quiz::setPartialResult(r : Result)
30    post resultScoreOk: self.result.score = r.score
31
32  context Quiz::answerQuestion(userAnswer : Integer)
33    pre statusRunning: self.status = #running
34
35  context Quiz::endedQuestions() : Boolean
36    pre statusRunning: self.status = #running
37    post statusEnding: self.status = #ending
38
39  context QuestionGenerator::getQuestions() : Sequence(Question)
40    pre statusRunning: self.quiz.status = #running
41    post numberQuestionsEqualsConfigure: self.quiz.numberOfQuestions = self.question->size()
42    post generatorHaveAllQuestions: self.questions->size() = self.question->size()
```

Nas Figuras C.1 e C.2 há os diagramas de seqüência gerados pela ferramenta USE referente ao cenário de caso de uso “Answer Question” e “Quiz Finish”.

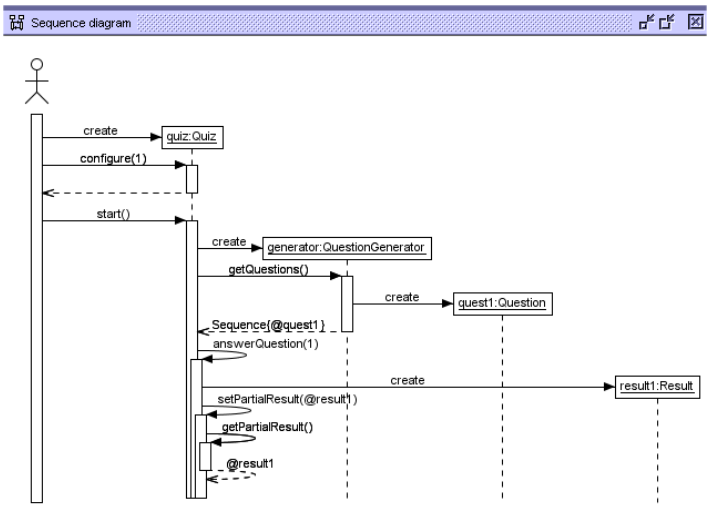

Figura C.1: Diagrama de seqüência gerado por USE para o caso de teste 06 do Quiz.

Após a execução da técnica de inspeção guiada automática, foram encontrados no total 16 defeitos válidos, sendo que destes defeitos 11 foram do tipo *MsgUnFoundError*, 2 foram do tipo *MsgPositionError* e 2 foram alertas do tipo *MsgDiffAlert* e 1 foi referente à uma ambigüidade que havia na especificação de requisitos. Além disso, 10 defeitos foram detectados durante a inspeção automática e 6 defeitos foram detectados durante a criação dos casos de teste para realizar a inspeção guiada automática. Estes 6 defeitos podem ser visualizados na Tabela C.1.

Tipo	Descrição
MsgUnFoundError	Faltam os 4 diagramas de classe para os casos de uso 02,03,04 e 05.
MsgUnFoundError	Deveria haver um método *time* na classe *Quiz* do diagrama de classes.
MsgUnFoundError	Falta um método para exibir o *help* na classe *Quiz* do diagrama de classes.
Ambigüidade	Ambigüidade com relação ao momento de quando a pausa acaba.
MsgUnFoundError	Falta *getAcumulatedTime* na classe *Quiz* do diagrama de classes.
MsgUnFoundError	Falta *getScore* na classe *Result* do diagrama de classes.

Tabela C.1: Defeitos encontrados no diagrama de classes do Quiz.

Os outros 10 defeitos podem ser visualizados na Figura C.3. Dentre os defeitos listados, alguns foram descartados, por exemplo, os defeitos relativos às mensagens de "*create*", pois estas entidades já haviam sido criadas em outro diagrama de seqüência.

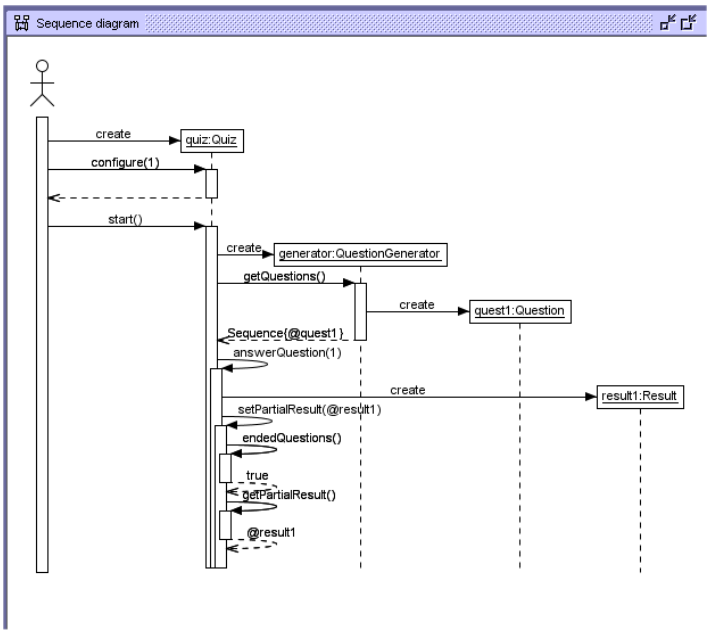

Figura C.2: Diagrama de seqüência gerado por USE para o caso de teste 07 do Quiz.

As Figuras C.4 e C.5 ilustram visualmente a presença dos defeitos encontrados pela inspeção guiada automática dentro dos diagramas de seqüência de projeto. A Figura C.6 apresenta os defeitos encontrados no diagrama de classes. Os defeitos foram marcados com uma letra “D” e os alertas com uma letra “A”, nestas figuras.

C.2 Resultados da Inspeção Guiada Manual

Os casos de teste utilizados para realizar a inspeção guiada manual está presente na Tabela B.1 do Apêndice B. O resultado da execução desta técnica sobre o diagrama de classes e de seqüência para verificar a conformidade destes modelos de projeto com relação à especificação de requisitos pode ser visualizado na Tabela C.2.

Caso de Teste	**Tipo**	**Defeito**
Inspeção no Diagrama de Classes		
Caso de Teste 01	Omissão	[1] Passo 4 - Falta um método para *setAcumulatedTime()* na classe Quiz.
Caso de Teste 02	-	OK
Caso de Teste 03	-	OK
Caso de Teste 04	-	OK
Caso de Teste 05	-	OK
Caso de Teste 06	-	OK
Caso de Teste 07	-	OK
Inspeção nos Diagramas de Seqüência		
Diagrama de Seqüência - Quiz Execution		
Caso de Teste 01	Omissão	[2] passo 4 - ação de *accumulated time is started* não pôde ser executada neste Diagrama de Seqüência.
Diagrama de Seqüência - Quiz Answer question		
Caso de Teste 06	Omissão Omissão	[3] passo 3 - ação de *answers the question* não está presente neste Diagrama de Seqüência. [4] passo 3 - é necessário criar uma linha de vida pra um *Question* neste Diagrama de Seqüência.
Caso de Teste 07	Omissão	[5] passo 8 - a ação de exibir o tempo acumulado não está presente neste Diagrama de Seqüência.

Tabela C.2: Defeitos encontrados nos diagramas de seqüência do Quiz.

C.3 Resultados de *Perspective-Based Reading* (PBR)

Questões na Perspectiva do Usuário

1. **Passo 1 Localize o Diagrama de Seqüência**

   1.1. Cada objeto tem ao menos uma mensagem enviada ou recebida?

   Sim.

   1.2. Os nomes de cada objeto e de cada mensagem estão definidos?

Sim.

2. **Passo 2 Localize os Diagramas de Seqüência e a Especificação de Requisitos**

2.1. Todos os objetos do diagrama de seqüência estão relacionados ao domínio descrito na especificação de requisitos?

Sim.

2.2. Há alguma inconsistência entre os diagramas de seqüência e a especificação de requisitos?

Sim.

[1]. Não há uma mensagem no diagrama de seqüência "Start Execution" para a situação de "abort" feita pelo usuário.

[2]. Não há mensagens no diagrama de seqüência para a ação de "help" e "pause".

[3]. Não há mensagens no diagrama de seqüência para calcular o tempo gasto.

3. **Passo 3 Localize os Diagramas de Seqüência e os Diagramas de Caso de Uso**

3.1. Cada caso de uso do diagrama de caso de uso está implementado em algum diagrama de seqüência?

[10]Não. Os Casos de Uso 02, 03, 04 e 05 não foram implementados.

3.2. Todos os objetos e mensagens importantes entre os objetos estão presentes nos diagramas de seqüência?

Não.

Questões na Perspectiva do Analista

1. **Passo 1 Localize o Diagrama de Classes**

1.1. O nome de cada classe está definido?

Sim.

1.2. A multiplicidade de todas as associações estão definidas?

Sim.

2. **Passo 2 Localize o Diagrama de Classes, Especificação de Requisitos e Diagramas de Caso de Uso**

   2.1. Todos os objetos listados na especificação de requisitos estão representados no diagrama de classes?

   Sim.

   2.2. Há algum elemento redundante no diagrama de classes?

   Não.

   2.3. Todas as classes e associações estão definidas?

   Sim.

   2.4. Todas as classes necessárias para criar os casos de uso do diagrama de caso de uso estão definidas no diagrama de classes?

   Sim.

3. **Passo 3 Localize o Diagrama de Classes e os Diagramas de Seqüência**

   3.1. Todos os objetos dos diagramas de seqüência estão definidos no diagrama de classes?

   Sim.

   3.2. Todas as mensagens entre os objetos no diagrama de seqüência correspondente estão definidas como métodos ou atributos na classe correspondente no diagrama de classes?

   Não.

   [4]. DS 01 - Mensagem “evaluateFinalResult()” não possui os mesmos parâmetros do método do diagrama de classes.

   [5]. DS 02 - Mensagem “setNumberQuizes(n)” do objeto “QuizManager” não faz parte do diagrama de classes.

   [6]. DS 02 - Mensagem “evaluateResult(r)” não possui os mesmos parâmetros do método do diagrama de classes.

[7]. DS 03 - Mensagem “setPartialResult(result)” do objeto “Result” não pertence a este objeto no diagrama de classes.

[8]. DS 03 - Mensagem “updateResultByCategory()” do objeto “CategoryResult” não pertence a este objeto no diagrama de classes.

3.3. A relação entre dois objetos que existem no diagrama de seqüência existe também entre as mesmas classes de objeto no diagrama de classes?

Sim.

3.4. Há algum elemento redundante ou faltando no diagrama de seqüência?

Sim.

[9]. O objeto “Question”.

Cada defeito apresentado utilizando a técnica PBR pode ser visualizado nas Figuras C.7, C.8 e C.9 de acordo com as numerações indicadas para cada defeito.

DEFECTS REPORT

TestCase	Quiz System - Test Case 07			Description	Action Quiz finish.				
	MsgUnFoundError		MsgSyntaxError		MsgPositionError		AbstractMsgError		MsgDiffAlert
Number	**Type**	**Description**		**Actual Sender**	**Actual Message**	**Actual Receiver**	**Expected Sender**	**Expected Message**	**Expected Receiver**
1		The expected message **create** should be in the sequence diagram, but it was not found.		Quiz		QuestionGenerator		create	
2		The expected message **create** should be in the sequence diagram, but it was not found.		QuestionGenerator		Question		create	
3		The expected message **answerQuestion** should be in the sequence diagram, but it was not found.		Quiz		Quiz		answerQuestion	
4		The expected message **create** should be in the sequence diagram, but it was not found.		Quiz		Result		create	
5		Verify the sender **Quiz** and the receiver **Result** of this message.		Quiz	setPartialResult	Result	Quiz		Quiz
6		Verify the sender **Result** and the receiver **CategoryResult** of this message.		Result	updateResultByCategory	CategoryResult	Quiz		Result
7		The expected message **endedQuestions** should be in the sequence diagram, but it was not found.		Quiz		Quiz		endedQuestions	
8		The expected message **true** should be in the sequence diagram, but it was not found.		Quiz		Quiz		true	
9		The expected message **getPartialResult** should be in the sequence diagram, but it was not found.		Quiz		Quiz		getPartialResult	
10		Verify the sender **Quiz** and the receiver **QuizManager** of this message.		Quiz	r : Result	QuizManager	Quiz		Quiz
11		The message **create** was not found in the test case sequence diagram.		Quiz	create	Result			
12		The message **create** was not found in the test case sequence diagram.		Result	create	CategoryResult			
13		The message **setPartialResult** was not found in the test case sequence diagram.		Quiz	setPartialResult	Result			
14		The message **getPartialResult** was not found in the test case sequence diagram.		QuizManager	getPartialResult	Quiz			
15		The message **evaluateResult** was not found in the test case sequence diagram.		QuizManager	evaluateResult	QuizManager			

Figura C.3: Relatório de defeitos da inspeção guiada automática para o sistema QUIZ.

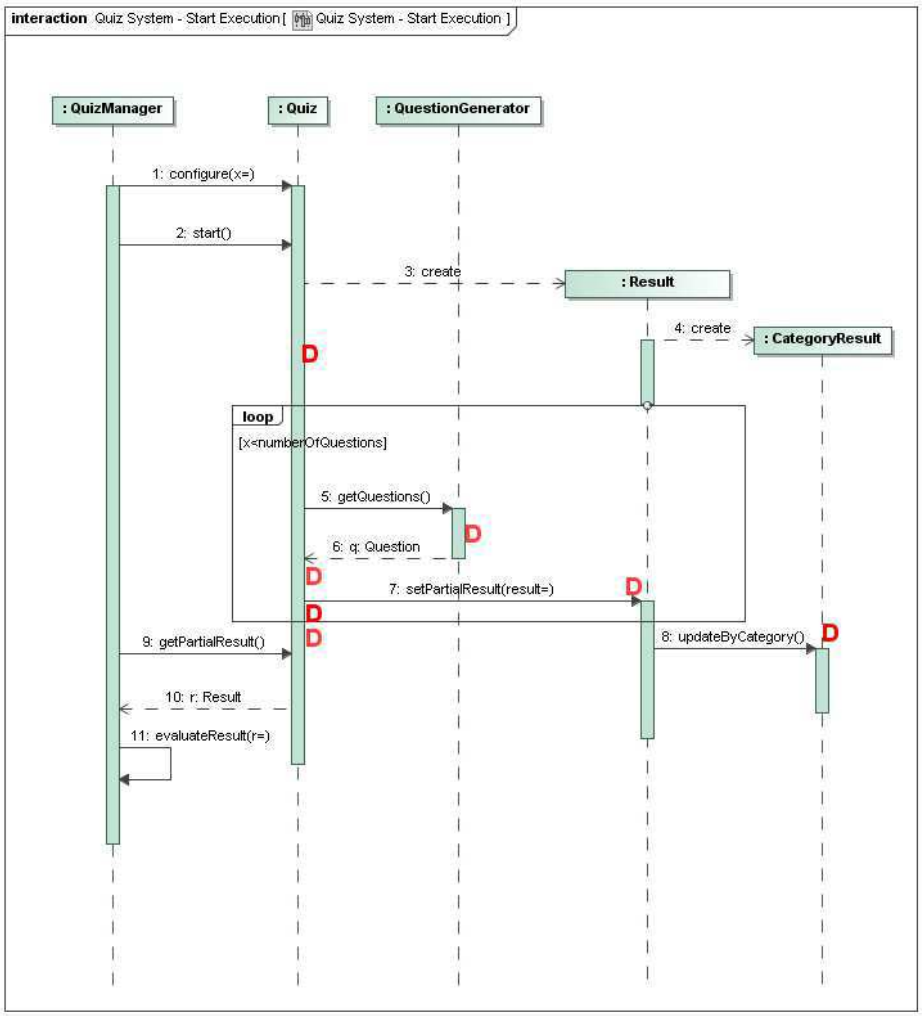

Figura C.4: Defeitos encontrados no diagrama de seqüência do caso de uso Answer Question na inspeção guiada automática.

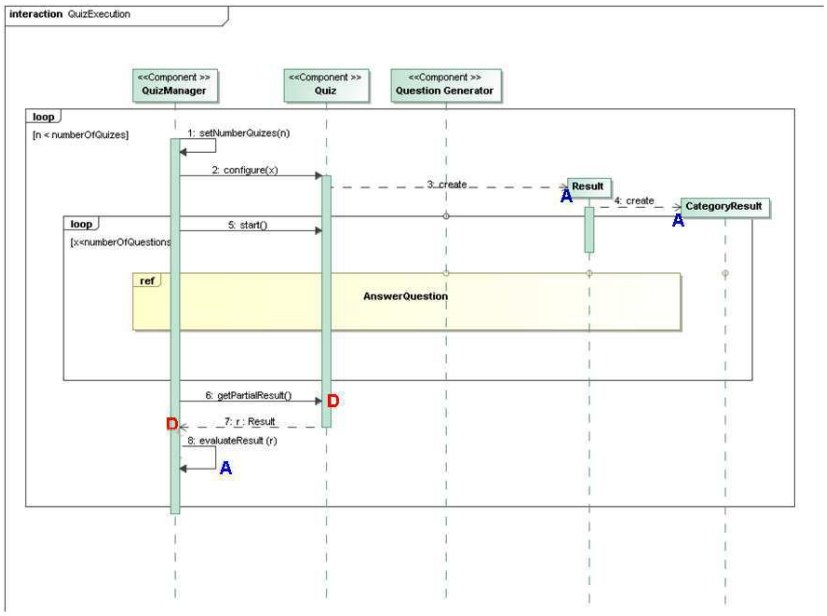

Figura C.5: Defeitos encontrados no diagrama de seqüência do caso de uso Quiz Finish na inspeção guiada automática.

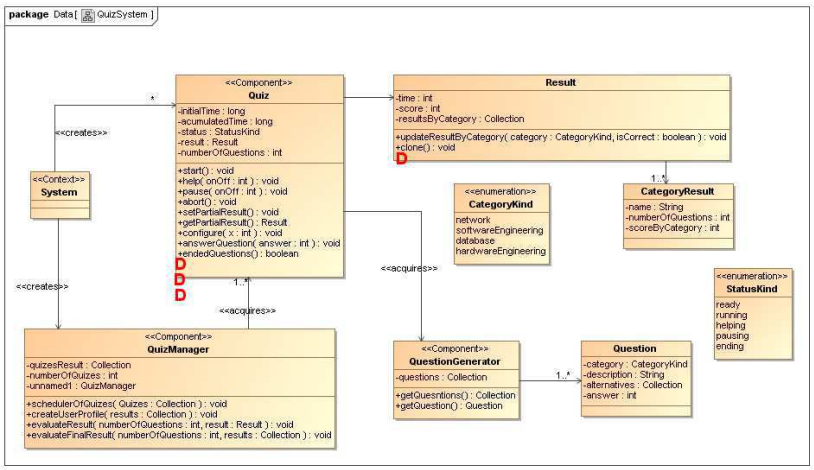

Figura C.6: Defeitos encontrados no diagrama de classes na inspeção guiada automática.

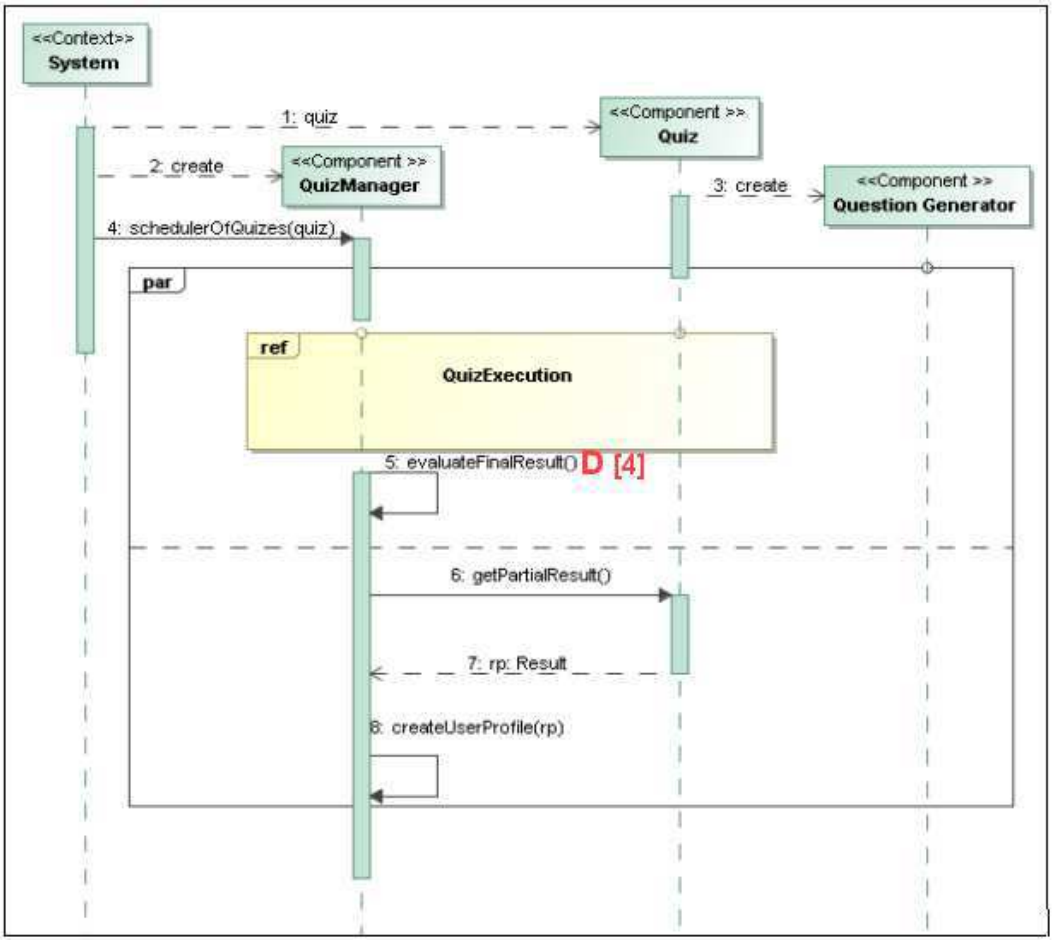

Figura C.7: Defeitos encontrados no diagrama de seqüência Quiz Manager usando PBR.

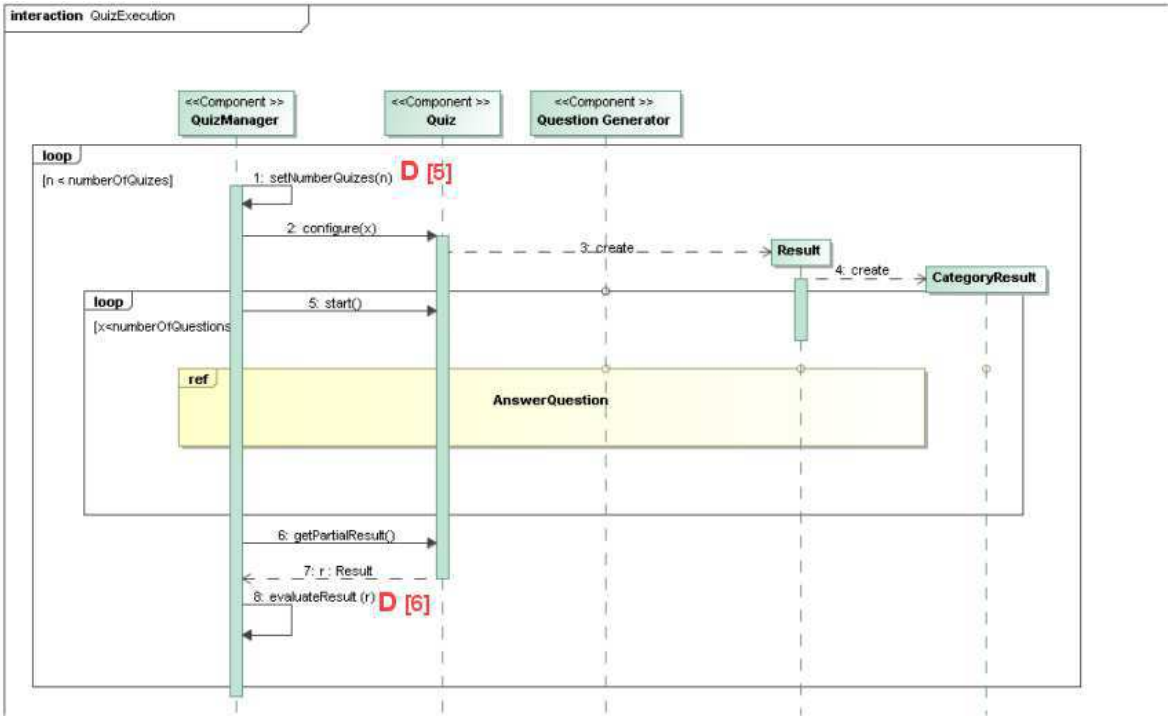

Figura C.8: Defeitos encontrados no diagrama de seqüência do caso de uso Quiz Execution usando PBR.

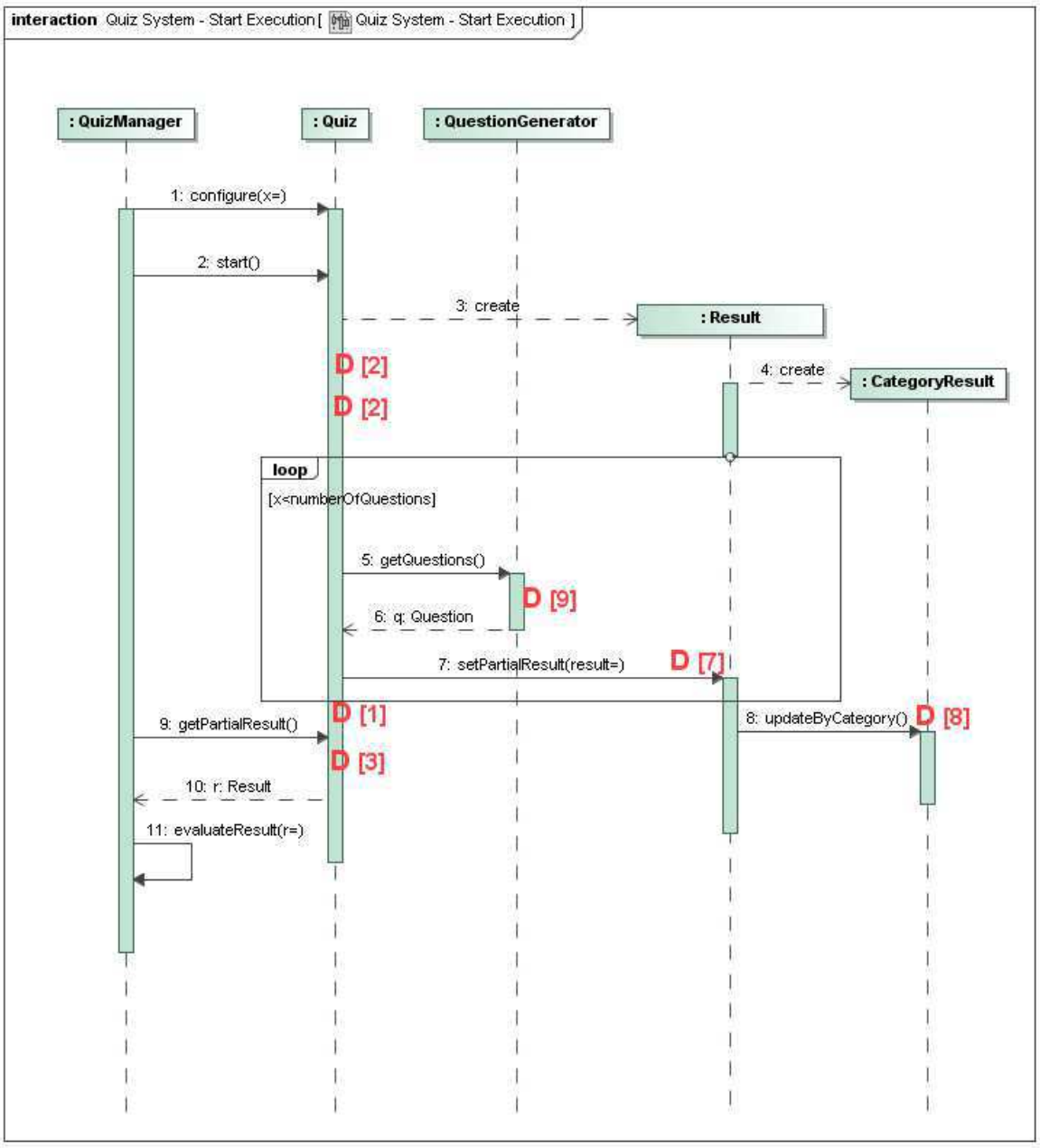

Figura C.9: Defeitos encontrados no diagrama de seqüência do caso de uso Quiz Answer usando PBR.

Apêndice D

Especificação do Sistema de ATM

D.1 Descrição do Domínio

Um banco tem várias máquina (ATM), que são conectadas em servidor via uma rede. Cada ATM tem um "Card reader", um "Cash dispenser", um "Keyboard/display", e um "Receipt printer". Ao utilizar o ATM, o cliente é reconhecido, o sistema valida o cartão e verifica se este cartão não expirou. Além disso, valida se a senha (PIN) que o cliente digitou está correta, como também verifica se este cartão não está descriminado como "roubado". O cliente só pode errar a senha por três vezes, caso contrário o cartão é confiscado. Se o cartão for roubado, ele também será confiscado.

Caso o cliente digite a senha corretamente, na tela do ATM irá aparecer as opções "withdraw", "query" e "transfer". Antes das operações de "withdraw" e "transfer" o sistema verifica se a conta possui fundos suficiente para o saque ou transferência da quantia. Além disso, o sistema verifica também o limite máximo diário permitido para aquela conta. Outra verificação é com relação à quantidade de dinheiro que há na máquina onde será realizado o saque, pois deve ser maior que o valor a ser sacado. Para realizar a transferência entre contas, o sistema verifica se as contas são válidas. O cliente poderá cancelar qualquer operação a qualquer momento. Ao final de cada operação a máquina imprime os dados da transação.

D.1.1 Diagrama de Caso de Uso

- Ator: *ATMCustomer*.

- Casos de uso para o *ATMCustomer*: *WithdrawFunds from an Account (Cheque or Savings), QueryAccount, TransferFunds from one account to another*
- Cada um destes casos de uso requerem o caso de uso *PIN validation*.

A Figura D.1 apresenta o diagrama de caso de uso para o sistema ATM, onde os casos de uso são relacionados com o ator "ATMCustomer" do sistema.

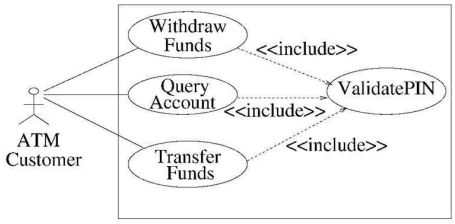

Figura D.1: Diagrama de caso de uso para o sistema ATM.

D.1.2 Diagrama de Classes

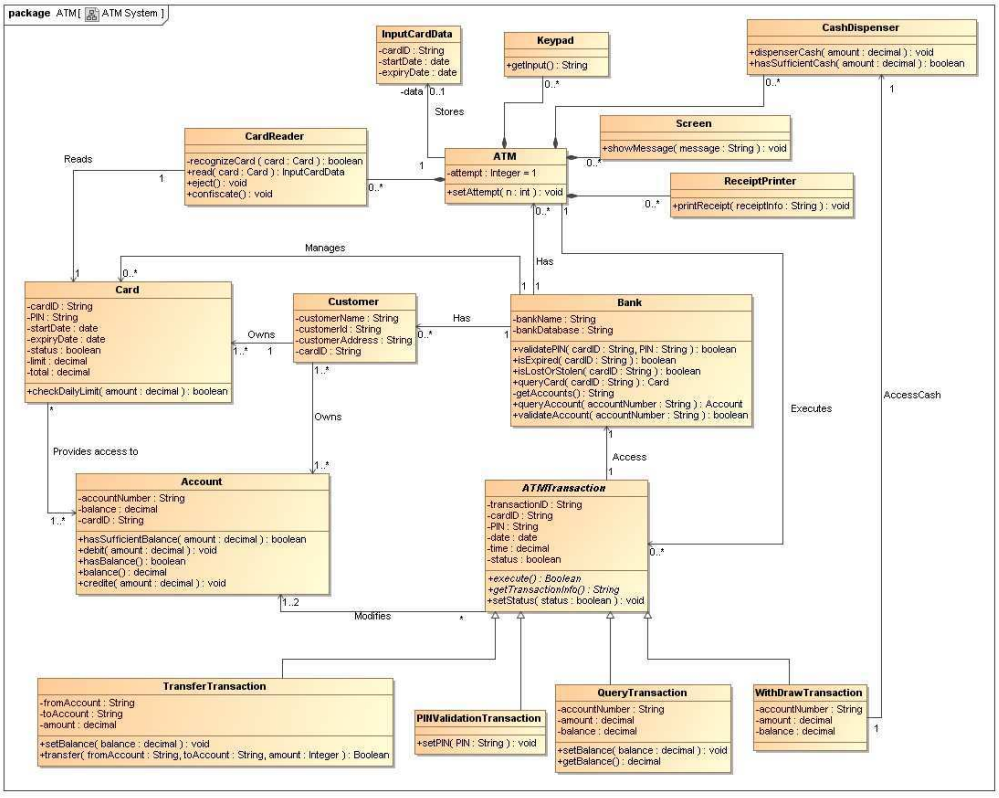

Figura D.2: Diagrama de classes para o sistema ATM.

D.1.3 Diagramas de Seqüência

As Figuras D.3, D.4, D.5 e D.6 representam os diagramas de seqüência para os cenários de caso de uso da especificação de requisitos.

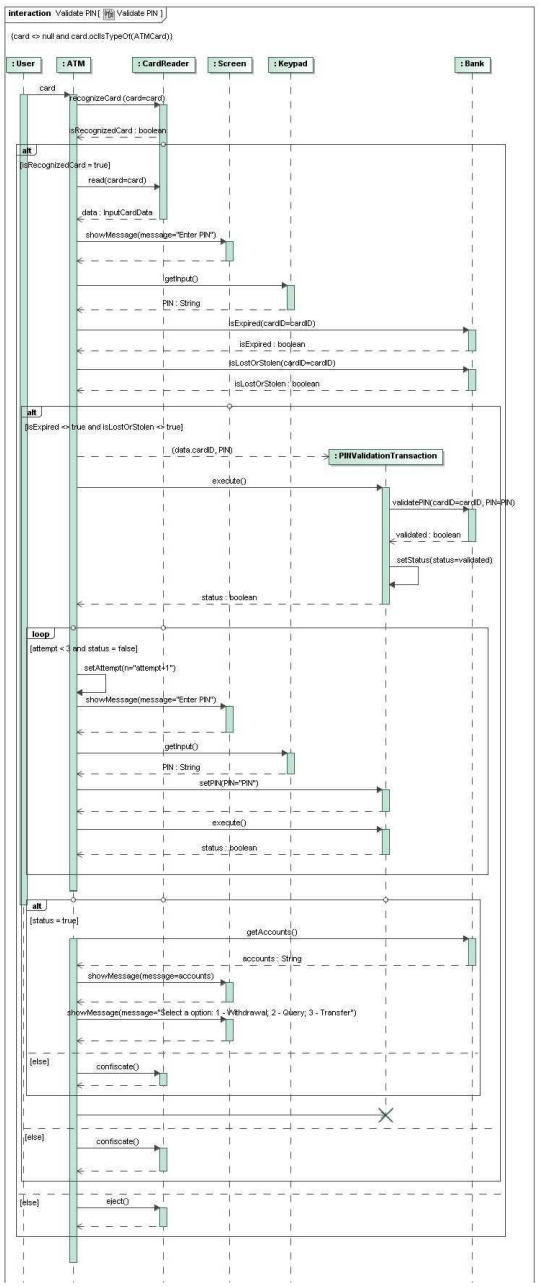

Figura D.3: Diagrama de seqüência do caso de uso *Validate PIN* para o sistema ATM.

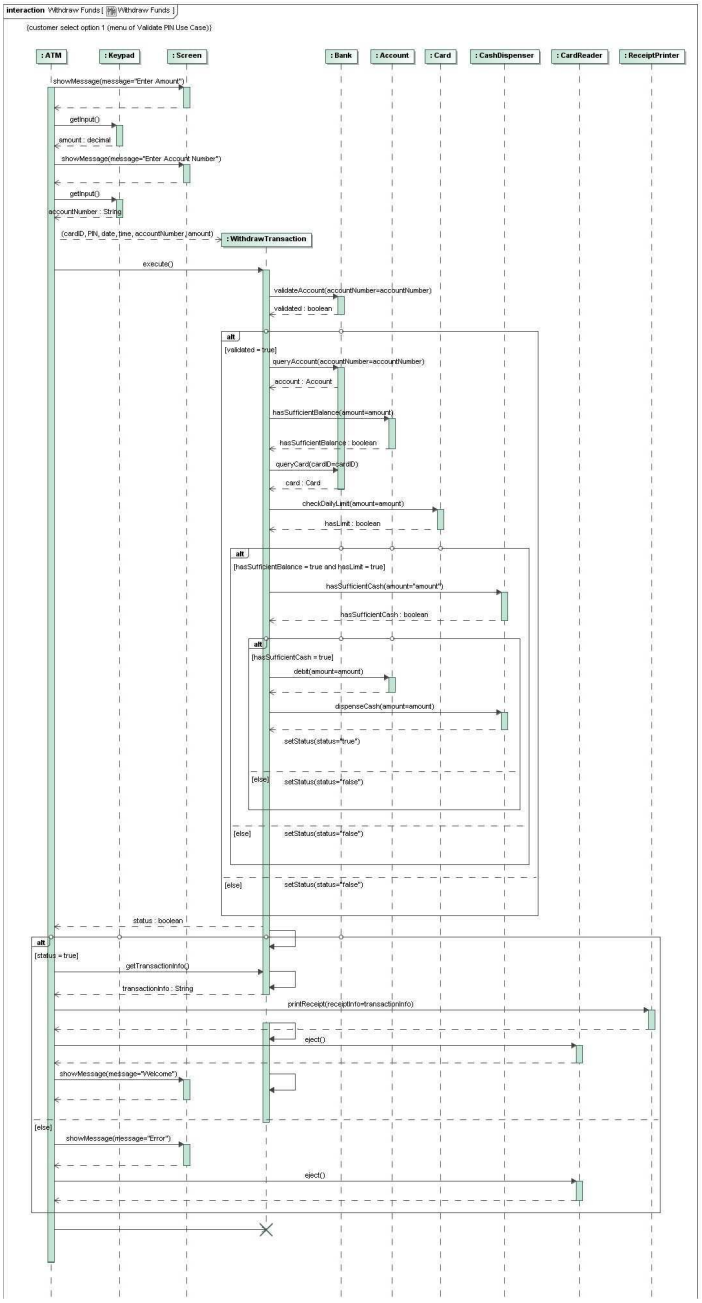

Figura D.4: Diagrama de seqüência do caso de uso *Withdraw Funds* para o sistema ATM.

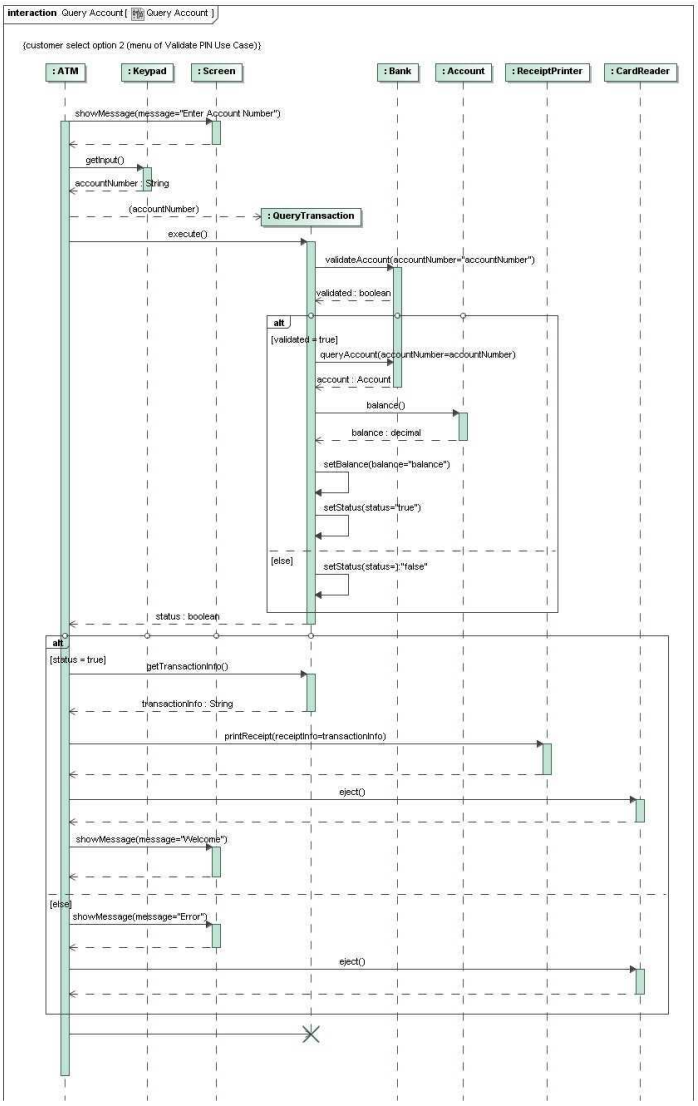

Figura D.5: Diagrama de seqüência do caso de uso *Query Account* para o sistema ATM.

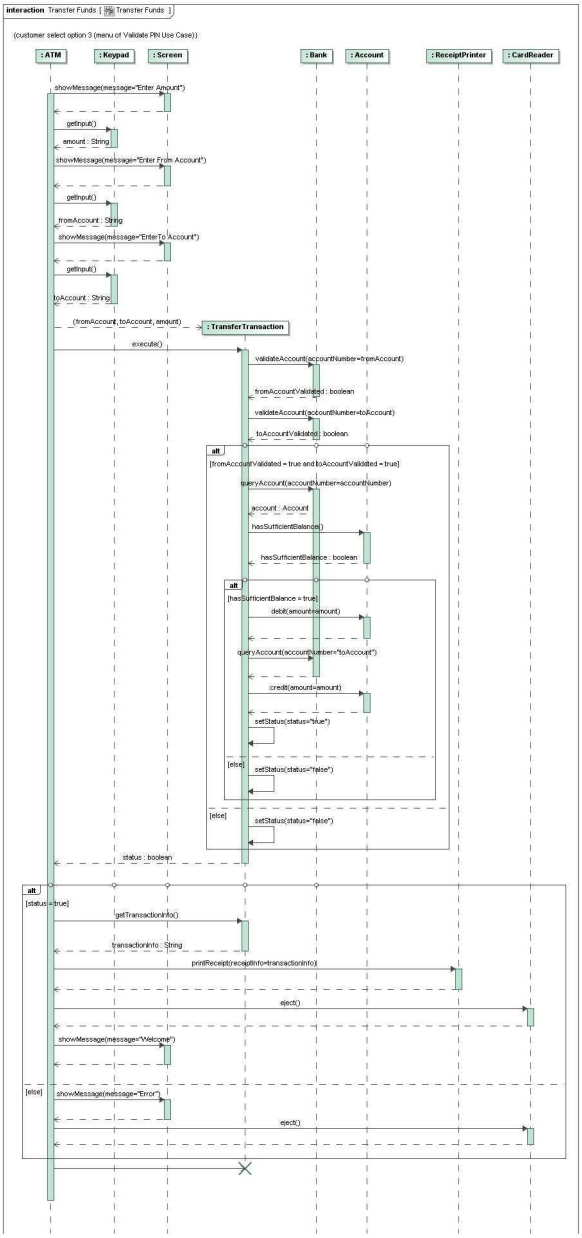

Figura D.6: Diagrama de seqüência do caso de uso *Transfer Funds* para o sistema ATM.

D.1.4 Cenários de Caso de Teste do Sistema ATM

A Tabela D.1 apresenta os casos de teste para cada cenário de caso de uso *Validate PIN* do sistema ATM. Estes casos de teste serão utilizados durante a realização da técnica de inspeção guiada tanto manual quanto automática.

Número: 001 **Cenário:** System validates customer PIN. **Pré-condição:** ATM Machine is idle.
Passos
1. Customer inserts the ATM Card into the Card Reader.
2. The system recognizes the card, it reads the card number.
3. System prompts customer for PIN number.
5. System checks the expiry date and whether the card is lost or stolen.
6. The card is valid, the system then checks whether the user-entered PIN matches the card PIN maintained by the system.
7. PIN numbers match, the system checks what accounts are accessible with the ATM Card.
Resultado Esperado: System displays customer accounts and prompts customer for transaction type: Withdraw, Query, or Transfer.
Número: 002 **Cenário:** The card hasn't recognized. **Pré-condição:** ATM Machine is idle.
Passos
1. Customer inserts the ATM Card into the Card Reader.
2. The system doesn't recognizes the card.
Resultado Esperado: The card is ejected.
Número: 003 **Cenário:** The card has expired. **Pré-condição:** ATM Machine is idle.
Passos
1. Customer inserts the ATM Card into the Card Reader.
2. The system recognizes the card, it reads the card number.
3. System prompts customer for PIN number.
4. Customer enters PIN.
5. The system determines that the card has expired.
Resultado Esperado: The card is confiscated.
Número: 004 **Cenário:** The card has expired. **Pré-condição:** ATM Machine is idle.
Passos
1. Customer inserts the ATM Card into the Card Reader.
2. The system recognizes the card, it reads the card number.
3. System prompts customer for PIN number.
4. Customer enters PIN
5. The system determines that the card has been reported lost or stolen.
Resultado Esperado: The card is confiscated.

Número: 005 **Cenário:** The customer-entered PIN doesn't match. **Pré-condição:** ATM Machine is idle.
Passos
1. Customer inserts the ATM Card into the Card Reader.
2. The system recognizes the card, it reads the card number.
3. System prompts customer for PIN number.
4. Customer enters PIN
5. System checks the expiry date and whether the card is lost or stolen.
6. The card is valid, the system then checks whether the user-entered PIN matches the card PIN maintained by the system.
7. The customer-entered PIN does not match the PIN number for the card.
Resultado Esperado: The system prompts for another PIN.
Número: 006 **Cenário:** The customer-entered incorrect PIN three times. **Pré-condição:** ATM Machine is idle.
Passos
1. Customer inserts the ATM Card into the Card Reader.
2. The system recognizes the card, it reads the card number.
3. System prompts customer for PIN number.
4. Customer enters PIN
5. System checks the expiry date and whether the card is lost or stolen.
6. The card is valid, the system then checks whether the user-entered PIN matches the card PIN maintained by the system.
7. The customer enters the incorrect PIN three times.
Resultado Esperado: The card is confiscated.
Número: 007 **Cenário:** The customer enters Cancel. **Pré-condição:** ATM Machine is idle.
Passos
1. Customer inserts the ATM Card into the Card Reader.
2. The system recognizes the card, it reads the card number.
3. System prompts customer for PIN number.
4. the customer enters Cancel.
Resultado Esperado: The system cancels the transaction and ejects the card.

Tabela D.1: Casos de teste o caso de uso *Validate PIN* do sistema ATM.

A Tabela D.2 apresenta os casos de teste para o caso de uso *Withdraw Funds*.

Número: 008 **Cenário:** Customer withdraws a specific amount of funds from a valid bank account. **Pré-condição:** ATM Machine is idle and a ValidPIN has been entered.
Passos
1. Include ValidatePlN use case.
2. Customer selects Withdraw, enters the amount, and selects the accountNumber.
3. System checks whether customer has enough funds in the account and whether the daily limit will not be exceeded.
4. All checks are successful, system authorizes dispensing of cash.
5. System dispenses the cash amount.
6. System prints a receipt showing transaction number, transaction type, amount withdrawn, and account balance.
Resultado Esperado: System ejects card and displays Welcome message.
Número: 009 **Cenário:** The account number is invalid. **Pré-condição:** ATM Machine is idle and a ValidPIN has been entered.
Passos
1. Include ValidatePlN use case.
2. Customer selects Withdraw, enters the amount, and selects the accountNumber.
3. The system determines that the account number is invalid.
Resultado Esperado: The system displays an error message and ejects the card.
Número: 0010 **Cenário:** The customer hasn't sufficient funds. **Pré-condição:** ATM Machine is idle and a ValidPIN has been entered.
Passos
1. Include ValidatePlN use case.
2. Customer selects Withdraw, enters the amount, and selects the accountNumber.
3. The system determines that there are insufficient funds in the customer's account.
Resultado Esperado: The system displays an apology and ejects the card.
Número: 0011 **Cenário:** The maximum allowable daily withdraw amount has been exceeded. **Pré-condição:** ATM Machine is idle and a ValidPIN has been entered.
Passos
1. Include ValidatePlN use case.
2. Customer selects Withdraw, enters the amount, and selects the accountNumber.
3. The system determines that the maximum allowable daily withdraw amount has been exceeded.
Resultado Esperado: The system displays an apology and ejects the card.

Número: 0012 **Cenário:** The ATM is out of funds. **Pré-condição:** ATM Machine is idle and a ValidPIN has been entered.
Passos
1. Include ValidatePlN use case.
2. Customer selects Withdraw, enters the amount, and selects the accountNumber.
3. The ATM is out of funds.
Resultado Esperado: The system displays an apology and ejects the card and shuts down the ATM.

Tabela D.2: Casos de teste para o caso de uso *Withdraw* do sistema ATM.

A Tabela D.3 apresenta os casos de teste para o caso de uso *Query Account* do sistema ATM.

Número: 013 **Cenário:** Customer receives the balance of a valid bank account. **Pré-condição:** ATM Machine is idle and a ValidPIN has been entered.
Passos
1. Include ValidatePlN use case.
2. Customer selects Query, enters account number.
3. System reads account balance.
4. System prints a receipt showing the transaction number, transaction type, and the account balance.
Resultado Esperado: The System ejects card and displays Welcome.
Número: 0014 **Cenário:** Customer receives the balance of a valid bank account. **Pré-condição:** ATM Machine is idle and a ValidPIN has been entered.
Passos
1. Include ValidatePlN use case.
2. Customer selects Query, enters account number.
3. The system determines that the account number is invalid.
Resultado Esperado: The System displays the error message and ejects the card.

Tabela D.3: Casos de teste para o caso de uso *Query Account* do sistema ATM.

A Tabela D.4 apresenta os casos de teste para cada cenário de caso de uso *Transfer Funds* do sistema ATM.

Número: 0015 **Cenário:** Customer transfers funds from one valid bank account to another. **Pré-condição:** ATM Machine is idle and a ValidPIN has been entered.
Passos
1. Include ValidatePlN use case.
2. Customer selects Transfer and enters amount, fromAccount, and toAccount.
3. The system determines that the customer has enough funds in the fromAccount, it performs the transfer.
4. System prints a receipt showing transaction number, transaction type, amount transferred and account balance.
Resultado Esperado: The System ejects card and displays Welcome.
Número: 0016 **Cenário:** The fromAccount number is invalid. **Pré-condição:** ATM Machine is idle and a ValidPIN has been entered.
Passos
1. Include ValidatePlN use case.
2. Customer selects Transfer and enters amount, fromAccount, and toAccount.
3. The system determines that the fromAccount number is invalid
Resultado Esperado: The System displays an error message and ejects the card.
Número: 017 **Cenário:** The toAccount number is invalid. **Pré-condição:** ATM Machine is idle and a ValidPIN has been entered.
Passos
1. Include ValidatePlN use case.
2. Customer selects Transfer and enters amount, fromAccount, and toAccount.
3. The system determines that the toAccount number is invalid.
Resultado Esperado: The System displays an error message and ejects the card.
Número: 0018 **Cenário:** The toAccount hasn't sufficient funds. **Pré-condição:** : ATM Machine is idle and a ValidPIN has been entered.
Passos
1. Include ValidatePlN use case.
2. Customer selects Transfer and enters amount, fromAccount, and toAccount.
3. The system determines that there are insufficient funds in the customer's fromAccount.
Resultado Esperado: The System displays an apology and ejects the card.

Tabela D.4: Casos de teste para o caso de uso *Transfer Funds* do sistema ATM.

Apêndice E

Resultados para o Experimento do Sistema ATM

Neste capítulo são apresentados todos os artefatos do sistema ATM que compõem os resultados do Experimento 2 realizado através das técnicas de inspeção guiada automática, inspeção guiada manual e PBR.

E.1 Resultados da Inspeção Guiada Automática

O Código Fonte E.1 contém todas as restrições definidas em OCL para cada método do diagrama de classes do sistema ATM definidas na ferramenta USE. Estas restrições auxiliaram o inspetor no momento de construção dos diagramas de seqüência gerados por USE.

Código Fonte E.1: Arquivo de USE com as restrições OCL para cada método do diagrama de classes do sistema ATM.

```
1  --constraints
2  context CardReader::recognizeCard(card : Card) : Boolean
3    pre bankIsDefined: self.atm.bank.isDefined()
4    pre atmIsDefined: self.atm.isDefined()
5    pre cardIsDefined: card.isDefined()
6    pre cardIsNotDefined: not self.card.isDefined() -- não associados
7    post cardReaderIsDefined: self.card.isDefined()
8    post cardIsRecognized: result = self.card.cardID.isDefined()
9
10 context CardReader::read(card : Card) : InputCardData
11   pre atmIsDefined: self.atm.isDefined()
12   pre inputDataIsNotDefined: not self.atm.data.isDefined()
13   post inputDataIsDefinided: self.atm.data.isDefined()
14   post cardIDInputDataEqualsCardID: self.atm.data.cardID = card.cardID
15   post inputData: result = self.atm.data
16
```

```
17  context Screen::showMessage(message : String)
18    pre atmIsDefined: self.atm.isDefined()
19    pre messageOk: message.isDefined()
20
21  context Keypad::getInput() : String
22    pre atmIsDefined: self.atm.isDefined()
23    post keypadIsInputedOk: result = self.input
24
25  context Bank::validatePIN(cardID : String, PIN : String) : Boolean
26    pre cardIDIsDefined: cardID.isDefined()
27    pre PINIsDefined: PIN.isDefined()
28    pre atmIsLinkedWithBank: self.atm->notEmpty()
29    post cardIsDefined: self.card->select(c| c.cardID = cardID)->notEmpty()
30    post PinValidationIsDefined: self.atmTrans->notEmpty()
31    post AttemptIsMoreThanZero: self.atm->select(a|a.attempt > 0)->notEmpty()
32    post PINIsOK: result = (self.atmTrans.cardID = cardID and self.atmTrans.PIN = PIN)
33
34  context Bank::isExpired(cardID : String) : Boolean
35    pre cardIDIsDefined: cardID.isDefined()
36    pre cardIsDefined: self.card->select(c| c.cardID = cardID)->notEmpty()
37    pre atmTransDateIsDefined: self.atmTrans.date.isDefined()
38
39    post cardIsNotExpired: result = not (self.atmTrans.cardID = cardID and self.card->
40          select(c| c.cardID = cardID)->select(c| c.expiryDate.year > self.atmTrans.date.year)->notEmpty())
41
42  context Bank::isLostOrStolen(cardID : String) : Boolean
43    pre cardIDIsDefined: cardID.isDefined()
44    pre cardIsDefined: self.card->select(c| c.cardID = cardID)->notEmpty()
45    pre statusIsDefined: self.atmTrans.status.isDefined()
46    post cardIsnotLostNorStolen: result = not (self.atmTrans.status = true)
47
48  context Bank::getAccount(cardID : String) : String
49    pre cardIsDefined: self.card->notEmpty()
50    pre cardIDIsDefined: cardID.isDefined()
51    post customerIsDefined: self.customer->notEmpty()
52    post customerOwnsCard: self.customer->select(c | c.cardID = cardID)->notEmpty
53    post accountIsDefined: self.customer->select(c | c.account->notEmpty())->notEmpty()
54    post customerOwnsAccount: self.customer->select(c| c.account->
55                    select(a | a.cardID = cardID)->notEmpty())->notEmpty()
56
57  context CardReader::eject()
58    pre cardIsDefined: self.card.isDefined()
59    post cardIsEjected: not self.card.cardID.isDefined() or
60                        if (self.card.cardID.isDefined()) then
61                           self.card.account->select(a|a.balance>0)->notEmpty() or
62                           self.card.account->select(a|a.accountNumber.isDefined)->isEmpty()
63                        else
64                           true
65                        endif
66
67  context CardReader::confiscate()
68    pre cardIsDefined: self.card.isDefined()
69    post cardIsConfiscate: not (self.card.expiryDate.year > self.atm.bank.atmTrans.date.year)
70                    or self.atm.bank.atmTrans.status = false or self.atm.attempt = 3
71
72  context ATM::setAttempt(n : Integer)
73    pre cardIsRecognized: self.data.cardID.isDefined()
74    post attemptIsSetted: self.attempt > 0
75
76  context Bank::validateAccount(accountNumber : String) : Boolean
77    pre accountIsDefined: self.customer->select(c|c.account->notEmpty())->notEmpty()
78    post accountNumberIsDefined: result = accountNumber.isDefined()
79
80  context Bank::queryAccount(accountNumber : String) : Account
```

```
81    pre accountIsDefined: self.customer->select(c|c.account->notEmpty())->notEmpty()
82    post accountNumberIsDefined:accountNumber.isDefined()
83    post accountIsValid: self.customer->select(c|c.account->notEmpty()).account->
84          select(a|a.accountNumber = accountNumber)->notEmpty
85
86  context Account::hasSufficientBalance(amount : Integer) : Boolean
87    pre amountIsDefined: amount.isDefined()
88    pre customerHasAccount: self.customer->notEmpty()
89    pre balanceIsDefined: self.balance.isDefined()
90    post hasSufficienteBalance: result = self.balance > amount
91
92  context Account::debit(amount : Integer)
93    pre amountIsDefined: amount.isDefined()
94    post wasDebited: self.balance = self.balance@pre - amount
95
96  context Account::credite(amount : Integer)
97    pre amountIsDefined: amount.isDefined()
98    post wasCredited: self.balance = self.balance@pre + amount
99
100 context Card::checkDailyLimit(amount : Integer) : Boolean
101   pre amountIsDefined: amount.isDefined()
102   pre limitIsDefined: self.limit.isDefined()
103   post hasDailyLimite: result = amount < self.limit
104
105 context CashDispenser::dispenserCash(amount : Integer)
106   pre amountIsDefined: amount.isDefined()
107   pre withDrawTransIsDefined: self.withDraw.isDefined()
108   pre withDrawTransBalanceOk: self.withDraw.balance.isDefined()
109   post cashIsDispenser: self.withDraw.balance > amount
110
111 context CashDispenser::hasSufficientCash(amount : Integer) : Boolean
112   pre amountIsDefined: amount.isDefined()
113   pre withDrawTransIsDefined: self.withDraw.isDefined()
114   pre withDrawTransBalanceOk: self.withDraw.balance.isDefined()
115   post hasSufficienteCash: result = amount <= self.withDraw.balance
116
117 context Account::balance() : Integer
118   pre balanceIsDefined: self.balance.isDefined()
119   post balanceValue: result = self.balance
120
121 context QueryTransaction::getBalance(accountNumber : String) : Integer
122   pre accountIsDefined: self.account->notEmpty()
123   pre accountNumberIsDefined: accountNumber.isDefined()
124   post queryIsRealized: result = self.balance
125
126 context TransferTransaction::transfer(fromAccount : String, toAccount : String, amount : Integer) : Boolean
127   pre fromAndToAccountsAreDefined: self.account->size() = 2
128   post transferDebitIsRealized: if self.account->select(a|a.accountNumber = fromAccount)->notEmpty() then
129                     result = self.account->select(a|a.balance@pre > a.balance@pre - amount)->notEmpty()
130                 else
131                     false
132                 endif
133   post transferCrediteIsRealized: if self.account->select(b|b.accountNumber = toAccount)->notEmpty() then
134                     result = self.account->select(b|b.balance@pre < b.balance@pre + amount)->notEmpty()
135                 else
136                     false
137                 endif
```

Os principais diagramas de seqüência de caso de teste gerados por USE podem ser visualizados nas Figuras E.1, E.2, E.3 e E.4. Foram criados um diagrama de seqüência para cada caso de teste. Estes casos de teste foram passados como entrada para realizar o passo

de inspeção guiada automática para gerar um relatório de defeitos. Os casos de teste em linguagem natural foram exibidos no Apêndice D, que contém a especificação de ATM.

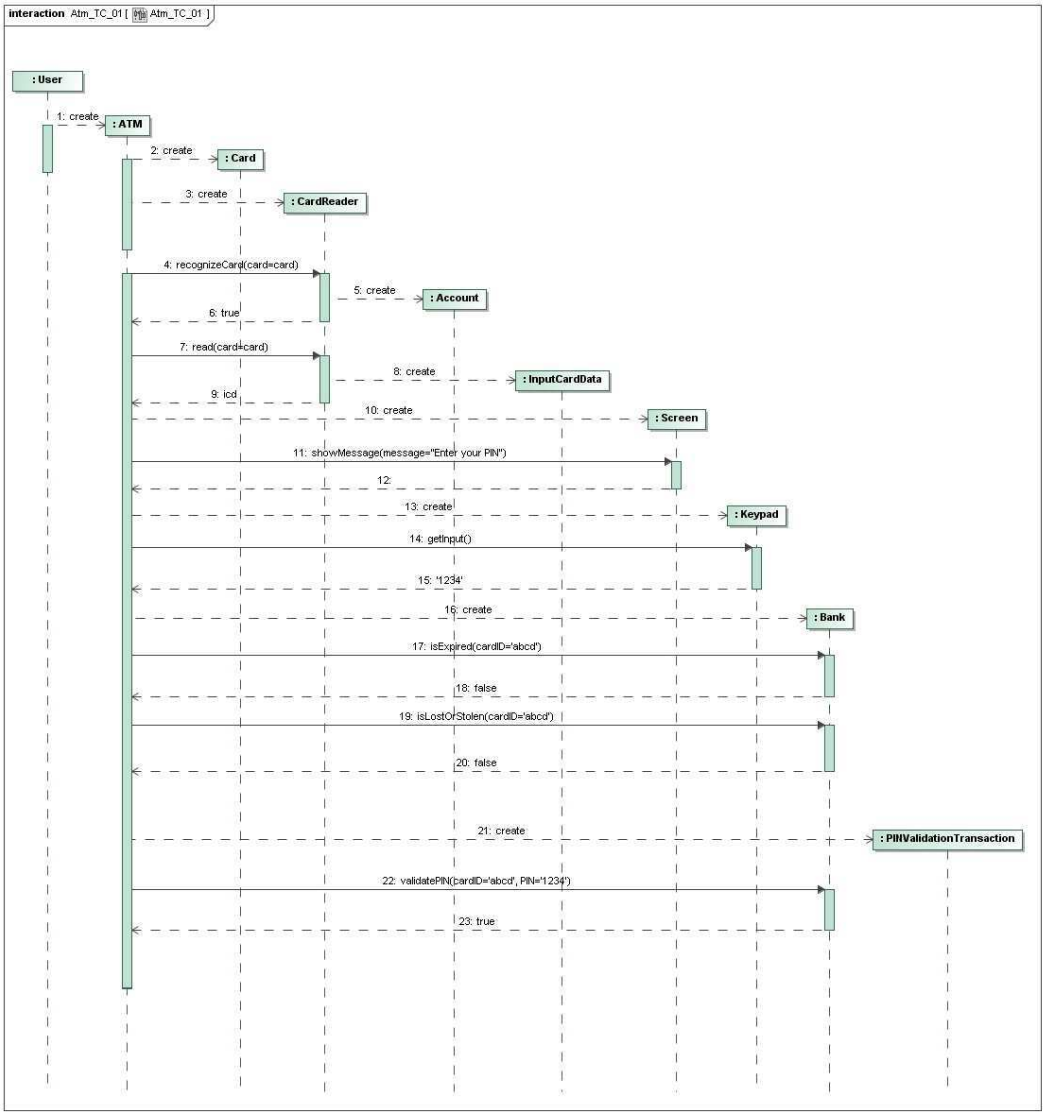

Figura E.1: Diagrama de seqüência do caso de teste 01 para o sistema ATM.

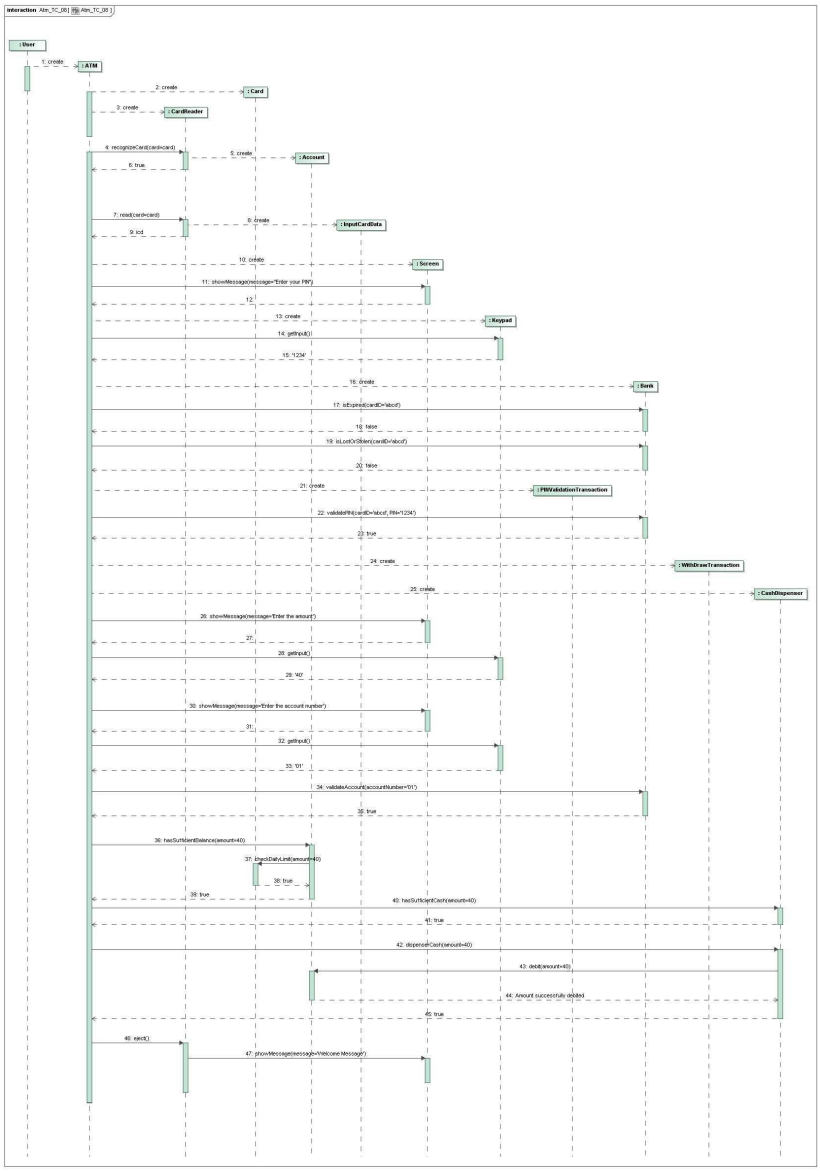

Figura E.2: Diagrama de seqüência do caso de teste 08 para o sistema ATM.

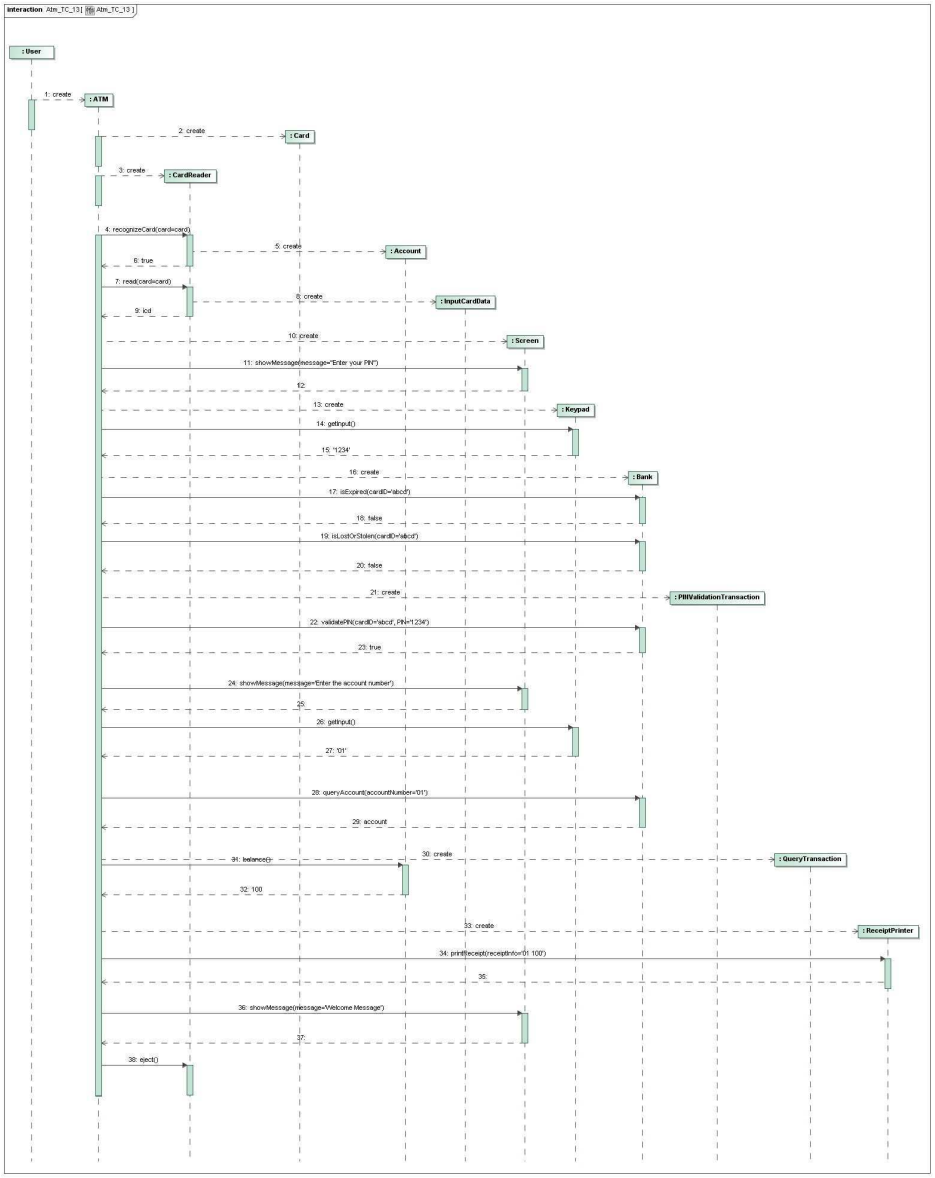

Figura E.3: Diagrama de seqüência do caso de teste 13 para o sistema ATM.

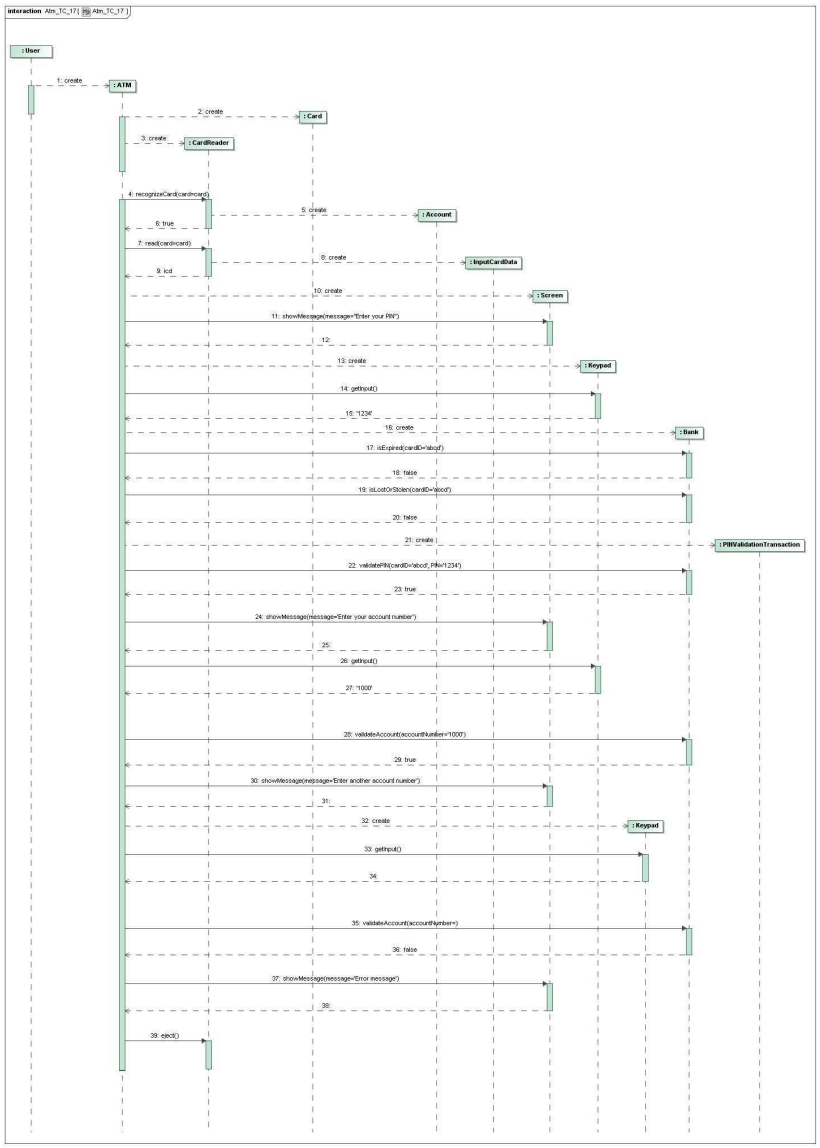

Figura E.4: Diagrama de seqüência do caso de teste 17 para o sistema ATM.

Após a execução da técnica de inspeção guiada automática, foram encontrados no total 28 defeitos válidos, sendo que destes defeitos 19 foram do tipo *MsgUnFoundError*, 4 foram do tipo *MsgSyntaxError*, 3 foram do tipo *AssociationError*, 1 foi do tipo Ambigüidade e 1 foi alerta do tipo *MsgDiffAlert*. Além disso, 17 defeitos foram detectados durante a inspeção automática e 11 defeitos foram detectados durante a criação dos casos de teste para realizar a inspeção guiada automática. Estes 11 defeitos podem ser visualizados na Tabela E.1.

Tipo	**Descrição**
MsgUnfounfError	A classe ATMCard é desnecessária, ela foi substituida pela classe Card inclusive a associação Reads.
AssociationError	Foi retirada a associação entre a classe ATMCard e CardReader.
AssociationError	A associação Has entre as classes ATM e Bank deve ser multidirecional para que seja possível navegar da classe ATM para a classe Bank.
Ambigüidade	Na classe Card o atributo *status* deve possuir o tipo *boolean* havia uma inconsistência com a classe ATMTransaction.
MsgUnfounfError	Na classe Account foi acrescentado atributo cardID.
MsgUnfounfError	A classe Account não é uma classe concreta.
AssociationDefect	Foi criada uma associação *Query* entre a classe Bank e Account para que seja possível a navegação entre as classes exigida pela operação queryAccount() na classe Bank.
MsgUnfounfError	Na classe Customer foi acrescentado o atributo cardID.
MsgSyntaxError	Os métodos da classe CardReader deveria ter como parâmetro um objeto do tipo ATMCard.
MsgUnfounfError	Faltou a multiplicidade entre CashDispenser e WithdrawTransaction.
MsgUnfounfError	Não há um caminho para realizar a ação de Cancelar a operação de validação e fazer eject()

Tabela E.1: Defeitos encontrados no diagrama de classes do ATM.

Os outros 17 defeitos podem ser visualizados nas Figuras E.5 e E.6.

DEFECTS REPORT

TestCase	Atm_TC_01			Description				
	MsgUnFoundError		MsgSyntaxError		MsgPositionError		AbstractMsgError		MsgDiffAlert
Number	**Type**	**Description**		**Actual Sender**	**Actual Message**	**Actual Receiver**	**Expected Sender**	**Expected Message**	**Expected Receiver**
1		The expected message **create** should be in the sequence diagram, but it was not found.		User		ATM		create	
2		The expected message **create** should be in the sequence diagram, but it was not found.		ATM		Card		create	
3		The expected message **create** should be in the sequence diagram, but it was not found.		ATM		CardReader		create	
4		The expected message **create** should be in the sequence diagram, but it was not found.		CardReader		Account		create	
5		The expected message **create** should be in the sequence diagram, but it was not found.		CardReader		InputCardData		create	
6		The expected message **create** should be in the sequence diagram, but it was not found.		ATM		Screen		create	
7		The expected message **create** should be in the sequence diagram, but it was not found.		ATM		Keypad		create	
8		The expected message **create** should be in the sequence diagram, but it was not found.		ATM		Bank		create	
9		The message's syntax **queryAccount(accountNumber=accountNumber)** is not in conformity with the message in the class diagram.		TransferTransaction	queryAccount(accountNumber=accountNumber)	Bank		queryAccount(accountNumber=fromAccount)	

Figura E.5: Relatório de defeitos para o sistema ATM.

DEFECTS REPORT

TestCase	Atm_TC_01			Description				
	MsgUnFoundError		MsgSyntaxError		MsgPositionError		AbstractMsgError	MsgDiffAlert
Number	**Type**	**Description**		**Actual Sender**	**Actual Message**	**Actual Receiver**	**Expected Sender**	**Expected Message**	**Expected Receiver**
10		The message's syntax **hasSufficientBalance()** is not in conformity with the message in the class diagram.		TransferTransaction	hasSufficientBalance()	Account		hasSufficientBalance(amount)	
11		The message's syntax **queryAccount(accountNumber='toAccount')** is not in conformity with the message in the class diagram.		TransferTransaction	queryAccount(accountNumber='toAccount')	Bank		queryAccount(accountNumber=toAccount)	
12		The expected message **create** should be in the sequence diagram, but it was not found.		CardReader		Account		create	
13		The expected message **create** should be in the sequence diagram, but it was not found.		ATM		RerceiptPrinter		create	
14		The expected message **create** should be in the sequence diagram, but it was not found.		ATM		CashDispenser		create	
15		The expected message **create** should be in the sequence diagram, but it was not found.		CardReader		Account		create	
16		The expected message **create** should be in the sequence diagram, but it was not found.		ATM		RerceiptPrinter		create	
17		The message **setStatus** was not found in the test case sequence diagram.		PINValidationTransaction	setStatus	PINValidationTransaction			

Figura E.6: Relatório de defeitos para o sistema ATM (continuação).

As Figuras E.7, E.8, E.9, E.10 ilustram visualmente a presença dos defeitos encontrados pela inspeção guiada automática dentro dos diagramas de seqüência de projeto. A Figura E.11 apresenta os defeitos encontrados no diagrama de classes. Os defeitos foram marcados com uma letra “D” e os alertas com uma letra “A”, nos diagramas de seqüência e com um “X” no diagrama de classes.

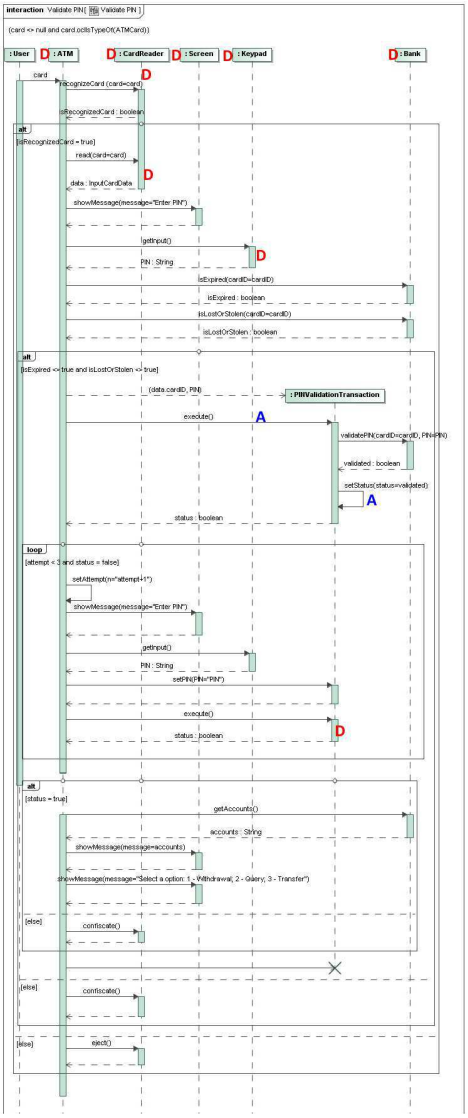

Figura E.7: Defeitos no diagrama de seqüência do caso de uso Validate PIN.

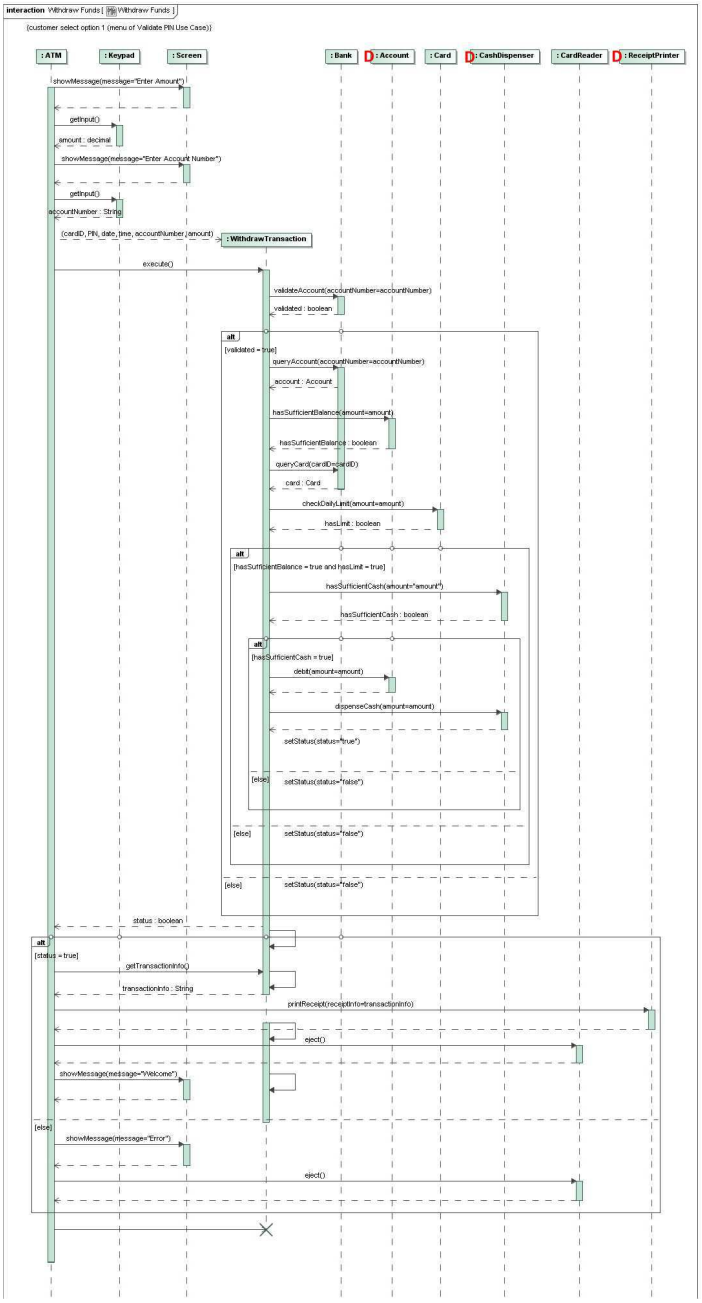

Figura E.8: Defeitos no diagrama de seqüência do caso de uso Withdraw Funds.

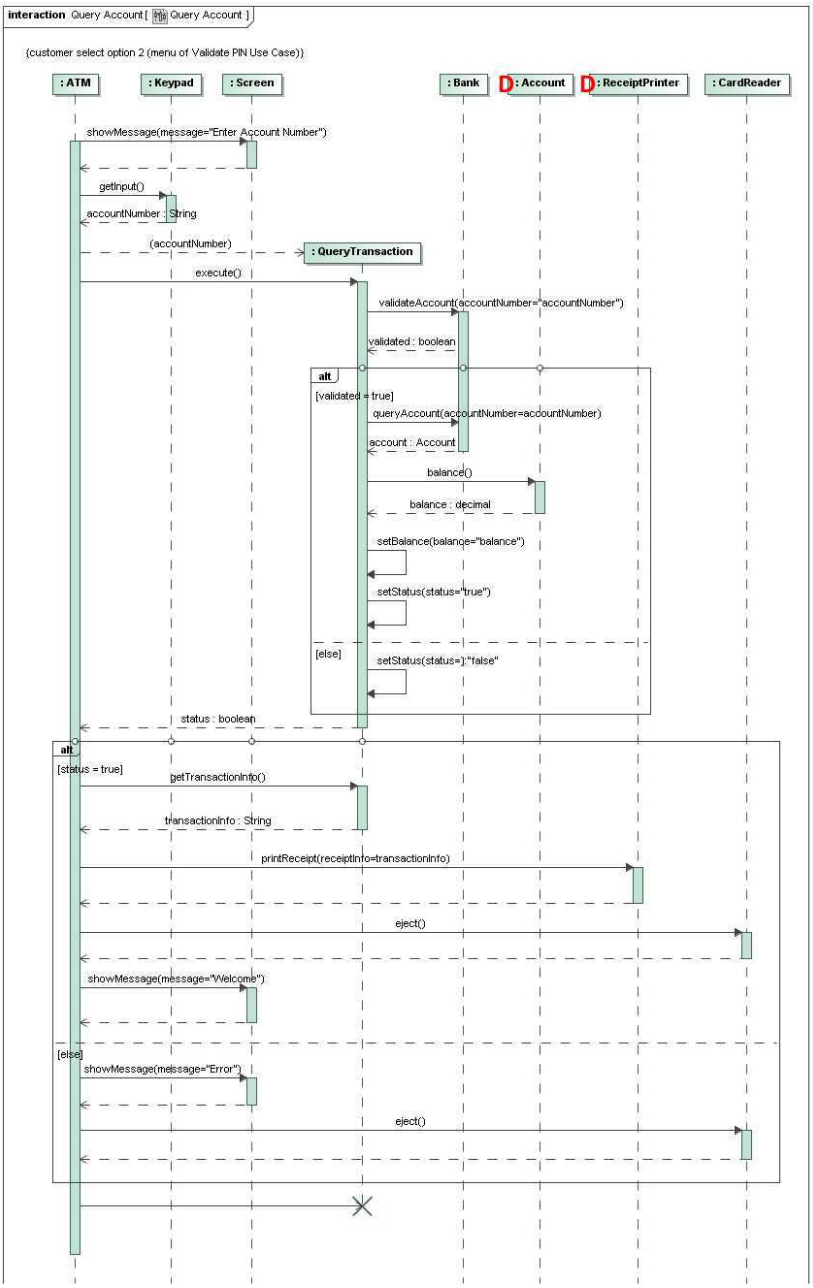

Figura E.9: Defeitos no diagrama de seqüência do caso de uso Query Account.

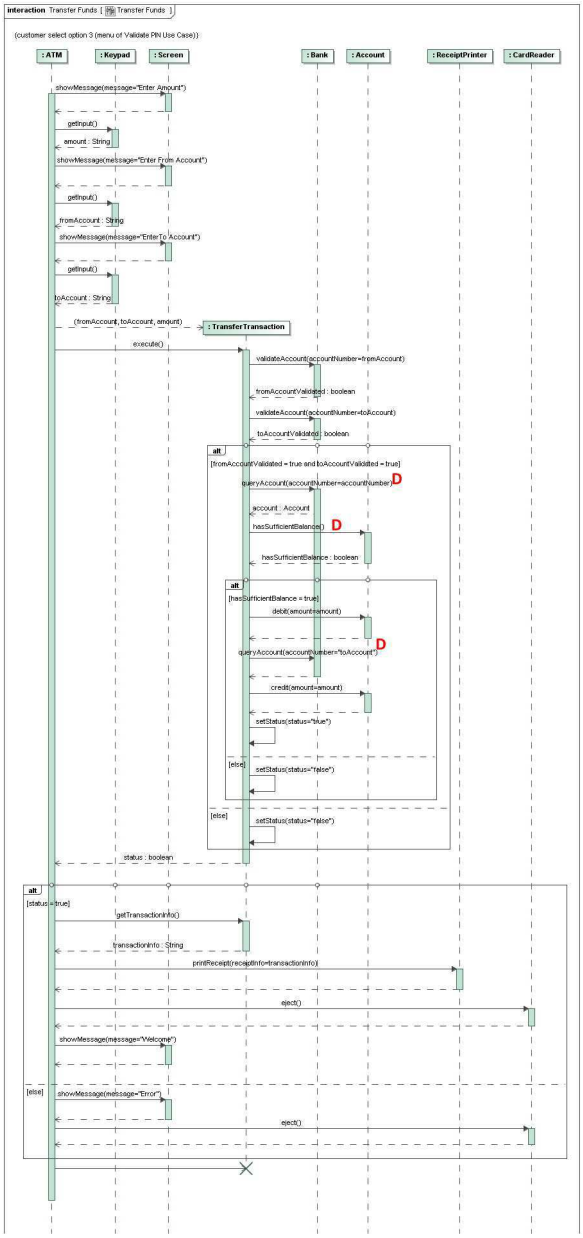

Figura E.10: Defeitos no diagrama de seqüência do caso de uso Transfer Funds.

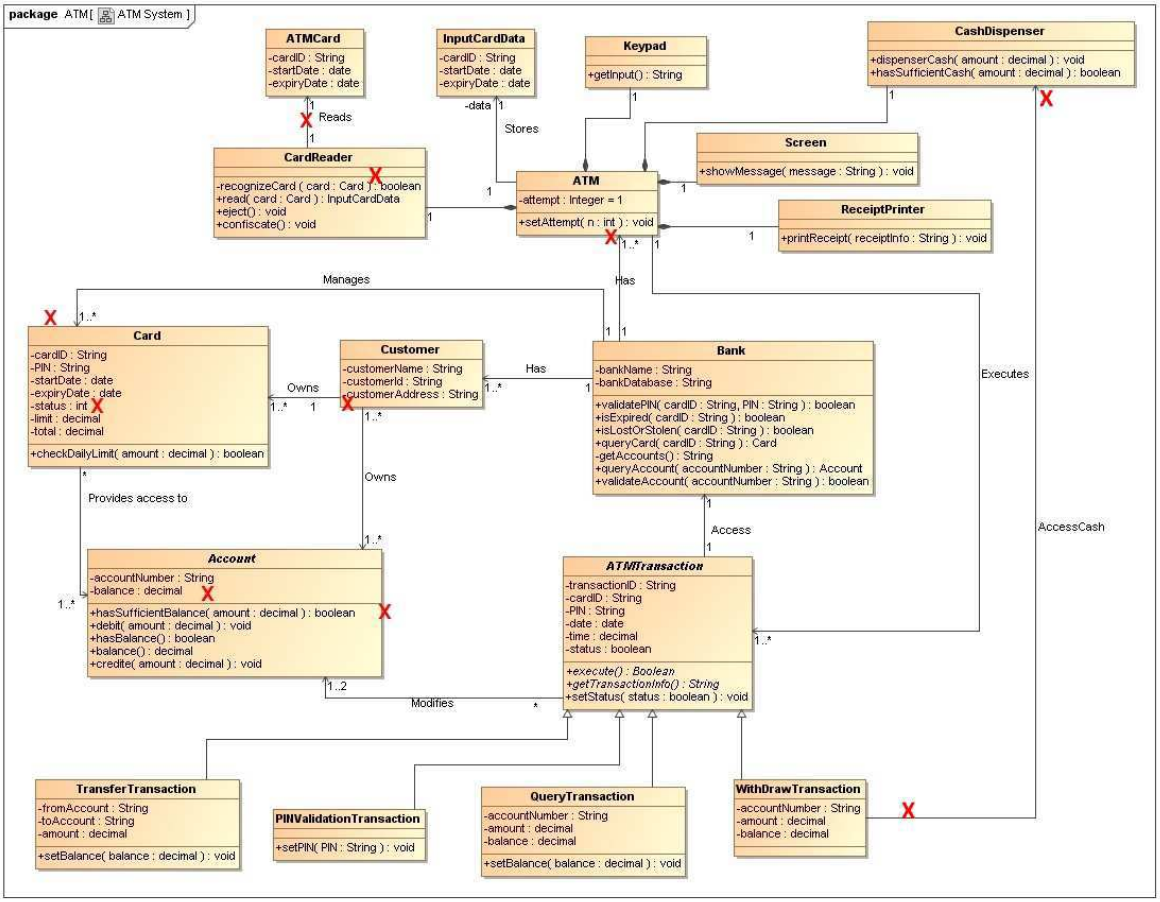

Figura E.11: Defeitos no diagrama de classes do sistema ATM.

E.2 Resultados da Inspeção Guiada Manual

Os casos de teste utilizados para realizar a inspeção guiada manual está presente nas Tabelas D.1, D.2, D.3 e D.4 do Apêndice D. O resultado da execução desta técnica sobre o diagrama de classes e de seqüência para verificar a conformidade destes modelos de projeto com relação à especificação de requisitos pode ser visualizado nas Tabelas E.2 e E.3.

Caso de Teste	**Tipo**	**Defeito**
Inspeção no Diagrama de Classes		
UC 01 - Validate PIN		
Caso de Teste 01	-	OK
Caso de Teste 02	-	OK
Caso de Teste 03	-	OK
Caso de Teste 04	-	OK
Caso de Teste 05	-	OK
Caso de Teste 06	-	OK
Caso de Teste 07	Omissão	[1] Não encontrei a ação de *Cancelar* no diagrama de classes.
UC 02 - Withdraw Funds		
Caso de Teste 08	Omissão	[2] Não há uma ação de selecionar a opção.
Caso de Teste 08	Omissão	[3] Não há uma ação para *WithDraw*.
Caso de Teste 09	-	OK
Caso de Teste 10	-	OK
Caso de Teste 11	-	OK
Caso de Teste 12	Omissão	[4] Ação de *shutdown* do *CashDispenser*.
UC 03 - QueryAccount		
Caso de Teste 13	-	OK
Caso de Teste 14	-	OK
UC 04 - TransferFunds		
Caso de Teste 15	-	OK
Caso de Teste 16	-	OK
Caso de Teste 17	-	OK
Caso de Teste 18	-	OK

Tabela E.2: Defeitos encontrados no diagrama de classes do Sistema ATM.

Caso de Teste	**Tipo**	**Defeito**
Inspeção no Diagrama de Classes		
UC 01 - Validate PIN		
Caso de Teste 01	-	OK
Caso de Teste 02	-	OK
Caso de Teste 03	-	OK
Caso de Teste 04	-	OK
Caso de Teste 05	Omissão	[5] Faltou executar novamente o método *ValidatePIN()*.
Caso de Teste 06	-	OK
Caso de Teste 07	Omissão	[6] Não há a mensagem pra *cancelar* a operação.
UC 02 - Withdraw Funds		
Caso de Teste 08	Omissão Inconsistência Inconsistência Inconsistência	[7] No diagrama de classes não há como selecionar a opção do menu. [8] Especificação não solicita a validação da *account*. [9] A mensagem *queryCard()* não foi prevista na especificação. [10] Especificação não solicita verificar se o caixa tem dinheiro suficiente.
Caso de Teste 09	-	OK
Caso de Teste 10	Fato Incorreto	[11] Mensagem errada: deveria ser *apology*.
Caso de Teste 11	-	[12] Mensagem errada: deveria ser *apology*.
Caso de Teste 12	Fato Incorreto	[13] Mensagem errada: deveria ser *apology* e faltou a mensagem para desligar a maquina.
UC 03 - QueryAccount		
Caso de Teste 13	Inconsistência	[14] Especificação não solicita a validação da *account*.
Caso de Teste 14	-	OK
UC 04 - TransferFunds		
Caso de Teste 15	Fato Incorreto Fato Incorreto	[15] Na mensagem queryAccount deveria ter no parametro *fromAccount*. [16] A mensagem de retorno de *queryAccount(toAccount)* deveria retornar um account, mas não retorna nada.
Caso de Teste 16	-	OK
Caso de Teste 17	-	OK
Caso de Teste 18	Inconsistência	[17] A mensagem deveria ser *Apology* para aparece como um erro.

Tabela E.3: Defeitos encontrados nos diagramas de seqüência do Sistema ATM.

E.3 Resultados de Perspective-Based Reading (PBR)

1. **Passo 1 Localize o Diagrama de Seqüência**

   1.1. Cada objeto tem ao menos uma mensagem enviada ou recebida?

   Sim.

   1.2. Os nomes de cada objeto e de cada mensagem estão definidos?

   [1] Nenhuma linha de vida possui objeto principal nomeado.

   [2] O "card" não foi definido/nomeado (ValidatePIN).

   [3] Account não foi definido.

   [4] Uso indevido de aspas (ex: queryAccount(acount="acount")), pois deveria tratar-se de passagem de objetos, não de strings, nesses casos.

2. **Passo 2 Localize os Diagramas de Seqüência e a Especificação de Requisitos**

   2.1. Todos os objetos do diagrama de seqüência estão relacionados ao domínio descrito na especificação de requisitos?

   Sim.

   2.2. Há alguma inconsistência entre os diagramas de seqüência e a especificação de requisitos?

   Sim.

   [5]. No diagrama, não há menção ao usuário pressionar Cancelar para cancelar a operação como um todo (eject()).

   [6]. A alternativa 2 não está sendo seguida, pois o diagrama mostra apenas que o sistema mostra "Erro" e não uma mensagem de "apology".

   [7]. Tem um "validate number" a mais no diagrama.

   [8]. O diagrama não é suficiente para saber se o requisito 4 foi realmente realizado.

   [9]. No diagrama, de alguma forma o toAccount passou a ser chamado de account apenas, fazendo com que o requisito 3 não batesse exatamente.

3. **Passo 3 Localize os Diagramas de Seqüência e os Diagramas de Caso de Uso**

   3.1. Cada caso de uso do diagrama de caso de uso está implementado em algum diagrama de seqüência?

   Sim.

   3.2. Todos os objetos e mensagens importantes entre os objetos estão presentes nos diagramas de seqüência?

   Sim.

Questões na Perspectiva do Analista

1. **Passo 1 Localize o Diagrama de Classes**

   1.1. O nome de cada classe está definido?

   Sim.

   1.2. A multiplicidade de todas as associações estão definidas?

   [10]. Falta multiplicidade entre CashDispenser e WithDrawTransaction.

2. **Passo 2 Localize o Diagrama de Classes, Especificação de Requisitos e Diagramas de Caso de Uso**

   2.1. Todos os objetos listados na especificação de requisitos estão representados no diagrama de classes?

   Sim.

   2.2. Há algum elemento redundante no diagrama de classes?

   Não.

   2.3. Todas as classes e associações estão definidas?

   Sim.

   2.4. Todas as classes necessárias para criar os casos de uso do diagrama de caso de uso estão definidas no diagrama de classes?

   Sim.

3. **Passo 3 Localize o Diagrama de Classes e os Diagramas de Seqüência**

3.1. Todos os objetos dos diagramas de seqüência estão definidos no diagrama de classes?

Sim.

3.2. Todas as mensagens entre os objetos no diagrama de seqüência correspondente estão definidas como métodos ou atributos na classe correspondente no diagrama de classes?

Não.

[11]. Existem chamadas à getInput que não retorna strings, mas decimal (Keypad - withdral funds).

[12]. O método hasSufficienteBalance às vezes não recebe o parâmetro "amount" (transfer funds).

3.3. A relação entre dois objetos que existem no diagrama de seqüência existe também entre as mesmas classes de objeto no diagrama de classes?

Sim.

3.4. Há algum elemento redundante ou faltando no diagrama de seqüência?

Não.