MARCELO SANTOS ARAUJO

ANÁLISE DA CORRESPONDÊNCIA ENTRE OS MODELOS DE ANÁLISE E DE PROJETO UTILIZANDO CONCEITOS DE ENGENHARIA DIRIGIDA POR MODELOS

Monografia apresentada ao PECE – Programa de Educação Continuada em Engenharia da Escola Politécnica da Universidade de São Paulo como parte dos requisitos para conclusão do curso de MBA em Tecnologia de Software.

São Paulo
2014

MARCELO SANTOS ARAUJO

ANÁLISE DA CORRESPONDÊNCIA ENTRE OS MODELOS DE ANÁLISE E DE PROJETO UTILIZANDO CONCEITOS DE ENGENHARIA DIRIGIDA POR MODELOS

Monografia apresentada ao PECE – Programa de Educação Continuada em Engenharia da Escola Politécnica da Universidade de São Paulo como parte dos requisitos para a conclusão do curso de MBA em Tecnologia de Software.

Área de Concentração: Tecnologia de Software

Orientador: Prof. Dr. Fábio Levy Siqueira

São Paulo
2014

DEDICATÓRIA

Dedico este trabalho ao meu pai, José Nascimento (in-memorian), a minha mãe, Iraci (in-memorian), a minha esposa Daniela, ao meu filho, Filipe Miguel e ao meu irmão, Diego.

AGRADECIMENTOS

Em primeiro lugar agradeço a Deus por tudo o que Ele tem me proporcionado.

A minha esposa Daniela pela compreensão, apoio e incentivo que recebi durante todo esse período.

Ao meu filho Filipe Miguel pelo amor recebido e por me agraciar todos os dias com um belo sorriso.

Ao meu irmão Diego pelas palavras de motivação.

É difícil transformar em palavras o sentimento de respeito e gratidão que tenho ao Professor Dr. Fábio Levy Siqueira, que durante o desenvolvimento desta monografia, sempre demonstrou boa vontade em me ajudar, competência, muita paciência e dedicação.

Aos meus companheiros de trabalho da Eclética Informática e do Departamento de Estradas e Rodagem (DER) pelo o incentivo.

A todos os professores e colegas do curso de MBA – Tecnologia de Software.

A todos que, direta ou indiretamente, contribuíram para a realização deste trabalho.

RESUMO

Um dos objetivos da Engenharia de Software é obter softwares com qualidade, baixo custo e nos prazos estimados. Apesar de existirem processos maduros que auxiliam o desenvolvimento de software, muitos deles propõem atividades que devem ser realizadas manualmente, permitindo a ocorrência de falhas que demandam tempo e dinheiro para corrigi-las. Utilizando conceitos de Engenharia Dirigida por Modelos, este trabalho propõe analisar a correspondência dos modelos de análise e de projeto, verificando se o modelo de projeto é um refinamento válido do modelo de análise. Para isso, são apresentados um metamodelo de correspondência, um conjunto de regras de transformação e um script que executa a transformação dos modelos. Este *script* gera o modelo de correspondência obtendo as informações do que se equivale entre os modelos de análise e de projeto, o que facilita a verificação se o modelo de projeto é refinamento válido do modelo de análise.

Palavras-chave: modelo de análise, modelo de projeto, engenharia dirigida por modelos, transformação de modelos.

ABSTRACT

One of the goals of the Software Engineering is develop software with high quality, low cost, and in the estimated time frame. Although there are processes that help developing software, they propose manually executed activities that may lead to mistakes that demand time and money to fix it. Using concepts of Model-Driven Engineering, this work proposes analyzing the match between the analysis model and the design model, checking if the design model is a valid refinement of the analysis model. It is presented a metamodel for model match, a group of transformation rules, and a script that execute the model transformation. This script generates the model for the model match informing what is equivalent between the analysis models and the design models and generating a new model that help the engineers to check if the design model is a valid refinement of the analysis model.

Keywords: analysis model, design model, model-driven engineering, transformation.

LISTA DE ILUSTRAÇÕES

Pág.

Figura 3.1: Alterada de Kleppe; Warmer e Bast (2003) 24
Figura 4.1: Metamodelo de análise de correspondência de modelos 34
Figura 4.2: Esquema geral da análise de correspondência 35
Figura 4.3: Transformação proposta 36
Figura 4.4: Correspondência entre duas classes 41
Figura 4.5: Correspondência entre dos atributos 42
Figura 4.6: Atributos presentes no Modelo de Análise 42
Figura 4.7: Atributos removidos no Modelo de Projeto 43
Figura 4.8: Associação do Modelo de Análise 43
Figura 4.9: Associação do Modelo de Projeto 44
Figura 4.10: Multiplicidade do Modelo de Análise 44
Figura 4.11: Multiplicidade do Modelo de Projeto 45
Figura 4.12: Classe no Modelo de Análise 45
Figura 4.13: Classes do Modelo de Projeto 46
Figura 4.14: Classes do modelo de análise e de projeto 46
Figura 4.15: Modelo de análise na IDE Eclipse 47
Figura 4.16: Modelo de projeto na IDE Eclipse 47
Figura 4.17: Modelo criado com resultado da correspondência entre os modelos de análise e projeto 48

LISTA DE TABELAS

Pág.

Tabela 2.1: Nível de abstração: Visão abstrata versus visão concreta Génova; Valiente e Marrero (2009) 19

Tabela 4.1: Pontos importantes para duas abordagens MTBE e exemplos individuais 38

LISTA DE ABREVIATURAS E SIGLAS

ATL	*Atlas Transformation Language*
EMF	*Eclipse Modeling Framework*
MDA	*Arquitetura Dirigida por Modelos (Model Driven Architecture)*
MDE	*Engenharia Dirigida por Modelos (Model Driven Engineering)*
MOF	*Meta Object Facility*
MTBE	*Model Transformation By Example*
OMG	*Object Management Group*
PIM	*Platform Independent Model*
PSM	*Platform Specific Model*
QVT	*Query/View/Transformation*
UML	*Linguagem de Modelagem Unificada (Unified Modeling Language)*
UP	*Processo Unificado (Unified Process)*

SUMÁRIO

Pág.

1. INTRODUÇÃO 12

1.1. Objetivo 12

1.2. Justificativa 13

1.3. Escopo 13

1.4. Estrutura do Trabalho 14

2. DESENVOLVIMENTO DE SOFTWARE 15

2.1. Diferenças entre modelo de Análise e Projeto 17

2.2. Modelo de Análise Orientada a Objetos 19

2.3. Modelo de Projeto Orientada a Objetos 20

2.4. Considerações do capítulo 21

3. ENGENHARIA DIRIGIDA POR MODELOS 22

3.1. Modelo 23

3.2. Metamodelo 23

3.3. Metametamodelo 24

3.4. Operação entre modelos 25

3.5. Considerações do capítulo 27

4. ANÁLISE DO REFINAMENTO DO MODELO DE ANÁLISE NO MODELO DE PROJETO 28

4.1. Implementação da Análise com a Operação de Correspondência 28

4.2. Linguagem de Transformação 31

4.3. Metamodelo de Correspondência 32

4.4. Estrutura da análise de correspondência 34

4.5. Método para Criação das Regras de Correspondência 36

4.6. Regras de Correspondência 40

4.7. Exemplo Ilustrativo da análise de correspondência dos Modelos 46

5. CONSIDERAÇÕES FINAIS 50

5.1. Trabalhos Futuros 51

REFERÊNCIAS 52

APÊNDICE A – Sistema de Venda 55

APÊNDICE B – Regras de Correspondência 66

1. INTRODUÇÃO

Com a exigência de softwares de qualidade e com custos menores, percebe-se que é necessário buscar alternativas no campo da Engenharia de Software. Uma possibilidade é a automação de atividades de desenvolvimento de software usando a Engenharia Dirigida por Modelos (MDE). A MDE é uma abordagem em que os modelos são o centro do desenvolvimento de software. Ele busca automatizar o trabalho realizado nas atividades de desenvolvimento de software, por meio dos modelos, pois os modelos ajudam a visualizar, racionar e manipular a complexidade do problema de uma maneira mais simples (MELLOR et al., 2004).

Dentro da área de desenvolvimento de software, escopo deste trabalho é analisar o resultado das atividades de análise e de projeto *(design)*. A atividade de análise busca entender o problema das partes interessadas, enquanto a atividade de projeto busca fornecer uma solução do problema analisado. Os modelos criados nestas atividades devem ser consistentes, já que realizar essa análise manualmente poder ser difícil e custoso. Por isso este trabalho sugere os resultados das atividades de análise e projeto utilizando conceitos da MDE.

1.1. Objetivo

O objetivo deste trabalho é propor uma forma de analisar a correspondência entre os modelos de análise e de projeto utilizando conceitos da Engenharia Dirigida por Modelos.

A análise será realizada utilizando um *script* criado para analisar a correspondência automática entre ambos os modelos, buscando fornecer informações importantes sobre a correspondência dos modelos como, por exemplo, analisar se as classes, os atributos e as associações que foram criados no modelo de projeto possuem correspondência com os elementos definidos no modelo de análise.

Isso ajuda os engenheiros a descobrirem de forma automática se o modelo de projeto é um refinamento válido do modelo de análise.

1.2. Justificativa

Segundo Génova; Valiente e Marrero (2009), mudar o foco de desenvolvimento de software para os modelos é benéfico, pois os modelos estão mais perto do entendimento humano. Desta forma o trabalho com modelos terá um ganho na qualidade do software maior do que trabalhar com linguagens de programação que são mais complexas e que, devido a essa complexidade, aumentam a probabilidade da qualidade do software diminuir. O benefício em trazer o centro de desenvolvimento de software para os modelos é que outras etapas podem ser analisadas com o intuito de automatiza-las, pois com a automação é possível obter ganhos de produtividade e diminuir os custos dos softwares. Por exemplo, em um projeto de desenvolvimento de software, em que os modelos construídos para fornecer a solução do problema (modelo de projeto) não representam realmente o problema compreendido (modelo de análise), a chance do software final não atender as necessidades dos interessados no software é muito grande.

Em outro exemplo, se o modelo de análise e o modelo de projeto forem desenvolvidos por equipes distribuídas será necessário analisar se o modelo de projeto reflete o modelo de análise, ou seja, se ele reflete o problema compreendido. Mas se essa verificação for feita manualmente o prazo e os custos deste projeto poderão ficar comprometidos, pois se ambos os modelos forem complexos a resposta do refinamento pode ser demorada e ainda não ser a correta.

1.3. Escopo

A análise e o projeto podem ser feitas de diversas formas. E uma dessas formas pode ser envolvendo diagramas da UML como, por exemplo, o diagrama de classe e o diagrama de sequência (os quais foram utilizados neste trabalho). Mas como proposta inicial para analisar a correspondência entre os modelos de análise e de

projeto será utilizado apenas o diagrama de classe da UML, ou seja, somente o diagrama de classe será utilizado para analisar se o modelo de projeto é um refinamento válido do modelo de análise. O motivo pela escolha do diagrama de classes foram os seguintes:

- As classes são os blocos de construção mais importantes de qualquer sistema orientado a objetos (BOOCH; RUMBAUGH; JACOBSON, 2006).
- O diagrama de classes pode ser encontrado com maior frequência na modelagem de sistemas orientados a objetos (BOOCH; RUMBAUGH; JACOBSON, 2006).
- O diagrama de classes também é a base para criação de outros diagramas como por exemplo, os diagramas de componentes e de implantação (BOOCH; RUMBAUGH; JACOBSON, 2006).

1.4. Estrutura do Trabalho

O Capítulo 2 DESENVOLVIMENTO DE SOFTWARE: apresenta uma visão geral do processo de desenvolvimento de software, enfatizando a diferença entre o modelo de análise e o modelo de projeto orientados a objetos.

O Capítulo 3 CORRESPONDÊNCIA DE MODELOS: apresenta a Engenharia Dirigida por Modelos (MDE), discutindo os conceitos envolvidos e as operações que podem ser empregradas.

O Capítulo 4 ANÁLISE DO REFINAMENTO DO MODELO DE ANÁLISE NO MODELO DE PROJETO: apresenta a implementação do *script* que realiza a verificação de correspondência entre os modelos de análise e projeto, por meio de regras de transformação de modelos.

O Capítulo 5 CONSIDERAÇÕES FINAIS: descreve os Trabalhos Futuros e a Conclusão.

2. DESENVOLVIMENTO DE SOFTWARE

O emprego de um processo de desenvolvimento de software adequado ajuda a desenvolver software com qualidade, baixo custo e nos prazos estimados. O processo dentro da engenharia de software não é uma prescrição rígida, pelo contrário, é uma abordagem que possibilita as pessoas selecionarem um conjunto apropriado de ações e tarefas (PRESSMAN, 2011). O processo guia os engenheiros no que deve ser feito para atender os objetivos do software por meio de um conjunto de atividades que possuem uma coleção de ações e onde cada ação é definida por um grupo de tarefas.

Pressman (2011) apresenta cinco atividades de um processo genérico: Comunicação, Planejamento, Modelagem, Construção e Emprego. Essas atividades podem ser usadas tanto em projetos de desenvolvimento de softwares pequenos e simples a até projetos de aplicações grandes e complexas. As atividades do processo genérico são organizadas em fluxo descrevendo como cada ação e tarefa acontecem dentro de cada atividade (PRESSMAN, 2011):

- **Fluxo de processo linear**: neste fluxo as atividades são executadas em sequência, ou seja, só será executada uma nova atividade após o término da atual. Após avançar para a próxima atividade fica difícil de realizar correções na atividade anterior devido a complexidade. Este fluxo é similar ao modelo de desenvolvimento cascata.
- **Fluxo de processo iterativo**: neste fluxo é possível repetir uma ou mais vezes a mesma atividade antes de prosseguir para a etapa seguinte.
- **Fluxo de processo evolucionário**: neste fluxo as atividades são executadas de forma circular, em que cada volta pelas cinco atividades é produzida uma versão mais completa do software. Este fluxo é similar ao modelo de processo iterativo e incremental.
- **Fluxo de processo paralelo**: neste fluxo é executado uma ou mais atividades em paralelo com as outras atividades.

Neste trabalho é utilizado como base para o desenvolvimento de software o Processo Unificado (UP) que é um processo evolucionário, seguindo o modelo iterativo e incremental. O UP tenta aproveitar os melhores recursos e características dos processos de software tradicionais, mas se diferenciando pela a abordagem de desenvolvimento guiado por miniprojetos com duração fixa, em que cada entrega é fornecida um sistema parcial executável, testável e integrável (LARMAN, 2007). Outro ponto importante é que o UP considera o paradigma *Orientado a Objetos*, usando a Linguagem de Modelagem Unificada *(UML, do inglês – Unified Modeling Language)* como linguagem de modelagem. As iterações do UP são organizadas em quatro fases (concepção, elaboração, construção e transição), enquanto que as atividades de trabalho são descritas como disciplinas (LARMAN, 2007). As disciplinas são divididas da seguinte forma (KRUCHTEN, 2004), (LARMAN, 2007):

- **Modelagem de negócios:** disciplina que busca desenvolver o modelo de domínio com objetivo de visualizar conceitos importantes do domínio.
- **Requisitos:** disciplina que tem como objetivo criar o modelo de caso de uso e obter os requisitos funcionais e não funcionais do sistema.
- **Análise e Projeto:** disciplina que cria os artefatos para projetar os objetos de software.
- **Implementação:** disciplina que cria o modelo de implementação, transformando o modelo de projeto em código para obter um sistema de software.
- **Teste:** disciplina que cria o modelo de teste para descrever como os testes de integração e de sistemas verificarão os componentes de software criados a partir do modelo de implementação.
- **Desdobramento ou implantação:** disciplina que produz os lançamentos dos produtos e entrega o software para seus interessados finais.
- **Gestão de configuração e mudança:** disciplina que controla as mudanças feitas nos artefatos de um projeto mantendo integridade dos artefatos alterados.
- **Gerenciamento de projeto:** disciplina que fornece diretrizes para planejar, montar equipe, executar e monitorar os projetos.

- **Ambiente:** disciplina que tem como objetivo dar o suporte necessário para a organização do desenvolvimento com processos e ferramentas.

Segundo Larman (2007), os artefatos são tratados no UP como qualquer produto de trabalho como, por exemplo: código, esquemas de banco de dados, documentos em texto, diagramas, modelos, etc. Especificamente em relação aos modelos de análise e projeto orientado a objetos, eles são criados a partir das disciplinas de análise e de projeto.

2.1. Diferenças entre modelo de Análise e Projeto

De uma forma geral pode-se dizer que o modelo de análise visa compreender um problema, enquanto o modelo de projeto visa desenvolver uma solução para o problema analisado. Mas, segundo Génova; Valiente e Marrero (2009), esta definição é bastante simples, pois diferentes modelos de análise podem ser criados e nem todos esses modelos visam compreender um determinado problema. Os modelos de análise e de projeto são usados como fases preliminares antes da implementação do software, ou seja, antes que um programador comece a desenvolver o modelo de implementação (criar o código propriamente dito). Porém é difícil dizer em que momento termina o modelo de análise e começa o modelo de projeto. Por meio desta dificuldade a comunidade da engenharia de software não tem um entendimento aceito por unanimidade de como diferenciar o modelo de análise do modelo de projeto (GÉNOVA; VALIENTE; MARRERO, 2009); o que existe é uma forma de estruturar e verificar esta diferença. Kent (2002) apresenta a modelagem de dimensões, na qual é necessário categorizar as perspectivas dos modelos – como é realizado dentro da arquitetura dirigida por modelo *(*MDA, do inglês – Model-Driven Architecture*),* com PIMs *(*Modelo Independente de Plataforma, do inglês – Platform Independent Model*)* e PSMs *(*Modelo Especifica de Plataforma, do inglês – Platform Specific Model*).* Ou seja, é necessário identificar os critérios ortogonais que podem ser utilizados para diferenciar entre uma e outra perspectiva dos modelos.

Génova; Valiente e Marrero (2009) propõem três dimensões a serem consideradas para diferenciar o modelo de análise do modelo de projeto. As três dimensões são:

- **Realidade representada pelo modelo:** modelos podem ser usados como modelos de sistema ou modelos de domínio. Um modelo de sistema representa um sistema ou uma parte de um sistema, enquanto um modelo de domínio representa uma parte da realidade que afeta e é afetada pelo sistema de software. A ideia desta dimensão é considerar a diferença entre esses dois tipos de modelos. O modelo de sistema pode incluir conceitos comuns do domínio da aplicação, mesmo que a correspondência não seja perfeita devido ao objetivo de um sistema de software ir além de conceitos do domínio da aplicação.
- **Propósito do modelo:** modelos podem ser usados para a especificação ou para a descrição. Um modelo usado para especificação trata de algo que deve ser construído e um modelo usado para a descrição trata de algo que já existe. Dentro do campo da Engenharia de Software a especificação é conhecida como engenharia para frente e a descrição como engenharia reversa. O que é analisado é o objetivo das duas engenharias, pois a meta de ambas é oposta: enquanto a engenharia para frente visa obter a solução do problema, a engenharia reversa tem como objetivo obter um modelo para poder compreender o sistema existente ou o domínio do problema expresso no sistema.
- **Nível de abstração:** modelos podem tratar de diferentes níveis de abstração. Nesta dimensão em geral diz-se que o modelo de análise é abstrato, pois é um modelo conceitual que tem como objetivo entender o problema. Por outro lado, o modelo de projeto é representado pelo nível concreto, pois trata de obter a solução do problema analisado (próxima da implementação).

Essas dimensões são ortogonais, portanto os modelos de especificação e/ou descrição (nível propósito) podem ser tanto abstratos como concretos. Da mesma forma, um modelo de domínio (nível realidade), seja ele uma especificação ou descrição, também pode ser abstrato ou concreto. Assim os níveis ficam representados conforme a tabela 2.1.

Tabela 2.1: Nível de abstração: Visão abstrata versus visão concreta Génova; Valiente e Marrero (2009).

Especificação	Visão abstrata	Modelo abstrato da especificação do domínio	Modelo abstrato da especificação do sistema
	Visão concreta	Modelo concreto da especificação do domínio	Modelo concreto da especificação do sistema
Descrição	Visão abstrata	Modelo abstrato da descrição do domínio	Modelo abstrato da descrição do sistema
	Visão concreta	Modelo concreto da descrição do domínio	Modelo concreto da descrição do sistema
		Domínio	Sistema

2.2. Modelo de Análise Orientada a Objetos

Dentro do processo de desenvolvimento de software, o termo modelo de análise pode ser empregado em diferentes dimensões como dito anteriormente. Para este trabalho o termo modelo de análise é usado representado o modelo abstrato da especificação do sistema. Ou seja, considera-se, seguindo o Génova; Valiente e Marrero (2009), que o modelo de análise é abstrato tendo como foco representar um sistema ou parte de um sistema – um modelo de sistema – que é um modelo conceitual que tem como objetivo entender o problema.

Os modelos criados na atividade de análise devem ser validados e verificados. A validação deve assegurar que as necessidades das partes interessadas (*stakeholders*) estão sendo atendidas, validando se o que está sendo construído está correto, consistente, realista e sem ambiguidades, enquanto que a verificação deve assegurar que os modelos construídos estão em conformidade com os requisitos levantados, verificando a exatidão de cada modelo separado e a consistência entre os modelos (BEZERRA, 2007).

No UP, uns dos componentes desse modelo são os diagramas de classes, o qual representam atributos, relacionamentos e classes; diagrama de interação (diagrama de sequência ou de colaboração) o qual define o comportamento geral do software; e o diagrama de pacotes, que reflete a organização das classes.

2.3. Modelo de Projeto Orientada a Objetos

O modelo de Projeto orientado a objetos, o qual será chamado apenas como modelo de projeto, é o refinamento do modelo de análise. Ele trata da definição da solução conceitual que satisfaz os requisitos (LARMAN, 2007). Portanto, seguindo Génova; Valiente e Marrero (2009), ele é aqui considerado como um modelo concreto da especificação do sistema, que tem como objetivo fornecer uma solução para o problema.

Na atividade de projeto existem alguns aspectos que devem ser considerados como, por exemplo, a arquitetura do sistema, o padrão de interface gráfica, a linguagem de programação, o sistema gerenciador de banco de dados, etc. Existem duas atividades principais de projeto (BEZERRA, 2007):

- **Projeto de Arquitetura ou Projeto de alto nível**: consiste em distribuir as classes de objetos relacionados do sistema em subsistemas e seus componentes.
- **Projeto detalhado ou Projeto de baixo nível**: consiste em modelar as colaborações entre objetos de cada módulo com o objetivo de realizar as funcionalidades do módulo.

Como o intuito do modelo de projeto é descrever uma solução computacional, podem-se usar modelos dinâmicos e estáticos da UML (LARMAN, 2007), para representar esta solução computacional.

2.4. Considerações do capítulo

Como o objetivo deste trabalho é a analisar se o modelo de projeto é um refinamento válido do modelo de análise, o Processo Unificado (UP) se encaixa muito bem, pois o UP considera o paradigma orientado a objetos, utilizando a Linguagem de Modelagem Unificada (UML). A UML neste trabalho será a base para representar os modelos de análise e de projeto, limitados ao diagrama de classes.

3. ENGENHARIA DIRIGIDA POR MODELOS

Mesmo com avanços dentro da área da engenharia de software, os problemas de desenvolvimento de software ainda persistem e continuam aumentando cada vez mais devido a complexidade dos artefatos de software (LUCRÉDIO, 2009). Justamente por conta desta complexidade, está acontecendo uma transição para métodos automatizados por meio da noção de modelagem, metamodelo (Lopes et al, 2006) e transformação de modelos (BÉZIVIN, 2005), com o intuito de apoiar o desenvolvimento, manutenção e evolução de sistemas de software.
A engenharia dirigida por modelos (MDE, do inglês *Model Driven Engineering*) é uma abordagem de desenvolvimento de software que cria modelos que produzem elementos de um sistema de software (SCHMIDT, 2006). Segundo Favre (2005), existem pesquisas sobre MDE no interesse de que modelos possam ser capazes de preencher a lacuna entre palavras e códigos, ou seja, as partes interessadas usam palavras para expressar seus problemas e necessidades, e o papel dos engenheiros de software é transformar essas palavras em códigos que auxiliem as partes interessadas com seus problemas e necessidades.

Realizando uma comparação entre o processo de desenvolvimento de software tradicional e o ciclo de desenvolvimento dentro da abordagem MDE, as principais diferenças são os elementos criados (que na verdade são os modelos) e a automatização ou semi-automatização na transição de um modelo para outro (avanço de uma etapa para outra dentro do ciclo de vida de desenvolvimento de software tradicional).

A seguir são apresentadas as definições dos principais conceitos de Engenharia Dirigida por Modelos e algumas das operações que podem ser executadas. Entre essas operações está a operação de correspondência, que tem como objetivo verificar os modelos de entrada e retornar como resultado um novo modelo com as informações da correspondência.

3.1. Modelo

Segundo Favre (2005), a noção de modelo por ser muito ampla e gera muitos debates na área da Engenharia Dirigida por Modelos. Dentro desta noção bem ampla, Kleppe; Warmer e Bast (2003), define modelos como "*uma descrição de um sistema ou parte de um sistema expressa em uma linguagem que respeita o formato preciso (uma sintaxe) e um significado (uma semântica), o qual é adequado para a interpretação automatizada de um computador*". Um modelo também é uma simplificação de um problema ou necessidade no qual é possível visualizar, raciocinar e manipular diminuindo a complexidade do problema ou necessidade estudada (MELLOR et al., 2004). Na definição apresentada por Mellor et al. (2004) fica evidente o benefício que um modelo apresenta no campo da engenharia de software, pois o modelo deve ser capaz de responder perguntas no lugar do sistema atual.

3.2. Metamodelo

Um metamodelo é um modelo de um modelo que fornece a base para a criação de um modelo (FAVRE, 2005). Para que ocorra esta criação, o metamodelo define a estrutura, a semântica e as restrições para um conjunto de modelos, formando uma família de modelos que unidos compartilham uma sintaxe e semântica (MELLOR et al., 2004) (SHAN; ZHU, 2009). O metamodelo expressa uma definição precisa sobre as regras necessárias para criar e instanciar os modelos. Ou seja, os elementos presentes dentro dos modelos como, por exemplo, classes, atributos e associação foram definidos pelo metamodelo. Cada elemento que está contido no metamodelo é a principal forma para simplificar a comunicação sobre os modelos (MELLOR et al., 2004).

Um exemplo de metamodelo definindo um modelo é exemplificado na figura 3.1 onde são apresentadas duas classes (Pessoa e Empresa) que representam o modelo e mais duas classes (UML Classe e UML Atributo) que representam elementos do metamodelo. Tanto a classe Pessoa com a classe Empresa são instâncias da metaclasse UML Classe. Essa instanciação ocorre por meio da relação

“Instancia de” que é a responsável pela ligação entre o modelo e o metamodelo. Da mesma forma os atributos da classe Empresa (Nome e CNPJ) são instâncias da metaclasse UML Atributo.

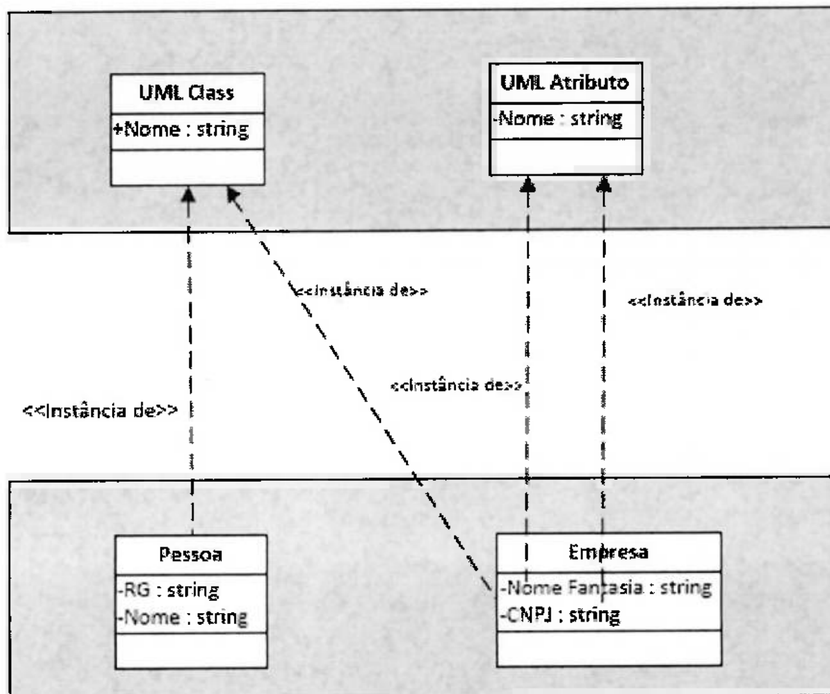


Figura 3.1: Alterada de Kleppe; Warmer e Bast (2003)

Assim como nos modelos os elementos de um metamodelo também são instâncias de elementos de um metametamodelo. O metametamodelo tem como maior responsabilidade definir a linguagem para especificar um metamodelo (OMG, 2011).

3.3. Metametamodelo

Segundo Bézivin (2006), com o crescente número de metamodelos surgiu a necessidade de uma estrutura que pudesse integra-los: os metametamodelos. Com isso, os metamodelos seguem a estrutura de representação definida por um metametamodelo. Da mesma forma como a UML é utilizada como uma linguagem de especificação de modelos específicos (diagramas estáticos e dinâmicos), o MOF (*Meta Object Facility*) é quem define a linguagem de especificação do metamodelo da UML. Ele foi desenvolvido pela OMG *(Object Management Group)*, sendo uma

linguagem abstrata e um framework para especificação, construção e gerenciamento de metamodelos independentes de tecnologia de implementação (DOS SANTOS; DE BARROS; FONSECA, 2003).

3.4. Operação entre modelos

Normalmente ao a adotar o desenvolvimento baseado em modelos, esses modelos são construídos por equipes distribuídas, onde cada equipe trabalha com uma visão parcial do sistema (BRUNET et al., 2006). Para auxiliar na consistência desses modelos, Bernstein (2003) e Brunet et al. (2006) falam a respeito dos operadores de modelos. Esses operadores têm como objetivo controlar as relações dos diversos modelos, como apresentados a seguir:

- **Fusão *(Merge)*:** usa dois modelos como, por exemplo, A e B e por meio de um mapeamento entre os dois modelos se obtém a união de A e B tendo como resultado um modelo C. Esta operação de fusão retorna uma cópia de todos os objetos dos modelos de entrada, unificando os objetos dos modelos de entrada idênticos, em um único objeto de saída (BERNSTEIN, 2003).

- **Correspondência *(Match)*:** usa dois modelos como entrada e retorna um mapeamento entre eles. O mapeamento é identificado por meio de combinações dos modelos. Esta combinação é definida com base em algum dado externo de igualdade e semelhança, como por exemplo, dois objetos que podem ser baseados por seus identificadores ou nomes (BERNSTEIN, 2003).

- **Diferença *(Diff)*:** usa o resultado do mapeamento de outra operação, como por exemplo, a operação correspondência, para retornar os objetos que não foram referenciados no mapeamento entre os modelos mapeados (BERNSTEIN, 2003).

- **Composição *(Compose):*** usa um mapeamento entre os modelos A e B e outro entre B e C, e retorna um mapeamento entre A e C (BERNSTEIN, 2003).

- **Aplicar *(Apply):*** usa um modelo e uma função *f* arbitrária como entrada e aplica *f* para cada objeto do modelo. Em muitos casos modifica determinadas propriedades e relações de cada objeto para reduzir a navegação de um objeto no modelo (BERNSTEIN, 2003).

- **Copiar *(Copy):*** usa um modelo como entrada e retorna uma cópia desse modelo. O modelo retornado contém todas as relações do modelo de entrada, incluindo aqueles que conectam seus objetos para objetos fora do modelo (BERNSTEIN, 2003).

- **Transformação *(ModelGen):*** a transformação usa um modelo A como entrada e retorna um novo modelo B baseado em A e um mapeamento entre A e B. Geralmente as aplicações de gerenciamento de modelos envolvem a geração de um modelo-alvo por meio de um modelo-fonte definidos por um metamodelo, que de acordo com o mapeamento forma um conjunto de regras de mapeamento (BERNSTEIN, 2003).

- **Enumerar *(Enumerate)*:** usa um modelo como entrada e retorna um "cursor" como saída. O próximo operador quando aplicado a um cursor devolve um objeto (BERNSTEIN, 2003).

- **Dividir *(Split)*:** produz uma partição do modelo. Este operador pode ser visto como o inverso do operador fusão *(Merge)*, pois ele divide os objetos dos modelos (BRUNET et al., 2006).

- **Fatiar *(Slice)*:** produz uma projeção ou uma visão parcial de um modelo com base em um critério, em que o modelo resultante contém todas as peças do modelo de entrada que satisfaz o critério e assim pode ser usado para construir aspecto do modelo de entrada (BRUNET et al., 2006).

- **Verificar Propriedade *(Check-Property)*:** verifica as propriedades de um modelo retornando um valor booleano (verdadeiro ou falso) como resultado desta verificação. As propriedades verificadas nesta operação estão relacionadas ao comportamento e estrutura dos modelos (BRUNET et al., 2006). No comportamento são analisadas as operações, precondições e pós-condições de um modelo e a estrutura analisa as classes, associações, atributos e invariantes (restrições estruturais) do modelo (GOGOLLA; BÜTTNER; RICHTERS, 2007).

- **É consistente (Is-consistent):** verifica se existe consistência entre os modelos (BRUNET et al., 2006). Desta forma este operador analisa o que é inconsistente antes que outras operações possam ser executadas (SABETZADEH; EASTERBROOK, 2005).

3.5. Considerações do capítulo

Dentre as operações entre modelos, para analisar o refinamento do modelo de análise no modelo de projeto é possível utilizar os operadores *é consistente, correspondência e verificar propriedade.* No capítulo seguinte será analisado sobre qual operador é mais adequado para analisar a correspondência entre os modelos de análise e de projeto.

4. ANÁLISE DO REFINAMENTO DO MODELO DE ANÁLISE NO MODELO DE PROJETO

A análise do refinamento do modelo de análise no modelo de projeto foi motivada pelo fato de existir projetos nos quais os modelos são extensos, complexos e construídos por equipes distribuídas. Nesses casos pode ser difícil afirmar se o modelo de projeto é um refinamento válido do modelo de análise. Por causa desta dificuldade este trabalho tem como proposta analisar a correspondência entre os modelos de análise e de projeto. Esta análise será realizada automaticamente utilizando um script desenvolvido neste trabalho, buscando oferecer informações sobre o que se corresponde entre os modelos analisados. Existem algumas vantagens em realizar a análise de correspondência automaticamente. Entre essas vantagens se destaca o ganho de tempo e custo em analisar a correspondência automaticamente. De acordo com Brunet at al. (2006) modelos construídos por equipes distribuídas, onde cada equipe pode ter utilizado diferentes vocabulários, ou aplicado um vocabulário compartilhado de forma inconsistente, a automatização por meio de regras de correspondência irá trazer não só um ganho em tempo e custo, mas também um direcionamento para os pontos que estão inconsistentes no modelo de forma eficiente e clara.

4.1. Implementação da Análise com a Operação de Correspondência

Existem diversas operações que podem ser realizadas nos modelos, como foram discutidos na seção 3.2. Dentre essas operações existem as operações correspondência (*match*), é consistente (*is-consistent*) e verificar propriedade (*check-property)*, que podem ser utilizadas para analisar a correspondência entre os modelos de análise e projeto. Porém para utilizar o operador *"é consistente"* é necessário um mapeamento entre os modelos, pois este operador analisa se os modelos estão consistentes antes que outro operador possa ser executado. Ou seja, como este operador necessita de um mapeamento o que é justamente o que faz o operador "correspondência". Portanto esse operador foi desconsiderado. Em relação ao operador *"verificar propriedade"*, ele foi descartado pois o problema deste

trabalho é analisar um refinamento de um modelo em outro e não uma propriedade isolada de um modelo. Dessa forma, será utilizada a operação *"correspondência"* como base para analisar o refinamento entre os modelos.

A operação correspondência também é conhecida como mapeamento (BRUNET et al., 2006) e tem como entrada dois ou mais modelos, retornando um mapeamento entre eles. O mapeamento identifica as combinações dos objetos nos modelos com base na definição da igualdade e semelhança. Em alguns casos a definição é bem simples, como por exemplo, verificar se os objetos são iguais por meio de seus identificadores ou nomes, mas em outros casos a identificação não é tão trivial. Bernstein (2003) apresenta duas definições para a verificação da operação de correspondência entre os modelos:

- **Correspondência Simples:** a operação de correspondência simples se baseia em uma simples definição de igualdade, em que o mapeamento entre os objetos é baseado na igualdade de seus identificadores ou nomes dos objetos. Este operador é utilizado quando uma simples definição de igualdade é capaz de produzir um mapeamento preciso (BERNSTEIN, 2003).
- **Correspondência Complexa:** é baseado em definições complexas de igualdade. Mesmo não precisando definir a expressão de mapeamentos entre os objetos, deve diferenciar os conjuntos de objetos que são iguais daqueles que são apenas semelhantes. A semelhança neste caso quer dizer que os objetos têm relacionamentos, mas que não são idênticos (BERNSTEIN, 2003). Para ajudar o mapeamento entre os modelos, a operação de correspondência pode utilizar informações de predecessores comuns, restrições de domínio, correspondência de vocabulário e outros, para encontrar um melhor relacionamento entre os modelos candidatos (BRUNET et al., 2006). Na verdade a operação de correspondência complexa não é um algoritmo que retorna um mapeamento, mas sim um ambiente de projeto com o intuito de ajudar os projetistas a encontrar o melhor mapeamento entre os modelos. Esta operação tem gerado grandes benefícios em diversos campos da tecnologia como: isomorfismo de grafos para identificar a similaridade estrutural de grandes modelos, processamento de linguagem natural para identificar semelhança de nomes ou analisar documentos de texto de um

modelo, glossários de domínio específico, aprendizado de máquina e mineração de dados usando semelhança de instâncias de dados para inferir a igualdade de objetos do modelo (BERNSTEIN, 2003).

Segundo Rahm e Bernstein (2001), para a realização da correspondência entre os modelos é importante considerar alguns critérios de classificação:

- **Esquema versus Instância:** abordagem de correspondência pode considerar os dados do modelo (ou seja, o conteúdo do modelo) ou apenas informações do metamodelo.
- **Elemento versus Estrutura da Correspondência:** a correspondência pode ser realizada por elementos individuais do metamodelo, como atributos, ou combinações de elementos, tais como estruturas complexas do metamodelo.
- **Linguagem versus Restrição:** uma correspondência pode usar uma abordagem linguística (por exemplo, com base em nomes e descrições textuais de elementos do metamodelo) ou uma abordagem baseada em restrições (por exemplo, baseada em chaves e associações).
- **Cardinalidade Correspondentes:** o resultado global da coincidência pode relacionar um ou mais elementos de um metamodelo a um ou mais elementos de outro metamodelo, obtendo-se quatro casos: 1:1, 1:N, N:1, N:N. Além disso, cada elemento do mapeamento pode se relacionar com um ou mais elementos dos metamodelos e pode haver diferentes associações de correspondência no nível do modelo.
- **Informações Auxiliares:** A maioria das correspondências não conta apenas com metamodelo, mas também com informações auxiliares, tais como dicionários dados, decisões anteriores de correspondência e entrada do usuário.

Para o desenvolvimento deste trabalho é necessário a operação de correspondência complexa, uma vez que a operação de refinamento altera a semântica dos elementos do modelo. Para executa-la é necessário de um ambiente de projeto para encontrar um mapeamento entre os modelos. Por este motivo foi utilizado o *framework* de modelagem do Eclipse *(EMF, do inglês - Eclipse Modeling Framework)*

pela facilidade de trabalhar com modelos UML. Dentro do framework de modelagem do eclipse (EMF), foi analisado o plugin *EMF Compare* (EMF Compare, 2014) que implementa a correspondência entre os modelos. Porém não foi possível usá-lo neste trabalho simplesmente por não permitir definir correspondências complexas (ele trabalha apenas com correspondência simples).

Buscando alternativas para implementar a correspondência (pois não foram encontradas outras ferramentas que realizassem a correspondência complexa) a solução escolhida foi a utilização de outro operador: a transformação de modelos. Utilizando esse operador, é possível implementar a correspondência complexa por meio de regras de transformação. Ou seja, por meio da transformação que é implementada por uma linguagem de transformação como, por exemplo, QVTo (que foi utilizada neste trabalho) é possível realizar a correspondência complexa de acordo com a regras definidas. Seguindo Kleppe; Warmer e Bast (2003), a transformação tem como conceito gerar regras que em conjunto transformam o modelo de origem em um ou mais modelos de destino, ou seja, por meio de um modelo é criado um ou vários novos modelos.

Várias abordagens de transformação de modelo têm sido desenvolvidas na última década. Na próxima seção serão discutidas essas linguagens de transformação e o motivo no qual levou a escolha do QVTo.

4.2. Linguagem de Transformação

Diversas linguagens de transformação de modelos foram desenvolvidas ao longo dos anos, dentre elas QVT (Query*/View/Transformation)* da OMG (OMG, 2009) e ATL *(Atlas Transformation Language)* da Universidade de Nantes – França (ATLAS, 2006).

O ATL é uma linguagem híbrida que aceita tanto construções declarativas e imperativas. As construções declarativas são baseadas em regras de correspondência (*matched rule*), em que se estabelecem ligações entre os padrões de origem e padrões de destino. Já a construção imperativa possui duas formas

conhecidas como regras invocadas *(called rules)* e bloco de ações *(action blocks)*, (ATLAS, 2006).

O QVT é uma linguagem de transformação de modelos, que possui uma especificação híbrida, assim como ATL. As construções em QVT são declarativas e imperativas, é formada por três sub-linguagens de transformação de modelos que atuam em qualquer metamodelo que esteja em conformidade com metametamodelo MOF. Duas linguagens possuem transformações bidirecionais e são declarativas: núcleo *(core)* e relações *(relations)*. A terceira é imperativa e se chama mapeamentos operacionais *(operational)*, permitindo apenas transformações unidirecionais (OMG, 2009).

Para o desenvolvimento deste trabalho foi usada a linguagem de transformação QVT por ser o padrão adotado pela OMG. Dentre os tipos de linguagens que podem ser utilizadas em QVT, foi usada a linguagem QVT Operational (QVTo), pois ela é suficiente para realizar a correspondência entre os modelos e considerou-se mais fácil usar uma linguagem imperativa (devido a uma maior familiaridade do autor com este paradigma).

4.3. Metamodelo de Correspondência

Dentro da Engenharia Dirigida por Modelos (MDE), os metamodelos são cruciais para que transformações de modelos possam acontecer (OMG, 2009). Neste trabalho o metamodelo proposto visa analisar a correspondência entre os modelos de análise e o modelo de projeto, analisando somente as classes, atributos e associações - considerando que esta é uma proposta inicial. Na atividade de análise não será analisado a correspondência entre os métodos, pois foi adotado que no modelo de análise não seriam criados os métodos candidatos, sendo assim, os métodos das classes são definidos durante a atividade de projeto – o que depende do processo seguido.

Para representar os modelos de análise e de projeto é utilizado o SimpleUML (OMG, 2009), que na verdade é um metamodelo simplificado do metamodelo UML. Como é

uma proposta inicial, o metamodelo SimpleUML atende as necessidades e é muito mais simples que o complexo metamodelo da UML. Os elementos definidos neste metamodelo são apenas a classe, atributo e associação.

O metamodelo construído para realizar a análise de correspondência do modelo de análise com o modelo de projeto é apresentado na figura 4.1. Esse metamodelo possui os seguintes elementos para realizar a análise de correspondência dos modelos:

1) **Resultado:** é a generalização dos elementos da correspondência. Esta classe possui os seguintes atributos:
   - **nome:** representa o nome dos elementos do modelo (classe, atributo e associação) que foi analisado.
   - **eRefinamento:** booleano que informa se o elemento analisado é um refinamento.
   - **estaAnalise:** booleano que informa se a classe, atributo ou associação está presente no modelo de análise.
   - **estaProjeto:** booleano que informa se a classe, atributo ou associação está presente no modelo de projeto.
2) **Modelo:** elemento que representa o modelo resultante da análise de correspondência.
3) **Classe:** elemento que representa uma classe a ser analisada a correspondência.
4) **Atributo:** elemento que representa um atributo a ser analisado a correspondência.
5) **Associação:** elemento que representa uma associação a ser analisada a correspondência.

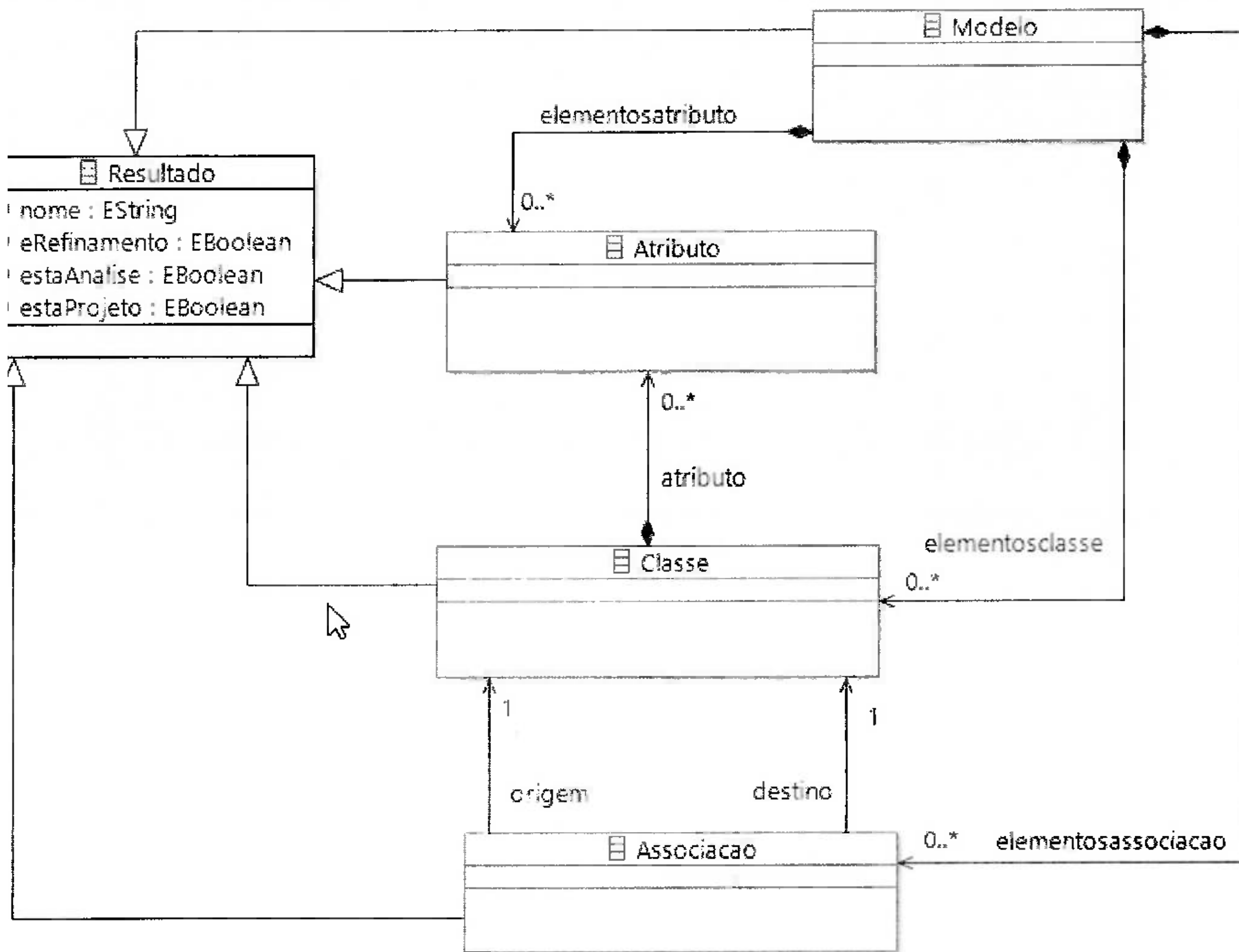


Figura 4.1: Metamodelo de análise de correspondência de modelos

4.4. Estrutura da análise de correspondência

O *script* criado para a análise de correspondência dos modelos de análise e de projeto foi implementado por meio do EMF *(Eclipse Modeling Framework)* que é um framework que facilita a criação de modelos baseados em simples definições. O *script* de análise de correspondência de modelos foi desenvolvido na forma de plug-in para a IDE Eclipse, o qual recebeu o nome de Modelo de Correspondência.

A figura 4.2 ilustra o esquema geral de análise de correspondência dos modelos e seu respectivo resultado. A análise de correspondência tem como entrada o modelo de análise e de projeto. A verificação da correspondência é realizada de acordo com algumas regras (que serão apresentadas na seção 4.3) que são implementadas por

meio da transformação de modelos. Cada regra cria um trecho de um novo modelo e ao fim da transformação este novo modelo tem uma resposta do que é idêntico e diferente entre os modelos. Esta resposta se dá por meio de cada elemento existente nos modelos verificados como, por exemplo: classe, atributo e associação. Estes elementos verificados retornarão um valor booleano, ou seja, se a correspondência existir o resultado será verdadeiro (*true*), mas se a correspondência não existir o valor será falso (*false*).

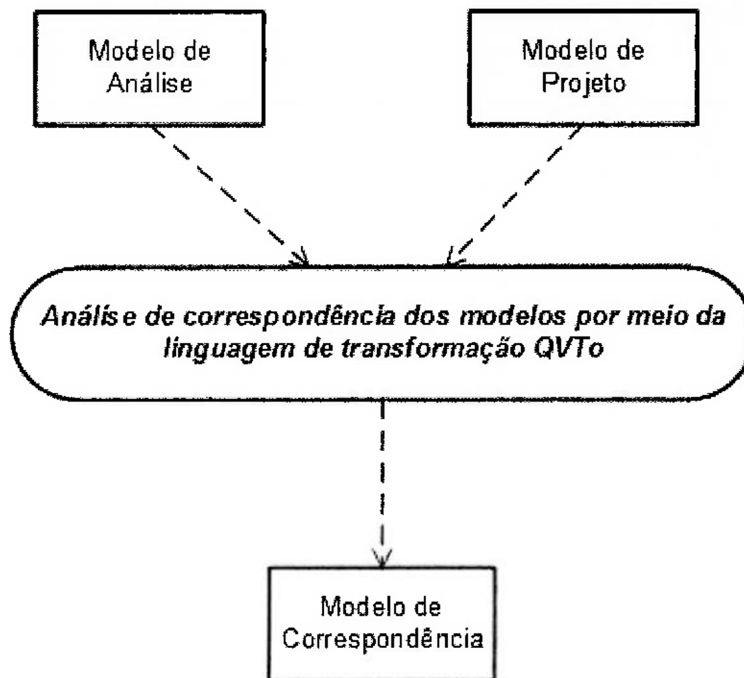


Figura 4.2: Esquema geral da análise de correspondência

O esquema da transformação é apresentado na figura 4.3. Neste esquema apresentado que os modelos de análise e de projeto são instanciados pelo metamodelo SimpleUML e que a transformação acontece usando uma transformação em QVTo. O modelo com resultado da correspondência é instanciado pelo metamodelo de correspondência. Todos os metamodelos estão conforme o metametamodelo MOF.

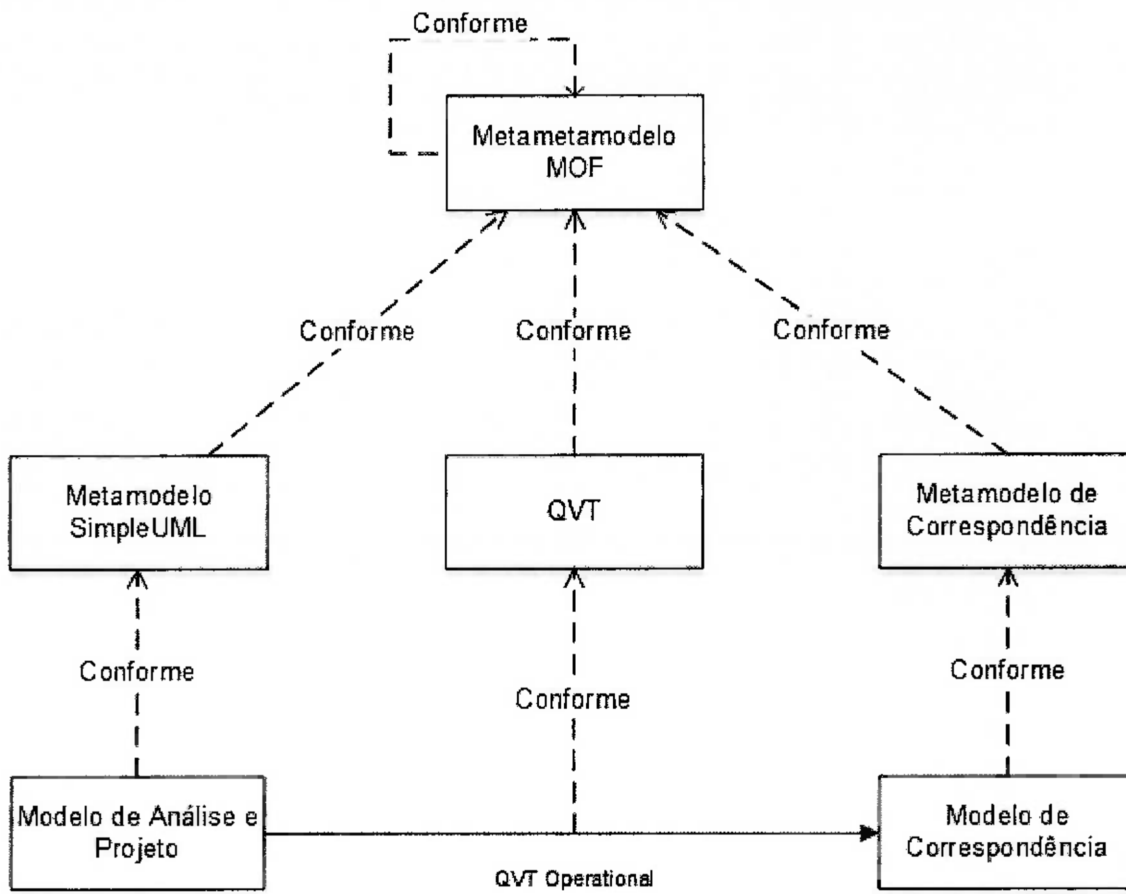


Figura 4.3: Transformação proposta

4.5. Método para Criação das Regras de Correspondência

Neste trabalho a transformação é realizada com o intuito de analisar se o modelo de projeto é um refinamento válido do modelo de análise, obtendo como resultado um modelo com as informações da correspondência entre eles. A transformação será realizada por meio de um conjunto de regras. Para obtê-las, existem dois caminhos: o primeiro é abordagem de Transformação de Modelos por Exemplo *(MTBE, do inglês – Model Transformation By Example)* para especificação e projeto de uma transformação. Essa abordagem utiliza exemplos da transformação para criar regras, podendo ser executada de forma semiautomática (VARRÓ, 2006) ou manual (SIQUEIRA; MUNIZ SILVA, 2013).

A segunda alternativa é imaginar possíveis regras e, posteriormente, propor situações pra exemplifica-las – o que será chamado aqui de exemplos individuais.

Essa abordagem depende da criatividade do desenvolvedor, pois as regras nasceriam de acordo com o que ele acha que é uma regra válida, o que pode gerar regras que não são necessariamente adequadas.

Para entender um pouco melhor a diferença entre as duas possibilidades de criar as regras de transformação serão explicados alguns pontos entre as duas abordagens.

1) MTBE
    a. Exemplo completo: são regras obtidas com base nos modelos existentes seguindo uma ordem lógica, ou seja, vão surgindo regras de elementos simples que são refinadas.
    b. Método para gerar as regras: método fornece novas regras de acordo com a necessidade ou problema.
    c. Há uma evidência (exemplo) da validade da regra: para validar a regra é necessário ter um exemplo que evidencie a aplicabilidade da regra.
    d. Pode gerar regras complexas: pelo fato de possuir métodos que geram as regras é possível surgir regras difíceis de ser compreendidas.
    e. É necessário outro exemplo para testar: existe a necessidade de testar com outros exemplos para verificar se as regras criadas são realmente válidas em diferentes situações.

2) Exemplos individuais
    a. Exemplos isolados: não existe um ponto de partida para criar a regras. A regra surge de uma simples ideia em analisar a correspondência.
    b. Regras precisam ser pensadas: é imaginada uma regra e aplicada na transformação.
    c. Um exemplo é gerado para criar uma evidência: um exemplo é criado para evidenciar as regras criadas.
    d. A complexidade é definida pelo criador: o desenvolvedor define a complexidade da regra de acordo com seu conhecimento ou problema a ser resolvido.
    e. Um exemplo real pode ser usado como teste: para realizar o teste das regras criadas basta utilizar um exemplo de transformação para analisar as regras.

Na tabela 4.1 são apresentados os pontos importantes na utilização das duas abordagens para a criação das regras.

Tabela 4.1: Pontos importantes para as duas abordagens MTBE e exemplos individuais.

MTBE	Exemplos individuais
Exemplo completo	Exemplos isolados
Método para gerar as regras	Regras precisam ser pensadas
Há uma evidência (exemplo) da validade da regra	Um exemplo é gerado para criar uma evidência
Pode gerar regras complexas	A complexidade é definida pelo criador
É necessário outro exemplo para testar	Um exemplo real pode ser usado como teste

Considerando as vantagens e desvantagens dessas duas abordagens, neste trabalho as regras serão criadas baseadas em um modelo de análise e de projeto utilizados como exemplo, usando um método manual baseado no conceito MTBE para a criação de novas regras. O principal motivo da escolha dessa abordagem é que se o resultado não for o esperado, o desenvolvedor possui um método que o guia na melhoria das regras.

O método que será usado é baseado no trabalho de Siqueira e Muniz Silva (2013), no qual o objetivo é criar ou especializar regras de correspondência. O desenvolvedor cria novas regras manualmente a partir das inconsistências do modelo de saída frente ao modelo esperado (o exemplo). O modelo de saída é obtido por meio de um conjunto de regras transformação iniciais que foram criadas anteriormente. O método inicia-se com quatro entradas: dois modelos de entrada (são os modelos de análise e o de projeto) – o exemplo –, um modelo de saída (resultado da correspondência), e um conjunto de regras iniciais. A partir destas quatro entradas, as etapas para executar o método são as seguintes (SIQUEIRA; MUNIZ SILVA, 2013):

1) Aplicar as regras de transformação existentes nos modelos de entrada e obter um modelo de saída.

2) Comparar cada elemento do modelo de saída obtido (resultado esperado). Se o elemento analisado (pelo desenvolvedor) for semanticamente diferente do resultado esperado (isto é, representa uma informação diferente) ou se não houver um elemento no modelo de saída que corresponda a um elemento esperado no modelo de saída, execute:
    a. Se o elemento for diferente, procurar o exemplo nos modelos de entrada que gerou este elemento.
        i. Identificar a regra de transformação que gerou esse elemento no modelo de saída.
        ii. Analisar se a regra de transformação deve ser especializada, aperfeiçoada ou revisada, buscando uma alternativa para que a regra possa ser criada, ou se não é possível melhorar a regra.
        iii. Se uma regra necessitar de aperfeiçoamento ou revisão, em primeiro lugar verificar se a mudança é relevante para todas as outras regras. Se a resposta for sim, adicione esta nova regra aperfeiçoando todas as regras já existentes, ou revise as regras de transformação; se não, a regra deve ser especializada.
        iv. Se a regra de transformação requer especialização, identificar a condição na qual ela será especializada, tendo em conta os exemplos dos modelos de entrada. Em seguida, definir a nova regra, considerando-se os exemplos dos modelos de entrada e o elemento esperado, com base no exemplo de modelo de saída.
        v. Se não for possível melhorar a regra de transformação, continuar com a etapa b.
    b. Se não houver um elemento no modelo obtido, ou seja, se não existe uma regra relacionada, criar uma nova regra.
        i. Pesquisar nos modelos de entrada os elementos que têm a informação necessária para a regra.
        ii. Analisar a condição necessária para que a regra seja válida.
        iii. Criar a regra.
3) Se alguma alteração foi feita no conjunto de regras, executar novamente o método do passo 1.

A ideia principal do método é a de melhorar o conjunto de regras existentes, seja para especializar, revisar ou incluir novas regras. Para melhorar o conjunto de regras, existe uma dependência da experiência do desenvolvedor para que possa sugerir ou melhorar as regras existentes. Assim como em outras abordagens de MTBE, as regras dependem tanto do sistema em que o método foi aplicado, e na qualidade do exemplo usado como entrada (SIQUEIRA; MUNIZ SILVA, 2013).

4.6. Regras de Correspondência

Para definir as regras de correspondência usando uma transformação, os modelos de análise e de projeto foram usados como modelos de entrada, tendo como resultado um modelo de saída com as informações da correspondência, seguindo o metamodelo de correspondência apresentado na seção 4.3. Foi realizada uma análise entre os modelos de análise e de projeto buscando criar regras que fossem pertinentes e simples para a realização da análise de correspondência entre os dois modelos.

As regras obtidas ao aplicar o método são apresentadas a seguir, usando fragmentos dos modelos de análise e de projeto do sistema de vendas que é apresentada no Apêndice A.

1) Se o nome da classe de projeto possuir o mesmo nome da classe de análise a correspondência entre as classes será verdadeiro, como exemplificado na figura 4.4.

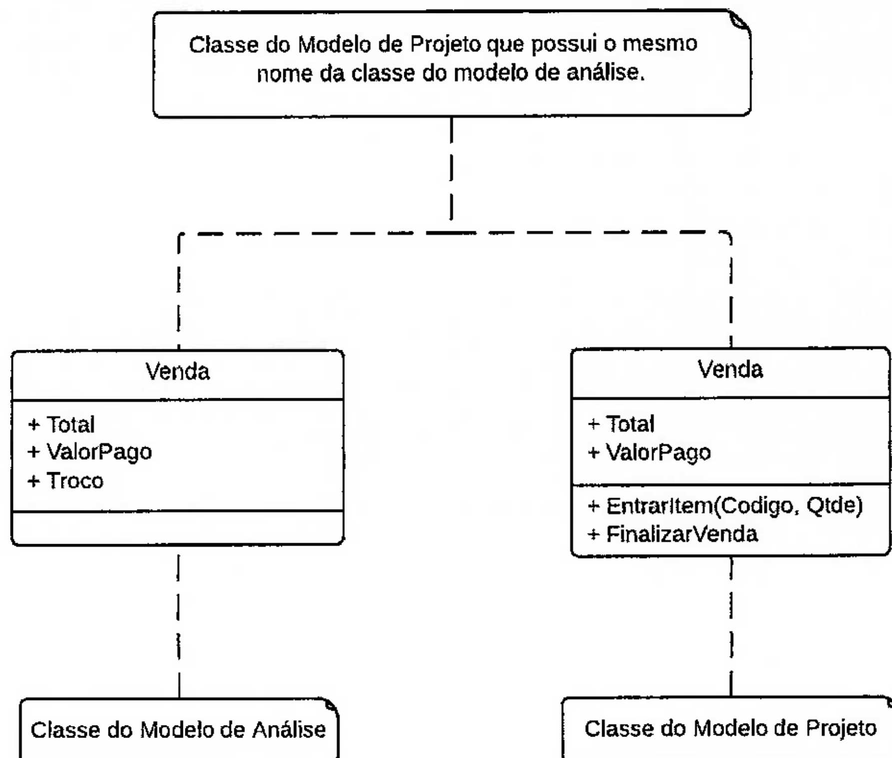


Figura 4.4: Correspondência entre duas classes

2) Se o nome dos atributos da classe de projeto for idêntico aos nomes dos atributos da classe de análise a correspondência entre os atributos será verdadeira, conforme exemplificado na figura 4.5.

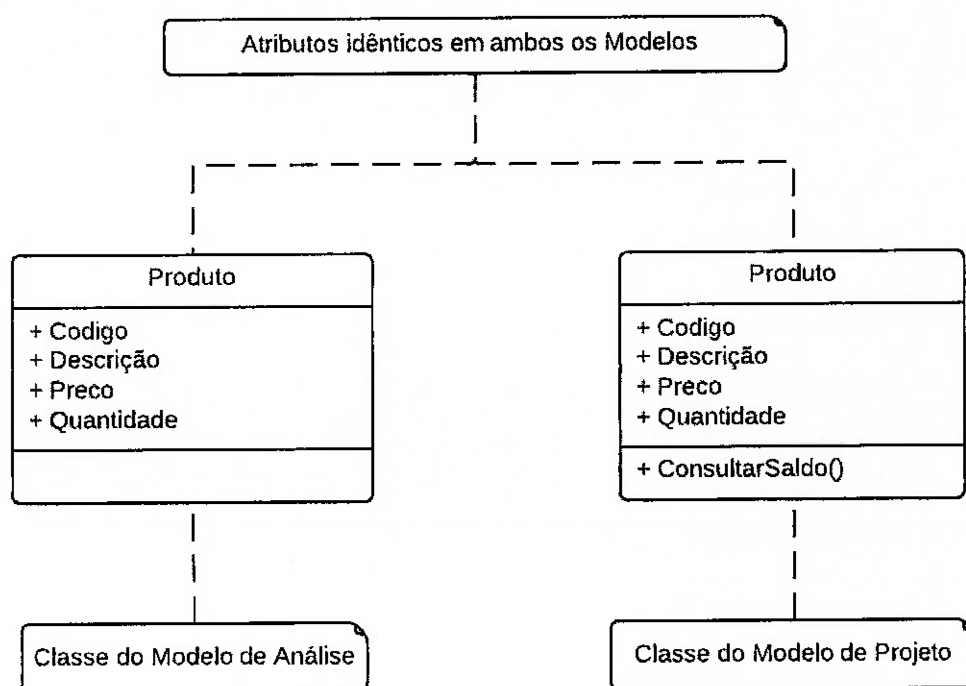


Figura 4.5: Correspondência entre dos atributos

3) Se atributos da classe de projeto não existirem na classe de análise, a correspondência será verdadeira se tiver pelo menos um atributo em ambas as classes, conforme exemplificado nas figuras 4.6 e 4.7.

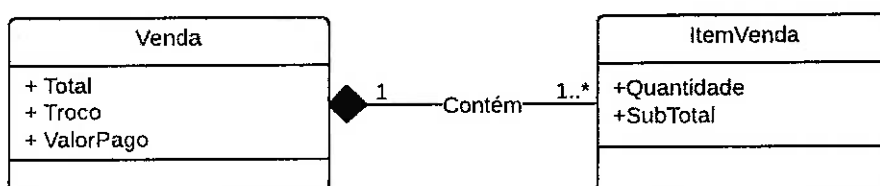


Figura 4.6: Atributos presentes no Modelo de Análise

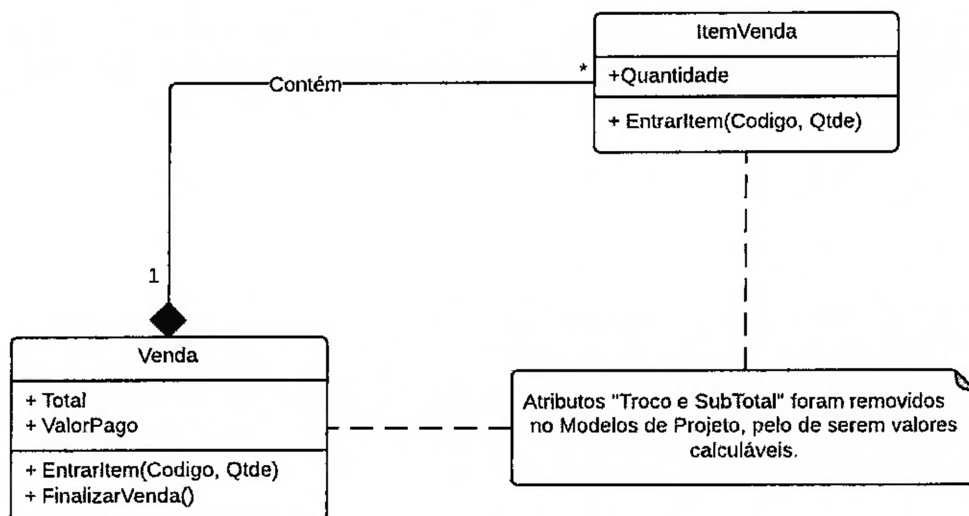


Figura 4.7: Atributos removidos no Modelo de Projeto

4) Se duas classes possuírem uma associação tanto no modelo de análise quanto no modelo de projeto, a correspondência será verdadeira, de acordo com as seguintes regras e conforme as figuras 4.8 e 4.9:

   a. Nomes das classes de origens são idênticos;
   b. Nomes das classes de destino são idênticos;
   c. A multiplicidade da associação do modelo de projeto for igual ao modelo de análise;

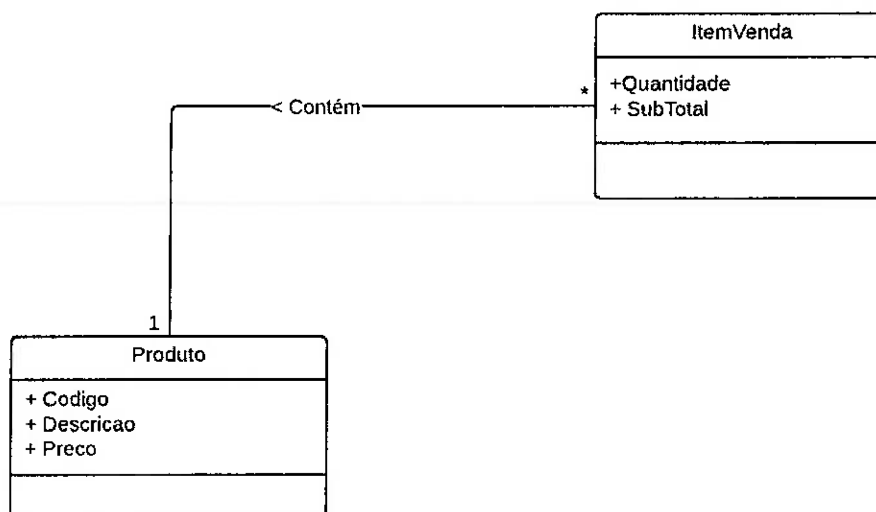


Figura 4.8: Associação do Modelo de Análise

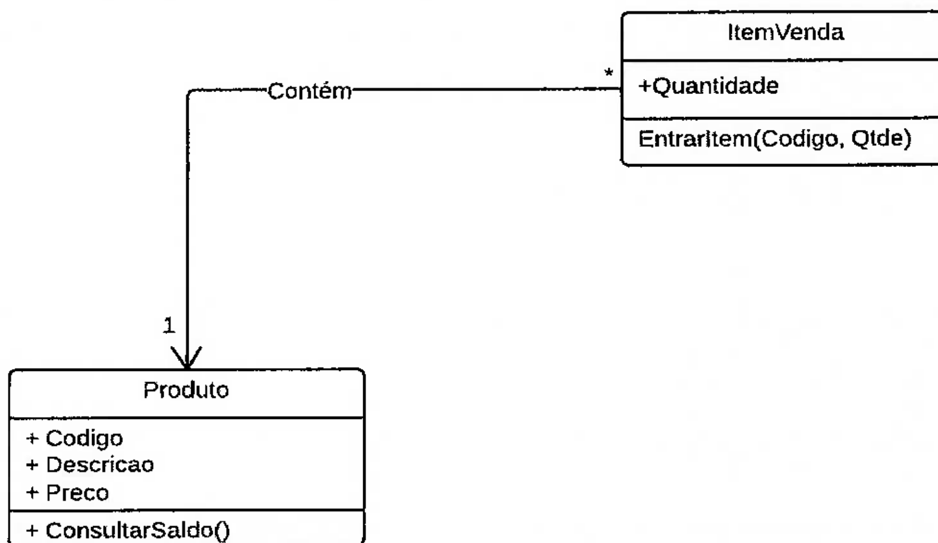


Figura 4.9: Associação do Modelo de Projeto

5) Na regra 4, se a multiplicidade da associação não corresponder, será realizada a seguinte verificação:

   a. Se apenas uma das multiplicidades forem correspondentes, então o resultado da análise de correspondência da multiplicidade será verdadeiro, conforme exemplificado nas figuras 4.10 e 4.11.

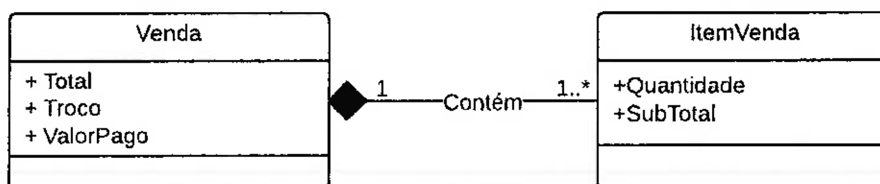


Figura 4.10: Multiplicidade do Modelo de Análise

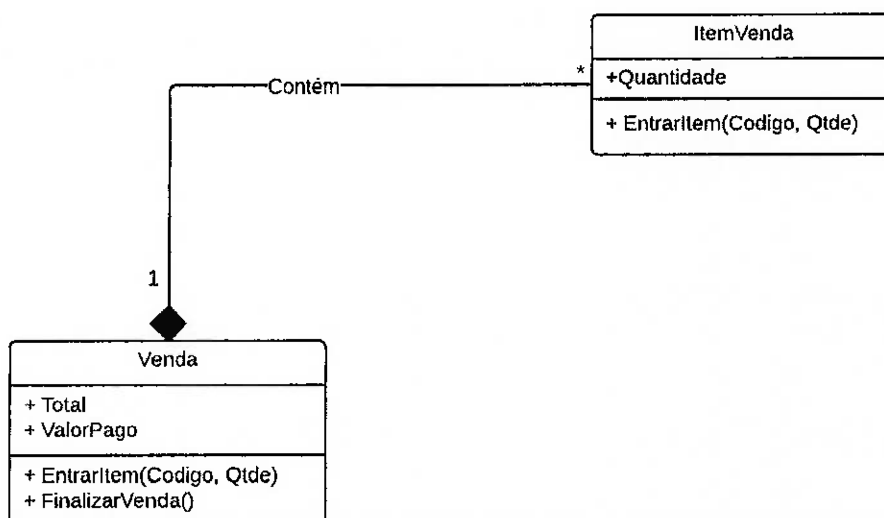


Figura 4.11: Multiplicidade do Modelo de Projeto

6) Se existir alguma classe que está presente no modelo de projeto, mas que não existe no modelo de análise será realizada a seguinte verificação:

   a. Se a classe estiver associada com alguma classe existente em ambos os modelos, a correspondência será verdadeira. Esta correspondência ocorre por meio do nome das classes, ou seja, se ambos os nomes forem idênticos a correspondência será verdadeira, conforme as figuras 4.12 e 4.13.

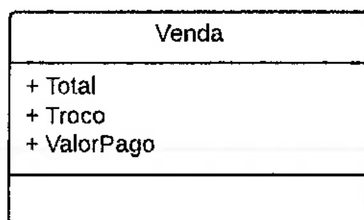


Figura 4.12: Classe no Modelo de Análise

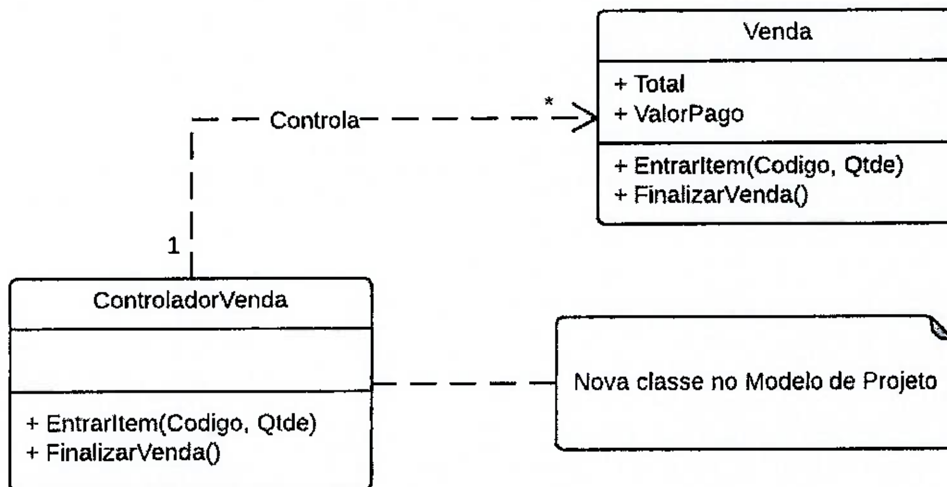


Figura 4.13: Classes do Modelo de Projeto

4.7. Exemplo Ilustrativo da análise de correspondência dos Modelos

Para ilustrar a aplicação da análise de correspondência, a seguir será apresentado um pequeno exemplo utilizando o *script* de correspondência de modelos. Neste exemplo terá como entrada apenas a classe *Venda* e seus atributos que estão dentro do modelo de análise e no modelo de projeto como é apresentado na figura 4.14.

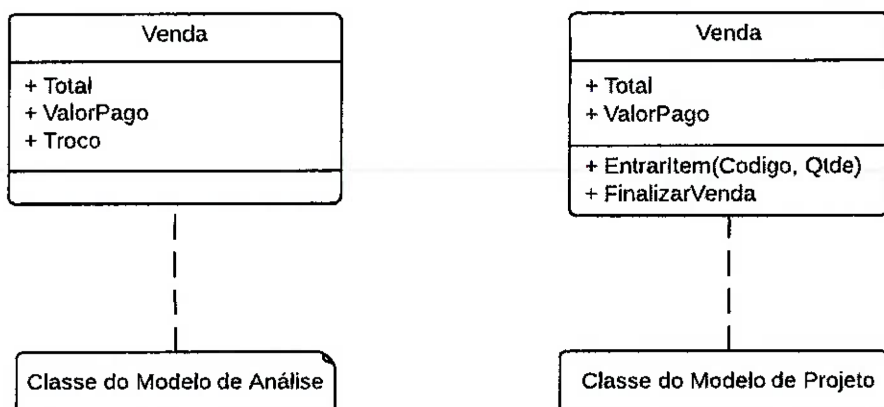


Figura 4.14: Classes do modelo de análise e de projeto

Para este exemplo foram criados os modelos de análise e o modelo de projeto utilizando o metamodelo SimpleUML. Na figura 4.15 é exemplificado o modelo de análise contendo apenas a classe *Venda* com seus atributos *Total, ValorPago e Troco.*

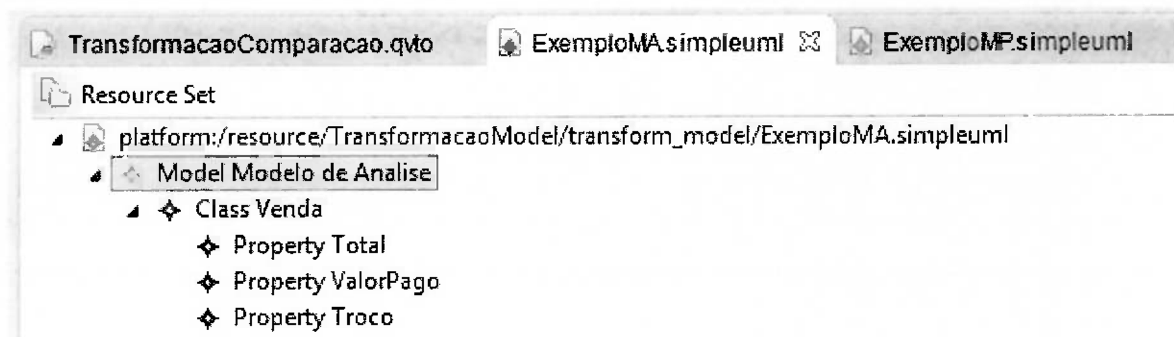


Figura 4.15: Modelo de análise na IDE Eclipse

Na figura 4.16 é exemplificado o modelo de projeto contendo a classe *Venda* com seus atributos *Total e ValorPago.* No modelo de projeto possui uma pequena diferença em relação ao modelo de análise, pois não existe o atributo *Troco*. O motivo pelo qual não existe este atributo se deve pelo fato de ser um atributo que pode ser calculado, ou seja, no modelo de projeto foi tomada a decisão que não haveria necessidade deste atributo.

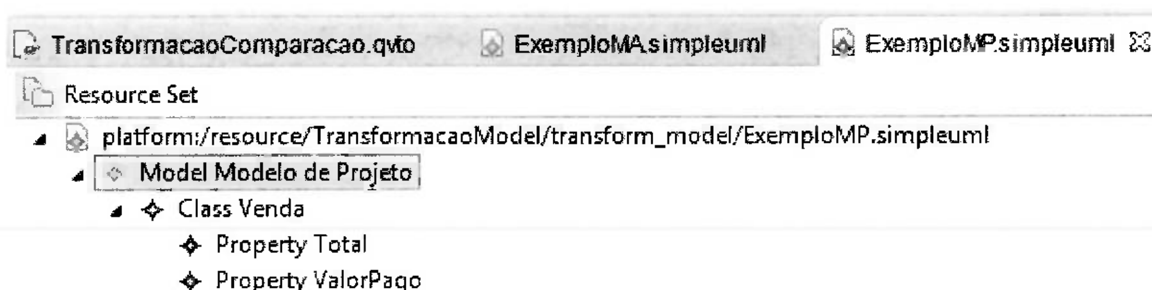


Figura 4.16: Modelo de projeto na IDE Eclipse

No apêndice B é apresentado o script criado para realizar a análise de correspondência entre os modelos de análise e de projeto. Esse script foi implementado por meio da linguagem QVTo utilizando algumas regras apresentadas na seção 4.6. As regras que foram implementas foram a 1, 2, 3 e a 6. As demais regras não foram implementadas, ficando para uma segunda etapa, juntamente com

a melhoria das regras de correspondência para uma melhor análise entre os modelos de análise e de projeto.

Ao executar o *script* de correspondência de modelos é criado um novo modelo contendo informações do que é idêntico e diferente na análise dos modelos de análise e de projeto como é exemplificado na figura 4.17. O refinamento será válido se a classe pertence tanto ao modelo de análise como ao modelo de projeto e se os atributos pertencerem também ao modelo de análise e de projeto.

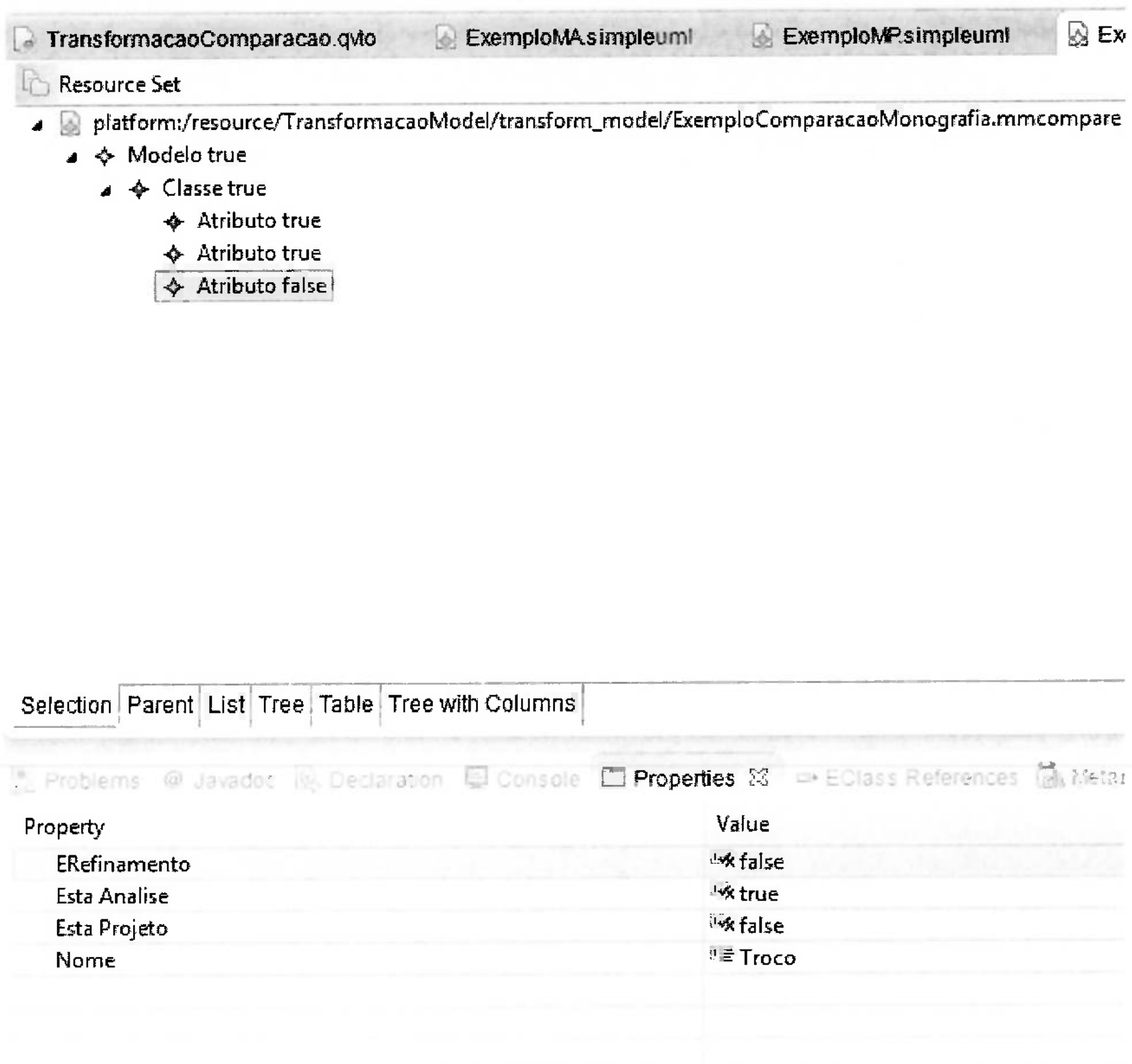


Figura 4.17: Modelo criado com resultado da correspondência entre os modelos de análise e projeto

O resultado mostra os detalhes de cada elemento encontrado nos modelos de análise e de projeto. Neste caso foi selecionado o atributo *Troco*, conforme a figura 4.17 exemplifica, pois é o único elemento diferente entre os dois modelos. Percebe-se que o elemento *eRefinamento* apresentou como resultado false (falso), pois de acordo com os dois modelos o atributo *Troco* não é um refinamento no modelo de projeto. O elemento *estaAnalise* apresenta como resultado true (verdadeiro), pois atributo pertence ao modelo de análise, enquanto o elemento *estaProjeto* apresenta o resultado false (falso) que de acordo com o modelo de projeto este atributo não existe no modelo. E o último elemento é o *Nome* do atributo que é o *Troco.*

Apesar de ser utilizado um exemplo simples, para modelos grandes e complexos esta análise de correspondência seria de grande ajuda, pois ela auxilia os engenheiros a perceberem pequenos detalhes que talvez passasse despercebidos sem uma análise minuciosa de ambos os modelos.

5. CONSIDERAÇÕES FINAIS

Este trabalho apresentou uma proposta para analisar a correspondência entre o modelo de análise e o modelo de projeto com o objetivo de obter a informação se o modelo de projeto é um refinamento válido do modelo de análise, usando conceitos da Engenharia Dirigida por Modelos (MDE). Para isso foi definido um metamodelo que exibe o resultado da operação de correspondência implementada como uma transformação. A transformação ocorre por meio de um conjunto de regras iniciais implementadas na linguagem de transformação QVTo. Essas regras iniciais foram geradas após uma análise dos elementos gerados no modelo de projeto que estão presentes no modelo de análise, considerando um exemplo. Novas regras podem ser acrescentadas seguindo a mesma abordagem dirigida a exemplos empregrada por este trabalho.

O resultado da transformação indica quais são os elementos que pertencem a ambos os modelos ou somente a um dos modelos, por meio de valores booleanos (verdadeiro ou falso). Porém não foi possível gerar regras para concluir de forma automática que o modelo de projeto é realmente um refinamento válido do modelo de análise. Mas se todas as regras tivessem sido implementadas seria possível obter um resultado automático, informando de maneira geral se os modelos se correspondem. Mesmo assim ainda seria necessário analisar a correção das regras, o que envolveria a aplicação do *script* em outros modelos.

Como não foi possível fornecer uma resposta automática, o *script* permite obter uma resposta manual. Essa resposta é obtida pela análise do modelo gerado, que é mais simples de ser analisado do que analisar ambos os modelos. Ou seja, mesmo não tendo como resultado uma informação geral se o modelo de projeto é um refinamento válido do modelo de análise, foi criado um facilitador para realizar essa análise. Outro ponto importante é que a análise passa a ser em apenas um modelo e não em dois modelos (modelo de análise e de projeto), como aconteceria normalmente.

Este trabalho apresentou duas contribuições na área de Engenharia Dirigida por Modelos. A primeira contribuição é a proposta de uma forma para analisar a correspondência entre os modelos de análise e de projeto de forma automática por meio da operação correspondência complexa. A segunda contribuição é o desenvolvimento de um script utilizando conceitos da Engenharia Dirigida por Modelos usando a linguagem de transformação QVT por meio do EMF.

5.1. Trabalhos Futuros

Existe a possibilidade de realizar alguns trabalhos futuros a partir desta pesquisa.

- Analisar a aplicação das regras de transformação.
- Propor mais regras de transformação, usando mais exemplos.
- Analisar se o modelo de implementação (código) é um refinamento válido do modelo de análise, utilizando conceitos da engenharia dirigida por modelos.

REFERÊNCIAS

ATLAS GROUP. **ATL: Atlas transformation language.** User Manual v.7, 2006.

BERNSTEIN, P. A. **Applying Model Management to Classical Meta Data Problems.** In: Biennial Conference on Innovative Data Systems Research (CIDR), p. 209-220, 2003.

BÉZIVIN, J. **On the Unification Power of Models.** Software Systems Model, v.4, n.2, p. 171-188, 2005.

BÉZIVIN, J. **Model Driven Engineering: Na Emerging Technical Space.** In: Generative and Transformational Techniques in Software Engineering, LNCS 4143, p. 36-64, 2006.

BEZERRA, E. **Princípio de Análise e Projeto de Sistemas com UML.** Elsevier, 2007.

BOOCH, G.; RUMBAUGH, J.; JACOBSON, I. **UML: guia do usuário.** Elsevier Brasil, 2006.

BRUNET, G; CHECHIK, M.; EASTERBROOK, S.; NEJATI, S.; NIU, N.; SABETZADEH, M. **A manifesto for model merging.** In: Proceeding of the 2006 international workshop on Global integrated model management. ACM, p. 5-12, 2006.

DOS SANTOS, H. L.; DE BARROS, R. S. M.; FONSECA, D. **Uma Proposta para Gerenciamento de Metadados nos Padrões XML e DTD em Repositórios MOF.** In: SBBD. p. 41-55, 2003.

EMF COMPARE. **Eclipse foundation. EMF COMPARE.** Disponível em: http://www.eclipse.org/emf/compare/. Acesso em: 20-julho de 2014.

FAVRE, J. **Megamodelling and Etymology.** Transformation Techniques in Software Engineering, v. 5161, 2005.

GÉNOVA, G.; VALIENTE, M. C; MARRERO, M. **On the difference between analysis and design, and why it is relevant for the interpretation of models in Model Driven Engineering.** Journal of Object Technology, v. 8, n. 1, p. 107-127, 2009.

GOGOLLA, M.; BÜTTNER, F.; RICHTERS, M. **USE: A UML-based specification environment for validating UML and OCL.** Science of Computer Programming, v. 69, n. 1, p. 27-34, 2007.

KENT, S. **Model driven engineering.** In: Integrated formal methods. Springer Berlin Heidelberg. p. 286-298, 2002.

KRUCHTEN, P. **Introdução ao RUP: rational unified process.** Ciência Moderna, 2003.

KLEPPE, A. G.; WARMER, J. B.; BAST, W. **MDA explained: the model driven architecture: practice and promise.** Addison-Wesley Professional, 2003.

LARMAN, C. **Utilizando UML e padrões: Uma introdução à análise e ao projeto orientado a objetos e ao desenvolvimento iterativo.** 3º. Ed. Porto Alegre: Bookman, 2007.

LOPES, D.; HAMMOUDI, S. ; DE SOUZA, J. ; BONTEMPO, A. **Metamodel matching: Experiments and comparison.** In: Software Engineering Advances, International Conference on. IEEE. p. 2-2, 2006.

LUCRÉDIO, D. **Uma abordagem orientada a modelos para reutilização de software.** Sl: Tese de Doutorado, p. 101-110, 2009.

MELLOR, S. J.; SCOTT, K; UHL, A.; WEISE, D. **MDA DISTILLED: Principles of Model-Driven Architecture.** Addison Wesley, 2004.

OMG – Object Management Group. **Meta Object Facility (MOF) 2.0.** Version 2.4.1, Final Adopted Specification, ptc/11-09-13, 2011.

OMG – Object Management Group. **Meta Object Facility (MOF) 2.0 Qyery/View/Transformation Specification.** Version 1.1, Final Adopted Specification, ptc/09-12-19, 2009.

PRESSMAN, R. S. **Engenharia de Software: Uma abordagem profissional.** 7ª Edição. Ed: McGraw Hill, 2011.

RAHM, E.; BERNSTEIN, P. A. **A survey of approaches to automatic schema matching**. the VLDB Journal, v. 10, n. 4, p. 334-350, 2001.

SABETZADEH, M.; EASTERBROOK, S. **An algebraic framework for merging incomplete and inconsistent views.** In: Requirements Engineering, 2005. Proceedings. 13th IEEE International Conference on. IEEE. p. 306-315, 2005.

SCHMIDT, D. C. **Model-driven engineering.** IEEE Computer, v. 39, n. 2, p. 25, 2006.

SHAN, L.; ZHU, H. **Semantics of metamodels in UML.** In: Theoretical Aspects of Software Engineering, 2009. TASE 2009. Third IEEE International Symposium on. IEEE. p. 55-62, 2009.

SIQUEIRA, F. L.; MUNIZ SILVA, P. S. **Applying MTBE Manually: a Method and an Example.** In: MDEBE@ MoDELS. p. 2-11, 2013.

VARRÓ, D. **Model transformation by example.** In: Model Driven Engineering Languages and Systems. Springer Berlin Heidelberg. p. 410-424, 2006.

APÊNDICE A – Sistema de Venda

A seguir são apresentados três casos de uso simples que serão usados para realizar a análise de correspondência dos modelos de análise e de projeto.

O caso de uso Abrir Caixa - descreve os passos com qual o operador de caixa deve realizar para que caixa seja aberto e assim poder realizar vendas.

O caso de uso Realizar Venda – descreve os passos que deverão ser realizados para que seja possível realizar vendas.

O caso de uso Consultar Saldo – descreve os passos para realizar a consulta do saldo em estoque de um determinado produto.

Nome: Abrir_Caixa

Descrição: Este caso de uso descreve os passos para a abertura do caixa.
Evento iniciador:
Atores: Operador de Caixa
Pré-condição:

Cenários:
Fluxo Principal:

1. Sistema apresenta tela de Usuário e Senha
2. O operador de caixa informa:
    a. Usuário e
    b. Senha.
3. Sistema pede o valor de abertura do Caixa
4. O operador de caixa informa o valor de abertura.
5. Sistema abre o caixa.

Fluxo Alternativo:

1a. (Passo 3) O Usuário ou Senha inválida

1. Sistema avisa que o usuário ou senha informada não é válida e retorna para o passo 2.

2a. (Passo 4) Valor de abertura não informado

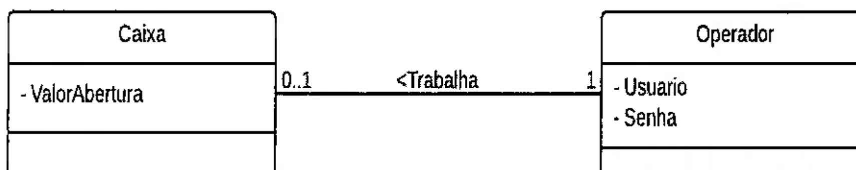
1. Sistema pergunta se operador deseja abrir o caixa sem valor de abertura.
2. O Operador informa que não
3. Sistema retorna para o passa 4.

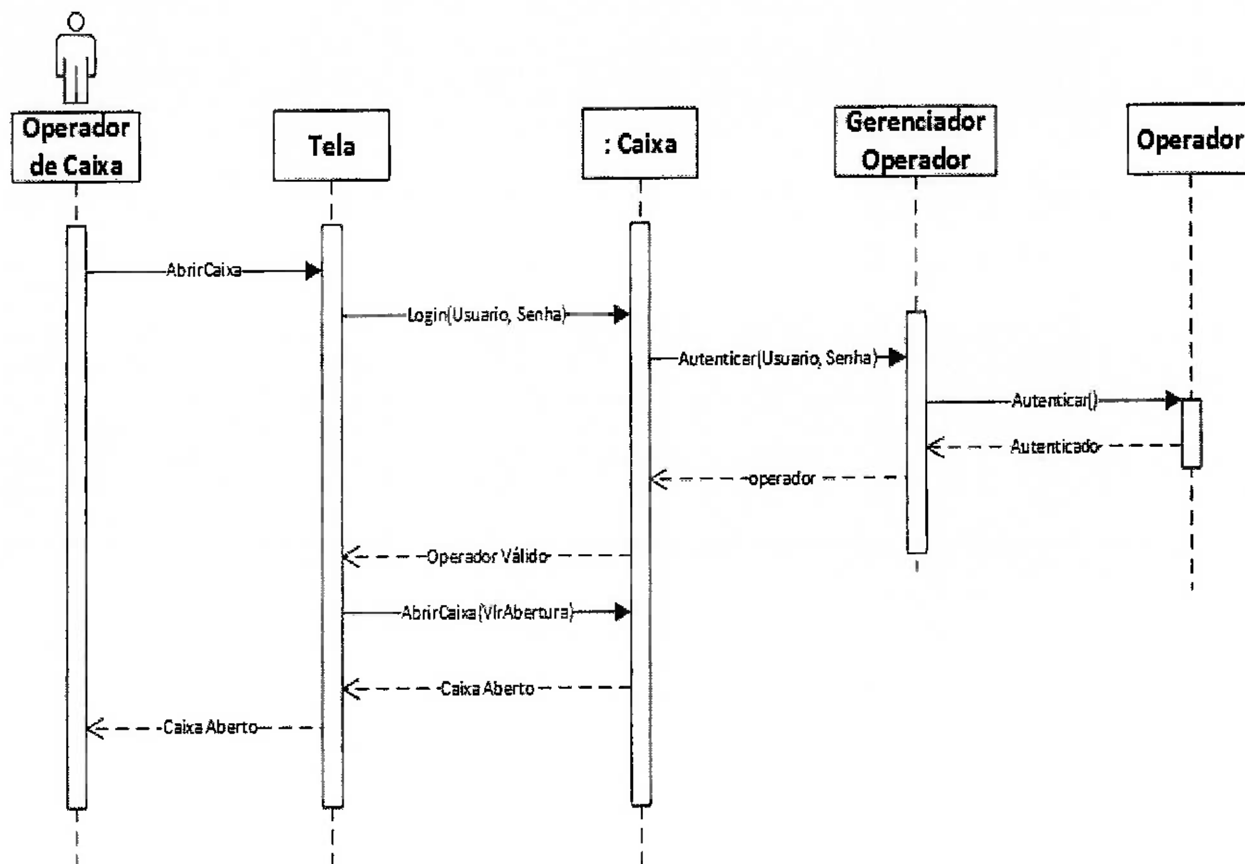
2b. (Passo 2 do fluxo alternativo 2a) Operador informa que sim

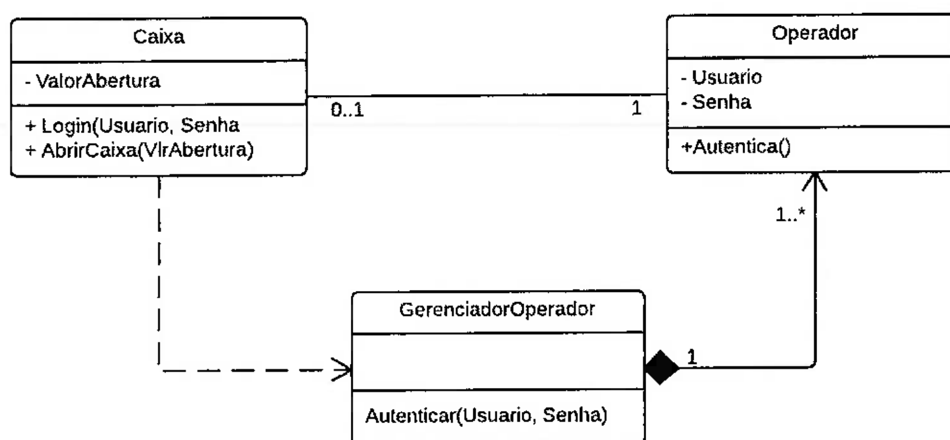
1. Sistema abre o caixa com valor de abertura zero.
2. Sistema retorna para o passa 5.

Pós-condição: Caixa aberto

Inclusões: Não há.




Operador de Caixa
Tela
: Caixa
Gerenciador Operador
Operador
AbrirCaixa
Login(Usuario, Senha)
Autenticar(Usuario, Senha)
Autenticar()
Autenticado
operador
Operador Válido
AbrirCaixa(VlrAbertura)
Caixa Aberto
Caixa Aberto


Caixa
- ValorAbertura
+ Login(Usuario, Senha
+ AbrirCaixa(VlrAbertura)
0..1
1
Operador
- Usuario
- Senha
+Autentica()
1..*
GerenciadorOperador
Autenticar(Usuario, Senha)
1

Nome: Realizar_Venda

Descrição: Este caso de uso descreve os passos para a realização de uma venda.

Evento iniciador: Abrir Caixa

Atores: Operador de Caixa

Pré-condição: Caixa sem vendas finalizadas;

Cenários:

Fluxo Principal:

1. O operador informa:
    a. Quantidade do produto e
    b. Código do produto.
2. Sistema apresenta na lista de itens de venda:
    a. Descrição do item,
    b. Quantidade adquirida,
    c. Preço do item e
    d. Total parcial da venda
3. O operador de caixa repete os passos 2 e 3 até tenha acabado os produtos.
4. Sistema apresenta o total da compra.
5. O operador de caixa informa o valor pago.
6. Sistema finaliza a venda.

Fluxo Alternativo:

1a. (Passo 3) O código do produto é inválido

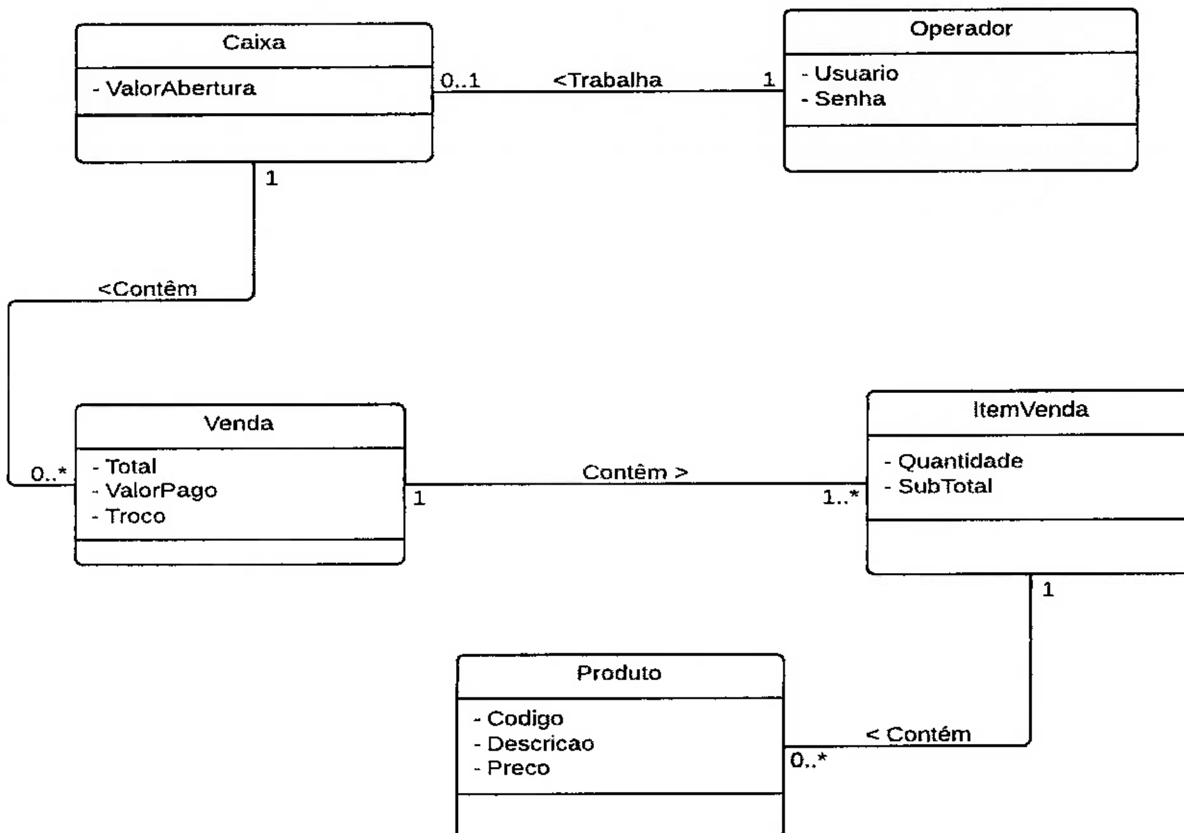
1. Sistema avisa que o código informado não é válido
2. Sistema apresenta uma lista com os produtos
3. O operador de caixa seleciona o produto na lista e retorna para o passo 2.

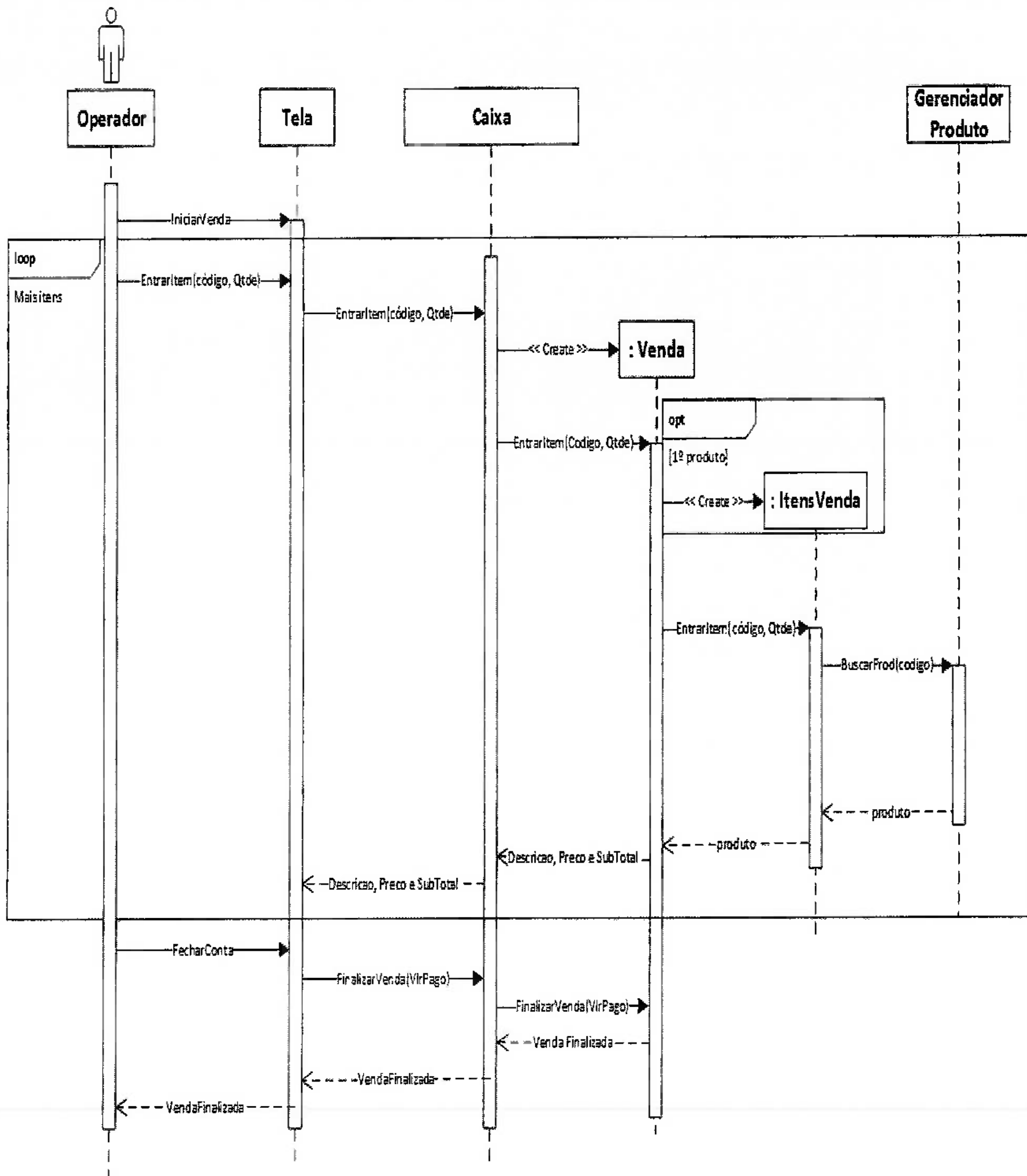
2a. (Passo 5) Emitir troco

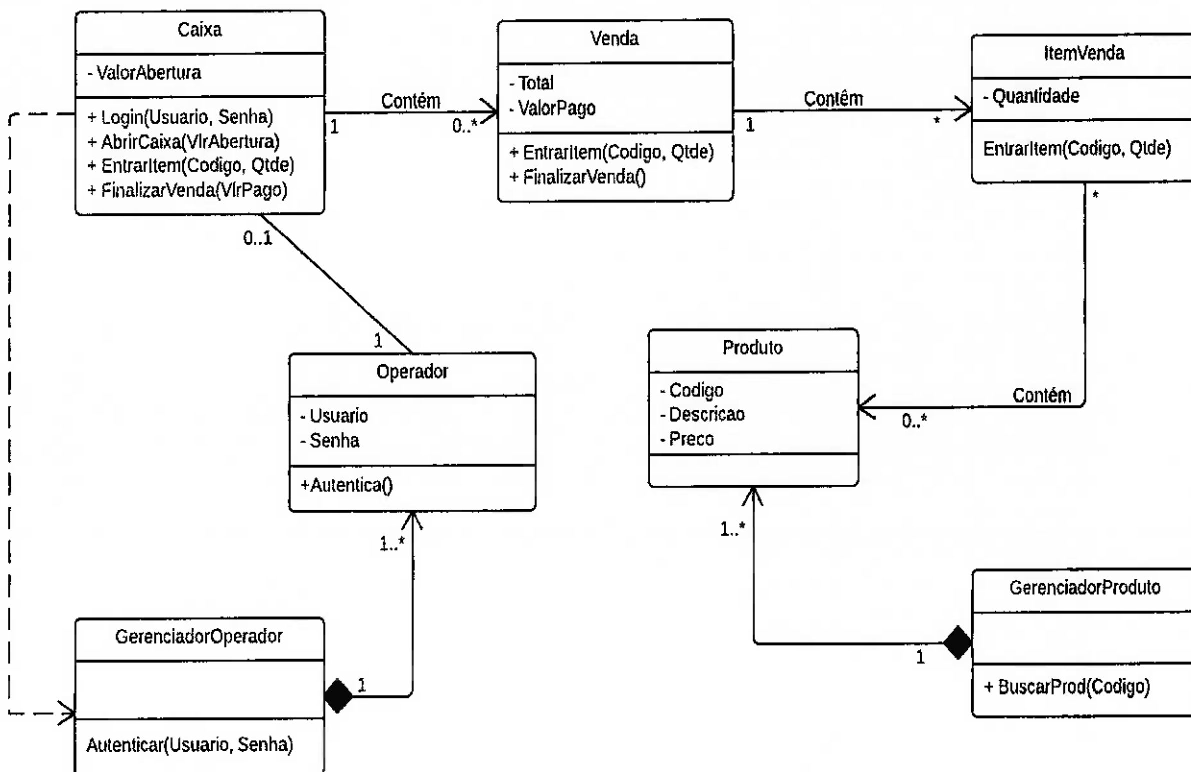
1. Sistema informa o valor do troco
2. O Operador de caixa entrega o valor
3. Sistema retorna para o passo 6.

Pós-condição: Venda finalizada

Inclusões: Não há.




Operador
Tela
Caixa
Gerenciador Produto
IniciarVenda
loop
Mais itens
EntrarItem(código, Qtde)
EntrarItem(código, Qtde)
<< Create >>
: Venda
opt
[1º produto]
EntrarItem(Codigo, Qtde)
<< Create >>
: ItensVenda
EntrarItem(código, Qtde)
BuscarProd(codigo)
produto
produto
Descricao, Preco e SubTotal
Descricao, Preco e SubTotal
FecharConta
FinalizarVenda(VlrPago)
FinalizarVenda(VlrPago)
Venda Finalizada
VendaFinalizada
VendaFinalizada


Caixa
- ValorAbertura
+ Login(Usuario, Senha)
+ AbrirCaixa(VlrAbertura)
+ EntrarItem(Codigo, Qtde)
+ FinalizarVenda(VlrPago)
Contêm
1
0..*
Venda
- Total
- ValorPago
+ EntrarItem(Codigo, Qtde)
+ FinalizarVenda()
Contêm
1
*
ItemVenda
- Quantidade
EntrarItem(Codigo, Qtde)
*
0..1
1
Operador
- Usuario
- Senha
+Autentica()
Produto
- Codigo
- Descricao
- Preco
Contém
0..*
1..*
1..*
GerenciadorOperador
Autenticar(Usuario, Senha)
1
GerenciadorProduto
+ BuscarProd(Codigo)
1

Nome: Consultar_Produto_no_Estoque

Descrição: Este caso de uso a consulta de saldo de um produto no estoque.

Evento iniciador:

Atores: Operador de Caixa

Pré-condição: O Caixa está Aberto

Cenários:

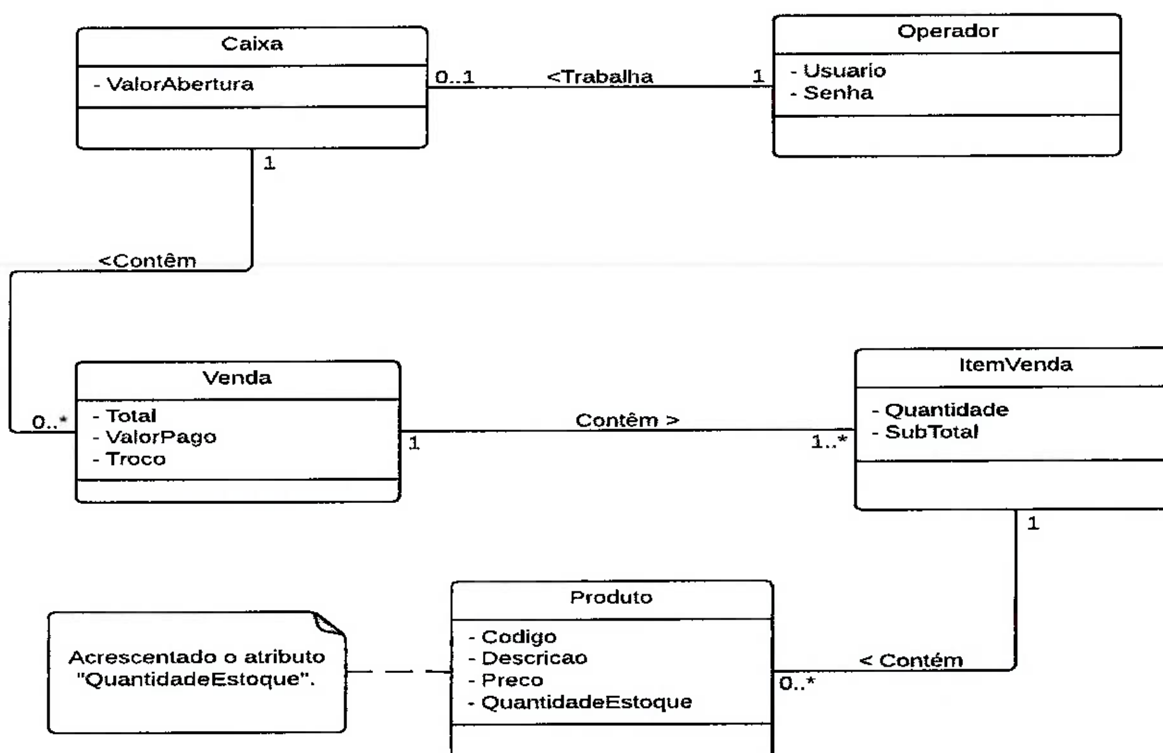
Fluxo Principal:

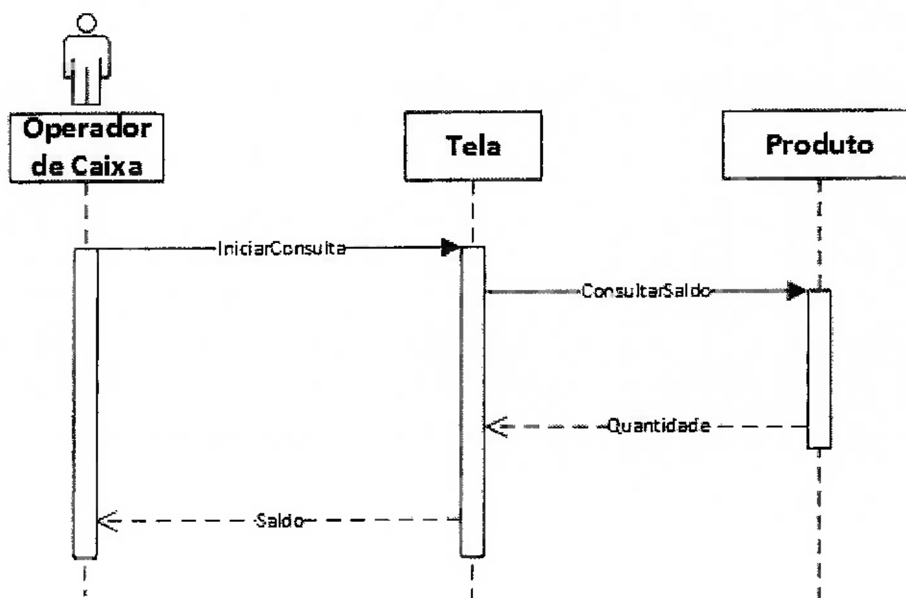
1. Operador de Caixa clica na opção consultar Saldo.
2. O Sistema apresenta:
   a. Campo Código do Produto.
3. O operador de caixa digita o código do produto e clica na opção consultar.
4. Sistema apresenta a quantidade disponível do produto.

Fluxo Alternativo: Não há

Pós-condição: Quantidade do produto

Inclusões: Não há.




Operador de Caixa
Tela
Produto
IniciarConsulta
ConsultarSaldo
Quantidade
Saldo

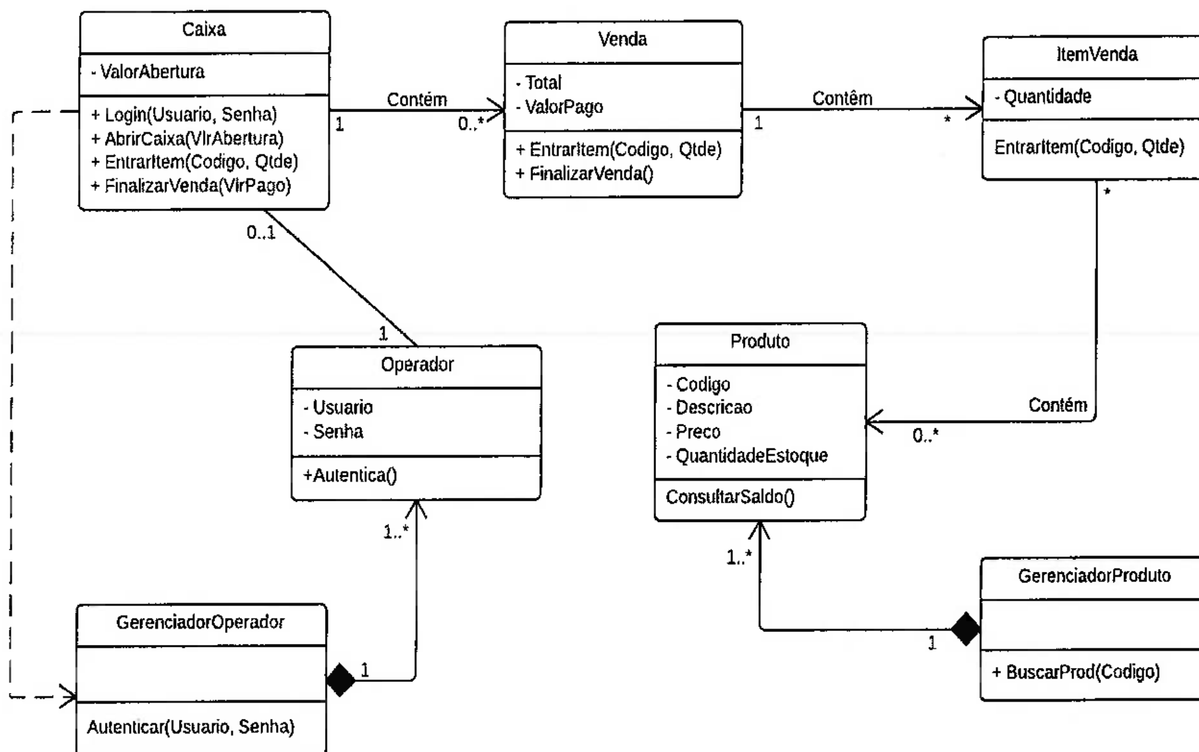

Caixa
- ValorAbertura
+ Login(Usuario, Senha)
+ AbrirCaixa(VlrAbertura)
+ EntrarItem(Codigo, Qtde)
+ FinalizarVenda(VlrPago)
Contém
1
0..*
Venda
- Total
- ValorPago
+ EntrarItem(Codigo, Qtde)
+ FinalizarVenda()
Contêm
1
*
ItemVenda
- Quantidade
EntrarItem(Codigo, Qtde)
*
0..1
1
Operador
- Usuario
- Senha
+Autentica()
1..*
Produto
- Codigo
- Descricao
- Preco
- QuantidadeEstoque
ConsultarSaldo()
Contém
0..*
1..*
GerenciadorOperador
Autenticar(Usuario, Senha)
1
GerenciadorProduto
+ BuscarProd(Codigo)
1

Diagrama de Classe de Análise Completo

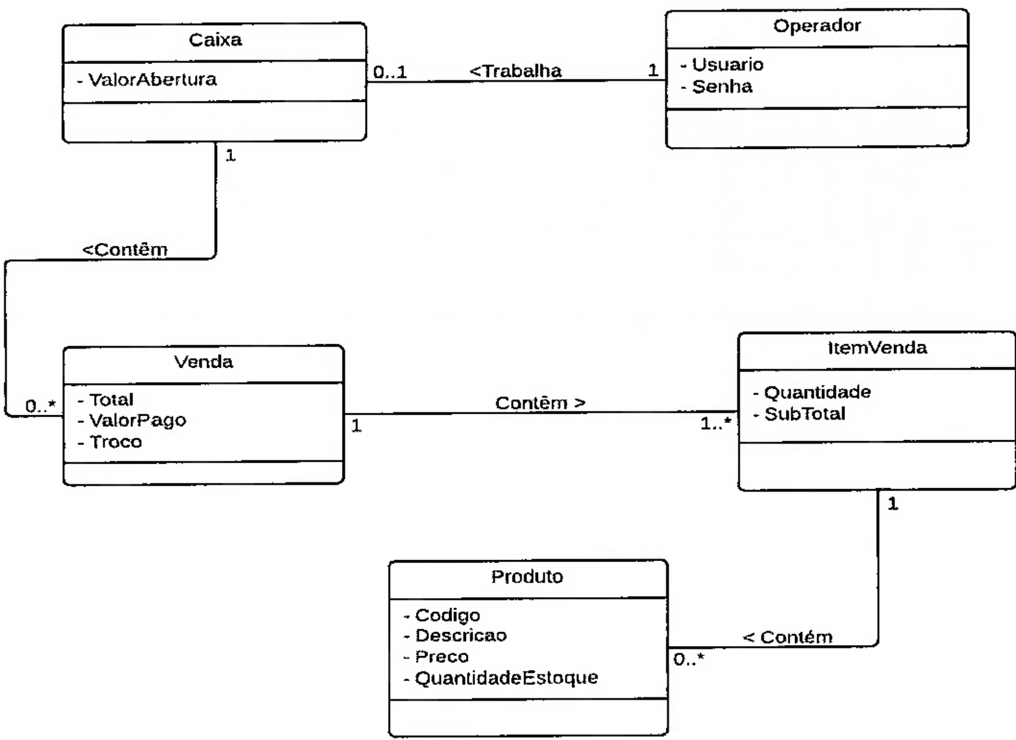
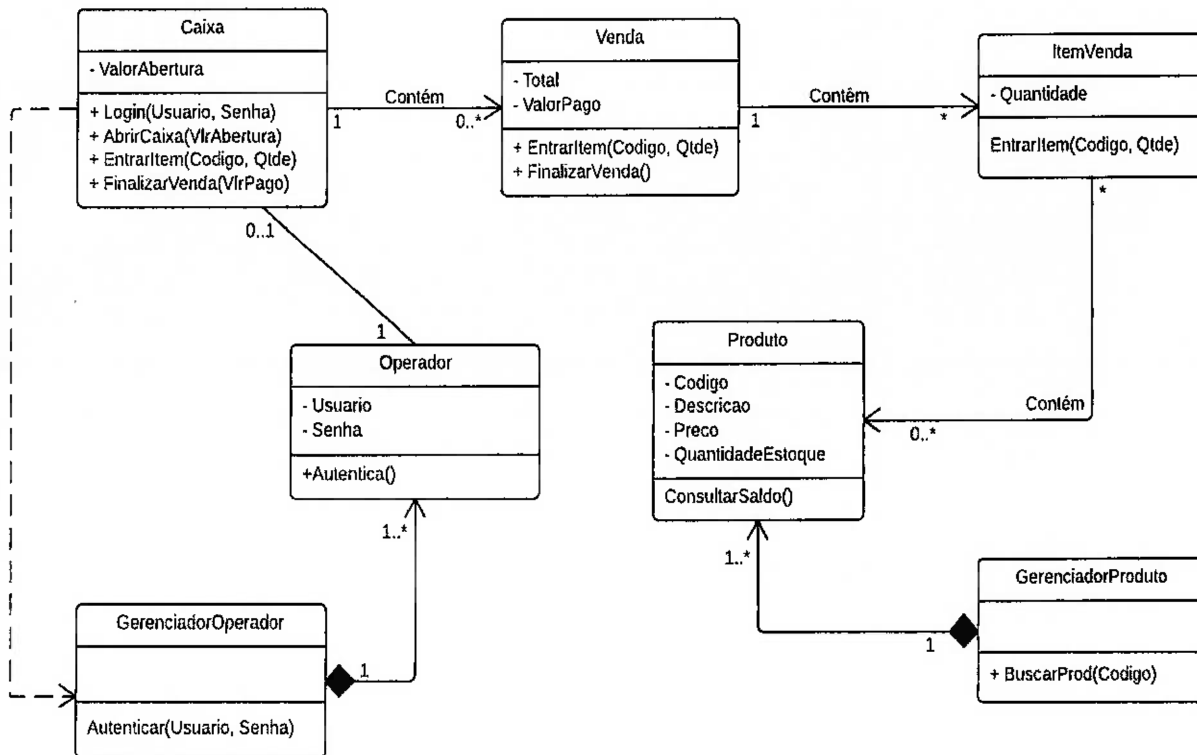


Diagrama de Classes de Projeto Completo



APÊNDICE B – Regras de Correspondência

A seguir é apresentado o script criado para realizar a análise de correspondência entre os modelos de análise e de projeto, feito em QVTo.

```
modeltype UML uses
'http://www.eclipse.org/qvt/1.0.0/Operational/examples/simpleuml';
modeltype CMP uses 'http://mmcompare/1.0';

transformation TransformacaoCorrespondencia(in MA:UML, in MP:UML, out CMP);

main() {
	var ma:Model = MA.rootObjects() [Model]->asOrderedSet()->first();

	MP.rootObjects()[Model]->map MapearModelos(ma);
}

/*
	Mapeia o Modelo de Projeto, passando como parâmetro o modelo de
Análise.
*/
mapping UML::Model::MapearModelos(ma : UML::Model) : CMP::Modelo
{

	// Atribuindo o nome ao resultado do Modelo
	result.nome := "ResultadoCorrespondencia";

	// O Modelo sempre será um Refinamento
	result.eRefinamento := true;

	//Realiza a análise de correspondência das classes de Análise e de
Projeto
	result.elementosclasse += self.ownedElements->select(c |
c.oclIsKindOf(UML::Class))[Class]->map AnalisarCorrespondenciaClasses(ma);


	//Verificando se existe alguma Classe do Modelo de Análise que não
está presente no Modelo de Projeto
	result.elementosclasse += ma.ownedElements->select(c |
c.oclIsKindOf(UML::Class))[Class]->map ObterDadosClasseAnalise(self);
}

mapping UML::Class::AnalisarCorrespondenciaClasses(ma : UML::Model) :
CMP::Classe
{

	result.nome := self.name; // Nome da classe do modelo de projeto


	//Como é uma classe do modelo de projeto é atribuido o valor true
para elemento "estaProjeto".
	result.estaProjeto := true;

	/*
		Verifica se a classe corrente está no modelo de análise.
		Se o resultado for verdadeiro é atribuído ao elemento
"estaAnalise" valor true.
```

```
            Caso a classe não seja encontrada o elemento "estaAnalise"
recebe valor false.
        */
      if(ma.ownedElements[UML::Class]->exists(ca | ca.name = self.name))
then {

            result.estaAnalise := true;

          var ca:Class := ma.ownedElements[UML::Class]->map
PegarDadosClasseAnalise(self)->asOrderedSet()->first();
          result.atributo += self.attributes[UML::Property]->map
AnalisarAtributosModeloAnalise(ca);
            result.atributo += ca.attributes[UML::Property]->map
AnalisarAtributosModeloProjeto(self);

            result.eRefinamento := result.atributo->exists(eR |
eR.eRefinamento );

      } else {

            result.estaAnalise := false;
            result.eRefinamento := true;

      } endif;

}

mapping UML::Property::AnalisarAtributosModeloProjeto(in cp : UML::Class):
CMP::Atributo
when { not cp.attributes[UML::Property]->exists(aa | aa.name = self.name) }
{

      result.nome := self.name;
      result.estaProjeto := false;
      result.estaAnalise := true;
      result.eRefinamento := false;

}

mapping UML::Property::AnalisarAtributosModeloAnalise(in ca : UML::Class):
CMP::Atributo

// É analisado as propriedades de cada elemento
when { ca.attributes[UML::Property]->exists(aa | aa.name = self.name) }
{

      result.nome := self.name;
      result.estaProjeto := true;
      result.estaAnalise := true;
      result.eRefinamento := true;

}

mapping UML::Class::ObterDadosClasseAnalise(mp : UML::Model) : CMP::Classe
when { not mp.ownedElements[UML::Class]->exists(ca | ca.name = self.name) }
{

      result.nome := self.name;
      result.eRefinamento := false;
      result.estaAnalise := true;
      result.estaProjeto := false;
```

```
	result.atributo += self.attributes[UML::Property]->map
ObterDadosAtributoAnalise();

}

mapping UML::Property::ObterDadosAtributoAnalise(): CMP::Atributo
{

	result.nome := self.name;
	result.estaAnalise := true;

}

/*
	Mapeia as classes de análise, verificando se o nome da classe de
análise é igual
	ao nome da classe de projeto.
*/
mapping UML::Class::PegarDadosClasseAnalise(cp : UML::Class) : UML::Class
when { self.name = cp.name }
{
	init { result := self; }
}
```