UNIVERSIDADE ESTADUAL DO NORTE DO PARANÁ

***CAMPUS* LUIZ MENEGHEL - CENTRO DE CIÊNCIAS TECNOLÓGICAS**

SISTEMAS DE INFORMAÇÃO

ALLAN ALVES BONIFÁCIO

UM ESTUDO SOBRE DESENVOLVIMENTO DIRIGIDO A MODELOS

Bandeirantes

2015

ALLAN ALVES BONIFÁCIO

UM ESTUDO SOBRE DESENVOLVIMENTO DIRIGIDO A MODELOS

Trabalho de Conclusão de Curso submetido à Universidade Estadual do Norte do Paraná, como requisito parcial para obtenção do grau de Bacharel em Sistemas de Informação.

Orientador: Prof. Dr. André Luís Andrade Menolli

Bandeirantes

2015

ALLAN ALVES BONIFÁCIO

UM ESTUDO SOBRE DESENVOLVIMENTO DIRIGIDO A MODELOS

Trabalho de Conclusão de Curso submetido à Universidade Estadual do Norte do Paraná, como requisito parcial para obtenção do grau de Bacharel em Sistemas de Informação.

COMISSÃO EXAMINADORA

_____
Prof. Dr. André Luís Andrade Menolli
UENP –*Campus* Luiz Meneghel

_____
Prof. Me. Glauco Carlos Silva
UENP – *Campus* Luiz Meneghel

_____
Prof. Me. José Reinaldo Merlin
UENP – *Campus* Luiz Meneghel

Bandeirantes, __ de_____ de 2015.

RESUMO

Esse trabalho apresenta uma pesquisa exploratória, analisando o desenvolvimento dirigido a modelos na prática, identificando sua viabilidade e possíveis benefícios que o mesmo pode trazer. Desenvolvedores vem utilizando a padronização UML para se espelhar em seus modelos de diagramação e produzir seus códigos, mas existe uma lacuna neste processo, muitas vezes os programas não refletem fielmente o modelo criado, tornando-se ambos incoerentes. Uma tecnologia que vem sendo estudada é o desenvolvimento dirigido a modelos, que possui como finalidade a geração de código de forma automática por meio destes modelos, além de manter coerência com os modelos acelerará o processo de desenvolvimento de software, já que desenvolvedores perdem muito tempo criando códigos de forma manual. Além da abordagem UML executável que transforma os modelos UML em código, existem outros meios para transformação do modelo visual em textual, como por exemplo a abordagem EMF que gera código de forma automática a partir de um metamodelo. O grande problema dessas abordagens é a falta de padronização e ferramentas totalmente eficazes para realização desses processos. Sendo assim um estudo sobre o desenvolvimento dirigido a modelos será feito, discutindo a viabilidade dessas abordagens.

Palavras-chave: Desenvolvimento dirigido a modelos; Padronização UML; UML executável; Modelos de diagramação; Abordagem EMF.

ABSTRACT

This work presents an explanatory research, analyzing the development directed models in practice, identifying its viability and possible benefits, which it can brings. Developers in that area have been using UML pattern, to reflect in their models of diagram and produce their codes, but there is a gap in this process. Many times, the program does not reflect faithfully the model created, making it both incoherent. A technology that have been studied is development directed models, which its goal is an automatic code generation by an model, besides of keeping coherence with the models, it will increase the software development process, now that developers lose a lot of time creating manually codes. Despite of the approach UML Executable that transforms UML models in code, are there other means for of transformation from the visual model to the textual one, such as the EMF approach that generates codes in an automatic way starting from a metamodel. The major problem of this kind of approach is the lack of standardization and fully effective tools for the achievement of this process. Being that way, a study on a directed development to this kind of models will thrive, identifying the viability of this subject.

Key-words: Directed development to models; UML standardization; Executable UML; Diagram models; EMF approach.

LISTA DE FIGURAS E QUADROS

LISTA DE SIGLAS

Alf	*Action Language Foundational*
CWM	*Common Warehouse Metamodel*
EMF	*Eclipse Modeling Framework*
fUML	*Foundational Unified Modelling Language*
GRASP	*General Responsibility Assignment Software Patterns*
IBM	*International Business Machines*
MDA	*Model-Ddriven Architecture*
MDD	*Model Driven Development*
MDE	*Model-Friven Engineering*
MOF	*Meta-Object Facility*
MTL	*Model to Text transformation*
OCL	*Object Constraint Language*
OMG	*Object Management Group*
SCXML	*Stands for State Chart eXtensible Markup Language*
SysML	*Systems Modeling Language*
UML	*Unified Modelling Language*
XML	*eXtensible Markup Language*
xUML	*Executable Unified Modelling Language*

SUMÁRIO

1 INTRODUÇÃO

Grande parte dos softwares são desenvolvidos por equipes de profissionais que detêm conhecimentos específicos sobre linguagens de programação e assim possuem reais condições de produzir códigos e criar sistemas dos mais simples aos mais complexos. É conveniente que todos os envolvidos para elaboração de um novo projeto possuam conhecimentos cientifico e empírico, atendendo assim as necessidades que venham a surgir durante o andamento do projeto. O conhecimento empírico está associado a experiências práticas embasada em tudo que foi adquirido durante as percepções vividas (REZENDE, 2005).

A todo momento os envolvidos na área de produção de software procuram formas para melhorar a produtividade e a qualidade, uma forma bem comum é a reutilização de código. No entanto apenas isto não é o suficiente, pois existe a necessidade de programar toda lógica para o determinado sistema (PRESSMAN, 2011).

Efetuando essa reutilização de código, torna-se perceptível a redução do tempo estimado para realização deste processo de criação de software, pode-se notar que nos dias de hoje, o fator tempo é crucial, cada minuto deve ser bem proveitoso e produtivo, a organização do mesmo e de todos os processos é primordial para que sejam bem definidos, principalmente por causa da alta concorrência do mercado. Por conta disso, levando em consideração a relevância do tempo e a qualidade do software, é fundamental a utilização de uma padronização que potencializa o desenvolvimento do código e contribui para que este seja elaborado conforme o planejado garantindo organização (BOOCH; RUMBAUGH; JACOBSON, 2005).

Os engenheiros de software e programadores de modo geral, vêm utilizando bastante a padronização *Unified Modelling Language* (UML), que possui a função de representar a modelagem do sistema, por meio de diagramas compostos por elementos básicos que servem principalmente para visualização e especificação do software. No entanto, existe uma lacuna neste processo, muitas vezes os programas não estão em conformidade com os modelos criados. Por exemplo, quando se modifica o código, o modelo fica incoerente com o mesmo. Ainda existe necessidade de aprimoramento nesses fatores, para fazer com que o modelo UML fique o mais coerente possível com o código (BOOCH; RUMBAUGH; JACOBSON, 2005).

Uma tecnologia que vem sendo estudada há alguns anos para melhorar o processo de desenvolvimento e auxiliar os programadores, é o desenvolvimento dirigido a modelos "*Model-Driven Development*" (MDD). A utilização do MDD facilitará bastante a vida dos programadores e de todos os envolvidos no projeto, pois os códigos serão gerados automaticamente sem a necessidade de serem produzidos totalmente de forma manual, sendo sujeito a erros e perda de tempo, agilizando assim a entrega do produto final. Isso de fato seria uma grande evolução na engenharia de software, sem contar que o grande problema da incompatibilidade entre o modelo criado e o código desenvolvido seria supostamente sanado, já que a produção automática do código será embasada completamente no modelo desenvolvido pelo engenheiro de software (JIANG; ZHANG; MIYAKE, 2007).

Tendo em vista pesquisas constantes realizadas em livros e na internet, ainda não se tem nenhuma padronização que execute esta função perfeitamente como o desejado, o que tem-se, são apenas determinadas ferramentas de modelagens, como o *Astah* e o *Netbeans*, que geram o código das classes e seus métodos em variadas linguagens como por exemplo em *Java*, a partir de um diagrama de classes, más também é possível encontrar alguns *frameworks* mais específicos de MDD, sendo eles o *Papyrus* juntamente com o *Acceleo*. Isso sem dúvidas é muito pouco perto do que realmente precisa-se e procura-se para sanar os problemas do desenvolvimento dirigido a modelos relacionados a falta de ferramentas e abordagens totalmente eficazes para produção de código, a fim de melhorar o processo de desenvolvimento de software (FOWLER, 2005).

O que se busca de fato é a produção de algo mais profundo, não gerar apenas o código do escopo de uma classe, mas possivelmente gerar uma lógica que desempenhe essas funções e melhore as abordagens do MDD. Podendo desta forma integrar mais de um diagrama e produzir um código com a junção dos mesmos se completando um ao outro e assim desempenhando o desfecho de um software completo e coerente com o modelo criado.

Infelizmente, ainda não se encontra com facilidade materiais disponíveis na internet que possa dar uma real situação do tema. Desta maneira um estudo sobre o assunto será feito para poder detectar o que de fato já foi alcançado e o que está por vir para auxiliar neste problema que atinge todos os relacionados a esta área.

Por esse motivo, este trabalho apresenta como proposta a necessidade de um estudo mais aprofundado de algumas abordagens do desenvolvimento dirigido a modelos e sua viabilidade de aplicação.

1.1 FORMULAÇÃO DO PROBLEMA

A muito tempo se vem estudando o *Model-Driven Development* (MDD), uma técnica dentro da engenharia de software, que possui os processos de *Executable UML* (xUML) e *Eclipse Modeling Framework* (EMF). Existem propostas para gerar códigos automaticamente, mas esta não é uma tarefa trivial, foi escolhida uma ferramenta especifica que permite a geração de código, mas infelizmente existe pouca documentação e estudos sobre a mesma. Espera-se analisar alguns métodos de transformação de modelos em código, discutindo as abordagens levantadas nesse estudo.

1.2 OBJETIVOS

1.2.1 Objetivo Geral

O objetivo desse trabalho é pesquisar, estudar e utilizar o desenvolvimento dirigido a modelos, e discuti-lo.

1.2.2 Objetivos Específicos

a) Identificar as principais abordagens de desenvolvimento dirigido a modelos.
b) Identificar as principais ferramentas, linguagens e artefatos necessários para a implementação das abordagens escolhidas.
c) Propor objetos de estudo para as principais abordagens do desenvolvimento dirigido a modelos.
d) Fazer uma discussão acerca das abordagens escolhidas.

1.3 JUSTIFICATIVA

Gerar códigos em diferentes linguagens a partir de diagramas UML ou modelos distintos de forma automática ainda é um desafio, o que se tem hoje em dia

não se sabe até que ponto é viável, já que até então, há uma extrema necessidade da utilização do programador para implementar código gerado por meio destes diagramas e modelos.

Apesar do conceito ser bem definido e existir a abordagem MDD, este feito ainda possui lacunas referente a padronização da UML Executável. Efetuando pesquisas na internet, pode-se constatar que o contexto relacionado a este assunto até então é vago, com informações de geração de código automática bem sucedida sendo escassas, não abordando totalmente o assunto e nem mesmo repassando o domínio necessário para executar este procedimento que gera código automaticamente.

Desta forma, é possível notar que existem diversos problemas envolvendo a realização da execução de UML Executável, por isso que não se trata de uma tarefa trivial. A complexidade envolvida nesta ação, sem contar problemas que podem prejudicar este processo, como engenheiros de software desenvolverem diagramas incompletos incapazes de produzirem o código que realmente necessita-se, juntamente com o problema da inconsistência entre os modelos UML, levando a complicações sobre qual modelo seguir no exato momento da transformação do código-fonte. A gravidade destes problemas pode ser destacada dependendo da forma que os modelos forem desenvolvidos e de acordo com que as inconsistências são exibidas (BURDEN; HELDAL; SILJAMAKI, 2011).

Por esse motivo, existem diversas tecnologias com recursos diferentes para transformação de diagramas UML e outros modelos em código, não se sabe até que ponto é viável. As variadas ferramentas que temos disponíveis hoje, possuem uma determinada complexidade e não são totalmente eficientes, muito se fala sobre o desenvolvimento dirigido a modelos. Mas na prática é difícil encontrar material que mostre a viabilidade desta abordagem. Em cima de uma abordagem já existente, o MDD, um estudo será feito com intuito de identificar o real estado da arte, realizando uma discussão entre as abordagens em foco para assim verificar o quanto elas podem ser eficazes.

1.4 METODOLOGIA

Com o intuito de esclarecer questões relacionadas a falta de padronização e a viabilidade da utilização do desenvolvimento dirigido a modelos, é necessário

realizar diversas pesquisas sobre o tema, para assim chegar a conclusões relacionadas a este assunto. Segundo Gil (2002), existe a necessidade da construção de um modelo conceitual e operacional de pesquisa, para assim conseguir realizar uma comparação entre a visão teórica analisada e os dados da realidade.

Desta forma, esta pesquisa pode ser classifica como um estudo exploratório, pois há uma necessidade em entender a fundo suas metodologias e esclarecer dúvidas relacionadas a viabilidade do tema, explorando as possíveis soluções e benefícios que o desenvolvimento dirigido a modelos pode fornecer.

Por meio deste estudo e análises aprofundadas, informações do assunto devem ser levantadas, coletadas e discutidas, para assim tirar conclusões a respeito deste método dentro da engenharia de software.

Esta pesquisa é qualitativa, buscando a compreensão de idéias e dados, sendo analisados especificamente, havendo uma compreensão das características tratadas, para assim chegar a uma conclusão exata a respeito desta abordagem que ainda não está bem definida.

Para compreender melhor a metodologia aplicada a este estudo, o presente trabalho foi dividido em quatro fases que podem ser compreendidas detalhadamente no Capitulo 4.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo visa abordar os principais conceitos do desenvolvimento dirigido a modelos, efetuando uma revisão teórica necessária para o seu entendimento. Nas seções abaixo será enfatizado suas principais abordagens e ferramentas.

2.1 *UNIFIED MODELING LANGUAGE* (UML)

A UML é uma linguagem de modelagem, utilizada para visualização, especificação, construção e documentação de sistemas de software, por meio de diagramas padronizados (BOOCH; RUMBAUGH; JACOBSON, 2005). Este método é indicado para projetos de software considerados complexos. A UML contém diversos recursos de modelagem, que permitem que os mais diversos tipos de sistemas sejam modelados, tais como sistemas coorporativos, distribuídos, sistemas *web* e até mesmo sistemas bem mais complexos, como sistemas embutidos ou de tempo real.

Mesmo a UML sendo muito expressiva, pois abrange todas as visões necessárias ao desenvolvimento e implantação de um sistema, não é considerada difícil de compreendê-la. Por se tratar de uma notação padrão de diagramação, é fácil de ser compreendida, pois possui uma representação gráfica de notações simples e eficientes. Quando se diz que a UML é uma linguagem de especificação, subentende-se que engenheiros de software constroem modelos específicos, sendo cada um com sua respectiva complexidade. De maneira mais afunilada, a UML atende as decisões de análise, projeto e implementação, as quais são cruciais para sistemas de software com uma obscuridade significativa (PILONE; PITMAN, 2005).

Essa padronização da UML faz com que muitas empresas a utilizem como linguagem de modelagem. De acordo com a OMG (2000), diversas organizações utilizam a UML para desenvolver artefatos de seus projetos, pelo fato de possuir um processo intencionalmente independente e, se gerenciado de maneira correta, pode ser o ponto fundamental para o sucesso de um projeto. Os processos da UML devem ser adaptados de acordo com as necessidades dentro da organização e do problema em questão, mesmo que ela não exija um processo especifico, seus utilizadores reconhecem a relevância de um caso de uso impulsionado, arquitetura centralizada, processo iterativo e incremental, sendo assim pode-se permitir, mais não exigir a utilização de processos, desde que o enfoque seja na melhoria da qualidade.

Basicamente as organizações e pessoas envolvidas no ramo da engenharia de software utilizam a UML para três principais fins: esboço, projeto e linguagem de programação. O esboço é o mais tradicional dos três, por meio dele desenvolvedores podem utilizar a UML para auxiliar a transmitir maiores detalhes de um sistema, já que primeiramente se desenha um diagrama para a partir deste produzir o código. O principal objetivo é utilizar estes esboços para transmissão de ideias e possíveis alternativas para soluções ainda não previstas em um projeto, citando detalhes do esboço que deseja visualizar antes do início da programação. Estes esboços são bem informais, é necessário que o mesmo seja criado com rapidez e colaboração, sendo úteis também em documentos com o enfoque em comunicação (FOWLER, 2005).

Contudo, segundo Fowler (2005) os esboços são informações incompletas que destacam as informações mais relevantes, enquanto os projetos são mais completos com o intuito de minimizar a programação. Conforme se lida com a UML, percebe-se que a programação fica mais automatizada, e algumas ferramentas realizam geração de código, mesmo que de forma básica e genérica, facilitando ainda mais no desenvolvimento de um sistema.

Mas antes de criar esboços e projetos deve-se entender como funciona a estruturação da UML. A estrutura da UML é composta por diversas notações de fácil entendimento. Este conjunto de notações pode ser aplicado em diferentes tarefas de acordo com as necessidades que forem surgindo. Existem diversos elementos básicos, os quais definem os modelos, as relações entre estes elementos básicos que relacionam os objetos do diagrama e os diagramas que fazem os agrupamentos de todos estes elementos. Desta forma pode-se construir a modelagem de um sistema (SILVA; VIDEIRA, 2001).

A modelagem da UML é utilizada para fornecer conceitos, relacionamentos, tomadas de decisões e os requisitos em notações claramente definidas, que podem ser aplicadas em diferentes tarefas. De maneira geral um modelo UML pode possuir um ou mais diagramas e a junção de vários diagramas pode formar um modelo completo, pois este modelo visualmente representa elementos e relacionamentos entre eles. Esses elementos básicos podem possuir representações por meio de objetos no mundo real, fabricação de software puro, ou descrição de comportamento de qualquer outro objeto. Cada diagrama representa um interesse especifico em particular, ou do que está sendo modelado naquele determinado momento (PILONE; PITMAN, 2005).

Tratando-se de modelagem visual pode-se destacar que a mesma é de grande vantagem. Quando se trabalha com UML é normal que haja uma exploração da capacidade cerebral de identificar símbolos, unidades, relacionamentos, e suas notações tudo de maneira explicitamente visível. Os diagramas auxiliam na leitura visual fazendo com que se possa explorá-lo de maneira mais fácil e ágil, observando relacionamentos entre elementos de análise ou software, instantaneamente permitindo que detalhes menos relevantes sejam descartados ou deixados de lado (LARMAN, 2007).

Tendo em mente que a UML é um tipo de linguagem, deve-se conceituar o que isso realmente significa. As linguagens possuem as características de fornecer regras para combinação de certo vocabulário, a fim de comunicar algo. Uma linguagem de modelagem possui a característica ímpar de representar o conceito-físico de um sistema. Modelagem vem da palavra modelo, que se subentende que não é algo inteiramente suficiente. É necessária a combinação de vários modelos para que assim, entenda-se visualmente o que está sendo transmitido (BOOCH; RUMBAUGH; JACOBSON, 2005).

Para criar representações destes modelos por meio de diagramas, é necessário que seja feito a escolha de uma ou mais ferramentas de modelagem UML. A independência da UML é algo a ser relevado, ela é totalmente livre de qualquer linguagem de programação ou qualquer tipo de ferramenta especifica, até mesmo de processos de desenvolvimento. Apesar de existirem algumas indicações de determinados softwares para modelagem nas especificações da UML, cabe aos engenheiros de software e suas organizações optarem pela escolha das ferramentas a serem utilizadas, já que envolvem representações e denotações por meio de desenhos de diagramas, cores, navegação de esquemas e escolhas de objetos (SILVA; VIDEIRA, 2001).

De acordo com Booch, Rumbaugh e Jacobson (2005) a UML não pode ser considerada uma linguagem visual, mas seus modelos podem e, muitas vezes, estão conectados de uma forma direta a outras linguagens de programação. É a representação de tudo que possa ser melhor representado em gráficos, enquanto as linguagens de programação são voltadas mais para termos textuais. Por meio de um ou mais modelos UML é possível gerar código para uma linguagem de programação. Da mesma forma, só que de maneira inversa, pode-se reconstruir um modelo a partir

da sua implementação, revertendo-a a UML. A engenharia reversa necessita de uma combinação entre suporte de ferramentas e intervenção humana.

A UML pode ser utilizada em qualquer tipo de sistema de diferentes domínios, tais como: sistemas de informações corporativos; serviços bancários e financeiros; telecomunicações; transportes; defesa em espaço aéreo; vendas de varejo; eletrônica médica; científicos; serviços distribuídos baseados na *Web*. A UML não é restrita apenas à modelagem do software; também pode ser usada como o fluxo do trabalho do sistema legal, a estrutura e o comportamento de sistemas de saúde e o projeto de hardware. Basicamente a UML é composta por três itens de blocos de construção, sendo eles itens, diagramas e relacionamentos. Os itens são as abstrações identificadas, os relacionamentos fazem a ligação entre esses itens, e os diagramas agrupam todos estes itens (BOOCH; RUMBAUGH; JACOBSON, 2005).

O padrão UML é definido por meio de 13 tipos distintos de diagramas, mas estes diagramas não são particularmente rígidos, podemos utilizar elementos de um determinado diagrama em outro sem problema algum. O principal objetivo dos diagramas é agrupar seus elementos básicos em uma forma lógica ou estrutural, sendo que estes diagramas podem se relacionar entre si. Cada um possui sua principal finalidade, mas todos são desenhados para facilitar a visualização de sistemas em diferentes concepções, compreendendo aquele determinado sistema por meio do diagrama. A seguir são apresentados os diagramas do padrão UML 2.0 (FOWLER, 2004):

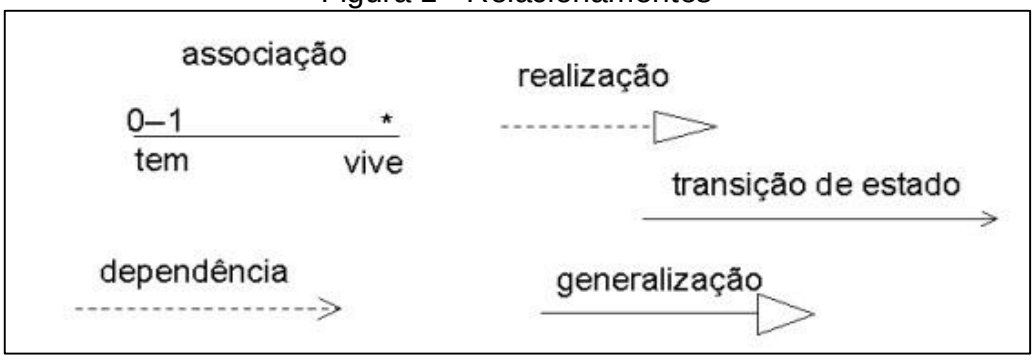
1. Atividades- Comportamento procedimental e paralelo.
2. Classes- Classe, características e relacionamento.
3. Comunicações - Interação entre objetos, ênfase nas ligações.
4. Componentes- Estrutura e conexão de componentes.
5. Estruturas compostas- Decomposição de uma classe em tempo de execução.
6. Distribuição- Distribuição de artefatos nos nós.
7. Visão geral da interação- Mistura de diagrama de sequência e de atividades.
8. Objetos- Exemplos de configurações de instâncias.
9. Pacotes- Estrutura hierárquica em tempo de compilação.
10. Sequências- Interação entre objetos, ênfase na sequência.

11. Maquinas de estados- Como os eventos alteram um objeto no decorrer de sua vida.
12. Sincronismo- Interação entre objetos, ênfase no sincronismo.
13. Casos de uso- Como os usuários interagem com um sistema.

É essencial que usuários iniciantes que pretendem desenvolver suas próprias modelagens UML, estudem o conceito e a utilidade de cada um dos 13 tipos de diagramas. Com o conhecimento especifico a respeito de cada um deles, o erro de desenvolver uma modelagem de seu sistema em cima de um único diagrama já será evitado. Existem momentos que o conceito pode ser expressado em mais de um tipo de diagrama, mas cabe ao usuário escolher qual o mais apropriado e que concederá mais informações relevantes (PILONE; PITMAN, 2005).

Segundo Silva e Videira (2001), "a estrutura de conceitos do UML pode ser vista por meio das seguintes noções 'coisas' ou elementos básicos, com base nos quais se definem os modelos; relações, que relacionam elementos; e diagramas, que agrupam elementos". Na Figura 1 pode-se notar alguns tipos de relações UML, estas são as mais utilizadas, no exato momento da modelagem, sendo elas do tipo associação, dependência, realização, generalização e transição de estado.

Figura 1 - Relacionamentos



Fonte: Silva; Videira (2001).

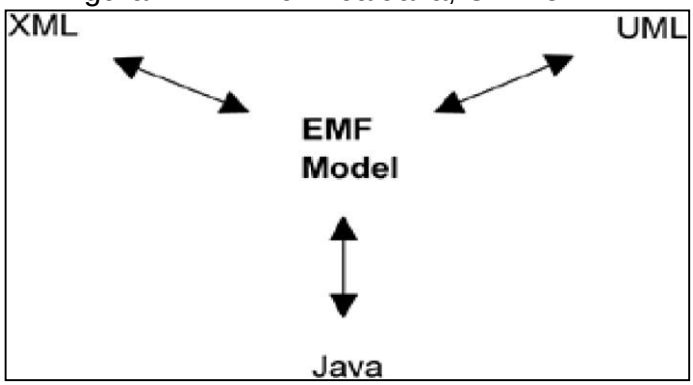
2.2 *ECLIPSE MODELING FRAMEWORK* (EMF)

O EMF é um mecanismo de modelagem e geração de código que facilita a construção de aplicações baseadas em um estruturado modelo de dados, criado no âmbito da arquitetura MDA (STEINBERG et al., 2009). O EMF fornece ferramentas de modelagem e suporte de execução para modelos, com editores sofisticados capazes de conceber implementações completas, incluindo a persistência, e implementação

das regras de negócio (BÉZIVIN et al., 2005).

Segundo Budinsky *et al.* (2003), o EMF permite que o utilizador defina um modelo em linguagens distintas como UML, *Java* e XML, e que possa gerar outros modelos ou classes de implementação, como é mostrada na Figura 2. O EMF unifica essas três importantes linguagens, independentemente de qual delas foi utilizada. Ainda assim, para se ter uma noção maior dos efeitos que o EMF pode proporcionar ao utilizador, imagina-se que ao determinar uma função o modelo EMF gera um diagrama de classes UML, e outra função ocasiona um conjunto de classes de implementação *Java* para manipular um determinado XML. Desse modo, o modelo EMF pode ser definido por meio de apenas uma dessas linguagens, e não necessariamente utilizando essas três simultaneamente.

Figura 2 - EMF unifica Java, UML e XML



Fonte: Budinsky (2003).

Além da gama de tecnologias que são compatíveis com o EMF e as ferramentas disponibilizadas para a elaboração e execução de modelos, o EMF dispõe do seu próprio modelo representativo nomeado como "*ecore*". O *ecore* é em si um modelo EMF, e assim, é o seu próprio metamodelo, essa foi uma medida para padronizar modelos dentro do EMF. No entanto, antes da utilização de um gerador de código de EMF, o utilizador deve definir inicialmente um metamodelo *ecore* que posteriormente de forma automática pode ser transformado em um modelo *ecore* e assim executado para gerar classes de implementação (DALY, 2007).

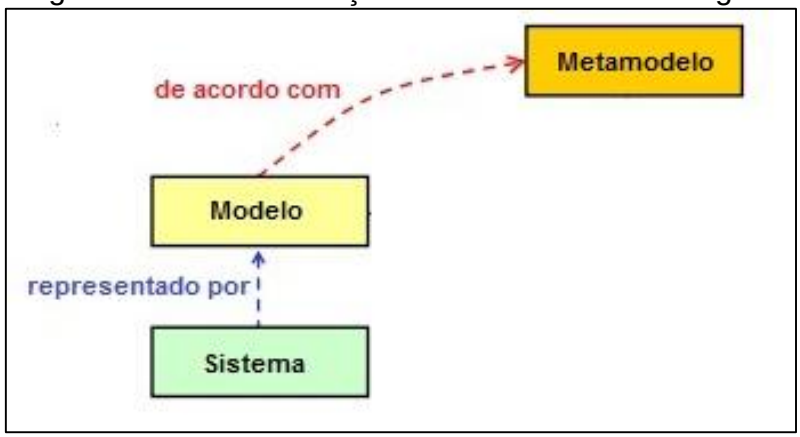
O modelo *ecore* possui uma estrutura similar ao diagrama de classes da UML, possuindo classes, atributos e relacionamentos bem parecidos. Segundo (BUDINSKY et al., 2003), a diferença entre ambos os modelos é destacada pelo fato do modelo *ecore* ser um pequeno subconjunto da UML simplificada, já a UML oferece o suporte completo de modelagem muito mais ambicioso do que o núcleo de apoio na EMF. Por

exemplo, a UML permite modelar o comportamento de uma aplicação, bem como a sua estrutura de classe, sendo mais detalhada e mais complexa que o modelo *ecore* EMF, por isso que o EMF necessariamente precisa do seu próprio modelo.

Apesar do *ecore* ser o dialeto oficial do *framework* de modelagem oficial gerenciado pelo *Eclipse*, o EMF, também pode-se dizer que o *ecore* é o arquivo que contém as informações das classes definidas. No arquivo *ecore* é possível definir alguns elementos, tais como: *EClass* (representa uma classe), *EAttribute* (representa um atributo com nome ou tipo), *EReference* (representa a associação entre duas classes), e por fim o *EDataType* (representa o tipo do atributo, por exemplo, *int*, *string* ou *float*). O modelo *ecore* possui um objeto raiz representado por todo o modelo, possuindo crianças que representam pacotes, filhos que representam as classes, enquanto as crianças das classes representam os atributos destas classes. Entretanto sabe-se que além do *ecore*, o *genmodel* também constitui um metamodelo. O *genmodel* é a outra parte do metamodelo, inclui informações adicionais para a geração de código, como por exemplo o caminho de arquivos (VOGEL, 2015).

A fim de compreender melhor o que é um metamodelo e sua diferença para o modelo, pode-se dizer que o metamodelo descreve a estrutura do modelo e o modelo é o exemplo concreto deste metamodelo. É por meio do metamodelo que se cria as restrições, regras e modelos da classe do problema que foi estabelecido. Sendo assim, cada modelo deve ser uma instância de um metamodelo, e este metamodelo a especificação de um conjunto de modelos, como pode-se notar na Figura 3. Portanto sabe-se que um modelo pode ser criado a partir do metamodelo seguindo todas as suas propriedades, e acrescentando características especificas individuais deste modelo (LEVENDOVSZKY, 2011).

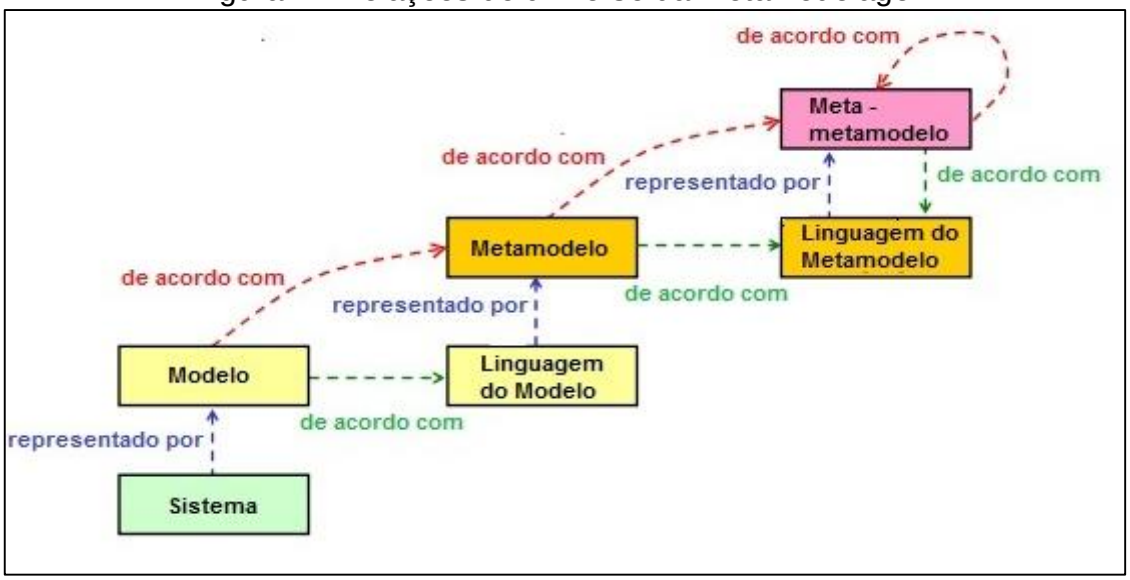
Figura 3 - As duas relações básicas metamodelagem



Fonte: Adaptado de Génova (2009).

Analisando de uma maneira geral o âmbito no qual essas representações estão presentes, entende-se que as relações de metamodelagem podem ser mais complexas do que uma simples ligação entre sistema, modelo e metamodelo, Na Figura 4 é exemplificado visualmente a representação do sistema, modelo, linguagem do modelo, metamodelo, linguagem do metamodelo e meta-metamodelo, afim de demonstrar o meio como um todo, de forma que além do modelo se embasar no metamodelo esse mesmo metamodelo está em conformidade com uma linguagem cuja sintaxe abstrata é representada por meio de um meta-metamodelo, que por sua vez esse meta-metamodelo é representado por si só (GÉNOVA, 2009).

Figura 4 - Relações do universo da metamodelagem



Fonte: Adaptado de Génova (2009).

2.3 *MODEL-DRIVEN ARCHITECTURE* (MDA)

A MDA utiliza modelos para melhorar a performance de planejamento, *design*, e outros ciclos, com o intuito de aprimorar a qualidade e durabilidade dos produtos, basicamente sua função é transformar um modelo visual de software em código executável, seja integralmente ou parcialmente isso pode variar de acordo com as necessidades. Esse padrão MDA além de efetuar a transformação do modelo em código, possui três vantagens importantes que contribuem neste processo de automatização de código, sendo eles (OMG, 2014):

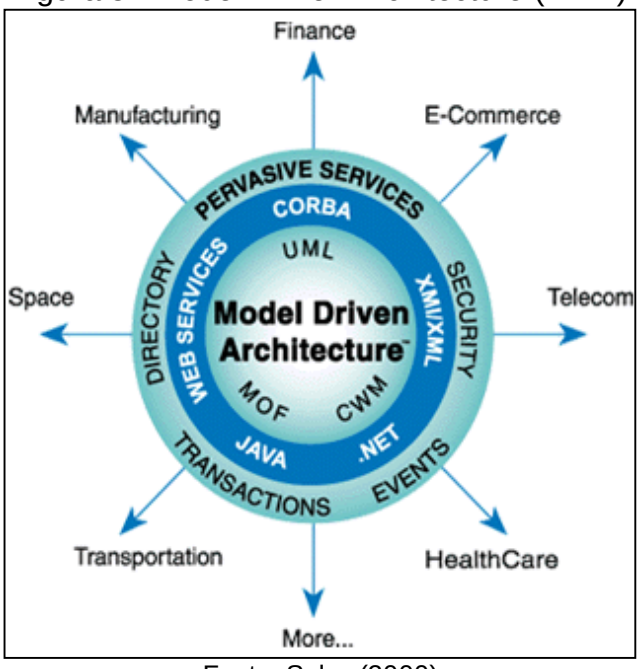
- Estabelece a definição de termos, ícones e notações que contribuem na compreensão de um modelo.

- Disponibiliza a base para modelos de dados semânticos a serem gerenciados.
- Disponibiliza biblioteca de modelos reutilizáveis, tais como regras, modelos de objetos de negócios ou padrões de projetos de arquitetura.

Essa automatização gerada pela MDA reduz os custos e o tempo da realização do sistema como um todo, até mesmo beneficiando os processos de manutenções futuras, garantindo a consistência entre o modelo e o código gerado. Sendo assim, essa abordagem torna a geração de código mais rápida e confiável. Mas para efetuar essa geração automática de código, é importante que o modelo seja bem detalhado, assim pode ser executado, e o código fonte é interpretado. Desta maneira uma linguagem de programação é produzida, de acordo com o especificado no código intermediário encontrado entre o modelo e a linguagem de programação gerada (OMG, 2014).

A Figura 5 representa a arquitetura da MDA, no centro da imagem é possível identificar os padrões de modelagem da *Object Management Group* (OMG), sendo eles a UML, *Meta-Object Facility* (MOF) e *Common Warehouse Metamodel* (CWM), a camada em volta dos modelos representa a linguagem de programação a ser gerada, como no exemplo, estas linguagens são representadas pelo *Java*, *Corba*, .net, *eXtensible Markup Language* (XML) e *Web*. As flechas simbolizam os artefatos gerados pela transformação do modelo escolhido em código (SOLEY, 2000).

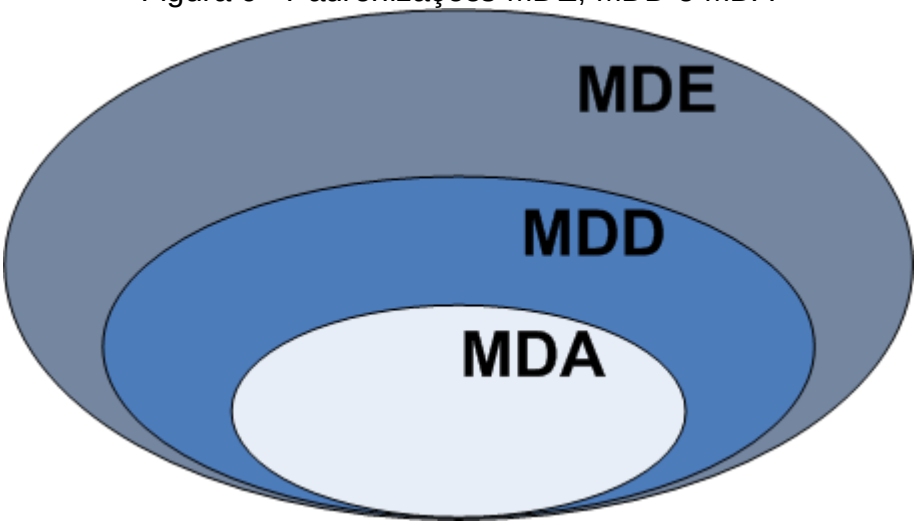
Figura 5 - *Model-Driven Architecture* (MDA)



Fonte: Soley (2000)

Para entender melhor o padrão MDA, a Figura 6 apresenta os conjuntos de padronizações que trabalham com esse processo de transformação de modelo em código.

Figura 6 - Padronizações MDE, MDD e MDA



Fonte: LANGUAGES (2014)

O padrão que engloba todos os outros neste método de interpretação de modelos é o *Model-Driven Engineering* (MDE), diferente do MDA, esse padrão submete outras atividades entorno da engenharia de software, suas tarefas não são tão especificas, já que esta padronização visa também modelos de processos completos, como por exemplo, efetua a engenharia reversa de um sistema legado (LANGUAGES, 2014).

Desta forma, MDD é um processo de desenvolvimento que possui como comportamento utilizar modelos para efetuar a geração de código automaticamente, seja parcialmente ou gerar o sistema como um todo. Sendo assim, fica mais claro que a MDA é apenas uma linha de pensamento da OMG a respeito do processo MDD. Provando que a OMG utilizou um padrão já existente para criar sua própria padronização de uma forma mais compreensível (LANGUAGES, 2014).

2.4 UML EXECUTÁVEL

A UML Executável consiste na utilização de um compilador de modelos para transformar modelos UML em código fonte, se tornando assim um sistema. Basicamente sua principal função é interpretar um ou mais modelos UML e converter

estes modelos visuais em uma plataforma de programação. Para realizar esta conversão não é necessária a utilização do padrão UML por completo, já que muitos elementos básicos da UML são considerados irrelevantes, desta forma, não são utilizados (FOWLER, 2005).

Sendo assim, pode-se dizer que a UML Executável se trata de uma solução baseada em modelo evolutivo para expressar o software, ao invés de elaborar um modelo de diagramação UML e depois criar toda a codificação de forma manual em cima deste modelo. Os desenvolvedores irão desfrutar de ferramentas que traduzirá os modelos de desenhos visuais em código de forma automática, construindo assim entidades executáveis. Portanto, o trabalho em si desenvolvido pelo encarregado da tarefa, é apenas da criação do modelo, a transformação deste modelo em código é apoiada por ferramentas de UML Executável (JIANG; ZHANG; MIYAKE, 2007).

Com esta transformação automática, o desenvolvedor não se preocupa se existe incoerência entre o modelo e o código, algo que acontece muito quando se trata do processo de desenvolvimento tradicional, já que as ferramentas fazem sua parte de forma totalmente automática e coerente com o modelo, sendo traduzível para várias implementações e uma vasta quantidade de linguagens de programação. Este processo de automatizar uma tarefa normalmente trabalhosa, facilita e faz com que aumente significativamente a reutilização de código, além de reduzir de forma notável o custo do desenvolvimento de software (JIANG; ZHANG; MIYAKE, 2007).

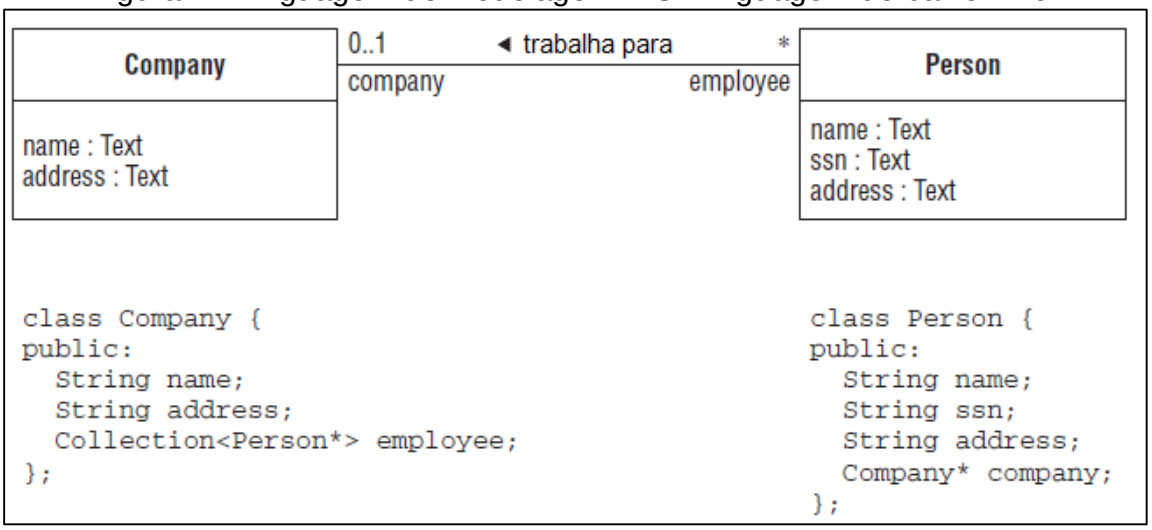
Antes mesmo de surgir a UML Executável, durante vários anos a etapa de geração de código não era um procedimento totalmente automático, algumas ferramentas UML geravam apenas uma parte do código e o restante devia ser implementado manualmente para assim poder completar todo o código a ser compilado. O grande problema desta técnica de complementar o código de forma manual com a intenção de corrigi-lo, melhorá-lo ou otimizá-lo se prende exatamente a essa questão da incoerência estabelecida entre o modelo e o código gerado (CHARFI; MRAIDHA; BOULET, 2012).

No entanto, apesar da UML Executável aparentar ser uma solução imediata para este problema de inconsistência entre o modelo e o código, sendo uma tecnologia que surgiu para facilitar a automatização deste procedimento que até então era responsabilidade do desenvolvedor, a mesma pode trazer algumas limitações, como impedir otimizações relacionadas a semântica da linguagem de modelagem UML. Isto faz com que detalhes semânticos se percam no exato momento da

transformação do modelo em código. Essa perda ocorre por conta das linguagens de programação estarem em um nível mais baixo de abstração, não fornecendo assim os mesmos conceitos das linguagens de modelagem. Por este motivo, para resolver esta deficiência que ocorre no exato momento da geração do código, novas abordagens que compilam diretamente no modelo vêm sendo estudadas com o intuito de solucionar estas lacunas ainda existentes (CHARFI; MRAIDHA; BOULET, 2012).

Na Figura 7 pode-se identificar inicialmente a representação por meio de diagramação de uma linguagem de modelagem visual altamente abstrata. Após a transformação desse modelo em código, pode ser visualizado um modelo textual em baixo nível, neste exemplo gerado na linguagem *Java*. Ambos os modelos representam a descrição de uma pessoa (nome, número de segurança social e endereço). Conforme indicado pelo relacionamento apresentado, uma pessoa pode trabalhar para uma empresa (descrita com nome e endereço), e uma empresa pode empregar muitas pessoas (MILICEV, 2009).

Figura 7 - Linguagem de modelagem V.S. Linguagem de baixo nível



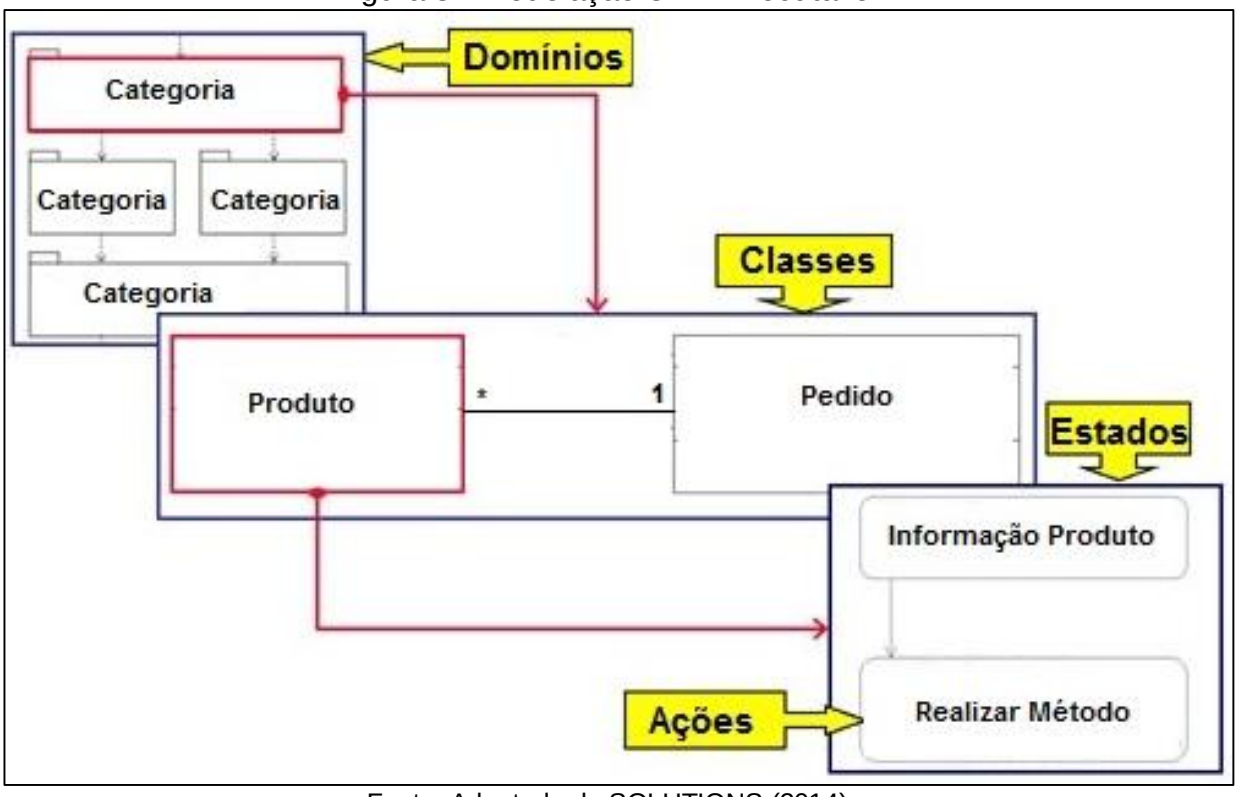
Fonte: MILICEV (2009)

No exato momento da transformação do modelo visual para o modelo textual, muito se fala na perda que pode ocorrer com os detalhes semânticos da linguagem de modelagem, além das restrições que são complexas e impossíveis de serem expressadas adequadamente em representações diagramáticas. Para minimizar este problema foi criada a linguagem *Object Constraint Language* (OCL). A OCL é puramente textual e descreve regras aplicáveis aos modelos UML, foi desenvolvida pela *International Business Machines* (IBM) mas passou a fazer parte do padrão UML.

Esta linguagem possibilita a aplicação de restrições em um modelo que não pode ser especificado por meio de diagramas, implicando em pré e pós-condições de comportamento, modificando o estado do sistema, além de suportar algumas funções computacionais básicas fazendo assim um complemento de regras a estes modelos UML (ALMEIDA, 2006).

Diversas abordagens e meios de utilização da UML Executável vem sendo estudados, formas distintas e de executar modelos gráficos de diagramas vem sendo discutidas para efetuar com precisão a transformação integral ou parcial deste em uma linguagem de programação. Este método que elimina a necessidade de programar o sistema de software pode ser construído por meio da modelação primária da UML Executável. Como pode ser notado na ilustração da Figura 8, que se inicia com cada sistema dividido em domínios, representando áreas de especialização, cada domínio é dividido em classes, que juntas vão cumprir todas as exigências de cada domínio. Individualmente cada classe pode possuir uma máquina de estados, que executa todas as ações dos estados, e para finalizar cada classe pode possuir operações para realizaram o processamento (SOLUTIONS, 2014).

Figura 8 - Modelação UML Executável



Fonte: Adaptado de SOLUTIONS (2014)

Ainda com a UML 2.0 em processo de finalização a OMG já estava pensando em lançar uma linguagem semântica para modelos executáveis. Foi quando surgiu a *Foundational UML* (fUML) com foco em abranger a modelagem orientada a objetos. Segundo a OMG (2013), a notação da modelagem visual UML não foi criada para utilizar tal nível de detalhamento semântico, por este motivo que grande parte das ferramentas comerciais que executam modelos fornecem algum tipo de linguagem para especificação do comportamento detalhado. Desta forma, a linguagem fUML ou *Action Language Foundational* (Alf) como é conhecida, é uma notação textual para o comportamento UML, esta notação torna mais fácil para o usuário a especificação no contexto de um modelo UML, ao invés de representar as ações em um modelo visual.

A relevância da linguagem Alf pode ser representada por meio do seguinte exemplo: Suponha-se que em um diagrama de classes, exista um relacionamento entre as classes cliente e conta, que todos os saldos de todas as contas de um cliente especifico serão somadas, utilizando *Java* como linguagem de programação para tratar este campo é possível obter um código como (CABOT, 2011):

```
Integer totalBalance = 0;
for (Account account: myCustomer.accounts) {
for (Integer balance: account.balance) {
totalBalance += balance;
  }
}
```

O problema neste trecho de código se dá pois, "*myCustomer.accounts*" devolve um objeto em *Java* que é dissonante com a semântica UML, tipos de decisões de programação devem ser tomadas, diferentes aspectos podem ser utilizados para obter o resultado, podendo utilizar por exemplo um *ArrayList* ou um vetor. Por isso é necessário utilizar uma linguagem de ação como a Alf, que possua o mesmo nível semântico que a UML, padronizando esta execução que poderia ser realizada de diversas maneiras seja em *Java*, C, .Net. Além da integração semântica com a UML, a linguagem Alf fornece algumas vantagens adicionais (CABOT, 2011), como pode ser visto pelo código abaixo.

```
totalBalance = 0;
for (balance in myCustomer.accounts.balance) {
totalBalance += balance;
}
```

O comando "*myCustomer.accounts.balance*" faz com que exista uma navegação entre as associações das contas retornando todos os objetos integrados. A linguagem Alf inclui recursos poderosos como a filtragem e mapeamento, que é realizado pelas linguagens tradicionais conhecidas, desta forma o ciclo acima pode ser escrito de forma mais compacta (CABOT, 2011).

```
myCustomer.accounts.balance -> reduzir ?? + ??;
```

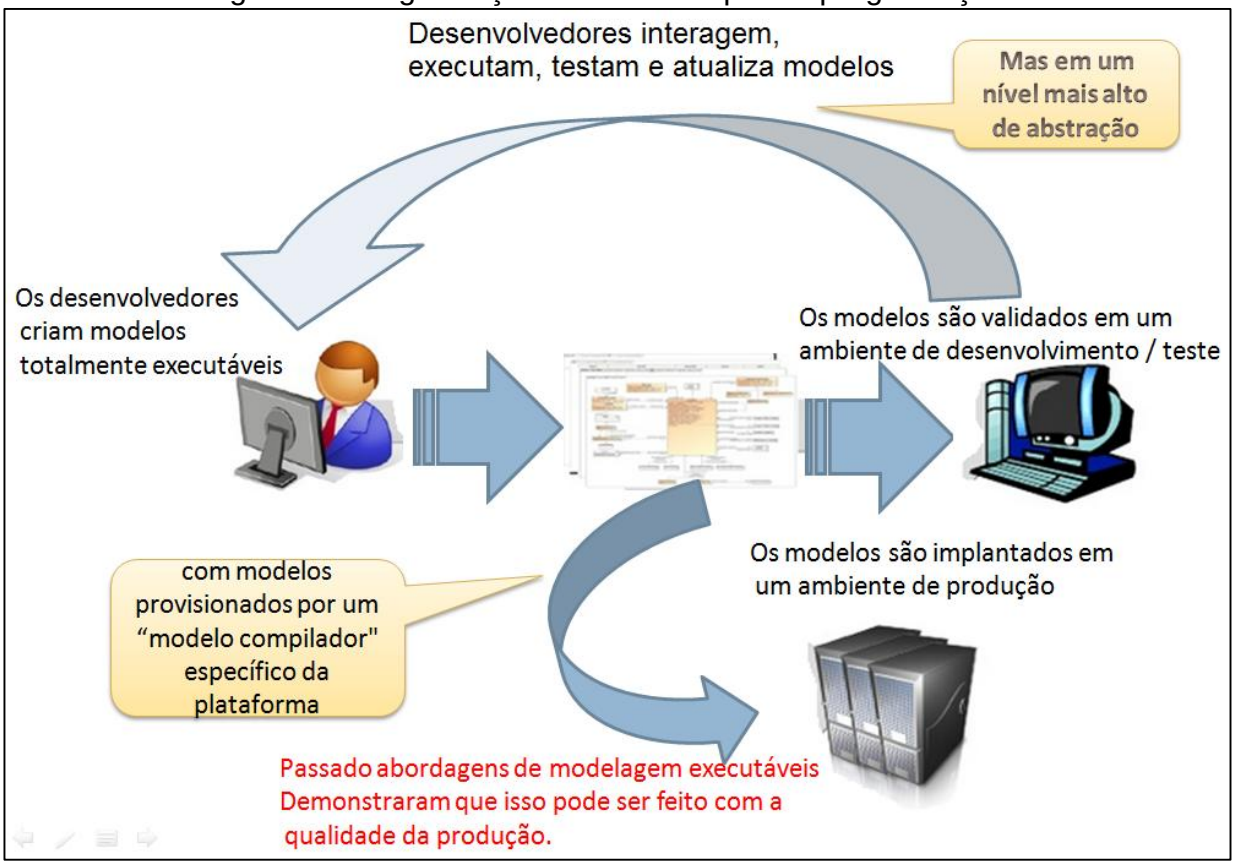
Sendo assim a expressão é mapeada para reduzir à uma única ação UML, além disso o "*myCustomer* " pode ser selecionado com base no endereço de e-mail, a partir dos clientes existentes (CABOT, 2011).

```
myCustomer = Cliente -> selecione c (c.email == myCustomerEmail);
```

A linguagem de ação Alf permite maior liberdade na execução do comportamento especificado no nível mais alto de abstração imposta pela semântica de atividade UML em plataformas de implementações distintas. A linguagem Alf oferece uma sintaxe detalhada que pode ser bem aproveitada em alguns casos como permissão textual UML quando necessária, digitação de dois pontos para qualificação de nome, inclui a utilização de caracteres especiais, fornece um sistema de nomenclatura para referenciar elementos fora de uma atividade, além de possuir expressividade de OCL no uso de manipulação de valores. Apesar do objetivo principal da Alf ser uma linguagem de ação, a mesma disponibiliza uma sintaxe concreta para modelagem estrutural (CABOT, 2011).

De acordo com a Figura 9, programação em UML é apenas programação, a linguagem desenvolvida no padrão MDD deve ser tratada como outra linguagem padronizada já existente, distinguindo o diagrama modelado com o que pode ser gerado por meio da UML Executável, utilizando um modelo de compilador que produz a saída de uma linguagem bem definida, destacando que pode ser programável da mesma forma que um desenvolvedor compila seu código criado manualmente, mas desta vez elaborando modelos totalmente executáveis (OMG, 2013). Na Figura 9 é mostrado um processo a ser seguido para obter o resultado esperado.

Figura 9 - Programação em UML é apenas programação



Fonte: Adaptado de OMG (2013)

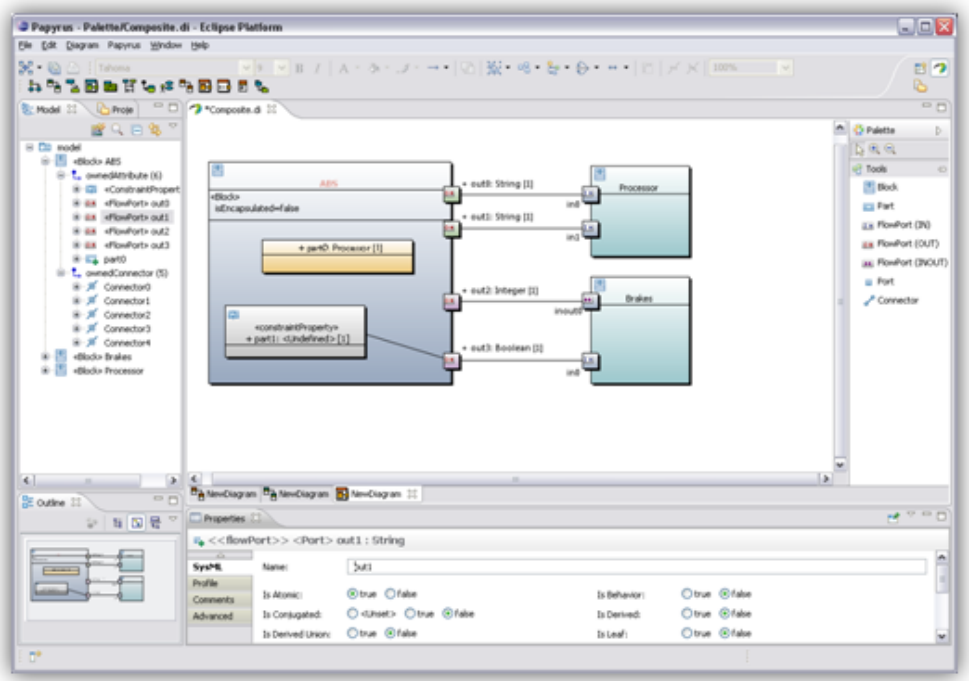
Apesar deste procedimento automatizado aparentar ser muito eficaz, existem alguns problemas críticos a serem destacados, visto que as ferramentas utilizadas para geração deste código de forma automática não funcionam como deveriam, e as abordagens capazes de realizar todo esse processo de transformação de maneira totalmente eficiente apresentam lacunas. Olhando de uma forma mais crítica, enquanto este método não realizar exatamente o que for imprescindível, mesmo que já consiga gerar algo proveitoso, talvez a programação manual seja a melhor saída, até uma ferramenta realmente eficaz provar ao contrário (FOWLER, 2005).

2.4.1 Ferramentas de UML Executável

De acordo com que são lançados novos padrões de UML Executável, novas ferramentas são desenvolvidas e disponibilizadas para que essas tecnologias possam ser desfrutadas. Esse novo procedimento de transformação de modelos UML em código fonte vem se tornando mais popular a cada dia (CABOT, 2011). Dentre as principais ferramentas que auxiliam no processo da UML Executável se destacam:

Papyrus (PAPYRUS, 2014): Esta ferramenta gratuita, apresentada na Figura 10, foi desenvolvida em 2008, se tornando logo em seguida componente oficial para plataforma *Eclipse*. É conhecida por ser um modelador gráfico de qualquer tipo de modelo EMF, apoiando linguagens de modelagens como *Systems Modeling Language* (SysML) e UML. Uma das principais características do *Papyrus* se dá ao conjunto de mecanismos de personalização dos diagramas UML, oferecendo aos usuários recursos necessários para criar e modificar todos os diagramas do padrão UML. Este editor de diagramas é gráfico e textual, sendo assim, pode-se editar elementos do modelo utilizando editores de texto que possibilitam a inclusão de sintaxe. Por ser um editor de diagramas e não produzir nenhum tipo de código, este trabalha em conjunto com outros componentes, como por exemplo o *Acceleo*.

Figura 10 - Ferramenta *Papyrus*

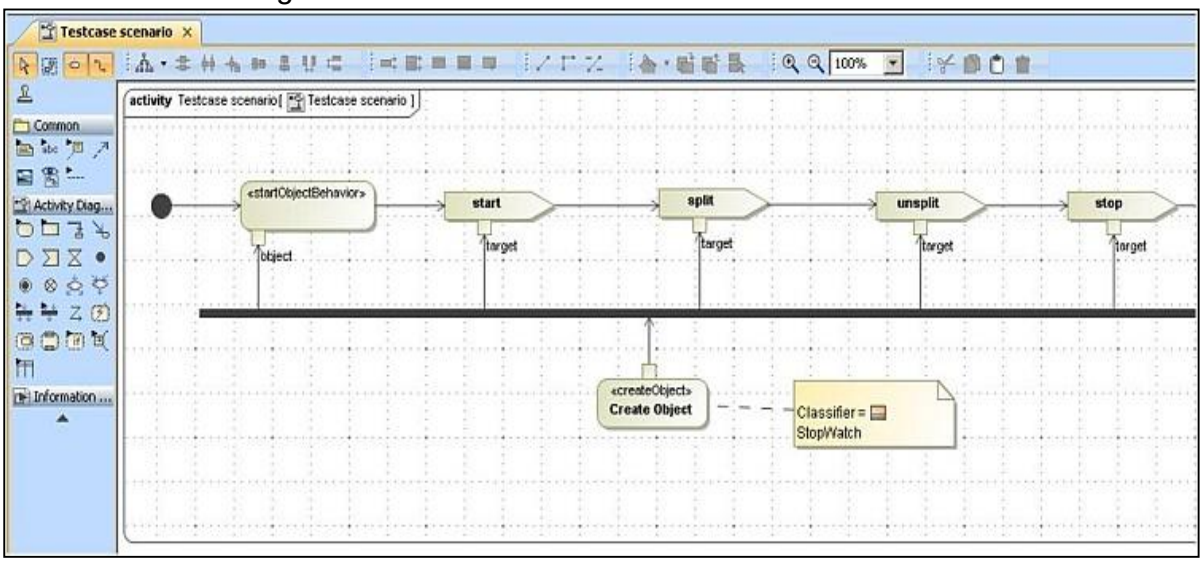


Fonte: PAPYRUS (2014)

Cameo Simulation Toolkit (NOMAGIC, 2014): O *Cameo Simulation Toolkit* se trata de um *plugin* comercial, sua principal funcionalidade é depurar modelos SysML e UML. Este *plugin* disponibiliza para o usuário a função de como o sistema reage a interação do usuário, exporta máquina de estado UML para arquivo de formato *Stands for State Chart eXtensible Markup Language* (SCXML), e executa casos de cenários de testes *model-driven.* Possui suporte para atividades e ações semânticas de elementos da UML 2, incluindo objetos e controles de fluxo. Na Figura 11 é mostrado

na prática a execução de uma sequência de eventos que são criados por meio de um diagrama de sequência, no qual o cenário é preparado para testes automatizados.

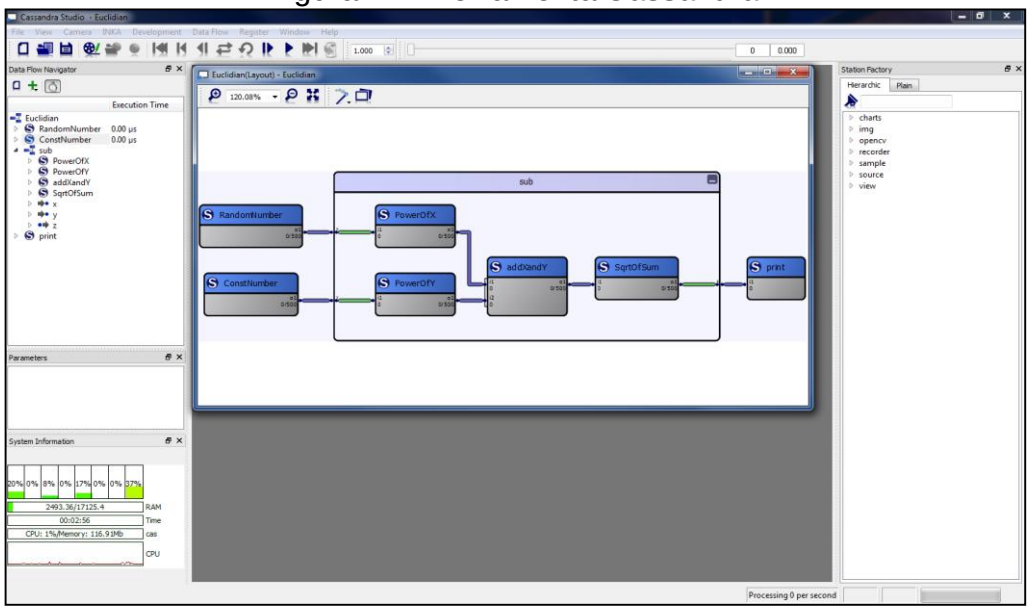
Figura 11 - Ferramenta *Cameo Simulation Toolkit*



Fonte: Nomagic (2014)

Cassandra (KNOWGRAVITY, 2014): O Cassandra, apresentado na Figura 12, é uma plataforma gratuita que analisa os dados de projetos UML, gera perguntas associadas ao modelo e propõe quais devem ser as etapas seguintes a serem executadas. Possui a função de simular modelos de UML, suportando suas semânticas. Esta ferramenta é focada principalmente no modelo de casos de uso, disponibilizando as opções de herança e comportamento, operações temporais, conjuntos de regras, tais como persistência e transações do modelo.

Figura 12 - Ferramenta *Cassandra*

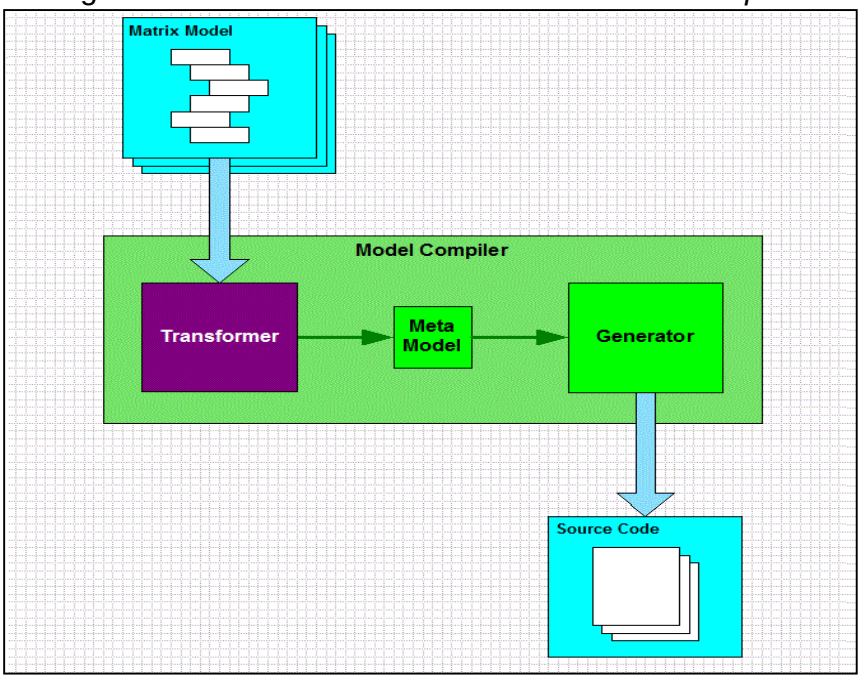


Fonte: Knowgravity (2014)

Matrix - Free Model Compiler (MATRIX, 2014): *Matrix* é uma nova linguagem de modelagem textual que foi projetada para traduzir modelos de diagramas em linguagens de programação, tais como C e *Java*. Seu compilador pode ser adquirido de forma gratuita, as semânticas *Matrix* foram criadas para lidar especialmente com conceitos detalhados de modelagem. No processo de desenvolvimento da ferramenta *Matrix*, o código fonte é gerado a partir de um modelo, o mesmo deve estar livre de detalhes de implementação, já que este é adicionado de forma automática pelo compilador no exato momento de sua transformação.

A função do compilador *Matrix* se inicia com a transformação de um modelo de diagramação UML criado em qualquer ferramenta de modelagem gráfica, em um modelo textual *Matrix* com suas respectivas notações, fazendo assim o mapeamento completo e significativo para linguagem *Matrix*. Na Figura 13, pode-se notar como é realizado esse procedimento de geração de código de forma automática pelo compilador *Matrix* (MATRIX, 2014).

Figura 13 - Ferramenta *Matrix - Free Model Compiler*

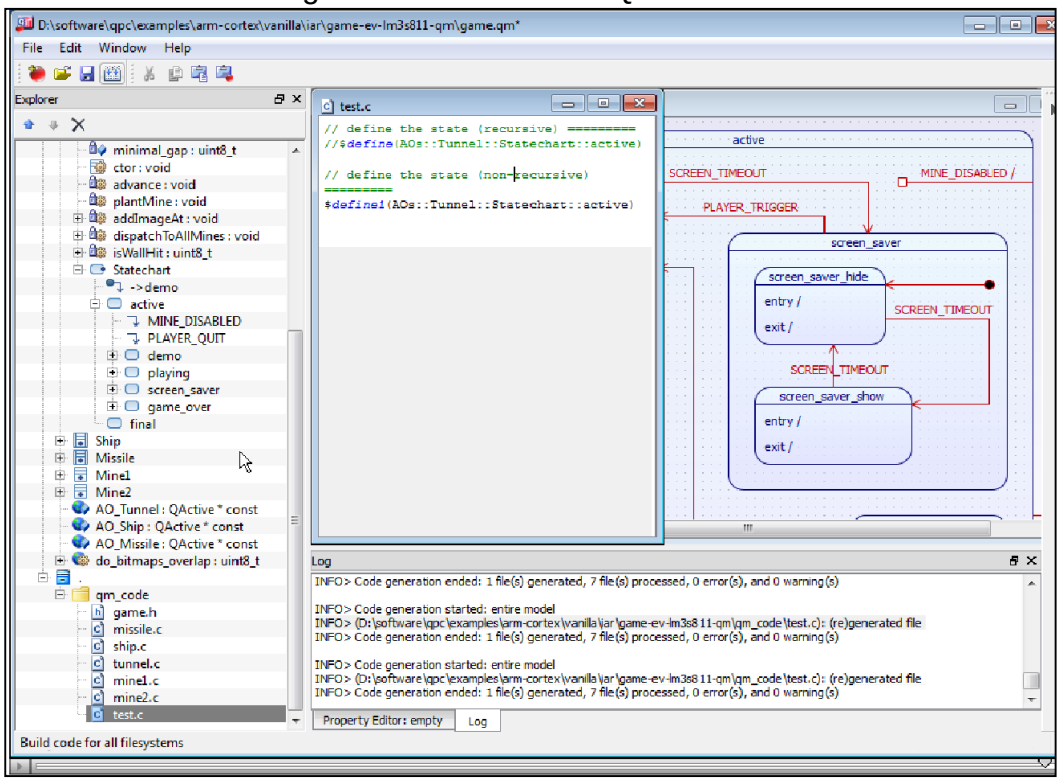


Fonte: MATRIX (2014)

QP Modeler (QUANTUM, 2014): Esta é uma ferramenta de modelagem gráfica gratuita, apresentada na Figura 14, é baseada especialmente em diagramas de máquinas de estados UML, focada em implementação de software embarcado em tempo real. Além e ser uma ferramenta para modelagem, disponibiliza a função para geração de código automaticamente para as linguagens C e C++. Atualmente a

ferramenta QM está disponível para *Windows*, *Linux* e *Mac OS X* (QUANTUM, 2014). Seu maior problema fica por conta da limitação de diagramas a ser trabalhados, já que esta ferramenta é focada apenas para diagrama de máquinas de estados não possibilitando que outros diagramas UML possam ser inclusos no projeto para modelagem do sistema.

Figura 14 - Ferramenta *QP Modeler*



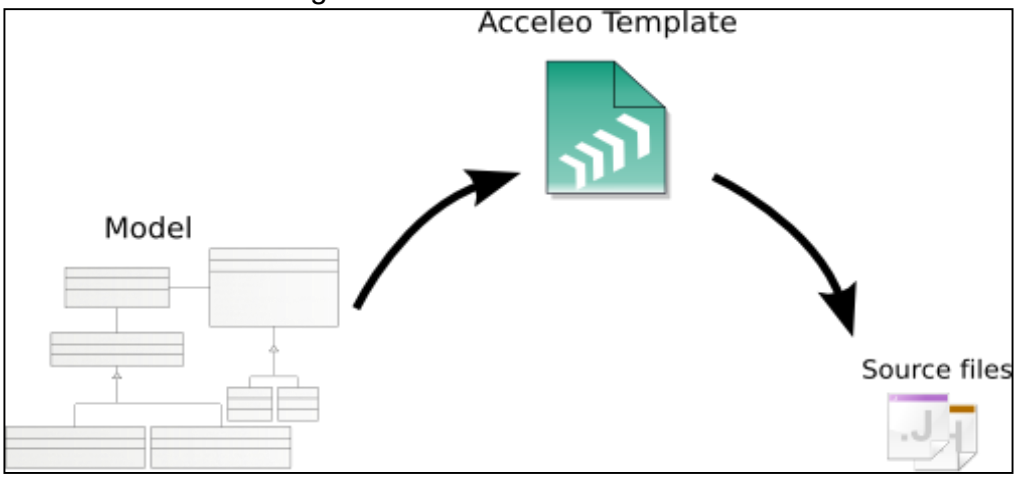
Fonte: QUANTUM (2014)

Acceleo (ACCELEO, 2014): O *Acceleo* é um *plugin* de código aberto com foco em melhora de desenvolvimento de software, sua abordagem enfatiza o processo MDA. Desenvolvido pela *Object Management Group* (OMG) para o *Eclipse*, a principal vantagem dessa ferramenta perante as demais fica por conta da facilidade de seu uso, já que a mesma fornece exemplos de geradores de códigos já prontos, existindo apenas a necessidade de importar um modelo de diagramação e executá-lo juntamente com o gerador de código para iniciar a transformação do modelo nas linguagens *Java*, C, *Python*, entre outras.

O *Acceleo* possui como funcionalidade importar um modelo de diagrama seja ele UML ou SysML, e convertê-lo em alguma linguagem de programação textual, como mostrado na Figura 15. Desta forma este *plugin* não possui a opção de modelagem de diagramas, assim o usuário deve utilizar outras ferramentas de

modelagem, para poder criar a representação de um modelo visual para depois utilizá-lo no *Acceleo*. Por não possuir essa funcionalidade, o *Acceleo* permite a conexão com o *Papyrus* que é responsável em possuir todos os recursos necessário para elaboração de qualquer modelo de diagrama (ACCELEO, 2014).

Figura 15 - Ferramenta *Acceleo*

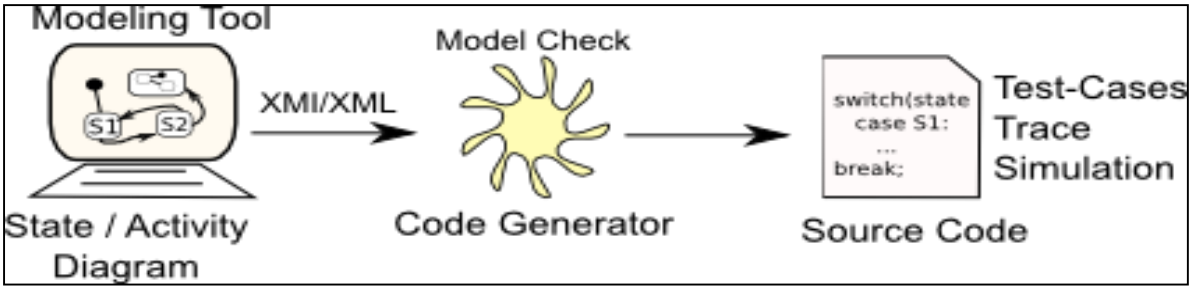


Fonte: ACCELEO (2014)

Sinelabore (2014): A ferramenta *Sinelabore* foi desenvolvida com foco em sistemas embarcado, sua funcionalidade é a geração de código por meio de diagramas de máquinas de estados e atividades. A ferramenta importa um modelo de máquina de estados e converte de forma automática para uma linguagem de programação textual. O gerador suportar os principais recursos do diagrama, como estados hierárquicos, regiões história, sub-máquinas, entre outros recursos.

O código gerado é baseado em declarações de fácil entendimento como switch/case e if/else. Como apresentado na Figura 16, esta ferramenta possibilita que seja gerado partes do sistema por meio de modelos, dessa forma, o restante da implementação do sistema pode ser feita de forma manual, descartando o conceito generalizado de transformar o modelo integralmente em um sistema, sendo assim prioriza a utilização do programador (SINELABORE, 2014).

Figura 16 - Ferramenta *Sinelabore*

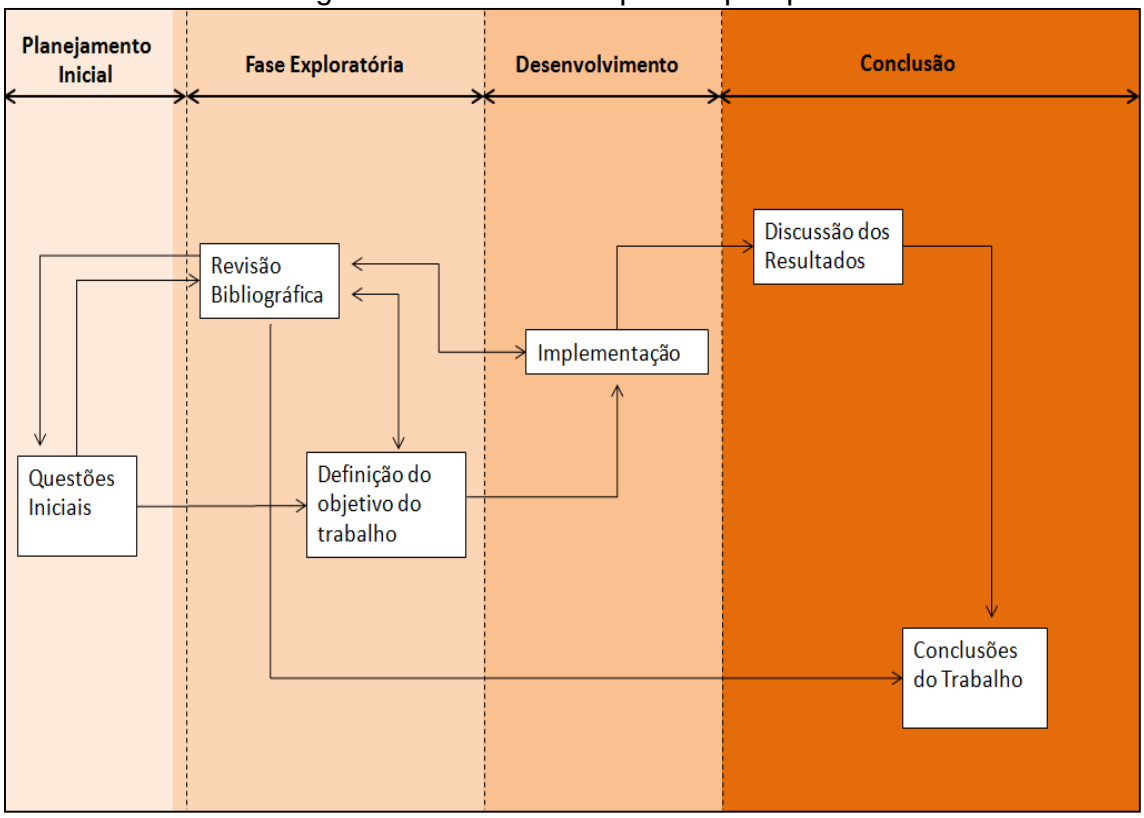


Fonte: SINELABORE (2014)

3 METODOLOGIA

Foi definido uma estrutura de pesquisa a ser seguida para cumprir com os objetivos iniciais, como pode ser visto na Figura 17. Esta estrutura é basicamente composta por quatro fases (Planejamento Inicial, Fase exploratória, Desenvolvimento e Conclusão), o qual estas fases podem possuir uma ou mais etapas que são ligadas entre si por meio de setas, que indicam a direção do fluxo e da ação decorrente.

Figura 17 - Fases e etapas da pesquisa



Fonte: da pesquisa (2015)

3.1 PLANEJAMENTO INICIAL

Após definir o campo de pesquisa para o trabalho, algumas questões iniciais foram abordadas para assim dar início ao estudo, tais como: "O que existe de MDD com UML Executável?", "Qual a viabilidade de se utilizar a o MDD?", "Como é realizado o processo de gerar código automaticamente?", "Quais as ferramentas que auxiliam neste processo?", "Quais diagramas UML podem ser utilizados?", "Em que dimensão o MDD é capaz de contribuir para engenharia de software?", "O desenvolvimento tradicional ainda é a melhor opção?".

No entanto, as questões iniciais levantadas servem apenas para traçar uma trajetória a ser seguida, desta forma, contribui para um direcionamento que possa enfatizar a pesquisa proposta. Depois de definir estas questões iniciais estabelecidas, pode-se, delimitar a área de estudo e o objetivo a ser tratado, porém para criar objetivos e concretizar a pesquisa, é fundamental explorar o campo de estudo, iniciando a próxima fase do trabalho, a exploratória.

3.2 FASE EXPLORATÓRIA

O principal objetivo da fase exploratória é proporcionar maior familiaridade com o problema (tornando-o explicito), podendo assim efetuar uma revisão bibliográfica aprofundada e bem detalhada sobre o assunto proposto, para assim, conseguir definir todos os objetivos do trabalho que ainda eram vagos na fase anterior, deixando-os compreensível, determinando o que de fato será realizado durante toda a diligência (GIL, 2002). Sendo assim, é necessário explorar o objetivo de estudo para conduzir o trabalho.

3.3 DESENVOLVIMENTO

O desenvolvimento é a fase que engloba todas as questões iniciais abordadas no planejamento inicial e todo o conhecimento, materiais e idéias levantadas na revisão bibliográfica que foi executada na fase exploratória, para possibilitar responder estas questões propostas.

Para atingir os objetivos apresentados neste trabalho, o desenvolvimento é dividido em duas etapas subsequentes e dependentes entre si, sendo elas: Implementação das abordagens escolhidas e a discussão dos meios utilizados para execução da primeira etapa e suas viabilidades.

Na implementação das abordagens escolhidas, será desenvolvido um ambiente relativamente simples, no qual transformará um diagrama UML em código *Java* e outros modelos em diferentes linguagens de programação, tudo de forma automática.

A partir do momento em que o código foi gerado automaticamente por meio do modelo, se dá início a segunda etapa, na qual haverá uma discussão para definir

qual a melhor abordagem que deve ser aplicada em cada caso, já que diferentes métodos de transformação de modelos em código foram realizados.

3.3.1 Conclusão

Por fim, a última fase apresentada na Figura 17 é a conclusão do trabalho, esta fase é dividida em duas etapas:

- Discussão dos resultados: é realizada uma análise e comparação entre o que foi gerado pelo metamodelo e o que foi produzido a partir da UML Executável, com o principal objetivo de identificar qual abordagem é a mais viável de acordo com cada situação, se tratando de produção de códigos automaticamente por meio de modelos.
- Conclusões do trabalho: após o levantamento geral da pesquisa, com todas as informações relevantes coletadas e compreendidas na fase de estudo do tema proposto, os pontos positivos e negativos das abordagens tratadas são destacados. Sendo assim, pode-se chegar a conclusão do trabalho.

4 DESENVOLVIMENTO

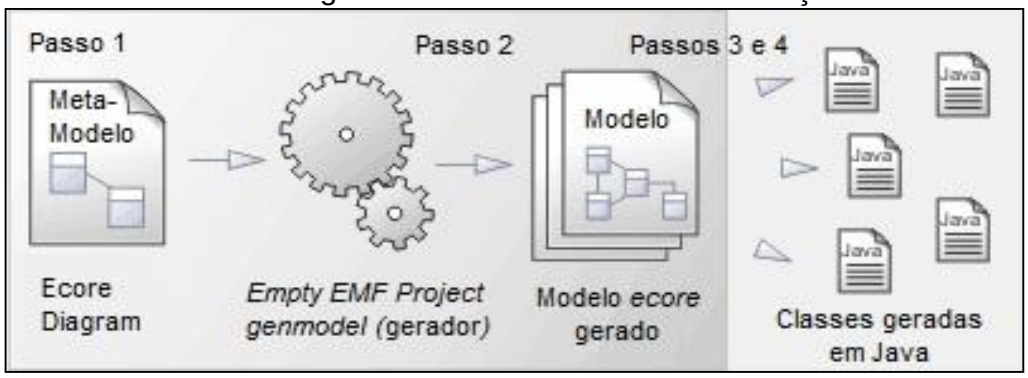
No desenvolvimento da pesquisa foram utilizadas diferentes abordagens para geração automática de código, tais como, geração a partir de metamodelo (EMF) com suas etapas apresentadas na seção 4.1, abordagem para geração de código por meio de um modelo (UML) com suas etapas demonstradas na seção 4.2, e por fim, a geração de código por meio de modelos aplicando regras de negócio conforme as etapas descritas na seção 4.3, definindo a linguagem *Model to Text* (M2T) como padrão para especificação dos geradores. Nas seções abaixo é descrito como cada abordagem foi implementada.

4.1 DESENVOLVIMENTO A PARTIR DE UM METAMODELO

O desenvolvimento dirigido a modelos, pode ser em muitos casos iniciado a partir de um metamodelo, que contém as regras, especificações, e informações relevantes que construirão os modelos. De certa forma, é comum a existência de um metamodelo por trás das cenas que decidirá o que é possível de ser realizado por meio de determinado modelo. Segundo Guedes (2012), o metamodelo define uma linguagem para expressar modelos, sendo mais compacto que um modelo que ele descreve.

Com a intenção de exemplificar na prática, todo o processo de elaboração até a transformação de um metamodelo em código, foram utilizados dois *plugins* da ferramenta *Eclipse Modeling Project,* além da criação de um projeto em EMF. Inicialmente o *plugin Ecore Diagram* foi utilizado para efetuar a modelagem do metamodelo, em seguida foi criado o *Empty EMF Project* para a transformação do metamodelo em modelo, e por fim, o *plugin Acceleo* exerceu a função de produzir código *Java* automaticamente por meio do resultado desse modelo gerado pelo metamodelo. De acordo com a Figura 18, todo o desenvolvimento de geração de código a partir de um metamodelo ocorreu na respectiva ordem, seus passos são descritos nas seções abaixo.

Figura 18 - Processo de transformação

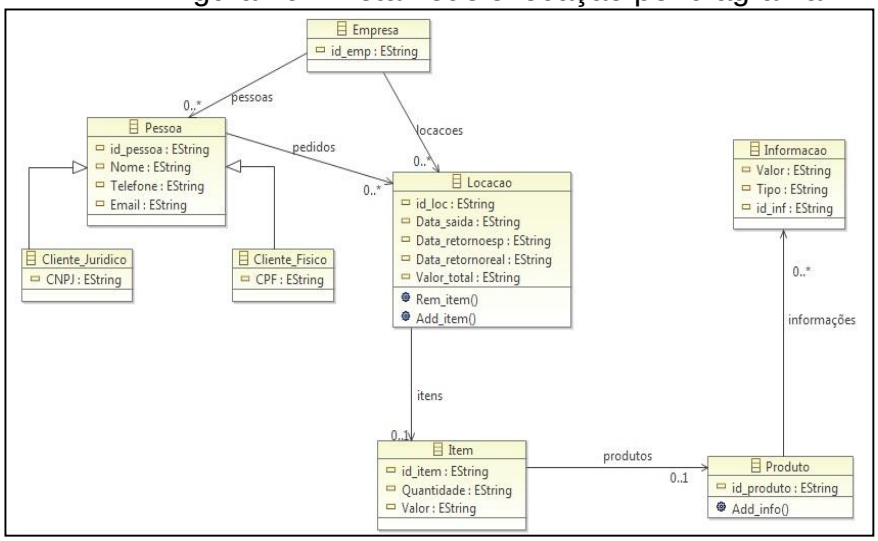


Fonte: Adaptado de SPARX SYSTEMS (2015)

4.1.1 Elaborando um Metamodelo

Para iniciar a criação do metamodelo representado como o Passo 1 na Figura 18, foi utilizado o *Ecore Diagram, plugin* que proporciona a criação de um modelo *ecore* de maneira amigável. Por meio do *Ecore Diagram,* é possível efetuar uma representação visual, estruturada e simplificada de um determinado conceito. Conforme pode-se observar na Figura 19, mediante *ao plugin Ecore Diagram* foi possível elaborar a modelagem desse metamodelo, que servirá para qualquer aplicação que envolva empréstimo, como por exemplo, locadora de filmes, locadora de veículos, e biblioteca. Dessa forma, o metamodelo gerado pode ser utilizado em diferentes situações que constam com a necessidade de realizar uma ou mais locações, podendo ser aplicado em setores distintos. Neste caso, o metamodelo engloba as entidades Empresa, Pessoa, Cliente_Juridico, Cliente_Fisico, Locação, Item, Produto e Informação.

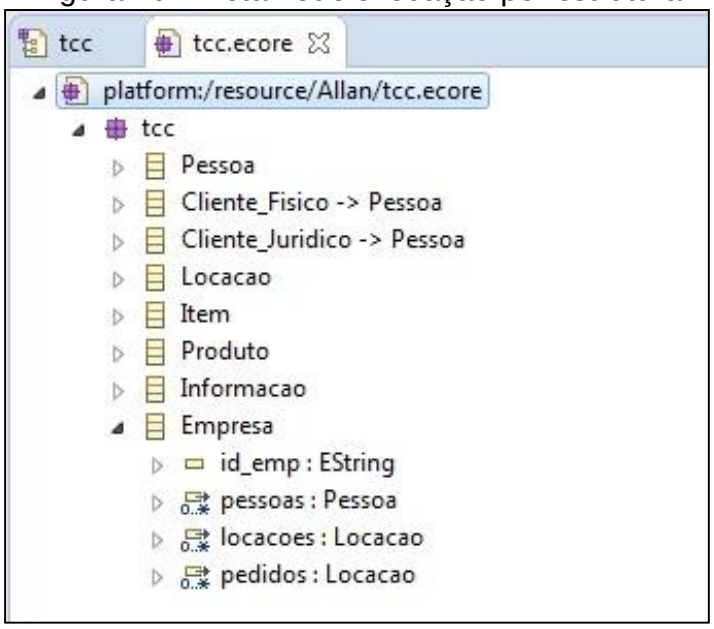
Figura 19 - Metamodelo locação por diagrama



Fonte: da pesquisa (2015)

A partir da modelagem do metadomodelo, desenvolvido no *Ecore Diagram,* conforme pode-se notar na Figura 20, o *plugin* automaticamente gera um outro arquivo *ecore*, com o mesmo conceito do modelo visual mas de forma estruturada. O metamodelo representado visualmente por meio do diagrama é associado com esse arquivo *ecore*, possuindo exatamente as mesmas propriedades, inclusive as suas classes, operações e relacionamentos. Sendo assim, esses dois arquivos *ecore* são vinculados, qualquer alteração realizada no diagrama é refletida simultaneamente no metamodelo *ecore* estruturado.

Figura 20 - Metamodelo locação por estrutura



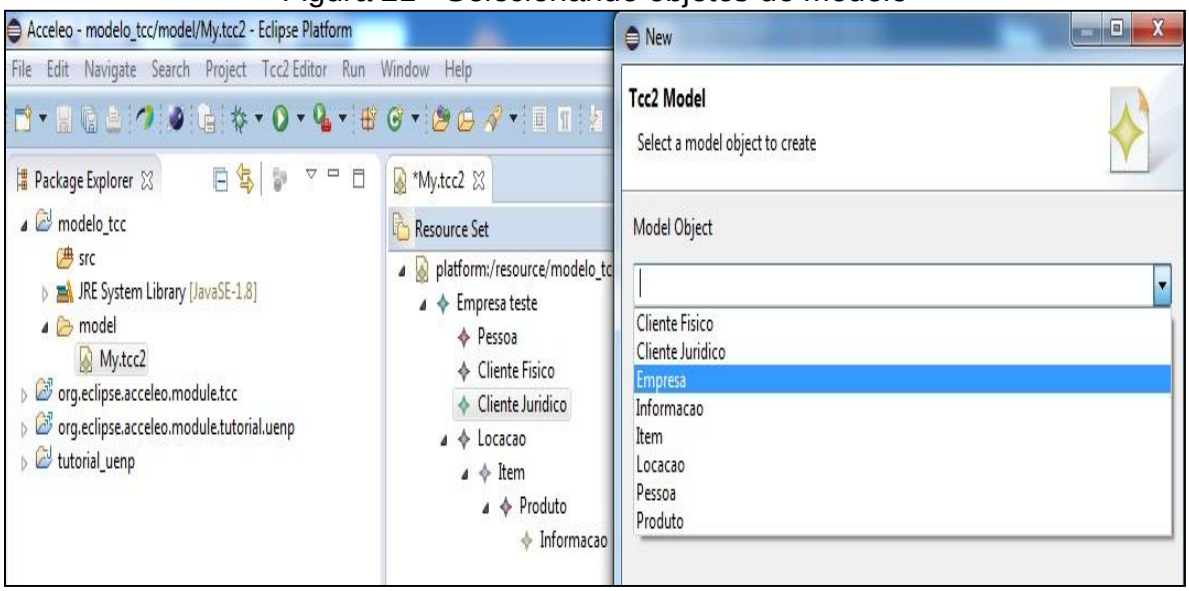
Fonte: da pesquisa (2015)

4.1.2 Transformando o Metamodelo em Modelo

Com o metamodelo elaborado, de acordo com o Passo 2 da Figura 18, é possível dar início a etapa de transformação do metamodelo em modelo. No entanto, para iniciar esse processo, é necessário que se crie um projeto EMF, que servirá como gerenciador desse metamodelo para transformá-lo em modelo. Sendo assim, foi criado um *Empty EMF Project* e dentro do mesmo uma pasta nomeada como *model*, na qual o metamodelo *ecore* estruturado foi inserido. A partir da pasta *model* dentro do *Empty EMF Project* é gerado o *genmodel*, tal como o gerador de modelo. Após executar o *genmodel*, automaticamente o modelo é gerado e o *Empty EMF Project* também deve ser executado, com isso o *Eclipse Application* será aberto, afim de promover uma nova etapa do processo de transformação.

O *Eclipse Application* é uma extensão que pode executar uma aplicação em concorrência com o *Eclipse*. Dessa forma, com o *Eclipse Application* inicializado e o modelo gerado, um novo *Acceleo Project* precisa ser originado. O modelo que foi gerado deve ser importado para a pasta *model* que está inclusa neste projeto que acabou de ser criado, com o intuito de iniciar a transformação do modelo em código. De acordo com a Figura 21, pode-se visualizar a opção de selecionar um objeto especifico do modelo, neste caso o objeto "Empresa" foi selecionado, sendo assim a entidade empresa importará todas as outras entidades que a mesma possui relacionamento e consequentemente outros objetos podem ser importados por meio dessas entidades desde que as mesmas possuam relacionamentos.

Figura 21 - Selecionando objetos do modelo



Fonte: da pesquisa (2015)

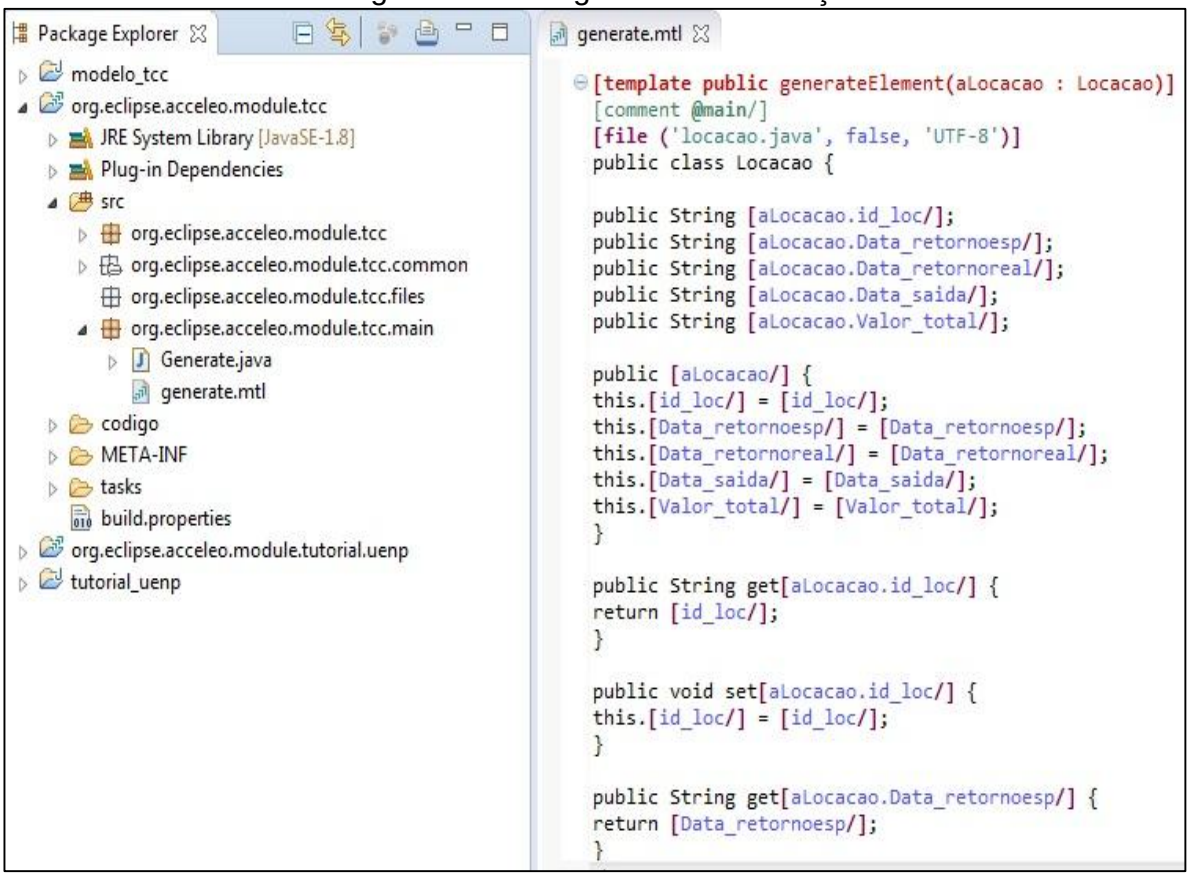
4.1.3 Transformando o Modelo em Código

A partir do modelo importado, de acordo com o Passo 3 da Figura 18, dá-se início a transformação do modelo em código por meio do *Acceleo Project*, neste caso o código a ser produzido é na linguagem *Java*, mas existe a possibilidade do *Acceleo* gerar código em outras linguagens de programação, como C, *Python*, linguagens *web*, entre outras. No próprio site do *Acceleo* é possível encontrar *scripts* que determinam em qual linguagem de programação gerar os códigos, além de permitir adicionar maiores detalhes na programação, como por exemplo, inserção de métodos.

Para iniciar a conversão do modelo visual em textual, é necessário editar o

arquivo gerador com o formato *Model to Text transformation* (MTL), responsável pela realização de todo processo, este arquivo gerador renomeado *generate.mtl* é acompanhado apenas pelo arquivo *Generate.java*. Sendo assim, é necessário executar o gerador *generate.mtl* que contém todo o código de transformação como apresentado na Figura 22.

Figura 22 - Código de transformação



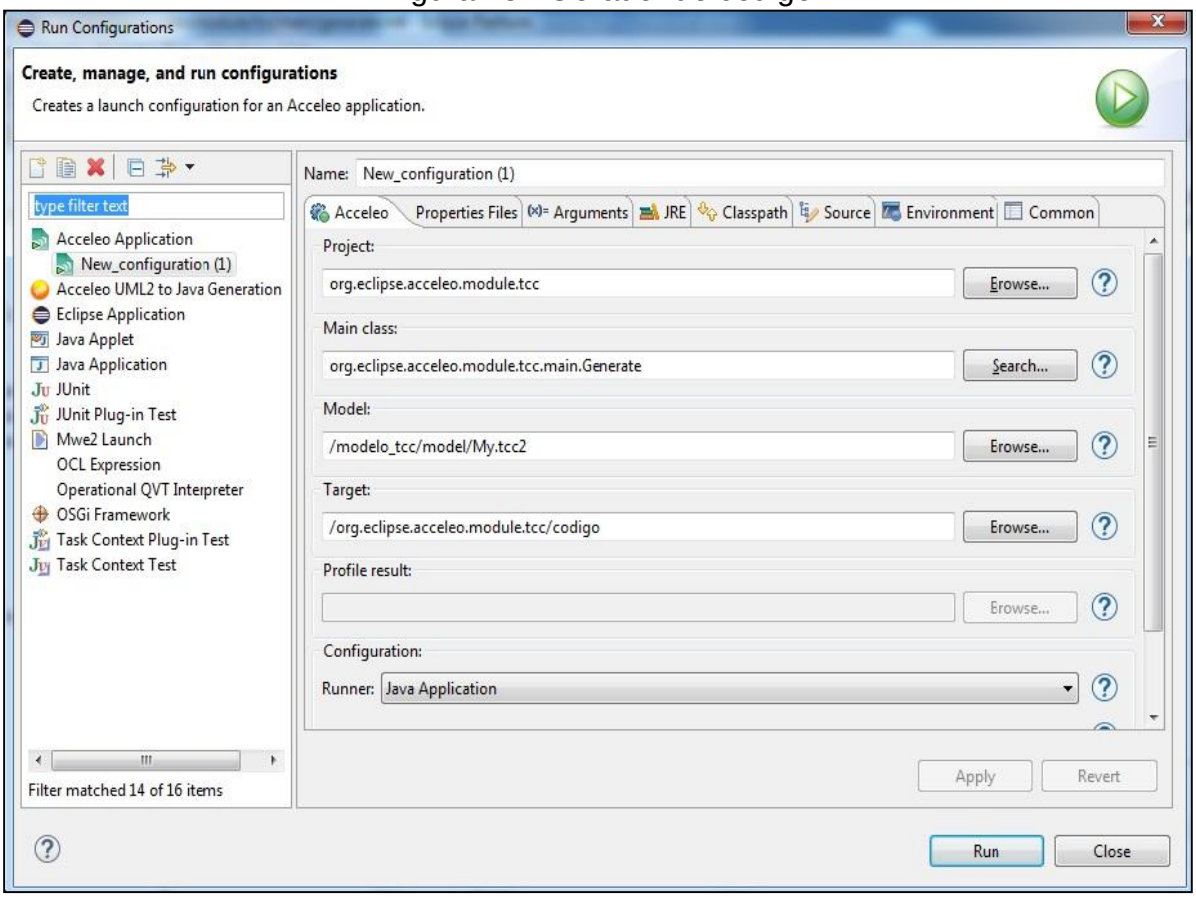
Fonte: da pesquisa (2015)

Neste arquivo no formato MTL o *script* é escrito em *Model to text transformation*, se trata de um padrão desenvolvimento pela OMG para realização do desenvolvimento na abordagem MDA. O padrão *Model to text transformation* tem como ênfase transformar um modelo em uma representação de texto, na qual uma abordagem baseada em modelo é utilizada. O determinado modelo especifica um texto com espaços reservados aos dados a serem extraídos, este espaços reservados são expressões especificadas para selecionar e extrair os dados de um diagrama, estes dados são convertidos em textos, utilizando uma linguagem de expressão que usufrui de uma biblioteca capaz de atender a exigência de transformações complexas (OMG, 2008).

4.1.4 Configurando o Gerador de Código

Seguindo a representação da Figura 18, conforme o arquivo *generate.mtl* for executado, o Passo 4 pode ser iniciado, contudo uma tela para configuração de saída do código será exibida de acordo com a Figura 23. Nesta tela há necessidade de configurar o caminho do projeto, o local que se encontra o arquivo principal *generate.mtl*, selecionar o modelo, e por fim definir a pasta de saída para o código gerado na linguagem *Java* a partir do modelo especificado.

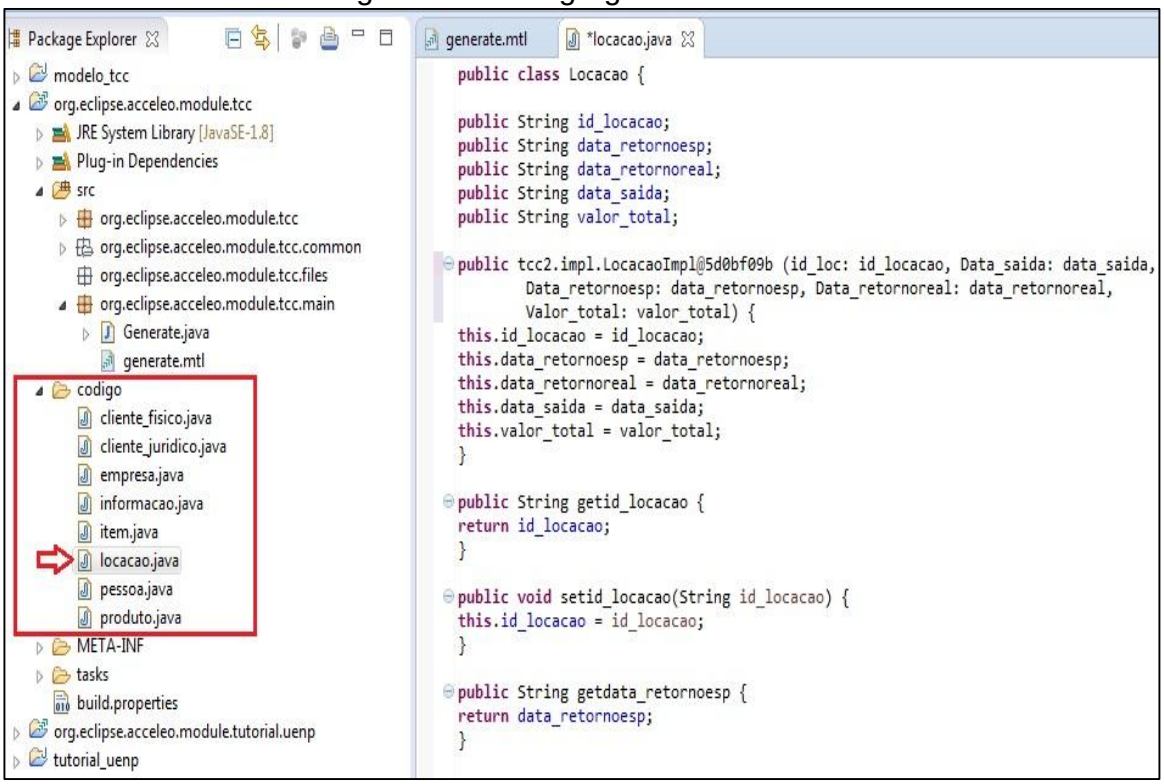
Figura 23 - Gerador de código



Fonte: da pesquisa (2015)

Assim que o *Acceleo Application* for configurado, com todos os respectivos campos obrigatórios preenchidos, basta aplicar as alterações e executá-las. Com o processo de transformação executado, em poucos segundos o código é gerado na "pasta alvo" definido no campo *Target*. Deste modo, é possível visualizar o que foi gerado na linguagem Java por meio do modelo em foco, conforme apresentado na Figura 24.

Figura 24 - Código gerado em Java



Fonte: da pesquisa (2015)

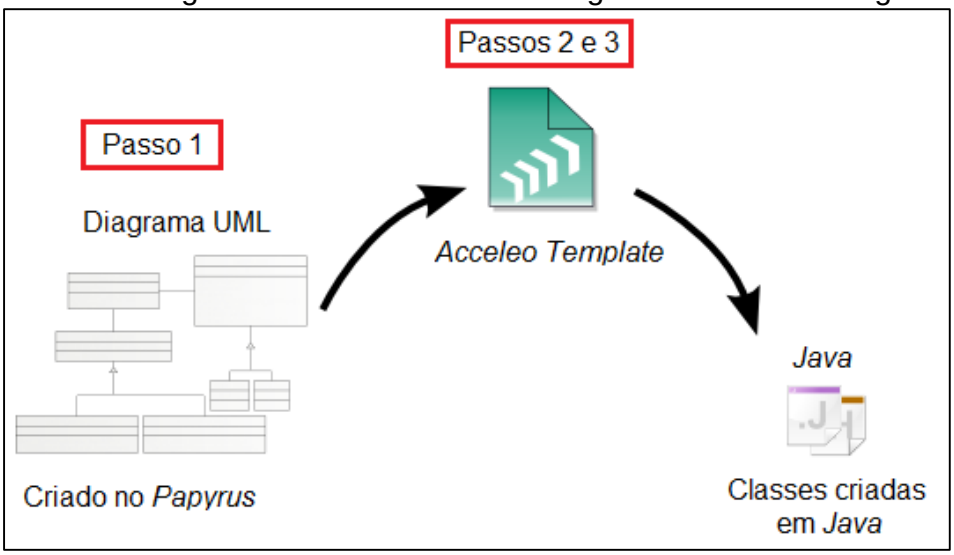
Futuras alterações podem ser realizadas no *template* e posteriormente novos códigos podem ser gerados, o resultado desta transformação varia de acordo com o que está descrito no arquivo MTL. Existem duas camadas, a do modelo, e acima a do *template*, escrito neste arquivo MTL gerador de código, então a camada do *template* que tomará a decisão. Portanto, após as devidas alterações serem realizadas basta executar novamente o gerador de código da mesma forma realizada na última etapa, para a realização do processo ser concluído.

4.2 DESENVOLVIMENTO A PARTIR DE UM MODELO UML

Como a proposta do trabalho é estudar o desenvolvimento dirigido a modelos, identificando suas principais abordagens, e utilizá-las para produzir código de forma automática, a abordagem de desenvolvimento a partir de um modelo UML, abrangida no conceito da UML Executável, será executada nesta seção. A abordagem em foco será aplicada com o auxílio de ferramentas, afim de transformar um modelo UML em código. Desta forma, pode-se ao final do trabalho efetuar uma discussão entre os métodos de produção de código abordados.

Para exemplificar na prática o funcionamento da UML Executável, foram utilizados dois *plugins* da ferramenta *Eclipse*, primeiramente o *Papyrus* e em seguida o *Acceleo*. É necessário a utilização de ambos os *plugins*, pois estes se completam de forma que o *Papyrus* é utilizado para fazer a modelagem do diagrama UML em questão, já que este componente do *Eclipse* dá suporte e oferece recursos suficientes para modelar qualquer diagrama UML, e *o Acceleo* importa o modelo criado no *Papyrus* e executa este diagrama com o intuito de produzir código na linguagem especificada. Sendo assim, de acordo com a Figura 25, pode-se notar que esse processo de transformação do diagrama UML em código, foi realizado por meio de passos descritos nas seções abaixo.

Figura 25 - Transformando diagrama UML em código

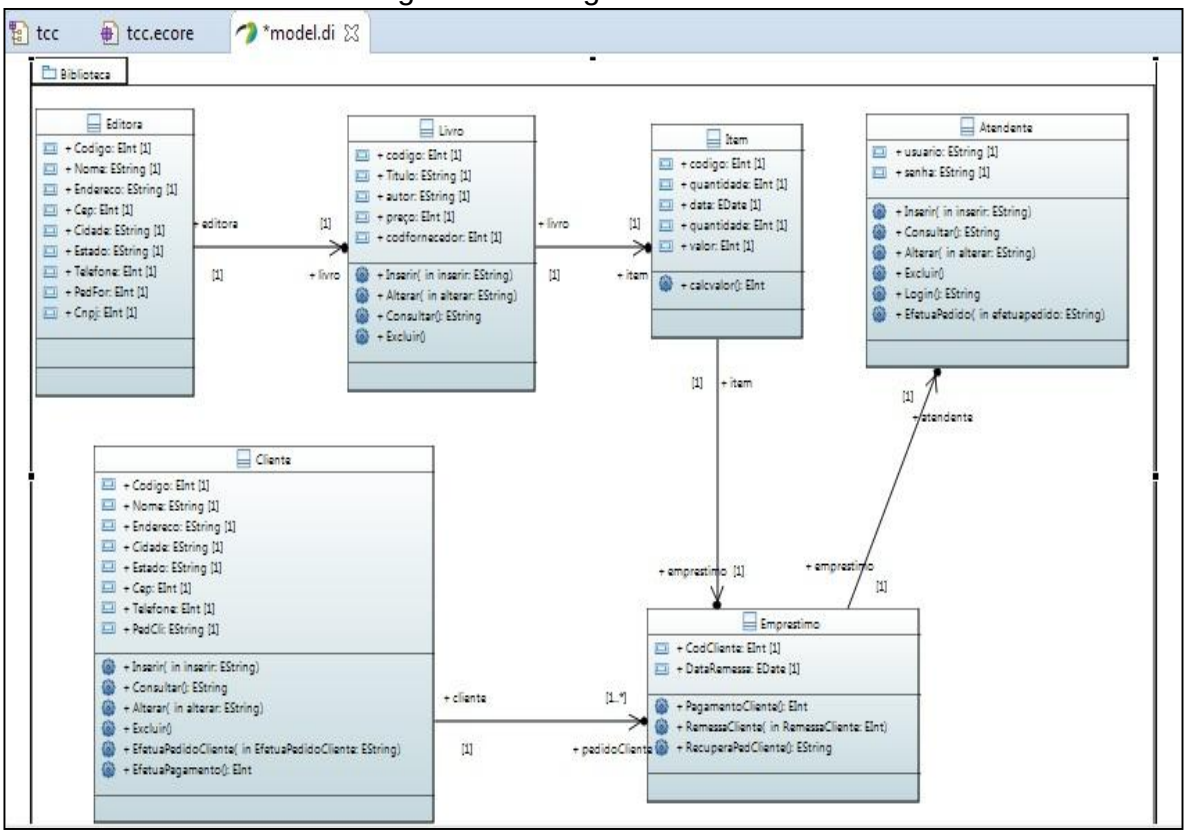


Fonte: Adaptado de ACCELEO (2014)

4.2.1 Elaborando um Diagrama de Classes UML

De acordo com o Passo 1 apresentado na Figura 25, foi realizado no *Papyrus* a modelagem de um diagrama de classes de uma biblioteca, que contém as classes Editora, Livro, Atendente, Item, Cliente, e Empréstimo. Obrigatoriamente sempre que for efetuar a geração de código a partir de um modelo UML, é necessário que todas as classes do modelo estejam dentro de um pacote, caso todos os objetos não estiverem inclusos em um pacote UML, a geração de código automática não será realizada com êxito, conforme é mostrado na Figura 26.

Figura 26 - Diagrama de Classes



Fonte: da pesquisa (2015)

4.2.2 Transformando Modelo UML em Código Java

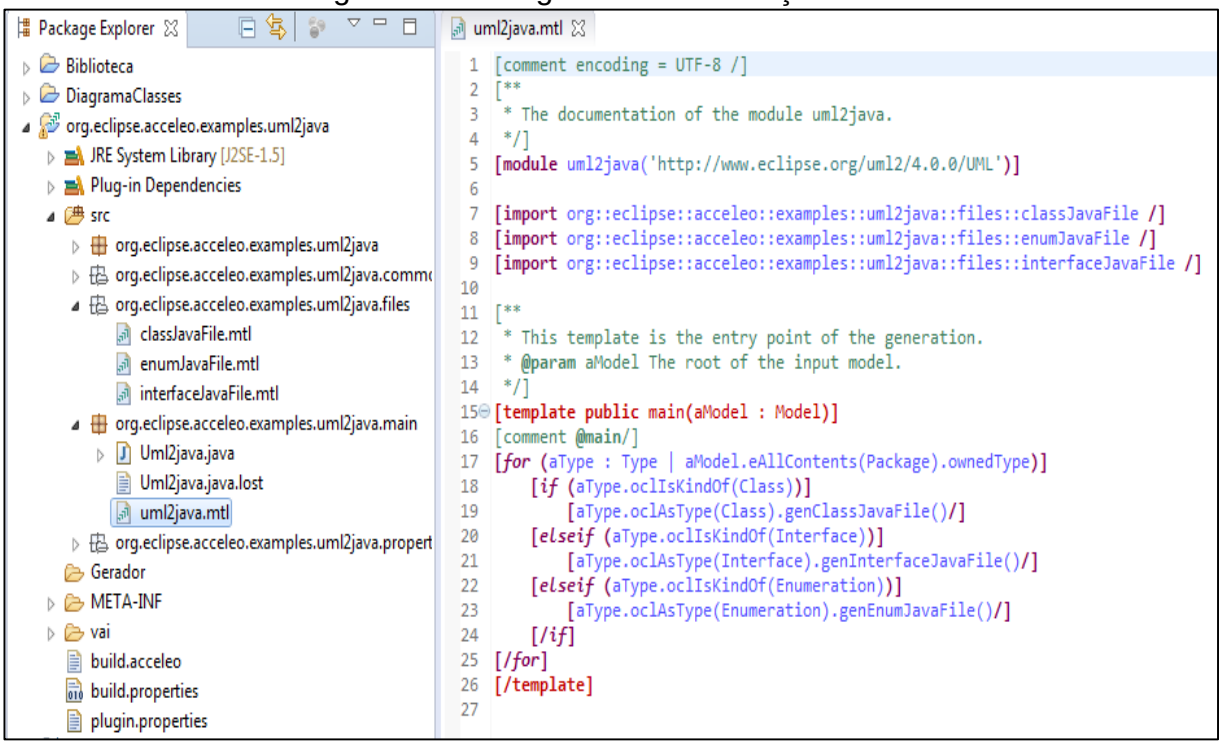
Com a modelagem do diagrama de classes criada no *Papyrus*, é possível dar início a etapa de transformação do modelo UML em código, seguindo o Passo 2 mostrado na Figura 25. Conforme foi descrito na sessão 4.1.3 Transformando o modelo em código, é possível gerar código automático por meio do modelo em diferentes linguagens, a escolhida nesse processo de transformação de modelo UML em código trata-se da linguagem *Java*.

Para iniciar a conversão do modelo visual em textual, é necessário criar um novo projeto no *Acceleo*, no qual ofereça a opção UML para *Java*, que diferente do processo realizado na sessão 4.1.3 Transformando o modelo em código, já contém o *script* pré formatado para executar este processo de transformação. A partir disso, será criado um arquivo gerador com o formato MTL, responsável pela realização de todo esse processo, este arquivo gerador nomeado *uml2java* comanda todos os outros que também estão no formato MTL, como o arquivo MTL transforma classe, *interface* e enumerações. O gerador ativa estes três arquivos conforme vão sendo

encontrados, portanto é necessário executar o gerador que contém todo o código de transformação.

A linguagem utilizada no arquivo MTL para realizar esse processo de transformação do diagrama UML em código é a *Model to Text Transformation,* a mesma utilizada para gerar código *Java* a partir do modelo *ecore*, possuindo as características e restrições equivalentes as que foram descritas na sessão 4.1.3 Transformando o modelo em código, como pode-se notar na Figura 27.

Figura 27 - Código de transformação UML



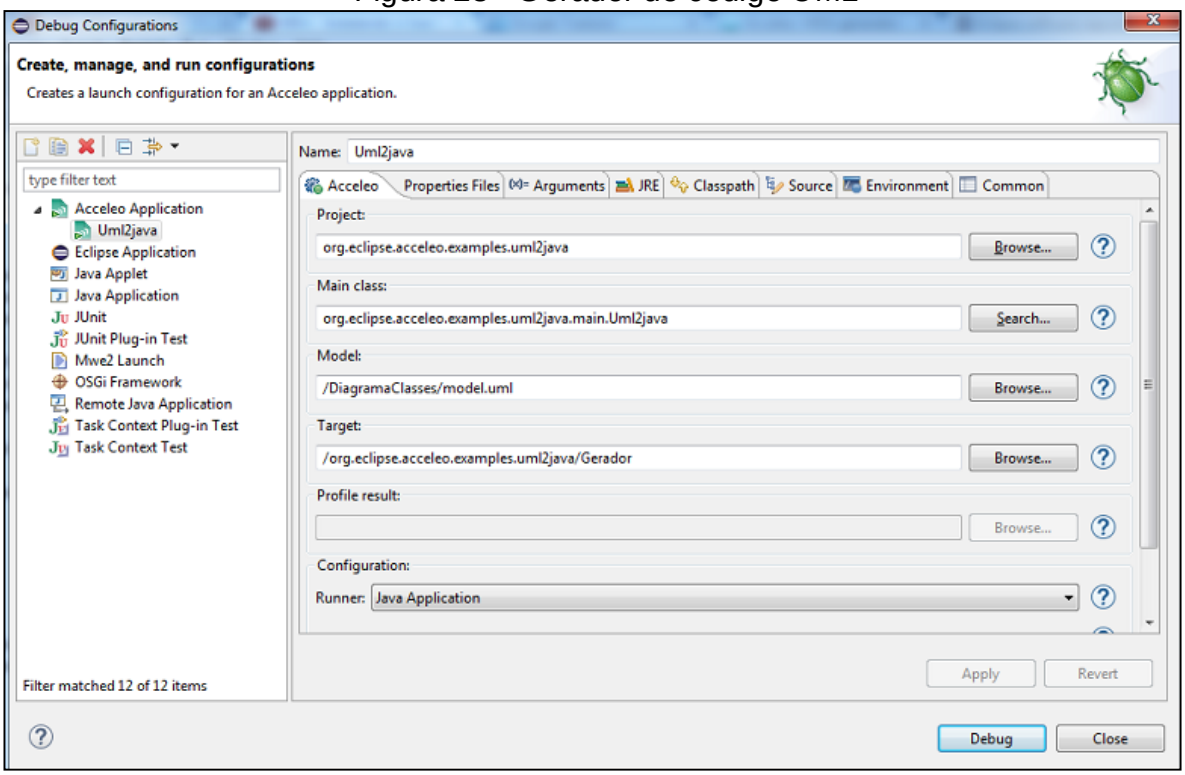
Fonte: da pesquisa (2015)

Contudo, apesar do *Acceleo* gerar código automaticamente por meio de um *script* elaborado pelo desenvolvedor, é viável que se tenha conhecimento da linguagem *Model to text transformation*, com o intuito de efetuar algumas mudanças ou incluir especificações para melhorar o código e adicionar novas funcionalidades, além da necessidade de dominar a linguagem que será gerada, neste caso em *Java*, que também compõe o *script*. Esta linguagem *Model to text transformation* possui compatibilidade com a semântica OCL, sendo assim, poderá ser definido diretamente no modelo algumas restrições com a intenção de melhorar essa geração automática, detectando todos os detalhes semânticos da UML que acaba se perdendo no exato momento da execução dos diagramas (OMG, 2008).

4.2.3 Configurando o Gerador de Código

Para finalizar o processo de transformação do diagrama UML em código, é preciso realizar o Passo 3 da Figura 25. No qual se trata da configuração do gerador de código por meio do *plugin Acceleo*. A partir da execução do arquivo *uml2java*, uma tela para configuração de saída do código será exibida de acordo com a Figura 28. Nesta tela é necessário configurar o caminho do projeto, o local que se encontra o arquivo principal *uml2java* com o formato MTL, selecionar o modelo UML criado anteriormente no *Papyrus*, e por fim definir a pasta de saída para o código gerado na linguagem *Java* a partir do modelo especificado.

Figura 28 - Gerador de código UML

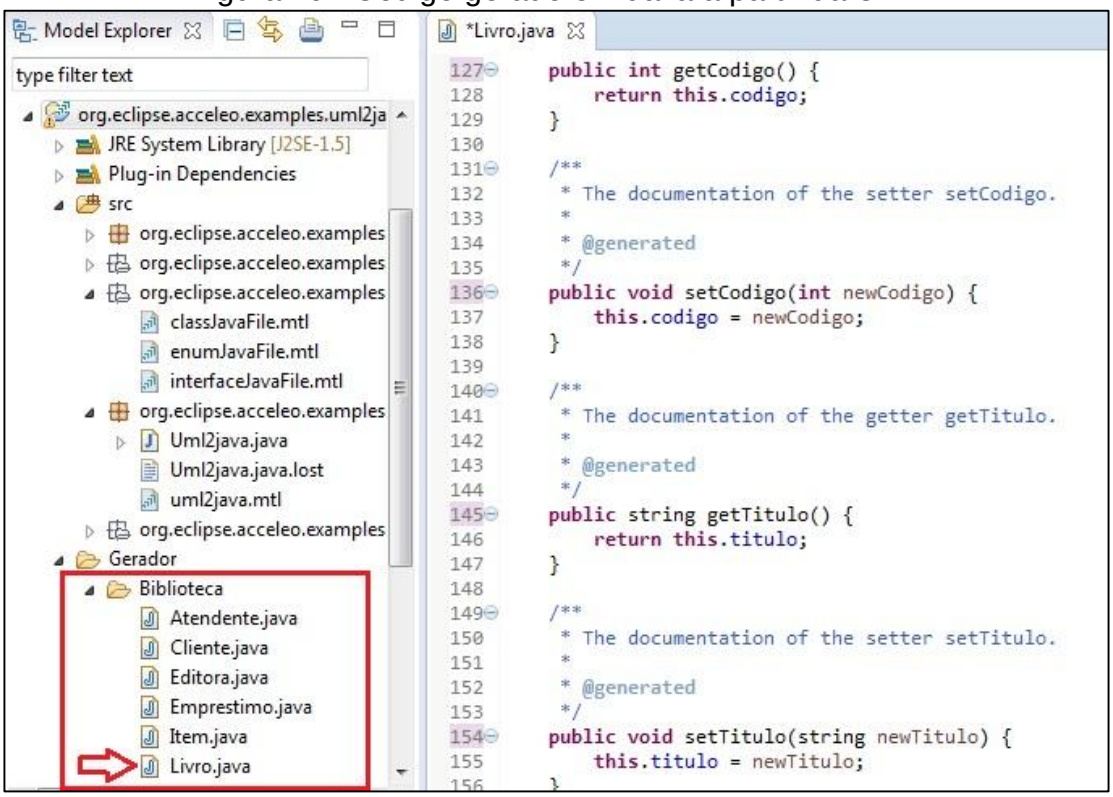


Fonte: da pesquisa (2015)

A partir do momento em que o comando para execução do processo de transformação for realizado, é possível verificar conforme mostrado na Figura 29, o que foi produzido na linguagem de programação *Java*, por meio do modelo de diagrama de classes. Como pode-se notar, foram geradas todas as classes do diagrama, e dentro dessas respectivas classes o código *Java*, nota-se que tudo que foi especificado no modelo UML foi gerado nas classes, como o nome da classe, suas propriedades e operações, sendo elas pegando e inserindo valores, além de definir

se a classe é pública ou privada, pode-se determinar qual o seu tipo, *string*, *int*, entre outros.

Figura 29 - Código gerado em Java a partir da UML



Fonte: da pesquisa (2015)

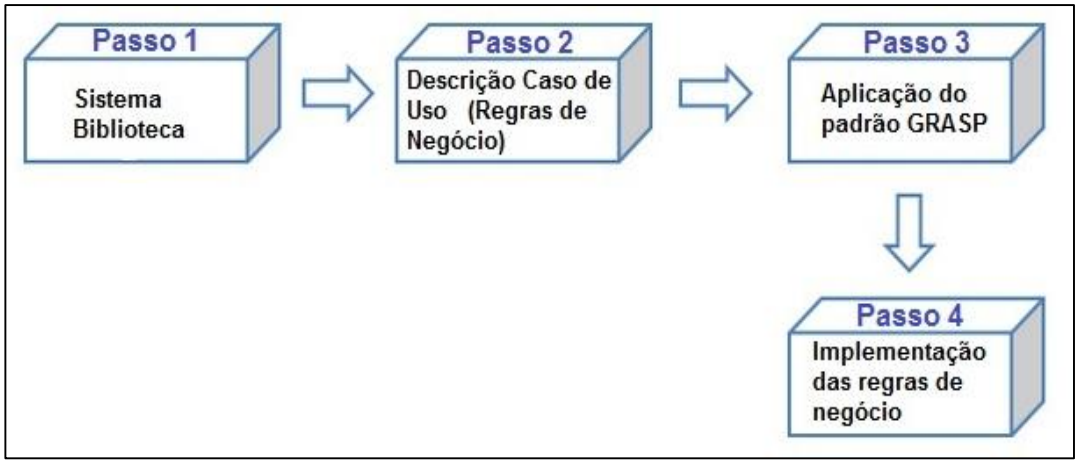
Caso sejam necessárias realizar novas alterações no modelo de diagrama de classes UML, não há problema algum em efetuar uma nova geração de código, basta realizar as alterações desejadas diretamente no modelo UML utilizando o *Papyrus*, e apenas executar o arquivo *uml2java* aplicando no mesmo exemplo, já que as configurações estarão pré-definidas.

4.3 Aplicando Regras de Negócio na Geração de Código

Nas seções anteriores foi possível demonstrar na prática exemplos de desenvolvimento dirigido a modelos de acordo com a proposta deste estudo. Na seção 4.1, foi desenvolvido um metamodelo *ecore,* que automaticamente foi transformado em um modelo *ecore*, e posteriormente também de forma automática, código foi gerado por meio deste modelo. Em seguida na seção 4.2 mais uma abordagem do desenvolvimento dirigido a modelos foi colocada em prática, de modo que código foi gerado automaticamente por meio do diagrama de classes UML, executando o padrão da UML Executável.

Uma nova abordagem estudada será aplicada na seção atual, na qual seguindo o desenvolvimento dirigido a modelos, regras de negócio serão utilizadas na geração automática de código por meio de modelos. Com o intuito de colocar em prática esta abordagem, dois *plugins* do *Eclipse* foram utilizados, além da criação do *Empty EMF Project* para a transformação do modelo UML em modelo *ecore*. O *Papyrus* foi o *plugin* escolhido para realizar a modelagem de todos os diagramas UML presentes nesta seção, inclusive o diagrama de classes utilizado para efetuar a geração automática de código. Já o *Acceleo* é o *plugin* responsável por executar o modelo *ecore* gerado pelo modelo UML que foi desenvolvido no *Papyrus,* transformando este modelo visual em código. Desse modo, como mostrado na Figura 30, pode-se notar quais foram os passos seguidos para realização desta etapa, ambos os passos foram descritos nas seções a seguir.

Figura 30 - Etapas do desenvolvimento da seção 4.3

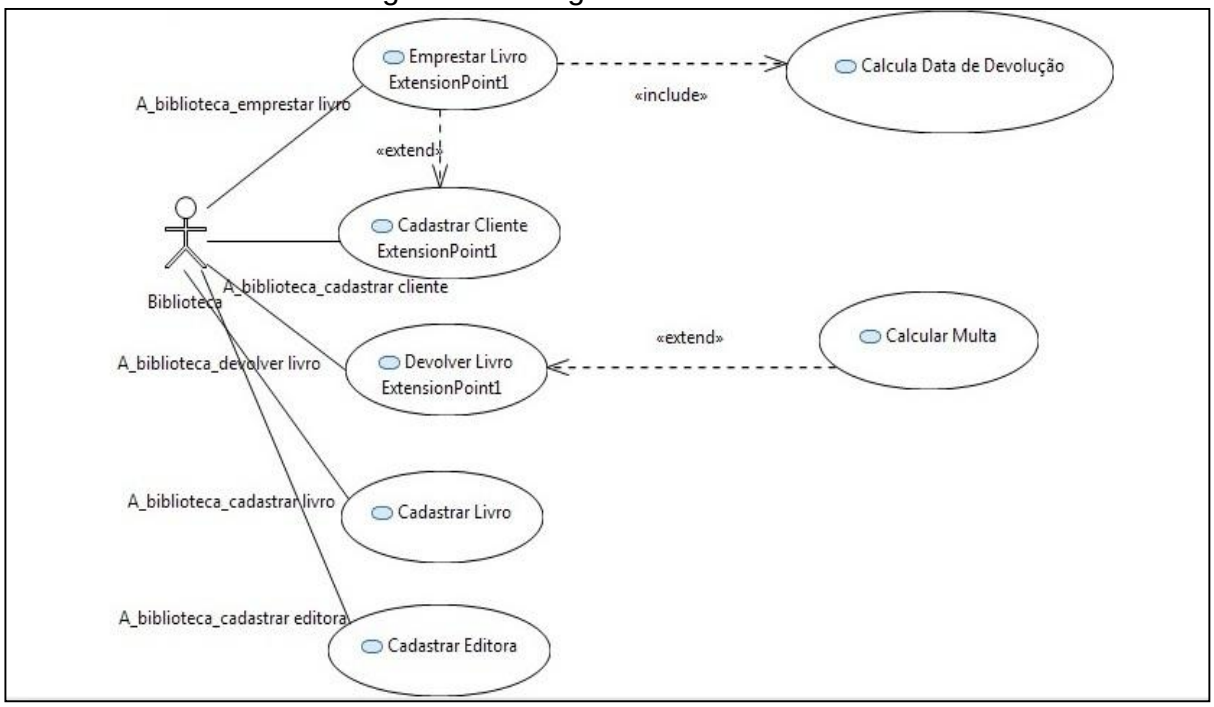


Fonte: da pesquisa (2015)

4.3.1 Descrição do Sistema de Biblioteca

Com o propósito de aplicar regras de negócio na geração automática de código, foi definido um cenário de locação que vem sendo trabalhado desde o início do estudo, neste caso um sistema de biblioteca foi modelado a fim de ser utilizado para exemplificação durante toda a seção de acordo com o Passo 1 da Figura 30. Desse modo, considera-se o sistema de biblioteca mostrado na Figura 31, no qual por meio de um diagrama de casos de uso foi possível fazer a representação do mesmo.

Figura 31 - Diagrama casos de uso



Fonte: da pesquisa (2015)

Na Figura 31, pode-se observar mediante aos casos de uso como será o funcionamento desse sistema de biblioteca e sua finalidade. Basicamente o sistema efetua o cadastro de editora, livro e cliente. Ao efetuar o empréstimo de um livro é necessário que o cálculo de devolução do mesmo seja realizado, além disso caso a devolução do livro seja realizada após o prazo de devolução, a multa do mesmo deverá ser calculada.

4.3.2 Descrição do Caso de Uso

Como se trata apenas de um exemplo de um sistema de biblioteca, será exposto a implementação do caso de uso "Calcula Data de Devolução", mostrado na Figura 31. A fim de compreender melhor as funcionalidades desse caso de uso e as regras de negócio que foram aplicadas dentro do "Calcula Data de Devolução", conforme o Passo 2 da Figura 30, a seguir pode-se observar a descrição desse caso de uso apresentado no Quadro 1.

Quadro 1 - Descrição do caso de uso

Calcula Data de Devolução	
- Fluxo Principal	- Fluxo Alternativo
1. Pega o número de livros do empréstimo.	2.a Caso o cliente empreste 3 ou mais livros.
1.1 Pega o prazo de devolução de cada livro.	2.a.1 - Adiciona mais 2 dias para cada livro. após o 2º livro emprestado.
1.2 Calcula a data de devolução do livro.	2.a.2 - Calcula a nova data.
2. Selecione o maior prazo dentre todos os livros.	2.a.3 - Retorna ao passo 3.
3. Retorna a data de devolução dos livros.	

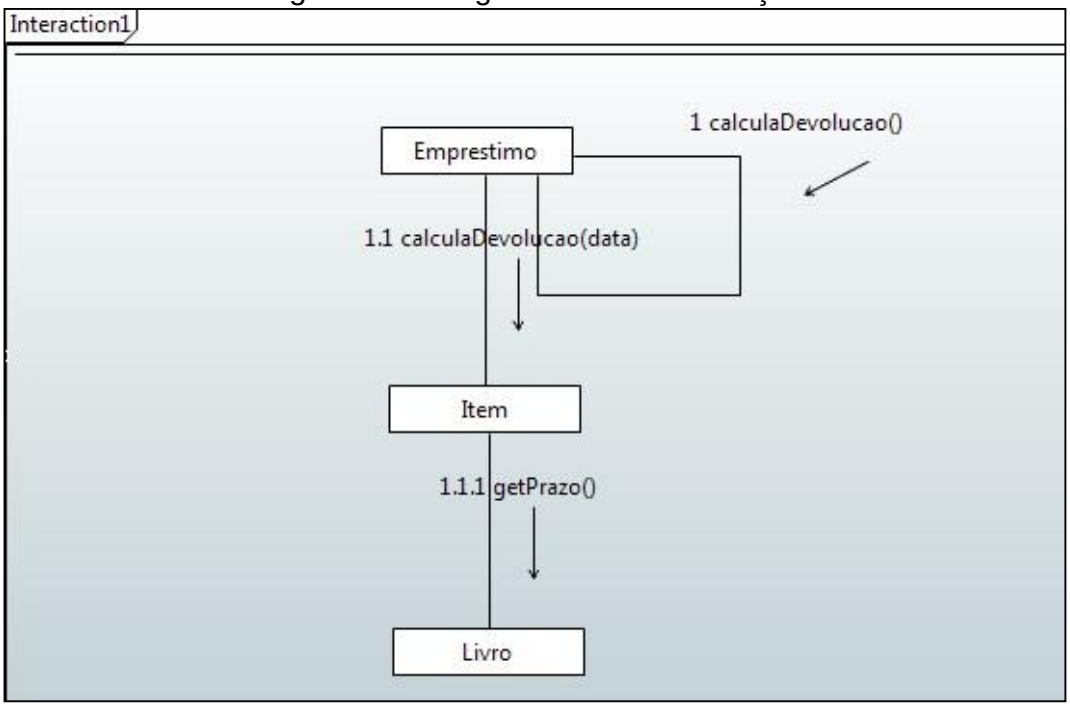
Fonte: da pesquisa (2015)

4.3.3 Aplicação do Padrão GRASP

De acordo com o Passo 3 da Figura 30, o padrão *General Responsibility Assignment Software Patterns* (GRASP) *Expert* será aplicado nos modelos UML subsequentes do sistema de biblioteca escolhido para exemplificação desse estudo. Segundo Larman (2005), o padrão GRASP *Expert* consiste na atribuição de responsabilidades de classes em objetos, de forma que essas responsabilidades delegadas incluem os métodos da classe, determinando as informações necessárias para cumpri-las e armazená-las, essas informações atribuirá as responsabilidades sobre aquela determinada operação a ser executada. No exemplo da biblioteca, aplicando o padrão GRASP Expert, pode-se atribuir a responsabilidade de adicionar todos os livros da biblioteca em sua respectiva classe, e responsabilizar um método que verifique todos os livros que foram emprestados e retorne o seu prazo de devolução.

As classes utilizadas para relatar o caso de uso "Calcula Data de Devolução", descrito no Quadro 1, pode ser representado por meio do diagrama de comunicação mostrado na Figura 32.

Figura 32 - Diagrama de comunicação

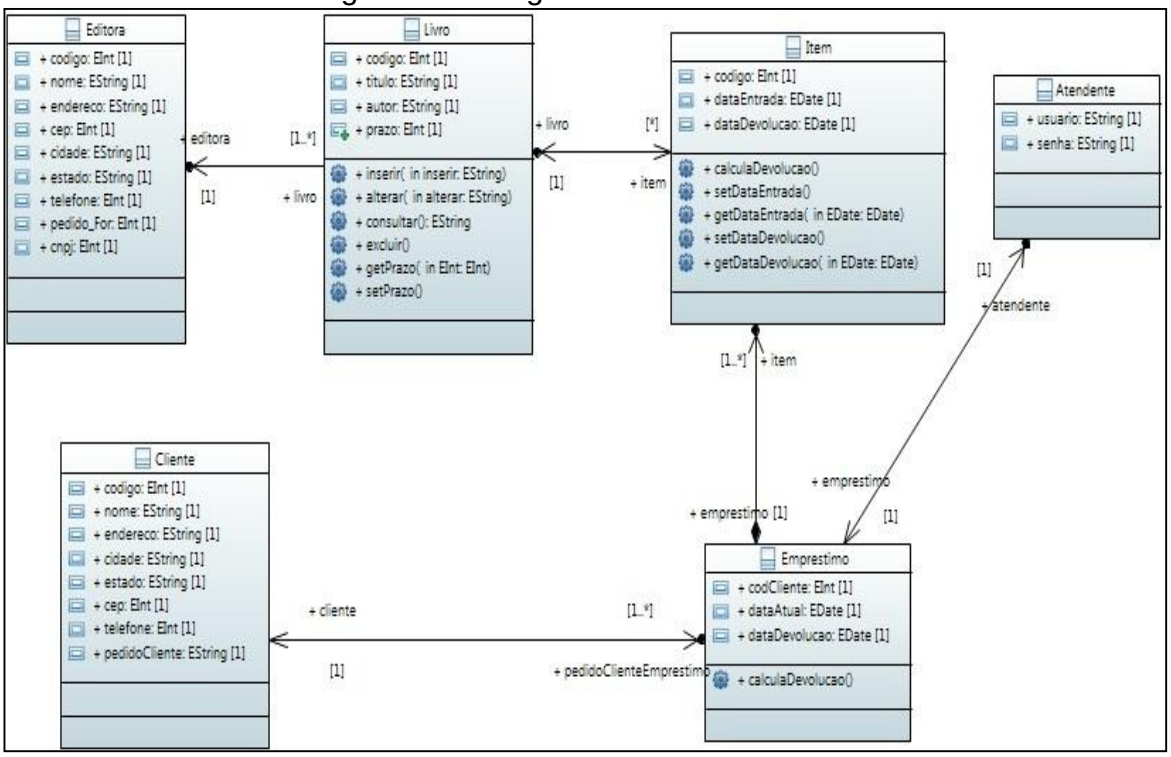


Fonte: da pesquisa (2015)

As três classes utilizadas para calcular a devolução do livro no sistema de biblioteca evidenciado na seção 4.3.1, foram as classes, livro responsável por todas as informações de cada livro inserido no sistema, a classe item responsável de pegar o prazo de cada livro e adicionar esse prazo em cima da data de saída do livro, efetuar o cálculo de devolução e retornar a data de devolução de cada livro, e por fim a classe empréstimo é responsável por atribuir a regra de negócio 2.a retratada no Quadro 1, que é selecionar o maior prazo dentre todos os livros e retornar a data de devolução desses livros.

O diagrama de casos de uso da Figura 31, foi implementado pelo modelo do diagrama de classes da Figura 26, para implantar as funcionalidades retratadas no caso de uso “Calcula Data de Devolução", descrito no Quadro 1, foi efetuada a aplicação do padrão GRASP em cima deste modelo de diagrama de classes, o padrão GRASP atribuirá as responsabilidades as suas respectivas classes, ou seja, definirá os métodos das classes. Com o diagrama de classes atualizado, pode-se observar o resultado deste modelo após a aplicação do padrão GRASP por meio da Figura 33.

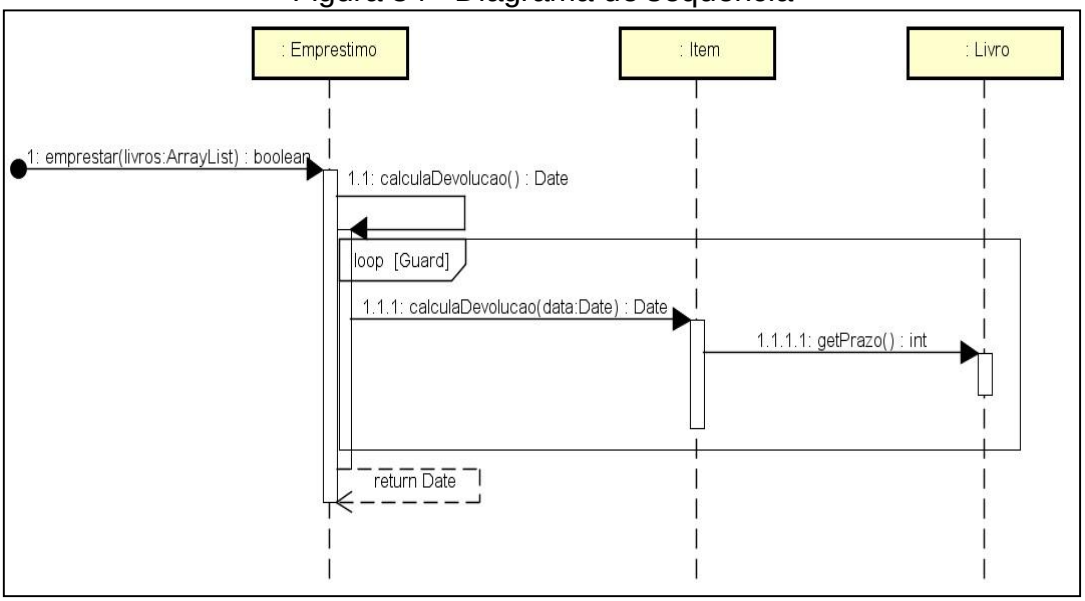
Figura 33 - Diagrama de classes GRASP



Fonte: da pesquisa (2015)

A implementação do sistema de biblioteca será baseada no caso de uso "Calcula Data de Devolução", descrito no Quadro 1, que contém as regras de negócio criadas para essa exemplificação, além do diagrama de classes da Figura 33, já com o padrão GRASP aplicado, e por fim embasado no diagrama de sequência mostrado na Figura 34, é possível notar a interação das classes para a implementação do caso de uso retratado.

Figura 34 - Diagrama de sequência



Fonte: da pesquisa (2015)

4.3.4 Implementação das Regras de Negócio

Durante todo o estudo, foi possível apresentar diferentes formas de geração de código automática por meio do desenvolvimento dirigido a modelos. Desde a transformação utilizando o padrão EMF, até mesmo a execução na prática da UML Executável gerando código automaticamente por meio de modelo. De acordo com o Passo 4 da Figura 30, nesta seção ocorrerá a implementação das regras de negócio definidas no Calcula Data de Devolução que foram apresentadas no Quadro 1 da seção 4.3.2. Logo em seguida, essas regras de negócio serão aplicadas diretamente no gerador de código do *plugin Acceleo*.

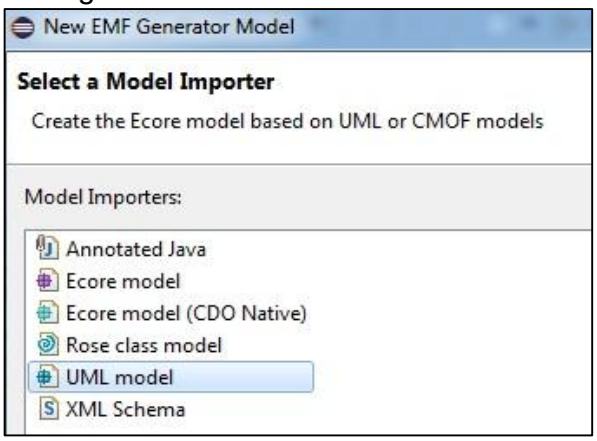
Sendo assim, o código será gerado automaticamente por meio de um modelo *ecore* que foi originado mediante a um modelo UML, mais especificamente o diagrama de classes do sistema de biblioteca já atualizado com o padrão GRASP *Expert* mostrado na Figura 33. As classes em foco responsáveis pelas principais funcionalidades desse sistema de biblioteca foram definidas, sendo estas as classes Livro, Item e Empréstimo, que possuem os principais atributos e métodos para a execução do caso de uso Calcula Data de Devolução apresentado na Figura 31, acompanhado das regras de negócio expostas no Quadro 1 da seção 4.3.2.

Colocando em prática a exemplificação proposta na seção atual, antes mesmo de iniciar a geração de código automaticamente por meio do modelo UML utilizando o *plugin Acceleo*, é necessário que seja feita a transformação desse modelo UML mostrado na Figura 33 em um modelo *ecore*, afim de tornar esse modelo visual em forma estrutural e com a padronização EMF, visto que esse processo é necessário para importar e coletar todas as informações do modelo no gerador, podendo assim inserir novas regras e especificações. O modelo deve ser semanticamente padronizado, ao contrário de quando é apenas executado e transformado de forma automática sem a inserção de regras de negócio, conforme apresentado na seção 4.2, no qual o script é genérico ao ponto de ser aplicado em qualquer modelo UML.

Dessa forma, para realizar a transformação do modelo UML em *ecore*, com o *Eclipse* aberto, é necessário efetuar a criação de um *Empty EMF Project,* e dentro do mesmo uma pasta nomeada como *model*, no qual o modelo UML será inserido. Neste projeto EMF que acabou de ser originado, é preciso criar o EMF *Generator Model*, tal como o gerador de modelo, conforme mostrado na Figura 35, em que o modelo UML selecionado é transformado em um modelo *ecore*. Esse processo de transformação

de um modelo em outro, é similar ao procedimento empregado na seção 4.1.2, "Transformando o metamodelo em modelo".

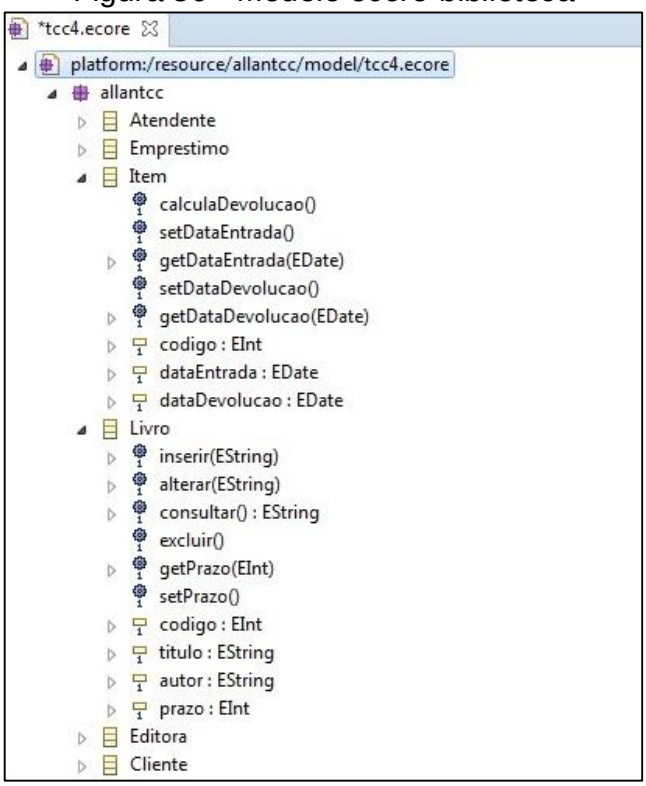
Figura 35 - Gerando modelo *ecore*



Fonte: da pesquisa (2015)

A partir do momento em que o modelo *ecore* foi gerado, é possível iniciar a próxima etapa do processo de transformação, no qual o *Empty EMF Project* deve ser executado e com isso o *Eclipse Application* será inicializado. No *Eclipse Application,* um projeto *Java* deve ser criado e no mesmo o modelo *ecore* importado, com o propósito do gerador visualizar o modelo conforme mostrado na Figura 36 e promover a geração automática de código por meio desse modelo *ecore*.

Figura 36 - Modelo *ecore* biblioteca



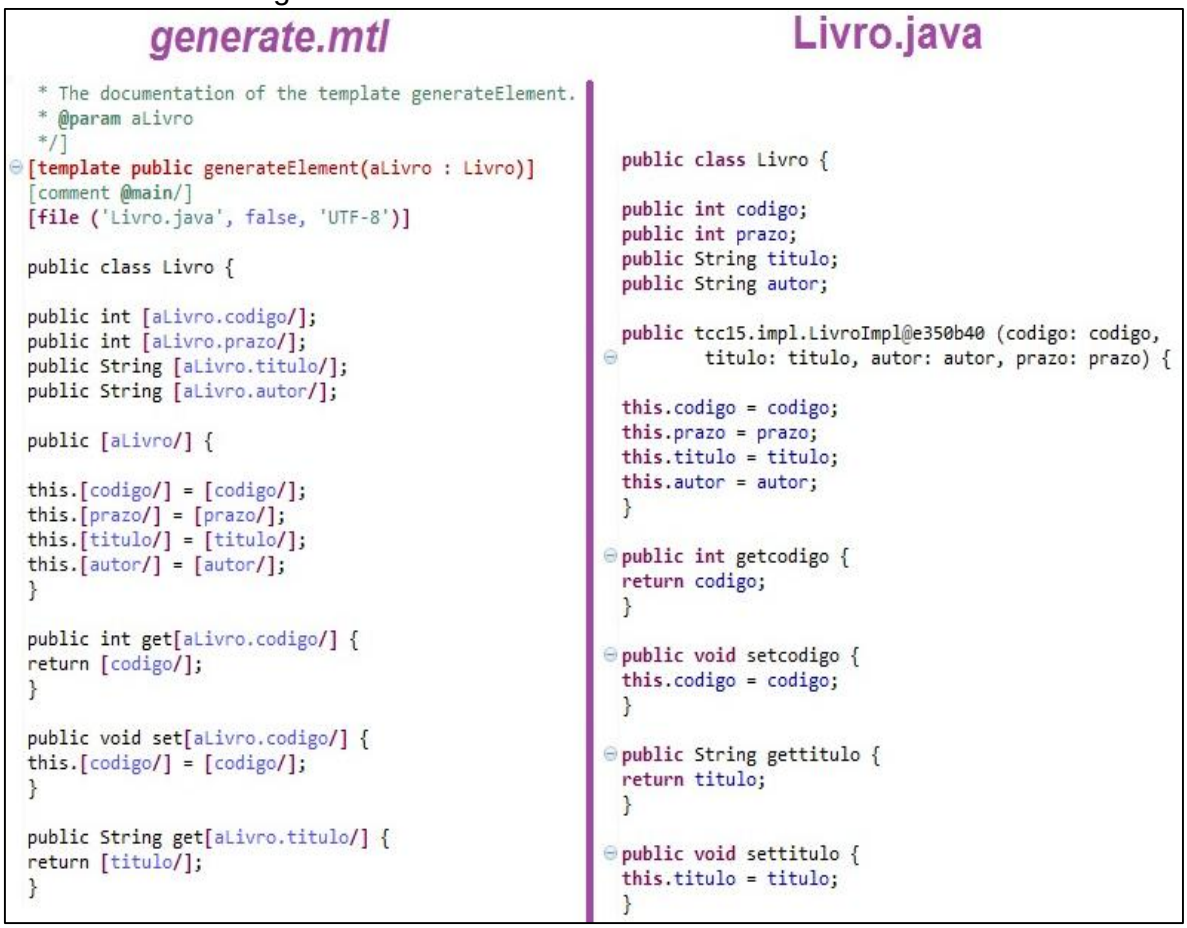
Fonte: da pesquisa (2015)

Com o modelo importado, pode-se criar o *Acceleo Project* a fim de gerar o arquivo *generate.mtl*, no qual todas as especificações, restrições e regras de negócio são inseridas, sendo este o mediador entre o modelo e o código a ser produzido. Os passos para criação do *Acceleo Project* e informações sobre seu funcionamento são os mesmos relatados na seção 4.1.3 "Transformando o modelo em código".

Por meio desse arquivo *generate.mtl* do *Acceleo Project,* todas as classes do sistema de biblioteca apresentado no diagrama de classes da Figura 33 foram geradas, e dentro dessas respectivas classes está presente o código *Java*. O código é gerado apenas após o arquivo *generate.mtl* estar com o *script* pronto, escrito na linguagem M2T e na linguagem escolhida para geração. Para executar este *script* é preciso configurar o gerador do *Acceleo*, definindo o respectivo projeto, arquivo *generate.mtl*, modelo e a pasta de saída do código, os passos para configuração desse gerador foram descritos na seção 4.1.4.

O código *Java* foi gerado embasado na descrição do caso de uso "Calcula Data de Devolução" relatado no Quadro 1, no qual foi realizado o cálculo de devolução de um ou mais livros no sistema de biblioteca mostrado no caso de uso da Figura 31. Para implementação do caso de uso "Calcula Data de Devolução" foram necessárias a utilização das classes Livro, Item e Emprestimo, representadas pelo diagrama de classes mostrado na Figura 33. Na Figura 37 pode-se notar a inserção dessas regras de negócio diretamente no arquivo *generate.mtl,* responsável por gerar o código *Java* e ao seu lado o resultado desse código gerado no momento em que foi produzido nessa abordagem do desenvolvimento dirigido a modelos, conforme também é mostrado nas Figuras 38 e 39.

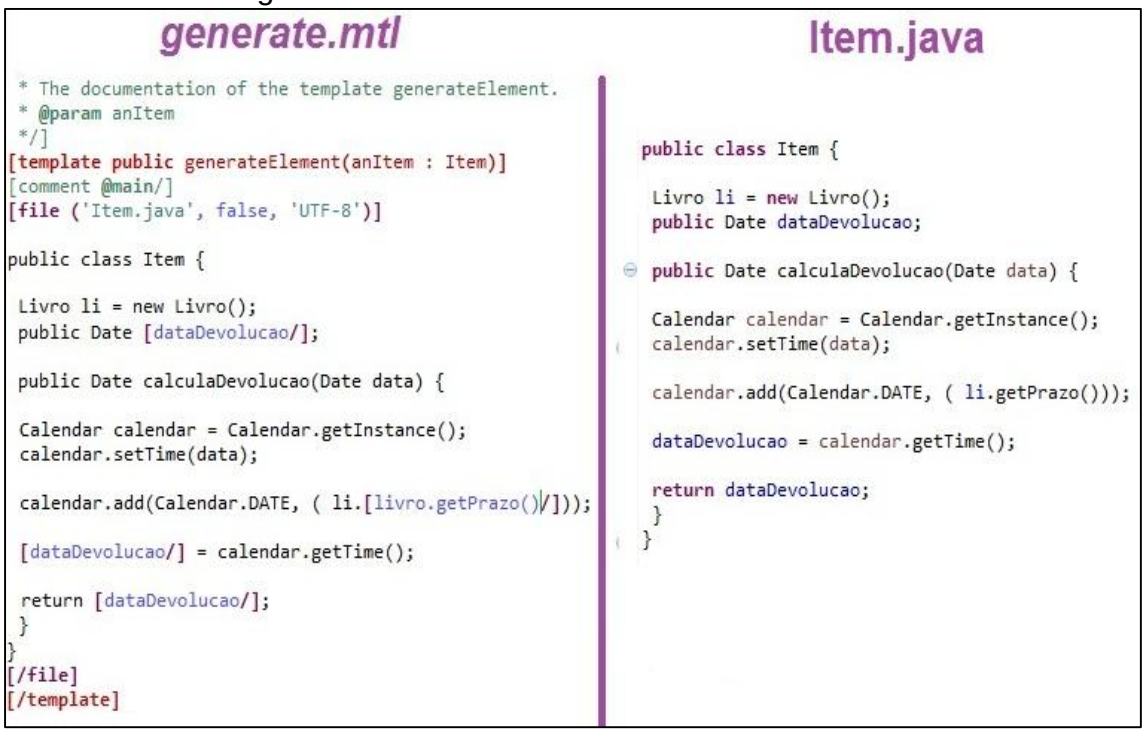
Figura 37 - Classe Livro do sistema de biblioteca



Fonte: da pesquisa (2015)

O script do gerador e o código da classe Livro são apresentados na Figura 37. É possível notar primeiramente o arquivo *generate.mtl* responsável por todas as informações de cada livro, no qual foi produzido um *script* para geração de código *Java*. Deste modo, pode-se constatar que a linguagem M2T foi utilizada perante boa parte do *script,* mesmo com a inserção da linguagem *Java* que foi necessariamente empregada em algumas partes do código, principalmente na tipagem. Ao lado do *generate.mtl*, nota-se o resultado da geração automática desse código gerado na classe Livro.*java*, basicamente com as funcionalidades de inserir e apanhar informações de cada livro no sistema de biblioteca.

Figura 38 - Classe Item do sistema de biblioteca



Fonte: da pesquisa (2015)

Da mesma maneira, na Figura 38 é mostrado o arquivo *generate.mtl* e a classe *Java* com o resultado do código gerado de forma automática. Apesar da linguagem M2T ser utilizada nesse *script* com a intenção de apanhar as informações do modelo, percebe-se uma frequência maior da linguagem *Java* ao decorrer da codificação de acordo com o aumento da utilização das regras de negócio. Na classe Item.*java*, pode-se observar o código gerado efetuando o cálculo de devolução do livro, no qual a data atual é somada com o prazo do livro, e assim é possível retornar a data de devolução exata.

Figura 39 - Classe Empréstimo do sistema de biblioteca

generate.mtl

```
[template public generateElement(anEmprestimo : Emprestimo)]
[comment @main/]
[file ('Emprestimo.java', false, 'UTF-8')]
public class Emprestimo {

Scanner entrada = new Scanner(System.in);
ArrayList<Item> items = new ArrayList();

public Date [calculaDevolucao()/] {

Date data = new Date();
Calendar cl = Calendar.getInstance();
Date aux;

if (items.size() > 2) {
for (int i = 0; i < items.size(); i++) {
if (i > 0) {
cl.setTime(data);
cl.add(Calendar.DAY_OF_MONTH, 2);
aux = items.get(i).[calculaDevolucao(cl.getTime())/];
} else {
aux = items.get(i).[calculaDevolucao(data)/];
       }
System.out.println(aux);
}
} else {
for (int i = 0; i < items.size(); i++) {
aux = items.get(i).[calculaDevolucao(data)/];
System.out.println(aux);
}
}
Date dataMaior = null;
dataMaior = items.get(0).[dataDevolucao/];
for (int j = 0; j < items.size(); j++) {
if (dataMaior.before(items.get(j).[dataDevolucao/])) {
dataMaior = items.get(j).[dataDevolucao/];
}
}
System.out.println("A data atual é:");
System.out.println(data);

System.out.println("A maior data é: ");
return dataMaior;
}
}
[/file]
[/template]
```

Emprestimo.java

```
public class Emprestimo {

Scanner entrada = new Scanner(System.in);
ArrayList<Item> items = new ArrayList();

public Date calculaDevolucao() {

Date data = new Date();
Calendar cl = Calendar.getInstance();
Date aux;

if (items.size() > 2) {
for (int i = 0; i < items.size(); i++) {
if (i > 0) {
cl.setTime(data);
cl.add(Calendar.DAY_OF_MONTH, 2);
aux = items.get(i).calculaDevolucao(cl.getTime());
} else {
aux = items.get(i).calculaDevolucao(data);
       }
System.out.println(aux);
}
} else {
for (int i = 0; i < items.size(); i++) {
aux = items.get(i).calculaDevolucao(data);
System.out.println(aux);
}
}
Date dataMaior = null;
dataMaior = items.get(0).dataDevolucao;

for (int j = 0; j < items.size(); j++) {
if (dataMaior.before(items.get(j).dataDevolucao)) {
dataMaior = items.get(j).dataDevolucao;
}
}
System.out.println("A data atual é:");
System.out.println(data);

System.out.println("A maior data é: ");
return dataMaior;
}
}
```

Fonte: da pesquisa (2015)

Com um nível de utilização de regras de negócio bem mais alto, conforme apresentado no arquivo *generate.mtl* e na classe Emprestimo.*java* mostrados na Figura 39, entende-se que a utilização da linguagem a ser gerada aumenta demasiadamente no *generate.mtl*, mesmo com a utilização da linguagem M2T para conceber informações do modelo e efetuar essa transformação visual para textual, a linguagem de saída está presente em praticamente todo *script*, isso ocorre principalmente por conta das regras de negócio serem especificas semanticamente para cada tipo de linguagem, levando em consideração que o *script* deve ser o mais genérico possível para poder padronizar essa codificação e não especificar tanto a semântica da linguagem a ser produzida, mas em casos que as regras de negócios são aplicadas, a utilização da linguagem de saída se torna imprescindível. Contudo, a classe Emprestimo.*java*, apresenta a implementação das regras de negócio definidas pelo caso de uso "Calcula Data de Devolução", descritas no Quadro 1.

5 DISCUSSÃO

Nesse capítulo, são discutidas as abordagens desenvolvidas no Capítulo 4, a fim de concluir a pesquisa. De acordo com as abordagens MDD utilizadas no desenvolvimento deste estudo, pode-se levantar algumas considerações referentes a sua viabilidade de aplicação e em quais situações cada uma das abordagens tratadas é mais adequada. Ponderando todas as etapas realizadas no desenvolvimento, qualquer abordagem possui suas respectivas vantagens e desvantagens, no qual cada uma delas será discutida a seguir.

Na seção 4.1 foi realizado o desenvolvimento a partir de um metamodelo, que foi utilizado para representar um modelo, de modo que regras e especificações foram aplicadas diretamente nesse metamodelo, que gerou um modelo automaticamente com essas devidas restrições inicialmente implantadas no metamodelo. Dessa maneira, não é necessário que se desenvolva modelos específicos, e sim metamodelos genéricos que possam posteriormente instanciar modelos, mas isto só é vantajoso quando se tem um domínio genérico, que é possível gerar um metamodelo que representa a regra geral dos modelos deste domínio, como o exemplo de locação, utilizado nesse estudo para gerar modelos que envolvam diferentes categorias de empréstimos.

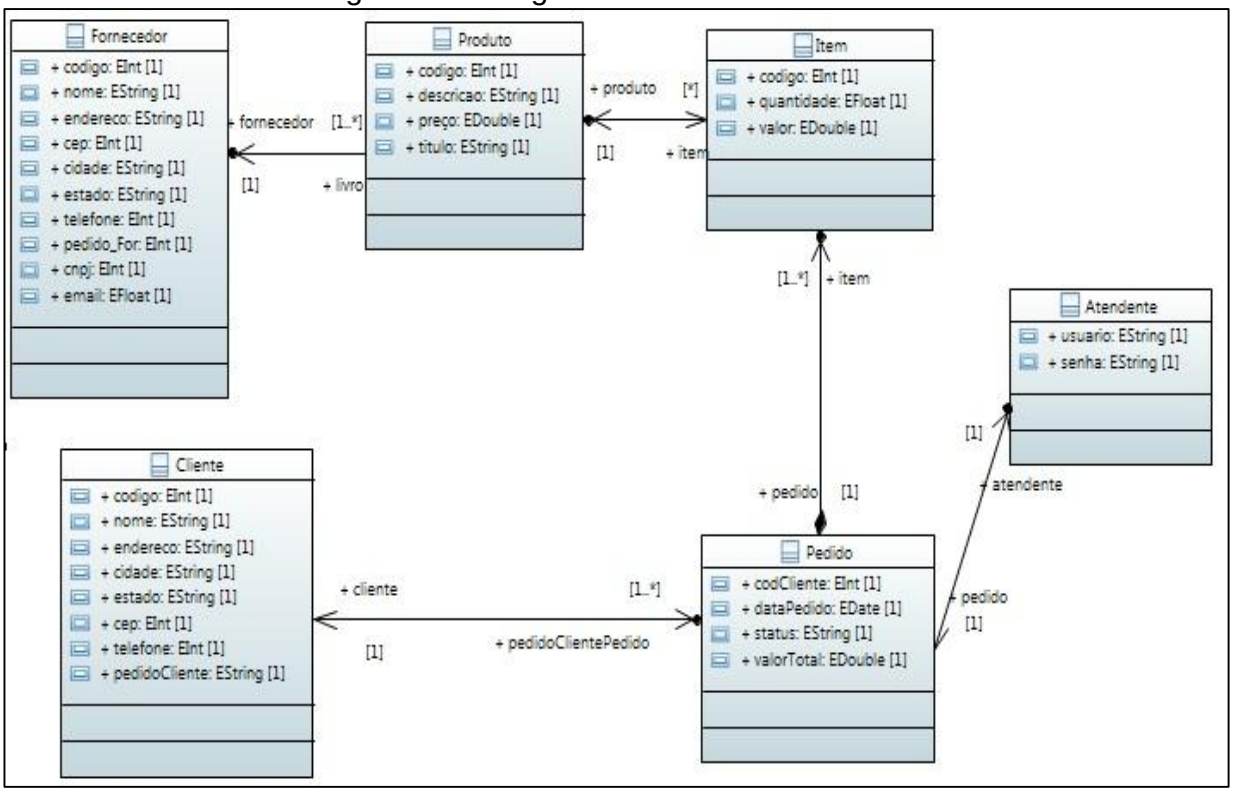
Portanto, essa abordagem pode ser considerada eficaz quando se trata do desenvolvimento dirigido a modelos, pelo fato do metamodelo poder gerar modelos e posteriormente código de forma automática.

A linguagem do metamodelo e modelo gerado é a *ecore,* dialeto oficial do EMF, ou seja, sua semântica é voltada para a transformação de modelos, diferente da UML que oferece suporte completo para modelagem de um sistema, sendo assim mais detalhada e complexa que o modelo *ecore*. Em vista disso, pode-se dizer que a linguagem do metamodelo e modelo dessa abordagem é padronizada.

Com o propósito de destacar a viabilidade da abordagem do desenvolvimento a partir de um metamodelo, sendo este apto a instanciar modelos de forma semi-automática, nota-se que o metamodelo pode gerar diferentes modelos capazes de utilizar todas ou apenas algumas classes do metamodelo, assim como adicionar novos atributos nas classes. A fim de exemplificar na prática, pode-se verificar que a partir do metamodelo da Figura 19, que descreve as classes e relacionamentos

necessários para uma locação, foi possível instanciar o modelo de empréstimo mostrado na Figura 33 e o modelo de vendas mostrado na Figura 40.

Figura 40 - Diagrama de classes vendas



Fonte: da pesquisa (2015)

Na abordagem do desenvolvimento a partir de um modelo UML, desenvolvida na seção 4.2, vale destacar sua agilidade para geração de código por meio de um diagrama UML, no qual não existe a necessidade de transformar o modelo UML em *ecore*, colocando em prática o padrão da UML Executável, de forma que o código é gerado automaticamente por meio do diagrama UML. Mas apesar desse processo ser funcional, apresenta suas limitações. Pode-se destacar como principal limitação, o fato do modelo UML não ser transformado em *ecore*, o que implica que código gerado não recebe regras de negócio e nem inserções de novas especificações, pelo fato de tudo ser gerado automaticamente por meio do arquivo *uml2java* que possui seu *script* já definido exclusivamente para essa geração. Assim, o código produzido é exatamente conforme o conceito apresentado no modelo.

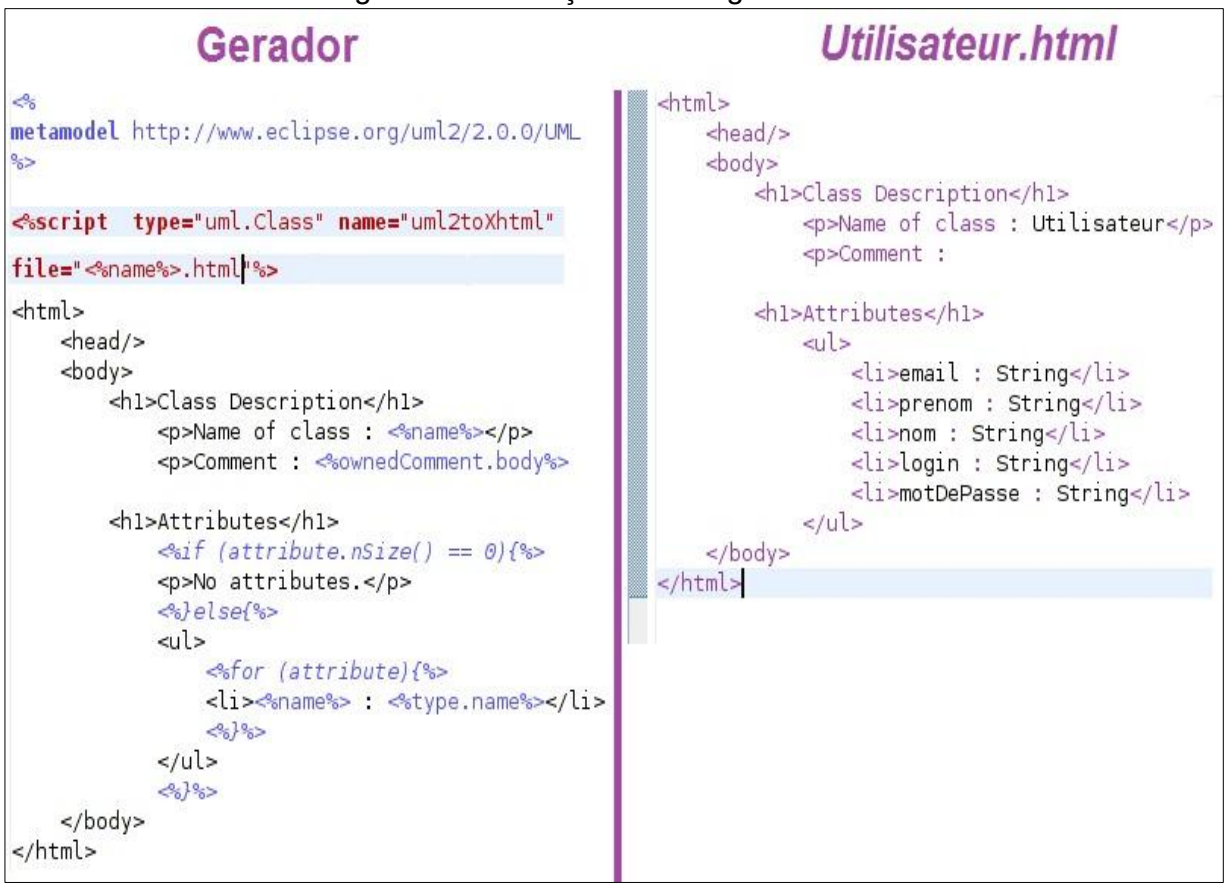
Na seção 4.3, aplicando regras de negócio na geração de código, a partir de um modelo UML, código fonte foi gerado. No entanto, para esse processo acontecer, o modelo UML inicialmente foi transformado em *ecore*, com intuito de adequar o modelo à linguagem padrão do EMF. Isto foi necessário para inserir regras de negócio,

que foram definidas diretamente no arquivo *generate.mtl,* responsável por conter o *script* para geração do código. Contudo apesar do modelo UML ter sido transformado em *ecore* antes de sua execução, e novas regras podendo ser inseridas, esta abordagem apresentou alguns pontos a serem destacados, tais como:

- O *script* do gerador precisa conter o mínimo possível de código da linguagem específica a ser gerada;
- E quando regras de negócios são inseridas, é imprescindível a utilização da linguagem em foco, predominando assim o arquivo *generate.mtl* com a linguagem de saída.

Estes pontos fazem que o arquivo *generate.mtl* receba grande quantidade de código fonte específico (*java*, html, entre outros). No entanto, este deveria receber em grande parte a linguagem M2T para se tornar o mais genérico possível. Sendo assim, com intuito de exemplificar um código gerado em html, conforme mostrado na Figura 41, pode-se notar que esse processo ainda é altamente dependente de codificação em linguagem fonte de saída.

Figura 41 - Geração de código em html



Fonte: ACCELEO (2015).

Desse modo, pode-se discutir até que ponto a inserção de regras de negócio são viáveis no MDD, mas para afirmar isto, é preciso realizar mais testes nessa abordagem, com a utilização de regras de negócio na geração automática de código. Em vista disso, vale destacar que as regras de negócio ainda não são genéricas como deveriam, sendo muitas vezes específicas demais de acordo com cada linguagem, essas regras de negócio se trata de um problema no MDD que precisa ser melhorado.

Mas vale a pena ressaltar a viabilidade da utilização do MDD quando o desenvolvimento é realizado a partir de um metamodelo, além do processo ser bem definido utilizando integralmente o padrão EMF, modelos e código são gerados de forma totalmente automática, apresentando sua viabilidade quando realmente são genéricos, ao ponto de produzirem diferentes modelos e consequentemente código em diferentes linguagens.

Conforme as abordagens desenvolvidas, entende-se que para o desenvolvimento dirigido a modelos ser posto em prática se tornando consideravelmente viável, é necessário que o arquivo gerador responsável por transformar os modelos em código, contenha a linguagem M2T empregada com maior regularidade e o modelo receba um grande número de informações, regras e restrições, até mesmo a tipagem deve ser genérica, conforme mostrado no Quadro 2.

Quadro 2 - Definindo tipagem no MDD

Demonstração de aplicação de tipagem no MDD	
nome-tipo: string, char	nome: nome
valor-tipo: int, float	valor: valor

Fonte: da pesquisa (2015)

De acordo com o Quadro 2, entende-se a necessidade da codificação ser genérica, para assim gerar código em linguagens distintas. Pois se a tipagem for definida com algum tipo de linguagem específica, a geração de código será voltada para aquela determinada linguagem, como por exemplo, nome-tipo *string*, apesar de variáveis do tipo *string* serem aceitas em *java*, as demais linguagens talvez não permita esse tipo. Portanto, a tipagem também deve ser genérica afim de se enquadrar em qualquer dialeto.

Algo relevante que deve ser considerado na hora da construção do metamodelo ou até mesmo do modelo UML, é a integração das classes, de modo a

elevar a importância de uma modelagem bem definida, e relacionamentos empregados de forma correta. Visto que para geração de código, e até mesmo para o arquivo *generate.mtl* reconhecer todas as classes do modelo, podendo coletar suas informações, é imprescindível que o modelo esteja integrado, caso contrário, cada classe deverá conter seu próprio gerador, ao invés de facilitar gerando todo o código por meio de um único *script* considerado o *main,* capaz de reconhecer o modelo como um todo.

Portanto de acordo com cada finalidade estabelecida, pode-se definir qual a melhor abordagem a ser utilizada. Por conta do MDD ser um processo automatizado para realização de processos de transformação de modelos em código de acordo com cada linha de pensamento proposta, diferentes meios do MDD podem ser empregados. Sendo assim nota-se necessária a realização de mais testes para ver até onde o MDD é viável e em que situações.

Vale ressaltar as dificuldades encontradas para conduzir o estudo, pois além de não encontrar materiais necessários sobre as respectivas abordagens estudadas, pouco foi encontrado sobre métodos de aplicação e execução dessas abordagens, no qual foram escassas as informações para conduzir os passos necessários a serem seguidos para realização de qualquer etapa do desenvolvimento, mesmo que as informações fornecidas pelas documentações das ferramentas utilizadas auxiliaram para realização do desenvolvimento. Sendo assim, problemas encontrados durante a execução dessas abordagens dificilmente possuíam suas soluções disponibilizadas em alguma fonte.

Em vista disso, além de estudar e utilizar o MDD por meio das abordagens tratadas no desenvolvimento do estudo, por conta dessa falta de materiais e informações sobre o MDD, este estudo especificou detalhadamente como realizar as etapas de transformações desde metamodelo em modelo, e modelo em código a partir de meios distintos, sendo assim concebível a colaborar com trabalhos futuros, que possam seguir cada etapa abordada aplicando o MDD, dando sequência ao estudo realizando novos testes afim de verificar sua viabilidade, e até que ponto pode-se atingir com o MDD.

Destacando os problemas encontrados durante o estudo, é relevante afirmar que as ferramentas e abordagens existentes atualmente possuem suas debilitações, mesmo que as principais ferramentas e abordagens do MDD foram estudadas e testadas conforme apresentado na capítulo 2, para assim escolher quais ferramentas

utilizar de acordo com aquelas que apresentaram melhor desempenho para execução das abordagens escolhidas, tornando evidente a necessidade de melhorar tanto a usabilidade dessas ferramentas como principalmente o que as mesmas são capazes de gerar.

Principais motivos nos quais as ferramentas utilizadas foram as escolhidas:

- São ferramentas gratuitas desenvolvidas especialmente para serem utilizadas no MDD.
- Todos os *plugins* utilizados para o desenvolvimento se comunicam entre si. Os *plugins* são de fácil acesso, estando disponíveis para *download* no *Eclipse*.
- O *Ecore Diagram, Acceleo* e o *Papyrus* foram testados antes de serem utilizados, além de possuírem suas respectivas documentações.
- Apresentaram melhor performance para execução do desenvolvimento dirigido a modelos perante as demais ferramentas testadas.

Portanto, pode-se dizer que no MDD existe um longo caminho a ser percorrido, em que muitos estudos devem ser realizados, ferramentas aprimoradas, abordagens já existentes melhoradas e novas abordagens criadas a fim de aperfeiçoar ainda mais esse processo.

6 CONSIDERAÇÕES FINAIS E TRABALHOS FUTUROS

Com o fim do trabalho proposto, pode-se reverenciar algumas considerações relevantes sobre o desenvolvimento dirigido a modelos, no qual vale reforçar que existem várias abordagens que dependem de diferentes tecnologias para serem aplicadas com êxito. Por meio do estudo, é possível aprimorar essas abordagens existentes para que um novo patamar do MDD seja alcançado, elevando seu nível de utilização de forma a torná-lo uma prática comum de acordo com o aumento de sua viabilidade.

Com as diferentes abordagens tratadas durante o estudo, e com seus passos bem descritos, pode-se desenvolver um outro estudo a partir do que foi realizado, além é claro, de repetir o que foi feito em diferentes práticas do dia a dia. Quando se pensa na construção de um metamodelo e pretende-se transformá-lo em modelo, é totalmente viável utilizar o MDD, pois neste aspecto só tende a trazer vantagens como a equivalência estabelecida entre o metamodelo e o modelo, além da praticidade de poder gerar código automático em conformidade com o modelo pré-definido.

Sabe-se que além das abordagens destacadas no estudo, existem outras nos quais também precisam ser melhoradas para que se possa torná-las usuais. Além dessas abordagens já existentes é importante ressaltar que novas abordagens aprimoradas precisam ser criadas juntamente com ferramentas aperfeiçoadas voltadas ao MDD, mas para que isso de fato aconteça, é necessário realizar estudos constantes sobre o MDD, e assim efetuar novos testes com intuito de comprovar sua viabilidade.

Desse modo, entende-se que o MDD tem como finalidade a construção e evolução de modelos de software com intuito de automatizar tarefas cotidianas do usuário, se tornando nítida a exigência de melhoria no processo de depuração no nível de modelagem, elevando a necessidade de evolução dos metamodelos e modelos, além de buscar uma padronização dessas abordagens existentes.

A fim de comprovar a viabilidade do desenvolvimento dirigido a modelos, e aprimorar o estudo, pode-se sugerir como trabalho futuro a geração de código em diferentes linguagens por meio de interface gráfica com a utilização de banco de dados efetuando a integração de outros diagramas UML para geração automática de código.

Por fim, entende-se que todos os objetivos deste estudo foram cumpridos abrangendo algumas abordagens do MDD e utilizando-as de forma a contribuir para trabalhos futuros que estão por vir.

REFERÊNCIAS

ALMEIDA, Vinícius Coelho de. *Uso da linguagem OCL no contexto de diagramas de classe da UML e programas em Java.* 2006. 75 f. Trabalho de Conclusão de Curso (Graduação em Ciência da Computação) – Instituto de Ciências Exatas, Universidade Federal de Minas Gerais, Belo Horizonte, 2006.

ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS. *NBR 6023:* 2002: informação e documentação: referências: elaboração. Rio de Janeiro: ABNT, 2002.

_____. *NBR 14724:* 2002: informação e documentação: trabalhos acadêmicos: apresentação. 3. ed. Rio de Janeiro: ABNT, 2011.

BÉZIVIN, Jean *et al.* Bridging the MS/DSL Tools and the Eclipse Modeling Framework. *Proceedings of the International Workshop on Software Factories,* Nantes, p.1-7, Mar. 2005.

BOOCH, Grady; RUMBAUGH, James; JACOBSON, Ivar. *UML:* guia do usuário. Rio de Janeiro: Elsevier, 2005.

BUDINSKY, Frank *et al. Eclipse modeling framework:* a developer's guide. Boston: Pearson Education, 2003.

BURDEN, Hakan; HELDAL, Rogardt; SILJAMAKI, Toni. Executable and translatable UML: how difficult can it be? *Asia-Pacific Software Engineering Conference,* Gothenburg, 18th, p. 1-8, 2011.

CABOT, Jordi. *The new executable UML standards:* fUML and Alf. 2011. Disponível em: <http://modeling-languages.com/new-executable-uml-standards-fuml-and-alf/>. Acesso em: 20 out. 2014.

CHARFI, Asma; MRAIDHA, Chokri; BOULET, Pierre. An optimized compilation of UML state machines. *Ieee International Symposium On Object/component/ service-oriented Real-time Distributed Computing,* Villeneuve D'ascq Cedex, p. 1-8, 2012.

DALY, Christopher. Method, apparatus and program storage device for representing eclipse modeling framework (EMF) ecore models in textual form. *International Business Machines Corporation,* Chattanooga, p.1-12, Apr. 2007.

FOWLER, Martin. *UML distilled:* a brief guide to the standard object modeling language. Boston: Pearson Education, 2004.

_____. *UML essencial:* um breve guia para a linguagem-padrão de modelagem de objetos. Porto Alegre: Artmed, 2005.

GÉNOVA, Gonzalo. Modeling and metamodeling in model driven development: what is a metamodel: the OMG's metamodeling infrastructure. *Knowledge Reuse Group,* Madrid, v. 100, n. 100, p. 9-36, May 2009.

GIL, Antônio Carlos. *Como elaborar projetos de pesquisa***.** 4. ed. São Paulo: Atlas, 2002.

GUEDES, Gilleanes Thorwald Araujo. *Um metamodelo UML para modelagem de requisitos em projetos de sistemas multiagentes.* 2012. 229 f. Tese (Doutorado em Computação) – Universidade Federal do Rio Grande do Sul, Porto Alegre, 2012.

JIANG, Ke; ZHANG, Lei; MIYAKE, Shigeru. Using OCL in executable UML. *Easst,* Beijing, v. 9, p. 1-12, 2007.

KNOWGRAVITY. *Cassandra.* Disponível em: <http://www.knowgravity.com/ cassandra-en>. Acesso em: 22 out. 2014.

LARMAN, Craig. *Applying UML and patterns.* 3. ed. Upper Saddle River: Pearson Education, 2005.

_____. *Utilizando UML e Padrões.* 3. ed. Porto Alegre: Artmed, 2007.

LEVENDOVSZKY, Tihamer *et al*. Model reuse with metamodel-based transformations. *Institute for Software Integrated Systems,* Budapest, p. 1-14, Oct. 2011.

MATRIX. *Dark matter systems.* Disponível em: <http://analysisdesignmatrix.com/>. Acesso em: 23 out. 2014.

MILICEV, Dragan. *Model-driven development with executable UML.* Indianapolis: Wiley Publishing, 2009.

LANGUAGES MODELING*. Clarifying concepts:* MBE vs MDE vs MDD vs MDA. Disponível em: <http://modeling-languages.com/clarifying-concepts-mbe-vs-mde-vs-mdd-vs-mda/>. Acesso em: 10 out. 2014.

NOMAGIC, Cameo Simulation Toolkit*. Executable UML with magicdraw.* Disponível em: <http://www.omg.org/news/meetings/tc/agendas/va/xUML_pdf/ Jankevicius.pdf>. Acesso em: 22 out. 2014.

OMG. Action language for foundational UML (Alf). *Object Management Group,* p. 1-455, Oct. 2013.

_____. Model driven architecture (MDA)*. Object Management Group,* p. 1-15, June 2014.

_____. MOF model to text transformation language. *Omg Available Specification***,** p. 1-38, Jan. 2008.

OMG. *OMG Unified Modeling Language Specification.* 2000.

PILONE, Dan; PITMAN, Neil. *UML 2.0 in a nutshell.* Sebastopol: O'reilly Media, 2005.

PRESSMAN, Roger S. *Engenharia de software:* uma abordagem profissional. Nova York: McGraw-Hill, 2011.

QUANTUM leaps. Disponível em: <http://www.state-machine.com/qm/>. Acesso em: 23 out. 2014.

REZENDE, Denis Alcides. *Engenharia de software e sistemas de informação.* Rio de Janeiro: Brasport, 2005.

SILVA, Alberto; VIDEIRA, Carlos. *UML, metodologias e ferramentas CASE.* Lisboa: Atlântico, 2001.

SINELABORERT. Disponível em: <http://www.sinelabore.com/doku.php>. Acesso em: 25 out. 2014.

SOLEY, Richard. Model driven architecture. *Object Management Group,* p. 1-12, Nov. 2000.

SOLUTIONS. *XUML - Executable UML.* 2014. Disponível em: <http://www.abstractsolutions.co.uk/XUML/executableumldescription.php>. Acesso em: 16 out. 2014.

SPARX SYSTEMS. *Model transformation.* 2015. Disponível em: <http://www.sparxsystems.com/enterprise_architect_user_guide/9.3/model_transformation/mdastyletransforms.html>. Acesso em: 14 abr. 2015.

STEINBERG, Dave *et al*. *EMF:* eclipse modeling framework. 2. ed. Boston: Pearson Education, 2009.

ACCELEO 2014. The Eclipse Foundation, 2014. Disponível em: <https://www.eclipse.org/ acceleo/>. Acesso em: 24 out. 2014.

ACCELEO 2015. The Eclipse Foundation, 2015. Disponível em: <http://www.acceleo.org/pages/first-generator-module/e>. Acesso em: 14 abr. 2015.

PAPYRUS. The Eclipse Foundation, 2014. Disponível em: <http://www.eclipse.org/ papyrus/>. Acesso em: 22 out. 2014.

VOGEL, Lars. *Eclipse modeling framework (EMF):* tutorial. Disponível em: <http://www.vogella.com/tutorials/EclipseEMF/article.html>. Acesso em: 01 abr. 2015.