UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO APLICADA

GLAUBER ZÁRATE VALENÇA

CONTRIBUIÇÃO PARA A MATERIALIZAÇÃO DO PARADIGMA ORIENTADO A NOTIFICAÇÕES (PON) VIA *FRAMEWORK* E *WIZARD*

DISSERTAÇÃO

CURITIBA
2012

GLAUBER ZÁRATE VALENÇA

CONTRIBUIÇÃO PARA A MATERIALIZAÇÃO DO PARADIGMA ORIENTADO A NOTIFICAÇÕES (PON) VIA *FRAMEWORK* E *WIZARD*

Dissertação apresentada ao Programa de Pós-Graduação em Computação Aplicada - PPGCA - da Universidade Tecnológica Federal do Paraná – UTFPR, como requisito parcial para obtenção do grau de “Mestre em Computação” – Área de Concentração: Engenharia de Software.

Orientador: Prof. Dr. Jean Marcelo Simão
Co-orientador: Prof. Msc. Robson Ribeiro Linhares

CURITIBA
2012

Dados Internacionais de Catalogação na Publicação

V152 Valença, Glauber Zárate
Contribuição para a materialização do paradigma orientado a notificações (PON) via *framework e wizard* / Glauber Zárate Valença. — 2012.
205 f. : il. ; 30 cm

Orientador: Jean Marcelo Simão.
Coorientador: Robson Ribeiro Linhares.
Dissertação (Mestrado) – Universidade Tecnológica Federal do Paraná. Programa de Pós-graduação em Computação Aplicada, Curitiba, 2012.
Bibliografia: f. 197-205.

1. Paradigma orientado a notificações. 2. Interface de programas aplicativos (Software). 3. Software de aplicação – Desenvolvimento. 4. Simulação (Computadores). 5. Computação – Dissertações. I. Simão, Jean Marcelo, orient. II. Linhares, Robson Ribeiro, coorient. III. Universidade Tecnológica Federal do Paraná. Programa de Pós-graduação em Computação Aplicada. IV. Título.

CDD (22. ed.) 004

Biblioteca Central da UTFPR, Campus Curitiba

Ministério da Educação
Universidade Tecnológica Federal do Paraná
Câmpus Curitiba
Diretoria de Graduação e Educação Profissional
Departamento Acadêmico de Informática – DAINF
Programa de Pós-Graduação em Computação Aplicada - PPGCA
Mestrado Profissional

Título da Dissertação Nº: 02

"Contribuição para a Materialização do Paradigma Orientado a Notificações (PON) via *Framework* e *Wizard*"

por

Glauber Zárate Valença

Esta dissertação foi apresentada como requisito parcial à obtenção do grau de MESTRE COMPUTAÇÃO APLICADA - Área de Concentração: Engenharia de Sistemas Computacionais, pelo PPGCA - Programa de Pós-Graduação em Computação Aplicada - Mestrado Profissional – da Universidade Tecnológica Federal do Paraná - UTFPR - Câmpus Curitiba, às 14 horas do dia 23 de agosto de 2012. O trabalho foi aprovado pela Banca Examinadora, composta pelos professores:

Prof. **Jean Marcelo Simão, Dr.**
presidente - (UTFPR - CT)

Prof. **Paulo Cezar Stadzisz, Dr.**
(UTFPR - CT)

Prof. **Fabiano Silva, Dr.**
(UFPR)

Prof. **João Alberto Fabro, Dr.**
(UTFPR-CT)

Prof. **Carlos Raimundo Erig Lima, Dr.**
(UTFPR-CT) - Suplente

DAINF
Departamento Acadêmico de Informática - DAINF
Programa de Pós-Graduação em Computação Aplicada - PPGCA
Câmpus Curitiba
Bloco B - ramal 3310-**4644** • Fax: 3310-**4646**

RESUMO

VALENÇA, Glauber Z. CONTRIBUIÇÃO PARA A MATERIALIZAÇÃO DO PARADIGMA ORIENTADO A NOTIFICAÇÕES (PON) VIA *FRAMEWORK* E *WIZARD.* Dissertação – Programa de Pós Graduação em Computação Aplicada, Universidade Tecnológica Federal do Paraná. Curitiba, 2012.

O emergente Paradigma Orientado a Notificações (PON) está materializado em um *Framework* desenvolvido na linguagem de programação C++. Este foi projetado para fornecer uma *Application Programming Interface* (API) e estruturas de alto nível que facilitasse o desenvolvimento de *software* segundo sua orientação. Entretanto, isto induz a uma sobrecarga de processamento computacional em cada aplicação PON. Ainda, uma embrionária interface amigável denominada *Wizard* foi concebida para prover recursos de alto nível para o desenvolvimento de certas aplicações PON. Assim, este trabalho propõe uma nova materialização do Framework PON e a evolução de sua interface *Wizard.* Ao final, estes são validados por meio de comparações quantitativas e qualitativas em relação aos seus artefatos precedentes. A comparação quantitativa diz respeito a desempenho de instâncias de *Framework*, enquanto que a qualitativa sobre facilidades de suas composições em relação ao conjunto interface e *Framework* PON.

Palavras-Chaves: Paradigmas de Programação, Materialização do Paradigma Orientado a Notificações, Concepção de Aplicações do PON via *Wizard.*

ABSTRACT

VALENÇA, Glauber Z. CONTRIBUITION TO THE MATERIALIZATION OF NOTIFICATION ORIENTED PARADIGM (NOP) VIA FRAMEWORK AND WIZARD. Dissertação – Programa de Pós Graduação em Computação Aplicada, Universidade Tecnológica Federal do Paraná. Curitiba, 2012.

The Emerging Notification Oriented Paradigm (NOP) is materialized in a Framework developed in the C++ language programming. The NOP Framework was designed to provide an Application Programming Interface (API) and high-level structures that would facilitate the development of software according to their orientation. However, this implies to a computational processing overhead of each NOP application. Still, an embryonic friendly Interface called Wizard was designed to provide high-level resource for the development of certain NOP applications. Thus, this work proposes a new version of materialization of NOP Framework and the evolution of its Wizard Interface. At the end, these are validated by quantitative and qualitative comparisons, in a relation to its previous artifacts. The quantitative comparisons are referent of the performance to the Framework instances, whereas the qualitative comparisons are about the facilities of yours compositions.

Keywords: Programming Paradigm, Materialization of Notification Oriented Paradigm, Building NOP Applications via Wizard.

RESUMO ESTENDIDO

Os atuais paradigmas de programação de *software*, mais precisamente o Paradigma Imperativo (PI) e o Paradigma Declarativo (PD), apresentam deficiências que afetam: (a) o desempenho das aplicações; (b) a facilidade de programação no PI ou as funcionalidades na programação no PD; e (c) a obtenção de 'desacoplamento' (ou acoplamento mínimo) entre os módulos de *software*, o que dificulta seus reaproveitamentos bem como o uso de programação distribuída.

Na verdade, o PI e o PD são similares ao serem baseados em buscas ou no percorrer sobre entidades passivas, que consistem em dados (e.g. fatos ou estados de variáveis ou de atributos de outras entidades) e comandos de decisão (e.g. expressões causais como *se-então* ou regras). Nestes, as buscas afetam o desempenho das aplicações por gerar redundâncias de processamento e dificultam o alcance de uma dependência mínima dos módulos por gerar acoplamento implícito entre eles.

Com o objetivo de prover uma solução para este conjunto de deficiências, foi desenvolvido o Paradigma Orientado a Notificações (PON) derivado de uma teoria de controle e inferência precedente chamado Controle Orientado a Notificação (CON). O PON se inspira nos conceitos do PI (e.g. objetos) e do PD (e.g base de fatos e regras) oferecendo melhores qualidades do que estes paradigmas. Basicamente, o PON usa objetos para tratar de fatos e regras na forma de composições de outros objetos menores. Todos estes objetos apresentam características comportamentais de certa autonomia, independência, reatividade e particularmente colaboração por meio de notificações pontuais. Estas características são voltadas à realização participativa das funcionalidades do *software* por esses objetos.

Outrossim, o PON encontra-se materializado em um *Framework* desenvolvido na linguagem de programação C++. Este foi projetado para fornecer uma *Application Programming Interface* (API) e estruturas de alto nível que facilitasse o desenvolvimento de *software* segundo sua orientação. Ademais, sua composição para realizar o cálculo lógico causal ímpar do PON é formada de estruturas de alto nível como classes que utilizam contêineres da *Standard Template Library* (*STL*). Tal camada implica em estruturas de dados computacionalmente custosa uma vez que, objetos de classes da *STL* são formados por uma estrutura de coleção genérica.

Ainda, uma embrionária interface amigável inicialmente denominada *Wizard* CON (Controle Orientado a Notificação) foi concebida anteriormente. A principal responsabilidade da ferramenta *Wizard* CON é a composição de instâncias de controle que permite escrever regras e gerar automaticamente, a partir delas, as entidades *Rules.* Isto facilita o processo de criação de instâncias de controle que antes se dava tecnicamente por meio da linguagem de programação C++, ainda que apoiado em um *Framework* do meta-modelo de controle também em C++.

Neste âmbito, este trabalho propõe uma nova materialização do PON que se constitui de uma evolução da versão pré-existente do *Framework* PON e interface amigável (*Wizard*-CON), os quais são validados por meio de comparações quantitativas e qualitativas em relação aos artefatos precedentes. A comparação quantitativa diz respeito a desempenho de instâncias de *Framework*, enquanto que a qualitativa sobre facilidades de suas composições em relação ao conjunto interface e *Framework* PON.

As evoluções realizadas no *Framework* refletiram de forma significativa no desempenho das aplicações PON. Para isto, técnicas de otimizações locais, técnicas de otimizações globais e demais refatorações foram aplicadas em diversos pontos do código fonte. Neste âmbito, uma das principais otimizações refere-se à estrutura de dados que comporta o processo de notificação do PON. Ao final desta fase de implementação, obteve-se ganhos substanciais de desempenho em relação ao seu artefato precedente, as quais variam entre 40% a 70% entre as aplicações utilizadas como casos de estudo.

Ainda, quanto às melhorias projetadas e implementadas sobre a ferramenta *Wizard* PON, as mesmas refletiram vantagens na concepção de aplicações PON. O estudo de caso elaborado, de fato, apresentou vantagens em relação a sua implementação predecessora que se dava de maneira “manual”.

Em suma, este trabalho apresenta a evolução e validação da materialização precedente do *Framework*-Interface PON, sob o viés de desempenho e facilidade de sua composição através de aplicações PON. Ainda, ao final do projeto, percebe-se um novo direcionamento das pesquisas sobre o PON visando a implementação de um compilador integrado ao seu respectivo ambiente de desenvolvimento propriamente dito.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ALGORITMOS

LISTA DE SIGLAS

SIGLA	Tradução Original	Tradução Português
API	*Application Programming Interface*	Interface de Programação para Aplicação
ASP	*Applicatin Service Provider*	Provedor de Serviços para Aplicação
CASE	*Computer-Aided Software Engineering*	Engenharia de Software Apoiada por Computador
CONN	Controle Orientado a Notificação	Notification Oriented Controll
CH	Controle Holônico	Holonic Controll
CRUD	*Create Retrieve Update Delete*	Criação Pesquisa Atualização Exclusão
DON	Desenvolvimento Orientado a Notificações	Notification Oriented Development
FBE	*Fact Base Element*	Elemento da Base de Fatos
FIFO	*Fist In, First Out*	Primeiro a Entrar, Primeiro a Sair
GCC	*GNU Compiler Collections*	GNU Coleções de Compiladores
GOF	*Gang of Four*	Gangue dos Quatro
GNU	*Gnu is Not Unix*	Gnu Não é Linux
IDE	*Integrated Development Environment*	Ambiente de Desenvolvimento Integrado
JEE	*Java Enterprise Edition*	Edição Empresarial Java
JPA	*Java Persistence API*	API de Persistência para Java
JSF	*Java Server Faces*	*Java Server Faces*
LIFO	*Last in, First Out*	Último a Entrar, Primeiro a Sair
MDA	*Model-Driven Architecture*	Arquitetura Dirigida a Modelo
MVC	*Model View Controller*	Modelo Visão Controle
OO	Orientação a Objetos	Objected Oriented
PC	*Personal Computer*	Computador Pessoal
PON	Paradigma Orientado a Notificações	Notification Oriented Paradigm
PHP	*Php Hypertext Preprocessor*	*Php Hypertext Preprocessor*
POO	Paradigma Orientado a Objetos	Object Oriented Paradigm
PD	Paradigma Declarativo	Declarative Paradigm
PI	Paradigma Imperativo	Imperative Paradigm
RAM	*Random Access Memory*	Memória de Acesso Aleatório
RMI	*Remote Method Invocation*	Invocação de Método Remoto
RUP	*Rational Unified Process*	Processo Unificado da Rational
SBR	Sistema Baseado em Regras	*Rules* Based Systems
STL	*Standard Template Library*	Biblioteca de Template Padrão
SQL	*Structured Query Language*	Linguagem de Questão Estruturada
UML	*Unified Modeling Language*	Linguagem de Modelagem Unificada
XML	*eXtensible Markup Language*	Linguagem de Marcação eXtensível

LISTA DE SÍMBOLOS

Δ: Delta - Variação de desempenho obtida entre a versão do artefato original do *Framework* PON e sua versão atualizada.

SUMÁRIO

1 INTRODUÇÃO

Este trabalho relaciona-se diretamente com o Paradigma Orientado a Notificações (PON). Em linhas gerais, o PON resolve certos problemas existentes nos paradigmas usuais de programação, nomeadamente o Paradigma Declarativo (PD) e o Paradigma Imperativo (PI). Na verdade, o PON unifica as principais características e as vantagens do PD (*e.g.* representação do conhecimento em regras) e do PI (*e.g.* flexibilidade de expressão e nível apropriado de abstração), resolvendo em termos de modelo várias de suas deficiências e inconvenientes em aplicações de *software*, supostamente desde ambientes monoprocessados a completamente multiprocessados (SIMÃO e STADZISZ, 2008; 2009a; BANASZEWSKI, 2009; SIMÃO *et al.*, 2012a). Outrossim, a materialização dos conceitos do PON foi inicialmente concebida através de um *Framework* prototipal desenvolvido na linguagem de programação C++, sendo esta materialização o objeto de estudo deste trabalho.

Este capítulo introdutório está organizado como segue. A seção 1.1 apresenta de forma sucinta a classificação das linguagens de programação em paradigmas de programação. A seção 1.2 descreve as origens do chamado Paradigma Orientado a Notificações (PON). A seção 1.3 apresenta o PON propriamente dito. A seção 1.4 relata sobre uma ferramenta prototipal *Wizard* relativa ao Controle Orientado a Notificações (CON). A seção 1.5 aborda o problema sobre o qual este trabalho de pesquisa atuará. A seção 1.6 descreve as principais justificativas de elaboração deste projeto de pesquisa. A seção 1.7 apresenta o objetivo geral e os objetivos específicos. Por fim, a seção 1.8 apresenta a organização dos capítulos que compõem este projeto.

1.1 PARADIGMAS DE PROGRAMAÇÃO

Na ciência da computação, o termo paradigma é empregado como uma maneira de abstrair o pensamento do desenvolvedor em uma determinada estrutura computacional, capaz de definir a execução de um programa. Os paradigmas se diferem em conceitos e abstrações utilizadas para representar os elementos de um

programa (*e.g.* objetos, funções, variáveis, restrições etc.) e a maneira com que esses interagem de maneira a ditar o fluxo de execução de tal programa (*e.g.* atribuições, avaliações causais (*se-então*), repetições, empilhamento, recursividade etc.) (FLOYD, 1979).

As linguagens de programação, por sua vez, seguem paradigmas de programação, que definem suas características positivas e negativas. Exemplos de paradigmas são o Paradigma Imperativo (PI) e o Paradigma Declarativo (PD). Em geral, o primeiro apresenta maior dificuldade de programação, mas proporciona maior flexibilidade em termos algorítmicos e de acesso a *hardware*. O segundo, por sua vez, tende a facilitar a programação, mas se apresenta normalmente menos flexível nesses termos considerados (SIMÃO e STADZISZ, 2008) (BANASZEWSKI, 2009) (SIMÃO, *et.al.*, 2012a).

Outrossim, os paradigmas de programação atuais, ressaltando PI e PD, apresentam algumas deficiências, como redundâncias e acoplamentos de expressões causais. Isto existe devido ao fato do processo de avaliação ou inferência causal (*i.e.* execução de expressões *se–então* por assim dizer) ser baseado em pesquisa, normalmente monolítica, o que causa comprometimento de desempenho, reuso e também distribuição (SIMÃO, STADZISZ, 2008, 2009).

De fato, tais expressões causais são frequentemente avaliadas desnecessariamente, degradando seu desempenho. Isto pode ser exemplificado em um código em POO, sendo o POO considerado o paradigma dominante do PI. Mais precisamente, o código seria um conjunto de expressões se-então em POO, que avaliam os estados de objetos dentro de um laço de repetição dito ‘infinito’. Cada expressão condicional avalia certos estados dos atributos de objetos e, se aprovada, chama alguns métodos dos objetos que podem mudar os estados dos atributos. Isto é apresentado no Algoritmo 1 (BROOKSHEAR, 2006) (GABBRIELLI e MARTINI, 2010) (SIMÃO e STADZISZ, 2008, 2009).

```
1  ...
2  enquanto ( verdade ) faça
3      se ( ( objeto1.atributo1 = 1 ) e ( objeto2.atributo1 = 1 ) ) então
4           objeto1.método1(); objeto2.método2();
5      fim-se
6      . . .
7      se ( ( objeto1.atributoN = N ) e ( objeto2.atributoN = N ) ) então
8           objeto1.métodoN(); objeto2.métodoN();
9      fim-se
10 fim-enquanto
```

Algoritmo 1 – Pseudocódigo do Paradigma Imperativo

Neste exemplo, é observado que o laço de repetição força o percorrimento ou pesquisa avaliativa (ou ainda inferência) de todas as condições de maneira sequencial. Entretanto, muitas delas são desnecessárias porque somente alguns objetos têm o valor de seus atributos modificado. Isto até pode ser considerado não importante neste exemplo simples e pedagógico, sobretudo se o número (n) de expressões causais for pequeno. Entretanto, se for considerado um sistema complexo, integrando muitas partes como aquela, pode-se ter uma grande diferença de desempenho (SIMÃO e STADZISZ, 2008, 2009) (SIMÃO, *et.al.*, 2012).

Na PI, tanto em linguagens procedimentais quanto em linguagens Orientadas a Objetos (OO), o programador determina para o computador “o que fazer”, “como fazer”, e “em que ordem fazer” por meio de comandos (*e.g.* comandos de decisão e repetição) que são compilados para código de máquina ou até mesmo interpretados por uma camada sobre o código de máquina. Nesta abordagem, as linhas de códigos implementadas tendem a se tornar interdependentes e as expressões causais são avaliadas passivamente, ocasionando as chamadas redundâncias estruturais e temporais na execução dos programas (SIMÃO, STADZISZ, 2008, 2009).

Oportunamente, a redundância estrutural ocorre quando uma expressão lógica não é compartilhada (ou mais precisamente, quando seu valor booleano não é compartilhado) entre outras expressões causais pertinentes, causando reavaliações desnecessárias. A redundância temporal, por sua vez, ocorre quando uma avaliação (lógica-causal) é realizada repetidas vezes sobre um elemento já avaliado e inalterado. De fato, ambos os tipos de redundância ocorrem no código exemplo apresentado, o qual poderia ser otimizado via esforço de programação adicional. Isto, em escala, torna a programação PI difícil, o que se constitui em outro problema (BANASZEWSKI, 2009) (SIMÃO, STADZISZ, 2008, 2009).

A seu turno, no PD, do qual o principal exemplo são os Sistemas Baseados em Regras (SBR), a construção de *software* seria mais simples. A programação seria feita pela descrição de fatos relativos a entidades de um determinado domínio e a descrição de relações/regras causais acerca destas entidades factuais em um ambiente ou linguagem de alto nível. A execução destas regras baseadas nos fatos se daria automaticamente pelo uso de motores de inferência, sem preocupações outras no tocante a desenvolvimento de algoritmos. Um motor ou máquina de

inferência deduz novos fatos ou ações a partir de confrontações dos fatos existentes para com as regras causais (ou decisionais), sendo que isto se dá por pesquisas em bases causais (i.e. regras) e factuais (e.g. estados de atributos de entidades) (BANASZEWSKI, 2009) (SIMÃO, STADZISZ, 2008, 2009).

Neste âmbito, algoritmos de inferência incrementais, como: *RETE* (FORGY, 1982), *TREAT* (MIRANKER, 1987), *LEAPS* (MIRANKER, BRANT, LOFASO e GADBOIS, 1990) e *HAL* (LEE e CHENG, 2002) foram concebidos para minimizar problemas de processamento na fase de *matching* dos SBRs justamente pela eliminação de considerável parte das redundâncias estruturais e temporais. Não obstante a organização e mesmo eficientes algoritmos de inferência (como os supracitados), a programação em PD normalmente é computacionalmente cara em termos de processamento de estruturas de dados utilizadas para evitar as citadas redundâncias, sendo normalmente mais caras que a POO não obstante as redundâncias desta (SCOTT, 2000) (SIMÃO e STADZISZ, 2008, 2009).

Outra característica implícita neste paradigma está relacionada à estrutura genérica pelo qual as regras tratam os fatos, ou seja, em termos de classes, ao invés de objetos (BANASZEWSKI, 2009). Muito embora isto possa permitir alguma economia de uso de memória, dificulta a otimização de processamento uma vez que é necessário pesquisar/combinar objetos específicos a luz de regras genéricas. Ademais, tal abordagem dificultaria a implementação em ambientes distribuídos uma vez que objetos de uma mesma classe seriam agrupados juntos (BANASZEWSKI, 2009) (SIMÃO, *et.al.*, 2012).

Ainda, em uma análise mais profunda, PI e PD são similares no tocante à inferência, a qual normalmente se dá por entidades monolíticas baseadas em pesquisas sobre entidades passivas ou voltadas à passividade (*i.e.* fracamente reativas) que conduzem a programas com passos de execução interdependentes. Estas características contribuem para a existência de sobre-processamento e forte acoplamento entre expressões causais e estrutura de fatos/dados, o que dificulta inclusive a execução dos programas de maneira otimizada, bem como paralela ou distribuída (GABBRIELLI e MARTINI, 2010) (SIMÃO e STADZISZ, 2008, 2009). Ainda que haja alternativas outras de programação, como orientações a eventos e mesmo orientação a dados, elas apenas atenuam ou fatoram o problema, conforme discutido em (BANASZEWSKI, 2009) (SIMÃO e STADZISZ, 2008, 2009) (SIMÃO, *et.al.*, 2012a).

Neste âmbito, com a motivação de minimizar as dificuldades acima descritas, uma nova maneira de desenvolvimento de *software* (salientando a programação) foi elaborada e denominada Paradigma Orientado a Notificações (PON) (BANASZEWSKI, 2009) (SIMÃO e STADZISZ, 2008, 2009) (WIECHETECK, 2011) (WIECHETECK, *et. al*., 2011) (SIMÃO, *et.al.*, 2012a).

1.2 ORIGENS DO PARADIGMA ORIENTADO A NOTIFICAÇÕES (PON)

O Paradigma Orientado a Notificações (PON) embrionariamente remonta a uma solução de controle aplicada sobre um simulador chamado ANALYTICE II (SIMÃO, 2001) (SIMÃO, *et. al.*, 2001) (SIMÃO e STADZISZ, 2002), (SIMÃO, STADZISZ e KÜNZLE, 2001, 2003). Esta ferramenta de projeto e simulação ANALYTICE II se constitui em tema de pesquisa prioritário do então Laboratório de Sistemas Inteligentes de Produção (LSIP) do CPGEI/UTFPR, a qual visava inclusive o domínio do desenvolvimento de (novas) tecnologias de simulação e de controle (SIMÃO, 2001, 2005).

Desta solução de controle voltada ao ANALYTICE II surgiu uma solução chamada de Controle Holônico (CH) (SIMÃO, 2005); (SIMÃO e STADZISZ, 2002, 2008, 2009); (SIMÃO, STADZISZ e MOREL, 2006); (SIMÃO, STADZISZ e TACLA, 2009). A finalidade disto foi a de propor novos mecanismos de controle que suprissem as necessidades relacionadas com os sistemas modernos de produção, como agilidade e variabilidade na produção, onde as colaborações entre as entidades de manufatura (*e.g.* equipamentos e produtos) são realizadas por um CH. Este CH permitiria propriedades como criação de instância em alto nível, exploração da flexibilidade da capacidade de manufatura instalada, tempo de processamento apropriado, distribuição e robustez (SIMÃO, 2005).

Nessa solução de CH em particular, o qual se constitui em um metamodelo de CH, cada relação causal do sistema é concebida na forma de regra causal e instanciada na forma de entidades denominadas *Rule*. As *Rules* decidem sobre as cooperações de entidades (*e.g.* equipamentos e produtos inteligentes) com base em dados específicos recebidos deles. Para tal, o metamodelo oferece um mecanismo de inferência ímpar que permite compor e executar apropriadamente *softwares* de CH no domínio de sistemas de manufatura modernos. Este fato é confirmado pela

análise do metamodelo sobre diversas perspectivas e também pela implementação na forma de um arquétipo (ou *framework*) e validação via instanciação sobre o ANALYTICE II (SIMÃO, 2005); (SIMÃO *et.al.*, 2008); (SIMÃO e STADZISZ, 2002); (SIMÃO, STADZISZ e MOREL, 2006); (SIMÃO *et. al.,* 2008); (SIMÃO, STADZISZ, TACLA, 2009).

Particularmente, a inferência das expressões causais ocorre de um modo ímpar, por meio de uma cadeia de notificações pontuais entre os elementos envolvidos. Esta abordagem baseada em notificações evita pesquisas e, consequentemente, a ocorrência de avaliações e acoplamentos desnecessários. Este meta-modelo de inferência, baseada em cadeias notificantes, fomentou novos estudos, cunhando o que se conhece atualmente como Controle Orientado a Notificações (CON), Inferência Orientada a Notificações (ÍON) e particularmente Paradigma Orientado a Notificações (PON). O PON se propõe a amenizar algumas das deficiências dos atuais paradigmas em relação a avaliações causais desnecessárias e acopladas, evitando o processo de inferência monolítico baseado em pesquisas, uma vez que seu mecanismo de inferência é baseado no relacionamento de entidades computacionais notificantes (BANASZEWSKI *et al.*, 2007) (BANASZEWSKI, 2009) (SIMÃO e STADZISZ, 2008, 2009) (SIMÃO, *et.al.*, 2012).

1.3 PARADIGMA ORIENTADO A NOTIFICAÇÕES (PON)

Em linhas gerais, o chamado Paradigma Orientado a Notificações (PON) resolve certos problemas existentes nos paradigmas usuais de programação, nomeadamente o Paradigma Declarativo (PD) e o Paradigma Imperativo (PI). Na verdade, o PON unifica as principais características e as vantagens do PD (*e.g.* representação do conhecimento em regras) e do PI (*e.g.* flexibilidade de expressão e nível apropriado de abstração), supostamente resolvendo em termos de modelo várias de suas deficiências e inconvenientes em aplicações de *software,* desde ambientes monoprocessados a completamente multiprocessados (SIMÃO e STADZISZ, 2008, 2009) (BANASZEWSKI, 2009) (SIMÃO, *et.al.*, 2012a).

O PON permite desacoplar expressões causais do código-fonte, ao considerar cada uma destas e seus fatos relacionados como entidades

computacionais (objetos de *software* nas atuais implementações) notificantes, o que permite desempenho apropriado e distribuição (se houver algum multiprocessamento). Isto é diferente dos programas usuais do PI (salientando os Orientados a Objetos - OO) e do PD (salientado os chamados Sistemas Baseados em Regras – SBR) onde expressões causais são passivas e (fortemente) acopladas a outras partes do código, além de haver algum ou mesmo muito desperdício de processamento (conforme o caso) (SIMÃO e STADZISZ, 2008, 2009) (SIMÃO, *et.al.*, 2012).

No PON, a entidade computacional que trata de uma expressão causal é chamada de *Rule*. As *Rules* gerenciam o conhecimento sobre qualquer comportamento causal no sistema. O conhecimento causal de uma *Rule* provém normalmente de uma regra (se-então), o que é uma maneira natural de expressão deste tipo de conhecimento. Não obstante, este conhecimento causal pode ser representado em outro formalismo equivalente quando se fizer pertinente (salientando aqui as redes de Petri) (SIMÃO e STADZISZ, 2008, 2009) (WIECHETECK, 2011).

A Figura 1 apresenta um exemplo de *Rule*, justamente na forma de uma regra causal. Uma *Rule* é composta por uma *Condition* (ou Condição) e por uma *Action* (ou Ação), cf. mostra a figura em questão. A *Condition* trata da decisão da *Rule*, enquanto a *Action* trata da execução das ações dela. Assim sendo, *Condition* e *Action* trabalham para realizar o conhecimento causal da *Rule*. Na verdade, tanto a *Condition* quanto a *Action* são entidades computacionais agregadas em uma *Rule* (SIMÃO e STADZISZ, 2008, 2009).

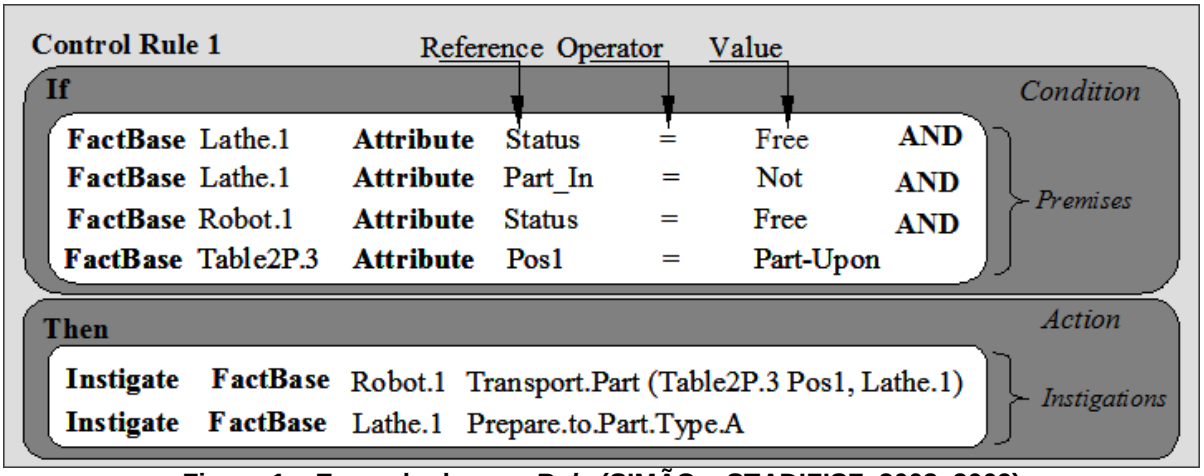


Figura 1 – Exemplo de uma *Rule* (SIMÃO e STADIZISZ, 2008, 2009)

A *Rule* apresentada na Figura 1 faria parte de um sistema de controle de manufatura avançado, onde equipamentos integrados ao computador são tratados a partir de entidades computacionais (*i.e. smart-drivers*). A *Condition* desta *Rule* lida com a decisão de transporte de peça a partir de uma 'Mesa' (*Smart-Table*) para um 'Torno' (*Smart-Lathe*) utilizando um 'Robô' (*Smart-Robot*). Na verdade, cada um destes elementos computacionais, analisáveis por *Conditions*, são chamados de *FactBaseElements* (*FBE*) no PON (SIMÃO e STADZISZ, 2008, 2009).

Conforme ilustrado na Figura 1, a *Condition* daquela *Rule* em questão é composta por três *Premises* (ou Premissas) que se constituem em outro tipo de entidade computacional. Estas *Premises* em questão fazem as seguintes verificações sobre os *FBEs*: a) o Torno está livre e sem peça? b) o Robô está livre? c) há alguma peça na posição 2 da Mesa?. Assim sendo, conclui-se (em geral) que os estados dos atributos dos *FBEs* compõem os fatos a serem avaliados pelas *Premises* (SIMÃO e STADZISZ, 2008, 2009).

Na verdade, os estados de cada um dos atributos de um *FBE* são tratados por meio de uma entidade chamada *Attribute* (ou Atributo). Além do mais, e principalmente, para cada mudança de estado de um *Attribute* de um *FBE*, ocorrem automaticamente avaliações (lógicas) somente nas *Premises* relacionadas com eventuais mudanças nos seus estados. Igualmente, a partir da mudança de estado das *Premises*, ocorrem automaticamente avaliações somente nas *Conditions* relacionadas com eventuais mudanças de seus estados (SIMÃO e STADZISZ, 2008, 2009) (SIMÃO, *et.al.*, 2012a).

Isto tudo se dá por meio de uma cadeia de notificações entre entidades computacionais, cf. ilustra a Figura 2, o que se constitui no ponto central da inovação do PON. Em suma, cada *Attribute* notifica as *Premises* relevantes sobre seus estados somente quando se fizer efetivamente necessário. Cada *Premise*, por sua vez, notifica as *Conditions* relevantes dos seus estados usando o mesmo princípio. Baseado nestes estados notificados é que a *Condition* pode ser aprovada ou não. Se a *Condition* é aprovada, a respectiva *Rule* pode ser ativada executando sua *Action* (SIMÃO e STADZISZ, 2008, 2009) (SIMÃO, *et.al.*, 2012a).

Uma *Action* também é uma entidade computacional que se conecta a entidades computacionais de outro tipo, as *Instigations* (ou Instigações). No exemplo dado, a *Action* contém duas *Instigations* para: a) ativar o Robô para transportar peças da Mesa (posição 2) para o Torno; e b) preparar o Torno para receber a peça.

Efetivamente, o que uma *Instigation* faz é instigar um ou mais métodos responsáveis por realizar serviços ou habilidades de um *FBE* (SIMÃO e STADZISZ, 2008, 2009).

Certamente, cada método de *FBE* é também tratado por uma entidade computacional, que é chamado de *Method* (ou Método). Geralmente, a execução de um *Method* muda o estado de um ou mais *Attributes*. Na verdade, os conceitos de *Attribute* e *Method* representam uma evolução dos conceitos de atributos e métodos de classe do POO. A diferença é o desacoplamento implícito da classe proprietária e a "inteligência" colaborativa pontual para com *Premises* e *Instigations* (SIMÃO e STADZISZ, 2008, 2009) (SIMÃO, *et.al.*, 2012a).

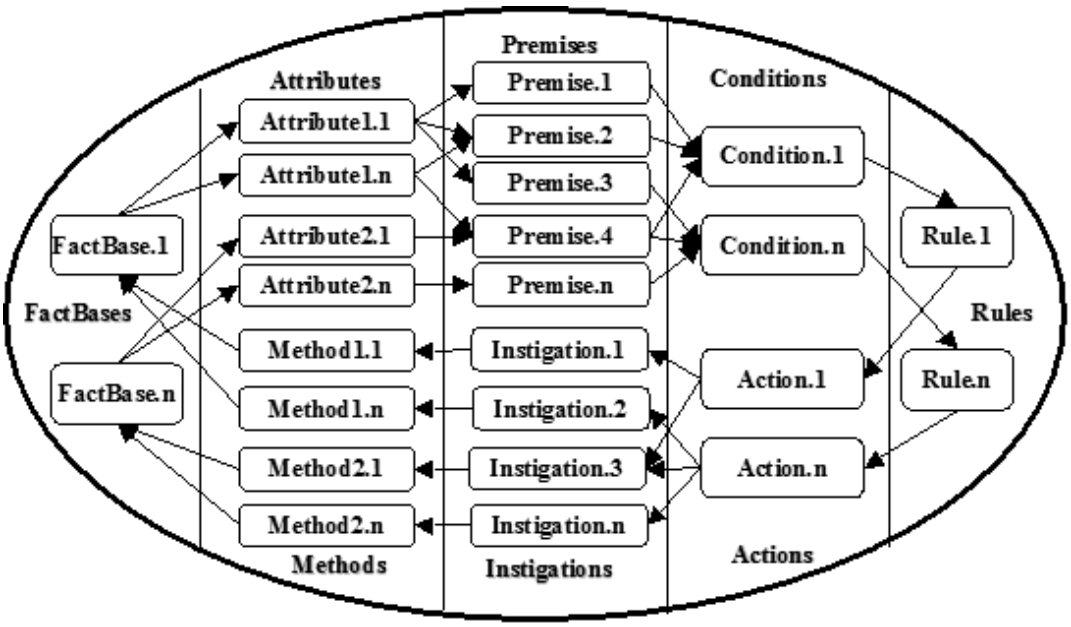


Figura 2 – Cadeia de Notificações (SIMÃO e STADZISZ, 2008, 2009)

Com isto considerado, salienta-se que a ciência de qual elemento deve notificar qual, se dá na própria composição das *Rules*, que poderia ser feito em um ambiente amigável na forma de regras causais. Em suma, cada vez que um elemento referenciar outro, o referenciado o considera como elemento a ser notificado quando houver mudanças em seu estado. Por exemplo, quando uma *Premise* faz menção a um dado *Attribute*, este considera tal *Premise* como elemento a ser notificado. Isto permite emergir o mecanismo ou inferência por notificações sem esforços do desenvolvedor do *software* para tal (SIMÃO e STADZISZ, 2008, 2009) (SIMÃO, *et.al.*, 2012a).

Os conceitos descritos nos parágrafos precedentes do PON estão materializados na forma de um *Framework* sobre uma linguagem de programação

imperativa. Esta materialização foi elaborada pelo trabalho de BANASZEWSKI (2009). Particularmente, o PON está concebido sobre a linguagem de programação C++, uma linguagem clássica do POO. Com isto, o *Framework* PON, é considerado uma *API* de desenvolvimento de aplicações PON, a qual fornece meios de compor os *FBEs* e suas respectivas *Rules*.

1.4 WIZARD CON

Uma primeira versão elaborada de ferramenta para facilitar a criação de *Rules* do PON se deu embrionariamente no âmbito do uso do PON para controle, atualmente chamado de Controle Orientado a Notificação (CON). Esta ferramenta em questão foi denominada *Wizard* CON e foi concebida no trabalho de LUCCA (2008) e posteriormente aprimorada no trabalho de WITT (2010).

A principal responsabilidade da ferramenta *Wizard* CON é a composição de instâncias de controle que permite escrever regras e gerar automaticamente, a partir delas, os agentes *Rules* no âmbito do CON. Isto facilita o processo de criação de instâncias de controle que antes se dava tecnicamente por meio da linguagem de programação C++, ainda que apoiado em um *Framework* do CON (LUCCA, 2008). O *Wizard* CON utiliza ambiente gráfico (*Windows Forms*) permitindo ao usuário compor regras no clássico formato *se então*, em ambiente amigável (cf. mostra a Figura 3).

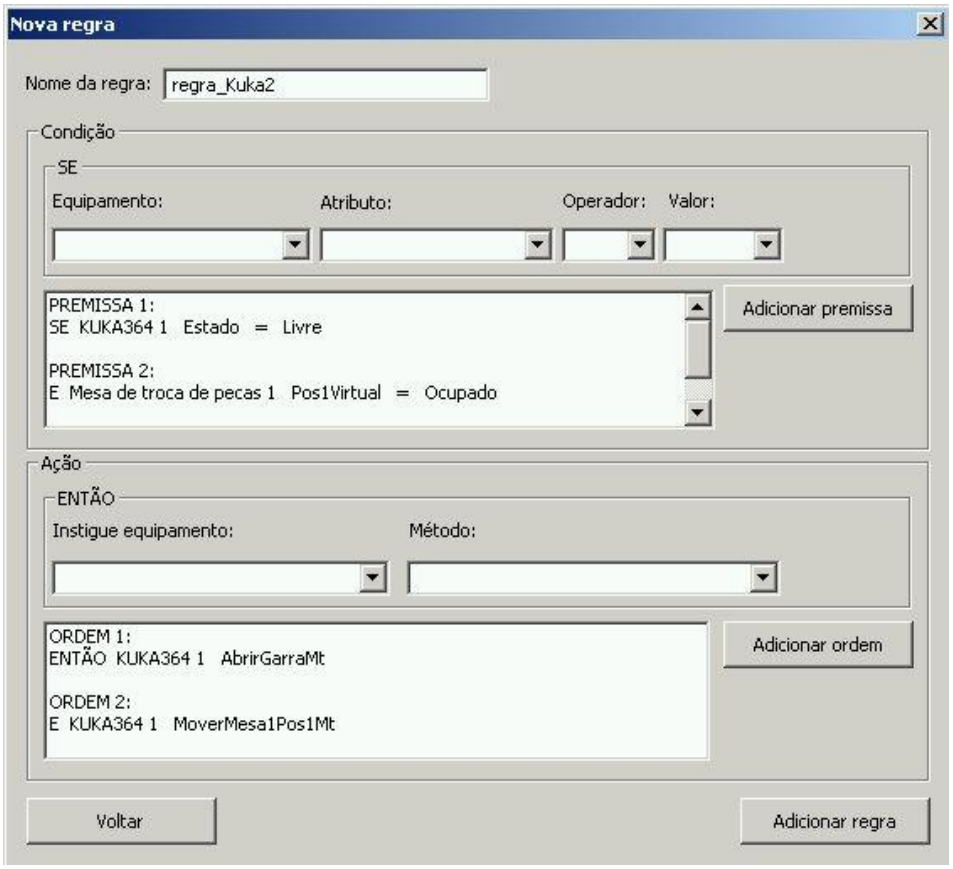


Figura 3 – *Wizard* e seu ambiente gráfico (LUCCA, 2008).

Esta ferramenta de fato auxilia na criação de regras de controle, utilizada neste caso em conjunto ao *Framework* CON sobre o simulador ANALYTICE II (SIMÃO, 2005). Assim de maneira conjunta e desacoplada, as ferramentas *Wizard* CON e ANALYTICE II com *Framework* CON, trabalham para agilizar a criação e execução automatizada de regras de controle. Esta integração é realizada por meio da importação e exportação de arquivos do tipo *e**X**tensible **M**arkup **L**anguage (XML)* entre a ferramenta *Wizard* CON e o simulador ANALITYCE II com *Framework* CON.

1.5 APRESENTAÇÃO DO PROBLEMA

Atualmente os conceitos do PON estão materializados sobre o paradigma dominante, especificamente sobre o Paradigma Orientado a Objetos (POO), através de um *Framework* desenvolvido na linguagem de programação C++. Este *Framework* introduz uma camada intermediária entre uma aplicação PON e sua execução propriamente dita, conforme observado na Figura 4. Ademais, utiliza-se de estruturas de alto nível como classes que implementam contêineres da *Standard*

Template Library (*STL*). Isto implica em camadas de código e estruturas de dados computacionalmente custosas para a execução de aplicações PON.

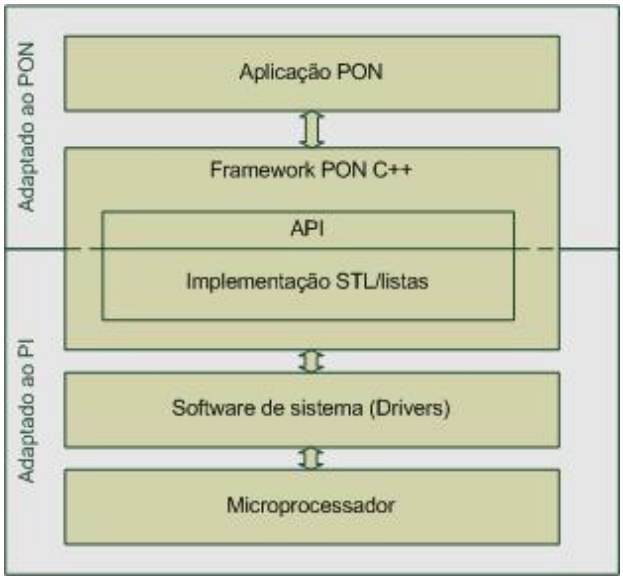


Figura 4 – Camadas da arquitetura do *Framework* PON

Isto dito, algumas deficiências encontradas na arquitetura de sua implementação (*Framework* PON) podem ser corrigidas e então aprimoradas, pelo repensar desta materialização. Particularmente, além de refatorações e otimizações de código, a estrutura de dados que comporta a cadeia de notificação, responsável pelo processo de avaliação das expressões causais, pode ser aperfeiçoada. Estas alterações visam principalmente melhorar o desempenho das aplicações que executam sobre o referido *Framework* do PON.

Ainda, sobre o viés de composição de aplicações PON em alto nível, a ferramenta *Wizard* CON permanece em estágio embrionário. Desta forma, a mesma carece de melhorias, evoluções e generalização para o PON (além de apenas aplicações ditas de controle). Essas evoluções visam principalmente agregar facilidades de composição de aplicações do PON em alto nível, através de formulários de entrada, bem como a geração de código PON em C++ inerente as novas evoluções propostas e implementadas para o novo *Framework* PON.

1.6 JUSTIFICATIVA DO PROJETO

O foco da implementação do *Framework* PON, em sua versão original (BANAZESWKI, 2009), foi a de materializar os conceitos relativos ao PON e demonstrar suas funcionalidades. Assim, a versão do PON materializado sobre seu *Framework* não foi otimizado. Deste modo, o próprio *Framework* induz a uma sobrecarga (*overhead*) significativa de processamento, principalmente em função das diversas iterações em listas utilizadas pelo processo de notificação de elementos que estão associados entre si. Mais precisamente, uma estrutura de dados de alto nível via *STL* é utilizada para comportar os objetos notificantes do PON.

Tal camada implica em estruturas de dados computacionalmente custosa uma vez que objetos de classes da *STL* são formados por uma estrutura de coleção genérica. Portanto, a implementação das relações entre os objetos colecionados se dá através de um protocolo de chamadas de métodos (acesso à sua interface) que não estão necessariamente otimizados para uso do mecanismo do PON.

A otimização da estrutura que comporta as relações entre os objetos participantes do mecanismo de notificação é fundamental para o desempenho de aplicações que executam sobre o *Framework* PON. Deste modo, além da própria otimização desta estrutura, este trabalho visa otimizar o *Framework* PON como um todo, com o viés de eficiência de processamento e/ou recursos (memória, *cache etc*). Assim, esforços como refatoração de código, estratégias e técnicas de otimização e remodelamento da arquitetura vão de encontro à obtenção de melhor desempenho de aplicações que executam sobre o *Framework* PON.

Ainda, para que seja possível a composição de aplicações PON em alto nível é necessária a construção da ferramenta *Wizard* PON, inspirando-se no *Wizard* CON existente. O foco está principalmente em fornecer à composição e geração de código PON em liguagem C++ inerente a versão atualizada de seu respectivo *Framework.* Neste âmbito, a ferramenta *Wizard* em questão será denominada de *Wizard* PON, referenciando-se especificamente para a criação e elaboração de aplicações PON ao invés de apenas instâncias de controle como era anteriormente responsável a precedessora ferramenta *Wizard*.

Isto dito, este trabalho de pesquisa visa unificar a utilização da ferramenta *Wizard* PON com seu referido *Framework*. Neste âmbito, a nova versão elaborada do *Framework* PON visa desempenho das aplicações que executam sobre ele, principalmente através da implementação de novas estruturas de dados que comportem o processo de notificação, bem como otimizações e refatorações de código. Em relação à ferramenta *Wizard* PON, a seu turno, o desenvolvedor utilizará tal ferramenta na fase de construção da aplicação PON propriamente dita, a qual realiza a geração de código C++ da aplicação PON conforme a nova versão (projetada e implementada) do *Framework* do PON.

Outrossim, antes do uso do *Wizard* PON, o desenvolvedor criaria as aplicações PON a partir de requisitos e particularmente a partir de modelos de artefatos gerados pelo chamado processo de Desenvolvimento Orientado a Notificações (DON) (WIECHETECK, 2011), ainda que esta temática em particular não seja o foco efetivo desta dissertação.

1.7 OBJETIVOS

Os objetivos deste presente trabalho são apresentados nesta seção, começando-se com o objetivo geral e subsequentemente derivando-se para os objetivos específicos.

1.7.1 Objetivo geral

O objetivo geral é efetuar a reengenharia e validação de melhorias da atual materialização do Paradigma Orientado a Notificações (PON) na forma de uma nova materialização do PON focada em otimização algoritmo-estrutural do arquétipo/*Framework* e no desenvolvimento de uma interface-amigável (*Wizard*) de instanciação de *software* em PON.

1.7.2 Objetivos específicos

Dentre os principais objetivos específicos destacam-se:

- Reengenharia do modelo de análise e projeto (*design*) do arquétipo ou *Framework* do PON constituindo um novo *Framework* do PON.
- Otimização algorítmo-estrutural do novo *Framework* do PON.
- Concepção de uma Interface Amigável ou *Wizard PON* (a partir de protótipo limitado anterior) para facilitar a implementação em PON.
- Composição de nova materialização do PON pela junção inerente do *Framework* e seu respectivo *Wizard.*
- Implementação de casos de estudo com a finalidade de determinar o Δ, em termos de desempenho e facilidade de programação, da nova materialização do PON (*i.e.* novo *Framework* e *Wizard*) em relação à antiga materialização dele (*i.e.* prévio *Framework* e protótipo de *Wizard*).

1.8 ESTRUTURA DO TRABALHO

O capítulo 1 introduziu o problema atual e contextualizou a justificativa para elaboração deste trabalho e, por fim, definiu o objetivo geral e os objetivos específicos. O capítulo 2 apresenta a fundamentação teórica para o restante do trabalho, no que diz respeito ao PON, processo de desenvolvimento DON, ferramente *Wizard* CON, ferramentas geradoras de código fonte e técnicas de refatoração. O capítulo 3 detalha o método adotado para as devidas implementações do novo *Framework* PON. Ademais, este capítulo descreve os casos de estudo bem como o desempenho obtido em relação ao *Framework* PON original e a sua nova versão. O capítulo 4 descreve as evoluções realizadas sobre a ferramenta *Wizard* PON e seu respectivo caso de estudo. Por fim, o capítulo 5 finaliza com as conclusões e trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Primeiramente, a seção 2.1 introduz as dificuldades encontradas na computação as quais remontam as linguagens de programação usuais e seus paradigmas. A seção 2.2, mais especificamente, descreve características inapropriadas relacionadas a programação imperativa e declarativa. A seção 2.3 está voltada para uma revisão aprofundada do PON. A seção 2.4, detalha a materialização do PON através de um *Framework* desenvolvido na linguagem de programação C++. A seção 2.5, por sua vez, descreve o processo de desenvolvimento do PON denominado Desenvolvimento Orientado a Notificações (DON). A seu turno, a subseção 2.6 aborda os detalhamentos de implementação da ferramenta denominada *Wizard* CON. A seção 2.7, descreve sobre as ferramentas geradoras de código. Por fim, a seção 2.8 relata o tema referente à refatoração de código Estes temas supracitados dão o suporte necessário para as devidas evoluções referentes a materialização do PON e a composição de aplicações PON no âmbito da ferramenta *Wizard* PON.

2.1 REFLEXÃO SOBRE PROGRAMAÇÃO

O poder de processamento computacional tem crescido a cada ano e a tendência é que a evolução da tecnologia contribua para a criação de tecnologias de processamento ainda mais eficientes (SIMÃO, *et. al*., 2012a) (KEYES, 2006). Mesmo este cenário sendo positivo em termos de evolução da tecnologia, em geral, ele não motiva os profissionais da tecnologia da informação a otimizarem o uso de recursos de processamento quando eles desenvolvem *software* (SIMÃO, *et. al*., 2012a) (RAYMOND, 2003).

Este comportamento tem sido tolerado na maioria dos desenvolvimentos de *software*, onde não há necessidade de tratamento intensivo ou restrições de processamento. No entanto, tal comportamento não é aceitável para as certas classes de *software*, como *software* para sistemas embarcados. Tais sistemas normalmente empregam menor poder de processamento devido a fatores como

restrições no consumo de energia e preço do sistema para um determinado mercado (WOLF, 2007) (SIMÃO, STADZISZ, 2008) (SIMÃO, *et. al*., 2012a).

Além disso, o uso de processamento computacional indevido em *software* também pode causar o uso excessivo de um dado processador, implicando em atrasos de execução (SIMÃO, *et. al*., 2012a) (SIMÃO, STADZISZ, 2008) (OLIVEIRA, STEWART, 2006). Ainda, em *softwares* complexos, isso pode até esgotar a capacidade do processador, exigindo um processador mais rápido ou até mesmo algum tipo de distribuição (*e.g.*, *dual-core*) (SIMÃO, *et. al*., 2012a) (SIMÃO, STADZISZ, 2008) (HUGHES, HUGHES, 2003). Na verdade, uma programação orientada para otimização poderia minimizar tais inconvenientes e custos relacionados (SIMÃO, *et. al*., 2012a) (SIMÃO, STADZISZ, 2008).

Portanto, ferramentas de engenharia de *software* adequadas para o desenvolvimento, incluindo linguagens de programação e seus ambientes, deveriam facilitar o desenvolvimento de código correto e otimizado (SIMÃO, *et. al*., 2012a) (BANASZEWSKI, STADZISZ, TACLA, SIMÃO, 2007) (BANASZEWSKI, 2009) (HERLIHY, SHAVIT, 2008) (HAREL, *et al.,* 1990). Caso contrário, custos relacionados à engenharia para produzir código otimizado poderão superar os custos de melhoria da capacidade de processamento (SIMÃO, *et. al*., 2012a) (SIMÃO, STADZISZ, 2009) (BANASZEWSKI, STADZISZ, TACLA, SIMÃO, 2007) (BANASZEWSKI, 2009) (HERLIHY, SHAVIT, 2008).

Ainda, tais ferramentas deveriam também agilizar o processo de desenvolvimento de código distribuível uma vez que distribuição pode ser fidedignamente necessária em dados contextos (SIMÃO, *et. al*., 2012a) (SEVILLA, GARCIA, GÓMEZ, 2008) (JOHNSTON, HANNA, MILLAR, 2004) (COULOURIS, DOLIMORE, KINDBERG, 2001) (GRUVER, 2007). No entanto, a distribuição em si é um problema uma vez que, sob diferentes perspectivas, pode exigir estratégias como balanceamento de carga para evitar excesso de comunicação ou distribuição de código com granularidade fina para aproveitar recursos de processamento disponíveis (SIMÃO, *et. al*., 2012a) (SIMÃO e STADZISZ, 2009) (SEVILLA, GARCIA, GÓMEZ, 2008) (JOHNSTON, HANNA, MILLAR, 2004) (GAUDIOT, SOHN, 1990).

Neste contexto, destaca-se o fato de que linguagens de programação usuais (como Pascal, C/C++ e Java) não apresentam facilidades reais para desenvolver código otimizado e ‘desacoplado’ (ou minimamente acoplado para ser preciso). Isto dificulta o uso de processamento disponível por não otimizar recursos. Isto também

dificulta a distribuição de código no caso de sistema com paralelismo, sendo, particularmente difícil a distribuição de código com granularidade fina (SIMÃO, *et. al*., 2012a) (RAYMOND, 2003) (SIMÃO e STADZISZ, 2009) (GAUDIOT, SOHN, 1990) (BANERJEE, *et al.,* 1995). Isso acontece nas linguagens de programação usuais devido à estrutura e natureza de execução imposta pelos seus respectivos paradigmas (SIMÃO, *et. al*., 2012a) (HUGHES, HUGHES, 2003) (BANASZEWSKI, *et. al.*, 2007) (BANASZEWSKI, 2009).

2.2 PROGRAMAÇÃO IMPERATIVA E DECLARATIVA

As linguagens de programação usuais são baseadas no Paradigma Imperativo (PI) que abrangem sub-paradigmas tais como o Procedimental e o Orientado a Objetos (BANASZEWSKI, 2009) (ROY, HARIDI, 2004) (KAISLER, 2005). Outrossim, o último é normalmente considerado melhor do que o primeiro, devido ao seu supostamente melhor mecanismo de abstração. De qualquer forma, ambos apresentam problemas devido à sua natureza imperativa (BANASZEWSKI, 2009) (ROY, HARIDI, 2004) (GABBRIELLI, MARTINI, 2010) (SIMÃO, *et. al*., 2012a).

Essencialmente, o paradigma imperativo impõe pesquisas orientadas a laços de repetições sobre os elementos passivos, relacionando os dados (*i.e*, variáveis, vetores e árvores) a expressões causais (*i.e*, se-então ou declarações similares). Tais relacionamentos normalmente impactam negativamente nas aplicações devido a sua estrutura monolítica, prolixa e acoplada, o que gera a execução de código não otimizado e interdependente (SIMÃO, *et. al*., 2012a) (SIMÃO, STADZISZ, 2007) (BANASZEWSKI, STADZISZ, TACLA, SIMÃO, 2007) (GABBRIELLI, MARTINI, 2010) (BROOKSHEAR, 2006).

O Paradigma Declarativo é uma alternativa ao Paradigma Imperativo. Essencialmente, ele permite um nível maior de abstração e maior facilidade de programação (SIMÃO, *et. al*., 2012a) (KAISLER, 2005) (GABBRIELLI, MARTINI, 2010). Além disso, a fim de otimizar o processamento, algumas soluções declarativas podem evitar as redundâncias de execução, tais como os Sistemas Baseados em Regras (SBRs) com base em algoritmos de inferência otimizados como *HAL*L ou *RETE* (CHENG, CHEN, 2000) (KANG, CHENG, 2004) (FORGY, 1982) (LEE, CHENG, 2002). No entanto, os programas construídos usando

linguagens usuais do Paradigma Declarativo (*e.g.*, LISP, PROLOG, e RBS) ou mesmo usando uma solução otimizada (*e. g.*, SBR basedo em *RETE*) também apresentam desvantagens (SIMÃO, *et. al*., 2012a) (SIMÃO e STADZISZ, 2007) (BANASZEWSKI, *et.al.*, 2007).

Soluções do PD são compostas por estruturas de dados de alto nível, as quais são normalmente caras em termos de processamento. Isso, de fato, agrega em custos de processamento consideráveis, impactando diretamente no desempenho das aplicações. Assim, mesmo com código redundante, soluções do PI são normalmente melhores em desempenho do que as soluções do PD (BANASZEWSKI, 2009) (SCOTT, 2000).

Além disso, similarmente a programação do PI, a programação em PD também gera código acoplado devido ao processo similar de inferência baseada em pesquisa sobre entidades passivas (SIMÃO e STADZISZ, 2009) (SIMÃO e STADZISZ, 2007) (GABBRIELLI e MARTINI, 2010). Ainda, outras abordagens entre o PI e o PD, tais como a programação dirigida por eventos ou a programação funcional, não resolvem tais problemas. Em alguns casos, essas reduzem certas redundâncias, em outros, apenas atenuam ou fatoram tais problemas (BROOKSHEAR, 2006) (SCOTT, 2000) (SIMÃO, *et. al*., 2012a).

2.2.1 Questões sobre a Programação Imperativa

As principais desvantagens da Programação Imperativa estão voltadas para a redundância de código e acoplamento (SIMÃO, STADZISZ, 2009). A primeira afeta principalmente o tempo de processamento e a segunda o processo de reaproveitamento de módulos e desacoplamento-distribuição, conforme detalhado nas próximas seções.

2.2.1.1 Redundância em Programação Imperativa

Na Programação Imperativa, incluindo Programação Orientada a Objetos, a presença de código redundante e interdependente é resultado da maneira com que as expressões causais são organizadas e consequentemente avaliadas. Isto é exemplificado no pseudocódigo na Figura 5 que representa um código usual e

elaborado sem grandes esforços técnicos e intelectuais. Isto significa que o pseudocódigo foi elaborado de uma forma não complicada, como idealmente deveriam ser concebidas as aplicações (SIMÃO, *et. al*., 2012a) (SIMÃO, STADZISZ, 2007) (BANASZEWSKI, 2009).

```
...
while (true) do
  if((object_1.attribute_1 = 1) and
     (object_2.attribute_1 = 1) and
     (object_3.attribute_1 = 1))
  then
      object_1.method_1();
      object_2.method_1();
      object_3.method_1();
  end_if
  ...
  if((object_1.attribute_1 = 1) and
     (object_2.attribute_n = n) and
     (object_3.attribute_n = n))
  then
      object_1.method_n();
      object_2.method_n();
      object_3.method_n();
  end_if
end_while
...
```

Figura 5 – Código Imperativo

Nesse exemplo, é observado que o laço de repetição força a avaliação (ou inferência) de todas as condições de maneira sequencial. No entanto, a maioria das avaliações é desnecessária, uma vez que somente alguns atributos apresentam alterações em seus estados em cada iteração. Isso até pode ser considerado não importante nesse exemplo simples e pedagógico, sobretudo se o número (n) de expressões causais for pequeno. Entretanto, se for considerado um sistema complexo, integrando muitas partes como aquela, pode-se ter uma grande diferença de desempenho. Ademais, esse tipo de código apresenta os problemas chamados de redundância temporal e estrutural (SIMÃO, *et. al*., 2012a) (PAN *et al.*, 1998) (SIMÃO, STADZISZ, 2008, 2009) (FORGY, 1982).

As redundâncias de código podem resultar, por exemplo, na necessidade de um processador mais poderoso do que é realmente necessário (SIMÃO, *et. al*., 2012a) (SIMÃO, STADZISZ, 2008) (HUGHES, HUGHES, 2003). Além disso, eles podem resultar na necessidade de uma distribuição de código para os processadores, implicando em outros problemas, como a divisão modular das aplicações e as devidas sincronizações. Esses problemas, mesmo se solucionáveis,

são questões adicionais ao desenvolvimento de *software*, cuja complexidade aumenta tanto quanto é exigida a demanda por distribuição de “código” (ou de processamento para ser preciso) com granularidade fina, particularmente em termos da realização de cálculo lógico causal (*i.e*, "se-então") (SIMÃO, STADZISZ, 2009) (HUGHES, HUGHES, 2003) (BANASZEWSKI, *et.al.*, 2007) (SIMÃO, *et. al.*, 2012a).

2.2.1.2 Acoplamento em Programação Imperativa

Além das avaliações usuais repetitivas e desnecessárias no código imperativo, os elementos avaliados em expressões causais são passivos, embora eles sejam essenciais neste processo. Por exemplo, uma dada declaração *if-then* ou se-então (ou seja, uma expressão causal) e as variáveis (ou seja, elementos avaliados) não tomam parte na decisão com relação ao momento em que devem ser avaliados (SIMÃO, STADZISZ, 2009) (SIMÃO, *et. al.*, 2012a).

A passividade das expressões causais e seus respectivos elementos definem a forma com que esses são avaliados durante a execução de um programa. As avaliações de tais expressões causais e seus respectivos elementos são realizadas sequencialmente pela linha de execução de um programa (ou, pelo menos, em cada *thread* de um programa), comumente guiadas por meio de um conjunto de laços de repetição. Como essas expressões causais e seus elementos não são ativos no tocante a sua própria execução (ou seja, eles são passivos), a sua interdependência não é explícita em cada execução do programa (SIMÃO, *et. al*., 2012a) (SIMÃO, STADZISZ, 2009).

Sendo assim, as expressões causais e seus respectivos elementos avaliados, dependem dos resultados das avaliações ou estados de outros elementos. Isso significa que elas são acopladas de alguma maneira e que deveriam ser dispostas conjuntamente, pelo menos no contexto de cada módulo. Esse acoplamento aumenta a complexidade do código, o que dificulta, por exemplo, uma eventual distribuição de uma simples parte do código. Isso faz com que cada módulo, ou até mesmo o programa como um todo, seja entendido como uma entidade computacional monolítica (SIMÃO, *et. al*., 2012a) (SIMÃO, STADZISZ, 2009).

2.2.1.3 Dificuldade de Distribuição em Programação Imperativa

Quando a distribuição (*e.g.* distribuição de processo, processador e/ou distribuição por *cluster*) é pretendida, uma análise de código poderia identificar um conjunto de código menos dependente, o que facilitaria sua posterior divisão e distribuição. No entanto, esta atividade é normalmente complexa devido ao código acoplado e a complexidade resultante da programação imperativa (SIMÃO, *et. al*., 2012a) (WACHTER, MASSART, MEUTER, 2004) (BANERJEE, *et al.,* 1995).

Neste sentido, um *software* bem concebido, composto por módulos minimamente acoplados, usando de avançadas (e complicadas) técnicas de engenharia de *software*, como aspectos (SEVILLA, GARCIA, GÓMEZ, 2008) e projeto axiomático (PIMENTEL, STADZISZ, 2006), poderia ajudar no processo de distribuição (SIMÃO, *et. al*., 2012a). Ainda, *middlewares* como CORBA e RMI poderiam ser úteis em termos de infraestrutura para alguns tipos de distribuição, caso exista um desacoplamento suficiente entre os módulos de *software* (SIMÃO, *et. al*., 2012) (SEVILLA, GARCIA, GÓMEZ, 2008) (AHMED, 1998) (REILLY, REILLY, 2002).

Apesar desses avanços, a distribuição de cada elemento de código ou até mesmo de cada módulo de código ainda é uma atividade complexa, exigindo esforços de pesquisa (SIMÃO, *et. al*., 2012a) (WACHTER, MASSART, MEUTER, 2004) (SEVILLA, GARCIA, GÓMEZ, 2008) (JOHNSTON, HANNA, MILLAR, 2004) (GAUDIOT, SOHN, 1990) (TILEVICH, SMARAGDAKIS, 2002). Neste âmbito, seriam ainda necessários esforços adicionais para alcançar a facilidade de distribuição (*e.g.*, distribuição automática, rápida e em tempo real), bem como a correta distribuição (*e.g.* distribuição balanceada e minimamente interdependente) (SIMÃO, STADZISZ, 2009).

A dificuldade de distribuição é um problema, uma vez que existem contextos em que a distribuição é realmente necessária (SIMÃO, *et. al*., 2012a) (HUGHES, HUGHES, 2003) (COULOURIS, DOLIMORE, KINDBERG, 2001) (GRUVER, 2007). Por exemplo, um dado programa otimizado e que, ainda assim, excede a capacidade de um processador disponível, poderia ter seu processamento dividido em um conjunto de processadores (OLIVEIRA, STEWART, 2006). Estas características podem ser encontradas em aplicações voltadas a planta nuclear (DÍAZ,*et. al.*, 2007), manufatura inteligente (SIMÃO, 2005) (SIMÃO, TACLA,

STADZISZ, 2009) (DEEN, 2003) (TIANFIELD, 2007) e controle cooperativo (KUMAR, LEONARD, MORSE, 2005) (SIMÃO, 2005).

Além disso, existem outros aplicativos que são inerentemente distribuídos e precisam de uma distribuição flexível, tais como os de computação ubíqua. Exemplos mais precisos são das redes de sensores e alguns sistemas de controle de produção inteligente (LOKE, 2006) (TIANFIELD, 2007). Ainda, em geral, a facilidade e a correta distribuição seriam esperadas, uma vez que existe uma crescente redução de preços de processadores, bem como avanços nos modelos de comunicação em rede (BANASZEWSKI, 2009) (TANENBAUM, STEEN, 2002) (SIMÃO, *et. al.*, 2012a).

2.2.1.4 Dificuldade de Desenvolvimento em Programação Imperativa

Além das questões de otimização e problemas de distribuição, o desenvolvimento de programas com a Programação Imperativa (PI), pode ser visto como difícil devido à sintaxe complicada e uma diversidade de conceitos a serem aprendidos, tais como: ponteiros, variáveis de controle e laços aninhados (GIARRATANO, RILEY, 1993).

O processo de desenvolvimento seria propenso a erros, uma vez que quase todo o código é realizado de forma manual, utilizando tais conceitos. Neste contexto, o algoritmo imperativo exemplificado (Figura 5) poderia certamente ser otimizado, no entanto sem facilidades significativas nesta atividade, ainda mais quando considerado a atividade de desacoplamento, uma vez que tais expressões causais apresentam forte acoplamento entre si.

Seria necessário investigar soluções melhores do que aquelas fornecidas pelo Paradigma Imperativo. A solução para resolver alguns de seus problemas pode ser o uso de linguagens de programação de outros paradigmas, como a Programação Declarativa que automatiza o processo de avaliação de expressões causais e seus elementos (ROY, HARIDI, 2004) (RUSSEL, NORVIG, 2003) (SIMÃO, *et. al.*, 2012). Ainda, essa abordagem proporciona abstrações que minimizam a realização de algumas tarefas (*i.e.* programa-se “o que fazer” ao invés de “como fazer”).

2.2.2 Questões sobre a Programação Declarativa

Um exemplo bem conhecido da Programação Declarativa e sua natureza é o Sistema Baseado em Regras (SBRs) (SIMÃO, *et. al*., 2012a) (SIMÃO, STADZISZ, 2009) (GIARRATANO, RILEY, 1993). Os SBRs provêm uma programação em alto nível baseado na composição de regras causais, o que minimiza o contato dos desenvolvedores com particularidades algorítmicas (SIMÃO, *et. al*., 2012) (GIARRATANO, RILEY, 1993). Os SBRs são compostos por três entidades modulares genéricas (Base de Fatos, Base de Regras e Motor de Inferência), as quais possuem responsabilidades distintas. Na verdade, esta forma é usual em linguagens declarativas (*i.e*. PROLOG e CLIPS) (SIMÃO, *et. al*., 2012a) (RUSSEL, NORVIG, 2003).

Na Programação Declarativa, os estados das variáveis são tratados em uma Base de Fatos e o conhecimento causal em uma Base Causal (ou Base de Regra na programação SBR), as quais são automaticamente comparadas por meio de um Motor de Inferência (KANG, CHENG, 2004) (GIARRATANO, RILEY, 1993). Além disso, alguns algoritmos de inferência evitam a maioria das redundâncias temporais e estruturais (BANASZEWSKI, 2009). Exemplos de tais algoritmos de inferência são *RETE* (FORGY, 1982) (CHENG, CHEN, 2000) (LEE, CHENG, 2002) (KANG, CHENG, 2004), *TREAT* (MIRANKER, 1987) (MIRANKER, LOFASO, 1991), *LEAPS* (MIRANKER, *et. al.*, 1992), e *HAL* (LEE, CHENG, 2002). No entanto, as estruturas de dados usadas para resolver estes problemas implicam em consumo extra de processamento (FORGY, 1982) (SIMÃO, *et. al*., 2012a).

O uso da Programação Declarativa só compensa quando o *software* em desenvolvimento apresenta numerosas redundâncias e alguma variação de dados. Além disso, em geral, um motor de inferência relacionado a uma determinada linguagem declarativa limita a criatividade do desenvolvedor, o que dificulta algumas otimizações algorítmicas e obscurece acesso ao *hardware*, o que pode ser inadequado em determinados contextos (BANASZEWSKI, 2009) (BROOKSHEAR, 2006) (SCOTT, 2000) (WATT, 2004) (SIMÃO, *et. al*., 2012a).

A solução para estes problemas poderia ser a simbiose entre a programação Declarativa e Imperativa (ROY, HARIDI, 2004) (WATT, 2004). Na verdade, tal abordagem foi proposta em soluções como CLIPS++ (GIARRATANO, RILEY, 1993)

e ILOG *Rules* (ALBERT, 1994). No entanto, eles não são populares devido a fatores como a mistura da sintaxe, mistura de paradigmas e razões técnicas culturais (BANASZEWSKI, 2009). De qualquer forma, mesmo o Paradigma Declarativo sendo uma solução relevante, ele não resolve alguns problemas.

De fato, além da sobrecarga de processamento, a Programação Declarativa também apresenta acoplamento em seu código. Cada programa declarativo tem também uma execução ou política de inferência cuja essência é uma entidade monolítica (*i.e*, uma máquina de inferência) responsável por analizar cada entidade passiva (base de fatos) e sua base de expressões causais (base de regras). Assim, a inferência baseada em técnica de pesquisa (*i.e. matching*) implica em uma forte dependência entre os fatos e suas regras causais (SIMÃO, STADZISZ, 2009) (SIMÃO, *et. al.*, 2012a).

2.2.3 Outras Abordagens de Programação

Melhorias no contexto do Paradigma Imperativo e Declarativo têm sido aplicadas para reduzir os efeitos de códigos baseados em pesquisas redundantes, tais como a Programação Orientada a Eventos e a Programação Funcional (BANASZEWSKI, 2009) (RUSSEL, NORVIG, 2003) (FAISON, 2006). A Programação Orientada a Eventos e a Programação Funcional, têm sido aplicada para diferentes tipos de *software*, como controle discreto, interfaces gráficas e sistemas multiagentes (BANASZEWSKI, 2009) (RUSSEL, NORVIG, 2003) (FAISON, 2006) (SIMÃO, *et. al.*, 2012a).

Essencialmente, na Programação Orientada a Eventos, cada evento (um botão pressionado, uma interrupção de *hardware* ou uma mensagem recebida) desencadeia uma dada execução (procedimento, processo ou método), geralmente em um tipo determinado de módulo (bloco, objeto ou até mesmo agente), em vez de análises repetidas das condições para a sua execução. O mesmo princípio se aplica à chamada Programação Funcional, cuja diferença seria as chamadas de um conjunto de funções através de outra função, em substituição a eventos. Ainda, função neste contexto significaria procedimento, método ou alguma unidade similar. Outrossim, programação funcional e orientada a eventos usadas em conjunto seria algo usual (SIMÃO, *et. al*., 2012a).

No entanto, o algoritmo em cada processo, método ou função é constituído usando a Programação Declarativa ou Imperativa. Isto implica nas deficiências encontradas nestes estilos de programação, como redundância de código, acoplamento etc. De fato, se cada módulo possuir uma considerável quantidade de código causal, eles podem ser um problema em conjunto, em termos de mau uso de processamento e dificuldade de distribuição correta. Isto pode exigir esforço de projeto em especial para conseguir otimização e desacoplamento de módulos (SIMÃO, *et. al.*, 2012a).

Outrossim, uma abordagem alternativa de programação é a chamada Programação Orientada a Fluxo de Dados (JOHNSTON, HANNA, MILLAR, 2004), que supostamente deveria permitir a execução do programa orientada por dados, em vez de uma linha de execução com base na pesquisa sobre os dados. Portanto, isso permitiria o desacoplamento e distribuição (JOHNSTON, HANNA, MILLAR, 2004). A distribuição da Programação Dirigida por Fluxo de Dados é obtida no processamento aritmético, porém não é realmente alcançada em processamento do tipo lógico-causal (JOHNSTON, HANNA, MILLAR, 2004) (GAUDIOT, SOHN, 1990). Este processamento seria realizado por intermédio de avançados motores de inferência, tais como *RETE* (GAUDIOT, SOHN, 1990) (TUTTLE, EICK, 1992) (SIMÃO, *et. al.*, 2012a).

O fato é que os motores atuais de inferência tentam alcançar uma abordagem orientada ao fluxo de dados. No entanto, o processo de inferência ainda se baseia em pesquisas, mesmo que se utilizando de assaz otimizadas “árvores” ou grafos de dados. Assim, os problemas relatados persistem (SIMÃO, *et. al.*, 2012a).

2.2.4 Melhorias na Programação

Em suma, conforme explanado, a Programação Imperativa e Declarativa não alcançam de maneira fácil e conjunta alguns requisitos, como a facilidade de obtenção de código otimizado, facilidade e flexibilidade na composição de programas, divisão de código/módulos e distribuição balanceada (SIMÃO *et al.*, 2012a).

Isto é um problema, principalmente quando se considera a demanda de mercado por *software,* onde facilidade de desenvolvimento, código otimizado e

processos de distribuição são requisitos atuais (SOMMERVILLE, 2004; PAES e HIRATA, 2008; WATSON *et al.*, 2009). Na verdade, esta demanda por *software* estimula novas pesquisas e soluções para tornar mais simples a tarefa de construir *software* com tais requisitos (SIMÃO *et al.*, 2012a).

Neste contexto, um novo paradigma de programação chamado Paradigma Orientado Notificação (PON) foi proposto para resolver alguns dos problemas apontados. Oportunamente, o PON foi introduzido na seção 1.3 e será detalhado nas próximas seções.

2.3 PARADIGMA ORIENTADO A NOTIFICAÇÕES (PON).

Em linhas gerais, o Paradigma Orientado Notificação (PON) encontra inspirações no PI, tais como flexibilidades algorítmicas e a abstração em forma de classes/objetos do POO e a reatividade da programação dirigida a eventos. O PON também aproveita conceitos próprios do PD, como facilidade de programação em alto nível e a representação do conhecimento em regras dos SBR. Assim, o PON provê a possibilidade de uso (de parte) de estilos de programação em seu modelo, ainda que os evolua e mesmo os revolucione (de certa maneira) no tocante ao processo de inferência ou cálculo lógico-causal (SIMÃO e STADZISZ, 2008, 2009) (BANASZEWSKI, 2009). A relação do PON com os paradigmas que inspiraram a sua essência está ilustrada na Figura 6

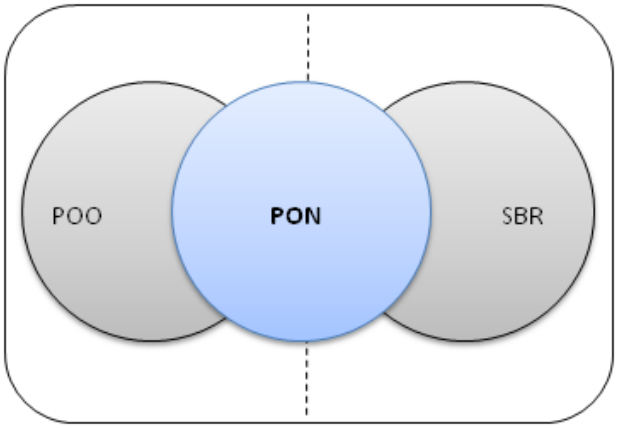


Figura 6 – Relação entre PON e os demais paradigmas (WIECHETECK, 2011)

Neste âmbito, o PON apresentaria resposta aos problemas desses paradigmas, como repetição de expressões lógicas e reavaliações desnecessárias delas (*i.e.* redundâncias estruturais e temporais) e, particularmente, o acoplamento forte de entidades no tocante às avaliações ou cálculo lógico-causal. Justamente, o PON apresenta outra maneira de realizar tais avaliações ou inferências por meio de entidades computacionais de pequeno porte, ativas e desacopladas que colaboram por meio de notificações pontuais e são criadas a partir do 'conhecimento' de regras (SIMÃO, *et. al.*, 2012a).

Esta seção como um todo detalha o Paradigma Orientado a Notificações (PON). Mais precisamente, a seção 2.3.1 apresenta o mecanismo de notificação do PON. Por sua vez, a seção 2.3.2, aborda o mecanismo de resolução de conflitos e garantias de determinismo em aplicações do PON. A seção 2.3.3 reflete e contextualiza as propriedades inerentes ao PON. Ainda, a seção 2.3.4 define as características de utilização e compreensão do PON. A seção 2.3.5 detalha a função assintótica do PON em relação ao seu processo de resolução do cálculo lógico causal. Por fim, as seções 2.4 e subsequentes, descrevem sua materialização, que foi realizada por meio de um *Framework* desenvolvido na linguagem de programação C++

2.3.1 Mecanismo de Notificação do PON

O Paradigma Orientado a Notificações (PON) introduz um novo conceito para a concepção, construção e execução de aplicações de *software*. As aplicações PON são compostas por pequenas entidades reativas e desacopladas, que colaboram por meio de notificações precisas e pontuais, ditando assim o fluxo de execução dessas (SIMÃO e STADZISZ, 2008, 2009) (SIMÃO, *et. al*., 2012a). Essa nova maneira de concepção de *software* tende a proporcionar uma melhora no desempenho das aplicações e, potencialmente, tende a facilitar suas concepções, tanto para ambientes não distribuídos como para ambientes distribuídos (SIMÃO, STADZISZ, 2009) (SIMÃO, *et. al*., 2012a).

O fluxo das iterações das aplicações do PON é realizado de maneira transparente ao desenvolvedor, graças ao orquestramento da cadeia de notificações pontuais entre os elementos PON. Isto é diferente do fluxo de iterações encontrados

em aplicações do PI, particularmente do subparadigma OO, onde o desenvolvedor deve de maneira explícita informar o laço de iterações através de comandos, como *while* e *for.* No PON a repetição ocorre de forma natural na perspectiva de execução da aplicação, conforme exemplificado na Figura 2 e esboçado no diagrama de classes conceitual da Figura 7.

O fluxo de execução ocorre em função da mudança de estado de um objeto *Attribute* de um respectivo *FBE.* Após a mudança de estado do objeto *Attribute*, ele notifica todas as *Premises* pertinentes, para que estas reavaliem seus estados lógicos. Se o valor lógico da *Premise* se altera, a *Premise* colabora com a avaliação lógica de uma ou de um conjunto de *Conditions* conectadas, o que ocorre por meio da notificação sobre a mudança de seu estado lógico a elas. Consequentemente, cada *Condition* notificada avalia o seu valor lógico de acordo com as notificações da *Premise* e com o operador lógico (de conjunção ou disjunção) utilizado. Assim, no caso de uma conjunção, quando todas as *Premises* que integram uma *Condition* são satisfeitas (em estado verdadeiro), a *Condition* também é satisfeita, resultando na aprovação da sua respectiva *Rule* que pode ser executada (BANASZEWSKI, 2009) (SIMÃO, STADZISZ, 2008, 2009) (SIMÃO, *et. al*., 2012a).

Ainda, quando uma dada *Rule* aprovada está pronta para executar (*i.e.* com conflitos resolvidos conforme discute a próxima subseção), a sua *Action* é ativada. Uma *Action* é conectada a um ou vários *Instigations*. Os *Instigations* colaboram com as atividades das *Actions*, acionando a execução de algum serviço de um objeto *FBE* por meio dos seus objetos *Methods*. Geralmente, as chamadas para os *Methods* mudam os estados dos *Attributes* e o ciclo de notificação recomeça (BANASZEWSKI, 2009).

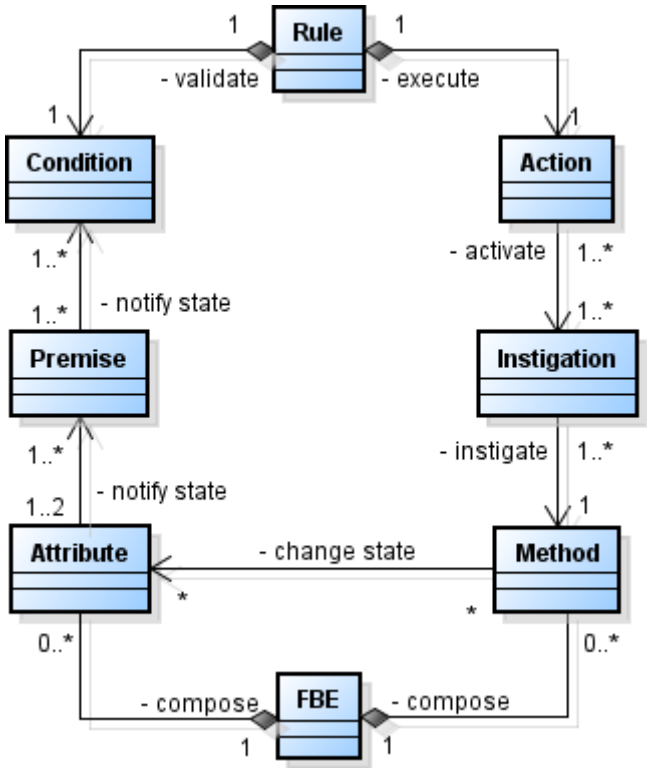


Figura 7 – Principais entidades do PON e seus relacionamentos por notificação (BANASZEWSKI, 2009)

Oportunamente, as conexões entre os objetos notificantes são estabelecidas em tempo de criação e por emergência. Por exemplo, na criação de um objeto *Premise* pelo menos um objeto *Attribute* é considerado. Uma vez que um *Attribute* é referenciado em uma *Premise*, o *Attribute* considera automaticamente esta *Premise* como sendo interessada em receber notificações sobre o seu estado. Assim, o *Attribute* identifica todas as *Premises* interessadas e notifica-as quando o seu estado muda. Ainda, mecanismo similar ocorre em relação as *Premises* e as *Conditions,* bem como em relação as *Conditions* e a *Rules* (BANASZEWSKI, 2009).

2.3.2 Resolução de Conflitos no PON

Um conflito ocorre quando duas ou mais *Rules* referenciam um mesmo *FBE* e demandam exclusividade de acesso a este *FBE*. Deste modo, as *Rules* concorrem para adquirir acesso exclusivo a este *FBE*, sendo que somente uma destas *Rules* em conflito pode executar por vez, a qual obteve o acesso exclusivo. Neste âmbito, para resolver as questões de resolução de conflitos entre as *Rules*, basicamente o

fluxo de sua execução é determinada segunda uma estratégia pré-estabelecida. Estas estratégias podem variar para alcançar o fluxo de execução pretendido pelo desenvolvedor tanto em ambientes monoprocessados quanto em ambientes multiprocessados.

Em um ambiente monoprocessado, a resolução de conflitos ocorre para estabelecer a ordem de execução das *Rules*, onde apenas uma *Rule* pode executar por vez. Em um ambiente multiprocessado, a resolução de conflitos ocorre para evitar o acesso concorrente a um recurso referenciado por várias *Rules* a fim de manter a consistência da aplicação PON (BANASZEWSKI, 2009).

Em se tratanto de ambientes monoprocessados, basicamente é empregado um escalonador de *Rules* formado por uma estrutura de dados do tipo linear (*e.g.* pilha, fila ou lista) (BANASZEWSKI, 2009). Estas estruturas guardam referências para as *Rules* aprovadas, conforme ilustra a Figura 8. Assim, tais estruturas recebem as *Rules* na ordem em que elas são aprovadas, podendo reorganizá-las de acordo com os preceitos de cada estratégia adotada (BANASZEWSKI, 2009).

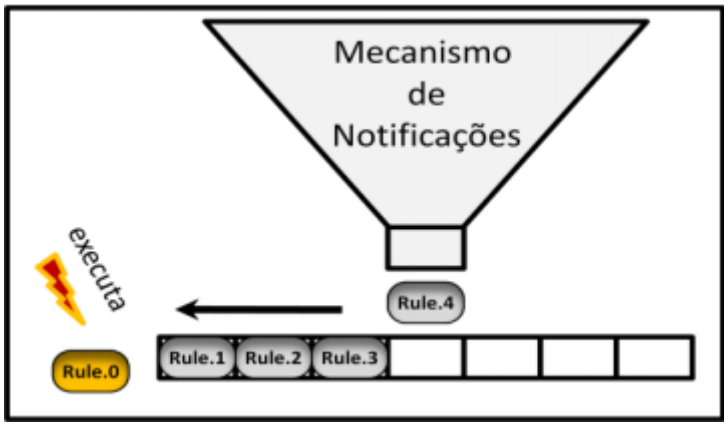


Figura 8 – Modelo Centralizado de Resolução de Conflitos (BANASZEWSKI, 2009)

Desta forma, conforme a estratégia de resolução de conflitos pré-determinada pelo desenvolvedor, as *Rules* em questão serão efetivamente executadas. Neste âmbito, os modelos de resolução de conflitos empregados para o PON em ambientes monoprocessados são:

1. ***BREADTH***: se baseia no escalonamento *First In, First Out (FIFO)*, ou seja, refere-se à execução de entidades *Rule*, seguindo uma estrutura de dados do tipo fila*;*

2. ***DEPTH***: se baseia no escalonamento *Last in, First Out (LIFO)*, ou seja, refere-se à execução de entidades *Rule*, seguindo uma estrutura de dados do tipo pilha;
3. ***PRIORITY***: organiza as entidades *Rule* de acordo com as prioridades definidas nas mesmas;

Quando nenhuma estratégia for definida pelo desenvolvedor, utiliza-se a estratégia *NO_ONE*, a qual faz com que as entidades *Rules* não sejam enviadas ao escalonador/estrutura de dados e sejam aprovadas e executadas imediatamente. Basicamente, a definição do tipo da resolução de conflito adotada pelo desenvolvedor implica no método utilizado para a execução das *Rules* da aplicação. A definição de resolução de conflitos de *Rules* deve ser adotada principalmente em aplicações distribuídas e/ou concorrentes.

As soluções para evitar os conflitos apresentados são particularmente aplicáveis a soluções PON monoprocessadas, ainda que até possam ser úteis em soluções multiprocessadas e distribuídas (BANASZEWSKI, 2009). Entretanto, na verdade, a definição de resolução de conflitos de *Rules* deve ser adotada em aplicações concorrentes e/ou distribuídas com soluções que lhe sejam mais apropriadas. Neste sentido, ainda que este trabalho não seja relativo à aplicação de PON em sistemas concorrentes ou distribuídos, nestes trabalhos (BANASZEWSKI, 2009) (SIMÃO, 2005) (SIMÃO e STADZISZ, 2010) (SIMÃO, *et. al.*, 2010) encontram-se soluções úteis ao PON para resolução de conflitos em aplicações PON distribuídas, bem como soluções correlatas para a garantia de determinismo.

Isto dito, com os conflitos solucionados (e determinismo garantido), uma dada *Rule* aprovada está pronta para executar o conteúdo da sua *Action*. Uma *Action* é conectada a um ou vários *Instigations*. Os *Instigations* colaboram com as atividades das *Actions*, acionando a execução de algum serviço de um objeto *FBE* por meio dos seus objetos *Methods*. Geralmente, as chamadas para os *Methods* mudam os estados dos *Attributes* e o ciclo de notificação recomeça (BANASZEWSKI, 2009).

2.3.3 Propriedades Inerentes ao PON

Nota-se que a essência da computação no PON está organizada e distribuída entre entidades autônomas e reativas que colaboram por meio de notificações

pontuais. Este arranjo forma o mecanismo de notificações, o qual determina o fluxo de execução das aplicações. Por meio deste mecanismo, as responsabilidades de um programa são divididas entre os objetos do modelo, o que permite execução otimizada e ‘desacoplada’ (*i.e.* minimamente acoplada) útil para o aproveitamento correto de mono-processamento, bem como para o processamento distribuído.

Isto dito, toda esta colaboração por meio de notificações pontuais e precisas representaria a solução para as principais deficiências dos atuais paradigmas de programação. Ao evitar buscas sobre entidades passivas, o PON implicitamente evita as redundâncias estruturais e temporais que tanto afetam o desempenho das aplicações no PI e mesmo no PD (BANASZEWSKI, 2009) (SIMÃO, *et. al.*, 2012a).

Ademais, observa-se que os objetos participantes da cadeia de notificação do PON se apresentam desacoplados, devido à comunicação realizada por meio de notificações pontuais. Neste âmbito, pode-se dizer que aplicações no PON possuem características apropriadas para a execução em ambientes multiprocessados, uma vez que ‘somente’ se faz necessário os objetos notificantes conhecerem os endereços dos objetos a serem notificados para a inferência por notificação ocorrer (BANASZEWSKI, 2009).

2.3.4 PON – Utilização x Compreensão

A natureza do PON leva a uma nova maneira de compor *software*, onde os fluxos de execução são distribuídos e colaborativos nas entidades. Assim sendo, o PON permite uma nova maneira de estruturar, executar e pensar os artefatos de *software*. Ainda, muito embora o PON permita compor *software* em alto nível na forma de regras sem o conhecimento desta sua essência, anda assim este conhecimento é importante (SIMÃO, *et. al*., 2012a).

Por exemplo, é importante saber dos impactos de desempenho, das estratégias de resolução de conflitos e das estratégias de distribuição. Neste último caso, um exemplo mais preciso seria a forma de agrupamento de elementos de maior fluxo de notificações juntos, evitando assim comunicações desnecessárias nos canais de comunicação (e.g. redes) (SIMÃO, *et. al*., 2012a).

Ainda, a compreensão dos princípios do PON é importante para aplicações complexas, onde o fluxo de notificações é intenso e precisa de mais formalismo e

rastreabilidade, como em aplicações de tempo real e controle discreto. Na verdade, esse tipo de aplicação pode exigir apoio de ferramentas formais para elaboração do projeto (SIMÃO, *et. al*., 2012a).

Um exemplo particular de formalismo é a rede de Petri. Na verdade, redes de Petri são compatíveis com os sistemas baseados em regras, em geral, em termos de expressão de relações causais (SHEN, JUANG, 2008). Além disso, são particularmente compatíveis com os princípios do PON também em termos da sua essência (SIMÃO, STADZISZ, 2009). Neste contexto, seria necessário conhecer o PON e os princípios do domínio de rede de Petri, compreendendo que ambos são naturalmente compatíveis (SIMÃO, STADZISZ, 2009).

2.3.5 Cálculo Assintótico da Inferência do PON

A complexidade assintótica polinomial do PON, no pior cenário, é representada por $O(n^3)$ ou *O(FactBaseSize * nPremises * nRules),* onde *FactBaseSize* corresponde ao tamanho máximo de objetos *Attributes*, *nPremises* corresponde ao tamanho máximo de objetos *Premises* notificados por estes *Attributes* e *nRules* corresponde ao tamanho máximo de objetos *Conditions* notificados por estas *Premises* (SIMÃO, 2005) (BANAZEWSKI, 2009).

A função assintótica apresentada para o PON, no pior cenário, demonstra uma solução bastante similar ao mecanismo de notificações do algoritmo *HAL* (BANAZEWSKI, 2009). Ainda a função temporal polinomial do *HAL*[1] ($O(n^3)$) se apresenta mais eficiente do que os algoritmos de inferência *RETE*, *TREAT* e *LEAPS* (BANAZEWSKI, 2009).

Esta função assintótica representa a quantidade de notificações entre os objetos colaboradores que também corresponde à quantidade de avaliações lógicas. A constatação desta função assintótica pode ser realizada pela análise da Figura 9,

[1] Em relação ao HAL, apesar das grandes semelhanças, o mecanismo de notificações se diferencia na comunicação mais pontual entre os objetos. Enquanto que no HAL somente os componentes genéricos se comunicam, no mecanismo de notificações as próprias instâncias podem ser comunicar e de forma direta, evitando que entidades genéricas despendam buscas para relacionar as instâncias comunicantes (BANAZEWSKI, 2009).

a qual demonstra as relações por notificações entre os objetos colaboradores. Nesta os *Attributes*, *Premises*, *Conditions* e *Rules* correspondem respectivamente aos símbolos com abreviações *Att, Pr, Cde* e *Rl* (BANAZEWSKI, 2009).

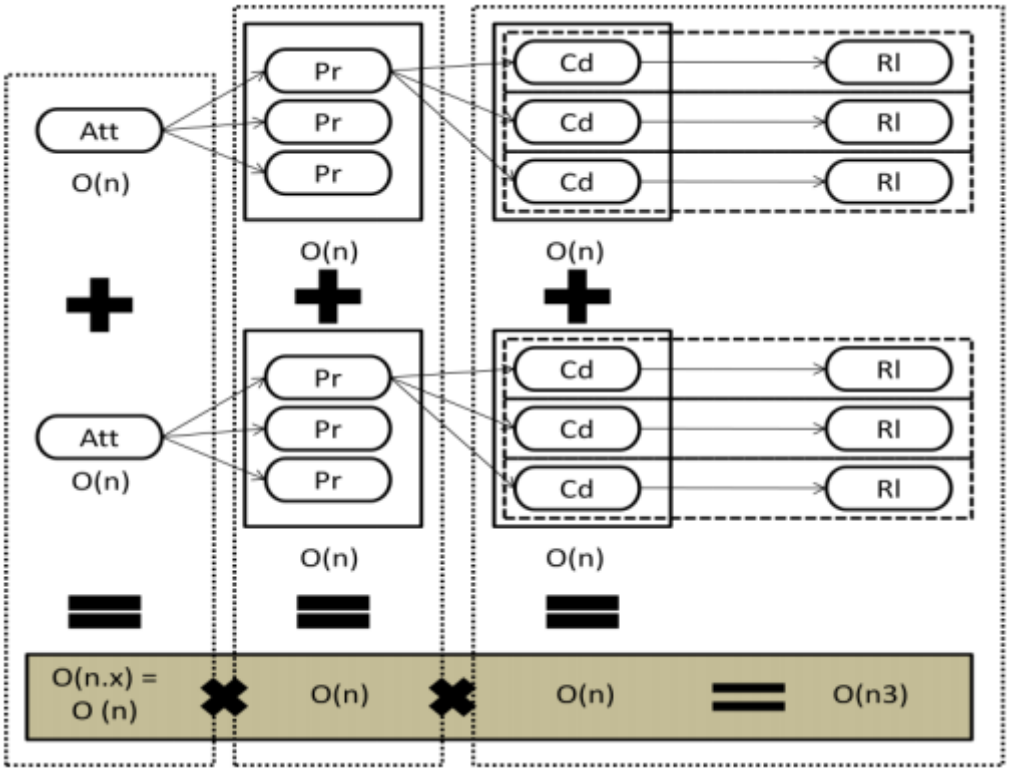


Figura 9 – Cáculo assintótico do mecanismo de notificações (BANAZEWSKI, 2009)

Ademais, outra forma adequada de analisar a complexidade polinomial do PON é considerar o caso médio. A análise da complexidade do caso médio é iniciada analisando-se o começo do processo de notificação do PON através da entidade *Attribute*. Assim, as principais variáveis envolvidas em uma notificação de um *Attribute* são demostradas pela equação da Figura 10.

$$FB_{AT}() = NumPremises + NumRules$$

Figura 10 – Complexidade da Notificação *Attribute* (SIMÃO, 2005).

A variável "*NumPremises"* é a soma de entidades *Premises* ao respectivo *Attribute* e a variável "*NumRules"* é a soma das entidades *Rules* a cada entidade *Premise* contada em *"NumPremises".* Portanto, se for considerado simplesmente cada ciclo de inferência como a instigação de um *Attribute*, uma média possível

seria: *TMedium(x) = (FBAT.1()+ … + FBAT.w()) / w*, onde (*w*) é o numero de todos os *Attributes* existentes. Assim, o resultado desta média seria uma ordem de (n), o que implicaria em uma complexidade linear *O(n)* (SIMÃO, 2005).

2.4 MATERIALIZAÇÃO DO PON

Os conceitos do PON propriamente dito foram primeiramente materializados sobre o POO, através de um arquétipo ou *framework* desenvolvido com a linguagem de programação C++. A versão prototipal feita por Simão e versão original do *Framework* PON feita por Banaszewski (BANASZEWSKI, 2009) foram implementadas especificamente para ambientes monoprocessados, contemplando os conceitos do paradigma PON apresentado na seção anterior. Dado que o original se demonstra melhor que prototipal, em termos de desempenho, o primeiro que será considerado neste trabalho.

Estruturalmente, o *Framework* PON (original) materializa as entidades colaboradoras do paradigma em forma de classes/objetos relacionados através de estruturas de dados com referências às entidades interessadas em seus estados. Outrossim, nesta materialização, o desenvolvedor se preocupa somente em instanciar *Rules* em alto nível no código C++ baseado no *Framework* PON, onde o encadeamento das estruturas notificantes será realizada em tempo de compilação em cada aplicação PON.

Neste âmbito, uma estrutura de pacotes foi projetada, conforme Figura 11, para modelar (e explicar) esta materialização do PON. Isto dito, para desenvolver aplicações com o *Framework* PON em questão, é necessário que o desenvolvedor estenda a classe *Application* contida no pacote *Application,* a qual define uma ponte de ligação entre uma aplicação PON e seu *Framework.* Ainda, esta classe *Application* se relaciona com as classes contidas no pacote *Core.* O pacote *Core* por sua vez*,* contém as classes responsáveis pelo processo de notificação e realização do cálculo lógico causal do PON.

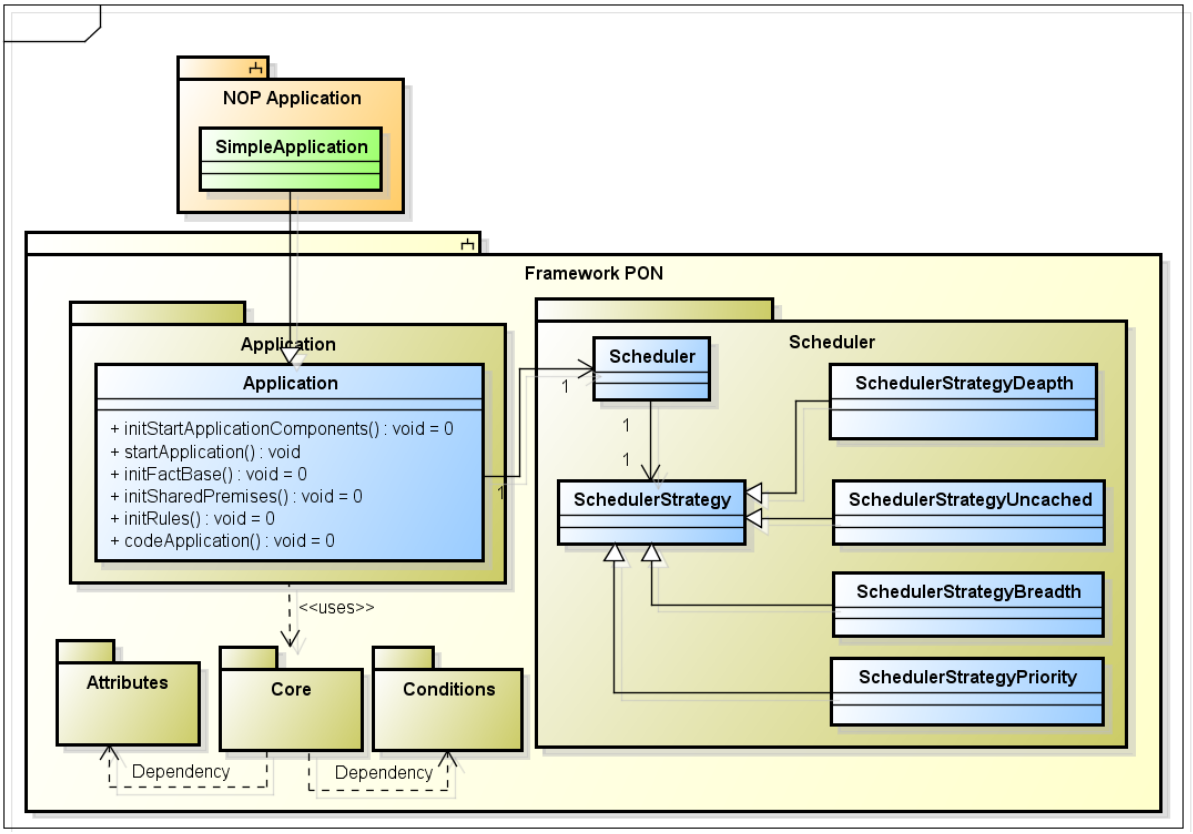


Figura 11 – Estrutura do *Framework* (original) do PON

Conforme ilustrado na Figura 11, o *Framework* PON é subdividido em alguns pacotes, porém as entidades notificadoras que dão “vida” às aplicações desenvolvidas nesse paradigma estão concentradas no pacote *Core*, o qual é apresentado na seção 2.4.1. Ainda, a seção 2.4.2 e 2.4.3 apresentam respectivamente a definição e utilização da classe *FBE* e *Rule.* A seção 2.4.4 descreve sobre as particularidades de implementação de algumas entidades do PON. A seção 2.4.5, por sua vez, explica a implementação da cadeia de notificações do PON. A seção 2.4.6, particularmente, contextualiza o uso dos contêineres da *STL* no âmbito do processo de notificações da materialização desse paradigma. Por fim, a seção 2.4.7 apresenta uma reflexão sobre a materialização do PON.

2.4.1 Pacote *Core*

O pacote *Core* é formado pelas classes colaboradoras que realizam o processo de notificação do *Framework* PON. A Figura 12 ilustra o diagrama de classes do pacote *Core*.

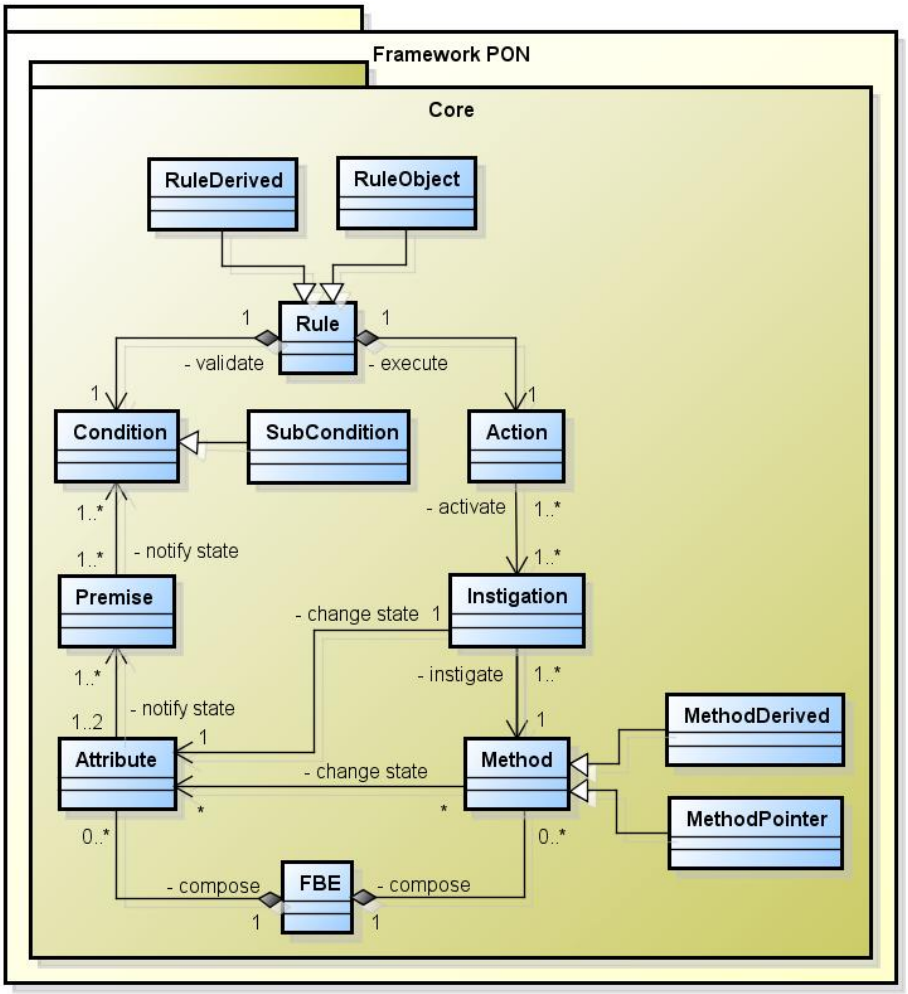


Figura 12 – Estrutura do pacote *Core*

As classes *Rule* e *FBE* se apresentam nas extremidades opostas e se relacionam por meio de suas classes colaboradoras – *Attribute*, *Premise*, *Condition*, *Action*, *Instigation* e *Method* – sendo que a colaboração entre os objetos destas classes determina o fluxo de execução de uma aplicação do PON. Ademais, as classes *Method* e *Rule* que definem as entidades puras do PON são estendidas de modo a proporcionar funcionalidades adicionais.

Neste âmbito, a título de exemplificação, a representação do *FBE Sale* da Figura 13 contempla a definição de certos métodos. Estes métodos consistem em instâncias de subclasses da classe *Method*. Estas subclasses consistem na classe *MethodPointer* e *MtNotifyClient*, onde esta última é definida explicitamente pelo desenvolvedor ao estender a classe *MethodDerived*.

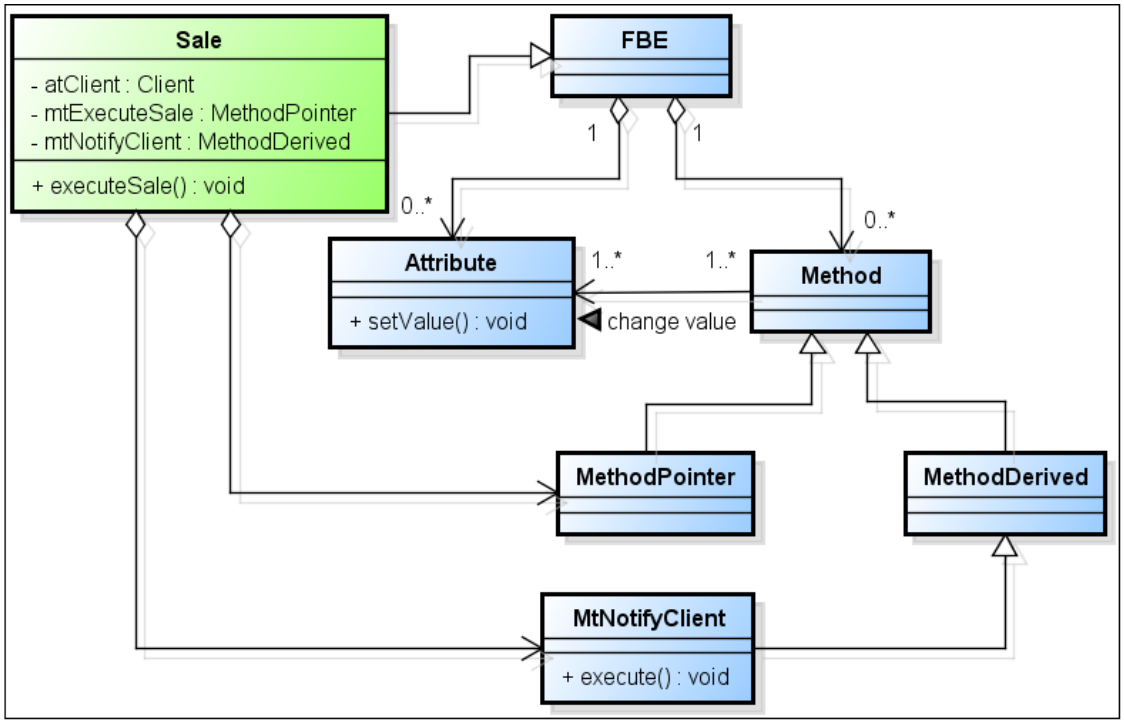


Figura 13 – Utilização de *Methods* no PON

A seu turno, uma classe *MethodPointer* modela os objetos que, cada qual, encapsula a chamada de um método definido e declarado na estrutura de um *FBE*, sendo o endereço deste método atribuído ao objeto no momento de sua criação. No exemplo da Figura 13, o objeto *mtExecuteSale* da classe *Sale* é uma instância de *MethodPointer*, a qual encapsula a chamada para o método *executeSale()* definida em sua própria classe.

A classe *MtNotifyClient*, por sua vez, diferentemente de *MethodPointer*, foi definida explicitamente pelo desenvolvedor. Nesta, o desenvolvedor estende a estrutura de *MethodDerived* e implementa o respectivo comportamento (*i.e.* conjunto de instruções) para notificar o respectivo cliente. A instância *mtNotifyClient* da classe *Sale*, quando invocada pelo respectivo *Instigation*, executará o método implementado em seu próprio escopo. Esta prática atribui maior capacidade de

distribuição aos objetos, uma vez que a estrutura dos *FBEs* se encontra mais desacoplada de suas implementações.

Outrossim, a classe *Rule* possui duas classes derivadas: *RuleObject* e *RuleDerived,* conforme esboçado pela Figura 12. A classe *RuleObject* representa o tipo de regra padrão utilizado no *Framework*, em que uma *Rule* é estruturada em sua forma completa constituída de todos seus elementos (*Premise*, *Condition*, *Action* e *Instigation*). No entanto, sob outro viés de execução, a classe *RuleDerived* implementa em seu escopo o conteúdo de execução de uma *Action.* Assim, o método *execute()* é implementado em seu próprio contexto de execução. Nesta, a ação da *Rule* é implementada com toda a flexibilidade da PI, permitindo atribuições de valores e chamadas de métodos implementados de forma imperativa. O Algoritmo 2 exemplifica a utilização da classe *RuleDerived*.

```
1  RuleDerived* ruleExecuteSale = new RuleDerived(Condition::CONJUNCTION);
2  ruleExecuteSale->addPremise(client->atStatus, true, Premise::EQUAL);
3  ruleExecuteSale->addPremise(client->atCreditLimit,
4                   saleOrder->atTotalSaleOrder, Premise::GREATEROREQUAL);
5  ...
6  //método a ser executado após a aprovação da Rule.
7  RuleDerived::execute() {
8        cout << " :..Sale Order Confirm..:" << "Congratulations " <<
9              this->atClient->atName->getValue() << endl;
10       cout << " :..Your Limit Credit is: " <<
11             this->atClient->atCreditLimit->getValue() << endl;
12       cout << " :..Total Sale is:" <<
13             this->atTotalSaleOrder->getValue() << endl;
14       cout << " :..Done..:" << endl;
15 }
```

Algoritmo 2 – Exemplo do uso de *RuleDerived*

No exemplo do Algoritmo 2, após a aprovação da *Rule "ruleExecuteSale",* o método *execute()* da linha 7 será invocado. Desta forma, o fluxo considerado padrão no processo de notificação nomeadamente (*Action, Instigation* e *Method*) não é executado. Particularmente estes três (tipos de) elementos possuiriam maior utilidade em ambientes multiprocessados, a fim de representar maior desacoplamento e independência entre as partes constituintes responsáveis pela execução da *Rule*.

Ainda, o pacote *Core* é composto também pelos subpacotes *Attributes* e *Conditions*. Conforme apresenta a Figura 14, o subpacote *Attribute* é formado pelas classes responsáveis por encapsular os tipos primitivos da POO. Estas classes

(*Boolean, Char, Double, Integer e String*) introduzem reatividade aos tipos primitivos, permitindo que estes façam parte de estruturas causais do PON.

Ademais, conforme a Figura 14, o subpacote *Conditions* é formado pelas classes que fazem parte da composição de uma respectiva *Condition*. Particularmente, a classe *LogicalOperator* e suas derivadas (*Conjunction, Disjunction* e *Single*), definem a operação lógica utilizada pela *Condition* de maneira a aprovar uma determinada *Rule*.

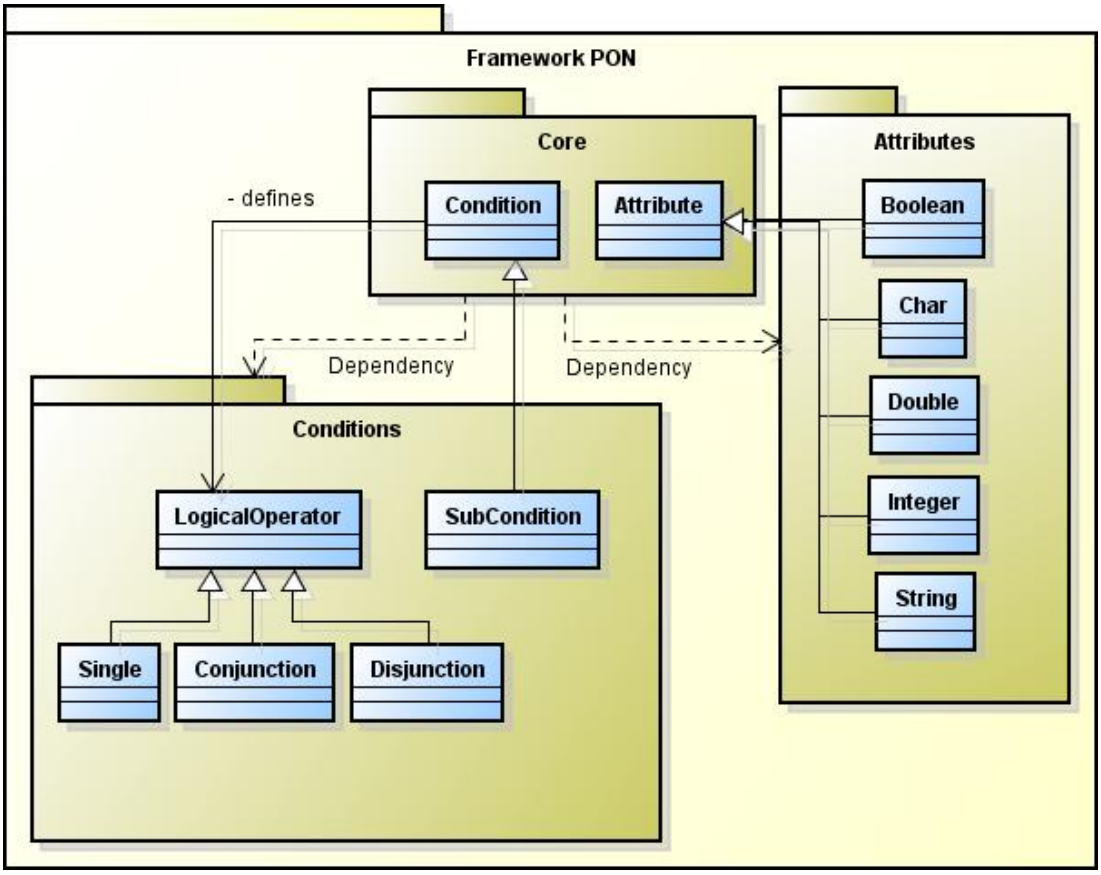


Figura 14 – Estrutura dos subpacotes *Attributes* e *Conditions*

2.4.2 Estrutura da Classe *FBE*

A classe *FBE* representa os elementos da base de fatos, como as entidades do mundo (real ou abstrato) representadas em um problema computacional, e respectivamente descreve os estados e o comportamento desses elementos por meio de seus *Attributes* e *Methods*. Cada novo tipo na base de fatos é representado por uma classe/objeto derivado da classe *FBE*. A Figura 15 ilustra um exemplo com

três subclasses da classe *FBE*, denominadas *Sale, Client* e *Product* em notação *UML*.

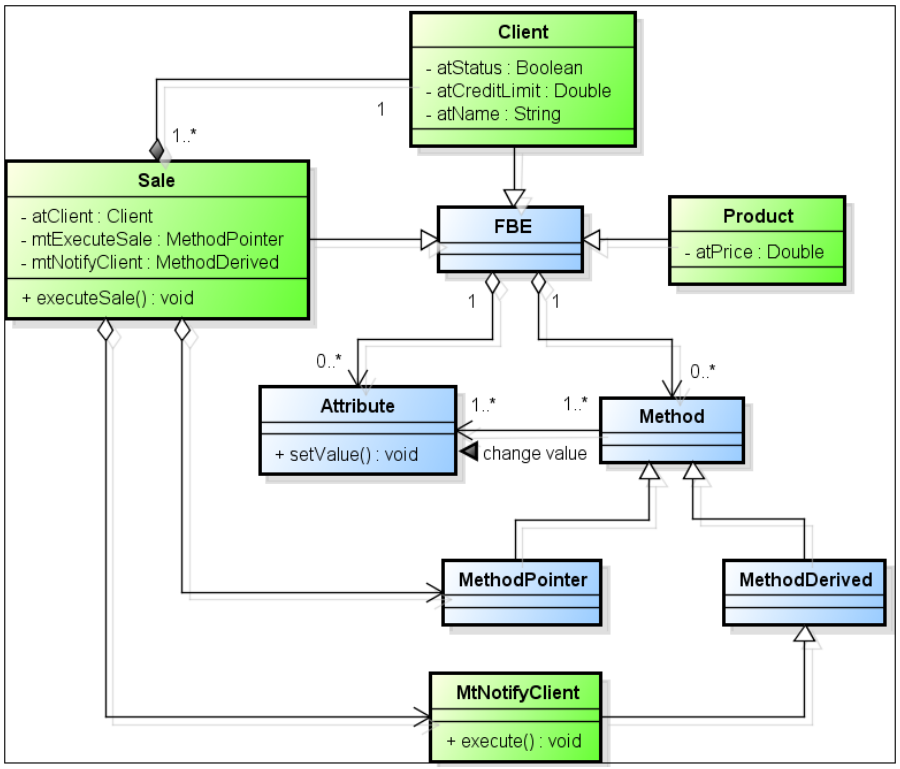


Figura 15 – Exemplo da Representação de Elementos Derivados de *FBE*

Cada objeto de classe derivada da classe *FBE*, como as classes *Sale, Client* e *Product*, representada pelo diagrama de classe da Figura 15, é capaz de reagir após a mudança de seu estado, assim como se relacionar a outros *FBE*s. Ainda na Figura 15, as classes em azul (*FBE, Attribute, Method, MethodPointer, MethodDerived*) representam classes do *Framework* PON enquanto as classes em verde (*Sale, Client, Product* e *MtNotifyClient*) representam as classes implementadas pelo desenvolvedor.

Outrossim, em cada uma destas subclasses do *Attribute*, há a implementação de uma versão particular de um método chamado *setValue* (*value*). Basicamente, a responsabilidade deste método é receber um valor por parâmetro e atribuí-lo ao dado encapsulado na instância em questão, podendo assim, gerar notificações aos objetos *Premises* conectados, caso este valor seja diferente do anterior.

Ainda, na representação do *FBE Sale*, existe a definição de dois métodos *mtExecuteSale* e *mtNotifyClient,* os quais foram devidamente explicados na seção

anterior. Estes métodos consistem em instâncias da classe *Method,* que por sua vez, são responsáveis por encapsular funcionalidades/serviços inerentes ao *FBE Sale.* Desta forma, quando instigados pelos objetos da classe *Instigation,* estes *Methods* serão devidamente executados.

Por fim, com a identificação das classes *FBE*s e supondo que estes já foram implementados, os objetos *FBE*s poderão ser definidos no escopo de uma aplicação PON. O código do Algoritmo 3 apresenta um exemplo de sua instanciação bem como de utilização.

```
1  Client *client;
2  Sale *sale;
3  ...
4  void myApplication::initFacts(){
5
6        client = new Client();
7        client->atName->setValue("João");
8        client->atStatus->setValue(true);
9        client->atCreditLimit->setValue(1000);
10
11       sale = new Sale();
12       sale->atCliente->setValue(client);
13
14 }
```

Algoritmo 3 – Instanciação e Utilização de *FBE*

Nas linhas 1 e 2, a definição das instâncias *FBE*s *client* e *sale,* (respectivamente das classes *FBEs Client* e *Sale*) é realizada. Após a instanciação do *FBE client* na linha 6, é atribuído ao seu *Attribute atName* o valor "*João*" (linha 7), ao seu *Attribute atStatus* o valor "*true"* (linha 8) e, por fim, o *Attribute atCreditLimit* o valor 1000 (linha 9). O *FBE* sale, ao seu turno, se relaciona com o *Attribute client*, conforme observado na linha 12.

2.4.3 Estrutura da Classe *Rule*

Conforme descrito previamente, uma *Rule* é formada por um conjunto de entidades colaboradoras que atuam na aprovação e execução dela. De maneira geral, cada instância *Rule* é composta por uma instância *Condition* e uma instância *Action*. A Figura 16 ilustra o diagrama de classes do pacote *Core* pela perspectiva da estrutura da classe *Rule*.

Conforme ilustra a Figura 16, a parte *Condition* de uma *Rule*, responsável pela aprovação desta, recebe notificações pontuais das *Premises* que a compõem. Ademais, quando uma nova entidade *Premise* é criada, ela se conecta automaticamente às respectivas entidades *Attribute* referenciadas. Assim, com qualquer alteração nos valores de tais *Attributes*, os mesmos terão como notificar as *Premises* conectadas a esse.

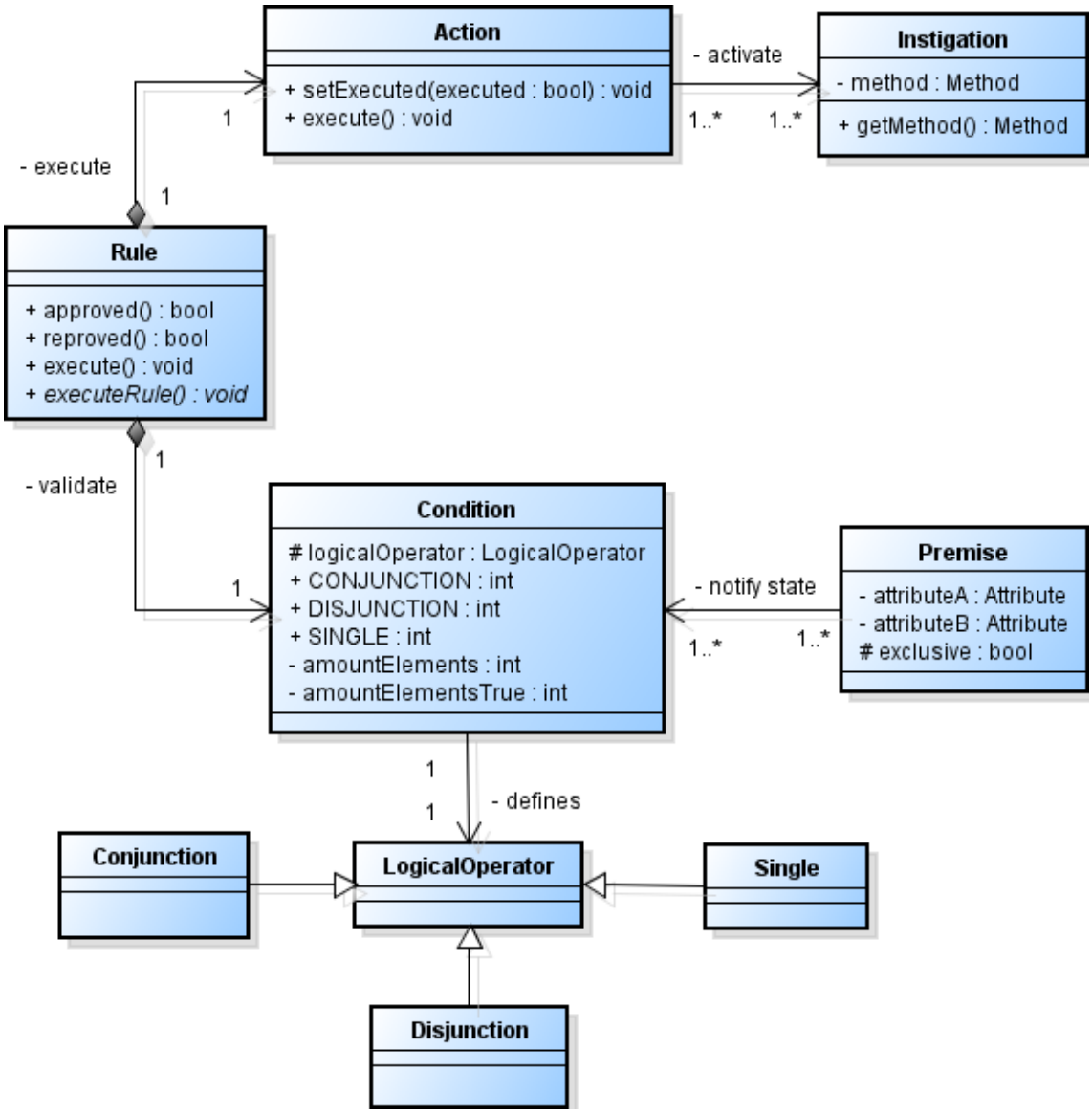


Figura 16 – Diagrama de classes - Estrutura de uma classe/objeto *Rule*

Uma *Action* pode possuir referências para uma ou mais entidades *Instigations*, que constituem o seu corpo. Quando uma entidade *Rule* é aprovada, a entidade *Action* conexa a ela instiga as entidades *Instigations* relacionadas, que por

sua vez invocam os serviços das entidades *Methods*. Por meio de cada *Method,* um conjunto de serviços de um *FBE* é ativado e executado (WIECHETECK, 2011).

A título de exemplificação a estrutura de uma dada *Rule,* a *ruleExecuteSale,* é apresentada na Figura 17, a qual faria parte do escopo da implementação de um sistema de pedido de vendas, a ser apresentado no capítulo 3. Essencialmente, esta *Rule ruleExecuteSale* apresenta a sua *Condition* composta por três objetos *Premises* ((*client->atStatus == ACTIVE*) (*client->atCreditLimit >= sale->atTotalSale*) e (*product->atStock >= product->minimumStock*)) e sua *Action* composta por dois objetos *Instigations* ((*sale->mtExecuteSale*) e (*sale->mtNotifyClient*)).

Cada instância *Premise*, particularmente possui três parâmetros (necessários para a invocação do seu respectivo método construtor) sendo:

- ***Reference***: guarda referência a um objeto *Attribute*, o qual notifica o objeto *Premise* quando ocorre mudança em seu estado.
- ***Operator***: operador lógico usado para fazer comparações entre os valores *Reference* e *Value*. Em PON um operador lógico pode ser dos tipos: *DIFFERENT* (diferente), *GREATERTHAN* (maior que), *SMALLERTHAN* (menor que), *GREATEROREQUAL* (maior ou igual) e *SMALLEROREQUAL* (menor ou igual).
- ***Value***: pode ser uma constante ou uma referência para outro objeto *Attribute*.

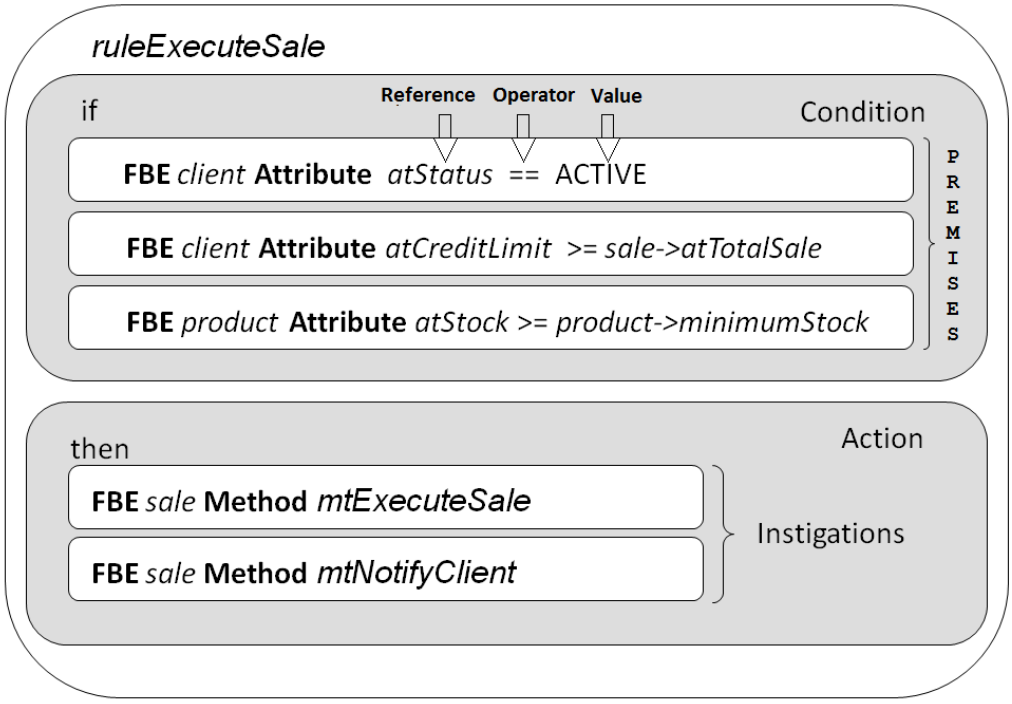


Figura 17 – Representação de uma Regra em PON

Para uma melhor exemplificação, o Algoritmo 4 representa a implementação da *ruleExecuteSale* em *Framework* PON sobre a linguagem de programação C++. A *Rule* em questão é representada pela instanciação de um objeto da classe *RuleObject* e o seu conhecimento é inserido nesta estrutura por meio da passagem de parâmetros aos métodos intuitivos da *Rule*.

No construtor da classe *RuleObject* é definido no primeiro parâmetro, o nome da *Rule*, na sequência é definida a estratégia de sua execução e por fim, a respectiva condição da *Rule.* Entre as linhas 3 e 7, as *Premises* são adicionadas a *Rule* através do método *addPremise*(...). Ainda, nas linhas 8 e 9, são atribuídas as *Instigations* a *Rule* através da chamada do método *addInstigation*(...), o qual encapusula a chamada de *Methods* do PON. Em tempo de compilação, de acordo com os parâmetros, a instancia da classe *RuleObject* cria os objetos colaboradores (*Attribute*, *Premise*, *Condition, Instigation, Method* e *Action*) e os conecta a fim de que possam constituir o mecanismo de notificações.

```
1  RuleObject* ruleExecuteSale = new RuleObject("ruleExecuteSale",
2  scheduler, Condition::CONJUNCTION);
3  ruleExecuteSale->addPremise(client->atStatus, ACTIVE, Premise::EQUAL);
4  ruleExecuteSale->addPremise(client->atCreditLimit, sale->atTotalSale,
5  Premise::GREATERTHAN);
6  ruleExecuteSale->addPremise(produtc->atStock, product->atMinimumStock,
7  Premise::GREATEROREQUAL);
8  ruleExecuteSale->addInstigation(sale->mtExecuteSale);
9  ruleExecuteSale->addInstigation(sale->mtNotifyClient);
10 ruleExecuteSale->end();
```

Algoritmo 4 – *Rule* Executar Venda

As *Premises* são relacionadas pela *Condition* exemplificada por meio de uma conjunção. Como observado na estrutura da *Rule* exemplificada, o operador lógico de uma *Condition* é definido por meio da passagem de uma constante identificadora ao construtor da *Rule* (linhas 1 e 2). Estas constantes podem denotar uma conjunção (pelo identificador *CONJUNCTION*), uma disjunção (pelo identificador *DISJUNCTION*) ou nenhum operador (pelo identificador *SINGLE*). O uso do identificador *SINGLE* é apropriado quando a *Condition* apresenta apenas uma *Premise* conectada, dispensando avaliações por meio de operadores lógicos.

Na *Rule* apresentada, a *Action* corresponde a dois objetos *Instigations* conectados. A execução destes *Instigations* depende da forma pela qual foram definidos. Na definição da *Rule* em questão, estas *Instigations* foram conectados a *Action* por meio da execução do método *addInstigation*. Nas linhas 8 e 9 do

Algoritmo 4, a *Instigation* conectada recebe uma referência para um objeto *Method* do PON (“*mtExecuteSale” e “mtNotifyClient”*) a fim de invocar o respectivo método OO encapsulado por eles.

Outrossim, existe outra variação definida ao construtor da classe *Instigation*, o qual apresenta uma referência direta a um respectivo *Attribute* bem como seu valor (*value*). O Algoritmo 5 na linha 2 apresenta a *Instigation* conectada a uma referência para um *Attribute (product->atStock)* e um valor (*quantity*) a ser atribuído diretamente ao respectivo *Attribute*.

```
1 ...
2       ruleLowStock->addInstigation(product->atStock, quantity);
3 ...
```

Algoritmo 5 – Acesso direto ao *Attribute*

2.4.4 Particularidades do *Framework* PON

Nesta seção, algumas particularidades de implementação do *Framework* PON serão apresentadas. Essas particularidades são referentes a alguns elementos do processo de notificação do PON, tais como: *Attribute*, *Premise* e *Conditon.*

2.4.4.1 Renotificações

A funcionalidade de renotificação força a notificação de uma entidade, mesmo quando o estado desta permanecer inalterado. A implementação padrão de um *Attribute* somente notifica os objetos *Premises* vinculados àquele *Attribute* quando o estado deste for alterado. Todavia, existem situações que demandam que uma *Rule* seja reavaliada e executada novamente, mesmo quando os estados de suas entidades colaboradoras não tenham sido alterados.

Assim, para que uma *Rule* seja executada mesmo quando o estado de um *Attribute* permanecer inalterado, a funcionalidade de renotificação deve ser utilizada. Para isso, o método *setValue()* da classe *Attribute* proporciona uma *flag* para iniciar o fluxo de notificações sob quaisquer circunstâncias (*i.e. Attribute::RENOTIFY*). A utilização de tal funcionalidade é exemplificada na Algoritmo 6.

```
1  ...
2        atStatus->setValue(true, Attribute::RENOTIFY);
3  ...
```

Algoritmo 6 – Exemplo do uso de renotificações

Conforme apresenta o Algoritmo 6, ao atribuir o estado *true* para a entidade *Attribute atStatus*, fornecendo como parâmetro a *flag Attribute::RENOTIFY*, o fluxo de execuções será disparado, notificando todas as entidades *Premise* pertinentes, mesmo que o estado do respectivo *Attribute* mantenha-se inalterado.

2.4.4.2 Implementação da *Premise*

Há situações particulares em que uma *Premise* não precisa notificar todas as *Conditions* de sua lista, bastando que as notificações ocorram até que uma das *Conditions* seja satisfeita. Nestes casos, somente um único objeto *Rule* é executado em relação ao estado de uma *Premise* específica. Em PON, esse comportamento é possível por meio da utilização da *Exclusive Premise* (premissa exclusiva) – definida pelo atributo *exclusive* da classe *Premise.*

Para que o comportamento de uma *Premise exclusive* seja ativado, o desenvolvedor deve explicitamente informar esta opção por meio da atribuição da *flag Premise::EXCLUSIVE* ao construtor de uma *Premise*. O Algoritmo 7 exemplifica a utilização da funcionalidade *Premise exclusive* no exemplo do robô-motorista de Banaszewski (2009). Neste exemplo, a utilização da *flag* pode ser observada nas linhas 5, 13 e 20, na *Premise* que confirma o direcionamento do veículo em um cruzamento.

```
1  ...
2
3  RuleObject* ruleCurveToLeft = new RuleObject(Condition::CONJUNCTION);
4  ruleCurveToLeft->addPremise(semaphore->atSignal, Semaphore::GREEN,
5  Premise::EQUAL, Premise::EXCLUSIVE);
6  ruleCurveToLeft->addPremise(crosswalk->atHasWalker, false,
7  Premise::EQUAL);
8  ruleCurveToLeft->addPremise(car->atCurve, Car::LEFT, Premise::EQUAL);
9  ruleCurveToLeft->addInstigation(car->mtCurveToLeft);
10
11 RuleObject* ruleNoCurve = new RuleObject(Condition::CONJUNCTION);
12 ruleNoCurve->addPremise(semaphore->atSignal, Semaphore::GREEN,
13 Premise::EQUAL, Premise::EXCLUSIVE);
14 ruleNoCurve->addPremise(crosswalk->atHasWalker, false, Premise::EQUAL);
15 ruleNoCurve->addPremise(car->atCurve, Car::NO, Premise::EQUAL);
16 ruleNoCurve->addInstigation(car->mtNoCurve);
17
```

```
18 RuleObject* ruleCurveToRight = new RuleObject(Condition::CONJUNCTION);
19 ruleCurveToRight->addPremise(semaphore->atSignal, Semaphore::GREEN,
20 Premise::EQUAL, Premise::EXCLUSIVE);
21 ruleCurveToRight->addPremise(crosswalk->atHasWalker, false,
22 Premise::EQUAL);
23 ruleCurveToRight->addPremise(car->atCurve, Car::RIGHT, Premise::EQUAL);
24 ruleCurveToRight->addInstigation(car->mtCurveToRight);
```

Algoritmo 7 – Exemplo do uso de *Premise exclusive*

Neste exemplo, ao se aproximar de um cruzamento, o robô-motorista recalcula os seus direcionamentos e define a sua próxima manobra. Neste contexto, ao detectar o sinal verde do semáforo, uma das três *Rules* apresentadas no Algoritmo 7 deve ser executada, uma vez que o motorista pode seguir apenas em uma direção em um determinado momento (BANAZEWSKI, 2009).

Na implementação esboçada no Algoritmo 7, as *Premises* descritas nas linhas 5, 13 e 20, fazem parte das três condições relacionadas. No momento em que a lista de *Conditions* relacionada à respectiva *Premise* é iterada, quando houver uma condição satisfeita, e a mesma ativar uma *Rule*, a iteração sobre as demais *Conditions* é interrompida. Com isto, a *Premise* exclusiva encerra o seu ciclo de notificações, evitando que as próximas *Conditions* sejam notificadas.

Este mecanismo de decisão é esquematizado na Figura 18, onde a *Premise Exclusive* percorre a sua lista de três *Condition*s de forma sequencial, da esquerda para a direita. Na primeira notificação, a respectiva *Condition* não é satisfeita, confirmando este estado para a *Premise Exclusive*. Na seqüência, a segunda *Condition* é notificada, a qual atualiza o seu estado lógico e responde com uma confirmação de aprovação à *Premise Exclusive*. Com isto, a *Premise Exclusive* encerra o seu ciclo de notificações, evitando que as próximas *Condition*s sejam notificadas, pois estas certamente não serão satisfeitas pelo seu estado (BANAZEWSKI, 2009).

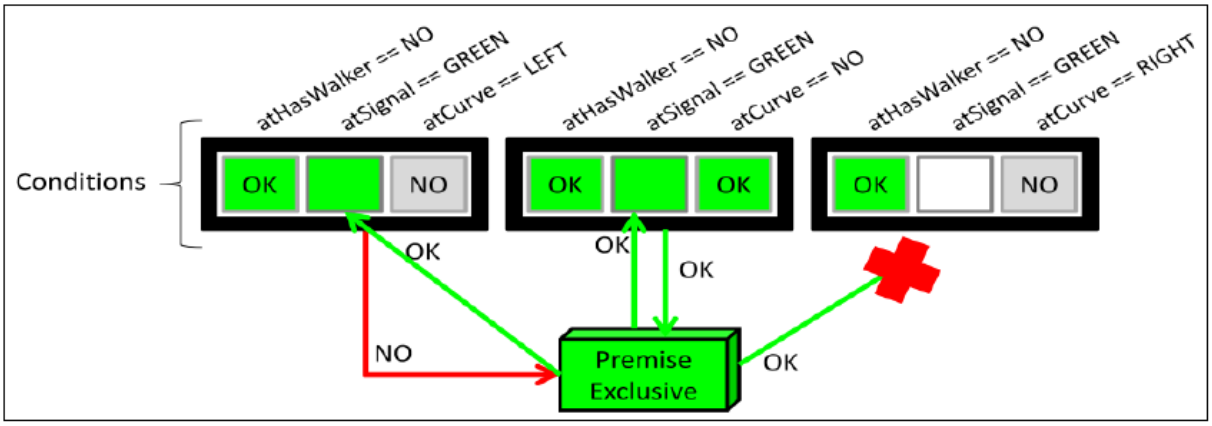


Figura 18 – *Premise Exclusive* (BANAZEWSKI, 2009)

2.4.4.3 Implementação da *Condition*

Para que seja possível implementar validações lógicas compostas em PON, as classes *Conditions* possuem uma funcionalidade opcional. Com esta funcionalidade é possível que as *Conditions* organizem a relação entre suas *Premises* por meio de diferentes combinações de operadores lógicos. Existem casos que requerem a utilização de mais de um operador lógico, como o caso do Algoritmo 8 que apresenta uma implementação de uma expressão causal composta no POO.

```
1 ...
2 if ( (semaphore->atSignal == Semaphore::RED) &&
3      (car->atCurve == Car::LEFT  ||
4       car->atCurve == Car::RIGHT ||
5       car->atCurve == Car::NO))
6 ...
```

Algoritmo 8 – Exemplo de uma expressão causal composta em C++

É possível utilizar mais de um operador lógico por *Condition* a fim de representar expressões causais compostas. Mas, neste caso, deve-se agrupar os objetos *Premise* que estão relacionados por um mesmo tipo de operador lógico, por meio do uso de uma especialização da classe *Condition*, a classe *SubCondition* (Figura 14). Com a utilização de *SubConditions*, as *Premises* notificam a *Condition* indiretamente, por meio das suas *SubConditions*. Neste contexto, as *SubConditions* fazem o papel das *Premises*, notificando as respectivas *Condition*s no momento em que estas tiverem seus estados lógicos alterados. Para uma *Condition* é indiferente receber notificações de *SubConditions* ou *Premises*, uma vez que os seus estados lógicos são atualizados da mesma maneira.

Utilizando entidades *SubCondition,* a expressão causal do Algoritmo 9 é representado em apenas uma entidade *Rule*. Na representação padrão da *Condition* (*i.e.*, sem *Subconditions* e apenas *Premises*) seriam necessárias duas regras para representar essa expressão causal.

```
...
RuleObject* Rulestop = new RuleObject(Condition::CONJUNCTION);

Rulestop->addSubCondition(Condition::SINGLE);
Rulestop->addPremiseToSubCondition(semaphore->atSignal, Semaphore::RED,
                                   Premise::EQUAL);
Rulestop->addSubCondition(Condition::DISJUNCTION);
Rulestop->addPremiseToSubCondition(car->atCurve, Car::LEFT,
                                   Premise::EQUAL);
Rulestop->addPremiseToSubCondition(car->atCurve, Car::RIGHT,
                                   Premise::EQUAL);
Rulestop->addPremiseToSubCondition(car->atCurve, Car::NO,
                                   Premise::EQUAL);
Rulestop->addInstigation(car->mtStop);
...
```

Algoritmo 9 – Representação de uma expressão causal composta no PON

2.4.5 Implementação da Cadeia de Notificações do PON

A concepção de aplicações do PON é realizada através da criação de *Rules.* Ao passo em que a execução de uma aplicação PON propriamente dita, é realizada pela cadeia de notificações, a qual é orquestrada através de notificações pontuais e precisas entre todas as entidades pertinentes a esta cadeia de notificação (Figura 12). Tais entidades pertencentes à cadeia de notificação do PON são comportadas por estruturas de dados de alto nível, especificamente sobre estruturas das classes *list* e *vector* da *STL*. A título de exemplificação, o Algoritmo 10, apresenta a implementação do método *addPremise(...)* da classe *Rule.* Assim, no momento da adição de uma *Premise* da respectiva *Rule,* os elementos PON pertinentes a eles são instanciados e conectados.

```
void Rule::addPremise(Attribute* attributeA, Attribute* attributeB, int
operattor, bool exclusive, int attributesToSave){

      Premise* premiseTmp = new Premise(attributeA, attributeB,
      operattor, exclusive);
      this->condition->addPremise(premiseTmp);
      ...
}
```

Algoritmo 10 – Método *addPremise()* da classe *Rule*

Deste modo, após a chamada do método *addPremise(...),* a *Premise* é instanciada, conforme apresenta a linha 4 do Algoritmo 10. Na sequência, a *Premise* em questão é adicionada ao objeto *condition* da classe *Condition* na linha 6. Assim, após a mudança de estado desta *Premise*, a mesma notifica pontualmente sua *Condition* relacionada. Ainda, o Algoritmo 11, apresenta a implementação do construtor da classe *Premise.*

```
1  Premise::Premise(char* name, Attribute* attributeA, Attribute*
2  attributeB, int operattor, bool exclusive){
3        this->attributeA = attributeA;
4        this->attributeB = attributeB;
5        this->operattor = operattor;
6        this->exclusive = exclusive;
7        setInit(true);
8        setLogicValue( this->logicCalculus() );
9        this->attributeA->addPremise(this);
10       this->attributeB->addPremise(this);
11       setActive(true);
12       setName(name);
13       conditionExclusive = 0;
14 }
```

Algoritmo 11 – Construtor da classe *Premise*

Observa-se, que no construtor da classe *Premise,* os objetos da classe *Attribute* descritos nas linhas 9 e 10 relacionados a ela, se conectam a sua respectiva *Premise* através da execução do método *addPremise*. Neste caso, após a alteração de estado de um destes *Attributes* referenciados, os memos notificarão pontualmente sua(s) *Premise(s)* relacionada(s). Ainda, é importante salientar que todo elemento PON agrega em seu escopo uma estrutura de dados da classe *list* e *vector* da *STL*. Assim, conforme esboçado no Algoritmo 12, o método *addPremise(...)* da classe *Attribute*, adiciona um elemento *Premise* à lista de *Premises* denotado pelo objeto *premisesList* da classe *list* da *STL,* através da chamada do método *push_back*, conforme observado na linha 2*.*

```
1  void Attribute::addPremise(Premise* premise){
2        premisesList.push_back(premise);
3  }
```

Algoritmo 12 – Método *addPremise()* da Classe *Attribute*

Neste âmbito, toda cadeia de notificações do PON é suportada por objetos da *STL,* especificamente sobre objetos da classe *list* e *vector* da *STL.* Assim, cada elemento (*Attribute, Premise, Condition* e *Instigation*) responsável pelo fluxo de notificações do PON, agrega em seu escopo uma lista de elementos a serem notificados. A título de exemplificação, o Algoritmo 13 apresenta a classe *Attribute*, a qual agrega um objeto da classe *list* da *STL* denominada *premisesList.* Importante salientar que os demais métodos e atributos da classe foram suprimidos de modo a facilitar a leitura e compreensão da respectiva classe.

```
1  class Attribute {
2        ...
3  private:
4        Attribute* value;
5        list<Premise*> premisesList;
6        list<Premise*>::iterator it;
7
8  public:
9        const static int RENOTITY = 1;
10
11 public:
12       Attribute();
13       ...
14 }
```

Algoritmo 13 – *Objeto premisesList* da Classe *List da STL*

Conforme observado, a linha 5 contém o objeto *premisesList* da classe *list* da *STL*. Ainda, a linha 6 define seu respectivo objeto iterador *it.* Desta forma, após a mudança de estado de um *Attribute* relacionado a um respectivo *FBE,* a cadeia de notificações é iniciada. Neste caso, o *Attribute* em questão, itera sobre os elementos *Premises* adicionados a sua lista de *Premises "premisesList",* conforme esboçado pelo Algoritmo 14*.*

```
1  void Attribute::renotifyPremises(){
2      Premise* premiseTmp = 0;
3        list<Premise>::iterator it;
4        for( it = premisesList.begin(); it != premisesList.end(); ++it )
5        {
6              premiseTmp = (Premise*)(*it);
7              if(premiseTmp->isExclusive())
8                    premiseTmp->renotifyConditionsAboutPremiseExclusive();
9
10             else
11                   premiseTmp->renotifyConditions();
12       }
13 }
```

Algoritmo 14 – Método *notifyPremises()* da classe *Attribute*

Neste contexto, todas as *Premises* as quais o respectivo *Attribute* se relaciona são iteradas. Da mesma maneira, cada elemento PON *Premise*, notifica seu conjunto de elementos PON *Condition*, através da chamada do método *notifyConditionsAboutPremiseExclusive()* da linha 8 ou *notifyConditions()* da linha 11. Estes dois métodos supracitados, também iteram sobre seus respectivos elementos PON, especificamente sobre elementos *Conditions*, por meio de objetos da classe *list* ou *vector* da *STL.*

Outrossim, uma vez aprovada, a *Rule* seria executada via escalonador pré-definido pelo desenvolvedor da aplicação PON. Para tanto, conforme a política de escalonamento adotada, a *Rule* executaria conforme descrito na seção 2.3.2. Para exemplificar, a resolução de conflitos denominada *BREADTH* é descrita no Algoritmo 15.

```
1  void SchedulerStrategyBreadth::execute(){
2        Rule* ruleTmp = 0;
3        isInferenceExecuting = true;
4
5        while(!listRules.empty()){
6              ruleTmp = listRules.front();
7              if(ruleTmp->isDerived()){
8                    ruleTmp->setExecuting(true);
9                    ruleTmp->executeRule();
10                   ruleTmp->setExecuting(false);
11                   ruleTmp->reproved();
12             }
13             else{
14                   ruleTmp->setExecuting(true);
15                   ruleTmp->execute();
16                   ruleTmp->setExecuting(false);
17                   ruleTmp->reproved();
18             }
19       }
20       isInferenceExecuting = false;
21 }
```

Algoritmo 15 – Resolução de Conflitos *BREADTH*

A resolução de conflitos *BREADTH,* se baseia no escalonamento *First In, First Out (FIFO)*, ou seja, refere-se à execução de entidades *Rules*, seguindo uma estrutura de dados do tipo fila. Para isto, a classe *SchedulerStrategyBreadth* agrega um objeto denominado *listRules* da classe *queue* da *STL.* Este objeto tem por referência objetos da classe *Rule* e a iteração é realizada conforme Algoritmo 15. Neste âmbito, enquanto a fila não for vazia (linha 5), a primeira *Rule* da fila é obtida (linha 6). Caso a *Rule* em questão for uma *Rule* derivada, ou seja, da classe *RuleDerived*, o método a ser executado será o método *executeRule()* da linha 9.

Caso contrário, o método a ser executado será o método *execute()* da linha 15. Por fim, após sua execução, a *Rule* é reprovada (linhas 11 ou 17). Sendo assim, todo o orquestramento da cadeia de notificações é realizado a partir de iterações ou percorrimentos sobre contêineres da *STL,* que correspondem a estruturas de dados de alto nível.

2.4.6 Contêineres da *STL*

Contêineres geralmente possuem controles específicos, tais como verificação de tamanho de *arrays*, facilidades de busca, iterações unidirecional ou bidirecional. Ainda, contêineres possuem estruturas avançadas como árvores binárias, tabelas *hash*, *hash maps* etc. É comum utilizar a classe contêiner como um *template*, onde o tipo de objeto é fornecido como um parâmetro do *template*. Não há custo extra de desempenho ao se utilizar *templates* em contêineres (VANDEVOORDE e JOSUTTIS, 2002) (FOG, 2011).

Vários modelos de classes contêineres estão disponíveis para muitas finalidades. O mais comumente usado em C++ é o conjunto de contêineres da *Standard Template Library* (STL), que normalmente vem com mais modernos compiladores C++ (MEYERS, 2001) (MEYERS, 2005). A vantagem de usar contêineres prontos como *STL* é que se economiza tempo pelo (re)uso das funcionalidades lá disponíveis. Os contêineres *STL* são universais, flexíveis e bem testado. No entanto, a *STL* é projetada para garantir generalidade e flexibilidade, enquanto a velocidade de execução, economia de memória, eficiência de *cache* e tamanho do código têm baixa prioridade (FOG, 2011).

Alguns *templates* STL, tais como *list, vector, set* e *map*, são propensas a atribuir mais blocos de memória do que a quantidade de objetos que há no contêiner (FOG, 2011). A *STL queue* aloca um bloco de memória para cada quatro objetos. A *STL vector* armazena todos os objetos no mesmo bloco de memória, todavia se esta memória necessitar ser realocada, o desperdício de tempo é relativamente grande.

Em um experimento, onde 10 elementos foram inseridos, um por um, em um STL *vector*, originou sete realocações de memória de tamanhos: 1, 2, 3, 4, 6, 9 e 13 objetos, respectivamente (*MS Visual Studio* 2008) (FOG, 2011). Este desperdício de comportamento pode ser evitado chamando o operador *vector::reserv* com uma

previsão ou estimativa de tamanho final necessário, antes de adicionar o primeiro objeto ao vetor. Somente a *STL vector* possui essa funcionalidade (MEYERS, 2001) (FOG, 2011).

A maioria dos contêineres da *STL* é formada por listas duplamente encadeadas, como objetos da classe *list.* Apesar de serem convenientes em termos de operações e funcionalidades, elas são ineficientes em termos de desempenho (FOG, 2011). Na maioria dos casos, listas lineares são por vezes mais eficientes. Ainda, os chamados iteradores, que são usados pela *STL*, não são necessários, caso possa ser usado uma lista linear com um simples índice de iteração.

A generalidade da *STL* também custa em termos de tamanho do código (FOG, 2011). Os objetos armazenados em um contêiner *STL* têm permissão para ter construtores e destruidores. Os construtores e destruidores de cada objeto são executados cada vez que um objeto é movido na memória, o que pode ocorrer com muita frequência (FOG, 2011).

Existem alternativas eficientes que podem ser utilizadas onde a velocidade de execução, economia de memória e tamanho do código tem maior prioridade que a generalidade e flexibilidade encontradas na *STL*. Neste contexto, o procedimento mais importante é a utilização de *memory pool*. É mais eficiente armazenar objetos juntos em um bloco de memória do que armazenar cada objeto em seu próprio bloco. Um bloco de memória contendo muitos objetos pode ser copiado ou movido com uma única instrução de chamada do método *memcpy* ao invés de mover cada objeto separadamente (FOG, 2011).

2.4.7 Reflexão sobre a Materialização do PON

A atual materialização do PON serviu para validar os conceitos relacionados a este paradigma. Principalmente em questões da resolução do cáculo lógico causal e assim a eliminação das redundâncias temporais e estruturais. Na aplicação Mira ao Alvo, elaborada no trabalho de pesquisa de BANAZEWSKI (2009), o PON se mostra já efetivo nesta materialização, mas em outras não, conforme discutido em (LINHARES, *et.al.,* 2011) (BATISTA, *et.al.,* 2011) (RONSZCKA, *et.al.,* 2011) (SIMÃO, *et. al.*, 2012b) (SIMÃO, *et. al.*, 2012c).

As dificuldades descritas nos trabalhos de (LINHARES, *et.al.,* 2011) (BATISTA, *et.al.,* 2011) (RONSZCKA, *et.al.,* 2011) (SIMÃO, *et. al.*, 2012b) (SIMÃO, *et. al.*, 2012c), são referentes principalmente sobre a camada extra de execução das aplicações PON, conforme esboçado na Figura 4. Esta camada refere-se justamente ao *Framework* PON. Isto indica que o mesmo necessita de melhorias, principalmente sobre questões de desempenho.

Ao que tudo indica, o grande gargalo de execução encontra-se no orquestramento da execução da cadeia de notificações do *Framework* PON, o qual se encontra desenvolvido sobre uma estrutura de dados de alto nível, conforme relatado na seção 2.4.5. Ademais, melhorias no sentido de refatorações de código são extremamentes pertinentes. Neste sentido a seção 2.8, descreverá sobre os conceitos relacionados.

Outrossim, o cálculo assintótico do PON demonstra que o mesmo possui capacidades inerentes de evolução. Neste sentido, esforços sobre sua materialização se tornam indispensáveis, de modo a melhorar o desempenho de suas aplicações. Neste âmbito, notificações baseadas em estados, poderiam amenizar as iterações realizadas pela cadeia de notificações do PON. Por exemplo, somente alguns estados de elementos do tipo *Attribute* são verificados por elementos do tipo *Premise.* Neste sentido a função assintótica do PON seria amenizada, uma vez que não é necessário o percurso de todas as *Premises* de um respectivo *Attribute*, conforme observado na Figura 9.

2.5 DESENVOLVIMENTO ORIENTADO A NOTIFICAÇÕES

O método denominado Desenvolvimento Orientado a Notificações (DON), foi criado pelos trabalhos realizados por WIECHETECK (2011). O método é empregado para projetos de *software* baseados em desenvolvimento de aplicações PON programados com o *Framework PON*. O Método DON é compatível com as práticas atuais de engenharia de *software*, porém apresenta técnicas específicas de modelagem orientadas ao desenvolvimento de *softwares* no PON.

Neste âmbito, a criação do método de desenvolvimento em questão requeriu duas etapas: (1) a criação de um perfil em *Unified Modeling Language* (*UML*) denominado *Perfil* PON, que define os principais conceitos do PON por meio da

utilização de mecanismos de extensão da *UML*; e (2) a criação do método DON, propriamente dito, que faz uso do *Perfil* PON (*NOP Profile*) e apresenta uma sequência de passos para a construção de projetos no PON. As próximas seções detalham os dois passos supracitados.

2.5.1 Perfil *UML* para o PON – *NOP Profile*

De maneria sucinta o perfil *UML* para o PON propõe a extensão do metamodelo da *UML,* a qual permite uma nova sintaxe e semântica aos seus elementos constituintes. Um conjunto desses elementos, agrupados dentro de um pacote, formam o chamado Perfil *UML*. Os mecanismos que permitem extensão de novos elementos de modelagem ao metamodelo da *UML* são (LIMA, 2008):

- **Estereótipo**: permite a classificação de um elemento de modelo de acordo com um elemento de modelo base já existente na *UML*. Estereótipos devem possuir restrições e valores etiquetados para adicionar informações necessárias aos novos elementos (ex: *idClass* na Figura 19)
- **Valor etiquetado**: permite explicitar uma propriedade de um elemento, sendo que essas informações podem ser adicionadas a qualquer elemento do modelo. No metamodelo, os valores etiquetados são representados como atributos da classe que define um estereótipo (ex: identificador na Figura 19), mas no modelo são apresentadas em um novo compartimento dos elementos, denominado “*tags*”.
- **Restrição**: é uma informação semântica anexada a um ou mais elementos do modelo para expressar uma condição que o sistema deve satisfazer. Tal especificação é escrita em uma determinada linguagem de restrições, conforme observado na Figura 19.

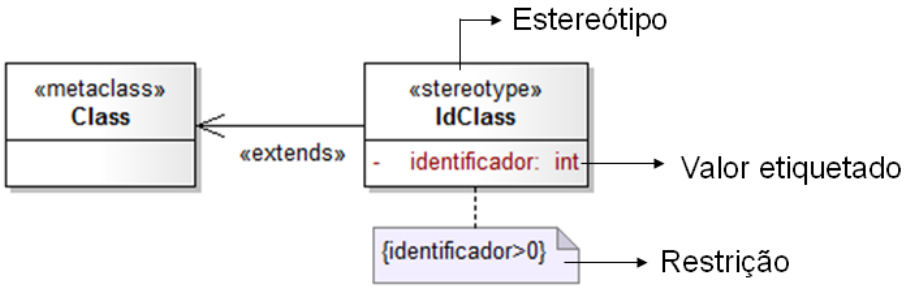


Figura 19 – Mecanismo de extensão da *UML* (WIECHETECK, et. al., 2011)

Neste âmbito, o mescanismo de extensão da *UML* foi atribuído às entidades participantes do modelo de domínio de uma aplicação PON. Um modelo de domínio pode ser construído usando-se os elementos de modelagem da *UML* como classes, pacotes e associações. Assim, para a concepção de aplicações PON, as entidades participantes do modelo de domínio pertencem basicamente às classes que fazem parte do pacote *application* esboçado pela Figura 11 e do pacote *core* esboçado pela Figura 12 (WIECHETECK, *et. al.*, 2011).

Após a identificação dos modelos do domínio do PON, o próximo passo foi a criação do perfil *UML* para o PON, denominado o *NOP Profile*. O *NOP Profile* é composto por um pacote *UML* estereotipado (<<*profile*>>) com mesmo nome do perfil, que compreende outros dois pacotes – *NOP Profile Core* e *NOP Profile Application* – um para cada pacote de classes do *Framework* do PON (WIECHETECK, *et. al.*, 2011).

O *NOP Profile* - Core é composto por elementos de extensão da *UML* (estereótipos, valores etiquetados e restrições) que representam e descrevem os objetos participantes do mecanismo de notificações do PON. Esses elementos de extensão foram obtidos a partir da análise do modelo do domínio da Figura 12 (que ilustra o diagrama de classes do *Framework* PON do pacote *core*). Primeiramente, foi criado um estereótipo para cada elemento relevante definido no modelo do domínio e, em seguida, foram associados esses estereótipos aos elementos do metamodelo da *UML* por meio do relacionamento de extensão (<<*extends*>>). A Figura 20 exibe o perfil *NOP Profile – Core* criado (WIECHETECK, *et. al.*, 2011).

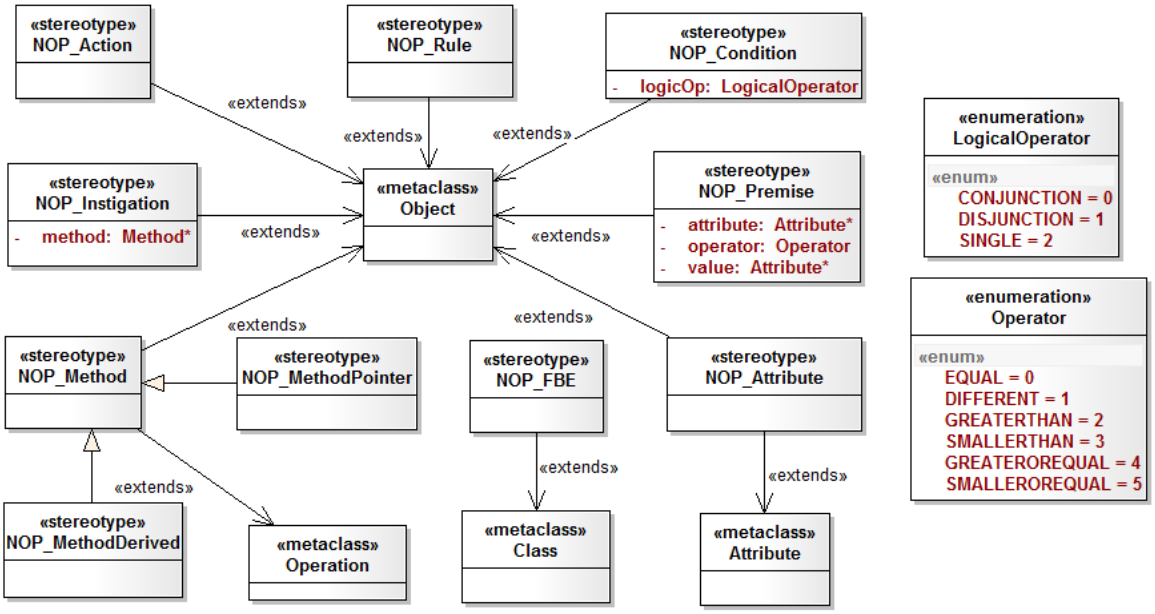


Figura 20 – *NOP Profile* do pacote *core* (WIECHETECK, et. al., 2011)

Seguindo os mesmos passos para a criação do *NOP Profile Core*, foi construído o *NOP Profile Application* analisando-se o modelo de domínio da Figura 11. Neste perfil, a classe *Application* do modelo do domínio foi transformada em um estereótipo – *NOP_Application* – que estende a metaclasse *Class* do metamodelo da *UML*, uma vez que uma nova aplicação em PON é representada como uma subclasse da classe *Application*. Já os métodos da classe *Application* do *Framework* PON foram transformados em estereótipos que estendem a metaclasse *Operation* da *UML*, sendo eles: *initFactBase* e *initRules*. Também foi criada uma enumeração – *SchedulerStrategy* – que define as estratégias de resolução de conflitos entre regras existentes em PON, que podem ser: *BREADTH*, *PRIORITY*, *DEPTH*, *UNCACHED* e *NO_ONE*. A Figura 21 ilustra o pacote de perfil *NOP Profile Application* criado (WIECHETECK, *et. al.*, 2011).

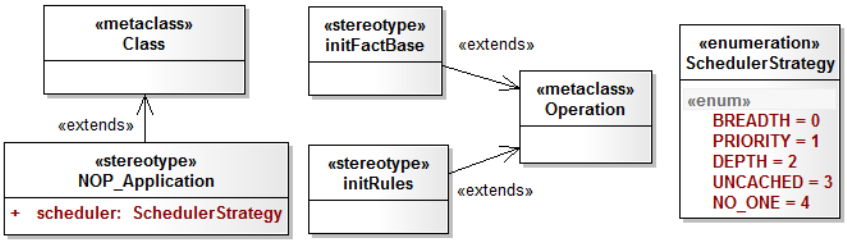


Figura 21 – *NOP Profile Application* do pacote *application* (WIECHETECK, et. al., 2011)

2.5.2 Processo de Desenvolvimento Orientado a Notificações

Após a elaboração do perfil *UML* para o PON (*NOP Profile*), o arquiteto de *software* de aplicações do PON poderá usufruí-lo dentro do processo de Desenvolvimento Orientado a Notificações (DON). O DON é um método para projetos de *softwares* que compreende as fases de requisitos e projeto de um processo de *software* PON (*i.e. Framework* PON para ser preciso). Mais especificamente, quando aplicado ao (já universalisado) processo *Rational Unified Process* (*RUP*), o DON envolve a disciplina de "Requisitos" e adapta a disciplina de "Análise e Projeto" a fim de satisfazer as necessidades de modelagem do PON. A Figura 22 ilustra o *RUP* e a contextualização do DON dentro desse processo (WIECHETECK, 2011).

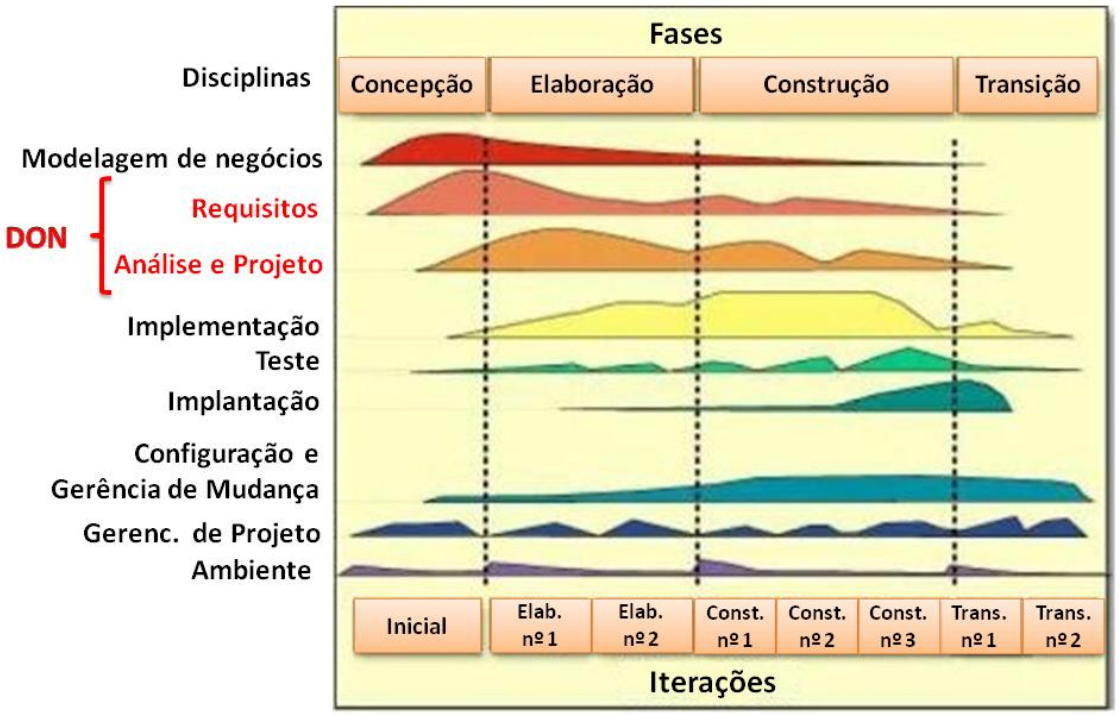


Figura 22 – DON contextualizado no *RUP* (WIECHETECK, 2011)

Aplicando-se nessas disciplinas, o DON apresenta-se como um método iterativo e incremental viabilizando sua adaptação ao *RUP*. O DON é composto de oito etapas que envolvem a criação de diagramas estruturais e comportamentais. O primeiro grupo descreve os elementos estruturais que compõem o sistema,

representando suas partes e seus relacionamentos. Ao seu turno, o segundo grupo descreve o comportamento dos elementos e suas interações (WIECHETECK, 2011).

As oito etapas do método DON são: Capturar Requisitos, Criar Modelo de Casos de Uso, Criar Modelo de Classes, Criar Modelo de Estados de Alto Nível, Criar Modelo de Componentes, Criar Modelo de Sequência, Criar Modelo de Comunicação e Criar Modelo de Redes de Petri. As duas primeiras são referentes à disciplina de “Requisitos” do *RUP*, e as seis restantes são referentes à disciplina de “Análise e Projeto”. A Figura 23 exibe o Método DON e suas etapas (WIECHETECK, 2011).

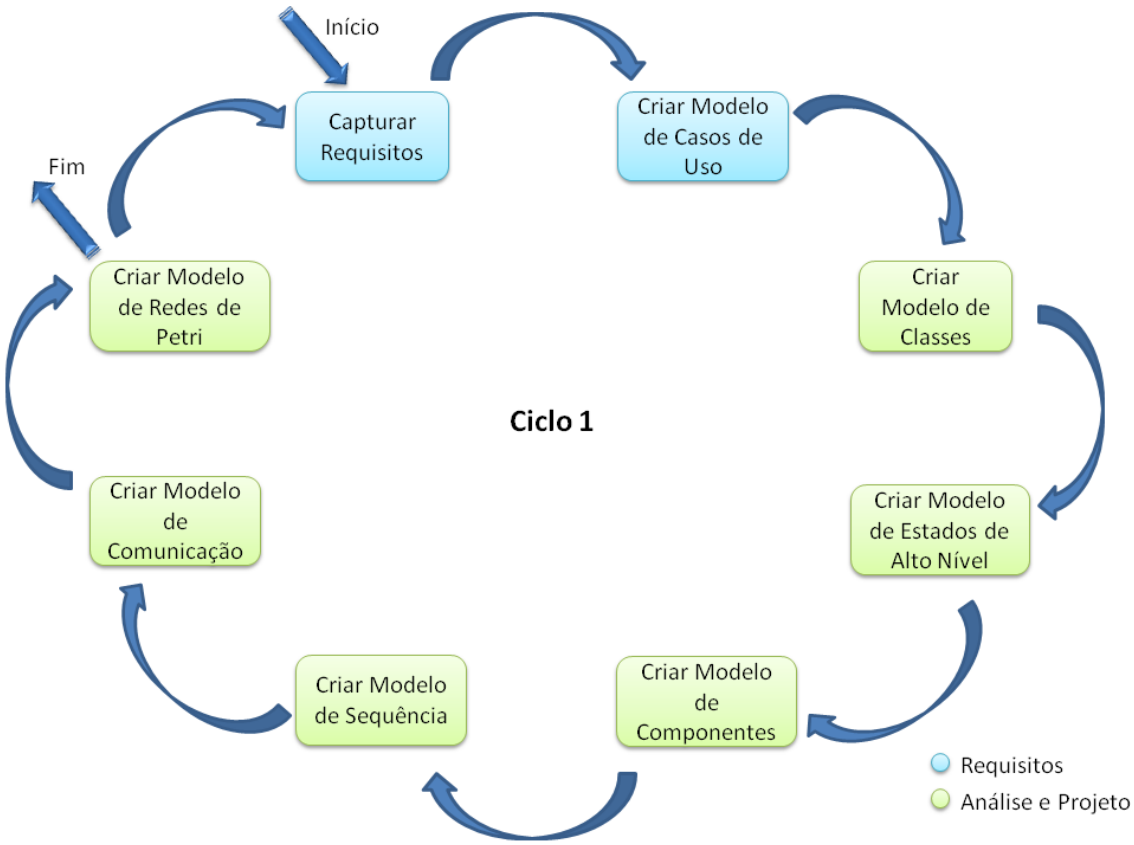


Figura 23 – Método DON (WIECHETECK, 2011)

Fazendo uso do *NOP Profile*, o DON guia os projetistas no desenvolvimento de projetos em PON por meio de etapas. Basicamente o método inicia-se com o levantamento dos requisitos e definição dos casos de usos no Modelo de Casos de Uso. Na sequência o projetista deve criar o Modelo de Estados de Alto Nível a fim de identificar as regras do *software*. Uma vez identificadas, as regras têm sua estrutura estática modelada, de forma inovadora, pelo Modelo de Componentes. Com esta

estrutura definida, as regras têm suas iterações modeladas nos Modelos de Sequência e de Comunicação. Por fim, a modelagem dinâmica das regras é modelada no Modelo de Redes de Petri, obtido a partir do mapeamento do Modelo de Componentes (WIECHETECK, 2011).

2.5.3 Reflexão

Através do método DON com a utilização do *NOP Profile* é possivel conceber os artefatos relacionados às fases de análise de requisitos e projetos do *software,* as quais compõem as fases de concepção e elaboração do modelo *RUP.* Neste âmbito, a ferramenta denominada *Wizard* PON se adequaria particularmente, a fase de construção do modelo *RUP*, o qual agregaria em produtividade na concepção de aplicações do PON.

2.6 *WIZARD* CON

Esta seção tem por objetivo explicar os serviços implementados para a composição da ferramenta denominada *Wizard* CON. Oportunamente, uma descrição sucinta a respeito da ferramenta *Wizard* CON foi descrita de antemão na seção 1.4.

2.6.1 Ambiente de Desenvolvimento de Controle

A ferramenta *Wizard* CON foi inicialmente concebida por (LUCCA, 2008) (LUCCA, 2009a) (LUCCA, 2009b) e posteriormente aprimorada nos trabalhos de (WITT, 2010a) (WITT, 2010b) e ainda aprimorado (apenas para fins de controle) no trabalho de (BENGHI, 2011). Sua finalidade é facilitar o desenvolvimento e a composição de instâncias de controle, que permite escrever regras e gerar automaticamente a partir delas os agentes *Rules* no âmbito do Controle Orientado a

Notificações (CON). Sua utilização está diretamente ligada à ferramenta de projeto e simulação conhecida como ANALYTICE II (WITT, 2010).

Ainda, nos trabalhos de LUCCA (2008) a ferramenta *Wizard* CON era um módulo interno ao simulador ANALYTICE II, no qual se criavam regras dinamicamente. As evoluções implementadas por WITT (2010a) desacoplaram a ferramenta *Wizard* CON da ferramenta ANALYTICE II. Desta forma, a comunicação entre elas era realizada através de arquivos *XML,* onde os *FBEs* eram exportados do simulador ANALITYCE II para o *Wizard* CON e por fim as regras eram exportadas do *Wizard* CON para o ANALYTICE II.

Neste âmbito, a ferramenta *Wizard* CON facilitaria o processo de criação de instâncias de controle que antes se dava tecnicamente por meio da linguagem de programação C++, ainda que apoiado em um *Framework* do CON (LUCCA, 2008) (LUCCA, 2010) (GÓES, *et. al.*, 2010). Este *Framework* foi criado por Simão (2005) servindo de “protótipo” para o *Framework* PON subsequentemente criado por Banaszewski (2009). Outrossim, o *Wizard* CON utiliza ambiente gráfico (*Windows Forms*) permitindo ao usuário compor regras no clássico formato *se-então*, em ambiente amigável.

2.6.2 Utilização da Ferramenta *Wizard* CON

Um primeiro passo para a utilização da ferramenta *Wizard* CON é a exportação dos elementos da base de fatos (*FBEs*) em um arquivo *XML*. Essa ação pode ser executada pela interface gráfica interna ao ANALYTICE II que é representada na Figura 24. Com isto, os *FBEs* podem ser importados pelo *Wizard* CON por meio de sua interface externa ao simulador.

Figura 24 – Exportação de *FBE* via ANALYTICE II

Outra interface importante que compõem o ambiente é apresentada na Figura 25, à esquerda. Todas as *Rules* criadas são listadas nela. Outra funcionalidade do *Wizard* CON é a visualização dos *Attributes*, o que é possível a partir da janela apresentada na Figura 25, à direita. A visualização dos *Attributes* significa listá-los, bem como disponibilizar todos os seus valores possíveis. Na verdade, também existe a possibilidade de alteração dos valores destes *Attributes*. O *Wizard* CON ainda permite registrar o conteúdo das *Rules* decorrentes em arquivos (*XML*), além de permitir que estas sejam excluídas, editadas ou recuperadas (LUCCA, 2008).

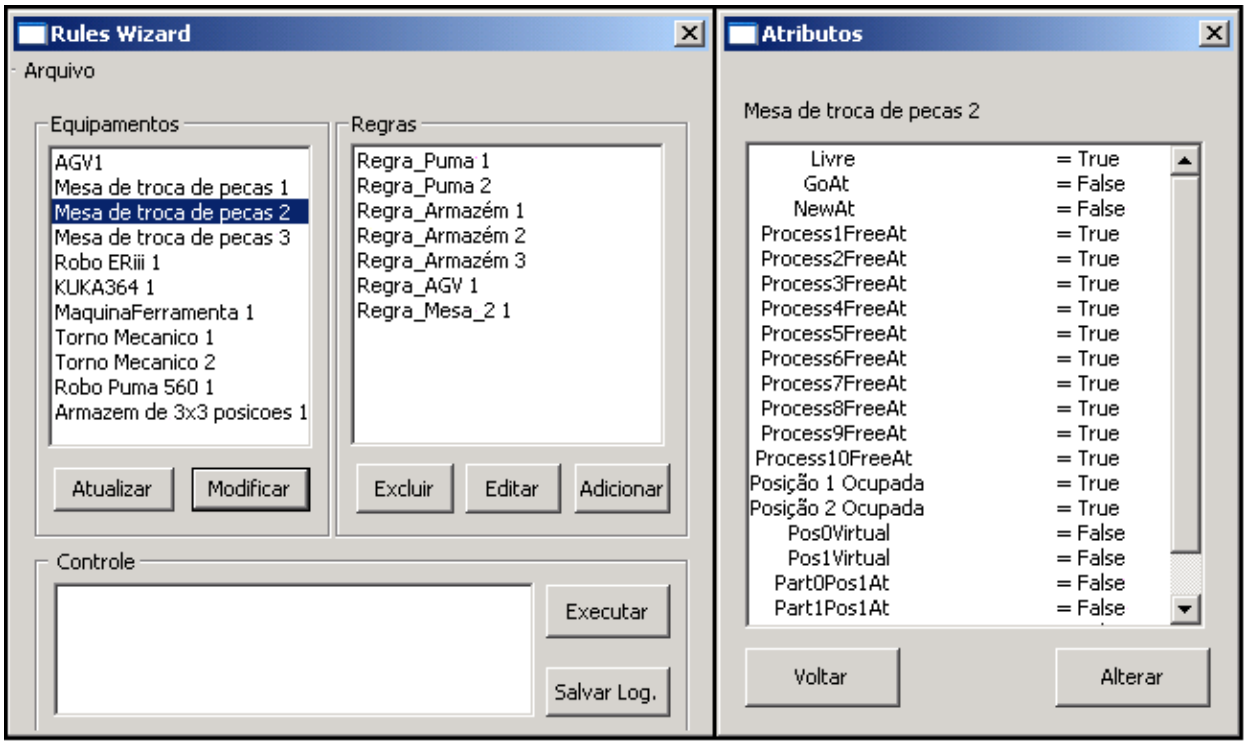


Figura 25 – Listagem de *Rules* e *Attributes* via *Wizard* CON (LUCCA, 2008).

Essencialmente, sua interface principal é apresentada na Figura 26. Neste âmbito, após a importação da base de fatos pela ferramenta *Wizard* CON é possível a criação de *Rules* no clássico formato *se-então*. No exemplo da figura é apresentada a criação de uma *Rule* que avalia o estado do *Attribute* “Livre” do *FBE* “CAEAGV1”. Caso este seja igual a “Verdadeiro” a *Rule* estará aprovada, sua ação será executada, e o método “*MoverMt*” do *FBE* “CAEAGV1” será ativado (WITT, 2010).

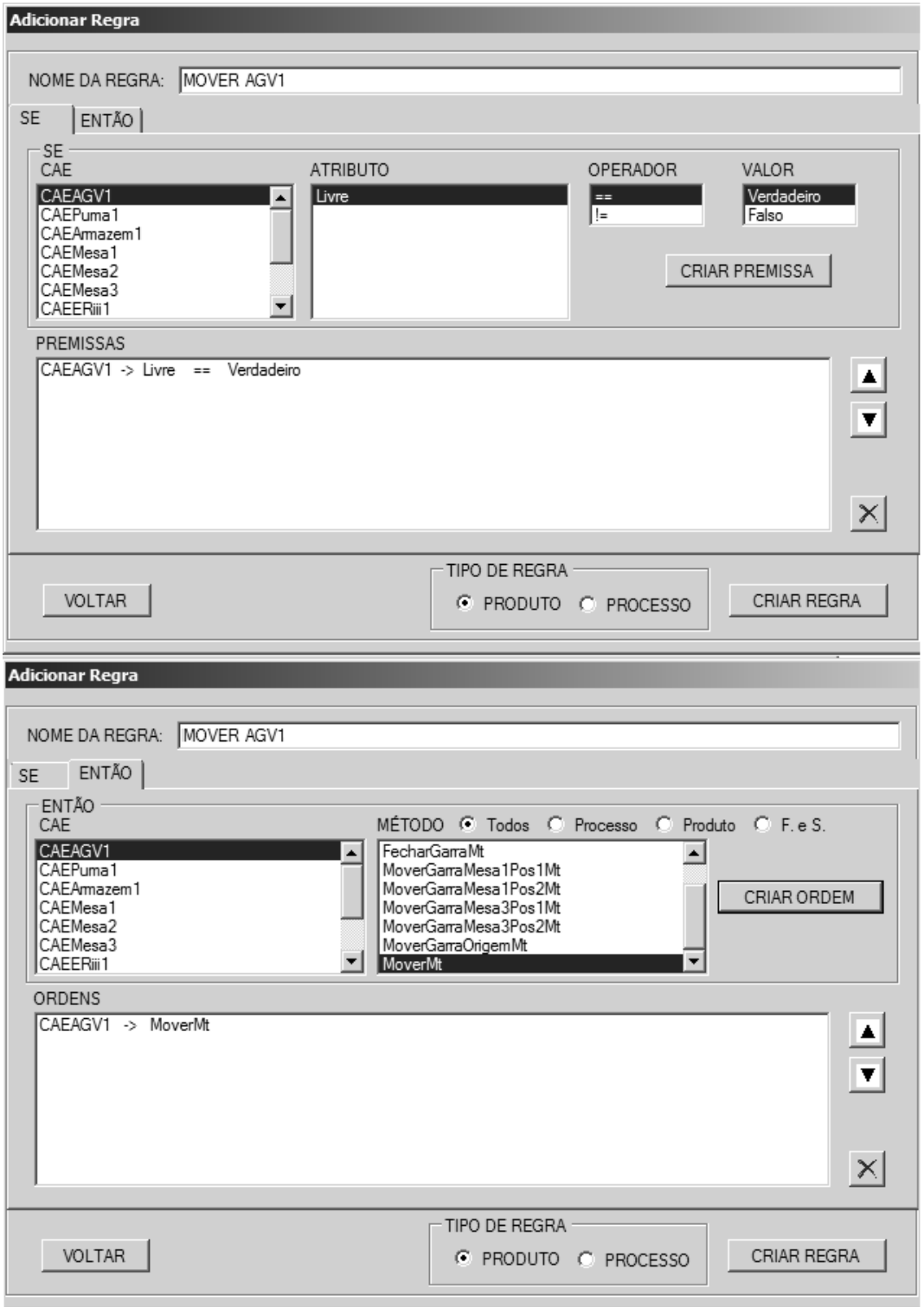


Figura 26 – Composição de Regras via *Wizard* CON (WITT, 2010)

Após a criação das *Rules*, esta ferramenta permite a exportação das informações de regras em arquivos *XML*, sendo estes passíveis de importação pela interface interna ao ANALYTICE II, cf. Figura 24. A título de exemplificação o

diagrama de casos de usos da Figura 27, apresenta os serviços implementados para a ferramenta *Wizard* CON.

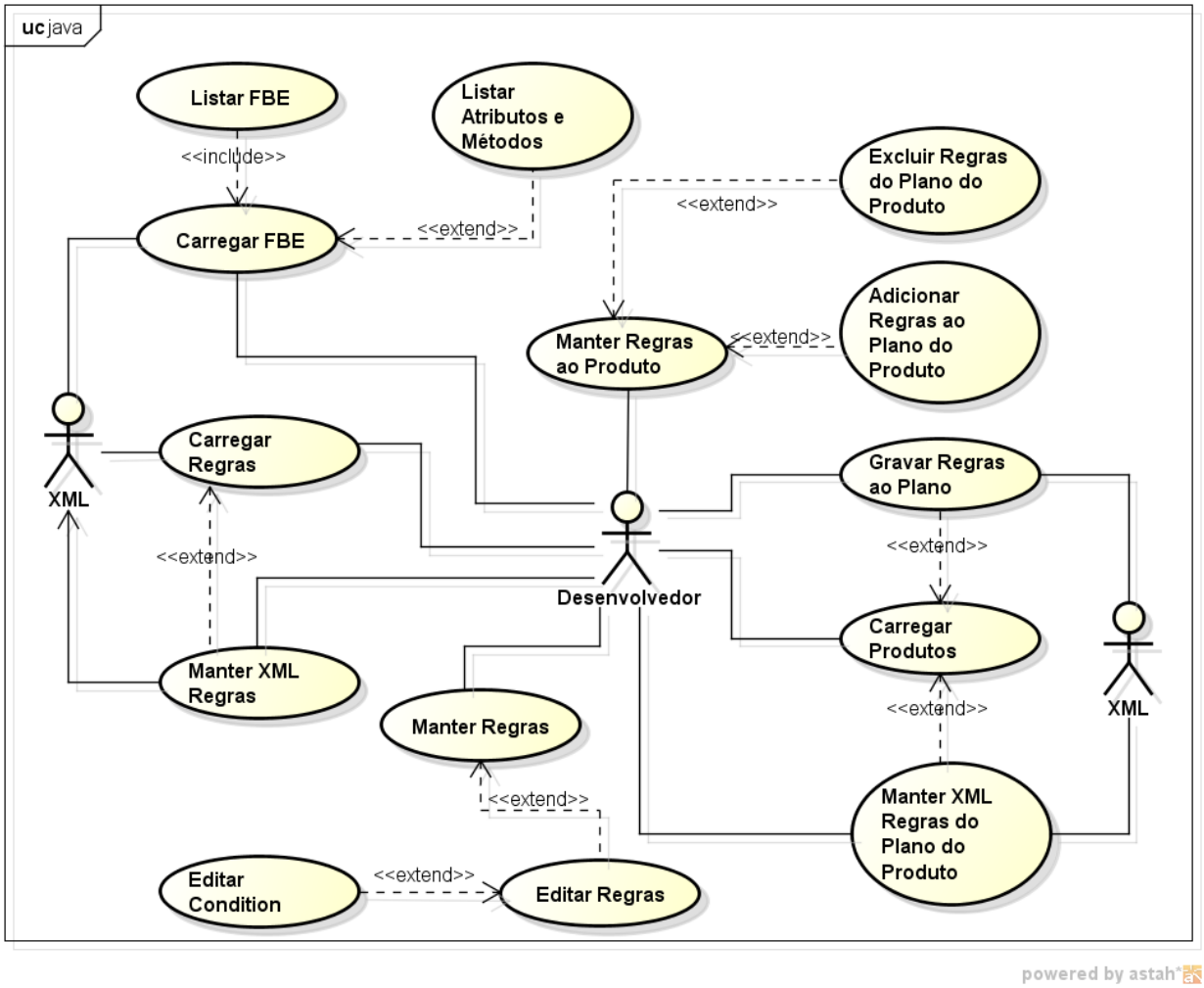


Figura 27 – Casos de uso da Ferramenta *Wizard* CON

É importante ressaltar os seguintes casos de uso: “Manter Regras ao Produto”, “Excluir Regras do Plano do Produto”, “Adicionar Regras ao Plano do Produto”, “Gravar Regras ao Plano” e “Carregar Produtos”, fazem parte do contexto da utilização sobre o simulador ANALYTICE II. Ainda, estes casos de uso foram concebidos na proposta de melhoria implementada nos trabalhos de WITT (2010). Ademais, fazem parte do controle orientado ao produto, o que por sua vez, não possui uma relação direta a este projeto de pesquisa e não serão abordados.

2.6.3 Reflexão

A ferramenta *Wizard* CON está associada a ferramenta ANALYTICE II no tocante aos *FBEs* e ainda gera *Rules* em formato *XML* para integrar a esta ferramenta. Neste âmbito, esta ferramenta necessitaria de evoluções para atender o requisito de geração de código PON sobre a nova versão de seu referido *Framework.* Assim, além da concepção de *Rules* em alto nível, seria possível gerar automaticamente o código fonte desta, o qual corresponderia parte da dinâmica de execução de uma aplicação PON propriamente dita.

Neste âmbito, o capítulo 4 descreve as evoluções implementadas para a respectiva ferrramenta bem como o seu estudo de caso. Ao final, são descritas as comparações qualitativas em relação à uma mesma implementação via ferramenta *Wizard* PON e via “manual”.

2.7 FERRAMENTAS DE GERAÇÃO DE CÓDIGO

Segundo (WHITEHEAD, 2007), (NAKAGAWA, 2006) e (PRESSMAN, 2006), a engenharia de *software* vem enfrentando problemas quanto à existência de métodos e ferramentas que auxiliem os engenheiros de *software* em todas as etapas de desenvolvimento. Atualmente é grande o número de ferramentas de apoio direcionadas à solução desses problemas. No entanto, tem-se observado que poucas atendem plenamente às necessidades dos desenvolvedores, principalmente na fase de construção do *software* propriamente dito. Quando atendem, existe o problema relacionado ao custo elevado, que inviabiliza sua aquisição por empresas de pequeno e médio porte (THOMAS, 2004).

Neste âmbito, surge a engenharia de *software* auxiliada por computador ou também conhecida como ferramentas CASE (*Computer Aided Software Engineering*), que podem ser tão simples como uma única ferramenta que suporta uma atividade em específico dentro do processo de desenvolvimento de *software* ou mesmo tão complexo como um completo “ambiente” o qual engloba ferramentas, banco de dados, pessoas, *hardware*, redes, sistemas operacionais e etc (PRESSMAN, 2006). Ferramentas *CASE* podem ser classificadas por função, por

seu papel como instrumento para os gestores ou pessoas técnicas, por sua utilização em várias etapas do processo da engenharia de *software* ou pelo apoio a arquitetura do ambiente (*hardware* e *software*) (PRESSMAN, 2006).

No escopo das ferramentas *CASE* para o processo de desenvolvimento de *software*, as ferramentas de automatização de código foram um grande marco na área de engenharia de *software*. Nos últimos anos, tem-se observado um aumento significativo no uso deste tipo de ferramenta no processo de desenvolvimento, devido aos interesses por automatização das atividades e redução em prazos e custo (HERRINGTON, 2003).

De acordo com (GRAY, 2000), as ferramentas de geração de código reduzem de maneira significativa o tempo e o custo de desenvolvimento em um projeto de *software*. O autor menciona a existência de várias ferramentas para propósitos específicos e a cooperação que deve existir entre elas na execução de um projeto. Em seu trabalho é possível encontrar uma breve classificação das ferramentas: aquelas direcionadas aos problemas específicos e aquelas mais abrangentes direcionadas a várias fases do ciclo de vida de um *software*.

Dessa forma, gerador de código é aquela ferramenta que possui a capacidade de gerar código a partir de um determinado modelo de *software*. Inclusive, de acordo com alguns pontos de vista e a partir das características específicas do tipo de gerador de código, ele passa a ser conversor de códigos de linguagens distintas. Isso acontece, por exemplo, em conversões de diagramas de classes em *UML* a partir dos quais se gera a parte estrutural de código em linguagens como *C++, Java e Dot Net* (HERRINGTON, 2003), linguagens estas normalmente pertencentes ao domínio da Programação Imperativa.

Tradicionalmente, muito do esforço gasto no desenvolvimento de soluções de *software* consiste na criação repetida de código. Como exemplo de tais situações, em um modelo tradicional de desenvolvimento de aplicações *Create Retrieve Update e Delete (CRUD),* a operação de acesso a camada de dados é geralmente a mesma, onde muito do que se escreve é igual, com mínimas variações. Ferramentas de apoio ao desenvolvimento no modelo chamado *Model View Controller* (MVC) tem se mostrado eficaz neste quesito (VELOCITY, 2012) (JPA, 2012) (JSF, 2012) (STRUTS, 2012).

Outrossim, um bom *design* pode auxiliar na redução de códigos repetitivos (FREEMAN *et al.*, 2004), porém nenhum paradigma atualmente vigente tem se

mostrado capaz de lidar com o problema de uma maneira geral e eficaz. Os problemas causados por este fenômeno vão muito além da óbvia queda de produtividade criada por tal repetição, mas também se manifestam em situações como (NOYES, 2006):

- **Incentivo a más práticas de programação:** fontes repetitivas são desinteressantes para programadores. Atitudes como o copiar e colar código se tornam padrão e com tal prática vários problemas em potencial (como a possibilidade de esquecer a necessidade de adaptar determinados passos executados) costumam ocorrer com grande frequência.
- **Diminuição da padronização:** enquanto é possível padronizar o comportamento externo de módulos de *software* através do uso de interfaces, o comportamento interno pode facilmente variar e provavelmente o fará com o passar do tempo, normalmente com trechos da aplicação apresentando comportamentos mais sofisticados conforme os desenvolvedores vão melhor dominando o uso da plataforma de desenvolvimento. Como consequência, torna-se difícil (senão impossível) prever como o programa final se comportará frente a mudanças.
- **Aumento de custo da manutenibilidade:** como consequência dos itens acima, é muito provável ter problemas de manutenção em projetos com alta repetição de código gerado manualmente. Ao refatorar dificilmente se pode assumir que os mesmos passos foram seguidos em cada módulo, o que possivelmente implicará em alterações manuais para cada trecho de código sensível à mudança proposta. Além disso, na ausência de uma boa suíte de testes, más práticas de programação podem facilmente levar a erros só descobertos quando executados por clientes.

2.7.1 Categorias de Geradores de Código

Geradores de código são ferramentas que a partir de um conjunto de instruções de entrada, oferecem como saída código fonte em uma ou mais linguagens de escolha do usuário (HERRINGTON, 2003). Algumas ferramentas podem se enquadrar neste conceito, como por exemplo, *IDEs* de desenvolvimento, ferramentas *CASE* e ainda, compiladores. De fato diversas bibliografias sobre

compiladores (AHO, 2001), costumam chamar a etapa final do processo de compilação (que se dá quando o código objeto será de fato escrito) de geração de código.

Porém, o termo em sua acepção mais comumente utilizada é referente a programas especializados em criar código fonte de alto nível para posterior compilação. Enquanto existem diversas maneiras de classificar estas aplicações, possivelmente a mais importante seja a que as separa conforme a generalidade de seu uso (HERRINGTON, 2003) e quanto à sua forma de operação. Neste âmbito, diversas categorias distintas se destacam (HERRINGTON, 2003):

- **Força bruta**: Basicamente, são aplicativos de geração que recebem um conjunto bastante restrito de dados de entrada e que emitem código específico em uma seleção bastante limitada de linguagens finais (DOLLARD, 2004). Como vantagem, esta abordagem costuma requerer um mínimo de treinamento para seu uso;
- **Baseadas em modelos**: frequentemente associadas à Arquitetura Baseada em Modelos ou *Model-Driven Architecture (MDA)*, estes geradores costumam receber como entrada um conjunto de diagramas que servirão de referência para a criação do código. Como principal vantagem, podemos destacar o fato de que o processo de criação de código também serve para a sua documentação, auxiliando na manutenção do projeto. O maior complicador reside no fato de o usuário precisar se acostumar com o uso de alguma linguagem de modelagem que pode, para soluções não triviais, tornar-se bastante complicado.
- **Baseadas em *Wizards***: costumam empregar “assistentes” de criação de código, isto é, possuem uma interface com o usuário bastante amigável que através de uma série de formulários de entrada de dados obtém as informações necessárias para a geração. Esta é mais uma categoria que costuma apresentar um fácil aprendizado e em contrapartida uma baixa flexibilidade. Exemplos: criação de *DataSets no Visual Studio, IronSpeed* (NOYES, 2006);
- **Baseadas em *templates***: normalmente menos formalistas que os baseados em modelos, mais flexíveis que os baseados em *Wizards* e em força bruta. Os geradores desta categoria recebem como entrada um ou mais *templates* – arquivos com informações de formatação misturadas a texto corrido. O modo de funcionamento de uma aplicação como esta se assemelha bastante às linguagens de *Scripting Server-Side* (*e.g*. *ASP* e *PHP*), onde basicamente se lê

um arquivo e o copia tal qual para a saída alterando seu comportamento sempre que encontra instruções especiais de formatação. Exemplos: *CodeSmith, MyGeneration, QCodeGenerator*;

- **Mistas**: existem geradores que misturam características de duas ou mais das categorias citadas acima. Exemplos: *Codify, Hibernate* e suas variantes;

Ferramentas para a geração de código vêm sendo usadas há bastante tempo no desenvolvimento de sistemas, mas até o momento não aconteceu a adoção de uma metodologia específica como padrão. Soluções *adhoc* têm sido comumente adotadas por causa de uma falta de formalismo ou padronização na área (BORK, *et. al.*, 2008).

2.7.2 Reflexão

Ferramentas de geração de código, de fato, auxiliam no processo de desenvolvimento de *software*. Entretanto, a maioria das ferramentas auxilia a geração de código em um primeiro *release* de *software*. Assim, depois de gerado, em uma aplicação do mundo real, o código é modificado pelo desenvolvedor. Desta forma, para manter o código modificado sinérgico ao modelo, a engenharia-reversa é necessária.

Todavia, a maioria das ferramentas de geração de código não realiza tal tarefa (GEIGER, *et. al.*, 2005). Alguns geradores (baseados em *templates*) utilizam-se de um analisador de linguagem comum (independente da linguagem de programação) para realizar o *parse* do código fornecido. Contudo, possuem deficiências graves de desempenho, bem como problemas de *parse* de sua gramática atribuída (BORK, *et. al.*, 2008).

Para a programação imperativa, especificamente do sub-paradigma de programação OO, as ferramentas *CASE* que dão suporte a *UML* geram código fonte em uma linguagem OO determinada (*e.g.* Java, C++, *Dot Net* etc). Entretanto, tais ferramentas não têm recursos de gerar o código dinâmico do *software.* Basicamente, a partir de alguns artefatos criados, especificamente o diagrama de classes, o código estrutural da aplicação é concebida automaticamente.

Essencialmente a estrutura das classes é concebida, através da concepção de seus atributos e métodos. Neste âmbito, esforços são realizados através da chamada Programação Dirigida por Modelos (ALMEIDA, 2008). Entretranto, ainda se faz necessária a implementação manual de boa parte da dinâmica da execução da aplicação. Isto é observado nos trabalhos de (ALMEIDA, 2008), onde uma simples aplicação do tipo *HelloWorld*, necessita de implementação interna, principalmente na camada de controle do padrão *MVC*. Ainda, boas ferramentas de suporte ao *MDA* não estão disponíveis de modo *open-source* ou mesmo *freeware*. Em verdade, não haveria ferramenta de desenvolvimento de *sofware* que permita desenvolver assaz facilmente a dinâmica de *software* de maneira geral.

Conforme esboçado na seção 1.4 e detalhado na seção 2.6, a ferramenta *Wizard* CON carece de melhorias e aperfeiçoamentos. O principal objetivo, para este trabalho de pesquisa, é evoluir suas funcionalidades de modo a propiciar a geração automática de código PON, em linguagem de programação C++, orientado ao seu novo *Framework.* Ainda seu uso poderá ser acoplado ao processo de desenvolvimento DON, descrito na seção 2.5, especificamente na fase de construção da aplicação PON. Por fim, salienta-se que a geração automática de *FBEs*, *Rules* e seus colaboradores (*Conditions*, *Premises* etc) e consequentemente da cadeia de notificações, significaria que considerável parte da dinâmica do *software* seria gerada automanticamente via ferramental *Wizard* PON.

2.8 REFATORAÇÃO DE CÓDIGO

Refatoração é o processo pelo qual os desenvolvedores re-estruturam o sistema sem mudar suas funcionalidades para remover duplicidades, melhorar a comunicação, simplificar ou adicionar flexibilidade, entre outros. Refatoração é o processo de melhorar a estrutura do código preservando sua funcionalidade original. Ainda, refatoração de código é a arte de alterar a estrutura interna do código fonte sem alterar seu comportamento externo. Cada transformação (chamada *refactoring*) isoladamente faz pouco, mas uma sequência de transformações pode produzir uma reestruturação significante de código (FOWLER, 2004).

Com a refatoração pode-se melhorar estrutura ruim ao redesenhá-la dentro de um código bem estruturado, onde cada passo é executado envolvendo atividades

como alterar um atributo de uma classe para outra, mover um trecho de código de um método para dentro de um novo e colocar um método acima ou abaixo de sua hierarquia atual. O efeito cumulativo destas pequenas alterações pode melhorar em muito a estrutura do código como um todo (FOWLER, 2004).

De acordo com as técnicas apresentadas por (FOWLER, 2004), a apresentação das refatorações seguem um conjunto de pequenas práticas. Dentre as principais razões pelas quais elas são aplicadas, destacam-se:

1. Melhorar o projeto de *software* como um todo. Assim, à medida que as refatorações ocorrem ao longo do tempo em um determinado projeto, o mesmo tende a se tornar mais legível e possuir maior facilidade de manutenção.
2. Melhorar o projeto pontualmente visando facilitar a localização de falhas à priori e a implementação do *software* propriamente dita. Desta forma, isto ajudará no processo de desenvolvimento do *software*, uma vez que a base para a implementação de um *software* com certa rapidez está diretamente ligada à construção prévia de um bom projeto.

A realização de pequenas mudanças no código, desconsiderando o projeto, para atingir objetivos em curto prazo, bem como mudanças feitas sem o conhecimento do projeto do *software*, resulta em uma desestruturação do código, tornando-o de mais difícil compreensão (FOWLER, 2004). Neste âmbito, (BECK, 2000) identifica ainda quatro outras situações que torna os programas difíceis de modificar, cf. apresentado na Tabela 1. Estas situações são indicativos explícitos que o código em questão e, portanto, possivelmente seu respectivo projeto necessitam de refatoração.

Tabela 1 – Partes que tornam os programas difíceis de trabalhar (BECK, 2000)

Situação	**Dificuldades de Modificação**
Baixa legibilidade	Dificulta o trabalho do desenvolvedor durante o desenvolvimento e manutenção.
Lógica duplicada	Difícil modificação, onde a alteração de uma das implementações impacta nas demais.
Necessidade de alterar códigos existentes para inclusão novas funcionalidades	Programas que para inclusão de novas funcionalidades, requerem a alteração de código existente.

Lógica condicional complexa	Quando existem muitas condições aninhadas dificultando o entendimento e a manutenção do código.

Geralmente, códigos mal desenvolvidos fazem as mesmas tarefas em diferentes lugares, ou seja, há uma duplicação de código. Um importante aspecto da aprimoração do projeto utilizando refatoração é justamente eliminar esses códigos duplicados, fato que normalmente não impacta no desempenho do programa, mas sim normalmente impacta positivamente neste desempenho e facilita ademais possíveis modificações futuras (FOWLER, 2002).

2.8.1 Aplicação da Técnica de Refatoração ao Processo de Desenvolvimento e Manutenção de *Software*

Segundo Fowler (2004), decidir quando começar a refatorar e quando parar é tão importante quanto a mecânica da técnica de refatoração. Explicar como proceder com a mecânica da refatoração propriamente dita é uma questão simples de aplicar e realizar, todavia tentar explicar quando se deve fazer, ou mesmo onde, não é algo tão consolidado ou mesmo trivial, principalmente em função do conhecimento do desenvolvedor sobre o projeto em si.

As indicações apresentadas na Tabela 2 por Fowler (2004) são problemas que podem ser resolvidos através da refatoração. Entretanto, cabe ao analista de sistemas, engenheiro de *software* e/ou mesmo ao desenvolvedor a própria percepção sobre quais problemas são pertinentes (*e.g.* quantas variáveis de instâncias são demais, quantas linhas de código em um método são excessivas, quais classes estão acoplada etc).

Tabela 2 – Quando se deve refatorar um *Software* (FOWLER, 2004).

Situação	**Momento de Refatorar**	**Justificativa**
Código duplicado	Quando existe o mesmo código em mais de um lugar, dificultando a manutenção e	Buscar um meio de unificá-los, melhora a legibilidade do código e o processo de manutenção.

	aumentando o número de falhas durante a manutenção do *software*.	
Método longo	Quando um método tem muito código.	Quanto maior for o método, mais difícil será de entendê-lo.
Classes grandes	Quando uma classe tem muitas variáveis de instâncias e possui mais de uma responsabilidade.	Tendência de alto acoplamento e baixa coesão no projeto de *software* como um todo.
Lista longa de parâmetros	Quando um método possui muitos parâmetros.	Dificulta o entendimento, porque se tornam inconsistentes e difíceis de usar, uma vez que alguma alteração sempre acontecerá a medida que precisar de mais dados.
Comentários	Quando existir comentários supérfluos	Comentários que estão no código sem explicar o porquê, são considerados supérfluos. O comentário deveria ser usado quando não sabe o que fazer e indicar áreas sobre os quais o desenvolvedor não está seguro. Ou ainda, um comentário deveria dizer o porquê fez algo e que tipo de informação ajudará os futuros modificadores do *software*.

2.8.2 Reflexão

O processo de refatoração de um *software* está fortemente ligado a sua manutenção e aprimoramento. Refatorar significa alterar o código-fonte propondo

melhorias na qualidade interna do *software*, tornando o código mais legível e coeso, possibilitando que desenvolvedores que participaram ou não de sua produção possam compreender sua estrutura de maneira mais simples e direta.

Especificamente sobre o *Framework* PON, o processo de refatoração deve ser muito bem pensado e planejado. À medida que cada objeto, método e/ou classe sofrer alteração, os mesmos deverão ser testados, verificados e validados. Isto é possível, pois, uma cópia original do *Framework* PON, escrito pelo trabalho de BANAZEWSKI (2009) é mantida inalterada. Assim, as aplicações PON escritas sobre a nova implementação do referido *Framework*, deverão ser executadas sobre estas duas versões. Neste âmbito, o comportamento das aplicações deverão ser exatamente as mesmas nos diferentes *Frameworks.*

Outrossim, é importante ressaltar que a versão do *Framework* PON elaborado por BANAZEWSKI (2009), é uma materialização dos trabalhos realizados e avançados particularmente por SIMÃO (SIMÃO, 2005) (SIMÃO, STADZISZ, 2008, 2009), conforme discutido na seção 1.2. Deste modo, o *Framework* PON vigente trata-se de uma versão consideravemente bem projetada e implementada, ao passo que torna o processo de refatoração e aplicações de suas técnicas uma atividade não trivial. Por fim, a re-engenharia e refatorações de código sobre a ferramenta *Wizard* PON se faz pertinente a este trabalho de pesquisa.

2.9 CONCLUSÃO

Este capítulo, inicialmente, teve por objetivo explanar sobre as dificuldades encontradas na programação imperativa e declarativa, principalmente no que tange ao desempenho das aplicações, redundância de processamento, e acoplamentos. Neste âmbito, surge a proposta do chamado Paradigma Orientado a Notificações (PON).

O PON propõe soluções efetivas para as deficiências apresentadas pelos paradigmas supracitados, conforme apresentado na seção 2.3. Ainda, de maneira detalhada, sua respectiva materialização sobre um *Framework* desenvolvido na linguagem de programação C++ foi apresentado. Entretanto, conforme discutido na seção 2.4.7, o mesmo pode ser aperfeiçoado, principalmente referente à estrutura de dados de alto nível, a qual comporta a cadeia de notiticações do PON.

Subsequentemente relatou-se o processo de Desenvolvimento Orientado a Notificações (DON), na seção 2.5. Este processo foi concebido pelos trabalhos de WIECHETECK (2011), o qual relata o ciclo de desenvolvimento de aplicações do PON com a utilização do chamado *NOP Profile* nas fases de concepção e elaboração do modelo *RUP.*

Neste âmbito, detalhou-se ainda a ferramenta *Wizard* CON (seção 2.6), a qual será evoluída neste trabalho de pesquisa. Isto se deve, primeiramente, para entrar em certa sintonia ao processo de desenvolvimento de aplicações PON, neste caso, especificamente a fase de construção do modelo *RUP*, o qual se baseia o método DON. Nesta fase, o desenvolvedor poderia conceber aplicações PON em alto nível através do apoio da ferramenta *Wizard* PON. Outrossim, seria possível obter o código fonte da estrutura bem como parte da dinâmica de execução da aplicação PON propriamente dita.

A respeito da composição de *software* em alto nível e posterior geração de código, a seção 2.7 abordou sobre o tema de geradores de código. De maneira sucinta as ferramentas de geração de código agregam no processo de desenvolvimento. Entretanto, estão ainda em um estágio da arte embrionária, uma vez que tais ferramentas não geram a dinâmica de execução da aplicação propriamente dita. Em se tratando de ferramentas que comportam a especificação da *UML,* sob o viés da programação imperativa, especificamente sobre o POO, tais ferramentas apenas geram o código estrutural da aplicação, baseadas principalmente em um respectivo diagrama de classes.

Esforços concebidos pela programação dirigida a modelos (do inglês *Model Driven Architecture - MDA*) apenas atenuam o problema da geração de código, principalmente sobre a camada de controle da arquitetura *MVC,* uma vez que tal camada corresponde justamente à dinâmica de execução da aplicação. Ainda, tais ferramentas, processos e/ou metodologias geralmente estão atreladas a arquiteturas pré-definidas, como por exemplo, a ferramenta *open source AndroMDA,* a qual relaciona-se à especificação *Java Entreprise Edition* (*JEE*).

Por fim, a seção 2.8, abordou o tema de refatoração de código. Isso se deve principalmente ao fato do *Framework* PON ter também evoluído em sua concepção arquitetural, *design* e desempenho. Para isso, é necessário o perfeito entendimento do *Framework* como um todo, além de posteriores validações das respectivas

refatorações. Uma vez que o objetivo de cada *refactoring* deve aprimorar a estrutura interna do código sem alterar seu comportamento externo.

3 MATERIAIS E MÉTODOS PARA ALTERAÇÕES NO *FRAMEWORK* PON

Este capítulo descreve as mudanças realizadas sobre o *Framework* PON, de forma tal a constituir um novo *Framework* PON. A seção 3.1 descreve as principais alterações relacionadas à estrutura de dados que comporta a cadeia de notificação do PON. As seções 3.2 e 3.3 exemplificam os modelos de implementações que comportam as diferentes estruturas de dados sobre o *Framework* PON. A seção 3.4, ao seu turno, apresenta o modelo de pacotes e suas responsabilidades implementadas para a nova materialização do *Framework* PON. As seções 3.5 e 3.6 detalham respectivamente as alterações no *Framework* PON sobre o viés de otimizações e refatorações. A seção 3.7 descreve as reflexões sobre as alterações realizadas sobre o *Framework* PON. Da seção 3.8 até a seção 3.12, são detalhados os caso de estudos utilizados para validar as evoluções propostas e implementadas para o *Framework* PON. Por fim, a seção 3.13 apresenta as conclusões sobre este capítulo.

3.1 CONTÊINERES DA CADEIA DE NOTIFICAÇÃO DO *FRAMEWORK* PON

De maneira geral, as deficiências relacionadas ao desempenho das aplicações desenvolvidas com o *Framework* PON original, instigaram otimizações e refatorações no código deste. Em tais otimizações e refatorações de código, a estrutura que comporta o ciclo de notificações referente ao diagrama de classes da Figura 12, foi a que sofreu maior impacto de alterações.

As próximas seções descrevem as evoluções sobre as estruturas de dados mencionadas. Inicialmente, a seção 3.1.1 esboça a estrutura implementada pelo *Framework* PON em sua versão original, composta por objetos de classes da *Standard Template Library* (*STL*). Da seção 3.1.2 em diante são detalhadas as novas implementações e abordagens realizadas para comportar a estrutura de dados da cadeia de notificações do *Framework* PON.

3.1.1 Materialização do PON via *STL – Standard Template Library*

Conforme exemplificado na seção 2.3.1, o núcleo da execução das aplicações PON está em sua forma ímpar de realizar o cálculo lógico causal, ou seja, da execução das expressões causais (“*if-then*”) em formato de *Rules*. Este processo é efetivado por uma cadeia de notificações pontuais entre os elementos PON. Tais elementos constituem-se de entidades que devem ser endereçadas e notificadas, de forma a orquestrar a cadeia de notificação, com o objetivo de realizar o cálculo lógico causal das aplicações do PON.

O cálculo lógico causal realizado pelo *Framework* PON proposto por BANASZEWSKI (2009), continha objetos de classe da *STL*, especificamente objetos das classes *list* e *vector,* para comportar o processo de notificação entre os elementos PON. O Algoritmo 16 demonstra sua utilização sobre a classe *Attribute*, através do método *notifyPremises()* e sua iteração sobre a lista de objetos *Premises.*

```
1  void Attribute::notifyPremises(){
2     Premise* premiseTmp = 0;
3     list<Premise*>::iterator it;
4     for (it = premisesList.begin(); it != premisesList.end(); ++it)
5     {
6           premiseTmp = ((Premise*)(*it));
7
8         if(premiseTmp->isExclusive()){
9              premiseTmp->notifyConditionsAboutPremiseExclusive();
10         }
11          else{
12             premiseTmp->notifyConditions();
13         }
14    }
15 }
```

Algoritmo 16 – *notifyPremises* (BANASZEWSKI, 2009)

O método *notifyPremises()* é um dos principais métodos dentro do escopo de implementação da cadeia de notificação do *Framework* PON. A princípio, em toda alteração de estado de um respectivo *Attribute,* o método *notifyPremises*() será invocado. Este método tem por responsabilidade percorrer a lista de *Premises* (*premiseList*) relacionadas ao *Attribute.* Cada *Premise* notificada, por sua vez, avalia seu estado lógico e, em caso de mudança, notifica sua(s) *Condition(s)* por meio da chamada ao método *“notifyConditions”* esboçado na linha 12.

Neste exemplo, a estrutura de dados que comporta os elementos PON, especificamente o elemento *Premise*, é formada por um objeto (*premiseList*) da classe *list* da *STL*. Assim, as iterações ocorrem conforme esboçado no Algoritmo 16. Inicialmente um iterador denominado "*it*" é declarado (linha 3). Este iterador é responsável pela navegação entre os elementos contidos na lista. Na linha 4 o iterador é inicializado na posição inicial da lista. Em seguida, a condição de parada é verificada (*it != premisesList.end()*). Por fim, a sobrecarga do operador (++) é utilizada para navegação do próximo elemento contido na lista (*premisesList*), até que o mesmo seja nulo.

Conforme esboçado pelo Algoritmo 16, todas as demais iterações sobre as entidades participantes da cadeia de notificação são realizadas conforme descrito de antemão. Ainda que contêineres da *STL* sejam flexíveis, bem testados, genéricos e universais, sua execução induz a uma sobrecarga de processamento computacional sobre as iterações realizadas nas entidades PON, conforme descrito por VALENÇA (2011) e detalhado na seção 2.4.6 deste presente trabalho. Sendo assim, as demais seções descrevem as novas estruturas de dados implementadas para o *Framework* PON.

3.1.2 *PONLIST*

Conforme exemplificado na seção 2.4.6, contêineres da *STL* são universais, flexíveis, bem testado e muito útil para muitas finalidades. No entanto, a *STL* é projetada para garantir generalidade e flexibilidade, enquanto a velocidade de execução, economia de memória, eficiência de utilização da *cache* de dados e o tamanho de código possuem baixa prioridade (FOG, 2011).

A otimização da estrutura que comporta as relações entre os objetos participantes do mecanismo de notificação é fundamental para o desempenho de aplicações que executam sobre o *Framework* PON. Assim, a primeira modificação em relação à estrutura de dados foi a implementação de um contêiner constituído por uma lista simplesmente encadeada unidirecional. O diagrama de classes da Figura 28 demonstra as classes implementadas para comportar a estrutura de dados denominada *PONLIST*.

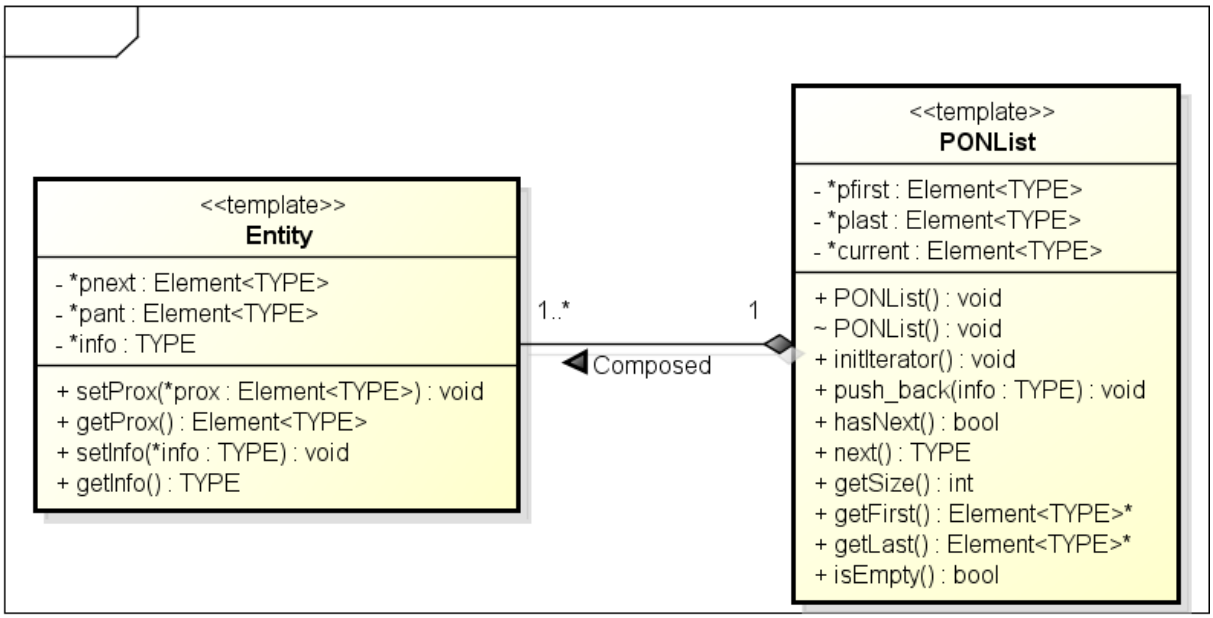


Figura 28 – Diagrama de Classes *PONLIST*

Basicamente a classe *PONLIST* possui os principais atributos e métodos relacionados à manipulação de uma estrutura de dados do tipo linear. Por exemplo, o método *hasNext()* é responsável por verificar se existe um próximo elemento na lista encadeada baseada em seu ponteiro corrente, ou seja, através da validação do atributo *current.* Ademais, o método *next()* obtém o elemento corrente da lista e navega para o próximo. Desta maneira, simples e enxuta, é possível realizar uma navegação unidirecional utilizando-se da estrutura de dados *PONLIST* do diagrama de classes da Figura 28. O Algoritmo 17 exemplifica sua utilização sobre o método *notifyPremises* da classe *Attribute*.

```
1  void PONLISTAttribute::notifyPremises() {
2
3        Premise* tmpPremise;
4        premisesList.initIterator();
5
6        while (premisesList.hasNext()) {
7              tmpPremise = premisesList.next();
8              if (tmpPremise->isExclusive())
9                tmpPremise->notifyConditionsAboutPremiseExclusive();
10             else
11               tmpPremise->notifyConditions();
12       }
13 }
```

Algoritmo 17 – *notifyPremises() PONLIST*

Similarmente ao Algoritmo 16, o Algoritmo 17 também é responsável por notificar todas as *Premises* relacionadas a um *Attribute.* Entretanto, diferentemente

de objetos *list* da *STL*, a iteração sobre uma lista que se utiliza da estrutura de dados *PONLIST* emprega somente os métodos realmente necessários para uma navegação unidirecional sobre a respectiva estrutura de dados. O processo de iteração inicia-se após a chamada do método *initIterator(),* responsável por iniciar o ponteiro corrente para o inicio da lista (linha 4). Logo após, o laço principal é validado através da chamada do método *hasNext()* (linha 6). Por fim, o método *next()* da linha 7 é invocado para recuperar o elemento corrente da iteração e navegar para o próximo.

A implementação da estrutura de dados denominada *PONLIST* valoriza a simplicidade de iteração sobre os elementos PON e não se utiliza de iteradores. Ainda *PONLIST* é formada por uma classe extremamente enxuta ao se comparar com as classes *list* e *vector* da *STL*. Isto se deve ao fato das iterações sobre as listas de elementos PON, em *PONLIST,* não necessitarem de operações como navegações bidirecionais, circulares ou particularmente operações de ordenações rebuscadas, como as encontradas em objetos da classe *list* ou *vector* da *STL*.

3.1.3 *PONVECTOR*

A utilização da estrutura denominada *PONVECTOR*, por sua vez, tem por objetivo principal realizar uma melhor utilização da memória disponível através da técnica denominada *memory pool*. Desta forma, o armazenamento dos elementos sobre a estrutura *PONVECTOR* é realizado de forma sequencial, através de um vetor de ponteiros para elementos PON. Neste âmbito, o acesso/iteração dos elementos PON é realizada também de maneira sequencial através da utilização de aritmética de ponteiros.

Deste modo, esta nova estrutura de dados implementada para o *Framework* PON, visa estabelecer um espaço prévio de alocação para os elementos que serão atribuídos a respectiva estrutura. A classe *PONVECTOR* esboçada pela Figura 29 representa a nova estrutura de dados desenvolvida para comportar os elementos da cadeia de notificação.

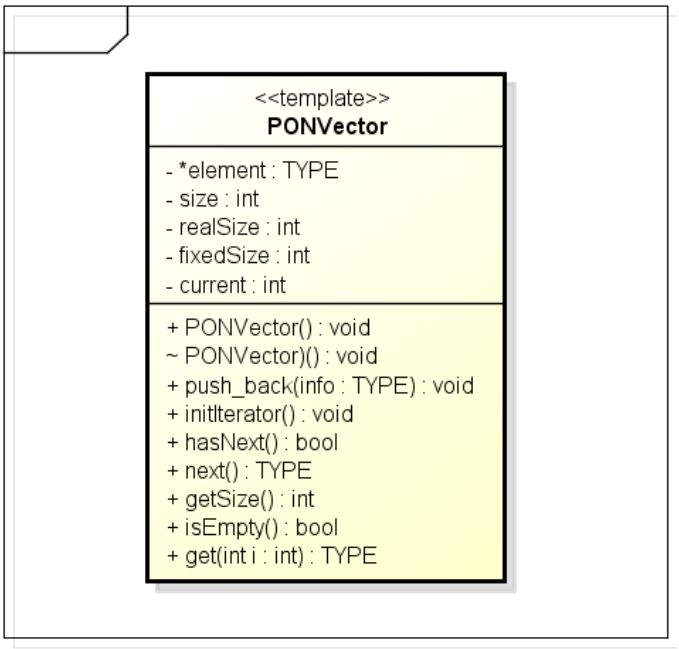


Figura 29 – Classe *PONVECTOR*

No momento da inicialização da classe *PONVECTOR*, um espaço em memória é reservado previamente. O tamanho do espaço é calculado conforme o tipo de elemento (*TYPE*) a ser armazenado na estrutura, a qual comportará inicialmente 10 (dez) posições. À medida que os elementos da cadeia de notificação são criados, estes são inseridos na respectiva posição informada do vetor. Assim o armazenamento dos elementos é realizado de forma contínua na memória, conforme observado na Figura 30 (a).

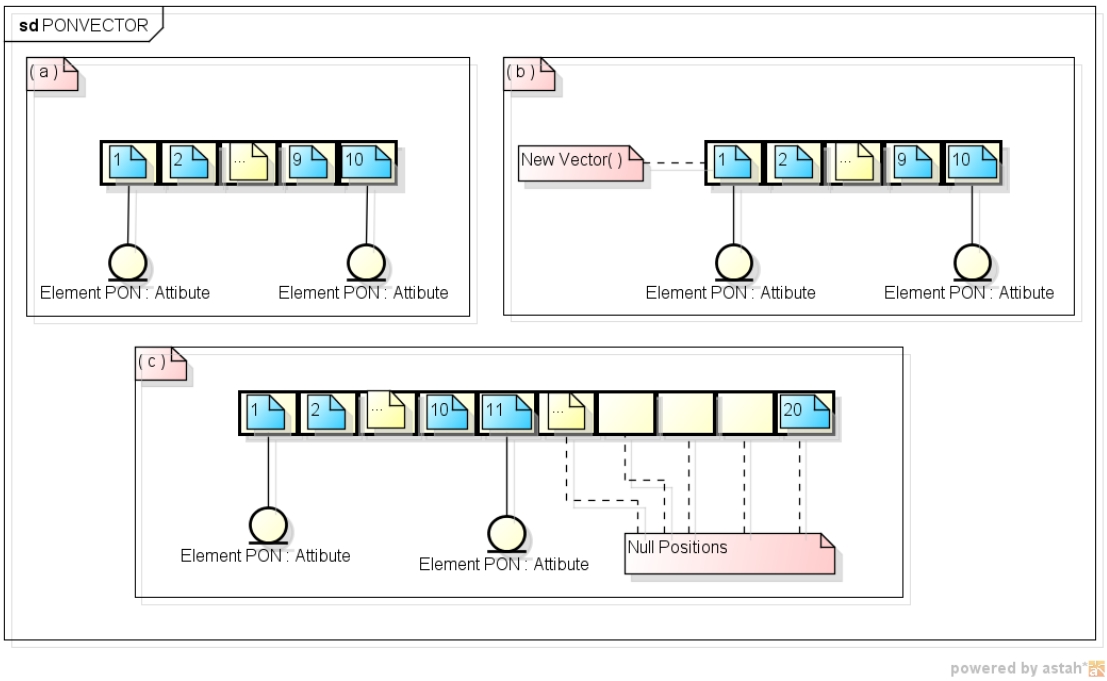


Figura 30 – Estrutura de memória com a utilização de *PONVECTOR*

Ademais, quando o limite máximo do tamanho do respectivo vetor é atingido, conforme Figura 30 (a), o algoritmo de realocação de memória é ativado, conforme Figura 30 (b). Neste caso, é executada a realocação de mais dez posições contínuas na memória, para serem utilizadas pelo respectivo elemento PON. Por fim, um novo vetor que comportará os novos elementos PON é criado, conforme Figura 30 (c).

É importante salientar que a iteração sobre os elementos PON sobre a estrutura de dados *PONVECTOR*, ocorre somente em regiões de memória que de fato esteja ocupada por um elemento PON. Desta forma, a condição de parada de uma iteração sobre os elementos PON, será sempre na primeira posição não ocupada por elementos PON.

De modo a exemplificar a funcionalidade descrita no parágrafo anterior, o Algoritmo 18 apresenta o método construtor da classe *PONVECTOR.* Na linha 7, o vetor de *elements* é instanciado, multiplicando-se por dez o respectivo tamanho do elemento (*TYPE*) a ser armazenado. Desta forma a estrutura de dados poderá comportar inicialmente até dez ponteiros de elementos PON.

```
1  template <class TYPE >
2  PONVECTOR< TYPE >::MyVector(){
3        size = 0;
4        realSize = 10;
5        fixedSize = 10;
6        current = 0;
7        element = (TYPE *) malloc (fixedSize * sizeof(TYPE));
8  };
```

Algoritmo 18 – Construtor da Classe *PONVECTOR*

Entretanto poderão existir casos em que dez posições no vetor não sejam o suficiente para o tamanho da estrutura. Assim, a partir do momento em que os elementos são inseridos, com a utilização do método *push_back()* o contador da estrutura será sempre incrementado. Conforme observado no Algoritmo 19, o respectivo contador (*size*) é incrementado a cada inserção realizada na estrutura de dados *PONVECTOR*.

No instante em que o atributo *size* for maior que o atributo *realSize,* que inicialmente continha o valor 10 (dez), uma nova área de memória será atribuída para comportar mais elementos PON. Uma cópia dos elementos é realizada para a nova área de memória. Por fim, este novo espaço em memória poderá comportar o novo elemento PON, bem como mais nove posições adicionais. Este procedimento é realizado a partir da chamada do método *realloc* (linha 8).

```
1  template <class TYPE>
2  void PONVECTOR< TYPE >::push_back(TYPE &info)
3  {
4   size++;
5   if (size > realSize)
6    {
7     realSize += fixedSize;
8     element = (TYPE *) realloc(elemento, sizeof(TYPE) * realSize);
9    }
10  element[size-1] = info;
11 };
```

Algoritmo 19 – Adicionar elemento PON na estrutura *PONVECTOR*

A iteração sobre a estrutura de dados denominada *PONVECTOR* utiliza-se da mesma interface (*i.e.* nome) de métodos implementadas sobre as estruturas *PONLIST* e objetos de classes (*list* e *vector*) da *STL*. Outrossim, uma peculiar funcionalidade é realizada pela estrutura *PONVECTOR*. Esta funcionalidade é o emprego de aritmética de ponteiros e suas vantagens em termos de desempenho para iterar sobre os elementos contidos na respectiva estrutura.

Uma vez que os elementos PON encontram-se posicionados sequencialmente em memória, a navegação sobre eles é realizado através do método *next()* do Algoritmo 20. Deste modo, ao tempo que se retorna um elemento da estrutura de dados, navega-se para o próximo elemento (linha 4). Isto permite o aproveitamento da *cache* de dados ser realizado de maneira eficiente. De fato, a utilização de aritmética de ponteiros mellhorou o desempenho da iteração de seus elementos, conforme demonstram os experimentos das seções 3.9 em diante.

```
1  template <class TYPE >
2  TYPE PONVECTOR< TYPE >::next()
3  {
4        return elemento[current++];
5  };
```

Algoritmo 20 – Método *next PONVECTOR* com Aritimética de Ponteiros

Conforme apresentado nas seções anteriores, o Algoritmo 21 apresenta a iteração sobre a estrutura *PONVECTOR* sob o viés do método *notifyPremises().*

```
1  void PONVECTORAttribute::notifyPremises() {
2
3        Premise* tmpPremise;
4        premisesList.initIterator();
5
6        while (premisesList.hasNext()) {
7              tmpPremise = premisesList.next();
8              if (tmpPremise->isExclusive())
9                tmpPremise->notifyConditionsAboutPremiseExclusive();
10             else
11               tmpPremise->notifyConditions();
12       }
13 }
```

Algoritmo 21 – *notifyPremisses() PONVECTOR*

A iteração apresentada pelo Algoritmo 21 possui semelhanças ao Algoritmo 17 utilizado em *PONLIST*. As duas diferenças encontram-se nos métodos *initIterator()* da linha 4 e o método *next()* da linha 7, de ambos algoritmos supracitados. No primeiro, o índice *current* da classe *PONVECTOR* é inicializado com o valor 0 (zero). O segundo, por sua vez, utiliza a técnica de aritmética de ponteiros para realizar a navegação entres os elementos PON contidos na estrutura *PONVECTOR*, fazendo assim o melhor uso da memória através da chamada *memory pool*. Esta nova implementação em *PONVECTOR* de fato agregou em desempenho em relação a estrutura *PONLIST*, conforme detalhado nos casos de estudos descritos da seção 3.9 em diante.

3.1.4 *PONHASH*

No *Framework* PON a essência de execução de um *Attribute* é detectar mudanças em seu estado e notificar um conjunto de *Premises* interessadas neste evento. A priori, as referências para estas *Premises* são reunidas em uma estrutura de dados do tipo lista encadeada. Desta forma, o *Attribute* percorre sequencialmente cada uma destas *Premises* a fim de notificar a sua mudança de estado. No entanto, esta prática se torna ineficiente quando o número de *Premises* a serem notificadas é muito alto. Esta ineficiência se deve porque grande parcela das *Premises* não é afetada por um determinado estado de um *Attribute* e mesmo assim são notificadas. Este fato é constatado no esquema (simplificado) à esquerda na Figura 31 (BANASZEWSKI, 2009).

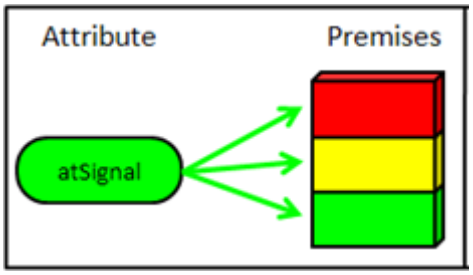


Figura 31 – Notificações baseadas em lista encadeada (BANASZEWSKI, 2009)

Nesta figura, o *Attribute atSignal* teve seu estado alterado de vermelho (*RED*) para verde (*GREEN*), devendo notificar apenas as *Premises* interessadas nestes dois estados para que atualizem os seus valores lógicos. Entretanto, devido ao fluxo de notificação guiado pela lista/vetor encadeada, a *Premise* que verifica se *atSignal* é amarelo (*YELLOW*) também é notificada, gerando uma avaliação lógica desnecessária. Neste simples exemplo, o efeito no desempenho é imperceptível, mas em exemplos que apresentam maior variedade de estados, como em avaliações numéricas, esta prática pode ser inaceitável (BANASZEWSKI, 2009).

Para evitar estas avaliações desnecessárias, foi empregado e desenvolvido um novo contêiner. Este contêiner se baseia na estrutura de dados da *STL* conhecida como *hashtable*. Desta forma com o emprego da estrutura denominada *PONHASH,* a chave corresponde ao estado ao qual a *Premise* compara um *Attribute*

e o valor corresponde ao próprio endereço da *Premise*. Com esta alteração estrutural, o *Attribute* notifica somente as *Premises* interessadas na mudança de seu estado, como ilustra o esquema da Figura 32.

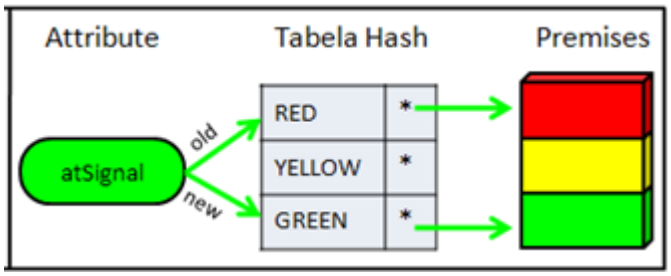


Figura 32 – Notificações baseadas em estrutura *hash*

Neste esquema, o *Attribute atSignal* notifica somente as duas *Premises* pertinentes, as quais se referem ao estado anterior e ao atual do respectivo *Attribute*. Com esta solução, a *Premise* que atesta o sinal amarelo não é notificada, poupando recursos de processamento. Portanto, a estrutura *PONHASH* foi empregada como alternativa para prover maior eficiência ao modelo proposto. O diagrama de classes da Figura 33 apresenta a nova estrutura de dados implementada para comportar o processo de notificação do *Framework* PON.

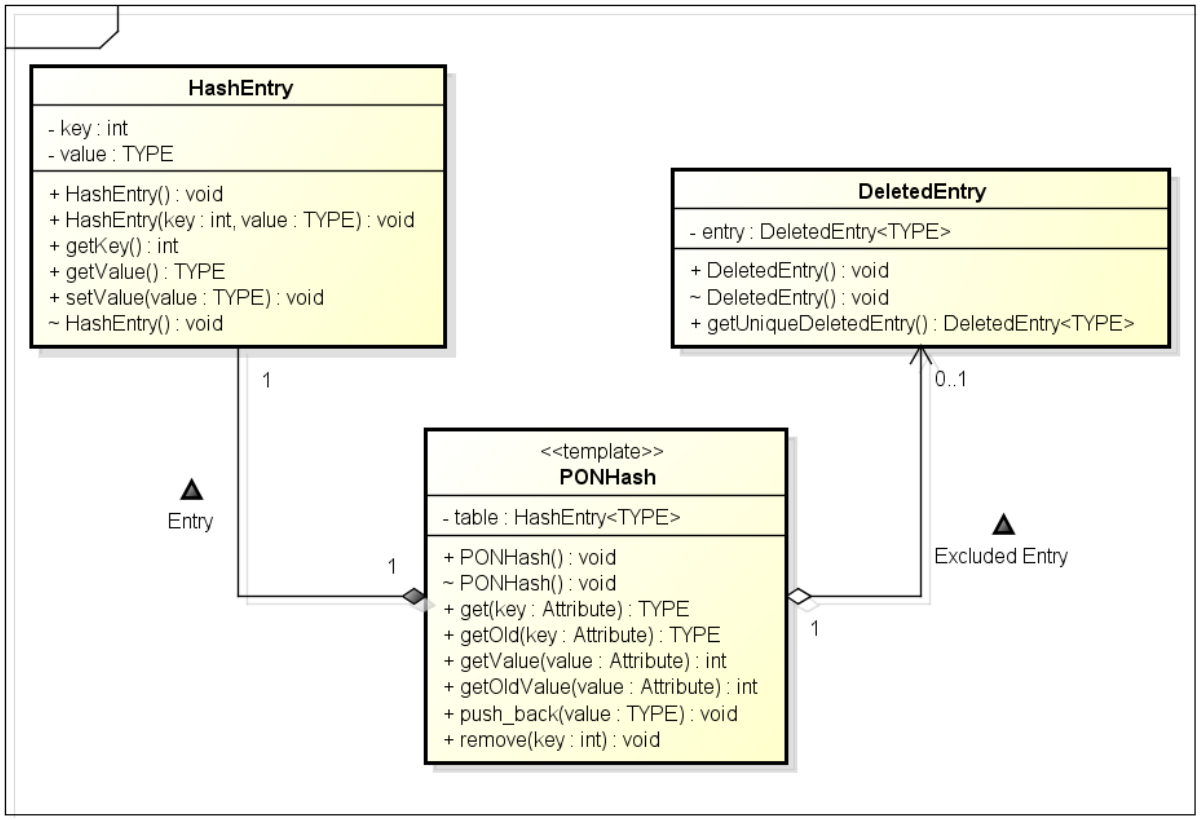


Figura 33 – Diagrama de Classes *PONHASH*

A implementação da estrutura *PONHASH* associa uma classe denominada *HashEntry,* o qual possui os atributos chave (*key*) e valor (*value*). A *key* da estrutura *PONHASH* é relacionado ao estado de um *Attribute* e o *value* é a referência ao endereço de sua *Premise.* A classe *DeletedEntry* será responsável por deletar uma respectiva entrada sobre a estrutura de dados *PONHASH*. Por fim, a classe *PONHash* corresponde à classe principal de manipulação e iteração sobre a estrutura de dados.

Conforme exemplificado para as demais estruturas, o Algoritmo 22 representa a iteração de elementos PON através da execução do método *notifyPremises(),* com a utilização da estrutura de dados *PONHASH*. Inicialmente a *Premise* é obtida através do método *get(this)* da linha 3. Conforme descrito de antemão, somente a *Premise* que está atrelada ao estado do *Attribute* é notificada. Desta forma, a *Premise* avalia seu estado lógico e notifica as condições a ela relacionadas (linhas 8 ou 10).

Entretanto, no processo de iteração do método *notifyPremises()* é necessário também validar o estado anterior do *Attribute*, caso o mesmo exista. Assim, o método *getOld(this)* é invocado na linha 4. Este processo é necessário para

desaprovar um estado anterior válido pelo *Attribute,* uma vez que a iteração *PONHASH* não percorre toda a lista, conforme executado pelas outras estruturas de dados mencionadas nas seções anteriores. Desta forma, caso o retorno do objeto *tmpPremiseAux* da linha 4 não for vazio, seu estado lógico anterior será desaprovado, assim será possível aprovar novamente uma *Rule* pertinente ao contexto da *Premise tmpPremiseAux* em questão.

```
1  void HashMapAttribute::notifyPremises() {
2  Premise* tmpPremise, *tmpPremiseAux;
3  tmpPremise = premisesList.get(this);
4  tmpPremiseAux = premisesList.getOld(this);
5    if (tmpPremise){
6       if (tmpPremise->isExclusive())
7        tmpPremise->notifyConditionsAboutPremiseExclusive();
8       else
9        tmpPremise->notifyConditions();
10   }
11   if (tmpPremiseAux){
12     if (tmpPremiseAux->isExclusive())
13       tmpPremiseAux->notifyConditionsAboutPremiseExclusive();
14     else
15       tmpPremiseAux->notifyConditions();
16   }
17 }
```

Algoritmo 22 – *notifyPremises* estrutura *PONHASH*

Conforme demonstrado na Figura 34, a iteração sobre uma estrutura *PONHASH* é realizada conforme o valor de um respectivo *Attribute.* Assim, no início do processo de notificação em que o estado de um *Attribute* é alterado, as *Premises* relacionadas ao *Attribute* deverão ser notificadas. Todavia com a utilização da estrutura *PONHASH,* somente a(s) *Premise(s)* que possuem em seu escopo o correspondente o valor do *Attribute* serão notificadas.

Entretanto, para que a notificação ocorra, é de antemão necessário realizar o cálculo da função *PONHASH,* através do valor do estado do respectivo *Attribute.* Assim, após o retorno da chave (*key*) da estrutura *PONHASH*, o acesso é realizado de forma “direta” a sua respectiva *Premise* com seu endereço obtido pelo atributo *value*. No exemplo da Figura 34, quando o estado de um *Attribute* (representado pela variação de temperatura) for alterado, a *Premise* correspondente será notificada somente quando o valor do *Attribute* for igual a uma respectiva condição associada à *Premise.*

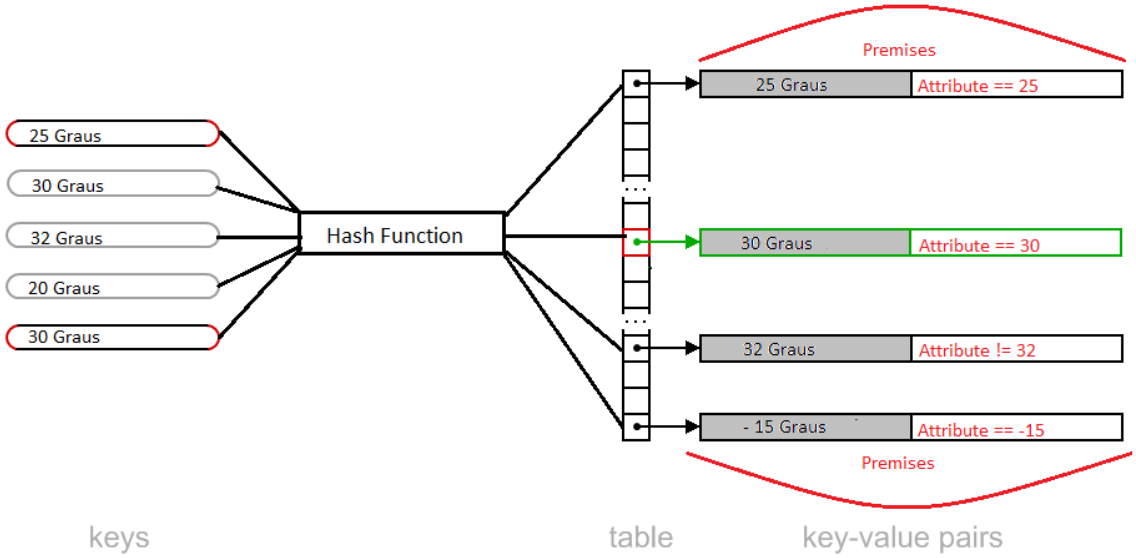


Figura 34 – Método de Notificação *PONHASH*

O Algoritmo 23 apresenta o código responsável pela realização do cálculo da função *PONHASH.* Internamente o método *getValue()* retorna o valor do respectivo *Attribute* passado como parâmetro para calcular a chave/*key* da estrutura *PONHASH.* Assim, na linha 2 do Algoritmo 23, o *Attribute* (*value*) é passado como parâmetro do método. Ainda, na linha 5, seu valor é verificado em função do seu respectivo tipo de *Attribute.* O algoritmo então calcula a função *hash*, baseado no estado deste *Attribute* e conforme seu tipo. Ademais, para cada tipo de *Attribute* é necessário convertê-los para os tipos de dados relacionados ao *Attribute* em questão, conforme observado, por exemplo, na linha 6, onde um valor booleano é convertido para inteiro. Uma vez calculado, o valor da chave/*key* é utilizada para realizar o acesso direto sobre a *Premise* correspondente ao estado do respectivo *Attribute*.

```
template<class TYPE>
int PONHash<TYPE>::getValue(Attribute *value) {
int key = 0;
switch (value->getAttributeType()) {
case Attribute::BOOLEAN_NOP:
  key = static_cast<int>(dynamic_cast<Boolean*>(value)->getValue());
  break;
case Attribute::INTEGER_NOP:
  key = static_cast<int>(dynamic_cast<Integer*>(value)->getValue());
  if (key < 0)
    key = key * (-1);
  break;
case Attribute::DOUBLE_NOP:
  key = static_cast<int>(dynamic_cast<Double*>(value)->getValue());
  if (key < 0)
    key = key * (-1);
  break;
case Attribute::CHAR_NOP:
  key = static_cast<int>(dynamic_cast<Char*>(value)->getValue());
  break;
}
return key;
}
```

Algoritmo 23 – *getValue PONHASH*

Ainda, é necessário realizar o tratamento de colisões. A estrutura *PONHASH* resolve através do método *push_back(),* esboçado no Algoritmo 24. Para tal, o procedimento de pesquisa linear *(linear probing)* é utilizado (DALE e NELL, 2003). Assim, no caso da posição da estrutura *PONHASH*, indicado pelo índice retornado da função *PONHASH*, já estiver ocupada, o algoritmo procura encontrar a próxima posição vazia na estrutura de dados. Ademais, problemas inerentes à pesquisa linear como buscas infinitas por espaços vazios são solucionados, por meio do próprio método mencionado através do Algoritmo 24.

```
while (hash != initialHash
       && (table[hash] ==
      DeletedEntry<TYPE>::getUniqueDeletedEntry()
      || (table[hash] != 0 && table[hash]->getKey() != key))) {
}
```

Algoritmo 24 – Pesquisa Linear *PONHASH*

A estrutura *PONHASH* foi implementada de maneira a realizar notificações pontuais, focando as entidades que apresentem o estado desejado igual ao estado atual da entidade em questão. Para isso, é necessário realizar de antemão o cálculo da função *hash* para direcionar as notificações apenas para as entidades interessadas. Ademais, as entidades que possuíam o estado, dito como anterior, são igualmente notificadas para terem seus estados lógicos revalidados.

3.2 ESTRUTURA GENÉRICA DO PROCESSO DE NOTIFICAÇÃO

Inicialmente, para que o *Framework* PON pudesse comportar as estruturas de dados mencionadas na seção 3.1, uma classe base denominada *Structure* foi implementada. Esta é uma classe abstrata que contém a definição virtualmente pura de métodos utilizados para comportar uma coleção de objetos (a serem notificados), sua manipulação e iteração. Ainda, neste âmbito, o padrão de projeto “Fábrica” ou *SimpleFactory* dos chamados e assaz universalizados *Gang of Four* (*GoF*) foi utilizado (GAMMA, HELM, JOHNSON e VLISSIDES, 1995).

Em suma, uma fábrica denominada *SimpleFactory* determina qual o tipo de estrutura deve ser utilizada em tempo de execução. Assim, conforme requerido pelo desenvolvedor, a determinada estrutura é informada a ‘fábrica’ para sua devida criação. A Figura 35 corresponde ao diagrama de classes implementado para comportar as novas estruturas do *Framework* PON.

A implementação *SimpleFactory* não é denominada necessariamente como um padrão de projeto, mas sim como uma definição de implementação (FREEMAN, ROBSON, BATES, SIERRA, 2004). Todavia ela é comumente utilizada, e tem sua importância vital neste tipo de implementação. Sua composição é importante ao ponto de poder agregar ao *Framework* PON a possibilidade de utilização de uma ou mais estruturas de dados, conforme a necessidade do desenvolvedor.

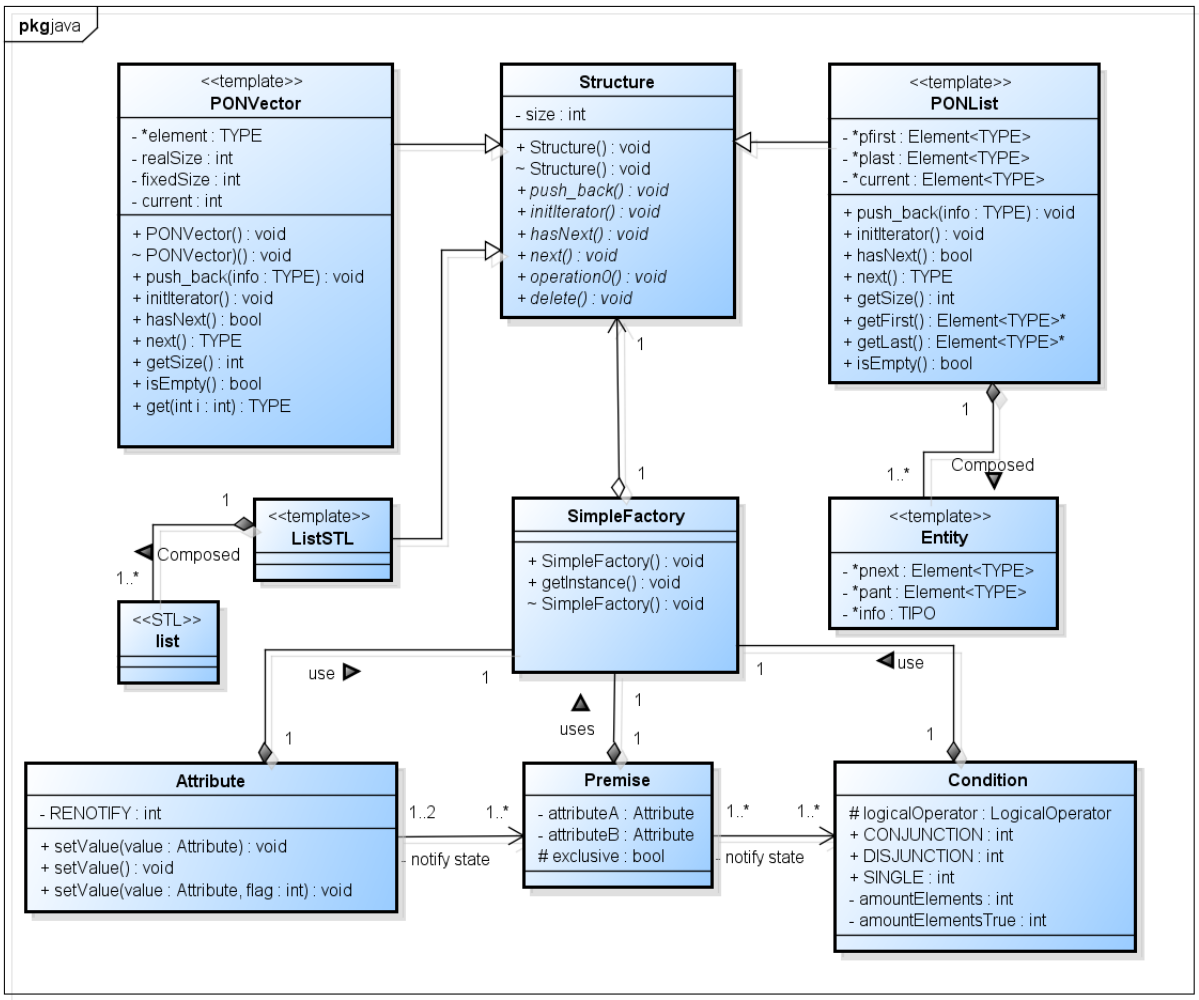


Figura 35 – Estrutura Genérica do Processo de Notiticação

Neste âmbito, a classe *Structure* define o modo de manipulação e iteração das estruturas de dados implementadas para *Framework* PON, especificamente as estruturas de objetos da classe *STL*, *PONLIST* e *PONVECTOR*. Ainda, para compor objetos de classe da *STL*, nomeadamente objetos da classe *list* da *STL*, foi implementado uma classe *template* denominada *ListSTL*, conforme observado no diagrama de classes da Figura 35. Neste âmbito, a iteração sobre elementos PON que se utilizam da estrutura *ListSTL*, é definida pela classe *Structure.* Outrossim, a classe *SimpleFactory* é responsável pela abstração da utilização das estruturas dados, uma vez definida sua utilização pelo desenvolvedor.

Deste modo, muito embora a relação explícita no diagrama de classes entre as classes *SimpleFactory* e *Structure,* na verdade ela representaria na prática as classes especializadas de *Structure*. Neste sentido, a execução e iteração das classes as quais representam as estruturas de dados seriam realizadas via polimorfismo.

Assim, a classe *Structure* define a manipulação de seus elementos através dos métodos *push_back*() e *delete(),* responsáveis, respectivamente, por adicionar e remover um elemento da estrutura de dados. Outrossim, os métodos *initIterator()*, *hasNext()* e *next()*, são utilizados para iterar sobre os elementos da estrutura. Desta forma, inicializa-se a iteração através da chamada do método *initIterator()*, verifica-se a existência de um elemento através do método *hasNext()* e, por fim, obtém-se e navega-se para o próximo elemento pela chamada do método *next()*.

3.3 ESTRUTURA ESPECIALIZADA DO PROCESSO DE NOTIFICAÇÃO

Não obstante, o diagrama de classes da Figura 35, sob o viés da classe *Structure,* de fato, resolva as questões de iterações e manipulações sobre as estruturas de dados, ela determina um fluxo padronizado para todas as demais estruturas que compõe o *Framework* PON. Desta maneira, toda e qualquer nova estrutura a ser implementada para o *Framework* PON, deverá seguir o padrão de manipulação e iteração de seus elementos baseadas na definição da classe base *Structure*.

Em se tratando particularmente da estrutura de dados *PONHASH,* detalhado na seção 3.1.4, a sua forma peculiar de realizar manipulações e principalmente sua maneira ímpar de iterar os elementos que a compõe, não está necessariamente relacionado a uma estrutura de classe genérica conforme esboçado na Figura 35.

Uma vez que a iteração sobre a estrutura *PONHASH* é realizada de maneira pontual, ou seja, diferentemente das iterações através de percorrimentos sobre as demais estruturas, o acesso a seus elementos é realizada de forma “direta”. Neste âmbito, uma nova abordagem para realização e utilização das estruturas de dados foi implementada. Esboçado na Figura 36, este modelo de classes permite uma maior flexibilidade em relação às operações e principalmente maneira de iterar os elementos constituintes de cada estrutura de dados.

Assim, cada estrutura especializa sua própria forma particular de realizar suas operações, bem como a maneira de iterar seus elementos constituintes. Para isso, uma fábrica denominada *ElementsFactory* é utilizada para definição de qual estrutura empregar para toda aplicação, sendo esta responsável pela criação de cada entidade que constituí o processo de notificação do *Framework* PON. Ainda,

neste âmbito, o padrão de projeto *Abstract Factory* foi utilizado (GAMMA, HELM, JOHNSON e VLISSIDES, 1995). O diagrama de classes da Figura 36 ilustra a aplicação desse padrão na estrutura do *Framework* PON.

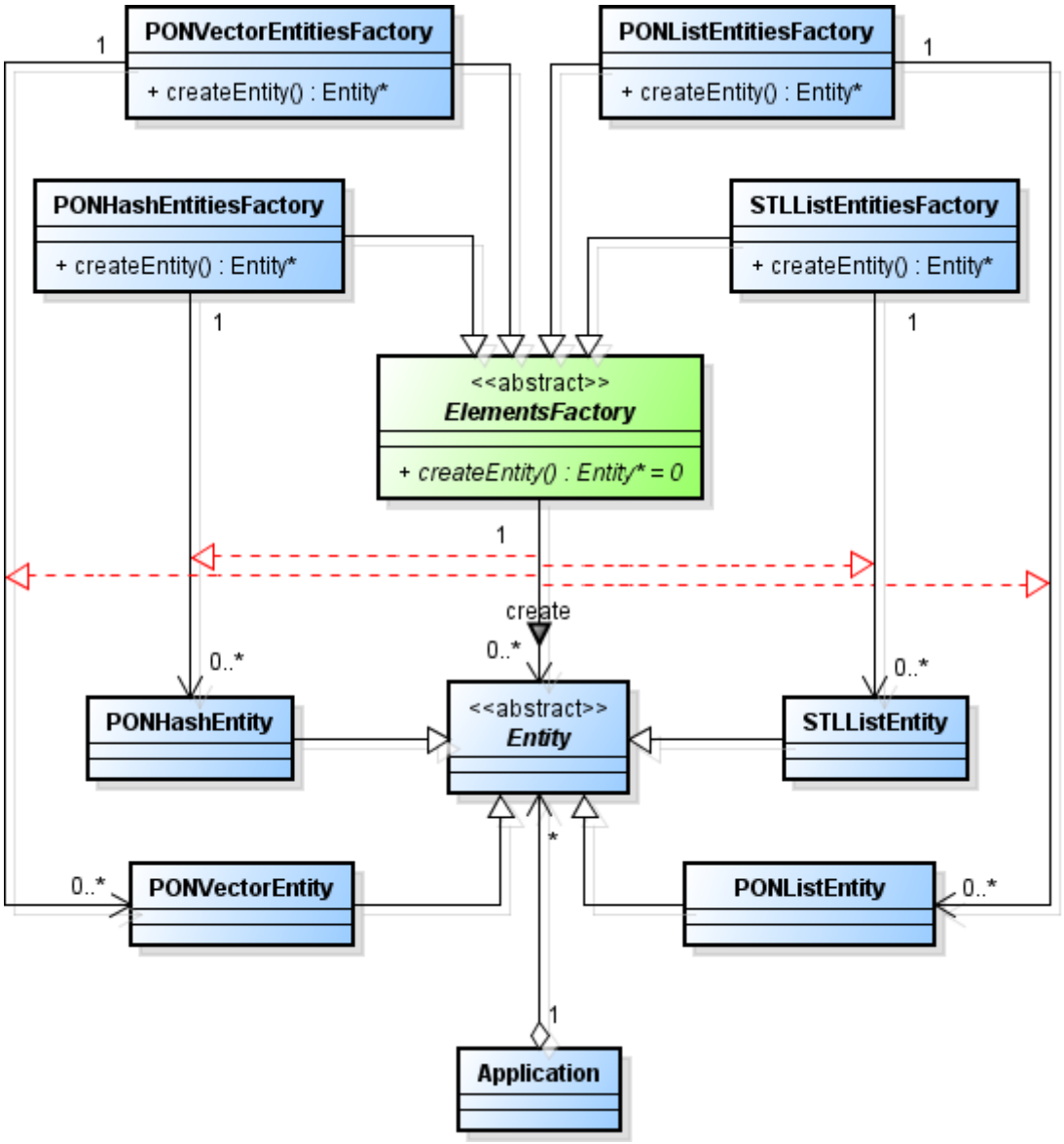


Figura 36 – Estrutura Especializada do Processo de Notiticação

Conforme apresenta a Figura 36, a classe principal desde modelo é a fábrica abstrata (*i.e. ElementsFactory*). Essa classe é responsável pela definição de todos os métodos abstratos responsáveis pela criação de entidades PON. O diagrama, em particular, abstrai a implementação real com a representação da classe abstrata *Entity*, a qual simboliza as entidades PON (*e.g. Attribute*, *Premise*, *Action etc*.).

Outrossim, o diagrama ilustra a presença de outras fábricas, ditas concretas, que tem por finalidade implementar os métodos herdados da fábrica abstrata. Tais

métodos têm por finalidade a criação de entidades PON concretas, as quais implementam internamente suas respectivas particularidades para cada estrutura de dados existente. Ainda, tal abordagem facilita a criação de eventuais novas estruturas de dados que possam vir a ser implementadas em trabalhos futuros.

O exemplo do Algoritmo 25, demonstra a forma de utilização e instanciação da fábrica PON sobre o método *initStartApplicationComponentes.* Este método é responsável por inicializar a fábrica PON, definindo principalmente qual estrutura de dados utilizar para o processo de notificação do *Framework* PON. Sua instância é obtida através do método *getInstance*() da classe *SingletonFactory.* Por fim, o método *startApplication()* é invocado, iniciando neste momento a execução da aplicação propriamente dita. Oportunamente, o exemplo esboçado pelo Algoritmo 25, instancia a estrutura de dados *PONHASH* (linha 2).

```
1  void Marksmanship::initStartApplicationComponents() {
2    SingletonFactory::changeStructure(SingletonFactory::PONHASHMAP);
3    elementsFactory = SingletonFactory::getInstance();
4    this->startApplication();
5  }
```

Algoritmo 25 – Inicialização da Fábrica PON

Neste âmbito, após definir a estrutura de dados, o desenvolvedor irá obter uma instância da fábrica PON através do método *getInstance()* do algoritmo da Figura 37. Deste modo, todo e qualquer elemento PON será atribuído a respectiva estrutura de dados instanciada pelo método *getInstance()*. O Algoritmo 25 inicialmente define a estrutura *PONHASH*, neste caso o objeto *elementsFactory* do algoritmo da Figura 37 instanciará a classe *NOPHashElementsFactory()* (linha 11).

```
1  ElementsFactory* SingletonFactory::getInstance() {
2
3      switch (elementsFactory) {
4          case SingletonFactory::PONLIST:
5              elementsFactory = new NOPListElementsFactory();
6              break;
7          case SingletonFactory::PONVECTOR:
8              elementsFactory = new NOPVectorElementsFactory();
9              break;
10         case SingletonFactory::PONHASHMAP:
11             elementsFactory = new NOPHashMapElementsFactory();
12             break;
13         case SingletonFactory::PONLISTSTL:
14             elementsFactory = new ListSTLElementsFactory();
15             break;
16     }
17     return elementsFactory;
18 }
```

Figura 37 – *Singleton getInstance()*

Assim, para a criação de um elemento PON, a classe utilizada será a classe *NOPHashMapElementsFactory(),* conforme ilustra a Figura 38. Desta forma, os elementos PON respectivos a cada estrutura, derivam de suas classes concretas (*Attribute, Premise, Conditon, Action, Instigation etc*) para classes base (*NOPHashMapAttribute*, *NOPHashMapPremise, NOPHashMapConditon, NOPHashMapAction, NOPHashMapInstigation etc*) da respectiva estrutura *PONHASH*. Isto, de fato, flexibiliza as operações de manipulações e iterações dos elementos PON, uma vez que cada estrutura poderá definir sua forma particular de realizar tais operações.

```
class NOPHashMapElementsFactory: public ElementsFactory {

    public:

        NOPHashMapElementsFactory();
        virtual ~NOPHashMapElementsFactory();

        Boolean* createBoolean(FBE* fact, bool value,
                               int status = Attribute::ACTIVE);
        Char* createChar(FBE* fact, char value,
                               int status = Attribute::ACTIVE);
        Double* createDouble(FBE* fact, double value,
                               int status = Attribute::ACTIVE);

        Premise* createPremise(Attribute* attributeA, Attribute* attributeB,
                               int logicalOperator, bool exclusive = false);
        Premise* createPremise(Attribute* attributeA, bool atBoolean,
                               int logicalOperator, bool exclusive = false);
        Premise* createPremise(Attribute* attributeA, int atInteger,
                               int logicalOperator, bool exclusive = false);

        Condition* createCondition(LogicalOperator* logicalOperator);
        Condition* createCondition(int logicalOperator);

        SubCondition* createSubCondition(LogicalOperator* logicalOperator, bool exclusive);
        SubCondition* createSubCondition(int logicalOperator, bool exclusive);

        RuleObject* createRuleObject(string name, Scheduler* scheduler, int logicalOperator);

        Action* createAction();

};
```

Figura 38 – *NOPHahsMapElementsFactory*

A vantagem nesta concepção que especializa os elementos PON, é o fato de amenizar o processo de iterações polimórficas e utilização de métodos virtuais. Uma vez que as classes derivadas realizam as operações de manipulação e iteração sobre os elementos PON sem necessitarem de uma classe base do tipo *Structure*, a qual define tais operações que são executadas via polimorfismo. No momento de sua execução, a iteração ocorre diretamente sobre os métodos das classes derivadas correspondente a cada estrutura de dados, evitando este nível de execução polimórfica no processo de notificação.

Por fim, a alocação dinâmica de memória sobre os elementos PON é realizada de forma controlada, ou seja, através da utilização da fábrica PON. Neste contexto, toda e qualquer criação de elementos PON, passa pela fábrica PON, conforme observado no diagrama de classes da Figura 36. Assim a instanciação de elementos PON não é realizada indiscriminadamente em seu *Framework* como em sua versão predecessora.

Especificamente, este problema é minimizado através da utilização do padrão de projeto *Abstract Factory,* que permite a criação de elementos PON através do uso de uma interface abstrata. Conforme demonstra a Figura 36, a classe principal desde padrão é a fábrica abstrata (*i.e. ElementsFactory*). Essa classe é responsável pela definição de todos os métodos abstratos responsáveis pela criação de entidades PON.

3.4 PACOTES DO *FRAMEWORK* PON

Pacote é um mecanismo de agrupamento, onde todos os modelos de elementos podem ser agrupados. Em *UML*, um pacote é definido como: "Um mecanismo de propósito geral para organizar elementos semanticamente relacionados em grupos". Desta forma, para atender as novas responsabilidades das classes implementadas para o *Framework* PON e ainda, agrupá-las de modo a facilitar seu entendimento e compreensão, uma nova abordagem de pacotes foi proposta. Esta estrutura visa atender principalmente os diferentes contâineres implementados para o fluxo de notificação do PON. O seguinte pacote em *UML* corresponde ao modelo implementado conforme a Figura 39.

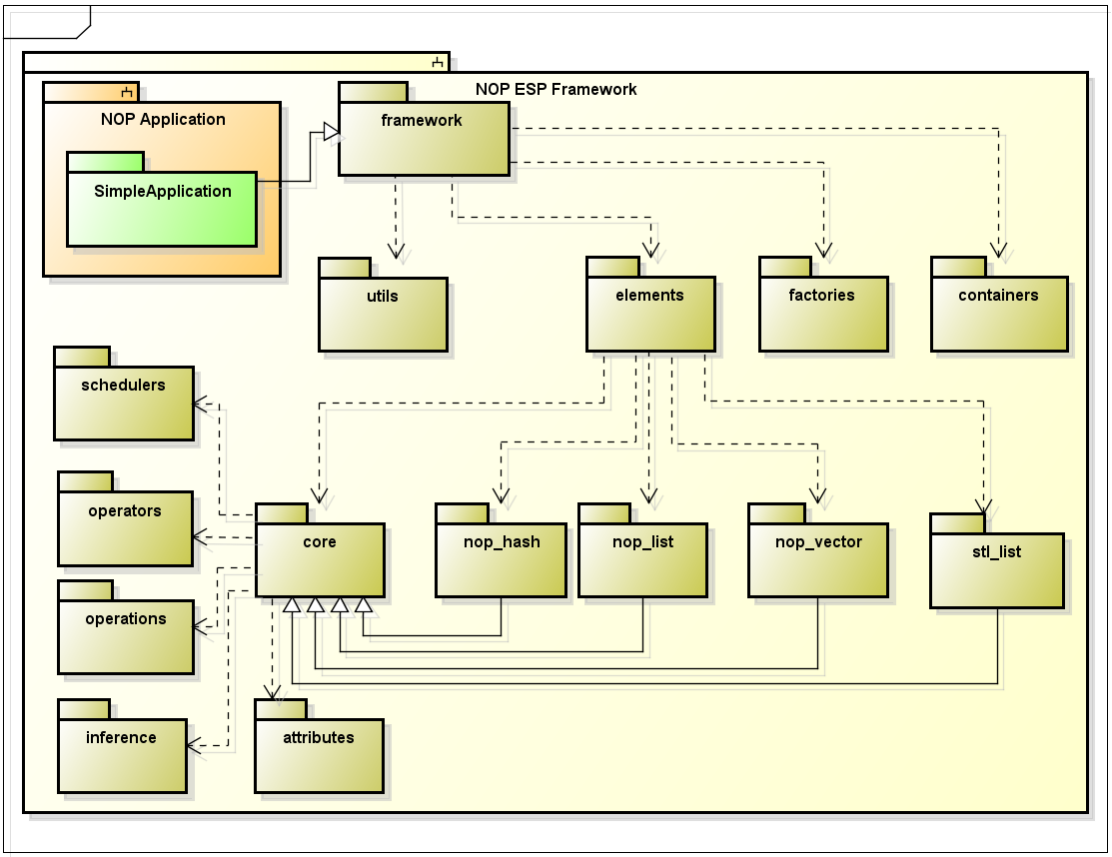


Figura 39 – *Framework* PON sob o viés de estrutura de pacotes

O pacote *framework* é o pacote unificador do *Framework* PON. Seus pacotes bases são formados pelos subpacotes *utils, elements*, *factories* e *containers*. O pacote *utils* possui as classes utilizadas para fornecer funcionalidades extras ao desenvolvedor do PON[2]. O pacote *elements*, ao seu turno, encapsula os subpacotes (*nop_hash, nop_list, nop_vector e stl_list*) responsáveis por compor as classes de instanciação de cada *container* criado para o *Framework* PON. Subsequentemente o pacote *factories* agrega em seu escopo as classes responsáveis pela criação da fábrica do *Framework* PON. Ademais, o pacote *containers* contém as implementações concretas de cada *container* do *Framework* PON.

O pacote *core* da Figura 39, por sua vez, possui os subpacotes correspondentes ao processo de realização do cálculo lógico causal do PON. Neste âmbito, o pacote *schedulers* contém as classes responsáveis pelo escalonamento de execução da *Rules* no PON. Subsequentemente, o pacote *operators* contém as

[2] Tais funcionalidades foram implementadas pelos trabalhos em andamento a serem apresentadas na dissertação de Adriano Francisco Ronsckza (RONSCKZA, 2012).

classes que realizam as operações lógicas para uma respectiva entidade *Condition.* Ainda, o pacote *operation*s, ao seu turno, encapsula as classes responsáveis por realizar as operações aritméticas no PON. Por fim, o pacote *inference*, é responsável por encapsular as classes referentes ao processo de inferência realizado para o cálculo lógico causal do PON.

As classes pertencentes à cadeia de notificação são instanciadas em relação ao tipo de estrutura dados (*container*) definidos pela fábrica de criação de elementos PON, contidos no pacote *factories,* conforme esboçado pelo diagrama de classes da Figura 40. Em relação ao pacote *attributes,* suas referidas classes também são instanciadas a partir da definição da estrutura de dados a ser utilizada pela aplicação PON. Oportunamente, o pacote *attributes* contém as classes que encapsulam os tipos primitivos da Orientação a Objetos (*i.e. Integer, Double, Boolean, Char* e *String*).

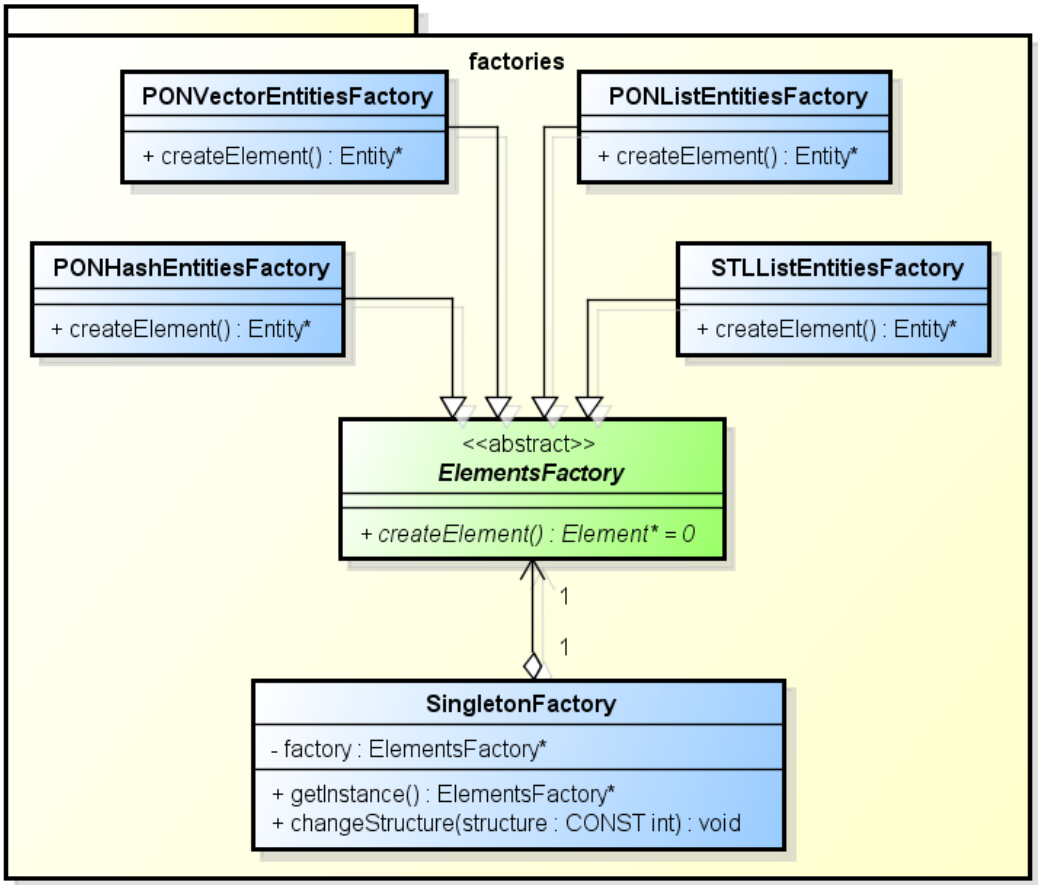


Figura 40 – Diagrama de classes referente às estruturas de dados

Toda e qualquer instanciação de elementos PON é definida através da fábrica PON, representada pelo pacote *factories e* esboçada no diagrama de classes da Figura 40. Outrossim, a definição da classe *SingletonFactory* é responsável por validar somente uma única instância de criação de elementos PON. Desta forma, toda e qualquer criação de elementos PON é realizada através da instância obtida por meio do método *getInstance()* da classe *SingletonFactory.*

Basicamente as classes que possuem manipulação (adição, remoção) e iteração (percorrimento) de elementos PON são especializadas para realizarem estas operações em suas respectivas estruturas de dados. Com isso, os métodos responsáveis por realizar a manipulação/iteração dos elementos nas estruturas de dados são sobrescritos e implementados em suas classes derivadas. De modo a exemplificar a especialização mencionada, a Figura 41 demonstra a devida especialização de cada pacote correspondente às estruturas de dados implementadas.

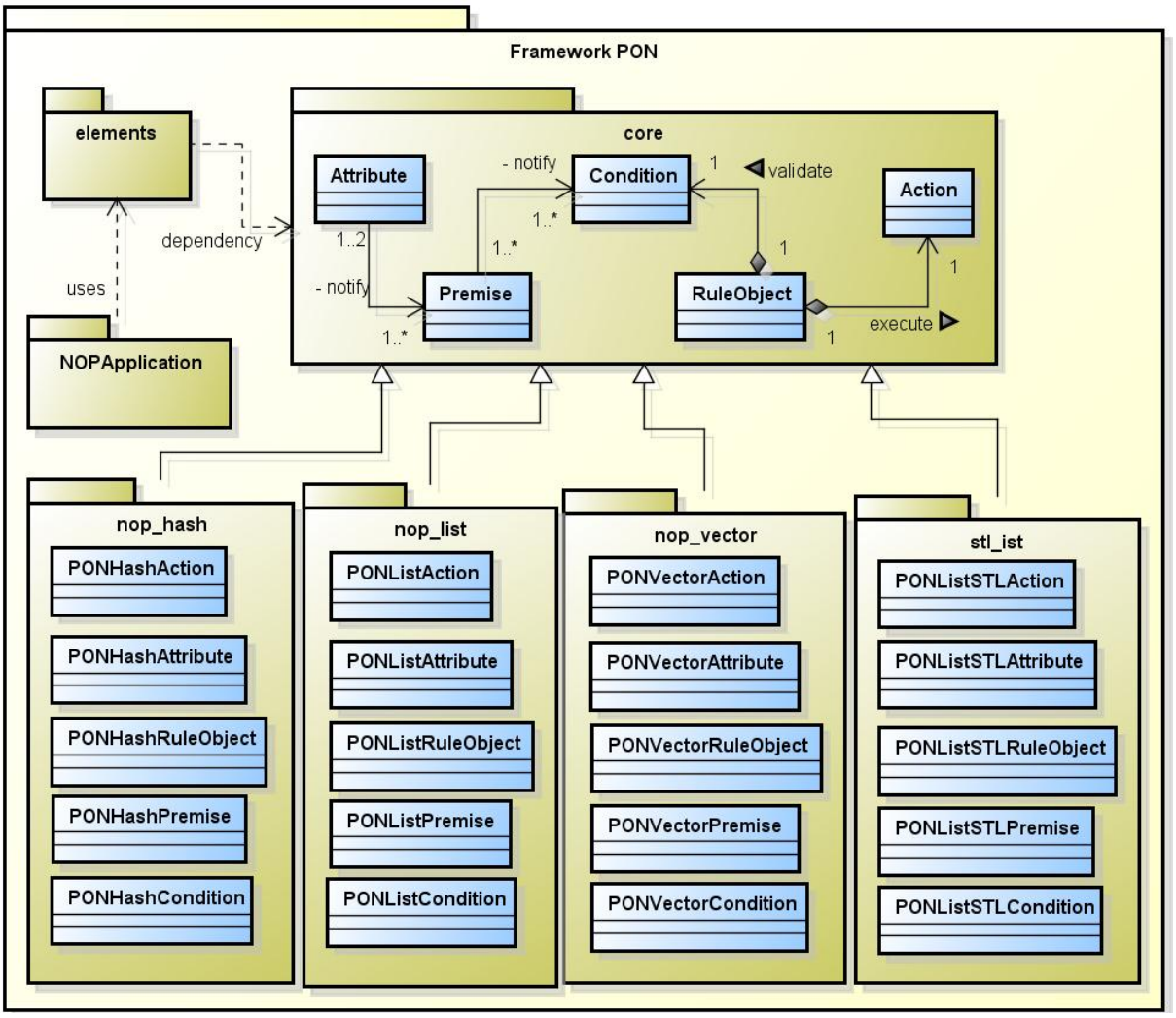


Figura 41 – Classes que possuem elementos iterados

No exemplo acima, as estruturas de dados são encapsuladas em seus respectivos pacotes. A estratégia é realizar a iteração dos elementos PON em cada instância de classe correspondente a uma respectiva estrutura de dado. Assim os métodos de manipulação dos elementos PON das classes pertencentes ao pacote *core*, são formadas por métodos virtuais puros, os quais são implementados em suas classes derivadas. Isto, de fato, flexibiliza o poder de manipulação dos elementos PON em cada estrutura de dados implementada sobre *Framework* PON.

Esta nova composição de pacotes se fez necessária para organizar as classes e suas responsabilidades. Assim, de maneira coesa, as classes foram agrupadas para um melhor entendimento e legibilidade. Ademais, esta composição permite agregar novas estruturas de dados de maneira rápida, clara e eficaz, uma vez que cada componente deve ser implementado seguindo esta nova composição de pacotes elaborada.

3.5 *FRAMEWORK* PON SOB O VIÉS DE OTIMIZAÇÕES

As técnicas de otimizações podem ser classificadas como independentes de máquina, quando podem ser aplicadas antes da geração do código na linguagem *assembly*, ou dependentes de máquina, quando aplicadas na geração do código *assembly* (AHO e ULLMAN, 1972). A otimização independente de máquina tem como requisito o levantamento dos blocos de comandos que compõem o programa. Essa etapa da otimização é conhecida como a análise de fluxo, que por sua vez subdivide-se em análise de fluxo de dados.

Ainda, estratégias que podem ser aplicadas analisando um único bloco de comandos são denominadas estratégias de otimização local, enquanto aquelas que envolvem a análise simultânea de dois ou mais blocos são denominadas estratégias de otimização global (AHO e ULLMAN, 1972). Neste âmbito, técnicas independentes de máquina foram aplicadas no código do *Framework PON*, onde se destacam:

1. O fluxo de notificações nem sempre é desejável a cada mudança de estado de um *Attribute*. Assim, o parâmetro *NO_NOTIFY* foi introduzido de maneira a evitar o disparo indesejado do fluxo de notificações quando ocorrer uma alteração no estado de um *Attribute*. Esta funcionalidade é útil em casos onde a alteração do estado de um *Atrribute* não implica diretamente na aprovação ou desaprovação de regras ou quando o *Attribute* varia excessivamente, evitando assim, o fluxo de notificações impertinente e indesejável.
2. Em contrapartida, o parâmetro *RENOTIFY*, força a realização do fluxo de notificações, uma vez que um determinado *Atrribute* mantenha o seu estado inalterado.
3. Ainda, o código responsável pelo cálculo lógico causal das *Premises* foi alterado, de modo a realizar o cálculo lógico causal somente na criação da respectiva *Premise*. Assim, evita-se o cálculo lógico redundante e desnecessário quando uma mesma *Premise* é compartilhada por mais de uma *Rule*.
4. A técnica de otimização global foi executada, ao se substituir objetos da classe *list* e *vector* da *STL* pelas diferentes estruturas de dados implementados para o *Framework* PON. Essa alteração ecoou em vários blocos de códigos de classes que utilizavam o percurso sobre listas de objetos da classe *list* e *vector*. Ainda foi

responsável pela completa reformulação do *Framework* PON sob o viés da estrutura especializada descrita na seção 3.3.

Outrossim, em algumas operações booleanas no código do *Framework* PON foi possível aplicar o operador *bit-a-bit XOR* (^), também conhecido como OU EXCLUSIVO. Esta técnica foi implementada especificamente sobre a classe *Boolean* e em trechos de código onde comparações booleanas são realizadas. A classe *Boolean* é responsável por agregar reatividade aos tipos booleanos primitivos da Orientação a Objetos. O Algoritmo 26 demonstra o trecho de código onde a operação *bit-a-bit* é realizada.

```
1 void Boolean::setValue(bool value) {
2       myBoolOld = myBool;
3       if (this->myBool ^ value) {
4             this->myBool = value;
5             notifyPremises();
6       }
7 }
```

Algoritmo 26 – *Operação bit-a-bit Boolean*

O método *setValue()* da classe *Boolean*, é responsável por atribuir um novo estado ao respectivo *Attribute*. Entretanto, a verificação deste estado é realizada através de uma operação *bit-a-bit,* ou operação binária, conforme esboçado no Algoritmo 26 linha 3. Deste modo, todas as validações sobre os *Attributes* do tipo *Boolean*, ocorrem de maneira otimizada por meio da utilização de operações *bit-a-bit* ao invés de operações comparativas booleanas tradicionais.

3.6 *FRAMEWORK* PON SOB O VIÉS DE REFATORAÇÕES

Inicialmente as refatorações de código ocorreram sobre as classes que encapsulam os tipos primitivos da linguagem de programação orientada a objetos. Estas classes, por sua vez, são responsáveis por adicionar reatividade aos tipos primitivos passivos encontrados na OO. O Diagrama de classes da Figura 42 apresenta as classes *Integer*, *Boolean*, *Char*, *Double* e *String*, responsáveis por agregar reatividade aos tipos primitivos da OO.

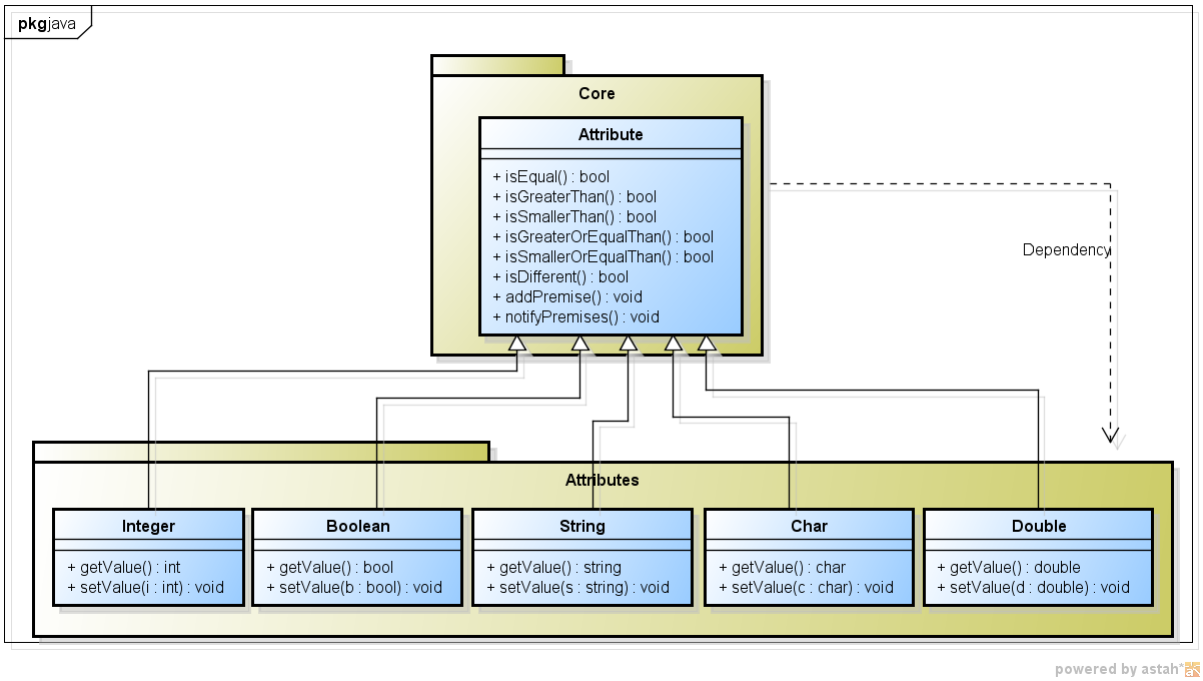


Figura 42 – Classes *Attributes*

As instâncias das classes da Figura 42 possuem comportamentos idênticos em relação aos seus métodos internos implementados. Sendo assim, após refatorar a classe *Boolean* as demais classes foram refatoradas seguindo a mesma linha de raciocínio. Dentre os métodos internos das classes que encapsulam os tipos primitivos, um método em específico é extremamente importante e utilizado por uma aplicação PON. Este método referido é o *setValue*(...), que por sua vez, poderá na maioria das vezes iniciar a cadeia de notificação entre os elementos PON.

Ainda, este método possui várias interfaces de parâmetros de métodos onde, por exemplo, o desenvolvedor PON poderá atribuir um simples tipo primitivo (*setValue(bool b)*) ou mesmo um respectivo elemento *Attribute (setValue(Attribute *Attribute)).* Neste âmbito, a versão original de uma das implementações de *setValue(..)* é esboçada pelo algoritmo da Figura 43.

```
1  /**
2   * Update the Boolean Attribute value.
3   * This method start a notification flow if the assigned value is different of the last value.
4   * @param value a primitive Boolean value to be assigned
5   */
6  void Boolean::setValue(bool value){
7      this->myBoolTmp = value;
8      if(this->myBool != this->myBoolTmp){
9          this->myBool = this->myBoolTmp;
10         notifyPremises();
11     }
12 }
13
14 /**
15  * Update the Boolean Attribute value.
16  * This method start a notification flow if the assigned value is different of the last value.
17  * @param value a Boolean object
18  */
19 void Boolean::setValue(Attribute *value){
20     if(isValueChange(value)){
21         myBool = ((Boolean*)value)->getValue();
22         notifyPremises();
23     }
24 }
25
26 /**
27  * Verify if the actual value is changed.
28  * @param value assigned object value
29  * @return the result of the comparison
30  */
31 bool Boolean::isValueChange(Attribute* value){
32     if(myBool != ((Boolean*)value)->getValue())
33         return true;
34         return false;
35 }
```

Figura 43 – *setValue Framework* Original

Na linha 6 o método *setValue(..)* recebe um tipo primitivo *bool (bool value).* Na sequência o método verifica o estado deste atributo e chama a execução do método *notifyPremises()* caso seu estado seja diferente do estado anterior. O método *setValue(...)* da linha 19, por sua vez, recebe como parâmetro um objeto do tipo *Attribute.* Na sequência o método valida então seu respectivo estado através da chamada ao método *isValueChange(...),* caso seja verdadeiro o método *notifyPremises()* é então executado.

A exemplificação do algoritmo da Figura 44 não muda sua funcionalidade original, entretanto agrega em maior simplicidade, desempenho (tendo em vista a eliminação da chamada do método *isValueChange(...)*) e ainda, melhora sua estrutura interna. Apesar de aparentemente ser uma simples mudança, o método em questão é utilizado em todo o início do fluxo de notificações do *Framework* PON.

```
1  /**
2   * Update the Boolean Attribute value.
3   * This method start a notification flow if the assigned value is different of the last value.
4   * @param value a primitive Boolean value to be assigned
5   */
6  void Boolean::setValue(bool value) {
7      myBoolOld = myBool;
8      if (this->myBool ^ value) {
9          this->myBool = value;
10         notifyPremises();
11     }
12 }
13
14 /**
15  * Update the Boolean Attribute value.
16  * This method start a notification flow if the assigned value is different of the last value.
17  * @param value a Boolean object
18  */
19 void Boolean::setValue(Attribute* value) {
20     setValue((Boolean*)(value)->getValue());
21 }
```

Figura 44 – *setValue Framework* Atual

Na linha 6 o método *setValue(...)* é mantido como o original, entretanto o método *setValue(...)* da linha 19, executa a chamada para método *setValue(...)* da linha 6. Assim o próprio método verificará o estado do respectivo *Attribute* e executará o método *notifyPremises*(), caso seu estado seja diferente do anterior.

Ainda, um terceiro viés sobre o método *setValue(...)* é explanado no algoritmo da Figura 45. Observa-se que em sua versão original a utilização do atributo *flag* no parâmetro do método é utilizado única e exclusivamente para invocar o método *renotifyPremises(),* caso o estado do atributo *value* se mantenha. Neste âmbito, de forma a elucidar sua real utilidade e consequentemente melhorar a legibilidade do código, o método *setValue(..)* da Figura 45 foi refatorado.

```
1  /**
2   * Update the Boolean Attribute value.
3   * This method start a notification flow if the assigned value is different or
4   * even the same than the last one.
5   * This method is used when the renotification mechanism is desired.
6   * @param value a primitive Boolean value to be assigned
7   * @param flag should be passed the value of the Attribute::RENOTIFY,
8   * it denotes the desire of the renotification mechanism
9   */
10 void Boolean::setValue(bool value, int flag){
11
12     this->myBoolTmp = value;
13     if(this->myBool != this->myBoolTmp){
14         this->myBool = this->myBoolTmp;
15         notifyPremises();
16     }
17     else{
18         renotifyPremises();
19     }
20 }
```

Figura 45 – *renotify Framework* Original

O método refatorado é ilustrado pelo algoritmo da Figura 46. Desta forma percebe-se que inicialmente o atributo *flag* na linha 13 é validado, pois, na maioria dos casos, o método *renotifyPremises()* será invocado. No algoritimo da Figura 45, entretanto, será mais provável ser executado a instrução "*else"* da linha 17, do que a validação da instrução *"if"* da linha 13. Ainda é possível observar a clareza da utilização do atributo *flag,* sendo neste caso utilizado para ativar o processo de renotificação ou mesmo desativar o processo de notificação através da constante *Attribute::NO_NOTIFY,* onde casos em que existam *Attributes* de *FBE* que não estejam vinculados a nenhum elemento *Premise.*

```
1  /**
2   * Update the Boolean Attribute value.
3   * This method start a notification flow if the assigned value is different or
4   * even the same than the last one.
5   * This method is used when the renotification mechanism is desired.
6   * @param value a primitive Boolean value to be assigned
7   * @param flag should be passed the value of the
8   *  - Attribute::RENOTIFY, it denotes the desire of the renotification mechanism
9   *  - Attribute::NO_NOTIFY, it denotes the desire of not notifying the value change
10  */
11 void Boolean::setValue(bool value, int flag) {
12     myBoolOld = myBool;
13     if (flag == Attribute::RENOTIFY) {
14         myBool = value;
15         renotifyPremises();
16     } else if (myBool ^ value) {
17         myBool = value;
18         if (flag != Attribute::NO_NOTIFY)
19             notifyPremises();
20     }
21 }
```

Figura 46 – *renotify Framework* Atual

Conforme esboçado e apresentado as devidas refatorações de código para a classe *Boolean*, as demais classes pertencentes ao pacote *Attributes* foram alteradas na mesma sintonia e proporção. Ainda, é importante ressaltar, que a versão do *Framework* realizada por Banaszewski (2009) é advinda da evolução do *Framework* inicialmente proposto pelos trabalhos de Simão (2005). Neste âmbito, esta versão encontrava-se consideravelmente bem estruturada, mas não o suficiente para explorar características do PON com certo refino.

Desta forma, todo o código fonte do *Framerwork* PON foi refatorado e reimplementado. Este fenômeno ocorreu principalmente devido à nova composição de classes e pacotes proposta e implementadas para o *Framerwork* PON, conforme foi descrito na seção 3.4. A título de exemplificação, as classes que fazem parte do contexto da cadeia de notificações do *Framework* PON, foram todas refatoradas. No exemplo da Figura 47, observa-se que os métodos de manipulação de elementos PON, até então implementados em sua classe origem, foram desacoplados e implementados em suas classes derivadas.

```
1  void Attribute::addPremise(Premise* premise) {
2      // Virtual method, it does nothing here...
3  }
4
5  void Attribute::removePremise(Premise* premise) {
6      // Virtual method, it does nothing here...
7  }
8
9  void Attribute::notifyPremises() {
10     // Virtual method, it does nothing here...
11 }
12
13 void Attribute::renotifyPremises() {
14     // Virtual method, it does nothing here...
15 }
```

Figura 47 – Classe *Attribute*: manipulação de elementos PON

Conforme observado na classe *Attribute* da Figura 47, os métodos de manipulação de elementos PON (*Premises*), como o método *addPremise(...)* da linha 1, são definidos como métodos virtuais nesta classe. Neste âmbito, sua implementação ocorrerá nas classes derivadas de cada estrutura de dados implementada para o *Framework* PON. Deste modo, a Figura 51 apresenta a implementação da classe derivada *NOPVectorAttribute.*

```
1  void NOPVectorAttribute::addPremise(Premise* premise) {
2      premisesList.push_back(premise);
3  }
4
5  void NOPVectorAttribute::removePremise(Premise* premise) {
6      premisesList.remove(premise);
7  }
8
9
10 void NOPVectorAttribute::notifyPremises() {
11     Premise* tmpPremise;
12     premisesList.initIterator();
13     while (premisesList.hasNext()) {
14         tmpPremise = premisesList.next();
15             if (tmpPremise->isExclusive())
16                 tmpPremise->notifyConditionsAboutPremiseExclusive();
17             else
18                 tmpPremise->notifyConditions();
19     }
20 }
21
22 void NOPVectorAttribute::renotifyPremises() {
23     Premise* tmpPremise;
24     premisesList.initIterator();
25     while (premisesList.hasNext()) {
26         tmpPremise = premisesList.next();
27             if (tmpPremise->isExclusive())
28                 tmpPremise->renotifyConditionsAboutPremiseExclusive();
29             else
30                 tmpPremise->renotifyConditions();
31     }
32 }
```

Figura 48 – Classe *PONVectorAttribute*

Este processo de refatoração de código exemplificado nos parágrafos anteriores forma a base para o procedimento descrito na seção 3.3. Neste âmbito, todas as classes que fazem parte da cadeia de notificação, esboçada na Figura 12, sofreram refatoração de código em relação aos métodos responsáveis por manipular elementos PON.

3.7 REFLEXÕES

Este capítulo abordou as principais mudanças realizadas para o *Framework* do PON. O foco principal foi em relação à estrutura de dados que comporta a cadeia de notificação do *Framework* PON, conforme discutido na seção 2.3.1. Neste

sentido, de modo a comportar as diferentes estruturas de dados incluídas no *Framework* PON, uma primeira abordagem foi proposta e implementada, conforme descreve a seção 3.2. Nesta abordagem, uma classe genérica define a forma de iterar e manipular os elementos PON.

Em seguida, uma nova abordagem foi proposta e implementada, conforme descreve a seção 3.3. Nela, a iteração e manipulação de elementos PON, é especializada para cada classe correspondente a estrutura de dado implementada para o *Framework* PON. Isto de fato, flexibilizou a manipulação sobre os elementos PON. Ademais, caso uma nova estrutura venha a ser implementada para o *Framework* PON, esta poderá realizar a manipulação e iteração sobre os elementos conforme determina seu estilo, não dependendo assim, de uma classe genérica para executar tais procedimentos.

Outrossim, técnicas de otimização foram implementadas, como a utilização de aritméticas de ponteiros sobre a estrutura de dados *PONVECTOR* , bem como a utilização de operações binárias sobre *Attributes* do tipo *Boolean.* Todas estas novas estruturas de dados implementadas visam melhorar o desempenho de aplicações do PON. Ainda refatorações de código foram implementadas sobre as classes que encapsulam os tipos primitivos da orientação a objetos e sobre as classes que orquestram a cadeia de notificações do *Framework* PON.

As próximas seções tem por objetivo, por à prova as implementações realizadas sobre o *Framework* PON. Neste âmbito, é descrito o escopo das aplicações PON utilizadas bem como suas características de implementações. Por fim, são realizados os testes de desempenho sobre as duas versões do *Framework* PON, ou seja, sobre sua versão concebida por BANASZESWKI (2009) e sua nova versão materalizada neste trabalho de pesquisa.

3.8 ESCOPO E DESEMPENHO DAS APLICAÇÕES PON

Esta seção descreve o escopo de cada aplicação empregada como estudo de caso utilizado sobre a nova versão do *Framework* PON e como base de comparação para com o *Framework* precedente. É pertinente ressaltar que a implementação da versão tracional do jogo Mira ao Alvo descrito na seção 3.9 foi inicialmente implementada por BANASZEWSKI (2009). Entretanto, para os testes de desempenho, foram adicionados novos estados e algumas variações desta implementação. Ainda, a implementação do aplicativo Vendas, foi inicialmente realizada por (BATISTA, *et. al.,* 2011) (SIMÃO, *et. al.*, 2012b). Ao seu turno, o aplicativo Terminal Telefônico foi desenvolvido por (LINHARES, *et. al.,* 2011). Por fim o aplicativo *Pac-Man* foi implementado por (RONSZCKA, *et. al.,* 2011) (SIMÃO, *et. al.*, 2012c). Entretanto, todas as *Rules* dos referidos aplicativos foram reescritas para a última versão do *Framework* PON.

3.9 MIRA AO ALVO

Basicamente, o jogo consiste em um ambiente onde as entidades do tipo mira interagem ativamente com as entidades do tipo alvo. Mais precisamente, as entidades miras e as entidades alvos são representadas respectivamente por arqueiros e maçãs, sendo que há uma maçã para cada arqueiro, cf. Figura 49. Em termos de implementação, cada arqueiro e maçã são representados por meio de objetos, os quais interagem de acordo com a validação de suas expressões causais pertinentes. Estas expressões causais relacionam os atributos/estados e métodos/serviços destes objetos (BANASZEWSKI, 2009) (SIMÃO, *et. al*., 2012a).

Neste contexto, cada arqueiro e cada maçã recebem um identificador numérico, sendo que um arqueiro somente pode flechar uma maçã que apresente o identificador numérico correspondente ao seu. Ainda, os arqueiros também apresentam um atributo que denota os seus estados de pronto (i.e. *status*) para agir sobre o cenário (i.e. flechar a respectiva maçã). Outrossim, um terceiro elemento denominado arma de fogo, tem a função de sinalizar com o seu disparo o início de

uma iteração, permitindo que os arqueiros interajam com as respectivas maçãs (BANASZEWSKI, 2009) (SIMÃO, *et. al.*, 2012a).

Em relação às maçãs, estas também apresentam um atributo que denota os seus estados de pronto (i.e. *status*). Ainda, estas apresentam um atributo que explicita se a mesma já foi perfurada por uma flecha ou ainda não (*i.e. isCrossed*). Também, as maçãs apresentam um atributo que se refere a sua coloração (*i.e.* color), uma vez que as maçãs podem se apresentar em duas diferentes cores: vermelha ou verde (BANASZEWSKI, 2009) (SIMÃO, *et. al.*, 2012a).

Com a intenção de melhor elucidar esta aplicação, a Figura 49 ilustra a interação entre os personagens arqueiros e maçãs. Nesta, os arqueiros e maçãs estão organizados em fila de acordo com os seus números identificadores, o que assegura que um arqueiro está posicionado absolutamente à frente da sua respectiva maçã (BANASZEWSKI, 2009) (SIMÃO, *et. al.*, 2012a).

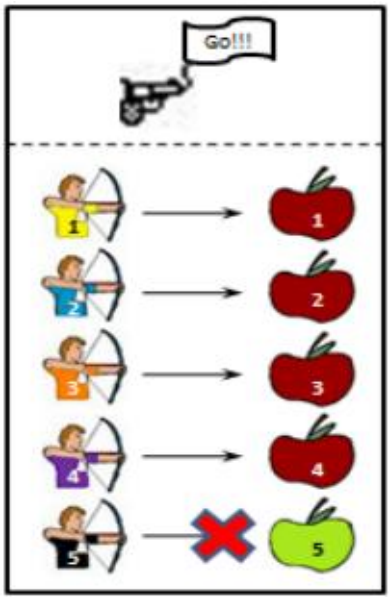


Figura 49 – Mira ao Alvo (BANAZEWSKI, 2009)

Neste cenário, cada arqueiro somente pode interagir com a sua respectiva maçã após a constatação de quatro premissas: (a) se a cor da maçã que está posicionada diretamente em sua frente é vermelha, (b) se a maçã que está posicionada diretamente em sua frente está pronta para ser atingida, (c) se ela é identificada pelo seu número correspondente e ainda, (d) o estado da arma de fogo é atirar (BANASZEWSKI, 2009) (SIMÃO, *et. al.*, 2012a).

Se as quatro condições forem satisfeitas, o arqueiro está liberado para atingir a respectiva maçã com a projeção de sua flecha. Desta forma, percebe-se que para cada par de arqueiros e maçãs deve haver uma expressão causal para comparar os seus estados (BANASZEWSKI, 2009). Ainda o diagrama de casos de uso da Figura 50 apresenta os casos de uso desenvolvidos para a aplicação Mira ao Alvo.

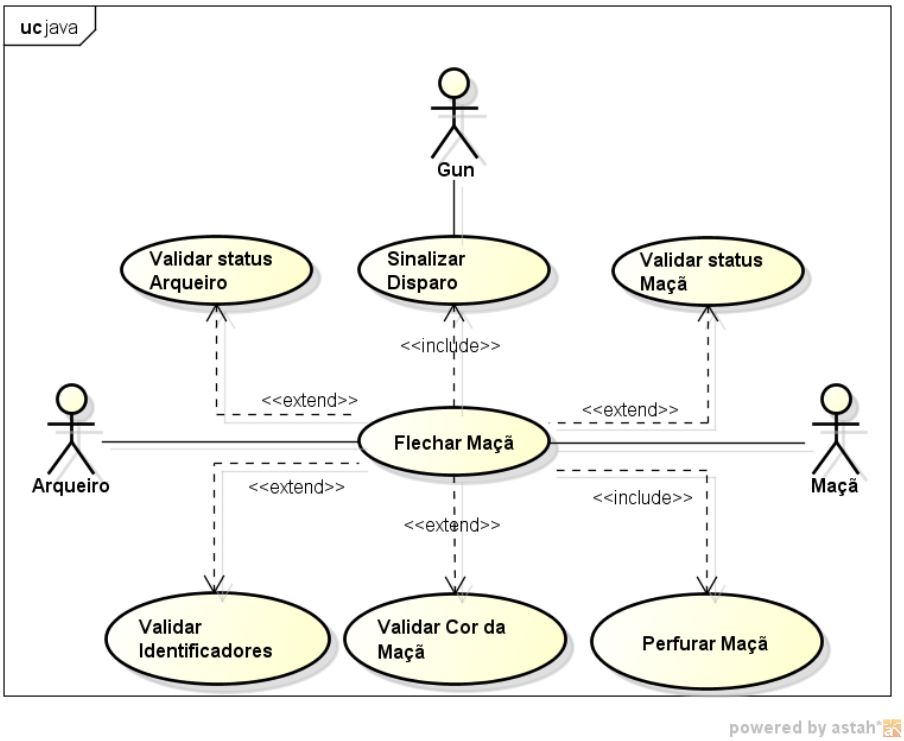


Figura 50 – Casos de Uso Mira ao Alvo

O caso de uso principal corresponde ao caso de uso “Flechar Maçã”. Para tal é necessário a validação das expressões causais descritas de antemão. Tais expressões causais na forma de *Rules* são consideradas no âmbito dos subcasos de uso, do caso de uso principal “Flechar Maçã”. Ainda, quando todas as Premises de ao menos uma *Rule* são satisfeitas, a implementação relativa ao caso de uso “Perfurar Maçã” será ativado. Neste momento a respectiva maçã não poderá mais ser atingida e entrará em estado inabilitado e/ou perfurado.

Outrossim, o diagrama de classes da Figura 51 esboça o projeto da aplicação mira ao alvo. O projeto e subsequente a implementação tem basicamente dois momentos, que se constituem na concepção dos elementos da base de fatos e na posterior concepção de *Rules.* Deste modo, as classes implementadas como *Gun*, *Apple* e *Archer*, representam os *FBEs* desta implementação PON, as quais

estendem da classe base *FBE*. Estas classes possuem seus estados representados pela agregação das classes *Boolean, Integer* e *String,* as quais são derivadas da classe base *Attribute.*

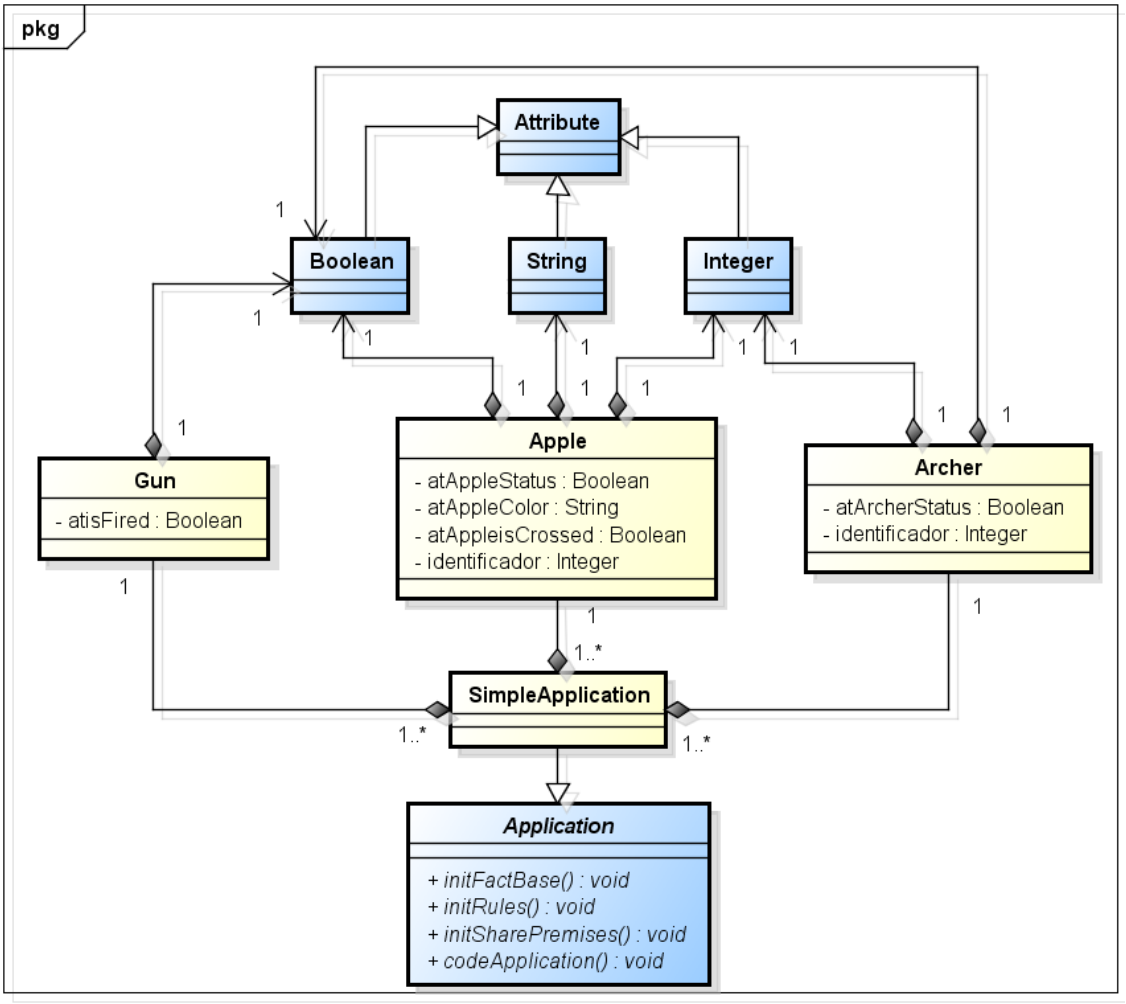


Figura 51 – Mira ao Alvo: Diagrama de Classes

A instanciação dos *FBE*s e inicialização de valores ocorrem na definição do método *initFactBases()* da classe *SimpleApplication*. Ao seu turno, a criação de *Rules* é realizada através do método *initRules*() da mesma classe. A Figura 52 demostra um exemplo de *Rule* com as respectivas *Premises* (descritas de antemão) definidas após a instanciação dos *FBE*.

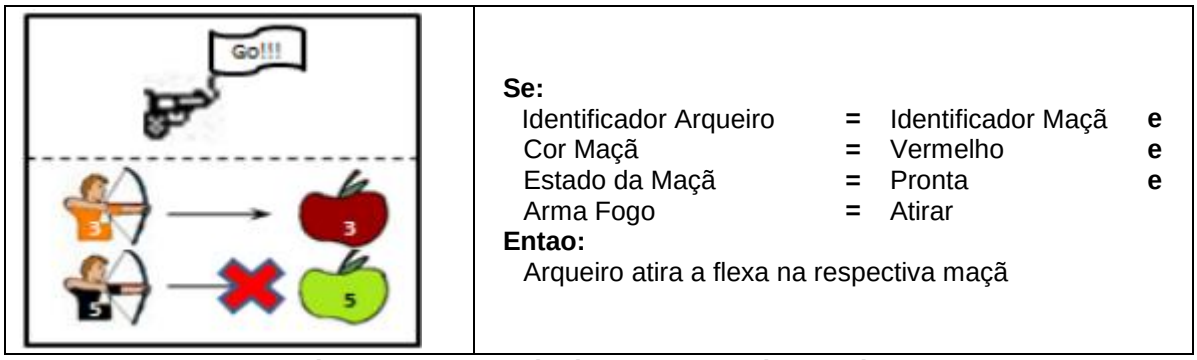


Figura 52 – Exemplo de uma regra Mira ao Alvo

Neste exemplo, a *Rule* que determina o momento do arqueiro flechar uma maçã será executada após a satisfação das *Premises* atribuídas a ela, conforme especificado na Figura 52. Por fim, após criação e instanciação dos *FBEs* bem como a definição de suas *Rules,* a aplicação poderá ser executada, com a implementação do método *codeApplication().* O fluxo da execução da aplicação Mira ao Alvo pode ser observado pelo diagrama de atividades da Figura 53.

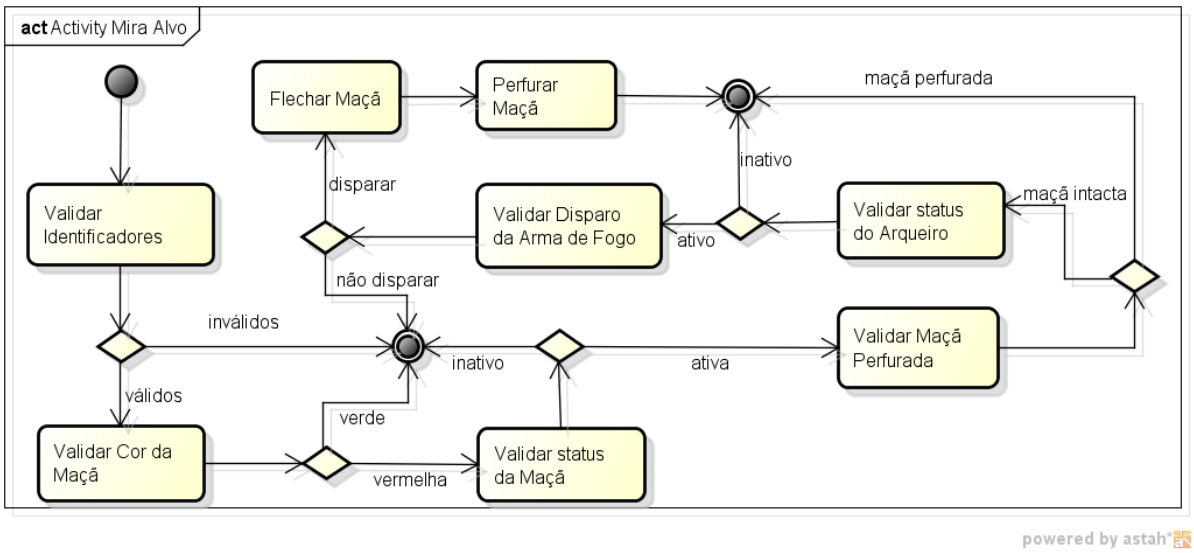


Figura 53 – Diagrama de Atividades Mira ao Alvo

Neste experimento, os arqueiros são representados pela classe *Archer* e as maçãs pela classe *Apple*, ambas derivadas da classe *FBE*. Assim, cem arqueiros interagem com a mesma quantidade de maçãs por um milhão de iterações. Os cem arqueiros e maçãs são agrupados em estruturas de dados do tipo *vector,* respectivamente chamadas de *ArcherList* e *AppleList*, conforme apresenta o algoritmo da Figura 54 nas linhas 2 e 4.

```
1  void SimpleApplication::initFactBase() {
2      archerList = new vector<Archer*>();
3      gun = new Gun();
4      appleList = new vector<Apple*>();
5      archer= new Archer();
6      for (int i = 0; i < 100; i++) {
7          Archer* archerTmp;
8          archerTmp = new Archer(i);
9          archerList->push_back(archerTmp);
10         Apple* appleTmp;
11         appleTmp = new Apple(i);
12         appleList->push_back(appleTmp);
13     }
14 }
```

Figura 54 – Instanciação dos *FBE*

Desta forma, estas estruturas (*ArcherList* e *AppleList*) são percorridas em um laço de repetição para a criação das cem expressões causais pertinentes, i.e. cem *Rules* pertinentes. O algoritmo da Figura 55 apresenta o código que inicia a respectiva base de regras, i.e. conjunto de *Rules*. Nela observa-se a implementação propriamente dita em PON das expressões causais à luz do exemplo dado na Figura 52.

```
1  void SimpleApplication::initRules() {
2      Premise* p = elementsFactory->createPremise(gun->atIsFired, true, Premise::EQUAL, false);
3      for ( int i = 0; i < appleList->size(); i++) {
4          Apple* appleTmp = appleList->at(i);
5          Archer* archerTmp = archerList->at(i);
6
7          RuleObject* rlFireApple = elementsFactory->createRuleObject("rule",
8                  scheduler, Condition::CONJUNCTION);
9          rlFireApple->addPremise(elementsFactory->createPremise(appleTmp->atIdentity,
10                 archerTmp->atIdentity, Premise::EQUAL, false));
11         rlFireApple->addPremise(elementsFactory->createPremise(appleTmp->atAppleColor,
12                 true, Premise::EQUAL, false));
13         rlFireApple->addPremise(elementsFactory->createPremise(appleTmp->atAppleStatus,
14                 true, Premise::EQUAL, false));
15         rlFireApple->addPremise(elementsFactory->createPremise(appleTmp->atAppleIsCrossed,
16                 false, Premise::EQUAL, false));
17         rlFireApple->addPremise(elementsFactory->createPremise(archerTmp->atArcherStatus,
18                 true, Premise::EQUAL, false));
19         rlFireApple->addPremise(p);
20         rlFireApple->addInstigation(elementsFactory->createInstigation(appleTmp->mtStatusOff));
21     }
22
23 }
```

Figura 55 – Instanciação das *Rules*

Os testes foram executados na plataforma *Linux* versão *Debian* em um *PC Core i5*, 2.45 *Ghz*, 6 *Gigabytes* de memória *RAM*. A Figura 56 apresenta os testes,

neste ambiente, sobre o *Framework* original e sua versão atualizada sobre o viés das novas estruturas de dados implementadas para o *Framework* PON. Os tempos são apresentados em milissegundos. A quantidade de iterações processadas foi de cem mil e o percentual de avaliações causais aprovadas variam de 10%, 50% e 100%.

Iterações	%	Framework Original	PONLIST	PONVECTOR	PONHASH
100.000	10%	6701	1430	1391	1903
100.000	50%	10549	4251	3873	6440
100.000	100%	18723	6362	5611	10561

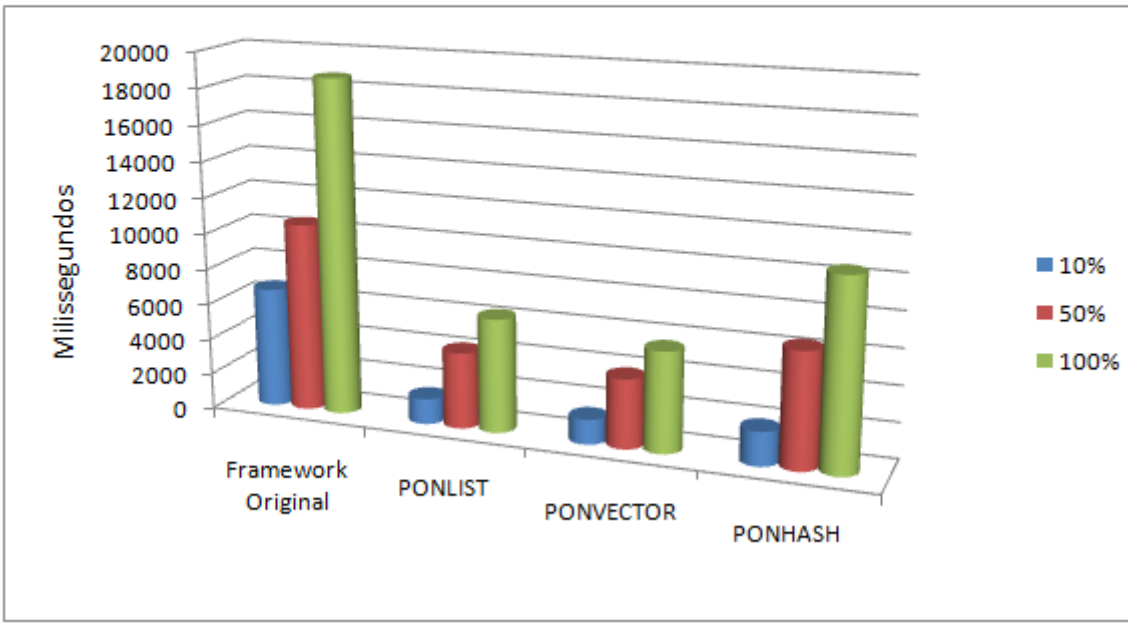


Figura 56 – Tempos de execução Mira ao Alvo - Linux

A estrutura de dados *PONVECTOR* apresentou em média 30% do tempo de processamento utilizado em relação a versão original do *Framework* PON, o que corresponderia a um ganho de desempenho de cerca 3 (três) vezes, conforme observado na Figura 56. A estrutura *PONLIST*, por sua vez, manteve-se com os valores aproximados aos tempos encontrados sobre a estrutura *PONVECTOR*, todavia, sempre com pior desempenho. Por fim, a estrutura de dados *PONHASH* obteve melhores tempos de execução em comparação ao *Framework* original, entretanto, obteve piores tempos em relação às demais estruturas de dados.

Conforme descrito na seção 3.1.4, a estrutura *PONHASH* necessita realizar algumas operações antes de acessar um elemento PON propriamente dito. Primeiro,

é necessário realizar o cálculo da função *hash*. Segundo, o acesso ao respectivo elemento PON é realizado. Por fim, é necessário reavaliar o seu estado dito como anterior. Neste âmbito, para casos em que a lista a ser “percorrida” é pequena e ainda o seu respectivo estado possua muita variação, o processamento executado por *PONHASH* torna-se ineficiente em comparação as demais estruturas de dados implementados para o *Framework* PON.

De modo a realizar experimentos adicionais para a estrutura *PONHASH*, o escopo do aplicativo Mira ao Alvo foi alterado. Esta alteração visa estabelecer as condições em que um determinado *Attribute* de um respectivo *FBE* possua em seu escopo uma quantidade relativamente grande de *Premises.* Ademais, tal *Attribute* deve possuir uma grande variação de estados ao longo da execução da aplicação PON em questão.

Neste âmbito, o *Attribute atIdentityOfBullet* foi incluído ao *FBE Gun.* Este *Attribute* especifica um respectivo identificador para cada *bullet* disparado pelo *FBE Gun*. Assim, cada *Attribute atIdentityOfBullet* receberá um identificador único, o qual deve ser igual ao identificador único do *FBE Archer.* Neste contexto, um *FBE Archer* somente poderá flechar uma maçã quando o *Attribute atIdentityOfBullet* for igual ao seu respectivo identificador, denotado pelo *Attribute atIdentity*. Assim, o algoritmo da Figura 57 corresponde a *Premise* incluída as expressões causais relacionadas a Figura 52.

```
1  void SimpleApplication::initRules() {
2
3      for ( int i = 0; i < appleList->size(); i++) {
4
5          ...
6          Archer* archerTmp = archerList->at(i);
7          rlFireApple->addPremise(elementsFactory->createPremise(
8                                      gun->atIdentityOfBullet, archerTmp->atIdentity,
9                                      Premise::EQUAL, false));
10         ...
11
12     }
13 }
14
```

Figura 57 – Inclusão *Attribute atIdentityOfBullet*

Observa-se no algoritimo da Figura 57, que o *Attribute atIdentityOfBullet* varia em 100 (cem), onde 0 (zero) representa o identificador do primeiro *FBE ArcherTmp* e

100 (cem) o último. Deste modo, após uma mudança de estado do respectivo *Attribute "gun->atIdentityOfBullet",* o mesmo percorrerá uma lista de cem posições em estruturas do tipo *list* e *vector* da *STL*, *PONLIST* e *PONVECTOR*. Entretanto, sobre estrutura *PONHASH*, o acesso a *Premise* correspondente será "direta". Assim, os resultados obtidos são apresentados na Figura 58. Neste experimento foi utilizado um computador de processador *Core i5* 2.45 *Ghz,* 6 *Gigabytes* de memória *RAM* e Sistema Operacinal *Linux Debian.* A quantidade de iterações processadas foi de cem mil e o percentual de avaliações causais aprovadas variam de 10%, 50% e 100%. Os tempos encontram-se em milissegundos.

Iterações	%	Framework Original	PONLIST	PONVECTOR	PONHASH
100.000	10%	15652	13362	12589	2642
100.000	50%	51808	34441	32259	9892
100.000	100%	112980	85439	77895	18987

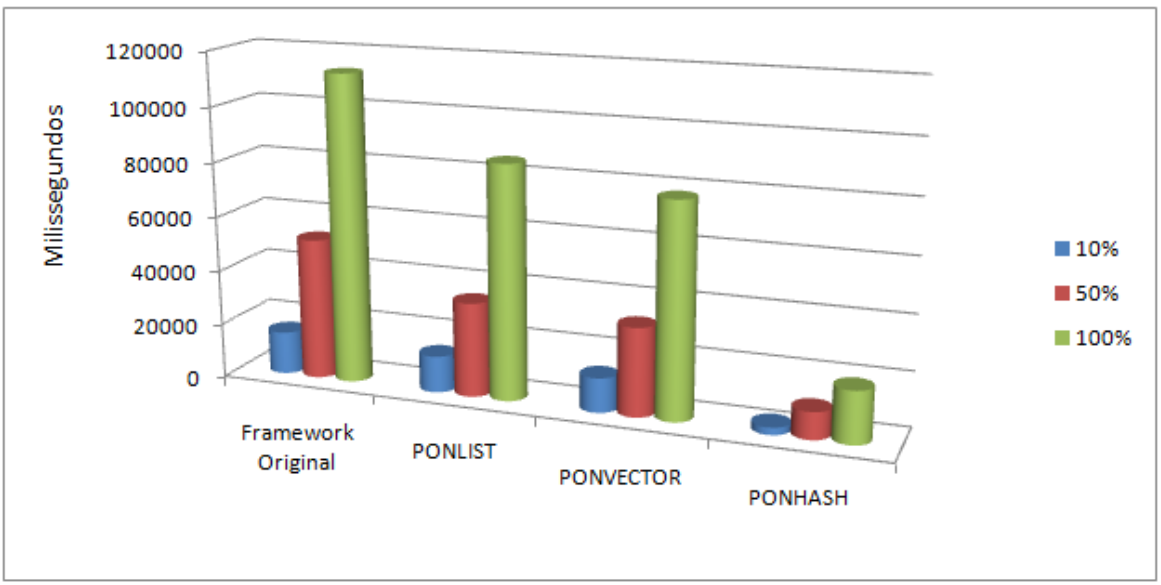


Figura 58 – Tempo de execução Mira ao Alvo - *PONHASH*

A Figura 58 apresenta a eficiência obtida pela estrutura denominada *PONHASH,* a qual representa aproximadamente 18% do tempo de processamento em relação a versão do *Framework* PON original, o que corresponde a um ganho de desempenho de cerca de 5 (cinco) vezes. Uma vez que o acesso a lista de *Premise* do *Attribute atIdentityOfBullet* é realizada de forma "direta" e nas demais estruturas é realizado um percorrimento sobre todas as suas *Premises*. Isto, de fato, comprova a eficiência da estrutura *PONHASH* em casos descritos na seção 3.1.4.

Isto tudo considerado, outro assunto a ser ressaltado é que nos trabalhos realizados por BANASZEWSKI (2009) foram realizados experimentos comparativos entre o *Framework* PON definido por ele e o (sub) paradigma Orientado a Objetos do PI. Estes experimentos demonstraram a ineficiência do mecanismo imperativo em alguns cenários e as vantagens do mecanismo de notificações (BANASZEWSKI, 2009) nestes mesmos cenários considerados.

De maneira suscinta, nos cenários descritos por BANASZEWSKI (2009), quando o percentual de expressões causais aprovadas se encontra abaixo de 27% (isto é, há 73% das expresões causais que não são aprovadas) e tais expressões causais possuem somente problemas de redundância temporal, o mecanismo de inferência do PON é mais eficiente que o mecanismo de inferência da OO. Ainda quando inserido os problemas de redundância estrutural no contexto da aplicação, o PON mostra-se em todos os casos descritos mais eficiente (BANASZEWSKI, 2009).

Outrossim, os resultados obtidos com os estudos comparativos realizados por BANASZEWSKI (2009) entre o PON e a PD, especificamente os SBRs com o eficiente motor de inferência *RETE*, demonstraram a eficiência superior do mecanismo de notificações do PON. Desta vez, o mecanismo de notificações se mostrou mais eficiente ao evitar as deficiências existentes no algoritmo *RETE* como a sua complexidade estrutural, armazenamento e execuções redundantes, entre outras (BANASZEWSKI, 2009). É importante ressaltar que o motor de inferência *RETE* é considerado o estado da técnica em máquina de inferência para SBRs. Atualmente, o *RETE* é o algoritmo de inferência de maior impacto industrial, sendo aplicado em,*shells* como: *OPS5, ART, CLIPS, RuleWorks, ILOG Rules e JESS* (BANASZEWSKI, 2009).

3.10 VENDAS

Esta seção apresenta um segundo caso de estudo que consiste na implementação de um sistema de pedido de vendas usual. Essa aplicação seria uma tradicional aplicação do estilo *CRUD* (acrônimo de *Create, Retrieve, Update* e *Delete*) que permite criar, recuperar, atualizar e eliminar dados (usualmente cadastrais) (BATISTA, *et. al.,* 2011) (SIMÃO, *et. al.,* 2012b). Em suma, o escopo da aplicação Vendas é composto pelo diagrama de casos de uso da Figura 59.

O ator Administrador é responsável por manter as informações do cadastro de formas de pagamentos. O ator Setor de Compras, por sua vez, é responsável por cadastrar e atualizar as informações de produtos e do próprio setor de compras. Ainda, o ator Cliente, solicita uma venda a um respectivo ator Vendedor. Este, por sua vez, cadastra o cliente e efetua a venda propriamente dita.

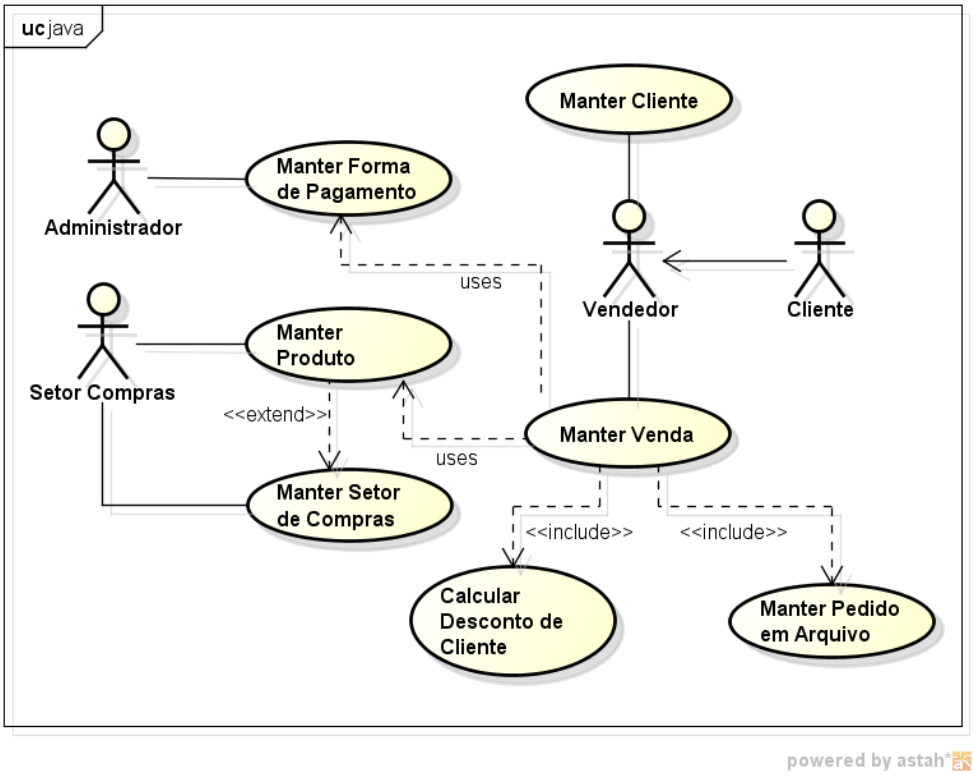


Figura 59 – Caso de uso vendas.

Para elucidar o comportamento e as responsabilidades de cada caso de uso da Figura 59, a Figura 60 apresenta um diagrama de atividades da execução de um pedido de venda. Inicialmente o cliente (denotado pelo ator cliente) solicita a venda para um respectivo vendedor (denotado pelo ator vendedor). Assim o vendedor informará o respectivo cliente que realizará o pedido. Uma vez escolhido e aprovado a venda para determinado cliente, devem ser informados os produtos que irão compor o pedido. O sistema possui validações quanto à existência de produtos e clientes. Ademais, verifica-se o estoque disponível de tais produtos. Se o produto escolhido para venda pertencer ao setor de perecíveis, verifica-se ainda a sua data de validade. Na ocorrência de produtos vencidos, a venda não será permitida (BATISTA *et al.*, 2011) (SIMÃO, *et. al.,* 2012b).

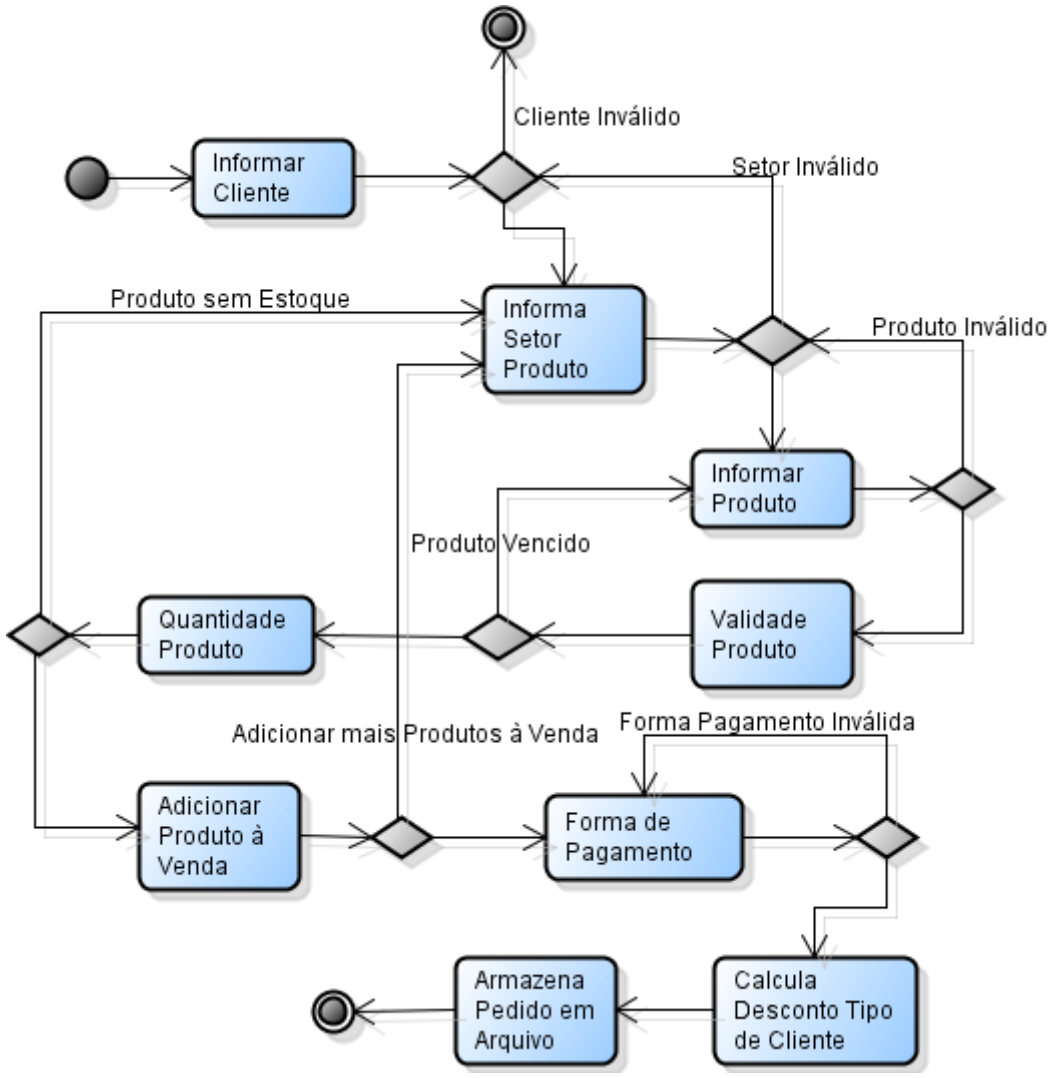


Figura 60 – Diagrama de atividades - sistema de vendas, adaptado de (VENÂNCIO *et al.*, 2011)

Após todo o ciclo de informe de produtos, a venda poderá ser finalizada após a inserção da forma de pagamento. Na implementação desse sistema, existem apenas duas formas de pagamento possíveis, à Vista ou à Prazo. O cliente, em seu cadastro, possui uma informação sobre seu limite de crédito. Caso a forma de pagamento escolhida tenha sido à Prazo, o sistema verifica se o cliente tem permissão para efetuar a compra, confrontando o valor total do pedido com seu limite de crédito. Ademais, no cadastro do cliente há uma informação que lhe concede um tipo de classificação. Utiliza-se tal classificação para a concessão de descontos especiais durante a finalização da venda. Para tanto, existe um total de 20 tipos de classificação de clientes que dispõem de descontos que variam de uma faixa de 5% a 95% (VENÂNCIO *et al.*, 2011) (SIMÃO, *et. al.,* 2012b).

De modo a facilitar o entendimento do escopo desse sistema, o diagrama de classes ilustrado na Figura 61 externa a essência da implementação do sistema de Vendas por meio de suas principais classes.

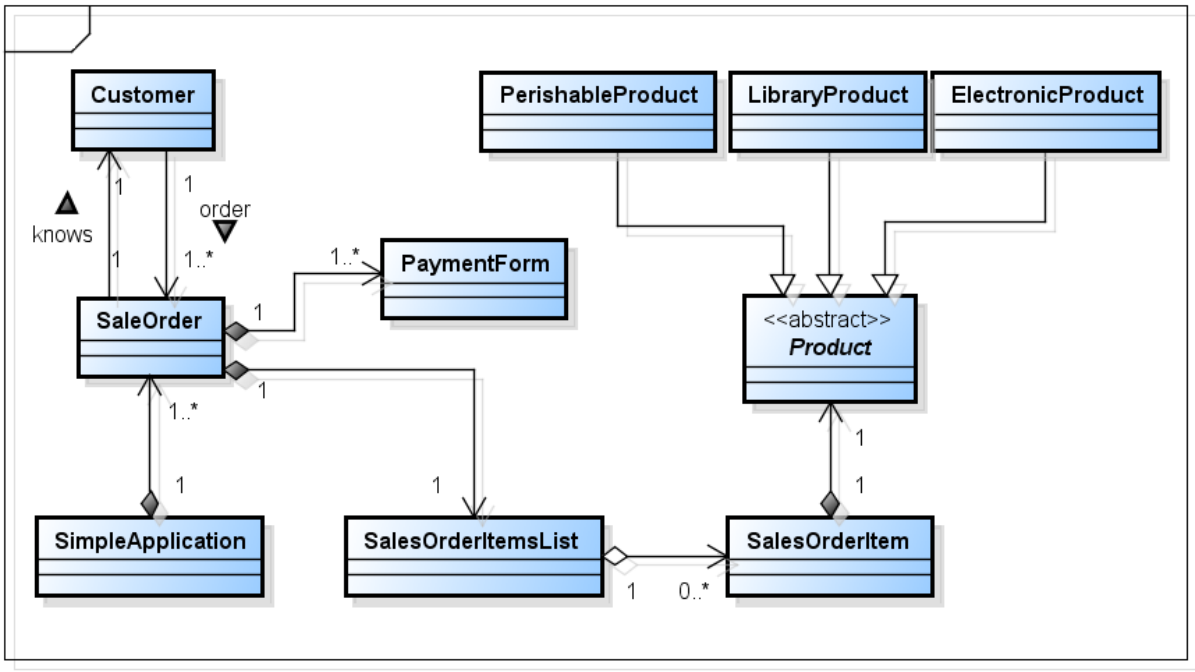


Figura 61 – Diagrama de classes - sistema de vendas, adaptado de (VENÂNCIO *et al.*, 2011)

Conforme ilustrado na Figura 61, a classe *SaleOrder* possui a responsabilidade de armazenar os dados de um pedido. Basicamente, um pedido é composto por um cliente, uma forma de pagamento e uma lista de itens do pedido. Em tal lista são armazenados os produtos e suas respectivas quantidades. A classe *Product* é definida como abstrata, podendo ser estendida para quaisquer tipos de produtos que possam a vir a ser implementados. Ainda a classe principal denominada *SimpleApplication*, define a instanciação dos *FBEs* e as suas respectivas *Rules.* Ademais, a execução da aplicação propriamente dita ocorre através da execução do método *codeApplication* da classe *SimpleApplication.*

A título de exemplificação, a Figura 62 demonstra a composição da *Rule* responsável por finalizar uma venda. Nela estão relacionadas às *Premises* que deverão ser satisfeitas para que a finalização da venda ocorra. Assim, a primeira *Premise* verificaria se a forma de pagamento selecionada foi a prazo. Neste caso é necessário validar o limite de crédito disponível para o cliente, o qual faria parte da segunda *Premise* da *Rule* em questão. A terceira e última *Premise* validaria o tipo de

desconto concedido para o cliente em questão (dentro os 20 possíveis tipos de descontos).

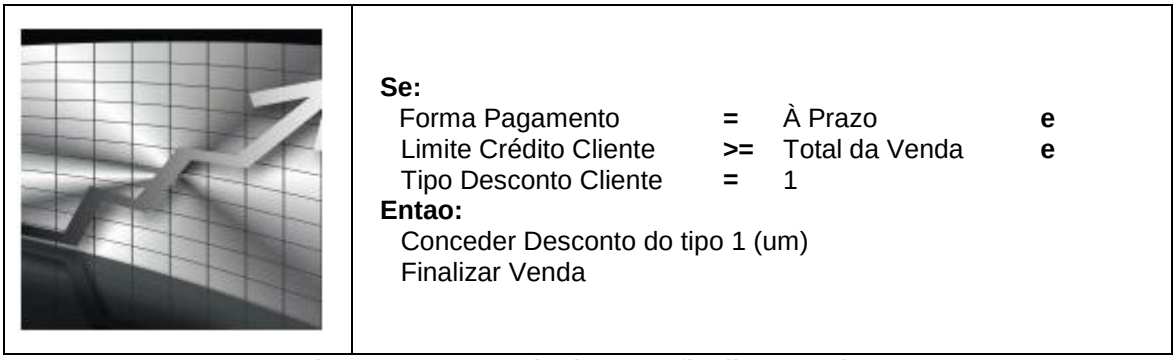


Figura 62 – Exemplo de regra finalizar vendas

Conforme descrito no parágrafo anterior, existem 20 tipos de descontos que poderão ser concedidos aos clientes. Neste caso, para cada tipo de desconto será criado uma *Rule* correspondente, conforme apresentada na Figura 62. Desta forma, após a satisfação das *Premises* supracitadas, a *Rule* em questão concederia o desconto do pedido de vendas para o cliente e por fim finalizaria sua venda. Neste caso o método denominado finalizar venda gravaria os dados em arquivo.

Ainda a *Rule* esboçada pela Figura 63 validaria a inclusão de um determinado produto aos itens de compra do respectivo cliente. Assim, a cada *FBE Product* inserido, a *Rule* em questão verificaria se a data de validade do produto é maior que a data atual e se o estoque atual do produto é maior ou igual a quantidade solicitada pelo cliente. Após a satisfação das *Premises* supracitadas, o método responsável por adicionar o produto à lista de produtos seria invocado.

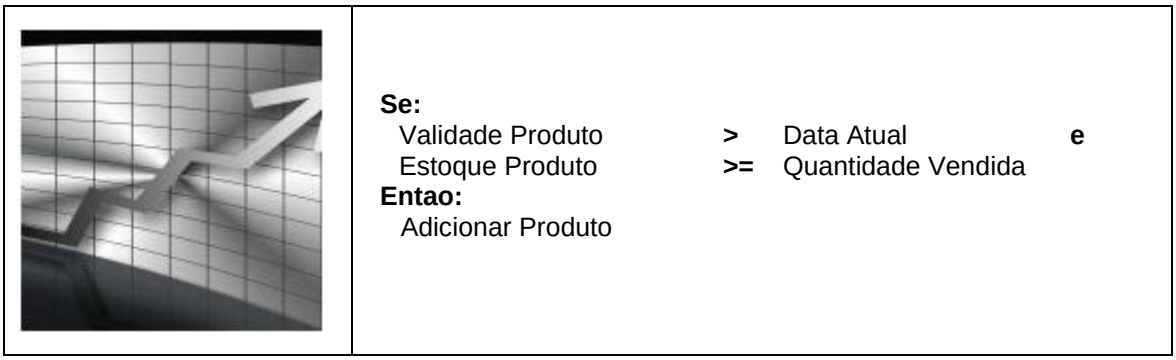


Figura 63 – Exemplo de uma regra adicionar produto

Em todas as etapas dos experimentos foram criados vários pedidos de venda com o mesmo conteúdo, ou seja, mesmo cliente, cinco itens de produto e a mesma

forma de pagamento (a prazo). Ainda, foi adicionado um indicador de tipo de cliente que oferece 20 (vinte) tipos distintos de classificação. Este indicador provê um desconto especial ao término de cada venda efetuada. Por fim um total de 48 regras foi criado, onde, neste cenário, 21 (vinte e uma) regras são ativadas e executadas. De modo a formalizar esta sequência de execução, o diagrama de sequência da Figura 64 foi elaborado.

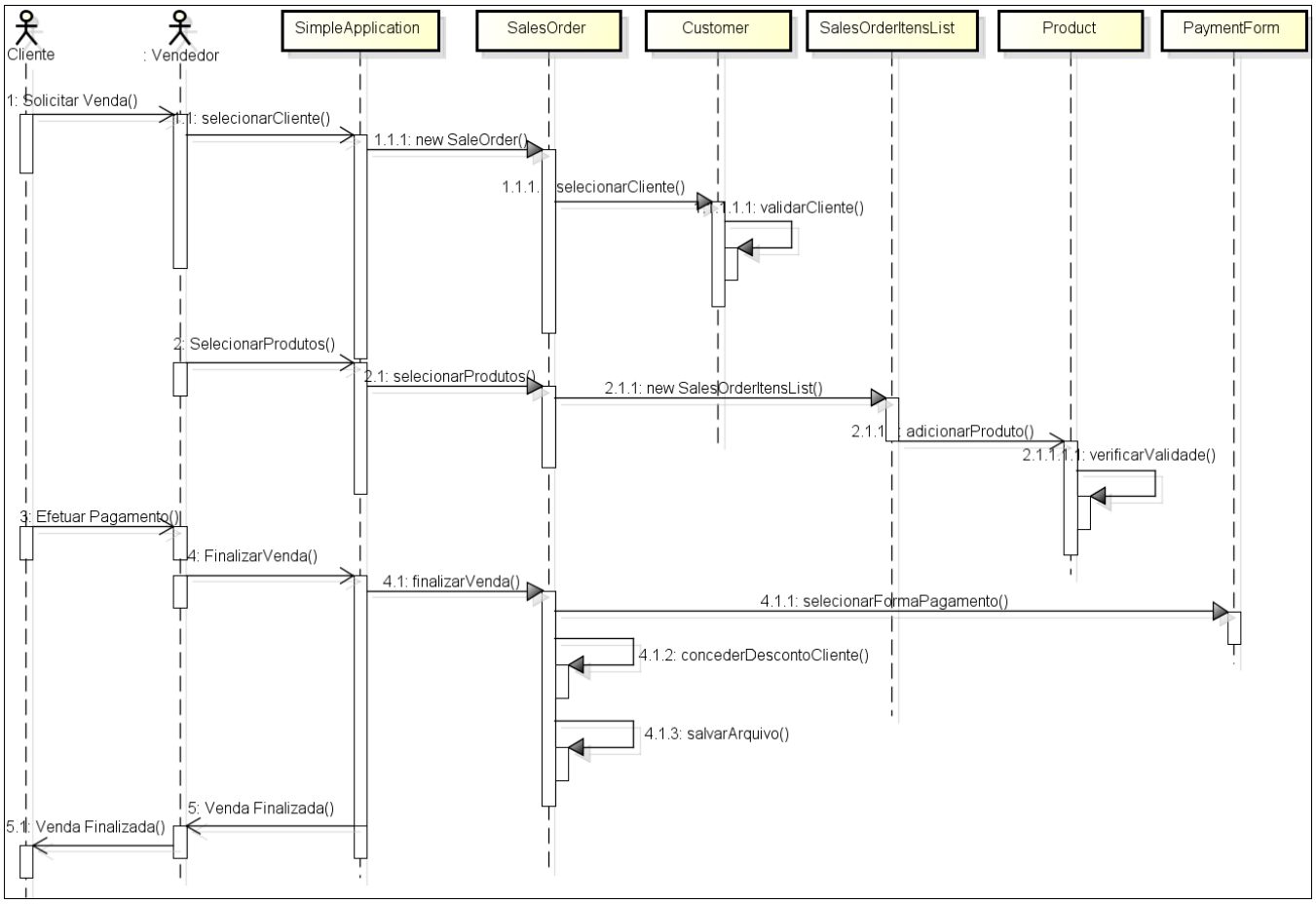


Figura 64 – Diagrama de sequência Vendas

Esse cenário foi executado para a versão do sistema puramente desenvolvida sobre as duas versões do *Framework* PON. Foram criados 100 (cem), 1000 (mil) e 10000 (dez mil) pedidos para os casos de teste, sendo realizadas três repetições/iterações do mesmo experimento para garantir a veracidade dos dados (BATISTA, *et. al.*, 2011). Ainda, os primeiros experimentos foram executados em um *PC* com processador *Core 2 Duo* com 3 *Gigabytes* de memória *RAM*. Para que fosse possível executar os testes de maneira mais idônea, o ambiente utilizado foi o *prompt* de comando (*Ms-Dos*) sendo executado em modo de segurança do *Microsoft*

Windows 7 com o carregamento mínimo de serviços de terceiros. A Figura 65 informa os valores obtidos em milissegundos após os testes de desempenho aplicados às duas versões do *Framework* PON, sobre as estruturas *PONLIST* e *PONVECTOR* do *Framework* Atualizado.

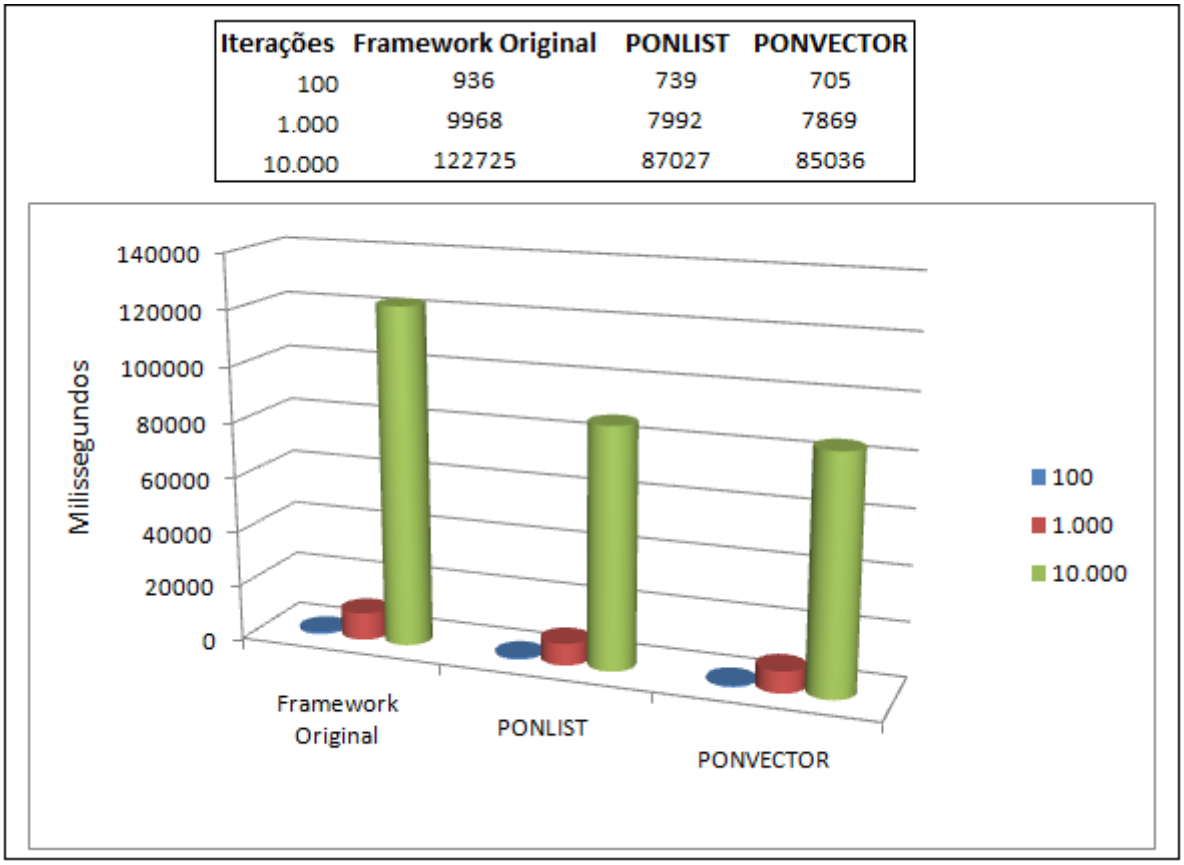


Figura 65 – Tempo de execução Vendas no SO Windows - adaptado de (VENÂNCIO *et al.*, 2011)

Ainda o mesmo aplicativo foi executado em um *PC Core i5*, 6 *GB* de *RAM* com o sistema operacional *Linux Debian*. A Figura 66 informa os valores obtidos em milissegundos após os testes de desempenho aplicados a versão original do *Framework* PON e sua versão atualizada, sobre as estruturas *PONLIST*, *PONVECTOR* e *PONHASH*.

Iterações	Framework Original	PONLIST	PONVECTOR	PONHASH
100	589	325	269	1290
1.000	7589	4269	3533	15890
10.000	69875	53259	50489	114567

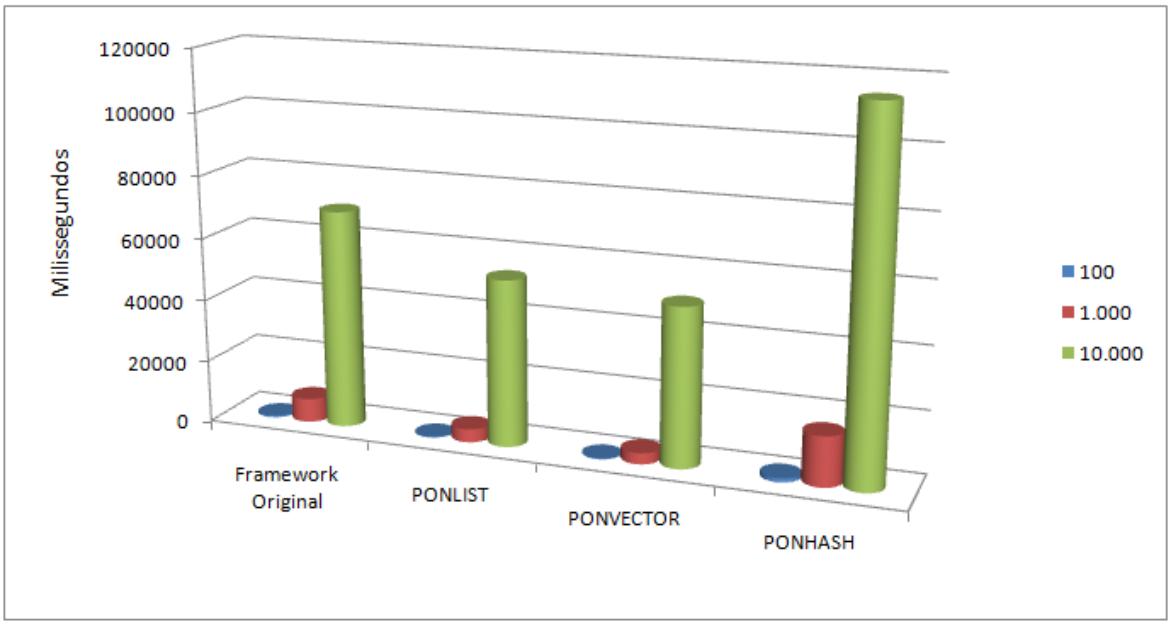


Figura 66 – Tempo de execução Vendas no SO Linux - *Hash*

Neste aplicativo, de cunho corporativo, observa-se que as estrutura de dados *PONLIST* e *PONVECTOR*, são melhores em desempenho em aproximadamente 54% do tempo de processamento em relação ao artefato original do *Framework* PON, o qual corresponde um ganho de desempenho de cerca de 2 (duas) vezes. Este cálculo é realizado a partir da média de todos os tempos de execução obtidos em 100, 1.000 e 10.000 iterações da aplicação Vendas. Entretanto os piores tempos são apresentados na Figura 66 pela estrutura *PONHASH* em relação às demais comparações. Apesar da aplicação possuir um *Attribute* que determina um tipo especial de desconto a um respectivo cliente, este *Attribute* somente muda de estado ao final do processamento da venda, conforme observado pela chamada do método "*concederDescontoCliente*" no diagrama de sequência da Figura 64.

É importante ressaltar que o sistema de Vendas descrito nessa seção, possui uma relativa implementação sobre a PI, especificamente em métodos onde é realizada a implementação do caso de uso denominado "Manter Pedido em Arquivo" da Figura 59. Isso de fato influenciou nos tempos obtidos, uma vez que o processo de escrita em disco é comprovadamente oneroso em questões de desempenho em relação ao acesso de dados em mémoria.

Ainda, o aplicativo em questão foi comparado em termos de desempenho com a sua versão implementada na PI, especificamente através do sub-paradigma de programação OO (BATISTA, *et. al.*, 2011) (SIMÃO, *et. al.*, 2012)[3].

3.11 SIMULADOR *PACMAN*

De modo geral, este caso de estudo consiste na implementação de um simulador com características do clássico jogo *Pacman* [4] (PITTMAN, 2011). Tal qual o jogo de inspiração, o ambiente possui corredores que formam um labirinto, limitando as ações de movimento dos personagens no cenário, nomeadamente o *Pacman* e seus inimigos, os Fantasmas. Ainda, os personagens do simulador apresentam comportamento autônomo e predeterminadado, ou seja, não são controlados por um usuário (RONSZCKA *et al.*, 2011). A Figura 67 ilustra o ambiente gerado pelo simulador.

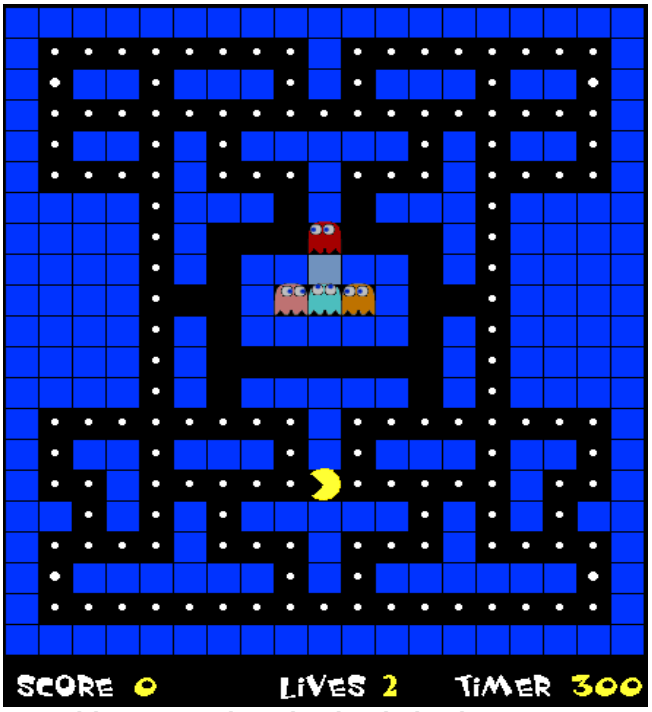


Figura 67 – Ambiente gerado pelo simulador (RONSZCKA *et al.*, 2011)

[3] Para fins de detalhamento os artigos em questão (BATISTA, *et. al*., 2011) (SIMÃO, *et al*. 2012b), encontram-se no Apêndice A e B, respectivamente.

[4] Os direitos autorais pertencem à pessoa que a produziu ou a empresa que publicou. O uso neste trabalho refere-se apenas para estudo acadêmico.

Para o *Pacman*, o objetivo principal do jogo é maximizar os pontos ganhos. O *Pacman* acumula pontos percorrendo os corredores do labirinto em busca de pastilhas, que são encontradas por todo o labirinto, dispondo de 300 passos para tal. Cada pastilha normal que o *Pacman* absorve lhe proporciona 10 pontos, enquanto que as pastilhas energizadoras lhe proporcionam 50 pontos. Ao absorver uma pastilha energizadora, os papéis do jogo se invertem por um determinado tempo, onde o *Pacman* deixa de ser a presa e passa a ser o predador (RONSZCKA *et al.*, 2011).

No período de latência, após a absorção de uma pastilha energizadora, os Fantasmas ficam assustados e assumem o papel de presa. Com isso, procuram evitar o contato com o *Pacman*. Neste período, o *Pacman* pode acumular pontos colidindo com cada um dos Fantasmas assustados. Ao colidir com um Fantasma assustado, ele recebe 200 pontos. A cada colisão adicional no mesmo período de latência, ele acumulará duas vezes mais pontos do que a colisão anterior. Desta forma, as colisões adicionais valerão 400, 800 e 1.600 pontos, respectivamente (RONSZCKA *et al.*, 2011).

A pontuação máxima possível atingida pelo *Pacman* é de 13.660 pontos. O *Pacman* tem um determinado tempo para acumular o maior número de pontos possíveis, dispondo de três vidas para tal. Fora do período de latência, cada colisão entre o *Pacman* e um Fantasma, resulta na perda de uma de suas vidas e no reposicionamento dos personagens em suas posições iniciais no *grid* (RONSZCKA *et al.*, 2011). Os casos de uso do aplicativo *Pacman* são apresentados no diagrama de casos de uso da Figura 68.

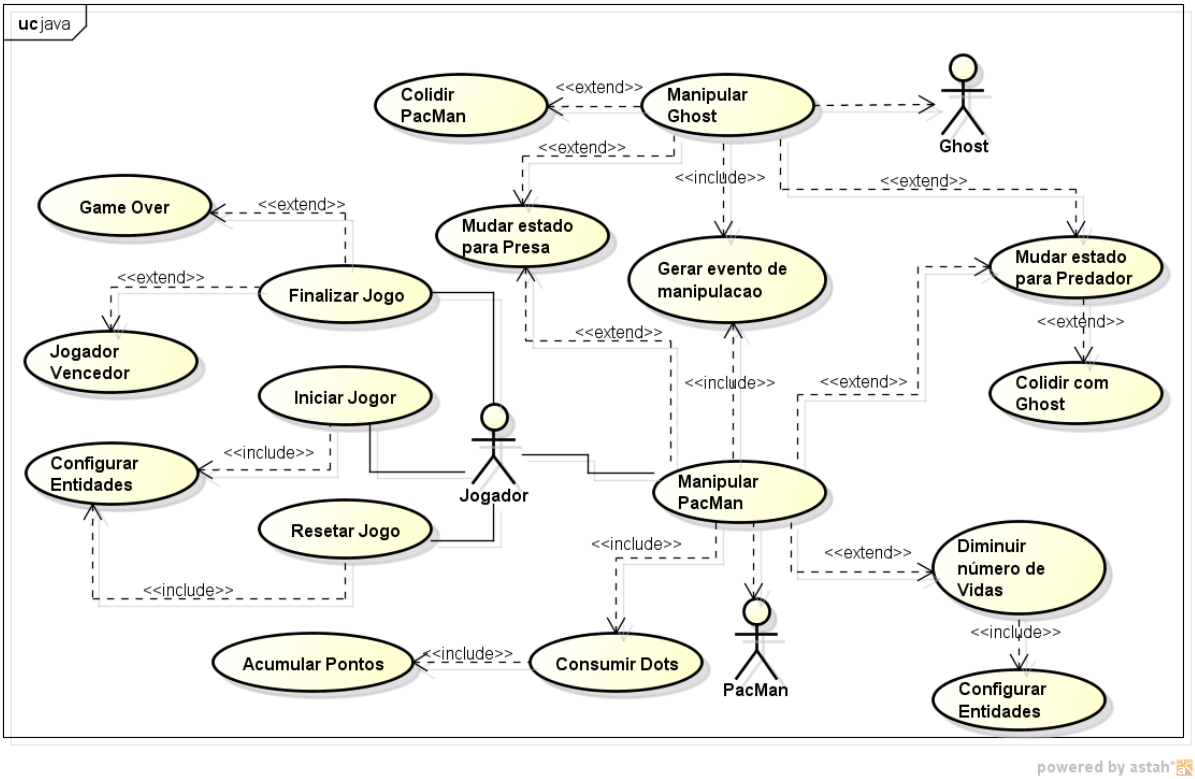


Figura 68 – Casos de uso *Pacman*

Particularmente, neste simulador, os personagens possuem um campo visual que representa a profundidade com que eles enxergam os corredores do labirinto. A Figura 69 ilustra um exemplo de campo visual com profundidade máxima, de valor 5. A visão dos personagens abrange todas as direções, sendo que o alcance de visão desses é representado por círculos, limitando-se ao se deparar com o fim de um corredor (RONSZCKA *et al.*, 2011).

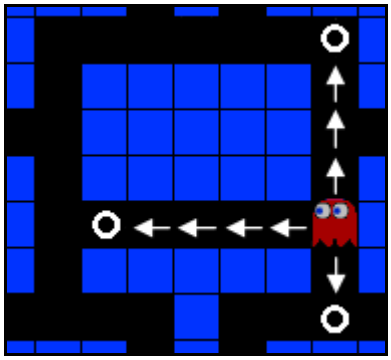


Figura 69 – Campo visual com profundidade 5 (RONSZCKA *et al.*, 2011)

O simulador apresenta particularidades que beneficiam a implementação de regras para a movimentação dos personagens nos corredores do labirinto. Dentre tais particularidades, tem-se a classificação de esquinas em categorias. As esquinas

representam o encontro ou cruzamento de dois ou mais corredores que compõem o labirinto, cada qual com seu formato distinto (RONSZCKA *et al.*, 2011).

O labirinto é formado por 9 diferentes formatos de esquinas, com o intuito de minimizar a quantidade de regras, visto que o tratamento das ações dos personagens se baseia nessa classificação e não em todas as partes do labirinto. Conforme ilustra a Figura 70, cada formato particular de esquina é representado por um valor, presente em uma escala de 1 a 9 (RONSZCKA *et al.*, 2011).

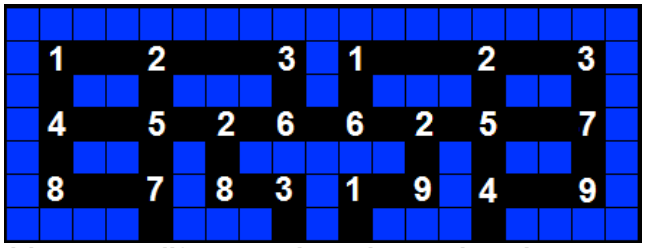


Figura 70 – Labirinto em 9 diferentes tipos de esquinas (RONSZCKA *et al.*, 2011)

Conforme apresentado na Figura 69, os personagens possuem certa capacidade de visão que lhes permite perceber os elementos que compõem o labirinto. Por exemplo, o *Pacman* enxerga pastilhas, paredes, Fantasmas e corredores vazios. Similarmente, os Fantasmas enxergam paredes, corredores vazios, o *Pacman* e os demais Fantasmas. Desta forma, baseado em sua visão, os personagens devem tomar decisões coerentes para se moverem pelos corredores do labirinto em busca de alcançar seus objetivos (RONSZCKA *et al.*, 2011).

As ações de movimentação adotadas pelos personagens estão ligadas aos elementos percebidos em seus campos visuais, por meio dos corredores do labirinto. Para cada um dos personagens, eventos são disparados à medida que esses detectam elementos em seus campos visuais. Normalmente, mais de um elemento é detectado a cada vez que um personagem se encontra em uma esquina. De forma a evitar conflitos, somente um evento pode ser disparado a cada momento, sendo necessário definir uma escala de prioridades para tal. Os eventos que incitam as regras que movimentam o *Pacman* apresentam a seguinte ordem de prioridade (RONSZCKA *et al.*, 2011):

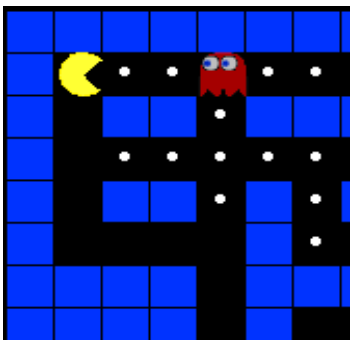
- Fantasma em estado normal detectado: é o evento com a maior prioridade; ele é incitado quando o *Pacman* avistar um Fantasma em estado normal em seu campo visual;

- Fantasma em estado assustado detectado: é o evento com a segunda maior prioridade; ele é incitado no período de latência, após a absorção de uma pastilha energizadora, quando o *Pacman* avistar um Fantasma em estado assustado em seu campo visual;
- Pastilha detectada: este é o evento com a terceira maior prioridade; ele é incitado quando o *Pacman* avistar uma pastilha normal ou uma pastilha energizadora em seu campo visual;
- Corredor vazio: este é o evento com a menor prioridade; ele é incitado quando nenhum dos outros eventos tiver sido instigado.

Por sua vez, os eventos que incitam as regras que movimentam os Fantasmas apresentam a seguinte ordem de prioridade:

- *Pacman* detectado enquanto o Fantasma estiver em estado assustado: é o evento com a maior prioridade; ele é incitado quando o Fantasma, em estado assustado, avistar o *Pacman* em seu campo visual;
- *Pacman* detectado enquanto o Fantasma estiver em estado normal: é o evento com a segunda maior prioridade; ele é incitado quando o Fantasma, em estado normal, avistar o *Pacman* em seu campo visual;
- Outro Fantasma detectado: é o evento com a terceira maior prioridade; ele é incitado quando o Fantasma avistar outro Fantasma no mesmo corredor em seu campo visual.
- Corredor vazio: este é o evento com a menor prioridade; ele é incitado quando nenhum dos outros eventos tiver sido instigado.

Os personagens só alteram sua direção quando estiverem sobre uma esquina, podendo retornar ao caminho pelo qual vieram ou escolher outro corredor para seguir. A Figura 71 ilustra exemplos de regras de movimentação para o personagem *Pacman* (RONSZCKA *et al.*, 2011).



Se:
esquina atual = 1 **e**
evento detectado = fantasma [direita] **e**
evento detectado = pastilha [direita] **e**
Entao:
"Fugir do fantasma movendo-se para baixo"
Se:
esquina atual = **1** **e**
evento detectado = pastilha [direita] **e**
Então:
"Absorver pastilha avançando para direita"

Figura 71 – Regras de movimentação, adaptado de (RONSZCKA *et al.*, 2011)

De fato, as regras que movimentam os personagens nos corredores do labirinto são baseadas nas percepções desses. Tais percepções representam a esquina atual, os eventos detectados ordenados por prioridade e a localização (direção) do elemento prioritário detectado (*i.e.* no caso da Figura 71, a primeira regra é composta pelo evento de maior prioridade, disparada do corredor à direita) (RONSZCKA *et al.*, 2011). Ademais, as regras que movimentam os personagens em cada situação particular são predefinidas pelo usuário do simulador. Ainda, de modo a facilitar o entendimento do fluxo principal de execução da aplicação *Pacman,* o diagrama de atividades da Figura 72 esboça o núcleo de sua execução.

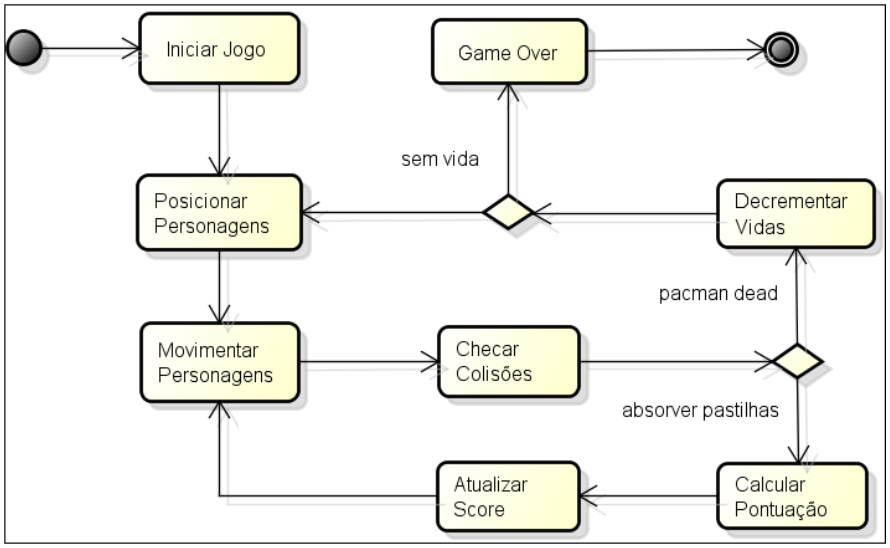


Figura 72 – Diagrama de Atividades *Pacman*

Ainda, para apresentar algum detalhamento de implementação da aplicação Simulador *Pacman*, um diagrama de classes simplificado dele é ilustrado na Figura 73. Ademais, devido às dimensões do diagrama original, o diagrama de classes em

questão abstrai a complexidade do simulador, externando a essência do mesmo por meio de suas classes principais.

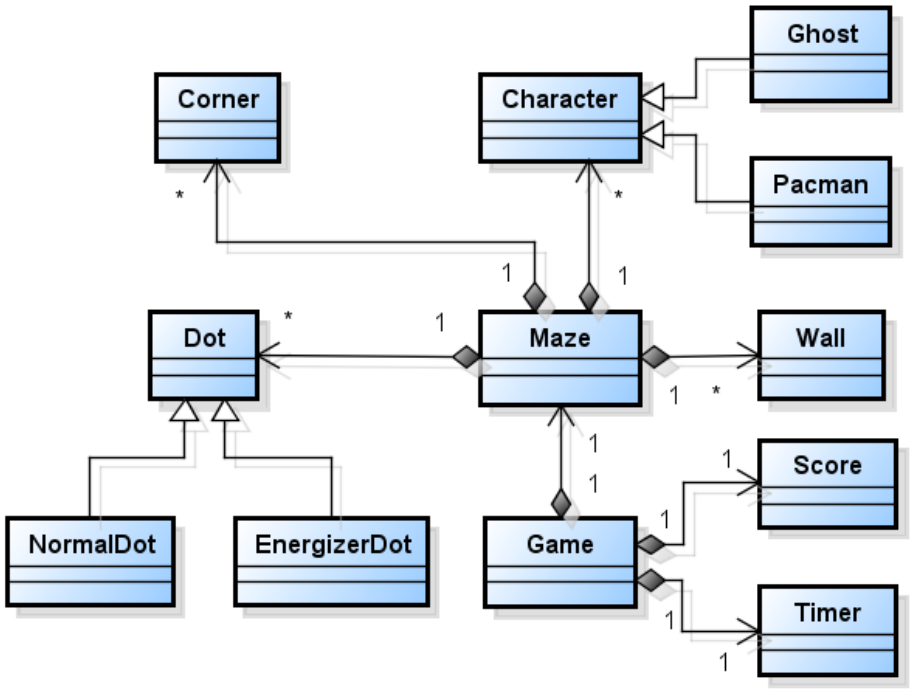


Figura 73 – Diagrama de classes do simulador do jogo, adaptado de (RONSZCKA *et al.*, 2011)

A classe *Game* comporta a estrutura de funcionamento do simulador, composta pelas classes *Maze*, *Score* e *Timer*. A classe *Maze* representa a estrutura do Labirinto, formado por Paredes (*Walls*), Esquinas (*Corners*), Pastilhas (*Dots*) e Personagens (*Characters*). O cerne do simulador, por sua vez, é constituído pelas *Rules* que movimentam os personagens nos corredores do labirinto, concentradas nas classes *Ghost* e *Pacman*.

Na implementação realizada no PON, cada *Rule* que colabora para movimentar os personagens é composta por uma *Condition* e por uma *Action*. A *Condition* representa o conjunto de percepções do personagem em um dado momento, com as *Premises* que avaliam a respeito da esquina atual, do elemento de maior prioridade detectado e da localização de tal elemento. A *Action*, por sua vez, é composta pela *Instigation* que ativa determinado *Method*, resultando na movimentação adequada do personagem para aquele conjunto de *Premises* aprovado.

O Algoritmo 27 externa a essência de composição de programas com as entidades colaboradoras do PON. Como dito anteriormente, as classes *Ghost* e

Pacman são responsáveis pela movimentação dos personagens nos corredores do labirinto e, portanto, todas as *Rules* do PON se encontram nela.

***Rule* 1: *Premise* –** Esquina = 1
***Premise* –** Campo visual = Fantasma em estado normal
***Premise* –** Direção do elemento detectado = Direita
***Instigation* –** Instiga *Method* moverParaBaixo()
***Rule* 2: *Premise* –** Esquina = 1
***Premise* –** Campo visual = Fantasma em estado assustado
***Premise* –** Direção do elemento detectado = Direita
***Instigation* –** Instiga *Method* moverParaDireita()
. . .
***Rule* 80: *Premise* –** Esquina = 9
***Premise* –** Campo visual = Nenhum elemento detectado
***Instigation* –** Instiga *Method* moverParaCima()

Algoritmo 27 – Algoritmo do simulador – 80 *Rules* PON interbloqueadas

Após a implementação do escopo da aplicação PACMAN previamente descrita, foram realizados experimentos. Estes experimentos foram realizados sobre o mesmo ambiente com os mesmos critérios de parada (*i.e.* fim dos 300 passos, fim das três vidas do Pacman ou fim das 150 pastilhas presentes no ambiente) para ambos os *Frameworks* do PON. Uma vez que as regras de movimentação seguem a mesma estrutura condicional, o resultado de cada execução do jogo reflete em movimentações idênticas para todos os personagens em todos os experimentos.

Ainda, os experimentos foram executados em um *notebook* com processador *AMD Turion X2* com 3 *Gigabytes* de memória *RAM*. De maneira a evitar resultados imprecisos com sistemas operacionais que apresentam maior número de preempções, os testes foram executados em um sistema operacional *Linux* versão *Debian*. Ademais, o ambiente *Linux* oferece medidas de tempo mais precisas e, portanto, a unidade de medida apresentada na Figura 74 é milissegundos, com as casas decimais representando os microssegundos. Para analisar a eficiência das diferentes versões do simulador, foram realizadas 100 repetições para cada tipo de experimento, em 10, 100 e 1000 execuções. Assim, a eficiência apresentada na Figura 74 representa a média de todos os experimentos gerados pelas 100 repetições.

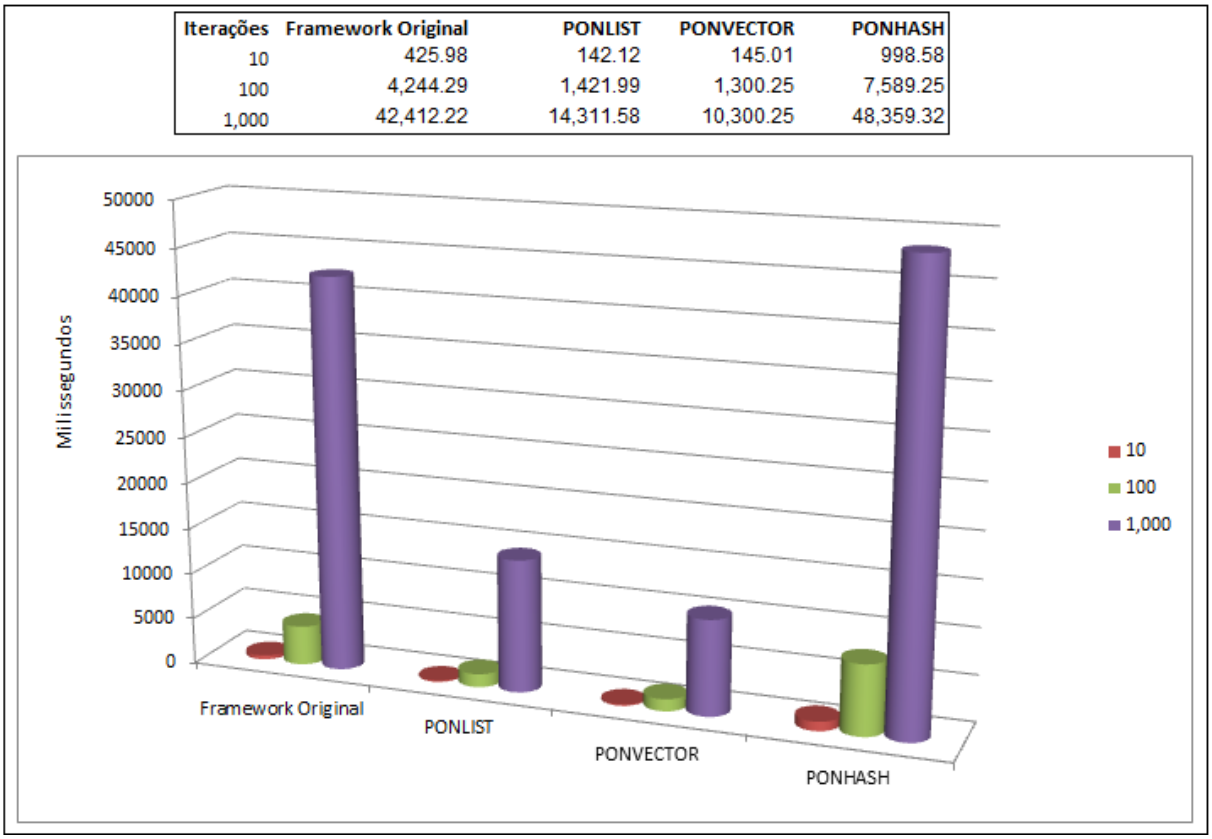


Figura 74 – Tempo de execução *Pacman*

Observa-se na Figura 74 a diferença de desempenho de quase 3 (três) vezes encontrado nos testes realizados no aplicativo *Pacman*, sobre as estruturas *PONLIST* e *PONVECTOR* em realação ao artefato original do *Framework* PON, o qual corresponde cerca de 33% do tempo de processamento. A estrutura *PONHASH*, por sua vez, foi extramemente ineficiente em sua execução. Isto se dá porque neste caso não há situações favoráveis a sua utilização. Ainda para todos os casos, os testes foram executados com o tempo médio de várias iterações sobre a execução da aplicação.

Por fim, o aplicativo em questão foi comparado em termos de desempenho com a sua versão implementada na PI, especificamente através do sub-paradigma de programação OO (RONSZCKA, *et. al,* 2011). Neste experimento, foi utilizada a versão dita como original do *Framework* PON, concebida nos trabalhos de BANASZEWSKI (2009), bem como sua nova versão. Entretanto, apesar da nova versão ser em média 3 (três) vezes mais rápido que sua versão precedente, ela obteve tempos um tanto piores em relação a implementacao no POO. Um dos

principais motivos dessa ineficiência se deveria ao fato da implementação do jogo *Pacman* no PON, não ter sido concebida inteiramente sobre os princípios do PON[5].

3.12 TERMINAL TELEFÔNICO

O caso de estudo proposto consiste na implementação de um simulador de sistema de telefonia. Este simulador possui as seguintes características (LINHARES, *et. al.,* 2011):

- O sistema é composto por 2 terminais telefônicos simulados, operados por meio de comandos de texto enviados via portas seriais RS-232, e por 2 terminais telefônicos internos operados por meio de simulação de eventos.
- Cada terminal telefônico processa os seguintes eventos externos, gerados através de comandos: comando de retirada do gancho (*off hook*); comando de recolocação no gancho (*on hook*); comando de digitação (*digit*); envio de caracter através de conexão (*send*).
- Cada terminal telefônico processa os seguintes eventos internos, gerados automaticamente: tempo limite para toque de chamada (*ring timeout*); tempo limite para conversação em estado suspenso (*suspend timeout*). Um terminal entra no estado suspenso quando está em conversação, é o terminal de destino da chamada em curso e executa um comando de recolocação no gancho. A chamada não é finalizada durante este estado, podendo ser retomada caso o terminal de destino retire do gancho antes do tempo limite.
- As transições entre os estados e os eventos que as ativam devem ocorrer conforme a máquina de estados da Figura 75.
- Todos os eventos, tanto internos quanto gerados via comando, devem ser registrados internamente para posterior análise do desempenho de execução.
- Em um dado instante de tempo, cada terminal se encontra em um dos seguintes estados: ocioso (*idle*); aguardando por dígito (*waiting for digit*); tocando (*ringing*); chamando (*calling*); em conversação, origem da chamada (*talking_src*); em

[5] Para fins de detalhamento os artigos em questão (RONSZCKA, *et. al.*, 2011) (SIMÃO *et. al*, 2012c), encontram-se no Apêndice C e D, respectivamente.

conversação, destino da chamada (*talking_dst*); ocupado (*busy_src* ou *busy_dst*); conversação suspensa (*suspended*).

- O sistema deve implementar um modo automático, no qual eventos são gerados automaticamente para os terminais internos. Este modo permite que o desempenho de processamento dos eventos seja melhor avaliado, uma vez que é eliminada a necessidade de que cada evento seja gerado por meio de uma mensagem manual enviada via porta RS-232.
- Cada terminal deve implementar os seguintes comandos de diagnóstico/relatório: obter a média do tempo de execução de cada comando pelo simulador, para cada estado possível do terminal que executou aquele comando; iniciar/parar o modo automático do sistema.

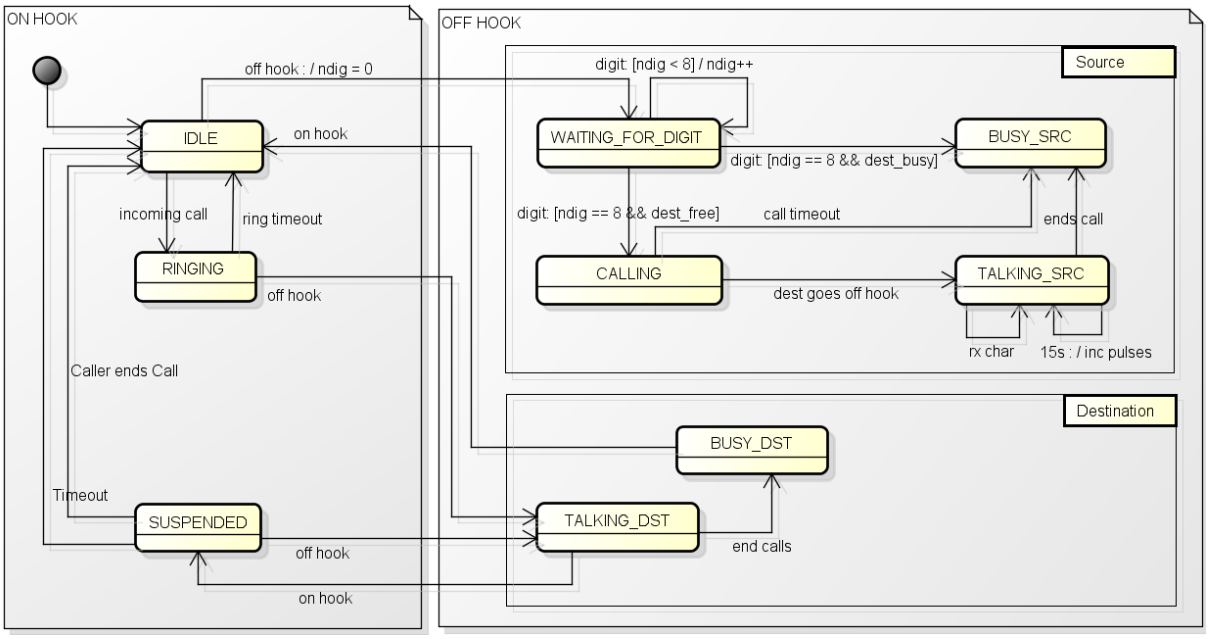


Figura 75 – Diagrama de Estados Terminal Telefônico, adaptado de (LINHARES, et. al., 2011)

Com base nesta especificação descrita, propôs-se o modelo estrutural em *UML*, na forma de um diagrama de classes, para a implementação do escopo descrito nos parágrafos anteriores e conforme apresentado na Figura 76 (LINHARES, *et. al.,* 2011).

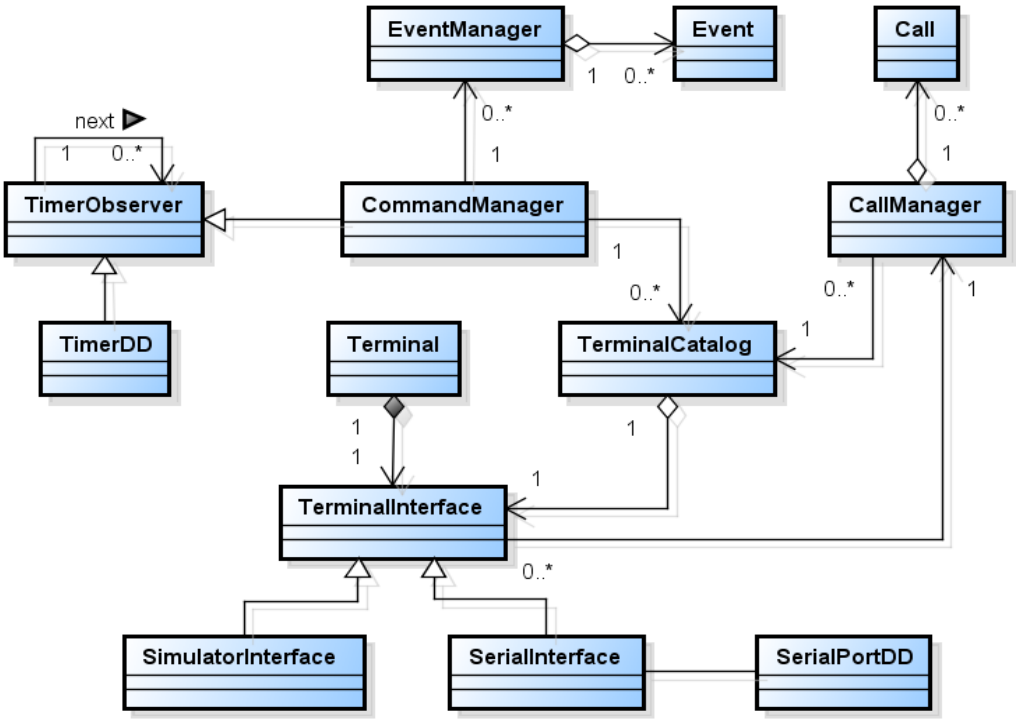


Figura 76 – Diagrama de Classes Terminal Telefônico, adaptado de (LINHARES, et. al., 2011)

A principal classe do modelo é a classe *Terminal,* que modela a entidade que se constitui em um terminal telefônico no sistema de telefonia. A classe *Terminal* possui uma relação de composição com a classe abstrata *TerminalInterface*, indicando que todos os terminais são compostos por um objeto que implementa a sua interface com o usuário, através da qual recebem comandos e enviam respostas (LINHARES, *et. al.,* 2011).

A classe *TerminalInterface* é estendida por duas classes concretas, *SimulatorInterface* e *SerialInterface.* A primeira implementa uma interface simulada, ou seja, de geração interna de comandos e eventos, sendo utilizada então pelos 2 terminais internos operados no modo automático. A segunda, por sua vez, implementa a comunicação e buferização do *driver* da interface serial física (instância da classe *SerialPortDD*) (LINHARES, *et. al.,* 2011).

As classes *TerminalCatalog* e *CallManager* gerenciam a efetivação e finalização de chamadas telefônicas. A classe *TerminalCatalog* é responsável por encapsular um *array* com os endereços dos objetos *Terminal* presentes no sistema, enquanto que a classe *CallManager* implementa os métodos de busca de estado do terminal de destino e encapsula objetos que descrevem uma chamada em curso (classe *Call*) (LINHARES, *et. al.,* 2011).

A classe *CommandManager* implementa o *parser* de comandos, tanto recebidos pelos terminais simulados quanto pelos terminais internos. Além disso, possui métodos para gerenciar o incremento periódico de pulsos, cuja temporização é determinada por um temporizador de *hardware* que é controlado por objeto da classe *TimerDD.* Este mesmo objeto oferece a temporização para a geração dos eventos *ring_timeout* e *suspend_timeout* junto aos terminais, o que ocorre dentro do método *UpdateTimers()* da classe *CTerminal* (LINHARES, *et. al.,* 2011).

Finalmente, as classes *EventManager* e *Event* e as enumerações e estruturas relacionadas são utilizadas para medição de tempo e armazenamento dos eventos que são gerados junto aos terminais. Além disso, elas oferecem métodos para a verificação da média e do desvio padrão do tempo de execução de cada evento, o que é utilizado para a avaliação de desempenho do sistema (LINHARES, *et. al.,* 2011).

Com base nesta descrição de classes identificaram-se 18 *Premises* e 23 *Rules* para a operação de cada par de terminais que efetuam uma conexão, conforme listado na Tabela 3. Ainda, as *Rules* identificadas (e seus constituintes) foram posteriormente utilizadas como base para a instanciação dos objetos colaboradores da cadeia de notificações na solução implementada segundo o PON (LINHARES, *et. al.,* 2011).

Tabela 3 – *Rules* Terminal Telefônico (LINHARES *et al.*, 2011)

Rules	***Conditions com suas Premises***	***Actions com suas Instigations***
R1	Src.Offhook && Src.State == Idle	Src.State ←Waiting_For_Digit
R2	Src.Digit && Src.State == Waiting_For_Digit	NDigRepos++
R3	Src.State == Dst_Found && Dst.State != Idle	Src.State ← Busy
R4	Src.State == Dst_Found && Dst.State == Idle	Src.State ← Calling, Dst.State ← Ringing
R5	Src.State == Calling && Dst.State == Ended_Dst	Src.State ←Busy
R6	Src.State == Calling && Dst.State == Ringing && Dst.Offhook	Src.State ←Talking_Src, Dst.State ←Talking_Dst
R8	Dst.State == Talking_Dst && Dst.Onhook	Dst.State ←Suspended
R9	Dst.State == Suspended && Dst.Offhook	Dst.State ←Talking_Dst
R10	Src.State == Ended_Src && Dst.State == Suspended	Src.State ←Idle, Dst.State ←Idle
R12	Dst.State == Busy && Dst.Onhook	Dst.State ←Idle
R13	Src.State == Talking_Src && Dst.State == Ended_Dst	Src.State ←Busy, Dst.State ←Idle
R14	Src.State == Waiting_For_Digit && Src.Onhook	Src.State ←Clear_Digit
R15	Src.State == Clear_Digit && NDigRepos > 0	NDigRepos--
R16	Src.State == Clear_Digit && NDigRepos == 0	Src.State ←Idle
R17	Src.State == Calling && Src.Onhook	Src.State ←Ended_Src

R18	Src.State == Busy && Src.Onhook	Src.State ←Idle
R19	Src.State == Talking_Src && Src.Onhook	Src.State ←Ended_Src
R20	Src.State == Ended_Src && Dst.State == Talking_Dst	Src.State ←Idle, Dst.State ←Busy
R21	Dst.State == Suspended && Dst.Susp_Timeout	Dst.State ←Ended_Dst
R22	Src.State == Ended_Src && Dst.State == Ringing	Src.State ←Idle, Dst.State ←Idle
R23	Dst.State == Ringing && Dst.Ring_Timeout	Dst.State ←Ended_Dst

O sistema simula dois terminais telefônicos envolvidos em uma chamada. Assim, o terminal de destino de uma conversação somente pode ser determinado após a identificação deste terminal em um catálogo de possíveis terminais de destino, o que ocorre com a avaliação dos dígitos que disparam a *Rule RC* e resultam na execução ou da transição *RC*.found ou da transição *RC.not_found*.

Portanto, pode-se afirmar que o *FBE* não é conhecido à priori. Por consequência, se o *FBE* correspondente ao terminal de destino não é conhecido, não é possível instanciar estaticamente os elementos da cadeia de notificações do PON que apresentem alguma dependência em relação a ele.

Sendo assim, a implementação em PON para o caso de estudo em questão prevê a criação de *Rules* dinamicamente a partir do momento em que se conhece qual o *FBE* correspondente ao terminal de destino, bem como a sua destruição a partir do momento em que uma chamada é encerrada. A seguinte abordagem é utilizada (LINHARES, *et. al.,* 2011):

- As *Rules R1, R2, R14, R15, R16, R18, RC* são instanciadas estaticamente para cada terminal.
- Todas as demais *Rules* são instanciadas dinamicamente por ocasião da execução de *RC*, imediatamente após o terminal de destino ter sido determinado.
- Todas as *Rules* instanciadas dinamicamente são destruídas pela execução de *R5, R10, R13, R20* ou *R22*. Estas *Rules* são dependentes das premissas *"State == Ended_Src"* ou *"State == Ended_Dst"*, portanto somente executadas quando uma chamada em curso é finalizada.

Outrossim, a versão PON do simulador de telefonia foi implementada para execução em uma plataforma embarcada. Ao contrário da implementação em plataforma de computador pessoal genérica, esta abordagem dispensa o uso de um sistema operacional e as contaminações nas medições de desempenho decorrentes

deste uso, dado que o sistema proposto apresenta uma interação mínima com dispositivos de *hardware* (somente portas seriais e *timer*). Isto permite que o *software* seja construído na forma de código binário iniciado imediatamente quando da inicialização do processador (*boot*) (LINHARES, *et. al.,* 2011).

A plataforma embarcada escolhida foi a placa eAT55 da *eSysTech*, que contém um processador *Atmel* AT91M55800A com núcleo *ARM7* baseado em *clock* de 33 *MHz*. Este processador integra, além de um núcleo *ARM7 RISC* de 32 *bits*, periféricos mapeados em memória que permitem a implementação do temporizador e das portas seriais necessárias para o funcionamento do sistema. A Tabela 4 informa os valores obtidos (em ciclos de *clock*) após os testes de desempenho aplicados as duas versões do *Framework* PON (LINHARES, *et. al.,* 2011).

Tabela 4 – Desempenho entre versões do *Framework* PON - Terminal Telefônico

	Evento	***Estado Inicial***	***Média de Eventos***	***Ciclos de clock***	
				Framework Original	***Framework Atualizado PONLIST***
Terminal	OFF HOOK	IDLE	51	221216	147495
Terminal	OFF HOOK	RINGING	2	513700	229324
Terminal	OFF HOOK	TALKING_DST	2	133238	053230
Terminal	OFF HOOK	BUSY	7	096992	027360
Terminal	ON HOOK	IDLE	62	103708	059485
Terminal	ON HOOK	WAITING_FOR_DIGIT	47	342974	232944
Terminal	ON HOOK	TALKING_SRC	1	549048	361516
Terminal	ON HOOK	TALKING_DST	2	342103	123522
Terminal	ON HOOK	BUSY	5	174020	100078
Terminal	DIGIT	WAITING_FOR_DIGIT	64	080228	039313
Média				**255722**	**137426**

Percebe-se que os números de ciclos de *clocks* gerados pela nova versão do *Framework* PON em média, sobre o viés da estrutura *PONLIST,* foram diminuídos pela metade, considerando-se assim cerca de 50% do tempo de processamento. O fato da aplicação Terminal Telefônico não possuir preempção externa, como as encontradas em execuções sobre Sistemas Operacionais (SO), garante ainda mais a veracidade das informações de desempenho obtidas. Uma vez que a aplicação roda diretamente via *hardware* via código binário.

Ainda, a respeito dos resultados obtidos, é pertinente observar que o *software* é inteiramente executado no processador *ARM7* da placa eAT55, incluindo as rotinas de interrupção do temporizador (para geração dos eventos de *timeout*) e das portas seriais. O tratamento assíncrono da interrupção das portas seriais não interfere de forma significativa no desempenho, visto que a banda efetivamente utilizada é pequena e a transmissão de dados é sincronizada com o fim do tratamento dos eventos (mensagens de *status* e estado atual).

Ademais, o aplicativo em questão foi comparado em termos de desempenho com a sua versão implementada na PI, especificamente através do sub-paradigma de programação OO (LINHARES, *et.al.,* 2011). Neste experimento, foi utilizada somente a nova versão do *Framework* PON. A versão implementada para o POO apresentou resultados de desempenho significativamente melhores do que os da versão implementada para o PON. Um dos fatores é o fato das implementações (POO e PON) não possuírem redundâncias temporais e estruturais significativas (LINHARES, *et.al.,* 2011).

Ademais, a implementação POO é orientada a eventos, ou seja, somente avalia o estado do terminal quando um evento ocorre e tipicamente altera este estado em função da ocorrência do evento. Além disso, a opção por avaliar o estado do terminal através da instrução *switch case* em C++ OO uma única vez dentro de cada método tratador de eventos, elimina a ocorrência de redundâncias estruturais (LINHARES, *et.al.,* 2011)[6]. De fato, o uso de *switch case* era aplicavél neste caso uma vez que as relações causais visadas não eram complexas em termos de elementos avaliados.

Outrossim, nos trabalho de (WITT *et. al.*, 2011) o aplicativo Terminal Telefônico foi reimplementado em *hardware,* tanto no PON quanto na POO. O desempenho dos sistemas POO e PON foi avaliado por meio da geração automática de eventos para cada um dos dois terminais telefônicos simulados. Cada evento teve o seu tempo de processamento medido por meio da leitura de contadores implementados na *Field-Programmable Gate Arrays* (*FPGA*). O armazenamento dos valores dos contadores é realizado de maneira a evitar-se ao máximo qualquer

[6] Para fins de detalhamento o artigo em questão (LINHARES, *et.al.,* 2011), encontra-se no Apêndice E.

overhead ou latência na medição dos tempos, permitindo uma avaliação mais precisa do desempenho.

Verificou-se que a solução PON foi muito mais intuitiva e fácil de ser composta em *hardware* do que a solução POO. Isto se dá justamente pela semelhança do PON com o *hardware*, principalmente no paralelismo e na simples correlação entre as entidades PON e as entidades em *hardware*. Ademais, os resultados comparativos demonstram que a solução PON apresenta melhor desempenho que a POO. Ainda, esta vantagem aumenta quando o número de condições causais for grande, uma vez que PON otimiza tais avaliações causais. (WITT *et. al.*, 2011).

Por fim, ressalta-se que não foram feitos esforços de otimização das avaliações causais em POO com *hardware* procurando assim manter esforço similar a composição do sistema em PON com *hardware*. Isto difere da versão POO em *software* citada acima, a qual foi objeto de implementação enxuta. Isto dito, esta implementação POO em *software* poderia servir de base para outras implementações em *hardware* e comparações para com o PON em *hardware* em experimentos futuros.

3.13 CONCLUSÃO

Percebe-se, de fato, que após a mudança da estrutura de dados que comporta a cadeia de notificações, as devidas otimizações e as refatorações de código, obteve-se ganho de desempenho. Este ganho foi em média de 50% do tempo de processamento sobre os diferentes estudos de caso descritos neste capítulo, o qual corresponde um ganho de desempenho de no mínimo 2 (duas) vezes.

Em se tratando especificamente de *PONLIST* e *PONVECTOR* os tempos de desempenho são aproximados. Entretanto em todos os casos *PONVECTOR* é ligeiramente mais eficiente. Isto é causado pelo armazenamento sequencial dos elementos PON, o qual faz uso da chamada *memory pool* e consequente utilização de aritmética de ponteiros, de modo a realizar o processo de iteração sobre os elementos PON pertencentes a cadeia de notificações.

Sobre a estrutura *PONHASH*, é importante salientar que existem casos típicos onde sua utilização torna a execução de uma aplicação PON extremamente eficiente, conforme observado nos resultados da Figura 58. Estes casos são quando um respectivo *Attribute* possui em seu escopo muitas *Premises* e, ainda, o estado deste *Attribute* possua variações relativamente frequentes a cada iteração da aplicação PON.

Conforme discutido na seção 2.3.5, o cálculo assintótico do PON propiciava maiores otimizações do que aquela encontrada em sua versão dita como original, realizada pelos trabalhos de BANASZEWSKI (2009). Assim, a medida que as otimizações e refatorações deixaram o *Framework* PON otimizado, os tempos de execução das aplicações PON começaram a tender sobre aquilo que se esperava pelo observar do cálculo assintótico (particularmente quando o caso médio é considerado).

Neste âmbito, a direção dos trabalhos de otimização de execução e desempenho de aplicações PON, vão ao encontro da construção de um compilador próprio. Isto implicará, principalmente, na eliminação das estruturas de dados que comportam o orquestramento de notificações do PON. Presume-se que após sua implementação, mesmo em uma arquitetura monoprocessada (como *Von-Neumann* ou *Harvard*), as aplicações PON terão desempenhos bem superiores às encontradas atualmente.

4 FERRAMENTA *WIZARD* NO DESENVOLVIMENTO DE *SOFTWARE EM PON*

As ferramentas de geração de código para o apoio ao processo de desenvolvimento de *software* possuem diversas categorias, conforme descrito na seção 2.7.1. Dentre elas destacam-se as ferramentas *Wizards*, também conhecidas como assistentes. Estas ferramentas propiciam especificamente o desenvolvimento de *software* em alto nível para um determinado projeto alvo. Basicamente estes assistentes possuem formulários/telas desenvolvidos sobre interfaces amigáveis ao desenvolvedor de *software.* A partir de entradas de dados, o assistente gera o código fonte sobre uma linguagem de programação pré-definida. Neste âmbito, as próximas seções descrevem as melhorias propostas e implementadas para o assistente denominado *Wizard* PON, o qual tem por responsabilidade conceber aplicações PON em alto nível.

4.1 MELHORIAS NA FERRAMENTA *WIZARD* PON

Em se tratando de ferramentas para o apoio ao desenvolvimento de *software*, especificamente para o desenvolvimento de aplicações PON, a ferramenta inicialmente denominada *Wizard* CON, descrita na seção 2.5, foi aprimorada e generalizada para o PON. Sua melhoria e generalização tem o objetivo principal de conceber aplicações PON a partir de formulários/interfaces de alto nível e a geração automatizada de código fonte sobre respectivo *Framework* atualizado do PON. Desta forma, a ferramenta atual passou a ser denominada *Wizard* PON, referenciando-se explicitamente como uma ferramenta de apoio ao desenvolvimento de aplicações do PON.

Neste âmbito, o diagrama de casos de uso da Figura 77 esboça os novos casos de uso implementados para a ferramenta *Wizard* PON. Oportunamente, os casos de uso em amarelo correspondem aos novos casos de uso, enquanto os casos de uso em azul representam à melhoria de casos de uso existentes com relação à ferramenta prévia. A melhoria na ferramenta *Wizard* PON está relacionada

diretamente para atender o requisito de geração automatizada de código no projeto alvo. Neste caso, a geração de código fonte apoiado sobre a nova versão *Framework* do PON.

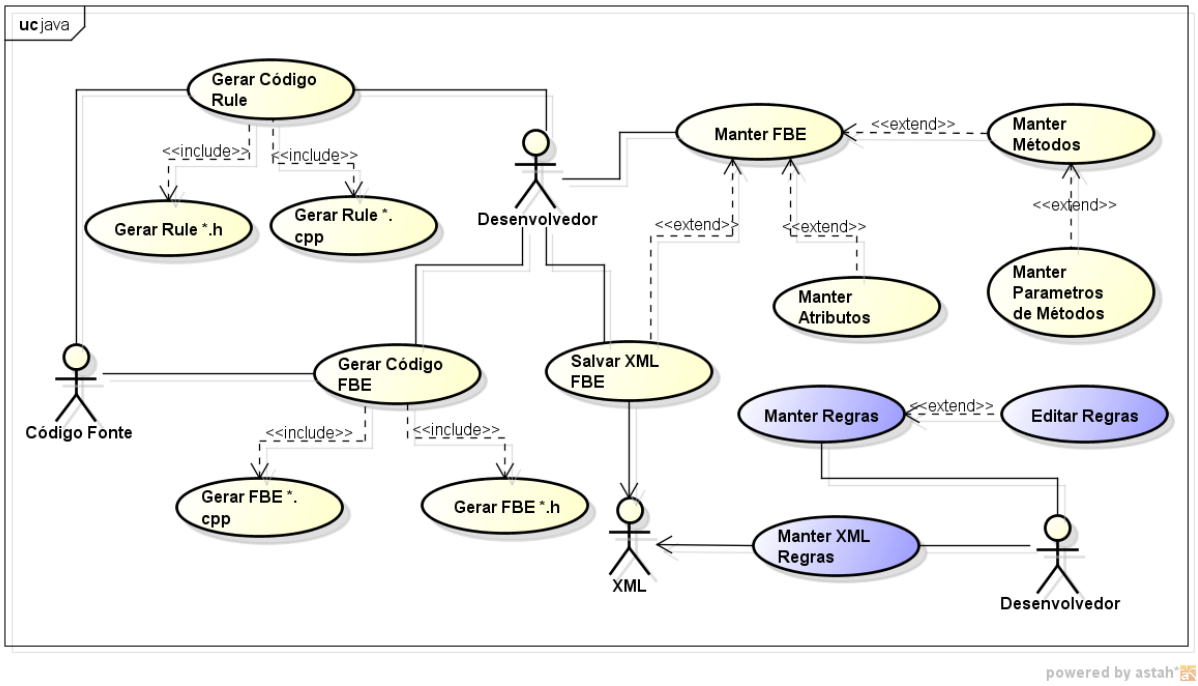


Figura 77 – Casos de uso implementados para a ferramenta *Wizard*

4.1.1 Caso de Uso - Manter *FBE*

A criação ou mesmo a edição de uma classe *FBE* é realizada por meio da interface esboçada pela Figura 78. O desenvolvedor inicialmente nomeará a classe *FBE*, logo após poderá definir seus *Attributes,* tipos de *Attribute*s, *Methods* e parâmetros de *Methods*. Ainda, o desenvolvedor poderá adicionar o código do *Method* caso assim o desejar através da própria ferramenta *Wizard*. É importante ressaltar que neste momento o desenvolvedor criará uma classe derivada de *FBE,* a qual representa um subconjunto de elementos da base de fatos. Outrossim, é possível realizar a importação de *FBE*s em formato *XML* para o *Wizard* PON, para sua posterior manipulação nesta ferramenta.

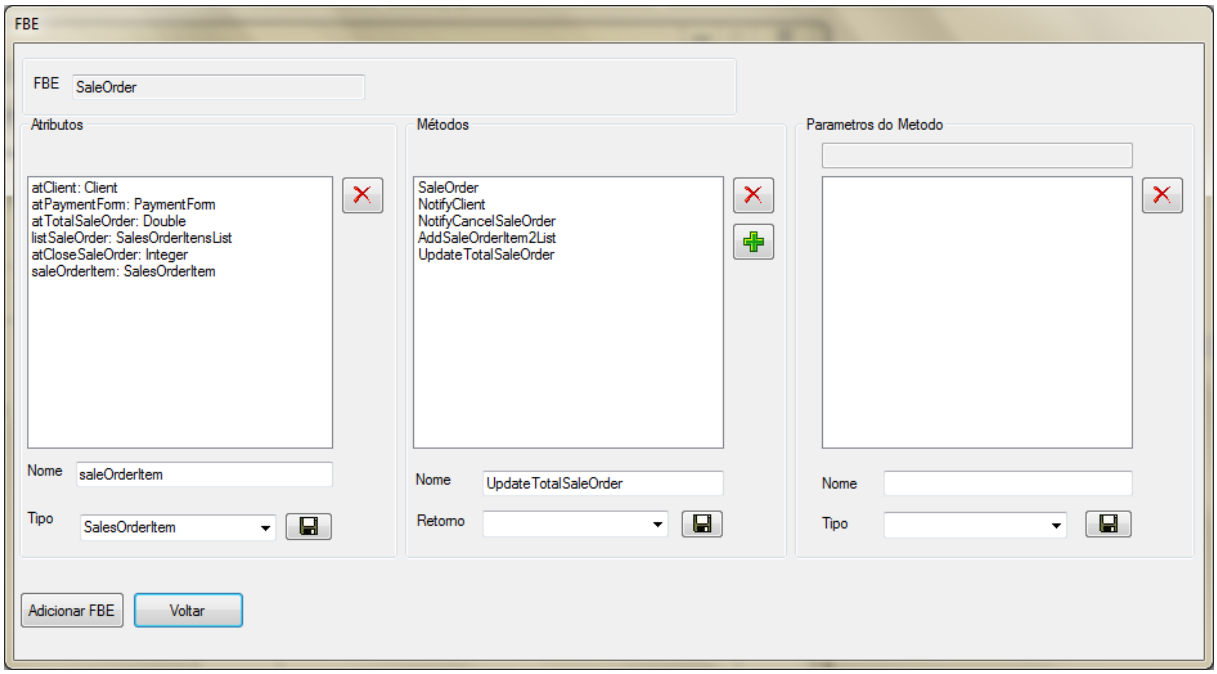


Figura 78 – Manter *FBE Product*

A classe derivada de *FBE* da Figura 78 refere-se à classe *FBE SaleOrder* do Sistema de Vendas previamente apresentado na seção 4.2. Os seus *Attributes,* os quais representam o estado de um *FBE*, são compostos pelo cliente (*atClient*), forma de pagamento (*atPaymentForm*), valor total da venda (*atTotalSaleOrder*), lista de produtos (*listSaleOrder*), atributo que representa o fechamento de uma respectiva venda (*atCloseSaleOrder*) e, item (produto e quantidade) de uma venda (*saleOrderItem*).

Os *Methods,* por sua vez, correspondem aos serviços disponibilizados pela classe *FBE SaleOrder.* Dentre eles se destacam o *Method notifyClient*, responsável pela notificação de venda ao respectivo cliente. O *Method notifyCancelSaleOrder*, responsável por notificar um cliente do cancelamento de uma venda. O *Method updateTotalSaleOrder,* o qual é responsável por atualizar o valor total da venda logo após a inserção de um *saleOrderItem* (produto e quantidade).

Ademais, é possível atribuir o respectivo código fonte a cada *Method* criado a partir da ferramenta *Wizard* PON, conforme observado pela Figura 79. Neste *Method,* a primeira linha atualizaria o total da venda e o atribuiria a variável "*totSale".* Ao passo que a linha subsequente, alteraria o estado do *Attribute atTotalSaleOrder* com o valor total da venda atualizada.

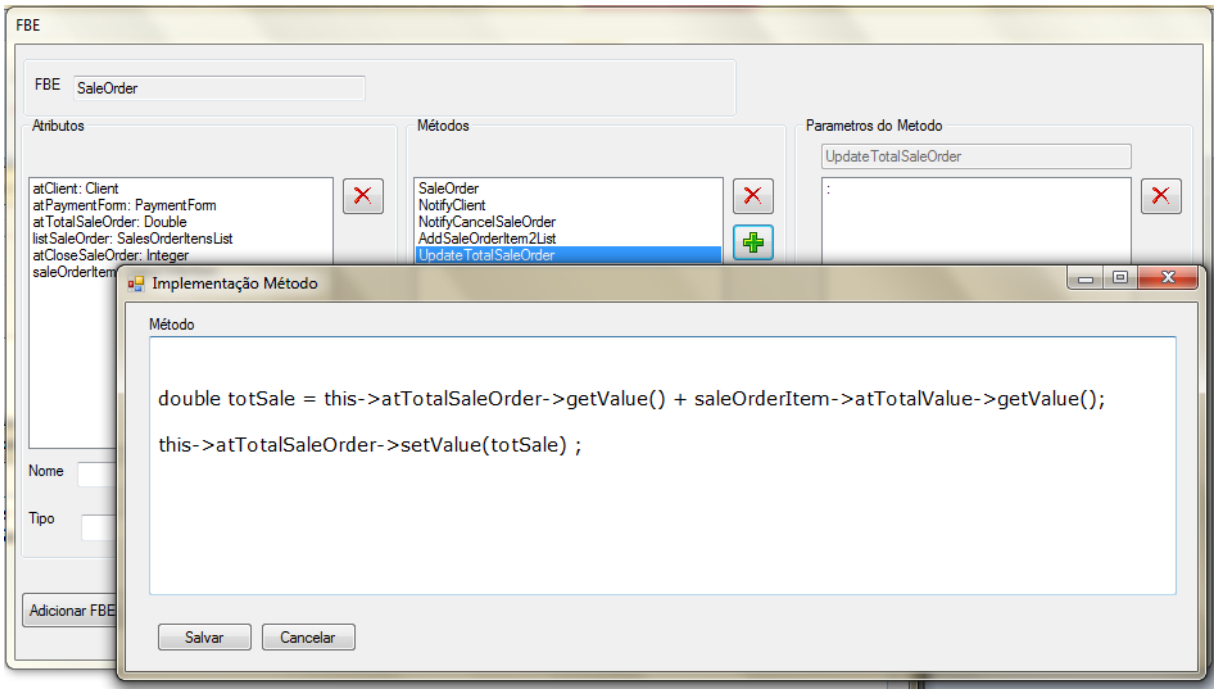


Figura 79 – Implementação de *Method* via *Wizard*

Após a criação da respectiva classe *FBE* o desenvolvedor poderá criar as instâncias dos elementos da base de fatos, conforme é observado na Figura 80. Assim, basta selecionar a classe *FBE* e logo após descrever o nome da instância do *FBE* propriamente dito. Por fim, basta clicar na opção de adição de *FBE* do formulário. Esta é a maneira pelo qual o desenvolvedor instancia os *FBEs* através da ferramenta *Wizard* PON. Ainda, seria possível visualizar os *Attributes* e *Methods* do *FBE* após selecioná-los, conforme apresentado pela Figura 80.

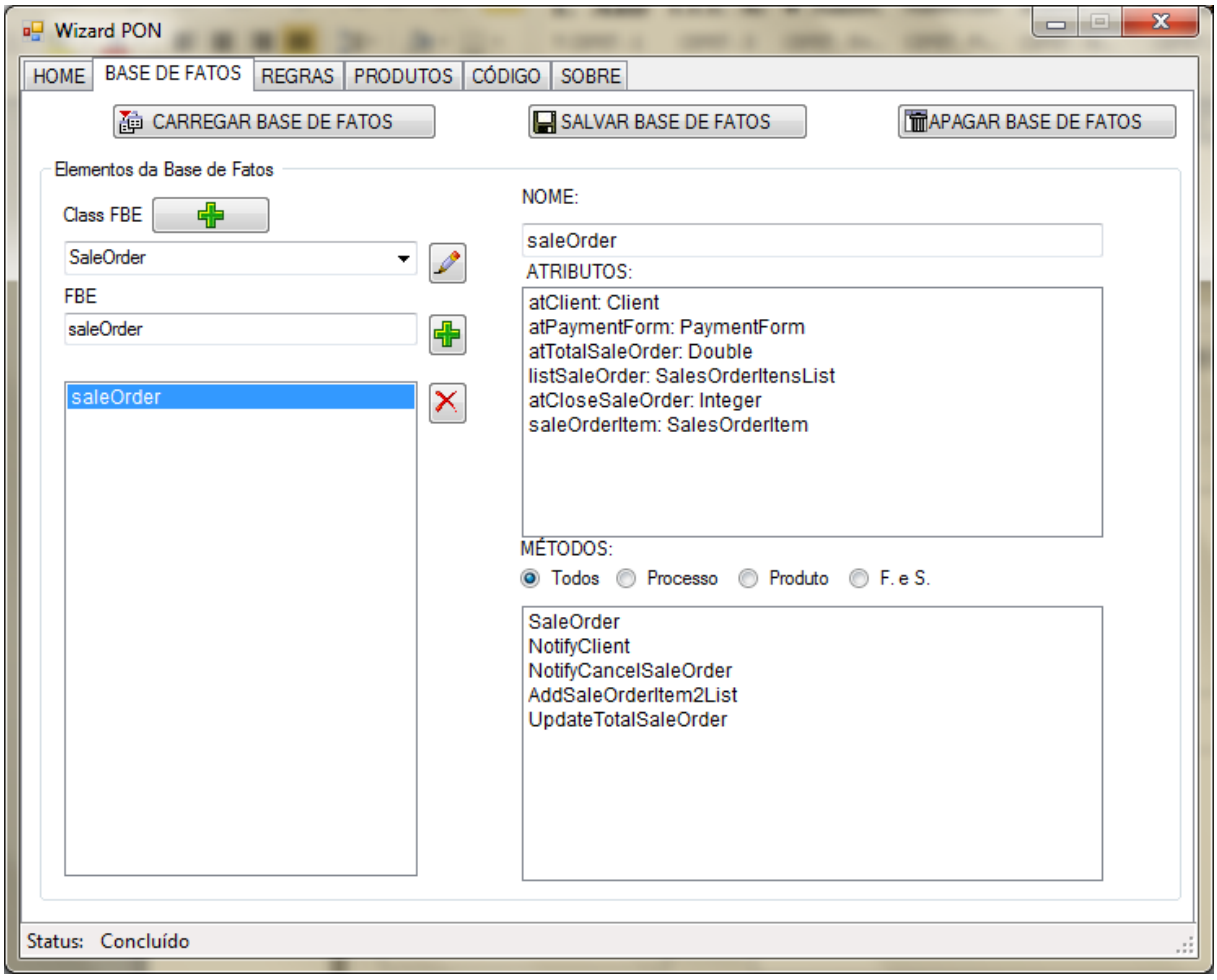


Figura 80 – Lista de *FBEs*, *Attributes* e *Methods*

4.1.2 Caso de Uso - Manter *XML FBE*

Após a criação e/ou edição dos *FBE*s o desenvolvedor poderá armazená-los em arquivos de formato *XML* para sua posterior utilização na ferramenta *Wizard* PON. O padrão *XML* salvo é demonstrado conforme mostra a Figura 81. Desta forma, conforme observado, todos os *Attributes*, *Methods*, parâmetros de *Methods* e implementações de *Methods* são armazenados em seus respectivos rótulos (*tags*) no arquivo *XML* em questão.

```
<FBEs>
    <FBE>
        <Nome>saleOrder</Nome>
        <Tipo>Object</Tipo>
        <Atributos>
            <Atributo>
                <Nome>atTotalSaleOrder</Nome>
                <Tipo>Double</Tipo>
            </Atributo>
            ....
            ....
        </Atributos>
        <Metodos>
            <Metodo>
                <Nome>UpdateTotalSaleOrder</Nome>
                <Implementacao>
                    double totSale = this->atTotalSaleOrder->getValue() + saleOrderItem->atTotalValue->getValue();
                    this->atTotalSaleOrder->setValue(totSale)
                </Implementacao>
                <Parametros>
                    <Parametro>
                        <Nome></Nome>
                        <Tipo></Tipo>
                    </Parametro>
                </Parametros>
            </Metodo>
            ....
            ....
        </Metodos>
    </FBE>
</FBEs>
```

Figura 81 – *XML FBE*

4.1.3 Casos de Uso - Manter Regras, Editar Regras e Manter *XML* Regras

A melhoria dos casos de usos “Manter Regras”, “Manter *XML* Regras” e “Edtar Regras”, referem-se principalmente a nova composição adotada para a criação e utilização das *Rules* no PON. Nesta versão, é possível que o desenvolvedor crie uma *Rule* no clássico formato “se-então” com maior riqueza de detalhes em relação a sua versão predecessora.

Primeiramente, é possível identificar o tipo da *Rule* em sua composição, neste caso definido pelos tipos *CONJUNCTION, DISJUNCTION* e *SINGLE.* Subsequentemente, é possível determinar a classe onde a respectiva *Rule* será implementada. Ademais, para a criação de *Premises* nesta nova versão, não está limitada apenas a *Attribute* do tipo *Boolean.* Neste caso é possível que o desenvolvedor compare *Attributes* com valores constantes como, por exemplo, para a criação da seguinte *Premise* (*saleOrder->atCloseSaleOrder == 1*). Neste caso, o desenvolvedor poderá atribuir o valor 1 (um) para a comparação ao *Attribute atCloseSaleOrder.*

Outrossim, é possível que o desenvolvedor relacione dois *Attribute* de dois *FBE*s distintos. Por exemplo, a criação da seguinte *Premise*: (*saleOrder->atTotalSaleOrder <= client->atCreditLimit*), conforme esboçado na Figura 82. Esta *Premise* é composta por dois *Attributes* correspondentes de dois diferentes *FBE*s (*saleOrder* e *client*). Assim este tipo de arranjo também poderá ser definido pelo desenvolvedor através da nova versão da ferramenta *Wizard* PON.

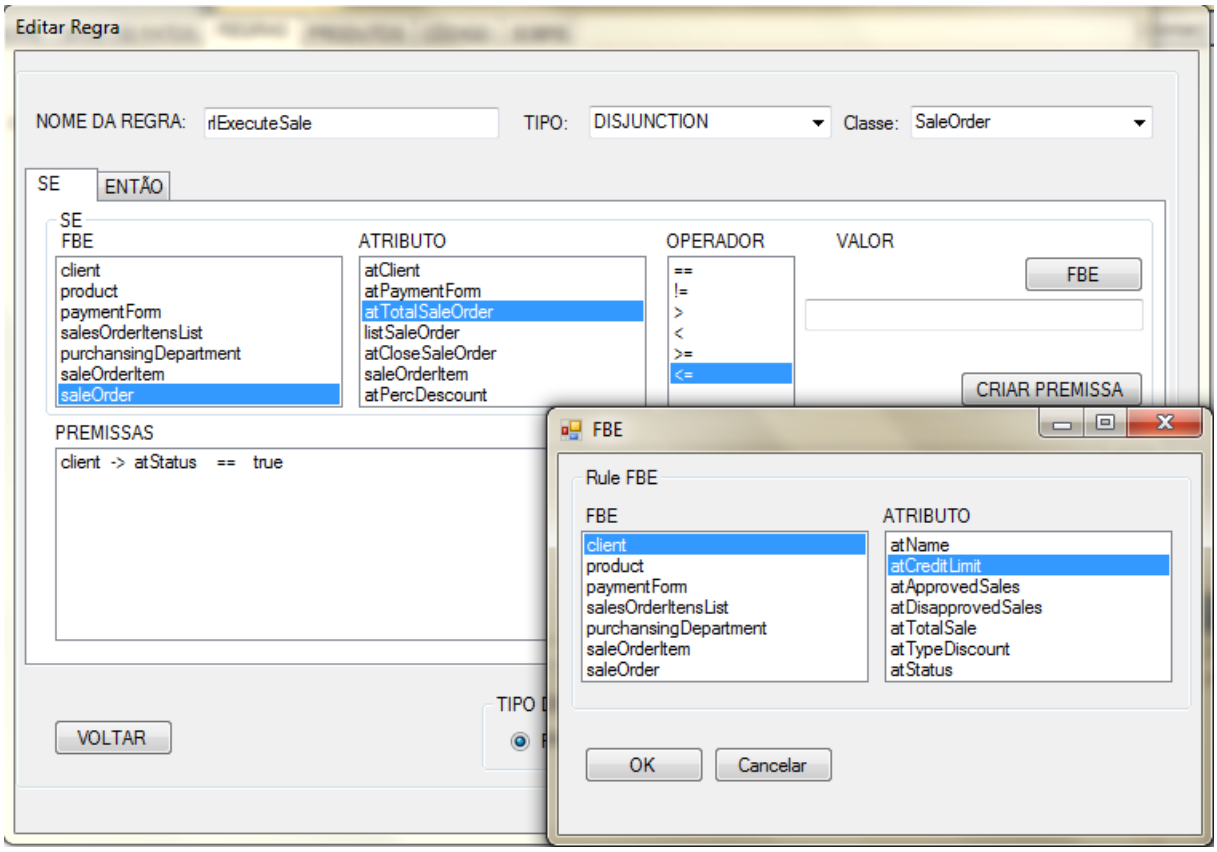


Figura 82 – Implementação *Rule* com dois *FBEs*

Desta maneira, os passos para a composição de uma *Rule* se dão inicialmente através da especificação de seu tipo. Susequentemente, a composição de uma *Rule* avança pela criação da(s) *Premise*(s), as quais fazem parte de uma *Condition*, composta pela parte “se” de uma expressão causal condicional. Finalmente, a composição de uma *Rule* se finda através da determinação dos *Methods* de execução, composta pela parte “então” de uma expressão causal condicional.

No exemplo da Figura 83, a *Rule* nomeada como *“ruleExecuteSale”* é utilizada pelo sistema de Vendas. Em suma, a primeira parte da composição de uma *Rule* é em relação a sua condição (se), formada neste exemplo por três *Premises*. A

primeira verificaria se o *status* do cliente é ativo, ou seja, verdadeiro (*client.>atStatus == true*). A segunda verificaria se o total da venda efetuada pelo cliente é menor ou igual o limite de crétido disponível para o respectivo cliente (*sale->atTotalSale <= client->atCreditLimit*). Por fim, a última *Premise,* validaria se o *Attribute "atCloseSaleOder"* contém o valor 1 (um).

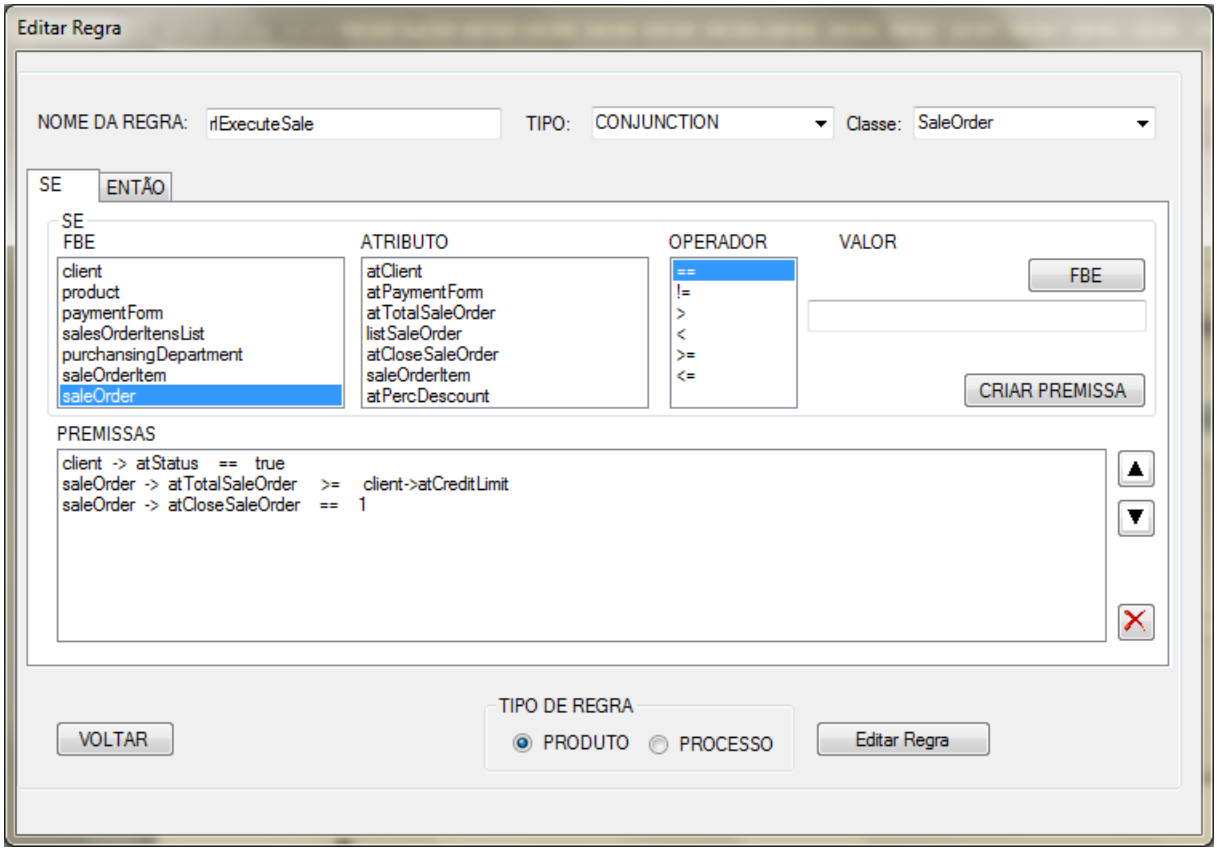


Figura 83 – Implementação *Rule* (Se)

Ainda, após a definição da *Condition,* formada pela(s) sua(s) *Premise*(s), o desenvolvedor determinaria o(s) *Methods* de execução da *Rule*. No exemplo da Figura 84, definiu-se como execução da *Rule* os métodos *"notifyClient" e "lowInventory"*. Para esta definição é necessário selecionar o *FBE* em questão e seu respectivo *Method*. Neste exemplo o *Method "notifyClient"* notificaria a compra efetuada para o respectivo *FBE client.* Por fim, o *Method "lowInventory"* realizaria a baixa de estoque contida na lista de produtos da compra efetuada pelo cliente.

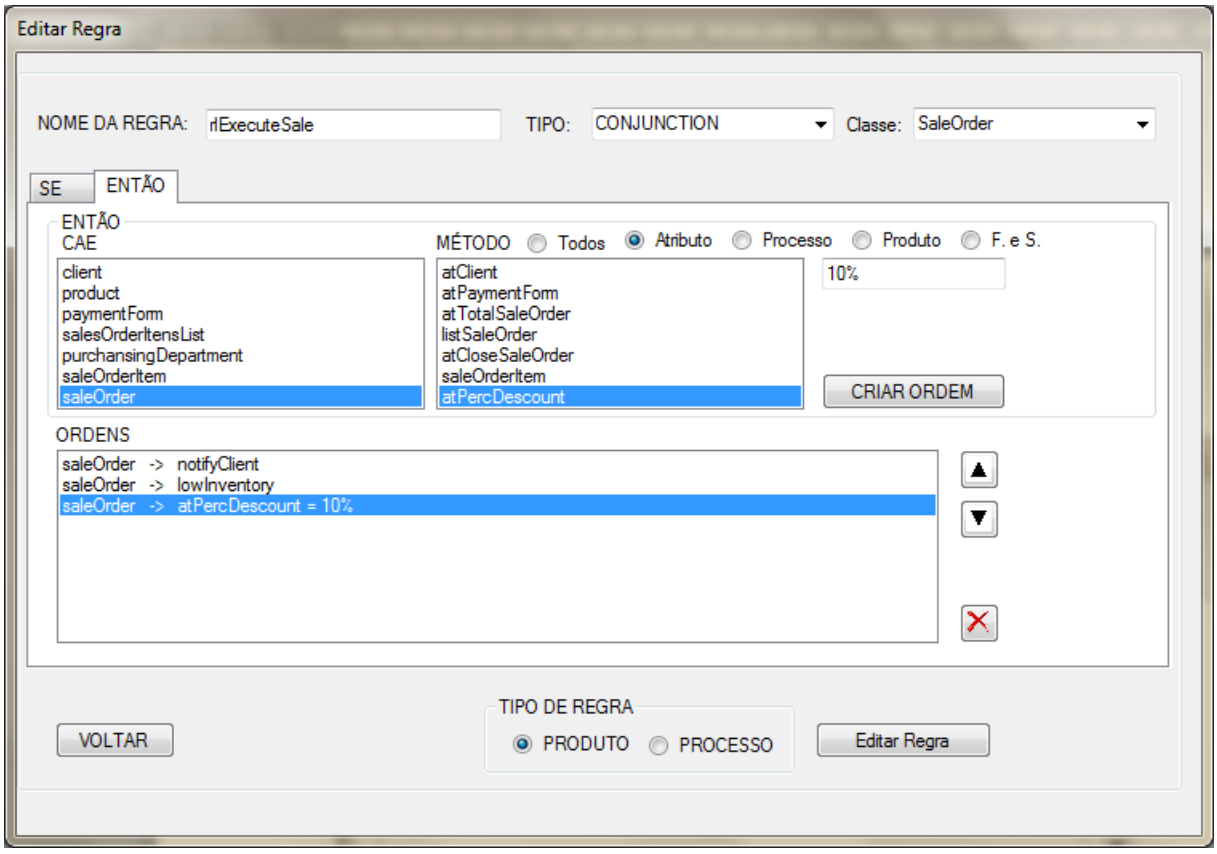


Figura 84 – *Rule* (Então)

Ademais, além de executar um *Method* tracional no PON, a ferramenta *Wizard* proporcionaria o acesso direto ao respectivo *Attribute.* Neste caso, conforme esboça a Figura 84, o *Attribute "atPercDescount"* do *FBE saleOrder,* receberia o valor (10%), onde neste caso será atribuído ao *FBE client* um desconto de 10% sobre o valor total da venda efetuada. Isso é implementado através do acesso direto realizado pela *Instigation* ao respectivo *Attribute*. Tal procedimento foi detalhado na seção 2.4.4.

Outrossim, é possível visualizar os detalhes inerentes a formação da *Rule*. Para isto, bastaria o desenvolvedor selecionar a respectiva *Rule*. Neste momento será apresentado conjuntamente o nome da *Rule,* a sua *Condition* formada pelas suas respectivas *Premises* e por fim a sua *Action*, formada pela execução de seus *Methods*. A Figura 85 exemplifica o formulário relativo a tal funcionalidade.

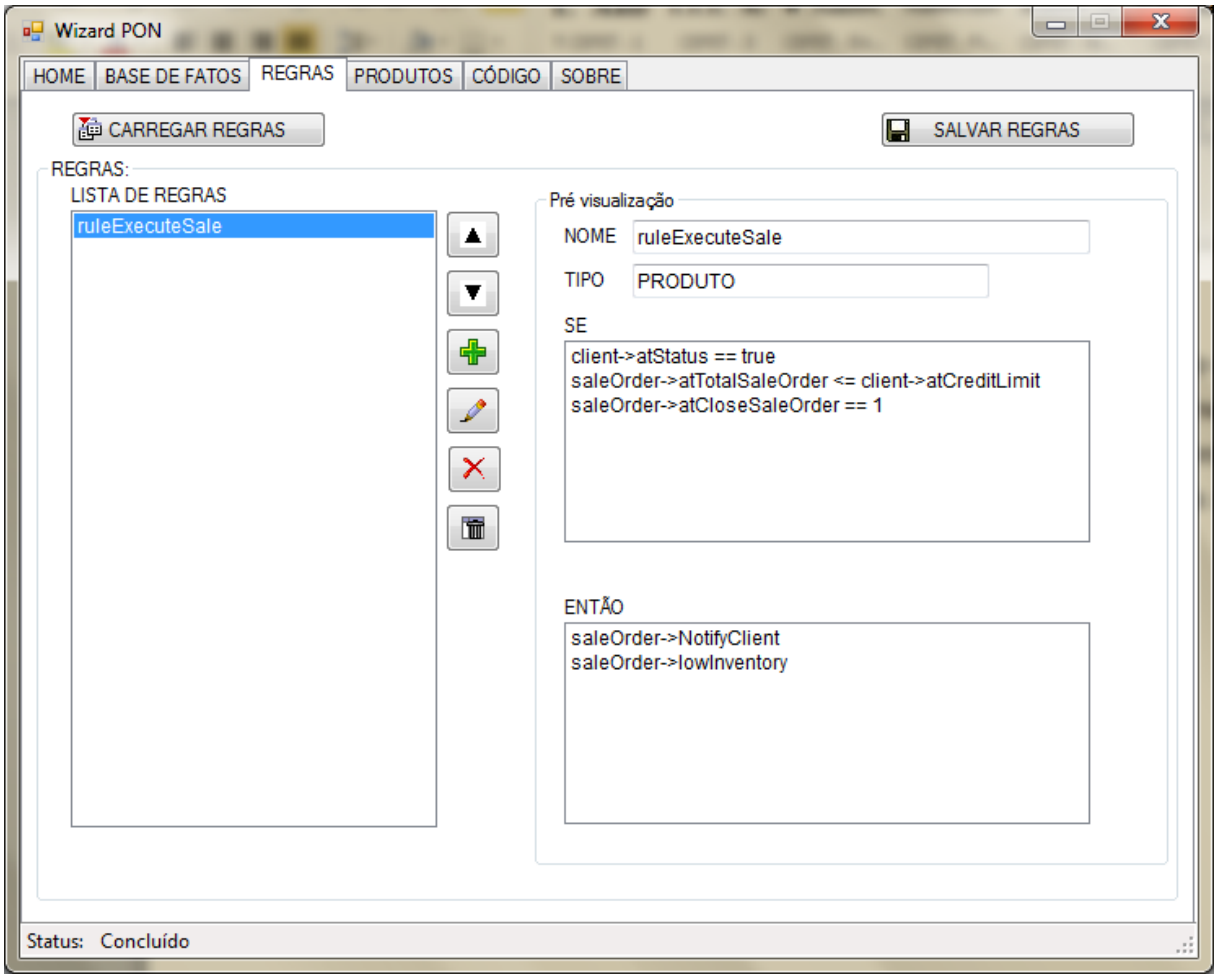


Figura 85 – Lista de *Rules*

Assim como implementado para os *FBE*s, as *Rules* criadas pelo desenvolvedor poderão ser armazenadas em arquivos de formato *XML*. A estrutura deste arquivo é exemplificada pela Figura 86. Desta maneira é possível armazenar e recuperar as informações referentes ao *FBEs* e suas *Rules* associadas, para posterior manipulação através da própria ferramenta *Wizard* PON ou mesmo em outra eventual ferramenta.

```
<Regras>
    <Regra>
        <Nome>rlExecuteSale</Nome>
        <TipoCondicao>DISJUNCTION</TipoCondicao>
        <Classe>SaleOrder</Classe>
        <Condicao>
            <Premissa>
                <CAE>client</CAE>
                <Atributo>atStatus</Atributo>
                <Operador>==</Operador>
                <Valor>true</Valor>
            </Premissa>
            <Premissa>
                <CAE>saleOrder</CAE>
                <Atributo>atTotalSaleOrder</Atributo>
                <Operador> >= </Operador>
                <Valor>client->atCreditLimit</Valor>
            </Premissa>
            <Premissa>
                <CAE>saleOrder</CAE>
                <Atributo>atCloseSaleOrder</Atributo>
                <Operador>==</Operador>
                <Valor>1</Valor>
            </Premissa>
        </Condicao>
        <Acao>
            <Ordem>
                <CAE>saleOrder</CAE>
                <Metodo>notifyClient</Metodo>
            </Ordem>
            <Ordem>
                <CAE>saleOrder</CAE>
                <Metodo>lowInventory</Metodo>
            </Ordem>
        </Acao>
    </Regra>
</Regras>
```

Figura 86 – *XML Rule*

4.1.4 Caso de Uso - Geração de Código PON

O caso de uso “Gerar Código PON” engloba o caso de uso “Gerar código *FBE*”, o qual inclui os subcasos de uso “Gerar *FBE* *.h” e “Gerar *FBE* *.cpp”. Ainda, engloba o caso de uso “Gerar Código *Rule*”, o qual inclui os subcasos de uso “Gerar *Rule* *.h” e “Gerar *Rule* *.cpp”. Em suma estes casos de uso são responsáveis pela geração de código PON através dos *FBE*s e *Rules* criadas pelo desenvolvedor através da ferramenta *Wizard* PON. Assim, conforme definido pelo desenvolvedor, o

próximo passo é a geração de código PON. A Figura 87 representa a interface reponsável por esta funcionalidade.

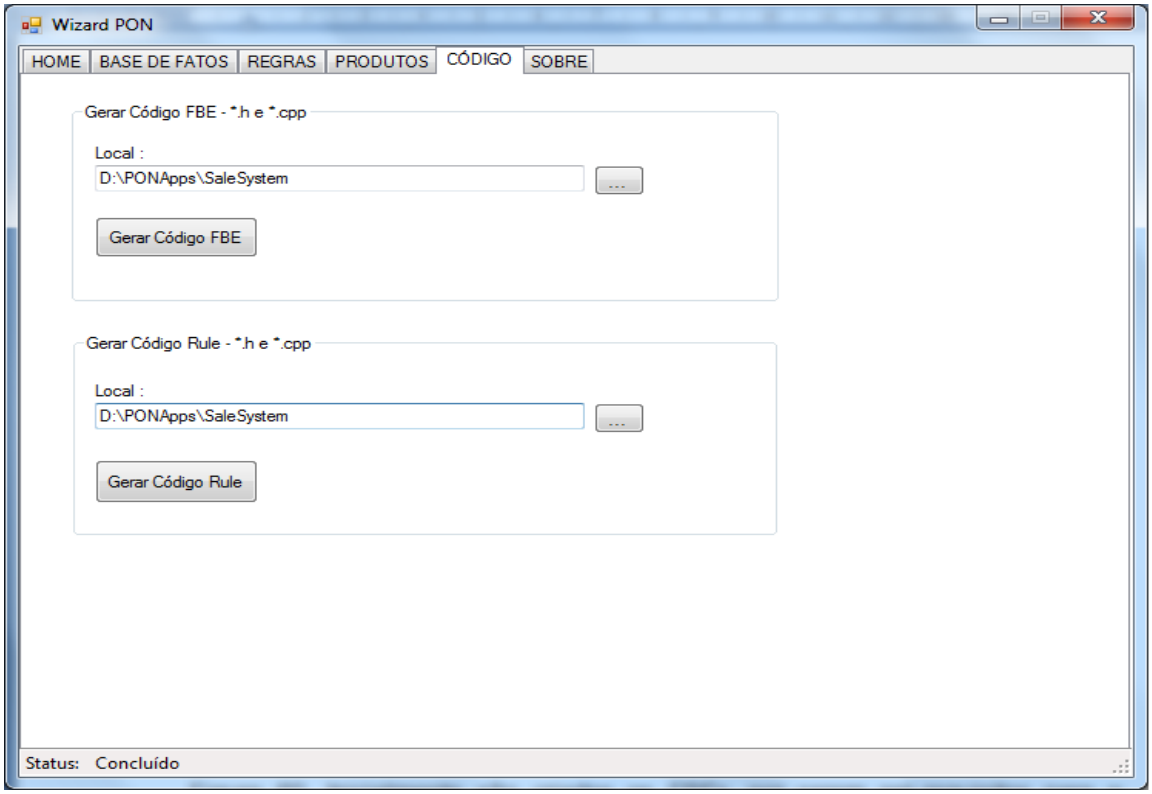


Figura 87 – Geração de Código PON

O conjunto de classes/arquivos gerados pelo *Wizard* PON é exemplificada na Figura 88. Inicialmente são criados os *FBE*s, por serem pré-requisitos para a definição das *Rules*. Os *FBEs* criados, são divididos em arquivos (*.h), que definem a estrutura da classe e em arquivos (*.cpp), que contém a implementação propriamente dita dos *FBEs*. Logo após, as *Rules* são criadas sobre o viés da nova implementação do *Framework* PON. Basicamente elas estão definidas na classe *SaleOrderApp*. Por fim, o arquivo principal (*main*) é criado para a execução da aplicação.

De modo a não conter texto prolixo, para maiores detalhes, a seção 4.3.2, a qual corresponde a implementação do estudo de caso (sistema de Vendas) através da ferramenta Wizard PON, demonstrará o código efetivamente gerado pelo *Wizard* PON.

Figura 88 – Estrutura de Arquivos

4.2 UTILIZAÇÃO DA FERRAMENTA *WIZARD* PON

Inicialmente o desenvolvedor de uma aplicação PON deverá compor os *FBEs* sobre a ferramenta *Wizard* PON. O diagrama de atividades correspondente a Figura 89 representa a dinâmica de sua utilização para compor os *FBEs*. Neste âmbito, o desenvolvedor poderá optar por importar os *FBE*s de um arquivo *XML*, correspondente estrutura definida pela Figura 81, ou criá-los manualmente através do formulário da Figura 78. Após a criação ou edição das classes *FBE*, o desenvolvedor deverá criar as instâncias dos respectivos *FBEs*. Por fim, poderá visualizar seus *Attributes* e *Methods* através do formulário correspondente a Figura 80.

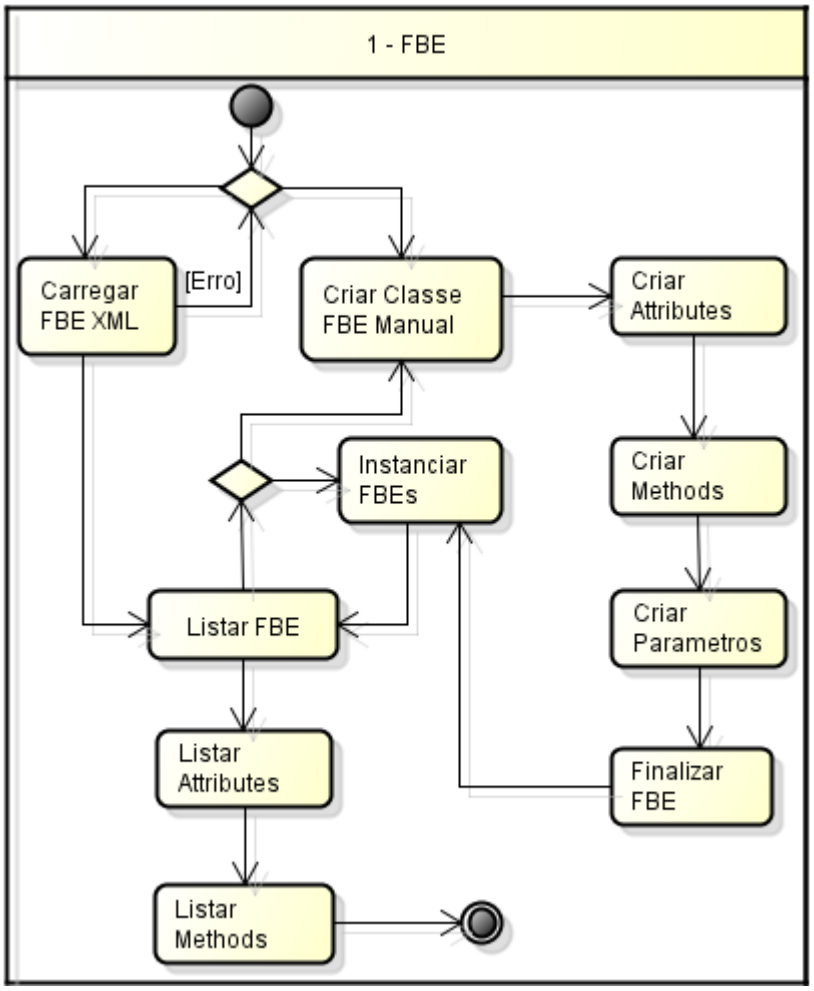


Figura 89 – Criação *FBE* no *Wizard*

Em um segundo momento, o desenvolvedor criará as *Rules,* conforme apresentado no diagrama de atividades da Figura 90. As mesmas poderão ser importadas de um arquivo *XML* ou criadas através do formulário da Figura 84. A composição de uma *Rule* é realizada através da criação de sua *Condition*, que por sua vez, é composta pela criação da(s) *Premise*(s), que fazem parte da opção “se” de uma expressão causal. Para finalizar a *Rule*, a *Action* da mesma deverá ser elaborada através da seleção de seu(s) *Method*(s), ou seja, a parte denominada “então” de uma expressão causal. Ainda o desenvolvedor poderá visualizar as *Rules* concebidas através do formulário da Figura 85.

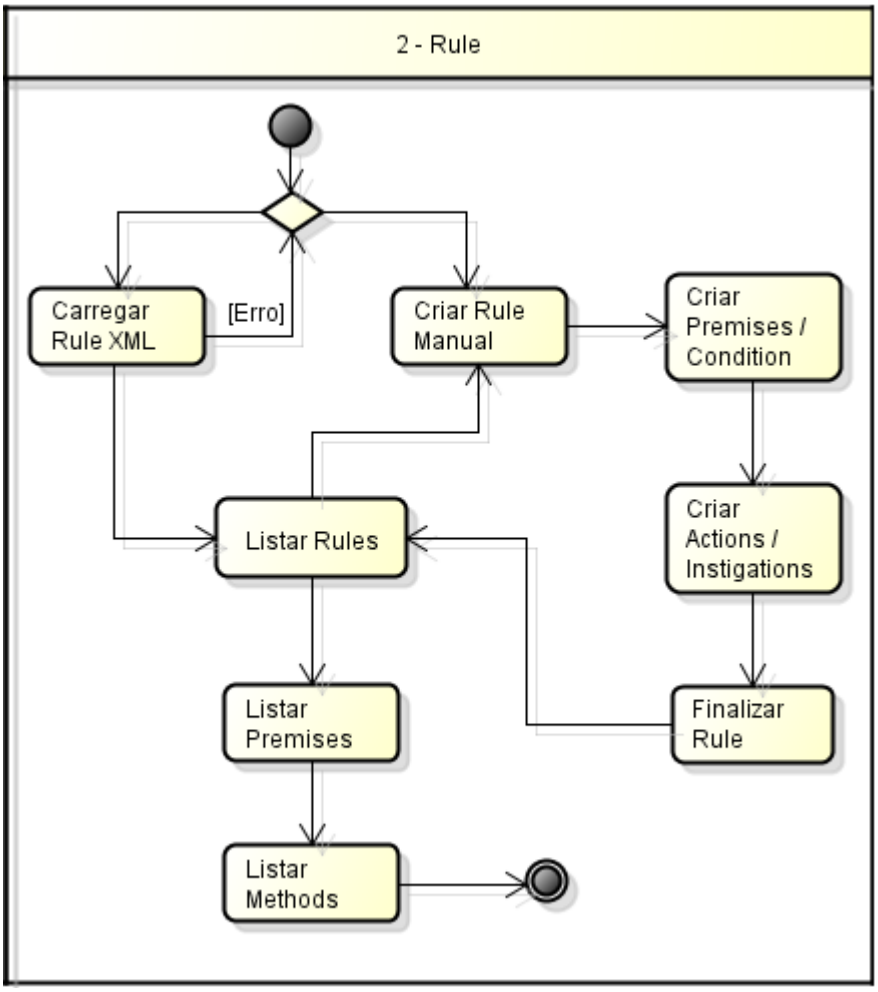


Figura 90 – Criação *Rule* no *Wizard*

O terceiro e último passo da utilização da ferramenta *Wizard* PON é observada pelo diagrama de atividade da Figura 91. Neste momento, o desenvolvedor poderá gerar o código fonte em PON do projeto elaborado através da ferramenta *Wizard* PON. Pela sequência de execução, inicialmente os passos definidos pela Figura 89 e Figura 90 deverão ser finalizados. Por fim, o processo de geração de código se dá pela composição de código relacionado aos *FBE*s e suas respectivas *Rules*.

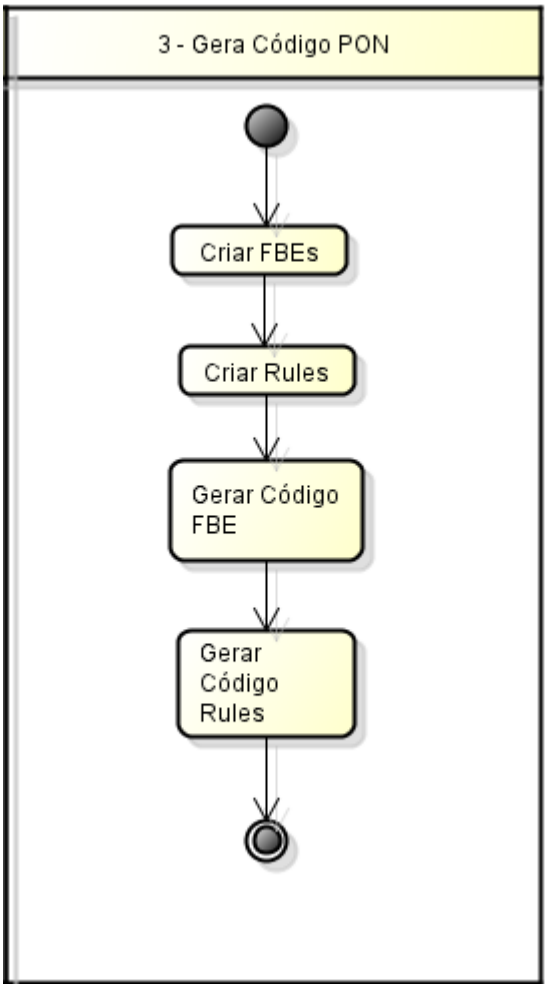


Figura 91 – Geração de Código PON *Wizard*

4.3 PROCESSO DE DESENVOLVIMENTO *WIZARD* E CASO DE ESTUDO

A seção 4.3.1, aborda o processo de desenvolvimento utilizado para a implementação de aplicações PON via ferramenta *Wizard* PON. Ademais, tal processo encaixaria na fase de construção do modelo de Desenvolvimento Orientado a Notificações (DON). A seção 4.3.2, por sua vez, detalha o estudo de caso utilizado para validar as melhorias e evoluções implementadas para a ferramenta *Wizard* PON.

4.3.1 Processo de Desenvovimento *Wizard* PON

O processo de desenvolvimento do aplicativo Vendas foi iniciado através da ferramenta *Wizard* PON. As etapas do desenvolvimento foram correspondentes ao diagrama de atividades das Figura 89, Figura 90 e Figura 91. Neste âmbito, o ciclo de desenvolvimento de aplicações PON com o auxílio da ferramenta *Wizard* PON pode ser observado na Figura 92.

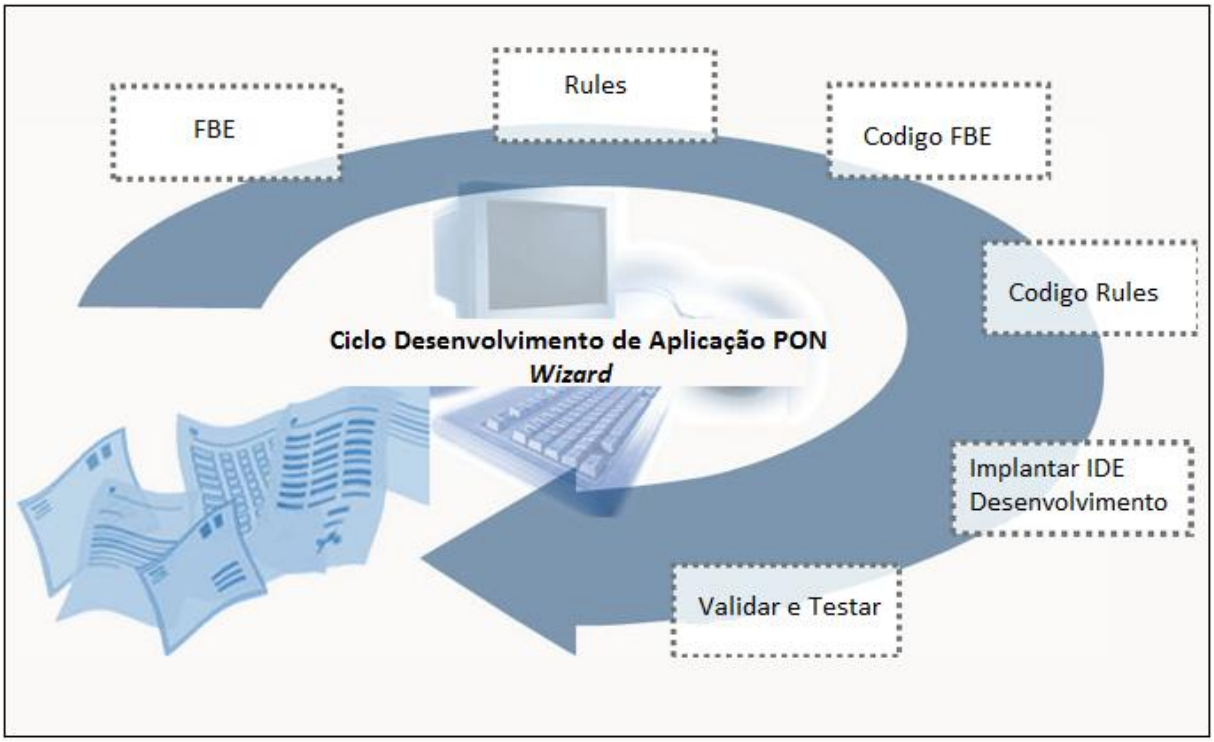


Figura 92 – Ciclo de Desenvolvimento no *Wizard* PON

Desta maneira, o processo inicia-se pela concepção dos *FBEs*, posteriormente de suas *Rules.* Logo após o código fonte dos *FBEs* e das *Rules* são gerados. O código fonte é implantado ao respectivo ambiente de desenvolvimento (*IDE*) utilizado pelo desenvolvedor. Por fim, os testes de compilação e execução da aplicação são validados. O ciclo (re)incia pela ferramenta *Wizard* PON, caso as fases de validação/testes indicarem falhas na execução da aplicação.

É importante ressaltar os trabalhos realizados por WIECHETECK (2011) sobre um processo de desenvolvimento de *sofware* para o PON, denominado Desenvolvimento Orientado a Notificações (DON), detalhado na seção 2.5. Nele,

estão descritos os passos necessários para a concepção de levantamentos de requisistos, análise e projeto de aplicações do PON. Neste âmbito, após a criação dos diagramas estruturais e comportamentais, relacionados às fases de concepção e elaboração da Figura 22, a ferramenta *Wizard* PON é utilizada na fase de construção, ou seja, referente à disciplina de implementação da aplicação PON propriamente dita.

4.3.2 Estudo de Caso – Sistema Vendas

O objetivo do estudo de caso para o Sistema de Vendas foi de realizar sua composição por meio da ferramenta *Wizard* PON, validando assim as implementações e melhorias realizadas nesta ferramenta. Para isto, os principais casos de uso do Sistema de Vendas, descrito na seção 3.10, foram reimplementados a partir da ferramenta *Wizard* PON. Assim, a nova composição do Sistema de Vendas contempla o escopo definido pelo diagrama de casos de uso da Figura 93.

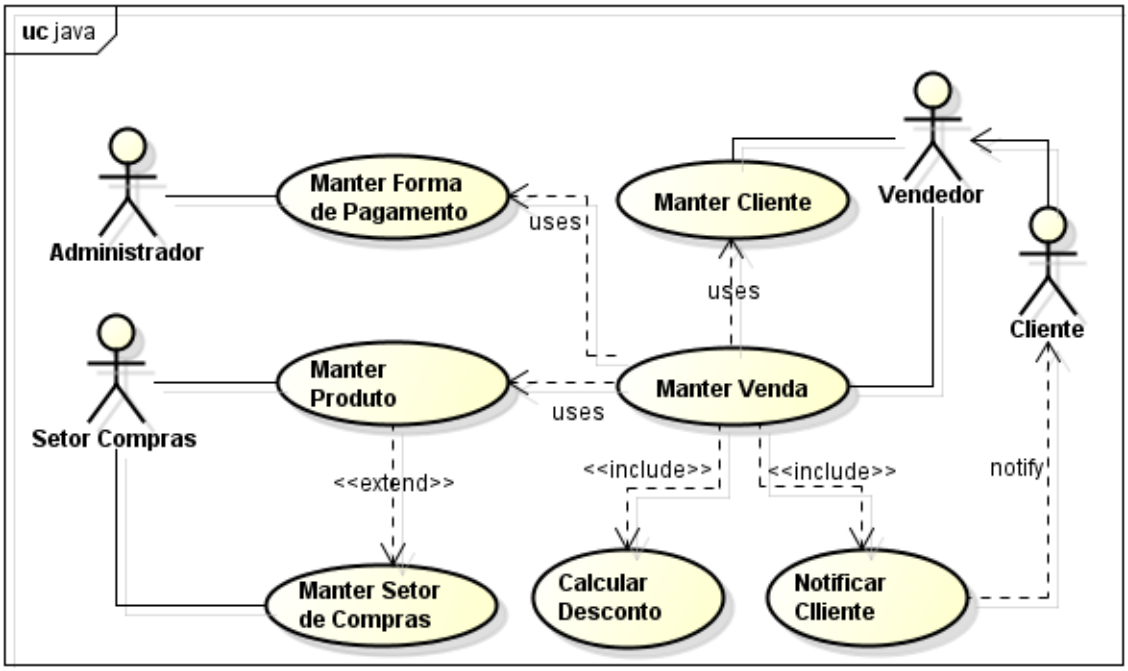


Figura 93 – Casos de uso sistema de vendas

Basicamente os principais casos de uso do aplicativo Vendas foram reimplementados com a utilização da ferramenta *Wizard* PON. Dentre eles, destaca-

se o caso de uso de efetivação de vendas, nomeado como “Manter Venda”. Para isso, inicialmente foi realizada a criação dos *FBE*s: *Client*, *PaymentForm, Product* e *SaleOrder*, bem como seus *Attributes* e *Methods.* Estes *FBE*s são representados, respectivamente, pelos casos de uso “Manter Cliente”, “Manter Forma de Pagamento”, “Manter Produto” e “Manter Venda”. A composição das *Rules*, ao seu turno, foi elaborada a partir do término da composição dos *FBEs*.

4.3.2.1 Composição de *FBE* no *Wizard* PON

A partir do processo de desenvolvimento esboçado na seção anterior, os *FBEs* foram inicialmente concebidos. Neste âmbito, a título de exemplificação o *FBE SaleOrder,* o qual corresponde ao *FBE* pedido da venda é apresentado na Figura 94.

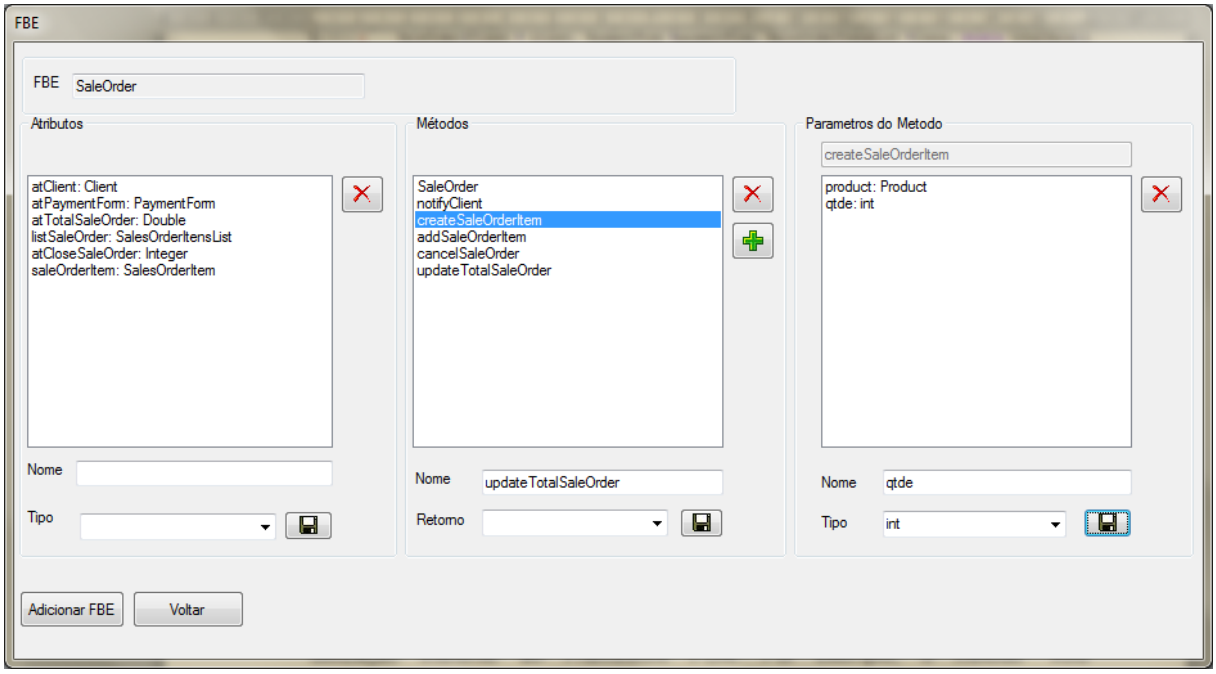


Figura 94 – *FBE SaleOrder*

Inicialmente os *Attributes* são criados pelo desenvolvedor, para cada *Attribute* é necessário definir seu nome e tipo. Para a opção tipo, serão disponibilizadas as classes responsáveis por atribuir reatividade aos tipos primitivos da OO, conforme é apresentado na Figura 42 sobre o pacote *Attribute*. Na sequência, o desenvolvedor definirá os métodos, para isso, basta definir seu nome e caso necessário seu tipo de

retorno. Por fim, o desenvolvedor poderá definir os parâmetros de cada método criado.

Outrossim, é possível escrever a implementação dos métodos através do *Wizard* PON, conforme esboçado pela Figura 95. O método exemplificado na Figura 95, por sua vez, é responsável por notificar uma venda efetuada com sucesso para um respectivo *FBE "atClient"*.

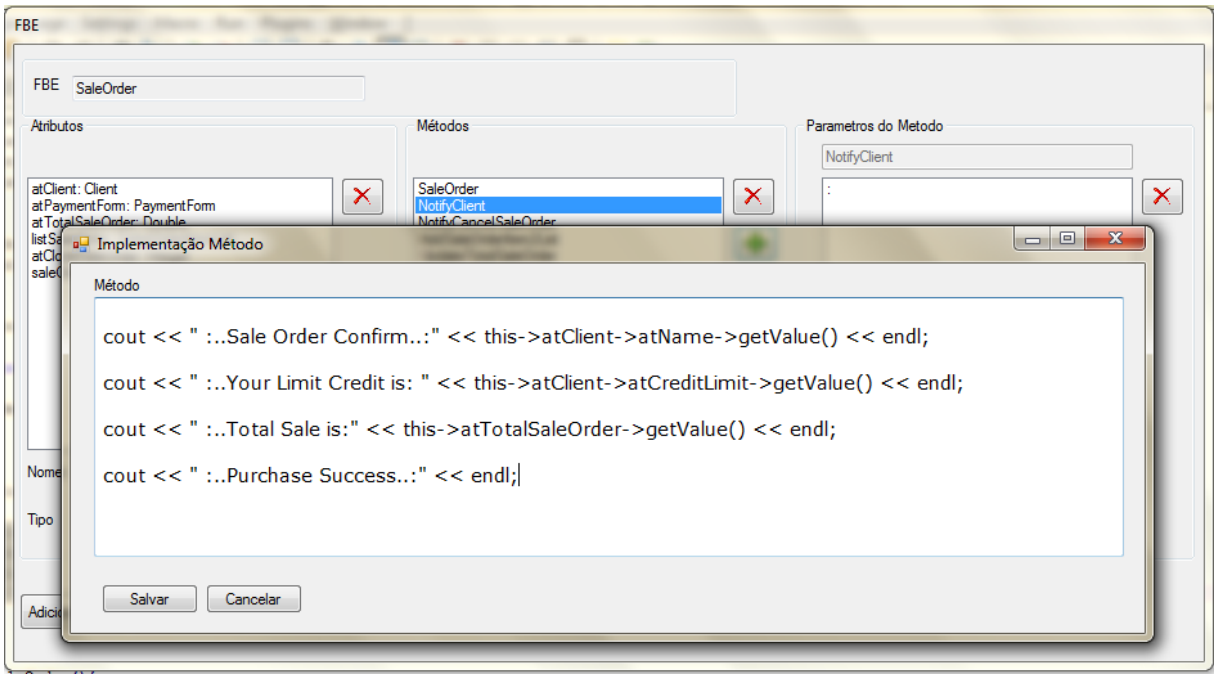


Figura 95 – Criação *FBE - Method*

4.3.2.2 Geração de Código *FBE* no *Wizard* PON

Após a concepção dos *FBEs*, o desenvolvedor poderia gerar o código fonte correspondente. Neste âmbito, o algoritmo da Figura 96 apresenta a definição da classe *FBE SaleOrder* (*SaleOrder.h*) e o algoritmo da Figura 97 apresenta a implementação de seus métodos (*SaleOrder.cpp*). Este *FBE (SaleOrder)* foi representado respectivamente pelas Figura 94 e Figura 95. Ainda, a título de informação, alguns *Attributes* e *Methods* foram suprimidos de modo a melhorar a legibilidade do código fonte apresentado.

```
1  //******* FBE Gerado Automaticamente pelo Wizard PON ************
2
3  class SaleOrder: public FBE {
4
5  public:
6
7      SaleOrder();
8      SaleOrder(Client * client, PaymentForm *paymentForm,
9                SalesOrderItensList *itens, double totalSale);
10     SaleOrder(Client * client);
11     virtual ~SaleOrder();
12     void notifyClient();
13     void createSaleOrderItem(Product* product, int qtde);
14     void addSaleOrderItem();
15     void cancelSaleOrder();
16     void updateTotalSaleOrder();
17
18 public:
19
20     Client* atClient;
21     PaymentForm* atPaymentForm;
22     Double *atTotalSaleOrder;
23     SalesOrderItensList* listSaleOrder;
24     MethodPointer<SaleOrder>* mtNotifyClient;
25     MethodPointer<SaleOrder>* mtNotifyCancelSaleOrder;
26     Method* mtCalculateTotalValue;
27     Premise * prCheckStatusSaleOrderItem;
28     RuleObject * ruleApproveAndAddSaleOrderItem;
29     Method* mtUpdateTotalSaleOrder;
30     SalesOrderItem* saleOrderItem;
31     MethodPointer<SaleOrder>* mtAddSaleOrderItem2List;
32     Integer* atCloseSaleOrder;
33 };
34
```

Figura 96 – *FBE SaleOrder.h*

A cada método OO criado pelo desenvolvedor através da ferramenta *Wizard* PON, o seu *Method* PON correspondente será automaticamente criado para sua utilização inerente ao *Framework* PON. Por exemplo, o método OO "*void updateTotalSaleOrder()*" na linha 16, tem a sua definição em PON correspondente através do atributo: "*mtUpdateSaleOrder*" do tipo "*MethodPointer<SaleOrder>*",* descrito na linha 31, o qual por sua vez, encapsula a chamada para o método OO correspondente *"updateTotalSaleOrder()"*.

```
1  //***** FBE gerado automaticamente pelo Wizard *****
2  #include "SaleOrder.h"
3
4  SaleOrder::SaleOrder() : FBE() {
5    this->name = "SaleOrder";
6    DOUBLE(this, this->atTotalSaleOrder, 0);
7    mtNotifyClient = new MethodPointer<SaleOrder>(this,   &SaleOrder::notifyClient);
8    mtUpdateTotalSaleOrder = new MethodPointer<SaleOrder>(this,   &SaleOrder::updateTotalSaleOrder);
9    mtAddSaleOrderItem2List = new MethodPointer<SaleOrder>(this,  &SaleOrder::addSaleOrderItem);
10   mtNotifyCancelSaleOrder = new MethodPointer<SaleOrder>(this, &SaleOrder::cancelSaleOrder);
11   listSaleOrder = new SalesOrderItensList();
12   INTEGER(this, this->atCloseSaleOrder, 0);
13   DOUBLE(this, this->atPercDescount, 0);
14 }
15
16 SaleOrder::~SaleOrder() {
17 }
18
19 void SaleOrder::addSaleOrderItem(){
20   listSaleOrder->addSaleOrderItem(saleOrderItem);
21 }
22
23 void SaleOrder::updateTotalSaleOrder(){
24   double totSale = this->atTotalSaleOrder->getValue() + saleOrderItem->atTotalValue->getValue();
25   this->atTotalSaleOrder->setValue(totSale);
26 }
27
28 void SaleOrder::notifyClient() {
29   cout << " :..Sale Order Confirm..:" << this->atClient->atName->getValue();
30   cout << " :..Your Limit Credit is: " << this->atClient->atCreditLimit->getValue() << endl;
31   cout << " :..Total Sale is:" << this->atTotalSaleOrder->getValue() << endl;
32   cout << " :..Done..:" << endl;
33 }
```

Figura 97 – *FBE SaleOrder.cpp*

O algoritmo da Figura 97 apresenta a implementação dos métodos da classe "*SaleOrder.h*" apresentada na Figura 96. No construtor da classe os *Methods* do PON são criados, como a instanciação do *Method "mtUpdateTotalSaleOrder"* na linha 8. Desta maneira, após a aprovação de uma *Rule* correspondente a execução deste *Method*, o mesmo executará o seu respectivo método OO encapsulado. Neste caso, executará o método OO *updateTotalSaleOrder()* da linha 23. Ainda, os *Attributes* do *FBE* em questão são instanciados, como na linha 12, através da instanciação do *Attribute "atCloseSaleOrder",* o qual recebe inicialmente o valor 0 (zero).

4.3.2.3 Composição de *Rule* no *Wizard* PON

A partir da elaboração dos *FBEs*, o desenvolvedor poderá compor as *Rules* da aplicação PON. A título de ilustração, a *Rule* esboçada na Figura 98 *"ruleApproveProductItem"*, é referente à implementação da *Rule* responsável por

efetivar um item de produto a uma respectiva venda. Para isso seria necessário a aprovação da seguinte *Premise:* - o estoque atual do *FBE product* deve ser maior ou igual à quantidade do respectivo produto a ser adicionado à lista de compras do cliente. Após a validação da *Condition*, representada pela *Premise* supracitada, os métodos *"mtLowerStock"* e *"mtCalculateTotalValue"* serão executados. O primeiro realizaria a baixa de estoque do *FBE product* vendido e o segundo atualizaria o valor da compra do *FBE product* selecionado.

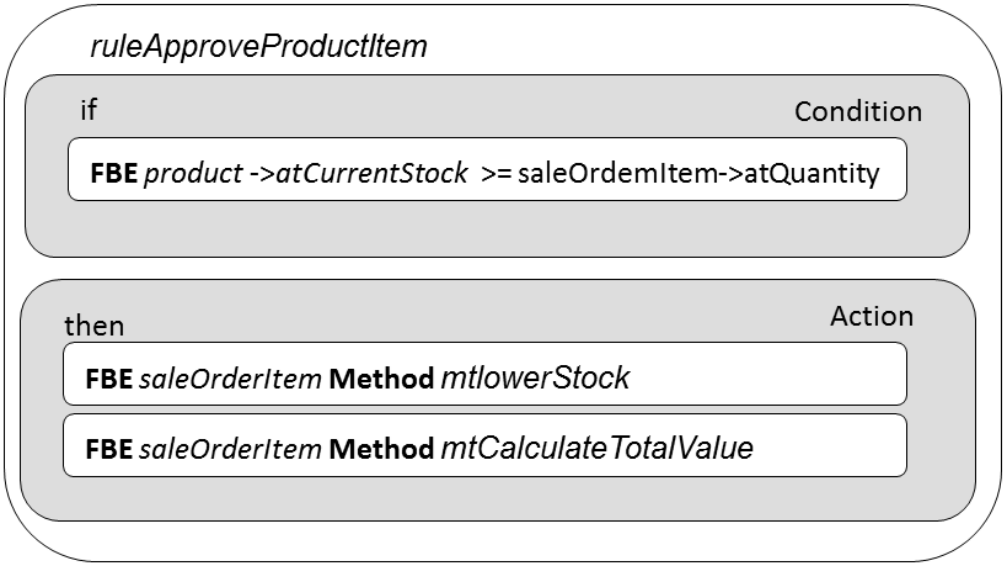


Figura 98 – *Rule ruleApproveProductItem*

Para melhor exemplificar a criação de *Rules* através da ferramenta *Wizard* PON, a Figura 99 exemplifica a *Rule "rlApproveProductItem"* esboçada pela Figura 98 em sua parte *Condition* e a Figura 100 representa sua criação da parte *Action* da mesma *Rule* em questão.

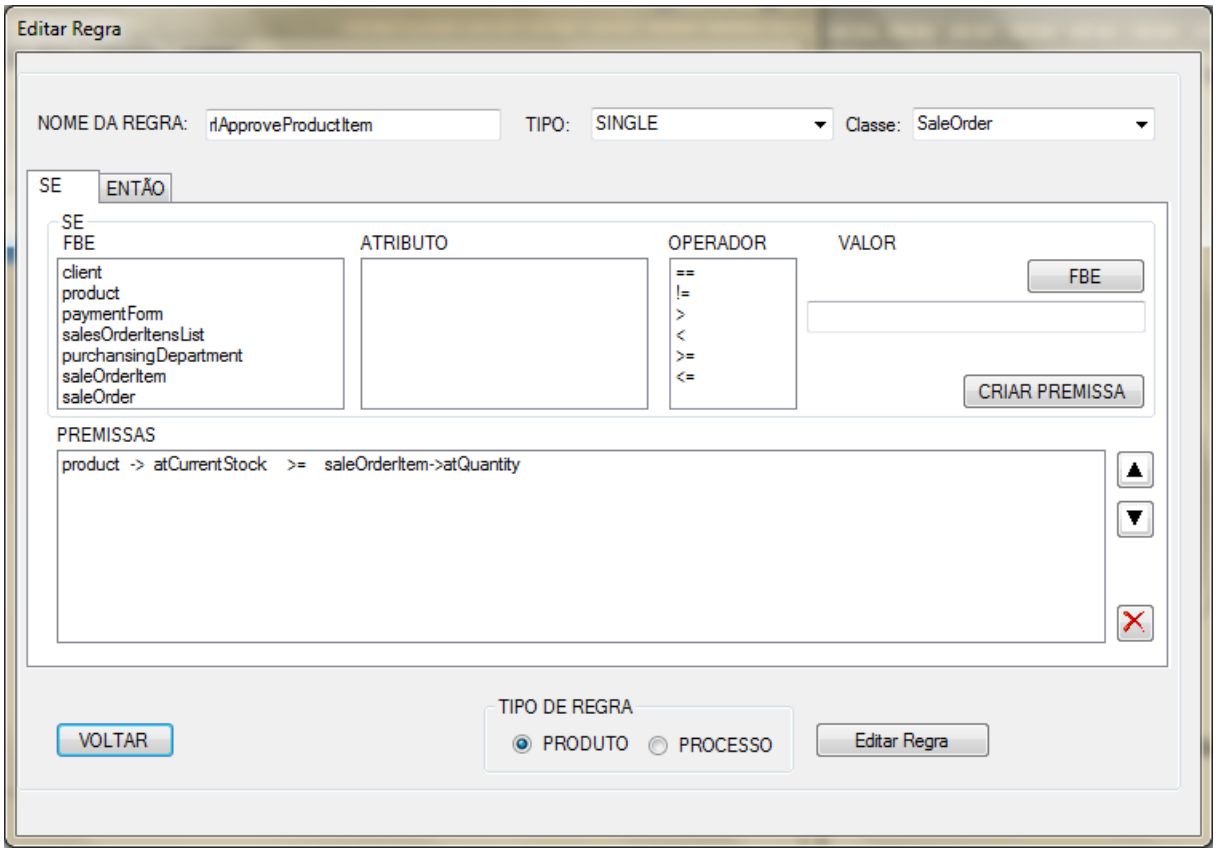


Figura 99 – *Rule ruleApproveProductItem* (*Condition*) via *Wizard* PON

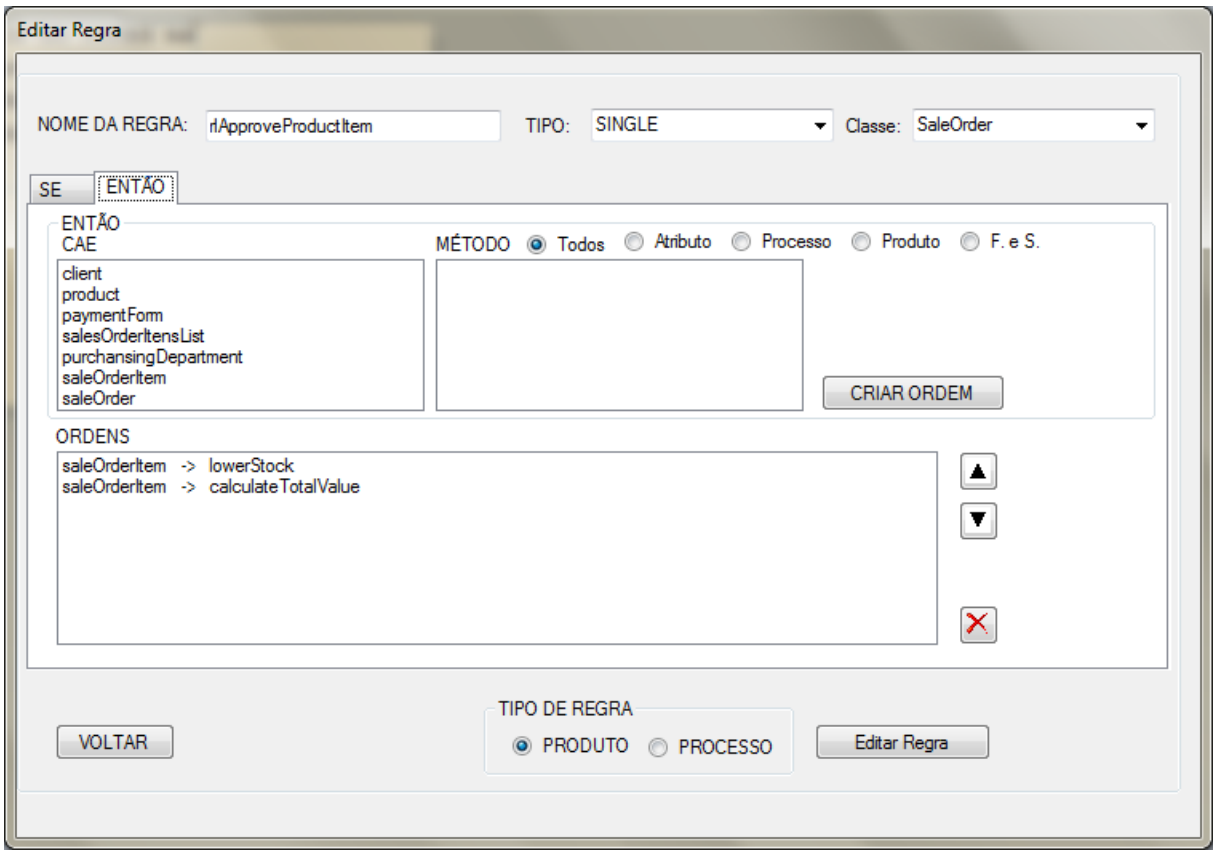


Figura 100 – *Rule ruleApproveProductItem* (*Action*) via *Wizard* PON

Ainda existem outras *Rules* que fazem parte do contexto da aplicação Vendas. A título de ilustração, a *Rule "ruleApproveAndAddSaleOrderItem",* esboçada na Figura 101, é responsável por adicionar um item de produto a cesta de compras do cliente. Neste âmbito, a *Premise* relacionada à *Rule* verificaria se o produto foi aprovado para a venda. Quando a *Condition* é validada os seguintes *Methods* são executados: "*mtAddSaleOrderItem2List",* responsável por adicionar a compra em sua respectiva lista de itens de *FBE product* e o *Method* "*mtUpdateTotalSaleOrder",* responsável por atualizar o valor total da compra com um todo efetuada pelo cliente até o momento.

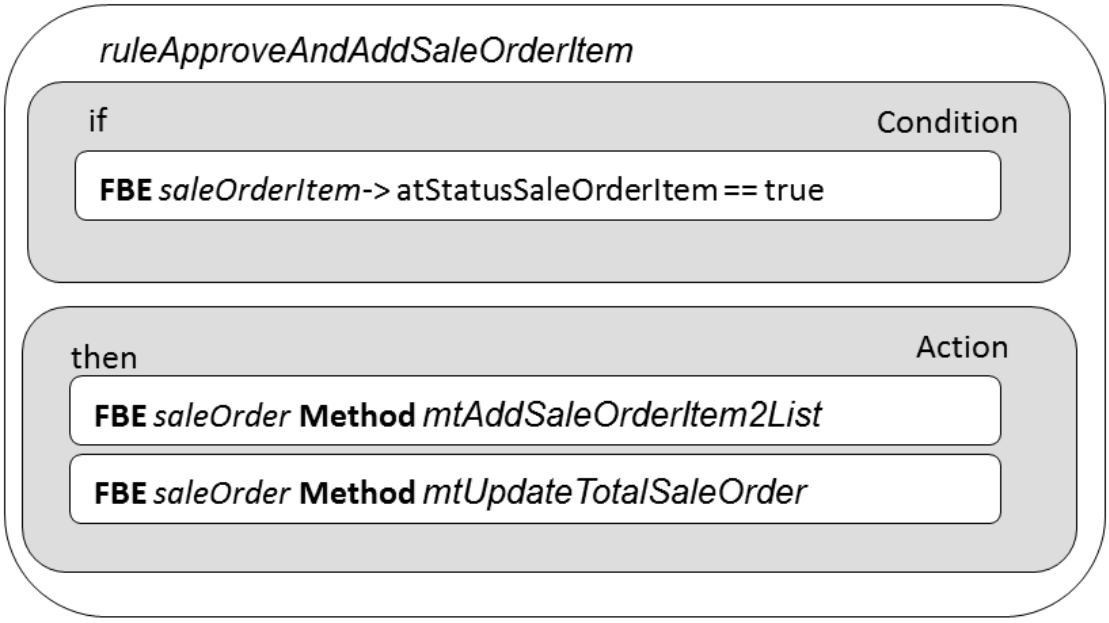


Figura 101 – *Rule* aprovar compra

Neste âmbito, a Figura 102 representa a criação da *Rule* da Figura 101 em sua parte *Condition*. Ainda, a Figura 103 representa a criação da *Action* da *Rule* em questão.

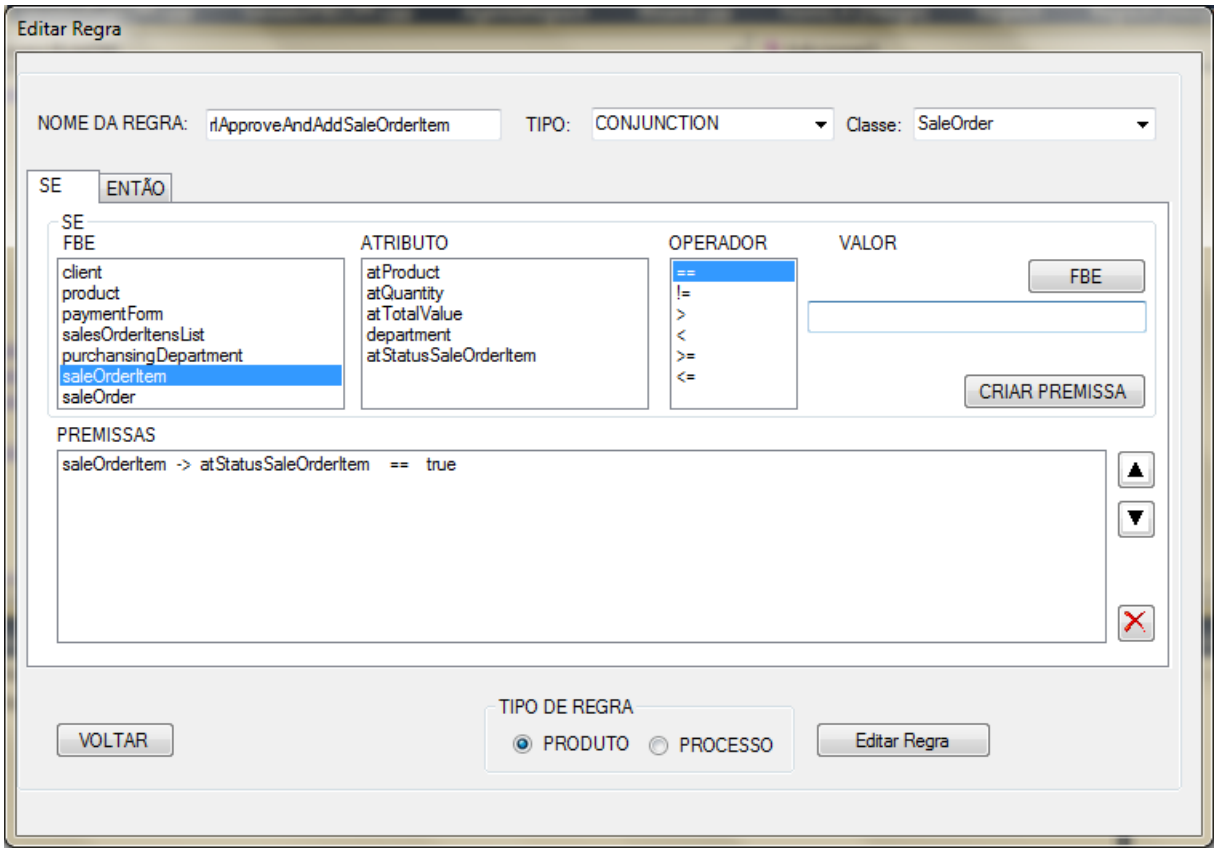


Figura 102 – *Rule rlApproveAndAddSaleOrderItem* (*Condition*) via *Wizard* PON

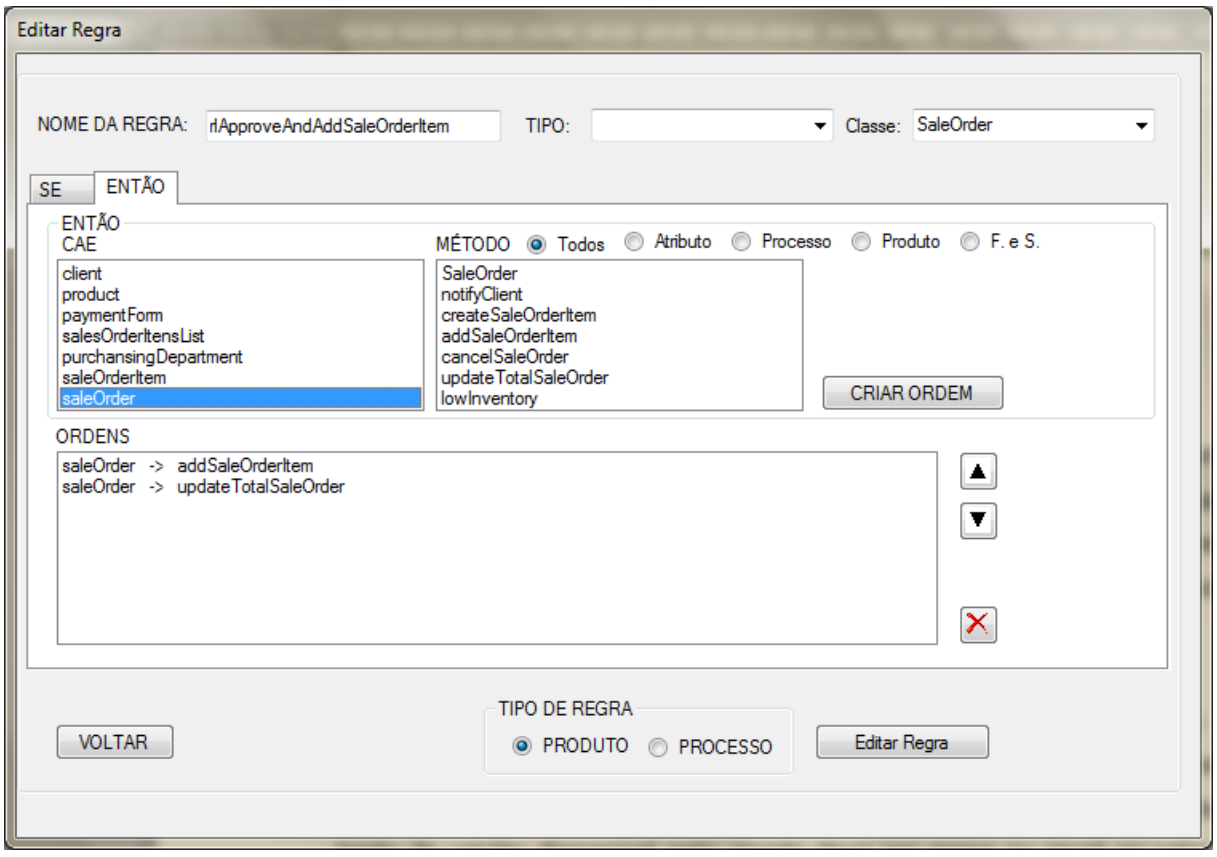


Figura 103 – *Rule rlApproveAndAddSaleOrderItem* (*Action*) via *Wizard* PON

As *Rules* correspondem ao cerne de uma aplicação PON, onde de fato ocorrerá parte da dinâmica de sua execução. Algumas *Rules* criadas para o projeto foram descritas nos parágrafos precedentes e logo após salvas em formato *XML*. Ao todo foram concebidas 25 *Rules* e suas respectivas entidades para a execução da aplicação Vendas como um todo.

4.3.2.4 Geração de Código *Rule* no *Wizard* PON

O próximo passo a ser executado através da ferramenta *Wizard* PON é a geração de código PON referente às *Rules* criadas. Neste âmbito, conforme definido pelo desenvolvedor, as implementações das *Rules* serão descritas conforme a classe informada pelo desenvolvedor na ferramenta *Wizard* PON. Caso não seja informada a classe de criação da *Rule* em questão, a mesma será criada por padrão na classe principal da aplicação sobre a implementação do método *initRules*().

Neste âmbito, o algoritmo da Figura 104 apresenta a definição da classe "*SaleOrderApp.h" e* o algoritmo da Figura 105 apresenta a implementação de seus métodos em "*SaleOrderApp.cpp"*, esta por sua vez, correponde a classe principal de execução da aplicação Vendas. Assim, a classe "*SaleOrderApp*" estende a classe *Application* do *Framework* PON, como pode ser observado na linha 1 da Figura 104. Logo após os métodos virtuais puros contidos na classe base *Application* são esboçados entre as linhas 16 e 21. Ainda, as *Rules*, *Premises* e demais *FBEs* são definidos nesta classe, conforme apresentados entre as linhas 25 e 33.

```
1  class SalesOrderApp: public Application {
2
3  public:
4
5      SalesOrderApp(Scheduler* scheduler);
6      SalesOrderApp(int schedulerStrategy);
7      virtual ~SalesOrderApp();
8
9  public:
10
11     SalesOrderItensList *listSaleOrder;
12     MyVector<Product*> *listProducts;
13
14 private:
15
16     void initStartApplicationComponents();
17     void initFactBase();
18     void initSharedPremises();
19     void configureStartApplicationAction();
20     void initRules();
21     void codeApplication();
22
23 private:
24
25     ElementsFactory* elementsFactory;
26     RuleObject* ruleCheckMinimumStock;
27     PurcharsingDepartment * department;
28     RuleObject* ruleCheckInventory;
29     SaleOrder* saleOrder;
30     Client* client;
31     RuleObject* ruleExecuteSale;
32     RuleObject* ruleExecuteTotalSale;
33     RuleObject* ruleCancelSaleOrder;
34 };
```

Figura 104 – *FBE SaleOrderApp.h*

Após a definição dos *Attributes* e *Methods* da classe *"SaleOrderApp.h",* sua implementação é apresentada no algoritmo da Figura 105. Inicialmente após realizar a instância da classe *SaleOrderApp*, a estratégia de execução das *Rules* é repassada a classe *Application* e instanciada nesta, conforme observado nos construtores da classe *SaleOrderApp* das linhas 3 e 8. Na sequência, o método *initStartComponents()* é invocado (linhas 5 e 10). Este método tem por responsabilidade iniciar a respectiva estrutura de dados responsável pelo processo de notificação do *Framework* PON e, ainda, obter a instância da fábrica PON criada, conforme observado nas linhas 20 e 22, respectivamente.

```
1  #include "SalesOrderApp.h"
2
3  SalesOrderApp::SalesOrderApp(Scheduler* scheduler) :
4          Application(scheduler) {
5      initStartApplicationComponents();
6  }
7
8  SalesOrderApp::SalesOrderApp(int schedulerStrategy) :
9          Application(schedulerStrategy) {
10     initStartApplicationComponents();
11 }
12
13 SalesOrderApp::~SalesOrderApp() {
14
15 }
16
17 void SalesOrderApp::initStartApplicationComponents() {
18
19     //Define Estrutura de Dados
20     SingletonFactory::setStructure(SingletonFactory::PONVECTOR);
21     //Obtém a instancia da Fábrica PON
22     elementsFactory = SingletonFactory::getInstance();
23     //Inicia a Execução da Aplicação
24     this->startApplication();
25 }
```

Figura 105 – *FBE SaleOrderApp.cpp*

Na sequência, o método *startApplication()* é invocado, conforme algoritmo da Figura 105 linha 24. Este método, por sua vez, chama o método responsável por inicializar a base de fatos (*initFactBase()*) e o método responsável pela criação das *Rules* (*initRules()*). O algoritmo da Figura 106 a título de exemplificação, apresenta a criação da *Rule "ruleExecuteSale"* e da *Rule "ruleCancelSaleOrder".*

```
1  void SalesOrderApp::initRules() {
2      ...
3      prValidateClient = elementsFactory->createPremise(
4                                          client->atStatus,
5                                          elementsFactory->createInteger(1),
6                                          Premise::EQUAL, false);
7      prValidateCreditLimitClient = elementsFactory->createPremise(
8                                                    client->atCreditLimit,
9                                                    saleOrder->atTotalSaleOrder,
10                                                   Premise::GREATEROREQUAL, false);
11     prCloseSaleOrder = elementsFactory->createPremise(
12                                         saleOrder->atCloseSaleOrder,
13                                         elementsFactory->createInteger(1),
14                                         Premise::EQUAL, false);
15     ruleExecuteSale = elementsFactory->createRuleObject("ruleExecuteSale",
16                                        scheduler, Condition::CONJUNCTION);
17     ruleExecuteSale->addPremise(prValidateClient);
18     ruleExecuteSale->addPremise(prValidateCreditLimitClient);
19     ruleExecuteSale->addPremise(prCloseSaleOrder);
20     ruleExecuteSale->addInstigation(elementsFactory->createInstigation(
21                                                    saleOrder->mtNotifyClient));
22
23     prCancelSaleOrder = elementsFactory->createPremise(
24                                          client->atCreditLimit,
25                                          saleOrder->atTotalSaleOrder,
26                                          Premise::SMALLERTHAN, false);
27     ruleCancelSaleOrder = elementsFactory->createRuleObject("ruleCancelSaleOrder",
28                                            scheduler, Condition::CONJUNCTION);
29     ruleCancelSaleOrder->addPremise(prValidateClient);
30     ruleCancelSaleOrder->addPremise(prCancelSaleOrder);
31     ruleCancelSaleOrder->addPremise(prCloseSaleOrder);
32     ruleCancelSaleOrder->addInstigation(elementsFactory->createInstigation(
33                                         saleOrder->mtNotifyCancelSaleOrder));
34     ...
35 }
```

Figura 106 – *initRules SaleOrderApp*

No método *initRules()* inicialmente é realizada a criação das *Premises.* Deste modo, a *Premise "prValidateClient"* verificaria o *status* do cliente (linhas 3 à 6), a *Premise "prValidateCreditLimitClient"* validaria o limite de crédito concedido ao cliente (linhas 7 à 10) e a *Premise "prCloseSaleOrder"* validaria o momento do fechamento de uma respectiva venda (linhas 11 à 14). Observa-se que para cada elemento PON do tipo *Premise* criado, faz-se uso da fábrica PON, obtida através da chamada do método *elementsFactory->createPremise(...)*.

Na sequência, as *Rules* são criadas. Conforme observado, a *Rule "ruleExecuteSale"* da linha 15 é instanciada. Da mesma forma em que as *Premises* são criadas, as *Rules* fazem uso da fábrica PON correspondente (*elementsFactory->createRuleObject*(...)). A *Rule "ruleExecuteSale"* adiciona em seu escopo as seguintes *Premises: "prValidateClient", "prValidateCreditLimitClient"* e a *Premise*

“prCloseSaleOrder”, as quais já de antemão foram detalhadas*.* Por fim, a *Rule* em questão adiciona sua *Instigation* a execução do *Method “mtNotifyClient”* do *FBE saleOrder* (linhas 20 e 21).

A criação da *Rule “ruleCancelSaleOrder”*, ao seu turno, engloba ao seu escopo as 3 *Premises* supracitadas. Entretanto, a *Premise* que verificaria o limite de crédito do respectivo cliente é alterada. Neste caso, a validação é feita justamente ao contrário, ou seja, caso o limite de crédito concedido ao cliente seja menor que o valor da venda. Assim, quando satisfeita a *Condition* da respectiva *Rule*, a venda seria cancelada. Neste caso o cliente seria notificado com a chamada do *Method “mtNotifyCancelSaleOrder”* (linhas 32 e 33). Finalizado o processo de criação das *Rules* sobre o método *initRules(),* o método *codeApplication()* seria invocado. Este método dá início a execução da aplicação propriamente dita.

4.3.2.5 Reflexões

A junção da ferramenta *Wizard* PON com o seu referido *Framework,* torna-se indispensável para a otimização do ciclo de vida do desenvolvimento de uma aplicação PON. A grande vantagem desta abordagem é utilização de programação em alto nível no clássico formato “se-então” inerente ao PD, com a geração de código fonte no formato de *Rules*, especificamente sobre a linguagem de programação C++ orientado a nova versão do *Framework* PON.

Ademais, após a geração de código PON através da ferramenta *Wizard* PON, o projeto foi portado ao IDE de desenvolvimento *Eclipse* em ambiente C++ com o sistema operacinal *Linux* versão *Debian*. O projeto *SaleOrderApp* em questão foi criado manuamente junto ao IDE de desenvolvimento e as respectivas classes foram portadas ao projeto em questão. Ainda, as devidas configurações foram realizadas para o projeto, como a definição das pastas as quais incluem os arquivos com extensão *“*.h”.* Por fim, o projeto foi compilado, executado, validado e testado.

No contexto do processo de desenvolvimento de aplicações PON, com o auxílio da ferramenta *Wizard* PON, é importante salientar que, para a aplicação funcionar como um todo, ainda foi necessário à implementação manual via código fonte. Isto se deu principalmente para a criação e instanciação dos *FBEs*. Entretanto isto ocorreu pelo fato da aplicação não possuir a implementação da camada de

apresentação. Assim, foi necessário atribuir valores pré-definidos aos *FBEs* para a execução da aplicação.

Quando erros de lógica na programação das *Rules* ou mesmo dos *FBEs* foram detectados, os mesmos foram alterados a partir da ferramenta *Wizard* PON e não diretamente no código fonte do projeto. Isto trouxe algumas dificuldades, tendo em vista o processo moroso de copiar e colar fontes a partir da geração de código pelo *Wizard* PON para o projeto alvo na IDE de desenvolvimento. Este ciclo repetiu-se enquanto não foi possível obter a execução da aplicação em confiormidade aos seus requisitos.

4.4 CONCLUSÃO

As evoluções propostas e implementadas para a ferramenta *Wizard* PON, de fato, auxiliam o processo de desenvolvimento de aplicações PON. Isto se comprova através da elaboração do estudo de caso da aplicação Vendas, comparada ao seu desenvolvimento manual, concebido inicialmente pelos trabalhos de (BATISTA, *et.al.,* 2011). Ou seja, em um primeiro momento, esta aplicação foi desenvolvida por (BATISTA, *et.al.,* 2011) sem o auxílio da ferramenta *Wizard* PON. Em um segundo momento, a aplicação foi reimplementada através da utilização da ferramenta em questão. Percebe-se, de modo qualitativo as vantagens de sua utilização ao processo de desenvolvimento de aplicações PON, onde destacam-se:

- O desenvolvedor PON não precisa conhecer detalhadamente a sintaxe e composição das aplicações PON. De modo que ao gerar o código da aplicação concebida através da ferramenta *Wizard,* o desenvolvedor possuirá exemplos de sua utilização em código da linguagem de programação C++, com a sintaxe apropriada relativa a nova versão do *Framework* PON.
- A concepção em alto nível de regras no clássico formato se-então, facilita a composição de aplicações PON através da ferramenta *Wizard* PON. Ainda, o fato das *Rules* serem desacopladas de seus respectivos *FBEs*, agrega valor e produtividade ao conceber aplicações PON através da ferramenta *Wizard* PON.

- Através da concepção de aplicações PON com a ferramenta *Wizard* PON, garante-se que em uma primeira versão da aplicação, a mesma utilizará os bons princípios de desenvolvimento adotados para o desenvolvimento de aplicações PON. Basta então, que o desenvolvedor siga estes princípios e técnicas de desenvolvimento gerado a partir deste ferramental.
- Incentivo as boas práticas de programação em PON.
- Aumento da padronização de código PON.

Ademais, ferramentais apoiadas na *UML,* voltadas a PI, comumente oferecem a geração de código estrutural da aplicação baseadas em modelos de classes. Desta forma, a implementação da dinâmica da execução da aplicação é realizada de forma manual pelo desenvolvedor. Ainda, mesmo com adoção de ferramental que suporta a *MDA,* tais dificuldades na geração de código fonte que correponde a dinâmica de execução da aplicação ainda permanecem. Entretanto, com a concepção de aplicações PON sobre a ferramenta *Wizard* PON, é possível gerar o código fonte relativo à (parte considerável da) dinâmica de execução da aplicação, uma vez que a implementação das *Rules* corresponde justamente a (parte) da sua dinâmica.

Outrossim, é importante salientar os trabalhos realizados por WIECHETECK (2011), os quais definem os artefatos de projeto a serem elaborados de antemão ao desenvolvimento propriamente dito de uma aplicação PON, que são extremamentes importantes ao processo de desenvolvimento de uma aplicação PON, uma vez que a implementação das *Rules* sem os prévios artefatos de projeto poderia ficar inconsistente.

Ademais, a partir dos artefatos de projetos elaborados, a concepção de uma aplicação PON com auxílio da ferramenta *Wizard* PON, se torna extremamente eficiente e rápido, tendo em vista que a criação dos *FBEs* e suas *Rules* estariam pré-definidas nestes artefatos. Neste âmbito, através do conforto proporcionado pela ferramenta *Wizard* PON, através de formulários de alto nível, o desenvolvedor conceberá a aplicação PON na fase de construção do modelo *RUP* apresentado na Figura 22.

Todavia, dificuldades ainda são encontradas ao desenvolver aplicações PON a partir da ferramenta *Wizard* PON. A principal delas é manter a sincronicidade entre o código fonte da aplicação em relação ao assistente *Wizard* PON. Desta forma, caso o desenvolvedor altere diretamente o código fonte da aplicação PON, o mesmo

não poderá ser gerado novamente através do *Wizard* PON. Ainda, existem dificuldades quanto ao transporte do projeto *Wizard* a respectiva *IDE* de desenvolvimento.

5 CONCLUSÕES

De maneira geral, a materialização original do *Framewowrk* PON concretizada no trabalho de Banaszewski (2009) apresentou resultados excelentes em comparação a SBR-*RETE* e resultados bons em comparação a programas do POO (em C++) sendo estes programas com certa quantidade de redundância. Em contrapartida em experimentos subsequentes, a materialização original do *Framewowrk* PON não demonstrou vantagens satisfatórias ou mesmo desmontrou desvantagens (do ponto de vista de desempenho) em comparação a implementações POO C++ para programas com pouca ou mesmo mediana quantidade de redundância (BANASZEWSKI, 2009).

Entretanto, o cálculo assintótico do processo de inferência do PON sugeriria que o PON apresentaria resultados melhores aos obtidos. Ademais sua própria materialização apresentava sinais explícitos de que era passível evoluções, como o uso de estrutura de dados mais enxutas. Neste âmbito, melhorias na estrutura geral do *framework* PON eram desejáveis para um melhor uso do proposto na teoria que cerca o PON. Sucintamente, nesse trabalho foram propostas otimizações e refatorações de código em toda a estrutura do *Framework*, acumulando ganhos de desempenho, que de fato agregaram em sua melhoria como um todo.

Particularmente, observou-se que o processo de notificações, baseado em estruturas de dados que comportam os endereços das entidades responsáveis por tal processo, representava o ponto concentrador da maior parte do custo de processamento envolvido na execução de um programa PON. Sendo assim, os esforços de otimização foram concentrados principalmente na proposta de novas estruturas de dados para minimizar os custos envolvidos nesse processo. Outrossim, os experimentos comparativos realizados demonstraram que a nova versão do *Framework* PON obteve ganhos substanciais de desempenho em relação à versão precedente.

Ademais, melhorias e evoluções foram propostas e implementadas no âmbito da ferramenta *Wizard* PON. Estas melhorias e evoluções tiveram por objetivo agregar as vantagens relacionadas às facilidades de concepção de aplicações PON através de *Rules* em alto nível. Outrossim, foi possível conceber a geração

automatizada de código fonte de aplicações PON sobre a linguagem de programação C++ orientada a nova versão do *Framework* PON. Por fim, o estudo de caso elaborado apresentou vantagens qualitativas em relação a sua implementação manual. As próximas seções detalham as conclusões de maneira pontual e relatam os possíveis desdobramentos deste trabalho em trabalhos futuros.

5.1 CONSIDERAÇÃO SOBRE O DESEMPENHO DAS APLICAÇÕES PON

De modo geral, percebeu-se que o processo de notificações realizado pela inferência do PON representava o ponto de maior concentração de processamento. Ademais, observou-se que descuidos com a implementação desse processo resultavam em um grande impacto no desempenho das aplicações PON. Apesar de terem sido realizadas outras otimizações no âmbito do *Framework*, bem como refatorações de código precisas, os maiores ganhos de desempenho adviram essencialmente da mudança das estruturas de dados utilizadas nesse processo.

Outrossim, observou-se que as diferentes estruturas de dados propostas apresentam vantagens e desvantagens que variam de acordo com o domínio da aplicação. Todavia a variação de melhora de desempenho após as devidas otimizações e refatorações no referido *Framework*, foi de 50% nos casos de estudos aplicados descritos no capítulo 3, o que corresponde a um ganho de desempenho de no mínimo 2 (duas) vezes. O grande impacto deu-se através das otimizações implementadas sobre o orquestramento das notificações realizadas entre as entidades PON.

Na verdade, em aplicações de maior porte, as estruturas de dados deveriam ser aplicadas pontualmente nos casos em que apresentarem vantagens em relação às demais estruturas. Sendo assim, um programa PON poderia ser dotado de entidades PON com estruturas de dados distintas, o que de fato não influenciaria na execução do sistema, uma vez que tais entidades se apresentam desacopladas e independentes de suas estruturas de dados.

Ademais, conforme descrito na seção 2.3.5, o cálculo da função assintótica do PON, particularmente em seu caso médio, demostrava que sua atual materialização

era passível de melhoras na questão de desempenho. Outrossim, após a (re) análise de sua antiga materialização, constatou-se que melhoras eram extremamente pertinentes. Isto, de fato, é observado após as devidas otimizações, refatorações e resultados alcançados descritos no capítulo 3.

5.2 CONSIDERAÇÕES SOBRE AS ESTRUTURAS DE DADOS

As estruturas *PONLIST* e *PONVECTOR* possuem desempenhos aproximados, conforme observado no capítulo 3. A diferença peculiar entre as duas estruturas estão em suas formas de realizar o processo de iteração sobre seus respectivos elementos. Elementos PON que se utilizam de *PONLIST*, iteram seus elementos através de *jumps* de memória, ou seja, os mesmos estão dispersos em áreas distintas da memória principal. Assim o acesso a entidade PON é determinado logo após a computação de seu respectivo endereço.

Em relação à estrutura *PONVECTOR*, sua maneira de iteração é realizada através de aritmética de ponteiros, graças ao uso da técnica denominada *memory pool*. Isso trás vantagens em termos de desempenho de execução sobre a iteração em elementos que utilizam tal estrutura. Todavia, a sua composição pode ser onerosa e gerar custo extra de processamento, principalmente em consideração ao algoritmo de realocação de memória, exemplificado na seção 3.1.3.

De fato, ao passo que a quantidade de elementos PON aumenta o limite inicial da estrutura *PONVECTOR* pode exceder. Neste caso será acionado o algoritimo de reposição dos elementos PON. Dado que as *Rules* e demais elementos são definidos *à priori*, este processo pode ser oneroso em tempo de compilação ainda que não influenciará no tempo de execução da inferência do PON propriamente dita e, portanto, não influenciando no tempo da aplicação PON como um todo. Entretanto, se no porvir houver aplicações onde a forma das *Rules* varie no tempo, mesmo a execução da aplicação PON como um todo poderia ser afetada.

Para a estrutura *PONHASH*, por sua vez, observa-se que a mesma é diretamente influenciada pelo número de *Premises* de um respectivo *Attribute,* bem como sua variação de estado. Através da aplicação Mira ao Alvo, obteve-se esta

conclusão ao avaliar a quantidade de *Premises* para um respectivo *Attribute* e sua variação de estado*.* Assim, quando a quantidade de *Premises* para um *Attribute* é baixa, ou seja, entre 1 (uma) a 20 (vinte) *Premises*, a utilização da estrutura *PONHASH* torna-se ineficaz em termos de desempenho. Esta ineficácia se deve ao fato da realização do cálculo da função *hash* para obtenção da respectiva *Premise* e a desaprovação das respectivas *Premises* compostas pelo *Attribute* com o estado anterior aprovado.

Entretanto, a partir de certa quantidade de *Premises* (algo como acima de 30) para um único *Attribute*, a utilização da estrutura *PONHASH* é extremante eficaz, conforme observado nos tempos apresentados pela Figura 58. Ainda é importante considerar a variação de estado para o *Attribute* em questão. Isto foi observado no caso de estudo do sistema de vendas. Neste caso de estudo, um único *Attribute* denominado *(tipoDeDesconto),* o qual concederia descontos a venda efetuada pelo cliente, possuia muitas *Premises* referenciadas. Todavia, a estrutura *PONHASH* não trouxe vantagens de desempenho, tendo em vista que este *Attribute* somente variava de estado mais ao final da execução da (inferência na) aplicação.

Isto dito, a aplicação e utilização das respectivas estruturas de dados mencionadas, deverão ser inerentes ao contexto da aplicação PON criada. Assim, em determinado momento, uma respectiva estrutura poderá se comportar de maneira mais eficiente em termos de desempenho em relação à outra. Esta avaliação está diretamente ligada ao número de *Premises* relacionado a um *Attribute* e a proporção de variação deste. Outrossim, tal funcionalidade poderia ser implementada no âmbito da ferramenta *Wizard* PON, onde essa adequaria a criação de entidades PON baseado na estrutura da aplicação concebida nela. Ademais, para os casos em que a ferramenta *Wizard* PON utilizasse à estrutura *PONVECTOR* a mesma poderia determinar seu tamanho inicial baseada na quantidade de elementos PON concebidos.

5.3 FERRAMENTA *WIZARD* PON

As ferramentas de geração de código em geral proporcionam a geração de código estrutural de uma aplicação baseadas principalmente em artefatos como o

diagrama de classes da *UML* para a PI, especificamente para a POO. Ainda conforme descrito na seção 2.7, as ferramentas que suportam a *MDA* atenuam e/ou fatoram este problema. Isto se deve, principalmente pelo fato de ambas não gerarem, para qualquer caso, o código da dinâmica da execução da aplicação e na maioria das vezes estarem relacionadas a arquitetura específicas de desenvolvimento, tal como a ferramenta *openSource* denominada *AndroMDA*, a qual relaciona-se especificamente a arquitetura *JEE*.

Na verdade, entretanto, tais dificuldades encontradas são advindas do Paradigma Imperativo (PI) de desenvolvimento, que regem tais ferramentas. No PI, a dinâmica é regida pelos algoritmos orientados a laços de repetição, o que gera os problemas de redundância e acoplamento conforme discutido anteriormente neste trabalho. Isto impele complexidade nos códigos decorrentes e dificuldades de modelagem precedente na fase de análise e projeto. Isto explicaria, em parte, porque as análises e projetos em *UML* valorizam os diagramas de classes, enquanto encontram dificuldades em modelar toda a dinâmica com o ferramental de Diagramas de Sequência, Colaboração, Atividades e Estados. Havendo dificuldade de modelagem da parte dinâmica, haveria também dificuldade de geração de código correspondente.

Neste âmbito, a partir da concepção de aplicações PON sobre o ferramental *Wizard* PON, é possível gerar justamente considerável parte da dinâmica de execução de uma aplicação PON, o qual é realizado a partir da criação de cada *Rule* e suas entidades pertinentes. Deste modo, no âmago da computação em PON, as entidades *Rules* corresponderiam à parte da dinâmica desta execução, havendo assim um ganho em termos de paradigma.

Em PON, efetivamente, tanto a implementação ou Programação Orientada a Notificações quanto na análise/projeto ou Desenvolvimento Orientado a Notificações (DON), as *Rules* tem um papel fundamental na definição de considerável parte da dinâmica da aplicação. De fato, o chamado DON (previamente considerado) prevê estruturas de modelagem conformadas a *Rules*, inclusive as sinérgicas Redes de Petri, o que se constitui inclusive em um ganho ferramental em relação aos atuais diagramas da *UML.*

Tal evidência descrita no parágrafo anterior, em termos de programação particularmente, é constatada pelo caso de estudo relacionado ao Sistema de Vendas, o qual foi inteiramente concebido através da ferramenta *Wizard* PON.

Assim, a dinâmica de sua execução é concebida através da criação das *Rules* em alto nível, as quais correspondem o fluxo de execução de uma aplicação PON. Ainda, o desenvolvedor seria poupado de maiores conhecimentos ou ao menos de maiores manipulações do código PON na sua materialziação vigente, que neste momento é considerada sobre a nova versão *Framework* PON descrito neste trabalho.

De fato, a facilidade de composição de aplicações no PON derivaria do seu reaproveitamento de algumas características do Paradigma Declarativo (PD), particularmente à concepção de aplicações através de composições de regras causais em alto nível. Entretanto, o PON considera os *FBEs, Rules* e as demais entidades que os compõem, como entidades desacopladas. Ainda, possui o processo de inferência inerente a criação destas entidades, diferenciando-se assim da PD, especificamente da programação em SBRs, que dependem de um motor de inferência para ditar o fluxo de execução de aplicações. Sendo assim, a concepção de aplicações no PON seria ainda mais facilitada do que aquelas concebidas por SBR, uma vez que a implementação de aplicações PON é fidedigna a *Rules* elaborada sobre a ferramenta *Wizard* PON, não dependendo fortemente de nuância de máquinas de inferências acoplantes.

Outrossim, a utilização dessa ferramenta poderia garantir que o código-fonte gerado seguiria um conjunto de padrões e boas práticas de implementação (RONSZCKA, 2012), as quais trariam benefícios em relação a possíveis implementações realizadas por desenvolvedores que não dominam totalmente a programação nesse paradigma.

5.4 TRABALHOS FUTUROS

Trabalhos de pesquisa relacionados ao PON vão desde temas relacionados a engenharia de *software* até temas relacionados ao desenvolvimento de aplicações PON em *hardware*. Ainda, fora as próximas seções deste capítulo, BANASZEWSKI (2009) apresentou diferentes vertentes de pesquisa sobre o PON. Algumas estão em andamento como, a programação distribuída e outras se encontram em aberto como, a programação em *hardware*. As próximas seções descrevem as vertentes de

pesquisas para com o Paradigma Orientado a Notificações no âmbito deste trabalho de pesquisa e desenvolvimento.

5.4.1 Compilador PON

Para tornar os programas ainda mais eficientes quando desenvolvidos com o PON, se faz necessário a construção de um compilador particular que otimize as relações entre os objetos participantes do mecanismo de notificações. Estas otimizações incluem a eliminação de estruturas de dados para armazenar os objetos notificados (i.e. *Premises* e *Conditions*), uma vez que os objetos notificantes e notificados seriam conectados em tempo de compilação.

Este compilador poderia consistir em uma extensão de compiladores de código aberto disponível para as linguagens de programação que viessem dar suporte ao PON. Esta extensão se encarregaria de compilar apenas as relações entre os objetos do mecanismo de notificações. Deste modo, os desenvolvedores continuariam fazendo uso da linguagem de programação pertinente e de todas as capacidades do compilador estendido.

Neste cenário, pode-se dizer que o ideal seria uma composição de entidades pertencentes à cadeia de notificação, diretamente via linguagem máquina. Neste sentido, o ideal seria conceber as relações entre os elementos participantes do cálculo lógico causal, através da implementação direta via compilador. Assim, por exemplo, após a instanciação de uma *Rule*, por parte do desenvolvedor, o compilador interpretaria esta definição como uma palavra reservada e assim encadearia as relações entre os demais elementos participantes da cadeia de notiticação, através da linguagem de máquina.

Ainda, este compilador poderia ser integrado à ferramenta *Wizard* do PON e ser usado de maneira transparente ao programador. Com esta integração, a partir do desenvolvimento feito no *Wizard*, o compilador geraria o código de máquina ou *assembly* pertinente sem o programador precisar instigar os serviços deste compilador ou muito menos tomar conhecimento sobre o compilador que está sendo usado.

5.4.2 Ambiente de Desenvolvimento do PON

Em se tratando da ferramenta *Wizard* PON a mesma poderia ser integrada plenamente ao ciclo de Desenvolvimento Orientado a Notificações (DON). Este procedimento poderia ser realizado através da integração de uma ferramenta *CASE* da *UML*, a qual suporte o perfil *UML* proposto nos trabalhos de WIECHETECK (2011) com a própria ferramenta *Wizard* PON. Isto é possível com a comunicação via arquivos *XML*. Na verdade, a metodologia (DON) (WIECHETECK, 2011) e o respectivo ferramental integrados à ferramenta *Wizard* PON se apresentariam como um ambiente integrado ao processo de desenvolvimento de aplicações PON como um todo.

Ademais, a ferramenta *Wizard* PON ainda carece de melhorias, principalmente em relação a adição de certas “inteligências” em seu escopo. Por exemplo, após a concepção de uma aplicação PON, a mesma poderia elencar automaticamente a melhor estrutura de dados (*PONLIST, PONVECTOR* ou *PONHASH*) sobre o viés de cada entidade PON concebida. Ainda, a ferramenta poderia ser capaz de definir o tamanho da estrutura denominada *PONVECTOR* de modo a eliminar a chamada do algoritmo de realocação esboçado na seção 3.1.3. Isto, de fato agregaria em desempenho sobre a execução das aplicações PON.

5.4.3 Ambiente Multiprocessado e Distribuído

Para que o PON seja adotado efetivamente na construção de aplicações multiprocessadas (paralelas e distribuídas) ainda se fazem necessárias algumas melhorias. Nestas melhorias há diferenças para ambientes que compartilham memória entre os nós de processamento (i.e. ambientes paralelos) e ambientes nos quais cada nó apresenta a sua memória particular (i.e. ambientes distribuídos) (BANASZEWSKI, 2009).

Em ambientes distribuídos, faz-se necessário conceber uma plataforma que automatize e torne transparente ao programador todas as questões de comunicação entre os nós de processamento remotos e um mecanismo de balanceamento de carga. A plataforma a ser concebida poderia adotar uma camada de

software/middleware pré-definida como o CORBA (*Common Object Request Broker Architecture*), DCOM (*Distributed Component Object Model*) ou o RMI (*Remote Method Invocation*), que auxiliariam na implementação das abstrações relativas à comunicabilidade entre os nós de processamento (BANASZEWSKI, 2009).

O mecanismo de balanceamento de carga poderia potencialmente ser concebido baseando-se, por exemplo, em soluções fundamentadas em algoritmos evolucionários. Assim, os objetos participantes do mecanismo de notificações poderiam ser distribuídos de forma balanceada e cooperar (por notificações) conforme os endereços definidos na plataforma (BANASZEWSKI, 2009).

Em ambientes paralelos, por sua vez, onde as particularidades de comunicação pela rede são desnecessárias, apenas a implementação do modelo de escalonamento de *Rules* seria suficiente para as aplicações executarem efetivamente. No entanto, em ambos os ambientes se faz necessário um modelo eficaz para a resolução de conflitos entre as *Rules*. Para isto, o modelo de resolução conflitos para CON desenvolvido por Simão (2005) deveria ser utilizado/melhorado (BANASZEWSKI, 2009). Neste sentido, há melhorias descritas nos pedidos de patentes (SIMÃO e STADZISZ, 2010a) e (SIMÃO, *et.al.*, 2010b).

5.4.4 Outras Abordagens Computacionais para o PON

Ainda a criação de novas abordagens computacionais para o PON se faz extremante pertinente. Neste âmbito, trabalhos de pesquisa vêm sendo elaborados e concebidos para tal. Um exemplo é a utilização de técnicas do PON, para a especificação e projeto de sistemas em lógica reconfigurável. Casos de estudo prévios foram concebidos nos trabalhos de (WITT, *et. al,* 2011), no qual o PON se apresentou de fato superior as mesmas implementações feita sobre o POO em lógica reconfigurável, ainda que a implementação POO tenha sido feito sem maiores esforços de otimização conforme destacado em (WITT, *et. al*., 2011) (SIMÃO, *et. al.*, 2012d).

Outrossim, trabalhos relacionados a composição dos conceitos do PON em *hardware* também são pertinentes. Para tal, trabalhos relacionados a implementação de um coprocessador para a aceleração de aplicações em PON está em curso (PETERS, 2012). Neste trabalho, os resultados comparativos demonstraram que a

solução PON apresenta melhor desempenho que a POO. Ademais, esta vantagem aumenta quando o número de condições causais é grande, uma vez que PON se torna ainda mais propício em tais condições.

6 REFERÊNCIAS

AHO, A. V., ULLMAN, J. D. **The Theory of Parsing, Translation, and Compiling, Vol. 1, Parsing**. Englewood Cliffs, New Jersey, EUA: Prentice *HAL*l, 1972. 542 p. 2 vol. vol. 1. ISBN 0-13-914556-7

AHO, A. D. ; SETHI, R. ; ULLMAN, J. D. **Compilers: Principles, Techniques and Tools.** EUA: Addison Wesley, 2001.

AHMED, S. **CORBA Programming Unleashed**. Sams Pub., 1998.

ALBERT, P. **ILOG *Rules*, Embedding *Rules* in C++: Results and Limits.** OOPSLA´94 -Workshop Embedded Object-Oriented Production Systems (EOOPS), 1994.

ALMEIDA, P.; ***MDA – Model Driven Architecture*, Improving Software Development Productivity in Large-Scale Enterprise Applications**. Master Thesis. Software Engineering Group Department of Informatics University of Fribourg Switzerland. December, 2008.

BAKER, H. G.. **Iterators: signs of weakness in object-oriented languages**, ACM OOPS Messenger, Vol. 4, 1993, pp. 18-26.

BANASZEWSKI, R. F. **Paradigma Orientado a Notificações: Avanços e Comparações**. Dissertação (Mestrado em Informática Industrial) – Programa de Pós Graduação em Engenharia Elétrica e Informática Industrial, Universidade Tecnológica Federal do Paraná, Curitiba, 2009. Disponível em: http://arquivos.cpgei.ct.utfpr.edu.br/Ano_2009/dissertacoes/Dissertacao_500_2009.pdf.

BANERJEE, P., CHANDY, J. A., HODGES, M. G., HOLM, J. G., LAIN, A. "**The Paradigm Compiler for Distributed Memory Multicomputer**". IEEE Computer, 28 (10), pp. 37-47, 1995.

BATISTA, M. V.; BANASZEWSKI, R. F.; RONSZCKA, A. F.; VALENÇA, G. Z.; LINHARES, R. R.; STADZISZ, P. C.; TACLA, C. A.; SIMÃO, J. M.. **Uma comparação entre o Paradigma Orientado a Notificações (PON) e o Paradigma Orientado a Objetos (POO) realizado por meio da implementação de um Sistema de Vendas**. COMTEL 2011. Lima, Peru, 2011.

BECK, K. **Extreme Programming Explaned**. Embrace Change. 2000.

BENGHI, F. M., **Emergência de Agentes-Regra a Partir do Plano de Processo de Agentes-Produto, no Âmbito do Meta-Modelo de Controle Holônico e de seu Wizard.** SICITE, Seminário de Iniciação Científica e Tecnológica da UTFPR. Ponta Grossa – Pr, 2011.

BORK M., GEIGER L., SCHNEIDER C., ZÜNDORF A. **Towards Roundtrip Engineering – A template-based Reverse Engineering approach**. Kassel University, *Software* Engineering Research Group. ECMDA-FA '08 Proceedings of the 4th European conference on Model Driven Architecture: Foundations and Applications 2008.

COULOURIS, G., DOLLIMORE, J., KINDBERG, T. **Distributed Systems – Concepts and Designs**. Reading, MA: Addison-Wesley, 2001.

CHENG, A. M. K., CHEN J-R. "**Response Time Analysis of OPS5 Production Systems**". IEEE Transactions on Knowledge and Data Engineering, vol. 12, n.3, pp. 391-409, 2000.

BROOKSHEAR, J. G. ***Computer Science: An Overview***. Addison Wesley, 2006.

DALE, N. (2003). ***C++ Plus Data Structures*****.** Sudbury, MA: Jones and Bartlett Computer Science.

DOLLARD, K. **Code Generation in Microsoft .Net**. EUA: Apress, 2004.

FLOYD, R., W. **The paradigms of programming.** Commun. ACM, New York, NY, USA, Volume 22, Number 8, Pages 455-460, ACM, 1979.

FREEMAN E. T.; ROBSON E; BATES B.; SIERRA K. **Head First Design Patterns**. Publisher: O'Reilly Media. Released: October 2004. Pages: 688.

FORGY, C. ***RETE*****: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem**. *Artificial Intelligence , 19* (1), pp. 17-37, 1982.

FOWLER, M. **Improving the Design of Existing Code** 1. ed. Porto Alegre - RS: Bookman, 2004.

FOG, A. ***Optimizing software in C++. An optimization guide for Windows, Linux and Mac platforms.*** Copenhagen University College of Engineering. Copyright © 2004 - 2011. Disponível em: http://www.agner.org/optimize/optimizing_cpp.pdf. Acessado em: 10 de agosto de 2011.

GABBRIELLI, M., MARTINI, S. "**Programming Languages: Principles and Paradigms. Series: Undergraduate Topics in Computer Science**". 1st Edition, 2010, XIX, 440 p., Softcover. ISBN: 978-1-84882-913-8.

GAMMA, E., HELM, R., JOHNSON, R., VLISSIDES, J. **Design Patterns: Elements of Reusable Object-Oriented *Software***. Addison Wesley, 1995

GAUDIOT, J-L, SOHN, A. "**Data-Driven Parallel Production Systems**". IEEE Trans. on Software Eng.. V. 16. No 3, pg 281-293, 1990.

GEIGER L., SCHNEIDER C., RECORD C. **Template- and model based code generation for MDA-Tools**, 2005.

GIARRATANO, J., RILEY, G. **Expert Systems: Principles and Practice**. Boston, MA: PWS Publishing, 1993.

GIARRATANO, J., RILEY, G. **Expert Systems: Principles and *Programming*.** *Third Edition.* Boston, MA: PWS Publishing, 1998.

GÓES R. M., SIMÃO J. M., STADZISZ P. C., WITT F. A., BANASZEWSKI R. F. **Interface de Composição Facilitada de *Resource-HLs* no Meta-Modelo de Controle de um Simulador de Sistemas de Manufatura Holônico.** XX SICITE – Seminário de Iniciação Científica e Tecnológica da UTFPR. UTFPR. Curitiba – Brasil. 2010.

GRUVER, W. A., "**Distributed Intelligence Systems: A new Paradigm for System Integration**". Proceedings of the IEEE Int. Conference on Information Reuse and Integration (IRI), pg 14-15, 2007.

HERRINGTON, J. **Code Generation in Action**. 4th.ed. Greenwich, CT, USA:Manning Publications Co., 2003.

HUGHES C., HUGHES, T. ***Parallel and Distributed Programming Using C++***. Addison Wesley, 2003.

JOHNSTON W. M., HANNA J. R. P., MILLAR R. J., **Advances in Dataflow Programming Languages.** ACM Computing Surveys, Vol. 36, No. 1, March 2004, pp. 1–34. *University of Ulster.*

JPA, 2012. **Java Persistence API**. Disponível em: http://www.oracle.com/technetwork/articles/javaee/jpa-137156.html. Acessado em: 03/01/2012.

JSF. **JavaServer Faces Technology – Documentation.** Disponível em: http://java.sun.com/javaee/javaserverfaces/reference/docs/. Acessado em: 20/01/2012.

KAISLER, S. ***Software Paradigm*, Wiley-Interscience**, 1st Edition, 0471483478 John Wiley & Sons, 2005.

KANG, A., CHENG, A. M. K. "**Shortening Matching Time in OPS5 Production Systems**". IEEE Trans. on Software Eng. V. 30, N. 7, 2004.

KEITH., M. SCHINCARIOL., M. **Pro JPA 2 Mastering the Java™ Persistence API.** Apress; First edition. December 4, 2009.

KEYES, R., W. **The Technical Impact of Moore's Law**. IEEE solid-state circuits society newsletter. IBM T. J. Watson Research Center, 2006.
KUMAR, V., LEONARD, N., MORSE, A., S. **Cooperative Control**. New York: Springer-Verlag, 2005.

LEE, P.-Y., CHENG, A. M. ***HAL*: A Faster Match Algorithm**. IEEE Transactions on Knowledge and Data Engineering, *14* (5), pp. 1047-1058, 2002.

LINHARES, R. R.; RONSZCKA, A. F.; VALENÇA, G. Z.; BATISTA, M. V.; WITT, F. A.; LIMA, C. R. E.; SIMÃO, J. M. e STADZISZ, P. C.. **Comparações entre o Paradigma Orientado a Objetos e o Paradigma Orientado a Notificações sob o contexto de um simulador de sistema telefônico**. COMTEL 2011. Lima, Peru, 2011.

LUCCA, J. **Modulo de Interface Amigável sobre Meta-modelo de Controle conectado em Ferramenta de Simulação de Sistemas de Manufatura**. Relatório de Iniciação Científica. PIBIC/UTFPR, 2008.

LUCCA, J. **Interface sobre Meta-modelo de Controle do Simulador ANALYTICE II e suas utilizações para Comparações de Políticas de Controle de Manufatura**. SICITE, Seminário de Iniciação Científica e Tecnológica da UTFPR. Pato Branco – Pr, 2009a.

LUCCA, J. **Ambiente de Controle Holônico Sobre o Simulador Analytice II e Comparações de Políticas de Controle de Manufatura.** IX Simpósio Brasileiro de Automação Inteligente - SBAI 2009**.** Brasília, Setembro de 2009b.

LUCCA, J., SILVA, D., **Viabilidade de Integração de Robô Industrial em Ambiente de Programação C++ e da utilização do Controle Orientado a Notificações (CON) em caso real**. PROGRAMA INSTITUCIONAL DE INICIAÇÃO CIENTÍFICA RELATÓRIO FINAL DE ATIVIDADES. Julho de 2010.

MEYERS, S. "**Effective *STL*: *50 Specific Ways to Improve Your Use of the Standard Template Library*.**". Addison-Wesley Professional Computing Series. First Edition, 2001.

MEYERS., Scott. "***Effective C++: 55 Specific Ways to Improve Your Programs and Designs***". Addison-Wesley Professional Computing Series. Third Edition, 2005.

MIRANKER, D. P. **TREAT: A better Match Algorithm for AI Production Systems**. Sixth National Conference on Artificial Intelligence - AAAI'87, (pp. 42-47), 1987.

MIRANKER, D. P., BRANT, D. A., LOFASO, B., GADBOIS, D. **On the Performance of Lazy Matching in Production Systems**. 8th National Conference on Artificial Intelligence AAAI (pp. 685-692). AAAI Press / The MIT Press, 1990.

MIRANKER, D. P., LOFASO, B. **The Organization and Performance of a TREAT-Based Production System Compiler**. IEEE Transactions on Knowledge and Data Engineering , III (1), pp. 3-10, 1991.

NAKAGAWA, E. Y. **Uma contribuição ao projeto arquitetural de ambientes de engenharia de *software***. Tese de Doutorado do Institudo de Ciências Matemáticas e de Computação - ICMC-USP, 2006.

NEWTON., D. **Apache Struts 2 Web Application Development.** Packt Publishing June 15, 2009.

NOYES, B. **Data Binding in Windows Forms 2.0: Programming Smart Client Data Applications with .Net.** EUA: Addison Wesley Professional, 2006.

OLIVEIRA, S., STEWART, D. "**Writing Scientific Software: A Guided to Good Style**". Cambridge University Press, 2006.

PAN, G. N. DeSouza, KAK, A. C. "**Fuzzy Shell: A Large-Scale Expert System Shell Using Fuzzy Logic for Uncertainty Reasoning**". IEEE Transactions on Fuzzy Systems, vol. 6, N. 4, 1998.

PETERS, E. **Coprocessador para Aceleração de Aplicações Desenvolvidas Utilizando Paradigma Orientado a Notificações (PON)**. Dissertação (Mestrado em Informática Industrial) – Programa de Pós Graduação em Engenharia Elétrica e Informática Industrial, Universidade Tecnológica Federal do Paraná, Curitiba, 2012.

PITTMAN, J. **The Pac-Man Dossier**. Disponível em: http://home.comcast.net/~jpittman2/pacman/pacmandossier.html. Acessado em: 16 de Junho de 2011.

PRESSMAN, R., **Engenharia de *Software***. 6ª edição, ISBN: 8586804576, Editora Mc Graw Hill, 2006;

RAYMOND, E.S. **The Art of UNIX Programming**, pp.327, A.Wesley, 2003.

REILLY, D., REILLY, M. **Java Network Programming and Distributed Computing**. Addison-Wesley, 2002.

RONSZCKA, A. F.; BELMONTE, D. L.; VALENÇA, G. Z.; BATISTA, M. V.; LINHARES, R. R.; TACLA, C. A.; STADZISZ, P. C.; SIMÃO, J. M.. **Comparações quantitativas e qualitativas entre o Paradigma Orientado a Objetos e o Paradigma Orientado a Notificações sobre um simulador de jogo**. COMTEL 2011. Lima, Peru, 2011.

RONSZCKA, A. F.; **Contribuição para a Concepção de Aplicações no Paradigma Orientado a Notificações (PON) Sob o Víes de Padrões**. Dissertação (Mestrado em Informática Industrial) – Programa de Pós Graduação em Engenharia Elétrica e Informática Industrial, Universidade Tecnológica Federal do Paraná, Curitiba, 2012.

ROY, P, V., HARIDI, S. "**Concepts, Techniques, and Models of Computer Programming**". MIT Press, 2004.

RUSSEL, S.; NORVIG, P.. **Inteligência Artificial**. ed. Editora Campus. 2003.

SEVILLA, D., GARCIA, J. M., GÓMEZ, A. "**Using AOP to Automatically Provide Distribution, Fault Tolerance, and Load Balancing to the CORBA-LC Component Model**". Parallel Computing: Architectures, Algorithms and Applications, C. Bischof, M. Bücker, P. Gibbon, G. R. Joubert, T. Lippert, B. Mohr, F. Peters (Eds.), John von Neumann Institute for Computing, Jülich, NIC Series, Vol. 38, ISBN 978-3-

9810843-4-4, pp. 347-354, 2007. Reprinted in: Advances in Parallel Computing, Volume 15, ISSN 0927-5452, ISBN 978-1-58603-796-3 (IOS Press), 2008.

SCOTT, M. L. **Programming Language Pragmatics**. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc, 2000.

SIMÃO, J. M. "**Proposta de uma Arquitetura de Controle para Sistemas Flexíveis de Manufatura Baseada em Regras e Agentes**". Dissertação de Mestrado, Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial, CPGEI/UTFPR, Curitiba, 2001. Disponível em: http://arquivos.cpgei.ct.utfpr.edu.br/Ano_2005/teses/Tese_012_2005.pdf

SIMÃO, J. M.; STADZISZ, P. C.; KUNZLE, L. A. **Rule and Agent-Oriented Architecture to Disc*RETE* Control Applied as Petri Net Players. Frontiers in Artificial Intelligence and Applications.** "Advances in Logic, Artificial Intelligence and Robotics" LAPTEC 2003.

SIMÃO, J. M.; STADZISZ, P.C.; MOREL, G.: **Manufacturing Execution System for Customised Production**. Journal of Material Processing Technology (JMPT), Elsevier, Vol. 179, Issues 1-3 20/10/2006 Pp. 268 – Special Issues: 3rd COBEF, 2006.

SIMÃO, J. M.. ***A Contribution to the Development of a HMS Simulation Tool and Proposition of a Meta-Model for Holonic Control***. Tese de Doutorado, Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial, UTFPR, CPGEI, Curitiba, Brasil, 2005. Disponível em: http://arquivos.cpgei.ct.utfpr.edu.br/Ano_2005/teses/Tese_012_2005.pdf

SIMÃO, J. M., STADZISZ, P. C. **Paradigma Orientado a Notificações (PON) – Uma Técnica de Composição e Execução de *Software* Orientado a Notificações**. Pedido de Patente submetida ao INPI/Brasil (Instituto Nacional de Propriedade Industrial) em 2008 e a Agência de Inovação/UTFPR em 2007. No. INPI Provisório 015080004262. Patente submetida ao INPI. Brasil, 2008.

SIMÃO J. M., STADZISZ P. C. **Inference Based on Notifications: A Holonic Metamodel Applied to Control Issues.** IEEE TRANSACTIONS ON SYSTEMS, MAN, AND CYBERNETICS—PART A: SYSTEMS AND HUMANS, VOL. 39, NO. 1, JANUARY 2009.

SIMÃO, J. M.; STADZISZ, P. C. **An Agent-Oriented Inference Engine applied for Supervisory Control of Automated Manufacturing Systems**. In: J. Abe, & J. Silva Filho, Advances in Logic, Artificial Intelligence and Robotics (Vol. 85, pp. 234-241). Amsterdam, The Netherlands: IOS Press Books, 2002.

SIMÃO, J.M., STADZISZ, P.C., KUNZLE, L. ***Rule and Agent-oriented Architecture to DiscRETE Control Applied as Petri Net Player***. (G. Torres, J. Abe, M. Mucheroni, & C. P.E., Eds.) 4th Congress of Logic Applied to Technology - LAPTEC 2003 , 101, p. 217, 2003.

SIMÃO, J. M., TACLA, C. A., STADZISZ, P. C., “**Holonic Control Meta-Model**”. IEEE Transactions on Systems, Man and Cybernetics. Part A, Systems and Humans, 2009.

SIMÃO, J.M., STADZISZ, P.C., TACLA, C. A., BANASZEWSKI, R. F. **“Notification Oriented Paradigm (NOP) and Imperative Paradigm: A Comparative Study**”. A Journal of Software Engineering and Applications, 2012a.

SIMÃO, J.M.; BELMONTE, L.; VALENÇA, G. .Z; LINHARES, R. R.; BANASZEWSKI, R. F.; FABRO, J. A.; TACLA, C. A;. STADZISZ, P.C.; RONSZCKA, A. F.; BATISTA, V. **“Notification Oriented and Object Oriented Paradigms comparison via Sale System**”. Journal of Software Engineering and Applications, 2012b.

SIMÃO, J.M.; BELMONTE, L.; VALENÇA, G. .Z; BATISTA, V.; LINHARES, R. R.; BANASZEWSKI, R. F.; FABRO, J. A.; TACLA, C. A;. STADZISZ, P.C.; RONSZCKA, A. F. **“A Game Comparative Study: Object-Oriented Paradigm and Notification-Oriented Paradigm**”. Journal of Software Engineering and Applications, 2012c.

SIMÃO, J.M., STADZISZ, P.C, LIMA, E. C. R., WITT, F. A., LINHARES, R. R. "**Paradigma Orientado a Notificações em *Hardware* Digital**". Pedido de Proteção Industrial e Pedido de Patente enviados à Agência de Inovação da UTFPR respectivamente em 11/05/2012d e 17/07/2012d, Curitiba - PR, Brasil - Aguardando Aprovação da Agência para eventual envio para o INPI.

SIMÃO, J. M. ; STADZISZ, P. C. **An Agent-Oriented Inference Engine applied for Supervisory Control of Automated Manufacturing Systems**. In: Abe J. M., Silva Filho J. I.. (Org.). Frontiers in Artificial Intelligence and Applications (Advances in Logic, Artificial Intelligence and Robotics LAPTEC 2002). Amsterdam, The Netherlands: IOS PRESS BOOKS, 2002, v. 85, p. 234-241.

SIMÃO, J. M.; STADZISZ, P. C; KUNZLE, L. A. ; SILVA, P. R. O. **Arquitetura de Software de Controle Orientada a Regras e Agentes para Sistemas Automatizados de Manufatura**. In: V Simpósio Brasileiro de Automação Inteligente (SBAI), 2001, Canela-RS.

SIMÃO, J. M.; STADZISZ, P. C; Morel G. . **Manufacturing Execution Systems for Customised Production**. Journal of Materials Processing Technology, v. 179, p. 268-275, 2006.

SIMÃO, J. M.; STADZISZ, P. C; “**Mecanismo de Resolução de Conflito e Garantia de Determinismo para o Paradigma Orientado A Notificações (PON)**”. Patent pending submitted to INPI/Brazil (*Instituto Nacional de Propriedade Industrial*) in 02/2010a and Innovation Agency of UTFPR in 2009. INPI Number: PI1000296-0.

SIMÃO, J. M.; TACLA C. A., BANASZEWSKI R. F., STADZISZ, P. C; “**Mecanismo de Inferência Otimizado do Paradigma Orientado a Notificações (PON) e Mecanismos de Resolução de Conflitos para Ambientes Monoprocessados e Multiprocessados Aplicados ao PON**”. Patent pending submitted to INPI/Brazil (*Instituto Nacional de Propriedade Industrial*) in 03/2010b and Innovation Agency of UTFPR in 2010. INPI Number: PI1003736-5.

STRUTS. **Apache Struts 2 Documentation**. Disponível em: http://struts.apache.org/2.3.1.2/docs/home.html. Acessado em: 28/01/2012.

TANENBAUM, A. S., VAN S., M. "**Distributed Systems: Principles and Paradigms**", Prentice *HAL*l, 2002.
TIANFIELD, H. "**A New Framework of Holonic Self-organization for Multi-Agent Systems**" IEEE Int. Conf. on System, Man & Cyb., 2007.

TILEVICH, E., SMARAGDAKIS, Y. "**J-Orchestra: Automatic Java Application Partitioning**" 16th European Conf. on Object-Oriented Programming, pg 178-204, B. Magnusson (Ed), Springer, 2002.

THOMAS, D. **MDA: Revenge of the Modelers or UML Utopia?.** IEEE Computer Society, June 2004.

TUTTLE, S. M., EICK, C. F. "**Suggesting Causes of Faults in Data-Driven Rule-Based Systems**". Proc. of the IEEE 4th International Conference on Tools with Artificial Intelligence, pg 413-416, Arlington, VA., 1992.

VALENÇA, G. Z.; BANASZEWSKI, R. F.; RONSZCKA, A. F.; BATISTA, M. V.; LINHARES, R. R.; FABRO, J. A.; STADZISZ, P. C.; SIMÃO, J. M.. **Framework PON, Avanços e Comparações**. Simpósio de Computação Aplicada (SCA 2011). Passo Fundo, Rio Grande do Sul, Brasil, 2011.

VANDEVOORDE, D., JOSUTTIS M. N. "***C++ Templates: The Complete Guide***". November 22, 2002.

VELOCITY, 2012. **The Apache Velocity Project**. Disponível em http://velocity.apache.org/. acessado em 10/02/2012.

WACHTER, B. D., MASSART, T., MEUTER, C. "**dSL: An Environment with Automatic Code Distribution for Industrial Control Systems**", Proc. of the 7th Int. Conf. on Principles of Distributed Syst., 2003, La Martinique, France, V. 3144 of LNCS, pg 132-45, Springer, 2004.

WATT, D. **Programming Language Design Concepts**. J. W. & Sons, 2004.

WHITEHEAD, J. **Collaboration in *software* engineering: a roadmap**. FOSE '07 - Future of *Software* Engineering, 2007, pp.214-225, doi: 10.1109/FOSE.2007.4, 2007;

WIECHETECK, L. V. B.. **Um Método para Projetos de *Software* usando o Paradigma Orientado a Notificações**. Dissertação de Mestrado, CPGEI/UTFPR, Curitiba, 2011. Disponível em: http://repositorio.utfpr.edu.br/jspui/bitstream/1/212/1/CT_CPGEI_M_Wiecheteck%2c%20Luciana%20Vilas%20Boas_2011.pdf.

WIECHETECK, L.; STADZISZ, P. C.; SIMÃO, J. M.. **Um Perfil *UML* para o Paradigma Orientado a Notificações (PON)**. COMTEL 2011. Lima, Peru, 2011.

WITT, F. A. **Evolução do Wizard sobre Meta-Modelo de Controle Holônico do Simulador Analytice II e Comparações de Controle Orientado ao Processo e ao Produto**. SICITE, Seminário de Iniciação Científica e Tecnológica da UTFPR. Cornélio Procópio – Pr, 2010a.

WITT, F. A.. **Avanço do Módulo de Interface Amigável (*Wizard*) sobre Meta-modelo de Controle Orientado ao Processo para Orientação ao Produto de um Simulador de Sistemas de Manufatura Holônico**. Relatório de Iniciação Científica. PIBIC/UTFPR, 2010b.

WITT F. A.; SIMÃO J. M.; LINHARES R. R.; STADZISZ P. C.; LIMA, C. R. E. **Comparação Entre o Paradigma Orientado a Objetos (POO) e o Paradigma Orientado a Notificações (PON) em um Controle Discreto em Lógica Reconfigurável.** SICITE, Seminário de Iniciação Científica e Tecnológica da UTFPR. Ponta Grossa – Pr, 2011.

WOLF, W. **High-Performance Embedded Computing: Architectures, Applications, and Methodologies**. Morgan Kaufmann, 2007.