UNIVERSIDADE DO VALE DO RIO DOS SINOS – UNISINOS

CENTRO DE CIÊNCIAS EXATAS E TECNOLÓGICAS

PROGRAMA INTERDISCIPLINAR DE PÓS-GRADUAÇÃO EM

COMPUTAÇÃO APLICADA

TIAGO DA SILVA MINUZZI

USTORY-REFACTORY: FERRAMENTA DE REFATORAÇÃO DE REQUISITOS APLICADA EM CARTÕES *USER STORIES (CRC CARDS)*

São Leopoldo

2007.

UNIVERSIDADE DO VALE DO RIO DOS SINOS – UNISINOS

CENTRO DE CIÊNCIAS EXATAS E TECNOLÓGICAS

PROGRAMA INTERDISCIPLINAR DE PÓS-GRADUAÇÃO EM

COMPUTAÇÃO APLICADA

TIAGO DA SILVA MINUZZI

USTORY-REFACTORY: FERRAMENTA DE REFATORAÇÃO DE REQUISITOS APLICADA EM CARTÕES *USER STORIES (CRC CARDS)*

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

Orientador: Prof. Dr. Sérgio Crespo C. S. Pinto

São Leopoldo

2007.

Ficha catalográfica elaborada pela Biblioteca da
Universidade do Vale do Rio dos Sinos

M668u Minuzzi, Tiago da Silva

Ustory-refactory: ferramenta de refatoração de requisitos aplicada em cartões *user stories* (*CRC Cards*) / por Tiago da Silva Minuzzi. – 2007.
118p. : il. ; 30cm.

Dissertação (mestrado) — Universidade do Vale do Rio dos Sinos, Programa de Pós-Graduação em Computação Aplicada, 2007.
"Orientação: Prof. Dr. Sérgio Crespo C. S. Pinto, Ciências Exatas e Tecnológicas".

1. Engenharia de *software*. 2. Engenharia de requisitos. 3. Refatoração. I. Título.

Catalogação na Publicação:
Bibliotecária Eliete Mari Doncato Brasil - CRB 10/1184

DEDICATÓRIA

Dedico este trabalho as pessoas que me deram à oportunidade de estudar, e de realizar um sonho: meus pais, Ivar e Regina pelo seu exemplo de vida, às meus irmãos e amigos pelo afeto e companheirismos.

AGRADECIMENTOS

- A Deus, por me dar força para lutar atrás dos meus objetivos;
- Aos meus pais, Ivar e Regina, por sempre me darem apoio e acreditarem no meu esforço e capacidade;
- Ao grande companheiro, amigo e professor Dr. Sérgio Crespo Coelho da Silva Pinto, pela orientação, paciência e por ter acreditado na minha capacidade de desenvolver este trabalho;
- Aos amigos, em especial: Leone Cesca, Farlon Souto, Eduardo Lopes, Sidnei Franco, Igor Lorenzato, Paulo Luft, Tiago Dall'Igna, Sabrina Retamal, Cristiane Machado e ao André Tocchetto por suas conversas, conselhos, apoio e troca de idéias;
- A minha namorada e companheira, Vanessa Balim, pela sua compreensão, carinho e o constante incentivo nas horas difíceis;
- A Universidade do Vale do Rio dos Sinos por ter me incluído no projeto da TV Digital fornecendo o incentivo financeiro para finalizar este mestrado;
- Ao Programa Interdisciplinar de Pós-Graduação em Computação Aplicada da Unisinos através de seus professores, colaboradores e colegas de mestrado, pela amizade e oportunidade única de aprendizado adquirada;
- A empresa Tools & Technologies, por ser solidária sempre que requisitei me ausentar em períodos de trabalho;
- A todos que de alguma forma colaboraram para a realização deste trabalho.

"O covarde nunca tenta, o fracassado nunca termina, o vencedor nunca desiste." **(Norman Vincent Peale)**

RESUMO

O surgimento de novas metodologias ágeis para apoiar o desenvolvimento de sistemas, como a *Extreme Programming* (XP), vem causando impacto nas empresas de desenvolvimento de *software*, especialmente por sua flexibilidade nas mudanças de requisitos no decorrer do projeto. Assim, um melhor entendimento e representação estrutural dos requisitos tornam-se fundamental. Logo, esta pesquisa aplica o conceito das técnicas de refatoração de código dentro da Engenharia de Requisitos, que é focado na metodologia XP, por meios das *User Stories*. O trabalho aplica um conjunto de padrões e regras que permite aos requisitos expressos em cartões CRC serem refatorados através de pré e pós-condições, sendo que esses requisitos são descritos por mapas conceituais (MC) em formato OWL. Por sua vez, os MCs são convertidos em diagramas de classes da UML por meio da UML-MC que formaliza esta transformação. Dessa forma, o ambiente *UStory-Refactory* automatiza parcialmente o processo de refatoração e permite que os requisitos refatorados sejam exportados em formato OWL, promovendo, então, uma interoperabilidade entre diversos sistemas de *software*. Para o correto funcionamento da solução é necessário que seja utilizada por um analista de sistemas sênior, que tenha conhecimentos sobre processos de construção de *software*. Este garante a consistência e eficiência dos requisitos refatorados.

Palavras-Chaves: Engenharia de *Software,* Refatoração, Engenharia de Requisitos, *Extreme Programming,* UML, Ontologias.

ABSTRACT

The emergence of new agile methodologies to support systems development, as the Extreme Programming (XP), has been causing impact on software development companies, specially for its flexibility in the requirements changes during the project. Thus, a better understanding and structural representation of the requirements become basic. Then, this research applies the concept of the code refactoring techniques, inside of the Requirements Engineering, which is focused at XP methodology, through the User Stories. The work applies a set of standards and rules that allows the requirements expressed in CRC cards to be refactored through pre and post-conditions, and the requirements are described for conceptual maps (CMaps) in OWL format. In their turn, the CMaps are converted into UML classes diagrams by the UML-MC that formalizes this transformation. This way, the UStory-Refactory environment partially automatizes the refactoring process and allows the refactored requirements to be exported in OWL format, promoting interoperability among diverse software systems. For the correct functioning of the solution it is necessary that it is used by a senior systems analyst, who has knowledge on software process construction. This guarantees the consistency and efficiency of the requirements refactoring.

Key words: Software Engineering, Refactoring, Engineering Requirement, Extreme Programming, UML, Ontology.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

API	*Application Programming Interface*
COM+	*Extension of Component Object Model*
CORBA	*Common Object Request Broker Architecture*
COTS	*Commercial off-the-shelf software*
CRC	*Class, Responsibility and Collaboration*
DAML	*DARPA Agent Markup Language*
DOM	*Document Object Model*
DTD	*Document Type Definition*
GUI	*Graphical User Interface*
HP	*Hewlett-Packard*
HTTP	*HiperText Transfer Protocol*
IBM	*International Business Machines*
IDE	*Integrated Development Environment*
J2EE	*Java 2 Platform Enterprise Edition*
KIF	*Knowlege Interchange Format*
KMG	*Knowledge Modeling Group*
KSL	*Knowledge Systems Laboratory*
LEL	Linguagem Léxica Estendida
ODM	*Ontology Definition Metamodel*
OIL	*Ontology Inference Laye*
OWL	*Web Ontology Language*
RDF	*Resource Description Framework*
SAX	*Simple API for XML*
SGML	*Standard Generalized Markup Language*
SOAP	*Simple Object Access Protocol*
SRS	*Software Requirements Specification*
SWING	*Sun's lightweight GUI*
SWT	*Standard Widget Toolkit*
UML	*Unified Modeling Language*
URI	*Uniform Resource Identifier*
URL	*Uniform Resource Locators*

W3C	*World Wide Web Consortium*
WEB	*World Wide Web*
XML	*eXtensible Markup Language*
XP	*Extreme Programming*

SUMÁRIO

1. INTRODUÇÃO

Ao longo dos últimos anos, para que ocorresse o desenvolvimento de *softwares* com eficácia e qualidade, fez-se necessário determinar processos de execução de *software*. Surge, assim, a Engenharia de *Software,* que é utilizada para a criação e emprego disciplinado de princípios e métodos no projeto e construção de *software* de qualidade e de forma adequada (BOEHM, 1979).

Sendo assim, para a aplicação da Engenharia de *Software* no cenário contemporâneo, foram definidos elementos fundamentais que, segundo Pressman (PRESSMAN, 2002), são: métodos, ferramentas e procedimentos; sendo que os métodos detalham o projeto de desenvolvimento do *software*; as ferramentas garantem o apoio automatizado ao mesmo; e os procedimentos realizam as ligações entre os métodos e as ferramentas. Além disso, o processo de construção de *software* sofreu, com o passar do tempo, modificações que foram responsáveis pelo surgimento de novos paradigmas referentes à tecnologia de informação.

Portanto, o foco deste trabalho se concentra nos processos de construção de *softwares,* com ênfase para a Engenharia de Requisitos e a técnica de refatoração, consideradas como procedimentos que garantem a eficiência e eficácia no desenvolvimento de um *software*. Sendo que a primeira tem o objetivo de verificar as necessidades dos clientes/usuários através de um ciclo de vida por etapas, as quais garantem a excelência da execução do *software* (KOTONYA, 1998). Já a segunda é uma técnica de desenvolvimento pouco aplicada nos projetos, e tem o objetivo de proceder a uma modificação no código, garantindo que o comportamento externo da estrutura não seja alterado, e que a consistência do comportamento interno do projeto permaneça (VITTEK, 2003; FOWLER, 2004). Em outras palavras, a refatoração é utilizada como um processo de evolução do *software*, através da limpeza do código, realizando a extração do código duplicado, procedimento esse que proporciona a minimização dos *bugs* do sistema. Além disso, a refatoração é um processo em evolução, à medida que atualmente se apresenta com intensidade em uma das fases do ciclo de vida da metodologia de desenvolvimento *Extreme Programming* (XP) (BECK, 2000).

No entanto, as etapas do processo de Engenharia de Requisitos possuem algumas questões em aberto, destaca-se a etapa de elicitação de requisitos o problema de definição de

requisitos. Esta etapa é considerada uma fase crítica, pois através dela é possível identificar os requisitos funcionais e não-funcionais, que são dados levantados com os *stakeholders* (NUSEIBECH, 2000). Segundo Lamsweerde (LAMSWEERDE, 2000), a definição correta dos requisitos de um *software* é fundamental para a execução de uma operação bem-sucedida do mesmo, uma vez que os requisitos formam a base para o planejamento, acompanhamento e aceitação dos resultados do projeto.

Por outro lado, o desenvolvimento inadequado na definição dos requisitos gera conseqüências negativas, tais como: a produção de sistemas que não atendem às necessidades dos usuários, fato que causa aumento dos custos; a realização de atividades desnecessárias, que gera usuários insatisfeitos; o desentendimento entre engenheiros de requisitos; e o aumento da tarefa de trabalho (CARVALHO, 2002). Para evitar tais conseqüências, existem alguns procedimentos utilizados para a elicitação de requisitos, dentre os quais podemos citar as observações, as entrevistas, os modelos de caso de usos, os cenários e os cartões *User Stories (CRC Cards)* da metodologia XP, cujo objetivo é realizar o levantamento de requisitos.

1.1. MOTIVAÇÃO

As motivações desse trabalho são as pesquisas observadas na área da Engenharia de S*oftware* que englobam a técnica de refatoração e o processo de Engenharia de Requisitos, desde a elaboração conceitual até a prática, que é o desenvolvimento de ferramentas mostradas em pesquisas como de Opdyke (OPDYKE, 1992), Roberts (ROBERTS, 1999), e Lamsweerde (LAMSWEERDE, 2000).

O objetivo do processo de Engenharia de Requisitos é proporcionar o desenvolvimento do *software* de uma forma eficiente e eficaz, através da identificação dos *stakeholders* e de suas necessidades. No entanto, esse processo possui um estágio crítico, dado pela descoberta dos requisitos, e vem sendo executado de forma ineficiente, podendo-se dizer, inclusive, que o desenvolvimento do *software* fica comprometido ocasionando em atraso na entrega do *software* e este não atendendedo de forma adequada o cliente (NUSEIBECH, 2000). O processo de descoberta de requisitos envolve a utilização de métodos de elicitação de requisitos, que são conhecidos pelos engenheiros de *software*. Conforme dito na seção anterior, existem distintas formas de elicitar um requisito, mas este requisito pode apresentar-se em uma forma ambígua (redundante), tornando-se necessário, às vezes, aplicar-lhe a técnica de refatoração. Segundo Xu (XU, J., 2004), refatorar um requisito

é tornar ele mais simples e compreensível ao programador, buscando capturar novos requisitos e comportamentos que proporcionem uma melhor visibilidade do desenvolvimento do *software*.

A refatoração é uma área que tende a evoluir, e, segundo Kataoka (KATAOKA, 2001), ela deve ser aplicada constantemente no desenvolvimento de um projeto, pois somente assim torna-se possível prover uma consistência do *software,* através da aplicação da refatoração ao seu código fonte. A evolução do *software* é comprovada também em *"Cascaded Refactoring"* (XU, L., 2005), que provê a evolução do projeto de *software*, através da criação de um *framework*, e propõe a utilização da refatoração em todas as etapas do processo, desde a análise de requisitos até o seu código fonte. Esse fato proporciona eficiência nas alterações e documenta todas as fases da evolução do *software*.

A técnica de refatoração, no cenário contemporâneo, está sendo abordado no processo de Engenharia de Requisitos, por meio desse, é possível elevar o nível de abstração e prover um entendimento melhorado do problema, fato que permite determinar uma mudança eficaz, conseguindo proporcionar o reaproveitamento dos dados e a manutenção dos componentes do *software* (XU, J., 2004; YU, 2004). A comprovação desse processo pode ser obtida pela aplicação da técnica de refatoração aos requisitos, ou seja, ocorre no momento em que são desenvolvidos os diagramas de casos de uso e de cenários, proporcionando uma evolução dos requisitos.

1.2. PROBLEMA

Estudos realizados na área da Engenharia de *Software* demonstram que uma imensa quantidade de projetos de *software* é cancelada ou fracassa, porque não atendem completamente às necessidades dos *stakeholders,* ou até mesmo porque ultrapassam os prazos estipulados (ANKORI, 2004; BUTLER, 2001; KOTONYA, 1998). Não há uma explicação específica para esse caso, mas existem trabalhos que apontam as deficiências no processo de Engenharia de Requisitos, mostrando como ocorrem os fracassos nos projetos de *software* e os prejuízos ao orçamento dos mesmos (ANKORI, 2004; XU, J., 2004).

A eficiência e a exatidão de um requisito é um grande desafio para o sucesso do trabalho, sobretudo quando o enfoque recai no desenvolvimento de projetos de *software*. Isso ocorre porque a etapa de elicitar requisitos é uma fase complexa de um sistema de *software*, pois é nesse momento que se determina precisamente o que será construído, estabelecendo,

detalhadamente, os requisitos técnicos (LAMSWEERDE, 2000). Apesar de existirem variadas maneiras de elicitar requisitos, há uma técnica que se mostra eficiente no momento de melhorar a sua especificação, denominada técnica de refatoração. Segundo Xu (XU, L., 2005), não há uma regra definida para aplicar a refatoração sobre os requisitos, o que deve ser levado em conta é a determinação das pré-condições e pós-condições da aplicação da técnica de refatoração. O ponto crítico no desenvolvimento deste trabalho é, portanto, **a forma como deve ser aplicada a técnica de refatoração de código nas especificações de requisitos.**

Com o passar do tempo, a refatoração tornou-se uma etapa do processo e uma solução importante no desenvolvimento do *software*, uma vez que pode ser aplicada como transformação ou melhoramento de um projeto. Estudos realizados nessa área apontam que a técnica de refatoração, hoje, pode ser aplicada em diversas etapas no processo de desenvolvimento do *software*, sempre com o objetivo de achar aspectos redundantes e melhorá-los (YU, 2004). A solução proposta e desenvolvida neste trabalho tem o objetivo de melhorar a definição de requisitos através de padrões de refatoração que devem ser aplicados aos mesmos.

1.3. QUESTÃO DE PESQUISA

A questão específica deste trabalho está centrada na seguinte indagação: "Como refatorar especificações de requisitos no formato de cartões *User Stories (CRC Cards)* da metodologia de desenvolvimento *Extreme Programming?"*

1.4. OBJETIVOS

Este trabalho tem como objetivo principal modelar e desenvolver uma ferramenta que aplique a técnica de refatoração, em uma base de requisitos que representam o formato de cartões em *User Stories (CRC Cards)*, utilizados na metodologia de desenvolvimento XP.

A solução proposta para tal formulação possui uma arquitetura única – mostrada na Figura 1 – que, para ser desenvolvida, exigiu a concretização dos seguintes objetivos específicos:

- Estudar em profundidade tecnologias necessárias para o desenvolvimento da aplicação, tendo como ponto de partida as tecnologias: XML, RDF, OWL e API *Jena Ontology*;

- Aprofundar o estudo das técnicas de refatoração de código, tendo o objetivo de obter heurísticas para serem utilizadas para a refatoração de requisitos;

- Modelar uma ontologia no formato OWL, para a representação dos requisitos em cartões *User-Stories (CRC Cards)*, representando a camada de interoperabilidade da ferramenta;

- Definir padrões (modelos) de refatoração em cartões *User-Stories (CRC Cards)*, por meio de regras e restrições que desenvolvam o conceito da refatoração, para a concretização da camada de regra de negócio;

- Modelar os recursos a serem desenvolvidos por meio da UML (BOOCH, 2006), utilizando conceito e modelagem de especificação de *software*;

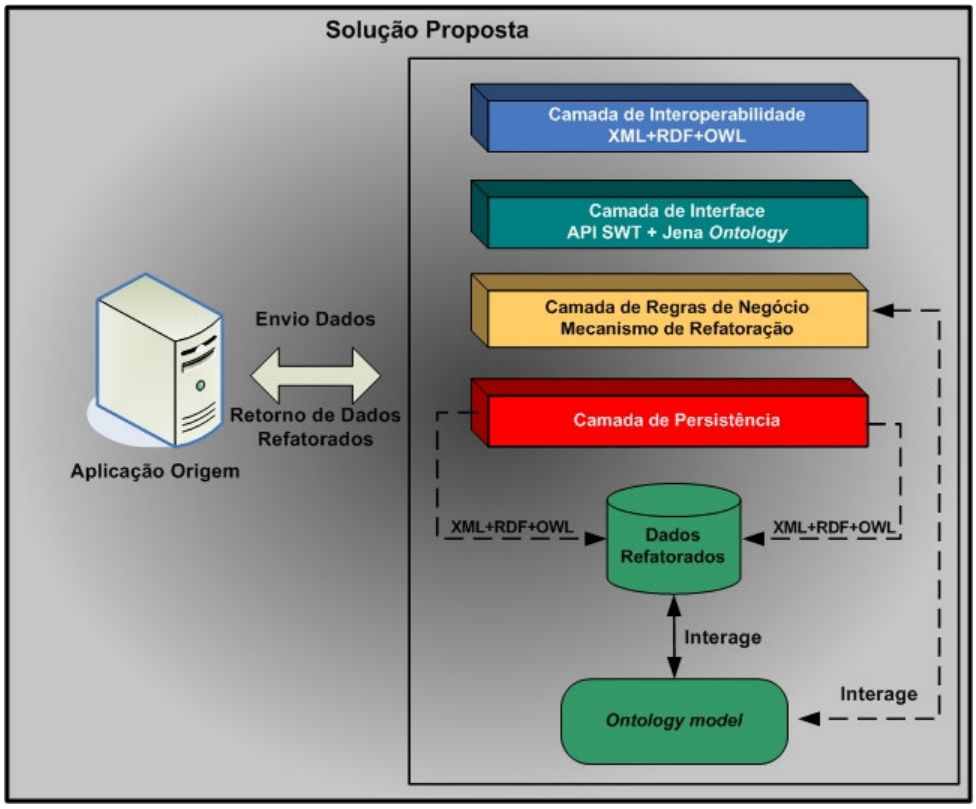


Figura 1: Solução proposta

Os principais aspectos favoráveis identificados nesta solução são:

- Proporcionar requisitos refatorados, garantindo a eliminação de redundâncias e ambigüidades para que o analista de *software* realize uma especificação do projeto de forma eficiente e eficaz;

- O desenvolvimento da ferramenta através de quatro camadas, proporcionando organização e delimitando tarefas a cada camada, o que pode ser analisado na Figura 1;

- O uso de tecnologias do tipo XML, RDF e OWL, garantindo uma ontologia, com o objetivo de proporcionar a interoperabilidade através da semântica dos dados entre aplicações;

- Apresentar uma nova forma de representar os diagramas de classes da linguagem de modelagem UML através da formalização dos dados no formato da linguagem OWL. Cabe ressaltar ainda que essa linguagem formaliza os diagramas de classe da UML por intermédio de uma forma abstrata;

- Aplicação da refatoração de requisitos dentro de um novo contexto, ou seja, na especificação de requisitos em cartões *User Stories (CRC Cards)* da metodologia de desenvolvimento XP, eliminando redundância na etapa de especificação de requisitos dessa metodologia.

É importante ressaltar ainda que a ligação e a interação entre a aplicação origem e destino estão fora do escopo deste trabalho, uma vez que, embora a relevância desse aspecto seja reconhecida, esse ponto não pôde ser desenvolvido, devido à necessidade de definir limites para a execução do trabalho.

1.5. ORGANIZAÇÃO DO VOLUME

Este volume está divido em sete capítulos, sendo que o primeiro a introdução, e os demais são:

- Capítulo 2: Conceitos Fundamentais – fase em que são descritos alguns conceitos fundamentais para o entendimento do contexto do trabalho. Assim, o texto fala da Engenharia de Requisitos e suas maneiras de elicitar requisitos, da técnica de refatoração e das ontologias;

- Capítulo 3: Tecnologias Utilizadas – nesse capítulo são descritas as tecnologias utilizadas para o desenvolvimento da aplicação, dentre as quais se destacam as do tipo XML, RDF, OWL e API *Jena Ontology*;

- Capítulo 4: Trabalhos Relacionados – apresenta a leitura de alguns trabalhos acadêmicos cujo tema se aplica à refatoração de requisitos;

- Capítulo 5: *UStory-Refactory* – descreve a solução desenvolvida, por meio de um estudo de caso descrevendo a aplicação dos padrões de refatorações;

- Capítulo 6: Estudo de Caso – descreve a utilização dos padrões de refatorações no início do desenvolvimento do ambiente *"AulaNet"*;

- Capítulo 7: Conclusão – apresenta algumas considerações quanto ao desenvolvimento do trabalho, indicando pontos de melhoria, bem como a continuidade do trabalho.

2. CONCEITOS FUNDAMENTAIS

Este capítulo tem por objetivo apresentar uma revisão bibliográfica sobre os assuntos que serão abordados nesta dissertação de mestrado.

2.1. ENGENHARIA DE REQUISITOS

Nesta subsecção, pretende-se abordar a Engenharia de Requisitos, de uma forma genérica, mostrando definições básicas de realização de procedimentos e de formas encontradas para elicitar requisitos. Por fim, será explicado como ocorre o descobrimento dos requisitos dentro da metodologia de desenvolvimento XP (*Extreme Programming*), pois é nesse tipo de requisito que está centrado o foco deste trabalho.

Primeiramente, é preciso salientar que o processo de especificação de um sistema para o computador pode ter diferentes níveis de aceitação. Assim, para os engenheiros de hoje, torna-se um desafio saber se o sistema foi corretamente especificado e se atende às expectativas dos clientes. No entanto, não existe uma resposta única para essa questão, porém, existe um procedimento sólido para a sua identificação: a Engenharia de Requisitos, que busca cobrir todas as atividades envolvidas no descobrimento, documentação, análise e manutenção de um conjunto de requisitos para um sistema baseado em computador. O uso do termo "engenharia" consiste em aplicar técnicas sistemáticas e repetitíveis a ser usadas para assegurar que os requisitos do sistema sejam completos, relevantes e eficazes (SOMMERVILLE, 1997).

De acordo com Nuseibech (NUSEIBECH, 2000), a Engenharia de Requisitos é uma etapa da Engenharia de *Software* que tem por objetivo realizar um levantamento das funções e necessidades de um *software*, bem como também identificar os *stakeholders*, que podem ser clientes, usuários finais ou a equipe de desenvolvimento do *software*, e suas necessidades, e documentar todo o processo de análise e documentação das etapas de desenvolvimento de *software*. No entanto, recentes estudos comprovam que, ao desenvolver um *software*, um dos principais problemas ocorridos é verificado no processo de Engenharia de Requisitos. Por exemplo, através da análise de 8.000 projetos de 350 companhias norte-americanas, foi possível observar que a maioria dos projetos obteve apenas acabamento inadequado devido a uma análise precária da Engenharia de Requisitos, o que acarretou aumento de custos e de cronograma de entrega (LAMSWEERDE, 2000).

O desenvolvimento correto da Engenharia de Requisitos ocorre através da identificação dos objetivos e operações que o sistema deverá suportar, e também busca descobrir responsabilidades para os agentes, tais como com os seres humanos e do *software*. Estudos contemporâneos comprovam que a preocupação das empresas é crescente com relação aos processos de Engenharia de Requisitos, pois através deles torna-se viável a elaboração de projetos em menor tempo, fato que proporciona ao cliente uma maior eficiência do produto desenvolvido (LAMSWEERDE, 2000).

2.1.1. Tipos de requisitos

Segundo Wiegers (WIEGERS, 2003), são usados muitos termos comuns para especificar a Engenharia de Requisitos. Assim, os requisitos de *software* são incluídos em três níveis distintos: de negócios, de usuários e funcionais. Além disso, os sistemas terão uma grande variedade de requisitos não-funcionais. A Figura 2 ilustra diversos tipos de requisitos que podem ser utilizados na especificação de um sistema.

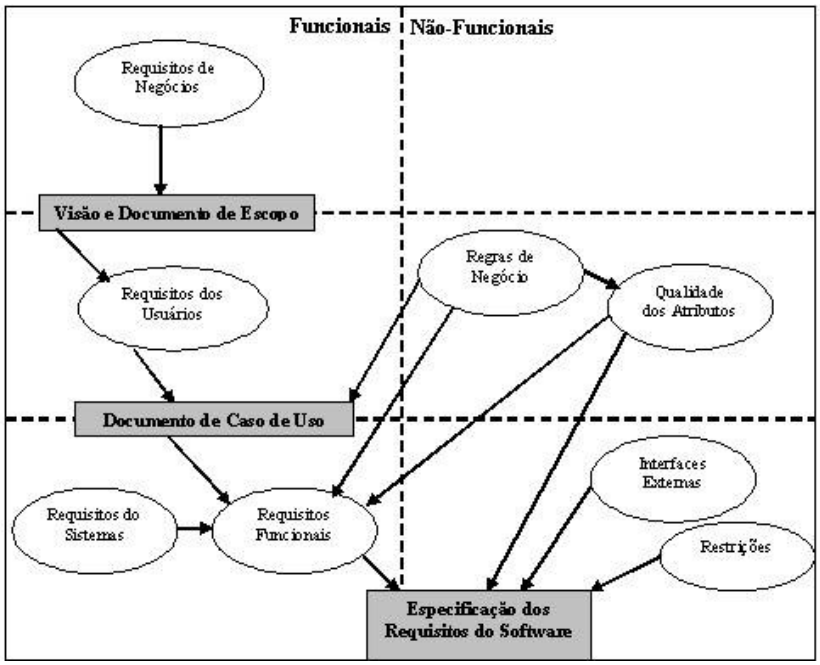


Figura 2: Relacionamento entre diversos tipos de requisitos (WIEGERS, 2003)

De acordo com a Figura 2, os requisitos de negócios representam os objetivos da organização ou as expectativas do cliente num nível mais elevado, sendo que, através deles é possível ter uma visão geral sobre o procedimento e a obtenção de um documento de escopo do sistema. Por sua vez, os requisitos dos usuários descrevem os objetivos e tarefas dos

usuários com o sistema, e podem ser representados através de documentos de casos de uso, de descrições de cenários e de eventos; e os requisitos funcionais especificam a funcionalidade do *software*, que habilita o usuário a desempenhar tarefas que satisfaçam aos requisitos de negócios. Já os requisitos de sistema dependem de um outro requisito para o seu correto funcionamento, como, por exemplo, ocorre num sistema operacional. As regras de negócios são políticas de sistemas, e não são consideradas requisitos, mas atuam para especificar as qualidades dos atributos que são implementados. Os requisitos funcionais são documentados em uma especificação de requisitos de *software* denominada *Software Requirements Specification* (SRS), que descreve todo o comportamento necessário do *software*. A SRS contêm também requisitos não-funcionais como qualidade dos atributos, descrevendo, assim, a funcionalidade do *software* através de características como portabilidade, eficiência e usabilidade. Outro requisito não-funcional que pode ser mencionado são as interfaces externas, que descrevem a relação existente entre o sistema e o mundo real, entre o projeto e a implementação das restrições, as quais representam exceções que devem ser levadas em consideração no desenvolvimento do projeto. Na Figura 2, foram analisados os requisitos em um fluxo *"top-down"*, ou seja, fica explícito o procedimento que descreve a interação entre os requisitos de negócio, de usuários e os funcionais.

Logo, os requisitos se dividem em dois grupos:

Requisitos funcionais: são aqueles que estão explícitos, ou seja, que são definidos como as funções ou as atividades que o sistema executa (ou executará), descritos de maneira clara e formal. De acordo com Mylopoulos (MYLOPOULOS, 1992), estes requisitos são fundamentais para qualquer projeto de *software* porque é onde fica definido o que o sistema fará através dos requisitos globais dos custos de desenvolvimento e dos custos operacionais, proporcionando, assim, a abrangência das questões de qualidade, que são os requisitos não-funcionais.

Requisitos não-funcionais: são aqueles que não podem ser definidos em termos de funcionalidade. A crescente necessidade de definir modelos conceituais capazes de lidar com metas, retratando a necessidade do mundo real, proporcionam a capacidade de representar requisitos não-funcionais, como confidencialidade, performance, qualidade e precisão. Com o crescente aumento da tecnologia, ocorre o surgimento de impactos, causados por restrições operacionais de *software* e de evolução da tecnologia; isso se torna hoje um grande desafio no momento de definição dos requisitos não-funcionais. Como exemplo de trabalhos

desenvolvidos na área, pode ser citado o esforço de Mylopoulos (MYLOPOULOS, 1992), que propõe um *framework* para representar os requisitos não-funcionais no processo de desenvolvimento do *software*, o qual é composto por cinco elementos básicos que provêm a representação dos requisitos não-funcionais através dos seus objetivos. Para cada objetivo é possível observar a evolução dos requisitos não-funcionais, e tais requisitos ainda buscam metas conflitantes, nas quais são identificadas, genericamente, e refinadas até que se chegue a um conjunto de requisitos que satisfaçam o principal. Este trabalho ainda busca ajudar nas decisões relativas ao domínio da aplicação, e auxiliar na detecção de erros provenientes da utilização desse método.

2.1.2. Desenvolvimento dos requisitos

O processo de desenvolvimento dos requisitos pode ser descrito em quatro atividades básicas, que são: a elicitação, a análise e negociação, a documentação, e a validação de requisitos. Tais atividades têm o objetivo de melhorar o desenvolvimento do *software,* provendo a evolução e documentação dos requisitos do seu projeto, e são descritas a seguir (WIEGERS, 2003).

- **Elicitação de Requisitos:** é a etapa em que ocorre o levantamento dos requisitos existentes e necessitados pela organização, sejam eles funcionais (funções ou atividades que o sistema faz ou fará) ou não-funcionais (restrições que se colocam sobre como o sistema deve realizar seus requisitos funcionais). Em outras palavras, nessa fase são identificadas as necessidades dos *stakeholders* com relação ao *software* a ser desenvolvido (NUSEIBECH, 2000). Cabe informar, porém, que a realização dessa etapa não é uma tarefa simples, pois podem ocorrer algumas incorreções ou dificuldades, tais como: problemas de escopo (quando o limite do sistema está mal definido, devido ao fato de o analista especificar requisitos desnecessários), problemas de entendimento (quando os *stakeholders* e o engenheiro de sistemas têm dificuldade ao se comunicar) e, por último, o problema de volatibilidade (o engenheiro projeta o sistema, mas não analisa que os requisitos podem mudar ao longo do tempo) (PRESSMAN, 2002). Essa etapa pode ser considerada a mais importante e a mais complicada, se for comparada com as demais. Das inúmeras ocorrências sobre projetos para a elicitação de conhecimento, tomamos como exemplo o desenvolvido por Ankori (ANKORI, 2004), que considera um sistema de aprendizado para a automação de alguns aspectos da fase de requisitos de *software*. Este trabalho engloba a aquisição de informações através da utilização de uma base de dados de

conhecimento. O sistema é embasado na metodologia de *Tecuci*, que é conhecida como *Disciple-MTL*, *Multistrategy Task-Adaptative Learning by Justification Trees Algorithm,* algoritmo cujo objetivo é coletar informações de vários *stakeholders* e integrar uma variedade de métodos de aprendizado (heurísticas) no processo de aquisição de conhecimento. O objetivo dessa manipulação é criar uma lista de requisitos essenciais ao desenvolvimento do sistema, através do processo de Engenharia de Requisitos.

- **Análise dos Requisitos:** após a complementação da primeira etapa, realizam-se a análise dos requisitos levantados, descobrindo possíveis conflitos e necessidade de elicitar outros requisitos. A análise distribui os requisitos em categorias, explorando as relações entre eles, e classifica-os segundo critérios de importância, tendo como parâmetro as necessidades dos *stakeholders*. De acordo com a consistência, omissões e ambigüidade, ordenam-se os requisitos quanto ao seu grau de importância para o desenvolvimento do projeto (NUSEIBECH, 2000). Nessa etapa, é comum ocorrerem erros conflitantes, pois o cliente ou os usuários propõem requisitos semelhantes, ou seja, que têm a mesma função no *software*. Segundo Pressman (PRESSMAN, 2002), para que isso não ocorra, perguntas são formuladas e respondidas logo ao início da atividade de análise de requisitos.

- **Especificação dos Requisitos:** nessa etapa, todos os requisitos são documentados com um nível apropriado de detalhe. Geralmente, para validar o processo é construído um documento de especificação de requisitos, de forma que se torne compreensível por todos os *stakeholders*. Caso um *stakeholders* não consiga entender o documento, recomenda-se rever o ciclo do processo da Engenharia de Requisitos (LAMSWEERDE, 2000). Nessa etapa é possível, ainda, identificar recursos necessários aos requisitos, unificar os requisitos, determinar regras de negócio aos requisitos e especificar o nível de qualidade dos requisitos (WIEGERS, 2003).

- **Validação dos Requisitos:** por último, essa etapa possui como objetivo examinar a especificação para garantir que todos os requisitos do sistema tenham sido declarados sem ambigüidades, inconsistências ou omissões. Caso tenha ocorrido a detecção de algum erro, ele deve ser corrigido para garantir qualidade ao projeto. Existem algumas maneiras de realizar um processo de validação de requisitos, o que pode ser recomendado é definido por Pressman (PRESSMAN, 2002), que adota a análise de

cada requisito através de um conjunto de *cheklist*, em outras palavras, um conjunto de perguntas. Assim, nessa etapa ocorrem três importantes processos, os quais são: a inspeção dos documentos dos requisitos, os testes dos requisitos e a definição de um critério de aceitação para os requisitos (WIEGERS, 2003).

A Figura 3 ilustra esse processo de desenvolvimento dos requisitos, no qual as atividades de elicitação, análise, documentação e validação ocorrem em uma seqüência linear, de forma evolutiva e incrementada em todas as etapas (WIEGERS, 2003).

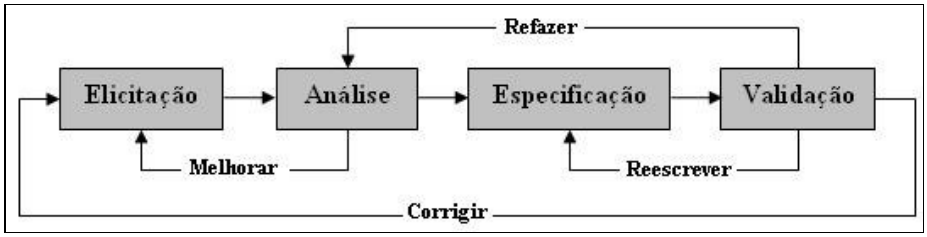


Figura 3: Processo interativo do desenvolvimento dos requisitos (WIEGERS, 2003)

2.1.3. Como elicitar requisitos?

Com a evolução das tecnologias no processo de desenvolvimento de *software*, novas técnicas são desenvolvidas para aperfeiçoar a especificação dos requisitos para um projeto de *software*. Assim, podem-se citar as técnicas que hoje são aplicadas na etapa de especificação de requisitos, quais sejam: as observações, as entrevistas, os casos de uso, os cenários e os *User Stories* e *CRC Cards,* sendo que todos possuem um objetivo comum, que é especificar os requisitos de *software* de uma maneira legível para o usuário e para o desenvolvedor (NUSEIBECH, 2000; BOOCH, 2006; REES, 2002).

2.1.3.1. Observação

A observação será o primeiro contato pessoal entre o pesquisador e o objeto da pesquisa, e isso possibilita algumas vantagens, uma vez que é permitido descobrir aspectos novos de um problema que já havia sido determinado. Como todo processo, também apresenta alguns aspectos negativos, uma vez que podem ocorrer alterações no ambiente ou comportamento das pessoas observadas, e por basear-se na interpretação pessoal, pode ocorrer que os pontos de vista apresentados sejam discordantes, ocasionando uma visão distorcida do fenômeno ou da representação da realidade (LAMSWEERDE, 2000). Pode-se concluir que essa técnica é válida quando proporciona ao engenheiro ter um escopo do problema, para o qual ele irá propor uma solução.

2.1.3.2. Entrevistas

É uma das técnicas mais utilizadas no momento da especificação de requisitos com o usuário, e ocorre quando o engenheiro de requisito ou analista discute o sistema com diferentes usuários, e, a partir daí, elabora um entendimento e compreensão dos requisitos. Segundo Pressman (PRESSMAN, 2002), há dois tipos de entrevistas: as fechadas, nas quais o engenheiro de requisito procura abordar perguntas específicas para um conjunto pré-definido de usuários; e as abertas, em que não há uma agenda pré-definida e o engenheiro aborda o assunto de modo aberto para ver o que os usuários querem do sistema. A entrevista é uma técnica estruturada que pode ser aprendida e os desenvolvedores podem conquistá-la com prática e treinamento, além disso, ela ainda permite que os desenvolvedores entendam os processos atuais da organização e percebam o que está faltando no sistema e as expectativas dos usuários em relação a esse novo sistema. No entanto, Sommerville (SOMMERVILLE, 1997) define que as entrevistas são menos efetivas para se ter um entendimento do domínio da aplicação.

2.1.3.3. Casos de Usos e Cenários

Atualmente, os modelos de casos de uso são técnicas bastante utilizadas na etapa de elicitação de requisitos para a explicitação de todas as diferentes funcionalidades, o que permitirá inferir e identificar claramente necessidades dos usuários; em outras palavras, são utilizados para especificar o comportamento do sistema ou parte do sistema, descrevendo um conjunto de seqüência de ações (BOOCH, 2006). Esse tipo de modelo é o primeiro a ser desenvolvido na elaboração de um sistema, pois com ele é possível abstrair as funcionalidades do mesmo (RUI, 2003). Além disso, ele possui alguns elementos fundamentais a serem definidos, como, por exemplo, os atores que representam um conjunto de papéis exercidos pelo usuário do sistema ao interagir com um determinado caso de uso, ou seja, o ator é o papel que um usuário desempenha em relação ao sistema. Outro desses elementos fundamentais é o caso de uso chamado de conjunto de seqüências, em que cada seqüência representará uma interação entre os atores e o sistema (OMG, 2003).

Segundo Cockburn (COCKBURN, 2001), os casos de uso são ações que descrevem o comportamento do sistema sob diversas condições conforme o sistema responde a uma requisição de um dos *stakeholders*, chamado de ator primário, o qual inicia uma interação com o sistema para atingir algum objetivo. Então, o sistema responde, protegendo os interesses de todos os *stakeholders*, e diferentes comportamentos (cenários) podem aparecer,

dependendo das requisições particulares feitas e das condições que cercam as requisições. Sendo assim, pode-se dizer que os casos de uso expressam uma função de detalhamento e de especificação dos requisitos.

Além disso, os casos de uso podem ter dois tipos de representações: a gráfica e a contextual; a primeira é definida por Booch (BOOCH, 2006) através da linguagem de modelagem UML, que provê a utilização de um conjunto gráfico de anotações que são largamente aceitas pela comunidade de desenvolvimento de *software,* e que fornece uma representação visual dos requisitos dos usuários; e a segunda, de acordo com Cockburn (COCKBURN, 2001), dá-se através de um *template*, em que o projetista de *software* preenche todos os passos de que um caso de uso necessita.

Um cenário é um componente de um caso de uso, e é definido como uma descrição de um caso de uso em uma seqüência de números limitados de eventos com uma ordem de tempo linear, sendo que um caso de uso pode gerar vários cenários. Assim, pode-se afirmar que cenários estão para casos de uso, assim como instâncias estão para classes, o que significa que cenário é basicamente instância de um caso de uso. Além disso, como um caso de uso descreve um conjunto de seqüências, cada seqüência determinada é chamada de cenário, e, ainda, um cenário pode ser descrito de diversas maneiras, desde uma maneira simples de descrição textual até a numeração de etapas, que indicam objetivos e ações a serem efetuadas. Em outras palavras, um cenário é basicamente uma instância de um caso de uso, logo, a UML divide cenários principais e cenários secundários; sendo que o primeiro são as seqüências principais, e o segundo aborda as seqüências alternativas (XU, J., 2004).

Por sua vez, Leite (LEITE, 1997) aborda uma outra definição para a descrição de um cenário através da redução de um caso de uso, dizendo que nesses cenários ocorre a descrição parcial do funcionamento do sistema que se concentra em um momento específico da aplicação. Sendo assim, os cenários não são formais, podendo representar uma variedade de recursos, ou seja, o cenário é uma descrição parcial do comportamento da aplicação, e cada descrição tem uma relação semântica com as outras. Portanto, para a realização e construção de cenários, é utilizado um vocabulário bem definido para realização das descrições através de uma “linguagem léxica estendida” (LEL).

2.1.3.4. *User Stories* e *CRC Cards*

Segundo Beck (BECK, 2000), a metodologia de desenvolvimento XP é um método leve, flexível, previsível, ágil e disciplinado para o desenvolvimento de *software,* que permite aos desenvolvedores produzir produtos de alta qualidade de uma forma dinâmica e rápida, consistindo em uma série de regras, ciclos e práticas que devem ser conhecidos pelos desenvolvedores. Além disso, essa metodologia é projetada para times de desenvolvimento entre três a dez engenheiros, tendo sempre um ou mais clientes interagindo no processo.

O processo de programação dessa metodologia é realizado diretamente por uma série de *User Stories*, que são desenvolvidas em reuniões com os usuários e o pessoal técnico, realizando, assim, o planejamento de cada interação. Portanto, as *User Stories* são os documentos de requisitos para o desenvolvimento do projeto, e sendo assim descritas na etapa do planejamento do projeto, têm por função colocar as funcionalidades mais valiosas em produção (REES, 2002).

É também importante salientar que as *User Stories* ocorrem na etapa do jogo do planejamento (definição das necessidades do *software*), em que esses cartões são preenchidos pelos jogadores ou clientes, ou seja, usuários específicos do sistema, sendo especificada a numeração das atividades e funcionalidades que o *software* deverá suportar; em outras palavras, o cliente escreve histórias de suas atividades (REES, 2002). Portanto, pode-se concluir que as *User Stories* e casos de uso apresentam o mesmo propósito, embora as *User Stories* não estejam limitadas a descrever uma interface com o usuário como os casos de uso. Além disso, a construção desse cartão depende principalmente do nível de habilidade do cliente com o sistema, assim, ele é escrito e baseado em uma linguagem natural, com um formato pré-determinado e em poucas linhas de texto.

Para Cohn (COHN, 2004), para se ter uma boa história, ela deve conter seis atributos fundamentais: ser independente, ser negociável, ser estimável, ser testável e dar valor aos usuários e clientes. Tendo esses seis atributos, pode-se dizer que uma história pode elicitar corretamente a necessidade dos clientes e usuários. Já uma história complexa é aquela que apresenta detalhes demais, e pode ser quebrada em várias histórias, expressando, assim, a necessidade do cliente de uma forma mais clara.

De acordo com Beck (BECK, 2000), cada time de desenvolvimento decide em conjunto quais informações serão relatadas em um cartão de história, mas se recomenda um

cartão-padrão com as seguintes informações: o número da *User Story*, seu título, a pessoa responsável, a data, a estimativa de tempo de implementação, o nível de risco, sua descrição, seu relacionamento, e a descrição dos testes. Sendo assim, conclui-se que todas essas informações são importantes, mas a informação de um cartão de história que requer mais cuidado é a descrição da *User Story*, pois através dela é possível saber se o requisito está explicado de maneira correta. Logo, pode-se citar esforço na área como Rees (REES, 2002) que desenvolve uma ferramenta *web* a qual aplica as *User Stories* nos requisitos, com o objetivo de padronizar uma formalização para todos os componentes de uma *User Story* através de uma linguagem de marcação.

Na Figura 4, é possível visualizar uma *User Story* que descreve a história de um jogador, sendo que, da esquerda para a direita, podem-se verificar os seguintes componentes de uma *User Story*: o número do cartão, a sua função, a estimativa de risco, e, por último, a escrita da sentença (BECK, 2000).

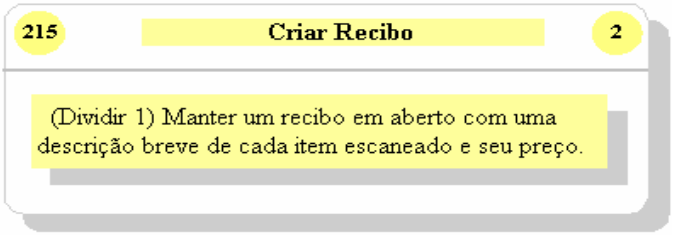


Figura 4: Exemplo de uma *User Story*

A segunda documentação refere-se aos *Use Class, Responsibilities, and Collaboration Cards* (*CRC Cards)*, que são uma forma de documentar o projeto através do time, e que demonstram um modo mais completo de apreciar a metodologia orientada a objetos. Essa documentação é também utilizada quando os *User Stories* estão explicados de uma maneira complexa, e permitem que as equipes do projeto contribuam com o *design*, pois assim é possível incorporar novas idéias construtivas ao projeto, sendo que o surgimento desse documento ocorre depois de analisadas as *User Stories*, quando essas são formalizadas por um grupo técnico de desenvolvimento de *software*. É importante ainda salientar que o XP utiliza os cartões *CRC Cards* para impor certa melhoria e disciplina a essa análise de requisitos (LEONARDI, 2002)**.**

2.1.4. Problemas em aberto na área de Engenharia de Requisitos

Segundo Nuseibeh (NUSEIBECH, 2000), a década de 90 trouxe grandes avanços no processo de Engenharia de Requisitos, ocorrendo o descobrimento de novas técnicas e ferramentas para auxiliar esse processo, uma vez que, durante esse período, identificaram-se novas idéias que desafiam a proposta do processo de Engenharia de Requisitos, como por exemplo:

- **Contexto e percepção social:** a modelagem e a análise de requisitos não podem ser informadas de uma maneira isolada e fixa sobre como o sistema deverá ocorrer. Portanto, para fugir dessa idéia, usam-se técnicas diferenciais, que proporcionam ao engenheiro ter uma análise mais ampla do cenário, como, por exemplo, a adoção de técnicas como a etnografia e a observação.

- **Modelagem do Ambiente:** a Engenharia de Requisitos não deveria se concentrar apenas na especificação da funcionalidade de um novo sistema, mas também na modelagem de propriedades indicativas e optativas que proporcionam uma descrição do ambiente e de que sistema se precisará nesse ambiente. Para isso, é necessário que o engenheiro não se preocupe apenas em identificar os requisitos, mas também que pense como o cenário irá ocorrer, para, assim, obter um escopo do projeto.

- **Inconsistências:** é complexo obter-se uma modelagem de requisitos completa e consistente. Portanto, dois problemas devem ser resolvidos: os requisitos conflitantes e a interatividade de comunicação do engenheiro com os *stakeholders.*

Por sua vez, Nuseibech (NUSEIBECH, 2000), impõe novos desafios que surgiram com o passar dos anos no processo de Engenharia de Requisitos, os quais são:

- **Modelagem e análise de propriedades do Ambiente:** desenvolveram-se novas técnicas para modelar e analisar o ambiente, e não mais o comportamento do *software*. Essas técnicas devem levar em conta a necessidade de solucionar problemas como modelos inconsistentes, incompletos e evoluídos. Assim, a expectativa é a de que ocorra um melhor suporte da Engenharia de Requisitos nessas áreas obscuras, como, por exemplo, a especificação das necessidades de recursos que um componente exige de seu ambiente.

- **Transição entre elicitação e análise:** as técnicas de elicitação contextuais e as técnicas formais para a especificação e análise são questões em aberto. Ou seja, as

abordagens contextuais como etnografias proporcionam um melhor entendimento do contexto organizacional para um novo sistema de *software*, mas não há correspondência direta com as técnicas de modelagem formal das propriedades atuais e desejadas do domínio do problema.

- **Análise e modelagem de requisitos não-funcionais:** esses são conhecidos como as restrições do sistema, em que se espera desenvolver modelos mais ricos para capturar e analisar os requisitos não-funcionais.

- **Visão arquitetural:** visa a um melhor entendimento dos impactos nas escolhas de arquitetura e evolução dos requisitos, e, para tanto, há uma concentração de esforços na área de arquitetura de *software* a fim de proporcionar uma melhor análise de seu comportamento.

- **Reuso de modelos de requisitos:** busca-se desenvolver modelos de referência para requisitos através de diversos domínios de aplicação, procurando reduzir o esforço para produção de modelos de requisitos, permitindo assim a redução do custo de vida de um projeto, e facilitando a seleção de *commercial off-the-shelf software* (COTS).

- **Treinamento da prática de requisitos:** o treinamento multidisciplinar para os profissionais de requisitos proporciona uma melhor prática e entendimento dos requisitos do sistema, pois são eles que aplicam as técnicas de requisitos para elicitar, analisar e especificar os requisitos como engenheiro de requisitos. Portanto, é de grande importância à empresa ter um profissional com essas qualidades, pois ele estará apto a interagir com os *stakeholders* e também terá habilidades técnicas para interagir com os analistas e programadores.

2.2. REFATORAÇÃO

A refatoração é considerada uma atividade de reengenharia, e consiste em um processo contínuo e controlado de reestruturação de código e dos dados de uma aplicação já existente, sendo que, para a realização desse processo, são estabelecidas regras que devem ser seguidas e adotadas de acordo com a situação com que o analista se deparar (FOWLER, 2004). De acordo com Kataoka (KATAOKA, 2001), através da refatoração é possível prover a evolução do *software* com a redução de custo e tempo. Além disso, a refatoração está presente em todo o desenvolvimento do *software*, e pode ser realizada quando necessário, ou se o

desenvolvedor da equipe perceber que é possível simplificar o módulo atual sem perder nenhuma funcionalidade.

O surgimento dessa técnica ocorreu na década de 80, através da linguagem de programação *smalltalk*, tendo sido proposta por duas teses de doutorado: primeiramente por Willian Opdyke, em *Refactoring of Object Oriented Frameworks* em 1992 (OPDYKE, 1992), e depois também por Don Roberts, em *Practical Analysis for Refactoring* em 1999 (ROBERTS, 1999).

Primeiramente, Opdyke (OPDYKE, 1992) define em sua tese de doutorado um conjunto de metodologias que reestruturem operações (refatorações) para suportar a evolução e reusabilidade para *frameworks* orientados a objetos. Assim, com essas refatorações é proposto resolver problemas como reusabilidade de componentes, prover a consistência entre os componentes e suportar a interação do projeto para um *framework* orientado a objetos. Ele define 23 refatorações primitivas e mostra três exemplos de refatorações compostas, sendo que para cada refatoração primitiva, um conjunto de pré-condições fornece a noção de preservação do comportamento do sistema; portanto, se cada refatoração primitiva preserva o comportamento, uma refatoração composta a partir dessas primitivas também o preservará. Além disso, as refatorações primitivas definidas por Opdyke foram classificadas de acordo com quatro categorias: criação de entidades (criação de uma classe vazia, uma variável ou função), remoção de entidades (remover classe, variável ou função não referenciada), alterar entidade (alterar nome da classe, alterar o nome da variável, adicionar parâmetro à função, etc) e mover variável (mover variável para superclasse e mover variável para subclasse); e as refatorações compostas são definidas para abstrair acesso a uma variável, converter segmento de código para função e mover classe.

Já Roberts (ROBERTS, 1999) propõe em sua tese de doutorado a criação de um arcabouço para linguagens orientadas a objetos, o que permite aos desenvolvedores criar as suas próprias ferramentas de refatoração. Sendo que, para comprovar a eficiência de sua pesquisa, ele desenvolve umas das primeiras ferramentas do tipo e propõe a utilização da técnica de refatoração chamada *Refactoring Browser*, baseada na linguagem de programação *smalltalk*, e cujo objetivo é o de aplicar as refatorações tornando mais fácil a vida do programador. Roberts propõe, também, estender o trabalho de Opdyke, aplicando pós-condições e pré-condições, as quais descrevem o que deve ou não haver depois da aplicação. Para aplicar a refatoração, tais condições devem ser analisadas para garantir a preservação do

comportamento. Assim, Roberts define o conceito de funções de análise, que descreve o relacionamento dos componentes do *software* como, por exempo, classes, métodos e variáveis. As funções de análise foram divididas em duas categorias: derivada e primitivas, a primeira é utilizada apenas nas pré-condições; e a última é utilizada tanto nas pré como nas pós-condições. Sendo assim, através desse conceito, ele redefine as refatorações propostas por Opdyke, dividindo-as em três grupos: refatorações de classe, refatorações de métodos, e refatorações de variáveis.

Por sua vez, Fowler (FOWLER, 2004) aumenta o nível de abstração da técnica de refatoração, centrando seu estudo de modo diferente dos autores anteriores, uma vez que se focou no processo de como realizar a refatoração, e define, assim, um catálogo de refatorações, explicando como ocorre o processo passo a passo.

Portanto, a refatoração tem se tornado uma técnica importante para as metodologias de desenvolvimento. Na metodologia XP, por exemplo, a principal idéia é o desenvolvedor trabalhar em um módulo por vez do sistema, e analisá-lo de uma forma coerente com a análise do *software*. No entanto, caso o módulo não esteja bem planejado, devem-se aplicar as refatorações do sistema até que atenda perfeitamente à análise de requisitos do *software*. Finalmente, é importante lembrar ainda que a refatoração está presente no ciclo de vida do processo XP (BECK, 2000).

2.2.1. Ciclo da Refatoração

A refatoração é representada através de um ciclo, objetivando mostrar como a refatoração do projeto é aplicada, mantendo a extensibilidade e durabilidade do mesmo. Portanto, para se ter um resultado eficaz com a aplicação da técnica de refatoração, é proposto um ciclo representado por duas fases: a de expansão (*expansion phase*), e a contraída (c*ontraction phase*) Na Figura 5, é mostrado como ocorre o ciclo de desenvolvimento através da refatoração, na qual se visualiza que seu grande problema são as duplicações de código (GORTS, 2004).

Primeiramente, a fase de expansão ou *"expansion phase"* é quando ocorre a melhoria estrutural de um projeto, o qual requer que sejam inseridos novos componentes em um *software*, sendo que esses componentes podem ser: variáveis, campos, métodos, classes ou até mesmo pacotes. Além disso, tais melhorias estruturais tendem a expandir o tamanho total da base do código.

Já a fase de contração ou *"contraction phase"* é aplicada na duplicação de código, o que é um erro comum de se corrigir com a refatoração, pois a manutenção do *software* fica muito onerosa, dificultando a evolução do mesmo. A segunda fase do ciclo de refatoração se concentra na eliminação do código duplicado, e tem como objetivo identificar novos níveis de abstração, resultando em uma melhoria do projeto, e proporcionando a redução do tamanho total da base do código.

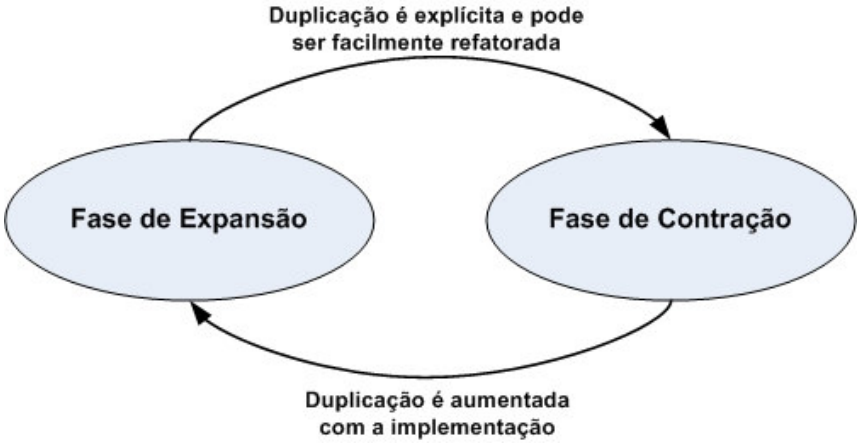


Figura 5: Ciclo da refatoração (GORTS, 2004)

2.2.2. Por que aplicar a Refatoração?

Embora a refatoração não seja a cura para todos os problemas do *software*, pode-se dizer que ela colabora para tal. Tornando-se uma ferramenta valiosa para manter o código seguro e consistente, portanto, a refatoração é um paradigma que pode e deve ser utilizado no desenvolvimento de um projeto. Além disso, a refatoração permite melhorar a qualidade interna do *software*, trazendo inúmeras vantagens na sua utilização (FOWLER, 2004), tais como:

- **Melhoria do Projeto de *Software*:** com o passar do tempo o *software* pode se deteriorar com as alterações executadas, pois diversos programadores podem alterar o código, possuindo assim diversas interpretações, o que ocasiona desestruturação do projeto. Geralmente, um sistema mal projetado necessita de mais código para fazer as mesmas coisas, porque o mesmo código aparece duplicado em diversos lugares. Assim para Ratzinger (RATZINGER, 2005), a utilização da refatoração proporciona uma melhoria do projeto evitando, que ele se deteriore.

- **Tornar o *Software* Legível:** é comum se ter um *software* que funciona perfeitamente, mas cujo processo não se consegue entender, uma vez que ele não está estruturado de uma forma compreensível. Portanto, a refatoração propõe tornar legível o relacionamento entre o programador e o código. Verifica-se isso na ferramenta "AIRES", que proporciona uma visualização mais legível do *software* através da identificação e detecção de códigos iguais (clones), pois, ocorrendo a localização desses códigos, ocorre a remoção e a refatoração dos códigos (HIGO, 2005). Um outro exemplo válido nesse contexto é a ferramenta de refatoração "KABA" (STRECKENBACH, 2004) cujo objetivo é refatorar classes, e que proporciona a refatoração desde o diagrama de classes, com seus devidos métodos, atributos e classes até a sua geração para o código fonte.

- **Encontrar Falhas:** a refatoração é um auxílio na identificação de falhas no *software*, uma vez que, quando se aplica a refatoração, é fácil encontrar erros, pois, ao mesmo tempo em que se está refatorando, também se está verificando a consistência do *software*, ocorrendo uma limpeza no código, e proporcionando uma maior clareza ao programador. Uma interessante forma de encontrar falhas é através da utilização dos testes, que são importantes ferramentas na identificação de possíveis erros, uma vez que um procedimento básico de testes consiste em testar rapidamente uma refatoração logo que ela seja realizada. Além disso, uma ferramenta importante para auxiliar nos testes é o *framework* orientado a objetos JUnit[1], que é aquele que possibilita a criação de testes unitários em Java[2], e também a criação das classes de testes, havendo um ou mais métodos para que tais testes sejam realizados.

- **Programar Rapidamente:** aplicando características da refatoração como reusabilidade e manutenibilidade, consegue-se um desenvolvimento de *software* rápido, pois a característica básica para um bom projeto é a sua manutenção no seu desenvolvimento. Assim, torna-se fácil identificar essa característica em metodologias ágeis como, por exemplo, o XP (JEFFRIES, 1999).

[1] http://www.junit.org/index.htm
[2] http://java.sun.com/

2.2.3. Etapas da Refatoração

Segundo Mens (MENS, 2004), os *software*s de hoje necessitam de ambientes consistentes e eficientes, que proporcionem flexibilidade e manutenibilidade, sendo que, para gerar essas características, essa técnica deve seguir um ciclo com seis etapas distintas:

1. **Identificar onde o *software* deverá ser refatorado:** a primeira decisão que necessita ser tomada é determinar o nível apropriado de abstração que será aplicado na refatoração, ou seja, ela pode ser aplicada nos componentes de *software* ou até mesmo nos requisitos. Um exemplo simples é a identificação do código duplicado em uma classe e, assim, é proposto aplicar a refatoração para eliminar esse código duplicado. Segundo Kataoka (KATAOKA, 2001), através da ferramenta *daikon*, é possível identificar onde a refatoração pode ser aplicada por uma detecção automática de variações do programa, as quais ocorrem nos parâmetros dos métodos já existentes, em funções com vários parâmetros.

2. **Determinar quais refatorações devem ser aplicadas nos lugares identificados:** a refatoração é identificada através de inconsistências, e, quando são localizadas, devemos ver qual refatoração é mais apropriada, e logo aplicá-la. Assim, para cada problema que for identificado há uma solução que dever ser seguida. Portanto, os autores Kent Beck, John Brant, Willian Opdyke, Don Roberts e Martin Flower determinam uma série de regras, que podem ser aplicadas em diversos casos como é mostrado nas pesquisas de Opdyke (OPDYKE, 1992), Roberts (ROBERTS, 1999) e Fowler (FOWLER, 2004).

3. **Garantia da preservação do comportamento:** depois de determinar onde e quais refatorações aplicar, o próximo passo é garantir que elas garantirão o comportamento do sistema. Assim, a definição de preservação do comportamento indica que para os mesmos valores de entrada, os valores de saída devem ser iguais antes e depois da refatoração, ou seja, a refatoração não pode proporcionar inconsistência no resultado final. A idéia de preservação do comportamento surgiu através de Roberts (ROBERTS, 1999), que diz que para um conjunto de dados de entrada, o conjunto de dados de saída deve ser idêntico, antes e depois da aplicação da refatoração, pois, assim, ele aperfeiçoa o conceito de pré-condições. Em outras palavras, tal conceito pode ser explicado como um conjunto de verificações que devem ser verdadeiras para

que a refatoração possa ser aplicada, sendo que cada refatoração tem um conjunto de pré-condições.

4. **Aplicar a refatoração:** a aplicação da refatoração se dá através da identificação do problema e da alteração do código fonte. Geralmente a aplicação da refatorações pode ocorrer através de ferramentas como, por exemplo, Eclipse[3] e o NetBeans[4], essas IDEs sendo que essas IDEs já possuem algumas refatorações que podem ser aplicadas automaticamente pelo desenvolvedor.

5. **Verificação da preservação do comportamento:** o comportamento de uma refatoração pode ser analisado através de características como extensibilidade, reusabilidade, desempenho e manutenção, as quais devem garantir qualidade ao *software*, sendo esses os atributos de qualidade do que está sendo refatorado. Logo, para garantir que o comportamento foi realmente preservado, é necessário saber se todas as operações da refatoração foram realizadas com sucesso. Sendo assim, aplica-se um conjunto de pós-condições que devem ser satisfeitas após a aplicação de uma refatoração (ROBERTS, 1999).

6. **Manter a consistência entre a refatoração do código e de outros componentes do *software* (documentação, especificações):** o desenvolvimento de *software* envolve uma série de componentes como especificações, arquiteturas de *software*, modelos de projeto, documentação, testes e outros; sendo que, para ocorrer a refatoração desses componentes, necessita-se manter a consistência entre eles, ou seja, é preciso mantê-los atualizados de acordo com as modificações ocorridas.

2.2.4. Problemas encontrados nos códigos

Como já foi visto em seções anteriores, a refatoração é uma técnica complicada de se aplicar, pois requer um bom conhecimento do desenvolvedor. Por isso, Fowler (FOWLER, 2004) define os *"bad smells"*, ou "maus cheiros", encontrados nos códigos, os quais são problemas que causam redundância no código, dificuldade no entendimento e, principalmente, a ineficiência do *software*.

Sendo assim, podem-se apontar alguns dos principais problemas encontrados nos códigos: código duplicado, método longo, classes longas, listas de parâmetros longos,

[3] http://www.eclipse.org/
[4] http://www.netbeans.org/

alterações divergentes, classe ociosa, campos temporários, herança recusada e outros. Segundo Ratzinger (RATZINGER, 2005), um exemplo disso é uma ferramenta que aplica refatorações que tratam do problema de código duplicado e métodos longos, e cujo objetivo é a redução do código e dos métodos do *software*. Outro exemplo é a ferramenta de refatoração que aplica a técnica em classes, a qual é demonstrada através de diagramas de classes para ocorrer uma visualização mais clara. Além disso, ela é proposta com o objetivo de solucionar o problema de classes longas, classes ociosas e até o problema de herança (STRECKENBACH, 2004).

Conforme já foi citado, Opdyke (OPDYKE, 1992) foi o primeiro autor a lançar um conjunto de refatorações, e, depois disso, foi proposto por Roberts (ROBERTS, 1999) abstrair e evoluir esse conjunto de refatorações. Assim Fowler (FOWLER, 2004) propõe um catálogo de refatorações mais refinado, explicando passo a passo como realizar tal refatoração. Resumidamente, o conteúdo desse catálogo, que é de 72 refatorações divididas em sete grupos, pode ser descrito da seguinte forma: um nome para a refatoração, um resumo da situação ao qual necessita da refatoração, a motivação para se aplicar a refatoração, a mecânica de como ocorre, e por último, exemplos ricamente comentados,

A seguir são descritos dois tipos de refatoração de código que integram o catálogo de refatoração de Fowler (FOWLER, 2004):

- Extrair Método (*Extract Method*): é quando há um fragmento de código que pode ser agrupado. Para tanto, a ação deve transformar um fragmento em um método cujo nome explique o propósito do mesmo. A Figura 6 e Figura 7 mostram um exemplo não refatorado e um exemplo refatorado desse caso de refatoração, e a seguir são descritas as etapas para o seu funcionamento:

    1. Criar um novo método e escolher um nome que explicite a sua intenção;

    2. Copiar o código extraído do método de origem para o novo método;

    3. Procurar por variáveis locais e parâmetros utilizados pelo código extraído:

        - Se variáveis locais forem usadas apenas pelo código extraído, passá-las para o novo método;

- Caso contrário, analisar se o seu valor é apenas atualizado pelo código. Nesse caso substitua o código por uma atribuição;

- Se for tanto lida quanto atualizada, passá-la como parâmetro;

4. Compilar e testar;

```
Exemplo não refatorado
void imprimirFilme(double quantidade) {
imprimirLista();
System.out.println("Nome do Filme: " + nomefilme);
System.out.println("Quantidade: " + quantidade);
}
```

Figura 6: Extrair método: não refatorado

```
Exemplo refatorado
void imprimirFilme(double quantidade) {
imprimirLista();
imprimirDetalhes(quantidade)
}

private void imprimirDetalhes(double quantidade) {
System.out.println("Nome do Filme: " + nomefilme);
System.out.println("Quantidade: " + quantidade);
}
```

Figura 7: Extrair método: refatorado

- Auto-Encapsular Campo (*Self Encapsulate Field*): ocorre quando se está acessando um campo diretamente, mas o acesso a tal campo está se tornando inadequado. Essa refatoração proporciona dois tipos de acesso: um dentro da classe em que a variável está definida, que se caracteriza por acesso direto às variáveis; e outro através de métodos, caracterizado por acesso indireto à variável. A Figura 8 e Figura 9 exemplificam este caso de refatoração e a seguir são decritas as etapas deste:

1. Criar métodos de leitura e gravação para o campo;

2. Encontrar todas as referências ao campo e substituí-las por um método de leitura ou de gravação;

3. Verificar novamente, se foram descobertas todas as referências;

4. Compilar e testar;

```
Exemplo não refatorado
private int baixo, alto;
boolean incluir (int arg) {
return arg >= baixo&& arg <= alto;
}
```

Figura 8: Auto-encapsular campo: não refatorado

```
Exemplo refatorado
private int baixo, alto;
boolean incluir (int arg) {
return arg >= getBaixo() && arg <= getAlto();
}
int getBaixo() {
return baixo;
}
int getAlto() {
return alto;
}
```

Figura 9: Auto-encapsular campo: refatorado

Portanto, através dos exemplos apresentados, é possível visualizar que o código ficou mais legível e melhor organizado, apresentando uma "cara mais limpa". Além disso, com todos esses estudos na área da refatoração, pode-se concluir que é possível obter um *software* consistente com a sua aplicação, embora se deva observar para que ela seja aplicada com tempo e por pessoas experientes nessa técnica.

2.2.5. Áreas de aplicação da Refatoração

Segundo Mens (MENS, 2004), com o passar do tempo a refatoração passou por importantes avanços, conseguindo-se propor a refatoração através de algumas técnicas conhecidas. Por essa razão, hoje em dia é possível aplicar a refatoração em diversas etapas do desenvolvimento, como, por exemplo, em modelos de projeto, em esquemas de banco de dados, na Engenharia de Requisitos, nas arquiteturas de *software* e nos componentes de *software*. Sendo assim, essa diversidade de estruturas permite ao desenvolvedor e analista não ficar apenas na refatoração de código, como também mostrar o outro lado, que é a parte de análise do sistema, além disso, é importante salientar que esses artefatos devem estar em constante sincronia. A seguir, serão descritas algumas áreas em que a refatoração está sendo utilizada.

2.2.5.1. Programação

A origem da técnica da refatoração está vinculada diretamente ao código fonte, pois é nessa linha de pensamento que ocorre a maioria dos trabalhos e pesquisas, visando ao desenvolvimento de ferramentas que oferecem apoio e automatização no processo de refatorações diretamente no código fonte da aplicação. Assim, a refatoração pode ser aplicada em diversas situações e partes do código, e sua aplicação pode ser feita através de diversas linguagens de programação e paradigmas de programação, como, por exemplo, de algumas linguagens orientadas a objetos (Java, C#[5]), inguagem funcional, ou orientada a aspectos (AspectJ [6]). No entanto, a refatoração em linguagens não orientadas a objetos é mais complexa de se aplicar no projeto, devido ao forte acoplamento presente nesse paradigma, pois se limita ao nível de função ou de bloco de códigos (WEISER, 1981; LI, 2003)**.**

Por sua vez, as aplicações da refatoração em linguagens orientadas a objetos em alguns casos também são complexas, uma vez que requerem um grande conhecimento e prática por parte do desenvolvedor e porque algumas refatorações são complicadas de se aplicar, como no caso das refatorações que lidam com herança, polimorfismo e interfaces (FOWLER, 2004). Contudo, devido a inúmeros esforços na área da refatoração, a automatização dessa técnica já é uma realidade, e um grande exemplo disso é o fato de ela já estar presente na IDE Eclipse, que é uma das aplicações mais utilizadas hoje no desenvolvimento de projetos de *software*. Além do mais, essa automatização proporciona ganho de tempo, eficiência e eficácia para os desenvolvedores.

Para finalizar, é importante dizer que esforços têm sido feitos na área de refatoração, como, por exemplo, a Refax, que é uma ferramenta que proporciona a ligação entre XML e refatoração, sendo um *framework* para refatoração baseado em XML. Além disso, essa ferramenta utiliza tecnologias do tipo XML como mecanismo e apresentação do código, com o objetivo de auxiliar na construção de ferramentas de refatorações, proporcionando, assim, extensibilidade e reusabilidade (MENDONÇA, 2004).

2.2.5.2. Casos de Usos e Cenários

A refatoração de casos de uso e cenários atinge diretamente os requisitos do projeto, uma vez que esse tipo de refatoração tem a função de promover o melhoramento e aperfeiçoamento de um ou mais conjuntos de requisitos de um *software*. Para isso, Rui (RUI, 2003) propõe a

[5] http://msdn.microsoft.com/vcsharp/
[6] http://www2.parc.com/csl/projects/aop/

refatoração de casos de uso e cenários determinado através de um meta-modelo de caso de uso para a aplicação das refatorações. Esse processo é divido em duas etapas, sendo que a primeira ocorre através da determinação de um meta-modelo para aplicar as refatorações de caso de uso; além disso, esse meta-modelo é definido por Regnell (REGNELL, 1999) e aplicado o conceito de refatoração em cascata definido por Xu (XU, L., 2005). Por sua vez, a segunda etapa é a determinação de que tipo de refatoração aplicar, e esse meta-modelo é acrescido de um conjunto de tarefas que descreve o papel que o usuário desempenhará no *software*, e de um conjunto de relacionamentos que especificam o relacionamento das ações do sistema.

Além disso, são determinados cinco níveis de aplicação da refatoração nos casos de uso e cenários: criando entidades de caso de uso, deletando entidade de casos de uso, alterando a entidade de casos de uso, movendo um elemento de caso de uso, e distribuindo o comportamento; sendo que para cada tipo é determinado um conjunto de refatorações (REGNELL, 1999). Convêm salientar que alguns modelos serão detalhados e especificados no capítulo 4, pois o foco dessa dissertação é a refatoração de requisitos.

2.2.5.3. UML

A refatoração é uma técnica que está em constante crescimento, podendo ser aplicada na área de artefatos de projeto de *software*, e um desses artefatos que é muito utilizado para a especificação de um projeto de *software* é a linguagem de modelagem UML (BOOCH, 2006), que tem como objetivo realizar as especificações dos requisitos através de uma linguagem de modelagem que seja capaz de ser interpretada pelos desenvolvedores de *software*. Segundo Sunyé (SUNYÉ, 2001), a evolução de um projeto ocorre através das modificações e de se tornarem certos elementos mais extensivos, permitindo a adição de novas características e componentes, proporcionando, assim, a refatoração no artefato de *software*.

Como já visto anteriormente, a refatoração possui duas etapas fundamentais de acordo com Opdyke (OPDYKE, 1992) e Roberts (ROBERTS, 1999): a determinação das pré-condições e das pós-condições para o correto funcionamento das refatorações. No modelo de refatoração para a UML, as pré-condições e as pós-condições são determinadas em nível de OCL – *Object Constraint Language* (OMG, 2003), que é uma linguagem de expressões para especificar restrições sobre modelos orientados a objetos ou a outros artefatos da linguagem UML. Portanto, a OCL surge para realizar uma melhor especificação dos dados em relação à linguagem de modelagem UML, sendo que as pré-condições em expressões OCL representam

as condições necessárias para que as operações sobre os objetos possam ser executadas, e as pós-condições representam o estado do sistema após ocorrer determinada operação. Portanto, muitas pesquisas como as de Sunyé (SUNYÉ, 2001) e Massoni (MASSONI, 2005) na área de refatoração de diagramas de UML adotam a OCL, para prover uma semântica ao projeto.

Segundo Sunyé (SUNYÉ, 2001), as refatorações ocorrem sobre dois tipos de diagramas: o de classes e o de estado. No primeiro, são realizadas refatorações baseadas em cinco operações básicas: adicionar, remover, mover, generalizar e especializar na modelagem dos elementos. A adição de componentes ocorre quando uma classe tem uma nova característica e assinaturas diferentes; já a remoção de um componente ocorre quando esses elementos não possuem referência ao modelo associado; e, por sua vez, as operações do tipo generalizações são aplicadas em elementos da própria classe, como atributos, métodos e operações, e consistem na integração de um ou mais elementos, que são transferidos para a superclasse. Sendo assim, as refatorações do tipo especialização são exatamente o oposto das de generalização, pois consistem no envio de um elemento para todas as subclasses possuídas. Além disso, na refatoração de diagrama de estado é possível melhorar a interação do contexto para refatoração, e para as refatorações baseadas nesse tipo de diagrama é utilizada a OCL que mostra e verifica como ocorrem as transformações antes e depois de serem realizadas as refatorações. Conseqüentemente, assim são determinados dois tipos de refatorações: as de estados e as de estados compostos, sendo que a primeira permite substituir um conjunto de ações para conectar as transações de um estado, e a segunda propõe substituir um conjunto de transições ou uma transição por um estado composto. Na Figura 10, é mostrado um exemplo com o objetivo de exemplificar como ocorre uma refatoração de diagrama de estados por meio de declarações OCL através da determinação das pré-condições e das pós-condições (SUNYÉ, 2001).

```
State: unfoldExit
pre:
self.exit→notEmpty()and self.outgoing.effect→isEmpty();
self.outgoing.target→forAll(s:State|s.container=selfcontainer);
post:
self.exit→isEmpty() and
self.outgoing.effect→forAll(a:Action| a.isEquivalent(self.exit))
```

Figura 10: Refatoração em diagramas de estados

2.3. INTEGRAÇÃO DE INFORMAÇÕES E SISTEMAS

O desenvolvimento da tecnologia tem ocorrido em um ritmo acelerado desde o surgimento e disseminação dos computadores nas grandes empresas, seguida pela popularização dos computadores pessoais, até o crescimento da *Internet*. Em poucos anos, empresas passaram a manter sistemas de grande porte, utilizando tecnologias defasadas, e, ao mesmo tempo, mantendo sistemas tecnologicamente atualizados, o que acarretou em dificuldades na comunicação entre esses sistemas por serem altamente distribuídos e heterogêneos, englobando assim diferentes funcionalidades e tecnologias (FONSECA, 2000).

Dessa maneira, a integração de sistemas tem sido um assunto bastante questionado, pois sistemas não integrados ocasionam problemas como redundância de dados, perda de semântica na tradução dos dados e perda de informações no momento de transferência de dados de um sistema para outro. No entanto, existem diversas maneiras de prover a integração entre os sistemas, por exemplo: no contexto de cliente-servidor, são utilizadas tecnologias do tipo COM+, J2EE e CORBA, já a troca de mensagens entre sistemas ocorre através da execução de procedimentos que retornam valores. Outra forma de integração é pela *web*, muito difundida, que é a utilização dos w*eb services* através da utilização do protocolo SOAP, que estabelece a comunicação entre os sistemas, assim como também de padronizações dos dados através de arquivos no formato XML (PARREIRAS, 2004).

Além disso, a construção de ontologias tem sido proposta como uma abordagem promissora para possibilitar a interoperabilidade entre sistemas e modelos de dados. Segundo Lustosa (LUSTOSA, 2004), as ontologias proporcionam o compartilhamento, a interoperabilidade, e a descoberta do conhecimento entre domínios, estruturando o domínio de forma que se permita sua compreensão com maior clareza e objetividade, permitindo assim a reutilização de conceitos em domínios.

2.4. ONTOLOGIA

A palavra ontologia provém da filosofia, e, por ser uma disciplina que estuda a natureza do ser e da existência, lida diretamente com os seres, as coisas enquanto seres, e os objetos enquanto coisas; assim, os seres e as coisas são denominados genericamente como entidades. Gruber foi quem introduziu o termo na área da inteligência artificial, motivado por razões como o compartilhamento e o reuso do conhecimento, tendo a idéia de especificação explícita de uma conceitualização (GRUBER, 1993). Essa conceitualização refere-se ao conjunto de

conceitos, relações, objetos e restrições que são definidos para um modelo semântico de algum domínio de interesse. Portanto, as ontologias expressam o formalismo dos conceitos e das relações acerca de um domínio (LUSTOSA, 2004).

2.4.1. Por que aplicar uma ontologia?

A aplicação das ontologias tem se tornado cada vez mais constante na área da Informática, portanto, visando a sua utilização na *web*, o W3C padroniza linguagens para a *web* e codifica o conhecimento, ligando os diversos tipos de tecnologias e semânticas existentes (PACHECO, 2001).

Destacam-se, assim algumas razões da aplicação das ontologias:

- Oportunidade para os desenvolvedores reusarem ontologias e bases do conhecimento, mesmo com adaptações e extensões;
- Compartilhamento e ligação entre estrutura de informações;
- Possibilidade de tradução em diversas linguagens e formalismos de representação do conhecimento;
- Desenvolvimento do domínio de conhecimento de uma forma clara e objetiva para que se possa compreender seu conteúdo.

2.4.2. Descrição de uma ontologia

Segundo Noy (NOY, 2004), uma ontologia formaliza o conhecimento através da utilização de cinco componentes:

1. Conceitos que são a representação de algo, ou de qualquer coisa, acerca do domínio em questão;
2. Relacionamentos que são as integrações entre os conceitos do domínio. Nessas relações ocorre a definição das cardinalidades;
3. Propriedades das classes e valores permitidos;
4. Axiomas que representam as sentenças que irão restringir a interpretação dos conceitos e relações;

5. Instâncias que representam os elementos de uma ontologia, ou seja, são as representações dos conceitos e relações que foram estabelecidos pela ontologia.

Além disso, uma ontologia é composta pelos seguintes itens: classes, *slots* e restrições. O primeiro tem a responsabilidade de descrever os conceitos de um domínio; o segundo descreve as propriedades e suas instâncias; e, por último, as restrições definem as propriedades específicas de um *slot* (NOY, 2004).

As integrações entre os processos de uma ontologia são representadas por uma árvore, em que cada nó representa uma ontologia, e cada relacionamento representa um relacionamento taxonômico. As taxonomias são as maneiras como se organizam os componentes de uma ontologia, ou seja, classes e subclasses, e, portanto, organiza-se o conhecimento usando relações de generalização/especialização através de heranças (NOVELLO, 2005).

2.4.3. Representação de ontologias

Segundo Lustosa (LUSTOSA, 2004), as linguagens propostas para a representação de ontologias utilizam valores lógicos, restrições e regras para a sua formalização. Além disso, a ontologia visa a desenvolver um conjunto de regras que possibilitem a abstração do significado semântico das informações disponibilizadas. Hoje, há inúmeros recursos específicos que possibilitam a descrição e criação de uma ontologia, que são:

- **Ontolingua[7]:** desenvolvida pelo KSL do departamento de ciência da computação da Universidade de *Stanford*, essa ferramenta é tanto uma linguagem quanto um ambiente para a edição colaborativa de ontologias. Esse ambiente reside em um servidor *web* e deve ser acessado com um navegador padrão. Além disso, a ontolíngua é baseada em um formato chamado KIF, e possui tradutores que permitem exportar as ontologias desenvolvidas para outros ambientes. Essa ferramenta disponibiliza ainda uma interface de acesso através de navegadores, possibilitando a qualquer usuário da *web* a visualização das ontologias criadas e disponíveis, possibilitando a criação de novas ontologias e do trabalho cooperativo.

[7] http://www.ksl.stanford.edu/*software*/ontolingua/

- **Protégé 2000[8]:** desenvolvido pelo KMG da Faculdade de Medicina de *Stanford*, é uma ferramenta integrada usada para o desenvolvimento de sistemas baseados em conhecimento, e possui uma interface gráfica que torna fácil a sua utilização. A versão atual é totalmente desenvolvida em Java, proporcionando portabilidade para qualquer ambiente, e inclui uma API para estender o ambiente para suportar aplicações em domínios específicos.

- **OntoEdit[9]:** é uma ferramenta de modelagem de ontologia em um nível conceitual, que possibilita a independência de linguagens concretas de representação e suporta RDFs[10] e OIL[11], além de possibilitar a exportação das ontologias criadas para outros formatos como, por exemplo, DTDs. As funções básicas do *software* e a interface gráfica são bastante semelhantes às do Protégé-2000, mas o OntoEdit possui um maior número de recursos, como a definição de axiomas, disjunções e meta-informações sobre a ontologia criada.

2.4.4. Como desenvolver uma ontologia?

Noy (NOY, 2004) define que uma ontologia consiste na representação do conhecimento, sendo que, para se conseguir uma representação eficiente e eficaz, é necessário seguir os seguintes passos:

1. **Determinar o domínio e o escopo da ontologia:** essa etapa consiste em definir elementos como: qual domínio essa ontologia irá cobrir? Para que será usada essa ontologia? Quem usará essa ontologia? Sendo que tais questões deverão ser respondidas durante o processo de desenvolvimento da ontologia.

2. **Considerar a reusabilidade de ontologias existentes:** reusar uma ontologia pode ser um requisito caso o sistema necessite interagir com outras aplicações que já possuam ontologia.

3. **Enumerar importantes termos na ontologia:** ocorre através da criação de uma lista de termos sem se preocupar com a sobreposição de conceitos, relação entre seus termos, ou propriedades que eles podem ter. Torna-se necessário, assim,

[8] http://protege.stanford.edu/
[9] http://www.ontoknowledge.org/tools/ontoedit.shtml
[10] http://www.w3.org/RDF/
[11] http://www.ontoknowledge.org/oil/

desenvolver uma hierarquia de classes e definir as propriedades dos conceitos (*slots*).

4. **Definir as classes e hierarquia das classes:** há dois tipos de processos de desenvolvimento: o *"top-down"* e o *"botton-up"*. O primeiro inicia-se com os conceitos mais importantes no domínio, e o segundo define as classes mais específicas. Por último ocorre o processo de combinação que é a junção e o desenvolvimento dos processos.

5. **Definir as propriedades das classes (*slots*):** as classes sozinhas não definem nenhuma informação, portanto, é necessário definir a estrutura interna dos conceitos.

6. **Definir as restrições dos *slots*:** um *slot* aplica diferentes restrições, descrevendo o tipo e a cardinalidade dos valores, entre outros.

7. **Criar Instâncias:** resume-se à criação de instâncias individuais de classes na hierarquia, portanto, uma instância individual requer: escolher a classe, definir uma instância individual dessa classe, e definir os tipos e os valores para os *slots*.

3. TECNOLOGIAS UTILIZADAS PARA IMPLEMENTAR A FERRAMENTA

Este capítulo tem por objetivo apresentar uma revisão bibliográfica sobre as tecnologias que foram nessa dissertação.

3.1. XML – *eXtensible Markup Language*

A XML é uma linguagem de marcação que possui raízes em SGML (*Standard Generalized Markup Language*) (HOLZNER, 2001). Usando a XML, pode-se definir qualquer número de elementos (*tags*), representados entre os sinais de "<" e "/>", que associam significado às informações. Ou seja, é possível descrever as informações e o que fazer com elas, utilizando para isso um ou mais elementos que possuem essa finalidade. A Figura 11 mostra a descrição de um arquivo XML.

```
            Exemplo simples de arquivo XML
    <Aluno>
            <nome> Tiago da Silva Minuzzi </nome>
            <curso> Ciência da Computação </curso>
            <Turno> Noite </Turno>
    </Aluno>
```

Figura 11: Exemplo simples de um arquivo XML

Como se pode observar, a Figura 11 mostra um arquivo XML que contem informações de um aluno. Portanto, conclui-se que esse exemplo não mostra apenas elementos que demonstram um dado, mas sim uma pequena estrutura que relaciona um grupo de dados, o qual pode conter diversas informações. No exemplo acima, para se iniciar uma *tag* define-se <Aluno> e para finalizar define-se </Aluno>, visualizando um agrupamento de dados, com os quais fica fácil de imaginar a realização de uma pesquisa sobre certos critérios em um sistema através da utilização do XML.

Sendo assim, é possível utilizar o XML na etapa de troca de dados, uma vez que ele permite que aplicações diferentes consigam entender e interpretar os dados da mesma forma, ou seja, se as duas partes de comunicação de dados compartilharem o mesmo padrão de esquema do arquivo XML, pode-se dizer que as aplicações conseguiram compartilhar os elementos (informações) entre as *tags*. Como exemplo disso, há a comunicação entre diferentes sistemas e os *web services* (COYLE, 2002). O foco da aplicação de XML e das

outras tecnologias nesse trabalho é padronizar um formato de representação de dados, no qual duas aplicações consigam se comunicar, garantindo a interoperabilidade entre dois sistemas. Segundo Holzner (HOLZNER, 2001), a utilização desse recurso tem sido adotada por diversas empresas e organizações nas comunicações de seus dados, pois proporciona diversas vantagens como: ter extensibilidade, ser autodescritivos, proporcionar a criação de estruturas complexas, ter flexibilidade, ter base de dados e possuir um padrão aberto.

3.1.1. Estruturando com esquemas

Quando se quer usar dados aplicando sua extensão a aplicações, clientes e fornecedores, é necessário definir o documento XML para prover essa estrutura de uma forma consistente, e, para isso se torna necessário ter um esquema.

Um esquema é um termo geral que descreve a forma dos dados, ou seja, define-se como um esquema a especificação formal utilizada para especificar um XML. Além disso, os esquemas são utilizados para validar o documento XML, determinando quando esses documentos estão de acordo com a gramática expressa pelo esquema, descrevendo a estrutura de um XML, e proporcionando a troca de informações entre as aplicações (COYLE, 2002).

No mundo do XML, de acordo com Coyle (COYLE, 2002), são proporcionados dois tipos principais de esquemas: DTD e *Schemas*. Na Figura 12, é ilustrada a definição da família de instâncias de documentos XML.

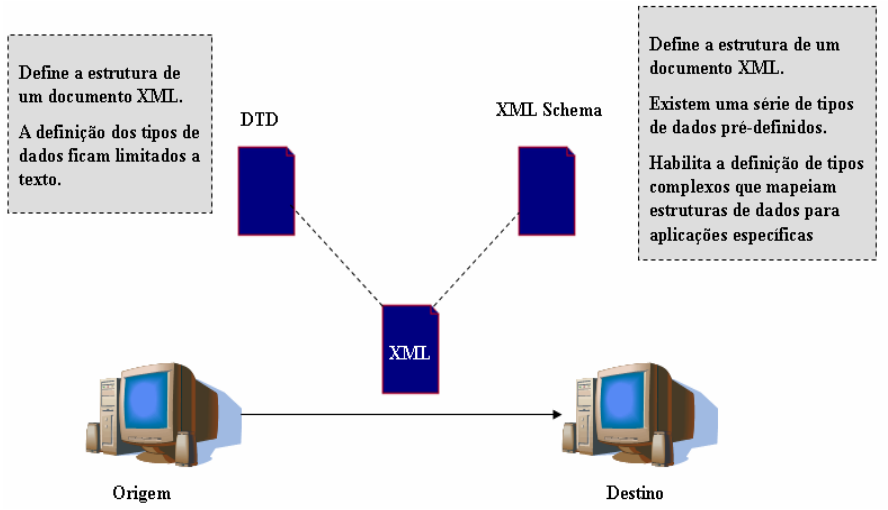


Figura 12: Representação de uma família XML (COYLE, 2002)

DTD provê uma simples estrutura de dados, e seu foco é a estrutura, permitindo a criação de um vocabulário que especifica seus elementos e os atributos apropriados para um conjunto de instância de documentos XML, e descrevendo, assim, tipos associados de um documento XML. Um exemplo de aplicação simples de um DTD é a realização de uma consulta. As três principais características de um esquema DTD é a utilização de uma sintaxe diferente da XML, a definição dos elementos e dos atributos que pode validar o documento, e, por último, a impossibilidade de validar a distinção entre diferentes tipos de dados (KLEIN, 2001).

Já o XML *Schema* é uma proposta sucessora do DTD, sendo recomendada pela W3C, por ser extremamente estável, uma vez que proporciona um maior nível de detalhamento e gerenciamento sobre os tipos de dados que aparecem em um documento XML. O XML *Schema* define um vocabulário e regras que proporcionam o gerenciamento do conteúdo dos elementos de um documento XML. A vantagem da adoção desse tipo de XML é que proporciona uma melhora na gramática para descrever a estrutura de elementos e uma faixa enorme de dados para a criação de tipos de dados complexos (KLEIN, 2001). Além do mais, o XML *Schema* permite tratar dados complexos, assim, para ocorrer a validação, o programador pode utilizar outros recursos através da utilização dos *parsers* (DOM e SAX), que são APIs de acesso a documentos XML e permitem que aplicativos consigam ler documentos XML sem se preocupar com a sintaxe deles.

3.2. RDF – *Resource Description Framework*

Essa tecnologia foi recomendada pela W3C, em 1999, com o objetivo de criar um modelo de dados simples, utilizando uma semântica formal, através da sintaxe do XML. Define-se como uma aplicação XML que serve como uma base para o processamento de meta-dados, e seu principal objetivo é definir uma arquitetura genérica de meta-dados que permite descrever semanticamente recursos no contexto *web*. Além disso, o RDF proporciona a interoperabilidade entre as aplicações através dos mecanismos que suportam convenções semânticas, sintaxe e estrutura. Esse modelo de dados abstrato define o relacionamento de recursos através de triplas, sendo que uma tripla é formada por três componentes: recurso, propriedades e valor da propriedade. O modelo de dados de um RDF é caracterizado por ter três tipos de objetos (BECKETT, 2004):

- Recurso (*Resource*): é qualquer objeto que é identificável por uma URI, que pode ser uma página *Web*.

- Propriedades (*Property*): é um aspecto específico ou característica, um atributo que descreve um recurso.

- Valor (*Literal*): é o valor que o objeto tem de acordo com a propriedade.

Logo, essas três partes da declaração de uma RDF são denominadas de sujeito, predicado e valor da propriedade, e originam a "sentença" (s*tatements*), que é uma informação estruturada composta dos três tipos de objetos de uma RDF (BECKETT, 2004). A Figura 13 apresenta uma declaração gráfica RDF que significa: uma página *web* http://www.unisinos.br tem curso de mestrado em computação, identifica-se que o recurso é a página *web* htpp://www.unisinos.br, a propriedade é "tem curso" e tem o valor "mestrado em computação". Na Tabela 1 demonstra-se em detalhes como fica a declaração RDF no formato de uma tripla.

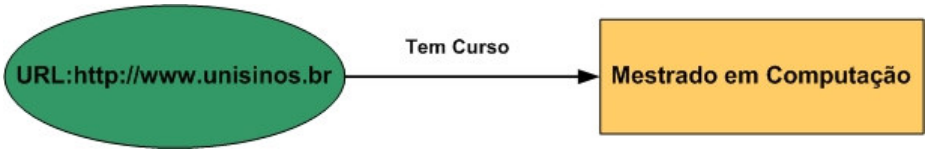


Figura 13: Declaração visualizada em uma figura (BECKETT, 2004)

Tabela 1: Demonstração de uma tripla no formato RDF (BECKETT, 2004)

Objeto	Atributo	Valor
http://www.inf.unisinos.br	Tem Curso	Mestrado em Computação

Os RDF *Schemas* são tipos de RDFs que definem as propriedades válidas para dar uma descrição RDF, que aplica certas características ou restrições a um conjunto de dados (MILLER, 1998). Para modelar novos RDF *Schemas,* são utilizadas as primitivas como*:* classe, subclasse, propriedade, sub-propriedade, instância e restrição. Segundo Beckett (BECKETT, 2004) comparando-se com o modelo básico de RDF, o RDF *Schema* proporciona uma maior interoperabilidade, semântica e extensibilidade dos dados das aplicações em relação ao RDF comum, pois, através dos *Schemas*, é possível representar os dados de uma forma mais consistente e detalhada. Esse tipo de RDF pode ser determinado como uma linguagem genérica para definição de sistemas de tipos, mas, para que se possa

descrever o domínio específico (ontologia), é necessário um poder maior de expressividade, para isso existe a OWL, que será vista na próxima subseção.

3.3. OWL – *Web Ontology Language*

A OWL é uma linguagem recomendada pela W3C para a definição e instanciação de ontologias na *web*, que suporta relacionamentos e restrições de interpretação por meio de axiomas, garantindo a interoperabilidade dos documentos *web* (MCGUINNES, 2004). Segundo Lustosa (LUSTOSA, 2004), é uma linguagem de ontologias para *Web* desenvolvida para a utilização do processamento do conteúdo de informações, em vez de somente apresentá-las aos usuários; portanto, ela é utilizada quando informações contidas em documentos precisam ser processadas por aplicações. Essa linguagem é definida pelas descrições das classes, propriedades e suas instâncias.

A OWL se origina de outras duas linguagens, a OIL e a DAML. Na Figura 14, pode-se pode-se visualizar a evolução das linguagens que representam ontologias, sendo que a linguagem OWL possui características do RDF (GONÇALVES, 2004).

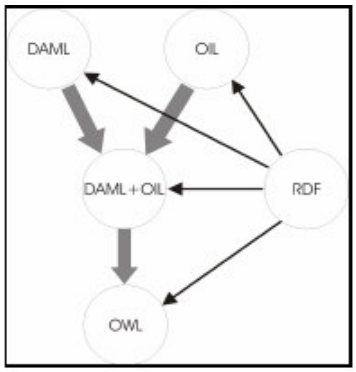


Figura 14: Evolução das linguagens de representação de ontologias

Segundo McGuinness (MCGUINNES, 2004), os recursos que a linguagem OWL oferece podem ser divididos em três sub-linguagens:

OWL *Lite* - dá suporte aos usuários que necessitam basicamente de uma hierarquia de classificação e funcionalidade com restrições simples. Apresenta apenas algumas características que a OWL oferece e também impõe algumas limitações aos recursos oferecidos.

OWL DL - é destinada aos usuários que necessitam de um pouco mais de

expressividade na formalização de ontologia. A OWL DL impõe todos os recursos apresentados pela OWL *Lite* e também todos os outros mecanismos que a linguagem OWL disponibiliza.

OWL *Full* - é a mais expressiva, pois, além de oferecer todos os vocabulários disponíveis pela linguagem OWL, não impõe restrição ao uso dos recursos. Essa sublinguagem não impõe restrição aos valores das classes, às propriedades, às instâncias e aos dados. Assim, é possível que uma classe seja tratada ao mesmo tempo como instância de uma classe.

3.4. JENA ONTOLOGY

A *Jena*[12] foi desenvolvida por Brian McBride da HP, e utilizada na criação e manipulação de grafos RDF. É uma API para a linguagem de programação Java para aplicações, que utiliza o padrão da *web* semântica, estabelecendo um formato de padronização e interpretação dos dados, e que proporciona uma programação de ambiente em RDF, RDF *Schema* e OWL, através da inclusão de restrições e de regras (JENA, 2003).

Segundo Gonçalves (GONÇALVES, 2004), a API *Jena* possui dois componentes, que são os objetos e classes para representar os modelos, recursos, propriedades e literais do RDF. As interfaces, representando recursos, propriedades e literais são chamadas de *Resource*, *Property* e *Literal*, portanto, um grafo RDF é chamado de modelo, e é representado por uma *Interface Model*.

Além de tudo, a API *Jena* está em constante progresso, para se ter uma idéia, na primeira versão, possuía apenas métodos que permitiam a manipulação de ontologias em DAML+OIL. Já na sua segunda versão, a API *Jena* é disponibilizada como um pacote específico, denominado API *Jena* 2 *Ontology*, a qual possui recursos para a manipulação de ontologias em RDF *Schema*, DAML+OIL e OWL. Para suporte das linguagens de ontologias, a API possui as seguintes classes: *OntClass* e *ObjectProperty*, e para cada uma das linguagens de ontologias existe um parâmetro que permite a construção de URI's de classes e propriedades, sendo que cada linguagem possui um parâmetro diferente, como exemplo na linguagem OWL, é *owl:ObjectProperty* e na DAML é *daml:ObjectProperty* (JENA, 2003).

[12] http://jena.sourceforge.net/

Finalmente, para criar um modelo na API *Jena* existem pacotes que realizam a interação com o RDF, e cada modelo criado possui uma interface chamada de *Reasoner*. Essa interface possui um conjunto de regras para cada linguagem de ontologia, como RDF *Schema*, DAML+OIL e OWL, sendo que essas regras são feitas para acessar a base RDF (GONÇALVES, 2004). Na Figura 15, é apresentada a base de um modelo de ontologia, interagindo com o RDF (JENA, 2003)**.**

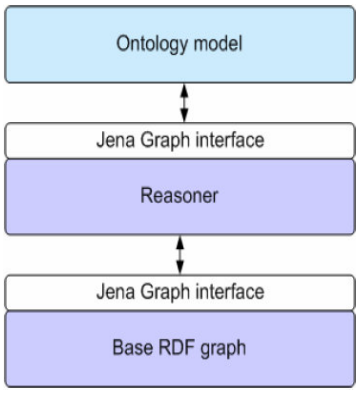


Figura 15: Modelo de ontologia com RDF (JENA, 2003)

3.4.1. Criando modelos de ontologia na API *Jena Ontology*

Um modelo de ontologia é uma extensão do modelo RDF da *Jena* que fornece poder de manipulação às ontologias, por exemplo, o *JenaModelfactory* provê a criação de modelos de ontologias, uma vez que, através da classe *OntModel*, é possível criar um modelo simples de ontologia. Além disso, a *Jena Ontology* possui capacidade para tratar a reusabilidade e importar diferentes tipos de dados, facilitando a comunicação. Na Figura 16, tem-se a declaração de um valor nesta API.

```
Exemplo de uma declaração Jena Ontology
<owl:Ontology rdf:about="">
<dc:creator rdf:value="Tiago Minuzzi"/>
<owl:imports rdf:resource=http://tiagominuzzi.com/teste/>
</owl:Ontology>
```

Figura 16: Exemplo de declaração na API *Jena*

A API *Jena Ontology* proporciona ainda a utilização de algumas classes principais como *DocumentManager*, a qual provê recursos para fazer manipulações de importação de ontologias; e a classe *OntDocumentManager*, que provê serviços para manipulação de documentos de ontologia, incluindo a importação dos documentos. A instância pode ser feita através de *OntDocumentManager.getInstance()*, e, para se ter um controle total do documento

gerado, cria-se um *OntModel* que recebe um objeto *OntDocumentManager* com parâmetro. Na Figura 17, verifica-se a criação de um *OntModel*.

```
Exemplo de criação de um OntModel
OntDocumentManager pdf = new OntDocumentManager();
OntModelSpec ont = new OntModelSpec( OntModelSpec.RDFS_MEM);
ont.setDocumentManager( pdf );
OntModel criarModelo = ModelFactory.createOntologyModel(ont, null);
```

Figura 17: Criação de um *OntModel*

Analisando a Figura 17, o modelo especificado (*OntModel*) faz referência ao novo documento criado, seguido de um parâmetro *"null"*, os quais identificam o tipo de ontologia utilizada, nesse caso, o modelo de ontologia do tipo OWL. Portanto, mudanças nas propriedades do documento original afetam os documentos que serão criados posteriormente.

Para acessar e manipular a API *Jena*, usa-se um pacote da HP chamado *"com.hp.hpl.jena.ontology"* que habilita a representação da ontologia em RDF. Nesse pacote são trabalhadas as linguagens OWL e a DAML+OIL (JENA, 2003).

Abaixo estão descritos alguns recursos oferecidos por esse pacote de manipulação (JENA, 2003):

- *OntModel:* é o modelo de ontologia que será visualizado pelo *Jena*, e que oferece recursos para os tipos de objetos esperados para a manipulação de uma ontologia, como: classes, propriedade e recursos. Na Figura 18 é descrito um exemplo de criação de um modelo de ontologia:

```
Exemplo de criação de um modelo de ontologia
modelo = ModelFactory.create.OntologyModel.createOntololyModel();
```

Figura 18: Criação de um modelo de ontologia

Quando ocorre a criação de um modelo de ontologia, conforme descrito no exemplo, utiliza-se a OWL como linguagem padrão. Logo, para se criar um modelo de ontologia, deve-se apenas especificar um URI da linguagem de ontologia, conforme a Tabela 2.

Tabela 2: URI's das linguagens suportadas pela API *Jena*

Linguagem de Ontologia	URI
RDFS	http://www.w3.org/2000/01/rdf-schema#
DAML+OIL	http://www.daml.org/2001/03/daml+oil#
OWL Full	http://www.w3.org/2002/07/owl#
OWL DL	http://www.w3.org/TR/owl-features/#term_OWLDL
OWL Lite	http://www.w3.org/TR/owl-features/#term_OWLLite

- *OntResource:* todas as classes nesta API *Ontology* que representam valores de ontologia possuem o *OntResource* como uma superclasse. Portanto, isso torna o *OntResource* um bom lugar para se colocar funcionalidades compartilhadas para todas as classes da ontologia. Logo, como a interface *OntResource* estende a interface *Resource* da *Jena* RDF, um método pode aceitar um recurso. Além disso, a *OntResource* possui diversos atributos que são expressos como métodos para a utilização de ontologias (JENA, 2003).

- *OntClass*: essa API *Ontology* representa uma classe através de um objeto *OntClass*. A representação de uma classe proporciona a criação de várias instâncias que apresentam as características dessa classe, o que pode ser demonstrado através de uma chamada ao método *getOntClass()* no modelo de ontologia, conforme é mostrada na Figura 19.

```
Exemplo de chamada de uma classe na Jena Ontology
String al = http://tiagominuzzi.com/ontology/aluno/#
OntClass aluno = m.getOntClass( al + "Aluno" );
```

Figura 19: Chamada na API *Jena*

Para realizar a criação de uma classe com um recurso que não existe, é possível utilizar o método *createClass()*, que tem esse objetivo, que é observado na Figura 20.

```
Exemplo de criação uma classe na Jena Ontology
String al = http://tiagominuzzi.com/ontology/aluno/#
OntClass disciplina = m.createClass( al + "Disciplina" );
```

Figura 20: Criar uma classe na API *Jena*

Criando um tipo de objeto *OntClass*, pode-se chamar os métodos definidos nessa classe que permitem a manipulação desses objetos. Logo, os atributos da *OntClass* são similares aos atributos do *OntResource*, com uma coleção de métodos para modificar *(set)*, adicionar *(add)*, listar *(list)* e remover *(remove)* valores. A API Jena *Ontology* proporciona a utilização de diversos recursos que podem ser verificados no portal da API (JENA, 2003).

4. TRABALHOS RELACIONADOS

Este capítulo tem por objetivo apresentar trabalhos que utilizam à técnica da refatoração nos requisitos e comparar com a solução desenvolvida.

4.1. CASCADED REFACTORING

Esse trabalho é desenvolvido por Butler (BUTLER, 2001) e Xu (XU, L., 2005), que propõem uma metodologia chamada de *"Cascaded Refactoring"* ou refatoração em cascata e cujo objetivo é definir modelo seqüencial (um após o outro) de refatoração para cada etapa de desenvolvimento de um *framework*, por meio de um conjunto de refatorações que são aplicadas inclusive nos requisitos do *software*.

Um *framework* é determinado por um conjunto de modelos como: modelo de característica, modelo de caso de uso, modelo arquitetural, modelo de projeto e o código fonte do *framework*; as quais são etapas que devem ser seguidas no processo de desenvolvimento de conjunto de aplicações ou até mesmo no desenvolvimento de um *software* (XU, L., 2005).

O processo de refatoração em cascata é composto por uma série de refatorações que devem ser seguidas de acordo com os modelos e suas conexões; e dentro desse processo podemos citar seis tipos de refatorações:

- **Refatorações de modelos de características:** abrangem os aspectos do *software* utilizado para caracterizar o sistema com os usuários, clientes ou desenvolvedores, identificando também variáveis, componentes comuns, características de funcionalidade que promovem a reusabilidade. As refatorações são aplicadas por sete refatorações que contemplam este modelo.

- **Refatorações de modelos de casos de uso**: têm o objetivo de refatorar os requisitos funcionais. Estas refatorações são desenvolvidas por meio de especificações de casos de usos através da descrição por templates e nestes são identificadas apenas as pré-condições. Um conjunto de refatorações para este modelo são propostas, estas são citadas no decorrer da apresentação desta seção;

- **Refatorações de modelos arquiteturais**: provêem uma elevada abstração dos componentes no desenvolvimento do *software*. As refatorações aplicadas por este

modelo garantem a preservação da funcionalidade do sistema, por meio da quebra de um sistema em subsistemas, ocorrendo assim à aplicação deste modelo por meio de cinco casos de refatoração;

- **Refatorações de modelos de projetos**: objetivam mostrar detalhes ilustrativos do sistema, através de uma linguagem de modelagem, a escolhida para a representação deste modelo é a UML, desta forma por meio da estrutura e comportamento da UML é enfatizado a organização estática para a refatoração dos objetos de um sistema, por meio das classes, interfaces, atributos e métodos;

- **Documentação da refatoração e do código fonte:** proporcionam melhor entendimento sobre como ocorreu todo o processo, desta forma na refatoração em cascata é sugerido a utilização da linguagem de programação C++, devido a esta suportar polimorfimo, heranças, templates, parametrizações e outros.

- **Conexão dos modelos:** esses modelos devem ser interligados através de mapas para conseguir se comunicar e capturar os aspectos comuns, e cujos impactos são tratados pelo mapeamento entre os modelos, tornando assim um dependente do outro.

Como o foco dessa dissertação são as refatorações de requisitos, serão abordados mais detalhes sobre como ocorrem as refatorações de modelos de casos de uso, sendo que, para tanto, as refatorações de modelos de casos de uso estão separadas em três conjuntos: o primeiro conjunto é aplicado aos atores e casos de uso *(Create_abstract_actor, Create_abstract_usecase, Merge_actors, Split_actor, Merge_behaviours)*; o segundo conjunto é aplicado à distribuição do comportamento dos episódios de casos de uso *(Make_episode_usecase)*; e o último propõe uma hierarquia de generalização para os casos de uso *(Move_episode_to_parent_usecase e Move_episode_to_child_usecase)*. Todas essas refatorações têm objetivos comuns, tornando os requisitos do modelo de refatoração em cascata menos redundantes, o que proporciona a evolução do modelo.

A seguir é mostrado um exemplo de refatoração de caso de uso aplicada por Xu (XU, L., 2005), no seu modelo de *"Cascaded Refactoring"*:

- **Nome da Refatoração:** *Merge_Behaviours*

- **Descrição:** identificam-se dois casos de uso u_1 e u_2, e, a partir desses, cria-se um caso de uso u com os episódios de u_1 e u_2, substituindo-os.

- **Parâmetros:** $u_1 ; u_2 ; u$

- **Pré-Condições:** identifica-se um grupo de casos de uso relacionados a um ator. Esses casos de uso são substituídos por um novo caso de uso, contendo todos os episódios dos anteriores, sendo que não é necessário existirem episódios iguais entre os casos de uso.

O desenvolvimento dessas refatorações em cascata segue os conceitos de Opdyke (OPDYKE, 1992) e Roberts (ROBERTS, 1999), proporcionando usabilidade, portabilidade e garantindo a mesma saída após a aplicação das refatorações. Além disso, através desse modelo é possível observar o conceito de refatoração em cascata, determinando um conjunto de modelos de refatorações, que podem e devem ser aplicados no desenvolvimento de um *framework* e em projetos que abrangem uma grande complexidade (XU, L., 2005).

4.2. USE CASE REFACTORING: A TOOL AND CASE STUDY

Essa ferramenta consiste em uma tese de doutorado desenvolvido no ano de 2004 e um dos seus principais objetivos é desenvolver uma ferramenta que apresente uma evolução de cenários, através da aplicação das refatorações para os modelos de casos de uso (XU, J., 2004), a qual é utilizada para capturar os requisitos do sistema, melhorando a reusabilidade, entendimento e desenvolvimento do projeto com requisitos eficazes e concretos. Para tanto, ele utiliza o conceito de refatorações de modelos de casos de uso apresentado em Butler (BUTLER, 2001), para a definição de um meta-modelo de caso de uso e uma lista de refatorações de caso de uso, é seguido o trabalho desenvolvido por Rui (RUI, 2003).

Para o desenvolvimento da ferramenta, é necessário definir três níveis: o nível de ambiente, que são os casos de uso relacionado com as entidades; o nível de estrutura, em que os casos de uso são mostrados juntamente com diferentes partes e ligações; e, por último, o nível de evento, em que os eventos individuais são caracterizados por um baixo nível de abstração. Essa ferramenta é desenvolvida na linguagem de programação Java, através de uma *framework* chamada *DrawLets*[13], que é uma aplicação gráfica que possui componentes prontos com o intuito de facilitar o desenvolvimento da aplicação. Na Figura 21, é mostrada a ferramenta de refatoração desenvolvida, destacando-se o nível de ambiente que mostra os diagramas de casos de uso, sendo que *"Episode Tree"* mostra a estrutura de cada caso de uso no formato de uma árvore, e *"Episode Table"*, por sua vez, uma lista dos episódios correntes.

[13] http://www.rolemodel*software*.com/drawlets/index.php

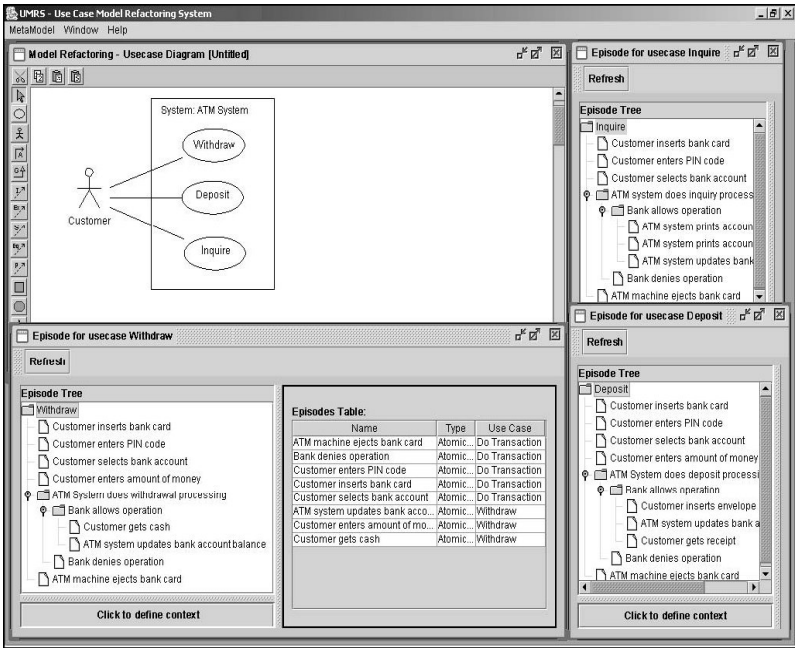


Figura 21: Tela da ferramenta desenvolvida (XU, J., 2004)

A aplicação de uma refatoração é de fácil utilização pelo usuário, uma vez que ocorre através da seleção de um caso de uso e de qual refatoração está disponível para ser aplicada através de menus, podendo ser visualizada na Figura 22. Já a ferramenta utiliza para a sua estruturação de dados e de armazenamento o XML, por meio da especificação *Document Type Definition* (DTD), que garante consistência no armazenamento dos dados.

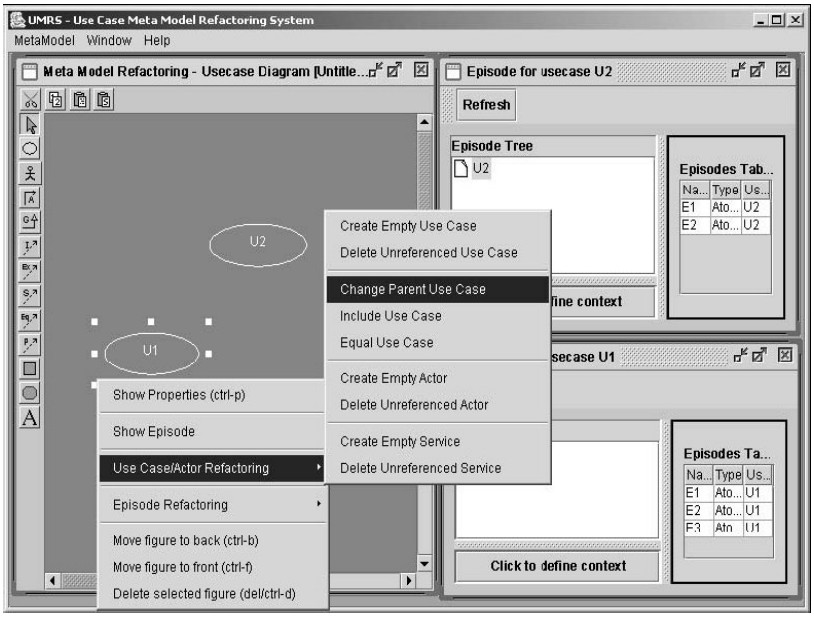


Figura 22: Aplicação de uma refatoração de caso de uso (XU, J., 2004)

No entanto, para ocorrer a especificação de uma refatoração, Xu (XU, J., 2004) determina passos para especificar cada caso de refatoração, sendo que cada uma deve ter: motivação, processo de refatoração, argumentos, pré-condições e pós-condições. Além disso, a ferramenta apresenta quatro principais módulos:

- **Nível de ambiente:** apresenta o módulo de definição dos casos de uso, ator e dos serviços que são usados para descrever o sistema (seguindo a especificação UML) e o módulo de descrição dos objetivos que descreve o objetivo do ator no sistema;

- **Nível de estrutura:** apresenta o módulo de descrição dos episódios, em que se define o contexto, as pré-condições e as pós-condições;

- **Nível de eventos:** nesse nível é determinado o módulo de descrição dos eventos através dos dados mostrados e interpretados, e dos módulos de refatoração.

É importante salientar ainda que as refatorações ocorrem através de restrições entre os níveis, alinhando um mapa de ligação entre eles. Então, assim são determinados os seguintes mapas (ligações): o do nível de ambiente para o nível de estrutura, o do nível de estrutura para o nível de ambiente, o do nível de estrutura para o nível de evento e o do nível de evento para o nível de estrutura; o que torna possível aplicar e definir os argumentos, as pré-condições e as pós-condições para cada refatoração.

A seguir, é apresentado um exemplo de refatoração que é aplicado e processado nessa ferramenta, a qual segue as seguintes descrições:

- **Motivação:** refatoração de generalização de caso de uso, que é realizada quando dois ou mais casos de uso são isomórficos (mesma forma), compartilham episódios primitivos comuns, e ainda existe um *episode tree*;

- **Processo de Refatoração:** um novo caso de uso pai é gerado e episódios comuns primitivos são removidos desse caso. São gerados relacionamentos de generalização daqueles casos de uso para o caso de uso pai. Essa refatoração ajuda a eliminar a redundância e melhora a reusabilidade;

- **Argumentos:** u_1 ; u_2 , casos de uso escolhidos;

- **Pré-Condições:**

- Casos de uso u_1 e u_2 são isomórficos;
- Caso de uso u_1 tem *episode tree* t_1, caso de uso u_2 tem *episode tree* t_2. Existe um *episode tree* t, em que t_1 e t_2 são casos especiais de t;
- t_1 e t_2 compartilham episódios comuns primitivos;

- **Pós-Condições:**
  - Um novo caso de uso u com os *episode tree* t são gerados;
  - Relacionamento de generalização de u_1 e u_2 para o caso de uso u é gerado;
  - Episódios primitivos são removidos de casos de uso filhos u_1 e u_2 para um caso de uso pai u;
  - Relacionamentos comuns entre $\{u_1, u_2\}$ e outros casos de uso ou atores são tirados de $\{u_1, u_2\}$ para o caso de uso u;

Na Figura 23, é demonstrada uma aplicação prática da refatoração de generalização de caso de uso, onde, à esquerda são demonstrados os casos de uso não refatorado, e pode-se verificar que possuem uma funcionalidade em comum, gerando, assim, ambigüidades; já à direita são visualizados os casos de uso no seu modelo refatorado, com a funcionalidade em comum agrupada em um novo caso de uso abstrato. Sendo assim, cada caso de uso possui seus respectivos episódios após a refatoração, os quais são as bases para refatorar.

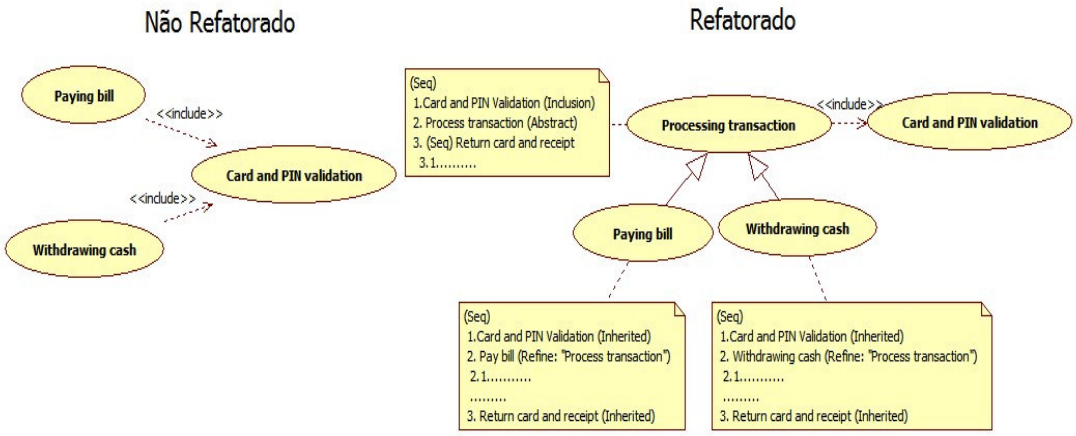


Figura 23: Exemplo de aplicação da refatoração (XU, J., 2004)

Sugere-se a refatoração, apresentada na Figura 23, porque diminui a redundância e melhora a reusabilidade, pois o processo para a aplicação dessa refatoração através da ferramenta ocorre por meio da automatização das pré-condições e das pós-condições dos componentes que serão refatorados e da determinação dos objetivos dos atores e dos episódios e eventos de cada caso de uso do sistema, sendo que a determinação dos episódios e eventos é de fundamental importância para a aplicação da refatoração, uma vez que através desse recurso são validadas as condições para sua aplicação.

Para finalizar, é importante observar que essa ferramenta provê a evolução e o desenvolvimento de um caso de uso, e os resultados demonstram que essas refatorações facilitam o processo de reusabilidade de fragmentos de requisitos, reduzindo o tempo de elaboração de um *software* e melhorando a qualidade dos requisitos. Portanto, ela desempenha um papel muito importante dentro do processo de Engenharia de Requisitos, especialmente na etapa de elicitação de requisitos, pois, através da técnica da refatoração é possível obter requisitos sem redundância de acordo com a interpretação do usuário da aplicação.

4.3. CONSIDERAÇÕES SOBRE TRABALHOS RELACIONADOS

Esse trabalho pode ser considerado inovador, uma vez que busca unificar dois conceitos que envolvem discussões no desenvolvimento de *software*: o de refatoração e o de processo de Engenharia de Requisitos. Assim, é feita a seguir uma breve comparação entre alguns projetos, conforme a Tabela 3, e posteriormente essa comparação é discutida a fim de justificar o desenvolvimento dessa dissertação.

Tabela 3: Quadro comparativo com trabalhos relacionados

Característica	***UStory-Refactory***	***Cascaded Refactoring***	***Use Case Refactoring***
Processo de Refatoração (Pré e Pós-Condições)	✓		✓
Aplicação de Ontologias	✓		
Processo de Engenharia de Requisitos	✓	✓	✓
Promove interoperabilidade entre aplicações	✓		
Aplicação da Técnica de Refatoração de Requisitos	✓	✓	✓
Base de requisitos refatorados	✓		
Tecnologia Java	✓		✓

O "*Cascaded Refactoring*" (XU, L., 2005) tem o objetivo de propor uma metodologia de refatoração em cascata, através da definição de um modelo de refatoração para cada etapa do desenvolvimento de um *software*, tornando cada refatoração dependente de outra refatoração. Além disso, esse trabalho aborda os requisitos através das refatorações de modelos de casos de uso, logo, são propostas pré-condições para a definição destas refatorações. Esse trabalho salienta também que, para que se tenha um desenvolvimento de *software* eficiente e eficaz, é necessário passar por um conjunto de refatorações, com o objetivo de obter a sua eficiência e eficácia, eliminando prováveis redundâncias.

Já o *"Use Case Refactoring"* (XU, J., 2004) procura abordar apenas a refatoração de requisitos através de modelos de casos de uso e cenários, sendo que, para aplicar essa refatoração, é desenvolvida uma ferramenta que proporciona a evolução de um *software* através de seu cenário refletido nos diagramas de casos de uso. Nesse trabalho são aplicados os tipos de refatorações de casos de uso definidos em Buttler (BUTLER, 2001) e Rui (RUI, 2003) e seu objetivo é proporcionar uma nova visão de aplicação da refatoração através da abstração dos requisitos, conseguindo capturar os requisitos de uma forma eficiente, para ajudar e facilitar o desenvolvimento do *software* por parte do desenvolvedor.

Assim, através de uma breve comparação com os trabalhos relacionados apresentados, pode-se concluir que eles têm o mesmo objetivo que o trabalho desenvolvido, ou seja, o da aplicação de refatoração aos requisitos, embora apresentem propósitos diferentes, uma vez que essa dissertação visa a aplicar a técnica de refatoração a um conjunto de requisitos formatados em cartões *User Stories (CRC Cards)* da metodologia de desenvolvimento XP, apresentando e formalizando uma nova forma e contextualização da técnica de refatoração de requisitos. Esses requisitos são representados por uma ontologia no formato da linguagem OWL, ao qual é aplicada a técnica de refatoração descrita nesta dissertação. Já nos outros trabalhos, a refatoração é aplicada nos modelos de casos de uso e cenários.

Portanto, nessa dissertação são tratados alguns aspectos diferentes em relação aos trabalhos relacionados, pois nela os dados são representados por uma ontologia OWL que representa um domínio de requisitos, tendo um significado, e nos outros, os dados são representados por uma sintaxe XML. Nesse domínio de requisito, propõe-se aplicar a técnica de refatoração de requisitos completo por meio de restrições e regras, adotando as premissas básicas de refatoração, ou seja, as pré e pós-condições. Outro aspecto relevante é a interoperabilidade entre as aplicações, que proporciona a mesma semântica dos dados, devido

a eles estar representados por uma sintaxe XML+RDF+OWL. E por último destaca-se a apresentação de uma base de requisitos refatorados que é persistida em um novo arquivo no formato OWL.

5. A FERRAMENTA *USTORY-REFACTORY*

Como mencionado anteriormente, grande parte da indústria de *software* tem demonstrado interesse na técnica de refatoração, buscando melhorar ou aperfeiçoar o código fonte, e, assim, tornando o *software* legível. No entanto, essa técnica não se resume apenas em nível de código fonte, sendo também estendida para um formato mais abstrato como exemplo nas especificações de requisitos de casos de uso (XU, J., 2004; YU, 2004). Seguindo essa idéia, é desenvolvida uma ferramenta e metodologia que aplica um novo modelo de extensão para a técnica de refatoração, que é o de aplicar essa técnica nas especificações de requisitos da metodologia de desenvolvimento *Extreme Programming*, as quais são *User Stories (CRC Cards)*.

Este capítulo tem o objetivo de apresentar o processo de design e implementação da ferramenta de refatoração *UStory-Refactory* e sua arquitetura, juntamente com o seu processo de modelagem UML. Primeiro, é importante dizer que essa ferramenta realiza a adaptação da técnica de refatoração, originalmente aplicada na programação (nível de código fonte), através da obtenção de regras e restrições, e aplicado no processo de refatoração em requisitos no formato de cartões *User Stories (CRC Cards)* utilizados na metodologia de desenvolvimento *Extreme Programming* (XP). Além disso, ela proporciona a geração de requisitos refatorados, e a interoperabilidade entre a *UStory-Refactory* e a ferramenta origem, que gera os dados a serem refatorados, através de uma ontologia com sintaxe XML+RDF+OWL, garantindo, assim, semântica e formatação dos dados.

A ferramenta é desenvolvida através da linguagem de programação Java, e no seu desenvolvimento destaca-se a abordagem de conceitos e paradigmas de orientação a objetos. Destaca-se a automatização parcial da técnica da refatoração, pois muitas vezes o usuário necessita interagir com a ontologia OWL, fornecendo um conceito mais apropriado aos dados refatorados. O usuário recomendado para a utilização desta ferramenta deve ser uma analista de sistemas considerado sênior, que tenha características de arquiteto de *software*, possuindo conhecimentos avançados referentes a processos de construção, de linguagem de modelagem e de linguagem de programação de *software*. A Figura 24 mostra uma visão macroscópica, de como se origina a obtenção dos dados em que é aplicada a técnica de refatoração de requisitos.

Figura 24: Visão macroscópica da ferramenta.

As principais características da ferramenta e seus modelos são apresentados a seguir.

5.1. ARQUITETURA DA *USTORY-REFACTORY*

Com o objetivo de agregar valores à área da Engenharia de *Software*, a ferramenta foi desenvolvida através de uma arquitetura que possibilita a comunicação entre duas aplicações através da mesma semântica dos dados, e apresentando também um mecanismo de refatoração de requisitos aplicados nos requisitos em formato de cartões *User Stories (CRC Cards)* utilizados na metodologia de desenvolvimento XP. Esse mecanismo interage com os requisitos por meio de uma ontologia, e sua arquitetura é mostrada na Figura 25, sendo que, a seguir, será descrito como ocorreu o desenvolvimento de cada camada.

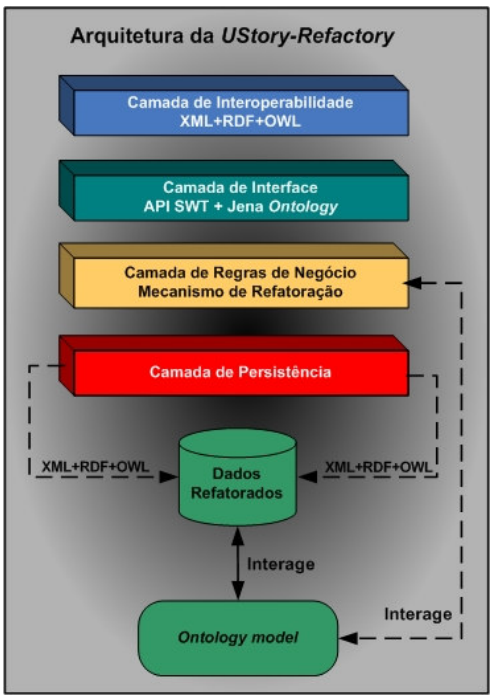


Figura 25: Arquitetura da *UStory-Refactory*

5.2. CAMADA DE INTEROPERABILIDADE

A camada de interoperabilidade é aquela que tem a responsabilidade de prover a comunicação entre a aplicação origem e a *UStory-Refactory*. Para tanto, a interoperabilidade entre as aplicações é proporcionada através de uma ontologia gerada com a sintaxe XML e tecnologias RDF e OWL com mesma representação semântica dos dados, sendo que é nessa ontologia que será aplicada a técnica de refatoração de requisitos. É nessa camada que ocorre o processo de formalização e obtenção dos dados (requisitos) para a aplicação da refatoração, estes seguem os seguintes passos, que podem ser vistos na Figura 26.

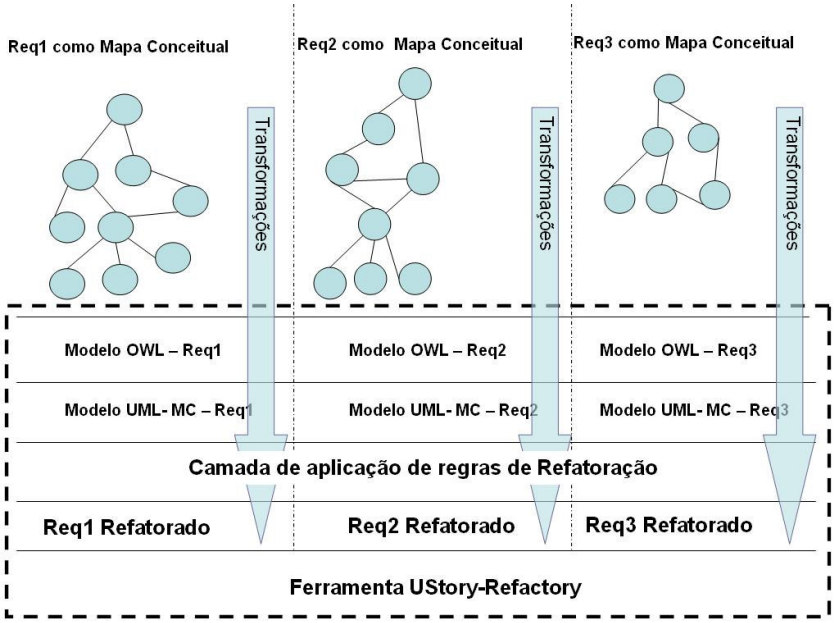


Figura 26: Processo de obtenção dos requisitos

- Primeiramente, os requisitos são modelados por meio de mapas conceituais, sendo posteriormente exportados para um arquivo em formato OWL que serve como entrada para a ferramenta *UStory-Refactory*;

- Após, os requisitos em formato OWL são transformados em um modelo de classes e objetos por meio da UML-MC (ROBINSON, 2004), que estende a UML para modelar mapas conceituais em diagramas de classes da linguagem de modelagemUML;

- E para finalizar essa transformação de mapas conceituais (em formato OWL) para classes em formato UML, o processo de refatoração inicia, verificando se cada

requisito é passível ou não de refatoração, por meio da ferramenta de refatoração *UStory-Refactory*.

É importante destacar a utilização do projeto proposto por Robinson (ROBINSON, 2004), denominado UML-MC, foi utilizado pela *UStory-Refactory* apenas para realizar a transformação e formalização dos requisitos de mapas conceituais para os diagramas de classes da UML. A exportação para OWL é de responsabilidade da *UStory-Refactory*. Desta forma na próxima subseção será descrito de uma forma mais detalha como ocorre o processo de formalização e obtenção da ontologia OWL desenvolvida em que será aplicada a refatoração.

5.2.1. Formalização dos dados

A formalização dos dados da ontologia é uma etapa importante, pois é nesse dado que será aplicada a refatoração nos requisitos, pois os dados são os objetos de aplicação da refatoração oriunda por meio de uma ferramenta que elicita os requisitos por mapas conceituais, os quais representarão as *User Stories (CRC Cards)* da metodologia XP. Assim, os mapas conceituais são representados por *"conceito+ligação+conceito"*, formando uma tripla, as quais, por sua vez, irão formar o modelo de acordo com a UML-MC (ROBINSON, 2004).

Portanto, a UML-MC (ROBINSON, 2004) tem o objetivo de estender diagramas de classes da linguagem de modelagem UML para suportar o uso de mapas conceituais, sendo que a formalização desses dados ocorre no formato OWL, por meio da *UStory-Refactory*, esta segue a formalização de acordo com o padrão de Smith (SMITH, 2002) e o ODM – *Ontology Definition Metamodel* (OMG, 2005), estes proporcionam uma ontologia que representa o conhecimento de um domínio de requisitos por meio de diagramas de classes da UML, os quais contemplam: classes, relacionamentos (associação com direção, associação, composição, agregação e generalização), atributos e operações. É importante destacar ainda a formalização da UML em OWL é bastante abstrata, logo, ela não estende elementos como tipo de atributos e de métodos e nem os parâmetros dos métodos, e, além disso, é importante salientar que a ontologia foi formalizada de acordo com a necessidade deste trabalho.

5.2.2. Gerador de Interoperabilidade

A ontologia deve ser gerada por uma ferramenta origem, mas, com o objetivo de agregar valores à área da Engenharia de Requisitos e de tornar a *UStory-Refactory* uma ferramenta independente, foi desenvolvido um gerador de interoperabilidade que realiza a formalização

dos diagramas de classes para o formato de uma ontologia no formato OWL. Logo a seguir, é mostrado como esta ontologia de requisitos é formada, uma vez que, na Figura 27, temos um exemplo de uma *User Story (CRC Card),* que conta a história de *"atividades de uma empresa"*, na qual pode ser aplicada o processo de refatoração. Já na Tabela 4, temos a formalização desses dados em triplas, e, na Figura 28 verifica-se a visualização de requisitos no formato de mapas conceituais, observando que eles são bastante dinâmicos e remodeláveis. Além disso, mais adiante, na Figura 29 observa-se a representação de mapas conceituais adaptados no padrão UML-MC (ROBINSON, 2004) através de um diagrama de classes da linguagem de modelagem UML, no qual é apresentada a história da *User Story (CRC Card)* na UML.

Exemplo de uma *User Story (CRC Card)* a ser Refatorado

"Uma empresa gera departamentos que tem o objetivo de ajudar na administração. Serviços podem ser pagos por ouro card, master card, american card, visa card e dinheiro. Estes possuem há operação efetuar pagamento em comum. Serviços têm o objetivo de executar formas de pagamentos. Destaca-se ainda a forma de pagamento ouro card, pois este possui vantagens em relação aos outros e possui o cálculo especial de bônus. Serviços proporcionam faturas. A empresa possui dois tipos de pessoas: funcionário e cliente, ambos possuem nome e endereço, e operações de cadastrar, consultar e remover em comum. A pessoa funcionário possui ainda o dado salário e a operação cálculo do salário. A empresa ainda fornece vale alimentação, vale transportes e seguro de vida, ambos possuem descontos."

Figura 27: *User-Story*, em que pode ser aplicado o processo de refatoração

Tabela 4: Representação de uma *User-Story* em triplas

Conceito	Ligação	Conceito
Empresa	Fornece	Serviços
Empresa	Possui	Pessoa
Empresa	Fornece	Vale Alimentação
Empresa	Fornece	Vale Transporte
Empresa	Fornece	Seguro de Vida
Empresa	Gera	Departamentos
Dinheiro	Paga	Serviços
OuroCard	Paga	Serviços
MasterCard	Paga	Serviços
VisaCard	Paga	Serviços
AmericanCard	Paga	Serviços
Serviços	Originam	Faturas
OuroCard	Proporciona	Vantagens
Pessoa	É um	Funcionário
Pessoa	É um	Cliente

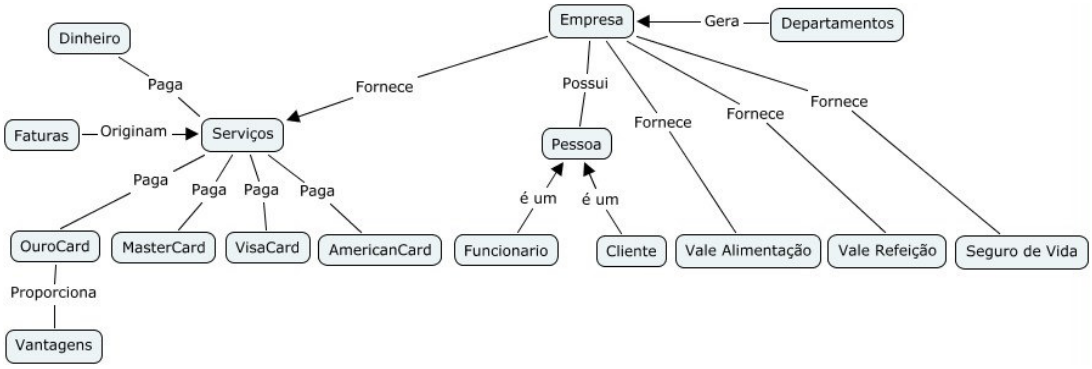


Figura 28: *User-Story*, no formato de mapas conceituais

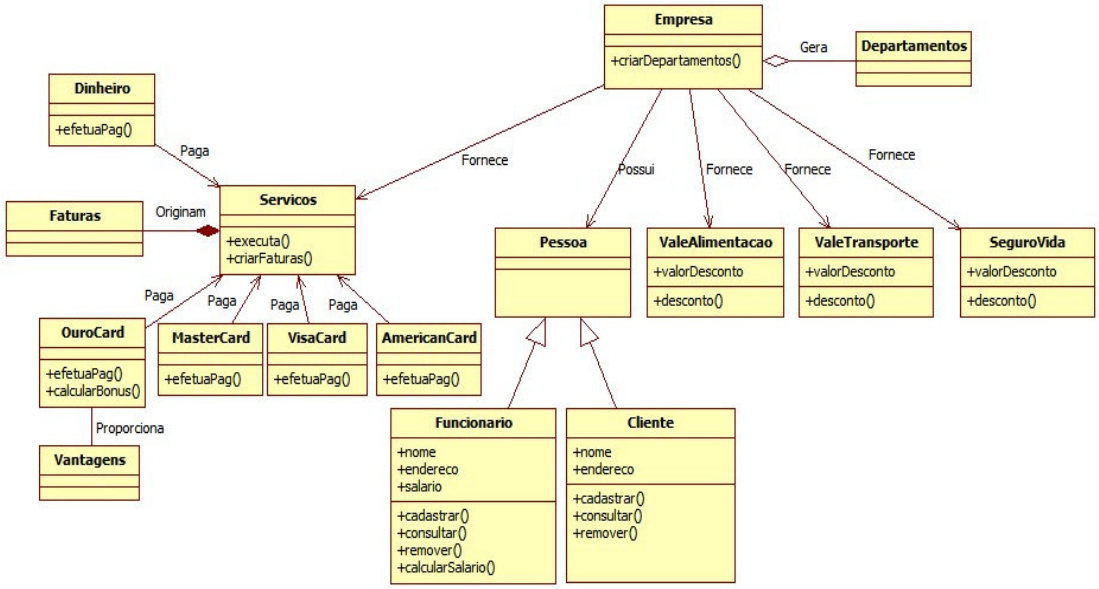


Figura 29: Formalização de acordo com UML-MC (ROBINSON, 2004)

Conforme já visto no parágrafo anterior, um gerador de interoperabilidade foi desenvolvido, para garantir a semântica dos dados a serem refatorados e a interoperabilidade entre as aplicações. Destacam-se, assim, a implementação desse gerador, e a declaração do método *ModelFactory.createOntologyModel (OntModelSpec.OWL_MEM)*, cujo objetivo é criar uma ontologia com as especificações OWL.

Além disso, esse gerador busca atender às especificações de um diagrama de classes da linguagem de modelagem UML, de acordo com a UML-MC (ROBINSON, 2004), e nele são inseridos os seguintes componentes de um diagrama de classe: as classes, os atributos, as operações e os relacionamentos do tipo associação, associação com direção, agregação, composição e generalização. Sendo que, para ocorrer à criação de um tipo de relacionamento

é necessário indicar a classe origem e a classe destino de um relacionamento, conseguindo assim saber a direção do relacionamento.

Portanto, o funcionamento do gerador de interoperabilidade ocorre através da interação com o usuário, o qual deverá selecionar a opção *"Gerar UML - OWL"* do menu arquivo da ferramenta. Como pode ser observado, a Figura 30 é expandida, e o usuário possui duas opções de preenchimento do formulário: *"Classes"* e *"Relacionamentos"*, os quais são obrigatórios para gerar a ontologia. Assim, quando o usuário selecionar o formulário *"Classes"*, ele poderá criar e remover as classes, atributos e métodos. Já quando for selecionado o formulário *"Relacionamentos"*, conforme a Figura 31, é possível mapear ou remover um tipo de relacionamento entre uma classe origem e destino. Para finalizar a criação do arquivo OWL é selecionado botão *"Salvar dados"*.

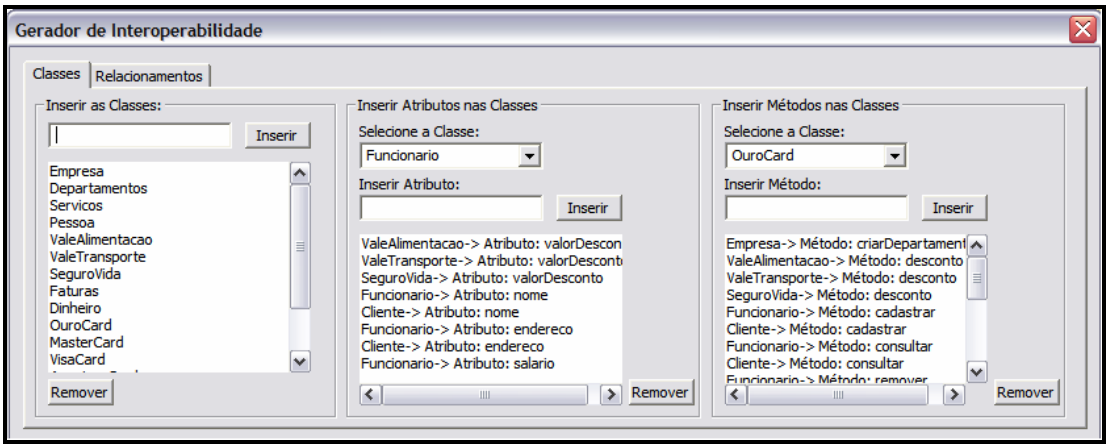


Figura 30: Inserção de classes, atributos e métodos do diagrama de classe

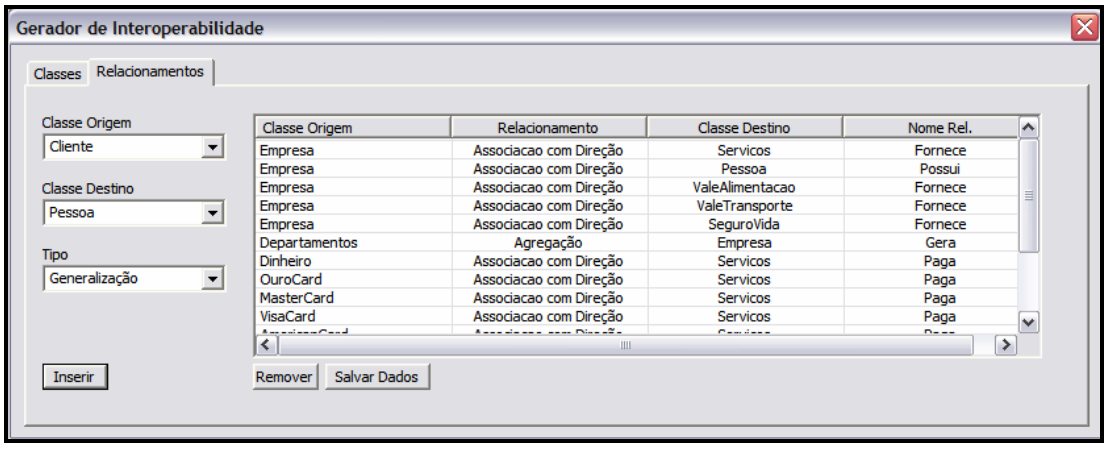


Figura 31: Mapeamento de relacionamentos entre as classes do diagrama de classe

A seguir são descrito fragmentos (seções) da ontologia, com a identificação dos respectivos elementos que esse gerador contempla.

O arquivo inclui a *tag* padrão de cabeçalho RDF, conforme a Figura 32 mostra:

```
<rdf:RDF
```

Figura 32: Cabeçalho padrão RDF

Por suas vez, os modelos RDF, RDF *Schema*, OWL e DAML estão inclusos no arquivo para que através deles se especifique a sintaxe do mesmo, sendo que, na Figura 33, é visualizada a sintaxe dos modelos.

```
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
    xmlns:daml="http://www.daml.org/2001/03/daml+oil#">
```

Figura 33: Sintaxe dos modelos

Já as classes são formas de representar cada indivíduo da ontologia, que é representada pela declaração *class*, conforme mostra a Figura 34.

```
  <owl:Class rdf:about="http://ontology.owl#Empresa"/>
```

Figura 34: Declaração de uma classe

A seguir, na Figura 35 é mostrada a declaração da classe *"Pessoa"*, que, por sua vez, possui apenas uma referência do tipo *subClassOf* para indicar a herança. Portanto, a representação *subClassOf*, através da OWL, pode ser interpretada como uma herança ou relacionamento do tipo generalização da representação UML. Logo, a classe *"Funcionario"* proporciona uma herança em relação à classe *"Pessoa"*.

```
  <owl:Class rdf:about="http://ontology.owl#Funcionario">
    <rdfs:subClassOf rdf:resource="http://ontology.owl#Pessoa"/>
  </owl:Class>
```

Figura 35: Declaração de uma generalização

Os atributos são propriedades nomeadas de uma classe que descreve um intervalo de valores que as instâncias da propriedade podem apresentar, e é representada pela propriedade OWL *DataypeProperty*, sendo que na Figura 36, é possível verificar a criação do atributo *"nome"*, que está presente em ambas as classes: *"Cliente"* e *"Funcionario"*.

```
  <owl:DatatypeProperty rdf:about="http://ontology.owl#nome">
    <rdfs:domain rdf:resource="http://ontology.owl#Cliente"/>
    <rdfs:domain rdf:resource="http://ontology.owl#Funcionario"/>
  </owl:DatatypeProperty>
```

Figura 36: Declaração de um atributo

A representação das operações (métodos) de uma classe é similar aos atributos, se diferenciando na propriedade OWL, pois as operações são especificadas pela propriedade *TransitiveProperty,* que se classificando como uma instância de uma propriedade dentro da OWL. Além disso, as operações são implementações de um serviço que pode ser solicitado (instanciado) por algum objeto de uma classe que está presente na ontologia. Verifica-se na Figura 37 a declaração de uma operação *"cadastrar()"*, que está presente nas classes *"Cliente"* e *"Funcionario"*. É importante notar ainda que essas operações possuem um mesmo comportamento.

```
<owl:TransitiveProperty rdf:about="http://ontology.owl#cadastrar">
  <rdfs:domain rdf:resource="http://ontology.owl#Cliente"/>
  <rdfs:domain rdf:resource="http://ontology.owl#Funcionario"/>
</owl:TransitiveProperty>
```

Figura 37: Declaração de um método

As propriedades de objetos, ou *ObjectProperty,* são propriedades cujo objetivo é o de especificar relacionamentos do tipo associação com direção, associação, agregação e composição entre as classes, sendo que, nessa representação há duas propriedades importantes: *domain* e *range*, e, além disso, ela indica a quem pertence o relacionamento (classe origem) e a que classe está vinculado o relacionamento (classe destino). Consequentemente, os relacionamentos da modelagem UML são representados através disso.

Por sua vez, o relacionamento do tipo associação com direção é especificado pela propriedade *ObjectProperty* juntamente com as *tags domain*, que especificam a classe origem, e pela *range* que especifica a classe destino, o que pode ser visualizado na Figura 38.

```
<owl:ObjectProperty rdf:about="http://ontology.owl#Paga">
  <rdfs:domain rdf:resource="http://ontology.owl#AmericanCard"/>
  <rdfs:domain rdf:resource="http://ontology.owl#VisaCard"/>
  <rdfs:domain rdf:resource="http://ontology.owl#MasterCard"/>
  <rdfs:domain rdf:resource="http://ontology.owl#OuroCard"/>
  <rdfs:range rdf:resource="http://ontology.owl#Servicos"/>
  <rdfs:domain rdf:resource="http://ontology.owl#Dinheiro"/>
</owl:ObjectProperty>
```

Figura 38: Declaração de uma associação com direção

Na seqüência, o relacionamento do tipo associação é especificado pela propriedade *ObjectProperty* juntamente com as *tags domain* e *range*, no entanto nesse relacionamento existe a *tag inverseOf* que específica que este relacionamento não possui direção, conforme mostra a Figura 39.

```
<owl:ObjectProperty rdf:about="http://ontology.owl#Proporciona">
  <owl:inverseOf>
    <owl:ObjectProperty rdf:about="http://ontology.owl#ProporcionaI"/>
  </owl:inverseOf>
  <rdfs:range rdf:resource="http://ontology.owl#OuroCard"/>
  <rdfs:domain rdf:resource="http://ontology.owl#Vantagens"/>
</owl:ObjectProperty>
```

Figura 39: Declaração de uma associação

Já a agregação, na Figura 40, segue o mesmo padrão de especificação de relacionamentos já apresentada, mas neste tipo de relacionamento é acrescida a *tag someValuesFrom*, esta tem o significado de agregação, ou seja, representa um relacionamento do tipo "tem-um", o que significa que um objeto contém os objetos das partes.

```
<owl:ObjectProperty rdf:about="http://ontology.owl#Gera">
  <rdfs:range rdf:resource="http://ontology.owl#Empresa"/>
  <rdfs:domain rdf:resource="http://ontology.owl#Departamentos"/>
</owl:ObjectProperty>
<owl:Restriction rdf:about="http://ontology.owl#Restricao2">
  <owl:someValuesFrom rdf:resource="http://ontology.owl#Empresa"/>
  <owl:onProperty rdf:resource="http://ontology.owl#Gera"/>
</owl:Restriction>
```

Figura 40: Declaração de uma agregação

E, por último, Figura 41 mostra o relacionamento do tipo composição, que é caracterizado pela *tag allValuesFrom,* a qual determina a ocorrência de um relacionamento do tipo composição, pois representa um relacionamento mais forte entre o objeto agregador e os objetos componentes, sendo que, a classe (componente) é destruída quando desaparece a composição.

```
<owl:ObjectProperty rdf:about="http://ontology.owl#Originam">
  <rdfs:range rdf:resource="http://ontology.owl#Servicos"/>
  <rdfs:domain rdf:resource="http://ontology.owl#Faturas"/>
</owl:ObjectProperty>
<owl:Restriction rdf:about="http://ontology.owl#Restricao1">
  <owl:allValuesFrom rdf:resource="http://ontology.owl#Servicos"/>
  <owl:onProperty rdf:resource="http://ontology.owl#Originam"/>
</owl:Restriction>
```

Figura 41: Declaração de uma composição

Finalmente, a finalização do arquivo ontológico de requisitos é realizada por *tags* padrão da linguagem XML, RDF e OWL, como é mostrado na Figura 42.

```
</rdf:RDF>
```

Figura 42: Finalização do arquivo

5.3. CAMADA DE INTERFACE

Essa camada é a que tem o objetivo de proporcionar a interação do usuário com a *UStory-Refactory*, e foi desenvolvida através das APIs SWT e *Jena Ontology*, as quais interagem com a linguagem de programação Java. Todavia, ambas têm funcionalidades diferentes, pois a primeira API tem o objetivo de criar a interface da ferramenta em si, e a segunda interage com o arquivo OWL e com a interface da ferramenta.

Assim, a SWT é uma API *open source* para o desenvolvimento de interfaces gráficas desenvolvida pela IBM para a linguagem Java, cuja escolha ocorreu principalmente por apresentar uma melhor performance comparada à tradicional interface SWING. Além disso, a SWT destaca-se por apresentar bibliotecas nativas do sistema operacional, ser mais versátil, ser totalmente portável e ainda por ser um padrão da IDE Eclipse, que foi utilizada no desenvolvimento da ferramenta (SWT, 2006).

5.3.1. Modelo de Caso de Uso

Para o desenvolvimento da camada de interface foi desenvolvido um diagrama de caso de uso da modelagem UML, que especifica em detalhes cada operação que o usuário deve realizar para refatorar requisitos a partir de *User Stories (CRC Cards)*, o que é mostrado na Figura 43.

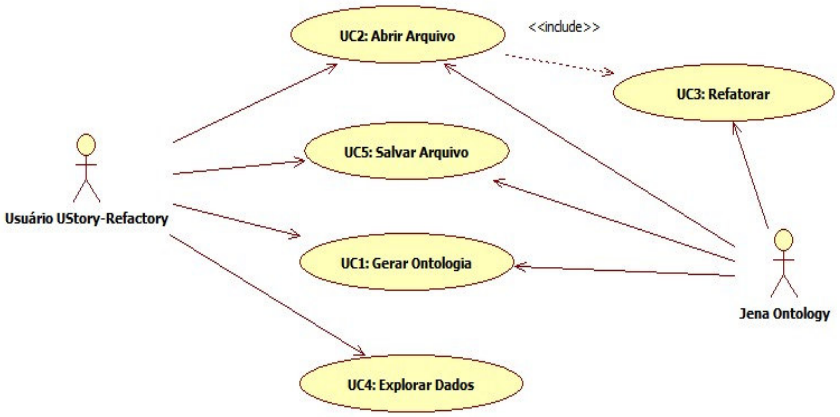


Figura 43: Modelo de Caso de Uso da ferramenta

Além disso, para a descrição do diagrama de casos de uso demonstrado na Figura 43 optou-se pelo modelo casual recomendado por Cockburn (COCKBURN, 2001), sendo que os quadros a seguir descrevem cada um dos cinco casos de uso implementados na ferramenta.

Tabela 5: UC1: Gerar Ontologia

Descrição resumida: permite ao usuário gerar o próprio arquivo de ontologia a ser refatorado. **Fluxo básico:** a. O usuário seleciona a opção gerar ontologia do menu arquivo; b. O sistema exibe um formulário com duas *tabs*: a primeira é para ser cadastrada as classes, os métodos e os atributos e a segunda é para relacionar as classes com seus devidos relacionamentos; c. O usuário interage (incluir e excluir dados) com os formulários e seleciona o botão salvar dados; d. O sistema salva o arquivo no formato OWL e mostra uma mensagem informando que o arquivo foi salvo com sucesso. **Fluxo alternativo:** a. No passo b, se o usuário inserir campos em brancos, o sistema retorna uma mensagem informando o erro; **Pré-condições:** não existem pré-condições **Pós-condições:** gerar um arquivo de ontologia no formato OWL em que se pode ser aplicada as refatorações.

Tabela 6: UC2: Abrir arquivo

Descrição resumida: permite ao usuário abrir um arquivo OWL para ser refatorado **Fluxo básico:** a. O usuário seleciona o botão abrir da interface da ferramenta; b. O sistema abre uma janela para selecionar o arquivo; c. O usuário seleciona o arquivo e pressiona o botão abrir; d. O sistema mostra os dados e habilita o botão refatorar e salvar; **Fluxo Alternativo:** a. No passo a, o usuário abre o arquivo através da opção abrir do menu arquivo; b. No passo c, o usuário pressiona o botão cancelar, e a janela é fechada. **Pré-condições:** ter um arquivo no formato OWL que aplique a especificação UML-MC. **Pós-condições:** o arquivo é interpretado pela *Jena Ontology* e mostrado na interface da ferramenta.

Tabela 7: UC3: Refatorar

Descrição resumida: permite à aplicação refatorar o arquivo OWL.
Fluxo básico:
a. O usuário clica no botão Refatorar;
b. A aplicação refatora os dados, exibe os dados no formulário da interface e mostra uma caixa de mensagem informando que os dados foram refatorados;
Fluxo Alternativo:
a. No passo a, o usuário abre o arquivo através da opção refatorar do menu arquivo;
b. No passo b, a aplicação exibe uma caixa de mensagem informando que nenhuma refatoração pode ser aplicada.
Pré-condições: UC2: Abrir Arquivo
Pós-condições: o arquivo é refatorado.

Tabela 8: UC4: Explorar Dados

Descrição resumida: permite ao usuário explorar os dados do arquivo antes e depois de refatorado.
Fluxo básico:
a. O usuário seleciona a classe dentro do *group box* "*Componentes das classes"*;
b. A aplicação mostra os atributos e métodos da classe;
Pré-condições: UC2: Abrir Arquivo (e/ou UC3: Refatorar)
Pós-condições: o usuário pode ver os dados não refatorados e refatorados de uma forma mais amigável.

Tabela 9: UC5: Salvar Dados

Descrição resumida: permite ao usuário salvar o arquivo
Fluxo básico:
a. O usuário seleciona o botão salvar;
b. O sistema abre uma janela para salvar o arquivo;
c. O usuário informa o nome do arquivo OWL e seleciona o botão salvar;
d. A aplicação salva os dados em um novo arquivo OWL.
Fluxo Alternativo:
a. No passo a, o usuário salva o arquivo através da opção salvar do menu arquivo da aplicação.
b. No passo c, o usuário pressiona o botão cancelar, e a janela é fechada.
Pré-condições: UC3: Refatorar
Pós-condições: os dados são salvos em um arquivo.

Sendo assim, destaca-se neste modelo de caso de uso a interação de dois usuários durante todo o processo de funcionamento da ferramenta: primeiramente o usuário da *UStory-Refactory*, cuja função é executar as operações da ferramenta, e por último a API *Jena Ontology*, que tem o objetivo de manipular os dados e de aplicar a refatoração.

Após a análise dos casos de uso, conclui-se que a interface da *UStory-Refactory* contempla as seguintes funcionalidades:

- O usuário pode criar o arquivo através de um gerador de interoperabilidade gerando a ontologia que é refatorada, o que pode ser visto na seção 5.2 deste capítulo;
- Abrir o arquivo a ser refatorado, refatora-lo e salva-lo em um novo arquivo no formato OWL;
- Visualizar os dados antes e depois de refatorados, sendo que, para isso foram escolhidas duas interessantes maneiras: primeiramente destaca-se a visualização em um formato ontológico, e depois no contexto de diagramas classes da UML, que é visualizado por meio de tabelas da API *SWT*, sendo que a interface principal da ferramenta é visualizada na Figura 44;

Sendo assim, como se pode observar na Figura 44, o usuário pode interagir com a ferramenta através de três botões principais: *"Abrir"*, *"Refatorar"* e *"Salvar"*, sendo que o primeiro tem a função de interpretar e mostrar os dados na interface; o segundo aplica as refatorações; e o último realiza a operação de persistência de dados em um novo arquivo OWL. Portanto, o menu *"Arquivo"*, além de proporcionar as mesmas funções apresentadas nos botões, também apresenta as opções de *"Gerar UML-OWL"* e *"Sair"*, sendo que a primeira é para a criação de um arquivo OWL, e a segunda é para finalizar a aplicação.

Destaca-se ainda que na interface da ferramenta há dois componentes principais, que são dois *group boxes*: à esquerda da figura está o *group box* com os componentes não refatorados, e à direita está o *group box* com os componentes refatorados, sendo que, para que o usuário possa verificar o que foi refatorado é necessária a interação manual dele com a ferramenta. Por exemplo, com os dois *group boxes* (*"Componentes das Classes"* e *"Componentes das Classes Refatoradas"*) o usuário poderá interagir via *click* do mouse, verificando e analisando em que classes foi aplicada a refatoração através da inserção e remoção de atributos e métodos das classes.

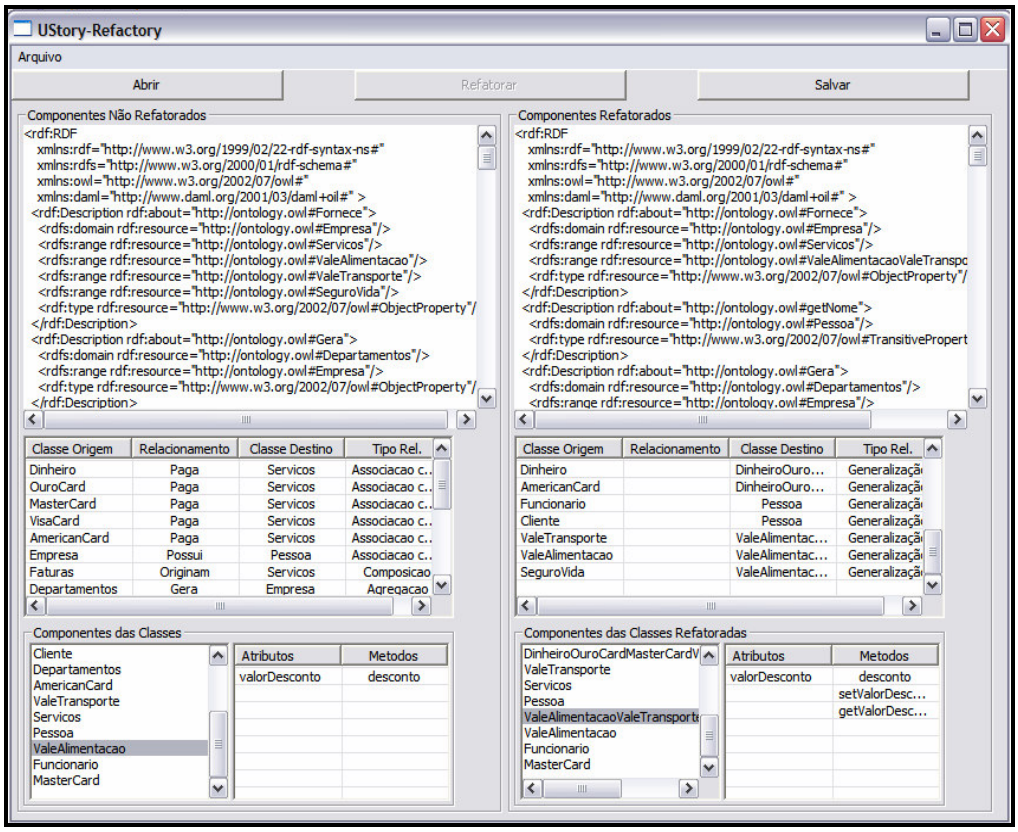


Figura 44: Interface da ferramenta desenvolvida

Para finalizar o processo de refatoração, a interface da ferramenta sempre fornece uma mensagem através de uma caixa de mensagem que pode enviar duas mensagens distintas: *"Refatoração aplicada"* ou *"Nenhuma refatoração foi aplicada"*. Portanto, pela interação com essa caixa de mensagem o usuário pode saber se o processo de refatoração foi aplicado.

5.4. CAMADA DE REGRAS DE NEGÓCIO

Esta camada é considerada o ponto chave dessa dissertação, pois ela envolve a criação do mecanismo de refatoração e de padrões de refatorações que são aplicados pelos mecanismos de refatoração desenvolvido. Sendo assim, tais itens são descritos em detalhes nas próximas subseções dessa camada.

5.4.1. Mecanismo de Refatorações

O mecanismo de refatoração é o ponto chave que determina como ocorre a aplicação da técnica de refatoração de requisitos na ferramenta, e cujo objetivo é aperfeiçoar e melhorar as especificações de requisitos baseados em cartões *User Stories (CRC Cards)*.

Visando esse objetivo, foi necessário determinar componentes que são de fundamental importância para a aplicação da técnica de refatoração no desenvolvimento deste mecanismo, dentre os quais se destacam:

- Ontologia: já descrita na seção 5.2 deste capítulo, ela é de fundamental importância para a aplicação da técnica de refatoração, pois os requisitos estão representados por meio dela, garantindo, assim, a representação dos dados (LUSTOSA, 2004) .

- Propriedades OWL: as propriedades OWL garantem a representação e o formato da camada ontologia de requisitos desenvolvida, e têm o objetivo de proporcionarem dados consistentes e com mesma semântica, para que consiga identificar todos os requisitos (GONÇALVES, 2004).

- Regras: possuem o objetivo de manipular os dados (requisitos) e de identificar as situações em que se pode aplicar a refatoração, sendo que como exemplo de regra destaca-se a análise de todos os relacionamentos da ontologia, que, e verifica se estes possuem as mesmas semânticas.

- Pré-Condições: representa a situação em que se analisa se os dados são refatorados, ou seja, é um conjunto de premissas básicas para que uma refatoração seja aplicada.

- Pós-Condições: é o último componente a ser aplicado, e seu objetivo é aplicar a técnica de refatoração e de preservar o comportamento dos requisitos no formato OWL.

Os componentes do mecanismo de refatoração são analisados na Figura 45 onde é ilustrada a composição do mecanismo de refatoração desenvolvido. Esse mecanismo de refatoração funciona através da interação entre ele e o *Ontology Model* (modelo de ontologia), o qual é criado no momento de leitura dos dados pela instância do método "*ModelFactory.createOntologyModel()*", que, por sua vez, tem o objetivo de ler e armazenar a ontologia na memória do computador para aplicar a refatoração. É importante salientar ainda que o processo de refatoração proporciona um mecanismo de refatoração que interage diretamente com o *Ontology Model*, que é alterado por meio de regras, pré-condições e pós-condições via interação com a memória do computador, e *Ontology Model* é enviado novamente para o mecanismo de refatoração, realizando uma nova interação. Além disso, esse processo é repetido diversas vezes até que não ocorram mais dados para serem refatorados, o

que possibilita, portanto, que esse mecanismo de refatoração dê suporte à aplicação de várias refatorações ao mesmo tempo.

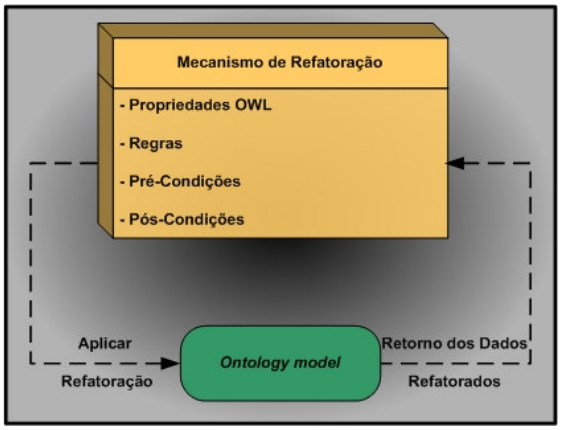


Figura 45: Mecanismo de Refatoração

Finalmente, após a união de todos esses componentes consegue-se proporcionar na ferramenta dados consistentes e eficazes, e, dessa maneira, gerar dados refatorados, proporcionando uma base de dados refatorados com a semântica OWL. A geração e construção da ferramenta são explicadas com detalhes na próxima subseção, cujo objetivo é detalhar a modelagem da ferramenta desenvolvida.

5.4.2. Modelo de Interações

Os modelos apresentados na Figura 46 e Figura 47 mostram o funcionamento da ferramenta de forma seqüencial, sendo que, através deles é possível verificar o processo de visualização dos dados e de refatoração dos dados.

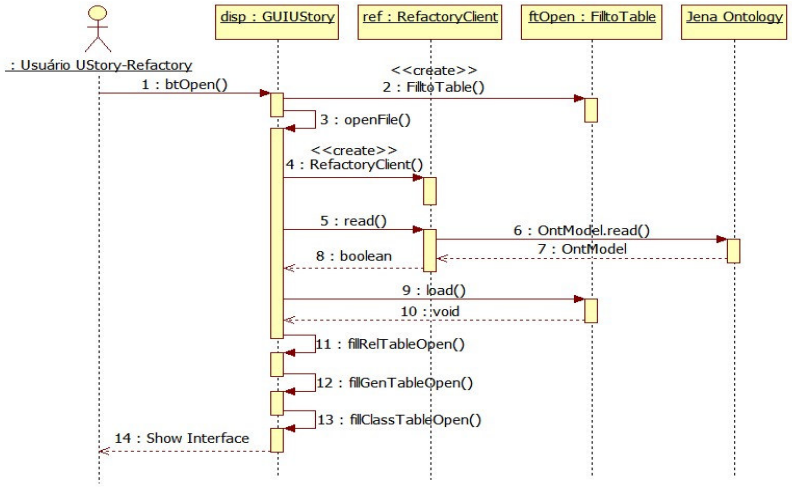


Figura 46: Modelo de interação para visualizar os dados

Como pode-se observar, o modelo de interação da Figura 47 mostra como ocorre o processo de visualização dos dados, o qual pode ser descrito da seguinte maneira: após ocorrer a instanciação da interface da ferramenta, o usuário seleciona o botão abrir e a interpretação dos dados ocorrerá através do método *read()*, interagindo juntamente com a API *Jena Ontology*, que por sua vez armazena os dados na memória, e, finalmente, esses dados são carregados através do método *load()* e mostrados na interface da ferramenta.

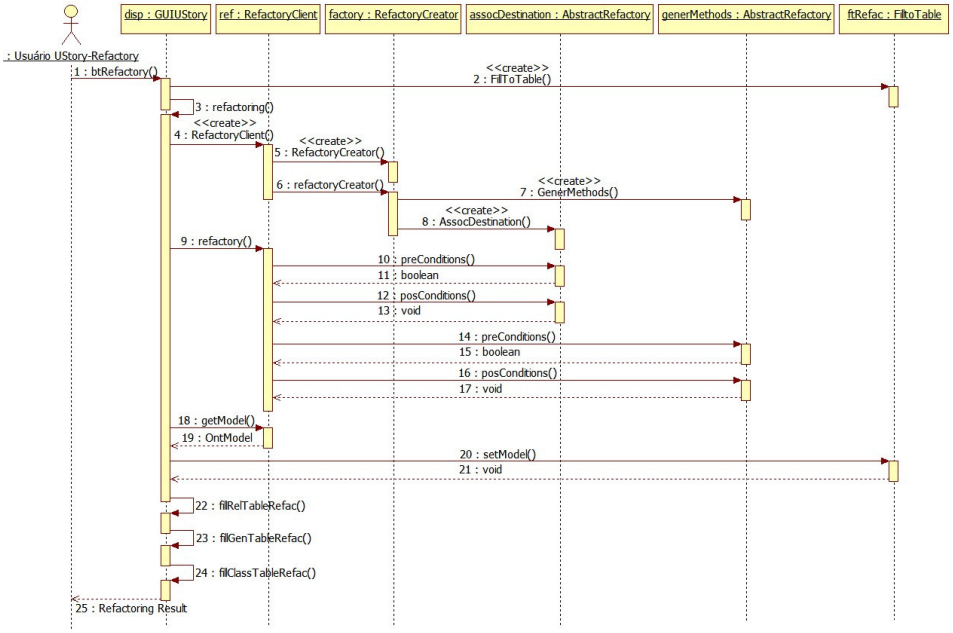


Figura 47: Modelo de interação do processo de refatoração

Sequencialmente, após realizar a visualização dos dados, o usuário dispara o processo de refatoração, de acordo com a Figura 47, o qual interage com a interface clicando no botão "refatorar", o que gera a criação dos objetos necessários que serão refatorados, e, então, aplica-se o processo de refatoração através das pré-condições e das pós-condições. Após aplicar o processo de refatoração, a aplicação recupera o modelo refatorado através do método *getModel()*, enviando-o para o objeto da classe *FilltoTable* através do método *setModel()*. Em seguida, os dados são visualizados na interface da ferramenta e o processo de refatoração é finalizado com uma mensagem de retorno da aplicação informando se a refatoração foi possível de ser aplicada.

Embora haja cinco tipos de refatoração na aplicação, no desenvolvimento do modelo de interação da Figura 47, apenas a execução de dois padrões de refatorações foi mostrada. Logo, a aplicação dos outros três padrões segue o mesmo processo daquele mostrado na

figura apresentada, sendo que, o motivo porque somente esse dois modelos de interações são mostrados neste trabalho justifica-se pelo fato de os mesmos serem os mais complexos.

5.4.3. Modelo de Classes

A Figura 48 mostra o modelo de classe da *UStory-Refactory*.

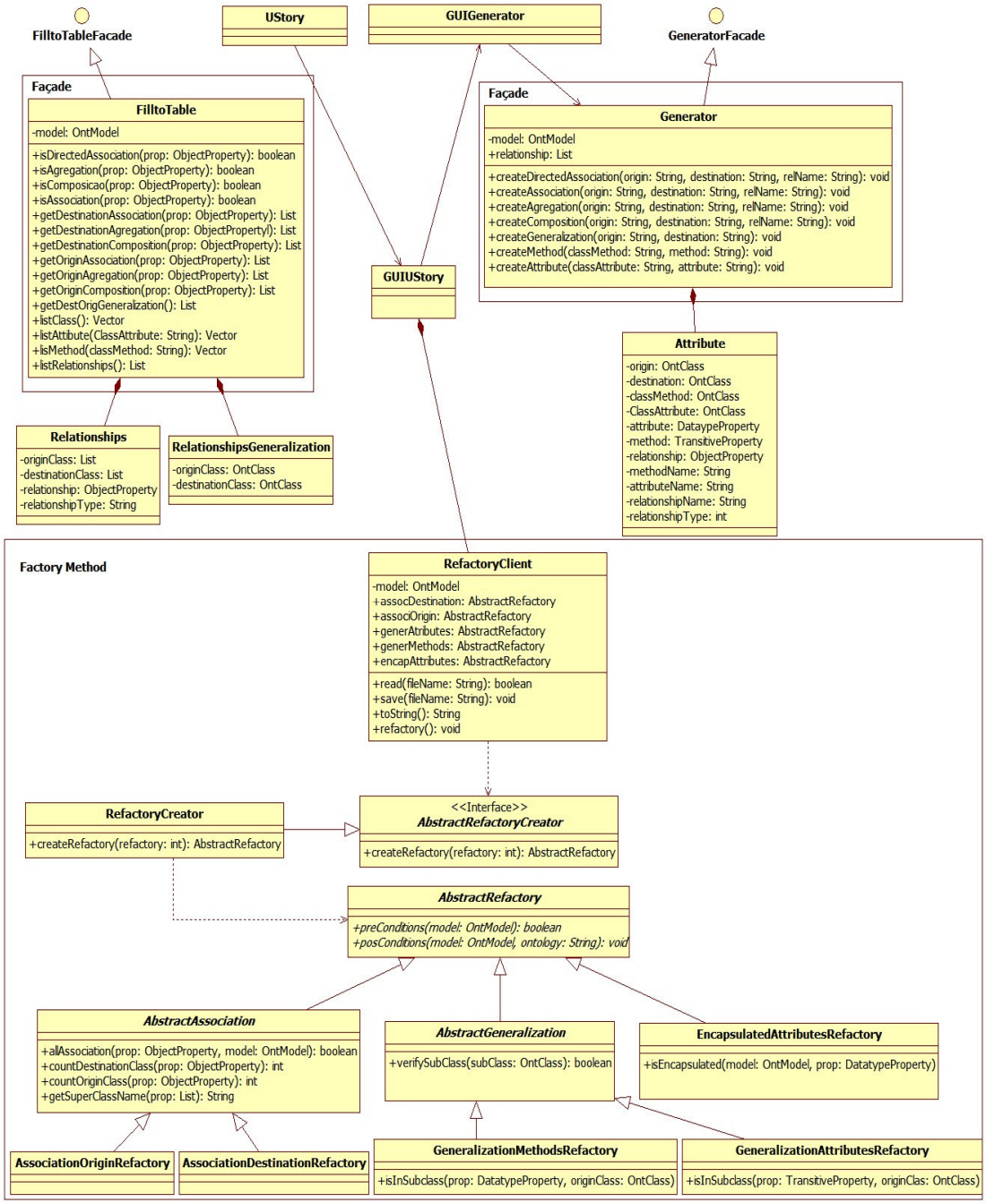


Figura 48: Modelo de Classes da *UStory-Refactory*

Cabe destacar nesse diagrama de classes que a classe *RefactoryClient* é responsável por realizar a persistência das refatorações. Salienta-se ainda a aplicação de dois padrões de projeto: o primeiro padrão aplicado é estrutural e chamado *Façade* (GAMMA, 2000), o qual tem o objetivo de fornecer uma interface mais simples em um subsistema, como é no caso das classes *FilltoTable* e *Generator*. Além disso, a primeira classe *Façade* (*FilltoTable*) é utilizada para fazer a comunicação com a API SWT e para preencher a GUI da aplicação representada pela classe *GUIUStory*. Em contraste, a segunda classe é utilizada para realizar a abstração da API *Jena Ontology* e criar os elementos da ontologia, sendo que a principal vantagem da utilização desse padrão é a redução do número de objetos com que os clientes têm que lidar, tornando o subsistema mais fácil de usar.

Já o segundo padrão de projeto utilizado no desenvolvimento da ferramenta é um padrão de criação denominado *Factory Method* (GAMMA, 2000), cujo o objetivo é facilitar a criação de objetos complexos. Assim, as refatorações são criadas através desse padrão, buscando abstrair a criação dos objetos das refatorações, sendo que a classe *RefactoryCreator* (Figura 48) tem o objetivo de instanciar um objeto do tipo *AbstractRefactory*. Dessa forma, a complexidade de criação dos objetos de refatoração não é responsabilidade da classe *RefactoryClient* (a qual aplica uma refatoração), pois ela somente deve conhecer a interface de tais objetos.

Na próxima subseção, será explicado com mais detalhes como o *Factory Method* é usado nessa aplicação, através da demonstração de um mecanismo de extensão para refatorações.

5.4.4. Mecanismo de Extensão de Refatorações

Para que seja reutilizável no futuro, a modelagem da *UStory-Refactory* segue conceitos de projetos de *software* orientado a objetos (GAMMA, 2000). Assim, com esse objetivo o mecanismo de refatoração da ferramenta foi desenvolvido através da adoção de premissas básicas da técnica de refatoração e de padrões de projeto, sendo que, através dessas duas premissas seguidas se dá origem a um *"Mecanismo de Extensão de Refatorações",* cujo objetivo é permitir que qualquer outra refatoração possa ser adicionada à ferramenta, através da inserção de alguns componentes, não alterando, dessa maneira, toda a aplicação. O modelo do mecanismo de extensão de refatorações é apresentado na Figura 49.

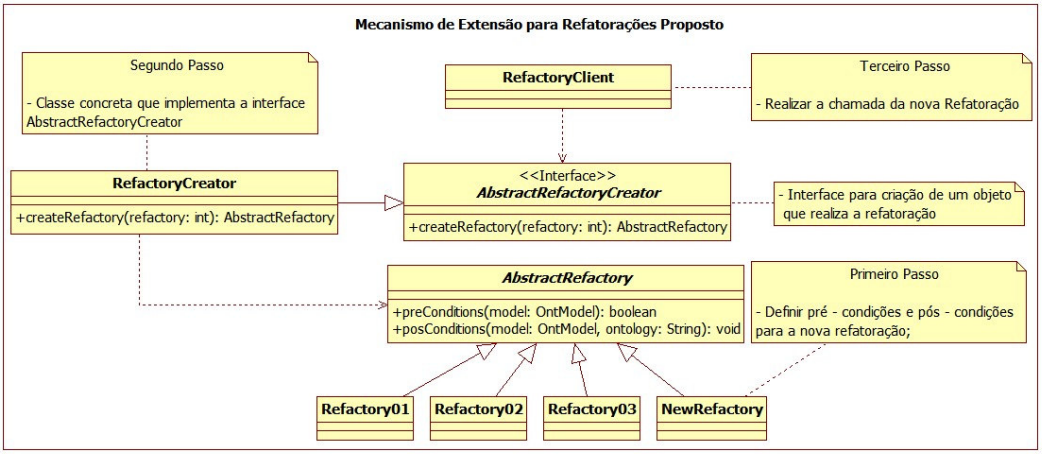


Figura 49: Mecanismo de extensão de refatorações

Para a criação do mecanismo de extensão de refatorações foi aplicado o padrão de projeto *Factory Method*, visto na Figura 49, que abstrai a criação dos objetos responsáveis pela refatoração propriamente dita. Além disso, esse padrão facilita a inclusão de novas classes de refatoração exigindo o mínimo de re-trabalho para o uso das novas classes, visto que temos uma classe especializada na criação dos objetos de refatoração.

A seguir será descrito os passos necessários para uma nova classe de refatoração:

1º passo: criar uma nova subclasse que estenda a classe abstrata *AbstractRefactory*, definindo as pré-condições e pós-condições de aplicação da nova refatoração;

2º passo: alterar a classe concreta *RefactoryCreator* para suportar a nova classe de refatoração;

3º passo: realizar a chamada da nova refatoração na classe *RefactoryClient;*

Portanto, com a aplicação dos passos acima, é possível notar a facilidade de extensão da ferramenta de refatoração desenvolvida.

5.4.5. Técnica de Refatoração aplicado em cartões *User Stories (CRC Cards)*

A técnica de refatoração de requisitos é aplicada em especificações de requisitos baseados em cartões *User Stories (CRC Cards)*, nos quais são aplicadas premissas desenvolvidas e analisadas por Roberts (ROBERTS, 1999), Mens (MENS, 2004) e Xu (XU, J., 2004). Tais premissas são aplicadas diretamente no código fonte, uma vez que a refatoração é

desenvolvida, visando ao melhoramento da codificação de um sistema já existente ou que está em desenvolvimento (FOWLER, 2004). Portanto, foi realizada a extensão dessa técnica de refatoração de código, para que ela fosse aplicada em requisitos no formato de *User Stories (CRC Cards)*. Sendo assim a ferramenta contempla a abordagem do conceito de refatoração por meio de regras que proporcionam a melhoria nos dados, buscando acabar com as redundâncias (dados duplicados) dos requisitos.

Definiram-se então cinco etapas fundamentais para a aplicação da refatoração, destaca-se:

1. **Análise dos requisitos e identificação do que deverá ser refatorado:** primeiramente deve ser analisada a base de requisitos para ver se os dados apresentam inconsistências de acordo com o conceito de refatoração a ser aplicado, definindo o local onde será aplicada a refatoração de requisitos e verificando-se o nível de abstração que se quer conseguir nas refatorações dos requisitos;

2. **Garantia da preservação do comportamento:** o próximo passo é garantir que as refatorações preservarão o comportamento do sistema, sendo que essa definição é aplicada ao contexto em que os valores de entrada e de saída devem ser os mesmos. Assim, na técnica de refatoração de requisitos será introduzido o conceito de pré-condições na preservação de comportamento de uma refatoração. Portanto, um requisito poderá ser refatorado, mas o seu objetivo final deve ser o mesmo de antes, embora o processo para chegar ao resultado final seja diferente, ou seja, nesse processo são aplicadas melhorias ao requisito, mostrando-o de uma maneira diferente da que ele apresentava antes de ser refatorado.

3. **Aplicar a refatoração:** a aplicação da refatoração aos requisitos será através da identificação do problema, e quem realiza esse processo é a ferramenta desenvolvida;

4. **Verificação da preservação do comportamento:** para a preservação do comportamento serão adotadas as pós-condições nas refatorações de requisitos, sendo que essa é uma verificação de que todas as operações nas refatorações foram garantidas e realizadas com sucesso, sendo então descritas após a aplicação de cada refatoração;

5. **Manter a consistência entre as refatorações:** essa é a última etapa, e será um momento de validação da refatoração. Assim, além das condições impostas como pré-condições e pós-condições, ela será validada e analisada por uma pessoa altamente capacitada na área de refatoração.

Portanto, pode-se notar que a aplicação da técnica de refatoração nos requisitos requer etapas que devem ser seguidas cuidadosamente, conforme descrito anteriormente. No entanto, na Figura 50 é mostrado um diagrama de atividades da modelagem UML, que tem o objetivo de mostrar as etapas da refatoração por meio de ações, buscando refinar o ciclo de vida do processo de refatoração aplicado pela *UStory-Refactory*.

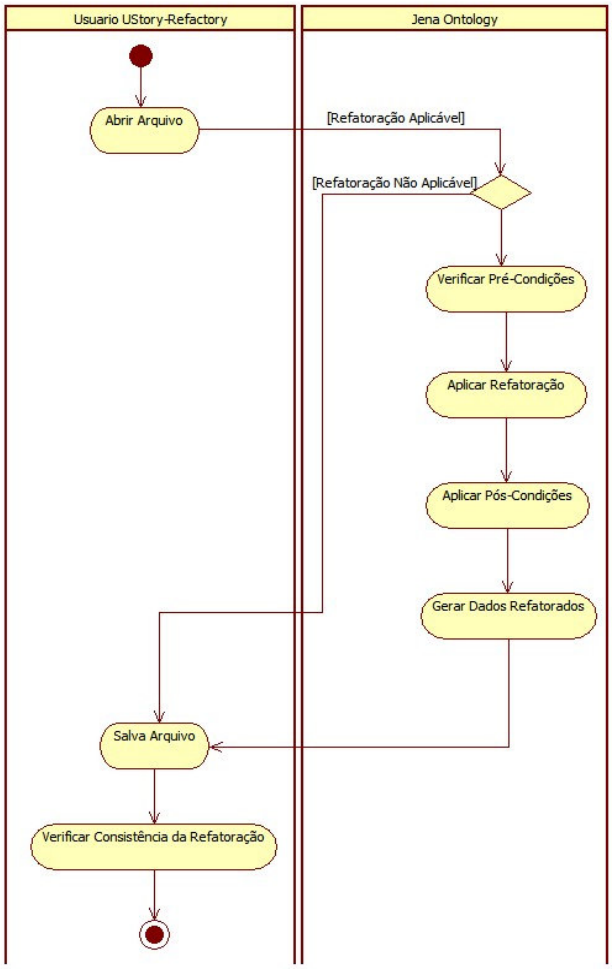


Figura 50: Diagrama de Atividades que demonstra a aplicação da técnica de Refatoração

Sendo assim, nesse diagrama de atividades, observa-se que há dois atores principais: o usuário da *UStory-Refactory* e o *Jena Ontology*, cada qual com suas devidas ações, uma vez

que o primeiro tem a ação de iniciar e finalizar a refatoração, e o segundo tem a ação de aplicar a refatoração nos requisitos de acordo com os padrões de refatorações que a ferramenta aplicada; sendo que na próxima subseção serão explicados esses padrões com mais detalhes.

5.4.6. Padrões de Refatorações aplicado em cartões *User Stories (CRC Cards)*

As refatorações aplicadas em requisitos do tipo *User Stories (CRC Cards)* têm o objetivo de aperfeiçoar os requisitos formalizados no formato UML-MC (ROBINSON, 2004), sendo que tal aperfeiçoamento segue a formalização de diagramas de classes da UML, nos quais são aplicadas regras, que dão origem ao que chamamos de padrões de refatorações aplicadas nas especificações de requisitos cartões *User Stories (CRC Cards)*.

Para descrever e analisar cada padrão de refatoração desenvolvido são seguidas as descrições de refatorações em especificações de requisitos baseados em casos de uso, que são proposto por Yu (YU, 2004) e Xu (XU, J., 2004). Logo, são definidos quatro itens que têm o objetivo de apresentar e explicar cada padrão de refatoração nas especificações de requisitos baseados em *User Stories (CRC Cards)*, os quais são:

- Nome da Refatoração: nome que é dado ao padrão de refatoração;
- Descrição: explicação para a aplicação do padrão de refatoração;
- Estrutura: *design* esperado para aplicar a refatoração.
- Pré-Condições: situações que são verificadas para se aplicar o padrão de refatoração;
- Pós-Condições: resultados após a aplicação do padrão de refatoração.

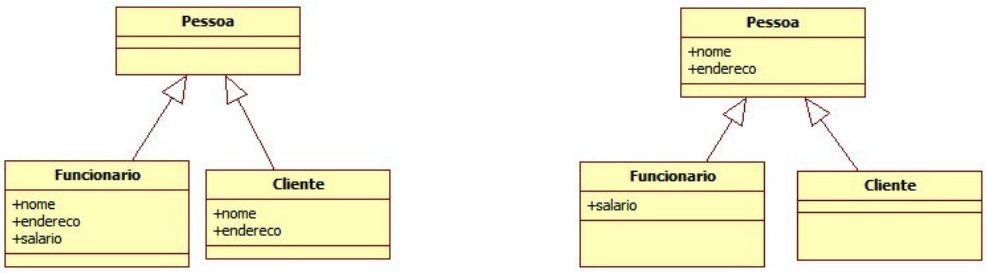
Assim, como já foi visto anteriormente, a refatoração tem o objetivo de aperfeiçoar e melhorar o código fonte, sendo que nos padrões de refatorações desenvolvidos foi procurado aplicar a refatoração no nível de análise de projetos seguindo a UML-MC (ROBINSON, 2004), ou seja, a refatoração é aplicada antes de se ter o código fonte da aplicação. Destaca-se ainda que para o desenvolvimento desses padrões de refatorações é fundamental identificar cada propriedade OWL que caracteriza os elementos da UML-MC (ROBINSON, 2004).

A seguir são descritos quatro padrões de refatoração que a metodologia e a ferramenta contemplam:

1. Subir Campo na Hierarquia

- **Descrição:** se um diagrama de classe apresentar generalização, e nas subclasses houver o mesmo atributo, este será movido (refatorado) para a superclasse, porque dessa forma através da especialização as subclasses vão continuar tendo acesso ao atributo refatorado.

- **Estrutura:** na Figura 51 (a), é mostrado o *design* esperado para que se possa aplicar esse padrão de refatoração, e na Figura 51 (b), o respectivo resultado da refatoração. Pode-se notar assim que os campos em comum entre as classes são movidos para a superclasse, denotando que eles serão herdados pelas subclasses. Destaca-se ainda na Figura 51 (b), que o atributo *"salário"* não é refatorado porque está presente em apenas uma subclasse. A seguir são descritas as etapas de aplicação desse padrão por meio das pré-condições e das pós-condições das especificações de requisitos:



(a) Não refatorado **(b) Refatorado**

Figura 51: Subir campo na hierarquia

- **Pré-Condições:**

  - Analisar todos os componentes de uma *User-Story* formalizada de acordo com a UML-MC, todos os componentes devem estar interligados;

  - Verificar os tipos de relacionamentos e nas subclasses deve ocorrer atributos duplicados;

  - Os atributos duplicados deverão estar apenas nas subclasses da generalização em que se vai aplicar refatoração;

- **Pós-Condições:**

  – Criar um novo campo com as mesmas características do campo duplicado na superclasse;

  – Apagar o atributo duplicado das subclasses e todas as suas referências sobre esse campo;

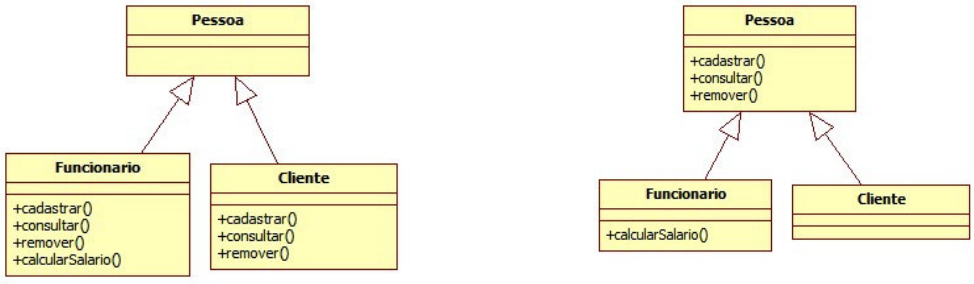
  – Não deverão existir referências ao campo refatorado;

  – Para finalizar, aplicar, se necessário, o padrão *Auto-Encapsular Campo*, que logo será apresentado.

2. Subir Método na Hierarquia

- **Descrição:** métodos repetidos em diversas subclasses também são refatorados semelhantemente ao primeiro padrão apresentado em linhas anteriores.

- **Estrutura:** na Figura 52 (a), é mostrado o *design* de entrada da refatoração, e, em seguida, na Figura 52 (b), é mostrado o resultado da refatoração. Nota-se que o método "*calcularSalario()"* não é refatorado porque está em apenas uma subclasse, e também verifica-se que esse padrão segue a mesma idéia que o anterior, só que desta vez são refatorados os métodos das classes. A seguir as etapas desse padrão são descritas:



(a) Não refatorado **(b) Refatorado**

Figura 52: Subir método na hierarquia

- **Pré-Condições:**

  - Os componentes de uma *User-Story* devem estar todos interligados de acordo com a formalização UML-MC;

  - Verificar se ocorrem relacionamentos do tipo generalização e se nas subclasses ocorrem métodos iguais;

  - Os métodos iguais devem estar apenas nas subclasses em que se vai aplicar a refatoração;

- **Pós-Condições:**

  - Criar um novo método na superclasse com as mesmas características do método duplicado;

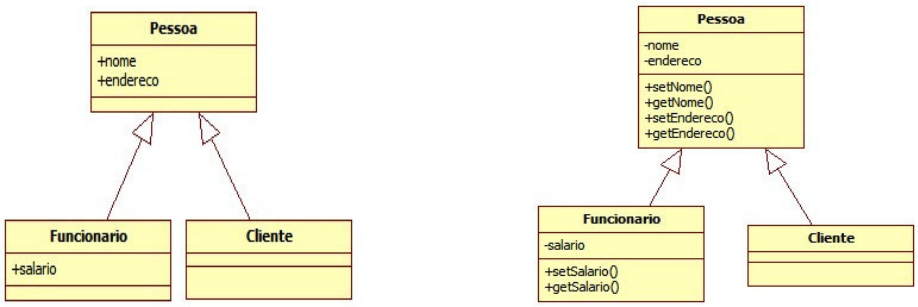
  - Apagar o método duplicado das subclasses, assim como também todas as suas referências;

  - Não deverão existir referências ao método refatorado;

3. Auto-Encapsular Campo

- **Descrição:** no diagrama de classes quando houver atributos públicos nas classes, os métodos de acesso *get* e *set* são criados automaticamente e o atributo é transformado em privado. Isso resolve, então, o problema de interferência externa sobre os dados de um objeto.

- **Estrutura:** na Figura 53 (a), é mostrado o *design* em que se aplica a refatoração, e, na Figura 53 (b), é mostrado o resultado da refatoração. Assim, verifica-se nesta figura que para cada atributo da classe são criados os métodos de acesso. Logo, as etapas de aplicação desse padrão são descritas do seguinte modo:



(a) Não refatorado **(b) Refatorado**

Figura 53: Auto-encapsular campo

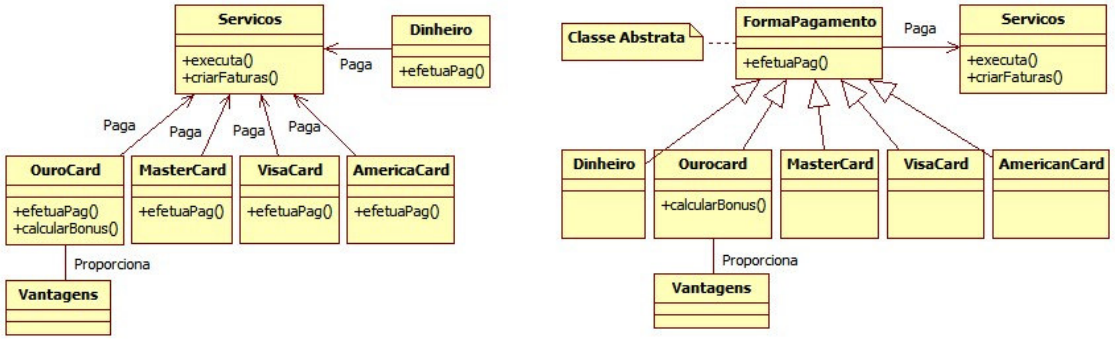
- **Pré-Condições:**
  - Analisar classes de uma *User-Story* formalizada de acordo com a UML-MC;
  - Se necessário, aplicar o padrão de refatoração *Subir Campo na Hierarquia,* para se ter a condição de aplicar este padrão proposto;
  - Verificar se as classes possuem atributos sem os métodos de acesso *get* e *set*;
- **Pós-Condições:**
  - Tornar os atributos privados e criar os métodos que acessam este atributo;
  - Os métodos devem ter os seguintes nomes: "*set<nome do Atributo>*" e "*get<nome do Atributo>*", sendo que a primeira letra do atributo deve ser maiúscula seguindo assim padrões de melhores práticas de programação;
  - Verificar se todos os atributos das classes possuem os métodos de acesso.

4. Criar Classe Abstrata

- **Descrição:** a redundância é um erro comum que ocorre no momento da análise do *software*, no padrão desenvolvido, quando ocorrer relacionamentos do tipo associação com direção e com papéis iguais entre as classes, e se nelas houver métodos iguais, pode-se aplicar esse padrão de refatoração. Assim, a eliminação da redundância se dá através da criação de uma classe abstrata (superclasse) que implementa os métodos refatorados e, portanto, as subclasses tornam-se filhas da nova superclasse. Além

disso, esse padrão de refatoração apresenta uma formalização adequada sendo descrito por meio de dois casos que apresentam a mesma estrutura, os quais serão descritos a seguir.

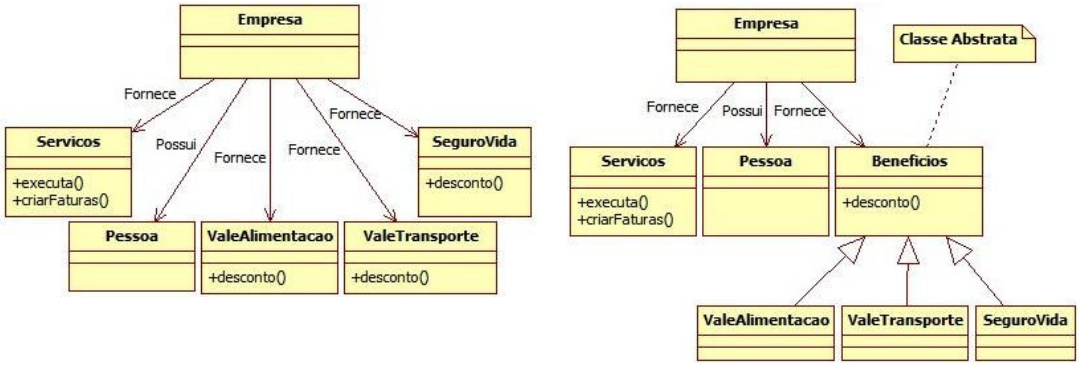
- **Estrutura:**
  - **Primeiro Caso - Relacionamentos iguais com classes origens diferentes:** para a ocorrência desse caso, devem-se ter duas ou mais classes origens relacionadas com uma classe destino, que devem estar ligadas por meio de associação com direção de mesmo papel. Além disso, as classes origens devem ter a ocorrência de métodos duplicados, conforme visto na Figura 54 (a), onde nota-se que as classes *"OuroCard"*, *"MasterCard"*, *"VisaCard"*, "*AmericanCard"* e *"Dinheiro"* instanciam a classe "*Serviços", e* apresentam relacionamentos e papéis iguais. Verifica-se ainda a presença do método *"efetuaPag()"* em todas classes origens, caracterizando duplicação de código. Sendo assim, no modelo refatorado, Figura 54 (b), é criada uma classe abstrata (*"FormaPagamento"*) que será extendida pelas classes que possuiam métodos duplicados. Finalmente, após a refatoracao obtense uma classe abstrata que implementa o método *"efetuaPag()"* e instancia a classe serviço, e suas subclasses herdam tal código. Destaca-se no modelo refatorado a não refatoração do método *"calcularBonus()"*,uma vez que ele não está presente em todas as classes origens.



(a) Não refatorado **(b) Refatorado**

Figura 54: Primeiro caso do padrão criar classe abstrata

- **Segundo Caso - Relacionamentos iguais com uma classe origem:** nota-se na Figura 55 que esse caso é semelhante ao apresentado anteriormente, no entanto, a diferença é que agora ocorrerá uma classe origem relacionada com diversas classes destinos, as quais são ligadas por meio de associação com direção e têm mesmo papel. Por sua vez, as classes destinos devem ter a ocorrência de métodos duplicados, conforme visto na Figura 55 (a), onde é mostrado o *design* em que se aplica este padrão; já na Figura 55 (b) é mostrado o resultado da refatoração, destacando-se que agora o método *"desconto()"* será implementado apenas uma vez pela classe abstrata(*"Benefícios"*), sendo que e as subclasses estendem-na.



(a) Não refatorado **(b) Refatorado**

Figura 55: Segundo caso do padrão criar classe abstrata

- **Pré-Condições:**

  – Verificar se todos os componentes de uma *User-Story* estão interligados de acordo com a UML-MC;

  – Analisar os tipos de relacionamentos, e verificar os papéis;

  – Identificar no relacionamento a direção da associação direcionada, para definir a classe origem e a destino;

  – Verificar se as classes origens ou destinos, dependendo do caso que se aplica, apresentam métodos duplicados.

- **Pós-Condições:**

- Criar uma nova classe abstrata;

- Inserir relacionamentos do tipo generalização entre a classe abstrata e as classes refatoradas;

- Os métodos duplicados são movidos para a classe abstrata;

- A classe abstrata mantém o mesmo relacionamento e o mesmo papel com a classe que não foi refatorada;

- Se necessário, aplicar a refatoração *Subir Campo na Hierarquia* e *Auto-Encapsular Campos*.

Além do mais, no padrão de refatoração apresentado, destaca-se a criação das classes abstratas, mas, por sua vez, na ferramenta desenvolvida essa criação ocorre através da concatenação das classes refatoradas (Figura 44). Assim, após a realização das refatorações o usuário deve editar o arquivo refatorado informando um nome que seja mais intuitivo e fácil de compreender, sendo que pode ser analisado na Figura 55 (b) que o resultado da classe abstrata pela ferramenta é "*ValeAlimentacaoValeTransporteSeguroVida"* e que pela interação do usuário é "*Benefícios"*.

Ë importante destacar que existem outros tipos de refatorações a nível de design propostas por Fowler (FOWLER, 2004) e Opdyke (OPDYKE, 1992) que poderiam ter sido abordadas nessa dissertação, mas estas não foram abordadas porque para a adoção destas, a ontologia desenvolvida deveria representar uma maior variedade de componentes da UML. Desta maneira foi necessário delimitar um escopo, sendo abordado apenas cinco tipos de refatorações devido ao tempo para o desenvolvimento deste trabalho. As regras de refatoração apresentadas nesta seção garantem a integridade e consistência por meio da interação de um analista de sistema sênior que tenha os conhecimentos citados no início da seção 5.

Desta maneira as regras de refatoração de requisitos apresentadas nesta seção se diferenciam das refatorações de código, pois as aplicadas nos requisitos buscam uma melhoria na representação do design e por meio delas é possível identificar situações inadequadas antes do desenvolvimento do código fonte, proporcionando assim que software tenha uma etapa de desenvolvimento mais eficiente.

5.4.7. Execução das Refatorações

Com o propósito de apresentar as execuções de todos os padrões de refatorações, é apresentada na Figura 27 a *User Story (CRC Card)* que origina o diagrama UML-MC apresentado na Figura 29. Verifica-se assim que o diagrama formalizado apresenta diversas redundâncias como métodos e atributos duplicados, relacionamentos com semântica e papéis iguais e atributos sem encapsulamento, sendo que tais redundâncias geram muitas inconsistências no *software*, deixando o código ilegível e complexo de entender.

Portanto, aplicando a ferramenta nessa *User Story (CRC Card)* é possível construir e representar a ontologia à qual será aplicada a refatoração, então, ela é depois interpretada e mostrada corretamente na interface da ferramenta. Além disso, usando-se o botão *"refatorar"*, são executados todos os padrões de refatorações, por meio da *UStory-Refactory*, desenvolvidos, obtendo-se, assim, um diagrama UML-MC com um *design* aperfeiçoado, assim como também ganha-se no momento da implementação reusabilidade e abstração para classificar as entidades, o que pode ser visualizado na Figura 56. É também importante dizer que esse diagrama é gerado através da interação humana e por meio da ontologia OWL refatorada pela ferramenta.

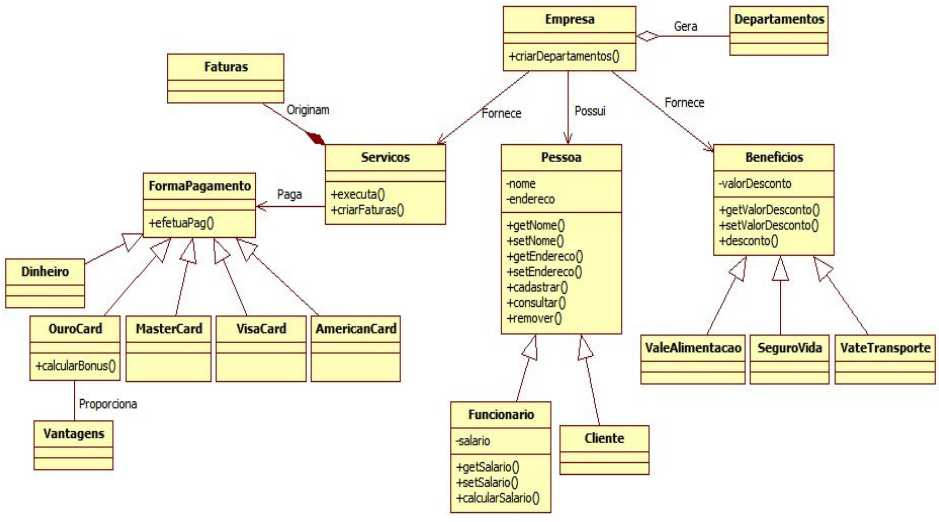


Figura 56: Diagrama de classes refatorado

Já nesse novo diagrama UML-MC, é possível analisar que os padrões de refatorações são aplicados várias vezes. Como exemplo disso há o padrão de refatoração *Auto-Encapsular Campo*, o qual é aplicado várias vezes, até que todos os atributos das classes estejam de

acordo com o resultado esperado. Têm-se também exemplos de não aplicação da refatoração, destacam-se assim classes como *"Departamentos"*, *"Faturas"* e *"Vantagens"*, uma vez que elas não estão adequadas às regras de refatorações desenvolvidas. Pode-se ainda verificar não ocorrência da refatoração nos métodos *"calcularBonus()"* e *"calcularSalario()"*, porque estes são operações próprias das suas subclasses. Além disso, nota-se também que o diagrama ainda apresenta relacionamentos de associação com direção com papéis iguais, embora nesse não seja aplicado porque, além de ter esta condição, é necessário haver métodos duplicados nas classes destinos.

Portanto, após a execução das refatorações por meio desse exemplo, é provado que nem tudo é possível refatorar em um diagrama de classes UML-MC, uma vez que existem casos em que a refatoração não pode ser aplicada. Sendo assim, busca-se aplicar a refatoração em casos complexos de análise, conforme é mostrado nos padrões de refatorações desenvolvidos.

5.5. CAMADA DE PERSISTÊNCIA

Garantir a persistência de dados é um problema de diversas aplicações (FONSECA, 2000), , mas, no caso da *UStory-Refactory*, a persistência se torna uma vantagem, pois a ferramenta desenvolvida atua diretamente com a API *Jena Ontology,* que formaliza os dados em uma ontologia no formato XML+RDF+OWL, a qual garante a semântica dos dados. Assim na Figura 57 é ilustrada o processo de persistência de dados da ferramenta.

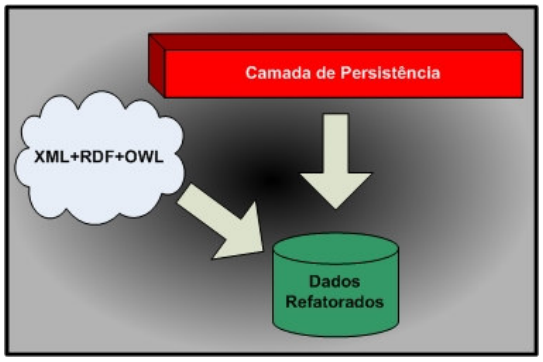


Figura 57: Persistência da ferramenta

Além do mais, a ferramenta garante a persistência dos dados através do desenvolvimento e interação de dois métodos: *read()* e *save()*, os quais serão descritos a seguir. Primeiramente, por um método criado denominado *read*, cujo objetivo é realizar a leitura de um arquivo OWL, na qual se destaca a criação de um modelo de ontologia através

da utilização do método *getModel()*, linha 59 da Figura 58, o qual retorna os dados da ontologia criada em que será aplicado o processo de refatoração. O código fonte em Java é mostrado a seguir na Figura 58.

```
53      /*
54       * Método que realiza a leitura do arquivo
55       */
56      public boolean read(String fileName) {
57          try {
58              if (fileName != null) {
59                  getModel().read(
60                          new InputStreamReader(new FileInputStream(fileName)),
61                          "");
62                  return true;
63              } else
64                  return false;
65          } catch (Exception e) {
66              return false;
67          }
68      }
```

Figura 58: Código para leitura da Ontologia

O segundo método criado é chamado *save,* o qual propõe o conceito de persistência, através da criação de um novo arquivo no formato OWL, sendo que a persistência é garantida pela interface *OntModel* da API *Jena Ontology,* que implementa um método *write()* o qual realiza a escrita do arquivo refatorado no formato OWL através do método *getModel()* que, por sua vez, retorna a ontologia refatorada, o que pode ser visualizado na linha 79 da Figura 59, onde é possível visualizar também o código fonte em Java.

```
74      /**
75       * Método que salva o arquivo refatorado
76       */
77      public void save(String fileName) {
78          try {
79              getModel().write(new PrintWriter(new FileOutputStream(fileName)),
80                      "RDF/XML-ABBREV");
81          } catch (Exception e) {
82          }
83      }
```

Figura 59: Código para gravar Ontologia refatorada

Sendo assim, a geração de um novo arquivo no formato OWL como elemento fundamental da camada de persistência garante a semântica dos dados, e destaca-se também que este novo arquivo tem um tamanho extremamente pequeno, facilitando,, assim no futuro, a adoção de tecnologias como *web services* (COYLE, 2002), para realizar a exportação dos dados refatorados para uma nova aplicação.

6. ESTUDO DE CASO

Com o objetivo de avaliar e verificar a utilização dos padrões de refatorações aplicados em cartões *User Stories (CRC Cards)*, é desenvolvido um estudo de caso que visa apresentar como esses padrões devem ser aplicados. Assim, buscou-se apresentar um estudo de caso que apresentasse situações em que ocorresse a necessidade de refatoração utilizando os padrões de refatorações propostas nesta dissertação.

O estudo de caso escolhido é do *"Sistema de ajuda do ambiente AulaNet"* proposta por Robinson (ROBINSON, 2004), que utiliza a UML-MC como ferramenta de modelagem de mapas conceituais em UML. O ambiente *"AulaNet"* é um sistema que disponibiliza serviços e recursos para professores e alunos de uma universidade . Este estudo de caso apresenta três cartões *User Stories (CRC Cards)* que modelam uma parte deste ambiente. A refatoração será aplicada, se pertinente, em cada cartão que representa partes da modelagem dos requisitos do *"AulaNet"*.

São considerados neste estudo de caso três *User Stories.* A seguir são apresentados os cartões:

- Criação de módulos para *"Professor"*, *"Administrator"* e *"Aluno"*;
- Criação do módulo *"Serviço"*;
- Criação do módulo para utilização de *"Software"*, *"Hardware"* e *"Rede"*;

A seguir, será descrito como ocorre o processo de refatoração.

- **Primeira *User Story (CRC Card)* modelada**

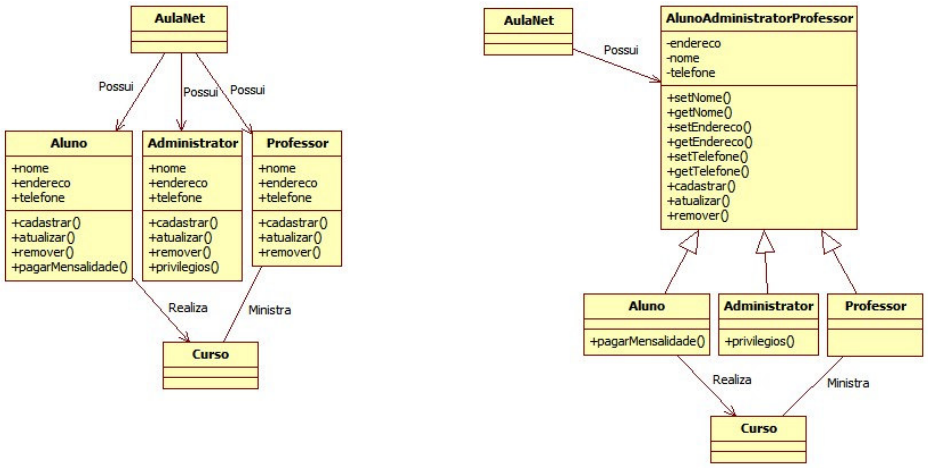
O primeiro cartão apresentado nesse estudo de caso refere-se a criação de um módulo que abrange o *"Administrator"*, o *"Professor"*, e o *"Aluno"* do ambiente. A Figura 60 mostra a descrição do requisito.

Módulo para Professor, Aluno e Administrator

"O ambiente AulaNet possui usuários Professor, Administrator e Aluno. Os campos como nome, endereço e telefone estão presentes em todos esses usuários, assim como também as operações cadastrar, atualizar e remover. Destacam-se operações que não são comuns em todos os usuários, por exemplo: a operação pagar mensalidade está presente somente em Aluno, e a operação privilégios está presente apenas no Administrador. Já o Professor pode ministrar um curso, que por sua vez é realizado por um conjunto de alunos."

Figura 60: Primeira *User Story (CRC Card)* Modelada

Após a descrição do cartão, ele é formalizado para a UML-MC, formando os requisitos em diagramas de classes da UML, que são apresentados na Figura 61 (a), nos quais são aplicados os padrões de refatorações de acordo com a necessidade encontrada. Sendo assim, analisando o diagrama de classes é verificada a ocorrência de relacionamentos do tipo associação com direção com mesmo papel, métodos e atributos duplicados. Portanto verificando o padrão de refatoração C*riar Classe Abstrata* conclui-se que é possível aplicar o segundo caso deste padrão, e após o seu uso pode-se aplicar a refatoração *Subir Campo na Hierarquia* e o *Auto-Encapsular Campo*. Aplicando-se a técnica e a ferramenta de refatoração desenvolvida, obtém-se a criação de uma nova classe abstrata, que é a superclasse das classes em que haviam métodos duplicados, os quais são movidos para a superclasse. Este diagrama refatorado é visualizado na Figura 61 (b). Pode-se verificar a criação da classe abstrata através da concatenação das classes refatoradas chamando-se *"AlunoAdministratorProfessor"*. Já na Figura 62, é apresentada a interface da ferramenta com os dados antes e depois de ser refatorada, verificando-se a criação da classe abstrata com os métodos e atributos duplicados.



(a) Não refatorado **(b) Refatorado**

Figura 61: Primeiro cartão: aplicando as refatorações

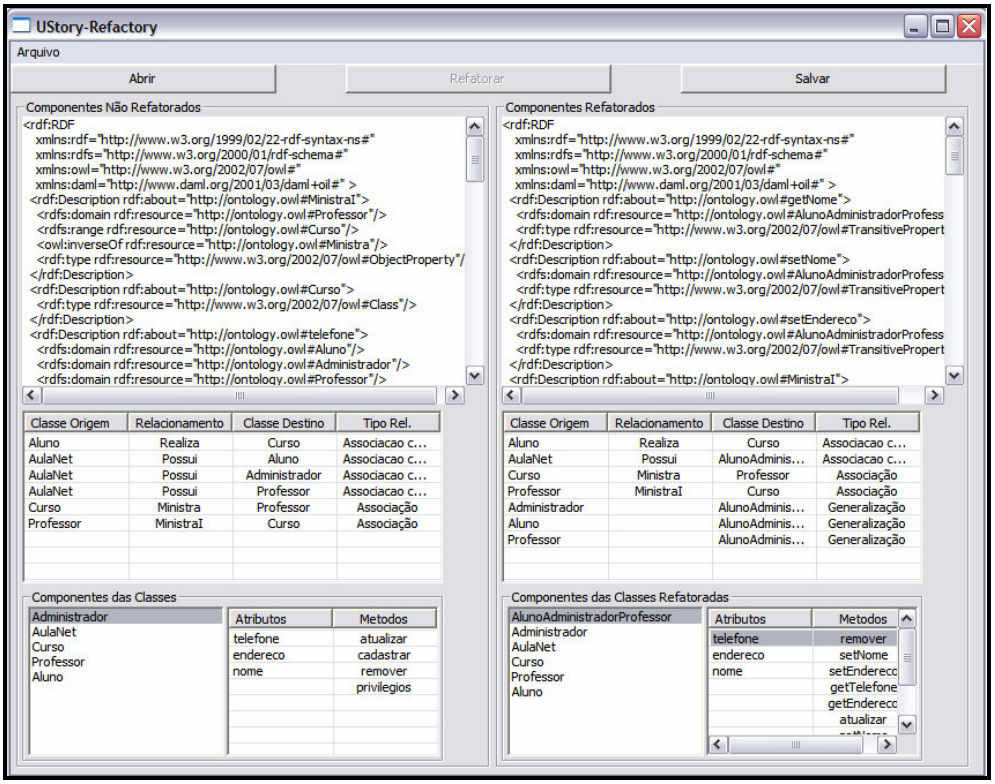


Figura 62: Aplicação da ferramenta no primeiro cartão

- **Segunda *User Story (CRC Card)* modelada**

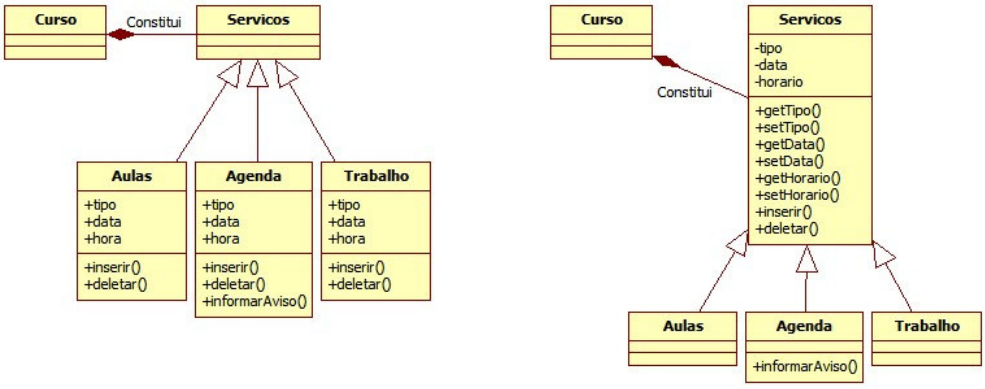
O segundo cartão apresentado refere-se a criação do Módulo Serviços, como pode ser visto na Figura 63.

Módulo de Serviços

"Um curso é constituído por serviços que podem ser de três tipos: aulas, agenda e trabalho. Todos possuem campos iguais como tipo de serviço, data e horário, e as operações inserir e deletar também estão presentes em todos os tipos de serviços. Destaca-se uma operação presente apenas na agenda que avisa o compromisso com um dia de antecedência."

Figura 63: Segunda *User Story (CRC Card)* modelada

A Figura 64 (a) apresenta o diagrama de classes do segundo cartão que modela os requisitos descrito na Figura 63 usando a UML-MC. Verifica-se, no exemplo, a existência de situações que podem ser refatoradas. Como às subclasses apresentam métodos e atributos duplicados, é possível aplicar por meio da ferramenta, inicialmente os padrões *Subir Campo na Hierarquia* e depois o padrão *Subir Método na Hierarquia*. Após a utilização destes padrões obtém-se um cenário favorável para aplicação do padrão *Auto-Encapsular Campo* permitindo o encapsulamento dos atributos. Após a aplicação da refatoração que pode ser visto na Figura 64 (b), constata-se, aqui, que o diagrama refatorado apresenta um design aperfeiçoado, que o torna mais eficiente de ser implementado.



(a) Não refatorado **(b) Refatorado**

Figura 64: Segundo cartão: aplicando as refatorações

A seguir, na Figura 65, pode ser visto como a ferramenta realiza a refatoração do segundo cartão.

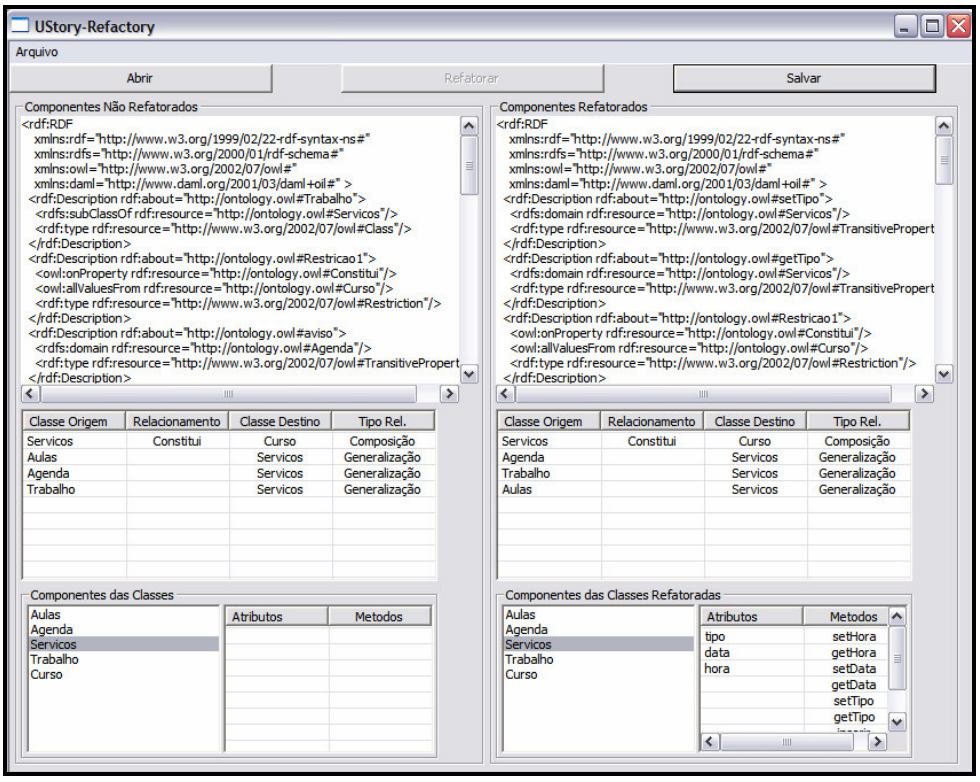


Figura 65: Aplicação da ferramenta no segundo cartão

- **Terceira *User Story (CRC Card)* modelada**

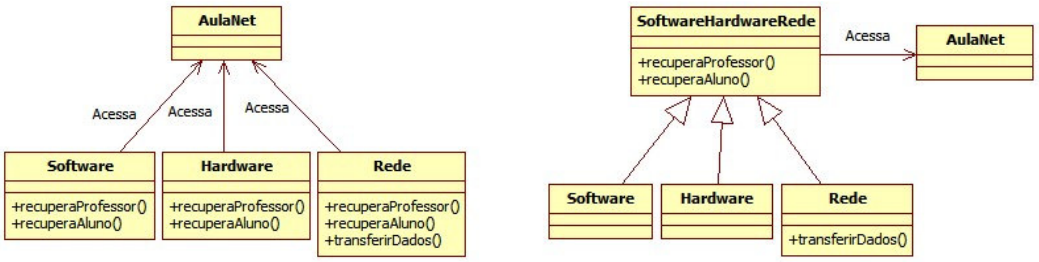
O terceiro cartão é descrito Figura 66, que descreve a utilização de *"Software"*, *"Hardware"* e *"Rede"* do ambiente *"AulaNet"*.

Módulo para a utilização de *Software*, Hardware e Rede

"A utilização de *software*, hardware e rede ocorre através do acesso pelo ambiente AulaNet. O *software*, hardware e rede apresentam operações em comum como recuperar aluno e professor. A rede ainda possui a operação de transferir dados."

Figura 66: Terceira *User Story (CRC Card)* modelada

Aplicando a formalização UML-MC tem-se o diagrama de classes apresentado na Figura 67 (a). Observa-se que melhoramentos podem ser feitos nos seguintes casos: (a) relacionamentos do tipo associação com direção com mesmo papel; (b) classes com métodos duplicados. Nestes casos pode-se aplicar apenas o primeiro caso do padrão de refatoração *Criar Classe Abstrata*. Com isso, ocorre a criação de uma classe abstrata chamada *"SoftwareHardwareRede"* que implementa os métodos duplicados. A Figura 68 mostra a interface da ferramenta da refatoração desse cartão com a concatenação das classes.



(a) Não refatorado **(b) Refatorado**

Figura 67: Terceiro cartão: aplicando as refatorações

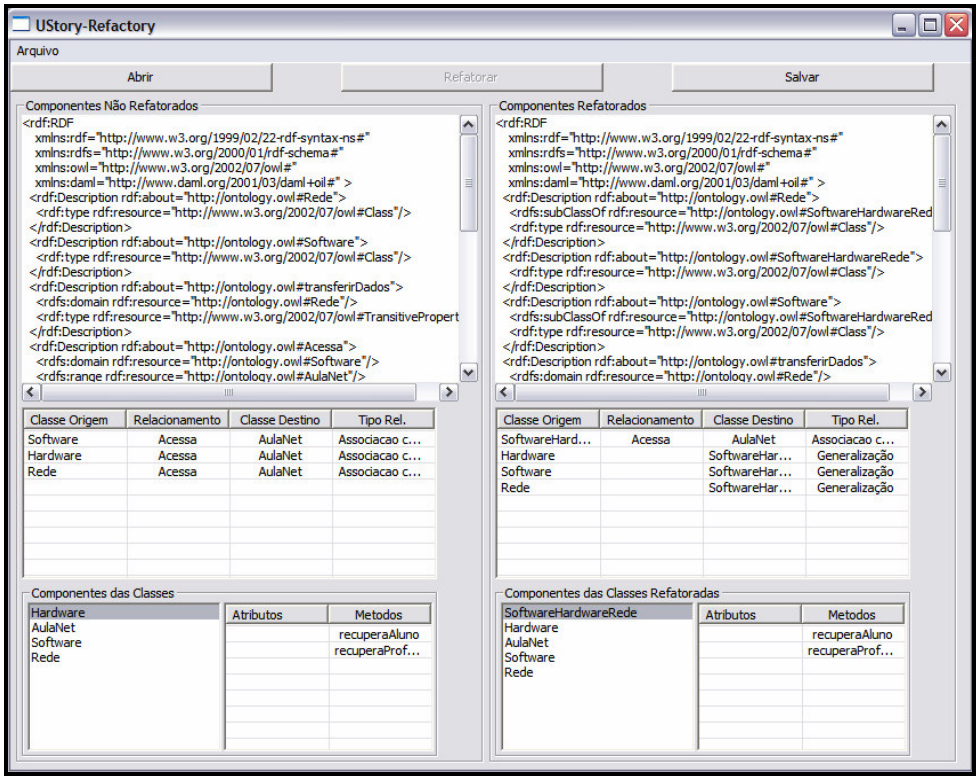


Figura 68: Aplicação da ferramenta no terceiro cartão

Portanto, analisando as *User Stories (CRC Cards)* desenvolvidas, percebe-se que a aplicação de apenas um padrão ou mais de refatoração depende da modelagem dos requisitos, podendo ou não serem passíveis de refatoração.

7. CONSIDERAÇÕES FINAIS

7.1. CONCLUSÃO

A realização desse trabalho permitiu evoluir e aprimorar as pesquisas na área da refatoração, introduzindo um novo processo de refatoração, em requisitos baseados em cartões *User Stories (CRC Cards), e* adquirindo requisitos que aperfeiçoem o projeto que está em desenvolvimento. Portanto, pode-se afirmar que esse trabalho introduz uma nova etapa dentro da metodologia de desenvolvimento *Extreme Programming*, que é a de aplicar a refatoração nos seus cartões de requisitos dessa metodologia.

Sendo assim, para o desenvolvimento desse trabalho, foram estudadas técnicas computacionais para a implementação da ferramenta e a disponibilização dos recursos, dentre as quais se destacam ontologias, OWL, API *Jena Ontology*, padrões de projeto e UML. Então, com a utilização desses, foi possível projetar a arquitetura da ferramenta, aplicar a técnica de refatoração e documentar o processo de construção do *software* por meio de diagramas de casos de uso, seqüência, atividades e classes.

Logo após finalizada a modelagem, foi possível atingir o objetivo principal, ou seja, o de desenvolver uma ferramenta que aplique padrões de refatoração aplicada em *User Stories (CRC Cards)* baseados de acordo com metodologia UML-MC. Portanto a pesquisa obteve os seguintes resultados:

- Implementação de cinco tipos de refatoração em nível de análise que proporcionam melhor *design* para a implementação do desenvolvedor;
- Criação de uma camada de ontologia que formaliza a UML para o formato de ontologia no formato OWL, proporcionando semântica e a interoperabilidade entre as aplicações;
- Desenvolvimento de um editor de interoperabilidade que garante a semântica dos dados no formato OWL e que deixa a ferramenta independente de outras aplicações, o que possibilita que os dados a serem refatorados possam ser gerados por essa ferramenta;

- Desenvolvimento de um mecanismo de extensão de refatoração que proporciona a inserção de novos padrões de refatorações por meio do padrão de projeto *Factory Method*. Sendo que, dessa maneira a ferramenta é totalmente modelável a novas alterações.

Além disso, a ferramenta apresenta a abordagem da metodologia UML-MC (ROBINSON, 2004) em OWL, o que proporciona uma representação abstrata dos dados, sendo, assim, o caso dessa dissertação. No entanto, essa representação abstrata em OWL não impediu que a aplicação dos padrões de refatorações desenvolvidos ocorresse com sucesso.

Assim, este trabalho fornece benefícios à Engenharia de *Software*, destacando-se primeiramente uma nova forma de aplicar a refatoração nos requisitos e por uma nova representação de um domínio de requisitos por meio de uma ontologia no formato OWL. Estas contribuições fornecem novas visões de como se pode aplicar a técnica de refatoração na indústria de *software*.

Dessa forma, portanto, pode-se concluir que os objetivos desse trabalho foram alcançados, na medida em que, dada a sua conclusão, a comunidade de desenvolvedores de *software* terá à sua disposição uma ferramenta que aplique a refatoração em requisitos baseados em *User Stories (CRC Cards).* Além disso, é importante salientar ainda que essa ferramenta proporciona uma segunda opção para aplicação da técnica de refatoração em requisitos. Desta maneira com a aplicação dos padrões de refatorações em *User Stories (CRC Cards)* no início do desenvolvimento de uma aplicação é possível abstrair inconsistências nos requisitos antes da finalização da aplicação, evitando assim que o cliente final identifique essas inconsistências. Desta forma pode-se dizer que a refatoração, quando aplicada juntament com o processo de Engenharia de Requisitos adquire-se requisitos consistentes, que garantem aos programadores a eficácia durante o desenvolvimento do projeto (*software*).

Está dissertação (PDF e fontes da ferramenta) estarão disponíveis na página do Professor Dr. Sérgio Crespo C. S. Pinto (http://www.inf.unisinos.br/~crespo).

7.2. TRABALHOS FUTUROS

A seguir são apresentados alguns trabalhos que poderão ser realizados como continuidades desta dissertação:

- Aprimorar a ontologia desenvolvida abordando maiores especificações e componentes da linguagem de modelagem UML;
- Inserir novos padrões de refatorações na ferramenta e na metodologia;
- Usar a *UStory-Refactory* como um *plugin* de uma ferramenta que tenha a elicitação de requisitos por meio de mapas conceituais;
- Tornar a *UStory-Refactory* uma ferramenta e interface *web*, aplicando tecnologias de *web-services* para que a ontologia seja exportada para outras aplicações.
- Desenvolver uma base de conhecimento através dos requisitos, proporcionando a reusabilidade dos requisitos;
- Testar a ferramenta com diferentes perfis de usuários, para verificar se ocorre a adequação de outros tipos de usuários na utilização da ferramenta e analisar se a ferramenta está apresentada de uma maneira usável;
- Desenvolver uma página *web* que apresente e explique a ferramenta;
- E, por último, bastante importante, desenvolver um editor gráfico que represente graficamente a representação de diagramas de classes da UML.

REFERÊNCIAS BIBLIOGRÁFICAS

(ANKORI, 2004) ANKORI, Ronit. **Automatic Requirements Elicitation in Agile Process**. In: IEEE International Conference on *Software* – Science Technology & Engineering (SwSTE'05), 2005.

(BECK, 2000) BECK, Kent. **Extreme Programming Explained: embrace change.** Upper Saddle River: Addison-Wesley, 2000.

(BECKETT, 2004) BECKETT, Dave; MCBRIDE, Brian. **RDF/XML Syntax Specification (Revised)** 2004. Disponível em: **<**http://www.w3.org/TR/rdf-syntax-grammar/**>** Acesso em: 10 mar. 2006.

(BOEHM, 1979) BOEHM, Barry W. ***Software* Engineering – As It Is**. In: IEEE Transactions on Computers - 4th International Conference on *Software* Engineering, Germany, 1979.

(BOOCH, 2006) BOOCH, Grady; RUMBAUGH, James; JACOBSON, Ivar. **UML Guia do Usuário.** 2º ed. Rio de Janeiro: Elsevier/Campus, 2006.

(BUTLER, 2001) BUTLER, Greg; XU, Lugang**. Cascaded Refactoring for Framework Evolution.** In: ACM Symposium on *Software* Reusability (SSR'01), 2001.

(CARVALHO, 2002) CARVALHO, Márcia; ABDELOUAHAB, Zair. **Um Método para Elicitação e Modelagem de Requisitos Baseados em Objetivos.** In: V Workshop de Engenharia de Requisitos, Espanha, 2002.

(COHN, 2004) COHN, Mike. **User Stories Applied: For Agile *Software* Development.** Canada: Addison-Wesley, 2004

(COCKBURN, 2001) COCKBURN, Alistair. **Escrevendo Casos de Usos Eficazes – Um Guia Prático para Desenvolvedores de *Software*.** Porto Alegre: Bookman, 2001.

(COYLE, 2002) COYLE, Frank P. **XML, Web Services and the Data Revolution.** Reading Addison – Wesley Information Technology Series. 2002.

(FOWLER, 2004) FOWLER, Martin. **Refatoração – Aperfeiçoando o Projeto de Código Existente.** Porto Alegre: Bookman, 2004.

(FONSECA, 2000) FONSECA, Frederico; EGENHOFER, Max; BORGES, Karla A. V. **Ontologias e Interoperabilidade entre SIGs.** In: II Brazilian Symposium on GeoInformatics (GEOINFO 2000), 2000.

(GAMMA, 2000) GAMMA, Erich; HELM, Richard; JOHNSON, Ralph; VLISSIDES, John. **Padrões de Projeto – Soluções Reutilizáveis de *Software* Orientado a Objetos.** Porto Alegre: Bookman, 2000.

(GONÇALVES, 2004) GONÇALVES, Paulo Roberto; BRITO, Parcilene Fernandes. **Manipulação de uma Ontologia desenvolvida em OWL através da API JENA 2**

Ontology. In: VI Encontro de Estudantes de Informática do Estado de Tocantis (ECOINFO 2004), 2004.

(GORTS, 2004) GORTS, Sven. **The Refactoring Cycle**. 2004. Disponível em: <http://www.refactoring.be/thumbnails/cycle/cycle.html>. Acesso em: 10 dez. 2005.

(GRUBER, 1993) GRUBER, Thomas R. **A Translation Approach to Portable Ontology Specifications.** In: Knowledge Acquisition, 1993.

(HIGO, 2005) HIGO, Yoshiki; KAMIJA, Toshihiro; KUSUMOTO, Shinji; INOUE, Katsuro. **ARIES: Refactoring Support Tool for Code Clone.** In: ACM International Conference on *Software* Engineering – Third Workshop on *Software* Quality (WoSQ'05), 2005.

(HOLZNER, 2001) HOLZNER, Steve. **Inside XML.** Estados Unidos: New Riders Publishing, 2001.

(JENA, 2003) **Jena 2 Ontology API**. 2003. Disponível em: <http://jena.Sourceforge.net/ontology>. Acesso em: 10 jan. 2006.

(JEFFRIES, 1999) JEFFRIES, Ron. **XProgramming.com – An Agile *Software* Development Resource**. 1999. Disponível em: <http://www.xprogramming.com/>. Acesso em: 17 jan. 2006.

(KATAOKA, 2001) KATAOKA, Yoshio; ERNST, Michael; GRISWOLD, Willian; NOTKIN, David. **Automated Support for Program Refactoring using Invariants.** In: IEEE Conference on *Software* Maintenance, 2001.

(KLEIN, 2001) KLEIN, Michael. Tutorial: The Semantic Web - XML, RDF, and Relatives. In: IEEE Intelligent Systems, 2001.

(KOTONYA, 1998) KOTONYA, Gerald; SOMMERVILLE, Ian. **Requirements Enginnering Process and Techniques**. New York: John Wiley & Sons Ltda, 1998.

(LAMSWEERDE, 2000) LAMSWEERDE, Axel Van. **Requirements Engineering in the Year 00: A Research Perspective.** In: IEEE 22th International Conference on *Software* Engineering, 2000.

(LEITE, 1997) LEITE, Julio Sampaio do Prado; ROSSI, Gustavo; BALAGUER, Federico; MAJORANA, Vanesa; KAPLAN, Gladys; HADAD, Graciela; OLIVEROS, Alejandro. **Enhancing a Requirements Baseline with Scenarios**. In: IEEE Third International Requirements Engineering Symposium, 1997.

(LI, 2003) LI, Huiging; REINKE, Claus; THOMPSON, Simon. **Tool Support for Refactoring Functional Programs.** In: ACM SIGPLAN Workshop, 2003.

(LEONARDI, 2002) LEONARDI, Maria; LEITE, Julio Sampaio do Prado. **Using Business Rules in Extreme Requirements.** In: 14th International Conference CAISE, 2002.

(LUSTOSA, 2004) LUSTOSA, Pollyane de Almeida; TEIXEIRA, Darlene; Brito, Parcilene Fernandes. **OWL no Desenvolvimento de uma Ontologia para um Sistema de**

Inteligência Competitiva. In: VI Encontro de Estudantes de Informática do Estado de Tocantis (ECOINFO 2004), 2004.

(MASSONI, 2005) MASSONI, Tiago; GHEYI, Rohit; BORBA, Paulo. **Formal Refactoring for UML Class Diagrams**. In XIX Simpósio Brasileiro de Engenharia de *Software*, 2005.

(MCGUINNES, 2004) MCGUINNESS, Deborah L.; HARMELEN, Frank Van. **OWL Web Ontology Language Overview.** 2004. Disponível em: <http://www.w3.org/TR/owl-features/>. Acesso em: 28 mar. de 2006.

(MENDONÇA, 2004) MENDONÇA, Nabor; MAIA, Paulo; FONSECA, Leonardo; ANDRADE, Rossana. **Refax: A Refactoring Framework Based on XML**. In: IEEE International Conference on *Software* Maintenance (ICSM'04), 2004.

(MENS, 2004) MENS, Tom; TOURWÉ, Tom. **A Survey of *Software* Refactoring**. In: IEEE Transactions on *Software* Engeneering, 2004.

(MILLER, 1998) MILLER, Eric. **An Introduction to the Resource Description Framework.** In: D-Lib Magazine, 1998.

(MYLOPOULOS, 1992) MYLOPOULOS, John; CHUNG, Lawrence; NIXON, Brian. **Representing and Using Non-Functional Requirements: A Process-Oriented Approach.** In: IEEE Transactions on *Software* Engineering, 1992.

(NOVELLO, 2005) NOVELLO, Taisa Carla. **Ontologias, Sistemas baseados em conhecimento e modelo de Banco de Dados.** 2005. Disponível em: <http://www.inf.ufrgs.br/~clesio/cmp151/cmp15120021/artigo_taisa.pdf> Acesso em: 20 mar. 2006.

(NOY, 2004) NOY, Natalya F.; MCCUINNESS, Deborah L. **Ontology Development 101: A guide to creating your first ontology.** In Stanford University, Stanford, 2004.

(NUSEIBECH, 2000) NUSEIBECH, Bashar; EASTERBROOK, Steve. **Requirements Engineering: A Roadmap**. In ACM Future of *Software* Engineering, 2000.

(OMG, 2003) OMG, Object Management Group. **Unified Modeling Language Specification Version 1.4.1.** 2003. Disponível em: <http://www.omg.org/docs/pas/04-03-01.pdf>. Acesso em: 01 fev. 2005.

(OMG, 2005) OMG, Object Management Group. **Ontology Definition Metamodel.** 2005. Disponível em: <http://www.omg.org/docs/ad/05-08-01.pdf> . Acesso em: 17 set. 2006.

(OPDYKE, 1992) OPDYKE, Willian. F. **Refactoring Objetc-Oriented Frameworks**. 1992. 206 f. These de Doutorado, University of IIIinois, Urbana-Chapaign, 1992.

(PACHECO, 2001) PACHECO, Roberto Carlos dos Santos; KERN, Vinícius Medina. **Uma Ontologia Comum para a Integração de Bases de Conhecimento sobre a Ciência e Tecnologia.** In. Ci. Inf. Brasília v.30, n.3, 2001. Disponível em: <http://www.scielo.br/pdf/ci/v30n3/7287.pdf>. Acesso em: 12 abr. 2006

(PARREIRAS, 2004) PARREIRAS, Fernando Silva. **Ontologias na Integração de Sistemas Corporativos.** 2004. Disponível em: <http://www.fernando.parreiras.nom.br/content /view/33/42/> Acesso em: 20 de mar. 2006.

(PRESSMAN, 2002) PRESMAN, Roger S. **Engenharia de *Software***. Rio de Janeiro: McGraw-Hill, 2002.

(RATZINGER, 2005) RATZINGER, Jacek; FISCHER, Michael; GALL, Harald. **Improving Evolvability Through Refactoring.** In: ACM Intenational Workshop on Mining *Software* Repositories (MSR'05), 2005.

(REES, 2002) REES, Michael. **A Feasible User Story Tool for Agile *Software* Development?** In: IEEE Ninth Asia-Pacific *Software* Engineering Conference(APSEC'02), 2002

(REGNELL, 1999) REGNELL, Bjorn. **Requirements Engineering with Use Cases – A Basis for *Software* Development.** 1999. 225 f. These de Doutorado. Departament of Communications Systems Lunde Institute of Tecnology, Lund University, 1999.

(ROBERTS, 1999) ROBERTS, Donald Bradley. **Practical Analysis for Refactoring**. 1999. 137 f. These de Doutorado, University of IIIinois, Urbana-Chapaign, 1999.

(ROBINSON, 2004) ROBINSON, Genessa; RHEINHEIMER, Letícia; SCOPEL, Marcelo; CRESPO, Sérgio; GIRAFFA, Lúcia; MENEZES, Crediné. **UML-MC: uma extensão da UML para representar Mapas Conceituais.** In: XV Simpósio Brasileiro de Informática na Educação, 2004.

(RUI, 2003) RUI, Kexing; BUTLER, Greg. **Refactoring Use Case Models: The Metamodel**. In: ACM 26th Australasian Computer Science Conference (ACSC'03), 2003.

(SMITH, 2002) SMITH, Michael K. **Web Ontology Language (OWL) – Guide Version 1.0.** 2002. Disponível em: <http://www.w3.org/TR/2002/WD-owl-guide-20021104/#AppendixA>. Acesso em: 10 de maio. 2006.

(SOMMERVILLE, 1997) SOMMERVILLE, Ian; SAWYER, Pete. **Requirements Engineering – A Good Practice Guide**. New York: J. Wiley & Sons, 1997.

(STRECKENBACH, 2004) STRECKENCACH, Mirko; SNELTING, Gregor. **Refactoring Class Hierarchies with KABA.** In: ACM Conference on Object Oriented Programming Systems Languagens and Applications (OOSPLA'04), 2004.

(SWT, 2006) ECLIPSE: an open development platform. **SWT – The Standard Widget Toolkit**. Disponível em: <http://www.eclipse.org/swt/>. Acesso em: 10 abr. 2006.

(SUNYÉ, 2001) SUNYÉ, Gerson; POLLET, Damien; TRAON, Yves; JÉZÉQUEL, Jean. **Refactoring UML Models.** In: ACM 4th International Conference on the Unified Modeling Language, Modeling Languages, Concepts and Tools, 2001.

(VITTEK, 2003) VITTEK, Marian. **Refactoring Browser with Preprocessor**. In: IEEE Seventh Conference On *Software* Maintenance and Reengineering (CSMR'03), 2003.

(WEISER, 1981) WEISER, Mark. **Program Slicing**. In: IEEE 5th International Conference on *Software* Engineering, 1981.

(WIEGERS, 2003) WIEGERS, Karl. ***Software* Requirements**. Canada: H. B. Fenn and Company Ltda, 2003.

(XU, J., 2004) XU, Jian; YU, Wei; RUI, Kexing; BUTLER, Greg. **Use Case Refactoring: a Tool and Case Study**. In: IEEE 11th Asia-Pacific *Software* Engineering Conference, 2004.

(XU, L., 2005) XU, Lugang. **Cascaded Refactoring for Framework Development and Evolution.** 2005. 260 f. These de Doutorado, Department of Computer Science and *Software* Engineering – Concordia University, Montréal, 2005.

(YU, 2004) YU, Wei; LI, Jun; BUTLER, Greg. **Refactoring Use Case Models on Episode.** In: IEEE 19th International Conference on Automated *Software* Engineering (ASE'04), 2004.