FUNDAÇÃO DE ENSINO “EURÍPIDES SOARES DA ROCHA”
CENTRO UNIVERSITÁRIO “EURÍPIDES DE MARÍLIA” – UNIVEM
TRABALHO DE CONCLUSÃO DE CURSO – TCC

JÚLIO CÉSAR BRANDT AGUILAR

UTILIZAÇÃO E ANÁLISE DE UM PROCESSO DE DESENVOLVIMENTO DE SOFTWARE ORIENTADO A ASPECTOS

MARÍLIA
2007

JÚLIO CÉSAR BRANDT AGUILAR

UTILIZAÇÃO E ANÁLISE DE UM PROCESSO DE DESENVOLVIMENTO DE SOFTWARE ORIENTADO A ASPECTOS

Monografia apresentada ao Curso de Ciência da Computação, da Fundação de Ensino Eurípides Soares da Rocha, Mantenedora do Centro Universitário Eurípides de Marília – UNIVEM, como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.

Orientador:
Prof. Dr. Valter Vieira de Camargo

MARÍLIA
2007

Aos meus pais Carlos e Rita de Cássia, que sonharam antes de mim com este dia, jamais poderei ser suficientemente grato. Pela realização deste ideal, minha homenagem, admiração e carinho.

AGRADECIMENTOS

A todos que me inspiraram aos sonhos, a sentir, a pensar e a viver.

Agradeço a Deus por conceder a luz para trilhar esta jornada.

Aos meus queridos pais Carlos e Rita de Cássia, tanto a agradecê-los pela vida e pelo exemplo, de força, garra e perseverança, jamais conseguiria retribuir tão grande carinho, a homenagem da mais profunda gratidão pele lição de vida que sabiamente me prestaram e, a tentativa modesta de externar o verdadeiro afeto filial em pálida retribuição pelo inesgotável amor com que sempre me cercaram.

A minha namorada Elisa que esteve ao meu lado me incentivando e me inspirando, companheira de todas as horas, que com paciência e carinho sempre torceu pela minha vitória.

Aos professores que nos transmitiram seus conhecimentos e experiências profissionais e de vida com dedicação e carinho, que nos guiaram para além das teorias, das filosofias e das técnicas, expresso o meu maior agradecimento e o meu profundo respeito, que sempre será insuficiente diante do muito que foi oferecido.

E em especial agradeço o meu professor e orientador Valter de Camargo com quem muito aprendi e que sempre com toda sua atenção me auxiliou para a conclusão deste presente trabalho.

“A maior recompensa do nosso trabalho não é o que nos pagam por ele, mas aquilo em que ele nos transforma.”

John Ruskin

AGUILAR, Júlio César Brandt. **Utilização e Análise de um processo de desenvolvimento de software orientado a aspectos.** 2007. 68 f. Monografia (Bacharel em Ciências da Computação) - Centro Universitário Eurípides de Marília, Fundação de Ensino Eurípides Soares da Rocha, Marília, 2007

RESUMO

A Programação Orientada a Aspectos (POA) é uma extensão da Programação Orientada a Objetos (POO) que une os conceitos, já bem estabelecidos da POO com os novos conceitos da POA. Esse novo paradigma de programação surgiu da necessidade de separar os interesses transversais, em módulos separados do sistema, tornando mais simples a manutenção, a compreensão e o reúso dos artefatos. Com o surgimento desse novo paradigma várias pesquisas surgiram na área de identificação e separação de interesses transversais. Entretanto, atualmente ainda há uma carência de pesquisas no sentido de averiguar os impactos que esse novo paradigma causa em todas as etapas de um processo de desenvolvimento de software. Esse trabalho apresenta a utilização de um processo de desenvolvimento de software orientado a aspectos denominado ProFT/PU. O objetivo do trabalho averiguar os prós e contras do processo e fazer uma comparação com outras abordagens existentes com relação à fase de identificação de interesses transversais. Foi desenvolvido um software utilizando-se o contexto de um sistema para automação de uma rede de postos de combustível aplicando as técnicas descritas neste documento. Com o estudo de caso realizado, notou-se que os artefatos e atividades propostas pelo ProFT/PU auxiliam a identificação e acompanhamento de aspectos durante o desenvolvimento, e que a fase de identificação de interesses desse processo é mais bem elaborada do que outros trabalhos propostos na literatura.

Palavras-chave: Processo de Desenvolvimento Orientado a Aspectos. Identificação de Intereses Transversais. Programação Orientada a Aspectos. Frameworks Transversais.

AGUILAR, Júlio César Brandt. . **Use and Analysis of a Development Process in Software Orientation and Aspects.** 2007. 68 f. Monografia (Bacharel em Ciências da Computação) - Centro Universitário Eurípides de Marília, Fundação de Ensino Eurípides Soares da Rocha, Marília, 2007.

ABSTRACT

The Aspect-Oriented Programming (AOP) is an extension of the Object-Oriented Programming (OOP) that joins the concepts, already established of the OOP with the new concepts of the AOP. This new paradigm has emerged from the necessity to separate the crosscutting concerns, in separate modules of the system, becoming simpler the maintenance, the understanding and reuse. With the sprouting of this new paradigm some research had appeared in the area of identification and separation of crosscutting concerns. However, nowadays still it has a lack of research in the direction to inquire the impacts that this new paradigm cause in all the stages of a software development process. This work presents the use of an aspect-oriented process called ProFT/PU. The objective of the work to inquire the advantages and disadvantages of the process and to make a comparison with other existing works with relation to the phase of identification of crosscutting concerns. A gas-station system was developed as a case study to follow the process. With the study, one noticed that the phases and activities proposed by ProFT/PU assist the identification and tracing of aspects during the development, and that the phase of identification of concerns of this process most is elaborated of that other works considered in literature.

Keywords: Process of Development Aspects Orientation. Identification of Crosscuting Concern. Aspects Orientation Programming. Crosscuting Frameworks.

SUMÁRIO

1 INTRODUÇÃO

Atualmente, há uma grande preocupação na área de Engenharia de Software para tentar melhorar a expressividade do paradigma orientado a objetos, e um dos mais promissores entre os novos paradigmas que endereçam esta questão é o Orientado a Aspectos.

A Programação Orientada a Aspectos (*Aspect-Oriented Programming*) ou simplesmente AOP, foi introduzida em 1996 por Gregor Kickzales, quando estava no Xerox Palo Alto Research Center. A AOP, assim como a Programação Orientada a Objetos (*Object-Oriented Programming*, OOP), introduz um novo paradigma e um conjunto de abstrações e mecanismos para facilitar o desenvolvimento de software. A idéia básica é permitir modularizar determinados interesses de um sistema, chamados de "interesses transversais", em módulos independentes.

A POA é baseada na idéia de que as metodologias estruturada e orientada a objetos não oferecem meios suficientes para separar de forma clara o código com diferentes propósitos em um sistema, visto que alguns requisitos do sistema (tais como: geração de log, persistência, sincronização, etc.) não podem ser claramente encapsuladas em um único módulo do sistema quando paradigmas tradicionais são utilizados.

Esta limitação encontrada nos paradigmas tradicionais leva a uma dificuldade extra para entender, manter e reutilizar o código onde estes aspectos se encontram. Por outro lado, fazendo uso do paradigma Orientado a Aspecto, um sistema pode ser projetado de tal forma que o código com diferentes propósitos torna-se encapsulado em diferentes unidades específicas, as quais podem ser combinadas depois seguindo algumas regras específicas (Kiczales *et al*., 1997).

Um software é composto pelos seus requisitos funcionais e não funcionais. Um requisito funcional pode ser definido como uma função ou atividade que o sistema faz (quando pronto) ou fará (quando em desenvolvimento). Devem ser definidos claramente e relatados explicitamente (Cysneiros *et al*., 1997). Requisitos não funcionais (Chung *et al*., 1999), ao contrário dos funcionais, não expressam função a ser implementada em um sistema, expressam condições de comportamento e restrições que devem prevalecer. A programação orientada a aspectos busca separar os requisitos funcionais e não funcionais, ou seja, dividir os interesses (*concernes*), para depois uni-los (*weaving*), formando um sistema completo. A POA dá assistência à POO para tratar a decomposição funcional da classe.

Na programação orientada a objetos o modelo de abstração trabalha com classes como unidades. No projeto procura-se separar cada interesse em uma classe distinta, porém nem sempre isso é possível. Quando o código de determinados interesses aparece misturado numa mesma classe chama-se “entrelaçamento”, e quando o código de um determinado interesse aparece espalhado em diversas classes chama-se “espalhamento”. Para aqueles interesses que tornam-se entrelaçados e espalhados dá-se o nome de “interesses transversais”. A programação orientada a aspectos foi criada para resolver a implementação destes interesses transversais para os sistemas modelados com orientação a objetos. Usando AOP é possível separar estes interesses em unidades chamadas aspectos.

Várias linguagens vêm sendo propostas para a implementação de aspectos, entre elas a mais disseminada é AspectJ (Kiczales *et al.*, 2001), que permite uma maneira de implementar os interesses transversais em separado no sistema de software. AspectJ é uma extensão do Java e provê algumas construções fundamentais que são: pontos de junção (*join points)*, conjuntos de junção (*pointcuts)*, adentos (*advices)*, introduções (*introductions)* e aspectos (*aspects)*.

O desenvolvimento de software orientado a aspectos (DSOA) propõe softwares ainda mais modulares, conseqüentemente mais fáceis de alterar e simples de manter. A modularidade promove a reusabilidade, sempre proposta pelas técnicas de engenharia de Software.

1.1 Motivação e Objetivos

A motivação para a realização desse estudo é a carência de trabalhos que descrevem de forma precisa como identificar e tratar interesses transversais durante as fases do ciclo de vida de um sistema.

Os objetivos são, realizar um estudo sobre a identificação de interesses transversais nas primeiras fases de desenvolvimento de um software e testar um processo de desenvolvimento de software orientado por aspectos denominado ProFT/PU proposto por Camargo (2006).

1.2 Estrutura do Trabalho

No Capitulo 2 são mostrados os principais conceitos da programação orientada a aspectos e também é relatado um *framework* de persistência desenvolvido por Camargo e Masiero (2005). No Capitulo 3 são relatados alguns trabalhos na área do desenvolvimento orientado a aspectos e identificação de interesses transversais, no Capitulo 4 é apresentado o processo de desenvolvimento de software orientado a aspectos denominado ProFT/PU, no Capitulo 5 é realizado um estudo de caso, no Capitulo 6 é feita uma comparação entre a proposta de Camargo (2006) e as de outros autores, e por fim o Capitulo 7 traz as considerações finais.

2 PROGRAMAÇÃO ORIENTADA A ASPECTOS

Na década de 70 surgiram os primeiros trabalhos que se referiam ao termo "interesse" como um conceito de modularização de software (Dijkstra, 1976). Muito do que se propôs nessa época e em pesquisas subseqüentes foi sendo gradativamente incorporado às linguagens de programação e aos modelos de processo de desenvolvimento de software, culminando na década de 90 com o desenvolvimento orientado a objetos e a padronização propiciada pela UML (*Unified Modelling Language*) (Jacobson *et al*., 1999).

A Programação Orientada a Aspectos (POA) (Gregor Kiczales *et al*., 1997) e a linguagem AspectJ (Kiczales *et al*., 2001) surgiram no final da década de 90 como uma importante contribuição para modularizar interesses transversais, que até então ficavam misturados e espalhados pelo código orientado a objetos, sem que fosse possível organizá-los em módulos independentes. A POA fornece novas abstrações que contribuem para a separação de interesses (*separation of concerns*) – um objetivo antigo da engenharia de software que visa a separar os interesses encontrados no código-fonte de um sistema (Dijkstra, 1976). Com a POA é possível implementar separadamente os interesses-base e os interesses transversais, o que até então era dificultado somente com orientação a objetos. Os interesses-base referem-se à funcionalidade principal do sistema e os transversais referem-se a restrições globais e a requisitos não-funcionais, como, por exemplo: persistência, distribuição, autenticação, controle de acesso e concorrência.

2.1 Conceitos Básicos da Programação Orientada a Aspectos

A separação de interesses é um conceito antigo da Engenharia de Software que visa a separar, também em nível de código, os interesses existentes em um sistema. O objetivo é melhorar a inteligibilidade do sistema e também sua manutenção, reúso e evolução. O mecanismo utilizado pelos paradigmas convencionais para a separação de interesses são chamadas sub-rotinas e hierarquias de herança (no paradigma orientado a objetos). Contudo, esses mecanismos são úteis para a separação de interesses-base de um sistema, que são aqueles ligados diretamente à funcionalidade da aplicação.

Tar e Ossher (2002) comentam que os paradigmas de desenvolvimento de software convencionais sofrem da Tirania da Decomposição Dominante, que consiste na existência de

apenas um tipo de módulo – o módulo dominante – para se encapsular mais de um tipo de interesse. Alguns exemplos de módulos dominantes são as classes na programação orientada a objetos, as funções nas linguagens funcionais e as regras na programação baseada em regras. Um paradigma que permite a decomposição em apenas um tipo de módulo é chamado de unidimensional, pois há apenas uma dimensão para a modularização dos interesses, não importando quantos tipos existem de interesses (Tar e Ossher, 2002).

A existência de apenas um tipo de módulo para se encapsular mais de um tipo de interesse faz com que aqueles interesses que envolvem restrições globais – chamados "interesses transversais" (Kiczales *et al*., 1997) – não fiquem adequadamente encapsulados e acabam misturando-se com os interesses-base. Kiczales *et al*. (1997) foram quem disseminaram o termo "interesses transversais" (*crosscutting concerns*), explicando que podem variar de requisitos de alto nível como segurança e qualidade de um serviço a noções de baixo nível como sincronização e manipulações de *buffer* de memória. Podem ser tanto funcionais, como regras de negócio, quanto não-funcionais, como gerenciamento de transações e persistência de dados.

Alguns exemplos são: sincronização, interação de componentes, persistência, distribuição, adaptabilidade, gerenciamento de registros (*logging*), controle de acesso e de segurança. Alguns são dependentes de um domínio específico, como definição de produtos financeiros, enquanto outros são mais gerais, como por exemplo, sincronização e fluxo de trabalho (*work-flow*). O termo "interesse transversal" faz analogia com o fato de que sua implementação com técnicas tradicionais de programação entrecorta transversalmente os módulos do sistema, afetando vários módulos e causando entrelaçamento (*tangling*) e espalhamento (*spreading*) de código de diferentes interesses. O entrelaçamento ocorre quando o código de um determinado interesse encontra-se misturado com o código de um outro interesse dentro de um mesmo módulo. O espalhamento ocorre quando o código de um interesse encontra-se em vários módulos do sistema. O entrelaçamento e o espalhamento de código de diferentes interesses causam problemas de manutenção, reúso e evolução.

Com o objetivo de tratar esse problema, Kiczales *et al*. (1997) criou a Programação Orientada a Aspectos (POA), que é baseada na idéia de que sistemas são programados de forma mais adequada distinguindo interesses que são "transversais" de interesses que podem ser considerados como "base". Assim, os interesses transversais são projetados e implementados separadamente dos interesses-base. Os interesses transversais são implementados em módulos aspectuais e os interesses-base em classes normais. Na segunda

etapa, que deve contar com auxílio automatizado, realiza-se a composição das duas partes em um único sistema.

A POA introduz novas abstrações de modularização e mecanismos de composição para melhorar a separação de interesses transversais. As novas abstrações propostas pela POA são: aspectos (*aspects*), pontos de junção (*join points*), conjuntos de junção (*pointcus*), adendos (*advices*) e declarações inter-tipos (*inter-type declarations/introductions*).

Aspecto é o termo usado para denotar a abstração da POA que dá suporte a um melhor isolamento de interesses transversais. Em outras palavras, um aspecto corresponde a um interesse transversal e constitui uma unidade modular projetada para afetar um conjunto de classes e objetos do sistema.

Os pontos de junção são pontos bem definidos da execução de um sistema. Alguns exemplos de pontos de junção são: chamadas a métodos, execuções de métodos, leitura de atributos e modificação de atributos. Por meio dos pontos de junção, torna-se possível especificar o relacionamento entre aspectos e classes. Tais pontos de junção são os locais no programa onde os aspectos podem atuar. Eles podem ser especificados em uma linguagem orientada a aspectos através da definição de conjuntos de junção. Um conjunto de junção é o mecanismo que especifica os pontos de junção em que aspectos e classes se relacionam.

A implementação do comportamento transversal modularizado por um aspecto é feita em "adendos". Adendo é um construtor semelhante a um método de uma classe, que define o comportamento dinâmico executado quando são alcançados um ou mais conjuntos de junção definidos previamente. Existem três tipos de adendos: os adendos anteriores (*before*), os posteriores (*after*) e os de substituição (*around*). Os adendos anteriores são executados sempre que os pontos de junção associados são alcançados e antes do prosseguimento da computação; os adendos posteriores são executados no término da computação, ou seja, depois que os pontos de junção forem executados e imediatamente antes do retorno do controle ao chamador; os adendos de substituição tomam o controle do código-base podendo ou não devolver o controle.

As declarações inter-tipos podem ser utilizadas quando se deseja introduzir atributos e métodos em classes-base do sistema. Ao contrário de adendos, que operam de forma dinâmica, as declarações inter-tipos operam estaticamente, em tempo de compilação.

Um aspecto pode afetar a estrutura estática ou dinâmica de classes e objetos. A estrutura dinâmica é afetada por meio da especificação de conjuntos de junção e adendos; a estrutura estática é afetada por meio de declarações inter-tipo. Além de especificarem

elementos que entrecortam classes de um sistema, aspectos podem possuir métodos e atributos internos, como classes OO.

São três as propriedades básicas da POA (Kiczales *et al*., 1997), (Elrad *et al*., 2001):

• dicotomia aspectos-base: diz respeito à adoção de uma distinção clara entre classes e aspectos. Os sistemas orientados a aspectos são decompostos em classes e aspectos. Os aspectos modularizam os interesses transversais e as classes modularizam os interesses-base;

• inconsciência: propriedade desejável da programação orientada a aspectos. É a idéia de que os componentes não precisam ser preparados para serem entrecortados por aspectos (Elrad *et al*., 2001). Pela propriedade de inconsciência, os componentes não percebem os aspectos que poderão afetá-los. Embora seja uma propriedade interessante, algumas pesquisas atuais começam a apontar vantagens quando o código-base possui consciência da existência dos aspectos (Kiczales e Mezini, 2005);

• quantificação: capacidade de escrever declarações unitárias e separadas que afetam muitos pontos de um sistema (Elrad *et al*., 2001). Pela propriedade de quantificação, é possível fazer declarações do tipo "em programas P, sempre que a condição C for verdadeira, faça a ação A". As linguagens de programação orientadas a aspectos devem apoiar a definição de atributos, métodos, conjuntos de junção, adendos e declarações inter-tipos em aspectos. A linguagem AspectJ (Kiczales *et al*., 2001), que é uma extensão orientada a aspectos da linguagem Java, oferece suporte à definição dessas abstrações.

Combinador é o mecanismo responsável pela composição de classes e aspectos. Quase todo o processo de combinação de aspectos e classes é realizado como um pré-processamento em tempo de compilação. Entretanto, a Versão 5 da linguagem AspectJ já oferece suporte ao processo de combinação dinâmico, pela associação de mecanismos da linguagem com *frameworks,* como *AspectWerkz*.

Há várias abordagens lingüísticas para a implementação de sistemas orientados a aspectos, por exemplo, AspectC, AspectC#, AspectS e Appostle e AspectJ. Porém, como AspectJ é a linguagem mais difundida e a que foi utilizada no contexto deste trabalho, será a única descrita com detalhes.

2.2 Modelo de Join Point

Os *join points* são os pontos na execução de um programa de componentes nos quais os aspectos serão aplicados. São conceitos abstratos em AspectJ, pois não possuem nenhuma construção de programa para representá-lo.

Conforme mostrado na Figura 1, o modelo de *join points* do AspectJ é baseado em um grafo dirigido de envio de mensagens a objetos. Os nós seriam os pontos onde as classes e objetos recebem uma invocação de método ou têm um atributo acessado. Os vértices representariam as relações de fluxo de controle entre os nós, partindo do que chama para o que é chamado. O fluxo de controle, na realidade, ocorre nos dois sentidos: no sentido do vértice, quando a ação é iniciada, e no sentido contrário, quando a ação realizada pelo sub-nó estiver concluída (STEIN, 2002).

Isto significa que um *join point* pode estar em uma das direções do fluxo de controle, ou seja, quando uma ação é iniciada e quando uma ação é terminada.

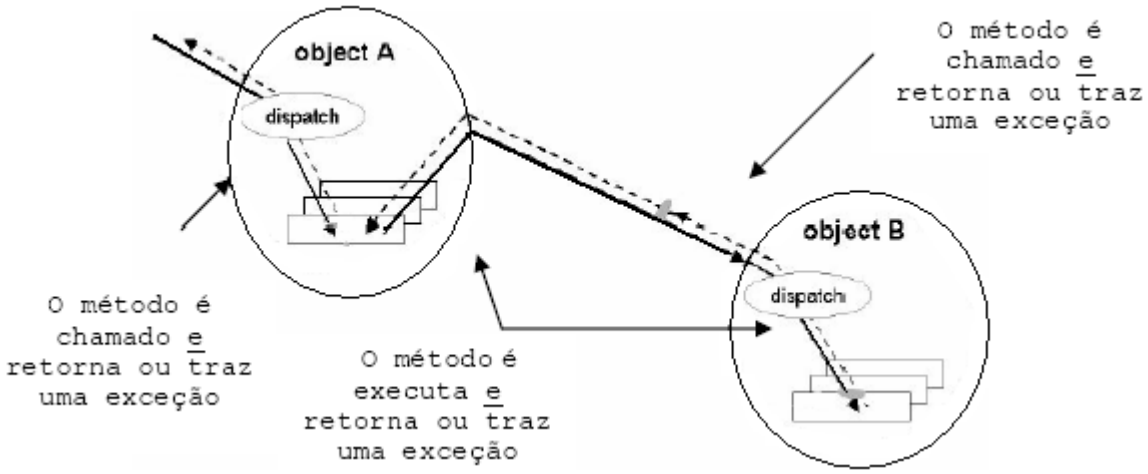

Figura 1 - Fluxo de controle em AspectJ (Kiczales, 2001).

2.3 Pointcut

Pointcuts são formados pela composição de *join points* através dos operadores (e), (ou), e (não). Utilizando *pointcuts* podemos obter valores de argumentos de métodos, objetos em execução, atributos e exceções dos *join points*. Na Figura 2 é mostrado a definição de um

pointcut que identifica as invocações de todos os métodos cujo nome inicia com "set", com qualquer tipo de retorno, nomes, ou parâmetros devido ao uso dos *wildcards* ***** *e "..".*

```
public pointcut pt1(): call(* Abastecimento+.set*(..));
```

Figura 2 – Definição de um PointCut

Além disso, este *pointcut* restringe os *join points* àqueles os quais o código em execução pertence aos subtipos (devido ao uso do *wildcard* +) da classe Abastecimento. Para definir *pointcuts*, identificando os *join points* a serem afetados, utilizamos construtores de AspectJ chamados designadores de *pointcut* (*pointcut designators*), como os apresentados na Tabela 1.

Tabela 1 – Construtores de AspectJ

call(Assinatura)	Invocação de método/construtor identificado por Assinatura
execution(Assinatura)	Execução de método/construtor identificado por Assinatura
Get(Assinatura)	Acesso a atributo identificado por Assinatura
Set(Assinatura)	Atribuição de atributo identificado por Assinatura
this(Padrão Tipo)	O objeto em execução é instância de Padrão Tipo
target(Padrão Tipo)	O objeto de destino é instância de Padrão Tipo
args(Padrão Tipo, ...)	Os argumentos são instância de Padrão Tipo
within(Padrão Tipo)	O código em execução está definido em Padrão Tipo

Onde Padrão Tipo é uma construção que pode definir um conjunto de tipos utilizando *wildcards*, como * e +. O primeiro é um *wildcard* conhecido, pode ser usado sozinho para representar o conjunto de todos os tipos do sistema, ou com depois de caracteres, representando qualquer seqüência de caracteres. O último deve ser utilizado junto ao nome de um tipo para assim representar o conjunto de todos os seus subtipos.

2.4 Advices

Advices são construções que definem código adicional que deverá executar nos *join points*. Na Figura 3 é mostrado dois *advices*, um do tipo `before()` e o outro do tipo `after()`. Ambos atuam sobre o *pointcut* `pt1()` definido anteriormente. Isso faz com que antes das chamadas aos métodos que iniciam com, o método `System.out.println("")` seja chamado, e o mesmo é feito depois dessas chamadas.

```
before(): pt1() {
      System.out.println("Antes de set ...");
}

after(): pt1() {
      System.out.println("Depois de set ...");
}
```

Figura 3 – Exemplo de Advices

2.5 Static Crosscutting

A linguagem AspectJ permite alterar a estrutura estática de um programa através da adição de membros de classe, da alteração da hierarquia de classes ou da substituição de exceções checadas por não checadas. O mecanismo que adiciona membros a uma classe é chamado de introduções ou declarações inter-tipos. Em AspectJ pode-se introduzir métodos concretos ou abstratos, construtores e atributos em uma classe. O aspecto mostrado na Figura 4 introduz em uma classe chamada Posto um atributo `bomba`, um método `setBomba()`, um método `getBomba()` e um novo construtor.

```
public aspect BombasPosto {

        private Bomba Posto.bomba;
        public void Posto.setBomba(Bomba bomba) {}
        public Bomba Posto.getBomba() {}
        public Posto.new() {}

}
```

Figura 4 – Aspecto Bombasposto

Na Tabela 2 apresenta outras construções em AspectJ que alteram a estrutura estática de um programa.

Tabela 2 – Construções em AspectJ

Declare parents : PadrãoTipo extends ListaTipos;	Declara que os tipos em PadrãoTipo herdam dos tipos em ListaTipos
Declare parents : PadrãoTipo implements ListaTipos;	Declara que os tipos em PadrãoTipo implementam os tipos em ListaTipos
Declare soft : PadrãoTipo : Pointcut;	Declara que qualquer exceção de um tipo em PadrãoTipo que for lançada em qualquer join point identificado por Pointcut será encapsulado em uma exceção não checada

2.6 Aspectos Abstratos

A linguagem AspectJ também permite a criação de aspectos abstratos, semelhantemente a uma classe abstrata em Java, porém, além de métodos abstratos o aspecto também pode possuir conjuntos de junção abstratos. Um conjunto de junção abstrato não tem conhecimento dos pontos de junção que serão afetados, pois esses pontos devem ser informados em um aspecto concreto que especializar o aspecto abstrato. Como um adendo pode ser definido sobre um conjunto de junção abstrato, pode-se implementar um comportamento transversal no aspecto abstrato sem saber de antemão quais são os pontos de junção, e conseqüentemente, qual é o código-base em que esse comportamento irá atuar. Aspectos abstratos é a base do desenvolvimento orientado ao reúso, e conseqüentemente, do desenvolvimento de *frameworks* orientados a aspectos.

2.7 Um Framework de Persistência Orientado a Aspectos

Um *Framework* Orientado a Aspectos de Persistência, também conhecido como *Framework* Transversal (FT) de Persistência (Camargo e Masiero, 2005) foi desenvolvido. Assim como outros *frameworks* de persistência orientados a objetos, como *Hibernate* (2006), *Cayenne* (2006) e *OJB* (2006), têm como objetivo facilitar o desenvolvimento de uma aplicação orientada a objetos que utiliza banco de dados relacional. Os mecanismos necessários para tratar as incompatibilidades entre os dois paradigmas (relacional e orientado a objetos) são implementados pelo *framework* e podem ser reusados durante o desenvolvimento de novas aplicações. Esse FT é uma evolução de um trabalho anterior (Camargo *et al*., 2003; Ramos *et al*., 2004) e possui como base o padrão de projeto Camada de Persistência (*Persistence Layer*) (Yoder *et al*., 1998).

O FT consiste em duas partes com objetivos distintos e que foram implementadas em módulos distintos. A primeira parte, chamada de "operações persistentes", é um conjunto de operações de persistência que devem ser herdadas por classes de aplicação persistentes e que podem ser utilizadas para armazenar, remover, atualizar e realizar consultas no banco de dados. A estratégia de implementação adotada no projeto do FT foi introduzir (com declarações inter-tipo) todas as operações de persistência em uma interface e fazer com que as classes de aplicação persistentes implementem essa interface. Algumas das operações de persistência disponibilizadas pelo FT podem ser vistas na Tabela 3. Essas operações constituem a interface disponibilizada pelo FT para a aplicação. A segunda parte, chamada de "conexão", é referente ao interesse de conexão com o banco de dados, que tem como objetivo identificar locais do código-base em que a conexão deve ser aberta e fechada. A primeira parte depende da segunda para conseguir executar as operações de persistência.

Tabela 3 – Operações de persistência disponibilizadas pelo FT

Assinatura da Operação	Descrição
`Public int getNextID()`	Utilizada para retornar o próximo ID disponível na tabela do banco de dados que é representada pela classe do objeto alvo.
`Public boolean save()`	Utilizada para armazenar o objeto alvo da chamada em sua respectiva tabela do banco de dados. Retorna "true" se a operação foi realizada com sucesso.
`Public boolean delete()`	Utilizada para remover o objeto alvo da chamada de sua respectiva tabela do banco de dados. Retorna "true" se a operação foi realizada com sucesso.
`Public ResultSet findlike()`	Utilizada para localizar e recuperar o objeto alvo no banco de dados. Se o objeto foi encontrado, retorna um `ResultSet` com os dados desse objeto, caso contrário retorna um `ResultSet` vazio.
`public ResultSet findlikeByField (String columnName, String columnValue)`	Utilizada para recuperar todos os registros da tabela do banco de dados, representada pela classe do objeto alvo, que a coluna `columnName` tenha o valor String `columnValue`.
`public ResultSet findlikeByField (String columnName, int columnValue)`	Utilizada para recuperar todos os registros da tabela do banco de 0dados, representada pela classe do objeto alvo, que a coluna `columnName` tenha o valor `columnValue` do tipo int.
`public ResultSet findlikeByField (String columnName1, String columnName2, String columnValue1, String columnValue2){`	Utilizada para recuperar todos os registros da tabela do banco de dados, representada pela classe do objeto alvo, que a coluna `columnName1` tenha o valor String `columnValue1` e que a coluna `columnName2` tenha o valor String `columnValue2`.
`public ResultSet findlike (String columnName, FrameworkDate columnValue){`	Utilizada para recuperar todos os registros da tabela representada pelo objeto alvo que a coluna `columnName` tenha o valor `columnValue` do tipo `FrameworkDate`.
`public ResultSet findlikeByField_intervalOfDates (String columnName1, String columnName2, String columnName3, String columnName4, Integer columnValue1, FrameworkDate initialDate, FrameworkDate finalDate, Integer columnValue2){`	Utilizada para recuperar os registros da tabela representada pela classe do objeto alvo que a coluna `columnName1` tenha o valor `columnValue1` e que a coluna `columnName2` tenha o valor `initialDate`, e que a coluna `columnName3` tenha o valor 0`finalDate` e que a coluna `columnName4` tenha o valor `columnValue2`.
`public ResultSet findlikeByField_intervalOfDates (String columnName1, String columnName2, String columnName3, Integer columnValue, FrameworkDate initialDate, FrameworkDate finalDate){`	Utilizada para recuperar todos os registros da tabela representada pela classe do objeto alvo em que a coluna `columnName1` tenha o valor `columnValue` e que a coluna `columnName2` tenha um valor maior ou igual a `initialDate` e que a coluna `columnName3` tenha um valor menor ou igual a `finalDate`.
`public ResultSet findlikeByField_intervalOfDates (String columnName1, String columnName2, String columnName3, String columnName4, Integer columnValue1, FrameworkDate`	Utilizada para recuperar os registros da tabela representada pela classe do objeto alvo que a coluna `columnName1` tenha o valor `columnValue1` e que a coluna `columnName2` tenha o valor `initialDate`, e que a coluna `columnName3` tenha o valor `finalDate` e que a coluna `columnName4` tenha o valor `columnValue2`.

Na Figura 5 é mostrado um diagrama que representa ilustrativamente as classes e aspectos que compõem as partes Operações Persistentes e Conexão, distinguidas pelas formas geométricas com linhas tracejadas. Nessa figura, os aspectos estão sendo representados por retângulos destacados em cinza, e os relacionamentos de associação que partem dos aspectos para alguma entidade representam que o aspecto afeta a entidade entrecortando (*crosscutting*) a execução/chamada de seus métodos ou introduzindo (*inter type declaration*) operações, por meio de declarações intertipo. Os demais relacionamentos possuem semântica convencional da UML. Todos os métodos com o estereótipo <<hook>> são abstratos e devem ser sobrepostos pelo engenheiro de aplicação em classes concretas (Camargo, 2006).

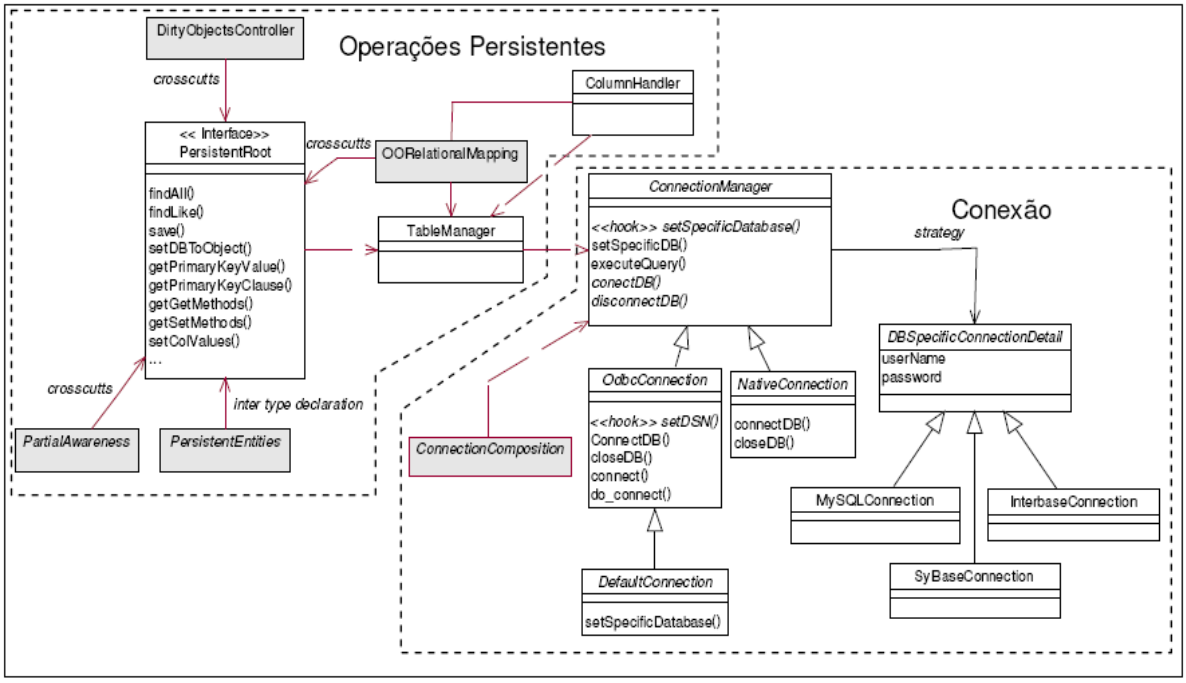

Figura 5 - Estrutura do Framework Transversal de Persistência (Camargo, 2006)

O processo de reúso do FT de Persistência, que consiste em acoplá-lo a uma determinada aplicação possui as etapas de instanciação e de composição. Na instanciação, três variabilidades precisam ser determinadas: o tipo da consciência (Total ou Parcial), o tipo da conexão com o banco de dados (ODBC ou *driver* nativo) e o próprio banco de dados. Na etapa de composição é necessário informar quais são as classes de aplicação persistentes e quais os locais do código-base em que a conexão deve ser aberta e fechada. Esses passos são o mínimo necessário para se acoplar o FT de persistência a um código-base.

A escolha das variabilidades da conexão é feita utilizando a classe `myConnectionVariabilities`, mostrada na Figura 6. Para realizar a conexão o usuário terá que informar o banco que será utilizado no caso mysql, e o nome do DSN criado no painel de controle do Windows, no caso do sistema em questão é "AutoPosto".

```
import persistence.connection.*;

public class myConnectionVariabilities extends OdbcConnection {

	public String setSpecificDatabase(){
		return "mysql";
	}
	public String setDSN(){
		return "AutoPosto";
	}
}
```

Figura 6 – Escolha das Variabilidades da Conexão

Na Figura 7 é mostrado um trecho de código do aspecto `MyPersistentEntities` que é uma especialização do aspecto `PersistentEntities` e que declara que as classes `Cliente`, `Bomba`, `Combustível`, `Estoque`, `Nota` e `Posto` são sub-tipos da interface `PersistentRoot`. Fazendo isso, os objetos dessas classes agora dispõem de operações de persistência para armazenar (`save()`), remover (`delete()`), atualizar (`update()`) e localizar (`find*()`) registros no banco de dados.

```
import persistence.*;

      public aspect MyPersistentEntities extends PersistentEntities {

            declare parents : Cliente      implements PersistentRoot;
            declare parents : Bomba        implements PersistentRoot;
            declare parents:  Combustivel  implements PersistentRoot;
            declare parents:  Estoque      implements PersistentRoot;
            declare parents:  Nota         implements PersistentRoot;
            declare parents:  Posto        implements PersistentRoot;
}
```

Figura 7 – Aspecto MyPersistentEntities

O FT mostrado neste capítulo foi disponibilizado por Camargo (2006) e será utilizado como estudo para a implementação de um software orientado a aspectos. Tudo que diz respeito ao FT de persistência foi desenvolvido por Camargo e Masiero (2005).

3 TRABALHOS RELACIONADOS

Kassab *et al* (2005) propuseram um método sistemático que permite tanto a identificação quanto o rastreamento de interesses transversais ao longo do processo de desenvolvimento.

O método proposto por Kassab *et al* (2005) é composto por cinco fases: Descoberta de Requisitos, Análise e Concretização de Interesses, Composição de Requisitos, Projeto e Implementação, como pode ser visto na Figura 8.

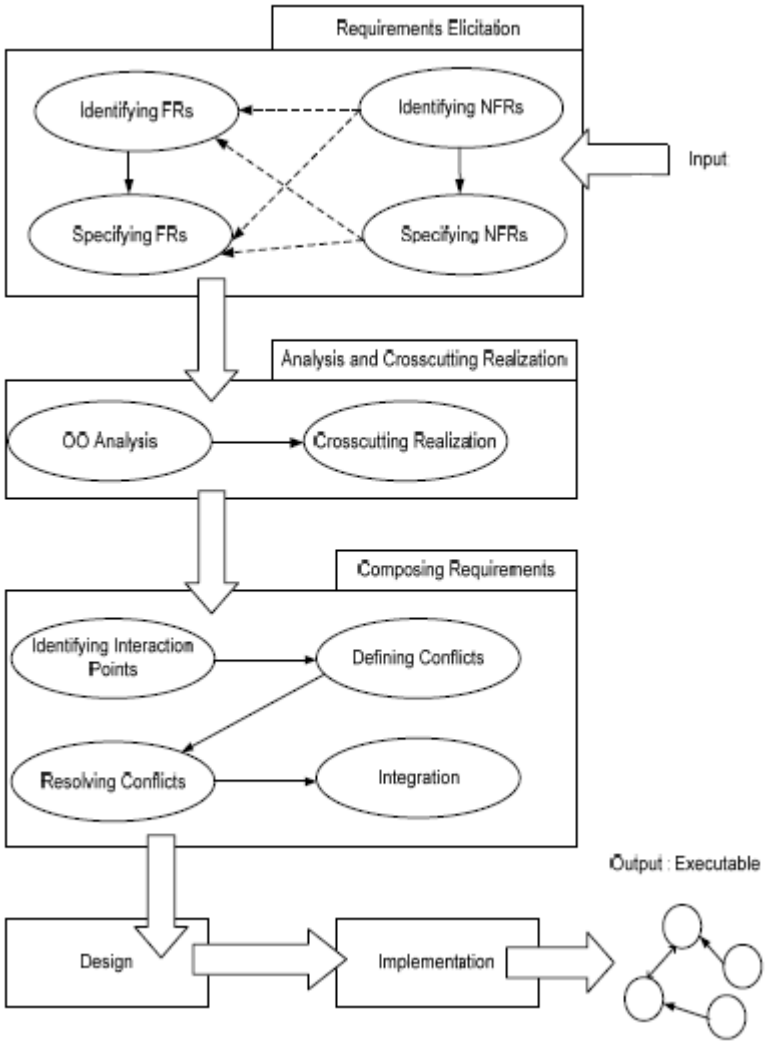

Figura 8 – Fases do modelo proposto por Kassab (retirado de Kassab *et al.* (2005))

A fase de Descoberta de Requisitos é utilizada para identificar os requisitos do sistema. Ela é composta de quatro atividades: identificar requisitos funcionais, especificar

requisitos funcionais, identificar requisitos não-funcionais e especificar requisitos não-funcionais.

Na fase de descoberta de requisitos funcionais o documento de requisitos pode ser o ponto de partida para sua identificação. Essa atividade envolve discussões com os colaboradores, revisando propostas, construindo protótipos e organizando reuniões para a descoberta dos requisitos.

Na fase de especificação de requisitos funcionais cada caso de uso funcional do sistema é refinado descrevendo o seu comportamento.

Na fase de identificação de requisitos não-funcionais são utilizadas várias técnicas para a descoberta dos requisitos não-funcionais, ficando a cargo do analista a decisão de qual técnica será utilizada.

Na fase de especificação de requisitos não-funcionais, Kassab *et al.* (2005) propuseram uma matriz para identificar e relacionar requisitos não-funcionais com os requisitos funcionais e o caso de uso que eles afetam.

No método proposto por Kassab *et al.* (2005), a fase de análise está composta de duas outras atividades: Análise OO (Orientada a Objetos) e Concretização de Interesses. Na fase de análise OO o objetivo é entender as descrições textuais e resumir o software em um modelo de análise OO, já na fase de concretização de interesses verificar se a funcionalidade de um caso de uso afeta outros casos de uso, se afetar diz-se que este caso é um interesse transversal. Os requisitos não-funcionais por serem considerados propriedades globais do sistema sempre são classificados como interesses transversais.

Na fase de composição de requisitos a meta é descrever interesses (funcional e não funcional) com o Diagrama de Caso de Uso e o Diagrama de Domínio. Isto é alcançado em uma série de quatro atividades: (1) identificar os pontos de interação, (2) identificar possíveis conflitos entre exigências a cada ponto de interação, (3) solucionar conflitos, e (4) integrar interesses.

Na fase de projeto, cada caso de uso é refinado. O diagrama de seqüência mostra detalhes das operações entre os objetos envolvidos em uma determinada operação do sistema. Diagramas de comunicação mostram o fluxo de mensagem entre os objetos e uma aplicação OO e também sugerem as associações básicas entre as classes.

A fase de implementação está associada ao desenvolvimento do sistema (programação) em que são levados em consideração todas as informações obtidas nas fases anteriores.

A proposta de Araújo *et al.* (2002) consiste em utilizar diagramas UML e modelos pré-definidos para identificar os requisitos funcionais e os interesses não-funcionais. A composição de requisitos não-funcionais e funcionais é representada graficamente por meio de adaptações de caso de uso e diagramas de seqüência.

Araújo *et al.* (2002) propuseram para a identificação e especificação dos interesses transversais um processo que é dividido verticalmente em módulos separados. Na Figura 9 é mostrado esquematicamente o processo utilizado para identificar e modelar interesses transversais.

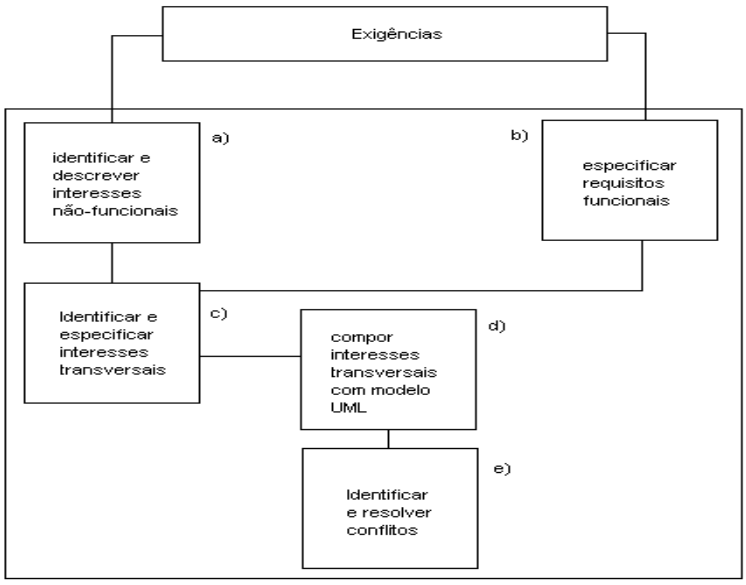

Figura 9 – Processo de identificação e especificação de interesses (retirado de Araújo *et al* (2002))

Na parte destacada com a letra (a), o objetivo é identificar e descrever os interesses não-funcionais (possíveis candidatos a aspectos), essa identificação é feita basicamente verificando se esse interesse é um exemplo clássico de interesse transversal; na parte destacada com a letra (b) é executada uma especificação tradicional de interesses funcionais, neste caso, usando a UML como aproximação onde o modelo de caso de uso é a técnica de especificação principal; na parte destacada com a letra (c) são identificados e especificados os interesses transversais; na parte destacada com a letra (d) é feita a composição de interesses transversais, utilizando a UML; e na parte destacada com a letra (e) são identificados e solucionados os conflitos que podem surgir do processo de composição. Foram adotados os conceitos de *overlapping*, *overriding* e *wrapping*.

- *Overlapping*: os interesses do aspecto modificam os interesses funcionais transversais. Neste caso, os interesses de aspecto podem ser requeridos antes do funcional ou podem ser requeridos depois.
- *Overriding*: os interesses do aspecto sobrepõem os interesses funcionais transversais. Neste caso, o comportamento descrito pelos aspectos substituem o comportamento dos requisitos funcional.
- *Wrapping*: os interesses do aspecto "encapsulam" os interesses funcionais transversais. Neste caso, o comportamento descrito pelos interesses funcionais é entrelaçado pelo comportamento descrito pelos interesses do aspecto.

A proposta feita por Araujo *et al.* (2002) é a utilização da UML para a identificação dos requisitos (funcionais e não-funcionais) e dos possíveis candidatos a aspectos, os autores não descrevem em detalhes o processo para a identificação dos requisitos apenas apresentam uma idéia para implementação de software orientado a aspectos.

Rashid *et al.* (2002) propuseram um modelo denominado AORE (*Aspect-Oriented Requirements Engineering*), para identificar interesses transversais no nível de requisitos. Segundo o modelo AORE, inicialmente é necessário identificar requisitos funcionais e não funcionais. A ordem dessas duas atividades não é determinada pelo modelo, ela dependerá da interação entre os engenheiros e os usuários do sistema.

Após a identificação dos requisitos funcionais e não-funcionais é utilizada uma matriz para relacionar os requisitos funcionais. Nesta matriz verifica se algum requisito possui alguma funcionalidade que afeta ou restringe um outro requisito como, por exemplo, verifica se o requisito 1 possui alguma função que afeta ou restringe o requisito 2 até o requisito N. Se um requisito influenciar ou restringir um ou mais requisitos funcionais, ele é considerado candidato a aspecto.

Tabela 4 - Matriz utilizada para relacionar requisitos (retirado de Rashid *et al.* (2003))

	Requisito 1	**Requisito 2**	**...**	**Requisito N**
Requisito 1				X
Requisito 2		X		X
...				
Requisito N	X	X		

A próxima atividade é responsável por analisar candidatos a aspectos em mais detalhes, identificando conflitos entre eles e estabelecendo prioridades. Por fim, deve-se

especificar o impacto de candidatos a aspectos em termos de duas dimensões: mapeamento e influência. A dimensão de mapeamento especifica como cada candidato a aspecto será manipulado nos estágios seguintes do desenvolvimento. Candidatos a aspectos podem ser mapeados em uma função, decisão arquitetural ou num aspecto. Por sua vez, a dimensão de influência estabelece quais os pontos do ciclo de desenvolvimento que o candidato a aspecto afetará. Na Tabela 5 apresenta um exemplo de saída dessa atividade. Em Rashid *et al.* (2003) não é apresentado o processo para determinar como o engenheiro de software estabelece as dimensões de candidatos a aspectos.

Tabela 5 - Exemplo de saída da atividade Especificar Dimensão de Aspectos Candidatos do modelo AORE (retirado de [Rashid *et al.* 2002])

Candidato a Aspecto	Influência	Mapeamento
Compatibilidade	especificação, arquitetura, projeto, manutenção	Função
Tempo de resposta	Arquitetura, projeto	Aspecto
Restrições	Especificação	Função
Corretude	Especificação, projeto	Função
Segurança	Arquitetura, projeto	Aspecto
Disponibilidade	Arquitetura	Decisão
Sistema multi-usuário	Arquitetura, projeto	Aspecto

Rashid *et al.* (2003) apresentam uma melhoria no modelo AORE original. A principal modificação realizada foi a inclusão de duas novas atividades: uma responsável por compor candidatos a aspectos e requisitos, determinando como um candidato a aspecto influencia ou restringe um requisito por meio de regras de composição, e a outra responsável por lidar com conflitos.

O modelo AORE proposto por Rashid *et al.* (2003), propõe inicialmente a identificação e separação dos requisitos funcionais e não-funcionais podendo assim relacioná-los em uma matriz, identificando os possíveis candidatos a aspecto. Após a identificação é utilizada outra matriz de detalhamento onde são definidos a prioridade de cada candidato a aspecto.

Jacobson (2003) defende que o mecanismo de extensão de caso de uso poderia ser utilizado para modelar aspectos no nível de requisitos. Em resumo, esse mecanismo modela o relacionamento onde um caso de uso de extensão adiciona comportamento a um caso de uso base, sem modificar o caso de uso original. O relacionamento de extensão necessita das seguintes informações: uma condição para a extensão (quando a extensão é condicional) e

uma referência para o ponto de extensão no caso de uso alvo (uma posição no caso de uso onde a adição de comportamento deve ser feita).

Na Tabela 6 exibem-se algumas das equivalências entre o mecanismo de extensão de casos de uso e alguns conceitos utilizados na orientação a aspectos, segundo Jacobson (2003).

Figura 6 - Equivalências entre conceitos de extensão de casos de uso e os de orientação a aspectos (retirado de Jacobson, (2003))

Extensão de Caso de Uso	Orientação a Aspectos
Caso de uso de extensão	Aspecto
Ponto de extensão	Ponto de junção
Comportamento de caso de uso estendido	Advice

Outra proposta de Jacobson (2003) é modificar a filosofia do desenvolvimento de software guiado por casos de uso. A idéia é que exista um modelo de análise e projeto e um modelo de implementação específicos para cada caso de uso, não havendo, a princípio, a preocupação em reusar partes de modelos de outros casos de uso. Apenas as classes, atributos e métodos essenciais à implementação do caso de uso são criados, mesmo que seja esperado que outros casos de uso possam precisar de classes conceitualmente similares com métodos e atributos adicionais. Assim, casos de uso poderão ser desenvolvidos independentemente durante todo o processo de desenvolvimento e depois combinados para compor o sistema.

A separação de casos de uso seria feita produzindo-se partes do sistema, nomeadas de módulos. Um módulo de caso de uso encapsularia todos os componentes necessários para sua concretização. Por fim, essas partes seriam combinadas para produzir o sistema desenvolvido. Por exemplo, a Equipe 1 fica responsável por todo desenvolvimento dos casos de uso A e B, a Equipe 2 fica responsável pelo caso de uso C e a Equipe 3 pelos casos de uso D e E. A aplicação é então composta através da combinação dessas implementações.

Contudo, Jacobson (2003), apenas apresenta essas idéias, mas não indica como isso poderia ser realizado na prática. Uma desvantagem da abordagem de Jacobson é que, partindo do princípio que os casos de uso serão desenvolvidos independentemente, possivelmente por equipes diferentes, é muito provável que ocorram redundâncias nas implementações; por exemplo, uma mesma função ser implementada por diferentes equipes. Outro problema é que nomes diferentes podem ser utilizados para um mesmo elemento conceitual ou pior, o mesmo nome pode ser usado para diferentes propósitos. A princípio, o processo de combinação pode resolver esses problemas, mesclando redundâncias e fornecendo métodos para corrigir

nomeações e outras divergências. Algumas práticas como, estabelecimento de convenções para nomenclatura; o uso de um glossário de termos e conceitos específico para o domínio da aplicação; e a comunicação freqüente com outras equipes, ajuda a minimizar esses problemas.

4 PROFT/PU

Camargo (2006) apresenta um processo de desenvolvimento de software orientado a aspectos apoiado por *Frameworks* Transversais (FTs) denominado ProFT/PU. Esse processo segue a estrutura de fases e disciplinas do Processo Unificado (PU) sendo as fases definidas como: Concepção, Elaboração, Construção e Transição como pode ser visto na Figura 10.

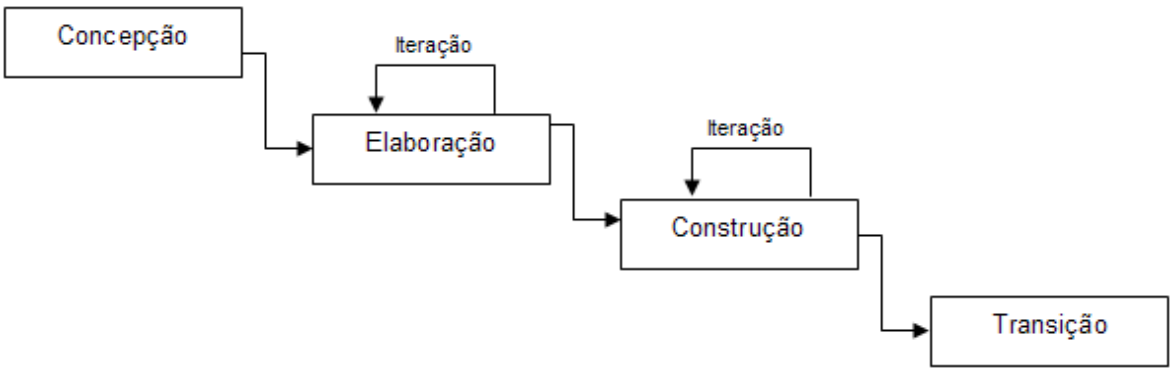

Figura 10 – Fases do ProFT/PU

Esse processo é iterativo e incremental. As disciplinas que apóiam ProFT/PU são: Disciplina de Requisitos, Disciplina de Análise, Disciplina de Projeto e Disciplina de Implementação. Cada disciplina possui atividades específicas como pode ser visto na Tabela 7, e cada disciplina se repete dentro de todas as fases do PU, porém com ênfases diferentes conforme o processo progride, dependendo da fase, a ênfase nessas disciplinas é maior ou menor.

Tabela 7 – Atividades do ProFT/PU

Atividades da Disciplina Requisitos
Identificar Atores e Detalhar Casos de Uso Funcionais
Identificar e Especificar Casos de Uso Não-Funcionais
Planejar Iterações
Atividades da Disciplina Análise
Identificar e Registrar Casos de Uso Colaboradores
Identificar e Registrar Casos de Uso Candidatos a Aspectos
Criar Diagramas de Seqüência do Sistema
Definir Contratos das Operações do Sistema
Desenvolver Modelo Conceitual
Atividades da Disciplina Projeto
Identificar Aspectos
Selecionar Aspectos para a Iteração Atual
Projetar Aspectos
Acoplar Aspectos com a Base
Desenvolver Diagramas de Iteração
Desenvolver Modelo de Classes de Projeto
Projetar Modelos de Dados
Atualizar Regras de Composição
Desenvolver Modelo de Projeto Composto
Desenvolver Visão da Arquitetura
Registrar Pontos de junção
Atividades da Disciplina de Implementação
Implementar Classes de Domínio
Implementar Banco de Dados
Atualizar Regras de Composição
Implementar Aspectos
Implementar Classes Controladoras
Implementar Interfaces
Testar

A fase de concepção é relativamente curta e não é realizada em iterações. O propósito dessa fase é estabelecer uma visão geral dos objetivos do projeto, determinar se é viável e decidir se realmente deve passar por uma investigação mais profunda na fase de elaboração. Nessa fase se dá uma ênfase maior na disciplina de requisitos.

A fase de elaboração se concentra no projeto e na implementação dos casos de uso alocados para a iteração atual e é realizada em iterações. Os casos de usos originais devem ser

obtidos na visão de iteração. O objetivo principal é implementar uma porção do sistema que esclareça os detalhes arquiteturais e de maior risco.

As primeiras iterações da elaboração possuem grande ênfase nas disciplinas de requisitos e na análise, já que muitos requisitos ainda não foram descobertos e registrados. Entretanto, com o decorrer das iterações, a ênfase deixa de ser nos requisitos e passa a ser na análise, no projeto e na implementação. Cada atividade das disciplinas podem se repetir várias vezes, dependendo da quantidade de iterações planejadas para o desenvolvimento, quando é executada sobre uma parte já desenvolvida do sistema, no caso de não ser a primeira iteração, os artefatos gerados por ela são atualizados e não gerados novamente.

A fase de Construção tem como objetivo finalizar o desenvolvimento do sistema concentrando-se nos casos de uso que possuem um impacto menor na arquitetura do sistema e não são tão importantes do ponto de vista da funcionalidade principal. A estrutura de disciplinas e atividades dentro desta fase é a mesma da fase de elaboração, contudo, a ênfase é bem maior no projeto e implementação.

Dependendo da estratégia de desenvolvimento que foi adotada, no final das iterações planejadas para a fase de construção pode ser necessário que vários aspectos ainda tenham que ser projetados e implementados. Se esse for o caso, deve-se realizar novamente um replanejamento com o objetivo de classificar esses aspectos e planejar como serão desenvolvidos.

As disciplinas e as atividades do ProFT/PU serão vistas com mais detalhes no decorrer do estudo, já que esse processo é a base para o realização deste trabalho.

5 EXEMPLO DE USO DO PROFT/PU

5.1 Considerações Iniciais

Como já comentado, para a realização desse estudo foi implementado como exemplo um sistema para automação de uma rede de postos de combustível. O sistema consiste basicamente no gerenciamento das bombas de combustível automatizando o controle de estoque e o controle do caixa, o sistema ainda permite o cadastramento de funcionários e clientes como também à emissão de diversos relatórios e consultas. O acesso às informações do sistema pode ser realizado por meio da Internet mediante a autenticação do usuário.

5.2 Fase de Concepção

Na fase de concepção o propósito é estabelecer uma visão inicial do projeto, determinar se é viável e decidir se realmente deve passar por uma investigação mais profunda na fase de elaboração.

As atividades dessa fase são: Identificar Atores e Detalhar Casos de Uso Funcionais; Especificar Casos de Uso Não-Funcionais e Planejar Iterações.

5.2.1 Disciplina Requisitos

5.2.1.1 Identificar Atores e Detalhar Casos de Uso Funcionais

A identificação e detalhamento dos casos de uso e atores foi baseado no Documento de Requisitos. Para a elaboração do documento foi realizada uma pesquisa de como um sistema para postos de combustível deveria se comportar e suas principais funções, também foram realizadas reuniões para a discussão dos possíveis casos de usos não funcionais. A seguir encontra-se o diagrama resultante (Figura 11) e a descrição do caso de uso Atualizar Caixa (Figura 12).

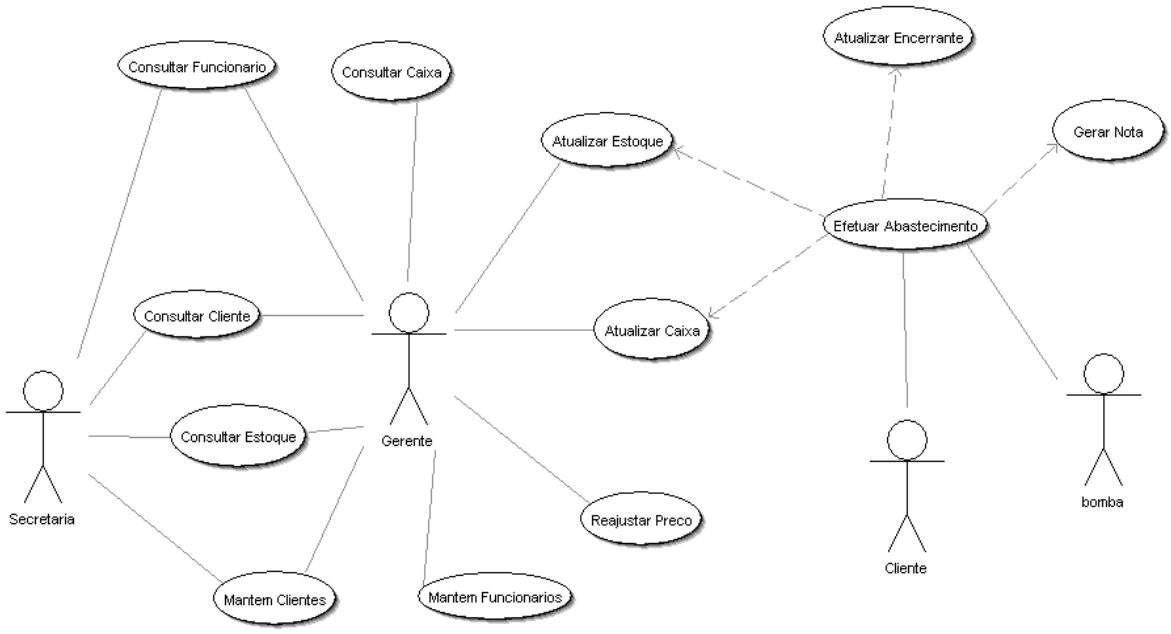

Figura 11 – Diagrama de Caso de Uso

Caso de Uso
Nome: Atualizar Caixa
Ator Principal: Bomba
Objetivo: Atualizar o caixa após o abastecimento
Numero no Documento de Requisitos: 1

O cliente abastece o automóvel.
A bomba gera as informações do abastecimento (litros, combustível, valor) para o sistema.
O cliente efetua o pagamento.
O sistema atualiza o saldo do caixa.

Figura 12 – Descrição do Caso de Uso Atualizar Caixa

Após a identificação e especificação dos atores e casos de uso, Camargo (2006) propõe a elaboração de uma tabela denominada "Tabela de Correspondência Caso de Uso-Requisito". Essa tabela é utilizada para registrar os casos de uso identificados até o momento e os requisitos que deram origem a esses casos. A tabela deve ser atualizada à medida que sejam identificados novos casos de uso no decorrer do desenvolvimento. Na tabela, a coluna

“É proprietário do” corresponde a um relacionamento de propriedade entre o caso de uso e o requisito. A coluna “Colabora com” lista os casos de uso que possuem um relacionamento de colaboração com o caso de uso em questão. A coluna D.C.U informa quais são os casos de uso que estão representados no diagrama de casos de uso, e não é obrigatória.

Esse artefato é de suma importância para o desenvolvimento de software já que neste ponto se tem uma visão geral do sistema identificando os atores e suas principais funções.

Na Tabela 8 são mostrados o relacionamento entre os casos de uso e requisitos do sistema de automação de postos. Foi identificado que o Caso de Uso Efetuar Pagamento colabora com o requisito 8 (impressão da nota fiscal).

Tabela 8 - Tabela de Correspondência Caso de Uso/Requisitos do Sistema de Automação

Nº	Descrição	É propriedade de	Colabora com	D.C.U
1	Atualizar Caixa	Req. 1		☑
2	Atualizar Estoque	Req. 1,2		☑
3	Atualizar Encerrante	Req. 1		☑
4	Controlar Caixa	Req. 3		☑
5	Reajustar Preço	Req. 4		☑
6	Mantém Funcionários	Req. 5		☑
7	Mantém Clientes	Req. 6		☑
8	Efetuar Pagamento	Req. 7	Req. 8	☑
9	Emitir Nota	Req. 8		☑
10	Consultar Estoque	Req. 9		☑
11	Consultar Caixa	Req. 10		☑
12	Consultar Funcionários	Req. 11		☑
13	Consultar Clientes	Req. 12		☑

5.2.1.2 Identificar e Especificar Casos de Uso Não-Funcionais

Neste ponto do processo são identificados e especificados os Casos de Uso Não-Funcionais, tomando como base o documento de requisitos. O foco desta atividade é apenas a identificação dos interesses não-funcionais, sem levar em consideração detalhes de interface e/ou arquitetura, pois isso será tratado em fases posteriores. Na elaboração do documento de requisitos do sistema de automação de posto foi adotado um formato onde existe uma seção específica que descreve os requisitos Não-funcionais tornando mais fácil a identificação dos mesmos.

A identificação dos Interesses Transversais se dá por meio do conhecimento que o engenheiro de software possui. No caso do sistema de automação os interesses identificados são tidos como exemplos clássicos na literatura. Na Tabela 9 são mostrados os Interesses Transversais identificados e os requisitos que são afetados por eles.

Tabela 9 – Tabela de Identificação de Interesses Não-Funcionais do sistema de automação de postos

Interesses Não-Funcionais	Requisitos Cobertos
Gerenciar Transação	1, 2, 3, 4, 5, 6
Autenticar Usuário	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
Criptografar Informações	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
Persistir Dados	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
Controle Acesso	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
Registro Acesso	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12

Após a identificação dos Interesses Não-Funcionais, é feito um refinamento da descrição através de entrevistas com o responsável pelo sistema ou revisando o documento de requisitos. O objetivo desse refinamento não é obter uma descrição detalhada dos interesses e sim refinar a descrição original para um melhor entendimento do seu comportamento.

Para a documentação da descrição foi utilizada uma tabela denominada “Tabela de Descrição de Casos de Uso Não-Funcionais”, cujo objetivo é explicar o funcionamento de cada caso de uso Não-Funcional. O ideal para a elaboração dessa descrição é a realização de entrevistas com o responsável pelo sistema já que muitos detalhes são difíceis de serem obtidos apenas por meio do documento de requisitos. Na Tabela 10 é mostrado a Descrição para o caso de uso não-funcional Autenticar Usuário do sistema de automação de postos.

Tabela 10 – Descrição dos Casos de Uso Não-Funcionais do sistema de automação de postos

Caso de Uso não Funcional:	Autenticar Usuário
Caso de Uso Relacionados:	Atualizar Caixa; Atualizar Estoque; Atualizar Encerrante; Controlar Caixa; Reajustar Preço; Mantém Funcionários; Mantém Clientes; Efetuar Pagamento; Emitir Nota; Consultar Estoque; Consultar Caixa; Consultar Funcionários; Consultar Clientes
Descrição Original:	O sistema deve possuir senhas de acesso e identificação para diferentes tipos de usuários: administrador do sistema e funcionários do estabelecimento. Uma restrição do sistema de segurança é que o erro consecutivo de três tentativas de acesso bloqueia a conta do usuário.
Descrição Refinada:	Para que o usuário tenha acesso as funções do sistema é necessário que o mesmo informe um usuário e senha que deve ser previamente cadastrado. O erro consecutivo de três tentativas bloqueia o usuário não permitindo mais o seu acesso ao sistema.

Após a elaboração da tabela de descrição, inclui-se o caso de uso em questão no diagrama de caso de uso. Aconselha-se a elaboração de visões individuais para melhorar a legibilidade do modelo. Na Figura 13 é mostrado um exemplo do sistema de automação em que foi criada uma visão para o requisito não-funcional Persistir Dados, mostrando quais casos de uso são afetados por ele.

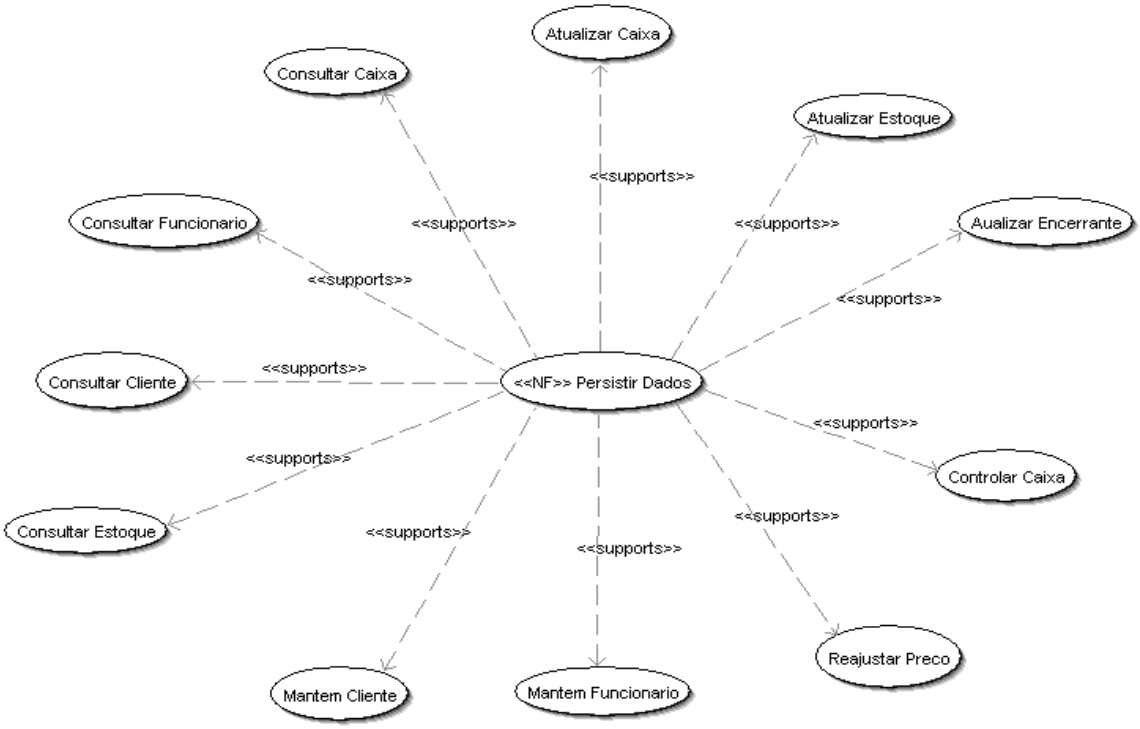

Figura 13 – Visão do Caso de Uso Persistir Dados

5.2.1.3 Planejar Iterações

Esta atividade tem como objetivo planejar o número de iterações que serão realizadas no decorrer do desenvolvimento. Os casos de uso relacionados com a funcionalidade principal do sistema devem ser implementados nas primeiras iterações, a idéia é que a cada iteração uma funcionalidade do sistema seja implementada por completo.

O processo de priorização de casos de uso pode ser feito subjetivamente, pelo engenheiro de software responsável pelo sistema, ou seguindo algum processo de priorização um pouco mais sistemático, por exemplo, a técnica AHP(*Analytic Hierarchy Process*) (Saaty, 1980).

Para a identificação do caso de uso principal do sistema de automação para postos, foi utilizada a seguinte questão “Qual a principal funcionalidade do sistema”, no caso a principal funcionalidade é o abastecimento, logo a primeira iteração implementará os casos de usos relacionados com Efetuar Abastecimento.

Na Tabela 11 são mostradas as iterações estipuladas para o sistema de automação de postos e os casos de uso que foram alocados para cada uma.

Tabela 11 - Tabela de Planejamento das Iterações do Sistema de Automação

	Iterações	Classificação	Caso de Uso
ELABORAÇÃO	1º Iteração	1	Persistir Dados
		2	Atualizar Caixa
		3	Atualizar Estoque
		4	Atualizar Encerrante
		5	Efetuar Abastecimento
	2º Iteração	6	Reajustar Preço
		7	Efetuar Pagamento
		8	Emitir Nota
	3º Iteração	9	Mantém Funcionários
		10	Mantém Clientes
CONSTRUÇÃO	1º Iteração	11	Autenticar Usuário
		12	Criptografar Informações
	2º Iteração	13	Consultar Estoque
		14	Consultar Caixa
		15	Consultar Funcionários
		16	Consultar Clientes
	3º Iteração	17	Controlar Acesso
		18	Registrar Acesso
		19	Gerenciar Transação

Como o ProFT/PU usa o conceito de desenvolvimento incremental deve ser desenvolvido o artefato de “visão de iteração”. Na Figura 14 é mostrado o diagrama com a visão da primeira iteração.

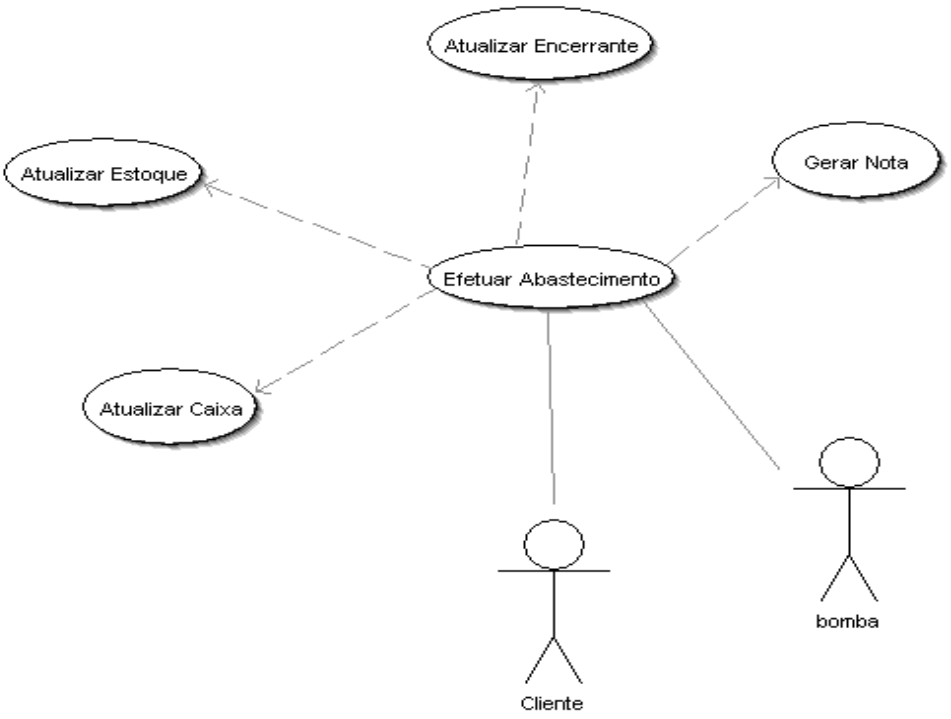

Figura 14 – Visão da 1ª Iteração

5.3 Fase de Elaboração – 1ª. Iteração

Essa fase se concentra no projeto e implementação dos casos de uso com maior prioridade do sistema, com o objetivo principal de implementar uma porção do sistema que esclareça os detalhes arquiteturais e de maior risco.

Na primeira iteração da fase de elaboração ainda há grande ênfase nos requisitos, já que muitos ainda não foram descobertos e registrados (Camargo, 2006). Na figura 15 são mostradas as atividades da fase de elaboração do ProFP/PU.

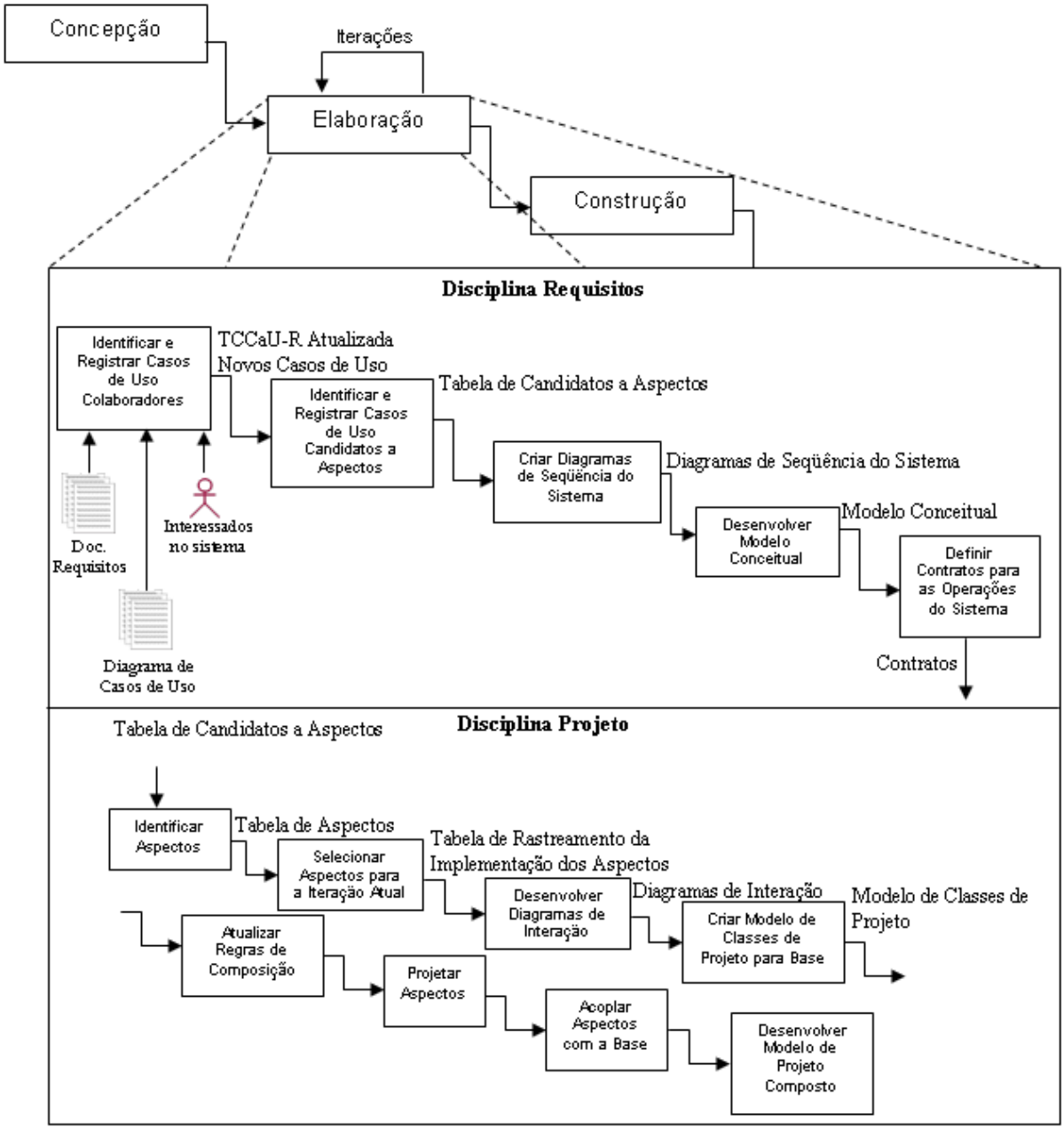

Figura 15 - Atividades da Fase de Elaboração retirado de Camargo, 2006

5.3.1 Disciplina Análise

Nesta fase, o objetivo é refinar os casos de uso escolhidos para esta iteração, descobrir novos requisitos e elaborar o diagrama de classe.

5.3.1.1 Identificar e Registrar Casos de Uso Colaboradores

Nessa atividade o objetivo é identificar casos de uso que não são identificados facilmente por inspeção do documento de requisitos, que são os casos de uso de níveis mais baixos, também são responsabilidades desta atividade aumentar o nível de detalhes do caso de uso tornando mais fácil a compreensão de sua funcionalidade, outro objetivo desta fase é alcançar um bom nível de modularização do sistema.

Para a identificação desses casos de uso, deve-se fazer uma análise na descrição do caso de uso escolhido para esta iteração e identificar possíveis trechos que são funcionalidades que mereçam ser transformadas em possíveis casos de uso. Algumas perguntas auxiliam a descobrir se um trecho de um caso de uso merece ser transformado em um novo caso de uso colaborador (Camargo, 2006). Na Tabela 12 é mostrado o Modelo de Decisão, que é uma ferramenta particular desta atividade.

Tabela 12 – Tabela de Decisão para Identificação de Novos Casos de Uso

No.	Pergunta	Resposta	Decisão (Deve Fatorar?)
1	A funcionalidade descrita pelo trecho é utilizada pelos outros casos de uso disponíveis na visão de iteração?	Sim	Sim
		Não	Pergunta 2
2	A funcionalidade descrita pelo trecho é essencial para o comportamento do caso de uso?	Sim	Não
		Não	Sim

No sistema de Automação de Postos não foi identificado nenhum Caso de Uso colaborador, não precisando atualizar a “visão” da primeira iteração, nem a Tabela de Correspondência Caso de Uso/Requisitos, vista anteriormente.

5.3.1.2 Identificar e Registrar Casos de Uso Candidatos a Aspectos

O objetivo dessa atividade é identificar possíveis casos de uso que apresentam indícios que sua implementação poderá ser feita utilizando a técnica de orientação a aspectos. Os casos de uso identificados nessa atividade são denominados “candidatos a aspecto”, por ainda não ter certeza se essa tecnologia será utilizada para a sua implementação. A decisão será feita durante a disciplina de projeto.

Quatro critérios são utilizados para a identificação dos possíveis “candidatos a aspectos” sendo eles:

- Quantidade de relacionamento onde são selecionados os casos de uso que possuem relacionamento com mais de um caso de uso, provavelmente a implementação desse caso de uso possuirá trechos de código espalhados por vários locais do sistema. São aqueles que são incluídos por dois ou mais casos de uso (relacionamento <<*include*>>), que estendem dois ou mais casos de uso (relacionamento <<*extend*>>) ou que restringem dois ou mais casos de uso (relacionamento <<*constraint*>>).

- Tipo do caso de uso onde são selecionados os casos de uso do tipo não-funcional, isto é, aqueles que possuem estereótipo <<NF>>. Essa identificação é facilitada já que esse processo possui a atividade Identificar e Especificar Casos de Uso Não-Funcionais realizada na fase de concepção,basta apenas verificar se os casos de uso alvo desta iteração se encontram na Tabela de Identificação de Interesses Não-Funcionais.

- Extensão onde são selecionados os casos de uso que se relacionam por extensão (<<*extend*>>) com um ou mais casos de uso. Esses casos de uso são selecionados por o comportamento dele ser funcionalidades extras que são adicionadas ao comportamento de um caso de uso base, e pode ser que tratem de funcionalidades adequadas para a implementação com aspectos.

- Essencialidade onde é visado identificar se o comportamento do caso de uso transversal é essencial ao comportamento genérico do caso de uso base, ou se é apenas uma adição de funcionalidade que é exclusiva do sistema que está sendo desenvolvido, geralmente relacionamentos de extensão se enquadram nessa situação.

Os quatros critérios de identificação de “candidatos a aspectos” são independentes, um não sofre influencia do outro, assim se algum caso de uso satisfazer algum critério ele pode ser considerado um caso de uso “candidato a aspecto”.

Para apoio dessa atividade é utilizada uma tabela denominada Tabela de Candidatos a Aspecto, onde é dividida em quatro colunas. Na primeira coluna é colocado o nome do caso de uso, na segunda seu tipo (funcional/não-funcional), na terceira o critério utilizado, a quarta coluna é reservada para as atualizações que ocorrerão nas próximas iterações. Se algum caso de uso candidato a aspecto for implementado com essa tecnologia a caixa dessa coluna deve ser marcada.

Para o sistema de automação foram identificados como candidatos a aspecto dois casos de uso: Persistir Dados e Registrar Operações (*log*), como pode ser visto na Tabela 13.

Tabela 13 – Tabela de Candidatos a Aspectos do Sistema de Automação de Postos

Caso de Uso	Tipo (F/NF)	Critério	Implementado
Persistir Dados	Não-funcional	Número de relacionamentos e Tipo	☐
Registrar Operações	Não-funcional	Tipo	☐

Esta atividade é apenas um apoio a uma das fases mais importantes do desenvolvimento de software orientado a aspectos, que é a identificação de possíveis casos de uso que podem ser implementados com a tecnologia. Existem poucos documentos na literatura que apóiam a identificação de aspectos, ainda depende dos conhecimentos da equipe de desenvolvido para a sua identificação.

Para o sistema de automação de postos a princípio não foi identificado nenhum caso de uso candidato a aspecto, a não ser os casos de uso não-funcionais. Persistir Dados e Registrar Operações como pode ser visto na Tabela 5 que são tidos como exemplos clássicos na literatura, mas isso não significa que outros casos de uso candidatos não possam ser descobertos durante a realização de outras atividades.

5.3.1.3 Criar Diagramas de Seqüência do Sistema

Nesta atividade é criado o diagrama de seqüência do sistema (DSS) para cada caso de uso da iteração. O DSS auxilia a identificação das principais operações do sistema e representa o comportamento do sistema em resposta a eventos gerados por entidades externas.

O diagrama de seqüência do caso de uso Efetuar Abastecimento é mostrado na Figura 16.

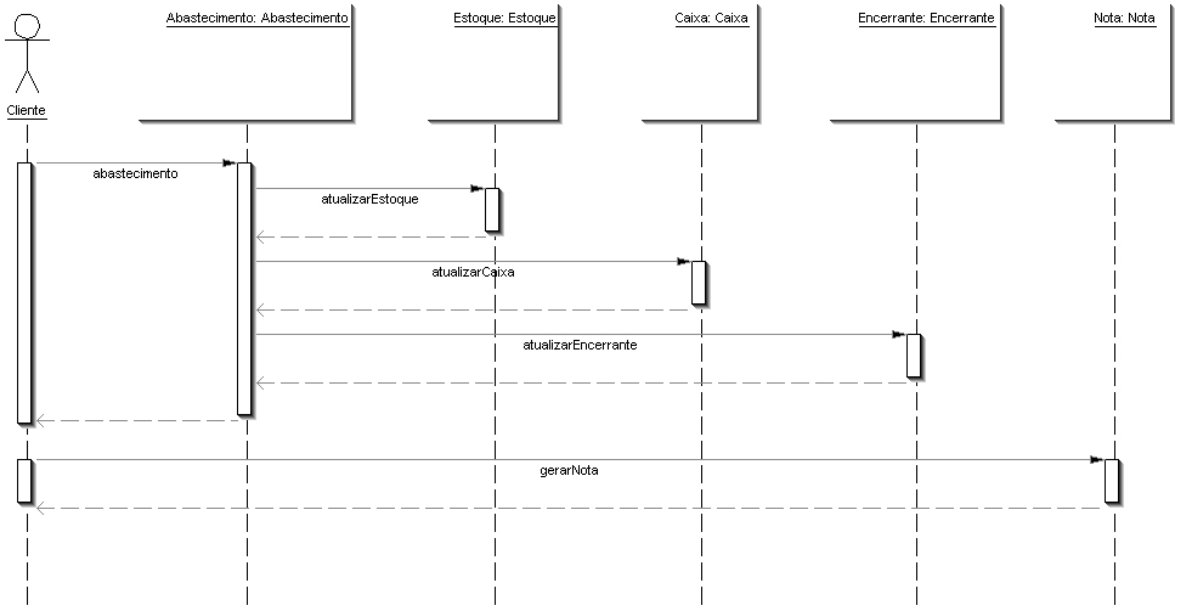

Figura 16 – Diagrama de Seqüência

5.3.1.4 Desenvolver Modelo Conceitual

A essa atividade cabe o desenvolvimento do Modelo Conceitual que devem se concentrar apenas nos casos de uso que estão sendo tratados nesta iteração. Analisando os casos de uso foi obtido o modelo conceitual mostrado na Figura 17.

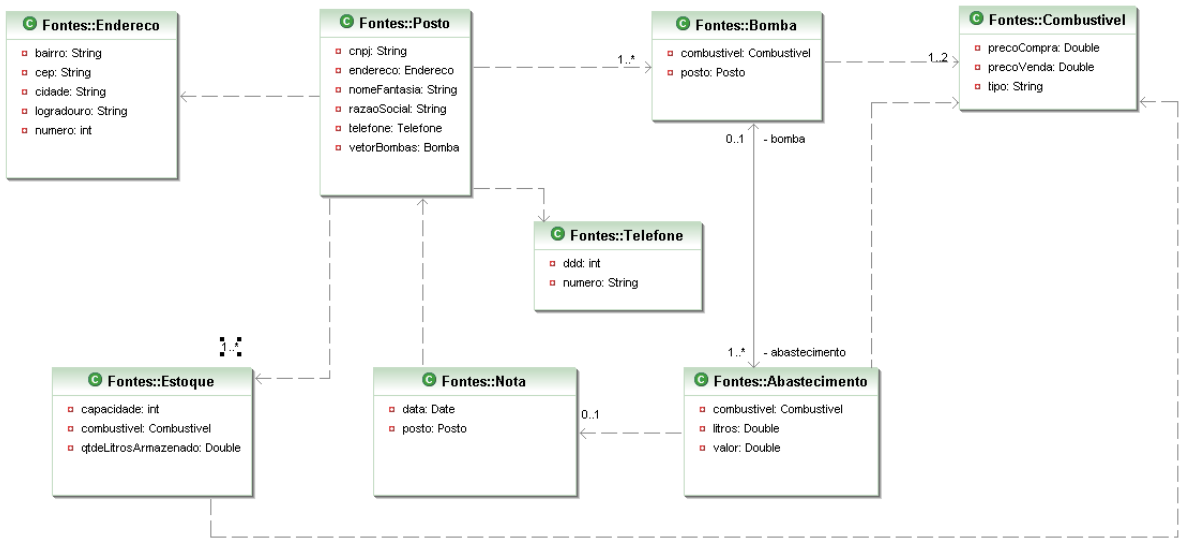

Figura 17 – Modelo Conceitual do Sistema de Automação de Postos

5.3.2 Disciplina Projeto

O objetivo desta disciplina é criar um modelo de classes de projeto, modelos de interação, e quaisquer outros modelos que auxiliem a documentar e registrar o comportamento e a estrutura do sistema. Na Tabela 14 é mostrada as atividades da disciplina de projeto.

Tabela 14 – Atividades da Disciplina Projeto

Atividades da Disciplina Projeto
Identificar Aspectos
Selecionar Aspectos para a Iteração Atual
Desenvolver Diagramas de Interação
Desenvolver Modelo de Classes de Projeto
Projetar Aspectos
Acoplar Aspectos com a Base
Desenvolver Modelo de Projeto Composto
Atualizar Regras de Composição

5.3.2.1 Identificar Aspectos

Nesta atividade o objetivo é identificar quais casos de uso listados na Tabela de Candidatos a Aspectos (Tabela 13) devem ser implementados com a tecnologia. Para realizar a análise de quais casos de uso vão ser implementados com aspectos, utilizam-se os artefatos: modelo de casos de uso da iteração atual, a descrição dos casos de uso, o documento de requisitos e se necessário o diagrama de iteração. Se um caso de uso for identificado como deve ser implementado com aspecto isso não quer dizer que sua implementação será feita na iteração atual podendo ser adiada para as próximas iterações.

Para identificar se um caso de uso candidato a aspecto realmente deve ser implementado com tal paradigma, alguns critérios devem ser verificados. O primeiro critério consiste em averiguar se o caso de um é um requisito não-funcional. Em caso afirmativo, a sugestão é que ele seja implementado com aspectos, entretanto a decisão final pode ser influenciada por outros fatores, como por exemplo a existência de um *framework* orientado a objetos, o que consumiria menos tempo de implementação, ficando a cargo da equipe de desenvolvimento a decisão de como será implementado.

O segundo critério que o ProFT/PU utiliza é uma adaptação do trabalho de Clarke e Baniassad (2002), esse critério só pode ser aplicado nos casos de uso transversais que possuem mais de um relacionamento. O objetivo é verificar se o caso de uso é atômico ou não, ou seja, se ele manipula apenas uma funcionalidade. O critério de atomicidade tem o objetivo de verificar se o requisito que originou o caso de uso transversal não pode ser dividido. Para isso a Tabela de Correspondência Caso de Uso/Requisitos (Tabela 8) deve ser analisada para verificar qual foi o requisito que originou no caso de uso transversal. Deve-se tentar dividir a sentença textual originadora do requisito para eliminar o compartilhamento.

O terceiro critério é o de volatilidade, neste critério cabe avaliar se o requisito tratado pelo caso de uso transversal é volátil. O requisito é classificado como volátil quando representa uma regra de negócio onde os interessados no sistema desejam mudar rapidamente, dependendo das demandas do mercado. Exemplos de requisitos voláteis são: "se o valor das movimentações financeiras de determinado cliente durante o ano for superior a quinhentos mil dólares, eles receberão um prêmio", ou " Os dez clientes que mais gastarem na loja durante a semana ganharão cinco por cento de desconto na próxima compra" (Camargo, 2006).

Quando os requisitos voláteis são implementados da forma tradicional eles geralmente se tornam fixos no sistema dificultando a sua manutenção. Segundo Moreira e Araújo (2004), a implementação de requisitos voláteis com aspectos é uma estratégia interessante porque facilita a inclusão/remoção/alteração desses requisitos depois que o sistema estiver pronto.

Para a identificação dos requisitos voláteis, deve-se analisar a descrição do caso de uso tentando identificar regras de negócios com tendência a mudanças. Essa identificação não é muito trivial, e é necessário que a equipe de desenvolvimento esteja a par das regras de negócio que envolve o sistema em questão.

O quarto e último critério procura identificar se há um FT (*Framework* Transversal) já desenvolvido que possa ser utilizado na implementação do caso de uso em questão. Esse critério é de muita importância já que a ausência de um FT pode inviabilizar a implementação do caso de uso utilizando a tecnologia de aspecto, por causa da dificuldade que um novo paradigma traz a uma equipe de desenvolvimento.

Os quatro critérios vistos acima devem ser aplicados para cada caso de uso na Tabela de Candidatos a Aspectos. Para a aplicação dos critérios deve-se criar uma tabela denominada "Tabela de Aspectos", essa tabela contém um parecer final que opta ou não pela

implementação com aspectos. A Tabela 15 mostra os casos de uso que foram selecionados para a implementação com aspectos.

Tabela 15 – Tabela de Aspectos do Sistema de Automação

Casos de Uso	Tipo	Atomicidade	Volatilidade	Framework Transversal	Implementa como Aspecto?	Implementado
Persistir Dados	NF	-	-	-	sim	☐
Registrar Operações	NF	-	-	-	sim	☐

No sistema de automação de postos foram identificados dois casos de uso candidatos a aspecto: Persistir Dados e Registrar Operações, por eles serem exemplos clássicos de aspectos e por existir um *framework* transversal, os demais critérios não foram aplicados para decidir que serão implementados com aspecto.

5.3.2.2 Selecionar Aspectos para a Iteração Atual

Nesta atividade é selecionado o aspecto da tabela de Aspecto que deve ser implementado na iteração atual. Para o sistema de automação de postos foi selecionado o aspecto Persistir Dados, para a decisão foi levado em consideração o fato de as operações para realizar um abastecimento estarem ligadas ao conceito de persistência.

Camargo (2006) utiliza para a documentação dos aspectos que serão implementados uma tabela denominada "Tabela de Rastreamento de Implementação de Aspectos", onde são listado os aspectos selecionados para serem implementados na iteração atual. Na Tabela 16 mostra-se os aspectos selecionados para o sistema de automação de postos.

Tabela 16 – Tabela de Rastreamento da Implementação de Aspectos para o Sistema de Automação de Postos

No. Iteração	Aspecto	F/NF
1ª. Da elaboração	Persistir Dados	NF

5.3.2.3 Projetar Aspectos Não-Funcionais

O objetivo desta atividade é desenvolver o modelo de projeto (diagramas de classes e de interação) para os aspectos não-funcionais alocados para iteração. Como para essa iteração o sistema de automação alocou somente o requisito não-funcional persistir dados e esse

aspecto possui um FT que será utilizado na sua implementação, não tem a necessidade de desenvolver o modelo de projeto para ele.

O FT que será utilizado no sistema de automação foi desenvolvido por Camargo (2006), e implementa as operações básicas de persistência e a parte de conexão com o banco, dentro do pacote do FT existe o diagrama de seqüência que detalham o comportamento deste FT.

5.3.2.4 Identificar Classes Persistentes da Aplicação

Essa atividade tem como objetivo identificar as classes da aplicação que devem possuir correspondência com tabelas do banco de dados e que devem ter seus objetos persistidos nessas tabelas. Essa identificação é de muita importância para o FT, pois são nelas que as operações de persistência serão colocadas.

As classes identificadas no sistema de Automação de Postos para essa iteração foram: Posto, Nota, Estoque, Combustível, Abastecimento, Bomba.

5.3.2.5 Adição de Atributos e Métodos de Acesso

Após as classes persistentes serem identificadas, o próximo passo é inserir em cada uma delas um atributo `ID` do tipo `int`. Esse atributo é exigido quando o FT de persistência é utilizado. Além disso, todos os atributos da classe devem possuir um método `set` e um `get` correspondente. Para que o método `setID()` de cada classe funcione corretamente é necessário que o parâmetro passado para o método seja do tipo `Integer`, essa é uma restrição imposta pelo FT de persistência. Uma outra condição imposta pelo *framework* é que cada classe terá que possuir um método construtor.

5.3.2.6 Projetar Aspectos Funcionais

Esta atividade consiste basicamente em projetar os aspectos funcionais selecionados para essa iteração. As tarefas realizadas nessa atividade são: desenvolver diagramas de classes de projeto e diagramas de interação.

No sistema de automação de postos não foi identificado nenhum aspecto funcional, portanto, não foi realizada nenhuma tarefa nessa atividade.

5.3.2.7 Desenvolver Modelo de Classes de Projeto

Essa atividade tem como objetivo a elaboração do modelo de classe de projeto do sistema de Automação de Postos. A elaboração foi realizada com cuidado levando em consideração a utilização de um FT de persistência que impõe certas restrições à arquitetura do sistema e, se essas restrições não forem consideradas no modelo de classes de projeto, o acoplamento do FT de persistência, seja neste ponto do processo ou no final, pode exigir mudanças de projeto significativas.

Para o desenvolvimento do sistema de Automação de Postos a decisão foi utilizar o FT desde o início, isto é, em todas as iterações que ele for necessário, com isso uma pequena versão do sistema será obtida já com a infra-estrutura real que será utilizada no sistema final. O modelo de classes de projeto do sistema de Automação pode ser visto na Figura 18.

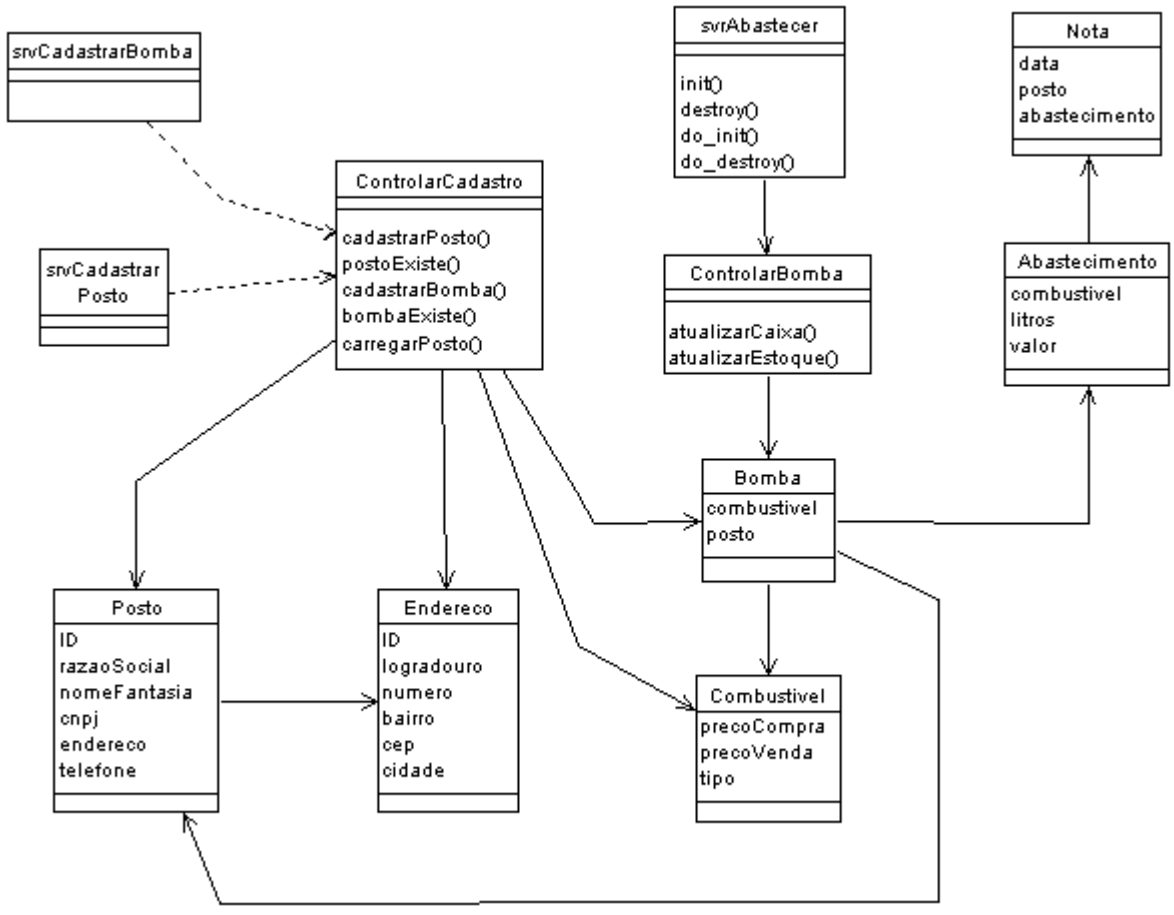

Figura 18 – Modelo de Classe de Projeto

5.3.2.8 Acoplar Aspectos com a Base

O objetivo dessa atividade é planejar o acoplamento dos aspectos identificados com a parte base do sistema. Segundo Camargo (2006), essa atividade consiste em desenvolver um modelo de projeto combinado, que una o modelo da parte base que foi desenvolvida até este momento com os modelos de projeto dos aspectos desenvolvidos na atividade anterior. Esse modelo de projeto combinado é o modelo de projeto final da porção do sistema que está sendo tratada na iteração atual.

No caso do sistema de Automação de postos será utilizado um FT de persistência desenvolvido por Camargo *et al.*, (2005), visto anteriormente. Nesse caso o processo consiste no reúso do FT, o qual possui a etapa instanciação e composição.

Segundo Camargo (2006), o FT de persistência possui partes bem distintas que podem ser acopladas em tempos diferentes a um código-base. A primeira é a persistência em si, que consiste em um conjunto de operações de armazenamento/consulta/remoção que são introduzidas nas classes da camada de domínio e que são utilizadas pelas classes da camada de interface e classes controladoras. A segunda parte que trata da conexão com o banco de dados, que exige a definição de pontos de código-base onde a conexão deve ser aberta e fechada.

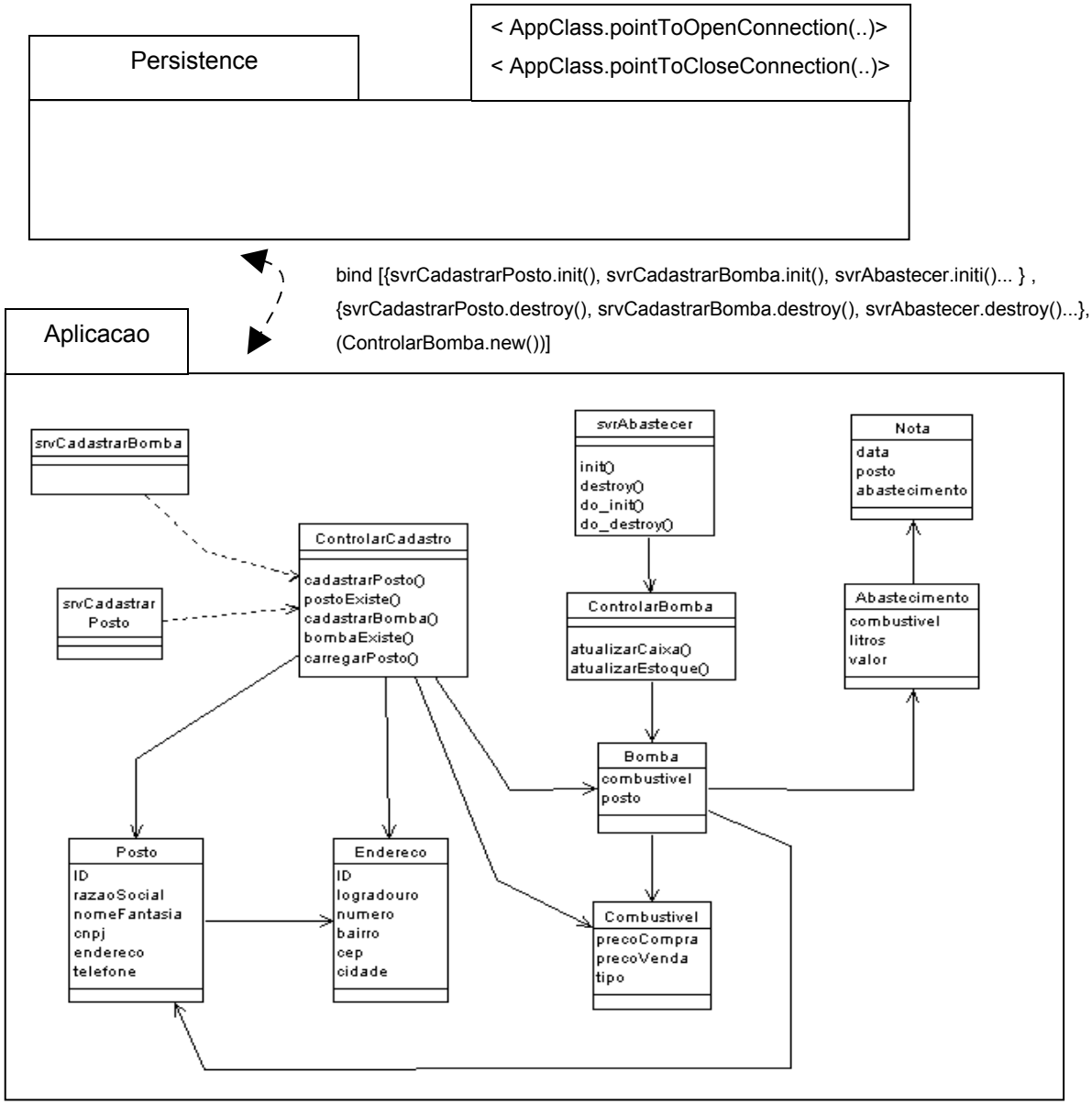

Figura 19 - Visão de Acoplamento do FT de Persistência com o Sistema de Automação de Postos

5.3.2.9 Projetar Modelo de Dados

Segundo Camargo (2006), essa tarefa pode auxiliar a garantir que o desenvolvimento do sistema não esteja fugindo das restrições arquiteturais impostas pelo FT de persistência, para isso o modelo de banco de dados é projetado em paralelo com o modelo de classes de projeto. A Figura 20 representa o modelo de dados criado para a porção do sistema em desenvolvimento

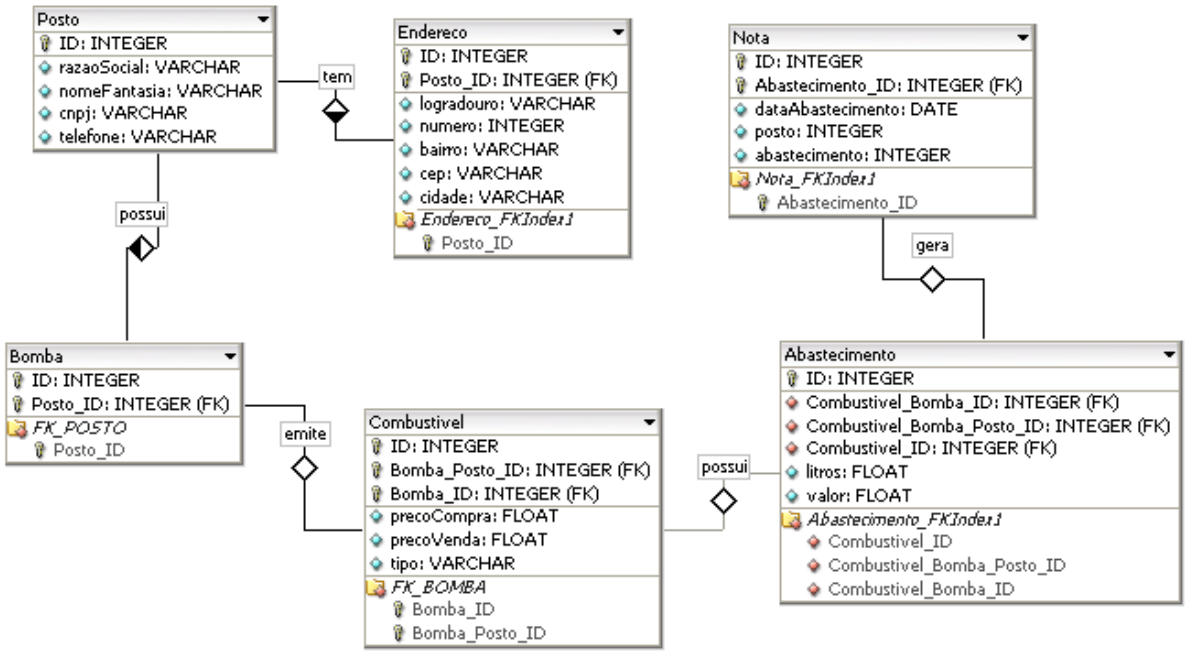

Figura 20 - Modelo de Dados

5.3.2.10 Desenvolver Visão Geral da Composição

Essa atividade tem como objetivo a elaboração de um diagrama em alto nível de abstração que represente os aspectos e FTs acoplados à porção do sistema projetada até este momento. Esse modelo deve ser constantemente atualizado à medida que novos aspectos são adicionados ao sistema durante o decorrer das iterações.

O modelo é denominado Visão Geral de Composição. Consiste em um diagrama de pacotes em que um pacote representa a aplicação que está sendo desenvolvida e os demais pacotes representam aspectos e FTs. (Camargo, 2006).

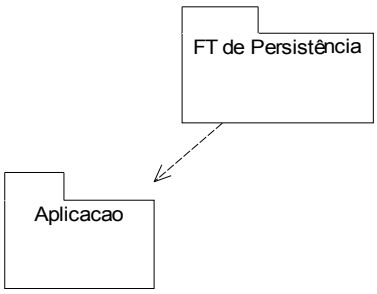

Figura 21 – Visão Geral de Composição

5.3.2.11 Registrar Pontos de Junção

Essa atividade tem como objetivo identificar os pontos de junção onde os aspectos utilizados no sistema vão atuar. Camargo (2006), propôs uma tabela para a elaboração do Registro dos Pontos de Junção, onde na primeira coluna devem ser colocados os FTs ou aspectos, e na segunda, terceira e quarta colunas devem ser discriminados os pontos de junção afetados por cada um. Podem ser utilizados caracteres coringa para representação dos pontos de junção, como por exemplo o asterisco (*) e os dois pontos horizontais (..). Dessa forma, aspectos homogêneos que afetam um número grande de pontos de junção podem ser representados de forma mais legível. A Tabela também deve ser utilizada para registrar pontos de junção de aspectos ou FTs afetados por outros aspectos ou FTs.

Apesar da elaboração da tabela seja feita na disciplina de projeto, ela deve ser atualizada durante a implementação, pois é só durante essa disciplina que os pontos de junção identificados aqui são utilizados, podendo haver modificações significativas ou ainda pode ser descobertos novos pontos de junção.

A Tabela 17 mostra o Registro dos Pontos dos Pontos de Junção do Sistema de Automação de Postos.

Tabela 17 - Registro dos Pontos de Junção

FT/Aspecto ⬇	**Pontos de Junção Afetados**		
	Método/atributo/constructor/adendo	***Unidades de Software***	
		Classe	**Aspecto**
FT de Persistencia	init() init() init() destroy() destroy() destroy() new()	srvCadastroBomba srvCadastroPosto srvAbastecer srvCadastrarPosto srvCadastrarBomba ControlarBomba	

5.3.3 Disciplina Implementação

Na Disciplina de Implementação os modelos de projetos são traduzidos em código fonte. Nessa atividade muitos detalhes de projeto serão descobertos, e a medida que esses detalhes vão surgindo os modelos obtidos até o momento devem ser atualizado.

5.3.3.1 Implementar Classes de Domínio

Essa fase é a fase inicial da implementação do sistema, onde as classes de domínio, com seus atributos e métodos de acesso (`set´s e get´s`) são construídas. Segundo Camargo (2006), os métodos que precisam realizar armazenamento, recuperação e remoção no mecanismo persistente só poderão ser implementados depois que o FT de persistência introduzir suas operações em cada uma das classes.

5.3.3.2 Implementar Banco de Dados

Neste ponto do processo é implementado o banco de dados, de forma convencional traduzindo o modelo de dados em tabelas de bancos de dados. Como o desenvolvimento do banco não é trivial e envolve o conhecimento de certas regras, visto que neste processo o foco não é este, fica a cargo do desenvolvedor decidir qual a melhor forma de desenvolvimento para o seu sistema.

5.3.3.3 Implementar Aspectos

Esta atividade do processo consiste na implementação dos aspectos selecionados para esta iteração. Como para o sistema de Automação de Postos foi identificado apenas o Aspecto de Persistência, e será utilizado um *framawork* de persistência, esta fase consiste apenas no reúso do *framework*.

As próximas atividades do processo descrevem como é feito o acoplamento do FT de persistência com a aplicação tomando como base o trabalho escrito por Camargo (2006).

5.3.3.4 Acoplar a Persistência

Para o acoplamento do FT com a aplicação é criado um aspecto que estende o aspecto OORelationalMapping, do FT. Na Figura 22 segue o aspecto para o sistema de Automação de Postos.

```
public aspect MyPersistentEntities extends PersistentEntities {

      declare parents : Cliente      implements PersistentRoot;
      declare parents : Bomba        implements PersistentRoot;
      declare parents:  Combustivel  implements PersistentRoot;
      declare parents:  Estoque      implements PersistentRoot;
      declare parents:  Nota         implements PersistentRoot;
      declare parents:  Posto        implements PersistentRoot;

}
```

Figura 22 – Aspecto MyPersistentEntities do Sistema de Automação de Postos

5.3.3.5 Desenvolver Classes Controladoras

Esta atividade tem como objetivo implementar as classes controladoras e suas operações. Segundo Camargo (2006), o modelo de classes de projeto é a base para a implementação das classes controladoras e os diagramas de colaboração e os casos de uso são a base para a implementação das operações e métodos de granularidade mais fina. Neste ponto o FT de persistência já esta acoplado à porção do sistema desenvolvido, portanto, as operações de sistema podem ser implementadas considerando a existência das operações de persistência disponibilizadas pelo *framework*.

5.3.3.6 Desenvolver Interfaces

O objetivo dessa atividade é implementar as classes de interface para o sistema que está sendo desenvolvido. Para o sistema de Automação deve-se desenvolver uma interface para a realização do abastecimento e para o cadastro do posto e cadastro da bomba.

5.3.3.7 Instanciar Conexão

Para a instanciação da parte de conexão do *framework* de persistência Camargo (2006), propõe o uso de uma classe que estenda uma das classes de variabilidade do FT. Na Figura 23 é mostrado o trecho de código que estende a classe OdbcConnection, que indica

que a conexão será via ODBC do Windows, no caso do sistema em questão será utilizado o banco de dados mySQL e o nome da conexão criada foi “AutoPosto”.

```
public class myConnectionVariabilities extends OdbcConnection
{

   public String setSpecificDatabase(){
            return "mysql";
   }

   public String setDSN(){
        return "AutoPosto";
   }
```

Figura 23 – Aspecto myConnectionVariabilities

5.3.3.8 Acoplar a Conexão

O aspecto mostrado na Figura 35 realiza o acoplamento propriamente dito. Esse aspecto está considerando que a classe de interface `srvFazerReserva` e a classe controladora `srvCadastro` já foram, pelo menos, parcialmente implementadas. Sendo assim, os pontos de junção identificados são os pontos reais de acoplamento, isto é, os pontos de junção em que o FT permanecerá acoplado depois que o sistema estiver em produção.

5.3.3.9 Testar Sistema

Nessa atividade a porção do sistema desenvolvida até este momento é testada. Após o teste, é finalizado a primeira iteração do processo. Neste ponto o software em questão já terá sua parte base funcionando. O próximo passo é repetir todas as fases descritas anteriormente para a segundo iteração e assim sucessivamente até que o software seja finalizado.

6 COMPARAÇÃO COM ABORDAGENS EXISTENTES

Neste capitulo é realizada uma comparação preliminar entre a proposta de Camargo, (2006), e as dos demais autores comentados no Capitulo 3. Kassab *et al.* (2005) e Camargo (2006) propuseram uma abordagem completa para o desenvolvimento de software orientado a aspectos, em que dentro de cada abordagem existe uma fase para a identificação de interesses transversais, enquanto Araújo *et al.* (2002) e Rashid *et al.* (2002), apenas descrevem a identificação de interesses transversais e Jacobson (2003) apresenta uma idéia para o desenvolvimento de software orientado a aspectos.

A abordagem proposta por Kassab *et al.* (2005) e Camargo (2006), permite a identificação e o rastreamento de interesses transversais ao longo do processo de desenvolvimento. Dentro de cada método há uma fase onde são identificados os interesses transversais. Tanto a abordagem proposta por Kassab *et al*. (2005) quando a proposta por Camargo (2006), são incrementais e possuem fases, sendo cada fase com suas atividades direcionam e auxiliam o desenvolvimento. O modelo proposto por Camargo (2006) é composto por quatro fases (Concepção, Elaboração, Construção, Transição), onde o sistema é desenvolvido em iterações, em que em cada iteração é desenvolvido um subconjunto do sistema até que o sistema todo seja implementado. Já o modelo proposto por Kassab *et al.* (2005), é composto por cinco fases que se inicia no planejamento e segue até a fase de desenvolvimento onde é identificado e detalhado cada caso de uso utilizando técnicas como matrizes para relacionamento entre os requisitos funcionais e não-funcionais, diagramas de caso de uso e diagramas de domínio. Nas demais abordagens pesquisadas para a realização deste trabalho o foco principal não é o desenvolvimento de software e sim a identificação de aspectos no desenvolvimento de software.

Os pesquisadores Kassab *et al.* (2005), Araújo *et al.* (2002) e Rashid *et al.* (2002), propõem a separação dos interesses funcionais dos não-funcionais utilizando as técnicas convencionais (UML) com algumas adaptações, e, após a separação os interesses não-funcionais são implementados com a tecnologia de aspectos. Camargo (2006) também propõe a separação de interesses funcionais e não-funcionais utilizando técnicas da UML com a diferença que Camargo (2006) apresenta um conjunto de critérios para a identificação de interesses transversais sendo eles: a quantidade de relacionamento, que são os casos de uso que possuem relacionamento com um ou mais casos de uso; casos de uso do tipo não-funcionais que são identificados pelo estereótipo <<NF>>; casos de uso de extensão, que se

relacionam por extensão (estereótipo <<extends>>) com um ou mais casos de uso; e o critério de essencialidade onde o objetivo é identificar se o comportamento do caso de uso transversal é essencial ao comportamento genérico do caso de uso base.

Jacobson (2003), diferente dos outros autores, apenas apresenta uma esboço, que é modificar a filosofia de desenvolvimento de software. A idéia é que exista um modelo de análise e projeto e um modelo de implementação específicos para cada caso de uso, não havendo, a princípio, a preocupação em reusar partes de modelos de outros casos de uso. Apenas as classes, atributos e métodos essenciais à implementação do caso de uso são criados, mesmo que seja esperado que outros casos de uso possam precisar de classes conceitualmente similares com métodos e atributos adicionais. Assim, casos de uso poderão ser desenvolvidos independentemente durante todo o processo de desenvolvimento e depois combinados para compor o sistema.

Em todas as abordagens o que há de comum é a utilização da UML para auxiliar a identificação e desenvolvimento de software orientado a aspectos. Outro ponto em comum é que em todas é necessário o conhecimento e habilidade da equipe de desenvolvimento para a identificação de aspectos. O ProFT/PU (Camargo, 2006), foi o que se mostrou mais completo descrevendo cada fase com detalhes e auxiliando o desenvolvimento desde a elaboração do Documento de Requisitos até a fase de teste do sistema.

7 CONSIDERAÇÕES FINAIS

Apesar de ter vários trabalhos relacionados ao desenvolvimento de software orientado a aspectos esse paradigma ainda está em processo de amadurecimento e muito ainda precisa ser feito para que seus fundamentos, práticas e mecanismos se estabilizem e que esse paradigma venha a ser largamente adotado. Atualmente, ele tem tido foco, principalmente, na pesquisa acadêmica e os avanços mais significativos nessa área concentraram-se em linguagens orientadas a aspectos. Com a crescente aceitação dessas linguagens, surgiu o interesse da comunidade em aplicar a orientação a aspectos em vários estágios do ciclo de vida do software. No entanto, faltam métodos, processos e técnicas que propiciem a sua utilização nas atividades do ciclo de desenvolvimento.

O ProFT/PU é um processo iterativo e incremental, fortemente baseado na identificação e acompanhamento de interesses transversais ao longo do processo de desenvolvimento de software. Ele fornece diretrizes, sugestões e define alguns passos a serem seguidos no desenvolvimento de um software orientado a aspectos. O principal objetivo do ProFT é cobrir todo o ciclo de vida mostrando como o processo todo pode ser conduzido, acompanhando os interesses transversais desde a análise até a implementação.

Uma limitação desse processo é a falta de ferramentas que apóiam a construção dos artefatos que envolvem o processo. Um outro problema encontrado é que mesmo utilizando o ProFT a identificação dos possíveis casos de uso candidatos a aspecto não é trivial e ainda depende muito dos conhecimentos e habilidades da equipe de desenvolvimento. Há muitas pesquisas nesse sentido, mais ainda a identificação dos aspectos nas primeiras fases de desenvolvimento de um software é um problema a ser resolvido.

Apesar das dificuldades encontradas na programação orientada a aspectos essa tecnologia se mostra muito promissora. A AOP traz maior produtividade, qualidade e facilidade para a implementação de novas funcionalidades, e já podemos tirar proveito prático deste novo paradigma de programação.

REFERÊNCIAS

AMBLER, S. **The Object Primer: the Application Developer's Guide to Object Orientation and the UML**, Cambridge University Press, 2001.

ARAÚJO, J; MOREIRA, A. **An Aspectual Use-Case Driven Approach**, Departamento de Informática, Faculdade de Ciências e Tecnologia Universidade Nova de Lisboa, Quinta da Torre, PORTUGAL, 2000.

ARAÚJO, J; MOREIRA, A.; BRITO, I; RASHID, A. **Aspect-Oriented Requirements with UML**, Workshop: Aspect-oriented Modeling with UML, UML 2002, Dresden, Germany, 2002.

CAMARGO, V. V. **Frameworks transversais: definição, classificações, arquitetura e utilização em um processo de desenvolvimento de software.** 2006. 255 f.Grau: Dissertação.USP – São Carlos, 2006.

CAMARGO, V.V; MASIERO, **P.C. Frameworks Orientados a Aspectos**. In: Anais do 19º Simpósio Brasileiro de Engenharia de Software. 2005, Uberlândia-MG. p. 200-215, out.2005.

CAYENNE. http://www.objectstyle.org/cayenne/ (último acesso em 6 de outubro de 2007).

CHUNG, L; NIXON, B; YU, E; MYLOPOULOS, J. **Non-functional requirements in software engineering** , Boston, Kluwer Academic, p. 439, 1999.

CLARKE, S; BANIASSAD, E. **Aspect-Oriented Analysis and Design: The Theme Approach**. Addison-Wesley, 2005.

CYSNEIROS, L.M; LEITE, J.C.S.P. **Definindo Requisitos Não Funcionais.** Simpósio Brasileiro de Engenharia de Software. 1997, Fortaleza – CE. p.49-54, Outubro 1997.

DIJKSTRA, E. **A Discipline of Programming**. Prentice-Hall, 1976.

ELRAD, T; FILMAN, R; BADER, A. **Aspect-Oriented Programming: Introduction**, Communications of the ACM, v 44 n°10, p.29-32, Oct. 2001

GRADECKI, J.D; LESIECKI, N. Mastering AspectJ – **Aspect Oriented Programming in Java**. Wiley Publishing, 2003.

HIBERNATE. www.hibernate.org (último acesso em 6 de outubro de 2007).

JACOBSON, I. **Use Cases - Yesterday, Today, and Tomorrow.** The Rational Edge, March 2003.

JACOBSON, I. **Use Cases and Aspects - Working Seamlessly Together,** In Journal of Object Technology, v 2, n° 4, July- August 2003, p. 7- 28.

JACOBSON, I.; BOOCH, G.; RUMBAUGH, J. **The Unified Software Development Process**, Addison-Wesley, 1999.

KASSAB, M; ORMANDJIEVA, O. **Towards an Aspect-Oriented Software Development Model with Tractability Mechanism**, Department of Computer Science and Software Engineering, Concordia University, 2005.

KICZALES, G.; HILSDALE, E; HUGUNIN, J; KERSTEN, M; PALM, J;GRISWOLD, W. **An Overview of AspectJ**. In J. L. Knudsen, editor, 15th European Conference on Object-Oriented Programming, v. 2072 , p. 327–353, Berlin, Heidelberg, and New York, Springer Verlag, 2001.

KICZALES, G.; LAMPING, J; MENDHEKAR, A; MAEDA, C; LOPES, C; LOINGTIER, J; IRWIN, J. **Aspect-Oriented Programming**. In M. Aksit and S. Mat-suoka, editors, European Conference on Object-Oriented Programming, v1241, p.220-242. Springer Verlag, 1997.

KICZALES, G.; HILSDALE, E; HUGUNIN, J; KERSTEN, M; PALM, J;GRISWOLD, **Getting Started With AspectJ, Communications of the ACM**, v. 44, n° 10, p.59-65, 2001.

KICZALES, G.; LAMPING, J; MENDHEKAR, A; MAEDA, C; LOPES, C; LOINGTIER, J; IRVING, J. **Aspect Oriented Programming**. In: Proceedings of 11 ECOOP. p. 220-242, 1997.

KICZALES, G; MEZINI, M. **Aspect-Oriented Programming and Modular Reasoning**. In: Proceedings of International Conference on Software Engineering (ICSE'05), St. Louis, Missouri, USA, p. 49-58, 2005.

OJB. http://db.apache.org/ojb/. (último acesso em 6 de outubro de 2007).

RASHID, A; SAWYER, P; MOREIRA, A; ARAÚJO, J. **Early Aspects. A Model for Aspect-Oriented Requirements Engineering**. IEEE Computer Society Press, p. 199-202, Essen, Germany, September, 2002.

SAARY, TL. **The Analytic Hierarchy Process**. NY, McGraw Hill, 1980.

STEIN, Dominik. **An Aspect-Oriented Design Model Based on AspectJ and UML**. Dissertação de Mestrado (Mestrado em Gerenciamento de Sistemas de Informação) – Universidade de Essen, Germany, 2002.

TARR, P; OSSHER, H; SUTTON, S. Hyper/JTM: **Multi-dimensional Separation of Concerns for Java**. In: Proc. of the 24th International Conference on Software Engineering. Orlando, Florida, 2002.

YODER, J.W; JOHNSON, R.E; WILSON, Q. D. **Connection Busines Objects to Relational Databases**. In: Proceedings on the Conference on the Pattern Languages of Programs, Monticello-IL, EUA. 1998.

APÊNDICE A – Documento de Requisitos do Sistema de Automação de Postos

Sistema para Automação de uma rede de Postos de Combustível

Documento de Requisitos

A - VISÃO GERAL DO SISTEMA

O sistema para Automação de Postos de Combustível, consiste no gerenciamento das bombas de combustível, controlando o estoque de combustível e o fluxo de caixa, tornando o posto totalmente automatizado. O sistema deve ainda emitir diversos tipos de relatórios e consultas, possibilitando um melhor gerenciamento do estabelecimento.

B – REQUISITOS FUNCIONAIS

B1 – Lançamentos diversos

1. O sistema faz a coleta dos dados gerado pela bomba de combustível, permitindo assim a atualização das informações, a bomba gera as seguintes informações: tipo de combustível, quantidade de combustível em litros, valor em reais e uma numeração incremental (encerrante).

2. O sistema deve permitir o gerenciamento do estoque de combustível, aumentando ou diminuindo a quantidade em estoque (baixa automática do estoque).

3. O sistema deve permitir o controle do caixa, como débito e crédito.

4. O sistema deve permitir a atualização do preço dos combustíveis.

5. O sistema deve permitir o gerenciamento dos funcionários do estabelecimento.

6. O sistema deve permitir o cadastro de clientes.

7. O sistema deve permitir o cliente efetuar o pagamento.

B2 – Impressão de diversos tipos de relatórios e consultas

8. O sistema deve permitir a impressão da nota fiscal mediante solicitação do cliente.

9. O sistema deve permitir a impressão de um relatório de estoque de combustível contendo a data e a hora atual, o tipo de combustível e a quantidade em litros.

10. O sistema deve permitir a impressão do caixa, por período (data), contendo todo o histórico de entrada e saída de valores, respectivo a data informada.

11. O sistema deve permitir a impressão de um relatório de funcionários.

12. O sistema deve permitir a impressão de um relatório de Clientes.

13. O sistema deve permitir a consulta on-line do caixa e do estoque de combustível, mediante a autenticação do usuário.

C – REQUISITOS NÃO FUNCIONAIS

C1. Confiabilidade

14. O sistema deve ter capacidade para recuperar os dados perdidos da última operação que realizou em caso de falha (gerencia de transação).

C2. Segurança

15. O sistema deve possuir senhas de acesso e identificação para diferentes tipos de usuários: administrador do sistema e funcionários do estabelecimento. Uma restrição do sistema de segurança é que o erro consecutivo de três tentativas de acesso bloqueia a conta do usuário.

Os papéis que os funcionários do posto exercem no sistema são: Presidente/Gerente, e Secretária.

Como o sistema deve operar na Web, alguns dados enviados para o servidor devem ser criptografados, como por exemplo, dados dos clientes e dados de faturamento.

16. O sistema de segurança também deve manter um registro dos usuários que acessaram o sistema com os seguintes campos: usuário, data de acesso, hora entrada e hora saída.

C3. Eficiência

17. O sistema deve responder a consultas on-line em menos de 5 segundos.

18. O sistema deve iniciar a impressão de relatórios solicitados dentro de no máximo 20 segundos após sua requisição.

C4. Portabilidade

19. O sistema deve ser executado em computadores Pentium 200mHz ou superior, com sistema operacional Windows 98 ou acima.

20. O sistema deve ser capaz de armazenar os dados em base de dados MySQL.