PUC

ISSN 0103-9741

Monografias em Ciência da Computação
n° 18/09

Suporte a uma abordagem para uma LPS de SMA com configuração dinâmica

Marcos Tadeu Silva

Carlos José Pereira de Lucena

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO

RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22451-900

RIO DE JANEIRO - BRASIL

Monografias em Ciência da Computação, No. 18/09 ISSN: 0103-9741
Editor: Prof. Carlos José Pereira de Lucena July, 2009

Suporte a uma abordagem para uma LPS de SMA com configuração dinâmica

Marcos Tadeu Silva e Carlos José Pereira de Lucena

mtsilva@inf.puc-rio.br, lucena@inf.puc-rio.br

Resumo. Linha de Produtos para Sistemas Multi-Agentes (LP-SMA) vêm sendo comumente utilizadas para envolver soluções altamente configuráveis. Nesse sentido, um dos objetivos a serem explorados com base nessa configuração é a utilização de sistemas auto-configuráveis. O trabalho aqui apresentado se define por disponibilizar uma abordagem que permite a configuração da LP-SMA de forma dinâmica, no qual um produto pode ser modificado em tempo de execução. Como estudo de caso foi re-utilizada a configuração de ANTs, um projeto específico da NASA e que faz parte de um dos trabalhos aqui relacionados. Contudo foram exploradas as diversas configurações existentes de modo a apresentar auto-adaptação em uma LP-SMA dinâmica.

Palavras-chave: Linha de Produtos para Sistemas Multi-Agentes, Auto-Adaptação, Programação Orientada a Aspectos, Programação de Variabilidades

Abstract. Multi-Agent Systems Products Lines (MAS-PL) have been used in order to produce high configurable solutions. In this sense, one of the goals is to characterize an approach for high configurable systems. The presented work aims at provide an approach for dynamic configuration of MAS-PLs, allowing runtime product derivation. As a Case-Study, it has been reused NASA Projects for ANTs configuration. However, several possible configurations have been explored in order to present self-adaptation in parallel with MAS-PL dynamic capability.

Keywords: Aspect-Oriented Programming, Variabilities Programming, Software Product Line Implementation Techniques

In charge of publications:
Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC-Rio Departamento de Informática
Rua Marquês de São Vicente, 225 - Gávea
22451-900 Rio de Janeiro RJ Brasil
Tel. +55 21 3527-1516 Fax: +55 21 3527-1530
E-mail: bib-di@inf.puc-rio.br
Web site: http://bib-di.inf.puc-rio.br/techreports/

Sumário

1 Introdução **1**

2 Fundamentação Teórica **1**

2.1 Linha de Produtos de Software 2

2.2 Linha de Produtos para Sistemas Multi-Agentes 2

2.3 Desenvolvimento de Software Orientado a Aspectos 2

2.4 AspectJ 3

3 Estudo de Caso: NASA ANTs **3**

4 Análise e Desenvolvimento do Estudo de Caso **4**

4.1 Modelagem 6

4.1.1 Diagrama de Casos de Uso 6

4.1.2 Diagrama de Features 7

4.1.3 Diagrama de Classes 8

4.1.4 Agentes 10

4.1.5 Aspectos 14

4.1.6 Componente de Adaptação 14

4.2 Implementação 15

5 Conclusão **18**

Referencias **18**

Lista de Figuras

1 Visão geral do processo de desenvolvimento definido para LP-SMA 5
2 *Features model* do estudo de caso 5
3 Diagrama de Casos de Uso 6
4 Diagrama de *Features*: Visão Lógica Geral 7
5 Diagrama de Domínio 9
6 Diagrama de Implementação: Visão de Proteção 1 10
7 Diagrama de Implementação: Visão de Proteção 2 11
8 Diagrama de Agentes: Visão *PowerAgent* 13
9 Diagrama de Aspectos 14
10 Diagrama de Classes: *Engine* de Adaptação 15

1 Introdução

Nos últimos anos sistemas de software vêm se tornando cada vez mais complexos e mediante as novas necessidades dos clientes, precisam se estabelecer com os novos requisitos exigidos. Nesse sentido, muitos sistemas devem ser evolutivos e, além disso, tais sistemas devem seguir os padrões de qualidade e princípios fundamentais da Engenharia de Software atual, assim como reutilização e extensibilidade.

Numa tentativa de dinimuir os custos e aumentar a rentabilidade, assim como a qualidade do produto, diferentes técnicas e abordagens vêm sendo tratadas e utilizadas. Geralmente, cada uma dessas soluções funciona para algum domínio de problema específico, não existindo então uma bala de prata que soluciona tudo e agrada a todos.

Como é de senso comum, a evolução de um sistema é dada segundo suas múltiplas versões, indicando que cada versão pode vir a ser um produto diferente. Dentro desse cenário, a abordagem de desenvolvimento de software com base em Linhas de produtos de Software [1] vêm sendo largamente utilizada. Além disso, a construção de famílias de produtos feitos sob o Paradigma Orientado a Agentes vem se mostrando cada vez mais necessária. Tal linha de produtos se caracteriza como Linha de Produtos para Agentes de Software (LP-SMA) [2].

Conforme a utilização de LP-SMAs como solução viável, houve uma pequena mudança no foco de desenvolvimento, a qual se estabelece por gerenciar da melhor forma possível as variabilidades agora identificadas. Desse modo, existe uma clara separação dos interesses existentes, pois tudo aquilo que não varia é classificado como parte do núcleo do sistema.

Entretanto, algumas vezes é necessária uma nova versão de um produto durante o tempo de execução do mesmo, o que caracteriza a obrigação de uma arquitetura capaz de evoluir também em tempo de execução. Tal arquitetura deve então possuir características muito comuns à computação autonômica, tais como Auto-Configuração e Auto-Adaptação.

O trabalho aqui apresentado se caracteriza pelo desenvolvimento de um mecanismo de configuração de variabilidades, tornando possível a implementação do modelo sugerido em [3]. Tal mecanismo torna possível a complementação do estudo de caso da NASA, apresentado também em [3]. Tal modelo foi feito utilizando-se uma LP-SMA e apresenta como principal característica a evolução dos produtos em tempo de execução. Para a implementação foi escolhida a programação orientada a aspectos (POA) [4]. Está técnica foi escolhida dado que muitos autores [5, 6, 7] convergem no que diz respeito a implementação de LPS com AOP.

Esse trabalho está estruturado da seguinte forma: a Seção 2 apresenta uma fundamentação teórica básica sobre Linha de Produtos tanto de Software comum quanto para Sistemas Multi-Agentes e Desenvolvimento de Software Orientado as Aspectos; na Seção 3 é apresentado estudo de caso completo definido pelo no trabalho do Peña [3].

2 Fundamentação Teórica

Nessa Seção são abordadas as principais tecnologias utilizadas nesse trabalho. Na Seção 2.1 é apresentado um breve referencial teórico sobre Linhas de Produtos de Software. Já na Seção 2.2 pode ser vista uma fundamentação básica sobre Linha de Produtos para Sistemas Multi-Agentes. Por fim, na Seção 2.3 é apresentada uma base sobre Desenvolvimento de Software Orientado as Aspectos e AspectJ.

2.1 Linha de Produtos de Software

Linha de Produtos de Software (LPS) [8, 1] é uma tecnologia que promove o suporte ao desenvolvimento baseado na derivação de uma aplicação em uma grande cadeia de produtos possíveis. Isso se dá por meio da criação de uma família de produtos de um mesmo domínio específico a partir de uma infra-estrutura comum, definida como núcleo ou cerne da aplicação (core asset) [9].

Dessa forma, é possível a construção de uma arquitetura de software altamente modular, capaz de endereçar diversas características do mesmo domínio escolhido, através da implementação dos pontos de variabilidade.

Além disso, as características (do inglês *features*) que definem uma LPS são usualmente representadas através de um diagrama chamado Diagrama de *Features* [10, 11]. Tal diagrama expõe a modelagem árvores de características, as quais representam as famílias de produtos.

2.2 Linha de Produtos para Sistemas Multi-Agentes

Em LPS existe todo um processo a ser utilizado para desenvolver uma família de produtos, da qual os produtos são deriváveis sistematicamente, ou mesmo de forma automática. Sendo assim, uma LP-SMA é uma linha de produtos que se constituem em Sistemas Multi-Agentes (SMA). Trabalhos muito recentes, como [2], vêm mostrando que a construção de LP-SMAs são cada vez mais práticas, pois existe um aumento na qualidade dos sistemas, inclusive em tempos de modelagem, implementação e teste. Desta forma é possível se obter a construção de produtos mais economicamente viáveis [3].

De acordo com [12] existem diversas metáforas e visões que se incluem no tocante a LP-SMA, tais visões são: *Acquaintance point of view* e*Structural point of view*. A primeira apresenta uma visão organização das iterações dos agentes como um conjunto de relacionamentos entre seus papéis. Já a segunda visão mostra agentes como artefatos que pertencem a alguma organização, grupo ou mesmo times, tendo em vista uma visão hierárquica estrutura, mimicando um comportamento estrutura de sociedade.

2.3 Desenvolvimento de Software Orientado a Aspectos

Para a implementação da Linha de Produtos adotamos a programação orientada a aspectos. Isso ocorreu, pois no tocante a utilização POA para LPS, muitos autores convergem no que se diz respeito à implementação de LPS através de POA [5, 6, 13], os quais citam POA como uma técnica efetiva no suporte de variabilidades. Dessa forma, adotamos tal tecnologia. Desse fato, fomos conservadores no que acreditamos ser uma solução viável também para a implementação de uma LP-SMA.

Desenvolvimento de Software Orientado a Aspectos (DSOA) [4, 14] foi proposto como uma técnica para melhorar a separação de interesses (*concerns*) na construção de software com o objetivo de obter maior reusabilidade e facilidade de evolução. Nesse sentido DSOA apóia a modularização de interesses considerados transversais (*crosscutting concerns*), ou seja, interesses que se caracterizam por estarem espalhados dentro do negócio.

De acordo com [14], a Programação Orientada a Aspectos (POA) [4] suporta o encapsulamento das características (*features*) do software de modo a torná-los interesses transversais. Desse modo, são construídas unidades modulares através de mecanismos de composição, tais como *pointcut-advice* e *inter-type declarations*.

2.4 AspectJ

AspectJ [15] é uma linguagem específica para o DSOA. Tal linguagem se caracteriza por ser uma extensão da linguagem Java, bastante utilizada no desenvolvimento de software orientado a objetos. Nesse caso, a programação com AspectJ permite o envolvimento não só de aspectos mas também de classe Java para a separação dos interesses considerados transversais. Em geral, os conceitos de negócio são definidos utilizando orientação a objetos, enquanto que os interesses transversais são separados em unidades definidas como *aspects*.

Em AspectJ, tais *aspects* possuem:

- ***pointcut designators*:** utilizados definir um subconjunto de pontos de junção **Join Points** a serem interpretados durante a execução do programa;

- ***advices*:** porção de código a ser inserida nos pontos de junção, feitos através dos *pointcut designators*;

- ***intertype declarations*:** estruturas que permitem a introdução de atributos e métodos nas classes, além de possibilitar a mudança na estrutura hierarquica entre classes (fazendo uma classe estender outra classe ou interface) e mudar uma exceção do tipo *checked* para *unchecked*.

Além dessas características, na linguagem AspectJ existem três tipos diferentes de os *advices* [16], listados a seguir: (i) *adivice* do tipo ***after***, utilizado para adicionar código após o *join point*; (ii) *adivice* do tipo **before**, permite adicionar código executável antes do *join point* e (iii) *adivice* do tipo **around**, mais expressivo que os tipos anteriores, adiciona código ao redor do *join point*, permitindo a execução de código de forma condicional.

3 Estudo de Caso: NASA ANTs

Esta Seção trata um estudo de casa apresentado nos trabalhos de Peña, 2006 e 2007, [17, 3]. O estudo de caso se caracteriza pela representação de um sistema de exploração espacial capaz de executar coleta de informações, enviando-as à Terra.

O estudo de caso analisado nesse trabalho visa uma das missões da NASA, na qual é utilizado o conceito de *Autonomous Nano Technology Swarm* (ANT). Um ANT nada mais é do que um grupo de agentes que trabalham de forma autônoma mas com o objetivo em comum de completar a missão.

Como é apresentado [17], a utilização de sistemas baseados em agentes vem sendo introduzida em um número significativo de pesquisas da NASA, principalmente no tocante a exploração espacial. Além disso, a tecnologia de agentes vem sendo cada vez mais empregada e sua aplicação em futuras missões espaciais vem sendo estudado.

A *Lander Amorphous Rover Antenna* (LARA) é uma missão prevista para ser iniciada entre 2015-2020 e que se constitui na utilização de batalhões de veículos (*rovers*) re-configuráveis. Alguns desses deverão ser utilizados para investigar os solos de outros corpos celestes, assim como Marte e Lua.

Dessa forma, há a necessidade de que tais veículos, possuam comportamentos que permitam examinar (*swarm*) o solo. Além disso, eles devem ser capazes de se auto-reconfigurar, alterando suas características. A exemplo, algumas formas pré-determinadas,

tais como 'cobra', cilindro e antena, provêm diferentes características e uma porção de funcionalidades diferentes com base em cada uma dessas formas.

O objetivo da missão LARA é justamente estudar materiais para a criação de possíveis bases lunares. Outra missão de nome *Prospecting Asteroid Mission* (PAM), também ainda conceito, tem por característica principal ser baseada em ANTs, os quais terão por objetivo explorar cinturões de asteróides. Além dessas a *Saturn Autonomous Ring Array* (SARA) também é uma missão conceito, muito similar à missão PAM, entretanto, o objetivo difere no sentido de que as análises serão feitas nos Anéis de Saturno.

Como pode ser visto, as missões possuem conceitos muito parecidos, diferindo em alguns princípios: (i) na missão PAM os veículos espaciais devem ser dotados de proteção contra chuvas solares; (ii) na missão SARA isso não é uma preocupação, mas sim um controle gravitacional mais refinado, já que é necessário se esquivar do empuxo gravitacional gerado pelas partículas dos Anéis de Saturno; (iii) por fim a missão LARA requer força motora bastante alta já que entra em contato direto com o solo a ser investigado, enquanto que as outras missões envolvem veículos capazes de sobrevoar asteróides utilizando propulsão a gás e energia solar.

4 Análise e Desenvolvimento do Estudo de Caso

O estudo de caso apresentado na Seção anterior mostra as missões SARA, PAM e LARA, conceituadas pela NASA. Na descrição do estudo de caso também são apresentadas os conceitos comuns e divergentes entre as missões.

Nesse sentido, o autor da abordagem original [3] segue um processo de desenvolvimento próprio para LP-SMA definido em [1]. Tal processo se caracteriza na divisão do desenvolvimento em dois estágios principais: *domain engineering* e *application engineering*. A primeira é responsável por prover os recursos classificados como reutilizáveis e que serão explorados durante a *application engineering*, que possui o foco na construção de aplicações específicas. Dado o contexto da *domain engineering*, as atividades que fazem parte do processo e referentes a construção da arquitetura base são [1]:

- *Build acquaintance organization:* primeira atividade a ser realizada, consiste no desenvolvimento de um conjunto de modelos em diferentes níveis de abstração, servindo de base para a obtenção do *traceability model*.
- *Build features model:* segunda atividade do processo, é responsável por detectar aquilo as características comuns e as variabilidades, adicionando-as no *traceability model* para criar o *feature model*.
- *Analyze commonalities:* executada após a segunda atividade, é necessário determinar uma analise sobre tudo que é definido como parte comum do software. Isto é feito para que seja possível encontrar quais características são definidas como *core features*, verificando o que é mais comum na família de produtos.
- *Compose core features:* finalmente, dado os *feature models* é possível utilizar operações de composição para se obter as *core features* e o *core architecture*.

Na Figura 1 é apresentado o processo para a construção da arquitetura principal de uma LP-SMA, seguindo a nomenclatura especificada em [1]. A descrição das atividades

utilizadas durante a construção dessa arquitetura, assim como os artefatos resultantes da execução desse processo fogem ao foco desse trabalho, para um melhor entendimento do processo em si, pode-se apreciar o trabalho de [3] e [1], onde o mesmo está mais bem detalhado.

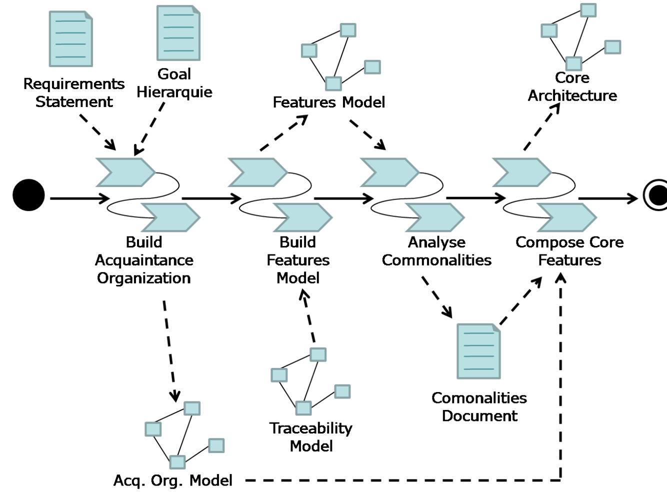

Figura 1: Visão geral do processo de desenvolvimento definido para LP-SMA

O *features model* resultante da execução do processo acima descrito pode ser visto na Figura 2. Nessa figura está o diagrama que mostra as variabilidades encontradas após pela atividade *Build features model*. Cada uma das "Camadas 1-4" representa um nível de abstração do *feature model*, nas quais quanto mais baixa a camada, mais detalhada estará a característica desejada. Ou seja, uma característica da "Camada 2" é muito mais abstrata do que as características da "Camada 3", e assim por diante.

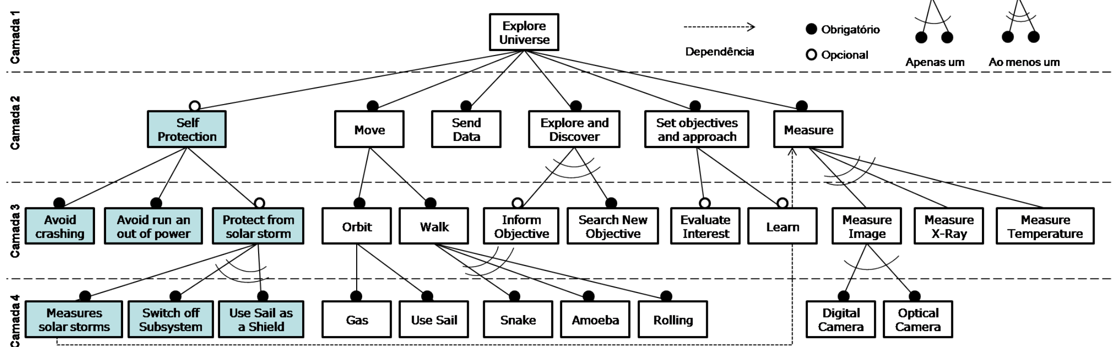

Figura 2: *Features model* do estudo de caso

De acordo com [17, 3], o diagrama contém informações adquiridas durante a etapa de rastreabilidade da aplicação. Como pode ser visto, o modelo representa uma família de produtos definida a partir da composição das características apresentadas no diagrama. Cada nó (*feature*) nó diagrama é caracterizada como obrigatória, opcional, alternativa ou até mesmo mutuamente exclusiva (*or-exclusive*). As dependências entre tais nós também estão dispostas no diagrama da Figura 2.

4.1 Modelagem

Nessa Seção é apresentada a modelagem da aplicação. Tendo em vista a utilização de orientação a objetos, a modelagem foi feita com utilizando a *Unified Modeling Language* (UML) [18]. O modelo se constitui da utilização dos construtores UML para definir representar os **Casos de Uso**, **Classes**, **Agentes**, **Aspectos** e **Características** do sistema. Para maior facilidade de compreensão o modelo foi separado em diagrams que ilustram as visões lógicas de acordo com os *building blocks* [19] previamente definidos. A seguir são apresentados os diagramas principais do modelo.

4.1.1 Diagrama de Casos de Uso

Primeiro passo da modelagem, os casos de uso foram levantados de acordo com as funcionalidades essenciais esperadas para a conclusão do projeto. A Figura 3 apresenta o diagrama de casos de uso. Tal diagrama se constitui de três casos de uso simples:

- ***Find an Asteroid:*** funcionalidade de busca e descoberta de um asteróide, dado que o sistema estará imerso no espaço sideral;
- ***Extract Data from Asteroid:*** funcionalidade de exploração e medição de um asteróide, quando encontrado, permitindo a extração de dados importantes a serem estudados;
- ***Send Data to Earth:*** funcionalidade que permite o envio dos dados extraídos sobre os asteróides encontrados para alguma base de dados que se encontra Terra.

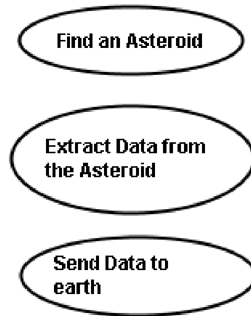

Figura 3: Diagrama de Casos de Uso

Como pode ser observado existe certa dependência entre os casos de uso levantados, embora ela não esteja explicitada no diagrama, pode-se concluir que um *Find an Asteroid* é pre-requisito de *Extract Data from Asteroid* que por sua vez, é requisito de *Send Data to Earth*.

Além disso, pode também ser observado que não existem atores para executar os casos de uso. Nesse sentido alguém pode sugerir um erro na modelagem desses casos de uso, entretanto, o que ocorre é que um ator em potencial são as próprias *threads* do sistema, uma vez que será um sistema baseado em agentes. Além desse fato, os casos de uso descritos estão num nível de abstração do negócio da aplicação.

4.1.2 Diagrama de Features

A modelagem do diagrama de *features* feito com base no modelo de *features* definido em [2] e [3]. Nesse sentido houve apenas um esforço para transformá-lo em parte integrante do modelo feito em UML, de forma *ad hoc*. Tal esforço se deu pela necessidade de mapeamento entre os *building blocks* do modelo do sistema.

Na Figura 4 é apresentado o diagrama do modelo de *features* em UML. Vale salientar que o diagrama é apenas uma visão lógica do modelo de *features*, não objetivando apresentá-lo por completo.

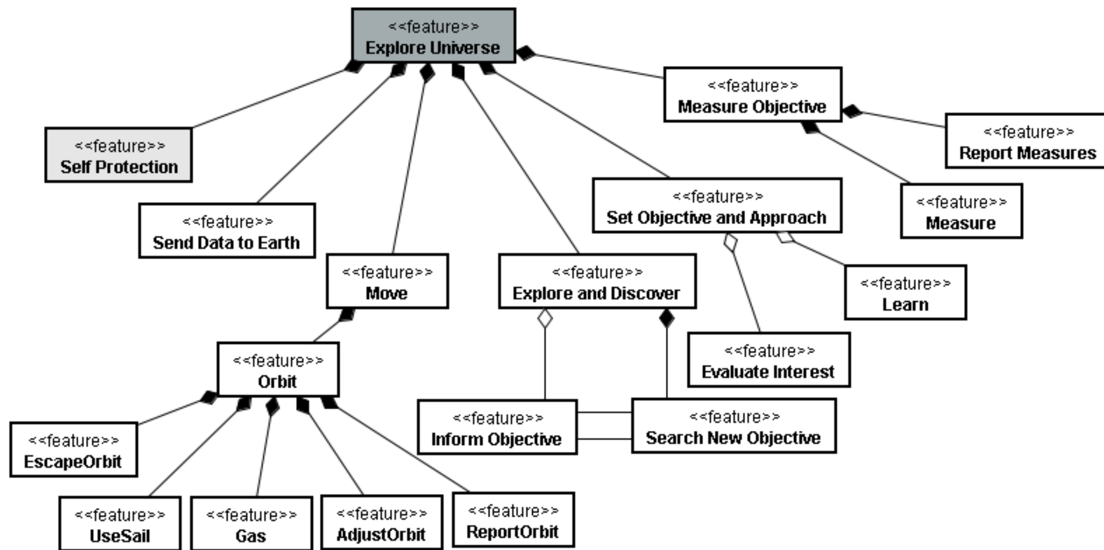

Figura 4: Diagrama de *Features*: Visão Lógica Geral

Por se tratar de parte do modelo UML, foram utilizados os construtores padrão da linguagem. Nesse caso, o modelo de *features* da figura 4 se diferencia do modelo definido em [2] (ver Figura 2) por questões de convenções de notação. Dessa forma, foi definida uma legenda de mapeamento para aumentar a compreensão do modelo:

- **estereótipo *«feature»*:** apenas classes UML de nível M2 podem ser classificadas com esse estereótipo. Toda classe marcada com *«feature»* se constitui em uma *feature*.
- **composição UML:** a semântica desse relacionamento quando dado entre duas classes marcadas com o estereóripo *«feature»* foi modificada. Todo relacionamento de composição entre duas *features* deve ser interpretado como uma obrigatoriedade. Ou seja, se uma *feature* filha é obrigatória a uma *feature* pai.
- **agregação UML:** similar a composição, a semântica desse relacionamento também foi modificada. Dado um relacionamento de agregação entre uma *feature* filha F com sua *feature* pai P deve ser interpretado como F é uma feature alternativa de P.
- **associação UML:** para representar o relacionamento de 'apenas um' foi utilizada uma associação entre cada umas das *features* participantes. Essa associação deve ser não navegável e de multiplicidade 1 para ambos *target* e *source*. Nesse caso, dados

um conjuto de features P, A, B e C, das quais A, B e C são filhas de P, definido '-' como a associação, podemos fazer:

  - deve existir apenas uma *feature* entre A e B: A - B.
  - deve existir apenas uma *feature* entre A, B e C: A - B - C, ou A - C - B ou B - A - C. Não importando então a ordem das associações.

- **dupla associação UML:** para representar o relacionamento de 'ao menos um' foi utilizada uma associação entre cada umas das *features* participantes. a dupla associação deve funcionar de forma análoga a associação simples, sendo assim as propriedades definidas no relacionamento 'apenas um' são válidas para essa associação.

4.1.3 Diagrama de Classes

Após a identificação das entidades básicas do problema, foi feito um mapeamento da solução de forma a representar o domínio do sistema utilizando o paradigma orientado a objetos. Durante a modelagem das classes pertencentes ao sistema foram feitos os níveis de **domínio** e **implementação**.

Durante a modelagem do domínio foram detectadas as classes:

- `ANT`: representa um *Autonomous Nano Technology Swarm* a ser utilizado para exploração espacial;

- `Sail`: representa uma vela a ser utilizada pelo `ANT`. Um `ANT` pode possuir várias instâncias de `Sail`, as quais são necessariamente utilizadas. Além disso, não faz sentido utilizar um `ANT` sem `Sails`, ou vice-versa. Portanto, esse tipo de relacionamento corresponde a uma composição;

- `Camera`: representa a camera utilizada pelo `ANT` para captar os dados durante a exploração. Um `ANT` deve possuir pelo menos uma câmera;

- `Universe`: representa o espaço sideral;

- `UniversThing`: representa de forma abstrata qualquer objeto pertencente ao espaço sideral;

- `Asteroid`: representa um asteróide como sendo um objeto concreto do espaço sideral. Um `ANT` necessariamente estará preocupado com os asteróides existentes no espaço;

- `SolarStorm`: representa uma chuva solar como sendo um objeto concreto do espaço sideral. Um `ANT` também estará preocupado com as chuvas solares, entretanto, a responsabilidade se dará para as velas, nesse caso, representada pela classe `Sails`;

O diagrama de classes que contém as entidades listadas e representa a modelagem do domínio pode ser visto na Figura 5.

O diagrama de implementação por sua vez possui as visões lógicas de acordo com o mapeamento de *features*. Dessa forma, o modelo de implementação foi dividido em sete visões lógicas. Cada visão lógica possui uma adição específica de elementos de implementação as classes (atributos, métodos, enumerações, etc):

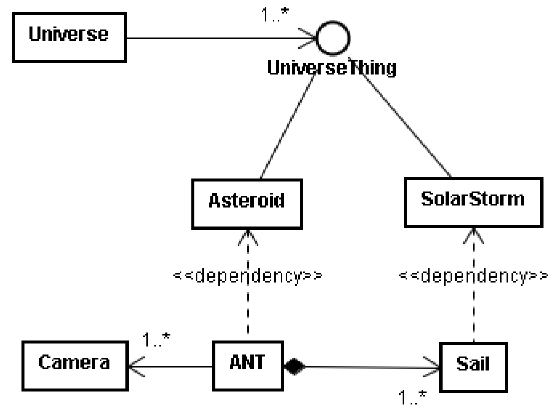

Figura 5: Diagrama de Domínio

- **Visão de Abordagem**: apresenta o mapeamento entre as classes que implementam o conjunto de *features* em *Set Objective and Approach*;
- **Visão de Descoberta**: apresenta o mapeamento entre as classes que implementam o conjunto de *features* em *Explore and Discover*;
- **Visão de Medição**: apresenta o mapeamento entre as classes que implementam o conjunto de *features* em *Measure*;
- **Visão de Movimentação**: apresenta o mapeamento entre as classes que implementam o conjunto de *features* em *Move*;
- **Visão de Proteção 1**: apresenta o mapeamento entre as classes que implementam o conjunto de *features* em *Self-Protection* contra queda de energia e colisão;
- **Visão de Proteção 2**: apresenta o mapeamento entre as classes que implementam o conjunto de *features* em *Self-Protection* contra chuvas solares;

Para melhor exemplificar, foram separadas duas visões lógicas bastante ilustrativas, contendo o mapeamento entre os elementos de implementação e as *features*. Nas Figuras 6 e 7 são apresentados as visões lógicas de Proteção 1 e 2, respectivamente. A Visão de Proteção 1 possui as classes e relacionamentos existentes para implementar de forma específica as *features Avoid Run out of power* e *Avoid Crashing*. Já a Visão de Proteção 2 possui as classes e relacionamentos que implementam as *features Use sail as shield* e *Switch-off subsystems*. Embora os agentes apareçam nessas visões lógicas, eles somente serão descritos em mais detalhes na seção 4.1.4. Nesse momento, o foco é demonstrar os mapeamentos existentes entre as features e seus respectivos elementos de implementação.

Como pode ser visto na Figura 6, foram adicionados diversos elementos de implementação, assim como agentes diferentes para cada uma das duas *features*. No lado esquerdo superior está a classe `ANT` e a enumeração de tipos `AllStatus`. Ao lado direito superior estão as *features* mapeadas e abaixo estão os agentes adicionados para a implementação dessas *features*.

Nesse cenário, `AllStatus` fornece as constantes necessárias para implementar a *feature Avoid Run out of power*. Pelo mesmo motivo foram adicionados também o atributo

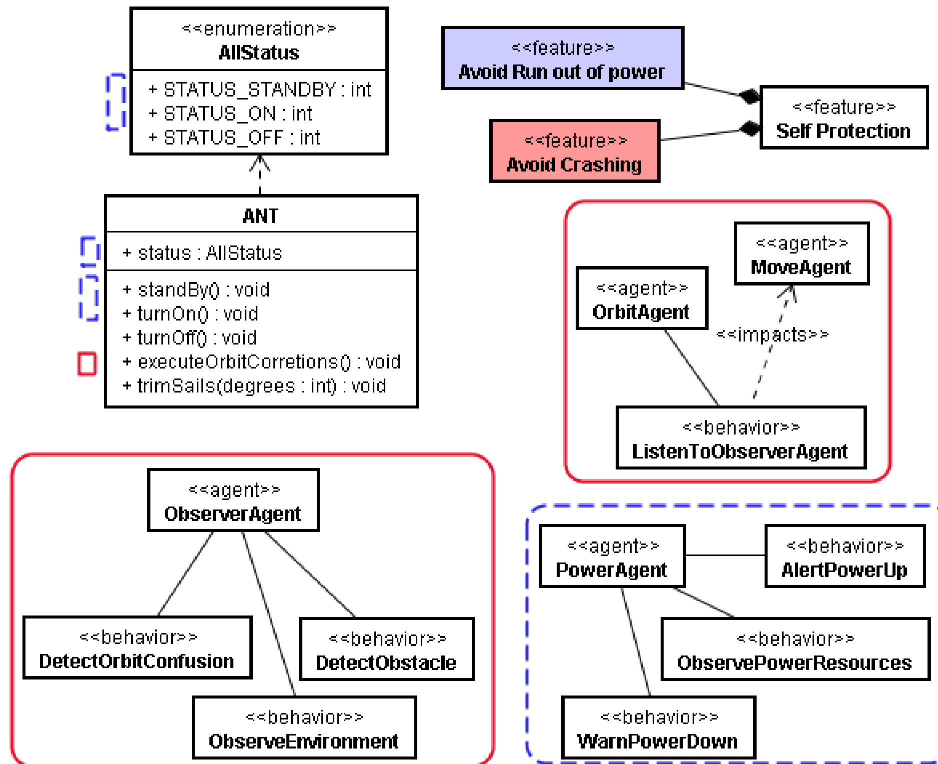

Figura 6: Diagrama de Implementação: Visão de Proteção 1

`status` e os métodos `standby()` e `turnoff()`. Todos esses elementos de implementação estão mapeados com retangulos de linha tracejada, inclusive os agentes, que estão envolvidos por um retangulo de mesmo tipo, conforme pode ser visto na Figura 6. Usando o mesmo conceito, os elementos de implementação utilizados para implementar a *feature Avoid Crashing* estão mapeados com retangulos de linha cheia.

Analogamente, na Figura 7 estão as classes (direita superior) e agentes (abaixo) utilizados para implementar as *features* correspondentes (esquerda superior).

Nesse caso, a *feature Use sail as shield* está associada com as classes `Sail` e `ANT`, assim como também os agentes que gerenciam os recursos de energia do sistema. Todos esses elementos podem ser identificados através dos retângulos de linha tracejada. Os elementos que estão identificados com retângulos de linha cheia são os correspondentes a feature *Switch off sub-systems*, assim como os métodos `Sail::trim()` e `ANT::trimSails()`.

4.1.4 Agentes

Na seção anterior foram apresentados alguns agentes e comportamentos referentes a implementação das *features*, de acordo com cada visão lógica. Nessa seção o foco é a descrição de cada um desses agentes, juntamente com seus relacionamentos e comportamentos.

Existem ao todo 6 agentes e 25 comportamentos, que juntos implementam as características ligadas a decisões durante a execução do sistema. A exemplo, pode-se tomar a ação de desligar o sistema utilizando o método (`ANT::turnOff()`), entretanto, é necessário que alguém tome essa decisão, baseado no fato de uma forte chuva solar estar a caminho.

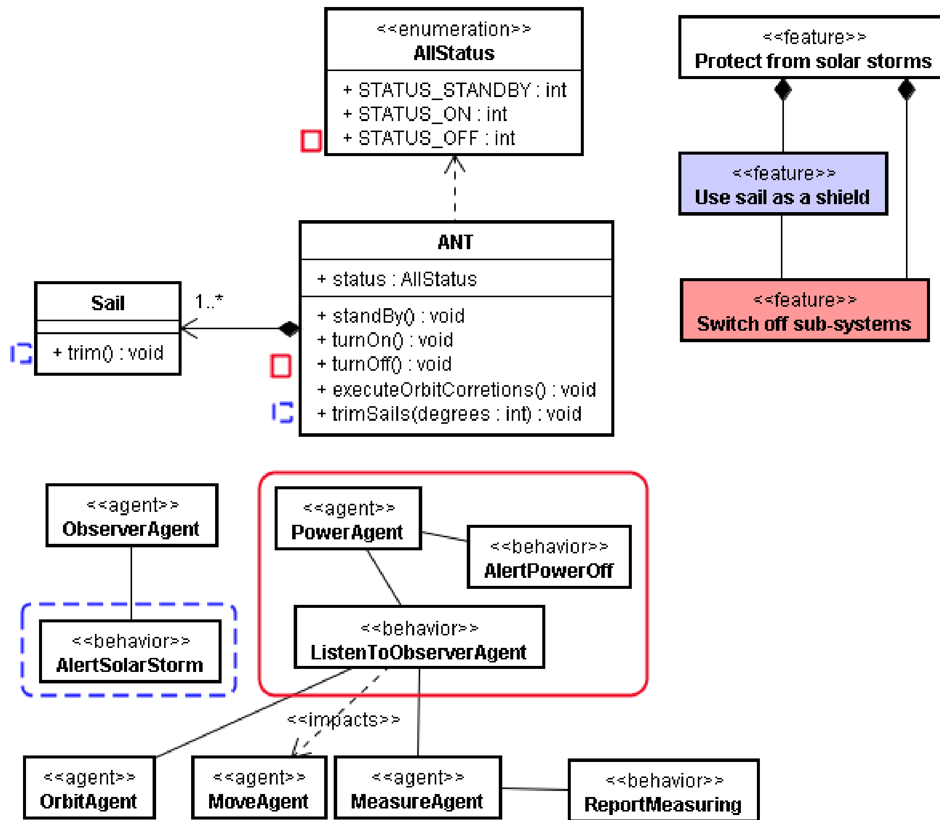

Figura 7: Diagrama de Implementação: Visão de Proteção 2

Dado que essa chuva foi detectada, o responsável por avaliar as condições e requisitar o método de desligamento será um agente.

Assim como no modelo de classes, a modelagem de agentes também está subdivida em visões lógicas para melhor uma compreensão. Entretanto, no contexto de agentes, cada visão lógica se limita a apresentar um único agente e seus comportamentos, contendo mesmo nome do seu respectivo agente. Sendo assim, ao mesmo tempo em que serão explicadas tais visões lógicas, os agentes e comportamentos internos também o serão:

- `Explorer Agent`: representa o agente e comportamentos responsáveis por explorar o espaço sideral em busca de asteróides. Os comportamentos existentes são:
  - `AnalyseObjective`: dado uma missão (geralmente encontrar asteróides) e um asteróide encontrado, esse comportamento requisita que ele seja analisado.
  - `InformDiscovery`: dado um asteróide no espaço sideral, esse comportamento informa ao detectá-lo.
  - `LookForObjectives`: dado um intervalo de tempo onde existe a necessidade de busca por mais asteróides, esse comportamento permite a exploração desses asteróides.
  - `WarnObjective`: dado um asteróide detectado e analisado, esse comportamento avisa que o objetivo já foi analisado.

- `Learn Agent`: representa o agente e comportamentos responsáveis por recuperar os dados coletados. Os comportamentos existentes são:
  - `ChooseObjective`: dado um conjunto de possíveis asteróides, esse comportamento escolhe um a ser investigado.
  - `EvaluateInterest`: avalia o interesse num dado asteróide. As opções de escolha por asteróides são feitas de acordo com as avaliações feitas por esse comportamento.
  - `LearnAboutData`: recupera os dados que foram coletados.
  - `ListenToDataAgent`: dado que existe uma coleta de dados, esse comportamento permite conversar com o agente que cuida do tratamento dos dados.
  - `ListenToExploreAgent`: dado que existe uma análise, sendo necessário saber quais asteróides devem ser analisados, esse comportamento permite a comunicação com o agente de observação.
  - `ReceiveObjective`: dado que um asteróide foi selecionado, esse comportamento recebe a missão de analisado.
  - `SelectApproach`: dado que deve ser feita uma análise em um asteróide, esse comportamento deve decidir qual será a abordagem utilizada para coletar os dados do asteróide.
- `Measure Agent`: representa um agente e seus comportamentos, sendo este responsável por criar os dados através de medições dado um asteróide. Os comportamentos existentes são:
  - `ListenToObserverAgent`: esse comportamento permite a comunicação com o agente de observação.
  - `ReportMeasuring`: esse comportamento permite reportar as medições feitas.
  - `RequestMeasuring`: esse comportamento requisita as medições, dado que foi encontrado um asteróide.
- `Move Agent`: representa um agente e seus comportamentos, sendo este responsável por assumir os controles de movimentação do `ANT`. Os comportamentos existentes são:
  - `DecideDirection`: comportamento que calcula uma boa direção a ser seguida.
  - `ListenToOrbitAgent`: dado que as direções são variáveis de acordo com a orbita assumida, quando há uma, esse comportamento permite a comunicação com o agente que avalia a orbita.
  - `ListenToExploreAgent`: esse comportamento prepara a comunicação com o agente de observação, permitindo saber quando o `ANT` deve ser movimentado.
- `Observer Agent`: representa um agente e seus comportamentos, sendo este, responsável por observar o ambiente em que o `ANT` se encontra. Os comportamentos existentes são:
  - `AlertSolarStorm`: esse comportamento detecta as chuvas solares.

- `DetectObstacle`: esse comportamento detecta se existe algum obstáculo no caminho entre o `ANT` até o seu objetivo.
- `DetectOrbitConfusion`: esse comportamento detecta eventuais problemas que podem ocorrer durante o percurso enquanto um `ANT` estiver em órbita, isto é, no sentido de desvios de órbita do asteróide orbitado.
- `ObserveEnvironment`: esse comportamento fica continuamente observando o ambiente.

- `PowerAgent`: representa um agente juntamente com seus respectivos comportamentos, o qual é responsável por controlar os recursos energéticos do sistema.
  - `AlertPowerOff`: requisita o desligamento sumario do sistema.
  - `AlertPowerUp`: requisita o re-ligamento do sistema.
  - `ListenToObserverAgent`: esse comportamento permite a conversação entre o `PowerAgent` e `ObserverAgent`.
  - `ObservePowerResources`: comportamento para a verificação contínua dos recursos de energia do sistema.
  - `WarnPowerDown`: esse comportamento certifica-se de que vai haver um desligamento do sistema, verificando se pode ser utilizado o "modo de espera".

Para melhor ilustrar as visões lógicas para agentes previamente definidas, a Figura 8 apresenta o diagrama de agentes para a visão lógica *PowerAgent*. Dado que foi utilizada a UML, foram anexados os estereótipos `«agent»` e `«behavior»` que representam um agente e um comportamento, respectivamente. Os comportamentos existentes em dado agente são feitos através de relacionamento de associação não navegável com multiplicidade 1 para ambos os relacionados. Além disso, a associação direcionada com estereótipo `«impacts»` aponta que durante a execução de um comportamento, outro agente ou comportamento poderá sofrer algum impacto.

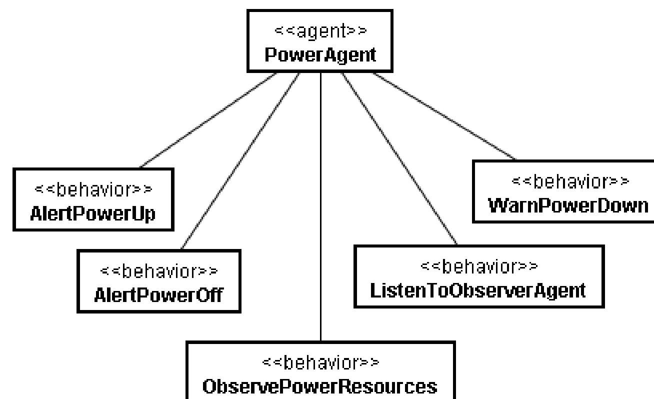

Figura 8: Diagrama de Agentes: Visão *PowerAgent*

4.1.5 Aspectos

Uma vez já apresentados os diagramas de casos de uso, classes, agentes e *features*, resta apenas apresentar o diagrama de mapeamento das variabilidades em aspectos. Esse diagrama tem por base a utilização de POA para implementar as variabilidades do sistema. A Figura 9 apresenta o diagrama de classes que representam as ligações entre os aspectos e *features*. Nesse mesmo diagrama, os aspectos estão representados com o estereótipo `«aspect»`, enquanto que as ligações são associações do tipo realização.

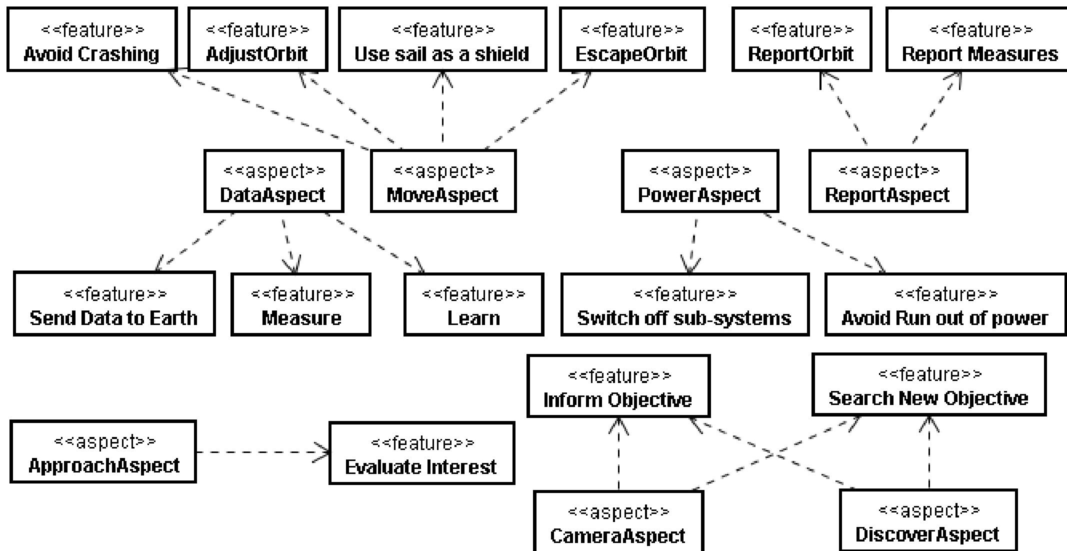

Figura 9: Diagrama de Aspectos

Embora aspectos tenham sido utilizados para implementar as variabilidades, geralmente explicitadas por *features*, existem várias ocorrências do tipo: um aspecto realiza uma *feature*; aspectos realizam uma *feature* e um aspecto realiza várias *features*.

4.1.6 Componente de Adaptação

Embora a modelagem do sistema tenha sido contemplada, há ainda a necessidade de apresentar uma solução para a capacidade de auto-adaptação. Tal característica é inerente a propriedade de auto-configuração, definido pelo problema de inserir `features` em tempo de execução. Para solucionar esse problema, um componente capaz de efetuar a adição e remoção de *features* em temo de execução foi criado.

A Figura 10 apresenta um diagrama que contém as classes que implementam o componente de adição/remoção de *features* em tempo de execução, sendo elas:

- classe `Feature`: representa uma *feature*, contendo entre outros, um conjunto de caracteres (*String*) `name` e um elemento booleano (*boolean*) denominado `added`. O atributo `name` representa o nome da *feature*, conseqüentemente um identificador. O atributo `added` permite saber se a *feature* está adicionada ao produto (`added=true`), ou não (`added=false`).

- classe `FeatureManager`: responsável por adicionar ou remover instâncias de `Feature` através dos métodos `add()` e `remove()` respectivamente. Essa classe permite o gerenciamento de *features* em tempo de execução.

- interface `FeatureRealizer`: representa de forma abstrata os elementos passíveis de implementação de uma *feature*. Como pode ser visto na figura, uma `Feature` possui um relacionamento de agregação com essa interface, permitindo definir quem são os responsáveis por sua implementação. O método `wrapFeature()` serve para encapsular uma instância de `Feature` a ser realizada. Enquanto que o método `realizeFeature()` permite a adição de mais realizadores para uma dada *feature* previamente encapsulada. Os métodos `activate()` e `desactivate()` possibilitam a ativação e desativação dos elementos de implementação.

- Elementos de mapeamento: existem duas classes concretas que realizam a interface `FeatureRealizer`. Uma delas é a classe `MappedAspect`, que define os aspectos mapeados e `MappedClassComponent`, que define os elementos de uma classe que implementam a *feature* dada.

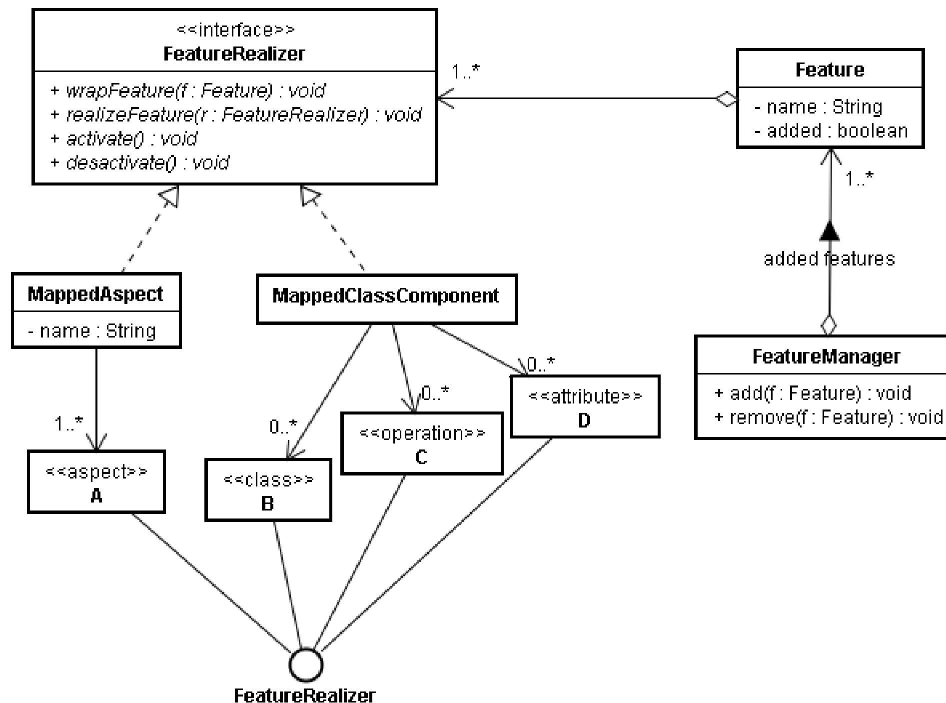

Figura 10: Diagrama de Classes: *Engine* de Adaptação

4.2 Implementação

A implementação foi feita sobre a modelagem apresentada na seção anterior. Diante desse fato, foram implementadas classes, agentes e aspectos, de forma a realizar as *features*

do modelo de *features*. Devido ao fato da implementação ser bastante extensa, serão apresentados apenas o que foi considerado mais relevante para o entendimento da solução.

Para a implementação do sistema foram utilizadas as tecnologias:

- linguagem de programação orientada a objetos Java SE [1]. Tanto o núcleo do sistema quanto os agentes foram implementados em linguagem Java;
- framework para desenvolvimento de sistemas multi-agentes JADE [2]. Os agentes e comportamentos foram implementados com apoio desse framework;
- framework com suporte a desenvolvimento de aplicações orientadas a aspectos *Spring*[3] juntamente com o módulo *Spring AOP*[4]. O mapeamento dos aspectos foi feito através do *Spring* justamente com o suporte de POA existente.

Dado que a codificação é bastante grande, foram selecionados apenas uma implementação para cada elemento de implementação. A seguir está um fragmento de código da implementação da classe ANT, julgada a mais importante do núcleo do sistema. Como pode ser visto esta classe possui apenas as assinaturas de seus métodos. Além disso, não há atributos e o construtor padrão também se encontra vazio.

```
/**
 * This class represents an Autonomous Nano Technology Swarm (ANT).
 * @author Marcos
 */
public class ANT {
    /** Constructs a new Ant. */
    public ANT() { }

    /** Approaches an objective. */
    public void approach() { }

    /** Controle the ANT orbit. */
    public void controlOrbit() { }

.   .   .

    /** Turn the system on. */
    public void turnOn() { }
}
```

Todos esses métodos que estão vazios serão preenchidos em acordo da composição com dos aspectos nos pontos entre-cortantes. A exemplo será mostrado o método `turnOn()`. Este método, como foi dito anteriormente, serve para definir o estado do ANT como ligado, geralmente utilizado em dois pontos: (i) ANT acaba de ser instanciado e (ii) ANT sofreu um desligamento e precisa ser re-estabelecido. A seguir está o aspecto responsável por realizar as *features Switch off sub-systems* e *Avoid run out of power*.

[1] www.java.com

[2] jade.tilab.com

[3] www.springsource.org

[4] static.springsource.org/spring/docs/2.5.x/reference/aop.html

```
/**
 * This class represents the Power Aspect.
 * @author Marcos
 */
@Aspect
public class PowerAspect {
    /** Composes the setUp method from PowerAgent. */
    @Around("execution(boolean PowerAgent.setUp())")
    public void powerAgentSetup() { ... }

    /** Composes the action method from PowerAgent. */
    @Around("execution(boolean PowerAgent.action())")
    public void powerAgentAction() {...}

    /** Composes the action method from AlertPowerUp. */
    @Around("execution(boolean AlertPowerUp.action())")
    public void alertPowerUpAction() {...}

    /** Composes the done method from AlertPowerUp behavior.
     * @param finished */
    @Around("execution(boolean AlertPowerUp.done())")
    public void alertPowerUpDone(final boolean finished) {...}

    /** Sets the system on composing the turnOn method from ANT. */
    @Around("execution(void ANT.turnOn())")
    public void turnSystemOn(ANT _ant) {  }
}
```

A classe `PowerAgent`, que na verdade representa o agente responsável pelo gerência dos recursos energéticos do sistema, está apresentada a seguir. Seus métodos `setUp()` e `action()` são implementados pelo aspecto `PowerAspect`, utilizando os adendos `powerAgentSetup()` e `powerAgentAction()`.

```
/**
 * This class represents the Power Agent.
 * @author Marcos
 */
public class PowerAgent extends Agent {
    /** Action method. */
    public void action() { }

    /** Sets up the agent. */
    protected void setUp() { }
}
```

O comportamento `AlertPowerUp` representa um comportamento do tipo simples, definido pelo próprio JADE. Este comportamento possui os métodos `done()` e `action()`,

implementados pelos métodos `alertPowerUpDone()` e `alertPowerUpAction()`, do aspecto apresentado anteriormente.

```
/**
 * This class represents the Alert Power Up behavior.
 * @author Marcos
 */
public class AlertPowerUp extends SimpleBehaviour {
    /** Action method. */
    public void action() { }

    /** Done Method.
     * @return <code>true</code> if this behavior has been done, or
     *         <code>false</code> otherwise. */
    public boolean done() {
        return true;
    }
}
```

5 Conclusão

Neste trabalho foi apresentada uma implementação para o estudo de caso da NASA, apontado por [3]. O estudo de caso na verdade é uma simulação para que fosse possível verificar algum ganho em relação a implementação utilizando LP-SMA. Dessa forma, este trabalho não foge desse contexto, no qual as expectativas são baseadas nos resultados de várias simulações possíveis.

Embora o autor tenha fornecido algumas abstrações no que se refere à modelagem do estudo de caso, esse trabalho se constitui numa revisão completa, já que foram utilizados outros elementos de implementação, como o *Spring AOP* por exemplo. Entretanto, o modelo de *features* foi totalmente reaproveitado, mesmo existindo a sua versão em UML, que de fato é idêntica semanticamente.

Em se tratando de implementação, o trabalho presente em [3] se dispõe de forma carente, não apresentando em sua abordagem, as técnicas de implementação que foram utilizadas. Por fim, esse trabalho foi feito com o objetivo de suprir tal necessidade, apresentando uma modelagem mais detalhada e com abstrações muito mais próximas da camada de implementação.

Referências

[1] K. Pohl, G. Böckle, and F. J. van der Linden, *Software Product Line Engineering : Foundations, Principles and Techniques.* Springer, September 2005. [Online]. Available: http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/3540243720

[2] J. Pena, M. G. Hinchey, and A. Ruiz-Cortés, "Multi-agent system product lines: challenges and benefits," *Commun. ACM*, vol. 49, no. 12, pp. 82–84, 2006.

[3] J. Pena, M. G. Hinchey, M. Resinas, R. Sterritt, and J. L. Rash, "Designing and managing evolving systems using a mas product line approach," *Sci. Comput. Program.*, vol. 66, no. 1, pp. 71–86, 2007.

[4] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," 1997, pp. 220–242. [Online]. Available: http://dx.doi.org/10.1007/BFb0053381

[5] V. Alves, P. M. Jr., L. Cole, A. Vasconcelos, P. Borba, and G. Ramalho, "Extracting and evolving code in product lines with aspect-oriented programming," in *Transactions on Aspect-Oriented Software Development*, 2007.

[6] S. Apel, T. Leich, and G. Saake, "Aspectual mixin layers: aspects and features in concert," in *ICSE '06: Proceedings of the 28th international conference on Software engineering.* New York, NY, USA: ACM, 2006, pp. 122–131.

[7] J. V. Gurp, J. Bosch, and M. Svahnberg, "On the notion of variability in software product lines," in *WICSA '01: Proceedings of the Working IEEE/IFIP Conference on Software Architecture.* Washington, DC, USA: IEEE Computer Society, 2001, p. 45.

[8] P. Clements and L. Northrop, *Software product lines: practices and patterns.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001, vol. 0201703327.

[9] V. Alves, "Implementing software product line adoption strategies," Ph.D. dissertation, Federal University of Pernambuco, 2007.

[10] P. Sochos, I. Philippow, and M. Riebisch, *Feature-Oriented Development of Software Product Lines: Mapping Feature Models to the Architecture*, Springer, Ed. Springer Berlin / Heidelberg, 2004. [Online]. Available: http://www.springerlink.com/content/ku9jm8lcnfc6q7gy

[11] K. Czarnecki, S. Helsen, and U. Eisenecker, "Staged configuration through specialization and multilevel configuration of feature models," *Software Process: Improvement and Practice*, vol. 10, no. 2, pp. 143–169, 2005. [Online]. Available: http://dx.doi.org/10.1002/spip.225

[12] J. J. Odell, H. Van, H. V. D. Parunak, and M. Fleischer, "The role of roles in designing effective agent organizations," in *Software Engineering for Large-Scale Multi-Agent Systems, LNCS 2603.* Springer, 2003, pp. 27–38.

[13] M. Mezini and K. Ostermann, "Variability management with feature-oriented programming and aspects," *SIGSOFT Softw. Eng. Notes*, vol. 29, no. 6, pp. 127–136, 2004.

[14] E. Figueiredo, N. Cacho, C. Sant'Anna, M. Monteiro, U. Kulesza, A. Garcia, S. Soares, F. Ferrari, S. Khan, F. Castor Filho, and F. Dantas, "Evolving software product lines with aspects: an empirical study on design stability," in *ICSE '08: Proceedings of the 30th international conference on Software engineering.* New York, NY, USA: ACM, 2008, pp. 261–270.

[15] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold, “Getting started with aspectj,” *Commun. ACM*, vol. 44, no. 10, pp. 59–65, 2001.

[16] C. V. Lopes and G. Kiczales, “Recent developments in aspectj.” Springer Verlag, 1998, pp. 398–401.

[17] J. Pena, M. G. Hinchey, A. Ruiz-Cortes, and P. Trinidad, “Building the core architecture of a nasa multiagent system product line,” in *In 7th International Workshop on Agent Oriented Software Engineering 2006*, 2006.

[18] J. Rumbaugh, I. Jacobson, and G. Booch, *Unified Modeling Language Reference Manual, The (2nd Edition).* Pearson Higher Education, 2004.

[19] Object Management Group, “Omg unified modeling language (omg uml), infrastructure, v2.1.2,” http://www.omg.org/docs/formal/07-11-04.pdf, November 2007.