UNIVERSIDADE FEDERAL DE ALFENAS
DEPARTAMENTO DE CIÊNCIAS EXATAS
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

Lucas Schmoeller do Prado Rodrigues

IMPLEMENTAÇÃO DA EXECUÇÃO DE PROCESSOS UTILIZANDO O META-MODELO SPEM NO JOGO EDUCATIVO SPARSE

Alfenas, 03 de dezembro de 2010.

UNIVERSIDADE FEDERAL DE ALFENAS
DEPARTAMENTO DE CIÊNCIAS EXATAS
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

IMPLEMENTAÇÃO DA EXECUÇÃO DE PROCESSOS UTILIZANDO O META-MODELO SPEM NO JOGO EDUCATIVO SPARSE

Lucas Schmoeller do Prado Rodrigues

Monografia apresentada ao Curso de Bacharelado em Ciência da Computação da Universidade Federal de Alfenas como requisito parcial para obtenção do Título de Bacharel em Ciência da Computação.

Orientadora: Profª. MSc Mariane Moreira de Souza

Alfenas, 03 de dezembro de 2010.

Lucas Schmoeller do Prado Rodrigues

IMPLEMENTAÇÃO DA EXECUÇÃO DE PROCESSOS UTILIZANDO O META-MODELO SPEM NO JOGO EDUCATIVO SPARSE

A Banca examinadora abaixo-assinada aprova a monografia apresentada como parte dos requisitos para obtenção do título de Bacharel em Ciência da Computação pela Universidade Federal de Alfenas.

Prof. MSc Humberto César Brandão de Oliveira

Universidade Federal de Alfenas

MSc Leonardo Aparecido Ciscon

Devex Tecnologia e Sistemas S.A.

Profª. MSc Mariane Moreira de Souza (Orientadora)

Universidade Federal de Alfenas

Alfenas, 03 de dezembro de 2010.

Dedico este trabalho ao meu pai por me guiar pelo caminho que escolhi seguir.

AGRADECIMENTO

Agradeço a Profª Dra. Melise Maria Veiga de Paula por ter feito com que eu continuasse mesmo quando estava pronto pra desistir;

Ao professor Humberto César Brandão de Oliveira por acreditar no meu potencial mesmo quando eu mesmo o questionava;

À professora Mariane Moreira de Souza por ser mais que uma orientadora, uma grande amiga;

Ao professor Rodrigo Martins Pagliares por introduzir novas perspectivas para mim, tanto ao referente a este trabalho quanto a vida profissional;

Aos amigos Alexsander Rodrigues, Juliana Ferreira e Edgar Franco por me ajudarem tanto no trabalho quanto durante toda a minha graduação;

Aos amigos Luci Shoji, Hugo Cardoso e Thiago Reis, que sempre foram verdadeiros amigos, pra qualquer hora;

Agradeço à minha irmã Marina, por sempre me estar ao meu lado mesmo quando pudesse desagradá-la;

E mais importante, agradeço a minha mãe por me ajudar em todo o momento que eu precisei de qualquer tipo de ajuda e por ele, junto a meu pai permitir que eu iniciasse e concluísse meus estudos sem preocupações.

"A educação é aquilo que permanece depois que tudo o que aprendemos foi esquecido"

Burrhus Frederic Skinner

RESUMO

A metodologia tradicional de ensino de Engenharia de Software, através de aulas teóricas complementadas com o desenvolvimento de pequenos projetos, utilizada na maioria dos cursos na área de Computação, apresenta falhas. Como conseqüência destas tem-se a falta de preparação de recém-graduados para atuação no mercado de trabalho. Para endereçar este problema, encontram-se na literatura diversas propostas. O SPARSE é um jogo educativo que procura resolver este problema, utilizando simulação com foco no ensino de modelos de processo de software. De forma a tornar o SPARSE uma ferramenta mais abrangente, este trabalho utiliza o meta-modelo SPEM como base para os modelos de processo simulados pelo jogo, tornando possível aumentar a capacidade do simulador e o nível de conhecimento agregado pelo mesmo. Este objetivo foi alcançado através da construção de um módulo de importação, que permite que processos SPEM, produzidos utilizando a ferramenta EPF *Composer*, sejam assimilados pelo SPARSE.

Palavras-Chave: Engenharia de Software, jogos e simulação, *enactment* de processos, SPEM, SPARSE

ABSTRACT

The traditional method of teaching Software Engineering, through lectures supplemented by the development of small projects, used in most courses in Computer Science presents flaws. As a consequence of these flaws there is a lack of preparation of recent graduates to operate in the market. To address this problem, there are several proposals in the literature. SPARSE is an educational game that aims to solve this problem by using simulation models with focus on teaching software process. To turn SPARSE into a more comprehensive tool, this work uses the SPEM metamodel as a basis for process models simulated by the game. By doing so, it can increase the capacity of the simulator and the level of knowledge assembled by the game. This was achieved through the building of an import module that allows SPEM processes, produced using the EPF *Composer* tool, be assimilated by SPARSE.

Keywords: Software Engineering, process enactment, games and simulation, SPEM, SPARSE

LISTA DE FIGURAS

LISTA DE ABREVIAÇÕES

EPF *Composer*	*Eclipse Process Framework Composer*
OMG	*Object Management Group*
OpenUP	*Open Unified Process*
SGML	*Standard Generalized Markup Language*
SPEM	*Software Process Engeneering Metamodel*
SPARSE	*Software Process semi-automated tool for Software Engineering*
XML	*eXtensible Markup Language*
W3C	*World Wide Web Consortium*
CASE	*Computer-Aided Software Engineering*
UML	*Unified Modeling Language*

SUMÁRIO

1
Introdução

Este capítulo apresenta as razões pelas quais este trabalho foi desenvolvido, bem como o que este pretende resolver e a proposta de como realizar esta resolução. Na seção 1.1 são apresentadas as justificativas para o desenvolvimento deste trabalho. A seção 1.2 apresenta o problema central que este trabalho procura resolver. Na seção 1.3 são apresentados os objetivos que se pretende atingir com este trabalho. E a seção 1.4 apresenta resumidamente a organização desta monografia

1.1 Justificativa e Motivação

Atualmente o ensino de Engenharia de Software é realizado em forma de aulas ministradas e complementação destas aulas através da realização de pequenos projetos afim de por em prática as técnicas e conceitos aprendidos em sala de aula (Navarro 2006, Baker *et al* 2004). Porém esta abordagem de ensino tem se mostrado falha, pois em um ambiente industrial é necessário que os graduados nos cursos de computação passem por treinamentos e preparações dentro da empresa para que estes possam realizar um trabalho efetivo (Baker *et al* 2004).

Navarro (2006) mostra que grande parte das falhas de software atualmente estão relacionadas a falhas no processo de Engenharia de Software destes. Isso ocorre devido a erros na execução de um processo de software, falha no gerenciamento de equipes, erros na configuração do tempo do projeto, entre outras disciplinas relacionadas à Engenharia de Software. Acredita-se ainda que a razão para tais erros seja a metodologia aplicada no ensino desta área (Navarro 2006).

Segundo Drappa *et al* (2000), a chave para um bom ensino é a motivação do aprendiz e, no caso da gerência de projetos de software, a melhor motivação pode ser adquirida através da experiência em projetos aplicados, que podem falhar devido à falta de gerenciamento necessário. Porém tanto Navarro (2006) quanto

Baker *et al* (2004) concordam ao dizer que não existe tempo hábil durante a graduação para que seja aplicado um projeto de tamanho suficiente para que os estudantes possam vivenciar a realidade do desenvolvimento de software na indústria.

Tendo em vista esta situação, vários acadêmicos vêm tentando encontrar formas diversificadas de ensino para a Engenharia de Software. Neste contexto o uso de jogos e simulação pode ser visto nos softwares SimSE (Navarro 2006) e SPARSE (Souza *et al.* 2010).

O SPARSE é um jogo educativo que tem com objetivo ensinar técnicas de gerenciamento de projetos e principalmente ensinar a aplicação de processos de desenvolvimento de software na realização destes projetos (Souza *et al.* 2010). Este trabalho tem por finalidade permitir ao SPARSE utilizar como base de ensino qualquer processo desenvolvido via EPF *Composer* (vide Cap. 2 . Seção 2.4) utilizando o meta-modelo SPEM (vide Cap. 2, Seção 2.3). Os objetivos deste trabalho e o problema que este pretende resolver são apresentados nas Seções subseqüentes.

1.2 Problematização

Levando em consideração o problema do ensino de Engenharia de Software supracitado e sabendo da tentativa dos acadêmicos de formular novas maneiras para o ensino da disciplina, pode-se afirmar, dentro do contexto do SPARSE, que: existe uma grande dificuldade no ensino de diferentes modelos de processo a alunos de graduação (Navarro 2006); os recém-graduados nos cursos de computação costumam cometer erros por falta de experimentação de situações reais de um ambiente de trabalho durante a graduação (Navarro 2006, Baker *et al.* 2004); o SPARSE tenta resolver este problema através do uso de jogos e simulação (Souza *et al.* 2010); e na versão atual do SPARSE é necessária a reprogramação do jogo para a adição de novos modelos de processo[1].

xxivxxiv

[1] Ver Capítulo 2 Seção 2.6

Sabendo disso torna-se necessário definir uma forma de reestruturar o SPARSE, para que este possa simular uma maior diversidade de modelos de processo sem que haja a necessidade de adição de código para cada um dos modelos.

Neste contexto deseja-se averiguar:

- Qual a forma de se importar processos de desenvolvimento de software para o SPARSE, criados utilizando o meta-modelo SPEM no software EPF *Composer*?

1.3 Objetivos

1.3.1 Gerais

Este trabalho visa permitir ao SPARSE se integrar ao SPEM de forma a possibilitar a simulação de todo e qualquer processo modelado via EPF *Composer*, e exportado para o jogo através do módulo de importação desenvolvido neste trabalho. Desta forma pode-se definir como objetivo geral deste trabalho a construção de um módulo de importação que integre processos criados via EPF *Composer* ao SPARSE de maneira a tornar possível a simulação dos mesmos dentro do jogo.

1.3.2 Específicos

Para que o objetivo geral deste projeto seja atingido é necessário que os seguintes objetivos específicos sejam alcançados:

- Escolha de um método de importação de processos disponível no EPF *Composer*;
- Exportação de um *capability pattern* do processo OpenUP Basic através do EPF *Composer*;
- Realização de modificações no SPARSE de maneira a permitir a simulação de qualquer processo SPEM;

- Criar um módulo para importar um *capability pattern* exportado pelo EPF *Composer* para o SPARSE;
- Importar um *capability pattern* do OpenUP Basic para o SPARSE.

1.4 Organização da Monografia

Este trabalho encontra-se organizado da seguinte forma: No Capítulo 2 apresenta-se todo o referencial teórico necessário para o entendimento deste trabalho; No Capítulo 3 é apresentada a proposta deste trabalho, bem como todos os passos utilizados para atingir os objetivos do mesmo; No Capítulo 4 são apresentados os resultados deste trabalho e a validação do mesmo; Finalmente no Capítulo 5 são apresentadas as considerações finais da monografia.

2
Revisão Bibliográfica

Este capítulo apresenta o estado da arte dos conceitos necessários para um melhor entendimento desta monografia. Inicialmente são definidos alguns termos de Engenharia de Software. Posteriormente apresenta-se o meta-modelo SPEM, a ferramenta para autoria de processos EPF *Composer, a linguagem XML e o jogo SPARSE, respectivamente. Finalmente é apresentado o modelo de processo OpenUP que será utilizado para a validação desta monografia.*

2.1 Considerações Iniciais

Para contextualizar este trabalho é necessário que os conceitos de Engenharia de Software utilizados sejam apresentados e que os termos utilizados sejam explanados. A Seção 2.2 apresenta estas definições.

A partir das definições da Seção 2.2 a Seção 2.3 apresenta o SPEM adotado como meta-modelo a ser utilizado na confecção deste trabalho.

Em seguida a Seção 2.4 apresenta o EPF *Composer*, software utilizado para a autoria de processos de software utilizando a estrutura do SPEM, este software será utilizado neste trabalho para a exportação de processos.

Posteriormente serão apresentados o XML (tecnologia utilizada para realizar a importação/exportação de processos), o SPARSE (software para a simulação de processos de Engenharia de Software com fins educativos) e finalmente o processo de software OpenUP Basic utilizado para a validação deste trabalho. Estes temas são apresentados nas Seções 2.5, 2.6 e 2.7, respectivamente.

2.2 Engenharia de Software

Antes de discutir o que é Engenharia de Software é necessário definir o termo software. Um software é um conjunto composto por programas, procedimentos, dados e documentação, associados a um sistema de computador (Pressman 1995, Sommerville 2007, Pfleeger 2004). Para Pressman (Pressman 1995) ainda devem ser levadas em consideração as seguintes características do software para defini-lo:

- O software não é manufaturado, é desenvolvido ou projetado por engenharia.
- Software não se desgasta, ou seja, o passar do tempo não induz erros ou ocorrência de falhas
- A maior parte dos softwares é feito sob medida e não é montada a partir de componentes pré-existentes

A Engenharia de Software é definida então como a aplicação dos princípios da engenharia em todos os aspectos da produção de software (Paula 2009, Pressman 1995, Sommerville 2007). Neste contexto, Pressman cita três elementos fundamentais que compõe a Engenharia de Software:

- Métodos – Definem maneiras de se desenvolver partes do software e as atividades a serem realizadas. Segundo Pfleeger (2004) são procedimentos formais para a realização de tarefas no desenvolvimento de software.
- Ferramentas – São instrumentos ou sistemas automatizados ou semi-automatizados que oferecem suporte à realização de uma ou mais atividades definidas nos métodos. Ferramentas que integram diversas atividades são denominadas CASE (Computer-Aided Software Engeneering) (Pressman 1995, Pfleeger 2004).
- Procedimentos – Segundo Pfleeger (2004) são o resultado da integração harmônica de métodos e ferramentas. Procedimentos são responsáveis por definir a seqüência de aplicação de métodos, a definição de produtos a serem entregues, controle de qualidade e

mudanças, e a definição de marcos para a avaliação de progresso de um projeto (Pressman 1995).

A Engenharia de Software compreende um conjunto de etapas envolvendo os três elementos supracitados. Este conjunto de etapas é chamado de paradigma (Pressman 1995).

Segundo Pfleeger (2004) um paradigma representa uma abordagem ou filosofia em particular para construção de um software. Vale ressaltar que um paradigma não é melhor ou pior que outro, de modo que cada paradigma tem um conjunto de vantagens e desvantagens (Pfleeger 2004).

Ao aplicar Engenharia de Software para a criação de um novo produto uma seqüência de etapas é utilizada para conclusão de tarefas, tarefas estas que devem ser realizadas em uma determinada ordem. É chamado processo uma série de atividades ordenadas para atingir um determinado objetivo (Paula 2009, Pressman 1995, Sommerville 2007).

É importante não confundir os conceitos de procedimento e processo. Para tal, Pfleeger (2004) define processo como um conjunto de procedimentos organizados para satisfazer objetivos e padrões determinados.

Na literatura é possível encontrar diversas descrições de processos, geralmente em forma de modelos, conhecidos como modelos de processo de software ou modelos de ciclo de vida (Paula 2009, Pfleeger 2004).

Paula (2009) descreve os modelos de ciclo de vida divididos da seguinte forma:

- Ciclos de vida em cascata – Os subprocessos deste processo são executados em seqüência (Requisitos, Análise, Desenho, Implementação e Testes). Isto torna o processo a princípio confiável para qualquer projeto, mas na prática é aconselhável para projetos pequenos pois demanda muito trabalho nas fases de análise e projeto (Paula 2009). Vale ressaltar que a denominação dos subprocessos deste modelo é encontrada de diversas formas na literatura. Este modelo de ciclo de vida é conhecido como o modelo clássico da Engenharia de Software (Pressman 1995).

- Ciclos de vida em espiral – Neste modelo o software é desenvolvido em diversas interações, cada uma destas utilizando os subprocessos definidos anteriormente no modelo de ciclo de vida em cascata (Paula 2009). Para cada uma das interações é possível que haja uma liberação de um estágio parcialmente operacional do produto para avaliação do usuário, chamados de liberação ou *release* (Paula 2009).
- Outros modelos de ciclo de vida – Paula cita dois tipos de modelo além dos supracitados, dirigidos por caso de uso e dirigidos por prazo. Os dirigidos por prazo, nos quais o produto é aquilo que é possível fazer dentro de um determinado prazo. Este último indicado quando requisitos essenciais são estabelecidos e possíveis de implementar no prazo estipulado. Os dirigidos por ferramenta são processos definidos por fabricantes de ferramentas CASE e a qualidade destes processos está diretamente ligada à qualidade da ferramenta.

É importante notar que existem diversas variações para estes ciclos de vida. Mais referências a modelos de ciclo de vida podem ser encontradas em Pressman (1995), Sommervile (2007), Pfleeger (2004) e Paula (2009).

Embora existam vários modelos de processo propostos, é um consenso na comunidade que não existe um que se aplica a todas as situações de desenvolvimento (Bezerra 2007).

Devido ao fato de existirem vários processos diferentes e a necessidade existente da simulação de diferentes processos para o aprendizado como dito no Capítulo 1, é necessário que se utilize um padrão para o desenvolvimento de processos de forma a facilitar a simulação de processos. Os meta-modelos são modelos para modelos (OMG 2006) e através deles é possível realizar um nível maior de abstração de processo. É possível encontrar diversos meta-modelos na literatura, porém para este trabalho será utilizado o SPEM apresentado na Seção subseqüente.

2.3 *Software Process Engineering Metamodel*

O SPEM (*Software Process Engineering Metamodel*) foi criado pelo *Object Management Group* (OMG), para ser um padrão de alto nível para processos usados em desenvolvimento de software (Henderson-Sellers 2005). Inicialmente, o SPEM foi criado como um meta-modelo independente e mais tarde foi estendido como um perfil da UML, ou seja, tornou-se possível a autores de processo transformar concepções criadas orientadas a processo em concepções orientadas a modelo (Henderson-Sellers 2005, Nardini 2008, Paula 2009). Atualmente o meta-modelo encontra-se na versão 2.0, faz uso de outras especificações da OMG e reutiliza a estrutura da UML (*Unified Modeling Language*) 2 sempre que possível (OMG 2008).

O SPEM trata um processo de desenvolvimento de software como uma colaboração de entidades ativas (papéis) direcionadas à realização de operações específicas, chamadas atividades, em um conjunto de artefatos, nomeados produtos de trabalho, até que estes artefatos são levados a um estado bem definido e declarado completo. O meta-modelo possui então uma estrutura central que representa um processo de software através de papéis, os produtos de trabalho pelos quais cada papel é responsável e as atividades que estes papéis realizam nestes produtos (Ransin, 2008). Essa organização geral é mostrada na Figura 1, ressaltando que a estrutura completa do SPEM é muito mais complexa que esta estrutura central. A arquitetura do SPEM é discutida com detalhes na próxima Seção.

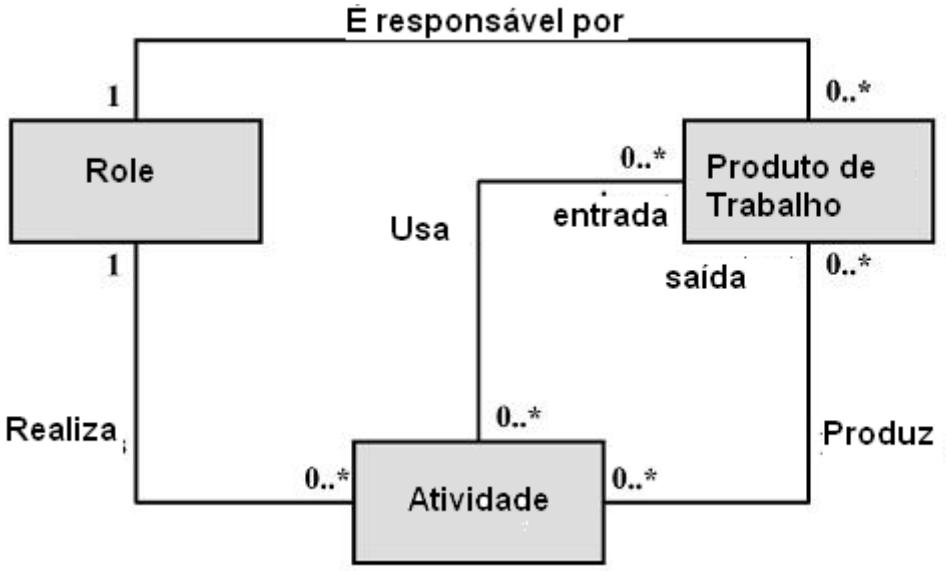

Figura 1 - Estrutura Central de um Processo como definida pelo SPEM (Ramsin 2008).

2.3.1 Arquitetura do SPEM

O SPEM 2.0 é dividido em sete principais pacotes de meta-modelos: *Core* (Núcleo), *Process Structure* (Estrutura do Processo), *Process Behavior* (Comportamento de Processo), *Managed Content* (Conteúdo Gerenciado), *Method Content* (Conteúdo de Métodos), *Process with Methods* (Processo com Métodos) e *Method Plugin* (*Plugin* de Métodos). Tais pacotes são responsáveis por encapsular algumas das capacidades do SPEM 2.0. As capacidades de cada pacote são apresentadas nas próximas Seções.

Como é mostrado na Figura 2, os pacotes de meta-modelos se relacionam através do *merge* definido na UML 2.0. Este relacionamento torna possível que classes de camadas mais baixas na arquitetura sejam utilizadas para realizar uma sub-implementação do SPEM sem a necessidade das classes mais superiores (OMG 2008). Essa característica está melhor apresentada na Seção 1.2.

A independência entre camadas se torna mais clara com a descrição dos pacotes e a apresentação dos pontos de conformidade do SPEM. As próximas subSeções descrevem brevemente as capacidades de cada um dos pacotes de meta-

modelo que compõe o SPEM 2.0 e a Seção 1.2 apresenta os pontos de conformidade do SPEM.

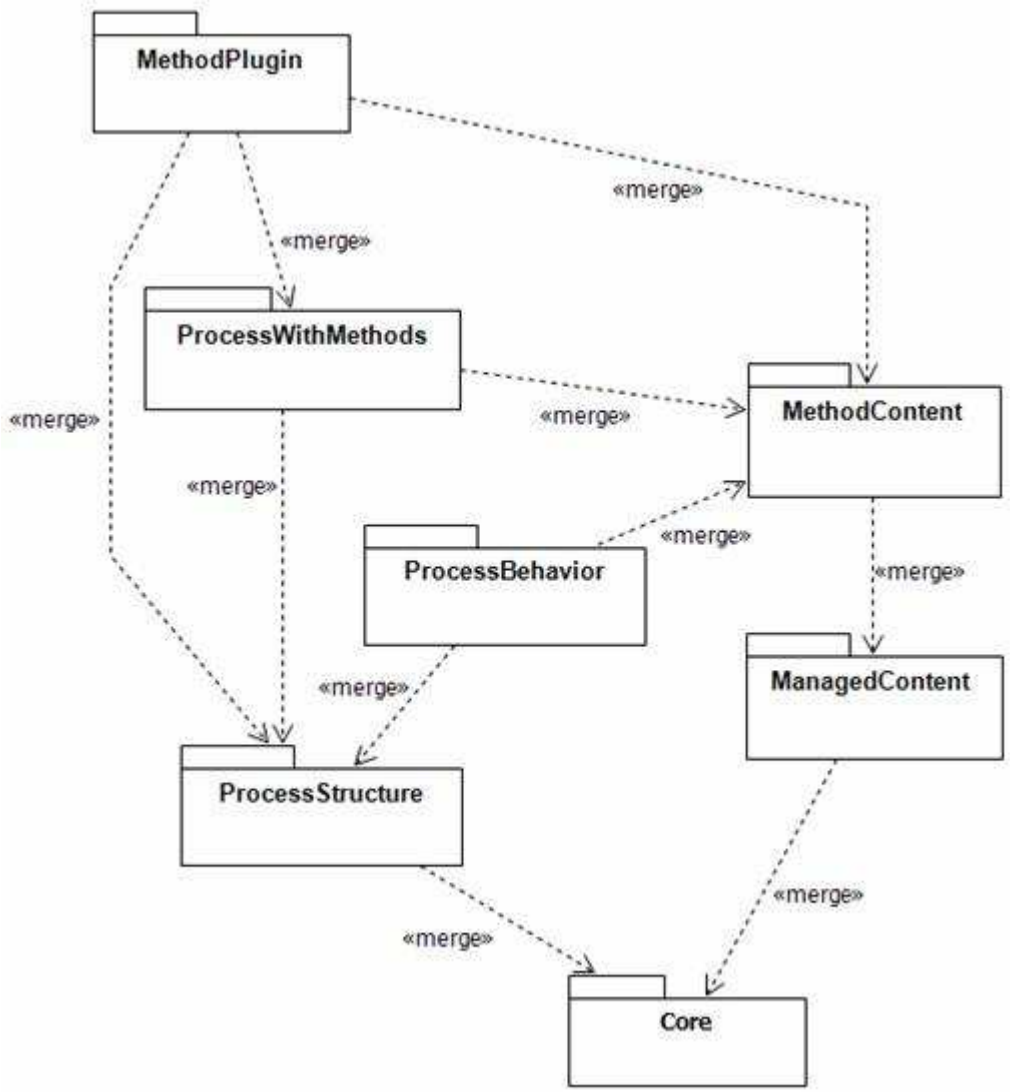

Figura 2 - Arquitetura do SPEM com subdivisão em 7 pacotes de meta-modelos (OMG 2008)

2.3.2 *Core*

Este pacote é composto por classes meta-modelo e abstrações que servem de base para todos os outros pacotes de meta-modelo. Basicamente define classes para duas capacidades do SPEM (OMG 2008):

- A capacidade de um usuário do SPEM de criar qualificações próprias para classes SPEM 2.0, permitindo assim a distinção entre diversos "tipos" de instâncias de classe SPEM 2.0.
- Um conjunto de classes abstratas para definição de trabalho expressado por processos SPEM 2.0

2.3.3 *Process Structure*

Define a base para todos os modelos de processo. Permite a criação de modelos de processo simples e flexíveis. A estrutura central deste pacote é uma estrutura analítica ou uma decomposição de atividades aninhadas que mantém referência às classes *Role* (Papel) responsáveis por estas, bem como para elementos de entrada e saída para cada uma das atividades. Além disso, este pacote permite a criação dinâmica de padrões de processo permitindo assim reuso de processo. As estruturas disponibilizadas neste pacote são ideais para a construção de processos ágeis (OMG 2008).

2.3.4 *Process Behavior*

Este pacote permite estender as estruturas do pacote de meta-modelo *Process Structure* com modelos comportamentais. O pacote não define abordagem própria de modelagem, mas provê "ligações" para modelos comportamentais existentes, permitindo reuso de outras especificações da OMG. Por exemplo, é possível uma estrutura de processo existente ser ligada a um diagrama de atividades da UML 2 (OMG 2008).

2.3.5 *Managed Content*

Além da representação por modelos, é comum em processos de desenvolvimento de software a existência de documentos em linguagem natural, e estes, por muitas vezes, são mais importantes que modelos que garantem obediência a um processo formal, pois proporcionam orientação inteligível para as melhores práticas.

O pacote de meta-modelo *Managed Content* apresenta conceitos para a captura de documentos informais através de meta-classes de orientação, por exemplo um documento que contenha linhas guia para um determinado processo pode ser capturados através de classes de direcionamento (*guidance*). Capturados, estes conceitos podem ser utilizados isoladamente ou em combinação com concepções de estruturas de processos. Ou seja, um processo SPEM pode ser composto somente por instâncias de meta-classes de orientação, ou composto por instâncias destas meta-classes combinadas a elementos de estrutura de processo, através de relacionamentos definidos no pacote de meta-modelo *Managed Content* (OMG 2008).

2.3.6 *Method Content*

Este pacote proporciona ao usuário meios para a construção de uma base de conhecimento em desenvolvimento de software, independente de processo e projeto específicos. Adiciona conceitos para a definição de ciclos de vida (ciclo de vida em cascata, ciclo de vida em espiral, etc) e elementos de conteúdo de método, que permitem a criação de uma base para conhecimento de métodos de desenvolvimento de software, técnicas e realizações concretas de melhores práticas. Este pacote é composto de explicações textuais passo-a-passo de como objetivos específicos de desenvolvimento são atingidos, por quais papéis, utilizando quais recursos e quais os resultados, independente da posição deste objeto dentro de um ciclo de vida específico. Processos podem fazer uso destas estruturas e definí-las em seqüências parcialmente organizadas que podem ser customizadas para projetos específicos, por exemplo, é possível utilizar práticas usadas em vários projetos anteriormente e customizá-las para um novo projeto de caráter semelhante à estes anteriores. (OMG 2008)

2.3.7 *Process with Methods*

Este pacote define novas estruturas e redefine estruturas para a integração de processos definidos nos conceitos do pacote de meta-modelo *Process Structure* com instâncias dos conceitos do pacote de meta-modelo *Method Content.* Assim como o pacote *Method Content* define métodos e técnicas fundamentais para o

desenvolvimento de software, processos colocam métodos e técnicas no contexto de um ciclo de vida composto, por exemplo, de fases e produtos de trabalho. Quando conteúdos de métodos, tais como tarefas (*Tasks*), papéis (*Roles)*, e produtos de trabalho (*Work Products*), são aplicados, classes de referência são criadas, podendo armazenar mudanças individuais a cada uma das classes de *Method Content* às quais se referem. Estas mudanças demonstram como e quais partes do método serão aplicadas em um ponto específico do processo (OMG 2008).

2.3.8 *Method Plugin*

Neste pacote encontram-se conceitos para o "desenho" e gerenciamento de bibliotecas ou repositórios de conteúdo de métodos e processos de larga-escala, reutilizáveis, configuráveis e que possam sofrer manutenção. Os conceitos introduzidos neste pacote permitem organizar diferentes conteúdos de tal biblioteca em diferentes camadas (similares a arquitetura de software em camadas). Com os conceitos deste pacote é possível definir processos que são estendidos, com base na granularidade, com mais e mais capacidades. Desta forma é possível ao usuário selecionar as capacidades de método nas quais esteja interessado, permitindo ao mesmo o acesso somente às configurações selecionadas. Isto torna possível que autores de processo construam processos para diferentes casos que são configuráveis de acordo com as características do usuário final (OMG 2008).

2.3.9 Níveis de Conformidade do SPEM

Como dito anteriormente nem sempre é necessário a um usuário utilizar todos os pacotes do SPEM para o desenvolvimento de um processo. Sendo assim, existem os pontos de conformidade do meta-modelo que apresentam quais pacotes devem ser utilizados e para quais tipos de usuários eles são indicados. Vale lembrar ainda que é possível utilizar partes do meta-modelo independente destes níveis de conformidade.

De acordo com a OMG, 2008, o SPEM possui três pontos de conformidade. Estes três pontos são apresentados nas subSeções seguintes.

2.3.9.1 Ponto de conformidade "*SPEM-Complete*"

Este ponto de conformidade compreende todos os sete pacotes de meta-modelo que compõem o SPEM, o nível de conformidade LM da biblioteca de infraestrutura UML 2[2] e o perfil da UML 2 (OMG 2008 pág 5).

O "*SPEM-Complete*" é recomendado para implementadores que necessitam de todas as capacidades do SPEM, com foco na distribuição de ferramentas CASE (*Computer-Aided Software Engineering*). Este ponto tem como objetivo o auxílio na gerência de diversos processos interrelacionados para empresas complexas. Além disso, é o único ponto de conformidade que contém o perfil UML 2, o que permite uma maior extensão da estrutura do SPEM 2.0. Algumas ferramentas de criação de processo utilizam este ponto de conformidade como, por exemplo, o EPF *Composer* da Eclipse Foundation (OMG 2008).

A Figura 3 representa a arquitetura do "*SPEM-Complete*". Os pacotes que não são apresentados na figura encontram-se inseridos no diagrama através do relacionamento de *merge* entre os pacotes de meta-modelo que compõe a arquitetura do SPEM 2.0, como dito anteriormente.

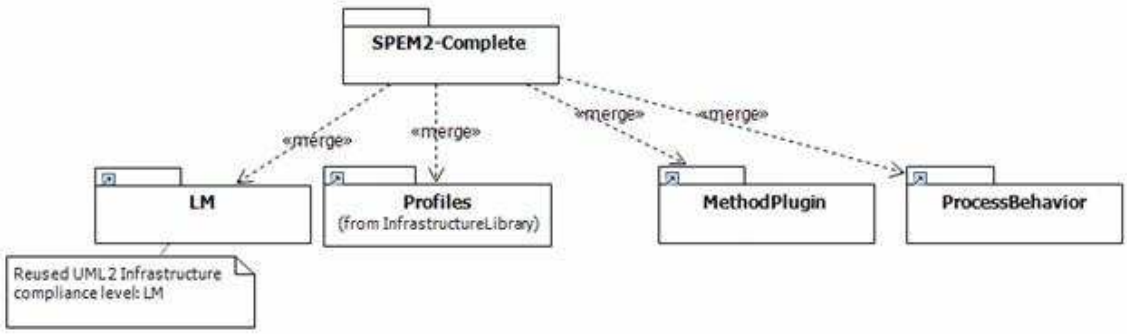

Figura 3 - Ponto de conformidade "SPEM-Complete" (OMG 2008)

2.3.9.2 Ponto de conformidade "SPEM Process with Behavior and Content"

xxxviixxxvii

[2] Vide OMG 2010

Este ponto de conformidade engloba 4 dos 7 pacotes de meta-modelo do SPEM 2.0 (*Managed Content, Process Structure, Process Behavior* e *Core*) e nível de conformidade LM da biblioteca de infraestrutura da UML 2.0.

O "*SPEM Process with Behavior and Content*" dedica-se a implementadores que se focam em um processo por vez e trabalham basicamente com estruturas analíticas do trabalho e diagramas de fluxo de trabalho. Este ponto de conformidade é muito próximo em capacidade à versão anterior do SPEM (SPEM 1.X) e não compreende as capacidades adicionadas no SPEM 2.0 para desenvolvimento de bases de conhecimento em conteúdo de métodos, dimensionamento e variabilidade (OMG 2008).

A Figura 4 apresenta a arquitetura deste ponto de conformidade.

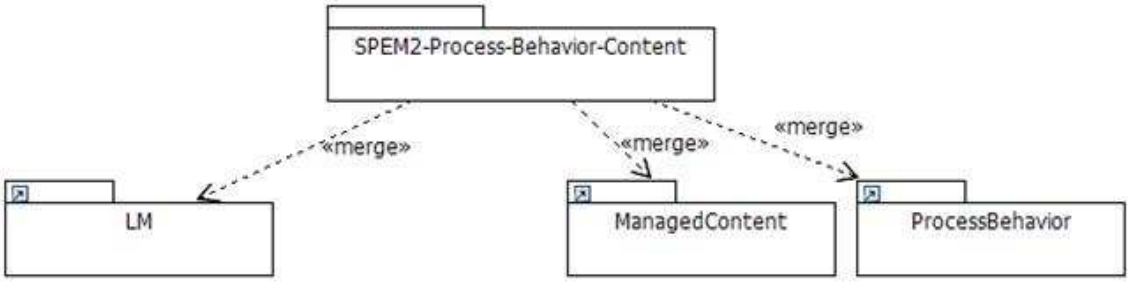

Figura 4 - Ponto de conformidade "SPEM Process with Behavior and Content"(OMG 2008)

2.3.9.3 Ponto de conformidade "*SPEM Method Content*"

Este ponto de conformidade compreende três pacotes de meta-modelo da arquitetura do SPEM 2.0 (*Managed Content, Core, Method Content*) e o nível de conformidade LM da biblioteca de infraestrutura da UML 2 como mostra a Figura 5.

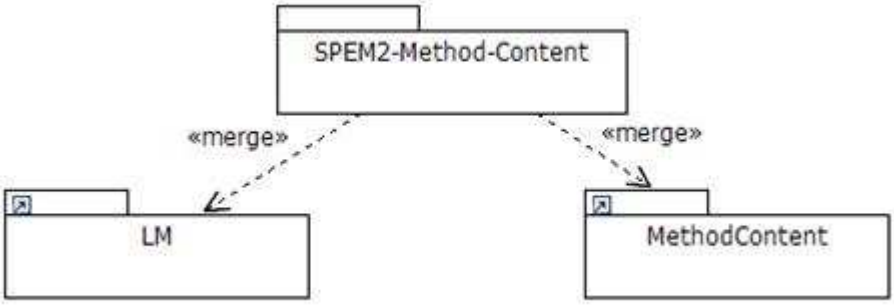

Figura 5 – Ponto de conformidade "SPEM Method Content" (OMG 2008)

O "*SPEM Method Content*" foca-se em desenvolvedores voltados a manter a documentação escrita de métodos de desenvolvimento, técnicas e melhores práticas. Estas informações podem ser utilizadas para o desenvolvimento de bases de conhecimento e sistemas de informação colaborativos, como *Wiki's* (OMG 2008).

O público-alvo deste ponto de conformidade normalmente não precisa ou não quer utilizar modelos formais de processo como representação para seus conteúdos. Normalmente eles utilizam descrições de seus métodos com o mínimo de conceitos possível, ou somente no formato de *white paper* (guia que apresenta problemas e maneiras detalhadas para resolve-los)ou outros meios de representação menos formais (OMG 2008).

2.4 *Eclipse Process Framework Composer*

Eclipse Process Framework Composer (EPF *Composer*) é uma plataforma de ferramentas para engenheiros de métodos e líderes de processo. Gerentes e líderes do processo são os responsáveis por implementar e dar manutenção aos processos para organizações de desenvolvimento ou projetos individuais (Haumer 2007).

O EPF *Composer* utiliza o meta-modelo SPEM, que é uma linguagem de modelagem, ou seja, um conjunto de construtores e regras para definir e modelar processos de software (OMG 2008). Com isso o EPF *Composer* é uma ferramenta que possibilita ao usuário a criação de métodos do zero, a customização, a publicação e o *enactment*, que é implantar o método criado no contexto de um projeto.

O EPF *Composer* tem dois objetivos principais (Haumer 2007):

- Fornecer para os praticantes de desenvolvimento um conhecimento que permita a busca, gerência e distribuição do conteúdo. Esse conteúdo pode ser licenciado, adquirido, e deve acomodar seu próprio conteúdo, por exemplo, definição de métodos, princípios, melhores práticas, procedimentos internos e regulamentações.
- Fornecer as capacidades de engenharia de métodos, oferecendo suporte aos engenheiros de métodos e aos gerentes de projeto para que possam selecionar, customizar, e rapidamente recolher processos para o desenvolvimento de projetos. O EPF *Composer* fornece catálogos de processos pré-definidos para as situações típicas de projeto que podem ser adaptadas às necessidades de cada indivíduo. Ele também fornece processos construídos por blocos chamados *capability patterns* que representam as melhores práticas de desenvolvimentos para disciplinas específicas, tecnologias, ou estilos de desenvolvimento.

Documentos criados com o EPF *Composer* podem ser publicados e distribuídos através de *web sites* independentes de manutenção, como mostra a Figura 6.

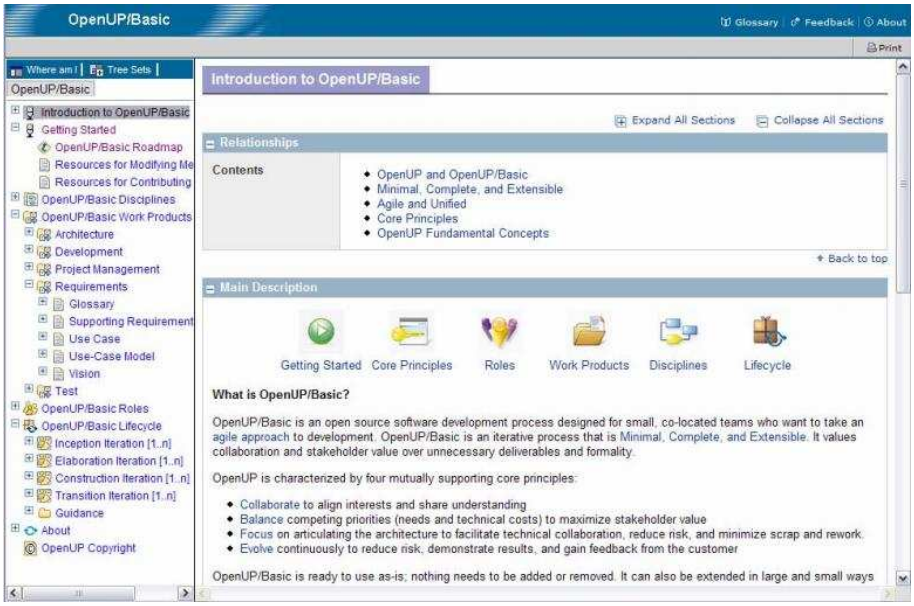

Figura 6 – Site publicado pelo EPF *Composer*

2.5 *Extensible Markup Language*

XML (*eXtensible Markup Language*) é uma linguagem para especificação de dados semi ou completamente estruturados. A linguagem vem sendo explorada cada vez mais tanto na área acadêmica quanto na industrial (Moro *et al.* 2009).

A XML descreve uma classe de objetos de dados chamados documentos XML e descreve parcialmente o comportamento de programas que os processam. É um perfil de aplicação ou uma forma restrita da SGML (*Standard Generalized Markup Language*) (ISO 8879). Desta forma documentos XML são por conformidade documentos SGML (W3C 2008).

A linguagem foi criada pelo *XML Working Group*, formado sobre a supervisão do W3C (*World Wide Web Consortium*) em 1996. Desenvolveu-se a XML com a finalidade de atingir os seguintes obejtivos (W3C 2008):

- Ser usada diretamente na Internet;
- Suportar uma grande variedade de aplicações;
- Ser compatível com SGML;
- Possibilitar a escrita de programas para processá-la facilmente;
- Manter no mínimo possível o número de capacidades adicionais do XML, idealmente zero;
- Criar documentos utilizando XML, de maneira inteligível para humanos.
- Criar *design* em XML rapidamente;
- O *design* da XML ser formal e conciso;
- Documentos XML serem fáceis de ser criados;
- Concisão em marcação XML ser de mínima importância.

Neste trabalho a XML é utilizada como linguagem para importação de *capability patterns,* exportados via EPF *Composer*, para o módulo de importação de processos SPEM. Além disso, também é utilizada para a geração do arquivo de

saída do módulo de importação que, por sua vez, servirá como arquivo de entrada para o SPARSE. A Figura 7 mostra um exemplo de arquivo XML[3].

Figura 7 – Estrutura de um arquivo XML

2.6 SPARSE

O SPARSE (*Software Project semi-Automated tool for Software Engineering*) é um jogo educativo desenvolvido para auxiliar no ensino e aprendizado de Engenharia de Software, combinando a teoria apresentada durante as aulas das disciplinas relacionadas à área com a prática, tornando possível a capacitação dos estudantes para a tomada de decisões necessárias em cenários reais (Souza et al. 2010).

O SPARSE utiliza um modelo de simulação baseado em regras para sua execução, ou seja, o sistema armazena um histórico de fatos ocorridos e permite a observação do comportamento individualizado de cada variável (Souza *apud* Barros 2001).

xliixlii

[3] Para mais informações sobre XML vide W3C 2008.

Alguns elementos essenciais no desenvolvimento de qualquer projeto de software foram considerados na construção do SPARSE (Souza *et al* 2010). São eles:

- Modelo de Processo: Um conjunto de atividades pré-definidas, e parcialmente organizadas para atingir um determinado objetivo, que um dado projeto deve seguir. Em um modelo de processo existem fases que representam estágios da produção de um software. Na implementação do SPARSE estas fases estão implementadas e se subdividem em etapas, que representam os passos para a execução de uma fase. Na versão atual do SPARSE estas etapas são criação, revisão e correção. A Figura 8 mostra a divisão em fases de um modelo de processo em cascata implementado na primeira versão do jogo (Souza *et al* 2010).

Figura 8 – Modelo cascata utilizado no SPARSE

- Projeto de Software: é um modelo de processo instanciado com uma equipe de trabalho, restrições de prazo e custo no projeto, onde o objetivo é entregar um produto de software para um cliente. Para o SPARSE o projeto é o elemento principal do jogo, pois por ele passam os comandos e interações presentes no mesmo. Ainda, é este o elemento através do qual é feito o gerenciamento pelo jogador, bem como grande parte da simulação é executada. O projeto define alguns elementos para torná-lo mais realista (Souza *et al* 2010). São eles:
    - Prazo: representa o tempo limite para a finalização do processo de desenvolvimento de software e entrega do produto de software;
    - Recursos monetários: representa a quantidade de gastos que pode ser utilizada para a realização do projeto;

- Qualidade: qualidade do produto de software em desenvolvimento, mensurada com base no número de erros no mesmo.

- Desenvolvedores: os desenvolvedores são os membros da equipe de trabalho aos quais podem ser atribuídas etapas de uma ou mais fases do processo, para que estas possam ser completadas. Os desenvolvedores apresentam algumas características que influenciam na execução do jogo como nível de energia, habilidade em uma determinada fase, experiência, salário e as atividades nas quais este se encontra alocado (Souza *et al* 2010).

- Ferramentas: são software ou técnicas de trabalho que agregam valor ao desenvolvimento de software, podendo diminuir o número de erros produzidos, aumentar a velocidade de execução de algumas atividades, etc. As ferramentas possuem três atributos importantes para a execução do jogo: o custo de aquisição, se a ferramenta se encontra em uso ou não no projeto e a influência da ferramenta em uma determinada atividade (Souza *et al* 2010).

- Regras: as regras do simulador foram definidas sobre um conjunto de boas práticas de Engenharia de Software. Estas regras definem a maneira mais correta que um projeto de software deve ser executado, e podem ser encontradas no trabalho de Navarro (Navarro 2006) (Souza *et al* 2010). Além das regras, existem alguns eventos que podem ser disparados por estas, ou aleatoriamente a fim de tornar a simulação mais próxima do ambiente real. A Figura 9 mostra o relacionamento entre as principais regras para determinar a completude de um projeto ou mesmo de uma fase no sistema (Souza *et al* 2010).

Na versão atual do SPARSE a utilização do jogo se dá através de uma interface tabular contendo todas as informações necessárias para que o jogador possa operar o jogo. Além disso, a interface também fornece um histórico das ações do jogador durante uma execução. A Figura 10 apresenta a interface tabular do jogo (Souza *et al* 2010).

Uma interface com vários recursos gráficos 3D vem sendo desenvolvida para o jogo. A Figura 11 apresenta um resultado parcial desta interface (Souza *et al* 2010).

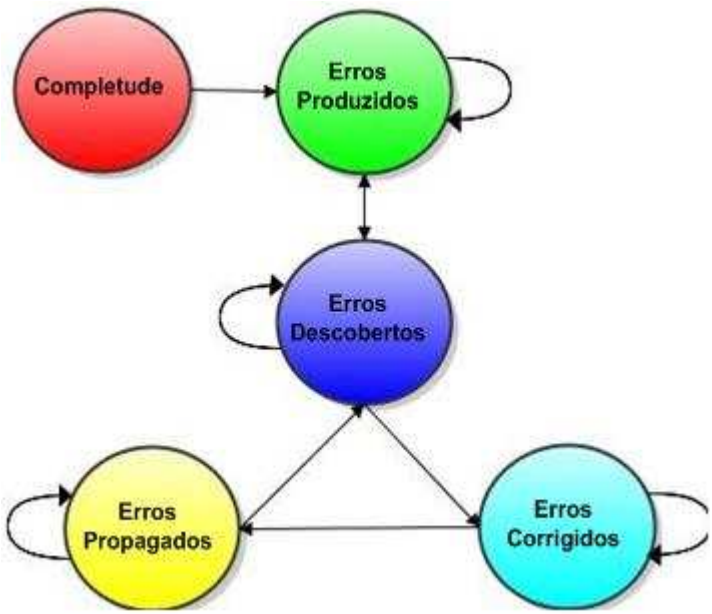

Figura 9 – Regras para determinação da completude de fases/projetos (Souza *et al.* 2010)

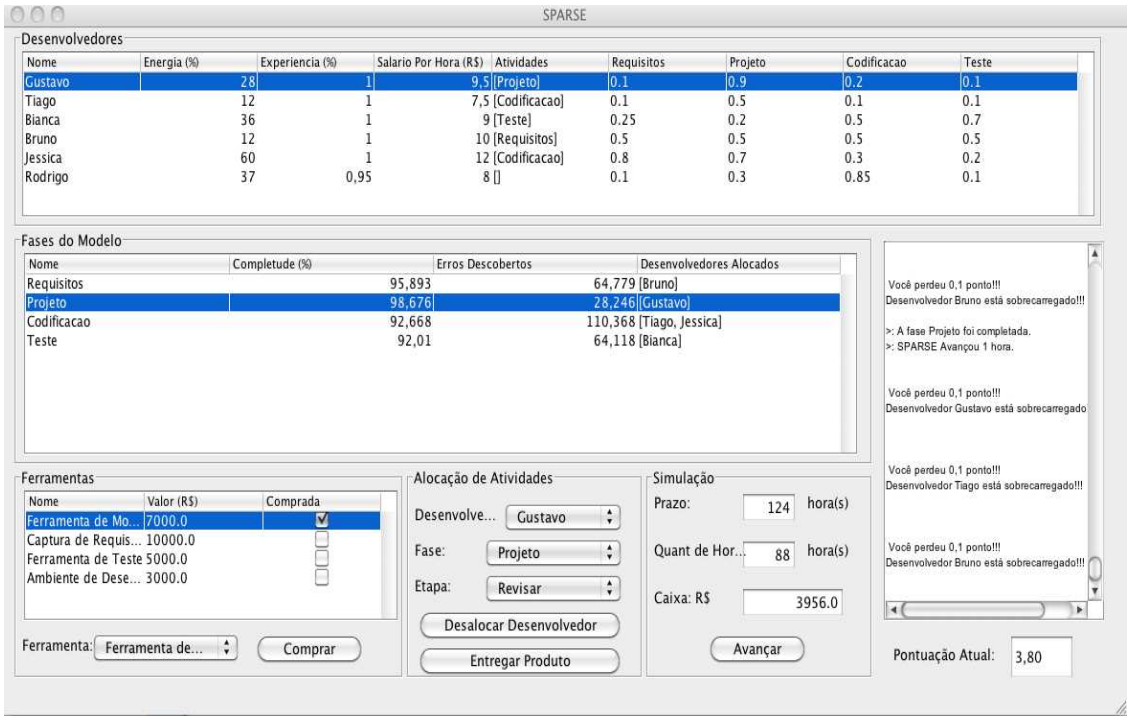

Figura 10 – Interface tabular do SPARSE (Souza *et al* 2010)

Figura 11 – Resultado Parcial da Interface 3D do SPARSE (Souza *et al* 2010)

2.7 OpenUP

O OpenUP é um Processo Unificado que aplica uma abordagem iterativa e incremental dentro de um ciclo de vida estruturado. O OpenUP adota uma filosofia pragmática e ágil que foca na natureza colaborativa do desenvolvimento de software. É um processo objetivo e independente de ferramenta, que pode ser estendido para direcionar uma grande variedade de tipos de projeto (OpenUP 2009).

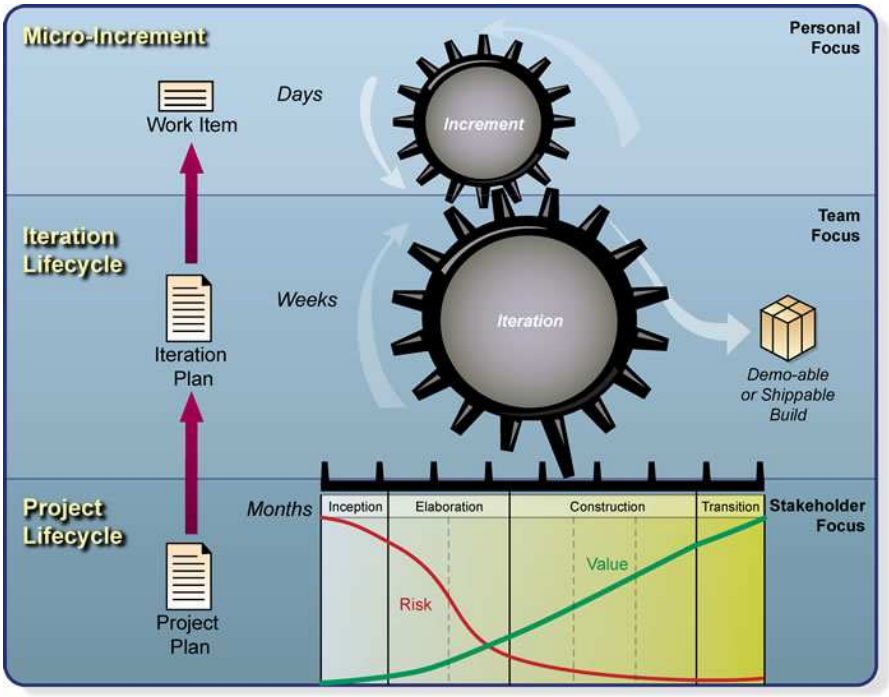

Figura 12 - Camadas do OpenUP (OpenUP 2009)

Ele é um processo considerado mínimo, completo e extensível, valorizando a colaboração entre a equipe e os benefícios aos interessados, ao invés da formalidade e entrega de itens desnecessários (OpenUP 2009). O processo pode ser facilmente entendido através das três camadas microincrementos (*Micro-increment*), ciclo de vida de iteração (Iteration Lifecycle), ciclo de vida do projeto (Project *Lifecycle*), exibidas na Figura 12.

O esforço pessoal em um projeto OpenUP está organizado em microincrementos. Eles representam pequenas unidades de trabalho que produzem um passo do progresso do projeto, constante e mensurável (normalmente medido em horas ou dias). O processo aplica a colaboração intensiva à medida que o sistema é desenvolvido incrementalmente, por uma equipe comprometida e auto-organizada. Estes microincrementos fornecem um ciclo de *feedback* extremamente curto, que direciona decisões adaptativas durante cada iteração (OpenUP 2009).

O OpenUP divide o projeto em iterações planejadas e com intervalos de tempo definidos, normalmente medidos em semanas. As iterações direcionam a equipe na entrega incremental do valor aos *stakeholders* de uma forma previsível. O plano de iteração define o que deve ser entregue durante a iteração, e o resultado é

uma construção demonstrável ou despachável. As equipes OpenUP se auto-organizam para definir como atingir os objetivos da iteração e entregar o resultado. Elas fazem isso definindo e distribuindo tarefas detalhadas de uma lista de itens de trabalho. O OpenUP usa um ciclo de vida de iteração que estrutura como os microincrementos são aplicados para entregar construções estáveis e coesas do sistema, que progridem incrementalmente na direção dos objetivos da iteração (OpenUP 2009).

O OpenUP estrutura o ciclo de vida do projeto em quatro fases: Concepção *(Inception),* Elaboração *(Elaboration),* Construção *(Construction)* e Transição *(Transition).*

- Concepção: primeira fase do processo, onde os *stakeholders* e os membros da equipe colaboram para determinar o escopo e os objetivos do projeto, e determinar se o projeto deve ou não continuar.

- Elaboração: segunda das quatro fases no ciclo de vida do projeto, quando riscos arquiteturais significantes são tratados.

- Construção: terceira fase do processo, a qual foca no detalhamento dos requisitos, no *design*, na implementação e no teste da maior parte do software.

- Transição: quarta fase do processo, a qual foca na transição do software para o ambiente do cliente e na obtenção da concordância dos *stakeholders* de que o desenvolvimento do produto está completo.

Estas quatro fases são mostradas na Figura 13**.** O ciclo de vida de projeto fornece aos *stakeholders* e à equipe de projeto, visibilidade e pontos de decisão durante o projeto. Isto permite uma efetiva supervisão para tomar decisões de "prosseguir ou parar" em momentos apropriados. Um plano de projeto define o ciclo de vida, e o resultado final é uma aplicação passível de ser utilizada (OpenUP 2009).

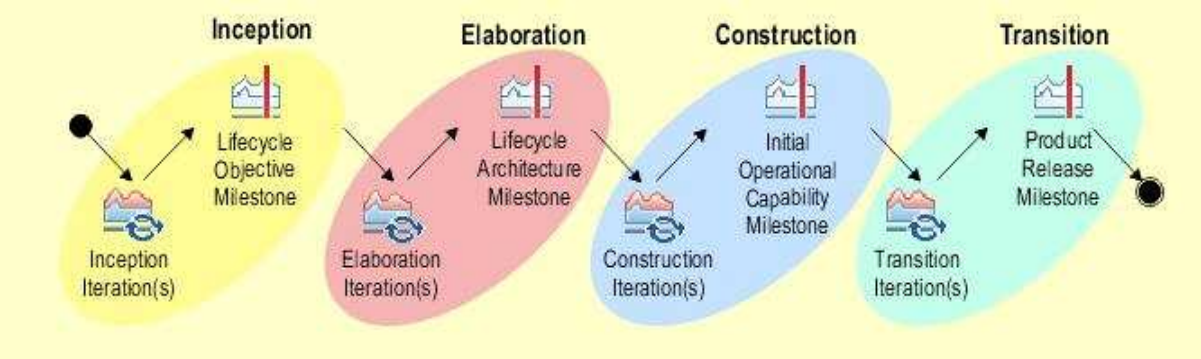

Figura 13 - Ciclo de vida de um projeto OpenUP (OpenUP 2009)

O OpenUP está baseado em quatro princípios fundamentais mutuamente suportados: (OpenUP 2009)

- Equilibrar as prioridades concorrentes para maximizar o benefício aos *Stakeholders*: Promover práticas que permitam aos participantes do projeto e aos *stakeholders* desenvolver uma solução que maximize os benefícios para o *stakeholder*, e que seja compatível com as restrições impostas ao projeto.
- Colaborar para alinhar os interesses e compartilhar o entendimento: Promover práticas que estimulem um ambiente de equipe saudável, que permitam a colaboração e desenvolvam uma compreensão compartilhada do projeto.
- Focar na arquitetura, o mais cedo possível, para reduzir o risco e organizar o desenvolvimento: Promover práticas que permitam à equipe focar na arquitetura para reduzir o risco e organizar o desenvolvimento.
- Evoluir para continuamente obter *feedback* e promover melhorias: Promover práticas que permitam à equipe obter *feedback* dos *stakeholders*, o mais cedo possível e de forma contínua, e demonstrar valor incremental para eles.

Por ser um modelo de processo criado no EPF *Composer*, o OpenUP utiliza a estrutura do meta-modelo SPEM. Neste trabalho o OpenUP será utilizado para a validação do módulo de importação produzido.

3
Módulo de Importação SPEM-SPARSE

Este capítulo apresenta todos os passos para a criação de um módulo para o SPARSE que permita a importação de capability patterns de processos criados no EPF *Composer. Na seção 3.1 as considerações inicias necessárias para contextualizar o módulo são apresentadas. Na seção 3.2 apresenta-se a estrutura de classes criada para a implementação de processos SPEM. Na seção 3.3 estão descritas as modificações necessárias realizadas no SPARSE para que este possa executar processos SPEM. Por sua vez a seção 3.4 apresenta a modelagem e o funcionamento do módulo de importação que integra os processos SPEM ao SPARSE.*

3.1 Considerações Iniciais

Este trabalho tem por finalidade produzir um módulo que possa ser acoplado ao SPARSE possibilitando a importação de processos produzidos no EPF *Composer* para o jogo. Desta forma torna-se possível que qualquer processo produzido utilizando esta ferramenta (processos que sigam o meta-modelo SPEM) possa ser simulado pelo SPARSE. Além disso, este trabalho também propõe mudanças na implementação atual do SPARSE para adaptá-lo à simulação de processos SPEM e inovar a estrutura do simulador.

Neste Capítulo apresentam-se os procedimentos realizados para que o SPARSE possa simular processos SPEM e para que qualquer processo criado via EPF *Composer* possa ser importado para o jogo. Dessa forma as Seções subseqüentes apresentam cada um dos procedimentos realizados para atingir esses objetivos.

3.2 Modelagem da estrutura de processos SPEM

Para tornar possível a criação da nova estrutura do SPARSE, bem como do módulo de importação de processos é necessário em um primeiro momento modelar a estrutura do processo segundo o meta-modelo SPEM.

Através da análise da especificação do SPEM (OMG 2008) e do Capítulo 5 do livro de Paula (2009) modelou-se um conjunto de classes abstratas que serviram de base para a criação da estrutura do processo dentro do SPARSE. A Figura 14 apresenta esta modelagem. Vale ressaltar que a figura apresenta apenas as classes e não os atributos.

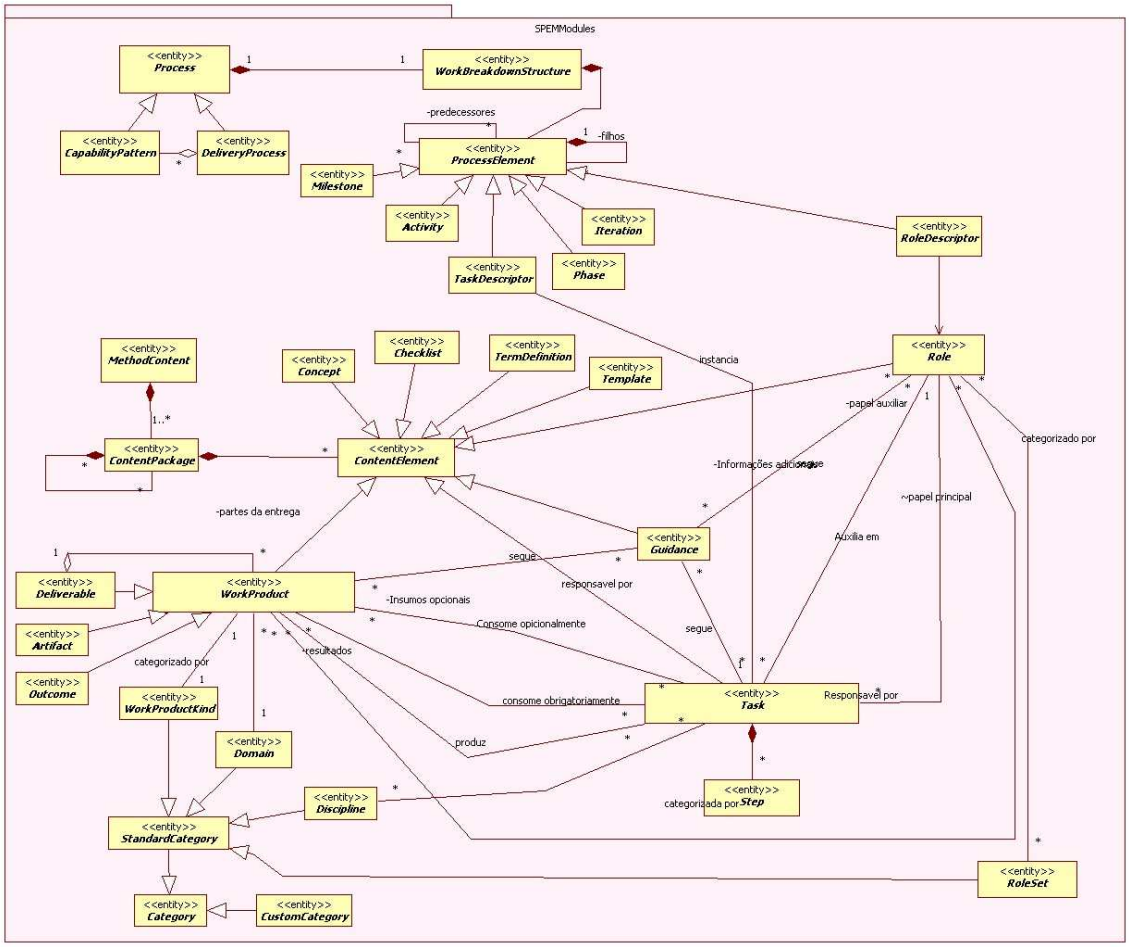

Figura 14 – Modelo de classes abstratas do SPEM

A partir do modelo de classes abstratas do SPEM criou-se as classes para implementação deste no SPARSE. A Figura 15 apresenta o modelo de classes de implementação do SPEM.

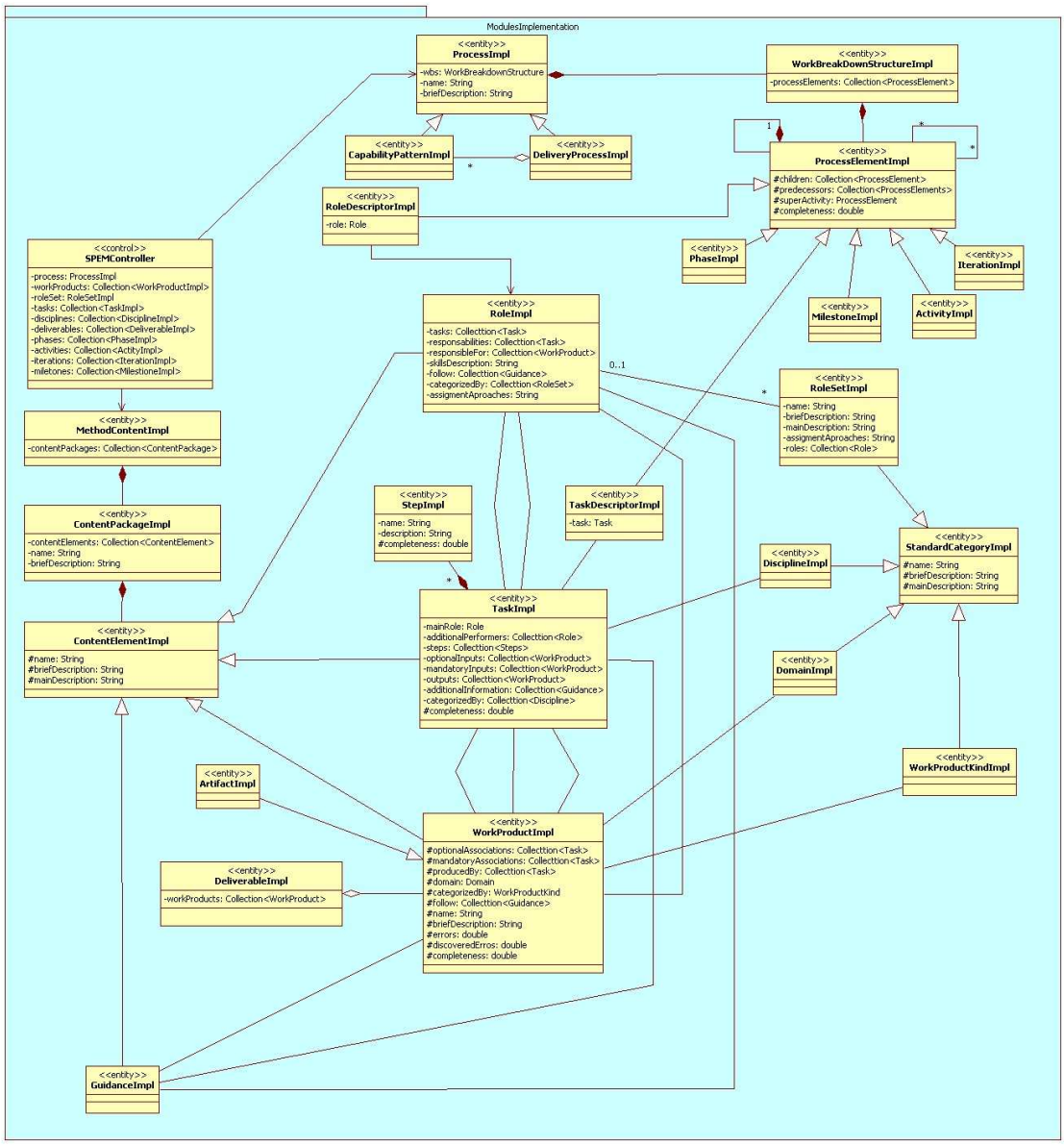

Figura 15 - Modelo de classes de implementação do SPEM no SPARSE

Estas classes formam um pacote que foi inserido numa nova versão do SPARSE, construída como parte deste trabalho em parceria com um projeto de iniciação científica do Laboratório de Engenharia de Software (LEnS) da Universidade Federal de Alfenas (Unifal-MG) e um projeto de doutorado da

Universidade Federal de Minas Gerais (UFMG). Esta nova versão do SPARSE é apresentada na Seção 3.3.

3.3 Adaptações no SPARSE

Para que o SPARSE pudesse suportar a simulação de processos SPEM foram necessárias mudanças na estrutura do software. Num primeiro momento analisou-se a versão antiga do jogo a fim de levantar as características a serem mantidas e quais alterações deveriam ser feitas. A Figura 16 mostra a arquitetura antiga do software.

Na versão anterior do SPARSE a abstração de processo para o simulador foi realizada através da combinação das classes Regra e Fase. A idéia consistia no fato de que, para cada modelo de processo, havia um conjunto de regras separado e um conjunto determinado de fases. Essa implementação de processo - apesar de efetiva quanto ao ensino de processos simples - mostrou-se falha, pois não apresentava aspectos importantes como produtos de trabalho, disciplinas da Engenharia de Software, entre outros. A utilização do SPEM como base para criação de processos a serem simulados resolve estes problemas.

Para a nova versão do SPARSE foi proposta uma melhor divisão entre os conceitos de processo e regras e uma maior capacidade ao simular um processo, melhorando assim o nível de aprendizado de um estudante que utilizá-lo. Na versão anterior era necessário criar uma nova versão do simulador para cada processo. Com a nova versão torna-se possível que qualquer processo, modelado via EPF *Composer*, seja simulado bastando apenas importá-lo através do módulo proposto neste trabalho.

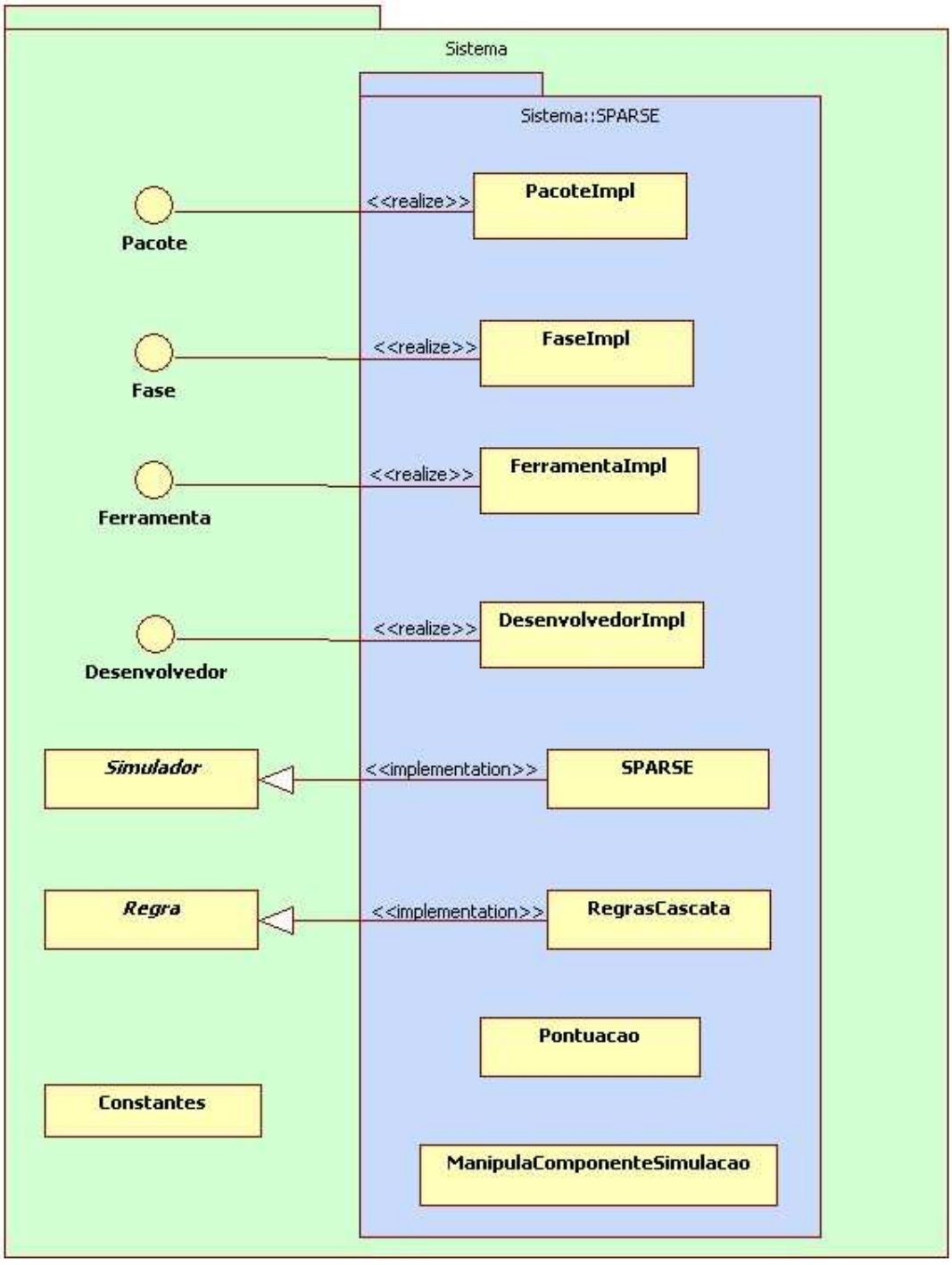

Figura 16 - Arquitetura da versão anterior do SPARSE

A arquitetura proposta para a nova versão do SPARSE é apresentada na Figura 17.

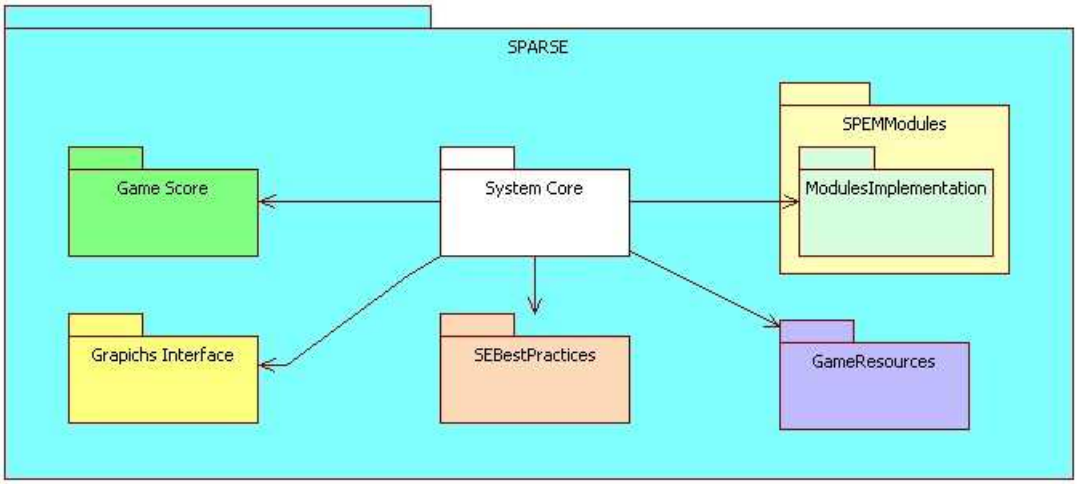

Figura 17 - Arquitetura proposta para a nova versão do SPARSE

O pacote Sistema da antiga arquitetura foi substituído pelo pacote SPARSE na nova e foram feitas subdivisões das capacidades do sistema em novos pacotes. Desta forma, foi possível produzir um software com alto nível de coesão[4] e baixo acoplamento[5]. Cada um desses pacotes é apresentado nas subseções seguintes.

3.3.1 *SPEMModules*

Este pacote contém as classes abstratas do SPEM e o pacote ModulesImplementation que contém as implementações destas classes. Estas classes representam a abstração de processo utilizada pela nova versão do SPARSE.

O pacote SPEMModules armazena apenas as classes abstratas e não armazena nenhuma lógica de negócio. Já o pacote ModulesImplementation armazena toda a lógica de negócio relacionada a processo, a qual o simulador precisa ter acesso. A Seção 3.2 apresenta o conteúdo destes pacotes, que podem ser visualizados na Figura 14 e na Figura 15.

lvilvi

[4] Coesão – medida da proximidade entre todas as responsabilidades, dados e métodos de uma classe (Paula 2009).

[5] Acoplamento – medida de interconexão entre classes e subsistemas (Paula 2009).

3.3.2 *GameResources*

Este pacote contém as classes que implementam a figura do desenvolvedor e ferramenta para a versão anterior do SPARSE e inclui uma figura de cliente não existente na mesma. A Figura 18 apresenta a estrutura do pacote e as classes que o compõe.

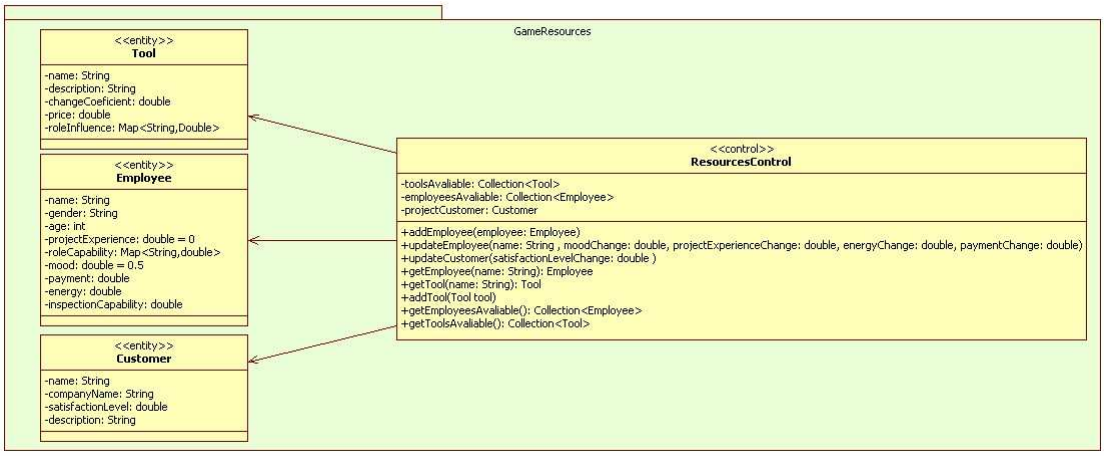

Figura 18 – Estrutura do pacote GameResources

Para as classes *Employee* (equivalente ao desenvolvedor) e a classe *Tool* (equivalente a ferramenta) foi necessária a adição de um atributo que fizesse o mapeamento entre elas e os possíveis *Roles* de um processo SPEM. Desta forma é possível determinar a eficiência de um desenvolvedor ao trabalhar num determinado *Role* e o quanto uma ferramenta pode auxiliar o trabalho relacionado a um *Role*.

Além do mapeamento, na classe *Employee* foi adicionado o atributo *mood* para simular as mudanças de humor, utilizando-o como um parâmetro de influência na qualidade e na velocidade do trabalho realizado pelo mesmo.

A classe *Customer* representa a figura do cliente que solicita um determinado projeto para a empresa que o software simula. Este cliente apresenta um conjunto de atributos estáticos e apenas um atributo dinâmico, o *satisfactionLevel,* que se refere ao nível de satisfação do cliente com o trabalho da empresa e com a qualidade do software em produção.

3.3.3 *SEBestPractices*

O pacote SEBestPractices é responsável por armazenar as informações sobre as boas práticas de Engenharia de Software[6] e retorná-las ao sistema quando for necessário apresentar alguma dessas práticas ao jogador, a fim de aumentar o conhecimento agregado pelo sistema. O pacote é composto por uma classe de entidade (*SEPractices*) responsável por armazenar essas informações e uma classe de controle (*PracticesController*) responsável por fazer a interação desse pacote com o restante do sistema. A estrutura do pacote é apresentada na Figura 19.

O pacote *SEBestPractices* e o pacote *Game Score* apresentado a seguir estão em desenvolvimento como trabalho de conclusão de curso por acadêmicos da Universidade Federal de Alfenas e não serão detalhados neste trabalho.

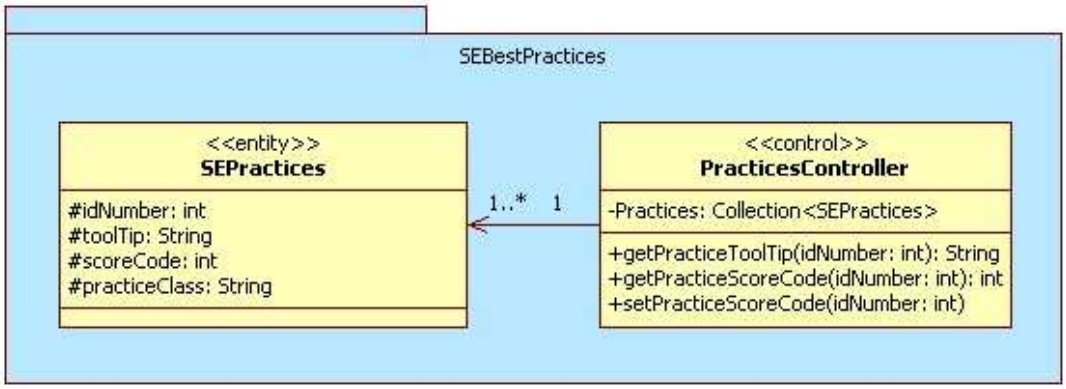

Figura 19 – Estrutura do Pacote SEBestPractices

3.3.4 *GameScore*

O SPARSE utiliza a pontuação do jogo como uma forma de avaliação de desempenho de um jogador durante uma partida. Neste contexto mostrou-se necessária a criação de um pacote específico para tratar desta funcionalidade.

lviiilviii

[6] Boa prática de Engenharia de Software – práticas que quando seguidas auxiliam no alcance do sucesso de um projeto de software. Para mais informações vide (Souza *et al* 2010).

O pacote *GameScore* é responsável por armazenar toda a lógica de negócio referente à pontuação dentro da nova versão do SPARSE. Este pacote é composto por uma classe de entidade e uma classe de controle, *Score* e *ScoreController* respectivamente como apresentado na Figura 20.

Basicamente este pacote tem por função receber possíveis erros do jogador durante o jogo, processá-los e manter a pontuação total. Por exemplo, numa determinada rodada um jogador comete 3 erros, cada um desses erros diminui um pouco a pontuação que este pode ganhar na rodada. É responsabilidade do pacote *GameScore* atualizar a pontuação de acordo com os erros cometidos.

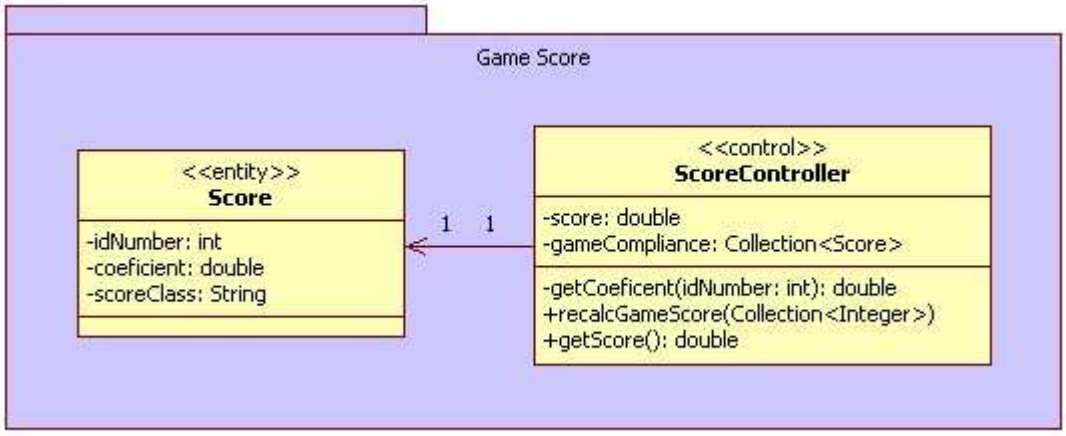

Figura 20 – Estrutura do Pacote *GameScore*

3.3.5 *SystemCore*

Como apresentado no Capítulo 2 a figura de projeto é a mais importante dentro do SPARSE. É através dela que se dá a integração com o jogador e a maioria dos comandos do jogo. O pacote *SystemCore* armazena todas as informações e regras de negócio relativas ao projeto. Além disso, neste pacote apresenta-se o simulador responsável por gerenciar as mudanças nas outras classes.

Este pacote interliga todos os outros pacotes do jogo e controla todo o funcionamento do simulador. Para cada um dos outros pacotes no sistema existe uma classe de controle dentro do *SystemCore* que é responsável por se relacionar com um pacote específico. A Figura 21 apresenta a estrutura do pacote.

Como apresentado na Figura 21 este pacote controla os conceitos de custo, qualidade, prazo e recursos de um projeto. A importância desses conceitos pode ser vista no Capítulo 2, Seção 2.6.

O pacote *GraphicsInterface* utiliza o padrão de projeto *Observer* (Bezerra 2007) com a classe Simulador para a interação com as interfaces gráficas. Este pacote é apresentado a seguir.

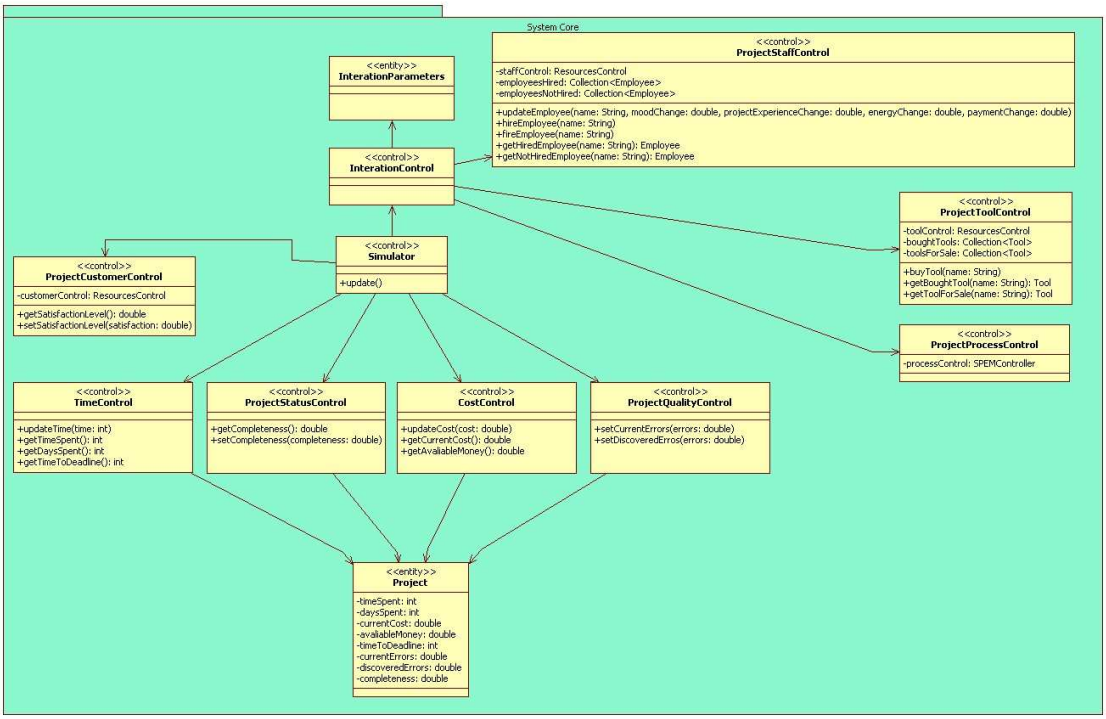

Figura 21 – Estrutura do pacote *SystemCore*

3.3.6 *GraphicsInterface*

O pacote *GrapichsInterface* armazena as interfaces gráficas disponíveis para o SPARSE, e uma interface genérica de acesso. Dessa forma é possível se desenvolver novas intefaces gráficas, desde que essas implementem a interface genérica.

Como dito anteriormente as interfaces gráficas se relacionam com o simulador utilizando o padrão de projeto *Observer*. A Figura 22 apresenta este pacote e a Figura 23 apresenta o relacionamento entre o pacote e o simulador.

Como apresentado no Capítulo 2 o SPARSE possui duas interfaces gráficas, uma delas ainda em desenvolvimento. Ambas as interfaces estão sendo reconstruídas, para que fiquem em conformidade com a nova versão do SPARSE, como projeto de iniciação científica na Unifal-MG.

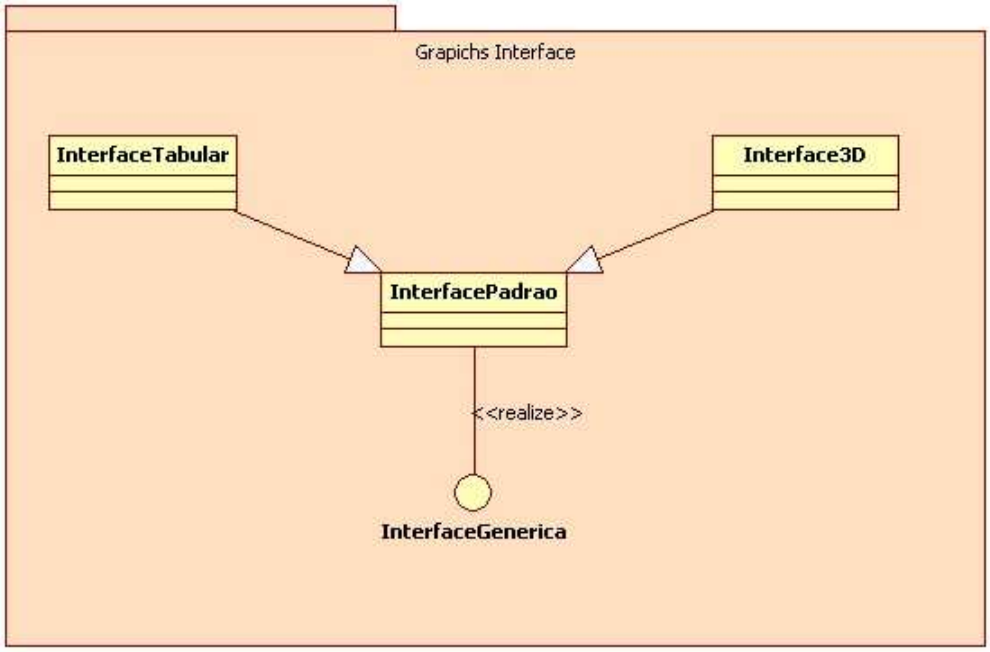

Figura 22 – Estrutura do pacote *GraphicsInterface*

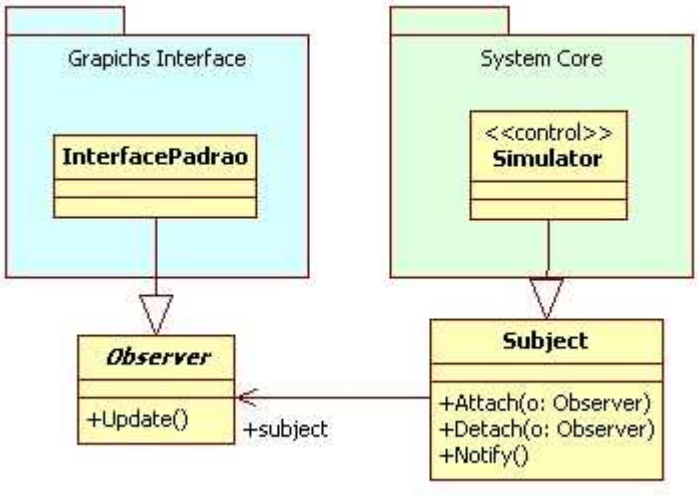

Figura 23 – Padrão de projeto *Observer* utilizado entre os pacotes *SystemCore* e *GraphicsInteface*

3.4 Criação do Módulo de Importação SPEM-SPARSE

O EPF *Composer* permite a exportação de processos criados de diversas formas. Para a criação do módulo de importação para o SPARSE foi utilizada a exportação de *capability patterns*. A exportação produz um arquivo XML com a estrutura mostrada na Figura 24. Pode-se identificar três *tags* mais relevantes:

- <Tasks> – Contém todas as tarefas que compõe um *capability pattern*. Cada uma delas tem um nome, um nível utilizado para identificar a hierarquia entre as tarefas, uma breve descrição e um identificador.
- <Resources> – Contém todos os papéis (*Roles)* necessários para o processo. Cada *tag* <Resource> contém o nome do papel e um número identificador.
- <Assignments> – Esta *tag* é responsável por fazer o relacionamento entre os papéis e as tarefas, identificando para cada tarefa, qual o papel principal responsável por esta e quais os papéis secundários.

```
- <Tasks>
  + <Task>
  + <Task>
  + <Task>
  - <Task>
      <UID>3</UID>
      <ID>3</ID>
      <Name>Implementar os Testes de Desenvolvedor</Name>
      <Type>0</Type>
      <OutlineLevel>2</OutlineLevel>
      <Start>2010-11-19T16:45:04</Start>
      <Milestone>0</Milestone>
      <Notes>Implementar um ou mais testes que permitam a validação dos componentes individuais de software através da execução.</Notes>
    </Task>
  + <Task>
  + <Task>
  + <Task>
  </Tasks>
- <Resources>
  + <Resource>
  + <Resource>
  - <Resource>
      <UID>2</UID>
      <ID>2</ID>
      <Name>Arquiteto</Name>
    </Resource>
  + <Resource>
  + <Resource>
  + <Resource>
  </Resources>
+ <Assignments>
</Project>
```

Figura 24 – Estrutura de um *capability pattern* exportado via EPF *Composer*

O objetivo do módulo de importação é ler um arquivo XML contendo um *capability pattern* e produzir uma estrutura de classe utilizando as classes especificadas para a nova versão do SPARSE. Além disso, o módulo também deve permitir o gerenciamento de desenvolvedores (*Employee*) e ferramentas (*Tool*), bem como permitir a configuração do relacionamento destes com o *capability pattern* exportado. Para atingir tal objetivo foi proposta a estrutura mostrada na Figura 25 para o módulo de importação.

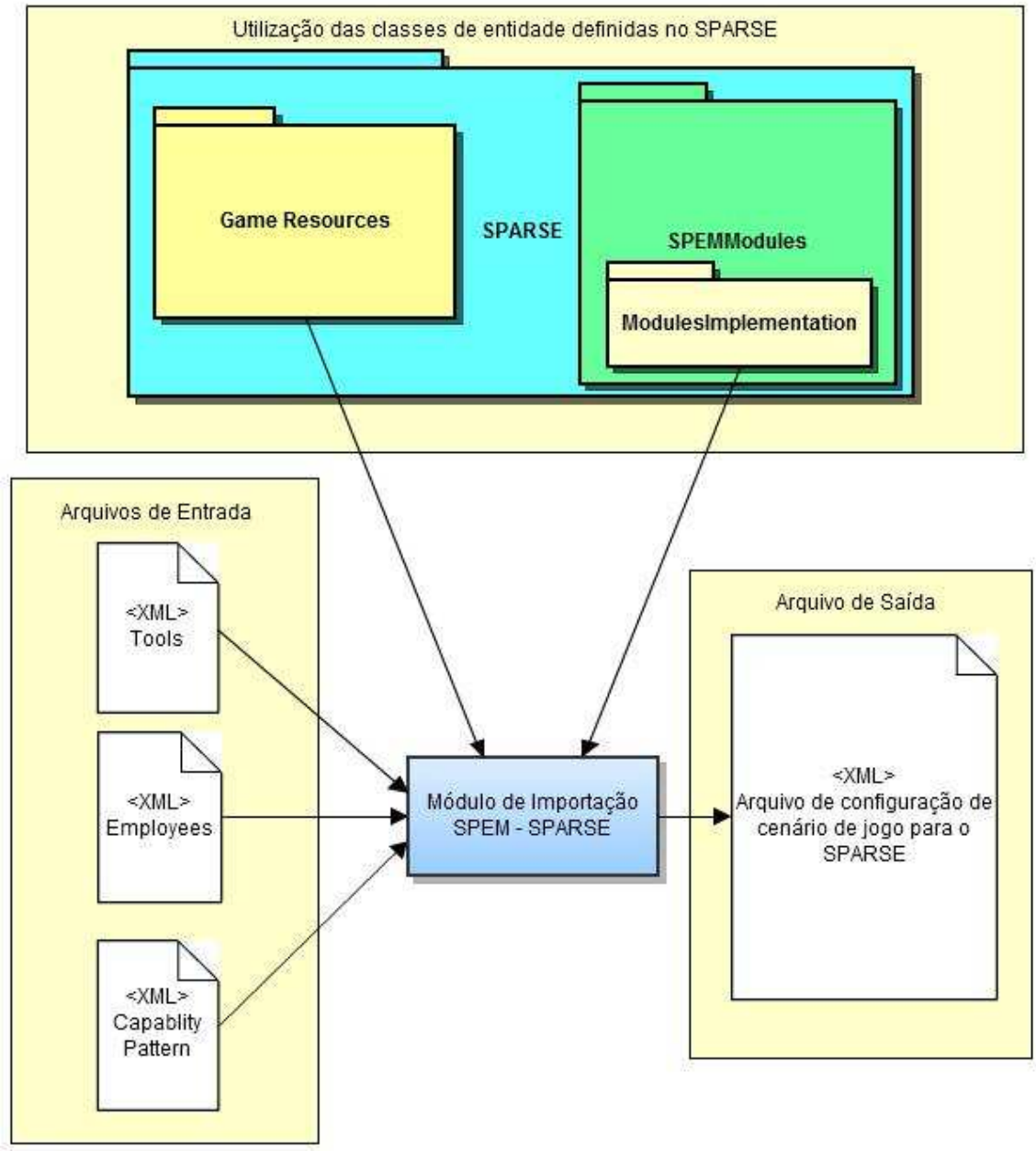

Figura 25 – Representação da estrutura do módulo de importação

Como mostrado na figura, através da leitura de arquivos XML contendo informações iniciais de desenvolvedores (*Employee*) e ferramentas (*Tool*), e um outro arquivo XML contendo um *capability pattern*, exportado via EPF *Composer*, o módulo constrói a estrutura de um processo no contexto de classes do SPARSE. Após a leitura, o módulo permite a edição de algumas informações de desenvolvedores (*Employee*) e ferramentas (*Tool*).

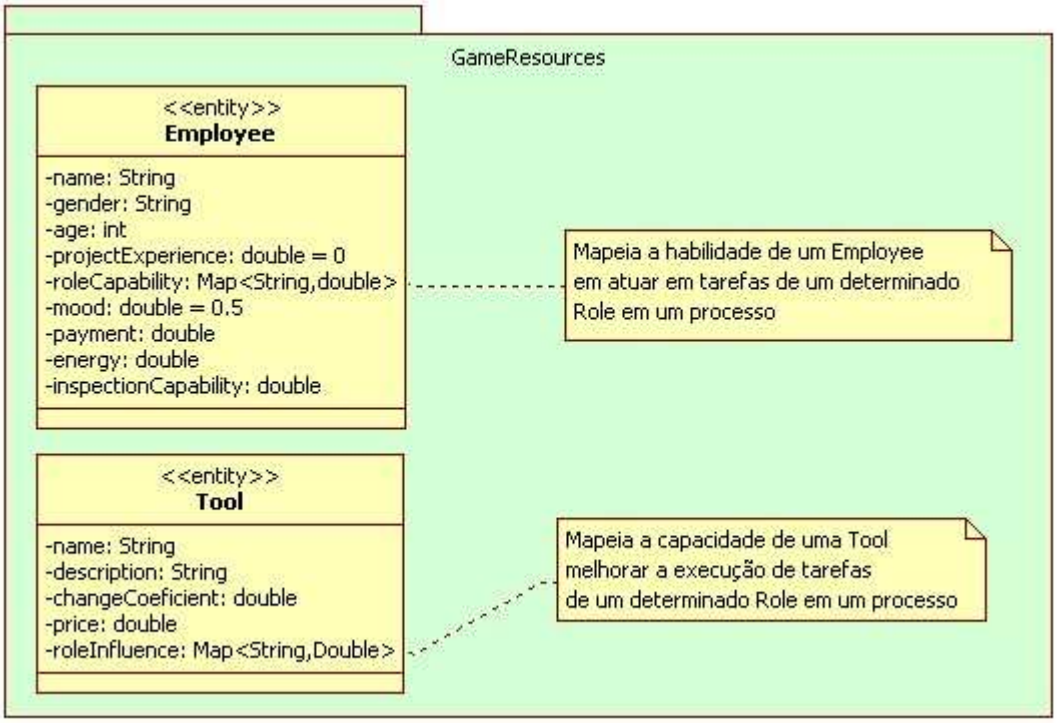

Figura 26 – Classes *Employee* e *Tool* com destaque para os atributos responsáveis por realizar o mapeamento entre estas classes e *roles* de um determinado processo SPEM

Para a classe *Employee* é possível a criação de novos objetos e alteração de atibutos como nome, idade, sexo e, o mais importante, alterar o nível de habilidade tanto em inspeção de tarefas como na realização de atividades relacionadas a um determinado papel no processo. Estas informações são utilizadas pelo SPARSE para realizar a simulação do processo.

Para a classe *Tool* também é possível a criação de novos objetos, alteração de nome, preço, coeficiente de mudança (capacidade de uma ferramenta acelerar tarefas quando estiver em uso). Ainda pode-se editar a relação entre uma ferramenta e um *role*, configurando a capacidade da ferramenta em questão ao auxiliar tarefas executas por um determinado *role*. A Figura 26 mostra a estrutura das classes *Tool* e *Employee* bem como identifica os atributos que realizam os mapeamentos supracitados.

Com todas as características mencionadas anteriormente, o módulo de importação permite, a um docente, criar um cenário de jogo de acordo com um *capability pattern* previamente selecionado, configurando uma equipe de desenvolvedores e conjunto de ferramentas. Dessa forma, para cada *capability pattern* exportado via EPF *Composer* e lido pelo módulo de importação, é possível a criação de vários cenários de jogo diferentes.

Após a especificação do cenário, o módulo permite sua exportação para um arquivo XML que pode ser lido diretamente pelo SPARSE. A Figura 27 mostra, resumidamente, um exemplo de arquivo de configuração de um cenário para o SPARSE .

Figura 27 – Estrutura resumida de um arquivo de saída produzido pelo módulo de importação

Para a validação deste módulo realizou-se um experimento, que será apresentado no próximo Capítulo.

4
Validação do módulo de importação SPEM-SPARSE

Este capítulo apresenta os resultados relacionados ao módulo de importação produzido neste trabalho. Ainda neste capítulo é apresentada a validação destes resultados através da importação de um capability pattern do OpenUP para o jogo SPARSE.

4.1 Considerações Iniciais

O módulo de importação SPEM-SPARSE foi criado no intuito de possibilitar a execução de *capability patterns* de processos SPEM no SPARSE. Para validar o módulo de importação produzido neste trabalho foram seguidos os seguintes passos: Seleção e exportação de um *capability pattern* do OpenUP através do EPF *Composer*; Importação deste *capability pattern* para o módulo de importação SPEM-SPARSE; Configuração de um conjunto de desenvolvedores e ferramentas para a criação de um cenário de jogo; Exportação do resultado da configuração anterior para o arquivo XML que servirá de entrada para o SPARSE; Importação do arquivo de configuração criado para o SPARSE. As Seções seguintes explicam cada um desses passos.

4.2 Seleção e exportação de um *capability pattern* do OpenUP

O primeiro passso para a validação deste trabalho foi a seleção de um dos *capability patterns* do OpenUP para ser exportado pelo EPF *Composer*. A Figura 28 apresenta a interface de exportação de *capability patterns* do EPF *Composer*.

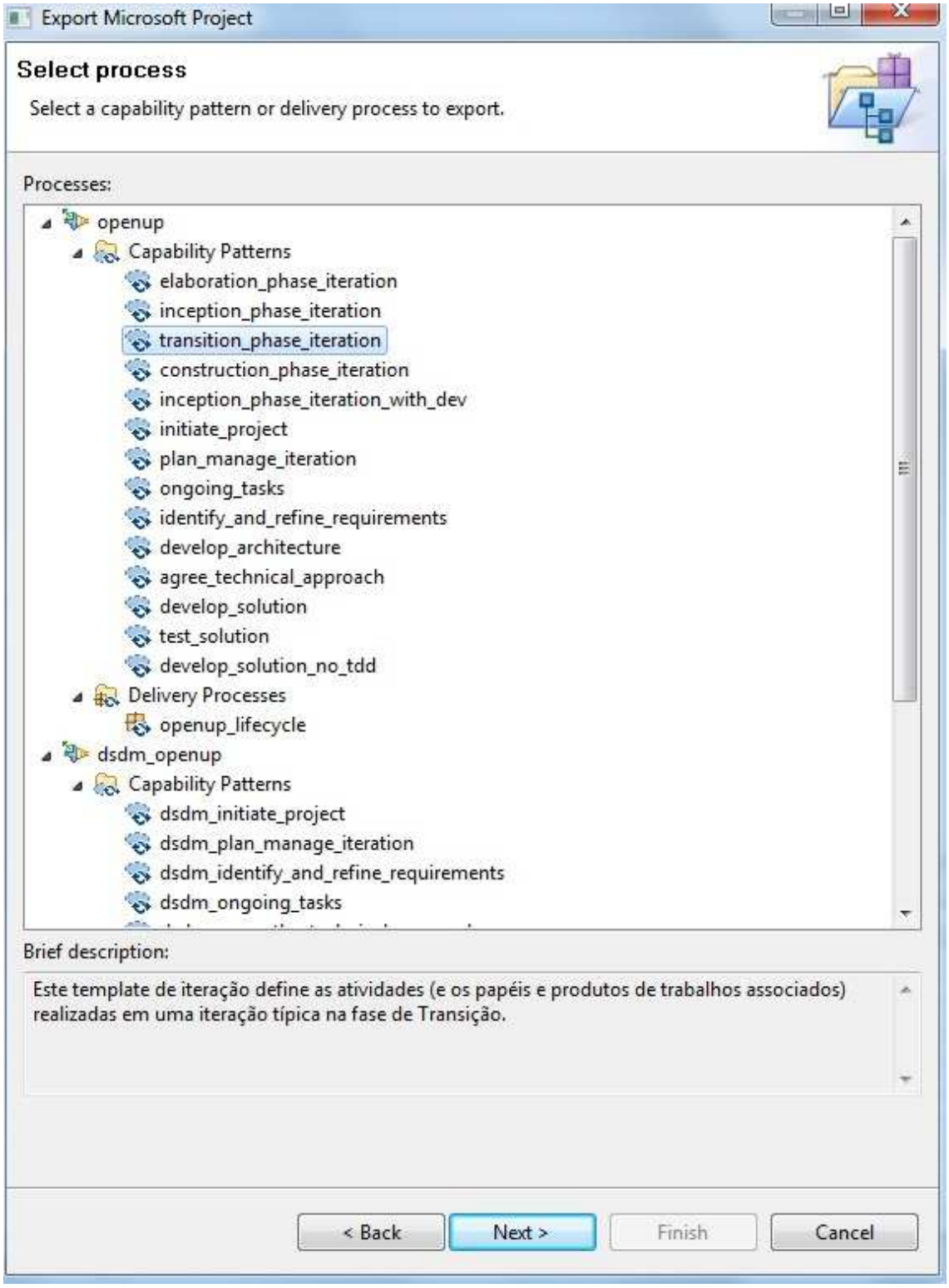

Figura 28 – Interface de exportação de *capability patterns* do EPF *Composer*

Selecionou-se aleatoriamente o *capability pattern "develop_solution"* para realizar a validação do módulo. A estrutura do arquivo exportado foi apresentada na Figura 24 e explicada na Seção 3.4.

4.3 Importação do *capability pattern* para o módulo de importação SPEM-SPARSE

Utilizando o módulo importou-se o *capability pattern* previamente exportado, obtendo-se como resultado o processo instanciado no contexto de classes do SPARSE. A Figura 29 apresenta o resultado dessa importação.

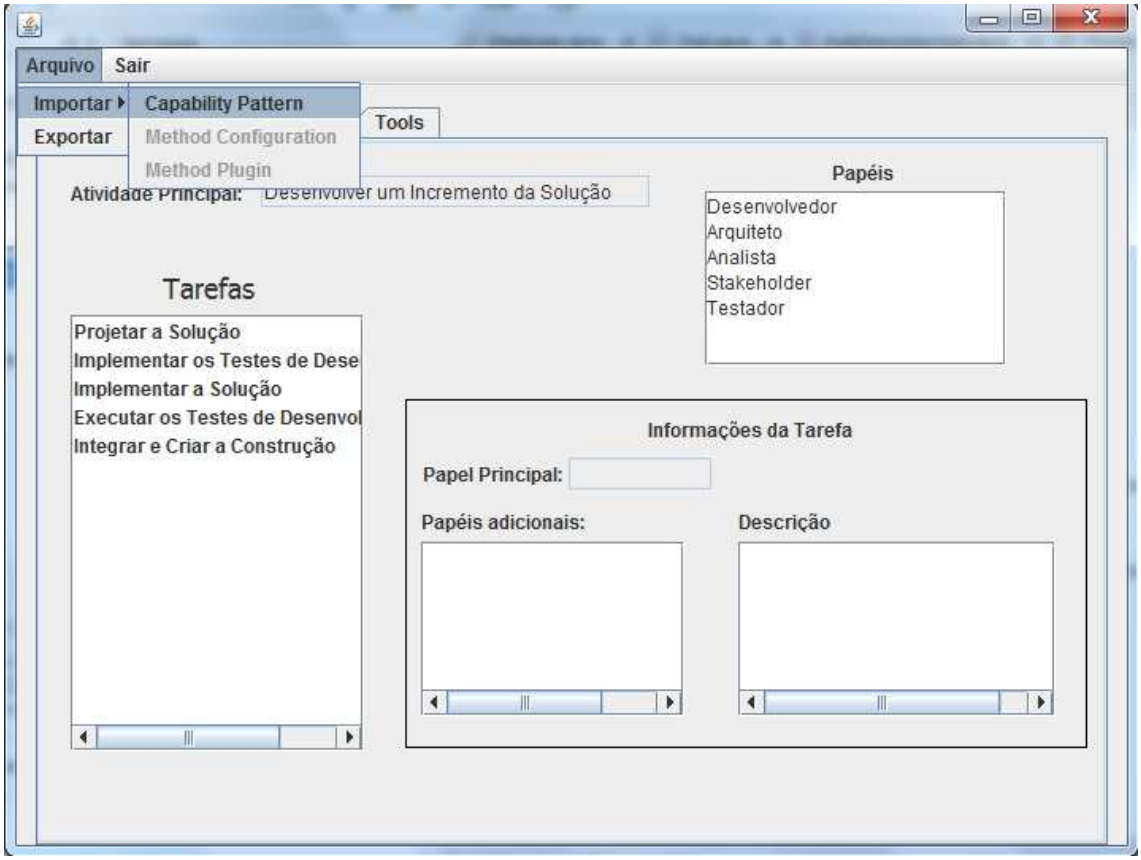

Figura 29 – Resultado da importação do *capability pattern "develop_solution"*

Como mostrado na Figura 29, após a importação o módulo apresenta todos os papéis (*roles*) e todas as tarefas (*tasks*) pertencentes ao *capability pattern*. Ao selecionar uma das tarefas é possível visualizar qual o papel principal responsável

pela tarefa, uma descrição resumida da mesma e quais os papéis auxiliares para a realização desta.

4.4 Configuração do cenário de jogo

Para a criação do arquivo de configuração de cenário para o SPARSE foi criado um conjunto de desenvolvedores (*Employees*) e um conjunto de ferramentas (*Tools*). As operações possíveis para criação e configuração destes conjuntos foram descritas na Seção 3.4. A Figura 30 apresenta a configuração dos desenvolvedores e a Figura 31 mostra a configuração de ferramentas escolhida para a validação do módulo de importação SPEM-SPARSE.

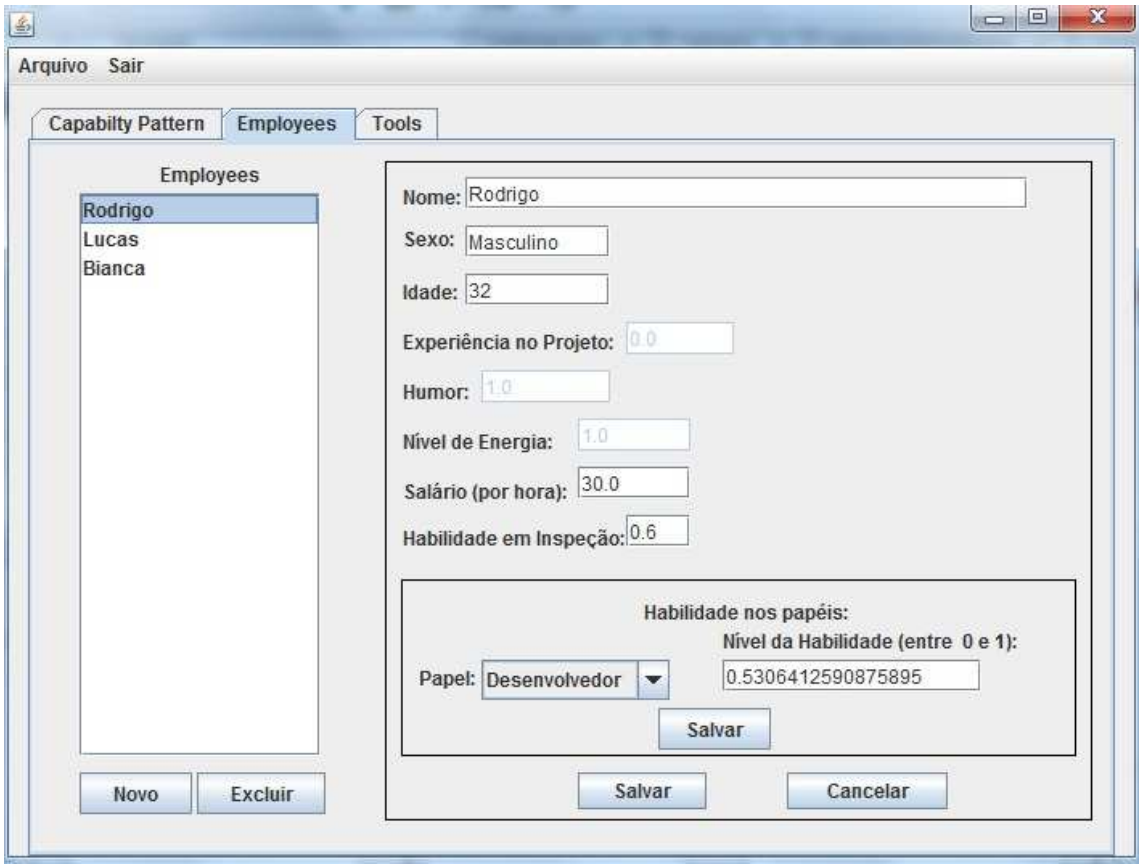

Figura 30 – Conjunto de desenvolvedores (*Employees*) configurado para a validação do módulo de importação.

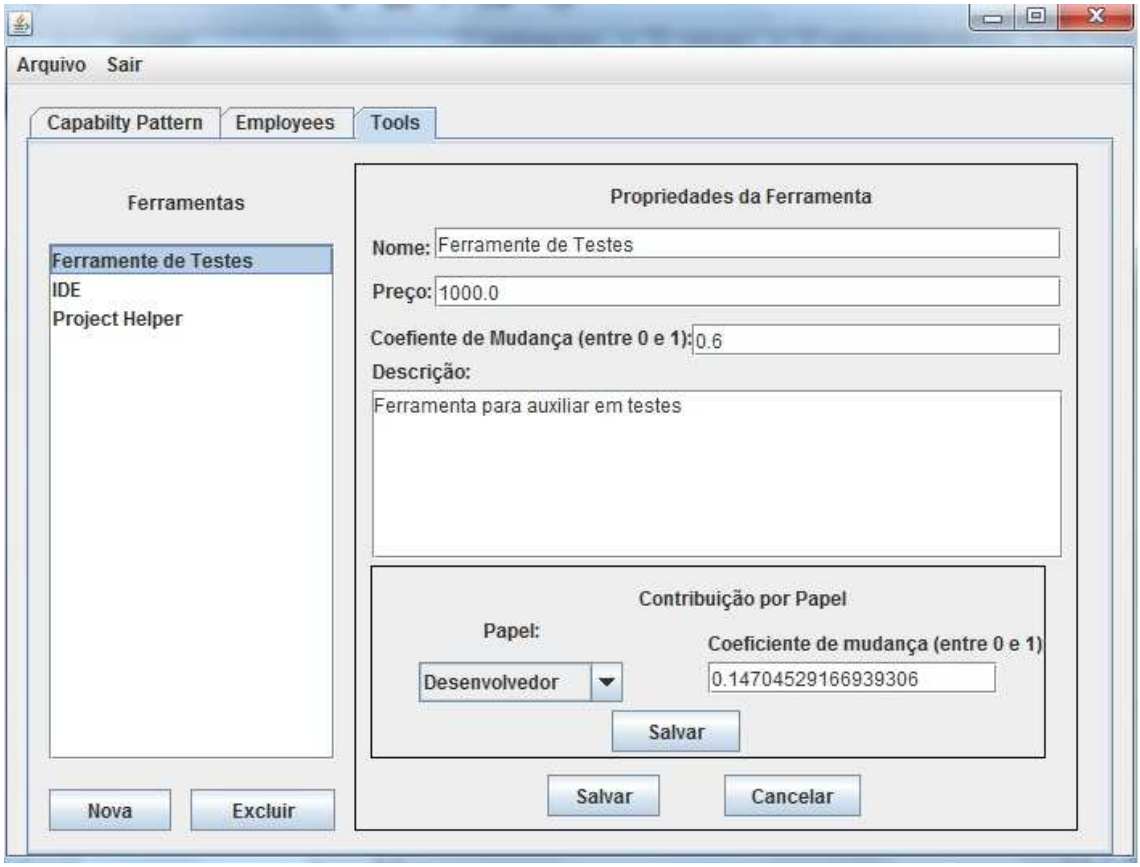

Figura 31 – Conjunto de ferramentas (*Tools*) configurado para a validação do módulo de importação

4.5 Exportação do cenário de jogo

Através da configuração dos conjuntos de desenvolvedores e ferramentas, o *capability pattern* foi configurado para funcionar como um cenário de jogo válido para o SPARSE. Em seguida, o cenário de jogo foi exportado para o arquivo de saída do módulo de importação, arquivo este que pode ser lido pelo leitor XML interno do SPARSE diretamente.

A estrutura básica do arquivo de saída é apresentada na Figura 27. Cada umas das *tags* apresentadas nesta figura é descrita e ilustrada a seguir.

A *tag* <Employees> armazena todos os desenvolvedores, com uma *tag* <Employee> para cada desenvolvedor do cenário. A Figura 32 apresenta a

estrutura desta *tag*. Os atributos presentes na estrutura da *tag* condizem com os definidos para a classe *Employee* na Seção 3.4.

Figura 32 – Estrutura da *tag* <Employees> no arquivo de configuração de cenário

A *tag* <Tools> armazena o conjunto de ferramentas para o cenário, com uma *tag* <Tool> para cada ferramenta. A estrutura da *tag* <Tool> é condizente com a classe *Tool* definida na Seção 3.4. A Figura 33 apresenta em detalhes a estrutura da *tag* <Tools>.

Figura 33 – Estrutura das *tags* <Tools> e <Tool> no arquivo de configuração de cenário do SPARSE.

Além dos conjuntos de desenvolvedores e ferramentas, o arquivo de configuração de cenário de jogo contém a estrutura do *capability pattern* armazenada no contexto de classes do SPARSE. A *tag* <CapabilityPattern> apresenta 3 tags internas mais relevantes: <Roles>, <Activities> e <TaskDescriptors>.

A *tag* <Roles> contém todas os papéis do *capability pattern*, e um identificador para cada um deles. Este identificador é usado no relacionamento dos papéis com as tarefas do *capability pattern*.

A *tag* <Activities> armazena a informação em relação à atividade principal do *capability pattern.* Esta atividade se subdivide nas tarefas do *capability pattern.* A Figura 34 apresenta a estrutura das *tags* <Roles> e <Activities>.

Figura 34 – Arquivo de configuração de cenário de jogo estrutura das *tags* <CapabilityPattern>, <Roles> e <Activities> para o *capability pattern* "*develop_solution*".

A *tag* <TaskDescriptor> contém um conjunto de *tags* <TaskDescriptor>, que armazenam uma tarefa (*Task*) do *capability pattern.* Cada <Task> possui um nome, uma descrição simples, um papel principal (que utiliza o número identificador das *tags* <Role>) e um conjunto de papéis que podem auxiliar na execução da tarefa. A Figura 35 apresenta a estrutura das *tags* <TaskDescriptor> e <Tasks>.

```
- <TasksDescriptor>
  - <Task>
      <Name>Implementar a Solução</Name>
      <BriefDescription>Implementar o código fonte </BriefDescription>
      <MainRole>1</MainRole>
    - <AdditionalPerformers>
      - <AdditionalPerformer>
          <RoleID>4</RoleID>
        </AdditionalPerformer>
      - <AdditionalPerformer>
          <RoleID>5</RoleID>
        </AdditionalPerformer>
      </AdditionalPerformers>
    </Task>
  </TasksDescriptor>
```

Figura 35 – Arquivo de configuração de cenário de jogo. Estrutura da *tag* <Task> para o *capability pattern* "*develop_solution*".

4.6 Considerações Finais

Após a produção do arquivo de configuração de cenário de jogo para o SPARSE foi possível importar o *capability pattern* e os conjuntos de empregados e ferramentas diretamente para a nova versão do SPARSE. Devido a nova versão do SPARSE estar em desenvolvimento, ainda não foi possível realizar a simulação do arquivo criado. Porém, o leitor XML do SPARSE, numa instância criada para teste, carregou o arquivo com sucesso.

5
Conclusões

Este capítulo apresenta as contribuições dessa monografia de maneira clara e concisa e algumas sugestões de como permitir a evolução deste trabalho de forma a torná-lo cada vez mais abrangente

5.1 Considerações finais

Para a resolução do problema do ensino de Engenharia de Software o SPARSE pode ser considerado uma proposta pertinente. Com os resultados produzidos por este trabalho, o SPARSE torna-se uma ferramenta mais abrangente no tocante à simulação de diferentes modelos de processo de software.

Através do módulo de importação SPEM-SPARSE produzido neste trabalho torna-se possível que qualquer *capability pattern* de um processo criado no EPF *Composer* possa ser simulado dentro do jogo SPARSE. Desta maneira, um processo específico de uma instituição, ou de uma empresa pode ser ensinado de maneira mais completa a alunos de graduação (desde que o processo tenha sido modelado utilizando o EPF *Composer*).

Utilizando o módulo de importação SPEM-SPARSE e a nova versão do jogo, desenvolvida em paralelo a este trabalho, aumenta-se as possibilidades de se atingir um novo nível de aprendizado, através da simulação de processos em um maior nível de detalhes, em contextos especificados por um docente ministrante de uma das disciplinas da área de Engenharia de software.

Além disso, é possível estender as funcionalidades do módulo de importação adicionando mais tipos de importação suportados pelo EPF *Composer*. Algumas destas adições são propostas na Seção 5.2.

5.2 Trabalhos futuros

Além da exportação de *capability patterns* o EPF *Composer* permite a exportação de processos em outras configurações como, por exemplo, *method configuration* e *method plugin*.

Para aprimorar este trabalho e torná-lo ainda mais abrangente podem ser adicionadas as importações para estes tipos de exportação do EPF *Composer*. Desta forma torna-se possível o aumento gradativo de informações do processo contidas num cenário de jogo do SPARSE.

Para validação do processo simulado dentro do SPARSE é necessário que o simulador seja desenvolvido obedecendo a arquitetura mostrada neste trabalho. Portanto, além do aumento do número de informações de processo é necessário que se construa o suporte as informações sobre processo já exportadas e as que serão provenientes dos novos tipos de importação de processo. Para realizar esta atividade propõe-se a construção do simulador da nova versão do SPARSE como uma continuidade necessária para o aprimoramento deste trabalho.

6 Referências Bibliográficas

Baker, A; Navarro, E; Hoek, A. *An experimental card game for teaching software engineering process*, The Journal of System and Sofwate, Elsevier, 2005.

Barros, M. (2001) "Gerenciamento de Projetos Baseados em Cenários". Tese de PhD, UFRJ, Rio de Janeiro, Brasil.

Bezerra, E. *Princípios de Análise e Projeto de Sistemas com UML*. Elsevier, 2007.

Drappa, A; Ludewig, J. *Simulation in Software Engineering Training*, ICSE 2000, ACM, 2000.

Haumer, P, *Overview to Eclipse Framework Composer,* Eclipse Foundation, 2007.

Henderson-Sellers, B., Gonzalez-Perez, C., *A comparsion of four process metamodels and the creation of a new generic standard*, Univesity of Technology, Sidney, Australia. Publicado em: Information and Software Technology nº 47, , páginas 49-65, 2005.

Moro, M; Braganholo, V; Dorneles, C; Duarte, D; Galante,R; Mello, R., *XML: Some Papers in a Haystack,* SIGMOD Record, 2009.

Nardini, E., Omincini, A., Denit, E., Molesini, A., *SPEM on test: the SODA case study,* SAC'08, páginas 16-20, Março, 2008, Fortaleza, Ceará, Brasil.

Navarro, E. *SimSE: A Software Engineering Simulation Environment for Software Process Education* , Universidade da Califórnia, Irvine, 2006.

OMG, *Software & System Process Engineering Metamodel Specification*, Versão 2.0, 2008.

OMG, *OMG Unified Modeling Language (OMG UML),* Infrastructure,Versão 2.3, OMG, Maio, 2010.

OpenUP, *"Open Unified Process". Versão 1.5.0.4. 10-08* 2009, disponível em: http://epf.eclipse.org/wikis/openup/ data de acesso: 23/11/2010.

Paula Filho, W. *Engenharia de Software, métodos e padrões.* LTC, 2009.

Pfleeger, S. *Engenharia de Software: Teoria e Prática*, Prentice Hall, 2004.

Pressman, R. *Engenharia de Software.* Pearson Makron Books, 1995.

Ramsin, R., Paige, F. *Process-Centered Review of Object Oriented Software Development Methods*, ACM Computing Survey, Vol 40, N 1, artigo 3, 2008.

Royce, W. W. (1970). *"Managing the development of large software systems: concepts and techniques"*. Proc. IEEE WESTCON, Los Angeles CA: IEEE Computer Society Press. (Cap. 4).

Sommerville, I, *Engenharia de Software*, Oitava Edição, Pearson Addison-Wesley, 2007.

Souza, M; Resende, R; Rodrigues, L; Rodrigues. A; Carvalho, F; Franco Junior, E; *SPARSE: Um Ambiente de Ensino e Aprendizado de Engenharia de Software Baseado em Jogos e Simulação*, SBIE, 2010.

W3C Recommentadion, *Extensibible Markup Language (XML) 1.0 (Fifth Edition)* W3C, 2008, disponível em: http://www.w3.org/TR/2008/REC-xml-20081126/ acessado em 16/11/2010.