Pontifícia Universidade Católica do Rio Grande do Sul
Faculdade de Informática
Programa de Pós-Graduação em Ciência da Computação

Autoria de Artefatos de Software

Marcos Tadeu Silva

Dissertação de Mestrado

Porto Alegre
2010

Pontifícia Universidade Católica do Rio Grande do Sul
Faculdade de Informática
Programa de Pós-Graduação em Ciência da Computação

Autoria de Artefatos de Software

Marcos Tadeu Silva

Dissertação apresentada como requisito parcial à obtenção do grau de mestre em Ciência da Computação

Orientador: Prof. Dr. Ricardo Melo Bastos

Porto Alegre
2010

Dados Internacionais de Catalogação na Publicação (CIP)

S586a Silva, Marcos
Autoria de artefatos de software / Marcos Silva.
Porto Alegre, 2008.
231 f.

Diss. (Mestrado) – Fac. de Informática, PUCRS.
Orientador: Prof. Dr. Ricardo Melo Bastos

1. Informática. 2. Software – Desenvolvimento.
I. Bastos, Ricardo Melo. II. Título.

CDD 005.1

Ficha Catalográfica elaborada pelo
Setor de Tratamento da Informação da BC-PUCRS

Pontifícia Universidade Católica do Rio Grande do Sul
FACULDADE DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

TERMO DE APRESENTAÇÃO DE DISSERTAÇÃO DE MESTRADO

Dissertação intitulada **"Autoria de Artefatos de Software"**, apresentada por Marcos Tadeu Silva, como parte dos requisitos para obtenção do grau de Mestre em Ciência da Computação, Sistemas de Informação, aprovada em 23/12/08 pela Comissão Examinadora:

Prof. Dr. Ricardo Melo Bastos – Orientador — PPGCC/PUCRS

Profa. Dra. Ana Paula Terra Bacelo - — PPGCC/PUCRS

Prof. Dr. Toacy Cavalcante de Oliveira – — University of Waterloo

Homologada em..02/03../2010, conforme Ata No. 03/10. pela Comissão Coordenadora.

Prof. Dr. Fernando Gehm Moraes
Coordenador.

PUCRS

Campus Central
Av. Ipiranga, 6681 – P32 – sala 507 – CEP: 90619-900
Fone: (51) 3320-3611 – Fax (51) 3320-3621
E-mail: ppgcc@pucrs.br
www.pucrs.br/facin/pos

Dedico este trabalho a minha família, por tudo que fomos, somos e seremos.

Agradecimentos

Agradeço...
À minha mãe, pela pessoa super especial que ela é.
Ao meu pai, por tudo que fez por mim.
À minha famíla, pela união leal e inabalável que possuímos.
Ao meu grande amigo Carlos Bragança, por toda a paciência e incentivo.
Aos meus amigos do CPES, pela paciência e incentivo.
Ao Prof. Dr. Toacy Oliveira, pela amizade e pelo incomensurável apoio técnico, como meu co-orientador.
Ao meu orientador, Prof. Dr. Ricardo Bastos.
Agradeço a AGT/PUC-RS e ao convênio PUC-RS/DELL, pelo apoio financeiro.
À todos aqueles que me ajudaram no decorrer da minha jornada.

Resumo

No contexto da engenharia de software, processos de desenvolvimento de software definem um conjunto de “atividades”, “papéis”, e “artefatos” que são utilizados durante o ciclo de vida da construção de um produto de software. Entretanto, tais processos representam artefatos como documentos monolíticos, que são compostos de informações fracamente estruturadas. Isto ocorre dado o fato da construção dos artefatos ser feita a partir de processadores de textos, levando a um problema na computação da estrutura, da organização e do conteúdo, uma vez que limita a potencial formalização dos artefatos utilizados. Diante deste contexto, o desafio é construir artefatos de software que podem ser controlados e gerenciados através de ferramentas que trabalhem com maior nível de formalização. Neste sentido, apresentamos uma abordagem que visa à representação de artefatos de forma bem estruturada, separando artefatos em dois níveis: “estrutura” e “conteúdo”. Além disso, este trabalho também auxilia na utilização de artefatos de forma genérica, independente de processo, o que é conhecido na literatura por camadas de “definição” de artefatos e “uso” em processos. Desta forma, nossa abordagem consiste na utilização de um paradigma diferente de manipulação de artefatos, numa tentativa de melhoria no controle da informação desde a fase de autoria e definição do processo de desenvolvimento de software.

Palavras-chave: Artefatos de Software, Autoria, Processos de Desenvolvimento de Software, Metamodelo, SPEM v2, UML.

Abstract

In the context of software engineering, software process defines a set of “activities, “roles”, and “artifacts” that are used throughout a software product life-cycle. However, these processes represent artifacts as monolithic documents of loosely structured information. This occurs due to the fact that the artifact construction is developed through text processing tools, leading to problems in the computation of the structure, in the organization, and in the content, since it limits the potential formalization of the used artifacts. In this context, the challenge is to build software artifacts through tools that work with a higher level of formalization. In this sense, we present an approach which aims to represent software artifacts in a structured way, slicing artifacts in two levels: “structure” and “content”. Besides, this work also helps to present artifacts as a generic process independent source. In the literature, this is known as separated “method definition” and “process structure use”. Therefore, our approach consists in using a different paradigm for artifact manipulation, trying to increase the information control at the software development process authoring and definition phase.

Keywords: Software Artifact, Authoring, Software Development Process, Metamodel, SPEM v2, UML.

Lista de Figuras

Lista de Tabelas

Lista de Siglas

PDS	Processo de Desenvolvimento de Software
PS	Produto de Software
AS	Artefato de Software
RUP	Rational Unified Process
OPEN	Object-oriented Process, Environment and Notation
EMF	Eclipse Modeling Framework
UC	Caso de Uso
ES	Engenharia de Software
RUP	Rational Unified Process
OMG	Object Management Group
MOF	MetaObject Facility
UML	Unified Modeling Language
MDA	Model Driven Architecture
EMOF	Essential MetaObject Facility
CMOF	Complete MetaObject Facility
L0	Level 0 (UML Infra-Structure)
LM	Metamodel Constructs (UML Infrastructure)
SPEM v2	Sotware and System Process Engineering Metamodel
XSD	XML Schema Definition
ER	Entidade Relacionamento
CDAM	Compound Document access and Management
URI	Uniform Resource Identifier
OCL	Object Constraint Language
OOXML	Office Open XML
WBS	Work Breakdown Structures
SwAT	Software Artifact Specification Tool
MDD	Model Driven Development
SWT	Standard Widget Toolkit

RCP	Rich Client Platform
GUI	Graphical User Interfaces

Sumário

1 Introdução

O Processo de Desenvolvimento de Software (PDS) caracteriza-se pela descoberta e documentação das informações sobre o domínio do sistema em construção. Geralmente isto é feito em um ciclo tipicamente incremental, de modo a sincronizar as necessidades dos usuários com a implementação dos desenvolvedores.

Entretanto, para construir um PDS que seja efetivo, não basta apenas escolher um ciclo de vida de processo, deve-se considerar a complexa inter-relação organizacional, cultural, tecnológica e econômica (Fuggetta, 2000). Sendo assim, deve-se ter em mente que criar um processo capaz de conservar todas essas variáveis não é uma tarefa fácil.

Dentre estas variáveis, estão definidas **atividades**, **papéis** e **artefatos** que serão utilizados durante o desenvolvimento de software. As atividades especificam as tarefas que devem ser executadas durante o processo, os papéis descrevem as pessoas que cumprirão com as atividades do processo e os artefatos armazenam as informações criadas durante a realização das atividades.

Neste sentido, as atividades do PDS estão concentradas na criação, manipulação e gerenciamento de Artefatos de Software que representam desde a especificação do Software até a sua implementação, passando por etapas como planejamento, testes e controle de qualidade.

A construção de um Produto de Software (PS) deve ser feita mediante a execução de um PDS, sendo que cada execução é única e gera diferentes tipos de artefatos e em diferentes quantidades (Laitinen, 1992).

Tais artefatos são necessários para que exista boa comunicação entre todos os interessados no desenvolvimento do produto. Para garantir uma comunicação eficiente faz-se necessário conhecer as informações contidas em um artefato, assim como os responsáveis por estas informações.

O Artefato de Software (AS) é vital para o PDS e deve ser tratado de maneira adequada para evitar o surgimento de problemas como, falha no preenchimento, má interpretação e duplicação das informações. Além disso, a manipulação torna-se mais trabalhosa quando eles apresentam inconsistência e ambigüidade, visto que isto é proveniente da má estruturação do seu conteúdo, já que artefatos são constituídos a partir de estruturas lógicas e representações particulares de informação (Quint & Vatton, 2004).

Portanto, neste trabalho apresentamos uma solução para que seja possível: (i) categorizar, organizar, classificar, versionar e definir níveis de maturidade de ASs; (ii) formalizar e definir representações na estrutura e nos tipos de informações de ASs; (iii) reutilizar a estrutura, a informação e as possíveis versões de ASs; em um PDS.

1.1 Motivação

Como já era esperado, muitos processos de desenvolvimento de software encontrados na literatura, assim como *Rational Unified Process* (RUP) (Kruchten, 2000, Kroll & Kruchten, 2003), *Object-oriented Process, Environment and Notation* (OPEN) (Graham et al., 1997) e SCRUM (Schwaber, 2004), compreendem artefatos de forma monolítica.

Ao analisarmos um processo específico é possível perceber a real importância dos artefatos para a obtenção do produto. No caso do RUP, são definidos aproximadamente 100 artefatos que, em sua grande maioria, são renderizados através de processadores de texto como o *Microsoft Office Word*™ou *Open Office Writer*[1].

Conforme já foi mencionado, ASs são utilizados à medida que um PDS é executado (*enactment*), porém eles são especificados durante a definição do PDS, ou seja, na autoria (*authoring*).

O nível de autoria de um PDS é realizado em duas fases, (i) **Definição** de elementos do PDS e (ii) **Uso** dos elementos pelo PDS. Desta forma, pode-se consumir os elementos que forem necessários para projetar um PDS efetivo para um determinado domínio.

1.1.1 Definição dos Elementos de PDS

Conceitualmente cada artefato deve ser preenchido com informações específicas que surgem durante a execução de algum projeto em que fazem parte. Entretanto, à medida que se conhece o comportamento do projeto e com base na experiência em construção dos artefatos, muitas vezes já se sabe o que e como devem ser preenchidos, mesmo que não exista um projeto, mas apenas o processo. Este é o tipo de informação necessária para que seja possível determinar um arcabouço conceitual que define como serão os artefatos.

Na Figura 1.1 podem ser vistos o modelo conceitual de artefato e sua instância na fase de autoria. Em **a** estão os elementos Definição de Artefato, que neste caso representa um AS, e Tipo de Informação, em um relacionamento **n:m** ou seja, o mesmo artefato pode possuir muitos tipos de informação e uma informação, por sua vez, pode estar em vários artefatos ao mesmo tempo. Além disso, em **b** é visto o que comumente acontece, um único tipo de informação (*T*) é representado através de diferentes tipos, muito parecidos (*T1*, *T2* e *T3*) que são utilizados várias vezes por diferentes artefatos (*A1 A2* e *A3*) sem que seja percebida.

Diante deste contexto, o mesmo tipo de informação será utilizado despercebidamente e perde-se o reuso. Outros pontos fracos são:

- a mesma estrutura geralmente será utilizada para representar a mesma informação. Sem reuso da estrutura o reuso da informação poderá não ser identificado;

[1] www.openoffice.org

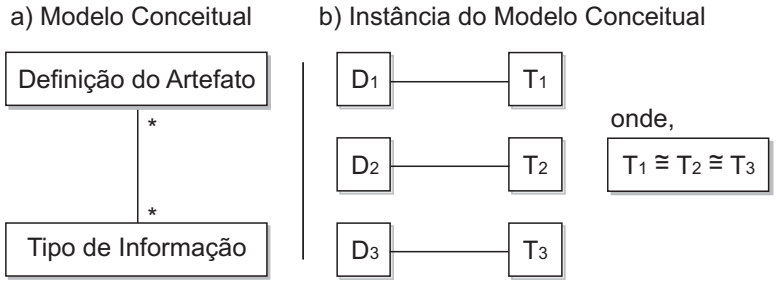

Figura 1.1: Modelo conceitual de Artefato na fase de Definição.

- existe muita confiança no papel que será responsável pela produção do artefato que, sem conhecer exatamente os padrões e estruturas estará livre para fazer o que quiser;
- sem possuir uma estrutura formal é computacionalmente impossível utilizar a informação, a exemplo disto, pode-se fazer um paralelo com linguagens de programação, que são formalizadas para que possam ser computáveis.

1.1.2 Uso dos Elementos já criados por um PDS

Na Figura 1.2 estão o modelo conceitual de artefato e sua instância no que se diz respeito ao seu uso. Em **a** estão os elementos Artefato e Informação, em uma agregação **n:m** ou seja, um artefato pode possuir muitos tipos de informação ou ao menos um, portanto, cada informação pode estar em mais de um artefato. Em **b** pode ser observado como se dá a sua construção e, supostamente, a mesma informação pode ser repetida em *I1*, *I2* e *I3*, estando dentro dos artefatos (*A1 A2* e *A3*).

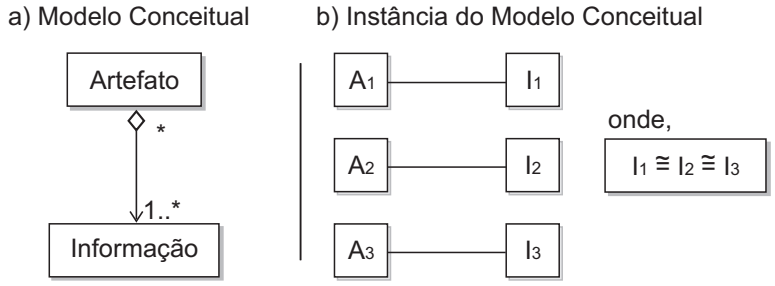

Figura 1.2: Modelo conceitual de Artefato na fase de Uso.

Sendo assim, existe pouco reuso da informação que, ao se repetir (utilização de copia e cola), gera redundância. Outros pontos fracos são:

- podem ser feitas muitas cópias da mesma informação;
- como são cópias de uma mesma informação, *I1*, *I2* e *I3* podem ter sido alteradas separadamente à medida que foram introduzidas nos artefatos, estando em versões diferentes e denominando falta de controle;

- não há rastreabilidade entre os artefatos e as informações, ou seja, difícil identificar se a mesma informação está disposta em diferentes artefatos.

1.2 Problemática

Segundo (Kruchten, 2000), a demanda por softwares cada vez mais complexos é maior do que a capacidade de produzi-los através dos processos e metodologias conhecidas até então. Baseados nesta idéia, muitos autores tenderam ao desenvolvimento de metodologias baseadas em casos de uso (*use-case methodology*) (Williams, 2004). Portanto, houve mudança na construção dos processos de software. Tal mudança foi feita para suportar uma nova metodologia dirigida pela documentação de um projeto.

Engenheiros do Software acreditam que a documentação de um produto ajuda no entendimento não só funcional, mas também em nível de projeto e em detalhes de implementação do mesmo. Sem documentação, os Engenheiros de Software são forçados a confiar apenas no código-fonte do produto (Hartmann et al., 2001). Uma boa documentação com qualidade ajuda e facilita a conclusão das tarefas, na qual várias e complementares perspectivas são fornecidas para melhor entendimento de um sistema (Tilley & Müller, 1991).

Produzir o melhor software possível, juntamente com a melhor documentação possível é um dos principais objetivos da Engenharia de Software (Visconti & Cook, 1993). Para (Hartmann et al., 2001), o controle de informações é feito a partir de um processo de documentação divido em três partes: coleta a partir de múltiplas fontes de informação, processamento de toda a informação coletada e publicação da mesma através de vários tipos de artefatos (geralmente documentos como manuais e descritores). Entretanto, algumas vezes a documentação é considerada plano de fundo e não tão importante quanto o próprio software. Para (Visconti & Cook, 1993), a falta ou baixa qualidade da documentação é a maior causa de erros e necessidades de manutenção em um produto de software. Portanto, é notável a necessidade e a importância de documentação em um processo de software.

Problematica 01 (P1) PDSs entendem artefatos como representações monolíticas, visto que todo o processo gira em torno da produção de ASs ao invés da informação contida no mesmo. Sendo assim, ao se deparar com um AS, não é possível saber como o mesmo está estruturado ou ao menos quais são as informações necessárias para preenchê-lo. Portanto, o processo enxerga a informação como um conjunto, não como os elementos de uma composição de várias estruturas de informação existentes.

Problematica 02 (P2) Linguagens de Modelagem de Processos não possuem os níveis de profundidade necessários para definir todos os elementos necessários de um PDS. Sendo assim, dentro do escopo de AS mais especificamente, tais linguagens não determinam alguma representação em níveis de informação. Isto demonstra que não é possível se obter

o detalhamento do AS, evitando que sua definição seja desmembrada de sua utilização. Tal fato implica em ambigüidade e inconsistência. Além disso, deixa dúvidas durante a execução do PDS, tais como:

- Qual formato dos artefatos? Como é sua estrutura interna? Quais informações devem preenchê-lo (e.g. gráficos, imagens, *links*)?
- Como artefatos devem ser preenchidos ou gerados? Existe alguma particularidade no estilo ou maneira de tornar um artefato mais familiar ou mais compreensível?
- Como versionar o artefato identificando onde houve modificação e por quem?
- Como identificar e evitar redundância tanto na estrutura quanto no conteúdo dos artefatos em uma tentativa de reuso?

Problematica 03 (P3) Não existe uma maneira clara para (i) determinar e especificar ASs e (ii) definir qual o fluxo a ser realizado durante a definição e o uso de ASs.

1.3 Questão de pesquisa

Conforme visto anteriormente, processos como RUP, OPEN e SCRUM, comumente utilizados tanto na indústria quanto na academia, tratam seus artefatos de maneira monolítica. Não obstante, linguagens para descrição de processos são bastante genéricas e tratam os componentes do processo da mesma forma.

Este tipo de prática pode levar a problemas como, por exemplo, ambigüidade e inconsistência na autoria dos artefatos e diminuição do grau de reuso de informação. Como estes artefatos são, em sua maioria, semi-estruturados, a falta de um formalismo pode causar problemas ao se computar as informações existentes. Portanto, surge a questão de pesquisa:

Como deve ser a autoria de Artefatos de Software de forma que estes não sejam monolíticos?

1.4 Objetivos

Embora artefatos sejam vistos como se fossem pedaços de informação sem estrutura ou forma definida, estes possuem estruturas lógicas internas e diferentes representações de informação. Nem sempre é possível perceber ou verificar a existência de redundância sem entender o artefato como um todo. Como existem diferentes ASs, a solução deste problema requer conhecimento sobre a estrutura de cada um deles desde que cada um possua uma estrutura interna diferente. Infelizmente, este tipo de tratamento não é verificado em linguagens de autoria ou execução de PDSs. Neste contexto, alguns desafios são:

1. Como construir artefatos de forma que sua organização interna esteja bem estruturada, facilitando sua compreensão e manipulação?

2. Como definir diversas versões, autores e estruturas de informação aos artefatos durante sua autoria, evitando a redundância de informação, de estrutura e retrabalho?

Baseados nestas dúvidas nosso objetivo é desenvolver uma abordagem que provê a autoria de artefatos determinando maiores detalhes sobre suas estruturas de informação. Diante disto nossos objetivos específicos são:

1. Identificar o fluxo de atividades (ou passos) necessárias para a realização de autoria nos níveis de definição e uso de ASs.

2. Estruturar ASs para permitir o reuso estrutural e flexibilidade em sua autoria e produção.

3. Especificar ASs com versionamento, maturidade e controle estrutural para que seja possível o controle e o gerenciamento da informação.

4. Permitir o reuso de estrutura e conseqüentemente de conteúdo das informações existentes.

5. Adquirir conhecimento sobre a construção de ASs já na fase de autoria de um PDS, removendo esta responsabilidade da fase de execução.

6. Propor uma linguagem de autoria de ASs capaz de suprir as necessidades identificadas.

7. Desenvolver uma ferramenta de software e produzir um protótipo, tendo como base a linguagem desenvolvida.

8. Elaborar casos de testes envolvendo a autoria de artefatos para um processo real que utilize o protótipo da ferramenta a ser desenvolvida.

1.5 Caracterização da Contribuição

A abordagem em desenvolvimento representa esforços para a solução dos problemas apontados nas seções anteriores. Além disto, este trabalho apresentará contribuição para autoria de artefatos. Um breve resumo das contribuições a serem geradas pela pesquisa descrita neste trabalho é apresentado a seguir:

Especificação de um Guia para a Autoria de Artefatos: conforme a necessidade de conhecer ASs e suas estruturas, foi desenvolvido um guia para definir a autoria de PDSs juntamente com a de ASs em detalhes, permitindo reuso de definição e uso dos mesmos. Além disso é apresentado como elaborar o desenvolvimento de artefatos em partes menores, onde papéis atuarão nestes fragmentos.

Levantamento de estruturas de informações contidas em artefatos conhecidos: com a obtenção de conhecimento sobre o formato estrutural dos ASs e seu preenchimento será possível um melhor entendimento sobre o ciclo de produção do mesmo. As informações que permeiam a forma do artefato poderão ser verificadas e avaliadas iterativamente, onde poderemos utilizar versionamento tanto na definição, quanto no uso dos artefatos. Além do mais, o aumento no compartilhamento de conteúdo e estrutura, baseado no reuso, poderá diminuir o tempo de elaboração e produção dos artefatos.

Criação de uma Meta-Linguagem: com uma linguagem de definição de ASs podemos construí-los em detalhes, utilizando estruturas lógicas e organização bem definida. Portanto, criamos um metamodelo que permite a autoria e produção de ASs em uma linguagem com vários níveis de formalismo, servindo como base para os esforços futuros na área de autoria de PDSs, assim como, para o desenvolvimento de aplicações neste contexto. Nesse sentido, foram desenvolvidos: (i) um metamodelo que define a sintaxe e semântica do problema e (ii) regras de boa formação que especificam as restrições necessárias;

Ferramenta Protótipo de Suporte: A implementação do metamodelo conforme um *plug-in* construído com base no *Eclipse Modeling Framework* (EMF) (Duddy et al., 2003) disponibilizará recursos para a autoria de ASs.

1.6 Desenho de Pesquisa

O trabalho está dividido em etapas que agrupam as atividades relacionadas. Tais etapas se relacionam através de fluxo de atividade que pode ser visualizado na Figura 1.3 e serão descritas conforme a ordem em que foram realizadas:

- **Etapa 1:** (i) realizar o levantamento bibliográfico sobre os PDSs. Esta etapa foi contínua e visou verificar o impacto de outros trabalhos na área na abordagem proposta; (ii) fazer levantamento sobre tipologia de artefatos através de estudo teórico; (iii) fazer aprofundamento de estudos sobre os mecanismos de extensão de metas-linguagem baseadas em autoria de PDSs ou ASs.

- **Etapa 2:** (i) identificar os artefatos mais utilizados nos PDSs analisados na Etapa 1; (ii) identificar a tipologia dos ASs, selecionando um mapeamento para cada estrutura de informação diferente; (iii) propor a criação, utilização ou extensão de uma linguagem ou meta-linguagem para permitir a quebra do AS em uma estrutura mais granular de acordo com as tipologias encontradas.

- **Etapa 3:** realizar a análise dos resultados obtidos das etapas 1 e 2 para: (i) especificar a meta-linguagem que defina as tipologias inspecionadas; (ii) elaborar o guia de autoria de ASs; (iii) propor uma ferramenta de suporte para verificação da abordagem; (iv) análise e projeto da ferramenta; (v) implementação da ferramenta proposta;

- **Etapa 4:** (i) realização de caso de teste sobre o protótipo, objetivando a prática do que foi desenvolvido nas etapas anteriores; (ii) finalização da escrita e submissão de artigos; (iii) entrega do seminário de andamento; e (iv) revisão, entrega e defesa da dissertação.

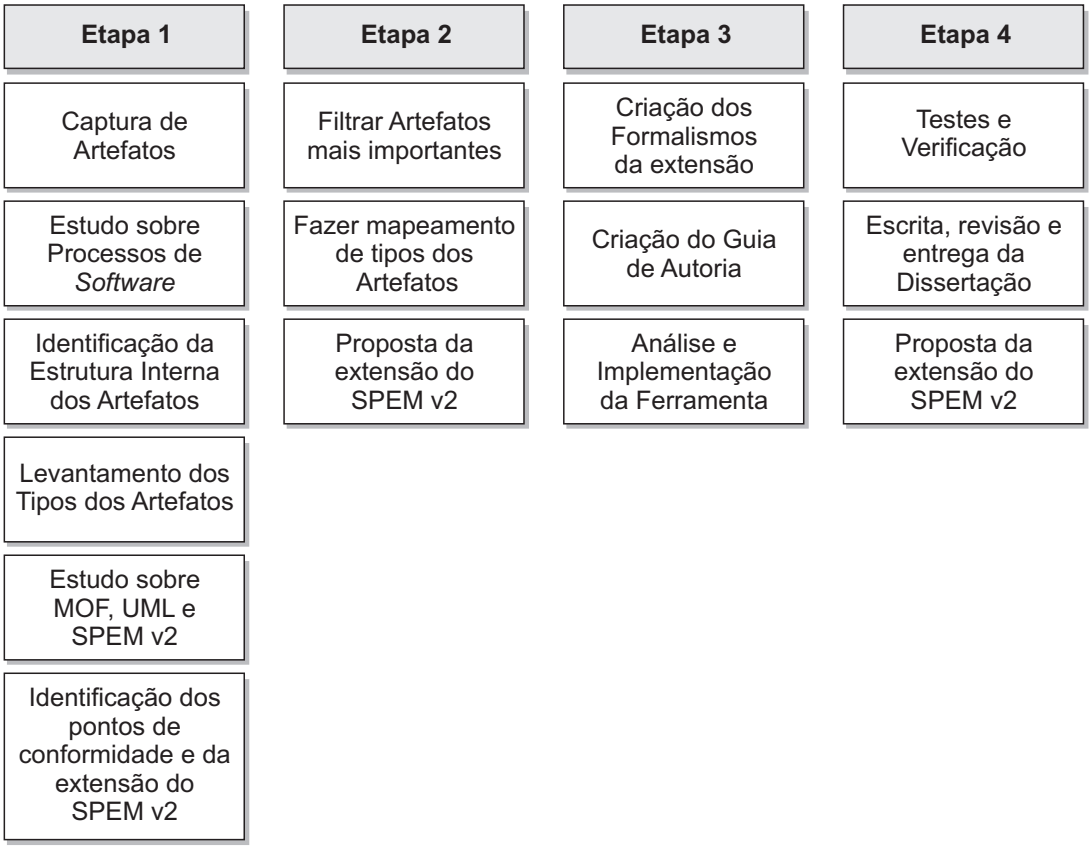

Figura 1.3: Diagrama de distribuição de Etapas x Atividades

Para a realização da Autoria de Artefatos em Processos de Software foram definidas as seguintes atividades (na Figura 1.4, é apresentado um quadro que representa a execução das atividades no decorrer do mestrado):

- **Atividade 1:** capturar e identificar os artefatos, conhecer suas estruturas internas (ex.: gráficos, textos, figuras, seções, listas), elaborar e estudar a fundamentação teórica e identificar trabalhos relacionados.

- **Atividade 2:** Estudo dos PDSs, ASs, tipologias e linguagens PDSs. Foram realizados estudos sobre PDS mais populares, a fim de examinar os tipos de ASs existentes. Também foi feito exame sobre as especificações de PDS e metamodelos que representam artefatos específicos de processos.

- **Atividade 3:** Realizar uma análise dos resultados obtidos das etapas 1 e 2 para: identificar e filtrar os artefatos mais utilizados nos PDSs analisados; identificar a tipologia dos ASs inspecionados, selecionando um mapeamento para cada estrutura de informação

diferente; propor a criação de uma metalinguagem que permita a quebra de um AS em uma estrutura mais granular. Nesta atividade definimos a utilização da linguagem padrão de PDS ao longo do trabalho, entendendo o seu funcionamento e definindo alguns pontos problemáticos existentes que inviabilizam a criação de ASs em detalhes até então.

- **Atividade 4:** Especificação formal das tipologias a serem utilizadas e especificação da meta-linguagem. Esta atividade foi fundamental para a definição dos conceitos necessários para a extensão da linguagem selecionada na Atividade 3. Sendo assim, foi definido Ponto de Conformidade (*Compliance Point*) a ser utilizado.

- **Atividade 5:** Análise e projeto da ferramenta.

- **Atividade 6:** Implementação de um protótipo da ferramenta.

- **Atividade 7:** Testes e verificação. Realização dos casos de teste sobre o metamodelo desenvolvido com o uso do protótipo da ferramenta a ser implementado na Atividade 6.

- **Atividade 8:** Preparação e apresentação de Seminário de Andamento.

- **Atividade 9:** Redação de artigos científicos.

- **Atividade 10:** Redação e revisão da Dissertação de Mestrado.

- **Atividade 11:** Entrega e defesa da Dissertação.

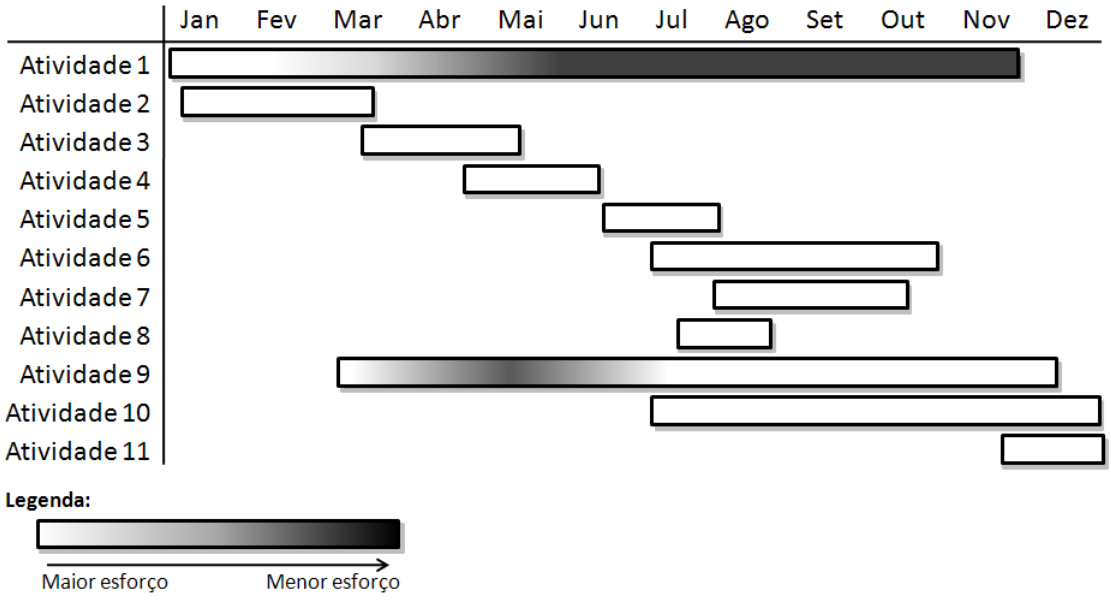

Figura 1.4: Diagrama de distribuição de esforços

1.7 Estrutura do Trabalho

Este trabalho se encontra organizado da seguinte maneira:

- Capítulo **2:** apresenta a fundamentação teórica das tecnologias e conceitos sobre os principais temas discutidos no decorrer do trabalho;
- Capítulo **3:** apresenta os principais trabalhos com soluções relacionadas à abordagem proposta, sob uma visão comparativa;
- Capítulo **4:** apresenta a solução encontrada assim como a descrição detalhada dos passos que foram seguidos para sua implementação;
- Capítulo **5:** apresenta os testes feitos através do uso da abordagem. Neste Capítulo também é apresentada a ferramenta de suporte a autoria de artefatos de software SwAT;
- Capítulo **6:** neste Capítulo é apresentada a conclusão do trabalho.

2 Fundamentação Teórica

Neste capítulo apresentamos um resumo explicativo sobre os principais conceitos utilizados nesta dissertação. Nosso objetivo é facilitar a leitura deste trabalho, determinando a base da fundamentação teórica estudada durante a pesquisa. Este capítulo está definido da seguinte forma: na Seção 2.1 está definido o conceito de AS juntamente com alguns exemplos; a Seção 2.2 apresenta uma breve história sobre PDS, desde seu conceito base até sua utilização conforme exemplos de PDS existentes e bastante conceituados; a Seção 2.3 apresenta fundamentação sobre modelagem e metamodelagem; e, por fim, a Seção 2.4 apresenta um breve resumo sobre a linguagem de modelagem de PDS utilizada no decorrer deste trabalho.

2.1 Artefatos de Software

Artefato de Software é identificado como um dos principais elementos que fazem parte do núcleo de PDSs, que descreve **quem** está fazendo **o que**, **como** e **quando**. Neste caso, um Artefato de Software é **o que está sendo feito**.

Segundo (Kroll & Kruchten, 2003), um artefato é um pedaço de informação que é produzida, modificada ou utilizada por um processo. O que é uma definição bem razoável de um ponto de vista mais amplo. Além disso, o mesmo autor ainda afirma que estes devem ser elementos tangíveis de um projeto, produzidos durante a construção de um produto final.

Entretanto, para (Kruchten, 2000), ASs são produtos de trabalho finais ou intermediários produzidos e usados durante os projetos, os quais são utilizados para capturar e transmitir informações do projeto.

Restringindo um pouco o domínio para PDSs, artefatos são inerentes à execução de alguma atividade, para a qual pode ser tanto uma entrada, para que a atividade possa ser executada, como uma saída ou resultado de uma execução que poderá ser utilizado em outras atividades.

Sendo assim, artefatos podem ser de várias formas ou formatos diferentes. Dentre eles podendo ser:

- um **modelo**, assim como Modelo de Casos de Uso da UML;
- um **elemento de um modelo**, que é um elemento dentro de um modelo, assim como um Caso de Uso (UC);
- um **documento**, como por exemplo o documento Visão do RUP;

- executáveis, como um Protótipo executável por exemplo.

Como artefato é um termo utilizado no RUP, em inglês *artifact*, outros processos podem utilizar termos bem diferentes como produto de trabalho, unidade de trabalho e outros, para designar o mesmo elemento. Sendo assim, não existe uma padronização.

Para tornar isto mais claro e por questões de definição de uma nomenclatura padrão, apresentamos a Tabela 2.1, que utiliza uma tradução para a terminologia em diversos outros processos. Como pode ser observado, embora o a utilização de *Product* seja feita com maior freqüência dentre os PDSs analisados, continuaremos a utilizar o termo Artefato de Software (AS).

Tabela 2.1: Mapeamento da terminologia utilizada em diversos Processos de Desenvolvimento de Software analisados com base em (OMG, 2005)

PDS	**Termo em Português**	**Termo original**
Rational Unified Process	artefato	*artifact*
Unisys QuadCycle	artefato, ativo	*artifact, asset*
OOSP	entregável	*deliverable*
PMBOK	entregável	*deliverable*
DMR Macroscope	produto entregável	*deliverable product*
Fujitsu SDEM21	documento, arquivo	*document, file*
IEEE 1074-1997	produto	*product*
ISO/IEC 12207	produto	*product*
Promoter	produto	*product*
OPEN	produto de trabalho	*work product*
IBM Global Services Method	descrição de produto de trabalho	*Work Product Description*

Por fim, em termos de construção, um AS pode ser documentado de duas formas (Kruchten, 2000, Kroll & Kruchten, 2003): (i) através de formalização, com uso de ferramentas ou (ii) informalmente, capturados através de *e-mails*, por exemplo.

2.2 Processos de Desenvolvimento de Software

A noção de processo é comum e faz parte das atividades humanas e em sua definição mais elementar é dado como sendo uma forma sistemática de criação de um produto de acordo com algumas tarefas pré-determinadas (Osterweil, 1987).

Processo é um verbete de origem direta do Latim *processus*, que significa a idéia do que segue adiante, do que avança no eixo do tempo. Este verbete exibe uma origem etimológica mais distante, seu significado tem a ver com a palavra grega *proodoz* (leia-se próodos) que traduz progresso, progressão, avanço.

Processos são geralmente utilizados para se criar formas efetivas e, possivelmente, genéricas na construção de algum produto. O processo, em sua essência, é visto como o alcance de uma

solução e, conforme as descrições dos passos do processo, executá-lo como uma instância. Esta instância servirá para resolver problemas mais específicos (Osterweil, 1987).

Entretanto, muitas vezes não é fácil entender um processo, e isto se dá por existir uma diferença fundamental entre a descrição e o próprio processo. Para Dijkstra, 1979, enquanto a descrição do processo é uma entidade estática e de fácil análise e compreensão, o processo em si é dinâmico e um pouco mais difícil de ser compreendido. Como exemplo, pode-se tomar uma receita de bolo: a receita simplesmente descreve como o bolo deve ser feito; o processo será o preparo do bolo. Neste exemplo, fica claro que enquanto um processo é um instrumento para se fazer algo, sua descrição especifica como algo deve ser feito.

No contexto da computação, um processo é uma tarefa em execução inserida em um dispositivo computacional (Osterweil, 1987). No entanto, não existe um consenso entre os autores no tocante a definição de um Processo de Desenvolvimento de Software (PDS).

De acordo com (Sommerville, 2004), um PDS pode ser definido como um conjunto de atividades, métodos, práticas e transformações que as pessoas empregam para desenvolver e manter o software e os produtos associados (por exemplo, planos de projeto, documentos de projeto, *design*, código, casos de teste, manual do usuário).

Não muito obstante, para Soares, 2004, um processo de software é um conjunto coerente de atividades e resultados associados que auxiliam na produção de software. Para (Fuggetta, 2000), processo de software nada mais é que um conjunto coerente de ações (*policies*), estruturas organizacionais, tecnologias, procedimentos e artefatos necessários para conceber, desenvolver, implantar e manter um produto de software.

Um PDS define a seqüência em que os métodos serão aplicados, como os produtos serão entregues, os controles que ajudam a assegurar a qualidade e a coordenar as mudanças, e os marcos de referência que possibilitam aos gerentes de software avaliar o progresso do desenvolvimento.

Sendo assim, PDSs são importantes porque tentam uniformizar a realização dos Produtos de Software (PS) através de projetos, reduzindo o tempo de produção e custos. Caso um processo esteja explicitamente correto, o desenvolvimento do software será feito de forma sistemática e ordenada (Cugola & Ghezzi, 1998). Conforme (Fuggetta, 2000), existe uma relação quase que direta entre a qualidade de um PS e o PDS que o produziu.

Desta forma, o PDS faz parte da camada mais importante da Engenharia de Software (ES) e se constitui no elo de ligação entre as ferramentas e os métodos, além de possibilitar um desenvolvimento racional do software (Sommerville, 2004). Neste caso, a ES caracteriza-se pela produção de um PS através de processos de qualidade (Cugola & Ghezzi, 1998).

Entretanto, para construir um PDS que seja efetivo, não basta apenas escolher um ciclo de vida de processo, deve-se considerar a complexa inter-relação organizacional, cultural, tecnológica e econômica (Fuggetta, 2000). Sendo assim, deve-se ter em mente que criar um processo capaz de conservar todas essas variáveis não é uma tarefa fácil.

Segundo (Noll, 2007), a criação, construção e utilização de processos envolve (i) a captura

e a análise de práticas existentes dentro da construção de algum produto ou fornecimento de serviço, (ii) redesenho e simulação, (iii) instalação e gerenciamento, (iv) captura e auditoria do histórico dos acontecimentos e (v) refinamento e melhoria.

O ciclo da gerência de processos é apresentado na Figura 2.1 e está dividido em três instantes. No instante 0, em branco, está a modelagem do PDS, do qual este instante é conhecido por autoria (*authoring*). O instante 1, em tom mais escuro, apresenta a execução (*enactment*) do PDS. Entre esses dois instantes, em tom claro, pode-se adicionar um instante de re-análise e remodelagem (*tailoring*).

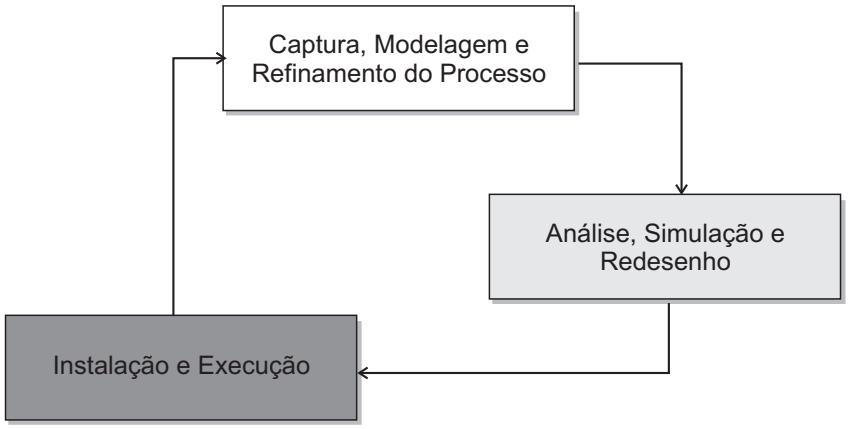

Figura 2.1: Ciclo de vida de um Processo de Desenvolvimento de Software

Nas próximas Seções serão apresentados PDSs já consolidados. Na Seção 2.2.1 é apresentado o RUP, um exemplo de PDS baseado em metodologias tradicionais de desenvolvimento; já a Seção 2.2.2 está o Scrum, como instância de metodologia ágil.

2.2.1 *Rational Unified Process* (Kruchten, 2000)

O *Rational Unified Process* (RUP) é um Processo de Desenvolvimento de Software desenvolvido pela *Rational Software* da IBM. Atualmente o RUP é bastante conhecido, pois provê uma abordagem disciplinada que determina tarefas e responsabilidades, e muito utilizado devido ao fato de ser um arcabouço de processos de software (*process framework*). Isto faz com que possa ser adaptado ou estendido de acordo com as necessidades do contexto organizacional. A estrutura do RUP é dividida em uma arquitetura com duas dimensões, Figura 2.2.

O eixo horizontal representa o tempo, caracterizando o ciclo de vida do processo. Este eixo representa os aspectos dinâmicos do processo (*enactment*), e utiliza ciclos, fases, iterações e *milestones*. Já o eixo vertical representa as principais disciplinas que agrupam um conjunto de atividades, onde estas são consideradas a parte estática do processo. Nesta dimensão estão os componentes do processo, ou seja, as atividades, disciplinas, artefatos e papéis.

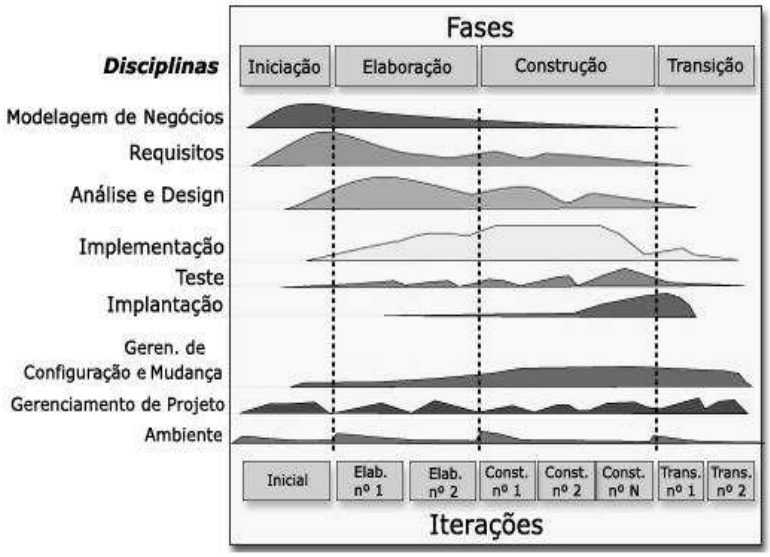

Figura 2.2: Estrutura do RUP em duas dimensões. Adaptado de (Kruchten, 2000)

Estrutura Estática - Descrição do Processo

O modelo do RUP descreve quem (*who*) está fazendo o que (*what*), como (*how*) e quando (*when*)! Portanto, o *Rational Unified Process* é representado por cinco elementos principais:

- **Papéis** (*Roles*), quem (*Who*). Define o comportamento e as responsabilidades de um indivíduo ou um grupo. Cada papel é associado a um conjunto de características que são requisitos para quem o atuará. Existem cinco tipos de papéis: *analyst*, *developer*, *tester*, *manager* e *production*/*support*;

- **Atividades** (*Activities*), como (*How*). É uma unidade de trabalho para que determinado indivíduo, atuando em determinado papel requerido, produza algo que faça sentido ao projeto em questão. Atividades são quebradas em etapas (*Steps*), dos quais podem ser divididos em três categorias principais: *thinking steps*, *performing steps* e *reviewing steps*;

- **Artefatos** (*Artifacts*), o que (*What*). Um artefato é qualquer pedaço de informação que é produzido, alterado, ou modificado através do processo. Além disto, artefatos são unidades tangíveis de um projeto. Os artefatos são apresentadas com maiores detalhes mais adiante;

- **Fluxos** (*Workflows*), quando (*when*). É uma sequência de atividades que produzem um resultado. O RUP possui três tipos de fluxos: *core workflows*, *workflow details* e *iteration plans*;

- **Disciplinas** (*Disciplines*). São contêineres que agrupam os primeiro quatro elementos anteriores. As disciplinas são apresentadas com maiores detalhes mais adiante.

Arquitetura

Uma boa parte do RUP é baseada em modelagem. Modelos ajudam a entender melhor um problema para imaginar uma solução. Um modelo na verdade é simplificação da realidade, ajudando a reduzir o domínio e diminuir a complexidade. Um único modelo não é suficiente para cobrir todos os aspectos do desenvolvimento de software. Desta forma, é necessário obter múltiplos modelos que devem estar corretamente relacionados e consistentes.

Baseado nas arquiteturas existentes, o RUP sugere uma abordagem com cinco perspectivas (conhecido por arquitetura 4 + 1): *Logical View*, *Implementation View*, *Process View*, *Deployment View* e *Use-Case View* Figura 2.3.

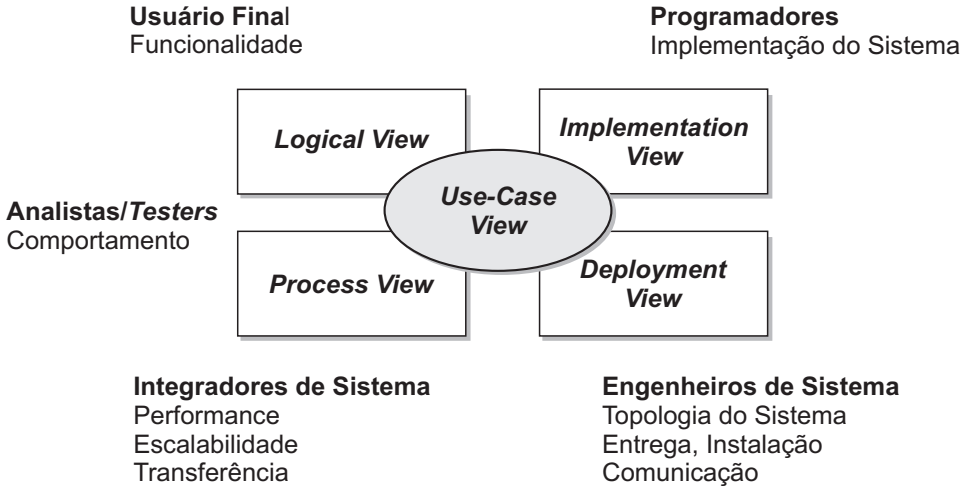

Figura 2.3: Modelo de arquitetura 4 + 1 do RUP. Adaptado de (Kruchten, 2000)

- ***Logical View***: Esta perspectiva endereça os requisitos funcionais do sistema, em outras palavras, o que o sistema deve fazer. Esta perspectiva é uma abstração do modelo, identificando como devem ser os pacotes, subsistemas, classes, entre outros.
- ***Implementation View***: Esta perspectiva descreve como a parte estática do software deve ser organizada em módulos (código-fonte, arquivos de dados, componentes, executáveis, entre outros), dentro de um ambiente em termos de empacotamente e configuração.
- ***Process View***: Esta perspectiva endereça os aspectos concorrentes de um sistema em tempo de execução. São analisados *deadlocks*, tempo de resposta, isolamento de funções e falhas.
- ***Deployment View***: Esta perspectiva define como os vários executáveis e outros componentes de execução são mapeados para uma plataforma computacional. São endereçados problemas como instalação e performance.
- ***Use-Case View***: A última perspectiva, baseada em casos de uso, contém os cenários do sistema. Inicialmente esta perspectiva é utilizada para direcionar a descoberta e projeto da arquitetura nas fases inicial e elaboração.

Artefatos

Um artefato é qualquer pedaço de informação que é produzido, alterado, ou modificado através das atividades de um PDS. Além disso, artefatos são unidades tangíveis de um projeto e sua soma determina o PS final. Um artefato pode ser de várias formas ou características, dentre elas: um modelo, um elemento de um modelo, um documento, código fonte e executáveis;

O RUP é muito famoso por pregar que sua documentação seja feita com alto grau de burocracia. São definidos aproximadamente 100 artefatos dos quais, aproximadamente 55, são editados através de processadores de texto. O padrão estabelecido para definição dos artefatos é baseado em *templates* e exemplos.

Tais *templates* são moldes em linguagem natural que definem como os artefatos devem ser preenchidos. Os exemplos são instâncias de *templates* já preenchidos.

Disciplinas

Disciplinas, são conjuntos de atividades do processo, organizadas logicamente. Existem nove disciplinas e elas são divididas entre técnicas e de suporte:

- **Disciplinas Técnicas**: Modelagem de Negócio, Requisitos, Análise e Projeto, Implementação, Teste e Instalação.

- **Disciplinas de Suporte**: Gerência de Projeto, Configuração e Mudanças e Ambiente.

2.2.2 Scrum (Schwaber, 2004)

O Scrum é um processo de gerenciamento de software do qual, pode-se criar um projeto de desenvolvimento de software de forma iterativa e incremental (Krebs, 2007). Tal PDS foi introduzido em 1995 pela *Advanced Development Methodologies*, ganhando popularidade depois da formação da *Agile Alliance*, em 2001. O processo em si é ágil (*agile*) e leve (*lightweight*), permitindo também, a combinação com novos processos ágeis, podendo funcionar como um empacotador.

Conforme (Krebs, 2007), o Scrum oferece uma abordagem empírica, permitindo que membros de uma equipe trabalhem de forma independente e coesiva, dentro de um ambiente criativo. Desta forma, este PDS reconhece a importância do aspecto social em ES.

Segundo o trabalho de Schwaber, em 2004, o processo em si é bastante rápido, adaptativo, e organizado, além de ser bastante diferente dos processos de software seqüenciais. Na verdade, o Scrum acredita que um software não deve ser desenvolvido como se fosse um produto manufaturado, ou seja, de uma forma repetitiva.

O coração e o esqueleto do Scrum

Todas as práticas do Scrum são feitas dentro de um esqueleto de processo incremental e iterativo, Figura 2.4.

Figura 2.4: Esqueleto do Scrum. Adaptado de (Schwaber, 2004)

O ciclo mais abaixo representa uma iteração para as atividades em desenvolvimento que ocorre uma após a outra. As saídas de cada iteração serão incrementos do produto. Enquanto que, o ciclo mais acima representa a inspeção diária feita sobre a iteração, onde os membros da equipe são convidados a inspecionarem-se (uns aos outros), verificando suas atividades a fim de adaptá-las caso seja conveniente. Uma iteração é direcionada através de uma lista de requisitos e este ciclo se repete enquanto o projeto for financiado.

Baseado neste conceito, o funcionamento do esqueleto do Scrum opera da seguinte maneira: ao iniciar uma iteração, o time revê tudo o que deve ser feito. Em seguida, o time seleciona as potenciais funcionalidades em que ele acredita que possam ser entregues até o final da iteração. A partir daí, o time utilizará o tempo restante da iteração para fazer as funcionalidades previstas. Ao final da iteração o time apresenta o incremento de funcionalidades que foi construído a partir de um *stakeholder*, que inspeciona o que foi feito e pode propor modificações no projeto.

Papéis do Scrum

O Scrum implementa seu esqueleto iterativo e incremental através de três papéis: ***Product Owner***; o time (***Team***); e o ***Scrum Master***. Todo o gerenciamento de responsabilidades de um projeto será dividido entre esses três papéis. O *Product Owner* é responsável por representar os interesses de todos os participantes do projeto e do sistema resultante.

O *Product Owner* consegue o financiamento através do levantamento dos requisitos globais do projeto, retorno do investimento (conhecido por *Return of Investment* - ROI) e os planos dos entregáveis. A lista de requisitos é chamada de ***Product Backlog***. O *Product Owner* é responsável por utilizar a *Product Backlog* para garantir que um recurso mais necessário seja feito primeiro; isto é alcançado priorizando a lista de requisitos, freqüentemente, utilizando uma fila de funcionalidades por ordem de necessidade.

O *Team* é responsável por desenvolver as funcionalidades. Os *Teams* são auto-gerenciáveis, auto-organizados, sendo responsáveis por descobrir como transformar a *Product Backlog* em uma funcionalidade incremental dentro da iteração, gerenciando o próprio tempo e trabalho. Os membros de um *Team* possuem a responsabilidade coletiva do sucesso do projeto em cada iteração até completá-lo.

O *Scrum Master* é o responsável pelo processo, por ensinar Scrum a todos os envolvidos no projeto e por implementar o Scrum de forma que se encaixe culturalmente na organização.

Fluxo do Scrum

No Scrum, um projeto inicia com visão sistêmica, ou seja, uma visão global de todo o sistema a ser desenvolvido. Esta visão poderá ser vaga inicialmente, mas será clareada a medida que o projeto for desenvolvido. O *Product Owner* é responsável pelo financiamento do projeto, visualizando uma maneira de maximizar o retorno do investimento. Além disso, o *Product Owner* deve formular um plano para executá-lo, plano este que inclui a *Product Backlog*.

Todo o trabalho é feito em etapas rápidas (***Sprints***). Cada *Sprint* é uma iteração de trinta dias consecutivos que é iniciada com uma reunião (***Sprint planning***), a qual traçará as metas e o planejamento do projeto. Nessas reuniões, o *Product Owner* e o *Team* se reúnem para decidir o que será feito durante esse *Sprint*. A partir daí é selecionado um *Product Backlog* de maior prioridade para que o *Product Owner* explique ao *Team* o que ele deseja obter. O *Team* então fala ao *Product Owner* quanto do esperado acredita-se que pode ser feito até o final do *Sprint*.

A *Sprint planning* não deve durar mais do que oito horas, sendo dividida em duas partes de quatro horas. As primeiras quatro horas são utilizadas para que o *Product Owner* apresente as *Product Backlogs* de maior prioridade, a partir daí, o *Team* negocia com o *Product Owner* e seleciona quantos *Products Backlogs* achar que pode transformar em funcionalidade até o final do *Sprint*. Na segunda metade da reunião, o *Team* planeja o *Sprint*. Dessa forma, as tarefas são traçadas e colocadas em uma ***Sprint Backlog***.

Todos os dias, o *Team* se reúne por quinze minutos (chamado de ***Daily Scrum***), onde, todos os membros do *Team* respondem três perguntas:

1. o que você fez desde o último *Daily Scrum*?

2. o que planeja fazer até o próximo *Daily Scrum*?

3. quais os impedimentos e obstáculos que apareceram neste *Sprint* e neste Projeto?

O propósito da reunião diária é sincronizar os membros do *Team*.

Ao final do *Sprint* é feita uma reunião informal (***Sprint review***), com duração média quatro horas. Nesta reunião o *Team* mostra ao *Product Owner* (*e os stakeholders envolvidos*), o que foi desenvolvido durante o *Sprint*. A idéia é decidir de forma colaborativa, o que o *Team* fará a seguir. Após o *Sprint review*, antes da *Sprint planning*, o *Scrum Master* faz uma nova reunião

(***Sprint retrospective***), onde ele motiva o *Team* a revisar as práticas do processo, para estarem mais preparados para o próximo *Sprint*.

Juntas, *Sprint planning*, *Daily Scrum*, *Sprint review* e *Sprint retrospective* constituem a inspeção empírica e as práticas de adaptação do Scrum. Na Figura 2.5 está uma visão geral do Scrum.

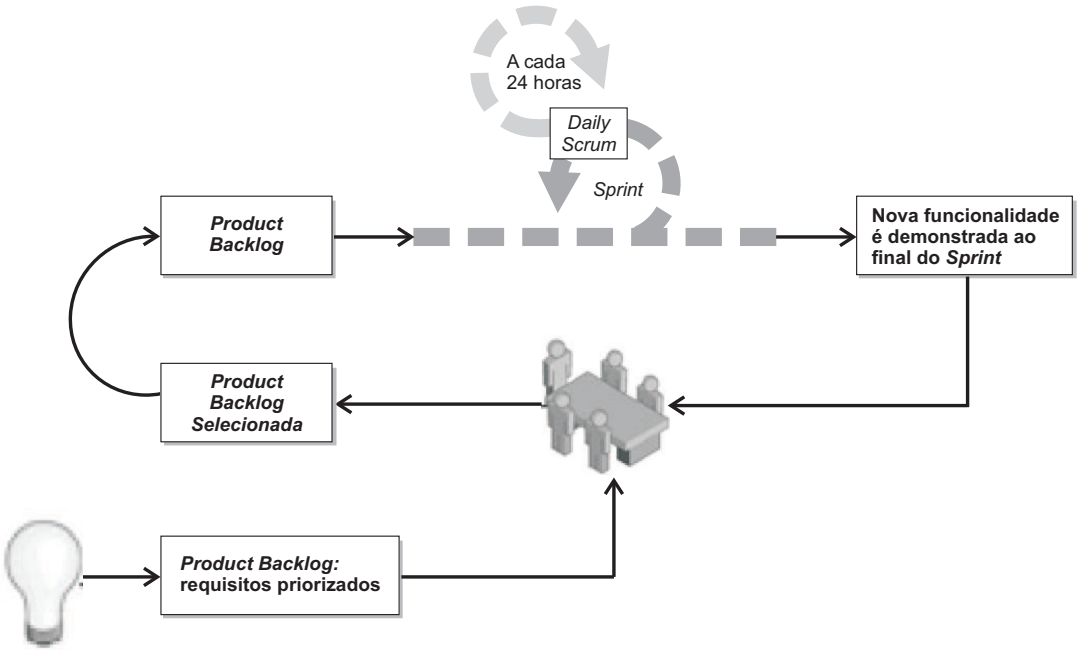

Figura 2.5: Visão geral do Scrum. Adaptado de (Schwaber, 2004)

Artefatos do Scrum

O *Scrum* possui dois artefatos principais, o *Product Backlog* e o *Sprint Backlog*:

- ***Product Backlog*** - os requisitos do sistema ou produto, a ser desenvolvido através de um projeto são listados na *Product Backlog*. O *Product Owner* é o responsável pelo conteúdo, priorização e avaliação desta lista. A *Product Backlog* nunca é completa e é utilizada para planejar as estimativas para os requisitos iniciais. Esta lista é constantemente atualizada, de acordo com a mudança ou identificação de novos requisitos, além disso, enquanto durar o projeto, a *Product Backlog* também existirá. Para visualizar a quantidade de trabalho através do tempo é utilizado um gráfico chamado *brundown chart*. Este gráfico é bastante utilizado para verificar a quantidade de trabalho restante em determinado ponto no tempo, ou seja, é possível visualizar o progresso do projeto.

- ***Sprint Backlog*** - define o trabalho, ou as tarefas, que o *Team* seleciona dentro de uma *Product Backlog*. O *Team* forma uma lista inicial das tarefas que serão feitas na segunda parte da *Sprint planning*. As tarefas são divididas de forma que, cada uma leve entre quatro e dezesseis horas para ser finalizada. Tarefas que levem mais tempo são consideradas

mal definidas. Apenas o *Team* pode modificar a *Sprint Backlog* e ela deve estar bem visível, talvez até em um quadro a ser alterado em tempo real, onde se constitui a quantidade de trabalho que o *Team* planeja fazer durante o *Sprint*.

2.3 Metamodelos e Modelos

De acordo com (Lee et al., 2002), um metamodelo é um modelo que serve para modelar um outro modelo conceitual, ou seja, é uma forma de descrever como um modelo deve ser modelado. Um metamodelo também pode ser utilizado para modelar metadados, assim como configuração de um software ou os metadados dos requisitos, por exemplo.

Metamodelos provêm mecanismos e soluções independentes de plataforma (OMG, 2003a, OMG, 2008a) que especificam:

- a estrutura, sintaxe e semântica para ferramentas, *frameworks* ou tecnologias com metamodelos compartilhados;
- um modelo compartilhado para qualquer espécie de metadado;
- uma padronização de formatos para permitir troca de dados.

Nas próximas Seções serão apresentadas linguagens padrões de metamodelagem e modelagem utilizadas durante o trabalho. Na Seção 2.3.1 é apresentado o *MetaObject Facility*, que se promove como linguagem padrão de construção para construção de metamodelos, da qual fizemos uso. Mais adiante, na Seção 2.3.2, é apresentada a *Unified Modeling Language*.

2.3.1 *MetaObject Facility* (OMG, 2006)

A *Object Management Group* (OMG), através do *MetaObject Facility* (MOF), adotou um padrão que, provê um arcabouço para gerenciamento de metadados. Além disso, o MOF contém um conjunto de serviços para o desenvolvimento e interoperabilidade de sistemas desenvolvidos a partir de Modelos da *Unified Modeling Language* (UML) e que utilizam metadados.

O MOF pode ser utilizado para especificar e integrar famílias de outros metamodelos, bastando utilizar o conceito de modelagem de classes. Como exemplo, na Figura 2.6 é apresentado a definição da UML que é especificada a partir do MOF v2. Nesta figura também é possível verificar que o MOF utiliza de *reflection* para se auto definir.

A especificação do MOF é integrada e reusa o pacote *Core* do metamodelo da UML, provendo um arcabouço mais consistente para a utilização de *Model Driven Architecture* (MDA)[1]. Tal pacote é discutido em mais detalhes na Seção 2.3.2.

[1] www.omg.org/mda

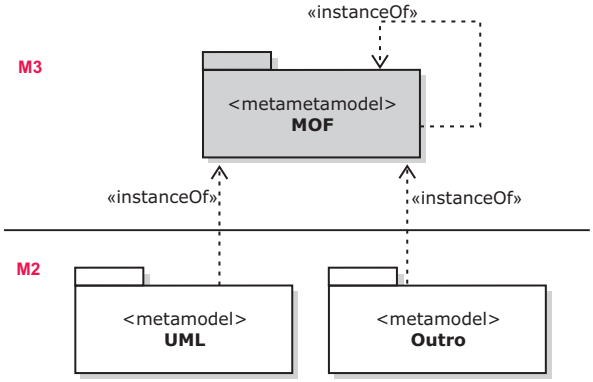

Figura 2.6: Exemplo de metamodelos formados a partir do MOF.

A MDA fundamenta-se na separação entre as especificações de um sistema e os detalhes de sua implementação onde, para isto, existe uma classificação das linguagens de acordo com o nível de abstração em que elas estão. A MDA possui uma arquitetura que contempla quatro camadas (Figura 2.7) onde, cada abstração está situada em uma das camadas que são chamadas de M0, M1, M2 e M3.

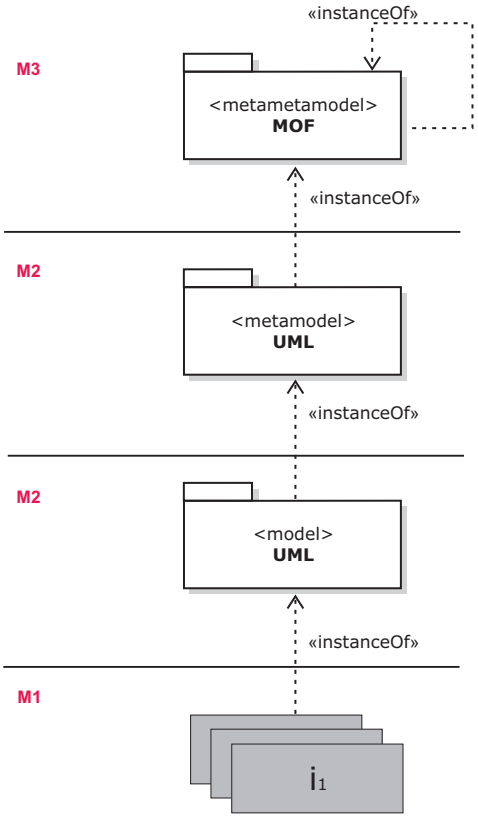

Figura 2.7: Camadas dos níveis de abstrações da MDA. Adaptado de (OMG, 2006)

No nível mais abstrato (M3), encontra-se a meta-metalinguagem, neste caso, a própria linguagem se auto-define e descreve as metalinguagens. O MOF é uma linguagem que possui artifícios que a torna capaz de se auto-definir. Somente a partir desta camada, as metalinguagens (M2) irão descrever as linguagens (M1), ou seja, o MOF irá descrever uma metalinguagem

capaz de definir uma linguagem. Contudo, as meta-metalinguagens, metalinguagens e linguagens estão nas camadas de meta metamodelos, metamodelos e modelos, da arquitetura do MOF, respecitivamente. Na última camada (M0) estão as instâncias do modelo.

Para maior facilidade de compreensão, a Figura 2.8 apresenta um exemplo de utilização dos mesmos níveis de abstração desenvolvidos na Figura 2.7 para a linguagem UML sendo que: na camada M3 está o MOF; na camada M2 está o metamodelo da UML criado a partir do MOF; na camada M1 estão os modelos de usuário utilizados para modelagem UML e definidos por M2; Na camada M0 está um objeto criado a partir da implementação do modelo UML em M1.

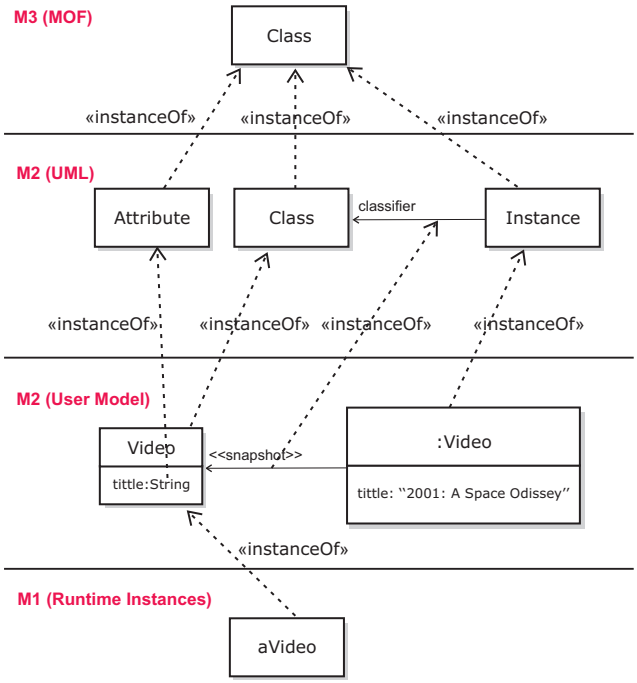

Figura 2.8: Exemplo de utilização das camadas abstrações da MDA utilizando UML. Adaptado de (OMG, 2006)

Modelos baseados no MOF utilizam alguns conceitos para aumentar o reuso através de outros modelos e metamodelos. Desta forma, diferentes tipos de metadados podem ser criados. A especificação do MOF é dividida em dois pacotes principais que sugeram dois Pontos de Conformidade (*Compliance Points*) diferentes: o ***Essential* MOF** (EMOF) e o ***Complete* MOF** (CMOF). Isto foi feito para separar em pacotes as diferentes capacidades (ou propostas de metamodelagem) do MOF.

O MOF contém algumas funcionalidades que são necessárias para a metamodelagem. Tais funcionalidades estão localizadas em pacotes diferentes que estão mesclados através de *package merge* entre os Pontos de Conformidade supracitados. As funcionalidades são:

- ***Reflexion*** (pacote de reflexão), provê habilidade de se auto-definir;
- ***Identifiers*** (pacote de identificadores), provê um identificador único para cada elemento;

- ***Extension*** (pacote de extensão), significa estender elementos do modelo, tanto o nome, quanto os valores existentes.

Ao se definir um metamodelo baseado em MOF pode-se utilizar diferentes níveis de formalismo, para: aumentar o nível de corretividade na descrição dos elementos da linguagem; diminuir ambiguidades e inconsistências; validar o metamodelo utilizando técnicas complementares; e melhorar a legibilidade;

Como é bastante comum para especificação de linguagens, primeiramente pode-se definir a sintaxe da linguagem, para depois acrescentar suas semanticas (*static semantics* e *dynamic semantics*). Nesse sentido, a sintaxe define os construtores da linguagem e como estes podem ser construídos a partir de outros construtores. Já a semântica é definida da seguinte forma: (i) *static semantics*, define como uma instâncias de construtorres devem estar conectadas para que exista algum sentido e geralmente são construídas a partir de regras de boa formação (*well-formed rules*) (ii) *dynamic semantics*, define o significado de um construtor desde que este esteja em seu estado correto (*well-formed*).

Diante disso, o MOF e seus metamodelos podem ser dotados de sintaxe e semântica. Além disso, foi definida uma notação que funciona de forma idependente, sendo uma linguagem gráfica. A partir do mapeamento entre a notação para uma linguagem gráfica, definindo uma nova sintaxe ((*i.e. abstract syntax*)) pode-se obter uma sintaxe concreta (*concrete syntax*).

Embora existam esses níveis de formalismos, a descrição dos modelos e metamodelos formados a partir do MOF não possuem uma especificação totalmente formal.

2.3.2 *Unified Modeling Language* (OMG, 2003b)

Conforme (Rumbaugh et al., 2004), a *Unified Modeling Language* (UML) é uma linguagem padrão para projetar, visualizar, especificar, construir e documentar um Produto de Software (PS).

O escopo da linguagem UML é muito abrangente e diversificado, cobrindo vários domínios em aplicações bem diferentes. Nem sempre toda esta capacidade de modelagem é utilizada para todos os domínios. Baseado neste contexto, a UML é dividida em diversos módulos que, podem ser selecionados de acordo com necessidade de utilização da linguagem. Sendo assim, a UML v2 possui duas especificações que são complementares:

- **Infrastructure** (OMG, 2007b), define os alicerces estruturais.
- **Superstructure** (OMG, 2007c), define as estruturas necessárias para interação em nível de usuário.

Na definição da UML existe um vocabulário e regras específicas que aumentam a facilidade de comunicação. Isto foi feito por que a utilização de uma linguagem de modelagem é feita

para aumentar o entendimento sobre o sistema. No entanto, apenas um único modelo não foi suficiente para representar todo um sistema, sendo necessária a existência de vários modelos que estejam interconectados. Portanto, a UML permite a utilização de diferentes visões de um mesmo sistema.

A UML dispõe de contêineres de abstrações visualizáveis na forma de diagramas. Os diagramas servem para representar um mesmo sistema ou software sobre diversas perspectivas. Para que isto seja possível, a UML dispõe de treze diferentes tipos de diagramas: Diagrama de Classes; Diagrama de Objetos; Diagrama de Componentes; Diagrama de Casos de Uso; Diagrama de Seqüência; Diagrama de Colaboração; Diagrama de Estados; Diagrama de Atividades; Diagrama de Implantação; Diagrama de Pacotes; Diagrama de Tempo; Diagrama de Interação; e Diagrama de Estrutura.

Como em qualquer linguagem, a UML possui uma série de regras para especificar como deve ser um modelo bem formado (*well-formed*). Estes tipos de modelos precisam ser semanticamente auto-contidos e estar em harmonia com modelos relacionados. Tais regras são tanto sintáticas quanto semânticas a fim de definir:

- **nomes**, como devem ser os nomes das abstrações, dos relacionamentos e dos diagramas;
- **escopo**, o contexto em que estão os nomes, garantido sentido;
- **visibilidade**, como os nomes podem ser vistos e utilizados por outros;
- **integridade**, como as abstrações devem ser relacionadas;
- **execução**, execução ou simulação de um modelo dinâmico.

Além disso, para uma melhor organização e padronização, a UML possui quatro mecanismos que a fazem consistente:

- ***Specification***, provê um pano de fundo que garante valor semântico para as construções feitas graficamente;
- ***Adornments***, cada elemento da UML possui uma notação gráfica única, provendo representação visual;
- ***Common divisions***, na modelagem de sistemas orientados a objetos, o mundo sempre é dividido em diferentes caminhos para se chegar a um mesmo destino;
- ***Extensibility mechanisms***, para que a UML pudesse ser utilizada, representando um número maior de domínios específicos, foram definidos mecanismos de extensão:
  - *stereotypes*, estende o vocabulário da UML, permitindo a criação de novos blocos de construção para resolver um problema específico;

– *tagged values*, estende as propriedades de um *stereotype*, permitindo criar novas informações;
– *constraints*, estende o valor semântico dos blocos de construção da UML, permitindo a criação de novas regras ou a modificação de uma já existente.

Diante da possibilidade de restringir a UML para determinado domínio, foram criadas duas estratégias de extensão (OMG, 2003b). A primeira, conhecida como *lightweight built-in extension*, utiliza um mecanismo de extensão chamado *profiles mechanism*, baseando-se em *stereotypes* e *tagged values*. Conforme especificado pela MDA, elementos de um modelo independente de plataforma (*platform specific model* - PSM) são marcados com estereótipos capazes de transformá-lo em um modelo específico de plataforma (*platform specific model* - PSM) (OMG, 2003a). A segunda estratégia é chamada de *heavyweight extensibility mechanism*, sendo o seu principal objetivo estender a UML adicionando novas metaclasses e outros metaconstrutores (OMG, 2006). A seguir estão descritas a duas especificações da UML: UML *Infrastructure* e UML *Superstructure*.

UML *Infrastructure* (OMG, 2007b)

O metamodelo da UML é uma representação de linguagem dividida em diversos módulos, os quais constróem o pacote UML *Core*, também denominado por *Infrastructure Library*. O principal objetivo deste pacote é ser reutilizado por diversos outros metamodelos como base para metamodelagem. Como podemos ver na Figura 2.9, além de outras abordagens, o próprio MOF utiliza este pacote para definir sua família de metamodelos.

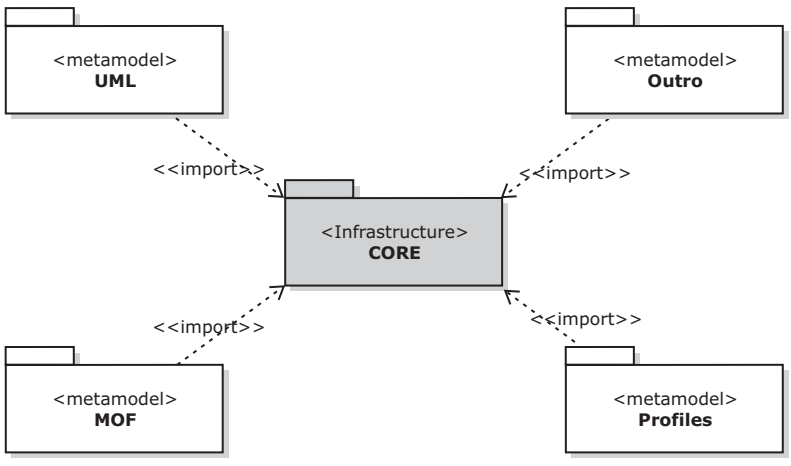

Figura 2.9: Reutilização do pacote UML *Core*

No entanto, o excesso de flexibilidade acaba por permitir a incompatibilidade entre duas ferramentas para modelagem UML distintas, que suportarem diferentes subconjuntos da linguagem. Conseqüentemente, há algumas regras que exigem um balanço entre os módulos e a facilidade de troca de dados. Entretanto, como a flexibilidade precisa ser mantida, UML provê o conceito de *language units*.

Uma *language unit* consiste em uma coleção de conceitos de modelagem bem definidos que provêm o poder de representar aspectos de um sistema de acordo com um paradigma ou formalismo pré-determinado e escolhido. Isto significa que um usuário somente precisa conhecer o subconjunto que julgar necessário de acordo com os modelos que irá utilizar. Contudo, os grupos providos pelos *language units* e os incrementos que se constituem, servem para simplificar a definição das regras de conformidade (*compliance rules*) da UML.

Isto é, o conjunto de conceitos do modelo da UML é particionado em camadas horizontais, chamadas de níveis de conformidade (*compliance levels*). Cada ponto em que seja necessária alguma conformidade será chamado de ponto de complacência ou conformidade (*compliance point*). Para facilitar a troca de modelos existem apenas dois níveis de conformidade definidos para a UML *Infrastructure*:

- **Nível 0 (L0)**. Este nível contém apenas uma *language unit* que provê os tipos de estruturas (todas *class-based*) necessárias para modelar as linguagens OO mais populares. Portanto, existe um primeiro nível (nível de entrada) que permite a modelagem. Contudo, esta camada serve como base para a existência de interoperabilidade entre diferentes categorias de ferramentas para modelagem.

- ***Metamodel Constructs* (LM)**. Este nível adiciona alguns *language unit* extras para obter estruturas (também *class-based*) mais avançadas. Estas estruturas são utilizadas para a criação de metamodelos, assim como da própria UML (utilizando CMOF).

Para que os níveis de conformidade sejam suportados, existem mecanismos que são utilizados a partir da especificação da UML. Estes mecanismos permitem que os conceitos de modelagem definidos em um nível determinado possam ser estendidos sem que percam suas características. Para que os mecanismos funcionem, é necessário que os níveis de conformidade estejam em um mesmo *namespace*. Por esta razão, todos os níveis de conformidade estão definidos como extensões do pacote *core* da UML. Este pacote define um *namespace* comum para todos os níveis de conformidades. O nível 0 (L0) é definido pelo metamodelo da Figura 2.10.

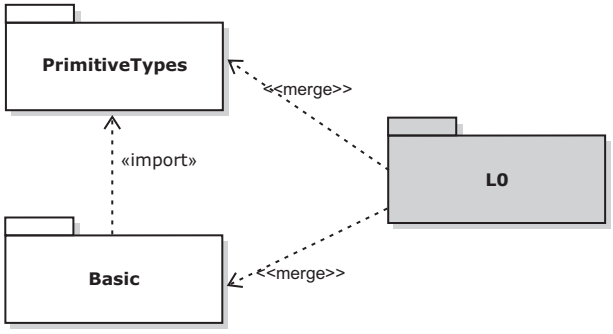

Figura 2.10: Diagrama de pacotes do nível 0 - L0. Adaptado de (OMG, 2007b)

No modelo da Figura 2.10, UML é, originalmente, um pacote vazio que simplesmente estende (utilizado *merge*) o conteúdo do pacote *Basic* do UML *Infrastructure*. Este pacote possui os conceitos mais elementares tais como *Class*, *Package*, *DataType*, *Operation*.

No nível LM, o conteúdo do pacote UML (que inclue os pacotes e seus conteúdos mesclados do L0), mesclam (utilizam *package merge*) o pacote *Constructs*, Figura 2.11. Como pode ser visto, o nível LM não se mescla (*merge*) explicitamente com o pacote *Basic*, pois essse já esta incorporado no pacote *Constructs*.

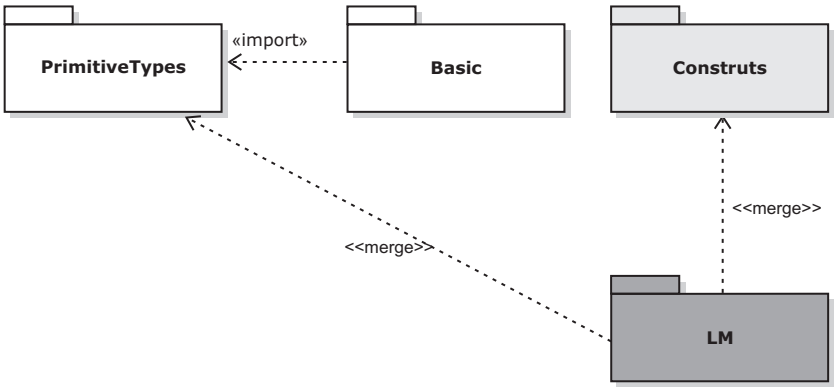

Figura 2.11: Diagrama de pacotes do LM. Adaptado de (OMG, 2007b)

Arquitetura da UML

Toda a especificação da UML é definida utilizando o metamodelo desenvolvido pela OMG. Este metamodelo ajusta, ou melhor, acomoda as especificações formais e técnicas da UML. Embora esta o metamodelo não apresente todas as formalidades da especificação rigorosamente, ele oferece vantagens por ser mais intuitivo.

A arquitetura da UML foi desenvolvida baseada em alguns princípios e padrões, sendo eles:

- ***Modularity*** - O princípio básico de alta coesão e baixo acoplamento. Este padrão é aplicado ao se determinar grupos e separá-los em pacotes, agrupando as características em *metaclasses*;

- ***Layering*** - *Layering* é aplicado no metamodelo da UML de duas formas:

  1. A estrutura de pacotes é dividida em camadas, separando as principais construções do metamodelo das construções que as utilizam.
  2. É aplicado um padrão de quatro camadas arquiteturais do metamodelo a fim de separá-lo (especialmente no que se refere a instanciação) através de camadas de abstração.

- ***Partitioning*** - Utilizado para organizar os conceitos de uma mesma área, dentro da mesma camada. No caso da biblioteca da *Infrastructure*, é utilizado um particionamento com

granulação bastante fina, afim de provêr maior flexibilidade. No caso do metamodelo da UML, o particionamento é pouco granular, aumentando a coesão entre os elementos internos de um pacote e baixando o acoplamento entre eles.

- ***Extensibility*** - A UML pode ser extendida de duas formas diferentes:

  1. uma pequena variação da linguagem UML (dialeto) pode ser definida utilizando *Profiles* para customizá-la, tornando-a particular em algum domínio específico.
  2. Uma nova linguagem relacionada com a UML pode ser especificada utilizando parte do pacote *Infra-Struture Library*, aumentando-a com as devidas *metaclasses* e *metarelationships*.

  No primeiro caso é definido um novo dialeto da UML, no segundo, é definido um novo membro da família de linguagens baseadas na UML.

- ***Reuse*** - Uma biblioteca de metamodelo flexível e com alta granularidade é fornecida desde que seja reutilizada para definir o metamodelo da UML, assim como outros metamodelos relacionados (como o *Meta Object Facility*).

UML *Superstructure* (OMG, 2007c)

A UML *Superstructure* (OMG, 2007c) é a segunda parte de duas especificações complementares. A primeira delas (*Infrastructure*) (OMG, 2007b) é o alicerce arquitetural para a *Superstructure*. Essas duas especificações se constituem na especificação completa da UML 2. A UML *Superstructure* não está dentro do escopo deste trabalho.

2.4 SPEM v2 (OMG, 2008b)

O *Software and System Process Engineering Metamodel* (SPEM v2) (OMG, 2008b) é um metamodelo para engenharia de PDSs assim como um arcabouço conceitual, provendo os conceitos necessários para modelar, documentar, apresentar, gerenciar, inter-mutabilidade e execução de métodos e processos de software.

2.4.1 Estrutura do Processo x Conteúdo

O SPEM v2 separa a engenharia do processo em dois momentos principais: criação de uma biblioteca (*Method Library*), que armazenará o conteúdo (*Method Content*) e sua utilização em um PDS (*Method Structure*), Figura 2.12. *Method content* é tudo aquilo que provê explicações

passo-a-passo, descrevendo como os objetivos serão alcançados independente ciclo de desenvolvimento. Nesse sentido, um PDS fará uso dos elementos que formam o *Method Content* configurando-os para uma execução específica.

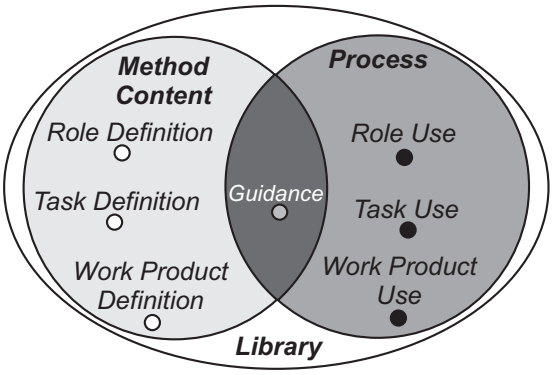

Figura 2.12: Exemplo de divisão entre *Method Content* e *Process Structure*

Desta forma, o SPEM v2 descreve estruturas necessárias para expressar formalmente o *Method Content* e sua utilização em um *Process Structure*, sendo definido através de um metamodelo e um UML *Profile*, permitindo o uso de estereótipos que estendem a UML, restringindo o domínio de PDS. Além disso, seu metamodelo foi definido através do MOF v2, que é uma meta linguagem de modelagem padrão definida pela OMG, neste caso, um meta-metamodelo.

2.4.2 Definição do metamodelo

Para a definição do SPEM v2 foi utilizado o MOF. A partir do MOF é possível definir novas metalinguagens, criando novos construtores. Da mesma forma, a UML também foi definida com a utilização de MOF. Mas, partindo desse princípio, quem define o MOF? Como o MOF possui operações de *reflection*, ele pode ser auto-definir, criando o seu próprio metamodelo. Este conceito é chamado de *bootstrapping*, onde o próprio MOF utiliza a UML, linguagem que ele próprio define, como ponto de início.

A Figura 2.13 apresenta o uso do MOF v2 e da UML v2 para a modelagem e utilização do SPEM v2. Como pode ser observado, a Figura define dois instantes: **a** e **b**.

Em **a** está a modelagem e definição do SPEM v2 mostrado em uma arquitetura de três camadas, utilizando o MOF. Na camada M3 está o MOF v2 que é instanciado pela camada M2 para definir o metamodelo da UML v2 e do SPEM v2, mesma camada em que o metamodelo do SPEM v2 estende o metamodelo da UML v2. Finalmente, na camada M1 está uma instância de M2, mostrando a definição da *Method Library* a partir de dois pontos diferentes, metamodelo do SPEM v2 ou através do *lightweight built-in extension* da UML com utilização de UML *Profiles* definidos especialmente para o este domínio.

Em **b** é posto um exemplo de utilização do SPEM v2 visto em uma arquitetura de quatro

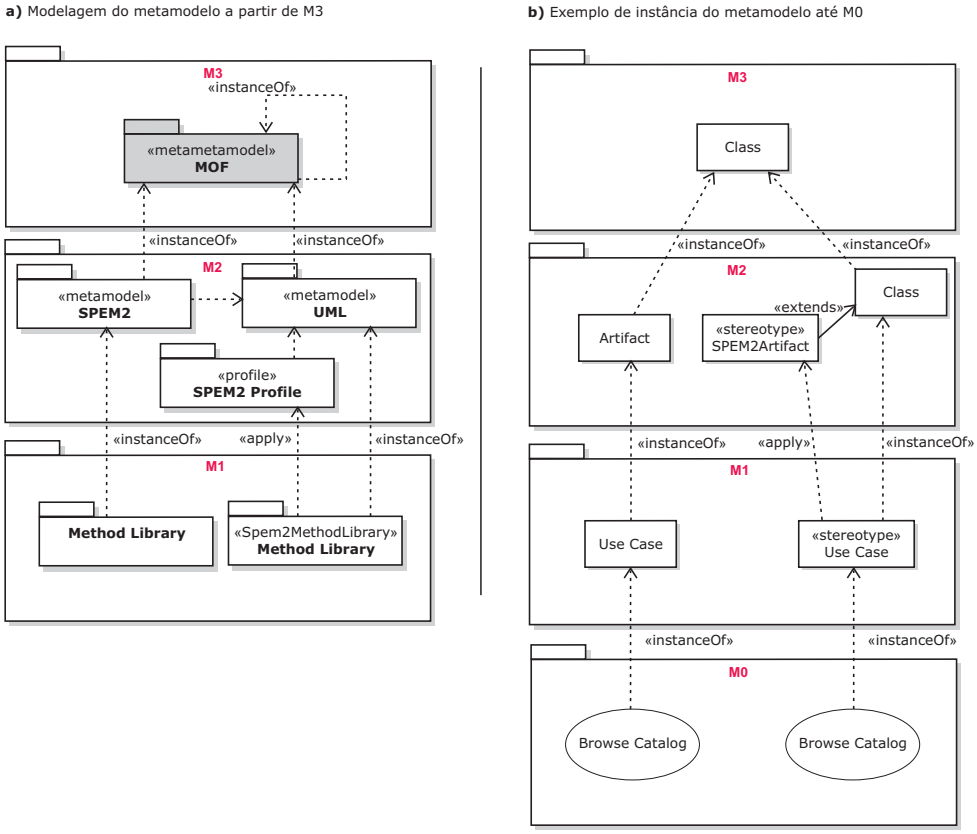

Figura 2.13: Camadas de modelagem e instância do SPEM v2.

camadas. Em M3 temos a definição de *classe* (*metaclass*), dada pelo MOF. Em M2, está um exemplo de instância de classe definindo um *Artefato* no caso do SPEM v2, ou o uso de UML *Profiles*, estendendo uma classe do metamodelo da UML. Já em M1 está a instância do metamodelo e do UML *Profile* do SPEM v2. Para este exemplo foi utilizado um Caso de Uso, que é uma instância de artefato. Para utilizar o metamodelo, basta instanciar um Artefato, mas para utilizar o UML *Profile* é necessário instanciar uma classe da UML e aplicar a ela o estereótipo de *Artefato*. Por fim, na camada M0 são definidos catálogos, que nada mais são do que conjuntos de instâncias de Casos de Uso.

2.4.3 Definição dos Pacotes

No tocante ao metamodelo, o SPEM v2 foi estruturado em sete pacotes principais, Figura 2.14, os quais são compostos aplicando-se o mecanismo *package merge* da UML.

Os pacotes apresentados na Figura 2.14 são abaixo descritos:

- ***Core*** - pacote que contém as classes e abstrações para a construção da base do metamodelo.

- ***Process Structure*** - pacote que define o metamodelo base modelagem de processos e que

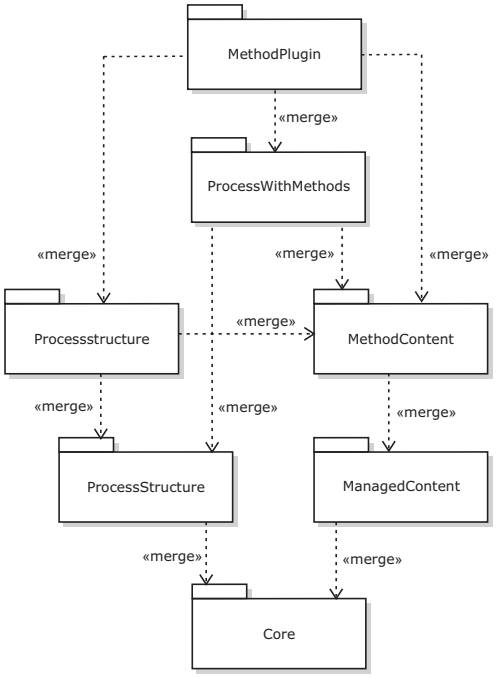

Figura 2.14: Estrutura do metamodelo do SPEM v2.

representa sua estrutura estática.

- ***Process Behaviour*** - pacote que permite uma extensão do metamodelo do SPEM v2 para que a execução de um processo possa ser acompanhada.

- ***Managed Content*** - pacote que adiciona conceitos para documentação e descrição textual.

- ***Method Content*** - pacote que permite que os usuários do SPEM v2 criem uma biblioteca com conhecimento reutilizável e independente de processos para uso posterior.

- ***Process with Methods*** - este pacote define novos conceitos e altera outros conceitos já existentes nos pacotes anteriores para integrar processos definidos pelo pacote *Process Structure* com seus conteúdos, definidos pelo pacote *Method Content*.

- ***Method Plugin*** - pacote que define os conceitos necessários para criar, gerenciar e manter bibliotecas e PDSs.

2.4.4 Pontos de Conformidade

Segundo a sua especificação, o SPEM v2 é definido a partir de três pontos de conformidade: ***SPEM Complete***, ***SPEM Process with Behaviour and Content***, e ***SPEM Method Content***, descritos a seguir.

SPEM Complete

O SPEM *Complete* compreende todos os sete pacotes do metamodelo do SPEM v2, descritos em 2.4.3. Este Ponto de Conformidade é recomendado para aqueles que desejam utilizar todo o metamodelo e suas capacidades. Na Figura 2.15 é apresentado um espaço de nomes chamado *SPEM2*, o qual utiliza-se de *package merge* para compor: (i) o LM, um nível de conformidade definido pela UML 2 *Infrastructure Library*, descrita em 2.3.2; (ii) o UML *Profiles*; e (iii) os pacotes *Method Plugin*, *Process Behavior*, compondo através de transitividade todos os outros pacotes do SPEM v2.

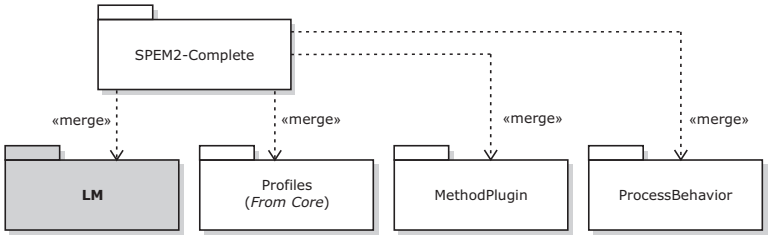

Figura 2.15: Ponto de Conformidade - *SPEM Complete*

Audiência: Fornecedores em larga escala de ferramentas para bibliotecas de PDSs e conteúdos.

SPEM Process with Behavior and Content

O SPEM *Process with Behavior and Content* compreende quatro pacotes do metamodelo do SPEM v2, descritos em 2.4.3. Na Figura 2.16 é apresentado um espaço de nomes chamado *SPEM2-Process-Behavior-Content*, o qual utiliza-se de *package merge* para compor: (i) o LM, um nível de conformidade definido pela UML 2 *Infrastructure Library*, descrita em 2.3.2; e (ii) os pacotes *Managed Content* e *Process Behavior* do SPEM v2, que por transitividade compõem também os pacotes *Process Structure* e *Core*.

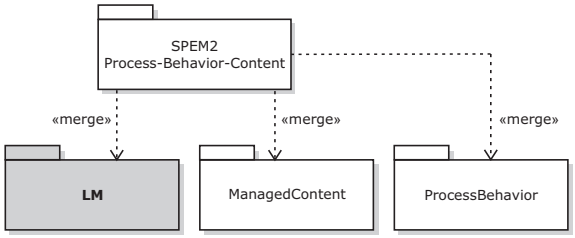

Figura 2.16: Ponto de Conformidade - *SPEM Process with Behavior and Content*

Audiência: Fornecedores de ferramentas com foco em compatibilidade com o SPEM v1.x e modelagem.

SPEM *Method Content*

O SPEM *Complete* compreende todos os três pacotes do metamodelo do SPEM v2, descritos em 2.4.3. Este Ponto de Conformidade é recomendado para aqueles que desejam utilizar todo o metamodelo e suas capacidades. Na Figura 2.17 é apresentado um espaço de nomes chamado *SPEM2-Method-Content*, o qual utiliza-se de *package merge* para compor: (i) o LM, um nível de conformidade definido pela UML 2 *Infrastructure Library*, descrita em 2.3.2; e (ii) o pacote *Method Content*, compondo através de transitividade os pacotes *Managed Content* e *Core* do SPEM v2.

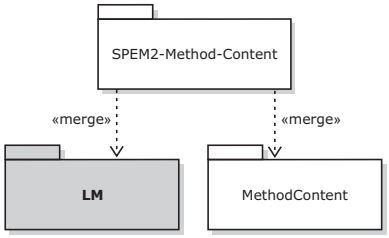

Figura 2.17: Ponto de Conformidade - *SPEM Method Content*

Audiência: Fornecedores de bases de conhecimento organizacional e responsáveis por documentação.

3 Trabalhos Relacionados

Nesta seção apresentamos os principais trabalhos relacionados à construção de ASs. Nosso objetivo foi analisar a literatura atual e observar se autoria de AS tem sido objeto de pesquisa na comunidade da Engenharia de Software. Porém, chegou-se a conclusão que os trabalhos mais recentes não possuem foco em Autoria de ASs, divergindo para outros fins. Portanto, como o principal interesse desta pesquisa está relacionado à identificação das necessidades e soluções para a especificação e construção de uma Autoria de ASs, foram analisados os trabalhos que mais contribuíram com nossa abordagem. Tais trabalhos são divididos nas Seções 3.1, que apresenta trabalhos relacionados no tocante a edição, classificação e organização de artefatos de software e 3.2, que demonstra a comparação com trabalhos que procuraram utilizar metamodelagem como meio da abordagem.

3.1 Comparação entre trabalhos sobre autoria de ASs

Nesta Seção, a análise comparativa das abordagens é baseada nos critérios de **Autoria**, **Representação**, **Capacidades** e **Avaliação** ou **Testes**. Além disso, cada abordagem será apresentada conforme sua descrição e relação com o trabalho aqui proposto.

No que se constitui a **Autoria**, analisamos como as abordagens atacam os problemas em autoria de ASs em si. Desta forma, definidos os atributos:

- (A) **Paradigma de construção**, como a abordagem trata os artefatos? artefatos são vistos como uma união de fragmentos de informação? artefatos possuem estrutura lógica?
- (B) **Guia de construção**, existe algum diagrama com fluxos? se sim, quais são os passos necessárias para se realizar a construção ou classificação dos artefatos?
- (C) **Escopo da abordagem**, a abordagem permite a construção de que tipos de artefatos?
- (D) **Separação de Conteúdo**, há uma clara divisão entre a estrutura do processo (*Process Structure*) e o seu conteúdo (*Method Content*)?

Em relação a **Representação**, verificamos como as abordagens atacam os problemas em autoria de ASs no tocante a utilização ou criação de uma linguagem própria. Baseados em (OMG, 2007b), analisamos os atributos:

1. **Semântica Estática**, existe descrição semântica sobre a utilização os contrutores?

2. **Semântica Dinâmica**, é definido algum modo de identificar se os construtores possuem algum sentido? Existe alguma descrição dos construtores?

3. **Linguagem Abstrata**, a linguagem possui alguma notação para faciliar seu uso? Existe alguma forma visual de representação gráfica?

4. **Linguagem Concreta**, a linguagem pode ser representada formalmente? Existe alguma representação em XML ou XMI por exemplo?

5. **Regras de boa formação**, a linguagem traz consigo regras que definem a boa construção dos ASs de modo formal ou em linguagem natural?

No tocante a análise das **capacidades** existentes na autoria de ASs, verificamos conceitos ligados a categorização, manuseio e extensibilidade de cada uma das abordagens. Os atributos utilizados foram:

- **Extensibilidade**, a abordagem admite ou possui pontos de extensibilidade, para posterior melhoria ou acréscimo de funcionalidades?

- **Classificação**, a abordagem permite que os artefatos sejam classificados ou categorizados de acordo com seus conteúdos?

- **Controle de Versões**, na abordagem é possível criar diferentes versões de um mesmo artefato? A abordagem permite o retorno a uma versão anterior de um artefato sem perder as alterações mais atuais?

- **Controle de Maturidade**, a abordagem dispõe algum mecanismo que permita definir o nível de maturidade de artefatos?

Finalmente, verificamos como as abordagens são **validadas**, **verificadas** ou quais **avaliações** foram feitas pelos autores dos trabalhos analisados. Partindo deste princípio, tomamos por base os atributos:

- (i) **Instanciação**, a abordagem foi instanciada através de exemplos? Foram feitas comparações analíticas com base na utilização da abordagem? Existe alguma validação científica, assim como estudo de caso real ou experimentos?

- (ii) **Implementação**, a abordagem foi implementada, existe alguma implementação padrão para utilização e testes?

- (iii) **Ferramental ou Protótipo**, foi desenvolvida alguma implementação específica? A abordagem foi consolidada em forma de ferramenta? Existe algum protótipo que automatize a utilização da proposta?

3.1.1 Abordagem Akpotsui et al., 1992

Esta abordagem considera que documentos são estruturas lógicas, também podem ser vistos como árvores estruturadas, feitas a partir de elementos que tornam a estrutura capaz de representar organização.

Descrição: Através de um modelo de estrutura, cada documento possui uma estrutura bem específica, derivada da estrutura genérica. Como cada nó possui um tipo, para a estrutura genérica existe um conjunto de tipos *T* construídos a partir de um conjunto básico de tipos β e um conjunto de construtores *C*: β = {Texto, Imagem, Gráfico}; e *C* = {Grupo Ordenado, Grupo Não Ordenado, Lista, *Choice*, Identidades}.

Autoria: esta abordagem é bastante complexa e trabalha com o paradigma de documentos com boa formação, sendo esta logicamente estruturada. Infelizmente ela não traz um guia que explique como fazer a autoria e seu escopo é limitado apenas a documentos. Como o foco desta abordagem não está em formação de ASs, ela também não compreende separação entre estrutura e conteúdo de PDSs.

Representação: esta abordagem se constitui de três níveis de representação. Inicialmente é apresentada a descrição semântica dos elementos a serem modelados. Depois disso, é utilizada uma linguagem abstrata simples de visualização em árvore, baseada na linguagem concreta baseada em XML, juntamente com a definição de um DTD, possuindo também regras de boa formação baseadas na teoria de conjuntos.

Capacidades: a abordagem não apresenta nenhuma das capacidades analisadas.

Validação ou Testes: nesta abordagem não foi encontrada nenhuma consideração sobre qualquer avaliação ou testes.

Relação com a abordagem proposta: esta abordagem não considera autoria de ASs, mas a criação e edição de documentos sob uma conceitualização de suas estruturas internas. A abordagem também não apresenta categorização, versionamento ou controle de maturidade, os quais pertencem ao escopo de edição de documentos. Embora possua boa representação, este trabalho não demonstra qualquer preocupação com a criação de pontos de extensão. Além disso, não é apresentada qualquer diferenciação entre a definição, uso e instância do documento.

3.1.2 Abordagem Buchner, 2000

Esta abordagem consiste na construção de um arcabouço para Documentos Compostos (*Compounding Documents*) chamado HotDoc, que permite a construção de documentos que não possuam somente texto, mas sejam flexívels a partir de partes dinâmicas, como tabelas, figuras, diagramas, gráficos, sons e vídeos.

Descrição: Cada documento é dividido em partes de diferentes tipos que podem ser aninha-

das até formarem uma árvore. Desta forma, as partes podem compartilhar mesmo modelo de dados através de ligações entre si, desde que as ligações sejam entre modelos compatíveis.

Autoria: esta abordagem adiciona funcionalidades para a autoria de documentos, permitindo que o autor combine diferentes partes do documento. Seu principal objetivo foi detectar a mudança de paradigma na construção de documentos. Esta abordagem não apresenta um guia que explique como fazer a autoria e seu escopo é limitado apenas a documentos. Além disso, não é apresentada separação entre estrutura e conteúdo de PDSs.

Representação: em nossas análises não detectamos nenhum tipo de representação ou modelo. Entretanto, algumas regras de formação são brevemente comentadas, mas sem nenhuma formalidade ou linguagem específica.

Capacidades: a abordagem não apresenta nenhuma das capacidades analisadas.

Validação ou Testes: esta abordagem consiste em uma ferramenta implementada a partir de um arcabouço para Documentos Compostos. O HotDoc foi implementado utilizando o padrão arquitetural *Model-View-Controller* (MVC) para *VisualWorks Smalltalk*. A ferramenta permite que documentos possam ser construídos a partir da inserção de novas *partes* em um tipo de área de trabalho de usuário (*workspace*). A interface com usuário permite o uso de objetos reposicionáveis e redimensionáveis, juntamente com a funcionalidade *drag and drop*.

Relação com a abordagem proposta: esta abordagem possibilita a criação de documentos baseando-se na montagem do mesmo por estruturas internas, tais documentos conhecidos como documentos compostos (*compounding documents*). Esta abordagem não foi desenvolvida especificamente para autoria de ASs, entretanto, abrange a montagem de documentos, ou seja, determina um escopo menor. Sobre questões de gerenciamento ou controle, não há qualquer referência sobre classificação, maturidade e versionamento. Além disso, nenhuma representação ou conceitualização foi apresentada e embora seja um arcabouço, não foram apresentados pontos de extensão. Finalmente, não foi identificado qualquer suporte para diferenciação entre a definição do documento, seu uso e sua instância.

3.1.3 Abordagem Cattaneo et al., 2000

Esta abordagem visa a permitir que documentos possam ser estruturados com base numa definição de XML.

Descrição: pensando na estruturação de artefatos, os autores definiram um gerenciador de artefatos baseado na *Web* chamado *Labyrinth*. Esta ferramenta permite a definição de artefatos através de *XML Schema* (XSD) .

Autoria: esta abordagem apresenta autoria com estruturação de artefatos e permite que o conteúdo esteja separado da definição. Além disso, esta solução possui escopo em artefatos, mas não contempla nenhuma espécie de guia ou fluxo de autoria.

Representação: a solução abordada no trabalho destes autores apresenta uma definição de

artefatos baseada em XSD e o protótipo foi modelado sobre diagrama Entidade Relacionamento (ER). Entretanto, não há qualquer explicação sobre o XSD e o modelo ER apresentado e não é possível concluir que exista alguma semântica bem definida. As linguagens utilizadas, por si só já possuem sintaxe e notação.

Capacidades: a abordagem não apresenta nenhuma das capacidades analisadas.

Validação ou Testes: para verificação foi desenvolvido um protótipo para *Web* e alguns exemplos de utilização são apresentados. O protótipo apresenta como características navegação e preenchimento de documentos, controle de incosistências e sistema de notificações, que avisa quando um artefato foi modificado.

Relação com a abordagem proposta: conforme os autores, este trabalho possibilita a criação de documentos compostos, baseando-se na montagem dos mesmos a partir de outras estruturas. Assim como em (Buchner, 2000), esta abordagem não foi desenvolvida especificamente para autoria de ASs, e seu escopo se reduz à montagem de documentos. Além disso, não há formas de efetuar classificação, maturidade ou versionamento e nenhum ponto de extensão foi apresentado. Por fim, esta abordagem não diferencia entre definição, uso e instância.

3.1.4 Abordagem Herzner & Hocevar, 1991

Esta abordagem apresenta um Banco de Dados Orientado a Objetos que permite a gerência de documentos.

Descrição: nomeada como *Compound Document access and Management* (CDAM) esta abordagem provê um tratamento de documentos de forma a se tornarem coleções de partes ou componentes, formando uma agregação de pedaços de informações estruturadas com a propriedade de se auto definir.

Autoria: no contexto de autoria, esta abordagem apresenta uma construção de documentos sob o mesmo paradigma apresentado em (Buchner, 2000), mas da mesma forma, não apresenta um guia explicativo para autoria e seu escopo também é sob documentos. Além disso, não é apresentada separação entre estrutura e conteúdo de PDSs.

Representação: nesta abordagem pode ser vista uma descrição semântica dos conceitos utilizados para autoria de documentos, utilizando linguagem natural. Nenhuma outra forma de representação foi encontrada.

Capacidades: a abordagem não apresenta nenhuma das capacidades analisadas.

Validação ou Testes: não é definida nenhuma validação mas sim uma verificação através do uso de um protótipo implementado sobre um sistema de nome CDAM. Tal sistema foi desenvolvimento seguindo os seguintes passos: (i) inicialmente foi projetado um modelo, (ii) este modelo foi mapeado para um ambiente de banco de dados orientado a objetos e (iii) deste ambiente, foi criado um banco de dados relacional; e que consiste em um Bancos de Dados Relacional, um Sistema de Banco de Dados Orientado a Objetos, uma API geral de controle e

uma API para suporte a métodos. O Banco de dados Orientado a Objetos possui as seguintes tabelas que contém todas as informações: *STRUCTURE_CLASS*: representa as partes; *STRUCTURE_INSTANCE*: representa as instancias; *TYPE_CLASSES*: representa os tipos de dados pré-definidos; *DATA*: representa os tipos de dados elementares.

Relação com a abordagem proposta: conforme os autores, este trabalho possibilita a criação de documentos estruturados, baseando-se na montagem dos mesmos a partir de outras estruturas definidas num documento XSD. Assim como em alguns trabalhos anteriores, esta abordagem não foi desenvolvida especificamente para autoria de ASs, e seu escopo se reduz à montagem de documentos. Além disso, não há formas de efetuar classificação, maturidade ou versionamento e nenhum ponto de extensão foi apresentado.

3.1.5 Abordagem Laitinen, 1992

Esta abordagem possui um foco em classificação de artefatos produzidos em um projeto, em uma tentativa de controlá-los e gerenciá-los uniformemente.

Descrição: o autor dessa proposta sugere uma nomenclatura com base em uma classificação gerencial d documentação, que segue: *Software Description*, *Utilization Documents*, *Development Plans*, *Quality Control Documents* and *Administrative Documents*.

Autoria: diferente das abordagens anteriores, este trabalho não consiste em autoria de artefatos. Seu foco é apenas em documentos no paradigma padrão. Entretanto, é apresentado um guia resumido para utilização da abordagem, a partir de um arcabouço de classificação.

Representação: nesta abordagem existe uma breve descrição semântica dos conceitos utilizados para classificação dos documentos, utilizando linguagem natural e uma tabela demonstrativa. Nenhuma outra forma de representação foi encontrada.

Capacidades: dentre as capacidades analisadas, esta abordagem demonstra classificação de artefatos do tipo documentos.

Validação ou Testes: o trabalho não apresenta nenhuma implementação, teste ou aplicação.

Relação com a abordagem proposta: como o foco desta abordagem é apenas classificar documentos produzidos durante a execução de um PDS, a única relação que pode-se fazer é conforme classificação. A abordagem proposta possui classificação para um escopo maior, no caso artefatos, enquanto esta abordagem apresenta apenas classificação para documentos. Por fim, não é apresentada qualquer validação, verificação ou teste.

3.1.6 Abordagem Tilley & Müller, 1991

Esta abordagem se constitui em um facilitador para a documentação de artefatos do tipo código-fonte. Segundo o próprio autor, sem uma boa documentação, o único meio de entender

o código-fonte é analisando-o e mesmo que o código esteja bem documentado, muitas vezes está fora do padrão utilizado pela companhia ou incompleto.

Descrição: O autor sugere uma ferramenta que age como um facilitador em hipertexto que permite anotar documentos de forma não intrusiva, o INFO. A documentação e todos os comentários pertinentes são persistidos em um arquivo texto, enquanto isso são utilizadas *tags* que correspondem as chamadas das informações contidas no código-fonte.

Autoria: o contexto desta abordagem é bastante específico, servindo apenas para determinar documentação de código-fonte. Portanto, esta abordagem não apresenta nenhuma característica para autoria de artefatos. Entretanto, o próprio autor demonstra como editar os documentos para acrescentar as *tags* previstas pela abordagem.

Representação: esta abordagem apresenta uma linguagem de anotações baseada em *tags*, devendo possuir sintaxe e semântica. Entretanto, existem apenas alguns exemplos de utilização e não é possível inferir que a semântica, embora exista, esteja bem definida. Mesmo assim, algumas regras de formação são especificadas.

Capacidades: a abordagem não apresenta nenhuma das capacidades analisadas.

Validação ou Testes: houve a implementação do conceito e o próprio autor demontra exemplos de utilização e aplicação da abordagem, conforme dados sobre a experiência do usuário. No tocante a ferramenta, o autor a julga desnecessária.

Relação com a abordagem proposta: esta abordagem apresenta um diferencial em relação as propostas anteriores, pois trabalha com artefato do tipo código-fonte. Embora o foco não seja estrutura, o autor apresenta uma maneira de documentação diferente das abordagens utilizadas na indústria. Além disso, esta abordagem não se trata de autoria, mas sim da criação de uma metalinguagem. Nesse contexto, não foi possível encontrar os níveis esperados de formalismo no tocante a descrição e especificação da linguagem. Por fim, não é possível concluir que exista alguma ferramenta específica ou protótipo para a utilização da abordagem.

3.1.7 Abordagem Visconti & Cook, 1993

Esta abordagem identifica um meio de melhorar processos de desenvolvimento de software por meio da definição de níveis de maturidade para a documentação.

Descrição: a abordagem dos autores identifica um meio de melhorar o processo e definir quatro níveis de maturidade para a documentação do mesmo: *ad-hoc*, inconsistente, definido e controlado. Cada nível é descrito pelos seguintes campos: nome, descrição simples, palavras-chaves, principais áreas do processo, principais práticas, principais indicadores, principais desafios e principais significados.

Autoria: assim como (Laitinen, 1992), esta abordagem não consiste em autoria de artefatos e seu foco é apenas em documentos. Além disso, é apresentado um guia resumido para utilização da abordagem, a partir de um arcabouço conceitual para definição de níveis de maturidade.

Representação: nesta abordagem é apresentada uma descrição semântica dos termos utilizados para definir os níveis de maturidade dos documentos, utilizando linguagem natural e uma tabela demonstrativa. Nenhuma outra forma de representação foi encontrada.

Capacidades: dentre as capacidades analisadas, esta abordagem demonstra apenas a maturidade de artefatos do documentos.

Validação ou Testes: o trabalho não apresenta nenhuma implementação, teste ou aplicação.

Relação com a abordagem proposta: como o foco desta abordagem é apenas definir níveis de maturidade em documentos produzidos durante a execução de um PDS, única relação que pode-se fazer é conforme maturidade. Esta abordagem proposta define níveis maturidade para um escopo menor, ou seja, apenas para documentos.

3.2 Comparação entre trabalhos sobre metamodelagem

Nesta Seção, a análise comparativa das abordagens é baseada nos critérios de **Tipo de Extensão da UML**, **Níveis de formalismo da Linguagem**, **Modificações feitas no metamodelo UML**, **Tipos de Modelos existentes** e **Validação** ou **Testes**. Além disso, cada abordagem será apresentada conforme sua descrição e relação com o trabalho aqui proposto.

No que se constitui ao **Tipo de Extensão da UML**, analisamos como as abordagens criam seus metamodelos em relação ao mecanismo de extensão da UML, podendo ser do tipo:

- *heavyweight extensibility mechanism*, durante a extensão foram criados novos *constructors*? Houve alteração ou remoção de *constructors* já existentes no metamodelo da UML?

- *lightweight built-in extension*, foi feita a criação de um UML *Profile* com *stereótipos* para expressar algum domínio específico?

Em relação a **Níveis de formalismo da Linguagem**, verificamos como as abordagens formalizam seus metamodelos nos seguintes níveis de linguagem:

- (1) **Semântica Estática**, existe alguma descrição semântica para utilizar os contrutores do metamodelo?

- (2) **Semântica Dinâmica**, é definido algum modo de identificar se os construtores possuem algum sentido? Existe alguma descrição dos construtores?

- (3) **Sintaxe Abstrata**, o metamodelo possui alguma notação para faciliar seu uso? existe alguma forma visual de representação gráfica?

- (4) **Sintaxe Concreta**, a linguagem pode ser representada formalmente? existe alguma representação em XML ou XMI por exemplo?

- (5) **Regras de boa formação**, a linguagem traz consigo regras que definem a boa construção da abordagem? Existem regras para auxiliar o entendimento?

No tocante a *Modificações feitas no metamodelo UML*, analisamos o que foi feito para que o metamodelo estendesse a UML a partir do MOF, sendo possível:

- **remover** construtores existentes para evitar alguma má construção que esteja ocorrendo;
- **adicionar** algum construtor, diante de alguma herança ou relacionamento para aumentar a especialidade da linguagem;
- **alterar** construtores existentes, melhorando e tornando o metamodelo mais específico.

Finalmente, verificamos como as abordagens são validadas ou quais avaliações são feitas pelos autores dos trabalhos analisados. Partindo deste princípio, tomamos por base os atributos:

- (i) **Instanciação**, utilização da abordagem a partir de ferramentas externas ou sem apoio computacional;
- (ii) **Implementação**, implementação da abordagem em alguma linguagem computacional ou implementação de processo de utilização não computacional;
- (iii) **Ferramental ou Protótipo**, implementação da abordagem criando alguma ferramenta ou protótipo.

3.2.1 Abordagem (Borsoi & Becerra, 2008)

Descrição: esta abordagem consiste na definição e modelagem de um Processo de Software, utilizando UML como linguagem de modelagem. Por se tratar de uma definição, os autores percorrem muitos conceitos de Orientação a Objetos e Processos de Software, formando a conceitualização de Processo como um Objeto. Segundo os autores, tal contribuição permite o uso de ferramentas para automatizar a execução do Processo.

Tipo de Extensão da UML: segundo nossas análises, para a conclusão desta abordagem não foi feita nenhuma extensão da UML.

Níveis de formalismo da Linguagem: neste caso, por não se tratar de uma especialização da UML, não fica explícita a criação de uma linguagem própria. Entretanto, é definido um domínio com características próprias a partir dos diagramas UML.

Tipos de Modelos existentes: nesta abordagem á apresentada apenas um nível de modelagem. O modelo conceitual criado para modelar Processos de Software é um modelo de nível de usuário, que utiliza-se da própria UML.

Validação ou Testes: os autores deste trabalho não apresentam nenhuma implementação ou ferramenta capaz de automatizar a utilização da abordagem. Também não foi encontrada

nenhuma verificação, qualquer tipo de teste ou experimento. Ao invés disto, é apresentado um estudo de caso vinculado a um exemplo, não sendo um estudo de caso real.

Relação com a abordagem proposta: os autores trazem um trabalho de modelagem de PDSs utilizando paralelamente a definição de objetos, assim como nossa abordagem, que se constitui na modelagem de Artefatos de PDSs utilizando Orientação a Objetos. Além disso, em nossa abordagem utilizamos os níveis de formalismo de linguagem abstrata e concreta e também apresentamos regras de boa formação e descrições semânticas. Outro ponto bastante relevante é sobre a representatividade do modelo, uma vez que os autores dessa abordagem utilizam apenas um modelo de usuário UML.

3.2.2 Abordagem (Lee et al., 2002)

Descrição: esta abordagem trabalha com a modelagem e metamodelagem dos conceitos de política de acesso a tarefas (*task assignment policy*) de Processos de Software. Neste trabalho foram explorados os recursos possíveis de executar tarefas diversas, com base em política de acesso conforme três aspectos: organização, com foco sobre papéis organizacionais, assim como Gerente de Projetos e Gerente de Configuração; processo, com foco em algumas fases do ciclo de vida, hierarquia de atividades e os papéis executores; e produto, no intuito de policiar artefatos de entrada e/ou saída, além de controlar os papéis responsáveis por tais artefatos.

Tipo de Extensão da UML: a UML foi especializada utilizando-se o mecanismo extensão *heavyweight*, caracterizando-se como uma modificação direta em seu metamodelo. Segundo os autores, tal alteração é bastante sensível e não altera o comportamento inicial da UML. Ao invés disso, esta abordagem apenas adiciona novos construtores de linguagem, tornando este trabalho flexível e com três níveis de modelo: meta, conceitual e de instância.

Níveis de formalismo da Linguagem: quanto aos níveis de formalismo, este trabalho traz uma metalinguagem baseada em MOF e que estende UML, trazendo consigo notação gráfica. Entretanto, outros níveis de formalismo não foram encontrados.

Tipos de Modelos existentes: são utilizados três diferentes tipos de modelos: meta, que consiste nos conceitos utilizados para criar metalinguagem; conceitual, definido pelo meta; e instância, que é justamente a instância do modelo conceitual definido pela metalinguagem criada pelos autores.

Validação ou Testes: no tocante a validação, este trabalho traz uma implementação feita através de *softwares* de geração de código. Além disso, mostra uma instância da abordagem utilizando modelos em diagramas UML e exemplos. Entretanto, nenhuma ferramenta é apresentada e não foram feitos estudos de caso ou experimentos para testar as funcionalidades, além disso, não há exemplos analíticos ou comparativos com casos reais e atuais.

Relação com a abordagem proposta: os autores desta abordagem utilizaram a extensão da UML, criando um metamodelo para permitir a modelagem de política de acesso em atividades

e artefatos em um PDS. Em comparação com nosso trabalho, esta abordagem foi feita a partir do mecanismo *heavyweight* de extensão da UML, enquanto nossa abordagem utiliza ambos os mecanismos de extensão. Outro ponto de comparação é em relação à falta de definição da semântica e regras de boa formação. Além disso, os autores não trazem nenhuma ferramenta.

3.2.3 Abordagem (Pérez-Martínez, 2003)

Descrição: baseados no fato de UML ser altamente aceita como linguagem de representação de *Softwares*, os autores desta abordagem propõem um metamodelo que estende a UML para permitir representação de domínio do estilo arquitetural específico. Como os próprios autores se reservam a afirmar, esta abordagem é bem menos ambiciosa, por se limitar aos conceitos de uma única linguagem específica de arquitetura.

Tipo de Extensão da UML: esta abordagem utiliza-se do mecanismo de extensão da UML, adicionando novos metaconstrutores e criando uma nova metalinguagem para definir e prover modelagem arquitetural de software. Segundo os autores, a utilização de estereótipos não seria suficiente para representar os elementos arquiteturais.

Níveis de formalismo da Linguagem: esta abordagem só não apresenta a linguagem de representação concreta. Neste caso, tanto as regras de boa formação quanto uma linguagem abstrata devem estar presentes, além das especificações semânticas.

Tipos de Modelos existentes: apresenta apenas o metamodelo e não dispõe de exemplos.

Validação ou Testes: os autores não apresentam nenhuma validação ou teste.

Relação com a abordagem proposta: neste trabalho foi desenvolvido um metamodelo para noção de Arquitetura de Software de acordo com o estilo C3. Tal abordagem foi desenvolvida utilizando o mecanismo *heavyweight* de extensão da UML. Assim como nossa abordagem, este trabalho também apresenta descrição semântica dos construtores da linguagem e uma linguagem abstrata, porém não descreve nenhuma linguagem concreta. Os autores não mostram nenhum modelo de instância ou mesmo o modelo conceitual criado a partir do metamodelo constituído. Por fim, não é apresentada nenhuma implementação, teste, ou ferramenta.

3.2.4 Abordagem (Rosener & Avrilionis, 2006)

Descrição: esta abordagem apresenta uma solução de modelo para os elementos que constituem a disciplina de Engenharia de Software. Os autores utilizaram UML para modelar sua especificação, desde o metamodelo até a instância real da Engenharia de Software. Estes modelos identificam os níveis de abstração da disciplina, desde seus pilares filosóficos, caindo por fim, na Engenharia de Software. Desta forma, neste trabalho a Engenharia de Software é uma especialização da Engenharia, que possui seus fundamentos na Ciência, que por sua vez se

baseia na filosofia.

Tipo de Extensão da UML: embora esta abordagem utilize-se de UML para construir os modelos, definindo vários domínios específicos, não há clara evidência de extensão da UML.

Níveis de formalismo da Linguagem: por não se tratar de uma especialização da UML, a linguagem descrita pelos autores é apresentada como um modelo UML definido pelos diagramas. Embora existam os níveis de modelagem previstos para os diferentes aspectos da disciplina de Engenharia de Software, não foi possível identificar como as ligações de instância desses modelos foram restringidas. Em outras palavras, não foi apresentado de forma clara como a Engenharia de Software pode ser uma instância da Ciência.

Tipos de Modelos existentes: embora não tenham sido identificados como modelos de definição de outros modelos, mas sim diferentes modelos de usuário da UML sem ligação entre si, aparentemente eles se constituem em metamodelo (Filosofia), modelo conceitual e (Ciência) e instância (Engenharia e Engenharia de Software).

Validação ou Testes: esta abordagem apresenta um exemplo para utilização de diferentes modelos chamados: *Functional Solution Model*, *Technical Problem Model* e *Technical Solution Models*. Estes modelos foram implementados em linguagem Java, com o banco de dados e seu acesso gerado em linguagem DDL e interface gráfica em HTML, tudo isto utilizando geradores de código. Em termos de validação ou testes, esta abordagem apresenta um exemplo de utilização de um Projeto *European Institute*, que se compromete em suportar processos de negócio entre os Estados-Membros Europeus e a *European Commission*.

Relação com a abordagem proposta: este trabalho não utiliza o MOF para criação de uma meta-linguagem, pois utiliza a própria UML para definir vários níveis de modelos que se instanciam entre si. Neste trabalho também é apresentado implementação, porém nenhuma realização de ferramenta específica ou protótipo foi encontrado.

3.3 Resultado da Análises

O objetivo desta seção é apresentar de forma sintética os resultados da comparação entre os trabalhos analisados e a abordagem definida nesta dissertação. Para isto foi realizado uma análise comparativa, descrita anteriormente. Os resultados desta análise estão ilustrados nas Tabelas nas Figuras 3.1 e 3.2 oriundas das Seções 3.3.1 e 3.3.2, respectivamente.

3.3.1 Resultado: autoria de ASs

Os critérios de comparação da Figura 3.1 foram definidos anteriormente na Seção 3.1. A partir da Figura 3.1 observa-se que:

	Linguagem					Autoria				Características				Testes		
Abordagem	**1**	**2**	**3**	**4**	**5**	**A**	**B**	**C**	**D**	**I**	**II**	**III**	**IV**	**Inst.**	**Impl.**	**Prot.**
Akpotsui1992		X	X	X	X	S		Documento								
Buchner2000					X	S		Documento							X	X
Cattaneo2000			X	X		S		Artefato	X						X	X
Herzner1991		X				S		Documento							X	X
Laitinen1992		X				N	X	Documento			X				X	
Tilley1991					X	N		Código-Fonte							X	
Visconti1993	X	X			X	N	X	Documento					X			
Nossa Abord.	**X**	**X**	**X**	**X**	**X**	**S**	**X**	Artefato	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**

1 - Semântica Estática
2 - Semântica Dinâmica
3 - Linguagem Abstrata (notação)
4 - Linguagem Concreta
5 - Regras de boa formação

A - Paradigma de construção
B - Guia de Etapas da construção
C - Escopo da abordagem
(**S -** Contempla Documentos Compostos,
N - Paradigma Tradicional)
D - Separação de Conteúdo

I - Extensibilidade
II - Classificação
III - Controle de Versões
IV - Controle de Maturidade

Inst. - Instância ou Uso
Impl. - Implementação Computacional
Prot. - Protótipo ou Ferramenta

X - Suportado pela abordagem

Figura 3.1: Tabela com resultados sintéticos da comparação de trabalhos sobre autoria de ASs.

- **Linguagem:** os níveis de formalismos de todas das abordagens analisadas são incompletas. Apenas (Visconti & Cook, 1993) possui semântica estática e de todas, a mais completa é (Grolman et al., 1997), que traz consigo os outros níveis de formalismo exigidos. Além disso, apenas em duas abordagens, das sete analisadas, existe notação e linguagem concreta.

- **Autoria:** no que se diz respeito a autoria, a grande maioria das abordagens (cinco) possuem foco apenas em artefatos do tipo documentos, apenas uma em artefatos do tipo código-fonte, e apenas uma outra compreende artefatos em geral. Como pôde ser visto, das abordagens analisadas, apenas os trabalhos de (Laitinen, 1992) e (Visconti & Cook, 1993) trouxeram evidências no tocante a existência de um Guia para a construção. Por fim, a separação de conteúdo não aparenta ser algo muito popular dentre as abordagens, entretanto, os paradigmas de construção de artefatos fica bastante dividido.

- **Características:** dentre as características analisadas, foi observado que poucas ou nenhuma das abordagens corresponderam ao esperado. Nenhuma das abordagens implementa ou sugere algum tipo de mecanismo para permitir extensibilidade da proposta e controle de versões dos artefatos.

- **Testes:** nenhuma da abordagens analisadas faz uso ou algum tipo de testes de instância, entretanto, a grande maioria utiliza-se de protótipos ou ferramentas computacionais.

3.3.2 Resultado: Metamodelagem

Os critérios de comparação da Figura 3.2 foram definidos anteriormente na Seção 3.3.2. A partir da Figura 3.2 observa-se que:

- **Mecanismo de extensão da UML:** entre as abordagens analisadas, apenas (Lee et al., 2002) e (Borsoi & Becerra, 2008) fazem extensão da UML. Tal extensão utiliza o mecanismo *heavyweight*, que como pode ser visto na Seção 2.3.2, caracteriza-se por adicionar ou alterar construtores da linguagem, conforme a necessidade de restrições.

- **Linguagem:** no que se refere aos níveis de formalismo, apenas (Pérez-Martínez, 2003) implementa o valor semântico de sua linguagem por completo, juntamente com regras de boa formação, o que não ocorre nas outras abordagens. Entretanto, todos os trabalhos analisados utilizam notação em linguagem abstrata, porém, apenas (Borsoi & Becerra, 2008) dispõe de uma linguagem concreta.

- **Modificações feitas sobre UML:** como pode ser observado, apenas as abordagens (Lee et al., 2002) e (Pérez-Martínez, 2003) alteram o metamodelo da UML, adicionando construtores a linguagem. Porém, os próprios autores advertem que preferiram não alterar a UML numa tentativa de tornar suas linguagens mais independentes.

- **Tipos de modelos utilizados:** para os tipos de modelos, foi analisada a existência de metamodelo e sua instância. Dos trabalhos analisados apenas (Borsoi & Becerra, 2008) não dispõe de metamodelo e apenas (Pérez-Martínez, 2003) não continha um modelo conceitual que instanciasse o metamodelo proposto.

- **Testes:** embora (Lee et al., 2002) e (Pérez-Martínez, 2003) tenham proposto implementação e uso de suas abordagens, seus trabalhos não apresentam nenhuma ferramenta ou mesmo um protótipo. Os outros trabalhos não apresentam nenhuma validação, verificação, formas de uso ou testes.

	Extensão		Linguagem					Modificações da UML			Modelos		Testes		
Abordagem	**H**	**L**	**1**	**2**	**3**	**4**	**5**	**A**	**B**	**C**	**MM**	**M**	**Inst.**	**Impl.**	**Prot.**
Borsoi2008			-	-	X	X		-	-	-		X			
Lee2002	X				X					X	X	X	X	X	
Mart'inez2003	X		X	X	X		X			X	X				
Rosener2006			-	-	X			-	-	-	X	X	X	X	
Nossa Abord.	**X**	**X**	**X**	**X**	**X**	**X**	**X**		**X**	**X**	**X**	**X**	**X**	**X**	**X**

H - Mecanismo de extensão da UML - Heavyweight
L - Mecanismo de extensão da UML - Lightweight

1 - Semântica Estática
2 - Semântica Dinâmica
3 - Linguagem Abstrata (notação)
4 - Linguagem Concreta
5 - Regras de boa formação

A - Remoção de Construtores
B - Modificação dos Construtores
C - Adição de novos Construtores

M - Nível de Metamodelo
MM - Nível de Modelo Conceitual

Inst. - Instância ou Uso
Impl. - Implementação Computacional
Prot. - Protótipo ou Ferramenta

X Suportado pela abordagem
- Não empregado

Figura 3.2: Tabela com resultados sintéticos da comparação de trabalhos sobre metamodelagem.

4 Autoria de Artefatos de Software

Neste Capítulo apresentamos os caminhos utilizados no tocante a uma solução dos problemas e desafios identificamos no Capítulo 1. Para atingir os objetivos, nossa abordagem consiste na identificação de uma estrutura lógica que foi desenvolvida para ser genérica, dentro do escopo dos ASs participantes dos PDSs analisados. Além disso, também foram identificados diferentes níveis de estruturação, tanto dos ASs quanto das informações armazenadas.

Desta forma, nossa proposta de estruturação de ASs consiste na criação de uma metalinguagem desenvolvida sobre o metamodelo da UML, sendo o seu cerne a utilização de contêineres de informações capazes de coordenar e organizar os artefatos estruturalmente, assim como utilizar os tipos de informações convenientes.

Este Capítulo está descrito nas Seções:

- 4.1 **Artefatos x Templates:** uma vez determinado o escopo da nossa proposta, nesta Seção são apresentados detalhes maiores sobre ASs e como se comportam dentre as fases existentes em PDSs.

- 4.2 **Identificação do escopo:** nesta Seção é apresentada a solução em questão, desde o seu princípio em PDSs até chegarmos ao foco do nosso trabalho, neste caso, Autoria de Artefatos de Software;

- 4.3 **Definição de Artefatos:** após a identificação do escopo e apresentação dos detalhes necessários para um melhor esclarecimento sobre como ASs e PDSs são entendidos dentro deste trabalho, nesta Seção está a identificação dos problemas existentes com maiores detalhes, juntamente com a apresentação de uma solução conceitual em detrimento da solução atual utilizada, comparando nossa abordagem.

- 4.4 **Uso de Artefatos:** último passo para concluir a autoria de artefatos. Essa Seção mostra como foi modelado o uso do que foi definido durante a Seção 4.3.

- 4.5 **Metamodelo e *UML Profile*:** nesta seção é vista uma visão geral sobre o modo como fizemos a extensão da UML e do SPEM v2.

- 4.6 **Guia para Autoria de Artefatos de Software:** nesta seção é apresentado um guia revisado para a construção de artefatos, desde a especificação até seu uso e instância.

- 4.7 **Regras para a Autoria de Artefatos de Software:** são apresentadas as principais regras definidas para a construção de artefatos bem estruturados.

- 4.8 **Níveis de Formalismo e Pontos de Conformidade:** nesta seção são apresentados as características da abordagem, os níveis de formalismo utilizados no Metamodelo e *UML Profile*, além dos pontos de conformidade existentes.

4.1 Artefatos x Templates

Artefatos de Software são os elementos principais e o objetivo de existência de um PDS, pois espera-se que após o término de sua execução, todos os artefatos necessários tenham sido produtos. Desta forma, o conjunto formado pelos artefatos produzidos durante a execução de um PDS permite a criação de um Produto Final.

Entretanto, mediante a existência de vários e diferentes PDSs, nem sempre os termos e nome dos elementos são os mesmos aqui empregados. Por isso, de acordo com o PDS utilizado, é bem comum encontrar nomenclaturas totalmente diferentes, mas que possuem o mesmo significado. Desta forma, para fins de conhecimento e um melhor vocabulário comum, a Tabela 4.1 traz uma tradução dos diferentes termos encontrados nos PDSs atualmente mais comuns, no que se refere a Papel, Atividade e Artefato.

Tabela 4.1: Resumo e tradução dos conceitos encontrados nos processos analisados conforme (OMG, 2005)

Processo	**Artefato**	**Papel**	**Atividade**
RUP	Artifact	Role	Activity/Step
Unisys QuadCycle	Artifact/Asset	Role	Activity/Step
DMR Macroscope	Deliverable Product	Role	Activity
OPEN	Deliverable	Role/Direct Producer	Task/SubTask
SCRUM	Artifact	Role	Step
MSF	Product	Role	Activity
IBM GSM	Work Product Description	Role	Task
SPEM	WorkProduct	ProcessRole	Activity/Step

Todo AS possui sua própria organização interna e isto é o que os diferencia dentre os demais. Por exemplo, um artefato do tipo *Código-Fonte* é totalmente diferente de um *Documento de Visão*. Mesmo assim, não existe um padrão que determine como deve ser um artefato e não sabemos quem, ou o que, os construirá. Por isso é complicado definir como os artefatos devem ser preenchidos sem a utilização de algum guia.

Partindo deste princípio e utilizando as facetas de definição, uso e instância pode-se definir artefatos para um Processo específico, assim como o RUP. O RUP utiliza-se de uma lista de *templates* (definição) que especificam como devem ser os artefatos. Além disso, são fornecidas informações sobre quais atividades e papéis devem ser utilizados para construir instâncias a partir do *template* (uso). Tais *templates* contam com breves descrições de conteúdo e estrutura para determinar como serão os artefatos (instância) do Projeto.

Por exemplo, para desenvolver o artefato **Visão**, do RUP, primeiro precisaríamos descobrir seu objetivo e conhecer seu *template*. Desta forma podemos investigar sua estrutura e organização, encontrando dados para saber como deve ser o seu preenchimento e utilização dentro do projeto. Após isso, devemos saber quais atividades criam, alteram ou lêem o documento Visão e quais são os papéis atuantes neste artefato. Após isto, segundo (Kruchten, 2000, Kroll & Kruchten, 2003) teremos informações suficientes para criar o artefato Visão durante um Projeto.

No entanto, artefatos devem ser especificados durante a fase de definição do PDS *authoring*, que por sua vez possui dois instantes de criação, *Method Content* e *Process Structure*. Diferentemente, seu preenchimento, ou seja, sua produção é feita no decorrer da execução do PDS (*enactment*) em forma de projeto.

Além disso, diferentes PDSs poderão produzir diferentes tipos de artefatos, que são na verdade a união de várias camadas para representação da informação. Vejamos:

(a) camada de informação ou conteúdo: (i) nesta está toda informação considerada realmente importante e que poderá ser transformada em conhecimento. Além disso, é levado em conta que (ii) diferentes informações possuirão diferentes níveis de restrições, os quais podem ser determinados através do tipo de informação (ex.: numérico, *string*, data) ou sua necessidade (ex.: *empty*, *null*, *unique*).

(b) camada de estrutura esta camada é responsável por (i) determinar a estrutura a ser utilizada para guardar o conteúdo da informação e verificar que (ii) a mesma informação pode ser estruturada de maneiras diferentes, sendo em suas formas (ex.: imagens, tabelas, textos) ou em sua lógica de apresentação (ex.: em capítulos, seções, subseções).

(c) camada de formato conforme a estrutura utilizada (i) faz-se necessário publicar, ou mesmo garantir que informações sejam visualizadas de forma a aumentar a compreensão sobre ela. Desta forma (ii) a mesma informação pode ser vista de diferentes maneiras, dependendo da necessidade de representação (ex.: um gráfico em barras ou pizza). Por fim, (iii) diferentes processos podem utilizar diferentes metodologias ou métodos e produzir diferentes tipos de artefatos como resultado (Ex.: planilhas, documentos, arquivos XML).

Essa estrutura de camadas para construção de artefatos ajuda na definição dos papéis responsáveis pelas fases do PDS. Diferente dos atores existentes em um PDS, os atores que o constrõem atuam em papéis exclusivos. Sendo assim, poderíamos definir papéis como: Engenheiros de Conteúdos ou Conhecimento, para investigar informações necessárias; Engenheiro de Documentos para definir a estrutura de documentos e criar o *Method Content*; e o Engenheiro de Processos para criar o *Process Structure*.

Portanto, sabemos que os artefatos armazenarão informações, utilizando-se de uma estrutura própria para o armazenamento e apresentando de forma com que o leitor entenda a informação disponível.

4.2 Identificação do escopo

Embora nossa solução cause impacto explicitamente sobre PDSs, esta acaba por ser um pouco mais pontual, no tocante a algumas fases específicas. Desta forma, para que pudessemos focar exatamente no ponto em que precisaríamos, procuramos entender um pouco melhor a natureza de PDSs, estudando também o seu ciclo de vida. De acordo com (Noll, 2007), um PDS possui três fases durante seu ciclo de vida de desenvolvimento e cada uma delas é responsável por pontos específicos: *authoring*, criação ou autoria; *tailoring*, adaptação ou configuração; e *enactment*, execução.

Além de determinarem o ciclo de vida, tais fases servem para definir as facetas admitidas para um PDS, de acordo com o ponto de vista em que ele for observado. A Figura 4.1 apresenta as diversas facetas que podem ser encontradas durante o ciclo de vida de um PDS.

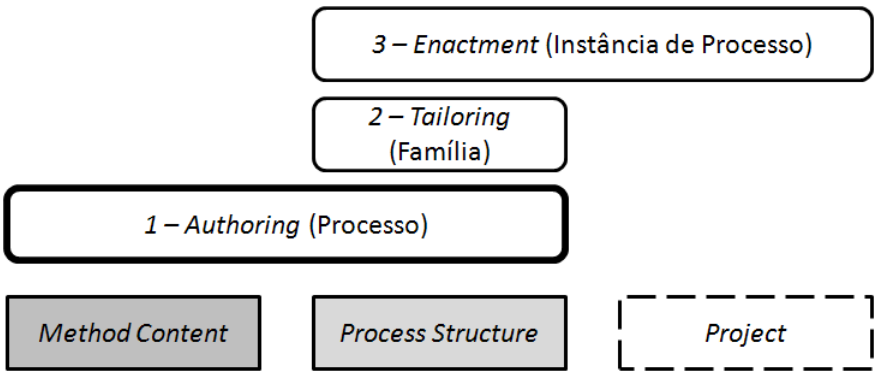

Figura 4.1: Diferentes facetas de um PDS.

Desta forma, com o PDS em fase de (1) *authoring*, poderemos encontrar duas facetas: sua *Method Content*, que é uma biblioteca de conteúdos com os elementos necessários para montar *Process Structure*, que é a estrutura do PDS feita a partir da biblioteca. Em fase de (2) *tailoring* será encontrado como Processo Padrão (*Standard Software Development Process*), o qual serve de base para um novo Processo Adaptado (*Specific Software Development Process*) (Pereira et al., 2008). Finalmente, na fase de (3) *enactment*, o processo é se transforma em uma instância de si mesmo, onde é executado e acompanhado com a faceta de Projeto, que segundo (PMI, 2004), é um esforço temporário empreendido para criar um produto, serviço ou resultado exclusivo.

Como pode ser visto na Figura 4.2, existe uma biblioteca com o conteúdo a ser utilizado para montar a estrutura do PDS, para só então, instanciá-lo como um projeto. Desta forma, são definidos três diferentes repositórios, dividindo os níveis de informação. Por fim podemos ver que as fases de *authoring* e *tailoring* podem se utilizar do mesmo repositório.

Devido a existência de ordem hierárquica entre as fases de um PDS, elas devem estar em conformidade de acordo com as restrições estabelecidas por cada camada. Além disso, quanto mais alto o nível em que o estiver um PDS, mais difícil será para gerenciá-lo. Por exemplo,

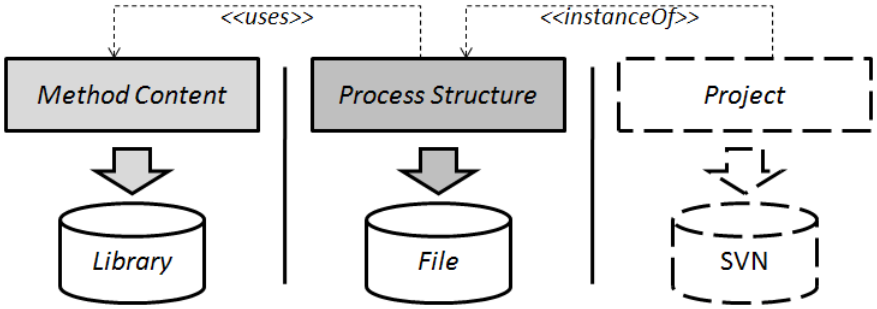

Figura 4.2: Diferentes repositórios utilizados para guardar os diferentes níveis de Informação.

uma vez que um já definido elemento é adicionado ao *Method Content* (nível 1) e utilizado para construir o *Process Structure* (nível 2), as alterações posteriores feitas na definição do elemento deverão ser reproduzidas. Caso o PDS estivesse em fase de *enactment* (nível 3), seria muito mais difícil refletir as alterações durante a execução do projeto.

Como visto na Seção 2.1, existem vários elementos que compõe um PDS, assim como Artefatos, Atividades e Papéis. Diante das fases existentes, tais elementos possuem dependências entre si, caracterizando diferentes facetas. Desta forma, em nível de *Method Content*, todos os elementos estarão com a faceta de **Definição**. Em nível de *Project Structure*, a faceta utilizado é a de **Uso** e somente na camada de projeto que o elemento se tornará uma instância real. Sendo assim, os elementos Artefatos, Atividades e Papéis estarão nas seguintes facetas da Figura 4.3.

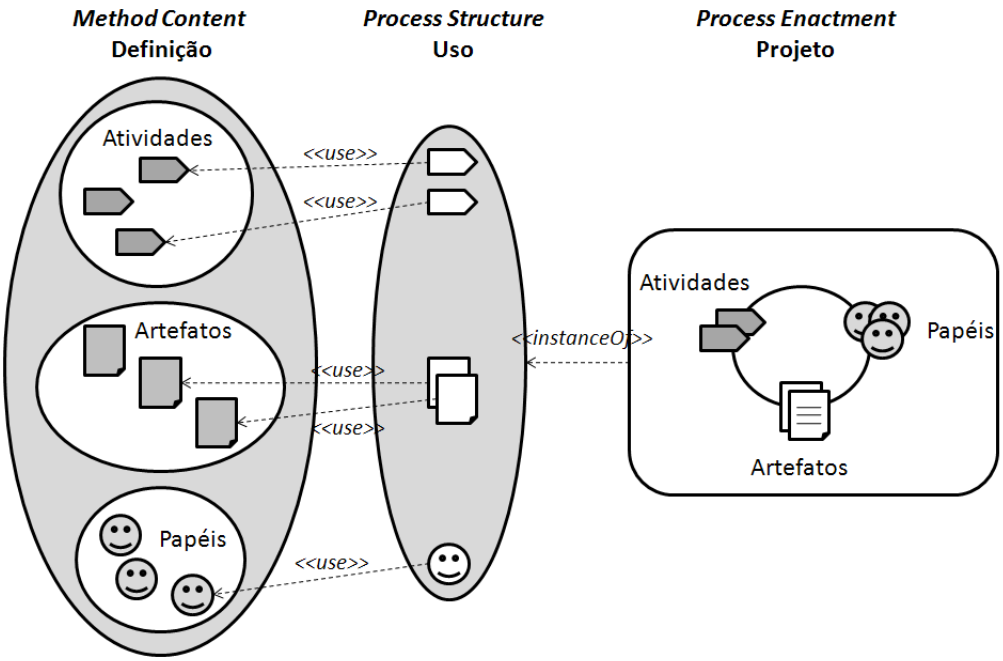

Figura 4.3: Diferentes facetas dos elementos de um PDS

De acordo com o alinhamento entre os problemas e objetivos apresentados no Capítulo 1, nosso escopo reduz-se ainda mais, explorando o elemento **Artefato**. Portanto todo o nosso foco estará em melhorar os problemas identificados neste elemento.

4.3 Definição de Artefatos

Assim como definido na especificação da UML, nossa idéia é definir um metamodelo para a autoria de ASs especificando a sintaxe e a semântica da linguagem, utilizando como base os princípios de: (i) modularidade; (ii) *layering*; (iii) *partitioning*; (iv) extensibilidade; (v) reuso; e utilizando:

- **sintaxe**: representada através de notação UML, mostrando as metaclasses que definem os construtores da linguagem (como por exemplo: *Association*, *Operation*, *Property*, entre outros) e seus relacionamentos; além da existência de uma sintaxe computável;
- **regras de boa formação:** trata-se de um conjunto de regras de boa formação que são descritas usando OCL ou linguagem natural;
- **semântica:** para cada construtor da linguagem é associado um significado. Para isto é usado linguagem natural.

O uso da linguagem natural para especificar os elementos que compõem a linguagem, como a semântica, as regras de boa formação e sintaxe abstrata citados anteriormente, favorece o surgimento de ambigüidade. Conseqüentemente, implica em dificuldades em estender o metamodelo da linguagem. Este trabalho não tem como um de seus objetivos apresentar uma especificação formal da extensão do SPEM v2, mas sim uma descrição de autoria de ASs, seguindo as recomendações de extensão do metamodelo apresentada da especificação da linguagem.

Embora na literatura existam diversas maneiras de definir como devem ser PDSs, a utilização de UML para representar uma linguagem gráfica que permita reproduzir a criação de processos vêm se tornando cada vez mais popular. Diversos metamodelos que explicam como devem ser os modelos de um processo determinado já foram ou estão sendo concluídos, assim como o metamodelo do RUP e do OPEN.

Entretanto, nenhum dos processos analisados possui um metamodelo que defina ASs de forma detalhada, constituindo-se a construção de forma manual, desde sua definição até o seu uso e instância. Geralmente, no tocante a definição, uso e preenchimento é utilizada linguagem natural (como português e inglês) o que dificulta análise computacional posterior. Desta forma, se quisessemos apenas obter ASs na forma tradicional, poderíamos determinar o nome e uma breve descrição e deixar a produção por conta dos papéis responsáveis.

Sendo assim, partindo do objetivo do trabalho, reutilizamos a linguagem UML para que seja possível a construção de um metamodelo que represente ASs detalhadamente, a partir da extensão do metamodelo do SPEM v2. Para evitar conflitos de nomes com os metamodelos da própria UML e SPEM v2, o metamodelo definido nesse trabalho também utilizada nomenclatura em inglês.

Conforme apresentado na Seção 2.3.2, ao se criar ou extender uma linguagem baseada no metamodelo da UML, deve-se utilizar o MOF. Na Figura 4.4 estão as camadas relativas ao

que estamos propondo. Na camada **M3** está o MOF, característico por permitir a criação de metamodelos, como os da camada **M2**.

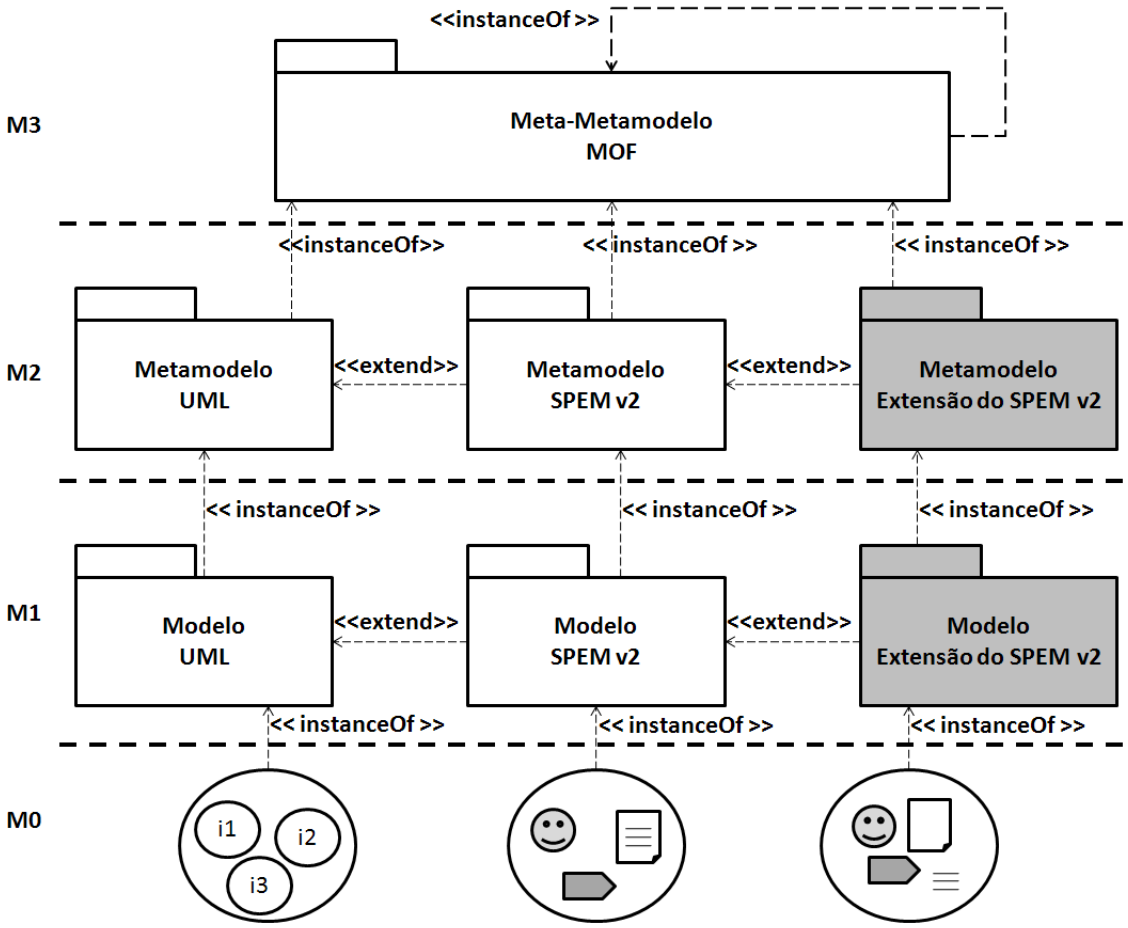

Figura 4.4: Camadas

O foco da modelagem da nossa proposta consiste em criar uma linguagem em nível M2, ou seja, uma metalinguagem, permitindo o seu uso na camada **M1**. O modelo de domínio determinado através da nossa proposta de metamodelo pode ser realizado na camada M1, mesmo nível dos usuários da UML. Na camada M0 estão as instâncias, ou seja, elementos reais do processo, assim como artefatos, papéis e atividades. Dado o SPEM v2, nossa extensão acrescenta os elementos cruciais para criação de artefatos em detalhes.

No decorrer desta Seção estão apresentados os diversos problemas identificados por este trabalho, juntamente com a solução encontrada. Tal solução é apenas uma breve descrição das partes modeladas para a constituição de um metamodelo comum ao SPEM v2 e a UML, sua especificação completa pode ser vista no Apêndice B.2 As análises foram feitas a partir de ASs definidos em dois processos de software diferentes, RUP (processo tradicional) e SCRUM (processo ágil). Para o RUP foram utilizados os artefatos definidos em (Kruchten, 2000) e comparados com os mesmos dos software *Rational Unified Process*, baseado na *Web*. Já para o SCRUM foram utilizados os artefatos definidos em (Schwaber, 2004).

4.3.1 Cenário 1 - Tipificação dos Artefatos envolvidos

Atualmente, ASs são preenchidos com a utilização de ferramentas específicas e que trabalham de acordo com o tipo do artefato. Porém, durante um projeto, estes tipos estão fortemente atrelados a informação da extensão do arquivo. Desta forma, se um artefato possui uma extensão compreendida por algum software específico, poderá então ser manuseado.

Alguns exemplos de ferramentas bastante utilizadas são: (i) *Microsoft Office Word*™ou *Open Office Writer*, para ASs que possam ser preenchidos como documentos; (ii) *Microsoft Office Excel*™ou *Open Office Calc*, para ASs a serem preenchidos como planilhas de cálculos ou matrizes; (iii) *IMB Eclipse*[1] ou *Sun Microsystems NetBeans*[2], geralmente utilizados para artefatos de código-fonte; (iv) *IBM Rational Modeler* [3] ou Enterprise Architect [4], para modelagem Orientada a Objetos utilizando UML.

Pontos Fracos

Conforme nossas análises, os pontos fracos desta abordagem são: (i) não existe uma lista que apresente os tipos de ASs existentes; (ii) o tipo de cada AS é determinado a partir de sua extensão ou ferramental utilizado.

Exemplos

Exemplo 1: O artefato Glossário, do RUP, é uma lista de termos juntamente com seus significados, entre outros dados significativos. Entretanto, não existe uma definição sobre o tipo deste AS. Desta forma, caso sua representação seja uma tabela com os termos, podemos afirmar que o Glossário é uma planilha. Apesar disto, um Glossário também poderia ser um documento com listas em tuplas do tipo termos/significado. Além desses, um Glossário poderia ser um arquivo XML, com os termos e seus significados mapeados através de *XML tags* definidas a partir de um *XML Schema* ou DTD.

Exemplo 2: O artefato *Sprint Backlog*, do SCRUM geralmente é montado sobre uma tabela que agrupa as informações sobre o andamento do projeto durante o *Sprint* corrente. Desta forma, também pode ser considerado uma planilha e diferente do Glossário, dificilmente poderia ser construído sobre um documento comum. Entretanto, existem possibilidades de se utilizar XML ou ferramentas com base em alguma linguagem específica.

Desta forma, partindo destes dois exemplos, fica mais evidente os problemas existentes por falta de uma definição do tipo dos ASs.

[1] www.eclipse.org
[2] www.netbeans.org
[3] http://www-01.ibm.com/software/awdtools/modeler/swmodeler
[4] http://www.sparxsystems.com.au

Análise e Resultados

Da análise feita sobre os processos, extraímos os seguintes resultados:

- **RUP**: 108 artefatos - dos tipos documento (50), modelo UML (32), código-fonte (3), planilha (3) e (2) binário. Infelizmente, 2 artefatos não foram encontrados e em outros 16 não foi possível concluir qualquer tipo. Entretanto, dos artefatos analisados, apenas 37 são fortemente estruturados. A grande maioria utiliza-se de *templates* como definição;

- **SCRUM**: 8 artefatos - dos tipos planilha (2), diagrama / gráfico (3), código-fonte (2) e binário (1). Nenhum deles é fortemente estruturado embora os artefatos do tipo planilha apresentem estrutura quase padrão de representação. Utiliza-se *template* como definição.

Desta forma, mostra-se necessário a definição de tipagem de artefatos para os tipos Documento, Planilha, Diagrama, Gráfico, Modelo UML, Código-Fonte e Binário. Portanto o metamodelo deverá possuir construtores para utilizar a tipificação de artefatos baseada nos resultados obtidos.

Solução encontrada

A solução escolhida para modelar o conceito apresentado neste cenário tem por base a criação de uma enumeração de tipos. Esta enumeração é capaz de representar os tipos existentes que, aliados a uma metaclasse Artefato, permite a tipificação. A Figura 4.5 apresenta a solução com base na metaclasse `ArtifactDefinition`, juntamente com um atributo *artifact kind* de nome `akind` que referencia os literais da enumeração `ArtifactKinds`.

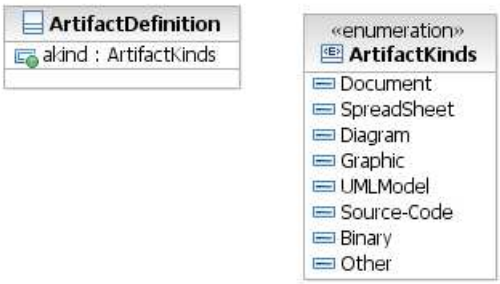

Figura 4.5: Metaclasses responsáveis pela tipificação do ASs

Comparação com abordagens atuais

Com a introdução de tipificação, podemos determinar os possíveis tipos de ASs, desde a definição até o preenchimento dos mesmos. Além disso, é possível criar categorias de ASs baseados nos tipos, uma vez que são conhecidos.

4.3.2 Cenário 2 - Definição das Estruturas das Informações participantes

Conforme analisado, diferentes ASs possuem diferentes informações. Tais informações carregam consigo uma determinada estrutura e possivelmente formas diferentes de visualização. Entretanto, atualmente não existe um padrão ou formato para estas informações, dificultando tanto sua definição estrutural, quando o seu preenchimento posterior.

Pontos Fracos

Esta abordagem tráz consigo alguns problemas como: (i) falta de conhecimento prévio da estrutura a ser utilizada em determinada informação; (ii) má interpretação do tipo ou estrutura a ser utilizada; (iii) preenchimento ambíguo por conta da decisão de estrutura tardia; (iv) falta de padronização estrutural.

Exemplos

Exemplo 1: O artefato Modelo de Caso de Uso demonstra alguns problemas em relação a sua estrutura. Apesar de ser baseado em UML, no RUP é utilizado um *Template* que especifica uma estruturação para cada caso de uso. Desta forma, temos que construir Casos de Uso UML a partir de sua especificação. Isto gera bastante confusão na estrutura a ser utilizada, pois pode-se utilizar tabelas, texto livre ou até uma estrutura específica para a especificação enquanto que seu modelo poderia ser em UML ou até mesmo uma representação em forma de figura. Portanto, fica difícil determinar o que será um caso de uso, pois é dependente de contexto.

Exemplo 2: O SCRUM apresenta o mesmo problema. O artefato *BurnDown Chart* é representado através de um gráfico em forma de imagem, seus dados são coletados em planilhas. Sendo assim, este artefato poderia ser representado através de uma tabela simplificada ou de listas. Embora isso ocorra, naturalmente existe uma padronização do gráfico, mas não da estrutura que este deve utilizar.

Análise e Resultados

Conforme verificações em *templates* de ASs, obtivemos os seguintes resultados:

- **RUP**: de 108 artefatos apenas 39 possuem *templates* e 21, exemplos; (i) não há artefatos de tipos binários, planilha e código fonte que possuam *templates* ou exemplos; (ii) de 32 artefatos do tipo modelo UML, apenas 3 possuem *templates* e outros 4 possuem exemplos; (iii) de 50 artefatos do tipo documento, 36 deles possuem templates e destes, 17 possuem exemplos;
- **SCRUM**: dos 8 artefatos analisados, todos possuem *templates* e apenas 2 não possuem exemplos.

Neste próximo passo classificamos os tipos de informações existentes, utilizando três tipos possíveis de classificação: *Simples* ou *Primitiva*, sendo os tipos mais simples possíveis; *Complexa*, informação que deve possuir alguma estruturação mínima e que utiliza os tipos simples; e *Específica*, geralmente informação bem estruturada mas encontradas especificamente no processo analisado.

Após a classificação, os resultados obtidos foram:

- **RUP**: 97 tipos de estruturas, sendo 2 simples, 9 complexas e 86 específicas.
- **SCRUM**: 14 tipos de estruturas, sendo 2 simples, 5 complexas e 7 específicas.

Concluindo, os tipos encontrados foram: **simples** - Texto, Texto com Etiqueta e Imagem; **complexo** - Comentário, Diagrama, Lista, Tabela, Grupo, Pergunta e Plano X, Y (conteúdo desenvolvido a partir de informação para montagem de eixo euclidiano x, y).

Solução encontrada

Para modelar o conceito apresentado neste cenário utilizamos a criação de diferentes tipos de informações, os quais, cada um possui uma diferente estrutura. Contudo, nossas análises indicam ao menos três tipos possíveis de estrutura, e isso esta representado no metamodelo. Desta forma, modelamos então três pacotes, conforme a Figura 4.6: `SimpleTypes`, `ComplexTypes` e `SpecificTypes`; além disso, adicionamos um quarto pacote, `DiagramTypes`, preparado para permitir a construção de modelos a partir de diagramas da *UML Superestructure*.

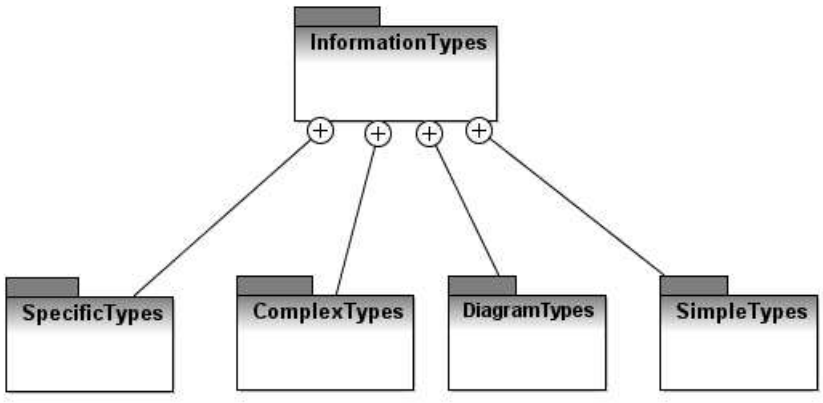

Figura 4.6: Metaclasses responsáveis pela tipificação do ASs

As metaclasses responsáveis pela estruturação da informação estão dentro dos seus respectivos pacotes. A Figura 4.7 apresenta as metaclasses Pai, ou seja, as super metaclasses de cada conjunto de tipos de estrutura que estão nos pacotes supracitados.

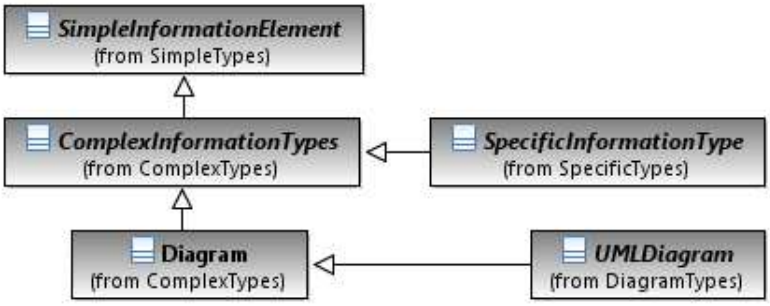

Figura 4.7: Super metaclasses responsáveis pela tipificação das Informações

Como pode ser observado na Figura 4.7, a metaclasse abstrata `SimpleInformation Element` generaliza a metaclasse `ComplexInformationElement` que é especificada pelas metaclasses `SpecificInformationElement` e `Diagram`. Além disso, também é possível notar que os diagramas UML são submetaclasses de `UMLDiagram`.

Comparação com abordagens atuais

Com a estruturação das informações podemos padronizar sua utilização nos ASs em que estarão dispostas. Além disso, existirá uma menor preocupação com corretividade estrutural da informação, uma vez que esta já foi determinada desde a definição do AS. Desta forma, durante a fase de definição do *Method Content* podemos utilizar os tipos de informação para montagem, ou seja, autoria dos ASs, fazendo com que se tornem *Compounding Documents*.

Com isso, podemos definir todos os tipos de informação existentes em cada um dos artefatos que desejamos criar. Já na fase de uso, ao se utilizar um AS específico, toda a hierarquia, assim como a estrutura afixada na definição do AS será também copiada, permitindo que o PDS base possua AS com estrutura de informações pré-definidas. Finalmente, durante a fase de projeto, por se tratarem de instâncias dos elementos do *Process Structure*, serão preenchidos base nas estruturas já determinadas.

Conforme apresentaremos mais adiante (em 4.3.4), informações tipificadas conforme suas estruturas possibilitam o reuso de definição estrutural e, posteriormente no preenchimento, reuso do próprio conteúdo.

4.3.3 Cenário 3 - Determinar a Estrutura Interna dos artefatos envolvidos

Cada AS possui uma estrutura interna que serve para viabilizar o acesso as informações expondo-as eficientemente. Conforme existem diferentes artefatos, também podem existir diferentes estruturas.

Além disso, muitas vezes um conteúdo informativo sofre avanços ou incrementos no decorrer do projeto. Outras vezes, a mesma informação sofre correções ou refinamentos. Desta

forma, a informação estará sendo exposta de diferentes maneiras a medida que evolui, sendo importante que a estrutura de um artefato esteja flexível o bastante para suportar essas mudanças na informação.

Geralmente, a estrutura escolhida para um artefato é dada para organizar informações. Caso contrário, informações estariam jogadas no artefato conforme a estrutura específica de cada uma.

Pontos Fracos

De acordo nossas análises, os pontos fracos desta abordagem são: (i) não existe uma informação que caracterize a estrutura do artefato em relação ao seu tipo; (ii) pelo preenchimento ser dependente de ferramenta do tipo Office, a abordagem torna-se bastante flexível, porém sem estrutura bem definida; (iii) não é possível determinar ou categorizar a informação estruturalmente, apenas visualmente.

Exemplo

O artefato Lista de Riscos possui uma estrutura baseada em seções, dividindo as informações sobre os riscos em vários níveis. Desta forma, embora a própria informação possua uma estrutura é importante que o próprio artefato possa categorizá-la, para melhor disponibilizar as informações existentes.

Análise e Resultados

Dados artefatos de ambos os processos supracitados, encontramos as estruturas do tipo: (i) árvore, comum em documentos; (ii) lista; e (iii) tabela, comum em planilhas;

Solução encontrada

Conforme observamos, todas as estruturas encontradas podem ser derivadas de uma única estrutura genérica baseada em composição, que é uma solução bastante comum em estrutura de dados. Optamos por utilizar uma estrutura mais genérica em detrimento de estruturas específicas pelo fato de haver maior flexibilidade pois, aparentemente, não parecia haver ganho em introduzir uma modelagem mais complexa com diferentes estruturas especificas de ASs.

A Figura 4.8 apresenta a modelagem que foi feita para tornar possível a estruturação do AS. Nesta mesma Figura está a metaclasse responsável por definir a estrutura geral de AS, a `ContainerDefinition`. Como pode ser visto, cada contêiner possui ainda dois relacionamentos do tipo pai/filho, melhor representado a partir da metaclasse `ContainerDefinition_Relationship` criando uma espécie de estrutura composta genérica que permite a criação de vários contêiners alinhados.

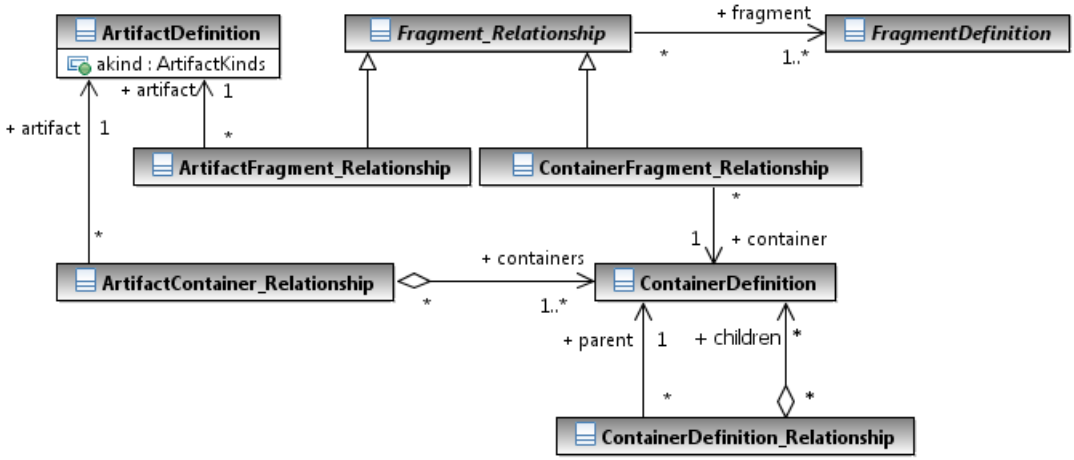

Figura 4.8: Metaclasses responsáveis pela estruturação do AS

Desta forma, cada contêiner pode ser utilizado para guardar as informações que estão estruturadas através de subclasses da metaclasse abstrata `FragmentDefinition`. A exemplo, a classe de tipos de informações `SimpleInformationElement` será uma dessas subclasses de representação da informação.

Por fim, a metaclasse de relacionamento `Artifact Container_Relationship` permite que cada contêiner esteja aliado a um artefato. Tal ligação também faz com que um AS, definido pela metaclasse `ArtifactDefinition` possa utilizar de muitos contêineres. Não diferente disto, a metaclasse abstrata `Fragment_Relationship` permite o relacionamento entre as informações dispostas nos fragmentos e seus contêineres e artefatos.

Comparação com abordagens atuais

Atualmente os artefatos possuem uma estrutura definida pelo tipo em que se apresenta. Caso seja um documento, este será estruturado como uma árvore hierárquica. Diferente disto, nossa abordagem permite que o mesmo artefato possua diferentes estruturas, mas devido ao fato das ferramentas atuais tornarem o preenchimento de artefatos bastante flexível, pode-se alcançar este mesmo feito a partir da abordagem atual. Entretanto, nossa abordagem define como a estrutura de um artefato deverá ser e isto ocorre independente de ferramental.

4.3.4 Cenário 4 - Classificação dos diferentes ASs

Como a idéia central para a produção de artefatos de software é armazenar informações pertinentes para concluir um produto de software específico. Entretanto, a falta de indexação pode causar problemas, tornando difícil um possível busca alguma informação necessária. Nesse contexto, basicamente não existem soluções de recuperação que não sejam ferramentas

de busca de terceiros. Mesmo assim, muitas vezes sabe-se sobre a informação, mas não se consegue recuperá-la através de buscas baseadas em palavras-chaves.

Conforme visto em (Laitinen, 1992), nesta abordagem também classificamos ASs de acordo com os tipos de conteúdo que eles conterão. Assim tornando mais completa a classificação do tipo de armazenamento a ser feito diante das informações de um artefato.

Pontos Fracos

Desta forma, sabemos que existem (i) dificuldades na classificação da informação armazenada, tendo em vista (ii) problemas de recuperação da informação por (iii) falta de organização dos ASs em um projeto.

Exemplo

A busca de informações durante o desenvolvimento de um Produto acaba por ser facilitada, desde que seja uma informação essencial ou bastante conhecida. Desta forma, caso desejássemos localizar alguma nova solicitação feita pelo *Stakeholder*, por exemplo, conseguiríamos encontrar o AS através das próprias divisões do processo, como disciplinas e atividades específicas, utilizando algum rastreamento. Em todo caso, uma vez terminado o projeto, determinar as solicitações ocorridas, quando esta informação já foi esquecida pelos antigos membros do projeto, tornar-se-á um problema.

Solução encontrada

A Figura 4.9 apresenta as metaclasses quer permitem a classificação de artefatos de acordo com a abordagem vista em (Laitinen, 1992). Como pode ser visto, a metaclasse abstrata `ClassifiedElement` possui um relacionamento com `ClassElement`, permitindo que suas subclasses sejam classificadas de acordo com a enumeração `ClassElementKind`. Todo elemento que possa ser classificado de acordo com os tipos definidos na enumeração deve ser um `ClassifiedElement`, neste caso os artefatos.

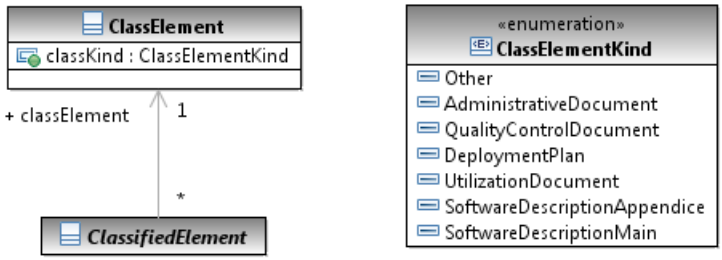

Figura 4.9: Metaclasses responsáveis pela classificação de ASs

Comparação com abordagens atuais

Classificações para informação são feitas geralmente sobre documentação e na maioria das vezes, sobre documentação pronta de um projeto. Além disso, faz-se o uso de classificações externas aos documentos, através de tabelas ou com utilização de hierarquias feitas por algum sistemas de arquivos.

Nossa abordagem provê uma implementação para o trabalho de (Laitinen, 1992) que se constitui em classificar mais do que documentos, ou seja, ASs e de forma que seja internamente, sem a necessidade de outra ferramenta ou recurso.

4.3.5 Cenário 5 - Definição de níveis de Maturidade

Partindo do princípio que artefatos são pontos de medidas para determinar o nível de maturidade de um PDS, adicionamos a definição de níveis de maturidade baseados em (Visconti & Cook, 1993). Desta forma, existe a gerência de artefatos em relação as modificações feitas durante a autoria e uso dos mesmos, verificando o quão próximo do ideal está o PDS.

Pontos Fracos

Atualmente as abordagens de maturidade são feitas com base nos Processos, inclusive PDSs. Desta forma, os ASs são analisados para determinar o nível de maturidade em que se encontram, medindo o PDS e a empresa que o adota. A grande maioria das análises é feita conforme gabarito externo aos artefatos que, mediante consulta e rastreabilidade, consegue-se determinar o quanto cada artefato está maduro dentro de um Projeto. Nestas abordagens, os pontos fracos são: (i) a forma subjetiva para determinar o nível de maturidade de cada um dos artefatos já preenchidos; (ii) a necessidade de rastrear os artefatos mediante tabela de maturidade; (iii) uso de gabarito comparativo que determina apenas o instantâneo em relação ao preenchimento dos artefatos, não a sua estrutura anterior.

Exemplo

No RUP, o artefato Visão do Negócio é um reflexo criado a partir do *Template* sugerido na fase de autoria do PDS. O objetivo do documento é capturar os objetivos relacionados a modelagem de negócios em um nível bastante abstrato.

O documento Visão do Negócio é criado na fase de Iniciação de um projeto e usado como base para o Caso de Negócio e como primeiro rascunho do documento de Visão. Por isso, está intrinsecamente relacionado ao projeto, desde o esforço de Engenharia do Software até o Caso de Negócios e o documento de Visão.

Desta forma, por ser um artefato complexo faz-se necessário verificar qual o nível de maturidade em que se encontra durante o projeto. Mesmo assim, apenas isso só poderá ser feito

após a execução de algumas atividades, por conta de necessidade de preenchimento.

De acordo com (Visconti & Cook, 1993), podemos verificar o nível de maturidade do artefato ainda em seu template, mas será medido o nível de maturidade do Processo, não sua execução.

Solução encontrada

Sendo assim, modelamos a solução vista em (Visconti & Cook, 1993), incluíndo os diversos níveis de maturidade definidos no mesmo trabalho. Na Figura 4.10 estão as metaclasses `Maturity Element`, que define elementos que possuem nívels de maturidade e `Matyrity Element Level`, que determina os indicadores e valores do elemento a ser analisado. Por fim, a enumeração de tipos `LevelTypes` determina o nível de maturidade através de classificações de acordo com a tabela de (Visconti & Cook, 1993).

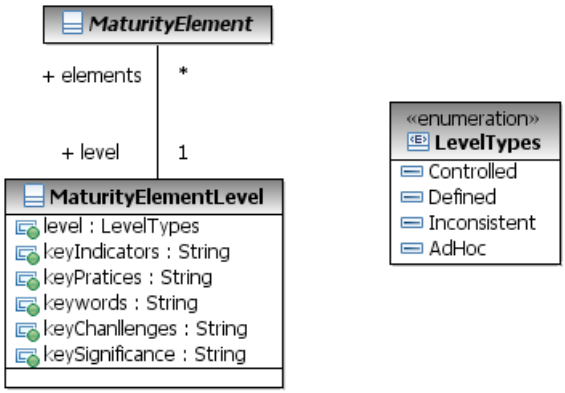

Figura 4.10: Metaclasses responsáveis pela definição de maturidade de ASs

Comparação com abordagens atuais

Com a utilização desta solução é possível definir de um modo mais granular o nível de maturidade de um projeto. Desta forma, podemos definir o quanto cada artefato nas diversas camadas do PDS, desde o *Method Content*, *Process Structure* até o *Process Enactment*.

4.3.6 Cenário 6 - Inserção de Mecanismo de Versionamento

Ao se trabalhar com informação, principalmente em grandes quantidades, são utilizados repositórios de dados, na busca por garantias de armazenamento. Muitos repositórios funcionam com base em instantâneos feitos durante períodos de tempo, garantindo que diferentes versões da mesma informação sejam guardadas.

Devido ao fato de PDSs se constituírem em três camadas (*authoring*, *tailoring* e *enactment*) o nível de dados existentes sobre um único PDS é bastante alto, necessitando de repositórios capazes de armazenas os diversos instantâneos gerados a medida que as alterações ocorrem.

Como pode ser visto na Figura 4.11, existem diferentes repositórios para armazenar as informações das diversas camadas, ou facetas, distribuídas a medida que um PDS muda de estágio.

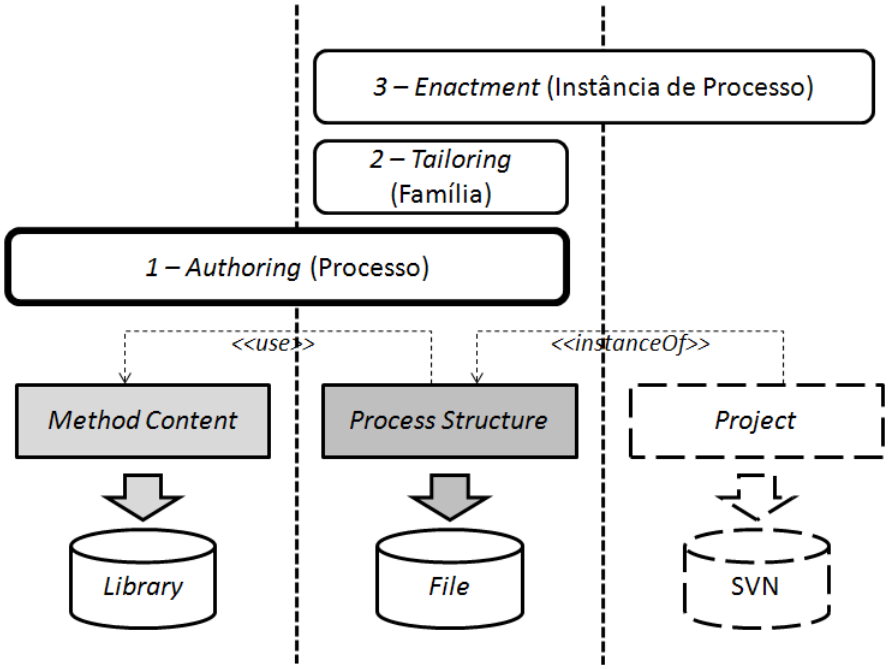

Figura 4.11: Diferentes repositórios que armazenam diferentes níveis de Informação de um PDS.

Desta forma, além de produzir histórico das informações, existe a necessidade de rastrear tais alterações futuramente. Portanto, devido a esta necessidade adicionamos esta característica aos artefatos, permitindo criação de várias versões com controle interno.

Pontos Fracos

Atualmente as informações são encapsuladas em artefatos que são guardados inteiramente nos repositórios dos projetos. Ou seja, (i) não existe identificação de que tipo de informação foi adicionada, removida ou alterada; (ii) difícil identificar como a informação foi alterada e por quem.

Diversas famílias de PDSs são criadas a partir de um PDS base. Tais famílias são modeladas em ferramentas com base em linguagens que não possuem versionamento. Desta forma, os arquivos que constituem as informações sobre os PDSs e suas famílias são armazenados por inteiro, apresentando os mesmos problemas vistos anteriormente. Além disso, nesta abordagem não (iii) é possível determinar as ligações entre o *Process Structure* e o *Method Content*.

Durante a criação do *Method Content* várias versões de Papéis, Atividades e Artefatos são feitas. Muitas vezes tais versões são alteradas, não havendo maneiras de serem reconstituídas,

por se tratarem, geralmente, de um conjunto de elementos de uma biblioteca estática. Basicamente, perde-se as versões existentes, mas mesmo que fossem versionadas (iv) rastrear as alterações e suas implicações em todas as camadas acima não é uma tarefa fácil.

Exemplos

Exemplo 1: Dado o preenchimento do artefato Modelo de Casos de Uso, do RUP, ao qual foram inicialmente definidos dois atores e três casos de uso, pode-se armazena-lo em um repositório. Mais adiante caso haja a necessidade de reparos ou atualizações, como adição de mais um caso de uso, por exemplo, este artefato será recolocado no repositório, juntamente com as informações sobre o que ocorreu. Entretanto, caso os comentários feitos pelo responsável pelas alterações não sejam suficientes, a menos que comparemos os artefato em ambas as versões nós não saberemos o que foi modificado. Além disso, a utilização de mecanismos de diferenciação é desencorajada, por não se tratar de um artefato constituído apenas de texto.

Exemplo 2: Durante a modelagem de um PDS podem existir diversas alterações e refinamentos. Dada a autoria, poderiam ser criadas duas atividades, 1 papel e 3 artefatos, juntamente com suas ligações. A partir deste momento pode-se enviar o arquivo com o modelo do PDS para o repositório. Caso seja necessário alterar a estrutura previamente definida e outro arquivo deva ir ao repositório, possivelmente existirão problemas na identificação do que foi alterado. Infelizmente, se o Engenheiro de Processos responsável pela modelagem não for bastante cuidadoso, terá que rastrear os elementos da própria linguagem de modelagem do PDS para verificar onde ocorreram as alterações.

Exemplo 3: Dado uma família de PDSs previamente definida, a qual cada PDS é derivado tornando-se uma versão do PDS Base, neste caso o RUP. Para uma melhor adaptação do PDS para um determinado domínio especifico atividades, papéis ou outros elementos do PDS Base foram excluídos, gerando a árvore de variabilidades de PDS. Sendo assim, da mesma forma que no Exemplo 2, caso não haja a informação explicita do que foi alterado, deve-se rastrear os arquivos de modelagem dos PDSs para verificar suas diferenças em relação ao PDS Base, percebendo então, as mudanças.

Exemplo 4: Existe um *Method Content* com quatro artefatos, os quais estão em primeira versão, e será necessário alterá-lo para que um dos artefatos seja dividido em dois. As alterações foram feitas e enviadas ao repositório. Após a alteração foi necessário verificar o que foi modificado, entretanto, não existe a informação de como um artefato se transformou em dois, nem ao menos qual artefato sofreu tais modificações e fora excluído. Além disso, deve ser avaliado o impacto da alteração, que ocorrerá nas camadas superiores, gerando mais problemas.

Solução encontrada

Conforme utilizaremos metamodelagem para definir o *Method Content* e *Process Structure*, escolhemos utilizar o próprio *MOF Repository* (OMG, 2007a) para gerenciar as versões nos

níveis de metamodelo e modelo.

O *MOF Repository* estende o MOF (OMG, 2006), adicionando metaclasses que representam controle de versão. Na Figura 4.12 estão as principais metaclasses responsáveis pelo versionamento. Desta forma, todos os elementos das camadas de meta-metamodelo, metamodelo e modelo poderão ser versionados e não apenas artefatos.

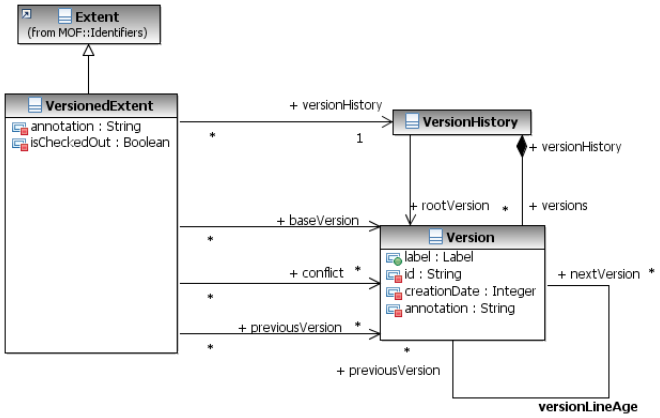

Figura 4.12: Algumas das Metaclasses do *MOF Repository*

Como se trata de uma extensão do MOF, o *MOF Repository* adiciona estas metaclasses na própria camada M3. Desta forma, qualquer linguagem definida a partir desta extensão do MOF, poderá ser versionada. Como pode ser visto na Figura 4.12 a metaclasse `VersionedExtent` herda `Extent`. No contexto do MOF, um `Extent` é responsável por aplicar algumas das capacidades a elementos do MOF, assim como gerar a identificação através de *Uniform Resource Identifier* (URI) . Através de `VersionedExtent` é possível obter versões e históricos do que foi alterado. O *MOF Repository* ainda contém metaclasses responsáveis por determinar *workspaces*, *Baselines* e definir sessões.

Comparação com abordagens atuais

Com a utilização do *MOF Repository* é possível verificar o que exatamente foi alterado dentro das camadas existentes em um PDS. Além disso, também é possível saber o papel responsável pelas alterações. Finalmente, é possível rastrear os elementos da modelagem nas diversas camadas modeladas para um PDS.

4.3.7 Cenário 7 - Relacionar os artefatos utilizando diferentes níveis de reuso e compartilhamento

Existe uma grande preocupação por parte da indústria no modo como é feito o armazenamento de informações durante a execução de um processo. Como sabemos, estas informações são guardadas nos artefatos, porém, a preocupação está em armazená-las corretamente.

Assim como em um banco de dados, quando tratamos com informações existem problemas relacionados a forma estrutural e o conteúdo, que são tratados como duas coisas diferentes.

Desta forma, para relacionar os artefatos com suas respectivas informações, de forma a reduzir a redundância e a inconsistência pode-se tentar aumentar o reuso.

Pontos Fracos

Um grande problema visto na maioria dos projetos é a manutenção dos artefatos. Seja um pequeno trecho de código alterado, que pode vir a reduzir a consistência com o modelo, ou algum documento de negócios ou de gerência, que pode vir a estar desatualizado.

Durante a execução do processo, na medida do possível tanto a estrutura quanto o conteúdo são reutilizados. Entretanto, isto geralmente é feito através de procedimentos de cópia-e-cola das informações. O que pode causar problemas de manutenção dos artefatos supracitados.

Um outro problema é a inconsistência causada por falta de comunicação, o que pode levar a redundâncias. Isto ocorre geralmente quando diferentes atores trabalham com diferentes artefatos que deveriam reutilizar estrutura ou conteúdo. Como os atores não se comunicaram, possivelmente os mesmos conceitos serão guardados de formas diferentes, e em diferentes artefatos.

Exemplos

Exemplo 2: Um bom exemplo de perda de reuso estrutural pode ser visto na configuração atual dos artefatos do RUP. A grande maioria desses artefatos possuem uma seção introdutória da seguinte maneira[5]:

```
1. Introduction
1.1 Purpose
1.2 Scope
1.3 Definitions, Acronyms, and Abbreviations
1.4 References
1.5 Overview
```

Esta estrutura poderia ser feita apenas uma vez, reaproveitanto-a em todos os artefatos que fosse necessária.

Exemplo 2: Ainda utilizando o Exemplo anterior, podemos afirmar que existe uma grande perda de reuso de conteúdo. Na Subseção 1.3 temos os definições, acronismos e abreviações que devem estar no Glossário ou algum Dicionário de Dados, ambos artefatos bastante utilizados.

```
1. Introduction
...
1.3 Definitions, Acronyms, and Abbreviations
```

[5]retirado aplicação Rational Unified Process da Rational Software

A partir do momento em que essas informações são repetidas, perde-se totalmente o reuso de conteúdo. Um problema ainda maior pode ser causado se essas informações foram alteradas no Glossário, sendo necessário uma conferência em todos os artefatos para a atualização dos dados.

Exemplo 3: Outro reuso bastante importante poderia ser feito utilizando-se as informações já existentes em um artefato, caso este fosse bem estruturado, ao invés de refazê-la. Isto também pode ser encontrado no primeiro exemplo.

```
1. Introduction
...
1.4 References
```

Todas as referências a artefatos, tanto externos quanto internos, são armazenadas na Seção Referências e assim como no Exemplo 2, qualquer alteração culminaria em uma grande perda de tempo nas atualizações, caso tudo ocorresse de forma correta.

Solução encontrada

A solução deste problema foi parcialmente resolvida na Seção 4.3.3, a partir do momento em que definimos uma maneira de organizar os artefatos e suas informações. Devido a utilização de contêineres e fragmentos de informação é possível constituir vários níveis de reuso, dependendo da necessidade. Ou seja, apenas com a utilização de contêineres pode-se agrupar as informações, fazendo com que desta forma os artefatos utilizem-se dos mesmos contêineres, evitando redundância de informação.

Especificamente, o reuso ocorre nas ligações entre artefatos e contêineres, artefatos e tipos de informações e entre contêineres e informações. Para aumentar ainda mais o nível de reuso, acrescentamos tipos para essas ligações, definido exatamente o tipo de reuso esperado. Os tipos definidos foram:

- *content*: as ligações pare esse tipo de reuso devem prover reuso tanto estrututural quando da informação contida. Este tipo deve ser utilizado apenas quando se tem certeza de que não apenas a estrutura mas o conteúdo a ser inserido será o mesmo nas diversas vezes que for utilizado. Um exemplo bastante típico é na utilização de termos ou definições de um glossário. A estrutura de um termo ou uma definição são praticamente imutáveis e podem ser reaproveitados, da mesma forma, o conteúdo das informações também serão os mesmos, não importando onde essas definições serão utilizadas.

- *extension*: as ligações desse tipo devem prover um mecamismo de reuso estrutural. Tipicamente, deve ser usado quando se deseja criar padrões reutilizáveis, para posteriormente adaptá-los para um possível domínio específico. Para facilitar o entendimento a Figura 4.13 apresenta um exemplo sobre este tipo de reuso.

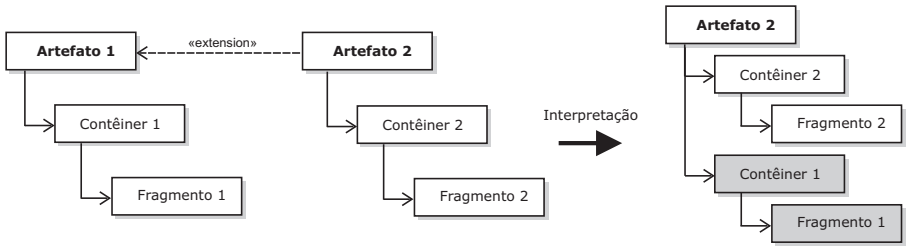

Figura 4.13: Exemplo de reuso de Artefatos utilizando *extension*

Como pode ser visto, o Artefato 2 (fonte) herda toda a estrutura existente do `Artefato 1` (alvo). Por padrão, este tipo de reuso é apenas estrutural, porém fixo, uma vez que é feita a extensão, a estrutura do elemento fonte variará conforme a estrutura herdada de seu alvo. Ou seja, se em algum momento o `Container 1` for excluído, isto será refletido no `Artefato 2`.

- *localContribution*: as ligações desse tipo devem prover um mecamismo de reuso definindo contribuições ou acréscimos especificamente locais. Este tipo de reuso pode ser utilizado quando é necessário adicionar novos sub-elementos a um conjunto de elementos existentes, Figura 4.14.

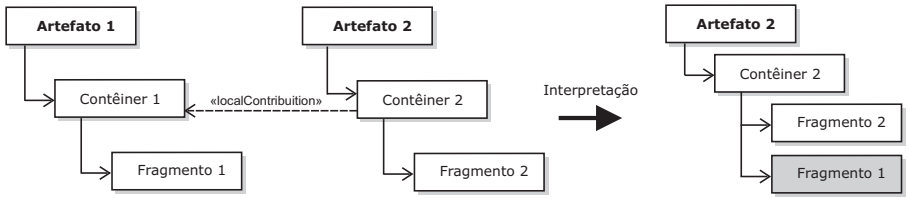

Figura 4.14: Exemplo de reuso de Artefatos utilizando *localContribution*

Como pode ser visto, o `Container 2` (fonte) é contribuído com a adição de mais um fragmento (`Fragmento 2`), que é adicionado após o `Fragmento 1`. Este tipo de ligação também provê apenas reuso estrutural, e da mesma forma como em *extension*, também é fixo.

- *localReplacement*: as ligações desse tipo devem prover um mecamismo de reuso para substituir tanto estrutura, quanto conteúdo de forma local. Com a utilização de ligações desse tipo pode-se substituir elementos existentes anteriormente a extensão, trocando-os pelos elementos herdados (Figura 4.15).

- *reference*: este é o tipo padrão utilizado para reuso estrutural de contêineres e tipos de informação. Na Figura 4.16 é apresentado o reuso de contêiner e fragmento utilizando duas ligações do tipo *reference*. Como pode ser visto, a segunda ligação não possui nenhuma indicação sobre o tipo, entretanto, por padrão é utilizado o tipo *reference*.

  Este tipo de reuso pode ser interpretado basicamente como um ponteiro para uma referência de um elemento reutilizável. Embora o `Fragmento 2` e `Contêiner 1` pareçam

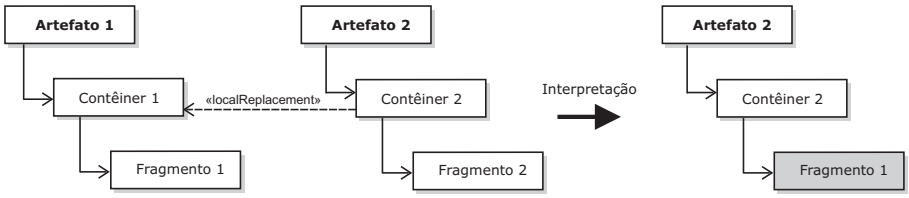

Figura 4.15: Exemplo de reuso de Artefatos utilizando *localReplacement*

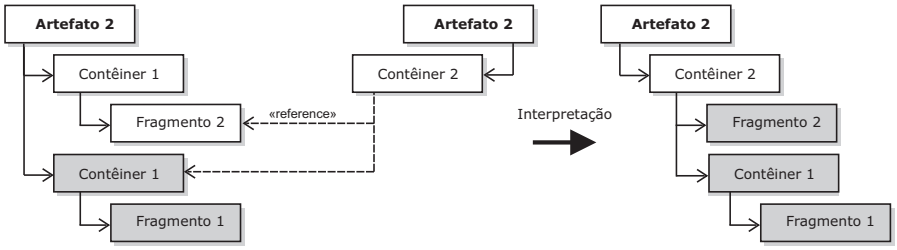

Figura 4.16: Exemplo de reuso de Artefatos utilizando *reference*

pertencer ao `Artefato 1`, estes são elementos livres, que possivelmente estão sendo reusados por ambos os artefatos.

Além disso, todos estes tipos de ligação podem ser feitos juntamente com um tipo de ligação *extension*, caso seja utilizado em um nível mais alto no que se refere à hierarquia, ou seja, em um elemento mais externo. Desta forma, o tipo de ligação mais interno sobrescreverá o *extension*.

Esta parte da nossa abordagem foi implementada conforme o metamodelo da Figura 4.17. Nesta figura estão as metaclasses responsáveis pelos relacionamentos entre:

- artefato e contêiner: `ArtifactContainer_Relationship`;
- artefato e tipo de informação: `ArtifactFragment_Relationship`, que é subclasse de `Fragment_Relationship`;
- e contêiner e tipo de informação: `ContainerFragment_Relationship`, que também é subclasse de `Fragment_Relationship`.

Assim como visto no Cenário 4 (Seção 4.3.3). Neste sentido, todas as metaclasses responsáveis pelos relacionamentos entre artefatos, contêineres e tipos de informação herdam da metaclasse `ReuseRelationship`, que possui o atributo `type`. Este atributo por sua vez é do tipo `ReuseType`, que nada mais é do que uma enumeração que contém os tipos de reuso existentes, citados anteriormente. Desta forma, ao definir qualquer um desses relacionamentos, podemos indicar qual o tipo de reúso desejado.

Comparação com abordagens atuais

Como pôde ser visto nos Exemplos 1, 2 e 3, as abordagens analisadas não contemplam o reuso de informações ja previamente utilizadas, ou seja, o reuso de conteúdo e estrutura, na maioria das vezes, é perdido, levando ao surgimento de redundância e inconsistência.

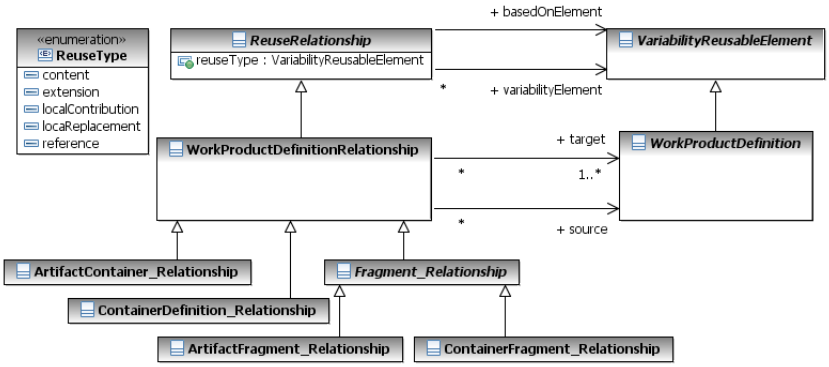

Figura 4.17: Metaclasses que definem o reuso em Artefatos de Software

Com nossa abordagem podemos reutilizar estruturas já previamente definidas. Os Exemplos 1, 2 e 3 podem ser resolvidos facilmente, utilizando as ligações de reuso. Além disso, pode-se evitar a criação de estruturas de forma repetitiva, evitando uma série de inconsistências, re-trabalho e redundância.

4.3.8 Cenário 8 - Envolver a estruturação de ASs com o PDS

No cenário atual de modelagem de PDS, papéis, artefatos e atividades estão interligados entre si. Desta forma, é possível identificar quais atividades são responsáveis por dado artefato e seus respectivos papéis. Conforme identificado em alguns PDSs, assim como RUP e OPEN, é bastante comum a utilização de sub-atividades, ou seja, dividir uma atividade em pedaços menores. Muitas vezes uma atividade chega a ser subdividida a ponto de ser tornar uma tarefa (*task*), empregada apenas um papel exclusivamente.

Além disso, também é bastante comum o desenvolvimento de grupos de papéis (*Composite Role*), que juntos contém competências, ou qualificações suficientes para atuar em atividades um pouco mais exigentes.

Assim como as tarefas são exclusivas e interligadas com papéis específicos, que muitas vezes são agrupados, um artefato também precisa se relacionar desta forma, para que as restrições de execução da atividade, em meio de tarefas e papéis, sejam correspondidas.

Portanto, diante do fato de nossa abordagem subdividir ASs, devemos também alinhar tais fragmentos e contêineres as atividades e papéis correspondentes.

Pontos Fracos

Atualmente, já que PDSs tratam ASs de forma monolítica, ou seja, inteiros e sem divisões, existem alguns problemas conforme a ligação entre os papéis atuantes e as atividades correspondentes, tais como: (i) artefatos são produzidos inteiramente, sem identificar quais pontos serão produzidos durante aquela atividade; (ii) necessária experiência e conhecimento prévio

sobre construção do artefato para seu preenchimento correto durante atividade; (iii) os papéis precisam conter maior quantidade de competência ou qualificações para executarem uma atividade dado o fato de produzirem artefatos inteiros; (iv) em um artefato não existem informações suficientes para identificar as atividades que foram responsáveis, (v) nem quais partes foram construídas.

Exemplo

A construção do artefato Visão, é feita parte a parte, ou seja, ao longo da execução de diversas atividades do PDS, neste caso o RUP. Para a execução dessas atividades são necessários diferentes papéis. A exemplo temos as atividades *Capturar um Vocabulário Comum* que requisita um *Analista de Sistemas*, *Localizar Atores e Casos de Uso de Negócios* que precisa de um *Analista de Processo de Negócios* e *Localizar Entidades e Trabalhadores de Negócios* que utiliza um *Designer de Negócios*, e assim por diante.

Como pode ser visto, um mesmo AS pode ser construído por diversos papéis diferentes, em diversas atividades diferentes. Embora a documentação do RUP apresente quais passos devem ser feitos para executar as atividades, não fica claro quais partes que constituem o artefato Visão devem ser feitas por **Quem (Papel)** e **Quando (Atividade)**. Desta forma, após o preenchimento do artefato, tais informações são praticamente impossíveis de rastrear dentro de um PDS.

Entretanto, muitas vezes a experiência adquirida em um PDS específico permite que este conhecimento seja formado ao longo do tempo. Porém, muitas vezes não se usa o mesmo processo inteiramente, além disso, suas execuções são diferentes entre si.

Solução encontrada

Para este fim o SPEM v2 foi estudado cuidadosamente, sendo identificados seus pontos de extensão. Por ser um metamodelo flexível e uma extensão da UML para desenvolvimento de um domínio específico, sua construção foi feita determinando diferentes níveis de conformidade (*compliance levels*).

Por fim, devemos integrar a extensão do metamodelo com o metamodelo original, combinando assim a autoria de ASs com a autoria de PDSs já definida pelo SPEM v2.

Como o próprio SPEM v2 já possui a conexão entre artefatos, papéis e atividades, dadas pelas metaclasses *WorkProductDefinition*/*WorkProductUse*, *RoleDefinition*/*RoleUse* e *Activity-Definition*/*ActivityUse*, para haver integração é necessário definir como será a interação entre os mesmos em relação aos conceitos acrescentados no metamodelo. Devido a extensão, artefatos e suas informações contêm tipo e estrutura interna, para adicionar este conhecimento aos papéis e atividades, deve existir uma ligação que permita tal sincronia.

Na Figura 4.18 estão algumas das metaclasses da extensão feita sobre o SPEM v2, em tons mais claros. As metaclasses `ArtifactDefinition`, `ContainerDefinition` e `FragmentDefinition`, já definidas anteriormente em 4.3.3, foram remodeladas como sub-

classes de `WorkProductDefinition`, herdando o auto relacionamento previamente definido pela metaclasse do SPEM v2 `WorkProductDefinitionRelationship`. A metaclasse abstrata `SimpleInformationElement` tambem é uma dessas subclasses, por transitividade. Desta forma, passa a existir uma conexão entre o metamodelo definido pela OMG e a extensão realizada nesse trabalho. Vale notar também, que qualquer alteração na metaclasse `WorkProductDefinition` será espelhada para suas subclasses.

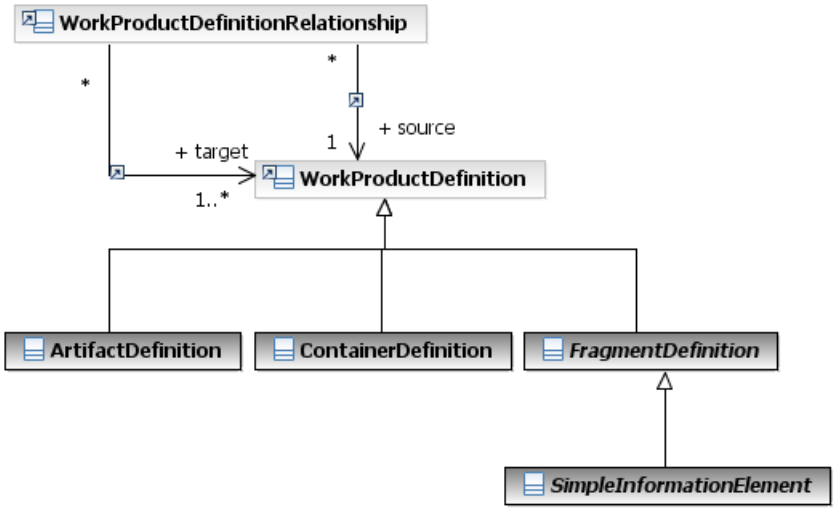

Figura 4.18: Base hierárquica de heranças que permite interação entre ASs e outros elementos do PDS. As metaclasses em tons mais claros fazem parte do SPEM v2 enquanto as de tons escuros são da extensão.

Além disso, na Figura 4.19 é apresentado um diagrama que contém a solução de modelagem para relacionar ASs com seus respectivos papéis proposta no metamodelo do SPEM v2. No metamodelo estão as metaclasses `RoleDefinition` e `WorkProductDefinition` que estão relacionadas através de `Default_ResponsibilityAssignment`. Embora apenas `ArtifactDefinition` esteja exposta no diagrama, todas as subclasses de `WorkProductDefinition` possuem o relacionamento com `RoleDefinition`.

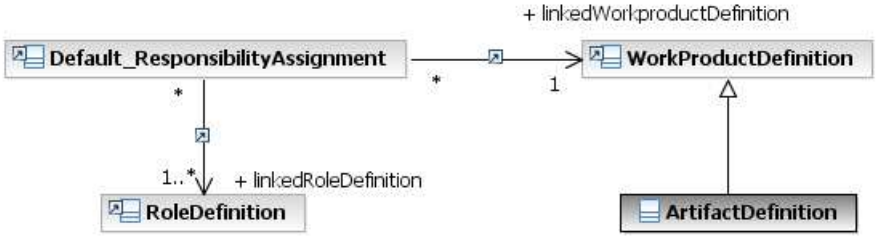

Figura 4.19: Relacionamento entre papel e artefato.

Para definir a interação entre artefato e atividade foi reutilizada a estrutura do SPEM v2. Na Figura 4.20 está definido um diagrama que representa esta ligação. Através da metaclasse

`Default_TaskDefinitionParameter` do SPEM v2 pode-se criar um elo de ligação entre as subclasses de `WorkProductDefinition` e `TaskDefinition`, que especializa `WorkDefinition`. A metaclasse `WorkDefinition` modela o conceito de trabalho a ser realizado podendo ser um sinônimo para atividade, embora mais geral.

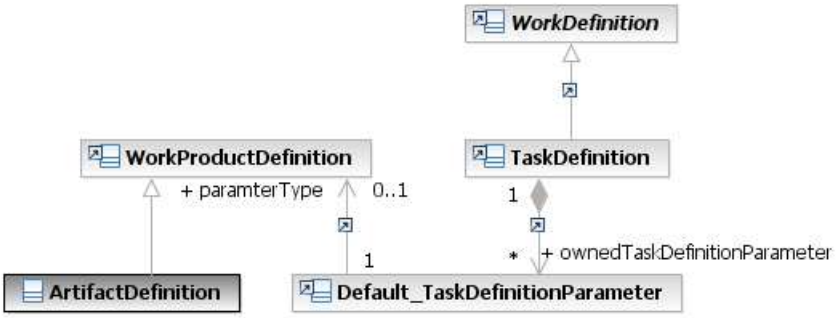

Figura 4.20: Relacionamento entre artefato e atividade.

Por fim, a Figura 4.21 apresenta as ligações existentes entre os conceitos de papel e atividade. Como pode ser observado, uma `TaskDefinition` especializa a metaclasse abstrata `WorkDefinition`, que é uma representação geral do conceito de atividade. Desta forma, existe a relação entre `TaskDefinition` e `RoleDefinition` através da metaclasse do SPEM v2 `Default_ResponsibilityAssignment`.

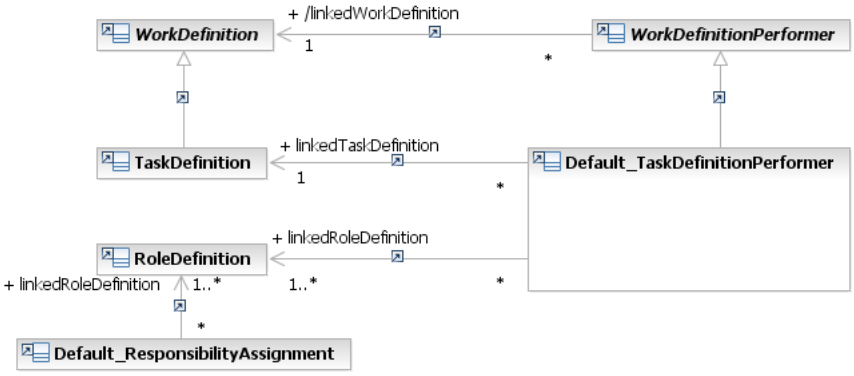

Figura 4.21: Relacionamento entre papel e atividade do SPEM v2.

Comparação com abordagens atuais

Desta forma, tanto a estrutura da informação, quanto a estrutura do artefato devem possuir relacionamentos com papéis e atividades. A partir deste ponto pode ser visto que é possível determinar, exatamente, o que cada papel será responsável em produzir, conforme ação específica a ser feita em uma atividade que deva manipular um artefato.

4.4 Uso dos Artefatos

Na Seção anterior houve a especificação da Definição dos ASs, tornando possível utilizar uma linguagem para a permitir a Autoria de ASs, conforme essa especificação. Entretanto, apenas definir como deve ser um artefato não basta para que ele possa estar em um PDS. Para solucionar este problema devemos adicionar uma camada que permite a utilização dos artefatos pelos PDSs, sem alterar suas definições. Portanto, para que exista uma autoria completa, apresentaremos nessa Seção como se deu a construção da camada de ligação entre a definição dos artefatos e a definição da estrutura de PDSs, conhecida por *Process Structure*, ou simplesmente por **uso**.

Assim como na última Seção, utilizaremos uma estrutura comparativa entre este paradigma e o atual, mostrando: pontos fracos do paradigma atual, a nossa solução e, por fim, a comparação entre as soluções.

4.4.1 Pontos Fracos

Dentro da abordagem tradicional, mais especificamente nos processos avaliados (Scrum e RUP), notamos a utilização constante de *templates*, exemplos e guias de utilização. Os *templates* são na maioria das vezes esqueletos dos artefatos originais, publicados de forma que sejam auto-explicativos. Além disso, notamos que os *templates* são utilizados para definir como devem ser artefatos do tipo documentos e dificilmente são feitos *templates* para outros tipos de artefatos, embora seja possível.

Nesse sentido, existe um grande problema: já que nem todos os artefatos possuem *templates*, como pode-se utilizá-los, já que não existe a própria definição?

Ou seja, o maior problema está na parte de utilização desses *templates*. Embora sejam documentos que indicam a estrutura de seções e como devem ser preenchidos, não há uma clara definição de certo ou errado quanto aos tipos de informação. Para tentar corrigir esta situação, são utilizados exemplos, guias e ferramentas específicas, porém, o sucesso depende puramente da interpretação do responsável pelo preenchimento do artefato.

Além disso, os *templates* nem sempre são utilizados por completo, ou seja, neste paradigma não ficam claras as alterações feitas no que se diz respeito a utilização dos *templates* (i.e. remoção, alteração, adição de definições), durante a criação do PDS.

A construção do artefato Visão, é feita parte a parte, ou seja, ao longo da execução de diversas atividades do PDS, neste caso o RUP. Para a execução dessas atividades são necessários diferentes papéis. A exemplo temos as atividades *Capturar um Vocabulário Comum* que requisita um *Analista de Sistemas*, *Localizar Atores e Casos de Uso de Negócios* que precisa de um *Analista de Processo de Negócios* e *Localizar Entidades e Trabalhadores de Negócios* que

utiliza um *Designer de Negócios*, e assim por diante.

4.4.2 Solução encontrada

Neste contexto, a solução que foi utilizada é a mesma encontrada na especificação do SPEM v2. O metamodelo da OMG já traz a noção de separação do conteúdo do Processo, de sua estrutura. Desta forma, os artefatos podem ser utilizados através do padrão *Proxy* (Gamma et al., 1993), conforme a Figura 4.22.

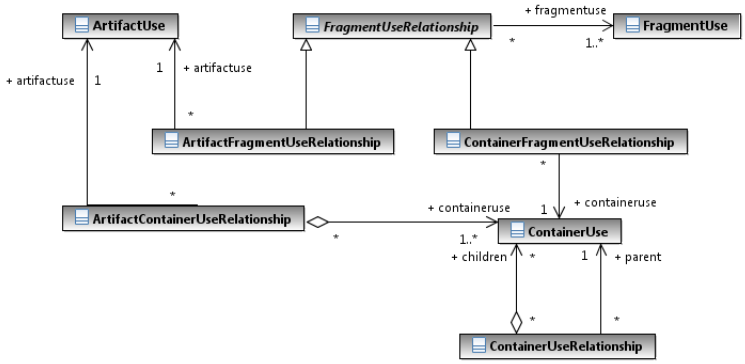

Figura 4.22: Diagrama com o Metamodelo para estrutura do uso de ASs

Observando este último diagrama, pode-se notar uma semelhança com o metamodelo apresentado na Figura 4.8. Isso não é por acaso, muito pelo contrário, o metamodelo exposto na Figura 4.22 também possui metaclasses. Desta forma é possível que as metaclasses modeladas para o uso de ASs possam se utilizar das metaclasses de definição. O uso se caracteriza por um relacionamento simples, do tipo **m:1** opcional e direcionado, ou seja, * para 0..1, Como pode ser visto na Figura 4.23. Além disso, as metaclasses `ArtifactUse`, `ContainerUse` e `FragmentUse` aparecem duas vezes: as metaclasses em cor mais escura, representam as metaclasses originais e não possuem ligação com a definição. Já as metaclasses oriundas da camada de ligação entre definição e uso, devem possuir estes relacionamentos. Por fim, as metaclasses da camada de defingção desconhecem tais relacionamentos, pois são direcionados e exclusivos da camada de uso.

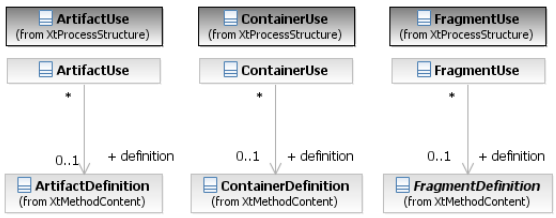

Figura 4.23: Diagrama com o Metamodelo com uso de ASs

4.4.3 Comparação com abordagens atuais

Em relação as abordagens atuais, nossa abordagem permite entender o relacionamento entre a definição dos ASs e o seu uso em PDSs. As abordagens atuais não permitem avaliar este relacionamento, pois não utilizam o conteúdo separado da estrutura, sendo assim, dificilmente conseguem estabelecer ligação entre a instância do artefato e seu *template*.

Em nossa abordagem o uso dos artefatos é feito a partir de uma definição, ou seja, **todos** os artefatos obrigatoriamente terão uma definição **única**. Por isso é possível saber qual foi o **tipo de uso**, percebendo se houve alguma alteração a partir do próprio modelo. Atualmente nem todos os artefatos possuem definição e mesmo o que são providos de *templates* ou até mesmo exemplos, dependem de interpretação de definições feitas em linguagem natural.

4.5 Metamodelo e *UML Profile*

Neste trabalho a extensão da UML e do SPEM foram feitas através do uso de dois mecanismo de extensão: *lightweight* e *heavyweight*.

A extensão feita a partir do mecanismo *lightweight* permite que os modelos gerados sejam utilizados por diversas ferramentas CASE já existentes. Como este mecanismo não altera o metamodelo da UML, é favorecida a troca de dados.

O próprio SPEM possui um *UML Profile* para em modelos UML. Desta forma, adicionamos alguns outros estereótipos ao catálogo, extendendo o *UML Profile* do SPEM para permitir a modelagem de artefatos de software no nível de usuário da UML. Neste trabalho, esta extensão é conservativa, ou seja, mantem os estereótipos existentes.

No Apêndice A.8 são apresentados os estereótipos adicionados ao *UML Profile* do SPEM, criando o *SPEMXt UML Profile* (Seção B.2), e os diagramas que apresentam a ligação entre os estereótipos adicionados e os já existentes (Seção B.1).

Já a extensão feita a partir do mecanismo *heavyweight* impossibilita a troca de dados entre ferramentas que implementam o metamodelo padrão da UML. Geralmente, uma extensão deste tipo também é acompanhada de ferramentas que suportem as alterações feitas. Entretanto, a troca de dados só poderá ser feita através dessas ferramentas específicas. Mesmo assim, a utilização desse tipo de extenção permite a criação de semântica muito mais rica do que *UML Profile*, além disso, as restrições podem ser mais elaboradas, permitindo a inserção de regras em linguagens diversas, assim como o *Object Constraint Language* (OCL) (Warmer & Kleppe, 2003) .

Para a implementação da nossa abordagem estendemos tanto a UML quanto o SPEM v2. Embora o metamodelo proposto possua mais de 100 metaclasses, nas seções 4.3 e 4.4 foram apresentados as metaclasses principais, constituindo o *cerne* deste trabalho. Para maiores deta-

lhes, em é apresentado em detalhes e por completo o metamodelo *SPEMXt*, que é a extensão do SPEM v2.

4.6 Guia para Autoria de Artefatos de Software

Diante das análises dos paradigmas atuais da Engenharia de Software, não foi encontrado consenso sobre as atividades e as particularidades inerentes a autoria de PDSs, muito menos uma descrição ou esboço das características de soluções em autoria de ASs. Como conseqüência, trabalhos que abordem o tema precisam de um grande investimento de tempo e esforço.

Com o objetivo de modificar tal realidade, foi analisado um conjunto de trabalhos visando fazer um levantamento dos requisitos básicos, das atividades e das necessidades pertinentes à autoria de PDSs. Feito isto, foi elaborado um guia que tem como principal objetivo auxiliar a construção de bibliotecas de PDSs e ASs.

Desse modo, o guia aqui elaborado especifica um fluxo de atividades a ser seguido a fim de obter um Processo de Software que utilize Artefatos de Software conforme a abordagem explicitada nesta dissertação. Além disso, o guia visa representar boas práticas e tornar tão compreensível quanto possível os papéis necessários para a concretização da autoria.

Na Figura 4.24 é apresentado um Diagrama de Atividades que representa um fluxo inicial sobre Processos de Software. Este fluxo é baseado no Ciclo de Vida para PDSs visto na Seção 2.2. Como pode ser visto, após ser iniciado, o fluxo se encaminha para uma tomada de decisão, a pergunta é bastante simples: Qual fase quer prosseguir? Neste ponto pode-se escolher entre (i) reconfigurar um PDS já existente, adaptá-lo para um domínio específico ou criar família de processos para soluções diversas; (ii) executar um PDS já anteriormente criado, concebendo um novo projeto; ou (iii) **criar um novo PDS**, que é o foco deste trabalho.

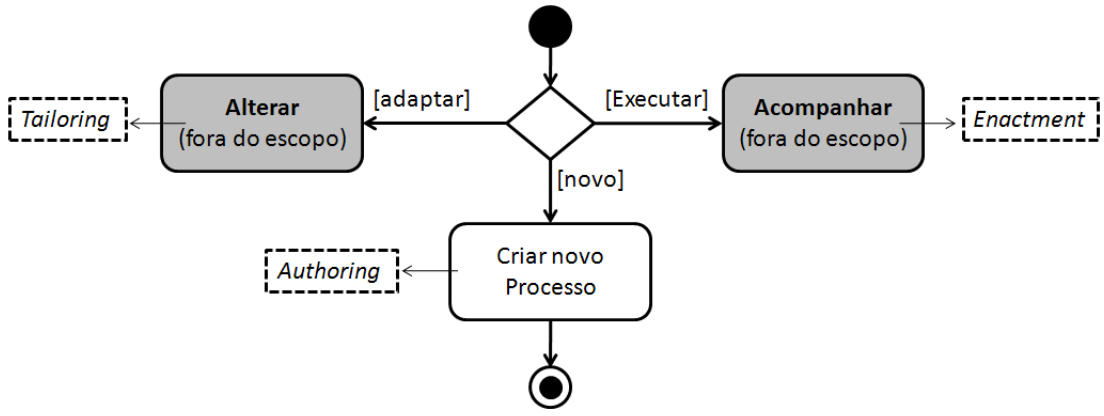

Figura 4.24: Fluxo de atividades inerentes ao ciclo de um PDS

A fase de criação de um PDS pode vir a ser bastante longa, consistindo na adição de elementos pertencentes ao processo, formando uma autoria. Por existirem muitas atividades inerentes

a autoria de PDSs, elas foram agrupadas em quatro macro-atividades, Figura 4.25:

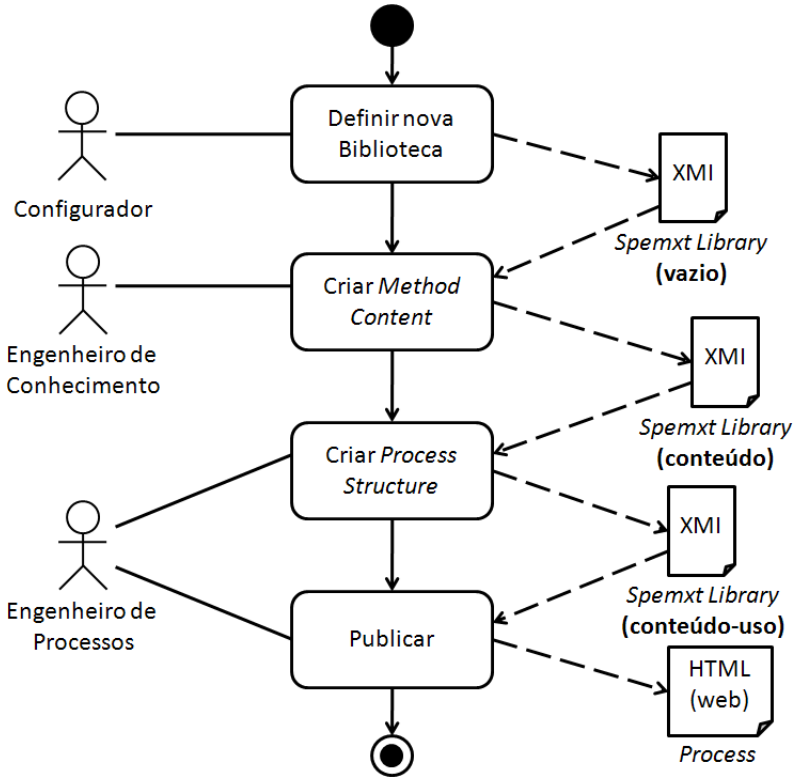

Figura 4.25: Guia para autoria de PDSs

- **Definir nova Biblioteca:** aqui estão agrupadas as atividades responsáveis por definir uma biblioteca de configuração. Esta biblioteca permitirá a criação de conteúdos e estruturas de PDSs.
- **Criar *Method Content*:** aqui estão agrupadas as atividades responsáveis por definir o conteúdo. Desta forma pode-se utilizar o conteúdo previamente definido para construir a estrutura de até mesmo uma família de PDSs.
- **Definir *Process Structure*:** aqui estão agrupadas as atividades responsáveis por definir a estrutura de um PDS. Tal estrutura especifica exatamente como deve ser o uso de um conteúdo prévio, fazendo ligações e relacionamentos com os elementos que constituintes da biblioteca de conteúdo.
- **Publicar Processo:** nesta última macro-atividade estão agrupadas atividades afins no contexto de publicar o PDS criado a partir das macro-atividades anteriores.

Além disso, a Figura 4.25 também apresenta os papéis responsáveis pela execução das atividades supracitadas, assim como os artefatos dos quais são responsáveis:

- **Configurador:** o ator deste papel deve ser responsável por configurar todo o **Ambiente de Autoria de Processos**. Desta forma, ele deverá produzir o artefato **Biblioteca** (*Spemxt*

Library) e prepará-lo para receber os dados das próximas atividades. Neste momento o *Spemxt Library* encontra-se vázio no que se refere a PDSs.

- **Engenheiro de Conhecimento:** o ator deste papel deve ser responsável por criar conteúdo reutilizável pelo processo, definido elementos como papéis, atividades e artefatos. O conteúdo produzido deverá entrar no *Spemxt Library*.
- **Engenheiro de Processos:** o ator deste papel deve ser responsável por criar a estrutura do Processo reutilizando os elementos produzidos pelo Engenheiro de Conhecimento. O Engenheiro de Processo deve ser capaz de combinar os diversos elementos existentes no conteúdo para criar um Processo geralmente específico para um domínio. Toda a estrutura produzida, assim como os relacionamentos criados para se utilizar os elementos do conteúdo deverão entrar no *Spemxt Library*. Além disso, o Engenheiro de Processo ainda deve publicar o PDS para que este possa ser utilizado, criando o Artefatos diversos, geralmente em páginas HTML.

Após descrever esta visão geral sobre o fluxo de atividades que descrevem o guia para autoria, apresentaremos as macro-atividades com maiores detalhes nas Seções a seguir.

4.6.1 Definir nova Biblioteca

Primeiramente, para criar uma nova biblioteca deve-se definir o espaço necessário para o armazenamento dos dados. Conforme nossa abordagem, toda a biblioteca será definida em um arquivo, entrentanto, deve-se ter em mente que podem ser utilizadas outras tecnologias, assim como banco de dados.

Logo após sua criação, a biblioteca deve ser nomeada. Além disso, também é interessante, mas não obrigatório, deixar breves descrições para que ela possa ser identificada mais adiante, preparando-a para o uso posterior. Conforme a necessidade de identificação, o nome, ou talvez um identificador externo deve ser único, evitando problemas de ambiguidade. O fluxo de atividades para a criação de uma nova Biblioteca pode ser visto na Figura 4.26.

4.6.2 Criar *Method Content*

Continuando o fluxo do Guia, o próximo passo é criar o *Method Content*, ou seja, definir o conteúdo dos PDSs. O *Method Content* possui diversos elementos a serem definidos, assim como papéis, atividades, tarefas e ferramentas. Desta forma, variando conforme o que será utilizado para a construção do PDS, pode-se obter diferentes elementos em um *Method Content*. Conforme o objetivo deste Guia, mostraremos apenas o fluxo para a construção de elementos de definição de ASs, visto na Figura 4.27.

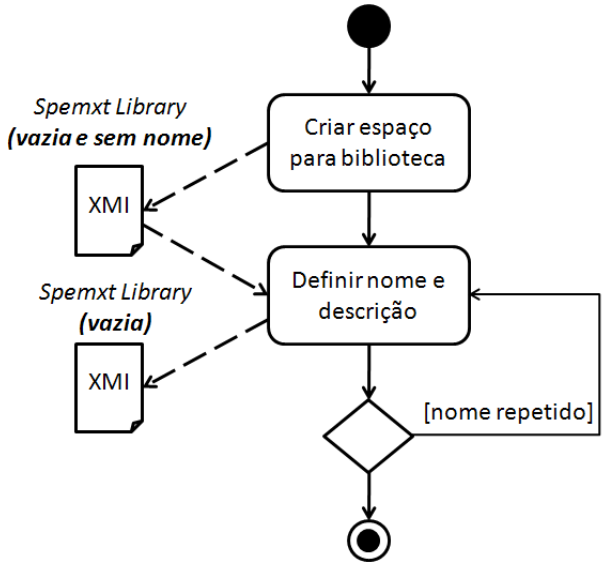

Figura 4.26: Fluxo de atividades inerentes ao ciclo de um PDS

O fluxo de atividades visto na Figura 4.27 apresenta o fluxo para a criação de Artefatos de Software e, como pode ser visto, a construção depende da criação de Tipos de Informação e Contêineres. Na primeira atividade de criação são adicionados tipos de informação ao *Method Content*. Posteriormente são adicionados os contêineres, reponsáveis por organizar os tipos de informações, agrupando-os. Por fim, a definição de artefato é criada através da seleção dos tipos de informações e contêineres desejados.

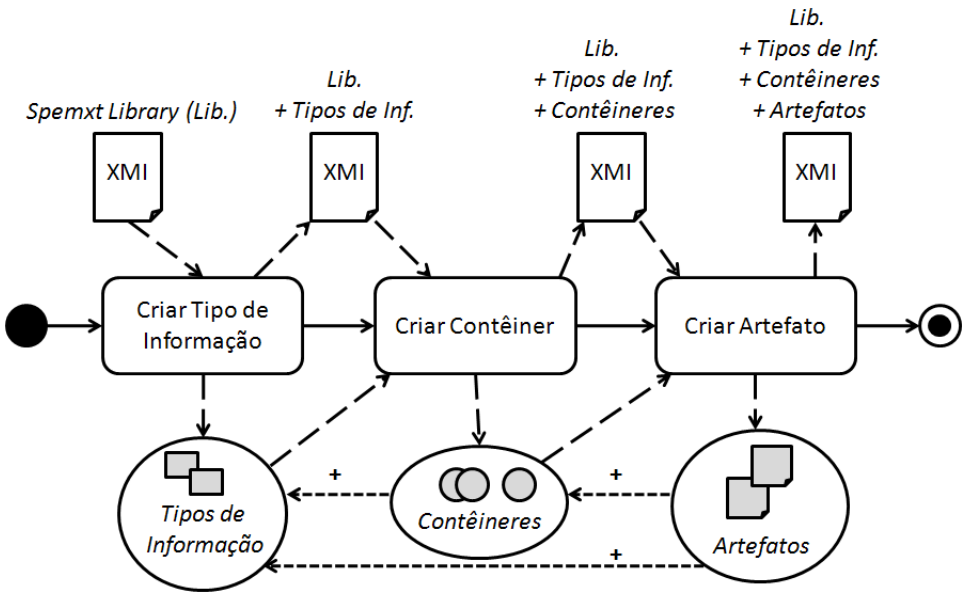

Figura 4.27: Fluxo de atividades para Criação de ASs.

A construção de ASs não possui um fluxo complexo, precisando apenas de dois, no máximo três, passos para ser concluída. A definição de ASs deve ser feita de forma detalhada, por isso é importante utilizar os contêineres, agrupando as informações quando necessário, porém,

isto não é obrigatório. Na Figura 4.28 é apresentado o diagrama que explica como configurar os artefatos conforme nossa abordagem. Esta configuração só pode ser feita quando os conteineres e tipos de informações a serem utilizados já estiverem criados. Conforme esse diagrama, deve-se inserir os contêineres selecionando também o tipo de relacionamento entre eles e o artefato.

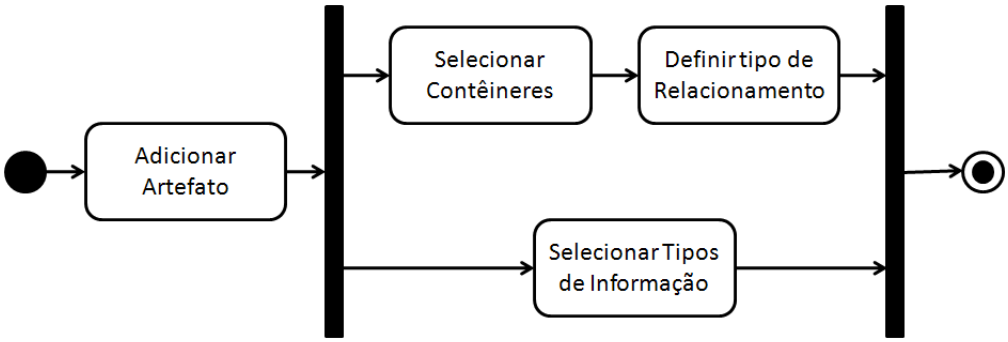

Figura 4.28: Fluxo de atividades para Definição do ASs em detalhes.

4.6.3 Definir *Process Structure*

Após criar o *Method Content*, segundo o Guia, já é possível incluir o *Process Structure*. Sendo assim, o próximo passo é criar a estrutura dos PDSs. Esta atividade é quase exclusiva, pois deve ser feita para cada PDS existente. O *Process Structure* também possui diversos elementos a serem definidos, na verdade, ele funciona como um espelho do *Method Content*, refletindo todos os elementos existentes no conteúdo, assim como papéis, atividades, tarefas e ferramentas, com exceção de que, esses elementos só deverão existir, caso sejam necessários para a estruturação do PDS. Conforme o objetivo deste Guia, mostraremos apenas o fluxo para a construção de elementos de definição de ASs, que pode ser visto na Figura 4.29.

O fluxo de atividades visto na Figura 4.29 apresenta o fluxo para o uso da definição de ASs. Após a seleção da definição de um ou mais artefatos (primeira Atividade), todo o resto deve ser também utilizado, ou seja, o uso dos contêineres (segunda Atividade) e tipos de informação (última Atividade) devem ser feitos em cascata. Durante a utilização dos elementos internos dos ASs, pode-se também selecionar o tipo de uso que deve ser feito. Esta característica permite que o uso de um artefato possa ser diferente de sua definição, desde que este seja um subconjunto. Ou seja, pode-se em tempo de uso desistir de alguns Tipos de Informação ou mesmo Contêineres, dependendo do que se desejar. Este tipo de utilização, embora seja possível neste momento é muito mais comum durante a fase de *tailoring*, adaptação e geração de famílias de processos.

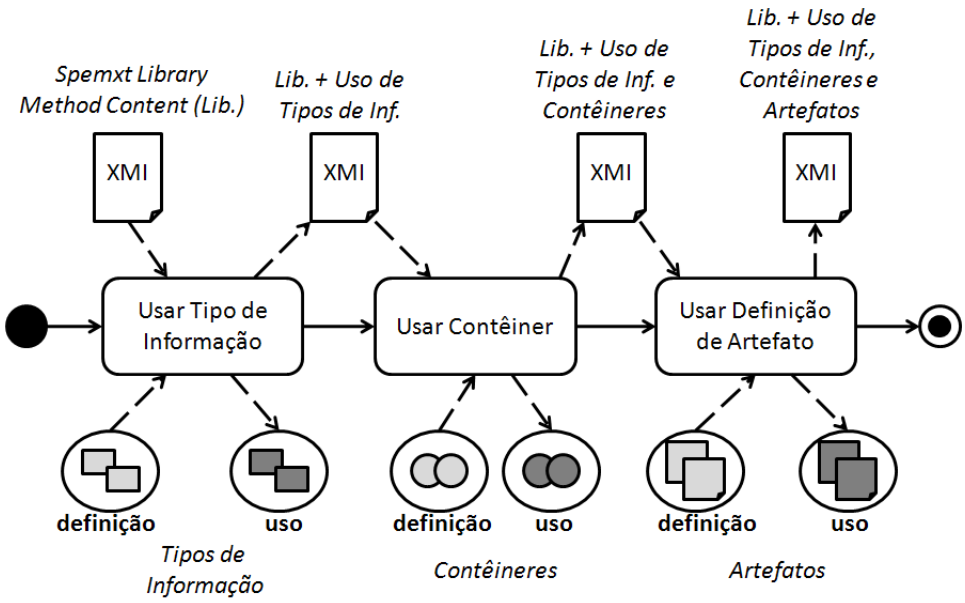

Figura 4.29: Fluxo de atividades para o Uso da Definição de ASs.

4.6.4 Publicar Processo

A última atividade do Guia é reponsável por exportar os dados armazenados na biblioteca. Como foi dito anteriormente, esses dados são armazenados em um arquivo XMI preparado para troca posterior. Porém, permitir o uso apenas de arquivos do tipo XMI não é suficiente para que exista troca de dados entre outras ferramentas. Ou seja, deve ser possível de se utilizar a biblioteca, ou ao menos os dados que ela possui, a partir de outras ferramentas, principalmente ferramentas comuns ao uso na indústria. Desta forma, a publicação permite troca de dados para diferentes ferramentas, mesmo que estas não utilizem arquivos do tipo XMI. Para a quantidade de ferramentas capazes de utilizar os dados contidos na biblioteca, pode-se exportá-los em arquivos do tipo HTML ou XHML, através da especificação padronizada pela W3C [6]e OOXML , padrão Microsoft para *Office Open XML's vocabularies and document representation and packaging* [7], entre outros.

Diante deste contexto, o Guia criado permite a a publicação a partir do momento em que se define qual **o tipo de arquivo** a ser exportado, partindo então para uma decisão sobre **o que deve** ser publicado. Para isto, deve-se escolher entre dois níveis de publicação: o primeiro nível (a) diz respeito a publicação do PDS como um todo; já o segundo nível (b) permite a publicação apenas de ASs, independente do PDS. Ambos os níveis possuem uma tomada de decisão sobre o que exatamente deve ser publicado. Conforme a Figura 4.30, as atividades referentes ao nível (a) permitem uma escolha entre *Method Content*, *Process Structure* ou toda a *Spemxt Library*. Desta forma pode-se extrair todos os dados existentes. O fluxo de atividades em (b) servem

[6]http://www.maujor.com/w3c/xhtml10_2ed.html
[7]http://www.ecma-international.org/publications/standards/Ecma-376.htm

para a publicação exclusiva de ASs, ecolhendo-se entre sua **definição**, o seu **uso**, ou **ambos**.

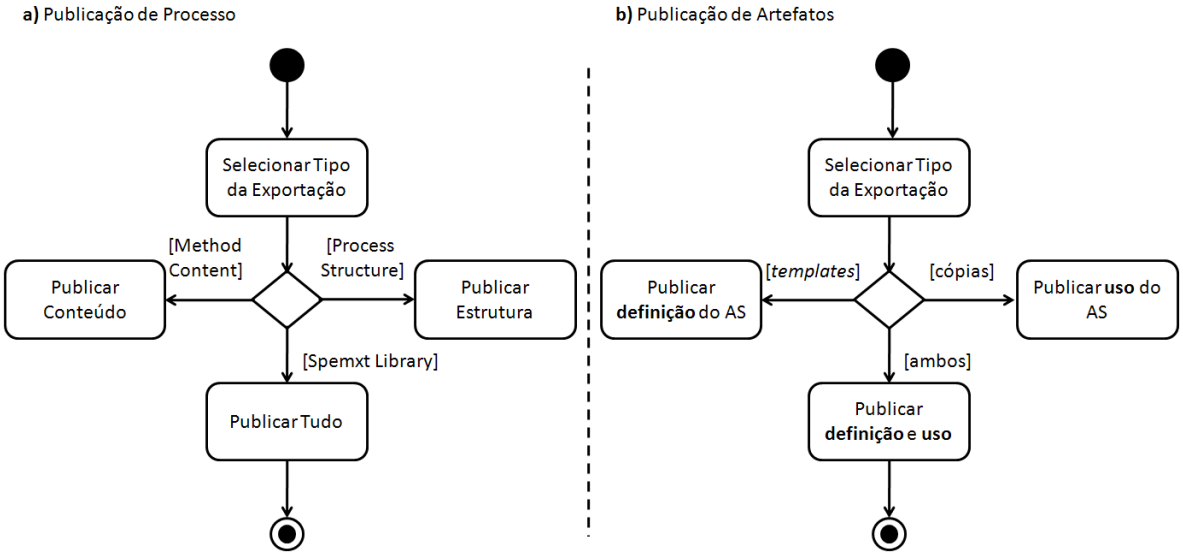

Figura 4.30: Fluxo de atividades para Publicação.

4.7 Regras para Autoria de Artefatos de Software

Em nossa abordagem, assim como na UML, utilizamos os termos *source* e *target* para definir os elementos participantes de um relacionamento: *source – has –> target*.

4.7.1 Regras Estruturais

As regras estruturais servem para garantir o valor semântico do domínio de autoria de artefatos, no tocante a utilização do metamodelo.

A regra estrutural fundamental é: **Regra E0:** nenhum auto-relacionamento deve ser feito entre a mesma instância. Evita-se então que um elemento seja “pai” ou “filho”, diferenciando-se um *target* de um *source*.

- **Regra E1:** um artefato deve possuir ao menos “um tipo de informação” ou “um container”. Ou seja, não devem existir artefatos vazios.
- **Regra E2:** os relacionamentos entre artefatos só podem ser feitos através da metaclasse *WorkProduct_Relationship*.
- **Regra E3:** os relacionamentos entre contêineres só podem ser feitos através da metaclasse *ContainerDefinition_Relationship*.

- **Regra E4:** os relacionamentos entre tipos de informação só podem ser feitos através de seus relacionamentos internos, definidos em nível M2.

- **Regra E5:** os relacionamentos entre artefatos e contêineres devem ser feitos através da metaclasse *ArtifactContainer_Relationship*.

- **Regra E6:** os relacionamentos entre artefatos e tipos de informação devem ser feitos através da metaclasse *ContainerFragment_Relationship*.

- **Regra E7:** os relacionamentos entre contêineres e tipos de informação devem ser feitos através da metaclasse *ArtifactFragment_Relationship*.

4.7.2 Regras para Reuso

Existem **cinco** níveis de reuso definidos na Seção 4.3.7: *content*, *extension*, *local contribution*, *local replacement* e *reference*.

A regra fundamental para a utilização dos conceitos de reuso é: **Regra R0:** nenhum dos níveis de reuso pode ser aplicado mais de uma vez entre os mesmos elementos. Isto é expressado em OCL da seguinte forma:

I - Quanto a utilização de apenas um nível em uma mesma hierarquia as regras definidas são:

- **Regra R1:** o reuso do tipo *content* deve reutilizar tanto a estrutura quanto o conteúdo. Desta forma, ao se aplicar esse tipo, o elemento *source* do relacionamento deverá adquirir toda a estrutura do elemento *target*, assim como todo o seu conteúdo.

- **Regra R2:** o reuso do tipo *content* não pode ser utilizado entre um elemento *source* Contêiner e um elemento *target* Artefato.

- **Regra R3:** o reuso do tipo *content* não pode ser utilizado entre um elemento *source* Tipo de Informação e um elemento *target* Artefato.

- **Regra R4:** o reuso do tipo *content* não pode ser utilizado entre um elemento *source* Tipo de Informação e um elemento *target* Contêiner.

- **Regra R5:** o reuso do tipo *content* deve reutilizar a estrutura. Desse modo, ao se aplicar esse tipo de reuso, o elemento *source* do relacionamento deverá adquirir toda a estrutura do elemento *target*.

- **Regra R6:** o reuso do tipo *extension* deve reutilizar toda a estrutura interna do elemento que está sendo reutilizado.

- **Regra R7:** o reuso do tipo *extension* somente pode ser utilizado entre elementos de mesma metaclasse.

- **Regra R8:** o reuso do tipo *local contribution* deve reutilizar a estrutura interna do elemento a ser reutilizado. Desta forma, um elemento *source* obterá a mesma estrutura hierárquica de seu *target*, aliada a estrutura que já possuia anteriormente.

- **Regra R9:** no reuso *local contribution*, todo elemento a ser reutilizado será sempre adicionado após o último elemento da mesma hierarqui no *source*.

- **Regra R10:** no reuso *local contribution*, todo elemento a ser reutilizado será sempre adicionado após o último elemento da mesma hierarqui no *source*.

- **Regra R11:** no reuso *local contribution*, caso haja conflito de reuso por motivos de existência do mesmo elemento em mesmo nível de hierarquia, este elemento não será afetado.

- **Regra R12:** no reuso *local replacement*, os elementos internos do elemento *target* deverão substituir os elementos internos de um *source*.

- **Regra R13:** no reuso *reference*, um elemento *source* apenas referencia um elemento *target*, entretanto não é criada estrutura física.

II - Quanto a utilização de mais de um nível em uma mesma hierarquia as regras definidas são (estension pode ser sobreescrito. vide reconstrucao da regra R5):

- **Regra R14:** o reuso do tipo *extension*, quando aplicado, deve ser feito de forma que englobe o elemento de maior nível hierárquico (mais próximo da raiz possível). Pois, não faz sentido aplicá-lo mais de uma vez numa mesma hierarquia.

- **Regra R15:** quanto a utilização de mais de um o reuso do tipo *extension* não deve aplicado de forma fique em nível hierárquico inferior a qualquer outro tipo de reuso.

- **Regra R16:** o reuso do tipo *extension* pode ser sobrescrito por todos os tipos de reuso, com exceção de *reference*.

4.8 Os Níveis de Formalismos e Pontos de Conformidade

No que se constitui a nossa abordagem para a Autoria de Artefatos de Software, foram definidas algumas características a serem atendidas. Tais características foram elaboradas para possibilitar uma melhoria em relação as propostas já existentes e analisadas no Capítulo 3.

Dessa forma, visamos construir uma série de conceitos para atingir o nosso objetivo de estrutura artefatos, no tocante a conseguir introduzir tais características de maneira tal que não comprometesse esse trabalho. Haja visto, foram abordados os seguintes conceitos:

- **Paradigma de construção** - em nossa abordagem os artefatos são vistos como uma união de fragmentos de informação definidos através de estrutura bem definida. Este paradigma se opõe ao paradigma atual, onde artefatos são elementos monolíticos, auto-contidos e fechados. A escolha por um paradigma praticamente oposto ao utilizado na maioria dos PDSs atuais, foi feita com base na nossa hipótese (**H1**) de que o controle das informações de um artefato estruturado pode ser melhorado, em relação a um artefato monolítico, no tocante a estruturação, categorização, organização, controle de versão e reuso;

- **Guia de construção** - foram definidos diagramas com fluxos de atividades mostrando quais atividades, etapas ou passos são necessárias para se realizar a autoria de artefatos. Embora nenhuma outra abordagem analisada apresente algo nesse sentido, acreditamos (hipótese **H2**) que a utilização de um Guia permite construções mais bem definidas, pois o guia representa um processo a ser seguido, diferentemente de produção *ad hoc*;

- **Escopo da abordagem** - a abordagem permite a construção de quaisquer tipos de artefatos, desde que eles estejam no conjunto dos artefatos analisados com base nos processos RUP e SCRUM. A modelagem de outros tipos de artefatos, vistos outros diferentes processos, não é garantida;

- **Separação de Conteúdo** - assim como no SPEM v2, nossa abordagem possui uma clara divisão entre a estrutura do processo (*Process Structure*) e o seu conteúdo (*Method Content*). Ou seja, a hierarquia de atividades, ou disciplinas, definidas na maioria das vezes por estruturas do tipo *Work Breakdown Structures* (WBS) está separada da definição dos papéis, atividades e artefatos, entre outros elementos de PDS. Com base nesse conceito definido no SPEM v2, acreditamos na hipótese (**H3**) de que separando um PDS nesses dois níveis é possível se obter adaptabilidade de processos. Desta forma, pode-se utilizá-la como princípio para estruturar os conceitos de reuso de estrutura e conteúdo, tanto de artefato quanto de suas informações.

Em relação a representação da nossa abordagem, ao optarmos por desenvolver uma linguagem própria, verificamos que seria necessário atender a uma série de requisitos, principalmente com relação a UML. Por isso, da mesma forma como foi definido na UML, utilizamos a técnica de especificação da sintaxe da linguagem, para só depois definir sua semântica. Temos por base as convenções e tipologia definidas em (OMG, 2007b). Nosso metamodelo compreende os seguintes níveis de formalismo:

- **Sintaxe:** define quais os contrutores existentes na linguagem, mostrando como podem ser construídos a partir de outros construtores. Foram utilizados dois níveis de sintaxe:

1. **Sintaxe Abstrata**, que se constitui em uma representação gráfica, funcionando como uma linguagem de notação independente.
2. **Sintaxe Concreta**, que é definda através de um mapeamento da notação definida pela sintaxe abstrata, tornando tal sintaxe computável e formalizada.

Como pode ser visto no Apêndice , nossa abordagem possui um metamodelo completo que utiliza a mesma notação da UML. Além disso, na Seção 5.1 é apresentada a linguagem concreta, feita em XMI e derivada da utilização do MOF.

- **Semântica:** define os significado dos construtores utilizados na linguagem, bem como o que representam. Em nossa abordagem foram definidos três níveis de semântica:

  1. **Semântica Estática:** apresenta valores semânticos da linguagem definido como as instâncias dos construtores devem ser conectadas entre elas, apresentado as formas possíveis de utilização para se conseguir algum sentido;
  2. **Semântica Dinâmica:** descreve o sentido de cada construtor da linguagem;
  3. **Regras de boa formação:** este último nível de formalismo é acrecestado a linguagem pois traz consigo regras que definem a boa construção dos ASs. No Apêndice é apresentado o valor semântico de cada construtor do metamodelo nos três níveis existentes, quando aplicável.

O UML Profile definido nesse trabalho e que está apresentado no Apêndice A.8 possui os níveis de formalismo definido em duas fases. A primeira fase foi definida pela própria UML, uma vez que o *SPEMXt UML Profile* se utiliza de seu metamodelo. A segunda fase tem por base a descrição semântica dos estereótipos em comparação ao metamodelo do Apêndice , uma vez que eles se constituem em uma mais simples representação do metamodelo.

Por fim, uma vez que o metamodelo foi definido utilizando os princípios de empacotamento definidos na UML, houve a necessidade de especificar os níveis de conformidade a serem utilizados, definindo os pontos de conformidade a serem implementados por *Tool Vendors*. Nesse sentido foram definidos dois níveis de conformidade:

- ***Extended Method Content***, que define a utilização dos pacotes responsáveis pela criação apenas de elementos de definição de PDS. Esse nível se constitui em um ponto de conformidade para implementadores que necessitem apenas definir bibliotecas de conteúdo;
- ***All***, que define a utilização de todos os pacotes do metamodelo. Este nível estabelece a implementação de todo o metamodelo, focando na construção de PDSs completos e reutilizáveis.

5 Testes e Verificação

Neste capítulo apresentamos as formas de verificação desta abordagem. As análises foram feitas com base em testes analíticos, comparando este trabalho com outras abordagens existentes. A verificação aqui apresentada aplica tudo o que foi discutido nos capítulos anteriores, demonstrando na prática exemplos de uso da abordagem proposta. Contudo, implementamos uma ferramenta de suporte para demonstração da nossa solução.

O restante deste capítulo se apresenta da seguinte forma: na Seção 5.1 é apresentado o protótipo desenvolvido para a aplicação da proposta, tecnologias utilizadas, arquitetura, funcionalidades e implementação; na Seção 5.2 apresentam-se os cenários de testes, utilizando comparações de forma analítica; e, por fim, a Seção 5.3 apresenta as nossas conclusões sobre a abordagem após a execução dos testes.

5.1 Protótipo SwAT: *Software Artifact Specification Tool*

Com o objetivo de automatizar e colocar em prática a abordagem proposta, foi desenvolvida uma ferramenta de autoria de artefatos de software chamada Software Artifact Specification Tool (SwAT) . Juntamente com sua utilização foi feita uma avaliação deste trabalho.

5.1.1 Tecnologias Utilizadas

Durante o desenvolvimento do protótipo foram utilizadas diferentes tecnologias e ferramentas. Nesta Seção serão apresentadas as principais tecnologias utilizadas.

Eclipse Modeling Framework

O Eclipse Modeling Framework (EMF) (Steinberg et al., 2008) trata-se de um arcabouço para desenvolvimento de modelos baseados no MOF (OMG, 2008a), com suporte a geração de código através de linguagem Java. Além disso, o EMF provê persistência dos dados utilizando XMI, linguagem padrão para definição de troca de dados da UML.

Quanto a geração do código, o EMF cria a base do modelo, constituída de *Java Interfaces* e sua implementação, utilizando Java Classes. Já a geração da Interface Gráfica é feita baseada no padrão *Adapter* (Gamma et al., 1993), possibilitando visualização e edição do modelo através

de um editor básico.

Os modelos definidos pelo EMF são baseados no ECore. O ECore pode ser considerado uma implementação simples do MOF, sendo utilizado para definir a sintaxe e a semântica dos modelos manipulados pelo EMF. Isto implica na capacidade de construir elementos como classes, interfaces e associações (Moore et al., 2004).

Basicamente, os modelos criados a partir do EMF são formados por `EClasses` (elementos equivalentes às classes definidas no MOF) as quais podem ter `EAttributes` (elementos equivalentes as propriedades definidas no MOF) e, por fim, é possível estabelecer relacionamento entre as `EClasses` (esses relacionamentos são equivalentes à associações definidas no MOF).

IBM Rational Modeler (RSM)

O *IBM Rational Software Modeler* (RSM) é uma ferramenta CASE desenvolvida pela *IBM Rational Software*. Foi construída para facilitar modelagem de sistemas Orientados a Objetos utilizando UML. O foco dessa ferramenta é sua capacidade de permitir a modelagem visual dos conceitos definidos pela *Model Driven Development* (MDD) (Atkinson & Kühne, 2003).

Desta forma, essa ferramenta possibilita: (i) modelagem específica de domínio utilizando UML; (ii) criação de diagramas simples, com base em uma modelagem visual e assistente de criação; (iii) suporte a customização e transformações de modelos; (iv) criação de *UML Profiles*; (v) gera documentos e relatórios a partir dos modelos UML; e (vi) suporte a desenvolvimento descentralizado, facilitando a decomposição, comparação e *merge* do modelo.

Além disso, essa ferramenta possui suporte para construção de modelos do tipo *ecore*, sendo compatível com o EMF. Devido a esse fato, o desenvolvimento do modelo feito sobre o RSM pode ser importado pelo EMF, permitindo a geração de código e implementação do modelo. Tal característica facilita os testes sobre a modelagem, uma vez que pode ser feita automaticamente.

5.1.2 Visão Geral

Embora existam esforços referentes a definição de PDSs, pouco se fala em Autoria de Artefatos de Software. Geralmente, durante um Projeto e de acordo com o PDS utilizado, diferentes ferramentas ou técnicas de captura de conhecimento serão realizadas, com objetivo de se conseguir os ASs necessários para obter um Produto de Software. Entretanto, toda a questão sobre a definição do Artefato é praticamente ignorada, dado o fato que tais ferramentas ou técnicas possuem maior foco em preenchimento da informação.

Além disso, a existência de diversos tipos de ASs diferentes, causa um aumento da dificuldade de seu preenchimento (responsável) e entendimento (leitor), ainda mais quando tais artefatos são feitos de forma distribuída. Sendo assim, a utilização de uma ferramenta que automatize parcial ou completa a definição e uso dos ASs poderá trazer significantes melhorias.

Diante desse contexto, o protótipo de ferramenta SwAT foi desenvolvido com o objetivo de permitir a definição e uso de ASs, dentro do domínio de PDSs.

5.1.3 Escopo e limites da ferramenta protótipo

Sob o ponto de vista do Engenheiro de Conhecimento, é necessário que exista uma interação que permita a definição dos ASs, para conduzi-lo a um correta configuração do *Method Content*. Para o Engenheiro do Processo, deve ser possível utilizar a biblioteca construída a partir do Engenheiro de Conhecimento, para montar o *Process Structure*.

Tendo isso em mente, foi elaborada uma ferramenta que auxilia a Autoria de Artefatos de Software. De um modo geral, a ferramenta permite:

- permite a Autoria de Artefatos conforme definido no Capítulo 4, utilizando os princípios de separação de conteúdo (*Method Content*) e estrutura (*Process Structure*);
- verificar problemas que surgem durante a autoria, ou seja, problemas de integridade, de campos entre outros;
- importar, carregar e salvar os modelos construídos, permitindo a troca de dados através de outras ferramentas que venham a implementar esta abordagem utilizando os mesmos princípios de extensão da UML e do SPEM v2.

A execução (*enactment*) de PDSs não faz parte do escopo dessa abordagem. Dessa forma, não é necessária sua implementação nessa ferramenta. Entretanto, é possível preencher os ASs e visualizá-los com base nas informações em que se constituem.

5.1.4 Arquitetura

A arquitetura da SwAT é dividida em camadas funcionais de acordo com a Figura 5.1.

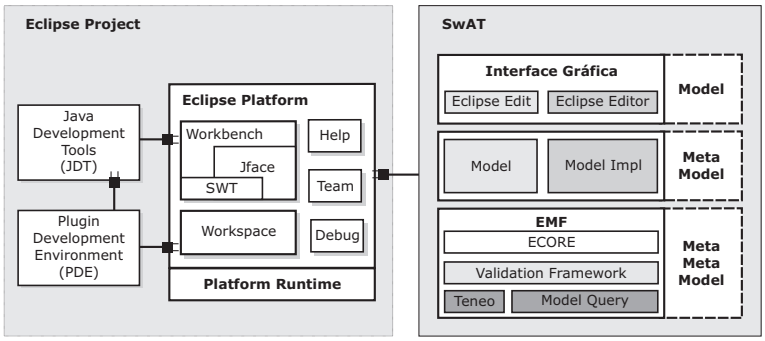

Figura 5.1: Arquitetura da Ferramenta SwAT

Infra-estrutura: *Eclipse Project*

Nessa camada estão as funcionalidades básicas para a criação de um ***plug-in*** baseado no Projeto Eclipse[1]. Eclipse é uma comunidade para produtos de código-livre, dos quais seus projetos possuem foco em construção de uma plataforma extensível e aberta. Dessa forma, o protótipo utiliza-se de *plug-ins* desenvolvidos para o Eclipse, reutilizando o seu editor e toda a infra-estrutura de seu *WorkBench*, assim como o **JFaces** e o **SWT** para construção de aplicações *Rich Client Platform* (RCP) .

Infraestrutura: EMF

Assim como o Eclipse, o *Eclipse Modeling Framework* também provê funcionalidades de infraestrutura reutilizáveis e por se constituir basicamente do **ECore**, fornece diversas funcionalidades de modelagem, tais como: persistência em XMI ou Objeto Relacional, validação de integridade de modelo e gerência dos dados do modelo com utilização de transação.

Modelo

A camada de modelo possui toda a implementação do metamodelo proposto no Capítulo 4. Toda a implementação foi feita com a utilização de linguagem **Java**, dessa forma, nesta camada também estão os contratos e restrições, feitos a partir de *Java Interfaces*.

Interface Gráfica

A interação entre a ferramenta e o seu usuário é realizada através desta camada. Toda a interface gráfica utiliza SWT e baseia-se no conceito de *Views*, *Perspectives* e *Wizards*, assim como os encontrados na plataforma Eclipse. Isto ocorre pelo fato da ferramenta SwAT funcionar como um *plugin* para o Eclipse, estendendo suas funcionalidades.

5.1.5 Funcionalidades

Nesta sessão estão dispostas as funcionalidades básicas da ferramenta de autoria SwAT. Para especificá-las, foi tomado como partida à visão do modelador. As funcionalidades aqui apresentadas foram modeladas com a utilização de UML em dois diagramas bastante comuns: Diagrama de Casos de Uso e Diagrama de Atividades, somente para as principais funcionalidades. Nesse sentido, utilizamos a ferramenta de modelagem *Rational Software Modeler*.

As funcionalidades principais da SwAT estão definidas nos pacotes:

- ***Artifact Authoring*** - as funcionalidades existentes nesse pacote possuem relação com a autoria de ASs, permitindo a criação dos construtores definidos na extensão do SPEM v2.

[1] http://www.eclipse.org/

- ***Process Authoring*** - as funcionalidades existentes nesse pacote possuem relação com a autoria de PDSs, permitindo a criação dos construtores de processos, já existentes no SPEM v2 e redefinidos na extensão proposta neste trabalho.

- ***Authoring Configuration*** - as funcionalidades existentes nesse pacote configuram o projeto de autoria, permitindo a criação de pacotes de bibliotecas, separando o conteúdo do processo em si.

Durante o projeto da ferramenta SwAT, mais especificamente durante sua fase de análise e elaboração, foi tomada a decisão de modelar cada funcionalidade como um caso de uso, tomando os devidos cuidados em relação as dependências e relacionamentos existentes.

Para facilitar a visualização, foram modeladas quatro perspectivas, agrupando os casos de uso conforme similaridade e que serão apresentadas com mais detalhes a seguir.

Perspectiva de Configuração

Nessa perspectiva estão os casos de uso responsáveis pelas funcionalidades referentes a configuração básica do projeto de autoria. Além disso, nessa mesma perspectiva são encontrados dois atores referentes aos papéis *Administrator* e *Authoring Project Configurer*, que são responsáveis pelos casos de uso desta perspectiva, Figura 5.2.

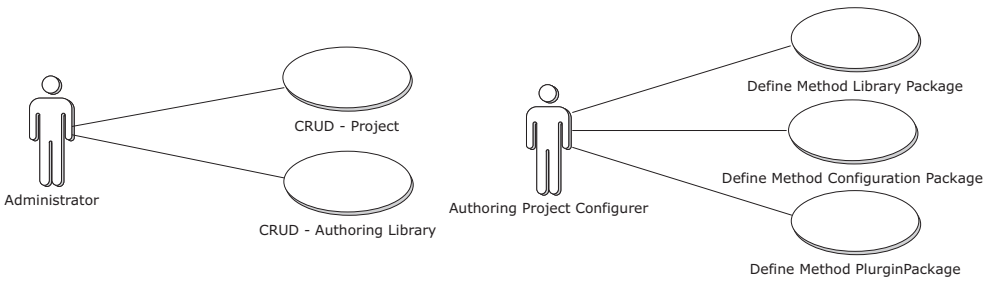

Figura 5.2: Diagrama de Casos de Uso: Perspectiva de Configuração

A seguir estão descritos os casos de uso o cujo papel *Administrator* é responsável:

`CRUD - Project` - este caso de uso é na verdade um caso de uso composto, pois agrupa as funcionalidades de criação, remoção, atualização e recuperação do Projeto de Autoria. Um Projeto de Autoria nada mais é do que um projeto SwAT que possibilita a autoria de PDSs, assim como também permite a autoria de ASs. Para uma melhor visualização desse caso de uso, os fluxos básicos para os cenários de criação e remoção se encontram nos diagramas de atividades das Figuras 5.3 e 5.4, respectivamente.

`CRUD - Authoring Library` - este caso de uso também é um caso de uso composto, pois agrupa as funcionalidades de criação, remoção, atualização e recuperação da Biblioteca de Autoria. Essa biblioteca será capaz de armazenar os dados necessários para se

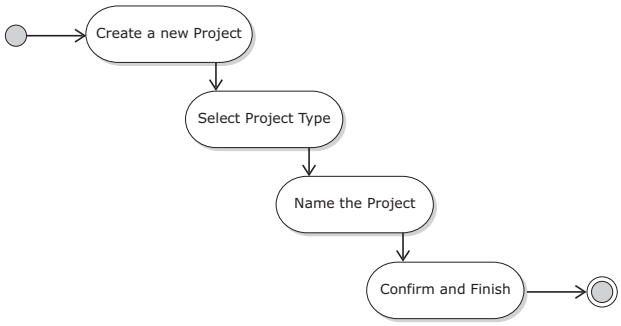

Figura 5.3: Diagrama de Atividades: Fluxo básico de criação de Projeto

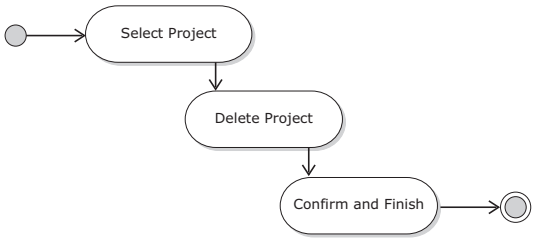

Figura 5.4: Diagrama de Atividades: Fluxo básico de remoção de Projeto

obter um PDS ou uma família deles. Para a ferramenta SwAT esta biblioteca será um arquivo do tipo XMI e que possibilita a autoria utilizando o nível de formalismo para linguagem concreta da UML. O fluxo básico para o cenário de criação deste caso de uso se encontra no diagrama de atividade da Figura 5.5.

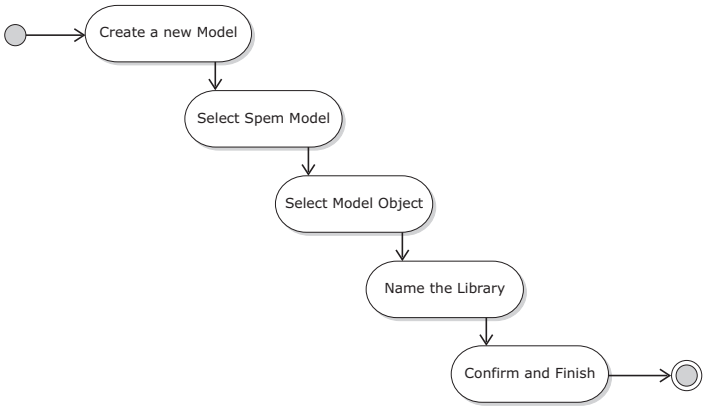

Figura 5.5: Diagrama de Atividades: Fluxo básico de criação de Biblioteca

Como pôde ser percebido através dos casos de uso anteriores, as funcionalidades de persistência de dados foram agrupadas. Para maior facilidade e reuso de modelo, os casos de uso foram modelados com a utilização de *UML Building Blocks*. Essa técnica de modelagem permite o aproveitamento de blocos de construção que podem ser utilizados em diversos pontos do mesmo projeto. O *Building Block* modelado para prover tal facilidade chama-se CRUD (sigla para as funcionalidades *Create*, *Retrieve*, *Update* e *Delete*) e pode ser visto na Figura 5.6.

Entretanto, é importante entender que embora as quatro funcionalidades de CRUD estejam agrupadas através de *UML Include Relationship*, marcadas através do estereótipo *include*, não existe a obrigatoriedade de implementação de todas estas funcionalidades. Embora o *Building Block* possa ser utilizado por completo, as vezes não existe tal necessidade, permitindo apenas um uso parcial. Mesmo assim, todas às vezes que utilizaremos o CRUD, este será por completo.

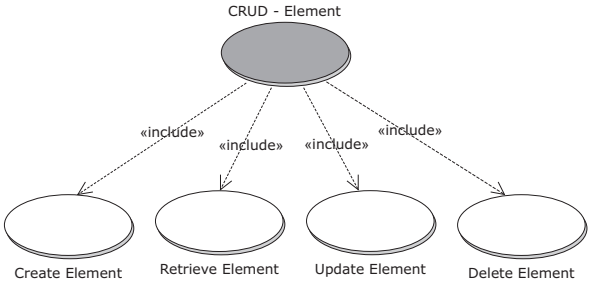

Figura 5.6: Casos de Uso relacionados às funcionalidades básicas

Perspectiva *Method Content*

Nesta perspectiva estão os casos de uso responsáveis pelas funcionalidades referentes à criação de uma biblioteca de conteúdo. O responsáveis pelos casos de uso desta perspectiva é o papel Engenheiro de Conhecimento, como visto na Figura 5.7.

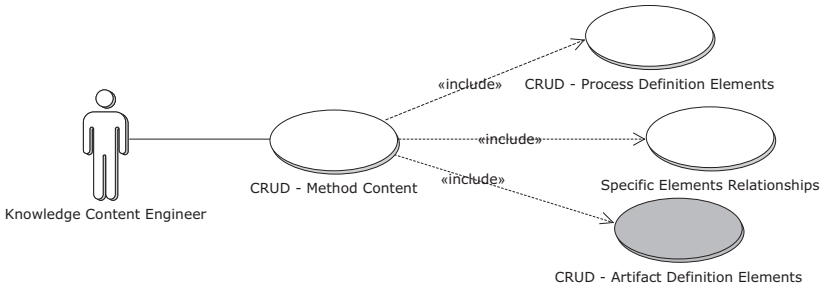

Figura 5.7: Diagrama de Casos de Uso: Perspectiva de *Method Content*

Destaque para o caso de uso `CRUD - Artifact Definition Elements`, que será detalhado mais adiante e apresenta o fluxo utilizado para construir a especificação dos artefatos.

Os casos de uso cujo papel de Engenheiro de Conhecimento é responsável são:

`CRUD - Method Content` - este caso de uso representa as funcionalidades de criação, remoção, atualização e recuperação do *Method Content*. Um *Method Content*, como visto anteriormente na Seção 2.4, nada mais é do que o conteúdo a ser utilizado para a estruturação de PDSs. Para a conclusão desse caso de uso é necessário também a execução de outros três casos de uso. Estes casos de uso estão descritos a seguir.

`CRUD - Process Definition Elements` - nesse caso de uso estão as funcionalidades de criação, remoção, atualização e recuperação dos elementos pertencentes ao *Method*

Content. Existem inúmeros elementos que podem ser adicionados, assim como papéis, atividades, tarefas, guias, ferramentas, entre outras. O fluxo básico para o cenário de criação desse caso de uso se encontra no diagrama de atividade da Figura 5.8, que apresenta um fluxo paralelo de possíveis elementos a serem adicionados.

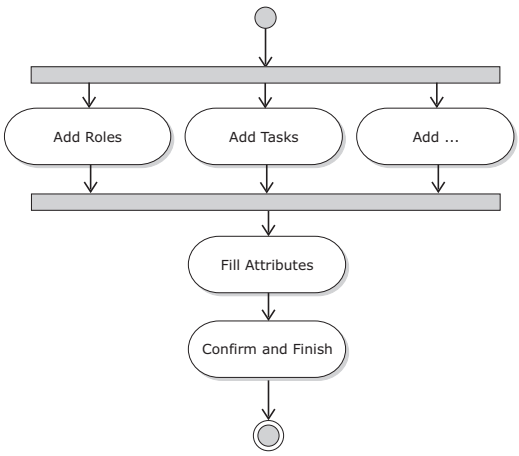

Figura 5.8: Diagrama de Atividades: Adição de elementos do Processo ao *Method Content*

`CRUD - Artifact Definition Elements` assim como o caso de uso anterior, neste estão funcionalidades para elementos pertencentes ao *Method Content*, exceto que, apenas elementos relacionados a extensão proposta. Desta forma, apenas os elementos artefato, contêiner e tipos de informações podem ser adicionados, removidos, alterados ou recuperados. Este caso de uso põe em prática um dos objetivos deste trabalho, pois permite a construção de artefatos de forma que sua organização interna esteja bem estruturada. Por ser bastante complexo, apresentaremos quatro diagramas de atividades, explicando-os em seguida.

A Figura 5.9 apresenta o diagrama de atividades com o fluxo principal. Como pode ser visto as atividades necessárias para que seja possível obter uma boa definição do artefato são: (i) criar os tipos das informações necessárias, (ii) agrupá-las em contêineres, quando necessário e (iii) finalmente a criação dos artefatos.

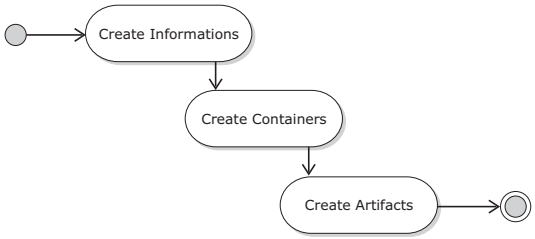

Figura 5.9: Fluxo principal para definição de Artefatos

Já na Figura 5.10 está o diagrama com o fluxo esperado para a atividade (i), que é utilizada para a criação dos tipos de informações. Tal fluxo é bastante simples, bastando determinar

o tipo da informação e, caso necessário, inserir um pequeno conteúdo descritivo para uma melhor identificação posterior. Essa descrição é totalmente opcional, porém com o seu uso é possível definir vários níveis de descrição (i.e. descrição principal, breve descrição, nome de apresentação, propósito) assim como vários níveis de seções para criar notas ou comentários em categorias.

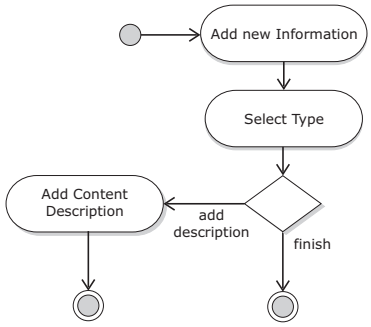

Figura 5.10: Fluxo da criação de Tipos de Informação

A criação de contêineres fica por conta do diagrama da Figura 5.11. Esse diagrama contém o fluxo esperado para a atividade (ii), que é utilizada para a criação dos contêineres. Ao criar um contêiner é necessário preencher todos os atributos obrigatórios e assim como no diagrama anterior, caso necessário, pode-se inserir um pequeno conteúdo descritivo.

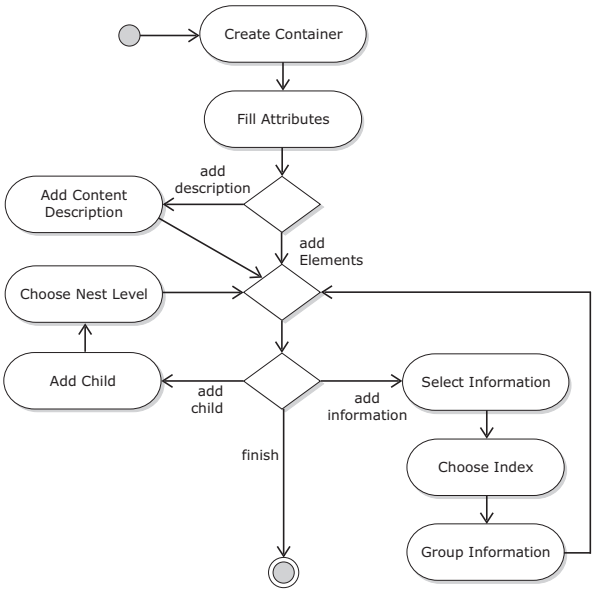

Figura 5.11: Fluxo da criação de Contêineres

Esse Diagrama de Atividades é um pouco mais complexo, pois apresenta duas iterações diferentes. Uma delas é a iteração para o agrupamento de tipos de informação que é feita em três atividades: seleciona-se um tipo de informação desejado; define-se o índice, como se trata de um relacionamento ordenado, caso não seja definido, ficará na ordem em que o tipo de informação foi adicionado; e, por fim, agrupa-se o tipo de informação confirmando

a escolha. A outra iteração é responsável pelo aninhamento de contêineres, permitindo relacionamentos do tipo *parent/child*. Além disso, como se trata de um relacionamento ordenado, pode-se alterar a ordem de construção determinando o "nível de aninhamento".

Por último, o fluxo para a criação da definição de ASs está representado através do diagrama da Figura 5.12. Como pode ser visto, para criar uma boa definição de Artefatos de Software deve-se primeiro definir **QUAIS** contêineres **OU** tipos de informações serão utilizados. Isso ocorre por que o fluxo permite a adição de ambos paralelamente. Na verdade, o que ocorre é que um AS pode possuir ambos, entretanto, é requisito, visto na **Regra E1** (Seção 4.7.1), que exista ao menos um deles. Ou seja, não deve ser possível que exista um AS vazio. Nesse fluxo pode ser visto também que durante a adição dos contêineres deve-se selecionar o tipo de relacionamento desejado e qual o índice.

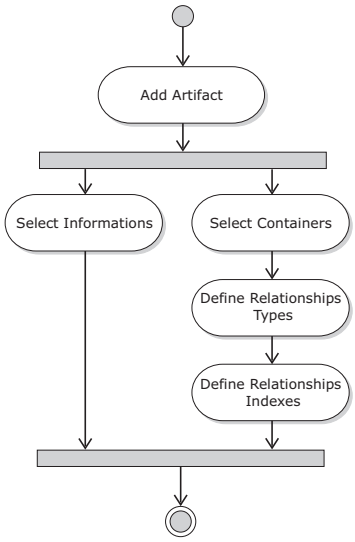

Figura 5.12: Fluxo da criação de Artefatos

`Specify Elements Relationships` - este caso de uso permite relacionar todos os elementos do *Method Content* através de relacionamentos específicos. Assim, é possível relacionar atividades com papéis, artefatos com tipos de informações, entre outros.

Perspectiva *Process Structure*

Nessa perspectiva estão os casos de uso responsáveis pelas funcionalidades referentes a criação da estrutura de PDS a partir de uma biblioteca existente. Dentro da nossa abordagem o único papel passível de construção de processos é o Engenheiro de Processo, que é o ator dos casos de uso dessa perspectiva. O diagrama de casos de uso pode ser visto na Figura 5.13.

Como pode ser visto na Figura 5.13, os casos de uso principais são `CRUD - Process Structure` e `Create Process Family and Adaptabilities`, descritos a seguir:

`CRUD - Process Structure` - este é um caso de uso composto, pois agrupa as funcionalidades de criação, remoção, atualização e recuperação da estrutura de PDSs. Na Figura

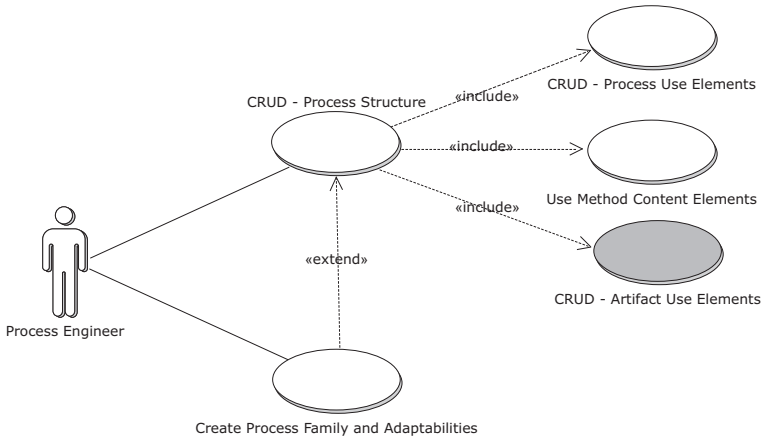

Figura 5.13: Diagrama de Casos de Uso: Perspectiva de *Process Structure*

5.13 são apresentados outros três casos de uso incluídos neste próprio. Estes casos de uso são: `CRUD - Process Use Elements`, `CRUD - Artifact Use Elements` e `Use Method Content Elements`, vistos a seguir.

`CRUD - Process Use Elements` - neste caso de uso estão as funcionalidades de criação, remoção, atualização e recuperação dos elementos pertencentes ao *Process Structure*. Existem inúmeros elementos que podem ser criados para formar a estrutura de um PDS, assim como papéis, atividades, tarefas, guias, ferramentas, entre outras.

`CRUD - Artifact Use Elements` - assim como o caso de uso anterior, neste também estão funcionalidades para elementos pertencentes ao *Process Structure*, exceto que, apenas elementos relacionados a extensão proposta. Dessa forma, apenas os elementos artefato, contêiner e tipo de informação podem ser criados, removidos, alterados ou recuperados. Este caso de uso põe em prática um dos objetivos deste trabalho, pois permite a criação do uso de ASs de forma estruturada.

`Use Method Content Elements` - por fim é possível alinhar a estrutura do processo conforme os elementos existentes no *Method Content*. Os elementos de uso, criados para a estrutura do processo, devem utilizar dos elementos de conteúdo. Este caso de uso põe em prática outro objetivo deste trabalho, pois permite a utilização da definição dos ASs.

`Create Process Family and Adaptabilities` - este caso de uso adiciona uma funcionalidade que está fora do escopo deste trabalho, porém, de bastante ajuda para àqueles que desejarem criar famílias de processos. Dessa forma, como o metamodelo proposto no Capítulo 4 estende as propriedades do SPEM v2, permitir a criação de famílias de processos e dar suporte à adaptabilidade de processos de software foi uma escolha feita para manter a compatibilidade com o nível de complacência escolhido, ou seja o *SPEM v2 Complete*. Além disso, este caso de uso estende as funcionalidades existentes no caso de uso *CRUD - Process Structure*.

Perspectiva de Artefato de Software

Nesta perspectiva estão os casos de uso responsáveis exclusivamente pelas funcionalidades sobre ASs. Como pode ser visto na Figura 5.14, nesta mesma perspectiva são encontrados os atores referentes aos papéis *Artifact Responsible* e *Artifact User*. Os casos de uso desta perspectiva representam funcionalidades além do foco do trabalho em questão e foram modelados para fins de complemento da ferramenta SwAT.

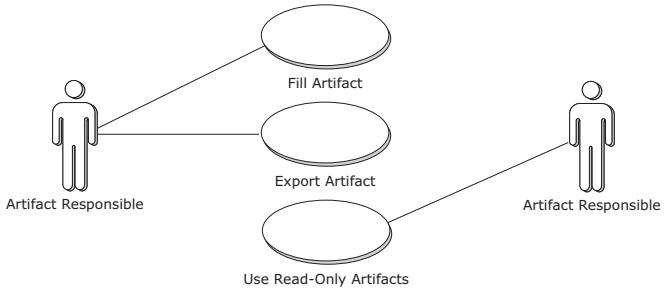

Figura 5.14: Diagrama de Casos de Uso: Perspectiva de Artefato

Os casos de uso desta perspectiva são:

`Fill Artifact` - a partir do momento em que a estrutura do Processo está pronta é possível criar instâncias dos elementos existentes. Desta forma, com este caso de uso é possível preencher os artefatos de um PDS já previamente criado. Conforme foi visto anteriormente, os artefatos deverão ser preenchidos de acordo com os tipos de informação contidos, mantendo a organização previamente definida na fase de autoria.

`Use Read-Only Artifacts` - da mesma forma que é possível preencher as instâncias dos artefatos, também pode-se utilizá-las somente para leitura. Sendo assim, o conteúdo das informações existentes nos AS previamente preenchidos podem ser visualizadas sem a necessidade exportação ou qualquer tipo de publicação.

`Export Artifact` - embora as instâncias de AS possam ser lidas dentro da própria ferramenta, isto é bastante limitado, pois a ferramenta não possui um editor avançado. Para corrigir esse problema, a funcionalidade de exportação de dados foi modelada, possibilitando a troca de dados através de outros tipos de arquivos.

5.1.6 Interface

Um ponto importante pra a autoria de ASs é a iteração entre a ferramenta e os atores usuários. Uma das formas para aumentar a compreensão e facilitar a interação entre os usuários e a ferramenta é através da utilização de uma interface amigável. Tais interfaces geralmente são ambientes gráficos do inglês *Graphical User Interfaces* (GUI) .

Dado o fato da ferramenta SwAT estender as funcionalidades do ambiente *Eclipse* através de *plug-ins*, são utilizados os conceitos de interfaces gráfica tais como: *views*, *menus* e *perspective*. A interface definida e integrada ao modulo de interface da arquitetura da SwAT.

Um visão geral da interface gráfica da SwAT é apresentada na Figura 5.15. Como pode ser visto a GUI está dividida entre várias janelas diferentes:

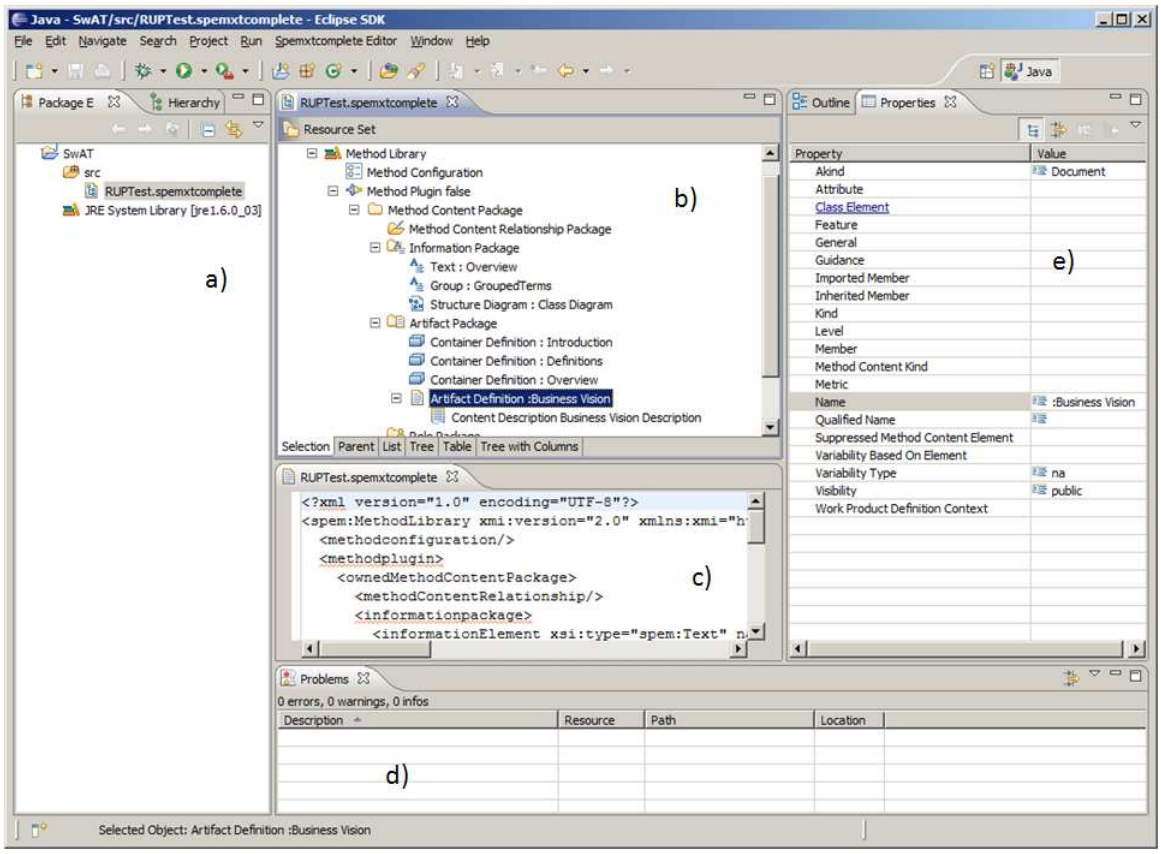

Figura 5.15: SwAT: Visão geral sobre a tela inicial da ferramenta

- em **a)** está a janela *Package Hierarchi View* que provê uma iteração com os projetos existentes e seus conteúdos internos, apresentados de forma hierárquica.

- em **b)** e **c)** estão as janelas que disponibilizam as funcionalidades da biblioteca **Spemxt**. Em b) a biblioteca é aberta através de uma *View* amigável, permitindo o manuseio da biblioteca a partir de seis modos diferentes, inclusive em árvore, constituindo a linguagem abstrata, entretanto, ainda sem uma notação em diagramas. Já em c) é apresentada em formato XMI o que consiste na linguagem concreta.

- em **d)** é mostrada a janela *Problems View* que permite ver os os problemas e erros encontrados na biblioteca durante o processo de autoria. Essa janela também disponibiliza a localização dos problemas encontrados durante a validação do modelo.

- por fim, em **e)** está a *Properties View*. Essa janela permite a iteração com os atributos e relacionamentos existentes entre os elementos do modelo. Em outras palavras, estão os atributos tanto da biblioteca quanto dos elementos que a constituem.

O objetivo da Figura 5.15 é apresentar um visão geral da ferramenta mostrando suas telas de maneira geral. Durante o restante desta Seção essas telas serão discutidas detalhadamente.

A janela *Package Hierarchi View*, vista em **a)** a partir da Figura 5.15, é reponsável pela criação e manutenção dos projetos atrelados a autoria de ASs. O principal objetivo dessa janela é permitir que atores do papel *Authoring Project Configurer* façam seus trabalhos, criando o projeto de autoria e a biblioteca *Spemxt* conforme os casos de uso apresentados anteriormente. A criação do projeto é padronizada pelo próprio Eclipse, entretanto, a criação da biblioteca seque os passos a seguir:

- **inicialização:** para criar uma nova biblioteca deve-se selecionar o projeto ao qual ela deva ser adicionada e utilizar o *pop-up* de criação através do botão direito do mouse. Nesse momento deve-se utilizar o caminho *New/Other...* no *pop-up menu*, Figura 5.16, fazendo com que surja uma tela de fácil entendimento para a criação da biblioteca, bastando agora apenas configurar algumas opções.

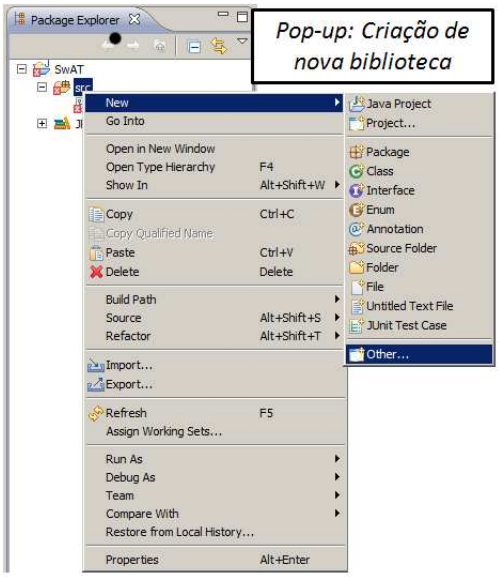

Figura 5.16: SwAT: *Pop-up menu* para criação de nova biblioteca

- **passo 0:** neste passo basta selecionar ***SpemxtComplete*** ou ***SpemxtMethodContent*** dentre as opções existentes, para criar uma nova biblioteca juntamente com o seu *wizard*, Figura 5.17. Vale lembrar que conforme o metamodelo definido no Capítulo 4 existem dois níveis de conformidade (*compliance levels*), o *All*, que implementa todo o metamodelo e o *Extended Method Content*, apenas para a utilização de **definições** e criação de conteúdo.

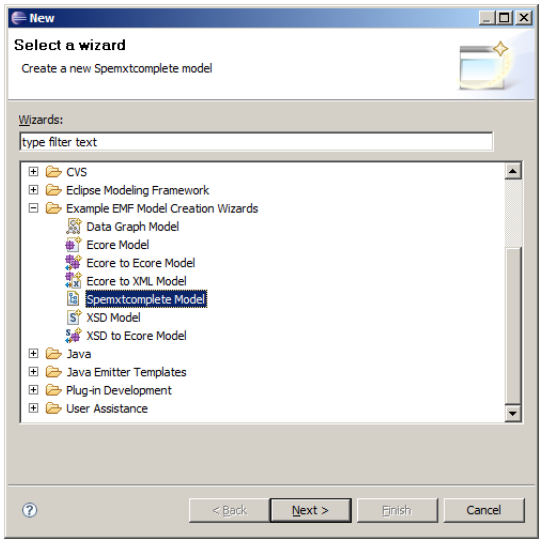

Figura 5.17: SwAT: seleção de *wizard*

- **passo 1:** após decidir qual tipo de biblioteca deve ser criada é necessário também decidir onde ela será criada. Este é o primeiro passo do *wizard* e por padrão define que a biblioteca deve sempre ser criada no projeto pré-selecionado e mais especificamente no pacote **src**. Esse passo permite configurar exatamente o caminho e projeto da biblioteca. Como exemplo de possível configuração, poderia ser criado um pacote bibliotecas, e configurar a criação nesse pacote ao invés do caminho padrão (Figura 5.18).

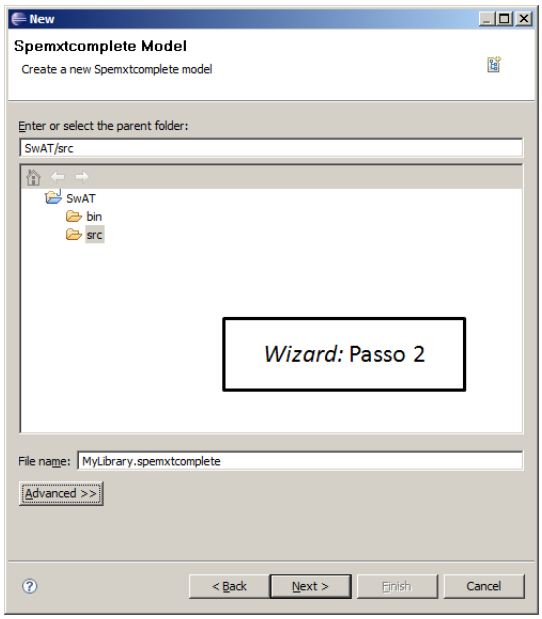

Figura 5.18: SwAT: *Wizard* passo 1

- **passo 2:** nesse último passo deve ser selecionado qual o construtor inicial. Como deve ser criada uma nova biblioteca, basta selecionar o construtor *Method Library*, Figura 5.19. Após isso, a biblioteca será criada no caminho e projeto especificados no passo anterior.

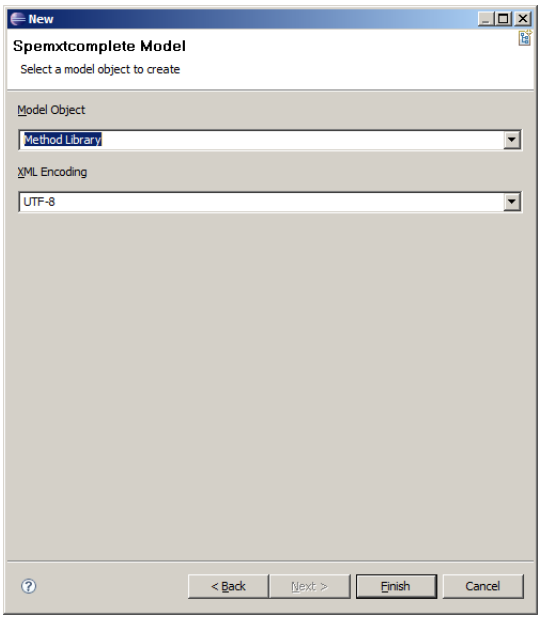

Figura 5.19: SwAT: *Wizard* passo 2

Na janela principal, vista em **b)** na Figura 5.15, estão as abas para visualização e manuseio do modelo criado e adicionado a biblioteca. A partir da Figura 5.20 podem ser vistas as abas utilizadas para permitir uma melhor iteração entre o usuário e a biblioteca.

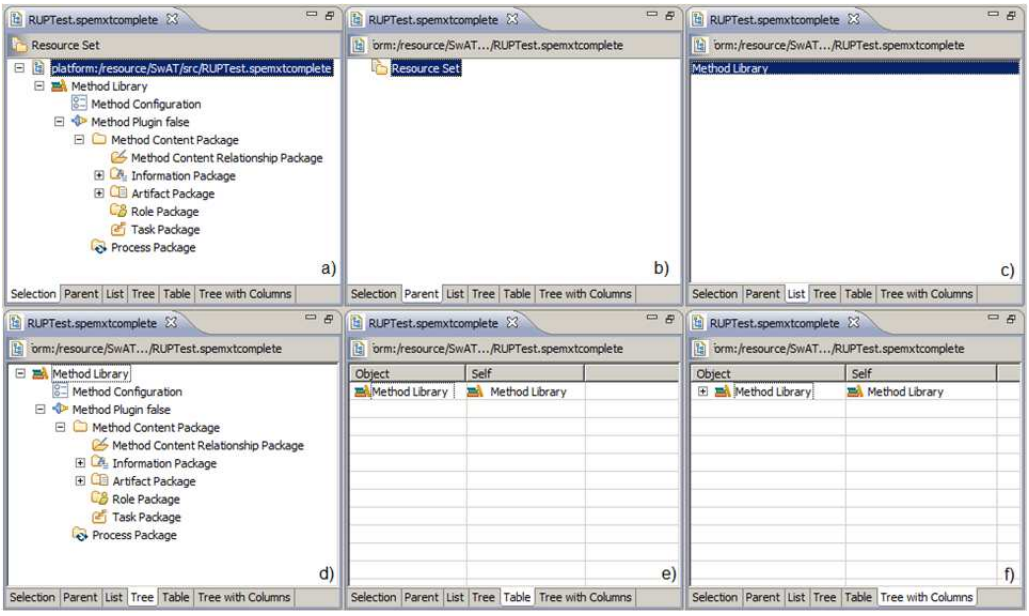

Figura 5.20: SwAT: Abas da tela principal

Tais abas são explicadas a seguir:

- em **a)** está a primeira aba, chamada *Selection*. Tomada como aba padrão ela apresenta a biblioteca de forma hierárquica dependendo de qual elemento está selecionado. A partir de um elemento e selecionado pode-se interagir com ele, criando novos elementos na

mesma hierarquia (irmão) ou interior (filho), conforme o metamodelo e suas restrições;

- em **b)** está a aba *Parent* que apresenta todos os elementos anteriores (pai) a um elemento e selecionado. Essa aba é útil quando existe a necessidade de detectar todos os elementos que estejam acima na hierarquia existente;
- em **c)** está a aba *List* que permite a visualização em lista de todos os elementos de mesmo nível hierárquico (irmãos) a um elemento e selecionado;
- em **d)** está a aba *Tree* que permite uma visualização em árvore. Desta forma, a partir de um elemento e selecionado, serão mostrados todos os seus elementos filhos;
- em **e)** está a aba *Table* que apresenta uma tabela utilizada para relacionar tipo e instância dos elementos selecionados. Desta forma pode-se estabelecer explicitamente qual a metaclasse que o elemento da biblioteca instancia, já que nem sempre é possível fazer esta verificação apenas visualizando os ícones de cada um desses elementos;
- por fim, em **f)** está a aba *Tree with columns* que apresenta uma tabela mais completa. Esta tabela relaciona todos os elementos construídos (instâncias) através de seus tipos, mas disponibilizando-os hierarquicamente.

A partir da aba *Selection* pode-se selecionar os elementos que se deseja alterar ou remover e visualizar os elementos já criados. Além disso, ainda é possível adicionar novos elementos, conforme a Figura 5.21. Para isto deve-se utilizar um *pop-up menu* que possibilita a criação de novos elementos (*New Child*) ou elementos irmãos (*New Sibling*).

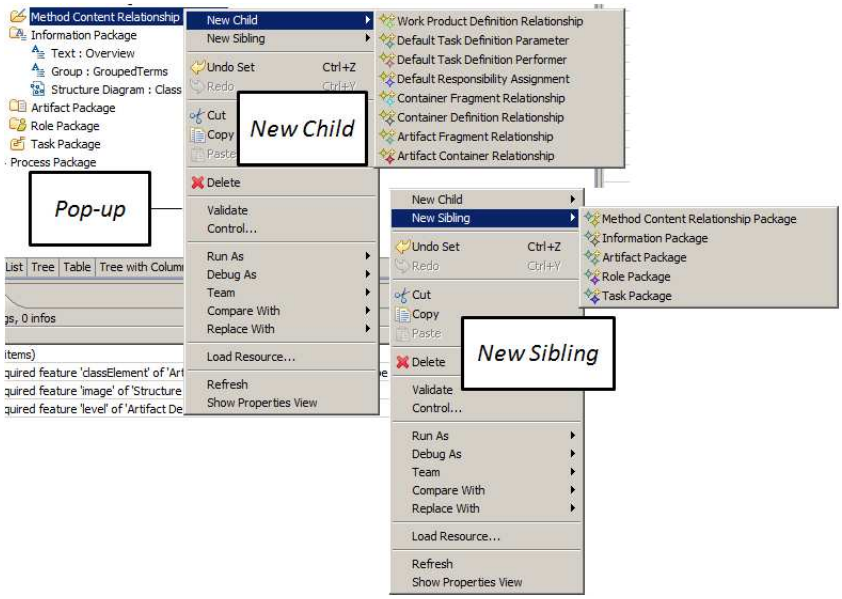

Figura 5.21: SwAT: Tela Inicial da Ferramenta

Na janela *Problems View*, vista em **d)** na Figura 5.15, está a janela responsável por relatar os problemas e erros existentes durante a criação da biblioteca. Como sabemos a criação da

biblioteca na verdade é uma instância do metamodelo apresentado no Capítulo 4. Por isso, ao criar uma biblioteca estamos na verdade criando um modelo que deve obedecer às restrições e regras existentes no metamodelo. Entretanto, mesmo com as restrições existentes nem sempre é possível deixar um modelo em um estado correto. Contudo, para descobrir a existência desses tipos de erros ou inconformidades devemos então validar o modelo.

A Figura 5.22 apresenta uma validação feita sobre um elemento da biblioteca. Conforme pode ser visto, existe uma série de conformidades, dispostas na janela de validação. Desta forma todos os erros encontrados serão apresentados, inclusive erros em elementos filhos. Isto ocorre por que a validação funciona em cascada, procurando por inconformidades existentes no interior de um elemento selecionado, portanto, seus relacionamentos fortes, ou seja do tipo composição, serão validados. Diferentemente, caso o elemento selecionado seja um elemento folha, apenas seus atributos e relacionamentos serão verificados. Como o elemento validado está em uma hierarquia bastante alta, todos os erros de conformidade existentes em seus elementos filhos serão revelados.

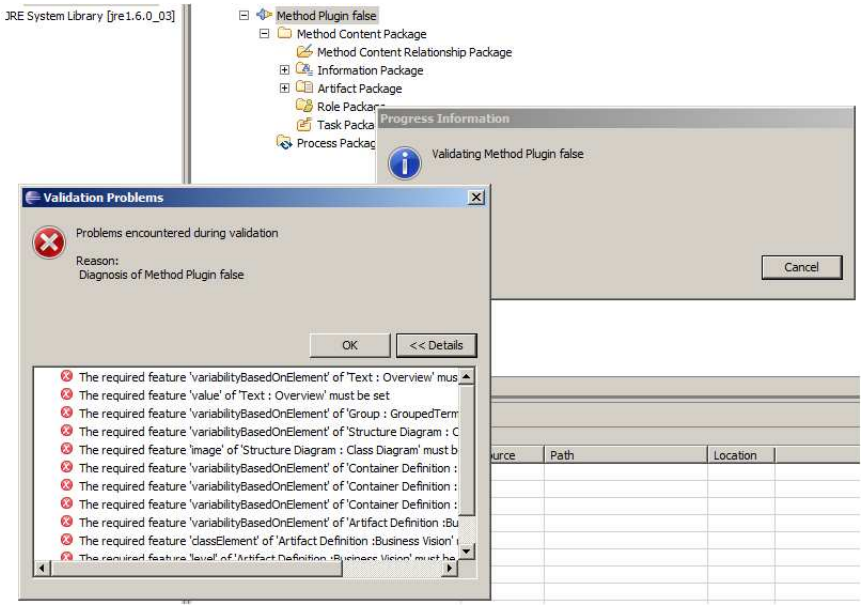

Figura 5.22: SwAT: apresentação de inconformidades

Esses problemas de validação estão dispostos através de mensagens e estas mensagens possuem um formato típico. O formato em que se apresentam não traduz a localização exata de problema, embora traga informações importantes sobre qual restrição e como supostamente resolver. Desta forma, é necessário completar a quantidade de informações para uma melhor correção desses problemas. Na Figura 5.23 está a janela *Problems View*, que complementa as informações sobre as inconformidades mostrando qual o elemento incorreto e sua localização. Além disso, todos os elementos que possuem problemas são marcados com um ícone diferente e bastante intuitivo, apontando os problemas localmente. Essas marcações são feitas em todos os problemas, marcando todo o caminho até encontrar o elemento problemático, tanto no *Package Explorer* quanto na janela principal.

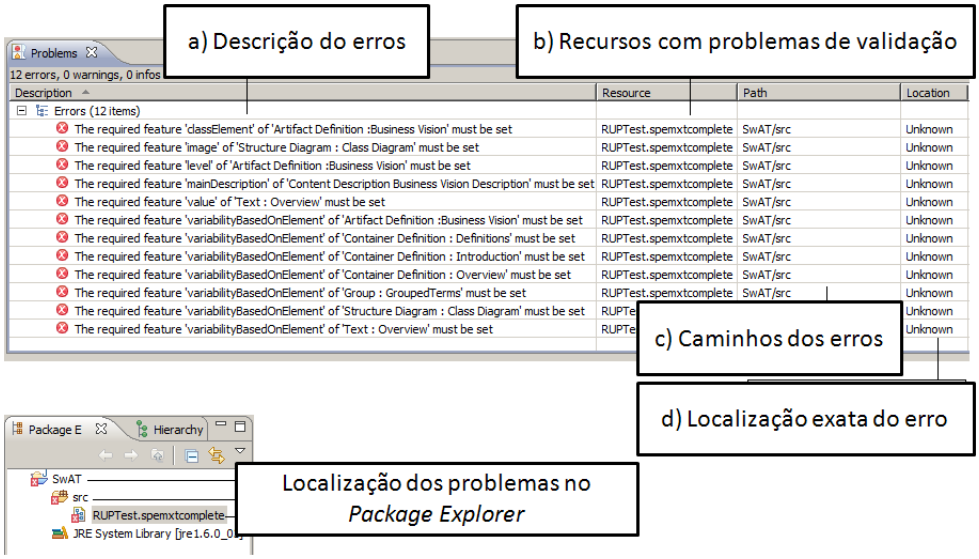

Figura 5.23: SwAT: Janela *Problems View*

A última janela a ser detalhada é a *Properties View*, Figura 5.24. Conforme supracitado, nessa janela deve estar os atributos e relacionamentos existentes em um elemento selecionado. Dessa forma é possível inserir os dados importantes como nome e descrição, assim como relacionar os elementos definindo os seus relacionamentos.

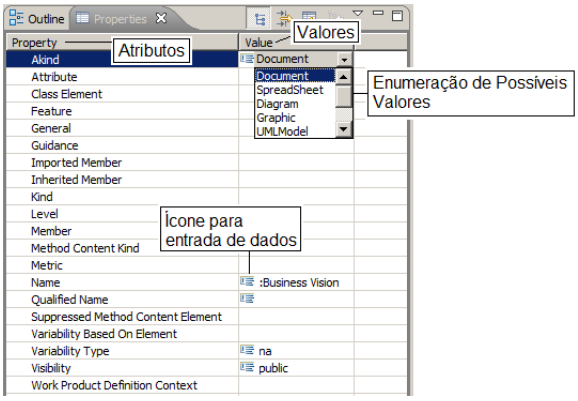

Figura 5.24: SwAT: janela *Properties View*

Como pode ser visto na Figura 5.24, os atributos existentes estão alinhados ao lado esquerdo da janela, enquanto que os valores devem ser informados ao lado direito. Além disso, pode ser visto o ícone para entrada de dados, indicando que o valor do atributo pode ser informado. Existem seis tipos de entradas: (i) entradas de texto, onde podem ser inseridos valores de caracteres; (ii) entradas numéricas, para entradas somente de números; (iii) entradas booleanas, decidindo entre verdadeiro ou falso; (iv) entradas para seleção de relacionamento `n..1`, mostrados através de uma lista de seleção simples; (v) seleção de um literal ou tipo enumerado, também se apresenta conforme uma lista de seleção; e (vi) definição de entrada para relacionamentos do tipo `1..m` ou `n..m`, Figura 5.25.

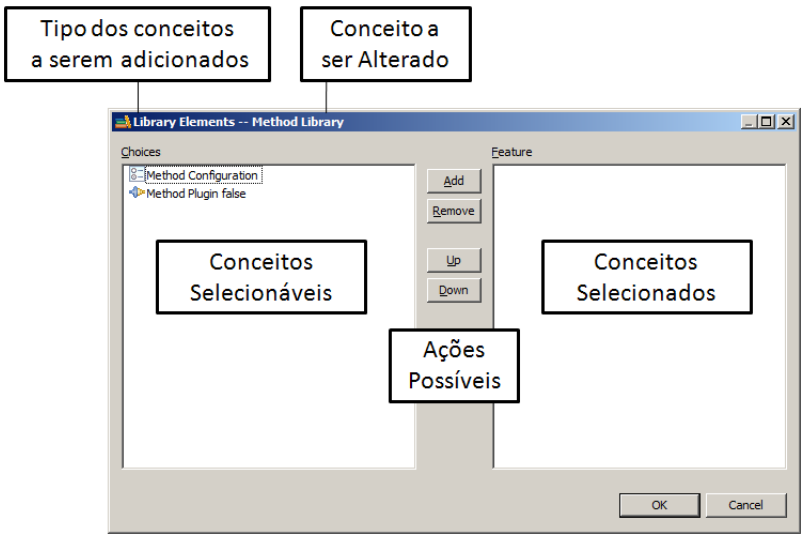

Figura 5.25: SwAT: janela de inserção de dados em relacionamentos do tipo 1/m .. n

5.2 Cenários de Teste

Nessa Seção serão apresentados os testes e exemplos feitos sobre a abordagem. Para isto foram criados dois testes diferentes, apresentando o uso da abordagem através do protótipo da ferramenta SwAT e o de diagramas UML.

Como se trata de uma extensão conservativa do SPEM v2 e da UML, essa abordagem se configura na criação de diversas camadas de abstração, previstas em (OMG, 2007b). Partindo desse pressuposto, uma idéia mais clara pode ser obtida a partir do ponto de vista da Figura 5.26, onde essas camadas de abstração são apresentadas, deixando bastante evidente que nossa abordagem está dividida em três níveis possíveis: metamodelo (M2), modelo (M1) e instância (M0).

Neste trabalho, como normalmente é feito na literatura, através de trabalhos como (Lee et al., 2002) e (Borsoi & Becerra, 2008), vistos nos Capítulo sobre Trabalhos Relacionados (mais especificamente na Seção 3.2), consideramos a camada M2 uma abstração para a criação da camada M1, a camada M2 como camada de abstração para a camada M1 e, por fim, a camada M1 definirá como será a camada M0. Neste sentido, a camada M3 é o próprio MOF. Para uma maior clareza, a Figura 5.2 ilustra essas quatro camadas.

A camada M3 é responsável por definir M2 e a si própria. Nessa camada estão o MOF e o *MOF Versioning*. O MOF é capaz de definir outros metamodelos e através de operações de *reflection* consegue se auto definir. Ainda utilizando *reflection*, o *MOF Versioning* acaba por inserir a capacidade de versionamento através de `VersionedExtent`, como pode ser visto na Figura 5.26.

Através da camada M2, definida como **Camada de Artefatos**, são definidos os elementos principais para a criação da definição e uso dos artefatos. A modelagem desta camada foi feita

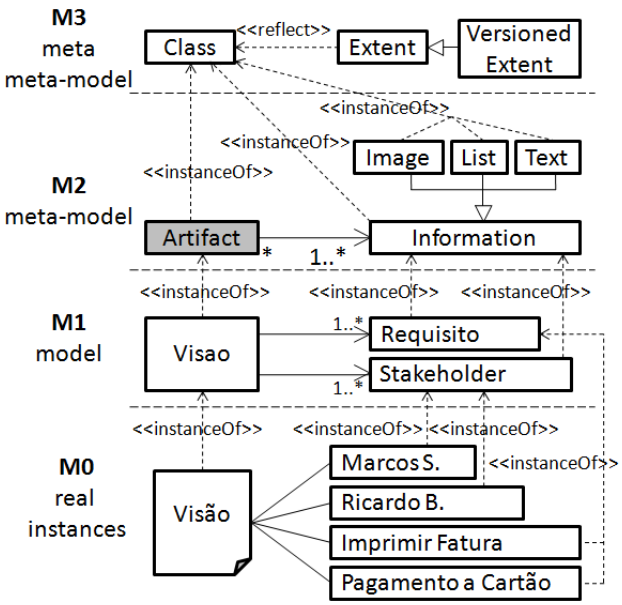

Figura 5.26: Camadas para a autoria de Artefatos.

de forma a deixar seus elementos os mais genéricos possíveis, ou seja, são elementos independentes de PDSs específicos. Isto faz com que esta camada seja abstrata em relação a processos. Como pode ser observado, a Figura 5.26 contém apenas uma pequena ilustração do metamodelo através de `Artifact` que se relaciona com uma coleção de `Information`, que pode do tipo `Text`, `Image` ou `List`. Isto foi feito para facilitar o entendimento do metamodelo, pois na realidade ele possui mais de 100 metaclasses. Utilizando o M2, poderíamos, por exemplo, criar um artefato `A` que possui título, uma imagem com rótulo da seguinte forma: *Artifact* A=(*Text*: título; *Image*: imagem; *Text*: rótulo).

Diferente da camada de artefatos, a camada M1, definida como **Camada de Processos**, possui elementos mais específicos, justamente para a modelagem dos artefatos específicos de processos de desenvolvimento de software. Nessa camada são utilizadas instâncias de elementos da camada M2. Em M1 devem estar instâncias das metaclasses definidas na camada de artefatos para a construção de classes que modelem os artefatos de um processo específico. Como resultado disso, não existe um modelo M1 definitivo, variando conforme a necessidade de criação ou adaptação do processo a ser modelado. A exemplo, na Figura 5.26 está modelado um documento de Visão, do RUP, que instancia Artefato, de M2. Nessa mesma figura pode ser visto ainda que o documento de Visão se relaciona com Requisitos e *Stakeholders*, ambos instâncias de tipos de Informação.

Por fim, na camada M0, neste trabalho definida como **Camada de Projetos**, estão as instâncias reais dos artefatos construídos com informações reais de um projeto que executa um PDS específico, definido em M1. Estas instâncias são condizentes com a camada M1, pois são instâncias do que foi modelado nessa camada. Desta forma, como pode ser visto na Figura 5.26, existe uma instância de Visão que contém dois *stakeholders*, Marcos S. e Bastos R. e dois requisitos,

Imprimir Fatura e Pagar em Cartão: Visão =(**Stakeholder**=Marcos S.; *Stakeholder*=Bastos R.; *Requirement*=Imprimir Fatura; **Requirement**=Pagamento em Cartão).

Embora o foco da nossa abordagem seja a autoria de ASs, programamos os conceitos existentes através de uma linguagem com base na UML. Da mesma forma como feito no SPEM v2, existem duas extensões claras: uma delas é definida através de um metamodelo que estende o SPEM v2 restringindo-o a um domínio mais específico; a outra é baseada na criação de novos estereótipos capazes de determinar a extensão do *UML Profile* definido também pelo SPEM v2.

Partindo deste princípio, foram feitos dois cenários de testes diferentes, monstrando: (1) a utilização abordagem através da ferramenta SwAT, (Seção 5.2.1), que é uma implementação do metamodelo proposto; e (2) a aplicação do catálogo de estereótipos definidos através de *UML Profile*, (Seção 5.2.1). Esta implementação a que permite a utilização ferramentas UML comuns.

Os Cenários foram testados a partir da utilização de artefatos do RUP. Ao invés de modelar todo o PDS, foram escolhidos diferentes artefatos das disciplinas desse processo, procurando atender aos pontos chaves da abordagem. Escolhemos artefatos conforme o nível de complexidade, utilização e necessidade. Entretanto, essas variáveis foram determinadas através de experiência própria. Nesse sentido e conforme nosso objetivo, escolhemos então modelar os artefatos: (A1) Glossário de Negócios, da disciplina de Modelagem de Negócios; (A2) Glossário e (A3) Especificação de Caso de Uso, da disciplina de requisitos; (A4) Modelo de Análise e (A5) Modelo de *Design*, da disciplina de Análise e *Design*.

Levando-se em consideração que tais artefatos já foram definidos pelo RUP, deve-se então fazer um mapeamento dos *templates* originais, geralmente em Microsoft Word™, formando um modelo conforme nossa abordagem. A Figura 5.27 mostra como isto deve ser feito. Inicialmente, em **a)** existe uma conceitualização de como é um artefato, partindo de seu *template*.

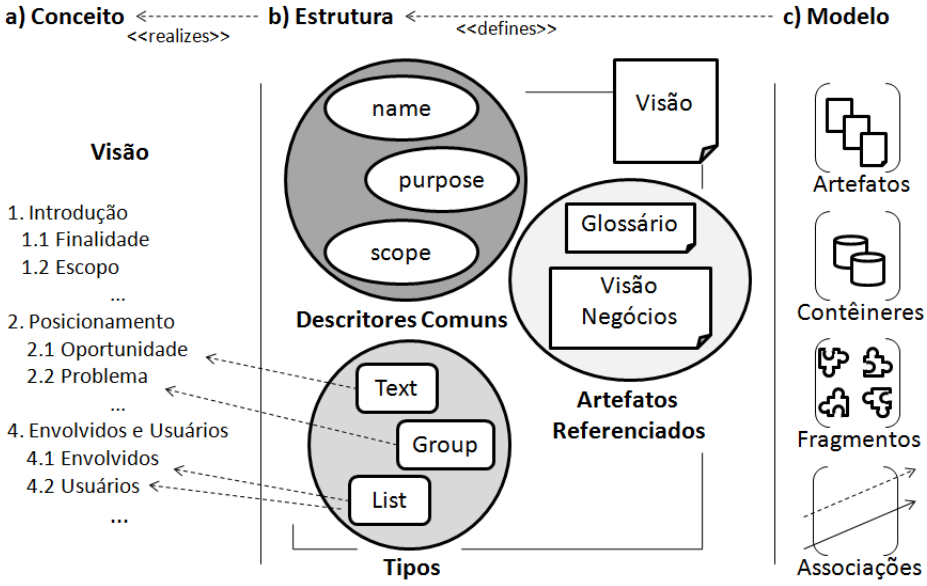

Figura 5.27: Mapeamento entre a conceitualização de um artefato e seu modelo M1.

Deve-se então selecionar quais os tipos de informação e como agrupá-las de forma a conseguir estruturar o artefato conforme o nossa proposta, mapeando os conceitos do *template*, para conceitos existentes no nível M2. Por fim, deve-se construir o modelo de nível M1 utilizando-se dos elementos existentes no nível M2, previamente mapeados.

O objetivo geral dos testes é mostrar que cada um desses artefatos pode ser definido através da camada M2, criando elementos na camada M1. Além disso, será apresentada uma amostra da camada M0, com o intuito de demonstrar que é possível se obter uma instância real de artefatos ao final do processo de autoria.

Todavia, para que isto seja possível deve-se levar em conta a ordem de construção ser feita. Tal ordem, previamente definida na Seção 4.6, será seguida, permitindo também que o Guia para Autoria de Artefatos de Software seja testado.

5.2.1 Cenário de Teste 1

Nesse primeiro cenário será utilizado o protótipo da ferramenta SwAT para instanciar a camada M2, criando assim a camada M1. Este protótipo implementa a camada M2, dando suporte a construção da camada M1. Além disso, a ferramenta de suporte também implementa o Guia para Autoria de Artefatos de Software, automatizando-o.

Criação de Novo Projeto de Autoria

Desta forma, vamos então criar um novo PDS, inciando uma autoria. Esse é o primeiro passo a ser executado, segundo o Guia. Como este passo já está implementado, basta utilizar a ferramenta para criar um novo projeto de autoria do Processo de nome *SwAT.Test*.

Os próximos passos a serem seguidos são: (i) criar uma nova biblioteca; (ii) criar um *Method Content* para a biblioteca e (iii) criar um *Process Structure*. O responsável pela publicação não está implementado nessa versão do protótipo.

Criação de Biblioteca

Seguindo então para o próximo passo, devemos criar uma nova biblioteca, Figura 5.28.

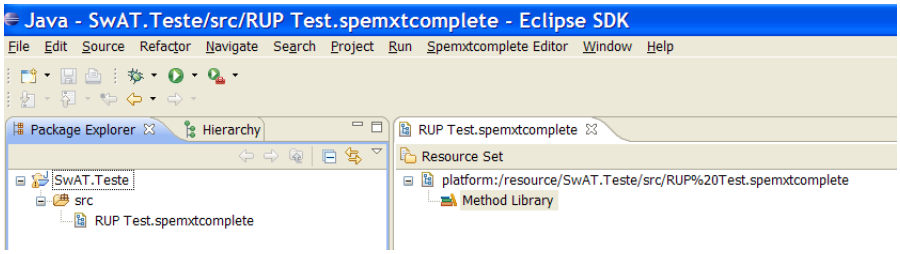

Figura 5.28: Criação da Biblioteca *RUP Test*

Este passo é composto por dois sub-passos: criar o espaço de biblioteca e nomear e descrever biblioteca. Sendo assim, criaremos um arquivo XMI que suporta o modelo de nível M1. Embora neste trabalho o arquivo XMI seja considerado a biblioteca (*SPEMxt Library*), na verdade ele é uma implementação de um dos níveis de formalismo definidos para a construção de linguagens com base na OMG (OMG, 2007b), sendo a linguagem concreta desta abordagem. Para completar, nomearemos esta biblioteca de `RUP Test` e a descreveremos como `Uma biblioteca para testes utilizando o RUP`, tudo isto pode ser visto na Figura 5.28.

Criação do *Method Content*

Após criamos a biblioteca vamos partir para a autoria de ASs iniciando com a criação do *Method Content*. Como já definido anteriormente, o *Method Content* é como se fosse uma biblioteca específica que guarda apenas os elementos que participam da definição do que pode ser utilizado em PDSs. Ou seja, definições de artefatos, atividades, papéis, entre outros. Para o nosso contexto, é importante definir a criação dos elementos necessários para a autoria de artefatos. Para isto, precisamos criar os tipos de informação e contêineres necessários para organizar detalhadamente todos os artefatos a serem modelados.

Embora seja simples, este passo requer um pouco mais de perícia para ser executado. Sendo assim, partindo do princípio descrito na Figura 5.27, primeiro faremos um mapeamento de todos os artefatos escolhidos para só então modelá-los. Ao identificarmos a estrutura, podemos obter então a modelagem em nível M1.

Na Figura 5.29, está uma amostra do resultado obtido no mapeamento. Nesta amostra estamos utilizando o artefato Especificação de Caso de Uso, que possui diversas seções e diferentes tipos de informação. Como o mapeamento foi feito a partir de *templates* definidos no software *Rational Unified Process*, optamos por manter os nomes e termos em inglês, evitando possíveis problemas na tradução.

Para uma melhor exposição, a Figura 5.29 está organizada em colunas, correpondendo as metaclasses utilizadas do modelo M2. As respectivas instâncias, agrupadas no modelo M1, estão abaixo de suas metaclasses. Como exemplo, a classe `FlowOfEvents`, afixada na segunda coluna (da esquerda para a direita) indica que ela é uma instância da metaclasse `ContainerDefinition`. É importante notar que a primeira coluna a esquerda é uma captura do artefato real, sendo possível observar seu nome e os títulos das seções que o constituem. Uma análise dos elementos modelados para especificar o documento Especificação de Caso de Uso pode ser vista a seguir:

- `Use-Case Specification`, como é a representação do artefato foi modelado como uma instância da metaclasse *ArtifactDefinition*;

- as seções desse artefato foram modeladas como instâncias de *ContainerDefinition*;

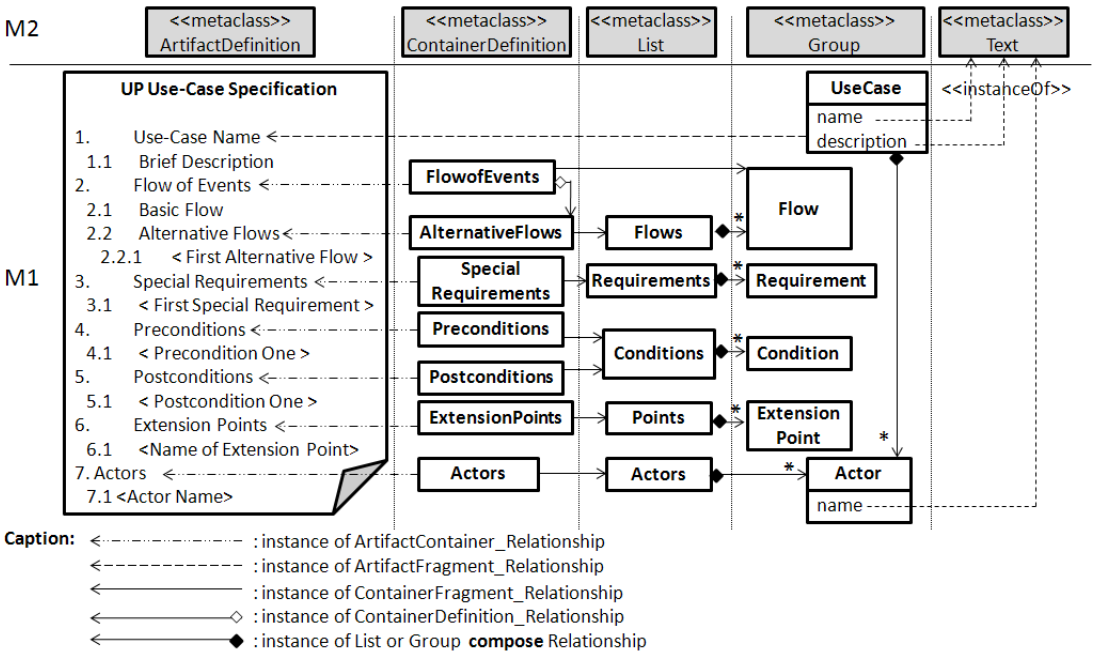

Figura 5.29: Estruturação do artefato Especificação de Caso de Uso

- como pode ser visto, a seção `Alternative Flows` é uma instância de *ContainerDefinition*, entretanto ela também é uma subseção de `Flow of Events`;

- neste exemplo nós apresentamos cinco tipos diferentes de informação. Todas elas são subclasses de *FragmentDefinition*:

  1. as classes que representam listas dos tipos estruturados de informação foram modelados através da metaclasse *List*, vistas na coluna correspondente;
  2. as classes `Flow`, `Requirement`, `Condition`, `Extension Point`, `Use-Case` e `Actor` são instâncias da metaclasse *Group*. Uma vez que essas classes agrupam diversos tipos de informação, estruturando-as, elas podem possuir conteúdos internos compostos. Na Figura 5.29, um exemplo disto pode em `Use-Case` que é composto de um conjunto de `Actor`. Na verdade estes tipos de informação poderiam ser modelados através de tipos de informação específicas de processo, ou seja, *SpecificInformationElements*. Uma vez que queremos testar a flexibilidade da nossa abordagem, preferimos utilizar os tipos genéricos em detrimento da definição de um tipo específico;
  3. a informação agrupada `name`, de `Actor` e `name/description`, de `Use-Case`, são instâncias do tipo *Text*;

- na Figura 5.29 também são mostrados os relacionamentos existentes entre esses elementos, demonstrados com a utilização de cinco diferentes tipos de setas. É necessário também deixar claro que cada seta representa uma metaclasse diferente:

1. os relacionamentos entre o artefato e suas correspondentes seções foram feitos através de instâncias da metaclasse `ArtifactContainer_Relatioinship`, representadas por ligações em setas do tipo traço-ponto-ponto;
2. o relacionamento entre o artefato e o tipo de informação `Use-Case` é uma instância da metaclasse *ArtifactFragment_Relationship* meta-class, pois `Use-Case` é uma instância de um fragmento do tipo *Group*. Este relaciomanento é denotado a partir de ligações em seta tracejada;
3. os relacionamentos entre as seções e os tipos de informação são instâncias da metaclasse *ContainerFragment_Relationship*. Na Figura 5.29 estes tipos de relacionamentos são representados através de ligações em setas de linha sólida;
4. o relacionamento entre `Flow of Events` e `Alternative Flows` é uma instância de *ContainerDefinition_Relationship*. Este tipo de relacionamento é apresentado como uma seta com diamante claro;
5. finalmente, o relacionamento entre *Group*/*List* com outros tipos de informação são instâncias de um auto relacionamento próprio do tipo composição. Esse auto relacionamento permite que essas metaclasses quando instanciadas possuam uma composição feita de conteúdo interno, já que as instâncias dessas metaclasses funcionarão como agrupadores e organizadores de outros tipos de informação. Este relaciomanento é denotado através de ligações utilizando uma seta com um diamante escuro.

Esta amostra foi escolhida por representar alguns elementos primordiais nessa abordagem, pois houve aumento no reuso e em sua organização estrutural, uma vez que o artefato agora está representado em uma linguagem *UML-Like*. Desta forma, os mesmo elementos podem ser utilizados para compartilhar informação sobre os possíveis tipos, a estrutura e o conteúdo.

A criação desse modelo utilizando a SwAT foi feita acrescentando-se os tipos de informação, contêineres e artefatos encontrados no mapeamento, tomando o cuidado de criar as ligações entre eles. Como era de se esperar, a implementação desse passo confere ao que está definido no Guia de Autoria, entretanto, na SwAT pode ser feito de forma mais flexível. Isto se deve ao fato de seguir estes três sub-passos (criar tipos de informação, criar contêineres e criar artefatos) exatamente como no guia ser menos vantajoso do que deixá-los livres, ou seja, menos engessados. Mesmo assim, para a utilização com maior clareza, recomenda-se seguir o Guia.

Desta forma, inicialmente mostramos a modelagem dos tipos de informação, presentes na Figura 5.30.

Os tipos de informação foram divididos em vários pacotes para permitir uma maior facilidade de identificação. Este tipo de divisão pode ser feito para facilitar a visualização dos elementos, pois o modelo tende a crescer e tornar-se muito grande. Os pacotes são:

1. esse primeiro pacote da Figura 5.30 (de cima para baixo) contém os tipos de informação referentes ao artefatos Glossário de Negócios e Glossário, que são:

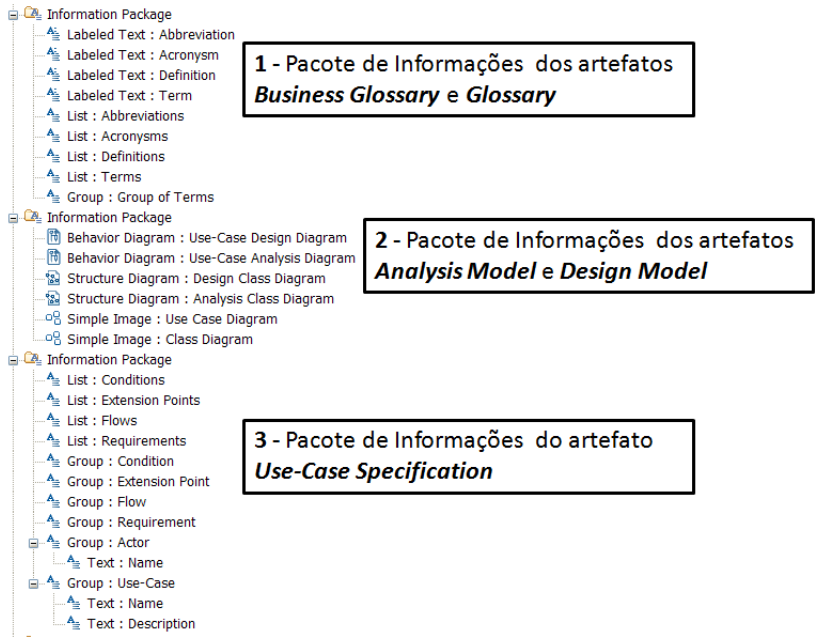

Figura 5.30: Tipos de Informação modelados a partir na ferramenta SwAT

- Abreviação - instância da metaclasse *LabeledText* que representa um texto rotulado. Um exemplo real da aplicação de uma abreviação seria: Ex.: Abreviação=(**rótulo:** Doutor, **valor:** Dr.);
- Acrônimo - instância da metaclasse *LabeledText*. Um exemplo real seria: Ex.: Acrônimo=(**rótulo:** RUP, **valor:** Rational Unified Process);
- Definição - instância da metaclasse *LabeledText*. Um exemplo real seria: Ex.: Definição=(**rótulo:** reta, **valor:** (Conceito primitivo) É um conjunto infinito de pontos alinhados de tal forma que os segmentos com extremidades em dois quaisquer desses pontos);
- Termo - instância da metaclasse *LabeledText*. Um exemplo real seria: Ex.: Termo= **rótulo:** Login, **valor:** login de usuário da aplicação;
- Abreviações, Acrônimos, Definições e Termos - instâncias da metaclasse *List* que representa uma estrutura de dados do tipo lista para um conjunto de elementos que podem ser ordenados, enumerados e rotulados. Estas listas agrupam os tipos de informação Abreviação, Acrônimo, Definição e Termo, respectivamente.Um exemplo real da aplicação de lista seria: Ex.: Acrônimos=(Acrônimo=(**rótulo:** RUP, **valor:** Rational Unified Process), Acrônimo=(**rótulo:** RS, **valor:**; Rio Grande do Sul)). Caso a lista seja ordenada, RS virá primeiro que RUP, caso seja enumerada são adicionados identificadores de numeração (1,2,3 ...), caso seja rotulada, pode-se adicionar um rótulo extra para cada elemento da lista.

2. o segundo pacote contém os tipos de informação utilizados pelos artefatos Modelo de Análise e Modelo de *Design*, que são:

- Análise e *Design* de Casos de Uso - ambos são instâncias da metaclasse *Behavior-Diagram* que representa um diagrama comportamental da UML;
- Análise e *Design* de Classes - ambos são instâncias da metaclasse *StructureDiagram* que representa um diagrama estrutural da UML;
- Diagrama de Casos de Uso e Diagrama de Classes - ambos são instâncias da metaclasse *SimpleImage* que representa uma imagem, figura ou instantâneo. No presente caso, ambas instâncias são utilizadas para representar os diagramas de análise e *design* anteriores.

3. o último pacote contém os tipos de informação utilizados pelo artefatos Especificação de Caso de Uso, que são:

   - Ator - instância da metaclasse *Group*, que representa um agrupamento de tipos de informação a serem estruturados de forma *ad hoc*, "sob medida" e exclusiva. Neste caso, estamos tratando de um Ator que executará um Caso de Uso. Como pode ser observado na Figura 5.30, o tipo de informação texto, representado através de uma instância de *Text* é agrupado para representar este tipo de informação. Embora este caso seja simples, poderíamos adicionar outros tipos de informação, tornando-o um caso mais complexo. Um exemplo real de utilização de um Ator seria: Ex.: Ator=(**nome:** Usuário);
   - Caso de Uso - instância da metaclasse *Group*. Neste caso, estamos tratando de um Caso de Uso como definido pelo RUP. Como pode ser observado na Figura 5.30, este Caso de Uso contém os tipo de informação texto nome e descrição, representados através de instâncias da metaclasse *Text*. Um exemplo real de utilização seria: Ex.: Caso de Uso=(**nome:** Login, **descrição:** caso de uso para a ação de login executada por usuários registrados ao sistema...);
   - Condição - instância da metaclasse *Group*. Neste caso, estamos tratando de uma condição a ser obedecida na execução de um Caso de Uso. Foram utilizados dois tipos de condição: Pré-condição, que sugerem uma condição a ser obedecida anteriormente ao início do Caso de Uso; e pós-condição, a ser obedecida após a execução do Caso de Uso;
   - Fluxo - instância da metaclasse *Group*. Neste caso, estamos tratando do fluxo a ser seguido durante a execução de um Caso de Uso. Nesse sentido foram definidos dois tipos de fluxos: o fluxo básico a ser seguido e fluxo alternativo, que pode determinar algum fluxo excepcional;
   - Ponto de Extensão - instância da metaclasse *Group*. Conforme a necessidade de alterações em um Caso de Uso, este artefato é definido pelo RUP com a capacidade de ser estendido a partir de um ponto de extensão. Sendo assim, um ponto de extensão

é um conjunto de informações que possibilita identificar como e onde um Caso de Uso deverá ser estendido;

- Requisito - instância da metaclasse *Group*. Um requisito é uma exigência imprescindível para a consecução de um Caso de Uso, sendo, é um agrupamento de informações sobre exigências especiais;
- Condições, Pontos de Extensão, Fluxos e Requisitos - instâncias da metaclasse *List* que, como visto anteriormente, representa uma estrutura de dados do tipo lista, permitindo a criação de um conjunto de elementos que podem ser ordenados, enumerados e rotulados.

Partindo para o próximo passo, após serem definidos os tipos de informação utilizados podemos então organizar estas informações utilizando os contêineres. Conforme a Figura 5.31, os contêineres estão agrupados no mesmo pacote dos artefatos. Entrentanto, falaremos da modelagem dos relacionamentos entre artefatos, contêineres e tipos de informação utilizando a SwAT posteriormente.

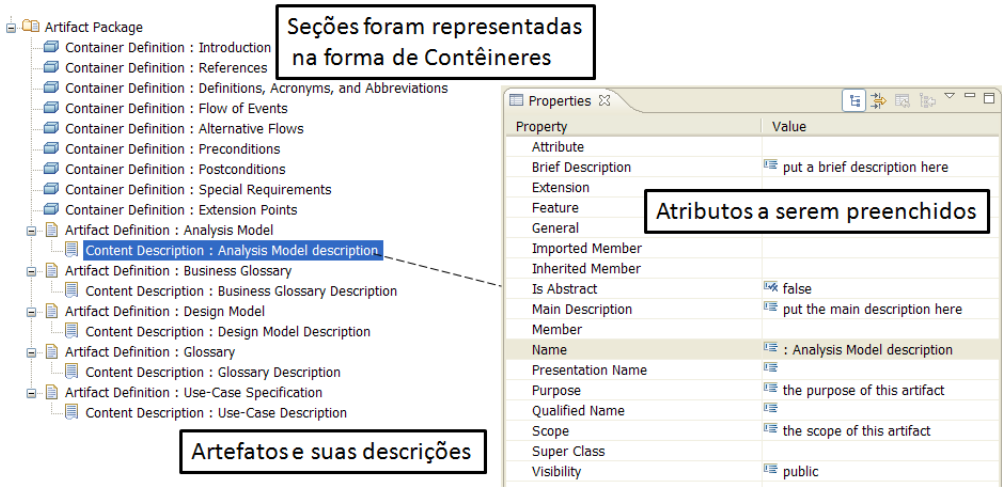

Figura 5.31: Artefatos e Contêineres modelados a partir na ferramenta SwAT

Na Figura 5.31, ainda podem ser vistos os atributos referentes a descrição do artefato Modelo de Análise. A utilização de um conteúdo de descrição permite introduzir informações específicas sobre um artefato, assim como finalidade, escopo e diferentes tipos de descrição.

No caso específico dos artefatos selecionados, utilizamos os contêineres para representar as seções dos artefatos. Sendo assim, todas as seções são instâncias da metaclasse *ContainerDefinitions*. Tais seções, vistas na Figura 5.31 são:

- Introdução - este contêiner foi modelado para poder agrupar o tipo de informação utilizada para descrever um dado artefato;
- Referências - modelado para agrupar os tipo de informação responsáveis por representarem as referências e citações utilizadas em dado artefato;

- Definições, Acrônimos e Abreviações - modelado para agrupar os tipos de informação que estruturam os termos de glossário utilizados em um dado artefato;

- Fluxo de Eventos e Fluxos Alternativos - modelado para conter os tipos de informação necessários para representar os fluxos existentes na execução de um Caso de Uso;

- Pré-condições e Pós-condições - modelado para organizar os tipos de informação necessários para a representação das condições de execução de um Caso de Uso;

- Requisitos Especiais- modelado para organizar os tipos de informação necessários para a representação das requisitos especiais, ou seja, além dos já existentes para um sistema de informação e que servem para a execução de um Caso de Uso;

- Pontos de Extensão - modelado para organizar os tipos de informação utilizados para identificar as extensões existentes em um Caso de Uso.

Os artefatos modelados foram: Especificação de Caso de Uso, Glossário, Glossário de Negócios, Modelo de Análise e Modelo de *Design*. Todos são instâncias da metaclasse *ArtifactDefinitions* e, além disso, possuem uma instância da metaclasse *ContentDescription*, que permite adicionar descrições como finalidade, nome de apresentação, contexto e escopo.

Por último na Figura 5.32 apresentamos a parte do modelo que permite utilização de instâncias de metaclasses de relacionamentos entre artefatos, contêineres e tipos de informação. Nessa mesma Figura, em (i) é visualizada a existência de uma instância da metaclasse *ArtifactContainer_Relationship*, que permite o relacionamento entre artefatos e contêineres, em (ii) estão instâncias da metaclasse *ContainerDefinition_Relationship*, que permite o auto relacionamento entre contêineres, em (iii) estão instâncias da metaclasse *ContainerFragment_Relationship*, que permite o relacionamento entre contêineres e tipos de informação, por fim, em (iv) estão instâncias da metaclasse *ArtifactFragment_Relationship*, que permite o relacionamento entre artefatos e tipos de informação.

Assim como foi visto na modelagem dos tipos de informação, durante a modelagem dos relacionamentos também foram criados três pacotes para facilitar a visualização:

1. este primeiro pacote contém a modelagem dos relacionamentos utilizados pelo artefato Especificação de Caso de Uso, que são:

    - casoDeUso_secoes - relacionamento do tipo “artefato x contêiner” que instancia a metaclasse *ArtifactContainer_Relationship* e relaciona as seções Introdução, Fluxo de Eventos, Pré-condições, Pós-condições, Requisitos Especiais e Pontos de Extensão ao artefato Especificação de Caso de Uso. Além disso, na Figura 5.32 são apresentados os atributos deste relacionamento. Os atributos mais importantes são `name`=casoDeUso_secoes; `indent`=0, denotando que as seções iniciam sem tabulação; `index`=1, definindo que as seções iniciam enumeração a partir de 1;

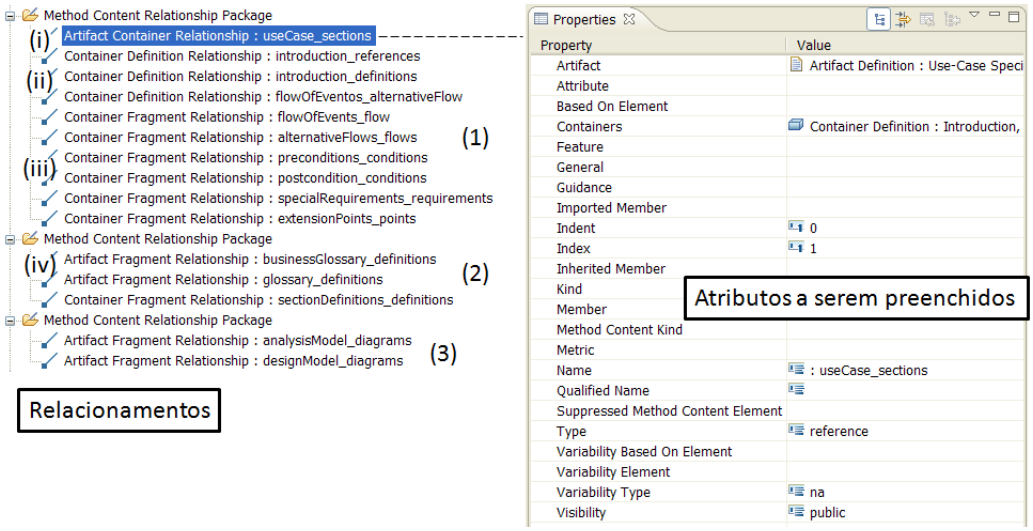

Figura 5.32: Relacionamentos modelados a partir na ferramenta SwAT

`type`=reference, indicando que o tipo do relacionamento possui reuso estrutural por referência; `Artifact`=UseCaseSpecification; `Containers`=Introduction, FlowOfEvents, Preconditions, Postconditions, SpecialRequirements e ExtensionPoints;

- introducao_referencias - relacionamento do tipo “contêiner x contêiner” que instancia a metaclasse *ContainerDefinition_Relationship* e relaciona as seções Introdução e Referências do artefato Especificação de Caso de Uso. Desta forma Introdução será “Pai” de Referências, que será subseção de Introdução;
- introducao_definições - relacionamento do tipo “contêiner x contêiner” que instancia a metaclasse *ContainerDefinition_Relationship* e relaciona as seções Introducao e Definições, Acrônimos e Abreviações, ambas fazem parte do artefato Especificação de Caso de Uso;
- fluxoDeEventos_fluxosAlternativos- relacionamento do tipo “contêiner x contêiner” que instancia a metaclasse *ContainerDefinition_Relationship* e relaciona as seções Fluxo de Eventos e Fluxos Alternativos, ambas fazem parte do artefato Especificação de Caso de Uso;
- fluxoDeEventos_fluxos- relacionamento do tipo “contêiner x tipo de informação” que instancia a metaclasse *ContainerFragment_Relationship* e relaciona a seção Fluxo de Eventos com o tipo de informação Fluxos que é uma instância de *List* e representa o fluxo básico. Fluxos é uma lista que possui o tipo de informação Fluxo, como conteúdo interno. Fluxo é uma instância da metaclasse *Group*;
- fluxosAlternativos_fluxos- relacionamento do tipo “contêiner x tipo de informação” que instancia a metaclasse *ContainerFragment_Relationship* e relaciona a seção Fluxos Alternativos com o tipo de informação Fluxos;
- precondicoes_condicoes- relacionamento do tipo “contêiner x tipo de informação”

que instancia a metaclasse *ContainerFragment_Relationship* e relaciona a seção Pré-condições com as condições a serem satisfeitas antes da execução do Caso de Uso. Como visto anteriormente, condições é uma lista de condição. Esses tipos de informação instanciam as metaclasses *List* e *Group*, respectivamente;

- poscondicoes_condicoes- relacionamento do tipo "contêiner x tipo de informação" que instancia a metaclasse *ContainerFragment_Relationship* e relaciona a seção Pós-condições com as condições a serem satisfeitas após a execução do Caso de Uso;
- requisitosEspeciais_requisitos - relacionamento do tipo "contêiner x tipo de informação" que instancia a metaclasse *ContainerFragment_Relationship* e relaciona a seção Requisitos Especiais com os requisitos necessários para a execução do Caso de Uso. Nesse sentido, como visto anteriormente, requisitos é uma lista de requisito e instanciam as metaclasses *List* e *Group*, respectivamente;
- pontosDeExtensao_pontos - relacionamento do tipo "contêiner x tipo de informação" que instancia a metaclasse *ContainerFragment_Relationship* e relaciona a seção Pontos de Extensão com os pontos de extensão existentes no Caso de Uso. Nesse sentido, como visto anteriormente, pontos de extensão é uma lista de ponto de extensão e instanciam as metaclasses *List* e *Group*, respectivamente.

2. esse segundo pacote contém a modelagem dos relacionamentos utilizados pelos artefatos Glossário de Negócios e Glossário, que são:

   - glossarioDeNegocios_definicoes - relacionamento do tipo "artefato x tipo de informação" que instancia a metaclasse *ArtifactFragment_Relationship* e relaciona o artefato Glossário de Negócios com os tipos de informação Abreviações, Acrônimos, Definições e Termos. Todos estes tipos de informação são instâncias da metaclasse *List*;
   - glossario_definicoes - relacionamento do tipo "artefato x tipo de informação" que instancia a metaclasse *ArtifactFragment_Relationship* e relaciona o artefato Glossário com os tipos de informação Abreviações, Acrônimos, Definições e Termos;
   - secaoDefinicoes_definicoes - relacionamento do tipo "contêiner x tipo de informação" que instancia a metaclasse *ContainerFragment_Relationship* e relaciona a seção Abreviações, Acrônimos e Definições com os seus respectivos tipos de informação.

3. esse último pacote contém a modelagem dos relacionamentos utilizados pelos artefatos Modelo de Análise e Modelo de *Design*, que são:

   - modeloDeAnalise_diagramas - relacionamento do tipo "artefato x tipo de informação" que instancia a metaclasse *ArtifactFragment_Relationship* e relaciona o arte-

fato Modelo de Análise com seus diagramas de Análise de Caso de Uso e de Classes;

- modeloDeDesign_diagramas - relacionamento do tipo “artefato x tipo de informação” que instancia a metaclasse *ArtifactFragment_Relationship* e relaciona o artefato Modelo de *Design* com seus diagramas de *Design* de Caso de Uso e de Classes.

Enfim concluímos todos os passo de criação do *MethodContent* utilizando a SwAT. Como foi apresentado, tanto os tipos de informação quanto os relacionamentos foram modelados em diferentes pacotes para fornecer uma melhor visualização. Entretanto, não é aconselhável a divisão de instâncias da mesma metaclasse em diferentes pacotes. Embora isso não faça diferença no modelo, pode-se cometer erros ao não visualizar as informações por estas estarem em outro pacote. Isso também pode ocasionar uma pequena confusão no conceito de tipos de informação e reuso, pois estes tipos de informação e contêineres não pertencem necessariamente a um artefato específico, pelo contrário, são elementos a serem compartilhados.

Criação do *Process Structure*

Partindo então para o último passo, usaremos tudo que foi definido no *Method Content* para criar o *Process Structure*. Na Figura 5.31 pode ser visto a utilização dos artefatos definidos no *Method Content*. Neste exemplo o uso de elementos de definição não é demonstrado de forma efetiva. Entretanto, isto permite que na camada M1, que contem modelos para PDS específicos, sejam criados processos independentes das modificações feitas no *Method Content*. Como nosso foco é testar a autoria de artefatos, não faz sentido modelar todo um PDS para provar que a divisão entre definição e uso adiciona ganhos na modelagem, uma vez que isso já foi demonstrado em (OMG, 2008b).

A Figura 5.33 apresenta a modelagem do *Process Structure* através da ferramenta SwAT. Como pode ser visto, a estrutura do processo foi criada a partir das classes existentes no *Method Content*. Sendo assim, cada classe do *Process Structure* necessariamente possui uma relação com alguma classe de mesmo tipo do *Method Content*. Este tipo de ligação é chamado de *Method Content Trace*.

A partir da construção do *Process Structure* utilizando a ferramenta SwAT, conseguimos colocar em prática essa parte da abordagem.

Versionamento e Reuso

Tendo em vista os testes feitos sobre a estruturação de artefatos, daremos continuidade a este cenário tratando sobre Versionamento e Reuso de informação, optando por utilizar uma visão com base nas camadas apresentadas na Figura 5.26.

Ao se tratar de grandes quantidades de informações estruturadas, duas grandes questões a serem levadas em consideração são o controle de versão e reutilização. De acordo com o cenário apresentado, essas questões transformam-se em necessidades, assim como em ambientes em que tais informações são manipuladas por várias pessoas e durante um longo período de tempo.

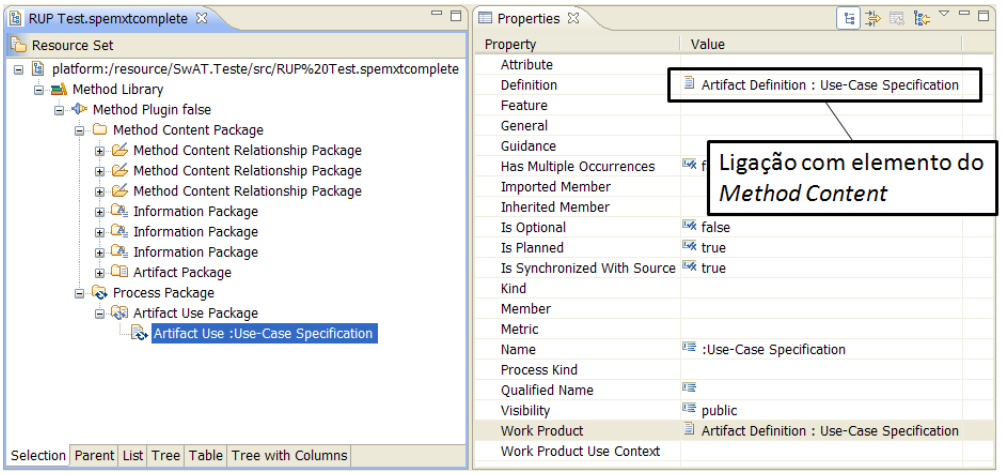

Figura 5.33: Utilização do *Method Content* para criar o *Process Structure*

Versionamento Conforme visto na Seção 4.3.6, por meio versionamento é possível inspecionar as mudanças que ocorreram sobre alguma informação, guardando seus estados. Desta forma, quando necessário, consegue-se reverter para uma mudança em um estado anterior. Entretanto, existem vários fatores que descaracterizam esse tipo de característica, portanto nem sempre é possível manusear todos os vários estados que pode possuir dada informação.

Atualmente, o controle de versões sobre artefatos é feito através de software do tipo SVN (Subversion) [2], dado o fato que apenas objetos reais, de nível M0 são versionados. Além disso Softwares como esse tratam artefatos conforme blocos monolíticos de informação, impossibilitando discussões sobre os detalhes das modificações e possíveis inferências, como qual informação sofreu maiores alterações, por exemplo.

Embora sistemas como o SVN sejam bem adaptados quanto a utilização de artefatos apenas texto, assim como código-fonte ou arquivos do tipo txt, eles falham quando se trata de documentos mais complexos, assim como documentos do tipo *Office*, como documentos do *Microsoft Word*™e *Microsoft Excel*™.

Reuso A reutilização permite que as informações sejam organizar com prudência, já que evita duplicações de partes de conteúdo ou estrutura comum. Hoje em dia a reutilização de artefatos de software é feita através do uso de *templates*, que possuem espaços em branco a serem preenchidos. Embora essa solução permita a reutilização da estrutura de artefatos, ela falha quanto ao reuso de conteúdo e, o que é ainda pior, reforça o uso de procedimentos do tipo ´´cópia-e-cola”, o que pode facilmente propagar as informações e gerar diversas inconsistências.

Em nossa abordagem, o reuso de informação é baseado em pontos de variabilidades. Tais pontos consistem em firmar relacionamentos entre elementos de mesmo tipo, determinando características comuns ou particularidades. Embora este tipo de nomenclatura seja mais usual em trabalhos sobre adaptação e *tailoring* de PDSs, também é possível determinar o reuso de infor-

[2] subversion.tigris.org

mação utilizando esse mesmo princípio. Nesse sentido, o próprio SPEM v2 especifica pontos próprios de variabilidade e, além disso, indica o curso a ser seguido para construir famílias de PDSs.

Entretanto, como o foco desta abordagem é a autoria de ASs, esta característica do SPEM v2 foi extendida, conforme pode ser visto na Seção 4.3.7. Dessa forma, adicionamos maior semântica e novas regras para a utilização do reuso, possibilitando, além do reuso de estrutura, o reuso de conteúdo.

Integração Desta forma, essa seção tem como objetivo mostrar como nossa abordagem funciona em relação a controle de versão e reuso. Tais características já estão integradas, provendo a capacidade de acompanhar as mudanças na estrutura e conteúdo de artefatos.

Para maior facilidade no entendimento, uma visão geral é apresentada Figura 5.34. Essa figura apresenta uma ilustração dessas capacidades, mostrando diversas revisões nas camadas M2, M1 e M0. Ela está organizada como uma tabela na qual as colunas indicam as revisões e as linhas indicam qual a camada e os os papéis que são responsáveis pelas alterações. Além disso em cada célula da tabela existe um conjunto de classes e observações que indicam o que foi modificado. É importante entender que na Figura 5.34, as barras cinzas no alto de cada linha diz respeito à versão do metamodelo, processo ou projeto.

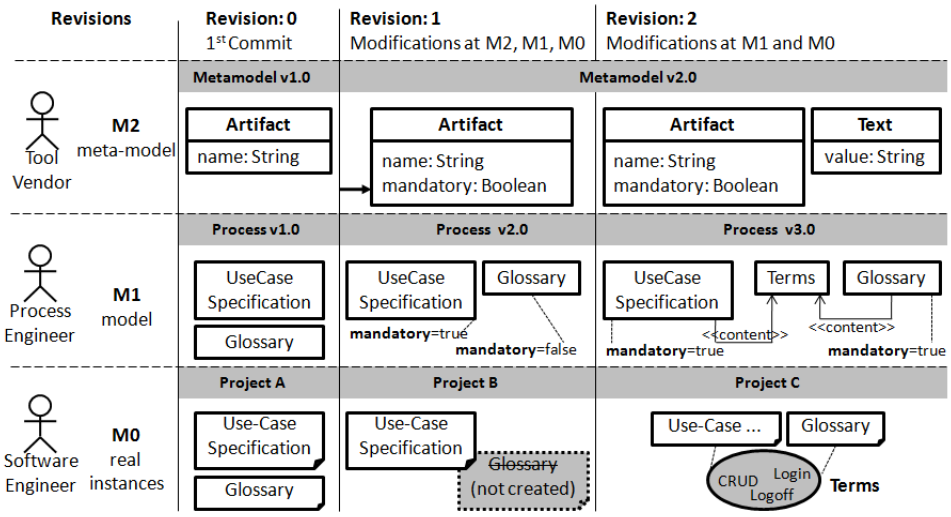

Figura 5.34: Exemplo de utilização de versionamento e reuso

A Figura 5.34 também apresenta três atores, na coluna mais a esquerda, indicando qual o papel responsável pelas modificações em cada camada: o *Tool Vendor* é responsável por manter o metamodelo para autoria de artefatos, ou seja, a camada M2; o Engenheiro de Processo é responsável por criar os modelos para definição e uso de artefatos utilizando a camada M2, ou seja, é responsável pela camada M1; por fim, o Engenheiro de Software é responsável por criar o projeto na camada M0 que utilizará um processo definido na camada M1.

O mecanismo de versionamento pode ser visto a partir da `Revision:0` (segunda coluna),

quando foi feito o primeiro *commit*. Nesta coluna uma representação da metaclasse *ArtifactDefinition* da camada M2, que foi utilizada para moldar os artefatos Especificação de Caso de Uso e Glossário, da camada M1.

Já na terceira coluna está a `Revisão - 1`. Nessa revisão houve uma pequena modificação em *ArtififactDefinition*. Agora, ela possui o atributo *mandatory* que indica quando um artefato é, ou não, obrigatório no processo de desenvolvimento, indicando se ele deve ser criado e preenchido. Ou seja, foi feita uma modificação apenas no nível M2, que possivelmente implicaria em uma nova versão da ferramenta de modelagem de artefatos, que deve ser feita pelo *Tool Vendor*. Por esta razão, os artefatos modelados a partir do metamodelo da `Revisão - 1` indicam se são obrigatórios. No exemplo apresentado na Figura 5.34, o artefato Especificação de Caso de Uso é obrigatório, enquanto que Glossário não o é.

Outro fato que pode ser visto na `Revisão - 1` é que conforme houve modificação em M2, os modelos M1 e M0 também foram afetados. Nesse sentido, ao alterar o metamodelo de nível M2, criando a nova `Revisão - 1`, todas as novas instâncias de nível M1 se utilizarão dessas alterações. Da mesma forma, um novo projeto de nível M0 que instanciar um processo modelado em M1 a partir da nova revisão em M2. Entretanto, em nossa abordagem, os projetos anteriores a essa revisão, assim como os processos modelados, não partilharão dessas alterações.

A `Revisão - 2`, que se encontra na quarta coluna, não apresenta modificações no nível M2, mas sim em M1. Embora a metaclasse *Text* esteja presente em ambas as revisões do metamodelo, ela, assim como o restante do metamodelo, foram suprimidos para evitar confusão. Sendo assim, conforme está apresentado na Figura 5.34, a camada M1 está organizada de maneira a demonstrar o reuso. Como se sabe, os elementos criados na camada M1 são de responsabilidade do Engenheiro de Processos. Sendo assim, no `Processo v3.0` foi feito reuso através da metaclasse *Terms*, que é utilizada por ambos os artefatos. A decisão de compartilhar essa informação foi feita para tornar os termos consistentes e sincronizados. No exemplo da Figura 5.34, o `Projeto C` utiliza instâncias da Especificação de Casos de Uso e do Glossário para compartilhar os termos *CRUD*, *Login* e *Logoff*.

É importante perceber que o mecanismo de versionamento utilizado na nossa abordagem é baseado na integração com a camada M3, como visto na Figura 5.26. Como resultado, todos os elementos do Metamodelo para a autoria de ASs estão relacionadas com a metaclasse *VersionedExtent*, do *MOF Versioning*. Além disso, o *MOF Versioning* provê metaclasses para: controle e gerência de versionamento através de operações de: *check-in, check-out, revert, delete, commit* e *look-up versions*; **identificação** de versão, utilizando ids e rótulos; marcação de versões através de datas e **timestamps**; histórico de versões; utilização de sessões em revisões inseridas em uma *workspaces*; e, por fima criação de versões de **baselines** conforme suas configurações.

Criação de instâncias reais de Nível M0

Por fim, embora o *enactment* não faça parte do objetivo deste trabalho, para que o cenário de uso esteja completo apresentamos também como se dá a camada M0, entretanto, isto não

é suportado pela ferramenta. Como definido anteriormente, esta camada contem as instâncias reais dos artefatos definidos no modelo M1. Desta forma, esta camada estará em conformidade com a camada M1. Uma amostra das instâncias feitas nessa camada pode ser vista na Figura 5.35.

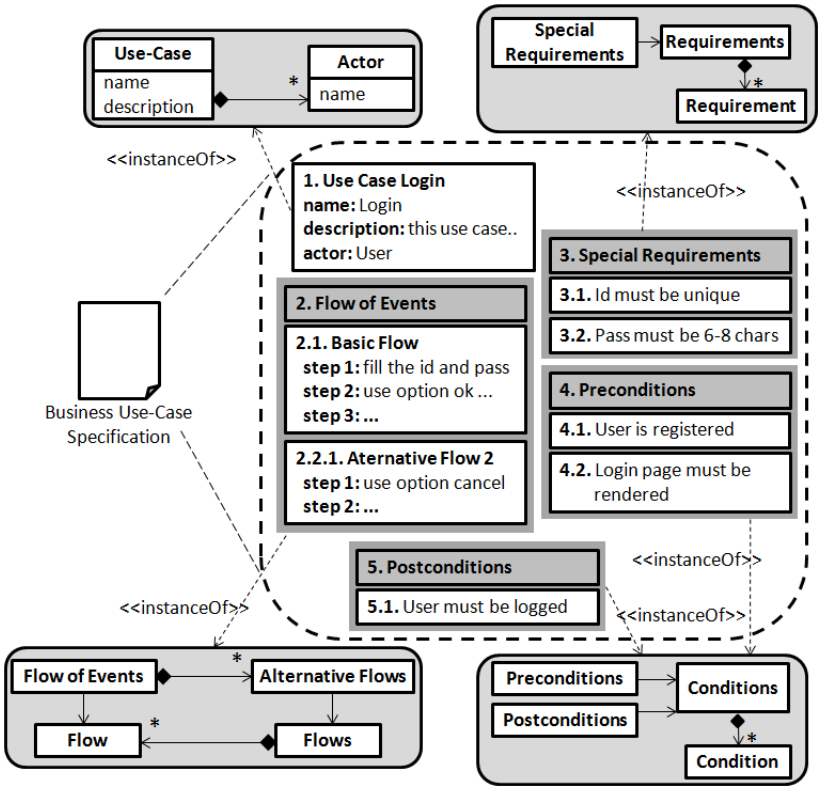

Figura 5.35: M0 Use-Case real instance using M1 Layer

Neste caso a Figura 5.35 ilustra uma instância real do artefato Especificação de Caso de Uso, utilizando anteriormente. Como pode ser visto, as caixas que estão dentro dos quadrados de contornos pontilhados representam os conteúdos deste artefato. Sendo assim, essas caixas são instâncias das classes correspondentes, que estão dentro das caixas em tom cinza e que foram definidas na Camada de Processos.

Ao observar a Figura 5.35 é possível visualizar que o artefato é uma instância da classe *UseCaseSpecification*, podendo então ser preenchido utilizando informações de um projeto real. Para fins de exemplo nós consideramos um caso de uso para o caso de `Login`. Esse caso de uso possui: nome; descrição; fluxos, um fluxo básico e alguns alternativos; requisitos; pré-condições e pós-condições; além de atores.

Nesse exemplo temos que:

- `Login` é um caso de uso bastante usual e que pode ser executado quando actor=*User*;
- Os fluxos definem o que o `User` deverá fazer para executar uma determinada ação do Caso de Uso. Ou seja, o fluxo básico define o fluxo mais evidente ou esperado (como

no exemplo: “*use the ok option*”), enquanto que os fluxos alternativos definem os passos para algum comportamento excepcional (como no exemplo: “*use the cancel option*”);

- os requisitos definem o que é necessário para executar um caso de uso, no exemplo existem os requisitos `3.1 Id must unique` e `3.2 Password must be 6-8 characters`, que na verdade mostram a necessidade de restrições para a validação dos campos `Id` e `Password`;

- as pré-condições 4.1 e 4.2 descrevem o estado inicial para a execução do caso de uso, ou seja, é necessário que o usuário a fazer *login* esteja cadastrado e que a página com o formulário de *login* esteja renderizada;

- a pós-condição 5.1 da uma idéia do estado esperado após a execução do caso de uso. Desta forma, é essencial que o `User` esteja “logado” na aplicação.

5.2.2 Cenário de Teste 2

Nesse segundo cenário é apresentada a utilização de *UML Profiles* para gerar instâncias do metamodelo da UML, criando assim um modelo UML no nível de usuário. O metamodelo para autoria de ASs e que define a camada M2 pode ser representado através de um catálogo de estereótipos, definindo o *SPEMXt UML Profile*. Entretanto, conforme a especificação da UML (OMG, 2007b), este tipo de extensão é bastante limitado em comparação a extensão por meio do próprio metamodelo.

Diante de algumas limitações, a autoria de artefatos não pode ser feita da mesma forma como no cenário de teste anterior, (Seção 5.2.2). Partindo desse princípio, temos em mente que ao utilizar o catálogo de estereótipos perderemos algumas restrições necessárias mas que não puderam ser definidas, assim como os valores semânticos existentes em alguns conceitos destacados nesta dissertação. As limitações são:

- Os relacionamentos obedecerão apenas os tipos de associações da UML, não suportando as restrições para relacionamentos entre diferentes tipos, assim como “artefato x contêiner”, “contêiner x artefato”, “contêiner x tipo de informação” e “artefato x tipo de informação”.

- Os níveis de reuso podem ser utilizados, entretanto são meramente ilustrativos.

- O versionamento não pode ser garantido. Uma vez que este mecanimos foi implementado na camada M3 é necessário que a distribuição UML a ser utilizada tenha sido construída com base no *MOF Repository*.

- A regras de restrição do metamodelo, definidas em linguagem OCL, foram utilizadas.

Assim como no cenário anterior, iremos utilizar uma ferramenta para facilitar esta tarefa. Embora tenhamos desenvolvido uma ferramenta própria para apresentar o Cenário de Teste 1, neste cenário, por motivos de preferência e maior viabilidade, utilizamos a *CASE Tool* proprietária *IBM Rational Software Modeler*. Entretanto, esta ferramenta não implementa o Guia de Autoria de Artefatos, sendo assim, seguiremos o Guia manualmente.

O catálogo com os estereótipos do *SPEMXt UML Profile* está definido no Apêndice A.8. As principais metaclasses foram mapeadas para estereótipos e seus atributos como *tagged values*.

Criação de Biblioteca

Da mesma forma que no primeiro cenário, queremos criar um novo PDS para iniciar a autoria. Como estamos utilizando o *IBM Rational Software Modeler* devemos criar um novo Projeto para modelar UML. A Figura 5.36 apresenta a criação desse novo Projeto de Modelagem, que pode ser visto em **a)**. A criação de projetos utilizando essa ferramenta é feita utilizando *wizards*. Sendo assim, o nome do Projeto, que foi denominado `Profile Test` pode ser visto em **b)**. Já em **c)** esta nova biblioteca que foi feita através da adição de um novo Modelo de nome `RUP Test`.

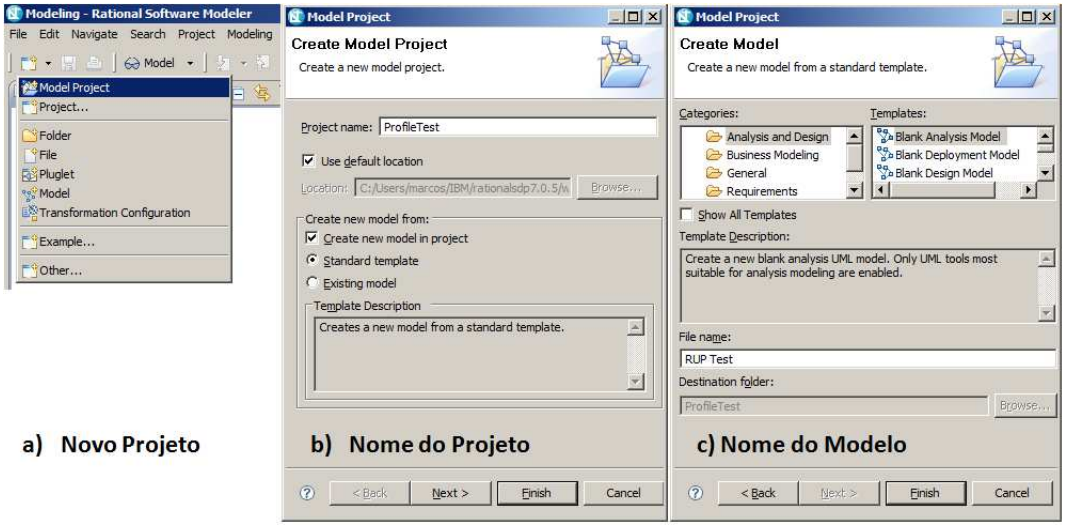

Figura 5.36: Nova autoria de PDS utilizando o *SPEMxt UML Profile*

O resultado que foi gerado a partir do *wizard* de criação pode ser visto na Figura 5.37. Essa Figura apresenta a estrutura gerada pela ferramenta e a criação do pacote de modelagem, vistos em **a)**. Na descrição do modelo colocamos: “Uma biblioteca para testes utilizando o RUP”, visto em **b)**, como no cenário anterior.

Agora já podemos criar a biblioteca adicionando um empacotador para o modelo. Como se trata do uso de UML em nível de usuário, adicionamos um pacote de nome RUP Library, marcando-o com o estereótipo `MethodLibrary`.

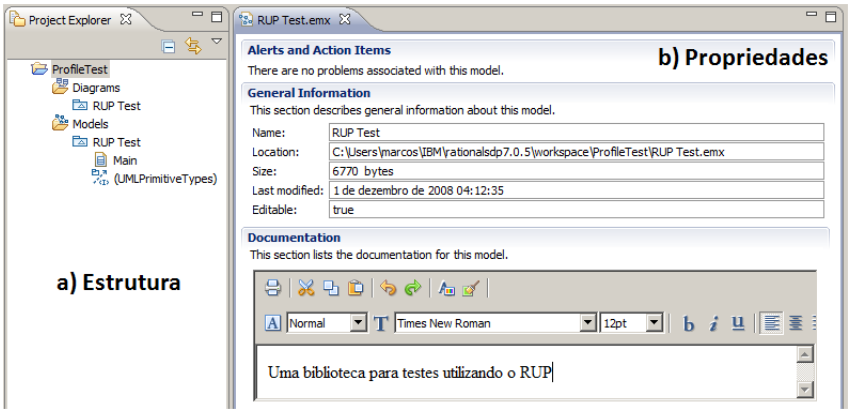

Figura 5.37: Projeto de modelagem utilizando *SPEMxt UML Profile*

Criação do *Method Content*

O próximo passo é criar um novo PDS dando início a autoria. Nesse sentido, a autoria de artefatos deve iniciar com a criação de um *Method Content*. Para este fim foi criado o pacote UML de nome *Content* utilizando o estereótipo *MethodContent*, para identificá-lo como um pacote de conteúdo da autoria.

Na Figura 5.38 estão os pacotes utilizados para a criação de um *Method Content*. Desta forma, nessa Figura fica evidenciada a criação de outros três pacotes, visto em (A): *artifacts*, *containers* e *fragments*, com os respectivos estereótipos, *ArtifactPackage*, *ContainerPackage* e *InformationPackage*. Com a utilização de estereótipos também é possível criar vários pacotes de informação. Isso pode facilitar a divisão de fragmentos, porém, dificulta a visualização de um possível reuso posterior. Sendo assim, optamos por dividir apenas a visualização dos artefatos, utilizando por base a criação de *namespaces*. Desta forma, foram definidos diagramas UML de nível de usuário. Tais diagramas são utilizados para gerar abstrações dos artefatos conforme a perspectiva necessária, visto em B.

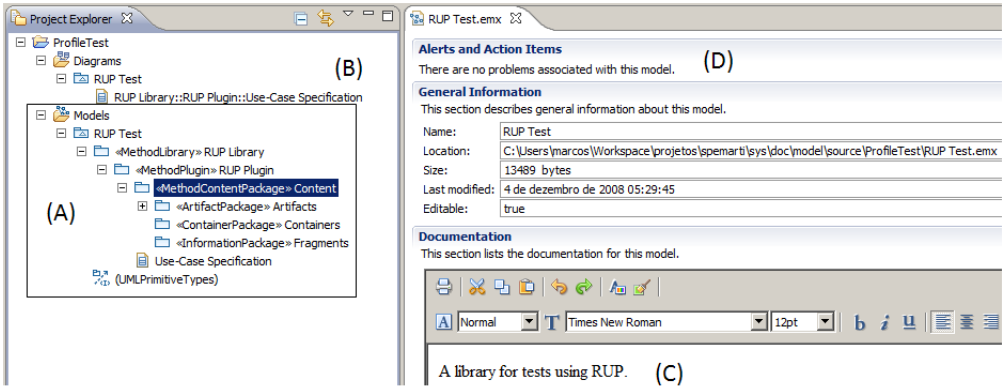

Figura 5.38: *SPEMxt UML Profile*: Definição do *Method Content*

Além disso, na Figura 5.38 também é possível visualizar o pacote de biblioteca criado anteriormente, estando marcado com o estereótipo *MethodLibrary*, também visto em (A). A descrição

da biblioteca pode ser vista em (C) e as propriedades e atributos estão dispostas em (D).

Durante a criação do *Method Content* foi feito um paralelo entre o conceito de artefato a ser criado e seu modelo, assim como no Cenário de Teste 1. Entretanto, no atual cenário de testes foi levado em consideração apenas o catálogo de estereótipos, que é mais restrito do que o metamodelo utilizado no cenário de teste anterior.

Para a utilização de um *UML Profile*, deve-se criar uma instância de uma metaclasse e definir qual estereótipo ela possuirá. Dessa forma, se quisermos aplicar um estereótipo a uma classe, devemos adicionar esta classe ao modelo UML e depois marca-la com o estereótipo. Entretanto, deve-se tomar cuidado em relação a metaclase **base** do estereótipo. A exemplo, caso o estereótipo possua uma metaclasse do tipo relacionamento, este não pode ser aplicado em classes, mas sim, em associações. Todos os estereótipos do catálogo podem ser vistos no Apêndice A.8, assim como suas respectivas metaclasses base.

Na Figura 5.39 está explicitado o mapeamento através do profile. Como pode ser visto, embora a o profile possua diversas limitações, o mapeamento de um artefato utilizando os estereótipos não se demonstra totalmente diferente do mesmo mapeamento feito através do uso do metamodelo. Desta forma, assim como no primeiro cenário, apresentaremos os tipos de informação, contêineres e artefatos modelados, assim como seus devidos relacionamentos.

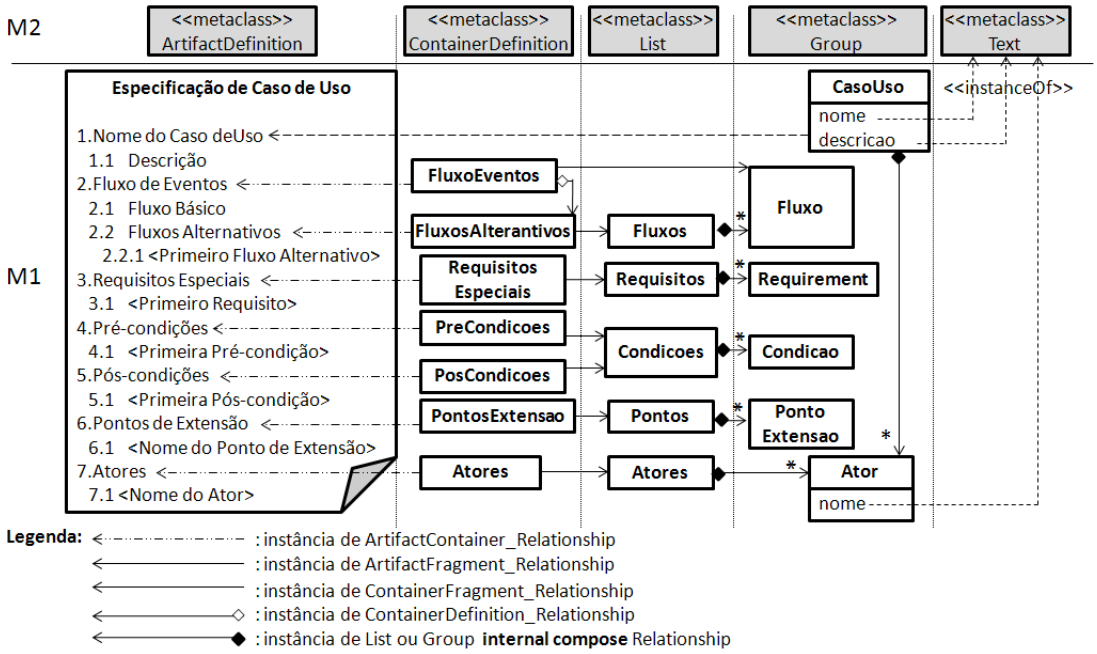

Figura 5.39: Exemplo de Mapeamento para o *SPEMxt UML Profile* utilizando o artefato Especificação de Caso de Uso

Embora sejam duas extensões totalmente diferentes do ponto de vista da UML, pode-se perceber através da Figura 5.39, que conceitualmente o resultado não parece demonstrar muitas diferenças. Entretanto, é muito mais difícultada se chegar nesse resultado através dos estereótipos do que através do metamodelo, uma vez que existe menos semântica e restrições.

Voltando o foco para a definição do modelo através dos *SPEMxt UML Profile*, no pacote *artifacts* foram inseridos todos os artefatos pretendidos. As seções desses artefatos foram modeladas dentro do pacote *containers*. Por fim, todas as informações foram introduzidas no pacote *fragments*.

Na Figura 5.40 são apresentadas todas as instâncias do tipos de informações modeladas através do uso de estereótipos do *SPEMxt UML Profile*. Esses tipos são os mesmos apresentados no primeiro cenário. Entretanto, no *Project Explorer* pode ser visto que todos eles são modelados a partir de classes UML estereotipadas. Em nosso modelo, tais classes possuem o nome do tipo da informação que representam no artefato e um estereótipo que determina qual o tipo da informação que está sendo utilizada. O diagrama *Fragments*, exposto em (A), agrupa todos os tipos de informação deste pacote, criando uma visão lógica que pode ser vista em (C). O auto-relacionamentos entre os tipos de informação estruturados são definidos pelo estereótipo *FragmentRelationship* e estão agrupados no pacote *associations* (B), que faz parte do modelo. A notação utilizada para expressar esses tipos de relacionamento é uma seta com diamente escuro, utilizado para denotar associações de composição da UML.

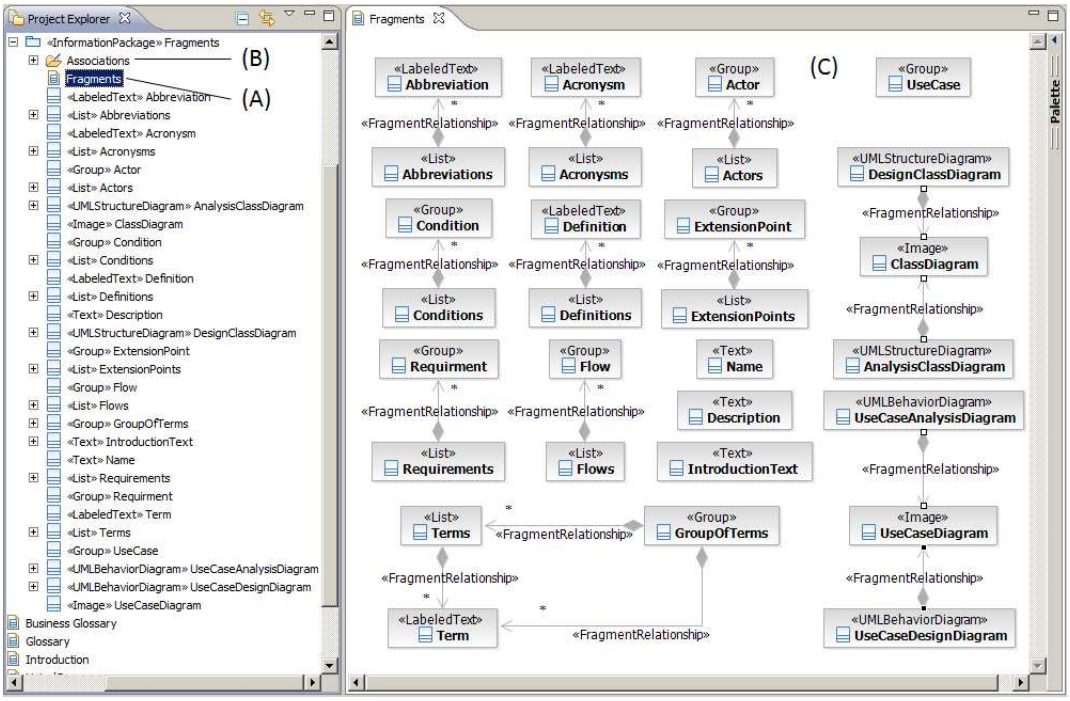

Figura 5.40: *SPEMxt UML Profile*: Modelagem dos Tipos de Informação

O próximo passo do Guia para Autoria é a modelagem dos contêineres. Desta forma, adicionamos todos os contêineres mapeados dentro do modelo adicionando-se o estereótipo *ContainerDefinition*. Além disso, foram definidos os relacionamentos entre os contêineres, todos feitos através do estereótipo *ContainerDefinitionRelationship* e expressados através de setas com diamante branco, utilizados para denotar associações de agregação da UML.

Esses contêineres podem ser vistos na Figura 5.41, que apresenta um diagrama de nome *Containers* (mostrado em (D)). Além disso, nessa Figura pode também ser visto a aplicação

do estereótipo *ContainerDefinition* ao contêiner *Actors* (em (E)), assim como os *tagged values* existentes já previamente preenchidos (pode ser visto em (F)):

- *briefDescription*, utilizado para expressar uma descrição resumida do contêiner;
- *mainDescription*, utilizado para expressar a descrição principal do contêiner;
- *kind*, herdado do catálogo de estereótipos do SPEM v2, esse atributo não faz sentido nesse contexto, por isso não foi utilizado nesse caso, entretanto, seu valor determina qual metaclasse da UML ele esta representando;
- *presentationName*, utilizado para expressar um nome de apresentação para um contêiner;
- *purpose*, utilizado para expressar o propósito ou a finalidade do contêiner;

Por fim, na Figura 5.41 estão todos os relacionamentos que foram adicionados ao modelo, vistos através da espansão do pacote *associations*.

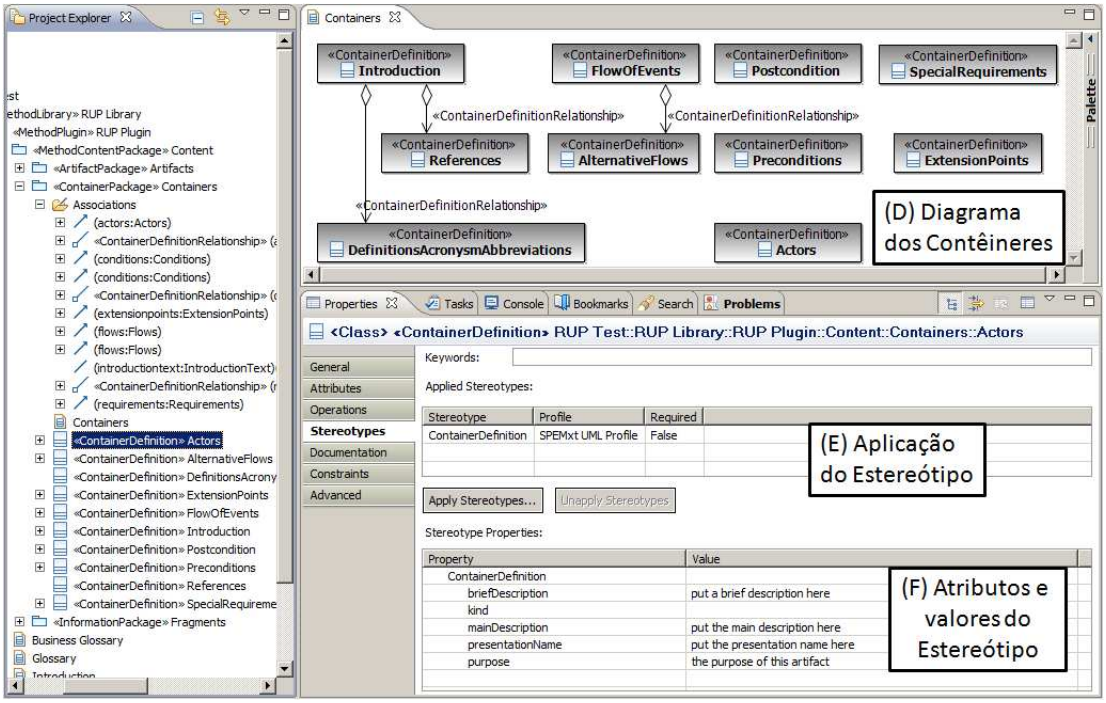

Figura 5.41: *SPEMxt UML Profile*: Modelagem dos Contêineres

O próximo passo a ser seguido é a modelagem dos artefatos. Adicionamos todos os artefatos mapeados, vistos na Figura 5.39, ao modelo, criando também os relacionamentos entre eles, os contêineres e os tipos de informação existentes. Os artefatos foram modelados através de classes UML com a aplicação do estereótipo *ArtifactDefinition*. Além disso, os relacionamentos existentes foram marcados com o estereótipos:

- *ArtifactContainer_Relationship*, quando os relacionamentos são ligações entre artefatos e contêineres;

- *ArtifactFragment_Relationship*, quando os relacionamentos são ligações diretas entre artefatos e tipos de informação;
- *WorkProductDefinition_Relationship*, quando são utilizados auto-relacionamentos entre artefatos.

A notação utilizada para representar estes relacionamentos é uma seta com linha sólida, mas apenas a notação do auto-relacionamento determinado pelo estereótipo *WorkProductDefinition_Relationship* possui navegabilidade. Conforme pode ser percebido, existe uma notação própria para cada tipo de relacionamento. Isto foi feito com o pensamento de obter uma maior representatividade no tocante a visibilidade dos modelos construídos a partir do *SPEMXt UML Profile*. Em todo caso, não existe nenhuma regra ou determinação semântica explicita para que isto seja feito.

Na Figura 5.42 é apresentada a parte final da modelagem do *Method Content*. Em (F) estão todos os artefatos, modelados através de classes UML com a aplicação do estereótipo *ArtifactDefinition*. Em (G) estão os diagramas criados para definir diferentes visões lógicas do modelo, em cada uma delas, está um dos artefatos presentes nos testes da abordagem. Todos esses diagramas podem ser visualizados no Apêndice B.2. Mais uma vez os relacionamentos podem ser vistos no pacote *associations*, em (H). Estes relacionamentos.

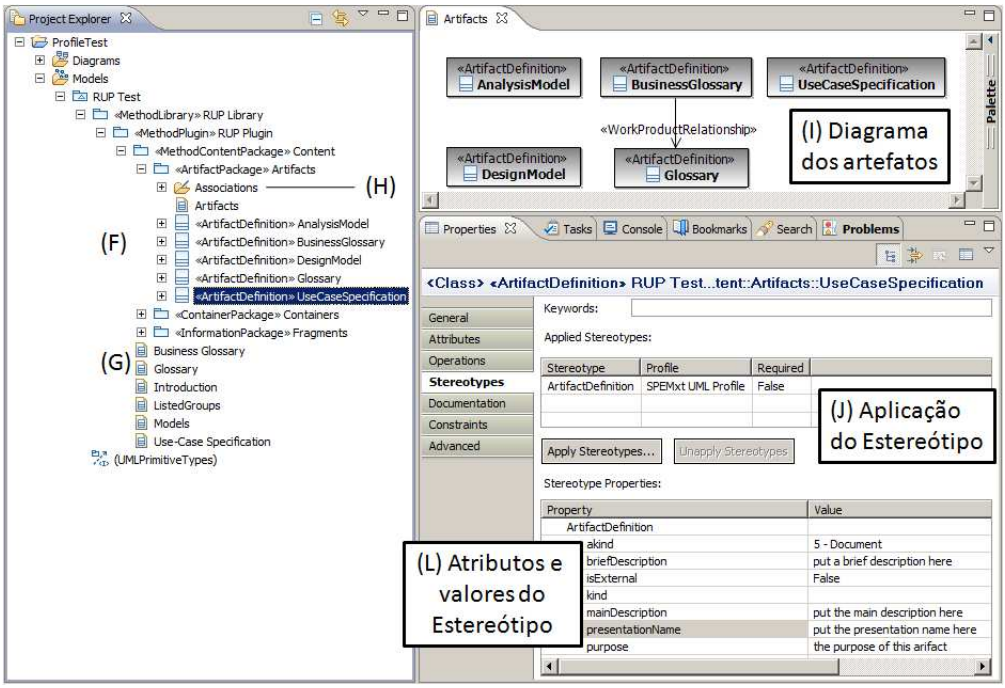

Figura 5.42: *SPEMxt UML Profile*: Modelagem dos Artefatos

Criação do *Process Structure*

A modelagem do *Process Structure* acontece da mesma forma como no primeiro cenário, ou seja, basta definir um elemento de processo que precise utilizar um elemento de definição

do *Method Content*. Conforme a Figura 5.43, criamos o uso do artefato Especificação de Caso de Uso, lado direito da Figura, que pertence ao pacote *AtifactsUse*, com o estereótipo *Artifact-Package*. Além disso, o pacote que estrutura o *Process Structure* possui o estereótipo Desta forma, utilizamos o diagrama ArtifactUse, visto a direita da Figura, para criar a visão lógica que representa a ligação entre o Uso do artefato e sua definição. Esta ligação foi feita através do relcionamento de dependência definido pelo próprio SPEM v2.

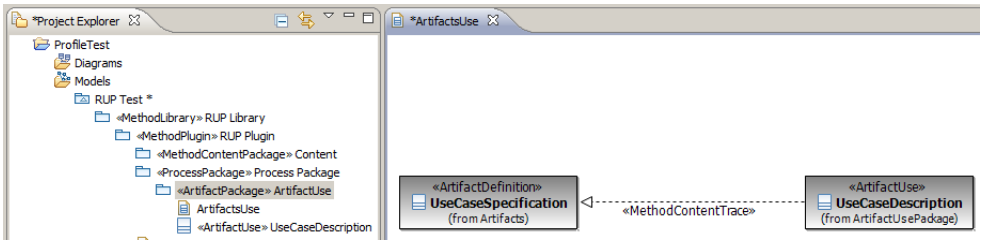

Figura 5.43: *SPEMxt UML Profile*: modelagem do *Process Structure*

5.3 Conclusão

Nesse capítulo foram utilizados dois cenários para avaliar a abordagem proposta, de forma a comprovar, mesmo que de forma analítica, as hipóteses levantadas na Seção 4.8. Com o uso de um metamodelo e o UML Profile, ambos para a modelagem de artefatos de software, foi possível definir sua autoria. A ferramenta SwAT, elaborada especificamente para fins de teste do mtamodelo se mostrou bastante prestativa, permitindo efetuar uma avaliação em forma de prova de conceito, apresentando os problemas e erros existentes, permitindo suas correções.

Além disso, os testes feitos através da modelagem de artefatos reais, de processos reais, mostraram que a abordagem permite atingir o objetivo deste trabalho, que era prover a autoria de artefatos determinando maiores detalhes sobre suas estruturas de informação, Seção 1.4. Por isso, afirmamos que, com bases nos testes, vencemos os desafios lançados no capítulo introdutório desse trabalho:

1. Como construir artefatos de forma que sua organização interna esteja bem estruturada, facilitando sua compreensão e manipulação?

2. Como definir diversas versões, autores e estruturas de informação aos artefatos durante sua autoria, evitando a redundância de informação, de estrutura e retrabalho?

Embora não tenham sido feitos testes formais, algum estudo de caso aplicado em ambiente real, ou algum tipo de experimento, somos confiantes em afirmar que, devido aos testes feitos, as hipóteses:

- (**H1**) pode ser ao menos parcialmente satisfeita, uma vez que: os artefatos estão estruturados através de UML, que é uma estruturação bem formada; há categorização dos artefatos; tipificação das informação; organização estrutural e de conteúdo; controle de versão; e reuso. Entretanto, não podemos provar formalmente, que nossa abordagem é melhor nesse sentido, embora exista um sentimento positivo quanto a isso.

- (**H2**) não pode ser comprovada sem um teste exaustivo ou prova formal. Entretanto, a utilização do Guia para a Autoria de Artefatos de Software, implementado tanto diretamente na ferramenta SwAt, no contexto do metamodelo, quanto que manualmente, durante a utilização do catálogo *SPEMXt UML Profiles*, facilitou a criação dos exemplos e dos testes. Podemos afirmar ainda, com base na experiência que tivemos na execução dos testes, que sem um Guia, a autoria de artefatos em detalhes poderia ser dificultada, uma vez que nenhuma das abordagens encontradas utiliza o mesmo paradigma.

- (**H3**) pode ser atendida, pois foi possível criar diferentes níveis de reuso através da utilização dos pontos de variabilidade definidos sobre elementos de *Method Content*. Essa hipótese pode ser ainda expandida para englobar PDSs, uma vez que a especificação do SPEM v2 possui muitas provas de conceito implementadas através de ferramentas, diagramas e provas analíticas.

Por fim, diante dos destes feitos, corrigimos os problemas encontrados tanto na abordagem, quanto na sua implementação. Tais problemas são descritos a seguir:

- **Problemas resolvidos no contexto da abordagem:**

    1. inicialmente, os níveis de reuso definidos foram *content* e *extends*, pois imaginamos que isso seria suficiente. Porém, durante as primeiras tentativas de testes foram encontrados problemas no reuso de algumas partes dos artefatos, principalmente artefatos com grandes quantidades de informação e em versões mais maduras. Desta forma, foram adicionados os conceitos de *reuse*, *local contribution* e *local replacement*, com base no que já havia sido definido no próprio SPEM v2.
    2. o versionamento fora pensado de forma com que fosse parecido com um sistema de controle de versão do tipo SVN. Desta forma resolveríamos o problema, porém, houve grande dificuldade na implementação desse mecanismo. Por isso, foi pensado em utilizar o próprio SVN como solução, mas utilizando artefatos estruturados. Entretanto, voltaríamos ao mesmo problema dos artefatos monolíticos, pois existiria a mesma dificuldade em identificar o que exatamente foi modificado. Como solução definitiva, utilizamos o próprio *MOF Versioning Life-Cicle*, que provê um repositório (*MOF Repository*) próprio para armazenar qualquer elemento definido a partir da camada M3, em qualquer versão.

- **Problemas resolvidos no contexto da implementação como metamodelo:**

    1. a enumeração com os valores para os tipos de reuso foi alterada para compartar todos os novos tipos da abordagem.
    2. os relacionamentos entre artefato e tipo de informação e entre contêiner e tipo de informação foram modificados, surgindo uma super classe para definí-los de forma a aproveitar sua estrutura.
    3. todos os relacionamentos ligados a artefato, diretamente ou por transitividade foram alterados estruturalmente. Cada um deles virou subclasse da metaclasse responsável por satisfazer a característica de reuso. Desta forma, define-se o reuso no próprio relacionamento entre os elementos, ou seja, "nas arestas" e não nos elementos, como havia sido modelado.
    4. a modelagem inicial do versionamento foi removida, dando espaço ao uso da camada M3, junto ao MOF, para este fim.

- **Problemas resolvidos no contexto da implementação como *UML Profile*:**

    1. foram adicionados os estereótipos *FragmentDefinition* e *ContainerPackage*. O estereótipo foi adicionado por falta de uma visualização explicita dos relacionamentos entre as informações que possuem conteúdos internos, assim como lista, grupo e tabela. Já o segundo, foi adicionado mediante problemas de visualização dos contêineres quando modelados dentro do mesmo pacote em que estariam os artefatos.

- **Problemas resolvidos no contexto da utilização da ferramenta SwAT:**

    1. foram criados pacotes para tipos de informação, relacionamentos e artefatos, provendo uma melhor organização do modelo. Após a introdução de alguns artefatos, notamos que a visibilidade do modelo era bastante comprometida.
    2. a perspectiva padrão foi alterada para prover maior visibilidade do modelo na ferramenta.
    3. foram adicionados ícones auto-descritivos numa tentativa de melhorar o entendimento do modelo, permitindo associar uma instância a sua metaclasse.

6 Conclusão

Conforme a necessidade de utilização de processos para a construção de produtos, em Engenharia de Software, o foco dos Processos de Desenvolvimento de Software é visar a um Produto de Software final a partir da construção de vários Artefatos de Software. Diante deste fato, problemas no tratamento de tais artefatos caracterizam um provável erro de paradigma, visto que ao tratá-los como elementos monolíticos, os processos desconhecem as interações e mudanças ocorridas nos artefatos durante sua evolução.

Diante deste contexto foram apresentados alguns desafios e os problemas descritos no Capítulo 1, todos culminando na solução de um objetivo comum:

> *Construir artefatos de forma que sua organização interna esteja bem estruturada e definir diversas versões, autores e estruturas de informação aos artefatos durante sua autoria.*

A partir do qual foi criada a Questão de Pesquisa para este trabalho:

> *Como deve ser a autoria de Artefatos de Software de forma que estes não sejam monolíticos?*

Neste sentido, a construção de um modelo capaz de facilitar a estruturação e manipulação desses artefatos traz um novo paradigma, uma vez que deixa de orientar o processo as atividades, aumentando o foco nos artefatos, uma vez que serão mais bem estruturados.

Desta forma, neste trabalho foi feito uso de linguagens de especificação que permitem a representação de informações relacionadas aos artefatos presentes no processo utilizando conceitos de modelagem Orientada a Objetos. Para que tal feito pudesse ser concretizado, houve a extensão do metamodelo SPEM v2 e da UML, apresentando soluções para versionamento, organização e estruturação lógica interna tanto no conteúdo quanto na estrutura, utilizando-se de informações com tipos bem definidos e compartilhados.

Por fim houve a criação de um ferramental para nossa abordagem que permitiu avaliar a capacidade de utilização do metamodelo. A idéia de autoria de artefatos, juntamente com seus detalhes foi concretizada a partir do seu uso no protótipo SwAT, visto no Capítulo 5, dando conta da solução das problemáticas citadas no Capítulo 1 e atendendo o objetivo supracitado.

6.1 Limitações do Estudo

Nesta seção são apresentadas algumas limitações deste trabalho:

1. Durante a avaliação da proposta não foram realizadas verificações da autoria de PDSs com foco em detalhamento de ASs em processos inteiros, assim como os utilizados em indústria de grande porte. Além disso, não foram feitos estudos empíricos.

2. A ferramenta SwAT ainda é apenas um protótipo, sendo necessário melhorá-la para uma melhor usabilidade e maior suporte ao usuário.

3. A abordagem não oferece suporte a publicação dos artefatos para que possam ser utilizados em outras ferramentas.

4. Os tipos determinados para as informações, assim como as restrições implementadas no metamodelo foram feitas para casos gerais, conforme estudo apresentado no Capítulo 4. Entretanto, existem diversos pontos de extensão que permitem tal melhoria.

6.2 Trabalhos Futuros

Nesta dissertação foram respondidas algumas questões importantes relacionadas à autoria de Artefatos de Software. Apesar desta contribuição, a abordagem ainda precisa ser melhorada, principalmente no quesito avaliação, com o objetivo de ser consolidada. Apesar de avaliações iniciais terem demonstrado a aplicabilidade e corretude da abordagem, maiores investigações devem ser feitas quanto sua aplicabilidade na indústria. Desta forma, alguns trabalhos a serem realizados futuramente estão listados a seguir:

- **Melhorias a serem feitas na Abordagem:**
  - **Executar mais casos de testes:** possivelmente, após a execução de novos casos de testes mais completos novos dados serão coletados, surgindo a oportunidade de se fazer novas avaliações;
  - **Realizar um experimento:** a realização de um experimento que seja capaz de comparar nossa abordagem com o paradigma atual pode ser muito útil, pois coletando dados suficientes, pode-se tirar conclusões mais acertadas, baseando-se em análises de variáveis qualitativas e quantitativas.
- **Ferramenta de Autoria de Artefatos:**

  - **Análise de desempenho:** apesar dos testes realizados, é necessária a execução de testes reais e de grande porte utilizados na industria, a fim de verificar o desempenho e o comportamento da ferramenta nestes cenários. Possivelmente possa ocorrer queda de desempenho conforme o aumento da biblioteca em que os artefatos se encontram. Desse modo, uma avaliação destes pontos pode vir a ser importante e representa um relevante e provável futuro investimento. O resultado desta avaliação permitirá prover melhorias na ferramenta.

- **Guia de Autoria de Artefatos:**

  - **Aplicabilidade do guia de Autoria de Artefatos:** A aplicação do guia de composição definido neste trabalho em diferentes tipos de artefatos é indispensável para avaliação de sua aplicabilidade, consolidação e evolução.

Referências

Atkinson, C., & Kühne, T. (2003). Model-driven development: A metamodeling foundation. *IEEE Software*, *20*, 36–41.

Borsoi, B., & Becerra, J. (2008). Definition and modeling of process using object orientation. *SIGSOFT Software Engineering Notes*, *33*, 1–6.

Buchner, J. (2000). Hotdoc: a framework for compound documents. *ACM Computing Surveys (CSUR)*, *32*, Article No. 33.

Cattaneo, F., Nitto, E. D., Fuggetta, A., Lavazza, L., & Valetto, G. (2000). Managing software artifacts on the web with labyrinth. *ICSE '00: Proceedings of the 22nd International Conference on Software engineering* (pp. 746–749). New York, NY, USA: ACM.

Cugola, G., & Ghezzi, C. (1998). Software processes: a retrospective and a path to the future. *Software Process: Improvement and Practice*, *4*, 101–123.

Dijkstra, E. (1979). Go to statement considered harmful. *Classics in Software Engineering*, *11*, 27–33.

Duddy, K., Gerber, A., & Raymond, K. (2003). *Eclipse modelling framework (emf): import/export from mof / jmi* (Technical Report). Co-operative Centre for Enterprise Distributed Systems Technology - Pegamento Project.

Fuggetta, A. (2000). Software process: a roadmap. *ICSE '00: Proceedings of the Conference on The Future of Software Engineering* (pp. 25–34). New York, NY, USA: ACM Press.

Gamma, E., Helm, R., Johnson, R. E., & Vlissides, J. M. (1993). Design patterns: Abstraction and reuse of object-oriented design. *ECOOP '93: Proceedings of the 7th European Conference on Object-Oriented Programming* (pp. 406–431). London, UK: Springer-Verlag.

Graham, I., Henderson-Sellers, B., & Younessi, H. (1997). *The open process specification*. Open Series. New York, NY, USA: Addison Wesley Longman Limited. First edition.

Grolman, E., Fortuin, J., Akpotsui, E., Quint, V., & Roisin, C. (1997). Type modelling for document transformation in structured editing systems. *Mathematical and Computer Modelling*, *25*, 1 – 19.

Hartmann, J., Huang, S., & Tilley, S. (2001). Documenting software systems with views ii: an integrated approach based on xml. *SIGDOC '01: Proceedings of the 19th Annual International Conference on Computer Documentation* (pp. 237–246). New York, NY, USA: ACM Press.

Herzner, W., & Hocevar, E. (1991). Cdam–compound document access and management: an object-oriented approach. *SIGOIS Bulletin*, *12*, 1–18.

Krebs, J. (2007). Rup in the dialogue with scrum. Disponível em http://www.ibm.com/developerworks/rational/library/feb05/krebs. Acessado em 15 de janeiro de 2009.

Kroll, P., & Kruchten, P. (2003). *The rational unified process made easy: a practitioner's guide to the rup*. Addison-Wesley Object Technology Series. Boston, MA, USA: Addison Wesley Longman Limited. First edition.

Kruchten, P. (2000). *The rational unified process: An introduction*. Addison-Wesley Object Technology Series. Boston, MA, USA: Addison Wesley Longman Limited. Second edition.

Laitinen, K. (1992). Document classification for software quality systems. *SIGSOFT Software Engineering Notes*, *17*, 32–39.

Lee, S., Shim, J., & Wu, C. (2002). A metal model approach using uml for task assignment policy in software process. *APSEC '02: Proceedings of the Ninth Asia-Pacific Software Engineering Conference* (p. 376). Washington, DC, USA: IEEE Computer Society.

Moore, B., Dean, D., Gerber, A., Wagenknecht, G., & Vanderheyden, P. (2004). *Eclipse development using the graphical editing framework and the eclipse modeling framework*. Chicago IL, USA: IBM Redbooks. First edition.

Noll, J. (2007). Process enactment: A foundation for managing knowledge intensive work processes. Disponível em www.cse.scu.edu/ jnoll/noll-mkids.pdf. Acessado em 15 de janeiro de 2009.

OMG (2003a). Mda guide. Disponível em www.omg.org/docs/omg/03-06-01.pdf. Acessado em 15 de janeiro de 2009.

OMG (2003b). Unified modeling language specification v1.5. Disponível em http://www.omg.org/docs/formal/03-03-01.pdf. Acessado em 15 de janeiro de 2009.

OMG (2005). Software process engineering metamodel specification 1.1. Disponível em http://www.omg.org/docs/formal/05-01-06.pdf. Acessado em 15 de janeiro de 2009.

OMG (2006). Meta object facility v2.0. Disponível em http://www.omg.org/docs/formal/06-01-01.pdf. Acessado em 15 de janeiro de 2009.

OMG (2007a). Meta-object facility versioning and development lifecycle specification v2.0. Disponível em http://www.omg.org/docs/formal/07-05-01.pdf. Acessado em 15 de janeiro de 2009.

OMG (2007b). Unified modeling language (omg uml), infrastructure, v2.1.2. Disponível em http://www.omg.org/docs/formal/07-11-04.pdf. Acessado em 15 de janeiro de 2009.

OMG (2007c). Unified modeling language (omg uml), superstructure, v2.1.2. Disponível em http://www.omg.org/docs/formal/07-11-02.pdf. Acessado em 15 de janeiro de 2009.

OMG (2008a). Mof 2.0 facility and object lifecycle. Disponível em http://www.omg.org/docs/ptc/08-02-20.pdf. Acessado em 15 de janeiro de 2009.

OMG (2008b). Software & systems process engineering metamodel specification 2.0. Disponível em http://www.omg.org/cgi-bin/doc?formal/08-04-01.pdf. Acessado em 15 de janeiro de 2009.

Osterweil, L. (1987). Software processes are software too. *ICSE '87: Proceedings of the 9th International Conference on Software Engineering* (pp. 2–13). Los Alamitos, CA, USA: IEEE Computer Society Press.

Pereira, E., Bastos, R., & Oliveira, T. (2008). Process tailoring based on well-formedness rules. *SEKE '08: Proceedings of the 20th International Conference on Software Engineering and Knowledge Engineering*.

Pérez-Martínez, J. E. (2003). Heavyweight extensions to the uml metamodel to describe the c3 architectural style. *SIGSOFT Software Engineering Notes, 28*, 5–5.

PMI (2004). *A guide to the project management body of knowledge (pmbok guide)*. Newtown Square, PA, EUA.: PMI Publications. Third edition.

Quint, V., & Vatton, I. (2004). Techniques for authoring complex xml documents. *DocEng '04: Proceedings of the 2004 ACM Symposium on Document Engineering* (pp. 115–123). New York, NY, USA: ACM.

Rosener, V., & Avrilionis, D. (2006). Elements for the definition of a model of software engineering. *GaMMa '06: Proceedings of the 2006 International Workshop on Global integrated model management* (pp. 29–34). New York, NY, USA: ACM.

Rumbaugh, J., Jacobson, I., & Booch, G. (2004). *The unified modeling language reference manual*. Boston, MA, USA: Pearson Higher Education. Second edition.

Schwaber, K. (2004). *Agile project management with scrum*. Redmond, WA, USA: Microsoft Press. First edition.

Soares, M. (2004). Comparação entre metodologias Ágeis e tradicionais para o desenvolvimento de software. *INFOCOMP Journal of Computer Science, 3*, 8–13.

Sommerville, I. (2004). *Software engineering*. International Computer Science Series. Edinburg Gate, Harlow, England: Addison Wesley Publishers Limited. Seventh edition.

Steinberg, D., Budinsky, F., Paternostro, M., & Merks, E. (2008). *Emf: Eclipse modeling framework*. Eclipse Series. Boston, MA, USA: Addison-Wesley Professional. Second edition.

Tilley, S., & Müller, H. (1991). Info: a simple document annotation facility. *SIGDOC '91: Proceedings of the 9th Annual International Conference on Systems Documentation* (pp. 30–36). New York, NY, USA: ACM.

Visconti, M., & Cook, C. (1993). Software system documentation process maturity model. *CSC '93: Proceedings of the 1993 ACM Conference on Computer Science* (pp. 352–357). New York, NY, USA: ACM Press.

Warmer, J., & Kleppe, A. (2003). *The object constraint language: Getting your models ready for mda*. Object Technology Series. Boston, MA, USA: Addison-Wesley Professional. Second edition.

Williams, A. (2004). The documentation of quality engineering: applying use cases to drive change in software engineering models. *SIGDOC '04: Proceedings of the 22nd Annual International Conference on Design of Communication* (pp. 4–13). New York, NY, USA: ACM.

Apêndice A Metamodelo extendido do SPEM v2

A seguir é apresentada a extensão do SPEM v2. Para melhor entendimento e leitura foi utilizado o mesmo modelo e as mesmas nomenclaturas existentes na especificação da UML (OMG, 2007b).

A.1 Escopo

Esta extensão do metamodelo do SPEM v2 adiciona elementos que complementam a especificação de processos para permitir autoria de artefatos detalhadamente. Na atual versão estão definidos construtores fundamentais que complementam os já existentes na UML e no SPEM v2. Contudo, este metamodelo foi construído com o intuíto de possuir infra-estrutura de metamodelagem, sendo a base para o crescimento do mesmo, permitindo a construção posterior de uma estrutura com novos estereótipos e diagramas adicionais.

A.2 Conformance

Assim como a UML, este metamodelo foi construído em vários módulos que encapsulam as diversas abstrações existentes, permitindo o particionamento em *language units*. Conforme já foi experimentado na UML, este tipo de flexibilidade aumenta o reuso estrutural e permite a escolha do que se deseja utilizar para cada domínio a ser implementado.

Para uma melhor definição e uso, foram definidos três pontos de conformidade (*Compliance Points*). Não obstante do SPEM v2 ou da própria UML, a implementação do metamodelo de acordo com algum destes pontos de conformidade é fortemente recomendada. Entretanto, caso necessário, pose-se utilizar qualquer combinação entre os pacotes que se desejar implementar.

A extensão em questão está definida tanto em metamodelo quanto em *UML Profile*. No caso de se utilizar *UML Profile*, os mesmos pontos de conformidades estão disponíveis.

A.2.1 Princípios do Projeto e Empacotamento

Este metamodelo é baseado no MOF, que reusa o pacote *UML Infrastructure library*. Alguns conceitos e estruturas existentes, assim como *Classifier* e *Package*, são vistos diretamente nesse pacote. Outros, neste caso os principais, foram reusados a partir do próprio metamodelo do SPEM v2. Por este motivo, existe a mesma divisão entre *Method Content*, *Process Structure* e *Process Behavior*, definidos na especificação do SPEM v2.

Cada pacote existente agrupa um conjunto de abstrações necessárias para se obter um conceito que fora definido no metamodelo. Dentro do pacote principal *XtSPEM* estão todos os elementos existentes que, juntamente com a aplicação do mecanismo de *package merge* em UML 2.0 *Infrastructure Library* e SPEM v2, gradualmente constroem o metamodelo.

A.2.2 Arquitetura

Embora o SPEM v 2 seja utilizado para definir Processos de Desenvolvimento, o propósito da extensão é reduzido ao detalhamento dos artefatos do processo em questão. O escopo do metamodelo foi propositalmente limitado apenas a autoria de artefatos. Contudo o foco é permitir definição e uso de artefatos particionando-os em pedaços menores, utilizando estruturas de informação genéricas e definindo pontos de extensão para a criação de construtores para cada domínio específico. Portanto, estão fora do escopo, quaisquer elementos, estruturas ou conceitos que não façam parte deste propósito, assim como ordem de atividades ou uso dos artefatos.

Este metamodelo esta estruturado em 6 (**seis**) pacotes principais que dividem o modelo em unidades lógicas e podem ser vistos na Figura A.1. Cada uma destas unidades lógicas provê estruturas e conceitos adicionais. O mecanismo de *package merge* da UML 2 foi aplicado em cada pacote gradualmente extendendo tanto a UML 2 quanto o SPEM v2. Desta forma, unidades definidas nas camadas mais baixas podem ser compreendidas como um subconjunto do metamodelo, contendo metaclasses em suas formas mais simples. Tais metaclasses vão sendo estendidas nas camadas mais altas via *package merge*, sendo acrescentados novos atributos e relacionamentos para se obter unidades mais complexas.

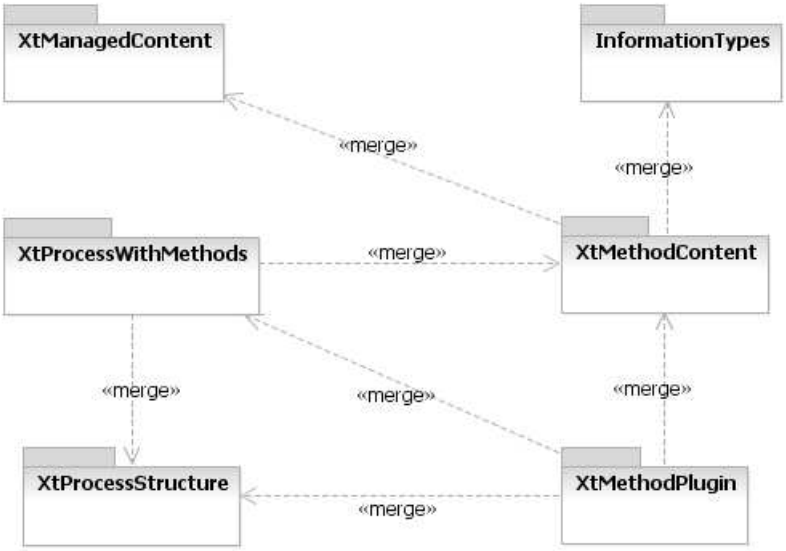

Figura A.1: Estrutura de pacotes do metamodelo de extensão do SPEM v2

Os pacotes da Figure A.1 provêm:

- **InformationTypes**, pacote responsável pelos tipos de informações existentes. Cada tipo de informação que será armazenado pelos Artefatos de Software e que serão utilizados pelo Processo de Desenvolvimento de Software estão metamodelados dentro dos respectivos pacotes internos ao `InformationTypes`.
- **XtManagedContent**, pacote que estende o pacote original `ManagedContent`. Este pacote que acrescenta os conceitos de classificação e maturidade. Este pacote é responsável por permitir gerencialmento de conteúdo dos artefatos.
- **XtMethodContent**, pacote que estende o pacote original `MethodContent`. Este pacote é responsável pela extensão dos conceitos de **definição** de produtos de trabalho (*Work*

Products) para se conseguir um Artefato de Software particionado em fragmentos que possam ser armazenados e organizados em conteineres. Além disso, foi metamodelada uma estrutura que permite o relacionamento entre atores, atividades, artefatos, conteineres e fragmentos.

- **XtProcessStructure**, pacote que estende o pacote original `ProcessStructure`. Este pacote é resposável pela extensão dos conceitos de **uso** de produtos de trabalho assim como os relacionamentos entre o uso de papéis, atividades, artefatos, conteineres e fragmentos.

- **XtProcessWithMethods**, pacote que estende o pacote original do SPEM v2 `ProcessWithMethods`. Este pacote é responsável por relacionar os elementos extendidos no pacote `XtMethodContent` com os elementos extendidos do pacote `XtProcessStructure`, caracterizando o conceito de uso de elementos de biblioteca para criação de um Processo de Desenvolvimento de Software.

- **XtMethodPlugin**, pacote que estende o pacote original `MethodPlugin` para fazer *merge* com os pacotes extendidos que suportam esta abordagem. Nenhuma metaclasse ou novo construtor foi adicionado a este pacote.

Dada a Figura A.1 que apresenta os pacotes que consistem no metamodelo extendido do SPEM v2, a Figura A.2 apresenta a relação desses pacotes com os pacotes originais.

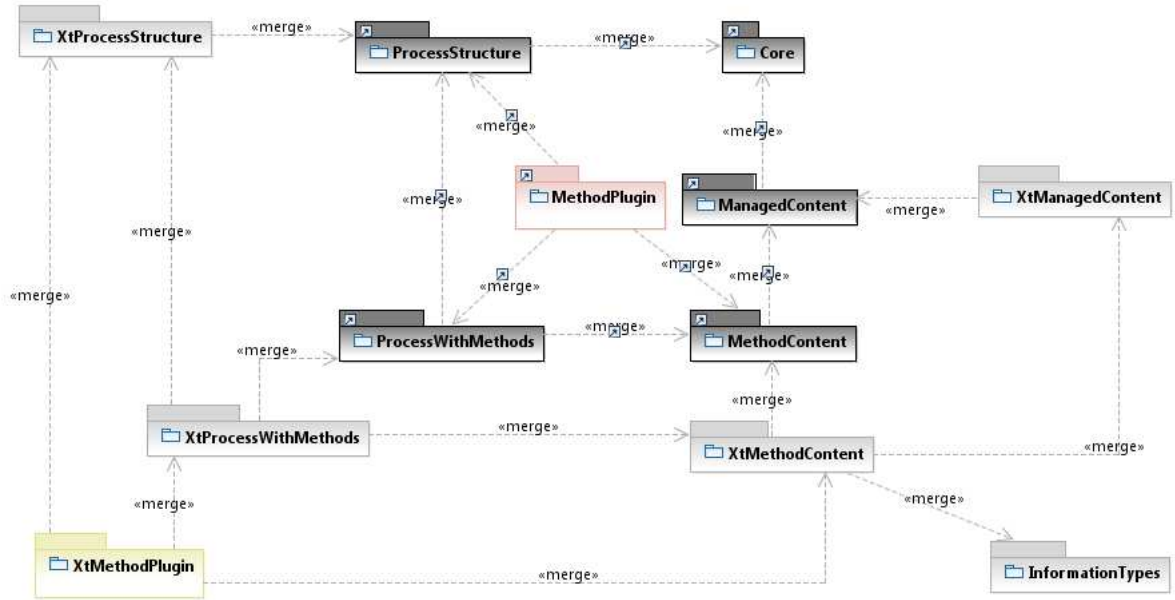

Figura A.2: Estrutura de pacotes com a extensão do SPEM v2

A.2.3 Nível de Conformidade *Extended MethodContent* (*XtMethodContent*)

O ponto de conformidade "*XtMethodContent*" consiste na utilização e implementação dos pacotes referentes a construção do *Method Content* juntamente com a extensão proposta no metamodelo SPEMXt. Este ponto de conformidade cria a *namespace SPEMxt Method Content*,

que na verdade é um *package merge* entre o ponto de conformidade LM, da *UML 2 Infrastructure Library* e o pacote *XtMethod Content* do SPEMXt, vistos na Figura C.4.. O pacote *XtMethod Content* inclui os pacotes *XtMethod Content* e *XtManaged Content*, do SPEMXt, além disso, também está incluído o pacote *Core*, do SPEM v2, através de transitividade transitividade, como pode ser visto na Figura A.2.

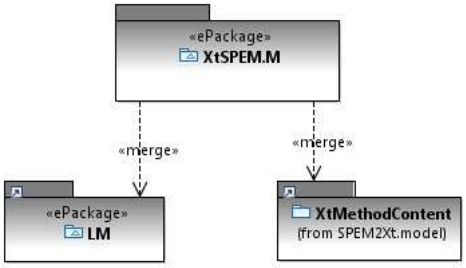

Figura A.3: Package Merge do Ponto de conformidade *XtMethodContent*

A.2.4 Nível de Conformidade *Extended Complete* (*XtComplete*)

Público: *Tool Vendors* e fornecedores de bibliotecas de processos.

O ponto de conformidade "*XtComplete*" consiste na utilização e implementação de todos os pacotes do metamodelo descrito em A.2.2 e visto na Figura A.1. O ponto de conformidade "*XtComplete*" cria um *namespace* chamada *SPEMxt All* que pode ser vista na Figura A.4. Como pode ser visto, este *namespace* é na verdade um *package merge* entre o ponto de conformidade LM, da *UML 2 Infrastructure Library*, o pacote *UML 2 Profiles*, o pacote *Process Behavior* do SPEM v2 e o *XtMethod Plugin* do SPEMXt. O pacote *XtMethod Plugin* inclui todos os outros pacotes existentes no SPEMXt por transitividade, como pode ser visto na Figura A.2

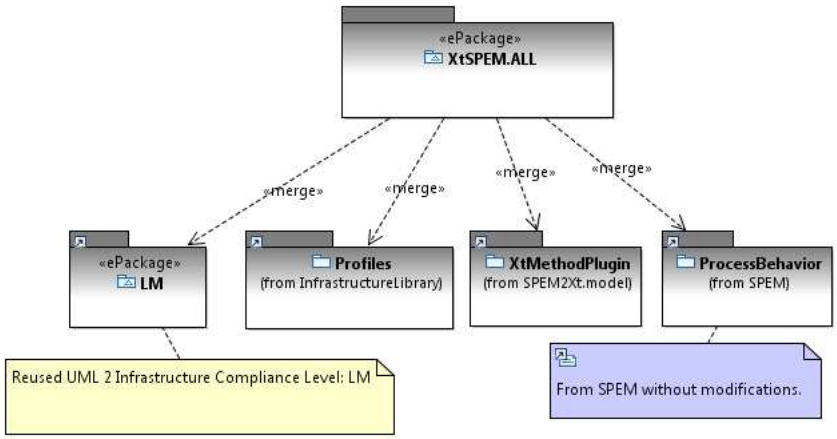

Figura A.4: Package Merge do Ponto de conformidade *XtComplete*

A.3 InformationTypes

O pacote `InformationTypes` é responsável pelos tipos de informação existentes em um Processo de Desenvolvimento de Software. Estes tipos serão utilizados pela definição dos Artefatos de Software e mais adiante, na fase de Projeto, serão acrescentados os dados necessários para definir o preenchimento desses tipos.

Como apresentado na Figura A.5, exitem quatro pacotes que armazenam diferentes tipificações: `SimpleTypes`, `ComplexTypes`, `DiagramTypes` e `SpecificTypes`. Cada pacote provê diferentes tipos de informação, estruturados conforme o nível de generalização.

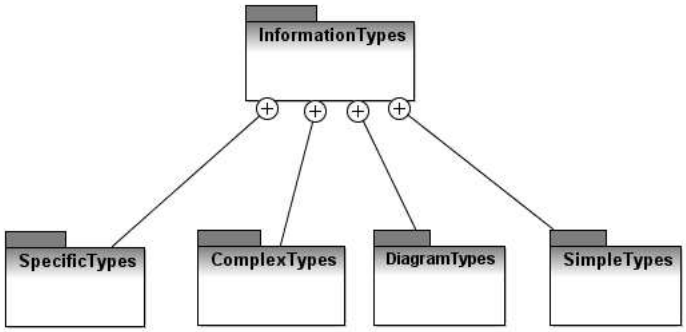

Figura A.5: Estrutura de pacotes para empacotamento dos tipos de informação.

A idéia principal deste empacotamente é dividir em diferentes camadas ou níveis os diferentes tipos de informação que podem ser utilizados durante a modelagem do PDS. Desta forma, o diagrama da Figura A.6 apresenta a modelagem hierárquica que viabiliza o uso destes níveis de informação com base no tipo necessário a ser armazenado nos Artefatos de Software em suas diferentes fases de desenvolvimento.

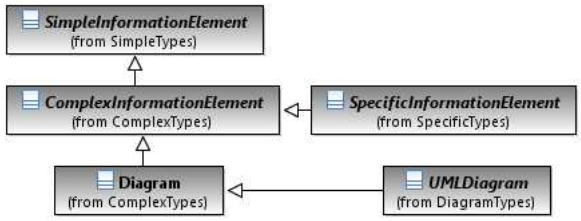

Figura A.6: Estrutura hierarquica das metaclasses utilizadas para tipificação da informação.

A hierarquia de classes que especializam a metaclasse `SimpleInformationElement` foi modelada para garantir que os diversos tipos de informações sejam sobrepostos, permitindo a utilização de estrutura de acordo com o nível requerido. Ou seja, caso fosse necessário ambos os tipos complexos e simples, bastaria utilizá-los como se fossem o mesmo tipo, aumentando a flexibilidade. Isto ocorre com com qualquer um dos elementos empacotados, bastando considerar a ordem da hierarquia, neste caso, da subclasse para a superclasse.

A.3.1 SimpleInformationElement

definido na Seção A.3.4

A.3.2 ComplexInformationElement

definido na Seção A.3.5

A.3.3 SpecificInformationElement

definido na Seção A.3.7

A.3.4 SimpleTypes

O pacote `SimpleTypes` se constitui apenas dos tipos mais simples de informação. Embora um pouco mais complexos que os tipos primitivos do pacote basico da própria UML, os tipos introduzidos neste pacote não apresentam uma nenhuma dificuldade no que se entende por modelagem. Todavia, seu uso é bastante comum em Artefatos de Software.

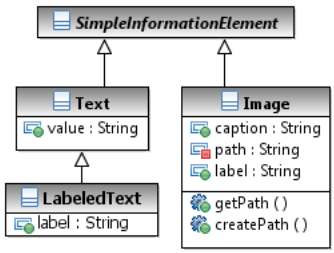

Figura A.7: Metaclasses para tipificação da informação do pacote `SimpleTypes`.

Image

Descrição: A metaclasse `Image` estrutura o tipo de informação Imagem.
Generalização:

- `Image` estende `SimpleInformationElement` definindo o conceito de imagem a um tipo simples de informação no metamodelo.

Atributos:

- **caption**: *String*, define o subtítulo da imagem a ser exposto durante sua apresentação. Este atributo possui uma restrição em relação ao seu tamanho, podendo ser no máximo de 30 caracteres. Isto é expresso em OCL da seguinte forma:
  ```
  Image
  self.caption.size() <= 30
  ```
- **path**: *String*, apresenta o caminho fisico da imagem;
- **label**: *String*, determina um marcador para a imagem, uma espécie de etiqueta para associação posterior. Este atributo possui uma restrição em relação ao seu tamanho, podendo ser no máximo de 20 caracteres. Isto é expressado em OCL da seguinte forma:
  ```
  Image
  self.caption.size() <= 20
  ```

Operações Adicionais:

- A operação `createPath()` deve gravar o caminho físico da imagem no atributo `path` e retornar um valor booleano, determinando o sucesso da criação com `true`. O insucesso da criação pode ser devido a existência de um caminho anterior. Isto está com uma pré-condição expressada em OCL como segue:
  ```
  Image::createPath()
  self.path->isEmpty()
  ```

- A operação `getPath()` retorna o caminho físico da imagem através do atributo `path`, criado pela operação `createPath()`.

Semântica: Artefatos de Software possuem imagens para representar algum conceito previamente definido ou para expor algo especificado de forma mais clara, uma vez que não utiliza apenas palavras. O conceito de imagem foi modelado através da metaclasse `Image`, que define também subtítulo, etiqueta e um caminho físico.

SimpleInformationElement

Descrição: `Simple Information Element` é uma metaclasse abstrata para generalizar qualquer elemento que modele o conceito de Tipo de Informação e que utilize apenas os tipos primitivos definidos através de *UML DataType* (OMG, 2007b).

Semântica: Independente do Processo de Software escolhido, algumas estruturas de informação são genéricamente utilzados. Conforme existem diversos tipos de informações, o `SimpleInformationElement` determina os tipos mais simples que possam existir, desde que sejam independentes de processo e utilizem apenas tipos de dados definidos previamente pela *UML DataType* (i.e. *Boolean*, *Integer*, *String* e *UnlimitedNatural*).

LabeledText

Descrição: A metaclasse `LabeledText` estrutura o tipo de informação Texto Rotulado, desde que seja necessária a utilização de algum rótulo para identificar um texto.

Generalização:

- `LabeledText` estende `Text` definindo o conceito de texto rotulado como um tipo simples de informação no metamodelo.

Atributos:

- **label**: *String*, conteúdo do rótulo identificador a ser preenchido. Este atributo possui uma restrição em relação ao seu tamanho, podendo ser no máximo de 15 caracteres. Isto é expressado em OCL da seguinte forma:
  ```
  LabeledText
  self.label.size() <= 15
  ```

Semântica: Artefatos de Software possuem textos para representar o conhecimento necessários através de palavras. Alguns desses textos precisam ser rotulados para uma melhor identificação, assim como "*name:* Marcos", por exemplo. O conceito de texto rotulado foi modelado através da metaclasse `LabeledText`, que define um atributo para armazenar o rótulo a ser preenchido.

Text

Descrição: A metaclasse `Text` estrutura o tipo de informação Texto.
Generalização:

- `Text` estende `SimpleInformationElement` definindo o conceito de texto a um tipo simples de informação no metamodelo.

Atributos:

- **value**: *String*, conteúdo do texto a ser preenchido.

Semântica: Artefatos de Software possuem textos para representar o conhecimento necessários através de palavras. O conceito de texto foi modelado simplesmente através da metaclasse `Text`, que define um atributo para armazenar o conteúdo a ser preenchido.

A.3.5 ComplexTypes

O pacote `ComplexTypes` se constitui dos tipos que utilizam os tipos simples de informação. São tipos mais complexos e são geralmente formados a partir das mateclasses `Text` e `Imagem`. Tais tipos são estruturados para que seja possível criar outros tipos de informação além dos supracitados. Dado o fato de serem tipos que precisam utilizar estruturalmente os tipos simples previamente modelados, não podem entrar na mesma categoria, por isso são separados em um pacote diferente.

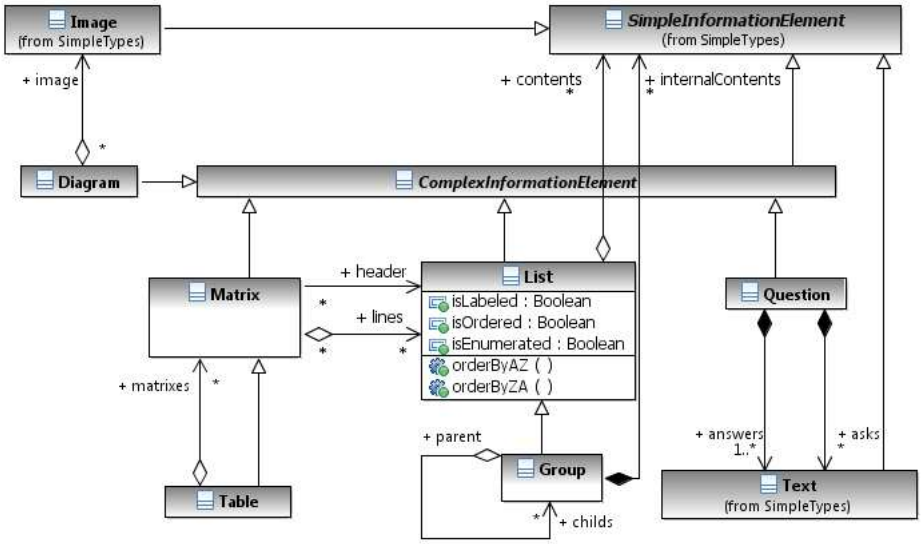

Figura A.8: Metaclasses para tipificação da informação do pacote `ComplexTypes`.

ComplexInformationElement

Descrição: `ComplexInformationElement` é uma metaclasse abstrata para generalizar qualquer elemento que modele o conceito de Tipo de Informação e que utilize os tipos primitivos definidos através de *UML DataType* e também os tipos simples modelads através da metaclasse `SimpleInformation Element`.
Generalização:

- `ComplexInformationElement` estende `SimpleInformationElement` especializando o conceito de tipo simples de informação para tipos complexos no metamodelo.

Semântica: Conforme a necessidade de divisão entre os diversos tipos de estruturas de informação existentes, a metaclasse `ComplexInformationElement` serve como superclasse genérica dos tipos complexos. Esta metaclasse torna os conceitos definidos em A.3.4 mais especializados, permitindo que haja relacionamento entre os tipos simples, formando assim estruturas complexas de informação.

Diagram

Descrição: A metaclasse `Diagram` estrutura o tipo de informação Diagrama.
Generalização:

- `Diagram` estende `ComplexInformationElement`, definindo o conceito de diagrama como um tipo complexo de informação no metamodelo.

Associações:

- **image:** *Image*, esta associação representa a ligação entre um diagrama e sua imagem.

Semântica: Artefatos de Software possuem formas de representação de conteúdo através de diagramas. Estes diagramas servem para representar o conhecimento necessários através de imagens, muitas vezes exemplificado algo que foi descrito anteriormente. O conceito de diagrama foi modelado através da metaclasse `Diagram`, que define um relacionamento para armazenar a imagem que ele representa.

Group

Descrição: A metaclasse `Group` organiza outros tipos de informação existente, agindo como um agrupador.
Generalização:

- `Group` estende `List`, definindo o conceito de agrupador como um tipo especial de lista, sendo um tipo complexo de informação no metamodelo.

Associações:

- **childs:** *Group*, esta associação define a existência de um conjunto sub agrupadores de um agrupador a ser utilizado. Isto permite a criação de agrupadores em árvore, ou seja, permite a criação de estruturas de informação agrupadas de forma hierárquica. Desta forma, cada agrupador, ou grupo de informações, pode sempre ser aninhado por outro grupo. Esta associação possui uma restrição em relação aos grupos possíveis, não permitindo que um grupo seja filho dele mesmo. Além disso, este atributo não pode agrupar a mesma informação mais de uma vez.

- **internalContents:** `SimpleInformationElement`, conforme a necessidade de agrupar outros tipos de informação, essa associação permite criar estruturas mais complexas, agrupando-as.

- **parent:** *Group*, assim como foi definida a existência de uma hierarquia de grupos, ou agrupadores, a partir do atributo `childs`, esta associação permite a identificação de um pai para um conjunto de sub agrupadores. Desta forma, os grupos inseridos no conjunto definido pela associação `childs` conseguem identificar a qual grupo estão agrupados. Esta associação possui uma restrição em relação a um pai possíveis, não permitindo que um grupo seja pai dele própaio.

Semântica: Algumas informações relevantes são agrupadas através de agrupadores de informações para que o Artefato de Software esteja melhor organizado. Essas informações são, em sua maioria, informações relacionadas, que juntas podem representam alguma outra informação ainda mais importante, de forma *ad hoc*. Além disso, no contexto de organização da informação, muitas vezes a utilização de agrupamentos facilita a visualização ou até mesmo o acesso a informação pretendida. O conceito de agrupamento foi modelado através da metaclasse `Group`, que possui relacionamentos para armazenar informações tanto internamente, em se tratando de composição de informações, quanto externamente, no tocante a agrupamento de diferentes informações relacionadas.

Image

definido na Seção A.3.4

List

Descrição: A metaclasse `List` estrutura o tipo de informação Lista, reponsável por agrupar os elementos sequencialmente.

Generalização:

- `List` estende `ComplexInformationElement`, definindo o conceito de lista como um tipo complexo de informação no metamodelo.

Atributos:

- **isLabeled**: *Boolean*, define se os elementos da lista possuirão rótulos adicionais;
- **isOrderer**: *Boolean*, define se os elementos da lista serão ordenados;
- **isEnumerated**: *Boolean*, define se os elementos da lista possuirão numeração identificadora com início em 1;

Associações:

- **contents:** *SimpleInformationElement*, conforme a necessidade de agrupar outros tipos de informação, essa associação permite inserir tipos de informação na lista.

Operações:

- orderByAZ(), ordena todos os elementos do conteúdo da lista, do menor para o maior elemento. Caso sejam do tipo *Integer*, será respeitada a ordem numérica. Caso sejam *String*, será respeitada a ordem alfabética.
- orderByZA(), ordena todos os elementos do conteúdo da lista, do maior para o menor elemento. Caso sejam do tipo *Integer*, será respeitada a ordem numérica invertida. Caso sejam *String*, será respeitada a ordem alfabética invertida.

Semântica: Algumas informações são organizadas em forma de listas. Num contexto geral, listas são bastante utilizadas, pode-se perceber seu uso em: aplicações bancárias, onde são utilizadas para expressar itens de forma organizada, geralmente com base em alguma ordem; e na própria lingua portuguesa, que utiliza listas para permitir enumerações e itenizações de descrições ou elementos existentes, geralmente destacados através de rótulos e enumerações. Nesse sentido, a metaclasse `List` é responsável por permitir a criação de conjuntos de tipos de informação para uma melhor organização desse tipos, atuando como um agrupador sequencial.

Matrix

Descrição: A metaclasse `Matrix` estrutura o tipo de informação com base em matrizes e tabulação.

Generalização:

- `Matrix` estende `ComplexInformationElement`, definindo o conceito de matrizes como um tipo complexo de informação no metamodelo.

Associações:

- **header:** *List*, utilizado para criar o cabeçalho da matriz caso necessário;
- **lines:** *List*, utilizado para incluir as linhas da matriz ou tabulação.

Semântica: Algumas informações são organizadas em forma de matrizes, conforme necessidade de organizá-las em formatos tabulares. Num contexto mais específico podemos citar seu uso em situações como por exemplo do RUP, que define algumas das suas informações em forma de tabela, tais como: perfil do *stakeholder*, levantamento de riscos e problemas a serem enfrentados. De um modo geral, podemos também inserir neste mesmo contexto, documentos feitos através do *Microsoft Excel* ™, o qual se caracteriza por ter uma planilha que se constitui em um matrizes. Diante da necessidade de possuir este tipo de informação, em nosso metamodelo, este conceito é definido pela metaclasse `Matrix`.

Question

Descrição: A metaclasse `Question` estrutura o tipo de informação “pergunta x respostas”, ou seja, define questões.

Generalização:

- `Question` estende `ComplexInformationElement`, definindo o conceito questão como um tipo complexo de informação no metamodelo.

Associações:

- **asks:** *Text*, utilizado para criar as perguntas a serem feitas;
- **answers:** *Text*, utilizado para incluir as respostas recebidas.

Semântica: Informações primodiais para a construção de aplicações são geralmente adquiridas no momento em que se entrevista um *stakeholder* ou se questiona um cliente específico. Tais informações são encontradas a medida que se obtem as respostas a alguma pergunta feita. De um modo geral, é possível observar que um fato bastante comum é documentar perguntas importantes, assim como dúvidas existentes durante o desenvolvimento, assim como as respostas obtidas. Diante da necessidade de possuir este tipo de informação, em nosso metamodelo, este conceito é definido pela metaclasse `Question`.

SimpleInformationElement

definido na Seção A.3.4

Table

Descrição: A metaclasse `Table` estrutura o tipo de informação Tabela.
Generalização:

- `Table` estende `Matrix`, definindo o conceito de tabela como um tipo de Matrix, sendo um tipo complexo no metamodelo.

Associações:

- **matrixes:** *Matrix*, define o conjunto de matrizes existentes em uma tabela;

Semântica: Assim como existem informações que são organizadas em matrizes, num contexto mais específico existem também informações organizadas de tal forma, que se parecem com um conjunto de matrizes. Documentos feitos através do *Microsoft Excel* ™, o qual se caracteriza por ter uma planilha que se constitui em um matrizes, desta forma, nota-se que um conjunto de matrizes pode permitir a formação de uma tabela. Diante da necessidade de possuir este tipo de informação, em nosso metamodelo, este conceito é definido pela metaclasse `Tabela`.

Text

definido na Seção A.3.4

A.3.6 DiagramTypes

O pacote `DiagramTypes` é baseado nos tipos de informações que utilizam-se de diagramas como representação de conhecimento. Neste trabalho diagramas são estruturados como tipos complexos e são genericamente definidos pela metaclasse *Diagram*. Além dos diagramas comuns também existem os diagramas formados a partir das mateclasses definidas pela *UML Superstructure* (OMG, 2007c), considerados diagramas UML.

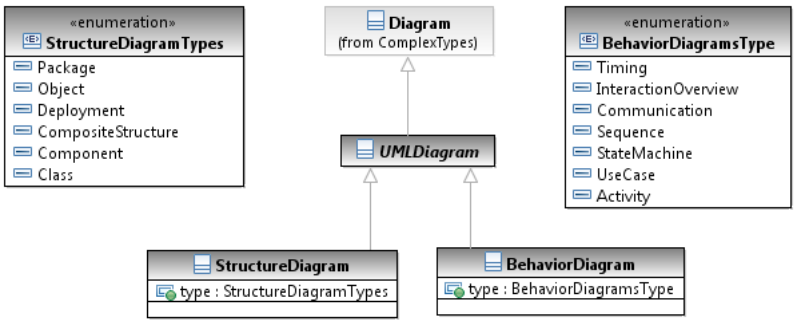

Figura A.9: Metaclasses para tipificação da informação do pacote `DiagramTypes`.

BehaviorDiagram

Descrição: A metaclasse `BehaviorDiagram` estrutura o tipo de informação Diagrama, no contexto de Diagramas Comportamentais da UML.

Generalização:

- `BehaviorDiagram` estende `UMLDiagram`, especializando os tipo possíveis de diagramas UML. Desta forma, esta metaclasse é definida para utilizar os diagramas Comportamentais.

Atributos:

- **type**: *BehaviorDiagramsType*, define qual o tipo de diagrama comportamental da UML representado;

Semântica: Durante o desenvolvimento de aplicações, vários tipos de tipos de diagramas são construídos para representar as diferentes visões do software. Algumas informações relacionadas aos estados e a estrutura de softwares orientados a objetos são representadas através de diagramas específicos, geralmente em linguagem UML. Tais diagramas são divididos em dois grandes grupos, Comportamentais e Estruturais, de acordo com a especificação da própria UML. Diante da necessidade de possuir este tipo de informação, em nosso metamodelo, utilizamos o conceito de Diagrama Comportamental na metaclasse `BehaviorDiagram`.

BehaviorDiagramTypes

Descrição: Esta enumeração define quais diagramas podem ser utilizados em uma instância de `BehaviorDiagram`.

Generalização:

- n/a: *Enumeration*

Literais:

- **Timing**: define que uma instância de `BehaviorDiagram` será utilizada como um Diagrama de Tempo;
- **IteractionOverview**: define que uma instância de `BehaviorDiagram` será utilizada como um Diagrama de Interatividade;
- **Communication**: define que uma instância de `BehaviorDiagram` será utilizada como um Diagrama de Colaboração (ou Comunicação);
- **Sequence**: define que uma instância de `BehaviorDiagram` será utilizada como um Diagrama de Sequência;
- **StateMachine**: define que uma instância de `BehaviorDiagram` será utilizada como um Diagrama de Transição de Estados;
- **UseCase**: define que uma instância de `BehaviorDiagram` será utilizada como um Diagrama de Caso de Uso;
- **Activity**: define que uma instância de `BehaviorDiagram` será utilizada como um Diagrama de Atividade;

Semântica: Partindo do princípio de que algumas informações são representadas através de diagramas específicos em acordo com a especificação da UML. É necessário especificar extamente quais são os tipos existentes de diagramas. Existem dois grandes grupos que dividem tais diagramas: o Grupo de Diagramas Comportamentais e o grupo dos Diagramas Estruturais. Todos os tipos dos diagramas Comportamentais estão modelados como literais da enumeração `BehaviorDiagramTypes`.

Diagram

definido na Seção A.3.5

StructureDiagram

Descrição: A metaclasse `StructureDiagram` estrutura o tipo de informação Diagrama, no contexto de Diagramas Estruturais da UML.

Generalização:

- `StructureDiagram` estende `UMLDiagram`, especializando os tipo possíveis de diagramas UML. Desta forma, esta metaclasse é definida para utilizar os diagramas Estruturais.

Atributos:

- **type**: *StructureDiagramsType*, define qual o tipo de diagrama estrutural da UML representado;

Semântica: Durante o desenvolvimento de aplicações, vários tipos de tipos de diagramas são construídos para representar as diferentes visões do software. Algumas informações relacionadas aos estados e a estrutura de softwares orientados a objetos são representadas através de diagramas específicos, geralmente em linguagem UML. Tais diagramas são divididos em dois grandes grupos, Comportamentais e Estruturais, de acordo com a especificação da própria UML. Diante da necessidade de possuir este tipo de informação, em nosso metamodelo, utilizamos o conceito de Diagrama Estrutural na metaclasse `BehaviorDiagram`.

StructureDiagramTypes

Descrição: Esta enumeração define quais diagramas podem ser utilizados em uma instância de `StructureDiagram`.

Generalização:

- n/a: *Enumeration*

Literais:

- **Package**: define que uma instância de `StructureDiagram` será utilizada como um Diagrama de Pacotes;
- **Object**: define que uma instância de `StructureDiagram` será utilizada como um Diagrama de Objetos;
- **Deployment**: define que uma instância de `StructureDiagram` será utilizada como um Diagrama de Instalação;

- **CompositeStructure**: define que uma instância de `StructureDiagram` será utilizada como um Diagrama de Estrutura;

- **Component**: define que uma instância de `StructureDiagram` será utilizada como um Diagrama de Componentes;

- **Class**: define que uma instância de `StructureDiagram` será utilizada como um Diagrama de Classes;

Semântica: Partindo do princípio de que algumas informações são representadas através de diagramas específicos em acordo com a especificação da UML. É necessário especificar extamente quais são os tipos existentes de diagramas. Existem dois grandes grupos que dividem tais diagramas: o Grupo de Diagramas Comportamentais e o grupo dos Diagramas Estruturais. Todos os tipos dos diagramas Estruturados estão modelados como literais da enumeração `StructureDiagramTypes`.

UMLDiagram

Descrição: A metaclasse `UMLDiagram` estrutura o tipo de informação Diagrama no contexto de Diagramas da UML.

Generalização:

- `UMLDiagram` estende `Diagram`, especializand-a para permitir a modelagem dos tipo possíveis de diagramas da UML.

Semântica: Durante o desenvolvimento de aplicações, vários tipos de tipos de diagramas são construídos para representar as diferentes visões do software. Algumas informações relacionadas aos estados e a estrutura de softwares orientados a objetos são representadas através de diagramas específicos, geralmente em linguagem UML. Nesse sentido, a metaclasse `UMLDiagram` é responsável por permitir a criação deste tipos de informação.

A.3.7 SpecificTypes

O pacote `SpecificTypes` foi modelado para que os tipos de informações específicos de cada Processo de Software possam ser modelados, estendendo assim, esta proposta. Devido ao fato de existirem diferentes tipos de informações dependentes de PDS, não foi interessante modelar todas estas estruturas, deixando o metamodelo genérico o suficiente para suportar diferentes PDSs.

Desta forma, caso seja necessário instanciar algum PDS específico, assim como o RUP, faz-se necessário estender esta proposta, modelando as estruturas de informação a serem utilizadas e reimplementando o metamodelo de acordo com o nível de conformidade desejado. Dado exemplo, no caso do RUP, poderiam ser modeladas estruturas como Requisito, *Stakeholder* e Descrição de Caso de Uso.

ComplexInformationElement

definido na Seção A.3.5

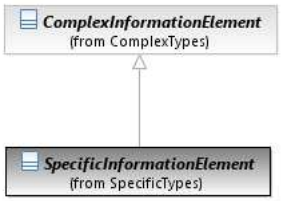

Figura A.10: Metaclasses para tipificação da informação do pacote `SpecificTypes`.

SpecificInformationElement

Descrição: *SpecificInformationElement* é uma metaclasse abstrata para generalizar qualquer elemento que modele o conceito de Tipo de Informação para processos específicos.

Generalização:

- `SpecificInformationElement` estende `SimpleInformationElement` especializando o conceito de tipo genérico e independente de PDS para os tipos específicos e dependentes.

Semântica: Conforme Processo de Software escolhido, algumas estruturas de informação específicas devem ser utilizadas. Tais tipos de informações, assim como Stakeholder, Cliente, Riscos, podem ser definidos como tiops totalmente específicos de processo, uma vez que nem todos os processos geram essa informação. A metaclasse `SpecificInformationElement` funciona como um separador de águas, determinando um ponto de extensão que permite os tipos mais específicos de PDSs serem modelados.

A.4 XtManagedContent

O pacote `XtManagedContent` se constitui dos elementos que comportam as informações sobre as diversas mais diversas descrições que um artefato pode possuir. Tais elementos são chamados de descritores. Desta forma, neste pacote estarão as metaclasses responsáveis por permitir inserção de uma visão geral, descrições, propósito, finalidade e objetivo, além de permitir utilizar recursos como classificação, categorização e controle de nível de maturidade. Na Figura A.11 é apresentado o diagrama com as metaclasses desse pacote.

A.4.1 ClassElement

Descrição: *ClassElement* é uma metaclasse que serve para determinar o tipo de classificação de um elemento classificável, utilizando as classificações possíveis.

Generalização:

- `ClassElement` estende `DescribableElemente` do SPEM v2, sendo então um tipo de elemento descritivo.

Atributos:

- **classKind**: *ClassElementKind*, define uma ligação com uma enumeração de tipos que representa as classificações possíveis.

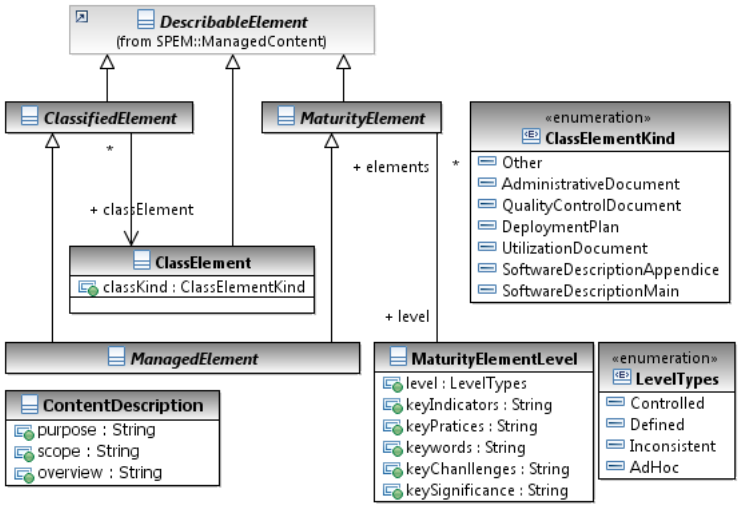

Figura A.11: Metaclasses dos conceitos de Classificação e Níveis de Maturidade

Semântica: Conforme visto em (Laitinen, 1992), cada documento de um processo possui uma classificação, sendo uma tentativa para tentar controlar e gerenciar artefatos uniformemente. Conforme o autor sugere, existe uma nomenclatura, que segue: *Software Description*, *Utilization Documents*, *Development Plans*, *Quality Control Documents* and *Administrative Documents*. A metaclasse *ClassElement* determina qual dessas possíveis classificações será aplicada a um artefato específico.

A.4.2 ClassElementKind

Descrição: Esta enumeração define quais classificações podem ser utilizados em uma instância de `ClassElement`.

Generalização:

- n/a: *Enumeration*

Literais:

- **Other**: sem classificação determinada;
- **AdministrativeDocument**: define uma classificação para documentos administrativo;
- **QualityControlDocument**: define uma classificação para documentos que lidam com controle de qualidade;
- **DeploymentPlan**: define uma classificação para documentos sobre a implantação de um sistema em desenvolvimento;
- **UtilizationDocument**: define uma classificação para documentos que lidam com material de utilização de um sistema em desenvolvimento;
- **SoftwareDescriptionAppendice**: define uma classificação para documentos que descrevem um sistema em desenvolvimento;

- **SoftwareDescriptionMain**: define uma classificação para documentos que escrevem as principais características de um sistema em desenvolvimento;

Semântica: Segundo (Laitinen, 1992), autor da abordagem de classificação, documentos de software devem ser classificados para melhor organizá-los. Partindo desse princípio, ele especifica seis áreas de atuação desses documentos, todas em relação a fases do desenvolvimento de um sistema. Em nossa abordagem, todos as classificações possíveis para os documentos estão modeladas através de literais da enumeração `ClassElementKinds`.

A.4.3 ClassifiedElement

Descrição: `ClassElement` é uma metaclasse abstrata que define um elemento capaz de ser classificado.

Generalização:

- `ClassifiedElement` estende `DescribableElemente` do SPEM v2, sendo então um tipo de elemento descritivo.

Associações:

- **classElement:** `ClassElement`, define a ligação com um elemento classificativo. Sendo assim, essa associação permite que `ClassifiedElement` seja passível de classificação.

Semântica: Conforme visto em (Laitinen, 1992), devem existir classificações para organizar os artefatos de um PDS uniformemente. Conforme o autor sugere, existe uma nomenclatura, que segue: *Software Description*, *Utilization Documents*, *Development Plans*, *Quality Control Documents* and *Administrative Documents*. A metaclasse `ClassifiedElement` define este conceito de forma a permitir que a utilização das classificações seja possível.

A.4.4 ContentDescription

Descrição: `ContentDescription` é uma metaclasse que serve para determinar o conteúdo base de um elemento descritivo.

Atributos:

- **purpose**: *String*, permite a descrição dos objetivos e finalidades de um elemento descritivo;

- **scope**: *String*, permite a descrição do escopo de um elemento descritivo;

- **overview**: *String*, *String*, permite a descrição de uma visão geral de um elemento descritivo;

Semântica: Para uma melhor compreensão, artefatos de software precisam possuir informação sobre seus objetivos, finalidades e escopo, assim como uma visão geral explicativa. A metaclasse `ContentDescription` redefine alguns elementos da metaclasse original, definida no pacote *Managed Content* do SPEM v2.

A.4.5 DescribableElement

Metaclasse pertencente ao SPEM v2. Verificar especificação em (OMG, 2008b).

A.4.6 LevelTypes

Descrição: Esta enumeração define os níveis de maturidade que podem ser utilizados em uma instância de `MaturityElementLevel`.

Generalização:

- n/a: *Enumeration*

Literais:

- **Controled**: define o nível de maturidade em que o documento possui uma versão bem controlado;
- **Defined**: define o nível de maturidade em que o documento possui uma versão em que sua estrutura já é bem definida;
- **Inconsistent**: define o nível de maturidade em que o documento possui uma versão em que sua estrutura ainda está inconsistente;
- **AdHoc**: define o nível de maturidade em que o documento possui uma versão em que sua estrutura está bastante pobre e sem propriedade;

Semântica: Segundo (Visconti & Cook, 1993), autor dessa abordagem, projetos produzem muita documentação, a qual deve ser controlada de acordo com o nível de maturidade em que se encontram. Tais documentos servem para guardar as informações adquiridas num projeto, o que significa que, conforme a documentação estiver bem feita, o projeto estará em um bom nível de maturidade. Partindo desse princípio, o autor especifica quatro níveis de maturidade: *ad hoc*, incosistente, definida e controlada. Em nossa abordagem, esses níveis estão modeladas através de literais da enumeração `LevelTypes`.

A.4.7 ManagedElement

Descrição: *ManagedElement* é uma metaclasse abstrata que define um elemento capaz de ser possuir níveis de maturidade e ser classificado.

Generalização:

- `ManagedElement` estende `ClassifiedElement`, sendo então um tipo de elemento classificável.
- `ManagedElement` estende `MaturityElement`, sendo então um tipo de elemento com níveis de maturidade.

Semântica: Conforme visto em (Laitinen, 1992), devem existir classificações para organizar os artefatos de um PDS uniformemente. Conforme o autor sugere, existe uma nomenclatura, que segue: *Software Description*, *Utilization Documents*, *Development Plans*, *Quality Control*

Documents and *Administrative Documents*. Já segundo (Visconti & Cook, 1993), devem existir níveis de maturidade para que os documentos possa ser definida como:*ad hoc*, incosistente, definida ou controlada. A metaclasse `ManagedElement` utiliza ambos os conceitos de forma a permitir que seja feita a classificação e utilização dos níveis de maturidade no mesmo elemento.

A.4.8 MaturityElement

Descrição: *MaturityElement* é uma metaclasse abstrata que define um elemento capaz de ser possuir níveis de maturidade.

Generalização:

- `MaturityElement` estende `DescribableElemente` do SPEM v2, sendo então um tipo de elemento descritivo.

Associações:

- **level:** *MaturityElementLevel*, define a ligação com um elemento que contém os níveis de maturidade. Essa associação faz com que `MaturityElementLevel` seja um elemento com níveis de maturidade.

Semântica: Conforme visto em (Visconti & Cook, 1993), devem existir níveis de maturidade para que os documentos possa ser definida como:*ad hoc*, incosistente, definida ou controlada. A metaclasse `MaturityElement` define este conceito de forma a permitir que a utilização dos níveis de maturidade seja possível.

A.4.9 MaturityElementLevel

Descrição: *MaturityElementLevel* é uma metaclasse que serve para determinar o tipo de maturidade de um elemento, utilizando os tipos possíveis e definindo suas descrições chaves.

Generalização:

- `MaturityElementLevel` estende `DescribableElemente` do SPEM v2, sendo então um tipo de elemento descritivo.

Atributos:

- **level**: *LevelTypes*, define uma ligação com uma enumeração de tipos que representa os níveis de maturidade possíveis;
- **keyIndicators**: *String*, permite a descrição dos principais indicadores que levam a crer no nível de maturidade em que o artefato se encontra;
- **KeyPratices**: *String*, permite a descrição das principais práticas adotadas e que levaram o artefato a estar no nível de maturidade em que se encontra;
- **KeyWords**: *String*, permite a descrição do conjunto de palavras chaves do documento;
- **KeyChallenges**: *String*, permite a descrição do conjunto de desafios principais encontrados durante a construção do documento;

- **KeySignificances**: *String*, permite a descrição do conjunto dos principais significados que determinam um sentido para ao documento.

Semântica: Conforme visto em (Visconti & Cook, 1993), devem existir níveis de maturidade para que os documentos possa ser definida. Além disso, cada um desses níveis deve ser descrito pelos seguintes campos: nome, descrição simples, palavras-chaves, principais áreas do processo, principais práticas, principais indicadores, principais desafios e principais significados.

A.5 XtMethodContent

O pacote `XtMethodContent` se constitui das metaclasses modeladas exclusivamente para a estruturação e reuso dos artefatos. Ou seja, as metaclasses para definiçaõ de artefatos, contêineres e tipos de informação podem ser encontradas nesse pacote. Além disso, todos os relacionamentos voltados para a conexão entre esses três elementos, assim como os níveis de reuso existentes também estão definidos. Na Figura A.12 é apresentado o diagrama com as metaclasses principais desse pacote. Já Na Figura A.13 está ilustrado o diagrama que permite visualizar quais os pacotes participam do *package merge*.

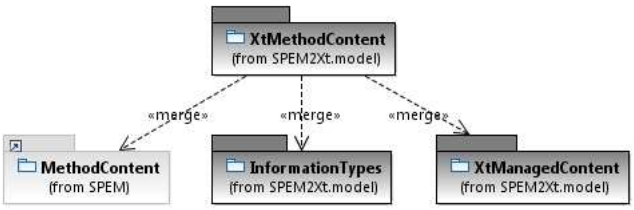

Figura A.12: Aplicação de *package merge* para no pacote `XtMethodContent`

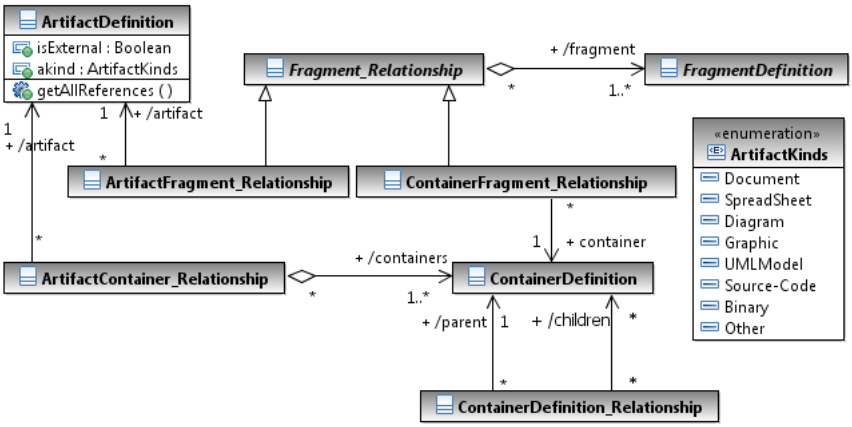

Figura A.13: Metaclasses para definição de autoria de Artefatos.

A.5.1 ArtifactContainer_Relationship

Descrição: `ArtifactContainer_Relationship` é uma metaclasse que determina o relacionamento entre artefatos e contêineres.

Generalização:

- `ArtifactContainer_Relationship` estende a metaclasse de definição de produtos de trabalho `WorkProductDefinitionRelationship`, sendo então um relacionamento entre produtos de trabalho.

Associações:

- **artifact:** *ArtifactDefinition*, esta associação representa a ligação com um artefato;
- **containers:** *ContainerDefinition*, esta associação representa a ligação com contêineres.

Semântica: Artefatos utilizam organizadores de informações para obter uma melhor estrutura interna. Tais organizadores são geralmente vistos como capítulos ou seções. Desta forma, uma vez que os contêineres servem para agrupar as informações, os artefatos podem então se relacionar com tais contêineres, afim de obter as informações já organizadas. A metaclasse `ArtifactContainer_Relationship` define esse tipo de ligação, provendo uma associação entre um artefato e seus contêineres.

A.5.2 ArtifactDefinition

Descrição: `ArtifactDefinition` é uma metaclasse abstrata que representa os produtos de trabalho do tipo artefato.

Generalização:

- `ArtifactDefinition` estende `WorkProductDefinition`, sendo então um tipo de produto de trabalho.

Atributos:

- **isExternal:** *Boolean*, determina se o artefato em si é construído dentro de um processo próprio, através das atividades, ou se é um artefato externo, já pronto mas que precisa ser utilizado;
- **aKind:** *ArtifactKinds*, define uma ligação com uma enumeração de tipos que representa determina o tipo de artefato com base nos tipos existentes, assim como documento, código-fonte, entre outros.

Semântica: Durante a execução de atividades, obtém-se como resultado vários tipos de produtos de trabalhos diferentes. Entre eles estão os artefatos. A metaclasse `Artifact Definition` serve como uma representação desses artefatos.

A.5.3 ArtifactFragment_Relationship

Descrição: `ArtifactFragment_Relationship` é uma metaclasse que determina o relacionamento entre um artefato e seus tipos de informação.

Generalização:

- `ArtifactFragment_Relationship` estende `Fragment_Relationship`, sendo então um relacionamento com objetivo específico de se associar com algum fragmento.

Associações:

- **artifact:** *ArtifactDefinition*, esta associação representa a ligação com um artefato;

Semântica: Artefatos são responsáveis por agrupar tipos de informações relacionados, obtendo uma conjunto de informações. Essas informações, separadas, podem ser consideradas fragmentos de um todo. A metaclasse `ArtifactFragment_Relationship` define essa associação entre um artefato e seus fragmentos de informação.

A.5.4 ArtifactKinds

Descrição: Esta enumeração define quais diagramas podem ser utilizados em uma instância de `ArtifactDefinition`.

Generalização:

- n/a: *Enumeration*

Literais: Todos utilizados em instâncias de `ArtifactDefinition`.

- **Document**: define que uma instância será do tipo documento;
- **SpreadSheet**: define que uma instância será do tipo planilha;
- **Diagram**: define que uma instância será do tipo diagrama;
- **Graphic**: define que uma instância será do tipo gráfico;
- **UMLModel**:define que uma instância será do tipo UML;
- **Source-Code**: define que uma instância será do tipo código-fonte;
- **Binary**: define que uma instância será do tipo binário;
- **Other**: não possui tipo definido;

Semântica: Partindo do princípio de que artefatos podem ser de diversos tipos, a enumeração `ArtifactKinds` possui literais que foram modelados conforme tipos de artefatos definidos nessa abordagem.

A.5.5 ContainerDefinition

Descrição: `ContainerDefinition` é uma metaclasse abstrata que representa os produtos de trabalho definidos como agrupadores de informação.

Generalização:

- `ContainerDefinition` estende `WorkProductDefinition`, sendo então um tipo de produto de trabalho.

Semântica: Durante a execução de atividades, obtém-se como resultado vários tipos de produtos de trabalhos diferentes. Entre eles estão as informações que geralmente são agrupadas dentro de algum artefato. Entretanto, os artefatos podem se utilizar de estruturas de organização. Tais estruturas geralmente são vistas como capítulos ou seções, funcionando como um esqueleto que estruturas informações relacionadas. A metaclasse `ContainerDefinition` representa esses agrupadores.

A.5.6 ContainerDefinition_Relationship

Descrição: `ContainerDefinition_Relationship` é uma metaclasse que determina o auto-relacionamento entre contêineres.

Generalização:

- `ContainerDefinition_Relationship` estende `WorkProductDefinition Relationship`, sendo então um relacionamento entre produtos de trabalho.

Associações:

- **parent:** *ContainerDefinition*, esta associação representa a ligação com um contêiner "pai";
- **children:** *ContainerDefinition*, esta associação representa a ligação com os contêineres "filhos";

Semântica: Contêineres são utilizados para agrupar informações semelhantes de forma a torná-las mais organizadas. Assim como pode ser utilizados como uma representação para capítulos e seções de um artefato. Conforme sabemos, seções são subpartes de um capítulo, sendo assim, tal estrutura também deve ser possível com a utilização de contêineres. Este auto-relacionamento é definido pela metaclasse `ContainerDefinition_Relationship`. Isto define uma associação do tipo "*child/parent*" entre os contêineres.

A.5.7 ContainerFragment_Relationship

Descrição: `ContainerFragment_Relationship` é uma metaclasse que determina um relacionamento entre contêineres e tipos de informação.

Generalização:

- `ContainerFragment_Relationship` estende `Fragment_Relationship`. Desta forma, torna-se um relacionamento com objetivo específico de se associar a algum fragmento.

Associações:

- **container:** *ContainerDefinition*, esta associação representa a ligação com um contêiner;

Semântica: Contêineres são responsáveis por agrupar e estruturar diferentes tipos de informações relacionados, mantendo-os organizados. A metaclasse `ContainerFragment_ Relationship` define essa associação entre um contêiner e seus tipos de informação.

A.5.8 FragmentDefinition

Descrição: `FragmentDefinition` é uma metaclasse abstrata que representa os produtos de trabalho definidos como tipos de informação.

Generalização:

- `FragmentDefinition` estende `WorkProductDefinition`, sendo então um tipo de produto de trabalho.

Semântica: Durante a execução de atividades, obtém-se como resultado vários tipos de produtos de trabalhos diferentes. Entre eles estão as informações que geralmente são agrupadas dentro de algum artefato. A metaclasse `FragmentDefinition` representa quaisquer tipos de informação.

A.5.9 Fragment_Relationship

Descrição: `Fragment_Relationship` é uma metaclasse abstrata que determina um relacionamento com tipos de informação.

Generalização:

- `Fragment_Relationship` estende `WorkProductDefinitionRelationship`, sendo então um relacionamento entre produtos de trabalho.

Associações:

- **fragment:** *FragmentDefinition*, esta associação representa a ligação com tipos de informação;

Semântica: Artefatos são responsáveis por agrupar tipos de informações relacionados, obtendo uma conjunto de informações. Além do mais, contêineres são utilizados para agrupar informações semelhantes de forma a torná-las mais organizadas. Desta forma, artefatos e contêineres devem possuir relacionamento com os tipos de informação necessários. A metaclasse `Fragment_Relationship` define uma associação abstrata com tipos de informação.

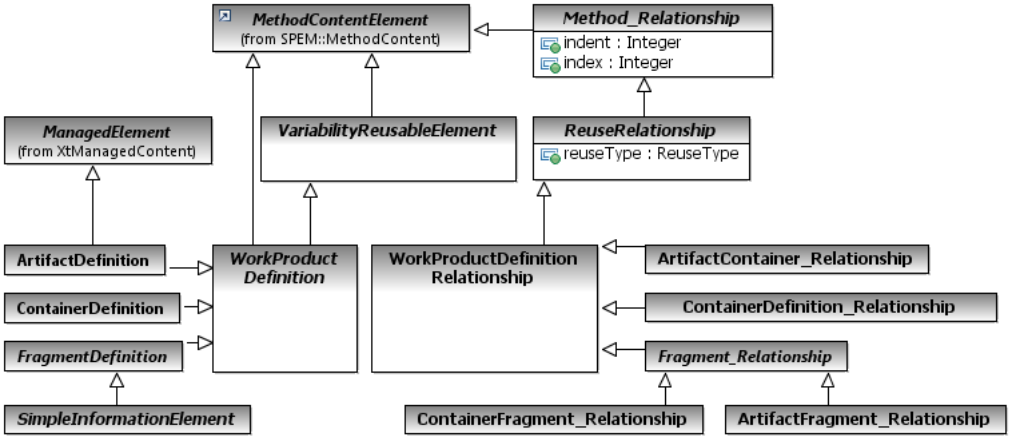

Figura A.14: Estrutura hierárquica das metaclasses para definição de autoria de Artefatos.

A.5.10 ManagedElement

definido na Seção A.4.7.

A.5.11 MethodContentElement

Metaclasse do SPEM v2. Verificar especificação em (OMG, 2008b).

A.5.12 Method_Relationship

Descrição: `Method_Relationship` é uma metaclasse abstrata que determina como devem ser os relacionamentos do *Method Content*.

Generalização:

- `Method_Relationship` estende `MethodContentElement` do SPEM v2, sendo então um elemento do *Method Content*.

Atributos:

- **indent:** *Integer*, determina o nível de identação entre os participantes do relacionamento. Ex.: em uma ligação entre um artefato "a" e dois contêineres "b", com indent=1 e "c" com indent=2, faz com que b e c sejam as seções x.1 e x.2, respectivamente. Independente da hierarquia;
- **index:** *Integer*, determina o nível de hierarquia entre os participantes do relacionamento. Ex.: em uma ligação entre um artefato "a" e dois contêineres "b", com index=1 e "c" com index=2, faz com que b e c sejam as seções "1.x" e "2.x", respectivamente. Independente do nível de indentação.

Semântica: As ligações entre artefatos, contêineres e tipos de informação possuem uma hierarquia que pode ser vista, geralmente, conforme números de capítulos ou seções. Além disso, um capítulo possui subpartes, que são as seções, por isso, se existe um capitulo "1", suas seções serão "1.1", "1.2", etc. A metaclasse `Fragment_Relationship` define uma associação abstrata com tipos de informação.

A.5.13 ReuseRelationship

Descrição: `ReuseRelationship` é uma metaclasse abstrata que determina como devem ser os relacionamentos que especificam níveis de reuso.

Generalização:

- `ReuseRelationship` estende `Method_Relationship` do SPEM v2, sendo então um relacionamento capaz de definir hierarquia e identação.

Atributos:

- **reuseType:** *ReuseType*, define qual o tipo de reuso que será utilizado;

Semântica: Em nossa abordagem os níveis de reuso são definidos através das ligações entre artefatos, contêineres e tipos de informação. A metaclasse `ReuseRelationship` define como isso pode ser feito através da especificação de relacionamentos com níveis de reuso.

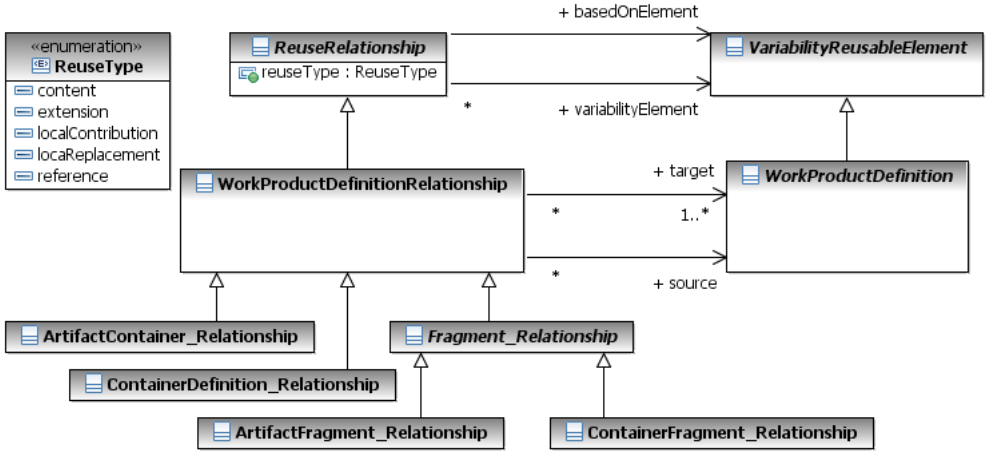

Figura A.15: Metaclasses para definição de reuso de Artefatos.

A.5.14 ReuseType

Descrição: Esta enumeração define quais diagramas podem ser utilizados em uma instância de `ArtifactDefinition`.

Generalização:

- n/a: *Enumeration*

Literais:

- **content**: define o tipo de reuso no nível de conteúdo;
- **extension**: define o tipo de reuso no nível de extensão;
- **localContribution**: define o tipo de reuso no nível de contribuição;
- **localReplacement**: define o tipo de reuso no nível de reposição;
- **reference**: define o tipo de reuso no nível de referencia;

Semântica: Existem vários níveis de reuso, conforme a necessidade de se obter diferentes resultados na aplicação de reusabilidade. Em nossa abordagem, o reuso é uma operação binária definida entre dois participantes, sendo assim, um relacionamento. Nesse sentido, a enumeração `ReuseType` determina exatamente quais os tipos possíveis de reuso a partir de seus literais.

A.5.15 SimpleInformationElement

definido na Seção A.3.4

A.5.16 VariabilityReusableElement

Descrição: `VariabilityReusableElement` é uma metaclasse abstrata que representa um elemento passível de reuso.

Generalização:

- `VariabilityReusableElement` estende `MethodContentElement` do SPEM v2, sendo então um elemento do *Method Content*.

Semântica: De acordo com nossa abordagem, alguns elementos são passíveis de reuso, assim como artifatos e tipos de informação. Nesse sentido a metaclasse `VariabilityReusable Element` representa de modo abstrato os elementos que reutilizam e os podem ser reutilizados.

A.5.17 WorkProductDefinition

Descrição: `WorkProductDefinition` é uma metaclasse que representa os produtos de trabalho (*Work Products*) resultantes da execução de atividades ou tarefas de um PDS.
Generalização:

- `WorkProductDefinitionRelationship` é uma metaclasse que estende `Reuse Relationship` do SPEM v2, sendo então um relacionamento capaz de definir níveis reuso.

Semântica: Existem diversos produtos de trabalho que são construídos durante a execução de alguma atividade ou tarefa do PDS. Tais produtos geralmente são tratados como artefatos, entretanto, em nossa abordagem, pode ser qualquer fragmento de informação, tipo ou seção que foi produzida. A metaclasse `WorkProductDefinition` representa quaisquer tipos de produtos de trabalho.

A.5.18 WorkProductDefinitionRelationship

Descrição: `WorkProductDefinitionRelationship` é uma metaclasse que define como deve ser um relacionamento entre produtos de trabalho (*Work Products*).
Generalização:

- `WorkProductDefinitionRelationship` estende `ReuseRelationship`, a metaclasse da especificação do SPEM v2, sendo então um relacionamento capaz de definir níveis reuso.

Semântica: Produtos de trabalho, construídos durante a execução de alguma atividade ou tarefa, podem ser vistos conforme a estrutura que possuem. Ou seja, um desses produtos pode ser uma união ou agrupamento de outros produtos. Nesse sentido, é necessário definir um auto-relacionamento capaz de atender a essa necessidade. A metaclasse `WorkProduct DefinitionRelationship` permite uma associação entre produtos de trabalho.

A.6 XtProcessStructure

O pacote `XtProcessWithMethods` se constitui das metaclasses que fazem uso do elementos que constrõem o `Method Content`. Essas metaclasses são responsáveis por manter a estrutura processual de acordo com o que estiver modelado na biblioteca de conteúdo. Desta forma, a extensão feita através da operação de *package merge* desse pacote com o pacote original `ProcessStructure` do SPEMv2, adiciona a qualidade de usar artefatos bem definidos e estruturados a partir dos tipos de informação.

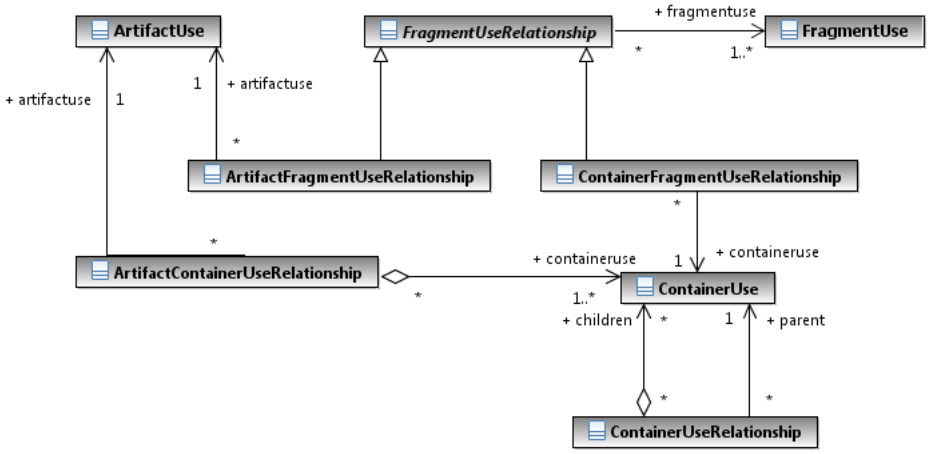

Figura A.16: Metaclasses para definição de uso de Artefatos.

A.6.1 ArtifactContainerUseRelationship

Descrição: `ArtifactContainerUseRelationship` é uma metaclasse que determina o relacionamento entre o uso de artefatos e o uso contêineres.

Generalização:

- `ArtifactContainerUseRelationship` estende a metaclasse de definição de relacionamentos `ProcessStructure_Relationship`, sendo então um elemento do *Process Structure*.

Associações:

- **artifactuse:** *ArtifactUse*, esta associação representa a ligação com um uso de artefato;
- **containeruse:** *ContainerUse*, esta associação representa a ligação com o uso de contêineres.

Semântica: Durante a modelagem de um PDS, no uso de artefatos são adicionados organizadores de informações, assim como capítulos ou seções. Desta forma, uma vez que os contêineres servem para agrupar as informações, os artefatos podem então se relacionar com tais contêineres, afim de obter as informações já organizadas. A metaclasse `ArtifactContainerUseRelationship` define esse tipo de ligação, provendo uma associação entre o uso do artefato e o uso de seus contêineres.

A.6.2 ArtifactFragmentUseRelationship

Descrição: `ArtifactFragmentUseRelationship` é uma metaclasse que determina o relacionamento entre um o uso de artefatos e uso de seus tipos de informação.

Generalização:

- `ArtifactFragmentUseRelationship` estende a metaclasse de definição de relacionamentos `ProcessStructure_Relationship`, sendo então um elemento do *Process Structure*.

Associações:

- **artifactuse:** *ArtifactDefinition*, esta associação representa a ligação com um uso de artefato;

Semântica: Artefatos são responsáveis por agrupar tipos de informações relacionados. A metaclasse `ArtifactFragmentUseRelationship` define uma associação entre esses dois conceitos no momento de criação da estrutura do processo.

A.6.3 ArtifactUse

Descrição: `ArtifactUse` é uma metaclasse que representa o uso dos produtos de trabalho do tipo artefato para a estruturação do processo.

Generalização:

- `ArtifactUse` estende `WorkProductUse`, sendo definido como um produto de trabalho no momento de estruturação do processo.

Semântica: Durante a modelagem do PDS é necessário selecionar quais os elementos do *Method Content* participarão do *Process Structure*, fazendo uso dos mesmos. Esta metaclasse serve como reflexo da metaclasse `ArtifactDefinition` do *Method Content*, permitindo assim o uso de artefatos durante a estruturação do Processo.

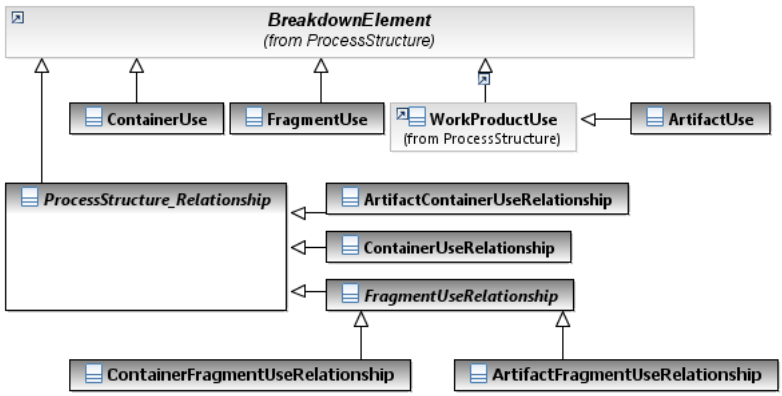

Figura A.17: Estrutura hierárquica das metaclasses utilizadas na extensão da estrutura de PDSs.

A.6.4 BreakDownElement

Metaclasse do SPEM v2. Verificar especificação em (OMG, 2008b).

A.6.5 ContainerFramentUseRelationship

Descrição: `ContainerFramentUseRelationship` é uma metaclasse que determina um relacionamento entre contêineres e tipos de informação no momento de estruturação do processo.

Generalização:

- `ContainerFragmentUseRelationship` estende a metaclasse de relacionamentos `FragmentUse_Relationship`, sendo então um relacionamento com objetivo específico de se associar o uso de fragmentos.

Associações:

- **containeruse:** *ContainerUse*, esta associação representa a ligação com o uso de um contêiner;

Semântica: Durante a modelagem de processos, contêineres são responsáveis por agrupar e estruturar diferentes tipos de informações relacionados. A metaclasse `ContainerFragment UseRelationship` define uma associação entre o uso de um contêiner e seus respectivos tipos de informação.

A.6.6 ContainerUse

Descrição: `ContainerUse` é uma metaclasse que determina o uso de contêineres durante a estruturação de PDSs.

Generalização:

- `ContainerUse` estende `BreakDownElemento`, sendo então um tipo de elemento da estrutura de WBS.

Semântica: Durante a modelagem do PDS é necessário selecionar quais os elementos do *Method Content* participarão do *Process Structure*, fazendo uso dos mesmos. Esta metaclasse serve como reflexo da metaclasse `ContainerDefinition` do *Method Content*, permitindo assim o uso de contêineres durante a estruturação do Processo.

A.6.7 ContainerUseRelationship

Descrição: `ContainerUseRelationship` é uma metaclasse que determina o auto-relacionamento entre o uso de contêineres.

Generalização:

- `ContainerUseRelationship` estende `ProcessStructure_Relationship`, sendo então um elemento do *Process Structure*.

Associações:

- **parent:** *ContainerUse*, esta associação representa a ligação com o uso de um contêiner "pai";

- **children:** *ContainerUse*, esta associação representa a ligação com o uso de contêineres "filhos";

Semântica: Na modelagem de artefatos, no momento de estruturação do processo, os contêineres serão tratados como seções e subseções. Sendo assim, este auto-relacionamento é definido pela metaclasse `ContainerUseRelationship`, definindo uma associação do tipo "*child/parent*" entre o uso de contêineres.

A.6.8 FragmentUse

Descrição: `FragmentUse` é uma metaclasse abstrata que representa o uso de tipos de informação.

Generalização:

- `FragmentUse` estende `BreakDownElement` do Spem v2, sendo então um tipo de elemento da estrutura de WBS.

Semântica: Durante a modelagem do PDS é necessário selecionar quais os elementos do *Method Content* participarão do *Process Structure*, fazendo uso dos mesmos. Esta metaclasse serve como reflexo da metaclasse `FragmentDefinition` do *Method Content*, permitindo assim o uso de tipos de informação durante a estruturação do Processo.

A.6.9 FragmentUseRelationship

Descrição: `FragmentUseRelationship` é uma metaclasse abstrata que determina um relacionamento com o uso dos tipos de informação.

Generalização:

- `FragmentUseRelationship` estende `ProcessStructure_Relationship`. Desta forma torna-se um elemento do *Process Structure*.

Associações:

- **fragmentuse:** *FragmentDefinition*, esta associação representa a ligação com o uso de tipos de informação;

Semântica: Durante a modelagem de processos, tanto artefatos quanto contêineres são utilizados para agrupar os tipos de informação através de relacionamento. Isto é definido pela metaclasse `FragmentUseRelationship` define uma associação abstrata com o uso de tipos de informação.

A.6.10 ProcessStructure_Relationship

Descrição: `ProcessStructure_Relationship` é uma metaclasse abstrata que determina como devem ser os relacionamentos do *Process Structure*.

Generalização:

- `ProcessStructure_Relationship` estende `BreakDownElement` do Spem v2, sendo então um tipo de elemento da estrutura de WBS.

Semântica: As ligações entre artefatos, contêineres e tipos de informação precisam ser elementos capazes de mapear através da utilização de uma estrutura WBS. A metaclasse `Process Structure_Relationship` define uma associação abstrata entre o uso desses elementos.

A.6.11 WorkProductUse

Metaclasse pertencente ao SPEM v2. Verificar especificação em (OMG, 2008b).

A.7 XtProcessWithMethods

O pacote `XtProcessWithMethods` se constitui das metaclasses modeladas para definição tanto de processos quanto de conteúdos a partir da estruturação dos artefatos. Além disso, todos os relacionamentos são voltados para a conexão entre a definição dos elementos do *Method Content* juntamente com o uso para a criação dos elementos do *Process Structure*. Na Figura A.18 apresenta o diagrama que permite a visualização da operação de *package merge* entre os pacotes participam da construção do pacote `XtProcessWithMethods`.

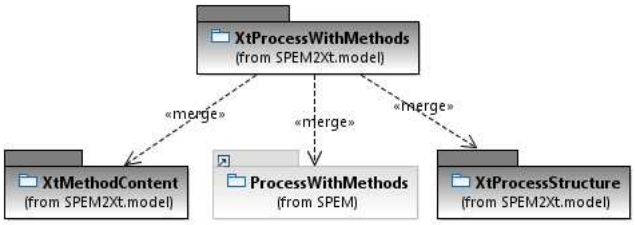

Figura A.18:

Na Figura A.19 é apresentado um diagrama que mostra a ligação entre metaclasses principais do *Method Content* e do *Processos Structure*. Esta ligação é definida por *trace*, onde um elemento do PDS depende de um elemento da definição. Embora não esteja explícito, os elementos de uso definidos no pacote `XtProcessStructure` fazem relacionamento com os elementos de uso do pacote XtProcessWithMethods através do padrão *Proxy*.

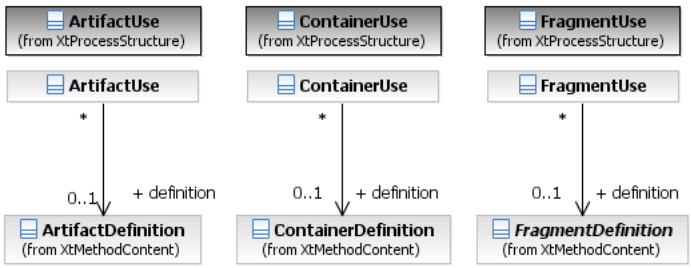

Figura A.19:

A.7.1 ArtifactDefinition

definido na Seção A.5.2

A.7.2 ArtifactPackage

Generalização:

- `ArtifactPackage` estende a metaclasse `MethodContentPackage`, sendo então um pacote para elementos do *Method Content*

Associações:

- **artifactElement:** *ArtifactDefinition*, esta associação denomina que instâncias da metaclasse *ArtifactDefinition* podem ser empacotadas;

- **containerElement:** *ContainerDefinition*, esta associação denomina que instâncias da metaclasse *ContainerDefinition* podem ser empacotadas;

Semântica: Durante a modelagem de PDS, notou-se que a quantidade de artefatos gerados, juntamente com seus respectivos contêineres, cresce de acordo com a quantidade de informação existente. Desta forma, para que fosse possível guardá-los de forma que não se misturasse com outros elementos do processos, houve a necessidade da criação de um pacote. A metaclasse `ArtifactPackage` é capaz de definir um pacote que possibilita o agrupamento desses artefatos e contêineres.

A.7.3 ArtifactUse

Descrição: `ArtifactUse` é uma metaclasse que representa o uso de artefatos a partir de um *trace* com a metaclasse *ArtifactDefinition*.

Associações:

- **definition:** *ArtifactDefinition*, esta associação, denominada *trace*, representa a ligação com a definição do artefato permitindo o uso;

Semântica: Durante a modelagem do PDS é necessário selecionar quais os elementos do *Method Content* participarão do *Process Structure*, fazendo uso dos mesmos. Esta metaclasse serve como reflexo da metaclasse `ArtifactDefinition` do *Method Content*, permitindo assim o uso de artefatos durante a estruturação do Processo.

A.7.4 BreakDownElement

Metaclasse do SPEM v2. Verificar especificação em (OMG, 2008b).

A.7.5 ContainerDefinition

definido na Seção A.5.5

A.7.6 ContainerUse

Descrição: `ContainerUse` é uma metaclasse que representa o uso de contêineres a partir de um *trace* com a metaclasse *ContainerDefinition*.

Associações:

- **definition:** *ContainerDefinition*, esta associação, denominada *trace*, representa a ligação com a definição do contêiner permitindo o uso;

Semântica: Durante a modelagem do PDS é necessário selecionar quais os elementos do *Method Content* participarão do *Process Structure*, fazendo uso dos mesmos. Esta metaclasse serve como reflexo da metaclasse `ContainerDefinition` do *Method Content*, permitindo assim o uso de contêineres durante a estruturação do Processo.

A.7.7 FragmentDefinition

definido na Seção A.5.8

A.7.8 FragmentUse

Descrição: `FragmentUse` é uma metaclasse abstrata que representa o uso de tipos de informação a partir de um *trace* com a metaclasse *FragmentDefinition*.

Associações:

- **definition:** *FragmentDefinition*, esta associação, denominada *trace*, representa a ligação com a definição de tipos de informação permitindo o uso;

Semântica: Durante a modelagem do PDS é necessário selecionar quais os elementos do *Method Content* participarão do *Process Structure*, fazendo uso dos mesmos. Esta metaclasse serve como reflexo da metaclasse `FragmentDefinition` do *Method Content*, permitindo assim o uso de tipos de informação durante a estruturação do Processo.

A.7.9 InformationPackage

Descrição: `InformationPackage` é uma metaclasse que agrupa os tipos de informação funcionando como um pacote.

Generalização:

- `InformationPackage` estende a metaclasse `MethodContentPackage`, sendo então um pacote para elementos do *Method Content*

Associações:

- **informationElement:** *FragmentDefinition*, esta associação, denomina que os elementos do tipo *FragmentDefinition* podem ser empacotados;

Semântica: Durante a modelagem de PDS, notou-se que a quantidade de tipos de informação geradas era muito grande. Desta forma, para que fosse possível guardá-los de forma que não se misturasse com outros elementos do processos, houve a necessidade da criação de um pacote. A metaclasse `InformationPackage` é capaz de definir um pacote que possibilita o agrupamento desses tipos de informação.

A.7.10 InformationUsePackage

Descrição: `InformationUsePackage` é uma metaclasse que agrupa o uso dos tipos de informação funcionando como um pacote.

Generalização:

- `InformationUsePackage` estende a metaclasse `ProcessPackage`, sendo então um pacote para elementos do *process Structure*

Associações:

- **fragmentuse:** *FragmentUse*, esta associação, denomina que os elementos do tipo *FragmentUse* podem ser empacotados;

Semântica: Durante a modelagem de PDS, notou-se que a quantidade de tipos de informação geradas era muito grande. Desta forma, para que fosse possível guardá-los de forma que não se misturasse com outros elementos do processos, houve a necessidade da criação de um pacote. A metaclasse `InformationPackage` é capaz de definir um pacote que possibilita o agrupamento desses tipos de informação.

A.7.11 MethodContentPackage

Metaclasse do SPEM v2. Verificar especificação em (OMG, 2008b).

A.7.12 MethodContentPackageableElement

Metaclasse do SPEM v2. Verificar especificação em (OMG, 2008b).

A.7.13 MethodContentUse

Metaclasse do SPEM v2. Verificar especificação em (OMG, 2008b).

A.7.14 MethodContentRelationshipPackage

Descrição: `MethodContentRelationshipPackage` é uma metaclasse que agrupa todos os relacionamentos entre os elementos do *Method Content*.

Generalização:

- `MethodContentRelationshipPackage` estende a metaclasse `MethodContentPackage`, sendo então um pacote para elementos do *Method Content*

Associações:

- **relationshipElement:** *Method_Relationship*, esta associação, denomina que os elementos capazes de definir relacionamentos, os quai são instâncias de *Method_Relationship* podem ser empacotados;

Semântica: Durante a modelagem de PDS, notou-se que a quantidade de tipos de informação geradas era muito grande. Desta forma, para que fosse possível guardá-los de forma que não se misturasse com outros elementos do processos, houve a necessidade da criação de um pacote. A metaclasse `InformationPackage` é capaz de definir um pacote que possibilita o agrupamento desses tipos de informação.

A.7.15 Method_Relationship

definido na Seção A.5.12.

A.7.16 Package

Metaclasse pertencente ao pacote *Constructs* da *UML Infrastructure Library*. Verificar especificação em (OMG, 2007b).

A.7.17 PackageableElement

Metaclasse pertencente ao pacote *Constructs* da *UML Infrastructure Library*. Verificar especificação em (OMG, 2007b).

A.7.18 ProcessStructure_Relationship

Descrição: `ProcessStructure_Relationship` é uma metaclasse abstrata que determina como devem ser os relacionamentos do *Process Structure*.

Generalização:

- `ProcessStructure_Relationship` estende `MethodContentUse` do Spem v2, sendo então um tipo de elemento de estrutura de PDSs.

Semântica: As ligações entre artefatos, contêineres e tipos de informação precisam ser elementos capazes de mapear através da utilização dos artefatos utilizados uma estruturação dos PDSs. A metaclasse `Process Structure_Relationship` define uma associação abstrata para o uso desses elementos.

A.7.19 ProcessPackage

Metaclasse do SPEM v2. Verificar especificação em (OMG, 2008b).

A.7.20 ProcessPackageableElement

Metaclasse do SPEM v2. Verificar especificação em (OMG, 2008b).

A.7.21 WorkProductDefinitionRelationship

Descrição: `WorkProductDefinitionRelationship` é uma metaclasse que define como deve ser um relacionamento entre produtos de trabalho (*Work Products*).

Generalização:

- `WorkProductDefinitionRelationship` estende a metaclasse abstrata `Method_Relationship`, sendo então um relacionamento capaz de definir hierarquia e identação.

Semântica: Produtos de trabalho, construídos durante a execução de alguma atividade ou tarefa, podem ser vistos conforme a estrutura que possuem. Ou seja, um desses produtos pode ser uma união ou agrupamento de outros produtos. Nesse sentido, é necessário definir um auto-relacionamento capaz de atender a essa necessidade. Desta forma, a utilização da metaclasse `WorkProductDefinitionRelationship` permite uma associação entre produtos de trabalho.

A.7.22 WorkProductUse

Metaclasse do SPEM v2. Verificar especificação em (OMG, 2008b).

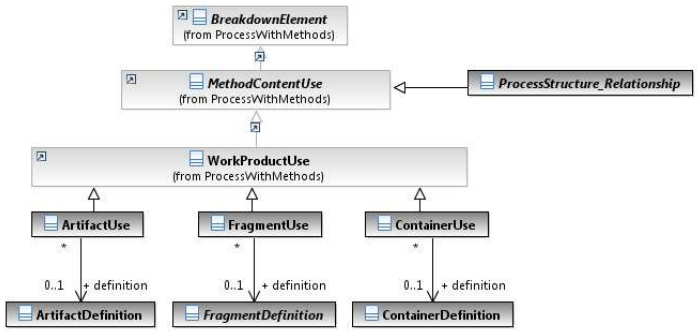

Figura A.20: *Trace* entre metaclasses do *Process Structure* e do *Method Content*

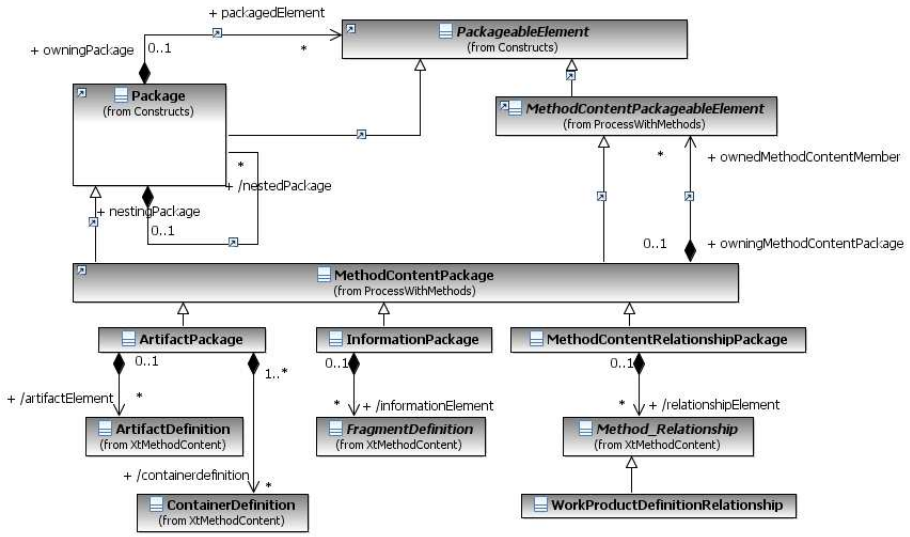

Figura A.21: Estrutura hierárquica das metaclasses para o empacotamento do *Method Content*

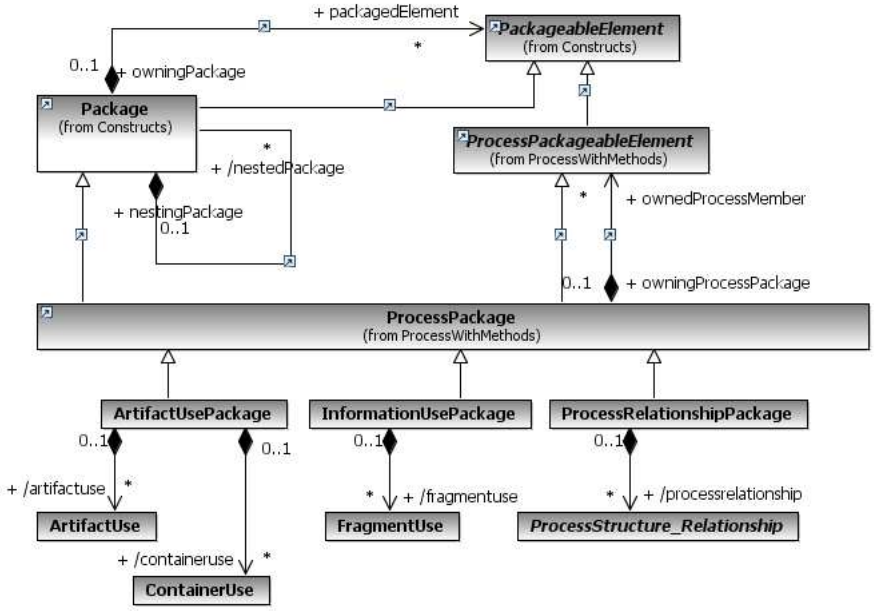

Figura A.22: Estrutura hierárquica das metaclasses para o empacotamento do *Process Structure*

A.8 XtMethodPlugin

O pacote `XtMethodPlugin` se constitui das metaclasses modeladas para definição de uma biblioteca capaz de permitir a utiliza de todos os elementos constituintes do metamodelo. Este pacote não adiciona nenhuma nova metaclasse ao pacote original (`MethodPlugin` do SPEM v2), cabendo a ele apenas fazer um merge com todas as alterações existentes. Na Figura A.23 é apresentado o diagrama que permite a visualização da operação de *package merge* entre os pacotes participam da construção do pacote `XtMethodPlugin`.

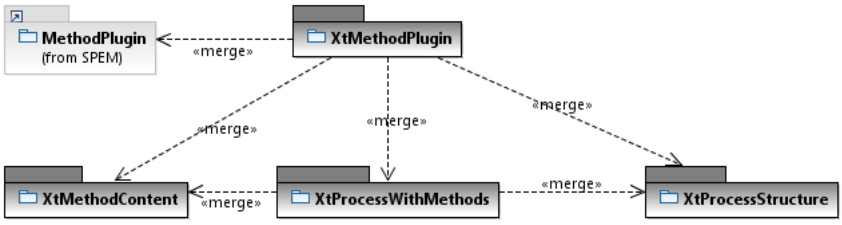

Figura A.23: Aplicação de *package merge* para no pacote XtMethodPlugin

Apêndice B Catálogo do SPEMXt UML Profile

B.1 Diagramas

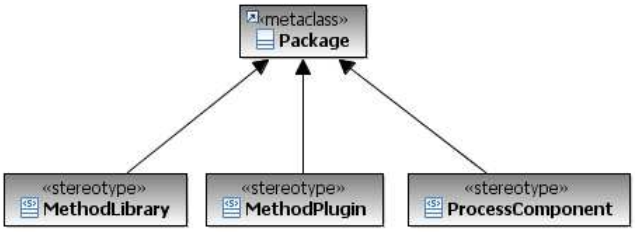

Figura B.1: Estereótipos relacionados ao *Method Library*

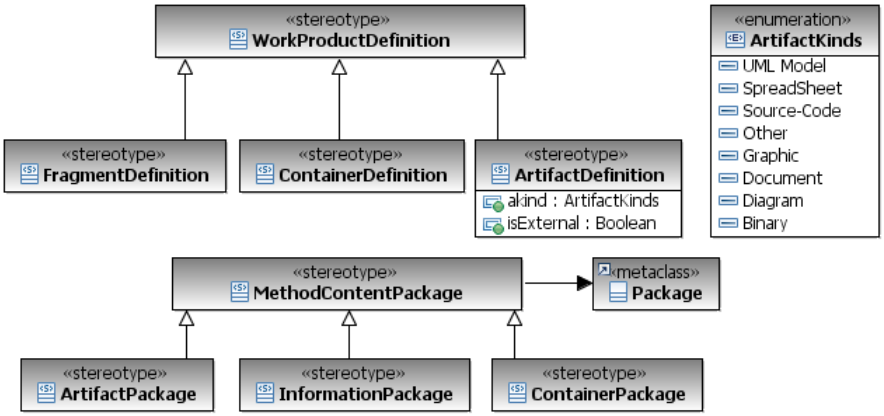

Figura B.2: Estereótipos relacionados ao *Method Content*

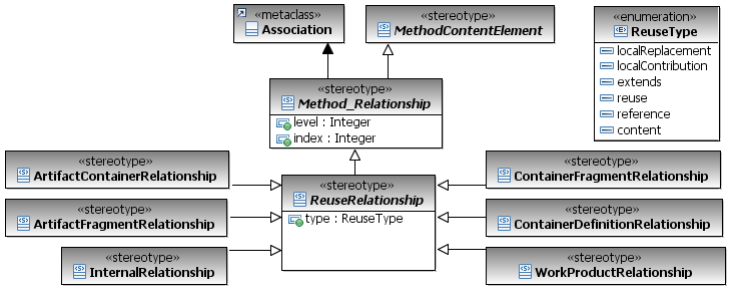

Figura B.3: Estereótipos relacionados aos relacionamentos do *Method Content*

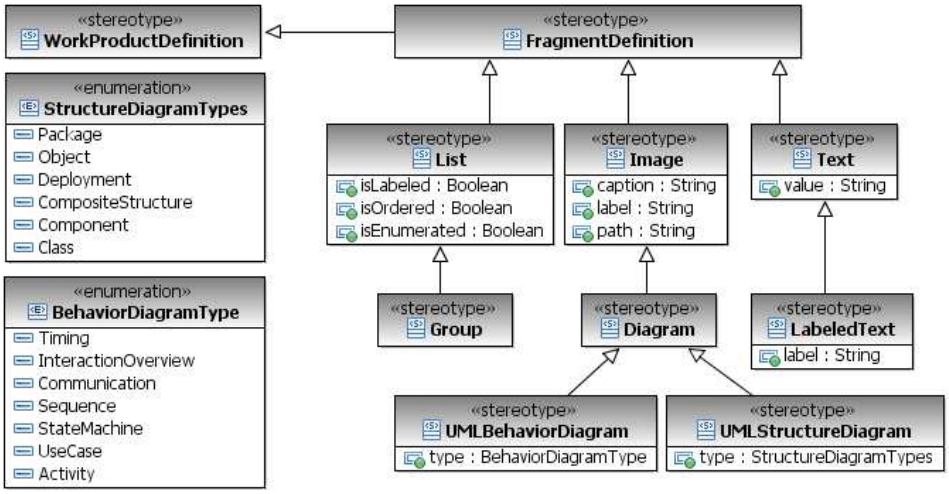

Figura B.4: Estereótipos relacionados aos *Information Types*

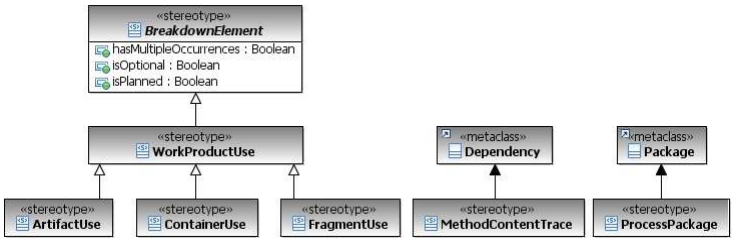

Figura B.5: Estereótipos relacionados aos *Process Structure*

B.2 UML *Profiles*

A Tabela B.2 apresenta um sumário dos estereótipos existentes no ***SPEMXt UML Profile***.

Tabela B.1: *UML Profile* da a Extensão do SPEM

Estereótipo	Meta-classe/Superclasse	Propriedades	Abstrata	Ícone
ArtifactContainer_Relationship	ReuseRelationship	n/a	no	n/a
ArtifactDefinition	WorkProductDefinition	aKind, isExternal	no	
ArtifactFragment _Relationship	Association/ ReuseRelationship	n/a	no	n/a
ArtifactPackage	MethodContentPackage	n/a	no	
ArtifactUse	WorkProductUse	n/a	no	
ContainerDefinition	WorkProductDefinition	n/a	no	
ContainerDefinition _Relationship	ReuseRelationship	n/a	no	n/a
ContainerFragment _Relationship	ReuseRelationship	n/a	no	n/a
ContainerPackage	MethodContentPackage	n/a	no	
ContainerUse	WorkProductUse	n/a	no	
Diagram	Image	n/a	no	
FragmentDefinition	WorkProductDefinition	n/a	yes	n/a
FragmentDefinition Relationship	ReuseRelationship	n/a	no	n/a
FragmentUse	WorkProductUse	n/a	no	
Group	List	n/a	no	
Image	FragmentDefinition	caption, label, path	no	
InformationPackage	MethodContentPackage	n/a	no	
LabeledText	Text	label	no	
List	FragmentDefinition	isEnumerated, isLabeled, isOrdered	no	
Method_Relationship	Association/ MethodContentElement	index, level	yes	n/a
ReuseRelationship	Method_Relationship	type	yes	n/a
Text	FragmentDefinition	value	no	

Apêndice C Visões Lógicas dos Artefatos

Neste Apêndice estão os diagramas modelados na Seção 5.2.2, na qual está o segundo cenário de testes. Esse cenário faz uso do *SPEMXt UML Profile*. As visões foram criadas para facilitar o entendimento da abordagem, uma vez que mostra, de forma completa, a modelagem dos artefatos participantes dos testes com a aplicação de seus respectivos estereótipos.

Os as visões lógicas estão criadas nos diagramas que estão representados através da Figuras:

- Figura C.1 apresenta o diagrama que possibilita a visão lógica do artefato Glossário de Negócios;
- Figura C.2 apresenta o diagrama que possibilita a visão lógica do artefato Glossário;Design;
- Figura C.3 apresenta o diagrama que possibilita a visão lógica do artefato Especificação de Caso de Uso;
- Figura C.4 apresenta o diagrama que possibilita a visão lógica dos artefatos Modelo de Análise e Modelo de

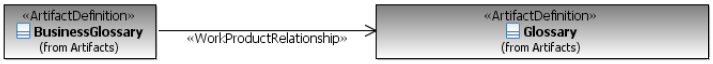

Figura C.1: *Diagrama*: Glossário de Negócios

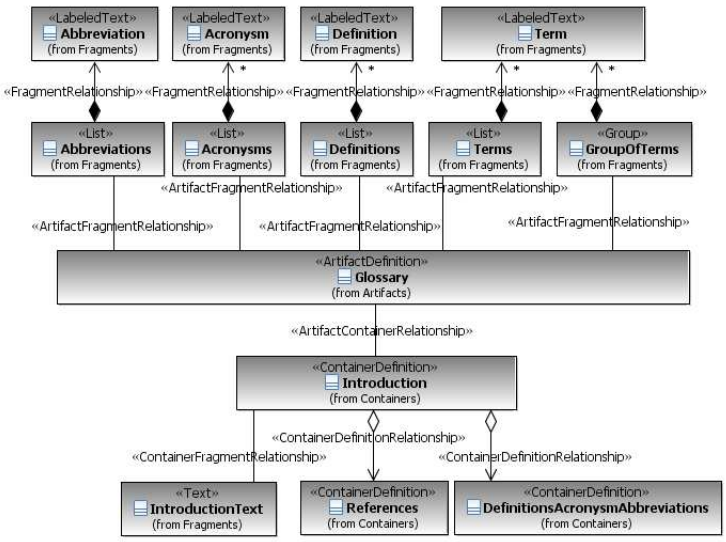

Figura C.2: *Diagrama*: Glossário

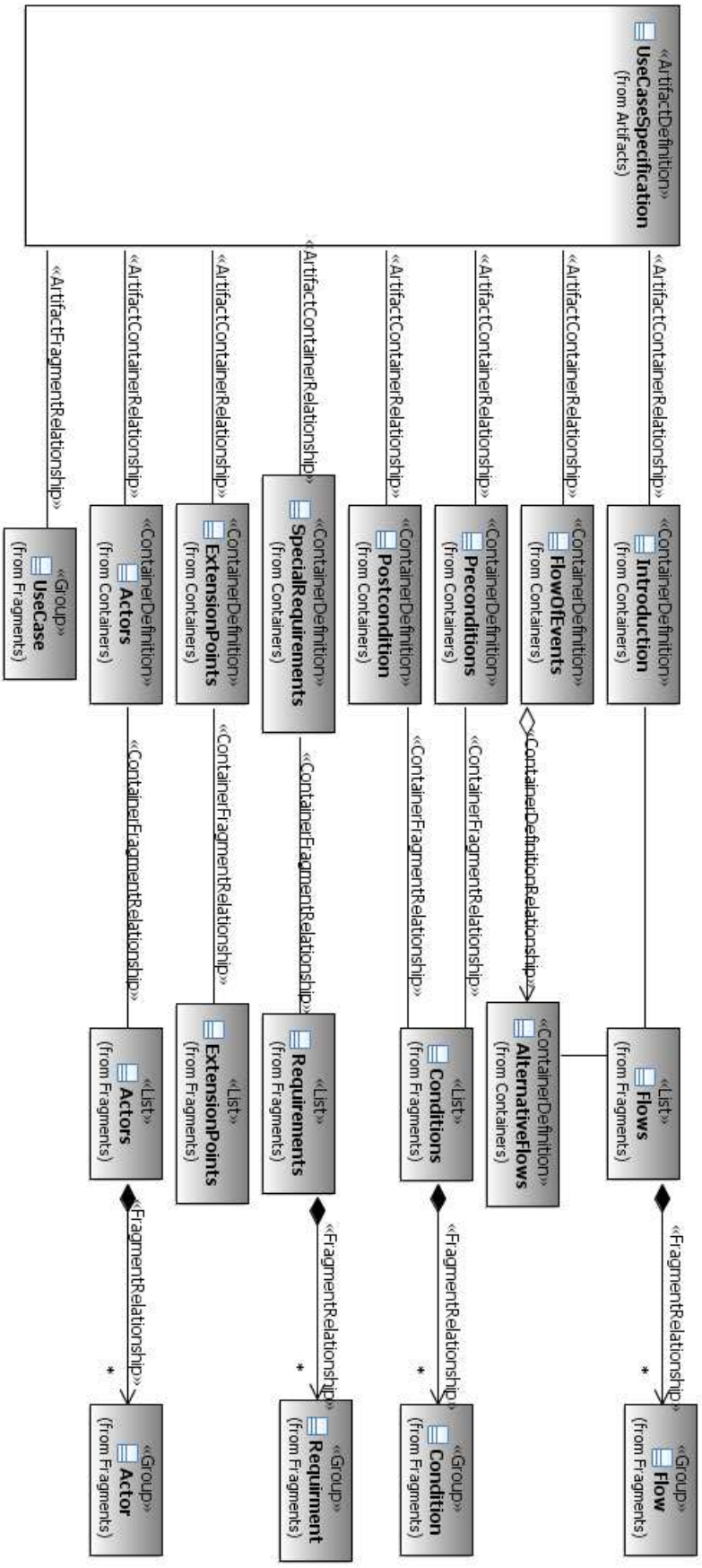

Figura C.3: *Diagrama*: Especificação de Caso de Uso

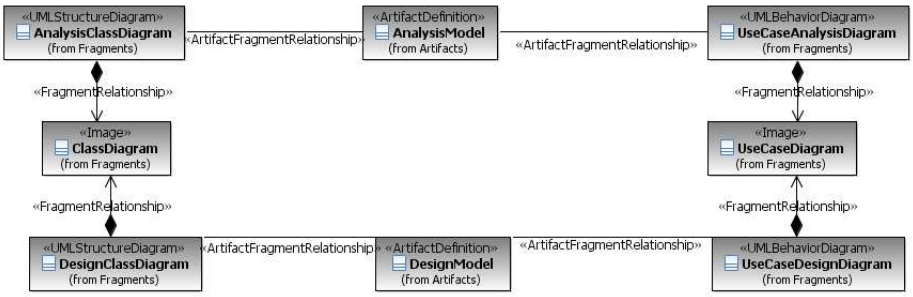

Figura C.4: *Diagrama*: Modelo de Análise e Modelo de Design

Apêndice D Histórico de Trabalhos

Durante a elaboração deste trabalho, procuramos elaborar artigos com o objetivo de amadurecer a abordagem e coletar um *feedback* da comunidade científica. Estes artigos foram submetidos para congressos nacionais e internacionais, buscando-se relatar a evolução e maturidade da abordagem. A seguir serão apresentados os artigos por ordem cronológica de submissão, tendo em vista que alguns deles foram publicados:

Artigo 1 - Março / 2008.
Evento: International Conference on Soft. Eng. and Knowledge Engineering (SEKE ’08).
Título: Towards Detailed Software Artifact Specification.
Resumo: Artigo com evoluções do artigo anterior e melhor extensão do SPEM v2.

Artigo 2 - Maio / 2008
Evento: Workshop de Teses e Dissertações em Engenharia de Software (WTES ’08)
Título: Autoria de Artefatos em Processos de Software: Uma abordagem baseada em Estruturas.
Resumo: Artigo com foco na descrição no trabalho que iria ser colocado na dissertação, apresentando uma visão geral da abordagem.

Artigo 3 - Agosto / 2008.
Evento: Simpósio Brasileiro de Sistemas Multimídia e Web (WebMedia ’08).
Título: Metamodelo Orientado a Objetos para definição de Artefatos de Software.
Resumo: Este artigo apresenta o desenvolvimento de documentação utilizando o protótipo de ferramenta SwAT conforme a abordagem de construção de artefatos com composição de informações.

Artigo 4 - Dezembro / 2008.
Evento: International Conference on Enterprise Information Systems (ICEIS ’09).
Título: Software Artifacts Specification with UML.
Resumo: Este artigo apresenta a estruturação de artefatos conforme a utilização de um metamodelo unido a um guia de etapas para a autoria de artefatos. Além disso, mostra a avaliação da abordagem com a utilização do protótipo de ferramenta SwAT.

Artigo 5 - Dezembro / 2008.
Evento: International Conference on Advanced Information Systems Engineering (CAiSE’09).
Título: Software Artifact Meta-model: An Approach to Software Artifact Authoring
Resumo: Este artigo apresenta a autoria de artefatos de software com base na extensão do metamodelo do SPEM v2, apresentando exemplos de utilização de todas as camadas de abstração da UML, ou seja, de M3 até M0.

Anexo I Metamodelo da UML

A seguir serão apresentados os principais diagramas do metamodelo da UML utilizados neste trabalho. Todos os diagramas aqui apresentados, com exceção da Figura I.1, fazem parte do pacote *Core::Constructs* da especificação da UML *Infrastructure* (OMG, 2007b), definida pela *Object Management Group* (OMG)[1].

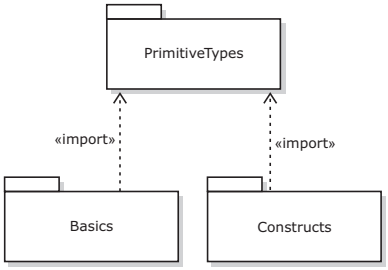

Figura I.1: Pacote *Core* da UML *Infrastructure Library*.

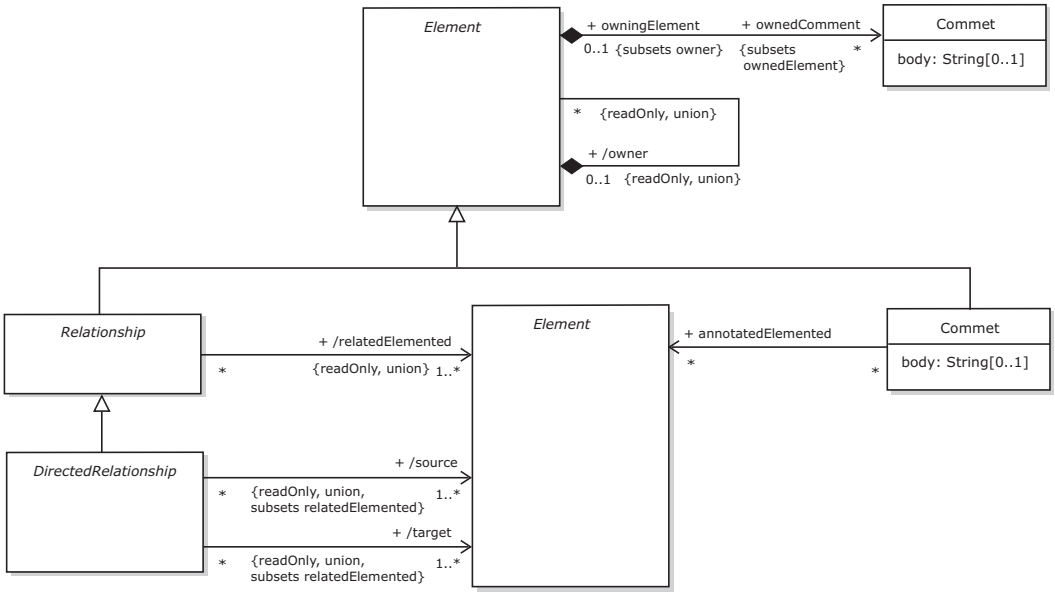

Figura I.2: Classes definidas no Diagrama Raiz.

[1] www.omg.org

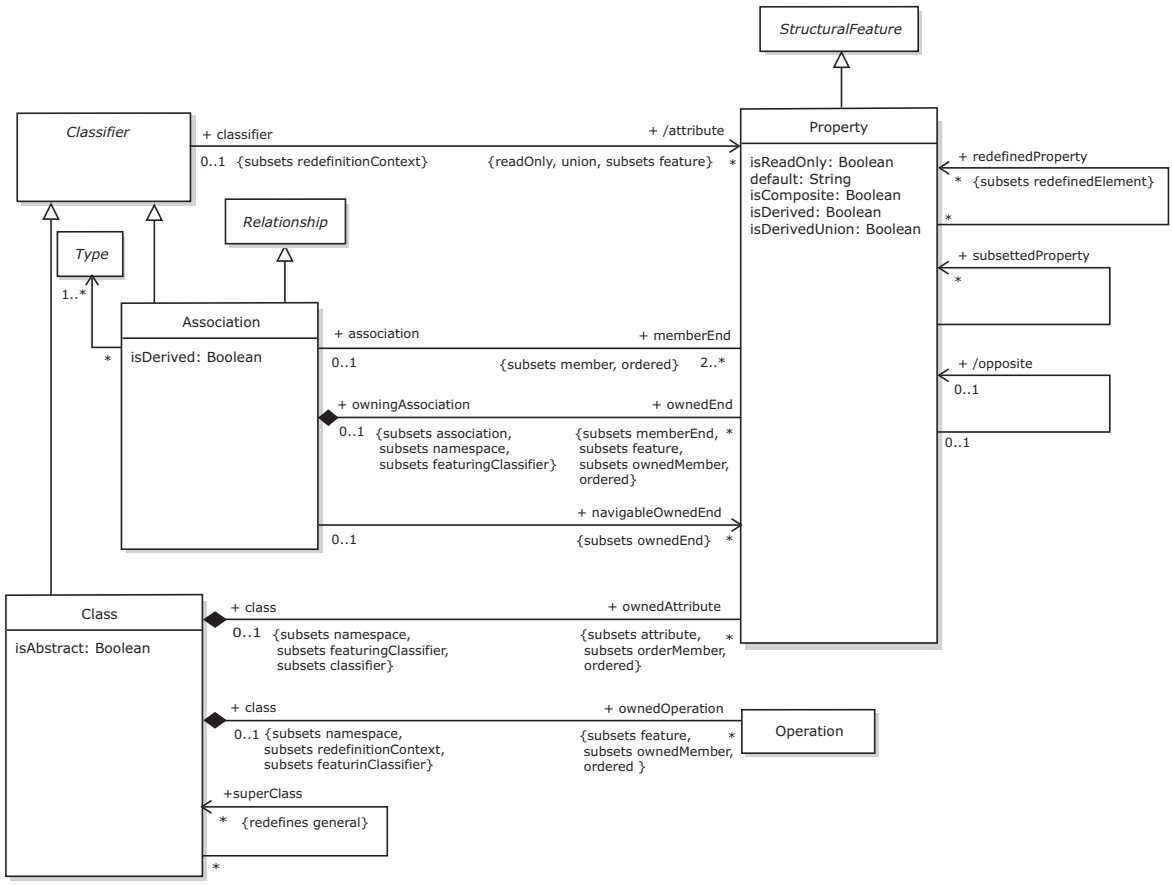

Figura I.3: Classes definidas no Diagrama de Classe..

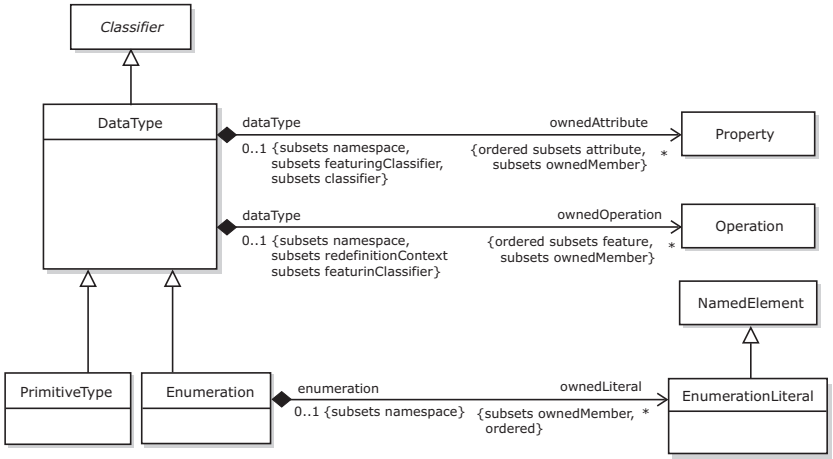

Figura I.4: Classes definidas no Diagrama de *DataTypes*.

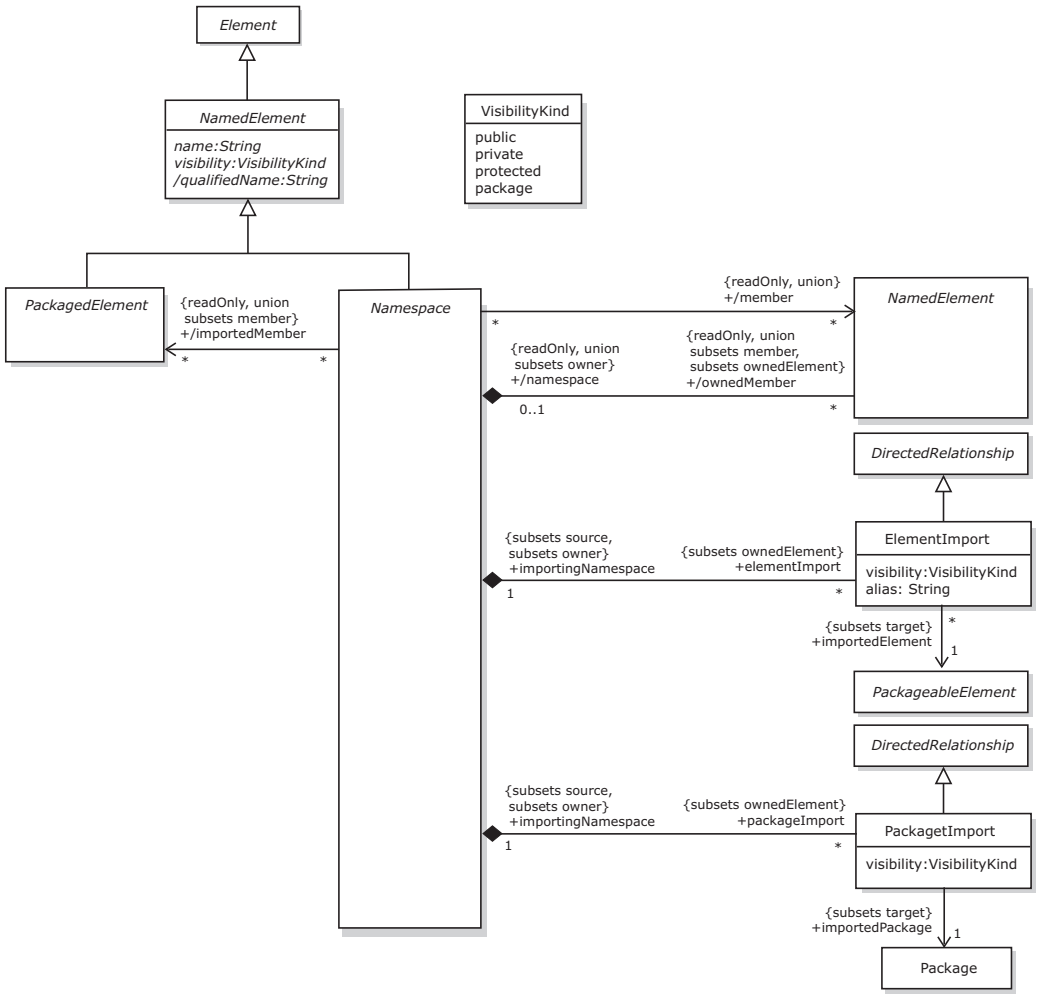

Figura I.5: Classes definidas no Diagrama de *Namespaces*.

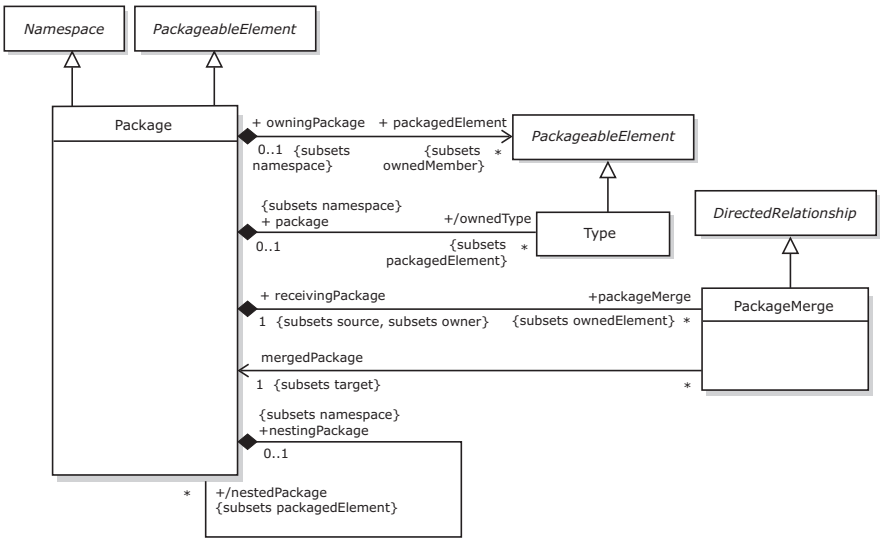

Figura I.6: Classes definidas no Diagrama de Pacote do pacote.

Anexo II Metamodelo do SPEM v2

A seguir serão apresentados os principais diagramas do SPEM v2.A seguir serão apresentados os principais diagramas do SPEM v2 utilizados neste trabalho. Todos os diagramas aqui apresentados podem se encontrados na especificação d SPEM v2 (OMG, 2008b), definida pela *Object Management Group* (OMG).

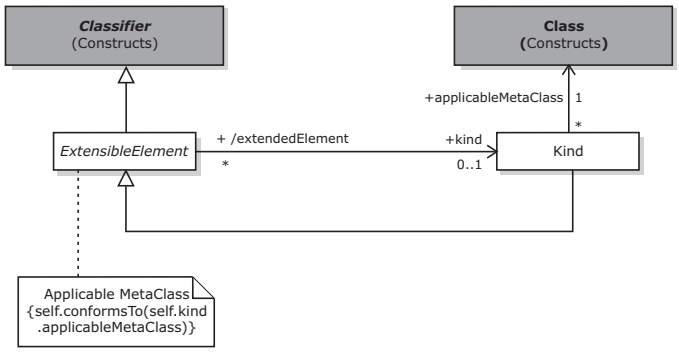

Figura II.1: Diagrama com as classes do pacote *Core* do SPEM v2.

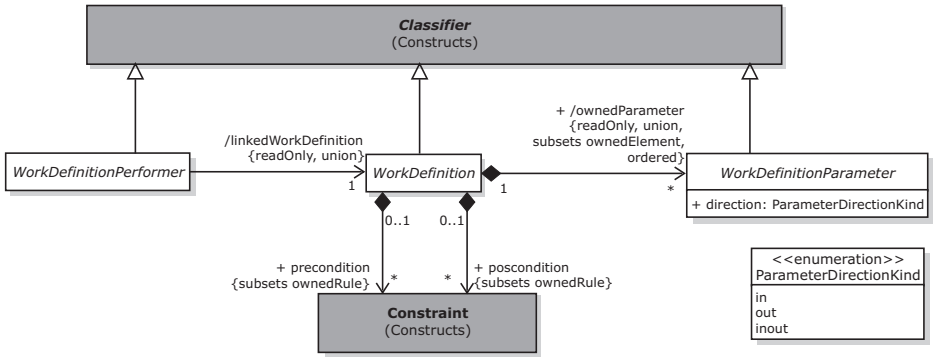

Figura II.2: Diagrama com as classes do pacote *Core* do SPEM v2.

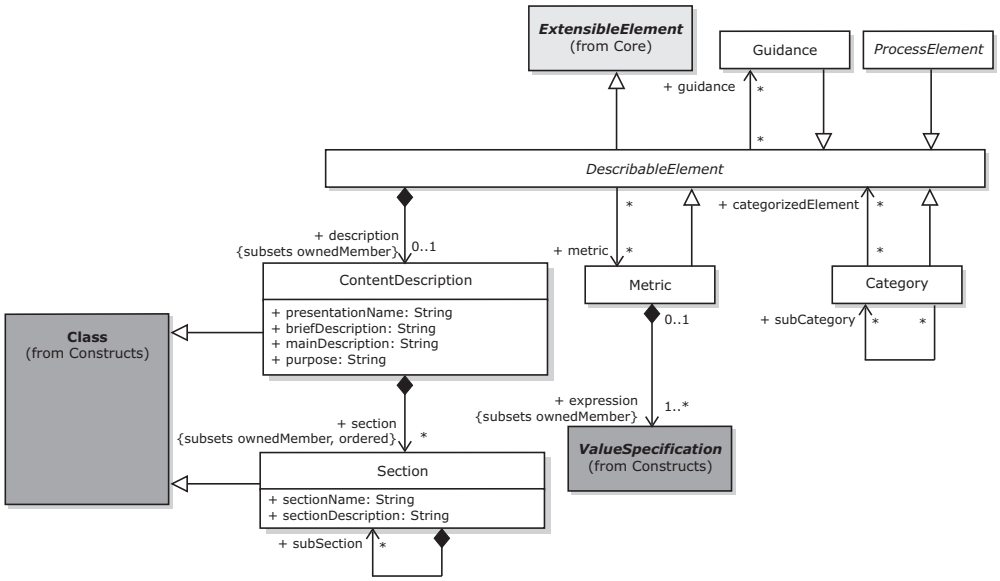

Figura II.3: Diagrama com as classes do pacote *Managed Content* do SPEM v2.

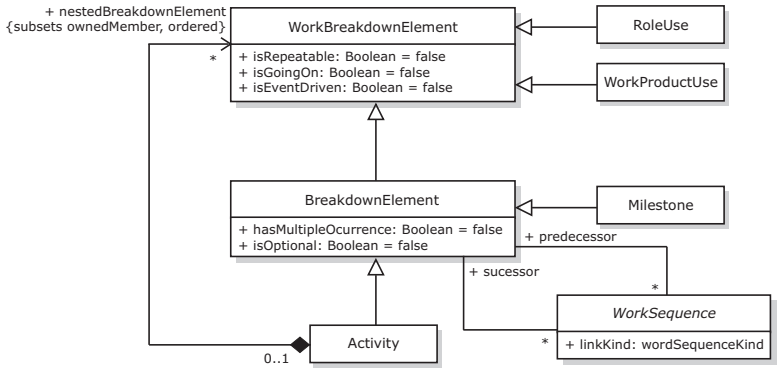

Figura II.4: Diagrama com as classes do pacote *Process Structure* do SPEM v2.

1

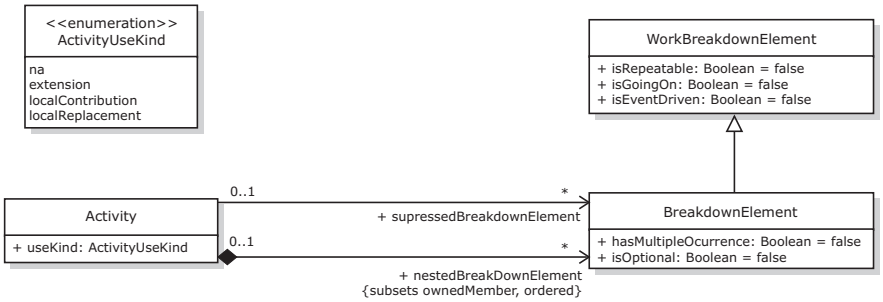

Figura II.5: Diagrama com as classes do pacote *Process Structure* do SPEM v2.

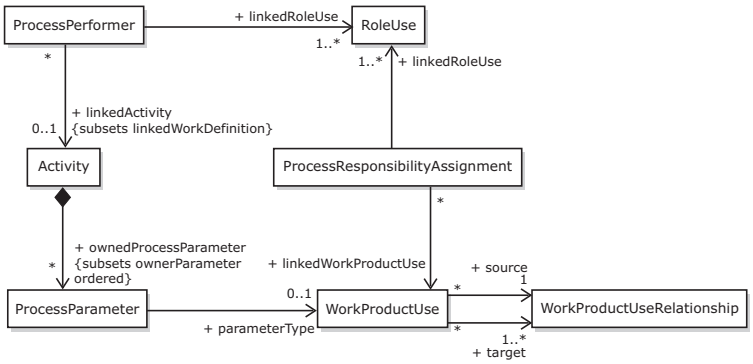

Figura II.6: Diagrama com as classes do pacote *Process Structure* do SPEM v2.

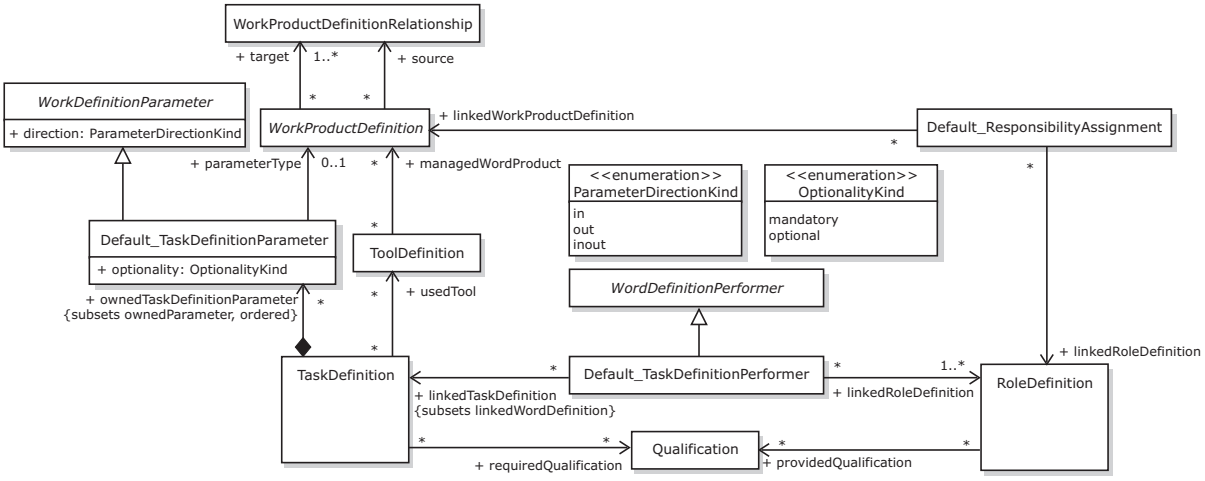

Figura II.7: Diagrama com as classes do pacote *Method Content* do SPEM v2.