ESCOLA POLITÉCNICA DA UNIVERSIDADE DE SÃO PAULO

HENRIQUE ROCHA DE FARIA

Um Modelo de Processo de Apoio ao Desenvolvimento de Software Baseado em Componentes, Orientado a Qualidade, e Centrado em um Repositório

São Paulo

2005

HENRIQUE ROCHA DE FARIA

Um Modelo de Processo de Apoio ao Desenvolvimento de Software Baseado em Componentes, Orientado a Qualidade, e Centrado em um Repositório

Dissertação apresentada à Escola Politécnica da Universidade de São Paulo para obtenção do título de Mestre em Engenharia de Computação.

Área de concentração: Sistemas Digitais
Orientador: Prof. Dr. Reginaldo Arakaki

São Paulo

2005

FICHA CATALOGRÁFICA

Faria, Henrique Rocha de
Um modelo de processo de apoio ao desenvolvimento de software baseado em componentes, orientado a qualidade, e centrado em um repositório / H.R. Faria. -- São Paulo, 2005.
193 p.

Dissertação (Mestrado) - Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia de Computação e Sistemas Digitais.

1.Engenharia de programação 2.CASE 3.Desenvolvimento baseado em componentes 4.Modelos de processo 5.Qualidade de software I.Universidade de São Paulo. Escola Politécnica. Departamento de Engenharia de Computação e Sistemas Digitais II.t.

FOLHA DE APROVAÇÃO

Henrique Rocha de Faria
Um Modelo de Processo de Apoio ao Desenvolvimento de Software Baseado em Componentes, Orientado a Qualidade, e Centrado em um Repositório

Dissertação apresentada à Escola Politécnica da Universidade de São Paulo para obtenção do título de Mestre.

Área de concentração: Sistemas Digitais

Aprovado em:

Banca Examinadora

Prof. Dr. _____

Instituição: _____ Assinatura: _____

Prof. Dr. _____

Instituição: _____ Assinatura: _____

Prof. Dr. _____

Instituição: _____ Assinatura: _____

Prof. Dr. _____

Instituição: _____ Assinatura: _____

Prof. Dr. _____

Instituição: _____ Assinatura: _____

Prof. Dr. _____

Instituição: _____ Assinatura: _____

DEDICATÓRIA

Ao eterno amigo Luciano, por ter sido o irmão que se comprometeu a ser em minha vida.

Resumo

A Engenharia de Software Baseada em Componentes (ESBC) envolve os processos de desenvolvimento de software a partir de partes embutidas prontas, a fim de se obter produtividade, reduzindo-se custos e tempo de lançamento no mercado, garantindo (e melhorando) a qualidade intrínseca de produtos de software, bem como flexibilidade de implementação, manutenção e integração de sistemas. O ciclo de vida de um componente de software, projetado para uma determinada arquitetura, para ser reutilizado e reciclado dentro de uma infra-estrutura de componentes, e para satisfazer atributos de qualidade, dependerá de um ambiente que permita que seu código evolua de maneira controlada; que suas interfaces sejam publicadas através de documentos; e que seus artefatos estejam sempre acessíveis por partes interessadas, como desenvolvedores, projetistas e arquitetos de software, gerentes de projeto, usuários etc. Isto sugere a organização de um processo que apóie a reutilização de componentes através de um repositório comum, justificando esforços de se projetar, implementar, testar e instalar estes componentes em diferentes soluções. Este trabalho tem a intenção de definir e descrever, através da linguagem e dos elementos de um meta-modelo, e através de uma proposta de implementação de um repositório de componentes, um modelo de processo alinhado a um subconjunto de requisitos estabelecidos pelos padrões ISO/IEC 12207 e ISO/IEC 9126, com o propósito de suporte de componentes a processos de desenvolvimento de software.

Abstract

Component-Based Software Engineering (CBSE) involves the software development from prepared built-in parts processes, in order to achieve productivity, reducing costs and time-to-market, assuring (and improving) the intrinsic quality of software products, as well as implementation, maintenance and systems integration flexibility. The life cycle of a software component designed for a given architecture to be reused and recycled, within a component infrastructure, and to satisfy quality attributes, will depend on an environment to allow its code to evolve in a controlled manner; its interfaces to be published through documents; and its artifacts to be always accessible from interested parties, like developers, software designers and architects, project managers, users etc. This suggests the organization of a process that supports the reuse of components through a common repository, justifying efforts to design, implement, test and install them in different solutions. This work intends to define and describe, through a meta-model language and elements, and through a component repository implementation proposal, a process model aligned to a subset of requirements established by the ISO/IEC 12207 and the ISO/IEC 9126 standards, with the purpose of development software processes support of components.

LISTA DE SIGLAS

API	- *Application Programmer Interface*
ATAM	- *Architecture Tradeoff Analysis Method*
CASE	- *Computer-Aided Software Engineering*
COTS	- *Commercial Off-The-Shelf*
CNS	- Cartão Nacional da Saúde
DBC	- Desenvolvimento Baseado em Componentes
DES	- *Data Encryption Standard*
DOM	- *Document Object Model*
DTD	- *Document Type Definition*
EIS	- *Enterprise Information Service*
EJB	- *Enterprise Java Beans*
ERP	- *Enterprise Resource Planning*
GQM	- *Goal/Question/Metric*
IDE	- *Integrated Development Environment*
IDEF	- *Integrated Definition*
J2EE	- *Java 2 Entreprise Edition*
JAXB	- *Java Architecture for XML Binding*
JDBC	- *Java DataBase Connectivity*
MOF	- *Meta Object Facility*
ODMG	- *Object Data Management Group*
OJB	- *Object Relational Bridge*
OMG	- *Object Management Group*

OO - Orientação a Objetos

OOSPICE - *Object-Oriented Software Process Improvement and Capability dEtermination*

OPEN - *Object-oriented Process, Environment and Notation*

OPF - *Open Process Framework*

OTM - *Object Transaction Manager*

PB - *Persistent Broker*

PDA - *Personal Digital Assistant*

RAS - *Reusable Asset Specification*

RM-ODP - *Reference Model - Open Distributed Processing*

RUP - *Rational Unified Process*

SAX - *Simple API for XML*

SCL - *Software Certification Laboratory*

SOAP - *Simple Object Access Protocol*

SPEM - *Software Process Engineering Metamodel*

SUS - Sistema Único de Saúde

TAS - Terminal de Atendimento do SUS

TI - *Tecnologia da Informação*

UML - *Unified Modeling Language*

XMI - *XML Metadata Interchange*

XML - *eXtensible Markup Language*

SUMÁRIO

1. Introdução

Este capítulo apresenta uma visão geral do estado do Desenvolvimento Baseado em Componentes (DBC) no mundo; relaciona as principais motivações por trás do trabalho, justificando os objetivos pretendidos pelo mesmo; e descreve a metodologia utilizada para o atendimento dos objetivos propostos.

1.1. ESBC no mundo

Segundo um histórico apresentado por Rossi (ROSSI, 2004) a reutilização de software existe desde que a programação foi inventada. Na década de 50, o uso de bibliotecas de sub-rotinas já era praticado na Universidade de Cambridge. Na década de 60, surgiu a primeira proposta de uma indústria de componentes de prateleira – hoje mais conhecidos como *Commercial Off-The-Shelf* (COTS) – em uma conferência da Organização do Tratado do Atlântico Norte. Na década de 70, começaram a ocorrer as primeiras iniciativas na direção do DBC.

O paradigma de componentes começou a ganhar força, entretanto, a partir do grande crescimento sofrido pela indústria de software nas duas últimas décadas, bem como das transformações ocorridas no mercado. A figura 1.1 ilustra o volume de vendas anual de softwares e serviços informatizados, durante as três últimas décadas, no Japão.

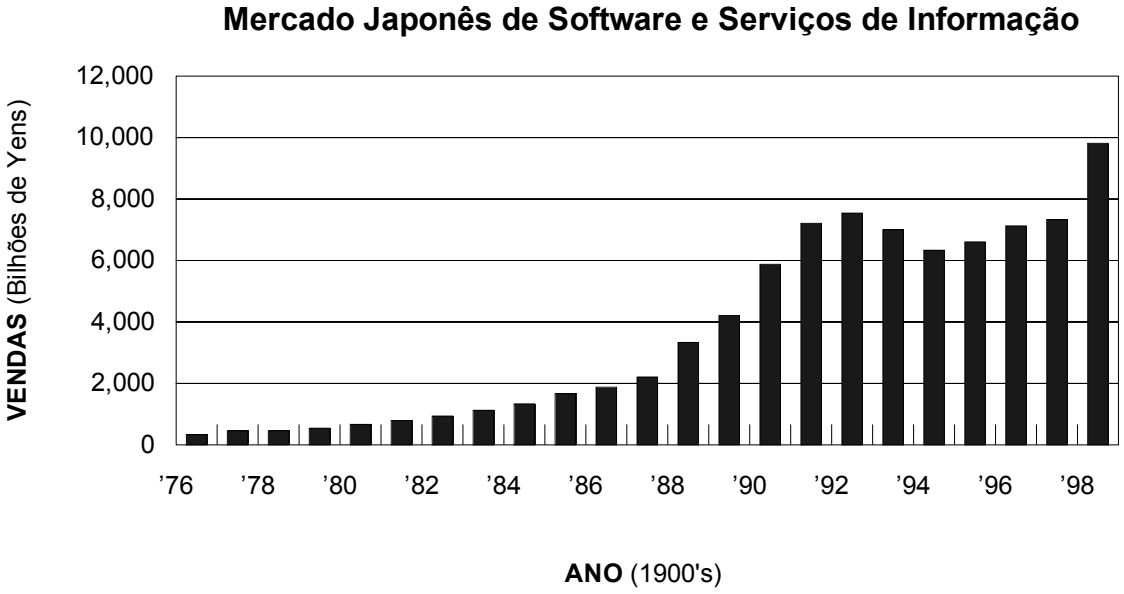


Figura 1.1. Mercado japonês de software e serviços de informação (AOYAMA, 2001).

Das transformações mencionadas no parágrafo anterior, ainda no Japão, podem-se citar duas (AOYAMA, 2001):

- *Na estrutura do mercado de software* – De sistemas personalizados (encomendados por grandes empresas e organizações) para pacotes e componentes de software.
- *Na estrutura da indústria de software* – De uma estrutura vertical e hierárquica de software para um cenário mais aberto e competitivo.

Neste último caso, o estabelecimento de padrões – seja por iniciativa de grandes empresas, consórcios ou grupos – exerce importante papel para que haja maior competitividade, bem como maior integração, entre fabricantes, e mais liberdade de escolha para todos os tipos de usuários de software. A arquitetura Java 2 Platform, Entreprise Edition (J2EE), por exemplo, proporciona uma plataforma padrão para a construção de componentes portáveis através de diferentes implementações, evitando-se a dependência exclusiva de um fornecedor; e diminui o tempo necessário para o lançamento de um produto no mercado, já que grande parte da

infra-estrutura provem de produtos implementados por fornecedores de acordo com a especificação J2EE (ALUR, 2002).

Ainda em relação à ESBC, Brown e Wallnau (BROWN, 1998) indicam fatores que levaram o desenvolvimento de software a caminhar, nos últimos anos, no sentido de processos e metodologias baseadas em componentes:

- Aplicações centralizadas baseadas em *mainframes*, sobre redes proprietárias, deram lugar a aplicações, compostas por inúmeras camadas, acessíveis por diferentes clientes, sobre redes heterogêneas. Tal transição acabou por exigir novas abordagens de desenvolvimento de software.

- O investimento de recursos financeiros e intelectuais, nas duas últimas décadas, que levaram à construção de inúmeros sistemas, em funcionamento até os dias de hoje, obrigou empresas a lidarem com a reutilização de seus legados para a implementação de novas soluções para seus negócios.

- Organizações, que antes eram responsáveis por todo o desenvolvimento de seus sistemas informatizados, ou que dependiam de um único produto de um fornecedor específico, começaram a adotar estratégias, que evitassem a construção de sistemas, a partir do zero, flexíveis o suficiente para evoluírem de acordo com as mudanças de requisitos.

- Finalmente, uma condição necessária para uma organização ser bem sucedida é a existência de certo grau de estabilidade e previsibilidade no suporte tecnológico de seus negócios, acompanhando mudanças no mercado e em seus processos internos.

Sobre o estado do DBC na Europa, McGibbon relata a migração (inevitável segundo o autor) para componentes, por parte dos principais grupos desenvolvedores europeus, a fim de que menores custos, menor tempo de lançamento de produtos no mercado, e serviços mais completos para o cliente possam ser obtidos (MCGIBBON, 2001).

No Brasil, o Ministério da Ciência e Tecnologia está investindo 1,8 milhões de reais no projeto da primeira biblioteca pública de componentes de software da América Latina, desenvolvido por empresas e instituições de pesquisa, com o objetivo de aperfeiçoarem-se gastos públicos, evitando-se investimentos redundantes (FUCAPI, 2004).

1.2. Motivações

Por trás das motivações da utilização de componentes, Boehm realiza uma análise econômica de três estratégias para o aumento da produtividade de software (BOEHM, 1999): (1) trabalhar mais rapidamente, (2) trabalhar mais inteligentemente, (3) evitar trabalho desnecessário através do reuso. Os resultados desta pesquisa, para o Departamento de Defesa dos EUA, indicaram uma taxa de redução de custos de 8% adotando-se a estratégia (1); de 17% adotando-se a (2); e de 47%, a (3), sugerindo que o projeto de componentes reutilizáveis de software é o meio mais eficaz para aumento de produtividade (produzir mais rapidamente, com menores custos).

Fez parte dos estudos desta dissertação a coleta de dados de uma empresa que teve uma experiência com o DBC. A *Singularity Systems*, em parceria com a *Hypercom do Brasil*, participou do desenvolvimento do projeto piloto do Cartão Nacional da Saúde (CNS) do Sistema Único de Saúde (SUS) para o Ministério da Saúde. Desenvolveu uma rede de servidores transacionais, para o tráfego de informações médicas, via Internet, através de 44 municípios espalhados por 11 estados brasileiros, com o objetivo de validar a arquitetura de hardware e os componentes de software idealizados para o atendimento de cerca de 13 milhões de pacientes do SUS. A rede (figura 1.2), construída sob uma estrutura hierárquica de servidores (explicada mais detalhadamente no Apêndice A), teve cada nó implementado como um núcleo computacional, constituído por componentes de negócio e infra-estrutura (LEMOS, 2003).

O sucesso do sistema CNS – através de uma solução baseada em componentes, quanto ao atendimento de requisitos não funcionais, como escalabilidade, disponibilidade, segurança, e manutenibilidade – acabou sendo, então, uma grande motivação para o emprego dos mesmos componentes de software, em outros sistemas, não apenas por representarem uma economia na reutilização de códigos e padrões, mas também por possuírem qualidade intrínseca comprovada.

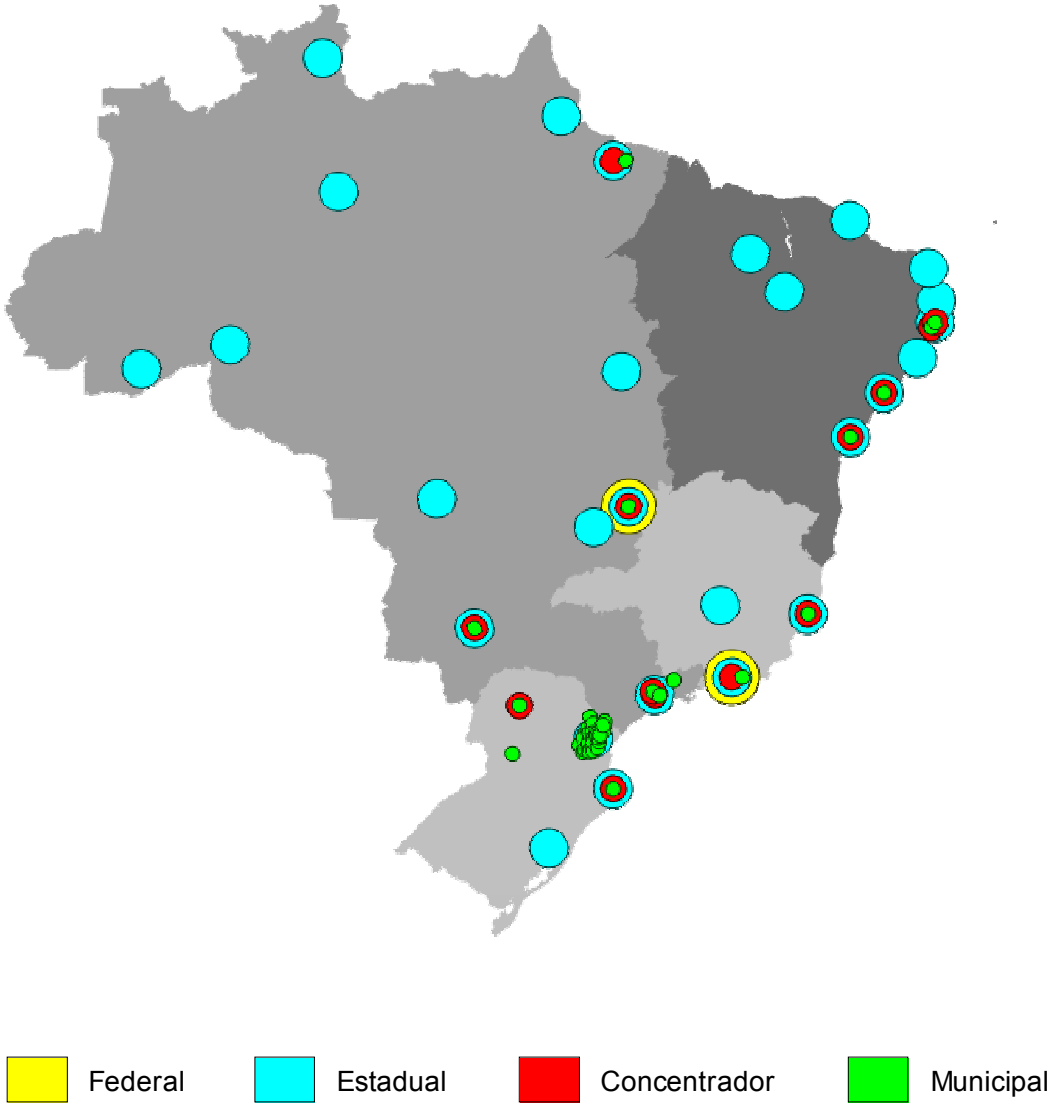


Figura 1.2. Rede de servidores transacionais do projeto CNS (LEMOS, 2003)[1].

O primeiro projeto, por exemplo, a reutilizar (e remodelar) os componentes de software do legado CNS provou o grande potencial de reutilização de código através do DBC. A figura 1.3 apresenta a quantidade de código fonte (em KB) reutilizada de componentes em relação ao código efetivamente implementado para o projeto de um servidor transacional (denominado *Ignition*) gerido pela *Singularity Systems*. Muitos dos componentes projetados e implementados internamente para o Cartão Nacional da Saúde vieram a se tornar padrões de

[1] A figura mostra a rede de servidores transacionais do projeto piloto de larga escala do Cartão Nacional da Saúde, espalhados por todo o território nacional. Os nós estão dispostos numa estrutura hierárquica de processamento de transações em quatro níveis: federal, estadual, concentrador, e municipal.

mercado (como os apresentados na figura 1.3): OJB (APACHE, 2005a), JAXB (SUN, 2005a), Tomcat (APACHE, 2005b), Log4J (APACHE, 2005c).

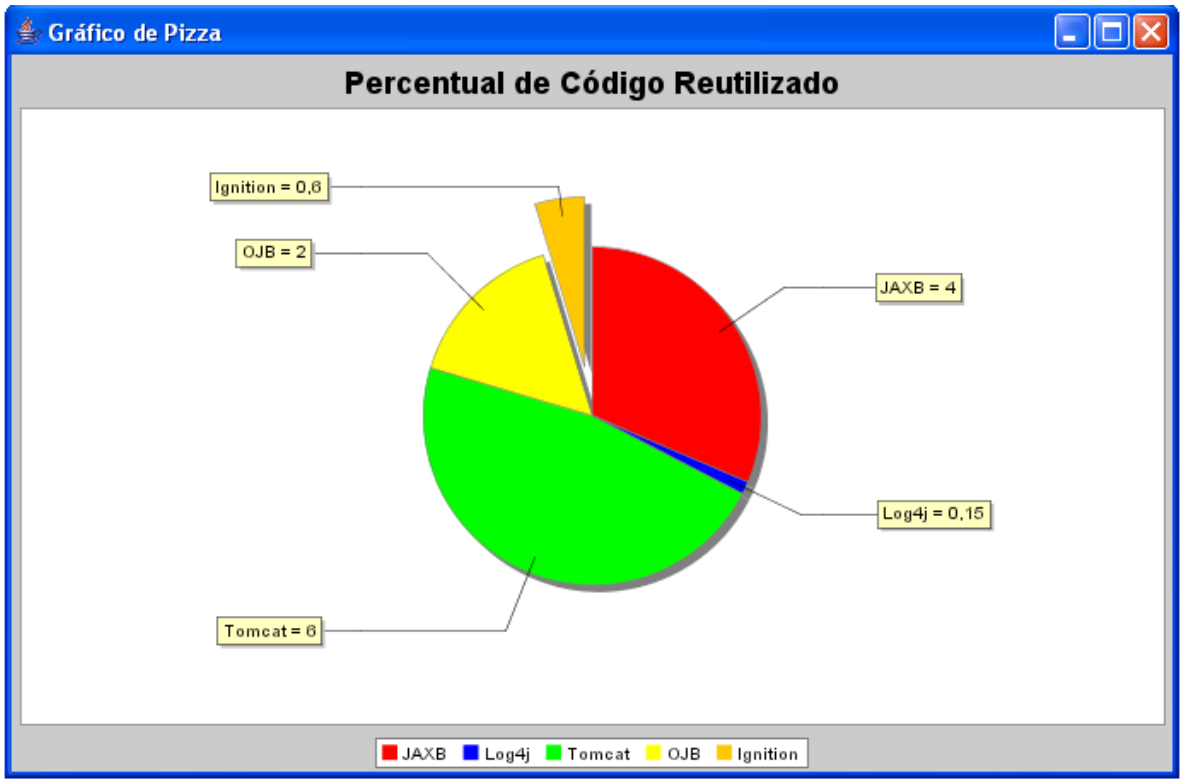


Figura 1.3. Distribuição de código reutilizado no servidor transacional Ignition, em KBs[2].

O reemprego dos componentes de software do legado CNS em outras soluções de Tecnologia Informação (TI) veio mostrar, na prática, a necessidade de um repositório de fácil acesso, para o reaproveitamento de todo tipo de artefato (especificações, arquiteturas, documento de requisitos, componentes de teste, e logicamente código fonte).

No DBC não basta, entretanto, apenas se ter um legado de componentes; deve haver um mínimo de organização na utilização deste legado. Muitos trabalhos enfatizam a importância

[2] COTS (Commercial Off-The-Shelf) foram intensamente empregados no projeto Ignition. A fatia do gráfico destacada na figura representa a quantidade de código efetivamente implementada pela equipe de desenvolvimento, retratando o reaproveitamento de software.

do estabelecimento de processos orientados ao DBC, para o sucesso de programas de reuso. De acordo com Apperly, “um dos caminhos primários para possibilitar um ciclo de vida de desenvolvimento de componentes de sucesso é implementar um processo, que inclua um ambiente integrado de gerência de configuração e biblioteca de componentes” (APPERLY, 2001). Apperly sugere a criação de não apenas um repositório, como também um processo que defina e descreva sua utilização.

Em um nível mais administrativo, pressões externas, como prazos curtos, planejamento equivocado, mau gerenciamento de projeto etc., frequentemente tornam difícil o projeto de componentes visando qualidade. A maioria dos processos existentes nas organizações, atualmente, ainda não se encontra completamente adaptada para trabalhar com o paradigma de componentes, e ao mesmo tempo há a necessidade de um conjunto comum de ferramentas, que permitam fácil comunicação de modelos, especificações, e código entre os atores de um processo (MCGIBBON, 2001).

Woodman e outros (WOODMAN, 2001) argumentam a importância das propriedades de um processo de desenvolvimento de software baseado em componentes, na previsibilidade das propriedades dos produtos deste processo. Tentam mapear, a partir de atributos pré-estabelecidos de qualidade (muitos deles em conformidade com a norma ISO/IEC 9126), como as propriedades de um componente refletem as propriedades do produto que vai compor.

Existe atualmente uma série de modelos de processo que já dão suporte ao DBC. O OOSPICE (STALLINGER, 2002), por exemplo, define um meta-modelo do qual derivam componentes de processo, que serão utilizados na definição de um processo orientado ao

DBC. Atende um conjunto de áreas da norma ISO/IEC 15504 e prevê ainda um modelo de avaliação da eficácia do processo. O *Rational Unified Process* (RUP) (KRUCHTEN, 2000) e o *OPEN Process Framework* (OPF) (FIRESMITH HENDERSON-SELLERS, 2002) são modelos orientados a objeto, que definem um vasto conjunto de elementos de processo, como fluxos de trabalho, atividades, tarefas, técnicas, diretivas, produtos, atores etc., com áreas específicas para o DBC. Cheesman e Daniels (CHEESMAN, 2001), através de um modelo mais simples, definem um processo, focado na especificação e montagem de componentes, totalmente baseado em UML, utilizando largamente diagramas de classes, casos de uso, colaboração, seqüência, componentes etc., como ferramentas de representação de cada uma das etapas de um fluxo de atividades.

Nenhum destes modelos, no entanto, define regras específicas para um processo de reutilização de componentes (seja de um legado, através de COTS, de componentes fabricados internamente ou terceirizados), tampouco uma infra-estrutura de armazenamento, para publicação, busca, e recuperação de componentes.

De acordo com Guo, em seu estudo sobre repositórios de componentes (GUO, 2000), deve haver uma transição das práticas correntes de Engenharia de Software, em que o reuso é feito de forma oportuna e aleatória, para um processo que institucionalize e torne a reutilização de componentes parte inseparável do desenvolvimento de software.

Na pesquisa realizada por Morisio, Tully e Ezran (MORISIO, 2002), foram analisadas 24 empresas européias que tiveram uma experiência na implantação de um programa de reutilização de software. Um terço aproximadamente destas iniciativas falhou, por não terem sido adotados processos específicos ao reuso, ou pelo fato dos processos convencionais, até

então em vigor, não terem sido adaptados aos programas de reutilização de componentes (além de fatores humanos responsáveis pelos fracassos). A crença de que o paradigma OO e a configuração aleatória de um repositório sejam suficientes, para o sucesso de um programa de reuso, é apontado por Morisio como um dos erros mais comuns cometidos por gerentes e líderes de projeto. É necessário, então, haver um processo regendo a reutilização de componentes via um repositório.

1.3. Objetivos

Sendo o desenvolvimento baseado em componentes uma real tendência verificada na indústria de software, atualmente, o objetivo deste trabalho é propor um modelo de processo de apoio do DBC, no que diz respeito à utilização e ao gerenciamento de um repositório de componentes de software.

Ao mesmo tempo em que os processos já existentes de suporte ao DBC (OOSPICE, OPF, Cheesman) carecem de um modelo que organize a reutilização de componentes de um repositório, os modelos e implementações de repositórios de software, como os relacionados no trabalho de Guo (GUO, 2000), por exemplo, também não prevêem processos sistemáticos de consumo, busca, e publicação de componentes. Dentro deste contexto, os principais objetivos pretendidos com esta dissertação são:

- Definir um modelo de processo – denominado *Komponento* – de apoio ao DBC na utilização e gerenciamento de um repositório de componentes de software, através da descrição de todos os elementos deste modelo, que atenda a um subconjunto de

diretivas estabelecidas pela norma ISO/IEC 12207, no que diz respeito à gerência de configuração, gerência de ativos e garantia de qualidade. O modelo deve ser concebido de maneira a ser adaptável a outros processos, uma vez que o objetivo é dar suporte ao desenvolvimento, promovendo um ambiente integrado de repositório de componentes e ferramentas CASE.

- Desenvolver um protótipo de repositório baseado na especificação de ativos reutilizáveis de software (RAS – *Reusable Asset Specification*) da OMG, representando os modelos UML de armazenamento de ativos, na forma de esquemas XML; e implementando o serviço de consulta a um repositório, como um conjunto de classes Java implemetando interfaces previamente especificadas (OMG, 2004).

- Uma estratégia de garantia de qualidade deve ser abordada pelo modelo de processo, uma vez que componentes devem de alguma forma atender a requisitos de qualidade que justifiquem sua reutilização. O ambiente constituído pelo repositório de componentes e pelas atividades de utilização deste repositório deve ser a espinha dorsal para a definição das estratégias de avaliação de qualidade.

1.4. Metodologia

A pesquisa bibliográfica, relacionada no capítulo 6, com ênfase em processos, componentes, normas e padrões, constituiu a base conceitual da pesquisa. A experiência com o projeto de informatização do SUS, em relação à reutilização dos componentes do legado CNS, permitiu o levantamento de problemas e soluções práticas, que puderam ser agregados ao trabalho. O

Apêndice A entra em maiores detalhes a respeito da arquitetura do sistema desenvolvido para o Ministério da Saúde.

A metodologia utilizada para a definição do processo proposto foi o levantamento dos requisitos deste processo com base na norma ISO/IEC 12207, no que diz respeito às áreas de gerência de configuração, garantia de qualidade, e gerência de ativos. Foi realizado, então, um mapeamento das diretivas da norma em propriedades implementadas pelo processo. O processo foi representado através do meta-modelo SPEM, visando-se compatibilidade com outros sistemas baseados no mesmo modelo aberto de definição de processos da OMG.

Foi desenvolvido, finalmente, um pequeno protótipo de um repositório de componentes de software, com base nos modelos de armazenamento e entrega de serviços, propostos pela especificação de ativos reutilizáveis RAS, também da OMG. Representando os modelos UML em esquemas XML, o protótipo visou atender os requisitos mínimos de compatibilidade estabelecidos pelo RAS e, através da herança de classes (interfaces Java) implementou um serviço de consulta de ativos também compatível com a especificação (OMG, 2004).

2. Fundamento conceitual

Sendo o *Komponento* um modelo de processo: (1) totalmente baseado em componentes (através da manutenção de um repositório de componentes), (2) orientado à entrega de qualidade, e (3) de suporte a outros processos (CBD ou não), e por isso representado através de um meta-modelo de processos, com os objetivos e características já definidas no capítulo 1, considerou-se relevante o aprofundamento dos conceitos a seguir.

2.1. Componentes

O termo "componente", infelizmente, tornou-se um dos termos mais indiscriminadamente utilizados e mal entendidos da indústria de software (VITHARANA, 2003). Tem sido empregado para referenciar uma série de conceitos que se sobrepõem, mas que diferem entre si, variando desde algumas linhas de código ou simples *widgets*[3], por exemplo, a complexos módulos e subsistemas de sistemas maiores.

Não existe, na verdade, uma definição absoluta do significado do termo "componente" (mais especificamente "componente de software"). As definições variam entre diferentes contextos: arquiteturas, tecnologias, domínios etc. O desenvolvimento baseado em componentes (CBD – *Componente Based Development*) tornou-se um novo paradigma dentro da Engenharia de Software (principalmente, após a larga difusão da abordagem orientada a objetos, contemporânea e ainda amplamente empregada).

[3] *Widgets* são elementos gráficos (ex. botões, campos de texto, listas) de uma interface, através dos quais o usuário interage com a aplicação.

Por trás da utilização de componentes, motivações mais do que justificáveis: aumento de produtividade, redução de custos e ganhos em qualidade. Segundo Guo (GUO, 2000), bibliotecas de reuso tem o propósito de auxiliar a reutilização de componentes a alcançar metas de melhor gerenciamento de custos e produtividade. Os fatores que levaram as empresas, que fizeram parte do estudo de Morisio, Tully e Ezran (MORISIO, 2000), a selecionarem estratégias de reuso (de componentes) foram: melhoria na produtividade de software, menor tempo de lançamento de produtos no mercado, e manutenibilidade. Stalinger e outros (STALLINGER, 2002) colocam a Engenharia de Software Baseada em Componentes, como um paradigma emergente de desenvolvimento, objetivando metas relacionadas à melhoria de qualidade, menor tempo de lançamento de produtos e menores custos.

Projetar componentes envolve, muitas vezes, a análise de requisitos vagos, por serem concebidos para darem suporte a diferentes soluções, de forma relativamente genérica. Devem prestar seus serviços dentro de protocolos pré-estabelecidos, e através de interfaces bem definidas. Devem oferecer funcionalidades dentro de um nível mínimo de qualidade, estando em conformidade com suas especificações e atendendo a requisitos de desempenho.

As definições, como já mencionado, variam muito entre diferentes contextos. Councill e Heineman colocam que a definição de componente de software envolve três conceitos, intimamente, relacionados (COUNCILL; HEINEMAN, 2001):

"Um *componente de software* é um elemento de software que esteja de acordo com um modelo de componentes e que possa ser, independentemente, instalado e composto sem modificações de acordo com um padrão de composição."

> Um *modelo de componentes* define padrões específicos de interação e composição. Uma implementação de modelo de componentes é o conjunto dedicado de elementos executáveis de software, requeridos para darem suporte à execução de componentes que estejam em conformidade com o modelo.
>
> Uma *infra-estrutura de componentes de software* é um conjunto de componentes de software que interagem, projetado para garantir que um sistema de software ou subsistema, construído usando estes componentes e interfaces, satisfará especificações de desempenho claramente definidas.

A definição do termo "container" se encaixa em uma infra-estrutura de componentes de software (CONAN, 2001): "uma entidade responsável por gerenciar propriedades não funcionais em benefício de um componente de software". O desenvolvedor escreve apenas o núcleo da lógica de negócio, e a infra-estrutura de componentes provê um meio de declarar o comportamento do componente. Em execução, o container deve gerenciar todos os requisitos não funcionais. Assim, o container é uma entidade visível tanto durante o desenvolvimento de componentes de software, quanto durante a execução. Nas fases de análise de arquitetura e projeto de software, o container orienta o desenvolvedor, o arquiteto ou projetista a separarem as partes técnicas e de negócio em diferentes visões; durante a execução, é responsável por gerenciar o acesso de componentes a serviços da infra-estrutura externa.

Um bom exemplo de container de componentes é a arquitetura *Entreprise Java Beans* (EJB), parte da plataforma J2EE, que proporciona um padrão para o desenvolvimento de componentes Java reutilizáveis, que são executados em um servidor de aplicações. O container EJB fornece então capacidades de processamento distribuído, consistência,

processamento de negócios e processamento de transações a aplicações empresariais. Alur relaciona alguns padrões alinhados à arquitetura EJB, caracterizando o papel do container nas fases de análise e projeto de sistemas (ALUR, 2002).

De acordo com Szyperski (SZYPERSKI, 1998),

"Um componente de software é uma unidade de composição com interfaces especificadas contratualmente e dependências de contexto explícitas somente. Um componente de software pode ser instalado, independentemente, e está sujeito a composição por terceiros."

Para uma organização, uma arquitetura representa um significante investimento de tempo e esforço de seus melhores (e mais caros) engenheiros. É natural que se queira, então, maximizar o retorno deste investimenro reutilizando-se uma mesma arquitetura para diferentes soluções, porém, com muitas características em comum (BASS, 1999). Uma linha de produção de software pode ser entendida de acordo com a definição abaixo (CLEMENTS; NORTHROP, 2002):

> Uma *linha de produção de software* é um conjunto de sistemas intensamente baseados em software, compartilhando um conjunto comum, gerenciado de características que satisfazem as necessidades específicas de um segmento de mercado ou missão particular, e que são desenvolvidos a partir de um conjunto comum de componentes base de forma predefinida.

Neste caso, componentes-base (*core assets*) seriam componentes dedicados a um tipo ou família específica de software. Este tipo de iniciativa (de se implantar uma linha de produção de software), embora mais raro na indústria, atualmente, pode também encontrar suporte no processo apresentado neste trabalho. Jaaksi (JAAKSI, 2002) descreve como a empresa Nokia

criou uma linha de produção para desenvolver navegadores móveis (*browsers* para plataformas móveis, como telefones celulares e PDAs).

O *OPEN Process Framework* (OPF) define, por sua vez, um componente de software como sendo (FIRESMITH; HENDERSON-SELLERS, 2002)

> [...] um componente consistindo totalmente de software. Um componente deve ser coeso, no sentido de ter um conjunto bem definido de responsabilidades relacionadas; deve ser acessível via um conjunto bem definido de interfaces, que permitam que seja substituído, dentro de um contexto de uma arquitetura bem definida.

Alguns exemplos citados ainda, na documentação dos elementos do OPF, de componentes de software:

- Softwares como produtos lançados no mercado;
- *Frameworks*[4];
- Softwares de teste;
- Pacotes;
- Classes;
- *Stored Procedures*[5];
- *Web Scripts*[6].

[4] Segundo (FIRESMITH; HENDERSON-SELLERS, 2002), um *framework* seria uma biblioteca reutilizável de classes colaborativas.

[5] Funções compiladas em servidores de banco de dados para automatização tarefas.

[6] Programas que passam parâmetros através de requisições HTTP, originadas por um cliente, para servidores *web*, onde são executados, e que retornam os resultados em resposta às requisições.

Já o RUP (KRUCHTEN, 2000) inclui na definição de componente mais uma característica importante, em sua definição de componente: a de ser substituível. Colocando assim que:

"Um componente é uma parte não trivial, quase independente, e substituível de um sistema, e que desempenha uma função clara no contexto de uma arquitetura bem definida. Um componente obedece e provê a realização de um conjunto de interfaces."

Algumas características se repetem nas definições dadas ao longo desta seção. Fazendo-se uma varredura, então, dos significados bastante semelhantes, apresentados acima, podem-se extrair as seguintes características comuns à maioria dos componentes de software:

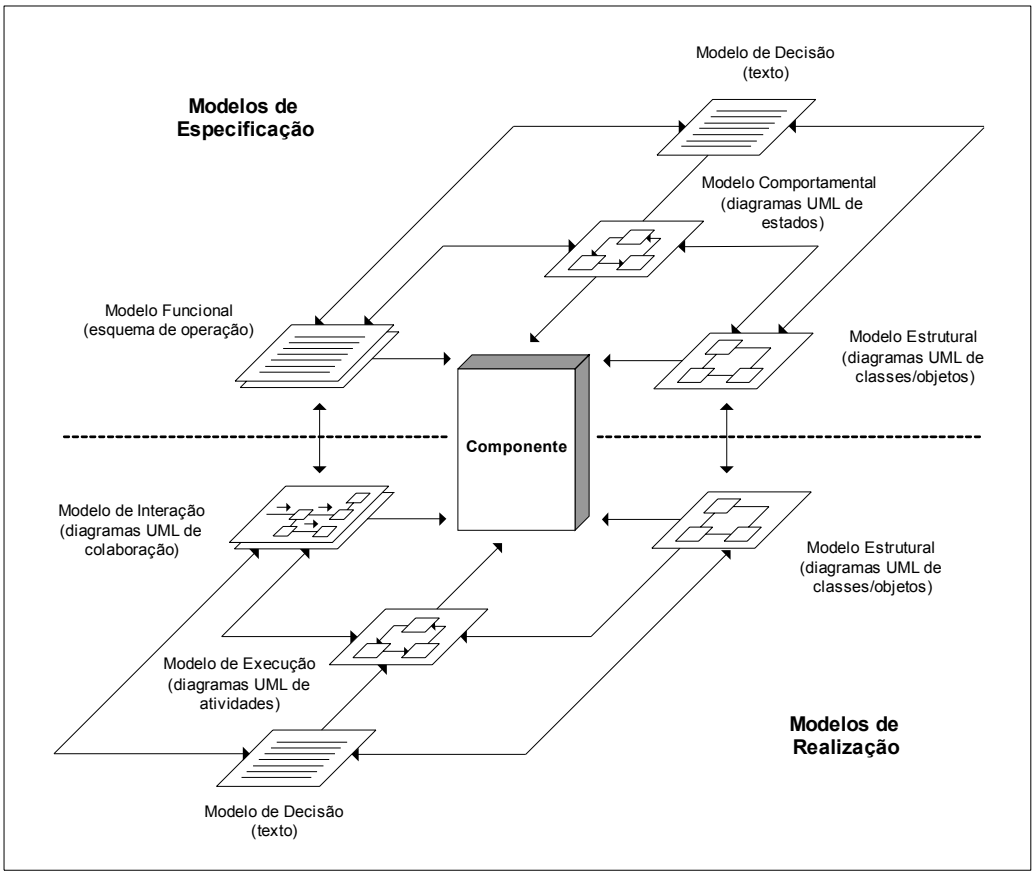


Figura 2.1. Modelagem de componentes baseada em UML segundo a abordagem Kobra (ATKINSON, 2000).

- Possuem duas facetas: a Especificação, que descreve as características visíveis externamente e assim define os requisitos que devem ser satisfeitos; e a Realização, que descreve como os componentes satisfazem suas especificações, através de suas implementações, interagindo inclusive com outros componentes que o compõem. Cheesman e Daniels também citam estes dois ângulos de visão (CHEESMAN, 2001).

  O modelo de componentes do *KobrA*, ilustrado na figura 2.1, é divido exatamente nas duas facetas de Especificação e Realização (ATKINSON, 2000). A primeira descreve as características externamente visíveis de um componente, definindo os requisitos que deve atender; e a segunda descreve como um componente satisfaz (implementa) estes requisitos em termos das interações com seus componentes de mais baixo nível na hierarquia de composição. A abordagem *KobrA* estabelece um modelo de composição constituído por uma estrutura hierárquica em forma de árvore de componentes, onde cada relacionamento pai/filho representa um nível de composição. Softwares, neste caso, também são considerados componentes, assim como componentes também podem ser compostos por outros componentes, recursivamente.

- Seguem um padrão de composição. O modelo RM-ODP (ISO/IEC, 1995) também define, além de um padrão de interface (Especificação) de componente, um padrão de composição. Composição de componentes, para formarem um sistema maior, ou até mesmo outro componente (SZYPERSKI, 1998). Este padrão de composição estabelece como os componentes interagem para constituírem uma arquitetura.

- A arquitetura pode ser entendida como a infra-estrutura sobre a qual será instalado o componente. Componentes estão intimamente relacionados à plataforma onde serão

executados (COUNCILL; HEINEMAN, 2001). A plataforma pode ser um sistema operacional, uma linguagem de programação, um *middleware*[7] etc.

- Devem atender a requisitos não-funcionais (em termos de qualidade). Um componente deve satisfazer atributos mínimos de qualidade, que garantam, conseqüentemente, um produto final com qualidade agregada. Um componente pode propagar má qualidade ao sistema que compõe, sendo uma peça de software do mesmo, por ter sido mal projetado, equivocadamente empregado, ou mesmo indevidamente configurado.

Vê-se, pela última característica acima relacionada, a importância de se garantir certo nível de qualidade, por parte dos componentes, uma vez que serão empregados em diferentes soluções. O ponto deste trabalho é, justamente, propor um processo de apoio a outros processos de desenvolvimento de software, de manutenção de um repositório de componentes, que evolua no sentido acumular componentes, atualizados e refinados constantemente, para serem empregados em diferentes soluções.

2.2. Processos orientados ao DBC

Pelo fato de ser um novo paradigma dentro da disciplina da Engenharia de Software, os processos que envolvem o desenvolvimento de software baseado em componentes exigem novas abordagens diferentes das tradicionais. Entre os principais motivos de fracassos em

[7] Uma camada de serviços genéricos sobre os quais aplicações específicas são instaladas. O J2EE (ALUR; CRUPI; MALKS, 2002), neste contexto, pode ser considerado um *middleware* de infra-estrutura de componentes.

programas de reutilização de componentes de software, estão a não introdução de processos específicos e a não adaptação de processos convencionais ao DBC (MORISIO, 2000).

Sommerville define um processo de software como um conjunto de atividades e resultados associados que levam à produção de um produto de software, podendo envolver o desenvolvimento de software a partir do zero, embora, cada vez mais, os processos ocorram a partir da expansão e modificação de sistemas já existentes (SOMMERVILLE, 2003). O autor propõe ainda uma abordagem de desenvolvimento de software orientada ao reuso, contando com uma base de componentes reutilizáveis e com alguma infra-estrutura de integração destes componentes. A figura 2.2 ilustra o fluxo de atividades deste processo.

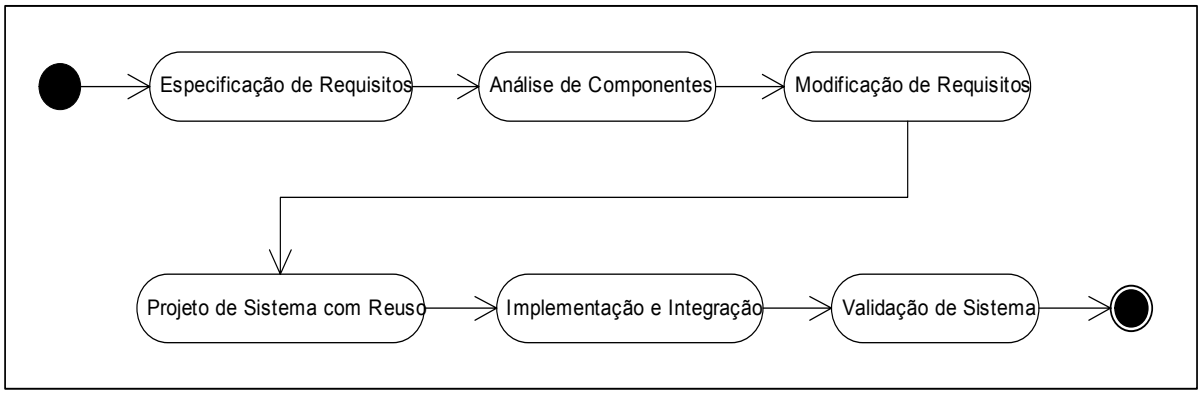


Figura 2.2. Processo proposto por Sommerville orientado ao reuso de componentes (SOMMERVILLE, 2003).

As atividades de análise e projeto do RUP (KRUCHTEN, 2000), por sua vez, são centradas na noção de arquiteturas e componentes. O modelo de processo do RUP dá, então, suporte ao desenvolvimento baseado em componentes das seguintes formas:

- A abordagem iterativa permite aos desenvolvedores a identificação progressiva de componentes e a decisão a respeito de quais componentes devem ser desenvolvidos, reutilizados e comprados.

- O foco em arquitetura de software permite a articulação de uma estrutura, que enumere os componentes e as formas com que são integrados, bem como os mecanismos fundamentais e os padrões utilizados.

- Conceitos como pacotes, subsistemas, e camadas são utilizados durante as fases de análise e projeto para organizar os componentes e especificar suas interfaces.

- Testes são conduzidos individualmente e depois, gradativamente, expandidos em conjuntos maiores de integração de componentes.

O fluxo de atividades principal da disciplina de análise e projeto do RUP envolve a utilização de componentes, bem como assim o faz a disciplina de implementação. Na figura 2.3, o fluxo de atividades de análise e projeto, segundo a notação do RUP (o fluxo de implementação é apresentado e utilizado como exemplo no capítulo 3). Percebem-se, então, duas atividades orientadas ao DBC: Projeto de Componentes e Projeto de Componentes de Tempo-Real.

Stalinger e outros (STALLINGER, 2002) também apontam a necessidade de uma adaptação dos processos vigentes e da estrutura de uma organização para se alcançar resultados positivos no desenvolvimento baseado em componentes. O OOSPICE é um projeto com base em pesquisas empíricas de práticas do DBC na indústria de software, com foco nos processos, tecnologia e qualidade de software de sistemas produzidos a partir de componentes. Dos objetivos do projeto OOSPICE, podem-se destacar: a proposta de um modelo de processo unificado para o DBC, baseado em um meta-modelo; o desenvolvimento de uma metodologia de avaliação de processos de DBC; a definição de perfis que retratem a

capacidade de provedores de componentes (ex. empresas fabricantes de COTS); a especificação de uma metodologia de desenvolvimento baseado em componentes, através de processos técnicos e administrativos, métodos e ferramentas; e a elaboração de um modelo estendido do padrão ISO/IEC 15504 para avaliação de processos de software baseados em componentes.

Assim como uma das propostas do projeto OOSPICE é propor um meta-modelo para processos de DBC, outros modelos de processo também se baseiam em um meta-nível de abstração. O OPEN é um meta-modelo de processo – um modelo de um modelo (FIRESMITH; HENDERSON-SELLERS, 2002). Encapsula as regras (gramática) de uma notação e as representa graficamente através de conceitos orientados a objeto, tais como classes e relacionamentos, porém em um meta-nível.

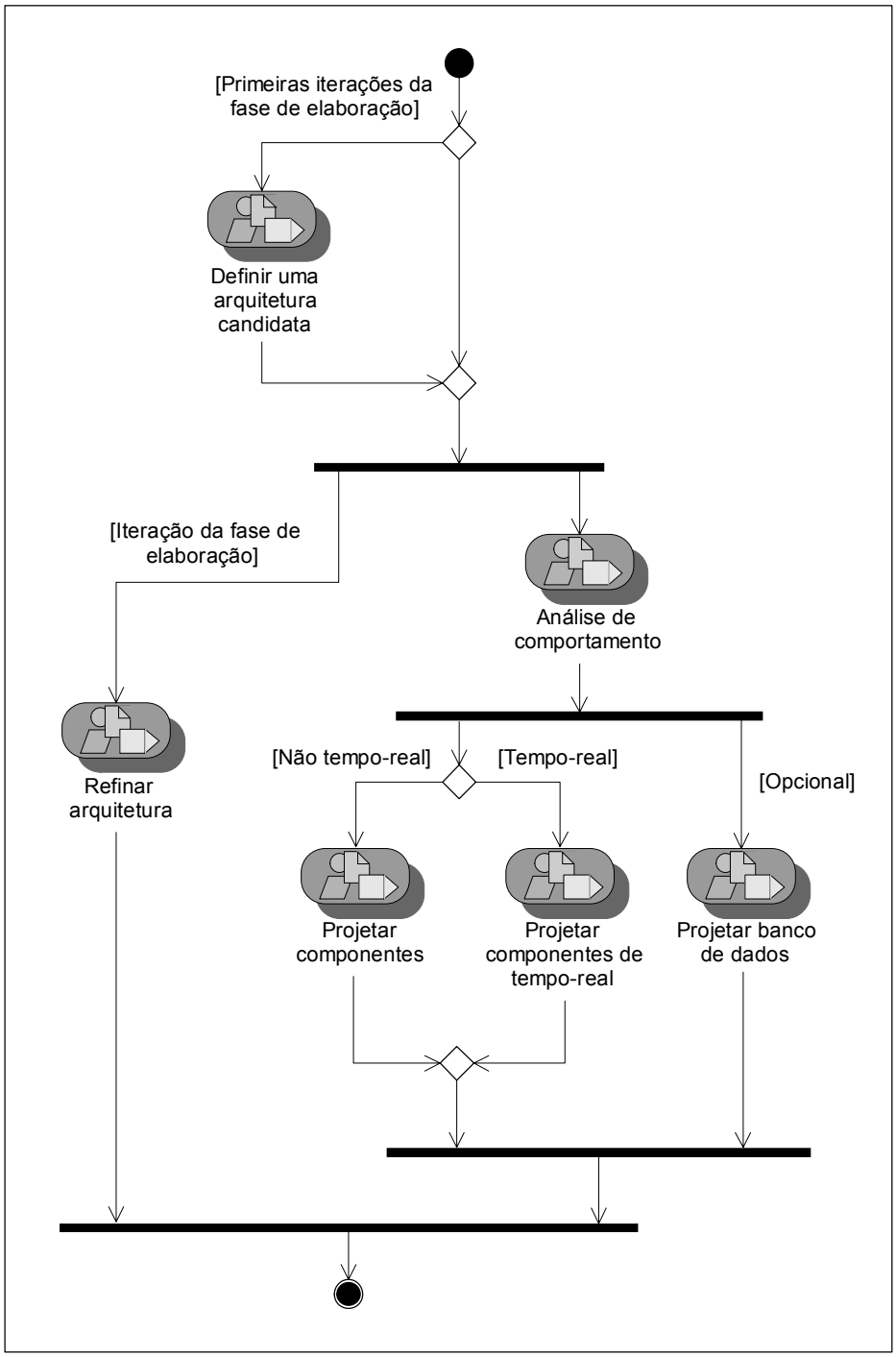


Figura 2.3. Fluxo de atividades da disciplina de análise e projeto do RUP (KRUCHTEN, 2000).

2.3. Meta-modelos de processo

O OPEN estabelece que todo processo pode ser definido por Produtores executando Unidades de Trabalho para produzirem Produtos de Trabalho. Diretivas (Guias) guiam o processo, que é organizado temporalmente em estágios. Tudo é documentado através de diferentes Linguagens. O OPF é uma biblioteca composta por componentes essenciais de um processo genérico, como instâncias dos meta-tipos apresentados na figura 2.4. O OPEN seria, então, o meta-modelo base do modelo de processo OPF. Analogamente, o *Unified Process* (UP) (JACOBSON; BOOCH; RUMBAUGH, 1999) poderia ser considerado o meta-modelo do qual deriva o RUP (HENDERSON-SELLERS, 2000). Os processos de uma organização, por fim, seriam as instâncias de algum modelo, assim como processos definidos segundo o OPF ou o RUP.

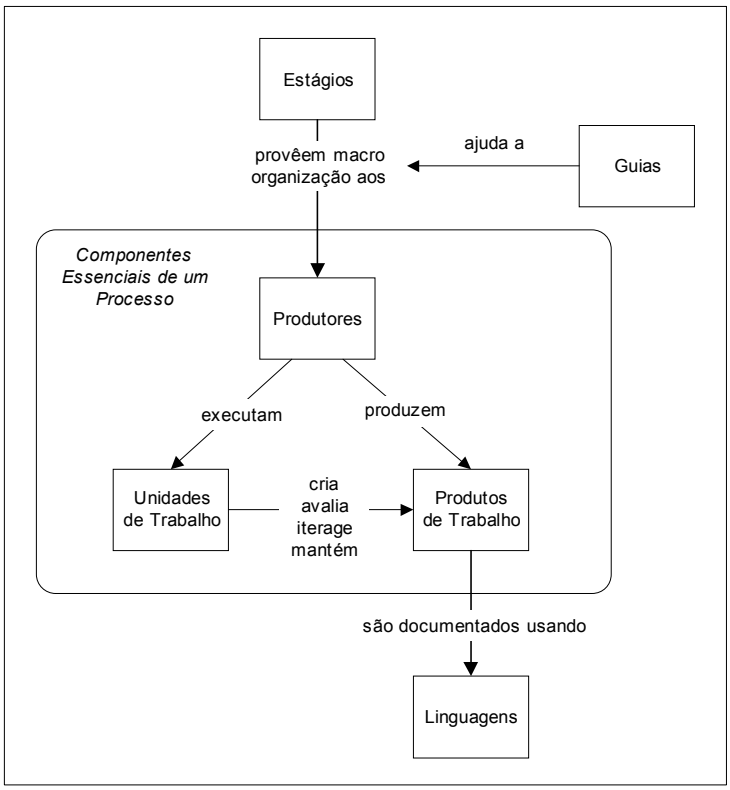


Figura 2.4. Estrutura básica do meta-modelo de processo OPEN (FIRESMITH HENDERSON-SELLERS, 2002).

2.3.1. MOF

O modelo MOF (OMG, 2002a) da OMG define uma arquitetura para definição e descrição de meta-dados em quatro camadas. Este modelo em camadas, para representar meta-informações em quatro níveis, não é na verdade exclusividade do MOF. Sua estrutura é apresentada na figura 2.5.

Os níveis podem ser interpretados conforme as descrições a seguir:

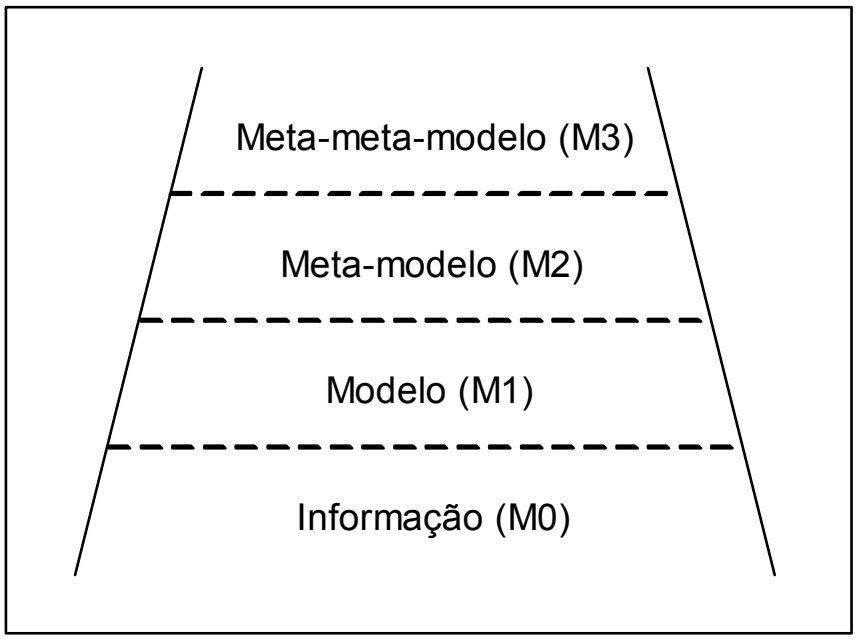


Figura 2.5. Modelo de arquitetura para descrição de meta-dados em quatro camadas (OMG, 2002a).

- *Informação (M0)*. Os objetos deste nível menos abstrato compõem a informação que se quer representar. Esta informação é tipicamente denominada de "dados".

- *Modelo (M1).* Este nível compreende os meta-dados que desecrevem a informação do nível M0.

- *Meta-modelo (M2)*. Esta camada define a estrutura e a semântica dos meta-dados do nível M1. Pode ser entendida como uma linguagem, na medida em que define uma gramática para notação de modelos.

- *Meta-meta-modelo (M3)*. Camada mais abstrata que descreve as regras sintáticas e semânticas dos dados descritos em M2 (meta-meta-dados). Uma linguagem para definir diferentes tipos de meta-modelos.

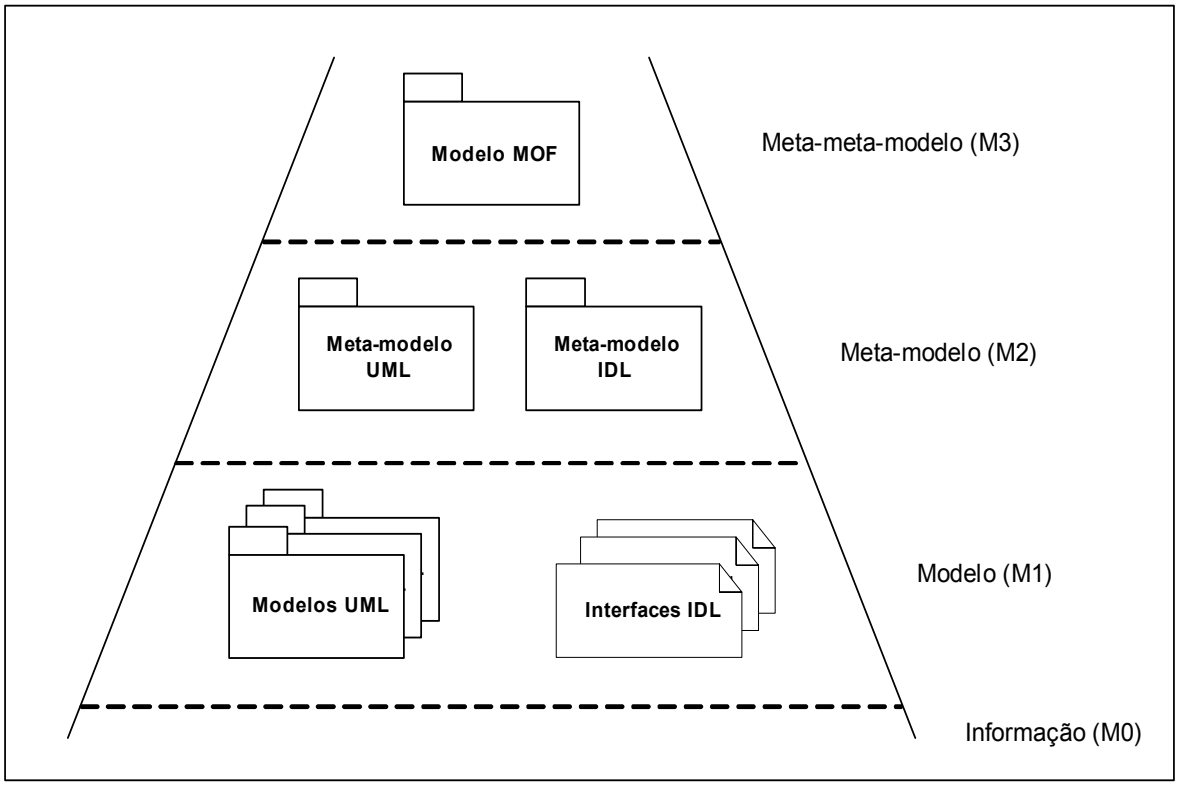


Figura 2.6. O modelo MOF baseado na arquitetura de definição e descrição de meta-dados em quatro camadas (OMG, 2002a).

O modelo MOF é baseado, então, na arquitetura de quatro camadas, porém se diferencia de outros modelos baseados na mesma arquitetura, por dois motivos principais:

- O modelo MOF é orientado a objetos, permitindo uma meta-modelagem alinhada aos elementos da UML. O esquema da figura 2.6 é representado através de pacotes UML.

- O modelo MOF é auto-descritivo, isto é formalmente definido através de suas próprias meta-contruções.

A figura 2.6 representa as camadas do modelo MOF. A notação ao estilo de pacotes UML significa o alinhamento à linguagem de modelagem unificada. Considere-se, por exemplo, uma classe UML no nível M1 de abstração. Esta classe pode ser descrita como uma instância do elemento (ou classe dentro do conceito de OO) *Class* do meta-modelo UML, que por sua vez é descrito como uma instância do elemento também denominado *Class* no modelo MOF. Finalmente, a classe *Class* do modelo MOF se descreve por si só.

2.3.2. Extensões UML

O meta-modelo UML (OMG, 2003) é definido como uma das camadas da arquitetura de meta-modelagem de quatro níveis de abstração, apresentada na seção anterior. Essa abordagem traz, entre outros benefícios, uma base arquitetural para definir futuras extensões e alinhar o meta-modelo UML com outros padrões baseados na arquitetura de quatro camadas, particularmente o MOF.

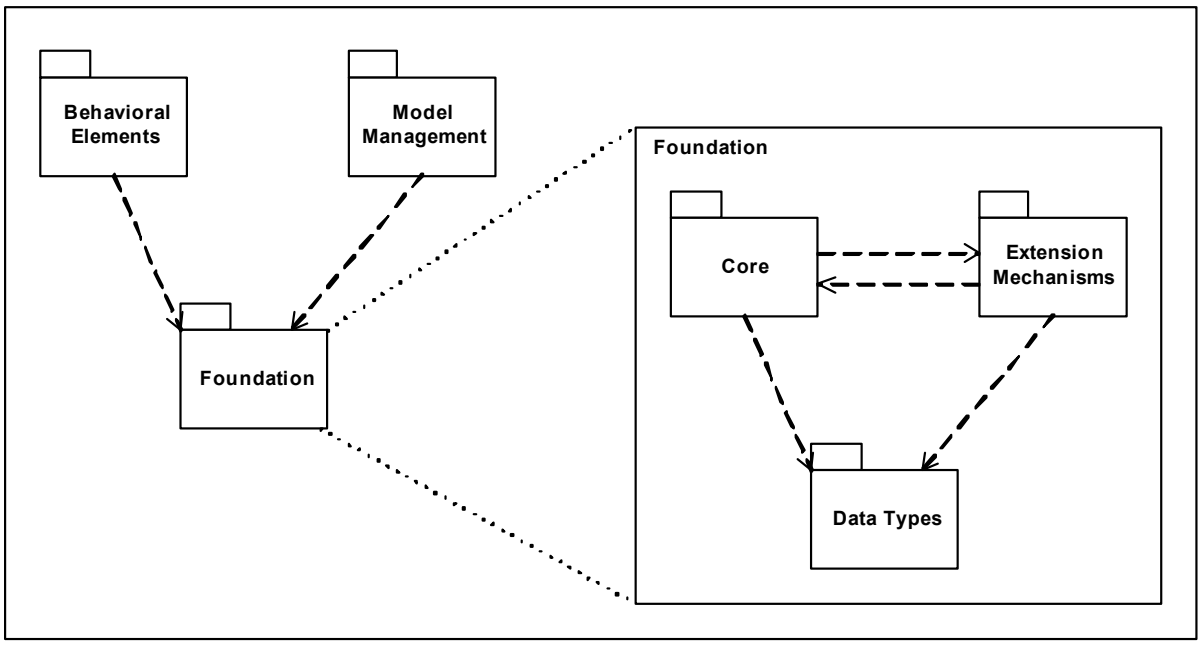


Figura 2.7. Os pacotes de mais alto nível de abstração do meta-modelo UML. No detalhe, a estrutura do pacote Foundation (OMG, 2003).

O meta-modelo UML encontra-se, então, no nível M2 do MOF. Sua complexidade é gerenciada através da organização de uma estrutura em pacotes de meta-classes extremamente coesas entre si. O meta-modelo pode ser decomposto nos pacotes de mais alto nível *Behavioral Elements*, *Model Management* e *Foundation* (figura 2.7), segundo a estrutura abaixo. O pacote *Foundation* é a infra-estrutura da linguagem que, especifica a estrutura estática dos modelos, e é decomposto nos pacotes *Core*, *Extension Mechanisms* e *Data Types*. O *Core* é o pacote mais fundamental; *Data Types* especifica os diferentes tipos de dados que definem a UML. O pacote *Behavioral Elements* especifica os conceitos dinâmicos básicos para o comportamento dos elementos da UML, constituindo-se dos sub-pacotes apresentados na figura 2.8. E, finalmente, *Model Management* define a semântica e as notações dos conceitos de pacotes, subsistemas e modelos UML.

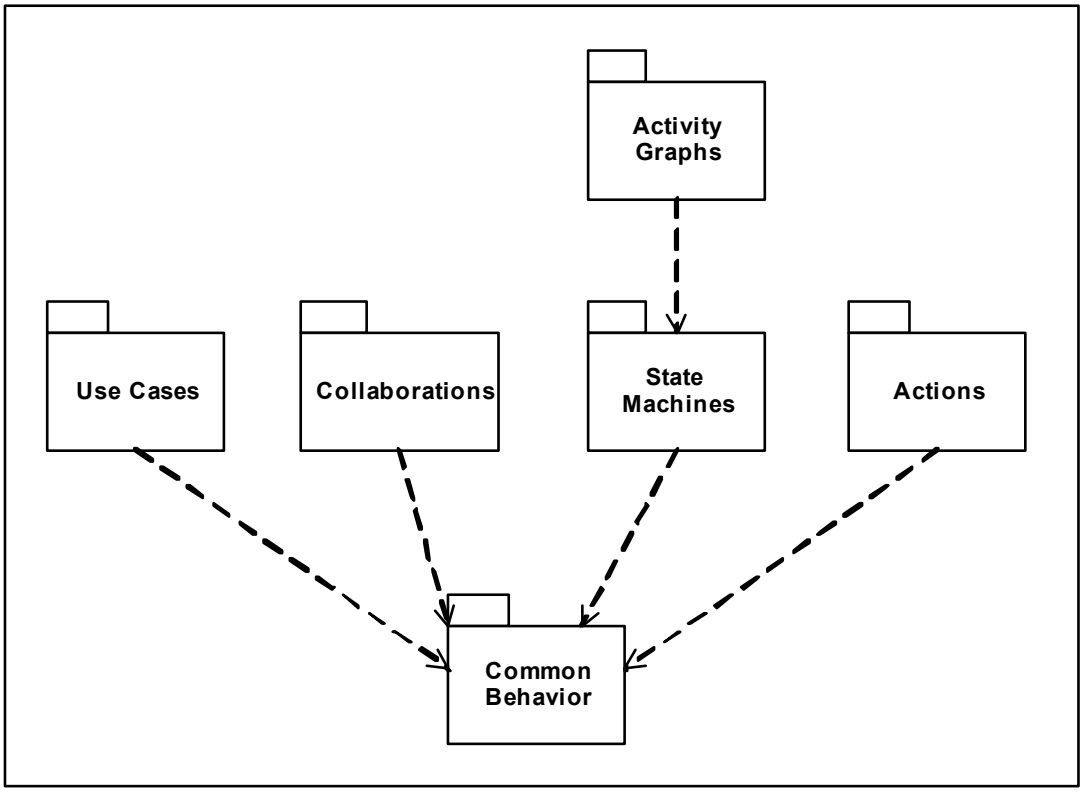


Figura 2.8. A estrutura do pacote Behavioral Elements do meta-modelo UML (OMG, 2003).

O pacote *Extension Mechanisms* define como modelos UML específicos são personalizados e estendidos com novas regras semânticas através de estereótipos (*stereotypes*), restrições (*constraints*) e propriedades (*tag definitions* e *tagged values*) (OMG, 2003). Os mecanismos de extensão da UML podem ser de dois tipos (ENGELS; HECKEL; SAUER, 2003):

- *Pesados (heavyweight).* Os mecanismos pesados de extensão UML são providos pelo MOF, o que significa que é possível se alterar e adaptar a UML, modificando-se o meta-modelo no nível M2. Este tipo de extensão tem grande impacto na linguagem e, portanto, não pode ser realizado por usuários individuais, ficando a cargo das organizações responsáveis pela manutenção de novos modelos de extensão e perfis UML, a serem utilizados como padrões de mercado.

- *Leves (lightweight)*. São mecanismos pré-definidos pela UML através dos quais quaisquer indvíduos envolvidos com atividades de modelagem podem adaptar a UML às suas necessidades. Estes mecanismos de extensão permitem que a linguagem unificada de propósito genérico seja adaptada a domínios específicos. O resultado desta adaptação por grupos de usuários interessados ou fabricantes de software constitui o que se costuma chamar de um perfil UML (*UML Profile*), dedicado a um contexto específico.

2.3.3. SPEM

O *Software Process Engineering Metamodel* (SPEM) (OMG, 2005) define um meta-modelo baseado em subconjuntos do meta-modelo UML, para descrever um processo concreto de desenvolvimento de software ou uma família de processos relacionados de desenvolvimento de software. Uma ferramenta baseada no SPEM seria uma ferramenta para autoria e personalização de processos (como é feito com o processo proposto neste trabalho).

Além de descrever um meta-modelo de processo, o SPEM é definido também como um perfil UML, isto é um pacote de elementos de modelagem personalizados para um propósito ou domínio específico, estendendo o meta-modelo através de estereótipos, propriedades e restrições. Como um perfil, o SPEM define adicionalmente novas representações gráficas de elementos, tratadas mais detalhadamente no capítulo 3 de descrição do processo.

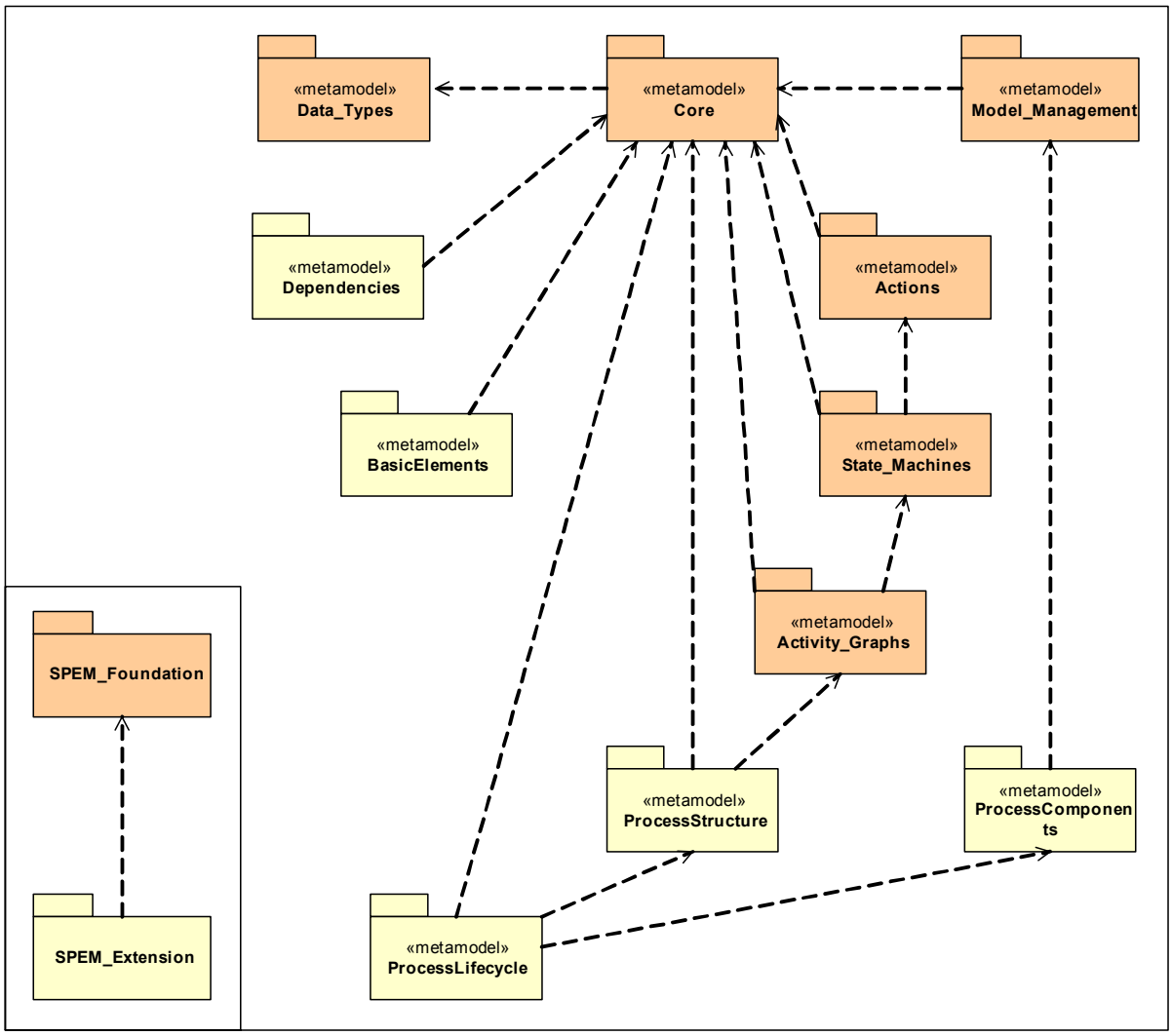


Figura 2.9. Estrutura de pacotes do meta-modelo SPEM (OMG, 2005).

O meta-modelo de processo descrito pelo SPEM é definido através do MOF. Os pacotes fundamentais do SPEM (*SPEM_Foundation*) estariam na camada M2, derivando-se dos seguintes pacotes do meta-modelo UML:

- *SPEM_Foundation::Data Types*
- *SPEM_Foundation::Core*
- *SPEM_Foundation::Actions*
- *SPEM_Foundation::State Machines*

- *SPEM_Foundation::Activity Graphs*
- *SPEM_Foundation::Model Management*

Os pacotes estendidos do SPEM seriam, como o nome já diz, extensões dos pacotes fundamentais. A estrutura de pacotes, então, do meta-modelo SPEM é apresentada na figura abaixo.

De forma bastante semelhante ao OPEN, cuja estrutura básica do meta-modelo (apresentada na figura 2.4) constitui-se por Produtores produzindo Produtos de Trabalho, através da execução de Unidades de Trabalho, o SPEM também define seu meta-modelo, envolvendo o núcleo conceitual retratado na figura 2.10. Assim, papéis (*Roles*) são responsáveis por produtos (*WorkProducts*), consumidos ou produzidos por uma atividade (*Activity*).

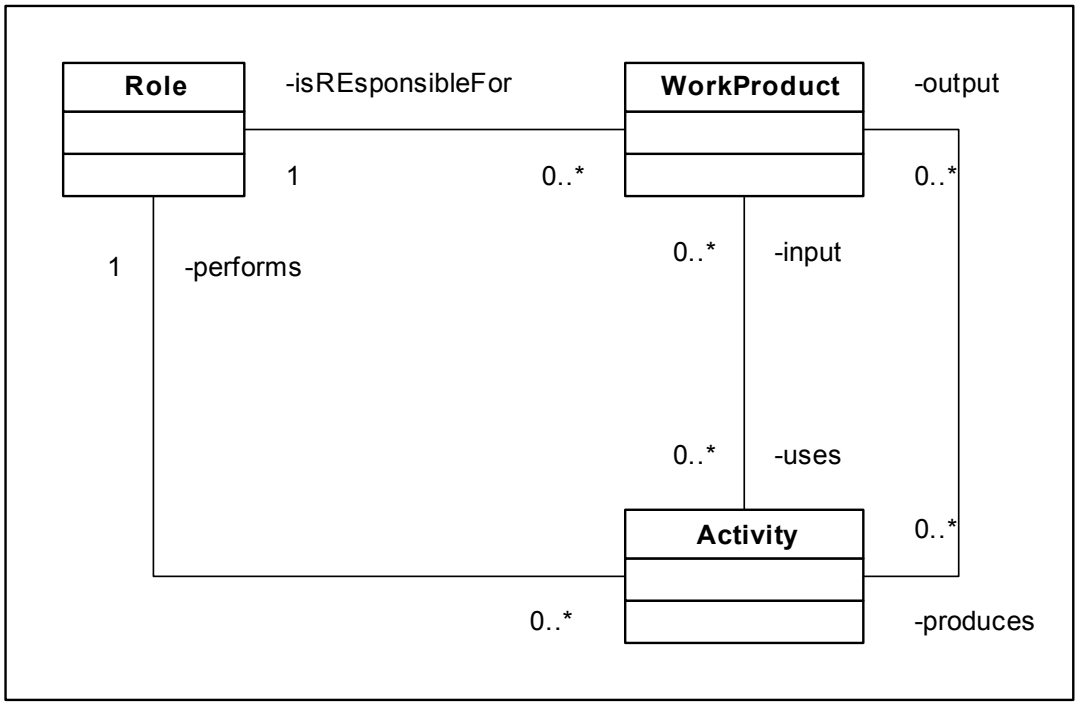


Figura 2.10. O modelo conceitual do SPEM: Roles, WorkProducts e Activities (OMG, 2005).

O modelo de processo proposto neste trabalho – denominado *Komponento* – é descrito através dos elementos do meta-modelo SPEM. Na arquitetura de meta-modelagem de quatro camadas do MOF, o *Komponento* estaria, então, no nível M1 de abstração. O capítulo a seguir é dedicado à descrição detalhada do modelo.

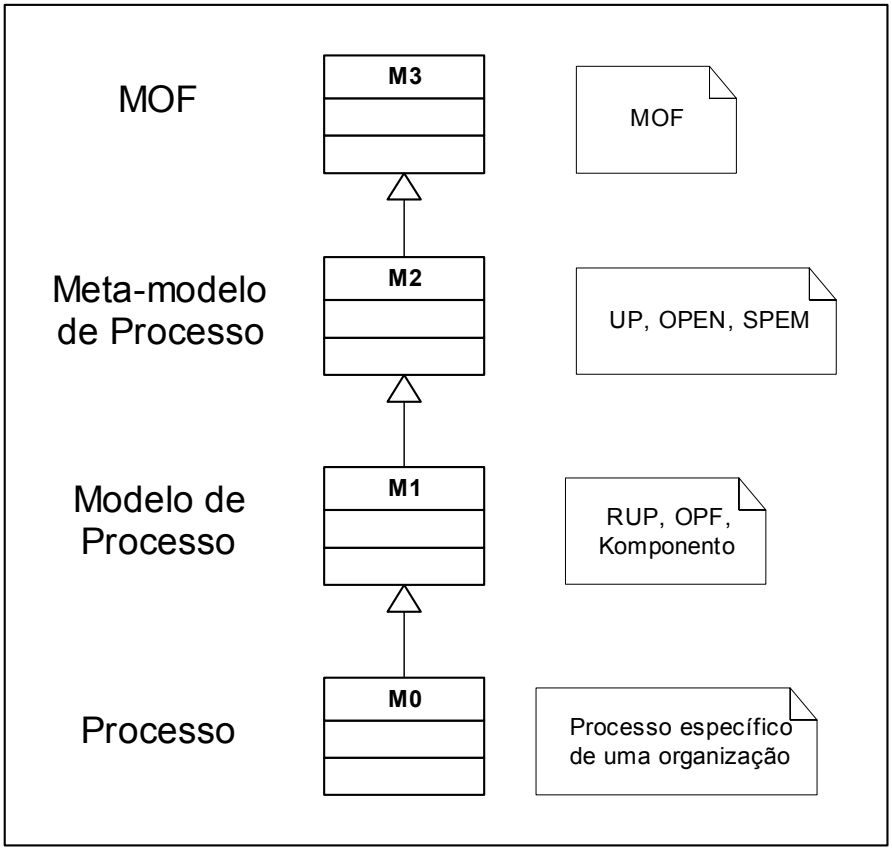


Figura 2.11. Mapeamento de processos baseados em meta-modelos na estrutura do MOF.

Situando-se os outros processos baseados em meta-modelos com suporte ao DBC no modelo MOF, teríamos a estrutura representada na figura 2.11. No nível M2, encaixar-se-iam os meta-modelos de processo UP (JACOBSON; BOOCH; RUMBAUGH, 1999), OPEN (FIRESMITH; HENDERSON-SELLERS, 2002) e SPEM. Um nível abaixo estariam os modelos de processo diretamente derivados: o RUP, como uma instância do UP, distribuído como um produto, dentro da filosofia de que processos de software também são software

(OSTERWEIL, 1987), sendo projetado, desenvolvido, distribuído, e mantido como qualquer ferramenta de software (KRUCHTEN, 2000); o OPF (FIRESMITH; HENDERSON-SELLERS, 2002) como uma biblioteca de elementos de processo instanciados do meta-modelo OPEN.

O *Komponento* estaria definido, então, no mesmo nível do RUP e do OPF, isto é no nível M1, derivando diretamente do meta-modelo SPEM, descrevendo um modelo de processo de apoio ao DBC, através de elementos (atividades, produtos, atores etc.), que seriam instâncias diretas dos meta-elementos do SPEM (*Activity*, *WorkProduct*, *Role* etc.).

Os termos "processo" e "modelo de processo" representam, de acordo com o MOF, dois níveis de abstração distintos: o primeiro define um modelo, a partir das regras estabelecidas por um meta-modelo; o segundo representa a instância deste modelo, concretizada para um determinado propósito ou organização específica (como retratado na figura 2.11). Ambos os termos "processo" e "modelo de processo", entretanto, doravante neste trabalho referentes ao *Komponento*, serão empregados com o mesmo significado (para maior comodidade e flexibilidade do linguagem): de um modelo de processo – definindo elementos derivados de um meta-modelo – que pode ser instanciado e personalizado ao ser instituído dentro de uma empresa ou organização.

2.4. Normas ISO

Este trabalho se baseou em duas normas ISO como referência para a definição de processos e para o atendimento de atributos de qualidade de software. São elas, respectivamente: ISO/IEC 12207 (*Information Technology – Software Life Cycle Processes*) e ISO/IEC 9126 (*Information Technology – Software product evaluation – Quality characteristics and guidelines for their use*).

2.4.1. ISO/IEC 12207

O padrão ISO/IEC 12207 (*Information Technology – Software Life Cycle Processes*) é o primeiro padrão internacional a prover um conjunto de processos, atividades, e tarefas do ciclo de vida de software como parte integrante de sistemas maiores, produtos independentes, e serviços de software. O padrão provê uma arquitetura comum de processos para aquisição, fornecimento, desenvolvimento, operação e manutenção de software (ISO/IEC, 1999).

A tabela 1 relaciona todas as áreas de processo cobertas pelo padrão ISO/IEC 12207. As áreas marcadas constituem as diretivas selecionadas para orientar a definição dos requisitos do modelo de processo *Komponento*. São elas “gerência de configuração”, “garantia de qualidade”, e “gerência de ativos”.

Processos	**Propósitos/Resultados**
Processo de aquisição	Preparação de aquisição
	Seleção de fornecedor
	Monitoramento de fornecedor
	Aceitação de cliente
	Avaliação de produto
Processo de fornecimento	Processo de fornecimento
Processo de desenvolvimento	Levantamento de requisitos
	Análise de requisitos de sistema
	Projeto de arquitetura de sistema
	Análise de requisitos de software
	Projeto de software
	Construção de software
	Integração de software
	Teste de software
	Teste & Integração de sistema
	Avaliação de produto
Processo de operação	Uso operacional
	Suporte ao cliente
Processo de manutenção	Processo de manutenção
Processo de documentação	Processo de documentação
Processo de gerência de configuração	Processo de gerência de configuração
Processo de garantia de qualidade	Processo de garantia de qualidade

Processo de verificação	Processo de verificação
Processo de validação	Processo de validação
Processo de revisão conjunta	Processo de revisão conjunta
Processo de auditoria	Processo de auditoria
Processo de resolução de problemas	Processo de resolução de problemas
Processo de usabilidade	Processo de usabilidade
Processo de gerência	Gerência da organização
	Gerência de projeto
	Gerência de qualidade
	Gerência de risco
	Alinhamento organizacional
	Métricas
	Gerência de ativos
	Gerência de programa de reuso
	Engenharia de domínio
Processo de melhoria	Estabelecimento de processo
	Avaliação de processo
	Melhoria de processo
Processo de gerência de recursos humanos	Treinamento
Processo de infra-estrutura	Processo de infra-estrutura

Tabela 1. Processos da norma ISO/IEC 12207 (ISO/IEC, 1999).

É muito difícil, entretanto, que todos estes processos sejam executados independentemente. Eles muitas vezes se sobrepõem. Os pontos a serem abordados, propostos por cada um dos

três processos selecionados na tabela acima, serão apresentados nas seções de requisitos de processo do próximo capítulo. O *Komponento* deverá atender, então, através de diferentes abordagens de implementação e de um único modelo, requisitos estabelecidos pelos processos de "gerência de configuração", "garantia de qualidade" e "gerência de ativos" estabelecidos pela norma ISO/IEC 12207. A seguir, uma pequena introdução conceitual para cada uma das três áreas.

2.4.1.1. Processo de gerência de configuração

Faz parte da configuração de um software grande quantidade de itens de informação, que sofrem constante alteração, durante todo o ciclo de vida. O processo de gerência de configuração permite que um controle seja realizado sobre a criação e as subseqüentes e sucessivas alterações destes itens.

> "*Item de configuração de software* é o nome dado ao item de configuração produzido durante o processo de desenvolvimento de software, para o qual é importante que seja realizado o controle das alterações. Um conjunto de itens de configuração de software compõe uma *configuração de software*". (SANCHES, 2001)

Ainda segundo Sanches (SANCHES, 2001), os pontos colocados pela ISO/IEC 12207, que cobrem a área de gerência de configuração, podem ocorrer tanto ao final de cada fase do processo, quanto de alguma outra forma considerada mais apropriada pelos gerentes de projeto. Os itens de configuração devem ser armazenados, então, num *repositório de itens de configuração*.

2.4.1.2. Processo de garantia de qualidade

O reuso promete certas vantagens, principalmente porque software não necessita ser reescrito (MORISIO, 2000). Encurta-se o tempo de desenvolvimento, pois menos produtos precisam ser construídos. E a qualidade melhora, uma vez que componentes já testados podem ser reutilizados. Ainda segundo Morisio, o nível de qualidade dos produtos deve atender aos requisitos do projeto, através de processos que qualifiquem e documentem estes produtos.

Alinhando-se as propriedades do *Komponento* aos requisitos estabelecidos pelos objetivos da área de garantia de qualidade imposta pela ISO/IEC 12207, deve-se prever um plano de avaliação de qualidade dos produtos gerados pelo processo e do próprio processo. A estratégia implementada pelo *Komponento* relaciona-se mais à garantia de qualidade dos produtos gerados (componentes, basicamente). A qualidade do processo em si, em termos de eficiência e maturidade, seriam mais responsabilidades das áreas de avaliação e melhoria de processos da ISO/IEC 12207. Trabalhos futuros dentro do contexto da pesquisa aqui abordada são sugeridos no capítulo 5, como a utilização dos modelos CMM (*Capability Maturity Model*) e ISO/IEC 15504.

2.4.1.3. Gerência de ativos

O propósito da gerência de ativos é administrar o ciclo de vida dos componentes reutilizáveis desde a concepção até a morte. Ativos reutilizáveis implementam uma solução de um problema para um contexto específico. Podem ser categorizados em *frameworks*, padrões e componentes (OMG, 2004). No contexto deste trabalho, é relevante a categoria de componentes, por ser este tipo de ativo o alvo da pesquisa.

Estas três categorias podem ser avaliadas segundo três dimensões, representadas na figura 2.12:

- *Granularidade.* A granularidade descreve quantos problemas particulares ou alternativas de solução podem estar empacotados em um ativo, desde um componente específico para uma determinada função ou para um pequeno conjunto de funções, até *frameworks* de desenvolvimento, segundo a definição de Gamma (GAMMA, 1995) para resolução de problemas mais complexos através de soluções mais genéricas.

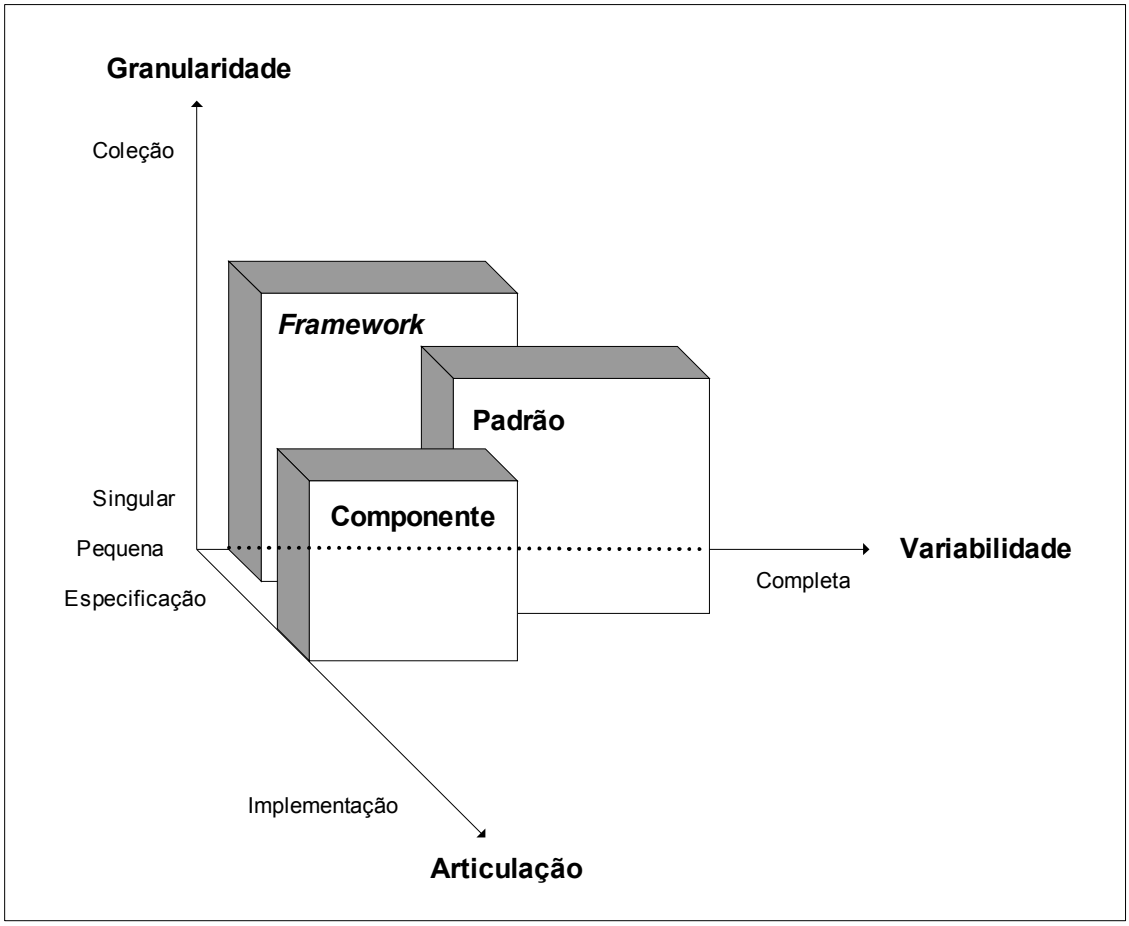


Figura 2.12. Tipos de ativos reutilizáveis de software (OMG, 2004).

- *Variabilidade.* Quanto à variabilidade, um componente (ativo de software do tipo componente) pode ser uma caixa preta, sem pontos de extensão de suas funcionalidades, ou uma caixa branca, onde o modelo de sua implementação é conhecido, e sua arquitetura flexível o bastante para suportar alterações.

- *Articulação.* Descreve o grau de completude com que o conjunto de artefatos de ativos provêem a solução. Especificações – sejam em formato de textos, modelos, ou até mesmo protótipos de tela – são ativos pouco articulados, enquanto que implementações e especificações, empacotadas juntamente com documentos de requisitos, casos de uso, teste etc., apresentam um grau de articulação mais elevado.

No RUP, a noção de artefato é definida como um pedaço de informação produzida, modificada, ou utilizada por um processo. Artefatos são quaisquer produtos de trabalho (*WorkProduct* dentro das meta-classes do SPEM) relativos ao ciclo de vida do software: documentos, modelos, códigos fonte, arquivos de configuração, casos de teste, *scripts* etc. Um ativo define, então, um conjunto de artefatos em um contexto específico de aplicação, em que deve ser empregado o software.

2.4.2. ISO/IEC 9126

O padrão 9126 definido pela ISO (International Organization for Standardization) e pela IEC (International Electrotechnical Commission) estabelece, com um mínimo de sobreposição, todas as características para se avaliar a qualidade de um software.

A norma não provê nenhum tipo de metodologia de avaliação de qualidade. Define 6 conjuntos de critérios no suporte a tecnologias para se especificar e avaliar a qualidade de produtos e processos de sofwtare (ISO/IEC, 1991). Os atributos de qualidade são então:

- *Funcionalidade (Functionality).* Um conjunto de atributos relativos a um conjunto de funções e a especificação de suas propriedades. As funções são aquelas que satisfazem necessidades implicadas ou estabelecidas.

    - *Capacidade de combinar (Suitability).* Atributos de software relativos à presença ou conveniência de um conjunto de funções para tarefas específicas.

    - *Precisão (Accuracy).* Atributos de software relativos à provisão de resultados ou efeitos corretos ou esperados.

    - *Interoperabilidade (Interoperability).* Atributos de software relativos a sua habilidade de interagir com sistemas específicos.

    - *Obediência (Compliance).* Atributos que fazem o software aderir a padrões ou convenções ou regulamentos legais relativos à aplicação ou prescrições similares.

    - *Segurança (Security).* Atributos de software relativos a sua habilidade de prevenir acesso não autorizado, acidental ou deliberadamente, a programas e dados.

- *Confiabilidade (Reliability).* Um conjunto de atributos relativos à capacidade que o software tem de manter seu nível de desempenho sob condições estabelecidas por um período de tempo estabelecido.

    - *Maturidade (Maturity).* Atributos de software relativos à freqüência de fracassos por falhas do sistema.

    - *Tolerância a falhas (Fault tolerance).* Atributos de software relativos a sua habilidade de manter um nível específico de desempenho em casos de falhas no software ou infrações de suas interfaces especificadas.

    - *Recuperação de falhas (Recoverability).* Atributos de software relativos a capacidade de restabelecer seu nível de desempenho e recuperar os dados diretamente afetados, em caso de falha, no momento e com o esforço necessário para tal.

- *Usabilidade (Usability).* Um conjunto de atributos relativos à facilidade de uso, e a avaliação individual deste uso, por um conjunto estabelecido ou implicado de usuários.

    - *Capacidade de entendimento (Understandability).* Atributos de software relativos ao esforço dos usuários para reconhecer os conceitos lógicos e sua capacidade de aplicação.

- *Capacidade de aprendizado (Learnability).* Atributos de software relativos ao esforço dos usuários para aprender sua aplicação.

- *Operabilidade (Operability).* Atributos de software relativos ao esforço dos usuários para operação e controle de operação.

- *Eficiência (Efficiency).* Um conjunto de atributos sobre o relacionamento entre o nível de desempenho do software e a quantidade de recursos utilizada, sob condições estabelecidas.

  - *Comportamento temporal (Time behaviour).* Atributos de software relativos aos tempos de resposta e processamento e às taxas de rendimento quanto ao desempenho de suas funções.

  - *Comportamento de recursos (Resource behaviour).* Atributos de software relativos à quantidade de recursos utilizados e à duração deste uso quanto ao desempenho de suas funções.

- *Manutenibilidade (Maintainability).* Um conjunto de atributos relativos à facilidade de se fazer alterações especificadas.

  - *Capacidade de análise (Analysability).* Atributos de software relativos ao esforço necessário para o diagnóstico de deficiências ou causas de falha, ou para a identificação de partes a serem modificadas.

- *Capacidade de alteração (Changeability).* Atributos de software relativos ao esforço necessário para modificação, remoção de falhas ou alteração de ambiente.

- *Estabilidade (Stability).* Atributos de software relativos ao risco de efeitos inesperados de modificações.

- *Testabilidade (Testability).* Atributos de software relativos ao esforço necessário para validação de modificações.

- *Portabilidade (Portability).* Um conjunto de atributos relativos à habilidade do software de ser portado de um ambiente para outro.

  - *Adaptabilidade (Adaptability).* Atributos de software relativos à oportunidade de sua adaptação a diferentes ambientes especificados, sem se aplicarem outras ações ou meios, que não aqueles providos para este propósito para o software considerado.

  - *Instalabilidade (Installability).* Atributos de software relativos ao esforço necessário para instalação do software em um ambiente especificado.

  - *Conformidade (Conformance).* Atributos de software que o fazem aderir a padrões e convenções relativos à portabilidade.

- *Capacidade de substituição (Replaceability).* Atributos de software relativos à oportunidade e esforço de utilizá-lo no lugar de outro software especificado no ambiente deste software.

3. Requisitos do processo

O objetivo deste capítulo é formalizar os requisitos que o *Komponento* deve atender, no que diz respeito às atividades que deve implementar e aos produtos gerados por tais atividades, em cada fase do processo.

De acordo com Eeles, um requisito descreve uma condição ou habilidade de um sistema; derivados diretamente de necessidades do usuário, ou declarados em um contrato, padrão, especificação, ou outro documento formalmente imposto (EELES, 2001).

Utilizando a mesma abordagem proposta por Osterweils (OSTERWEILS, 1997), em que processos de software podem ser tratados como softwares, propriamente ditos, uma vez que ambos são executados, ambos endereçam requisitos que precisam ser compreendidos, ambos se baseiam em modelos, ambos devem evoluir através de métricas, e assim por diante.

Assim como requisitos funcionais e não-funcionais devem ditar o comportamento de um sistema, os requisitos de processo também. Requisitos funcionais de processo seriam, por exemplo, artefatos a serem produzidos por determinada atividade ou prazos a serem atendidos durante cada fase de um projeto. Requisitos não-funcionais seriam interoperabilidade, através de um meta-modelo (HENDERSON-SELLERS, 2001), por exemplo, que represente elementos comuns entre diferentes processos; portabilidade (ainda a descrição em um meta-nível (OMG, 2002a) de processo (OMG, 2005) permite a interoperabilidade entre ferramentas CASE com suporte ao modelo); ou robustez, requisito que pode prever como o processo deve reagir em situações de perda de elementos chave como pessoas ou artefatos.

Os requisitos gerais e específicos, respectivamente, apresentados em 3.1 e 3.2, correspondem aos requisitos (tanto funcionais quanto não-funcionais) do *Komponento*, assim como as formas com que são endereçados tais requisitos são descritas, brevemente. O restante do capítulo descreve mais pormenorizadamente toda a estrutura do processo (modelo geral, atividades, tarefas, atores, artefatos, e ciclo de vida).

3.1. Requisitos gerais

Como já colocado no capítulo 2, os requisitos gerais do modelo de processo *Komponento* orientaram-se através das áreas de "gerência de configuração", "garantia de qualidade" e "gerência de ativos", implementando os resultados estabelecidos em cada uma delas pela norma ISO/IEC 12207.

3.1.1. Processo de gerência de configuração

Os pontos a serem atendidos pelo processo de gerência de configuração de software, segundo a norma, devem ser:

- *Uma estratégia de gerência de configuração será desenvolvida (1).* O *Komponento* é baseado no *Subversion* (PILATO, 2004), um sistema de controle de versão de distribuição gratuita e código aberto, responsável inclusive pelo atendimento da maioria dos requisitos exigidos pelo processo de gerência de configuração estipulados na norma ISO/IEC 12207 para este processo.

A adoção do *Subversion* vem do fato de ser um software baseado em código e licença abertos, para que a equipe que for instituir – implantar em sua empresa ou organização – o processo *Komponento*, não arque com custos na compra de ferramentas de software de suporte ao modelo, tornando o processo mais acessível.

- *Todos os itens gerados pelo processo ou projeto serão identificados, definidos, e servirão de base (2).* Esta na verdade é a função do *Komponento*. A atividade de produção de componentes representa justamente o armazenamento de itens de configuração no repositório, cuja implementação é baseada no *Subversion*.

- *Modificações e entrega dos itens serão controlados (3).* O *Komponento* descreve o papel do bibliotecário de reuso que, entre outras funções, é responsável pelo controle de usuários que irão acessar os componentes do repositório. A política de controle de acesso é definida pelo modelo da ferramenta de controle de versão utilizada.

- *Os estados dos itens e requisições de modificação serão registrados e relatados (4).* Mais uma vez o *Subversion* oferece funções de suporte, como *branching*[8] e *tagging*[9] (vide glossário), definindo meios de se criarem versões de produtos inteiros através de rótulos. Notificações de alteração no repositório podem ser efetuadas via correio eletrônico, através de scripts associados a determinados itens de configuração e lista de destinatários interessados.

---

[8] *Branching* é o ato de se manter linhas paralelas de desenvolvimento de um mesmo projeto original (PILATO, 2004).

[9] *Tagging* é o ato de se rotular um projeto, sob controle de versão, "congelando-se" os arquivos que compõem este projeto, em seus respectivos estados e versões, num dado momento (PILATO, 2004).

- *A integridade e consistência dos itens serão garantidas (5).* O *Subversion* identifica diferenças entre arquivos utilizando um algoritmo capaz de trabalhar tanto em textos legíveis por humanos, quanto em dados binários, sendo capaz de armazenar ambos os tipos com o mesmo nível de compressão. O mecanismo de *merge*[10] do *Subversion* garante a integridade e consistência de dados alterados, simultaneamente, por diferentes usuários, em um mesmo trecho de arquivo.

- *Armazenamento, manipulação, e entrega dos itens serão controlados (6).* O modelo de armazenamento dos itens de configuração do repositório segue o padrão RAS da OMG, que especifica os tipos de ativos de software a serem reutilizados. Um *facade* (GAMMA, 1995) pode ser implementado, encapsulando as funções do *Subversion* numa interface unificada em conformidade com os serviços de acesso sugeridos na especificação RAS (OMG, 2004).

3.1.2. Processo de garantia de qualidade

O plano de avaliação de qualidade dos produtos gerados pelo processo, então, deve apresentar os seguintes resultados:

- *Uma estratégia para conduzir a garantia de qualidade será desenvolvida, implementada e mantida (1).* A estratégia de garantia de qualidade dos produtos

[10] *Merge* significa juntar duas versões de um mesmo arquivo, que foram modificadas simultaneamente, resolvendo-se os eventuais conflitos.

gerados pelo *Komponento* será implementada através de uma atividade de certificação de componentes, através de uma homologação interna de documentos de casos de teste, e do emprego de uma metodologia de avaliação de arquiteturas (CLEMENTS; KAZMAN; KLEIN, 2002), baseada nos requisitos de qualidade sugeridos pela norma ISO/IEC 9126.

- *Evidência de garantia de qualidade será produzida e mantida (2).* A estrutura de armazenamento dos componentes, de acordo com os perfis propostos pela especificação RAS (OMG, 2004), também contempla a documentação de requisitos não-funcionais para ativos mantidos em um repositório de software.

- *Problemas ou inconformidades com os requisitos do contrato serão identificados.* A atividade de certificação de componentes representa uma maneira de garantir que o processo produza componentes com certo grau de qualidade, verificando e validando seus requisitos funcionais através de testes padronizados. Os processos que implementam garantia de qualidade podem ser encarados como uma certificação normativa (que obedece a um conjunto de normas estabelecidas), segundo a definição de Wallnau (WALLNAU, 2004), uma vez que se orientam pelos requisitos da norma ISO/IEC 9126 de avaliação de qualidade de software.

- *A aderência de produtos, processos e atividades aos padrões, procedimentos e requisitos aplicáveis será verificada objetivamente.* A atividade de avaliação de componentes, segundo o método ATAM (CLEMENTS; KAZMAN; KLEIN, 2002), realizada no processo *Komponento*, tem implícita uma metodologia de verificação de qualidade contra requisitos estabelecidos pelo padrão ISO/IEC 9126. As tarefas de se

manter a aderência de produtos ao formato de armazenamento, estipulado pela especificação de ativos reutilizáveis da OMG (OMG, 2004), e da modelagem de processos à notação SPEM (OMG, 2005), podem ser automatizadas através da utilização de ferramentas com suporte a XMI (OMG, 2002b), compatível com ambos os modelos.

3.1.3. Gerência de ativos

A ISO-12207 estabelece para o processo de gerência de ativos as seguintes orientações:

- *Um plano de gerência de ativos será documentado (1).* O modelo de processo *Komponento* descreve o plano de gerência de ativos, através do detalhamento da atividade de gerência (do repositório de componentes) nas subatividades de gerência de repositórios, usuários, catálogos, e também de alterações nos itens do repositório. As subatividades foram inspiradas diretamente das operações de gerência de repositórios e componentes, gerência de usuários, catálogos e versões de (APPERLY, 2001).

- *Um mecanismo de armazenamento e recuperação será operacionalizado (2).* O mecanismo de armazenamento do *Komponento* segue o formato RAS, baseado em arquivos de meta-informação, que descrevem a estrutura do componente. O mecanismo de recuperação é proposto como uma implementação piloto de serviços de acesso a um repositório, segundo a especificação de uma interface de programação proposta pela OMG (OMG, 2004), apresentada no capítulo 4.

- *A utilização de ativos será registrada (3).* Ativos, tratando-se especificamente do tipo componente, no contexto deste trabalho, tem a evolução de seus ciclos de vida controlada, através de históricos de versões. Tais históricos são mantidos, para cada item, pela ferramenta de gerência de configuração de apoio ao processo, com data e hora de modificação e usuário responsável pela transação.

- *Gerência de configuração será executada para os ativos (4).* O processo de gerência de configuração de ativos corresponde a todas as atividades relacionadas na seção anterior.

3.1.4. Outros

Outros processos do ciclo de vida do software podem ser relacionados aos requisitos do *Komponento*, porém, de maneira menos formal. A identificação de tarefas, atividades e produtos associados de um determinado processo padrão dentro de uma organização, juntamente com as características de desempenho esperadas, é um resultado proposto pela área de "estabelecimento de processo". Este resultado é de fato alcançado pela descrição do *Komponento*, apresentada no capítulo 4, através do detalhamento de atividades, tarefas, artefatos, atores, e ciclo de vida, dentro da notação para definição de processos SPEM.

Além do "estabelecimento de processo", as áreas de "avaliação de processo" e "melhoria de processo" também podem ser aplicadas à definição do *Komponento*. De acordo com a norma ISO/IEC 12207, enquanto a primeira visa (a) determinar a extensão com que os processos

padrões de uma organização contribuem para a realização de suas metas de negócio e (b) auxiliar esta organização a focar na necessidade de melhoria contínua de processo; a segunda tem o propósito de melhorar continuamente a eficácia e eficiência dos processos utilizados por uma organização no alinhamento com as necessidades de negócio. Mesmo sendo o *Komponento* um processo relativamente simples, com um conjunto não muito grande de elementos definidos, modelos como o CMM ou o ISO/IEC 15504 podem ter seus conceitos adaptados para atenderem a necessidades de muitas diferentes situações de desenvolvimento de software (WIEGERS, 1999), cobrindo desde grandes corporações com complexos processos definidos com extrema formalidade, até pequenas organizações com processos menos rígidos de software ou até inexistentes.

Apesar do processo de "gerência de programa de reuso" estar fortemente relacionado ao escopo deste trabalho, o objetivo do *Komponento* é o de dar apoio a outros processos, quanto à reutilização de um repositório de componentes. A instituição de um programa de reuso constitui uma atividade muito mais administrativa, estabelecendo os objetivos abaixo:

- Definir a estratégia de reuso da organização, incluindo seus propósitos, escopo, metas e objetivos;
- Identificar os domínios em que haja oportunidades de reuso a serem investigadas ou em que se pretenda praticar reuso;
- Avaliar a habilidade de reuso sistemático da organização;
- Avaliar cada domínio para se determinar seu potencial;
- Preparar um plano de reuso para implementar a prática de reuso na organização;

- Estabelecer mecanismos de realimentação, comunicação, e notificação, que operem entre administradores de programas de reuso, gerentes de ativos, engenheiros de domínio, desenvolvedores, operadores, e mantenedores;
- Executar o plano de implementação do programa de reuso;
- Monitorar e avaliar o programa de reuso;
- Melhorar o programa de reuso.

Desta forma, o *Komponento*, na realidade, estaria contido num programa de reuso, dando apoio, principalmente, aos processos da área de desenvolvimento de software, ao invés de ser responsável por implementar cada saída representada pelos itens descritos acima.

3.2. Requisitos específicos

Os requisitos específicos do *Komponento*, de cunho mais tecnológico, como padrões a serem seguidos e ferramentas de software a serem empregadas, são descritos nas três próximas seções.

3.2.1. Baseado em um meta-modelo

O *Komponento* deve ser definido com base em um meta-modelo a fim de se garantirem:

- *Compatibilidade de conceitos.* Processos definidos através do mesmo meta-modelo facilitam a integração entre seus elementos. Suponham-se três processos do ciclo de

vida: garantia de qualidade, documentação, e desenvolvimento. O primeiro descrevendo uma atividade de certificação de acordo com padrões de documentação; o segundo, uma atividade de documentação, propriamente dita, de código fonte; e o terceiro, atividades de implementação e teste do ciclo de vida de desenvolvimento. Os três processos podem ser integrados de acordo com um mesmo modelo de representação, como ilustra a figura 3.1.

Desta forma, a qualidade dos produtos da documentação de código fonte, gerada pela atividade de documentação, pode ser avaliada pela atividade de certificação, segundo uma formato padrão de documentação, simultaneamente, às atividades de implementação e teste de componentes.

- *Suporte de ferramentas*. A descrição de um processo em um meta-nível é a base para o suporte de ferramentas CASE (STALLINGER, 2002) e condizente com as atividades de padronização de processos da OMG.

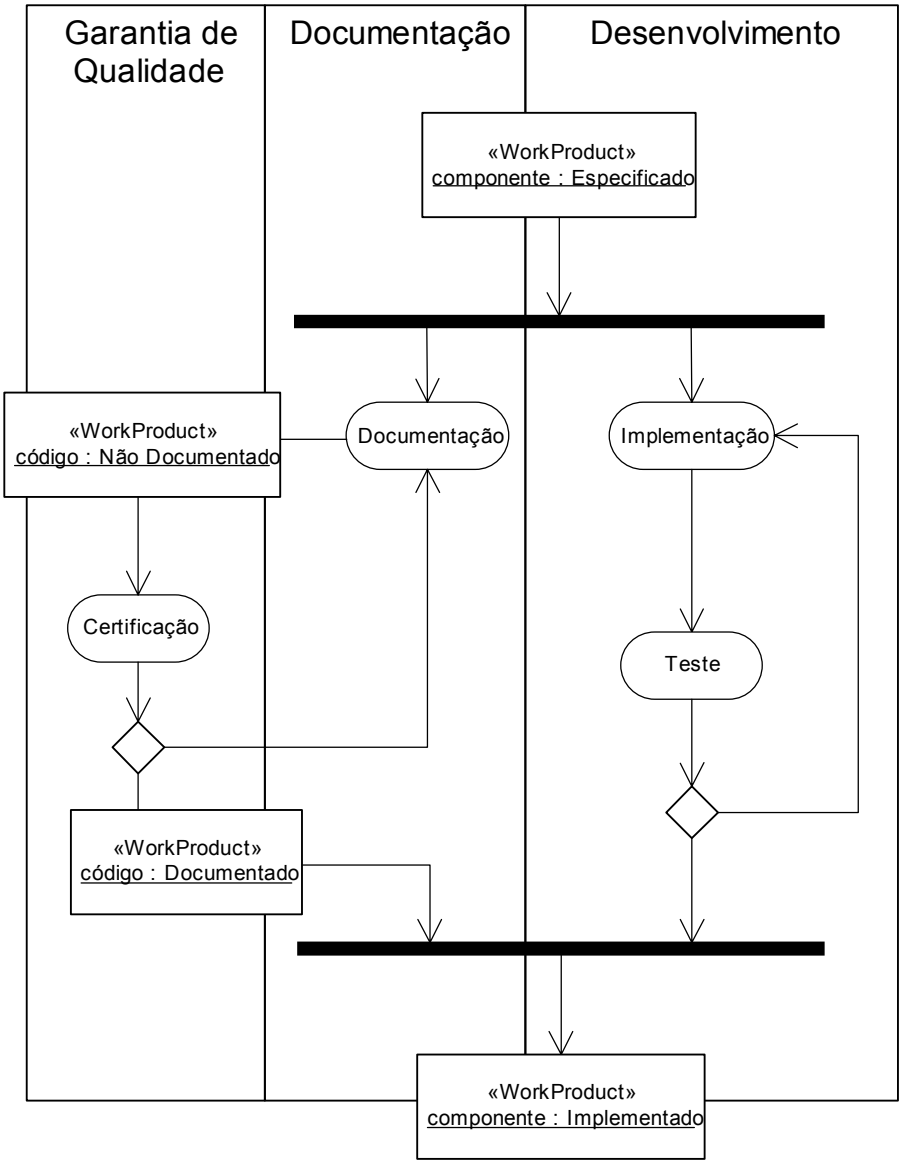


Figura 3.1. Os processos de garantia de qualidade, documentação, e desenvolvimento integrados através notação SPEM.

- *Interoperabilidade.* XMI (*XML Metadata Interchange*) é um formato de intercâmbio de dados largamente utilizado para o compartilhamento de objetos em XML (OMG, 2002b) Ferramentas CASE, ao armazenarem modelos em XMI, e ao serem capazes de interpretar estes e outros modelos representados segundo a mesma especificação, promovem interoperabilidade de processos, através da persistência de objetos de um

meta-modelo, em um formato comum. A estrutura de um processo, persistida em XMI, torna-se portável através de diferentes ferramentas com suporte ao padrão.

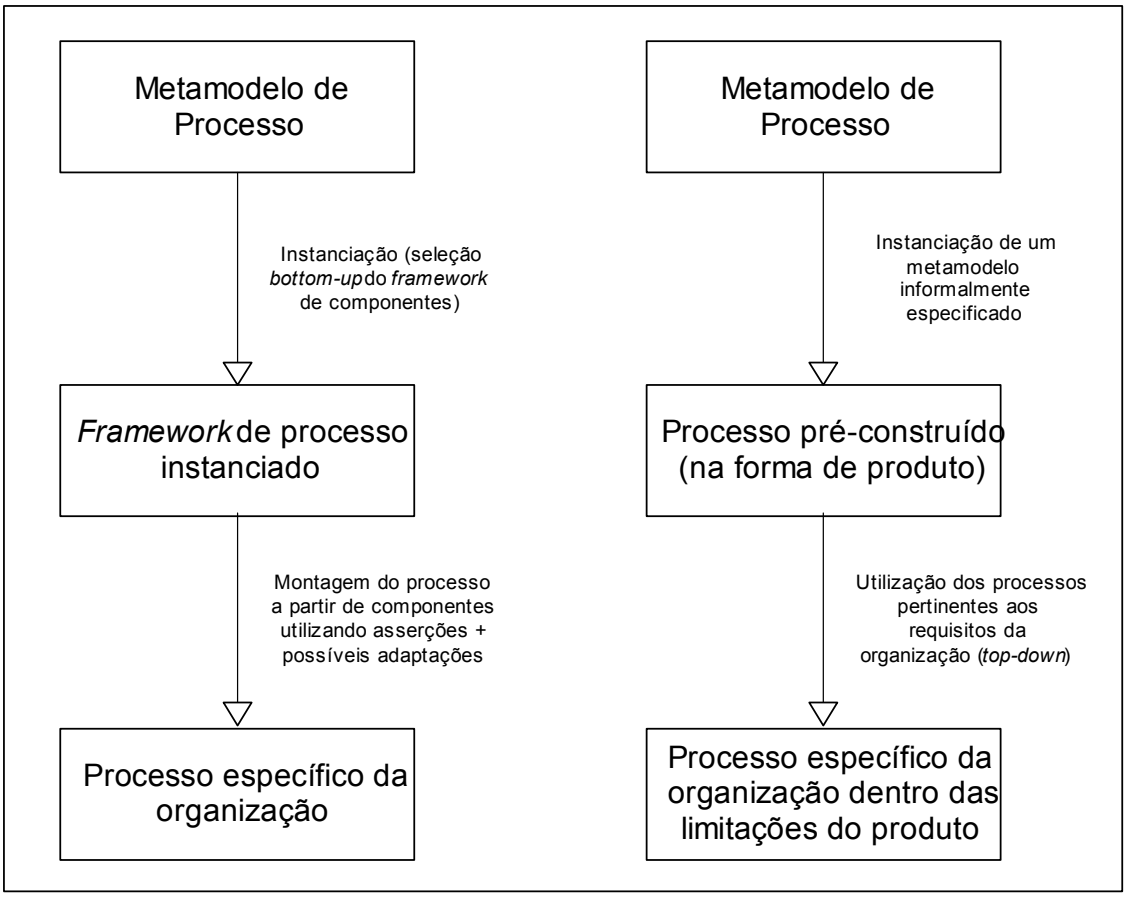


Figura 3.2. Níveis de abstração dos modelos de processo OPEN e RUP (HENDERSON-SELLERS, 2001).

- *Flexibilidade.* A conformidade com um *framework*[11], definido por um meta-modelo, permite que um processo seja construído, de maneira bastante flexível, a partir dos elementos deste *framework* (HENDERSON-SELLERS, 2001). É o caso do OPF (FIRESMITH; HENDERSON-SELLERS, 2002), que define uma biblioteca de

[11] Um *framework* de processo, segundo (FIRESMITH; HENDERSON-SELLERS, 2002), seria uma coleção coesa de classes predefinidas de componentes de processo (ex. produtos, atores, atividades) e um conjunto de diretivas para instanciação, extensão e configuração do *framework*.

elementos segundo o modelo OPEN, para a definição de um processo totalmente novo e alinhado às necessidades de uma organização.

Já o RUP, por sua vez, é uma instância pré-configurada de um meta-modelo (figura 3.2), informalmente definido, vendido como um produto completo, derivando processos através de uma abordagem *top-down*. O OPEN, ao disponibilizar e instanciar uma série de elementos de um meta-modelo, acaba proporcionando a definição de processos *bottom-up*, ao preço de todos os custos embutidos neste esforço.

3.2.2. Processo acoplável

Um dos principais meios de se obter com sucesso um ciclo de vida de desenvolvimento de componentes é o de se implementar um processo, que inclua um ambiente integrado de gerência de configuração e biblioteca de componentes (APPERLY, 2001). E ainda segundo Apperly, a chave para uma solução escalável são um processo e um conjunto de ferramentas também escaláveis. A proposta do *Komponento*, então, é ser um processo não somente escalável, mas também adaptável a outros processos e as suas mudanças.

Para se exemplificar um processo acoplável, imagine-se um processo de utilização de um repositório (como é o *Komponento*), definindo duas operações básicas: consumo e produção de componentes em um repositório. A figura 3.3 ilustra este processo bastante simples como um bloco de fluxo de atividades.

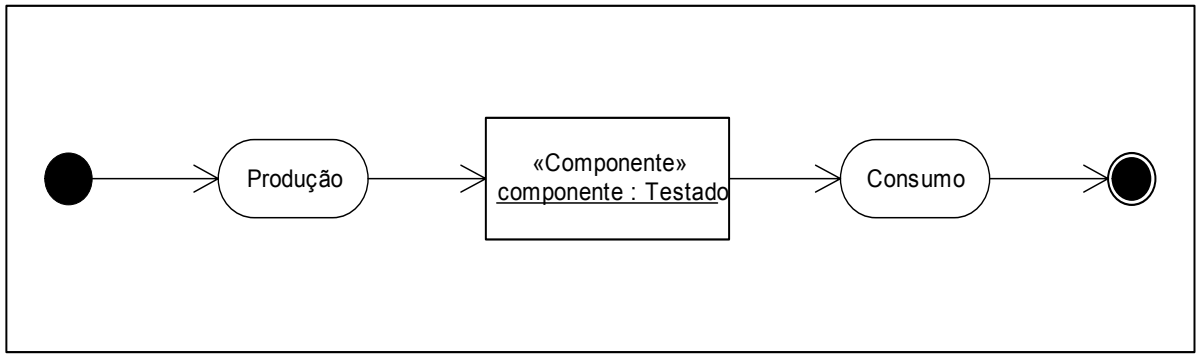


Figura 3.3. Fluxo de atividades de um processo hipotético simples de utilização de um repositório de componentes[12].

Considerando-se, então, o fluxo de Implementação proposto pelo RUP (KRUCHTEN, 2000), como um outro bloco de fluxo de atividades, ambos os processos poderiam ser integrados da forma apresentada na figura 3.4.

1) A atividade de consumo do processo de produção/consumo de componentes produziria componentes testados extraídos do repositório; estes componentes testados serviriam de entrada para a atividade de Implementar de Componentes, através de reengenharia, composição, ou apenas configuração de componentes;

2) Os componentes construídos para integrar o sistema testado e validado, ao final do fluxo de Implementação do RUP, constituiriam a entrada da atividade de produção do processo de produção/consumo de componentes, que selecionaria novos componentes a serem inseridos no repositório e atualizações de componentes já existentes.

[12] O processo descreve, através de um pequeno diagrama de atividades, a produção de componentes em um repositório (omitido na figura, mas representado pelo objeto "componente"), e o consumo de componentes deste mesmo repositório.

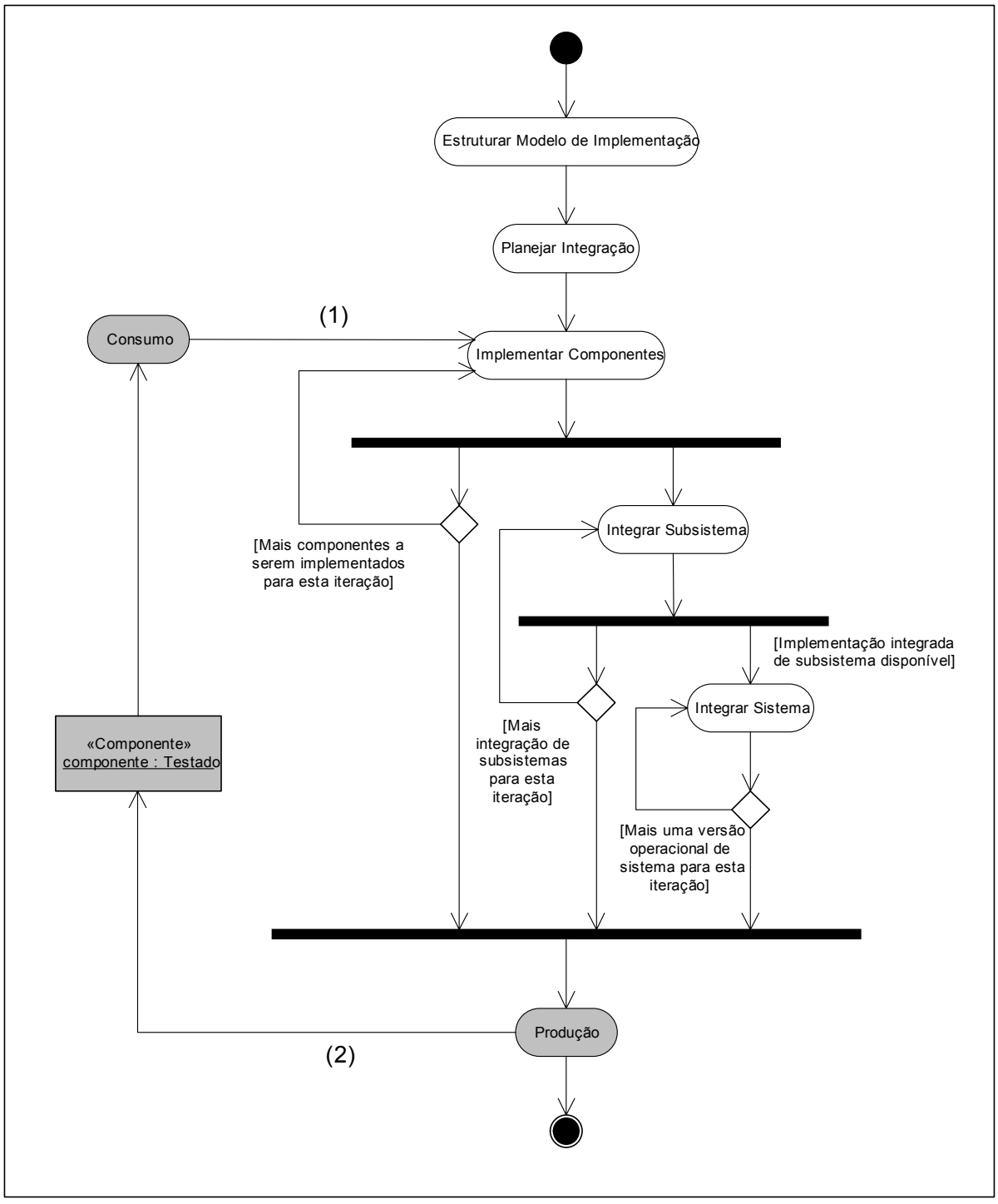


Figura 3.4. Processo de produção/consumo de componentes dando suporte ao fluxo de Implementação proposto pelo RUP (KRUCHTEN, 2000).

3.2.3. Utilização de um repositório

Para Henninger (HENNINGER, 1997) a estrutura de um repositório é a chave para se obterem melhores resultados de acesso a componentes. O que conta não é apenas um algoritmo eficiente de busca, mas também um repositório indexado e bem estruturado de componentes de software.

Aos elementos desejáveis de um sistema de biblioteca de componentes (GUO, 2000):

- Sistema automatizado de biblioteca, com uma interface gráfica de usuário, para navegação, busca e consumo;
- Estrutura padrão de componente (para incluir propósito, descrição funcional, nível de certificação, restrições chaves de ambiente, resultados históricos de utilização e restrições legais);
- Esquema eficaz de classificação para cada domínio;
- Documentação completa de sistema e de componente.

Poder-se-iam adicionar:

- Política de controle de usuários para o acesso ao repositório;
- Funções de controle de histórico de versões de um item de configuração;
- Manutenção de uma atividade de gerência do repositório, responsável, por exemplo, pelo gerenciamento de usuários, pelo fechamento de versões de produtos ou componentes, pela criação de novos repositórios etc.

A ferramenta de gerência de configuração *Subversion* faz o papel do sistema automatizado de biblioteca de componentes aconselhado por Guo, com suporte a navegação e consumo. A busca é implementada no protótipo de repositório, descrito no capítulo 4, segundo a API sugerida na especificação RAS. O formato padrão de armazenamento de componentes segue o modelo RAS, que define compatibilidade com XMI. A documentação de produtos e componentes é feita dentro dos perfis estabelecido pelo RAS, na forma de requisitos e de projeto de software. O *Subversion* oferece os mecanismos de controle de usuários e histórico de versões. Atividades como averiguar a qualidade de componentes adicionados ao repositório, catalogar e classificar componentes adicionados ao repositório, e gerenciar a adição e utilização de componentes do repositório (FIRESMITH; HENDERSON-SELLERS, 2002), devem ser responsabilidades do bibliotecário de reuso, principal papel a exercer a atividade de gerência, definida no modelo de processo *Komponento*, e detalhada nas próximas seções.

4. O processo *Komponento*

Este capítulo dedica-se à descrição detalhada do modelo de processo de utilização e gerência de um repositório de componentes de software proposto neste trabalho: o *Komponento*. E apresentado, primeiramente, o modelo geral do processo. Em seguida, suas atividades, artefatos, atores, fluxos de atividades, e modelo de ciclo de vida. Finalmente, um exemplo hipotético de aplicação é apresentado, no final do capítulo, a fim de se ilustrar os mecanismos de suporte ao DBC proporcionados pelo processo.

4.1. O modelo de processo

Tendo sido relacionados todos os requisitos do modelo de processo *Komponento*, no capítulo 3, a partir daqui serão descritos todos os elementos do processo, modelados segundo o padrão SPEM da OMG, já citado anteriormente.

Em linhas gerais, o *Komponento* deve fornecer uma infra-estrutura de apoio à reutilização de componentes através de um repositório. É, portanto, centrado num repositório de componentes, peça chave do processo. Daí dizer que o *Komponento* é centrado num repositório. Deve, obviamente, definir regras para as três atividades inerentes a todo processo de reutilização: consumo e produção em uma biblioteca de componentes, e a manutenção (gerência) da mesma.

Apperly (APPERLY, 2001) apresenta um processo de produção-gerência-consumo de componentes, (COTS, componentes sob encomenda, ou extraídos de um legado). A produção

deve ser uma atividade com um razoável nível de documentação. A gerência deve ter um foco em qualidade e disponibilidade do repositório. E o consumo implica em pesquisa, seleção, e utilização de componentes. A tabela abaixo traz um resumo das tarefas de cada uma das três atividades principais do processo de Apperly.

PRODUÇÃO • **Publicar componentes não documentados ou informalmente especificados** • **Utilizar especificação de componentes como ponto de partida para o projeto** • **Publicar componentes com especificações** • **Republicar componentes com especificações** • **Notificar consumidores sobre novos componentes ou problemas**
GERÊNCIA • **Gerenciar repositórios da biblioteca** • **Gerenciar usuários da biblioteca** • **Gerenciar catálogos** • **Garantir componentes de qualidade** • **Gerenciar componentes** • **Disponibilizar componentes** • **Gerenciar históricos de versões de componentes**
PRODUÇÂO • **Pesquisar por componentes necessários** • **Preencher lacunas identificadas** • **Especificar componentes por contrato** • **Reutilizar especificação de componentes** • **Reutilizar componentes** • **Instalar (*deploy*) componentes** • **Registrar interesse em componentes** • **Receber nova notificação de componente** • **Examinar novos componentes** • **Atualizar componentes e especificações**

Tabela 2. Serviços de um gerenciador de componentes (APPERLY, 2001).

O *Komponento* é uma adaptação do processo de Apperly, contribuindo com a descrição de novos elementos. As principais adaptações e contribuições foram:

- As tarefas de cada uma das atividades de produção, gerência e consumo são agrupadas em tarefas maiores, para maior flexibilidade de uso, e para tornar o processo mais eficiente, eliminando formalismos exagerados.

- Baseando-se num meta-modelo, o *Komponento* pode entregar interoperabilidade ao processo, através de uma modelagem uniforme de fluxos de trabalho, atividades, tarefas etc.

- Uma visão da dinâmica do processo pode ser obtida com o emprego de fluxos de atividades UML (para produção, gerência, e consumo). O SPEM, sendo uma extensão do meta-modelo UML, goza dos artefatos e recursos oferecidos pelo padrão.

- Define um modelo de repositório de componentes, baseado na ferramenta de gerência de configuração *Subversion*. Define uma estrutura de armazenamento de componentes, conforme a especificação de ativos reutilizáveis RAS da OMG.

4.1.1. Visão geral

A figura 4.1 dá uma visão geral da dinâmica do *Komponento*, através de um Diagrama de Fluxo de Dados (DFD). Apesar de ser um modelo mais antigo, o DFD é um recurso oportuno

para representar a estrutura das atividades, o fluxo dos produtos e, principalmente, o repositório de componentes, como uma fonte de dados, elemento crucial do processo.

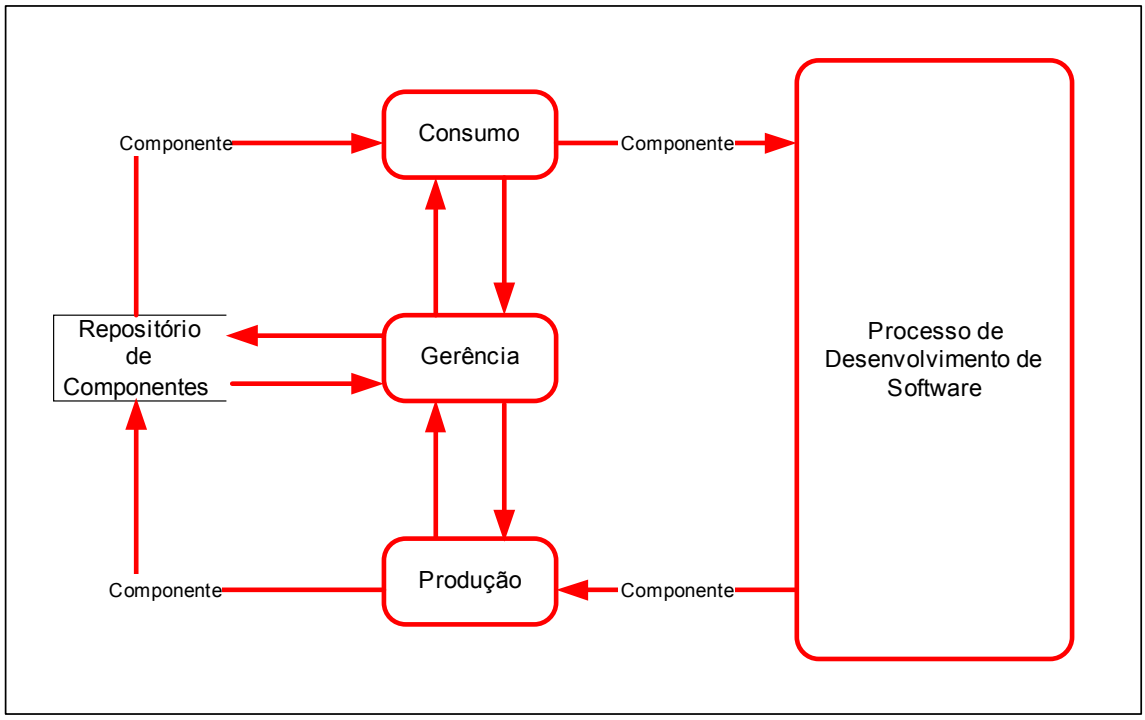


Figura 4.1. Visão geral do Komponento.

Assim, percebe-se que a meta do *Komponento* é dar apoio a processos do ciclo de desenvolvimento de software. Componentes extraídos do repositório, alinhados aos requisitos do sistema a ser desenvolvido, podem ser entradas para as fases de implementação ou projeto (implícitas no Processo de Desenvolvimento de Software da figura 3.7), através da atividade de consumo; componentes criados durante o Processo de Desenvolvimento de Software, por sua vez, podem ser incluídos no repositório de componentes, ao término de cada ciclo ou iteração. A gerência é uma atividade onipresente do *Komponento*. Ela rege as atividades de consumo e produção, bem como define subatividades a serem executadas, regularmente, sem uma ordem específica. Criação de usuários, controle de qualidade, e gerência de catálogos são exemplos de tarefas deste gênero.

As três operações básicas do *Komponento* definem, então, o conjunto de ações que devem ser tomadas, durante um projeto de utilização de um repositório de componentes:

- **Produção.** Esta atividade envolve a publicação de componentes (binários, fontes, *scripts*, bibliotecas) e suas especificações, bem como de todo o tipo de documentação relevante a ser incluída na base de componentes envolvida. A partir das especificações já existentes, outros componentes podem ser desenvolvidos. Usuários devem ser notificados, quando de alguma modificação relevante no repositório.

- **Gerência.** A gerência envolve toda a atividade relativa à manutenção e evolução do repositório de componentes. Deve-se garantir a consistência da estrutura de armazenamento dos componentes, o que pode ser em parte implementado por uma ferramenta de gerência de configuração. Os componentes e produtos devem estar bem catalogados, a fim de se proporcionar meios do usuário encontrar mais rapidamente um componente ou especificação. A catalogação (ou classificação) de componentes é uma tarefa que deverá ser feita por um bibliotecário de componentes, papel definido pelo *Komponento*. O controle de versões é outra prática que deve ser adotada para uma melhor organização e rastreamento de históricos no repositório. Diversas ferramentas de controle de versões, hoje, são comercializadas ou distribuídas gratuitamente. A gerência do repositório deve, por fim, garantir, através de alguma metodologia, a qualidade de seus componentes.

- **Consumo.** São basicamente a procura, seleção e extração de componentes do repositório de acordo com as necessidades do usuário; componentes que possuam o melhor custo/benefício em termos de qualidade ou que simplesmente atendam a um conjunto de requisitos. Consumir componentes muitas vezes requer codificação para integrá-los a outras soluções,

bem como reengenharia ou extensão de funcionalidades. O consumo nem sempre são de binários, executáveis, e bibliotecas, mas também de especificações, para serem reutilizadas na construção de novos componentes. Configuração e instalação de componentes também são tarefas que fazem parte desta atividade, bem como o registro de interesse em recebimento de notificações, por parte de um usuário, quando da alteração ou inserção de determinado elemento.

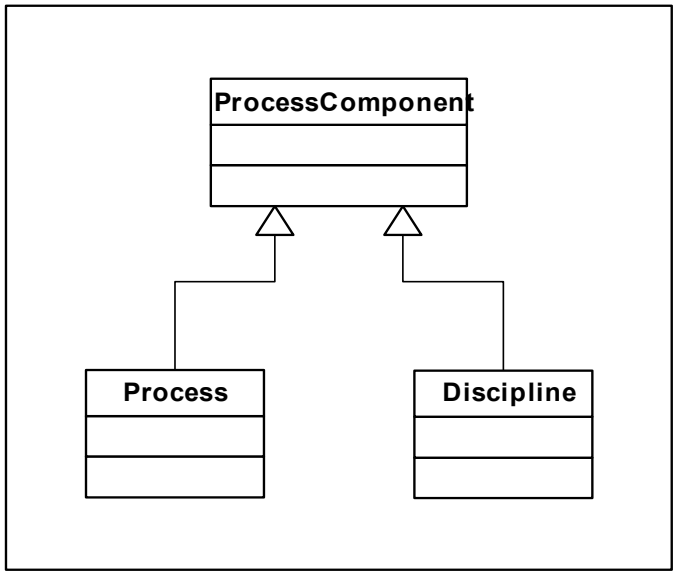


Figura 4.2. Estrutura do pacote ProcessComponent do modelo SPEM.

Na especificação do meta-modelo de engenharia de processo de software SPEM, é descrita a meta-classe *ProcessComponents::ProcessComponent* como "um bloco de descrição de processo, que é consistente, internamente, e que pode ser reutilizado com outros blocos, para montarem um processo completo". A classe *Process* (figura 4.2) herda as características de *ProcessComponent* e significa um processo independente como um todo. *Discipline* é um pacote especializado de atividades, semelhante ao conceito de disciplinas do RUP. Este elemento não é aplicado ao modelo de processo *Komponento*.

Segundo um diagrama de atividades, o *Komponento* poderia ser representado segundo o fluxo da figura 4.3. Nele, vêem-se as atividades de consumo e produção, mutuamente exclusivas.

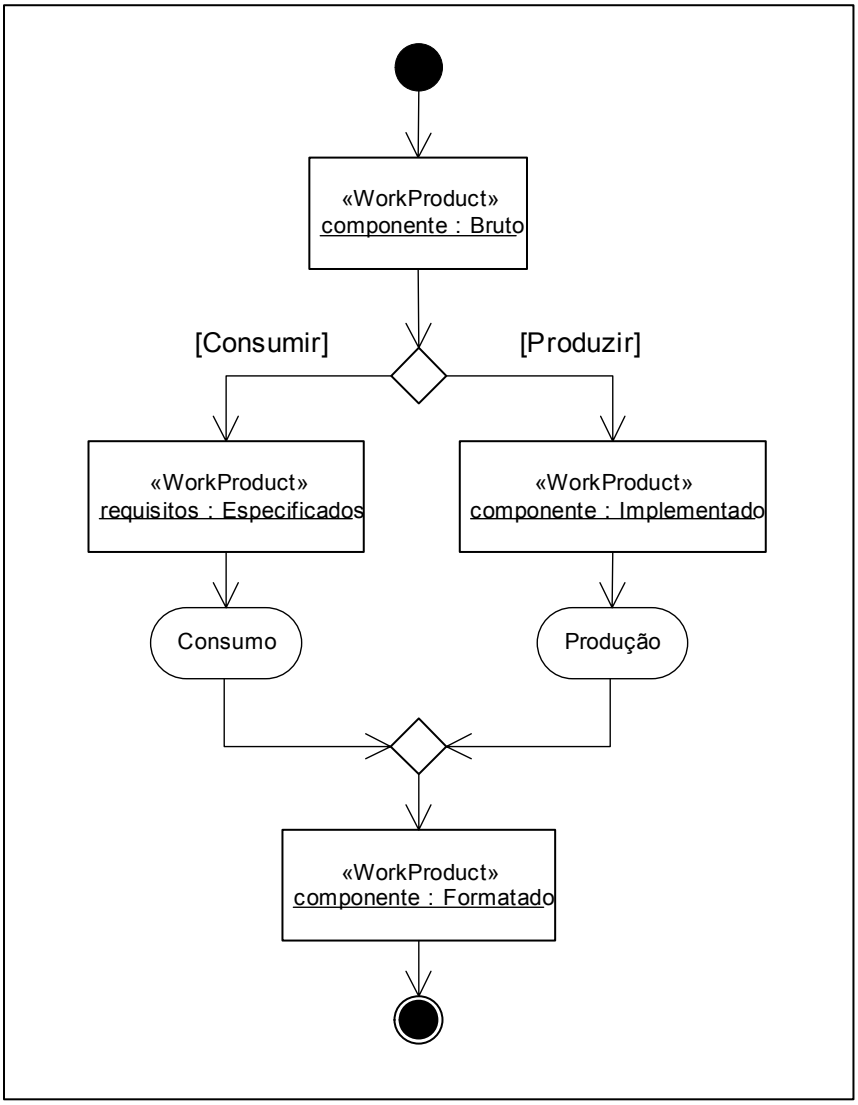


Figura 4.3. O Komponento como um bloco de fluxo de atividades na notação UML.

A atividade de gerência do *Komponento* não é representada no diagrama por não fazer obrigatoriamente parte do fluxo, sendo onipresente no processo de utilização e manutenção do repositório de componentes. Pode ocorrer paralelamente às atividades de consumo e produção, bem como independentemente dos processos aos quais dá suporte o *Komponento*.

Exemplos destas atividades independentes seriam: criação de usuários do repositório, catalogação de componentes, *backup* da base de componentes etc.

Assim, o bloco de fluxo de atividades da figura 4.3 pode ser representado por um *ProcessComponent* do SPEM. As entradas deste bloco de atividades seriam componentes em estado bruto; as saídas, componentes devidamente formatados para serem inseridos no repositório de acordo com um padrão estabelecido, no caso do *Komponento*, o RAS. Assim, um componente em estado bruto, para a atividade de Consumo, seriam suas especificações (formais ou informais). O componente que melhor atendesse os requisitos da especificação seria extraído do repositório. O componente em estado bruto, para a atividade de Produção, seria um produzido após um ciclo ou iteração de desenvolvimento de software, projetado para reutilização e, portanto, candidato ao repositório.

4.1.2. Atividades

As três operações básicas, mencionadas na seção anterior, são representadas como instâncias da classe *Activity* (atividade) do modelo SPEM. Segundo o modelo, a classe *Activity* é especialização de *WorkDefinition*, que implementa uma variação do padrão *composite* (GAMMA, 1995), de acordo com a figura 4.4.

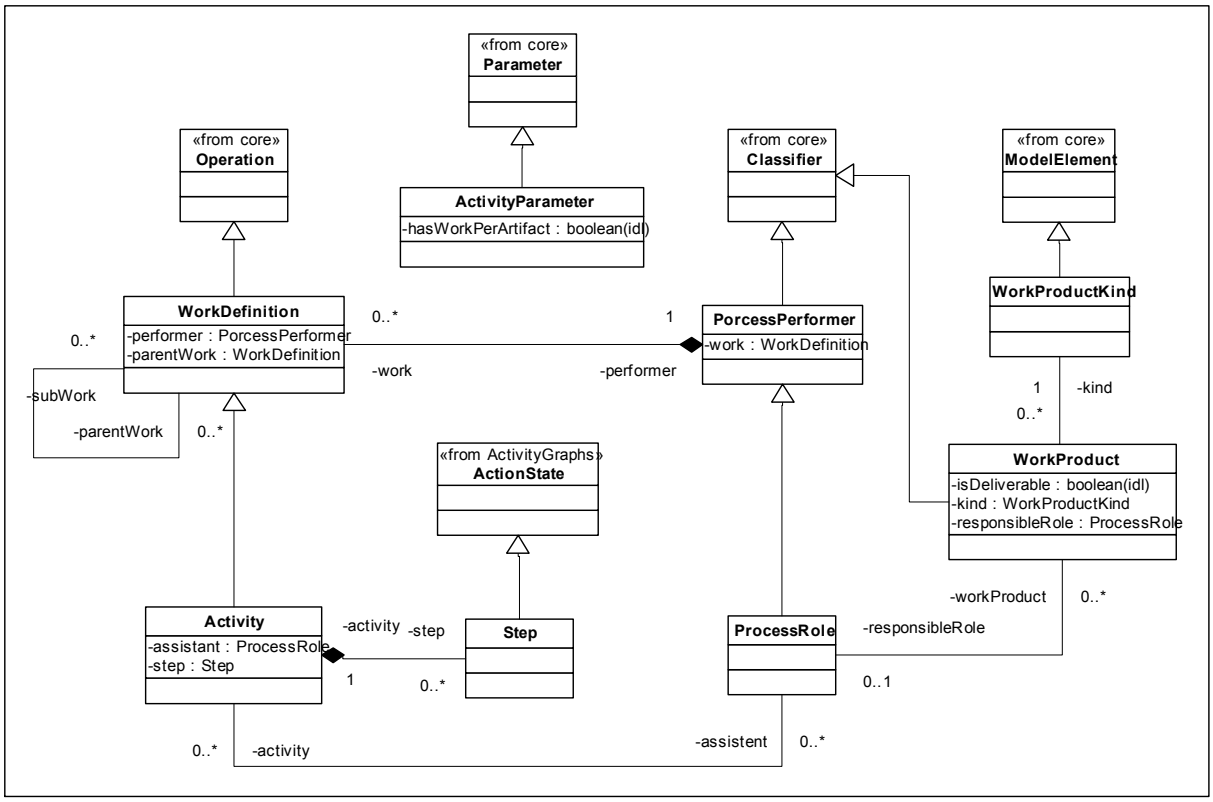


Figura 4.4. Pacote Process Structure do meta-modelo SPEM.

Desta maneira, uma atividade pode ser composta de subatividades, e assim recursivamente, até a decomposição de uma atividade em passos. O termo subatividade será doravante utilizado para expressar atividades que compõem outras atividades, pelo relacionamento *subWork* de *WorkDefinition*.

As atividades de Produção e Consumo (bem como a de Gerência) são, então, instâncias da classe *Activity*, de acordo com a figura 4.5. Ambas são constituídas por passos (*Step*) que devem ser executados segundo fluxos de tarefas específicos. Os fluxos destas atividades estão detalhados na seção 4.1.5.

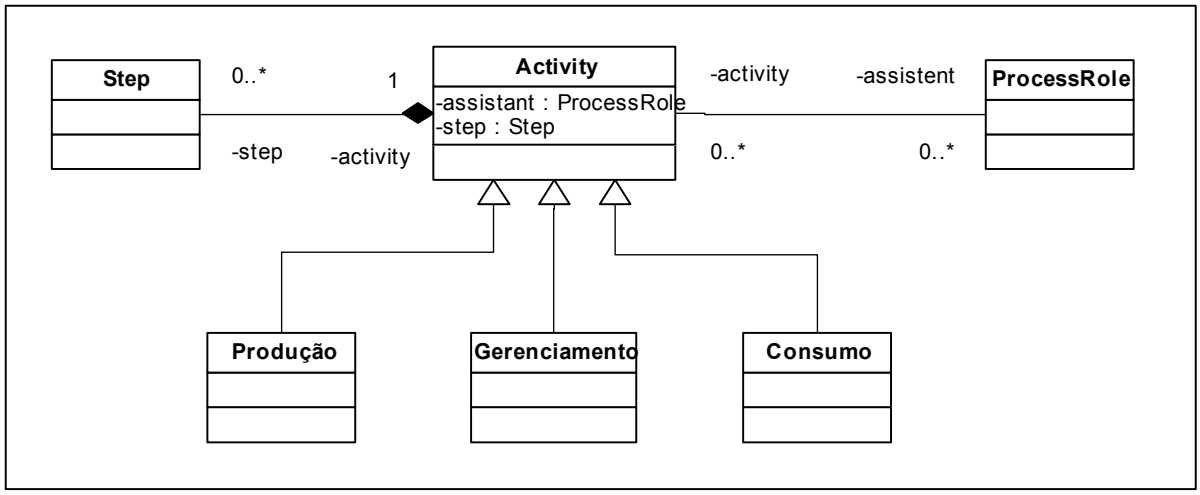


Figura 4.5. As operações básicas derivam diretamente da classe Activity da extensão SPEM da UML para definição de processos.

No caso da atividade de gerência, não existe um conjunto de passos que devem ser executados segundo uma ordem predeterminada. Isto pelo fato de se tratarem de tarefas mais administrativas. A melhor representação da estrutura da atividade de gerência do *Komponento* seria através de subatividades, isto é através do padrão *composite* retratado no relacionamento *subWork* na figura 4.4. O meta-modelo SPEM também prevê papéis (*ProcessRole*) associados a atividades, definindo então responsabilidades. As relações entre atividades e papéis do *Komponento* são abordadas na seção 4.1.4.

4.1.2.1. Produção

Os passos que constituem a atividade de produção de componentes são:

- **Publicar componentes e suas especificações.** Esta atividade define os passos e cuidados que devem ser tomados, durante a publicação de um componente no

repositório, de forma a permitir que usuários venham a consultá-lo, posteriormente, sem maiores problemas. Um componente, neste caso, pode ser um COTS, um componente previamente desenhado e desenvolvido para cumprir determinada, ou um sistema legado remodelado (no nível de código, arquitetura, e/ou especificação) para constituir um novo componente. COTS, por exemplo, ou mesmo componentes reutilizados dentro de uma organização, podem muitas vezes carecer de documentação. É dever do ator, que estiver publicando o componente, ter o cuidado de documentá-lo, a fim de viabilizar seu posterior consumo. A meta-informação do componente no repositório também deve refletir, com a maior fidelidade possível, as funcionalidades do componente, para fins de consumo. A estrutura de diretórios, em que um componente é armazenado, deve ser compatível com um formato padrão (este formato é definido pelos perfis da especificação RAS, abordados no capítulo 4), de forma a se obter consistência no armazenamento de documentos, uniformizando e tornando mais ágil a consulta.

- **Atualizar componentes e suas especificações.** A produção de um componente envolve apenas a primeira publicação do mesmo no repositório. Um software evolui após ser colocado em produção, devido à descoberta de falhas, alterações nos requisitos (especificação), remodelagem etc. Manter um componente atualizado, portanto, é uma tarefa de extrema importância e está estritamente relacionada ao controle de suas versões. O ator, que estiver atualizando um componente, deve controlar as versões dos documentos que estiver modificando. No caso de alteração de código, devem-se propagar as devidas modificações para todos os possíveis elementos afetados (documentos, modelos, bibliotecas, *scripts* etc.).

- **Notificar consumidores.** O consumo de componentes, para se tornar mais eficiente, deve contar com um comportamento pró-ativo por parte do repositório. Assim, o usuário não só deve consultar um componente, na busca de determinada função (para reutilizá-lo dentro de um projeto), como também deve ser notificado pelo repositório, quando da inserção de algum novo componente, ou do lançamento de uma nova versão. Isto pode ser feito, personalizadamente, tendo o usuário registrado interesse em novas versões de um componente, ou de forma genérica, através de envio de correio eletrônico a todos os usuários, quando da publicação de um componente totalmente novo. É interessante que a mensagem também contenha uma estrutura padrão, fornecendo de forma objetiva todas as características relevantes da atualização.

4.1.2.2. Gerência

A atividade de gerência é realizada paralela e independentemente das atividades de produção e consumo. Engloba uma série de tarefas de cunho administrativo, como gerenciar repositórios e usuários. Estas tarefas administrativas são normalmente demandadas, durante a produção ou o consumo de componentes. Desta maneira, quando um membro recém integrado a um projeto necessita, por exemplo, consumir um item do repositório, deve-se criar uma conta e uma senha de acesso para o mesmo, dentro da atividade de controle de usuários; quando, durante um ciclo de desenvolvimento de software, a equipe de desenvolvedores tem que lidar com alterações de requisitos, pode ser necessário uma reformulação na estrutura hierárquica dos itens de configuração, ou seja, pode ser necessário gerenciar um repositório.

As subatividades que constituem a atividade de gerência de componentes são:

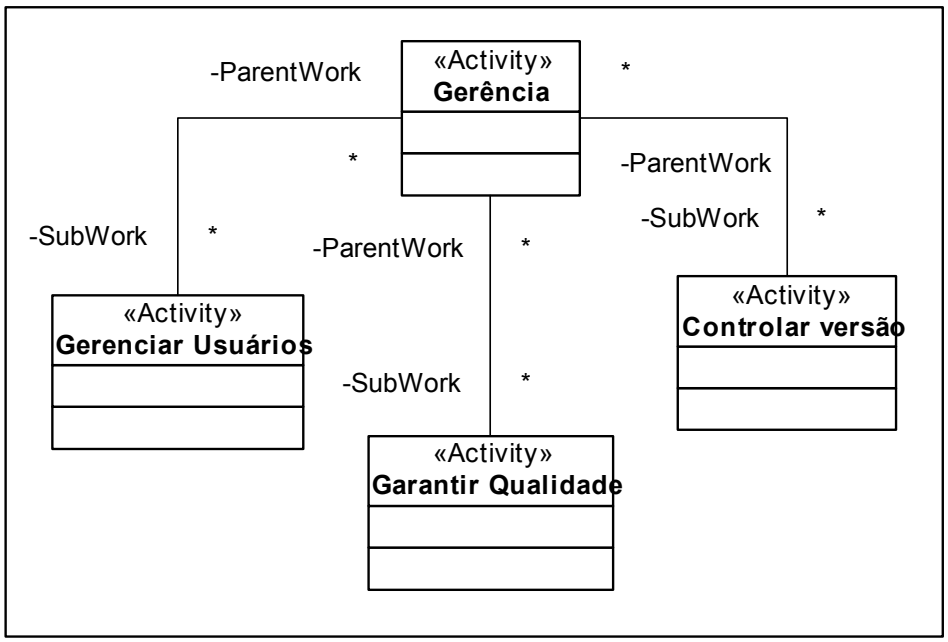


Figura 4.6. Diagrama UML de objetos. As subatividades da atividade de gerência: gerenciar usuários, garantir qualidade e controlar versão.

- **Gerenciar repositório.** O repositório do *Komponento* é implementado com base na ferramenta *Subversion* de gerência de configuração, que automatiza, entre outras funções, controle de usuários e controle de versões. Um protótipo de um sistema de buscas é apresentado no capítulo 4, segundo a especificação de serviços de consulta de ativos (OMG, 2004). Este repositório deve estar instalado como um serviço, e ser gerenciado como tal, sendo reiniciado regularmente para manutenção. O bibliotecário de reuso (sugerido pelo OPF) é o principal responsável por esta atividade, papel dedicado à gerência de um repositório de componentes.

- **Gerenciar usuários.** Políticas de controle de usuários podem variar através de diferentes ferramentas. Conceitos como usuários, grupos, e permissões, no entanto,

estão quase sempre presentes em todas elas, assim como no *Subversion*. Para prover, então, acesso controlado ao repositório de componentes, mais uma vez o bibliotecário de reuso deve criar usuários, atribuir-lhes permissões, se for o caso (muitas vezes atribuem-se permissões aos documentos), e removê-los. De qualquer forma, a gerência de usuários requer intervenção humana, pois cada organização possui sua política própria.

- **Gerenciar catálogos.** Outra subatividade de gerência do repositório é a catalogação de componentes. De acordo com Henninger (HENNINGER, 1997), a maioria dos algoritmos de extração requer uma estrutura pré-definida para que usuários do repositório possam buscar seus componentes. No mínimo, componentes devem ser, então, categorizados, para serem incluídos no repositório. Categorizar componentes é mais uma das tarefas do bibliotecário de reuso, que deve organizá-los, de acordo com uma estrutura de diretórios, de forma a se facilitar o acesso por parte dos consumidores. Esta atividade (subatividade de gerência) interage com as outras duas atividades do *Komponento*. Para se produzir um *komponento*, deve-se saber seu lugar na hierarquia de diretórios (ou de pacotes, ou de qualquer outra estrutura em vigor). O consumo, por sua vez, orienta sua busca pela estrutura do repositório.

- **Garantir qualidade.** Este é um dos requisitos do processo, previamente estabelecido na seção de requisitos. Esta atividade interage tanto com a atividade de produção, na medida em que deve ser constante a preocupação com a criação de artefatos, que auxiliem alguma metodologia de avaliação de qualidade, quanto com a de consumo, na consulta de documentação de atributos não funcionais de um componente ou produto. A garantia de qualidade é constituída de mais duas subatividades: um

processo de avaliação de arquiteturas, baseado na metodologia ATAM (CLEMENTS; KAZMAN; KLEIN, 2002) do SEI, e um processo de certificação de componentes. A seção 4.2 é inteiramente dedicada à descrição desta atividade, por ser um dos enfoques dos requisitos do *Komponento*.

- **Controlar versões.** Esta é uma tarefa quase que integralmente automatizada pela ferramenta de controle de versões *Subversion*. Considerando-se uma ferramenta como um ator do processo, assim como um projetista, um arquiteto, ou um gerente, a atividade de controle de versões, mesmo automatizada, deve fazer parte do processo. É executada, por exemplo, durante o fechamento de *builds*[13] e *releases*[14], ou durante a busca de uma determinada versão para um componente sem compatibilidade com versões mais recentes, por exemplo. A atualização de documentos é feita, então, sob a política de controle de histórico de versões da ferramenta *Subversion*.

4.1.2.3. Consumo

Os passos que constituem a atividade de consumo de componentes são (retratados na figura 4.7):

[13] Segundo (KRUCHTEN, 2000), *builds* são versões operacionais de um sistema ou parte de um sistema, que demonstra um subconjunto de funções e características a serem providas no produto final.

[14] *Releases* são as versões finais de software a serem colocadas em produção.

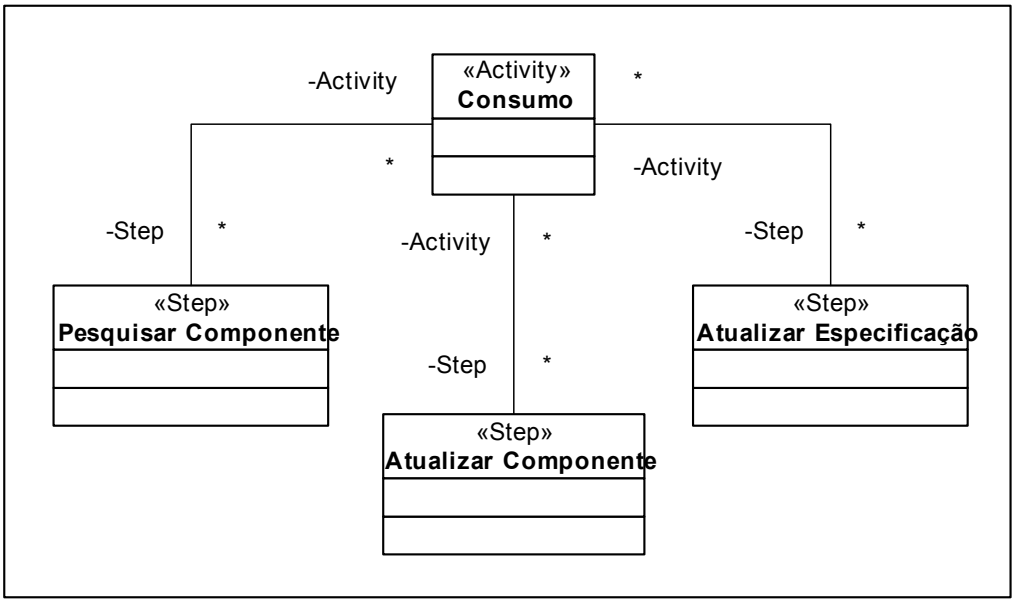


Figura 4.7. Diagrama UML de objetos. Os passos da atividade de consumo: pesquisar componente, atualizar componente e atualizar especificação.

- **Pesquisar componente.** O assunto envolvendo o tema da extração de componentes de um repositório ganhou a atenção da comunidade de reutilização de software. Vários trabalhos são citados por Henninger (HENNINGER, 1997). A pesquisa de componentes é o ato de se buscar num repositório componentes de software que realizem determinadas funções, sejam compatíveis com algum padrão, atendam certos requisito não-funcionais, sejam parte de outros componentes etc. Os critérios são inúmeros. O usuário pode fazer sua busca através da estrutura de catálogos (daí sua importância na gerência), do mecanismo de pesquisa por diferentes critérios, e do recebimento de notificação, bem como do registro de interesse em componentes.

- **Reutilizar componente.** Reutilizar um componente significa reempregá-lo em algum novo projeto, para substituir um outro componente com a mesma interface, ou para compor um novo módulo ou sistema. Reutilizar um componente nem sempre é um tarefa tão simples. Na maioria das vezes requer, além dos esforços de instalação e configuração, esforços de implementação e remodelagem de arquitetura. Alguns

mecanismos de variabilidade (a capacidade de um componente ser reutilizável) em componentes, propostos por Jacobson, e citados em (CLEMENTS; NORTHROP, 2002):

- Herança
- Extensão
- Utilização
- Configuração
- Parametrização
- Instanciação de *templates*[15]
- Geração

A documentação do componente deve descrever os meios para se realizarem tais mecanismos. Certas configurações ou extensões são pouco triviais e quase impraticáveis sem uma documentação razoável.

- **Reutilizar especificação.** Não somente a implementação e a arquitetura de um componente podem ser reutilizados, mas também sua especificação. Se um componente atende genericamente às necessidades funcionais e não-funcionais de consumidores e usuários do repositório, é uma boa prática publicar-se a especificação do componente, para que outros consumidores possam reutilizá-la (e também sua implementação de referência). Aqui se percebe a utilidade da separação entre especificação e implementação.

[15] O termo *template*, em programação, normalmente refere-se a um formato padrão compartilhado por diferentes objetos.

4.1.3. Artefatos

Os artefatos produzidos pelo *Komponento* são todo e qualquer elemento que deva permanecer sob controle de configuração no repositório de componentes. Podem ser modelos, documentos, casos de teste, arquivos de configuração, *scripts*, códigos fonte, bibliotecas, executáveis, e qualquer outro ativo que deva fazer parte de um pacote RAS. Para que não se confundam os termos ativo e componente, deve-se deixar claro que o conceito de ativo de software é mais amplo que o de componente. Ativo é qualquer elemento de software reutilizável. Dentro da classificação de ativos, têm-se as categorias padrões, *frameworks* e componentes. Componente é um conceito mais específico, cujas definições já foram tratadas, no capítulo 2. O esquema de armazenamento previsto na especificação RAS é analisado mais detalhadamente no capítulo 4.

Artefatos são instâncias de *WorkProduct* do meta-modelo SPEM, que define produto em sua especificação, como um objeto que pode ser produzido, consumido, ou modificado por um processo. Um ponto a ser observado aqui é o fato de os artefatos, que podem compor um componente ou produto, serem também instâncias de *Artifact* do RAS (a figura 4.8 ilustra esse polimorfismo dos artefatos do *Komponento*), ao serem armazenados e mantidos no repositório, cuja estrutura está justamente baseada numa extensão de *Default Component Profile* do RAS (OMG, 2004).

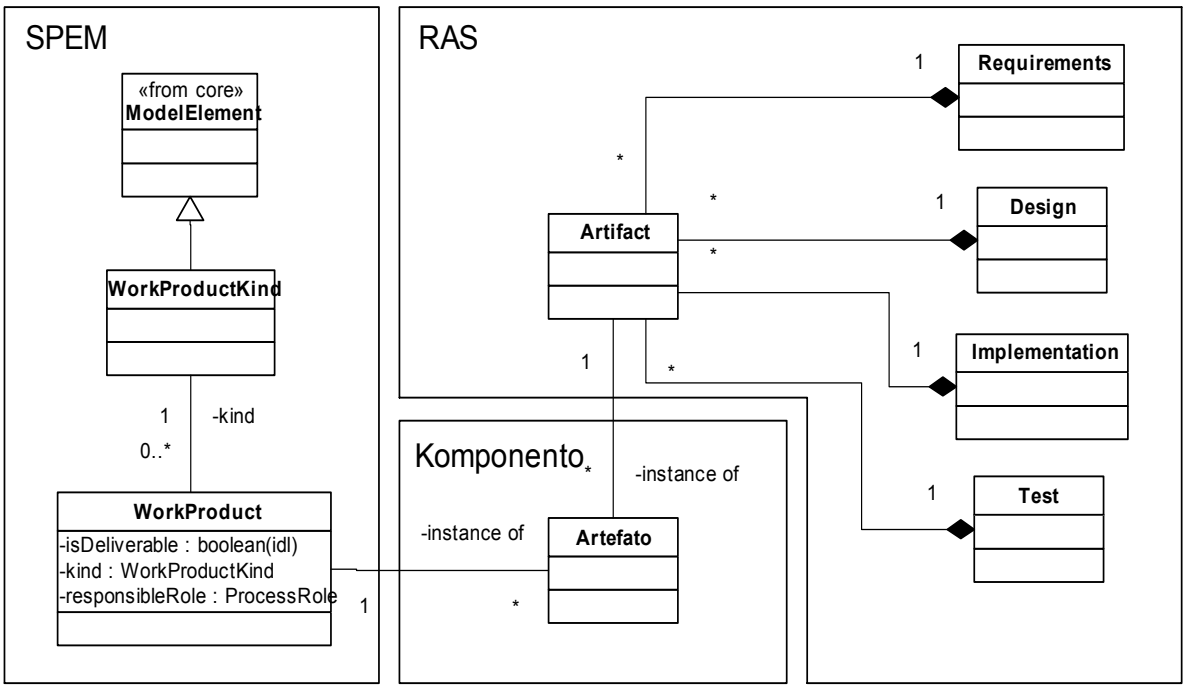


Figura 4.8. Um artefato do komponento é, ao mesmo tempo, instância de WorkProduct (SPEM) e Artifact (RAS).

Dentro da estrutura do *Default Component Profile* (mais bem detalhada no capítulo 5) do modelo RAS, um artefato, que venha a compor uma solução implementada por um determinado ativo, pode fazer parte de quatro grupos (ilustrados ainda na figura 3.18):

- **Requisitos (*Requirements*).** Os requisitos aqui podem ser tanto funcionais como não-funcionais. Um modelo composto por diagramas de classes e seqüência, por exemplo, é um artefato que pode expressar requisitos funcionais de uma arquitetura. Um documento em formato padrão, descrevendo os atributos de qualidade de um componente, com referência para modelos que representem o componente, é um artefato que faz parte dos requisitos não-funcionais de um ativo. Documentos de especificação em geral também entram no grupo de requisitos.

- **Projeto (*Design*).** Integram o projeto de componentes diagramas UML de classes, de seqüência, de objetos, de pacotes, de casos de uso etc. Do projeto também faz parte

outro tipo de artefato essencial para definição de um componente: a especificação De sua interface. A ferramenta Javadoc (SUN, 2005b), por exemplo, é um importante recurso para se documentar as interfaces públicas de classes e pacotes.

- **Implementação (*Implementation*).** Os artefatos da implementação são os códigos fonte, os binários, as bibliotecas, os *scripts*, os arquivos de configuração, enfim, todos os elementos de que um software dependa para ser executado.

- **Teste (*Teste*).** Teste é uma atividade fundamental para validação do desempenho de um componente. Casos de teste são cenários que simulam um determinado software em produção; são um guia para se validar a conformidade de um componente a sua especificação (funcional e não-funcional). Testes podem ser implementados na forma de trechos de código executável, feitos contra uma classe, módulo, ou subsistema. Possuem granularidade variável, podendo ir desde pequenos testes unitários, até grandes testes de integração de componentes.

As instâncias dos artefatos a serem produzidos e consumidos no processo serão especificadas pelo processo ao qual o *Komponento* estiver dando suporte. As regras a serem seguidas para o armazenamento destes artefatos em repositório são dadas pelo RAS.

Dentro da classificação mais genérica de artefatos a comporem um componente (requisitos, projeto, implementação e teste), podemos especificar para as atividades de Produção e Consumo os seguintes tipos de entradas e saídas minimamente necessárias:

	Atividades	
	Consumo	**Produção**
Entradas	• Especificação de requisitos funcionais e não funcionais de sistema (requisitos) • Desenho de arquitetura de sistema (projeto)	• Implementação de componente (implementação)
Saídas	• Especificação de requisitos funcionais e não funcionais de componente (requisitos) • Desenho de arquitetura de componente (projeto) • Implementação de componente testada (implemetação) • Testes de unidade de componente (teste)	• Especificação de requisitos funcionais e não funcionais de componente (requisitos) • Desenho de arquitetura de componente (projeto) • Implementação de componente testada (implemetação) • Testes de unidade de componente (teste)

Tabela 3. Entradas e saídas das atividades de Consumo e Produção do Komponento.

- *Especificação de requisitos funcionais e não funcionais de sistema (requisitos).* Documento contendo os requisitos funcionais (as funções que o sistema deve implementar) e não funcionais (dentro da classificação ISO/IEC 9126) do sistema a ser implementado, seja na forma de documentos em linguagem natural, formal, ou como modelos (ex. diagramas UML, IDEFs, DFDs etc.).

- *Desenho de arquitetura de sistema (projeto).* Modelos refletindo as diversas visões de arquitetura de um sistema. O RUP, por exemplo, sugere uma abordagem de cinco visões. São elas (KRUCHTEN, 1995):

    - *Visão lógica.* A visão lógica suporta basicamente os requisitos funcionais - os serviços que o sistema deve prover a seus usuários. Descreve um modelo de objetos, no caso de uma metodologia orientada a objetos, um diagrama ER, no caso de um modelo de dados etc.

    - *Visão de processo.* Descreve os aspectos de concorrência e sincronização do projeto, levando em consideração requisitos não funcionais do sistema, como desempenho, disponibilidade, integridade e tolerância a falhas.

    - *Visão de desenvolvimento.* Descreve a organização estática do software em seu ambiente de desenvolvimento. O software é empacotado em pequenos blocos – bibliotecas de programas ou subsistemas. Os subsistemas são organizados em uma estrutura hierárquica de camadas.

    - *Visão física.* Descreve o mapeamento do software no hardware e reflete seus aspectos distribuídos. O software executa numa rede de computadores (nós de processamento), e os elementos das demais visões (lógica, de processo e de desenvolvimento) são mapeados nos nós.

- *Especificação de requisitos funcionais e não funcionais de componente (requisitos).* Documento contendo os requisitos funcionais (as funções que o componente deve

implementar) e não funcionais (dentro da classificação ISO/IEC 9126) do componente consumido ou publicado no repositório, seja na forma de documentos em linguagem natural, formal, ou na forma de modelos (ex. diagramas UML, IDEF0, DFDs etc.).

- *Desenho de arquitetura de componente (projeto).* Assim como o sistema possui uma arquitetura e diferentes visões da mesma, o componente, dependendo de sua complexidade, também pode vir a ter uma micro-arquitetura (na realidade, podem ser arquiteturas complexas, no caso de *frameworks* (GAMMA, 1995) de desenvolvimento ou COTS de maior porte).

- *Implementação de componente.* A implementação do componente, propriamente dita, contendo todos os artefatos necessários à execução do mesmo: códigos fonte, bibliotecas, executáveis, *scripts*, arquivos de configuração etc.

- *Implementação de componente testada.* A implementação do componente testada, unitária e integralmente, contra os casos de teste, de maneira informal.

- *Testes de unidade de componente.* Casos de teste, com cenários contra os quais devem ser testados funcionalidade e nível de qualidade do componente consumido ou publicado no repositório. Testes de unidade na forma de componentes a serem executados para validação de desempenho e conformidade à especificação funcional do componente.

Os artefatos produzidos pela atividade de gerência, por sua vez, podem ser:

- *Registro para controle de versões (design).* Podem ser documentos padrão para registrar a inserção ou atualização de uma nova versão na base de componentes, contendo informações como data, hora, usuário, principais modificações, número da versão etc. No *Subversion*, por exemplo, existe a opção de se formatar estes dados no conteúdo do próprio documento sob controle de versão.

- *Manual de operação.* É importante que se tenha um manual de operações da ferramenta instalada para implementar o repositório de componentes. Este manual personalizado deve conter informações do tipo: servidores em que o software se encontra instalado, procedimentos rotineiros essenciais à manutenção do sistema, principais configurações, mapa de repositórios e catálogos etc. O objetivo disto é o de não se concentrar conhecimento em pessoas sobre o correto funcionamento do repositório de componente (peça central do processo), e sim no processo, agregando-o na forma de documentos aos artefatos do processo.

- *Árvore de atributos de qualidade.* A árvore de atributos de qualidade também é peça fundamental no processo de garantia de qualidade do *Komponento*. É implementada na forma de documentos padronizados, refletindo a classificação de atributos de qualidade de software da norma ISO/IEC 9126 (maiores detalhes na seção 4.2.2).

4.1.4. Atores

Os papéis desempenhados pelos atores do *Komponento* também devem estar alinhados ao processo em questão. Produtores de componentes podem ser desenvolvedores, projetistas, e arquitetos, por exemplo. Consumidores são um universo ainda mais extenso, podendo envolver (incluindo os produtores já mencionados) desde usuários e operadores, até gerentes de projeto.

Dadas as amplas possibilidades de atores participantes do processo de utilização de um repositório de componentes, vem a necessidade, então, de se ter um acesso controlado ao mesmo, através de políticas de permissões na base de componentes. Assim, um determinado componente pode ser publicado por um arquiteto ou projetista, modificado por desenvolvedores de um determinado projeto, mas apenas consultados por membros de outro.

A figura 4.9 apresenta possíveis atores, retirados de (FIRESMITH; HENDERSON-SELLERS, 2002), de um processo de desenvolvimento de software que esteja utilizando o *Komponento* para consumo e produção de componentes. Para as atividades de Consumo e Produção de componentes, os papéis definidos pelo modelo do *Komponento* são:

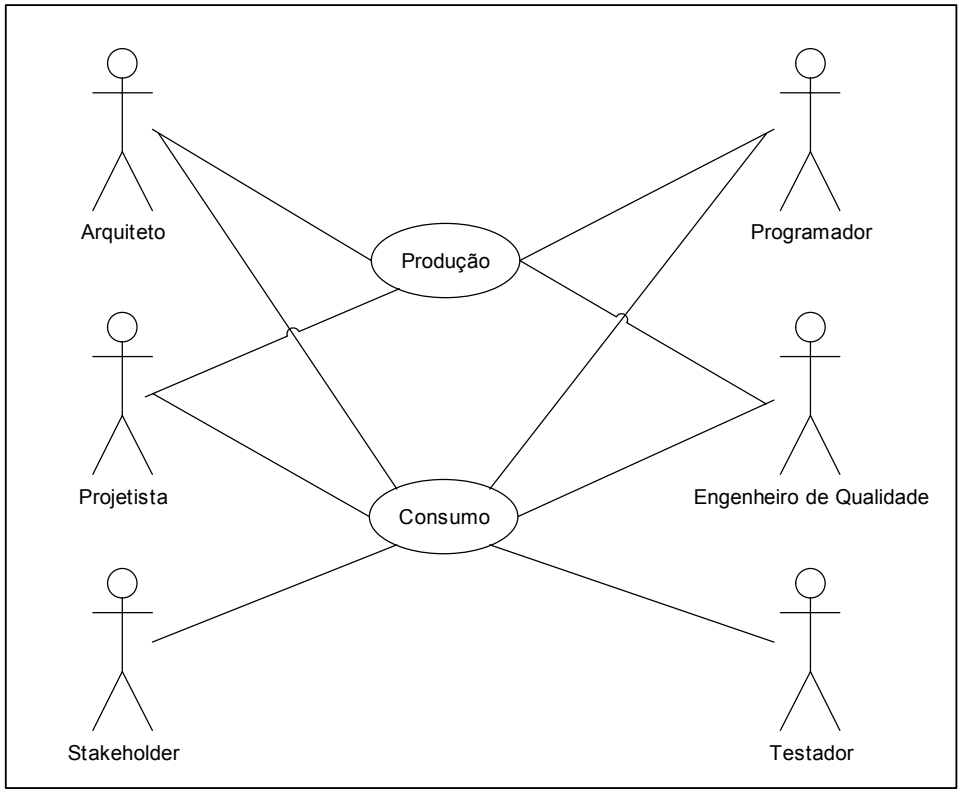


Figura 4.9. Possíveis atores das atividades de Produção e Consumo de componentes de um repositório.

- **Consumidores.** São os participantes de um projeto interessados na reutilização de algum componente que atenda suas necessidades. Executam as tarefas pertinentes à atividade de consumo, como pesquisa na base de componentes, reutilização de um componente (através de herança, extensão, utilização, configuração, parametrização etc.) e de suas especificações. Consumidores são, normalmente, arquitetos, projetistas e desenvolvedores de software, mas podem ter suas funções eventualmente realizadas por um gerente de projeto ou operador de sistema.

- **Produtores.** Este papel estabelece as responsabilidades relativas à atividade de produção. A inserção de um componente totalmente novo pode ser proveniente do desenvolvimento interno de software, da reutilização de sistemas legados, da

aquisição de COTS, ou da encomenda de produtos para terceiros. Estas atividades podem ser desempenhadas, respectivamente, por desenvolvedores, arquitetos, projetistas, e gerentes de projeto. A atualização de componentes, além da implementação propriamente dita, pode envolver também a documentação dos mesmos, refletindo uma alteração de requisito, uma funcionalidade extra, ou um novo parâmetro de desempenho a ser incluído na especificação.

Em relação à atividade de gerência, dois papéis, também sugeridos pelo OPF (FIRESMITH; HENDERSON-SELLERS, 2002), são estabelecidos pelo modelo de processo *Komponento*:

- **Gerente de configuração.** Responsável pela identificação dos itens de configuração, pelo controle de versões destes itens, pela gerência das modificações feitas na base de componentes, incluindo requisições de atualização, aprovação de tais requisições, e análise do impacto das atualizações em outros itens de configuração.

- **Bibliotecário de reuso.** Em abordagens CBD, segundo o OPF, este papel é de extrema importância. O bibliotecário de reuso deve: garantir que os componentes publicados no repositório satisfaçam determinados padrões de qualidade; catalogar e classificar tais componentes; e gerenciar tanto a adição (produção), quanto a utilização (consumo) da biblioteca de componentes.

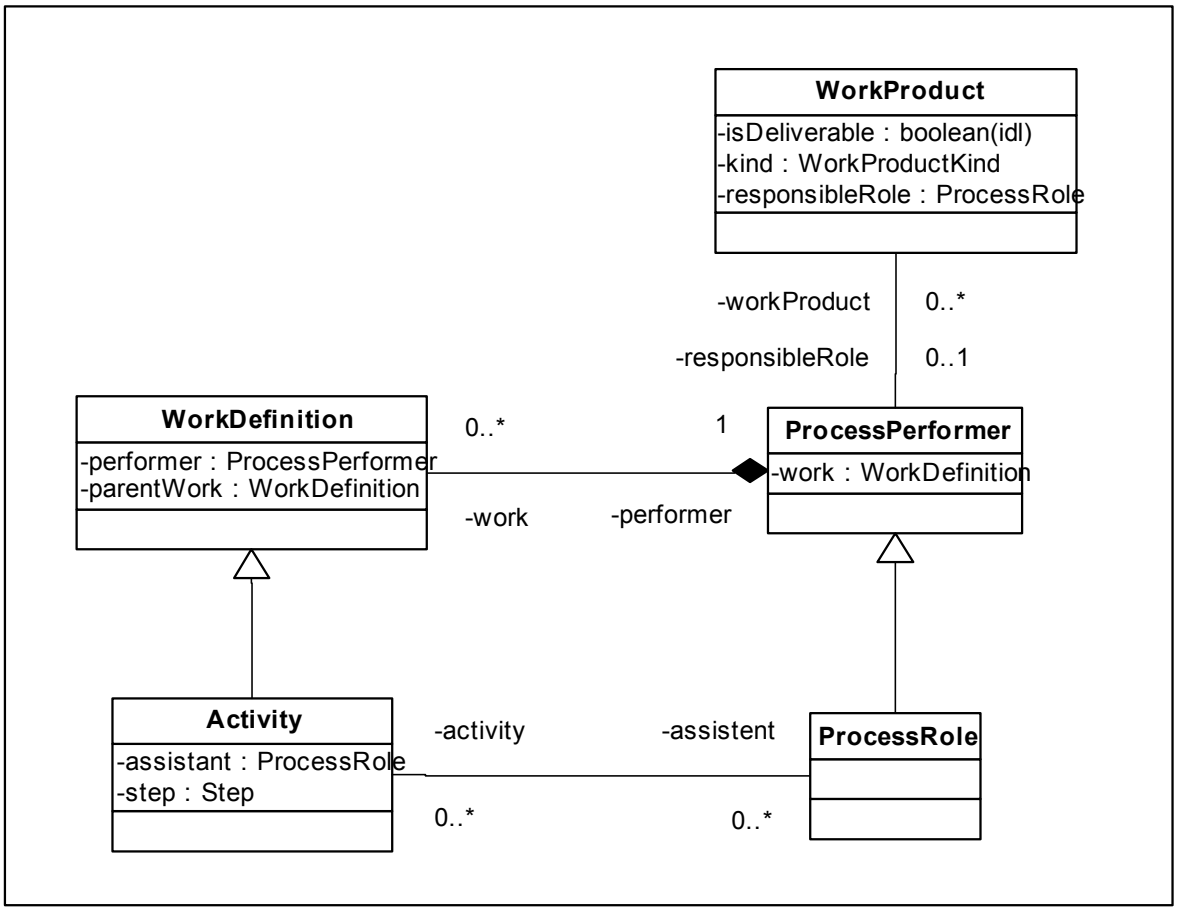


Figura 4.10. Os relacionamentos entre Activity e Process Role, e ProcessPerformer e WorkProduct do meta-modelo SPEM (OMG, , 2005).

Em grandes projetos, há a necessidade de se estabelecer papéis muito bem definidos, para uma melhor utilização de recursos. Um processo, desta forma, ao definir atores, auxilia a distribuição de responsabilidades entre os participantes de um projeto, promovendo um maior nível de organização. Em pequenas organizações, entretanto, o número de membros de uma equipe responsável por determinado projeto é limitado, e muitas vezes ocorre uma intercalação de papéis. Nestas organizações, é comum, então, um mesmo indivíduo desempenhar diferentes funções, tendo a definição de papéis por parte de um processo, uma importância secundária.

É comum, em diferentes modelos de processo, definir-se uma relação direta entre um papel e uma atividade, com o papel (*ProcessRole*) sendo responsável pela atividade (*Activity*). O OPEN define este conceito através da composição das classes *Producer* e *WorkUnit* em um terceiro elemento, o *TaskPerformance*. Define também um relacionamento direto entre *Task* e *WorkUnit*, o que não acontece no SPEM, que diferentemente define um relacionamento direto entre o papel (*ProcessPerformer*) e o produto (*WorkProduct*), conforme ilustrado na figura 4.10. Obviamente, as atividades de produção e consumo de componentes são desempenhadas, respectivamente, pelos papéis de produtor e consumidor. Os relacionamentos entre as subatividades de gerência, e o gerente de configuração e o bibliotecário de reuso, definindo as responsabilidades de tais papéis.

4.1.5. Fluxos de atividades

Assim como o RUP define para cada uma de suas disciplinas (gerência de projeto, modelagem de negócio, requisitos, análise e projeto, implementação, teste, gerência de configuração e mudança, ambiente, e instalação) fluxos de atividades específicos, o *Komponento* também assim o faz, estabelecendo fluxos de tarefas bastante simples, para as atividades de Produção e Consumo de componentes do repositório, baseando-se nas subatividades relacionadas na seção 4.1.2.

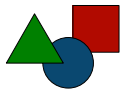
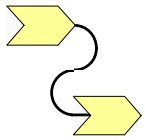
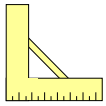
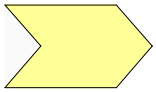
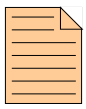
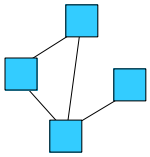
Estereótipo	Classe Base	Estereótipo Pai	Notação
WorkProduct	Core::Class ActivityGraphs::ObjectFlow State		
WorkDefinition	Core::Operation ActivityGraphs::ActionState UseCases::UseCase		
Guidance	Core::Comment		
Activity	Core::Operation ActivityGraphs::ActionState UseCases::UseCase	WorkDefinition	
ProcessPerformer	UseCases::Actor		
ProcessRole	UseCases::Actor	ProcessPerformer	
Document	Core::Class ActivityGraphs::ObjectFlow State	WorkProduct	
UMLModel	Core::Class ActivityGraphs::ObjectFlow State	WorkProduct	

Tabela 4. Proxies de notação dos elementos SPEM, estendidos dos pacotes Core, ActivityGraphs e UseCases da UML.

O SPEM, sendo um *UML Profile*, define uma série de estereótipos através de ícones para simbolizar seus diferentes elementos (tabela 4). Estes elementos podem ser representados, uniformemente, em diferentes diagramas UML. Para o caso do diagrama de atividades, a classe *WorkPoduct* aparece como instância da classe *ObjectFlowState*, e as classes *Activity* e *WorkDefinition* aparecem representadas por instâncias de *ActionState*, elemento do diagrama UML de atividades (assim como *ObjectFlowState*). Assim, os ícones descritos na tabela acima podem ser entendidos como *proxies* (GAMMA, 1995) de notação de suas classes base. As classes *WorkDefinition* e *Activity* seriam, então, *ActionsStates* em diagramas de atividades descrevendo fluxos de tarefas de um processo. *WorkProduct* seria, por sua vez, um *ObjectFlowState*, ambos compartilhando um mesmo *proxy* notacional.

Da mesma forma com que papéis (*ProcessPerformer* e *ProcessRole*) e atividades (*WorkDefinition* e *Activity*) são mapeados em elementos do diagrama de atividades da UML, eles também o são para elementos do diagrama de casos de uso. A tabela 4 traz tal informação, na coluna "Classe Base", através do pacote *UseCases*.

Desta maneira, os fluxos de tarefas das atividades de Produção e Consumo, podem ser definidos através de diagramas de atividades UML e dos estereótipos do SPEM. As tarefas (subatividades) são as descritas na seção 4.1.2, para cada uma das operações do *Komponento*. Para Produção e Consumo, têm-se então os fluxos de tarefas apresentados a seguir.

A atividade de Produção é composta pelos seguintes passos ou tarefas:

- Publicar componentes e suas especificações
- Atualizar componentes e suas especificações

- Notificar consumidores

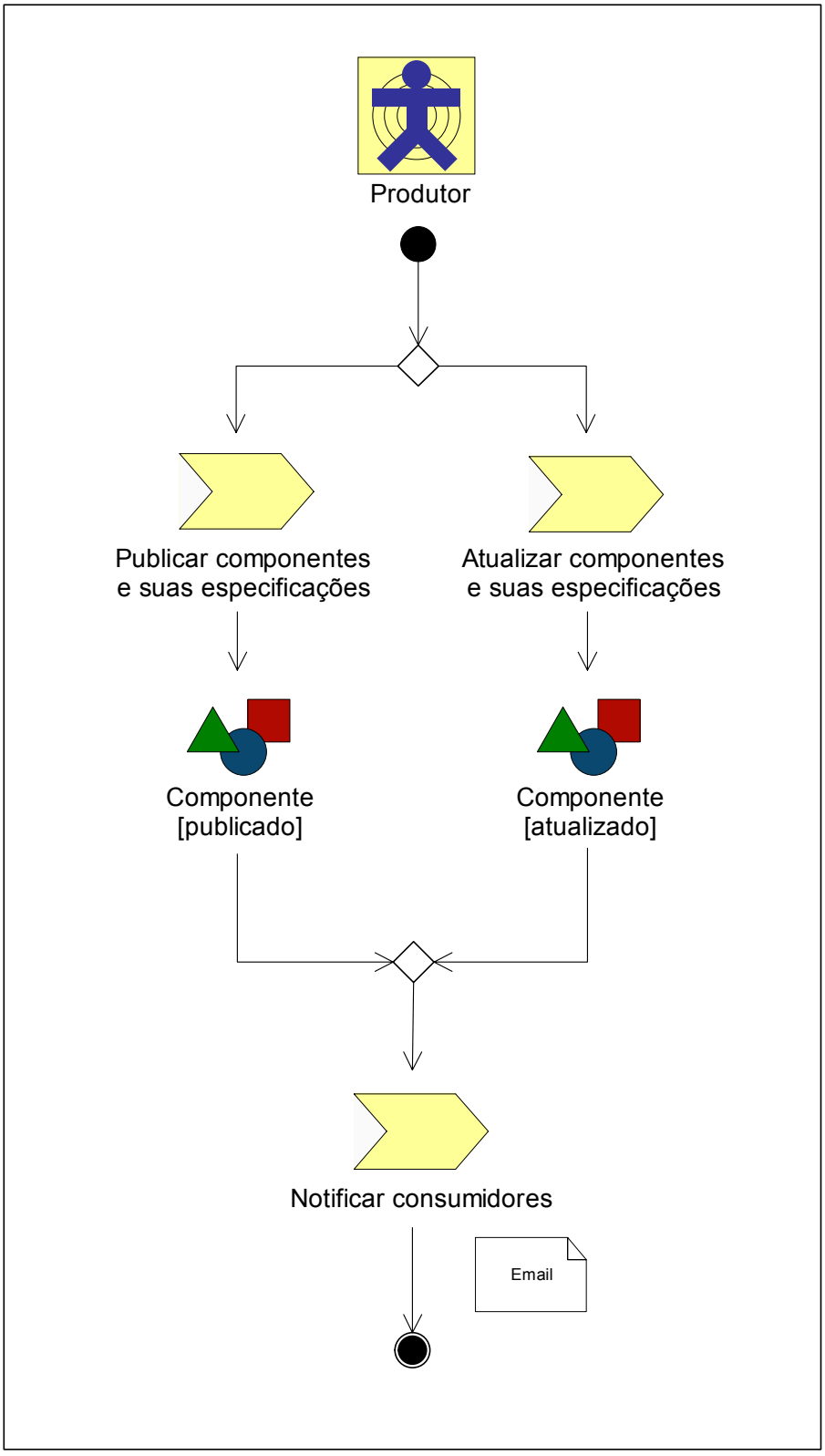


Figura 4.11. Fluxo da atividade de Produção de componentes.

Pela figura 4.11, percebe-se que o produtor é o indivíduo responsável pela publicação ou atualização de um ativo na base de componentes. Pode ser qualquer um dos atores mencionados na seção anterior, de acordo com o processo envolvido. O produto gerado pela publicação e atualização de um componente é, respectivamente, o próprio componente em sua primeira versão, e o mesmo com algum (ou alguns) de seus artefatos modificados pelo produtor, devidamente armazenado no repositório, dentro do modelo RAS. Em seguida, a operação é notificada aos usuários que registraram interesse no componente em questão, por correio eletrônico, ou qualquer outro mecanismo de mensagens, ao estilo do padrão *Observer* (GAMMA, 1995).

A atividade de Consumo é constituída pelos seguintes passos ou tarefas:

- Pesquisar componente
- Reutilizar componente
- Reutilizar especificação

De acordo com a figura 4.12, a atividade de consumo é sempre iniciada com a pesquisa de algum componente no repositório, pelo consumidor interessado. Tendo o sistema de busca encontrado algum componente ou produto no repositório, satisfazendo os critérios de busca entrados pelo consumidor, o componente pode ser então reutilizado num projeto, sofrendo os devidos ajustes (remodelagem, configuração etc.); ou apenas a especificação do componente pode ser reutilizada. Uma especificação, ao possuir diferentes implementações, confere ao componente a capacidade de ser substituível, em diferentes contextos.

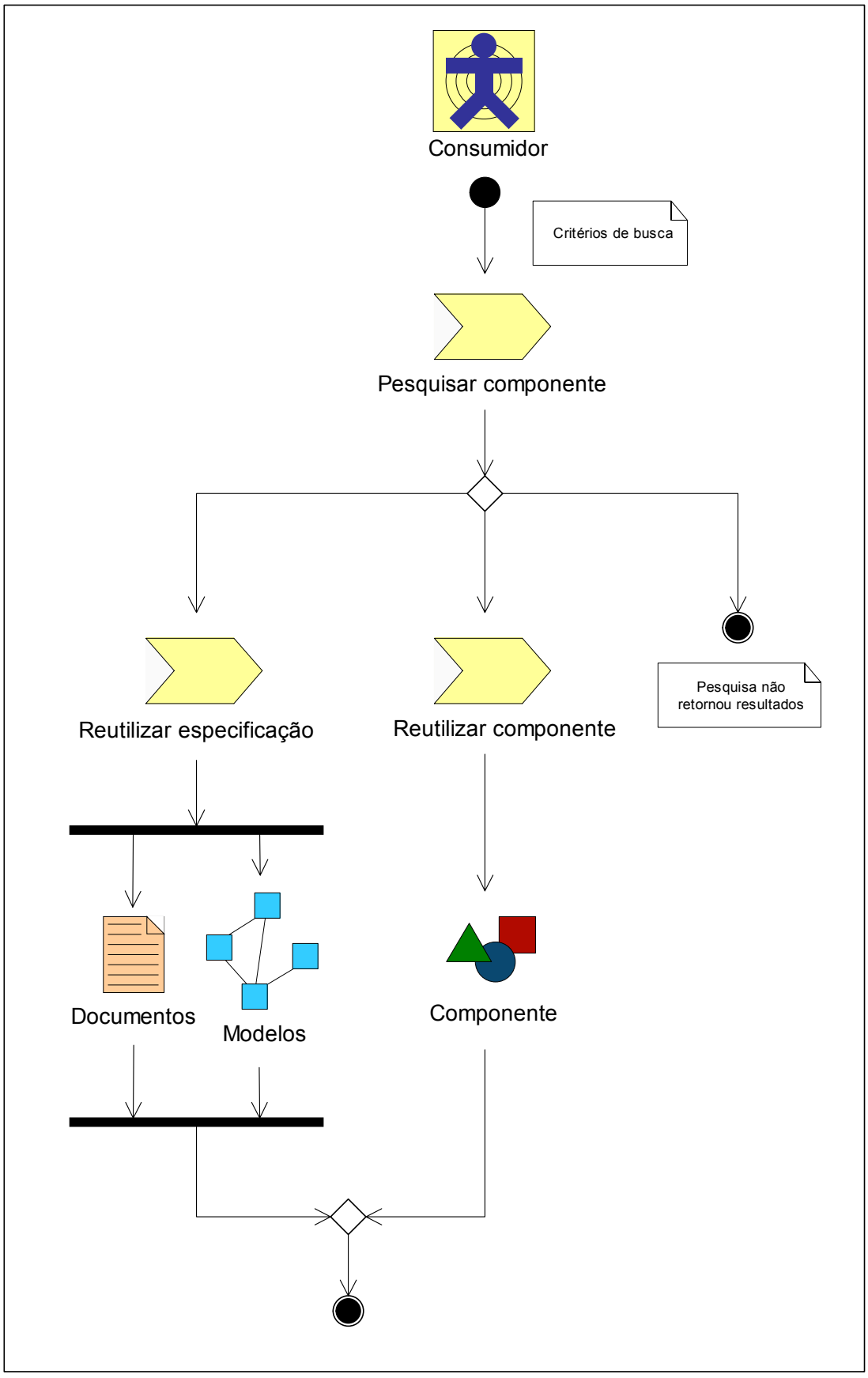


Figura 4.12. Fluxo da atividade de Consumo de componentes.

A atividade de gerência não possui um fluxo de tarefas específico, e suas tarefas são executadas pelo gerente de configuração e pelo bibliotecário de reuso, papéis estabelecidos pelo *Komponento*, já explicados em seções anteriores. Mais uma vez, as tarefas (subatividades segundo o SPEM) de gerência do repositório de componentes são:

- Gerenciar repositório
- Gerenciar usuários
- Gerenciar catálogos
- Garantir qualidade
- Controlar versões

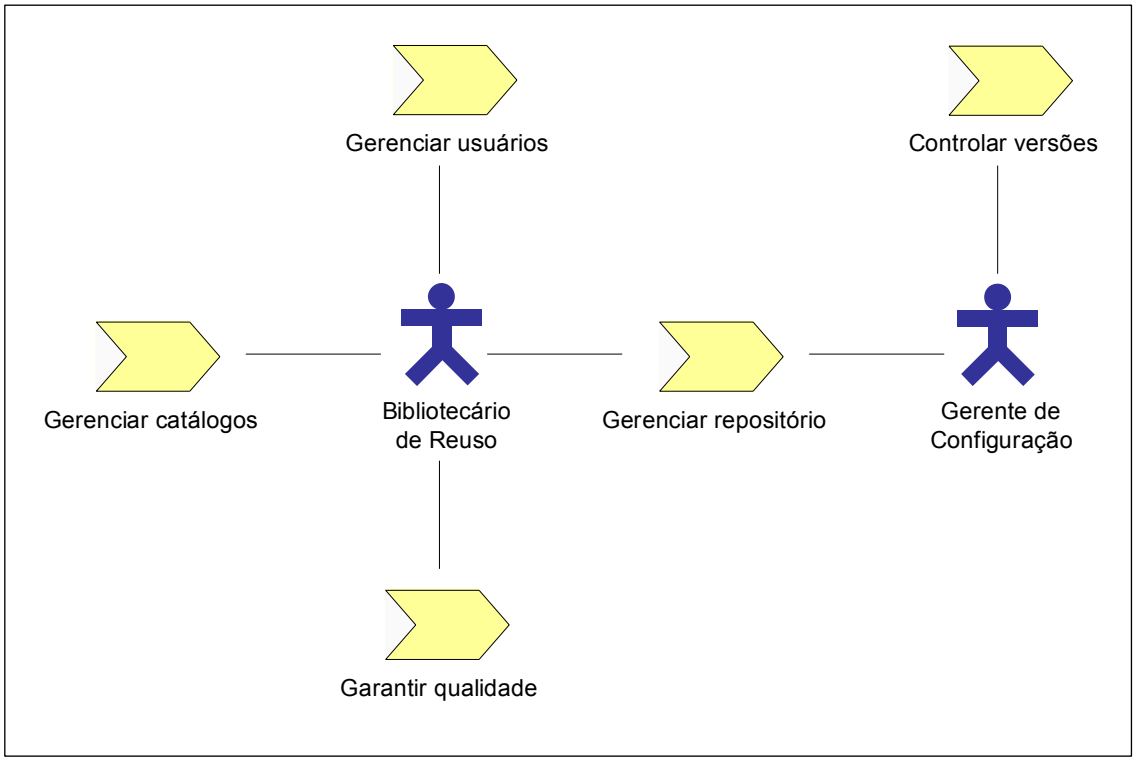


Figura 4.13. Diagrama de casos de uso para a atividade de gerência.

As subatividades de Gerência não se relacionam através de um fluxo de controle, como as tarefas das atividades de produção e consumo. Relacionam-se pelo fato de fazerem parte da gerência do processo. São executadas pelo bibliotecário de reuso e pelo gerente de configuração ao longo de todo o ciclo de vida do processo. A atividade de gerência, então, é mais bem representada através do caso de uso da figura 4.13.

4.1.6. Ciclo de vida

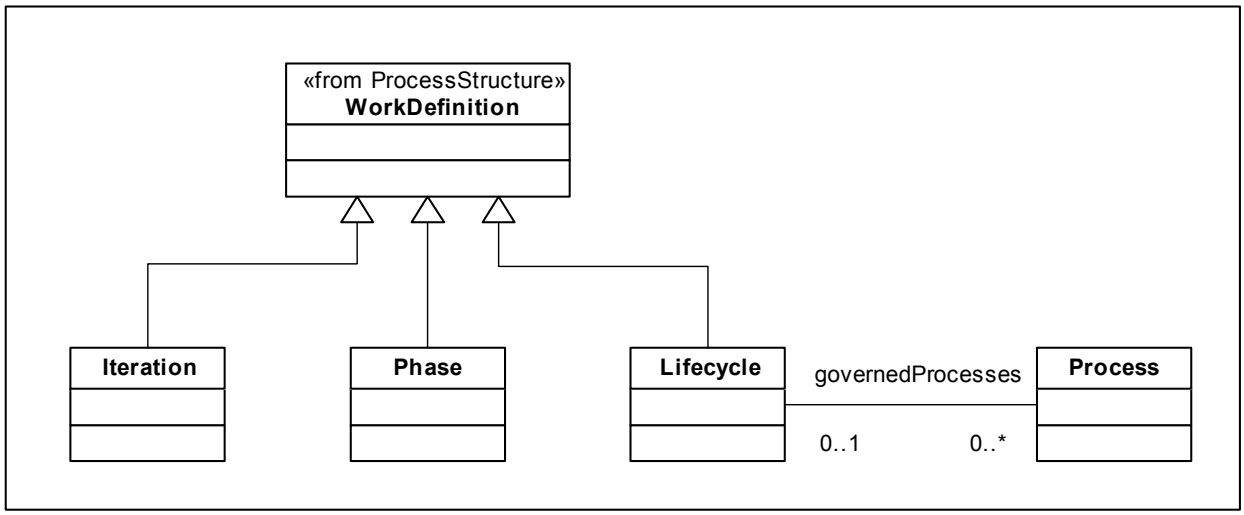


Figura 4.14. O ciclo de vida (Lifecycle) de um processo, segundo o modelo SPEM.

O ciclo de vida do *Komponento*, devido a sua característica acoplável, reflete o ciclo de vida do processo a que dá suporte. O meta-modelo SPEM especializa a classe *WorkDefinition* nas classes *Iteration*, *Phase*, e *Lifecycle*, conforme ilustrado na figura 4.14. O exemplo de aplicação do *Komponento* da seção 4.3 torna mais clara a estrutura do ciclo de vida do processo.

4.2. Garantia de Qualidade

Por ser um dos focos do processo proposto neste trabalho, dedicou-se esta seção inteira à atividade de garantia de qualidade (subatividade de gerência). No *Komponento*, garantia de qualidade é implementada através de duas subatividades, certificação de componentes e avaliação de arquiteturas, descritas a seguir.

A certificação valida funcionalidade e desempenho de componentes, através da aplicação de um plano de testes, especificado em um documento de casos de teste. A avaliação de arquiteturas de componentes se dá através da metodologia ATAM, com iterações de análise e priorização das principais abordagens de arquitetura. Ambas as tarefas, por fazerem parte da subatividade de garantia de qualidade da atividade de gerência, podem ser executadas ao longo de todo o ciclo de vida do *Komponento*. O processo de certificação, assim, pode ser aplicado juntamente com a produção de componentes, desenvolvidos internamente, para serem depois comercializados. Pacotes de teste são ferramentas para se avaliar de forma objetiva um produto. O método de análise de arquiteturas, por sua vez, pode ser executado, por exemplo, durante a seleção de COTS (LIU, 2003).

4.2.1. Certificação de componentes

De acordo com o *IEEE Standard Glossary of Software Engineering Terminology*, "certificação é uma garantia escrita, uma demonstração formal, ou o processo de se confirmar que um sistema ou componente cumpre com seus requisitos especificados e é aceitável para uso operacional". Esta confirmação poderia ser uma assinatura num relatório

ou formulário de sumário de testes, uma marca aplicada ao produto, ou um certificado acompanhando o produto (FLYNT; DESAI, 2001).

A primeira técnica de certificação, citada no parágrafo anterior, baseada na homologação de relatórios de testes, encontra apoio nos trabalhos de Morris e Bertolino. Morris defende a idéia de uma certificação implementada por casos de teste, projetados pelos desenvolvedores dos componentes, para serem validados pelos próprios usuários que vierem a adquirir os componentes (MORRIS, 2001). Esta abordagem teria, segundo Morris, diversas vantagens sobre os SCLs (*Software Certification Laboratories*) – agências especializadas independentes responsáveis pela execução de processos de certificação de software – como redução de custos, confiança garantida, conformidade confirmada, requisitos funcionais adicionais, e valor agregado ao componente de software.

Bertolino e Polini (BERTOLINO, 2003) citam que um dos principais obstáculos à evolução do DBC é justamente a falta de confiança dos usuários de componentes (organizações que adquirem componentes para integrá-los a um sistema maior) em quem os desenvolve (organizações que desenvolvem e lançam componentes). Este problema de confiança estaria, então, fortemente relacionado com a capacidade dos usuários de validar a adequação de um componente candidato a atender seus propósitos. Bertolino propõe, então, um conjunto comum de testes, para tanto usuários quanto desenvolvedores poderem avaliar seus componentes. Yacoub, Ammar e Mili (YACOUB; AMMAR; MILI, 1999) defendem a idéia de que é essencial que componente disponibilizem seus códigos para verificação e teste por parte de seus consumidores, aumentando o nível de confiança.

De acordo com Hausen (HAUSEN, 1992), software pode ser avaliado e certificado pela inspeção de seu processo de engenharia ou pela avaliação de suas características. No primeiro caso, tem-se certificação de processo; no segundo, de produto. O *Komponento* visa avaliar a qualidade de um produto (componentes de software), e segue a filosofia da utilização de casos de teste para se certificar a qualidade de um componente.

Casos de teste apesar de não cobrirem boa parte de aspectos não funcionais de software, como confiabilidade, usabilidade, manutenibilidade ou portabilidade, dão uma garantia concreta em relação à prova das funcionalidades que devem ser implementadas e às metas de desempenho que devem ser cumpridas. O processo de certificação do *Komponento* envolve um conjunto de testes documentados em diferentes cenários para se validar a conformidade de um componente em relação a sua especificação. Os documentos de teste (casos de teste) fazem parte do corpo de artefatos de um componente, sob controle de versão, e correspondem aos Testes de Unidade de Componente, descritos na seção 4.1.3.

Desta maneira, de acordo com os resultados da execução dos casos de teste, para cada um dos cenários, o componente pode ser certificado ou não. A certificação pode ser registrada em níveis discretos ou, simplesmente, aprovar ou reprovar um componente, de acordo com o resultado dos testes.

```
[ojb] ============================================================
[ojb]         OJB PERFORMANCE TEST SUMMARY
[ojb] 10 concurrent threads, handle 2000 objects per thread
[ojb]         - performance mode
[ojb] ============================================================
[ojb]      API  Period   Total  Insert   Fetch  Update  Delete
[ojb]            [sec]   [sec]  [msec]  [msec]  [msec]  [msec]
[ojb] ------------------------------------------------------------
[ojb]     JDBC   7.786   7.374    3951      76    2435     911
[ojb]       PB   9.807   8.333    5096     121    2192     922
[ojb]     ODMG  19.562  18.205    8432    1488    5064    3219
[ojb]      OTM  24.953  21.272   10688     223    4326    6033
[ojb] ============================================================
[ojb] PerfTest takes 191 [sec]
```

Tabela 5. Resultado de testes de desempenho para diferentes componentes de mapeamento objeto-relacional (WAIBEL, 2004).

Os dados da tabela 5 representam um sumário de testes de desempenho para os diferentes componentes da ferramenta OJB de mapeamento objeto-relacional. Os componentes JDBC, PB, ODMG e OTM são camadas de persistência de objetos em bases de dados. São medidos vários tempos para o mesmo conjunto de operações na base de dados (*insert*, *fetch*, *update* e *delete*, conforme a tabela) executado por cada um dos quatro componentes. Os resultados são apresentados na forma de um relatório. Estabelecendo-se, então, parâmetros de tempo e uma configuração definida de hardware, é possível se avaliar quantitativamente o nível de qualidade, pelo menos no que diz respeito ao atendimento de requisitos de desempenho, do componente.

Imagine-se agora o desenvolvimento de componentes para criptografia de dados segundo um algoritmo conhecido, O DES (*Data Encryption Standard*), por exemplo. Diferentes componentes, baseados em diferentes linguagens de programação, para executarem em diferentes plataformas de hardware, podem compartilhar uma mesma especificação DES de criptografia. O DES trabalha criptografando mensagens de 64 bits, utilizando chaves de 64 bits (ignorando-se os últimos bits de cada bloco de 8 bits), e obtendo como resultados cifras.

Texto Simples	Cifra
80 00 00 00 00 00 00 00	9A 90 BC 0B 75 C7 37 03
40 00 00 00 00 00 00 00	CC 68 43 59 8C 73 2B BE
20 00 00 00 00 00 00 00	13 72 95 35 09 B3 C1 4C
10 00 00 00 00 00 00 00	70 AA AA 84 18 E4 89 30
08 00 00 00 00 00 00 00	E4 B0 B4 A1 39 E8 54 6E
04 00 00 00 00 00 00 00	70 18 F7 13 66 14 6E AF
02 00 00 00 00 00 00 00	B3 8F 3D 7E 4F 2D 25 3D

Tabela 6. Testes de validação de conformidade com a especificação do algoritmo de criptografia DES (NIST, 2005).

A especificação funcional dos componentes deve conter, então, um conjunto de testes para verificação da conformidade da implementação com a especificação DES, através de uma lista contendo entradas, chaves e resultados esperados. A tabela 6 exemplifica um conjunto de testes unitários com o propósito de certificar (funcionalmente) componentes que implementem a especificação DES de maneira precisa.

4.2.2. Avaliação de arquiteturas

Segundo Bass, Clements e Kazman, "a arquitetura de software de um programa ou sistema computacional é a estrutura ou estruturas do sistema, que abrangem componentes de software, as propriedades externamente visíveis destes componentes, e os relacionamentos entre eles" (BASS, 1999).

O processo de avaliação de arquiteturas é realizado através do emprego da metodologia ATAM (*Architecture Tradeoff Analysis Method*) do SEI (CLEMENTS; KAZMAN; KLEIN, 2002). Sua aplicação no *Komponento* se dá tanto no nível de arquiteturas mais complexas (componentes podem ser compostos por outros componentes (KRUCHTEN, 2000) (ATKINSON, 2000), e assim recursivamente), quanto no nível de micro arquiteturas de componentes constituídas por um menor número de objetos e relacionamentos.

O processo de avaliação dos requisitos de qualidade de software, baseado no método ATAM, consiste em nove passos, divididos em quatro grupos. Os passos estão numerados de 0 a 9:

- *Apresentação.* Envolve a troca de informação com os clientes, onde são apresentadas (1) a metodologia (ATAM), (2) as metas de negócio, e (3) a arquitetura do produto ou componente proposto. Esta fase pode se tornar demasiadamente burocrática, em certas ocasiões, como para o caso do desenvolvimento de componentes para uso interno, podendo ser descartada nestas situações.

- *Investigação e análise.* Este grupo de tarefas avalia os atributos chaves de qualidade contra as abordagens de arquitetura da solução. Consiste em (4) identificar as abordagens de arquitetura, (5) gerar a árvore de atributos de qualidade, e (6) analisar as abordagens de arquitetura.

- *Teste.* Faz a verificação dos resultados de acordo com as necessidades de todas as partes envolvidas (arquitetos de software, desenvolvedores, especialistas em segurança, gerentes de projeto, usuários etc.), através dos passos de (7) priorização de cenários e (8) análise das abordagens de arquitetura (reiteração do passo 6).

- *Relatório.* Os resultados da aplicação do ATAM, com base nos dados colhidos durante as fases anteriores, são apresentados às partes interessadas.

A árvore de atributos de qualidade do passo (5) é um modelo no formato de uma árvore lógica, com raiz, nós e folhas. Sua estrutura é muito semelhante ao modelo Goal/Question/Metric (GQM). Basili coloca que o GQM parte da suposição de que para uma organização medir objetivamente a qualidade de seu software, deve primeiro especificar suas próprias metas e a de seus projetos (GOAL), alinhá-las aos dados que podem de alguma maneira as definir operacionalmente (QUESTION), e prover uma estrutura para interpretar os dados em relação às metas estabelecidas (METRIC) (BASILI, 1994). A figura 4.15 mostra um cenário com duas metas (desempenho e escalabilidade), para um servidor transacional. O modelo GQM é uma abordagem bastante prática e objetiva para se avaliar qualidade, estipulando parâmetros não-funcionais bem definidos que devem ser atendidos por um projeto de software. Dagnino apresenta um modelo, baseado no GQM, através do qual

podem-se determinar cenários de custo (em dinheiro) e esforço (homem/hora), num projeto baseado em componentes, e benefícios esperados (DAGNINO, 2003).

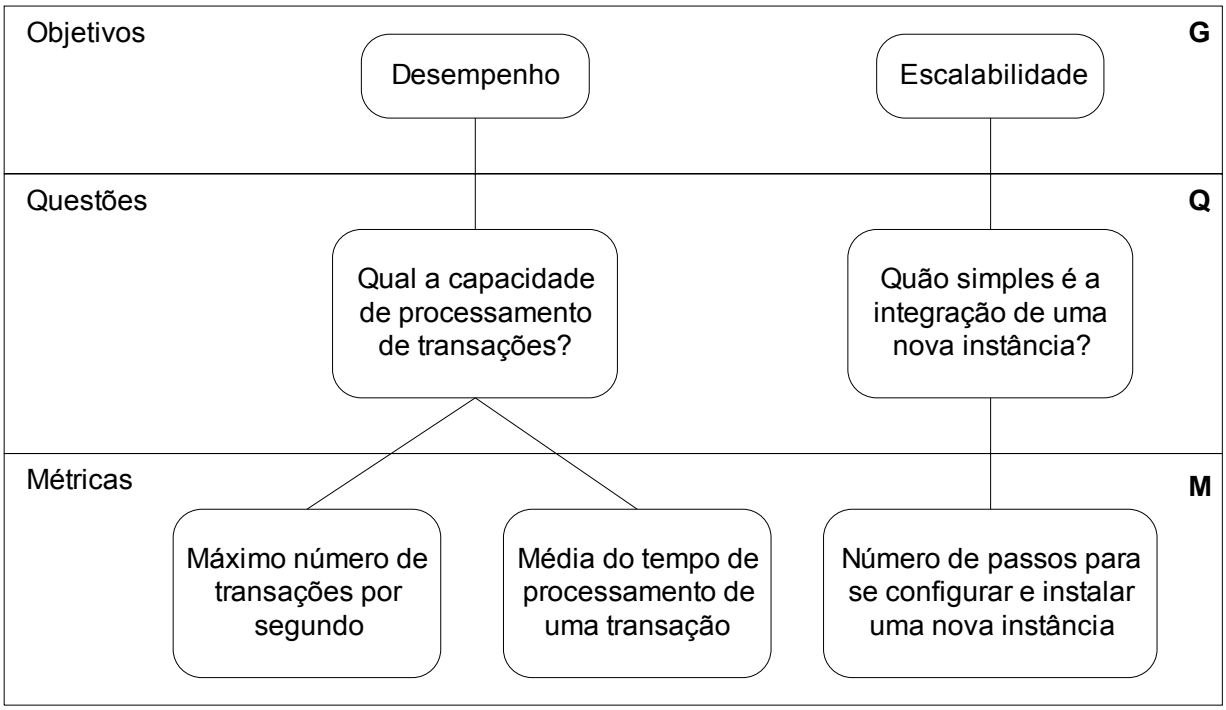


Figura 4.15. Um exemplo da estrutura do GQM para o atendimento dos requisitos de desempenho e escalabilidade de um servidor transacional.

A árvore de qualidade do ATAM é um artefato também baseado em níveis de detalhamento. A árvore de qualidade do *Komponento* é uma especialização da árvore do ATAM, no sentido de que os requisitos avaliados são os estabelecidos pela norma ISO/IEC 9126 (ISO/IEC, 1991), requisitos estes já detalhados na seção 2.4.2.

A figura 4.16 ilustra como seria uma árvore de qualidade do *Komponento*, para a avaliação de um servidor de transações em termos de portabilidade e confiabilidade. O primeiro nível (camada mais à esquerda) seriam os requisitos sugeridos pela ISO/IEC 9126, as metas genéricas de qualidade; o segundo nível (camada intermediária) seriam os cenários relevantes à avaliação daquela meta; e o último nível (camada mais a direita) seriam as implementações

das abordagens de arquitetura para os cenários considerados relevantes para o requisito em questão. Esse artefato (a árvore de qualidade), no *Komponento*, é implementado na forma de um documento de especificação de requisitos não funcionais a serem atendidos por um componente, num formato padronizado, refletindo a classificação ISO/IEC 9126 de atributos de qualidade de software.

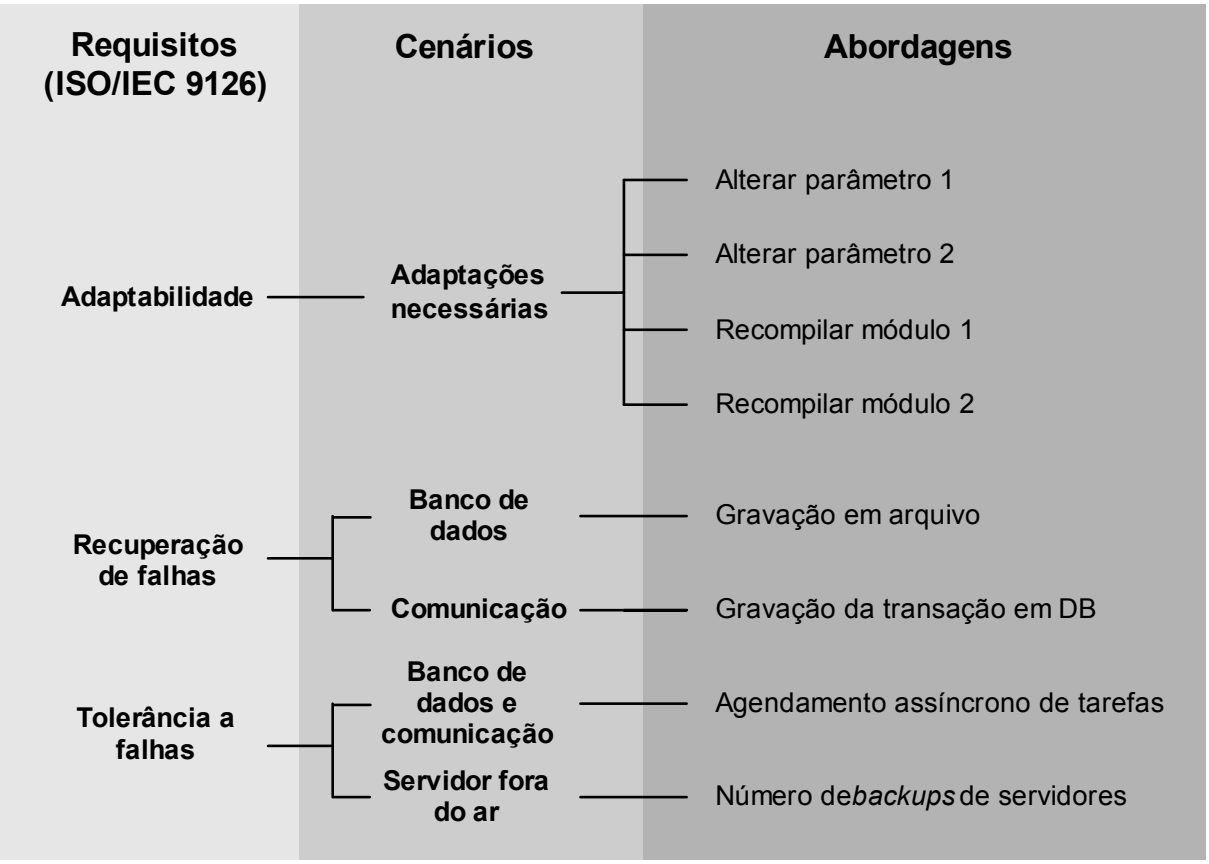


Figura 4.16. Árvore de qualidade do Komponento, segundo os requisitos de qualidade da norma ISO/IEC 9126.

A metodologia ATAM adaptada para o *Komponento* pode ser aplicada tanto para componentes sendo desenvolvidos internamente ou extraídos de um legado, quanto para a avaliação de COTS ou componentes encomendados. Para se exemplificar, então, a aplicação do método, imagine-se um sistema de acompanhamento de ordens de serviços, sendo

desenvolvido para clientes de uma determinada empresa de desenvolvimento de software. As etapas hipotéticas:

- *Apresentação.* Reuniões entre clientes, gerentes de projeto e desenvolvedores dão início ao processo. Considerando-se a principal meta de desenvolvimento a automatização de rotinas de acompanhamento de ordens de serviços, refletindo contratos entre a empresa prestadora destes serviços e seus clientes, as diretivas de negócio são, a princípio:

    1. Controlar o acesso aos contratos de produtos e serviços do cliente, de forma que as propriedades dos contratos só possam ser alteradas, de acordo com o estado da ordem de serviço, por pessoas autorizadas;

    2. Proporcionar certo nível de integridade dos dados nos contratos solicitados;

    3. Entregar um sistema que possa ser acessado por um grande número de usuários, distribuídos geograficamente, a partir de diferentes computadores, com mínimos esforços de instalação.

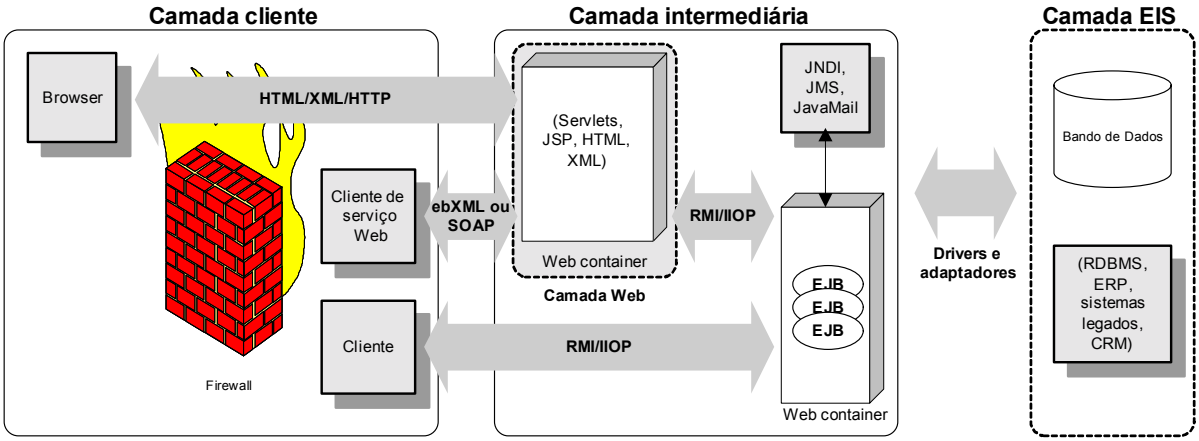


Figura 4.17. As camadas de software da arquitetura da solução para o sistema de acompanhamento de ordens de serviço (CAVANESS, 2003).

A arquitetura apresentada para o sistema, então, é a da figura 4.17. Uma solução baseada em três camadas: a camada cliente constituída por navegadores, clientes SOAP, e de acesso direto aos componentes EJB, de maneira distribuída; a camada intermediária composta por servidores *web* e containers EJB; e a camada EIS (*Enterprise Information Service*) formada por sistemas legados, ERPs (*Enterprise Resource Planning*), bancos de dados relacionais etc. Uma arquitetura em camadas traz, entre outros benefícios, separação de responsabilidades, reusabilidade e escalabilidade (CAVANESS, 2003).

- *Investigação e análise.* As diretivas de negócio aqui são mapeadas em abordagens de arquitetura. Para os requisitos ISO/IEC 9126 de qualidade de software, são levantadas as abordagens de arquitetura, para diferentes cenários, alinhados às diretivas de negócio. A seguinte árvore de atributos de qualidade (figura 4.18) pode ser montada.

A metodologia ATAM sugere que os cenários sejam priorizados conforme duas dimensões: a importância para o sucesso da implementação do sistema, e a

complexidade de implementação da abordagem proposta. Os níveis de prioridade são *Low* (L), *Medium* (M) e *High* (H), aplicados às duas dimensões.

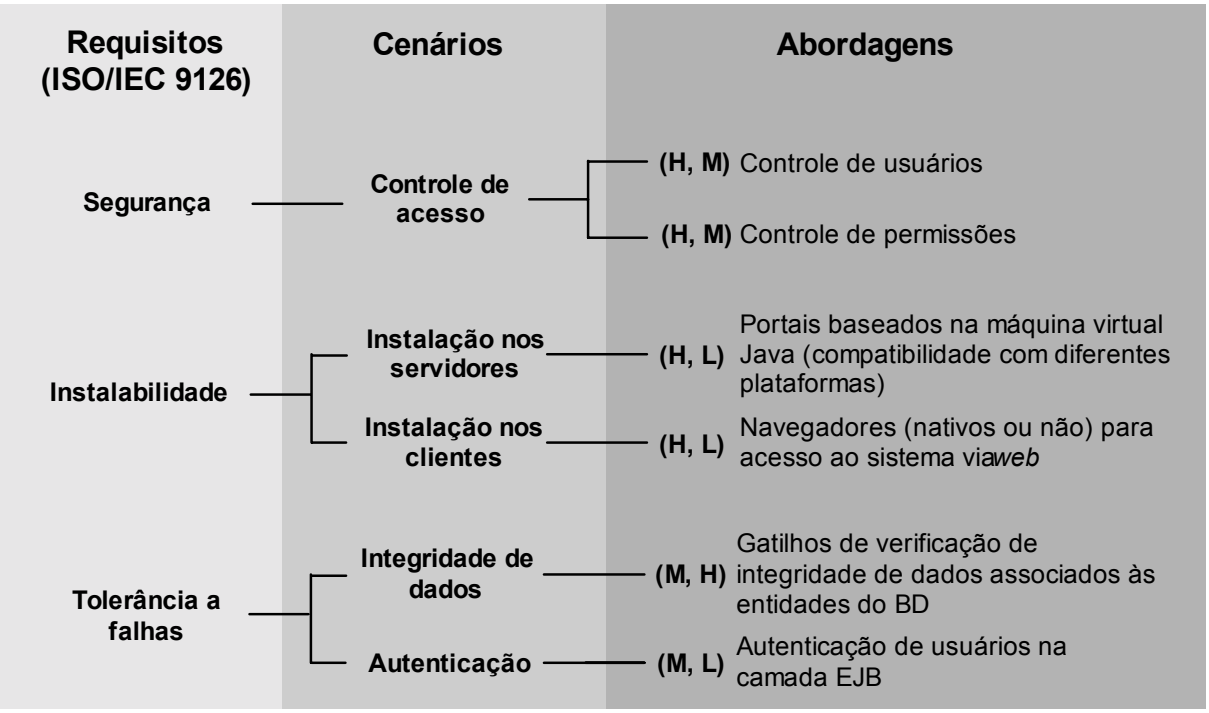


Figura 4.18. Árvore de qualidade com os cenários alinhados às diretivas de negócio apresentadas na fase anterior.

Desta maneira, tem-se, para o cenário de autenticação de usuários na camada EJB, por exemplo, uma prioridade média (M, L) de importância para o sistema e uma complexidade baixa de implementação (M, L).

- *Teste.* Esta fase é um refinamento da fase de investigação e análise, levantando novos cenários de acordo com novas necessidades, e priorizando todo o conjunto de abordagens de implementação relacionadas até então. Dois novos atributos são agregados aos requisitos de arquitetura do sistema, nesta fase, bem como prioridades são atribuídas às novas abordagens, conforme a figura 4.19.

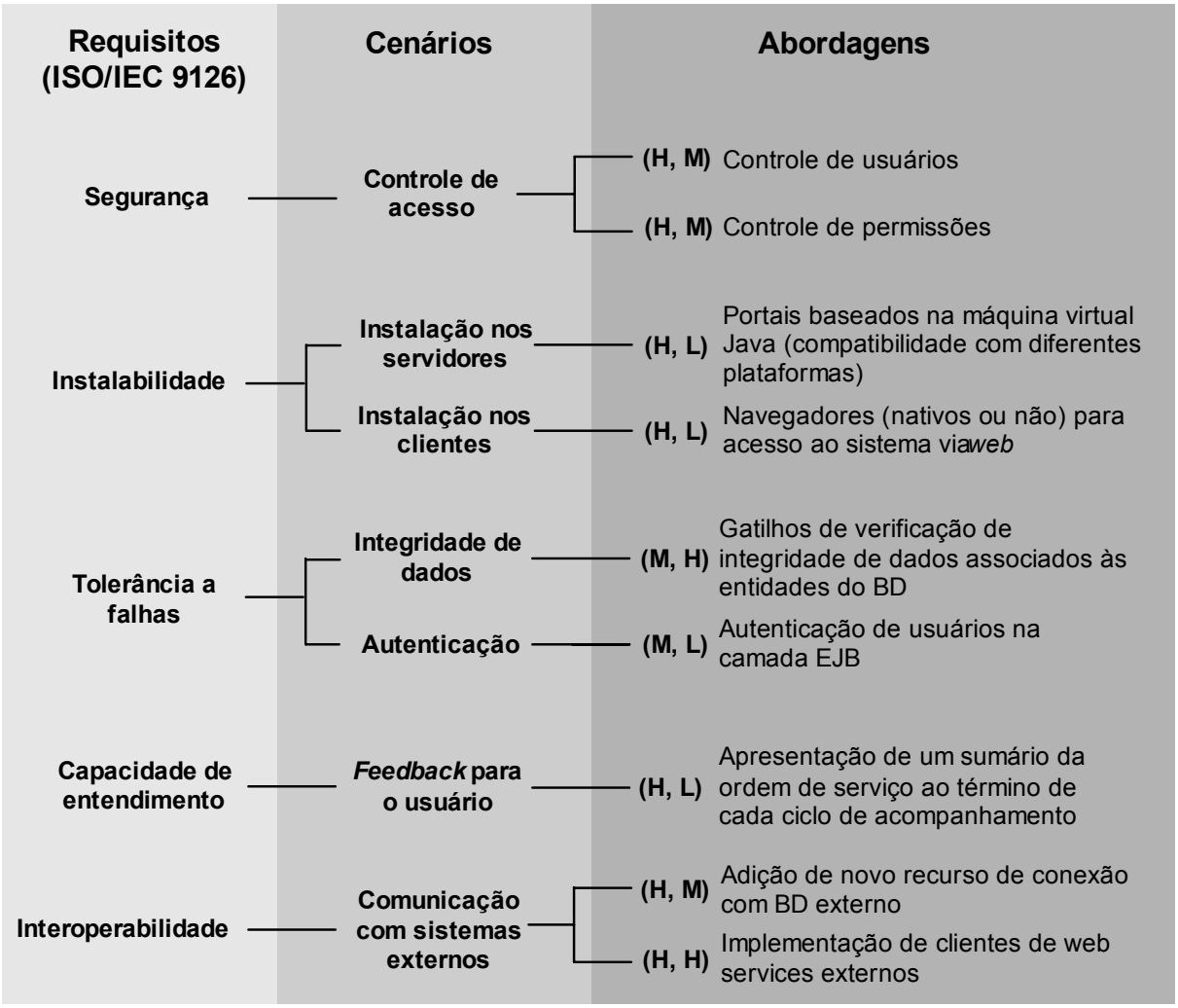


Figura 4.19. Novos cenários adicionados à árvore de atributos de qualidade produzida na fase anterior de investigação e análise.

- *Relatório.* Os resultados são apresentados na forma de um relatório, segundo um documento refletindo os requisitos ISO/IEC 9126 de qualidade, os cenários em que estes requisitos são necessários, e as abordagens de arquitetura utilizadas. Abaixo, um pequeno modelo de documento descrevendo as estratégias de implementação empregadas para o sistema de acompanhamento de ordens de serviço.

Sistema de Acompanhamento de Ordens de Serviço

Relatório de Requisitos Não-Funcionais

1. Funcionalidade

1.1. Segurança

Cenário	Abordagem
Controle de acesso	Controle de usuários
	Controle de permissões

1.2. Interoperabilidade

Cenário	Abordagem
Comunicação com sistemas externos	Adição de novo recurso de conexão com BD externo
	Implementação de clientes de *web services* externos

2. Portabilidade

2.1. Instalabilidade

Cenário	Abordagem
Instalação nos servidores	Portais baseados na máquina

	virtual Java (compatibilidade com diferentes plataformas)
Instalação nos clientes	Navegadores (nativos ou não) para acesso ao sistema via *web*

3. Confiabilidade

3.1. Tolerância a falhas

Cenário	**Abordagem**
Integridade de dados	Gatilhos de verificação de integridade de dados associados às entidades do BD
Autenticação	Autenticação de usuários na camada EJB

4. Usabilidade

4.1. Capacidade de entendimento

Cenário	Abordagem
Feedback para o usuário	Apresentação de um sumário da ordem de serviço ao término de cada ciclo de acompanhamento

4.3. Aplicação do modelo

Considerem-se as etapas do processo de desenvolvimento de software orientado a reuso, proposto por Sommerville (SOMMERVILLE, 2003): *Epecificação de Requisitos → Análise de Componentes → Modificação de Requisitos → Projeto de Sistema com Reuso → Implementação e Integração → Validação de Sistema*. O fluxo de atividades é retratado na figura 4.20. O processo de Sommerville é orientado ao reuso, com as atividades específicas de DBC destacadas na figura 4.20:

- *Análise de Componentes.* Considerando a especificação de requisitos, é feita uma análise de componentes que irão não apenas implementá-las, mas que serão projetados para implementar outras soluções.

- *Modificação de Requisitos.* Durante esse estágio, utilizando-se as informações sobre os componentes especificados, são analisados os requisitos, que são então modificados para o sistema poder comportar os componentes disponíveis.

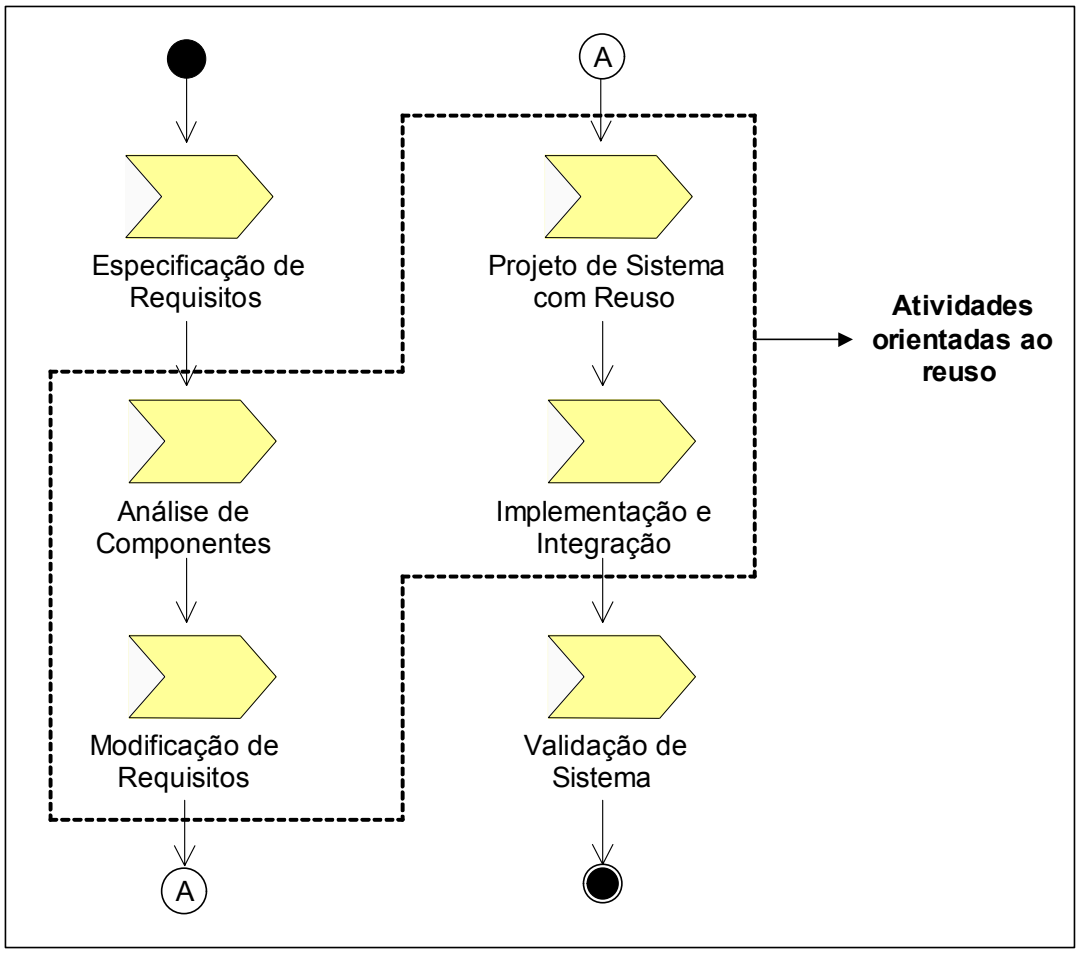


Figura 4.20. Processo proposto por Sommerville orientado ao reuso de componentes (SOMMERVILLE, 2003).

- *Projeto de Sistema com Reuso.* Durante essa fase, a infra-estrutura do sistema é projetada ou uma infra-estrutura existente é reutilizada. Os projetistas levam em consideração os componentes que são reutilizados e organizam a infra-estrutura para lidar com esse aspecto.

- *Desenvolvimento e Integração.* O software que não puder ser comprado será desenvolvido, e os componentes e sistemas COTS serão integrados, a fim de criar um sistema integral.

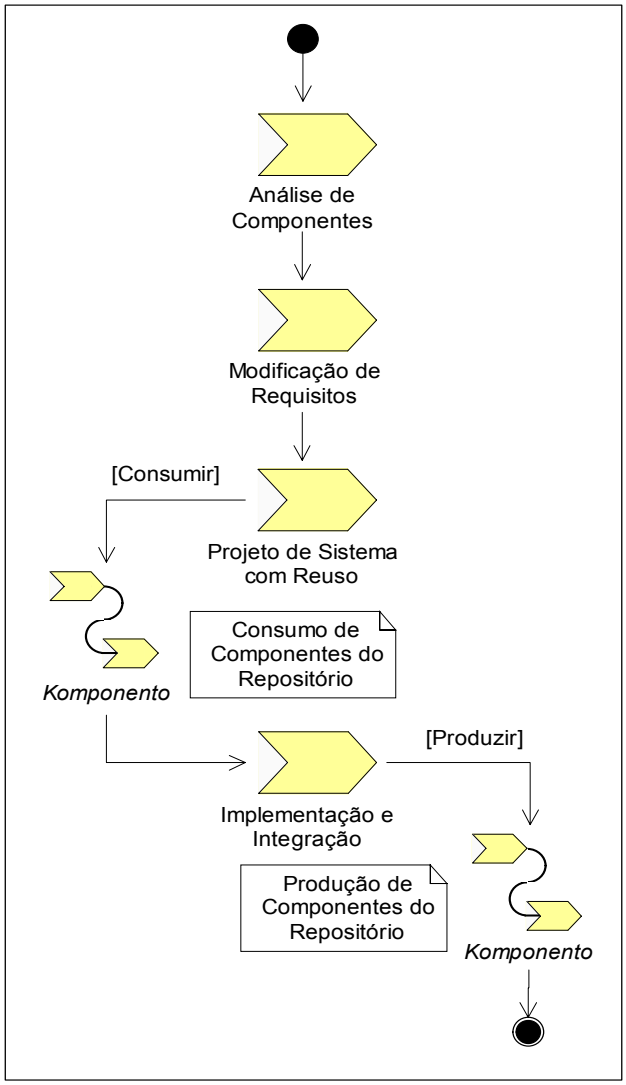


Figura 4.21. Komponento dando apoio ao processo de desenvolvimento baseado em componentes de Sommerville.

Apesar do processo de Sommerville prever a reutilização de componentes, não cobre justamente os pontos que o modelo de processo *Komponento* se propõe a atender: a manutenção de um repositório de componentes em constante evolução, que alimente os processos de desenvolvimento de software com componentes prontos e testados, e que de maneira inversa seja populado por novos componentes produzidos por estes processos.

Desta forma, o *Komponento* daria suporte ao processo de Sommerville de acordo com o

esquema da figura 4.21. A atividade de Consumo de componentes alimentaria a atividade de Implementação e Integração, que por sua vez, produziria novos componentes a serem integrados no repositório (Produção).

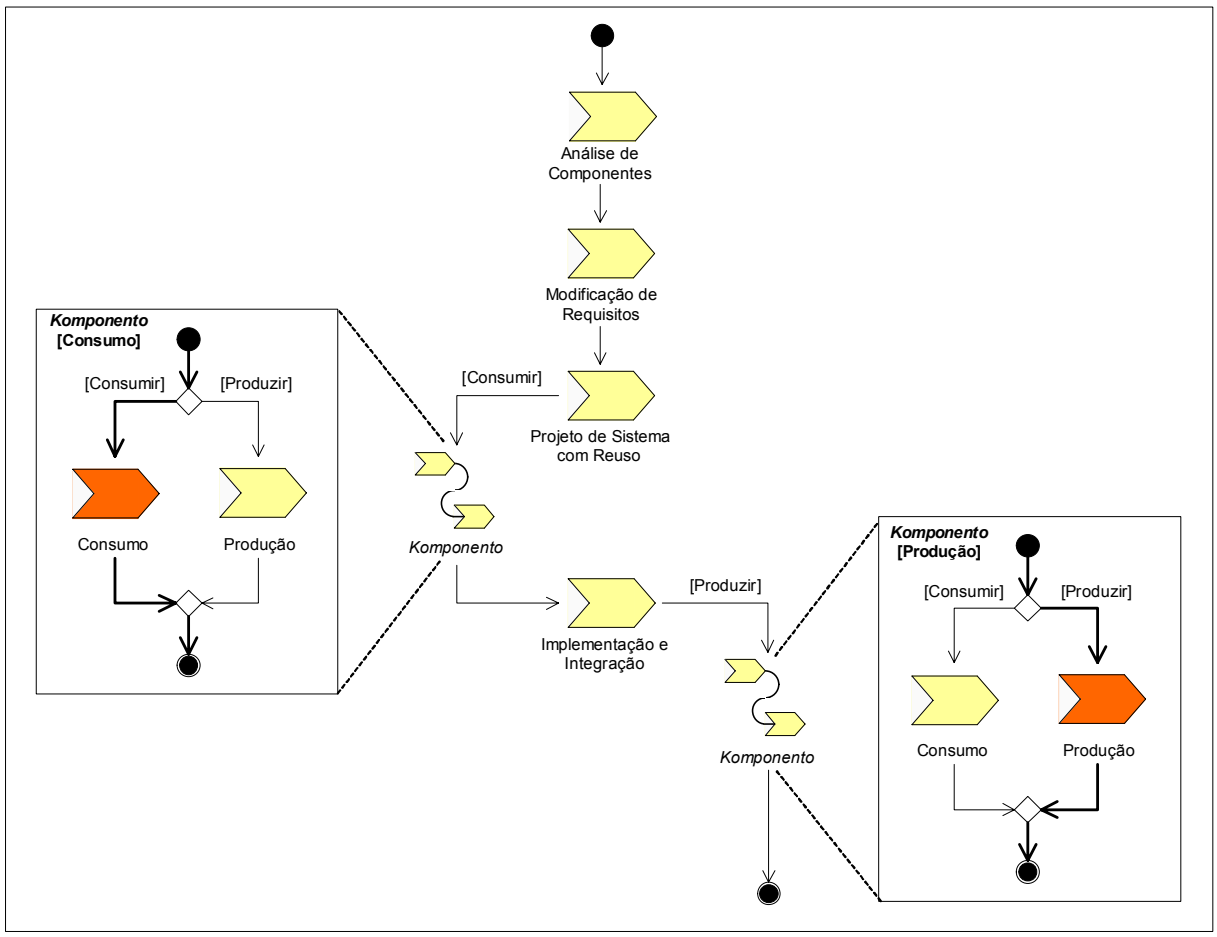


Figura 4.22. O bloco de atividades expandido do Komponento, dando suporte ao processo de Sommerville.

Detalhando-se a figura 4.21[16], os blocos de atividades do *Komponento*, podem ser expandidos no nível do diagrama de atividades apresentado na seção 4.1.1, ilustrado na figura 4.3. A necessidade de consumo de componentes de um repositório dispara a atividade

[16] De acordo com a especificação UML 2.0 (FOWLER, 2004), atividades podem ser decompostas em subatividades, através de níveis de detalhamento.

de Consumo, assim como a de produção dispara a atividade de Produção do *Komponento*.

Detalhando-se os processos, finalmente, nos níveis dos fluxos de tarefas descritos na seção 4.1.5, e atribuindo-se responsabilidades para cada uma das atividades ou grupos de atividades, representar-se-ia, por exemplo, a seqüência *Projeto de Sistema com Reuso* → *Consumo (de componentes)* → *Implementação e Integração* através do diagrama de atividades da figura 4.23.

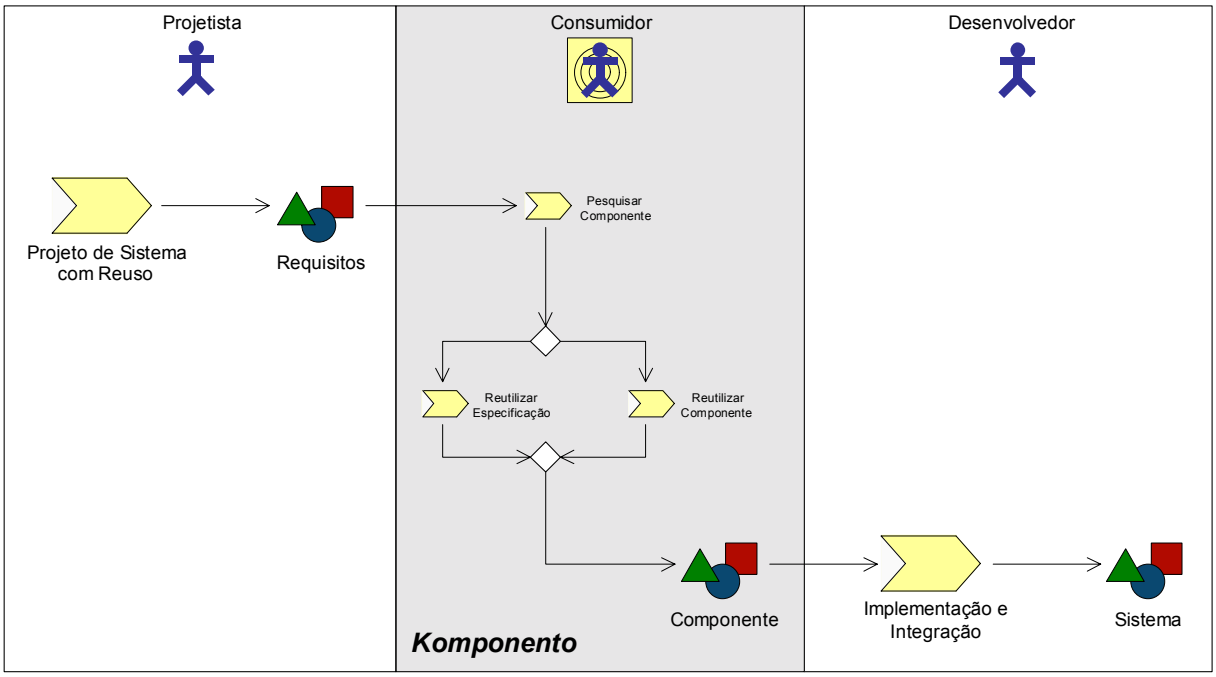


Figura 4.23. Diagrama de atividades da atividade de Consumo do Komponento dando suporte às atividades de Projeto e Implementação de Sommerville.

O Projeto de Sistema com Reuso entraria com requisitos para a atividade de Consumo do *Komponento*, expandida no fluxo de atividades da figura 4.12. Através de critérios de busca de acordo com as necessidades do projeto, realizar-se-ia a pesquisa de componentes que melhor atendesses a estas necessidades. Estes componentes e/ou suas especificações seriam reutilizadas, então, na implementação do novo sistema.

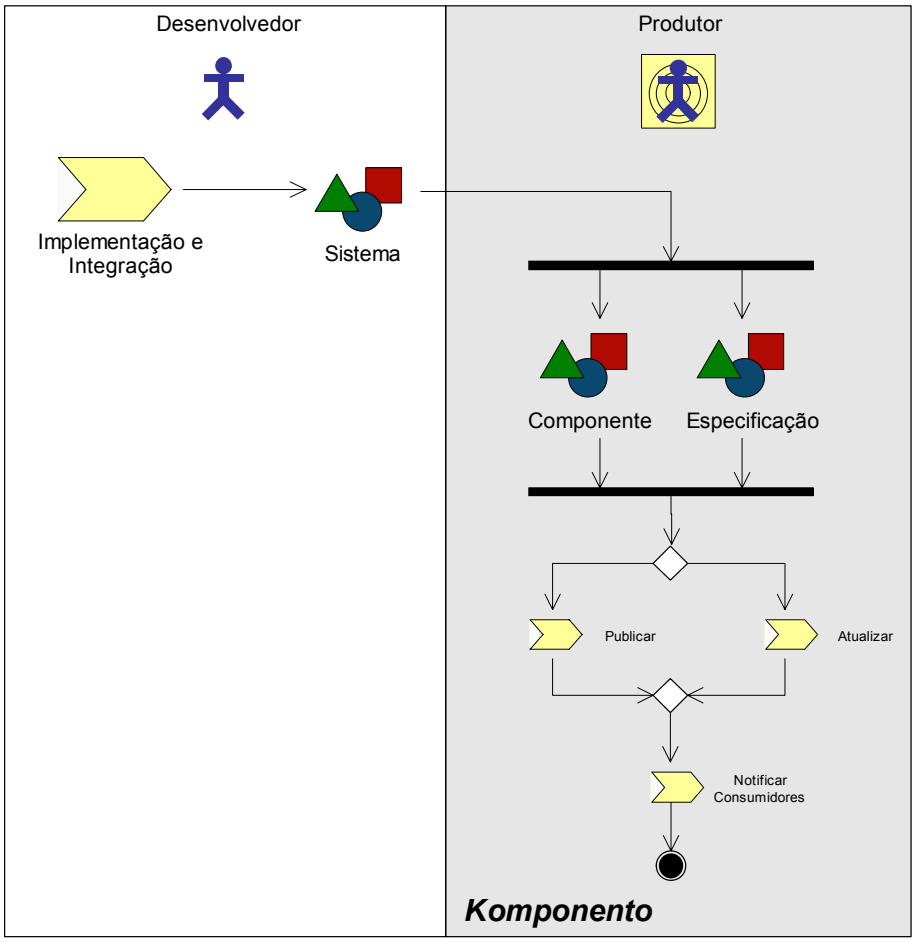


Figura 4.24. Diagrama de atividades da atividade de Produção do Komponento dando suporte à atividade de Implementação de Sommerville.

Analogamente, poder-se-ia representar, no mesmo nível da figura 4.24, a atividade de Produção do *Komponento* interagindo com as atividades de Implementação e Integração do processo de Sommerville. Novos componentes (e suas especificações) desenvolvidos nas etapas de implementação do sistema alimentariam o repositório, sendo aí publicados. Componentes (e suas especificações) reutilizados do repositório para a implementação do sistema, poderiam eventualmente ser atualizados da base de componentes ao fim do processo como um todo, ou de cada fase do mesmo. Assim, o *Komponento* garantiria meios de se organizar a evolução contínua de seu repositório.

Considere-se agora o ciclo de vida do processo de Sommerville, integrado às atividades de Consumo e Produção do *Komponento*, conforme o esquema da figura 4.25.

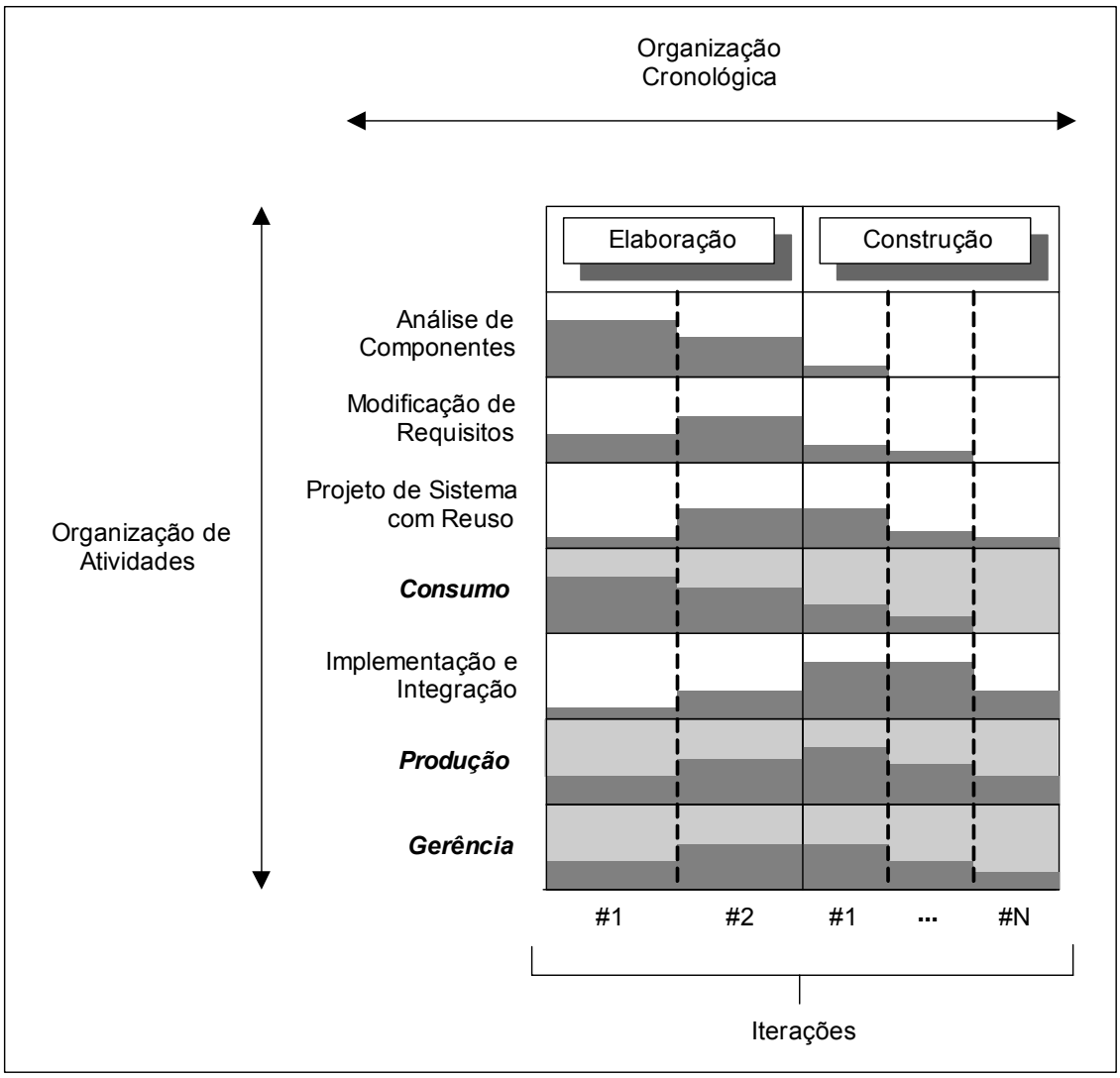


Figura 4.25. Ciclo de vida do processo de Sommerville com suporte do Komponento, baseado em fases e iterações hipotéticas.

Vemos o processo como um todo composto por duas fases principais (Elaboração e Construção), compostas por iterações. A participação de cada atividade varia de intensidade de acordo com interação e a fase em, que se encontra. Assim, as atividades de Análise de Componentes (Sommerville) e Consumo (*Komponento*) de componentes participam muito mais da fase de Elaboração do que de Construção; Implementação e Integração

(Sommerville) e publicação (atividade de Produção do *Komponento*) de itens no repositório, por outro lado, são muito mais intensas na fase de Construção.

Conforme já mencionado, então, na seção 4.1.6, percebe-se pela figura 4.25, que o cilco de vida do *Komponento* reflete o ciclo de vida do processo a que dá suporte. A atividade de Gerência do *Komponento* está também representada na figura 4.25, sendo mais intensa durante a Iteração #2 da fase de Elaboração e da Iteração #1 da fase de Construção.

5. O repositório

Este capítulo dedica-se à descrição do modelo de repositório de componentes utilizado no processo *Komponento*. Mostra a estrutura de armazenamento de componentes, compatível com o padrão RAS da OMG, bem como apresenta um protótipo de implementação de serviços oferecidos pelo repositório, especificados através de APIs (OMG, 2004), para consulta de componentes. O capítulo descreve também como as atividades do processo interagem com o repositório.

5.1. Modelo do repositório

Diversos estudos já foram realizados a cerca de modelos de bibliotecas de componentes reutilizáveis. Henninger enfatiza a importância da utilização de repositórios de componentes como meio de se atingirem programas eficazes de reuso (HENNINGER, 1997). A pesquisa de Guo (GUO, 2000) relaciona uma série de bibliotecas de componentes, apontando as principais características de cada uma delas.

Casanova, Straeten e Jonckers (CASANOVA, 2003), em vista das dificuldades intrínsecas de se reutilizar e de se manter um acervo de diferentes componentes e suas versões, e de se controlar a evolução de aplicações baseadas nestes componentes, propõem um modelo de repositório preocupado com a evolução dos próprios componentes, através da gerência de configuração, controle de versões, e métricas. Nesta mesma linha evolutiva, Henninger afirma que as estruturas requeridas para a utilização de componentes são, frequentemente, estáticas e incapazes de se adaptarem a contextos dinâmicos de desenvolvimento. Isto se torna um problema significativo, não só pelo fato da dificuldade de se projetar uma estrutura ideal numa primeira tentativa, mas também porque os domínios da computação evoluem

constantemente, por razões como mudanças tecnológicas, dinâmicas de projeto, e alterações de requisitos.

A solução encontrada pelo *Komponento*, para acomodar a evolução natural de um repositório de componentes, foi a de se basear num meta-modelo (o RAS da OMG) de armazenamento de componentes, mantendo-se uma estrutura de meta-classes e relacionamentos, flexível o suficiente para refletir mudanças de toda ordem.

Sobre o RAS, Allen afirma que duas condições necessárias, para que a indústria de software possa alcançar ganhos reais em qualidade e produtividade, são: (1) padrões para especificação de componentes devem ser fixados em todos os níveis, desde o técnico ao administrativo; (2) e um alto grau de qualidade deve ser alcançado a fim de se obter a confiança de clientes nos componentes especificados e implementados (ALLEN, 2002). No que diz respeito ao *Komponento*, o nível de qualidade dos produtos pode ser atingido tanto pelo processo de certificação, quanto pelo de avaliação de arquiteturas, já descritos nas seções 4.1.1 e 4.1.2; a utilização de um padrão de armazenamento de componentes, flexível para se adaptar a mudanças, pode ser conseguida pela adoção do RAS.

5.1.1. RAS

A especificação RAS (*Reusable Asset Specification*), proposta pela OMG, define uma forma padrão de empacotamento de ativos de software. Um ativo reutilizável de software pode ser considerado, de maneira genérica, uma coleção coesa de artefatos que solucionam um problema específico ou um conjunto de problemas, encontrados no decorrer do ciclo de desenvolvimento de software. Dentro do contexto do RAS, o pacote de um ativo reutilizável de software é um conjunto de arquivos, que implementam a solução, e uma estrutura de meta-informação que define e descreve o ativo como um todo. A especificação da OMG define três tipos de ativos reutilizáveis de software:

- *Componente.* Tipo de ativo que adere a uma interface bem documentada e que, tipicamente, oculta sua implementação (*black box*), tornando-a implícita.

- *Framework.* Empregado para solucionar um conjunto maior de problemas. Apresenta-se, normalmente, como uma coleção de ativos individuais ou *middlewares* sobre os quais aplicações são construídas. Vide definição de Gamma (GAMMA, 1995).

- *Padrão.* Uma abstração da estrutura e do comportamento de um sistema ou parte de um sistema. Um padrão pode ser aplicado a um sistema, no sentido de orientar a estrutura e a dinâmica dos elementos do mesmo. Vide definição de Gamma (GAMMA, 1995).

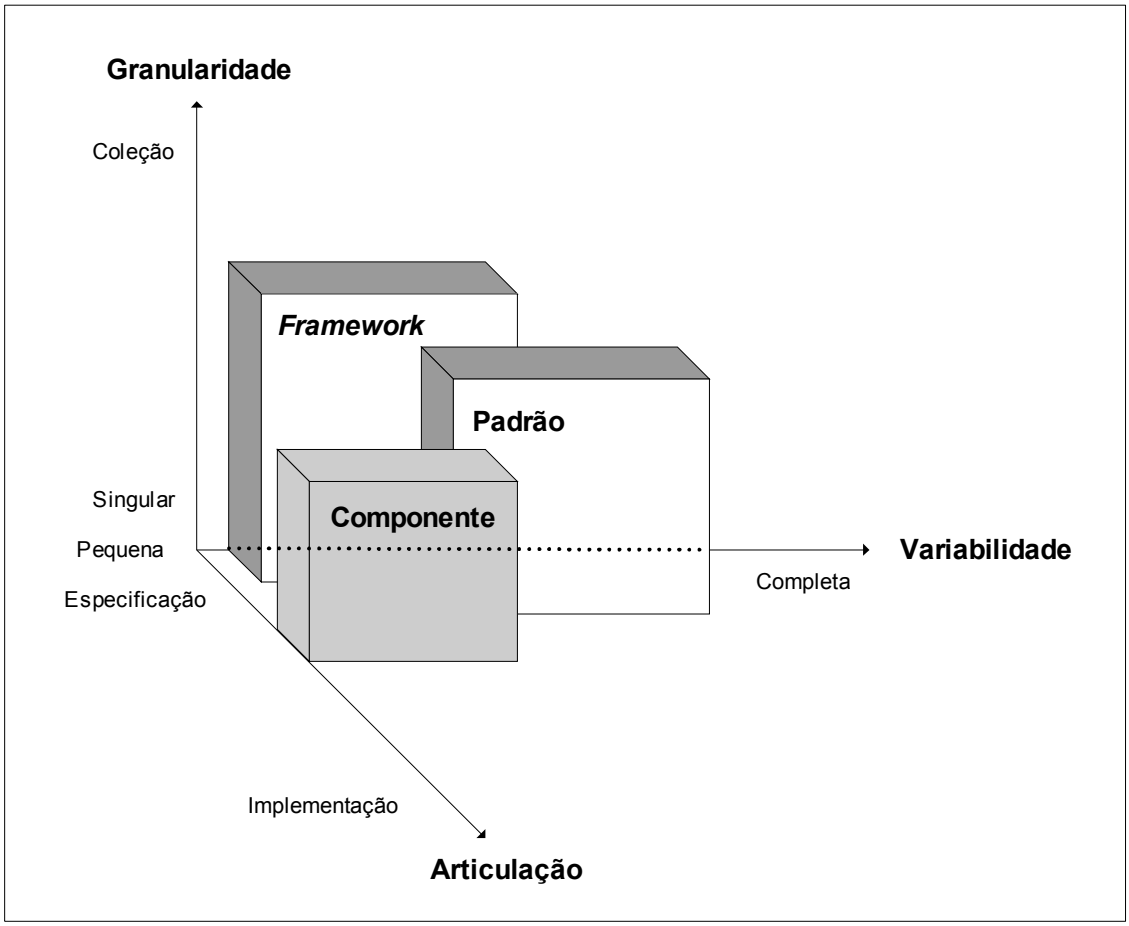


Figura 5.1. Tipos de ativos reutilizáveis de software (OMG, 2004).

Essa classificação não é discreta, mas sim contínua, visto a grande variedade de tipos distintos de componentes no mercado. O termo componente, mencionado neste trabalho, seria mais bem categorizado no grupo de ativos do tipo componente, conforme ilustrado na figura abaixo.

5.1.1.1. Meta-modelo

O meta-modelo RAS é definido em duas grandes categorias, o *Core* (núcleo) RAS e os *Profiles* (perfis). O *Core* RAS representa os elementos fundamentais da especificação de um ativo. Os *Profiles* descrevem extensões destes elementos fundamentais. Um perfil não deve

alterar a definição ou a semântica dos elementos do núcleo. O *Core* RAS não é instanciável, portanto, um ativo deve ser de um determinado *Profile*. O núcleo do meta-modelo RAS e seus perfis padrões, definidos pela especificação, estão ilustrados no esquema abaixo.

Default Profile (perfil padrão) deriva diretamente de *Core RAS*, que é classe abstrata. *Default Component Profile* (perfil padrão de componente) e *Default Web Service Profile* (perfil padrão de serviço *web*) derivam, por sua vez, de *Default Profile*. Um perfil pode estender *Core RAS* (como assim o faz *Default Profile*) ou pode estender outro perfil (como é o caso de *Default Component* e *Default Web Service Profile*). *Costumized Profile*, que aparece no final do pacote de perfis à esquerda da figura 5.2, representa um quarto perfil hipotético, que pode estender *Core RAS* ou *Default Profile*, assim como especializar *Default Component Profile* ou *Default Web Service Profile*.

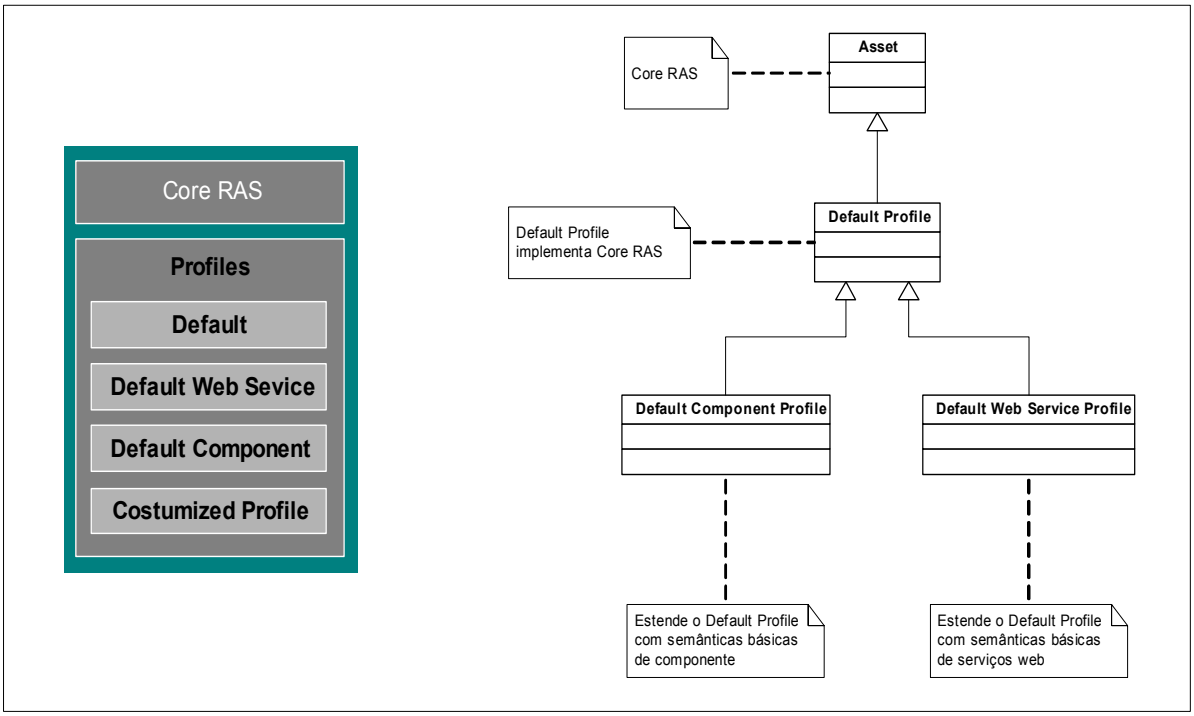


Figura 5.2. O pacote RAS constituído pelo núcleo do meta-modelo e pelos perfis derivados do núcleo (OMG, 2004).

Core RAS define, então, a estrutura básica que todo ativo de software deve conter. A figura 5.3 ilustra sem maiores detalhes esta estrutura básica, composta pelos seguintes elementos:

- *Classification* (classificação). Lista um conjunto de descritores para classificação de ativos, bem como uma descrição do contexto para o qual o ativo é relevante.
- *Solution* (solução). Descreve os artefatos que fazem parte do ativo.
- *Usage* (utilização). Contém as regras para instalação, personalização, e utilização do ativo.
- *Related Assets* (ativos relacionados). Descreve a relação deste ativo com outros.
- *Profile* (perfil). Representa o perfil do ativo, podendo ser um dos perfis padrões do RAS, uma extensão dos mesmos ou do próprio *Core RAS*.

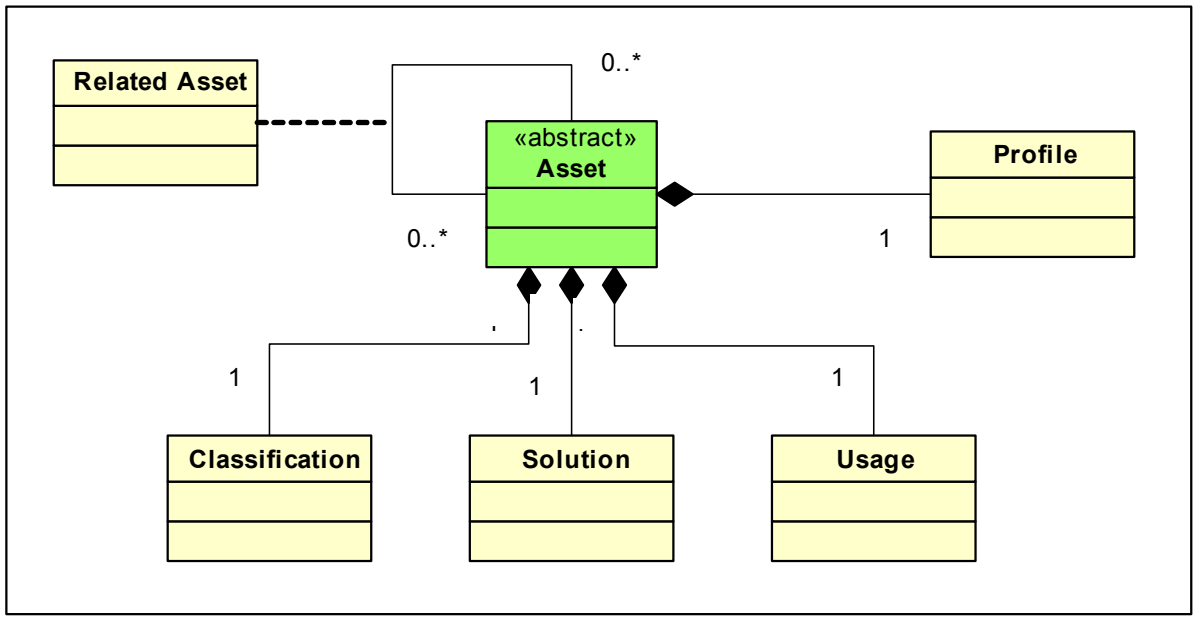


Figura 5.3. A estrutura geral do Core RAS (OMG, 2004).

Default Profile, na realidade, não adiciona novos elementos aos elementos de *Core RAS*, sendo aquele apenas uma implementação do modelo abstrato definido por este. Novos elementos passam, realmente, a ser agregados ao núcleo do meta-modelo do RAS, apenas pelas especializações de *Default Profile*. A figura 5.4 ilustra a estrutura de *Default Component Profile*, que adiciona à solução do ativo os elementos *Requirements* (requisitos), *Design* (projeto), *Implementation* (implementação), e *Test* (teste) – as quatro categorias de artefatos a comporem um componente de software (já mencionadas na seção 4.1.3). Artefatos podem também fazer parte da solução sem, no entanto, pertencerem a nenhum destes grupos.

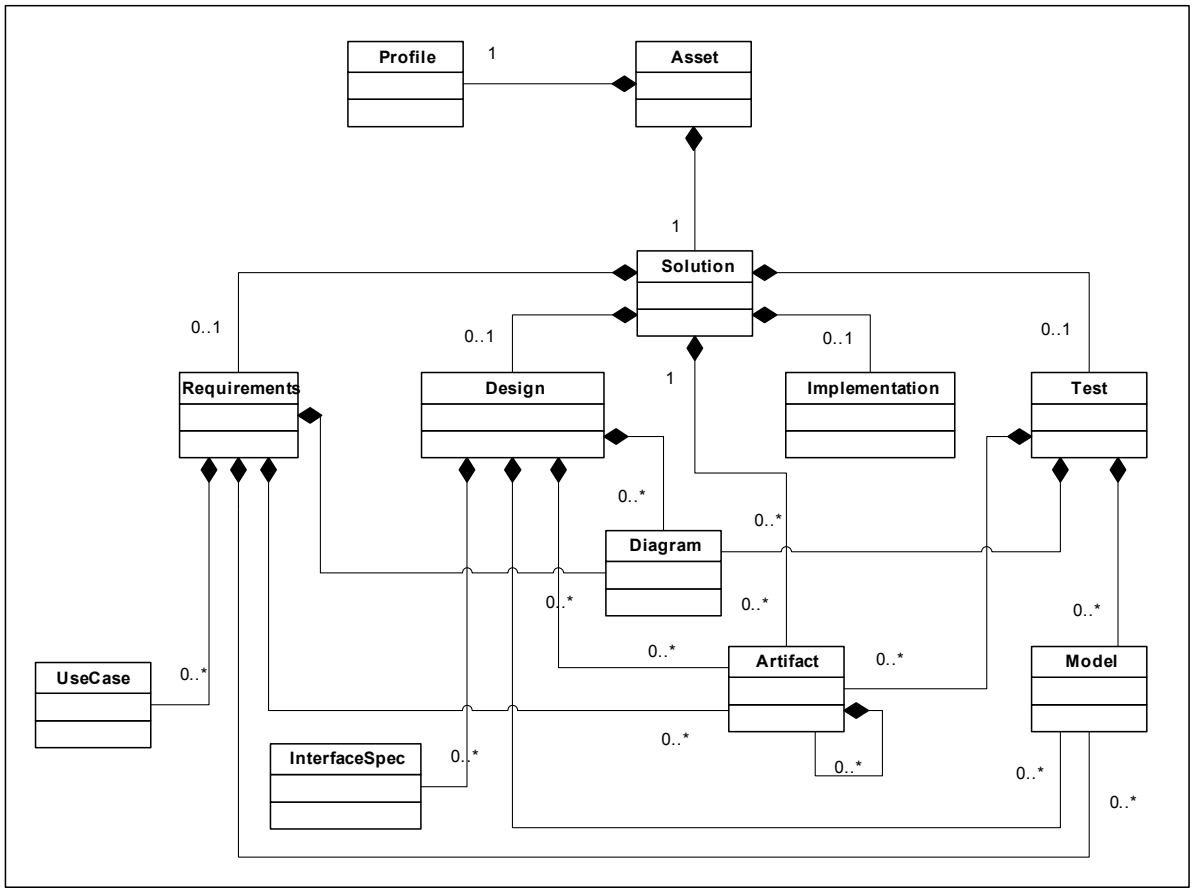


Figura 5.4. Estrutura de Default Component Profile (OMG, 2004).

5.1.1.2. Formato de armazenamento

A especificação RAS define um formato de armazenamento baseado em meta-informações, a respeito de um determinado ativo, num arquivo manifesto (*rasset.xml*). Este arquivo contém toda a relação de artefatos que compõem o ativo, bem como a localização destes artefatos. O *rasset.xml* nada mais é do que uma instância XML, definida segundo um esquema (.xsd, por exemplo) refletindo algum modelo de perfil RAS, como os que foram apresentados na seção anterior. A figura 5.5 mostra o encadeamento de conceitos: o meta-modelo definido por um perfil RAS é mapeado em um esquema XML, que é instanciado em um *rasset.xml*. É importante se notar a presença obrigatório de um arquivo XSD, que contenha um esquema de perfil (ex. *defaultProfile.xsd*, *defaultComponentProfile.xsd*, *DefaultWebServiceProfile.xsd* etc.), definindo a estrutura do arquivo manifesto de acordo com o perfil em questão. A seção 5.2.2.1 dá uma visão mais clara do formato de um arquivo XSD representando um perfil RAS.

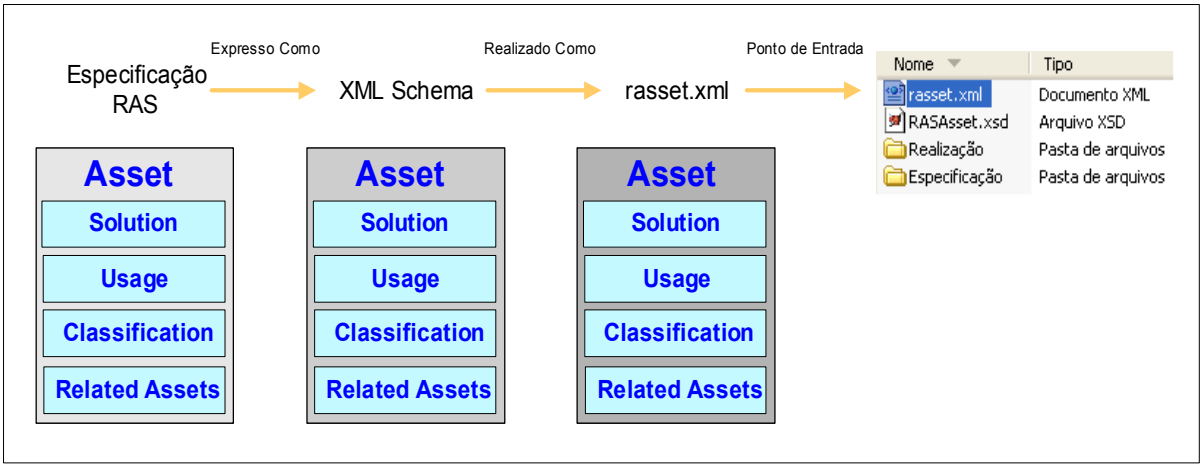


Figura 5.5. Os meta-modelos de perfis, expressos em UML, são concretizados em arquivos XML (OMG, 2004).

Um pacote de ativo é, então, uma coleção de artefatos (arquivos) mais um manifesto (*rasset.xml*). Existem, basicamente, duas formas de se armazenar um ativo, segundo a estrutura de armazenamento do RAS:

- *Empacotado como um único arquivo.* Aqui o ativo é armazenado como um pacote compactado em um único arquivo (.ras), contendo o manifesto, o arquivo XSD definindo o perfil a ser utilizado, e todos os artefatos que compõem o ativo. O caminho de cada artefato referenciado em *rasset.xml* é relativo à raiz do documento.

- *Desempacotado com os artefatos em localizações separadas.* Desenvolver componentes em equipe requer certo grau de paralelismo. Esta abordagem de armazenamento coloca o arquivo manifesto sob controle de versão, enquanto mantém os artefatos em suas localizações originais, dando maior flexibilidade na composição de componentes através apenas da meta-informação.

5.2. Implementação do repositório

Esta seção apresenta a implementação de um pequeno protótipo de repositório de componentes, baseado na ferramenta *Subversion* de controle de versões. Relembrando-se, então, os requisitos de um repositório de componentes, apresentados na seção 3.2.3, adotaram-se as seguintes abordagens de implementação para o *Komponento*:

1. *Sistema automatizado de biblioteca.* O sistema de controle de versões *Subversion* foi utilizado para controlar operações de leitura e escrita de artefatos armazenados em um

repositório. Morisio, Tully e Ezran (MORISIO, 2000) relacionam quatro estudos de caso de processos de suporte ao reuso de componentes. Todos os quatro utilizaram algum tipo de ferramenta, genérica ou não, de gerência de configuração. O *Subversion*, entre outras funções, implementa (PILATO, 2004):

- *Controle de versão de diretórios.* Mudanças na estrutura de diretórios são armazenadas em históricos. Para o *Komponento*, a estrutura de um componente pode ser representada por uma estrutura de diretórios, contendo scripts, código fonte, bibliotecas, arquivos de configuração etc. Desta forma a evolução de um componente pode ser gravada, não apenas em termos de seu conteúdo, mas também de sua estrutura (de diretórios).

- *Histórico flexível de controle de versão.* O Subversion provê mecanismos mais flexíveis do que os comumente implementados em outras ferramentas semelhantes. Operações como copiar um arquivo ou alterar seu nome são permitidas, sendo sempre registradas em histórico.

- *Alterações atômicas.* Todas as modificações são gravadas completamente no repositório, ou simplesmente não são gravadas (atomicidade), o que permite edições concorrentes de blocos dentro de um mesmo item sob controle de versão.

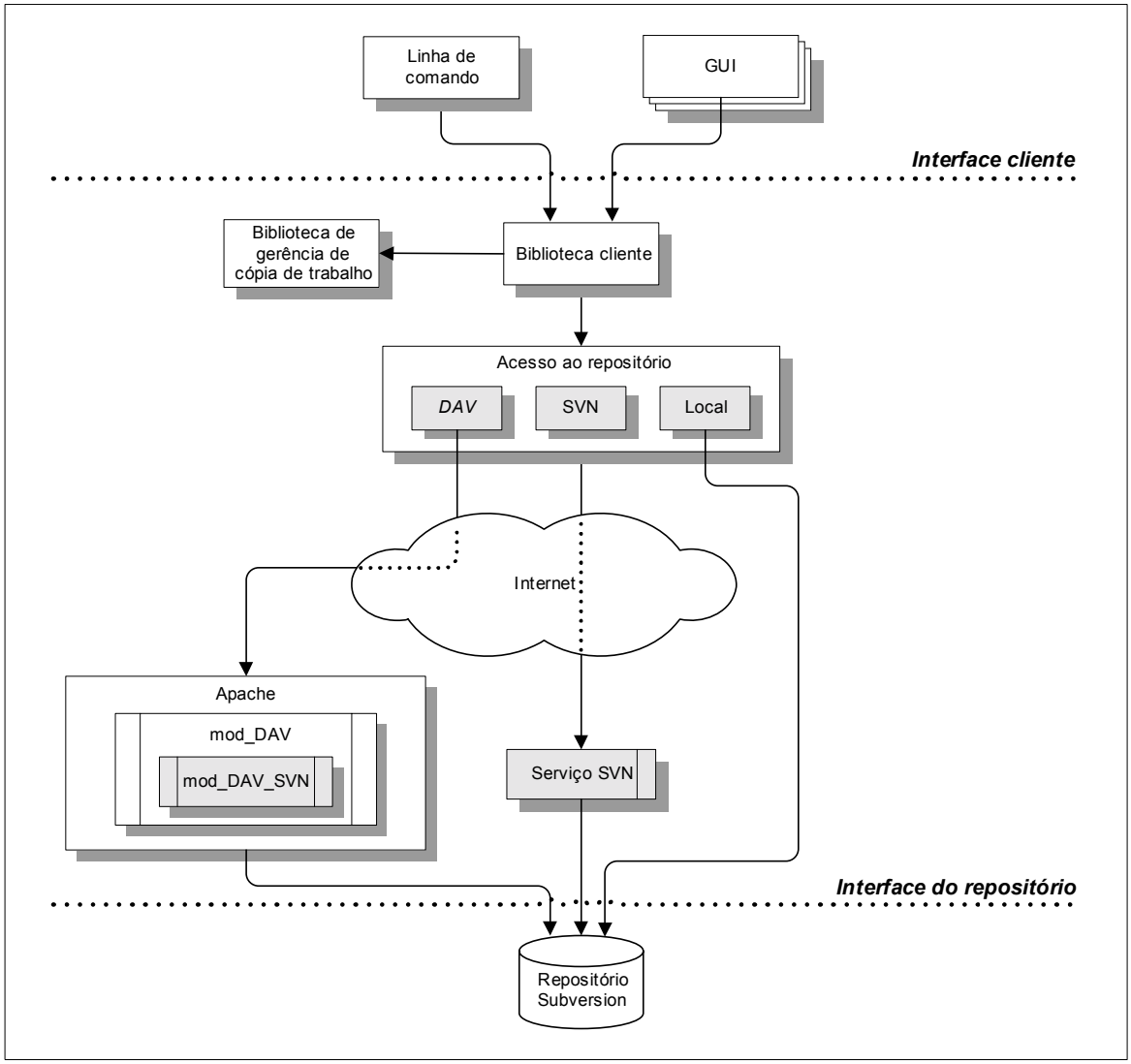


Figura 5.6. Arquitetura do Subversion (PILATO, 2004).

- *Controle de versão de meta-dados.* Cada arquivo ou diretório possui um conjunto de propriedades – pares de chave e valor – associado a ele. Pode-se criar e armazenar qualquer par arbitrário, como uma meta informação, em qualquer arquivo do repositório.

- *Interoperabilidade.* O *Subversion* possui um modelo abstrato de acesso a dados de um repositório, facilitando a integração com outras aplicações, com uma interface de acesso bem definida, comunicando-se através de protocolos

abertos, específicos, ou por chamada de função. A figura 5.6 ilustra a topologia de rede dos componentes do *Subversion*. Clientes baseados em interfaces gráficas com o usuário, bem como em linha de comando, comunicam-se com o repositório.

2. *Estrutura padrão de componente.* Seguindo-se a especificação RAS, foi possível se obter uma estrutura padrão de armazenamento e representação de componentes em um repositório.

3. *Esquema eficaz de classificação para cada domínio.* Apesar de um esquema de classificação de componentes não ter sido implementado no protótipo descrito nas seções seguintes, o modelo RAS prevê um elemento dedicado à classificação de ativos de software (*Classification*, ilustrado na figura 5.3). Um esquema de catálogos de componentes pode ser adaptado, então, ao repositório do *Komponento*, através da estrutura RAS, em trabalhos futuros.

4. *Documentação completa de sistema e de componente.* A documentação de sistemas e componentes é realizada através dos artefatos previamente definidos pelo modelo de processo *Komponento*, como especificação de requisitos funcionais e não funcionais de sistemas e componentes, desenho de arquitetura de sistemas e componentes, ou testes de unidade de componentes. Os documentos produzidos pelas atividades de certificação (casos de teste) e análise de arquitetura (relatório de requisitos não funcionais) também são instrumentos eficazes para se medir qualidade.

5. *Política de controle de usuários para o acesso ao repositório.* O *Subversion* pode ser integrado ao servidor Apache (ilustrado na figura 5.6) e gozar das funcionalidades de autenticação e autorização do mesmo.

6. *Funções de controle de histórico de versões de um item de configuração.* Mais uma vez, a ferramenta *Subversion* possibilita a manutenção de históricos de versões de todos os itens sob controle de configuração em um repositório, implementando ainda mecanismos de *branching* e *tagging*, já mencionados na seção 2.4.1.1.

7. *Definição de uma atividade de gerência do repositório.* A atividade de gerência do repositório já foi detalhada na seção 4.1.2.2, constituindo-se das subatividades de gerência de repositórios, usuários e catálogos, garantia de qualidade, e controle de versões.

5.2.1. Ferramentas

Além da ferramenta *Subversion* de controle de versões, já explicada em seções anteriores, para a implementação do protótipo de repositório do *Komponento*, utilizaram-se ainda:

- *Eclipse IDE.* O ambiente de desenvolvimento utilizado para a implementação das classes de serviços de consulta ao repositório do *Komponento* foi o *Eclipse* 3.0.2 (ECLIPSE, 2005). É uma plataforma altamente extensível, com suporte ferramental para o desenvolvimento de programas em linguagem Java. A extensão dos recursos se dá através de uma arquitetura de *plugins*.

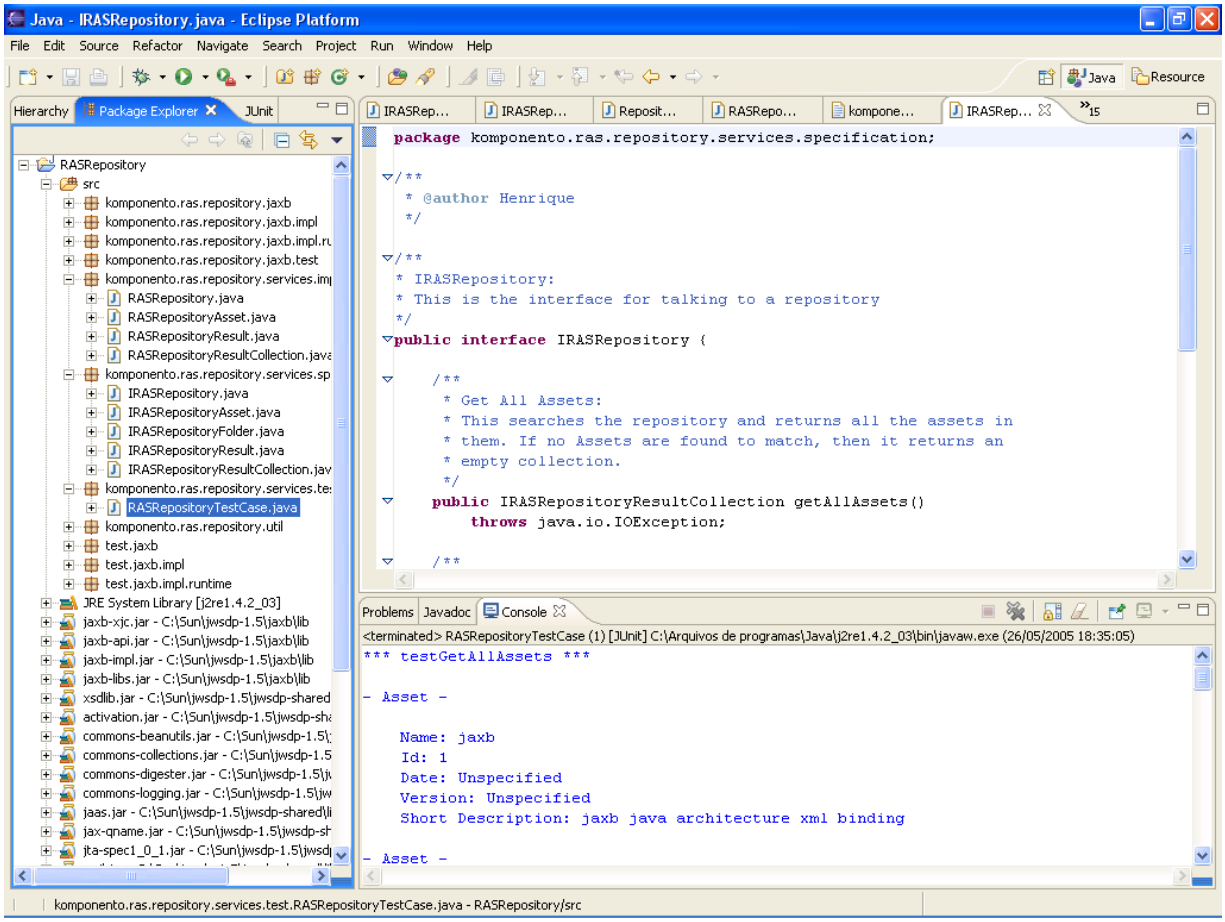


Figura 5.7. Ambiente de desenvolvimento da plataforma Eclipse.

O *plugin* é a menor unidade funcional do *Eclipse*, podendo ser integrado à plataforma, adicionando-lhe novas funcionalidades. Possuem pontos de extensão, podendo ser especializados. O protótipo de repositório apresentado na próxima seção faz uso de um *plugin* para testes de unidades em Java (*org.eclipse.jdt.junit plugin*). A figura 5.7 ilustra o ambiente de desenvolvimento da ferramenta.

- *JAXB (Java Architecture for XML Binding).* O JAXB (SUN, 2005a) permite o acesso e a utilização de documentos XML, através da linguagem de programação Java, de maneira eficiente. Ao invés de percorrer o documento, separá-lo logicamente em partes discretas, e repassar o conteúdo para a aplicação Java (ORT, 2003), assim

como o SAX (SAX, 2005) ou o DOM (W3C, 2005), o JAXB funciona em duas etapas.

- Compila-se, primeiramente, um esquema XML em códigos fonte de classes Java (figura 5.8).

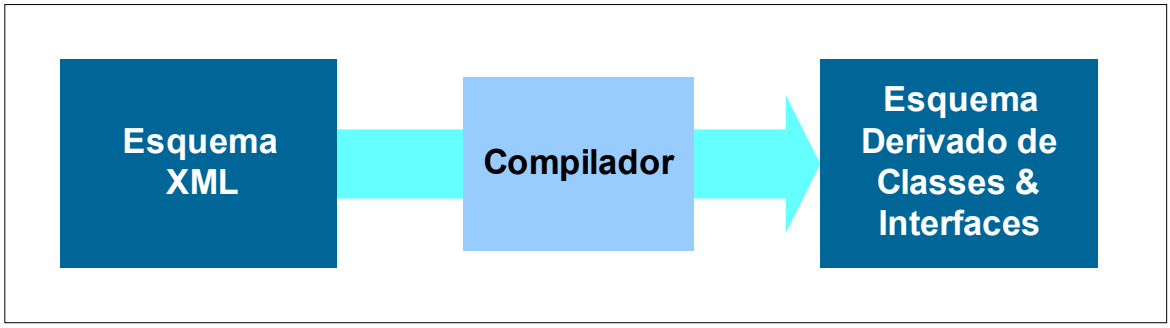


Figura 5.8. Mapeamento JAXB de um esquema XML em um conjunto de classes Java (ORT, 2003).

- Após o mapeamento (*binding*) de um esquema XML em um conjunto de classes, então, pode-se, através de uma aplicação que utilize as bibliotecas do JAXB e as classes geradas pelo compilador: (1) criar uma árvore de objetos em memória a partir de um documento XML (*unmarshalling*); (2) criar um documento XML a partir de objetos Java (*marshalling*); ou (3) atualizar um documento manipulando-se objetos em memória (figura 5.9).

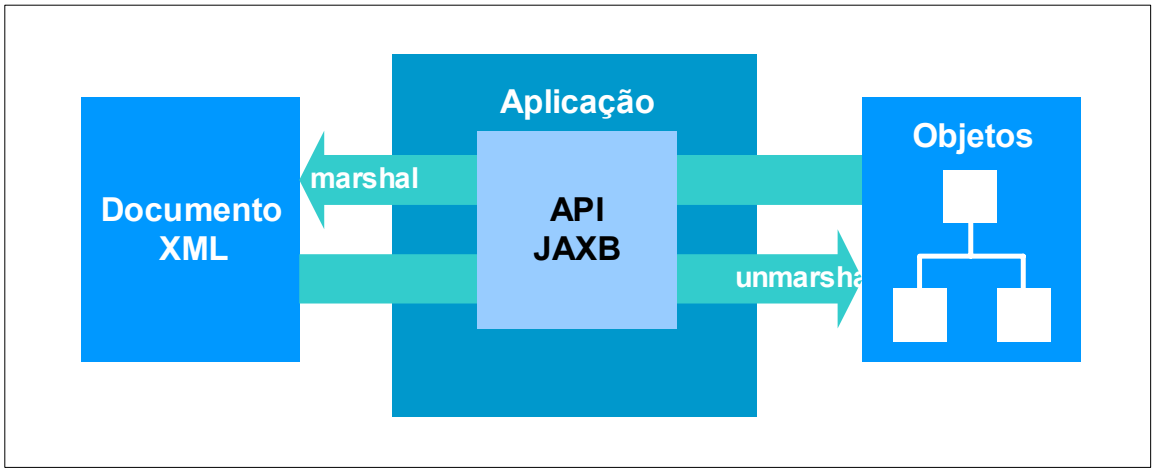


Figura 5.9. Marshalling de objetos em documentos XML e unmarshalling de documentos em objetos XML através do JAXB (ORT, 2003).

5.2.2. O protótipo

Entre as principais partes, que compuseram a implementação do protótipo de repositório do processo *Komponento*, podem-se identificar: um esquema XML simplificado, especificando os elementos mínimos de *Default Component Profile (Requirements, Design, Implementation* e *Test)*, num arquivo XSD; classes geradas pelo compilador JAXB, constituindo um mapeamento direto dos elementos do esquema XML, a serem instanciadas por serviços que implementem a API de consulta de ativos de um repositório, especificada pelo RAS; e, finalmente, a implementação das interfaces da API através de classes concretas (construídas através da linguagem Java). Estas partes estão descritas, respectivamente, nas seções 5.2.2.1, 5.2.2.2 e 5.2.2.3, a seguir.

5.2.2.1. Esquema XML

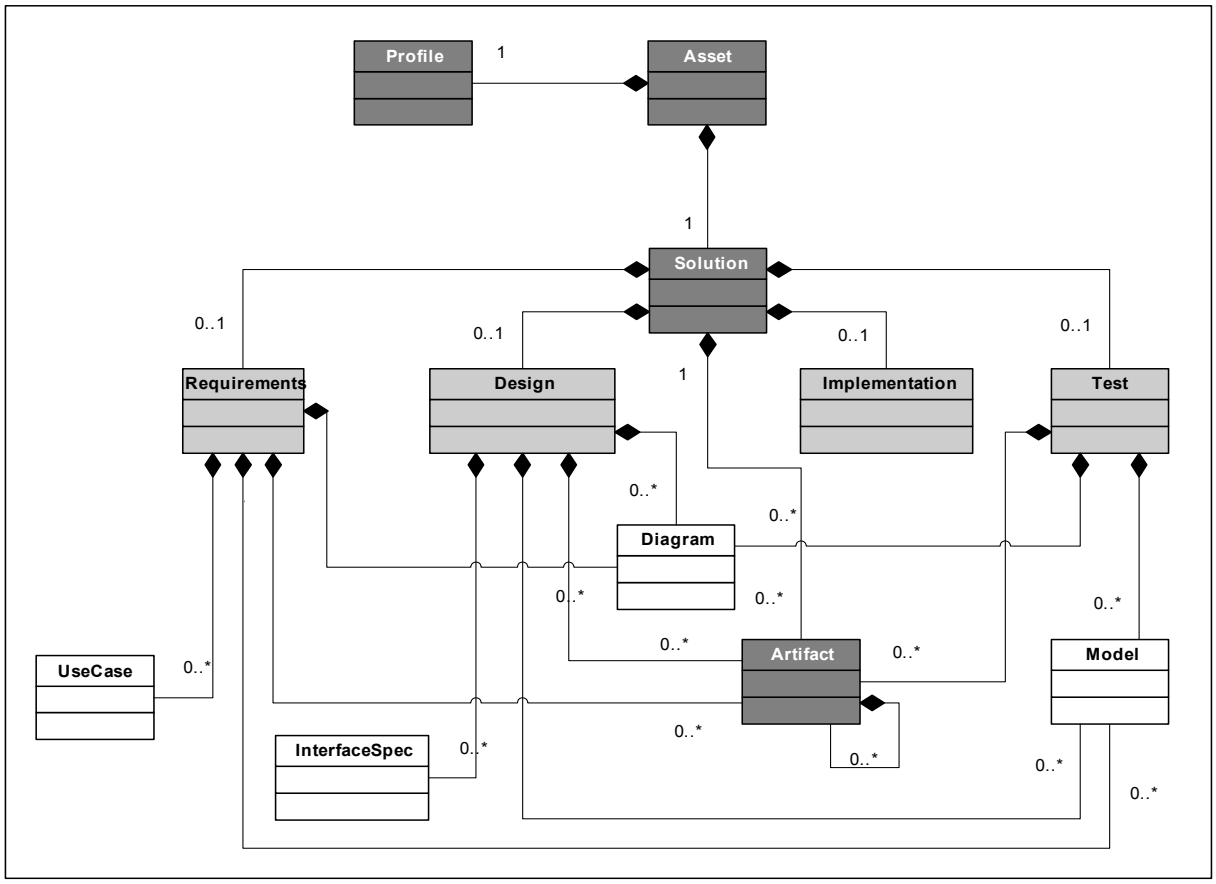


Figura 5.10. Elementos mínimos da estrutura de Default Component Profile para se garantir compatibilidade com o RAS (OMG, 2004).

O esquema XML montado para o protótipo, denominado *Komponento Profile*, retrata a estrutura de *Default Component Profile*. É um esquema simplificado, definindo apenas os elementos destacados na figura 5.10.

As classes e atributos mínimos obrigatórios para se obter compatibilidade com o modelo RAS estão relacionados na tabela abaixo. As classes mínimas são *Asset, Profile, Solution* e *Artifact* (na figura 5.10 estão destacadas numa tonalidade mais forte). Todas se encontram declaradas em *komponento_profile.xsd* (apresentado no final desta seção), que define ainda os elementos *Requirements*, *Design*, *Implementation* e *Test*, como coleções de artefatos.

Classe	Atributo	Descrição
Asset	*name*	Nome do ativo
Asset	*id*	Identificador único do ativo
Profile	*name*	Nome do perfil
Profile	*id-history*	Reflete o histórico de identificadores dos perfis de que herdam o perfil em questão.
Profile	*version-major*	Versão principal do perfil (ex. 1.5)
Profile	*version-minor*	Versão auxiliar do perfil (ex. 1.5)
Artifact	*name*	Nome do artefato
Artifact	*type*	Tipo de artefato
Artifact	*reference*	Localização de referência do artefato

Tabela 7. Classes e atributos mínimos para garatirem compatibilidade com o modelo RAS.

Os atributos obrigatórios de cada elemento (ex. *name* e *id* para o elemento *Asset*), no arquivo XSD aparecem como "required". A seguir, o arquivo XSD transcrito (*Komponento_profile.xsd*), definindo o perfil dos componentes armazenados no repositório do processo *Komponento*.

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
jaxb:version="1.0">

<xs:element name="asset">
  <xs:complexType>
```

```
    <xs:sequence>
      <xs:element name="profile" type="profileType"/>
      <xs:element name="solution" type="solutionType"/>
    </xs:sequence>
    <xs:attribute name="name" type="xs:string" use="required"/>
    <xs:attribute name="id" type="xs:string" use="required"/>
    <xs:attribute name="date" type="xs:date"/>
    <xs:attribute name="state" type="xs:string"/>
    <xs:attribute name="version" type="xs:string"/>
    <xs:attribute name="short-description" type="xs:string"/>
  </xs:complexType>
</xs:element>

<xs:complexType name="profileType">
    <xs:attribute name="name" type="xs:string" use="required"/>
    <xs:attribute name="id-history" type="xs:string" use="required"/>
    <xs:attribute name="version-major" type="xs:int" use="required"/>
    <xs:attribute name="version-minor" type="xs:int" use="required"/>
</xs:complexType>

<xs:complexType name="solutionType">
  <xs:sequence>

    <xs:element name="artifacts">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="artifact" type="artifactType"
maxOccurs="unbounded"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>

    <xs:element name="requirements" minOccurs="0">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="artifact" type="artifactType"
maxOccurs="unbounded"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
```

```
    <xs:element name="design" minOccurs="0">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="artifact" type="artifactType"
maxOccurs="unbounded"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>

    <xs:element name="implementation" minOccurs="0">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="artifact" type="artifactType"
maxOccurs="unbounded"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>

    <xs:element name="test" minOccurs="0">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="artifact" type="artifactType"
maxOccurs="unbounded"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>

  </xs:sequence>
</xs:complexType>

<xs:complexType name="artifactType">
    <xs:attribute name="name" type="xs:string" use="required"/>
    <xs:attribute name="type" type="xs:string" use="required"/>
    <xs:attribute name="reference" type="xs:string" use="required"/>
    <xs:attribute name="id" type="xs:string"/>
    <xs:attribute name="version" type="xs:string"/>
</xs:complexType>

</xs:schema>
```

5.2.2.2. Classes JAXB

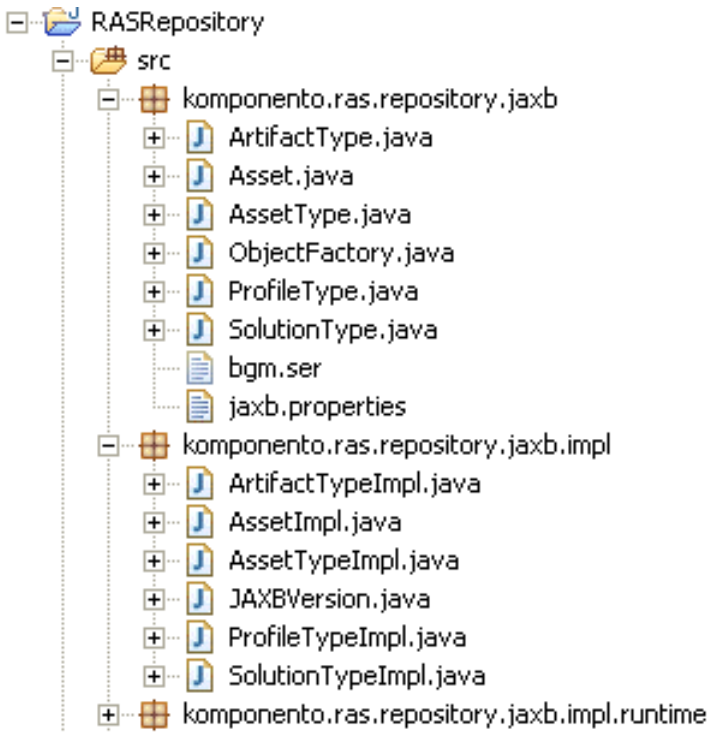


Figura 5.11. Estrutura de classes (no ambiente Eclipse) geradas pelo compilador JAXB a partir do esquema definido em komponento_profile.xsd.

A partir do esquema XML apresentado na seção anterior, então, geraram-se as classes de mapeamento dos elementos definidos pelo perfil de componente do RAS, através da ferramenta JAXB. A estrutura retratada abaixo é o produto do processo de compilação de *komponento_profile.xsd*.

Enquanto o pacote *komponento.ras.repository.jaxb* define as interfaces dos elementos de *Komponento Profile*, *komponento.ras.repository.jaxb.impl* define suas implementações. O pacote *komponento.ras.repository.jaxb.impl.runtime* é composto por classes de infra-estrutura da arquitetura JAXB.

Abaixo, a listagem dos métodos da interface *AssetType.java*, elemento principal definindo a raiz de *Komponento Profile*, a partir de onde são obtidos todos os demais sub-elementos (*ProfileType.java*, *SolutionType.java* e *ArtifactType.java*).

```
java.util.Calendar getDate();
java.lang.String getId();
java.lang.String getName();
java.lang.String getState();
java.lang.String getVersion();
java.lang.String getShortDescription();
komponento.ras.repository.jaxb.ProfileType getProfile();
komponento.ras.repository.jaxb.SolutionType getSolution();
void setDate(java.util.Calendar value);
void setId(java.lang.String value);
void setName(java.lang.String value);
void setState(java.lang.String value);
void setVersion(java.lang.String value);
void setShortDescription(java.lang.String value);
void setProfile(komponento.ras.repository.jaxb.ProfileType value);
void setSolution(komponento.ras.repository.jaxb.SolutionType value);
```

Assim, pode-se recuperar o pefil de um ativo através do método:

```
komponento.ras.repository.jaxb.ProfileType getProfile();
```

Ou o nome de um ativo através de:

```
java.lang.String getName();
```

ProfileType.java, por sua vez, disponibiliza os atributos de um perfil, através dos métodos:

```
java.lang.String getName();
int getVersionMajor();
int getVersionMinor();
java.lang.String getIdHistory();
```

5.2.2.3. Implementação da API de serviços

A implementação dos serviços de consulta de componentes de um repositório seguiu a API Java especificada pelo padrão RAS. A figura abaixo dá uma visão geral dos pacotes de: especificação (*komponento.ras.repository.service.specification*), que são as interfaces Java propriamente ditas e estabelecidas pela especificação RAS; e implementação, constituindo a parte principal do protótipo elaborado nesta pesquisa.

Figura 5.12. Os pacotes de implementação e especificação (no ambiente Eclipse) do protótipo de repositório compatível com a especificação RAS, para o processo Komponento.

A tabela abaixo relaciona as interfaces Java especificadas pelo RAS, dando a implementação, bem como uma descrição resumida de cada uma.

Interface	Implementação	Descrição
IRASRepository	RASRepository	O repositório RAS, disponibilizando serviços de consulta de componentes.
IRASRepositoryAsset	RASRepositoryAsset	Um ativo do repositório, representando um componente.
IRASRepositoryFolder	RASRepositoryFolder	Um diretório do repositório.
IRASRepositoryResult	RASRepositoryResult	Um resultado genérico individual.
IRASRepositoryResult Collection	RASRepositoryResult Collection	Coleção de artefatos genéricos retornados pelo resultado de uma consulta.

Tabela 8. Interfaces, implementações e descrições das classes da API de consulta a um repositório de componentes especificada por (OMG, 2004).

IRASRepository.java é a interface principal, fornecendo serviços de consulta de ativos de acordo com diferentes critérios (ex. *searchByLogicalPath*). A classe *IRASRepositoryResultCollection.java* representa uma coleção de ativos genéricos retornados por uma consulta no repositório. A interface dos ativos genéricos é especificada em *IRASRepositoryResult.java*. Os ativos, por sua vez, podem ser arquivos (*IRASRepositoryAsset.java*) ou diretórios (*IRASRepositoryFolder.java*). A figura abaixo retrata a estrutura estática das classes que implementam os serviços de um repositório RAS, através de um diagrama UML de classes.

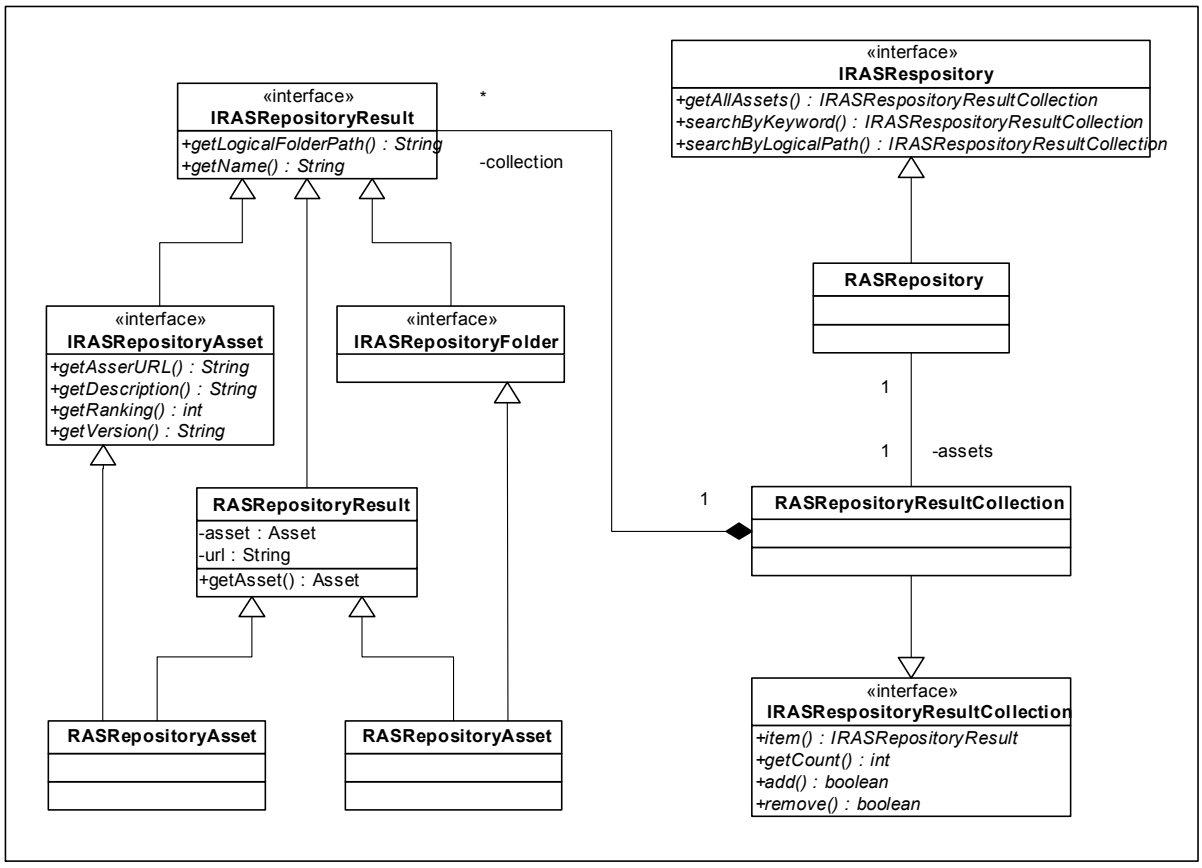


Figura 5.13. Diagrama de classes da implementação do repositório de componentes.

Os arquivos lógicos armazenados no repositório, representando os componentes disponíveis, são, na implementação do *Komponento*, documentos XML dentro do formato definido em *komponento_profile.xsd*. O conteúdo destes documentos é, basicamente, uma lista de artefatos identificados pelos seus atributos obrigatórios de acordo com o padrão RAS. Dentre estes atributos, *reference* dá a localização relativa ao repositório do artefato em questão.

Abaixo, um exemplo de estrutura lógica de um ativo armazenado no repositório:

```
<?xml version="1.0"?>
<asset name="jaxb" id="1" short-description="jaxb java architecture xml
binding">
```

```
  <profile name="Komponento Profile" id-
history="komponento_03052005::core_03052005"
           version-major="1" version-minor="0"/>

  <solution>

    <artifact name="jaxb" type="Folder" reference="/"/>
    <artifact name="jwsdp.properties" type="File"
reference="/LICENSE"/>

    <design>
      <artifact name="index.html" type="File" reference="/jaxb/docs"/>
    </design>

    <implementation>
      <artifact name="src" type="Folder"
reference="/jaxb/samples/unmarshal-validate"/>
      <artifact name="jwsdp.properties" type="File"
reference="/jaxb/conf"/>
    </implementation>

    <test>
      <artifact name="samples" type="Folder" reference="/jaxb"/>
    </test>

  </solution>

</asset>
```

O ativo acima, denominado “jaxb”, cujo perfil é *Komponento Profile*, possui os artefatos destacados em negrito e itálico, categorizados de acordo com a estrutura de *Default Component Profile* do RAS. A localização física de cada artefato é dada pela combinação dos atributos *reference* + *name*.

Os passos para se fazer uma busca de um componente no protótipo de repositório do *Komponento* seriam:

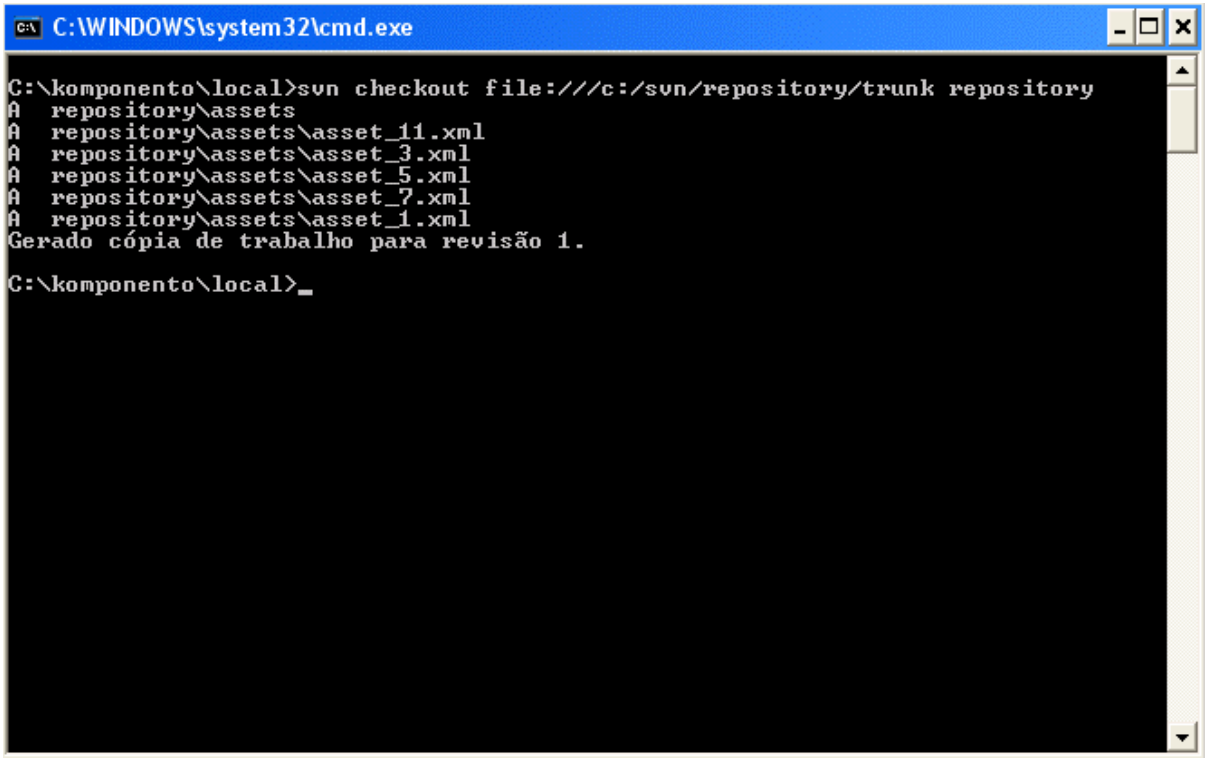


Figura 5.14. Exemplo de cópia local de repositório remoto de arquivos lógicos contendo a estrutura dos componentes disponíveis.

1. Baixar a cópia do repositório de componentes localmente. É importante lembrar que o repositório de componentes do protótipo é, na verdade, um diretório contendo uma lista de documentos XML. Isto evita a transferência de todos os dados, no caso de cópias físicas dos artefatos de cada ativo na base. Na figura abaixo, copiaram-se os componentes do repositório *c:/svn/repository/trunk* para o diretório local *c:/komponento/local/repository*.

2. Configurar o serviço de consulta ao repositório (implementado pelo pacote de classes da figura 5.12) para pesquisar o diretório copiado no passo 1. No protótipo do *Komponento*, isso pode ser feito alterando-se a constante REPOSITORY_PATH em *RASRespository.java* (ou simplesmente passando-se o caminho no construtor):

```
private static final String REPOSITORY_PATH =
    "c:/komponento/local/repository/assets";
```

REPOSITORY_PATH é o caminho que o sistema vai procurar para fazer a operação de *unmarshalling*, através dos componentes JAXB, em todos os arquivos XML contidos neste diretório. Isto é implementado no método *init()* da classe *RASRespository.java*, conforme transcrito abaixo:

```
1     private void init(String path) {
2           try {
3            JAXBContext jc =
4               JAXBContext.
5                   newInstance("komponento.ras.repository.jaxb");
6            Unmarshaller unmarshaller = jc.createUnmarshaller();
6            unmarshaller.setValidating(true);
7
8            File repository = new File(path);
10            File[] files = repository.listFiles();
11
12           for (int i = 0; i < files.length; i++) {
12               Asset asset =
13                     (Asset)unmarshaller.unmarshal(files[i]);
14               assets.add
15                 (new RASRepositoryAsset(asset,
16                         files[i].getAbsolutePath()));
17           }
18       } catch (Exception e ) {
19           e.printStackTrace();
20       }
21   }
```

Da linha 11 à 16, o código itera em cada um dos componentes, representados em documentos XML. Na linha 11 é feito o *unmarshalling* dos dados nos objetos JAXB (seção 5.2.2.2). Um *komponento.ras.repository.jaxb.Asset* é retornado pelo método *unmarshal*, e repassado para o construtor de *RASRepositoryAsset*, que o encapsula a fim de disponibilizar informações a respeito do ativo para os serviços de consulta do repositório. A variável *assets*, que aparece na linha 13 é a coleção de artefatos do repositório, instância da classe *RASRepositoryResultCollection*.

```
RASRepositoryResultCollection assets =
    new RASRepositoryResultCollection();
```

3. Realizar a pesquisa do componente desejado, através dos critérios disponíveis na interface de serviços, no diretório copiado localmente. Para o protótipo do repositório do *Komponento* não foi desenvolvida uma interface com o usuário, porém, foram projetados casos de teste para a consulta de componentes, executando uma consulta para cada um dos métodos de *IRASRepository*:

```
public IRASRepositoryResultCollection getAllAssets()
          throws java.io.IOException {
          return assets;
}

public IRASRepositoryResultCollection searchByKeyword
       (String theKeyword)
    throws java.io.IOException {
    RASRepositoryResultCollection result =
           new RASRepositoryResultCollection();

    for (int i = 0; i < assets.getCount(); i++) {
      if (((RASRepositoryAsset)assets.item(i)).
            getDescription().indexOf(theKeyword) != -1) {
```

```
            result.add(assets.item(i));
          }
        }

        return result;
    }

    public IRASRepositoryResultCollection searchByLogicalPath
            (String theLogicalPath)
        throws java.io.IOException {
        RASRepositoryResultCollection result =
               new RASRepositoryResultCollection();

        for (int i = 0; i < assets.getCount(); i++) {
          if (((RASRepositoryAsset)assets.item(i)).
                 getLogicalFolderPath().equals(theLogicalPath)) {
              result.add(assets.item(i));
          }
        }

        return result;
    }
```

O resultado das consultas é listado na saída padrão de acordo com o formato abaixo:

```
- Asset -

    Name:
    Id:
    Date:
    Version:
    Short Description:

    - Profile -

        Name:
        Id History:
        Version:

    - Solution -

        - Artifacts | Requirements | Design | Implementation | Test
-

            - Artifact Details -

                Name:
                Type:
```

```
            Reference:
            Version:
            Id:

        ...

        - Artifact Details -

            Name:
            Type:
            Reference:
            Version:
            Id:
```

O sistema realiza uma busca por palavras chaves, através do método *searchByKeyword*, consultando o atributo *short-description* e verificando a ocorrência da palavra chave no valor do atributo. Uma busca pela palavra chave "xml", no repositório do exemplo, traria o resultado abaixo:

```
*** testSearchByKeyword ***


- Asset -

    Name: jaxb
    Id: 1
    Date: Unspecified
    Version: Unspecified
    Short Description: jaxb java architecture xml binding

    - Profile -

        Name: Komponento Profile
        Id History: komponento_03052005::core_03052005
        Version: 1.5

    - Solution -

        - Artifacts -

            - Artifact Details -

                Name: jaxb
                Type: Folder
                Reference: /
                Version: Unspecified
                Id: Unspecified

            - Artifact Details -

                Name: jwsdp.properties
                Type: File
                Reference: /LICENSE
```

```
                Version: Unspecified
                Id: Unspecified

        - Design -

            - Artifact Details -

                Name: index.html
                Type: File
                Reference: /jaxb/docs
                Version: Unspecified
                Id: Unspecified

        - Implementation -

            - Artifact Details -

                Name: src
                Type: Folder
                Reference: /jaxb/samples/unmarshal-validate
                Version: Unspecified
                Id: Unspecified

            - Artifact Details -

                Name: jwsdp.properties
                Type: File
                Reference: /jaxb/conf
                Version: Unspecified
                Id: Unspecified

        - Test -

            - Artifact Details -

                Name: samples
                Type: Folder
                Reference: /jaxb
                Version: Unspecified
                Id: Unspecified

- Asset -

    Name: jaxp
    Id: 11
    Date: Unspecified
    Version: Unspecified
    Short Description: jaxp java api xml processing

    - Profile -

        Name: Komponento Profile
        Id History: komponento_03052005::core_03052005
        Version: 1.5

    - Solution -

        - Artifacts -

            - Artifact Details -
```

```
                Name: jaxp
                Type: Folder
                Reference: /
                Version: Unspecified
                Id: Unspecified

            - Artifact Details -

                Name: LICENSE
                Type: File
                Reference: /
                Version: Unspecified
                Id: Unspecified

        - Design -

            - Artifact Details -

                Name: index.html
                Type: File
                Reference: /jaxp/docs
                Version: Unspecified
                Id: Unspecified

        - Implementation -

            - Artifact Details -

                Name: jaxp-api.jar
                Type: File
                Reference: /jaxp/lib
                Version: Unspecified
                Id: Unspecified

            - Artifact Details -

                Name: DOMEcho.java
                Type: File
                Reference: /jaxp/samples/DOMEcho
                Version: Unspecified
                Id: Unspecified

        - Test -

            - Artifact Details -

                Name: samples
                Type: Folder
                Reference: /jaxp
                Version: Unspecified
                Id: Unspecified

- Asset -

    Name: jaxr
    Id: 3
    Date: Unspecified
    Version: Unspecified
    Short Description: jaxp java api xml registries

    - Profile -
```

```
    Name: Komponento Profile
    Id History: komponento_03052005::core_03052005
    Version: 1.5

- Solution -

    - Artifacts -

        - Artifact Details -

            Name: jaxr
            Type: Folder
            Reference: /
            Version: Unspecified
            Id: Unspecified

        - Artifact Details -

            Name: LICENSE
            Type: File
            Reference: /
            Version: Unspecified
            Id: Unspecified

    - Design -

        - Artifact Details -

            Name: index.html
            Type: File
            Reference: /jaxr/docs
            Version: Unspecified
            Id: Unspecified

    - Implementation -

        - Artifact Details -

            Name: jaxr-browser.bat
            Type: File
            Reference: /jaxr/bin
            Version: Unspecified
            Id: Unspecified

        - Artifact Details -

            Name: jaxr-browser.sh
            Type: File
            Reference: /jaxr/bin
            Version: Unspecified
            Id: Unspecified

    - Test -

        - Artifact Details -

            Name: samples
            Type: Folder
            Reference: /jaxr
            Version: Unspecified
```

```
Id: Unspecified
```

Com base nas informações sobre a localização de cada artefato no repositório, é possível se montar uma cópia local do componente, baixando-se os artefatos pertinentes ao mesmo, mantendo-se a estrutura de diretórios. Os arquivos XML e uma listagem de todos os ativos do repositório de exemplo encontram-se no apêndice B.

6. Considerações finais

Em relação ao emprego do *Komponento* no apoio a processos de desenvolvimento de software, podem-se citar vantagens como: centralização de uma base comum de componentes, visando-se retrabalho desnecessário, no caso da existência de algum tipo de solução de software, publicada no repositório, aplicável ao problema em questão; uma abordagem prática de certificação de componentes através dos casos de teste e dos relatórios de qualidade propostos pela metodologia ATAM; e, finalmente, um certo nível de sistematização dos processos de reutilização de software, no que diz respeito ao consumo e à produção de componentes em um repositório legado.

Das desvantagens, pode-se citar um problema muito comum relacionado à implantação de quaisquer processos: a dificuldade de se institucionalizar um novo processo dentro de uma organização. Hábitos e costumes devem ser mudados, bem como um sistema de métricas deve ser adotado para uma avaliação quantitativa dos resultados do processo.

6.1. Conclusões

A definição do modelo de processo *Komponento* de apoio a outros processos orientados ao DBC foi totalmente baseada nos elementos do meta-modelo SPEM. Todos os elementos mínimos de um processo (atividades, artefatos, papéis, ciclo de vida etc.) puderam ser representados através dos pacotes fundamentais, bem como das extensões do SPEM.

Quanto ao atendimento dos requisitos da norma ISO/IEC 12207, o processo apresentou abordagens e implementações mapeadas às diretivas de cada uma das áreas de processo com as quais se propôs alinhar. Considerando-se, então, os resultados que as áreas de processo

cobertas (processo de gerência de configuração, processo de garantia de qualidade, e gerência de ativos) devem alcançar:

1. Processo de gerência de configuração

    a. *Uma estratégia de gerência de configuração será desenvolvida.*
    b. *Todos os itens gerados pelo processo ou projeto serão identificados, definidos, e servirão de base.*
    c. *Modificações e entrega dos itens serão controlados.*
    d. *Os estados dos itens e requisições de modificação serão registrados e relatados.*
    e. *A integridade e consistência dos itens serão garantidas.*
    f. *Armazenamento, manipulação, e entrega dos itens serão controlados.*

2. Processo de garantia de qualidade

    a. *Uma estratégia para conduzir a garantia de qualidade será desenvolvida, implementada e mantida.*
    b. *Evidência de garantia de qualidade será produzida e mantida.*
    c. *Problemas ou inconformidades com os requisitos do contrato serão identificados.*
    d. *A aderência de produtos, processos e atividades aos padrões, procedimentos e requisitos aplicáveis será verificada objetivamente.*

3. Gerência de ativos

a. *Um plano de gerência de ativos será documentado.*

b. *Um mecanismo de armazenamento e recuperação será operacionalizado.*

c. *A utilização de ativos será registrada.*

d. *Gerência de configuração será executada para os ativos.*

E as implementações e abordagens realizadas pelo processo *Komponento* para este fim:

- *Subversion.* Os recursos da ferramenta *Subversion*, como controle de histórico de versões, integridade e consistência de dados sob controle de configuração (ex. mecanismo de *merge*), *branching* e *tagging*, atenderam muitos dos requisitos gerias e específicos de processo do *Komponento*.

- *Atividade de produção*. A atividade de produção de componentes descreve os passos a serem seguidos para publicação de componentes no repositório.

- *Bibliotecário de reuso.* O bibliotecário de reuso, papel estipulado pelo *Komponento*, deve garantir que os componentes publicados no repositório satisfaçam determinados padrões de qualidade; catalogar e classificar tais componentes; e gerenciar tanto a adição (produção), quanto a utilização (consumo) da biblioteca de componentes.

- *Impementações RAS.* O meta-modelo RAS de perfis, para armazenamento de ativos, e a API Java de serviços de consulta de um repositório RAS foram partes do padrão implementadas pelo processo.

- *Certificação de componentes*. O processo de certificação de componentes através de pacotes de testes para verificação e validação, principalmente, de funcionalidades e desempenho de componentes garante certo nível de qualidade, de maneira tangível.

- *ATAM.* A norma ISO/IEC 9126 orientou a avaliação de qualidade pela metodologia ATAM, através do modelo da árvore de qualidade, composta por documentos descrevendo as abordagens para o atendimento de requisitos dentro dos grupos de confiabilidade, usabilidade, manutenibilidade etc.

- *XMI.* A especificação XMI (*XML Metadata Interchange*) permite a persistência de meta-modelos baseados em UML através de esquemas XML, possibilitando o suporte deste padrão por ferramentas CASE. Este trabalho não cobre a especificação de formatos XMI para os meta-modelos utilizados, porém abre espaço para futuras pesquisas.

- *Atividade de gerência.* A atividade de gerência estabelece subatividades de cunho administrativo a serem realizadas durante a execução do processo *Komponento*, tais como gerência de repositórios, catálogos e usuários.

Pôde-se montar a tabela 9 de alinhamento do modelo de processo *Komponento* aos requisitos estabelecidos pelas áreas cobertas na norma ISO/IEC 12207.

Requisitos de processo / Implementações	Gerência de configuração (1)						Garantia de qualidade (2)				Gerência de ativos (3)			
	a	b	c	d	e	f	a	b	c	d	a	b	c	d
Subversion	X			X	X							X	X	X
Atividade de produção		X												X
Bibliotecário de reuso			X											X
Implementações RAS						X						X		X
Certificação de componentes							X	X	X					
ATAM							X	X	X	X				
XMI										X				
Atividade de gerência											X			

Tabela 9. Mapeamento dos requisitos estabelecidos pela norma ISO/IEC 12207 nas abordagens do processo Komponento.

O protótipo desenvolvido em Java, que seguiu a especificação RAS da API de consulta de componentes de um repositório, serviu como prova de conceito tanto do meta-modelo de armazenamento de ativos (mais especificamente do *Default Component Profile*), através da construção de esquemas XML, quanto da API de serviços de consulta, cuja implementação se baseou na arquitetura JAXB. O JAXB serviu como a ponte entre os esquemas e objetos XML, derivados destes esquemas, e manipulados pela interface RAS de consulta a um repositório. A implementação das APIs do RAS torna o serviço compatível, então, com aplicações que pretendam consultar ativos em um repositório, através das interfaces Java especificadas.

Como forma de se garantir qualidade a componentes de software, este trabalho utilizou a combinação de um modelo, organizando todas as atividades pertinentes a um processo de reutilização de componentes, e de um repositório, implementado-o na forma de protótipo para armazenamento e consulta de componentes. O modelo de processo e o repositório permitiram um ambiente integrado para a proposta de metodologias de garantia de qualidade

através dos processos de certificação de componentes e de avaliação de arquiteturas (alinhada à norma ISO/IEC 9126). Os produtos das subatividades de certificação de componentes e avaliação de arquiteturas (casos de teste e relatórios de requisitos não-funcionais, respectivamente) serviram como uma documentação bastante tangível à respeito da qualidade dos componentes armazenados no repositório.

O processo foi todo definido através de padrões abertos: o SPEM, como um perfil UML, forneceu elementos de processo que descreveram toda a estrutura do *Komponento*; e o meta-modelo RAS de perfis de armazenamento de ativos foi seguido para a implementação dos esquemas XML, bem como da API Java de serviços de consulta. A adoção de padrões abertos oferece vantagens como: portabilidade entre diferentes plataformas (ex. um documento no formato XMI pode ser interpretado por diferentes ferramentas CASE); liberdade de escolha, no sentido de que quanto mais alternativas para uma determinada solução forem implementadas para um determinado padrão, maior a flexibilidade de escolha entre diferentes softwares ou processos de software; menores custos de integração entre sistemas que são construídos para especificações abertas e comuns entre diferentes sistemas (BIRD, 1998). A escolha de padrões abertos foi a estratégia para tornar o processo o mais adaptável possível a outros ambientes de desenvolvimento de sofwtare, atendendo o objetivo de se projetar um modelo de processo acoplável. A utilização de um meta-modelo para a definição do modelo foi determinante para a compatibilidade do *Komponento* com outros processos. Os meta-elementos do SPEM proporcionaram uma linguagem comum para a combinação de processos (como o exemplo de aplicação apresentado na seção 4.3) em um nível mais alto de abstração.

6.2. Trabalhos futuros

Este trabalho mostrou ainda que, por mais pesados que possam ser determinadas normas ou padrões, como o próprio ISO/IEC 12207 (cobrindo uma diversidade de áreas em um nível bastante formal de detalhamento), é possível adaptá-los a sistemas menores. Em relação a trabalhos futuros, modelos de maturação de processos como o CMM ou o ISO/IEC 15504 podem ser aplicados em estudos sobre a instanciação do modelo *Komponento* através da execução do processo em um ambiente real de desenvolvimento de software. Faria e Arakaki (FARIA; ARAKAKI, 2003) apresentam um estudo de caso da adaptação do SW-CMM (modelo relativo aos processos específicos de software, englobado hoje pelo CMMI) em pequenas empresas, inspirado no trabalho de Paulk (PAULK, 1998). Alternativamente, empresas podem ainda adotar o ISO/IEC 15504. Paulk relaciona ainda diversas intersecções e semelhanças entre ambos os modelos (PAULK, 1999).

Finalmente, os esquemas XML de definição de componentes armazenados no formato RAS, em um repositório de componentes, podem ser implementados dentro do padrão XMI, para a persistência de dados de modelos UML, em sistemas de arquivos ou bancos de dados. Isto permitiria o suporte por ferramentas CASE e o armazenamento de ativos em formato de arquivos XSD no padrão XMI.

7. Referência bibliográfica

ALLEN, P. The Reusable Asset Specification, 2002. Disponível em: <http://www.cutter.com/research/2002/edge020212.html>. Acesso em: 13 junho 2005.

ALUR, D.; CRUPI, J.; MALKS, D. **Core J2EE Patterns:** As melhores práticas e estratégias de design. Tradução Altair Dias Caldas de Moraes; Cláudio Belleza Dias; Guilherme Dias Caldas de Moraes. Rio de Janeiro: Campus, 2002. 406 p.

AOYAMA, M. CBSE in Japan and Asia. In: HEINEMAN, G. T.; COUNCIL, W. T. **Component-Based Software Engineering.** 1ª ed. Addison-Wesley, 2001. p. 213-225.

APACHE. APACHE DB PROJECT. Object Relational Bridge, 2005. Disponível em: <http://db.apache.org/ojb>. Acesso em: 14 junho 2005.

APACHE. APACHE JAKARTA PROJECT. Apache Tomcat, 2005. Disponível em: <http://jakarta.apache.org/tomcat>. Acesso em: 14 junho 2005.

APACHE. APACHE SOFTWARE FOUNDATION. Log4j Project, 2005. Disponível em: <http://logging.apache.org/log4j>. Acesso em: 14 junho 2005.

APPERLY, H. Configuration Management and Component Libraries. In: HEINEMAN, G. T.; COUNCIL, W. T. **Component-Based Software Engineering.** 1ª ed. Addison-Wesley, 2001. p. 513-526.

ATKINSON C.; BAYER, J.; MUTHIG, D. Component-based product line development: The KobrA approach. In: 1st Software Product Line Conference, 2000, Pittsburg. **Proceedings…** 2000.

BASILI, V. The Goal Question Metric Approach. In: MARCINIAK, J. **Encyclopedia of Software Engineering.** Vol. 1, John Wilet & Sons, 1994, p. 528-532.

BASS, L.; CLEMENTS, P.; KAZMAN, R. **Software Architecture in Practice.** Massachusetts: SEI Series in Software Engineering, 1999. 452 p.

BERTOLINO, A.; POLINI, A. A Framework for Component Deployment Testing. In: 25th International Conference on Software Engineering, 2003, Portland. **Proceedings.** 2003, p. 221-231.

BIRD, G. The Business Benefits of Standards. **StandardView**, vol. 6, n. 2, p. 76-80, jun. 1998.

BOEHM, B. Managing Software Productivity and Reuse. **Computer**, vol. 32, n. 9 p. 111-113, set. 1999.

BROWN, A. W.; WALLNAU, K. C. The Current State of CBSE. **IEEE Software**, vol. 13, n. 9, p. 37-46, set./out. 1998.

CASANOVA, M.; STRAETEN, R. V.; JONCKERS, V. Supporting Evolution in Component-Based Development Using Component Libraries. In: Seventh European Conference on Software Maintenance and Reengineering (CSMR'03), 2003, Benevento. **Proceedings…** 2003, p. 123-132.

CAVANESS, C. **Programming Jakarta Struts.** O'Reilly, 2003. p. 441.

CHEESMAN, J.; DANIELS, J. **UML Components:** A Simple Process for Specifying Component-Based Software. Addison Wesley, 2001. 176 p.

CLEMENTS, P.; KAZMAN, R.; KLEIN, M. Evaluating Software Architectures: Methods and Case Studies. Addisin-Wesley, 2002. 323 p.

CLEMENTS, P.; NORTHROP, L. **Software Product Line:** Practices and Patterns. SEI Series in Software Engineering, 2002. 563 p.

CONAN, D.; PUTRYCZ, E.; FARCET N.; DEMIGUEL, M. Integration of Non-Functional Properties in Containers. In: Sixth International Workshop on Component-Oriented Programming (WCOP), 2001. **Proceedings…** 2001.

COUNCILL, B.; HEINEMAN, G. T. Definition of a Software Component and Its Elements. In: HEINEMAN, G. T.; COUNCIL, W. T. **Component-Based Software Engineering.** 1ª ed. Addison-Wesley, 2001. p. 5-19.

DAGNINO, A. et al. A model to evaluate the economic benefits of software components development. In: International Conference on Systems, Man and Cybernetics, 2003. **Proceedings…** 2003, p. 3792-3797, vol. 4.

ECLIPSE. Eclipse IDE, 2005. Disponível em: <http://www.eclipse.org>. Acesso em: 13 junho 2005.

EELES, P. Capturing Architectural Requirements. The Rational Edge, novembro, 2001. Disponível em: <http://www-106.ibm.com/developerworks/rational/library/content/RationalEdge/archives/nov01.html>. Acesso em: 23 setembro 2004.

ENGELS, G., HECKEL, R.; SAUER, S. UML – A Universal Modeling Language? In: 21st International Conference on Application and Theory of Petri Nets, 2000, Aarhus. **Proceedings.** Springer-Verlag GmbH, 2003, p. 24.

FARIA, H.; ARAKAKI, R. Um Estudo de Caso de Adaptação do Modelo SW-CMM em Microempresas de TI. In: SUCESU 2003 - Congresso Nacional de Tecnologia da Informação e Comunicação, 2003, Salvador. **Anais.** Anais da SUCESU 2003, 2003.

FIRESMITH, D. G.; HENDERSON-SELLERS, B. **The OPEN Process Framework.** Addison-Wesley, 2002. 330 p.

FLYNT, J.; DESAI, M. The Future of Software Components: Standards and Certification. In: HEINEMAN, G. T.; COUNCIL, W. T. **Component-Based Software Engineering.** 1ª ed. Addison-Wesley, 2001. p. 693-708.

FOWLER, M. **UML Distiled:** A Brief Guide to the Standard Object Modeling Language. 3ª ed. Addison-Wesley, 2004. 175 p.

FUCAPI. Brasil investe em biblioteca de software livre, 2004. Disponível em: <http://portal.fucapi.br>. Acessado em: 23 setembro 2004.

GAMMA, E.; HELM, R.; JOHNSON, R. VLISSIDES, J. **Design Patterns:** Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995. 416 p.

GUO, J.; LUQI. A Survey of Software Reuse Repositories. In: The Seventh IEEE International Conference and Workshop on the Engineering of Computer Based Systems. **Proceedings…** 2000, p. 92-100.

HAUSEN, H. A software Assessment and Certification Advisor. In: ACM Conference on Computer Science. **Proceedings…** 1992, p. 309-316.

HENNINGER, S. An Evolutionary Approach to Constructing Effective Software Reuse Repositories. **ACM Transactions on Software Engineering and Methodologies**, v. 16, n. 2, p. 111-140, abr. 1997.

HENDERSON-SELLERS, B.; COLLINS, G.; DUÉ, R.; GRAHAM, I. A Qualitative Comparison of Two Processes For Object-Oriented Software Development. **Information & Software Technology**, v. 43, n. 12, p 705-724, nov. 2001.

HENDERSON-SELLERS, B. et al. Third generation OO processes: a critique of RUP and OPEN from a project management perspective. In: Seventh Asia-Pacific Software Engineering Conference, 2000, Singapore. **Proceedings…** pp. 428-435.

ISO/IEC, ISO/IEC 12207 International Standard. **Information Technology – Software Life Cycle Processes.** International Organization for Standardization, International Electrotechnical Commission, Geneva, 1999.

ISO/IEC, ISO/IEC 9126 International Standard. **Information Technology – Software product evaluation – Quality characteristics and guidelines for their use.** International Organization for Standardization, International Electrotechnical Commission, Geneva, 1991.

ISO/IEC, ITU Recommendation X.902. **Open Distributed Processing-Reference Model – Part 2: Foundations.** International Organization for Standardization, International Electrotechnical Commission, Geneva, 1995.

JAAKSI, A. Developing Mobile Browsers in a Product Line. **IEEE Software**, Los Alamitos, v. 19, n. 4, p. 73-80, jul./aug. 2002.

JACOBSON, I.; BOOCH, G.; RUMBAUGH, J. **The Unified Software Development Process.** 1ª ed. Addison-Wesley Professional, 1999. 463 p.

KRUCHTEN, P. **The Rational Unified Process:** An Introduction. Addison-Wesley, 2000. 298 p.

KRUCHTEN, P. **The 4+1 View Model of Architecture.** Software, Los Alamitos, v. 12, n. 6, p. 42-50, nov./dec. 1995.

LEMOS, M.; LEÃO, B. The Brazilian National Health Card Project. In: 8th International Congress in Nursing Informatics, 2003, Rio de Janeiro. **Proceedings…** 2003.

LIU, A.; GORTON, I. Accelerating COTS Middleware Acquisition: The i-Mate Process. **IEEE Software**, Los Alamitos, v. 20, n. 2, p. 72-79, mar./apr. 2003.

MCGIBBON, B. Status of CBSE in Europe. In: HEINEMAN, G. T.; COUNCIL, W. T. **Component-Based Software Engineering.** 1ª ed. Addison-Wesley, 2001. p. 199-212.

MORISIO, M.; TULLY, C.; EZRAN, M. Diversity in Reuse Process. **IEEE Software**, Los Alamitos, v. 17, n. 4, p. 56-63, jul./ago. 2000.

MORRIS, J. et al. Software Component Certification. **Computer**, vol. 34, n. 9 p. 30-36, set, 2001.

NIST. National Institute of Standards and Technologies. U.S. Department of Commerce. Modes of Operation Validation System (MOVS): Requirements and Procedures. Disponível em: <http://csrc.nist.gov/publications/nistpubs/800-17/800-17.pdf>. Acesso em: 10 maio 2005.

OMG, Meta-Object Facility (MOF) Specification, 2002. Disponível em: <http://www.omg.org/technology/documents/formal/mof.htm>. Acesso em: 29 outubro 2004.

OMG, Reusable Asset Specification, 2004. Disponível em: <http://www.omg.org/cgi-bin/doc?ptc/2004-06-06>. Acesso em: 29 outubro 2004.

OMG, Software Process Engineering Metamodel, 2005. Disponível em: <http://www.omg.org/technology/documents/formal/spem.htm>. Acesso em: 29 outubro 2004.

OMG, Unified Modeling Language (UML), 2003. Disponível em: <http://www.omg.org/technology/documents/formal/uml.htm>. Acesso em: 5 junho 2005.

OMG, XML Metadata Interchange (XMI), 2002. Disponível em: <http://www.omg.org/technology/documents/formal/xmi.htm>. Acesso em: 29 outubro 2004.

ORT, E.; MEHTA, B. Java Architecture for XML Binding (JAXB), 2003. Disponível em: <http://java.sun.com/developer/technicalArticles/WebServices/jaxb/index.html>. Acesso em: 5 junho 2005.

OSTERWEIL, L. Software Processes are Software Too. In: 9th International Conference on Software Engineering, 1987, Monterey. **Proceedings…** 1987, p. 2-13.

OSTERWEIL, L. Software Processes are Software Too, Revisited: an invited talk on the most influential paper of ICSE 9, 1997, Boston. In: 19th International Conference on Software Engineering, 1997, Monterey. **Proceedings…** 1997, p. 540-548.

PAULK, M. Analyzing the Conceptual Relationship Between ISO/IEC 15504 (Software Process Assessment) and the Capability Maturity Model for Software. In: Ninth International Conference on Software Quality (9ICSQ), 1999, Cambridge. **Proceedings…** 1999.

PAULK, M. C. Using the Software CMM in Small Organizations. In: The Joint 1998 Proceedings of the Pacific Northwest Software Quality Conference and the Eighth International Conference on Software Quality, 1998, Portland. **Proceedings.** p. 350-361.

PILATO, C.; COLLINS-SUSSMAN, B.; FITZPATRICK, B. **Version Control with Subversion.** 1ª ed. O'Reilly, 2004. 304 p.

ROSSI, A. C. **Representação do Componente de Software na FARCSoft:** Ferramenta de Apoio à Reutilização de Componentes de Software. 2004. 237 f. Dissertação (Mestrado em Engenharia de Software) – Escola Politécnica da Universidade de São Paulo, São Paulo, 2004.

SANCHES, R. Gerência de configuração. In: ROCHA, A.; MALDONADO, J.; WEBER, K. **Qualidade de Software: Teoria e Prática.** São Paulo: Prentice-Hall, 2001. 319 p.

SAX. Simple API for XML, 2005. Disponível em: <http://www.saxproject.org>. Acesso em: 13 junho 2005.

SOMMERVILLE, I. **Engenharia de Software.** Tradução Maurício de Andrade. 6ª ed. São Paulo: Addison-Wesley, 2003. 592 p.

STALLINGER, F.; et al. Software Process Improvement for Component-Based Software Engineering: An Introduction to the OOSPICE Project, In: 28th Euromicro Conference, 2002, Dortmund. **Proceedings…** 2002, p. 318-323.

SUN. SUN MICROSYSTEMS. Java Architecture for XML Binding, 2005. Disponível em: <http://java.sun.com/xml/jaxb>. Acesso em: 13 junho 2005.

SUN. SUN MICROSYSTEMS. Javadoc Tool, 2005. Disponível em: <http://java.sun.com/j2se/javadoc>. Acesso em: 13 junho 2005.

SZYPERSKI, C. **Component Software:** Beyond Object-Oriented Programming. Addison-Wesley / ACM Press, 1998. 411 p.

VITHARANA, P.; ZAHEDI, F. M.; JAIN, H. Knowledge-Based Repository Scheme for Storing and Retrieving Business Components: A Theoretical Design and Empirical Analysis. **IEEE Transactions on Software Engineering**, v. 29, n. 7, jul. 2003.

W3C. Documento Object Model. Disponível, 2005. Disponível em: <http://www.w3.org/DOM>. Acesso em: 13 junho 2005.

WAIBEL, A.; MAHLER, T. OJB Performance, 2004. Disponível em: http://db.apache.org/ojb/docu/guides/performance.pdf>. Acesso em: 13 junho 2005.

WALLNAU, K. Software Component Certification: 10 Useful Distinctions. Technical Note CMU/SEI-2004-TN-031. Disponível em: <http://www.sei.cmu.edu/publications/documents/04.reports/04tn031.html>. Acesso em: 5 junho 2005.

WIEGERS, K. Software Process Improvement in Web Time. **IEEE Software**, vol. 16, n. 4, p. 78-86, jul./ago. 1999.

WOODMAN, M. et al. Issues of CBD Product Quality and Process Quality. In: Fourth ICSE Workshop on Component-Based Software Engineering – Component Certification and Systems Prediction, 2001, Toronto. **Proceedings…** 2001.

YACOUB, S.; AMMAR, H.; MILI, A. Characterizing a Software Component. In: International Workshop on Component-Based Software Engineering, 1999, Los Angeles.

APÊNDICE A – O Projeto Piloto CNS (Cartão Nacional de Saúde)

Muitas das práticas de desenvolvimento de software, que motivaram a elaboração da proposta de um processo de reutilização de componentes através da organização de um repositório, vieram da experiência com a utilização de componentes de software, no projeto piloto CNS idealizado pelo Ministério da Saúde.

Em 1996, o Conselho Nacional de Saúde recomendou a criação de um identificador nacional único para o sistema de saúde do Brasil. O projeto do Cartão Nacional da Saúde iniciou-se em 1999, com o objetivo de coletar informações de diversas unidades de saúde, permitindo a construção de um repositório nacional de registros de saúde, constituindo um sistema informatizado distribuído de servidores a serviço do SUS (Sistema Único de Saúde).

As cinco principais partes do sistema são: a infra-estrutura de telecomunicação; os terminais de atendimento; os servidores e ferramentas de banco de dados; os softwares instalados nos servidores e terminais; e aspectos de segurança e um conjunto de padrões para representar, transmitir, e armazenar informações médicas.

O piloto do Cartão Nacional de Saúde foi projetado para validar diferentes pontos da arquitetura do sistema, como software, hardware, infra-estrutura e treinamento, para posterior implantação em cadeia nacional. Foram selecionadas 44 cidades em 11 estados da federação, cobrindo uma população de quase 13 milhões de habitantes e cerca de 2200 provedores de atendimento na área de saúde.

O sistema CNS pode ser fisicamente dividido em dois blocos: a Rede Permanente e a Rede Discada. A primeira contém os servidores dos três níveis mais altos da topologia da rede (Federal, Estadual e Concentrador), conforme ilustrado na figura A.1, interconectados por IPs fixos; a segunda, integra os dois últimos níveis (Municipal e Ponto de Atendimento), através de conexões discadas sob demanda. Cada instância de servidor corresponde a sua cidade, grupo de cidades, estado e, finalmente, à federação representando o Ministério da Saúde como uma entidade que gera, transforma e armazena informações do sistema nacional de saúde.

Os Pontos de Atendimento, munidos de leitores de cartões magnéticos (segundo as especificações ISO 7810 e ISO 7811), para a identificação de usuários do sistema (médicos ou pacientes) em uma determinada unidade de saúde (ex. postos de saúde, hospitais), discam para os servidores Municipais, periodicamente, para atualizarem (via HTTP) os dados armazenados em seus terminais de acesso nos bancos de dados instalados em instâncias municipais. A comunicação entre os terminais, denominados TAS (Terminal de Atendimento do SUS), e os servidores é feita de forma assíncrona. As instâncias do nível Concentrador funcionam como *backups*: quando um TAS não consegue estabelecer conexão com o nível

Municipal, é para os Concentradores que são enviados os dados cadastrados das unidades de saúde.

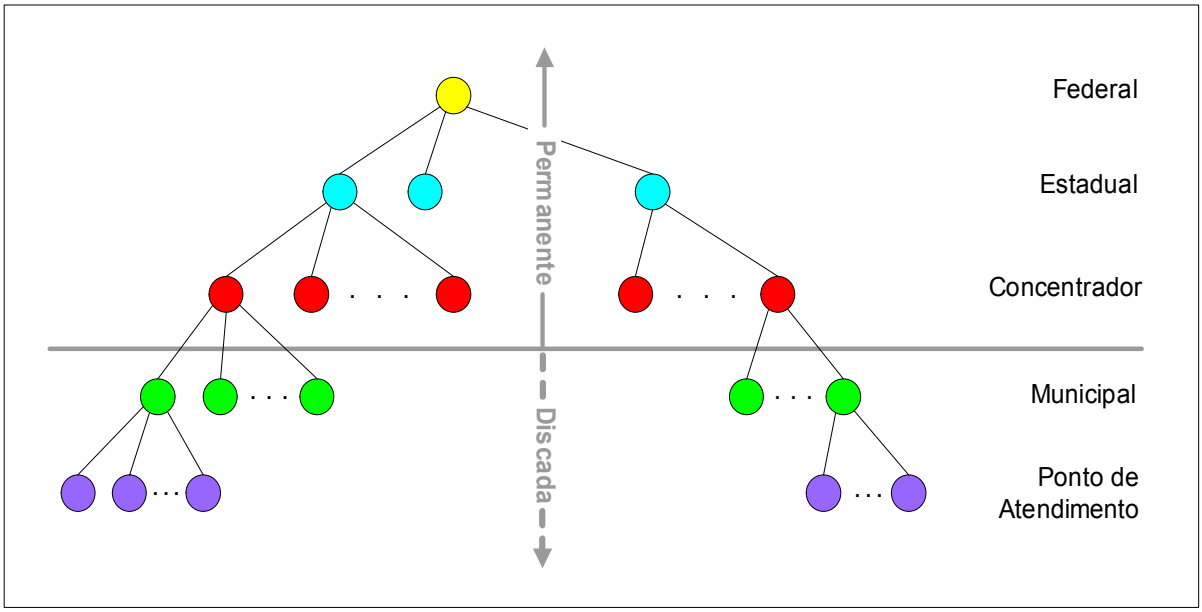


Figura A.1. Arquitetura do sistema CNS (LEMOS, 2000).

Uma vez que os dados são transferidos para o nível Municipais, a aplicação, executando nos servidores deste nível, gerencia e armazena os dados em um banco de dados local e, posteriormente, conecta-se com os Concentradores, enviando-lhes uma cópia destes dados. A partir daí, os dados coletados nos provedores de saúde do SUS alcançam a Rede Permanente, sendo compartilhados através dos níveis Federal/Estadual/Concentrador, respeitando-se a hierarquia de servidores.

A infra-estrutura de software, instalada nos servidores, foi desenvolvida em Java, constituindo o núcleo transacional do sistema CNS, com as seguintes funções: receber e manipular os dados dos TAS; propagar os dados para níveis superiores; e distribuir os dados entre as instâncias dos demais servidores para fins de consistência e sincronização. A figura A.2 representa a arquitetura de software do núcleo transacional do CNS. Os principais elementos são, então:

- *O bloco de instanciação.* Recebe transações gravadas em arquivos no formato XML, transformando-as em objetos XML.

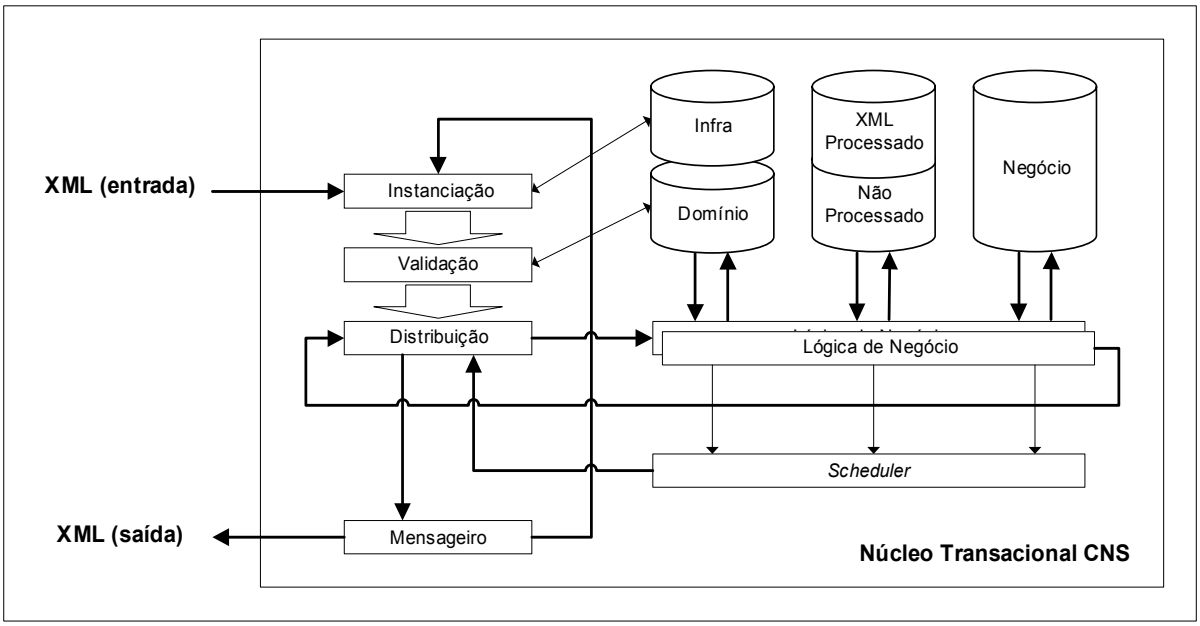


Figura A.2. Estrutura do núcleo transacional do CNS (LEMOS, 2000).

- *O bloco de validação.* Responsável pela validação das transações XML contra definições armazenadas em DTDs e regras de negócio armazenadas em banco de dados.

- *O bloco de distribuição.* Este bloco deve decidir que nó manipulará o objeto XML, processando-o localmente ou repassando-o para um outro servidor remoto.

- *O bloco de lógica de negócio.* Composto por vários componentes Java que implementam um série de tarefas relacionadas à lógica de negócio do sistema CNS.

- *O bloco mensageiro.* Uma vez decidido o processamento remoto de uma transação, pelo processo de distribuição, o mensageiro é invocado para enviar o objeto XML ao nó de destino, responsável pelo processamento definitivo da transação.

- *O bloco scheduler (planejador).* Responsável por executar qualquer tipo de tarefa agendada pelo sistema, como por exemplo o processamento de transações XML armazenadas em banco de dados.

O legado CNS, após a implantação do piloto, serviu como fonte de uma série de componentes em potencial, tanto pela reutilização de arquiteturas (como a do próprio núcleo

transacional) e especificações, quanto pelo reaproveitamento de código testado e executado em ambientes de produção. Uma série de implementações do projeto CNS vieram, na realidade, a se tornar padrões Java de mercado, hoje, como mostra o esquema abaixo:

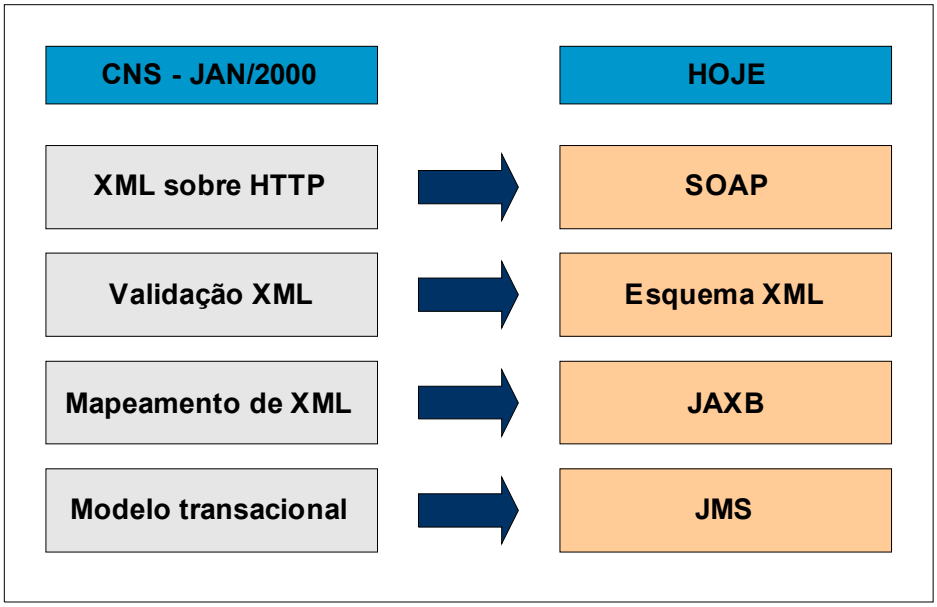


Figura A.3. Convergência de soluções implementadas no projeto CNS para padrões de mercado da tecnologia Java (LEMOS, 2000).

APÊNDICE B – Implementação do repositório RAS

Este apêndice contém a listagem dos códigos utilizados na implementação do protótipo de repositório RAS do processo *Komponento*.

```
/**
 * RASRepository.java
 */
package komponento.ras.repository.services.implementation;

import java.io.File;
```

```
import javax.xml.bind.JAXBContext;
import javax.xml.bind.Unmarshaller;

import komponento.ras.repository.jaxb.Asset;
import komponento.ras.repository.services.specification.*;

/**
 * Implementação de IRASRepository, para comunicação com um repositório RAS.
 *
 * @author Henrique Faria
 */
public class RASRepository implements IRASRepository {

    private static final String REPOSITORY_PATH = "c:/komponento/local/repository/assets";

    /*
     * Coleção de ativos do repositório
     */
    RASRepositoryResultCollection assets = new RASRepositoryResultCollection();

    /**
     * @param path
     */
    public RASRepository(String path) {
        init(path);
    }

    public RASRepository() {
        init(REPOSITORY_PATH);
    }

    private void init(String path) {
    // Inicia o repositório
        try {
        // Cria um contexto JAXB e instancia um unmarshaller
        JAXBContext jc = JAXBContext.newInstance("komponento.ras.repository.jaxb");
        Unmarshaller unmarshaller = jc.createUnmarshaller();
        unmarshaller.setValidating(true);

        File repository = new File(path);
        File[] files = repository.listFiles();

        for (int i = 0; i < files.length; i++) {
            /*
             * Cria um objeto a partir de um arquivo XML e adiciona-o ao
             * repositório.
             */
            Asset asset = (Asset) unmarshaller.unmarshal(files[i]);
                assets.add(new RASRepositoryAsset(asset, files[i].getAbsolutePath()));
            }
        } catch (Exception e ) {
            e.printStackTrace();
        }
    }
```

```
    /**
     * getAllAssets():
     * Faz uma busca no repositório e retorna todos os ativos encontrados nele.
     * Se nenhum ativo for encontrado, retorna-se uma coleção vazia.
     */
    public IRASRepositoryResultCollection getAllAssets()
        throws java.io.IOException {
        return assets;
    }

    /**
     * searchByKeyword():
     * Faz uma busca no repositório e retorna uma coleção de ativos, cujas
     * descrições coincidem com a palavra chave. Se nenhum ativo contiver a
     * palavra chave, retorna-se uma coleção vazia.
     */
    public IRASRepositoryResultCollection searchByKeyword(String theKeyword)
        throws java.io.IOException {
        RASRepositoryResultCollection result = new RASRepositoryResultCollection();

        for (int i = 0; i < assets.getCount(); i++) {
            if (((RASRepositoryAsset)assets.item(i)).
                getDescription().indexOf(theKeyword) != -1) {
                result.add(assets.item(i));
            }
        }

        return result;
    }

    /**
     * searchByLogicalPath():
     *
     * Search by Logical Path:
     * Faz uma busca no repositório e retorna uma coleção de ativos e
     * diretórios no caminho especificado. Se nenhum arquivo estiver no caminho
     * especificado, retorna-se uma coleção vazia.
     * A raíz é indicada por /
     */
    public IRASRepositoryResultCollection searchByLogicalPath(String theLogicalPath)
        throws java.io.IOException {
        RASRepositoryResultCollection result = new RASRepositoryResultCollection();

        for (int i = 0; i < assets.getCount(); i++) {
            if (((RASRepositoryAsset)assets.item(i)).
                getLogicalFolderPath().equals(theLogicalPath)) {
                result.add(assets.item(i));
            }
        }

        return result;
    }
}

-------------------------------------------------------------------------
```

```
/**
 * RASRepositoryAsset.java
 */
package komponento.ras.repository.services.implementation;

import komponento.ras.repository.jaxb.*;
import komponento.ras.repository.services.specification.*;

/**
 * Implementação de IRASRepositoryAsset, que representa um ativo em um
 * repositório RAS.
 *
 * @author Henrique Faria
 */
public class RASRepositoryAsset extends RASRepositoryResult implements IRASRepositoryAsset {

    /**
     * @param asset
     * @param url
     */
    public RASRepositoryAsset(Asset asset, String url) {
        super(asset, url);
    }

    /**
     * getVersion():
     * A versão do ativo.
     */
    public String getVersion() {
        return asset.getVersion();
    }

    /**
     *
     * getRanking():
     * A classificação (mínimo de 0 até máximo de 100) do item. É utilizada
     * para ordenar os resultados. Note que a classificações fora do intervalo
     * de 0 - 100 é atribuído o valor 0.
     */
    public int getRanking() {
        try {
            int ranking = Integer.parseInt(asset.getState());
            if (ranking < 0 || ranking > 100) {
                return 0;
            } else {
                return ranking;
            }
        } catch (NumberFormatException e) {
            return 0;
        }
    }

    /**
     * getAssetURL():
```

```
     * A URL que dá acesso ao ativo.
     */
    public String getAssetURL() {
        return url;
    }

    /**
     * getDescription():
     * Uma descrição breve do ativo.
     */
    public String getDescription() {
        return asset.getShortDescription();
    }
}

--------------------------------------------------------------------------

/**
 * RASRepositoryFolder.java
 */
package komponento.ras.repository.services.implementation;

import komponento.ras.repository.jaxb.Asset;
import komponento.ras.repository.services.specification.*;

/**
 * Implementação de IRASRepositoryFolder, que representa um caminho (lógico) em
 * um repositório RAS.
 *
 * @author Henrique Faria
 */
public class RASRepositoryFolder extends RASRepositoryResult
    implements IRASRepositoryFolder {

    /**
     * @param asset
     * @param url
     */
    public RASRepositoryFolder(Asset asset, String url) {
        super(asset, url);
    }
}

--------------------------------------------------------------------------

/**
 * RASRepositoryResult.java
 */
package komponento.ras.repository.services.implementation;

import komponento.ras.repository.jaxb.*;
import komponento.ras.repository.services.specification.*;

/**
 * Implementação de IRASRepositoryResult, que é o objeto base que representa o
```

```
 * resultado retornado de uma busca em um repositório RAS. Suas duas classes
 * derivadas são RASRepositoryAsset e RASRepositoryFolder.
 *
 * @author Henrique
 */
public class RASRepositoryResult implements IRASRepositoryResult {

    /*
     * Objeto que representa um ativo segundo o mapeamento JAXB
     */
    protected Asset asset;
    protected String url;

    /**
     * @param asset
     * @param url
     */
    public RASRepositoryResult(Asset asset, String url) {
        this.asset = asset;
        this.url = url;
    }

    /**
     * getLogicalFolderPath():
     * O caminho lógico do item retornado. É usado para organizar os ativos.
     * O diretório raíz é indicado por /
     */
    public String getLogicalFolderPath() {
        return url;
    }

    /**
     * getName():
     * O nome a ser exibido do item retornado.
     */
    public String getName() {
        return asset.getName();
    }

    public Asset getAsset() {
        return asset;
    }
}

-------------------------------------------------------------------------

/**
 * RASRepositoryResultCollection.java
 */
package komponento.ras.repository.services.implementation;

import komponento.ras.repository.services.specification.*;

import java.util.*;
```

```
/**
 * Implementação de IRASRepositoryResultCollection, interface que guarda uma
 * lista de ativos e diretórios.
 *
 * @author Henrique Faria
 */
public class RASRepositoryResultCollection implements IRASRepositoryResultCollection {

    // A coleção de ativos e diretórios
    List collection = new ArrayList();

    /**
     * item():
     * Retorna o IRASRepositoryResult do índice passado como parâmetro.
     */
    public IRASRepositoryResult item(int index) throws java.io.IOException {
        return (IRASRepositoryResult)collection.get(index);
    }

    /**
     * getCount():
     * O número de objetos IRASRepositoryResult na lista.
     */
    public int getCount() throws java.io.IOException {
        return collection.size();
    }

    /**
     * add():
     * Adiciona um item à coleção.
     */
    public boolean add(IRASRepositoryResult theResultItem) throws java.io.IOException {
        return collection.add(theResultItem);
    }

    /**
     * remove():
     * Remove um item da coleção.
     */
    public boolean remove(IRASRepositoryResult theResultItem) throws java.io.IOException {
        return collection.remove(theResultItem);
    }
}
```