UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

MARCELO HIDEKI YAMAGUTI

Uma Arquitetura Reflexiva baseada na Web para Ambiente de Suporte a Processo

Tese apresentada como requisito
parcial para obtenção do grau de
Doutor em Ciência da Computação

Prof. Dr. Roberto Tom Price
Orientador

Porto Alegre, dezembro de 2002

CIP - CATALOGAÇÃO NA PUBLICAÇÃO

Yamaguti, Marcelo Hideki

Uma Arquitetura Reflexiva baseada na Web para Ambiente de Suporte a Processo / Marcelo Hideki Yamaguti. – Porto Alegre: Programa de Pós-Graduação em Computação, 2003.

129 p.: il.

Tese (doutorado) - Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR-RS, 2003. Orientador: Price, Roberto Tom.

1. Ambiente de Desenvolvimento de *Software* 2. Ambiente de Suporte a Processo 3. Reflexão Computacional 4. Objetos Distribuídos 5. *World Wide Web*. I. Price Roberto Tom. II. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Reitora: Profª Wrana Maria Panizzi

Pró-Reitor de Ensino: Prof. José Carlos Ferraz Hennemann

Pró-Reitor Adjunto de Pós-Graduação: Prof. Jaime Evaldo Fensterseifer

Diretor do Instituto de Informática: Prof. Philippe Olivier Alexandre Navaux

Coordenador do PPGC: Prof. Carlos Alberto Heuser

Bibliotecária-chefe do Instituto de Informática: Beatriz Regina Bastos Haro

À minha esposa Marlise.

AGRADECIMENTOS

Ao meu orientador,
pela amizade e paciência demonstradas
durante a orientação deste trabalho.

A minha família,
pelo apoio nos momentos difíceis.

Aos meus amigos,
por compreenderem as ausências
necessárias.

Aos professores, funcionários e colegas de pós-graduação
que direta ou indiretamente contribuíram
para a realização deste trabalho.

A Deus por ter me iluminado e me acompanhado
durante toda esta jornada.

SUMÁRIO

LISTA DE ABREVIATURAS

ADS	Ambiente de Desenvolvimento *de Software*
API	*Application Programming Interface*
AWT	*Abstract Window Toolkit*
CASE	*Computer Aided Software Engineering*
CGI	*Common Gateway Interface*
COM	*Component Object Model*
CORBA	*Common Object Request Broker Architecture*
CSCW	*Computer-Supported Cooperative Work*
DCOM	*Distributed Component Object Model*
EJB	*Enterprise Java Beans*
GIOP	*General Inter-ORB Protocol*
HTML	*HyperText Markup Language*
HTTP	*HyperText Transfer Protocol*
I-CASE	*Integrated CASE*
IDL	*Interface Definition Language*
IIOP	*Internet Inter-ORB Protocol*
IIS	*Internet Information Services*
IOR	*Interoperable Object Reference*
IPSE	*Integrated Project-Support Environment*
JDBC	*Java DataBase Connectivity*
JDK	*Java Development Kit*
JFS	*Java File System*
JIT	*Just In-Time compiler*
JVM	*Java Virtual Machine*
MOP	*Meta-Object Protocol*

ODBC	*Open DataBase Connectivity*
OMG	*Object Management Group*
ORB	*Object Request Broker*
PML	*Process Modeling Language*
PSEE	*Process-centered Software Engineering Environment*
RMI	*Remote Method Invocation*
RPC	*Remote Procedure Call*
SEE	*Software Engineering Environment*
SGO	Sistema de Gerenciamento de Objetos
TCP/IP	*Transmission Control Protocol/Internet Protocol*
UML	*Unified Modeling Language*
WfMC	*Workflow Management Coalition*
WfMS	*Workflow Management System*
WRAPPER	*Web-based Reflective Architecture for Process suPport EnviRonment*
WWW	*World Wide Web*
XML	*eXtensible Markup Language*

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE QUADROS

RESUMO

A presente tese visa contribuir na construção de ambientes de desenvolvimento de *software* através da proposição de uma arquitetura reflexiva para ambiente de suporte a processo, nomeada WRAPPER (*Web-based Reflective Architecture for Process suPport EnviRonment*).

O objetivo desta arquitetura é prover uma infra-estrutura para um ambiente de suporte a processo de *software*, integrando tecnologias da *World Wide Web*, objetos distribuídos e reflexão computacional. A motivação principal para esta arquitetura vem da necessidade de se obter maior flexibilidade na gerência de processo de *software*. Esta flexibilidade é obtida através do uso de objetos reflexivos que permitem a um gerente de processo obter informações e também alterar o processo de *software* de forma dinâmica.

Para se obter um ambiente integrado, a arquitetura provê facilidades para a agregação de ferramentas CASE de plataformas e fabricantes diversos, mesmo disponibilizadas em locais remotos. A integração de ferramentas heterogêneas e distribuídas é obtida através do uso de tecnologias Web e de objetos distribuídos. Reflexão computacional é usada no ambiente tanto para extrair dados da execução do processo, quanto para permitir a adaptação do mesmo. Isto é feito através da introdução e controle de meta-objetos, no meta-nível da arquitetura, que podem monitorar e mesmo alterar os objetos do nível base.

Como resultado, a arquitetura provê as seguintes características: flexibilidade na gerência de processo, permitindo o controle e adaptação do processo; distribuição do ambiente na Web, permitindo a distribuição de tarefas do processo de *software* e a integração de ferramentas em locais remotos; e heterogeneidade para agregar componentes ao ambiente, permitindo o uso de ferramentas de plataformas e fornecedores diversos.

Neste contexto, o presente trabalho apresenta a estrutura da arquitetura reflexiva, bem como os mecanismos usados (e suas interações) para a modelagem e execução de processo dentro do ambiente de suporte ao processo de *software*.

Palavras-chaves: ambiente de desenvolvimento de *software*, ambiente de suporte a processo, reflexão computacional, objetos distribuídos, *World Wide Web*.

A Web-based Reflective Architecture for Process Support Environment

ABSTRACT

The present thesis aims to contribute in the construction of software development environments through the proposition of a Web-based Reflective Architecture for Process suPport EnviRonment, named WRAPPER.

The goal of this architecture is to provide an infrastructure for a process support environment, integrating World Wide Web technologies, distributed objects and computational reflection.

The main motivation for this architecture comes from the need for obtaining more flexibility in the software process management. This flexibility is obtained through the use of reflective objects that allows for a process manager to obtain information and even change the software process dynamically.

To obtain an integrated environment, the architecture provides facilities to aggregate CASE tools from different platforms and vendors, even in remote places. The integration of heterogeneous and distributed tools is obtained through the use of Web technologies and distributed objects.

Computational reflection is used in the environment for extracting data from the process enactment and for allowing its adaptation. This is done by the introduction and control of meta-objects, in the meta-level of the architecture, that can monitor and even change the objects of the base level.

As result, the architecture provides the following features: flexibility in the process management, allowing the control and the adaptation of the process; distribution of the environment in the Web, allowing the distribution of software process tasks and the tools integration in remote locations; and heterogeneity to aggregate components to the environment, allowing the use of tools from diverse platforms and suppliers.

In this context, the present work presents the structure of the reflective architecture, as well as the mechanisms (and their interactions) used for modeling and enacting the process in the environment for supporting software process.

Keywords: software development environment, process support environment, computational reflection, distributed objects, World Wide Web.

1 INTRODUÇÃO

A automação do desenvolvimento de *software* através de ferramentas CASE (*Computer Aided Software Engineering*) - Engenharia de *Software* Auxiliada por Computador visa a obtenção de *software* de alta qualidade com boa produtividade.

O termo CASE (FISHER, 1990; PRESSMAN, 2001; SOMMERVILLE, 1992), identifica usualmente uma ferramenta de *software* que provê suporte automatizado para alguma atividade no desenvolvimento de um sistema de *software*.

Um ADS (Ambiente de Desenvolvimento de *Software*) busca integrar diferentes ferramentas CASE para prover um suporte computacional mais vasto para o desenvolvimento de *software*.

Existem diversos níveis de integração de ferramentas CASE (SHARON, 1995). Um destes níveis é a integração do processo que as ferramentas darão suporte. A integração de processo é obtida quando as ferramentas do ADS dão suporte um processo de desenvolvimento de *software* comum.

Um PSEE (*Process-centered Software Engineering Environment*) - ambiente de desenvolvimento de *software* centrado em processo, é um ADS que busca prover suporte a modelagem e execução de algum processo de *software* particular.

A construção de PSEEs levanta algumas questões e problemas. Como qualquer outro ADS, um PSEE deve definir soluções para a integração de ferramentas, bem como para o seu uso de forma distribuída e remota. Além disso, para dar suporte um processo de *software*, um PSEE deve dispor de alternativas para especificar, executar e gerenciar este processo.

1.1 Motivação

Existem diversos PSEEs descritos na literatura. Cada PSEE apresenta soluções diferentes para os aspectos de modelagem, execução e gerência do processo. Algumas técnicas e paradigmas existentes que

influenciam a modelagem e execução do processo (CONRADI, 1994; GIMENES, 1994) são: regras, redes de Petri, transição de estados, objetos, entre outros.

Diversos PSEEs (NGUYEN, 1997; BALDI, 1994; BANDINELLI, 1992; BELKHATIR, 1994) utilizam o conceito de objeto como elemento básico para a modelagem e a execução do processo. Desta forma, um processo pode ser entendido como um conjunto de objetos que interagem entre si.

Uma característica importante de um processo de *software* que possui suporte por um PSEE é que o mesmo seja reflexivo (CONRADI, 1994), isto é, a partir da execução do processo, sejam extraídas informações sobre a própria execução do mesmo. O responsável pelo processo (gerente de processo), considerando sua responsabilidade na gerência de processo, utiliza estas informações como base para a adaptação do próprio processo.

Nos PSEEs existentes, que são baseados em objetos, apesar de considerarem a reflexão do processo como uma característica importante, não explicitam o uso de reflexão computacional sobre os objetos como uma solução para a extração dinâmica de dados sobre a execução do processo, nem como alternativa para a alteração da execução do mesmo, também de forma dinâmica.

Desta forma, a hipótese básica desta tese é que o uso de reflexão computacional sobre objetos de um processo em um PSEE pode ser utilizada para capturar informações sobre a execução do processo que está sendo desenvolvido, de forma a adaptá-lo dinamicamente.

1.2 Problema e questões norteadoras

Informações sobre a execução do processo são importantes para permitir o acompanhamento e melhoria do mesmo. Desta forma, deseja-se que sobre um processo já modelado e em andamento, seja possível, a qualquer momento, coletar novas informações sobre a sua execução. Além disso, um processo é dinâmico, de forma que possa ser alterado e otimizado. Assim, deve haver meios de permitir que um processo seja modificado, mesmo durante a sua execução.

O problema principal, neste contexto, a ser abordado por esta tese é **"como obter dinamicamente informações sobre a execução de um processo para dinamicamente alterar a execução deste mesmo processo?".**

De forma adicional, algumas questões que norteiam o desenvolvimento do presente trabalho são:

a. **como especificar o modelo do processo?** O ambiente deve prover facilidades para a modelagem do processo de *software* em desenvolvimento.

b. **como prover suporte a diferentes processos?** É possível que uma organização possua mais de um processo de *software*. Desta forma, o PSEE deve possuir meios de manter dois ou mais processos diferentes de forma independente.

c. **como definir artefatos, ferramentas, papéis e agentes específicos para uma execução do processo?** O PSEE deve também disponibilizar facilidades para que os elementos participantes da execução do processo possam ser criados, selecionados e alocados para o mesmo.

d. **como controlar a execução de um processo?** Durante a execução do processo deve haver meios de controlar e acompanhar cada etapa do mesmo. Por exemplo: quando da conclusão de uma tarefa definida, automaticamente ativar a próxima tarefa prevista verificando e disponibilizando os artefatos de entrada e invocando os agentes responsáveis.

Além disso, um PSEE como qualquer ADS levanta as seguintes questões:

a. **como integrar ferramentas CASE em um ambiente?** O PSEE deve prover mecanismos e padrões para que ferramentas (já existentes ou a serem construídas) sejam integradas ao ambiente, de forma que estas ferramentas possam ser disponibilizadas para o uso a qualquer usuário.

b. **como obter integração de dados, de interface com o usuário e de controle no ambiente?** O PSEE deve definir padrões que permitam que as ferramentas do ambiente possam compartilhar informações, apresentem aparência semelhante, e disponibilizem funcionalidades entre si.

c. **como gerenciar a distribuição de ferramentas?** Considerando que as ferramentas podem estar localizadas em pontos remotos, o ambiente deve prover mecanismos para controlar sua distribuição.

1.3 Soluções propostas

Para resolver os problemas expostos anteriormente, o presente trabalho propõe as seguintes soluções:

a. **Utilização de orientação a objetos:** cada PSEE representa seu respectivo processo através de algum modelo (GIMENES, 1994; CONRADI, 1994). Uma forma de modelar um processo é o uso

de objetos. Tal alternativa influenciou diversos PSEEs: E3 (BALDI, 1994), ADELE (BELKHATIR, 1994), EPOS (NGUYEN, 1997), SPADE (BANDINELLI, 1992).

Sob a perspectiva de orientação a objetos, quaisquer elementos envolvidos no processo de *software* (como tarefas e artefatos) podem ser vistos como objetos que podem ser manipulados e controlados pelo PSEE.

Estendendo-se esta perspectiva, pode-se mesmo considerar que os componentes e os usuários do PSEE (como ferramentas e agentes) são também objetos.

Desta forma, um processo de *software* pode ser modelado como um conjunto de objetos que representam o processo (por exemplo: atividades, artefatos, papéis, etc.) e seus relacionamentos. Além disso, as ferramentas integradas ao ambiente, e mesmo os usuários, também são tratadas, de forma homogênea, como objetos.

Com o uso da tecnologia de orientação a objetos, os seguintes problemas seriam solucionados:

- **como especificar o modelo do processo?** através do uso (e adaptação) de uma linguagem de modelagem de sistemas orientados a objetos, como por exemplo a UML (BOOCH, 1998), um processo poderia ser especificado através dos modelos disponíveis nesta linguagem. Alguns trabalhos relacionados já fizeram uso de linguagens de modelagem OO para a modelagem de processos (HAN, 1998; GROTH, 2002; LIMA, 2000; REIMER, 1998; WANG, 2002). A Tabela 1.1 sumariza a aplicação de diagramas UML para a modelagem de processos.
- **como prover suporte a diferentes processos?** com o uso de uma linguagem de modelagem de objetos, cada processo seria especificado por um conjunto próprio de modelos nesta linguagem. A execução de cada processo geraria os objetos correspondentes aos modelos especificados. Desta forma, no mesmo ambiente poderia haver mais do que um processo modelado e em execução simultaneamente.
- **como controlar a execução de um processo?** as diversas tarefas definidas em um processo são tratadas como objetos no ambiente. Desta forma, o controle da execução do processo corresponde ao controle de ativação dos objetos (tarefas) correspondente.

Tabela 1.1: Diagramas UML e Modelagem de processo

Diagrama UML	Modelagem de processo
Diagrama de Classes	Descrição de meta-modelo de processo com o uso de estereótipo <<meta-classe>> Modelagem dos elementos básicos (classes de objetos) do processo de software
Diagrama de Objetos	Descrição dos elementos (objetos) pertencentes a um projeto de software
Diagrama de Casos de Uso	Descrição das funcionalidades disponíveis para atores (agentes) do processo
Diagrama de Atividades	Descrição do *workflow* do processo
Diagramas de Interação	Modelagem das interações entre os elementos (objetos) de um projeto de software
Diagrama de Estados	Modelagem dos estados de um elemento (objeto) de um projeto de *software*
Diagrama de Componentes	Descrição dos componentes de software para implementação dos elementos (objetos) de um projeto de *software*.
Diagrama de Implantação	Descrição de distribuição dos componentes em plataforma de *hardware* (opcional)
Diagrama de Pacotes	Agrupamento de elementos do processo

- **como definir artefatos, ferramentas, papéis e agentes específicos para uma execução do processo?** como cada elemento existente em um processo é tratado de forma homogênea como sendo um objeto, a execução de um processo seria baseada na criação, seleção e alocação de objetos no ambiente.
- **como integrar ferramentas CASE em um ambiente?** ferramentas CASE também são tratadas como objetos. Desta forma, a integração de uma nova ferramenta CASE corresponde a criação de um objeto correspondente no ambiente, que poderá ser utilizada por outros objetos, tais como tarefas, agentes ou outras ferramentas.
- **como obter integração de dados, de interface com o usuário e de controle no ambiente?** a integração de dados seria obtida considerando-se que os dados são tratados como objetos, assim, os dados intercambiados são objetos controlados pelo ambiente. O uso de objetos também contribui no aspecto de integração de controle no ambiente, de forma que o controle entre ferramentas distintas seria mapeado pela troca de mensagens entre os objetos

correspondentes no ambiente. A implementação de interfaces com o usuário através de objetos também permitiria uma padronização do *look-and-feel* na utilização do ambiente.

- **como gerenciar a distribuição de ferramentas?** ferramentas distribuídas fisicamente seriam mapeadas por objetos distribuídos correspondentes.

b. **Uso de *workflow*:** a modelagem de um processo, além dos objetos componentes, também envolve a definição de seu modelo de *workflow* correspondente. O uso de *workflow* para a especificação e execução de processo de *software* possui alguns trabalhos desenvolvidos (ARAUJO, 1999; GEORGAKOPOULOS, 1995; OCAMPO, 2002; MANGAN, 1998; MANZONI, 2001).

O *workflow* define quais os objetos do próprio processo e do ambiente são ativados durante a sua execução. O uso de um WfMS (*Workflow Management System*) - sistema de gerenciamento de *workflow* (em especial da máquina de execução de *workflow* - *workflow engine*) - permite a execução e controle do processo através de seu modelo de *workflow* correspondente.

A utilização de *workflow* solucionaria os seguintes problemas:

- **como especificar o modelo do processo?** O *workflow* serviria como forma de especificação da seqüência de tarefas envolvidas em um processo, indicando a ordem em que os objetos correspondentes devem ser ativados.
- **como definir artefatos, ferramentas, papéis e agentes específicos para uma execução do processo?** Na especificação do *workflow* de um processo, seriam selecionados e alocados os objetos correspondentes necessários para a execução do mesmo.
- **como controlar a execução de um processo?** O uso de um WfMS (em especial do *workflow engine*) executaria um *workflow* definido e consequentemente ativaria os objetos envolvidos na execução do processo.

c. **Uso de reflexão computacional:** uma arquitetura de *software* reflexiva, baseada em objetos (CAMPO, 1997; LISBOA, 1997; OLIVA, 1998), fundamentalmente estrutura os objetos em dois níveis: o nível base e o meta-nível. O nível base é construído por objetos que implementam os aspectos funcionais do *software*, enquanto que o meta-nível possui objetos (os meta-objetos) que implementam aspectos não-funcionais ou adicionais do *software*.

Além disso, os meta-objetos podem inspecionar, bem como mudar os objetos do nível base.

Em um PSEE baseado em uma arquitetura reflexiva, os objetos do processo e do ambiente fazem parte do nível base da arquitetura (representando o aspecto funcional do PSEE). No meta-nível da arquitetura, meta-objetos podem ser instanciados e conectados a objetos do nível base. Estes meta-objetos compõem a estrutura não-funcional ou adicional do PSEE (como segurança, tolerância a falhas e aspectos de gerência).

O uso de reflexão computacional portanto solucionaria o problema principal deste estudo (como obter dinamicamente informações sobre a execução de um processo para dinamicamente alterar a execução deste mesmo processo?) da seguinte forma:

Durante a execução do *workflow* do processo, meta-objetos podem ser instanciados e inseridos no ambiente e passam a monitorar os objetos de nível base correspondente. A execução do *workflow* ativa os objetos de nível base. A troca de mensagens entre estes objetos, por sua vez, ativam os meta-objetos correspondentes. Através dos meta-objetos definidos, é possível inspecionar os objetos de nível base extraindo-se novas informações.

Meta-objetos instanciados podem mudar e estender os objetos do nível base e, consequentemente, mudar e estender o processo de *software* em execução.

d. **Utilização de tecnologias da *World Wide Web* e *middleware***: a popularização da rede de computadores Internet (em particular, da *World Wide Web*) para o acesso a informações remotas e distribuídas vem estimulando o aparecimento de diversos sistemas de informações baseados nesta rede.

   O uso da rede *Web*, como plataforma de implementação, origina-se da padronização provida através de mecanismos como o HTTP (*HyperText Transfer Protocol*), o HTML (*HyperText Markup Language*), os navegadores (*browsers Web*) e as linguagens de programação.

   De forma adicional, o uso de *middleware* (COULOURIS, 2001), como por exemplo CORBA (OMG, 2002), permite a comunicação entre sistemas (e objetos) distribuídos e heterogêneos.

   O uso destas tecnologias soluciona os seguintes problemas:

   - **como integrar ferramentas CASE em um ambiente?** O uso de objetos distribuídos, protocolos da WWW (World Wide Web) e middleware permitiria que ferramentas CASE,

disponibilizadas em locais remotos e por fabricantes diferentes, sejam integradas ao ambiente.

- **como obter integração de dados, de interface com o usuário e de controle no ambiente?** O uso de padrões de construção de interface na Web e o uso de navegadores permite que o ambiente, mesmo em plataformas heterogêneas possuam o mesmo formato de apresentação. De forma adicional, o uso de *middleware* permite que ferramentas, fornecidas por fabricantes diferentes em plataformas diversas sejam integradas ao ambiente.
- **como gerenciar a distribuição de ferramentas?** O uso de protocolos e padrões disponíveis na WWW, bem como de middleware, permite que ferramentas localizadas em pontos fisicamente distantes, possam ser controladas no ambiente.

Observa-se que para a resolução dos problemas expostos é necessário um conjunto de soluções interconectadas. Entretanto, o uso de reflexão computacional é o fundamento para a solução do problema principal.

1.4 Trabalhos relacionados

Considerando-se que o presente trabalho utiliza o conhecimento de diversas áreas, abaixo são apresentadas resumidamente os trabalhos semelhantes nas suas áreas específicas:

1.4.1 Ambientes de desenvolvimento de *software* centrados em processo

Existem diversos PSEEs desenvolvidos descritos na literatura. E3 (BALDI, 1994), SPADE (BANDINELLI, 1992), EPOS (NGUYEN, 1997), ADELE (BELKHATIR, 1994), TABA (OLIVEIRA, 2000), ExPSEE (GIMENES, 2000) são alguns exemplos.

Cada PSEE apresenta soluções diferentes para os aspectos de modelagem, execução e controle do processo. Apesar de considerarem a reflexão do processo (e do próprio ambiente) como uma característica importante para a evolução do processo, não explicitam o uso de reflexão computacional sobre objetos no próprio ambiente como alternativa para isto.

1.4.2 Arquiteturas reflexivas

O uso de reflexão computacional para o desenvolvimento de arquiteturas de sistemas também gerou vários trabalhos. Pode-se citar como exemplos: Guaraná (OLIVA, 1998) (arquitetura reflexiva para desenvolvimento

de *software* reflexivo em Java), Luthier (CAMPO, 1996; CAMPO, 1997) (um *framework* reflexivo que permite a introspecção de exemplos em Smalltalk), MOTF (LISBOA, 1995) (arquitetura reflexiva para desenvolvimento de aplicações tolerantes a falhas) e em (WU, 1998) Arquitetura reflexiva de transações baseada em componente. Estas arquiteturas são apresentadas, resumidamente, a seguir.

a. ***Framework* reflexivo Luthier**

O *framework* Luthier (CAMPO, 1996) é um ambiente que foi projetado para prover suporte a construção de ferramentas visuais para a análise dinâmica de programas na linguagem Smalltalk. Ele integra técnicas de reflexão computacional (baseado em meta-objetos) com hipertexto e técnicas de interface gráfica com o usuário.

A Figura 1.1 apresenta a estrutura básica da arquitetura. No nível mais baixo (nível base) situam-se os objetos que definem um *software* aplicativo ou um *framework* de aplicação.

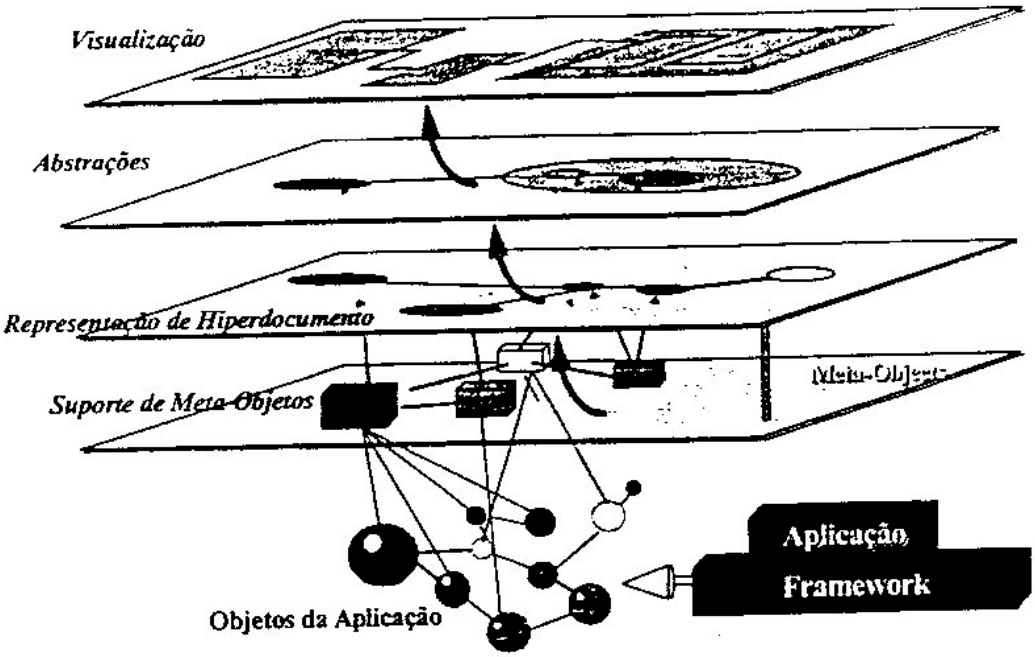

Figura 1.1: Arquitetura global de Luthier (CAMPO, 1996)

No nível imediatamente superior (meta-nível) estão implementadas as funcionalidades para suporte à reflexão computacional, na qual meta-objetos são relacionados ao nível base da aplicação. Este relacionamento pode ser para uma ou mais classes, ou para um ou mais objetos, ou para um ou mais métodos (mesmo de classes diferentes) do nível base. Esta flexibilidade de relacionamentos aumenta a capacidade do controlar a execução do nível base.

Os níveis superiores (representação do hiperdocumento, abstrações e visualização) são responsáveis pela preparação e apresentação gráfica dos dados coletados no meta-nível.

Luthier não é propriamente um ADS, classificando-se melhor como uma ferramenta (reflexiva) que permite o acompanhamento e visualização da execução de sistemas – uma característica importante para a depuração de sistemas.

b. Arquitetura reflexiva Guaraná

Guaraná (OLIVA, 1998), é uma arquitetura de *software* reflexiva independente de linguagem projetada para prover alto grau de reusabilidade de código de meta-nível, bem como obter de forma segura, flexibilidade e reconfigurabilidade do comportamento de meta-nível de objetos.

Uma implementação desta arquitetura foi realizada através de modificação da máquina virtual Java, provendo diversos mecanismos que permitem a introdução de reflexão em aplicação desenvolvidas nesta linguagem. Em (OLIVA, 1998) também é descrita uma biblioteca de componentes de meta-nível disponibilizados para a construção de aplicações distribuídas sobre o Guaraná.

Deve-se considerar que Guaraná não é realmente um ADS, porém oferece mecanismos e facilidades para a construção de aplicações distribuídas em rede, com características reflexivas.

c. **Arquitetura reflexiva para desenvolvimento de *software* tolerante a falhas**

Em (LISBOA, 1996) é descrita uma arquitetura reflexiva orientada a objetos para a construção de *software* tolerante a falhas. Esta arquitetura permite introduzir, de forma transparente e não-intrusiva, mecanismos de tolerância a falhas em uma aplicação orientada a objetos.

A arquitetura possui três níveis (Figura 1.2), o nível D_0 representa o nível base (ou da aplicação), que contém os objetos da aplicação. O nível D_1 representa o meta-nível onde ficam situados os meta-objetos responsáveis pela gerência das falhas dos objetos do nível base.

O nível D_2 é um nível acima do meta-nível (um meta-meta-nível) que provê mecanismos utilizados no nível D_1, tais como, controle de concorrência, distribuição e recuperação de estado.

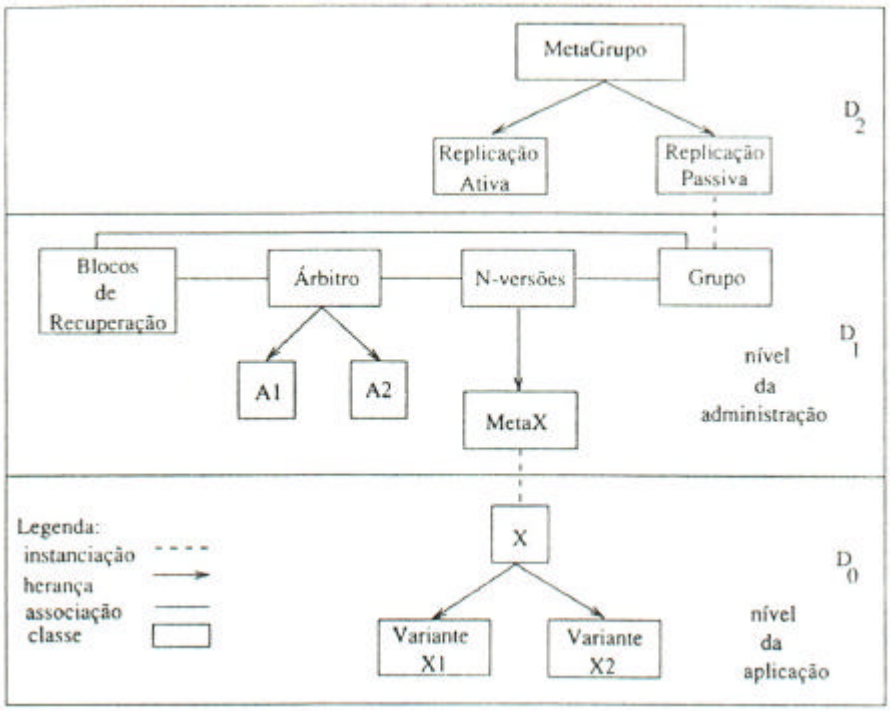

Figura 1.2: Arquitetura reflexiva proposta (LISBOA, 1996)

Nesta arquitetura, além de classes relativas à implementação de reflexão computacional, também são providas diversas classes relativas à implementação e gerência de mecanismos de tolerância a falhas.

Deve-se notar que a arquitetura proposta provê a estrutura para a criação de *software* com reflexão computacional, porém não contempla aspectos de suporte a gerência de projetos, ou processo de *software*.

d. **Arquitetura reflexiva de transações baseada em componentes**

Em (WU, 1998) é descrita uma arquitetura baseada em componentes, que provê facilidades para a construção de aplicações baseadas em transações usando componentes (Java *Beans*).

Esta arquitetura é composta por diversos componentes (Figura 1.3). Os principais são o *server component* que implementa a lógica de uma aplicação; o *client component* que implementa a interface com o usuário e a requisição de serviços; e o *server component container* que encapsula um *server component* provendo capacidades de persistência e de transação ao mesmo.

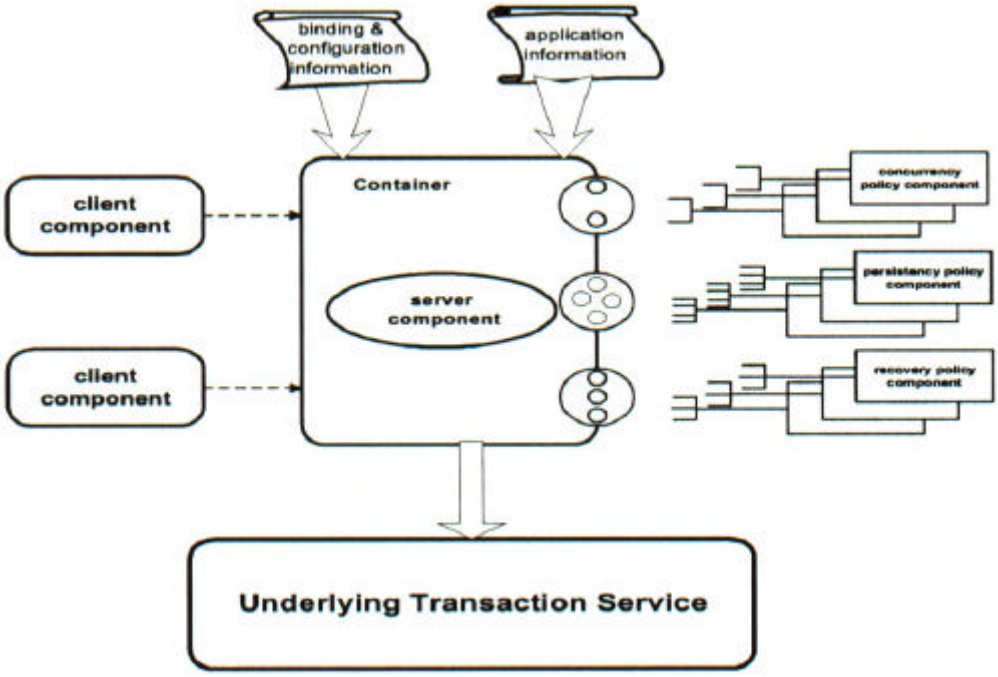

Figura 1.3: Arquitetura reflexiva de transações (WU, 1998)

Um *container* (apresentado na Figura 1.4) também provê facilidades de controle de concorrência e segurança. Um *container* intercepta as requisições de um componente cliente para um servidor através de objetos reflexivos (meta-objetos).

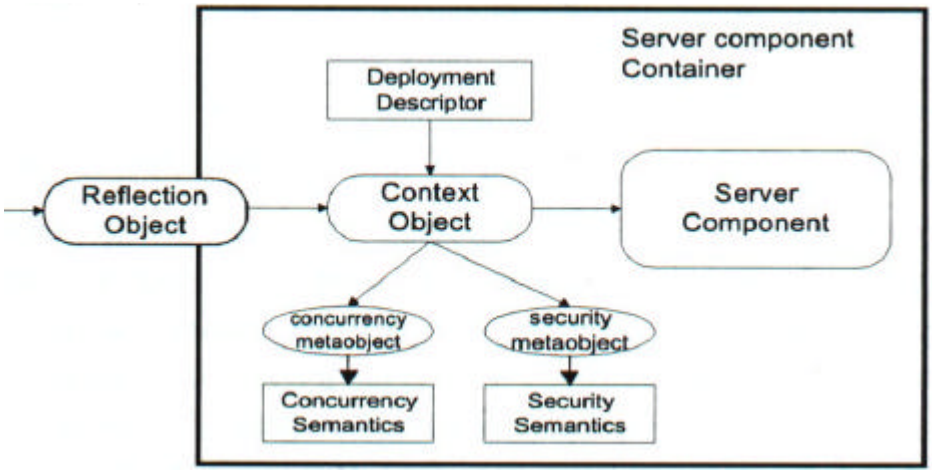

Figura 1.4: Estrutura de um *container* (WU, 1998)

A arquitetura é implementada com um dialeto Java, chamado de *Reflexive Java*, que permite a interceptação de mensagens entre objetos, e redirecionamento para meta-objetos. O protocolo de comunicação com o meta-nível é realizado através de um pré-processador da linguagem durante a compilação da aplicação.

A arquitetura proposta possui o mérito de prover mecanismos de desenvolvimento de aplicações baseados em componentes e suporte a aplicações baseadas em transações (uma característica importante em sistemas baseados na Web). Porém, não aborda mecanismos de controle de trabalho em equipe (cooperação/colaboração em uma equipe de projeto de *software*), nem aspectos relativos à definição e gerência de processo de *software*.

Observou-se que, diferentemente da arquitetura WRAPPER, as arquiteturas apresentadas não têm como objetivo o suporte a processo de *software*.

1.4.3 CORBA reflexivo

O uso de reflexão sobre objetos em CORBA já foi sugerido em alguns trabalhos (WEGDAN, 2000; DUCHIEN, 1999; COSTA, 2000a; SINGHAI, 1997). Contudo, nestes trabalhos a reflexão é utilizada como mecanismo para depuração de aplicações, gerência de objetos distribuídos ou gerência de meta-informação.

O uso de reflexão no *middleware* CORBA, entretanto, não é explorado como mecanismo de controle e adaptação de processo de *software* nestes trabalhos.

1.5 Objetivos e contribuições da tese

A partir da definição do problema, das questões norteadoras e das soluções propostas, determina-se como o objetivo principal do presente trabalho: **"definir uma arquitetura para um ambiente de desenvolvimento de *software* centrado em processo, baseado em objetos distribuídos e reflexivos, que permita a extração dinâmica de informações sobre o processo em desenvolvimento, bem como a alteração dinâmica da execução deste processo"**.

De forma adicional, tendo definido o objetivo principal, propõe-se como objetivos secundários a serem alcançados:

- **Definir alternativas para obter a distribuição do ambiente de suporte a processo**: com o uso de objetos distribuídos, tecnologias WWW e *middleware* definir como o ambiente pode ser utilizado de forma distribuída. A WWW é utilizada como plataforma de desenvolvimento e implantação distribuída da arquitetura proposta;
- **Definir alternativas para obter heterogeneidade na execução e alteração do ambiente de suporte a processo**: com o uso de objetos, tecnologias WWW e *middleware* definir como o ambiente

pode ser executado em plataformas heterogêneas e como integrar ferramentas heterogêneas ao ambiente. A WWW também é neste aspecto como plataforma de desenvolvimento e implantação que agrega diversas tecnologias heterogêneas para a arquitetura proposta.

A principal contribuição da tese é a especificação da arquitetura reflexiva para ambiente de suporte a processo, nomeada WRAPPER (*Web-based Reflective Architecture for Process suPport EnviRonment*).

O ambiente de suporte a processo definido por esta arquitetura traz como contribuições:

a. **extração dinâmica de informações sobre a execução de um processo**: a vantagem trazida pela arquitetura é que mesmo após um processo já modelado ter iniciado sua execução, novas informações sobre o mesmo podem ser extraídas a qualquer momento. Exemplos:
   - estatísticas de uso de ferramenta CASE: instancia-se um meta-objeto que monitora a ferramenta CASE desejada, de forma que a cada ativação, o meta-objeto armazene informações estatísticas (por exemplo: número de ativações, data/hora da ativação);
   - controle de um artefato: um meta-objeto correspondente a um artefato pode monitorar todas as alterações solicitadas de um determinado artefato;
   - estatísticas de um agente: um meta-objeto pode inspecionar um agente correspondente, extraindo informações como: quando o agente está logado no ambiente, quais ferramentas está ativando, etc.

b. **alteração dinâmica da execução do processo**: outra vantagem trazida pela arquitetura é que em um processo já em andamento é possível realizar modificações. Por exemplo:
   - adição de funcionalidade a uma tarefa: a ativação de uma tarefa (que possui um meta-objeto associado) é interceptada e além da funcionalidade já definida na tarefa, o meta-objeto pode adicionar uma nova funcionalidade como ativar uma nova ferramenta para gerar artefato adicional;
   - alteração do *workflow*: um meta-objeto associado a uma tarefa em um *workflow*, pode interceptar solicitação de ativação da tarefa e transferir esta ativação para outra tarefa do *workflow*;
   - balanço de carga: um meta-objeto associado a um agente que já esteja ocupado (realizando uma tarefa) pode interceptar uma chamada a este agente e redirecioná-la a um outro agente disponível no ambiente.

Como benefícios secundários o ambiente baseado na arquitetura permite:

a. **acesso e utilização do ambiente em locais remotos**: o ambiente pode ser utilizado em locais remotos através da Web;
b. **acesso e execução do ambiente em diversas plataformas**: o ambiente pode ser acessado e executado em plataformas heterogêneas;
c. **integração de componentes diversos (ferramentas de plataformas e fabricantes diversos) ao ambiente**: novas ferramentas podem ser integradas, pelos agentes, independentemente de plataforma ou de fabricante.

1.6 Estrutura do trabalho

O presente trabalho está organizado em cinco capítulos descritos a seguir.

No capítulo 2 são apresentados os conceitos básicos envolvidos no contexto em que a tese se insere, sendo também definidos os termos utilizados neste trabalho.

O capítulo 3 descreve a arquitetura reflexiva proposta nesta tese. As abordagens utilizadas para a modelagem de processo e a estrutura da arquitetura são detalhadas no desenvolvimento do capítulo. Também é apresentado com a arquitetura satisfaz os requisitos de um ambiente centrado em processo e como a arquitetura pode ser implementada com as tecnologias disponíveis.

O protótipo que implementa a arquitetura WRAPPER é apresentado no capítulo 4. Neste capítulo são apresentadas as alternativas tecnológicas utilizadas para a implementação. Exemplos de execução e os resultados obtidos também são discutidos no capítulo.

O capítulo 5 apresenta as considerações finais sobre o trabalho desenvolvido e trabalhos futuros a serem realizados.

2 CONCEITOS BÁSICOS

Este capítulo apresenta os conceitos básicos dentro do contexto do trabalho. Também especifica a terminologia utilizada neste trabalho, uma vez que algumas áreas envolvidas possuem diversas definições e termos que variam de autor para autor.

2.1 Integração de ferramentas CASE

Uma ferramenta CASE isolada tem a sua utilidade e custo relativo assegurados pela produtividade que traz na determinada tarefa que apóia. Entretanto, o poder de uma ferramenta CASE é aumentado de modo considerável se for ligada a outras ferramentas em um ambiente integrado, compartilhando informações.

Esta integração pode ser notada pelo usuário através da uniformidade na utilização das ferramentas disponíveis. Isto pode variar de uma fraca integração, onde as ferramentas compartilham de forma simples as informações comuns, a uma integração mais forte, onde as ferramentas compartilham informações de forma bem integrada, apresentando uma interface consistente e homogênea com o usuário.

As definições apresentadas por diversos autores (SHARON, 1995; THOMAS, 1992; BELL, 1995) para a integração de ferramentas CASE, podem ser resumidas em quatro categorias básicas:

a. **Integração de dados**: deve assegurar que toda a informação no ambiente seja gerenciada de forma completa e consistente, não importando como as partes dos dados são manipuladas e transformadas pelas ferramentas, oferecendo facilidades para que os dados de uma ferramenta possam ser utilizadas por outra;

b. **Integração de apresentação**: as ferramentas possuem interface com o usuário com aparência e comportamento (*look-and-feel*) semelhantes. Desta forma reduzem o esforço do usuário em interagir com as diversas ferramentas;

c. **Integração de controle**: permite a combinação flexível das funções de um ambiente de acordo com as preferências do projeto, processos básicos e suportes oferecidos pelo ambiente. Desta forma, as funções oferecidas por uma ferramenta devem estar disponíveis para serem utilizadas por outras ferramentas do ambiente;

d. **Integração de processo**: assegura-se de que as ferramentas interagem entre si no suporte a um processo comum de desenvolvimento de *software*. Ou seja, cada ferramenta incorpora um conjunto de suposições sobre o processo na qual ela pode ser utilizada. Assim, se duas ferramentas possuem suposições consistentes sobre o processo em que estão envolvidas, elas são consideradas bem integradas.

A Figura 2.1 a seguir apresenta estas categorias básicas de integração de ferramentas.

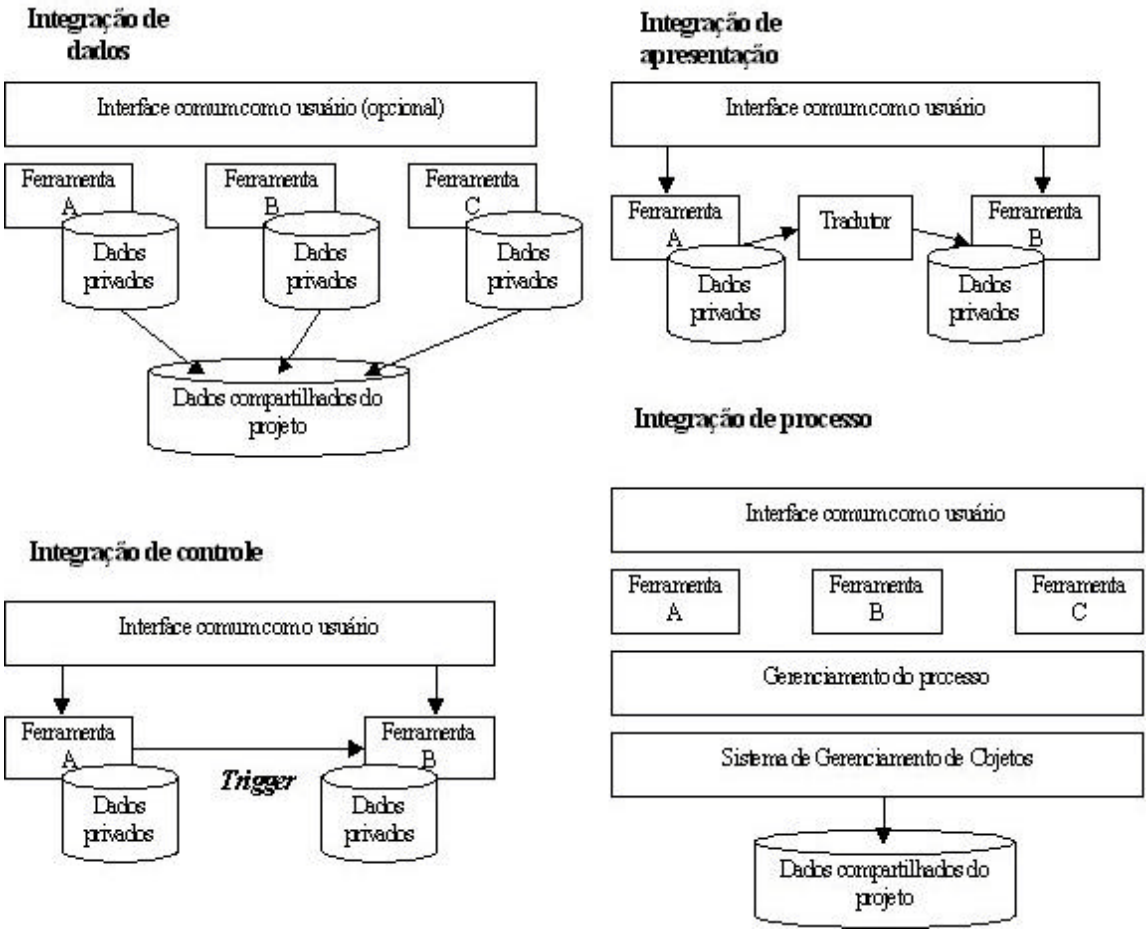

Figura 2.1: Níveis de integração de ferramentas CASE

Considerando-se as características das diversas formas de integração de ferramentas existentes, a estrutura básica de integração é apresentada na Figura 2.2.

Nesta estrutura, pode-se observar que as ferramentas atuam sobre um sistema operacional que deve prover um sistema básico de gerenciamento de arquivos e capacidades de gerência de processos. Entre as ferramentas deve haver um Sistema de Gerenciamento de Objetos (SGO). O SGO é responsável pelo mapeamento de entidades lógicas (como programas, projetos, etc.) em estruturas de armazenamento. Um SGO pode ser implementado de forma simples (controlando apenas os arquivos que armazenam as informações, por exemplo) ou de forma mais complexa (mapeando informações para classes de objetos e seus relacionamento, por exemplo).

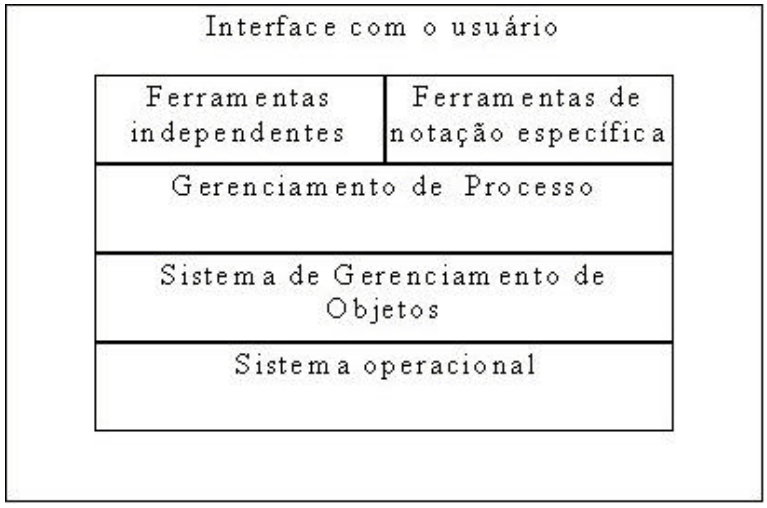

Figura 2.2: Estrutura básica de integração

O Gerenciamento de Processo é o responsável pelo controle das ferramentas integradas e interface com o Sistema de Gerenciamento de Objetos. Basicamente, as ferramentas que interagem com o Gerenciamento de Processo podem ser classificadas em duas classes distintas:

a. **Ferramentas independentes**: estas ferramentas não dependem de uma notação específica ou linguagem de programação. Usualmente fazem uso de mecanismos de intercâmbio limitado de dados;

b. **Ferramentas de notação específica**: estas ferramentas baseiam-se em uma notação específica, tais como linguagens de programação, notações de projeto, etc. Esta notação serve de base de integração às ferramentas.

Todas as ferramentas, o SGO e o sistema operacional possuem alguma forma de interface com o usuário. Caso a interface das ferramentas

também seja integrada, o aprendizado de novas ferramentas torna-se facilitado para o usuário.

Ambientes CASE integrados também podem ser divididos em abertos ou fechados. Ambientes CASE fechados definem mecanismos de integração que são compartilhados apenas pelas ferramentas oferecidas pelo ambiente. Dificilmente ferramentas externas podem ser integradas, impedindo que o usuário desenvolva suas próprias ferramentas caso a representação dos dados não seja tornada pública pelo fornecedor. Em contraposição, ambientes CASE abertos possuem a representação de integração pública, permitindo que novas ferramentas sejam desenvolvidas e integradas.

A seguir são apresentados os vários mecanismos de integração propostos por diversos autores (SOMMERVILLE, 1992; PRESSMAN, 2001; THOMAS, 1992; FUGGETTA, 1993).

2.1.1 Integração de dados

A integração de dados é baseada no conhecimento do formato da estrutura de dados entre as ferramentas a serem integradas. Este tipo de integração pode variar ainda em diversos níveis:

a. **Arquivos de caracteres**: as ferramentas compartilham o mesmo formato de arquivo. O formato mais comum é o arquivo de linhas de caracteres;

b. **Notações orientadas a linguagem**: as ferramentas são integradas em torno de um modelo de dados que representa as informações sintáticas e semânticas de uma linguagem de programação;

c. **Sistemas de gerenciamento de objetos**: a integração baseia-se em um sistema de gerenciamento de objetos (SGO) que deve descrever um modelo de dados público compartilhado por todas as ferramentas.

2.1.2 Integração de apresentação

A integração da interface do usuário significa que as ferramentas compartilham o mesmo estilo e padrões de interação (*look-and-feel*) com o usuário. Uma vez que as ferramentas possuem a mesma aparência, o usuário terá menor custo no aprendizado de uma nova ferramenta introduzida no ambiente.

Esta integração possui três subtipos:

a. Interface de sistema de janelas;

b. Interface de comandos;

c. Interface de interação.

Normalmente, ambientes integrados pela interface podem fazer uso de mais de um tipo de interface, sendo um destes tipos é o dominante ou o principal. Considerando a facilidade de integração de novas ferramentas, os ambientes fechados, que dificultam a integração de ferramentas externas, atingem um grau de integração de interface razoável. Em contraposição, ambientes abertos, graças à diversidade de funcionalidades e representações de informação, podem perder a característica de integração de interface a cada nova ferramenta integrada, uma vez que novas diretrizes podem ser adicionadas para suprir uma funcionalidade não identificada anteriormente.

Neste tipo de integração a ativação de diferentes ferramentas é realizada explicitamente pelo usuário e os dados podem possuir tradutores para as diferentes representações de informações manipuladas por cada ferramenta. A seguir são descritas as formas de integração por interface.

2.1.3 Integração de controle

Thomas e Nejmeh (THOMAS, 1992) definem a **integração de controle** como a capacidade das ferramentas compartilharem funcionalidade. De modo ideal, todas as funções oferecidas pelas ferramentas de um ambiente integrado devem estar disponíveis para todas as outras ferramentas. Sendo que as ferramentas (servidoras) que provêem funcionalidades não necessitam saber quais ferramentas (clientes) fazem uso destas funcionalidades.

Para que as ferramentas compartilhem funcionalidades elas devem ser capazes de indicar quais operações devem ser executadas, podendo ser ativadas através de gatilhos (*triggers*). Como normalmente uma operação necessita dados (na forma de parâmetros ou de dados compartilhados), considera-se que a integração de controle complementa a integração de dados.

A fim de obter esta integração, as ferramentas devem possuir duas propriedades básicas: disponibilidade e uso. A disponibilidade ocorre quando uma ferramenta dispõe seus serviços às outras ferramentas. Duas ferramentas são integradas quanto a disponibilidade, se uma ferramenta utiliza facilmente as funcionalidades da outra. O uso refere-se a como uma ferramenta utiliza as funcionalidades de outra ferramenta. Isto implica que as ferramentas devem ser construídas de forma modular, permitindo que o uso de uma funcionalidade seja independente da ferramenta que a implementa.

2.1.4 Integração de processo

A **integração de processo** diz respeito a como as ferramentas integradas provêem suporte um determinado processo de *software* (FINKELSTEIN, 1994; GIMENES, 1994). Isto é obtido com base em três dimensões: passo do processo, evento do processo e restrição do processo (uma limitação de algum aspecto do processo).

O passo do processo define uma unidade de trabalho que gera um resultado. Ferramentas são ditas bem integradas quanto ao passo do

processo, se os objetivos que elas alcançam são parte de uma decomposição do passo do processo, e se ao atingir estes objetivos, permite-se que outras ferramentas também atinjam seus próprios objetivos.

Evento de processo diz respeito a uma condição que ocorre durante o passo do processo e que pode implicar na execução de uma atividade. Uma ferramenta é bem integrada quanto ao evento de processo se as pré-condições de execução de uma ferramenta são satisfeitas pelas outras ferramentas envolvidas no passo do processo, e se o resultado de sua execução satisfaz a pré-condição das ferramentas seguintes.

Restrição do processo é uma limitação de algum aspecto do processo. Considera-se que ferramentas são bem integradas pelo aspecto de restrição de processo se elas possuem suposições similares sobre o conjunto de restrições que reconhecem e respeitam.

Neste tipo de integração, o ambiente deve possuir conhecimento embutido das atividades às quais dá suporte, suas fases e restrições. Desta forma é capaz de ativar e controlar a seqüência de execução das atividades.

O modelo do processo de desenvolvimento de *software* é conhecido pelo ambiente e o utiliza para dirigir as atividades deste processo. A criação de um modelo de processo envolve alguns passos como: identificação das atividades envolvidas no processo, identificação dos resultados das atividades, definição de como as atividades são coordenadas e suas dependências, alocação de profissionais para cada atividade, e especificação das ferramentas necessárias para a execução de cada atividade.

Algumas atividades são executadas em paralelo e devem ser mapeadas no modelo do processo. Como as atividades e suas coordenações são independentes, o modelo de processo deve ser dinâmico e mudar à medida em que mais informações a respeito das atividades são obtidas.

O modelo de processo é a base para o planejamento das atividades de um *software* e usualmente é feito e acompanhado de modo informal pelo gerente de um projeto. Porém para o acompanhamento automatizado do processo de desenvolvimento de *software*, este modelo deve ser formalizado.

Existem algumas dificuldades na integração de processo, uma delas é que apesar da existência de modelos de ciclos de vida de *software* que servem de base para a definição de processos (tais como cascata, prototipação, espiral, etc.) (PRESSMAN, 2001) estes modelos são muito genéricos, não detalhando as atividades e suas implementações, e dependem de uma interpretação particular para sua instanciação.

Outra dificuldade é que não existe um único modo de se desenvolver um *software*. Usualmente, as pessoas envolvidas em um desenvolvimento criam alternativas e mudam de atividade de acordo com o momento. A inclusão desta flexibilidade no modelo não é trivial.

Outro ponto a considerar, diz respeito ao fato de que os modelos podem prever os produtos que devem resultar, bem como as comunicações entre os desenvolvedores. Entretanto, a forma de coordenação e as decisões a serem tomadas são difíceis de se modelar, correndo-se o risco de que ao se

formalizar demais o processo de desenvolvimento, a equipe crie resistência para sua utilização.

2.2 Categorias de Ambientes de Desenvolvimento de *Software*

Um Ambiente de Desenvolvimento de *Software* (ADS) pode ser definido como "uma coleção de ferramentas de *software* e de *hardware*, combinados para prover suporte a produção de sistemas de *software* em um domínio particular de aplicação" (SOMMERVILLE, 1992; KRATZ, 1998). Esta seção tem como objetivo a apresentação de classificações existentes para ADS.

Ainda segundo Sommerville (SOMMERVILLE, 1992), existem diversos tipos de ADS em uso (Figura 2.3) que podem ser classificados em três grandes grupos:

a. **Ambientes de programação**: estes ambientes basicamente possuem recursos para o suporte à codificação, teste e depuração. O suporte para atividades de definição de requisitos, especificação e projeto são limitados;

b. ***Workbenches* CASE**: um *workbench* provê principalmente suporte à especificação e projeto de *software*, diferentemente de ambientes de programação. Alguns *workbenches* dispõem de suporte superficial de programação e usualmente são projetados para computadores pessoais e podem ser utilizados conjuntamente com ambientes de programação;

c. **Ambientes de Engenharia de *Software***: este tipo de ADS visa o suporte a grandes sistemas de *software* de longa duração, para os quais o custo de manutenção excede o de desenvolvimento, e que normalmente são desenvolvidos em equipe, ao invés de programadores individuais. Devem prover suporte a todas as atividades de desenvolvimento e manutenção.

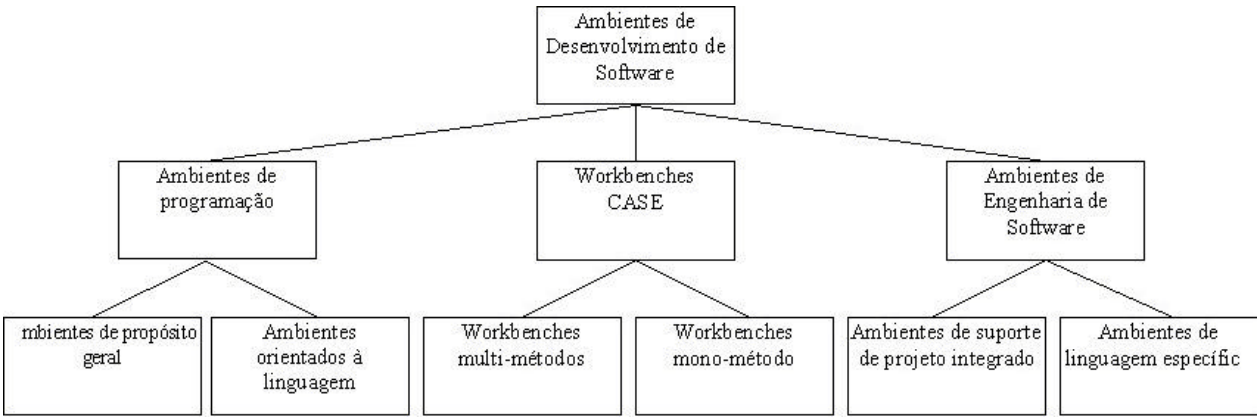

Figura 2.3: Classificação de ADS (SOMMERVILLE, 1992)

Cada categoria possui subdivisões. Obviamente o limite entre cada categoria não é absoluto, pois ambientes de programação podem incorporar facilidades de especificação e projeto de *software*, e mesmo *workbenches* podem ser adaptados a plataformas de maior porte e incorporar facilidades de suporte à programação, tornando-os mais próximos a categoria de um Ambiente de Engenharia de *Software*.

2.2.1 Ambientes de programação

Historicamente a atividade de programação (e logo após as atividades de teste e depuração) foram as primeiras a serem realizadas no processo de desenvolvimento de *software*. Desta forma, também os ambientes que oferecem ferramentas de suporte à codificação, teste e depuração foram os primeiros ADS a serem desenvolvidos. A maturidade alcançada por estas ferramentas produz ambientes mais robustos e confiáveis.

Os ambientes de programação podem ser divididos em dois subgrupos:

a. **Ambientes de propósito geral**: ambientes que provêem ferramentas de uso geral para a codificação em diversas linguagens de programação. Uma especialização deste subgrupo é representada pelos sistemas de desenvolvimento de microprocessador, que normalmente possuem ferramentas de *hardware*, como emuladores de circuitos;

b. **Ambientes orientados à linguagem**: ambientes projetados para oferecem suporte ao desenvolvimento de programas em uma determinada linguagem de programação.

2.2.2 *Workbenches* CASE

Um *workbench* CASE busca dar suporte ao desenvolvimento das fases de análise e projeto de *software*. Usualmente estes ambientes provêem suporte a técnicas diagramáticas presentes em diversas metodologias de desenvolvimento de *software*.

Os componentes básicos de um *workbench* CASE são:

a. **Repositório central de informação**: mantém as informações dos projetos de *software* já realizados ou em andamento;

b. **Sistema de edição de diagramas**: responsável pela criação e manipulação de diagramas de metodologias de desenvolvimento de *software*. Um editor diagramático não se resume apenas um editor gráfico, pois manipula as informações contidas no diagrama, armazenando-as no repositório central;

c. **Facilidades de checagem do projeto**: visa a detecção da consistência da modelagem do *software*. Devem estar integradas ao sistema de edição, permitindo que o modelador seja informado dos erros que comete durante a criação de seus diagramas;

d. **Facilidade de consulta**: provê mecanismos (linguagem de consulta) para que o desenvolvedor possa navegar sobre as informações relativas aos projetos desenvolvidos;

e. **Dicionário de dados**: mantém armazenadas, de forma ordenada, as informações relativas a um determinado projeto de *software*;

f. **Facilidades de geração de relatórios**: permite que informações sejam retiradas do repositório central e geram documentação do sistema;

g. **Ferramentas de geração de formulários**: permite que formatos de telas e relatórios sejam especificados;

h. **Facilidades de importação e exportação**: permitem que dados do repositório central sejam intercambiados com outras ferramentas de desenvolvimento;

i. **Suporte à geração de código**: visa a geração automática de código, ou partes de código, a partir das informações do repositório central.

2.2.3 Ambientes de Engenharia de *Software*

Um Ambiente de Engenharia de *Software* (*Software Engineering Environments* - SEE), também chamado de I-CASE (*Integrated* CASE) (PRESSMAN, 2001), ou de IPSE (*Integrated Project-Support Environment*) (SHARON, 1995) pode ser definido como (SOMMERVILLE, 1992): “uma coleção de *hardware* e ferramentas de *software* que podem atuar de modo integrado. O ambiente deve prover suporte a todos os processos de *software* desde a especificação inicial até os testes e liberação do sistema. De forma adicional, o ambiente deve dar suporte o gerenciamento de configuração de todos os produtos do processo de *software*”.

Nesta definição são levantados os seguintes aspectos:

- As facilidades do ambiente são integradas. Tais ambientes usualmente provêem suporte de integração por dados, por interface, por controle ou mesmo por processo;
- Todos os produtos podem estar sujeitos a gerenciamento de configuração. Os ambientes devem prover mecanismos de gerência de configuração durante a existência dos sistemas, para todas as suas versões, documentos como: especificações, projeto, código fonte, documentação do usuário, entre outros, de forma completa e consistente;

- As facilidades devem dar suporte a todas as atividades do desenvolvimento de *software*. Portanto as ferramentas disponíveis devem dispor de suporte à especificação, projeto, documentação, programação, teste, depuração.

Existem basicamente dois tipos de SEE: os ambientes orientados a linguagens, que são baseados no suporte a uma única linguagem de programação e os ambientes de suporte ao projeto integrado, que provêem suporte a diversas linguagens de programação com diversos métodos para o desenvolvimento de sistemas.

As vantagens possibilitadas pelo uso de SEE no suporte aos processos de desenvolvimento de *software* são:

- Todas as ferramentas possuem interface com um Sistema de Gerenciamento de Objetos (SGO). Isto permite que o resultado de uma ferramenta seja a entrada de outra, permitindo assim que a ativação de ferramentas seja feita de forma aleatória e não seqüencial;
- A utilização de um SGO permite que objetos de pequena granularidade sejam identificados, armazenados e sujeitos ao controle de configuração. Desta forma a estrutura do sistema de armazenamento reflete a estrutura do sistema de *software*;
- Permite a criação de ferramentas CASE mais poderosas, pois elas podem fazer uso dos relacionamentos armazenados no SGO;
- Os documentos produzidos, desde os estudos de viabilidade até os resultados de testes, podem ser gerenciados por ferramentas de gerenciamento de configuração. As facilidades do SGO permitem o relacionamento entre documentos, desta forma projetos podem ser ligados aos seus respectivos códigos e mudanças podem ser rastreadas automaticamente;
- Caso o ambiente possua uma integração homogênea, as ferramentas também compartilharão de uma interface consistente, permitindo aos usuários aprender novas ferramentas;

Gerenciamento de projeto pode acessar as informações do projeto e as ferramentas de gerenciamento podem usar dados reais coletados durante o andamento do projeto.

Uma vantagem adicional seria o fato de que a introdução e uso de SEE teria impacto significativo sobre o custo relativo durante todo o ciclo de vida de desenvolvimento de um *software*.

Entretanto, usualmente a implantação de um SEE é mais dispendiosa e difícil do que ambientes de programação e *workbenches* CASE, pois estes dois normalmente são mais aceitos pela menor complexidade de utilização.

2.2.4 Outras classificações

Em adição às classificações explicitadas anteriormente, ainda se apresentam as seguintes classificações:

Dart (DART, 1988) sugere uma taxonomia de ADS que envolve quatro categorias:

a. **Ambientes centrados em linguagens**: são aqueles construídos ao redor de uma linguagem de programação, provendo um conjunto de ferramentas adequado para esta linguagem. Via de regra, são altamente interativos e fornecem suporte limitado a projetos de grande porte;

b. **Ambientes orientados à estrutura**: provêem facilidades para manipulação da estrutura (gramática) de um programa diretamente, entretanto, de forma independente da linguagem;

c. **Ambientes *toolkit***: constituem-se em conjuntos de ferramentas de suporte independentes de linguagem, incluindo facilidades de gerenciamento de configuração e controle de versão, podendo possuir algum controle e gerenciamento sobre a utilização destas ferramentas;

d. **Ambientes baseados em métodos**: incorporam ferramentas relacionadas a alguma metodologia de desenvolvimento de *software* específica, como por exemplo, desenvolvimento estruturado ou orientado a objetos.

Em seu artigo, Fuggeta (FUGGETTA, 1993) buscou classificar os diferentes tipos de tecnologias CASE de modo geral. Basicamente, esta classificação proposta envolve três grandes classes:

a. **Ferramentas**: visam o apoio a tarefas específicas no desenvolvimento de *software*;

b. ***Workbenches***: provêem suporte a apenas algumas atividades do processo de produção de *software*;

c. **Ambientes**: dão suporte a todo, ou grande parte, do processo de desenvolvimento de *software*.

Por esta classificação, um ambiente CASE é um conjunto de diversas ferramentas e *workbenches* que dão suporte ao processo de *software*, e pode ser classificado em cinco categorias:

a. ***Toolkits***: constituem-se em conjuntos de produtos fracamente acoplados e facilmente extensível pela agregação de novas ferramentas e *workbenches*. Diferentemente de um *workbench,* um *toolkit* provê facilidades de suporte a diversas atividades no processo de produção de *software*, normalmente constituindo-se

em uma evolução de ferramentas de *software* básico providos por sistemas operacionais, tais como o UNIX. Como o acoplamento é fraco, a ativação de ferramentas deve ser realizada de forma explícita ou através de mecanismos de controle de ativação simples, tais como *pipes*. Também por este fato, *toolkits* podem ser mais facilmente estendidos, porém não é imposto nenhuma restrição quanto ao processo de desenvolvimento que os usuários utilizam;

b. **Ambientes centrados em linguagens**: a característica básica deste tipo de ambiente é o fato de ser implementado na mesma linguagem de programação que dá suporte, permitindo desta forma que seus usuários possam estendê-lo e mesmo utilizar partes do mesmo no desenvolvimento de aplicações. Tais ambientes, normalmente, possuem boa integração de apresentação e de controle, pois apresentam uma interface consistente com o usuário que pode ativar de forma padronizada as várias ferramentas providas. Entretanto, possuem fraca integração de dados e de processo, pois baseiam-se em estruturas internas que são invisíveis ou complexas para o usuário compreender e manipular;

c. **Ambientes integrados**: estes ambientes possuem mecanismos padrões que permitem que usuários possam integrar ferramentas. Possuem integração de apresentação, provendo interfaces com o usuário uniformes, consistentes e coerentes. A integração de dados é provida por mecanismos de manipulação de um repositório de dados que armazena toda informação produzida e manipulada pelo ambiente. Mecanismos de ativação dentro do ambiente provêem a integração de controle. Entretanto, ambientes integrados não visam diretamente a integração de processo, que os distingue dos ambientes centrados no processo;

d. **Ambientes de quarta-geração**: estes ambientes podem ser considerados como os precursores dos ambientes integrados, e mesmo considerados como um subconjunto deste. Entretanto as aplicações desenvolvidas nestes ambientes possuem características especiais como: operações são relativamente simples, enquanto que a estrutura de informação é complexa; a interface com o usuário é baseada em formulários para entrada, apresentação e modificação de dados; os requisitos de *software* não são claros e devem ser detalhados através de protótipos; o processo de produção usualmente é evolutivo. Estas características os tornam mais específicos que os ambientes integrados e mais ricos do que os ambientes centrados em linguagens, pois provêem diversas ferramentas de desenvolvimento além dos processadores de linguagens – usualmente chamadas de *linguagens de quarta geração*;

e. **Ambientes centrados no processo**: estes ambientes são baseados na definição formal do processo de *software*. Esta definição serve de base para que mecanismos no ambiente guiem a execução das atividades a serem desenvolvidas, ativem automaticamente as ferramentas, e atendam políticas específicas. Estes ambientes provêem integração de apresentação, de dados e de controle; entretanto, visam primordialmente o atendimento do processo de *software* definido. Desta forma, os ambientes são compostos essencialmente por partes que realizam duas funções: a primeira envolve a criação dos modelos de processos e a segunda, a execução do processo definido.

Como pôde-se observar nesta seção, a classificação de ADS por categorias é vasta e difere na visão de diferentes autores. Assim, cabe salientar novamente que o objetivo deste trabalho está voltado a definição de uma arquitetura que suporte os quatro níveis de integração (dados, apresentação, controle e processo), visando a integração baseada em processo de *software*.

2.3 Ambiente de desenvolvimento de *software* centrado em processo

Um ambiente que provê apoio automatizado para um processo de *software* é denominado de ambiente de desenvolvimento de *software* centrado em processo - PSEE (*Process-centered Software Engineering Environment*). Um dos objetivos específicos de um PSEE é fornecer apoio à especificação do modelo de processo.

Na estrutura de um PSEE existe um componente, o gerente de processo (*process driver*) ou motor de processo (*process engine*), que utiliza o modelo de processo definido para guiar as tarefas durante a sua execução: execução ordenada das tarefas definidas, ativação automática de ferramentas do ambiente, orientação dos agentes envolvidos nas tarefas, etc.

Existem diversos PSEEs que foram desenvolvidos para prover apoio ao processo de *software* (BANDINELLI, 1992; BEN-SHAUL, 1994; FINKELSTEIN, 1994; GIMENES, 2000; NGUYEN, 1997; OLIVEIRA, 2000). Cada PSEE possui arquitetura e provê soluções diversas para o apoio ao processo de *software*.

O presente trabalho propõe que o PSEE possua uma arquitetura reflexiva. Desta forma, nas seções a seguir são exploradas as características de um PSEE e de reflexão computacional para a posterior especificação da arquitetura proposta.

2.3.1 Desenvolvimento de *software* centrado em processo

Existem diversas possibilidades para o desenvolvimento de *software* visando uma melhor qualidade e produtividade, tais como o uso de protótipos, especificação formal, entre outras. Uma destas possibilidades é que o ambiente de desenvolvimento seja centrado (integrado) em **processo** (PRESSMAN, 2001; GIMENES, 1994; HUMPHREY, 1995).

A preocupação com o processo vem da percepção de que para obter um produto de boa qualidade (no caso, *software*) deve haver preocupação com cada detalhe da sua produção (coleta de dados, elicitação, análise, projeto, codificação, testes, manutenção, documentação, etc.).

A arquitetura proposta neste trabalho visa definir a estrutura de um ambiente centrado em processo de *software*. Neste contexto, são apresentados os principais conceitos envolvidos na modelagem e gerência do processo de *software*, para melhor compreensão dos seus componentes básicos e funcionalidades necessárias.

Diversos autores (CONRADI, 1994; GIMENES, 1994; HUMPHREY, 1990; SNOWDON, 1994; ROSSI, 1998; NGUYEN, 1997) e organizações, tais como o *Software Engineering Institute* (PAULK, 1993), *Workflow Management Coalition* (WORKFLOW MANAGEMENT COALITION, 2002a; WORKFLOW MANAGEMENT COALITION, 2002b), *International Organization for Standardization* (SPICE, 2002) realizaram estudos para o estabelecimento dos conceitos básicos relativos ao processo de *software*.

A seguir é apresentado um resumo dos conceitos básicos utilizados neste trabalho:

a. **Processo**: é o conjunto de todos os elementos envolvidos na produção e manutenção de um produto de *software*. É composto por um processo de produção, um meta-processo, e um suporte de processo;

b. **Processo de produção**: parte do processo responsável pelo desenvolvimento e manutenção de um produto de *software* a ser liberado;

c. **Meta-processo**: parte do processo encarregado de manter e melhorar o processo completo, isto é, o processo de produção, seus meta-processos e o suporte de processo;

d. **Suporte do processo**: parte do processo e a tecnologia envolvida para definir, modificar, analisar e executar o mesmo. A tecnologia envolve métodos e linguagens, bem como ferramentas de modelagem de processo e interpretadores do modelo, e o ambiente operacional destas ferramentas;

e. **Suporte de produção**: conjunto de métodos, formalismos e ferramentas associadas para dar suporte aos agentes de um processo no trabalho sobre artefatos;

f. **Modelo do processo**: descrição do processo expresso utilizando-se uma linguagem de modelagem de processo. Um modelo é uma abstração da realidade que representa, sendo usualmente uma descrição parcial desta realidade havendo,

portanto, partes ou aspectos do processo que não são capturados no modelo. Existem dois submodelos principais: o modelo de meta-processo e o modelo do processo de produção;

g. **Linguagem de modelagem de processo**: notação formal usada para expressar modelos de processo (meta-processo e processo de produção). Uma característica desejável em uma linguagem de modelagem de processo é que a mesma seja reflexiva, de forma que possa expressar em um único modelo tanto as atividades do meta-processo, quanto do processo em si, evitando-se, desta forma, formalismos diversos para a modelagem do processo. Trabalhos envolvendo o uso de orientação a objetos (HAN, 1998; REIMER, 1998), *workflow* (WORKFLOW MANAGEMENT COALITION, 2002a; MANGAN, 1998), e outras técnicas (ROSSI, 1998), buscam apresentar soluções para a modelagem de processos.

No prosseguimento deste trabalho são descritos os elementos envolvidos em um modelo de processo de *software*. As definições a seguir são baseadas em diversos trabalhos (CONRADI, 1994; LIMA, 1995; GIMENES, 1994; MANGAN, 1998), e em particular naqueles que buscam definir um modelo de processo baseado em orientação a objetos (WORKFLOW MANAGEMENT COALITION, 2002; GROTH, 2002; BIDER, 1998; WANG, 2002)**:**

a. **Agente**: pessoa que executa as atividades relacionadas a um papel. Um agente é caracterizado pelas propriedades do papel e sua disponibilidade. Um agente pode ser especializado em um agente humano (um indivíduo da organização disponível para desenvolver projetos) ou em um agente de *software* (que automatiza uma determinada tarefa que poderia ser realizada por um ser humano);

b. **Artefato**: é um (sub)produto de um processo. Um artefato produzido por uma tarefa do processo pode servir de base para outra tarefa na criação de um novo artefato. Um artefato pode ser especializado em diversas formas, como por exemplo: código fonte de um módulo de programa, documentos gerados durante a análise, etc.;

c. **Atividade**: representa uma etapa no processo de desenvolvimento de um *software*. As atividades podem ser ordenadas em redes organizadas por encadeamento (horizontal) ou hierarquia (vertical);

d. **Documento**: é um tipo especial de artefato que possui informações (textuais, gráficas, multimídia) que podem ser alteradas e trocadas entre os agentes do processo;

e. **Ferramenta**: é o suporte manual ou computadorizado do trabalho envolvido em uma tarefa. Uma ferramenta pode ser utilizada para o desenvolvimento de um produto de *software*,

bem como para a gerência de projetos, e mesmo a gerência e modelagem do processo de *software*;

f. **Papel**: representa uma função ou cargo de trabalho ocupado por um agente no desenvolvimento de um artefato de *software*, que possui responsabilidades, direitos e habilidades associadas. Um papel pode gerenciar outros papéis;

g. **Produto de *software***: é o resultado da composição de diversos artefatos que visa a solução de um determinado problema de uma organização;

h. **Projeto**: instância de um processo para o desenvolvimento de um produto específico em uma determinada organização, com objetivos e restrições específicos;

i. **Recurso**: é um elemento necessário à execução de uma tarefa, podendo ser uma ferramenta ou um agente;

j. **Tarefa**: é uma divisão de uma atividade que produz alteração de estado no produto pela geração ou alteração de artefatos. As tarefas estão associadas a papéis, ferramentas e artefatos.

Todos os elementos do processo podem ser compostos, bem como possuírem diversas versões. Desta forma, um controle de configuração responsável pelo controle de consistência e modificações, propagação de alterações é outro elemento importante.

Observa-se também que o conceito de processo de *software* envolve três dimensões que se complementam:

a. **Modelagem do processo**: envolve a definição e alteração dos modelos de processo disponíveis na organização, que servirão de base para a criação de projetos de desenvolvimento de *software*. Esta dimensão está relacionada ao conceito de meta-processo, responsável, portanto, pelo controle do modelo de processo que está descrito utilizando alguma linguagem de modelagem de processo;

b. **Execução do processo**: envolve a instanciação e execução de um determinado processo, gerando um projeto de desenvolvimento de *software*. Esta dimensão está relacionada ao processo de produção de *software* que visa a criação de produtos de *software*;

c. **Gerência do processo**: envolve o controle da execução do processo, isto é, o controle dos vários projetos em andamento (execução do processo), bem como a adaptação do processo.

Desta forma, deseja-se que um ambiente que suporte processo de *software*, contemple estas três dimensões simultaneamente.

2.3.2 Componentes básicos de um PSEE

Realizando um resumo da definição de diversos autores (PRESSMAN, 2001; SHARON, 1995; SOMMERVILLE, 1992; WASSERMAN, 1981), pode-se identificar os seguintes componentes básicos de um PSEE (Figura 2.4):

Figura 2.4: Componentes básicos de um PSEE

a. **Gerenciamento de interface com o usuário**

Este componente deve prover um conjunto de ferramentas para a criação de interfaces com o usuário, baseado em um protocolo de apresentação comum. As ferramentas de interface contêm uma biblioteca de elementos básicos de construção de interface com o usuário. A interface gerada deve ser consistente e se comunicar com cada ferramenta CASE individual.

O protocolo de apresentação define padrões de interface que dará às ferramentas CASE a mesma aparência e impressão, diminuindo o esforço do usuário na utilização do ambiente. Este protocolo inclui diretrizes para o *layout* de telas, nomes e organizações de menus, ícones, uso dos periféricos (como teclado e *mouse*) e o mecanismo de acesso às ferramentas.

b. **Ferramentas CASE**

Este componente envolve as diversas categorias de ferramentas CASE já descritas no capítulo 2. Deve-se observar, entretanto, que as

ferramentas CASE descritas englobam tanto a categoria de ferramentas de apoio às diversas tarefas relacionadas ao desenvolvimento de um sistema de *software*, como a categoria de ferramentas de apoio à atividade de gerenciamento do processo de desenvolvimento. Portanto, neste componente de ferramentas, considera-se a primeira categoria, uma vez que os outros componentes do ADS possuirão suas ferramentas de apoio correspondentes. As ferramentas CASE representam a interface do PSEE com o usuário.

c. **Gerenciamento de ferramentas**

Este componente é responsável pelo controle do comportamento das ferramentas CASE do ambiente. Durante a utilização das ferramentas, este componente é responsável pela sincronização e comunicação das mesmas, controlando o fluxo de informações com o sistema de gerenciamento de objetos. O gerenciamento inclui também operações como controle de segurança e auditoria, permitindo a extração de métricas sobre a utilização das ferramentas.

O gerenciamento de ferramentas deve prover, de forma adicional, facilidades tais como:

- **Registro**: para cada ferramenta integrada, informações pertinentes à interoperabilidade devem ser capturadas e registradas;
- **Encapsulamento**: pode ser necessário o estabelecimento de um *wrapper* (envoltório) sobre cada ferramenta integrada para realizar a apresentação, de uma maneira padronizada, das características da ferramenta;
- **Interoperabilidade**: o gerenciamento deve fornecer meios (por troca de mensagens ou compartilhamento de dados, por exemplo) para que as ferramentas integradas possam se comunicar e invocar mutuamente;
- **Interação**: cada ferramenta deverá possuir um conjunto bem definido de tarefas que poderão apoiar no desenvolvimento de sistemas.

Obviamente, a implementação do gerenciamento de ferramentas deverá levar em consideração os aspectos de integração de ferramentas descritos no capítulo 2.

d. **Sistema de gerenciamento de objetos**

O sistema de gerenciamento de objetos (SGO) constitui-se em um banco de dados com funções de controle de acesso que possibilitam aos outros componentes interagir com este banco de dados. Este banco de dados armazena todos os dados de projeto, com seus relacionamentos e dependências, permitindo compartilhamento de informação e coordenação de projetos. Dependendo do autor, este banco de dados é chamado de “dicionário de dados”, “enciclopédia de dados” ou “repositório”. Neste trabalho, será

adotado o termo “enciclopédia de dados”, pois o mesmo tem a função de reunir dados de diversos tipos e de diversas ferramentas CASE.

A coordenação de trabalho dos membros de um projeto e a definição de que informação será compartilhada entre os mesmos é função da metodologia ou processo utilizado, sendo função da enciclopédia de dados armazenar, controlar e compartilhar adequadamente a documentação, requisitos de *software*, projetos, código-fonte, e outros dados resultantes.

A enciclopédia de dados (NOTARI, 1999) permite que os membros de um projeto pensem em termos de objetos, componentes, subsistemas, instruções de comando, e outras construções. Informações como modelos de dados, atributos, relacionamentos e suas localizações devem ser definidos antes que os projetos de desenvolvimento, manutenção ou reengenharia iniciem.

O sistema de gerenciamento de objetos possui papel importante em projetos de reengenharia. Todas as informações existentes do sistema (que provavelmente estejam sob controle de uma equipe de manutenção) devem ser sincronizadas com os requisitos e especificações do novo sistema (que deve estar sob controle de uma equipe de reengenharia). Esta coordenação, com as informações compartilhadas entre a equipe de manutenção e de reengenharia, deve ser definida e implementada antes que o novo processo de desenvolvimento se inicie.

e. **Gerenciamento de processo**

Este componente deve estabelecer o *workflow*, tarefas, responsabilidades, atribuições de recursos, políticas e regras e liberações de produtos que implementem o processo que define cada projeto. O uso apropriado dos métodos e ferramentas é dirigido pelo processo que está sendo aplicado em um determinado problema (AMARAL, 1997; FINKELSTEIN, 1994).

Modelos de *workflow* podem variar de acordo com os tipos de problemas e métodos para resolvê-los. Mesmo metodologias de desenvolvimento de *software* também podem ser embutidas nestes modelos, resultando diversos modelos correspondentes. O importante é que cada membro de uma equipe de desenvolvimento saiba qual processo está sendo utilizado.

O gerenciamento de processo sobre o ciclo de vida de desenvolvimento de *software* deve definir que ferramentas serão usadas e quando utilizá-las. De forma adicional, também deve permitir controlar os projetos instanciados, a partir do processo modelado.

f. **Gerenciamento de projeto**

O gerenciamento de projeto é responsável pela definição da estrutura detalhada do trabalho realizado, cronogramas e custos de cada projeto, medindo os resultados atuais de um projeto com os resultados planejados. Os w*orkflows* definidos no gerenciamento de processo são ligados

ao gerenciamento de projeto para a atribuição de recursos, cronogramas, custos e para reunir as métricas de desenvolvimento de *software*.

Quando os recursos de um projeto são deslocados para outro, os efeitos desta mudança devem ser imediatamente conhecidos. Desta forma é necessário que o gerenciamento de projeto esteja conectado ao gerenciamento de processo.

Outra função importante deste componente é o controle de estatísticas e métricas de desempenho de projetos, possibilitando que o gerente de um projeto possa monitorar a produtividade e a qualidade atingidas.

g. **Gerenciamento de requisitos**

O gerenciamento de requisitos deve tratar dos requisitos técnicos para o sistema em construção. Ele deve obter todos os requisitos e informações de como estes estão sendo satisfeitos, relacionando cada requisito com os elementos do sistema que o satisfaz.

Assim, a qualquer momento durante a execução de um projeto, relatórios de situação atual podem ser preparados e emitidos pelo acesso às informações contidas no repositório.

h. **Gerenciamento de configuração**

Processos de desenvolvimento de *software* geralmente mudam, adaptando-se a novas necessidades e tendências. O componente de gerenciamento de configuração permite o controle de todas as atividades de desenvolvimento, manutenção e reengenharia de sistemas, através do gerenciamento e controle de configurações e versões dos modelos de processo, projetos, produtos liberados, sistemas de *software* e recursos.

O gerenciamento de configuração permite a aplicação de pedidos de mudança, notificações, análises de impacto, cronogramas, históricos e rastreamento de auditorias de forma consistente por todos os projetos e sistemas. Os pedidos de mudança são registrados e análises de impacto de mudança são realizadas. Novos planos de projeto são criados, notificações de mudança são enviadas para todos os membros da equipe afetados e um novo esquema de auditoria é estabelecido.

Mudanças no desenvolvimento de *software* são usuais. Desta forma, organizações que não possuem um sistema de controle de configurações formal correm o risco de perder o controle sobre seus sistemas.

i. **Gerenciamento de documentação**

Documentação é um produto resultante do desenvolvimento, manutenção ou reengenharia de sistemas. O gerenciamento de documentação deve prover armazenamento, recuperação, controle e gerenciamento de todos os documentos necessários para a condução de um projeto, bem como dos documentos criados e modificados por um projeto.

O gerenciamento de documentação trabalha de forma dependente do gerenciamento de processo, de projeto e de configuração, bem como da forma através da qual as equipes de projeto prevêem compartilhar suas informações.

j. **Verificação e validação de projeto**

'Verificar' significa assegurar que o *software* em construção satisfaz os requisitos definidos, enquanto que 'validar' significa assegurar que este *software* provê funções que o usuário realmente deseja.

A verificação e validação de projeto depende de outros componentes, e sua importância se dá pela comprovação de que a organização tem a documentação completa e procedimentos rigorosos para assegurar a qualidade do sistema liberado.

Isto tem conseqüências sobre outros aspectos do desenvolvimento de *software*, tais como o gerenciamento de processo e configuração, pois o suporte a verificação e validação dos sistemas em desenvolvimento garante a melhoria constante do processo.

2.3.3 Definição de funcionalidades de um PSEE

Considerando-se alguns papéis executados pelos diversos agentes no desenvolvimento de *software* (CONRADI, 1994; HUMPHREY, 1990; PORTELLA, 1998; LIMA, 1995; MANZONI, 2000) foram definidas as seguintes funcionalidades a serem providas por um PSEE.

a. **Gerente de processo**: este agente é o responsável pela criação e manutenção de processos de *software* de uma organização, pela modelagem dos processos de *software*, bem como pela gerência dos processos em execução na organização Entre as funcionalidades que devem estar disponíveis a este agente estão:

a.1. Modelagem de meta-processo de *software*: na tarefa de modelagem do meta-processo o PSEE deve permitir:

- Indicar os artefatos a serem gerados por cada tarefa de um processo;
- Definir os recursos necessários à execução de uma tarefa;
- Indicar um responsável (papel) pela execução de uma tarefa;
- Estabelecer o uso de produtos prontos em determinadas tarefas;

a.2. Programação de meta-processo de *software*: na tarefa de programação do meta-processo o PSEE deve permitir:

- Definir o fluxo de trabalho, indicando as etapas (atividades) do processo, bem como a decomposição de cada atividade em tarefas. De forma adicional, deve permitir o encadeamento de tarefas e etapas;

a.3. Propagação de alterações de um processo em projetos de *software* em andamento: devem existir mecanismos que permitam que ao se alterar a definição do processo, os projetos de *software* em andamento já instanciados sejam, ou atualizados conforme o novo processo, ou terminem sua execução conforme o processo original.

a.4. Instanciação de um projeto de *software* de um determinado processo de *software*: a partir de um processo modelado, um ou mais projetos podem ser instanciados. Nesta instanciação, o ADS deve prover as seguintes funcionalidades:

- Definir o gerente do projeto;
- Definir os recursos iniciais (profissionais, equipamentos, *software*);
- Definir o cronograma geral de projetos, prevendo prazos máximos e mínimos.

a.5. Acompanhamento de projetos em andamento: durante a execução dos projetos é necessário que haja as seguintes funcionalidades:

- Verificar o estado atual (real) do andamento de um projeto em relação ao estado projetado;
- Permitir o deslocamento de recursos entre projetos;
- Alterar um projeto pela criação, alteração ou remoção de atividades/tarefas previstas no projeto. Isto poderia implicar na atualização do processo a partir dos projetos instanciados.

a.6. Controle de recursos: os recursos da organização disponíveis para os projetos devem ser administrados. Desta forma o PSEE deve permitir cadastrar e manter os recursos de *software*, *hardware* e profissionais.

a.7. Comunicação com gerente(s) de projeto: deve existir mecanismos que permitam que o gerente de processo possa se comunicar com os diversos gerentes de projetos em andamento.

a.8. Planos de contingência: em caso de problemas durante a execução de um processo, deve ser possível definir soluções alternativas para a sua resolução.

a.9. Mudança de ambiente de suporte a projetos: caso haja uma alteração no ambiente de suporte a projetos, como por exemplo, mudança do protocolo de comunicação de rede ou de uma ferramenta CASE, deve existir algum mecanismo que possibilite que os projetos em andamento sejam adaptados ao novo ambiente, ou que o ambiente antigo seja mantido até a finalização dos projetos já existentes.

b. Gerente de projeto: é o responsável pelo planejamento e acompanhamento de um determinado projeto de *software*.

b.1. Planejamento de projeto: durante a fase de preparação do projeto de *software* devem estar disponíveis as seguintes funcionalidades:

- Detalhar tarefas previstas, através de definição de cronograma, com indicação, por exemplo, do prazo de início e fim de cada tarefa. Também devem ser definidos e alocados recursos necessários a cada tarefa;
- Projetar o orçamento, pela estimativa de custos associados a realização das tarefas;
- Especificar os requisitos de mudança, isto é, definir soluções alternativas no caso de um recurso não estar disponível, por exemplo.

b.2. Acompanhamento do projeto: durante o andamento do projeto devem estar disponíveis funcionalidades como:

- Verificar o estado atual (real) com o projetado;
- Alterar o cronograma previsto;
- Alterar os recursos para uma tarefa;
- Alterar o projeto pela criação, alteração, remoção de atividades/tarefas. Esta funcionalidade deve estar conectada ao gerente de processo;
- Comunicar-se com engenheiros de *software*: devem existir mecanismos de comunicação/coordenação dos membros de um projeto;
- Verificar o consumo de recursos alocados para o projeto
- Permitir a mudança de pessoal entre atividades do projeto;
- Verificar a estrutura do produto em determinado período de execução do projeto, permitindo analisar sua qualidade;

- Capturar os requisitos e informações de conclusão do desenvolvimento do projeto;
- Facilitar a organização do projeto: permitindo o desenvolvimento / manutenção / reengenharia de produtos.

b.3. Comunicação com gerente de processo: deve ser disponibilizado algum mecanismo para comunicação com o gerente de processo.

b.4 Planos de contingência: em caso de problemas durante a execução do projeto, deve ser possível definir soluções alternativas para a sua resolução.

c. Engenheiro de *software* (analista, programador, testador, documentador, etc.): é todo agente que realiza uma determinada tarefa relacionada ao desenvolvimento de um artefato de *software*. Para este agente o ADS deve prover as seguintes funcionalidades:

c.1. Controlar o *login/logout* de tarefa incumbida: deve ser possível vistoriar o início e conclusão de uma tarefa prevista;

c.2. Ativar uma ferramenta CASE: o PSEE deve disponibilizar ferramentas CASE ao Engenheiro de *Software* de forma que o mesmo possa realizar suas tarefas. O controle de ativação também é controlado pelo PSEE permitindo gerar estatísticas de uso das ferramentas, por exemplo;

c.3. Comunicar-se com outro engenheiro: dentro de um projeto deve ser possível realizar a comunicação entre os membros de sua equipe;

c.4. Comunicar-se com o gerente de projeto: através de algum mecanismo de comunicação um Engenheiro de *Software* pode se reportar ao seu gerente de projeto e vice-versa;

c.5. Verificação do cumprimento de prazos: o PSSE deve prover mecanismos para a verificação de prazos previstos para tarefas do projeto;

c.6. Controle de versões: devem ser disponibilizar mecanismos de controle de versões de um artefato em desenvolvimento;

c.7. Controle de uso: o PSEE deve prover mecanismos para o controle de acesso ao ambiente de desenvolvimento, o uso de recursos disponíveis e comunicação dentro do projeto.

Além das funcionalidades definidas anteriormente para cada papel, também foram levantadas as seguintes funcionalidades adicionais, considerando-se a implantação de um PSEE na Internet (estas funcionalidades

foram levantadas e discutidas em reuniões do grupo de pesquisa AMADEUS – Ambientes e Metodologias Adaptáveis de Desenvolvimento Unificado de *Software*).

a. **Ferramentas CASE**:

a.1. Programação de alto nível: espera-se que ferramentas CASE permitam a especificação de soluções em alto nível, através de linguagens de domínio da aplicação, ou mesmo de linguagem natural;

a.2. Geração automática de código: sempre que possível o código-fonte de uma aplicação deverá ser gerado automaticamente pelas ferramentas, retirando-se esta tarefa do desenvolvedor de *software*;

a.3. Sincronização entre níveis de especificação: quando houver a alteração em uma determinada especificação, todas as especificações correspondentes em níveis de detalhamento diferente devem ser atualizadas. Por exemplo: a atualização de um código-fonte de um programa deve alterar a especificação em alto nível deste programa automaticamente;

a.4. Reuso de maior nível: deseja-se ferramentas que permitam a reutilização de sistemas de *software* em nível mais elevado, como reutilização de especificações de requisitos, cenários, *frameworks*, e outras especificações relativas à análise e projeto de sistemas;

a.5. Editores cooperativos: introduzindo a tecnologia de CSCW (*Computer-Supported Cooperative Work*) no ADS, espera-se que algumas ferramentas CASE permitam o compartilhamento de trabalho entre membros de uma equipe de desenvolvimento;

a.6. Suporte a verificação de consistência: ao utilizar uma ferramenta CASE, o engenheiro de *software* pode realizar suas tarefas mesmo violando temporariamente regras de consistência. Tais regras seriam então verificadas posteriormente, e ações correspondentes seriam tomadas, para que a consistência fosse restabelecida. Tal funcionalidade permitiria uma maior liberdade em tarefas de modelagem complexa.

b. Gerenciamento de ferramentas:

b.1. Editores cooperativos: deseja-se que as ferramentas apoiem o trabalho conjunto, incluindo mecanismos que permitam a alteração síncrona de informações entre ferramentas;

b.2. Facilidades para *workgroup*: considerando-se que o trabalho de desenvolvimento seja realizado em conjunto, devem existir mecanismos que dêem suporte o trabalho em grupo;

b.3. Consistência entre ferramentas: resultado de um bom processo de *software*, deseja-se que os resultados temporários e finais das ferramentas CASE envolvidas em um projeto sejam consistentes.

c. Gerenciamento de interface com o usuário:

c.1. Ferramentas para criação da interface visual: o gerenciamento de interface deverá prover ferramentas para implementação de interfaces visuais a partir de componentes padronizados, gerando interfaces consistentes para todas as ferramentas.

d. Gerenciamento de configuração:

d.1. Ambiente genérico com suporte a aplicações de determinados domínios: o gerenciamento de configuração deve controlar quais as ferramentas, produtos esperados e outros dados, estão relacionados em um determinado projeto;

d.2. Propagação de mudanças: devem existir políticas e mecanismos que controlem a propagação de mudanças durante o desenvolvimento de um *software*, como por exemplo: a mudança de um código fonte (ou de um modelo mais detalhado) deve ser bloqueado, caso não possa ser propagado para os outros modelos;

e. Gerenciamento de requisitos:

e.1. Reuso de alto nível: deseja-se que o PSEE permita a reutilização não apenas de *software*, mas também de especificação de alto nível, como especificações de requisitos que possam acelerar o desenvolvimento de sistemas dentro de um mesmo domínio de aplicação.

f. Verificação e validação de projeto:

f.1. Ferramentas de avaliação: devem ser supridas ferramentas para análise de métricas capturadas durante o processo de desenvolvimento.

g. Gerenciamento de documentação:

g.1. Reestruturação de um *framework* a partir das aplicações: aplicações geradas a partir de *frameworks* podem sofrer alterações posteriores. A documentação associada auxiliará o

refinamento do *framework,* a partir das aplicações documentadas;

g.2. Reuso de maior nível: deseja-se a reutilização de sistemas de *software* em nível mais elevado, como reutilização de especificações de requisitos, cenários, e outras especificações relativas à análise e projeto de sistemas, que devem estar documentadas adequadamente.

h. Sistema de gerenciamento de objetos:

h.1. Cópia de um subconjunto do modelo para cada tarefa: cada tarefa deverá trabalhar sobre um conjunto restrito (visão) dos dados contidos na enciclopédia, desta forma deseja-se que haja um controle sobre estes conjunto de dados, como por exemplo bloqueio (*lock*) e granularidade do conjunto. Isto permitiria que as ferramentas CASE independentes, possam controlar melhor a consistência e integridade dos seus dados;

h.2. Controle de servidores e clientes: buscando maior autonomia no desenvolvimento de suas tarefas, o SGO deve prover facilidades para distribuir dados entre as ferramentas CASE;

h.3. Facilidades de armazenamento e recuperação de componentes: o SGO deverá fornecer mecanismos de armazenamento e recuperação de dados em vários níveis de granularidade, que são importantes para o desenvolvimento e configuração de sistemas.

As funcionalidades levantadas nesta seção serão utilizadas como requisitos básicos para a arquitetura proposta no capítulo seguinte.

2.4 Reflexão computacional

Uma das características inerentes ao processo de *software* é que o mesmo deve ser adaptável a partir de sua execução, permitindo que o processo seja alterado e melhorado continuamente. Esta característica indica que o processo de *software* deveria ser reflexivo, isto é, ser capaz de processar informações sobre ele mesmo.

Visando a apresentação da arquitetura reflexiva para um ambiente centrado em processo, são apresentados, inicialmente, alguns conceitos básicos relativos à reflexão computacional, e posteriormente, serão discutidos aspectos relativos a utilidade de reflexão para a implementação de um PSEE.

Um sistema é dito reflexivo se este é capaz de realizar processamento sobre seu próprio processamento (MAES, 1987; LISBOA, 1998). Em particular, na computação, um sistema reflexivo é aquele que obtém

dados sobre sua própria computação, podendo atuar e alterar seu comportamento durante sua execução.

A reflexão computacional está presente em diversos paradigmas existentes: imperativo, funcional, entre outros. Entretanto, no paradigma de orientação a objetos este conceito é melhor compreendido e sobre ele serão apresentados os conceitos a seguir:

a. **Níveis de computação**: um sistema reflexivo pode ser entendido como possuindo em princípio dois níveis (Figura 2.5):

   a.1. Nível base: engloba os objetos envolvidos no processo de execução do sistema dentro do domínio da aplicação. Estes objetos têm por objetivo a solução de um determinado problema dentro de uma organização;

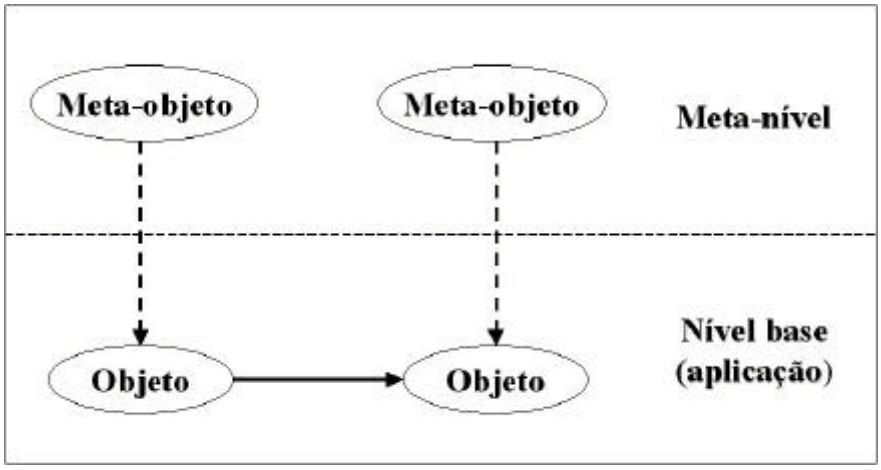

Figura 2.5: Níveis de computação reflexiva

   a.2. Meta-nível: engloba os objetos (meta-objetos) envolvidos com a própria computação do sistema. Pode-se entender que os meta-objetos monitoram objetos do nível base, e podem executar ações frente a determinados eventos que ocorrerem no nível base.

   Pode-se estender o conceito de meta-nível com meta-objetos que controlam os meta-objetos de um meta-nível inferior. Desta forma, é possível implementar inúmeros níveis de reflexão (gerando uma "torre de reflexão" (LISBOA, 1997)). Entretanto, por motivos de desempenho, usualmente os níveis de reflexão não são muito numerosos.

b. **Modelos de reflexão**: basicamente existem dois modelos de reflexão computacional orientados a objetos:

b.1. Baseado em classes: uma meta-classe pode manipular as informações sobre aspectos estruturais das classes do nível base, tais como: nomes e tipos de atributos, nomes e parâmetros de métodos, hierarquia de herança. Este modelo é menos flexível, pois a alteração de uma classe por uma meta-classe afetará todas as instâncias (objetos) desta classe;

b.2. Baseado em objetos: um meta-objeto pode manipular um objeto do nível base a ele associado. Este modelo é mais flexível, pois diferentes objetos de uma mesma classe podem ou não ter meta-objetos associados. Desta forma, cada instância de uma classe pode possuir um comportamento (devido à reflexão) diferenciado.

c. **Estilos de reflexão**: de acordo com o modelo de reflexão adotado existem os seguintes estilos de reflexão:

c.1. Reflexão estrutural: baseado no modelo de classes, permite apenas a manipulação de informações da estrutura das classes do nível base;

c.2. Reflexão comportamental: baseado no modelo de objetos, permite que um meta-objeto obtenha informações e realize transformações sobre o objeto associado. Dependendo da forma de atuação do meta-objeto, o objeto de nível base pode ter suas características alteradas, ou pela consulta/alteração de valores de atributos, ou pela interceptação de mensagens aos seus métodos, ou ambos. Um cenário é apresentado na Figura 2.6. Um objeto cliente envia uma mensagem ao objeto servidor (que possui um meta-objeto associado). A mensagem é capturada pelo meta-objeto. Este inspeciona o objeto de nível base correspondente, realiza computação adicionais e finalmente ativa o objeto de nível base que continua o processamento normal até a resposta ao objeto cliente.

d. **Instrospecção**: é a capacidade de um sistema obter informações sobre seus componentes, sem alterá-los. Um sistema reflexivo deve possuir pelo menos esta característica.

e. **Reificação** (ou materialização): é a transformação da atividade computacional do nível base em informações para o meta-nível.

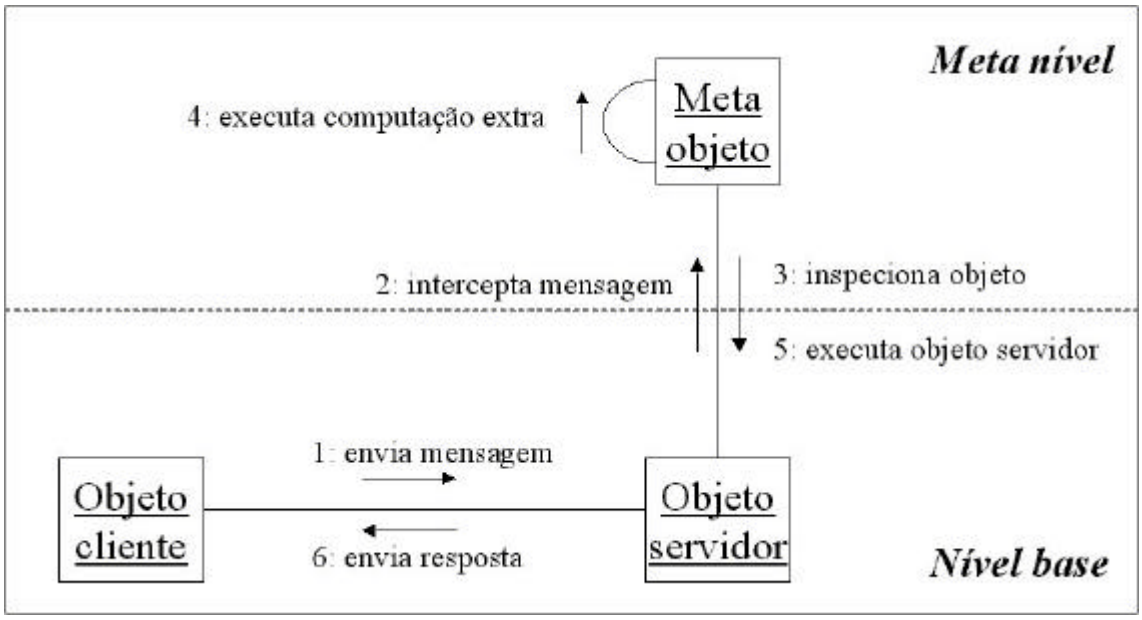

Figura 2.6: Reflexão comportamental

f. **Protocolo de meta-objetos**: o protocolo de meta-objeto ou MOP (*Meta-Object Protocol*) define como o nível base é associado aos meta-nível (CAMPO, 1997; LISBOA, 1998). O MOP define aspectos como:

- transformação das informações do sistema para o seu tratamento no meta-nível;
- associação de objetos do nível base com os meta-objetos (definindo se um meta-objeto está associado a apenas um objeto, a uma classe ou a um grupo de objetos, por exemplo);
- ativação do meta-objeto associado quando da execução de objeto(s) do nível base.

g. **Implementação de reflexão**: a implementação de reflexão em um sistema, do ponto de vista da linguagem de programação, pode ocorrer através dos seguintes mecanismos:

g.1. compilação: a conexão entre o nível base e o meta-nível ocorre durante o processo de tradução do código fonte em executável. Esta abordagem produz maior eficiência e normalmente é realizada no modelo de classes;

g.2. carga: durante a carga do programa no processador é realizada a ligação no meta nível com o nível base;

g.3. execução: a conexão entre o nível base e o meta-nível ocorre a qualquer momento durante a execução do sistema. Isto proporciona maior flexibilidade e permite a adaptação do comportamento dos objetos em tempo de execução.

2.4.1 Usos de reflexão computacional no desenvolvimento de *software*

No contexto de desenvolvimento de *software*, a reflexão computacional (LISBOA, 1998; OLIVA, 1998; DOURISH, 1995) pode proporcionar:

a. **redução de complexidade de implementação**: alguns detalhes do projeto do programa de nível base podem ser delegados ao meta-nível que poderá implementá-los através de meta-objetos específicos. Por exemplo, um método de um objeto da aplicação é implementado de forma simples, entretanto, mais tarde é necessário que haja uma implementação mais complexa do mesmo método. Ao invés de reimplementar um método de um objeto no nível base, alterando a aplicação, é implementado um meta-objeto que implementa este método. Quando o método original é chamado, esta chamada é interceptada e a execução do método é transferida para o meta-objeto;

b. **separação conceitual**: o nível base implementa as funcionalidades relevantes da aplicação, deixando ao meta-nível a implementação de requisitos não-funcionais (como por exemplo, segurança);

c. **estatísticas de desempenho**: obtenção de informações a respeito de consumo de recursos, cumprimento de prazos, satisfação de requisitos, etc. Por exemplo, quando ferramentas CASE implementadas através de objetos são ativadas, estas ativações podem ser interceptadas por meta-objetos que podem armazenar dados como data/hora de ativação, agente que ativou, etc.;

d. **coleta de informações para depuração**: durante o desenvolvimento do sistema, informações sobre o processo podem ser coletadas visando a melhoria do processo. Por exemplo, considerando que o processo de *software* foi modelado por objetos de nível base, meta-objetos irão coletar dados durante toda a execução do processo. Estas informações poderão ser utilizadas *a posteriori* para identificar partes do processo que podem ser otimizadas;

e. **introdução de mecanismos de tolerância a falhas**: um *software* aplicativo pode ser adaptado, pela inclusão de meta-objetos, passando a possuir mecanismos de tolerância a falhas para operações críticas, como por exemplo, o armazenamento

de uma informação na enciclopédia de dados do ADS que por problemas de rede não pode ser acessado. Um meta-objeto pode detectar a situação e executar ações para solucionar o problema;

f. **adaptabilidade**: com a introdução de reflexão computacional, é possível estender a funcionalidade original de uma aplicação através da modificação do comportamento "normal" dos objetos do nível base pelos meta-objetos do meta-nível. Por exemplo, na manutenção de um sistema orientado a objetos, ao invés de estender um método no nível base, quando este for ativado, um meta-objeto intercepta a chamada, o método original é executado. Porém antes de retornar a resposta, um método no meta-nível é executado também sobre o resultado gerado;

g. **monitoramento de execução**: em um produto de *software* liberado, a reflexão permitiria um acompanhamento posterior a sua liberação, permitindo realizar manutenções mais precisas. Caso um aplicativo contenha meta-objetos, quando for solicitada a manutenção do *software*, seria possível, através das informações coletadas pelos meta-objetos, identificar quais métodos deveriam ser alterados/adaptados;

h. **reconfiguração automática**: componentes de *software* podem ser inseridos dinamicamente no processo de desenvolvimento de um produto de acordo com novas necessidades. Por exemplo, na liberação de uma nova versão de um objeto, os meta-objetos detectam a ativação de objetos da versão anterior e a execução é transferida para o novo objeto.

2.5 *Workflow*

Identifica-se como *workflow* uma das tecnologias desenvolvidas com o propósito de minimizar o problema da coordenação do trabalho nos processos (de negócio), através da organização de diversas tarefas que realizarão tais processos. A tecnologia de *workflow* permite processos de análise, modelagem, implementação e revisão dos processos, trazendo consigo redução de tempos de execução, respostas e custos (AMARAL, 1997).

Segundo a WfMC (*Workflow Management Coalition*) (WORKFLOW MANAGEMENT COALITION, 2002a), "*workflow*" significa "a automação total ou parcial de um processo, durante a qual documentos, informações e tarefas são trocados entre os participantes do processo". Sendo que um processo consiste em uma rede de tarefas com relacionamentos e critérios de início e término, informações individuais sobre cada tarefa, participantes, aplicações e dados. Uma tarefa descreve um fragmento de trabalho que contribui para o cumprimento do processo. As tarefas do processo devem ser executadas de forma coordenada, respeitando-se a seqüência prevista para sua execução,

bem como o cumprimento das dependências e pré-condições existentes entre as mesmas.

Um sistema que provê a formalização na definição de processos e uma máquina de *workflow* para a execução dos processos definidos, é chamado de Sistema de Gerenciamento de *Workflow* (WfMS - *Workflow Management System*). O principal objetivo de um WfMS é assegurar que tarefas apropriadas sejam executadas pela(s) pessoa(s) certa(s), no tempo correto. Do ponto de vista computacional, um WfMS define um conjunto de interfaces para usuários e aplicações, através de APIs (*Application Programming Interfaces*) envolvidos nos processos de *workflow*. Desta forma, um WfMS pode ser entendido como um conjunto de ferramentas e aplicações de controle utilizado para projetar, definir, executar e monitorar processos.

O relacionamento de um WfMS com processo de *software* é que o *workflow* permite que o processo de desenvolvimento de *software* seja formalmente modelado e as tarefas definidas no processo possam ser acompanhadas durante sua execução (MANGAN, 1998).

De forma adicional, um WfMS está relacionado a um PSEE nos seguintes aspectos (ARAUJO, 1999):

- separação entre modelo de processo e suas instâncias em execução: um sistema de *workflow* busca separar a definição do processo de sua execução, permitindo maior flexibilidade para a sua evolução;
- apoio à cooperação: ao coordenar as atividades de um grupo de trabalho, um WfMS incentiva o trabalho cooperativo que pode ser explorado em um PSEE, visando maior produtividade;
- heterogeneidade: uma das preocupações de um WfMS é a integração com outros ambientes de *workflow*, que reflete em um PSEE a preocupação de integração de ferramentas diversas ao ambiente.

No contexto deste trabalho esta tecnologia é interessante para as características de modelagem do processo de *software*, bem como da gerência de execução do processo (FINKELSTEIN, 1994), através de um *workflow engine*, em um PSEE.

2.6 Objetos distribuídos e *middleware*

Um objeto distribuído como qualquer objeto provê métodos que podem ser ativados por um objeto cliente através de mensagens (HOQUE, 1998; UMAR, 1997). Entretanto, um objeto distribuído pode estar localizado em um local (computador ou ambiente) diferente do cliente. Desta forma, objetos distribuídos por estarem em diferentes locais, existindo um sistema de comunicação entre eles, podem ser implementados em linguagens e plataformas diversas. Estas características devem ser transparentes aos objetos clientes. Um cliente não necessita saber qual a linguagem de

implementação, o sistema operacional usado ou a localização do objeto distribuído, apenas precisa conhecer sua referência e sua interface. Com o uso de objetos distribuídos, na realidade não há uma clara distinção entre clientes e servidores, uma vez que um objeto pode ser cliente quando envia uma mensagem a outro objeto, porém pode ser um servidor para um terceiro objeto.

Para intermediar a troca de mensagens entre objetos distribuídos é necessário alguma forma de *middleware*. Um *middleware* (COULOURIS, 2001) é um *software* de comunicação que garante que quando um objeto em um sistema envia uma mensagem a outro objeto em outro sistema, esta mensagem chega ao outro sistema e ativa a execução de um método no outro objeto.

Outro elemento relacionado a objetos distribuídos é o conceito de ORB (*Object Request Broker*). O ORB é o *middleware* que estabelece os relacionamentos cliente-servidor entre os objetos. Utilizando um ORB, um cliente pode invocar transparentemente um método de um objeto no servidor, que pode estar na mesma máquina ou em qualquer ponto da rede. O ORB intercepta a chamada por parte do cliente e tem a responsabilidade: localizar o objeto que implementa a chamada, passar os parâmetros necessários a este objeto, fazer a chamada dos métodos e retornar dos resultados. Para o cliente a única coisa importante é a interface do objeto servidor. Ao realizar esta tarefa, o ORB oferece uma solução para a interoperabilidade entre aplicações em máquinas diferentes em ambientes distribuídos e heterogêneos, ao mesmo tempo que interconecta múltiplos sistemas baseados em objetos.

2.7 Arquitetura reflexiva para um PSEE baseado na Web

Como apresentado nas subseções anteriores, atualmente existem alguns ambientes e arquiteturas que provêem suporte ao desenvolvimento de *software*. Entretanto, nenhum foi projetado especificamente para contemplar simultaneamente o suporte ao processo de *software* aliado à reflexão computacional, baseado em ambiente distribuído heterogêneo, no caso, a Web.

O uso de reflexão computacional no processo de *software* foi abordado por alguns autores (BANDINELLI, 1993; SA, 1995; JAMART, 1994). Dentre os seus usos em um PSEE, a reflexão computacional poderia permitir:

a. **Conexão entre modelo de processo e de projeto**: a alteração no processo reflete em projetos instanciados. Se um processo é alterado quando alguns projetos já estão em andamento, os mesmos podem manter suas execuções até o final (por bloqueio da alteração), ou são adaptados automaticamente. Também seria possível que alterações em projetos instanciados pudessem alterar o processo;

b. **Configuração de *software*** (suporte de processo): alteração na estrutura de um produto de *software* (artefatos) reflete no controle de configuração de *software*. Desta forma, meta-objetos

correspondentes podem ativar ações como geração da documentação correspondente, chamada de produtos instanciados que possuam o artefato alterado, etc.;

c. **Controle de projeto**: a execução de um projeto (ativação de ferramentas, execução de atividades, cronogramas) reflete no controle (gerência) do projeto. Desta forma, ações que seriam realizadas por agentes humanos, podem ser executadas por meta-objetos que monitoram a execução de um projeto;

d. **Reconfiguração do ambiente**: a introdução de novas ferramentas automaticamente altera os projetos já em desenvolvimento. Por exemplo, uma nova ferramenta CASE é disponibilizada, desta forma, os meta-objetos responsáveis pelo gerenciamento de ferramentas podem transferir a ativação da ferramentas CASE antiga para a nova;

e. **Manutenção de *software***: sistemas computacionais já desenvolvidos com reflexão computacional, poderiam ser monitorados constantemente. Isto é, aplicações instanciadas, possuem meta-objetos que monitoram a execução e reportam as informações obtidas para o ambiente de desenvolvimento.

3 ARQUITETURA REFLEXIVA PARA AMBIENTE DE SUPORTE A PROCESSO

Uma vez contextualizado o trabalho, este capítulo descreve a arquitetura reflexiva proposta nesta investigação.

Inicialmente são apresentados os conceitos envolvidos na modelagem e execução do processo que possui suporte pela arquitetura; no restante do capítulo, a estrutura da arquitetura proposta é detalhada.

3.1 Modelagem de processo

Este trabalho propõe que o processo de *software* seja definido com um conjunto de objetos computacionais. Deste modo, um projeto de *software* em desenvolvimento é somente um conjunto de objetos instanciados representando objetos do mundo real (ferramentas CASE, agentes, artefatos).

a. **Cenário geral**

A Figura 3.1 apresenta um cenário geral de atuação dos gerentes de processo e de projeto.

Uma organização pode definir alguns processos de *software* que são usados para gerar sistemas de *software*. O gerente de processo possui algumas atividades de sua responsabilidade, tais como: definir novos processos de software, controlar os recursos disponíveis na organização e gerenciar projetos de desenvolvimento de software em andamento.

Cada sistema de *software* é desenvolvido como um projeto de *software*, que possui um gerente de projeto responsável. Quando um novo sistema de *software* é necessário, o gerente de processo é o responsável por iniciar a especificação, definir os recursos gerais e o cronograma básico do projeto de *software* a ser desenvolvido, também sendo o responsável por designar um gerente de projeto responsável pelo mesmo.

O gerente de projeto é o responsável por definir as características particulares do projeto para o desenvolvimento do sistema de *software*, sob as

diretrizes do processo de *software* correspondente. Uma responsabilidade fundamental do gerente de projeto é definir o *workflow* específico do projeto. Este *workflow* é definido sob um processo de *software* escolhido para o projeto.

Após definir estas características, o gerente de projeto instancia o seu *workflow*, definido as tarefas necessárias, os artefatos intermediários que compõem o produto de *software* final, os agentes (engenheiros de *software*) envolvidos, as ferramentas necessárias, entre outros aspectos. Após a fase de modelagem, a execução é iniciada através de um sistema de gerência de *workflow*.

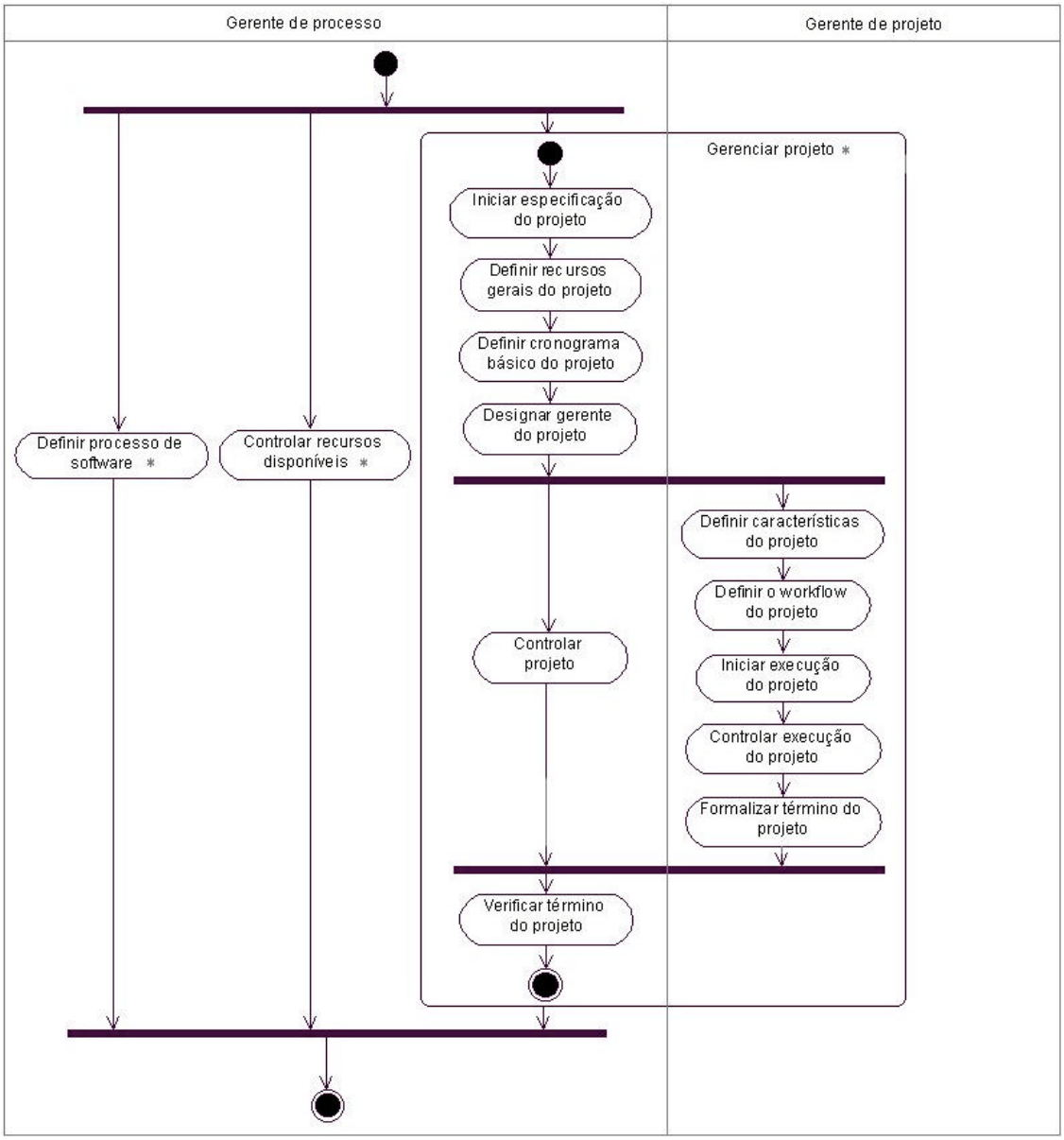

Figura 3.1: Cenário geral

Quando o sistema de gerência de *workflow* inicia uma tarefa, o engenheiro de *software* correspondente é avisado. O engenheiro de *software*

responsável deve então iniciar a execução da tarefa definida no *workflow* do projeto, utilizando artefatos já produzidos anteriormente e utilizando ferramentas para a geração de um novo artefato. Tarefas podem ser executadas em paralelo de forma que em um determinado período de tempo várias tarefas podem estar em execução simultaneamente.

Uma tarefa é terminada quando o engenheiro de *software* gera um novo artefato previsto para a mesma. Este artefato pode ser usado como entrada para outra tarefa. Ao final da execução de cada tarefa definida no *workflow* do projeto, outra tarefa é iniciada e outro engenheiro de *software* é chamado para executá-la.

Paralelamente à execução das tarefas pelos engenheiros de *software*, tanto o gerente de processo quanto o gerente de projeto controlam a execução do projeto (como por exemplo, cumprimento de cronograma e custo, artefatos gerados, execução de tarefas por engenheiros de *software*, etc.).

O projeto é finalizado quando todas as tarefas definidas no *workflow* foram concluídas gerando os artefatos necessários ao sistema de *software* desejado.

b. **Um cenário-exemplo**

Durante a execução do processo (desenvolvimento de um projeto de *software*) um engenheiro de *software*, por exemplo: Arthur, com o papel de um analista de sistemas, *loga*-se no ambiente usando um navegador Web, pois recebeu um *e-mail* informando-o que uma nova tarefa (realizar a análise de um caso de uso, segundo o Processo Unificado) (JACOBSON, 1998) foi atribuída para ele (Figura 3.2). Após o *log in,* é apresentado a ele uma lista de suas tarefas, bem como os recursos disponíveis e os artefatos esperados para serem desenvolvidos. O analista seleciona uma tarefa e realiza o *check-in* da mesma, indicando o aceite da mesma. Este analista pode também decidir usar uma ferramenta CASE particular para a realização da tarefa, desta forma deve possuir meios de incorporá-la ao ambiente.

A execução da tarefa pode levar muitos dias, portanto muitas sessões no ambiente são necessárias. Antes de cada execução da tarefa é realizado um *login* na mesma (que permite ao ambiente acessar os artefatos e ferramentas necessárias), ao final de cada execução é realizado um *logout*. Todo o trabalho desenvolvido pelo analista pode ser observado pelo gerente de projeto. Quando o analista termina a análise do caso de uso, ele submete o artefato resultante para o ambiente e a tarefa é concluída (*check-out*).

O gerente de projeto pode observar que a tarefa foi finalizada e o sistema de gerência de *workflow* automaticamente inicia a próxima tarefa.

Este gerente de projeto deve ter a possibilidade de adaptar o projeto em execução adicionando, removendo ou alterando as tarefas, artefatos, agentes e recursos definidos no *workflow* do projeto.

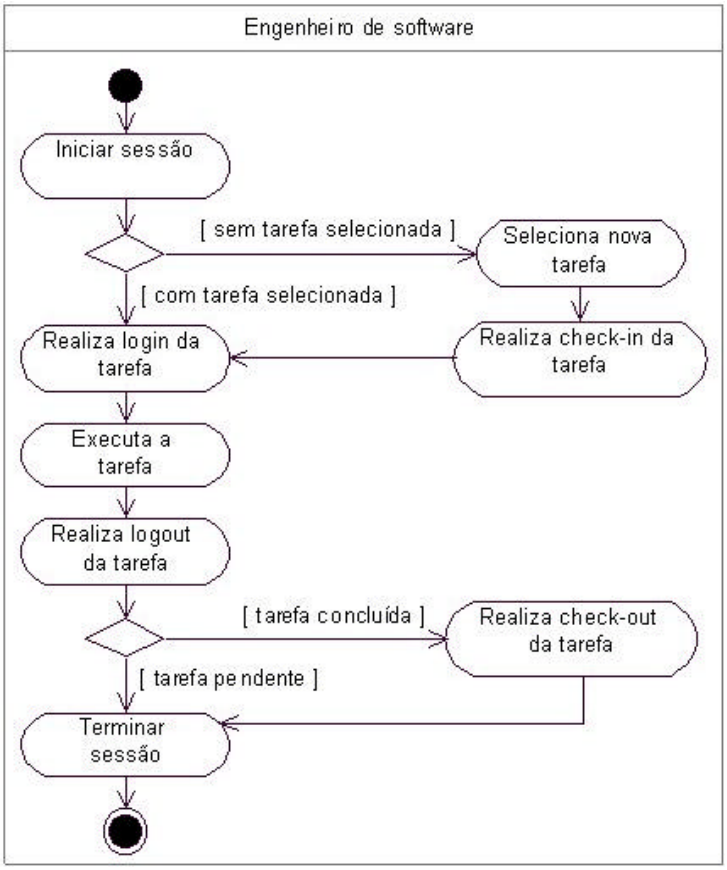

Figura 3.2: Cenário exemplo

Todos os projetos em execução na organização também estão sob o controle do gerente de processo, que pode alterar recursos de um projeto para outro, redefinir cronogramas e até mesmo mudar um processo básico, devido a dados coletados pelos projetos em execução.

Com base nos cenários apresentados acima, serão apresentados os conceitos utilizados neste trabalho para a modelagem e execução do processo de *software*. Os cenários também ilustram algumas das funcionalidades que devem ter suporte pelo PSEE.

3.2 Modelagem com objetos

Mapeando, o cenário geral e o cenário exemplo apresentados anteriormente, para uma visão computacional orientada a objetos, neste trabalho os conceitos básicos de processo de *software* seriam definidos da seguinte maneira:

a. **Linguagem de modelagem de processo (PML – *Process Modeling Language*)**: é definida por uma linguagem de

modelagem orientada a objetos. Este trabalho usa UML (*Unified Modeling Language*) (BOOCH, 1998) como linguagem de modelagem. Existem estudos similares que propõem linguagens específicas para a modelagem do processo, como SLANG (BANDINELLI, 1993), Tempo (BELKHATIR, 1994). UML é usada neste trabalho pois está se tornando um padrão *de fato* para a modelagem de objetos e porque esta linguagem possui mecanismos para estendê-la. Um diagrama de classes estendido pode ser utilizado para modelar as meta-classes envolvidas em processos genéricos da organização. A Figura 3.3 mostra um diagrama de classes UML envolvendo os elementos básicos de processos genéricos. Os elementos básicos de um processo utilizados neste trabalho foram apresentados na seção 2.3.1.

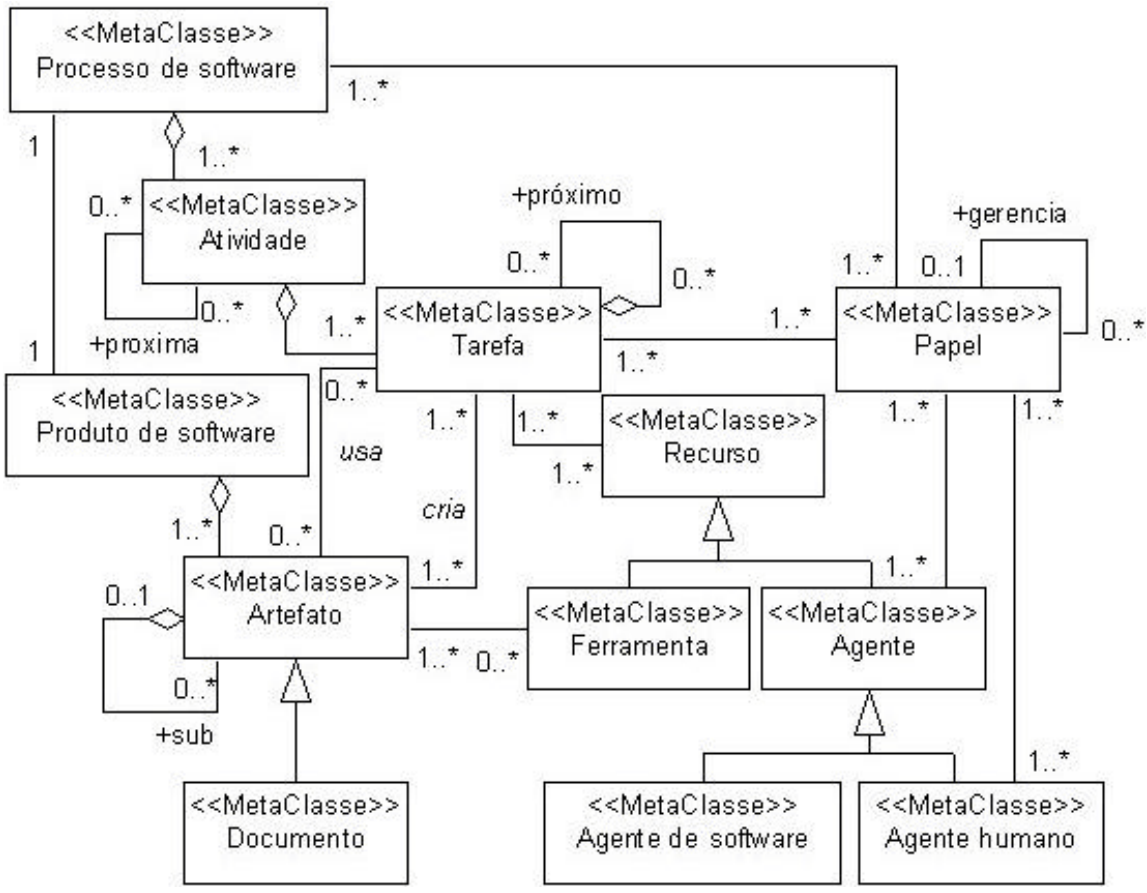

Figura 3.3: Meta-modelo para meta-processos

b. **Meta-processo**: é definido por um modelo que apresenta as classes instanciadas do meta-modelo que estão envolvidas em um processo de *software* particular. Este modelo é provido por um diagrama de classes representando as classes básicas envolvidas no meta-processo. A Figura 3.4 mostra um diagrama de classes UML que apresenta as classes envolvidas na

atividade de 'Análise' do Processo Unificado (adaptado de (JACOBSON, 1998)). As classes apresentadas foram instanciadas das meta-classes do meta-modelo: "Análise de Caso de Uso" é uma classe instanciada da meta-classe "Tarefa" que é antecedida da Tarefa "Análise Arquitetônica" e sucedida pela Tarefa "Análise de Classe". A Tarefa "Análise de Caso de Uso" é executada pelo Papel "Engenheiro de Casos de Uso", possui o Artefato "Caso de Uso" como entrada e gera os Artefatos "Caso de Uso - Análise" e "Classe de Análise [esboço]". A Ferramenta "Ferramenta de Caso de Uso" é utilizada pela Tarefa "Análise de Caso de Uso" pelo Papel "Engenheiro de Caso de Uso" e manipula os Artefatos "Caso de Uso", "Caso de Uso - Análise" e "Classe de Análise [esboço]".

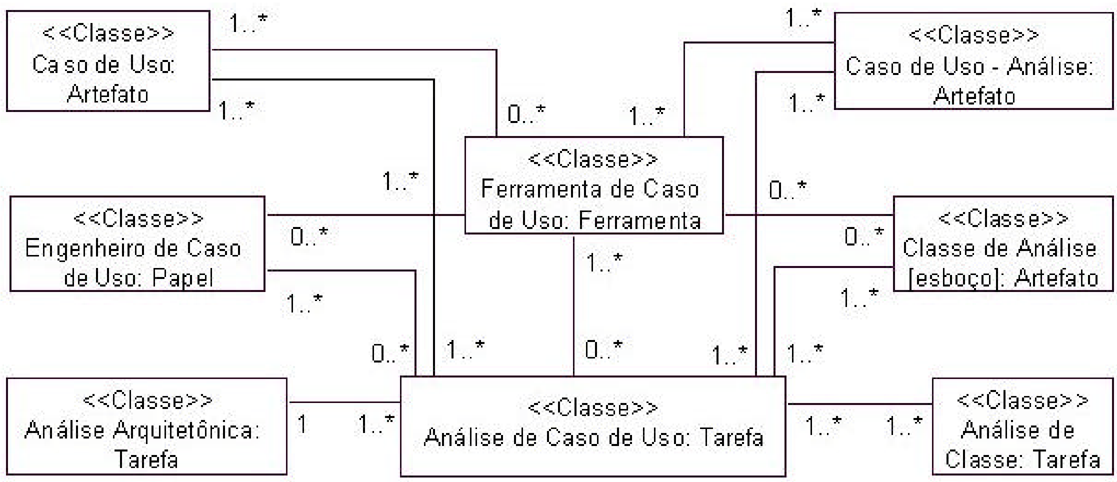

Figura 3.4: Um meta-processo – parcial

De forma adicional, a modelagem do meta-processo também envolve a definição de seu *workflow*. Neste trabalho, o *workflow* do processo é representado através de um diagrama de atividades UML que conecta as tarefas do processo. A Figura 3.5 mostra um diagrama de atividades relativo a atividade de 'Análise' do Processo Unificado.

c. **Projeto**: um projeto é definido como um conjunto de objetos. Estes objetos são instâncias do modelo do meta-processo. Neste trabalho, um projeto é representado por um diagrama de objetos UML. Cada objeto representa um elemento envolvido e alocado para um determinado projeto de *software*. De forma adicional, um diagrama de atividades UML é instanciado para modelar o *workflow* das atividades específicas do projeto. Além disso, a execução do projeto é controlada pela execução do seu *workflow* usando um WfMS (ARAUJO, 1999; WORKFLOW MANAGEMENT COALITION, 2002a).

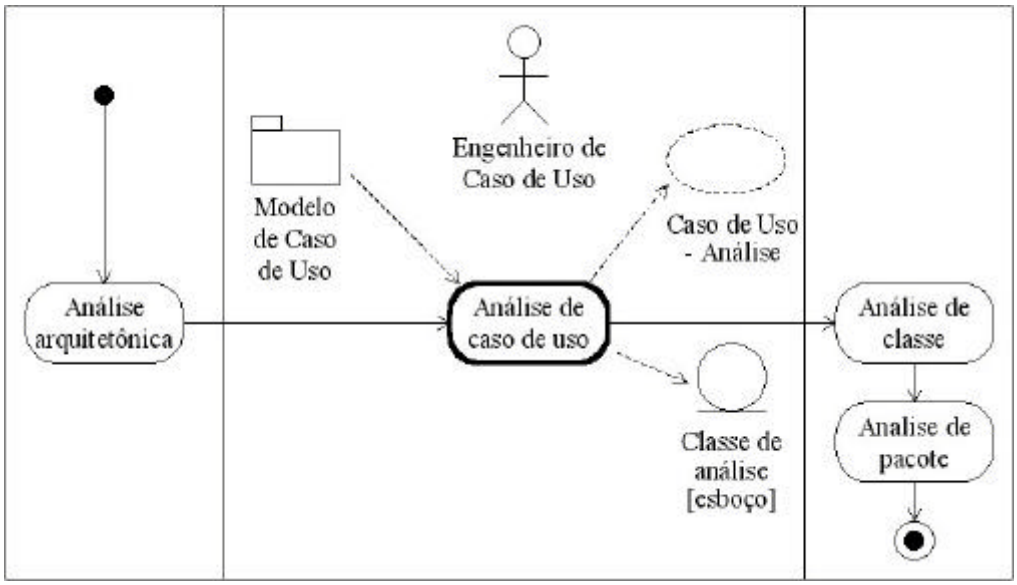

Figura 3.5: Um *workflow* de processo - parcial

A Figura 3.6 mostra um exemplo de um diagrama de objetos instanciado para um projeto de *software* sob o Processo Unificado.

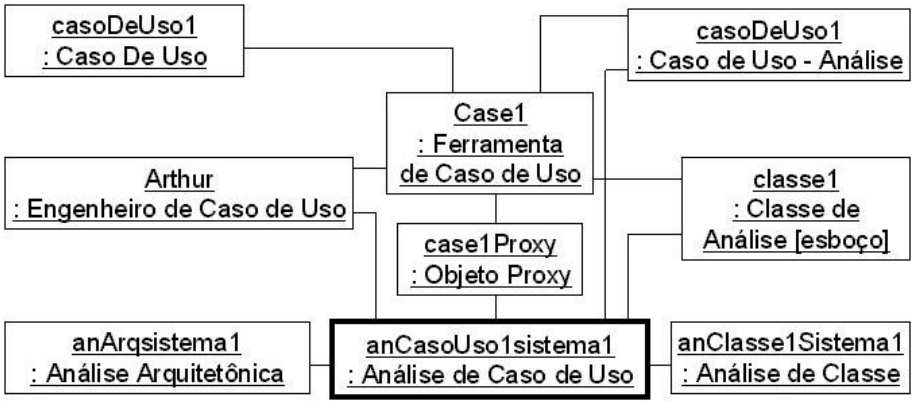

Figura 3.6: Um projeto – parcial

Ferramentas de modelagem e de controle dos modelos propostos devem fazer parte do PSEE. Estas ferramentas ficam disponíveis para os gerentes de projeto e de processo, a fim de que os mesmos realizem suas tarefas de gerência.

3.3 Estrutura da arquitetura

Com base nos conceitos apresentados, as seções seguintes descrevem a arquitetura reflexiva proposta, nomeada de WRAPPER (*Web-based Reflective Architecture for Process suPport EnviRonment*) - Arquitetura reflexiva baseada na Web para ambiente de suporte a processo.

A arquitetura básica do WRAPPER é organizada em camadas ou níveis (SHAW, 1996; BUSCHMANN, 1996). A estrutura básica define dois níveis: o nível base e o meta-nível. Sobre estes dois níveis é construída a camada da aplicação: o ambiente de desenvolvimento de *software* para suporte a processo de *software*. Todos os níveis levam em consideração sua distribuição na Web.

O nível base é representado pelos objetos da aplicação. Neste caso, os objetos representam os elementos básicos envolvidos no PSEE, bem como os elementos envolvidos nos processos.

O meta-nível é responsável por prover mecanismos de gerência de meta-objetos. Estes meta-objetos são responsáveis por monitorar os objetos do nível base.

Sobre estes dois níveis é criado um PSEE constituído dos componentes que provêem suporte as funcionalidades para a definição e controle de processos de *software*.

As seções seguintes descrevem os níveis em maior detalhe.

3.3.1 Nível base

O nível base da arquitetura é responsável pelo controle do registro e acesso de objetos em um ORB (*Object Request Broker*) – responsável pela intermediação de mensagens entre objetos distribuídos. A Figura 3.7 apresenta a estrutura do nível base. As setas largas representam fluxos de controle e de dados.

No registro, um objeto pode ser classificado em um dos seguintes tipos:

a. **Objeto aberto**: é um componente criado para o ambiente ou que disponibiliza acesso a sua estrutura interna (como por exemplo, uma ferramenta CASE de código aberto ou que possua mecanismos de exportação de dados, de forma que a arquitetura tenha acesso sobre as informações e funcionalidades disponibilizadas pela ferramenta). No registro deste objeto deve ser criada e disponibilizada uma especificação de sua interface, de forma que outros objetos do ORB tenham acesso a suas características;

b. **Objeto fechado**: representa um componente que não provê acesso a sua estrutura interna (por exemplo, uma ferramenta

CASE proprietária ou um sistema legado que não possua estrutura aberta, nem exportação de dados, nem mesmo disponibilize formas de ativação de suas funcionalidades internas pela arquitetura). Para este objeto, o ambiente cria e registra um objeto-*proxy*. Este objeto-*proxy* é registrado no ORB e é responsável por tratar as requisições ao componente. Obviamente, informações mais detalhadas do objeto fechado não são disponibilizadas no ORB, ficando as informações restritas àquelas que o objeto-*proxy* possa produzir.

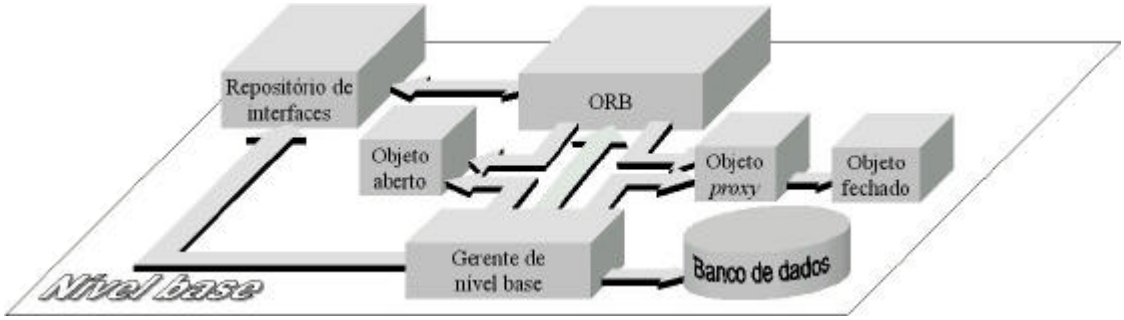

Figura 3.7: Estrutura do nível base

O nível base utiliza os serviços básicos providos pelo ORB para registro e consulta de objetos no mesmo. A arquitetura do WRAPPER possui um gerente de nível base, que é o responsável por intermediar as operações de registro e consulta de objetos no ORB. De forma adicional, este gerente atualiza o repositório de interfaces com novas definições de interface.

O gerente de nível também gerencia o armazenamento físico dos objetos persistentes. Objetos persistentes podem ser armazenados em arquivos, porém a arquitetura também utiliza um banco de dados que também pode armazenar os objetos persistentes deste nível.

Para o registro de um objeto sem acesso a sua estrutura interna (um objeto fechado), o gerente cria um objeto-*proxy* que é registrado no ORB. O processo de criação redireciona as referências do objeto fechado para o objeto-*proxy*, que é tratado como um objeto comum do ORB.

Além do controle de registros, o gerente de nível base é responsável por prover facilidades para os níveis superiores para a manipulação dos objetos do ORB.

Para facilitar o controle dos diversos tipos de objetos deste nível, o gerente de nível base é constituído de diversos sub-gerentes que são responsáveis pelo controle de acesso e armazenamento de seus objetos correspondentes, como por exemplo: sub-gerente de objetos-Tarefa, sub-gerente de objetos-Artefato, e assim por diante.

O nível base é implementado sobre um ORB. Este ORB provê facilidades básicas, tais como processamento de interfaces de objetos,

operações de manipulação do próprio ORB, bem como disponibiliza um repositório de interfaces para facilitar a manipulação da interface dos objetos.

A arquitetura WRAPPER prevê que o ambiente pode ser distribuído em vários ORBs remotos. Entretanto, um ORB é considerado o principal (servidor) e este é responsável pelo controle global do ambiente. Sobre este ORB, é previsto que o núcleo do PSEE seja implementado.

Na plataforma no lado de um cliente um ORB pode ser instalado de duas formas:

- cliente completo: na qual deve haver um processo de instalação de *software* na plataforma do cliente.
- execução por um navegador Web: na qual o ORB é "baixado" juntamente com páginas Web disponibilizadas no servidor.

Independentemente da forma utilizada no cliente é possível o registro de objetos locais ao mesmo, como por exemplo, ferramentas CASE locais.

3.3.2 Meta-nível

O meta-nível da arquitetura é representado por meta-objetos associados aos objetos do nível base. A Figura 3.8 apresenta a estrutura do meta-nível. As setas largas representam fluxos de controle e de dados. As setas tracejadas representam fluxos de controle e de dados entre os níveis.

Um meta-objeto pode ser criado a qualquer momento e sua criação e registro no ORB são intermediados por um gerente de meta-nível.

O gerente de meta-nível também provê uma API para permitir a criação e controle de meta-objetos neste nível.

Tabela 3.1: Papéis dos meta-objetos.

Meta-objeto	Papel
Coletor	Coletar informações de objetos ou classes do nível base. A disponibilização das informações pode ser realizada de duas maneiras: em arquivos de *log* ou em banco de dados. As informações coletadas são relativas a atributos disponíveis nos objetos e classes do nível base.
Proxy	Interceptar uma mensagem para um objeto do nível base e realizar processamento adicional, podendo ou não passar a mensagem ao objeto original. A interceptação da mensagem pode repassar a mensagem para outro objeto, sem que o objeto original seja notificado.
Genérico	Possui as características anteriores, como interceptação de mensagens e coleta de informações do nível base.

O objetivo de um meta-objeto depende de seu uso. Este pode ser apenas estatístico (consultando características dos objetos de nível base) ou mesmo alterar o comportamento de um objeto de nível base, executando computações adicionais toda vez que o objeto de nível base for ativado. A Tabela 3.1 sumariza os papéis realizados pelos meta-objetos na arquitetura.

No processo de registro de um meta-objeto, o gerente de meta-nível acessa o gerente de nível base para consultar os objetos registrados no ORB. Dependendo do protocolo de meta-objetos (MOP) um meta-objeto pode ser associado a diferentes elementos do nível base. Para permitir flexibilidade na associação do meta-objeto com o nível base, a arquitetura deve dar suporte às seguintes associações:

a. **qualquer objeto – qualquer operação**: o meta-objeto é associado a qualquer ativação de operação em qualquer objeto do nível base, ou seja, qualquer troca de mensagens que ocorrer no ORB é capturada pelo meta-objeto.

b. **um objeto – uma operação**: o meta-objeto é associado a uma determinada operação de um determinado objeto.

c. **um objeto – muitas operações**: a associação é de algumas operações de um determinado objeto do nível base.

d. **um objeto – todas as operações**: o meta-objeto é ativado quando o objeto associado receber alguma mensagem.

e. **grupo de objetos**: define-se um grupo de objetos que ao receber uma mensagem ativa o meta-objeto correspondente.

f. **uma classe – uma operação**: o meta-objeto é associado a uma determinada operação definida em uma classe. Desta forma, a ativação da operação em qualquer objeto desta classe será capturada pelo meta-objeto.

g. **uma classe – muitas operações**: é definido um conjunto de operações de uma classe. Quando alguma destas operações for chamada em algum objeto da classe definida, o meta-objeto é ativado.

h. **uma classe – todas as operações**: o meta-objeto intercepta mensagens enviadas a qualquer objeto da classe definida.

i. **grupo de classes**: define-se um grupo de classes de forma que o meta-objeto é ativado quando qualquer objeto das classes definidas receber uma mensagem.

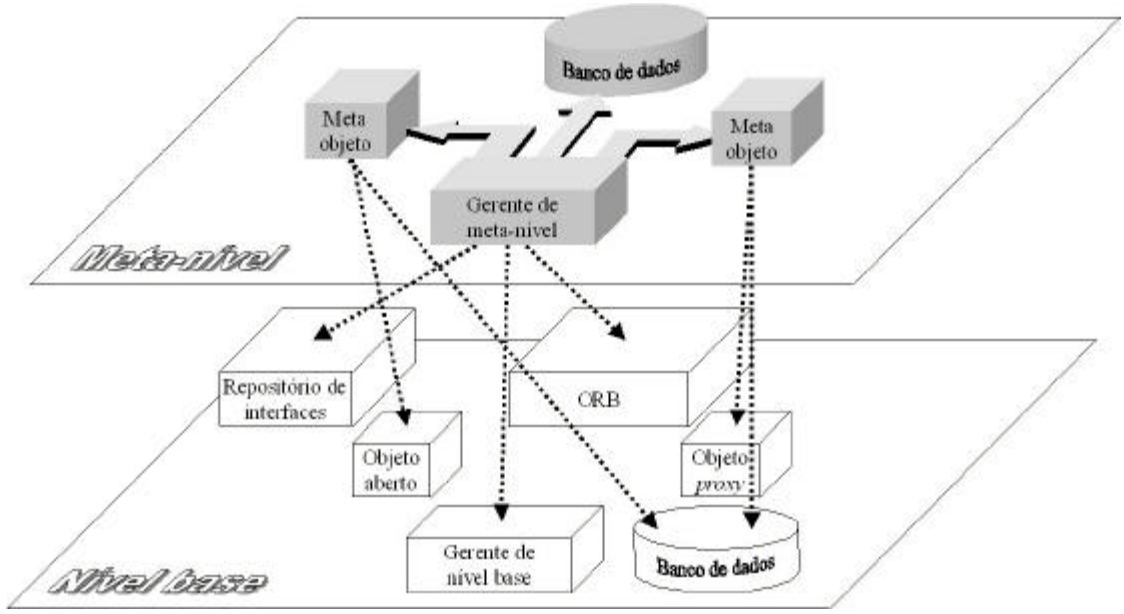

Figura 3.8: Estrutura do meta-nível

Dados coletados por meta-objetos podem ser armazenados em um banco de dados separado. O gerente de meta-nível também é responsável por prover facilidades de controle para este banco de dados.

O meta-nível necessita que haja algum mecanismo de captura (interceptador) de mensagens entre os objetos do ORB. Um interceptador, quando ativado pelo ORB, pode realizar diversas computações antes de passar a requisição adiante. A gerência dos interceptadores (chamados de *Wrappers* nesta arquitetura) também é responsabilidade do gerente de meta-nível.

Além da definição da associação entre meta-objetos e os objetos de nível base também é necessário especificar a funcionalidade de cada meta-objeto. Isto é realizado através de **Serviços *Wrapper*** que disponibilizam serviços diferentes, como por exemplo: captura de informações do nível base e armazenamento destas informações em um arquivo ou banco de dados, captura de mensagem do nível base e transferência de ativação para outro objeto. Para gerenciar os diferentes **Serviços *Wrapper*** é definido um sub-gerente correspondente.

3.3.3 Nível de aplicação

Sobre os dois níveis apresentados anteriormente, a arquitetura também provê um nível de aplicação. Este nível é implementado por diversas aplicações no lado do cliente e no servidor. A Figura 3.9 apresenta a estrutura do nível de aplicação. As setas largas representam fluxos de controle e de dados. As setas tracejadas representam fluxos de controle e de dados entre os níveis.

As aplicações do servidor interagem com o ORB e com os gerentes dos níveis inferiores. Basicamente, estas aplicações provêem interface gráfica para manipular os outros gerentes da arquitetura. Todos os componentes deste nível são acessados por um gerente do nível de aplicação.

Para o gerente de processo, o ambiente deve prover facilidades básicas como por exemplo:

a. **Modelagem dos meta-processos de *software***: a partir do meta-modelo de meta-processos um gerente de processo pode instanciar um modelo definindo um meta-processo específico de *software*;

b. **Instanciação de projeto de *software***: a partir de um meta-processo, pode-se gerar uma instância do mesmo, que corresponde a um projeto de *software*;

c. **Acompanhamento dos projetos**: o gerente de processo deve ser capaz de acompanhar o andamento dos projetos de *software* em execução, bem como atuar sobre os mesmos.

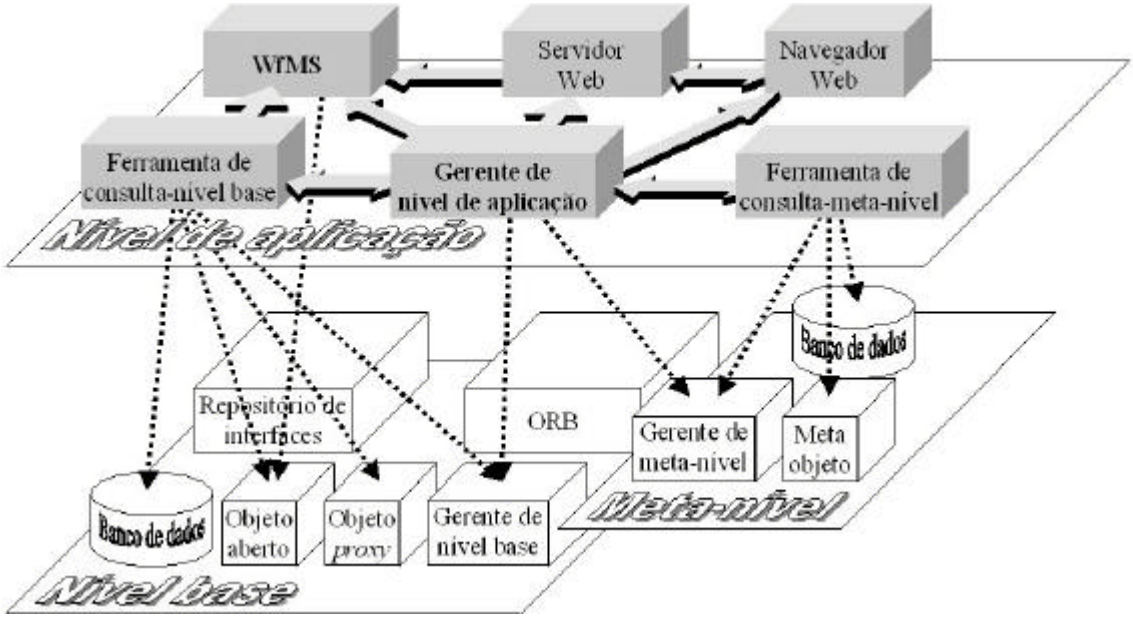

Figura 3.9: Nível de aplicação

O gerente de processo têm ao seu dispor as ferramentas de consulta nível-base (que permite controlar os objetos de nível base do processo) e de consulta meta-nível (que permite controlar os meta-objetos no meta-nível). Estas ferramentas são disponibilizadas, ao agente gerente de processo, através de um navegador Web.

Para um gerente de projeto o ambiente deve prover facilidades, como por exemplo:

a. **Planejamento do projeto**: a partir do meta-processo indicado, tarefas específicas devem ser instanciadas, o cronograma geral

deve ser especificado, os recursos devem ser previstos e alocados, profissionais técnicos devem ser indicados, etc.;

b. **Acompanhamento do projeto**: deve ser possível a qualquer momento verificar o estado atual do projeto (andamento do cronograma, artefatos produzidos, por exemplo), bem como adaptar a execução do projeto frente a novas necessidades ou restrições.

Para o gerente de projeto são disponibilizadas as mesmas ferramentas disponibilizadas ao gerente de processo, apenas com restrito à atividades permitidas a um gerente de projeto. Estas ferramentas também são disponibilizadas, ao agente gerente de projeto, através de um navegador Web.

O gerente de processo, na instanciação inicial de um projeto, deve ser capaz de acessar os objetos disponíveis no PSEE para conectá-los ao novo projeto. Também deve ser definido o *workflow* do projeto, com os respectivos objetos participantes.

A execução do processo (execução do projeto de *software*) corresponde à execução do seu *workflow*. A modelagem e execução de *workflow* são feitas utilizando-se um *Workflow Management System* (WfMS). Considerando os elementos do *workflow* também como objetos, estes são registrados no ORB e podem ser controlados por meta-objetos.

Para guiar a execução do projeto, além do *workflow* do mesmo, um sistema baseado em documentos hipertexto serve de guia para o desenvolvimento do *software*.

O uso de documentos hipertextos para um ADS (ORTIGOSA, 1995; LIMA, 2000) encaixa-se em um ambiente Web através de páginas que podem ser geradas dinamicamente e acessadas remotamente pelos agentes. Páginas HTML (BERNERS-LEE, 1995; LADD, 1998) podem ser geradas dinamicamente à medida em que o projeto é executado. Outra alternativa seria o uso e processamento de XML (*eXtensible Markup Language*) (HOLZNER, 2000; ROCKWELL, 2001) permite que o conteúdo seja separado da apresentação dos documentos, provendo um meio de distribuição de informação e formatação de apresentação de forma distribuída e heterogênea.

Considerando a execução do projeto por um WfMS, durante sua execução os objetos correspondentes no nível base são ativados. Um gerente de projeto pode introduzir meta-objetos correspondentes aos objetos do *workflow* de forma que, quando os objetos de nível base trocam mensagens, os meta-objetos são ativados e podem realizar suas operações, tais como: captura de dados estatísticos ou alteração da execução do *workflow*.

A Figura 3.10 apresenta uma situação exemplo: o WfMS executa o *workflow* definido. O *workflow* tenta executar o objeto correspondente. Como o objeto é registrado no ORB, o mesmo intercepta a requisição e verifica se há um meta-objeto correspondente. O meta-objeto executa a sua computação (no exemplo: consulta dados do objeto de nível base correspondente e armazena os dados em um banco de dados). Após sua execução, o meta-objeto passa adiante a requisição para o objeto de nível base.

O gerente de projeto poderia também extrair informações sobre o projeto em andamento pela introdução de tarefas (objetos) de gerência no *workflow* do mesmo. Pela abordagem sugerida neste trabalho, a informação é extraída por meta-objetos inseridos no meta-nível da arquitetura. Durante a instanciação do projeto, ou mesmo durante a execução do mesmo, o gerente, além de definir os objetos participantes, também pode criar meta-objetos associados aos objetos que deseja monitorar.

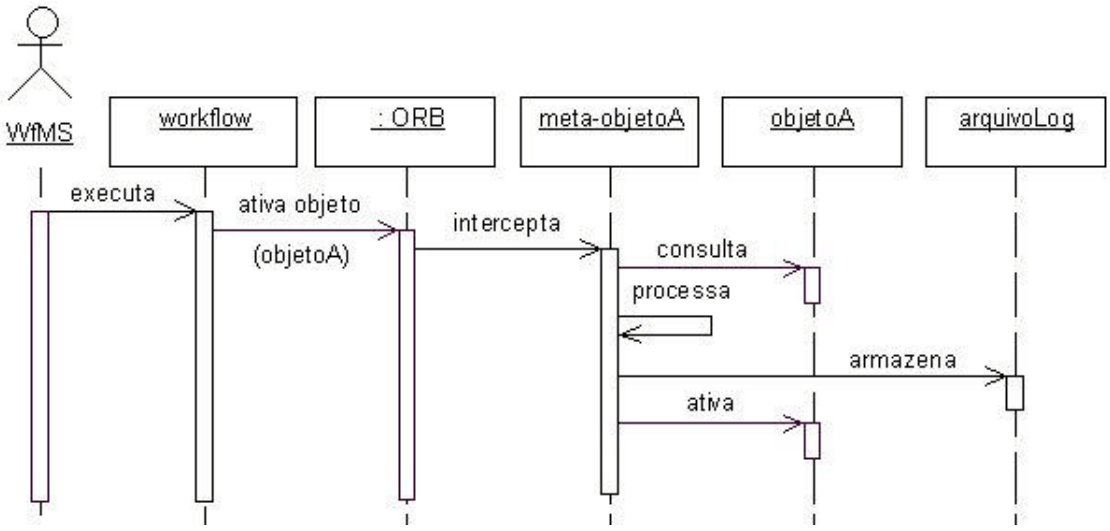

Figura 3.10: Execução de projeto através da execução de *workflow*

A combinação de objetos, meta-objetos e *workflow* permite, como resultado, um modo flexível do controle de execução e de alteração do processo de *software*, bem como do próprio ambiente de suporte ao processo.

3.4 Projeto e implementação das funcionalidades de um PSEE pela arquitetura WRAPPER

Esta seção descreve, a partir de diferentes exemplos, como a arquitetura WRAPPER busca satisfazer as funcionalidades previstas para um PSEE (definidas no capítulo anterior - seção 2.3.3.):

a. **Gerente de processo**:

a.1. Modelagem de meta-processo de *software*:

- Definir o fluxo de trabalho: **utilização de *workflow* para a modelagem. Em UML, modelagem através da utilização de diagrama de atividades**;
- Indicar os artefatos a serem gerados por cada tarefa de um processo: **utilização (em UML) de um diagrama de classes (instanciando-se a meta-classe Artefato)**;

- Definir os recursos necessários à execução de uma tarefa: **utilização (em UML) de um diagrama de classes (instanciando-se a meta-classe Recurso)**;
- Indicar um responsável (papel) pela execução de uma tarefa: **utilização (em UML) de um diagrama de classes (instanciando-se a meta-classe Papel)**;
- Estabelecer a adequação de uso de produtos prontos em determinadas tarefas: **utilização (em UML) de um diagrama de classes (instanciando-se a meta-classe Artefato)**.

a.2. Propagação de alterações de um processo em projetos de *software* em andamento: **por exemplo, suponha que a tarefa "Análise de Caso de Uso" fosse eliminada do processo. Um meta-objeto é instanciado para esta classe de forma que todas as ativações para os objetos já instanciados, sejam interceptados e repassados para outro objeto da classe Tarefa**.

a.3. Instanciação de um projeto de *software* de um determinado processo de *software*:

- Definir o gerente do projeto: **utilização (em UML) de um diagrama de objetos (alocando-se um objeto de uma classe instanciada da meta-classe Agente Humano)**;
- Definir os recursos iniciais (profissionais, equipamentos, *software*): **utilização (em UML) de um diagrama de objetos (alocando-se objetos de classes instanciadas a partir da meta-classe Recurso)**;
- Definir o cronograma geral de projetos, prevendo prazos máximos e mínimos: **utilização (em UML) de um diagrama de atividades que relacione objetos de classes instanciadas da meta-classe Tarefa e definição de atributos de controle de tempo, tais como: data de início previsto, tempo de execução previsto**.

a.4. Acompanhamento de projetos em andamento:

- Verificar o estado atual (real) do andamento de um projeto em relação ao estado projetado: **utilização do WfMS que consultará qual objeto (de uma classe instanciada da meta-classe tarefa) está em andamento no momento**;
- Permitir o deslocamento de recursos entre projetos: **criação de meta-objetos que transfiram as ativações para um objeto (de uma classe instanciada da meta-classe Recurso) para outro objeto**;

- Alterar um projeto pela criação, alteração ou remoção de atividades/tarefas previstas no projeto: **criar um meta-objeto sobre o objeto a ser alterado. As ativações para o objeto seriam interceptadas e tratadas, conforme o objetivo da alteração, pelo meta-objeto**.

a.5. Controle de recursos: **o Gerente de Nível Base é o responsável pelo cadastramento e controle dos objetos existentes no PSEE**.

a.6. Comunicação com gerente(s) de projeto: **meta-objetos podem ser instanciados para objetos (de classe instanciada a partir da meta-classe Agente Humano) que possuam o papel de "gerente de projeto" com o objetivo específico de enviar informações aos gerentes correspondentes, por exemplo, por endereço de *e-mail* cadastrado no ambiente**.

a.7. Planos de contingência: **durante a etapa de modelagem, pode-se utilizar (em UML) um diagrama de atividades indicando quais condições ativam objetos que tratem determinadas contingências. Durante a execução do processo, a qualquer momento podem ser criados meta-objetos que monitorem e tratem objetos "críticos" em caso de alguma condição especial**.

a.8. Mudança de ambiente de suporte a projetos: **criar meta-objetos que monitorem e tratem os objetos correspondentes a objetos (de classes instanciadas da meta-classe Recurso)**.

b. **Gerente de projeto**:

b.1. Planejamento de projeto:

- Detalhar tarefas previstas: **utilização (em UML) de diagrama de atividade e de objetos relacionando os objetos alocados para o projeto**;
- Projetar o orçamento: **acessar informações sobre objetos (de classes instanciadas da meta-classe Recurso) relativas a custo associado**;
- Especificar os requisitos de mudança: **durante a etapa de modelagem, pode-se utilizar (em UML) um diagrama de atividades que indicando quais condições ativam objetos que tratem determinadas situações. Durante a execução, a qualquer momento podem ser criados meta-objetos que monitorem e tratem objetos "críticos" em caso de alguma condição especial**.

b.2. Acompanhamento do projeto: durante o andamento do projeto devem estar disponíveis funcionalidades como:

- Verificar o estado atual (real) com o projetado: **utilização do WfMS que consultará qual objeto (de uma classe instanciada da meta-classe Tarefa) está em andamento no momento, verificando as informações atuais com as projetadas**;
- Alterar o cronograma previsto: **utilização do WfMS que deverá permitir a alteração de informações dos objetos (de uma classe instanciada da meta-classe Tarefa)**;
- Alterar os recursos para uma tarefa: **criar um meta-objeto sobre o objeto (de uma classe instanciada da meta-classe Tarefa) que redirecione os recursos do mesmo**;
- Alterar o projeto pela criação, alteração, remoção de atividades/tarefas: **criar um meta-objeto sobre o objeto a ser alterado. As ativações para o objeto seriam interceptadas e tratadas, conforme o objetivo da alteração, pelo meta-objeto**;
- Comunicar-se com engenheiros de *software*: **meta-objetos podem ser instanciados para objetos (de classe instanciada a partir da meta-classe Agente Humano) que possuam o papel de "engenheiro de *software*", com o objetivo específico de enviar informações aos engenheiros correspondentes, por exemplo, por endereço de *e-mail* cadastrado no ambiente**;
- Verificar o consumo de recursos alocados para o projeto: **criar um meta-objeto para uma classe instanciada da meta-classe Recurso, que colete dados dos objetos correspondentes**;
- Permitir a mudança de pessoal entre atividades do projeto: **criar um meta-objeto para um objeto (de classe instanciada da meta-classe Agente Humano) que repasse as ativações para outros objetos**;
- Verificar a estrutura do produto em determinado período de execução do projeto: **criar um meta-objeto para um objeto (de classe instanciada da meta-classe Artefato) que inspecione as informações estruturais do objeto**;
- Capturar os requisitos e informações de conclusão do desenvolvimento do projeto: **criar meta-objetos correspondentes a objetos correspondentes aos**

requisitos de *software* (objetos de classe instanciada da meta-classe Artefato);

- Facilitar a organização do projeto: **criar meta-objetos que monitorem objetos (de classe instanciada da meta-classe Artefato)**.

b.3. Comunicação com gerente de processo: **meta-objetos podem ser instanciados para objetos (de classe instanciada a partir da meta-classe Agente Humano) que possuam o papel de "gerente de processo" com o objetivo específico de enviar informações ao gerente correspondente, por exemplo, por endereço de *e-mail* cadastrado no ambiente**.

b.4 Planos de contingência: **durante a etapa de modelagem pode-se utilizar (em UML) um diagrama de atividades que indicando quais condições ativam objetos que tratem determinadas contingências. Durante a execução do processo, a qualquer momento podem ser criados meta-objetos que monitorem e tratem objetos "críticos" em caso de alguma condição especial**.

c. **Engenheiro de *software*** (analista, programador, testador, documentador, etc.):

c.1. Controlar o *login/logout* de tarefa incumbida: **criar meta-objeto sobre objeto (de classe instanciada da meta-classe Tarefa) relacionada ao engenheiro de *software***;

c.2. Ativar uma ferramenta CASE: **ativação dos objetos que mapeiam as ferramentas CASE disponibilizadas no ambiente**;

c.3. Comunicar-se com outro engenheiro: **meta-objetos podem ser instanciados para objetos (de classe instanciada a partir da meta-classe Agente Humano) que possuam o papel de "engenheiro de *software*" com o objetivo específico de enviar informações ao engenheiro correspondente, por exemplo, por endereço de *e-mail* cadastrado no ambiente**;

c.4. Comunicar-se com o gerente de projeto: **meta-objetos podem ser instanciados para objetos (de classe instanciada a partir da meta-classe Agente Humano) que possuam o papel de "gerente de projeto" com o objetivo específico de enviar informações ao gerente correspondente, por exemplo, por endereço de *e-mail* cadastrado no ambiente**;

c.5. Verificação do cumprimento de prazos: **utilização do WfMS que deverá permitir a inspeção de informações dos objetos (de uma classe instanciada da meta-classe Tarefa)**;

c.6. Controle de versões: **criar meta-objeto que controlem objetos (de classe instanciada da meta-classe Artefato) correspondentes**;

c.7. Controle de uso: **os componentes do nível de aplicação devem prover estas funcionalidades ao engenheiro de *software***.

Para as funcionalidades adicionais, a arquitetura WRAPPER propõe as seguintes alternativas:

a. **Ferramentas CASE**:

a.1. Programação de alto nível: **o componente Ferramenta CASE, no nível de aplicação, desenvolvido para o ambiente ou integrado ao mesmo, com esta finalidade, deve prover esta funcionalidade**;

a.2. Geração automática de código: **o componente Ferramenta CASE, no nível de aplicação, desenvolvido para o ambiente ou integrado ao mesmo, com esta finalidade, deve prover esta funcionalidade**;

a.3. Sincronização entre níveis de especificação: **o componente Ferramenta CASE, no nível de aplicação, desenvolvido para o ambiente ou integrado ao mesmo, com esta finalidade, deve prover esta funcionalidade. Outra alternativa seria criar um meta-objeto sobre o objeto (da classe instanciada da meta-classe Artefato) que intercepte qualquer alteração e ative modificação em outro objeto**;

a.4. Reuso de maior nível: **o componente Ferramenta CASE, no nível de aplicação, desenvolvido para o ambiente ou integrado ao mesmo, com esta finalidade, deve prover esta funcionalidade**;

a.5. Editores cooperativos: **o componente Ferramenta CASE, no nível de aplicação, desenvolvido para o ambiente ou integrado ao mesmo, com esta finalidade, deve prover esta funcionalidade**;

a.6. Suporte a verificação de consistência: **o componente Ferramenta CASE, no nível de aplicação, desenvolvido para o ambiente ou integrado ao mesmo, com esta finalidade, deve prover esta funcionalidade**. **Outra alternativa seria criar um meta-objeto sobre o objeto (da classe instanciada da meta-classe Artefato) que intercepte qualquer alteração e execute ações de verificação de consistência**.

b. **Gerenciamento de ferramentas**:

b.1. Editores cooperativos: **para sincronizar o trabalho entre ferramentas distintas poderiam ser criados meta-objetos correspondentes, que manteriam o sincronismo de informações entre as mesmas**;

b.2. Facilidades para *workgroup*: **para prover suporte ao trabalho conjunto poderiam ser criados meta-objetos que manteriam o sincronismo de informações entre os participantes**;

b.3. Consistência entre ferramentas: **basicamente controlado pelo WfMS. Entretanto, também seria possível criar meta-objetos para objetos (de classes instanciadas da meta-classe Artefato) que mantenham a consistência**.

c. **Gerenciamento de interface com o usuário**:

c.1. Ferramentas para criação da interface visual: **o componente Ferramenta CASE, no nível de aplicação, desenvolvido para o ambiente ou integrado ao mesmo, com esta finalidade, deve prover esta funcionalidade**.

d. **Gerenciamento de configuração**:

d.1. Ambiente genérico com suporte a aplicações de determinados domínios: **o componente Ferramenta CASE, no nível de aplicação, desenvolvido para o ambiente ou integrado ao mesmo, com esta finalidade, deve prover esta funcionalidade**;

d.2. Propagação de mudanças: **o componente Ferramenta CASE, no nível de aplicação, desenvolvido para o ambiente ou integrado ao mesmo, com esta finalidade, deve prover esta funcionalidade. De forma adicional, poderiam ser criados meta-objetos para objetos (de classe instanciada da meta-classe Artefato) que mantivessem a propagação de mudanças nos objetos correspondentes**;

e. **Gerenciamento de requisitos**:

e.1. Reuso de alto nível: **o componente Ferramenta CASE, no nível de aplicação, desenvolvido para o ambiente ou integrado ao mesmo, com esta finalidade, deve prover esta funcionalidade**.

f. **Verificação e validação de projeto**:

f.1. Ferramentas de avaliação: **o componente Ferramenta CASE, no nível de aplicação, desenvolvido para o ambiente ou integrado ao mesmo, com esta finalidade, deve prover esta funcionalidade**.

g. **Gerenciamento de documentação**:

g.1. Reestruturação de um *framework* a partir das aplicações: **o componente Ferramenta CASE, no nível de aplicação, desenvolvido para o ambiente ou integrado ao mesmo, com esta finalidade, deve prover esta funcionalidade**;

g.2. Reuso de maior nível: **o componente Ferramenta CASE, no nível de aplicação, desenvolvido para o ambiente ou integrado ao mesmo, com esta finalidade, deve prover esta funcionalidade**.

h. **Sistema de gerenciamento de objetos**:

h.1. Cópia de um subconjunto do modelo para cada tarefa: **o componente Ferramenta CASE, no nível de aplicação, desenvolvido para o ambiente ou integrado ao mesmo, em conjunto com o Gerente de Nível Base devem permitir este controle**;

h.2. Controle de servidores e clientes: **o Gerente de Nível Base em conjunto com o Gerente de Processo (no nível de aplicação) devem controlar a distribuição. O uso de objetos distribuídos e *middleware* permitem que clientes distribuídos remotamente, sejam integrados ao ambiente**;

h.3. Facilidades de armazenamento e recuperação de componentes, documentação: **o Gerente de Nível Base é o responsável pelo acesso aos dados (objetos) disponíveis no ambiente**.

3.5 Benefícios da arquitetura reflexiva WRAPPER

As seções anteriores apresentaram como o processo é modelado e executado no ambiente proposto. Também foram apresentados os vários componentes da arquitetura reflexiva WRAPPER. Em comparação a arquiteturas de PSEEs existentes baseadas em objetos, tais como: ADELE (BELKHATIR, 1994), E3 (BALDI, 1994), EPOS (NGUYEN, 1997), SPADE (BANDINELLI, 1992), a arquitetura WRAPPER traz os seguintes benefícios:

a. **extração dinâmica de informações sobre a execução do processo e sobre o próprio ambiente de suporte**: usualmente a extração de informações sobre a execução do processo em

um PSEE deve ser planejada durante a especificação do projeto, de forma que em determinados pontos de controle as informações possam ser coletadas e verificadas. Na arquitetura WRAPPER, como resultado da aplicação de reflexão computacional no desenvolvimento de *software* (seção 2.4.1) é possível que informações sobre a execução do processo, ou mesmo de qualquer outro componente da arquitetura, possam ser extraídas a qualquer momento. Para tal, basta definir um meta-objeto associado ao objeto correspondente ao componente desejado, seja ele uma ferramenta CASE, seja uma determinada tarefa de um projeto qualquer.

b. **alteração dinâmica do processo**: diversos PSEEs prevêem como requisito básico a adaptação do processo. Entretanto esta adaptação é normalmente realizada através da alteração da especificação do processo em sua PML que pode então ser re-interpretada ou recompilada pelo ambiente de suporte. Esta é abordagem utilizada nos ambientes SPADE e EPOS cujas respectivas PMLs, SLANG (BANDINELLI, 1993) e SPELL (NGUYEN, 1997), permitem a adaptação do processo pela alteração e re-interpretação da especificação do processo. Na arquitetura WRAPPER, também como resultado da reflexão computacional, é possível alterar um processo em execução de forma dinâmica sem alterar a especificação do processo. Através da criação de meta-objetos associados aos elementos envolvidos em um determinado processo é possível, por exemplo:

   - adicionar funcionalidade a uma tarefa: o meta-objeto intercepta mensagens ao objeto (tarefa) correspondente, executa computações adicionais e depois repassa a chamada ao objeto (tarefa) para a execução normal.
   - alterar o fluxo de execução do processo: meta-objetos podem ser inspecionar determinados objetos (tarefas) de forma que interceptem as mensagens enviadas e redirecione-as para outros objetos (tarefas).
   - alterar execução de ferramentas: através de meta-objetos é possível redirecionar a ativação de uma ferramenta CASE para outra ferramenta que pode ter sido integrada ao ambiente recentemente, sem ter que alterar a especificação do projeto de *software* em andamento.

c. **distribuição na Web**: alguns PSEEs prevêem a distribuição do ambiente em redes locais, mas poucos explicitam o uso da WWW como plataforma básica para a distribuição do ambiente. A arquitetura proposta usa a Web e tecnologias disponíveis para a mesma como plataforma de desenvolvimento, o que permite que o ambiente seja distribuído e também executado em ambientes heterogêneos.

d. **integração de ferramentas heterogêneas**: A inclusão de ferramentas diversas em um ambiente integrado é um dos desafios na construção de um ambiente integrado (SHARON, 1995; THOMAS, 1992). A arquitetura WRAPPER prevê a integração de ferramentas CASE de fornecedores e plataformas diversas. O uso de objetos distribuídos e tecnologias Web possibilita esta integração. Para ferramentas CASE proprietárias, o ambiente gera um objeto-*proxy* que permite a ativação da mesma seja pelo cliente local, como pelo servidor remoto utilizando-se dos protocolos disponíveis na Web. Para ferramentas abertas, pode-se criar interfaces de acesso as suas funcionalidades internas que poderão ser ativadas tanto localmente, quanto remotamente, também se fazendo uso de objetos remotos e protocolos da Web.

Observa-se que alguns PSEEs possuem também alguns dos benefícios citados, utilizando outras soluções, mas nenhum PSEE possui, de forma integrada, as características de uso de reflexão computacional, distribuição na Web e integração de ferramentas heterogêneas que a arquitetura WRAPPER propõe.

3.6 Adaptações para implementação da arquitetura WRAPPER

A arquitetura WRAPPER foi primariamente planejada para ser desenvolvida sobre CORBA (BEN-NATAN, 1998; OMG, 2002), pois a arquitetura CORBA possui características interessantes para a construção de um PSEE buscando-se satisfazer os requisitos de reflexão computacional, distribuição e heterogeneidade.

O protótipo da arquitetura implementado e descrito no capítulo seguinte está desenvolvido sobre CORBA pois esta arquitetura possui as seguintes características:

a. disponibiliza um ORB que permite o registro de objetos que ficam disponíveis para acesso remoto;

b. possui um repositório de interfaces que permite o armazenamento e disponibilização de especificações de interfaces dos objetos registrados no ORB;

c. possui interceptadores que permitem a interceptação de mensagens entre os objetos do ORB;

d. define padrões para comunicação entre ORBs de fornecedores diversos, permitindo que objetos heterogêneos (implementados em linguagens e plataformas diversas) consigam trocar mensagens.

Entretanto, seria possível adaptar a implementação da arquitetura para outras tecnologias. Os requisitos básicos para a implementação da arquitetura seriam:

a. existência de algum mecanismo de captura de informação estrutural dos objetos do ambiente (reflexão estrutural);
b. disponibilidade de algum mecanismo de interceptação de mensagens entre objetos do ambiente (reflexão comportamental);
c. os objetos devem ser capazes de trocar mensagens entre si remotamente, portanto deve haver algum *middleware* para permitir esta comunicação;
d. os objetos podem ser implementados de forma heterogênea (em linguagens de programação diversas e plataformas de execução diversas).

A linguagem Java e seu ambiente de desenvolvimento (SUN, 2002a), seria uma alternativa de tecnologia possível para a implementação da arquitetura WRAPPER. Em sua biblioteca de classes, Java possui o pacote java.lang.reflect que disponibiliza uma série de classes que permitem obter informação, bem como interceptar mensagens, dos objetos através de reflexão computacional. O uso de RMI (*Remote Method Invocation*) permite que objetos distribuídos troquem mensagens entre si. A única ressalva é que todos os objetos seriam, obviamente, objetos Java, não sendo totalmente heterogêneos. Ainda com a utilização de tecnologia Java outra alternativa, dentro da arquitetura J2EE (KURNIAWAN, 2002) seria a utilização de EJBs (*Enterprise Java Beans*) que permite a implementação de objetos com acesso remoto.

Uma outra alternativa possível seria utilizar DCOM (*Distributed Component Object Model*; MICROSOFT, 2002; COSTA, 2000b). A arquitetura DCOM é uma extensão do modelo COM (*Component Object Model*) da Microsoft. A tecnologia COM define componentes e como eles são manipulados, enquanto DCOM estende esta tecnologia para o suporte a objetos distribuídos em computadores remotos. Uma limitação estaria na heterogeneidade de plataforma, uma vez que aplicações DCOM são baseadas apenas em plataforma Microsoft.

4 IMPLEMENTAÇÃO DO PROTÓTIPO

Este capítulo descreve a implementação de um protótipo da arquitetura WRAPPER para atender aos objetivos iniciais propostos por esta arquitetura.

Inicialmente são introduzidos alguns conceitos básicos das tecnologias utilizadas para a implementação do protótipo. Em seguida, é apresentada a estrutura do protótipo desenvolvido e alguns cenários de execução e utilização do mesmo.

4.1 Objetos distribuídos em CORBA

CORBA (*Common Object Request Broker Architecture*) (BEN-NATAN, 1995; BEN-NATAN, 1998; HOQUE, 1998; ORFALI, 1998) é um *middleware* que permite a comunicação entre objetos distribuídos e heterogêneos, ou seja, objetos localizados e implementados em diferentes linguagens de programação e em diferentes plataformas computacionais.

A especificação CORBA foi introduzida em 1991 pela OMG (*Object Management Group*) (OMG, 2002b), um consórcio de diversas empresas envolvidas no desenvolvimento de sistemas orientados a objetos e distribuídos. As especificações CORBA, que visam a interoperabilidade entre sistemas orientados a objetos distribuídos, são publicadas como documentos OMG (OMG, 2002; BEN-NATAN, 1995).

Pela especificação CORBA, um objeto cliente, usando um ORB (*Object Request Broker*), transparentemente invoca um método de um objeto servidor, que pode estar na mesma máquina que o cliente ou em algum local remoto. O ORB recebe a chamada e é responsável por localizar um objeto servidor que implemente o método, por passar os parâmetros necessários, receber o resultado e retorná-lo ao objeto cliente. Um objeto cliente não necessita saber qual a linguagem de implementação, sistema operacional ou outros aspectos que não fazem parte da interface do objeto servidor. A especificação CORBA especifica que um objeto pode ser cliente, servidor, ou ambos, dependendo da aplicação.

Abaixo são apresentados os conceitos chaves de CORBA:

a. Qualquer objeto (ou aplicação) pode ser um cliente, servidor ou ambos. Para efeito de descrição, CORBA usa o modelo Cliente/Servidor onde objetos (clientes) solicitam serviços a outros objetos (servidores);

b. Qualquer interação entre objetos é realizada através de requisições. A informação associada a uma requisição é uma operação a ser executada, o objeto destino, zero ou mais parâmetros, e outras informações adicionais;

c. CORBA provê suporte à ligação estática e dinâmica. Ligação estática é usada para identificar objetos em tempo de compilação, enquanto a ligação dinâmica entre objetos usa a identificação, em tempo de execução, dos objetos e seus parâmetros;

d. Uma interface representa contratos entre aplicações clientes e servidores. Uma típica definição de interface define os parâmetros a serem utilizados para permitir a comunicação entre cliente e servidor. CORBA propõe uma linguagem de descrição de interface, a IDL (*Interface Definition Language*). *Stubs* para clientes e *skeletons* para o servidor são produzidos como parte da compilação IDL;

e. O protocolo de comunicação IIOP (*Internet Inter-ORB Protocol*), que é baseado sobre o protocolo TCP/IP (*Transmission Control Protocol/Internet Protocol*), permite a comunicação entre ORBs localizados na WWW;

f. Objetos CORBA não conhecem os detalhes de implementação, um objeto adaptador mapeia o modelo genérico para a implementação e é o meio primário pelo qual a implementação de um objeto acessa os serviços providos pelo ORB.

g. A arquitetura conceitual de CORBA (HOQUE, 1998; OMG, 2002) é apresentada na Figura 4.1. Esta arquitetura é constituída pelos seguintes elementos:

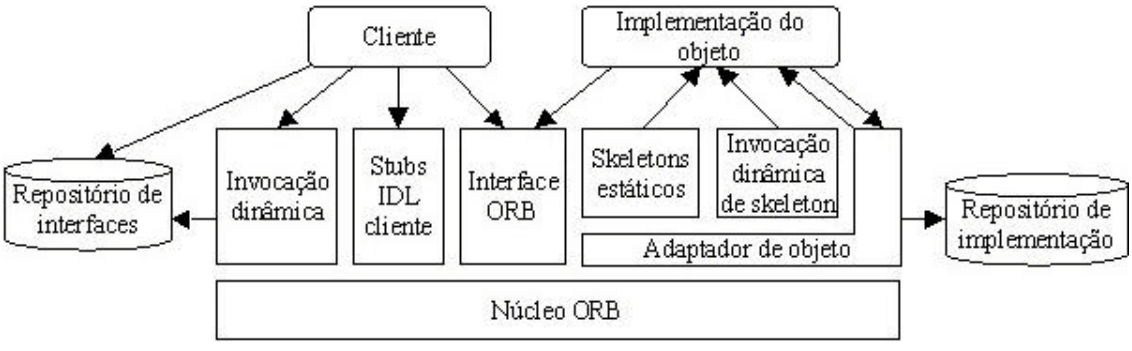

Figura 4.1: Visão conceitual da arquitetura de CORBA

- **Cliente**: um objeto ou aplicação que solicita a execução de um método de um objeto servidor. O cliente acessa o ORB e realiza a requisição passando as informações necessárias ao ORB;
- ***Stub* IDL cliente**: é utilizada para a execução de uma requisição de forma estática. O *stub* é gerado pela compilação da especificação IDL do objeto servidor. Cada *stub* é gerado na linguagem de programação desejada, e atua como um objeto *proxy* para acessar o objeto servidor. Para o objeto cliente o *proxy* atua como se a requisição ao método fosse local. O *stub* é responsável por realizar a operação de *marshaling* (codificação da requisição em uma mensagem simples que pode ser enviada ao servidor);
- **Interface de invocação dinâmica**: permite que um cliente descubra métodos em tempo de execução. Desta forma, um cliente pode descobrir a interface de um objeto servidor, mesmo que não possua um *stub* correspondente;
- **Repositório de interfaces**: atua como um banco de dados distribuído que possui as descrições IDL de objetos servidores. Este repositório possui meta-dados sobre objetos para ORBs;
- **Interface ORB**: é constituída de uma API (*Application Programming Interface*) que permite acesso a operações do ORB, como por exemplo: inicialização do ORB, transformação de uma referência de objeto em *string*, etc.;
- **Implementação de objeto**: representa a implementação do objeto servidor em alguma linguagem de programação específica;
- ***Skeleton* estático**: atua como um *proxy* para a implementação do objeto servidor. O *skeleton* é utilizado para requisições estáticas solicitadas pelo cliente e é gerado pela compilação da especificação IDL do objeto servidor;
- **Invocação de *skeleton* dinâmico**: permite a execução de um método do servidor e mesmo criar implementações de objetos, em tempo de execução;
- **Adaptador de objeto**: é o responsável pela instanciação de objetos servidores, pela passagem de requisições a eles, bem como a definição de identificadores aos mesmos. O adaptador de objetos também é responsável pelo controle das classes, e seus objetos, que são registrados no repositório de implementação;
- **Repositório de implementação**: é o local que armazena definições de classes que o ORB dá suporte, seus objetos e seus identificadores. Armazena também informações adicionais sobre a implementação do ORB.

Além da estrutura básica da arquitetura, a especificação CORBA também define serviços especiais a serem providos por um ORB, que são chamados de “Serviços CORBA” e "Facilidades CORBA". Atualmente, existe a definição de diversos serviços CORBA que possuem finalidades diversas. Por exemplo, o “Serviço de Nomes” (*Naming Service*) permite a localização de um objeto pelo seu nome, o objeto é organizado dentro de um contexto de nomes (semelhante a uma estrutura de diretórios) permitindo que outros objetos possam localizá-lo pelo seu nome. Facilidades CORBA oferecem serviços específicos para domínios de aplicação, como aplicações médicas e gerência de documentos.

A implementação de um ORB e seus respectivos serviços é dependente do fabricante de *software*. Existem, atualmente, diversas implementações de ORBs comerciais e acadêmicos (BORLAND, 2002; OMG, 2002). Para permitir a interconexão entre objetos de um ORBs de fornecedores diferentes, a especificação CORBA definiu o protocolo GIOP (*General Inter-ORB Protocol*). O GIOP define um conjunto de formatos de mensagens e representação de dados comuns para a comunicação entre ORBs. O IIOP (*Internet Inter-ORB Protocol*) é uma especialização do GIOP que define como as mensagens devem ser trocadas sobre uma rede com o protocolo TCP/IP, permitindo desta forma o uso da Internet como plataforma para a troca de mensagens entre ORBs diferentes.

Além de CORBA, existem outras alternativas para o desenvolvimento de aplicações distribuídas, tais como: DCOM (*Distributed Component Object Model*) da Microsoft (MICROSOFT, 2002), RMI (*Remote Method Invocation*) de Java, CGI (*Common Gateway Interface*) e RPC (*Remote Procedure Call*). Entretanto, apesar de ser considerado mais complexo que as outras alternativas, CORBA possui as seguintes vantagens (HOQUE, 1998; ORFALI, 1998; BEN-NATAN, 1997):

- é independente de plataforma, de linguagem de programação e de fabricantes de *software* e *hardware*;
- oferece suporte ao desenvolvimento de aplicações baseadas em objetos distribuídos;
- os parâmetros passados na chamada de um método em um objeto servidor podem ser de vários tipos e não apenas de *string* (tipo usualmente utilizado);
- a arquitetura CORBA é mais escalonável, permitindo o crescimento das aplicações e balanceamento de carga entre servidores;
- permite a integração com sistemas já existentes, mesmo não sendo orientados a objetos;
- com o uso de IIOP é possível utilizar a Internet para a comunicação entre os objetos distribuídos;
- tem suporte para diversas linguagens de programação e ORBs abertos ou comerciais;

- aliada a Java e a Internet, CORBA permite o desenvolvimento de aplicações distribuídas que podem ser executadas dentro de um navegador Web sem a instalação de um ORB em um cliente;
- provê facilidades que podem ser exploradas para a reflexão computacional.

Apesar das diversas vantagens trazidas por CORBA, existem algumas limitações:

- atualmente nenhum ORB oferece todos os serviços CORBA previstos na sua especificação. Entretanto, o presente trabalho apoia-se no uso da arquitetura básica sem a necessidade de algum serviço CORBA especial;
- a comunicação no ORB é basicamente através de mensagens síncronas, o que pode provocar queda no desempenho de execução do sistema. Neste trabalho, o desempenho do sistema não é um requisito primário, entretanto, alternativas para minimizar o problema de desempenho são previstas.

4.1.1 Reflexão computacional em CORBA

Em termos de reflexão computacional, CORBA provê mecanismos básicos que podem ser explorados para obter reflexão computacional.

Em relação à reflexão estrutural, CORBA define que os objetos registrados no ORB devem ter sua interface definida em uma especificação IDL. As especificações IDL são armazenadas no Repositório de Interfaces da arquitetura.

Para a manipulação das especificações IDL no Repositório de Interfaces é disponibilizada uma API que permite descobrir a interface de objetos registrados no ORB. Este é o mecanismo utilizado para a interface de invocação dinâmica, na qual um objeto cliente pode descobrir os serviços e outras características providas pela IDL do objeto servidor. Desta forma, o uso das especificações IDL no Repositório de Interfaces permite que um meta-objeto obtenha a interface de um objeto de nível base.

De forma adicional, CORBA provê uma característica especial que pode ser explorada pela reflexão computacional: interceptadores (OMG, 2002a). Através de interceptadores é possível interceptar as mensagens entre objetos do ORB (Figura 4.2).

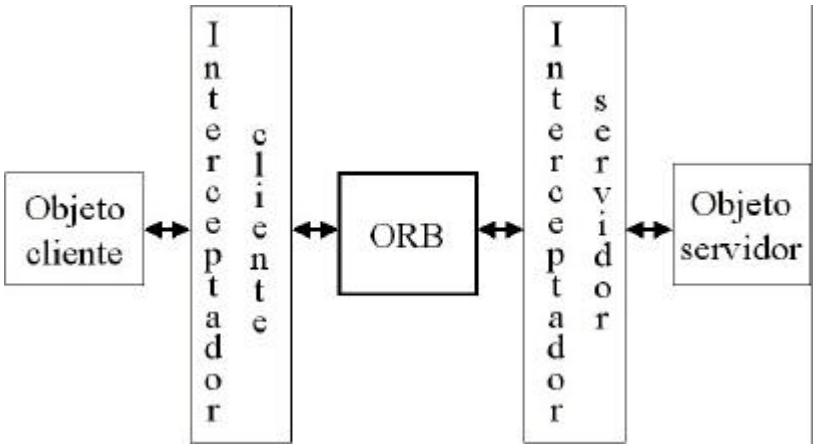

Figura 4.2: Interceptadores entre objetos do ORB

Interceptadores foram definidos para prover um meio de realizar extensões a um ORB. Usando interceptadores é possível modificar o ORB, chamando novas funcionalidades (tais como serviços de autenticação e criptografia) durante o processo de envio e/ou recebimento de mensagens.

Interceptadores são registrados no ORB e podem ser posicionados no lado do cliente, do servidor ou em ambos.

Quando o ORB captura uma requisição (mensagem) um interceptador é ativado. Este interceptador pode analisar o conteúdo da mensagem e realizar computações adicionais antes de passar a mensagem adiante.

Do ponto de vista de reflexão computacional, interceptadores podem ser utilizados para prover reflexão comportamental. Através dos interceptadores as mensagens entre os objetos registrados no ORB podem ser capturadas e um meta-objeto associado é ativado. Neste momento o meta-objeto pode inspecionar objetos e executar computação adicional antes de passar a mensagem adiante. Na realidade, dependendo do objetivo do meta-objeto, é possível que a mensagem nem seja passada ao objeto-destino, sendo transferida para um outro objeto; sendo possível mesmo retornar uma resposta ao objeto que enviou a mensagem sem mesmo ter ativado o objeto-destino.

Para o contexto do protótipo, CORBA foi escolhido pois a arquitetura disponibiliza os componentes necessários para o suporte das características básicas desejadas: uso de objetos, reflexão computacional, distribuição e heterogeneidade.

4.2 Java

Apesar das linguagens de *script* permitirem maior interação nas páginas *Web*, ainda assim são limitadas para o desenvolvimento de sistemas mais complexos. Desta forma, os *scripts* no cliente não tinham a capacidade de executar tarefas mais complicadas, e os *scripts* do servidor tendiam a

sobrecarregar o mesmo. A solução seria permitir que parte do processamento mais complexo de uma aplicação pudesse ser executada no cliente, porém transmitida pelo servidor *Web*.

A linguagem de programação Java veio solucionar este problema, em particular os *applets* Java (SUN, 2002a; WALSH, 1998), que são aplicativos Java. Os *applets* são transmitidos do servidor *Web* para o *browser* do cliente e, então, executados no mesmo.

Java, na realidade, é uma linguagem de programação que permite a criação de aplicativos "normais", a serem executados em alguma plataforma específica, bem como de *applets* que são executados em uma máquina virtual Java (JVM – *Java Virtual Machine*) que está implementada na maioria dos *browsers*.

Dentre as características importantes desta linguagem para o desenvolvimento de aplicativos estão:

a. **simplicidade**: provê a simplicidade de uma linguagem de programação (sua sintaxe é parecida com C++) sem características daninhas como manipulação de apontadores, conversão implícita de tipos, manipulação de memória, etc.;

b. **orientação da objetos**: provê suporte aos conceitos de orientação a objetos, de forma a permitir a estruturação de *software* segundo metodologias orientadas a objetos;

c. **portabilidade**: programas Java são traduzidos para um código intermediário (chamado de *bytecode*) que é interpretado pelo JVM implementado em diversos *browsers* e plataformas de computadores. Além disso a linguagem provê uma biblioteca padrão para a construção de interfaces gráficas, o AWT (*Abstract Window Toolkit*);

d. **distribuição**: esta linguagem foi projetada tanto para a distribuição de dados quanto de processamento. Para facilitar a distribuição de dados existem mecanismos de acesso a arquivos, como o JFS (*Java File* System) e mesmo a bancos de dados remotos, como o JDBC (*Java DataBase Connectivity*). A distribuição de processamento é realizada pelo próprio conceito de *applet* (que é transmitido do servidor para o cliente) ou de *servlet* (que permanece no servidor), mas também pelos mecanismos de RMI (*Remote Method Invocation*) ou de *sockets* que permitem a comunicação com objetos remotos;

e. **robustez**: o projeto da linguagem prevê mecanismos como tipagem forte e verificação estática de correção. Além disso, em tempo de execução existem verificação de correção de ligação válidas, índices, etc., além de existirem mecanismos de tratamento de exceção;

f. **segurança**: o ambiente distribuído traz inerentemente o problema de segurança na troca de informações e no processamento. O JVM executa diversas verificações já na

carga do *applet* e limita o mesmo pelo conceito de *sandbox* (“caixa de areia”), na qual um *applet* remoto não possui direito de realizar ações daninhas ao cliente, como por exemplo: alterar o sistema de arquivos, execução de programas locais, etc.;

g. **interpretação**: o *bytecode* correspondente a um programa Java pode ser interpretado por JVM implementado em diversas plataformas e *browsers*. Além disso no próprio *bytecode* existem informações a respeito das classes implementadas que permitem sua depuração a qualquer instante;

h. **rapidez**: o compilador Java busca gerar um *bytecode* otimizado a fim de acelerar sua execução. Existem mecanismos como o JIT (*Just In-Time compiler*) que buscam gerar código executável específico para cada plataforma;

i. **paralelismo**: Java provê suporte ao conceito de *threads* que permite a execução de processos concorrentes, bem como de monitores para a sincronização destes processos;

j. **dinamismo**: Java está baseada na *Web,* que atualmente é um dos ambientes mais dinâmicos, por conseqüência, deve também dar suporte a tais mudanças. Além disso o projeto da linguagem prevê a ligação de componentes dinamicamente, de forma que a troca de partes de um sistema não necessita de recompilação.

Com todas estas características apresentadas, Java tornou-se a maior “aposta” dos desenvolvedores para implantação de sistemas baseados na *Web*.

Historicamente, a linguagem Java surgiu da evolução da linguagem Oak, criada em 1991 por James Gosling para a implementação de dispositivos inteligentes (projeto Green). Em 1994, alguns anos após a empresa Sun Microsystems incorporar o projeto Green, a linguagem Oak foi rebatizada para Java.

Em 1996 é liberada a versão 1.0 de Java em um kit de desenvolvimento (com ferramentas e bibliotecas) oferecido pela Sun, o *Java Development Kit* (JDK).

Além do JDK da Sun, outras empresas lançaram produtos com dialetos Java, como o Visual J++ da Microsoft e o JBuilder da Borland. Tais “dialetos” surgiram do problema do JDK: a falta de ferramentas apropriadas para o rápido desenvolvimento de aplicativos. Entretanto, isso tem criado o problema do aparecimento de extensões que não fazem parte do padrão *de facto* criado pela Sun. Em função disto, está em elaboração a especificação de uma linguagem Java ANSI padronizada, da qual diversos fabricantes de produtos Java estão envolvidos.

Apesar de Java ser considerada uma promessa para o desenvolvimento de aplicações *Web* existem alguns problemas: *applets* são interpretados, o que implica em desempenho bem inferior a aplicações compiladas; diversos *browsers Web* não têm suporte a Java; a linguagem Java da Sun também está em evolução, a versão atual é o Java 2 versão 1.4.1.

Diante destas limitações alguns desenvolvedores estão evitando Java como solução de implementação e utilizando linguagens de *script* ou mesmo soluções proprietárias.

No contexto do protótipo, Java foi escolhida para permitir a implementação da arquitetura, especialmente no lado do cliente, baseada em *applets* embutidas em páginas HTML executando em *browsers Web*. Java pode ser combinada com CORBA de forma que o ORB pode ser carregada juntamente com a *applet*, evitando a instalação de um ORB no cliente.

Esta estratégia permite que no lado do cliente não seja necessário a implantação de *software* especial, bastando, obviamente, um *browser Web* com suporte a *applets* e acesso à Internet. Como conseqüência, o lado do cliente pode ser executado em plataformas heterogêneas.

4.3 Protótipo implementado

Para a demonstração da arquitetura proposta, um protótipo foi implementado. Este protótipo foi desenvolvido em duas frentes: a composição do nível base e do meta-nível, e um ADS de suporte a processo baseado em objetos, *workflow* e documentos hipertexto.

A implementação do protótipo foi realizada utilizando-se a linguagem de programação Java (SUN, 2002) sobre o ORB CORBA Borland Visibroker, versão 4.5 (BORLAND, 2002). A plataforma de desenvolvimento no servidor é o Windows 2000 Server (MICROSOFT, 2002b) que provê serviços básicos para o ORB CORBA.

O ADS foi desenvolvido em Java e utiliza o Web Server IIS (*Internet Information Services*) para prover acessos a páginas HTML.

O lado do cliente é executado através de navegadores Web, como Netscape Navigator (NETSCAPE, 2002) e Internet Explorer (MICROSOFT, 2002a) que dêem suporte a execução de *applets* Java (Figura 4.3). A interface ao usuário é provida pelas *applets* embutidas em páginas HTML.

As seções seguintes apresentam características da implementação do protótipo desenvolvido.

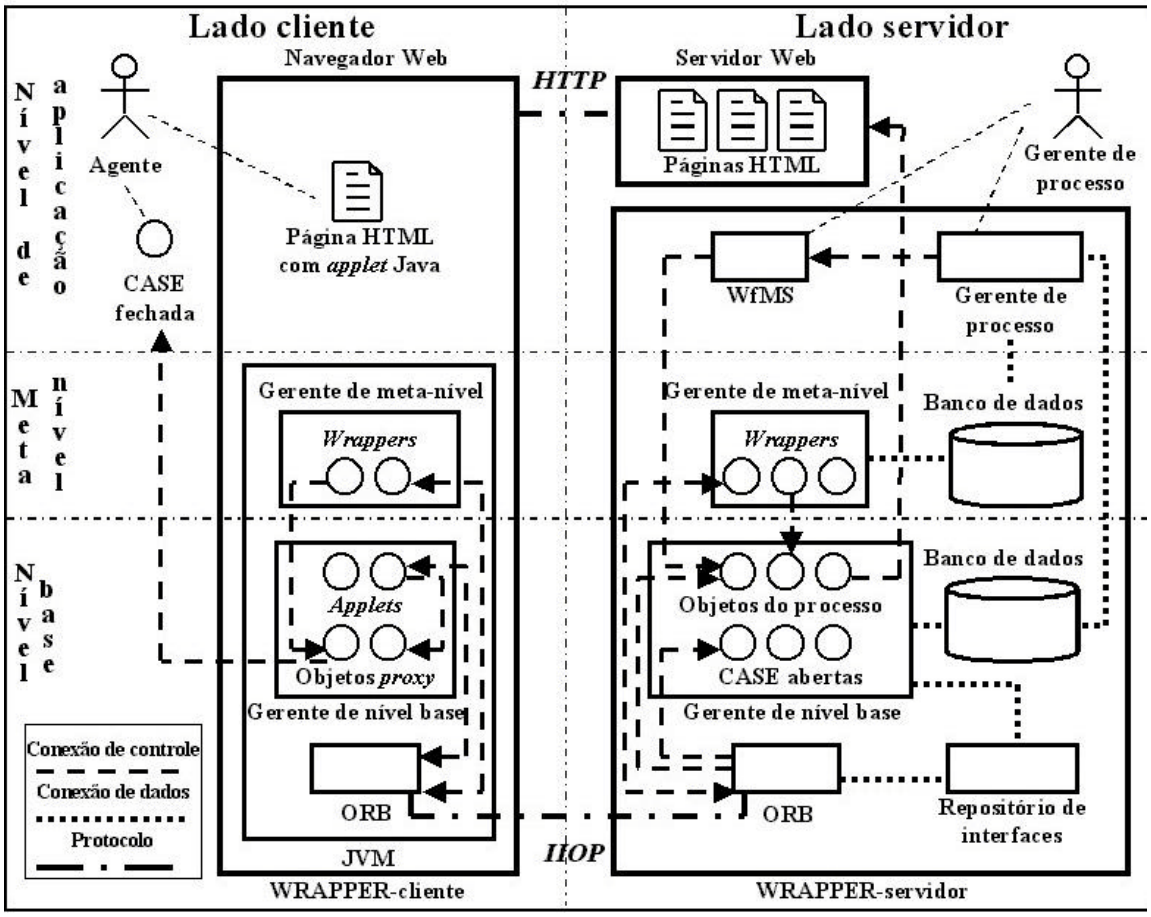

Figura 4.3: Arquitetura do protótipo WRAPPER

4.3.1 Nível base

O nível base da arquitetura é implementado em Java sobre o ORB CORBA provido pelo Borland Visibroker. O Visibroker provê facilidades de acesso a um Repositório de Interfaces que armazena definições IDLs para objetos no ORB.

O gerente de nível base é um objeto servidor implementado como uma aplicação Java. Este gerente é um objeto CORBA que pode ser chamado por qualquer outro objeto através de sua interface IDL. Ferramentas com interface gráfica foram implementadas para controlar este gerente.

Este gerente disponibiliza serviços para o controle de objetos registrados no ORB, bem como controle de armazenamento de objetos persistentes. Atualmente, este armazenamento é feito através de serialização de objetos.

Objetos abertos são registrados no ORB CORBA e sua especificação IDL correspondente é armazenada no Repositório de Interface. Um exemplo é apresentado no Quadro 4.1. Para registrar um objeto fechado, primeiramente é criado um objeto-*proxy* que referencia o objeto fechado, e

então o objeto-*proxy* é registrado no ORB. A IDL do objeto-*proxy* também é registrada no Repositório de Interface.

```
//=================== BASE LEVEL ========================
module baseLevel  // Object classes of the base level
{
  interface Artifact : BasicObject // An artifact
  {
    void setArtifact(in long artifactId, in string name, in long type);
    long getType();
};

typedef sequence< Artifact > ArtifactList;  // List of artifacts

interface ArtifactManager : BasicObjectManager // Artifacts list
  {
    void addArtifact(in long artifactId, in string name, in long type);
    Artifact getArtifact(in long artifactId);
    ArtifactList getAllArtifacts();
  };

interface BaseLevelManager : BasicObjectManager // Manager of the base
level
  {
    void connectAllManagers();
    ObjectList getData(in long GetTypeCode);
    Object getInterfaceDef(in Object anObject);
  };

};  // module baseLevel
```

Quadro 4.1: IDL parcial para o nível base.

A descrição acima representa a visão do lado do servidor da arquitetura. O lado do cliente, atualmente implementado através de acesso por navegador Web, também possui um gerente de nível base, porém com algumas limitações. Uma limitação é que o lado do cliente, nesta situação, não

possui um Repositório de Interface próprio. Desta forma, mesmo que um objeto seja registrado no lado do cliente, sua IDL será registrado no Repositório de Interface do servidor, mantendo a consistência de descrições IDL em apenas um local.

4.3.2 Meta-nível

O meta-nível também está implementado sobre o Visibroker e o gerente de meta-nível está implementado como uma aplicação Java.

Meta-objetos são implementados por meio de interceptadores especiais do Visibroker: os *wrappers*.

Wrapper é um tipo especial de interceptador CORBA que permite a interceptação, se necessário, de apenas um método de um objeto.

De forma semelhante ao funcionamento de um interceptador, um *wrapper* é ativado quando o ORB detecta uma requisição para um objeto ou método no nível base que tenha um *wrapper* associado.

Um *wrapper* pode executar computação adicional antes ou depois de passar a requisição adiante. Um *wrapper* pode também executar sua computação e retornar uma resposta sem chamar o objeto servidor, sem que o objeto cliente tenha conhecimento.

Wrappers podem ser inseridos do lado do cliente, do lado do servidor ou em ambos os lados. Além disso, é possível criar listas de *wrappers* (Figura 4.4) que permitem que *wrappers* sejam encadeados, e executem funções específicas.

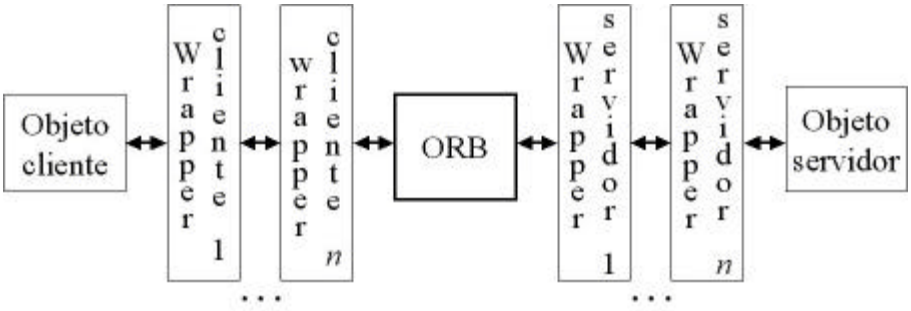

Figura 4.4: *Wrappers* em um ORB

Na arquitetura, os *wrappers* são definidos e controlados pelo gerente de meta-nível no ORB servidor.

O gerente de meta-nível está implementado por dois componentes: o gerente de *wrappers* e serviços-*wrapper*. O gerente de *wrappers* é responsável pela criação e controle de *wrappers* na arquitetura. Serviços-*wrappers* provêem serviços básicos que um *wrapper* pode executar.

Durante a criação de um *wrapper* define-se qual é o protocolo MOP de associação entre o novo *wrapper* e o nível base. De forma adicional é preciso definir qual sua função, como por exemplo, coleta de dados de um objeto de nível base. Um serviço-*wrapper* correspondente é definido e instanciado para cada *wrapper*. Para facilitar seu uso, ferramentas com interface gráfica são disponibilizadas ao usuário.

4.3.3 Nível de aplicação

O nível de aplicação, o ADS, está implementado através de aplicações e *applets* Java para a gerência e controle de processos e projetos de *software*.

O componente de gerência de processo é uma aplicação que permite a gerência de modelos de meta-processos, bem como o controle de dados de projetos. Este gerente basicamente permite ao usuário (uma pessoa no papel de gerente de processo) adicionar ou remover definições de classes do diagrama de classes, que representa o modelo de um determinado meta-processo.

O gerente de projeto é implementado como um WfMS. O WfMS permite que o usuário (uma pessoa no papel de gerente de processo ou de projeto) instanciar um diagrama de objetos correspondente a um projeto, a partir de um diagrama de classes do meta-processo correspondente. De forma adicional, também é definido o *workflow* correspondente e os objetos relacionados, através de um diagrama de atividades.

O WfMS pode executar, passo a passo, o diagrama de atividades que invoca os objetos correspondentes envolvidos no *workflow*.

O gerente de processo e de projeto possuem comunicação com os níveis inferiores da arquitetura para a definição e controle de objetos e meta-objetos.

4.4 Exemplos

No lado cliente, o ADS é acessado pelo usuário (um agente) através de um navegador Web. Inicialmente o usuário deve executar o *log in* no ambiente (Figura 4.5). Este procedimento é necessário para o sistema identificar o usuário e de acordo com o seu papel disponibilizar funcionalidades diferenciadas.

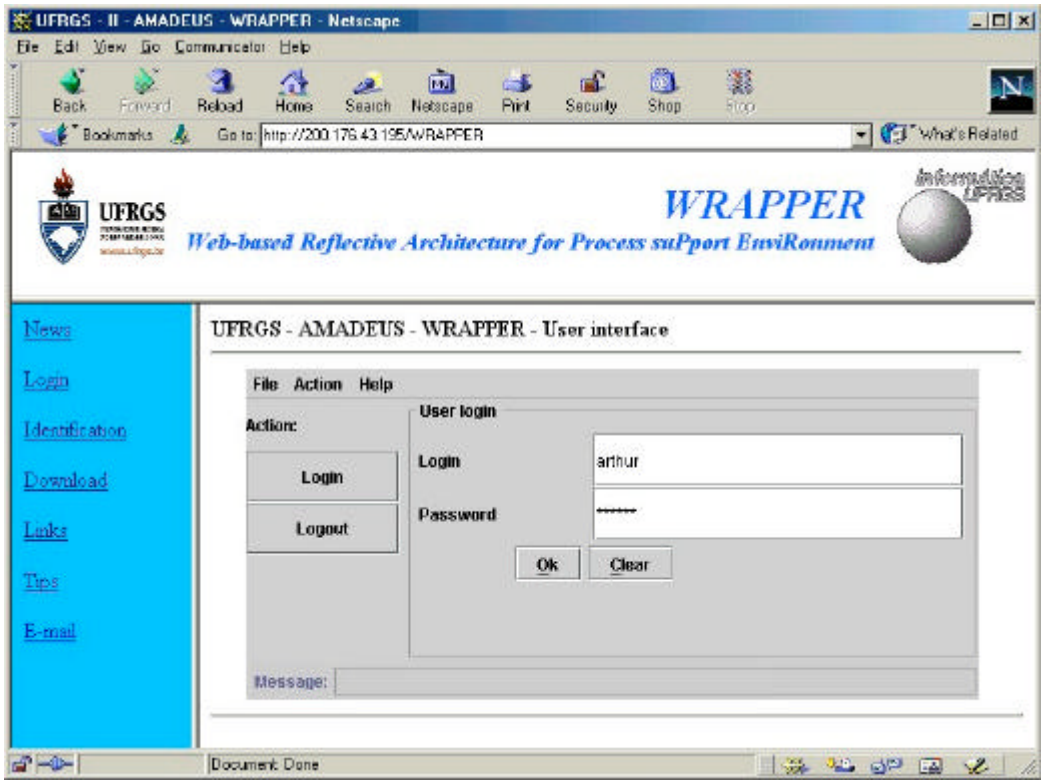

Figura 4.5: *Login* no ambiente

Após o *log in*, de acordo com o seu perfil, uma nova tela é apresentada apresentando uma lista de tarefas a serem executadas, bem como a possibilidade de execução de outras funcionalidades. No exemplo (Figura 4.6), um analista de sistemas recebeu duas tarefas para analisar casos de uso.

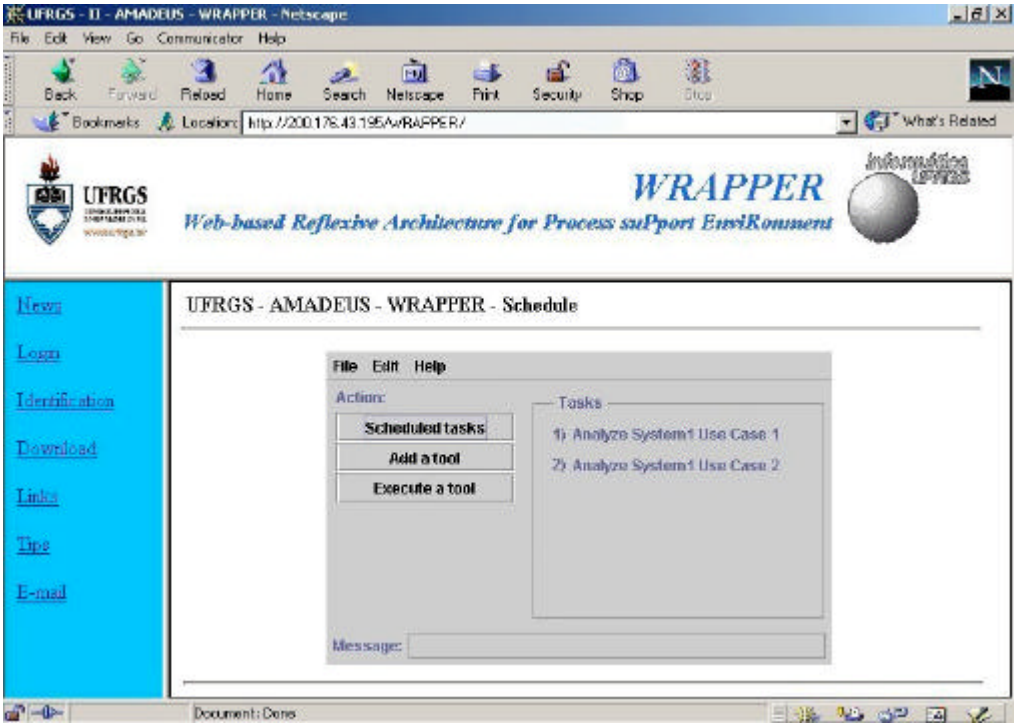

Figura 4.6: Tarefas a serem executadas

Usuários com papel de gerente de processo ou gerente de projeto podem acessar ferramentas para o controle do *workflow* do projeto.

Os cenários a seguir exemplificam o uso de meta-objetos (*wrappers*) para a extração de informação sobre a execução do projeto, e para a alteração desta execução.

4.4.1 Cenário 1: dados estatísticos

Neste exemplo, o gerente de projeto define um meta-objeto sobre um agente (*Arthur*), coletando informações sobre seus *log-ins* no ADS e gravando as informações em um arquivo de *log*.

Para realizar esta função, quando o projeto já está instanciado (diagramas de objetos e de atividades definidos), o gerente de projeto acessa a ferramenta de *workflow* e indica que deseja inserir um novo *wrapper* (Figura 4.7). Após escolher o objeto desejado, uma tela do serviços-*wrapper* é apresentado para escolher a ação desejada para o *wrapper*.

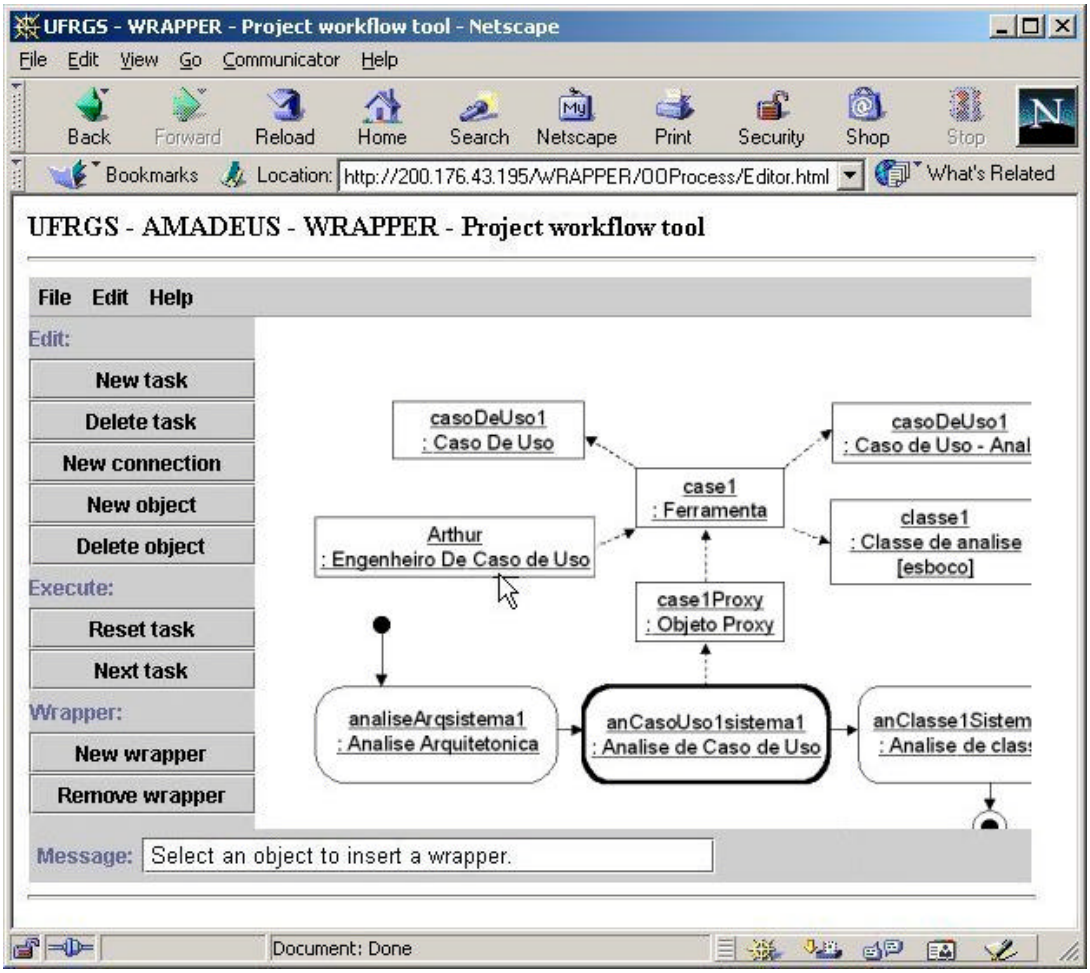

Figura 4.7: Ferramenta de *workflow*

Os serviços-*wrappers* disponíveis no protótipo são (Figura 4.8):

- ***Write data in log file***: informações do objeto são gravadas em um arquivo de *log* (que deve ser definido em outra tela);
- ***Write data in database***: ao invés de um simples arquivo, os dados podem ser gravados em um banco de dados (uma ferramenta específica é aberta para permitir ao usuário selecionar os dados do objeto para serem gravados em um banco de dados específico);
- ***Call an object***: permite que outro objeto registrado no ORB seja executado. É apresentada uma tela que permite que o usuário selecione um outro objeto do ambiente;

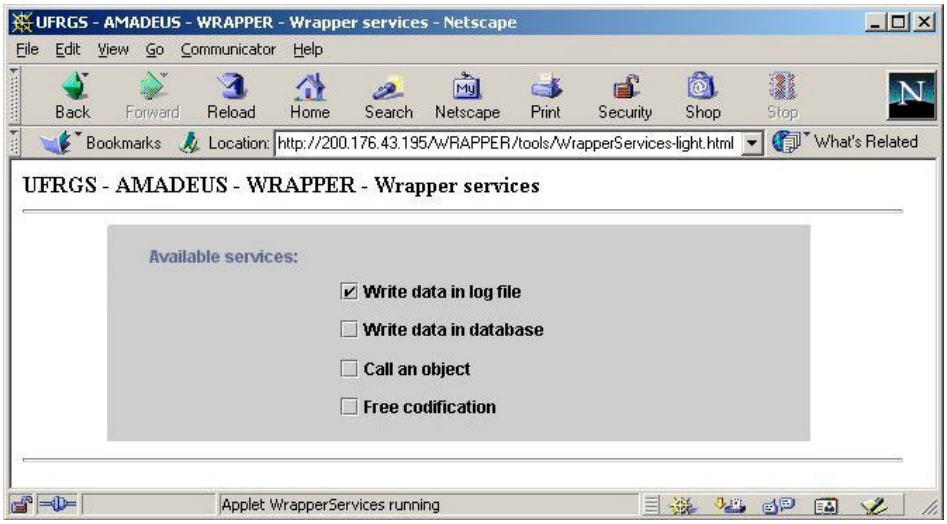

Figura 4.8: Seleção de serviço-*wrapper*

- ***Free codification***: quando uma ação complexa é desejada, é possível codificar ações desejadas a serem feitas (o próximo cenário apresenta esta opção).

Neste exemplo, a opção escolhida foi a gravação de um arquivo de *log*. Este arquivo de *log* ficará disponível para qualquer outro programa que necessitar manipular seu conteúdo. A Figura 4.9 apresenta uma ferramenta que mostra o conteúdo de um arquivo de log.

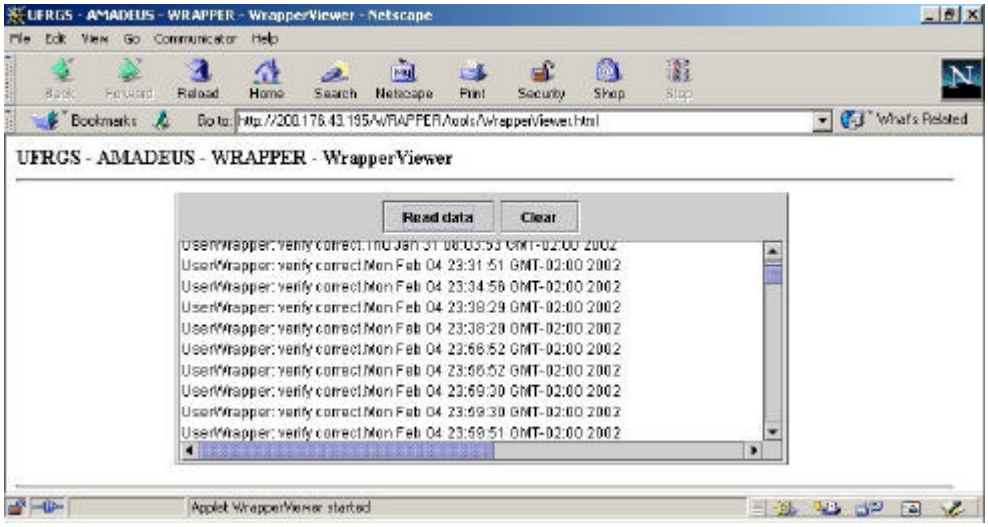

Figura 4.9: Visualizador de conteúdo de arquivo de *log*

4.4.2 Cenário 2: adaptação de processo (alteração de *workflow* de projeto)

Neste cenário o gerente de projeto decide alterar o *workflow* de projeto (por exemplo, suponha que se decidiu que a tarefa "Análise arquitetônica" será ignorada). Como a especificação de um projeto é gerada a partir de um determinado processo de *software*, as tarefas e encadeamento entre elas já estão definidos, bastando ao gerente de projeto apenas instanciar os objetos correspondentes envolvidos no projeto. Porém durante a execução do projeto o gerente pode desejar tomar ações corretivas, como por exemplo: não realizar uma determinada tarefa.

Usualmente em PSEEs que têm suporte à adaptação de processo, a alteração de um projeto já em andamento, ou é realizada através de ações gerenciais realizadas sem o suporte computacional (necessitando de correções manuais no ambiente de suporte para a simular o atendimento da especificação do *workflow* do projeto existente), ou pela alteração da especificação do *workflow* do projeto (implicando em uma recompilação ou re-interpretação do mesmo). Esta última abordagem é a utilizada pelos PSEEs EPOS (NGUYEN, 1997) e SPADE (BANDINELLI, 1993).

Na arquitetura WRAPPER ao invés de alterar a especificação de *workflow* existente e em execução, bastaria o gerente de projeto instanciar um meta-objeto associado à tarefa a ser ignorada. Este meta-objeto poderia capturar o momento da ativação da tarefa e repassá-la a outra tarefa dentro do próprio *workflow*.

Inicialmente o gerente de projeto selecionaria a tarefa desejada (utilizando por exemplo a ferramenta de *workflow*). Depois, como na criação de qualquer meta-objeto, deve-se definir o protocolo de associação entre o meta-objeto e o nível base (Figura 4.10) .

Após a definição do MOP, o gerente de projeto selecionaria o serviço-*wrapper* ***Call an object*** que transferiria a ativação do objeto associado a outro objeto. Desta forma quando o WfMS tentasse ativar a tarefa "Análise arquitetônica" a ser ignorada, o meta-objeto (*wrapper*) correspondente capturaria a mensagem e a repassaria ao outro objeto definido, sem alterar a especificação do *workflow* de projeto já em execução.

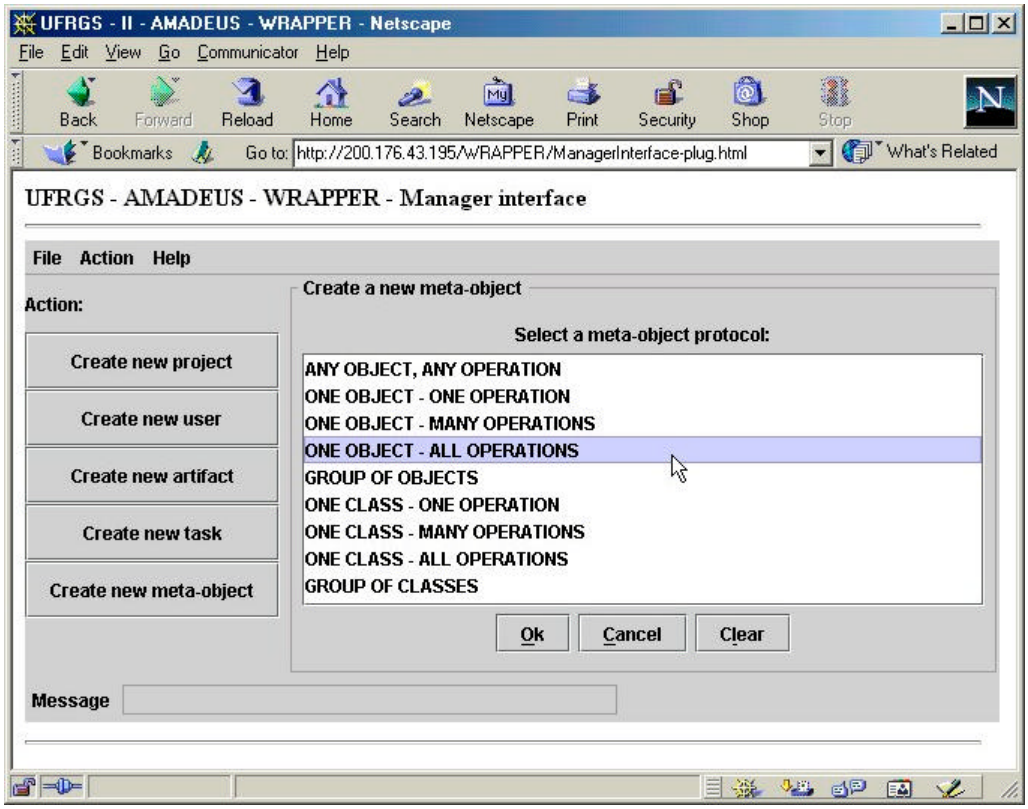

Figura 4.10: Seleção de protocolo do meta-objeto

A Figura 4.11 apresenta uma configuração de objetos e meta-objetos para os cenários 1 e 2.

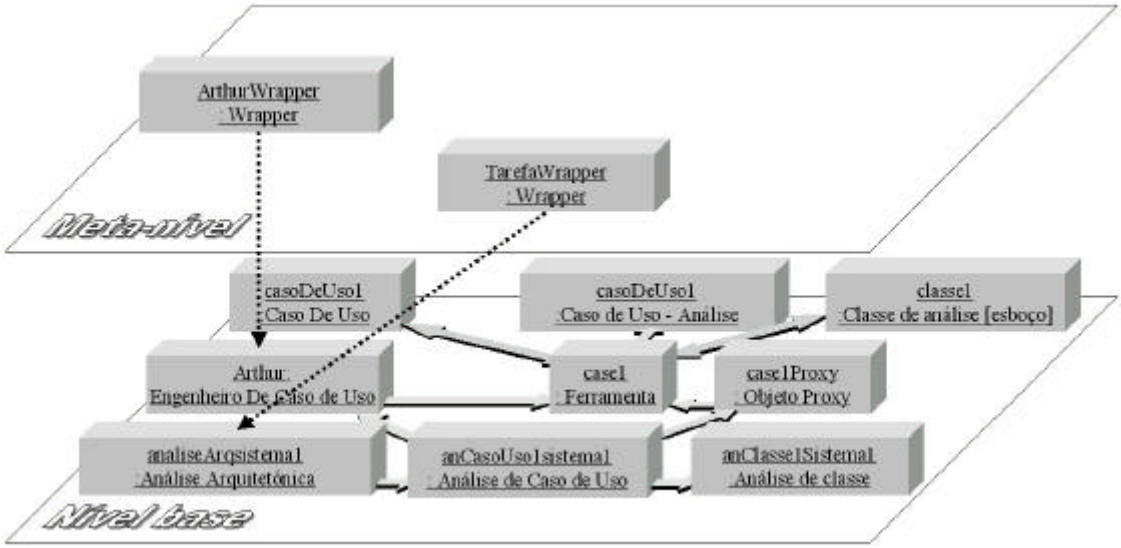

Figura 4.11: Objetos e meta-objetos em níveis

4.4.3 Cenário 3: adaptação de processo (balanceamento de carga de ferramenta)

Neste exemplo, pressupõe-se que se uma ferramenta já estiver em uso, o sistema automaticamente redirecionará a chamada para outra ferramenta. O registro de ferramentas no ADS utiliza uma ferramenta que apresenta a tela da Figura 4.12.

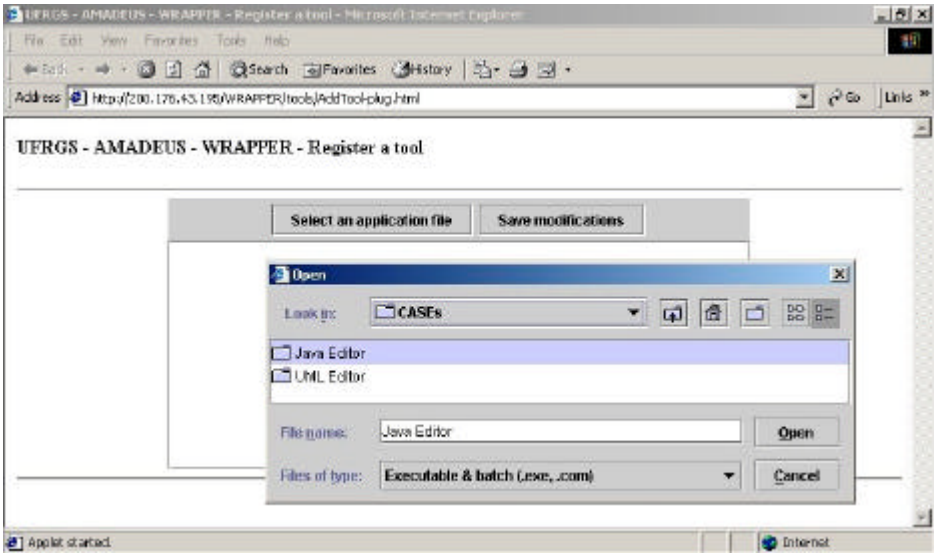

Figura 4.12: Registro de ferramenta

O registro de uma ferramenta gera um objeto-*proxy*, uma vez que uma ferramenta é tratada como um objeto fechado ao ambiente.

O gerente de projeto pode usar a ferramenta de *workflow* ou uma ferramenta separada para selecionar o objeto-*proxy* desejado (Figura 4.13).

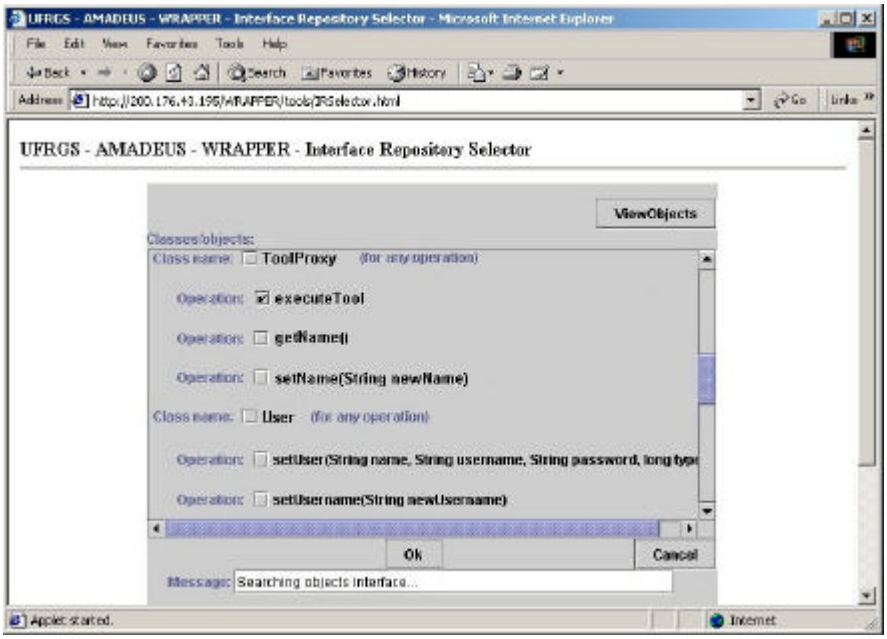

Figura 4.13: Seleção de objeto

Neste exemplo, após a seleção do objeto desejado, o gerente seleciona a opção de codificação livre (*free codification*) para codificar uma ação mais complexa. Ao selecionar esta opção, uma ferramenta de codificação é apresentada (Figura 4.14).

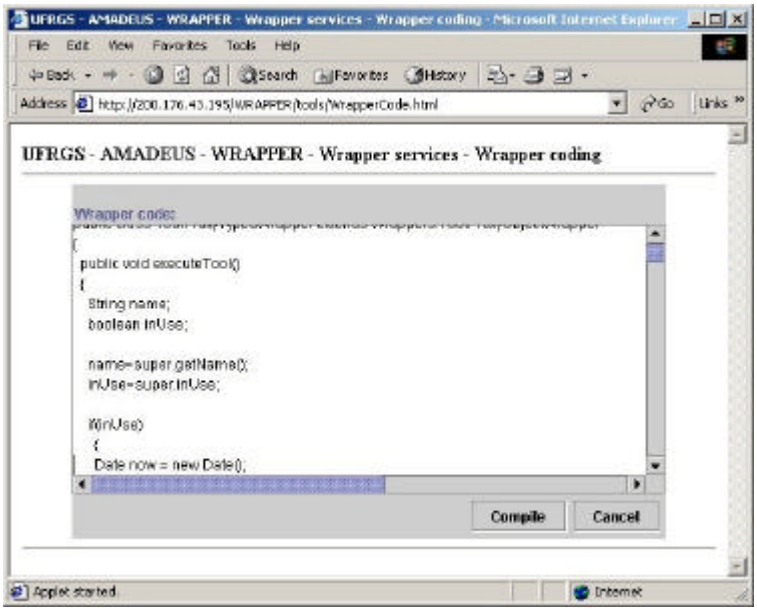

Figura 4.14: Codificação de *wrapper*

O gerente de projeto pode então codificar as ações desejadas, compilar o código e gerar o *wrapper* para ser registrado no ORB.

O diagrama de seqüência UML (Figura 4.15) mostra o processamento após a seleção do objeto e do serviço-*wrapper*. O objeto Serviços-*wrapper* chama o Repositório de Interface para acessar as definições do objeto selecionado. O código criado e a definição da IDL são usados para gerar o *wrapper* correspondente (atualmente é utilizado um compilador Java no servidor). Após a geração do *wrapper*, o passo final é registrá-lo no ORB.

A partir deste momento, o *wrapper* tratará qualquer mensagem à ferramenta monitorada e, dependendo da situação, poderá automaticamente ativar outra ferramenta do ambiente.

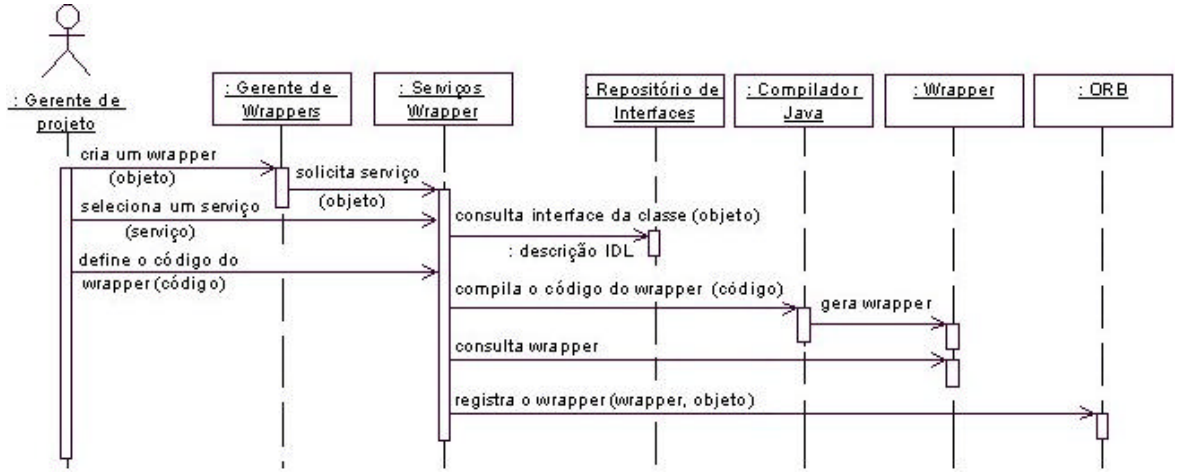

Figura 4.15: Processamento de codificação de *wrapper*

4.5 Aspectos de implementação

Esta seção apresenta algumas características e restrições tecnológicas que influenciaram a implementação do protótipo.

4.5.1 IDL e estrutura de interfaces

A implementação do protótipo, baseada na arquitetura CORBA, necessita obrigatoriamente de uma especificação IDL para a definição das classes de objetos do ambiente. Estas especificações ficam armazenadas no Repositório de Interfaces permitindo que as interfaces dos objetos registrados no ORB possam ser consultadas e processadas.

Qualquer interface definida em uma especificação IDL é descendente da interface OBJECT de CORBA. A interface básica da arquitetura WRAPPER é a *BasicObject*. Para o desenvolvimento do protótipo todo objeto da arquitetura possui duas informações mínimas: um identificador numérico único (*Id*) e uma descrição textual (*Name*). A interface *BasicObject* define operações de alteração (*get*) e de consulta (*set*) destas informações.

O Quadro 4.2 apresenta a IDL básica da arquitetura WRAPPER, da qual as demais classes são derivadas.

```
//================== BASIC DEFINITIONS ====================

typedef sequence< Object > ObjectList;  // List of objects

interface BasicObject
{
  void setId(in long newId);
  long getId();
  void setName(in string name);
  string getName();
};

interface BasicObjectManager : BasicObject
{
  long getNewId();
  boolean verify(in long objectId);
  void save();
  void load();
  void saveDB();
  void loadDB();
  ObjectList getAllObjects();
};
```

Quadro 4.2: IDL de interfaces básicas da arquitetura WRAPPER

Todo objeto CORBA possui um identificador único que pode ser intercambiado entre ORBs de fornecedores diversos, este identificador é chamado de IOR (*Interoperable Object Reference*) (BEN-NATAN, 1998). Em princípio poder-se-ia utilizar este IOR para identificar o objeto, entretanto, o IOR possui informações sobre a localização física do objeto, como por exemplo o número IP ou nome do computador em que o objeto está hospedado. Isto implica que se, por exemplo, o número IP de um computador cliente mudar, o IOR do objeto hospedado nesta computador também muda. Desta forma, o IOR não é uma informação adequada para a identificação de um objeto na arquitetura. Por este motivo é gerado um identificador único, pelo próprio ambiente, para facilitar a identificação dos objetos e seu armazenamento em banco de dados. A descrição textual, ou nome, é utilizada para facilitar a

identificação de objetos por usuários humanos, uma vez que uma informação numérica pode não possuir significado para os mesmos.

Como descendente direto há a interface *BasicObjectManager* (a Figura 4.16 apresenta um diagrama de classes representando a estrutura de interfaces básicas do WRAPPER).

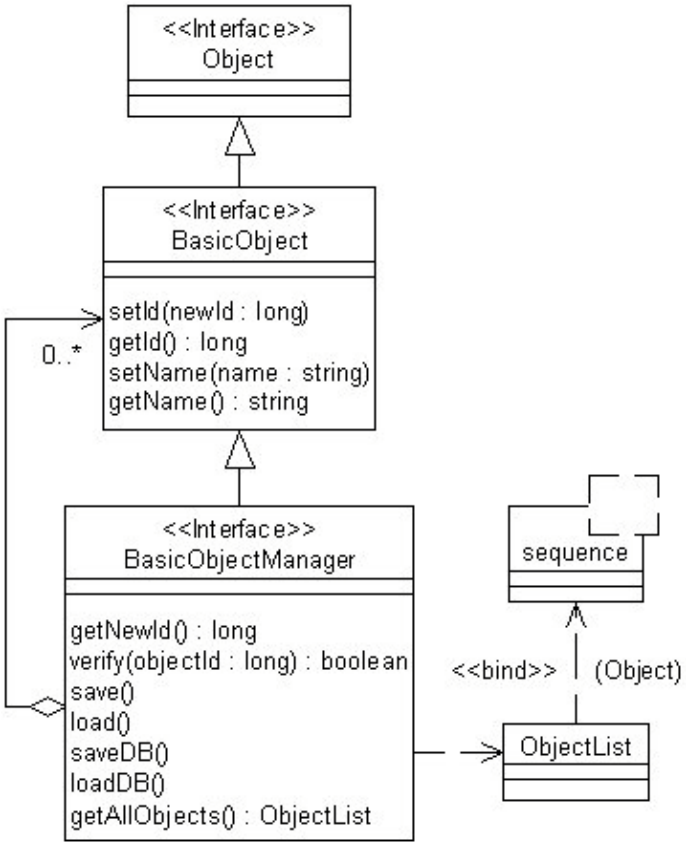

Figura 4.16: Estrutura das interfaces básicas do WRAPPER

A interface *BasicObjectManager* provê operações para o controle de vários objetos *BasicObject*. É responsabilidade, portanto, de um *BasicObjectManager* manter atualizadas as referências de seus objetos *BasicObject*, provendo operações de geração de um novo identificador único (*getNewId*), de verificação de existência de um objeto (*verify*), de controle de persistência em arquivos (*save* e *load*) ou em banco de dados (*saveDB* e *loadDB*) e de consulta a todos os seus objetos (*getAllObjects*).

A partir destas duas interfaces básicas são derivadas as interfaces específicas. Para os objetos básicos (como por exemplo: artefatos, tarefas e ferramentas) são especificadas novas interfaces derivadas de *BasicObject*, enquanto que para os gerentes dos objetos básicos (como por exemplo: subgerente de artefatos, subgerente de tarefas, gerente de nível base) são especificadas interfaces derivadas de *BasicObjectManager*.

4.5.2 *Applets* Java e CORBA: aspectos de segurança

A implementação do protótipo utiliza *applets* Java embutidas em páginas HTML para que as mesmas executem nos clientes através de *browser Web*. Isto possibilita que o ambiente seja acessado em qualquer computador que possua um *browser* Web (que dê suporte a *applets* Java) com acesso à Internet, implicando que o ambiente pode ser executado em clientes remotos e em qualquer plataforma.

De forma adicional, a combinação de Java e CORBA (ORFALI, 1998; BEN-NATAN, 1998) permite que o ORB CORBA também seja enviado com a página HTML. Desta forma, não é necessário a implantação de *software*, nem do ORB, no cliente.

Entretanto, o uso de *applets* traz algumas restrições na execução de funcionalidades. Pelas restrições definidas para *applets* (SUN, 2002a; WALSH, 1998) o acesso ao sistema de arquivos local é proibido. O acesso ao sistema de arquivos local é importante, uma vez que no cliente podem ser integradas e executadas novas ferramentas CASE, bem como criados e armazenados meta-objetos.

Para permitir que uma *applet* Java tenha os mesmos direitos de execução que uma aplicação Java local, é necessário que o gerente de segurança seja alterado por um arquivo de configuração (SUN, 2002a).

O arquivo de configuração .java.policy é utilizado pelo *plug-in* Java para permitir que uma *applet* tenha seus direitos alterados. O uso de *plug-in* Java também é necessário pois a interface gráfica do protótipo implementado é baseada no pacote java.swing que não tem suporte por todos os *browsers* Web.

O Quadro 4.3 apresenta o arquivo de configuração utilizado para permitir que as *applets* implementadas no protótipo WRAPPER possam executar corretamente no cliente.

A primeira alternativa apresentada no arquivo é o uso de *applets* assinadas (no caso, contendo a assinatura "amadeus"). Uma *applet* assinada possui uma assinatura que foi embutida pelo aplicativo *jarsigner*, sendo que a assinatura foi gerada pelo aplicativo *keytool* (SUN, 2002a).

A segunda alternativa é usada para definir que determinados *sites* possuem *applets* que têm permissões especiais no cliente. A vantagem das *applets* assinadas, em relação a esta alternativa, é que elas podem ser hospedadas em qualquer servidor, independentemente do nome ou endereço IP do mesmo.

A terceira alternativa apresentada é apenas para ilustrar que uma *applet* executada localmente também poderá ter suas permissões de execução alteradas.

```
keystore ".keystore";

grant signedBy "amadeus" {
  permission java.security.AllPermission;
  permission java.io.FilePermission "<<ALL FILES>>", "write, read, delete, execute";
};

grant codeBase "http:// 200.176.43.195/WRAPPER/-" {
  permission java.security.AllPermission;
  permission java.io.FilePermission "<<ALL FILES>>", "read, write, delete, execute";
};

grant codeBase "file:/InetPub/wwwroot/WRAPPER/-" {
  permission java.security.AllPermission;
  permission java.io.FilePermission "<<ALL FILES>>", "read, write, delete, execute";
};
```

Quadro 4.3: Arquivo de configuração de segurança Java

O uso de *plug-in* Java e do arquivo de configuração de segurança (.java.policy) são pré-requisitos para que o ambiente possa ser executado corretamente no cliente através de *applets*. Caso contrário seria necessário a instalação na plataforma cliente, tanto do ORB quanto de *software* cliente completo.

4.6 Resultados preliminares

O protótipo (YAMAGUTI, 2002a; YAMAGUTI, 2002b) não envolveu a implementação de um PSEE completo. Entretanto, a execução dos exemplos de cenários apresentados mostraram que utilizando o protótipo é possível observar o projeto em execução e adaptá-lo utilizando meta-objetos (*wrappers*) inseridos na especificação do projeto.

Um aspecto crítico observado até o momento é o baixo desempenho. O desempenho é afetado principalmente porque o cliente ao acessar o servidor a primeira vez, todo o ORB é copiado para a máquina

cliente, o que provoca alto tráfego de rede e queda no desempenho do sistema. Por ser baseado na Web, o protótipo é muito sensível a variações da conexão de comunicação entre o cliente e o servidor.

Soluções iniciais para este problema seriam: a instalação do ORB no cliente (diminuindo o tráfego inicial de rede) e a replicação do *workflow* do projeto entre os cliente participantes (permitindo um grau maior de independência dos clientes em relação ao servidor).

Resultados mais específicos e novas conclusões deverão surgir à medida que o protótipo incluir mais funcionalidades e ferramentas do ambiente de suporte a processo de *software*, a ser complementado em trabalhos futuros.

5 CONSIDERAÇÕES FINAIS

O presente trabalho iniciou motivado pela busca de uma arquitetura de um ambiente centrado em processo que, pelo uso de reflexão computacional, permitisse dinamicamente coletar dados sobre a execução do processo em desenvolvimento, bem como permitisse, também, dinamicamente alterar a própria execução deste processo.

O resultado foi a definição da arquitetura WRAPPER que pela combinação de reflexão computacional, objetos distribuídos, *middleware* e tecnologias Web permite ao ambiente de suporte a processo obter as seguintes características:

a. **flexibilidade**: o uso de meta-objetos para o controle de um processo de *software* permite que um gerente de processo possa, a qualquer momento, obter informações sobre a execução do mesmo, bem como alterar o próprio processo. Em adição, o uso de meta-objetos permite que o próprio ambiente seja modificado, uma vez que todos os elementos do ambiente são tratados, de forma homogênea, por objetos.

b. **distribuição**: a utilização de tecnologias Web e objetos distribuídos permite que o uso do ambiente seja realizado em locais remotos. Desta forma, as tarefas envolvidas em um processo podem ser distribuídas remotamente, e os agentes envolvidos em um processo podem realizar estas tarefas em qualquer ponto da *World Wide Web*.

c. **heterogeneidade**: *middleware* CORBA e tecnologias Web permitem que o ambiente possa agregar ferramentas de diversos fornecedores e plataformas ao ambiente. Além disso, o uso de tecnologias Web permite que o ambiente seja executado em qualquer plataforma cliente.

5.1 Limitações de uma arquitetura reflexiva

A utilização de reflexão computacional em sistemas aplicativos, e em particular em um PSEE, traria alguns benefícios como já apresentados ao longo deste trabalho.

Entretanto, apesar dos benefícios obtidos existem alguns pontos problemáticos a considerar pelo uso de reflexão computacional (LISBOA, 1997; BUSCHMANN, 1996):

a. **desempenho de sistemas reflexivos é prejudicado**: cada nível de reflexão inserido provoca uma perda de desempenho no sistema pelo redirecionamento do processamento computacional. Desta forma, um PSEE implementado, utilizando-se desta tecnologia, provavelmente tem sua eficiência prejudicada.

b. **a privacidade pode ser violada**: considerando-se um sistema baseado em objetos, a reflexão computacional pode violar o encapsulamento dos objetos do nível base.

c. **a segurança pode ser prejudicada**: uma vez que é possível controlar objetos de nível base, se os meta-objetos forem manipulados maliciosamente, todo o sistema pode ser violado.

d. **linguagens de programação**: ainda existe uma carência de suporte à reflexão provida por linguagens de programação. A tendência é a criação e adaptação de linguagens de programação que possibilitem fácil suporte ao desenvolvimento de sistemas. A exceção são as linguagens de programação Java (versão 1.4) (SUN MICROSYSTEMS, 2002) e Smalltalk (CAMPO, 1997).

e. **falta de padronização**: a implementação de reflexão, em particular na Internet, é um campo ainda em desenvolvimento e sem soluções padronizadas.

Os requisitos de desempenho e de segurança (tolerância a falhas) não são os principais pontos trabalhados no presente trabalho. Entretanto, algumas alternativas podem ser definidas para minimizar estas limitações. Abaixo são sugeridas algumas alternativas para minimizar as restrições de desempenho e segurança:

a. **Desempenho**:

- **replicação do *workflow* de processo nos clientes**: como sugerido em (YANG, 1998), permitiria que o *workflow* do processo seja distribuído entre os diversos clientes, minimizando a carga do sistema no servidor;
- **instalação completa do ambiente nos clientes**: permitiria que o ambiente fosse executado a partir de uma aplicação local, permitindo que o tráfego de informações em rede seja diminuído;
- **implementação de caches de objetos locais**: para diminuir o tráfego de objetos em rede, poderiam ser implementados mecanismos de replicação de objetos mais utilizados localmente

nos clientes como caches de objetos que seriam atualizados eventualmente quando uma operação crítica como remoção ou alteração de estado fosse realizada.

b. **Segurança**:

- **replicação do *workflow* de processo nos clientes**: permitiria que os clientes executassem suas funcionalidades mesmo na ocorrência de uma falha na rede ou de um dos clientes;
- **replicação de servidores**: permitir que um *workflow* de processo compartilhado e controlado por mais de um servidor, de forma que na eventualidade de falha em um servidor, outro possa continuar o processamento;
- **protocolos seguros e criptografia**: a implementação de protocolos seguros para comunicação entre clientes e servidor, bem como aplicação de algoritmos de criptografia para informações sigilosas, permitiriam melhorar a segurança das informações do sistema;
- **restrições de integridade no MOP**: para evitar que os meta-objetos causem inconsistências nos objetos de nível base, o protocolo de meta-objetos deve garantir que a integridade de consistência através de por exemplo, observação de asserções e invariantes nas classes dos objetos de nível base.

5.2 Contribuições

A principal contribuição da presente tese é a definição de uma arquitetura reflexiva que descreve como a reflexão computacional pode ser aplicada no contexto de processo de *software* (YAMAGUTI, 2002a; YAMAGUTI 2002b), permitindo que durante a execução de um processo informações possam ser dinamicamente extraídas e o mesmo possa ser alterado, também de forma dinâmica, mesmo sendo executado em um ambiente distribuído e heterogêneo.

Especificamente, o desenvolvimento deste trabalho trouxe as seguintes contribuições:

a. **Definição da estrutura de uma arquitetura reflexiva para um ambiente de suporte a processo de *software***: utilizando-se de conceitos de reflexão computacional sobre objetos, foi definida uma arquitetura reflexiva, com seus componentes e inter-relacionamentos, para um ambiente de suporte a processo;

b. **Definição de uso de reflexão computacional para captura dinâmica de informações sobre execução de processo**: foi apresentada uma estratégia para conciliar a modelagem de processo baseado em objetos e o uso de reflexão computacional

sobre os objetos definidos para, dinamicamente, extrair informações sobre a execução do processo;

c. **Definição de uso de reflexão computacional para a alteração dinâmica de um processo**: um processo de *software* deve ser flexível, de forma que possa ser adaptado a qualquer momento. O presente trabalho apresentou o uso de reflexão computacional como alternativa para a adaptação dinâmica de um processo, mesmo que este esteja em andamento.

5.3 Trabalhos futuros

A partir da estrutura da arquitetura definida e do protótipo implementado, prevê-se como tarefas futuras decorrentes deste trabalho:

a. **novos Serviços *Wrappers***: novos serviços devem ser desenvolvidos para facilitar a determinação do comportamento de meta-objetos criados no ambiente, de forma que um gerente de processo (ou de projeto), para ações mais complexas não tenha que codificar, em linguagem de programação, a lógica desejada para um meta-objeto. Outros serviços que podem ser disponibilizados seriam: controle de meta-objetos específicos para a gerência de *workflow*, geração de registros sobre as intervenções realizadas (como por exemplo: reconfiguração do *workflow*) e controle de 'torre de reflexão computacional' (LISBOA, 1997) de meta-objetos sobre meta-objetos.

b. **adição de mecanismo de integração de dados**: XML (WWW CONSORTIUM, 2002) deverá ser utilizada como mecanismo de integração de dados entre as ferramentas do ambiente, permitindo que artefatos, especificados nesta linguagem, sejam descritos e intercambiados entre as várias ferramentas do ambiente. Um experimento realizado comprovando a viabilidade desta abordagem foi desenvolvido em (PEREIRA, 2002);

c. **consolidação de meta-objetos**: atualmente um processo pode ser adaptado por diversos meta-objetos instanciados dinamicamente. Entretanto, para que novas execuções do processo também possam tirar proveito das adaptações definidas, é necessário que os meta-objetos sejam definitivamente incorporados (como objetos de nível base) ao processo alterado, sendo portanto, necessário um mecanismo de consolidação de meta-objetos ao modelo de processo.

d. **interoperabilidade**: experimentos com diferentes ORBs CORBA devem ser realizados para testar a integração de objetos de outros ORBs com o ambiente;

e. **melhoria de desempenho**: estratégias alternativas para melhorar o desempenho de execução do processo, como

replicação do *workflow* de processo nos clientes e instalação completa do ambiente nos clientes também deverão ser desenvolvidas.

REFERÊNCIAS

AMARAL, V. L. **Técnicas de Modelagem de *Workflow***. 1997. Trabalho Individual (Mestrado em Ciência da Computação) - Instituto de Informática, UFRGS, Porto Alegre.

ARAUJO, R. M.; BORGES, M. R. S. Sobre a Aplicabilidade de Sistemas de *Workflow* no Suporte de *Software*. In: WORKSHOP IBEROAMERICANO DE INGENIERÍA DE REQUISITOS Y AMBIENTES DE SOFTWARE, IDEAS, 2., 1999, San Jose, Costa Rica. **Memorias...** San Jose, Costa Rica: ITCR, 1999. p. 417-428.

BALDI, M. et al. E^3: object-oriented software process model design. In: FINKELSTEIN, A. et al. **Process Software Modelling and Technology**. Somerset: Research Studies Press, 1994. chap.11, p. 279-292.

BANDINELLI, S. et al. Process Enactment in SPADE. In: EUROPEAN WORKSHOP ON SOFTWARE PROCESS TECHNOLOGY, 2., 1992, Trondheim, Norway. **Proceedings**... [S.l.]: Springer, 1992. p. 67-83.

BANDINELLI, S.; FUGETTA, A. Computational Reflection in Software Process Modeling: the SLANG Approach. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 15., 1993, Baltimore. **Proceedings**... [S.l.]: IEEE Computer Society Press, 1993. p. 144-154.

BELKHATIR, N. et al. ADELE-TEMPO: an environment to support process modelling and enactment. In: FINKELSTEIN, A. et al. **Process Software Modelling and Technology**. Somerset: Research Studies Press, 1994. chap.8, p.187-222.

BELL, R.; SHARON, D. Tools to Engineer New Technologies into Applications. **IEEE Software**, Los Alamitos, v.12, n.2, p.11-6, Mar. 1995.

BEN-NATAN, R. **CORBA on the Web**. New York: McGraw-Hill, 1998.

BEN-NATAN, R. **CORBA**: a guide to common object request broker architecture. New York: McGraw-Hill, 1995.

BEN-NATAN, R. **Objects on the Web**: designing, building, and deploying object-oriented applications for the Web. New York: McGraw-Hill, 1997.

BEN-SHAUL, I. Z.; KAISER, G. E. A Paradigm for Decentralized Process Modeling and its Realization in the Oz Environment. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 16., 1994, Sorrento-Italy. **Proceedings**... Berlin: IEEE Computer Society Press, 1994. p.179-188.

BERNERS-LEE, T. **Hypertext Markup Language - 2.0**. D. Connoly: HTML Working Group - MIT/W3C, 1995.

BIDER, I.; KHOMYAKOV, M. Object-oriented Model for Representing Software Production Processes. In: ECOOP, 12., 1998, Brussels-Belgium. **Proceedings**... Berlin: Springer-Verlag, 1998.

BOOCH, G. et al. **The Unified Modeling Language User Guide**. Boston: Addison-Wesley, 1998.

BORLAND. **Visibroker for Java 4.5**: product documentation. Disponível em: < http://www.borland.com/techpubs/visibroker/visibroker45/vbj45-index.html >. Acesso em: 30 out. 2002.

BUSCHMANN, F. et al. **Pattern-Oriented Software Architecture**: a system of patterns. Chichester: John Wiley & Sons, 1996.

CAMPO, M. R.; PRICE, R. T. Um *Framework* Reflexivo para Ferramentas de Visualização de *Software*. In: SIMPÓSIO BRASILEIRO DE ENGENHARIA DE *SOFTWARE*, 10., 1996, São Carlos. **Anais**... São Carlos: UFSCAR, 1996. p. 153-69.

CAMPO, M. R. **Compreensão Visual de *Frameworks* através da Introspeção de Exemplos**. 1997. Tese (Doutorado em Ciência da Computação) - Instituto de Informática, UFRGS, Porto Alegre.

CONRADI, R. et al. Concepts for Evolving Software Process. In: FINKELSTEIN, A. et al. **Process Software Modelling and Technology**. Somerset: Research Studies Press, 1994. chap.2, p.9-31.

COSTA, F. M.; BLAIR, G. S. Integrating Meta-Information Management and Reflection in Middleware. In: INTERNATIONAL SYMPOSIUM ON DISTRIBUTED OBJECTS & APPLICATIONS. 2., 2000, Antwerp-Belgium. **Proceedings**... [S.l.: s.n.], 2000. p.133-143.

COSTA, F. M. et al. The Role of Reflective Middleware in Supporting the Engineering of Dynamic Applications. In: Reflection and Software Engineering. [S.l.]: Springer, 2000. p. 79-98.

COULOURIS, G.; DOLLIMORE, J.; KINDBERG, T. **Distributed systems**: concepts and design. 3rd ed. Wokingham: Addison-Wesley, 2001.

DART, S. A. et al. Software development environments. **Computer**, Los Alamitos, v.20, n.11, p. 18-28, 1988.

DOURISH, P. Developing a reflexive model of collaborative systems. **ACM Transactions on Computer-Human Interaction**, Baltimore, v. 2, n.1, p.40-63, Mar. 1995.

DUCHIEN, L.; SEINTURIER, L. Reflective observation of CORBA Applications. In: IASTED INTERNATIONAL CONFERENCE ON PARALLEL AND

DISTRIBUTED COMPUTING AND SYSTEMS, 11., 1999, Boston-USA. **Proceedings**... Boston-USA: IASTED, 1999. p.311-316.

FINKELSTEIN, A. et al. **Process Software Modelling and Technology**. Somerset: Research Studies Press, 1994.

FISHER, A. S. **CASE**: utilização de ferramentas para desenvolvimento de *software*. Rio de Janeiro: Campus, 1990.

FUGGETTA, A. A Classification of CASE Technology. **Computer**, Los Alamitos, v. 26, n.12, p.25-38, Dec. 1993.

GEORGAKOPOULOS, D.; HORNICK, M.; SHETH, A. An Overview of Workflow Management: from process modeling to workflow automation infrastructure. **ACM Distributed and Parallel Databases**, New York, n.3, p.119-153, Mar. 1995.

GIMENES, I. M. S. **ExPSEE**: um ambiente experimental de engenharia de *software* orientado a processos. Maringá: Universidade Estadual de Maringá, Depto. de Informática, 2000. Relatório de Projeto.

GIMENES, I. M. S. Uma Introdução ao Processo de Engenharia de *Software*. In: JORNADA DE ATUALIZAÇÃO EM INFORMÁTICA, 13., 1994, Caxambú. **Anais**... Caxambú:SBC, 1994.

GROTH, B. **Concepts for Integrating an Object Oriented Generic Process Model into a Software Development Environment**. Disponível em: < http://www.swt.cs.tu-berlin.de/Publications/DA-Groth.ps.gz >. Acesso em: 30 out. 2002.

HAN, J. Object-oriented Modeling of Software Processes and Artifacts: promisses and challenges. In: ECOOP, 12., 1998, Brussels-Belgium. **Proceedings**... Berlin: Springer-Verlag, 1998.

HOLZNER, S. **Inside XML**. Indianapolis: New Riders Publishing, 2000.

HOQUE, R. **CORBA 3**. Foster City: IDG Books, 1998.

HUMPHREY, W. S. **A Discipline for Software Engineering**. Reading: Addison-Wesley, 1995.

HUMPHREY, W. S. **Managing the Software Process**. Reading: Addison-Wesley, 1990.

JACOBSON, I. et al. **The Unified Software Development Process**. Reading: Addison-Wesley, 1998.

JAMART, P.; LAMSWEERDE, A. **A Reflexive Approach to Process Model Customization, Enacment and Evolution**. Disponível em: < ftp://ftp.info.ucl.ac.be/pub/publi/94/ICSP3.ps >. Acesso em: 30 out. 2002.

KRATZ, L. G. R. **Estudo Comparativo de Ambientes de Suporte ao Desenvolvimento de *Software***. 1998. Trabalho Individual (Mestrado em Ciência da Computação) - Instituto de Informática, UFRGS, Porto Alegre.

KURNIAWAN, B. **Java for the Web with Servlets, JSP, and EJB**: a developer's guide to J2EE solutions. Boston: New Riders, 2002.

LADD, E.; O'DONNELL, J. **Platinum Edition using HTML 4, Java 1.1 and *JavaScript* 1.2**. 2nd. ed. Indianapolis: QUE, 1998.

LIMA, C. A. G. de. **Estudo de Gerência no Processo de Desenvolvimento de *Software***. 1995. Trabalho Individual (Mestrado em Ciência da Computação) - Instituto de Informática, UFRGS, Porto Alegre.

LIMA, F. A. de. **WIDe**: uma extensão à UML para modelagem de sistemas de informação na internet baseados em documentos. 2000. Dissertação (Mestrado em Ciência da Computação) - Instituto de Informática, UFRGS, Porto Alegre.

LISBOA, M. L. B. Arquitetura Reflexiva para o Desenvolvimento de *Software* Tolerante a Falhas. In: SEMINÁRIO INTEGRADO DE *SOFTWARE* E *HADWARE*, 23., 1996, Recife. **Anais**... Recife: UFPE, 1996. p.155-66.

LISBOA, M. L. B. Arquiteturas de Meta-Nível. In: SIMPÓSIO BRASILEIRO DE ENGENHARIA DE *SOFTWARE*, 11., 1997, Fortaleza. **Tutorial**... Fortaleza: UFC, 1997. 35 p.

LISBOA, M. L. B. **Motf**: meta-objetos para tolerância a falhas. 1995. Tese (Doutorado em Ciência da Computação) - Instituto de Informática, UFRGS, Porto Alegre.

LISBOA, M. L. B. **Reflexão Computacional no Modelo de Objetos**. In: SEMANA DE INFORMÁTICA DA UFBA, 7., 1998, Salvador. **Anais**... Salvador: UFBA, 1998. p.1-44.

MAES, P. Concepts and Experiments in Computational Reflection. **SIGPLAN Notices**, New York, v.22, n.12, p. 147-169, Dec. 1987. Trabalho apresentado na OOPSLA, 1987.

MANGAN, C. A. G. **Aspectos de Implementação de um Modelo de Gerência do Processo de Desenvolvimento de *Software***: arquitetura e protocolos para um gerente de designflow. 1998. Dissertação (Mestrado em Ciência da Computação) - Instituto de Informática, UFRGS, Porto Alegre.

MANZONI, L. V. **Ambiente de Gestão do Processo de Desenvolvimento**. 2000. Trabalho Individual (Mestrado em Ciência da Computação) - Instituto de Informática, UFRGS, Porto Alegre.

MANZONI, L. V. **Uso de Sistema de Gerência de *Workflow* para Apoiar o Desenvolvimento de *Software* Baseado no Processo Unificado da Rational Estendido para Alcançar Níveis 2 e 3 do Modelo de Maturidade**. 2001. Dissertação (Mestrado em Ciência da Computação) - Instituto de Informática, UFRGS, Porto Alegre.

MICROSOFT. **Distributed Component Object Model (DCOM) - downloads, specifications, samples, papers, and resources for Microsoft DCOM**. Disponível em: < http://www.microsoft.com/com/tech/DCOM.asp >. Acesso em: 30 out. 2002.

MICROSOFT. **Internet Explorer Home Page**. Disponível em: < http://www.microsoft.com/windows/ie/default.asp >. Acesso em: 30 out. 2002.

MICROSOFT. **Windows 2000 Server Home**. Disponível em: < http://www.microsoft.com/windows2000/server/default.asp >. Acesso em: 30 out. 2002.

NETSCAPE. **Browser**. Disponível em: < http://channels.netscape.com/ns/ browsers/default.jsp >. Acesso em: 30 out. 2002.

NGUYEN, M. N. et al. Total Software Process Model Evolution in EPOS. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 7., 1997, Boston-EUA. **Proceedings**... Berlin: Springer-Verlag, 1997.

NOTARI, D. L. **Arquiteturas para Dicionários de Dados de Ambientes de Desenvolvimento de *Software***. 1999. Trabalho Individual (Mestrado em Ciência da Computação) - Instituto de Informática, UFRGS, Porto Alegre.

OCAMPO, C.; BOTELLA, P. **Some Reflections on Applying Workflow Technology to Software Process**. Disponível em: < http://www.lsi.upc.es/~cocampo/publications.html >. Acesso em: 30 out. 2002.

OLIVA, A. et al. **Guaraná**: uma arquitetura de *software* para reflexão computacional implementada em Java. 1998. Dissertação (Mestrado em Ciência da Computação) - Instituto de Informática, UNICAMP-IC, Campinas.

OLIVEIRA, K. et al. A Estação TABA e Ambientes de Desenvolvimento de *Software* Orientados a Domínio. In: SIMPOSIO BRASILEIRO DE ENGENHARIA DE *SOFTWARE*, 14., João Pessoa, 2000. **Anais**... João Pessoa: UFPB, 2000. p. 343-346.

OMG. **Complete Formal CORBA/IIOP 2.3 Specification**. Disponível em: < http://www.omg.org/cgi-bin/doc?formal/98-12-01 >. Acesso em: 30 out. 2002.

OMG. **Interceptors**. Disponível em: < http://www.omg.org/cgi-bin/doc?formal/99-07-25 >. Acesso em: 30 out. 2002.

OMG. **Object Management Group Home Page**. Disponível em: < http://www.omg.org/ >. Acesso em: 30 out. 2002.

ORFALI, R.; HARKEY, D. **Client/server Programming with Java and CORBA**. New York: John Wiley & Sons, 1998.

ORTIGOSA, A. M. **Proposta de um Ambiente Adaptável de Apoio ao Processo de Desenvolvimento de *Software***. 1995. Dissertação (Mestrado em Ciência da Computação) - Instituto de Informática, UFRGS, Porto Alegre.

PAULK, M. C. et al. Capability Maturity Model: Version 1.1**. IEEE Software**, Los Alamitos, v. 10, n. 4, p, 18-27, July 1993.

PEREIRA, M. Z.; YAMAGUTI, M. H. A Prototype for Data Integration between CASE Tools. In: ARGENTINE SYMPOSIUM ON SOFTWARE ENGINEERING, JAIIO, 31., 2002, Santa Fe - Argentina. **Anales...** Buenos Aires: SADIO, 2002. p. 168-179.

PORTELLA, E. et al. Teoria e Prática na Representação do Processo de Desenvolvimento de *Software*: um estudo de caso comparativo. In: CLEI, 1998, Quito. Quito-Ecuador: Pontificia Universidad Católica Del Ecuador, 1998. v. 2, p. 985-997.

PRESSMAN, R. S. **Software Engineering**: a practitioner's approach. New York: McGraw-Hill, 2001.

REIMER, W. et al. Towards a Dedicated Object Oriented Software Process Modelling Language. In: ECOOP, 12., 1998, Brussels-Belgium. **Proceedings**... Berlin: Springer-Verlag, 1998.

ROCKWELL, W. **XML, XSLT, Java, and JSP**: a case study in developing a web application. Indianapolis: New Riders, 2001.

ROSSI, S.; SILLANDER, T. A Practical Approach to Software Process Modelling Language Engineering. In: EWSPT, 6., 1998, Weybrigde-UK. **Proceedings**... [S.l.: s.n.], 1998.

SA, J.; WARBOYS; B.C. A Reflexive Formal Software Process Model. In: EWSPT, 4., 1995, **Proceedings**... Berlin: Springer-Verlag, 1995. p. 241-254.

SHARON, D.; BELL, R. Tools that Bind: creating integrated environments. **IEEE Software**, Los Alamitos, v.12, n.2, p.76-85, Mar. 1995.

SHAW, M.; GARLAN, D. **Software Architecture**: perpectives on an emerging discipline. Upper Saddle River: Prentice-Hall, 1996.

SINGHAI, A.; CAMPBELL, R. Reflective ORBs: supporting robust, time-critical distribution. In: ECOOP, 11., 1997, Jyväskylä-Finland. **Object-oriented Programming**: Proceedings. Berlin: Springer-Verlag, 1997.

SNOWDON, R. A.; WARBOYS, B. C. An Introduction to Process-centred Environments. In: FINKELSTEIN, A. et al. **Process software modelling and technology**. Somerset: Research Studies Press, 1994. chap.1, p.1-8.

SOMMERVILLE, I. **Software Engineering**. 4th ed. Reading: Addison-Wesley, 1992.

SPICE. Software Process Improvement and Capability dEtermination. **Software Process Assessment**: part 9 - vocabulary. Disponível em: < http://www.sqi.gu.edu.au/spice/docs/baseline/part9100.doc >. Acesso em: 30 out. 2002.

SUN MICROSYSTEMS. **Java(tm) Technology Home Page**. Disponível em: < http://java.sun.com/j2se/1.4.1/docs/index.html >. Acesso em: 30 out. 2002.

SUN MICROSYSTEMS. **The Java Tutorial**. Disponível na WWW em: < http://java.sun.com/docs/books/tutorial/ >. Acesso em: 30 out. 2002.

THOMAS, I.; NEJMEH, B. Definitions of Tool Integration for Environments. **IEEE Software**, Los Alamitos, v.9, n.2, p.29-35, Mar. 1992.

UMAR, A. **Object-oriented client/server internet environments**. Upper Saddle River, NJ: Prentice-Hall, 1997.

WALSH, A.; FRONCKOWIAK, J. **Java Bible**. Foster City: IDG Books, 1998.

WANG, I. A. **Use of Object Orientation in Process Modeling Languages**. Technical report about OO-technology in PMLs. Disponível em: < http://www.idi.ntnu.no/~alfw/work/publications.html >. Acesso em: 30 out. 2002.

WASSERMAN, A. I. The Ecology of Software Development Environments. **Computer**, Los Alamitos, v.14, p.28-33, 1981.

WEGDAN, M. et al. Using Message Reflection in a Management Architecture for CORBA. In: IFIP/IEEE INTERNATIONAL WORKSHOP ON DISTRIBUTED SYSTEMS: OPERATIONS & MANAGEMENT, DSOM, 11., 2000, Austin, Texas, USA. **Proceedings...** [S.l.: s.n.], 2000.

WORKFLOW MANAGEMENT COALITION. **A Common Object Model: discussion paper**. Disponível em: < http://www.wfmc.org/standards/docs/TC-1022_commom_Object%20Model_Paper.pdf >. Acesso em: 30 out. 2002.

WORKFLOW MANAGEMENT COALITION. **Interface 1 - process definition interchange process model**. Disponível em: < http://www.wfmc.org/standards/docs/TC-1016-P_v11_IF1_Process_definition_Interchange.pdf >. Acesso em: 30 out. 2002.

WORKFLOW MANAGEMENT COALITION. **Terminology and Glossary**. Disponível em: < http://www.wfmc.org/standards/docs/C-1011_term_glossary_v3.pdf >. Acesso em: 30 out. 2002.

WU, Z. Reflexive Java and a Reflexive Component-based Transaction Architecture. **SIGPLAN Notices**, New York, v. 33, n. 10, p. 6-10, Oct. 1998. Trabalho apresentado na OOPSLA, 1998.

WWW CONSORTIUM. **Extensible Markup Language (XML)**. Disponível em: < http://www.w3.org/XML/ >. Acesso em: 30 out. 2002.

YAMAGUTI, M. H.; PRICE, R. T. A Web-based Reflective Architecture for Process Support Environment. In: ARGENTINE SYMPOSIUM ON SOFTWARE ENGINEERING, JAIIO, 31., 2002. Santa Fe - Argentina. **Anales...** Buenos Aires: SADIO, 2002. p. 103-116.

YAMAGUTI, M. H.; PRICE, R. T. Uma Arquitetura Reflexiva Baseada na Web para Ambiente de Suporte a Processo. In: SIMPÓSIO BRASILEIRO DE ENGENHARIA DE *SOFTWARE*, 16., 2002, Gramado. **Anais**... Porto Alegre: Instituto de Informática - UFRGS, 2002. p. 284-299.

YANG, Y. Issues on Supporting Distributed Software Processes. In: EWSPT, 6., 1998, Weybrigde-UK. **Proceedings**... Berlin: Springer-Verlag, 1998. p.143-147.