Universidade de São Paulo
Instituto de Matemática e Estatística

MAC499 - Trabalho de Formatura Supervisionado

Sistema de Gerenciamento de Workflows

Monografia

Carlos H. de Fernandes - N^{o}USP:3286544 - carloshf at linux.ime.usp.br
Cleber Miranda Barboza - N^{o}USP:3286353 - cleberc at linux.ime.usp.br
Giuliano Mega - N^{o}USP:3286245 - giuliano at linux.ime.usp.br
Pedro Losco Takecian - N^{o}USP:3286307 - plt at linux.ime.usp.br

Orientador

Alfredo Goldman vel Lejbman - gold at ime.usp.br

29 de fevereiro de 2004

Sumário

1 Introdução **3**
1.1 Visão geral 3
1.2 Definições 3
1.3 Caso de uso 4

2 Arquitetura do sistema **5**
2.1 Visão geral 5
2.2 Camada de apresentação 6
2.2.1 Interface web 8
2.3 Camada de negócios 9
2.3.1 Núcleo 10
2.3.2 Extensões 27
2.4 Camada de dados 30

3 Organização do projeto **30**
3.1 Responsabilidades 30
3.2 Andamento 31
3.3 Situação atual 31
3.4 Ferramentas utilizadas 31

4 Experiência pessoal **32**
4.1 Desafios e frustrações 32
4.2 Disciplinas do BCC mais relevantes 33
4.3 Interação com os membros da equipe 34
4.4 Considerações Finais 34

5 Referências **34**

1 Introdução

1.1 Visão geral

Este trabalho teve como objetivo o desenvolvimento de um sistema de gerenciamento de processos de produção (*workflows* ou fluxos de trabalho), desenvolvido partindo-se da análise detalhada de diversos softwares já consolidados no mercado.

As *Workflow Engines* são ferramentas especificamente voltadas para a modelagem de *business processes*; isto é, processos dinâmicos cuja evolução é condicionada ao cumprimento de tarefas pré-definidas. Estes processos são constituídos por diversos estados; é trabalho da *Workflow Engine* fornecer o ambiente e as ferramentas necessárias para que seja possível modelar, com o maior grau de flexibilidade e abrangência possíveis, tais processos. É também papel da *Workflow Engine* refletir no sistema, a cada instante, o estado global desses processos, bem como distribuir as tarefas e colher os resultados das diversas partes integrantes.

Para uma organização, há varias vantagens em utilizar um Sistema de Gerenciamento de Fluxos de Trabalho, entre elas, podemos citar:

- melhora da eficiência da organização;
- aumento de produtividade;
- aperfeiçoamento de processos e relatórios de controle;
- maior aceitação, pelos empregados, dos regulamentos internos e externos;
- melhora da vantagem competitiva;
- aumento do conhecimento dos processos da organização.

Nos tópicos que se seguem, iremos abordar um caso de uso da aplicação, a arquitetura utilizada, algumas decisões de implementação, e a organização adotada durante o desenvolvimento do projeto.

1.2 Definições

Embora a definição de sistema de workflow apresentada seja razoavelmente precisa, alguns pontos referentes a sua utilização num ambiente empresarial podem ser melhor esclarecidas através de um exemplo. Mas antes de começar, é importante apresentar algumas definições com o intuito de estabelecer um vocabulário comum para o restante do texto.

- ***Processo (de negócio)***: Na acepção empregada no texto, *processo* representa todo e qualquer fluxo de trabalho concreto ou seu modelo, a ser gerenciado pelo sistema de *Workflow*.
- ***Workflow (WF)***: automação de um processo de negócio, por inteiro ou em parte, durante o qual informações, tarefas e documentos são passados de um participante para outro, respeitando um conjunto de regras procedurais. Poderíamos dizer que um *Workflow* é o resultado do casamento entre um *Processo de negócio* e o sistema de gerenciamento de *Workflows*.

- ***WorkItem***: chamamos de *WorkItem* uma atividade isolada em um *Workflow* qualquer. *Workitems* surgem à medida que o *Workflow* evolui e novas tarefas precisam ser cumpridas pelos diversos *agentes* participantes.

- **Agentes**: são chamados de agentes os *usuários* e *papéis*. *Papéis* são estruturas hierárquicas utilizadas na representação dos grupos de usuário. Cada usuário deve exercer pelo menos um papel; isto é, deve pertencer a pelo menos um grupo. Isso porque a distribuição das tarefas ou *workitems* entre os vários usuários é determinada, como se verá adiante, através do(s) papel(éis) ocupado(s) por cada um.

1.3 Caso de uso

Empresa de desenvolvimento de software:
Tomemos como exemplo uma empresa de desenvolvimento de software. Nessa empresa, como em muitas outras, há uma série de passos ou etapas que devem ser seguidos antes que se dê início a qualquer projeto.

Estas etapas são constituídas por diferentes atividades e exercidas por várias pessoas. Algo importante a ser notado é que essas etapas são interdependentes, isto é, as informações geradas em uma etapa são, em geral, utilizadas nas outras. O fluxo de informações tem, portanto, vital importância neste processo.

Basicamente, há o seguinte fluxo: um funcionário da empresa encontra-se com o cliente, com o intuito de determinar os requisitos do projeto. O funcionário tem como objetivo extrair aquilo que é realmente importante das informações que lhe foram passadas e, em seguida, entrar com os dados do projeto e do cliente no sistema interno da empresa.

Os dados coletados pelo funcionário irão, na próxima etapa, para as mãos de um projetista, que será responsável por desenvolver uma solução para o projeto, ou seja, fazer uma modelagem do sistema a ser desenvolvido.

Posteriormente, essa modelagem deverá ir para as mãos de outro funcionário, responsável por apresentá-la ao cliente, de um modo que seja fácil de entender e que permita ao cliente perceber se é isso o que ele realmente quer (pelo menos em teoria). Neste ponto, o cliente tem duas escolhas. Ou ele aceita o projeto, ou ele o rejeita. Caso ele aceite, esta informação será repassada ao gerente da empresa, que fará um contrato com o cliente e dará o aval para o início do desenvolvimento. Caso o cliente rejeite, esta informação também chega ao gerente, que decide então se o projeto volta para as mãos do projetista - para eventuais modificações - ou é descartado, terminando assim o processo. É possível notar, com o que foi visto até aqui, que houve um fluxo de trabalho entre os diversos participantes.

Um sistema gerenciador de *workflows* seria perfeito para esta empresa, pois ele tem justamente a função de mostrar para os participantes corretos quais atividades devem desempenhar e o momento certo em que devem agir. Além disso, o gerente poderia ter um controle completo sobre o andamento do projeto, sobre a produtividade de cada funcionário ou de um grupo de funcionários, fazer auditorias, encontrar possíveis gargalos no processo de produção, entre várias outras vantagens.

Suponha agora que a empresa deseje criar um novo departamento: controle de qualidade. Com a adição do novo departamento a modelagem passa a ser analisada, antes de ir parar nas mãos do funcionário que a apresenta ao cliente, pelo crivo de um funcionário do controle de qualidade. O que temos aqui é uma mudança no processo de produção da empresa. Um

sistema gerenciador de *workflows* tem também, como uma de suas características, tornar fácil este tipo de atualização e até mesmo a criação de novos processos.

2 Arquitetura do sistema

2.1 Visão geral

A arquitetura utilizada é baseada em *thin clients*, que impõem poucos requisitos aos clientes (apenas um navegador que suporte HTML). Essa arquitetura compreende 3 camadas fundamentais: camada de apresentação, camada de negócios e camada de dados. Os clientes comunicam-se apenas com a camada de apresentação. A lógica de negócios encontra-se no servidor, único que tem acesso à base de dados.

A escolha dessa arquitetura nos permitiu atingir alguns objetivos, tais como a facilidade para desenvolver um sistema distribuído, que fosse escalável, seguro e de alta disponibilidade, além de facilitar sua manutenção.

A opção por *thin clients* torna o sistema muito flexível, pois o acesso ao sistema não depende de um computador. O usuário pode, a princípio, acessá-lo de qualquer aparelho (como PDAs e celulares) que possua um navegador.

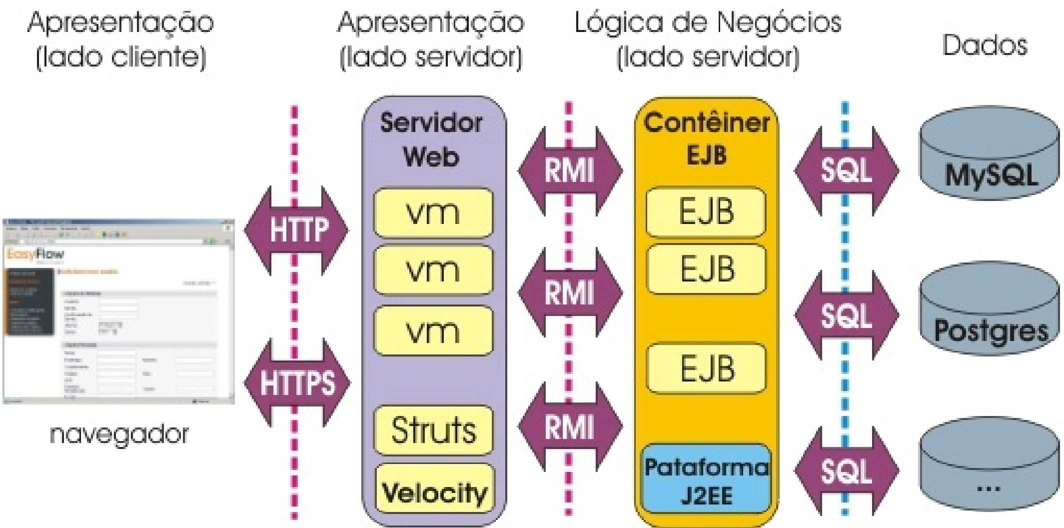

Figura 1: Arquitetura do sistema dividida em 3 camadas.

Nas 3 camadas foram utilizados softwares livres:

- servidor web: Tomcat;
- servidor de aplicação: JBoss;
- banco de dados: Postgres.

Optamos por um servidor de aplicação que implementasse a especificação J2EE pelos seguintes motivos:

- flexibilidade pelo fato de ser multiplataforma;

- é um padrão de fato, uma tecnologia muito disseminada;
- fornece serviços avançados de *middleware*;
- integra-se facilmente com outras tecnologias.

Nos próximos tópicos, cada camada será detalhada.

2.2 Camada de apresentação

A camada de apresentação é composta pelos seguintes itens:

- **controle de fluxo de atividades**: como o sistema possui uma interface web, é possível, a princípio, acessar qualquer página desde que se disponha de seu endereço. Isso porque não há, nos sistemas web convencionais, nada que force o usuário a acessar páginas numa ordem pré-definida. Esse comportamento constitui uma potencial falha de segurança do sistema, já que o controle de sessões é feito via servlets no nosso caso. O controle de fluxo vem, justamente, determinar quais são as seqüências de acessos que devem ser seguidas, forçando o usuário a seguir os passos corretos;
- **controle de acesso**: o acesso ao sistema só é permitido a usuários previamente cadastrados e que, portanto, possuem login e senha. Objetivando aumentar a segurança, utilizamos o protocolo https para a comunicação criptografada do cliente com o servidor. Assim, a senha fica protegida contra a interceptação de terceiros;
- **controle de visibilidade**: através desse controle é possível determinar o que o usuário pode ou não enxergar após ter acesso ao sistema, isto é, determina quais são os recursos do sistema que ficarão disponíveis ao usuário e quais ele não terá acesso;
- **desenho de processos (de negócio)**: o sistema fornece interfaces para o desenho de processos. O usuário que desenha os processos deve ter conhecimentos de Redes de Petri estendidas, como será abordado mais adiante.
- **controle de processo**: O sistema permite um nível de controle fino sobre instâncias de modelos de processos em execução, provendo acesso e permitindo a manipulação, em separado, de itens de trabalho individuais, de acordo com o *agente* responsável. Isso significa que é possível, através do sistema, obter uma visão detalhada da evolução dos processos em voga.

Para o desenvolvimento da camada de apresentação fez-se uso do modelo MVC (*Model-View-Controller*). Este modelo descreve a organização de uma aplicação em três módulos separados:

- ***Model*** **("Modelo")**: compreende o modelo de aplicação, contendo a representação de dados e lógica de negócios;
- ***View*** **("Apresentação")**: trata aspectos relacionados à apresentação e entrada de dados de usuário;
- ***Controller*** **("Controle")**: é responsável por despachar requisições e controlar seus fluxos.

Na literatura, o modelo MVC aplicado à apresentação é conhecido em duas versões que são chamadas de **Modelo 1** e **Modelo 2**. A arquitetura proposta no Modelo 1 diz que um navegador, por exemplo, tem acesso direto ao módulo de apresentação e é o módulo de apresentação que faz a comunicação direta com o módulo responsável pelo modelo. Já na arquitetura proposta no Modelo 2, há uma discussão sobre o módulo de controle que desempenha o papel descrito acima.

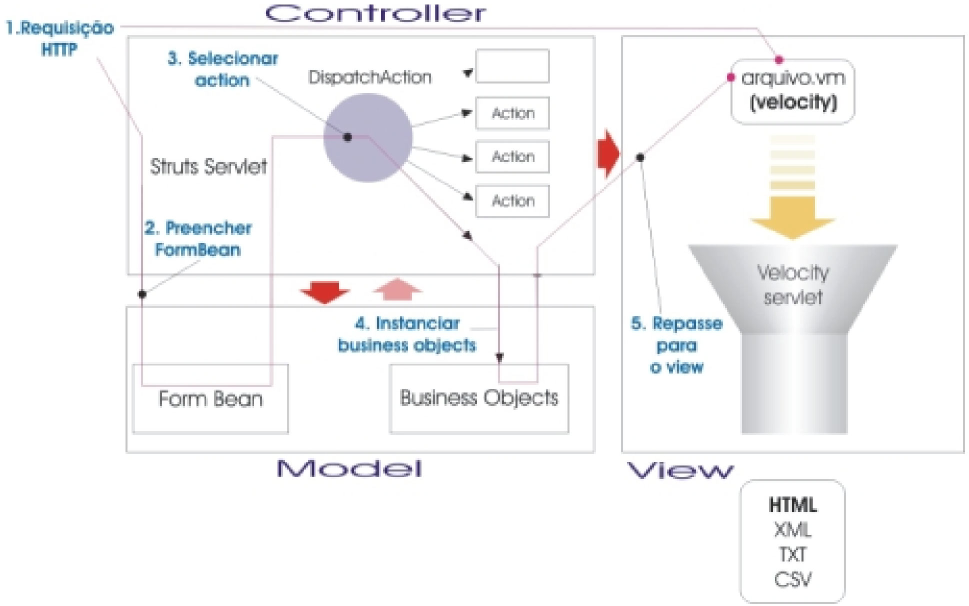

Figura 2: Modelo MVC 2 adaptado a este projeto.

Para a elaboração desse projeto, fez-se uso do Modelo 2. Através do modelo foi possível centralizar a lógica de despachar requisições na classe **DispatchAction**, facilitando a implementação do controle de segurança e acesso ao sistema.

Para facilitar o uso do Modelo 2, foi utilizado o *framework* ("arcabouço") Struts. O Struts é um *framework* de código aberto que fornece componentes de controle e interage com outras tecnologias de acesso à base de dados padrões como JDBC e EJB, além de interagir com várias ferramentas fornecidas por terceiros como Hibernate, iBATIS, ou Object Relational Bridge.

A princípio, o Struts também poderia ser utilizado como ferramenta para o módulo de apresentação. Entretanto, para que isso ocorresse, as páginas web deveriam ser implementadas utilizando-se JSP's (*Java Server Pages*), que apresentam a desvantagem de serem difíceis de serem depuradas.

Por este motivo, optou-se pelo uso da ferramenta de templates Velocity no módulo de apresentação. O Velocity possui uma linguagem de templates (*Velocity Template language* - VTL) simples, porém muito poderosa. Templates são arquivos que descrevem uma estrutura de formatação, contendo diretivas indicando valores a serem inseridos e ações a serem executadas pelo sistema.

2.2.1 Interface web

Houve uma preocupação particular com a interface, decorrente da potencial falta de experiência com informática esperada dos usuários finais desse tipo de sistema. Isso se traduziu em uma dose de cuidado extra na elaboração das telas, procurando torná-las tão simples e intuitivas quanto possível.

Vale notar que, ao mesmo tempo em que houve uma preocupação com facilitar a navegação, era de nosso interesse manter também o código HTML reutilizável - de maneira a facilitar a criação das novas páginas e manutenção das antigas. A complicação, neste caso, vem do fato de que nem sempre o *layout* de uma página é ideal para todos os tipos de dados que se deseja apresentar. A elaboração de um *layout* padrão não foi, portanto, uma tarefa fácil.

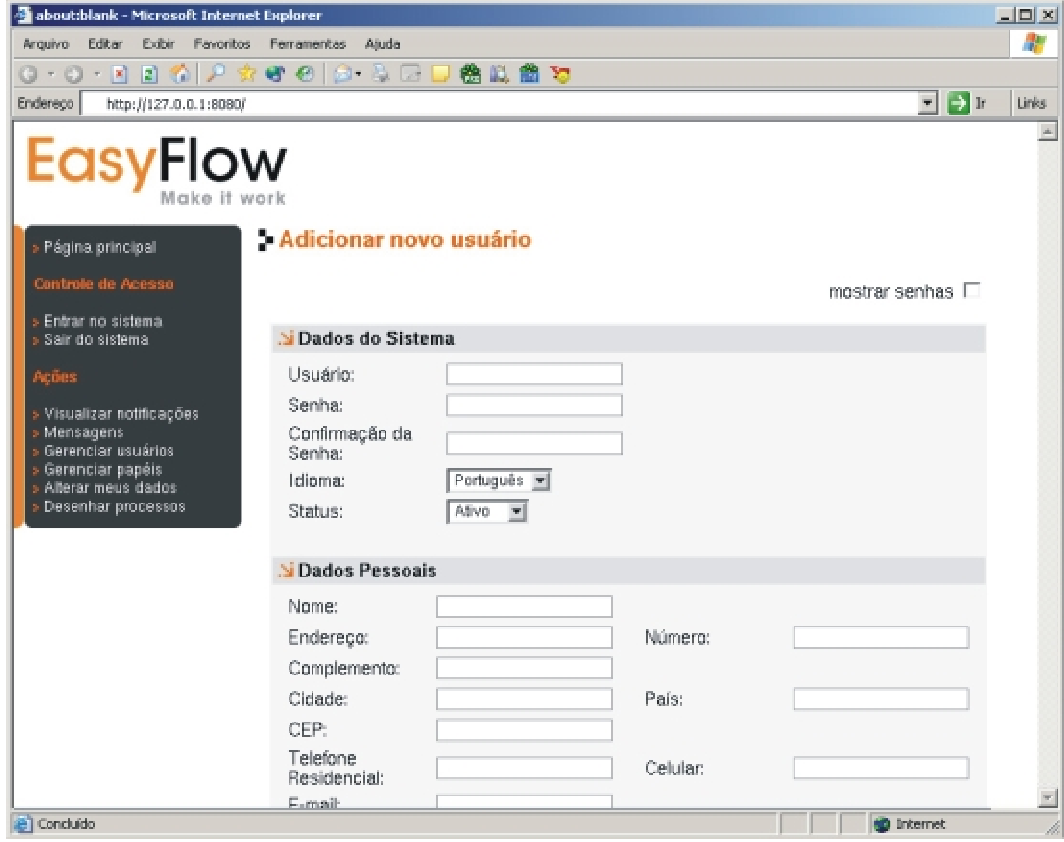

Figura 3: Tela de cadastro de usuários do sistema.

Uma das soluções adotadas - grande facilitadora do desenvolvimento das páginas - foi o controle de *layout*. Trata-se de uma extensão do Velocity, que está disponível na página do projeto Jakarta. Convencionalmente, um template possui diretivas que indicam inclusões de código de outras páginas (usadas para reutilização de código que são comuns a muitas páginas, como cabeçalho, rodapé e menu). Há porém uma desvantagem que resulta justamente da necessidade de repetir a diretiva em todas as páginas que a utilizam, dificultando dessa maneira futuras alterações. Na solução adotada, por outro lado, o arquivo de template contém apenas o corpo de uma página, que é inserido em uma segunda página (de *layout*) automaticamente pelo sistema de templates.

Sem a adoção do controle de *layout*, os arquivos de template (extensão vm) seriam dessa forma:

```
#include(inc/header.vm)
<div align="center">
<img src="/images/workflow.jpg">
</div>
#include(inc/footer.vm)
```

A diretiva *include*, utilizada tanto para os arquivos *header.vm* quanto para o *footer.vm*, deveria ser adicionada a todas as páginas do sistema. A função do *include* é adicionar o código de outros arquivos no arquivo onde a diretiva é utilizada.

Utilizando o controle de layout, a página acima ficaria assim:

```
<div align="center">
<img src="/images/workflow.jpg">
</div>
```

Quando a página for requisitada, o sistema de templates ficará encarregado de processá-la, inserindo-a na página de *layout*, no local da diretiva *$screen_content*:

```
#include(inc/header.vm)
$screen_content
#include(inc/footer.vm)
```

Uma outra dificuldade encontrada no desenho da interface esteve relacionada à montagem de um processo. O objetivo inicial era o de permitir ao usuário montar um processo da forma mais intuitiva possível. Dessa maneira, exigir desse usuário conhecimentos de Redes de Petri coloridas foi algo que nos pareceu, inicialmente, ir em desencontro com a simplicidade buscada. No entanto, encontrar um mapeamento entre um esquema simplificado e uma Rede de Petri que preservasse a expressividade do modelo provou ser um desafio formidável para o qual, infelizmente, não encontramos soluções aceitáveis.

Embora isso possa parecer um problema, notamos que a criação de um processo certamente não é uma tarefa rotineira. Além disso, pressupõe-se que projetistas de processos sejam pessoas razoavelmente qualificadas e, dado que as Redes de Petri coloridas são intuitivas em dose moderada, imaginamos que o custo de simplificar a representação seria maior que o benefício.

2.3 Camada de negócios

A camada de negócios compreende dois subconjuntos:

- **Extensões** englobando tudo aquilo que foi necessário para, partindo da biblioteca (Núcleo), construir um sistema completo de gerenciamento de workflow (interfaces para o controle de usuários, interface para uso de contêiners J2EE, mbeans para integração).
- Uma biblioteca genérica (**Núcleo**) para manipulação de Workflows que fornece facilidades de modelagem, verificação formal e um ambiente de execução para os modelos cujo mecanismo de auto-persistência é baseado na API JDO (RFS-12).

2.3.1 Núcleo

2.3.1.1 Introdução

O núcleo surgiu como conseqüência natural da filosofia de compartimentação de funcionalidades adotada no projeto. Foi formulado, inicialmente, como sendo a camada mais interna na arquitetura do sistema - seu papel seria o de prover funcionalidades básicas, tais como a criação, gerenciamento e atualização dos modelos de fluxo, através de primitivas pouco abstratas.

Logo no início do processo de modelagem, todavia, notamos ser o núcleo possuidor de uma grande vocação a biblioteca independente, já que fornece uma interface bem definida e um conjunto de primitivas genéricas, sendo completamente desacoplado do restante do sistema. Tais características permitiriam o seu uso em qualquer projeto que necessitasse de funcionalidades de workflows básicas ou simulações de fluxos (fluxos e workflows serão utilizados aqui como palavras equivalentes).

O núcleo provê, essencialmente, as seguintes funcionalidades:

- Interface para modelagem de fluxos, criação e instanciação de templates (metaclasses);
- facilidades que permitem acompanhar o estado dos workflows instanciados - incluem formas de determinar, a qualquer instante, quais as tarefas ativas e quais os participantes nelas envolvidos;
- controle centralizado de recursos, portando interfaces de acesso aos repositórios de modelos e participantes;
- controle interno de persistência através do JDO;
- comportamento transacional.

Além disso, existe uma coleção de beans de sessão (que cumprem o papel de fachadas), criados para uso exclusivo da biblioteca em ambientes gerenciados (contêiners J2EE, especificamente).

Conceitos básicos

Existem muitas maneiras conhecidas para representar workflows - a maior parte delas envolve grafos; a grande maioria cumpre bem os seus propósitos de modelagem. O problema encontrado em muitas dessas representações está, portanto, não nas representações em si, mas no fato delas serem heterogêneas e restritas a nichos ou grupos de desenvolvimento. Isso gera uma porção de dificuldades com relação às ferramentas de análise empregadas, que diferem significativamente de uma representação para outra. Além disso, certos modelos são mais próprios para determinados tipos de análise do que outros.

Ferramentas de análise são importantes na medida em que tornam possível avaliar um fluxo quanto a diversos aspectos qualitativos e quantitativos antes de colocá-lo em prática (evitando, potencialmente, prejuízos à empresa utilizando o sistema de workflows).

A escolha do nosso modelo interno de representação levou em conta, ainda, os seguintes aspectos:

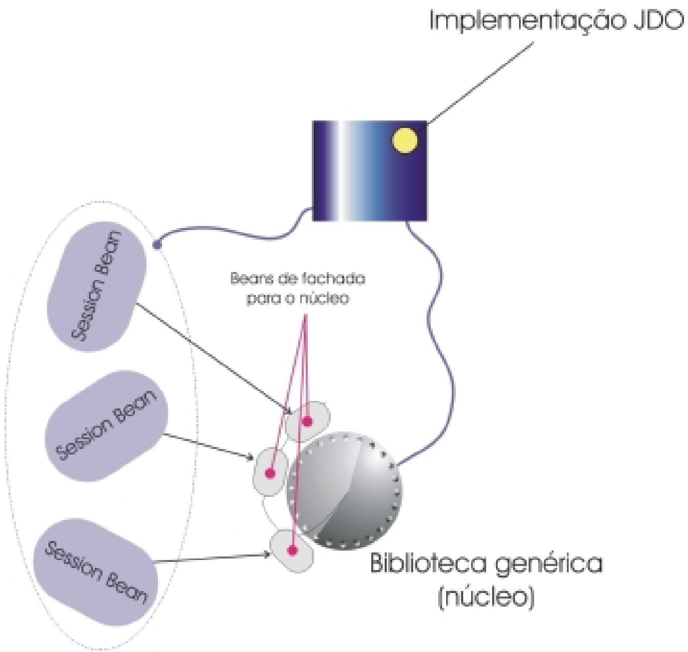

Figura 4: Núcleo e suas fachadas para uso em ambiente J2EE.

- **Manipulação** - o modelo não pode ser críptico. Um modelo muito complexo tem sua aplicabilidade condicionada à presença de profissionais qualificados. Gostaríamos de um modelo que fosse intuitivo.
- **Expressividade** - não adianta o modelo ser fácil de usar e possuir boas propriedades de análise se ele não for capaz de representar a diversidade de fluxos requerida nos ambientes aos quais a aplicação é voltada. Precisamos, portanto, de um modelo expressivo.
- **Propriedades** de análise - além de darmos preferência a modelos com boas propriedades de análise, foi necessário também separar quais dessas propriedades eram relevantes ao nosso usuário final esperado e quais não eram.

Após estabelecidas as metas, teve início um processo de pesquisa e seleção do modelo. A representação que melhor se enquadrou nos nossos requisitos foi a das *Workflow Networks*, ou *WF-nets*.

WF-Nets[3]

WF-Nets são, essencialmente, Redes de Petri acrescidas de algumas particularidades. De maneira semelhante às Redes de Petri clássicas, as *WF-Nets* possuem lugares, transições, arestas (com peso) e *tokens*.

Formalmente, temos:

Definição 1 (WF-Net): Uma Rede de Petri PN é uma *WF-Net* (*WorkFlow net*) se, e somente se:

1. PN tem dois lugares especiais: i e o, tais que i é uma fonte; isto é, $\bullet i = \emptyset$ e o é um sorvedouro; isto é, $o\bullet = \emptyset$.

2. Se adicionarmos uma transição t^* a PN que conecte o lugar i ao lugar o (i.e. $\bullet t^* = \{o\}$ e $t^*\bullet = \{i\}$), a Rede de Petri resultante será fortemente conexa.

Esta breve introdução não tem como finalidade discutir Redes de Petri clássicas. Uma discussão mais detalhada a respeito da teoria envolvida pode ser encontrada em [3]. Uma explicação pertinente, no entanto, diz respeito à maneira pela qual modelamos os construtos de roteamento e sincronização, mais especificamente os **AND-splits, AND-joins, OR-splits, OR-joins e XOR-joins**.

Optamos por permitir, nossas **WF-Nets**, que as arestas que partem de transições para lugares possuam, associadas a elas, expressões lógicas cujos átomos são propriedades. Essas propriedades assumem valores do tipo verdadeiro ou falso, que serão utilizados para avaliar a expressão associada. Durante o disparar de uma transição (onde *tokens* são consumidos do(s) lugar(es) de origem e produzidos no(s) lugar(es) de destino), apenas aquelas arestas cujas expressões forem determinadas verdadeiras produzem *tokens*.

Além disso, as arestas possuem também um peso (inteiro positivo) associado a elas. Esse peso determina quantos *tokens* são consumidos e/ou produzidos no disparar de uma transição. Temos, portanto:

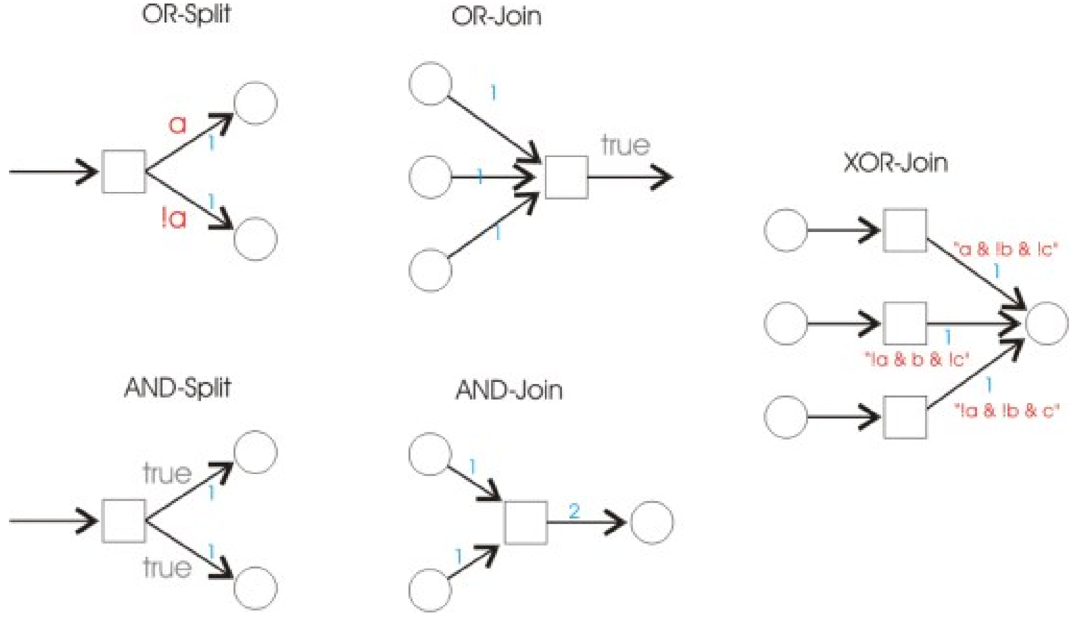

Figura 5: Construtos de roteamento.

É importante notar que os construtos **OR-Split** e **XOR-Join** dependem fortemente do valor assumido pelas propriedades associadas a eles. Isso pode ser visto como uma metáfora da cor do *token* nas Redes de Petri Coloridas - no nosso caso, ao invés do *token* possuir uma cor que determina o seu roteamento através da rede, temos propriedades externas "trocando a cor do *token*" a todo instante, conforme variam dinamicamente no curso de execução de um processo.

Vale ainda dizer que o construto **XOR-Join** pode ser modelado de inúmeras formas e que o seu correto comportamento depende da correta manipulação das propriedades conforme a evolução do sistema.

Segue abaixo um pequeno exemplo das WF-Nets sendo postas em prática na modelagem de um processo hipotético (empresa de gerenciamento de software):

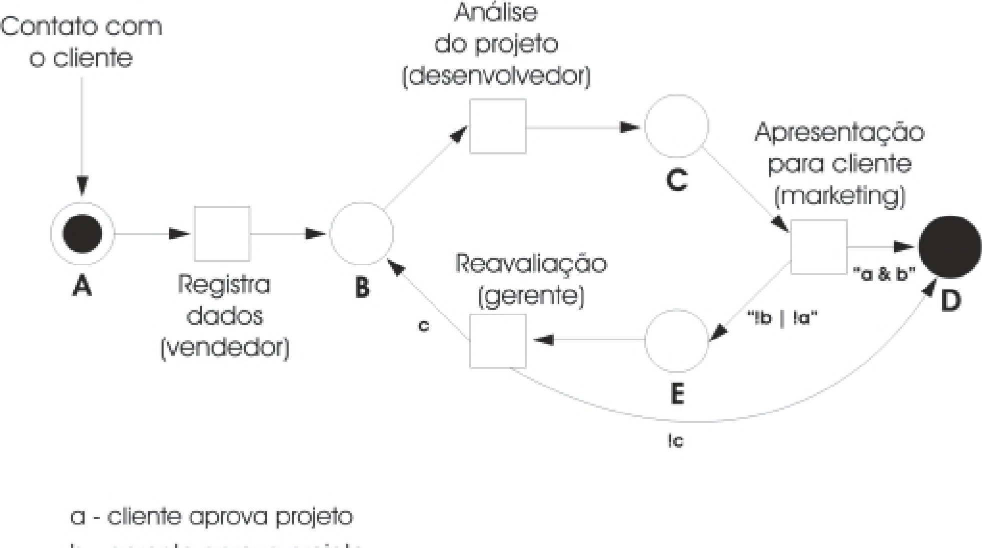

Figura 6: Um exemplo de uma WF-Net. Lugares são círculos, transições são quadrados.

Neste exemplo, um token que se encontra no lugar **C** irá passar, mediante uma transição, para o lugar **D**, se e somente se a expressão (**a**&**b**) for verdadeira; isto é, se **a** for verdadeira (cliente aprova projeto) e **b** for verdadeira (gerente aprova projeto). Caso contrário, se **!b** ou **!c** forem verdadeiros (isto é, o cliente **OU** o gerente não aprovam o projeto), a transição irá consumir o *token* em **C** e produzir um *token* em **D**. O estado da WF-Net num dado instante é representado pelo conjunto dos *tokens* e suas respectivas posições (em quais lugares eles se encontram). Quanto às transições, podem ser disparadas tanto por ação do usuário (por exemplo, o usuário completa um item de trabalho como o apresentado na transição "registra dados") quanto pelo expirar de um timer (timeouts). Para o núcleo isso não faz diferença, todas as entidades que participam de interações com as WF-Nets são representadas através de recursos (resources). Recursos podem representar usuários, grupos ou até mesmo timers e aplicações - o significado é dado pelas camadas mais externas. Como nas Redes de Petri clássicas, transições (na verdade workitems, como veremos mais adiante) só são ativadas mediantes à presença de um número suficiente de *tokens* nos lugares conectados a ela em leque de entrada (fan-in). Por outro lado, contrário às Redes de Petri clássicas, transições não são disparadas aleatoriamente - requerem, como descrito acima, a participação de um terceiro elemento, uma resource. Uma resource deve possuir permissão para disparar uma dada transição. Segue abaixo o diagrama de transição de estados para um item de trabalho (workitem), cujo papel é associar as transições (dadas num template) a uma dada instância de um template de fluxo de trabalho. Isso quer dizer que, sempre que uma transição for ativada numa dada instância de um processo (template), será gerado um workitem (ou talvez mais de um) relacionado àquela transição, para que o usuário competente possa cumprí-lo.

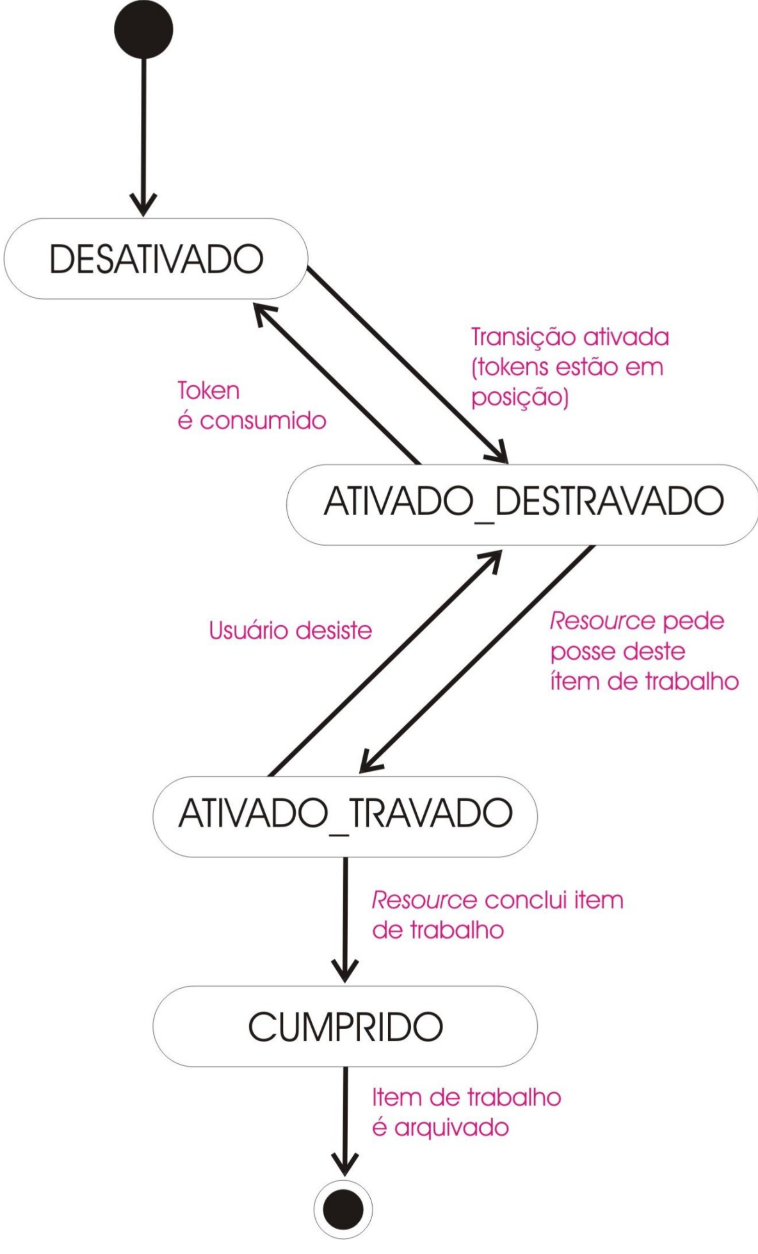

Figura 7: possíveis estados para um item de trabalho

O núcleo é, portanto, o responsável por criar, manter e fornecer acesso a todas essas informações.

2.3.1.2 Pequena documentação

Esta seção procura expor, de maneira sucinta e objetiva, o projeto e o estado atual de desenvolvimento do núcleo. A ênfase será dada na API e no modelo de classes, majoritariamente através da discussão de exemplos.

Geral

O núcleo está localizado no pacote **br.com.easyflow.engine** e é composto por quatro subpacotes, sendo o subpacote **core.managers** o que agrega os componentes mais relevantes (o restante dos pacotes contêm interfaces, classes utilitárias e ferramentas). O diagrama de classes abaixo revela a porção mais significativa do desenho:

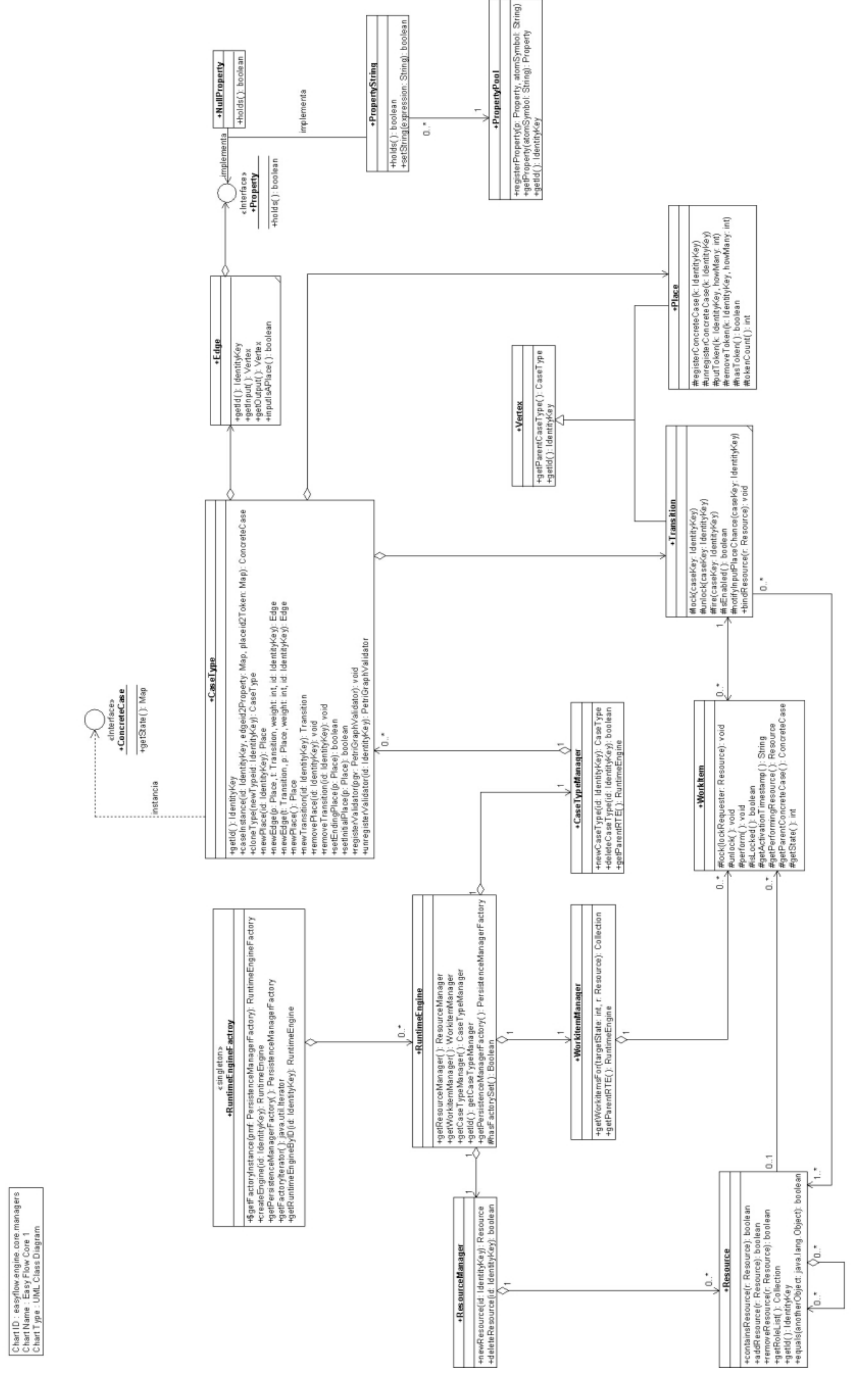

Figura 8: UML do subpacote **core.managers**

O modelo foi organizado de acordo com uma estrutura hierárquica de classes gerentes ou managers, apresentando fachadas e uma cadeia de fábricas que se distribuem sobre múltiplos participantes. A vantagem dessa divisão vem sob a forma de classes reduzidas e de uma compartimentação natural dos métodos de acordo com a sua funcionalidade, resultando numa API mais simples e mais intuitiva.

Uma outra razão importante para a divisão hierárquica está relacionada ao conceito de persistência por transitividade de um modelo de objetos. Dizemos que uma API ou um arcabouço de persistência implementa persistência por transitividade quando, ao persistir um objeto, o arcabouço persiste também o fecho transitivo de todas as referências alcançáveis partindo-se de. Cabe aqui uma digressão interessante a respeito do uso de arcabouços e **API's** de persistência de modelos de objetos. Embora sejam ferramentas cujo objetivo maior é livrar o programador do fardo dos mapeamentos objeto-relacionais, muitas das especificações e softwares presentes no mercado apresentam falhas que não permitem seu uso enquanto forma de persistência transparente dos modelos (isto é, os mapeamentos objeto-relacionais podem necessitar de alguns ajustes que derrubam a transparência do mapeamento). Com o passar do tempo, todavia, os programadores adquirem o mau-hábito de inserir recortes de código para manipulação direta da base de dados relacional, porque julgam ser mais fácil ou simplesmente por costume. Embora isso possa parecer algo inócuo, é importante notar que viola a filosofia do trabalho orientado a objetos e, com o passar do tempo, pode gerar código tão ou até mais difícil de manter do que sem o uso da API ou arcabouço de persistência de modelo de objetos. Tendo isso em mente, procuramos desenvolver um modelo que nos permitisse utilizar o arcabouço de persistência como único meio de persistência; isto é, sem enxertos de código para manipulação da base de dados. O **JPOX**, implementação de código aberto da especificação **JDO 1.0**, serviu bem aos nossos propósitos, implementando persistência por transitividade e recuperação automática de referências. Tendo tudo isso em mente, fica fácil inferir o quanto uma estrutura hierárquica colabora com o nosso intuito:

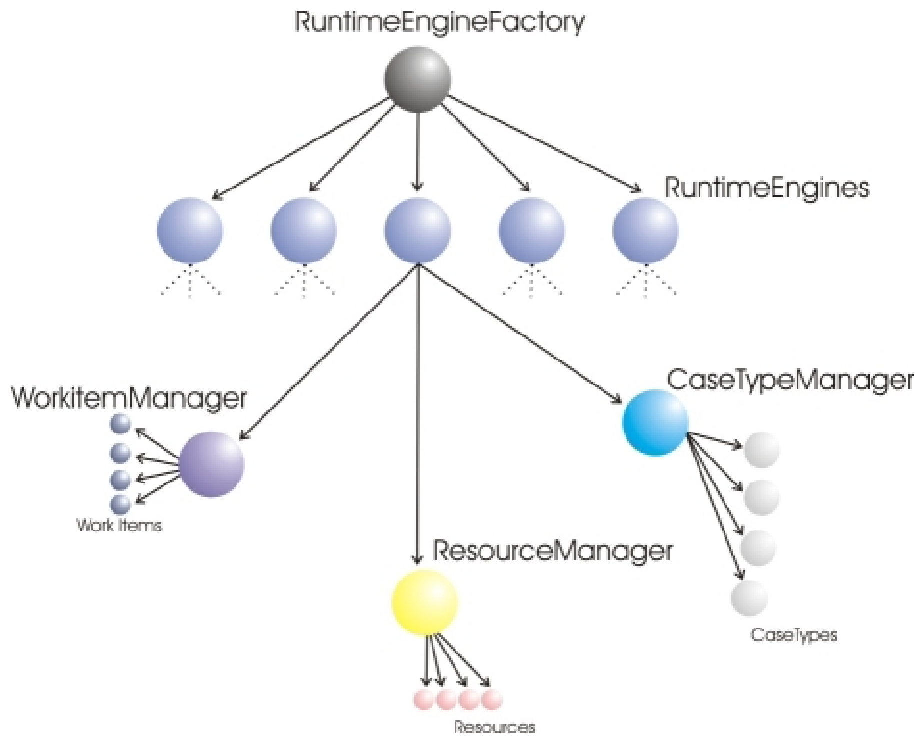

Figura 9: Hierarquia no modelo de objetos

A estrutura que lembra uma árvore permite a persistência transparente de todo o modelo de objetos através de uma única chamada da **API** que torna persistente a instância única de **RuntimeEngineFactory**. O restante dos objetos são persistidos por transitividade, de forma transparente. Obviamente o modelo foi concebido de maneira a existir sempre um caminho entre a instância única de **RuntimeEngineFactory** e qualquer instância do modelo de objetos.

Para reduzir ainda mais a preocupação com persistência e poupar o usuário da biblioteca do fardo de ter de conhecer a **API JDO** [4], a instância única de **RuntimeEngineFactory**, que implementa uma semântica de singleton enfraquecida (como veremos mais adiante), requer apenas que sejam passados ao seu método getFactoryInstance (fig. 10) alguns parâmetros bem documentados que dizem respeito à configuração da base de dados (é razoável supor que o usuário da biblioteca detenha tais informações), sob a forma de um objeto da classe PersistenceManagerFactory.

RuntimeEngineFactory

Como mencionamos anteriormente, a classe **RuntimeEngineFactory** implementa a semântica de um singleton enfraquecida:

«singleton»
+RuntimeEngineFactroy

+$getFactoryInstance(pmf: PersistenceManagerFactory): RuntimeEngineFactory
+createEngine(id: IdentityKey): RuntimeEngine
+getPersistenceManagerFactory( ): PersistenceManagerFactory
+getFactoryIterator( ): java.util.Iterator
+getRuntimeEngineByID(id: IdentityKey): RuntimeEngine

Figura 10: RuntimeEngineFactory

Essa semântica de singleton enfraquecida se dá por meio de uma associação feita entre uma base de dados e uma instância do singleton. Isso significa, para fins práticos, que é permitida no máximo uma instância da **RuntimeEngineFactory** por base de dados (daí a semântica ser enfraquecida). Como a cada base de dados está associada uma única **PersistenceManagerFactory**, que é uma classe JDO que pode ser instanciada partindo-se da configuração da base de dados, para recuperar todas as RuntimeEngines guardadas em uma base de dados basta chamar o método estático getFactoryInstance tendo como parâmetro a **PersistenceManagerFactory** para essa base de dados:

```
Properties properties = new Properties();


// Propriedades da base de dados e da implementação JDO
properties.setProperty("javax.jdo.PersistenceManagerFactoryClass",
"com.triactive.jdo.PersistenceManagerFactoryImpl");

properties.setProperty("javax.jdo.option.ConnectionDriverName",
"com.mysql.jdbc.Driver");

properties.setProperty("javax.jdo.option.ConnectionURL",
"jdbc:mysql//localhost:3306/giuliano");

// Gera a PersistenceManagerFactory a partir das propriedades da
// base de dados.
PersistenceManagerFactory pmfactory =
JDOHelper.getPersistenceManagerFactory(properties);

// Obtém o singleton da RuntimeEngineFactory para esta base de dados
RuntimeEngineFactory ref =
RuntimeEngineFactory.getFactoryInstance(pmfactory);
```

Note que, no conjunto de propriedades que configura a **PersistenceManagerFactory**, existe uma chave de nome **javax.jdo.PersistenceManagerFactoryClass**. O valor que

faz par com essa chave é quem determina qual implementação **JDO**, das implementações disponíveis, será utilizada. A **PersistenceManagerFactory** é criada então pela classe **JDOHelper**, que é distribuída pela própria Sun Microsystems e não faz parte de nenhuma implementação.

Qualquer implementação **JDO** que siga o padrão 1.0 deve funcionar sem maiores problemas com o núcleo. No entanto, embora possível e certamente permitido, não é aconselhado misturar implementações, ao menos não numa mesma base de dados.

Uso da API

Agora que já discutimos como obter a instância da **RuntimeEngineFactory**, é chegada a hora de explorar um pouco melhor a API fornecida pelo núcleo. A **RuntimeEngineFactory** permite ao usuário criar uma nova **RuntimeEngine**, que é o objeto que serve como ponto de entrada para as fachadas internas.

A partir da **RuntimeEngine**, podemos acessar os três gerentes:

- **ResourceManager** - Gerente de recursos (atores e grupos). É através do ResourceManager que novos recursos podem ser criados e os pré-existentes destruídos.
- **CaseTypeManager** - Gerente de CaseTypes. Permite criar e destruir CaseTypes.
- **WorkitemManager** - Gerente de workitems. Fornece uma interface para a recuperação de workitems associados a um dado recurso.

Com apenas esses três objetos, é possível criar e destruir grupos e usuários, criar novos processos e determinar as tarefas de qualquer grupo ou usuário em qualquer instante. No caso de um trabalho teórico, seria necessário discorrer sobre os conceitos em abstrato. Felizmente, por este texto tratar da documentação de uma implementação, podemos dar um exemplo antes de seguir adiante.

Exemplo 1:

```
// 1 - Cria uma RuntimeEngine
IdentityKey engineKey1 = new StringIdentityKey("Engine 1");

RuntimeEngine rte = rtf.createEngine(engineKey);

// 2 - Cria dois recursos participantes, o primeiro representa um
// grupo, o segundo representa um usuário.
ResourceManager rm = rtf.getResourceManager();

Resource r1 = rm.newResource(new StringIdentityKey("Users group");
Resource r2 = rm.newResource(new StringIdentityKey("Giuliano Mega");

// Adiciona o usuário ao grupo.
r1.addResource(r2);

// 3 - Cria uma WF-Net simples.
```

```
CaseTypeManager ctm = rte.getCaseTypeManager();

CaseType ct1 = ctm.newCaseType(new StringIdentityKey("Linear"));

Place a = ct1.newPlace(new StringIdentityKey("a"));
Place b = ct1.newPlace(new StringIdentityKey("b"));
Place c = ct1.newPlace(new StringIdentityKey("c"));

Transition t1 = ct1.newTransition(new StringIdentityKey("A"));
Transition t2 = ct1.newTransition(new StringIdentityKey("B"));

t1.bindResource(r1);

StringIdentityKey edge1 = new StringIdentityKey("<a,A>");
StringIdentityKey edge2 = new StringIdentityKey("<A,b>");
StringIdentityKey edge3 = new StringIdentityKey("<b,B>");
StringIdentityKey edge3 = new StringIdentityKey("<B,c>");

ct1.newEdge(a, A, 1, edge1);
ct1.newEdge(A, b, 1, edge2);
ct1.newEdge(b, B, 1, edge3);
ct1.newEdge(B, c, 1, edge4);

ct1.setInitialPlace(a); // Lugar onde se "inicia" o processo
ct1.setEndingPlace(c);  // Lugar onde se encerra o processo

// 4 - Registra o validador padrão para esta WF-Net
ct1.registerValidator(new DefaultPetriGraphValidator());

// 5 - Cria um conjunto de propriedades
Property p1 = new BooleanProperty(new StringIdentityKey("p1"), false);
Property p2 = new BooleanProperty(new StringIdentityKey("p2"), false);
Property p3 = new BooleanProperty(new StringIdentityKey("p3"), true);

PropertyPool pp = new PropertyPool();
pp.registerProperty(p1, "a");
pp.registerProperty(p2, "b");
pp.registerProperty(p3, "c");

PropertyString ps = new PropertyString("!(a | b) & c", pp);

Map propertyMap = new HashMap();

propertyMap.put(edge1, ps);

// 6 - Cria um mapa que representa um estado possível para a WF-Net
Map tokenMap = new HashMap();
```

```
tokenMap.put(a.getId(), new Integer(1));

// 7 - Antes de instanciar o processo, é necessário validá-lo
ct1.validate();

// 8 - Instancia o processo, sendo que o estado inicial desta instância
// é determinado pelo mapa de Tokens e pelo mapa de Propriedades.
ConcreteCase caseInstance = ct1.getCaseInstance(
new StringIdentityKey("Process 1"), propertyMap, tokenMap);
```

Uma discussão do exemplo acima será bastante elucidativa com relação à organização da API.

Inicialmente, cria-se uma **RuntimeEngine** para que seja possível acessar os três gerentes, como discutido anteriormente. Obviamente não é necessário criar uma **RuntimeEngine** sempre que se desejar obter acesso aos três gerentes já que, na maior parte dos casos, o que se deseja fazer é recuperar uma **RuntimeEngine** pré-existente.

Neste caso, podemos substituir a primeira parte por:

RuntimeEngine rte = rtf.getRuntimeEngineByID(new StringIdentityKey(”Engine 1”));

Que recupera a **RuntimeEngine** cuja chave é ”Engine 1”(fica a cargo do usuário guardar qual a chave de sua RuntimeEngine, ou ele pode usar um **FactoryIterator**, que não será documentado aqui).

O exemplo prossegue criando dois recursos, r1 e r2 que, embora sejam tratados pelo núcleo de maneira indiferenciada, representam entidades semanticamente distintas. Isso porque o padrão composite, solução de design bastante elegante para casos como este, permite tais construções:

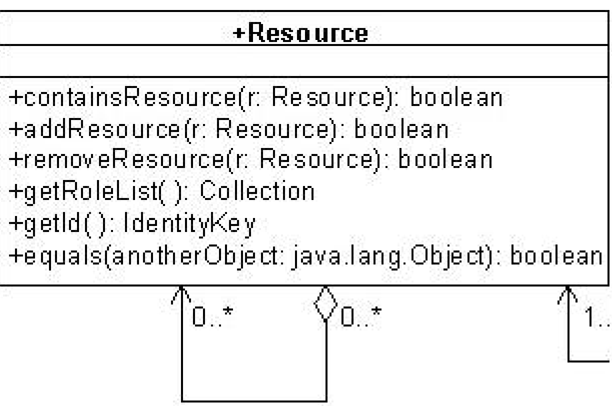

Figura 11: A classe **Resource**.

Usuários passam a ser tratados, portanto, sob a forma de grupos vazios. Dessa maneira, os usuários podem ser incluídos em múltiplos grupos e os grupos podem ser incluídos em outros grupos, tudo isso por meio de um modelo bastante simplificado.

Um objeto da classe **Resource** é capaz de informar quais são os grupos do qual ele participa através do método getRoleList, característica essencial para o controle de permissões

no sistema de gerenciamento de workflows. Além disso, é possível determinar se um dado usuário participa ou não num dado grupo através do método containsResource (que, embora seja redundante, facilita bastante a vida do programador).

Nas linhas que seguem à criação dos novos recursos e organização dos grupos (seção 3, "Cria um WF-Net simples") é demonstrado o uso da API de modelagem do núcleo. É importante notar que os objetos da classe **CaseType** incorporam o conceito de metaclasse; isto é, cada instância de **CaseType** possui um valor semântico comparável ao de uma classe estática. A vantagem do modelo orientado a instâncias é que as (meta)classes podem ser criadas dinamicamente, no nosso caso por meio da interface do próprio **CaseType**.

Conceitualmente, **CaseType** incorpora algumas das idéias utilizadas num modelo adaptativo de objetos ou AOM. Mais especificamente, a classe **CaseType** e a interface **ConcreteCase** constituem uma aplicação de uma variante do padrão **TypeSquare** [6]. Isso porque um objeto do tipo **CaseType**, sendo uma metaclasse, é capaz de gerar metaobjetos ou simplesmente objetos (metaobjetos e objetos não são radicalmente diferentes). Como nas linguagens orientadas a objeto convencionais, existe um vínculo fundamental que se forma entre os objetos de uma classe e a própria classe, sendo que as informações que não variam entre instâncias individuais são centralizadas na metaclasse.

Essa centralização, no nosso modelo, se dá por meio do uso do padrão flyweight. A informação intrínseca, no nosso caso, é dada, por exemplo, pela estrutura do grafo, pelos identificadores de arestas e pelos bindings de recursos; já a informação extrínseca é proveniente do estado de cada uma das instâncias num dado instante. A aplicação do padrão faz com que não seja necessário replicar a informação intrínseca, que é razoavelmente extensa, em todas as metainstâncias, de uma maneira similar às linguagens orientadas a objeto - a implementação binária dos métodos, por exemplo, não é replicada em cada objeto instanciado. Uma discussão do padrão flyweight, informações intrínsecas e extrínsecas pode ser encontrada em [10].

A porção do exemplo que modela o grafo é razoavelmente auto-explicativa - são apenas chamadas aos métodos de **CaseType** com o intuito de compor um grafo dotado de lugares, transições e arestas (uma WF-Net). Note que a distribuição inicial dos *tokens* não é parte do **CaseType**, que pode ter múltiplas instâncias, cada qual com a sua distribuição própria de *tokens*. Também não faz parte do CaseType o conjunto de objetos que descreve as propriedades associadas a cada aresta, isso porque pode haver casos em que se deseja um conjunto disjunto de propriedades associado a cada instância do **CaseType**.

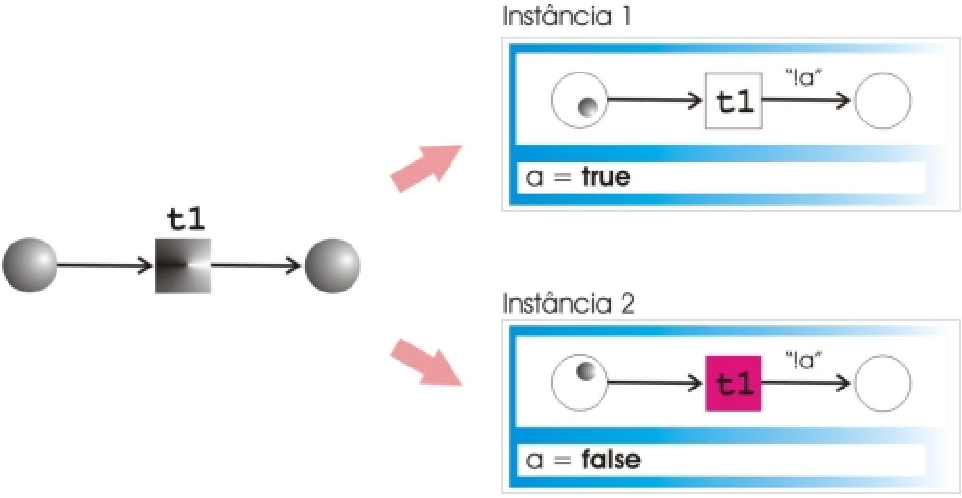

Figura 12: Conjuntos de propriedades separados por instância.

Na figura acima, a transição **t1** encontra-se ativa somente na **Instância 2**. Na **Instância 1**, em virtude do valor da expressão "!a"associada à aresta que parte de **t1** ser falso, há um bloqueio à passagem dos *tokens*, inviabilizando a transição.

Com relação às propriedades que podem ser associadas a cada aresta, o modelo é bastante flexível - basta que uma determinada classe implemente a interface **Property** para que seus objetos possam ser utilizados como propriedades. Com relação aos exemplos dados, a nossa famosa expressão lógica é um exemplo de tal classe:

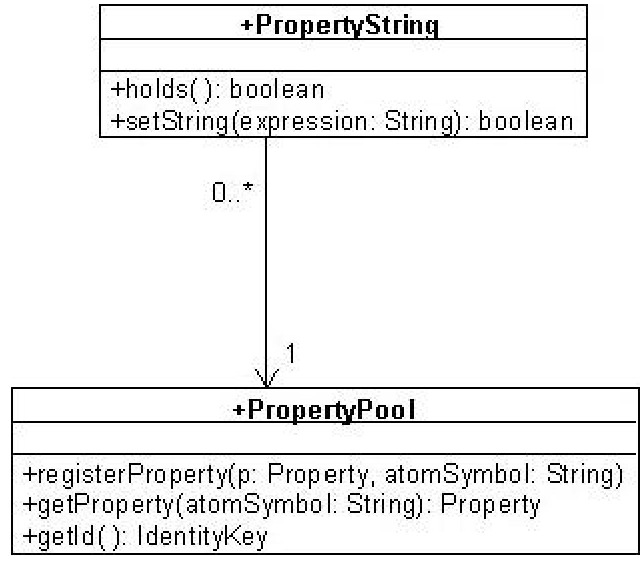

Figura 13: **PropertyString** e **PropertyPool**

Aqui, temos a classe **PropertyString** que implementa um parser de expressões lógicas e se utiliza do repositório de símbolos em **PropertyPool** sempre que precisa resolver um símbolo num átomo. Em outras palavras, na string "!a", a classe **PropertyString** iria, no momento de avaliar a expressão, buscar o objeto que implementa a interface **Property** no

repositório de símbolos **PropertyPool** a ele associado. A partir daí, **PropertyString** iria determinar o valor de a e então finalmente determinar o valor de **!a**.

Seguindo adiante no exemplo vemos um exemplo de tal procedimento. As linhas:

```
PropertyPool pp = new PropertyPool();
pp.registerProperty(p1, "a");
pp.registerProperty(p2, "b");
pp.registerProperty(p2, "c");

PropertyString ps = new PropertyString("!(a | b) & c", pp);

Map propertyMap = new HashMap();

propertyMap.put(edge1, ps);
```

Criam um **PropertyPool**, registram nele as propriedades p1, p2 e p3 sob os símbolos **a, b** e **c**, criam um **PropertyString** (cuja string de expressão é "!(**a**|**b**)&**c**"e associam o **PropertyPool** à **PropertyString**. O **HashMap** propertyMap contém associações entre chaves de arestas do **CaseType** e classes que implementam a interface **Property**. No momento em que o **CaseType** ct1 for instanciado, esse mapa, juntamente com um mapa que descreve a configuração inicial dos *tokens* (vide o exemplo), serão os responsáveis por definir os objetos-propriedade de cada aresta e o estado inicial da instância do processo.

Afora o comentado, restam quatro linhas no exemplo que não foram mencionadas:

```
ct1.setInitialPlace(a); // Lugar onde se "inicia" o processo
ct1.setEndingPlace(c);  // Lugar onde se encerra o processo

...

ct1.registerValidator(new DefaultPetriGraphValidator());

...

ct1.validate();
```

As duas primeiras linhas determinam o lugar inicial e final do processo. O lugar inicial sempre começa com um *token*, independentemente do mapa de distribuição de *tokens* passado como parâmetro (exceto no caso do usuário declarar explicitamente que o lugar inicial deve ter 0 *tokens*). Idealmente, processos novos são acompanhados de um mapa de distribuição de *tokens* vazio e apresentam apenas um *token* no lugar inicial. O recurso do mapa de *tokens* existe para permitir ao usuário instanciar um processo num ponto específico de sua evolução, mas tecnicamente é mais correto usar o lugar inicial (ver [3]). O lugar inicial deve sempre ser uma fonte.

O lugar final, por sua vez, é o lugar onde os *tokens* param. Uma vez que os *tokens* estejam todos no lugar final, a instância de um processo (ou **CaseType**) termina. O lugar final deve obrigatoriamente ser um sorvedouro.

A terceira linha registra o algoritmo de validação padrão no **CaseType**. Essa linha foi incluída por motivos meramente didáticos, uma vez que, por default, todos os **CaseTypes** contêm o **DefaultPetriGraphValidator** registrado na sua lista de algoritmos de validação. O algoritmo de validação padrão faz algumas checagens simples, tais como determinar se o grafo é conexo ou se o lugar inicial é uma fonte e o lugar final um sorvedouro. Análises mais sofisticadas (ausência de loops infinitos e deadlocks, por exemplo) podem ser conduzidas por outros algoritmos, basta que sejam registrados no **CaseType** através do método registerValidator.

A quarta e última linha executa a validação do **CaseType**. Note que instâncias de um **CaseType** só podem ser criadas se a etapa de validação for completada com sucesso (se o usuário tentar criar uma instância antes de validar um **CaseType**, uma exceção do tipo **WFUserException** será lançada).

Encerramos esta seção com um segundo exemplo, bastante simples quando comparado ao primeiro, que ilustra como uma aplicação do tipo do sistema de gerenciamento de workflows poderia, após ter autenticado um usuário, determinar quais tarefas estão disponíveis naquele instante para aquele usuário.

Exemplo 2:

<Suponha que exista uma referência para a **RuntimeEngine** correta numa variável de nome **rte** e que ao usuário autenticado corresponda uma instância da classe **Resource** de nome **user**>

```
WorkitemManager wim = rte.getWorkitemManager();
Collection worklist = wim.getWorkitemsFor(user);
```

A partir da coleção retornada por getWorkitemsFor, é possível determinar a quais tarefas correspondem cada workitem e oferecê-las ao usuário da maneira mais conveniente.

Conclusão

O núcleo é uma biblioteca que, embora desenvolvida com enfoque nas necessidades de workflow do sistema de gerenciamento de workflows, atua como camada independente e pode fornecer serviços básicos de workflow a qualquer aplicação que deles necessitem. A auto-persistência é uma característica intrínseca da biblioteca, que também concentra toda a lógica de modelagem e a estrutura de evolução dos processos.

Faltou uma discussão um pouco mais detalhada a respeito de alguns outros tópicos - tais como a integração da biblioteca com servidores de aplicação e até mesmo de alguns aspectos interessantes da própria **API** - mas a discussão dos exemplos e a **UML** são suficientes para que se tenha uma boa idéia de qual o papel do núcleo.

O código se encontra, para fins práticos, pronto; mas não funciona corretamente. Isso porque existem algumas limitações na implementação **JDO** escolhida que, embora consideradas desimportantes de início, acabaram promovidas a limitações críticas posteriormente. Tais limitações emperraram seriamente o nosso progresso - para que o código possa ser posto

em uso, é necessário reescrever uma boa parte das classes, especialmente aquelas que usam mapas e sets persistentes (as implementações **JDO** de código aberto são bastante limitadas com relação ao uso dessas interfaces).

2.3.2 Extensões

Embora o núcleo forneça um conjunto de interfaces adequadas para a manipulação e controle de fluxo em Redes de Petri, trata-se apenas de uma biblioteca - um sistema de gerenciamento de Workflows empresarial certamente requer mais do que um punhado de métodos de manipulação. Isso significa que existem inúmeros requisitos funcionais importantes que não fazem parte da lógica de processos, tais como interfaces gráficas, funcionalidades de cadastro e autenticação de usuários, acesso via WEB e facilidades para ambientes distribuídos. A essas funcionalidades extra-núcleo demos o nome de extensões.

A porção das extensões residente na camada de negócios é compreendida pelos seguintes:

1. **Extensões semânticas**

   Muitas das abstrações fornecidas pelo núcleo possuem uma semântica incompleta ou reduzida que, embora seja suficiente para os propósitos da biblioteca, é insuficiente num caso mais específico como o do sistema de gerenciamento de workflows empresarial. Tomemos, como exemplo dessa diferença, a semântica de usuários e grupos. Embora ambos sejam tratados indiferenciadamente pelo núcleo, essas entidades diferem significativamente quando inseridas no contexto de um sistema de cadastro - papéis não possuem endereços nem telefones; já usuários podem pertencer a papéis mas não a outros usuários.

2. **Extensões funcionais**

   As extensões funcionais fazem-se necessárias na medida em que o núcleo torna-se incapaz de prover determinados mecanismos essenciais ao nosso sistema em particular. Note que essa deficiência do núcleo se apresenta não como uma deficiência de projeto, mas como parte da própria filosofia adotada. Isso dito, podemos identificar como extensões funcionais, por exemplo, o sistema de controle e gerenciamento de papéis e usuários (cadastro e formas de acesso diferenciadas), o sistema de troca de mensagens (fornece um canal de comunicação entre usuários do sistema), o subsistema de persistência (que não é nada mais que uma implementação *opensource* do JDO), entre outros.

3. **Modelos da interface**

   Como mencionado anteriormente, nosso sistema é baseado no modelo MVC de integração aplicação-interface (*Model-View-Controller*). O gerenciamento e armazenagem dos modelos (*Model*) fica a cargo da camada de negócios.

   Um problema que surge nesta aplicação em específico é a necessidade de se criar um grande número de interfaces dinamicamente. Essa necessidade introduz um fator de complicação a mais na estrutura do modelo, que deixa de ser estático.

   É necessário também que haja uma maneira descomplicada de especificar o modelo e o fluxo de controle - idealmente, eles devem ser gerados automaticamente a partir da especificação da interface, podendo ser modificado nos casos onde isso for necessário (seguindo a filosofia “as coisas simples são simples de se fazer e as difíceis são possíveis”).

Um outro problema, talvez mais sério do que os outros dois, é a necessidade de criar novos *widgets* dinamicamente. Note que *criar* um novo *widget* é diferente de *instanciar* um *widget* pré-existente - o trabalho de criar é feito no nível de criar uma classe, enquanto que instanciar é simplesmente obter um objeto de uma classe pré-existente.

A estratégia adotada foi baseada num padrão de modelo adaptativo, como feito no núcleo. Essencialmente as metaclasses do núcleo sofrem uma extensão semântica e passam a contar com seqüências de telas associadas às transições, que são instanciadas também como metaobjetos em conjunto com as porções provenientes do núcleo.

Cabe aqui uma definição um pouco mais clara a respeito do que são *telas*, pela perspectiva da camada de negócios. *Telas* são composições de modelos (ou *value holders* na nomenclatura *Smalltalk*) de *widgets* (tais como áreas de texto, *combo boxes*, ou botões de rádio) que podem ser exibidos seqüencialmente pela camada de apresentação ao **iniciar** de um workitem.

Tanto as informações sobre quais *widgets* serão apresentados em uma tela, quanto os dados entrados no sistema pelos usuários são transferidos entre as camadas de apresentação e a de negócios encapsulados em objetos. Este padrão é conhecido como *Data Transfer Object* ("Objeto de Transferência de Dados"), também chamado de **DTO**, e permite uma comunicação eficiente entre as camadas, pois todos os dados a serem transferidos são encapsulados em um objeto, evitando a realização de diversas chamadas remotas a uma outra camada para obtenção dos mesmos.

Segue abaixo o diagrama de classe da porção mais significativa das extensões semânticas e funcionais:

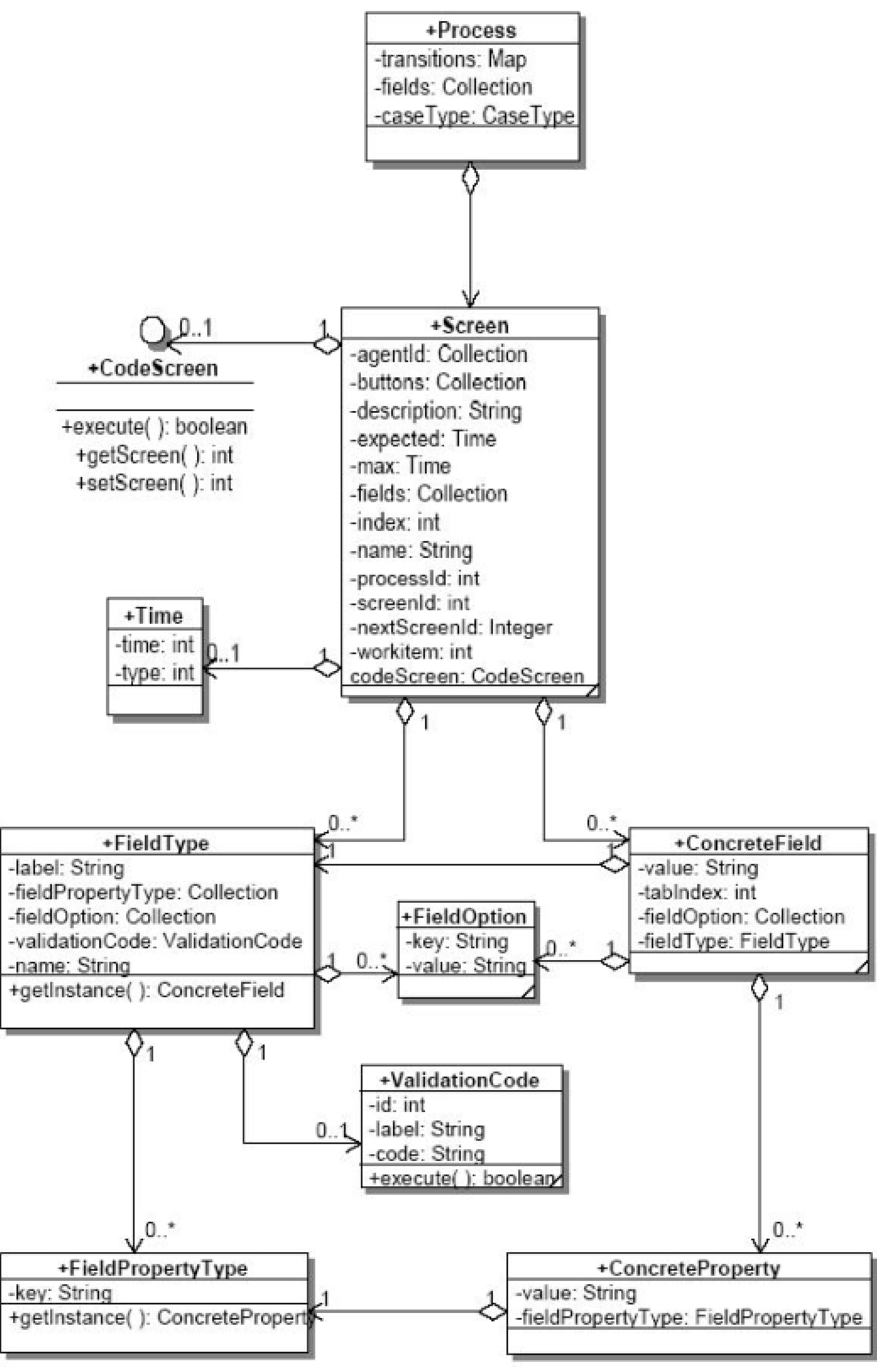

Figura 14: Diagrama de classe das extensões semânticas e funcionais.

Embora o código do núcleo e interface web encontram-se praticamente prontos, ainda há

o que ser implementado na parte das extensões.

2.4 Camada de dados

Parte da persistência dos dados no projeto (em especial dentro do núcleo) é feita através de uma implementação da especificação **JDO** (Java Data Objects) que consiste, como mencionado anteriormente, numa especificação de um conjunto de interfaces (**API**) cuja a finalidade é a de tornar mais simples a persistência e recuperação de modelos de objetos [4].

Uma das maiores vantagens da especificação **JDO** é a sua capacidade de persistir transparentemente modelos de objetos. Sem entrar muito no mérito da questão, apontamos apenas que o desenvolvimento do aplicativo torna-se muito mais simples quando há uma mudança de foco da camada de persistência para a lógica do próprio aplicativo. A maneira pela qual a persistência transparente permite essa mudança de foco deve ser clara.

Nós optamos pela implementação **JPOX**, que é livre e que fornece a maior parte das funcionalidades adicionais necessárias. Além disso, a **JPOX** pode ser utilizada em conjunto com **MySQL** ou o **PostgreSQL**, que são os bancos de dados relacionais livres mais difundidos no momento.

A especificação **JDO** define inúmeras facilidades que trazem ao nosso projeto características no mínimo interessantes. Dentre elas, podemos destacar:

- Demarcação de transações;
- persistência por transitividade;
- recuperação transparente (num dado grafo de objetos);
- possibilidades de buscar objetos satisfazendo determinado critério;
- outras facilidades de consulta.

Uma implicação direta disso é que o núcleo, por exemplo, apresenta comportamento transacional e possui um mecanismo de auto-persistência que garante as propriedades **ACID** (é, portanto, confiável).

3 Organização do projeto

Desde o início, tínhamos conhecimento da complexidade do projeto e sabíamos que a organização seria essencial para seu sucesso.

3.1 Responsabilidades

O projeto pode ser dividido em duas grandes etapas: a primeira, na qual nos dedicamos ao estudo das tecnologias e conhecimento do domínio da aplicação, e a segunda, na qual iniciamos a modelagem e a implementação do sistema.

A divisão do grupo em duas equipes ocorreu apenas na segunda etapa. Uma das equipes, formada pelo Carlos e pelo Pedro, ficou responsável por desenvolver a camada web e extensões, enquanto a outra, formada pelo Cleber e Giuliano, ficou encarregada de desenvolver o núcleo.

Embora o grupo tenha sido dividido, a comunicação continuou sendo através de uma lista de e-mail única. Dessa forma, decisões críticas poderiam ser discutidas por todo o grupo.

3.2 Andamento

Num primeiro momento, consideramos um projeto diferente do apresentado nesta monografia. Seria desenvolvido um conjunto de ferramentas para desenvolvimento de intranets empresariais. Esse projeto, todavia, mostrou-se ser por demais genérico, considerando a nossa falta de conhecimento do domínio de aplicação de tal sistema. Optamos, portanto, por desenvolver algo que nos parecesse mais viável dentro da atual situação do grupo. A escolha foi um sistema de Workflow Engines.

Inicialmente dedicamos o nosso tempo ao estudo de algumas ferramentas (Workflow Engines) disponíveis no mercado, para compreendermos melhor o domínio da aplicação. No decorrer do primeiro semestre, também estudamos as tecnologias e padrões de design que pretendíamos utilizar. Algum esforço de modelagem também foi feito mas sem que chegássemos, no entanto, em nada concreto.

A modelagem, que estava prevista para ser finalizada até o final de julho, só foi concluída no final de agosto (e alterada em alguns pontos durante a implementação). Embora a modelagem do sistema parecesse ser uma tarefa simples, foi uma das etapas mais trabalhosas e desgastantes, e foi o início do atraso nos prazos que havíamos estipulado.

Até o começo de setembro todos os integrantes do grupo trabalharam em conjunto. O objetivo era que todos atingissem praticamente o mesmo nível de conhecimento das tecnologias e ferramentas analisadas, e que fôssemos capaz de decidir quais seriam as melhores opções. Optamos também por realizar toda a modelagem do sistema trabalhando juntos, para que todos opinassem em todas as partes do sistema, e que adquirissem uma visão geral do mesmo. Neste período, fazíamos reuniões mensais, para discutirmos o andamento do projeto, pensarmos em assuntos que não havia ficado muito claro nas discussões que mantinhamos através da lista de discussão por e-mail.

A divisão do grupo em duas equipes ocorreu no início da implementação. Optamos por realizar essa divisão para facilitar o processo de implementação, pois programávamos aos pares sempre que possível. Percebemos, também, que seria difícil a reunião de todos os integrantes do grupo para o desenvolvimento, em virtude da diferença de horário livre entre os participantes e devido às severas restrições de tempo impostas pelo curso. Neste período, as reuniões com a participação de todos foram mais freqüentes, ocorrendo a cada duas semanas, devido ao aumento da dificuldade no desenvolvimento.

3.3 Situação atual

Apesar de todo o esforço, não foi possível terminar o projeto, ou ao menos um protótipo funcional, até o início de março.

Embora o código do núcleo e as funcionalidades da camada web encontram-se praticamente prontos, ainda há o que ser implementado na parte das extensões.

O código fonte do projeto [1] está disponível para consulta.

3.4 Ferramentas utilizadas

Segue abaixo uma breve descrição das principais ferramentas que nos auxiliaram no desenvolvimento do sistema.

- **Eclipse**: é um ambiente de desenvolvimento (IDE) baseado em princípios de código aberto, que possui uma arquitetura extensível baseada no uso e desenvolvimento de

plugins. A vantagem é o grande número de *plugins* disponíveis, que permite a integração com outras ferramentas disponíveis no mercado como o Tomcat, JBoss, CVS, Ant, entre outras.

- **CVS**: é um sistema de controle de versão, que nos permite manter as versões mais antigas dos arquivos armazenados (código fonte e outros arquivos) e mantém um log de quem, quando e como as mudanças ocorreram. O CVS opera não somente em um arquivo ou um diretório por vez, mas sim em coleções hierárquicas de diretórios. Ele ajuda a gerenciar as distribuições e a controlar a edição de arquivos por diversas pessoas, que ocorre concorrentemente.

  O CVS foi essencial para automatizar a integração do código que foi desenvolvido pelas equipes. Inicialmente tentamos utilizá-lo através do próprio Eclipse, porém encontramos alguns problemas no *plugin*, e optamos por executá-lo através da linha de comando.

- **Ant**: é uma ferramenta de código aberto, escrita em Java, desenvolvida pela "Apache Software Foundation", que tem como finalidade automatizar o processo de compilação de código. Esta ferramenta é a mais comumente usada para este fim em programas escritos em Java. O Ant é bastante portátil e simples de usar. Ele não depende da plataforma utilizada e nem do ambiente de desenvolvimento utilizado.

- **XDoclet**: é uma ferramenta geradora de código, que permite a Programação Orientada a Atributo. É possível adicionar mais significado ao código através da escrita de metadados (atributos) no código fonte Java. Sua utilização nos permitiu um grande aumento da produtividade diminuindo o número de linhas de código escritas.

4 Experiência pessoal

4.1 Desafios e frustrações

Certamente o maior desafio foi trabalhar em um grande projeto por um longo período. Havia muitos detalhes a serem pensados, e o pouco conhecimento do domínio da aplicação foi o grande desafio inicial.

Outra dificuldade foi o aprendizado de tecnologias muito recentes. O conhecimento dos participantes do grupo não era homogêneo, o que aumentou muito o ritmo dos estudos com objetivo de nivelar este conhecimento.

Vencido esse desafio deparamos com a dificuldade de modelar o sistema. A descoberta da aplicação de Redes de Petri para representação de WF facilitou a modelagem, porém dificultou o desenho da interface do sistema.

Os desafios foram muitos, e a medida que foram aparecendo, foram sendo superados. Durante todo o projeto, também havia o desafio da divisão do tempo entre as responsabilidades do curso, o projeto e o estágio.

Particularmente não tive muitas frustrações. Desde o início do projeto, o principal objetivo do grupo era de aprender muito, e este certamente foi atingido. Entre as frustrações, posso citar a impossibilidade de cumprir os prazos estipulados na proposta e a não conclusão da implementação.

4.2 Disciplinas do BCC mais relevantes

Muitas disciplinas influenciaram direta ou indiretamente com mais ou menos intensidade o desenvolvimento do projeto. Listarei abaixo apenas as disciplinas que foram mais importantes para sua concretização:

- **MAC110** - Introdução à Computação
  MAC122 - Princípios de Desenvolvimento de Algoritmos
  MAC323 - Estruturas de Dados

  Certamente essas disciplinas forneceram a base para todo o curso. Através delas, aprendi as técnicas mais básicas de programação, utilização de estruturas de dados mais complexas e a pensar algoritmicamente na solução de um problema. Foi o primeiro contato com o conceito de algoritmo e a preocupação com sua eficiência.

- **MAC211** - Laboratório de Programação I
  MAC242 - Laboratório de Programação II

  Laboratório de Programação I possibilitou o primeiro contato com um projeto maior, e Laboratório de Programação II introduziu uma linguaguem orientada a objetos: Java. Ambas as disciplinas foram muito importantes para o aprendizado da programação em equipe, divisão das tarefas entre o grupo e cumprimento dos prazos de entrega.

  Em Laboratório de Programação I também aprendemos uma linguagem de *script*, que é extremamente útil para automatização de tarefas rotineiras. Embora o projeto tenha sido desenvolvido em Java, foram escritos *scripts* em Perl que possibilitaram o desenvolvimento do projeto na Rede Linux. Estes *scripts* eram responsáveis por fazer o *download* e configuração de programas como o Tomcat, JBoss, e o banco de dados Postgres, em diretórios temporários.

- **MAC316** - Conceitos Fundamentais de Linguagens de Programação
  MAC332 - Engenharia de Software

  Não poderia deixar de citar disciplinas que forneceram importantes conceitos de programação. Conceitos Fundamentais de Linguagens de Programação apresentou os diversos paradigmas de programação, destacando suas vantagens e desvantagens, e ensinando-nos a escolher o ideal de acordo com a aplicação a ser desenvolvida. Engenharia de Software forneceu o conhecimento necessário para implementar e desenvolver um projeto de forma organizada, além de métodos para testar o sistema.

- **MAC328** - Algoritmos em Grafos
  MAC414 - Linguagens Formais e Autômatos

  Os conceitos apresentados nesta disciplina foram muito úteis para a modelagem de um WF utilizando Redes de Petri.

- **MAC338** - Análise de Algoritmos

  Como o próprio nome já diz, nessa disciplina foi ensinada a análise de algoritmos, detectando seus pontos ineficientes e provando formalmente seu correto funcionamento. O conhecimento adquirido foi utilizado para analisar alguns trechos de código do sistema.

- **MAC426** - Sistemas de Bancos de Dados

  Embora não tenhamos modelado o sistema e escrito consultas SQL, os conceitos aprendidos através desta disciplina foram essenciais para a compreensão e utilização de uma implementação do JDO (*Java Data Object*).

Algumas disciplinas não cursadas, cujo conteúdo estudei sozinho, foram essenciais para o desenvolvimento de um projeto orientado a objetos. Entre as quais posso citar Programação Orientada a Objetos (MAC441) e Tópicos de Programação Orientada a Objetos (MAC413).

4.3 Interação com os membros da equipe

Este projeto não foi muito diferente aos quais estávamos acostumados a enfrentar, quanto a interação entre os membros da equipe. Todos já haviam trabalhado juntos em outros projetos, e a convivência entre os integrantes sempre foi boa.

Além de uma lista de mensagens e reuniões para discutir o projeto, formas de comunicação que já haviam sido utilizadas em outros projetos, usamos o CVS.

A grande vantagem de desenvolver um projeto em equipe é que em qualquer fase do projeto sempre há uma pessoa motivada, que incentiva os outros a trabalharem.

A amizade e a liberdade entre todos os integrantes também foi uma desvantagem, pois muitas vezes foi difícil ordenar ou especificar uma tarefa para uma pessoa, além de não haver uma cobrança muito rígida entre os membros.

4.4 Considerações Finais

Desde o início da graduação em Bacharelado em Ciência da Computação, muitos, inclusive eu, questionaram a *utilidade* de algumas disciplinas muito teóricas, que pareciam não ser importantes.

O conhecimento adquirido nessas disciplinas e muitas outras, só foi notado quando precisei compreender novos conceitos que nunca havia estudado anteriormente. Nos primeiros cinco meses, o projeto resumiu-se em aprender novas tecnologias, e foi fascinante observar o crescimento intelectual que o IME ofereceu.

As disciplinas foram essenciais para aumentar nosso conhecimento dos fundamentos da Computação, desenvolver nossa capacidade criativa, estimular a auto-aprendizagem e o pensamento crítico.

Certamente essa formação sólida foi fundamental para o desenvolvimento deste projeto, além de ter colaborado para meu crescimento intelectual, profissional e pessoal.

5 Referências

Segue abaixo a lista das referências utilizadas no decorrer do desenvolvimento do projeto.

1. *Repositório do Projeto* em `http://easyflow.homelinux.net:5151/cgi-bin/viewcvs.cgi/easyflow/`.

2. CRUZ, Tadeu. *E-Workflow*, CENADEM, 2001.

3. van der Aalst, Wil M.P. *The Application of Petri Nets to Workflow Management*, 1998, em: `http://tmitwww.tm.tue.nl/staff/wvdaalst/Petri_nets/petri_nets.html`.

4. ROOS, Robin.*M. Java Data Objects*, Addison Wesley, 2003.

5. MARINESCU, Floyd. *EJB Design Patterns*, John Wiley & Sons, Inc., 1999.

6. YODER, Joseph W. e JOHNSON, Ralph. *The Adaptive Object-Model Architectural Style*, 2002, em: http://www.adaptiveobjectmodel.com

7. *Data Access Object*, em http://java.sun.com/blueprints/patterns/DAO.html

8. ROMAN, Ed e AMBLER, Scott e JEWELL, Tyler. *Mastering Enterprise JavaBeans*, 2002.

9. ALUR, Deepak e CRUPI, John e MALKS, Dan. *Core J2EE Patterns*, Prentice Hall PTR, 2001.

10. GAMMA, Erich e HELM, Richard e JOHNSON, Ralph e VLISSIDES, John. *Design Patterns*, Addison Wesley, 1995.

11. DEITEL, Harvey M. e DEITEL, Paul J. *Java: Como Programar*, Bookman, 2003.

12. ECKEL, Bruce. *Thinking in Java*, Prentice Hall PTR, 2000.

13. PRESSMAN, Roger S. *Engenharia de Software*, McGraw-Hill, 2002.

14. FOWLER, Martin e SCOTT, Kendall. *UML Essencial*, Bookman, 2000.

15. *Java Servlet Technology*, em http://java.sun.com/webservices/docs/1.0/tutorial/doc/Servlets.html

16. *The Tomcat 4 Servlet/JSP Container*, em http://jakarta.apache.org/tomcat/tomcat-4.1-doc/index.html

17. *Velocity User Guide*, em http://jakarta.apache.org/velocity/user-guide.html

18. *Velocity Layout Servlet*, em http://jakarta.apache.org/velocity/tools/view/layoutservlet.html

19. *Struts User Guide*, em http://jakarta.apache.org/struts/userGuide/index.html

20. *Apache Ant 1.5.4 Manual*, em http://ant.apache.org/manual/index.html

21. *CVS Manual*, em http://www.cvshome.org/docs/manual/cvs-1.11.10/cvs.html

22. *Java Persistent Objects - JPOX*, em http://jpox.sourceforge.net/