GIANCARLO GUIZZARDI

Uma abordagem metodológica de desenvolvimento para e com reuso, baseada em ontologias formais de domínio

VITÓRIA
2000

GIANCARLO GUIZZARDI

Uma abordagem metodológica de desenvolvimento para e com reuso, baseada em ontologias formais de domínio

RESUMO

Este trabalho tem, como objetivo, definir um modelo de processo genérico para o desenvolvimento de software, que associa de maneira complementar as perspectivas de desenvolvimento *para* e *com* reuso. No desenvolvimento para reuso é adotada a disciplina de engenharia de domínio, com objetivo de promover a adoção sistemática da prática de reutilização de software em um nível de abstração mais alto que o de codificação. Através de atividades de análise e projeto de domínio, a engenharia de domínio é capaz de desenvolver infra-estruturas reutilizáveis, que capturam a estrutura e o conhecimento de uma família de aplicações. A fim de suprir as deficiências existentes nos métodos atualmente adotados para a realização de suas atividades, uma nova abordagem é proposta. Essa abordagem é fundamentada em um processo bem estruturado de construção de ontologias formais de domínio e em um conjunto de diretivas, regras de transformação e padrões de projeto, capazes de conduzir de maneira sistemática a tradução das estruturas conceituais da ontologia em construções da infra-estrutura computacional de reuso. Por fim, o trabalho apresenta um estudo de caso, na área de sistemas multimídia distribuídos, que promove a experimentação das metas propostas anteriormente.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

Capítulo 1

Introdução

Por razões históricas, a evolução do processo de desenvolvimento de software, não atingiu a maturidade comum às demais disciplinas de engenharia. É inegável que algum avanço tenha sido alcançado, pois a forma de realização dessa atividade evoluiu de uma atividade realizada de forma quase artesanal, para um processo de desenvolvimento bem estruturado e que, nos melhores casos, contempla inclusive atividades de gerência e avaliação da qualidade.

Apesar disso, algumas questões fundamentais às outras disciplinas de engenharia continuam não sendo contempladas de forma sistemática pela ***engenharia de software***, como:

- desenvolvimento de artefatos genéricos passíveis de reuso;
- a construção de aplicações a partir da integração de componentes corretos e possuidores de interfaces bem definidas;
- a adoção de metodologias rigorosas de desenvolvimento, baseadas em teorias matemáticas bem fundadas, e que possam garantir o atendimento aos requisitos de qualidade pretendidos.

A falta de maturidade dessa disciplina faz com que a maioria dos desenvolvedores e organizações não seja capaz de enxergar a necessidade da abordagem dessas questões. Pelo mesmo motivo, na comunidade de engenharia de software, existe a instituição de um dogmatismo metodológico, que impede que a área possa se beneficiar de avanços alcançados em áreas afins.

Em suma, apesar da crescente complexidade das aplicações a serem desenvolvidas, a engenharia de software não tem demonstrado maturidade compatível

para desenvolver e evoluir a grande massa de programas existentes, dos quais as empresas e as pessoas se tornam cada vez mais dependentes.

1.1 - Justificativa

Nos anos recentes, o modelo de gestão de várias organizações tem passado por uma série de modificações, a fim de atender à crescente mudança dos cenários de negócios e, conseqüentemente, às constantes necessidades de adaptação a esses cenários.

Essas organizações, que tradicionalmente têm subestimado a complexidade envolvida no desenvolvimento de sistemas computacionais, se vêm obrigadas a mudar o foco de seus requisitos não-funcionais de *diminuição de prazo* para *confiabilidade* e de *redução de custos* para *adaptabilidade*.

Nesse novo cenário, não é mais suficiente que uma aplicação seja correta e atenda aos requisitos funcionais para os quais foi desenvolvida. Idealmente, essa mesma aplicação deve crescer, mudar e abordar problemas similares com o passar do tempo.

Dessa forma, além das perspectivas de *fazer a coisa certa*[1] e *fazer de forma certa*, os processos de desenvolvimento tem que estar preparados para *fazer a próxima coisa*:

a) *Faça a coisa certa:* esse tópico está diretamente ligado à validação ou, em outras palavras, à garantia de que o que está sendo construído é o que realmente é desejado. Nesse novo cenário, não somente os requisitos da aplicação precisam ser compreendidos, como esses requisitos podem mudar rapidamente durante o processo de desenvolvimento. Por esse motivo, é altamente desejável que o domínio no qual a aplicação foi construída seja compreendido como um todo. Com a utilização de uma abordagem formal de desenvolvimento, esse processo de validação pode ser conduzido de forma automática. Além disso, essa abordagem faz com que a comunicação e a documentação desses requisitos possam ocorrer sem ambigüidades, inconsistências e contradições, tornado os domínios mais compreensíveis e fazendo com que as suas aplicações tenham uma maior mantenabilidade. Por fim, esse tópico adiciona uma nova perspectiva ao conceito de reutilização: ao

[1] *Do the right thing, Do the thing right e Do the next thing.*

invés de reutilizarmos somente componentes de código, é importante que possamos desenvolver e reutilizar elementos mais abstratos, como, por exemplo, a experiência embutida em padrões de solução, arquiteturas de estruturação de elementos do domínio e, idealmente, estruturas capazes de capturar, de forma explícita, o conhecimento disponível em uma classe de problemas.

b) *Faça de forma certa*: esse tópico está relacionado à verificação, ou melhor, à garantia de que o que é desejado está sendo construído de maneira correta. Até então, sistemas de software têm sido usados, principalmente, como ferramentas de apoio e automatização de tarefas no contexto de uma organização. Nesse novo cenário, vários sistemas computacionais representam o próprio negócio, ou até mesmo a própria organização, de modo que o seu processo de desenvolvimento deve ser conduzido com rigor matemático próprio de outras disciplinas da engenharia. É importante salientar, porém, que as linguagens matemáticas são ferramentas teóricas que devem servir aos propósitos de um desenvolvedor humano, e não o contrário. Desse modo, é fundamental que o emprego de métodos formais seja conduzido com a visão mais abrangente, de que requisitos de naturezas diferentes requerem métodos formais diferentes.

c) *Faça a próxima coisa:* esse tópico está relacionado com a propriedade de ***adaptabilidade*** em suas várias formas: (i) *extensibilidade* - diz respeito à habilidade do sistema ser estendido sem que isso ocasione mudanças profundas em sua estrutura, como, por exemplo, para abrigar a introdução de uma nova ferramenta ou um novo tipo de dispositivo; (ii) *flexibilidade* - o sistema deve ser capaz de promover mudanças em seus mecanismos de interface ou em suas tecnologias de armazenamento, sem que isso tenha impacto na arquitetura; (iii) *escalabilidade* - a demanda dessa característica está relacionada à necessidade de adaptação ao novo cenário de distribuição, de aplicações multimídia e de crescente heterogeneidade das plataformas de hardware e sistemas operacionais. Os sistema devem ser construídos a fim de

prever mudanças de tecnologias e de plataformas, além de considerar previamente a necessidade da redundância e do balanceamento de informações.

1.2 - Objetivos

Este trabalho tem, como meta inicial, definir um modelo de processo genérico para desenvolvimento de software, que associa de maneira complementar as perspectivas de desenvolvimento *para* e *com* reuso. Na perspectiva de desenvolvimento para reuso, deve ser ressaltada a importância de se pensar de forma mais abrangente sobre o conceito de *produto reutilizável*, ou seja, ao invés de ser explorada apenas a nível de código, a reutilização deve acontecer através de artefatos situados em níveis mais altos de abstração, e que, portanto, permitam o reuso de itens como soluções de projeto, infra-estruturas de domínio e, idealmente, estruturas de conhecimento.

Desse modo, como processo de desenvolvimento desses artefatos, é adotada a disciplina de ***engenharia de domínio*** - uma espécie de engenharia de software que atua em um meta-nível, capaz de desenvolver infra-estruturas reutilizáveis, que capturam a estrutura e o conhecimento de uma família de aplicações.

De forma análoga à engenharia de software, a engenharia de domínio pode ser dividida nas seguintes fase: ***análise de domínio*** e ***projeto de domínio.***

A ***análise de domínio*** tem como objetivo a produção de um modelo de domínio capaz de identificar, capturar e organizar os elementos relevantes à representação do conhecimento embutido em uma classe de problemas. No entanto, os métodos atualmente existentes para realização dessa atividade possuem uma séria limitação no que diz respeito aos modelos de domínio gerados. Esses modelos permitem a representação do conhecimento apenas em um nível de estruturação, o que é insuficiente para representar, de forma explícita, o conhecimento envolvido na restrição da interpretação de elementos do domínio, na especificação de condições a serem cumpridas para que relações entre eles possam ser estabelecidas e, principalmente, na derivação de conhecimento a partir do conhecimento factual representado.

Dessa forma, este trabalho tem como objetivo propor uma nova abordagem para a realização dessa atividade, fundamentada em um processo sistemático de construção de ontologias formais. Ontologias vêm sendo usadas há vários séculos pela filosofia e pela

linguística e, mais recentemente, pela inteligência artificial, como uma importante ferramenta de representação de taxonomias de elementos, em um nível epistemológico, e de definição de teorias axiomáticas, em um nível ontológico. Além disso, como conseqüência do formalismo empregado, muitos são os benefícios alcançados, como, por exemplo, a verificação/validação automática do modelo de conhecimento construído, a interpretação não ambígua das definições dos elementos que a compõem e a possibilidade de uma transição sistemática (e idealmente automática) para a fase de projeto de domínio.

Com o intuito de prover uma disciplina estruturada para a fase de ***projeto de domínio***, este trabalho tem também como objetivo apresentar um conjunto composto de diretivas, regras de transformação e padrões de projeto, capazes de conduzir de maneira sistemática a tradução das estruturas conceituais da ontologia em construções da infra-estrutura computacional de reuso.

Por fim, o trabalho apresenta um estudo de caso, que promove a experimentação das três metas anteriormente propostas. O domínio de aplicação escolhido é o de Vídeo sob Demanda, por possibilitar a experimentação de diversos aspectos interessantes definidos no modelo. Para isso, será desenvolvida uma ontologia de Vídeo sob Demanda, e será derivada a infra-estrutura correspondente. A partir do reuso dessa infra-estrutura, bem como de outros componentes passíveis de reutilização, uma aplicação nesse domínio será, então, construída.

1.3 - Organização do Trabalho

Este trabalho contém, além desta Introdução, mais quatro capítulos.

O capítulo 2 - ***Um modelo genérico de processo de desenvolvimento de software para e com reuso*** - Discute os principais aspectos relacionados ao desenvolvimento *para* e com *reuso*, focando principalmente os fatores responsáveis pela sua fraca adoção nos processo de desenvolvimento vigentes. A análise desses fatores, assim como o estudo das características fundamentais dos principais componentes de reutilização, tem como objetivo definir um processo bem estruturado, que contempla os requisitos necessários à adoção sistemática dessa prática. Por fim, é proposto um modelo genérico de processo de desenvolvimento que associa, de forma complementar, as disciplinas de engenharia de software e engenharia de domínio.

O capítulo 3 - ***Análise de Domínio e Ontologias*** - Define a fase de análise de domínio em termos de seus principais conceitos e atividades. Estabelece um modelo de processo comum aos métodos de análise de domínio mais difundidos, e discute seus principais benefícios e limitações. A fim de suprir algumas dessas limitações, apresenta um processo para a realização dessa atividade, baseando-se na construção de ontologias formais de domínio. Por fim, justifica a adequação dessa abordagem, apresentando suas principais contribuições.

No capítulo 4 - ***Formalização de Ontologias e Projeto de Domínio*** - Completa o processo de sistematização da engenharia de domínio, apresentando uma abordagem para a representação de ontologias, composta de uma linguagem gráfica, para a estruturação de conceitos e relações, e uma linguagem formal para a definição de axiomas. Apresentada toda a fundamentação teórica da linguagem e discute questões ontológicas importantes, que influenciam decisões de modelagem. Finalmente, apresenta um conjunto de diretivas, regras de transformação e padrões de projeto, responsáveis pelo mapeamento da estrutura da ontologia em elementos da infra-estrutura de domínio.

O capítulo 5 - ***Hipervisão: Um estudo de caso*** - Apresenta um estudo de caso das várias contribuições propostas no trabalho. Primeiramente, é feita uma discussão detalhada a respeito dos requisitos metodológicos da área de sistemas multimídia distribuídos, a fim de instanciar o modelo proposto no capítulo 2. O domínio de Vídeo sob Demanda é o sub-domínio dos sistemas multimídia distribuídos escolhido para ser alvo da aplicação do processo instanciado. Esse domínio é profundamente discutido, e uma ontologia de gerência de Vídeo sob Demanda é desenvolvida. Em seguida, é gerada, a partir dessa ontologia, a infra-estrutura de domínio correspondente. Por fim, é apresentado um estudo de caso de desenvolvimento com reuso, desenvolvendo-se uma aplicação de vídeo sob demanda a partir da integração de componentes existentes.

O capítulo 6 - ***Conclusões e Trabalhos Futuros*** - contém as conclusões deste trabalho, evidenciando suas contribuições e perspectivas de futuros trabalhos.

Capítulo 2

Um Modelo Genérico de Processo de Desenvolvimento de Software *para* e *com* Reuso

2.1 - Introdução

Nos primeiros sistemas computacionais programados, entre as décadas de quarenta e sessenta, o contexto de desenvolvimento de software era marcado por pequenas equipes de programação, cujos membros eram considerados cultos e possuidores de práticas altamente individualistas. Estes programadores agrupavam as funções de análise, implementação e teste e, pelo fato de atuarem em um contexto completamente diferente do de seus gerentes, atividades de gerência e avaliação eram praticamente ignoradas.

No entanto, os produtos de software, encomendados por grandes empresas e governos, sinalizaram o fato de que toda a prática de desenvolvimento de software até então era feita de forma artística e gerenciada de maneira totalmente oracular. Diante desta situação, que mais tarde ficou conhecida como a *Crise do Software*, no final da década de sessenta o comitê científico da OTAN convenceu os cinqüenta principais cientistas da computação, programadores e gerentes da indústria de software da época a comparecerem ao - hoje considerado histórico - encontro em Garmisch, Alemanha (Gibbs, 1994). O principal objetivo dessa reunião era tentar estabelecer uma prática mais madura para este processo de desenvolvimento. Apesar de não conseguirem vislumbrar como este objetivo seria alcançado, todos os presentes concordaram que essa meta deveria se chamar *Engenharia de Software*.

Desse modo, houve um consenso de que o processo de desenvolvimento de software deveria ser estabelecido de forma análoga a outras disciplinas de engenharia, ou

seja, a escolha de métodos, técnicas, ferramentas e atividades arranjadas em um ciclo de vida deveria ser feita com base em um plano estruturado e bem definido de qualidade.

Simultaneamente ao surgimento dessa preocupação, é vislumbrada a necessidade de orientar o foco do desenvolvimento para a produção de componentes reutilizáveis de software, que possibilitariam a construção de novas aplicações a partir da integração de componentes previamente implementados e devidamente testados (Gall et al., 1995, McIllroy, 1968), uma prática comum a outras disciplinas de engenharia, especialmente a engenharia de hardware.

Atualmente, apesar da crescente complexidade das aplicações a serem desenvolvidas, a engenharia de software não tem demonstrado maturidade compatível, não conseguindo acompanhar a evolução do hardware, não conseguindo suprir a demanda do mercado por novos produtos e principalmente não possuindo habilidades (metodologias) para evoluir a grande massa de programas existentes dos quais as empresas e as pessoas se tornam cada vez mais dependentes.

O objetivo deste capítulo é apresentar um modelo de processo genérico para desenvolvimento de software, no qual a prática sistemática de reutilização desempenha um papel fundamental. O modelo é definido através de dois níveis: (i) nível de domínio - responsável pela produção de infra-estruturas de domínio passíveis de serem reutilizadas; (ii) nível da aplicação – responsável pela construção de aplicações a partir da especialização dessa infra-estrutura.

No contexto desse trabalho, as disciplinas de engenharia responsáveis pela geração de produtos computacionais – em ambos os níveis - são vistas como arquiteturas, compostas de ferramentas, métodos, atividades, artefatos e insumos, definidos com intuito de cumprir uma meta previamente negociada de qualidade. Essa abordagem é descrita na seção seguinte. Em seguida, na seção 2.3, são discutidos os principais aspectos relacionados ao desenvolvimento para e com reuso, focando principalmente os fatores responsáveis pela sua fraca adoção nos processo de desenvolvimento vigentes. A seção 2.4 apresenta os principais componentes de reutilização, além da forma que se relacionam a fim de compor uma infra-estrutura de domínio. É também discutida a dificuldade de construção dessas infra-estruturas, ressaltando a necessidade de um processo bem estruturado de engenharia de domínio. Por fim, a definição de um processo

de engenharia de domínio, a sua integração com a engenharia de software e apresentação de um modelo genérico de ciclo de vida são os objetivos da seção 2.5, culminando na proposição do modelo pretendido pelo capítulo.

2.2 – Engenharia de Software

A Engenharia de software pode ser organizada como uma arquitetura em camadas como é mostrada na figura 2.1.

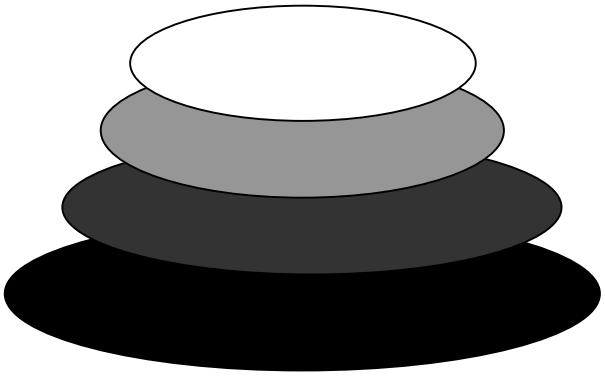


Figura 2.1 - Camadas que constituem a disciplina de Engenharia de Software

A pedra fundamental que suporta esta tecnologia é um plano de qualidade. Neste ponto, pode ser feita uma referência oportuna ao conceito de *QoS (Quality of Service)* empregado na área de Redes de Computadores. Este conceito pode ser sucintamente definido da seguinte forma (Nahrstedt & Smith, 1995):

> **QoS** *é uma estrutura formada por um conjunto de parâmetros que refletem o grau de qualidade de serviço desejado pelo usuário e que deve ser interpretada e transformada em recursos computacionais (protocolos de monitoramento, obtenção de recursos, garantia de confiabilidade entre outros) que possam garantir que os resultados finais estejam dentro dos padrões exigidos pelo usuário. Trata-se de um contrato, que deve ser respeitado por ambas as partes e cujo custo é proporcional aos serviços oferecidos.*

Analogamente, podemos endereçar a qualidade do produto em um processo de engenharia de software da mesma maneira, ou seja, o software é um produto cujos

requisitos de desempenho, confiabilidade, heterogeneidade, amigabilidade, entre outras coisas são definidos pelo cliente em uma das fases da análise e especificação de requisitos. Estes podem ser considerados os parâmetros de QoS para a engenharia de software. Uma vez definidos estes parâmetros, o usuário deve se comprometer a não modificá-los, e para isso deve estar ciente do impacto que isso terá no custo do projeto *("...custo proporcional aos serviços oferecidos..."*).

Do outro lado, os desenvolvedores também devem interpretar estes parâmetros, e usá-los a fim de garantir a qualidade prevista no contrato. Como acontece com os protocolos de Redes de Computadores, as tarefas associadas a esse processo de garantia de QoS devem estar ativas durante todo o processo. Portanto, no caso da engenharia de software, estes protocolos se materializarão sob a forma de planejamento do projeto, métricas e indicadores (sendo os principais: correção, mantenabilidade, integridade e usabilidade), verificação e validação, gerência de riscos, determinação do cronograma, acompanhamento do projeto, aplicação efetiva de métodos, técnicas e ferramentas, controle de documentação e obediência aos padrões seguidos. Em outras palavras, a gerência do projeto de software, que é uma atividade *de suporte*[2] e portanto, ocorre durante todo o projeto, pode ser considerada uma arquitetura de QoS para a engenharia de software.

A fundação é a segunda camada, que fala do processo. O processo pode ser descrito como um conjunto de atividades relacionadas, capaz de estruturar os métodos, técnicas e recursos e coordenar o uso de ferramentas, a fim de atingir a qualidade negociada no "contrato de qualidade", além de propiciar o suporte necessário ao gerenciamento de um projeto de desenvolvimento. Nesse contexto, um modelo de ciclo de vida desempenha um papel fundamental, descrevendo as atividades do processo, e estabelecendo entre elas, relações de dependência e precedência

Desse modo, os principais conceitos envolvidos na descrição de um processo são: atividade (de construção, gerência e avaliação da qualidade), procedimento (métodos, técnicas e diretrizes), recursos, artefatos produzidos e consumidos e modelo de ciclo de vida. Ao contrário do desenvolvimento de dispositivos de hardware, no qual o custo está ligado essencialmente no produto, o processo de software abriga quase todo o custo de

[2] *Umbrella activities*

desenvolvimento. Apesar disso, é impensável estabelecer um processo único, capaz de se adequar ao desenvolvimento de qualquer tipo de produto. Produtos diferentes possuem características diferentes que influenciarão diretamente na escolha desses elementos que compõem o processo.

Os principais fatores que influenciam essas escolhas são: a natureza do projeto e da aplicação, o paradigma a ser empregado, as tecnologias disponíveis, os métodos e ferramentas a serem usados, os produtos que devem ser gerados como saída de cada fase, além de outros fatores como disponibilidade de recursos (pessoas, tempo, dinheiro), experiência e tamanho da equipe, relação entre a produtividade das sub-equipes que compõem a equipe, conhecimento dos requisitos do problema, estabilidade dos requisitos do problema, facilidade de decomposição do problema e nível de qualidade desejado.

Apesar da inexistência de um processo único, existem fases genéricas que podem ser identificadas e que devem estar presentes em qualquer processo de desenvolvimento de software, independente da área de aplicação, do tamanho e da complexidade do projeto. Essas fases são descritas a seguir.

- fase de definição: Essa fase é focada na pergunta "*o que*": quais informações serão processadas, quais funcionalidades e requisitos de desempenho são desejáveis, que comportamento é esperado do sistema, quais interfaces serão estabelecidas, que requisitos de projeto existem e quais critérios de validação são requeridos. Apesar dos métodos presentes nesta fase variarem de acordo com características específicas do produto a ser construído, existem duas grandes tarefas que irão ocorrer de alguma forma: planejamento do projeto e análise de requisitos.
- fase de desenvolvimento (ou construção): Essa fase foca no "*como*": como os dados serão estruturados, como as funcionalidades desejadas serão implementadas sob forma de uma arquitetura de software, como as interfaces serão caracterizadas, como o projeto será traduzido em uma forma de codificação e como os testes serão conduzidos. De alguma forma, as seguintes tarefas estarão presentes: projeto, implementação e teste.
- fase de manutenção: Essa fase se concentra na *mudança*. Isso se faz necessário devido às mudanças ocorridas nos requisitos do produto ou erros cometidos durante o desenvolvimento. Assim, aplica-se novamente as atividades definidas

nas fases anteriores, mas faz isso no contexto de um programa existente. Essa fase pode envolver, inclusive, a definição de um novo processo, se necessário.

Assim, de uma maneira geral, todo processo envolve as seguintes atividades: planejamento, análise e especificação de requisitos, projeto, implementação e testes.

O planejamento baseia-se na determinação de estimativas precisas quanto a: recursos, tempo, custo e riscos envolvidos com o processo. Essas estimativas serão consideradas na elaboração de um cronograma de atividades, que facilitará a execução e o acompanhamento do processo.

A fase de análise e especificação de requisitos marca a delimitação do contexto do software, reconhecendo e modelando o problema no domínio do usuário e representando os requisitos fundamentais desse domínio. O analista trabalha junto ao cliente buscando reconhecer, analisar, sintetizar, modelar e finalmente especificar os requisitos do software a ser construído. Essa fase gera uma especificação que se tornará a fundação para todas as fases subsequentes do processo de desenvolvimento.

A fase de projeto serve como ponte entre o universo modelado do problema e sua solução computacional. O objetivo dessa fase é definir como os requisitos modelados na fase anterior podem ser contemplados na construção de um software. Hoje, é comum verificar, em ambientes corporativos, a fusão das atividades das fases de análise e projeto, o que resulta na construção de programas pouco flexíveis, dificultando a manutenção e o reuso.

Na fase de implementação, são escolhidos a linguagem de programação e o ambiente de desenvolvimento para a construção do software. Nessa etapa, o código é gerado, o banco de dados é implementado, bibliotecas de componentes são incorporadas, entre outras atividades.

Por fim, a fase de testes tem como principal finalidade validar e verificar o cumprimento dos requisitos especificados. Pode-se dizer que quanto mais erros um teste for capaz de encontrar, melhor ele alcançou seu objetivo. Ao contrário do que se possa imaginar, essa atividade deve acompanhar todo o processo, sendo planejada a cada etapa, a fim de diminuir o esforço para encontrar e corrigir erros.

2.3 – Desenvolvimento *para* e *com* reuso

Desde o seminário da OTAN em 1968, a possibilidade de desenvolvimento de sistemas a partir de partes previamente construídas, adaptáveis e provadas corretas, vem sendo vista como uma possível solução para a chamada *crise do software*. Nesse seminário, verificou-se a necessidade de criar tecnologias e paradigmas de desenvolvimento que possibilitassem, de um lado, a produção de componentes reutilizáveis com potencial e, de outro, a construção de sistemas a partir da integração de forma racional desses componentes (Gall et al., 1995). Segundo defendido em (Zand et al., 1997), a chave do sucesso dessa prática na engenharia está na habilidade de construir componentes baseados em abstrações simples, fáceis de serem encontrados, passíveis de uso em várias aplicações e cuja estrutura e comportamento encapsulados se tornam acessíveis através de interfaces bem definidas e matematicamente descritas.

É inegável que a reutilização vem acontecendo de alguma forma. Um exemplo claro disso é o reuso de algoritmos e estruturas de dados. Quando surge a necessidade de um algoritmo de ordenação atualmente, é impensável investir recursos como tempo e dinheiro tentando produzir um, ao contrário, o que é feito é o reuso de algoritmos estáveis, amplamente testados e cujo comportamento é bastante conhecido, como *quicksort* e *heapsort*, presentes em catálogos de algoritmos aclamados (Knuth, 1983). Apesar disso, o que de fato ainda não aconteceu é a inserção da prática de reutilização como parte integral do processo de desenvolvimento. Apesar de terem se passado mais de trinta anos desde que a meta foi lançada, pode-se observar que ela ainda está bem longe de ser alcançada, resultando, assim, em uma baixa produtividade no desenvolvimento e em produtos de baixa qualidade. As principais razões para esse fato podem ser divididas em três grupos de fatores: metodológicos, tecnológicos e culturais/econômicos:

a) **Metodológicos:** Confirmando uma tendência natural do desenvolvimento científico na área de computação, a comunidade de pesquisa em reuso de software se mostra como um grupo fechado que sofre de um terrível dogmatismo metodológico. Cada área de pesquisa possui seus próprios fóruns e conferências e parece insistir em perseguir a chamada bala de prata (*silver bullet*) da engenharia de software, ou seja, uma metodologia, paradigma ou procedimento que será a solução para todos os problemas no desenvolvimento de software. Dessa forma, por possuir uma mínima interseção com

outras disciplinas, essa comunidade obtém poucos benefícios de avanços em campos relacionados, como inteligência artificial e métodos formais, e menos ainda de outros campos aparentemente não relacionados, como filosofia e lingüística (Gall et al., 1995).

b) **Tecnológicos:** Como mencionado anteriormente, para que os componentes produzidos possam ser reutilizados, é fundamental que eles possam ser encontrados com facilidade. Por isso se faz necessário a construção de ambientes de desenvolvimento, que possam agrupar características que auxiliem os desenvolvedores a: localizar componentes a serem utilizados, entender os serviços oferecidos, pré-requisitos e conseqüências da utilização de um determinado componente e identificar oportunidades de reuso, entre outras coisas. Por outro lado, os ambientes devem facilitar a classificação e catalogação de componentes desenvolvidos para reuso e de suas respectivas informações (meta-dados). Além disso, deve haver uma política bem definida para a catalogação de componentes, de modo a evitar a proliferação de componentes com pouca possibilidade de reuso.

c) **Culturais e Econômicos:** Poulin (1997) defende a crença de que, do ponto de vista tecnológico, a área já atingiu maturidade suficiente para que o reuso possa ocorrer, e salienta o fato de que, na verdade, esse processo é guiado por questões organizacionais. Poulin critica a falta de trabalhos científicos que abordem esses aspectos não tecnológicos e prega a disseminação do conhecimento por parte dos pesquisadores para que as empresas comecem a conhecer e aplicar estes princípios. Segundo ele, enquanto não pudermos quantificar os benefícios do reuso em termos concretos como tempo e dinheiro economizados, a prática sistemática de reuso simplesmente não irá acontecer. Zand (1997) complementa essa idéia, afirmando que uma estratégia clara do ponto de vista organizacional é hoje o maior obstáculo para a adoção de um desenvolvimento para/com reuso em escala industrial e que as organizações geralmente desistem de um projeto de implantação da cultura de reuso sempre que têm que se confrontar com problemas orçamentários. Para ele, organizações que não possuem um plano sistemático de reuso, não possuem um sistema de desenvolvimento holístico e desconhecem as vantagens de uma vantagem estratégica de longo prazo, podendo ter um benefício apenas parcial do reuso de software.

Em outras disciplinas de engenharia como elétrica, mecânica e civil, a questão de reuso é tão óbvia que não é sequer discutida. Uma das principais razões disso é fato de que toda a formação acadêmica dessas disciplinas é também baseada nesse paradigma, ou seja, estudantes dos cursos de engenharia aprendem o projeto de artefatos através do uso de componentes (Gall et al., 1995). Esse aspecto, sem dúvida, contribui significativamente para a criação de uma cultura de reuso, na qual alguns fatores (fundamentais ao sucesso de uma engenharia de software orientada a reuso) são aceitos com naturalidade. Como exemplo desses fatores, podem ser citados (Jacobson et al., 1998):

- Para que os componentes produzidos possam ser testados e provados confiáveis, é necessário que um maior rigor matemático que os atuais rudimentos de formalismo, seja empregado no processo de desenvolvimento. Atualmente, abordagens baseadas em métodos formais são vistas como muito complexas pelas pessoas que atuam na indústria de software e que não possuem a formação matemática necessária. Processos completamente formais, também impõem dificuldades na comunicação com o cliente nas fases iniciais do desenvolvimento.
- A partir do momento que componentes são construídos usando práticas confiáveis e que facilitem a reutilização, a chamada atitude do "não inventado aqui" *(not invented here)* tende a acabar. Atualmente, é comum que programadores prefiram escrever todo o código a ser utilizado, para que possam ter controle completo sobre ele.
- Com medidas de produtividade difundidas, tal como *Linhas de Código* (LoC), programadores tendem a se sentir mais produtivos se escrevem mais linhas de código.
- Ainda não existe um padrão para especificação de componentes, o que facilitaria a sua maior disseminação e a interoperabilidade entre diferentes ambientes de desenvolvimento. Um exemplo de tentativa proprietária de padronização são os padrões JavaBeans/Enterprise JavaBeans (Tracz, 1997, Guerrieri et al., 1997).
- É preciso que seja estabelecido um modelo econômico de distribuição de componentes em larga escala, no qual questões de direito autoral sejam bem previstas e definidas.

Por fim, a reusabilidade de software é uma estratégia de engenharia, cujas várias implementações só podem ser justificadas em um contexto prático e econômico. Se as regras sociais e econômicas mudam, as estratégias de aplicação desse conceito devem acompanhar essas mudanças e, se não o fizerem, se tornam irrelevantes do ponto de vista de engenharia.

Como é possível notar, apesar de todas as vantagens óbvias, a inserção da prática de reuso, como parte integral do processo de desenvolvimento, requer antes de qualquer coisa um suporte completo da organização além de equipes dedicadas e com forte liderança (Griss et al, 1994). Primeiro, porque toda a prática de reuso acontece no contexto de uma organização, fazendo com que as atividades envolvidas no processo sejam fortemente marcadas por prioridades e restrições do próprio negócio. Desempenho, prioridades e eficiência são estabelecidos e medidos com relação aos objetivos da organização (Cornwell, 1996). Segundo, porque a inserção de novas atividades que suportem esse processo se mostra complexa e geralmente cara, dependendo da criação de toda uma cultura de apoio no ambiente em que se dará o desenvolvimento para reuso e com reuso. É necessário que prazos e orçamentos sejam previstos de forma realista para suportar efeitos colaterais resultantes da adoção desse paradigma como, por exemplo, a inicial queda de produtividade e resistência por parte de membros da equipe.

2.4 – Componentes e Técnicas de Reutilização

Como mencionado anteriormente, é inegável que o reuso de software vem acontecendo em alguma instância. Além dos exemplos de reutilização de algoritmos e estruturas de dados já citados, isso pode ser observado ao se analisar algumas aplicações, atualmente instalada em um computador pessoal, e constatar que é praticamente impossível descobrir quais arquivos pertencem unicamente àquela aplicação e quais arquivos são compartilhados por várias outras (ex. componentes COM usados por planilhas e editores de texto e DLL's compartilhadas por todos os softwares gráficos de um sistema operacional).

Ao analisarmos a evolução do reuso de software desde 1968, quando foi iniciada a construção das primeiras bibliotecas de funções científicas em FORTRAN, até hoje, quando todos os mais difundidos ambientes de desenvolvimento RAD (*Rapid Application Development*) incentivam fortemente o desenvolvimento a partir de catálogos de

componentes, uma perspectiva se mantém constante: é geralmente apenas no nível de código que o reuso vem sendo efetivamente aplicado. Apesar das vantagens de se reutilizar código, esses benefícios são maximizados quando aplicados em outras fases do processo de desenvolvimento, reutilizando artefatos de níveis mais altos de abstração, como estratégias de solução de problemas genéricos, modelos estruturais e comportamentais de domínios e, idealmente, conhecimento. Fazendo uma analogia com a engenharia de automóveis, os projetistas se beneficiam ao usar peças terceirizadas, com interfaces bem conhecidas, estáveis e abertas como, por exemplo, faróis e pneus. No entanto, eles obtêm um benefício ainda maior ao utilizarem infra-estruturas completas como painéis e motores. Além disso, os projetistas reutilizam metodologias de construção, estratégias de design consagradas e provadas bem sucedidas e teorias que explicam novos avanços em tecnologias específicas (ex.: injeção eletrônica).

No contexto desse trabalho, componentes de código são vistos apenas como componentes no nível mais baixo de granularidade. Portanto, a semântica de componente se refere tanto a componentes de código (como objetos, algoritmos e estruturas de dados), quanto a padrões de projeto (*desing patterns*), infra-estruturas de domínio (*frameworks*), e a estruturas de representação do conhecimento (ontologias). É importante, portanto tornar clara a diferença e o relacionamento entre esses vários tipos de componentes.

2.4.1 – *Frameworks*

Um *framework* pode ser definido como (Johnson, 1997):

- Um projeto reutilizável de uma ou todas as parte de um sistema, que é representado por um conjunto de classes e pela maneira que elas interagem.
- Um esqueleto formado por elementos abstratos de uma família de aplicações, passível de ser adaptado para atender necessidades de uma aplicação específica.

As definições, na verdade, não são conflitantes: a primeira descreve a estrutura de um *framework*, enquanto a segunda descreve seu propósito.

Johnson (1997) compara *frameworks* fazendo uma analogia a geradores de aplicações, no sentido de que ambos compilam uma linguagem de alto nível, específica de domínio, em uma infra-estrutura concreta. Nesse sentido, um *framework* é também uma arquitetura específica de domínio, capaz de estruturar entidades e relações comuns

ao domínio e, conseqüentemente, de prover facilidade de comunicação, uniformidade e padronização.

A relação entre *frameworks* e componentes de código decorre do fato de serem tecnologias complementares. Em primeiro lugar, *frameworks* fornecem um contexto de reuso para componentes de código, provendo uma maneira padronizada para manipulação de erros e eventos, para troca de dados e para troca de mensagens entre eles. Por outro lado, *frameworks* facilitam o desenvolvimento de novos componentes. Não importa quão boa é uma biblioteca, sempre será necessária a construção de novos componentes. Nesse sentido, *frameworks* oferecem padrões (*templates*) e especificações para construção de novos componentes, assim como a possibilidade de sua construção de componentes a partir de outros menores. Por fim, *frameworks* se diferenciam de bibliotecas de classes por possibilitarem o reuso também de projeto, em um nível mais alto de abstração.

No sentido de ser um artefato reutilizável de projeto, *frameworks* representam um compromisso entre simplicidade e poder. Geralmente possuem interfaces complexas e requerem um período de compreensão e aprendizado antes de serem aplicados. Por outro lado, constituem um tipo de componente que, quando bem concebidos, são flexíveis e poderosos, e podem reduzir o esforço de desenvolvimento de aplicações adaptadas em ordens de magnitudes. Vários frameworks têm sido implementados atualmente (OpenDoc, OLE, AWT, JavaBeans), quase sempre focando domínios horizontais como: interfaces gráficas com o usuário, acesso ao sistema de arquivos, acesso a repositórios de dados e desenvolvimento de aplicações distribuídas. Isso se deve ao fato de que muitos desses domínios já foram amplamente estudados e compreendidos. O mesmo não é verdade para domínios verticais como aviação, telecomunicações, direito, medicina, entre outros.

Johnson (1997), por exemplo, comenta a alta complexidade envolvida na criação de componentes reutilizáveis (e conseqüentemente os altos custos e longos períodos de desenvolvimento), e prega que a academia deve se concentrar na exploração de domínios verticais e a sua concretização através de frameworks. Apesar disso, várias empresas, como Motorola (Meekel et al., 1997), Verilog (Troy, 1993) e Hewlett Packard (Cornwell, 1996, Rix, 1992), têm relatado experiências bem sucedidas de redução do ciclo de

desenvolvimento de três a quatro vezes, ao construírem arquiteturas abstratas a partir de famílias de produtos, e posteriormente promoverem a especialização de aplicações, a partir de adaptações dessas arquiteturas.

A sua principal diferença em comparação a outras técnicas de reuso de projeto de alto nível (como *templates* e esquemas), é o fato de *frameworks* serem expressos em uma linguagem de programação, ou seja, além de representarem projeto em um nível mais alto de abstração, são também uma espécie de artefato de código. Essa característica traz vantagens e desvantagens. Por um lado, o fato de serem programas faz com que sejam mais fáceis de serem compreendidos e aplicados por programadores, além de não haver necessidade de outras ferramentas além das já utilizadas (ex.: compiladores). Por outro lado, tendem a ser específicos a uma linguagem de programação.

2.4.2 – Padrões de Projeto *(Design Patterns)*

O conceito de padrão foi originalmente proposto por Christopher Alexander para descrição de padrões arquitetônicos. Segundo esse autor, cada padrão descreve um problema que se apresenta recorrentemente em nosso ambiente e descreve o núcleo da solução desse problema, de uma forma que a solução pode ser usada milhões de vezes, sem que seja repetida nem uma vez (Alexander, 1977, 1979).

Em outras palavras, cada padrão representa uma solução genérica (ou seja, reutilizável) para um problema recorrente. E apenas um número relativamente pequeno deles é necessário para capturar a essência de todos os padrões arquiteturais. Os projetos de software e arquitetural possuem várias similaridades (Jia, 2000):

- Ambos são processos criativos que desdobram inúmeras possibilidades de projeto;
- O projeto resultante deve satisfazer às necessidades do cliente e ser factível para os desenvolvedores;
- Os projetistas devem balancear vários requisitos e restrições contrastantes;
- Os projetistas devem buscar algumas qualidades intrínsecas e dificilmente quantificáveis, como elegância e extensibilidade.

O trabalho pioneiro em padrões de projeto para software (*design patterns*) foi feito por Gamma et al. (1995) que publicou, em 1995, o primeiro catálogo de padrões.

Esse catálogo compila 23 padrões genéricos e independentes de domínio, que são comumente usados na descrição de problemas de projeto de software, classificando-os como padrões de criação, estruturais e comportamentais. Além disso, o catálogo provê uma descrição de cada padrão, consistindo de algumas das seguintes seções: nome do padrão, categoria (criação, estrutural ou comportamental), outro nome pelo qual o padrão é conhecido, aplicabilidade, custos e benefícios da sua utilização, pré-requisitos e estrutura – uma classe ou diagrama que representa os participantes do padrão e seus relacionamentos.

Em resumo, os principais propósitos do uso de padrões de projeto em software são: (a) capturar, documentar e posteriormente compartilhar a experiência adquirida no projeto de alguma solução, (b) desenvolver software a partir de soluções maduras e confiáveis e (c) prover um vocabulário comum de comunicação sobre projeto de software.

Apesar das similaridades entre *patterns* e *frameworks*, no sentido de que ambos são mecanismos para a captura de projeto reutilizável, eles são bastante diferentes e se relacionam de várias formas:

- Algumas especificações de *frameworks* têm sido implementadas por várias vezes e, nesse sentido, representam também um *pattern* seguindo a definição dada anteriormente. Um exemplo disso é o *framework* de interface com o usuário Model/View/Controller que é apresentado em (Brushman, 1996) como um *pattern*.
- Como mencionado anteriormente, *patterns* são descrições esquemáticas reutilizáveis para problemas de projeto e não programas concretos, e, portanto, independentes de linguagem. Por outro lado, *frameworks* são uma fonte para reuso de projeto e código e são, em última instância, programas compilados e escritos em uma linguagem específica de programação. Nesse sentido, *patterns* são mais abstratos que *frameworks*.
- Como um *framework* é um artefato de código, é necessário que eles passem por uma fase de projeto. Durante essa fase, é comum encontrar situações nas quais *patterns* podem ser reutilizados como uma solução. Nesse sentido, *patterns* são menores que um framework e podem ser vistos como elementos micro-

arquiteturais. No *framework* Model/View/Controller, por exemplo, são usados três grandes patterns (Observer, Strategy e Composite), além de outros menos importantes (Johnson, 1997).

Por fim, *frameworks* estão em um nível intermediário das técnicas de reuso. Eles são mais abstratos e flexíveis que componentes de código e são mais concretos e fáceis de serem reutilizados que projeto puro (mas menos flexíveis e menos prováveis de serem aplicados). A definição que melhor expressa todas estas características discutidas é a presente em (Jia, 2000): um *framework* é um conjunto de classes cooperantes, que são partes de um sistema semicompleto e que representam estratégias reutilizáveis de projeto de software em um domínio particular de aplicação.

Seguindo a definição acima, a semântica de *frameworks* nesse trabalho diz respeito a um artefato de projeto e código que refletem entidades e relações relevantes em um dado domínio de aplicação. A construção de um bom *framework*, no entanto, vem se mostrando um objetivo bastante complexo de ser alcançado. Os principais fatores dessa complexidade são: (i) a dificuldade de identificação do conjunto apropriado de elementos que possam representar o domínio em questão; (ii) a dificuldade de tradução de um modelo de domínio em uma infra-estrutura computacional que contemple requisitos de qualidade pré-estabelecidos. Deste modo, para que *frameworks* de qualidade sejam então construídos, é necessária a adoção de uma disciplina bem estruturada de desenvolvimento, também composta de métodos, técnicas, ferramentas e de um plano de qualidade. Essa disciplina é chamada de *Engenharia de Domínio*. Apesar de *frameworks* se tratarem de uma tecnologia de reuso orientada a objetos, a discussão nessa seção aborda a questão da engenharia de domínio segundo uma perspectiva mais ampla, na medida do possível, sem amarração a nenhum paradigma de desenvolvimento.

2.5 –Definição do modelo de processo

A tabela 2.1 compara os processos de engenharia de software e engenharia de domínio. Essa analogia apresentada em (Arango & Prieto-Diaz, 1994), considera a engenharia de domínio como uma engenharia de software atuando em um meta-nível, ou seja, ao invés de possuir atividades e procedimentos e gerar artefatos necessários à construção de aplicações específicas, o objeto de estudo são famílias de aplicações.

Engenharia de Software	Engenharia de Domínio
Análise de requisitos	Análise de domínio
Especificação do sistema	Especificação da infra-estrutura
Projeto e implementação	Implementação da infra-estrutura

Tabela 2.1 - Comparação entre a engenharia de software a engenharia de domínio (Arango, 1994b).

Na verdade, mais do que uma simples analogia, essas duas disciplinas se relacionam intimamente para contemplar um modelo genérico de desenvolvimento para/com reuso, como é mostrado na figura 2.2.

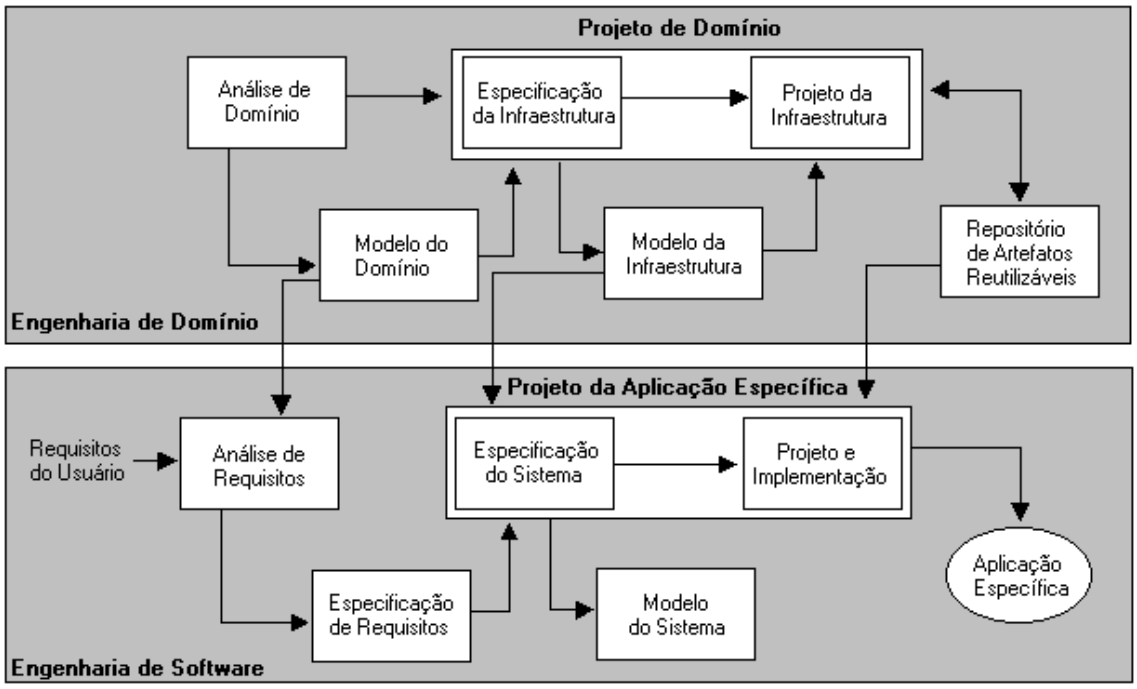


Figura 2.2 – Um modelo genérico de desenvolvimento para/com reuso

Como pode ser observado, o objetivo da engenharia de domínio é, durante o processo de conversão do conhecimento de uma certa comunidade em um repositório de componentes, produzir um conjunto de artefatos passíveis de utilização na engenharia de software.

Ao conjunto das atividades de *Especificação e Implementação da Infra-estrutura* é geralmente dado o nome de *Projeto de Domínio*. O principal produto dessa fase é um *framework* reutilizável, que é posteriormente especializado na fase de *Projeto da Aplicação Específica*, para concretizar uma nova aplicação, cujos requisitos foram especificados numa atividade de *Análise de Requisitos da Aplicação Específica* (Cima & Werner, 1997). Para que um *framework* represente as características do domínio de aplicação para o qual foi construído, é necessário que essas características sejam capturadas em um modelo do domínio através da realização da fase de *Análise de Domínio*.

A análise de domínio tem como objetivo identificar e organizar o conhecimento sobre alguma classe de problemas – o domínio do problema – para suportar a sua descrição e solução. Essa fase é amplamente discutida no capítulo seguinte deste trabalho, no qual os mais difundidos métodos são analisados e suas principais limitações são levantadas. O capítulo é concluído apresentando uma nova proposta de análise de domínio, utilizando ontologias de domínio para capturar o conhecimento envolvido em um nível mais alto de abstração que o alcançado pelos modelos conceituais usualmente empregados nessa fase.

O modelo do domínio é um produto intermediário entre a análise de domínio e a especificação da infra-estrutura. Essa especificação deve definir quais aspectos do domínio do problema devem ser suportados pelo repositório de componentes. Esse processo envolve a seleção das funcionalidades que devem ser traduzidas e como essas funcionalidades aparecerão na infra-estrutura no que diz respeito à hierarquia de componentes, e ao encapsulamento de funcionalidades, entre outras coisas. São especificados entidades e seus possíveis relacionamentos, suas características e atividades (ou serviços realizados). Essa especificação deve ser feita de forma objetiva, mostrando claramente a responsabilidade de cada um desses elementos no domínio do problema. Além disso, em alguns casos, é importante representar a obrigatoriedade (ou opcionalidade) de cada um desses elementos para o domínio, isto é, se um elemento passa a ser obrigatório a partir da presença de outro, ou se dois elementos são mutuamente excludentes. Essas informações auxiliam o desenvolvedor na especialização do *framework* na fase de projeto da aplicação específica (Cima & Werner, 1997). Como

resultado, esse processo contempla uma arquitetura que especifica o que estará disponível para reuso e os mecanismos necessários para que isso possa acontecer (mecanismos de adaptação). Desta forma, fica claro que alguns dos itens do modelo de domínio serão reutilizados como componentes de software e outros como componentes de conhecimento que guiarão decisões estratégicas no desenvolvimento com reuso.

É importante ressaltar que o modelo de domínio é também importante para a fase de análise da aplicação específica, ajudando na compreensão e na comunicação acerca do domínio e, conseqüentemente, contribuindo fortemente para a elicitação dos requisitos.

Como um *framework* é também um artefato de código, é necessário que exista, anteriormente a sua implementação, uma atividade de projeto de software (projeto arquitetural e detalhado). Nessa fase, são realizados o projeto e a implementação do conjunto de características provindo da especificação. Além das atividades comuns a fase de projeto no contexto da Engenharia de Software - são realizadas atividades como generalizações, especializações e recomposições que visem atingir uma estrutura mais estável.

Durante o Projeto e Implementação da infra-estrutura, o reuso de componentes também acontece e em vários níveis. Primeiro, podem ser agrupados ao *framework* vários padrões que por ventura possam oferecer soluções para problemas de projeto. Também podem ser utilizados componentes de código durante a fase de implementação. Por fim, podem existir casos em que um framework integra outros *frameworks* menores ou mais específicos.

No capítulo 4, é apresentado um método de derivação de *frameworks* de forma semi-automática a partir de ontologias de domínio. Depois de definidos, na fase de especificação, quais os elementos da ontologia deverão ser mantidos na infra-estrutura computacional, elementos ontológicos como conceitos, relações e o próprio conhecimento do domínio (representado através de axiomas formais), são incorporados à estrutura do *framework* para formar classes, relacionamentos entre classes, pré-condições e *invariantes*[3], implementadas de forma explícita no corpo dos métodos.

[3] Invariantes são expressões lógicas sempre avaliadas como verdadeiras. Em algumas linguagens como Eiffel, elas podem ser implementadas através de mecanismos embutidos da linguagem (Interactive…, 1999).

Depois de implementado, o *framework* deve ser classificado e armazenado em um repositório de componentes. O desenvolvedor deve, também, prover uma documentação para outros desenvolvedores que o utilizarão. Uma boa opção para este tipo de documentação é o padrão utilizado pela comunidade de padrões de projeto, discutido na seção anterior (Cima & Werner, 1997).

Como pode ser observado, em um modelo de desenvolvimento para/com reuso, tanto a atividade de Projeto de Domínio, quanto a de Projeto da Aplicação específica, dependem fortemente da existência de um repositório de componentes, cujo gerenciamento requer o suporte adequado às operações de armazenamento, organização, busca e recuperação.

Além das interações fundamentais mostradas na figura 2.2, existem outras interações importantes entre as fases de Projeto de Domínio, Análise de Domínio e Projeto da Aplicação Específica. Estas interações dizem respeito à necessidade de se manter atualizado o *framework* como estrutura de representação do domínio:

1. À medida que o domínio da aplicação vai evoluindo, a infra-estrutura deve ser adaptada às novas características desse domínio (interação entre análise e projeto de domínio)
2. À medida que os *frameworks* vão sendo especializados para formar aplicações concretas, estas podem apontar novas características que devem ser incorporadas ao modelo do domínio. Como discutido no capítulo 3, uma das fontes de obtenção do conhecimento do domínio é através da reengenharia de aplicações existentes (interação entre projeto da aplicação específica e análise de domínio);
3. Da mesma forma, a partir da tentativa de especialização de um *framework*, é possível identificar características que possam ser modificadas, a fim de melhorar seu grau de reutilização (interação entre projeto da aplicação específica e projeto do domínio).

É importante que, na engenharia de domínio, as fases de análise de domínio e especificação/implementação da infra-estrutura sejam divididas. Apesar de não serem absolutamente estáveis, domínios presentes no mundo real mudam muito lentamente de forma gradual e monotônica, sendo esse um dos fatores que fazem com que esta abordagem de reuso orientada a domínio seja tão vantajosa. A infra-estrutura, por outro lado, pode evoluir mais rapidamente para alcançar uma mais fácil integração, para

melhorar o desempenho ou para refletir mudanças de cunho tecnológico, como por exemplo, independência de plataforma ou distribuição.

Na fase de projeto da aplicação específica, são selecionados um ou mais componentes passíveis de utilização, de acordo com os requisitos identificados na fase de análise da aplicação específica. Normalmente, um dos *frameworks* representa uma arquitetura genérica para um domínio vertical (o próprio domínio ao qual a aplicação pertence), enquanto que os demais representam arquiteturas para domínios horizontais (ex. interface com usuário, distribuição, entre outros). Para concretização da aplicação, algumas vezes é necessária a modificação de algumas partes da infra-estrutura, além de implementação de outras classes que fazem a conexão do *framework* com o restante da aplicação. No entanto, é importante ressaltar a necessidade de compreensão dos efeitos colaterais que poderão advir dessas modificações. É interessante, também, que o desenvolvedor possa reorganizar a estrutura de classes específicas da aplicação, abstraindo algumas de suas características, a fim de promover uma forma de reutilização local.

Finalmente, tanto para a fase de engenharia de domínio, quanto para a engenharia de software, o modelo de ciclo de vida sugerido é o Modelo de Montagem de Componentes (figura 2.3). Esse modelo é proposto por ser inerentemente evolutivo e orientado à construção de sistemas a partir de componentes existentes. Da maneira que é apresentado em (Pressman, 1997), esse modelo diz respeito apenas a componentes de código. No entanto, no contexto deste trabalho, como citado anteriormente, o conceito componente tem um aspecto bem mais abrangente.

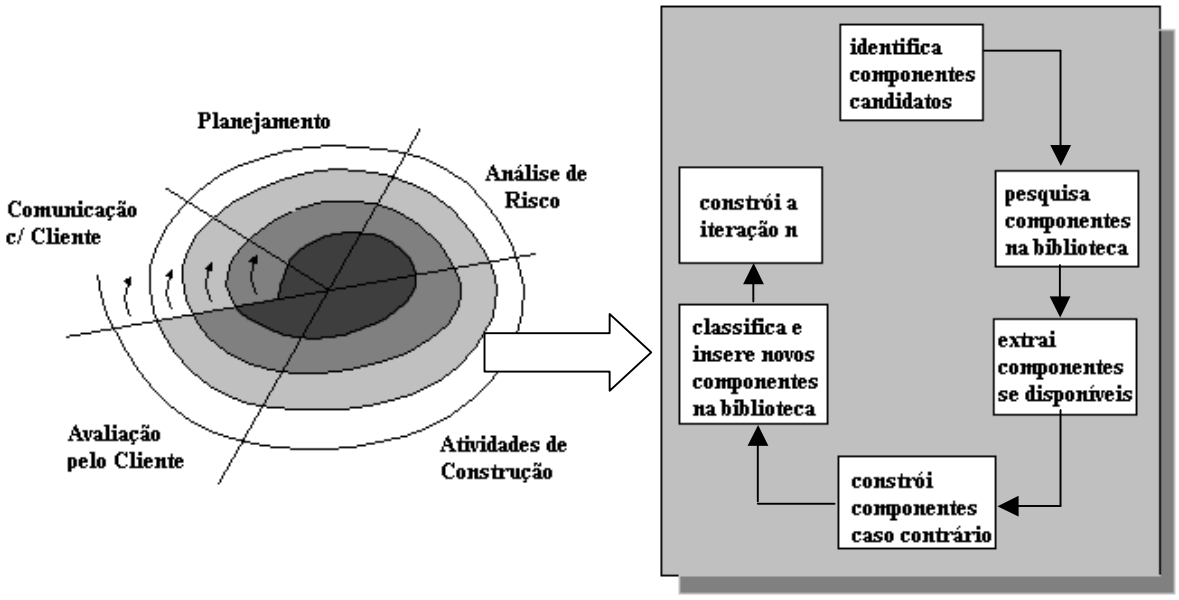


Figura 2.3 - O modelo de ciclo de vida Modelo de Montagem de Componentes

2.6 – Conclusões

Este capítulo discutiu os esforços da engenharia de software na busca de estabelecer um processo maduro e bem estruturado de desenvolvimento, equiparável a outras disciplinas de engenharia. Desde a década de sessenta, quando da identificação da chamada crise de software, a comunidade vem buscando aplicar ao processo de produção de software conceitos extremamente intuitivos às outras engenharias, como por exemplo, a idéia de sistemas abertos e interoperáveis, a prática de reutilização e o uso de modelos fundamentados em teorias matemáticas. É inegável que a área demonstrou uma constante evolução ao longo desses anos, com o advento de linguagens e paradigmas de desenvolvimento capazes de propiciar um nível mais adequado de abstração e com introdução de atividades de suporte como documentação, análise de risco e avaliação da qualidade.

Desde as discussões preliminares sobre o conceito de reutilização, ele sempre foi apontado como uma possível solução para a crise do software. Apesar de ser inegável que tenha acontecido de alguma forma, a reutilização se desenvolveu de maneira ad hoc, e praticamente se limitou ao nível de código.

Este capítulo defende a integração sistemática da prática de reutilização nos processos de desenvolvimento, associando de maneira complementar as perspectivas de

desenvolvimento *para* e *com* reuso. Na abordagem da primeira perspectiva, é adotado o emprego da engenharia de domínio, uma espécie de engenharia de software atuando em um meta-nível, capaz de desenvolver infra-estruturas reutilizáveis que capturam a estrutura e o conhecimento de uma família de aplicações. Uma vez construídas, essas infra-estruturas podem ser reutilizadas por atividades de um processo de engenharia de software, promovendo assim o desenvolvimento com reuso.

A engenharia de domínio é composta pelas seguintes etapas: análise de domínio, projeto de domínio e implementação de uma infra-estrutura de reuso. A análise de domínio é profundamente discutida no próximo capítulo, no qual é apresentada uma nova abordagem para sua realização, através da utilização de ontologias formais de domínio. As demais etapas são discutidas no capítulo 4, no qual é proposto um conjunto de regras de mapeamento da estrutura e do conhecimento, presentes na ontologia, em uma arquitetura de um *framework* orientado a objetos.

Capítulo 3

Análise de Domínio e Ontologias

...o que observamos não é a natureza
em si própria, mas a natureza exposta
ao nosso método de questionamento
Werner Karl Heisenberg

3.1 - Introdução

No capítulo anterior foi apresentado um modelo de processo de desenvolvimento de software em que as perspectivas de desenvolvimento *para* e com *reuso* são associadas de maneira complementar. No desenvolvimento *para* reuso, foi observada a importância do conceito de *produto reutilizável* ser pensado de forma mais abrangente, ou seja, ao invés de ser explorada apenas em nível de código, a reutilização deveria acontecer através de artefatos situados em níveis mais altos de abstração, e que portanto permitissem o reuso de itens como soluções de projeto, infra-estruturas de domínio e idealmente estruturas de conhecimento.

Se comparada com o reuso de componentes de código, a abordagem de reutilização de infra-estruturas de domínio insere no processo inúmeras vantagens, sendo as principais delas ligadas à comunicação e o entendimento do domínio em questão, a captura da experiência de casos similares e, no caso de um processo formal de

desenvolvimento, a obtenção de um estrutura estável, confiável, robusta e com alto grau de mantenabilidade. No entanto, a obtenção desses benefícios dependente da qualidade da infra-estrutura utilizada, que por sua vez dependente diretamente de modelo de domínio correspondente.

Dessa forma, a complexidade de identificação, captura e organização dos elementos relevantes para representação do conhecimento embutido em uma classe de problemas impõe a necessidade de definição de um processo bem estruturado de análise de domínio.

Este capítulo tem como objetivo explorar a atividade de análise de domínio, apresentando seus principais conceitos, objetivos e métodos. São também discutidas as principais limitações desses métodos e dos modelos por eles gerados, principalmente no que diz respeito à representação explícita do conhecimento. Por fim, várias das características analisadas são agrupadas na definição de um novo método, fundamentado no uso de ontologias formais de domínio, propondo uma nova abordagem para a realização dessa atividade.

O capítulo é dividido da seguinte forma: a próxima seção define análise de domínio e seus principais conceitos envolvidos e finaliza estabelecendo um modelo de processo comum aos métodos mais difundidos. A seção 3.3 apresenta os conceitos de ontologias e mais especificamente de ontologias de domínio, também finalizando com a definição de um processo para construção de ontologias. A seção 3.4 mostra porque ontologias de domínio podem ser usadas de forma bastante satisfatória para a realização da análise de domínio, inclusive acrescentando importantes contribuições. Finalmente, a seção 3.5 apresenta as conclusões do capítulo.

3.2 - Análise de Domínio

A disciplina de análise de domínio começou a ser considerada importante pela comunidade de engenharia de software a partir da necessidade de se diminuir o custo desproporcional da manutenção de software, resultante da introdução de mudanças arbitrárias (Arango, 1994), assim como do consenso relacionado à importância do desenvolvimento para reuso e com reuso neste cenário.

O termo *Análise de Domínio* foi introduzido por Neighbors (1981) com a seguinte definição:

A Análise de Domínio é uma tentativa de identificar os objetos, operações e relações entre o que peritos em um determinado domínio percebem como importante.

Se considerarmos, por exemplo, o domínio de transporte aéreo, objetos típicos são assentos, vôos, aeroportos e tripulação. Operações e ações incluem escalonamento de vôos, reserva de assentos e relacionamento entre tripulação e vôos. Uma linguagem específica de domínio pode ser criada para representar estes objetos, operações e relações, que pode, posteriormente, ser utilizada para descrever outros sistemas neste domínio.

Intuitivamente, esta atividade pode ser considerada equivalente à análise de requisitos convencional, no entanto atuando em um meta-nível e, portanto, ao invés de explorar requisitos de uma aplicação específica, os requisitos explorados dizem respeito a uma família de aplicações de uma determinada área (Arango & Prieto-Díaz, 1994).

Apesar desta definição informal do termo contribuir com uma boa idéia inicial, uma definição mais rigorosa se faz necessária como base para as discussões que aparecem no decorrer do capítulo. A seguir são descritos alguns termos necessários à construção desta definição.

3.2.1 - Conceitos de Análise de Domínio

a) Domínio do problema: O domínio do problema representa um conjunto de itens de informação presentes em um certo contexto do mundo real, interrelacionados de forma bastante coesa, e que desperta o interesse de uma certa comunidade. Esta definição cobre duas perspectivas (Arango, 1994):

(1) Domínio do problema como um conjunto de problemas relacionados, o que aproxima a análise de domínio da teoria de problemas;

(2) Domínio do problema como uma taxonomia de componentes que torna explícita as partes comuns de aplicações presentes e futuras identificadas como similares.

b) Modelo do Domínio: Pode ser descrito como um sistema formal de termos, relações entre termos, regras de composição de termos, regras para raciocínio usando estes termos e regras para mapeamento de itens do domínio do problema para

expressões neste modelo e vice-versa. O modelo do domínio define entidades, operações, eventos e relações que abstraem similaridades e regularidades em um determinado domínio, formando uma arquitetura de componentes comuns às aplicações analisadas e criando modelos que tornam possível identificar, explicar e prever fatos difíceis de serem observados diretamente. Depois de pronto, este modelo deve servir como uma fonte unificada de referência, quando ambigüidades surgirem em discussões sobre este domínio, e como um repositório de conhecimento comum, auxiliando de forma direta a comunicação, o aprendizado e reuso em um nível mais alto de abstração (Arango, 1994).

c) Análise e Modelagem do Domínio: Um conjunto de atividades cujo propósito é reduzir a complexidade da nossa percepção de um determinado domínio, impondo uma coerente organização aos dados adquiridos através de experimentos, elicitação de especialistas e engenharia reversa de sistemas existentes (Arango, 1994).

3.2.2 - Um processo comum para Análise de Domínio

Como enfatizado por Neighbors (1981), a chave do desenvolvimento de software com reutilização é capturada pela análise de domínio com seu foco abrangente de promover o reuso de itens mais abstratos do que somente código. Entre estes itens podem ser citados como exemplos arquiteturas de software, soluções de projeto, modelos de requisitos e, idealmente, conhecimento. No entanto, essa atividade se mostrou complexa e geralmente cara, dependendo da criação de toda uma cultura de apoio no ambiente em que o desenvolvimento para reuso e com reuso vá ocorrer. Além disso, o sucesso depende diretamente do modelo gerado no que diz respeito à relevância e à qualidade da representação dos itens de informação, bem como de suas relações.

Em (Arango, 1994), é salientada a dificuldade de se identificar, capturar e organizar estes itens apropriados de informação, de forma que se torna clara a necessidade de um processo bem definido e estruturado. Deste modo, a análise de domínio toma a forma de um conjunto interdisciplinar que envolve adaptações de técnicas comumente usadas nas áreas de engenharia de software, engenharia de requisitos, modelagem conceitual, aquisição de conhecimento e representação de conhecimento, podendo ser vista como versão em meta-nível da engenharia de requisitos

e como um processo de engenharia do conhecimento direcionado a suportar uma tarefa particular de resolução de problemas.

Em (Arango, 1994), vários dos mais difundidos métodos de análise de domínio são analisados, entre eles:

- *McCain's Product-Oriented Paradigm*
- *Prieto-Díaz Domain Analysis for reusability*
- *Simon's Domain Analysis for building a Organon*
- *SEI's Feature-Oriented Domain Analysis*
- *Software Productivity Consortium Domain Analysis*
- *Lubar's Domain Analysis in Intelligent Design Aid*
- *Vitaletti and Guerrieri*
- *Bailin*

Através da análise desses métodos e de métodos mais recentes como o apresentado em (Cornwell, 1996), um processo comum pode ser descrito. A seguir são apresentadas as principais fases e atividades deste processo:

a) **Planejamento:** Um sistema para reuso sempre existe no contexto de uma organização e, portanto, a análise de domínio é fortemente marcada por prioridades e restrições do próprio negócio. Desempenho, prioridades e eficiência são estabelecidos e medidos com relação aos objetivos da organização (Cornwell, 1996). Esta fase é marcada por atividades como análise do negócio e análise de risco. As primeiras perguntas as serem feitas são: Vale a pena fazer análise de domínio neste caso, considerando custos e benefícios? O domínio é conhecido e estável o bastante? A seguir é feita a caracterização do problema, com definição do escopo do domínio (o que é? o que é relevante?), da abordagem a ser utilizada e das métricas que identificarão a adequação do processo aos objetivos do negócio. O produto final desta etapa é uma especificação de requisitos do domínio.

b) **Aquisição e seleção dos dados:** Aqui são identificadas as fontes de dados que são disponíveis. As informações são geralmente capturadas de fontes como: literatura técnica (livros, normas e padrões, artigos, revistas científicas), registros históricos do domínio, entrevistas com especialistas no domínio, e/ou através de

engenharia reversa de sistemas similares existentes e sistemas em áreas similares. Em (Cornwell, 1996), é sugerido como regra geral que sejam analisados pelo menos três sistemas similares e que sejam imaginadas pelo menos três aplicações futuras que poderão se utilizar do reuso a partir do conhecimento modelado. Cada uma destas fontes têm vantagens e limitações. Peritos são excelentes fontes para modelos conceituais gerais e, geralmente, são a única fonte de justificativas e explicações de "porque as coisas são como são". A memória de um especialista é, geralmente, repleta de preciosas informações históricas que não podem ser encontradas em outras fontes. Por outro lado, peritos humanos representam um recurso escasso e caro e possuem características humanas como fadiga e limitação de tempo. A literatura técnica provê uma fonte precisa e detalhada de informações, além de ser geralmente barata e facilmente disponível. No entanto, livros e artigos não podem ser utilizados como fontes de justificativas, elaborações, dados históricos e principalmente de "*insights*". Resumindo, as fontes de informações devem complementar umas às outras (Arango, 1994).

c) **Análise dos dados e modelagem do domínio :** Nesta etapa, o conhecimento capturado é inicialmente avaliado quanto à consistência, correção e completude e, então, modelado identificando entidades, relações, funções e axiomas comuns às diversas fontes analisadas. O modelo produzido pode variar em complexidade e formalidade, desde uma simples taxonomia a uma rica estrutura de conhecimento formada, por exemplo, por redes semânticas, *frames* ou axiomas formais. De forma geral, o modelo do domínio é composto por um modelo conceitual (que mostra a presença de entidades do domínio e como elas se relacionam) e por um léxico (ou dicionário) do domínio. Este léxico apresenta da maneira menos ambígua possível as definições dos elementos que compõem o modelo conceitual, desempenhando um papel fundamental quanto à economia de tempo, minimização dos problemas de comunicação e, conseqüentemente, promoção de um diálogo mais eficiente e consistente acerca do domínio modelado. Outras técnicas usuais nesta etapa são modelos de entidades e relacionamentos, modelos de objetos, pré-condições e invariantes. No entanto, é importante ressaltar que o modelo produzido não deve refletir decisões de implementação. Esta etapa finaliza com a definição da hierarquia,

abstração e classificação das entidades. Esta atividade é baseada em fatos como interdependência ou exclusão mútua das entidades selecionadas.

3.3 - Representação do conhecimento e Ontologias

Acredita-se, hoje em dia, que a representação formal do conhecimento tenha começado na Índia do primeiro milênio A.C. com o estudo da gramática de *Shastric Sanscrit*. No entanto, da forma como a vemos atualmente, esta disciplina tem um relação muito próxima dos trabalhos realizados na Grécia antiga, principalmente por *Aristóteles* (384-322 A.C.) nos campos da lógica, ciências naturais e filosofia metafísica (Russel & Norvig, 1995).

As primeiras discussões no campo da Inteligência Artificial focavam a questão da representação do ponto de vista do *problema* e não do *conhecimento*. Com a proliferação dos sistemas especialistas, a representação do conhecimento era feita com o objetivo claro de extrair o conhecimento do perito e formalizá-lo em uma base de conhecimento, ou seja, a mente do perito era vista como um "mina" e o papel do engenheiro de conhecimento era explorá-la. Por outro lado, a maior parte dos esforços para incorporar conhecimento aos sistemas concentrava-se na construção de mecanismos uniformes e gerais de representação. A forma como este processo era conduzido teria uma influência direta em alguns aspectos (Falbo, 1998) :

- Uma vez que a máquina de inferência era de propósito geral, a estratégia para resolver um problema era embutida como parte da base de conhecimento. Desta forma, era praticamente impossível separar os conhecimentos do domínio, da aplicação e da tarefa a ser realizada, tornado a reutilização do conhecimento praticamente inviável. O conhecimento do domínio não podia ser usado em outras aplicações dado que era adquirido para uma tarefa específica.
- O problema da reutilização era ainda agravado pelo modo no qual o conhecimento é associado e conseqüentemente disponibilizado por parte dos peritos. O conhecimento elicitado de especialistas em entrevistas é disponibilizado de forma bastante compilada através de heurísticas, o que

dificulta a separação dos seus diversos tipos e praticamente inviabiliza o seu reuso.

- O uso do conhecimento do especialista como única fonte de conhecimento é, por si só, uma falha. Como foi discutido anteriormente, outras fontes de conhecimento, como literatura técnica e sistemas existentes, desempenham papéis igualmente importantes, devendo ser utilizadas de forma complementar. Ao relegar estas fontes, a estratégia de transferência do conhecimento do especialista para o sistema não apenas tornava a tarefa de aquisição mais difícil, como também reforçava o problema da superficialidade.

Nesta época, toda vez que um sistema especialista tivesse que ser construído em um mesmo domínio, mas com o objetivo de realizar uma diferente tarefa, todo o processo de elicitação e codificação do conhecimento deveria ser refeito, expondo o processo a erros e inconsistências que já poderiam ter sido resolvidas, além de provocar perda de tempo, esforço e conseqüentemente recursos.

Diante desta situação, surge, então, a necessidade de uma nova abordagem para construção desta classe de sistemas, buscando um processo que pudesse modelar e isolar os diferentes tipos de conhecimento, possibilitando o reuso em seu mais alto nível de abstração: *o reuso de conhecimento*. Clancey (1993) propõe a mudança desta perspectiva, argumentando que o foco da Engenharia de Conhecimento deve ser a modelagem de sistemas e não a tentativa de reproduzir a maneira como os especialistas raciocinam, defendendo a visão de que uma base de conhecimento deve ser vista como um produto de uma atividade de modelagem e não um repositório de conhecimento especializado. Desta forma, a modelagem passa a ser o aspecto central da Engenharia de Conhecimento e a aquisição de conhecimento passa a ser essencialmente um processo construtivo, no qual o engenheiro de conhecimento usa todos os tipos de informação disponíveis e estabelece as decisões finais de modelagem. Dentro da comunidade de representação do conhecimento surgiu, então, um grupo de defensores da idéia de que o conhecimento embutido em uma determinada porção da realidade poderia (e deveria) ser representado em um nível de abstração tal que fosse independente e reutilizável ao longo de várias tarefas (Guarino, 1997). Ao adotar este paradigma, esta comunidade entrou em um território anteriormente

explorado unicamente por filósofos da ciência e da linguagem, fazendo com que, devido à imposição de sua disciplina, esta área fosse investigada de forma mais rápida e profunda do que quando era um domínio exclusivo da filosofia. Ao produto desta área inicialmente criada por Aristóteles com seu abrangente sistema de classificação, taxonomização e de representação do conhecimento de forma geral, chamamos hoje de *Ontologias*.

3.3.1 - O que são Ontologias ?

De acordo com o dicionário Webster (Woolf, 1981), a palavra "ontologia" pode ser definida como uma teoria particular que diz respeito à natureza dos seres e das coisas em si. Já há bastante tempo, filósofos têm usado ontologias para descrever domínios naturais, ou seja, as coisas naturais do mundo como os tipos de existência e as relações temporais. Apesar de sua difusão e do longo tempo em que vem sendo usado, ainda não há um consenso (principalmente na comunidade de Ciência da Computação) sobre a semântica do termo "ontologia". Em alguns casos, ele é usado apenas como um nome mais rebuscado, denotando o resultado de atividades familiares como modelagem de domínio e análise conceitual. No entanto, em muitos outros casos, as ditas ontologias apresentam algumas peculiaridades como a forte ênfase na necessidade de uma abordagem altamente formal e interdisciplinar, na qual a filosofia e a lingüística desempenham um papel fundamental.

No sentido filosófico, o termo "ontologia" é referido como um sistema particular de categorias que versa sobre uma certa visão do mundo e pode ser visto como um sinônimo de metafísica. Seu propósito é classificar as entidades de uma porção da realidade, definindo seu vocabulário e as formulações canônicas de suas teorias (Smith, 2000). Desta forma, este sistema não depende de uma linguagem particular. Por exemplo, uma ontologia de Aristóteles é sempre a mesma, independente da linguagem usada para expressá-la. Por outro lado, para a comunidade de Ciência da Computação, o termo se refere a um artefato de engenharia, constituído de um vocabulário de termos organizados em uma taxonomia, suas definições e um conjunto de axiomas formais usados para criar novas relações e para restringir as suas interpretações segundo um sentido pretendido (Noy & Hafner, 1997). Apesar da relação entre essas duas definições, com o intuito de resolver este impasse terminológico, em (Guarino, 1998), Guarino propõe que a definição

da comunidade de computação seja adotada para o termo "ontologia" e que para a definição filosófica seja dado o nome de *conceituação*.

A conceituação tem uma importância fundamental em qualquer atividade de modelagem do conhecimento, pois é impossível representar o mundo real, ou mesmo uma parte dele, em sua completa riqueza de detalhes. Todo modelo de conhecimento é, portanto, comprometido com alguma conceituação, implícita ou explicitamente (Gruber, 1995). Para representar um certo fenômeno ou parte do mundo, a que chamamos domínio, é necessário concentrar a atenção em um número limitado de conceitos, suficientes e relevantes, para criar uma abstração do fenômeno em questão.

Um conceito é uma idéia ou noção que aplicamos a elementos do domínio, e que possui dois aspectos importantes - *intenção* e *extensão*. A *intenção* do conceito é seu significado ou sua completa definição, enquanto que a *extensão* é o conjunto de elementos de um determinado universo para os quais o conceito se aplica. Por exemplo, o conceito *Mortal* pode ser definido como "tudo que morre". A extensão desse conceito seria então o conjunto de coisas de um determinado universo que possuem a característica de morrer (Russel, 1938).

Segundo Guarino (1998), uma conceituação pode ser definida formalmente do seguinte modo:

> *Seja a estrutura <D,W> tal que D representa o domínio em questão e W representa todos as possíveis estruturações de significado entre os elementos existentes em D. Esta estrutura é chamada de Espaço de Domínios. Uma conceituação C é uma estrutura <D,W,ℜ> sendo ℜ o conjunto de relações escolhidas como pertinentes para representar este domínio. Desta forma, uma conceituação define uma estrutura pretendida do mundo, representada por S. Do ponto de vista filosófico, várias conceituações (e várias estruturas do mundo) podem ser feitas para uma mesmo domínio, a escolha da semântica dada a conceitos e relações será determinada pela natureza do universo de problemas que se tem em mente (Guarino, 1998). Do ponto de vista computacional, para que uma conceituação possa ser efetivamente usada, ela precisa ser especificada em uma determinada linguagem L. A estrutura de S é, então, mapeada para constantes e predicados da linguagem L, seguindo um*

função de interpretação. Este mapeamento ocorre tanto do ponto de vista de intenção quanto de extensão. A esta "interpretação intencional" da estrutura de S é dado o nome de compromisso ontológico.

Gruber (1995) define uma ontologia como sendo uma especificação de uma conceituação. Guarino (1998) estende essa definição dizendo que uma ontologia é na verdade uma especificação parcial e explícita que tenta, da melhor forma possível, aproximar a estrutura de mundo definida por uma conceituação. Uma ontologia, portanto, passa a ter compromisso apenas com a consistência em um determinado domínio e não com a completude. Ao conjunto de elementos de um domínio que podem ser representados em uma ontologia é dado o nome de *universo de discurso*.

Nesse trabalho, o termo ontologia é usado em concordância com a definição de Guarino, ou seja, ontologias são tratadas como um artefato computacional composto de um vocabulário de conceitos, suas definições e suas possíveis propriedades, um modelo gráfico mostrando todas as possíveis relações entre os conceitos e um conjunto de axiomas formais que restringem a interpretação dos conceitos e relações, representando de maneira clara e não ambígua o conhecimento do domínio. É importante realçar que, de posse dessa base de conhecimento formalizada como uma teoria lógica, a ontologia não descreve apenas conhecimento imediato, isto é, conhecimento factual que pode ser obtido diretamente a partir da observação do domínio, mas também conhecimento derivado, ou seja, conhecimento obtido através de inferência sobre o conhecimento imediato disponível. Um modelo de domínio utilizando-se ontologias, portanto, não é somente uma hierarquia de termos, mas uma infra-estrutura teórica que versa sobre o domínio em questão.

3.3.2 - Tipos de Ontologias

Segundo Guarino (Guarino, 1997, 1998), com base em seu conteúdo as ontologias podem ser classificadas nas seguintes categorias:

- *ontologias genéricas*: descrevem conceitos bastante gerais, tais como, espaço, tempo, matéria, objeto, evento, ação, etc., que são independentes de um problema ou domínio particular;

- *ontologias de domínio*: expressam conceituações de domínios particulares, descrevendo o vocabulário relacionado a um domínio genérico, tal como Medicina e Direito.
- *ontologias de tarefas*: expressam conceituações sobre a resolução de problemas, independentemente do domínio em que ocorram, isto é, descrevem o vocabulário relacionado a uma atividade ou tarefa genérica, tal como, diagnose ou vendas;
- *ontologias de aplicação*: descrevem conceitos dependentes do domínio e da tarefa particulares. Estes conceitos freqüentemente correspondem a papéis desempenhados por entidades do domínio quando da realização de uma certa atividade;
- *ontologias de representação*: explicam as conceituações que fundamentam os formalismos de representação de conhecimento.

As ontologias de domínio são construídas para serem utilizadas em um micro-mundo. São o tipo mais comumente desenvolvido, sendo que diversos trabalhos são encontrados na literatura, enfocando áreas como química (Gómez-Pérez et al., 1996), modelagem de empreendimento - TOVE (Toronto Virtual Enterprise) (Gruninger, 2000, Uschold, 1996), *Design* (Varejão, 1999) - DORPA e YMIR (Alberts, 1994), medicina - UMLS (Unified Medical Language System) (Humphreys & Lindberg, 1993), modelagem de processos de software (Falbo, 1998), biologia molecular e bioquímica - GENSIM (Karp, 1993) e ciência dos materiais - PLINIUS (Van der Vet & Mars, 1994, 1995), entre outros.

A pesquisa enfocando ontologias genéricas procura construir teorias básicas do mundo, de caráter bastante abstrato, aplicáveis a qualquer domínio (conhecimento de senso comum). Entre os trabalhos nesta categoria, destacam-se os projetos CYC (Lenat, 1995), WORDNET (Miller, 1990), Generalized Upper Model (Bateman et al., 1994) e as ontologias de John Sowa (1995) e de Kathleen Dahlgren (1995). Estes trabalhos estão bastante alinhados com o uso de ontologias nas áreas filosóficas de categorização e lingüística e procuram descrever a natureza das coisas. Tipicamente, ontologias genéricas definem conceitos tais como coisa, estado, evento, processo, ação, etc., com o intuito de serem especializados na definição de conceitos em uma ontologia de domínio.

Ontologias de representação procuram tornar claros os compromissos ontológicos embutidos em formalismos de representação de conhecimento. Um exemplo desta categoria é a ontologia de *frames*, utilizada em Ontolingua (Gruber, 1992).

O estudo de ontologias de tarefas é a vertente mais recente do estudo de ontologias. Sua principal motivação é facilitar a integração dos conhecimentos de tarefa e domínio em uma abordagem mais uniforme e consistente, tendo por base o uso de ontologias. Trabalhos nesta categoria incluem (Chandrasekaran & Josephson, 1997, Musen et al., 1995).

Guarino (1998) propõe que ontologias sejam construídas segundo seu nível de generalidade, como é mostrado na figura 3.1. Os conceitos de uma ontologia de domínio ou de tarefa devem ser especializações dos termos introduzidos por uma ontologia genérica. Os conceitos de uma ontologia de aplicação, por sua vez, devem ser especializações dos termos das ontologias de domínio e de tarefa correspondentes.

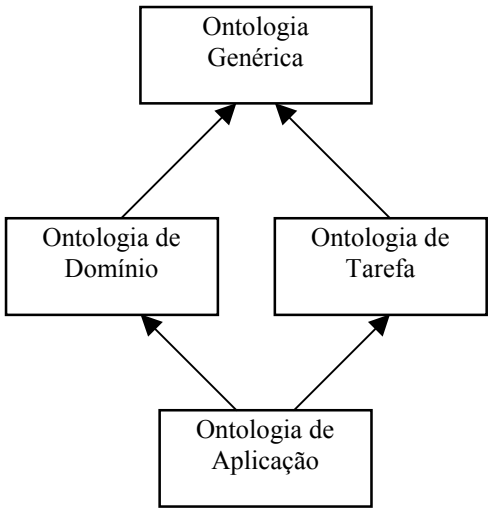


Figura 3.1 - Tipos de Ontologias, segundo seu Nível de Dependência em Relação a uma Tarefa ou Ponto de Vista Particular (Guarino, 1998).

O objetivo deste capítulo é utilizar ontologias de domínio e, conseqüentemente, seu processo de construção para a realização da atividade de análise de domínio. Deste modo, a partir deste momento, o termo *ontologia* será utilizado para se referir a esse tipo particular de ontologia.

3.3.3 – Benefícios e Problemas do uso de ontologias

À medida que tem crescido o interesse por ontologias pela comunidade de Ciência da Computação, elas têm sido utilizadas de diferentes maneiras. Muitas vezes são usadas para descrever domínios já consagrados, como Medicina, Engenharia e Direito, a fim de promover consenso entre a comunidade de agentes interessada no domínio em questão. Outra vezes para promover integração entre bases de conhecimento de Sistemas Baseados em Conhecimento distintos. De forma geral, ontologias constituem uma ferramenta poderosa para suportar a especificação e a implementação de sistemas computacionais de qualquer complexidade. Ao usar esta abordagem na fase de *análise e especificação do domínio,* vários benefícios obtidos, em três principais áreas:

- ***Comunicação*:** ontologias são ferramentas úteis para ajudar as pessoas a se comunicarem, sob várias formas, acerca de um determinado conhecimento. Em primeiro lugar, elas podem ajudar as pessoas a raciocinar e a entender o domínio do conhecimento e, portanto, atuam como uma referência para a obtenção do consenso numa comunidade profissional sobre o vocabulário técnico a ser usado nas suas interações. Além disso, ontologias constituem um excelente guia no processo de elicitação de conhecimento das diversas fontes.

- ***Formalização*:** devido à natureza formal da notação usada, a especificação do domínio elimina contradições e inconsistências envolvendo as restrições, resultando, portanto, em uma especificação não ambígua. Um outro ponto a ser destacado é que, já que uma notação formal é usada, a especificação formalizada pode ser automaticamente verificada e validada, se um provador automático de teoremas existe para aquela notação. Com um mecanismo de inferência, é também possível derivar novos conhecimentos de forma automática, a partir da base de conhecimento já presente na ontologia. Por fim, esta característica torna

possível a obtenção de um processo de geração de infra-estruturas computacionais de maneira sistemática e idealmente automática.

- ***Representação do conhecimento e reuso*:** A ontologia forma um vocabulário de consenso e representa o conhecimento do domínio de forma explícita no seu mais alto nível de abstração, possuindo um potencial enorme de reuso. O conhecimento formalizado na camada de domínio pode ser especializado em diferentes aplicações, servindo diferentes propósitos, por diferentes equipes de desenvolvimento, em diferentes pontos do tempo.

Apesar de todas as vantagens citadas, o uso de ontologias também apresenta alguns problemas. O'Leary (1997), por exemplo, identificou os seguintes impedimentos: (i) A escolha de uma ontologia é um processo político, já que nenhuma ontologia pode ser totalmente adequada a todos os indivíduos ou grupos. (ii) Ontologias não são necessariamente estacionárias, i.e., necessitam evoluir. Poucos trabalhos têm enfocado a evolução de ontologias. (iii) Estender ontologias não é um processo direto. Ontologias são, geralmente, estruturadas de maneira precisa e, como resultado, são particularmente vulneráveis a questões de extensão, dado o forte relacionamento entre complexidade e precisão das definições. (iv) A noção de bibliotecas de ontologias sugere uma relativa independência entre diferentes ontologias. A interface entre elas constitui, portanto, um impedimento, especialmente porque cada uma delas é desenvolvida no contexto de um processo político. Ontologias desenvolvidas independentemente podem não se integrar efetivamente com outras por vários motivos, desde similaridade de vocabulário até visões conflitantes do mundo (Falbo, 1998).

O pior problema, no entanto, é do ponto de vista metodológico. Apesar de uma grande quantidade de ontologias já ter sido desenvolvida por diferentes grupos, sob diferentes abordagens e usando diferentes métodos e técnicas, poucos trabalhos foram publicados sobre como proceder, mostrando as práticas, critérios de projeto, atividades, métodos e ferramentas usadas para sua construção. A conseqüência é clara: a ausência de atividades padronizadas, ciclos de vida e métodos sistemáticos, assim como de um conjunto de critérios de qualidade, técnicas e ferramentas, expõem o desenvolvimento de ontologias aos mesmos problemas comentados no capítulo anterior para a engenharia de

software, ou seja, a sua realização é conduzida de forma artística e não como uma atividade de engenharia (Falbo, 1998).

Da mesma forma que foi feito anteriormente para a atividade de análise de domínio, a próxima seção analisa as principais propostas existentes de metodologias para construção de ontologias a fim de descrever um modelo de processo comum.

3.3.4 - Construção de Ontologias

Algumas propostas de metodologias para construção de ontologias têm sido apresentadas na literatura nos últimos anos, como por exemplo a "metodologia inicial" apresentada por Uschold (Uschold & King, 1995), METHONTOLOGY (Férnandez et al., 1997) e a apresentada no contexto do projeto TOVE (Toronto Virtual Enterprise) (Uschold & Gruninger, 1996). Apesar disso, os modelos apresentados ainda não demostram um processo suficientemente estruturado a ponto de suportar a construção de ontologias como uma verdadeira disciplina de engenharia. Nessa seção, é apresentada uma abordagem sistemática para construção de ontologias, assim com descrita por Falbo (Falbo et al., 1998). Esta abordagem, além de unir as principais características das metodologias citadas, discute as várias atividades do processo de construção de ontologias, apresentando algumas orientações de como proceder na sua realização. As atividades descritas abaixo são organizadas em um ciclo altamente interativo no qual os processos de avaliação da qualidade e documentação ocorrem como atividades guarda-chuva (*umbrella activities*). É importante ressaltar que, devido à complexidade envolvida nas atividades que compõem este ciclo, a área de engenharia de ontologias urge pelo surgimento de ferramentas computacionais (CASE) que possam realizar a automatização (ou pelo menos semi-automatização) do processo.

a) Identificação de Propósito e Especificação de Requisitos

A primeira atividade a ser realizada no processo de construção de uma ontologia é identificar claramente o seu propósito e os usos esperados para ela (Falbo, 1998), i.e., a competência da ontologia. A competência de uma representação diz respeito à cobertura de questões que essa representação pode responder ou de tarefas que ela pode suportar. Ao se estabelecer a competência, temos um meio eficaz de delimitar o que é relevante para a ontologia e o que não é. É útil, também, identificar potenciais usuários e os

cenários que motivaram o desenvolvimento da ontologia em questão. Uma vez definido o propósito, deve-se especificar os requisitos da ontologia. Esses devem contemplar os usos projetados para a ontologia e podem ser expressos em termos de questões de competência: as questões que a ontologia deve ser capaz de responder (Uschold & Gruninger, 1996). Ao se especificar um relacionamento entre as questões de competência e os cenários de motivação, está se dando uma justificativa para a ontologia e, mais importante, se está provendo um mecanismo para sua avaliação. Uma analogia pode ser feita entre o papel que as questões de competência desempenham para a engenharia de ontologias, comparando-o ao dos modelos de casos de uso no contexto da engenharia de software orientada a objetos. Ambas as técnicas norteiam todo o processo de desenvolvimento, auxiliando deste a atividade de especificação de requisitos até a atividade de avaliação.

b) Captura da Ontologia

Esta é, sem dúvida, a etapa mais importante no desenvolvimento de uma ontologia. O objetivo é capturar a conceituação do universo de discurso, com base na competência da ontologia. Os conceitos e relações relevantes devem ser identificados e organizados. Um modelo utilizando uma linguagem gráfica pode ser de grande utilidade para facilitar a comunicação com os especialistas do domínio. Este modelo deve ser acompanhado de um vocabulário de termos (Falbo, 1998).

Conceitos primitivos, isto é, aqueles que não são passíveis de uma definição em termos de outros conceitos da ontologia, devem ser definidos utilizando linguagem natural e exemplos, tomando o devido cuidado para se evitar ambigüidades e inconsistências. A escolha dos termos a serem usados para referenciar as categorias de conhecimento deve ser feita cuidadosamente, evitando-se termos com interpretação duvidosa. Conceitos passíveis de descrição em termos de outros conceitos, devem ser definidos com referências claras a estes, com o objetivo de facilitar a formalização (Falbo, 1998). Deve-se, ainda, construir taxonomias, organizando categorias e sub-categorias interconectadas do conhecimento do domínio de interesse.

Os conceitos e relações formam a base da ontologia. Mas uma característica essencial de ontologias é a definição de axiomas. Simplesmente propor uma taxonomia ou um conjunto de termos básicos, não constitui uma ontologia. Axiomas devem ser

providos para definir a semântica dos termos. Os axiomas especificam definições de termos na ontologia e restrições sobre sua interpretação. Neste momento, não há necessidade de se escrever axiomas formais mas, ao contrário, estes devem ser descritos em linguagem natural, refletindo simplesmente as restrições existentes sobre o universo de discurso. Os axiomas em uma ontologia podem apresentar duas formas e propósitos diferentes: *axiomas de derivação* e *axiomas de consolidação*. Axiomas de derivação são aqueles que permitem explicitar informações a partir do conhecimento previamente existente. Assim, são meios para a dedução e representam conseqüências lógicas neste processo. Axiomas de consolidação, por sua vez, não são utilizados para derivar informação, mas apenas para descrever a coerência das informações existentes. Neste sentido, não representam conseqüências lógicas. Tipicamente, os axiomas de consolidação definem condicionantes para o estabelecimento de uma relação ou para a definição de um objeto como instância de um conceito (Falbo, 1998).

Os axiomas de derivação podem ter origem no significado dos conceitos e relações da ontologia ou na forma como são estruturados. Quando axiomas são descritos para mostrar restrições impostas pela forma de estruturação dos conceitos, eles são ditos axiomas epistemológicos. Quando descrevem restrições de significação impostas no domínio, são ditos axiomas ontológicos (Falbo, 1998).

Esta classificação quanto à natureza dos axiomas é uma boa diretriz para guiar a definição dos axiomas de uma ontologia, ou seja, devemos estar atentos para capturar axiomas que considerem a estruturação dos conceitos e relações (os axiomas epistemológicos), seus significados e restrições (os axiomas ontológicos) e as leis de integridade que os regem (os axiomas de consolidação) (Falbo, 1998).

O processo de definição de axiomas é, talvez, o aspecto mais difícil na construção de ontologias. Entretanto, esse processo pode e deve ser fortemente guiado pelas questões de competência. Os axiomas devem ser necessários e suficientes para expressar as questões de competência e para caracterizar suas soluções. Além disso, qualquer solução para uma questão de competência deve ser descrita pelos axiomas da ontologia e deve ser consistente com eles. Se os axiomas propostos não forem suficientes para esse propósito, então conceitos, relações ou axiomas adicionais devem ser introduzidos na ontologia. Por outro lado, axiomas redundantes ou que não contribuem para responder a

uma questão de competência devem ser eliminados (Falbo, 1998). Neste sentido, a captura de uma ontologia é um processo iterativo e fortemente ligado à avaliação (Uschold & Gruninger, 1996).

c) Formalização da Ontologia

Para a realização desta etapa, é necessário que um formalismo de representação das diversas categorias de conhecimento da ontologia seja escolhido. À primeira vista, qualquer linguagem de representação formal do conhecimento, ou mesmo informal, poderia ser usada para representar ontologias (Falbo, 1998). Na prática, entretanto, apenas poucas linguagens têm sido usadas para este propósito, entre elas: *lógica de primeira ordem*, *KIF* (*Knowledge Interchange Format*) (Gruber, 1992), *Ontolingua* (Gruber, 1995), *CML (Conceptual Modelling Language*) (Breuker & Van de Velde, 1994) e *Description Logic* (Russel & Norvig, 1995).

A validação de uma teoria sobre um universo de discurso é, sem dúvida, melhor realizada quando esta é descrita em uma linguagem formal, ou seja, uma linguagem fundamentada em um modelo matemático. Nesta linguagem, em contraste com a linguagem natural, tem-se símbolos não ambíguos e formulações exatas e, portanto, a clareza e a correção de uma dedução podem ser testadas com maior facilidade e precisão. Uma dedução em linguagem natural, geralmente, envolve pressuposições implícitas que entram desapercebidas no processo de dedução. O tratamento teórico de qualquer domínio consiste em propor sentenças sobre os objetos neste domínio (sentenças atribuindo certas propriedades e relações aos objetos em questão) e em estabelecer regras de acordo com as quais outras sentenças possam ser derivadas a partir das sentenças dadas.

É importanate ressaltar que todas essas linguagens possuem vantagens específicas e assumem compromissos ontológicos em níveis variados, e portanto a escolha de que linguagem usar depende diretamente do propósito da ontologia.

d) Integração com Ontologias Existentes

Durante os processos de captura e/ou formalização, pode surgir a necessidade de integrar a ontologia em questão com outras já existentes, visando aproveitar conceituações previamente estabelecidas. De fato, é uma boa prática desenvolver

ontologias funcionais modulares, que sejam gerais e mais amplamente reutilizáveis, e, quando necessário, integrá-las, obtendo o resultado desejado (Falbo, 1998).

e) Avaliação

Finalmente, a ontologia deve ser avaliada para verificar se satisfaz os requisitos estabelecidos na especificação. Esta etapa deve ser realizada em paralelo com as etapas de captura e formalização. Gruber (1995) apresenta um conjunto de critérios para guiar tanto o desenvolvimento, quanto para avaliação da qualidade das ontologias construídas. Os principais critérios definidos são: clareza, coerência, extensibilidade e compromissos ontológicos mínimos. Em (Uschold & Gruninger, 1996), é defendido que, adicionalmente, as questões de competência devem ser usadas principalmente para avaliar a adequação da axiomatização realizada.

f) Documentação

Todo o desenvolvimento da ontologia deve ser documentado, incluindo propósitos, requisitos e cenários de motivação, as descrições textuais da conceituação, a ontologia formal e os critérios de projeto adotados. Como foi dito anteriormente, assim como a avaliação, a documentação é considerada uma atividade guarda-chuva do processo, ou seja uma etapa que deve ocorrer durante todas as iterações do ciclo em paralelo com as demais. Os termos capturados na conceituação do universo de discurso devem ser descritos em um Dicionário de Termos, considerando dois princípios importantes: o princípio do vocabulário mínimo e o princípio da auto-referência. O princípio do vocabulário mínimo diz respeito ao vocabulário utilizado na definição dos termos da ontologia. Este vocabulário deve ser o menor possível e não deve apresentar ambigüidades. Qualquer termo que não tenha um significado claro e não ambíguo, deve ser definido como uma entrada no Dicionário. O princípio da auto-referência indica que a definição de um termo no Dicionário deve, sempre que possível, ser feita utilizando outros termos do Dicionário. Com base neste princípio, o uso de hipertextos surge como uma potencial abordagem para a documentação de ontologias. Esta tecnologia mostra-se adequada, tendo em vista que torna natural a definição de novos termos a partir de outros mais primitivos, permitindo navegação entre definições, exemplos e a formalização, incluindo os axiomas (Falbo, 1998).

3.4 - Análise de Domínio e Ontologias

Na seção 3.2.1, foi descrito o objetivo de um modelo de domínio do ponto de vista da análise de domínio. Segundo a definição dada, este objetivo é definir entidades, operações, eventos e relações que abstraem similaridades e regularidades em um determinado domínio, criando modelos que tornam possível identificar, explicar e prever fatos difíceis de serem observados diretamente. Depois de pronto, este modelo deve servir como uma fonte unificada de referência quando ambigüidades surgirem em discussões sobre este domínio, além de como um repositório de conhecimento comum, auxiliando de forma direta a comunicação, o aprendizado e reuso em um nível mais alto de abstração. Como pode ser observado, esta definição está de acordo e pode ser totalmente atendida pelas ontologias de domínio da maneira que foram descritas nesse capítulo.

As áreas de construção de ontologias e análise de domínio apresentam inúmeras similaridades que vão desde a concordância do que pode ser definido como domínio (*conjunto de itens de informação presentes em um certo contexto do mundo real, interrelacionados de forma bastante coesa*) até a definição do seu objetivo final. No entanto, os modelos conceituais gerados pelos métodos de análise de domínio estudados (e citados na seção 3.2.2), como por exemplo, diagramas de entidades e relacionamentos e modelo de objetos, são pobres para a representação de conhecimento, estabelecendo apenas significados particulares de estruturação para os conceitos envolvidos. Ontologias, por outro lado, pelo fato de atuarem em um nível mais alto de abstração, e por representarem de maneira explícita o conhecimento da conceituação empregada, resolvem esta limitação de maneira bastante satisfatória.

Diante dessa convergência de propósitos e dos benefícios introduzidos, faz sentido pensar em um modelo de processo para uma análise de domínio orientada a ontologias. Nesse modelo, as ontologias de domínio têm um papel central, norteando todo o processo, não só atendendo a todos requisitos próprios da fase em questão como também apresentando importantes contribuições.

A seguir as fases definidas para ambas as disciplinas são associadas para fazer surgir um novo processo comum. Neste processo as atividades de documentação e

avaliação da qualidade continuam tendo a mesma importância e devem continuar aparecendo como atividades guarda-chuva em um ciclo iterativo de desenvolvimento.

a) Planejamento

A fase de planejamento continua sendo altamente centrada no contexto da organização em que o desenvolvimento com/para reuso vá ocorrer. As atividades de análise do negócio e análise de risco continuam sendo realizadas, inclusive avaliando questões como complexidade e conhecimento disponível acerca do domínio a ser representado. As questões específicas da organização são fundamentais na definição dos cenários de motivação, dos propósitos e do uso esperado da ontologia. As atividades relativas à definição do contexto do problema como a definição do que é relevante para ser representado e a escolha da abordagem a ser utilizada guiarão, respectivamente, o processo de elaboração das questões de competência e da escolha do formalismo empregado. A fase culmina em uma especificação de requisitos da ontologia.

b) Aquisição de dados

No modelo de construção de ontologias proposto na seção 3.3.4, a etapa de aquisição de dados é apresentada como uma sub-etapa da captura da ontologia. Apesar disso, as atividades realizadas nessa etapa são praticamente idênticas tanto na análise de domínio quanto na construção de ontologias. Desse modo esta fase permanece inalterada.

c) Construção do modelo do domínio

Esta etapa continua sendo a mais importante de todo o processo. Agrupando agora as sub-atividades de captura e formalização da ontologia descritas na seção 3.3.4. Em ambas as disciplinas, o objetivo dessa fase é construir um modelo do domínio baseado no conhecimento capturado na fase anterior. Ambas propõem também que o modelo produzido seja composto de um léxico (vocabulário, dicionário), descrevendo os itens selecionados para representar aquele domínio, junto com suas definições, além de contemplar um modelo conceitual que apresenta uma taxonomia e mostra as diversas relações envolvendo esses itens. O modelo conceitual gerado na construções de ontologias, composto de conceitos e relações, geralmente é situado em um nível mais alto de abstração.

Na construção de ontologias, além do modelo conceitual e do vocabulário de termos, o modelo de domínio é composto de um conjunto de axiomas. Esses axiomas representam de maneira formal o conhecimento do domínio, permitindo a derivação de novos conhecimentos a partir dos já formalizados, a verificação da consistência e validação automática desses conhecimentos, a restrição da interpretação da semântica de conceitos e relações e, principalmente, a possibilidade de geração automática de infra-estruturas computacionais reutilizáveis a partir do modelo de domínio (como será mostrado no próximo capítulo). Uma vez especificados, os axiomas formais devem ser formalizados em uma linguagem capaz de servir aos propósitos da ontologia e/ou da atividade de engenharia de domínio e, ao mesmo tempo, incorporar a menor quantidade possível de compromissos ontológicos adicionais.

Apesar de todas as vantagens citadas, a principal contribuição dos axiomas formais da ontologia ainda é a possibilidade de representação explícita do conhecimento do domínio em um nível de significação (nível ontológico) ao contrário do nível de estruturação (epistemológico), que é o máximo alcançado quando se tem apenas os outros dois modelos citados. A figura 3.2 é usada, a seguir, para exemplificar essa diferença. Nela, é apresentado um modelo de entidades e relacionamentos, mostrando uma pequena parte de um sistema acadêmico para uma universidade (Falbo, 1998).

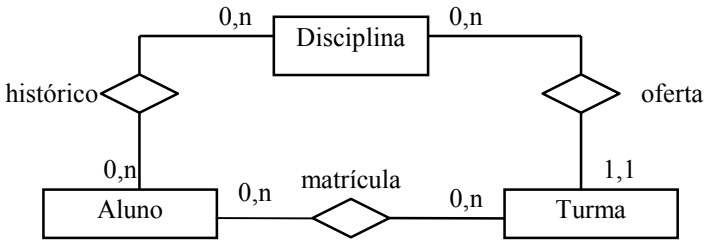


Figura 3.2 - Extrato de um Modelo ER para um Sistema Acadêmico.

Essa representação de nível epistemológico mostra os conceitos aluno, disciplina e turma, e os relacionamentos de matrícula, histórico e oferta entre instâncias desses conceitos. Pode-se perceber que, neste nível, só é possível representar a estrutura das coisas no mundo. Entretanto, algumas questões importantes permanecem em aberto, entre elas: Que elementos do conjunto de entidades *aluno* podem estar relacionados com uma

instância específica de *turma*? O que é exatamente uma instância de *turma*? Que subconjunto do produto cartesiano das instâncias de *aluno* e *disciplina* caracteriza precisamente o relacionamento *histórico*? Tais questões não são respondidas por esse modelo, já que ele é um modelo do nível epistemológico. A solução normalmente adotada nestes casos consiste em embutir este conhecimento dentro do código de um programa, o que dificulta o seu reuso e compartilhamento. Construindo-se ontologias formais, é possível estabelecer significados formais para alguns termos do vocabulário do domínio, assim como pode-se restringir a interpretação da teoria com base na axiomatização da ontologia.

No caso do sistema acadêmico, axiomas da ontologia poderiam, por exemplo, impor condicionantes para o estabelecimento do relacionamento *matrícula*, tal como: se *a* é um aluno e existe um relacionamento *histórico(a,d)*, indicando que este aluno já cursou a disciplina *d* tendo sido aprovado, então não é possível haver o relacionamento *matrícula(a,t)*, se a turma *t* for uma oferta da disciplina *d*, (isto é *oferta(t,d)*) já cursada pelo aluno. A seguinte sentença, em lógica de primeira ordem, é a formalização deste axioma:

$$(\forall\, a,d,t)\ (aluno(a) \wedge disciplina(d) \wedge turma(t) \wedge$$
$$histórico(a,d,Aprovado) \wedge oferta(t,d) \rightarrow \neg\, matrícula(a,t))$$

d) Integração

Essa é outra etapa que se mantém praticamente inalterada de acordo com sua definição no processo de construção de ontologias. O método de McCain (Arango, 1994) para análise de domínio dá uma forte ênfase na idéia de que, uma vez produzido, o modelo do domínio deve passar por uma fase chamada *análise de domínio orientada a componentes,* na qual sofrerá algumas modificações a fim de maximizar sua facilidade de integração e minimizar o seu custo de adaptação[4]. Esta prática favorece diretamente a estensibilidade da ontologia, que é um dos principais requisitos de qualidade apontados por Gruber (1995).

Como citado anteriormente, essas atividades devem ser agrupadas em um ciclo fortemente iterativo, do modo ilustrado na figura 3.3. Esse ciclo é uma adaptação do ciclo

[4] *customization*

proposto por Falbo et al. (1998) para construção de ontologias, a fim de contemplar as atividades deste processo de análise de domínio orientada a ontologias.

A etapa de captura pode apontar novos requisitos ainda não identificados. Na avaliação, pode-se perceber que os termos descritos são insuficientes para o propósito planejado, impondo um retorno à etapa de captura. Situações semelhantes podem ocorrer na etapa de formalização: incoerências podem ser detectadas, provocando uma revisão das especificações e dos termos definidos na ontologia. Finalmente, quando for necessário integrar uma ontologia com outras existentes, este processo pode ter substancial impacto na definição e formalização dos termos (Falbo, 1998).

As etapas do processo de desenvolvimento de uma ontologia e suas interdependências são ilustradas pela figura 3.3. As linhas tracejadas indicam que há uma interação constante, embora mais fraca, entre as etapas associadas. As linhas cheias mostram o fluxo principal de trabalho no processo de construção de uma ontologia. A linha envolvendo as etapas de captura e formalização da ontologia realça a forte interação e, por conseguinte iteração, que ocorre entre essas etapas (Falbo, 1998).

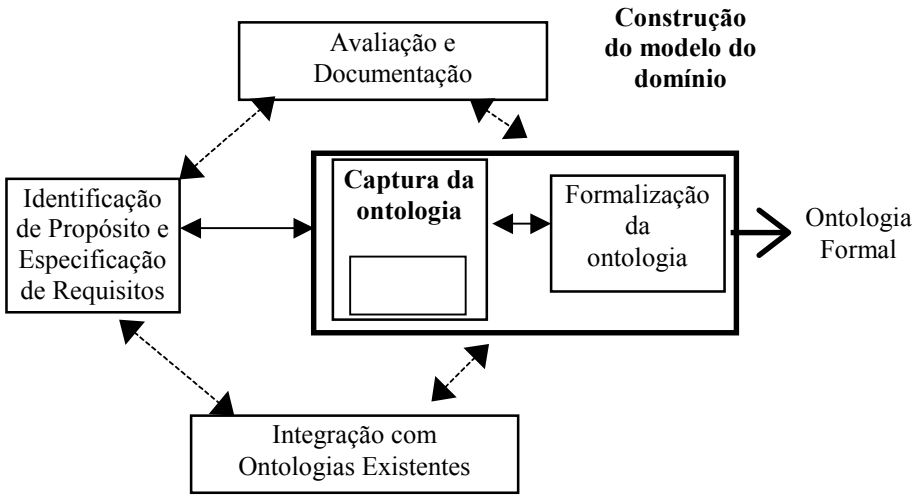


Figura 3.3 - Atividades do processo proposto para Análise de Domínio orientada a ontologias

3.5 - Conclusões

Neste capítulo, foi mostrada a importância da atividade de análise de domínio no contexto de um processo de desenvolvimento *para* e com *reuso*. Ao realizar a análise do

domínio e a exploração do conhecimento envolvido na atual aplicação (e em aplicações similares), os modelos gerados tendem a ser mais estáveis, as mudanças a serem isoladas e as aplicações a se tornarem conseqüentemente mais robustas. A arquitetura produzida pode endereçar tanto requisitos funcionais quanto não funcionais ao explorar familiaridades de arquiteturas de sistemas semelhantes que foram capazes de produzir soluções com as características desejadas. Além disso, o modelo produzido é importante por auxiliar a comunicação propriamente dita acerca do domínio em questão, por guiar o processo de obtenção de consenso sobre os termos e relações que melhor representam aquele domínio e por viabilizar o entendimento e o raciocínio de forma organizada sobre o que é feito e por que.

No que diz respeito ao reuso, o grande benefício introduzido por essa atividade, se dá através da representação de estruturas de conhecimento, permitindo a reutilização de artefatos mais abstratos que somente código. No entanto, ao serem estudados os métodos mais difundidos, foi observado que sua principal limitação está também ligada à capacidade dos modelos conceituais empregados serem capazes de atuar somente em um nível de estruturação de conhecimento (nível epistemológico), o que é insuficiente para representar de forma explícita, o conhecimento envolvido na restrição da interpretação do significado de conceitos e relações, na especificação de condições a serem cumpridas para que relações possam ser estabelecidas e, principalmente, na derivação de conhecimento a partir do conhecimento factual representado.

Para resolver esta limitação, ontologias de domínio são apresentadas como uma alternativa promissora, pois além de seus modelos epistemológicos de estruturação de conceitos e relações, possui um conjunto de axiomas formais que, em forma de uma teoria lógica, permitem a representação formal em nível de significação (nível ontológico). Como conseqüência do formalismo empregado, muitos são os benefícios alcançados, como por exemplo, a verificação/validação automática do modelo de conhecimento construído, a interpretação não ambígua das definições de conceitos e relações e a possibilidade de geração sistemática (e idealmente automática) de infra-estruturas de domínio, como será mostrada no próximo capítulo.

Por fim, o capítulo apresenta um modelo de processo para a realização de análise domínio, mostrando atividades (planejamento, aquisição de dados, construção do modelo

e integração), critérios claros para avaliação da qualidade (clareza, coerência, extensibilidade, compromissos ontológicos e de codificação mínimos) e um modelo iterativo de ciclo de vida. O próximo capítulo completa essa definição discutindo a escolha da linguagem a ser usada para a representação dos elementos que compõem a ontologia. Além disso, o capítulo aborda a transição dessa fase para a de *Projeto de Domínio*, propondo um método de geração de *frameworks* orientados a objetos a partir de ontologias de domínio.

Capítulo 4

Formalização de Ontologias e Projeto de Domínio

pensamentos abstratos demandam
linguagens abstratas

Aristophanes

4.1 - Introdução

Como tem sido mencionado ao longo deste trabalho, seu objetivo é introdução de forma sistemática nos processos da engenharia de software, de atividades que propiciem um desenvolvimento formal, altamente orientado à prática de reutilização em todos os níveis. Para isso, no Capítulo 2, é proposto um modelo genérico que associa de forma complementar à engenharia de software, um processo de engenharia de domínio. A engenharia de domínio é composta pelas seguintes macro-atividades: análise de domínio e projeto de domínio.

As principais vantagens introduzidas pela atividade de análise de domínio estão ligadas à comunicação e ao reuso. Primeiro, por auxiliar a comunicação propriamente dita acerca do domínio em questão, por guiar o processo de obtenção de consenso sobre os termos e relações que melhor representam aquele domínio e por viabilizar o entendimento e o raciocínio de forma organizada sobre o que é feito e porque. No que diz respeito ao reuso, o grande benefício se dá através da representação de estruturas de conhecimento, permitindo a reutilização de artefatos mais abstratos que somente código.

No capítulo anterior, foi. apresentada uma abordagem para realização da fase de análise de domínio, através da utilização de ontologias formais de domínio. Com o término dessa fase, a ontologia produzida deve dar origem uma infra-estrutura computacional, passível de posterior reutilização por um processo de engenharia de software. Dessa forma, na fase de projeto de domínio, a ênfase é a construção dessa infra-estrutura de reuso e duas atividades principais merecem destaque: especificação da infra-estrutura e projeto/implementação (Arango & Prieto-Díaz, 1994).

A perspectiva de especificação da atividade de especificação diz respeito à escolha de quais elementos da ontologia (inclusive axiomas) devem ser considerados na infra-estrutura computacional, visto que existem elementos ligados unicamente a funções de comunicação e entendimento a respeito do domínio. Essa perspectiva analisa também como será feito o processo de tradução, considerando o paradigma utilizado para a implementação da infra-estrutura.

Neste trabalho, é discutido o processo de tradução para o paradigma de objetos, visando a construção de um *framework*. Como um *framework* é também um artefato de código, uma vez obtida sua estrutura inicial, a fase de projeto e implementação tem por objetivo fazer com que uma estrutura mais estável seja alcançada e que o seu potencial de reutilização seja maximizado. Para isso são realizados processos de generalização, especialização e recomposição, além de integração com padrões de projeto.

Este capítulo completa o processo de sistematização da engenharia de domínio iniciado no Capítulo 3. Na seção 4.2, é mostrada a abordagem utilizada para representação de ontologias, composta de uma linguagem gráfica para estruturação de conceitos e relações (LINGO) e uma linguagem formal para a definição de axiomas. Além disso, a seção apresenta toda a fundamentação teórica da linguagem e discute, em um nível ontológico, distinções importantes para a escolhe de como algumas entidades do domínio devem ser modeladas. Na seção 4.3, são apresentadas diretivas de mapeamento da estrutura da ontologia (conceitos, relações e axiomas epistemológicos) em elementos do *framework* (classes, relacionamentos e padrões de projeto), bem com um conjunto de regras de transformação para geração automática de invariantes a partir de axiomas ontológicos. Por fim, a seção 4.4, apresenta um método de geração de padrões de projeto

que asseguram pré-condições derivadas dos axiomas de consolidação. A seção 4.5 finaliza o capítulo, apresentado as conclusões obtidas.

4.2 – O Modelo de Representação de Ontologias

Conforme mencionado no capítulo anterior, a escolha feita neste trabalho foi a de usar uma abordagem composta na definição do modelo de representação da ontologia. Essa abordagem utiliza-se de duas notações: uma gráfica e uma textual. A linguagem resultante permite ao desenvolvedor formalizar, descrever, representar e raciocinar sobre o universo de discurso, sem perder a clareza e a intuitividade necessárias à comunicação entre especialistas do domínio e engenheiros de software.

O uso de uma linguagem gráfica é reconhecido como de extrema importância, pois age como um elemento facilitador da comunicação entre os agentes envolvidas no processo. Em uma primeira análise, existem várias linguagens gráficas passíveis de utilização para este propósito, por exemplo, diagramas de classes e modelos de entidades-relacionamentos (E-R). Apesar disso, em (Falbo, 1998) é definida uma notação gráfica chamada LINGO (Linguagem Gráfica para descrever Ontologias). Segundo os autores, a necessidade de criação de uma nova linguagem é devida à necessidade de minimizar a incorporação de compromissos ontológicos, eliminando da descrição da ontologia a semântica embutida nas linguagens citadas.

Em um modelo de objetos, objetos representam abstrações do mundo real que possuem estado (dado por seus atributos), comportamento (dado por suas operações) e identidade própria. Classes, por sua vez, agrupam objetos que possuem os mesmos atributos e relacionamentos e exibem o mesmo comportamento. Em se tratando de uma ontologia, o propósito de uma notação gráfica é modelar uma conceituação - conceitos extraídos do domínio (universo de discurso) e as relações entre eles. Poder-se-ia argumentar que conceitos poderiam ser descritos como classes em um modelo de objetos, mas há diferenças fundamentais: objetos em uma classe exibem um comportamento dado pelas operações da classe; para conceitos em uma ontologia, isto não faz sentido; geralmente, classes em um modelo orientado a objetos possuem atributos[5], em

[5] Alguns métodos orientados a objetos, como o proposto por COAD et al. (Coad & Yourdon, 1992), utilizam como critério para inclusão de classes em um modelo, o fato da classe possuir mais do que um atributo.

ontologias, ainda que conceitos possam apresentar atributos, esta não é uma característica obrigatória. Este aspecto é uma razão também para não adotarmos um modelo de entidades e relacionamentos para descrever ontologias, ainda que alguns métodos, como o proposto no âmbito do Projeto TOVE, o façam (Falbo, 1998).

A linguagem LINGO não é formal, mas possui uma meta-ontologia, e a semântica de suas notações pode ser diretamente mapeada em um conjunto equivalente de axiomas. Ao descrever graficamente os elementos da linguagem, está sendo descrito o conjunto de axiomas que eles representam.

Este trabalho segue a mesma abordagem, adotando LINGO como a linguagem gráfica de representação. No entanto, a linguagem textual na qual os axiomas correspondentes são descritos não é a mesma usada em (Falbo, 1998). Ao invés de lógica de primeira ordem, é adotada uma notação híbrida, adicionando uma forte base de teoria dos conjuntos.

A lógica de primeira é o formalismo que inclui a menor quantidade de compromissos ontológicos adicionais. Quando os elementos da conceituação (conceitos e relações) são mapeados para essa linguagem, eles aparecerão como predicados que não embutem nenhuma semântica adicional à já descrita em suas definições. No entanto, devido ao objetivo desse trabalho, é necessário que o formalismo adotado esteja situado em um nível intermediário de abstração entre a lógica pura e a orientação a objetos, desempenhando o mesmo papel da lógica na axiomatização dos elementos e servindo de modelo intermediário na transformação para *frameworks*.

Finalmente, a decisão de não utilizar outro formalismo baseado em teoria dos conjuntos – por exemplo, Z (Spivey, 1988) – é novamente devido à necessidade de minimização dos compromissos ontológicos embutidos, além de prover uma linguagem mais clara e amigável que facilite a compreensão e comunicação no processo de interação entre os agentes.

A seguir, é apresentada a fundamentação teórica desse modelo, enquanto que as subseções seguintes demonstram como cada uma das primitivas de modelagem de LINGO é mapeada no modelo formal. É importante ser mencionado que na apresentação dessa fundamentação, o rigor matemático e uma extensiva definição axiomática são algumas vezes sacrificados em prol da legibilidade e da clareza de comunicação do

conteúdo. Desse modo, nos permitimos alguns exageros na semântica de alguns termos, como por exemplo, no caso de algumas expressões lógicas que chamamos de axiomas.

4.2.1 – Fundamentação Teórica

Conjuntos podem ser entendidos como coleções de zero ou mais elementos. Os elementos contidos em um conjunto são únicos e a ordem em que aparecem é imaterial. Esses elementos (também conhecidos como instâncias) serão aqui referenciados por letras minúsculas, enquanto que os conjuntos por letras maiúsculas.

Conjuntos podem possuir um número finito ou infinito de elementos. Quando esse número é finito e pequeno, geralmente eles são representados através da enumeração dos seus elementos (ex. C = {x,y,z}). Caso contrário, são representados através de regras de formação (ex. B = { n : IN | n > 3}), ou seja, no caso geral, um conjunto é uma união de elementos que compartilham características comuns.

A tabela 4.1 apresenta os principais operadores da simbologia usada para representar as operações da teoria de conjunto.

Símbolo	**Nome**	**Interpretação**
$\in$	Pertinência	x $\in$ **C:** É verdadeira se o elemento x pertence ao conjunto C
$\notin$	Não-pertinência	x $\notin$ **C:** É verdadeira se o elemento x não pertence ao conjunto C
$\subseteq$	Subconjunto	A $\subseteq$ B**:** Operador aplicado a conjuntos. A expressão é verdadeira se todos os elementos de A são também elementos de B. Nesse caso diz-se que A está contido em B
$\subset$	Subconjunto próprio	A $\subset$ B**:** Operador aplicado a conjuntos. A expressão é verdadeira se todos os elementos de A são também elementos de B, mas existe pelo menos um elemento em B que não seja elemento de A. Em outras palavras A está contido em B, mas não é igual a B
=	Igualdade	A = B: Um conjunto A é igual a outro conjunto B se e somente se A $\subset$ B e B $\subset$ A
$\not\subset$	Não-está-	A $\not\subset$ B**:** Operador aplicado a conjuntos. A expressão é

	contido	verdadeira se existe pelo menos um elemento de A que não seja elemento de B
$\cup$	União	A $\cup$ B: O operador é aplicado a dois conjuntos, formando como resultado um novo conjunto contendo todos os elementos dos dois primeiros
$\cap$	Interseção	A $\cap$ B: O operador é aplicado a dois conjuntos, formando como resultado um novo conjunto contendo apenas os elementos que pertencem a ambos os conjuntos
#	Cardinalidade	#C: O operador é aplicado a um conjunto, retornado o número de elementos do conjunto
\	Diferença	A \ B: O operador é aplicado a dois conjuntos e forma um conjunto a partir da remoção de A dos elementos pertencentes a B
$\times$	Produto Cartesiano	A $\times$ B: O operador é aplicado a dois conjuntos, formando um conjunto de todos os possíveis pares (a,b), sendo que a $\in$ A e b $\in$ B
$\wp$	Conjunto potência	$\wp$(A): O operador é aplicado a um conjunto, formando todos os seus possíveis subconjuntos. Um conjunto de conjuntos é geralmente chamado de uma família de conjuntos
$\varnothing$	Conjunto vazio	Um tipo especial de conjunto com zero elementos, que possui a propriedade de ser subconjunto de qualquer outro conjunto. Duas identidades úteis envolvendo o conjunto vazio são: (a) $\varnothing \cup A = A$; **(b)** $\varnothing \cap A = \varnothing$**;**

Tabela 4.1 – Sumário da Simbologia de Teoria dos Conjuntos

Além da simbologia apresentada, este trabalho conta com os operadores lógicos de conjunção ($\wedge$), disjunção ($\vee$), disjunção exclusiva ($\oplus$), negação ($\sim$), condicional ($\rightarrow$) e bidicondicional ($\leftrightarrow$), além dos quantificadores Universal ($\forall$), Existencial ($\exists$) e um

variação do quantificador Existencial simples **(∃!)** cuja semântica denota a existência de um e somente um elemento.

4.2.1.1 - Relações

Uma relação *n*-ária pode ser definida pela *n*-tupla $R = (C_1,C_2...C_n, p(x_1,x_2...x_n))$, sendo C_i cada um dos conjuntos envolvidos na relação e $p(x_i)$ a função proposicional aberta em *n* variáveis que assume um valor V (verdadeiro) ou F (falso) para cada elemento pertencente ao produto cartesiano $C_1 \times C_2 \times ... C_n$.

Toda relação R define um conjunto R^* chamado de conjunto solução de R, que contém todos os elementos e_i pertencentes a $C_1 \times C_2 \times ... C_n$ para os quais $p(e_i)$ é verdadeira, ou seja, $R^* = \{(x_i,x_j...,x_n) \mid x_i \in C_1, x_j \in C_2, x_n \in C_n , p(x_i,x_j,...,x_n) = V\}$.

Devido ao propósito desse trabalho - que é utilizar este formalismo para modelar conceitos e relações de uma ontologia – o interesse é altamente direcionado para relações binárias, que são as mais freqüentes nesse cenário. Especializando a forma acima para relações binárias, temos que $R_2 = (A,B, p(x,y))$, sendo A e B dois conjuntos e p(x,y) uma função proposicional aberta em duas variáveis que tem um valor lógico para todos os pares (a,b) pertencentes a $A \times B$. O conjunto R_2*, nesse caso, será $\{(a,b) \mid a \in A, b \in B, p(a,b) = V\}$.

Um exemplo de uma relação deste tipo seria R = {Homens, Mulheres, relacionamento(x,y)}, entre os conjuntos de *Homens* e *Mulheres* de um determinado universo, e que define uma função proposicional *relacionamento,* que assume um determinado valor verdadeiro ou falso para cada par ordenado $(h,m) \in \textbf{\textit{Homens}} \times \textbf{\textit{Mulheres}}$. Para cada relação R, existem dois subconjuntos D e E que são chamados respectivamente de *Domínio* e *Imagem* (ou Amplitude) de uma relação e podem ser definidos por: $D = \{a \mid a \in A, (a,b) \in R^*\}$ e $E = \{b \mid b \in B, (a,b) \in R^*\}$.

De maneira geral, é comum encontrar na literatura a expressão p(x,y) sendo referenciada como uma relação. Isso é devido ao fato de que é considerado implicitamente que as variáveis x e y, têm amplitude respectivamente sobre alguns conjuntos A e B, isto é, que p(x,y) é uma função de proposições definidas em algum conjunto produto de $A \times B$ (Lipschutz, 1974). Essa será a abordagem adotada nesse trabalho.

Por fim, um tipo especial de relação que merece uma atenção nesse trabalho é o das relações transitivas. Uma relação é transitiva se obedece a seguinte regra:

$$\forall a,b,c\ ((a,b) \in R^*) \wedge ((b,c) \in R^*) \rightarrow ((a,c) \in R^*)$$

4.2.1.2 – Funções

Seja f uma relação de A em B, f é uma função de A em B se e somente se, para todo elemento a de A existe um e somente um elemento b de B com o qual a se relaciona, ou de maneira mais formal:

$$\mathbf{\forall a{:}A \rightarrow (\exists b{:}B\ (a,b) \in f^*) \wedge ((\forall\ c,d{:}B\ ((a,c) \in f^*) \wedge ((a,d) \in f^*) \leftrightarrow (c = d))}$$

A semântica de $a{:}A$ é a de $a \in A$. Essa será a simbologia adotada de aqui em diante.

Uma função é referenciada pela simbologia $f{:}A \rightarrow B$ - lê-se f é uma função de A em B[6]. O conjunto B é chamado de *contradomínio* da função. Além disso, dado um par ordenado $(a,b) \in f^*$, b é chamado de *imagem* de a pela função f e é referenciado pela simbologia $b = f(a)$.

Da mesma forma que para relações, existem algumas funções que merecem atenção especial. Quando nunca dois elementos de A possuem a mesma imagem, a função é chamada de *biunívuca*. Se a amplitude de uma função for o próprio contradomínio então a função é chamada de *sobrejetiva*. Uma função *bijetiva* é uma função que é simultaneamente biunívuca e sobrejetiva. Uma outra função importante é a função reversa ou adjunta. Por exemplo, seja $f(w) = \{a,b,c\}$ e $f(q) = \{b,d\}$, a função reversa ou adjunta de f, tem a forma t: $B \rightarrow \wp(A)$, ex. $t(b) = \{w,q\}$ e $t(d) = \{q\}$. É importante ressaltar a diferença da função reversa para a inversa, que tem a forma f^{-1}: $\wp(B) \rightarrow \wp(A)$, ex. $f^{-1}(\{a,b,c\}) = \{w\}$.

Uma função de suma importância no contexto desse trabalho é a função Imagem (Im), definida a seguir.

---

[6] No caso de conjuntos não numéricos, a função $f{:}A \rightarrow B$ é geralmente chamada de uma transformação de A em B (Lipschutz, 1974).

Def: Imagem (Im): Seja Φ o conjunto de todas as relações binárias R presentes no universo analisado. A função Im pode ser definida como: $Im:A \times \Phi \rightarrow \wp(B)$. A função recebe como argumentos um elemento $a \in A$ e uma relação R e retorna um elemento B´ pertencente ao conjunto potência de B. O conjunto B´ nesse caso contém todos os elementos de B com os quais *a* se relaciona segundo R, ou em outras palavras, o conjunto imagem de *a* no escopo da relação R.
Formalmente temos :

$$\forall a:A, R:\Phi, B':\wp(B)\ Im(a,R) = B' \leftrightarrow \forall\ b:B'\ (a,b) \in R^{*}$$

Por outro lado, para todo *a* relacionado com um elemento $b \in B'$, *a* também pertence à imagem da função adjunta Im(b,R), ou seja,

$$\forall a:A, R:\Phi, b:B'\ (b \in Im(a,R)) \leftrightarrow (a \in Im(b,R))$$

Usando o exemplo da relação citado anteriormente da relação relacionamento = (Homens, Mulheres, relacionamento(h,m)) temos,

$$Im(h_1, relacionamento) = \{ m_1, m_2, m_3 \}$$

Estendendo a definição da função Im para relações $R = \{(C_1,C_2,...,C_n, p(x_1,x_2,...,x_n)$, de aridade maior que dois, a forma geral da função passa a ser $Im:C_1 \times \Phi \rightarrow \wp(C_2 \times C_3 \ldots \times C_n)$.

Finalmente, é importante ressaltar que Im é uma função distributiva, ou seja

$Im(\{h_1,h_2\},relacionamento) = Im(h_1, relacionamento) \cup Im(h_2, relacionamento)$

O que faz, nesse caso, que a forma geral passe a ser, então, $Im:\wp(A) \times \Phi \rightarrow \wp(B)$.

4.2.1.3 – Conceitos

Em se tratando de ontologias, uma linguagem gráfica deve possuir primitivas básicas capazes de representar a conceituação de um domínio. Neste sentido, em sua forma mais simples, LINGO possui primitivas para representar apenas conceitos e relações, cujas notações estão mostradas na figura 4.1 (Falbo, 1998).

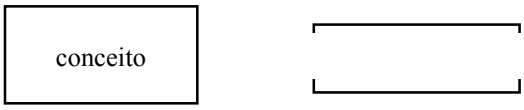


Figura 4.1 - Notações Utilizadas para Conceitos e Relações.

Um conceito pode ser definido como: *Uma idéia ou noção que aplicamos com intuito de classificar as coisas que nos rodeiam*. Em outras palavras, em (Langer, 1967), Susan Langer afirma que: "... estamos de forma consciente, deliberadamente abstraindo a forma de todas as coisas que a tem. Esta forma abstrata é chamada de conceito". Segundo Russel (1938), conceitos possuem dois aspectos importantes – *intenção* e *extensão*. A *intenção* de um conceito é seu significado, ou sua completa definição, enquanto que a *extensão* é o conjunto de coisas do universo analisado para o qual a intenção se aplica. Por exemplo, a intenção do conceito *mortal* poderia ser definida como "tudo que morre". A extensão nesse caso é o conjunto de elementos do universo que possuem a propriedade de morrer. É importante ressaltar que, apesar da simbologia usada ser a de teoria de conjuntos, ao se afirmar que $x \in C$, x está sendo classificado como sendo uma instância do conceito C e isso tem implicações em termos de intenção e extensão.

Na linguagem textual, os conceitos são mapeados em predicados unários. A função *C(x)* - sendo *C* o nome do conceito – mapeada para um valor verdadeiro indica o comprometimento do elemento *a* com o conceito *C* em termos de intenção e extensão. Portanto, como pode ser observado, o conjunto extensão geralmente não é fixo, mas definidos através de uma forma proposicional. Usando o exemplo do conceito *mortal*, a expressão *mortal(a)*, diz que a definição "tudo que morre" se aplica a *a*, ou seja, que *a* é uma instância do conceito *mortal* e que, por outro lado, *a* pertence ao conjunto *M* (extensão do conceito *mortal*).

4.2.1.4 – Relações

A figura 4.2 mostra um exemplo de uma relação binária entre os conceitos de Pessoa e Empresa, no escopo de uma organização, indicando que empresas contratam pessoas, e que por outro lado, pessoas são contratadas por empresas. O objetivo de

representar a relação como sendo um substantivo e não um verbo (como é usual no contexto de bancos de dados e mesmo de orientação a objetos), é mostrar que as relações de associação são sempre simétricas e, portanto bidirecionais. As relações entre conceitos são diretamente mapeadas em relações definidas na teoria de conjuntos. Nesse exemplo, temos contrato = ((Pessoa, Empresa, p(x,y)), e portanto contrato* seria o subconjunto de Pessoa × Empresa, para os quais p(x,y) seria verdadeira. Como citado anteriormente, a função proposicional contrato(x,y) será usada como sinônimo de relação.

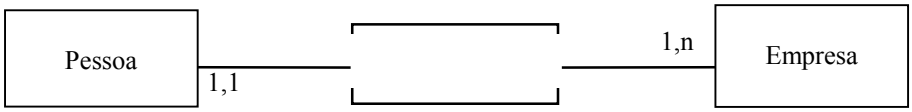


Figura 4.2 - Exemplo de uma Relação Binária entre Conceitos.

Como pode ser observado na figura 4.2, cardinalidades são usadas para mostrar quantas instâncias de um conceito podem participar da relação. De maneira geral, são quatro as possibilidades de cardinalidades quanto aos limites inferior e superior: (0..1), (0..n), (1,1) e (1..n). A cardinalidade (0..n) não impõe nenhum axioma e, portanto, não é representada graficamente (Falbo, 1998). O uso das cardinalidades de limite inferior igual a um, na figura 4.2, por exemplo, induz o seguinte axioma $\forall$ p Pessoa(p) $\rightarrow \exists$ e Empresa(e) $\wedge$ contrato(p,e), ou de forma mais simplificada:

$$\forall \text{ p:Pessoa} \rightarrow \exists \text{e:Empresa contrato(p,e)} \qquad \text{(A1)}$$

O uso da cardinalidade com limite superior igual a 1, por outro lado, impõe o axioma

$$(\forall \text{ p:Pessoa},e_1 ,e_2\text{: Empresa}) (\text{contrato}(p,e_1) \wedge \text{contrato}(p,e_2) \rightarrow e_1 = e_2) \qquad \text{(A2)}$$

As cardinalidades (1,1) e (1,n), portanto, podem ser representadas, respectivamente, pelos seguintes axiomas:

$$\forall \text{ p:Pessoa}, \exists! \text{e: Empresa contrato(p,e)} \qquad \text{(A3)}$$

ou de forma equivalente,

$$\forall \text{ p:Pessoa } \#\text{Im(p,contrato)} = 1$$

$$\forall \text{ e: Empresa } \#\text{Im(e,contrato)} \geq 1 \qquad \text{(A4)}$$

Como as relações nesse trabalho são sempre bidirecionais, a função Im pode ser usada tanto na forma Im:Pessoa $\rightarrow \wp$(Empresa) quanto na direção oposta Im:Empresa $\rightarrow \wp$(Pessoa), denotando a função adjunta correspondente.

Apesar do exemplo dado, não se deve pensar que as relações estão restritas a relações binárias. Relações de ordem superior, tais como relações ternárias, são igualmente válidas, e da mesma forma são diretamente mapeadas em relações entre conjuntos. A figura 4.3 mostra uma exemplo de relação ternária.

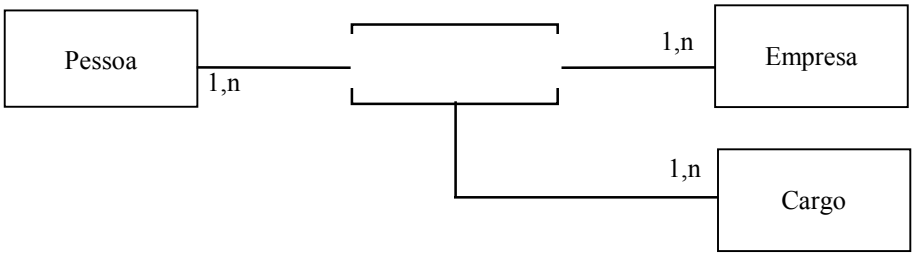


Figura 4.3 - Exemplo de uma Relação Ternária entre Conceitos.

A relação contrato nesse caso é definida por uma função proposicional aberta em três variáveis contrato(x,y,z). Do ponto de vista do formalismo empregado, ela também pode ser vista como uma relação (R_2) entre os conjuntos contrato* - subconjunto de Pessoa $\times$ Empresa, definido pela relação contrato (R_1) da figura 4.2 - e o conjunto Cargo. Dessa forma, a relação pode ser definida como uma recursão de relações do tipo $R_2(R_1(p,e),c)$, sendo *p*, *e* e *c* respectivamente instâncias de Pessoa, Empresa e Cargo. Além disso, a função Im, como mencionado na subseção 4.2.1.2, passa a ter a forma Im:X $\rightarrow \wp$(Y $\times$ Z). Como exemplo podemos ter:

$$\text{Im(Rosane,contrato)} = \{\text{(UFES,Professor),(UFES,Coordenador)}\}$$

Relações entre instâncias de um mesmo conceito também são válidas (figura 4.4). Em algumas relações (como essa), é importante o uso de papéis (*role*), cujo principal

objetivo é deixar claro as responsabilidades de cada uma das partes envolvidas na relação. A subseção 4.2.1.5 discute este tópico em maior profundidade.

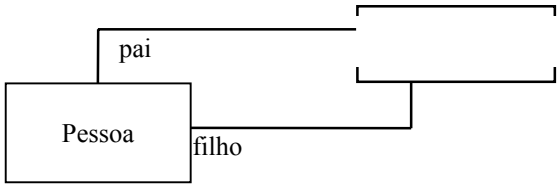


Figura 4.4 – Relação entre instâncias de um mesmo conceito

Neste caso, o seguinte axioma pode ser derivado:

$$(\forall\ p_1,p_2)\ (pai(p_1,p_2) \leftrightarrow filho(p_2,p_1)) \qquad (A5)$$

Outra facilidade de LINGO é a possibilidade de ser usada para expressar condicionantes entre relações. Tomemos o seguinte caso: dados três conceitos A, B e C e duas relações R_1 e R_2, entre instâncias dos conceitos A e instâncias dos conceitos B e C, respectivamente, queremos expressar uma condicionante entre as relações, dizendo que se uma instância de A está relacionada com uma instância de B, então ela não pode estar relacionada com uma instância de C. Para tal, a seguinte notação foi proposta (Falbo, 1998):

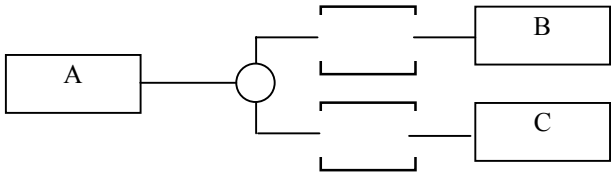


Figura 4.5 - Condicionante Ou-exclusivo entre Relações.

Ao ser utilizada a notação da figura 4.5, está sendo assumido que:

$$\forall\ a{:}A\ ((\exists\ b{:}B)\ \wedge R_1(a,b)) \rightarrow \sim ((\exists\ c{:}C) \wedge R_2(a,c))) \qquad (A6)$$

$$\forall\ a{:}A\ ((\exists\ c{:}C) \wedge R_2\ (a,c)) \rightarrow \sim ((\exists\ b{:}B) \wedge R_1\ (a,b))) \qquad (A7)$$

Analogamente, é introduzida a notação da figura 4.6, para estabelecer uma condicionante de obrigatoriedade entre relações: se uma instância de A está relacionada com uma instância de B, então ela obrigatoriamente tem de estar relacionada com uma instância de C (Falbo, 1998):

$$\forall\ a{:}A\ ((\exists\ b{:}B) \wedge R_1\ (a,b)) \rightarrow ((\exists\ c{:}C) \wedge R_2\ (a,c))) \qquad (A9)$$

$$\forall\ a{:}A\ ((\exists\ c{:}C) \wedge R_2\ (a,c)) \rightarrow ((\exists\ b{:}B) \wedge R_1\ (a,b))) \qquad (A10)$$

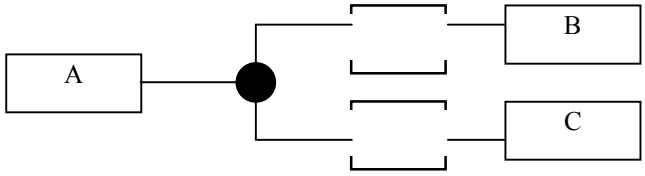


Figura 4.6 - Condicionante de Obrigatoriedade entre Relações.

Em (Falbo, 1998), os autores da linguagem reforçam que a proposta da linguagem não deve ser considerada definitiva. Ao contrário, LINGO deve ser considerada uma linguagem aberta, suscetível a extensões para capturar outras necessidades da modelagem no nível ontológico.

4.2.1.5 – Conceitos, Propriedades e Papéis

A figura 4.7 introduz na relação entre instâncias de Pessoa e Empresas dois novos elementos: propriedades e papéis. Em (Guarino, 1994), é apresentada uma proposta de organização do conhecimento em uma estrutura composta de cinco níveis: implementacional, simbólico, epistemológico, ontológico e lingüístico. Segundo Guarino, nos níveis simbólico e até mesmo epistemológico, é impossível distinguir as razões que levam a escolha de classificação de um elemento como conceito ou como propriedade de um conceito. No nível simbólico, ambos são implementados através de predicados unários, por exemplo, maçã(x) e vermelho(x). A primeira função proposicional responde se x é uma maçã ($x \in$ Maçã), e a segunda se x é vermelho ($x \in$ Vermelho). Do ponto de vista ontológico, no entanto, é necessário restringir a semântica que reflete o compromisso ontológico intrínseco à classificação de um determinado elemento do universo como conceito.

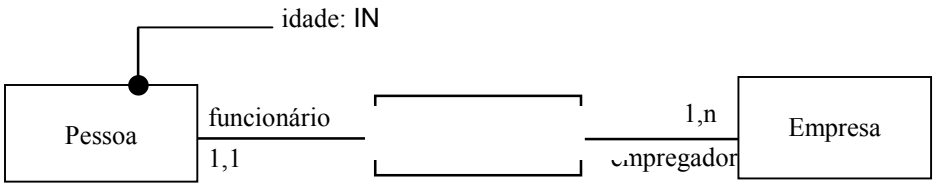


Figura 4.7 – Introdução de propriedades e papéis de conceitos

Uma primeira propriedade a ser analisada é se o elemento é *sortal* ou não. Um elemento sortal não pode ter a propriedade que o define retirada (*intenção*) sem que ele deixe de existir. A maçã é um exemplo deste tipo de elemento, já que para qualquer instância pertencente à extensão do conceito, se for retirada a *intenção* deste elemento, ele deixa de ser maçã e, portanto, deixa de existir. O mesmo não ocorre para vermelho. Seja o conjunto V de todas as coisas do universo para as quais o "conceito" vermelho se aplica, se retirarmos de um membro de V a propriedade de ser vermelho, ele continua existindo, e sendo outra coisa, como por exemplo, maçã.

O fato de ser sortal, porém, não garante que um elemento deva ser usado como conceito. Além disso, ele deve ser temporalmente neutro e ontologicamente rígido. Estas duas propriedades dizem, respectivamente, que: (i) a intenção deve se aplicar ao elemento em qualquer ponto do tempo, ou seja, o fato de ter esta definição não pode ter validade temporal; (ii) a intenção deve sempre se aplicar ao elemento dentro do domínio de interesse; Pessoa é um exemplo de elemento que possui ambas as propriedades, funcionário não. Uma mesma instância de Pessoa pode ser ora encarada como funcionário, ora como filho, ou como irmão, entre outras possíveis classificações. Uma pessoa pode, ainda deixar de ser funcionário e passar a ser Autônomo em um determinado ponto do tempo. Essas questões são de fundamental importância para que o objetivo de promover o reuso entre diversas aplicações dentro de um mesmo domínio seja alcançado.

A figura 4.8 ilustra a classificação de elementos representados por predicados unários, segundo as características citadas. Elementos não sortais são classificados como propriedades. Elementos sortais, mas cuja definição não é simultaneamente

ontologicamente rígida e temporalmente neutra, são classificados como papéis. Finalmente, elementos que possuem as três propriedades são conceitos.

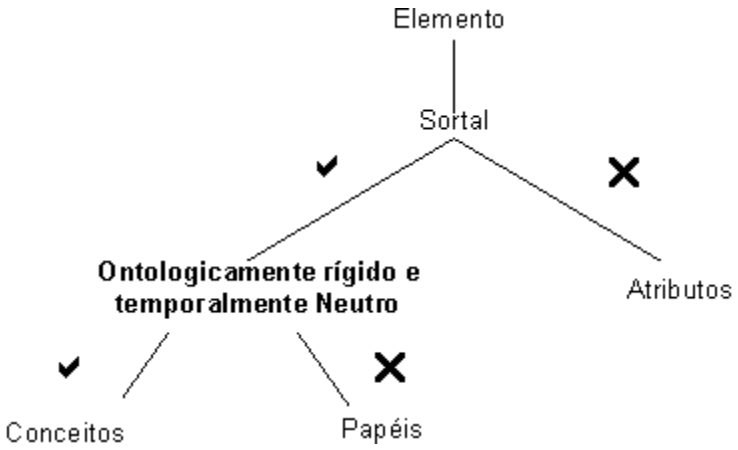


Figura 4.8 – Taxonomia de predicados unários em um nível ontológico

Desta maneira, no nível epistemológico, a propriedade idade (de Pessoa na figura 4.7) representa um relação ordinária idade(x,y) entre instâncias de Pessoa e instâncias de números naturais. Na verdade, como mostrado na figura 4.9, a relação acontece entre os conjuntos Pessoa e X, sendo X um subconjunto de IN, tal que todos os elementos de X têm a propriedade de representar a quantidade de anos de existência de ***algum outro elemento com o qual se relaciona***,

$$(X \subset IN) \wedge (\forall\, x{:}X, \exists y\ idade(y,x))$$

Como pode ser observada no grifo anterior, a própria definição da intenção do ora pretenso conceito X depende da existência de um outro conceito, deixando claro que do ponto de vista ontológico, uma instância de X e uma instância de Pessoa não podem estar no mesmo nível.

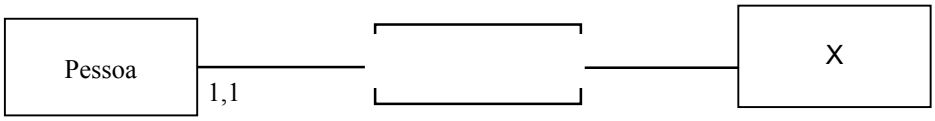


Figura 4.9 – Representação de uma propriedade como uma relação

No modelo formal, a variável que representa a instância de uma propriedade é sublinhada com o objetivo de salientar a distinção ontológica comentada anteriormente. Por exemplo,

$$\forall\ p{:}Pessoa,\ \exists!\underline{a{:}X}\ idade(\underline{a},p) \qquad (A11)$$

O axioma acima é derivado da cardinalidade (1..1) do relacionamento. Como propriedades podem ser obrigatórios ou não, e mono ou multivalorados, suas relações estão sujeitas às mesmas restrições de cardinalidades que as relações entre conceitos e, portanto, devem derivar os mesmos axiomas.

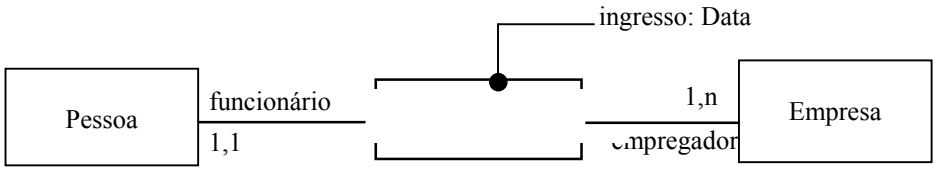


Figura 4.10 – Propriedade de uma relação

Além de representar características de conceitos, propriedades também podem caracterizar relacionamentos (figura 4.10). Usando este exemplo, pode ser observado que, do ponto de vista do modelo, a propriedade ingresso, é na verdade um relacionamento entre instâncias de contrato* - pares ordenados (p,e), tal que p ∈ Pessoa e e ∈ Empresa - e instâncias de um conjunto Z , subconjunto de Data, cujas instâncias representam datas de ingresso de um pessoa em uma empresa (figura 4.11). Formalmente,

$$\forall\ p{:}Pessoa,e{:}Empresa\ contrato(p,e) \rightarrow \exists!\underline{z}{:}Z\ ingresso(\underline{z},contrato(p,e))$$

ou simplesmente,

$$\forall\ c{:}\ contrato^*,\ \exists!\underline{z}{:}Z\ ingresso(\underline{z},c) \qquad (A12)$$

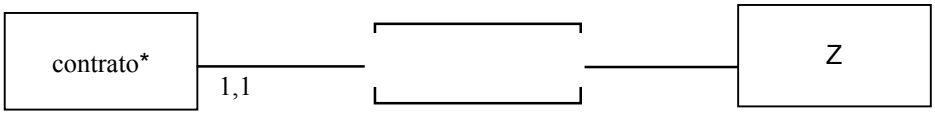


Figura 4.11 – Propriedade de uma relação vista como uma relação

Finalmente, papéis podem ser formalizados no modelo através de funções capazes de possibilitar a navegação entre instâncias de conjuntos participantes em uma relação – dada uma instância a:A, é possível obter o elemento de b:B relacionado a a - e de expressar escolhas de direcionamento para as relações. Usando o exemplo ilustrado pela figura 4.10, temos as funções: funcionário:Empresa $\rightarrow \wp$(Pessoa) e empregador:Pessoa $\rightarrow \wp$(Empresa). As funções de papéis devem que ser definidas como um mapeamento de um conjunto no conjunto potência do outro, para que a presença de cardinalidades de limite superior maior que 1 não vá de encontro à definição matemática de função, permitindo assim, preservar as funções como próprias, e ao mesmo tempo, com respostas multivaloradas. Nesse caso, dada um empresa (empregador), é possível descobrir quais são seus funcionários e dada um pessoa (funcionário), é possível descobrir seu empregador. É importante notar que a função de papéis produz resultados equivalentes a função Im, por exemplo, empregador(p) = Im(p,contrato) e funcionário(e) = Im(e,contrato). Por fim, vale ressaltar que as funções funcionário e empregador são funções adjuntas (ou reversas) e não inversas. Seja por exemplo, funcionário(UFES) = {Rosane, Ricardo}, uma função inversa faria o mapeamento de {Rosane, Ricardo} em UFES (f({Rosane, Ricardo}) = {UFES}) e teria a forma $\wp$(Pessoa) $\rightarrow \wp$(Empresa), o que não é o desejado.

4.2.1.6 – Relações todo-parte

Alguns tipos de relações têm uma semântica forte e, na verdade, escondem por detrás uma ontologia genérica, tal como a relação todo-parte. Para estes tipos de relações, uma notação especializada é proposta. De fato, esta é a característica marcante de LINGO e que a faz diferente de outras representações gráficas: qualquer notação proposta, além das notações básicas para conceitos e relações, visa incorporar uma teoria. Uma vez que a teoria é incorporada, axiomas podem ser automaticamente gerados. Esses axiomas dizem respeito simplesmente à estruturação dos conceitos e são ditos *axiomas epistemológicos*. Assim, ainda que LINGO seja uma representação de nível epistemológico, ela incorpora um mecanismo de inclusão de teorias no nível ontológico (Falbo, 1998).

Ao ser utilizada, uma relação todo-parte importa e aplica uma teoria abstrata de composição de elementos ao conteúdo da ontologia em desenvolvimento. Assim, é

importante atribuir uma notação especial para este tipo de relação, de modo que, ao utilizarmos tal representação, estejamos implicitamente incluindo a ontologia genérica que a especifica (Falbo, 1998). A figura 4.12 mostra a notação empregada para representar relações de composição.

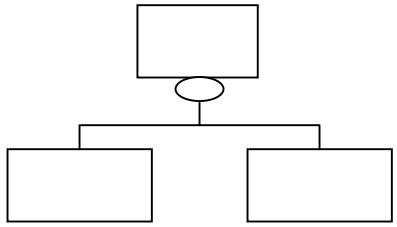


Figura 4.12 – Notação para relações todo-parte

No modelo formal, estas relações são descritas como tipos especiais de relações transitivas e anti-simétricas. As relações individuais entre o todo e cada uma de suas partes podem ser vistas como relações binárias com os respectivos papéis de *todo* e *parte* (figura 4.13).

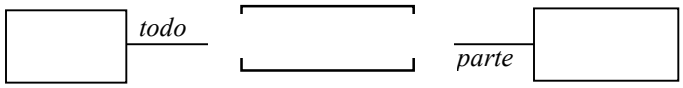


Figura 4.13 – Relação todo-parte mapeada em uma relação binária com papéis

Quaisquer que sejam os elementos envolvidos em uma relação todo-parte, os seguintes axiomas são sempre válidos:

$$\forall x,y \quad (y \in \text{parte}(x)) \rightarrow x \notin \text{parte}(y) \qquad \text{(A13)}$$

$$\forall x,y,z \quad (z \in \text{parte}(y) \wedge y \in \text{parte}(x)) \rightarrow z \in \text{parte}(x) \qquad \text{(A14)}$$

$$\forall x,y \quad \text{disjunto}(x,y) \leftrightarrow \sim\exists z\ ((z \in \text{parte}(x)) \wedge (z \in \text{parte}(y))) \qquad \text{(A15)}$$

$$\forall x \quad \text{atômico}(x) \leftrightarrow \sim\exists y\ ((y \in \text{parte}(x)) \qquad \text{(A16)}$$

Visando propor uma extensão à notação de LINGO, será introduzida uma nova primitiva para modelar um tipo especial de relação todo-parte denominado *composição* (figura 4.14). Esta relação adiciona um outro axioma ao conjunto básico desta teoria:

$$\forall x,y \ (y \in \text{parte}(x)) \rightarrow \sim\exists z \ (y \in \text{parte}(z)) \qquad \text{(A17)}$$

Como pode ser observado, em uma relação de composição as partes são exclusivas de um único todo. Uma vez feita essa distinção, as relações todo-parte ordinárias passarão a ser chamadas de *agregação* daqui em diante.

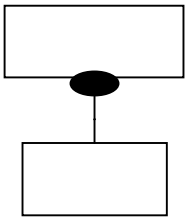


Figura 4.14 – Notação para relações de composição

É importante ressaltar que estes axiomas representam apenas um subconjunto de uma teoria muito mais abrangente de relações todo-parte. Alguns trabalhos exploram relações semânticas e de interdependência complexas entre o todo e suas partes. Um exemplo de trabalho expressivo nessa direção é apresentado em (Artale et al., 1996, Guarino, 1996).

4.2.1.7 – Relações de Especialização e Generalização

Seja um conceito *A*, em que cada uma de suas instâncias é também instância de um conceito *S*. Nesse caso, *A* é chamado de uma especialização de *S*, que por outro lado, é chamado de uma generalização de *A*. Esta relação é também conhecida como uma relação *subtipo-de*/*supertipo-de*, nesse caso *S* o supertipo e *A* o subtipo. A figura 4.15 abaixo mostra um exemplo deste tipo de relação na notação de LINGO (Falbo, 1998).

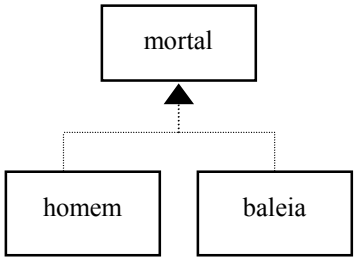


Figura 4.15 – Notação para hierarquia de conceitos

Uma relação subtipo-de se dá entre conceitos e não entre suas instâncias, desta forma, por razões de diferenciação, é usada uma linha pontilhada para representá-la, com uma seta apontando para o supertipo.

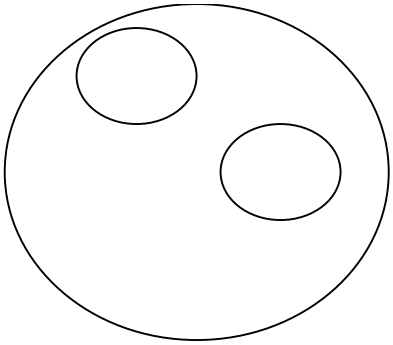


Figura 4.16 – Relação entre super-tipo e seus sub-tipos

Assim como as relações todo-parte, esta relação estabelece uma taxonomia de conceitos e apesar de atuar em um nível epistemológico, se compromete implicitamente com uma teoria de nível ontológico. A figura 4.16 acima ilustra esta relação. Como pode ser observado, toda instância de homem é também uma instância de mortal, ou mais formalmente:

$$\forall \text{ x homem(x)} \rightarrow \text{mortal (x)} \qquad \text{(A18)}$$

ou representando a relação entre os conceitos,

$$\text{Homem} \subset \text{Mortal} \qquad \text{(A19)}$$

Assim como o operador de classificação ($\in$), este operador de subconjunto se aplica em termos de intenção e extensão. É interessante notar que, do ponto de vista de extensão, Homem $\subset$ Mortal representa uma relação entre subconjuntos, de modo que o conjunto de homens possui menos elementos que o conjunto de mortais. Do ponto de vista de intenção, o axioma significa que a definição de homem deve conter a definição de mortal, sendo deste modo mais específica. Em outras palavras, no sentido descendente da hierarquia de generalização, a intenção aumenta enquanto que a extensão diminui. Por fim, (A19) ainda inclui, implicitamente, outro axioma: $\forall$ x,y (subtipo (y,x) $\rightarrow$ ~ supertipo(x,y)).

4.3 – Sistematização do processo de projeto de domínio

Como dito anteriormente, a atividade de projeto de domínio tem como objetivo a geração de infra-estruturas computacionais a partir do resultado da fase de análise de domínio. No contexto deste trabalho, o modelo do domínio é uma ontologia formal descrita na notação proposta anteriormente, e as infra-estruturas computacionais são *frameworks* de objetos.

A atividade de Projeto de domínio é dividida nas seguintes sub-atividades: especificação da infraestrutura, e seu projeto e implementação. O objetivo dessa seção é apresentar um conjunto de diretivas para a realização de forma sistemática do mapeamento das estruturas conceituais do modelo do domínio em perspectivas de especificação e implementação. Dessa forma, a seção aborda a tradução de todas as primitivas definidas pela linguagem LINGO, bem como a implementação das construções da notação formal, na forma de operações definidas sobre o tipo conjunto.

Uma vez que um *framework* é também um artefato de código, é necessária a escolha de uma linguagem de programação para implementá-lo. No caso deste trabalho, a escolha foi feita pela linguagem JAVA (Horstmann & Cornell, 1997), por implementar todos os conceitos de orientação a objetos necessários à cobertura dos requisitos de implementação das estruturas do modelo. Como benefício adicional dessa escolha, a infra-estrutura produzida é também independente de plataforma.

Além dessas construções de nível epistemológico, ou seja, que dizem respeito à estruturação dos elementos do modelo, uma das principais contribuições do modelo do

domínio, se comparado com outros modelos conceituais, é a existência de axiomas que impõem restrições de relacionamento (axiomas de consolidação) e axiomas que derivam conhecimento, respondendo às questões de competência da ontologia (axiomas ontológicos). Esses axiomas, são respectivamente, implementados através de pré-condições e invariantes, que em nosso caso, são embutidas em métodos do *framework*, pelo fato da linguagem, JAVA, adotada não suportar diretamente essas construções. A seção, então, apresenta um conjunto de regras de derivação desses métodos a partir do conjunto inicial de axiomas. Além disso, é apresentado um padrão de projeto (*design pattern)*, que modela uma solução genérica para as restrições de consolidação do relacionamento todo-parte. Este padrão é, então, implementado na linguagem escolhida.

Embora direcionado à linguagem JAVA, a princípio o processo de tradução pode ser adaptado para outras linguagens orientadas a objetos, desde que possuam algumas características importantes como a possibilidade de implementação de estruturas genéricas e polimórficas, a checagem forte de tipos e o suporte a mecanismos de reflexão computacional (meta-classes). A seção seguinte apresenta um *framework* que implementa o tipo conjunto. Esse framework será altamente utilizado daqui em diante para a manipulação dos conjuntos definidos no contexto da ontologia. As seções 4.3.2 e 4.3.3 apresentam diretivas, regras de transformação e padrões de projeto, que constituem o conjunto de ferramentas necessárias para que a transição entre ontologias e infra-estruturas de domínio possa ocorrer de forma sistemática.

4.3.1 – O tipo conjunto

A figura 4.17 mostra um *framework* que modela e implementa as funcionalidades requeridas e as propriedades matemáticas descritas na fundamentação teórica apresentada na seção 4.2.1 (vide tabela 4.1). A elemento central desse *framework* é a classe Set, cujos métodos implementam as operações do tipo conjunto. A tabela 4.2 descreve resumidamente essas operações.

Operação	Forma de invocação do método	Resultado
$\supseteq$	A.contains (B)	Verifica se o conjunto B está contido em A
=	A.equals (B)	Verifica se o conjunto B é igual a A
$\cup$	A.union (B)	Retorna o conjunto A $\cup$ B
$\cap$	A.intersection (B)	Retorna o conjunto A $\cap$ B
#	A.cardinality ( )	Número de elementos de A
{C | C $\subseteq$ A}	A.subset("C")	Retorna o conjunto C, sendo C um subconjunto de A
/	A.difference(B)	Retorna a diferença entre os dois conjuntos
$\in$	A.in(b)	Verifica se o elemento b pertence ao conjunto A
+	A.add(b)	Adiciona ao conjunto A o elemento b
-	A.remove(b)	Remove do conjunto A o elemento b

Tabela 4.2 – Métodos da classe Set **que implementam as operações fundamentais do tipo conjunto**

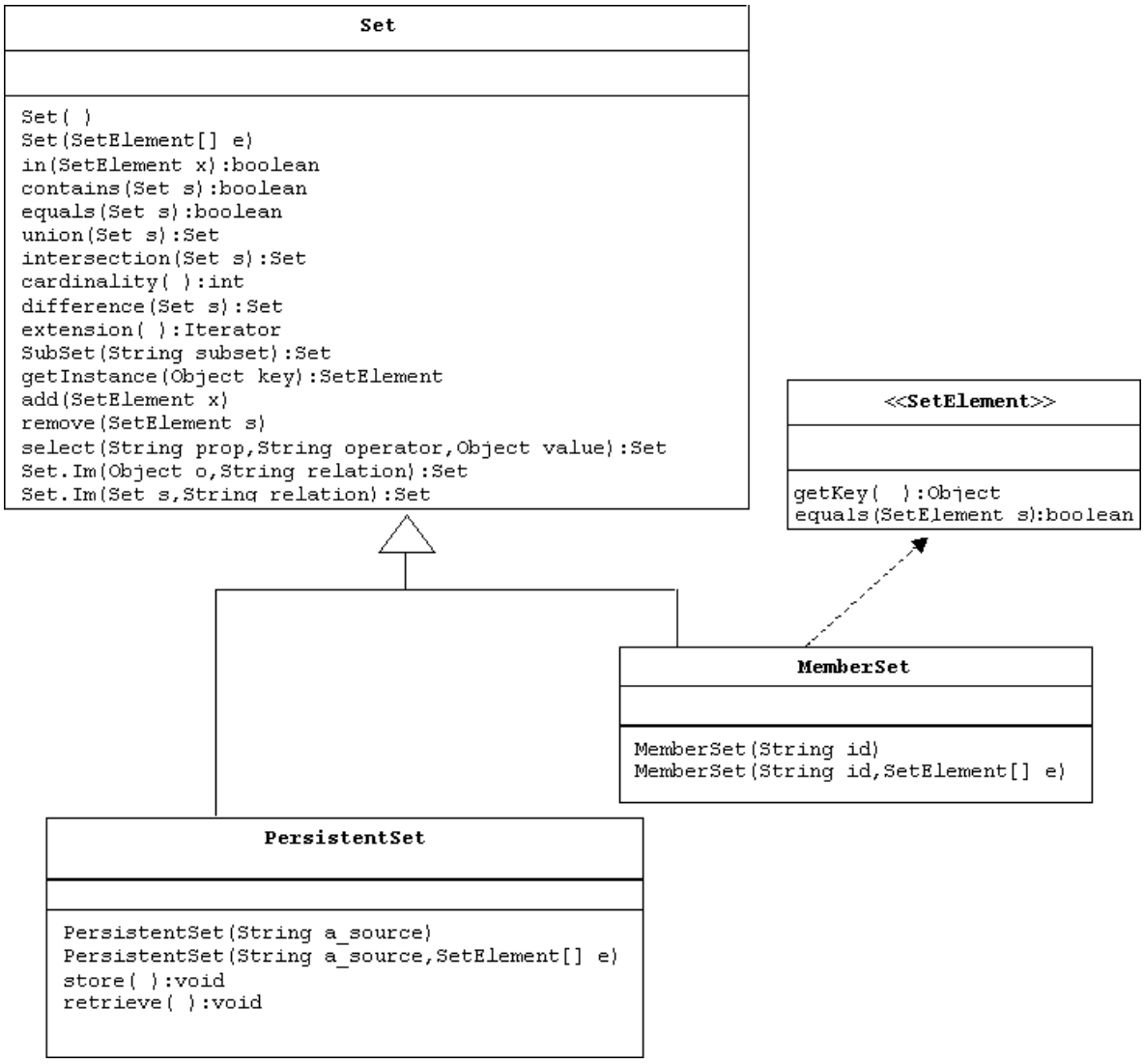


Figura 4.17 - *Framework* que implementa o tipo conjunto (Set)

Além dessas operações fundamentais, a classe Set implementa três outros métodos que merecem especial atenção. O primeiro deles é o método extension. Esse método retorna os elementos que compõem o conjunto, organizados em um objeto chamado Iterator. Esse objeto implementa um padrão de projeto definido em (Gamma et al., 1995) que é de grande utilidade para prover uma maneira uniforme de iteração em coleções, cuja ordem dos elementos é imaterial, como é o caso de um conjunto.

Um outro método implementado é o método select. Esse método tem como objetivo selecionar de um conjunto, um subconjunto formado por elementos cujas propriedades atendem a uma condição. Um exemplo disso seria selecionar, dentro de um

conjunto de carros, todas aquelas instâncias cujo ano de fabricação seja 2000. Para uma descrição mais formal desse exemplo, é utilizado o operador de seleção (σ) da álgebra relacional (Silberchatz et al., 1997).

$$X \leftarrow \sigma_{(ano = 2000)}(Carro)$$

A álgebra relacional é uma linguagem de consulta procedural composta de um conjunto de operações que atuam sobre relações. O operador de seleção representado pela letra grega sigma (σ) atua sobre uma relação, selecionando tuplas que satisfaçam a um determinado predicado. Como os relacionamentos entre conceitos e suas propriedades constituem relações, esse operador pode ser usados para selecionar elementos de um conjunto.

De uma forma geral, sendo A um conjunto, cujos elementos têm uma propriedade w, o conjunto B abaixo representa o subconjunto de A, em que todos elementos possuem a propriedade w se relacionando através do operador ***op*** com o valor z, a chamada do método A.select("w",***op***,"z") é então formalizada como,

$$B \leftarrow \sigma_{(w\ op\ z)}(A)$$

sendo a expressão (w op z) o predicado que deve ser satisfeito e A o conjunto argumento da seleção.

Por fim, a classe Set, implementa a função Im, definida na seção 4.2.1.2. Esta função é implementada de duas formas diferentes através de métodos da classe. A primeira forma implementa a função exatamente da maneira que foi definida, ou seja, dado um elemento do domínio e uma relação, o método retorna o seu conjunto imagem. A segunda forma aplica esse método para todos os elementos do conjunto A, provendo à função a capacidade de ser invocada recursivamente e mantendo a sua propriedade distributiva.

Para que fossem mantidos com o maior grau de generalidade possível e, conseqüentemente, com um maior potencial de reutilização, os métodos que implementam a função Im e o operador select foram desenvolvidos através do uso de meta-objetos. Arquiteturas de meta-nível orientadas a objetos (ou arquiteturas de meta-objetos) consideram toda a aplicação em execução, bem como as classes dos objetos que a compõem como objetos de primeira categoria. Essa tecnologia, também conhecida

como reflexão computacional, permite, em tempo de execução, que a estrutura de uma classe seja examinada, que seus atributos sejam acessados e seus métodos invocados. Além disso, mudanças feitas em um sistema reflexivo são imediatamente refletidas em seu estado dinâmico como um todo, fazendo com que sistemas desse tipo tenham um alto potencial de metamorfose e adaptatividade. No caso da função Im, por exemplo, o parâmetro relation indica o nome do método que será invocado em um ou mais objetos. Essa decisão só pode ser tomada no momento que o método Im for usado e, portanto, o nome do método invocado não pode ser conhecido previamente pelo método. O mecanismo de reflexão computacional implementado utiliza o pacote java.lang.reflect da SUN (Horstmann & Cornell, 1997).

A linguagem JAVA possui a característica polimórfica que faz com que todos os seus objetos sejam subclasses da classe Object. Dessa forma, os conjuntos podem ser genericamente implementados como uma coleção de instâncias desse tipo. Porém, para que cada um de seus elemento possa ser posteriormente recuperado pelo método getInstance(String key), é necessário que eles sejam identificados univocamente. Por esse motivo, para que um objeto de uma classe qualquer possa ser elemento de um conjunto, é necessário que ele implemente a interface SetElement, provendo assim através do método getKey, um mecanismo de identificação. Além disso a interface declara o método equals que tem o objetivo de verificar se dois elementos são iguais baseado em seus identificadores.

Para completar o framework, existem também as classes PersistentSet e MemberSet, ambas subtipos de Set. Um PersistentSet, como o próprio nome induz é um conjunto capaz de fazer sua própria persistência de forma transparente para o usuário da classe. Usando a tecnologia de serialização de objetos (Horstmann & Cornell, 1997), o conjunto e todas as suas instâncias são feitos persistentes pelo método store. Posteriormente, o mesmo conjunto é recuperado do sistema de arquivos, através do método retrieve, mantendo todas as relações entre suas instâncias.

Como será observado nos exemplos mostrados nesse capítulo e no posterior, o uso da teoria de conjuntos além dos benefícios já mencionados com relação a sua adequação para a formalização da ontologia, adiciona vários outros benefícios no nível de implementação. Conjuntos são estruturas matemáticas altamente estáveis e confiáveis

para serem usadas como estruturas de dados, e suas operações são úteis para a resolução de forma rápida e clara de vários problemas encontrados no projeto de algoritmos. Além disso, conjuntos podem ser usados como um alternativa interessante para a implementação de bases de dados. Nesse caso, uma base de dados pode ser encarada como uma **família de conjuntos** $\mathfrak{I}$ que contém todos os conjuntos existentes para uma dada aplicação, sendo que $\mathfrak{I}$ é uma instância de PersistentSet. Como nesse caso, os conjuntos serão membros de um outro conjunto, da mesma forma que para os elementos ordinários, cada um deve ter um identificador único. Para isso, é provida a subclasse MemberSet, cujas instâncias são conjuntos (Set) que implementam a interface SetElement e que, portanto, possuem o mecanismo necessário para recuperação de tal identificador.

Uma tabela *Hashing* foi a estrutura de dados escolhida como mecanismo de controle das instâncias de um conjunto. A adequação dessa escolha é devido ao fato dessa estrutura ser altamente eficiente para operações de adição e checagem de pertinência (***O(1)***) - que são algumas das mais usadas pelo tipo - em comparação com outras estruturas como, por exemplo, árvores binárias (***O(logn)***).

4.3.2 – Derivando *frameworks* orientados a objetos a partir de ontologias de domínio

A figura 4.18 mostra o resumo de uma parte específica de uma ontologia de gerência de serviços de entrega de produtos. Essa ontologia será usada como exemplo em várias situações daqui em diante, para ilustrar aspectos específicos do processo de geração do *framework*.

A infra-estrutura gerada a partir dessa ontologia tem como objetivo gerenciar para um consórcio de instituições, as relações entre seus veículos e seus motoristas contratados. Cada empresa possui alguns tipos de veículos (motocicleta, carros de passeio e veículos de carga) e contrata alguns tipos de funcionários, que podem ser motoristas do tipo A, B ou C, cada um deles possuindo habilidades específicas. Motoristas do tipo A podem dirigir qualquer tipo de veículo da empresa, motoristas do tipo B só podem dirigir veículos de passeio e motoristas do tipo C só podem dirigir motocicletas. Uma questão de competência central a essa ontologia é: *Dado um motorista contratado por uma empresa, quais são os veículos que podem ser alocados para ele ?* E uma restrição de

relacionamento importante é que uma empresa só pode contratar um motorista se possuir pelo menos um veículo.

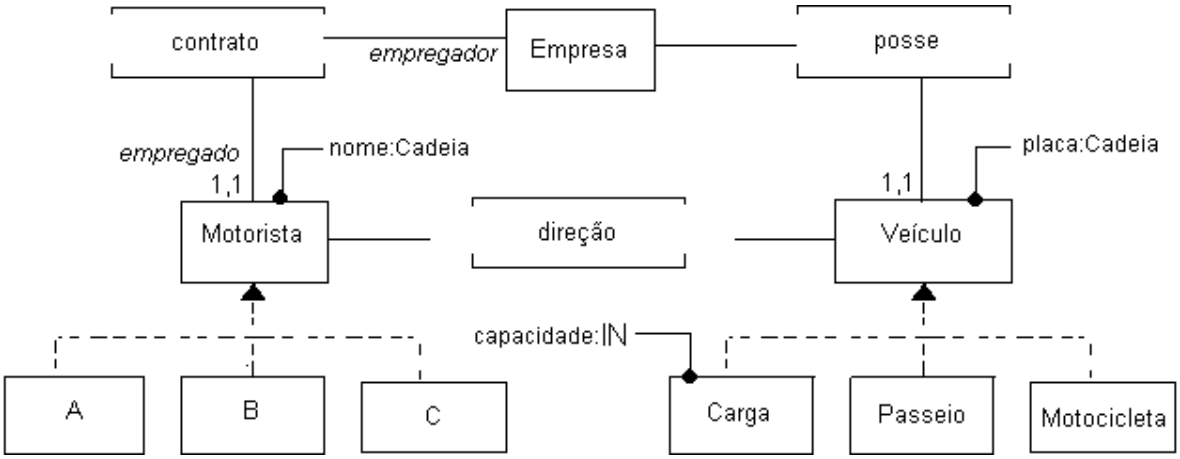


Figura 4.18 - Simplificação de uma ontologia de gerência de um consórcio de empresas de entrega de produtos

As subseções seguintes mostram a axiomatização dessa ontologia, usando-a como instrumento para apresentar o processo de tradução. A seguinte convenção é usada: axiomas do tipo (D) são de definição, (AE) de especialização, (AC) de cardinalidade, (AT) referentes a propriedades, (AO) axiomas ontológicos e, finalmente, (AR) são axiomas de consolidação.

a) Definição dos conjuntos extensão e das relações entre eles

(D1) M = Motorista	(AE1) $A \subset M$
(D2) E = Empresa	(AE2) $B \subset M$
(D3) V = Veículo	(AE3) $C \subset M$
(D4) G = Carga	(AE4) $T \subset M$
(D5) T= Motocicleta	(AE5) $G \subset M$
(D6) P= Passeio	(AE6) $P \subset M$

Os axiomas do tipo D_i acima definem conjuntos extensão para os conceitos da ontologia. Em um nível conceitual, esses conjuntos podem ser mapeados diretamente

para classes em um paradigma de orientação a objetos. Como o próprio nome já diz, classes embutem um mecanismo de classificação, definindo uma regra de formação para suas possíveis instâncias. Portanto, seguindo a definição de conjunto dada na seção 4.2.1, classes podem ser encaradas como conjuntos de objetos, inclusive em um arquitetura de meta-nível, e podem ser manipuladas como tal. Dessa maneira, as relações de classificação da linguagem formal não requerem nenhuma implementação específica, pois relações do tipo $a \in A$, são resolvidas pela tipagem da linguagem de programação, através da criação de um objeto *a* do tipo *A*. No nível de projeto e implementação, no entanto, um conceito inicialmente mapeado em uma classe conceitual, pode ser implementado através de várias classes, ou de outra forma, várias classes podem ser usadas para mapear um único conceito.

Da mesma forma que para a relação de classificação, a relação de especialização/generalização não requer nenhuma implementação adicional, ou seja, relações de sub-tipagem entre conceitos podem ser diretamente mapeadas em relações de sub-tipagem entre classes. Um axioma do tipo $A \subset M$ significa que todos os elementos pertencentes a A também pertencem ao conjunto M e que, portanto, todo elemento $a \in A$ é também um elemento do tipo Motorista. O conceito A é uma especialização do conceito Motorista, nos pontos de vista de extensão e de intenção. Um ponto importante a ser ressaltado é que os subconjuntos do conjunto M são, na verdade, partições desse conjunto, ou seja, não existe nenhum elemento no superconjunto que não seja elemento de um de seus subconjuntos, além disso a interseção entre os subconjuntos é sempre nula. De uma maneira mais formal temos,

$$((A \subset M) \wedge (B \subset M) \wedge (C \subset M)) \leftrightarrow ((\forall x \in M) \rightarrow ((x \in A) \oplus (x \in B) \oplus (x \in C)))$$

Por esse motivo, o conceito que representa um super-tipo é sempre mapeado em um classe abstrata. No nível de projeto e implementação, existem várias alternativas de implementação das relações de generalização/especialização entre classes (Martin & Odell, 1996). Nesse trabalho, no caso geral, essas relações são mapeadas pelo mecanismo de herança da linguagem. No caso de um sub-tipo possuir mais de um super-tipo, então o mapeamento é feito através de mecanismos de delegação, isso é devido ao fato da linguagem JAVA não possibilitar a herança múltipla.

b) Relações, axiomas de cardinalidade e propriedades

(D7) contrato = (Empresa, Motorista, contrato(e,m))

(D8) posse = (Empresa, Veículo, posse(e,v))

(AC1) ∀v:V #Im(v, posse) = 1 (AT2) ∀v:V ∃!p:Cadeia placa(p,v)

(AC2) ∀m:M #Im(m, contrato) = 1 (AT3) ∀g:G ∃!c:IN capacidade(c,g)

(AT1) ∀m:M ∃!n:Cadeia nome(n,m)

No nível ontológico, o direcionamento das relações é abstraído. Por esse motivo, para que seja mantido o mesmo grau de generalidade, é interessante que nos diagrama de classes, os relacionamentos também sejam implementados como sendo bidirecionais. Por exemplo, a relação contrato existente entre instâncias de Empresa e Motorista, faz com que em ambas as classes exista tanto uma variável de referência quanto um método com o mesmo nome da relação. Nesse caso, por exemplo, através da invocação do método contrato() no objeto empresa e_1, é possível acessar os seus motoristas contratados (Im(e_1,contrato)). Por outro lado, o mesmo método invocado no objeto motorista m_1 retorna a empresa que o contrata (Im(m_1,contrato)), ou em outras palavras, seu empregador, como é mostrado pelo papel.

Do ponto de vista da linguagem formal, como foi mostrado na seção 4.2.1.5, as funções que representam os papéis de cada um dos conceitos têm o mesmo resultado que a função Imagem (Im) aplicada ao conceito em questão e à relação na qual o papel é definido. Por exemplo, a função empregado(e_1) retorna o mesmo conjunto de motoristas que a função Im(e_1, contrato). Nas classes que esses conceitos são mapeados, os resultados das funções de cada papel são também equivalentes aos resultados dos métodos que representam as relações em cada uma das classes. Desse modo, essas funções são também mapeadas em métodos da classes com o mesmo nome dos papéis. Portanto, usando o exemplo anterior, temos que, o conjunto retornado por e1.contrato() é igual ao conjunto retornado por e1.empregado().

O tipo das variáveis de referência das relações e os valores de retorno dos métodos associados dependem diretamente da cardinalidade das relações. Como um empresa e_1 possui vários motoristas, a relação é mapeada em uma variável contrato do tipo Set que encapsula todas as instâncias de Motorista com os quais e_1 tem uma relação

de contrato. O método contrato() invocado em e1, retorna exatamente essa variável. Por outro lado, um motorista m_1 é contratado por uma e exatamente uma empresa e a relação contrato na classe Motorista é, portanto, mapeada em uma variável do tipo Empresa que é, então, retornada pelo método contrato() invocado em m1. Axiomas de cardinalidade que impõem um limite inferior igual a 1 têm essa restrição refletida nos construtores da classe. Por exemplo, nesse caso, no construtor da classe Motorista, um dos parâmetros é uma instância de Empresa com a qual o motorista tem relação, assegurando assim o cumprimento da restrição.

Como toda relação R define um conjunto R*, o conjunto de pares ordenados que satisfazem a relação, qualquer uma delas, a princípio, pode ser mapeada para uma classe. Essa foi a escolha adotada para relações de aridade maior que dois. Na figura 4.3, é apresentado um exemplo de relação ternária entre instâncias de Empresa, Pessoa, Cargo. Do ponto de vista matemático, a relação contrato é uma tupla contendo instâncias dos três conjuntos envolvidos. Por esse motivo, para modelar tal situação, é criada uma classe Contrato que mantém referência para os três elementos participantes da relação através dos métodos empresa(), pessoa() e cargo(), que representam exatamente o papel que cada um deles desempenha na relação. Da mesma forma, como foi mostrado no exemplo dessa figura, matematicamente a função Imagem pode ser usada, por exemplo, em uma instância de Pessoa para acessar o conjunto de pares ordenados cargos e empresas com os quais a pessoa se relaciona. Usando o exemplo em questão temos:

$$Im(Rosane,contrato) = \{(UFES,Professor),(UFES,Coordenador)\}$$

A modelagem desse situação é feita através da implementação do método contrato() nas três classes envolvidas na relação. Em cada uma delas, da mesma forma que para as relações binárias, esse método é responsável por referenciar os elementos pertencentes aos outros conjuntos com os quais o objeto se relaciona. A figura 4.19 ilustra essa situação.

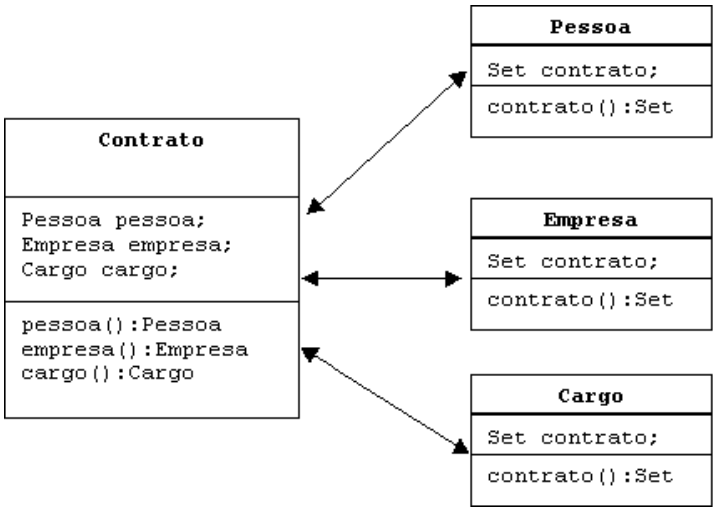


Figura 4.19 - Exemplo de Modelagem de uma relação ternária

Como citado anteriormente, conceitos e relações podem possuir relações com subconjuntos de conjuntos básicos como, por exemplo, conjuntos numéricos (naturais, inteiros, reais), conjuntos de cadeias de caracteres e conjuntos *booleanos*. Essas relações representam características que podem ser atribuídas a esses elementos e, portanto, são chamadas de propriedades ou atributos. Propriedades são modelados da mesma forma que outras relações. É importante salientar que, no caso de uma propriedade de uma relação, por exemplo, binária, a relação é vista na verdade como uma relação ternária e a solução adota é a mesma apresentada acima para relações ternárias comuns. Por fim, um último tópico que deve ser pensado é como os conjuntos básicos são mapeados para tipos básicos da linguagem de programação. A tabela 4.3 mostra esse mapeamento para o caso da linguagem JAVA. Deve-se ressaltar que, como a linguagem não possui um tipo numérico sem sinal (*unsigned*), para que essa restrição implícita ao conjunto dos números naturais seja assegurada, os métodos que modificam um atributo desse tipo devem garantir que essa pré-condição seja satisfeita.

Conjunto	Tipo da linguagem
Naturais (IN)	byte, short, int, long
Inteiros (Z)	byte, short, int, long
Reais (IR)	Double, float
Booleanos	Boolean
Cadeia	String

Tabela 4.3 - Mapeamento de conjuntos matemáticos básicos em tipos básicos da linguagem JAVA

A seguir, é mostrado um diagrama contendo os elementos da aplicação (classes e relacionamentos), derivados a partir da ontologia (figura 4.20). São também mostradas as implementações das classes Carga, Veículo e Empresa, a fim de exemplificar algumas dessas soluções de mapeamento comentadas. É importante ser notado que a notação padrão da UML (Fowler, 1997) usada no diagrama inverte o sentido de representação das cardinalidades.

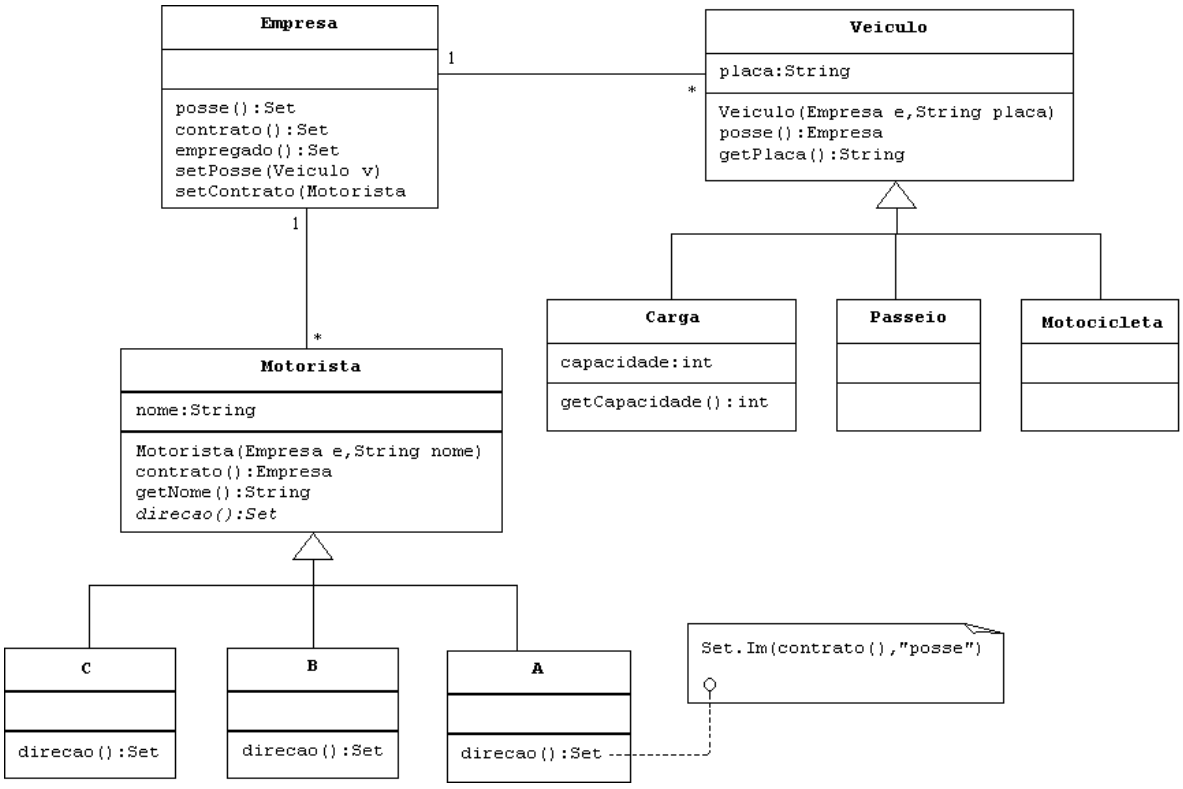


Figura 4.20 - Diagrama de classes da aplicação de gerência de um consórcio de empresas de entrega de produtos

```
public class Empresa implements SetElement
{
  //Atributos
  private String id;
    private Set posse = new Set();
    private Set contrato = new Set();
  Empresa(String codigo){ id = codigo;}

    public void setPosse(Veiculo v)
    {
        posse.add(v);
    }
```

```
    public void setContrato(Motorista m)
    {
        contrato.add(m);
    }

    public Set posse()
    {
        return posse;
    }

    public Set contrato()
    {
        return contrato;
    }

    public Set empregado()
    {
        return contrato;
    }

  // Métodos da interface SetElement
  public Object getKey()
    {
        return id;
    }

  public boolean equals(SetElement s)
    {
        return (this.getObject() == s.getObject());
    }
}
```

```
public abstract class Veiculo implements SetElement
{

    private String  placa;
    private Empresa posse;

  Veiculo(Empresa e, String p)
    {
        placa = p;
        posse = e;
        e.setPosse(this);
    }

    public Empresa posse()
    {
        return posse;
    }

    public String getPlaca()
    {
        return placa;
    }

    // Métodos da interface SetElement
  public Object getkey()
    {
        return placa;
    }
```

```
  public boolean equals(SetElement s)
    {
        return (this.getObject() == s.getObject());
    }
    }
public class Carga extends Veiculo
{
    private int capacidade;

  Carga(Empresa e, String placa, int cap)
    {
        super(placa,e);
        if (cap >=0)
            capacidade = cap;
    }

   public int getCapacidade(){ return capacidade;}
}
```

c) Axiomas ontológicos

(AO1) $\forall$a:A, v:V direção(a,v) $\leftrightarrow$ v $\in$ Im(Im(a, contrato),posse)

(AO2) $\forall$b:B, v:V direção(b,v) $\leftrightarrow$ v $\in$ (Im(Im(b, contrato),posse) $\cap$ P)

(AO3) $\forall$c:C, v:V direção(c,v) $\leftrightarrow$ v $\in$ (Im(Im(c, contrato),posse) $\cap$ T)

Os axiomas ontológicos são formalizados de modo a responder às questões de competência da ontologia. Os axiomas acima, por exemplo, respondem à questão: *Dado um motorista* m_1, *quais são os veículos que ele pode dirigir?* O conjunto resposta dessa pergunta deve ser o conjunto retornado pela invocação do método direcao() em um objeto m1 da classe Motorista. No entanto, para que esse método possa ser derivado a partir dos axiomas desse tipo, é necessário antes que algumas regras de transformação sejam definidas.

T0: □ x:X, □ y:Y $r_1(x,y) \leftrightarrow$ □$y \in C \Rightarrow$

Im(x, r_1):Tipo ≡ C, sendo que **se** # Im(x, r_1) = 1 **então** Tipo = Y **senão** Tipo = Set

Essa regra diz que: se para uma dada instância x do tipo X, x se relaciona com todas as instâncias y de um conjunto C (e somente com instâncias desse conjunto) em um relação r_1, o conjunto retornado pela função Im(x, r_1) será exatamente o conjunto C. O tipo retornado pela função implementada na classe, depende da cardinalidade da relação, se x se relaciona apenas com uma instância de Y, o valor retornado será do tipo Y, caso contrário será do tipo Set.

T1: □ x:X, □ y:Y $r_1(x,y) \leftrightarrow$ □ □$y \in C) \wedge$ (propriedade$_1$(y) operador expressão), tal que expressão = propriedade$_2$(x) $\oplus$ constante $\Rightarrow$

Im(x, r_1):Tipo ≡ $\sigma_{\text{propriedade(y) operador expressão}}$(C).

Seja D um subconjunto de C no qual todos os elementos tem uma de suas propriedades satisfazendo a uma relação específica com uma expressão. Essa expressão pode denotar uma propriedade de x, instância do conjunto X, com o qual C está envolvido através da relação r1, ou ainda um valor constante. Um exemplo do primeiro caso seria: suponha que todo motorista tivesse uma relação de direção com todos os veículos pertencentes a empresa que o contrata, mas que tivessem o atributo "ano de fabricação" menor que o seu atributo "ano de nascimento"

□ m:Motorista, v:V direção(m,v) $\leftrightarrow$

$\sigma_{\text{ano de nascimento(m) > ano de fabricação (v)}}$(Im(Im(m,contrato),posse))

Nesse caso, o conjunto retornado pela função Im(m,direção) será exatamente o subconjunto de Im(Im(m,contrato),posse) resultante da aplicação do operador seleção da álgebra relacional. Assim como na regra anterior, o tipo retornado pela função direcao() implementada na classe Motorista depende diretamente da cardinalidade dessa relação.

T2: Im(x, r_1) $\Rightarrow$ x.r_1()

T3: r_1(x,y) $\Rightarrow$ x.r_1()

T4: r_1(x) $\Rightarrow$ x.r_1()

Como mencionado anteriormente, uma relação (papel ou propriedade) r_1 entre dois conceitos X e Y é mapeada nas classes que representam esses conceitos como métodos com o mesmo nome da relação. Tais métodos retornam o conjunto de objetos da outra classe participante da relação, com quais a instância da classe se relaciona.

T5: $Im(A, r_1) \Rightarrow Set.Im(A," r_1")$

T6: $\sigma_{\text{propriedade(x) operador propriedade(y)}}(C) \Rightarrow$

Set.select(propriedade(x),operador, propriedade(y), C)

Essas regras promovem a substituição direta das formas matemáticas da funções *Imagem* ($Im: \wp(A) \times \Phi \rightarrow \wp(B)$) e do operador *Seleção*, pelas sintaxes através das quais eles são implementados pelos métodos da classe Set, como descrito na seção 4.3.1. O método select (que implementa o operador de *Seleção*) recebe como parâmetro referente ao operador uma String correspondente. O valor desse parâmetro segue a convenção descrita abaixo e tem o seu conjunto de possibilidades dependente do tipo dos operandos.

(i) Os operandos são dois objetos: = (equals), ≠(not_equals)

(ii) Os operandos são dois tipos básicos: =, ≠, ≥(GTET), ≤ (LTET), <(LT), >(GT)

(iii) Os operandos são, um objeto e um conjunto: ∈(in),∉(not_in)

T7: $x.r_1():Y \equiv C \Rightarrow$ public class X

```
{
    public Y r1( )
    {
        return C;
    }
}
```

Por fim, essa regra traduz diretamente o axioma escrito na forma à esquerda, para a sintaxe de implementação correspondente na linguagem de programação escolhida. Todas as referências à instância x existentes no conjunto C são substituídas pela palavra reservada this, para que a referência a métodos dentro da própria classe possam ser feitas.

A seguir é apresentado o processo de derivação do axioma A01, bem como a sua implementação na classe MotoristaA (vide figura 4.20).

(AO1) ∀a:A, v:V direção(a,v) ↔ v ∈ Im(Im(a, contrato),posse)

1. Im(a,direcao):Set ≡ Im(Im(a, contrato),posse)	A01, T0
2. a. direcao():Set ≡ Im(a.contrato(),posse)	1, T2
3. a. direcao():Set ≡ Set.Im(a.contrato(),"posse")	2, T5
4. public class MotoristaA	3, T7

```
{
    public Set direcao()
    {
        return Set.Im(contrato(),"posse");
    }
}
```

```
public class MotoristaA extends Motorista
{
    MotoristaA(Empresa e, String id, String nome)
    {
        super(e,id,nome);
    }

    public Set direcao()
    {
        return Set.Im(contrato(),"posse");
    }
}
```

d) Axioma de consolidação

(AR1) ∀e:E, m:M contrato(e,m) → #Im(e,posse) ≥ 1

Os axiomas de consolidação têm como objetivo descrever pré-condições que devem ser satisfeitas para uma relação possa ser estabelecida entre dois elementos. No caso do axioma AR1, para que uma empresa e_1 possa contratar um motorista m_1, e_1 precisa possuir pelo menos um veículo, ou seja, para que uma relação de contrato possa ser estabelecida entre os dois objetos, a pré-condição $\#Im(e_1,posse) \geq 1$ deve ser satisfeita.

De uma forma geral, seja um axioma de consolidação $\forall x:X, y:Y\ r_1(x,y) \rightarrow$ (pré-condição$_1$) $\wedge$ (pré-condição$_2$) $\wedge$ ... $\wedge$ (pré-condição$_n$), esse axioma pode ser traduzido em um padrão capaz de garantir a avaliação de cada uma das condições antes que a relação seja efetivada. O modelo desse padrão é mostrado a seguir:

```
public class X
{
    public boolean setr1 (Y y)
    {
        boolean result = false;
        if (result = (checkCondição1(...) && checkCondição2(...) ... &&
        checkCondiçãon(...))
       {
                r1.add(y2);
                y.setr1(this);
        }
        return ok;
   }

    private boolean checkRelacao1 (...)
    private boolean checkRelacao2 (...)
    private boolean checkRelacaon (...)
}
```

Esse padrão segue o modelo do padrão comportamental de projeto *Template Method* (Gamma et al., 1995), tendo como *template method* o método **setr$_1$**, e como *hook methods* os métodos que avaliam as pré-condições.

A seguir o padrão é aplicado ao axioma AR1:

```
public class Empresa
{
    public boolean setContrato (Motorista m)
    {
        boolean result = false;
        if (result = (checkPosse( ))
       {
                contrato.add(m);
                m.setContrato(this);
        }
        return ok;
    }

    private boolean checkPosse ( )
    {
        return ((this.posse()).cardinality() >= 1);
    }
}
```

Na seção seguinte, esse padrão - a partir de agora chamado de padrão ***Pré-Condição*** - é usado para compor um outro padrão de projeto, o padrão ***Todo-Parte***. Esse último contempla uma solução genérica para implementação da relação todo-parte, embutindo todos os axiomas de consolidação presentes na teoria que a descreve (seção 4.2.1.6).

4.3.3 - O padrão *Todo-Parte*

Como definido na seção 4.2.1.6, as relações todo-parte de maneira geral definem um conjunto de axiomas – A13 a A17 - que constituem um teoria subjacente incorporada à sua utilização.

$$\forall x,y \quad (y \in \text{parte}(x)) \rightarrow x \notin \text{parte}(y) \qquad \text{(A13)}$$

$$\forall x,y,z \quad (z \in \text{parte}(y) \wedge y \in \text{parte}(x)) \rightarrow z \in \text{parte}(x) \qquad \text{(A14)}$$

$$\forall x,y \quad \text{disjunto}(x,y) \leftrightarrow \sim\exists z\,((z \in \text{parte}(x)) \wedge (z \in \text{parte}(y))) \qquad \text{(A15)}$$

$$\forall x \quad \text{atômico}(x) \leftrightarrow \sim\exists y\,((y \in \text{parte}(x)) \qquad \text{(A16)}$$

$$\forall x,y \quad (y \in \text{parte}(x)) \rightarrow \sim\exists z\,(y \in \text{parte}(z)) \qquad \text{(A17)}$$

Esses axiomas ressaltam um conjunto de pré-condições exigidas para o estabelecimento da relação entre o *todo* e suas *partes*. Mesmo os axiomas definidos como de definição (A15 e A16) desempenham este papel. O axioma A14 que é uma axioma de derivação, ao invés de atuar diretamente nesse sentido, impõe que a pré-condição definida em (A13) seja verfificada recursivamente.

Uma vez que *todo* e *parte* desempenham papéis específicos e que a teoria subjacente é válida para qualquer relação desse tipo, uma estratégia genérica definindo um padrão de solução pode ser modelada. A figura 4.21 mostra o padrão *Todo-Parte*. Esse padrão é composto pelo padrão *Pré-Condição* descrito na seção anterior, e conta com quatro classes genéricas fundamentais, além da presença de duas classes concretas (A e B):

a) ***Todo***: Essa classe representa o papel do *todo* da relação. Como pode ser observado, ele é divido hierarquicamente nos papéis *Agregação* e *Composição* e encapsula de forma adequada o conjunto de axiomas pertencentes à teoria. Através da implementação do padrão *Pré-condição*, essa classe é responsável por garantir à classe concreta a qual está associada (ex. A), a verificação de um conjunto de pré-condições necessárias para que o relacionamento entre ela e suas pretensas *partes* possa ocorrer. Esse serviço é prestado à classe concreta através de um mecanismo de delegação. No caso desse padrão, o método responsável pelo estabelecimento do relacionamento é setParte e as pré-condições são:

(i) *aciclicidade* (A13 e A14): para que uma *parte* possa compor (ou agregar) um *todo*, é necessário que o *todo* não componha (ou agregue) a *parte*, nem nenhuma das *partes* dessa *parte*;

(ii) uma pré-condição específica das subclasses - PreCondEspec: no caso do *todo* representar uma *Composição*, a pré-condição é a de *exclusividade* (A17), ou seja, para que uma *parte* possa compor um *todo*, então é necessário que ela não já componha um outro *todo*. Se o *todo* for uma *Agregação*, então é necessário que a *parte* não agregue nenhum dos objetos disjuntos desse *todo* (A15).

Para auxiliar os verificadores dessas pré-condições, essa classe implementa ainda o método parte-de(ITodo w, IParte c), cujo objetivo é verificar se o segundo argumento é parte do primeiro.

Por fim, essa classe é responsável por manter a referência ao conjunto de *partes* de um *todo*, acessível através do método parte.

b) ***Parte***: Representa o papel da *parte* na relação, mantendo a referência ao seu *todo* - no caso da relação de *Composição*, ou ao conjunto deles - caso contrário. Desse modo, assim como a classe Todo, essa classe funciona como um manipulador dos elementos envolvidos na relação.

c) ***ITodo*** e ***IParte***: Essas interfaces devem ser implementadas pelas classes concretas (A e B) que constituem respectivamente o *todo* e a *parte* no contexto do padrão. Para um objeto que implementa a interface ITodo, o método todo() retorna o objeto Todo correspondente. Uma situação análoga ocorre para as classes que implementam IParte. Além de proverem às classes concretas referências aos seus respectivos manipuladores (Todo e Parte), essas interfaces asseguram a presença dos métodos que elas descrevem nas classes concretas que as implementam. A certeza da implementação desses métodos, possibilita aos manipuladores a realização de forma genérica das tarefas de verificação das pré-condições.

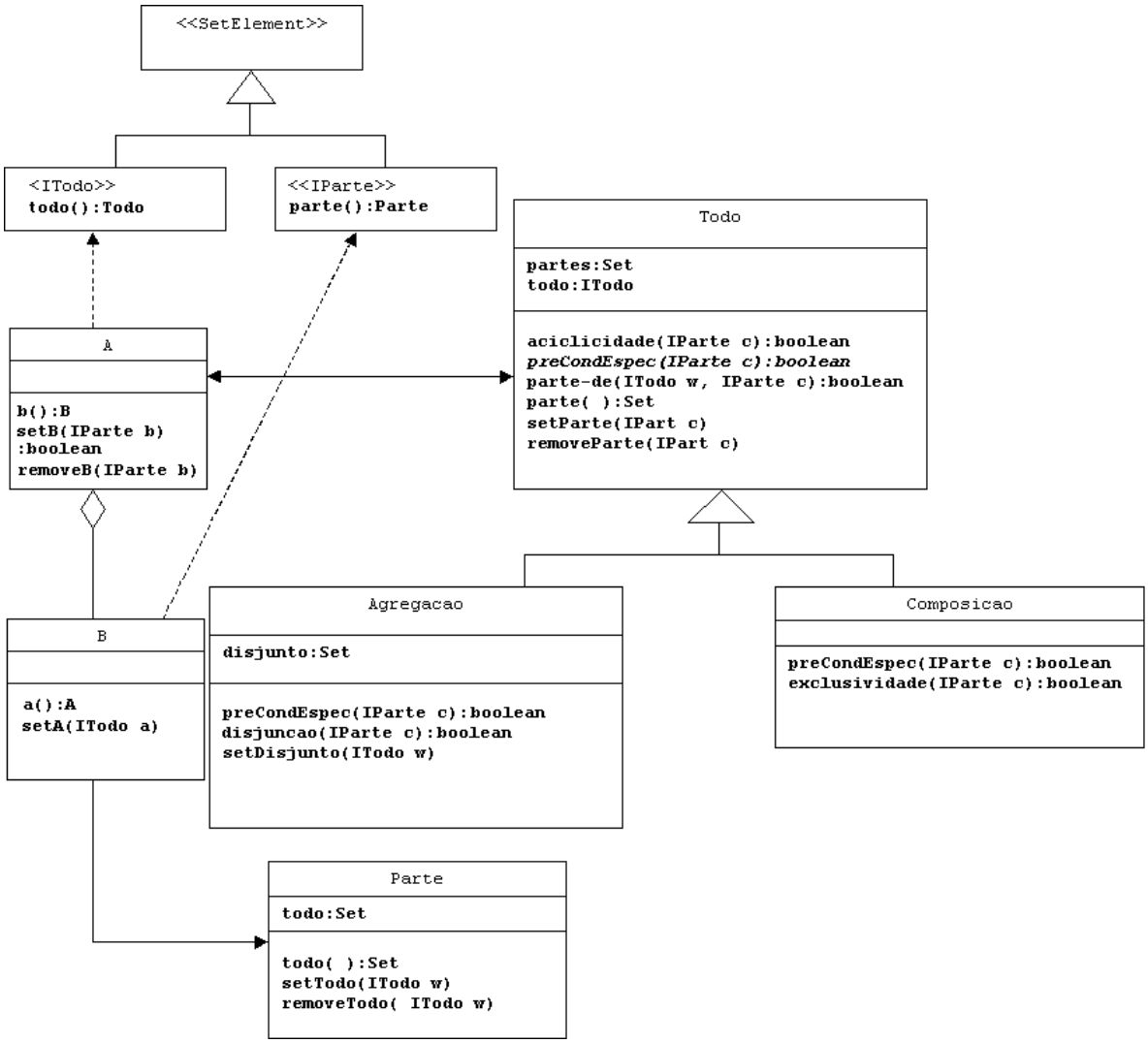


Figura 4.21 - Padrão de projeto *Todo-Parte*

Conforme mencionado anteriormente, as relações individuais entre o *todo* e cada uma de suas *partes* podem ser vistas como relações binárias com os respectivos papéis de *todo* e *parte*. Desse modo, as mesmas sugestões de derivação para relações binárias e papéis descritas na seção 4.3.2 podem ser aplicadas. A figura 4.22 mostra essa situação para as classes concretas Carro e Motor, usando o mesmo exemplo da figura 4.13 - relacionamento de agregação entre um objeto Carro e suas partes (Chassi e Motor) .

Figura 4.22 - Relação de agregação entre um objeto Carro e seu Motor como uma relação binária com papéis

A seguir uma implementação para as classes Carro e Motor é apresentada.

```
public class Carro implements ITodo
{
    Agregacao a = new Agregacao(this);
    Motor motor;

    public Todo todo()
    {
        return a;
    }

  public boolean setMotor(IParte m)
    {
        boolean result = a.setParte(m);
        if (result)
            motor = m;
        return result;
    }

    public Motor motor()
    {
        return motor;
    }
```

```
    public void removeMotor(IPart m)
    {
        motor = null;
        a.removeParte(m);
    }
}

public class Motor implements IParte
{
    Parte p = new Parte();
    Carro carro;

    public Parte parte()
    {
        return p;
    }

    public void setCarro(ITodo c)
    {
        carro = c;
        p.setTodo(c);
    }

    public Carro carro()
    {
        return carro;
    }
}
```

4.4 – Conclusões

Este capítulo completa o processo de sistematização da engenharia de domínio iniciado no capítulo anterior. Inicialmente, é mostrada a abordagem utilizada para representação de ontologias, composta de uma linguagem gráfica para estruturação de conceitos e relações (LINGO) e uma linguagem formal para a definição de axiomas. O uso dessa abordagem composta se mostrou extremamente apropriado, por permitir a representação do modelo do domínio com o rigor matemático necessário, porém sem perder a clareza e a intuitividade, necessárias à comunicação entre especialistas do domínio e engenheiros de software.

O uso de uma linguagem apropriada para descrição de ontologias, ao invés de outras linguagens gráficas, como diagramas de classes e modelos E-R, também se mostrou altamente adequado. Primeiramente, pelo fato de LINGO não embutir os compromissos ontológicos existentes nessas outras linguagens. Além disso, apesar de ser uma linguagem de estruturação, LINGO permite a inclusão de teorias em um nível ontológico, fazendo com que axiomas ontológicos e de consolidação possam ser automaticamente gerados, a partir do uso de primitivas de nível epistemológico, como é o caso das relações *todo-parte* e *generalização/especialização.* Por fim, por ser uma linguagem aberta, foi possível a introdução de novas primitivas, como a *relação de composição* e a *notação de representação de propriedades*.

É também apresentada, da maneira mais clara e legível possível, a fundamentação teórica da linguagem formal proposta. Essa linguagem, que é um misto de lógica de primeira ordem e teoria dos conjuntos, possui o nível de abstração necessário para a representação de ontologias e, ao mesmo tempo, serve como ferramenta teórica para a transformação de ontologias em infra-estruturas de domínio orientadas a objetos. Além disso, o capítulo discute distinções ontológicas importantes que influenciam decisões de modelagem, como é o caso da escolha do que deve ser modelado como conceito, propriedade ou papel.

Com o intuito de prover uma disciplina estruturada para a fase de projeto de domínio, é apresentado um conjunto composto de diretivas, regras de transformação e padrões de projeto, capazes de conduzir de maneira sistemática a tradução das estruturas conceituais da ontologia em construções da infra-estrutura de domínio, e da linguagem de

programação adotada (JAVA). Esse processo é suportado por um *mini-framework*, implementado nessa linguagem, que concretiza as operações do tipo conjunto.

Por fim, vale salientar que assim como LINGO, a linguagem formal proposta não deve ser encarada como uma solução completa, mas sim uma proposta aberta, passível de extensão. Da mesma forma, apesar da abordagem de derivação de infra-estruturas estar direcionada à linguagem de programação adotada, ela pode ser adaptada para outras linguagens, desde que possuam algumas características importantes, como a possibilidade de implementação de estruturas genéricas e polimórficas, a checagem forte de tipos e o suporte a mecanismos de reflexão computacional (meta-classes).

O capítulo seguinte apresenta um estudo de caso que faz uso da metodologia de Engenharia de domínio desenvolvida neste capítulo e no anterior.

Capítulo 5

Hipervisão: Um Estudo de Caso

nada se torna real até que seja experimentado

John Keats

5.1 - Introdução

Ao longo dos capítulos anteriores, esse trabalho tem sido estruturado de forma que cada capítulo pudesse suprir uma demanda gerada no capitulo anterior. Do mesmo modo, o objetivo desse capítulo é suprir a demanda de vários estudos de caso gerada por todas as propostas que têm sido feitas ao longo dessa dissertação. No Capítulo 2, foi proposto um modelo de processo genérico de desenvolvimento de software para/com reuso, que contempla, de forma complementar, as disciplinas de engenharia de software e engenharia de domínio. Além disso, o capítulo defende o fato de que o desenvolvimento de software deve ser encarado como uma disciplina de engenharia, ou seja, deve agrupar atividades gerenciais como avaliação da qualidade, documentação, análise de risco e planejamento, além de pregar que a escolha das atividades de construção deve ser baseada nas características da equipe de desenvolvimento, nos recursos disponíveis, no conhecimento disponível acerca do domínio e, principalmente, nos requisitos funcionais e não funcionais do produto a ser desenvolvido. Para desenvolvimento das atividades referentes à Engenharia de Domínio, uma metodologia foi proposta nos capítulos 3 e 4. O capítulo 3 apresenta um método sistemático para a realização da fase de análise de

domínio através do uso de ontologias formais e o capítulo 4 introduz um uma linguagem matemática para formalização de ontologias e um conjunto de ferramentas metodológicas para a derivação sistemática de infra-estruturas de domínio (no caso *frameworks* orientados a objetos).

Dessa forma, esse capítulo é estruturado de modo a agrupar de forma complementar os vários estudos de caso requeridos, culminando no desenvolvimento de uma aplicação. Primeiramente, a seção 5.2 promove uma discussão detalhada a respeito dos requisitos metodológicos da área de sistemas multimídia distribuídos, a fim de instanciar o modelo proposto no capítulo 2. Como resultado, é proposto um processo de desenvolvimento, que supre inúmeras demandas não cobertas pelos métodos atualmente empregados para o desenvolvimento dessa classe de aplicações.

O domínio de Vídeo sob Demanda é o sub-domínio dos sistemas multimídia distribuídos escolhido para ser alvo da aplicação do processo instanciado. Por isso, esse domínio é profundamente discutido na seção 5.3. Após essa fase, na seção 5.4, são apresentados dois estudos de casos, referentes às propostas feitas nos capítulos 3 e 4, abordando a perspectiva de desenvolvimento para reuso. Primeiramente, é desenvolvida uma ontologia de gerência de Vídeo sob Demanda e em seguida é gerado o *framework* correspondente.

Por fim, a seção 5.5 apresenta um estudo de caso de desenvolvimento com reuso, ao desenvolver uma aplicação de vídeo sob demanda a partir de dois componentes existentes: o *framework* de gerência, desenvolvido na seção 5.4, e um outro componente chamado *Java Media Framework*. A seção 5.6 apresenta as conclusões do capítulo.

5.2 - Sistemas Multimídia Distribuídos

Os anos recentes experimentaram um enorme avanço nas tecnologias de computadores e comunicação de dados, os quais constituíram o suporte tecnológico necessário para o surgimento de uma nova e promissora classe de aplicações - as **aplicações multimídia** - e abriram espaço para a concepção de novos serviços nas mais variadas áreas do conhecimento humano, como, por exemplo, educação, medicina, entretenimento, turismo, publicidade, negócios e indústria, entre outras.

A multimídia constitui uma tecnologia interdisciplinar que orienta aplicações que visam atender às necessidades multi-sensoriais da natureza humana. Sistemas multimídia

tornam isso possível devido às novas alternativas de integração, manipulação, armazenamento, transmissão, exibição e interação de diferentes formatos de mídia (texto, gráfico, animação, áudio e vídeo).

Os sistemas distribuídos são caracterizados pela existência de vários nós autônomos de processamento, possivelmente distribuídos geograficamente, que cooperam através da coordenação de atividades e troca de informações para a realização de objetivos comuns - o provimento de um conjunto de serviços. O casamento dessas duas tecnologias - sistemas multimídia e sistemas distribuídos - tendo como suporte de comunicação, idealmente, uma rede multi-serviços de alta-velocidade, resulta nos sistemas multimídia distribuídos (Gonçalves, 1996).

Estas aplicações revelam novos problemas e introduzem conceitos que afetam profundamente as arquiteturas de comunicação clássicas. Nesses últimos anos, várias pesquisas vêm sendo realizadas nestas áreas, porém, boa parte dos trabalhos aborda o problema do ponto de vista tecnológico, tentando obter desempenhos ótimos de meios de armazenamento e comunicação. A seção seguinte discute essa classe de sistemas sob uma visão diferente, focando nos aspectos metodológicos de desenvolvimento.

5.2.1 – Aspectos metodológicos de Sistemas Multimídia Distribuídos

À primeira vista, os principais requisitos dos sistemas multimídia distribuídos estão ligados a problemas de projeto, como sincronização de mídias contínuas, fortes relações temporais, fortes requisitos de tempo-real e de desempenho, forte comunicação entre objetos - especialmente objetos distribuídos - e grande presença de objetos baseados em estado. Requisitos dessa natureza necessitam de um apoio metodológico robusto, capaz de especificá-los, validá-los e simulá-los. Dessa forma, especificação e validação, incluindo verificação, simulação e teste, são de importância crucial no ciclo de vida dos sistemas distribuídos, sobretudo para o desenvolvimento de suas aplicações. A especificação não pode ser ambígua e deve ser livre de erros tanto quanto possível. Para que haja uma integração bem sucedida de novos produtos, é necessário que testes criteriosos sejam realizados em relação à especificação (Souza, 1999).

As metodologias de desenvolvimento vigentes apresentam sérias deficiências nessas áreas. Técnicas informais ou semi-formais, normalmente geram especificações incompletas, contendo problemas como ambigüidades, contradições, falta de precisão e

falta de controle da separação de níveis diferentes de abstração (Pressman, 1997). A dificuldade de uma abordagem formal, também para verificação e validação, faz com que erros sejam freqüentemente detectados somente em fases tardias do processo de desenvolvimento.

Com o aumento da complexidade de tais sistemas e com a necessidade de estabelecimento de padrões, Técnicas de Descrição Formal (TDFs) foram desenvolvidas, visando a produção de especificações claras, concisas e consistentes. Com base na sintaxe e semântica formais da TDF utilizada para a especificação, validações da própria especificação e do projeto do sistema podem ser realizadas. Utilizando um compilador para a TDF empregada, implementações semi-automáticas podem ser geradas. Com base na especificação formal do sistema, seqüências de teste podem ser geradas e linhas de execução da implementação, podem ser verificadas (Souza, 1999).

Vários exemplos de uso de TDFs no ciclo de vida de sistemas distribuídos complexos são relatados na literatura, entre eles o projeto ESPRIT LOTOSPHERE (Bolognesi et al., 1995) e o projeto francês "Conception Formelle de Systèmes Multimédias Coopératifs à Hauts Débits (Cesame)", fruto de uma colaboração entre o Centre National d'Études de Télécommunications (CNET) e o Centre National de Recherche Scientifique (CNRS), sendo este último um dos mais significativos no momento (Diaz & Pays, 1994).

Desse modo, tradicionalmente, os modelos de processo propostos pela comunidade de sistemas multimídia distribuídos são fundamentados no uso de refinamentos sucessivos de especificações construídas com técnicas de descrição formal, como LOTOS (Language of Temporal Ordering Specification) (ISO IS 8807, 1989), ESTELLE (Extended State Transition Language) (ISO IS 9074, 1989) e SDL (Specification and Definition Language) (ITU-T Z 100, 1994), entre outras, altamente difundidas em áreas como engenharia de protocolos e sistemas de tempo real. Essas técnicas tratam apropriadamente requisitos como controle de sincronização e de tempo real, possibilitando sua simulação e verificação com alto formalismo e rigor matemático.

Seguindo a mesma filosofia, em 1996 foi criado no Brasil um projeto multi-institucional denominado DAMD (Design de Aplicações Multimídia Distribuídas) com objetivo de abordar aspectos metodológicos da construção dessa classe de sistemas. Este

projeto, financiado pelo CNPq, programa temático ProTem-CC, Fase III, contou com a presença das seguintes instituições: Universidade de São Carlos (UFSCar), Universidade Federal de Santa Catarina (UFSC), UFRGS (Universidade Federal do Rio Grande do Sul), Universidade Federal do Espírito Santo (UFES) e a Universidade de Twente (Enschede, The Netherlands).

O projeto DAMD foi concebido visando três linhas mestras:

1. desenvolvimento de uma metodologia, suportada por um modelo formal (E-LOTOS[7]) e ferramentas para a especificação, validação, tradução e teste de aplicações multimídia distribuídas;

2. desenvolvimento de aplicações para a avaliação da metodologia e teste dessas aplicações em ambiente distribuído;

3. pesquisa visando propor uma nova TDF para dar suporte ao design de futuras aplicações distribuídas.

O projeto foi concluído em dezembro de 1998, produzindo os seguintes resultados:

(a) Uma ferramenta de autoria de apresentações multimídia;
(b) Um ambiente para criação de especificações em E-LOTOS usando um linguagem gráfica (E-DART);
(c) Um Simulador e um Verificador para especificações E-LOTOS;
(d) Um conjunto de diretivas para especificação de Documentos Multimídia em E-LOTOS;
(e) Um Tradutor E-LOTOS/MHEG-5[8];
(f) Estudos de caso de modelagem de aplicações multimídia distribuídas usando diferentes formalismos.

O sistema de vídeo sob demanda Hipervisão, apresentado ao longo deste capítulo, foi construído neste contexto (Guizzardi & Gonçalves, 1999b), a fim de explorar a segunda linha de atuação do projeto. No processo de desenvolvimento desta aplicação, foi observado que a metodologia proposta era mais adequada para o desenvolvimento de

[7] E-LOTOS: Enhancements to LOTOS - É uma extensão temporizada de LOTOS

uma classe específica de aplicações multimídia distribuídas. Além disso, devido ao escopo do projeto, o processo proposto não contemplava de forma sistemática um desenvolvimento para/com reuso e nem previa a adoção de atividades não tecnológicas como avaliação da qualidade, planejamento e análise de risco.

A metodologia proposta no DAMD, assim como a enorme maioria das metodologias existentes, não considera o fato de que essas aplicações são geralmente utilizadas em domínios imaturos do conhecimento, de natureza complexa e pouco conhecidos, e que, portanto, definem um novo conjunto de requisitos de natureza completamente diferente - os requisitos relacionados ao domínio do problema. Quando é considerada, a fase de análise de requisitos é tratadas por esses métodos de duas maneiras: (1) através do uso técnicas informais complementares, ou (2) empregando as mesmas TDFs usadas para resolver os problemas de projeto.

No primeiro caso, a especificação de requisitos será exposta a todos os problemas já discutidos no uso de abordagens informais, tornando possível que a presença de inúmeras restrições, inconsistências e contradições presentes no domínio, façam do modelo de requisitos - que em um sistema desse tipo é produto de uma atividade extensa e custosa - um artefato praticamente impossível de ser validado e reutilizado.

No segundo caso, como foi observado no desenvolvimento do sistema Hipervisão, o tipo de semântica formal empregado nas TDFs utilizadas, embora apropriado para modelagem de problemas de projeto, dificulta seu uso na modelagem de requisitos em níveis mais altos de abstração como, por exemplo, os níveis equivalentes às fases de análise e especificação de requisitos e análise de domínio. Essa limitação deve-se ao fato de ser altamente indesejável que a construção de um modelo do problema esteja limitada a fortes características semânticas de uma linguagem específica, assumindo compromissos ontológicos que não existem no mundo real.

Dessa forma, a área de sistemas multimídia distribuídos, devido a sua natureza complexa e heterogênea, apresenta inúmeras características que justificam a existência de metodologias específicas para a modelagem e construção de suas aplicações. A concepção dessas metodologias apresenta-se como um grande desafio, principalmente

---

[8] MHEG: Multimedia and Hypermedia Expert Group - Padrão ISO para Modelo de Documentos Hipermídia (ISO 13522-1, 1994)

pela necessidade de comportar, em um mesmo processo de desenvolvimento, uma flexibilidade de modelagem que permita o emprego de diversos níveis de formalismo.

Reforçando a idéia da necessidade de combate ao dogmatismo metodológico instituído, em (Bowan & Hinchley, 1994) são apresentados os chamados "Dez mandamentos dos métodos formais" - um conjunto de diretivas para aplicação de métodos formais em engenharia de software. As diretivas reforçam a necessidade de manutenção das atividades de estimativas de custos, documentação e avaliação da qualidade, e defendem a idéia de que o processo de desenvolvimento tem que ser encarado do ponto de vista de engenharia, ou seja, a escolha de métodos, técnicas e ferramentas deve ser feita de forma apropriada, dependendo principalmente do produto a ser construído. Desta forma, várias conclusões importantes são apresentadas:

(a) Métodos formais não têm necessariamente que ser aplicados a cada aspecto de um sistema;
(b) É bastante desejável pensar na integração de métodos formais com métodos tradicionais orientados a objetos. Cada um deles tem vantagens, que associadas de forma complementar podem produzir excelentes resultados;
(c) É interessante pensar em uma abordagem multi-formalismo, na qual vários métodos formais diferentes possam ser associados para resolver problemas específicos a diferentes fases do processo de desenvolvimento.

Várias metodologias vêm sendo criadas, a fim de integrar métodos formais em um ciclo tradicional de desenvolvimento orientado a objetos. Alguns exemplos são: INSYDE (Holz et al., 1996, Sinclair et al., 1995) que, por ser uma metodologia de co-design de hardware e software, integra o método OMT(Rumbaugh et al., 1991) com as TDFs SDL para especificação de sistemas de software e VHDL (ANSI/IEEE, 1993) para sistemas de hardware, ARENA/SOMT (Auchter, 1997, Telelogic, 1996), que integra OMT com a uma combinação de SDL e MSC[9], e Metamorphosis (Araújo & Sawyer, 1998) - integra OMT a Object-Z. Apesar disso, nenhuma das metodologias apresenta um suporte

[9] *Message Sequence Charts* : linguagem formal, gráfica e textual, para a descrição e especificação do comportamento de interações entre componentes de um sistema. Pode ser usada para especificação de requisitos, simulação, validação e documentação de sistemas de tempo-real (ITU-T Z 120, 1994).

sistemático ao desenvolvimento para/com reuso, nem tão pouco aborda questões relativas aos problemas de domínio .

Desse modo, a fim de suprir os requisitos não atendidos pelos métodos atualmente existentes, este trabalho instancia o modelo genérico de desenvolvimento para/com reuso proposto no Capítulo 2, aplicando-o ao desenvolvimento de sistemas multimídia distribuídos. O modelo de processo instanciado é discutido detalhadamente na subseção 5.2.2 e utiliza de maneira complementar conceitos oriundos das áreas de métodos formais, engenharia de software orientada a objetos e sistemas baseados em conhecimento.

É importante ressaltar que, apesar deste modelo ter sido concebido a partir de necessidades identificadas na construção de sistemas multimídia distribuídos (vídeo sob demanda, sistemas de conferência multimídia, educação a distância, entre outros), acredita-se que os benefícios por ele oferecidos não são limitados a esta classe de sistemas.

5.2.2 - Um processo alternativo de desenvolvimento

Tradicionalmente, o modelo de processos da engenharia de software orientado a objetos é composto das seguintes fases: *Análise e Especificação de Requisitos*, *Construção* (*projeto* e *implementação*) e *Testes*. A fase de Análise e Especificação de Requisitos é construída com base no estudo dos requisitos conceituais necessários para o desenvolvimento do sistema. Nessa fase, o modelo de requisitos é desenvolvido usando mecanismos informais (linguagem textual) e semi-formais (modelo de objetos do domínio do problema, modelo de interação entre objetos), levando em conta a perspectiva do usuário, isto é, como o sistema irá oferecer o serviço aos seus potenciais clientes. O sistema é expresso em termos de seqüências de eventos com comportamentos relacionados, que encapsulam um serviço atômico oferecido ao usuário (os chamados *casos de uso*). Uma vez concluída, esta especificação representa uma versão estruturada da visão do cliente acerca do domínio. Neste mesmo estágio, a especificação é organizada em termos de objetos lógicos, os quais, então, serão adequados para o universo computacional e implementados na fase de construção.

Nessa seção, um modelo orientado a objetos para o desenvolvimento de sistemas multimídia distribuídos é instanciado a partir do modelo genérico proposto no Capítulo 2.

Esse modelo visa contemplar soluções tanto para os problemas de domínio, quanto para os problemas de projeto existentes nessa classe de sistemas, conforme discutido anteriormente (Guizzardi & Gonçalves, 1999c,d).

Primeiramente, para abordar os problemas de domínio, o modelo utiliza a abordagem de engenharia de domínio presente no modelo genérico. Para a realização dessa fase, será utilizada a metodologia proposta nos Capítulos 3 e 4. Esta abordagem permite a comunicação a respeito do domínio e a sua validação de maneira formal, além de incentivar a prática de reutilização em um nível de conhecimento. Por fim, vale salientar a importância do modelo de domínio produzido no auxílio à elicitação de requisitos específicos da aplicação.

Em segundo lugar, a fim de abordar os problemas de projeto, o modelo propõe uma extensão do processo tradicional, através da formalização de casos de uso selecionados. Na fase de projeto de um processo convencional de desenvolvimento OO, geralmente são empregados diagramas semi-formais para abordar os problemas descritos anteriormente. Estes diagramas podem ser de três tipos: (a) *diagramas de atividade* - para modelar atividades concorrentes; (b) *diagramas de transição de estado* - para modelar a mudança de estado de alguns objetos; (c) *diagramas de interação* - para modelar a troca de mensagens entre objetos (Fowler, 1997). Desta maneira, são identificadas as funcionalidades que devem ser oferecidas, isto é, as operações a serem implementadas por cada classe. No entanto, esses diagramas são limitados por sua natureza informal.

O modelo que será instanciado propõe que na fase de projeto, os casos de uso que apresentam a existência dos problemas citados anteriormente, sejam selecionados e descritos em uma combinação de SDL/MSC e que, portanto, possam ser devidamente formalizados. Além de impor um maior formalismo às relações dinâmicas envolvendo estes objetos, o uso de uma TDF baseada em um modelo de MEFEs[10], cujas interações acontecem através da troca de mensagens, permite, de maneira cômoda, agrupar as funcionalidades dos três tipos de diagramas citados (atividade, transição de estados e interação) em uma só especificação. Esta combinação de TDFs é escolhida por suportar diretamente os conceitos de orientação a objetos e por sintetizar de maneira conveniente as funcionalidades de todos esses diagramas geralmente empregados. É importante

[10] Máquinas de estados finitos estendidas

ressaltar que o emprego de métodos formais é proposto como uma alternativa à formalização dos casos de uso, empregando esta técnica somente naqueles casos em que ela se mostra necessária.

A figura 5.1 mostra as atividades de construção da fase do desenvolvimento com reuso desse processo. Os retângulos na figura representam as atividades, enquanto que os retângulos com bordas arredondadas representam os diversos modelos gerados por essas atividades. Como pode ser observado, na fase de projeto, dois caminhos alternativos podem ser seguidos, sendo a escolha feita baseada no grau de formalidade necessário ao tratamento de um caso de uso.

Um vez definido, este processo será utilizado nas seções seguintes para o desenvolvimento de uma ontologia de gerência de vídeo sob demanda, de um *framework* derivado dessa ontologia e, finalmente, de um aplicação de vídeo sob demanda.

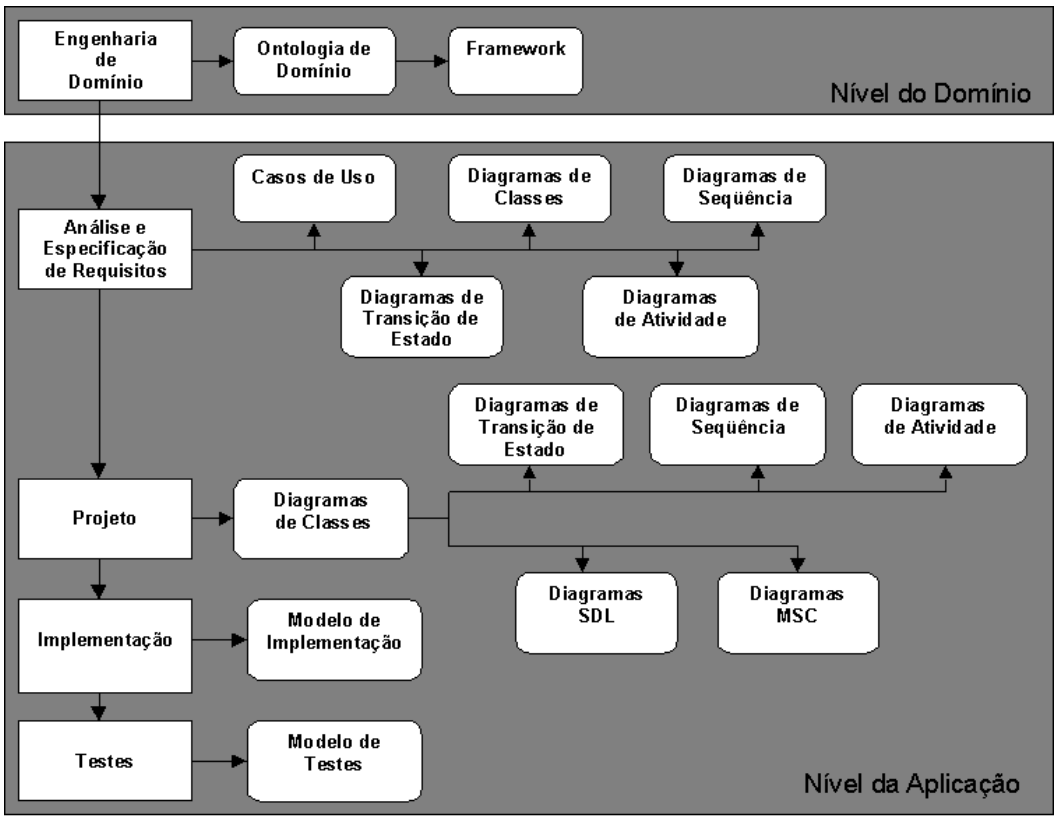


Figura 5.1 - Atividades de construção de um processo orientado a objetos para construção de sistemas multimídia distribuídos

Por fim, um aspecto importante da utilização conjunta de métodos OO e TDFs são as regras de transformação que permitem as transições entre esses dois paradigmas. Estas transformações ocorrem em duas etapas. Primeiramente, é necessário que se possa derivar de forma sistemática especificações formais em nível de projeto, para os diagramas de classes de análise dos casos de uso selecionados. Em segundo lugar, é também necessário que, a partir dessas especificações, possa ser gerado código em uma linguagem orientada a objetos (idealmente de forma automática).

Nesse trabalho, apenas para a primeira etapa de transformação foram utilizadas algumas diretivas propostas no projeto ESPRIT INSYDE. Nesse projeto, são criadas várias regras formais de transformação de modelos OO em diagramas SDL. O projeto apresenta uma notação intermediária entre as fases de análise e projeto - OMT* - que apresenta um subconjunto das expressões válidas de OMT, mas que contém uma semântica bem definida (Wosowski et al., 1996, Jonckers et al., 1995, Verschaeve et al., 1996).

Um outro método que aborda estas questões é o SOMT (Telelogic, 1996) que apresenta as diversas formas de interação entre modelos OO, diagramas SDL e MSC. No entanto, apesar dos trabalhos existentes, a definição da rastreabilidade (*traceability*) entre os diversos modelos produzidos nesse contexto não é uma tarefa trivial, podendo ser explorada em diversos trabalhos futuros.

5.3 - O Domínio de Vídeo sob Demanda

Diferentemente de um sistema de televisão tradicional, onde o telespectador é um agente passivo, que tem acesso ao serviço sem nenhuma interação com as informações a ele transmitidas, um sistema de vídeo sob demanda (*VoD – video on demand system*), por outro lado, é um sistema multimídia que oferece aos seus usuários a funcionalidade de acesso a um servidor remoto de vídeos (por exemplo, a partir de um terminal presente em casa ou escritório), atuando sobre o mesmo em vários níveis de interação e com ampla liberdade de escolha de programação. As conexões de vídeo são estabelecidas sob demanda, através de uma rede de comunicação que liga o cliente ao servidor de vídeo.

Sistemas de vídeo sob demanda têm um potencial enorme de aplicação e alguns protótipos vêm sendo desenvolvidos, em particular nas áreas de educação, medicina e entretenimento. Diversos estudos de comportamento feitos com usuários nos EUA mostram que o serviço de APPV (*Advanced Pay-Per-View*), que é um dos tipos de serviço de vídeo sob demanda, será o serviço mais utilizado nos próximos anos, sendo no futuro superado pelos serviços de jogos interativos sob demanda e o verdadeiro serviço de vídeo sob demanda, ambos sempre se apresentando com um potencial de mercado muito grande (Cecilio & Rodrigues, 1996b) (figura 5.2).

Figura 5.2- Evolução da Demanda dos Serviços de TV Interativa (Cecilio & Rodrigues, 1996b)

Os sistemas de VoD podem ser classificados como a seguir, de acordo com seu grau de interatividade (Little & Venkatesh, 1994):

1 **Broadcast (No VoD):** Sistema de difusão de TV sobre o qual o usuário não tem nenhuma interação. Dessa forma, não constitui na verdade um sistema de VoD, sendo classificado apenas em caráter de comparação.

2 **PPV (Pay-per-view):** Sistema similar ao que existe hoje na TV por assinatura, em que o usuário escolhe e paga por uma determinada programação que deseja assistir (similar aos serviços PPV atualmente existentes nos ambientes de CATV e DBS - *Direct Broadcast Satellite*.

3 **QVod (Quasi VoD):** Esta modalidade é baseada em segmentos temporais, múltiplos da duração do filme. Cada usuário tem direito a escolher o vídeo que deseja assistir. Imediatamente após a escolha, o sistema tenta agrupar dentro de um mesmo segmento

temporal todos os usuários que demandam por um mesmo vídeo. Se no exato momento em que o usuário escolhe um vídeo para assistir a exibição deste filme estiver começando, o usuário irá recebê-lo imediatamente, caso contrário será obrigado a esperar até o início do próximo segmento.

4 **NVoD (Near VoD):** Este caso se apresenta como uma extensão do QVoD, no que diz respeito ao poder de controle disponibilizado para o usuário. Apesar de ainda não possuir uma transmissão exclusiva para cada usuário, possibilita algum controle sobre a exibição do vídeo. Porém, esta interação está associada aos segmentos temporais, no sentido de que o usuário só pode retroceder, adiantar ou fazer uma pausa no vídeo em intervalos temporais múltiplos destes segmentos. É importante ressaltar a preocupação que deve ser tomada na escolha do segmento temporal, pois o que temos aqui é uma troca entre um maior nível de interação (segmentos menores) e uma maior economia de largura de banda (segmentos maiores).

5 **TVoD (True VoD):** Neste caso, o usuário tem acesso a todas as funções de um videocassete virtual, podendo então exercer o controle total sobre a execução do vídeo. É como se existisse um canal exclusivo para cada usuário, pois mesmo que dois usuários decidam assistir a um mesmo filme simultaneamente, muito provavelmente os dois irão interagir com o mesmo de formas diferentes.

O domínio modelado neste trabalho é um serviço de TVoD. Apesar disso, é importante ser dito que apenas a nomenclatura vídeo sob demanda ou simplesmente VoD será usada daqui em diante, com o intuito de endereçar esta classe específica de sistemas de vídeo sob demanda.

Para que um serviço de vídeo sob demanda possa ser oferecido, vários são os requisitos tecnológicos necessários. Alguns deles exercem uma influência direta nas possibilidades de concretização dos subsistemas que compõem a arquitetura do serviço, nos seus modelos de distribuição e, em última instância, na ontologia de VoD que será produzida. A seguir, são discutidos os principais requisitos deste tipo e as alternativas existentes para endereça-los, tanto do ponto de vista tecnológico quanto econômico. Por fim, é interessante observar a influência que requisitos dessa natureza têm na proposição de arquiteturas conceituais.

5.3.1 - Requisitos Tecnológicos

A Figura 5.3 mostra uma ilustração simplificada de como um serviço de vídeo sob demanda é visto do ponto de vista do usuário. O cenário é composto de três subsistemas: o terminal através do qual o usuário tem acesso ao serviço, o servidor de vídeos digitalizados e o sistema de comunicação responsável por fazer a conexão entre os dois.

Figura 5.3 - Visão simplificada de um cenário de VoD

Os principais requisitos tecnológicos impostos a um sistema desse tipo estão ligados ao seu comprometimento com o transporte, armazenamento e apresentação de mídias contínuas, como áudio e vídeo, e dessa forma, exercem diretamente um impacto em todos os subsistemas citados. Esses tipos de mídia são notórios pela alta demanda de desempenho que impõem ao sistema, principalmente no que diz respeito ao espaço de armazenamento e largura de banda de transmissão requeridos e pela validade temporal de seus dados. Para melhor exemplificar esta necessidade, o seguinte exemplo será usado:

1. Suponha um *frame* de vídeo com resolução de 640x480 pixels e com profundidade de 24 bits. Este frame sozinho ocupa um espaço de 640x480x3 bytes = 900 Kbytes.

2. Considerando uma taxa de *frames* de 30 frames/s (padrão NTSC), então um segundo deste vídeo necessita de uma largura de banda de transmissão de aproximadamente 27 Mbytes/s.

3. Em média, um filme tem duração de 90 minutos (1 1/2 hora), portanto, um único filme requer, em média, 143 GB de espaço de armazenamento.

4. Agora suponha que este vídeo possui também um áudio *stereo* associado, com qualidade de CD (16 bits/amostra e taxa de amostragem de 44100 Hz). Um segundo deste áudio ocupa aproximadamente 88 KB e, portanto, o áudio de um único filme de 90 minutos ocupa cerca de 475 MB de espaço de armazenamento.

Por estas razões proibitivas, o desenvolvimento de bons algoritmos de compressão específicos para estes sistemas tornou-se crucial para o desenvolvimento de aplicações multimídia distribuídas em geral. Os principais formatos usados para codificação de mídias contínuas são os padrões ITU-T H.263 e ISO MPEG - Motion Pictures Expert Group (Saywood, 1996), sendo esse último o recomendado pelo DAVIC(Digital..., 1998)[11] para aplicações de VoD. Taxas de compressão típicas alcançadas pelo MPEG são de 100:1 para vídeo e 10:1 para áudio. Apesar da redução significativa alcançada, a largura de banda de transmissão requerida ainda é grande o bastante para influenciar diretamente na escolha das alternativas de implementação dos subsistemas que compõem o serviço. Os padrões MPEG atualmente existentes para codificação de vídeo são: MPEG-1, MPEG-2 e MPEG-4 (o padrão MPEG-3 que seria usado nas aplicações de HDTV [12]foi descartado pelo grupo). A tabela 5.1 faz uma comparação entre estes padrões, considerando a qualidade suportada e a taxa de compressão de vídeo endereçada (Rowe et al.,1994).

[11] DAVIC: Digital Audio Visual Council – associação sem fins lucrativos, contando com 222 empresas em 25 países, comprometida com a padronização da interoperabilidade fim-a-fim de aplicações audiovisuais interativas.
[12] High Definition Television (Cecilio & Rodrigues, 1996a)

Padrão MPEG	*Qualidade suportada*	*Taxa de bits (MB/s)*
MPEG-1	**Super-VHS**	**1.5 a 2.0**
MPEG-2	**TV e HDTV**	**2.0 a 6.0**
MPEG-4	**Vídeofone**	**0.0048 a 0.064**

Tabela 5.1 - Comparação entre os padrões MPEG existentes

5.3.1.1 - Subsistema de Apresentação e Interação

O terminal através do qual o usuário tem acesso ao serviço deve ser capaz de receber, descompactar, decodificar e apresentar o fluxo contínuo de vídeo vindo do sistema de comunicação. Outra função do terminal é a de enviar comandos de interação ao servidor de vídeos remoto, de acordo com as instruções de um usuário humano, a fim de controlar a exibição do vídeo. O dispositivo que aparece na figura 5.3 ligado ao terminal é chamado de *set-top box (STB)* e é citado desde as idéias preliminares de concepção deste serviço como responsável pelo conjunto de requisitos funcionais ligados ao terminal. Várias são as possibilidades de implementação de um STB. Tradicionalmente, eles são implementados como dispositivos de hardware, no entanto, atualmente, com o crescimento do poder computacional disponível do lado do cliente, implementações híbridas ou puramente de software têm se tornado viáveis (Rowe et al.,1994).

5.3.1.2 - Subsistema de Comunicação

A rede pela qual o usuário tem acesso ao serviço é chamada de rede de acesso local (RAL). Ao contrário de conexões típicas usadas no ambiente tradicional de computação, caracterizadas por curtos períodos de duração e tráfego em rajadas, o esquema usado para a ligação do usuário ao provedor do serviço deve suportar conexões de algumas horas de duração, tráfego contínuo de dados e grande largura de banda. As alternativas de implementação mais viáveis do ponto de vista tecnológico e econômico são as abordagens que utilizam a rede pública de telefonia, devido à alta capilaridade alcançada por esta rede e, conseqüentemente, a possibilidade de grande cobertura e baixo

custo do serviço. Entre as principais propostas existentes atualmente está a família de tecnologias ADSL (*Asymetrical Subscriber Digital Line*), que oferecem tipicamente taxas entre 1.544 e 6 Mbps para *downstream* (do provedor para o usuário) e entre 128 e 384 Kbps no sentido inverso. Porém, essas taxas são alcançadas somente em um raio de distância fixo, geralmente em torno de 5,5 Kilômetros da fonte de transmissão.

5.3.1.3 - Subsistema de Transmissão e Armazenamento

Por fim, o sistema de armazenamento tem como responsabilidade cuidar do espaço necessário para armazenar a programação em formato comprimido, além da transmissão do fluxo de vídeo, quando requisitado para exibição. O servidor de vídeo deve ser capaz também de responder aos comandos de interação do usuário, a fim de prover a ele o controle da sessão de exibição do vídeo, implementando as funções de um videocassete virtual (iniciar, interromper, dar uma pausa, adiantar, voltar). Para atender a muitos usuários e oferecer uma grande variedade de informações, o sistema de armazenamento do servidor de vídeos deve possuir alta capacidade, além de suportar o acesso concorrente de vários usuários simultaneamente. Para o projeto de um servidor de vídeos, várias considerações devem ser tomadas:

- Mesmo comprimido a uma taxa de 100:1 um único filme comercial ainda ocupa cerca de 1.45 GB de espaço de armazenamento (sem o áudio correspondente). Supondo que o servidor possua 500 filmes disponíveis, seria preciso um espaço de armazenamento de aproximadamente 725 Gbytes.
- O número de sessões interativas que um servidor pode suportar é proporcional, entre outras coisas, à taxa de transferência do seu sistema de entrada/saída. Mesmo com tecnologias como SCSI-2 e FC-AL que podem alcançar taxas de 100 MB/s, um servidor só pode atender a um número de sessões simultâneas relativamente limitado para um serviço comercial de larga escala.
- Um fato conhecido através da experiência com outros serviços sob demanda (locadoras de vídeo, bibliotecas públicas) é que nem todos os itens disponíveis em um catálogo são igualmente populares. De acordo com a “lei de Zipf”(Chervenak, 1994), experimentalmente, quando existem N filmes disponíveis, o percentual de todas as

requisições para o k-ésimo filme mais popular é de aproximadamente C/k, onde C é computado para normalizar a soma em 1.

$$^{C}/_{1} + {}^{C}/_{2} + {}^{C}/_{3} + {}^{C}/_{4} + \ldots + {}^{C}/_{N} = 1$$

$$C = 1 / (1 + {}^{1}/_{2} + {}^{1}/_{3} + {}^{1}/_{4} + \ldots + {}^{1}/_{N})$$

Com isso, o filme mais popular é sete vezes mais requisitado que o sétimo filme mais popular.

Com estas considerações, podemos identificar duas características desejáveis que devem ser contempladas de alguma forma por nossa arquitetura. São elas:

(a) Estrutura Hierárquica: É extremamente custoso para o provedor que todos os vídeos que os seus usuários possam querer assistir estejam armazenados em seus servidores. Portanto, os servidores de um provedor devem armazenar apenas aqueles filmes mais populares entre o público alvo daquele provedor (por exemplo, 50 ou 100). Os outros filmes devem estar armazenados em um outro dispositivo que o provedor tenha acesso, podendo ser um dispositivo de memória secundária (como por exemplo, uma fita), ou ainda um outro servidor pertencente a outro provedor.

(b) Flexibilidade: É previsível que durante a vida de um provedor, vários servidores venham a ser inseridos em seu domínio. Desta forma, a introdução de um novo servidor neste esquema não deve ser custosa a ponto de afetar a operação normal do sistema.

As exigências de flexibilidade impostas pelos subsistemas de armazenamento e comunicação terão um grande impacto tanto nos modelos de distribuição do serviço (como será visto na próxima subseção), quanto nos mecanismos de gerência do sistema. A fim de prover escalonabilidade, adaptatividade e robustez, o mecanismo de gerência implementado deverá controlar de forma eficiente todas as relações e restrições envolvendo os agentes pertencentes a esse cenário - vídeos, servidores, provedores, terminais e centrais.

5.3.2 - Configurações de Distribuição do Serviço

Em qualquer sistema de distribuição de conteúdo de massa, uma das principais preocupações é como este conteúdo será distribuído. As duas principais alternativas de configuração são (Sampson et al., 1997):

1. Um sistema centralizado, com poucos servidores de altíssima capacidade de armazenamento e capacidade para atender a um número enorme de sessões simultâneas;
2. Um sistema descentralizado, com muitos servidores estrategicamente bem localizados.

A fim de caracterizar essas configurações de distribuição, deve-se fazer a distinção entre duas sub-redes presentes no contexto: a Rede de Acesso Local (*RAL)* e a Rede de Acesso Remoto *(RAR*). A *RAL* é a sub-rede na qual o terminal do usuário está conectado, enquanto que a *RAR* conecta servidores distribuídos geograficamente. Em um ambiente de VoD comercial, a *RAL* poderia ser um canal *ADSL,* ao qual o terminal do usuário teria acesso, e a *RAR* poderia ser uma rede ATM de alta velocidade, interligando os servidores de vídeo. Deste modo, podemos observar que a *RAL* se apresenta como uma alternativa de estrutura mais simples e lenta que a *RAR*, mas também mais acessível ao usuário final, o que é interessante em um sistema comercial. Já em um ambiente educacional, a *RAL* pode ser uma LAN pertencente a um departamento e a *RAR* uma WAN conectando departamentos ou universidades.

5.3.2.1 - Distribuição Centralizada

Em uma configuração centralizada (Figura 5.4), temos um número limitado de enormes servidores de vídeo (*video warehouses*) ligados às centrais telefônicas através de uma rede de alta velocidade (consideraremos aqui esta rede como sendo uma rede ATM). Neste cenário, a central telefônica possui tanto o equipamento de terminação da rede ATM, quanto *buffers* locais para converter o tráfego ATM em rajadas, para um fluxo contínuo de dados até o terminal do usuário. No futuro, com uma possível evolução e queda de preço da tecnologia ATM, o equipamento de terminação ATM poderá ser movido para a casa do usuário.

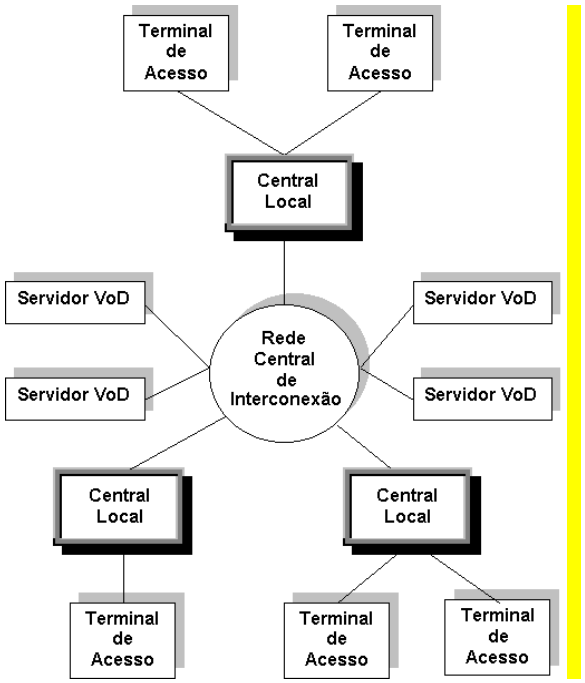


Figura 5.4 - Configuração de distribuição centralizada

Uma clara desvantagem da abordagem centralizada em um serviço inerentemente interativo como o que queremos modelar é que o comportamento de interação de cada usuário demanda trocas automáticas do conteúdo dos *buffers* (tanto os *buffers* locais, quanto os presentes nos nós da rede central), que vão interferir diretamente no tráfego da rede de interconexão. Neste cenário, o mercado de vídeo sob demanda seria dominado por grandes cadeias de comunicação com a participação direta das empresas de telefonia.

5.3.2.2 - Distribuição Descentralizada

Na abordagem descentralizada (figura 5.5), existem servidores menores, com capacidade de armazenamento de, por exemplo, 50 a 100 vídeos, ligados diretamente às centrais telefônicas locais. Desta forma, o usuário acessa diretamente os servidores locais através da rede de acesso local. Estes servidores podem ser *stand-alone* ou podem estar conectados via rede de alta velocidade a outros servidores presentes em outras centrais telefônicas.

Caso um usuário opte por um vídeo que não esteja presente em um servidor local, então este vídeo pode ser copiado via rede de alta velocidade de um servidor remoto qualquer, seguindo políticas de gerência estabelecidas para o serviço. A fim de minimizar o tráfego na rede que conecta as centrais telefônicas, uma política de escalonamento deve ser adotada, premiando os usuários que optem por fazer esta cópia em um horário de baixo tráfego na rede.

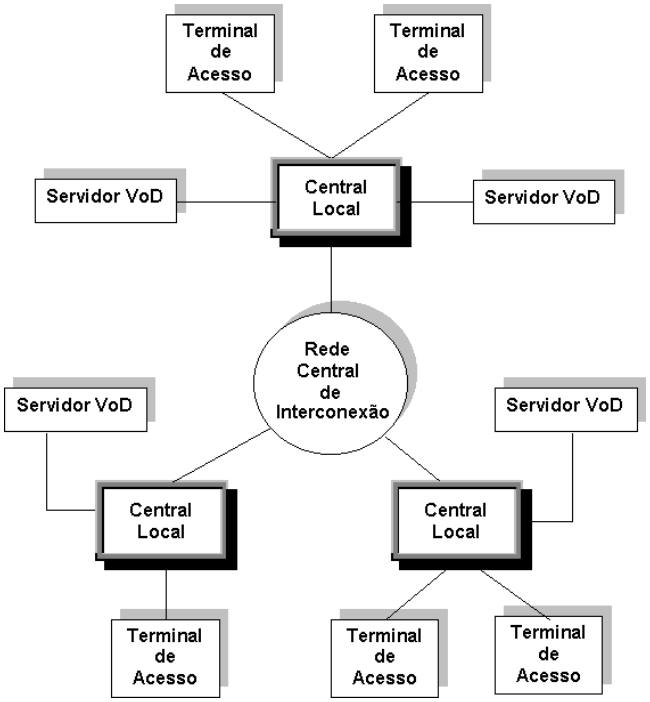


Figura 5.5 - Configuração de distribuição descentralizada

5.3.2.3 - Comparação Entre as Duas Abordagens

Além da já citada desvantagem da abordagem centralizada, no que diz respeito à influência que o comportamento interativo de usuários individuais tem sobre o tráfego da rede, acreditamos que existem várias outras vantagens da abordagem descentralizada. As principais são:

- As empresas de telefonia não têm necessariamente participação direta como provedoras do serviço de VoD, podendo participar apenas com o oferecimento das funcionalidades das centrais locais para outras empresas que serão, por sua vez, as provedoras do serviço para o usuário final;

- Os provedores podem começar com um serviço em pequena escala (servidores *stand-alone*), que crescerá de acordo com a demanda, possibilitando um investimento inicial mais baixo;
- Possibilidade de adequação a públicos locais, no sentido de prover uma programação (e também propagandas) direcionadas para um público alvo específico;
- Taxas de transmissão mais baixas;
- Maior separação dos recursos de aplicação e redes;
- Maior flexibilidade de configuração.

5.3.3 - Os Níveis de Gerência e de Sistema

Independentemente da configuração de distribuição adotada, fica claro que o subsistema de comunicação será considerado um serviço à parte, muito provavelmente oferecido por um provedor diferente daquele do serviço de VoD propriamente dito. Desta forma, do ponto de vista deste trabalho, o desenvolvimento de aplicações de vídeo sob demanda deve estar comprometido apenas com os subsistemas de Apresentação/Interação e Armazenamento/Transmissão. A infra-estrutura de comunicação é vista como uma *caixa-preta* capaz de encapsular as funcionalidades requeridas. Além disso, o conjunto de responsabilidades atribuídas à interação entre os subsistemas abordados será distribuída entre dois níveis funcionais: ***o nível de gerência e o nível de sistema***. Os três principais sistemas funcionais participantes dessa arquitetura - Terminal do usuário, Servidor de Gerência e Servidor de Sistema - receberão nessa seção a mesma nomenclatura que em (Eilley et al., 1994).

No nível de gerência, o terminal do usuário - chamado de *EUT (End-User Terminal)* - interage com um servidor funcional chamado de *AMS* (*Application Management System*). Esse servidor é uma espécie de gerente dos agentes envolvidos no serviço (provedores, servidores, terminais, centrais e vídeos). Os usuários inicialmente se conectam a um AMS para escolher o serviço que vão usar e o vídeo que vão assistir. Para isso, o AMS se comunica com um banco de meta-dados que contém os catálogos de todos os servidores gerenciados por ele. Uma vez escolhido o vídeo, o AMS conecta o usuário ao servidor de vídeo correspondente que faz a distribuição do vídeo. O AMS é responsável por todas as funções de gerência dos servidores de vídeo, como controle de acesso, *marketing*, controle comercial e serviços de nomeação, além de saber responder a questões como que servidor pertence a que provedor, que servidor oferece qual serviço e que servidor armazena qual vídeo, além de outras coisas. O AMS também arquiva informação sobre todos os vídeos assistidos e registra os dados estatístico para propósitos futuros.

Uma vez escolhido, o vídeo a ser assistido, o EUT passa a interagir com outro servidor, chamado de *VoDS (Video on Demand Server)*. Juntos eles cooperam na realização de funções do chamado nível de sistema, como por exemplo sincronização e transporte de mídias contínuas, decodificação e apresentação de mídias em vários formatos e provimento das funções de interação do usuário, implementando as funcionalidades de videocassete virtual.

É importante, ainda, salientar que esta abordagem contribui diretamente para se conseguir uma arquitetura robusta no que diz respeito à escalonabilidade e à flexibilidade. Algumas de suas principais vantagens em relação a termos um único tipo de servidor realizando ambas as tarefas são:

1. O AMS funciona como um gerente de provedores (e, conseqüentemente, dos seus VoDS), encapsulando em um único subsistema todos aqueles serviços não relativos à exibição do vídeo propriamente dita e que são comuns a todos os provedores, como serviços de concretização dos requisitos relacionais (que provedor possui qual servidor, que servidor armazena qual vídeo, que cliente contrata qual provedor), segurança e controle de acesso, *marketing*, entre outros.

2. Com o AMS funcionando como servidor no nível de gerência para vários VoDS, seria extremamente fácil a inclusão, por exemplo, de um novo VoDS no sistema.
3. Ao encapsular todas as funções do nível de gerência, o AMS livra o usuário final da preocupação de saber que VoDS armazena que vídeo e da preocupação de saber os parâmetros necessários para se conectar a cada servidor separadamente.

Por fim, é importante salientar como a existência desses dois níveis ressalta a adequação deste estudo de caso para a experimentação da metodologia proposta. O nível de gerência deve encapsular restrições complexas de relacionamento entre os diversos agentes que atuam no sistema e a análise e abstração de suas necessidades será a base para a construção de uma ontologia de gerência de vídeo sob demanda na próxima seção. O nível de sistema, por outro lado, encapsula restrições complexas de problemas de projeto comuns aos sistemas multimídia distribuídos. Para endereçar esses requisitos de projeto, na seção 5.5, é utilizado um *framework* chamado *JMF (Java Media Framework)*. Desta forma, as duas próximas seções apresentam, respectivamente, exemplos de desenvolvimento para e com reuso no contexto de um mesmo sistema.

5.4 - Aplicação da metodologia ao Domínio de Estudo

A seção anterior discutiu de forma detalhada o domínio de Vídeo sob Demanda. Primeiramente, foram identificados os requisitos tecnológicos necessários para que o provimento do serviço possa ser realizado. Em segundo lugar, foram analisadas as influências que esses requisitos têm sobre os modelos de distribuição do serviço e sobre as alternativas de implementação. Finalmente, foi apresentada uma proposta de divisão funcional do serviço de VoD em níveis de gerência e sistema. Essa seção tem como objetivo especificar uma ontologia de VoD comprometida com esse nível de gerência. Uma vez especificada, essa ontologia será submetida à metodologia proposta nos capítulos anteriores, a fim de gerar uma infra-estrutura reutilizável desse domínio.

Ao fim desse processo de aquisição e análise de dados, foram identificados os principais agentes participantes desse cenário, as relações existentes entre eles, as pré-condições que devem ser cumpridas para que possam se relacionar e as principais questões de competência que devem ser respondidas. É importante salientar que, por se tratar de uma ontologia, a representação do domínio produzida deve ser a mais genérica

possível, capturando somente os elementos comuns aos diversos membros de uma família de aplicações.

A Figura 5.6 apresenta um modelo em LINGO que identifica os *conceitos* e *relações* nesse cenário (Guizzardi & Gonçalves, 1998). Em seguida é apresentada a descrição da intenção de cada um dos conceitos. Algumas dessas descrições expõem a intenção das relações em que estão envolvidos, sendo que para aquelas relações em que isso não acontece, a descrição de suas intenções também é dada.

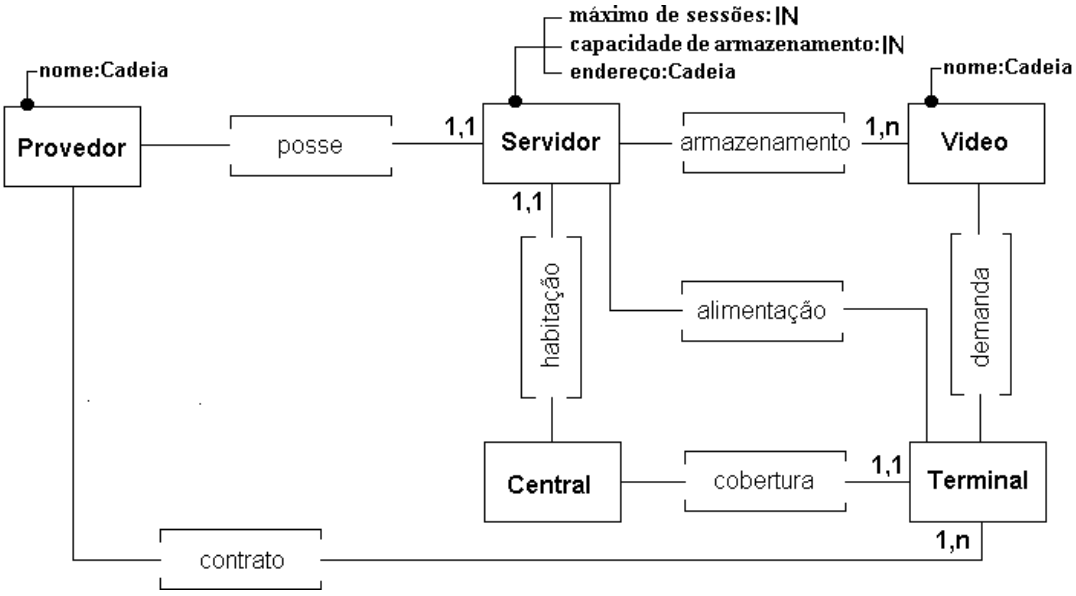


Figura 5.6 - Diagrama conceitos e relações do domínio de gerência de um serviço de VoD

5.4.1 - Descrição dos Conceitos

- ***Terminal*:** O *Terminal* representa o equipamento através do qual o usuário final tem acesso ao sistema. É através dele que o usuário escolhe o *Vídeo* que irá assistir e, uma vez escolhido, interage com a exibição desse *Vídeo*. Independentemente de como será materializado, um *Terminal* deve ser capaz de receber e apresentar o fluxo contínuo de *Vídeo* proveniente de sua comunicação com o *Servidor*, sendo compatível com o formato de codificação utilizado por ele. Se necessário, deve ser capaz de realizar as funções de decodificação e descompactação dos dados recebidos. Por fim, com o objetivo de prover as funcionalidades de um videocassete virtual (pausar, adiantar,

retroceder, parar), ele deve ser capaz de interagir com este *Servidor*, enviando-lhe os resultados dos comandos de interação do usuário.

- ***Servidor*:** Armazena os *Vídeos* e os transmite aos *Terminais*, em um formato suportado por qualquer *Terminal* participante. Deve suportar o acesso aleatório a posições dos *Vídeos* armazenados, com o objetivo de responder aos comandos de interação de um ou mais *Terminais* simultaneamente. Cada *Servidor* é identificado através de um sistema de endereçamento, próprio da tecnologia de comunicação empregada. Além disso, cada *Servidor* suporta um número máximo de *Vídeos* armazenados e de sessões de transmissão. No entanto, é importante salientar que essa descrição não define como os *Vídeos* são armazenados ou transmitidos, nem tampouco quantos deles podem ser armazenados ou transmitidos simultaneamente, sendo essas decisões tomadas por cada uma das aplicações específicas.

- ***Vídeo*:** É, provavelmente, o principal conceito neste domínio. Um *Vídeo* diz respeito a uma entidade que identifica univocamente um título que compõe o catálogo de um *Provedor*. Um *Vídeo*, portanto, existe no escopo de um *Provedor*. O conceito não referencia diretamente o material digitalizado, codificado em um formato específico e armazenado nos *Servidores*. Um mesmo *Vídeo* pode possuir várias cópias (padrões de magnetismo em um meio de armazenamento) em *Servidores* diferentes, desde que esses *Servidores* pertençam a um mesmo *Provedor*, ou de maneira mais formal

$$\forall v{:}V,\ s_1,s_2{:}S\ \text{armazenamento}(v, s_1) \wedge \text{armazenamento}(v, s_2) \rightarrow$$
$$\text{provedor}(s_1) = \text{provedor}(s_2)$$

- ***Provedor*:** o *Provedor* é uma entidade que possui *Servidores* e é contratada por *Terminais*, sendo responsável por manter os catálogos de *Vídeo* em seus *Servidores* e por gerenciar os usuários que os contratam.

- ***Central*:** Essa é a única entidade que aparece no modelo devido aos requisitos tecnológicos necessários para o oferecimento do serviço. Apesar de ser uma entidade cujo controle está fora do alcance dos provedores do serviço, essa entidade é modelada devido a sua importância do ponto de vista lógico (como será observado na descrição das relações em que participa). Sem a sua presença, o modelo se tornaria

inadequadamente livre, no sentido de que qualquer *Servidor* poderia transmitir uma sessão de exibição para qualquer *Terminal* - o que não é verdade em muitos dos casos. Portanto, essa entidade desempenha o papel de uma central telefônica, central de TV a cabo (CATV) ou, ainda, uma rede local (LAN) no cenário educacional, associando *Servidores* e grupos de *Terminais* mutuamente excludentes.

A extensão desses conceitos é representada pelos seguintes conjuntos:

(D1) P = Provedor
(D2) S = Servidor
(D3) C = Central
(D4) V = Vídeo
(D5) T= Terminal

5.4.2 - Descrição das Relações

- ***cobertura:*** Enfatiza a necessidade que deve existir, em um sistema de comunicação, entre *Terminal* e *Central* para que esse possa assistir a *Vídeos* transmitidos por um *Servidor*. Esse sistema de comunicação deve garantir a qualidade e confiabilidade do sinal de *Vídeo* recebido.

- ***habitação:*** Um *Servidor* deve possuir uma conexão com uma *Central* para que possa transmitir *Vídeos* para os *Terminais* cobertos por ela. O *Servidor* pode estar localizado fisicamente ou não na *Central*, desde que exista entre eles uma conexão que garanta que o sinal de *Vídeo* enviado seja de qualidade e confiabilidade suficientes para que possa ser interpretado pelos *Terminais*. A escolha de que *Servidor* habitará que *Central* e de que maneira, é resultante de decisões do ponto de vista de negócio realizada pelos *Provedores* que os possuem e, portanto, sua discussão foge ao escopo dessa definição.

- ***alimentação*:** Esta relação encapsula um conjunto de pré-condições que devem ser satisfeitas para que um *Terminal* possa ser alimentado por um *Servidor*, ou seja, para que uma sessão de exibição de *Vídeo* possa ser firmada entre duas instâncias desses conceitos. A primeira delas diz que, para um *Servidor* alimentar um *Terminal*, ele deve pertencer a um dos *Provedores* contratados por esse *Terminal*, ou seja,

□s:S,□□t:T alimentação (t,s) →□s ∈ (Im(Im(t,contrato),posse))

Além disso, o *Servidor* deve estar conectado à mesma *Central* que cobre o *Terminal* para que os requisitos de qualidade necessários ao oferecimento do serviço (ex. largura de banda disponível) possam ser endereçados. De maneira mais formal, temos:

□s:S,□□t:T alimentação(t,s) →□□c:C habitação(s,c) ∧ cobertura(c,t)

ou de forma equivalente,

□s:S,□□t:T alimentação(t,s) →□ s ∈ (Im(Im(t,cobertura),habitação))

Esses axiomas de pré-condições podem ser compostos em um único axioma ontológico que responde a seguinte questão de competência: *dado um Terminal, que Servidores são capazes de alimentá-lo?*

∀ t:T,s:S alimentação(t,s) ↔

s ∈ (Im(Im(t,contrato),posse) ∩ Im(Im(t,cobertura), habitação))

Por fim, essa questão pode ser também vista do ponto de vista do Servidor: d*ado um Servidor, que Terminais são alimentados* por ele*?*

∀s:S, t:T alimentação(s,t) ↔

t ∈ (Im(Im(s,posse),contrato) ∩ Im(Im(s,habitação),cobertura))

- ***demanda*:** Essa relação é definida em função de outras relações e é responsável por responder à principal questão de competência da ontologia: *Que Vídeos podem ser assistidos por um dado Terminal?* A resposta a essa questão é dada pelo axioma abaixo:

∀t:T, v:V demanda(t,v) ↔ v ∈ Im(Im(t,alimentação),armazenamento)

Ou seja, o catálogo de *Vídeos* disponível a um *Terminal* é composto pelo conjunto de *Vídeos* armazenados nos *Servidores* que alimentam aquele *Terminal*.

A seguir, são apresentados, para todas as relações e propriedades, os axiomas de definição, e de restrição de suas cardinalidades:

(D6) posse = (Provedor, Servidor, posse(p,s))

(D7) contrato = (Provedor, Terminal, contrato(p,t))

(D8) armazenamento = (Servidor, Vídeo, armazenamento(s,v))

(D9) habitação = (Servidor, Central, habitação(s,c))

(D10) cobertura = (Central, Terminal, cobertura(c,t))

(AC1) $\forall s{:}S\ \#Im(s, posse) = 1$

(AC2) $\forall s{:}S\ \#Im(s, habitação) = 1$

(AC3) $\forall t{:}T\ \#Im(t, contrato) \geq 1$

(AC4) $\forall t{:}T\ \#Im(t, cobertura) = 1$

(AC5) $\forall v{:}V\ \#Im(v, armazenamento) \geq 1$

(AP1) $\forall s{:}S\ \exists!\underline{e}{:}$Cadeia endereço($\underline{e}$,s)

(AP2) $\forall s{:}S\ \exists!\underline{c}{:}$IN capacidade de armazenamento($\underline{c}$,s)

(AP3) $\forall s{:}S\ \exists!\underline{m}{:}$IN máximo de sessões($\underline{m}$,s)

(AP4) $\forall v{:}V\ \exists!\underline{n}{:}$Cadeia nome($\underline{n}$,v)

(AP5) $\forall p{:}P\ \exists!\underline{n}{:}$Cadeia nome($\underline{n}$,p)

Ao tornar explícito o conhecimento inerente a um domínio de interesse, uma ontologia permite a compreensão e a comunicação a respeito desse domínio. Por possuir uma descrição rigorosa, feita em uma linguagem matemática, seus axiomas podem ser formalmente simulados e validados. Além disso, por se tratar de uma estrutura de representação em um nível de conhecimento, uma ontologia se torna um componente altamente confiável e com grande potencial de reutilização. Por fim, através de uma metodologia sistemática de projeto, a partir dela, uma infra-estrutura de domínio pode ser gerada.

Nas seções seguintes, os axiomas que definem restrições e que derivam conhecimento nesse modelo de domínio são formalmente especificados. Em seguida, através da aplicação do conjunto de diretivas, regras de transformação e padrões de projeto anteriormente discutidos, um *framework* de gerência de Vídeo sob Demanda é gerado a partir da ontologia. Os conceitos e relações, com suas respectivas características e restrições de cardinalidade, são mapeados em classes, relacionamentos, atributos e parâmetros embutidos nos construtores das classes adequadas. Além disso, os axiomas

ontológicos e de consolidação derivam, respectivamente, invariantes e pré-condições, incorporados aos métodos do *framework*. O resultado desse processo é mostrado na figura 5.7.

5.4.3-Axiomas ontológicos

(AO1) ∀s:S, t:T alimentação(s,t) ↔

t ∈ (Im(Im(s,posse),contrato) ∩ Im(Im(s,habitação),cobertura))

(AO2) ∀ t:T,s:S alimentação(t,s) ↔

s ∈ (Im(Im(t,contrato),posse) ∩ Im(Im(t,cobertura), habitação))

(AO3) ∀t:T, v:V demanda(t,v) ↔ v ∈ Im(Im(t,alimentação),armazenamento)

A seguir, as regras de transformação definidas no capítulo anterior são aplicadas a esses axiomas, a fim de promover a geração dos métodos do *framework* que responderão as questões de competência levantadas anteriormente.

(AO1) ∀s:S, t:T alimentação(s,t) ↔

t ∈ (Im(Im(s,posse),contrato) ∩ Im(Im(s,habitação),cobertura))

1. alimentação(s,t) ≡ (Im(Im(s,posse),contrato) ∩ Im(Im(s,habitação),cobertura)) A01, T0
2. s.alimentacao():Set ≡ (Im(s.posse(),contrato) ∩ Im(s.habitação(),cobertura)) 1, T2
3. 3. s.alimentacao():Set ≡ Set.Im(s.posse(),"contrato") ∩ Set.Im(s.habitação(),"cobertura") 2, T5
4. public class Servidor 3, T7

```
{
    public Set alimentacao()
    {
        Set aux = Set.Im(this.posse(),"contrato");
        return aux.intersection(Set.Im(this.habitação(),"cobertura"));
    }
}
```

(AO2) ∀ t:T,s:S alimentação(t,s) ↔

s ∈ (Im(Im(t,contrato),posse) ∩ Im(Im(t,cobertura),habitação))

1. alimentação(t,s) ≡ (Im(Im(t,contrato),posse) ∩ Im(Im(t,cobertura), habitação)) A01, T0
2. t.alimentacao():Set ≡ (Im(t.contrato(),posse) ∩ Im(t.cobertura(),habitação)) 1, T2
3. t.alimentacao():Set ≡ Set.Im(t.contrato(),"posse") ∩ Set.Im(t.cobertura(),"habitacao") 2, T5
4. public class Terminal 3, T7

```
{
    public Set alimentacao()
    {
        Set aux = Set.Im(this.contrato(),"posse");
        return aux.intersection(Set.Im(this.cobertura(),"habitacao"));
    }
}
```

(AO3) ∀t:T, v:V demanda(t,v) ↔ v ∈ Im(Im(t,alimentação),armazenamento)

1. demanda(t,v) ≡ Im(Im(t,alimentação),armazenamento) A01, T0
2. t.demanda():Set ≡ Im(t.alimentação(),armazenamento) 1, T2
3. t.demanda():Set ≡ Set.Im(t.alimentação(),"armazenamento") 2, T5
4. public class Terminal 3, T7

```
{
    public Set demanda()
    {
        return Set.Im(this.alimentação(),"armazenamento");
    }
}
```

5.4.4-Axioma de consolidação

(AR1) $\forall v{:}V, s_1,s_2{:}S$ armazenamento$(v, s_1) \wedge$ armazenamento$(v, s_1) \rightarrow$
provedor(s_1) = provedor(s_2)

Aplicando a esse axioma o padrão ***Pré-Condição*** definido no capítulo anterior, o seguinte código é gerado:

```
public class Video
{
    public boolean setArmazenamento (S s2)
    {
        boolean result = false;
        if (result = checkArmazenamento(S2))
        {
                armazenamento.add(S2);
                S2.setArmazenamento(this);
        }
        return ok;
   }

   public boolean checkArmazenamento(Servidor s2)
   {
        Set aux = this.armazenamento();
       Servidor s1 = Set.any(aux);
       return s1.equals(s2);
    }
}
```

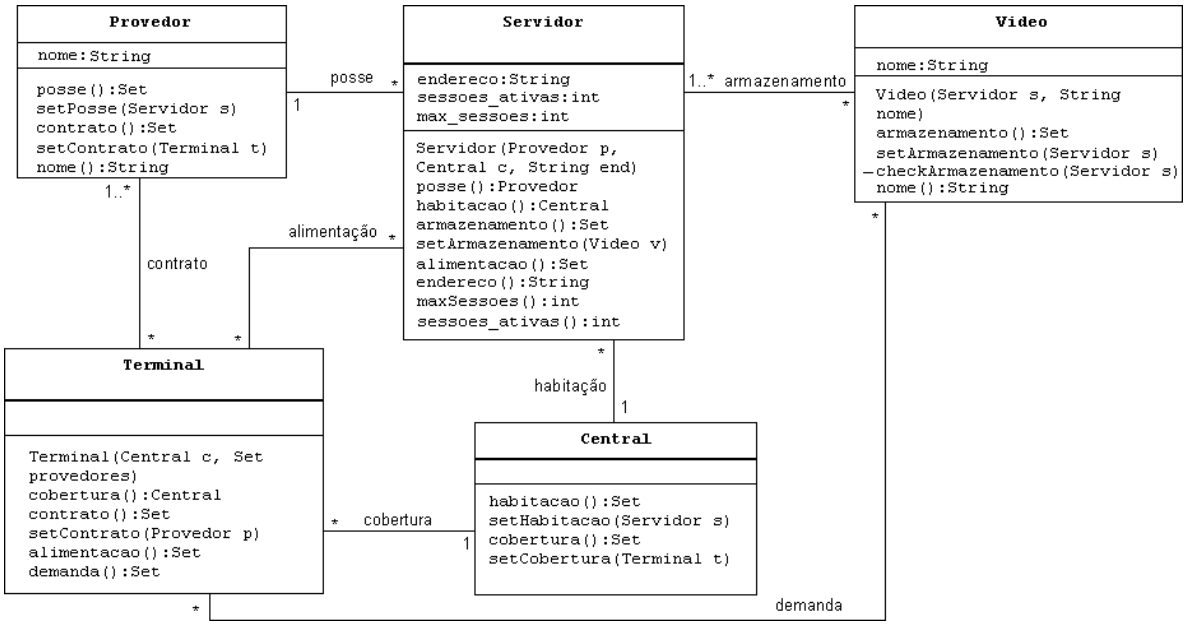


Figura 5.7 - *Framework* de gerência de Vídeo sob Demanda derivado a partir de uma ontologia de domínio

5.5 - Hipervisão: de uma ontologia de domínio a aplicação de vídeo sob demanda

Como citado anteriormente (5.2.1), o sistema Hipervisão foi desenvolvido no contexto do projeto DAMD, com o objetivo de experimentar os diversos aspectos da metodologia proposta. O sistema tem como requisito geral criar uma estrutura interdisciplinar, conectando vários departamentos acadêmicos, possivelmente de várias instituições em diferentes países, a fim de oferecer aos seus usuários finais (alunos) o acesso a vídeos digitalizados. Para isso, foi instanciada a arquitetura conceitual, sugerida pela ontologia da seção anterior, em um contexto educacional e, em seguida, foram desenvolvidas as fases pertencentes ao nível de aplicação apontadas pelo modelo de processo concebido na seção 5.2 (análise, projeto arquitetural, projeto detalhado e implementação). Desta forma, o *Hipervisão* será mostrado aqui como um estudo de caso da utilização desse processo.

O objetivo dessa seção não é mostrar toda a especificação do sistema construído (mesmo porque isto ocuparia bem mais que uma seção), mas sim salientar pontos importantes na transição de uma perspectiva de domínio para a de uma aplicação, no que

diz respeito ao nível de gerência, bem como ilustrar o desenvolvimento orientado a objetos de uma aplicação multimídia distribuída, abordando aspectos do nível de sistema.

Apesar da ontologia desenvolvida ser genérica e, portanto, independente da configuração de distribuição adotada, daqui em diante a discussão será centrada na distribuição descentralizada. Isso é devido ao fato dessa ter sido a opção escolhida no projeto Hipervisão, pela sua adequação ao propósito do sistema e ao cenário de implementação.

A figura 5.8 ilustra a disposição dos elementos envolvidos nessa arquitetura. Os terminais do usuário são agrupados em centrais (Redes de Domínio Local), que são, por sua vez, conectadas umas com as outras através de uma Rede de Acesso Remoto (RAR). Independentemente das escolhas tecnológicas para a materialização de cada um desses elementos, as relações e restrições em que estão envolvidos são da mesma forma válidas.

Figura 5.8 – arquitetura conceitual de vídeo sob demanda

Abaixo são exemplificados dois diferentes cenários de concretização dessa arquitetura:

1. **Cenário comercial:** Numa implementação comercial, os terminais são materializados como televisões e *set-top boxes* (STB), e a central é uma central telefônica conectando através de um enlace ADSL usuários dentro de um mesmo círculo de vizinhança (distância máxima exigida pela tecnologia ADSL para que se alcance a taxa de dados necessária para transmissão de vídeo digital). As centrais telefônicas são conectadas por um *backbone* ATM de alta velocidade (metropolitano ou mesmo nacional). Neste cenário, provedores aparecem como empresas de telecomunicações e de entretenimento (figura 5.9).

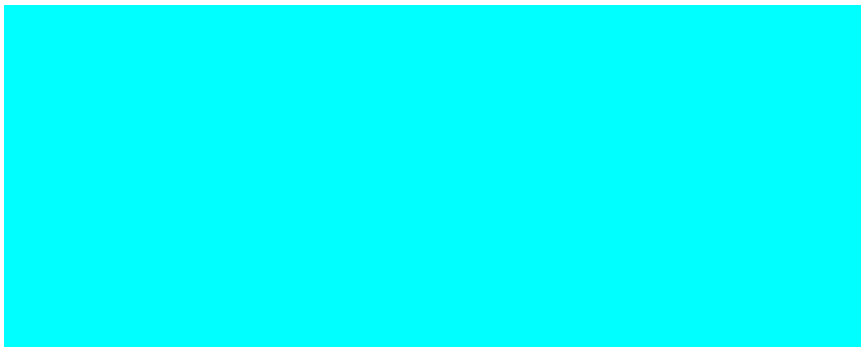


Figura 5.9 –Implementação de um serviço de vídeo sob demanda em um cenário comercial

2. **Cenário educacional:** Por um outro lado, em um cenário educacional, no qual os terminais são computadores ligados a uma rede local departamental. Provedores podem ser colegiados de cursos acadêmicos, departamentos ou centros. A RAR pode ser um *backbone* institucional conectando os vários departamentos de uma mesma instituição ou até mesmo a Internet conectando centros educacionais ao redor do mundo (figura 5.10).

Figura 5.10 –Implementação de um serviço de vídeo sob demanda em um cenário educacional

O projeto *Hipervisão*, por se tratar de um projeto acadêmico, o último cenário foi o escolhido para ser implementado. Portanto, as centrais neste caso, são redes locais TCP/IP conectadas umas as outras através da Internet. A figura 5.11 exemplifica este cenário. Neste exemplo, as centrais C_1 e C_2 representam, respectivamente, os departamentos de Computação e Engenharia Elétrica de uma mesma universidade. Neste caso, P_2 é o curso de mestrado em Ciência da Computação e P_3 o curso de graduação em

Engenharia Elétrica. O provedor P_1 é do curso de especialização em Redes de Computadores, oferecido de maneira conjunta pelos dois departamentos. Suponha que um aluno X do Departamento de Informática cursa o mestrado em ciência da computação e o curso de especialização em Redes de Computadores, X contrata os provedores P_1 e P_2 e tem acesso ao sistema se conectando à central C_1.

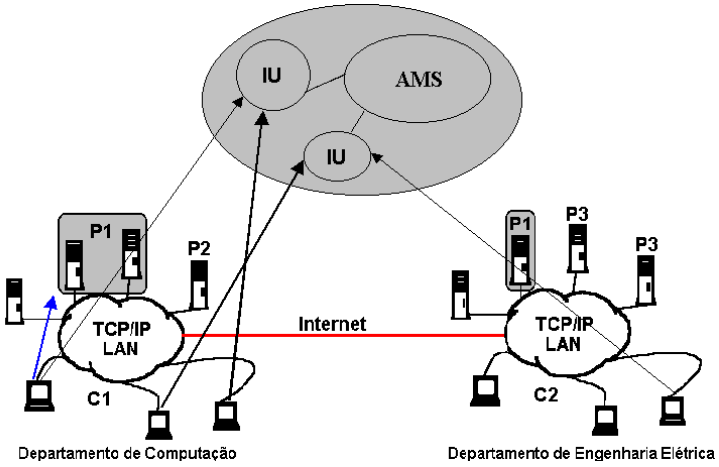


Figura 5.11 – Arquitetura do sistema Hipervisão

Este sistema de vídeo sob demanda possui três tipos de usuário: o administrador do AMS[13], os administradores de cada um dos provedores (ex.P_1) e o usuário final do sistema (EUTUser), esse último se dividindo em *ParentUser* - que é o usuário responsável pelo terminal - e *ChildUser*. Um usuário do tipo *ChildUser* está sempre ligado (e é de alguma forma dependente) do primeiro. Esta distinção é feita para que restrições de gênero de vídeo (erótico, guerra, entre outros) possam ser impostas a usuários *ChildUser* por usuários do tipo *ParentUser*. Na figura 5.12, são mostradas as principais funcionalidades do sistema oferecidas a estes usuários (casos de uso).

[13] A seção 5.3.3 apresenta uma descrição detalhada dos papéis do AMS e do EUT

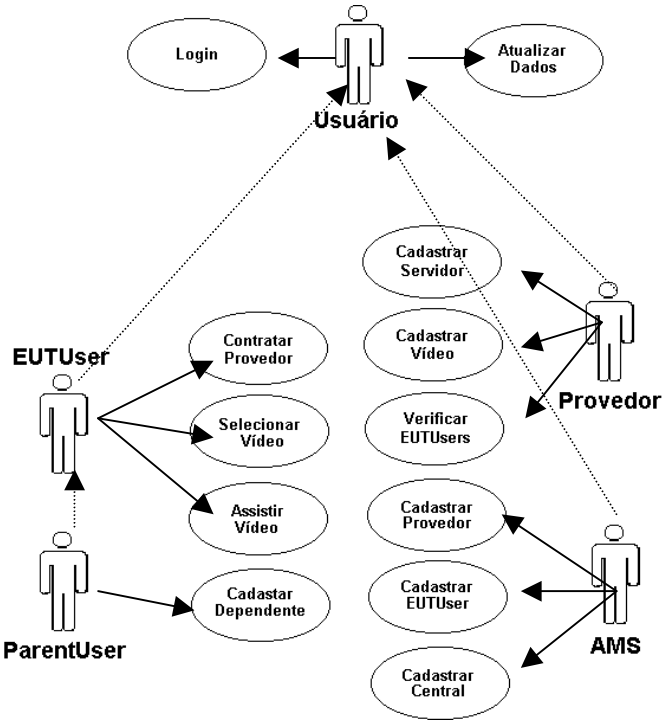


Figura 5.12 - Casos de Uso identificados na fase de análise da aplicação específica - Hipervisão

Como pode ser observado, existem dois casos de uso que são compartilhados por todos os usuários do sistema. O primeiro diz respeito ao processo de autenticação a que todos devem ser submetidos para terem acesso ao sistema (caso de uso *login*). Se este processo não for bem sucedido, uma mensagem de erro é retornada pelo sistema, caso contrário a entrada do usuário é permitida (e registrada) e a interface do ambiente é gerada, oferecendo as funcionalidades adequadas ao tipo do usuário.

O outro caso de uso comum aos diversos tipos de usuário é *Atualizar Dados*. Como o próprio nome já diz, através dele os usuários têm acesso aos seus dados, ou os dados da instituição que representam, como é o caso de provedor e AMS.

Os demais casos de uso são específicos a um dos tipos de usuário e suas descrições são sucintamente apresentadas a seguir, agrupadas pelos tipos de usuário. A única exceção é *Assistir Vídeo,* que pertence ao nível de sistema e é apresentado na seção 5.5.2.

a) **AMS:**

- **Cadastrar Provedor:** Para fazer parte do sistema Hipervisão, os provedores devem ser cadastrados no AMS. O sistema deve conhecer os dados do provedor (nome, localização e a instituição a que pertence), bem como de seu administrador (nome, email, telefone e senha de acesso ao sistema). O administrador do AMS também deve ser capaz de verificar os provedores cadastrados e seus respectivos servidores.
- **Cadastrar EUTUsers** Da mesma forma, para poderem utilizar o sistema, contratando provedores e, conseqüentemente, tendo acesso ao seus catálogos de vídeo, os usuários finais devem possuir um cadastro no AMS. Os dados necessários para o cadastro são nome, telefone, data de nascimento, email, login e senha. Na verdade, o administrador do AMS cadastra apenas o usuário responsável pelo terminal (*ParentUser*), sendo ele responsável por cadastrar seus dependentes. No momento do cadastro, os usuários são associados à central através da qual terão acesso ao sistema. O AMS deve, também, ser capaz de verificar os usuários responsáveis cadastrados com seus respectivos dependentes.
- **Cadastrar Central:** É fundamental que, inicialmente, existam centrais cadastradas no sistema. São para essas centrais que os provedores alocam servidores de vídeo para que possam oferecer seus serviços. É também através delas que os usuários finais têm acesso ao sistema. Uma central também possui um nome, uma localização e pertence a uma instituição. Possui também um administrador cadastrado, cujos dados são idênticos ao dos administradores dos provedores.

b) **Provedor:**

- **Cadastrar Servidor:** O administrador de um provedor deve ser capaz de cadastrar seus servidores de vídeo, associando-os a uma das centrais cadastradas no sistema. Cada servidor deve possuir um endereço IP único. Deve também ser configurado para aceitar um número máximo de sessões de exibição simultânea, a fim de não degradar a qualidade mínima exigida pelo serviço. O administrador do

provedor, deve ser capaz de verificar todos os seus servidores cadastrados, a quais centrais estão ligados e vídeos que cada um armazena.

- **Cadastrar Vídeo:** Os vídeos devem ser cadastrados, contendo nome, duração, descrição, gênero (terror, comédia, ficção, entre outros), diretor, elenco, nacionalidade e ano. Além disso, a cada vídeo é associada uma imagem (possivelmente de alguma cena do vídeo), para que possa compor a apresentação do vídeo vista pelos usuários finais. Como mencionado na seção 5.3.1, as mídias contínuas como áudio e vídeo, demandam dos servidores altos requisitos de capacidade de armazenamento. A fim de minimizar estes requisitos, bem como aumentar as possibilidades de associação dessas mídias, os arquivos digitalizados de vídeo, áudio e legenda (entre outras possíveis mídias no futuro) devem ser cadastrados separadamente. Para cada vídeo é inicialmente cadastrado o seu arquivo de mídia digitalizado, seu formato de codificação e as dimensões recomendadas para a tela de exibição. Uma vez feito isso, é possível cadastrar para um mesmo vídeo várias opções de áudio e legenda. Da mesma forma, para cada uma dessas opções deve ser cadastrado o arquivo de mídia digitalizado com o respectivo formato de codificação e uma descrição sucinta. Finalmente, um vídeo pode ser cadastrado em vários servidores, desde que pertençam ao mesmo provedor ($\forall$ v:V,s_1,s_2:S armazenamento(v,s_1) $\wedge$ armazenamento(v,s_2) $\rightarrow$ Im(s_1, posse) = Im(s_2,posse).
- **Verificar EUTUsers:** O administrador de um provedor p1 deve ser capaz de conhecer todos os usuários que o contratam (Im(p_1,contrato)), bem como seus respectivos dependentes.

c) Usuário Final (EUTUser):

- **Contratar provedor:** Um usuário final e_1 deve ser capaz de verificar seus provedores contratados, além de verificar outros provedores existentes com os quais não possui um contrato no momento (~ Im(terminal(e_1),contrato)[14]) .

[14] O símbolo ~ nesse caso representa a operação de complemento de um conjunto e não a negação lógica

- **Cadastrar Dependente:** O usuário responsável pelo terminal, deve ser capaz de cadastrar seus dependentes, assim como impor restrições de gênero a cada um deles. Por exemplo, um responsável pode querer que um de seus dependentes não assista filmes cujo gênero seja erótico ou guerra.
- **Selecionar Vídeo:** No caso de uma distribuição centralizada, o conjunto resultante da relação definida no axioma **(A03)** é igual ao conjunto abaixo:

$$(\forall t{:}T, \forall v{:}V\ demanda(terminal(e_1),v) \leftrightarrow v \in Im(Im(Im(terminal(e_1), contrato), posse), armazenamento))$$

ou seja, todos os vídeos armazenados em todos os servidores pertencentes aos provedores que um usuário contrata. Por outro lado, em uma configuração de distribuição descentralizada - como é o caso dessa aplicação - é importante que essa distinção seja ressaltada, visto que o primeiro conjunto (A03) pode ser um subconjunto do segundo. Nesse caso, cada provedor que um terminal contrata pode possuir servidores espalhados em várias centrais. Desta forma, o catálogo do terminal não é composto somente pelos vídeos armazenados nos servidores que o alimentam $Im(Im(terminal(e_1),alimentação),armazenamento)$**,** mesmo que este último seja o conjunto de vídeos que o usuário pode efetivamente assistir no momento em que desejar, ou como poderia ser chamado, o seu catálogo *on-line*. Se o vídeo escolhido pelo terminal não está em seu catálogo *on-line,* então ele precisa ser transportado do servidor em que está armazenado para um servidor conectado à mesma central desse terminal ($s \in Im(Im(terminal(e_1),cobertura),habitação)$).

O usuário final deve, portanto, ser capaz de visualizar e selecionar para exibição vídeos pertencentes em seus catálogos *on-line* e *off-line*. Quando um vídeo é selecionado, a fronteira entre os níveis de gerência e sistema é transposta, tendo início o caso de uso *Assistir Vídeo*.

Por fim, o catálogo de um usuários do tipo *ChildUser* é o subconjunto do respectivo catálogo de seu responsável, em que nenhum dos vídeos é classificado em um dos gêneros restringidos para aquele usuário.

A partir da requisitos levantados na atividade de análise dessa aplicação específica, é identificada a adequação do reuso do *framework* de gerência de VoD desenvolvido na seção 5.4. Porém, para que os requisitos específicos da aplicação (documentados nos casos de uso) possam ser satisfeitos, é necessário que serviços adicionais sejam derivados e que novas classes seja incorporadas ao *framework* original. Além disso, esta fase pode gerar novas questões de competência específicas da aplicação, agregando novos axiomas, e conseqüentemente impulsionando tanto a evolução do framework original quanto a criação de uma ontologia de aplicação. O *framework* derivado nesse processo se comporta como um corpo de conhecimento capaz de responder às questões de competência tanto do domínio quanto da aplicação.

A seção 5.5.1 discute os aspectos mais importantes desse processo de especialização. As seções 5.5.2 e 5.5.3 discutem, respectivamente, aspectos relativos ao projeto e implementação dos casos de uso dos níveis de gerência e sistema.

5.5.1 - Especialização do framework de gerência de VoD

Nesta atividade, subclasses específicas da aplicação são derivadas a partir das classes originais do framework através do uso de herança. A fim de eliminar os efeitos colaterais decorrentes da má utilização dessa técnica - como quebra de encapsulamento e sobrescrita inadequada de métodos - as interfaces contratuais das superclasses são declaradas de forma a serem impassíveis de serem sobrescritas, se comportando assim como um verdadeiro *framework* caixa-preta (*black-box framework*).

No caso geral, por uma convenção aqui adotada, as classes derivadas possuem o mesmo nome de suas superclasses precedido pelo sinal _, sendo a exceção os casos em que um nome específico adiciona legibilidade e clareza ao modelo.

A seguir cada uma das partes do modelo derivado é discutida em maiores detalhes.

a) Central e Provedor

Como pode ser observado, as classes derivadas adicionam métodos, atributos e relacionamentos com objetos próprios da aplicação. Na figura 5.13, por exemplo, as classes derivadas de *Central* e *Provedor* se relacionam com um objeto *Adm*, que representa seus administradores.

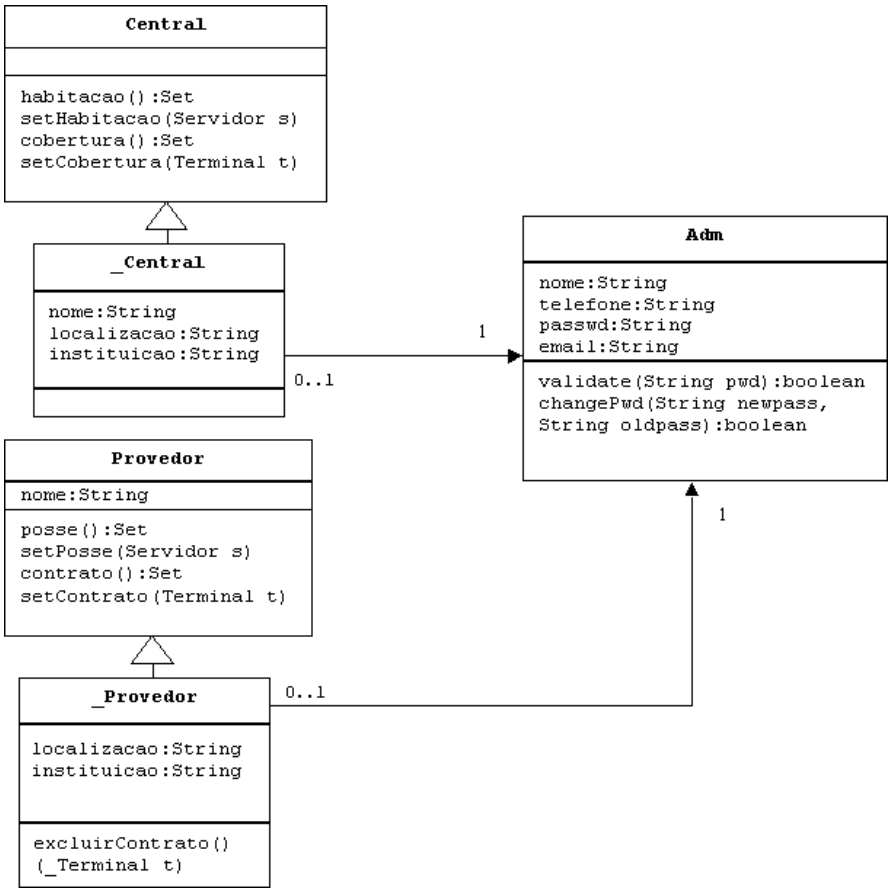


Figura 5.13 - Especialização das classes *Provedor* e *Central*

b) Servidor e Vídeo

Como descrito no caso de uso Cadastrar Vídeo, as mídias contínuas áudio e vídeo devem ser modeladas e implementadas como objetos separados. Na figura 5.14, é apresentada a solução adotada. A classe derivada _Video se relaciona com um objeto MidiaPrimaria, que representa o material de vídeo digitalizado em algum formato suportado (MPEG, AVI, H.263, entre outros), e pode, opcionalmente, se relacionar com um ou mais objetos do tipo MidiaSecundaria (áudio ou legenda). Dessa forma, um mesmo vídeo, por exemplo, *A Lista de Schindler*, pode possuir várias opções de áudio (ex. inglês e espanhol) e de legenda (ex. português e holandês).

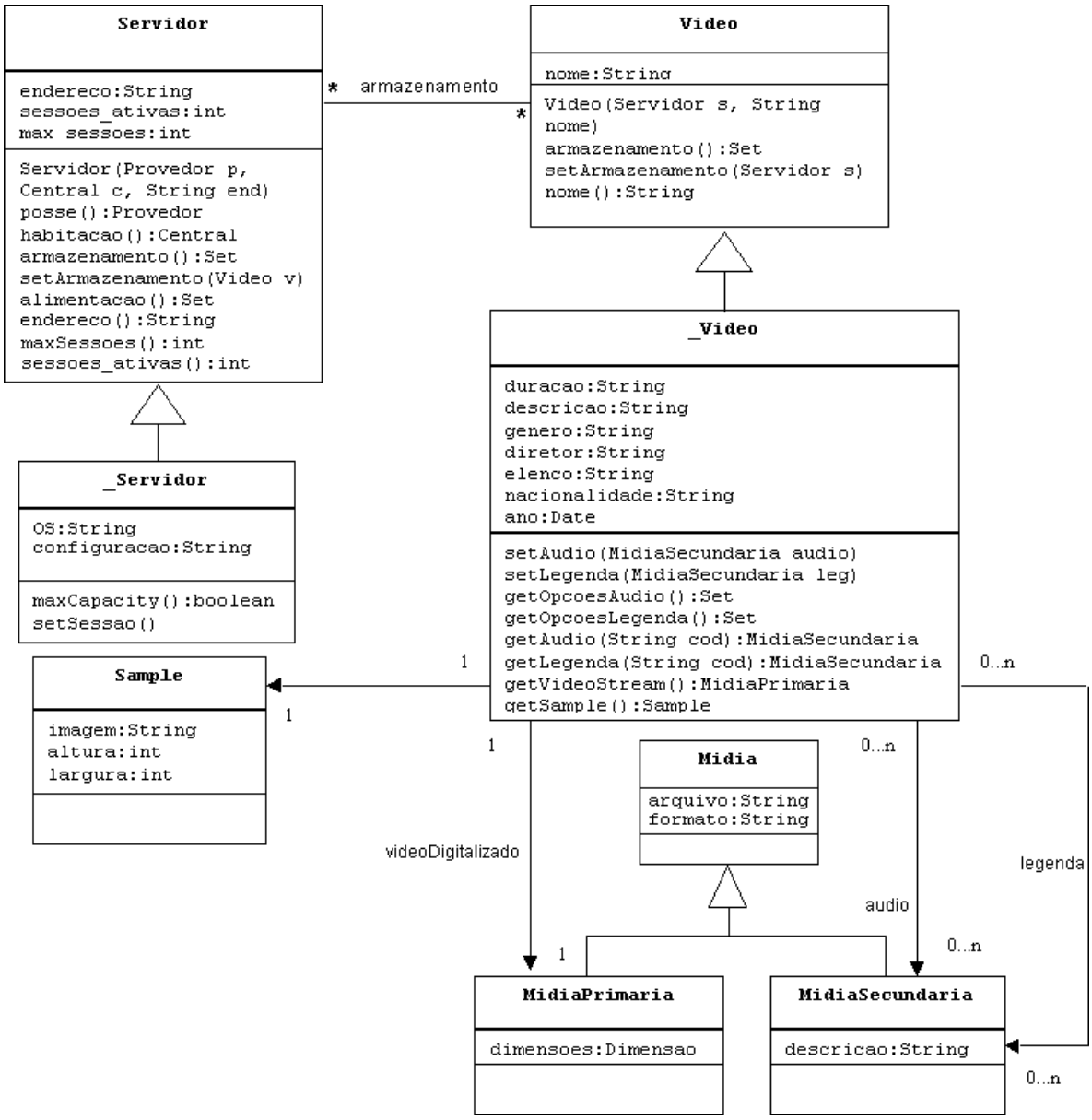


Figura 5.14 - Especialização das classes *Servidor* e *Vídeo*

Como um mesmo vídeo pode ser armazenado em vários servidores, a aplicação faz uso das funcionalidades de controle de sessões de exibição, implementas pela classe _Servidor, com o objetivo de promover o balanceamento de carga entre eles.

c) Terminal

À classe derivada _Terminal, podem estar indiretamente associados vários usuários finais (EUTUser). Um único usuário ParentUser é responsável pelo terminal.

Esse usuário, no entanto, pode possuir vários dependentes, aos quais pode impor restrições de gênero. Desta forma, o conjunto de vídeos que um ChildUser pode assistir pode ser um subconjunto daquele definido pela relação ***demanda*** existente entre Terminal e Video. Isso vale tanto para os catálogos *on-line* quanto *off-line*.

Deste modo, a principal questão de competência da ontologia dá origem a uma outra no nível de aplicação: *Dado um EUTUser, qual são seus catálogos on-line e off-line?*

Para responder a essa questão são derivados os seguintes axiomas:

W:_Terminal

X:ParenteUser

Y:ChildUser

(A04) w:W, □ v:V catalogo(w,v) ↔ v ∈ (Im(Im(w,alimentação) /
Im(Im(Im(w,contrato),posse),armazenamento))

(A05) x:X, □ v:V demanda (x,v) ↔□v ∈ Im(terminal(x),demanda)

(A06) x:X, □ v:V catalogo (x,v) ↔□v ∈ Im(terminal(x),catalogo)

(A07) □ y:Y, □ v:V demanda (x,v) →□v ∈ _Video

(A08) □ y:Y, □ v:V demanda (y,v) ↔□
□v ∈ Im(responsavel(x),demanda)) ∧ (genero(v) ∉ restricoes(y)

(A09) □ y:Y, □ v:V catalogo (y,v) ↔□
□v ∈ Im(responsavel(x),catalogo)) ∧ (genero(v) ∉ restricoes(y)

Como pode ser observado, o formalismo definido no Capítulo 4 e usado para descrever o conhecimento do domínio capturado pela ontologia, também pode ser aplicado para formalizar axiomas de derivação e consolidação pertencentes ao nível de aplicação. Da mesma forma, as regras de transição podem ser aplicadas para derivar as invariantes e pré-condições que devem estar presentes no diagrama de classes da aplicação derivado a partir do framework. A figura 5.15 mostra o diagrama de classes correspondente para os conceitos envolvidos nos axiomas acima descritos.

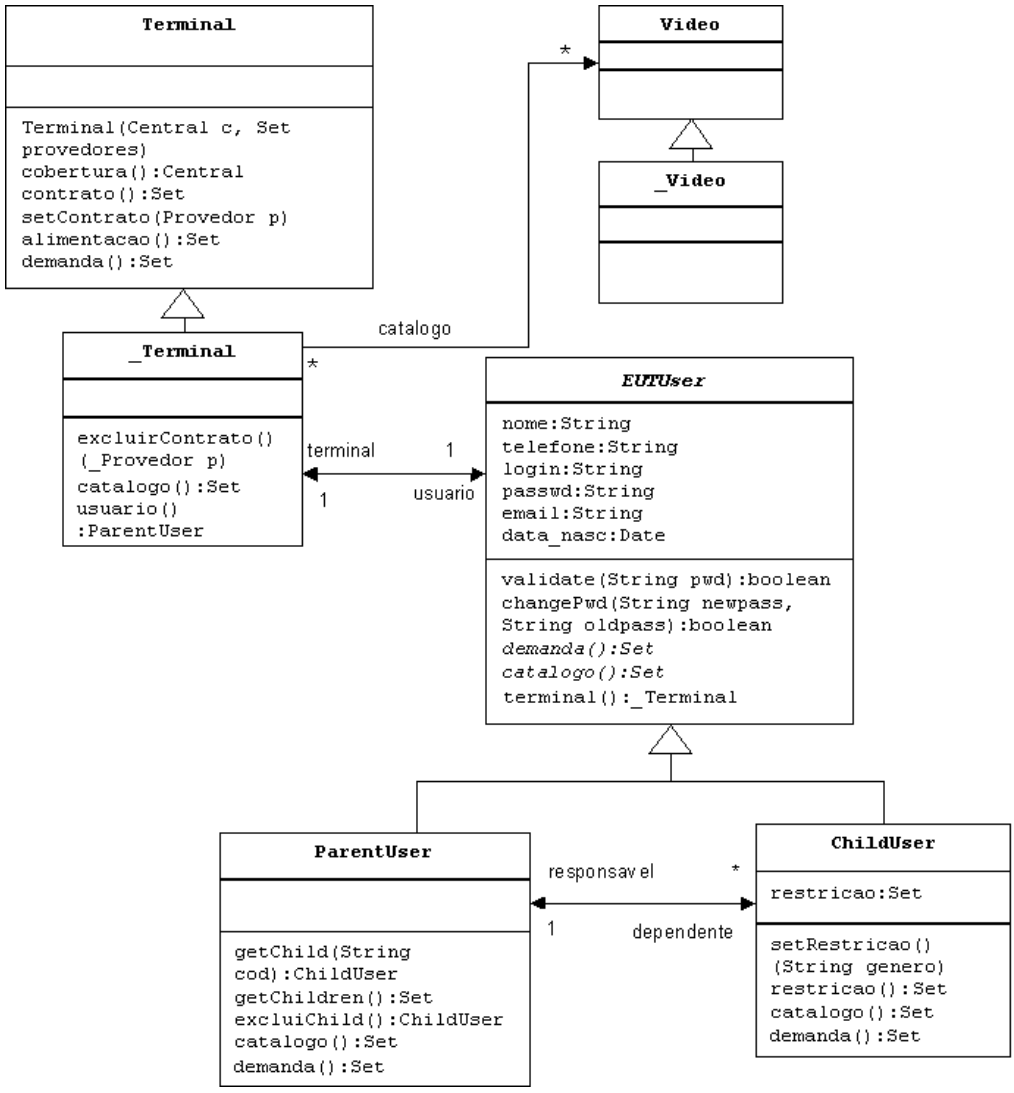


Figura 5.15 - Especialização da classe *Terrminal*

5.5.2 - Projeto e Implementação do nível de gerência

Todas as decisões de projeto e implementação tomadas - como a escolha da linguagem de programação e dos modelos de interface e protocolos, bem como a definição da arquitetura de software - têm como objetivo atingir os requisitos não-funcionais de escalabilidade, flexibilidade e adaptatividade que vêm sendo discutidos ao longo desse trabalho. A figura 5.16 mostra as soluções adotadas para implementação das funções do nível de gerência nas camadas de interface, lógica do negócio e persistência.

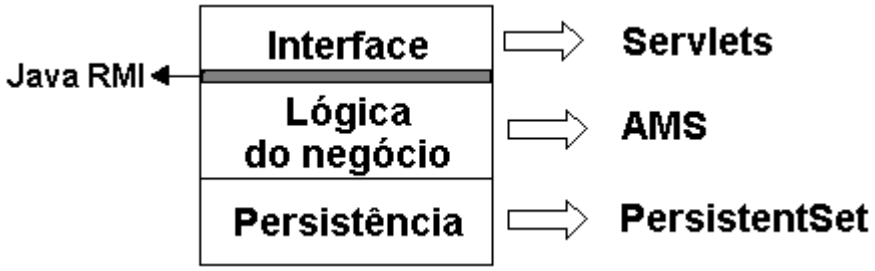


Figura 5.16 - Desenvolvimento em camadas do nível de gerência no *Hipervisão*

Nesse trabalho, como é ilustrado pela figura 5.11, a camada de interface é implementada através de diversos servidores de interface (IU) distribuídos. Os usuários têm acesso ao sistema se conectando através de um *browser* a um desses servidores. A escolha de fornecer uma interface via WWW deve-se ao fato de ser um alternativa padronizada, aberta, bem difundida e que permite a utilização de estações clientes com baixa capacidade de memória e baixo poder de processamento. Desse modo, a estação cliente cuida apenas dos aspectos relativos à apresentação do conteúdo e interação com o usuário final, enquanto que todo o processamento da interface - como controle de sessão, geração dinâmica do conteúdo e interação com as camadas de lógica do negócio e persistência - é feito pelos servidores (IU). Além disso, é extremamente interessante que tanto a navegação do usuário, quanto a escolha de como a interface é apresentada a ele, sejam controladas pelos AMS, possibilitando um maior grau de personalização do serviço.

Para a implementação das funções de processamento da interface, foi utilizada a tecnologia de *Servlets*. Essa tecnologia especificada pela SUN (Voss, 2000, Lubling & Razorfish, 2000, McPherson, 2000) tem como intuito oferecer uma nova abordagem para programação de servidores HTTP[15]. Seu objetivo é oferecer a esses servidores, o mesmo papel desempenhado pelas *Applets* para o lado cliente (*browsers*), ou seja, propiciar um

mecanismo independente de plataforma para a extensão dinâmica de funcionalidades - carregamento dinâmico de *bytecodes*[16] Java. Além da independência de plataforma, este mecanismo oferece algumas outras vantagens se comparado a tecnologias alternativas como CGI (Common Gateway Interface) (Ben-Natan, 2000), ASP (Active Server Pages) (Francis et al., 1998) e PHP (Professional Home Page) (Professional…, 2000):

- *Rapidez de resposta*: Cada vez que um CGI é requisitado, um novo processo é criado pelo sistema operacional do servidor WWW. Ao invés disso, um processo só é criado para uma servlet a primeira vez que ela é invocada (a menos que o seu código seja alterado). A partir de então, todas as requisições subsequentes são feitas ao processo residente.

- *Gerência de Sessão*: Pelo fato do protocolo HTTP ser independente de estado (*stateless*), o conceito de sessão de trabalho é inexistente, ou seja, requisições subseqüentes de um mesmo usuário a um mesmo servidor HTTP são tratadas de forma completamente independente. Em (Ben-Natan, 2000), é apresentada uma tecnologia desenvolvida pela Netscape chamada *cookies* que permite endereçar essa questão, ainda que de maneira rudimentar. Além dessa opção, as servlets permitem a criação de uma verdadeira sessão de trabalho, na qual objetos podem ser armazenados e posteriormente recuperados, permitindo assim a comunicação entre servlets. Dessa maneira, é possível não apenas reconhecer quem é o usuário que vem interagindo com a aplicação, como construir modelos complexos de comportamento.

- *Capacidade de desenvolvimento*: Ao contrário das linguagens usadas em ASP e PHP (que são meras linguagens de script), servlets utilizam uma verdadeira linguagem de programação (Java) e, com exceção de algumas restrições de segurança, podem fazer qualquer coisa que um objeto Java ordinário é capaz. Exemplos dessas habilidades são o atendimento paralelo e a sincronização de múltiplas requisições simultâneas e a capacidade de delegação de uma

[15] Hyper Text Transfer Protocol - protocolo usada para transferência de documentos HTML na World Wide Web (WWW)
[16] Código intermediário gerado para a Máquina Virtual Java (JVM)

requisição a outros servidores (ou outras servlets), a fim de realizar funções como balanceamento de carga entre servidores redundantes.

Portanto, em cada um dos servidores IU, existe um conjunto de servlets com objetivo de permitir (e controlar) o acesso a funções específicas de gerência, implementadas pelo AMS. A estratégia de alocação de funcionalidades às servlets é a seguinte: (i) primeiro, para cada caso de uso do nível de gerência é feita um decomposição funcional; (ii) uma vez feito isso, para cada uma das funções terminais encontradas, é, então, associada uma servlet a ser implementada. No final desta subseção, este processo é exemplificado, usando os casos de uso *Login* e *Selecionar Vídeo*.

O conjunto de todas funcionalidades associadas às servlets - resultantes do processo de decomposição funcional aplicado a todos os casos de uso - define também a interface contratual do AMS, ou seja, todos os serviços que ele deve oferecer. O framework especializado na seção 5.4, se apresenta como uma base de conhecimento capaz de responder todas as questões de competência tanto do domínio quanto da aplicação, o acesso a esse conhecimento (ou a parte dele) é provido de maneira organizada pelo AMS

O AMS desempenha, também, o papel de interface com a base de dados persistentes. Como os frameworks são desenvolvidos utilizando o *package* Set, a persistência é feita através do tipo PersistentSet. Um objeto PersistentSet é um conjunto capaz de fazer sua persistência (e de todos elementos que o compõem) de maneira transparente ao usuário do tipo. Elementos que devem ser persistentes devem implementar a interface SetElement. A base de dados é definida, portanto, como uma família, ou seja, um conjunto de conjuntos. Cada conjunto membro de uma família deve implementar a interface MemberSet. As classes Set e PersistentSet, bem como as interfaces SetElement e MemberSet foram definidas na seção 4.3.1. Ao ser construído, o AMS recupera cada conjunto membro da base de dados, passando a referenciá-los diretamente.

A figura 5.17 mostra o uso dessa abordagem para o caso do sistema Hipervisão.

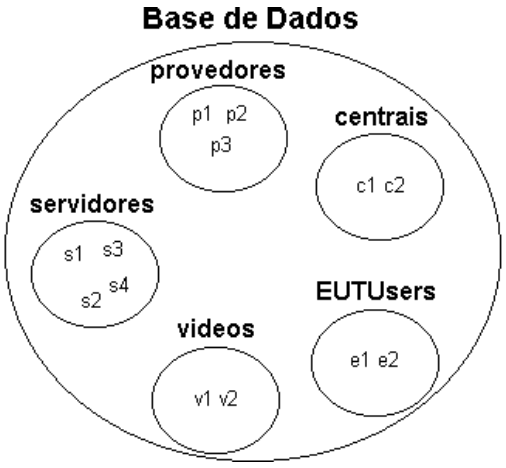


Figura 5.17 - Base de Dados como uma família de conjuntos

Como os servidores de interface (IU) e o AMS estarão no caso geral remotamente distantes, a sua comunicação é feita através de um ORB (*Object Request Broker*) que gerencia a comunicação entre os objetos distribuídos. Este componente permite tanto a invocação remota de métodos quanto a troca de objetos passados como parâmetro através da rede. Esta tecnologia que teve origem com o advento das RPCs (Remote Procedure Call) conta atualmente com algumas alternativas de implementação, sendo as principais delas:

- **CORBA (*Common ORB Arquitecture*) - OMG:** Provê um suporte completo a formas complexas de comunicação remota de objetos, implementados em diversas linguagens de programação e executando em diversas plataformas (Ben-Natan, 2000).
- **DCOM (Distributed Common Object Model) - Microsoft:** Suporta a comunicação de objetos implementados em diversas linguagens diferentes, executando na plataforma Windows (Ben-Natan, 2000).
- **Java RMI (Remote Method Invocation) - SUN:** Suporta a comunicação de objetos Java em diversas plataformas (Horstmann & Cornell, 1997, McPherson, 2000, McCluskey, 2000).

Um importante requisitos deste trabalho é manter a independência de plataforma, eliminando a possibilidade do uso de DCOM. Sem dúvida o padrão CORBA é a mais completa das três alternativas, mas como nesse trabalho todos os objetos são

implementados na mesma linguagem (JAVA), para efeito de simplificação a opção foi feita pelo uso de Java RMI.

Assim como nas RPCs, a comunicação remota em RMI é intermediada por objetos *proxy* chamados *stubs* e *skeletons*. Esses objetos são gerados a partir de uma interface e, portanto, possuem os mesmos métodos declarados por ela. Desse modo, um objeto remoto deve implementar uma interface que encapsule todos seus métodos passíveis de serem invocados remotamente. Além disso, é necessário que ele implemente o mecanismo necessário à transferência de parâmetros através da rede (*marshalling*) ou estenda uma classe que implemente este mecanismo. A classe UnicastRemoteObject, provida pelo pacote java.rmi.server da SUN, é um exemplo de classe deste tipo.

Finalmente, para que possa ser encontrado pelos clientes, o objeto remoto deve ser registrado em um serviço de nomeação. Um serviço deste tipo é provido pela classe Naming do pacote java.rmi.

A figura 5.18 apresenta o diagrama de classes do AMS. Nesse diagrama, o objeto remoto AMS implementa a interface RI_ AMS, na qual são declarados todos os seus métodos acessíveis a clientes remotos. O objeto AMS é instanciado, registrado no mecanismo de nomeação e mantido continuamente em execução pelo AMSServer. Para garantir que somente uma instância de AMS é gerada, este objeto utiliza o padrão ***Singleton*** apresentado em (Gamma et al., 1995).

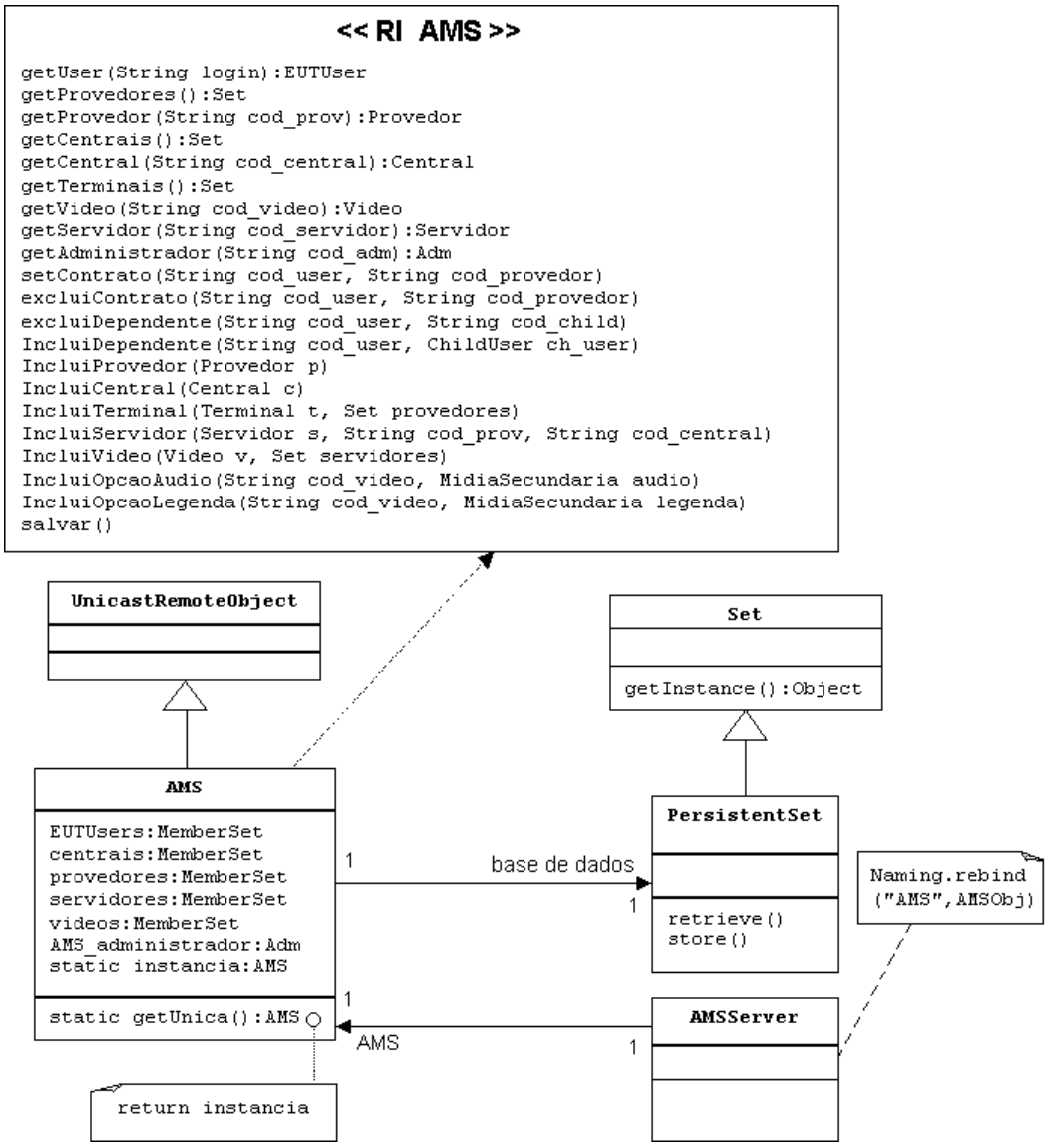


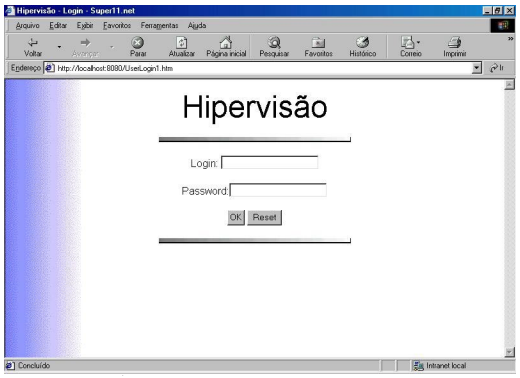
Figura 5.18 - Diagrama de Classes do *Application Management System* (AMS)

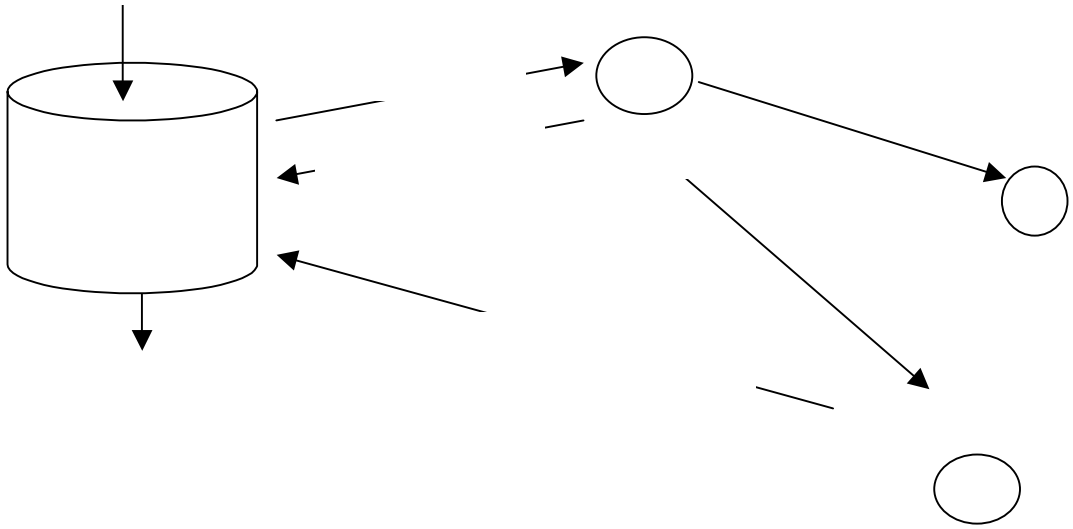
5.5.2.1 - Decomposição funcional de casos de uso com delegação de responsabilidade a *servlets* de controle

Como citado anteriormente, a interface com usuário nesse projeto é implementada através da interação de um cliente HTTP com os servidores de interface (IU). Os IUs são compostos por um servidor HTTP padrão (JavaWebServer) e um conjunto de *servlets*, cujo objetivo é gerar conteúdo HTML dinâmico a partir da interação com as outras

camadas da arquitetura de software - lógica do negócio e persistência. Por utilizar o protocolo HTTP para a troca de dados, as *servlets* estão amarradas ao modelo *request/response* do protocolo, ou seja, a cada interação do usuário é feita uma requisição ao servidor, que entrega a uma servlet específica a tarefa de respondê-la com uma nova página HTML de interface gerada dinamicamente. A figura 5.19 exemplifica este processo através do caso de uso *Login*.

UserLogin.htm





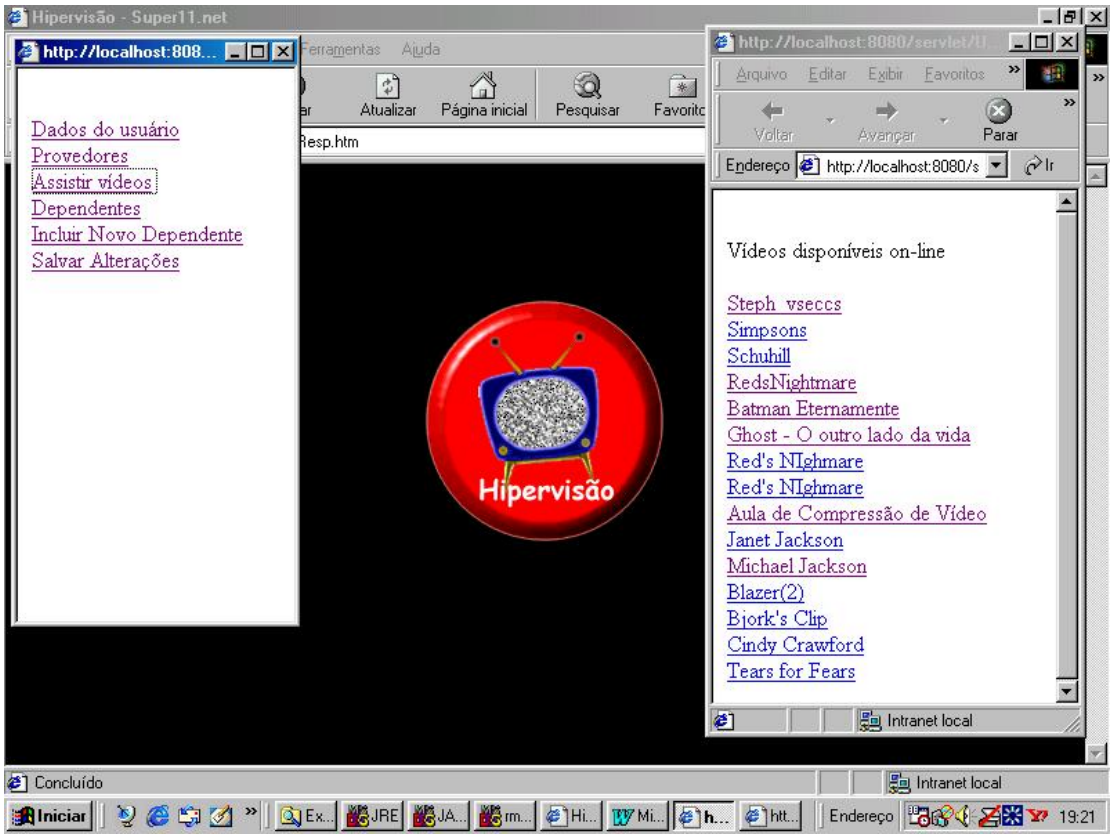


Figura 5.19 - Processo de autenticação do usuário

Nesse caso, o processo de autenticação é decomposto em duas funções: a verificação da senha e a geração do menu do usuário. A realização de cada uma dessas funções é delegada a uma *servlet* - respectivamente UserLoginServlet e MenuUserServlet. Cada *servlet* é, portanto, um objeto de controle responsável pela gerência de uma das funções que compõe o caso de uso. Nesse caso, excepcionalmente, as duas funções são conduzidas dentro de um mesmo ciclo *request/response*. Porém, isso não acontece na maioria dos casos de uso deste projeto, sendo geralmente necessários vários desses ciclos, cada um deles comprometido com a realização de uma tarefa específica. Dessa forma, o emprego de uma técnica para decomposição funcional dos casos de uso mostrou-se bastante apropriada.

A técnica escolhida foi a definida pelo padrão ITU-T Z.100 (ITU SDL, 1994) pela sua clareza e simplicidade. A figura 5.20 mostra um resumo da notação proposta por ela.

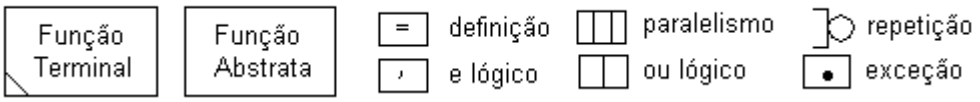


Figura 5.20 - Resumo da notação proposta na norma Z.100 para decomposição funcional

A seguir, é mostrada a aplicação dessa técnica para o caso de uso *Selecionar Vídeo* (figura 5.21). A tabela 5.3 apresenta cada função terminal, associando-as às *servlets* correspondentes. São também descritas as tarefas realizadas e o resultado gerado por cada uma das *servlets*.

Figura 5.21 - Decomposição funcional do caso de uso *Selecionar Vídeo*

Função	Servlet	Página HTML retornada
Mostrar opção de seleção	UserVideos.java	Quando o usuário acessa no menu a opção de *Selecionar Vídeo*, uma lista de *hiperlinks* é gerada contendo todos os vídeos pertencentes aos catálogos on-line e off-line do usuário.
Mostrar detalhes vídeo	VideoDetails.java	Para um dado vídeo, são mostradas todas as suas informações, como nome, diretor, gênero, nacionalidade, imagem de exemplo (*sample*), entre outras. Além disso, é oferecido ao usuário um mecanismo de escolha de uma das opções de áudio em questão (vide o exemplo da figura 5.22.)
Configurar Sessão	SessionConfig.java	Quando o usuário seleciona um vídeo para assistir, então é configurada uma sessão de exibição entre o terminal do usuário e um dos servidores que armazena o vídeo em questão (Im(v_1,armazenamento)). A escolha do servidor é feita com base no número de sessões de exibição simultâneas que cada um deles está envolvido no momento, a fim de promover um mecanismo simples de balanceamento de carga entre os servidores.
Criar Sessão	Session.java	Este é objeto é, na verdade, uma *applet* e não uma *servlet*, já que todo o seu processamento é feito no terminal do usuário. Essa *applet* marca a fronteira entre os níveis de gerência e sistema, que somente é transposta ao comando de início de sessão dado pelo usuário (figura 5.23). Quando isso acontece, um objeto gerente de exibição é criado (JPlayer). Esse objeto é discutido detalhadamente na seção seguinte.

Tabela 5.2 - Funções terminais do caso de uso assistir vídeo e suas respectivas *servlets* associadas

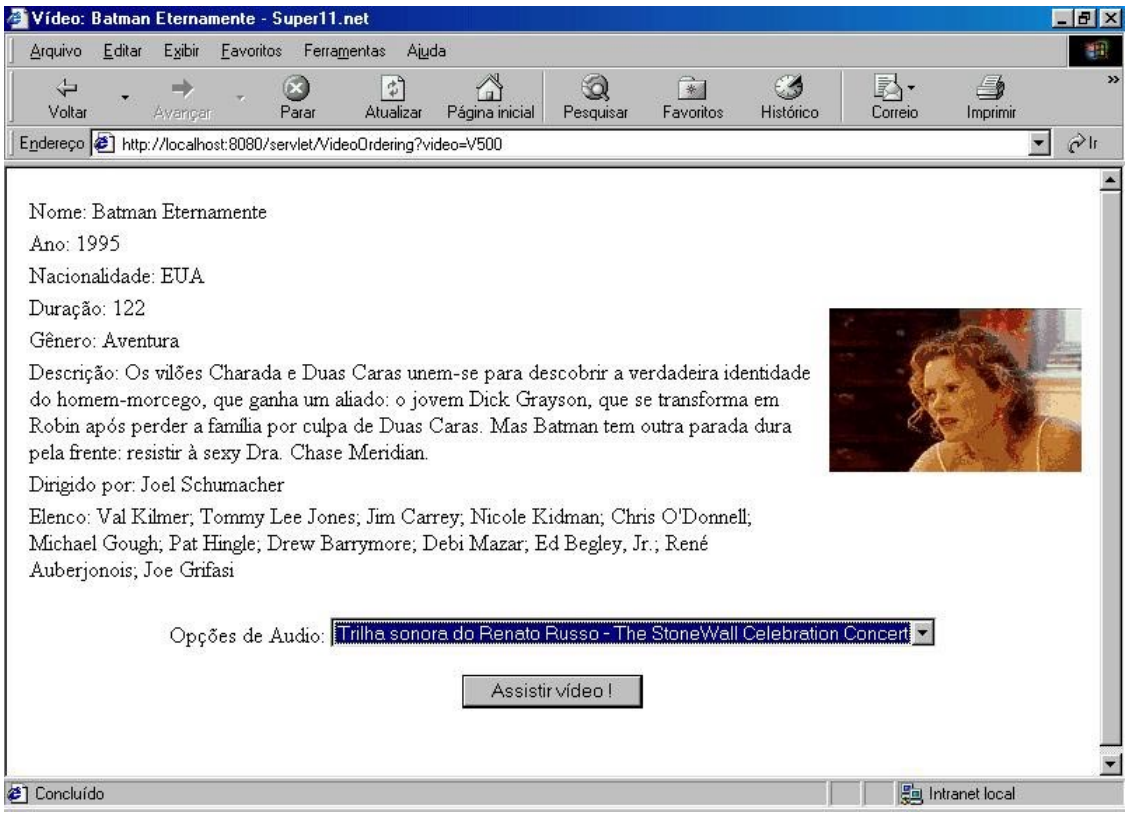


Figura 5.22 - Página de detalhes de um vídeo gerada pela *servlet VideoDetails*

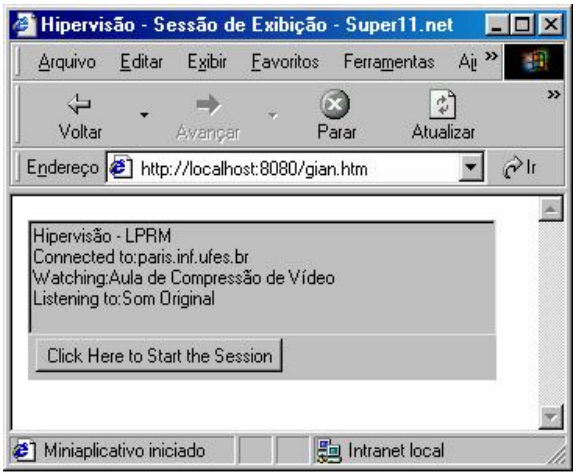


Figura 5.23 - *Applet* de configuração da sessão de exibição gerada pela *servlet SessionConfig*

5.5.3 - Projeto e Implementação do nível de sistema

O nível de sistema é percebido pelo usuário unicamente através do caso de uso *Assistir Vídeo*. Esse caso de uso é composto por duas funções distintas, mas altamente interrelacionadas:

(i) Aquisição dos dados referentes às mídias que serão exibidas;

(ii) Oferecer ao usuário os comandos de interação do videocassete virtual (exibir, retroceder, adiantar, dar pausa, controle de volume, entre outras) e cuidar para que a infra-estrutura usada possa dar suporte a essa interação;

(iii) Cuidar do inter-relacionamento entre estas duas funções. Um exemplo do impacto que um comando de interação pode causar na infra-estrutura é o caso onde o usuário adianta o vídeo (*FF*), referenciando uma posição cujos dados (ex. vídeo, áudio) ainda não foram adquiridos. Nesta situação, os objetos deste caso de uso devem cuidar de questões como: informar ao usuário da situação ocorrida, gerenciar o transporte e a re-sincronização das mídias envolvidas, entre outras tarefas.

Assim como no projeto e implementação do nível de gerência, as decisões aqui tomadas visam empregar soluções abertas, padronizadas e arquitetonicamente neutras, a fim de atingir os requisitos de escalonabilidade, flexibilidade e adaptatividade pretendidos. Além disso, o desenvolvimento é mais uma vez conduzido com um foco direto na prática de reutilização. A figura 5.24 ilustra, através de uma arquitetura em camadas, as decisões de linguagem, protocolos e *framework* adotadas.

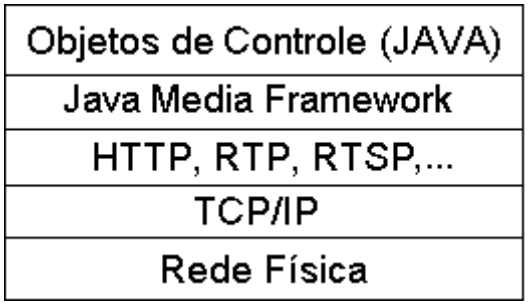


Figura 5.24 - Desenvolvimento em camadas do nível de sistema do *Hipervisão*

Como citado anteriormente, este caso de uso tem um forte relacionamento com a função abstrata *configurar exibição* descrita anteriormente e é exatamente na interseção entre eles que é marcada a fronteira entre os níveis de gerência e sistema. Quando o usuário pressiona o botão "Assistir Vídeo", é iniciado o processo de configuração da sessão de exibição. Esse processo comandado pela servlet *SessionConfig* consiste na configuração da infra-estrutura de sistema necessária para que o usuário possa assistir e interagir com vídeo escolhido. A gerência dessa infra-estrutura é delegada a um conjunto de objetos gerentes implementados em Java, cada um responsável por uma função específica (camada superior da figura 5.24). Primeiro, é criado um gerente de sessão (*Session*) que se conecta ao AMS, recuperando as instâncias apropriadas de *Video* e *Servidor*, além da opção de áudio escolhida para aquele vídeo. De posse desses objetos, o gerente de sessão cria um gerente de exibição (*JPlayer*) que configura as dimensões da tela de apresentação, cria um painel de interação (*InteractionPanel*), contendo a janela de exibição de vídeo e um painel de controle. Por fim, é também criado um gerente de apresentação (*PlaybackManager*), responsável por criar, configurar e associar *Players*, que guiarão a aquisição de um mínimo de dados para dar início à apresentação, bem como a sincronização e apresentação destas mídias. A figura 5.25 mostra as relações envolvendo essas classes.

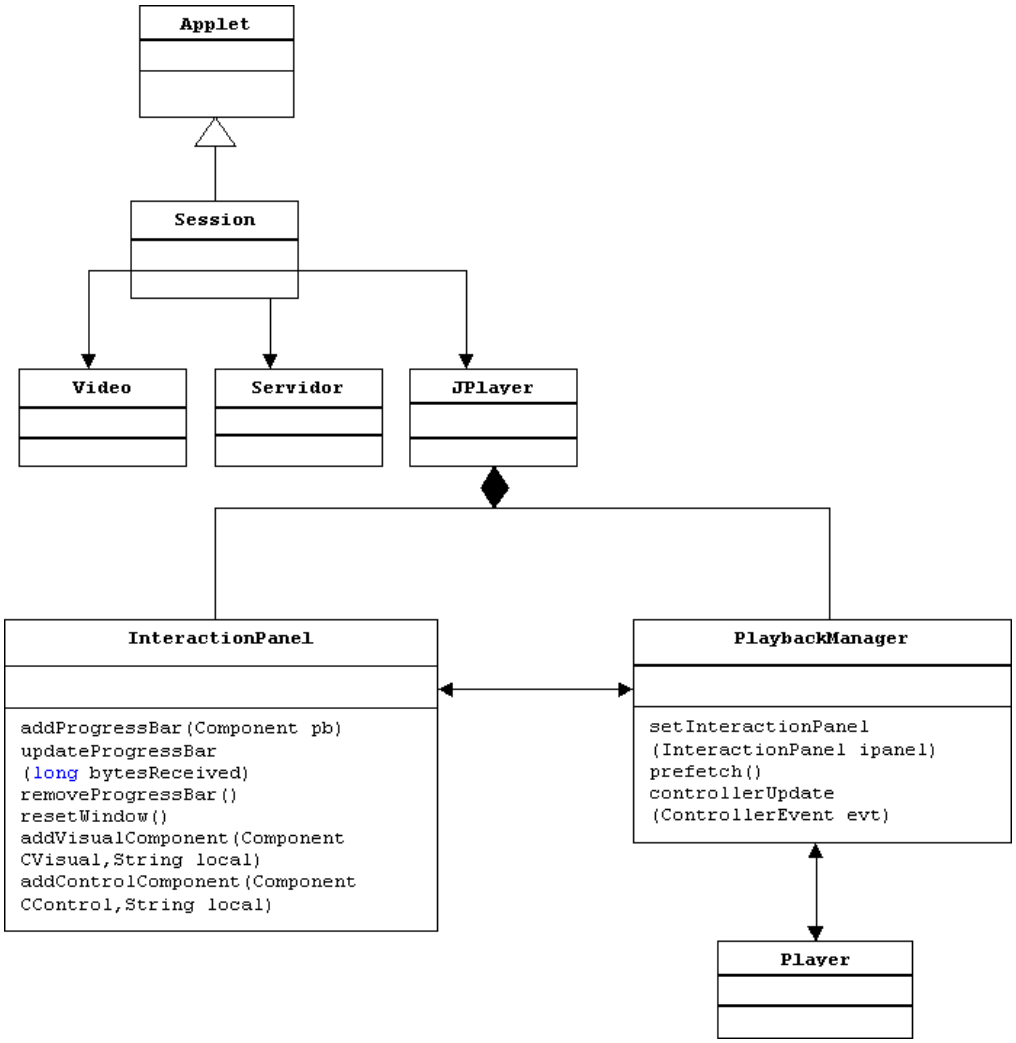


Figura 5.25 - Objetos envolvidos em uma sessão de exibição

Apesar de suas inúmeras vantagens, as primeiras versões da linguagem JAVA apresentavam um suporte extremamente limitado ao desenvolvimento de aplicações multimídia. A API básica suportava áudio de forma rudimentar (arquivos .au de 8 bits) e o suporte para vídeo era inexistente. Para suprir esta lacuna, um consórcio liderado pela SUN e contendo empresas como IBM, SGI e INTEL se comprometeu a desenvolver um extensa arquitetura chamada *Java Media and Communications API*. Esta arquitetura compreende um completo conjunto de API's para dar suporte à apresentação (*playback*) e criação de formatos multimídia avançados, como por exemplo:

- Síntese de MIDI e áudio de alta qualidade (Java Sound)
- Reconhecimento de voz e síntese de texto-palavra (Java Speech)

- Aplicações de telefonia (Java Telephony)
- Animações e Gráficos avançados (Java 2D e 3D)
- Capture e *playback* de mídias contínuas, ferramentas de conferência multimídia e trabalho cooperativo (Java Media Framework)

Como um dos objetivos deste trabalho é a implementação de um sistema de VoD, o foco da discussão é apenas no subconjunto do Java Media Framework (Java Media Player API) responsável pelas funcionalidades de *playback* (figura 5.26)[17].

Figura 5.26 - O Java Media Player no contexto do Java Media and Communications API

O Java Media Player API (Sullivan et al., 1998) é um conjunto estruturado de classes e interfaces que encapsula estratégias de resolução dos principais problemas encontrados no desenvolvimento de aplicações envolvidas no transporte, sincronização, controle e apresentação de mídias contínuas.

Como benefício adicional, por ser feito em Java, o *framework* é independente de plataforma. Além disso, são utilizados mecanismos difundidos para o desenvolvimento de conteúdo multimídia para o ambiente WWW, sem demandar a presença de *plug-ins* adicionais na estação cliente. Além disso, suporta os principais formatos padronizados de codificação (MPEG-1, MPEG-2, H.261/H.263 -Quicktime, MIDI, AVI, WAV, AIFF, AU) e protocolos (HTTP-Streaming, RTP[18], RTSP[19], MediaBase).

[17] Informações detalhadas sobre as outras API's podem ser encontradas em http://java.sun.com/products/java-media/jmf/index.html.
[18] Real Time Protocol
[19] Real Time Stream Protocol: Artigos, listas de discussão e ferramentas sobre RTP e RTSP podem ser encontradas em http://www.cs.columbia.edu/~hgs/rtp/

A peça chave do framework é a interface *Player*. Um objeto implementando esta interface é capaz de ler e processar fluxos de dados multimídia lidos a partir de uma determinada fonte, e exibi-los em um determinado intervalo temporal. Um *Player* agrupa as funcionalidades definidas nos seguintes elementos:

(a) **Clock:** Relógio de controle de tempo, que provê as funcionalidades básicas de sincronização, usadas para controlar a exibição dos dados;

(b) **Controller:** Estende as funcionalidades do Clock para prover mecanismos de aquisição de recursos e de notificação de eventos;

(c) **Duration:** Provê métodos para determinar a duração da mídia que está sendo apresentada;

(d) **Media Handler e DataSource:** Funcionam, respectivamente, como abstrações de um decodificador de mídia e um protocolo de transporte e controle;

Como pode ser observado, a interface *Player* especializa as interfaces *MediaHandler* e **Controller** para encapsular um mecanismo padronizado de apresentação e interação, independente de conteúdo codificado e de protocolo.

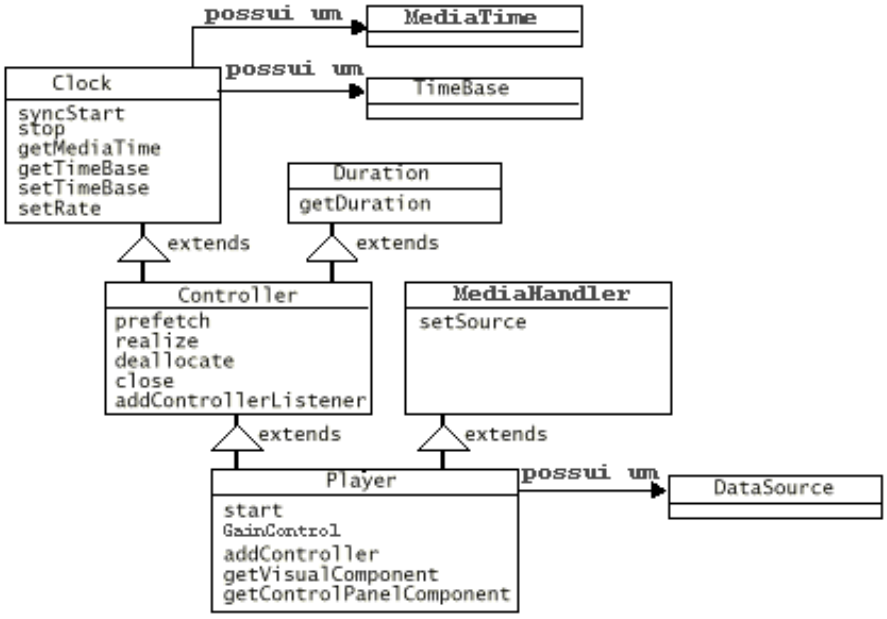


Figura 5.27 - Hierarquia das interfaces associadas ao *Player*

Antes que um *Player* possa exibir os dados recebidos, algumas operações devem ser realizadas. Pelo fato de muitas delas serem assíncronas e por ocorrem em intervalos indeterminísticos de tempo, o *framework* faz com que seja possível controlá-las, definindo um conjunto de estados operacionais para um *Player* e provendo um mecanismo para movê-lo por estes estados (métodos *start*, *prefetch*, *realize*, *stop*).

A interface *Clock* define os primeiros dois estados: *started* e *stopped*. Em seguida, o estado *stopped* é divido em três outros - *unrealized*, *realized* e *prefetched* - pelo Controller. Por fim, como as transições para *realized* e *prefetched* são assíncronas, são definidos dois estados intermediários: *realizing* e *prefetching*. A seguir, uma descrição sucinta desses estados é apresentada:

1. **Unrealized:** quando um *controller* é criado, ele se encontra neste estado. Ele não sabe o tipo de dado que irá processar nem os recursos que serão necessários;
2. **Realized:** neste estado ele determina os recursos necessários e adquire aqueles recursos que não requerem exclusividade. Por possuir informações suficientes sobre a mídia a ser exibida, após esse estado, um *Player* pode disponibilizar à aplicação os componentes de interface adequados, como por exemplo, uma janela visual na qual um vídeo pode ser apresentado e/ou um painel de interação;
3. **Prefetched:** na transição entre *realized* e *prefetched*, o *controller* adquire os recursos exclusivos mínimos para que a exibição da mídia possa ter início. Quando o *Player* atinge este estado ele está pronto para iniciar a exibição em um tempo de latência inicial conhecido;
4. **Started:** mídia em execução. Como a exibição de uma mídia é geralmente iniciada sem que todo seu conteúdo tenha sido recebido, um *Player* em execução pode voltar ao estado de *prefetching*, após um evento que tenha gerado uma inanição de dados (*DataStarvation*). Exemplos de situações capazes de gerar eventos deste tipo são: (i) comando de reposicionamento do usuário, acessando uma posição da mídia que ainda não tenha sido recebida; (ii) taxa de transferência de dados na rede é menor que a taxa de exibição;
5. **Stopped:** execução interrompida, não iniciada ou finalizada.

O processo de transição entre estes estados possui restrições de natureza complexa, como por exemplo a opcionalidade de algumas transições e a existência de métodos que só podem ser invocados em determinados estados, sob a pena de ocasionar erros e exceções. Desta forma, o *framework* utiliza um protocolo essencial de notificação que envia para os objetos interessados mensagens de exceções, erros e de transições de estados (ex.*RealizeComplete*, *PrefetchComplete*, entre outros), além de mensagens que dizem respeito ao estado da mídia exibida (ex. *endOfMedia* e *DataStarved*), como mostra a figura 5.28. Este mecanismo faz com que a aplicação possa sempre garantir que um *Player* se encontra em um estado apropriado antes de invocar um serviço.

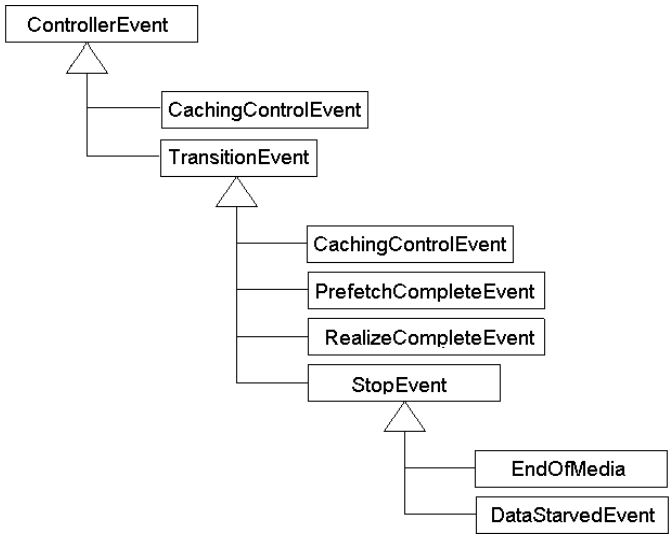


Figura 5.28 - Eventos gerados por um Controller (Player)

A figura 5.29 ilustra este mecanismo de notificação. Um objeto interessado em receber as notificações deve implementar a interface *ControllerListener* e, em seguida, ser adicionado à fila de interessados do *Player*. A partir de então, sempre que um evento ocorrer, o *Player* notifica cada um dos objetos interessados, enviando a mensagem *ControllerUpdate*.

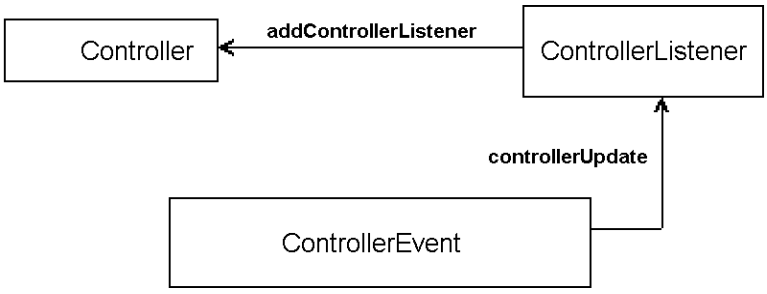


Figura 5.29 - Mecanismo de notificação de um Controller (Player)

Com a utilização desse *framework*, grande parte dos problemas de design de sistemas multimídia distribuídos discutidos na seção 5.2 é automaticamente endereçada, simplificando substancialmente o trabalho de formalização necessário para sua resolução. No entanto, é fundamental que os objetos que se relacionam com o *Player* tenham sempre conhecimento do estado em que ele se encontra, e organizem de forma ordenada o acesso a seus métodos, a fim de eliminar a possibilidade de condições de corrida (*race conditions*). Por este motivo, a interação entre as classes *PlaybackManager* e *Player* é devidamente formalizada e simulada, conforme definido pelo processo de desenvolvimento instanciado na seção 5.2.2. Para a tradução de modelos de projeto orientados a objetos em diagramas SDL, foi utilizado o processo proposto em (Verschaeve et al., 1996). Dessa forma, foram gerados os diagramas no nível de sistema[20] (figura 5.30) para o *JPlayer*, e em nível de bloco para o *PlaybackManager* (figura 5.31) e *Player* (figura 5.32). Além disso, foram especificados os canais de comunicação contendo as mensagens válidas a serem trocadas por esses objetos em cada um das direções, bem como os diagramas em nível de processo contendo as máquinas de estados dos objetos *PlaybackManager* (figura 5.33) e *Player* (figura 5.34). A natureza formal da linguagem permite a execução de simulações, nas quais podem ser observadas as trocas de mensagens entre os processos ativos e como cada máquina de estados reage aos estímulos recebidos. O resultado de uma simulação realizado é mostrado na forma de um MSC na figura 5.35. Nessa figura, a entidade **env** (*environment*) representa o ambiente externo do sistema sendo simulado.

[20] O nível de sistema citado aqui diz respeito à divisão entre níveis de sistema, bloco e processo feita pela linguagem SDL e não possuem nenhuma relação com os níveis de gerência e sistema citados anteriormente.

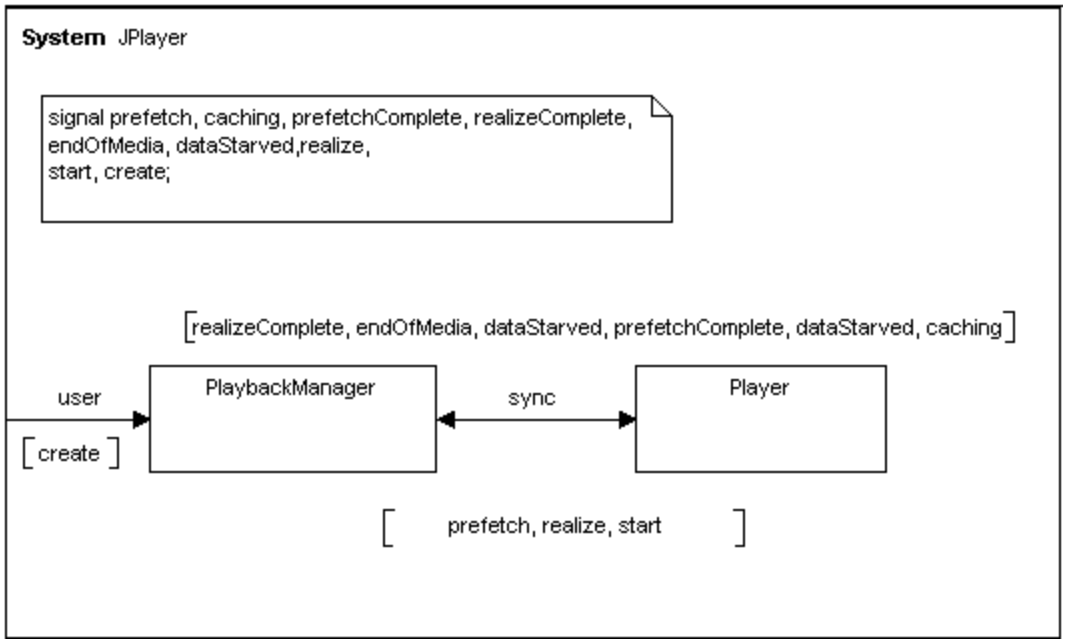


Figura 5.30 - Diagrama de sistema do objeto *JPlayer*

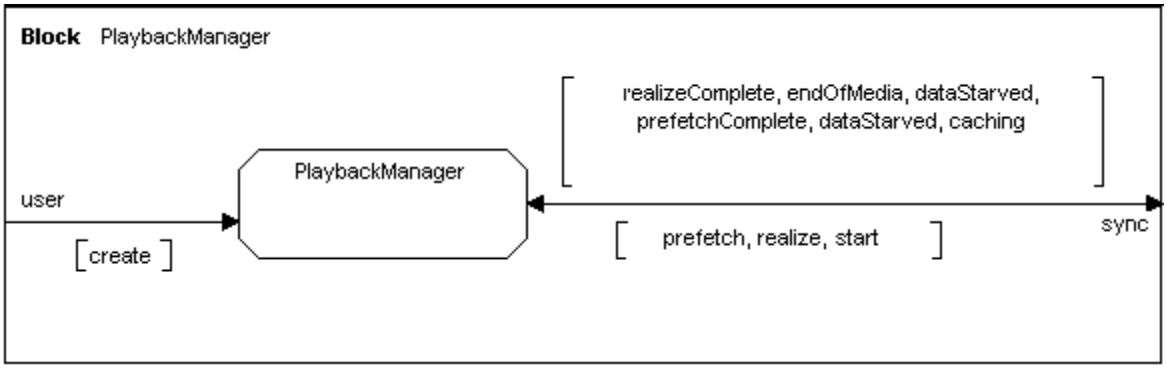


Figura 5.31 - Diagrama de bloco do objeto *PlaybackManager*

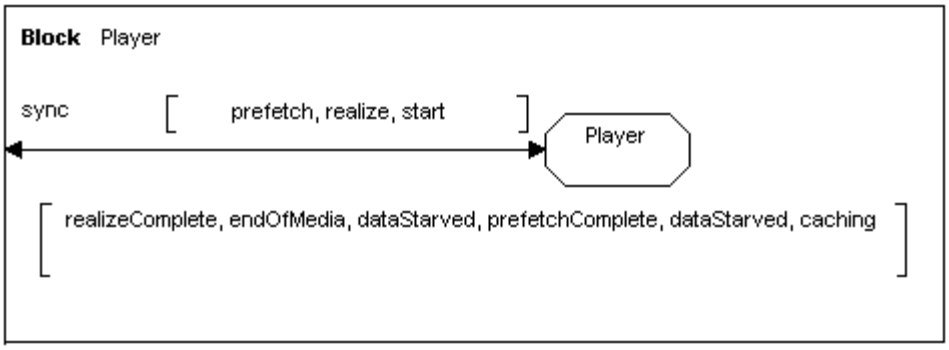


Figura 5.32 - Diagrama de bloco do objeto *Player*

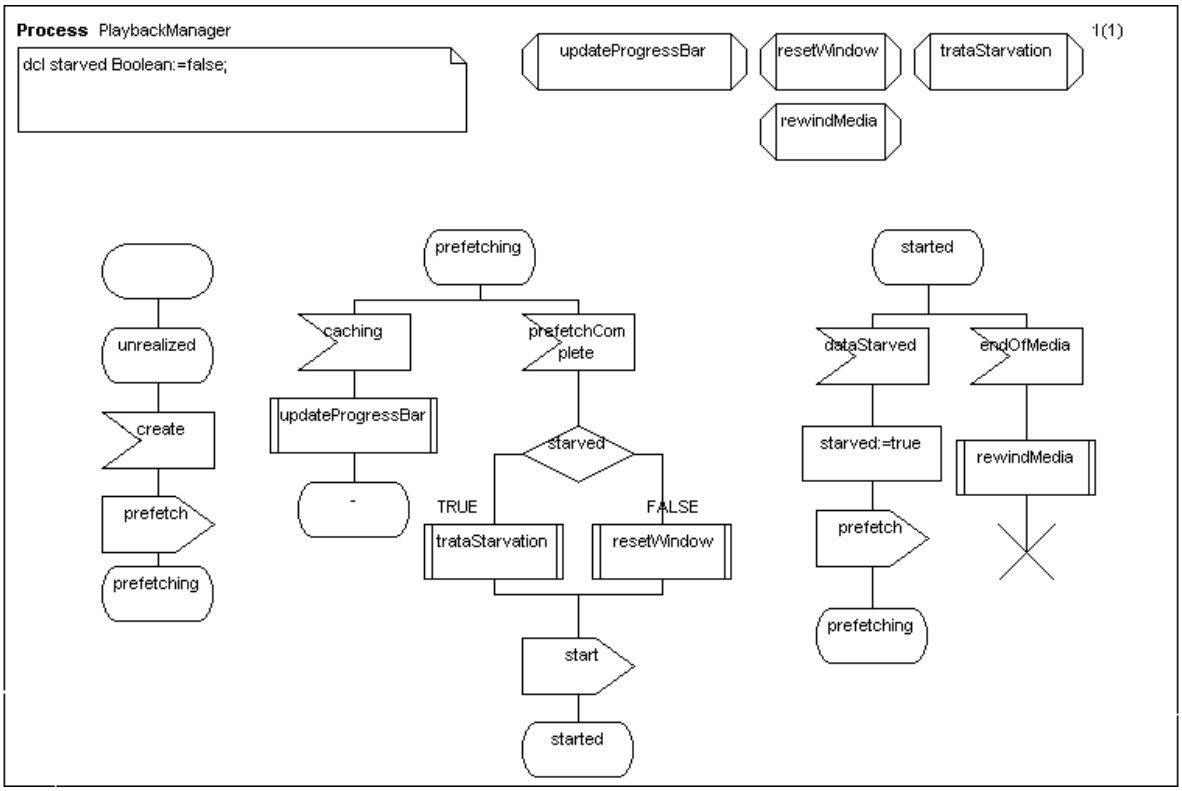


Figura 5.33 - Diagrama de processo do objeto *PlaybackManager*

Como citado anteriormente, o processo de aquisição dos dados necessários para dar início à apresentação (*prefetch*) é assíncrono e não-determinístico. Dessa forma, uma vez dado o comando de início de sessão, a apresentação das mídias pode demorar um tempo maior que o esperado pelo usuário, principalmente se o vídeo escolhido pertencer ao seu catálogo *off-line*. Para abordar esse problema, o objeto *Player* notifica o

PlaybackManager o progresso da operação de aquisição dessas mídias, através de mensagens do tipo *caching* (figura 5.33). O *PlaybackManager*, então, delega ao *InteractionPanel* - através do procedimento updateProgressBar - a tarefa de reportar ao usuário o estado da operação (figura 5.36).

É importante salientar que o *PlaybackManager* atinge o estado de *prefetch* assim que o conteúdo adquirido for suficiente para começar a apresentação, ou seja, o usuário não precisa esperar que as mídias sejam totalmente transportadas para seu terminal para que a apresentação tenha início. Assim que isso acontece, o procedimento *resetWindow* é executado, requisitando ao *InteractionPanel* a transformação da janela de interação, a fim de prover os comandos necessários para que o usuário possa assistir e interagir com o vídeo (figura 5.37).

Finalmente, assim que a apresentação termina, as mídias que compõem o vídeo são automaticamente retornadas ao início pelo procedimento *rewindMedia* (figura 5.33) para que o usuário possa reiniciar a apresentação.

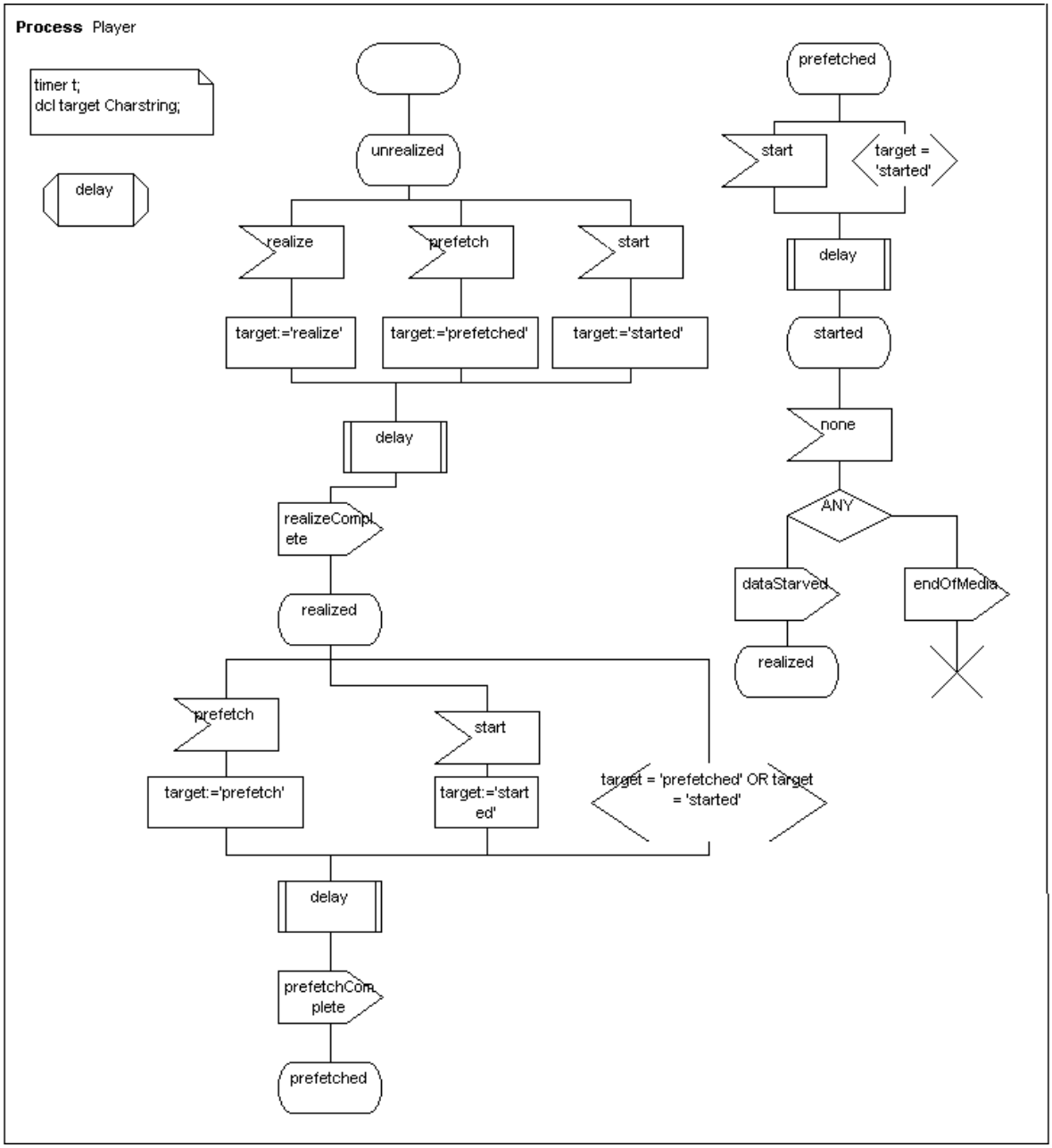


Figura 5.34 - Diagrama de processo do objeto *Player*

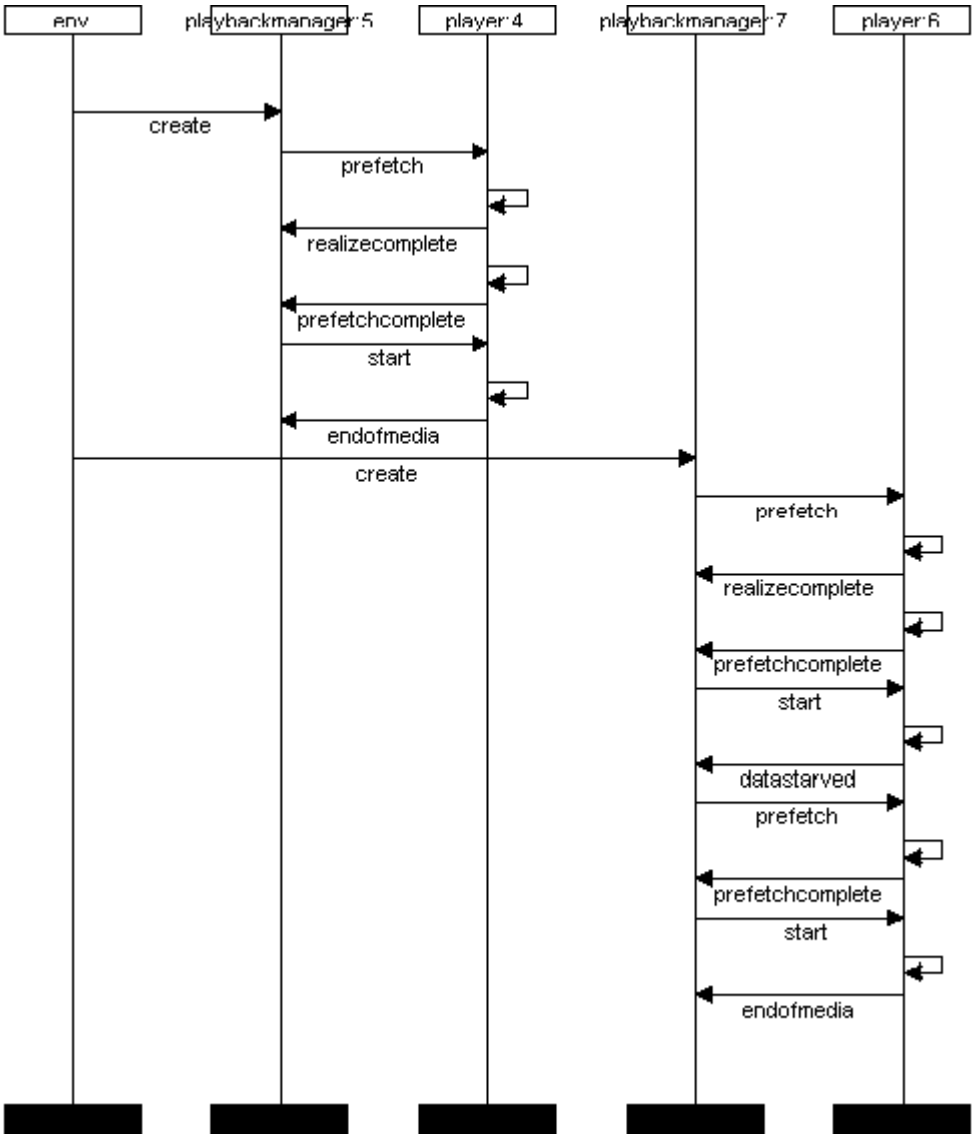


Figura 5.35 - Diagrama de MSC ilustrando a simulação de interações entre os objetos *PlaybackManager* e *Player*

Figura 5.36 – Janela de informação e controle do processo de aquisição de dados

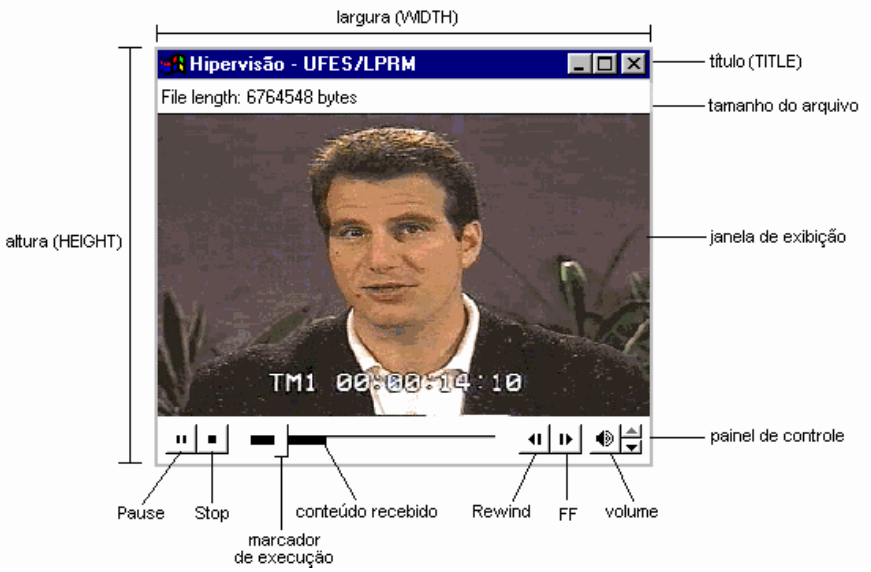


Figura 5.37 – Janela de apresentação e interação

5.6 - Conclusões

Este capítulo apresentou um estudo de caso para a metodologia de desenvolvimento para/com reuso proposta neste trabalho, agrupando experimentações

individuais da aplicação de cada uma de suas partes. Primeiramente, o modelo de processos proposto foi instanciado para o desenvolvimento de uma aplicação de Vídeo sob Demanda. O modelo instanciado associa, de forma complementar, atividades, métodos e técnicas das áreas de inteligência artificial, métodos formais e engenharia de software orientada a objetos, formando um processo de desenvolvimento com múltiplos formalismos, capaz de suprir algumas demandas não atendidas pelos métodos atualmente existentes para essa classe de aplicações.

Em seguida, como experimentação da abordagem de Engenharia de Domínio proposta, uma ontologia de gerência de Vídeo sob Demanda foi construída e o *framework* correspondente foi derivado. A metodologia se mostrou eficaz, capturando o conhecimento do domínio, sem impor muitos compromissos ontológicos, e gerando uma infra-estrutura de objetos, capazes de responder às questões de competência levantadas.

Por fim, a análise de uma aplicação específica de Vídeo sob Demanda foi feita, identificando-se atores e casos de uso, além de restrições e questões de competência específicas do nível de aplicação. Com essa atividade, foi identificada a adequação de dois componentes existentes, passíveis de serem reutilizados: o *framework* de gerência, desenvolvido na seção 5.4, e um outro componente chamado *Java Media Framework*.

O desenvolvimento desse sistema constitui tanto um abrangente estudo de caso de desenvolvimento com reuso, quanto um exemplo de integração de Técnicas de Descrição Formal em um processo de desenvolvimento orientado a objetos. Isso se deve a três fatores: a especialização do *framework* de gerência (um *framework* caixa branca); a utilização do *JMF* (um *framework* caixa preta) e a realização das atividades propostas no processo instanciado

Por fim, é importante ressaltar a adequação do *framework* JMF como componente de reuso, capaz de encapsular, de forma flexível, soluções para grande parte dos problemas de projeto existentes nos sistemas multimídia distribuídos. Neste trabalho, o reuso do componente JMF facilitou substancialmente o processo de formalização necessário ao desenvolvimento da aplicação. Esse processo também foi facilitado pela escolha das TDFs adotadas (SDL/MSC), principalmente pelo fato de utilizarem conceitos comuns à orientação a objetos.

Capítulo 6

Conclusões e Trabalhos Futuros

6.1 - Considerações Finais

Este trabalho tem, como meta inicial, promover a discussão sobre os principais aspectos relacionados ao desenvolvimento ***para*** e ***com reuso***, focando principalmente os fatores responsáveis por sua fraca adoção nos processos de desenvolvimento vigentes. A análise desses fatores, assim como o estudo das características fundamentais dos principais componentes de reutilização, levaram à definição de um modelo de processo genérico de desenvolvimento *para* e com *reuso*, que associa, de maneira complementar, as disciplinas de ***engenharia de software*** e ***engenharia de domínio***.

Para grande parte das disciplinas de engenharia, é comum a idéia de sistemas abertos e interoperáveis, bem como a prática sistemática da reutilização e o uso de modelos fundamentados em teorias matemáticas. Esses conceitos, entretanto, não são difundidos nas práticas de engenharia de software. Neste trabalho, busca-se por uma abordagem madura e bem estruturada de desenvolvimento de software, considerando esses conceitos fundamentais. Desse modo, todas essas disciplinas empregadas no trabalho são descritas como arquiteturas metodológicas, compostas de ferramentas, métodos, atividades, artefatos e insumos, definidos com o intuito de cumprir uma meta previamente negociada de qualidade.

É importante salientar, por outro lado, que o emprego desses conceitos em uma escala industrial de desenvolvimento está condicionado à construção de ambientes de desenvolvimento que auxiliem o desenvolvedor em diversas tarefas, como: a localização e a compreensão dos componentes a serem utilizados, a identificação de oportunidades de reuso, a classificação e a catalogação dos componentes desenvolvidos para reuso. Esses

ambientes devem também conduzir, de forma organizada, os processos de desenvolvimento empregados, integrando ferramentas de validação, verificação, documentação e avaliação. Além disso, sempre que possível, o ambiente deve promover a transição semi-automática entre os modelos gerados pelas fases do processo.

Observa-se que, em geral, compreende-se por reuso a reutilização apenas de código. Este trabalho enfatiza a possibilidade de reusar artefatos mais abstratos como, por exemplo, soluções de projeto, infra-estruturas de domínio e, idealmente, estruturas de conhecimento.

Se comparada com o reuso de componentes de código, essa abordagem de reutilização insere, no processo, inúmeras vantagens, sendo as principais delas ligadas à comunicação e ao entendimento do domínio em questão, à captura da experiência de casos similares e, no caso de um processo formal de desenvolvimento, à obtenção de uma estrutura estável, confiável, robusta e com alto grau de mantenabilidade.

Atualmente, a grande maioria das experiências de construção de infra-estruturas de domínio tem sido realizada em domínios horizontais, estáveis e bem conhecidos, como, por exemplo, os domínios de interface gráfica com o usuário (AWT) e de comunicação remota de objetos (DCOM). O desenvolvimento dessas estruturas em domínios verticais, como a medicina, o direito, e as empresas virtuais impõe inúmeras outras dificuldades, que devem ser tratadas por um método bem estruturado de ***engenharia de domínio***. Este trabalho segue a abordagem da divisão da engenharia de domínio nas fases de ***análise de domínio*** e ***projeto de domínio***.

Essa abordagem se mostrou muito adequada, primeiro porque os modelos de domínio tendem a ser muito mais estáveis que as infra-estruturas correspondentes. Isso se deve ao fato de que, domínios presentes no mundo real mudam de forma gradual e monotônica, enquanto infra-estruturas computacionais são mais sensíveis à influência das aplicações nas quais são integradas e às mudanças de caráter tecnológico. Em segundo lugar, a divisão nessas fases é importante para que os artefatos produzidos em cada uma delas sejam utilizados com diferentes propósitos, por diferentes fases de um processo de engenharia de software. Um exemplo disso é o uso do modelo de domínio na fase de análise da aplicação específica, o qual ajuda na compreensão e na comunicação acerca do

domínio e, conseqüentemente, contribui fortemente para a elicitação dos requisitos da aplicação.

A ***análise de domínio*** tem, como objetivo, a produção de um modelo de domínio capaz de identificar, capturar e organizar os elementos relevantes à representação do conhecimento embutido em uma classe de problemas. Este trabalho promove uma discussão sobre os principais conceitos que definem essa atividade, promovendo, através da análise dos seus métodos mais difundidos, a elaboração de um processo comum de desenvolvimento. Durante essa análise, foram verificadas sérias limitações nesses métodos, principalmente no que diz respeito à capacidade de representação de conhecimento dos modelos gerados. Por atuarem somente em um nível epistemológico, esses modelos são incapazes de representar, de forma explícita, questões pertencentes a um nível de significação e, muito menos, de derivar novos axiomas a partir de um conhecimento factual representado.

Desse modo, é apresentada uma nova abordagem para a realização dessa atividade, fundamentada em um processo sistemático de construção de ontologias formais de domínio. Seguindo a filosofia defendida por este trabalho, o processo apresentado conta com atividades, métodos, técnicas e ferramentas, arranjadas em um ciclo iterativo de desenvolvimento. Além disso, são discutidas questões relativas aos padrões de documentação e à definição de métricas claras para a avaliação da qualidade.

A abordagem de construção de ontologias proposta produz um modelo de domínio, composto por uma taxonomia de conceitos, um vocabulário de termos e, principalmente, de uma teoria lógica, que versa sobre esse universo de discurso. Por esse motivo, essa abordagem pode satisfazer às responsabilidades delegadas à análise de domínio. Além disso, como conseqüência do formalismo empregado, muitos são os benefícios alcançados, como, por exemplo, a verificação/validação automática do modelo de conhecimento construído, a interpretação não ambígua das definições dos elementos que a compõem, e a possibilidade de uma transição sistemática (e idealmente automática) para a fase de projeto de domínio.

Apesar de serem usadas a vários séculos em áreas como filosofia e lingüística, o uso de ontologias é relativamente novo dentro da ciência da computação. Como conseqüência dessa pouca maturidade, algumas questões ainda precisam ser abordadas

como: a necessidade de trabalhos enfocando a integração, a extensão e a evolução de ontologias, e a carência de ferramentas computacionais que automatizem suas atividades.

Para a representação e a formalização da ontologia, é utilizada uma abordagem composta por uma linguagem gráfica (LINGO), para descrever sua estrutura, e uma linguagem textual formal, para descrever seus axiomas. A linguagem gráfica adotada se mostrou uma excelente escolha. Primeiramente, por ser uma proposta aberta, foi possível a inclusão de novas primitivas na linguagem, tais como, a notação de representação de propriedades e a relação de composição. Em segundo lugar, por ter sido criada para a representação de ontologias, foi possível modelar os elementos da conceituação, sem a introdução de compromissos ontológicos desnecessários. Por fim, apesar de ser uma linguagem de nível epistemológico, LINGO provê um mecanismo de inclusão de teorias em um nível ontológico, por isso, toda vez que as suas primitivas são usadas, são gerados, automaticamente, os axiomas da teoria subjacente.

Com o fim da fase de análise de domínio, a ontologia produzida deve servir de base para a construção de infra-estruturas computacionais, passíveis de serem reutilizadas por atividades da engenharia de software. No contexto deste trabalho, essa infra-estrutura é materializada sob a forma de um *framework* orientado a objetos. Desse modo, a linguagem de formalismo lógico proposta é orientada a esse objetivo. Com a inclusão de uma fundamentação de teoria de conjuntos, a linguagem se torna menos abstrata, porém serve como importante ferramenta de modelagem para sistematizar a tradução de estruturas conceituais da ontologia em elementos do paradigma da orientação a objetos.

De maneira geral, os objetivos da fase de projeto de domínio são: a escolha de que elementos do modelo de domínio deverão aparecer na infra-estrutura computacional, a decisão de como isso será feito e a realização desse processo. A fim de prover uma abordagem de sistematização dessas tarefas, propõe-se um conjunto de diretivas de mapeamento da estrutura da ontologia (conceitos, relações e axiomas epistemológicos) em elementos do *framework* (classes, relacionamentos e padrões de projeto), bem como um conjunto de regras de transformação para a geração automática de invariantes a partir de axiomas ontológicos. São apresentados, ainda, dois padrões de projeto. O primeiro é responsável por assegurar a verificação de pré-condições derivadas dos axiomas de consolidação (Padrão Pré-Condição), e o segundo, derivado a partir desse, provê a

verificação dos axiomas que descrevem a teoria subjacente à relação todo-parte (Padrão Todo-Parte). Por fim, para servir de infra-estrutura de implementação nesse processo, é implementado um *mini-framework*, que materializa as operações do tipo conjunto (Set). Através do uso do mecanismo de reflexão computacional, esse componente se torna genérico o suficiente para ser utilizado na realização de várias outras tarefas, como, por exemplo, na definição de uma nova abordagem para a implementação da persistência de objetos.

Como um *framework* é também um artefato de código, a fase de projeto de domínio provê atividades de projeto de software, para tonar a sua estrutura mais estável e buscar atingir os requisitos de adaptabilidade defendidos na Introdução deste trabalho. Esse fato também impõe a necessidade da escolha de uma linguagem de implementação. Apesar da linguagem adotada (JAVA) ter sido bastante satisfatória, a abordagem de projeto de domínio proposta pode ser redefinida para outras linguagens, desde que possuam as seguintes características: a possibilidade de implementação de estruturas genéricas e polimórficas, a checagem forte de tipos e o suporte a mecanismos de reflexão computacional (meta-classes).

Por fim, o domínio de Vídeo sob Demanda é escolhido, a fim de servir como estudo de caso das várias propostas apresentadas durante essa dissertação. Primeiramente, são identificados os requisitos específicos dessa classe de aplicações, para que um processo de desenvolvimento adequado possa ser definido, o que é feito a partir da intanciação do modelo genérico de desenvolvimento para e com reuso proposto. Em seguida, como estudo de caso de desenvolvimento para reuso, uma ontologia de gerência de Vídeo sob Demanda é desenvolvida e o *framework* correspondente derivado. A partir do reuso dessa infra-estrutura, bem como de outros componentes passíveis de reutilização, construiu-se então uma aplicação nesse domínio.

O domínio de Vídeo sob Demanda foi escolhido por diversas razões, entre elas: o seu enorme potencial de crescimento como serviço, a sua possibilidade de aplicação em diversas áreas, que vão desde entretenimento até tele-medicina e, principalmente, pela diversidade de seus requisitos, o que a tornou bastante adequada para a aplicação da metodologia.

6.2 - Trabalhos Futuros

As perspectivas de trabalhos futuros originados a partir desse trabalho podem ser divididas em três principais áreas:

- ***Modelo de Representação e Derivação de Ontologias:*** Assim como a linguagem LINGO, tanto a linguagem formal proposta, quanto as diretivas, regras de transformação e padrões de projeto, devem ser consideradas propostas abertas e passíveis de extensão. Um abrangente campo de estudos nessa direção é a extensão desses modelos para que possam tratar questões relativas ao comportamento temporal dos elementos do domínio, bem como a interação entre eles. Um outra linha promissora de trabalho é o estudo da integração de ontologias de domínio e ontologias de tarefas, visando tornar mais abrangente o processo de especialização de aplicações orientadas a objetos.

- ***Construção de Ferramentas e Ambientes de Desenvolvimento:*** Como citado anteriormente, a adoção efetiva das propostas apresentadas nesse trabalho, dependem da existência de ferramentas computacionais, capazes de automatizar o seu processo de desenvolvimento e abrandar a complexidade envolvida na realização de suas atividades. Um trabalho especialmente interessante, é a construção de um ambiente de desenvolvimento para a fase de engenharia de domínio. A seguir são enumeradas algumas das principais ferramentas que devem estar contidas nesse ambiente: (1) Uma ferramenta gráfica para construção de diagramas em LINGO capaz de produzir de forma automática os axiomas gerados pela utilização de suas primitivas; (2) Suporte à descrição dos axiomas de derivação e consolidação da ontologia; (3) Suporte à definição da intenção de conceitos e relações; (4) Um provador automático de teoremas capaz de verificar a consistência do conjunto de axiomas e de validar as questões de competência da ontologia; (5) Uma ferramenta capaz de produzir, em formato hipertextual, a documentação referente ao vocabulário de termos da ontologia; (6) Suporte aos mecanismos de derivação da estrutura do framework, bem como à geração das invariantes e pré-condições presentes em seus métodos.

- ***Aplicação de Vídeo sob Demanda:*** O sistema Hipervisão, utilizado como estudo de caso deste trabalho, teve a sua implementação concluída, e vem sendo alvo de investigação de diversos outros trabalhos acadêmicos. Dois desses trabalhos propuseram melhorias ao sistema, através da substituição dos protocolos de transporte e iteração de mídias contínuas (Berger, 1999). Em outro trabalho, o sistema foi utilizado como base para a proposta de soluções de problemas de deseonvimento de Sistemas Tutores Inteligentes no contexto da internet (Guizzardi et al, 1999a). Por fim, os trabalhos descritos em (Almeida, 1999, 2000) constroem sobre ontologia de gerência de vídeo sob demanda desenvolvida nesse trabalho, um cenário de comércio eletrônico mediado por agentes.

Referências Bibliográficas

1. ALBERTS L. K. **YMIR: a sharable ontology for the formal representation of engineering design knowledge**, Formal Design Methods for Computer-Aided Design, Elsevier/IFIP, 1994.
2. ALEXANDER, C. et al. **A Pattern Language – Towns, Buildings, Construction**. Oxford University Press, 1977.
3. ALEXANDER, C. **The Timeless Way of Building**. Oxford University Press, 1979.
4. ALMEIDA, J. P. A., GUIZZARDI, G., GONÇALVES, J. An Architecture for Video on Demand Agent-Mediated Electronic Commerce. In: WORKSHOP DE COMPUTAÇÃO, São José dos Campos, 1999. **Anais…** José dos Campos, 1999.
5. ALMEIDA, J. P. A., GUIZZARDI, G., GONÇALVES, J. G. Agent-Mediators in Media-On-Demand Electronic Commerce. In: INTERNATIONAL CONGRESS OF NEW TECHNOLOGIES AND COMPUTER APPLICATIONS, 7., Cuba, 2000. **Anais…** Cuba, 2000
6. ANSI/IEEE. IEEE Standard VHDL Language Reference Manual, Std.1076-1993, 1993.
7. ARANGO, G. **Domain Analysis Methods**, In: WORKSHOP ON SOFTWARE ARCHITECTURE, 1994, Los Angeles. **Anais…** Los Angeles: USC Center for Software Engineering, 1994.
8. ARANGO, G., PRIETO-DÍAZ, R. Domain Analysis Concepts and Research Directions. In: WORKSHOP ON SOFTWARE ARCHITECTURE, 1994, USC Center for Software Engineering, Los Angeles, EUA. **Anais…** Los Angeles,1994.
9. ARAÚJO JUNIOR, J., SAWYER, P. Integrating Object-Oriented Analysis and Formal Specification. **Journal of the Brazilian Computer Society**. v.1, n. 5, 1998.
10. ARTALE, A. et al. Part-Whole Relations in Object-Centered Formalisms: an Overview. Data and Knowledge Engineering, n. 20, 1996.

11. AUCHTER, D. **From Requirements Capture to Design**: Combining the ARENA and SOMT method. 1997. Relatório Técnico – Advanced Software Technology Competence Center (ASTEC), Suécia.
12. BATEMAN, J. A. et al. The Generalized Italian, German, English Upper Model. In: WORKSHOP COMPARISON OF IMPLEMENTED ONTOLOGIES, 1994, Amsterdã. **Anais…**Amsterdã: N.J.I. Mars, 1994.
13. BEN-NATAN, R. **Objects on the Web.** McGraw-Hill, 1997.
14. BERGER, V. **Avaliação do RTSP como alternativa ao protocolo HTTP-Streaming na transmissão de mídias contínuas com restrições de Tempo Real em redes IP**. Projeto Final de Graduação (Bacharelado em Ciência da Computação), Departamento de Informática, Universidade Federal do Espírito Santo, 1999.
15. BOLOGNESI, T., LAGEMAAT, J. V., VISSERS, C. **LOTOSPHERE**: Software Developtment with LOTOS. Kluwer Academic Publishers, The Netherlands, 1995.
16. BOWAN, J. P., HINCHLEY, M. G. **Ten Commandments of Formal Methods**.1994. Relatório Técnico nº 350 – Universidade de Cambridge, Cambridge, UK.
17. BREUKER, J., VAN DE VELDE, W., 1994, **CommonKADS Library for Expertise** *Modelling*, IOS Press.
18. BUSHMAN, F. et al. **Pattern-Oriented Software Architecture: A System of Patterns.** Willey, West Sussex, Inglaterra, 1996.
19. CECILIO, E. L., RODRIGUES, R. F. **Televisão de Alta Definição (HDTV).** 1996. Relatório Técnico TM10 – Pontifícia Universidade Católica do Rio de Janeiro.
20. CECILIO, E. L., RODRIGUES, R. F. **Video sob demanda**. 1996. Relatório Técnico TM10 – Pontifícia Universidade Católica do Rio de Janeiro.
21. CHANDRASEKARAN, B., JOSEPHSON, J. R. **The Ontology of Tasks and Methods.** 1997. Stanford University, California.
22. CHERVENAK, A.L. **Tertiary Storage**: An evolution of new applications. 1994. Tese (Doutorado) – Universidade da Califórnia.
23. CIMA, A. M., WERNER, C. M. L. **A Reutilização de Software e a Orientação a Objetos.** 1997. Relatório Técnico ES-433/97 – Engenharia da Sistemas e Computação, COPPE, Universidade Federal do Rio de Janeiro.

24. CLANCEY, W. J. The knowledge level reinterpreted: modelling socio-technical systems. **International Journal of Intelligent Systems**, 1993.
25. COAD, P., YOURDON, E. **Análise Baseada em Objetos**. Editora Campus, 1992.
26. CORNWELL, P.C. HP Domain Analysis: Producing Useful Models for Reusable Software. **Hewlett-Packard Journal**, 1996.
27. DAHLGREN, K. A Linguistic Ontology. **International Journal of Human-Computer Studies**, v. 43, 1995.
28. DIAZ, M., PAYS, G. "The Cesame project: formal design of high speed multimedia cooperatie systems", **Ann. Télécommun.**, 1994, 49(5-6), pp. 220-229.
29. DIGITAL Audio Visual Council, DAVIC 1.4 Specifications. [on-line] Disponível: http://www.davic.org [capturado em 1998].
30. EILLEY, E. S., SNIJDERS, W. A. M., MA, C. Specification of Component Systems Architectures, RACE Project, DIAMOND - Distributed IBC Applications for Multimedia on Demand Deliverable no. 10, 1994.
31. FALBO R. A. **Integração de Conhecimento em um Ambiente de Engenharia de Software**. 1998. Tese (Doutorado em Informática) - Engenharia de Sistemas e Computação, COPPE, Universidade Federal do Rio de Janeiro.
32. FALBO, R.A., MENEZES, C. S., ROCHA, A. R. C. A Systematic Approach for Building Ontologies. In: IBERO-AMERICAN CONFERENCE ON ARTIFICIAL INTELLIGENCE, 6, 1998, Lisboa. **Anais...** Lisboa: H. Coelho, 1998,
33. FÉRNANDEZ, M., GÓMEZ-PÉREZ, A., JURISTO, N. **METHONTOLOGY: From Ontological Art Towards Ontological Engineering.** 1997. Ontological Engineering - Working Notes, Stanford University, California.
34. FOWLER, M. **UML Distilled**. Addison-Wesley, 1997. 183 p.
35. FRANCIS, F. et al. **Active Server Pages 2.0**. Wrox Press, 1998.
36. GALL, H., JAZAYERI, M., HÖSCH, R. Research Directions in Software Reuse: Where to go from here? In: SSR'95 - ACM SYMPOSIUM ON SOFTWARE REUSABILITY, 1995, Seattle, EUA. **Anais…** Seattle, 1995.
37. GAMMA, E. et al. **Design Patterns**: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.
38. GIBBS, W. W. Software's Chronic Crisis. **Scientific American**, set. 1994.

39. GÓMEZ-PÉREZ, A., FERNÁNDEZ, M., VICENTE, A. J. Towards a Method to Conceptualize Domain Ontologies. In: ECAI'96 - WORKSHOP ON ONTOLOGICAL ENGINEERING, 1996, Budapest. **Anais…** Budapeste, 1996.
40. GONÇALVES, J. **Um Sistema para Especificação de Apresentações Multimídia Distribuídas numa Arquitetura de Comunicação Integrada**. 1996. Tese (Doutorado em Engenharia Elétrica) – Escola Politécnica da Universidade de São Paulo.
41. GRISS, M. et al. Systematic Software Reuse: Objects and Frameworks are not enough. In: SSR'95 - ACM SYMPOSIUM ON SOFTWARE REUSABILITY, 1995, Seattle, EUA. **Anais…** Seattle, 1995.
42. GRUBER, T. R. **Ontolingua: A mechanism to support portable ontologies, version 3.0.** 1992. Relatório Técnico. Knowledge Systems Laboratory, Stanford University, California.
43. GRUBER, T. R. Toward Principles for the Design of Ontologies used for Knowledge Sharing**. Int. J. Human-Computer Studies**, v. 43, n. 5/6, 1995.
44. GRUNINGER M. **Integrated Ontologies for Enterprise Modelling** [online]. Disponível: http://www.ie.utoronto.ca/EIL/tove/ontoTOC.html [capturado em 05 de jan. 2000].
45. GUARINO, N. The Ontological Level. In: WITTGENSTEIN SYMPOSIUM, 4., Kirchberg, Austria, 1993. Anais… Kirchberg, Austria, 1993
46. GUARINO, N. Open Problems for Part-Whole Relations. In: INTERNATIONAL WORKSHOP ON DESCRIPTION LOGICS, Boston/MA, 1996. **Anais…** Boston/MA, 1996.
47. GUARINO, N. Understanding, building and using ontologies. **Int. Journal Human-Computer Studies**, v. 45, n. 2/3, fev./mar. 1997.
48. GUARINO, N. Formal Ontologies and Information Systems. In: FIRST INTERNATIONAL CONFERENCE (FOIS), 1., 1998, Trento, Itália. **Anais...** Trento: IOS Press, 1998.
49. GUERRIERI, E. et al. The Impact of Java on Software Reusability. In: SSR'97 - ACM SYMPOSIUM ON SOFTWARE REUSABILITY, 1997, Boston, EUA. **Anais…** Boston, 1997.

50. GUIZZARDI, G., GONÇALVES, J. Uma metodologia baseada em objetos para descrição lógica de sistemas de vídeo sob demanda. In: SIMPÓSIO BRASILEIRO DE SISTEMAS MULTIMÍDIA E HIPERMÍDIA, 4., 1998, Rio de Janeiro. **Anais...** Rio de Janeiro, 1998.
51. GUIZZARDI, G., CURY, D., GONÇALVES, J. A framework proposal to hypermedia Intelligent Tutoring Systems development in the Internet. In: SIMPÓSIO BRASILEIRO DE INFORMÁTICA NA EDUCAÇÃO, 10., 1999, Curitiba. **Anais...** Curitiba, 1999.
52. GUIZZARDI, G., GONÇALVES, J. Hipervisão: de uma ontologia de domínio a uma aplicação de vídeo sob demanda. In: SOUZA, W. L. **Projeto DAMD – Design de Aplicações Multimídia Distribuídas** - Livro em fase de impressão, 1999.
53. GUIZZARDI, G., GONÇALVES, J. LogicOO: uma metodologia para modelagem e construção de sistemas multimídia distribuídos. In: WORKSHOP ON REQUIREMENTS ENGINEERING, 2., 1999, Buenos Aires, Argentina. **Anais…** Buenos Aires: editora, 1999.
54. GUIZZARDI, G., GONÇALVES, J. Um framework para modelagem e construção de sistemas de vídeo sob demanda. In: SOUZA, W. L. **Projeto DAMD – Design de Aplicações Multimídia Distribuídas** - Livro em fase de impressão, 1999.
55. HOLZ, E. et al. Methodology, INSYDE Integrating Method for Evolving Systems Design, CEC ESPRIT III, Ref: P8641, 1996.
56. HORSTMANN, C. S., CORNELL, G. **Core Java**: Fundamentals. 2. Ed.. Sun Press, 1997. 2 v.
57. HUMPHREYS, B. L., LINDBERG, D. A. B. The UMLS project: making the conceptual connection between users and the information they need, **Bulletin of the Medical Library Association**, v. 81, n. 2, 1993.
58. INTERACTIVE Software Engineering, Building bug-free OO software: An introduction to Design by Contract [on-line]. Disponível: http://eiffel.com/doc/manuals/technology/contract/page.html [capturado em 20 de dez. 1999].

59. ISO IS 8807, “Information processing systems - Open Systems Interconnection - LOTOS - A formal description technique based on the temporal ordering of observational behaviour", 1989.
60. ISO IS 9074, "Information processing systems - Open Systems Inteconnection - Estelle - A formal description technique based on an extended state transition model", 1989.
61. ISO 13522-1. Information Technology - Coding of Multimedia and Hypermedia Information - Part 1: MHEG Object Representation, Base Notation (ASN.1), 1994.
62. ITU-T Rec. Z.100. Specification and Description Language (SDL), 1994.
63. ITU-T. Rec. Z.120. Message Sequence Charts (MSC). In Criteria for the use and applicability of formal description techniques, 1994.
64. ITU. SDL Methodology Guidelines, SDL Bibliography, 1994.
65. JACOBSON, I. et al. **Object-Oriented Software Engineering:** A Use Case Driven Approach. Addison-Wesley, 1998. 528 p.
66. JIA, X. **Object-Oriented Software Development Using Java: principles, patterns and frameworks**. Addison-Wesley, 2000.
67. JOHNSON, R. E. Frameworks = (Components + Patterns): How frameworks compare to other object-oriented reuse techniques, **Communications of the ACM**, v. 40, n. 10, 1997.
68. JONCKERS, K. et al. OMT*: Bringing the Gap Between Analysis and Design , INSYDE Integrating Method for Evolving Systems Design, CEC ESPRIT III, Ref: P8641, 1995.
69. KARP P. D. A Qualitative Biochemistry and its Application to the Regulation of the Trytophan Operon, Artificial Intelligence and Molecular Biology, Ed: L.Hunter, Califórnia, AAAI Press, 1993.
70. KNUTH, D. E. **The Art of Computer Programming**. Reading: Addison-Wesley, 1971/1983. 3 v.
71. LANGER, Susanne K. **An Introduction to Symbolic Logic**, 3. ed. New York Dover Publications, 1967.
72. LENAT, D. B. CYC: A Large-Scale Investment in Knowledge Infrastructure. **Communications of the ACM**, n. 38, 1995.

73. LITTLE, T. D. C., VENKATESH, D. Prospects for Interactive Video-on-Demand. **IEEE Multimedia**, v. 1, n. 3, 1994.
74. LIPSCHUTZ, S. **Teoria dos Conjuntos**. McGraw-Hill, 1974.
75. LUBLING, O., RAZORFISH, L.M. **Developing Scalable, Reliable, Business Applications with Servlets** [on-line] Disponível: http://developer.java.sun.com/developer/technicalArticles/Servlets/Razor/index.html [capturado em 30 mar. 2000].
76. MARTIN, J., ODELL, J..J. **Object-Oriented Methods**: Pragmatic Considerations, Englewood Cliffs: Prentice-Hall, 1996.
77. MCILLROY, M. D. Mass-produced Software Components. In: NATO CONFERENCE ON SOFTWARE ENGINEERING, 1968, Garmisch, Alemanha. **Anais…** Garmish, 1968.
78. MCCLUSKEY, G. **Remote Method Invocation:** Creating Distributed Java-to-Java Applications [on-line] Disponível: http://developer.java.sun.com/developer/technicalArticles/RMI/CreatingApps/index.html [capturado em 30 mar. 2000].
79. MCPHERSON, S. **Java Servlets and Serialization with RMI** [on-line] Disponível: http://developer.java.sun.com/developer/technicalArticles/RMI/rmi/ [capturado em 30 mar. 2000].
80. MEEKEL, J. et al. From Domain Models to Architecture Frameworks. In: SSR'97 - ACM SYMPOSIUM ON SOFTWARE REUSABILITY, 1997, Boston, EUA. **Anais…** Boston, 1997.
81. MILLER, G. A. WORDNET: A On-Line Lexical Database. **International Journal of Lexicography**, n. 3/4, 1990.
82. MUSEN, M. A. et al. PROTEGE-II: Computer support for development of intelligent systems from libraries of components. In: MEDINFO'95 - WORLD CONGRESS ON MEDICAL INFORMATICS, 8, 1995, **Anais…**
83. NAHRSTEDT, K., SMITH, J. M. The QoS Broker. **IEEE Multimedia**, 1995.
84. NEIGHBORS J. **Software Construction Using Components**. 1981. Tese (Doutorado) - Universidade da Califórnia, Irvine, EUA.

85. NOY, N. F., HAFNER C.D. The State of Art in Ontology Design: A Survey and Comparative Review. **AI Magazine**, 1997.
86. O'LEARY, D. E. Impediments in the use of explicit ontologies for KBS development. **Int. J. Human-Computer Studies**, v. 46, n. 2/3, 1997.
87. POULIN, J. S. On the Contributions of Reuse Research and Development to the State-of-the-Practice in Reuse. In: SSR'97 - ACM SYMPOSIUM ON SOFTWARE REUSABILITY, 1997, Boston, EUA. **Anais…** Boston, 1997.
88. PRESSMAN, R.S. **Software Engineering: A Practioner's Approach**. 4. ed. McGraw-Hill, 1997.
89. PROFESSIONAL Home Page [on-line] Disponível: http://www.php.net [capturado em 30 mar. 2000].
90. RIX, M. Case Study of a Successful Firmware Reuse Program, In: HP SOFTWARE PRODUCTIVITY CONFERENCE, 1992, **Anais…**
91. ROWE L. et al. MPEG Video in Software: Representation, Transmission and Playback, High Speed Networking and Multimedia Computing. In: IS&T/SPIE Symposium on Eletrical Imaging Science and Technology, San Jose, CA, 1994. **Anais…**, San Jose, CA, 1994.
92. RUMBAUGH J. et al. **Object-Oriented Modelling and Design**. International Editions. Prentice-Hall, 1991.
93. RUSSEL, B. **Principles of Mathematics**. New York: Norton, 1938.
94. RUSSEL, S., NORVIG, P. **Ar tificial Intelligence:** A Modern Approach. Prentice-Hall, 1995. Cap. 8, Building a Knowledge Base.
95. SAMPSON, P., SNIJDERS, W. M., ROSSAVIK, K. Overall system specification and architecture, RACE Project, DIAMOND - Distributed IBC Applications for Multimedia on Demand Deliverable, n. 2, 1997.
96. SAYWOOD, K. **Introduction to Data Compression**. Morgan Kaufmann, 1996.
97. SILBERCHATZ, A. et al. Database System Concepts, 3. ed. McGraw-Hill, 1997.
98. SINCLAIR, D. et al. An Object-Oriented Methodology from Requirements to Validation, INSYDE Integrating Method for Evolving Systems Design, CEC ESPRIT III, Ref: P8641, 1995.

99. SMITH, B. **Ontology: Philosophical and Computational** [online]. Disponível:http://wings.buffalo.edu/philosophy/faculty/smith/articles/ontologies.htm [capturado em 25 de jan. 2000].

100. SOUZA, W. L. et al. **Projeto DAMD – Design de Aplicações Multimídia Distribuídas**, Livro em fase de impressão, 1999.

101. SOWA, J. F. Top-level ontological categories. **International Journal of Human-Computer Studies**, v. 43, 1995.

102. SPIVEY, J. M. **Understanding Z:** A specification language and its formal semantics. Cambridge University Press, 1988.

103. SULLIVAN, S. C. et al. **Programming with the Java Media Framework**. New York: Wiley Books, 1998.

104. TELELOGIC SDT 3.1. Methodology Guidelines Part I: The SOMT Method, Telelogic AB, Malmö, 1996.

105. TRACZ, W. Developing Reusable Java Components. In: SSR'97 - ACM SYMPOSIUM ON SOFTWARE REUSABILITY, 1997, Boston, EUA. **Anais…** Boston, 1997.

106. TROY, R. Software Re-use, In: OBJECTWORLD CONFERENCE, 1993. **Anais…**

107. USCHOLD, M., KING, M. Towards a Methodology for Building Ontologies. In: WORKSHOP ON BASIC ONTOLOGICAL ISSUES IN KNOWLEDGE SHARING,1995.

108. USCHOLD, M., GRUNINGER M. Ontologies: principles, methods and applications. **The Knowledge Engineering Review**, v. 11, n. 2, p. 93-136, 1996.

109. VAN DER VET, P. E., MARS, N. J. I. The Plinius Ontology of Ceramic Materials. In: WORKSHOP COMPARISON OF IMPLEMENTED ONTOLOGIES, 1994, Amsterdã. Amsterdã: N.J.I. Mars, 1994.

110. VAN DER VET, P. E., MARS, N. J. I. **Bottom-up construction of ontologies:** the case of and ontology of pure substances. 1995. Relatório Técnico - Departamento de Ciência da Computação, Universidade de Twente, Holanda.

111. VAREJÃO F. M. **DORPA: Uma ontologia de Design que integra requisitos, artefatos e processos**. 1999. Tese (Doutorado em Informática) – Pontifícia Universidade Católica, Rio de Janeiro.

112. VERSCHAEVE K. et al. Translating OMT* to SDL, Coupling Object-Oriented Analysis with Design, IFIP Methods Engineering'96, 1996.

113. VOSS, G. **JavaServer Technologies** [on-line] Disponível: http://developer.java.sun.com/developer/technicalArticles/Servlets/JavaServerTech1/index.html [capturado em 30 mar. 2000].

114. WOOLF H. B. **Webster's New Collegiate Dictionary**. Springfield. Mass: G&C, Merriam, 1981.

115. WOSOWSKI, M. et al. The Complete OMT*, INSYDE Integrating Method for Evolving Systems Design, CEC ESPRIT III, Ref: P8641, 1996.

116. ZAND, M. et al. Reuse Research and Development: Is it on the right track?. In: SSR'97 - ACM SYMPOSIUM ON SOFTWARE REUSABILITY, 1997, Boston, EUA. **Anais…** Boston, 1997.