



Universidade Estadual de Maringá
Centro de Tecnologia
Departamento de Informática

**Uma Ferramenta para Apoiar a Definição de Requisitos no
Desenvolvimento de Software Distribuído**

Marco Aurélio Graciotto Silva

TG-2001

Maringá - Paraná

Brasil

Universidade Estadual de Maringá
Centro de Tecnologia
Departamento de Informática

Uma Ferramenta para Apoiar a Definição de Requisitos no Desenvolvimento de Software Distribuído

Marco Aurélio Graciotto Silva

TG-2001

Trabalho de Graduação apresentado ao Curso de Ciência da Computação, do Centro de Tecnologia, da Universidade Estadual de Maringá.

Orientadora: *Profa. Dra. Elisa Hatsue Moriya Huzita*

Maringá - PR
2002

Marco Aurélio Graciotto Silva

**Uma Ferramenta para Apoiar a Definição de
Requisitos no Desenvolvimento de Software
Distribuído**

Este exemplar corresponde à redação final da monografia aprovada como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação da Universidade Estadual de Maringá, pela comissão formada pelos professores:

Orientadora: Profa. Dra. Elisa Hatsue Moriya Huzita
Departamento de Informática, CTC, DIN

Profa. Dra. Itana Maria de Souza Gimenes
Departamento de Informática, CTC, DIN

Profa. Dra. Tânia Fátima Calvi Tait
Departamento de Informática, CTC, DIN

Maringá, Maio de 2002.

Universidade Estadual de Maringá

Departamento de Informática

Av. Colombo 5790, Maringá-PR

CEP 87020-900

Tel: (044) 2262727 R. 219/324 Fax: (044) 2232676

Agradecimentos

Aos meus pais, Apolo e Rosa, e irmãos, Melissa e Victor, pelo imprescindível apoio durante toda minha vida.

À profa. Elisa, orientadora de meu Projeto de Iniciação Científica e deste trabalho, pelos incentivos, paciência e compreensão.

Aos professores Flávio, Dino e Maurício, por terem me auxiliado na busca por um programa de mestrado.

Aos integrantes do Departamento de Informática como um todo, por propiciarem infraestrutura para uma boa formação acadêmica.

Aos integrantes da OAE, por compartilharem comigo momentos de alegria (ou não).

A todos os colegas de curso, em especial àqueles que percorreram comigo estes quatro anos e meio de vida acadêmica.

Resumo

As recentes tendências do mercado têm mostrado que a complexidade do software continuará a crescer drasticamente nas próximas décadas. Aliada a isto, a globalização acaba por envolver organizações de diferentes portes, com políticas peculiares de tomadas de decisão. O volume de dados a ser utilizado cresce ao mesmo tempo que temos uma descentralização deste. Neste novo panorama, sistemas isolados, monolíticos, são uma solução pouco eficaz, dando lugar aos sistemas distribuídos. O advento de sistemas distribuídos leva a aplicações mais complexas, implicando que desenvolver software usando métodos tradicionais torna-se ineficiente. A redução desta complexidade pode ser obtida através da decomposição, estruturação e delegação de tarefas, empregando para isto metodologias de desenvolvimento adequadas. A criação de ferramentas que dêem suporte a tais metodologias é desejável. O objetivo deste trabalho é o desenvolvimento de uma ferramenta que auxiliará na definição de requisitos de sistemas distribuídos, a ser utilizada na Metodologia de Desenvolvimento Baseado em Objetos Distribuídos Inteligentes - MDSODI. Foram estudadas várias abordagens que podem ser utilizadas para a definição de requisitos: pontos de vista; uso de padrões na construção de cenários; padrões de requisitos e utilização de modelos para descrição, qualificação, análise e validação de requisitos. Destas, duas foram escolhidas: pontos de vista por possibilitar a rastreabilidade dos casos de uso, ser facilmente utilizados nos diversos processos de engenharia de software e de fácil aplicabilidade; e utilização de modelos para descrição, qualificação, análise e validação de requisito, que possibilitem a classificação dos pontos de vista com base em critérios estabelecidos pelos engenheiros de requisito. O projeto desta ferramenta está documentado em UML e implementado em Java, utilizando o OpenORB como middleware CORBA e o banco de dados PostgreSQL como mecanismo de persistência.

Abstract

The recent market's tendencies has shown that software complexity will keep growing in the next decades. Moreover, the globalization involves organizations of different size, kinds, enterprise focus, technologies and decision making policies, with massive local data generation which reutilization is vital. In this new panorama, sole systems, monoliths, are not an efficient solution, giving place to distributed systems. However, theirs advent turns the applications more complex, implying that tradicional software development methods will be inefficient. The reduction of this complexity can be reached throught novel methods, taking into account this new paradigm. This work objective is the development of a tool that will help the distributed software requirements definition using the Distributed Intelligent Objects Based Development Methodology. Many approaches for requirements definition have been studied: viewpoints, pattern usage in scenario construction, requirements pattern and models for description, qualification, analysis and validation of requirements. From these, two were chosen: viewpoints, because they allow rastreability of use cases and is easily integratable into current software development process, and the usage of models for requirement description and qualification, allowing a classification of viewpoints based on criteries defined by the requirement engineer. The tool's design is documented in UML and implemented in Java, using the OpenORB as CORBA middleware and PostgreSQL as the persistance mechanism.

Sumário

1	Introdução	1
2	Sistemas distribuídos	3
2.1	Introdução	3
2.2	Definição de sistemas distribuídos	3
2.3	Características de Sistemas Distribuídos	4
2.3.1	Heterogeneidade	4
2.3.2	Abertura	5
2.3.3	Segurança	6
2.3.4	Escalabilidade	7
2.3.5	Tratamento de falhas	8
2.3.6	Concorrência	8
2.3.7	Transparência	9
2.4	Considerações finais	10
3	Engenharia de Software	12
3.1	Introdução	12
3.2	Processo	13
3.3	Métodos	15
3.4	Ferramentas	15
3.5	Considerações finais	16
4	Engenharia de requisito	17
4.1	Introdução	17
4.2	Definição	18
4.3	Problemas encontrados durante o processo de engenharia de requisitos .	18
4.4	Técnicas aplicáveis no processo de engenharia de requisitos	21
4.4.1	Uso de padrões na construção de cenários	21
4.4.2	Identificação de Padrões de Reutilização de Requisitos de Sistemas de Informação	25
4.4.3	Pontos de vista	27

4.4.4	REQAV: Modelo para Descrição, Qualificação, Análise e Validação de Requisitos	37
4.4.5	Usando diferentes meios de comunicação na negociação de requisitos	38
4.4.6	Análise das diversas técnicas e métodos de engenharia de requisitos estudadas	40
4.5	Considerações finais	42
5	Guia para Engenharia de Requisitos	43
5.1	Introdução	43
5.2	Conceitos básicos	43
5.3	Criação de um modelo sintetizado	43
5.4	Descoberta de stakeholders	44
5.5	Captura de visões	44
5.6	Identificação de casos de uso e atores	45
5.7	Análise e resolução de conflitos	46
5.8	Considerações finais	46
6	Ferramenta Proposta	47
6.1	Introdução	47
6.2	CORBA	47
6.3	Persistência utilizando banco de dados	49
6.4	Metodologia de Desenvolvimento Baseado em Objetos Distribuídos Inteligentes	51
6.5	Extensão da linguagem UML	56
6.6	Ferramenta Proposta	58
6.6.1	Framework veryhot	59
6.6.2	Kernel	61
6.6.3	CoolCase	62
6.7	Considerações finais	70
7	Conclusão	71
	Referências bibliográficas	73

Lista de Figuras

4.1	Descrição do padrão Negociação Terminada com Produção	24
4.2	Exemplo de árvore de decisão para a seleção de padrões	26
4.3	Exemplo de regra	30
4.4	Pontos de vista e estrutura das informações	33
4.5	Modelo do processo VORD	34
4.6	Classes de ponto de vista do VORD	34
4.7	Cinco configurações de grupo: (a) cara a cara e (b) distribuída	39
6.1	Mecanismos para requisição e acionamento de objetos	49
6.2	Interfaces do Object Request Broker	49
6.3	Representação dos relacionamentos	59
6.4	Representação dos atores	59
6.5	Representação dos casos de uso	59
6.6	Diagrama de classes - Pacote veryhot	61
6.7	Diagrama de classes - Pacote veryhot.tool	62
6.8	Diagrama de classes - Pacote kernel	63
6.9	Diagrama de classe da CoolCase	64
6.10	Diagrama de classe relacionando figuras e elementos do modelo de caso de uso	65
6.11	Infraestrutura da ferramenta proposta	66
6.12	Arquitetura básica do sistema	67
6.13	Diagrama de classe simplificado da UML quanto aos casos de uso . . .	68
6.14	Protótipo da ferramenta em funcionamento	70

Capítulo 1

Introdução

As recentes tendências do mercado têm mostrado que a complexidade do software continuará a crescer drasticamente nas próximas décadas. Aliada a isto, a crescente globalização acaba por envolver diferentes organizações em diferentes lugares, cada uma com políticas peculiares de tomadas de decisão. As organizações também possuem um grande volume de dados, gerados durante o processo de engenharia de software, distribuído em seus computadores. Deste modo, os produtos de software isolados estão caindo em desuso à medida que são cada vez mais disseminadas a Internet e as Intranets.

O advento de sistemas distribuídos heterogêneos leva a aplicações mais complexas, implicando que desenvolver software usando métodos tradicionais tem se tornado cada vez mais inadequado. Este fato nos leva à necessidade de adotar novas tecnologias e condutas no processo automatizado para o desenvolvimento deste tipo de software. Em [Huz 95], são analisadas várias ferramentas que dão suporte ao desenvolvimento de software paralelo: PO, GRASPIN, VISTA, PROOF, TRAPPER, PARSE. Tais ferramentas trabalham os aspectos de programação, não apresentando preocupações quanto ao processo de engenharia de software (à exceção de PROOF e PARSE). Também encontram-se na literatura referências a alguns poucos ambientes de desenvolvimento de software: ONIX [Sat 94], ABACO [SOU 98], PROSOFT Distribuído [Sch 95], sendo somente estes dois últimos destinados ao desenvolvimento de software distribuído. Assim, encontra-se em andamento um projeto para definir uma metodologia para desenvolvimento de software baseados em objetos distribuídos inteligentes (MDSODI), oferecendo recursos de reusabilidade de componentes.

Um aspecto fundamental no processo de engenharia é a definição de requisitos. A utilização de novas tecnologias que possibilitem uma melhor captura e análise dos requisitos é vital. Neste trabalho, são estudadas várias técnicas: pontos de vista, qualificação automática de requisitos, identificação de padrões de reutilização e de cenário, técnicas para negociação de requisito. Combinando com a MDSODI, a proposta é criar uma

ferramenta que apóie a definição de requisitos de software distribuído.

Assim o trabalho encontra-se estruturado da seguinte forma: os capítulos 2, 3 e 4 abordam conceitos básicos de sistemas distribuídos, engenharia de software e engenharia de requisito; o quarto capítulo também engloba um estudo sobre técnicas para análise de requisitos; o quinto e sexto capítulos tratam dos resultados deste trabalho, diretrivas para um método de engenharia de requisitos e detalhes sobre a implementação da ferramenta.

Capítulo 2

Sistemas distribuídos

2.1 Introdução

Entender o que é um sistema distribuído é um requisito básico para identificar as características relevantes que podem ser abordadas na definição de requisitos. Ao longo deste capítulo será exposta a definição de um sistema distribuído, suas características e como estas podem ser identificadas na definição dos requisitos.

2.2 Definição de sistemas distribuídos

Um sistema distribuído compõe-se por componentes, localizados em computadores conectados através de uma rede, que se comunicam e coordenam suas ações através de passagem de mensagens [COU 2001].

A principal razão de um sistema distribuído é o compartilhamento de recursos (impressoras, arquivos, bases de dados, etc). Em um nível mais baixo, podemos considerar a economia com um melhor aproveitamento de *hardware*. Por exemplo: compartilhar uma impressora com todos os computadores de um laboratório, utilizar sistemas de pouca capacidade computacional e obter um bom desempenho, compartilhar arquivos. Sistemas distribuídos com estas características existem a anos e não são de concepção complexa.

Em um nível mais elevado, pode-se pensar em uma utilização inteligente dos recursos, com um sistema distribuído provendo serviços mais refinados que tão somente compartilhar arquivos. Por exemplo, o usuário deseja um texto sobre avestruzes. Com todos os arquivos do sistema à mão, é possível encontrar o que ele deseja. Mas não seria melhor utilizar um serviço de busca para conseguir este texto? Com certeza o

tempo de resposta seria menor. Mas, na verdade, nosso usuário deseja um texto que relate a avestruz ao aquecimento global. Dificilmente alguém escreveria um texto específico sobre isto, mas talvez exista alguma pesquisa sobre aves e efeito estufa. Sendo a avestruz uma ave, efeito estufa uma das causas do aquecimento global, um sistema de inferência, utilizando talvez os serviços providos pelo mecanismo de busca, poderia trazer esta informação para o nosso usuário. Se na primeira solução, buscando arquivo por arquivo, ele demoraria dias para encontrar a informação, utilizando o mecanismo de busca, talvez horas, e, com este último serviço, ele conseguiria a informação quase instantaneamente.

Tecnicamente uma solução como a exposta acima poderia ser implementada, centralizada, em um único computador. Claro, considerando que tem-se apenas um usuário usando o sistema. Porém, com uma capacidade semântica deste nível, é de se esperar que muitas pessoas o utilizem, provavelmente simultaneamente. E se cada pessoa, além de consumir dados, também produz? Armazenar tudo em um único computador começará a afetar mais ainda o desempenho e o custo da máquina. Decide-se então distribuir estes dados entre um conjunto de computadores ou, quem sabe, até talvez nos computadores dos próprios usuários. Melhor: os usuários podem passar a vender seus dados. Mas como tratar agora da questão de segurança, ou seja, garantir que os dados, agora distribuídos, não serão inadvertidamente alterados (corrompidos). Ou então garantir que somente a pessoa que comprou a informação poderá recebê-la, impossibilitando que “espiões” observem o canal de comunicação e se usurpam dos dados que nele trafegam? Eis que começam a surgir os grandes problemas, os desafios, dos sistemas distribuídos: heterogeneidade, abertura, segurança, escalabilidade, tratamento de falhas, concorrência e transparência.

2.3 Características de Sistemas Distribuídos

2.3.1 Heterogeneidade

Um sistema distribuído geralmente é construído sobre diferentes tipos de rede, sistemas operacionais, linguagens de programação, computadores das mais variadas arquiteturas e implementações de componentes diversos. Garantir que tudo isto se comunique har-

monicamente é uma tarefa complicada. Por exemplo, existem inúmeros tipos de rede: Ethernet, TokenRing, ATM, GSM, etc. Pode-se ter um sistema no qual tem-se todas presentes e, mesmo assim, deseja-se que um componente em uma rede Ethernet consiga se comunicar com outro na rede GSM. O estabelecimento de um protocolo comum a ser executado sobre esta camada física (IP, por exemplo), com *gateways* interconectando as distintas sub-redes, é uma solução.

Outra questão é relativa aos computadores. Alguns trabalham com números no formato *big endian*, outros em *little endian*. O mesmo problema se aplica à cadeia de caracteres: adotam-se caracteres de 7, 8 ou 16 bits? A solução mais comumente adotada é a utilização de um *middleware*. Ele provê um modelo computacional uniforme para uso pelos programadores de servidores e aplicações distribuídos. Alguns *middlewares* utilizados atualmente: CORBA e Java RMI.

Por fim, tem-se a questão dos códigos móveis. Se existem problemas quanto ao formato de *strings* e números, quanto mais a nível de instruções. A solução comumente adotada é utilização de máquinas virtuais. Com o *hardware* da máquina local, faz-se a emulação de uma máquina hipotética, existente somente em *software*. Os exemplos mais conhecidos são o Java, da Sun Microsystems, e o .NET, da Microsoft.

2.3.2 Abertura

A documentação das interfaces dos componentes (e de sua completa especificação, se possível) permite a construção de novos componentes e ajuda a garantir a interoperabilidade entre eles. A implicação disto é que o sistema pode ser implementado de várias maneiras sem comprometer a interoperabilidade com os demais componentes. Por exemplo, pode-se implementar um componente otimizado para alto desempenho ou então de tamanho adequado para utilização em sistemas embarcados. Outra implicação é que não existe mais a dependência por uma única empresa, qualquer uma pode (ao menos teoricamente) implementar um componente compatível, para isto respeitando a interface e seus comportamentos conforme consta na especificação.

Devido a complexidade inerente aos sistemas distribuídos, ao invés de cada empresa propor sua própria solução, o caminho escolhido foi a criação de grandes grupos, cons-

tituindo assim organizações que, a partir de propostas de seus elementos, criam uma especificação única, aceita por todos. O processo de criação de uma especificação é, na maioria das vezes, muito lento: são várias propostas estudadas, modificações discutidas e votações realizadas. Em compensação, tem-se um resultado bem consistente, abrangente e, principalmente, de grande influência no mercado. Um exemplo é o CORBA, criado pela OMG.

2.3.3 Segurança

A garantia da confiabilidade, integridade e disponibilidade dos componentes e seus produtos têm ganho grande enfoque ultimamente com a preocupação das pessoas quanto a segurança de seus dados. Aquisição ou adulteração de informações por pessoas não autorizadas ou até mesmo o vandalismo, impedindo o correto funcionamento do sistema, causam enormes prejuízos às empresas e seus clientes, justificando o investimento em soluções para estes problemas.

A primeira coisa a ser estudada sobre segurança é a maneira como ela pode ser comprometida. Segundo [COU 2001], os ataques podem ser de cinco tipos:

- *Eavesdropping*: Obtenção de cópia de mensagens sem autorização.
- Mascaramento: Envio ou recebimento de mensagens usando a identidade de uma outra pessoa (sem a autorização da mesma).
- Adulteração de mensagens: Interceptação de mensagens e posterior alteração do conteúdo das mesmas antes de chegar ao destinatário.
- Repetição: Retransmissão de mensagens previamente gravadas.
- Negação de serviço: Superutilização de um recurso, impedindo a utilização do mesmo por outras pessoas.

Várias técnicas foram desenvolvidas para evitar estes problemas e garantir a segurança dos sistemas: criptografia (simétrica ou assimétrica), assinaturas digitais, controle de acesso:

- Criptografia: A criptografia assimétrica é comumente utilizada para trocar dados de maneira segura sem precisar revelar uma senha, algo muito desejável. Porém, esta técnica possui um baixo desempenho. A alternativa é a utilização de criptografia simétrica, que requer o conhecimento de uma senha por ambas as partes trocando dados. Ela possui um alto desempenho, porém existe a transmissão da senha para a outra pessoa. Repassar este dado por meios comuns (email, telefone) invalida toda a segurança que a criptografia simétrica possui. Por isto, uma solução comumente adotada, é utilizar-se da criptografia assimétrica para trocar estas chaves para então utilizar-se da criptografia simétrica para repasse de dados.
- Assinatura digital: Garante a autenticidade das informações, torna-as inolvidáveis e não-repudiáveis.
- Controle de acesso: Restringem o acesso a um recurso.

A criptografia é a técnica que mais se destaca por ser aplicável a todos os tipos de ataque (com exceção da negação de serviço). Conjuntamente com a assinatura digital, permite a criação de sistemas extremamente seguros. Mas, devido ao custo de implementação das mesmas, é sempre desejável aplicar métodos mais simples para filtrar o máximo possível de mensagens indesejadas. Eis que o controle de acesso se aplica, evitando que mensagens vindas de pessoas não autorizadas tenham acesso ao recurso.

2.3.4 Escalabilidade

Um sistema é caracterizado como escalável se ele se mantiver utilizável com a adição de recursos e usuários. Isto implica em racionalizar os gastos com recursos físicos (*hardware*) no sistema de maneira que a perda de performance com o aumento de utilização do sistema seja controlável (através da utilização de algoritmos mais eficientes, por exemplo) e que os recursos lógico (*software*) não se esgotem. Possíveis abordagens para manter a escalabilidade de um sistema são: utilização de caches, duplicação de dados e *pools* de servidores, além, é claro, de algoritmos eficientes.

2.3.5 Tratamento de falhas

As falhas ocorridas num sistema distribuído deverão ser parciais, não afetando os demais componentes do sistema. Para alcançar isto, primeiramente o sistema deve ser capaz de detectar, direta ou indiretamente, falhas. Identificada a falha, pode-se então:

- Mascarar: Consiste na re-execução do comando que causou a falha.
- Tolerar: As falhas são repassadas aos programas clientes e, provavelmente, ao usuário.
- Recuperar: O estado do sistema antes da falha ocorrer é restaurado, garantindo assim a consistência do sistema.
- Redundância: Através da duplicação de componentes, pode-se tolerar as falhas em um componente através da requisição a um dos componentes redundantes.

Um correto tratamento das falhas permite aos sistemas distribuídos um alto grau de disponibilidade, característica esta extremamente importante para os usuários do sistema. O não-funcionamento de um serviço no momento que um usuário necessita é um grande desestímulo, eventualmente culminando, se a disponibilidade não aumentar, no abandono do sistema.

2.3.6 Concorrência

Um componente acessando recursos compartilhados deve garantir que suas operações serão executadas corretamente em um ambiente concorrente. Uma solução simples seria serializar todas as requisições recebidas, mas isto implicaria numa perda de performance muito grande. A utilização de várias linhas de execução e mecanismos de sincronização (*fine-grained* preferencialmente) são possíveis soluções (a melhor solução depende das características do sistema que se deseja valorizar, do tipo de recurso compartilhado, o modo de acesso que ele permite, etc).

2.3.7 Transparência

Transparência pode ser descrita como o acobertamento, para o usuário e para a aplicação, da separação dos componentes de um sistema distribuído, fazendo o sistema ser percebido como um todo ao invés de uma coleção de componentes independentes. Existem vários tipos de transparência:

- Transparência de acesso: Recursos locais e remotos são acessados com operações idênticas.
- Transparência de localidade: Utilização de recursos mesmo desconhecendo sua localização.
- Transparência de concorrência: Execução simultânea de vários processos utilizando um mesmo recurso e sem interferir um no outro.
- Transparência de replicação: A duplicação de um recurso, objetivando geralmente o aumento de performance ou confiabilidade, sem que as aplicações ou usuários tenham conhecimento.
- Transparência de falha: Encobrimento de defeitos, permitindo aos usuários e aplicações completar suas tarefas.
- Transparência de mobilidade: A mudança de localização de recursos e clientes em um sistema sem afetar as operações de usuários e programas.
- Transparência de performance: Reconfiguração do sistema para melhorar a performance conforme a carga do sistema.
- Transparência de escalabilidade: Expansão em escala do sistema e aplicações sem necessitar de mudanças na estrutura do sistema ou dos algoritmos empregados.

As transparências de acesso e localidade são as mais sentidas pelos usuários. Significa que ele não precisa conhecer a localização dos recursos do sistema para poder utilizá-los e que tanto recursos locais quanto remotos são utilizados do mesmo modo. Este tipo de transparência, também denominada transparência de rede, é bem comum no dia a dia. Um exemplo seria uma URL num navegador: a localização do recurso é desconhecida, o

sistema se encarrega de descobrir o número IP da máquina e mostrar a página requerida ao usuário. O acesso pode ser facilmente identificado em um navegador de arquivos, nos quais arquivos locais ou remotos (disponibilizados por NFS, por exemplo) são mostrados e utilizados pelo usuário da mesma forma.

Um tipo de transparência muito em destaque atualmente é a de mobilidade. Com o recente desenvolvimento de tecnologias de rede sem fio e sistemas embarcados, as pessoas passam a querer usufruir a partir de seu celular ou *PDA* das mesmas facilidades disponíveis em seu computador pessoal. Atualmente já existem celulares que podem acessar a serviços de *HomeBanking*, consulta de cotações, endereços, até mesmo *Internet*. O mesmo também ocorre para *PDAs* conectados através de rede *Ethernet* sem fio, como em Nova Iorque. Por trás desta comodidade, existe todo um esforço em garantir o acesso ao sistema. Por exemplo, o celular tem seu sinal captado através de antenas. Cada antena é responsável por um conjunto único de células. Caso o usuário se movimente e passe para uma célula que não é coberta pela mesma antena da célula anterior, o sistema deverá ser capaz de redirecionar as requisições para o nova antena. A operação desconectada, quando um usuário entra num túnel, por exemplo, também deve ser prevista e corretamente tratada.

De igual importância são: a transparência a falha, permitindo ao usuário completar sua tarefa não importando o que aconteça; e as transparências de performance, escalabilidade, concorrência e replicação, garantindo o uniforme desempenho do sistema, atendendo às expectativas dos clientes do sistema quanto ao tempo necessário a suas requisições.

2.4 Considerações finais

Construir um sistema distribuído não é uma tarefa simples. Alcançar todas suas características poderia ser classificado por alguns como utópico com os métodos de engenharia atuais. A primeira consideração da implementação de um sistema é definir os aspectos nos quais o sistema deve ser mais eficiente a fim de atender os requisitos do usuário e, ao mesmo tempo, permitir a construção de um sistema robusto e durável.

As características mais percebidas pelo usuário são geralmente as relativas às trans-

parências. Elas permitem enxergar o sistema como um todo às vistas do usuário. As demais características (segurança, heterogeneidade, abertura, escalabilidade, manuseio de falhas e concorrência) são o suporte para alcançar esta uniformidade. Neste trabalho, vários dos tipos de transparência (localidade, acesso, concorrência e, parcialmente, mobilidade) serão identificados nos requisitos e posteriormente aplicados aos casos de uso, atores e seus interrelacionamentos, o que virá a apoiar as etapas subsequentes de desenvolvimento do software.

Capítulo 3

Engenharia de Software

3.1 Introdução

O software é o meio básico para tornar o hardware útil e permitir sua utilização para armazenamento, obtenção e extração de dados, inferir informações e até gerar conhecimento. Ao longo das décadas, muito se fez em termos de desenvolvimento de software, passando da anarquia das décadas de 50 e 60 para processos bem definidos, específicos, previsíveis e produtivos. A exigência do mercado por soluções fez nascer toda uma ciência sobre o software, a engenharia de software:

O estabelecimento e uso de sólidos princípios de engenharia para que se possa obter economicamente um software que seja confiável e que funcione eficientemente em máquinas reais.

Fritz Bauer[NAU 69]

O cerne da definição até hoje é válido. As mudanças variam quanto a abrangência dos termos. Um software, atualmente, tem de ser não apenas confiável, tem de ser de qualidade. Isto implica não somente um correto funcionamento, mas sim também garantir o correto atendimento aos requisitos do usuário. E, um conceito talvez não tão relevante à época, é a questão do tempo. Com a concorrência atual, o prazo para desenvolvimento de software deve ser o mais curto possível. Resumindo:

A Engenharia de Software objetiva a construção de programas eficientes, baratos e de qualidade no prazo mais reduzido possível.

Esta ciência concentra seus esforços em três frentes: métodos, ferramentas e processos. Os métodos definem como fazer. As ferramentas automatizam esta execução. Os processos coordenam a utilização de métodos e ferramentas. Claro, isto é uma explicação muito simplista que será detalhada nas próximas seções.

3.2 Processo

O processo de engenharia de software define-se como um conjunto de atividades e produtos que resultam em um produto de software. Segundo [SOM 96], compõe-se de quatro atividades básicas:

- Especificação: Definição das funcionalidades e restrições sobre as quais o software irá trabalhar.
- Desenvolvimento: Produção do software com base nas especificações.
- Validação: Garantia de que o software atende às necessidades do usuário.
- Evolução: Suportar mudanças, adequando-se a novas necessidades do usuário.

Uma das primeiras atitudes necessárias para melhorar a engenharia de software e atender às exigências do mercado é a adoção de um processo de engenharia, organizando assim as atividades necessárias à conclusão de um sistema (em síntese, pode-se definí-lo como “quem vai fazer o que, quando e como”). O início do fim da “arte de fazer software” começa a dar espaço à “manufatura de software” como se fosse mais um produto¹.

Os primeiros processos de engenharia eram bem simples: modelos em cascata, desenvolvimento local, paradigmas nem sempre adequados, dificuldade de comunicação entre equipe de desenvolvimento e clientes, controle de configuração inexistente ou pouco eficiente, qualidade em segundo plano. Não que nestas últimas décadas não tenham sido desenvolvidas técnicas eficientes, pelo contrário. A questão é que criar métodos eficientes demanda tempo, assim como ensinar, como aplicá-los:

¹A DMCA (Digital Millennium Copyright Act), lei sobre direitos autorais dos Estados Unidos da América, assinada em 1998, talvez seja um dos marcos mais importantes desta mudança, equiparando software a produtos industrializados[GRO 2002].

Existe um longo espaço de tempo entre o surgimento de novos conhecimentos e seu refinamento em tecnologia utilizável. Então há um outro longo período antes do aparecimento desta tecnologia no mercado na forma de produtos, processos ou serviços.

Peter F. Drucker[DRU 85]

Aos poucos, novas soluções, bem eficientes, surgiram: O *Unified Process*[JAC 99], por exemplo, com sua estratégia iterativa e incremental, dirigida a caso de usos e centrada na arquitetura, fruto de um esforço iniciado em 1987, e com raízes em 1968, com o processo criado pela Ericson [JAC 99].

A adoção de processos eficientes é um fator vital para o sucesso. Pode-se optar por comprá-los e adaptá-los às necessidades do domínio da empresa ou então criar processos próprios. Ou seja, em algum ponto desta cadeia, alguém terá de definir um processo. Decididamente isto não se constitui em uma tarefa fácil, como pode ser observado abaixo em uma pequena listagem de características importantes de um processo:

- Primeiro e mais importante: Deve usar a melhor tecnologia disponível no momento.
- Esta tecnologia deve ser facilmente empregável.
- Conseqüentemente, o desenvolvimento de ferramentas que o automatizem é essencial.
- Possuir capacidade de evolução mas, ao mesmo tempo, se ater à realidade.

A primeira afirmativa talvez seja até óbvia. O principal objetivo de um processo é tornar a engenharia mais eficiente. Isto implica em utilizar novas técnicas. No entanto, é natural esperar que a complexidade cresça proporcionalmente aos benefícios alcançados, tornando a execução do processo uma tarefa exaustiva, anti-produtiva, justamente o contrário do desejado. Surge então a necessidade de desenvolvimento de ferramentas que automatizem a utilização do processo e permitam aos engenheiros um maior esforço nos resultados e não nos meios.

3.3 Métodos

Os métodos definem detalhadamente os procedimentos para a construção de um software (ou parte de um software). Envolve “o planejamento e estimativa de projeto, análise de requisitos de software e de sistemas, projeto da estrutura de dados, arquitetura de programas e algoritmo de processamento, codificação, teste e manutenção” [PRE 95]. Geralmente são criados a partir de técnicas individuais (que não deixam de ser métodos, porém extremamente específicos) como, por exemplo, utilização de um léxico extendido na definição do Universo de Domínio, utilização de padrões no reutilização de requisitos, etc.

Um método geralmente é idealizado e desenvolvido em um ambiente controlado, sem grandes interferências. Seria muito difícil partir de um idéia para uma solução perfeita para o mundo real. Uma analogia pode ser feita com a criação de um novo mecanismo de solda a laser e a sua posterior utilização na linha de montagem de carros: com certeza a técnica, quando aplicada na indústria, terá de ser ajustada, adaptada às condições reais de uso, à interação com outros métodos previamente existentes, com características de ambientes distintas da qual onde fora idealizada. Em engenharia de software, cabe ao processo esta tarefa de transportar o método para a aplicação real na indústria do software.

3.4 Ferramentas

O objetivo das ferramentas, conforme anteriormente dito, é automatizar métodos e processos. Ao invés de todos os engenheiros executarem manualmente todas as atividades descritas, a ferramenta contribuirá ao bom desempenho na sua execução:

- Ajudando a realizar as atividades.
- Facilitando o gerenciamento dos produtos destas atividades.
- Sendo fácil de usar.
- Não sendo restritiva, estimulando a experimentação.

Atualmente existem várias ferramentas disponíveis (por exemplo: Rose², ClearCase³ e Requisite Pro⁴, da Rational; Together⁵, da TogetherSoft; Poseidon⁶, da Gentleware; CVS⁷, da Cyclic Software; BitKeeper, da BitMover⁸), auxiliando no gerenciamento de requisitos, controle de versões, criação de diagramas e tanta outras atividades inerentes ao processo de engenharia. Muitas delas, porém, não são específicas a um processo, o que torna o seu uso um tanto quanto mais difícil ou, pelo menos, não tão produtivo quanto poderia ser.

3.5 Considerações finais

Um software é feito para atender as necessidades de um usuário, ultimamente cada vez mais exigente. Conforme novas tecnologias surgem e se tornam comuns no dia a dia, é natural o ensejo por tais graus de facilidade, funcionalidade e qualidade. A crescente presença de sistemas computacionais em eletrodomésticos, automóveis, comunicação e entretenimento, exigindo programas complexos, em espaços de tempo reduzidos e custos mínimos completam o ambiente, no qual se tem atualmente milhares, senão milhões de empresas no mercado, digladiando-se. O resultado é que as menos eficientes não prosseguem, não alcançam o sucesso.

Processo, método e ferramenta são essenciais para a construção de software nas condições acima expostas. Permitem que programas sejam produzidos no prazo esperado, com a qualidade necessária e atendendo aos requisitos do cliente eficientemente. Além disto, possibilitam uma maior lucratividade na empresa graças aos gastos reduzidos com treinamento e a maior eficiência com que seus empregados trabalharão, além de permitir a criação de processos mais ágeis, consistindo de (produtivas e curtas) iterações, detectando riscos e erros prematuramente além de aumentar a confiança dos funcionários e dos clientes quanto ao sucesso do produto.

²<http://www.rational.com/products/rose/>

³<http://www.rational.com/products/clearcase/>

⁴<http://www.rational.com/products/reqpro/>

⁵<http://www.togethersoft.com/products/controlcenter/>

⁶<http://www.gentleware.com/products/>

⁷<http://www.cvshome.org>

⁸<http://www.bitkeeper.com>

Capítulo 4

Engenharia de requisito

4.1 Introdução

A engenharia de requisitos consiste em desenvolver as especificações de um sistema de maneira a atender as necessidades dos usuários e restrições do domínio da aplicação. Trata-se de uma atividade interativa e iterativa, evoluindo ao longo de todo o processo de engenharia do software.

A importância da engenharia de requisitos é facilmente detectada a partir do momento que tudo no processo de engenharia de software depende dela e erros em sua execução são fatais ao projeto ou, no mínimo, custam muito caro. Veja os exemplos abaixo:

- Um domínio de aplicação com limites não bem definidos ou muito amplo implica em maiores problemas de projeto, mais tempo para apurar as informações de maneira a deixar a especificação o mais consistente e não-ambigüa possível, uma quantidade de código muito grande a ser gerada e testada. No final, o cliente finalmente descobre que a programa não resolve sua necessidade, que todas aquelas funcionalidades são fúteis. Os engenheiros tentam então reverter isto, tentando, agora no final do projeto, rever os requisitos e corrigir o sistema em prazos curtíssimos, sobrecregando-se e levando a uma queda de produtividade. Se o sistema um dia ficar pronto, provavelmente terá uma qualidade baixa e um custo muito maior do que o previsto.
- A não execução da análise dos riscos quanto aos requisitos, possivelmente levando a atrasos na execução do projeto e insegurança na equipe de desenvolvimento.

4.2 Definição

Um processo de engenharia de requisitos, segundo Linda [MAC 96], compreende as seguintes fases:

- Identificação da necessidade de automação ou problema a ser resolvido e determinação de um domínio no qual estará localizado o trabalho.
- Análise das necessidades do usuário e restrições do domínio, construindo assim um melhor entendimento do problema a ser resolvido.
- Escolher o que vai ser feito tendo em vista prazos, custos e factibilidade.
- Produção de um documento de requisitos.

A primeira fase, a delimitação do escopo da aplicação, seu domínio, é a mais simples porém extremamente importante. Ela evita a definição de requisitos que não estejam dentro do domínio da aplicação, permitindo a concentração dos esforços naqueles requisitos que colaborarão ao atendimento às necessidades do cliente.

A segunda etapa e a terceira interagem muito entre si. A escolha do que vai ser feito elimina ou restringe requisitos propostos durante a análise. Nesta análise então aprofunda-se o detalhamento dos requisitos, além de se incluir novos. Porém, este processo pode ser repetido infinitamente se assim for permitido, tornando-se necessário o estabelecimento de critérios que balanceiem as necessidades do usuário com o custo e tempo de entrega disponíveis, facilitando assim a determinação de um *baseline*. Tendo este sido estabelecido, cria-se o documento de requisitos que servirá de base para o resto do processo de engenharia de software.

4.3 Problemas encontrados durante o processo de engenharia de requisitos

Durante o processo de engenharia de requisitos, os seguintes problemas são geralmente encontrados:

- Inconsistências.
- Falta de completude.
- Ambigüidade.
- Domínio da aplicação mal definido.

Destes, o de mais difícil solução é a solução de inconsistências. Inconsistências são “uma situação em que duas partes de uma especificação não obedecem algum relacionamento mantido entre elas” [EAS 95]. Estes relacionamentos podem se referir tanto a aspectos sintáticos quanto semânticos, além dos relacionamentos inerentes ao próprio processo.

As inconsistências podem ser geradas por erros ou conflitos entre duas ou mais partes da especificação, falta de informações e falhas na aplicação de um método de especificação de requisitos. O aspecto evolutivo dos requisitos contribui ainda mais para aumentar a complexidade do problema, ocasionando a criação de inconsistências nos mais diversos momentos do processo de desenvolvimento ou a reativação de antigas inconsistências devido a invalidação de soluções previamente aplicadas. Somando a isto o desenvolvimento distribuído, a resolução destes problemas fica ainda mais difícil. Portanto, técnicas para tratar as inconsistências, abordando sua descoberta e gerenciamento, devem ser utilizadas.

Técnicas para descoberta de inconsistências, segundo [EAS 95], consistem em desenvolver maneiras de criar relacionamentos entre as informações da especificação da maneira mais automática e completa possível: criando relações explicitamente, inferindo a partir dos dados constantes na especificação, utilizando técnicas de particionamento.

Quanto ao gerenciamento de inconsistências, existem duas abordagens: manter a especificação sempre consistente ou tolerar as inconsistências. A primeira alternativa é difícil de implementar e cara, sendo de difícil aplicação em projetos muito grandes e sem um alto grau de formalismo. A segunda é mais adequada para os processos de engenharia de software evolutivos, permitindo uma maior liberdade para projetar o sistema, evitando decisões prematuras e garantindo o atendimento do maior número possível de requisitos.

Todavia, o gerenciamento tolerante é mais complexo. Torna-se necessária a criação de mecanismos que permitam descobrir a causa das inconsistências. Uma maneira de conseguir isto é através de controle de versão, possibilitando comparações entre as partes antigas da especificação com as mais recentes e, consequentemente, a descoberta das causas das inconsistências. Exemplos de soluções que utilizam controle de versão: *pollution markers* [BAL 91], CONMAN [SCH 88] (ferramenta de gerenciamento de configuração). Uma outra abordagem é a *lazy consistency*, proposta por Narayananwamy e Goldman [NAR 92]. Nela, as mudanças efetuadas no sistema são anunciadas, permitindo a descoberta de inconsistências pelos desenvolvedores. Outra solução, para especificações formais, é proposta por Besnard e Hunter [BES 95], utilizando lógica paraconsistente. Nela são particionadas especificações inconsistentes até que cada parte seja internamente consistente mas, na junção, inconsistências sejam encontradas.

O gerenciamento de inconsistências (tolerante), segundo [EAS 95], geralmente consiste nas seguintes atividades:

- Detecção: Procura por informações da especificação que quebram uma regra de consistência.
- Classificação: Identifica o tipo de inconsistência.
- Manuseio: Determina a ação a ser tomada na presença de inconsistências:
 - Resolver: Elimina a inconsistência imediatamente.
 - Ignorar: Simplesmente ignora a inconsistência.
 - Adiar: Não trata da inconsistência no momento, postergando seu manuseio.
 - Amenizar: Melhora a situação da inconsistência mas não necessariamente a resolve.
 - Contornar: Não considera como uma inconsistência, ela provavelmente é uma exceção.

4.4 Técnicas aplicáveis no processo de engenharia de requisitos

Encontram-se na literatura diversas técnicas a respeito do processo de engenharia de requisito: uso de padrões, pontos de vista, modelos para qualificação de requisitos, abordagens para a negociação de requisitos. Nas sub-seções abaixo, são sucintamente descritas estas técnicas e, ao final, faz-se uma breve análise delas.

4.4.1 Uso de padrões na construção de cenários¹

Uma alternativa aos métodos que empregam casos de uso é a utilização de cenários. Eles possuem mais informações do que os casos de uso, são tipados, empregam-se tipo de atores ao invés de atores reais do domínio da aplicação. Apesar destes acréscimos, mantém-se a acessibilidade deste método quando comparado ao de casos de uso.

Uma técnica efetiva para construir tais cenários é através do vocabulário do Universo de Discurso, ou seja, as palavras mais utilizadas na aplicação em questão. Utiliza-se uma estrutura denominada LEL (Léxico Extendido da Linguagem) que permite registrar tal vocabulário e sua semântica, deixando para uma etapa posterior a compreensão do problema. Cada símbolo descoberto é identificado por uma palavra ou frase relevante no domínio da aplicação, utilizando-se linguagem natural para isso, facilitando a comunicação com o *stakeholder*.

Seguindo a estratégia “divisão e conquista”, os cenários são tratados como compostos de subcenários ou diversos episódios. Para cada um destes são considerados os seguintes aspectos:

- Número de atores envolvidos;
- Atores requerem resposta ou não;
- A resposta deve ser imediata ou pode ser adiada;
- O papel desempenhado pelo ator.

¹[RID 2000]

De todos os aspectos acima, o mais importante é o papel desempenhado pelo ator, sua participação nos sub-cenários ou episódios. Baseando-se nisto, propõe-se a seguinte classificação para os episódios:

- **p** (produção): um único ator, de maneira autônoma, realiza uma troca com o macrosistema.
- **s** (serviço): um dos atores adquire o papel de ator ativo e realiza uma ação em benefício de um ou mais atores passivos.
- **c** (colaboração): dois ou mais atores realizam uma ação que requer a participação de todos eles, produzindo um efeito global no sistema.
- **d** (demanda): um dos atores desempenha um papel ativo e um ou mais são passivos, sendo que as ações do ator ativo exigem, implicitamente, a resposta dos atores passivos.
- **r** (resposta): um ator, que fora passivo em um episódio do tipo **d**, assume o papel ativo e atende o pedido (responde a requisição do ator ativo no episódio **d**).
- **i** (interação): são episódios que reunem as propriedades dos episódios de resposta (**r**) e demanda (**d**), atendendo um pedido prévio e gerando um novo.

Definidas as classificações dos episódios e sub-cenários, pode-se construir inúmeras situações com características bem definidas. Por exemplo, uma seqüência de episódios que começa com um do tipo **d** e continua com vários do tipo **i** implica na existência de dois ou mais atores realizando uma atividade interativa na qual uma ação de um ator provoca uma ação de outro ator e assim por diante. Esta seqüência de episódios denomina-se **Negociação**. Porém, a classificação das situações não se restringe somente aos episódios. Todo e qualquer elemento que pertença ou influencie o cenário pode ser considerado. Vários tipos de situações são definidos de acordo com estes critérios:

- **Produção**: realização de uma atividade produtiva que provocará um efeito sobre o macrosistema;
- **Serviço**: prestação de um serviço que é necessário para um dos atores;
- **Colaboração**: associação de vários atores para realizar uma atividade cooperativa com um objetivo comum;

- **Negociação inconclusiva:** iniciação de uma atividade que requer uma sequência coordenada de ações por parte dos atores, necessitando de outra situação para concluir a negociação;
- **Negociação inconclusiva com disparo de cenários:** iniciação de uma atividade que requer uma sequência coordenada de ações por parte dos atores, criando a necessidade de várias outras situações;
- **Final de negociação:** sequência coordenada de ações por parte dos atores que finaliza uma atividade iniciada em outro cenário;
- **Etapa de negociação:** sequência coordenada de ações por parte dos atores que continua uma atividade de uma situação anterior e cuja finalização é inconclusiva;
- **Etapa de negociação com disparo de cenários:** sequência coordenada de ações por parte dos atores que continua uma atividade de uma situação anterior e cuja finalização resultará em várias outras situações;
- **Negociação terminada:** fim de uma atividade que requer uma sequência coordenada de ações por parte dos atores.

As situações podem ser compostas por várias seqüências distintas de episódios, formando novas situações:

- Produção + Serviço + Colaboração;
- Negociação inconclusiva com Produção ou Serviço ou Colaboração;
- Fim de negociação com Produção ou Serviço ou Colaboração;
- Etapa de Negociação com Produção ou Serviço ou Colaboração;
- Negociação terminada com Produção ou Serviço ou Colaboração;
- Negociação inconclusiva com disparo de cenários e Produção ou Serviço ou Colaboração;
- Etapa de negociação com disparo de cenários e Produção ou Serviço ou Colaboração.

Os padrões de construção de cenários seguem a estrutura definida por Leite [LEI 97]: título, objetivo, contexto, atores, recursos, episódios e exceções. Além destes dados, foram acrescentados textos complementares. Por exemplo, quanto aos episódios, pode-se acrescentar uma descrição dos tipos de episódios, a quantidade de episódios de cada tipo, a ordem em que estão. Um exemplo pode ser visto na figura 4.1.

Negociación terminada con producción

Título: Ejecución de una actividad centrada en transacciones

Objetivo: Realizar una actividad que requiere una secuencia coordinada de acciones por parte de los actores, junto con actividades de producción intercaladas

Contexto:

- Ubicación geográfica: generalmente, el lugar de trabajo del actor principal
- Ubicación temporal: : generalmente determinado por el actor principal y posiblemente breve

Atores: Varios, al menos dos

Recursos: Al menos uno, generalmente muchos

Episodios:

- Por lo menos uno como el siguiente:
 - Un actor realiza una acción que requiere respuesta inmediata de otro actor y varios o ninguno como el siguiente
 - Un actor realiza una acción que responde a una acción anterior y que a su vez, requiere respuesta inmediata de otro actor
 - (debe estar precedido por una acción que requiera respuesta inmediata)
- y por lo menos uno de los siguientes:
 - Un actor realiza alguna actividad que produce algún efecto sobre el macrosistema, y que es indispensable para continuar con la transacción
- y por lo menos uno como el siguiente:
 - Uno de los actores realiza una acción que responde a una acción anterior y que no requiere respuesta
 - (debe suceder a todas las acciones que requieran respuesta inmediata)

- Sólo es necesario respetar orden donde está explícitamente indicado, pudiendo existir grupos no secuenciales

Excepción: Circunstancia que obstaculiza el cumplimiento del objetivo

Figura 4.1: Descrição do padrão Negociação Terminada com Produção [RID 2000].

A partir dos métodos usuais de aquisição de dados para elicitação de requisitos (entrevistas, questionários, etc), definem-se situações compostas por vários episódios. Classificam-se estes de acordo com o número de atores envolvidos, se requerem ou não resposta (e, se for o caso, se a resposta é necessária imediatamente ou pode ser

deferida) e, principalmente, pelo papel desempenhado pelos atores. Conseqüentemente, pode-se classificar as situações de acordo com a classificação dos episódios que as compõem (ou seja, de acordo com um padrão). Apesar de existir um leque grande de padrões devido às combinações de tipos de episódios existentes em cada cenário, utilizando-se uma heurística baseada em árvores de decisão esta tarefa torna-se muito fácil.

No entanto, de nada adiantaria todo este esforço se não houvesse uma maneira viável de empregá-lo. Portanto, a seguinte heurística é empregada:

1. O primeiro passo é produzir uma primeira versão dos cenários a partir do léxico do domínio da aplicação. Propõe-se que este seja criado através da observação, leitura de documentação, entrevistas, dentre outras técnicas possíveis. Enfim, definem-se várias situações que proverão um meio para verificação e validação dos cenários.
2. Identificam-se os atores do universo do domínio da aplicação e extraem-se os efeitos causados por estes atores. Cada um destes será um novo cenário que será incorporado à lista de cenários candidatos.
3. Através de um sistema especialista (figura 4.2), tenta-se extrair o máximo possível de informações sobre os cenários candidatos.

A aplicação da heurística, conforme pode ser notado, não é complicada, o que torna sua implementação mais simples. Outra característica interessante é a possibilidade de reuso com um alto grau de abstração, logo no início do processo de engenharia de software, permitindo que muitos erros sejam evitados ou detectados prematuramente, tornando o processo mais rápido e garantindo uma melhor qualidade.

4.4.2 Identificação de Padrões de Reutilização de Requisitos de Sistemas de Informação²

A aplicação de modelos e padrões de requisitos, uma vez padronizados quanto à maneira como são especificados, permite identificar padrões de reutilização de requisitos, tanto

²[TOR 2000]

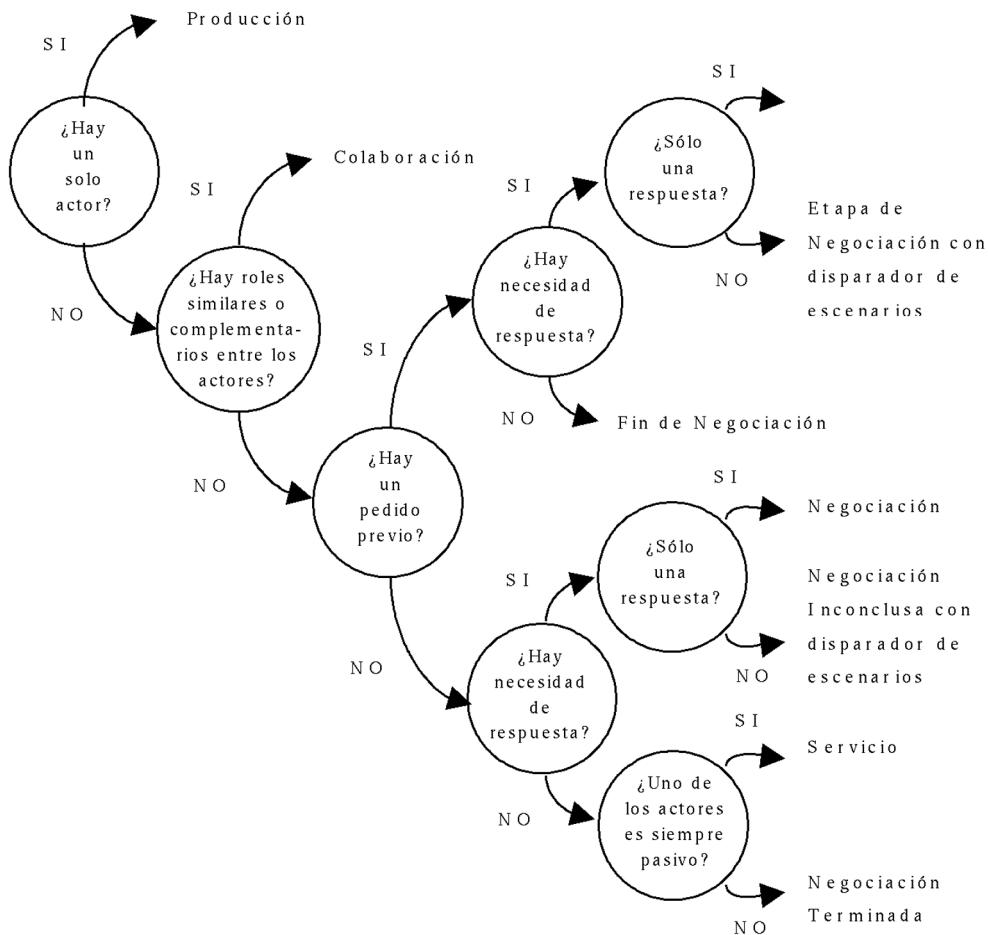


Figura 4.2: Exemplo de árvore de decisão para a seleção de padrões [RID 2000].

dos requisitos do cliente (requisitos-C) como dos requisitos do desenvolvedor (requisitos-D), permitindo assim um desenvolvimento mais rápido e eficiente do software. Outro fato é que, graças à rastreabilidade entre os requisitos-C, requisitos-D e elementos de mais baixo nível de abstração (tais como componentes de software), pode-se também reutilizar estruturas mais complexas tais como código fonte, ou seja, um reuso vertical, abrangendo diversos níveis de abstração do software.

Os requisitos-C podem ser de três tipos:

- Requisitos de informação: Informações que devem ser armazenadas no sistema para satisfazer as necessidades dos clientes e usuários. Em geral é uma descrição de atributo que os objetos devem conter e possíveis restrições nos valores dos mesmos.

- Requisitos funcionais: Casos de uso do sistema, contendo informações tais como o evento de ativação, as pré-condições, as pós-condições, os passos que compõe o caso de uso e suas exceções.
- Requisitos não funcionais: características não funcionais que o cliente e o usuário desejam no sistema.

Dentre estes tipos de requisitos-C, identificam-se vários padrões-R_c. No caso de requisitos de informação, por exemplo: cliente/sócio, produto/artigo, empregado, venda/fatura, fornecedor, pedido ao fornecedor, nota fiscal. Destes padrões, segundo [TOR 2000], o que ocorre com maior frequência é o primeiro, cliente-sócio (mais de 90% dos casos). Depois tem-se produto/artigo com 60% e assim por diante. Tão interessante quanto isto é que estes padrões-R_c de requisitos de informação são diretamente utilizáveis, necessitando de mínimas modificações. O mesmo já não acontece com os padrões-R_c de requisitos funcionais. Estes são padrões baseados em parâmetros, o que demanda em um maior esforço para abstrair o padrão e, depois, os parâmetros que serão aplicados.

Além destes, existem os padrões de reutilização de requisitos-D (padrões-R_d). Eles sempre se relacionam com os seus respectivos padrões-R_c. A diferença entre um padrão e outro é o nível de detalhamento, trabalhando-se em um grau de abstração mais baixo. Os padrões-R_d para requisitos de informação são bem próximos de uma definição de classe, com a especificação explícita dos tipos de dados envolvidos. O mesmo acontece para os padrões-R_d para requisitos funcionais, utilizando OCL por exemplo.

4.4.3 Pontos de vista

A engenharia de requisitos orientada a ponto de vistas vem do reconhecimento que os requisitos do sistema são gerados por várias fontes distintas e que tal realidade deve ser incluída explicitamente no processo. Esta visão não é absolutamente nova, na verdade desde o final da década de 70, com o SADT [SCH 77], houve este reconhecimento. Porém, a utilização disto nunca ocorreu na proporção em que deveria, visto sua abrangência. Um exemplo isolado seria o CORE [MUL 79], utilizado pelo ministério de defesa da Inglaterra, do qual não se tem muitas informações nem ferramentas

disponíveis a preços razoáveis.

Durante as últimas décadas, várias pesquisas foram desenvolvidas na área, surgindo vários modelos de ponto de vistas [FIN 90, LEI 89, FIC 91]. A origem de modelos diferentes surge das características intrínsecas dos projetos para o qual o método foi criado, tomando definições de ponto de vistas que facilitassem o desenvolvimento dos sistemas. Por exemplo, em [SOM 96] são descritos os seguintes tipos de pontos de vista:

- Uma fonte ou sumidouro de dados: Os pontos de vista são responsáveis por produzir ou consumir dados. Analisando o que é produzido e consumido, podemos detectar, por exemplo, dados gerados mas não utilizados e vice versa.
- Um framework para representação: Cada ponto de vista é considerado como um tipo particular de modelo do sistema (por exemplo, um modelo entidade-relacionamento, um modelo de máquina de estados, etc). Comparando-os, torna-se possível a descoberta de vários requisitos que não seriam detectados sem a utilização desta técnica.
- Um receptor de serviços: Os pontos de vista são externos ao sistema e recebem serviços deste.

Devido à natureza do presente trabalho, que visa uma ferramenta que suporte a definição dos requisitos, tendo como base um método de análise de requisitos voltado ao usuário (casos de uso), a concepção de um ponto de vista como um receptor de serviços mostra ser a mais adequada.

Viewpoint framework³

Este *framework* foi desenvolvido por Stelve Easterbrook e Bashar Nuseibeh. Criado para abordar a engenharia distribuída de requisitos, ele utiliza o particionamento da especificação em partes com sobreposições e gerencia inconsistências utilizando uma técnica semelhante à *lazy consistency* [NAR 92].

³[EAS 95]

Os pontos de vista são definidos como objetos distribuídos fracamente acoplados, localmente gerenciados, que encapsulam conhecimento parcial sobre um sistema e seu domínio, e conhecimento parcial do processo de desenvolvimento. Eles são compostos por:

- Estilo: Maneira pela qual o ponto de vista expressa seu conteúdo. Esta maneira tem seu esquema e notação especificado no método adotado no processo de engenharia. Por exemplo, o estilo poderia ser UML ou Statecharts.
- Domínio: Área em que o viewpoint se concentra.
- Especificação: *Statements*, no estilo escolhido, descrevendo o domínio em questão.
- Plano de trabalho: Ações pelas quais a especificação pode ser criada e o modelo de processo que guiará a aplicação destas ações. Na verdade, trata-se do método empregado no desenvolvimento do software.
- Registro de trabalho: Histórico das ações efetuadas no viewpoint.
- Proprietário: Pessoa que criou o viewpoint.
- Lista de inconsistências.

Os pontos de vista são criados a partir de um gabarito (*template*). Este contém a mesma estrutura de um ponto de vista, mas só possui definido seu estilo e seu plano de trabalho. Quando criado um ponto de vista, escolhe-se um gabarito para servir de base.

O plano de trabalho guia todo o processo de criação e manipulação de um ponto de vista. Ele especifica as notações, como elas se relacionam e possibilidade de utilização das mesmas conjuntamente, como será feito o particionamento dos pontos de vista, quais as regras de verificação de consistência serão aplicadas, enfim, tudo que é necessário para o desenvolvimento do ponto de vista. Em [EAS 95], utiliza-se um método para especificar o comportamento de dispositivos. Ele possui as seguintes características:

- Especifica uma notação para expressar diagramas com estados e transições, incluindo extensões para expressar super e sub-estados. Os diagramas de estado

utilizados são os propostos em [HAR 87]. Eles estendem a notação a fim de incluir superestados e estados subordinados. As transições de um estado subordinante estão disponíveis para todos os subordinados. As transições são executadas não somente devido a um estímulo, devendo atender antes uma condição (identificada entre parênteses após o nome do estímulo).

- Define técnicas de particionamento de pontos de vista, permitindo a representação de um subconjunto do comportamento de dispositivos.
- Possui um conjunto de regras de consistência que testa se os pontos de vistas particionados de um mesmo dispositivo são consistentes um com o outro. Cada regra define, para uma determinada configuração, a existência de um determinado relacionamento. Se este não existir na especificação, trata-se de uma inconsistência. Mas, além destes relacionamentos (implícitos), podem ser definidos relacionamentos entre estados de maneira explícita pelo usuário. Com a criação de regras específicas para avaliar estes relacionamentos, torna-se possível otimizar o método adotado.
- Procedimento para análise, permitindo tratar pontos de vistas como dispositivos separados que interagem.
- Conjunto de regras que permite testar se os dispositivos em interação possuem comportamento consistente.

As regras são definidas como uma relação entre objetos de um ponto de vista origem e os de um ponto de vista destino. Elas não são aplicadas diretamente, sendo ativadas por entradas no modelo de processo de ponto de vista. Sua aplicação ocorre somente se um conjunto de pré e pós-condições forem satisfeitas. Isto permite criar regras específicas para cada estágio do processo de desenvolvimento, algo necessário quando se está gerenciando inconsistências. Abaixo (figura 4.3) está um exemplo de uma regra completa: o conjunto vazio indica que não existem pré-condições, tem-se a aplicação da regra R_1 sobre o ponto de vista VP_s e, em seguida, especificam-se as pós-condições.

$$\{ \} \quad \Rightarrow \quad [VP_S, R_1] \quad \{ \exists_1(\text{transition}(X, Y), VP_D.\text{transition}(X, Y)) \} \cup \\ \{ \text{missing}(\text{transition}(X, Y), VP_D.\text{transition}(X, Y), R_1) \}$$

Figura 4.3: Exemplo de regra [EAS 95].

Definidos os conceitos, torna-se possível explicar o processo. Ele é composto dos seguintes passos:

1. Criação de pontos de vista a partir de um modelo pré-definido no sistema.
2. Particionamento da especificação, escolhendo alguns pontos de vista para serem analisados. Uma provável abordagem é tomar uma partição aleatória (provavelmente o sistema inteiro) e dividí-la em partições menores até que se tenham partições vistas como consistentes internamente porém inconsistentes quando juntas. Esta verificação pode ser feita através de regras de consistência definidas em um método.
3. Aplicação de regras de consistência em cada ponto de vista. O resultado destas regras fica gravado no “Registro de Trabalho”. Se alguma regra for quebrada, registra-se na “Lista de Inconsistências” as inconsistências.
4. Resolução de inconsistências. Para cada uma no “Registro de Trabalho”, mostra-se uma lista de ações, da qual escolhe-se uma ação para resolver o problema (ou simplesmente escolhe por ignorá-lo). O histórico de trabalho pode ser consultado para decidir com melhor precisão qual a melhor ação a ser tomada.
5. Nova aplicação da regra no ponto de vista, verificando-se o resultado e tentando-se corrigir o problema novamente se for necessário.

Interessante notar que uma ação, apesar de seu efeito influenciar o outro ponto de vista da partição, não implica instantaneamente na mudança deste último (ou seja, tolerância a inconsistências). Por exemplo, uma ação é tomada no ponto de vista V1, que possui uma inconsistência com o ponto de vista V2, e esta ação acaba por resolver a inconsistência. Mas esta só é removida da lista de V1, em V2 ela continuará existindo até que as regras de consistência sejam reaplicadas em V2.

Outro aspecto importante é que a aplicação da regra com sucesso, num momento, não significa que não existerão inconsistências por toda a vida do ponto de vista. A partir do momento em que um ponto de vista é alterado e suas alterações influenciam os outros, novas inconsistências podem surgir, e estas só serão realmente detectadas com uma nova aplicação das regras.

Viewpoints Oriented Requirements Definition⁴

O *Viewpoints Oriented Requirements Definition* (VORD) é um método para engenharia de requisitos desenvolvido por Kotonya e Sommerville. Ele abrange desde a descoberta inicial dos requisitos até a modelagem detalhada do sistema.

Um ponto de vista é uma entidade cujos requisitos são responsáveis ou impõe restrições no desenvolvimento de um sistema. Ele é estruturado conforme mostrado na figura 4.4, composto por atributos, requisitos, restrições e cenários de evento. O ponto de vista pode ser classificado em direto e indireto:

- Ponto de vista direto: Correspondem aos requisitos específicos de um serviço provido pelo sistema.
- Ponto de vista indireto: São entidades que possuem requisitos sobre os serviços fornecidos pelo sistema, abordando geralmente algum aspecto comum a todos estes, porém sem interagir com os mesmos.

O método discute as seguintes etapas:

- Identificação e estruturação de pontos de vista
- Documentação de pontos de vista
- Análise e especificação de requisitos de pontos de vista

Olhando o modelo do processo (figura 4.5), podemos visualizar mais facilmente os processos e suas interações. Primeiramente são identificados os pontos de vista relevantes no domínio do problema, tendo como base disto alguns dados sobre as necessidades da organização e classes abstratas de pontos de vista. A seguir, documenta-se cada um destes pontos de vista, identificando-os com um nome, especificando seus requisitos, restrições e sua fonte. Finalmente, estes pontos de vista são organizados em um documento, de acordo com os stakeholders envolvidos e o tipo de documentação que eles desejam (e que o sistema exige para mostrar a especificação).

⁴[KOT 95]

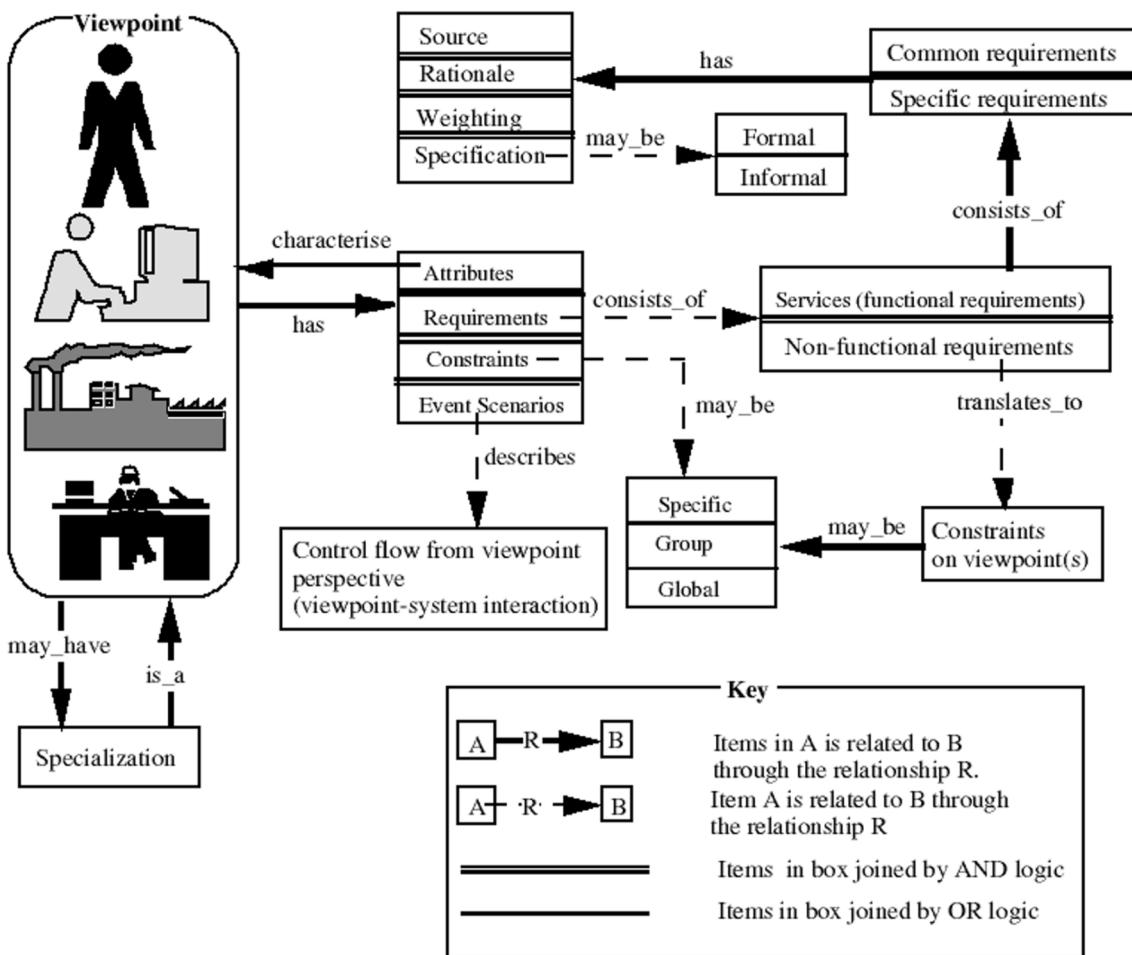


Figura 4.4: Pontos de vista e estrutura das informações [KOT 95]

Agora observando com mais detalhes cada uma destas etapas. Na identificação de pontos de vista, a seguinte heurística é tomada para identificar os pontos de vistas relevantes:

- Eliminação dos pontos de vista irrelevantes ao sistema com base nos pontos de vista abstratos pré-definidos (figura 4.6).
- Identificação dos *stakeholders*, certificando-se de que todos estejam sempre representados em algum ponto de vista.
- Identificação de outros pontos de vistas relevantes bom base nos modelos de uma arquitetura de sistema (provavelmente já existentes na maioria dos casos).

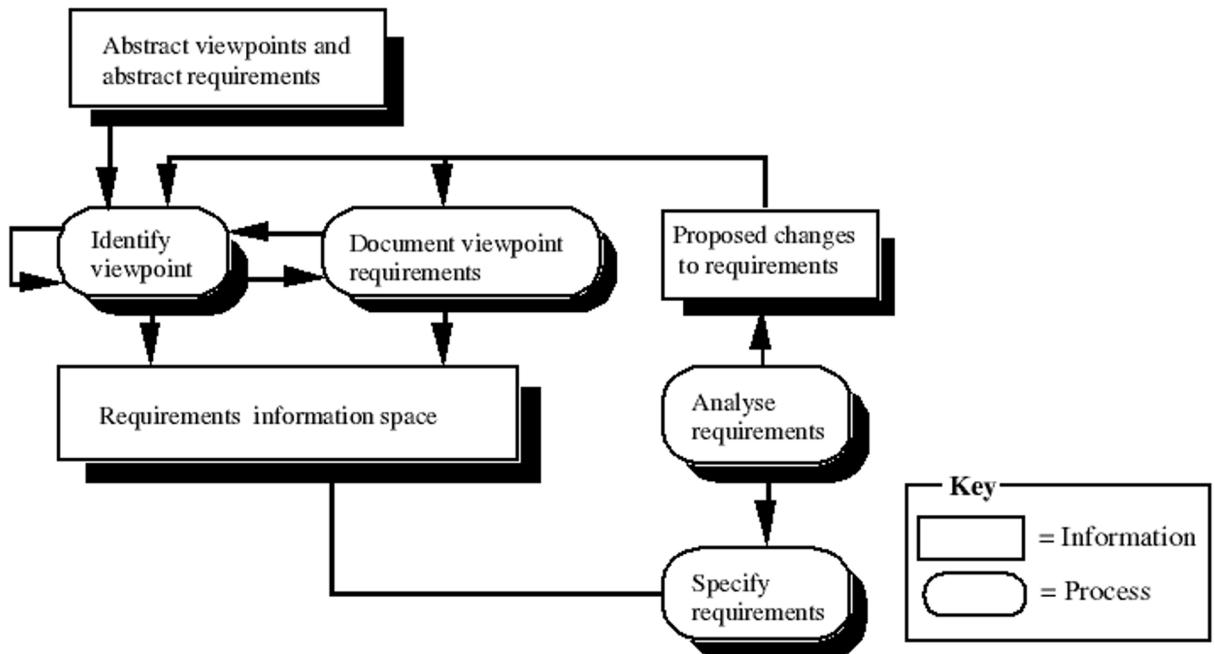


Figura 4.5: Modelo do processo VORD [KOT 95]

- Identificação dos operadores do sistema que o utilizam constantemente, ocasionalmente ou que mandam os outros fazerem as coisas para eles. Estes são potenciais pontos de vista.
- Identificação dos papéis de pessoas que estejam associadas a classes de pontos de vista indiretos. Geralmente tem-se pontos de vista associados com cada papel.

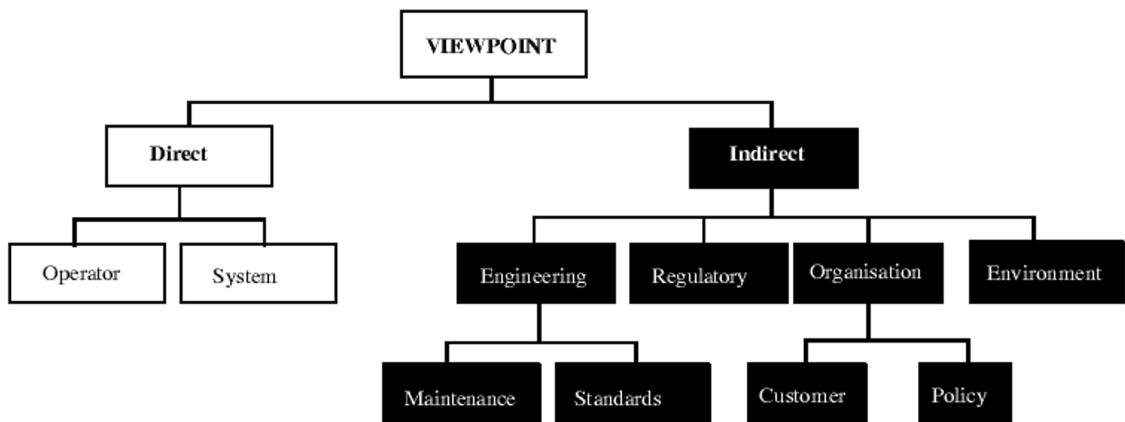


Figura 4.6: Classes de ponto de vista do VORD [KOT 95]

Uma vez determinados os pontos de vistas relevantes, tem-se de documentá-los adequadamente. Para cada ponto de vista associa-se um conjunto de requisitos (funcionais, não funcionais e de controle), fontes e restrições. Os requisitos de controle descrevem a sequência de eventos envolvidos no intercâmbio de informações entre um ponto de vista direto e o sistema. As restrições descrevem como os requisitos são afetados por requisitos não funcionais definidos por outros pontos de vista.

Um aspecto relevante do método são os cenários de eventos. Eles são uma sequência de eventos, as exceções que podem surgir durante a troca de informações e o sistema. Os eventos podem ser a respeito do ponto de vista (refletindo a percepção que o usuário possui da aplicação dos requisitos de controle) ou serem eventos em nível de sistema, refletindo os requisitos de controle. Esta distinção permite:

- Rastrear requisitos de controle a partir da perspectiva do usuário.
- Rastrear controles de nível de sistema para pontos de vista.
- Expor conflitos entre requisitos de controle.
- Capturar a natureza distribuída e em camadas dos controles.

Process and Requirements Engineering Viewpoints⁵

O *Preview* é uma evolução do *VORD*. Ele é voltado à elicitação de requisitos. Nele os requisitos são expressos em qualquer notação (linguagem natural, por exemplo). A análise, diferentemente do *VORD*, é dirigida por *concerns*. Um ponto de vista tem um escopo limitado, descrevendo explicitamente sua perspectiva.

Um ponto de vista no *Preview* é composto por:

- Nome: Identifica o ponto de vista, geralmente refletindo seu foco.
- Foco: Perspectiva adotada pelo ponto de vista.
- *Concern*: Preocupação, reflete o objetivo organizacional, comercial e limites que dirigem o processo de análise.

⁵[SOM 97]

- Fontes: Pessoas, documentos, coisas que foram utilizadas para criar o ponto de vista.
- Requisitos: Os requisitos do ponto de vista.
- Histórico: Mudanças efetuadas no ponto de vista.

Como dito anteriormente, a análise é dirigida pelos *concerns*. Eles refletem os objetivos estratégicos do sistema. Por exemplo: segurança, manutenibilidade, disponibilidade. Os *concerns* podem ser divididos em *sub-concerns* e assim por diante. A cada um deles são associadas perguntas, sendo estas utilizadas para dirigir o processo de descoberta de requisitos e servindo como um *checklist* durante a análise dos mesmos.

O foco é a característica que determina o ponto de vista. Cada ponto de vista tem um foco único, podendo, eventualmente, sobrepor os focos de outros pontos de vista, caracterizando assim fontes de possíveis conflitos. De certa forma, o foco mapeia itens do domínio da aplicação e do sistema. Veja algumas vantagens que podem ser destacadas da utilização de focos:

- Provê uma base para análise de cobertura.
- Ajuda a identificar pontos de vista com requisitos conflitantes.
- Permite descobrir fontes de requisitos.
- Os focos que se restringem ao domínio da aplicação ajudam a identificar pontos de vista que encapsulam requisitos reusáveis.

Com a explicação a respeito dos itens que compõem um ponto de vista, torna-se possível descrever o processo adotado no Preview:

1. Descoberta dos requisitos
 - (a) Identificação de *concerns*: Determinar que propriedades fundamentais o sistema deve exibir se ele deve ser bem sucedido.
 - (b) Elaboração de *concerns*: Detalhar os *concerns*, especificando seus requisitos não-funcionais.

- (c) Identificação de pontos de vista.
 - (d) Descoberta dos requisitos de cada ponto de vista: Pode levar à decomposição de pontos de vista se os requisitos internos a cada ponto de vista não forem coerentes e conflitarem.
2. Análise dos requisitos: Busca identificar requisitos inconsistentes com os *concerns* e com outros requisitos, descobrindo assim conflitos internos e externos dos pontos de vista. Os conflitos internos são identificados através de *concerns*, os conflitos externos através do foco.
 3. Negociação de requisitos.

4.4.4 REQAV: Modelo para Descrição, Qualificação, Análise e Validação de Requisitos⁶

A necessidade de uma definição clara do software a ser construído é vital para o processo de engenharia de software. O REQAV é um modelo que aborda esse problema, propondo critérios de valor e peso à informação dos *stakeholders* para estabelecer condições de análise e validação dos requisitos.

O processo é composto por onze etapas, agrupadas em cinco fases: descrição do requisito, qualificação do requisito, qualificação da fonte de informação, aplicação de parâmetros de qualificação e composição do quadro de avaliação de risco de implementação do requisito.

A descrição dos requisitos consiste em planejamento, pesquisa inicial do material existente, identificação do stakeholder, descrição inicial dos requisitos, estruturação dos dados e composição da versão inicial do documento de requisitos. Ao final desta etapa, gera-se um documento preliminar de descrição de requisitos e um quadro descriptivo de requisitos.

A fase de qualificação dos requisitos obtém a qualificação de cada requisito e a relação de dependência entre eles, analisando, para isso, três aspectos: qualificação funcional, área de origem e a relação de dependência entre eles. Adotando uma qualificação

⁶[ZAN 2000]

variando de 1 a n, teríamos n^3 possíveis combinações (ou seja, n^3 níveis de qualificação do requisito).

A qualificação da fonte de informação obtém a qualificação do stackholder em função do seu ponto de vista, sua qualificação funcional na organização e a exigência da informação. O raciocínio segue o mesmo da qualificação dos requisitos (n^3 níveis de qualificação possíveis).

A aplicação de parâmetros de qualificação compreende a apropriação dos resultados das etapas de qualificação do requisito, qualificação da fonte de informação e o comparativo dos resultados para avaliação de risco.

Finalmente, temos a fase de composição do quadro de avaliação de risco. Esta consiste em, a partir da avaliação das informações obtidas na qualificação dos requisitos e das fontes de informações, juntamente com a aplicação de parâmetros de qualificação, gerar um quadro de avaliação de risco.

A aplicação do modelo proposto possui inúmeras vantagens:

- Os critérios adotados permitem uma visualização dos requisitos prioritários.
- Estes mesmos critérios possibilitam identificar requisitos que terão de ser revisados.
- A aplicação do modelo facilita a manutenção do foco durante o desenvolvimento do sistema.
- As informações geradas durante o processo servem de fundamento para a negociação dos requisitos.

4.4.5 Usando diferentes meios de comunicação na negociação de requisitos⁷

Há tempos percebe-se a necessidade de aproximação dos clientes e desenvolvedores para definir os requisitos, principalmente para resolver os conflitos encontrados. Sempre

⁷[DAM 2000]

imaginou-se que a maneira mais efetiva de fazê-lo era através de um encontro cara a cara entre as pessoas que vão negociar os conflitos. Em [DAM 2000], investiga-se a performance de grupo e relacionamento interpessoal na engenharia de requisitos distribuída, confrontando-se, então, a comunicação utilizando o computador e seus recursos multimídias (som e imagem) como a forma de comunicação que até então se acreditava ser mais efetiva.

Fora analisado o efeito da comunicação no desempenho do grupo na negociação de requisitos e os efeitos da configuração do grupo. Tomam-se como variáveis independentes o modo de comunicação e o arranjo do grupo. As variáveis dependentes são o desempenho do grupo e percepção pessoal. Destas variáveis, a mais importante é a de desempenho do grupo na negociação dos requisitos. Esta negociação pode ser distributiva (os conflitos são resolvidos através da eliminação de um, ou seja, o sistema atende somente uma parcela dos *stakeholders*) ou integrativa (os conflitos são negociados e, no fim, atende-se os requisitos de todos os envolvidos da melhor maneira possível).

O experimento retratado em [DAM 2000] consistiu na negociação de requisitos funcionais de um sistema de gerenciamento bancário. Estudaram-se cinco configurações de grupo: uma cara a cara e outras quatro distribuídas. As configurações utilizadas podem ser vistas na figura 4.7.

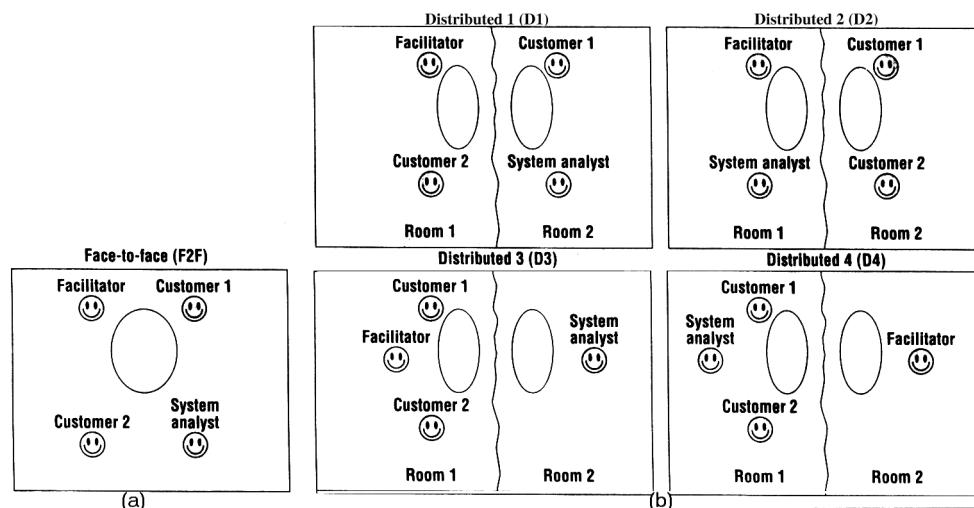


Figura 4.7: Cinco configurações de grupo: (a) cara a cara e (b) distribuída. [DAM 2000]

Os resultados do experimento são interessantes. Verificou-se que a comunicação utilizando o computador como meio é tão eficiente quanto e até melhor que a comunicação

face a face. Mais ainda, observou-se em D1 os melhores resultados, melhores até que o F2F (que fora utilizado como referência para comparação). A explicação para este resultado é que, em D1, os *stakeholders* estavam separados. Outro fato interessante foi que em D2 e D3, no qual o analista de sistema está separado dos clientes e estes estão juntos, a negociação dos requisitos foi distributiva, consequência da persuasão que um cliente exerce sobre o outro, da proximidade entre as pessoas e relacionamentos interpessoais. Isto foi confirmado pela análise da percepção pessoal, na qual as pessoas, apesar de gostarem desta proximidade, mencionam que esta permite que uma pessoa influencie a outra mais facilmente, o que prejudica a realização da tarefa. A reduzida capacidade de perceber as emoções das pessoas, como no caso D1, permitiram aos clientes um melhor entendimento das necessidades, um raciocínio mais claro e possibilitaram que o analista de sistema se mantivesse mais imparcial.

4.4.6 Análise das diversas técnicas e métodos de engenharia de requisitos estudadas

Implementar uma ferramenta que automatize a definição de requisitos, de acordo com a metodologia MDSODI, utilizando apenas uma das abordagens estudadas, talvez fosse insuficiente, para não dizer um desperdício. Por exemplo, a idéia de seleção de requisitos com base em conhecimento aplicada a pontos de vista permite a determinação de prioridades de maneira fácil e transparente, algo muito desejável em um sistema no qual teremos um universo de atores muito distintos que geram uma quantidade impressionante de requisitos. Por mais que se queira, é impossível atender as necessidades de todos. Deve-se, portanto, se concentrar no foco do problema e, na medida do possível, atender as outras necessidades de menor prioridade.

A utilização de padrões de casos de uso permite casar modelos de projetos diferentes. Esta possibilidade de reuso tão prematura permitiria o reaproveitamento de arquiteturas, a descoberta de requisitos que tinham sido esquecidos. No caso de sistemas desenvolvidos com base em casos de uso (utilizando o Unified Process, por exemplo), o nível de reaproveitamento seria ótimo. No entanto, sua implementação através de um pequeno sistema especialista talvez não seja suficientemente boa. Provavelmente, nos sistemas objetivados pela ferramenta proposta neste trabalho, os modelos de caso de uso com que iremos nos deparar serão muito grandes ou com um nível de

complexidade tal que impediria a fácil utilização de uma solução como a proposta em [RID 2000]. O ideal seria a identificação automática de padrões através da análise dos grafos correspondentes e casamento de padrões.

Na tabela 4.1, apresenta-se uma breve avaliação das diferentes técnicas analisadas. Os dados para tal avaliação vêm dos problemas apontados nos próprios artigos estudados [RID 2000, TOR 2000, EAS 95, KOT 95, SOM 97, ZAN 2000, DAM 2000] e do entendimento quanto a cada técnica estudada. A não disponibilidade de ferramentas que automatizem a aplicação de tais técnicas impede a coleta de dados concretos e, consequentemente, uma comparação mais apurada:

Técnica	Abrangência	Usabilidade	Implementabilidade	Disseminação
Ponto de Vista	Boa	Boa	Boa	Muito Boa
REQAV	Boa	Razoável	Ruim	Boa
Padrões na Construção de Cenários	Razoável	Boa	Ruim	Boa
Padrões de Reutilização de Requisitos	Boa	Ruim	Boa	Boa
Utilização de diferentes meios de comunicação	Boa	Boa	Ruim	Boa

Tabela 4.1: Comparação das técnicas estudadas.

Apesar de ser uma avaliação superficial, ela é melhor do que nada para ajudar na escolha das técnicas a serem implementadas na ferramenta. A técnica de ponto de vista se destaca por ter poucos pontos negativos. Todas as restantes possuem algumas características não satisfatórias, dificultando um pouco a escolha de uma técnica complementar. Considerando a utilização de pontos de vista e a consequente necessidade de filtrar os inúmeros pontos de vista existentes (e visões geradas), a REQAV se destaca, podendo aplicar seus conceitos em um processo rápido e eficiente para o requerido processo.

4.5 Considerações finais

A engenharia de requisitos é de suma importância no processo de engenharia de um software. Com o auxílio de técnicas específicas para esta etapa, torna-se possível a obtenção de especificações de requisito mais rapidamente e de alta qualidade, ao mesmo tempo que, com o auxílio de ferramentas, beneficia-se de uma integração com o processo como um todo, apoiando as demais etapas do desenvolvimento do programa e facilitando a realização de um produto final de qualidade e em curto prazo.

Capítulo 5

Guia para Engenharia de Requisitos

5.1 Introdução

Alcançada uma compreensão dos vários métodos existentes para a engenharia de requisitos, chega o momento de identificar um novo processo que concilie o proposto na MDSODI e técnicas específicas à requisitos previamente estudadas.

O método sugerido para este trabalho é organizados em quatro fases: criação de um modelo de negócio, descoberta de *stakeholders*, captura de visões, identificação de casos de uso e atores, estruturação do modelo, análise e resolução de conflitos. A execução destas fases não é em cascata e sim feita em várias iterações.

5.2 Conceitos básicos

5.3 Criação de um modelo sintetizado

O modelo sintetizado de caso de uso visa estabelecer as principais funcionalidades que o sistema deverá oferecer. Não consiste em um detalhamento dos processos que o software a ser desenvolvido virá a desempenhar, trata-se simplesmente de uma síntese do que será o sistema, delimitando assim o escopo, o domínio da solução. A representação deste modelo pode ser feita através de uma diagrama de caso de uso.

5.4 Descoberta de stakeholders

Os *stakeholders*, no contexto deste método, não são somente as pessoas que têm poder de decisão no projeto. Define-se aqui uma quantização deste poder de decisão, ou seja, todas as pessoas com algum poder de influência nos requisitos do projeto são consideradas. O quanto relevante será o requisito dos *stakeholders* será em função de seus atributos, seus pontos de vista e suas visões.

O primeiro *stakeholder* definido num projeto é o engenheiro de requisitos. Ele será o responsável por criar as primeiras visões e, a partir destas, criar um esboço de um modelo de negócio (conseqüentemente criando casos de uso e atores). Com este primeiro produto, identificam-se os próximos *stakeholders*.

Para cada ator tem-se ao menos um *stakeholder* responsável (na primeira iteração, o engenheiro de requisitos será o *stakeholder* relacionado aos elementos deste modelo). Na maioria dos casos, o *stakeholder* acaba por se tornar efetivamente um ou mais atores (mas nem sempre, como, por exemplo, o dono da empresa, criando restrições organizacionais que influenciam os casos de uso mas não caracterizam sua participação nos mesmos).

5.5 Captura de visões

A visão é um artefato contendo uma descrição, crítica ou correção criada por um *stakeholder* tendo como alvo alguma parte do sistema. A fim de melhor avaliá-la, especifica-se também o aspecto sobre o qual a visão se aplica, um raciocínio para justificar o seu conteúdo (se necessário) e o grau de importância (ou severidade) da visão.

O conteúdo da visão é tratado como um conjunto de requisitos, podendo estes ser uma descrição, uma crítica ou correção. Uma descrição é um texto que expõe, em detalhes, o funcionamento ou exigências e restrições. A crítica vem salientar possíveis erros na especificação do alvo especificado na visão, diferentemente da correção, que é uma afirmação de que a especificação está incorreta.

Não incomum é a necessidade de justificar uma visão, principalmente aquelas que tem requisitos do tipo correção. Por este motivo, pode-se fundamentar o raciocínio com exemplos, documentos oficiais ou qualquer outro tipo de dado.

A criação de visões pode ser feita com base em modelos pré-cadastrado no sistema. Por exemplo, uma visão sobre o sistema de autenticação de usuários. Ao invés do *stakeholder* fazer toda a especificação da visão, ele se utiliza do modelo já existente, apenas restando-lhe a tarefa de adequar algum detalhe. A reutilização evita a duplicação de esforços, a possibilidade de repetição de erros e agiliza o processo. Mais ainda, como uma visão armazena o modelo do qual ela foi criada, uma alteração no modelo invalidará a visão, facilitando possíveis manutenções. Se a mudança no modelo não se aplicar à visão em questão, basta restaurar a versão anterior ou validar a atual.

5.6 Identificação de casos de uso e atores

Após definidas algumas visões, pode-se começar a identificação por atores e casos de uso. A ferramenta não dispõe de mecanismos para interpretar as visões e gerar prováveis casos de uso e atores, esta tarefa ainda cabe ao engenheiro. No entanto, é possível classificar automaticamente as visões com base em seu conteúdo e do *stakeholder* que a criou. Este mecanismo é extremamente importante já que o número de visões existentes no sistema tende a ser bem grande.

A classificação de visões pode utilizar como parâmetro qualquer objeto que esteja relacionado na visão. Por exemplo: os tipos de requisito, a prioridade (grau de severidade), o aspecto especificado, o grau de importância do *stakeholder* que a originou, a existência de justificativa. Na verdade, as regras são *scripts* armazenados em um banco de regras. Assim, pode-se escolher regras de acordo com o projeto e o que deseja ser analisado na especificação. Nas primeiras iterações, pode-se desejar usar regras que valorizem mais visões com requisitos do tipo descrição do que crítica e correção. Nas iterações posteriores, pode-se enfatizar apenas os que são de correção. Assim, o engenheiro, no momento em que analisar as visões, terá apenas um subconjunto para trabalhar, ficando mais simples focar seu trabalho.

5.7 Análise e resolução de conflitos

Com as primeiras análises feitas, torna-se possível descobrir os atores e os casos de uso do sistema a partir das visões e, a partir disto, definir os relacionamentos entre eles, formando assim o embrião do modelo de caso de uso. A criação de atores e casos de uso também pode ser realizada a partir dos respectivos modelos assim como pode-se testar um ator ou caso de uso através deste modelo, verificando assim se o mesmo está definido corretamente.

5.8 Considerações finais

Aqui foram apresentadas alguma diretrizes sobre o que seria desejável em um método para engenharia de requisitos utilizando as técnicas de visão e qualificação de visões e requisitos. Uma melhor elaboração do método, definindo com precisão seus processos, atividades e artefatos, assim como a integração dele com outros métodos, gerando um processo de engenharia de software completo, é um ponto que deve ser abordado em futuras pesquisas devido a sua grande importância.

Capítulo 6

Ferramenta Proposta

6.1 Introdução

Primeiramente serão detalhadas as tecnologias estudadas para a implementação da ferramenta: ORBs como meio de comunicação de objetos e fornecimento de serviços básicos, mecanismos de persistência baseados em banco de dados, adaptação e melhoria de soluções desenvolvidas em trabalhos anteriores. Descreve-se também a maneira como foi implementada as extensões da MDSODI quanto aos casos de usos, por meio dos mecanismos de extensão da UML. Enfim, tem-se uma explicação sobre a ferramenta e sua especificação.

6.2 CORBA

Um sistema distribuído consiste de vários componentes, localizados em computadores ligados por uma rede, que se comunicam e coordenam através de passagem de mensagens. Disto derivam-se várias características: concorrência de componentes, ausência de uma hora global (dificultando a sincronização) e falhas dos componentes. Um sistema deste tipo precisa atender características tais como segurança, independência de escala, abertura, heterogeneidade dos componentes, transparência. Construir um sistema com tamanha capacidade é, obviamente, uma tarefa difícil. A fim de facilitar a realização de tais sistemas, várias tecnologias foram desenvolvidas: CORBA, RMI, Jini, DCOM, SOAP. O CORBA se destaca por ser uma solução aberta, madura, com várias implementações gratuitas e livres, capacidade de funcionamento em ambientes heterogêneos. Além disto, ele foi utilizado com bons resultados em projetos anteriores [BES 2000], o que encorajou sua utilização neste projeto.

O Common Object Request Broker Arquitecture (CORBA) foi criado pela Object

Management Group¹ (OMG), uma organização internacional com mais de 800 membros (empresas, engenheiros de software e usuários). Seu objetivo é servir como plataforma para a construção de sistemas distribuídos baseados em objetos e, ao mesmo tempo, possibilitar a utilização de software legado no sistema. Sua arquitetura compõe-se de quatro componentes:

- Object Request Broker (ORB): permite a comunicação transparente entre os objetos no sistema distribuído
- Object Service: provêem serviços de baixo nível, tal como localização, persistência.
- Common Facility: são ferramentas que permitem a construção de sistemas num domínio específico.
- Application Object: são as aplicações existentes no sistema distribuído, geralmente feitas com auxílio dos serviços e facilidades CORBA.

O ORB provê meios para que requisições sejam feitas pelos objetos de maneira transparente, provendo assim interoperabilidade entre aplicações em diferentes máquinas em sistemas distribuídos heterogêneos.

Para fazer uma requisição, um cliente pode utilizar-se da invocação dinâmica ou de stubs IDL. O ORB, por sua vez, precisa repassar a requisição para a implementação do objeto requerido apropriadamente. Isso pode ser feito utilizando-se esqueletos IDL ou através de esqueletos dinâmicos. Essa flexibilidade se deve aos mecanismos de definição de interfaces do CORBA: interfaces estáticas definidas com a linguagem IDL (Interface Definition Language) e repositórios de interfaces, no qual as interfaces são definidas em tempo de execução. A figura 6.1 mostra esta estratégia.

Sua complexidade é proporcional à sua importância na arquitetura, sendo necessário o estabelecimento de interfaces padronizadas quando visto de fora e proprietárias quanto a arquitetura interna de cada ORB. Na figura 6.2 é possível observar as diversas interfaces do CORBA. As interfaces preenchidas em preto são comuns a todo ORB enquanto que as cinza são proprietárias, variando de implementação para implementação.

¹<http://www.omg.org>

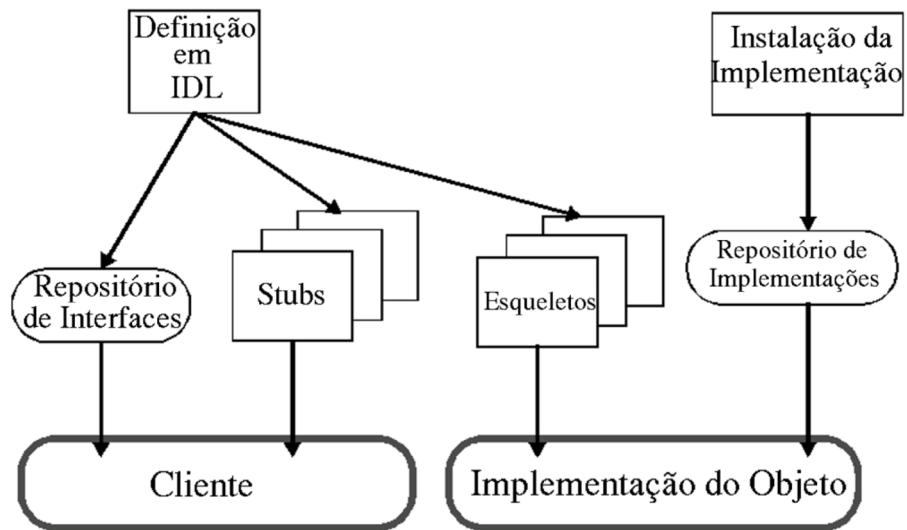


Figura 6.1: Mecanismos para requisição e acionamento de objetos

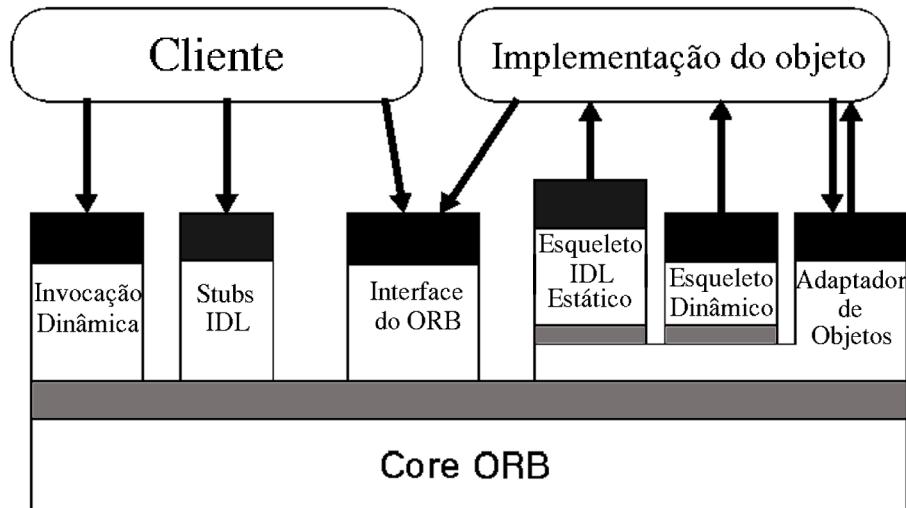


Figura 6.2: Interfaces do Object Request Broker

6.3 Persistência utilizando banco de dados

A necessidade por consultas eficientes dentre os casos de uso e pontos de vista do sistema tornam a utilização de persistência nativa do Java, baseada em serialização em arquivo, inadequada: a velocidade seria baixa, o acesso concorrente seria complexo. Após uma extensa pesquisa, verificou-se que o banco de dados PostgreSQL possuía mecanismos de serialização compatíveis com o Java, porém armazenando os dados em sua base dados.

Aliando a isto a possibilidade de criar um Corba Query Service utilizando este sistema gerenciador de banco de dados, qualquer tipo de pesquisa a ser efetuada torna-se muito mais rápida e prática.

Primeiro, necessita-se de uma explicação sobre o PostgreSQL. Ele é um sistema gerenciador de banco de dados objeto-relacional, suporta *stored procedures*, transações, *clustering*, enfim, uma gama interessante de recursos. E, não menos importante, ele é disponibilizado sob a licença BSD, tratando-se portanto de um software livre, facilitando possíveis modificações que tenham de ser feitas em seu código.

O PostgreSQL possui um driver JDBC (Java Database Connectivity) *Type 4*, incluindo algumas extensões interessantes. Uma delas permite a serialização de objetos para o banco de dados através de objetos da classe org.postgresql.util.Serialize. Ela implementa um mecanismo que possibilita armazenar as referências a objetos encontradas num objeto Java, para isso criando tabelas nas quais os campos podem ser nomes de outras tabelas. Por exemplo:

```
test=> create table users (username name, fullname text);
CREATE
test=> create table server (servername name, adminuser users);
CREATE
test=> insert into users values ('peter', 'Peter Mount');
INSERT 2610132 1
test=> insert into server values ('maidast', 2610132::users);
INSERT 2610133 1
test=> select * from users;
username|fullname
-----+-----
peter   |Peter Mount
(1 row)

test=> select * from server;
servername|adminuser
-----+-----
maidast   | 2610132
(1 row)
```

Na tabela *server*, como pode ser notado, criou-se uma referência ao usuário *peter* da tabela *users* (que possui um número de identificação 2610132), ou seja, torna-se possível guardar, além de atributos como *String*, *int* e *double*, referências para outros objetos. Isso permitirá restaurar todos os objetos associados a um em específico, tornando mais simples a realização de persistência.

Além destas facilidades, outra, não relacionada à persistência mas extremamente importante, é o tipo *text*. Ele permite o armazenamento de cadeias de caracteres de

tamanho arbitrário e permite que campos deste tipo sejam pesquisados, semelhantemente a uma pesquisa em campos do tipo *varchar*. Isto será de extrema valia quando realizando a serialização de objetos Java com atributos do tipo *String*, possivelmente com muitos parágrafos.

6.4 Metodologia de Desenvolvimento Baseado em Objetos Distribuídos Inteligentes[HUZ 99]

A Metodologia de Desenvolvimento Baseado em Objetos Distribuídos Inteligentes (MDSODI) é uma metodologia de engenharia de software dirigida a casos de uso, centrada na arquitetura e com desenvolvimento iterativo e incremental, semelhantemente ao *Unified Process* [JAC 99]. Seu grande diferencial é a identificação e análise, desde o início de seu desenvolvimento, dos aspectos distribuídos do software: concorrência, comunicação, sincronização e distribuição. Estas características são representadas graficamente através de diagramas UML, com extensões propostas na MOOPP [HUZ 95], permitindo uma visualização rápida e eficiente.

A MDSODI é composta por cinco fases: requisitos, análise, projeto, implementação e testes. Estas fases serão detalhadas nas próximas seções, tendo como base [GRA 2000].

Requisitos

Esta fase tem como objetivo identificar as funcionalidades necessárias para o desenvolvimento do sistema de uma forma adequada e eficiente. As atividades que a compõe são:

- Obtenção de informações sobre as funcionalidades do sistema. Sugere-se a utilização de entrevistas com o solicitante do produto, com questões como:
 - Qual o objetivo do programa de uma forma geral?
 - Quais as funções essenciais a serem cumpridas pelo programa?

- Quem serão as pessoas que interagirão com o sistema e qual o nível de conhecimento dos mesmos?
 - Qual a capacidade das máquinas disponíveis?
- Identificação dos requisitos funcionais do sistema com base nas respostas obtidas nas entrevistas.
 - Elaboração do Modelo de Domínio para ajudar o entendimento sobre o contexto do sistema.
 - Elaboração dos requisitos funcionais através de descrição textual.
 - Identificação das pessoas ou grupos que interagirão com o sistema.
 - Identificação dos casos de uso candidados do sistema com base nos requisitos identificados.
 - Identificação dos candidados a atores do sistema.
 - Elaboração de uma versão preliminar do diagrama de casos de uso, desconsiderando requisitos de implementação, utilizando como base os requisitos funcionais, atores e casos de usos previamente identificados.
 - Identificação e elaboração de descrição textual dos requisitos não funcionais do sistema, com especial atenção àqueles relacionados à sistemas distribuídos: concorrência, distribuição.
 - Classificação, quanto aos aspectos distribuídos, dos casos de uso e atores, tendo como base os requisitos não funcionais previamente identificados. Têm-se as seguintes possibilidades:
 - Casos de uso seqüênciais: Casos de uso que agrupam um conjunto de funcionalidades que devem ser executadas seqüencialmente.
 - Casos de uso distribuídos: Casos de uso que podem estar em diferentes locais no sistema.
 - Atores exclusivos: Atores que trabalham seqüencialmente, que disparam a execução de casos de uso de forma seqüencial em relação a outros atores.

- Atores paralelos: Atores que disparam a execução dos casos de uso de forma paralela a outros atores.
- Atores distribuídos: Atores que se encontram localizados em diferentes nós do sistema.
- Atores paralelos e distribuídos: Atores que são, ao mesmo tempo, paralelos e distribuídos.
- Relacionamento seqüencial entre casos de uso: Casos de uso que são executados seqüencialmente uns com os outros. O primeiro caso de uso a ser executada é o que está no ponto final do relacionamento; o que está na ponta inicial, portanto, é o último a ser executado.
- Relacionamento paralelo entre casos de uso: Casos de uso que são executados paralelamente uns com os outros.
- Avaliação e alteração dos diagramas de casos de uso considerando os diferentes tipos de casos de uso que podem existir.
- Elaboração do diagrama de colaboração a partir do modelo de casos de uso proposto para avaliação.

Análise

Consiste na análise dos requisitos descritos na fase anterior e conseguinte refinamento da estrutura, adquirindo-se assim um entendimento mais preciso da descrição dos requisitos. Esta fase é composta das seguintes atividades:

- Análise dos requisitos identificados na fase de requisitos e refinamento da estrutura, verificando-se quais são realmente importantes e eventuais acréscimos necessários.
- Análise dos requisitos, verificando-se a existência de redundâncias e inconsistências.
- Definição das classes e objetos do sistema.
- Definição dos atributos e operações das classes identificadas.

- Análise dos casos de uso definidos em relação às classes existentes, realizando-se as alterações necessárias para uma maior consistência.
- Elaboração do diagrama de classes.
- Identificação dos aspectos de paralelismo, concorrência, distribuição e comunicação no diagrama de classes. As classes e objetos podem ser dos seguintes tipos:
 - Classes e objetos exclusivos: Todas as suas operações são executadas seqüencialmente.
 - Classes e objetos parcialmente paralelos: Algumas de suas operações são executadas seqüencialmente enquanto que outras paralelamente.
 - Classes e objetos totalmente paralelos: Todos as suas operações são executadas concurrentemente.
 - Classes e objetos distribuídos: Classes e objetos localizados em diferentes nós do sistema.
- Reavaliação dos diagramas de caso de uso de acordo com as novas definições de classe, realizando-se alterações se necessário.
- Alteração do diagrama de classes proposto inicialmente, considerando os aspectos de paralelismo/concorrência e distribuição identificados nas classes anteriormente.
- Análise das classes e objetos, certificando-se assim de sua consistência.
- Divisão do diagrama de casos de uso em pacotes. As recomendações para esta divisão são:
 - Casos de uso específicos de um ator devem fazer parte do mesmo pacote.
 - Casos de uso com forte iteração devem fazer parte do mesmo pacote.
- Identificar aspectos de paralelismo, distribuição e sincronização entre pacotes através de descrição textual.
- Elaboração de diagrama de pacotes considerando os aspectos de sistemas distribuídos.

Projeto

O objetivo desta fase é modelar o sistema levando em consideração os requisitos não funcionais e questões como linguagem de programação, interface com o usuário, reutilização de componentes, etc. Esta fase é composta pelas seguintes atividades:

- Identificação da linguagem de programação adequada para o desenvolvimento do sistema de acordo com os requisitos funcionais e não funcionais identificados na fase anterior.
- Identificação da possibilidade da reutilização de componentes.
- Análise dos diagramas de casos de uso, localizando os casos de uso e atores em subsistemas diferentes.
- Análise do diagrama de classes, levando em consideração concorrência e localização de operações, classes e objetos.
- Elaboração do diagrama de seqüência, preocupando-se com aspectos de comunicação entre os diferentes objetos do sistema (parallelismo e sincronização).
- Listagem dos requisitos de implementação, identificados anteriormente, através de descrição textual, especificando: linguagem e ambiente de programação, localização física, questões de concorrência entre operações.
- Divisão do sistema em camadas, objetivando assim um melhor gerenciamento de aspectos funcionais e não funcionais. Os critérios para a divisão são:
 - Camada de aplicação genérica: Aspectos mais gerais e funcionais do programa.
 - Camada de aplicação específica: Aspectos funcionais específicos a cada um dos aspectos genéricos identificados.
 - Camada *middleware*: Aspectos não funcionais como, por exemplo, tipos de serviço oferecidos, aspectos de concorrência/parallelismo.
 - Camada de sistema de *software*: Configurações de rede necessárias para a comunicação entre as demais camadas.

- Detalhamento dos algoritmos a serem utilizados na implementação, com base nas operações identificadas no diagrama de classes.

Implementação

Esta fase tem como objetivo principal a implementação do sistema, tendo como base aspectos identificados nas fases de requisitos, análise e projeto, além de outros aspectos não cobertos totalmente na fase de projeto. As atividades que compõe esta fase são:

- Verificação dos aspectos identificados no projeto no que se refere a implementação (linguagem de implementação, geração de código).
- Definição das interfaces entre os subsistemas identificados na fase de projeto.
- Análise de aspectos arquiteturais, acrescentando a divisão dos subsistemas de projeto, suas interfaces e dependências. Para isto, utiliza-se como entrada a visão arquitetural da fase de projeto, bem como os subsistemas e camadas definidos neste modelo.
- Detalhamento e implementação das operações das classes identificadas nas fases anteriores.
- Identificação de mecanismos de controle de sincronização (por exemplo: exclusão mútua, monitores).
- Identificação de mecanismos a serem utilizados para tratar do balanceamento de carga entre os nós de processamento, considerando os requisitos de distribuição, paralelismo, sincronização e comunicação identificados no projeto.

6.5 Extensão da linguagem UML

As extensões da linguagem UML utilizadas na definição de requisitos são implementadas através do uso de esteriótipos e *tagged values*. Estes mecanismos de extensão da UML, definidos no pacote *Extension Mechanisms* do *Foundation Packages*, foram

feitos justamente para acomodar possíveis dados extras necessários aos modelos e processos de software. Outra possibilidade para acomodar as necessidades deste projeto seria criar uma extensão do metamodelo da UML, definindo as novas metaclasses e metaconstrutores através da MOF. Porém, o uso dos mecanismos de extensão da UML são suficientes para atender as necessidades do trabalho.

No pacote *Extension Mechanisms* da UML, definem-se duas metaclasses: *Stereotype* e *TaggedValue*. Além disso, tem-se a metaclassse *Constraint*, do pacote *Core*. Estes mecanismos podem ser aplicados a qualquer *ModelElement* ou derivado deste, possuindo um valor semântico maior que qualquer outro mecanismo da UML (especialização, relacionamentos).

Um esteriótipo é uma metaclassse que altera a elemento do modelo de maneira que ele pareça ser uma instância de um metamodelo virtual (virtual porque ele não é definido na UML, mas parece que é). Pode haver, no máximo, um esteriótipo associado a um modelo de elemento. Todos os *tagged values* e restrições aplicados em um esteriótipo também são válidos no modelo de elemento, atuando assim como uma pseudo metaclassse descrevendo o elemento. Outra característica interessante é que pode-se derivar um esteriótipo de outro esteriótipo (um esteriótipo é uma especialiação de *modelElement*).

Tagged values são propriedades arbitrárias associadas a um elemento do modelo, representadas por uma tupla (nome,valor). Pode-se associar qualquer número de *tagged values* a um elemento, salvaguardando a restrição destas serem únicas quanto ao elemento.

Constraints permitem que sejam definidas restrições semânticas ao elemento do modelo, utilizando para isto uma linguagem, tal como a OCL (que foi feita especificamente para isto), uma linguagem de programação, notação matemática, linguagem natural. Geralmente utilizam-se linguagens definidas formalmente, possibilitando assim uma aplicação destas regras através de ferramentas. Uma restrição pode possuir o atributo *body* e a associação *constrainedElement*. O corpo *body* pode possuir uma expressão booleana que define a restrição. Esta expressão sempre deve ser verdadeira para instâncias de elementos com esta restrição quando o sistema está estável, ou seja, não está sendo feita nenhuma operação no sistema. Caso contrário, é dito que o modelo está inconsistente. A associação *constrainedElement* é uma lista ordenada dos elementos do modelo sujeitos à uma restrição. Se o elemento em questão for um esteriótipo, todos

os elementos que possuem aquele esteriótipo também estarão sujeitos à restrição.

Estudados os mecanismos de extensão da UML, definiu-se, então, as extensões da UML a serem utilizadas a fim de representarmos nessa linguagem. Escolheu-se pela utilização de *tagged values*. Para os atores, foram definidos os seguintes:

- isDistributed
- isParallel
- isExclusive

Para os casos de uso:

- isSequential
- isDistributed

E, finalmente, para os relacionamentos:

- isSequential
- isParallel

Em nível de implementação, os *tagged values* são armazenados em um Hashtable, tendo como chave o nome da *tag*. Quanto à representação gráfica destes, o procedimento que faz a pintura leva em consideração os *tagged values*, resultando em gráficos semelhantes aos representados nas figuras 6.3, 6.4 e 6.5.

6.6 Ferramenta Proposta

A ferramenta em si não é somente o programa que permite a criação dos diagramas de caso de uso, sendo composta de todo um conjunto de subsistemas que permitem a

(a) Relacionamento entre casos de uso
casos de uso paralelos

(b) Relacionamento entre casos de uso paralelos

Figura 6.3: Representação dos relacionamentos

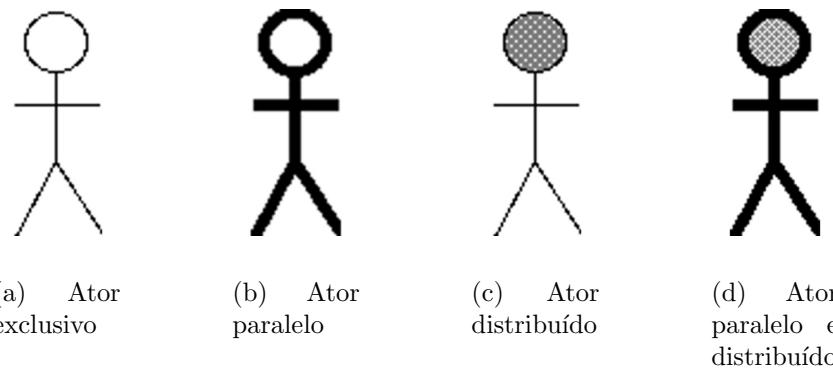


Figura 6.4: Representação dos atores

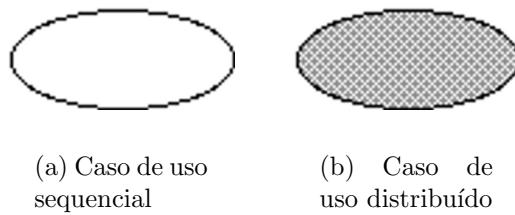


Figura 6.5: Representação dos casos de uso

criação automática de um modelo de caso de uso a partir de visões inseridas no sistema. A seguir será exposta a arquitetura dos principais subsistemas e como eles vão interagir sob o comando da ferramenta gráfica, a CoolCase, a ser utilizada pelo engenheiro de requisitos.

6.6.1 Framework veryhot

A necessidade de criação de diagramas de casos de uso atendendo a notação utilizada neste trabalho motivou a criação de um pacote que facilite esta tarefa. Seu nome é

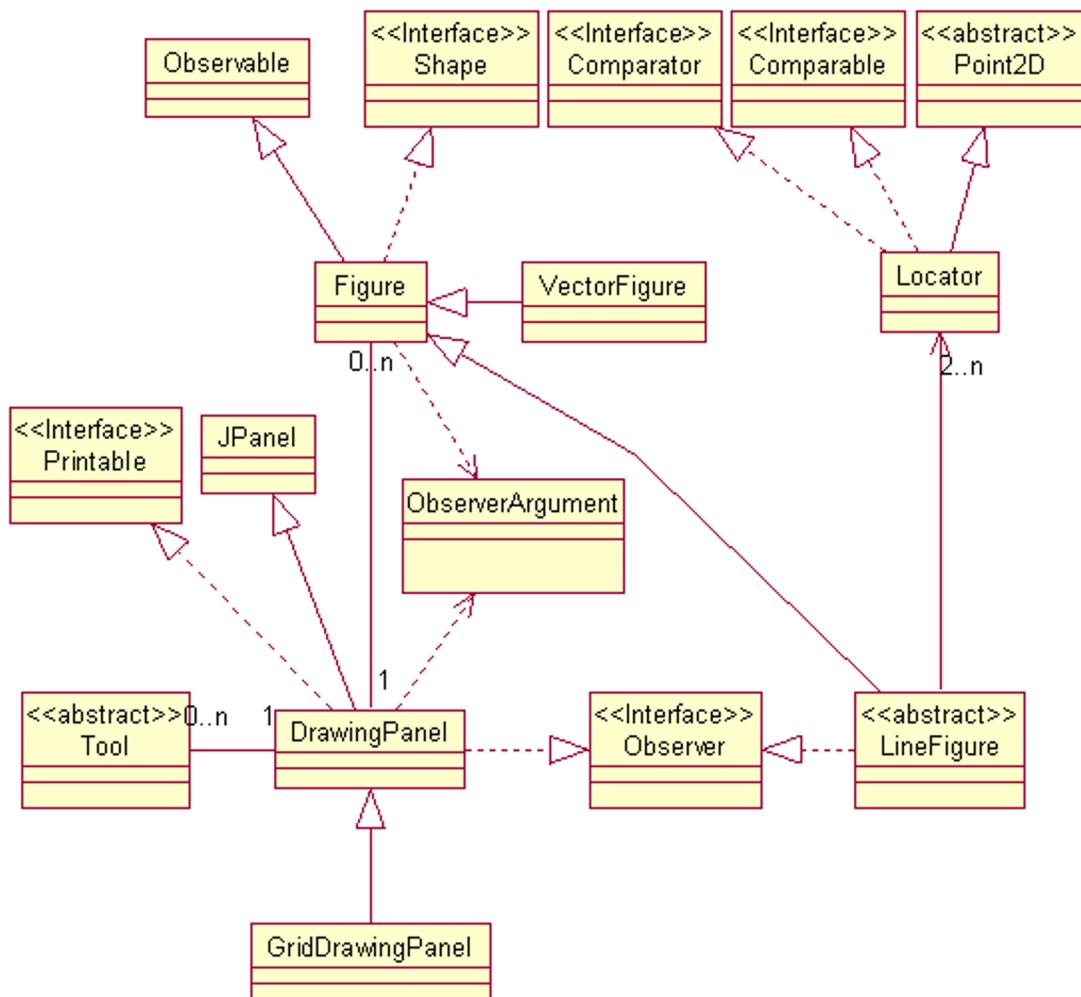
veryhot. Ele é uma evolução de um pacote Java desenvolvido em [YAN 97]. Fora feita uma reengenharia do mesmo, simplificando-o (diminuição de números de classes) sem perder sua funcionalidade original, facilitando futura manutenção.

O sistema possui quatro componentes principais:

- Figure: Figura que pode ser manipulada pelo pacote.
- DrawingPanel: Responsável por armazenar as figuras e desenhá-las adequadamente.
- Tool: Ferramentas que criam e manipulam as figuras contidas no *DrawingPanel*.
- ObserverArgument: Utilizado para repassar as alterações ocorridas em uma figura entre os diversos objetos que a observam.

Nos diagramas 6.6 e 6.7, pode-se observar a estrutura do pacote. Algumas características que merecem ser destacadas:

- Simplicidade: Para criar novas figuras (um *Actor*, por exemplo), basta extender o *VectorFigure* ou criar uma nova classe que especialize a *Figure*.
- Versatilidade das ferramentas: Elas podem modificar as figuras e não precisam se preocupar com o redesenho da figura, podendo o desenvolvedor concentrar-se na funcionalidade das ferramentas e não na apresentação da figuras que manipula.
- Sistema de notificação avançado: A comunicação sobre mudanças nas figuras é assíncrono, baseado em eventos, obtendo um desempenho bom ao mesmo tempo que é de fácil implementação. A utilização do *ObserverArgument* permite passar informações complexas entre os objetos.
- Documentação: este *framework* está muito melhor documentado que a versão utilizada como base, contendo tantos diagramas (mostrados neste trabalho) quanto comentários no código-fonte, permitindo seu uso e manutenção mais facilmente.

Figura 6.6: Diagrama de classes - Pacote `veryhot`

6.6.2 Kernel

Novamente, reutilizando soluções criadas em trabalhos anteriores [BES 2000], decidiu-se por utilizar um kernel para implementar os serviços básicos requeridos por cada serviço. As modificações feitas em relação ao anterior foi a remodelação do sistema quanto ao armazenamento de objetos, retirando componentes como *Cache* e *PersistenceManager* (responsáveis pelos serviços de cache e persistência), substituindo-os por um novo componente, *MemoryManager*. Este, por sua vez, gerencia vários *MemoryDevice*, que são os locais onde os objetos estão efetivamente armazenados. O *MemoryManager* permite a criação de uma hierarquia de memórias, resultando em um gerenciamento mais eficiente e rápido dos objetos. A nova solução também evita possíveis problemas

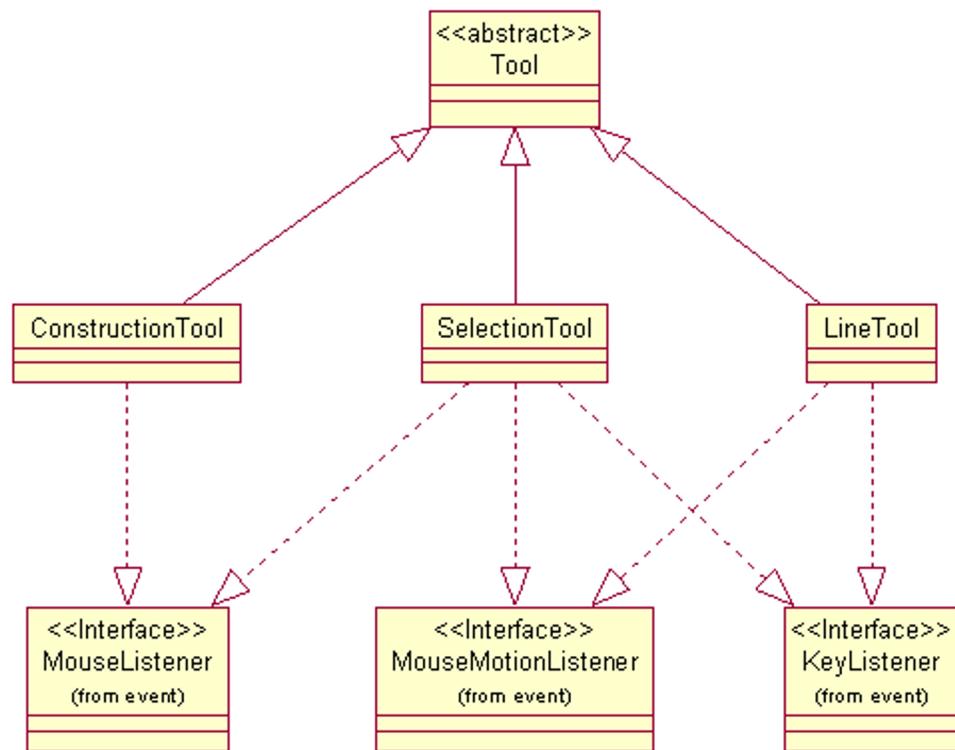


Figura 6.7: Diagrama de classes - Pacote veryhot.tool

de acesso concorrentemente a um mesmo objeto que existia no sistema anterior.

Atualmente, três dispositivos de memória estão implementados:

- `CacheMemory`: Trata-se de uma memória cache, sua função é otimizar o acesso aos objetos, evitando a leitura deles do disco constantemente.
- `DBMemory`: Realiza a persistência do objeto em banco de dados PostgreSQL.
- `CastorMemory`: Realiza a persistência do objeto em arquivo, utilizando o mecanismo de serialização em arquivos disponível no Java.

6.6.3 CoolCase

A ferramenta proposta neste trabalho, doravante denominada CoolCase, tem como elemento visível principal uma ferramenta que permite trabalhar com o modelo de

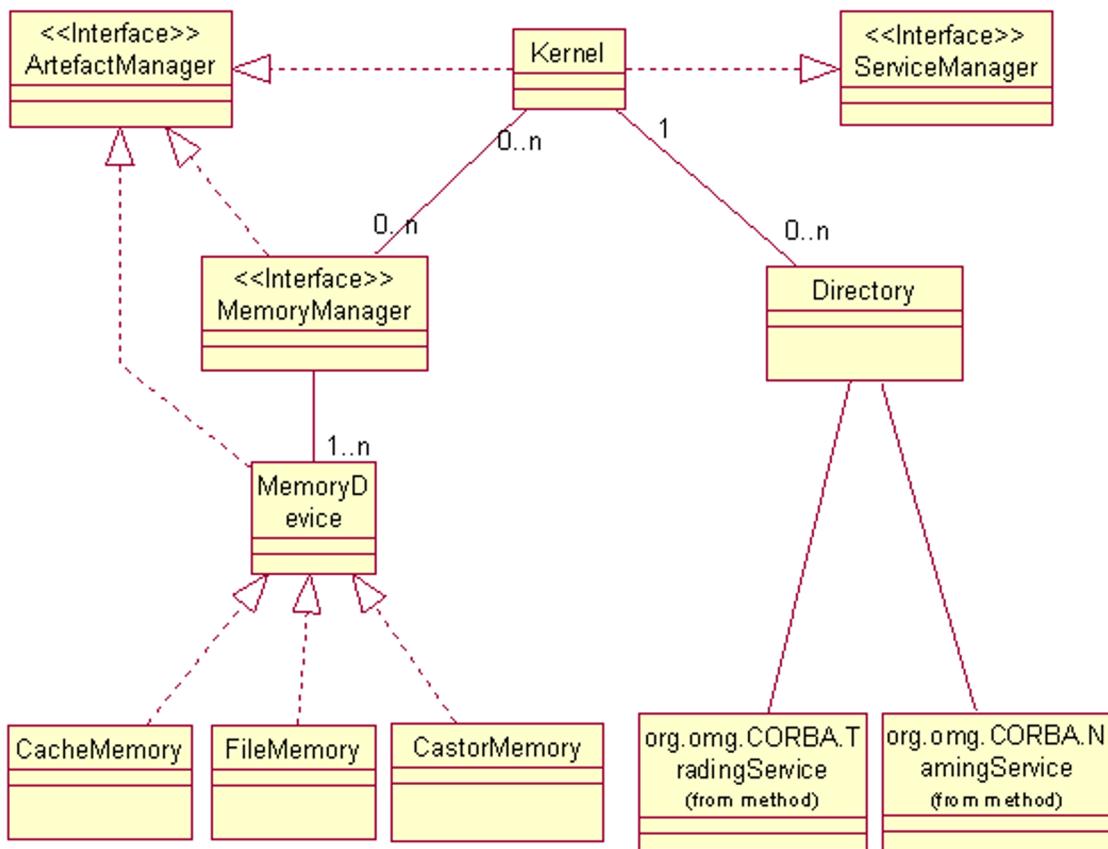


Figura 6.8: Diagrama de classes - Pacote kernel

casos de uso, permitindo ao engenheiro criar e editar diagramas de caso de uso. Na figura 6.9 pode-se observar o diagrama de classes da ferramenta e em 6.10, como são relacionados os elementos gráficos dos diagramas e o artefatos contidos no modelo de caso de uso.

A criação dos diagramas na ferramenta requer a interação com os vários serviços existentes (*EntityService*, *UseCaseModelService*, *ViewService*, *EvaluationService*, *TemplateService*), permitindo a definição e utilização de visões, requisitos e dos serviços que subsidiam a criação dos casos de uso e atores (principalmente o *EvaluationService*):

- Entity Service: Contém todas as entidades do sistema.
- Use Case Model Service: Gerencia os casos de uso, os atores e seus intra-relacionamentos. Na figura 6.13, tem-se o modelo dos mesmos e seus possíveis relacionamentos.

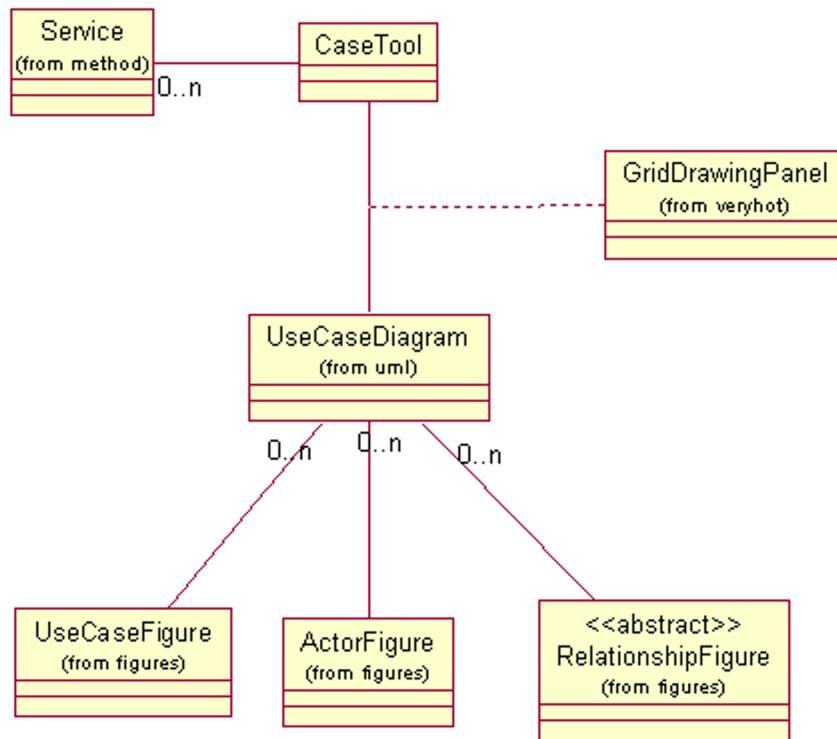


Figura 6.9: Diagrama de classe da CoolCase

- View Service: Gerencia as visões.
- Evalution Service: Aplica regras previamente definidas pelo engenheiro de requisito, permitindo assim avaliar os artefatos do sistema. Utilizado basicamente para classificar as visões de acordo com sua importância relativa ao projeto.
- Template Service: Modelos de artefatos, ajudando a criação de novos artefatos com base em padrões já existentes.

Na figura 6.11, é possível visualizar como eles se relacionam (a relação entre *Tool* e *Service* já foi mostrada na figura 6.9).

A ferramenta manipula sete tipos de objetos: *entity*, *requirement*, *use case*, *actor*, *relationship*, *use case model* e *view*, conforme pode ser visto em 6.12. Vários deles são uma especialização de *Artifact* (figura 6.12, sobre o qual existe um controle de versão, facilitando assim a rastreabilidade de modificações.

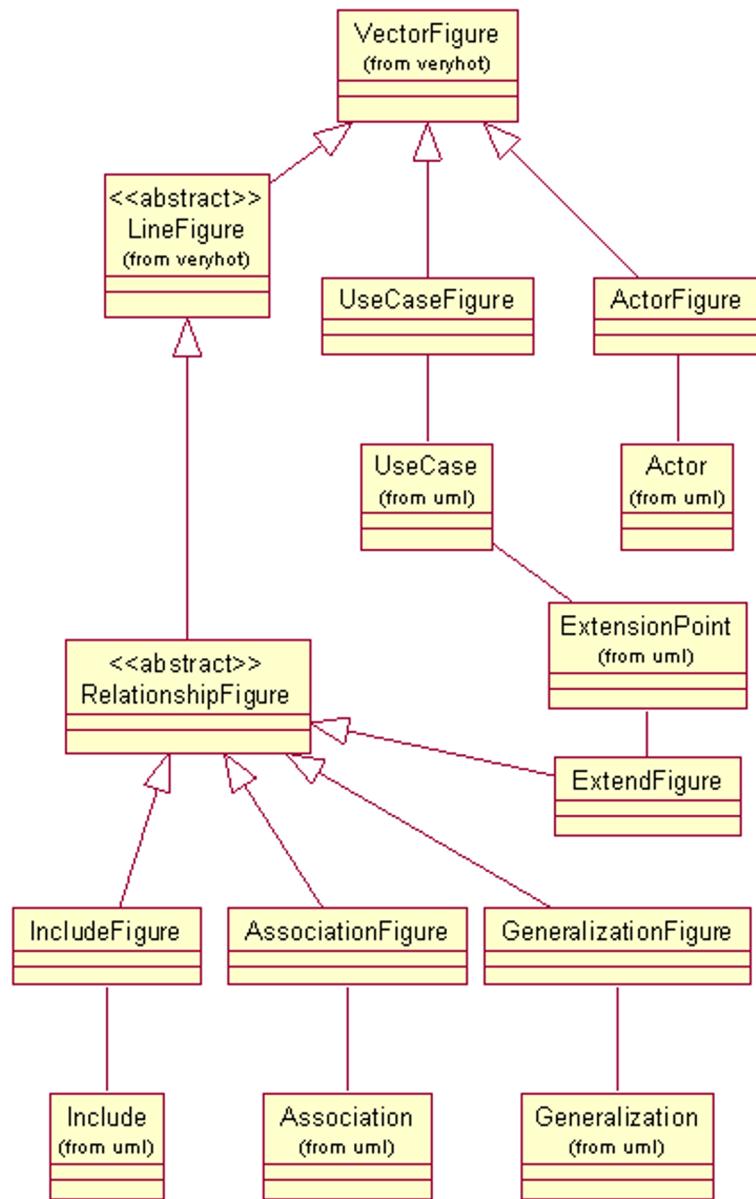


Figura 6.10: Diagrama de classe relacionando figuras e elementos do modelo de caso de uso

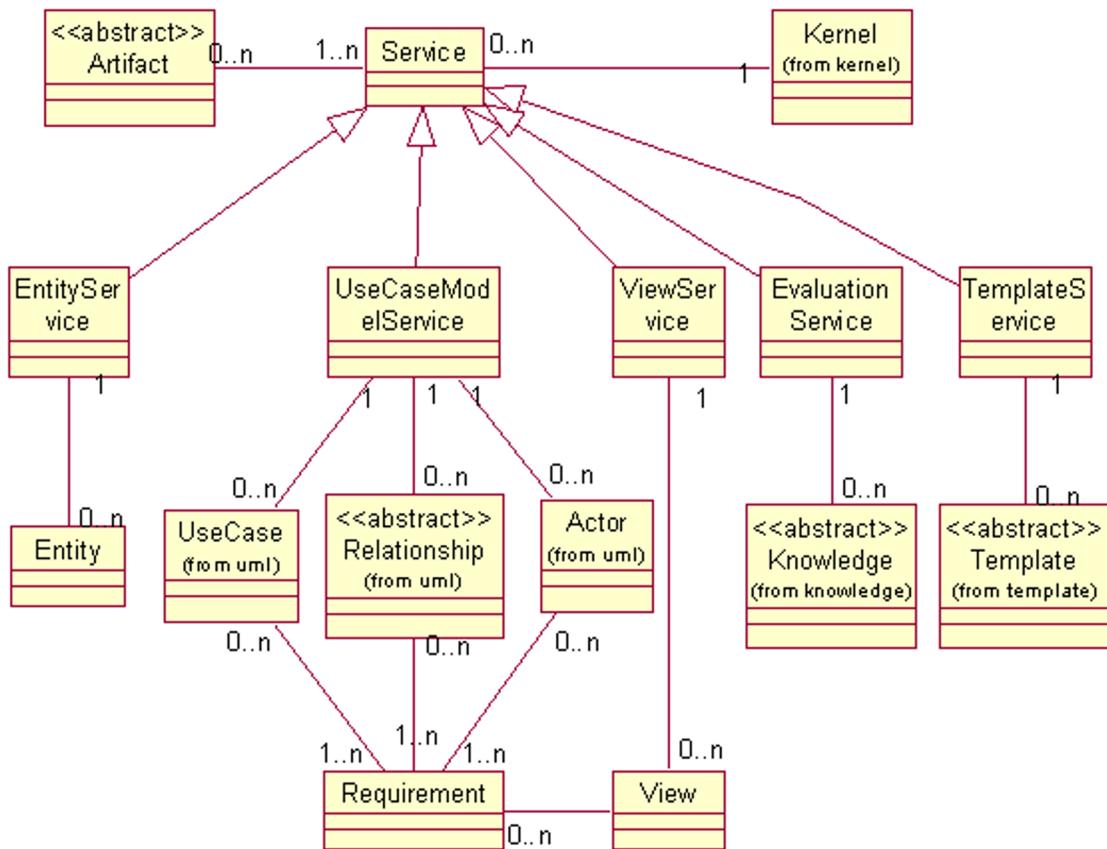


Figura 6.11: Infraestrutura da ferramenta proposta

Entity Toda visão está relacionada a uma entidade, garantindo assim uma certa rastreabilidade. Mas o mais importante é a capacidade de avaliar artefatos, principalmente visões, de acordo com as entidades que as criaram. Para isso, armazena-se vários atributos sobre uma entidade, tanto a nível de sistema como um todo, como também a nível de um projeto específico.

Requirements São informações utilizadas para criar um caso de uso ou ator, os requisitos sobre o sistema. São partes do conteúdo de uma visão. Graças a eles, a rastreabilidade de alterações que causam inconsistência é extremamente facilitada, o que agiliza a resolução de conflitos.

Use Case e Ator Estes seguem a estrutura definida na UML 1.3 contendo as extensões para suporte a MDSODI.

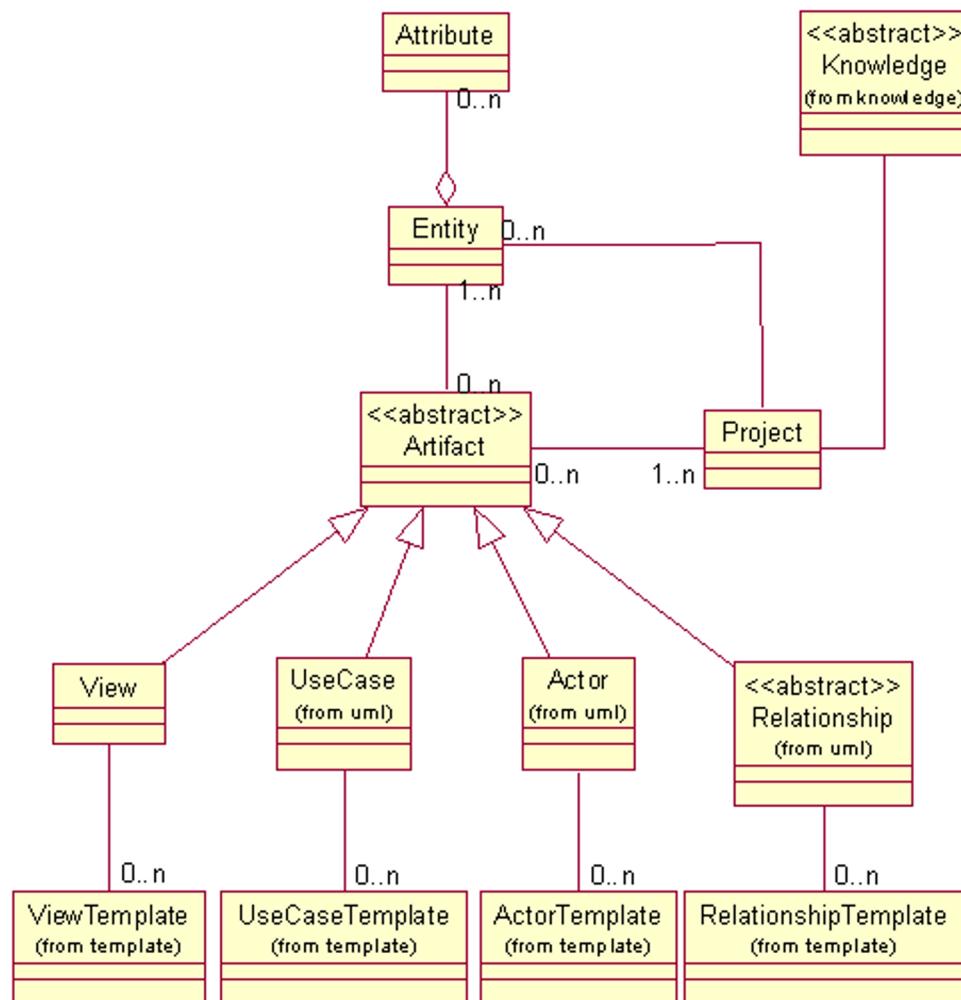


Figura 6.12: Arquitetura básica do sistema

Use Case Model Os modelos de caso de uso são compostos pelos atores, casos de uso e seus relacionamentos. Na figura 6.13, os elementos que compõe o modelo e como eles se relacionam pode ser visualizado.

View As visões são o meio de entrada dos dados necessários para a construção dos requisitos. Seus atributos são:

- Assunto
- Grau de importância (do ponto de vista da entidade que a enviou);

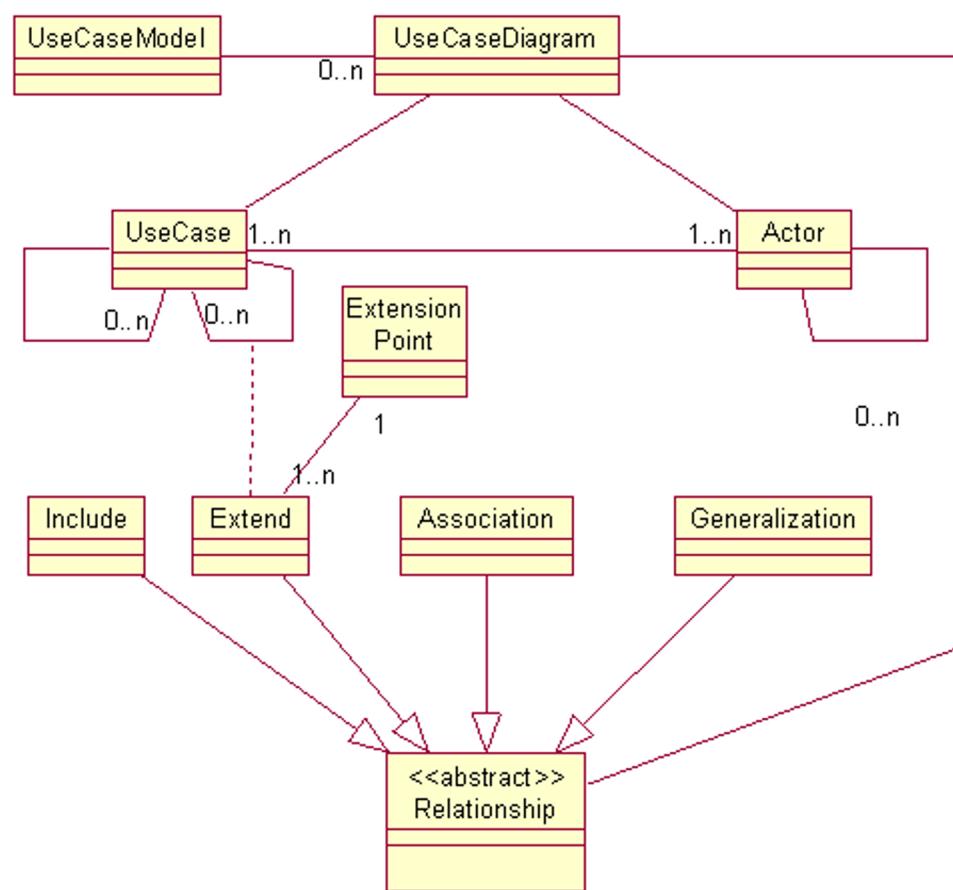


Figura 6.13: Diagrama de classe simplificado da UML quanto aos casos de uso

- Foco (a que componente ou aspecto específico esta visão está tratando)
- Requisitos
- Anexos

Esta estrutura é bem semelhante à do *Preview*. O maior acréscimo é a possibilidade de anexar materiais a uma visão (documentos grandes, sons, coisas que tem importância na explicação de uma visão mas que não são um requisito propriamente dito, geralmente se tratando de uma justificativa do requisito).

Template Os *templates* têm como principal função facilitar a criação de visões. Sua estrutura é muito semelhante a de uma visão, exceto que não possui um grau de importância e seus requisitos são, na verdade, indagações, orientações, informações que visam auxiliar a criação de pontos de vista com o mesmo assunto e/ou foco do modelo.

Outra função desempenhada pelos *templates* é facilitar a avaliação de uma visão em uma revisão ou em um processo de resolução de conflitos, servindo de parâmetro para verificar a validade da visão e a necessidade de dividir uma visão por esta abordar vários assuntos e possuir vários focos, por exemplo.

Knowledge Consiste em uma base de conhecimento sobre artefatos. São fórmulas de avaliação de artefatos, tendo como base em suas características intrínsecas e interrelacionamentos com outros artefatos (como, por exemplo, no modelo de caso de uso). Estas fórmulas são, na verdade, pequenos programas (em Python ou LISP, por exemplo). Isto garante uma flexibilidade para criação de bases de conhecimento mais especializadas para um ou outro tipo de sistema, abrindo caminho para novos projetos quanto a otimização das mesmas, com consequente ganho de produtividade e qualidade dos projetos.

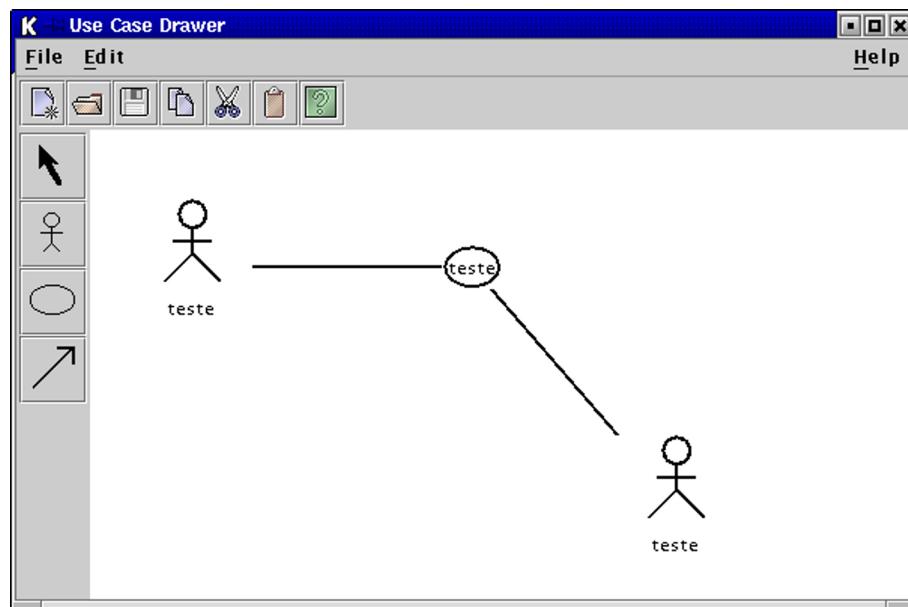


Figura 6.14: Protótipo da ferramenta em funcionamento

6.7 Considerações finais

A ferramenta tem um grande potencial a ser explorado, assim como as demais técnicas descritas neste capítulo. Por exemplo, o desenvolvimento de conjunto de regras, para avaliar visões e requisitos, e modelos, para guiar a criação de visões, para tipos específicos de *software* (sistemas embarcados, sistemas baseado em Web, etc), permitirá a otimização do processo de engenharia de requisitos (e possivelmente do processo de engenharia de software como um todo).

Na figura 6.14, pode-se visualizar um protótipo da ferramenta em execução.

Capítulo 7

Conclusão

Muitos dos trabalhos da área de engenharia de requisitos vêm sido desenvolvidos desde a década de 90. Existem várias técnicas disponíveis, métodos desenvolvidos e processos sendo aperfeiçoados. No entanto, ainda não existem métodos que agreguem várias técnicas, suportando os padrões de projeto dos software atuais e suas notações. Da vasta pesquisa feita por técnicas para definição de requisitos, nenhuma enfoca objetos, quanto mais objetos distribuídos.

A utilização de pontos de vista e visões, reconhecendo e se aproveitando do fato de um software ser construído com base em diferentes *stakeholders*, é de extrema valia, permitindo uma rastreabilidade às origens dos requisitos de maneira extremamente ágil, algo extremamente importante na resolução de conflitos. Os dados destas visões, juntamente com todos os outros dados disponíveis no sistema, através do mecanismo de aplicação de regras, permitem selecionar o que vai ser especificado da melhor maneira possível. Alguns cenários: No início do projeto, adotar regras que valorizem visões de stakeholders com alto poder de decisão e com requisitos de descrição. Em fases mais avançadas, apenas aqueles com requisitos de correção. No início de um novo ciclo, dar uma ênfase nas visões com requisitos de descrição ou crítica e que não estão relacionadas com nenhum caso de uso ou ator, ou seja, que não foram utilizadas no ciclo anterior.

Não menos importante é a metodologia MDSODI, permitindo identificar, prematuramente, características de distribuição e paralelismo dos casos de uso, atores e seus relacionamentos. Durante a realização deste trabalho, observou-se que soluções que permitam a identificação automática destas características são necessárias, porém extremamente complexas. E isto a faz extremamente interessante para alvos de novas pesquisas, já que o impacto do benefício de novas técnicas nesta área abrange todas as outras fases do processo de engenharia de software: análise, projeto, implementação, teste.

O protótipo da ferramenta ainda tem alguns pontos a melhorar. Seu mecanismo de

armazenamento de objetos ainda não possui um controle de versão, algo que é necessário. A API definida *aparenta* ser satisfatória. No entanto, desenvolver um mecanismo de persistência que permita consultas com alto desempenho, utilize cache, permita um uso transparente dos objetos e que faça o gerenciamento de configuração provavelmente exigirá alguns ajustes.

Referências Bibliográficas

- [BAL 91] BALZER, R. Tolerating inconsistency. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 13., 1991, Austin, Texas, USA. **Anais...** IEEE Computer Society Press, 1991. p.158–165.
- [BES 95] BESNARD, P.; HUNTER, A. Quasi-classical logic: non-trivializable classical reasoning from inconsistency information. **Symbolic and Quantitative Approaches to Uncertainty**, 1995.
- [BES 2000] BESSANI, A. N. **Uma arquitetura baseada na web para um ambiente de engenharia de software orientado a processos**. Relatório de Iniciação Científica.
- [BOR 2000] BORTOLI, L. A.; PRICE, A. M. Um método para auxiliar a definição de requisitos. In: IDEAS 2000: JORNADA IBERO AMERICANA DE INGENIERIA DE REQUISITOS Y AMBIENTES DE SOFTWARE, 2000, Cancun-México. **Anais...** IDEAS 2000: Jornada Ibero Americana de Ingeneria de Requisitos Y Ambientes de Software, 2000. p.1–12.
- [BOS 98] BOSAK, J. et al. **Extensible markup language (xml) 1.0**. 1.ed. [S.l.: s.n.], 1998.
- [CAL 2000] CALDWELL, N. H. M. et al. Web-based knowledge management for distributed design. **IEEE Intelligent Systems**, p.40–47, May/June 2000.
- [COU 2001] COULOURIS, G.; DOLLIMORE, J.; KINDBER, T. **Distributed systems concepts and design**. 3.ed. [S.l.]: Addison-Wesley, 2001. (International Computer Science Series).
- [DAM 2000] DAMIAN, D. E. H. et al. Using different communication media in requirements negotiation. **IEEE Software**, p.28–36, May-June 2000.
- [DÍAS 99] DÍAS, I.; METTEO, A. Objectory process stereotypes. **JOOP**, p.29–38, June 1999.
- [DIA 2000] DIAZ, J. S.; FERRAGUD, V. P.; PELOZO, E. I. Un entorno de generación de prototípico de interfaces de usuário a partir de diagramas de

- interacción. In: IDEAS 2000: JORNADA IBERO AMERICANA DE IN-
GENERIA DE REQUISITOS Y AMBIENTES DE SOFTWARE, 2000,
Cancun-México. **Anais...** IDEAS 2000: Jornada Ibero Americana de
Ingeneria de Requisitos Y Ambientes de Software, 2000. p.145–154.
- [DRU 85] DRUCKER, P. F. The discipline of innovation. **Harvard Business Review**, p.149–157, Maio-Junho 1985.
- [EAS 95] EASTERBROOK, S.; NUSEIBEH, B. Using viewpoints for inconsistency management. **IEEE Software Engineering Journal**, November 1995.
- [FIC 91] FICKAS, S.; LAMSWEERDE, A. van; DARDENNE, A. Goal-directed concept acquisition in requirements elicitation. In: INTERNATIONAL WORKSHOP ON SOFTWARE SPECIFICATION AND DESIGN, 6., 1991, Como, Italy. **Anais...** [S.l.: s.n.], 1991. p.14–21.
- [FIN 90] FINKELSTEIN, A.; KRAMER, J.; GOEDICKE, J. K. Viewpoints oriented software specification. In: INTERNATIONAL WORKSHOP ON SOFTWARE ENGINEERING AND ITS APPLICATIONS, 3., 1990, Toulouse, France. **Anais...** [S.l.: s.n.], 1990. p.337–351.
- [FOW 2000] FOWLER, M. **Uml distilled**: a brief guide to the standard object modeling language. 2.ed. [S.l.]: Addison Wesley Longman, 2000. (Object Technology).
- [FOW 97] FOWLER, M.; SCOTT, K. **Uml distilled**: applying the standard object modeling language. [S.l.]: Addison Wesley, 1997.
- [GRA 2000] GRAVENA, J. P. **Aspectos importatnes de uma metodologia para desenvolvimento de software com objetos distribuídos**. Maringá-PR: [s.n.], 2000. Monografia final de curso.
- [GRO 2002] GROSSO, A. Why the digital millennium copyright act is a failure of reason. **Communications of the ACM**, v.45, n.2, p.19–23, February 2002.
- [HAH 99] HAHN, J.; KIM, J. Why are some diagrams easier to work with? effects of diagrammatic representation on the cognitive integration process of system analysis and design. **ACM Transations on Computer-Human Interaction**, v.6, n.3, p.181–213, September 1999.

- [HAR 87] HAREL, D. A visual formalism for complex systems. **Science of Computer Programming**, v.8, p.231–274, 1987.
- [HUZ 95] HUZITA, E. H. M. **Uma metodologia para auxiliar o desenvolvimento de aplicações para processamento paralelo**. São Paulo-SP: [s.n.], 1995. Universidade de São Paulo, Tese de doutorado.
- [HUZ 99] HUZITA, E. H. M. **Uma metodologia de desenvolvimento baseado em objetos distribuídos inteligentes**. Projeto de Pesquisa.
- [JAC 99] JACOBSON, I.; BOOCH, G.; RUMBAUGH, J. **The unified software development process**. 2.ed. Massachusetts: Addison Wesley Longman, 1999. (Object Tecnology).
- [JAC 92] WEGNER, P. (Ed.). **Object-oriented software engineering: a use case driven approach**. [S.l.]: Addison-Wesley, 1992.
- [KNA 98] KNAPIK, M.; JOHNSON, J. **Developing intelligent agents for distributed system: exploring architecture, technologies and applications**. EUA: McGraw Hill, 1998.
- [KOT 95] KOTONYA, G.; SOMMERVILLE, I. **Requirements engineering with viewpoints**. Lancaster, LA1 4YR, UK: Lancaster University, Computing Departament, 1995.
- [LEI 97] LEITE, J. Enhacing a requirements baseline with scenarios. **Requirements Engineering Journal**, v.2, n.4, 1997.
- [LEI 89] LEITE, J. C. P. Viewpoint analysis: a case study. **ACM J. Software Engineering Notes**, v.14, n.3, p.111–119, 1989.
- [MAC 96] MACAULAY, L. A. **Requirements engineering**. [S.l.: s.n.], 1996.
- [MUL 79] MULLERY, G. P. A method for controlled requirements specifications. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 4., 1979. **Anais...** [S.l.: s.n.], 1979. p.126–135.
- [NAR 92] NARAYANASWAMY, K.; GOLDMAN, N. Lazy consistency: a basis for cooperative software development. In: INTERNATIONAL CONFERENCE

- CE ON COMPUTER-SUPPORTED COOPERATIVE WORK, 1992, Toronto, Ontario, Canada. **Anais...** [S.l.: s.n.], 1992. p.257–264.
- [NAU 69] NAUR, P.; RANDELL, R. **Software engineering**: a report on a conference sponsored by the nato science committee. [S.l.]: NATO, 1969.
- [O'LE 98] O'LEARY, D. E. Using ai in knowledge management: knowldege bases and ontologies. **IEEE Intelligent Systems**, v.13, n.3, p.34–39, may/june 1998.
- [PFA 2001] PFAFFENSELLER, M.; PFAFFENSELLER, M.; KROTH, E. Uma ferramenta de apoio ao gerenciamento de componentes. **Workshop de Desenvolvimento Baseado em Componentes**, Junho 2001.
- [PRE 95] PRESSMAN, R. S. **Engenharia de software**. 3.ed. [S.l.]: Makron Books do Brasil, 1995.
- [RAB 2000] RABARIJAONA, A. et al. Building and searching an xml-based corporate memory. **IEEE Intelligent Systems**, p.56–63, May/June 2000.
- [RID 2000] RIDAO, M.; DOORN, J.; PRADO LEITE, J. C. S. do. Uso de patrones en la construcción de escenarios. In: WER 2000, 2000. **Anais...** WER, 2000. p.140–157.
- [SAT 94] SATO, L. M. et al. Onix: an environment for the development of parallel object oriented software. In: INTERNACIONAL WORKSHOP ON HIGH PERFORMANCE COMPUTING, 1994, São Paulo. **Anais...** [S.l.: s.n.], 1994. p.167–183.
- [SCH 95] SCHELEBBE, H. **Distributed prosoft**. Germany: University of Stuttgart, 1995.
- [SCH 77] SCHOMAN, K.; ROSS, D. T. Structured analysis for requirements definition. **IEEE Transactions on Software Engineering**, v.3, n.1, p.6–15, 1977.
- [SCH 88] SCHWANKE, R. W.; KAISER, G. E. Living with inconsistency in large systems. In: ., 1988, Grassau, Germany. **Anais...** B. G. Teubner: Stuttgart, 1988. p.98–118.

- [SCH 2000] SCHWARTZ, D. G.; TE'ENI, D. Tying knowledge to action with kmail. **IEEE Intelligent Systems**, p.33–39, May/June 2000.
- [SOM 96] SOMMERVILLE, I. **Software engineering**. 5.ed. Massachussets: Addison Wesley, 1996. (International Computer Science).
- [SOM 97] SOMMERVILLE, I.; SAWYER, P. Viewpoints: principles, problems and a practical approach to requirements engineering. ., 1997.
- [SOU 98] SOUZA, C. T.; OLIVEIRA, M. Ábaco. Um ambiente de desenvolvimento baseado em objetos distribuídos configuráveis. In: XII SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, 1998, Maringá-PR. **Anais...** [S.l.: s.n.], 1998. p.205–220.
- [STD 2000] ST-DENIS, G.; SCHAUER, R.; KELLER, R. K. Selecting a model interchange format. In: HAWAII INTERNATIONAL CONFERENCE ON SYSTEM SCIENCES, 2000. **Anais...** [S.l.: s.n.], 2000.
- [SZY 2000] SZYKMAN, S. et al. Design repositories: engineering design's new knowledge base. **IEEE Intelligent Systems**, p.48–55, May/June 2000.
- [TOR 2000] TORO, A. D. et al. Identificación de patrones de reutilización de requisitos de sistemas de información. In: WER2000, 2000. **Anais...** [S.l.: s.n.], 2000.
- [WOO 99] WOOLDRIDGE, M. J.; JENNINGS, N. R. Software engineering with agents: pitfalls and pratfalls. **IEEE Internet Computing**, v.3, n.3, p.20–27, may-jun 1999.
- [YAN 97] YANAGI, E. Relatório de Iniciação Científica.
- [ZAN 2000] ZANLORENCI, E. P.; BURNETT, R. C. Reqav: modelo para descrição, qualificação, análise e validação de requisitos. In: IDEAS2000: JORNADA IBERO AMERICANA DE INGENIERIA DE REQUISITOS Y AMBIENTES DE SOFTWARE, 2000. **Anais...** [S.l.: s.n.], 2000. p.61–72.