UNIVERSIDADE ESTADUAL DE MARINGÁ
PROGRAMA DE INICIAÇÃO CIENTÍFICA - PIC
DEPARTAMENTO DE INFORMÁTICA
ORIENTADORA: Prof^{a}. Dr^{a}. Elisa Hatsue Moriya Huzita
ACADÊMICO: Marco Aurélio Graciotto Silva

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

Maringá - PR, março de 2002.

UNIVERSIDADE ESTADUAL DE MARINGÁ
PROGRAMA DE INICIAÇÃO CIENTÍFICA - PIC
DEPARTAMENTO DE INFORMÁTICA
ORIENTADORA: Prof^{a}. Dr^{a}. Elisa Hatsue Moriya Huzita
ACADÊMICO: Marco Aurélio Graciotto Silva

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

Relatório final de projeto
de iniciação científica

Maringá - PR, março de 2002.

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 adequada. A criação de ferramentas que dêem suporte a tais metodologias é desejável. O objetivo deste projeto 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 (HUZITA, 1999). 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 possibilitam 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 ORBacus como middleware CORBA e o banco de dados PostgreSQL como mecanismo de persistência.

Sumário

Lista de Figuras

Lista de Tabelas

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 a necessidade de adotar novas tecnologias e condutas no processo automatizado para o desenvolvimento deste tipo de software. Em (HUZITA, 1995), 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 outros ambientes de desenvolvimento de software: ONIX (SATO, 1994), ABACO (SOUZA; OLIVEIRA, 1998), PROSOFT Distribuído (SCHELEBBE, 1995), 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 baseado em objetos distribuídos inteligente (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 reuso e de cenário, técnicas para negociação de requisito. Também foram abordados aspectos quanto ao armazenamento das informações geradas durante o processo de engenharia, através da análise dos repositórios kMail, WebCADET, NIST Repository Project e Osirix, assim como métodos de intercâmbio de informações entre repositórios e ferramentas, em especial a XMI. Todos estes estudos culminam numa ferramenta que permite a criação de diagramas de caso de uso, identificando visualmente características de distribuição e paralelismo , através de uma notação extendida da UML.

2 Objetivos

O objetivo deste projeto é desenvolver um protótipo de uma ferramenta para apoiar a definição de requisitos para projeto de software distribuído. Os objetivos específicos são:

- Estudo de processo de desenvolvimento de software;
- Estudo da MDSODI - Metodologia de desenvolvimento baseada em objetos distribuídos inteligentes;
- Definição da arquitetura do protótipo;
- Especificação da interface do protótipo da ferramenta;
- Implementação de um protótipo da ferramenta para dar suporte à especificação de requisitos;
- Avaliação do protótipo utilizando-o em estudos de caso.

3 Materiais e métodos

3.1 Processo de Desenvolvimento de Software

3.2 Metodologia de Desenvolvimento Baseada em Objetos Distribuídos Inteligentes

3.3 Repositórios

Repositórios são aplicações que possibilitam o armazenamento de dados utilizados e gerados no processo de engenharia. Eles permitem um acesso transparente às informações nele armazenadas, possuem mecanismos de controle de versão, aplicam controle de acesso aos dados. Além disto, os repositórios geralmente armazenam algum tipo de metadado que possibilita a extração de conhecimento do conteúdo neles armazenados. Neste trabalho foram estudados alguns repositórios: kMail, WebCADET, o desenvolvido pelo projeto NIST Design Repository e o Osirix. Cada um deles tem enfoque diferente (CAD, gerenciamento de artefatos, etc) e soluções específicas em relação a representação dos dados e aos mecanismos de gerenciamento, extração e inferência de conhecimento.

3.3.1 kMail (SCHWARTZ; TE'ENI, 2000)

As empresas, ao longo de sua existência, acumulam muito conhecimento sobre sua área: soluções sobre determinados problemas, idéias, problemas encontrados. Uma prática importante seria disponibilizar todo este conhecimento para uso interno, podendo reagir mais rapidamente às situações futuras. Muitas empresas armazenam estas informações mas não conseguem utilizá-las de maneira adequada, provavelmente devido a falhas em alguma das atividades de gerenciamento desta base: aquisição de novos dados, a organização destes ou sua distribuição. Em pesquisa realizada por Daniel O'Leary(O'LEARY, 1998), são apontadas algumas razões desta subutilização:

- As atividades realizadas nas bases de conhecimento devem ser orientadas a ações;
- As bases de conhecimentos devem possuir mecanismos eficientes para facilitar a incorporação de novos dados, geração de conhecimento e sua consequente atualização;
- O conhecimento não deve substituir a criatividade e sim estimulá-la e guiá-la.

O *kMail* é uma ferramenta (dividida em duas partes: uma cliente e outra servidor) que torna disponível o conhecimento organizacional de acordo com as ações das pessoas, usando para isto uma base de conhecimento acessível pela Internet (através de URLs), uma base de metaconhecimento e a ferramenta que, utilizando-se destas bases, fornece a informação mais adequada de acordo com as ações em execução, resolvendo os pontos criticados por O'Leary quanto a subutilização das bases de conhecimentos das empresas.

Um aspecto importante do *kMail* é que as ações consistem em trocas de emails. A escolha deste meio de comunicação deve-se ao fato de que, muito provavelmente, este serviço já esteja implementado na empresa, o que torna a curva de aprendizado para este novo sistema muito mais suave. Pesquisas também provam que o email é utilizado, na maioria dos casos, para

requisitar ou responder algo, ou seja, exatamento no momento em que os dados que temos na base de conhecimento organizacional fazem-se necessários.

Além do serviço de email, o *kMail* utiliza-se de bases de conhecimento, previamente existente na empresa e acessível pela Internet (identificados por URIs), e um sistema de metaconhecimento. Este último armazena os dados dos emails enviados (remetente, destinatário, assunto, data de envio) e relaciona-os com os perfis dos usuários (cargo, experiência, etc), determinando o contexto situacional que vai possibilitar a criação de visões da base de conhecimento. Destas escolhe-se a mais relevante para ser utilizada.

Detalhando mais o funcionamento do *kMail*, cita-se um pseudo-exemplo de como seria a utilização do sistema:

1. Envia-se um email para o setor de marketing da empresa a respeito de um novo produto: a calça jeans vermelha XYZ. Para criar este email, utiliza-se um cliente *kMail*.

2. O cliente *kMail* faz uma análise do email, identificando atributos importantes do email necessários à criação do contexto: rementente, destinatário. Em seguida, envia-se este email para o servidor kMail.

3. O servidor *kMail*, com os dados recém-recebidos, os perfis de usuários e o meta-conhecimento já existente sobre a base de dados, cria várias visões (pessoal, supervisional, relacionada ao projeto, relacionada ao cargo).

4. Esses dados são repassados ao cliente *kMail*. Agora, o autor do email deve inserir links que considere relevantes, valida os já existentes e, enfim, confirma o repasse do email para os destinatários.

5. Os destinatários recebem o email em programa de email comum (que aceite trabalhar com emails no formato HTML). Esses emails vem com os links selecionados pelo autor do email, a visão que ele selecionou.

Com isto, consegue-se tirar proveito da base de conhecimento de maneira transparente e rápida. A validação dos links (pelo autor) e seus subsequentes acessos (pelos destinatários) permitem determinar a importância dos dados da base, possibilitando a obtenção de métricas para melhorar a organização da base, a criação de visões e o mecanismo de meta-conhecimento, fazendo-o considerar estes diversos graus de utilização dos dados.

3.3.2 WebCADET (CALDWELL, 2000)

O *WebCADET* consiste numa ferramenta para suporte a decisão baseada na Web, permitindo assim seu uso por várias pessoas em diferentes locais do mundo, ou seja, de maneira distribuída. Adota-se o paradigma de "IA como texto", trabalhando com conhecimento na forma de texto, legível por seres humanos (obedecendo regras de gramática e vocabulário adequados para isso), garantindo assim uma maior transparência aos usuários do sistema.

A representação do conhecimento empregada pelo *WebCADET* é estruturada de acordo com o grau de detalhamento, começando no nível mais baixo. Por exemplo, o sistema começa com a opção de escolher o setor ao qual o produto a ser projetado melhor se adequa, posteriormente permitindo a escolha do tipo de produto e as características específicas do mesmo, o penútimo

nível da hierarquia do sistema. Aos dados determinados até então são aplicadas então regras que temos sobre os produtos, o último nível.

As regras, que são a base fundamental do suporte à decisão, juntamente com o mecanismo de inferência, são estruturadas da seguinte forma:

- **rule_id**: Identificador único da regra em todo o sistema.
- **name**: Nome da regra.
- **preconditions**: Pré-condições que devem ser atendidas.
- **conditions**: Permite qualificar os atributos do produto sendo proposto. Basicamente são regras condicionais como, por exemplo, "if button_number_size gt 3 then 3 else 0".
- **scale**: Fator normalizador.
- **history**: Histórico da regra: quando foi criada e por quem, quando foi alterada, o que foi alterado, etc.
- **keywords**: Palavras chaves para ajudar a identificar o contexto da regra.

O sistema tem três modos de uso: avaliação de projetos, consulta à base de conhecimento e adição de novos conhecimentos. Em avaliação de projetos, o projetista detalha os dados de seu projeto e o sistema, aplicando as regras cabíveis, verifica quão bem sucedido seria o produto (atribuindo-lhe uma nota). Pode-se também verificar o porquê de cada nota, com o sistema retornando um texto com base nas regras (figura 1) aplicadas (aqui temos o emprego da "IA como texto"). No modo de operação de consulta, pode-se percorrer a base de conhecimento, atividade útil para verificar os dados que os sistema armazena. Por fim, tem-se o modo de inserção de conhecimento, no qual novos produtos podem ser definidos e novas regras podem ser adicionadas a base de conhecimento do sistema.

```
rule_id phone_easy_to_dial_1
name easy_to_dial
precondition  product_type phone
conditions
[
if button_shape iaiof easy_dial_button_shape then 4 else 0.
 if button_length gt 5 and button_length lt 18 then 5 else 0.
 if button_width gt 5 and button_width lt 18 then 5 else 0.
 if button_configuration iaiof easy_dial_button_configuration then 3 else 0.
 if button_spacing gt 6 and button_spacing lt 20 then 5 else 0.
 if button_material has_aspect easy_dial_button_material then 2 else 0.
 if button_number_size gt 3 and button_number_size lt 16 then 3 else0.
 ]
scale  27
history
[amendent(author(`Paul Rodgers`,`EDC, Department of Engineering,
Cambridge University, UK`,
`pr2@eng.cam.ac.uk`),creationdate(30,`April,`1993),
`This attribute was determined to be relevant for this type of product as
 the result of field work undertaken by the author`)]
keywords [`generic`].
```

Figura 1: Exemplo de regra aplicável a um telefone, definindo condições para avaliar o teclado de um telefone.

Algumas críticas foram levantas a respeito do *WebCadet* em testes de usabilidade realizado pelos idealizadores da ferramenta: impossibilidade de salvar sessões no sistema (o que obrigaria a criação de produtos em questões de minutos, sem interrupções), navegabilidade (um tanto quanto confusa). A natureza do sistema, acessado por meio de um navegador, ao mesmo tempo que permite acesso distribuído, também é a principal responsável pelos problemas encontrados nos testes.

3.3.3 Repositório de Projetos do NIST (SZYKMAN, 2000)

O Repositório de Projetos NIST objetiva a criação de depósito de projetos. Destinado principalmente às indústrias e aos artefatos por elas produzidos, ele se destaca por descrever não somente a forma mas também a função e o comportamento dos artefatos. Outro ponto importante é a utilização de padrões já existentes (STEP para a descrição geométrica dos objetos) e a definição de novas especificações como no caso da descrição de funções, fluxos e comportamentos. A criação de taxonomias de funções e fluxos, facilitando a procura por dados na repositório, também é de suma importância.

A representação dos artefatos do depósito é feita em torno de três parâmetros: forma, função e comportamento. A forma é definida em STEP (*Standard for the Exchange of Product Model Data*), podendo ser visualizada também utilizando-se VRML (*Virtual Reality Modeling Language*). As funções e fluxos são especificados utilizando uma linguagem específica, seguindo uma estrutura pré-definida. Uma função, por exemplo, estrutura-se da seguinte maneira:

- Name: Nome da função, único na base.
- Type: Classe a qual pertence a função.
- Documentation: Texto, em linguagem natural, descrevendo a função.
- Methods: Descrição da função porém em uma linguagem mais apropriada para processamento automático por software.
- Input_flow: Fluxos de entrada da função.
- Output_flow: Fluxos de saída da função.
- Subfunctions: Funções das quais é composta.
- Subfunction_of: Função da qual faz parte.
- Referring_artifact: Artefato a qual a função se refere.

O depósito é gerenciado através de uma interface Web, a partir da qual pode-se criar, modificar, atualizar e acessar artefatos armazenados no repositório de projetos. Novamente, similarmente ao WebCADET, são detectados problemas quanto à utilização de uma interface baseada em páginas HTML, principalmente quanto à velocidade de resposta das ações tomadas pelo usuário.

3.3.4 Osirix (RABARIJAONA, 2000)

O *Osirix* é um sistema gerenciador de conhecimento que, utilizando documentos XML validados com DTDs, possibilita a procura por informações de maneira ágil e eficaz. A criação das DTDs e inserção de novos documentos XML no repositório pode ser assim descrito:

1. O primeiro passo para a construção do depósito é a definição da ontologia para uma memória corporacional (base de conhecimento da empresa). Isto é feito com a linguagem CML (*CommonKADS Conceptual Modeling Language*).

2. Define-se então um modelo padrão para os documentos XML que serão armazenados no repositório, criando a *core DTD*.

3. O *Osirix* transforma a ontologia definida em CML em um DTD e integranda-a com a *core DTD*.

4. Pessoas alimentam o *Osirix* com documentos devidamente formatados e válidos de acordo com a DTD presente no repositório.

5. O repositório valida este documento XML. Caso esteja correto, armazena-o.

O mecanismo de busca do repositório adota um modelo híbrido, fazendo primeiramente uma busca tradicional e, depois, uma busca utilizando-se da ontologia dos documentos encontrados. O primeiro método de busca usa mecanismos semelhantes aos do Google[1], Altavista[2] e afins, sendo extremamente rápido porém muito limitado, suportando apenas restrições do tipo AND, OR, NOT. A segunda busca é feita observando os atributos dos documentos XML encontrados e as palavras chaves da buscas. Este segundo método é o mais importante de todos, por isso cabe a ele um maior detalhamento:

1. Identificam-se os elementos XML dos documentos XML encontrados na busca anterior.

2. Procura-se a presença semântica de uma palavra chave (conceito ou propriedade) requerida na pesquisa. Esta presença semântica significa a presença de uma palavra chave como um *tag* na descrição ontológica do documento.

3. Verifica-se se a tag encontrada possui o valor requisitado na pesquisa. Caso positivo, cria-se uma página HTML a partir do documento XML encontrado (utilizando para isso XSL) e a envia como resposta da pesquisa.

A utilização de XML se destaca no Osirix. Muitas ferramentas suportam documentos produzidos nesta linguagem, várias novas tecnologias se utilizam dela, tornando assim seu uso muito interessante. Também é feito usufruto do caráter semântico do XML, tornando as pesquisas muito mais eficientes.

[1] http://www.google.com
[2] http://www.altavista.com

3.4 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 deriva-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, heterogeniedade 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 (BESSANI, 2000), o que encorajou sua utilização neste projeto.

O Common Object Request Broker Arquitecture (CORBA) foi criado pela Object Management Group[3] (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 disbribuído
- 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 2 demonstra toda 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 3 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.

[3] http://www.omg.org

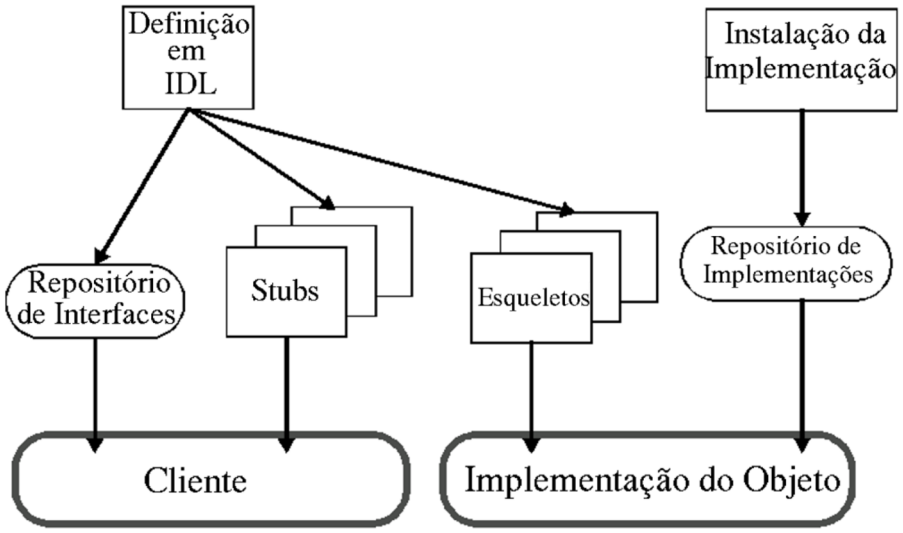


Figura 2: Mecanismos para requisição e acionamento de objetos

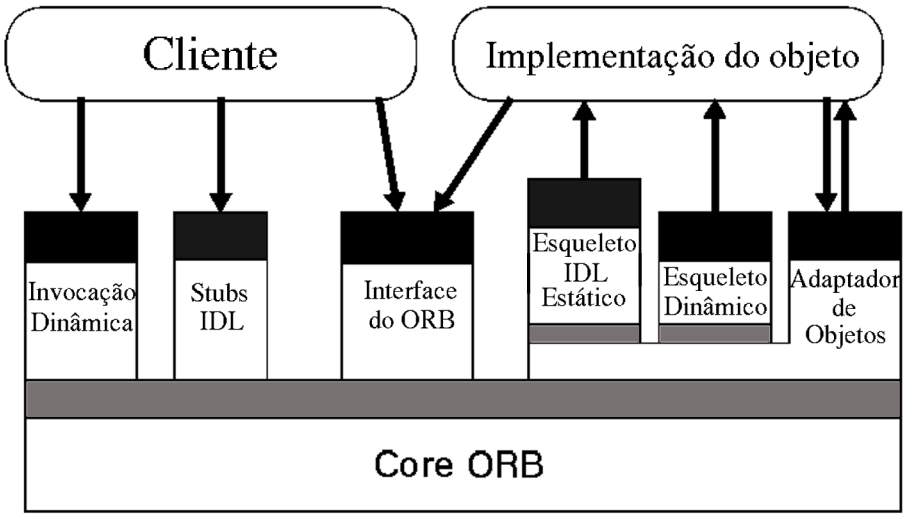


Figura 3: Interfaces do Object Request Broker

3.5 Engenharia de requisito

3.5.1 Uma breve 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 de software.

A importância da engenharia de requisitos é facilmente detectada a partir do momento que tudo no processo de engenharia de software depende dela. 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. Em resumo,

geralmente tem-se custos além dos projetados e atrasos na entrega do produto, muitas vezes com qualidade prejudicada. Todos estes problemas são facilmente evitáveis com um bom processo de engenharia de requisitos, aplicado corretamente.

Um processo de engenharia de requisitos, segundo Linda (MACAULAY, 1996), 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, factibilidade.
- Produção de um documento de requisitos.

A primeira fase, a delimitação do escopo da aplicação, seu domínio, é a mais simples e ao mesmo tempo a mais complexa. O problema é que o usuário nunca sabe o que quer exatamente até ver alguma coisa concreta. Logo, nas primeiras iterações, os stakeholders tem uma idéia e, com o decorrer do desenvolvimento do sistema, esta explode em tamanho e complexidade com muita facilidade, na maioria das vezes fugindo do domínio especificado inicialmente.

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 incluir-se 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 *deadline*. Tendo este estabelecido, cria-se o documento de requisitos que servirá de base para o resto do processo de engenharia de software.

3.5.2 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
- Domínio da aplicação mal definido

Destes, o de mais difícil solução é a solução de inconsistências. Mas antes, vamos definir o que ela é exatamente: "Inconsistência é uma situação em que duas partes de uma especificação não obedecem algum relacionamento mantido entre elas" (EASTERBROOK; NUSEIBEH, 1995). 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 existentes na especificação, 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 contribuem ainda mais para

aumentar a complexidade do problema, ocasionando a criação de inconsistências nos mais diversos momentos do processo de desenvolvimento, reativação de antigas inconsistências devido a invalidação de antigas soluções. Somando a isto o desenvolvimento distribuído, a resolução destes problemas fica ainda mais difícil. Portanto, técnicas foram desenvolvidas para tratar as inconsistências, abordando sua descoberta e gerenciamento.

Técnicas para descoberta de inconsistências 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. Existem várias possibilidades, variando muito de acordo com o método de desenvolvimento escolhido, o tipo de sistema, etc.

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, permite uma maior liberdade para projetar o sistema, evitando decisões prematuras, 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 mecanismos 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 (BALZER, 1991), CONMAN (SCHWANKE; KAISER, 1988) (ferramenta de gerenciamento de configuração). Uma outra abordagem é a lazy consistency, proposta por Narayananwamy e Goldman (NARAYANASWAMY; GOLDMAN, 1992). 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 (BESNARD; HUNTER, 1995), 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) geralmente consiste nas seguintes atividades:

- Detecção: Procura por informações da especificação que quebra 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: Melhorar a situação da inconsistência mas não necessariamente resolvê-la.
  - Contornar: Não considera como uma inconsistência, ela provavelmente é uma exceção.

3.6 Técnicas aplicáveis no processo de engenharia de requisitos

3.6.1 Uso de padrões na construção de cenários (RIDAO; DOORN; PRADO LEITE, 2000)

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 deste 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.

De todos os aspectos acima, o mais importante, representativo, é o do 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 autonôma, 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 episódos 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.

Além disto, as situações são compostas de diferentes tipos de episódios, ou seja, novos tipos de cenários:

- 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 (LEITE, 1997): 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.

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. Consequentemente, 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 tarefe torna-se muito fácil.

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

Actores: 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: Descrição do padrão Negociação Terminada com Produção.

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 do-

mínio 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 a lista de cenários canditados.
3. Através de um sistema especialista (figura 3.6.1), tenta-se extrair o máximo possível de informações sobre os cenários candidatos.

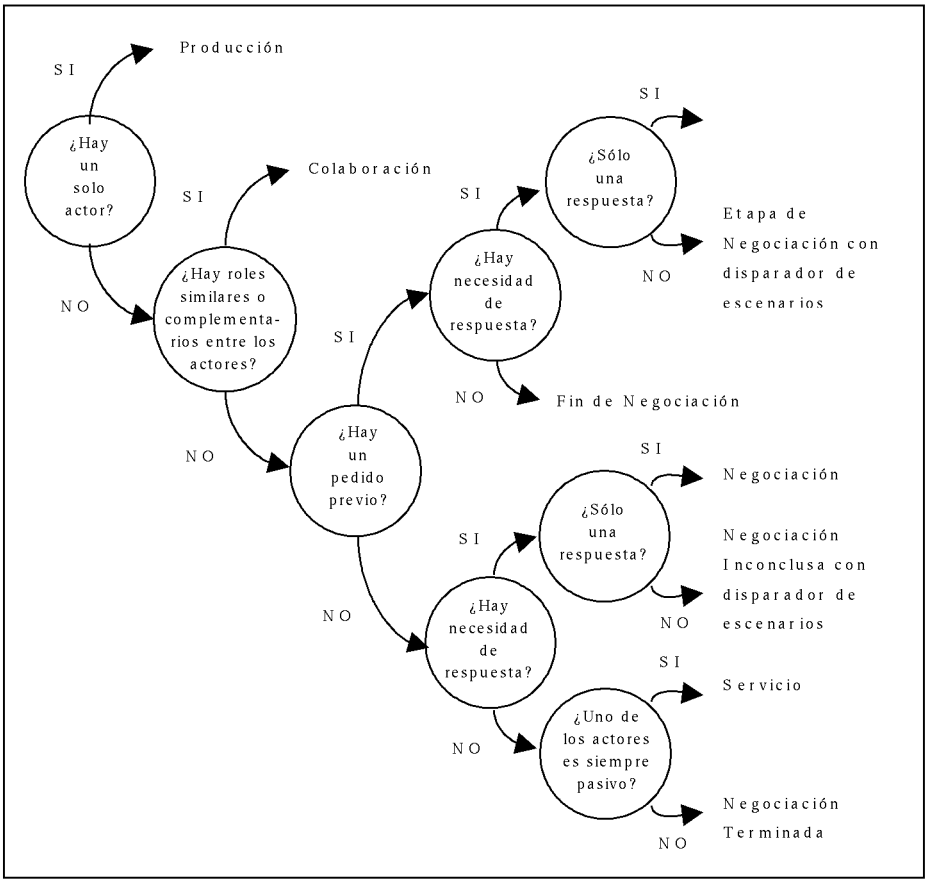


Figura 5: Exemplo de arvore de decisão para a seleção de padrões (retirado do artigo estudado).

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, agilizando o processo mais rápido e garantindo uma melhor qualidade.

3.6.2 Identificação de Padrões de Reutilização de Requisitos de Sistemas de Informação (TORO, 2000)

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

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 no 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 tipo 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 dados do artigo estudado, o que ocorre com maior frequência é o primeiro, cliente-sócio (mais de 90% dos casos). Depois temos produto/artigo com 60% e assim por diante. Tão interessante quanto isto é que estes padrões-R 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, temos 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_c 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.

3.6.3 Pontos de vista

A engenharia de requisitos orientada a ponto de vistas vêm 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 (SCHOMAN; ROSS, 1977), houve este reconhecimento. Porém, a utilização disto nunca ocorreu na proporção em que deveria, visto sua abrangência. Um exemplo isolado seria o CORE (MULLERY, 1979), 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 (FINKELSTEIN; KRAMER; GOEDICKE, 1990; LEITE, 1989; FICKAS; LAMSWEERDE; DARDENNE, 1991). A origem de modelos diferentes surge

das caracteristicas intrinsecas dos projetos para o qual o método foi criado, tomando definições de ponto de vistas que facilitassem o desenvolvimento dos sistemas. Por exemplo, em (SOMMERVILLE, 1996) são descritos os seguintes tipo de pontos de vistas:

- 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 a natureza deste projeto, 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 é a mais apropriada.

3.6.4 Viewpoint framework (EASTERBROOK; NUSEIBEH, 1995)

Este framework (não nomeado) 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 a "lazy consistency" (NARAYANASWAMY; GOLDMAN, 1992).

Os Viewpoints, neste framework, são definidos como objetos distribuídos fracamente acoplados, localmente gerenciados, que encapsulam conhecimento parcial sobre um sistema e seu domínio (devidamente especificado com uma notação particular e adequada) e conhecimento parcial do processo de desenvolvimento. Ele é composto por:

- Estilo: Maneira pela qual o ponto de vista expressa seu conteúdo.
- 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 para gerar sua aplicação.
- Registro de trabalho: Histórico das ações efetuadas no viewpoint.
- Proprietário: Pessoa que criou o viewpoint.
- Lista de inconsistências.

Os viewpoints são criados a partir de um modelo (template). Este contém a mesma estrutura de um viewpoint, mas só possui definido seu estilo e seu plano de trabalho. Quando criado um

viewpoint, escolhe-se um modelo para servir de base. A partir deste momento, o viewpoint não depende mais do modelo.

O plano de trabalho guia todo o processo de criação e manipulação de um ponto de vista. Ele especifica a notação, como será feito o particionamento, regras de consistência, técnicas de análise entre os viewpoints. No método proposto neste artigo, é utilizado 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.
- Técnica de particionamento, permitindo a criação de diagramas que representam um subconjunto do comportamento de um dispositivo.
- Conjunto de regras de consistência que testa se os diagramas particionados de um mesmo dispositivo são consistentes um com o outro.
- 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.

Em resumo, o processo seria composto destes passos:

1. Cria-se um viewpoint a partir de um modelo pré-definido no sistema.
2. Particiona-se a especificação, tomando alguns viewpoints.
3. Aplicam-se as regras de consistência ao particionamento criado.
4. Aplicam-se as regras de consistência em cada viewpoint. O resultado fica gravado no "Registro de Trabalho".
5. Se alguma regra for quebrada, registra-se na "Lista de Inconsistências" as inconsistências. Uma lista de ações é mostrada, da qual escolhe-se uma 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.
6. Aplica-se novamente a regra no viewpoint, verifica-se o resultado e tenta-se corrigir o problema novamente se for necessário.

Interessante notar que uma ação, apesar do efeito influenciar o outro ponto de vista da partição, não implica na mudança deste último. Por exemplo, uma ação é tomada no ponto de vista V1, que possui uma inconsistência com 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 uma vez não significa que não existem inconsistências por toda a vida do viewpoint. A partir do momento em que um viewpoint é alterado e suas alterações influenciam os outros, novas inconsistências podem surgir, e estas só serão realmente detectadas com a aplicação das regras no viewpoint.

3.6.5 Viewpoints Oriented Requirements Definition (KOTONYA; SOMMERVILLE, 1995)

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 sistemas.

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 6, 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 o serviços fornecidos pelos sitema, abordando geralmente algum aspecto comum a todos este, mas, porém, sem interagir com os mesmos.

O método, no artigo estudado, 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 7), podemos visualizar mais facilmente os processos e suas interações. Primeiramente são identificados os pontos de vistas 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-o 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).

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:

- Com base nos pontos de vista abstratos pré-definidos (figura 8), elimine aqueles que não são relevantes ao aspecto do sistema sendo especificado.
- Tenha certeza de que os stakeholders estejam sempre representados em algum destes pontos de vista.
- Usando modelos de uma arquitetura de sistema (provavelmente já existente na maioria dos casos), identifique outros pontos de vistas relevantes.
- Identique operadores do sistema que utilizam o sistema constantemente, ocasionalmente ou que mandam os outros fazerem as coisas para eles. Estes são potenciais pontos de vista.

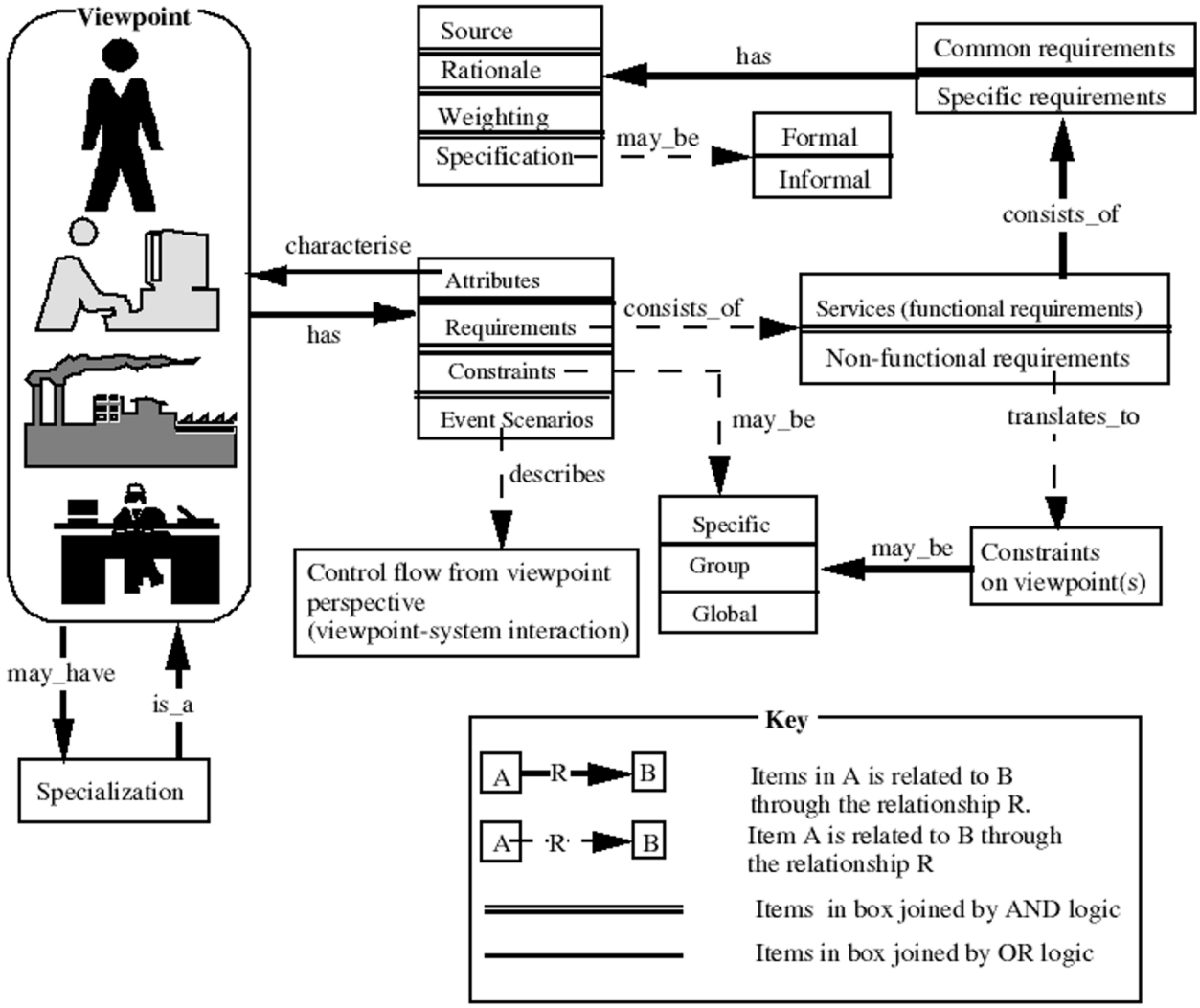


Figura 6: Pontos de vista e estrutura das informações

- Identifique os papéis de pessoas que estejam associadas a classes de pontos de vistas indiretos. Geralmente tem-se pontos de vistas associados com cada papel.

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 em nível de sistema. Esta distinção permite:

- Rastrear requisitos de controle a partir da perspectiva do usuário.

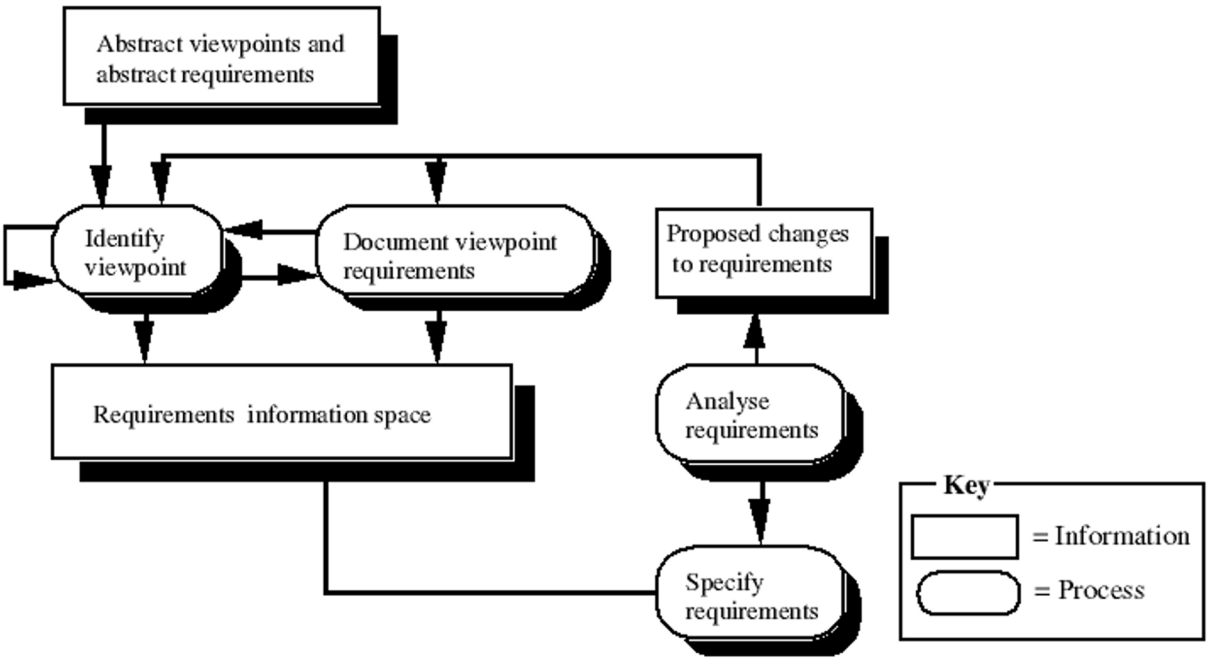


Figura 7: Modelo do processo VORD

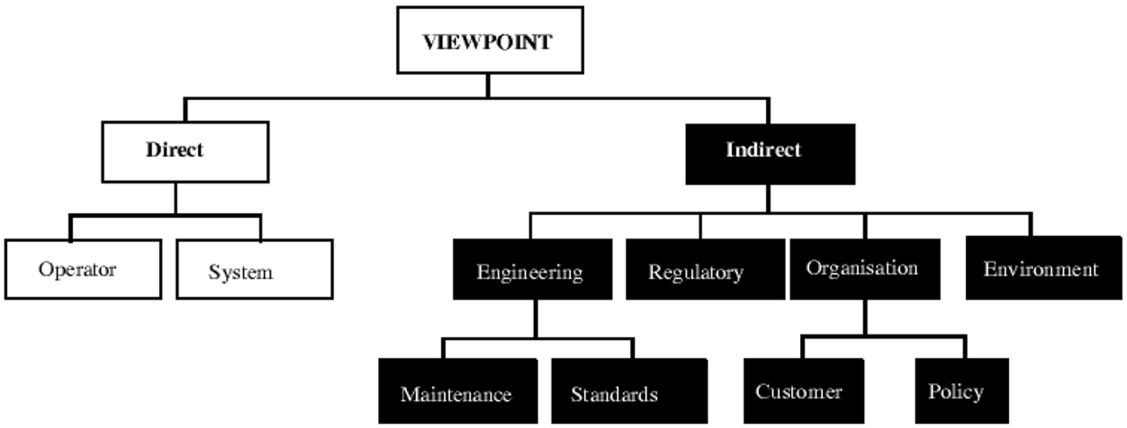


Figura 8: Classes de ponto de vista do VORD

- 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.

3.6.6 Process and Requirements Engineering Viewpoints - Preview (SOMMERVILLE; SAWYER, 1997)

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.
- 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 exeplo: 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, se sobrepor, 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 identifiar 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*: Realizada através de requisitos externos e questionários.
   (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.

3.6.7 REQAV: Modelo para Descrição, Qualificação, Análise e Validação de Requisitos (ZANLORENCI; BURNETT, 2000)

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 descritivo 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 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.

3.6.8 Usando diferentes meios de comunicação na negociação de requisitos (DAMIAN, 2000)

Há tempos percebe-se a necessidade de aproximação dos clientes e desenvolvedores para definir os requisitos, principalmente para resolver os conflitos encontrados. Sempre 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. No artigo estudado, 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) com a forma de comunicação que até então se acreditava ser mais efetiva.

Aborda-se o efeito da comunicação no desempenho do grupo na negociação de requisitos e os efeitos da configuração do grupo. Tomou-se como variáveis independentes o modo de comunicação e o arranjo do grupo. As variáveis dependentes foram o desempenho do grupo e percepção pessoal. Destas variáveis, a mais importante foi 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 no artigo consistiu na negociação de requisitos funcionais de um sistema de gerenciamento bancário. Estudou-se cinco configurações de grupo: uma cara a cara e outras quatro distribuídas. Abaixo temos as configurações utilizadas:

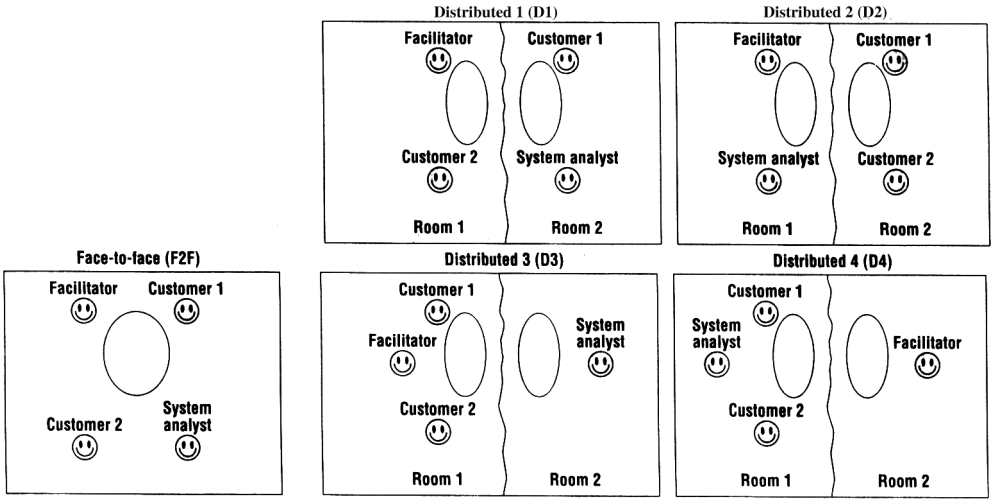


Figura 9: Cinco configurações de grupo: (a) cara a cara e (b) distribuída.

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, conseqüencia da persuação que um cliente exerce sobre o outro, do proximididade 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 reduzidade capacidade de perceber as emoções das pessoas, como no caso D1, permitiram aos clientes um melhor entendimento das necessidades, permitindo um raciocínio mais claro, além de possibilitar que o analista de sistema se mantenha mais imparcial.

3.7 Formatos para intercâmbio de modelos

Uma das maiores barreiras no desenvolvimento de software, principalmente agora com a participação de grupos geograficamente distantes, é o intercâmbio de dados entre as diferentes ferramentas utilizadas. Uma solução simples seria forçar a todos a utilização de uma mesma ferramenta. Claro que isto é impossível na maioria dos casos, além de ser pouco eficiente. O melhor seria a criação de um formato padrão para realizar esta troca de informações. Em (ST-DENIS; SCHAUER; KELLER, 2000), vários requisitos foram definidos quanto a tal padrão:

- Transparência: o processo de codificação/decodificação especificado pelo formato de intercâmbio de modelos não deve remover, adicionar ou alterar qualquer informação contida no modelo original.
- Independência de escala: deve ser adequado a projetos reais, de grande porte. Algumas características que devem ser observadas são a compressibilidade, a possibilidade de fazer o intercâmbio de modelos parciais (somente as diferenças entre dois modelos, por exemplo), estabelecimento de ligações (referências) entre os modelos (ao invés de duplicar os dados).
- Simplicidade: talvez o mais óbvio e, ao mesmo tempo, o mais difícil. O formato deve atacar a raiz do problema, o resolvendo de maneira eficiente (utilizando um mínimo de recurso computacional e humano). A simplicidade contribui também para uma menor complexidade das ferramentas necessárias para a manipulação dos dados gerados, além de reduzir a chance de ter erros no padrão.
- Neutralidade: garante que o padrão acomoda (ou simplesmente ignora) aspectos específicos da plataforma na qual está sendo utilizado. Por exemplo: linguagem, extensões da linguagem sendo transportada.
- Formalidade: A especificação deve estar definida formalmente, eliminando assim interpretações diferentes (e muitas vezes conflitantes) do padrão. Isto é vital para a construção de ferramentas que automatizem a aplicação do formato de intercâmbio de modelos.
- Flexibilidade: capacidade de acomodar os mais diferentes tipos de modelos, sejam estes completos ou incompletos.
- Capacidade de evolução: o formato deve ser capaz de atender futuros requisitos.
- Popularidade: se o formato não for aceito pela maioria, sua função principal deixa de existir.
- Completude: o formato deve ser completo o suficiente, evitando que os usuários (e ferramentas) tenham de incluir funções comumente usadas porém não diretamente suportadas pelo formato.
- Identidade com metamodelos: utilizar um metamodelo universal ou utilizar metamodelos específicos no processo.
- Reuso de padrões já existentes.

- Legibilidade: apesar do formato ser destinado a manipulação por ferramentas, é desejável que ele seja legível o suficiente para que um engenheiro de software consiga entendê-lo e modificá-lo sem a utilização das mesmas.

- Integridade

De acordo com estes requisitos, pode-se identificar que vários podem ser atendidos com a utilização de XML (Extensible Markup Language) (BOSAK; OTHERS, 1998): ela é simples, flexível, possui um bom suporte quanto a ferramentas, sua popularidade é extremamente elevada, possui boa legibilidade, reuso de padrões, neutralidade. A linguagem XML é bem recente, foi recomendada pela W3C em fevereiro de 1998, chegando a ser estranho uma linguagem tão recente ser tão popular. Mas, na verdade, os alicerces do XML são extremamente sólidos: trata-se de um subconjunto da SGML (Standard Generalized Markup Language, ISO 8879:1986); todo o seu processo de criação foi acompanhado por empresas tais como IBM, Microsoft, todo o movimento opensource, sendo definida por uma organização sem fins lucrativos (W3C).

No entanto, somente a utilização da XML não é o suficiente. São necessários mecanismos para suportar diferentes metamodelos de maneira fácil. A primeira solução imaginada, a utilização direta de metamodelos em DTDs ou XML Schema, era pouco flexível. A OMG decidiu criar, então, o XMI.

Ele define regras para transformar metamodelos definidos em MOF em DTDs e, futuramente, XML Schemas. O MOF é a base na definição dos metamodelos utilizados pela OMG, logo todas as linguagens por ela definidas podem ser transportadas. Oficialmente, existem DTDs criadas para a UML e a MOF, mas nada impede que empresas criem suas próprias a partir de metamodelos definidos em MOF ou até mesmo UML, utilizando para isso os mecanismos do XMI.

O único problema atual desta tecnologia é que nem todas as empresas adotam o mesmo metamodelo oficial. Por exemplo, o Rational Rose 2000 utiliza um metamodelo da UML ligeiramente diferente daquele definido na especificação da UML 1.3.

Em agosto de 2000, foi realizado um teste utilizando as ferramentas ArgoUML 0.8, Rational Rose 2000 com suporte a XMI, MagicDraw UML 3.6 e Together 4.0. Utilizou-se um simples diagrama de caso de uso, composto por um ator relacionado a dois casos de uso. De todas as possibilidades testadas, a única que funcionou foi a importação pelo Together 4 de um documento XMI criado pelo Rational Rose. E mesmo assim, o sucesso foi parcial. Do ponto de vista de modelo, ambos ficaram idênticos. A visão que tem-se deste modelo (o diagrama em si), no entanto, ficou diferente.

Na verdade este resultado já era esperado. A UML consegue padronizar o modelo mas não sua visualização. Dados sobre posicionamento dos elementos do modelo e visualização de esteriótipos são dependentes da ferramenta sendo utilizada. Apesar de ser possível transportar tais dados num documento XMI, por não ser existir um padrão para representação destes dados, não há garantias que a ferramenta que lerá o documento os conseguirá interpretar corretamente. Isto não é exatamente um problema e sim uma característica: o modelo é corretamente importado, ferramentas conseguirão trabalhar com ele. O único porém é que será um pouco difícil a visualização dos diagramas pelas pessoas. Algumas ferramentas (Rational Rose, por exemplo) podem organizar automaticamente o layout de um diagrama, o que deveria resolver este problema. Infelizmente, tal função produz resultados nem sempre agradavéis, as vezes nem utilizáveis de tão confusos.

O problema de interoperabilidade apontado no teste também foi confirmado através de listas de discussão via email[4] e esforços foram realizados desde então para resolver este problema. Atualmente discute-se a criação de um repositório de DTDs e XML Schemas gerados com o XMI, além de documentos exemplos (casos de teste), permitindo assim que as empresas tenham alguma referência quanto a compatibilidade a ser alcançada.

3.8 Persistência utilizando banco de dados

A necessidade por pesquisa 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, descobriu-se que o banco de dados PostgreSQL possuia mecanismos de serialiação 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 de banco de dados, qualquer tipo de pesquisar a ser efetuada torna-se muito mais rápida e prática.

Primeiro, necessita-se de uma explicação sobre o PostgreSQL. Ele é um 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 muito bom e completo, inclusive com 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 objetos 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). Agora que é possível guardar, além de atributos como String, int, double, referências para outros objetos. Isso permitirá restaurar todos os objetos associados a um em específico, tornando mais simples a realização de persistência.

Alem destas facilidades, outra, não relacionada a persistência mas extremamente importante, é o tipo *text*. Ele permite o armazenamento de cadeias de caracteres de tamanho arbitrário e

[4]Lista sobre XMI do Distributed System Technology Centre (xmi@dstc.edu.au) e Request Task Force do XMI (xmi-rtf@emerald.omg.org)

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

3.9 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 ExtensionMechanisms 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 projeto.

No pacote Extension Mechanisms da UML, temos definidos duas novas metaclasses: Stereotype, TaggedValue. Além disso, temos a metaclasse 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 metaclasse que altera a elemento do modelo de maneira que ele pareça ser uma instância de um metamodelo virtual (virtual porque ele não é definido da 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 metaclasse 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 a este.

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 nossa linguagem. Escolheu-se pela utilização de tagged values. Para os atores, foram definidos os seguinte:

- isDistributed
- isParallel

- isExclusive

Para os casos de uso:

- isSequential
- isDistributed

E, finalmente, para os relacionamentos:

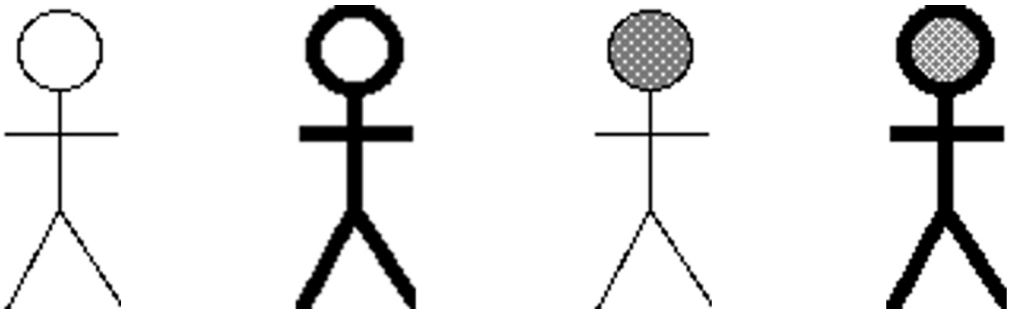
- isSequential
- isParallel

A nível de implementação, os *tagged values* são armazenados em um Hashtable, tendo como chave o nome da *tag* e o valor armazenado os dados referentes a chave. Quanto a representação gráfica destes, o procedimento que faz a pintura leva em consideração os *tagged values*, resultando em gráficos semelhantes aos representados nas figuras 10, 11 e 12.

(a) Relacionamento entre casos de uso

(b) Relacionamento entre casos de uso paralelos

Figura 10: Representação dos relacionamentos



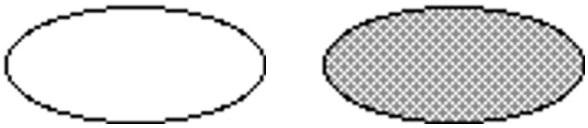
(a) Ator exclusivo

(b) Ator paralelo

(c) Ator distribuído

(d) Ator paralelo e distribuído

Figura 11: Representação dos atores



(a) Caso de uso sequencial

(b) Caso de uso distribuído

Figura 12: Representação dos casos de uso

4 Resultados e Discussão

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

Como pode ser notado, foi dada uma ênfase em abordagens que evitassem um alto grau de formalismo. Não que sua aplicação não seja válida, esta escolha se deve, principalmente, ao fato de sistemas distribuídos serem, geralmente, de grande porte, o que dificultaria sua definição de maneira exata e sem ambiguidade em todos os aspectos, o que torna difícil o emprego de tais técnicas.

Implementar uma ferramenta que automatize a definição de requisitos, de acordo com a metologia 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 muitos 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 projeto, 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 no artigo estudado na seção 3.6.1. O ideal seria a identificação automática de padrões.

Temos na tabela abaixo uma breve comparação das diferentes técnicas analisadas. Os dados para tal comparação vêm dos problemas apontados nos próprios artigos estudados e do entendimento do autor deste relatório quanto a cada técnica estudada. A não disponibilidade de ferramentas que implementem tais técnicas impede a coleta de dados concretos e, conseqüentemente, de uma comparação mais apurada.

Apesar de ser uma comparação 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 não tendo nenhum ponto negativo. Todas as restantes possuem algumas características não satisfatórias, dificultando um pouco a escolha de uma técnica complementar. Considerando

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 1: Comparação das técnicas estudadas.

a utilização de pontos de vista e a consequente necessidade de filtrar os inúmeros pontos de vistas existentes (e visões geradas), a REQAV se destaca, podendo aplicar seus conceitos em um processo rápido e eficiente para o requerido processo.

4.2 Estudo sobre características de sistemas gerenciadores de conhecimento

O estudo sobre os diversos sistemas de gerenciamento de conhecimento sugere a importância da adoção de sistemas que utilizem dados de uma base de conhecimento. Porém, a utilização de bases de conhecimento para apoiar o processo de engenharia de requisitos esbarra em alguns problemas, principalmente na complexidade da construção de tais sistemas de gerenciamento de conhecimento. A criação de mecanismos de inferência e ferramentas para extração de conhecimento (provavelmente utilizando analisadores estatísticos), sem esquecer dos problemas inerentes da distribuição deste conhecimento (principalmente segurança, problema não abordado em nenhum dos sistemas estudados no projeto), exige um esforço muito grande. O amplo espectro de tal sistema, que poderia suportar todo o processo de desenvolvimento de software, e não apenas a engenharia de requisitos, leva também à direção de que trabalhos futuros sejam desenvolvidos nesta área, abordando não apenas as necessidades da engenharia de requisitos mas também de todas as outras etapas do processo de engenharia de software.

Algumas características desejáveis de tal sistema podem ser desde já delineadas. A separação do conhecimento e meta-conhecimento permite um gerenciamento mais fácil das informações, facilitando a evolução dos sistemas de maneira independente. A estratégia do *Osirix*, recebendo dados como documentos XML, com validação do mesmo antes de entrar efetivamente no sistema, facilita a manipulação de dados. Considerando que existe uma maneira de transformar os modelos desenvolvidos durante o desenvolvimento de software em documentos XML (através do XMI), a criação de mecanismos semelhantes não seria muito complicada.

A utilização de transformações do XML para apresentação do conteúdo dos documentos aos usuários (através de XSL, XSLT, etc) também é algo a ser explorado. Não somente quanto à visualização pela Internet dos dados armazenados no repositório, mas também para uso pelas ferramentas que utilizem tais dados, recebendo os documentos XML e convertendo-os em um

formato mais adequado para uso interno.

Por fim, adicionando a criação de contextos situacionais, de maneira semelhante ao *kMail*, tem-se então um poderoso repositório, com ótimas capacidades de pesquisa e boas perspectivas de interoperabilidade com outros sistemas (devido a adoção de padrões como XML e XMI para os dados e XSL para transformar os dados).

4.3 Ferramenta

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 criação automático de um modelo de caso de uso a partir de visões entradas no sistema. A seguir será exposta a arquitetura dos principais subsistemas e como eles vão interagir sob o comando da CoolCase, que é o frontend gráfico a ser utilizado pelo engenheiro de requisitos.

4.3.1 Framework veryhot

A necessidade de criação de diagramas de casos de uso atendendo a notação utilizada neste trabalho e a possível futura necessidade de novos diagramas do mesmo tipo motivou a criação de um pacote que facilite esta tarefa. Seu nome é veryhot. Na verdade, ele é uma evolução de um pacote Java desenvolvido em (YANAGA, 1997). Fora feita uma reengenharia do mesmo, simplificando-o (diminuição de números de classes) sem perder suas características originais, 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 13 e 14, pode-se observar a estrutura do pacote:

Tendo conhecimento do pacote, é possível enumerar alguma de suas características principais:

- Simplicidade: Para criar novas figuras (um Actor, por exemplo), basta extender o VectorFigure ou criar uma nova classe que especialize a Figure. Muito provavelmente só necessitará de alterações o método paint().
- 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.

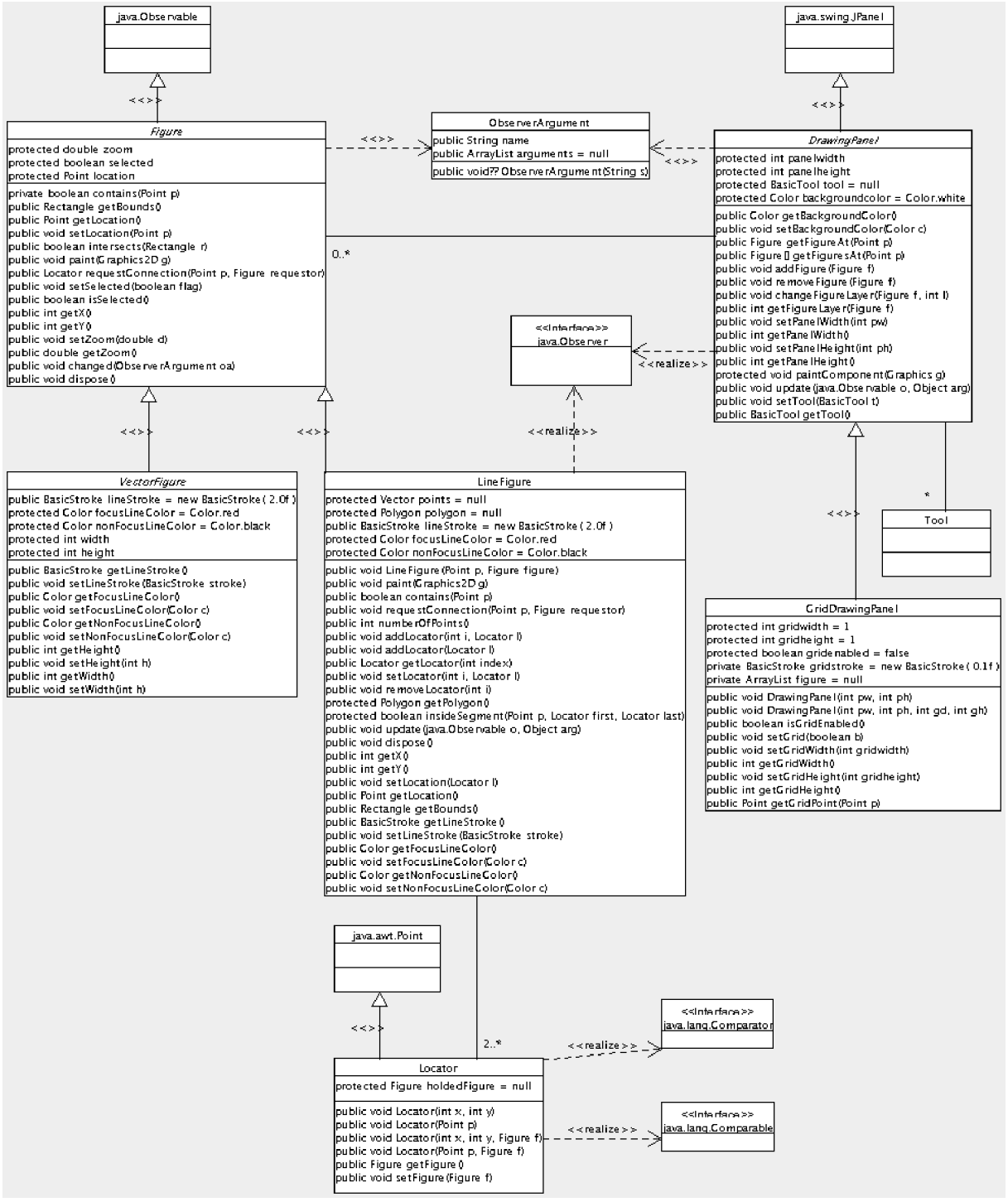


Figura 13: Diagrama de classes - Figures, DrawingPanel, ObserverArgument (package veryhot)

- Documentação: este framework está muito melhor documentado que a versão utilizada como base. Unindo a reengenharia com uma documentação mais detalhada, sua utilização ficou mais simples.

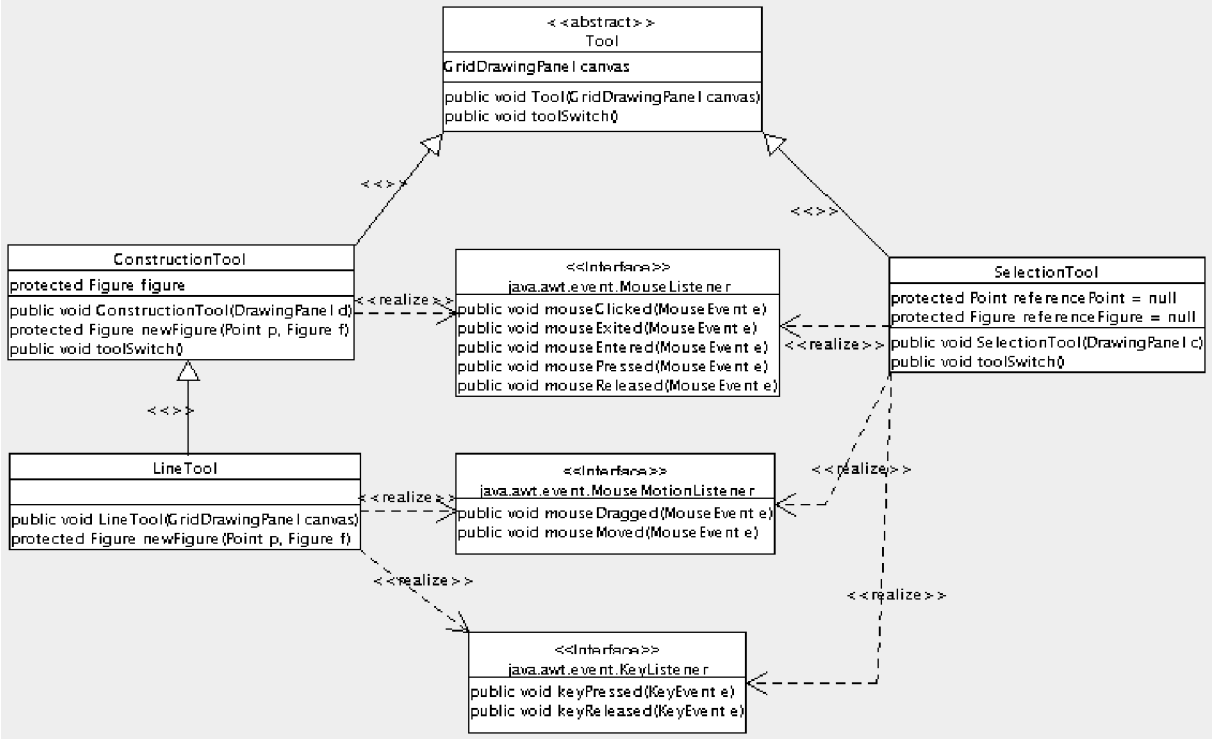


Figura 14: Diagrama de classes - Tool - (package veryhot.tool)

4.3.2 Kernel

Novamente, reutilizando soluções criadas em trabalhos anteriores (BESSANI, 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 hierárquia de memórias, resultando em um gerenciamento mais eficiente e rápido dos objetos. A nova solução também evita possíveis problemas de acesso concorrentemente a um mesmo objeto que existia no sistema anterior.

Atualmente, três dispositovos 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.
- FileBasedMemory: Realiza a persistência do objeto em arquivo, utilizando o mecanismo de serialização em arquivos disponível no Java.

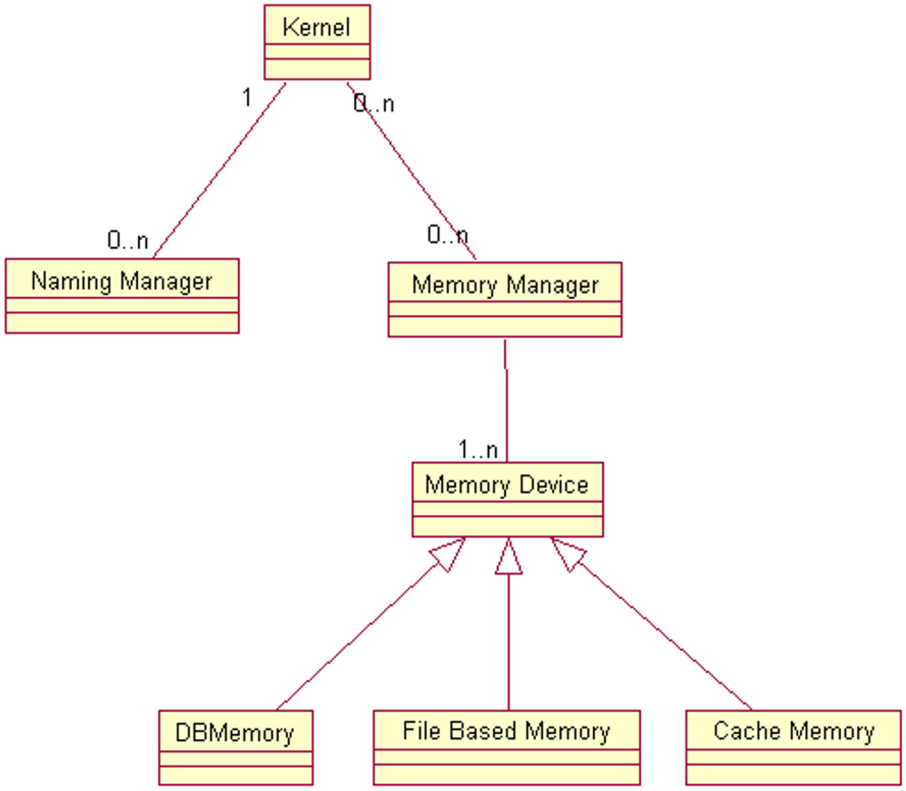


Figura 15: Diagrama de classes - Kernel

4.3.3 Arquitetura

Em linhas gerais, a arquitetura do sistema pode ser descrita como mostrado na figura 16. A ferramenta, CoolCase, interage com os seguintes serviços:

- Entity Service: Contém todas as entidades do sistema.
- Use Case Model Service: Gerencia os casos de uso, os atores e seus intra-relacionamentos.
- Use Case Diagram Service: Guarda diagramas de caso de uso.
- View Service: Gerencia as visões.
- Evalution Service: Aplica regras, 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.

4.3.4 Componentes do sistema

O sistema manipula sete tipos de componentes: entity, requirement, use case, actor, use case model e view, conforme pode ser visto em 17. Vários deles são uma especialização de Artefact, sobre o qual existe um controle de versão, facilitando assim a rastreabilidade de modificações.

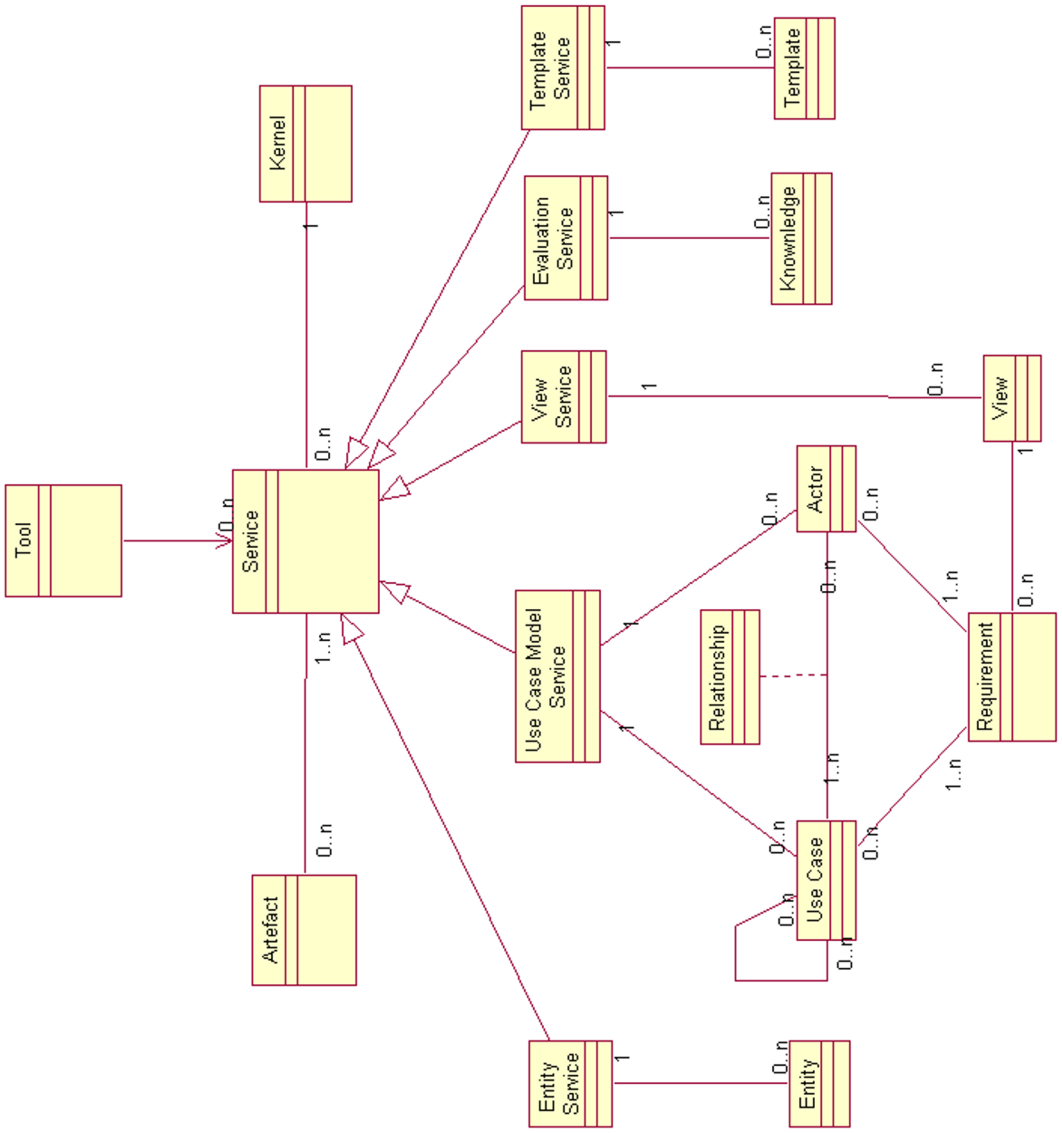


Figura 16: Arquitetura da ferramenta

Entity Toda visão está relacionada uma entidade, garantindo assim uma certa rastreabilidade. Mas o mais importante é a capacidade de avaliar artefatos, principalemnte visões, de acordo com as entidadas 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 1.4, contendo as extensões para suporte a MDSODI, conforme explicado anteriormente na seçao 3.9.

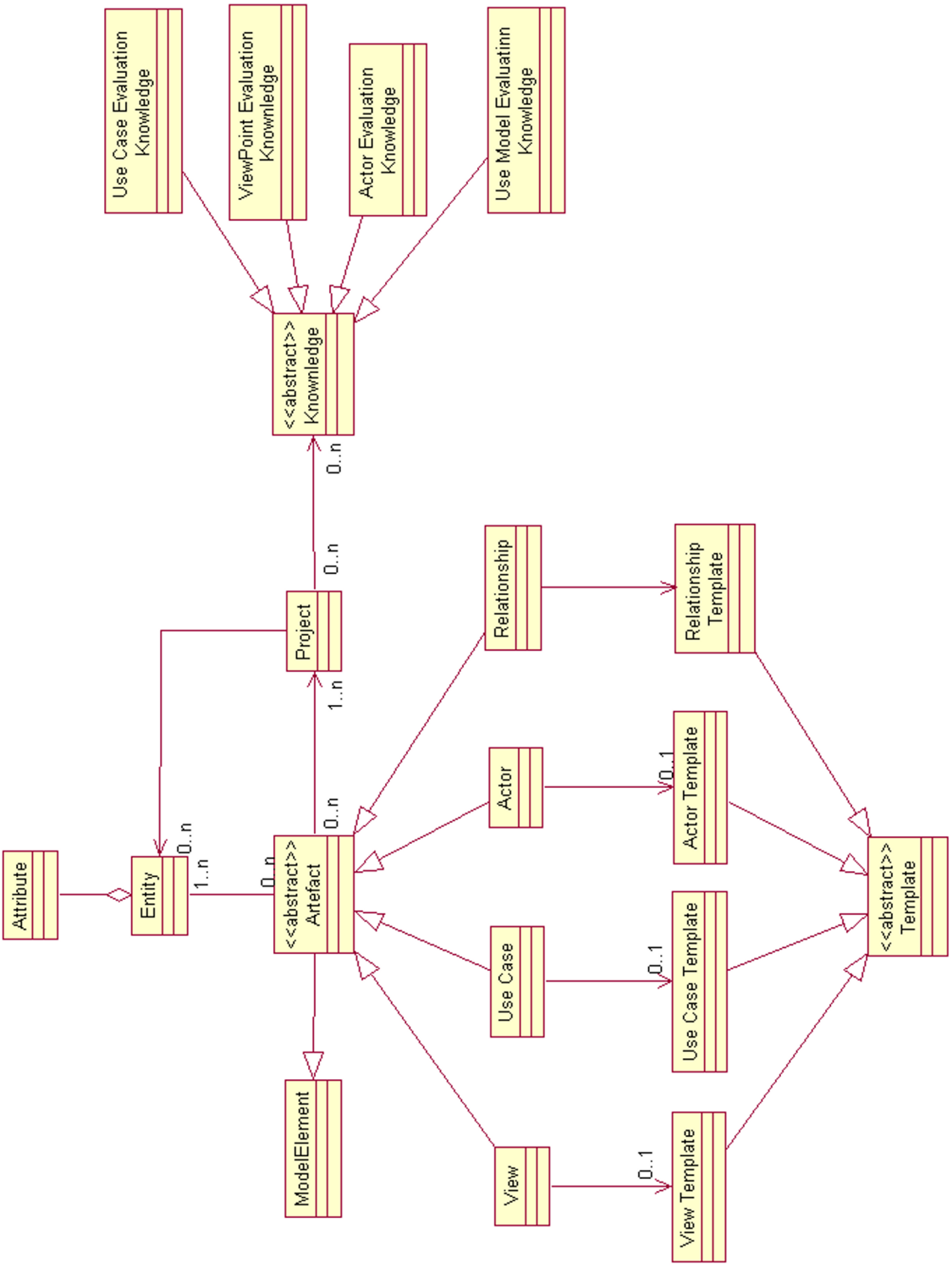


Figura 17: Arquitetura básica do sistema

Use Case Model Os modelos de caso de uso são compostos pelos atores, casos de uso e seus relacionamentos.

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);
- 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 templatess tem como principal função facilitar a criação de pontos de vista. Sua estrutura é muito semelhante a de um ponto de vista, exceto que não possui um grau de importância e seus requisitos são, na verdade, indagações, orientações, informações que visam a 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 intrísecas 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.

4.3.5 CoolCase

Denominou-se CoolCase a parte da ferramenta com a qual o engenheiro irá trabalhar. Neste serão criados e editados os diagramas, sendo possivel acessar, através dela, as visões, requisitos, enfim, todos os elementos citados neste projeto. Na figura 18 pode-se observar o diagrama de classes da ferramenta. Como todo o resto do sistema, fora implementada em Java.

Na figura 19, pode-se visualizar um protótipo da ferramenta em execução.

4.4 Método de Engenharia de Requisitos Proposto

O objetivo da ferramente é automatizar um processo de engenharia de requisitos, ajudando a aplicação de um método. Em (GRAVENA, 2000), é descrito um processo de engenharia de requisitos que foi utilizado para visionar este método.

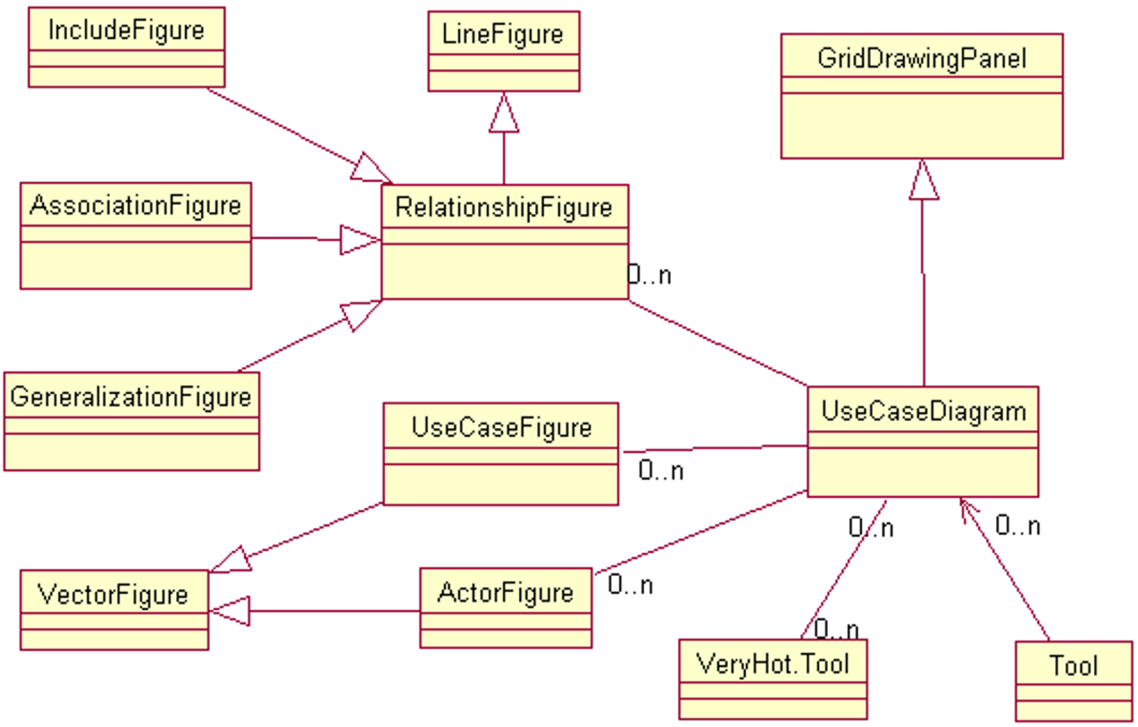


Figura 18: Diagrama de classe da CoolCase

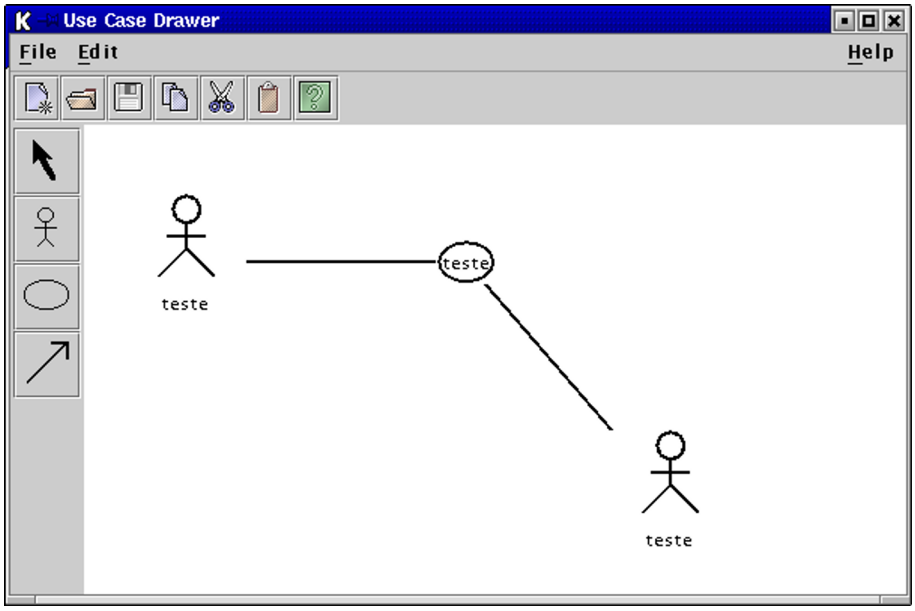


Figura 19: Protótipo da ferramenta em funcionamento

4.4.1 Captura inicial de dados

A primeira etapa consiste na obtenção dos dados sobre o sistema: entrevistas, questionários, observações *in loco*, possíveis sistemas já existentes, leis que devem ser obedecidas. A entrada de todos estes dados é feita através de visões. Como no início do processo não se conhece todos os envolvidos no sistema, o cadastro destas pessoas (entidades) e suas visões é feita simultaneamente. Conforme são feitas iterações do processo, o normal é que as entidades envolvidas se estabilizem e somente novas visões sejam criadas.

A criação de visões é ajudada com os modelos de visão já cadastrados no sistema. Escolhido o foco e o assunto (*concern*), pode escolhar um modelo para ajudar na criação da visão. Se não existir um modelo correspondente, posteriormente o engenheiro de requisitos poderá criar um modelo a partir dos dados, idéias e dúvidas que surgem das visões que são utilizadas como base.

4.4.2 Análise

Com as informações recém-coletadas, torna-se possível definir o modelo de domínio. Nesta etapa escolhe-se também as regras que serão utilizadas para avaliar as visões, casos de uso e atores. As regras mais importantes são as sobre visões, cuja responsabilidade será tentar classificar da melhor maneira possíveis as visões em relação ao ponto de vista tomado e a entidade que a criou.

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 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 agregem várias técnicas, suportando os padrões de projeto dos software atuais e suas notações. Na pesquisa aqui relatada, uma grande quantidade de técnicas foi abordada, nenhuma com enfoque em objetos, quanto mais objetos distribuídos.

Embora o protótipo não tenha sido concluído, todo este estudo indica possíveis direções para novos projetos, em diferentes níveis de abrangência. Em se tratando de engenharia de requisitos, a definição de regras adequadas, que consigam eliminar o maior número de visões e mesmo assim garantir a satisfação dos stakeholders é um alvo de estudos importante. Com a maior organização dos dados obtidos na engenharia de requisitos, novas métricas podem ser definidas para esta fase de um processo de engenharia de software, podendo avaliar melhor os caminhos a serem tomados. Uma outra área de pesquisa a ser trabalhada é a de sistemas de conhecimento, que possui efeito benéfico inclusive no processo em si e não apenas em seus resultados.

Este trabalho, por ser alvo de meu trabalho de graduação, ainda será continuado por alguns meses. Várias das características e abordagens acima descritas serão melhor trabalhadas, principalmente as que tangem a área de engenharia de requisitos, esperando-se assim a finalização de um protótipo que agregue técnicas modernas e que seja mais funcional.

Referências

BALZER, R. Tolerating inconsistency. In: **13th International Conference on Software Engineering**. Austin, Texas, USA: IEEE Computer Society Press, 1991. p. 158–165.

BESNARD, P.; HUNTER, A. Quasi-classical logic: Non-trivializable classical reasoning from inconsistency information. **Symbolic and Quantitative Approaches to Uncertainty**, 1995.

BESSANI, A. N. Relatório de Iniciação Científica. 2000.

BORTOLI, L. A.; PRICE, A. M. Um método para auxiliar a definição de requisitos. In: . Cancun-México: IDEAS 2000: Jornada Ibero Americana de Ingeneria de Requisitos Y Ambientes de Software, 2000. p. 1–12.

BOSAK, J. et al. **Extensible Markup Language (XML) 1.0**. First ed. [S.l.], February 1998.

CALDWELL, N. H. M. et al. Web-based knowledge managment for distributed design. **IEEE Intelligent Systems**, p. 40–47, May/June 2000.

DAMIAN, D. E. H. et al. Using different communication media in requirements negotiation. **IEEE Software**, p. 28–36, May-June 2000.

DíAS, I.; METTEO, A. Objectory process stereotypes. **JOOP**, p. 29–38, June 1999.

DIAZ, J. S.; FERRAGUD, V. P.; PELOZO, E. I. Un entorno de generation de protótipo de interfaces de usuário a partir de diagramas de interacción. In: . Cancun-México: IDEAS 2000: Jornada Ibero Americana de Ingeneria de Requisitos Y Ambientes de Software, 2000. p. 145–154.

EASTERBROOK, S.; NUSEIBEH, B. Using viewpoints for inconsistency management. **IEE Software Engineering Journal**, November 1995.

FICKAS, S.; LAMSWEERDE, A. van; DARDENNE, A. Goal-directed concept acquisition in requirements elicitation. In: IEEE COMPUTER SOCIETY PRESS. **6th International Workshop on Software Specification and Design**. Como, Italy: [s.n.], 1991. p. 14–21.

FINKELSTEIN, A.; KRAMER, J.; GOEDICKE, J. K. Viewpoints oriented software specification. In: IEEE COMPUTER SOCIETY. **3rd International Workshop on Software Engineering and its Applications**. Toulouse, France: [s.n.], 1990. p. 337–351.

FOWLER, M.; SCOTT, K. **UML Distilled: Applying the Standard Object Modeling Language**. [S.l.]: Addison Wesley, 1997.

FOWLER, M. **UML distilled: a brief guide to the standard object modeling language**. Second ed. [S.l.]: Addison Wesley Longman, 2000. (Object Technology).

GRAVENA, J. P. Aspectos importatnes de uma metodologia para desenvolvimento de software com objetos distribuídos. Monografia final de curso. 2000.

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.

HUZITA, E. H. M. Uma metodologia para auxiliar o desenvolvimento de aplicações para processamento paralelo. Tese de doutorado. Universidade de São Paulo. 1995.

HUZITA, E. H. M. Uma metodologia de desenvolvimento baseado em objetos distribuídos inteligentes. Projeto de Pesquisa. 1999.

JACOBSON, I.; BOOCH, G.; RUMBAUGH, J. **The Unified Software Development Process**. Second ed. Massachusetts: Addison Wesley Longman, 1999. (Object Tecnology).

KNAPIK, M.; JOHNSON, J. **Developing Intelligent Agents for Distributed System: Exploring Architecture, Technologies and Applications**. EUA: McGraw Hill, 1998.

KOTONYA, G.; SOMMERVILLE, I. **Requirements Engineering With Viewpoints**. Lancaster, LA1 4YR, UK, 1995.

LEITE, J. Enhacing a requirements baseline with scenarios. **Requirements Engineering Journal**, v. 2, n. 4, 1997.

LEITE, J. C. P. Viewpoint analysis: A case study. **ACM J. Software Engineering Notes**, v. 14, n. 3, p. 111–119, 1989.

MACAULAY, L. A. **Requirements Engineering**. [S.l.: s.n.], 1996.

MULLERY, G. P. A method for controlled requirements specifications. In: IEEE COMPUTER SOCIETY. **4th International Conference on Software Engineering**. [S.l.: s.n.], 1979. p. 126–135.

NARAYANASWAMY, K.; GOLDMAN, N. Lazy consistency: A basis for cooperative software development. In: SIGOIS, A. S. . (Ed.). **International Conference on Computer-Supported Cooperative Work**. Toronto, Ontario, Canada: [s.n.], 1992. p. 257–264.

O'LEARY, D. E. Using ai in knowledge managment: Knowldege bases and ontologies. **IEEE Intelligent Systems**, v. 13, n. 3, p. 34–39, may/june 1998.

PFAFFENSELLER, M.; PFAFFENSELLER, M.; KROTH, E. Uma ferramenta de apoio ao gerenciamento de componentes. **Workshop de Desenvolvimento Baseado em Componentes**, Junho 2001.

RABARIJAONA, A. et al. Building and searching an xml-based corporate memory. **IEEE Intelligent Systems**, p. 56–63, May/June 2000.

RIDAO, M.; DOORN, J.; PRADO LEITE, J. C. S. do. Uso de patrones en la construcción de escenarios. In: . [S.l.]: WER, 2000. p. 140–157.

SATO et al. Onix: An environment for the development of parallel object oriented software. In: **Internacional Workshop on High Performance Computing**. São Paulo: [s.n.], 1994. p. 167–183.

SCHELEBBE, H. **Distributed PROSOFT**. Germany, 1995.

SCHOMAN, K.; ROSS, D. T. Structured analysis for requirements definition. **IEEE Transactions on Software Engineering**, v. 3, n. 1, p. 6–15, 1977.

SCHWANKE, R. W.; KAISER, G. E. Living with inconsistency in large systems. In: . Grassau, Germany: B. G. Teubner, Stuttgart, 1988. p. 98–118.

SCHWARTZ, D. G.; TE'ENI, D. Tying knowledge to action with kmail. **IEEE Intelligent Systems**, p. 33–39, May/June 2000.

SOMMERVILLE, I. **Software Engineering**. Fifth ed. Massachussets: Addison Wesley, 1996. (International Computer Science).

SOMMERVILLE, I.; SAWYER, P. Viewpoints: principles, problems and a pratical approach to requirements engineering. 1997.

SOUZA, C. T.; OLIVEIRA, M. Ábaco. Um ambiente de desenvolvimento baseado em objetos distribuídos configuráveis. In: **XII Simpósio Brasileira de Engenharia de Software**. Maringá-PR: [s.n.], 1998. p. 205–220.

ST-DENIS, G.; SCHAUER, R.; KELLER, R. K. Selecting a model interchange format. In: IEEE (Ed.). [S.l.: s.n.], 2000.

SZYKMAN, S. et al. Design repositories: Engineering design's new knowledge base. **IEEE Intelligent Systems**, p. 48–55, May/June 2000.

TORO, A. D. et al. Identificación de patrones de reutilización de requisitos de sistemas de información. In: . [S.l.]: WER, 2000.

WOOLDRIDGE, M. J.; JENNINGS, N. R. Software engineering with agents: Pitfalls and pratifalls. **IEEE Internet Computing**, v. 3, n. 3, p. 20–27, may-jun 1999.

YANAGA, E. Relatório de Iniciação Científica. 1997.

ZANLORENCI, E. P.; BURNETT, R. C. Reqav: Modelo para descrição, qualificação, análise e validação de requisitos. In: **IDEAS2000: Jornada Ibero Americana de Ingeneria de Requisitos Y Ambientes de Software**. [S.l.: s.n.], 2000. p. 61–72.