Um Estudo Sobre os Riscos Inerentes à Implantação do Reuso de Componentes no Processo de Desenvolvimento de Software

Mauricio Gruhn Sanches

Dissertação de Mestrado Profissional

Instituto de Computação

Universidade Estadual de Campinas

Um Estudo Sobre os Riscos
Inerentes à Implantação do Reuso de Componentes
no Processo de Desenvolvimento de Software

Mauricio Gruhn Sanches

Campinas, 28 de junho de 2005.

Banca Examinadora:

Profª. Dra. Ana Cervigni Guerra (orientadora)
Centro de Pesquisas Renato Archer – CenPRA/MCT

Prof. Dr. Mário Lúcio Côrtes
Instituto de Computação - UNICAMP

Profª. Dra. Silvia Mª. Fonseca Silveira Massruhá
CNPTIA / EMBRAPA

Profª. Dra. Ariadne Mª. Brito Rizzoni Carvalho (suplente)
Instituto de Computação - UNICAMP

FICHA CATALOGRÁFICA ELABORADA PELA

BIBLIOTECA DO IMECC DA UNICAMP

Bibliotecário: Maria Júlia Milani Rodrigues – CRB8a / 2116

Sanches, Mauricio Gruhn

Sa55e Um estudo sobre os riscos inerentes à implantação do reuso de componentes no processo de desenvolvimento de software / Mauricio Gruhn Sanches -- Campinas, [S.P. :s.n.], 2005.

Orientadora : Ana Cervigni Guerra

Trabalho final (Mestrado Profissional) - Universidade Estadual de Campinas, Instituto de Computação.

1. Engenharia de software. 2. Componentes de software. 3. Avaliação de riscos. 4. Sotware - Reutilização. I. Guerra, Ana Cervigni. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Título em inglês: Study about implementation risks of component reuse in the software development process.

Palavras-chave em inglês (Keywords): 1. Software engineering. 2. Component software. 3. Risk evaluation. 4. Software reusability.

Área de concentração: Engenharia de Software

Titulação: Mestre em Computação

Banca examinadora: Prof. Dr. Mário Côrtes (IC-UNICAMP)
Profa. Dra. Silvia Massruhá (EMBRAPA/CNPTIA)

Data da defesa: 28/06/2005

TERMO DE APROVAÇÃO

Trabalho Final Escrito defendido e aprovado em 28 de junho de 2005, pela Banca Examinadora composta pelos Professores Doutores:

Profa. Dra. Silvia Maria Fonseca Silveira Massruhá
EMBRAPA

Prof. Dr. Mário Lucio Cortes
IC - UNICAMP

Profa. Dra. Ana Cervigni Guerra
CENPRA

Um Estudo Sobre os Riscos
Inerentes à Implantação do Reuso de Componentes
no Processo de Desenvolvimento de Software

Este trabalho corresponde à redação final da Dissertação devidamente corrigida e defendida por Mauricio Gruhn Sanches e aprovada pela banca examinadora.

Campinas, 28 de junho de 2005.

Profa. Dra. Ana Cervigni Guerra

(orientadora)

Dissertação apresentada ao Instituto de Computação da UNICAMP como requisito parcial para a obtenção do título de Mestre em Computação.

Resumo

No mundo globalizado de hoje, a necessidade de se prover sistemas para o gerenciamento do grande volume de informações gerado a cada dia torna imperativa a busca contínua por novas práticas e formas de desenvolvimento de software para a manipulação dessas informações. Isso está vinculado também à necessidade de se conseguir ganhos de qualidade, produtividade e redução de custos em tais desenvolvimentos, pois esses são fatores fundamentais no sucesso do negócio de empresas ligadas à tecnologia da informação. Nesse cenário, o reuso de componentes no processo de desenvolvimento de software vem a exercer um papel importante na concretização desses objetivos.

Este trabalho apresenta os conceitos envolvidos no contexto de um processo de desenvolvimento de software baseado em componentes reusáveis. São apresentadas as diferentes formas de reuso, o conceito de componentes reusáveis e as formas que esses componentes podem assumir dentro de um domínio de aplicação. Componentes possuem características diferentes, podem ser originados a partir de diversas fontes e carregam detalhes importantes que devem ser considerados durante a sua documentação. Todos esses aspectos influenciam o grau de reuso de um componente dentro do projeto ou organização. São apresentadas algumas tecnologias que promovem a aplicação do reuso de componentes e os conceitos envolvidos na definição, gerenciamento e implantação de repositórios de componentes reusáveis, fundamentais na consolidação do processo de reuso. Novos papéis e responsabilidades envolvidos no processo de reuso são apresentados, além de exemplos de métricas a serem utilizadas com o intuito de auxiliar a medição dos benefícios proporcionados pelo reuso dentro de um processo já consolidado.

O resultado desse trabalho é uma análise crítica em relação aos benefícios do reuso e os riscos associados à sua implementação no processo de desenvolvimento de software, considerando-se os aspectos tecnológicos, humanos e econômicos envolvidos em cada tópico abordado ao longo do trabalho.

Abstract

In a globalized world, the needs to release new systems to manage the large volume of information generated every day force a continuous search for new software development practices in order to handle this information. This is related to the needs to obtain quality and productivity improvements, as well as cost reduction in such development, because these are the main success factors of Information Technology companies. In this scenario, the component reuse in the software development process plays an important role in order to achieve these goals.

This work presents the concepts around the context of a software development process based on reusable components. Different types of reuse are explained, as well as the concept of a reusable component and types of components inside an application domain. Components have different characteristics, are originated from different sources and carry on important details that should be analyzed during its documentation. All these aspects have influence on the reuse level of the components inside the project or the company. Some technologies that promote the component reuse and the concepts involved in the definition, management and implementation of reusable software components, which are the basis to consolidate the reuse process, are presented. New roles and responsibilities involved in the reuse process are shown, and also some metrics used to help to measure the benefits due to a consolidated reuse process.

The results of this work are a critical analysis about the reuse benefits and the risks associated to its implementation in the software development process, taking into account the technological, human and economic factors involved on each topic discussed along this work.

Agradecimentos

Agradeço a Deus pelas graças recebidas, pela saúde e proteção a mim dedicada, e pelas pessoas colocadas em minha vida.

Agradeço aos meus pais e irmã pela educação, pelo companheirismo, pelas lições de honestidade, humildade, solidariedade e, sobretudo, pelo grande amor recebido em todos os momentos da vida.

À minha namorada, Eliana, pelo incentivo, paciência e tolerância à minha ausência devido às atividades relacionadas a esse trabalho.

À minha orientadora, Ana Cervigni Guerra, que tanto me ajudou através de idéias, cobranças e incentivos que resultaram no meu amadurecimento acadêmico e profissional.

À minha colega de trabalho Ana Florêncio, cujo apoio foi fundamental para concretizar a escolha do tema e o início do presente trabalho.

À UNICAMP e sua estrutura, em especial aos professores e funcionários do IC, que me proporcionaram essa oportunidade.

Aos meus amigos e familiares, que indiretamente contribuíram para esse trabalho simplesmente por estarem ao meu lado. O calor humano é indispensável ao sucesso de qualquer atividade.

Obrigado.

À minha família.

Índice

CAPÍTULO 1 - INTRODUÇÃO 1

1.1. OBJETIVO DO TRABALHO 2
1.2. MOTIVAÇÃO 2
1.3. ESTRUTURA DA DISSERTAÇÃO 2
1.4. DEFINIÇÃO DE COMPONENTES 3
1.5. EVOLUÇÃO DAS TÉCNICAS DE REUSO 5
1.6. CATEGORIAS DE REUSO 6

CAPÍTULO 2 - COMPONENTES REUSÁVEIS 9

2.1. CLASSIFICAÇÃO DOS COMPONENTES 11
2.2. GRANULARIDADE DOS COMPONENTES 13
2.3. MODELOS DE COMPONENTES REUSÁVEIS 15
2.4. CARACTERÍSTICAS DE COMPONENTES REUSÁVEIS 16
2.5. MODELO 3-C 19
2.6. AQUISIÇÃO DE COMPONENTES 21
2.7. EVOLUÇÃO DOS COMPONENTES 22
2.8. DOCUMENTAÇÃO DO COMPONENTE 23
2.9. PROCESSO DE CRIAÇÃO DE COMPONENTES 24

CAPÍTULO 3 - TECNOLOGIAS DE REUSO 27

3.1. TÉCNICAS OO 28
3.2. DESIGN PATTERNS 30
3.3. APPLICATION FRAMEWORKS 31
3.4. DESENVOLVIMENTO BASEADO EM COMPONENTES 37
3.5. SISTEMAS BASEADOS EM AGENTES 41
3.6. ARQUITETURAS 42
3.7. COMPARATIVO ENTRE AS TÉCNICAS APRESENTADAS 44

CAPÍTULO 4 - REPOSITÓRIO 45

4.1. VANTAGENS DO REPOSITÓRIO 46
4.2. CONTEÚDO DO REPOSITÓRIO 47
4.3. FUNÇÕES DO REPOSITÓRIO 49
4.4. CENÁRIOS DE UTILIZAÇÃO 51

4.5. IMPLEMENTAÇÃO 53
4.6. REPOSITÓRIOS PÚBLICOS 56

CAPÍTULO 5 - PROCESSOS 61

5.1. CLASSIFICAÇÃO DOS PROCESSOS....63
5.2. CLASSIFICAÇÃO DOS PAPÉIS E RESPONSABILIDADES....66
5.3. O FATOR HUMANO 68
5.4. ANÁLISE CRÍTICA 69

CAPÍTULO 6 - MÉTRICAS 71

6.1. EXEMPLOS DE MÉTRICAS 72
6.2. MÉTRICAS DO REPOSITÓRIO....74
6.3. MÉTRICAS DO PROJETO 76

CAPÍTULO 7 - RISCOS 77

7.1. RECOMENDAÇÕES PARA O SUCESSO DO REUSO 84

CONCLUSÃO 87

TRABALHOS FUTUROS....93

BIBLIOGRAFIA 95

APÊNDICE A - OBSERVER PATTERN 101

Lista de tabelas

TABELA 3-1 - COMPARATIVO ENTRE TECNOLOGIAS ACTIVEX E JAVABEANS 40
TABELA 3-2 - TECNOLOGIAS: VANTAGENS E DESVANTAGENS 44
TABELA 5-1 - PROCESSOS DE REUSO 65
TABELA 6-1 - OBJETIVOS PARA MÉTRICAS DE REUSO 73
TABELA 6-2 - MÉTRICAS DE REUSO 74

Lista de figuras

FIGURA 2-1– EVOLUÇÃO DOS COMPONENTES E POTENCIAL DE REUSO 22
FIGURA 2-2 - AS TRÊS FASES DA PROGRAMAÇÃO ORIENTADA A COMPONENTES..... 25
FIGURA 3-1 - APLICAÇÃO DESENVOLVIDA TOTALMENTE 35
FIGURA 3-2 - APLICAÇÃO DESENVOLVIDA REUTILIZANDO CLASSES DE BIBLIOTECA 35
FIGURA 3-3 - APLICAÇÃO DESENVOLVIDA REUTILIZANDO UM *FRAMEWORK*..... 35
FIGURA 4-1 - MODELO DE IMPLEMENTAÇÃO DE REPOSITÓRIO 48
FIGURA 4-2 - REPOSITÓRIO DE COMPONENTES E PROCESSOS ENVOLVIDOS 55
FIGURA A-1 - DIAGRAMA DE CLASSES DO *OBSERVER PATTERN*..... 102
FIGURA A-2 - DIAGRAMA DE SEQÜÊNCIA ENTRE CLASSES DO *OBSERVER PATTERN*..... 103

Lista de Abreviaturas

ACE	*Adaptive Communication Environment*
ADL	*Advanced Distributed Learning*
API	*Application Programming Interface*
B2B	*Business to Business*
B2C	*Business to Consumer*
COM	*Component Object Model*
CORBA	*Common Object Request Broker Architecture*
DCMI	*The Dublin Core Metadata Initiative*
DCOM	*Distributed Component Object Model*
GQM	*Goal-Question-Metric*
GUI	*Graphical User Interface*
IDL	*Interface Definition Language*
JGAP	*Java Genetic Algorithms Package*
MVC	*Model-View-Controller*
OLE	*Object Linking and Embedding*
OMG	*Object Management Group*
OMT	*Object Modeling Technique*
OO	*Object Oriented*
OOA	*Object Oriented Analysis*

OOD	*Object Oriented Design*
OOP	*Object Oriented Programming*
OOSE	*Object-Oriented Software Engineering*
OOT	*Object Oriented Techniques*
OSGi	*Open Services Gateway Initiative*
RAS	*Reusable Asset Specification*
RETSINA	*Reusable Environment for Task-Structured Intelligent Networked Agents*
RPC	*Remote Procedure Call*
SA4J	*Structural Analysis for Java*
SCORM	*The Sharable Content Object Reference Model*
UML	*Unified Modeling Language*
WebDAV	*Web-based Distributed Authoring and Versioning*
XML	*Extensible Markup Language*

Capítulo 1

Introdução

No desenvolvimento de software busca-se alcançar os requisitos da aplicação a ser desenvolvida de maneira rápida, fácil, com baixos custos e alta qualidade. Para que se possa atingir estes objetivos, a Engenharia de Software atua na aplicação de abordagens ao desenvolvimento de software, buscando a melhora da qualidade e o aumento da produtividade [Klabunde00].

A reutilização de software é um fator que pode levar ao aumento da qualidade e da produtividade na atividade de desenvolvimento de software [Silva00] e isto se baseia na perspectiva de que o reuso de artefatos de software[1] já desenvolvidos e depurados reduza o tempo de desenvolvimento, o tempo de testes e as possibilidades de introdução de falhas na produção de novos artefatos.

Reuso é a chave para o paradigma do grande aumento da produtividade e qualidade no desenvolvimento de software em larga escala [Silva96] e também um dos principais benefícios da tecnologia de objetos e componentes, possibilitando o rápido desenvolvimento de sistemas complexos.

---

[1] A expressão “artefato de software” é usada nesse trabalho de forma genérica, não se referindo necessariamente a código, podendo abranger também produtos de análise e projeto, como documentos de *design*, por exemplo. Além disso, pode se referir a uma aplicação, *framework* ou componente.

1.1. Objetivo do trabalho

O presente trabalho visa primeiramente estudar e compilar o estado da arte no que se refere a reusabilidade de componentes em projetos de software de sistemas voltados ao mercado corporativo, analisando os conceitos envolvidos na implantação de um processo de reuso. Com base nesse estudo são apresentados os riscos envolvidos na implantação do processo de reuso de componentes, bem como fatores para a redução dos riscos, levando-se em consideração restrições como dinamismo de mercado, tempo de implantação de produtos, abrangência dos sistemas beneficiados pelo reuso e limitações tecnológicas.

1.2. Motivação

A motivação para o estudo do assunto envolvendo reuso de componentes vem do fato que atualmente existe uma preocupação do governo brasileiro no sentido de incentivar a pesquisa no tema e para isso a Finep (Financiadora de Estudos e Projetos), agência de fomento do Ministério da Ciência e Tecnologia (MCT), financiará projetos de pesquisa no valor de R$ 1,8 milhão para a criação da primeira biblioteca pública de componentes de software da América Latina. Existe também na comunidade uma percepção das vantagens em se adotar um repositório de componentes de software e se aplicar metodologias de reuso em processos de desenvolvimento de software. Nesse âmbito é importante que todo o processo de reuso seja compreendido e seus riscos plenamente avaliados.

1.3. Estrutura da dissertação

Na presente introdução é apresentado um histórico da evolução do reuso, a definição de componentes reusáveis e as categorias existentes de reuso. No capítulo 2 os conceitos em torno de componentes reusáveis são explicados.

As tecnologias mais importantes empregadas no desenvolvimento de projetos voltados ao reuso são apresentadas no capítulo 3. O propósito não é fornecer detalhes técnicos de como utilizar essas tecnologias, mas sim apresentar a importância delas na facilitação e alcance dos objetivos do reuso dentro de um determinado projeto. O capítulo 4 é dedicado exclusivamente ao repositório de componentes. Item importante no processo de reuso, o repositório deve ser planejado, implantado e gerenciado apropriadamente. O capítulo apresenta as características e funções do repositório, um modelo de implementação, os cenários envolvidos na utilização do repositório e exemplos de repostórios públicos disponíveis na Internet.

No capítulo 5 o processo de implantação do reuso dentro da organização é detalhado, apresentando os sub-processos envolvidos e a importância da definição de novos papéis e responsabilidades para as pessoas envolvidas nas atividades de reuso de componentes. O capítulo 6 define algumas métricas de reuso, de forma a permitir a quantificação das vantagens e da evolução do processo de reuso.

Com base nos conceitos e processos apresentados nos capítulos anteriores, os riscos envolvidos na implantação e manutenção do reuso de componentes são apresentados no capítulo 7. Os aspectos técnicos e humanos são considerados na avaliação dos riscos, que podem ser de maior ou menor grau, dependendo de diversos fatores. Recomendações para minimizar os riscos e alcançar o sucesso do reuso também são apresentadas.

1.4. Definição de componentes

Neste item apresentam-se as definições que serão utilizadas para o entendimento da dissertação e que são de certo modo uma generalização ou uma adaptação de alguns autores.

Um componente de software é um elemento de software que se

submete a um modelo de componentes, pode ser instalado independentemente e se compor sem modificação conforme um padrão de composição. Também se define um modelo de componentes como sendo quem estabelece os padrões de interação e de composição para os componentes de software de um domínio especifico. Um padrão de interação define a forma e as operações que permitem, direta ou indiretamente, a interação entre elementos de software. Um padrão de composição define os requisitos que permitem a composição de elementos de software de forma a criar elementos de software maiores [Guimaraes04].

Outro conceito de componente de software, referenciado por [Klabunde99], é definido por Clemens Szyperski e foi apresentado no *1996 European Conference on Object-Oriented Programming* (ECOOP'96): "Um componente de software é uma unidade de composição com interfaces de contrato especificadas e contextos explícitos de dependências. Um componente de software pode ser desenvolvido independentemente e é o objeto de composição para outras partes" [Szyperski03].

A definição de componente reusável consiste num conjunto de produtos de software interdependentes que podem ser reusados em mais de uma aplicação [Ezran02]. Fala-se num conjunto de produtos porque o componente pode ser dividido em várias partes que, isoladas, não são passíveis de reuso. No presente trabalho, a menção de componentes, quando não indicado em contrário, se refere a componentes reusáveis.

A chegada dos componentes de software tem se mostrado o conceito mais importante na indústria de software desde o advento das linguagens de programação de alto nível. Não há nada que se possa fazer utilizando componentes que seja impossível fazer sem utilizá-los. A diferença está no conceito de reuso, tempo de desenvolvimento, qualidade e viabilidade empregados no desenvolvimento da aplicação [Szyperski03].

1.5. Evolução das técnicas de reuso

No início da ciência da computação, no período anterior à década de 60, o conceito de reuso era praticamente inexistente. Linhas de código eram analisadas e simplesmente copiadas entre programas quando necessário.

A partir da década de 60 começaram a surgir os conceitos de sub-rotinas, funções e macros. A presença de *mainframes* nas organizações exigia um aumento cada vez maior da necessidade de se otimizar o desenvolvimento de software e com isso o reuso passou a ganhar maior importância. Surgiram as bibliotecas de funções e de sub-rotinas, o código objeto (resultado da compilação de programas) passou a ser reusado por programas diferentes. Nesse tempo, que se estendeu até meados da década de 70, o mecanismo de interface entre programas eram as chamadas de funções e sub-rotinas ou diretivas de macro ou pré-processamento. Era a época anterior ao advento da computação distribuída.

No final da década de 70, início dos anos 80, acontece uma revolução com o surgimento do conceito OO, de Orientação a Objetos. O reuso ganha força com os conceitos de classes e objetos reusáveis, dissemina-se a importância do desacoplamento entre objetos através do conceito de interfaces. Surge a idéia de componentes, que possuem funcionalidades complexas e passam a se comunicar através de suas interfaces. [Facetas04]

Na década de 90, com a computação distribuída consolidada e com a disseminação da Internet em todas as áreas do conhecimento, o reuso se consolida como conceito essencial em qualquer processo de desenvolvimento de software. Nessa época as interfaces eram as RPCs (*Remote Procedure Calls*), foram elaboradas as teorias sobre *design patterns*, *frameworks* orientados a objeto, arquiteturas cliente-servidor e o que se chamou de *middleware*, como o CORBA (*Common Object Request Broker Architecture*).

Agora seria a era pós-Internet e ao lado de todos os componentes e interfaces anteriores estariam os *web-services*, arquiteturas multi-camadas,

produtos de integração de aplicações sofisticados, os chamados *application servers* como *IBM WebSphere* e *BEA WebLogic*, além da consolidação de padrões abertos como XML (*Extensible Markup Language*) e o *J2EE Connector Architecture*. Verifica-se um interesse crescente por arquiteturas B2B (*Business to Business*) e B2C (*Business to Consumer*), facilitando ainda mais a disseminação (e necessidade) do reuso, pois tais arquiteturas por definição devem ser escalonáveis, altamente confiáveis e seguras [Tracz04].

Esta evolução exerce um profundo impacto sobre a engenharia de software, pois a mudança de paradigmas do conceito de programação orientada a objetos para o conceito de programação orientada a componentes de software facilita a implementação de um projeto e a instalação da aplicação através da sua decomposição em um conjunto de elementos executáveis que podem ser independentemente projetados, construídos, executados, testados, agrupados e mantidos.

1.6. Categorias de reuso

Quando se fala em reuso, deve-se considerar que existem duas categorias básicas de reuso, conforme definidas em [McConnell96] e [Schmidt02]:

- reuso oportunístico (também chamado reuso *ad-hoc*);
- reuso planejado (também chamado reuso sistemático).

1.6.1. Reuso oportunístico

O reuso oportunístico é usado quando o desenvolvedor descobre que um sistema existente possui alguns pontos em comum com o sistema que está para ser desenvolvido. O tempo e custo do projeto são reduzidos quando o desenvolvedor então copia partes do sistema existente para o novo sistema,

técnica conhecida como “*Cut-and-Paste*” [Facetas04].

O reuso oportunístico é o que geralmente ocorre na maioria das empresas em maior ou menor escala, podendo trazer alguns ganhos, mas não necessariamente incorporado a uma política de reuso, como será tratado a seguir. É normalmente implantado de forma individual e sem maiores formalidades, como documentação, por exemplo.

A maioria das empresas que faz o reuso oportunístico apenas duplica código, num esquema caso a caso. O componente, ou o pedaço de software reusado, é compartilhado somente por um grupo pequeno de desenvolvedores e não pela empresa ou departamento [Ezran02]. Isso funciona de forma limitada a desenvolvedores individuais ou equipes pequenas, não sendo possível implantar ao longo das áreas de negócio da empresa, pois normalmente se trata de uma prática individual e aleatória ao longo dos projetos. Não necessariamente reduz o custo ou melhora a qualidade. Pode até mesmo piorar o ciclo de desenvolvimento pois à medida que os códigos copiados começam a se proliferar e a se diferenciar do código original (o que ocorre rapidamente [Ezran02]), o tempo de desenvolvimento aumenta porque os desenvolvedores são forçados a corrigir as mesmas falhas diversas vezes e em locais diferentes (a solução para as falhas encontradas no código original deve ser replicada em todos os lugares onde o código foi copiado, o que nem sempre é possível pois o rastreamento do que seja o código copiado pode se perder) [Schmidt02].

Uma boa prática é a adaptação de um sistema para atender a novos requisitos, ao invés de copiar partes de um sistema existente, o que também é classificado como reuso oportunístico [McConnell96]. A vantagem da primeira abordagem (copiar partes do sistema existente) é a necessidade de se compreender apenas pequenas partes do sistema, enquanto que na segunda abordagem (adaptar um sistema existente) é necessário um entendimento do sistema como um todo. É claro que para um mesmo desenvolvedor, ou no caso de dois sistemas similares, a escolha da adaptação será a mais indicada.

Estudos apontam que os desenvolvedores geralmente aplicam o reuso oportunístico em cima de seu próprio código, desenvolvido em outros sistemas, podendo reduzir de 15 a 35% seu esforço individual [McConnell96]. Embora haja registros de casos de sucesso através desse tipo de reuso, ele não será recomendado ao longo desse trabalho.

1.6.2. Reuso planejado

O reuso planejado, também chamado de reuso sistemático [Schmidt02] é ao que geralmente se refere quando se fala em reuso e consiste no esforço intencional e concentrado de se reutilizar componentes, arquiteturas de software, *frameworks*, *patterns*, documentos e outros artefatos de software ao longo de uma linha de produtos ou, a longo prazo, em todas as áreas da empresa. Nessa categoria de reuso, a produtividade e qualidade do software aumentam ao mesmo tempo em que o custo do ciclo de desenvolvimento diminui, pois são eliminadas as etapas de re-*design*, re-codificação e re-teste de componentes comuns.

Trata-se do programa de reuso oficial a ser adotado pela empresa, a longo prazo é a categoria de reuso capaz de trazer os maiores benefícios em termos de redução de custos e tempo de projeto e será o objeto de estudo nesse trabalho. É importante enfatizar que, como é uma estratégia de longo prazo, os benefícios não serão vistos no primeiro projeto.

Capítulo 2

Componentes reusáveis

No capítulo 1 foram apresentadas algumas definições de componentes. Na literatura especializada existem dezenas de definições e algumas são mais restritas que outras. Grady Booch considera um componente qualquer elemento de software que pode ser empacotado, tais como bibliotecas, arquivos, códigos executáveis e documentos [Booch98]. Essa é uma definição mais genérica, mas que se aplica perfeitamente ao conceito de componentes reusáveis.

Um conceito fundamental de componente é que ele possui interfaces claramente definidas. Interface é uma abstração do comportamento de um componente que consiste de um subconjunto de interações deste componente, associado a um conjunto de restrições que descreve quando elas podem ocorrer. De forma a permitir que um componente seja reutilizado e interaja com outros componentes é necessário que os componentes provedores e clientes estejam de acordo a respeito de um conjunto de interfaces previamente definidas. Basicamente, a habilidade para integrar e armazenar componentes e, a partir desta integração, disponibilizar estes componentes depende fundamentalmente da noção de interface do componente [Guimaraes04]. As interfaces são definidas principalmente como uma coleção de pontos de acesso a serviços, ou *Service Access Points*, cada uma contendo uma especificação semântica [Bosch97]. Um componente não está sozinho em um ambiente e o seu propósito é proporcionar algum serviço para tal ambiente. O acesso aos serviços do componente é visível e disponibilizado a usuários da aplicação, a outros componentes, às plataformas de

suporte e a outros elementos de software, através destes pontos. Geralmente, um componente possui múltiplas interfaces correspondendo a diferentes pontos de acesso.

Para se alcançar efetivamente o reuso do componente é importante que exista uma independência do componente em si da tecnologia de implementação utilizada. Dessa forma pode-se afirmar que o componente é composto de duas camadas, a sua interface e sua implementação. No caso do reuso, é importante ressaltar que existem dois ciclos de vida distintos, um para cada camada: o ciclo de vida da interface e o ciclo de vida da implementação (do próprio componente, por assim dizer). O ciclo de vida da interface é maior que o ciclo de vida da implementação, pois as interfaces existem enquanto houver implementações disponíveis para o seu componente associado [Bosch97]. Em outras palavras, a interface continua existindo enquanto que uma implementação acaba e é substituída por outra, usando uma tecnologia diferente, por exemplo.

Dentro da empresa, cada componente reusável deve ser considerado como um produto completo de software e deve passar por todas as etapas de criação de um produto de software, a saber:

- definição do produto: a necessidade do componente deve ser avaliada antes de sua criação, bem como seu custo avaliado em relação ao benefício proporcionado pelo reuso;

- construção: depois de criado, o componente deve ser testado, documentado e empacotado a fim de se tornar reusável. O termo “empacotado” pode se referir também à definição dos limites do componente. No caso do componente se constituir de apenas um artefato, o pacote se traduz no próprio componente;

- propaganda: o componente deve ser divulgado a seu público de interesse. Os usuários devem estar cientes de sua existência e devem ser convencidos de sua utilidade;

- pré-venda: a assistência durante a fase de avaliação do componente deve ser prevista e fornecida de forma que o usuário utilize o componente corretamente e não perca tempo tentando compreender o seu funcionamento;

- manutenção e suporte: não é tarefa do usuário corrigir falhas ou fazer melhorias no componente. Essas tarefas são de responsabilidade do criador ou mantenedor do componente e o usuário deve ser informado quando novas versões se encontram disponíveis e, nesse caso, ser informado também sobre as diferenças da nova versão em relação à versão anterior;

- *feedback*: a opinião dos usuários deve ser coletada a fim de melhorar o componente e, dependendo do ramo de negócio e função do componente, sustentar a vantagem competitiva da empresa frente aos concorrentes.

2.1. Classificação dos componentes

Os componentes podem ser classificados em dois tipos distintos [Ezran02]:

- componentes verticais;

- componentes horizontais.

2.1.1. Componentes verticais

Os componentes verticais são específicos a um domínio de aplicação. São os componentes mais valiosos pois englobam as regras de negócio embutidas nas aplicações da empresa e com isso representam o *know-how* da empresa e sua vantagem competitiva frente à concorrência. Exemplos de componentes verticais:

- modelos de objetos financeiros;
- bibliotecas C++ para instrumentos médicos;
- *design* e código para módulo de gerenciamento de clientes;
- algoritmo de cálculo para geoprocessamento;
- modelo de objetos no domínio de um sistema de trafego aéreo;
- um *framework* para aplicações de telemetria.

2.1.2. Componentes horizontais

Os componentes horizontais são genéricos, mais fáceis de serem identificados e reusados, pois são independentes do domínio da aplicação. A maior restrição desse tipo de componente é que a arquitetura da aplicação deve ser compatível com o modelo arquitetural do componente. Exemplos de componentes horizontais

- componentes de GUI – *Graphical User Interface* (tabelas, botões, campos de texto);
- estruturas de dados e algoritmos básicos (como ordenação, por exemplo);
- bibliotecas de acesso a bancos de dados;
- bibliotecas de comunicação de redes;
- serviços de relatórios;
- serviços de *trace* e *logs* para depuração de programas;
- *frameworks* de gerenciamento de falhas.

2.1.3. Considerações na classificação dos componentes

Deve-se observar que a classificação dos componentes pode ser subjetiva. Um componente pode ser considerado vertical dentro de um domínio de aplicação, mas se esse domínio é dividido em sub-domínios, o componente pode ser compartilhado entre esses domínios, tornando-se então um componente horizontal. Isso ocorre também porque os domínios podem ser aninhados, isto é, pode-se definir domínios muito específicos (exemplo: gerenciamento de desempenho de redes) que pertençam a domínios mais genéricos (exemplo: gerenciamento de redes). Para evitar essa ambigüidade na classificação do componente, os domínios de aplicação devem ser definidos da forma mais precisa possível.

A classificação correta do tipo e domínio do componente é importante na busca e análise do componente, quando este se encontra disponível num repositório de componentes.

2.2. Granularidade dos componentes

Os componentes necessitam ser construídos com características de modularidade, onde são analisadas as propriedades do componente e as interfaces que devem ser construídas em conjunto, devendo assim, serem projetados e desenvolvidos de forma independente uns dos outros para então serem combinados. Estas construções generalizadas de componentes são apenas válidas se puderem ser reusadas em outras aplicações e a granularidade do componente influencia nisto [Szyperski03]. Dessa forma, é importante que o fator de granularidade dos componentes seja observado durante o seu desenvolvimento.

A granularidade adequada de um componente depende de aspectos associados ao particionamento do sistema para o qual ele se destina. Essas partições podem possuir um tamanho variável, mas que seja coerente com o

propósito da aplicação. Exemplos:

- uma função ou procedimento;
- uma classe;
- um grupo de classes;
- um subsistema, *framework* ou serviço;
- uma aplicação ou produto.

Os componentes são geralmente compostos, isto é, um componente contém outros componentes reusáveis, que também podem ser reusados de forma independente, a critério do usuário. É importante salientar que quanto maior o componente, mais cedo ele deve ser levado em consideração no processo de desenvolvimento do produto [Ezran02]. Além disso, Clemens Szyperski menciona alguns casos em que a granularidade do componente influencia no desenvolvimento de novos componentes ou aplicações [Szyperski03]:

- abstração: classificada em *whitebox* ou *blackbox*, como será apresentado adiante, possui grande influência no desenvolvimento do componente e é através da granularidade que se determinará o tipo de abstração mais adequado a ser utilizado para este componente;
- custo: em um desenvolvimento de sistemas grandes e complexos a granularidade dos componentes possibilita a visualização e planejamento do seu custo e tempo de desenvolvimento. Caso a granularidade seja muito baixa, o custo será maior que o previsto e caso seja muito alta, indicará que não foi precisa o suficiente para traçar os custos exatos;
- complexidade do sistema: em unidades de análise, uma boa granularidade dos componentes permite maior confiança na construção e compreensão de um sistema complexo. Uma granularidade muito alta não permite que detalhes do sistema sejam entendidos, enquanto que uma granularidade

muito baixa pode expor muitos detalhes do sistema, aumentando a complexidade do seu entendimento.

2.3. Modelos de componentes reusáveis

O componente é feito a partir de vários produtos que surgem e que são usados durante o seu ciclo de vida, a saber:

- definição de requisitos;
- definição de arquitetura;
- modelo de análise;
- modelo de design e código;
- programas de teste;
- cenários de teste;
- relatórios de teste.

Esses produtos podem representar o mesmo pedaço de software em diferentes camadas de abstração (definição de requisitos, análise, *design*, codificação e testes) e com isso cada um pode ser reusado nas diversas etapas que compõem o ciclo de desenvolvimento do produto [Ezran02]. Exemplo: programas e procedimentos de teste são altamente reusáveis. Uma empresa de telecomunicações pode reusar extensivamente os mesmos procedimentos de teste para validar a conformidade de diferentes equipamentos segundo diferentes normas técnicas.

Considerando os tipos de componentes, seus modelos e sua granularidade, e baseando-se no desenvolvimento de alguns de seus projetos de software, Michel Ezran listou alguns fatores de sucesso relacionados ao tipo de

componente a considerar no desenvolvimento do produto [Ezran02]:

- considerar componentes horizontais e verticais. Componentes horizontais são mais fáceis de se definir, mas os componentes verticais são mais críticos às regras de negócio da aplicação;

- usar componentes de alta granularidade (aplicações, *frameworks*, serviços) ao invés de componentes de baixa granularidade (classes, funções), pois são mais eficientes e trazem um ganho maior em termos de custo no desenvolvimento do produto;

- procurar modelar a arquitetura da aplicação de acordo com a arquitetura dos componentes. O reuso de componentes de alta granularidade também favorece a definição de uma arquitetura comum entre as aplicações;

- um grande número de pequenos componentes é mais difícil de gerenciar. É preferível gerenciar apenas um pequeno número de componentes que sejam críticos aos negócios da empresa;

- não limitar o reuso a simplesmente reuso de código. O código representa menos de 25% do custo de desenvolvimento do produto. Isso pode parecer estranho mas o alto custo de desenvolvimento de código muitas vezes é conseqüência de uma fase anterior deficiente (problemas na fase de análise de requisitos ou de *design*, por exemplo).

2.4. Características de componentes reusáveis

Um componente cuja função se espera ser definida como reusável deve obedecer a alguns critérios como:

- comprometimento com padrões, *guidelines*, codificação, documentação e outras regras de desenvolvimento aplicáveis;

- comprometimento com métodos de *design*, testes e processos de validação;
- possuir uma descrição detalhada de seu funcionamento;
- ser simples e fácil de entender;
- ser modular, de forma a se adaptar facilmente a outros componentes;
- não ser nem muito especifico (menos reusável) nem muito genérico (menos valioso).

Além das características acima, a fim de serem facilmente reusados e integrados a outros componentes, existem características técnicas que também devem ser observadas nos componentes reusáveis: interoperabilidade, portabilidade, separação entre interface e implementação, composição, autodescrição, transparência na execução, segurança e *plug and play* [Ezran02]. Essas características são descritas a seguir.

2.4.1. Interoperabilidade

É a habilidade do componente em se comunicar com outros componentes, implementados em outras linguagens de programação ou executando em diferentes plataformas de hardware, de forma a facilitar a sua integração em outros produtos. Aqui se aplica também o conceito de parametrização das interfaces do componente, que facilita a sua interoperabilidade com outros componentes.

2.4.2. Portabilidade

Consiste na habilidade do componente ser executado em diferentes plataformas de hardware ou software. Em geral, componentes não devem ser

dependentes de implementações ligadas a plataformas específicas, como bancos de dados ou interfaces gráficas. Para isso uma camada lógica para esses serviços deve ser fornecida para assegurar a portabilidade do componente. Essa característica é relativa, pois não se aplica no caso do componente ser desenvolvido para uma plataforma específica (biblioteca GUI para sistema operacional Windows, por exemplo).

2.4.3. Separação entre interface e implementação

A forma como o componente é reusado e a forma como ele é implementado são duas coisas distintas que não podem ser misturadas, pois elas podem evoluir separadamente. Mudanças na implementação não devem implicar mudanças na forma de reusar o componente.

2.4.4. Composição

Pequenos componentes reusáveis podem ser agrupados para formar um grande componente, também reusável, sem perder suas características de reusabilidade.

2.4.5. Autodescrição

O componente deve ser autodescritivo a ponto de poder ser manipulado como uma caixa-preta, onde o usuário precisa compreender apenas suas interfaces, sem a necessidade de saber detalhes de sua implementação.

2.4.6. Transparência na execução

O componente deve poder ser executado em diferentes máquinas sem

afetar o restante do sistema onde ele estiver sendo executado.

2.4.7. Segurança

Como o componente pode conter código executável e ser compartilhado como uma caixa-preta junto a uma comunidade de usuários desconhecidos, a autenticidade e origem do componente deve poder ser comprovada através do uso de chaves digitais ou outro sistema de segurança. O componente deve também possuir mecanismos para que o usuário controle o acesso do componente a seus recursos particulares. Exemplo: um componente *web* não pode apagar arquivos da máquina do usuário sem o seu consentimento.

2.4.8. Plug and play

A portabilidade do componente não necessariamente precisa ser verificada em tempo de construção, mas em tempo de execução do componente. Exemplo: um componente escrito em código Java deve poder ser executado em diferentes plataformas. Nesse caso, a implementação em linguagem Java assegura a portabilidade em tempo de construção, por ser uma característica da linguagem, mas em tempo de execução a portabilidade deve ser verificada, pois a execução depende da máquina virtual Java disponível em cada plataforma.

2.5. Modelo 3-C

O reuso de componentes numa organização freqüentemente tem sido discutido no contexto do esforço em se desenvolver componentes de software discretos que possam ser usados de uma forma no estilo *blackbox* (como será visto adiante) em configurações externas às quais esses componentes foram inicialmente concebidos [Norton03]. Esse objetivo não é trivial. Will Tracz elaborou o modelo 3-C, identificando três critérios para a avaliação da reusabilidade de um

componente [Tracz90]:

- **Conceito:** a abstração que o componente representa;
- **Conteúdo:** a implementação do componente;
- **Contexto:** o que é necessário para complementar a definição do componente em um certo ambiente.

Desses três aspectos, Will Tracz identifica o Contexto como sendo o mais importante e também o mais subestimado. Se um componente *black-box* de uma organização possui dependências não documentadas, é de se esperar claramente que ele irá falhar quando integrado num ambiente que não satisfaça tais dependências. A documentação e descrição de componentes reusáveis passa então a ser tão crítica quanto complexa [Tracz90].

A facilidade de identificação, compreensão e confiança são fatores importantes no sucesso de reuso de um componente, que devem ser observados em seu projeto e desenvolvimento [Kotula98].

A indústria tem se esforçado em solucionar essas questões relacionadas ao reuso. A antiga *Rational Corporation* [Rational], hoje uma divisão de software da empresa IBM, adotou uma iniciativa de reuso conhecida como *Rational Development Accelerators*, que incorpora padrões de reuso, *frameworks* para reuso de software e automação nos processos de catálogo e acesso a componentes reusáveis. O aspecto principal dessa iniciativa é a *Rational Corporation's Reusable Asset Specification* (RAS), uma coleção de modelos e padrões para a descrição, desenvolvimento e aplicação de diferentes tipos de componentes de software reusáveis [RAS]. De acordo com o RAS, um pré-requisito fundamental para o reuso é um acordo industrial sobre um modo comum de se descrever os componentes de software reusáveis.

Outras iniciativas compreendem o DCMI (*The Dublin Core Metadata Initiative*) [DCMI] e o SCORM (*The Sharable Content Object Reference Model*)

uma initiativa do grupo ADL (*Advanced Distributed Learning*) [ADL].

Deve-se considerar, porém, a escalabilidade e abrangência do reuso sendo abordado. O desenvolvedor de um pequeno projeto não necessita necessariamente de um modelo detalhado como o RAS para alcançar efetivamente o reuso. O tempo despendido para documentar um componente através do RAS pode custar algumas centenas de dólares, mas se esse tempo investido permitir encontrar e reusar várias vezes o componente mais tarde, então o investimento de centenas pode economizar milhares de dólares. Essa economia de escala, porém, não está presente no caso de desenvolvedores isolados, pois haverá prejuízo se o tempo necessário para documentar um componente for maior que o tempo para desenvolvê-lo completamente [Norton03].

2.6. Aquisição de componentes

A empresa ou a equipe de desenvolvimento de software pode considerar as seguintes opções se um componente for necessário mas ainda não existir no repositório de componentes utilizado pela empresa:

- comprar o componente, se um produto comercial já existir [Guerra04];
- procurar por componentes de domínio público na Internet ou em parceiros comerciais;
- desenvolver o componente;
- subcontratar uma empresa de software terceirizada para o desenvolvimento do componente.

A aquisição de um aplicativo pronto é a única forma garantida de reuso, permitindo que a empresa obtenha a funcionalidade desejada por um preço menor do que seria desenvolvê-la internamente [Ezran02]. Entretanto o aplicativo adquirido pode não oferecer a funcionalidade exata necessária para implantá-lo

em seu projeto, requerendo algum tipo de adaptação que irá aumentar o custo do produto. Tecnicamente não há razão alguma para desenvolver o componente internamente se é possível adquiri-lo de outra empresa por um custo equivalente (ou menor), mas podem existir razões políticas e de negócio envolvidas no desenvolvimento, já que a empresa pode querer garantir o domínio da tecnologia, que pode ser crítica a seu ramo de negócios [McConnell96].

2.7. Evolução dos componentes

A evolução do desenvolvimento de componentes foi descrita pela Oracle Corporation no trabalho de Umesh Bellur e estabelece uma relação entre o nível de abstração dos componentes com o seu potencial de reuso, de forma a evidenciar o aumento da produtividade do desenvolvedor com a aplicação do reuso [Bellur97]. Essa evolução é apresentada na Figura 2-1.

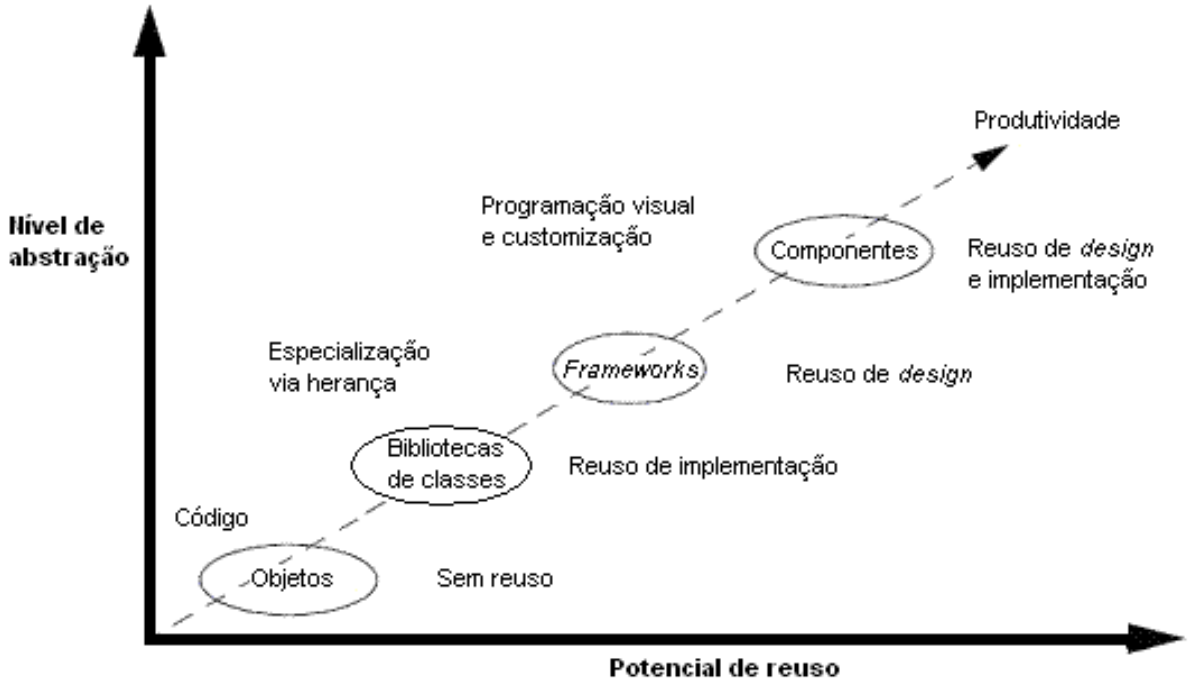

Figura 2-1– Evolução dos componentes e potencial de reuso

O nível de abstração reflete a evolução dos componentes conforme o

desenvolvimento de novas tecnologias, partindo desde o código puro até o advento da programação visual através das linguagens de alto nível. O potencial de reuso aumenta conforme a tecnologia empregada e o seu nível de abstração. Pela figura pode-se verificar que o potencial de reuso de um *framework* é maior que o potencial de reuso de um objeto. À medida que o nível de abstração aumenta (permitido devido ao uso de *frameworks*, por exemplo), a produtividade do desenvolvedor também aumenta devido ao aumento do potencial de reuso.

2.8. Documentação do Componente

Para utilizar corretamente um componente da maneira como que ele foi planejado, o usuário deve compreender suas características de desempenho, seqüência de chamada de funções e comportamento. Essas informações devem estar disponíveis na documentação do componente que, infelizmente, em muitos casos pode ser vaga, complexa ou até mesmo inútil, dependento de como e quais informações são documentadas [Kotula98].

Quando os componentes são projetados e implementados, o desenvolvedor certamente necessita utilizar diagramas de cenários e uma série de casos de uso [Kotula98]. Essas informações necessitam ser repassadas ao usuário do componente. Quando um usuário não compreende o propósito de um componente e para que o seu desenvolvedor o criou, ele muitas vezes irá utilizar o componente de forma incorreta criando falhas em seu sistema. No pior caso, o usuário ficará insatisfeito e irá descartar o componente.

A documentação é a melhor maneira do autor do componente informar de maneira eficiente para o usuário a transferência de conhecimentos sobre o componente. Esta é uma das mais importantes formas de melhorar a compreensão do software e reduzir seus custos de manutenção e desenvolvimento.

Apesar dos projetistas de componentes esforçarem-se em criar

interfaces fáceis e intuitivas de usar, alguns detalhes importantes continuam tendo que ser comunicados ao usuário. Ler o código para compreender o componente é muitas vezes cansativo e nem sempre o código se encontra disponível. Desta forma, sob o ponto de vista do usuário, a documentação é mais importante que a programação. A ótima implementação de um componente não usável por falta de documentação é inútil, apesar do componente ser de ótima qualidade. A importância da documentação como facilitadora do reuso reside então no fato que usuários possuem a expectativa de usar os componentes de forma simples e fácil e ficam rapidamente frustrados quando não o conseguem ou quando a maneira explanada para usá-los não é clara.

Especificações rigorosas para o desenvolvimento de componentes são recomendadas. Técnicas de orientação a objetos e linguagens de modelagem auxiliam na documentação dos componentes como, por exemplo, os diagramas de estados utilizando-se a notação da UML (*Unified Modeling Language*) e as documentações das pré e pós-condições que possam ocorrer durante a utilização do componente [Tracz95].

2.9. Processo de criação de componentes

A Figura 2-2, apresentada no trabalho de Martin Büchi, ilustra as três fases do processo de criação de componentes, omitindo os ciclos de *feedback*, que proporcionam a evolução dos padrões e de componentes [Büchi]. Na primeira fase, os fabricantes se baseiam nas interfaces padronizadas existentes no mercado para construir seus componentes. Se padrões não existirem, diferentes padrões de interfaces são desenvolvidos e descritos em público (em publicações especializadas ou instituições normativas, por exemplo). Na segunda fase, os fabricantes selecionam os padrões a serem utilizados em seus componentes, desenvolvem os componentes e os colocam no mercado. Finalmente na terceira fase os desenvolvedores (usuários) selecionam, adquirem e interconectam os componentes existentes no mercado para construir sistemas mais complexos.

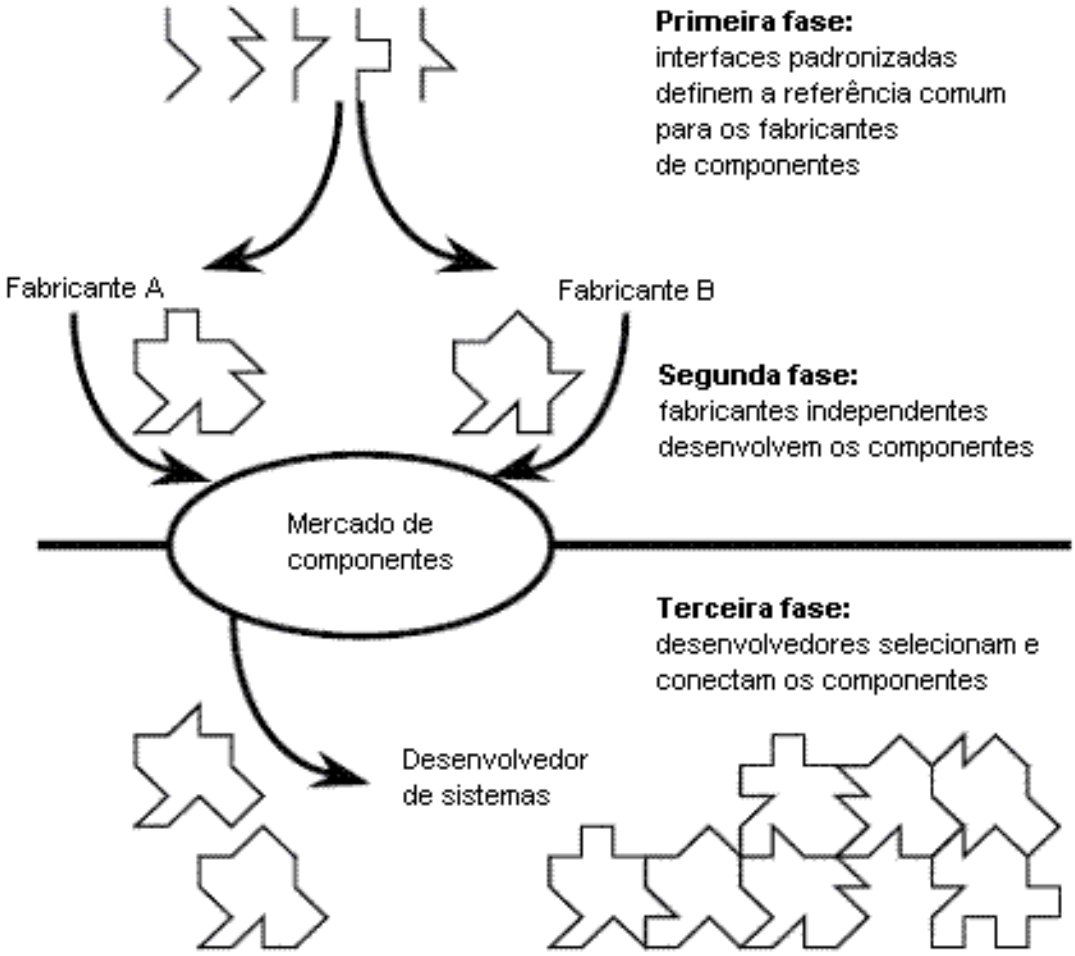

Figura 2-2 - As três fases da programação orientada a componentes

Não se pode esperar que os fabricantes de componentes sincronizem o trabalho de criação dos componentes entre si, mas eles podem criar os componentes baseados em padrões já existentes. Se os padrões forem definidos corretamente, os componentes criados de forma independente irão operar de forma *plug-and-play*. Não é suficiente, entretanto, usar apenas um padrão comum de conexão, como COM (*Component* Object *Model*), CORBA (*Common Object Request Broker Architecture*), ou uma linguagem de programação específica, como Java. Esses padrões definem apenas as convenções de chamadas para funções ou sub-rotinas [Kirtland00]. O *plug-and-play* de componentes requer ainda interfaces padrões dependentes do domínio da aplicação, especificadas tanto sintaticamente quanto semanticamente [Büchi]. Nesse caso, a especificação

sintática se refere à descrição técnica da interface. Exemplo: a função deve receber um número inteiro variando de 0 a 10 e uma cadeia de caracteres com o último caracter sendo igual ao caracter nulo. A especificação semântica se refere à descrição do contexto em que a função deverá ser chamada. Exemplo: a função deve receber o saldo e o nome do correntista após todas as transações pendentes terem sido processadas.

Capítulo 3

Tecnologias de reuso

A tecnologia de reutilização procura disponibilizar componentes que possam ser facilmente conectados para elaborar novos sistemas. Contudo, não se pode simplesmente selecionar componentes de catálogos, conectar esses componentes e implementar um sistema de software. A disciplina de inter-relacionamento de componentes é um dos problemas mais difíceis de serem resolvidos pela engenharia de software nos dias de hoje. Questões de como interações abstratas de componentes podem ser descritas; como o desempenho de sistemas pode ser garantido, como a flexibilidade e variedade de requisitos podem ser cobertas e como propriedades de sistemas críticos podem estar garantidos têm que ser cuidadosamente estudadas [Szyperski03].

Muitas técnicas têm sido desenvolvidas e propostas para atingir o objetivo de maximizar o reuso de componentes básicos, de produtos de *design,* de arquiteturas e até mesmo de aproveitar a experiência dos desenvolvedores na solução de problemas em contextos específicos. Todas essas técnicas oferecem visões e soluções parciais para o problema do desenvolvimento de software. Muitas vezes as técnicas se sobrepõem umas às outras e, individualmente, suportam apenas uma ou algumas fases do ciclo de desenvolvimento de software. São apresentadas a seguir as técnicas mais utilizadas no desenvolvimento voltado ao reuso.

3.1. Técnicas OO

As técnicas orientadas a objeto (OOT) provêem métodos e mecanismos para a estruturação de modelos e programação de código correspondente aos objetos (conceitos) encontrados no domínio do problema. Por exemplo, no domínio de uma aplicação de negócios, objetos podem descrever uma conta de banco ou um cliente através de atributos (número da conta, por exemplo), comportamentos (extratos de contas) e relacionamentos com outros objetos (a conta pertence a um cliente).

OOT é uma técnica que influencia positivamente o reuso, mas não garante que o reuso será alcançado. Há muitas razões (organizacionais, culturais e políticas) que determinam porque um excelente componente de software pode correr o risco de nunca ser reusado. Comparado com a programação estruturada tradicional, as técnicas OO têm a vantagem de possibilitar o desenvolvimento de sistemas altamente modulares, usando os princípios de abstração de dados e encapsulamento de informações. A aplicação deixa de ser vista como uma única função complexa que necessita ser decomposta em sub-rotinas, seguindo uma abordagem *top-down*. Ao invés disso, o desenvolvedor identifica os principais objetos no domínio da aplicação, suas responsabilidades e relacionamentos e desenvolve a aplicação tendo em vista quais objetos poderiam ser reusados, transformando esses objetos em componentes reusáveis.

As técnicas OO compreendem a análise (OOA), *design* (OOD) e implementação (OOP) de componentes [Booch93]. Muitos métodos OOA e OOD têm sido propostos na literatura, como Booch [Booch93] e OMT (Object Modeling Technique) [Rumbaugh91], mas a partir de 1996 a UML tem sido indicada como a metodologia padrão na modelagem de sistemas orientados a objetos. Na verdade a UML originou-se a partir da fusão dos métodos Booch, OMT e OOSE (*Object-Oriented Software Engineering*) devido à similaridade de conceitos existente entre eles [Braun]. A UML é uma linguagem para especificar, visualizar, construir e documentar os artefatos de sistemas de software [Ezran02]. O mercado tem uma

grande variedade de ferramentas para ajudar os analistas e desenvolvedores de software a capturarem as decisões de *design* numa forma gráfica. A mais conhecida delas é a ferramenta *Rational Rose* [Rose], que utiliza a UML como linguagem para descrição do *design*.

A base do OOP é o conceito de objetos como unidades de programa que encapsulam dados e algoritmos. Esse conceito distingue o paradigma da programação orientada a objetos do paradigma da programação puramente estruturada, onde o código que acessa e modifica uma dada informação fica confinado a um único ponto dentro do programa, e não espalhado de forma descontrolada por todo o programa. Objetos com propriedades e funções similares são agrupados em classes.

OOP suporta o reuso na forma de bibliotecas de classes, as quais englobam técnicas de encapsulamento, herança e polimorfismo. O desenvolvedor constrói a aplicação através do reuso de classes, que pode ocorrer de duas formas:

- construindo-se classes novas utilizando objetos de classes já existentes (na biblioteca de classes);
- derivando-se novas classes, especializando as já existentes.

A orientação a objetos não é nem necessária nem suficiente para a prática do reuso e também não garante um bom nível de reuso. De fato, a abordagem de bibliotecas de classes não têm sido muito mais bem sucedida no alcance do reuso do que as bibliotecas de funções das linguagens estruturadas. A principal deficiência das bibliotecas de classes é que elas não suportam reuso planejado de *design* e arquitetura. Devido a sua baixa granularidade elas são mais bem sucedidas no reuso no caso de implementações orientadas a objetos de estruturas básicas de dados para sistemas mais complexos [Ezran02]. Contudo, a separação dos objetos em classes facilita a implementação do reuso, pois induz os desenvolvedores a pensarem em arquiteturas 3-camadas, o que facilita a

identificação, implementação e integração dos componentes de reuso. A arquitetura 3-camadas consiste num modelo de implementação de software onde o produto de software é divido em três camadas, cada uma com funções específicas e independentes entre si: a camada de apresentação (ou interface com o usuário), a camada de aplicação (onde se encontram as regras de negócio do software) e a camada de dados (responsável por armazenar os dados utilizados).

A reutilização de componentes de software em larga escala é um dos argumentos a favor da abordagem de orientação a objetos. Em muitos casos, porém, constitui uma perspectiva frustrada, pois a reutilização não é característica inerente da orientação a objetos, mas é obtida a partir do uso de técnicas que produzam software reutilizável [Silva00].

3.2. Design Patterns

Os *design patterns* são uma tentativa de superar as limitações do simples reuso de código adotado pela abordagem das bibliotecas de classes através da ênfase na importância do reuso de *design* [Gamma94]. Essa técnica apareceu no mundo orientado a objetos como uma técnica para documentar soluções de *design*. A definição mais simples e genérica é apresentada por Erich Gamma, que define que os *design patterns* são soluções documentadas para problemas comuns num determinado contexto [Gamma94].

A apresentação de um *design pattern,* em sua forma literária, descreve o contexto de uma aplicação, um problema de *design* que ocorre naquele contexto e uma solução para o problema (que tenha sido provada por experiência que seja uma solução eficaz). Alguns autores estendem a descrição de seus *patterns* através da inclusão de outros itens como: as vantagens do *pattern*; exemplos de como o *pattern* pode ser aplicado em casos específicos; e técnicas para a implementação do *pattern* (ou a própria implementação do *pattern* para um dado

problema numa determinada linguagem) [Ezran02]. Dessa definição pode-se extrair as seis seções básicas que compõem a descrição de um *pattern*: Contexto, Problema, Solução, Vantagens, Implementação e Exemplo.

Atualmente existem várias bibliotecas de *design patterns* e a mais conhecida é proposta por Erich Gamma [Gamma94]. O desenvolvedor seleciona o *design pattern* que é mais aplicável ao problema em questão e reusa a solução correspondente. As seções Contexto e Problema da descrição do *pattern* facilitam para o usuário a escolha do *pattern* correto. Uma coleção estruturada de *design patterns* dedicada a um domínio específico da aplicação é chamada de *pattern language*. A *pattern language* é representada então pelas dependências entre cada *pattern* [Gamma94].

Outra vantagem dos *patterns* é facilitar a comunicação entre diversos desenvolvedores. Cada *pattern* possui um nome único, que passa a fazer parte do vocabulário dos desenvolvedores, permitindo que a descrição de problemas complexos e soluções para esses problemas sejam referenciados através de uma única palavra [Ezran02]. O Apêndice A - *Observer pattern* - apresenta a definição de um *pattern* conforme o modelo descrito por Erich Gamma e serve de referência para a documentação de novos *patterns*. [Gamma94]

Uma desvantagem é que geralmente a implementação de um *design pattern* não é fornecida junto a sua documentação e o mesmo *pattern* pode ser implementado de formas diferentes em diferentes partes da aplicação, por desenvolvedores diferentes. Os *design patterns* e *pattern languages* funcionam melhor quando a aplicação precisa ser implementada a partir do início.

3.3. Application frameworks

Um *framework* é um conjunto integrado de artefatos de software reusáveis e extensíveis para um domínio específico da aplicação [ACM97] e que podem ser especializados para atender a uma aplicação em particular

[Klabunde00]. Os *frameworks* são antigos, datando do final da década de 80, mas apenas recentemente vêm sendo amplamente empregados devido à sua integração com os *design patterns*.

Os *frameworks* orientados a objetos são uma tentativa de se combinar reuso de código (bibliotecas de classes) com reuso de *design* (*design patterns*) [Ezran02]. O primeiro consiste na utilização direta de trechos de código já desenvolvidos. O segundo consiste no reaproveitamento de concepções arquitetônicas de um artefato de software em outro, não necessariamente com a utilização da mesma implementação [Silva00].

Aplicações concretas são criadas através da customização do *framework*, que pode ser feita através da aplicação de conceitos classificados como *blackbox* ou *whitebox*. Freqüentemente os dois conceitos se encontram disponíveis no mesmo *framework* [Ezran02]. Esses conceitos se referem à visibilidade da implementação de um componente através de sua interface [Klabunde99].

- *Blackbox* (caixa preta): o *framework* provê implementações específicas para cada aspecto variável da aplicação. O usuário gera a aplicação selecionando e conectando os componentes [Ezran02]. Nenhum detalhe além da interface e de sua especificação é conhecido pelo usuário [Klabunde99].

- *Whitebox* (caixa branca): o framework provê implementações genéricas para cada aspecto variável da aplicação. O usuário então especializa cada componente para a aplicação concreta. Esse tipo de customização é mais flexível, mas requer um entendimento mais profundo do *design* e da implementação do *framework* [Ezran02]. Nesse caso a interface pode retratar encapsulamento e limitar quais os componentes que podem, através de implementação de herança, serem autorizados a realizar alterações na implementação genérica [Klabunde99].

Segundo estudos de Michel Ezran, poucas empresas empregam o conceito *blackbox* e nenhuma adotou exclusivamente esse modelo, devido ao fato que o conceito *whitebox* é mais flexível para o cliente do reuso, mais fácil de implementar, demanda menos esforços em termos de conhecimentos técnicos e possibilita um maior número de artefatos reusáveis [Ezran02]. A desvantagem do conceito *whitebox* é que em alguns casos o componente não pode ser substituído por um novo componente *whitebox*, pois as alterações realizadas na versão anterior irão fazer com que a inclusão da nova versão cause conflitos com o atual software da aplicação, tornando este inconsistente [Szyperski03].

Já o conceito *blackbox* traz consigo a vantagem da interligação de componentes independentemente da linguagem de programação, localização física e plataforma de execução. Outra vantagem é a fácil substituição de versões anteriores de componentes por versões mais recentes, ou de outros fabricantes, pelo fato de suas interfaces permanecerem as mesmas [Szyperski03].

Existe uma divisão do conceito *whitebox* em *glassbox*, onde *whitebox* permite manipular a implementação e *glassbox* permite apenas estudar a implementação [Klabunde99]. O conceito *glassbox* é aquele que contém e disponibiliza para visualização apenas parte da implementação que é importante para controlar o componente ou *framework*, contudo essa parte está muitas vezes definida na própria especificação do *framework*, onde desta forma esse conceito seria o mesmo de um *blackbox* [Szyperski03].

Em contraste com a maioria das técnicas de reuso, como *design patterns* e bibliotecas de classes, os *frameworks* de aplicações consistem de código reusável e *design* reusável. Isso significa que aplicações que são construídas baseadas no *framework* nunca são planejadas a partir do início, pois o *design* do *framework* é usado como ponto de partida para o *design* da aplicação.

A construção de um *framework* reusável nem sempre é fácil, pois requer um entendimento profundo do domínio da aplicação para o qual o *framework* é proposto em termos de entidades e relacionamentos que podem ser

definidos e reusados. Quanto mais detalhados os artefatos de software, mais específicas as suas interações se tornam e mais difícil se torna atingir os requisitos de futuras aplicações de forma a alcançar a reusabilidade. A flexibilidade nos padrões de interação entre os componentes é obrigatória para garantir a reusabilidade do *framework*. Por isso os *frameworks* são geralmente construídos baseados em múltiplos *design patterns* [Ezran02].

Os *frameworks* procuram resolver e fazer entender problemas muitas vezes complexos e isto faz com que sua estrutura seja grande e também complexa, levando muitos desenvolvedores a não utilizá-los. Procurando facilitar o desenvolvimento a partir de *frameworks*, estes são subdivididos em pequenas partes independentes denominadas *framelets* [Klabunde00]. O desenvolvimento de uma aplicação é feito então através da composição de *framelets*, tornando possível a adição de particularidades da aplicação e possibilitando o reuso de um número maior de artefatos.

Em seu trabalho, Ricardo Silva mostra que a diferença fundamental entre um *framework* e a reutilização de classes de uma biblioteca é que neste último são usados componentes de software isolados, cabendo ao desenvolvedor estabelecer a sua interligação, enquanto que no caso do framework é feita a reutilização de um conjunto de classes inter-relacionadas, inter-relacionamento este estabelecido no projeto do *framework* [Silva00]. As figuras 3-1, 3-2 e 3-3 ilustram essa diferença. As áreas sombreadas representam as classes e associações que são reutilizadas.

Um dos mais conhecidos *frameworks* de aplicação é o CORBA, um *framework* para a distribuição, interconexão e comunicação de componentes através de uma rede de sistemas distribuídos, independente de plataformas de hardware e de linguagens de programação. O padrão CORBA é definido pelo OMG (*Object Management Group*), um consórcio formado por mais de 700 empresas, incluindo líderes de mercado como IBM, Microsoft, Sun e HP.

Em termos ideais, um *framework* deve abranger todos os conceitos

gerais de um domínio de aplicação, deixando apenas aspectos particulares para serem definidos nas aplicações específicas. No caso ideal, na geração de aplicações, o usuário do *framework* não precisa criar classes que não sejam subclasses de classes abstratas do *framework*. Se isto for alcançado, o *framework* terá conseguido de fato ser uma generalização do domínio modelado [Silva00].

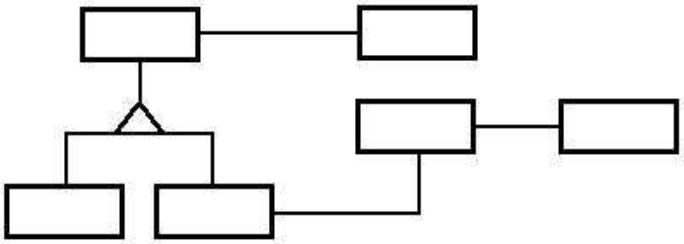

Figura 3-1 - Aplicação desenvolvida totalmente

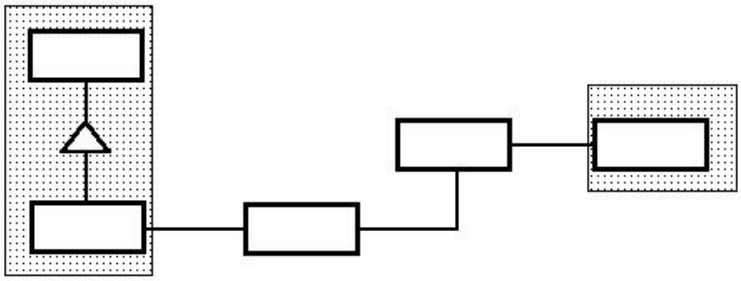

Figura 3-2 - Aplicação desenvolvida reutilizando classes de biblioteca

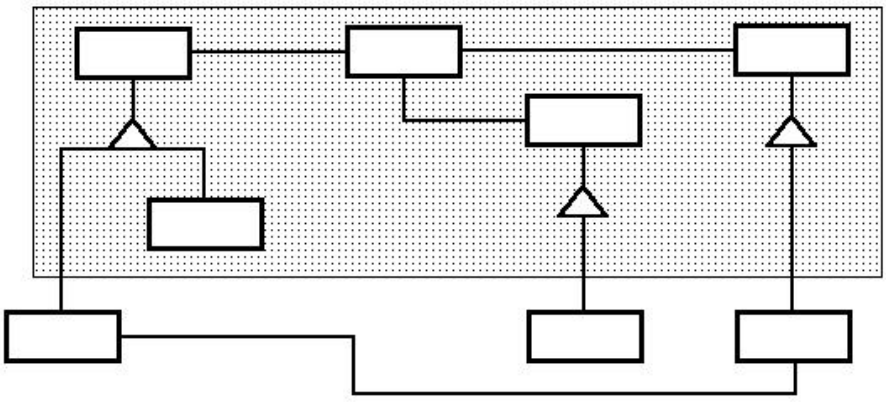

Figura 3-3 - Aplicação desenvolvida reutilizando um *framework*

Outros exemplos de *frameworks* são o ACE (*Adaptive Communication Environment*) e a plataforma Eclipse.

O ACE é um *framework* orientado a objetos desenvolvido em C++, *open-source*, usado no desenvolvimento de aplicações e serviços de comunicação em tempo real, visando portabilidade, confiabilidade e alto desempenho. O seu extenso conjunto de funções e suporte a reuso permitem um desenvolvimento eficaz de componentes ou serviços de comunicação para aplicações em sistemas distribuídos, bem como para o desenvolvimento de comunicação inter-processos entre aplicações locais [ACE] [Vairagade].

A plataforma Eclipse [Eclipse] também é um *framework* orientado a objetos, montado sob o paradigma *open-source*, que fornece um conjunto de ferramentas para os desenvolvedores criarem suas próprias aplicações baseadas no *framework* Eclipse, ou estender as funcionalidades do *framework* através do desenvolvimento de pequenas aplicações denominadas *plug-ins*, que são integradas ao *framework* através de APIs (*Application Programming Interfaces*) e pontos de extensão fornecidos pela plataforma. O *plug-in* se constitui dessa forma como sendo o elemento chave na obtenção do reuso através desse *framework*, pois o desenvolvedor pode elaborar a combinação de diferentes *plug-ins* a fim de obter a funcionalidade requerida ao seu domínio de aplicação. O *framework* é independente de plataforma, podendo ser executado em diversos sistemas operacionais, e é desenvolvido em sua maior parte pela empresa IBM, mas com profundo suporte de outras grandes corporações como Oracle, Intel, Borland, Red Hat e também pela comunidade *open-source* da Internet. Com a proposta de se tornar uma plataforma universal para a integração de software, provendo componentes livres de *royalties* e operando sobre padrões abertos da indústria, como OSGi (*Open Services Gateway Initiative*) e WebDAV (*Web-based Distributed Authoring and Versioning*), o *framework* Eclipse está se consolidando como uma técnica muito importante no desenvolvimento de componentes reusáveis [WebDAV] [OSGi].

3.4. Desenvolvimento baseado em componentes

Um componente é uma caixa preta com uma interface bem definida e uma complexidade que pode variar desde uma simples classe até um *framework* completo. A partir dessa definição o desenvolvimento baseado em componentes consiste em se agrupar componentes de software já existentes de forma a construir sistemas complexos. Geralmente os componentes são apresentados em sua forma binária, ou executável. O desenvolvedor fornece a interface dos componentes, mas não o código fonte. Os componentes podem variar desde pequenos componentes de interface gráfica (GUI *widgets*) até aplicações completas, como um processador de texto [Ezran02]. Apenas recentemente (desde 1999, aproximadamente), o uso de componentes se expandiu a outras áreas, pois até então o uso de componentes estava concentrado apenas na interface gráfica [Facetas04].

Novas aplicações são desenvolvidas através do reuso de componentes. O desenvolvedor provê o *design* de alto nível da aplicação e implementa a função principal que define o fluxo de controle e as informações trocadas entre cada componente. Como o *design* e a implementação de cada componente já estão definidos, basta reusar o componente pronto [Ezran02]. Em alguns casos é um *framework* que provê a estrutura de integração entre um conjunto de componentes. Isso corresponde à abordagem de customização *blackbox*, conforme visto anteriormente.

Um componente típico está organizado internamente em diversas classes, recursos e possivelmente alguns objetos, que são encapsulados pelo componente e definem a sua configuração padrão, sendo que sua funcionalidade é descrita pela publicação de suas interfaces. Um componente pode conter também rotinas (*procedures*), funções e variáveis globais, ou seja, ser desenvolvido usando-se uma linguagem de programação estruturada. Embora não necessária, a tecnologia de orientação a objetos auxilia melhor o desenvolvimento de componentes em relação à definição de suas características [Klabunde99],

mas mesmo outras tecnologias, como XML, podem ser usadas de forma a modelar e construir componentes que suportem o reuso [Bean03]

Do ponto de vista do desenvolvedor a análise do domínio é a parte mais crítica no desenvolvimento de componentes, porque os componentes precisam ser desenvolvidos de forma a atender os requisitos de uma grande variedade de aplicações num domínio específico [Ezran02].

O desenvolvimento baseado em componentes melhora o desenvolvimento de software através da redução da quantidade de código necessária a ser produzida, e também o agrupamento de componentes existentes reduz drasticamente o tempo para implementar e testar novas aplicações. Quanto mais o componente é reusado, menos ele custa para ser desenvolvido [Ezran02].

É uma técnica poderosa de reuso, mas o ponto fraco é o perfil técnico e gerencial necessário no time do projeto para integrar os componentes. A única fonte de informação disponível ao desenvolvedor para entender o comportamento e as propriedades do componente, de forma a integrá-lo a outras aplicações, é apenas a documentação técnica existente sobre o componente.

No mercado atual, duas tecnologias competem para se firmar como o padrão no desenvolvimento de aplicações baseadas em componentes: ActiveX, da Microsoft, e JavaBeans, da Sun Microsystems.

JavaBeans é o padrão para o desenvolvimento de componentes em Java. É compatível com CORBA e ActiveX e mais fácil de desenvolver que este último, apesar de não possuir tantas ferramentas de desenvolvimento quanto as fornecidas pela Microsoft para o desenvolvimento de componentes ActiveX. JavaBeans é o padrão adotado pelas empresas do OMG (*Object Management Group*), com exceção da Microsoft.

Como as tecnologias competem entre si, a Tabela 3-1 apresenta um comparativo entre as duas, com relação às seguintes características:

- interoperabilidade: dois componentes devem se comunicar mesmo que executem em plataformas diferentes e sejam implementados em linguagens diferentes;

- portabilidade: execução e comunicação entre plataformas ou *frameworks* diferentes;

- separação entre interface e implementação: o uso do componente não deve depender de sua implementação;

- composição: poder ser agregado em componentes maiores;

- auto-descrição: o usuário do componente deve descobrir dinamicamente a sua interface;

- transparência de localização: os componentes devem funcionar de forma independente de sua localização nos nós físicos da rede;

- segurança: controle da origem do componente e seu acesso aos recursos locais do usuário;

- *plug and play*: portabilidade em tempo de execução.

Os componentes de interface gráfica (*GUI widgets*) reduzem o tempo de desenvolvimento e melhoram a qualidade e consistência das aplicações, além de ser relativamente fácil obter o reuso de componentes GUI. O mercado possui uma enorme quantidade de bibliotecas de componentes GUI de boa qualidade e para os mais diversos fins, de modo que o desenvolvedor não precise codificar praticamente nada. Observa-se na Internet a grande oferta de componentes desenvolvidos por pequenas *software-houses*, sendo que muitas grandes companhias os oferecem gratuitamente junto com seus aplicativos. Apesar de útil, o reuso de componentes GUI provê apenas uma redução modesta nos custos, comparado ao custo total do desenvolvimento da aplicação [Radding98]

Critério	**ActiveX**	**JavaBeans**
Interoperabilidade	Não. Suportado praticamente apenas no Windows (poucos fabricantes, além da Microsoft, o suportam)	Sim. Suportado por inúmeras ferramentas e plataformas, e compatível com CORBA e ActiveX.
Portabilidade	Não. Somente Windows.	Sim. Independente de plataforma.
Separação entre interface e implementação	Sim. Através da IDL (*Interface Definition Language*)	Sim. Conceito de interfaces é nativo na linguagem Java.
Composição	Sim.	Sim.
Auto-descrição	Sim. As interfaces são declaradas no *Registry* (banco de dados do Windows)	Sim. Possui um mecanismo de introspecção, sem a necessidade de um repositório de interface.
Transparência de localização	Sim. Através da tecnologia DCOM (*Distributed Component Object Model*)	Sim. Através das tecnologias RMI (*Remote Method Invocation*), CORBA e DCOM.
Segurança	Não. Usa assinaturas digitais, mas o controle aos recursos locais é frágil.	Sim. A execução é feita usando-se o conceito de *sandbox* (caixa de areia) que protege os recursos locais.
Plug and play	Não.	Sim. Nativo da linguagem Java.

Tabela 3-1 - Comparativo entre tecnologias ActiveX e JavaBeans

Os componentes do lado do servidor (s*erver-side components*) que constituem a lógica do negócio da aplicação podem trazer ganhos substanciais em termos de redução do custo do projeto, mas exigem uma análise e *design* mais elaborados, além de ser necessário baseá-los numa arquitetura sólida, sob o risco de terem um tempo de vida muito curto.

3.5. Sistemas baseados em agentes

Recentemente, o conceito de agentes de software tem sido proposto como uma extensão ao modelo de objetos. Agentes são sistemas integrados que incorporam características de diversas áreas (inteligência artificial, bancos de dados) e se comunicam com outros agentes através de uma linguagem independente das características de cada agente [Ezran02].

Os agentes de software melhoram a reusabilidade de software em função das seguintes características:

- permitem a integração de software no nível semântico (transferência de conhecimento ao invés de transferência de dados);
- o comportamento do agente é altamente configurável e/ou auto-adaptável a novas condições operacionais e novos protocolos de operação (Exemplos: uso de inteligência artificial e tecnologias de bancos de dados voltadas para sistemas auto-controlados);
- é vantajoso para o desenvolvimento de sistemas distribuídos heterogêneos;
- a interoperabilidade entre agentes se dá através de mesma linguagem de comunicação e vocabulário comum.

Do ponto de vista do desenvolvedor, o *design* de alto nível é a fase mais importante no desenvolvimento do agente e, ao contrário do desenvolvimento baseado em componentes, o desenvolvedor tem um alto grau de liberdade na customização do comportamento de cada agente [Ezran02].

Apesar de ter grandes perspectivas de sucesso e aplicabilidade no campo do reuso, o desenvolvimento de agentes hoje se constitui muito mais numa forma de arte do que num ramo da engenharia. A maioria da pesquisa e desenvolvimento de agentes de software está baseada em arquiteturas proprietárias, que são desenvolvidas ou re-inventadas a partir do início a cada

pesquisa. Uma tendência promissora na aplicação de agentes de software no campo da reusabilidade é o desenvolvimento de *frameworks* de aplicação baseados em agentes de software. Vários exemplos podem ser vistos em [IntelligentLab], com destaque para a arquitetura RETSINA (*Reusable Environment for Task-Structured Intelligent Networked Agents*), um framework baseado em múltiplos agentes com aplicação nos campos de gerenciamento financeiro, comunicações móveis, planejamento logístico em operações militares e na aviação, entre outros [Sycara96].

3.6. Arquiteturas

Arquitetura é descrita em termos de identificação dos componentes do sistema, como são conectados e a natureza dessas conexões (protocolos de comunicação, sincronismo e acesso aos dados). Para elaborar a arquitetura deve-se primeiramente iniciar com a elaboração de um projeto das interfaces dos componentes e posteriormente sua implementação, de modo que a arquitetura nesse ponto seja representada por um conjunto de interfaces que, por sua vez, representam conexões a serem preenchidas por um ou mais componentes [Bosch97].

Essa descrição permite a identificação de algumas arquiteturas de propósito geral encontradas freqüentemente em sistemas de software:

- *pipes*/filtros;
- arquitetura cliente-servidor;
- arquitetura 3 camadas;
- arquitetura N camadas.

O desenvolvimento de arquiteturas de software é voltado muito mais ao reuso de *design* do que ao reuso de código, onde a estrutura de interconexão

dos componentes é reusada, e não o componente propriamente dito. Por outro lado, o reuso planejado de arquitetura e *design* é muito mais difícil que o reuso de código. O *design* precisa ser entendido pelo desenvolvedor de forma a fazer as adaptações e mudanças necessárias. Como as arquiteturas de software são geralmente independentes do domínio de aplicação, nenhum direcionamento é fornecido ao desenvolvedor de forma a escolher a arquitetura certa para a aplicação específica a ser desenvolvida [Ezran02].

A arquitetura deve mostrar também a relação entre as requisições do sistema (a chamada a seus serviços) e os componentes que o compõem. Para tanto, durante a elaboração da arquitetura devem ser consideradas e analisadas propriedades como capacidade, comunicação, consistência e compatibilidade entre componentes [Klabunde99]. É através da especificação da arquitetura do software que se pode alcançar um determinado nível de abstração, facilitando a descrição e compreensão de sistemas complexos e aumentar o reuso de componentes e arquiteturas.

Um exemplo simples mas bastante importante na demonstração do valor agregado existente na aplicação do reuso através da utilização de uma arquitetura bem fundamentada é o uso de *pipes* do UNIX [Krueger92]. Os componentes reusáveis nesse paradigma são os programas completos disponíveis no ambiente UNIX em questão. Os programas são integrados através da conexão da saída de um programa na entrada de outro programa (esse é o conceito fundamental da arquitetura *pipe*). O UNIX possui o comando *who*, que é um programa que lista os usuários conectados ao sistema UNIX, no formato de um usuário por linha. O comando *lc* do UNIX é um programa que conta o número de linhas a partir de uma lista de caracteres de entrada. Dessa forma, conectando-se a saída do programa *who* na entrada do programa *lc* cria-se um novo programa que conta o número de usuários conectados ao sistema UNIX.

3.7. Comparativo entre as técnicas apresentadas

A Tabela 3-2 apresenta um comparativo entre as tecnologias apresentadas do ponto de vista do desenvolvedor, no momento de escolher ou aplicá-las em seu projeto de software.

Tecnologia	**Vantagens**	**Desvantagens**
Técnicas OO	Promove a modularidade e encapsulamento de informações.	Exige grande esforço de modelagem.
Design patterns	Facilita a descoberta de soluções de *design* e provê padrões para o processo de desenvolvimento de software.	Precisa ser implementado a partir do início.
Application frameworks	Aplicações semi-completas disponíveis para domínios específicos, bastando serem customizadas. Reuso de modelo de objetos e arquitetura.	Requer grande experiência e conhecimento profundo do *design* do *framework*.
Componentes	Disponíveis para domínios específicos.	Pouco ou não customizável.
Agentes de software	Altamente customizável e adaptável, permitindo fácil reconfiguração de sistemas complexos.	Tecnologia imatura e não consolidada no mercado.
Arquiteturas	Permite verificação formal de propriedades do sistema. Simplifica o reuso de objetos de negócio.	Sem diretivas para a escolha da arquitetura correta.

Tabela 3-2 - Tecnologias: vantagens e desvantagens

Capítulo 4

Repositório

O repositório é o local onde os componentes de software são armazenados e deve conter um catálogo com todos os componentes disponíveis, bem como ser de fácil acesso e acessível a todas as pessoas interessadas, ou participantes do projeto. Deve-se levar em conta também que o repositório contém informações intelectuais importantes da empresa e, por isso, a forma de acesso e a identificação de cada acesso devem ser controladas e gerenciadas.

Geralmente quando se fala em repositório de componentes pensa-se logo num repositório centralizado, mas não existe nenhuma regra ou razão que defina a necessidade da empresa em centralizar o armazenamento dos componentes. Abaixo alguns fatores a considerar na implantação do repositório:

- número de desenvolvedores, equipes e ambientes de desenvolvimento da organização;
- número de componentes e número de domínios de aplicação aos quais os componentes pertencem.

Nos casos em que haja poucos componentes, ou poucos desenvolvedores, e exista boa informação acerca de onde se encontram os componentes a serem reutilizados, pode-se questionar a real necessidade da centralização do repositório.

À medida que o número de componentes cresce, é necessário definir e manter de forma eficiente um repositório com um catálogo dos componentes

reusáveis. Isso irá evitar situações em que o desenvolvedor não sabe quais componentes ele pode reutilizar, não sabe onde encontrar um determinado componente ou não sabe onde armazenar um componente reusável em potencial [Ezran02].

Um repositório de componentes pode ser gerenciado (seu controle está centralizado em uma única pessoa) ou não gerenciado (não há um responsável pelo seu controle), a diferença entre eles está no fato que, neste último, a eficácia do reuso reside no próprio desenvolvedor, que deve saber onde encontrar cada componente, que pode estar misturado em torno de outros componentes não reusáveis, e ainda correr o risco de utilizar uma versão obsoleta, ou não adequada, do componente.

Existe também o perigo de, a medida que o repositório cresce, ele acabar se tornando um repositório de componentes sem utilidade, ou com uma funcionalidade subutilizada, onde vários componentes executam a mesma função, ou simplesmente não irão executar nenhuma função no futuro, pois nunca serão reusados. Até mesmo nos repositórios para uso pessoal existe a necessidade de se eliminar componentes duplicados ou sem funcionalidade para se assegurar a utilidade do repositório a longo prazo [Norton03].

4.1. Vantagens do repositório

Michel Ezran apresentou as seguintes vantagens em se adotar o uso de um repositório centralizado de componentes [Ezran02]:

- lugar único para armazenar os componentes, bem como pesquisar por componentes existentes;
- forma padronizada de documentar, pesquisar e enumerar os componentes;
- forma padronizada de controlar mudanças e melhorias nos componentes.

4.2. Conteúdo do repositório

Já é sabido que um repositório de componentes reusáveis deve possuir, obviamente, componentes cujo reuso seja freqüente nos projetos desenvolvidos pela empresa. Mas é preciso considerar que os diferentes componentes disponíveis podem não ser iguais em termos de linguagem de programação, formato de arquivos, granularidade e documentação. Além disso, como visto anteriormente, o repositório deve conter um catálogo de componentes de forma a facilitar a busca e identificação dos componentes pelo desenvolvedor. Sendo assim, pode-se concluir que o repositório será composto de um catálogo e de vários componentes. Como os componentes armazenados podem ser diferentes entre si, para que o catálogo possa referenciar os diversos componentes existentes, cada componente deve ser composto de uma parte comum, a qual será referenciada no catálogo. Essa parte comum é a sua ficha cadastral, a outra parte seria o corpo do componente, que na verdade engloba todo o conjunto de artefatos que compõem o componente original [Ezran02].

A ficha cadastral do componente, também chamada de metadado do componente [Norton03], deve conter uma série de campos a fim de facilitar a sua busca no catálogo:

- nome único de identificação do componente;
- autor;
- data de criação;
- data da última atualização;
- versão atual;
- lista de palavras chave associadas ao componente;
- breve descrição do componente (área de uso ou tecnologia envolvida).

Detalhes de usabilidade, instalação e teoria do componente são descritos em documentos à parte, pertencentes ao corpo do componente;

- localização do componente (o componente pode estar armazenado dentro do repositório ou externamente, em algum sistema de controle de versão da empresa ou até mesmo externamente à empresa, num servidor *web* de outra companhia, por exemplo).

Um modelo básico de um repositório de componentes apresentado por [Ezran02] pode ser visto na Figura 4-1.

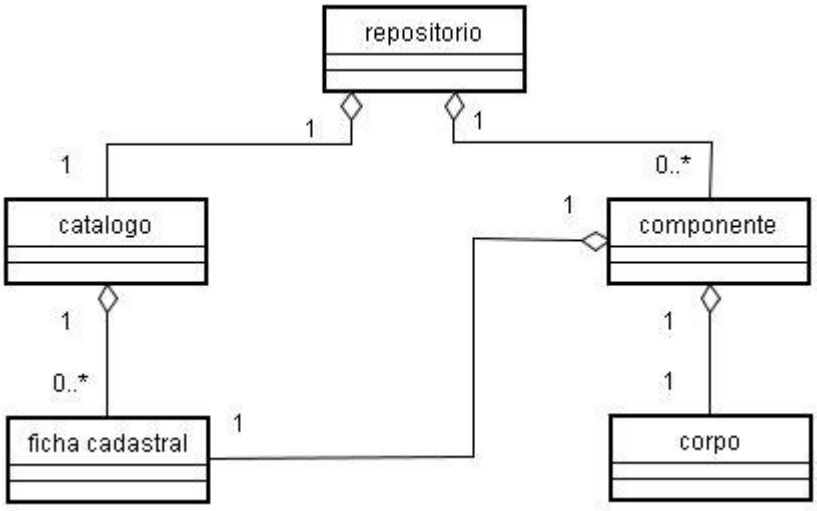

Figura 4-1 - Modelo de implementação de repositório

Esse modelo de implementação de repositório foi elaborado usando-se a notação da linguagem UML, de forma que seja possível implementá-lo em qualquer linguagem de programação. Seguindo-se a notação dos componentes e o relacionamento entre eles, vemos que o repositório é composto por 1 catálogo e um conjunto de 0 ou mais componentes, sendo que o catálogo e cada componente estão associados a um único repositório. O catálogo, por sua vez, é composto de 0 ou mais fichas cadastrais, e cada ficha cadastral pertence a somente 1 catálogo. Cada componente possui 1 ficha cadastral e 1 corpo associado. A mesma ficha cadastral que é parte do componente é a que faz parte

do catálogo e se constitui no elo de ligação entre o catálogo e os componentes do repositório. Sendo assim, a partir do repositório construir o catálogo e, através das fichas cadastrais existentes no catálogo, chega-se ao componente reusável, cujo corpo se constitui no artefato de software a ser reusado pelo usuário do repositório.

4.3. Funções do repositório

Basicamente, a função do repositório é listar seus componentes e prover meios de acesso a eles, mas pode ser aprimorado a fim de prover várias outras funções de forma a melhorar a sua utilização, tanto por parte dos desenvolvedores quanto da equipe de gerenciamento do repositório [Ezran02]. Diversos exemplos de funções possíveis de serem implementadas no repositório são apresentadas abaixo.

- identificação e descrição do componente: cada componente deve ser descrito por um único nome e possuir um conjunto padrão de atributos. Exemplo: o conjunto padrão de atributos poderia ser composto por nome, domínio, categoria, versão, autor, linguagem e descrição;

- inserção e remoção de componentes: possibilidade do gerenciador de reuso inserir novos componentes e retirar componentes obsoletos. Essa função pode ser estendida aos próprios usuários do repositório. A inserção do corpo de um componente deve estar atrelada à correspondente inserção da correta descrição do componente, conforme o item anterior. Uma extensão dessa funcionalidade é o cenário em que uma nova versão de um componente já existente é adicionada ao repositório;

- visualização do catálogo: deve ser possível a visualização de todo o catálogo, contendo a descrição de todos os componentes. É particularmente útil para encontrar rapidamente o componente quando o repositório é pequeno (possui poucas dezenas de componentes);

- mecanismo de busca: deve ser possível a busca por palavras chave, fornecidas pelo usuário, contidas em qualquer parte da ficha cadastral do componente. O resultado é a apresentação das fichas cadastrais dos componentes encontrados. Uma extensão é a busca diretamente no corpo do componente;

- *download*: uma vez encontrado o componente, deve ser possível obter o componente completo (o corpo do componente) de maneira simples, rápida e segura;

- organização do catálogo: como o repositório pode conter muitos componentes, a busca textual pode ser muito demorada ou ineficiente (ou ambos). É interessante que o catálogo seja organizado também por palavras chave (exemplos: estrutura de dados, código, tutorial), áreas de aplicação (exemplos: C++, Java, Biologia, Segurança) e sinônimos na descrição do componente (exemplo: estrutura de dados, com *links* para componentes similares como árvores binárias e listas ligadas). Em alguns repositórios essa organização ocorre através de menus hierárquicos (em forma de árvore), e a pesquisa textual pode ser restrita a algumas áreas do catálogo, tornando-se mais rápida e eficiente;

- histórico: o repositório deve conter a descrição do histórico de cada componente (autor, data de criação, lista de modificações). Outros dados podem ser registros de melhorias necessárias, resultados de integração e teste do componente, exceções ao reuso. Esses dados podem estar na ficha cadastral;

- estatísticas: o repositório deve coletar dados e registrar automaticamente quais os componentes mais acessados, quando, quantidade e tipo de buscas, número de componentes;

- controle de acesso: como os componentes podem conter informações confidenciais, devem ser restritos a grupos específicos, bem como a

gerência do repositório deve ser feita por uma única pessoa (ou um grupo pequeno de pessoas). O acesso deve ser controlado também por motivos de se garantir a integridade do repositório (minimizar os riscos de remoção indevida de componentes, por exemplo);

- controle de versão: o repositório deve conter mecanismos de controle de múltiplas versões de um mesmo componente;

- controle de mudanças: procedimentos devem ser definidos para propor, discutir, aceitar e implementar mudanças nos componentes do repositório;

- notificação de mudanças: mudanças nos componentes (adição, remoção) bem como mudanças no próprio repositório (novas funções, procedimentos) devem ser notificadas automaticamente a todos os usuários em potencial;

- notificação de atualizações: identificar de forma única cada usuário que esteja reusando cada componente do repositório, de forma a notificar esses usuários sobre atualizações (devido a correções de falhas, por exemplo) nos componentes que estejam reusando.

Nota-se que nem todas são necessárias, dependendo do contexto em que o repositório é utilizado dentro da empresa. Por exemplo, se o repositório é usado apenas em projetos internos dentro de uma mesma empresa, um controle de acesso pode mostrar-se desnecessário, ou se existirem poucos componentes no repositório, o mecanismo de busca pode ser eliminado, já que é mais simples apresentar sempre a lista completa dos componentes.

4.4. Cenários de utilização

Michel Ezran apresenta alguns cenários que demonstram como seria o uso de tais funções pelos usuários do repositório [Ezran02].

4.4.1. Busca de um componente por um usuário

O usuário analisa o catálogo de componentes, verificando a descrição de alguns componentes e em seguida faz uma busca no catálogo por uma palavra chave. Um número excessivo de componentes é retornado. O usuário então faz nova busca utilizando-se desta vez de duas palavras chave. Alguns componentes são retornados. O usuário analisa a descrição desses componentes, encontra o componente desejado e faz o seu *download*.

4.4.2. Usuário reusa um componente existente

Após ter procurado, encontrado e reusado um componente do repositório, o usuário faz a atualização do histórico do componente, melhora a descrição de uso e requisita ao autor correções ou melhorias no componente.

4.4.3. Um componente é alterado

O gerenciador de reuso recebe do autor uma nova versão melhorada de um componente existente e opta por inseri-lo no repositório. Depois de inserido o componente, o gerenciador de reuso notifica todos os usuários em potencial acerca da mudança (isso poderia ser feito também de forma automática pelo próprio repositório). Ao receber a notificação, usuários podem decidir pela atualização do componente nos locais onde ele foi reusado. Dependendo da política de reuso adotada, essa atualização pode se tornar obrigatória, o que simplifica também o gerenciamento do repositório, pois somente a última versão do componente precisaria ficar armazenada.

4.4.4. O gerenciador de reuso insere um novo componente

Um novo componente é desenvolvido e enviado ao gerenciador de

reuso pelo desenvolvedor. O gerenciador de reuso verifica que o componente é potencialmente reusável e inicia o processo de qualificação do componente (testes e documentação). Uma vez qualificado para reuso, o gerenciador de reuso e o desenvolvedor definem a descrição do componente a ser armazenado no repositório. O componente é inserido e uma notificação é enviada aos usuários informando que um novo componente se encontra disponível. Este cenário pode sofrer pequenas variações, dependendo do responsável por avaliar a reusabilidade do componente (se o gerenciador de reuso, o desenvolvedor ou uma terceira pessoa) e da maneira como o componente é apresentado ao gerenciador de reuso.

4.5. Implementação

A implementação de um repositório deve obedecer a uma estrutura planejada de forma a facilitar seu uso e manutenção ao longo de sua existência. Como muitas empresas estão centradas num modelo de redes (senão na própria Internet), é imprescindível que o repositório seja acessível de qualquer nó de rede dentro da empresa. Sendo assim, um modelo de arquitetura 3-camadas seria o recomendado para a implementação do repositório, pois essa arquitetura permite conciliar os requisitos de usabilidade, manutenibilidade e disponibilidade requeridos pelo repositório [Ezran02].

No caso específico de implementação de um repositório de software, uma camada seria dedicada ao armazenamento dos componentes, onde um banco de dados, sistema de gerência de configuração ou o próprio sistema de arquivos do servidor de componentes poderia ser usado, outra camada seria dedicada ao empacotamento dos componentes em objetos (itens) de reuso únicos a serem disponibilizados aos usuários e uma terceira camada seria a camada de acesso a partir da rede, que poderia ser um *web browser* ou uma aplicação cliente tradicional.

Os pontos a serem considerados na escolha ou implementação do repositório são:

- número de componentes a serem armazenados;
- natureza dos componentes;
- existência ou uso na empresa de outros repositórios, como sistemas de gerência de configuração ou ferramentas CASE.

Se um repositório já existe na empresa, adicionar um novo pode duplicar algumas informações, causando problemas de manutenção ao longo do tempo, pois as informações precisam estar sincronizadas entre os repositórios existentes. Há duas soluções possíveis: estender o repositório existente com funções e áreas específicas para os componentes de reuso; implantar um repositório de reuso que faça interface com o repositório existente a fim de manter as informações redundantes sincronizadas automaticamente.

O número de componentes também influi na dificuldade de se organizar o repositório e implementar suas funções. Quanto menor o número de componentes, mais simples são as funções de descrição e busca e mais simples de se implementar o repositório sobre as aplicações já existentes na empresa. Quanto maior o número de componentes, maior a necessidade de se utilizar um repositório especializado para reuso.

Em geral é preferível adaptar as ferramentas existentes para servir como repositórios de reuso. Se não há ferramentas que suportem isso, recomenda-se desenvolver um repositório simples com um catálogo de componentes, utilizando-se um banco de dados e um servidor *web*. À medida que o reuso de componentes seja disseminado e o número de componentes aumente, deve-se então reavaliar e investir num repositório especializado.

Embora os componentes possam ser armazenados em um repositório local, numa máquina sem acesso à rede ou à Internet, isso irá fazer com que o

desenvolvedor necessite se deslocar até a máquina onde se encontra o repositório para acessar os itens de interesse. Isso pode afetar a produtividade quando o desenvolvedor estiver longe do repositório e necessitar do componente de forma imediata. Por isso o repositório deve incluir alguma forma de acesso remoto, de preferência via Internet, seja através de uma aplicação *web* ou com uma aplicação local que forneça serviços através de *web-services* [Norton03]. Com a disseminação da Internet, esse tipo de recomendação já é considerada implícita em qualquer aplicação voltada ao trabalho colaborativo, não especificamente a repositórios de reuso.

Em seu trabalho, William Frakes apresenta os processos envolvidos em torno da implementação de um repositório de componentes reusáveis, que pode ser conferido na Figura 4-2 [Frakes96].

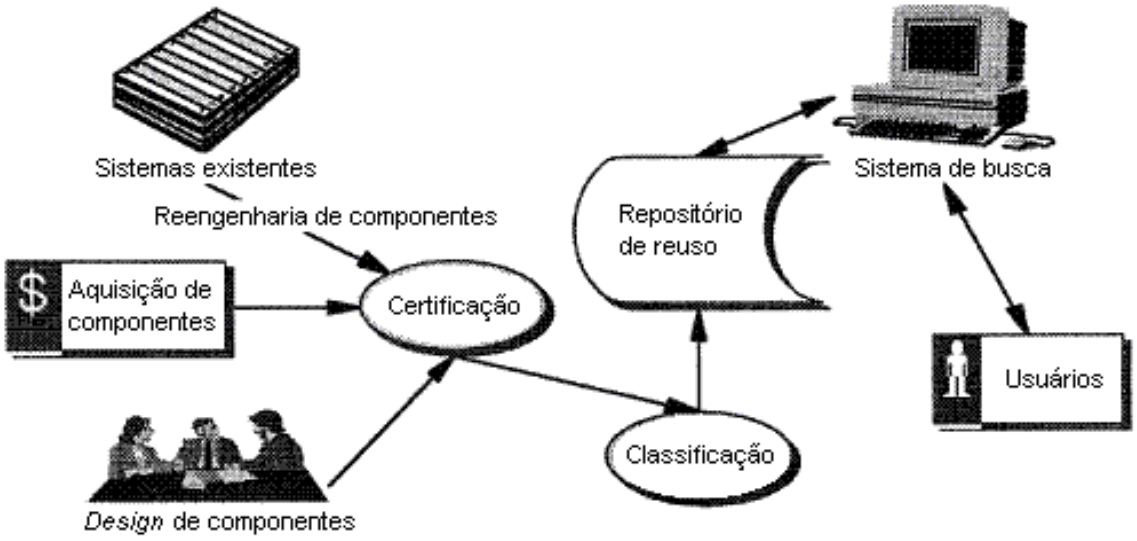

Figura 4-2 - Repositório de componentes e processos envolvidos

Os componentes são produzidos a partir de três fontes possíveis: o componente é adquirido externamente; os sistemas existentes passam por uma reengenharia de componentes a fim de se extrair possíveis componentes reusáveis; ou o componente é desenvolvido por uma equipe de *design* de componentes. Todos os componentes gerados passam por um processo de

certificação e em seguida por um processo de classificação antes de serem adicionados ao repositório de reuso. Os usuários por sua vez utilizam um sistema de busca, que tem acesso ao repositório de reuso, para consultar e obter os componentes reusáveis desejados.

4.6. Repositórios públicos

Encontra-se disponível na Internet uma grande quantidade de repositórios públicos, contendo componentes, exemplos de códigos, artigos, tutoriais e livros acerca de desenvolvimento de software para os mais diversos fins. Abaixo alguns exemplos:

- www.cetus-links.org : é um repositório de *links* para diversos assuntos, todos relacionados a orientação a objetos e componentes de software e abrangendo inúmeras tecnologias, desde C/C++, Java, UML, COM até Eiffel, ADA e Simula, que são linguagens de programação mais antigas, mas importantes na área de orientação a objetos. Através dos *links* o desenvolvedor tem acesso a artigos, tutoriais, aplicativos, bibliotecas de classes, documentos e muitos outros artefatos de software;

- www.developer.com : é um repositório de artigos técnicos voltados principalmente às linguagens Java e C++. Vários artigos trazem referências a bibliotecas de classes já existentes. A importância desse repositório está na apresentação de técnicas de resolução de problemas através de recursos de programação ou de ferramentas de desenvolvimento, de forma que o desenvolvedor reuse as técnicas disponíveis, ao invés de desenvolvê-las completamente;

- www.codeguru.com : segue o mesmo modelo de www.developer.com, com artigos técnicos que trazem referências a classes e programas utilitários. Embora mais especializado em programação para o ambiente Microsoft Windows, utilizando-se as linguagens C/C++ e C# para a plataforma .NET,

da Microsoft, a página possui *links* para outras tecnologias.

- www.jars.com : especializado na linguagem Java, esse *site* apresenta programas utilitários que auxiliam no desenvolvimento de projetos, como o SA4J (*Structural Analysis for Java*), componentes completos que podem ser integrados a outras aplicações e uma grande biblioteca de classes, separadas por categorias e sub-categorias como Biologia, Matemática, Gráficos 3D, Jogos, Mercado de Ações. Exemplo: na categoria Ciência, sub-categoria Biologia, encontramos o JGAP (*Java Genetic Algorithms Package*), um pacote de classes com algoritmos para a solução de problemas de simulação genética;

- www.sourceforge.net : o SourceForge.net é o maior repositório público de código e aplicações *open-source* disponível na Internet. Neste *site* o desenvolvedor pode encontrar aplicações de domínio público para serem incorporadas ou utilizadas em seu projeto, bem como utilizar a infra-estrutura do *site* para desenvolver o seu projeto e contar com a ajuda de outros desenvolvedores da comunidade *open-source*. O SourceForge é também uma ferramenta de controle de projetos que pode ser utilizada localmente na organização, provendo serviços de gerência de configuração, controle de falhas, acesso de usuários e controle de documentação de projeto, entre outros;

- www.softwarestore.com : é uma loja virtual de aplicativos, organizados por área de interesse, como Educação, Segurança e Escritório. O modelo de reuso nesse caso se refere à aquisição de software de empresas terceirizadas;

- www.componentsource.com : é um excelente repositório especializado em componentes ActiveX e para a plataforma Microsoft .NET. A divisão dos componentes em categorias é bastante detalhada, facilitando muito a busca e acesso ao conteúdo do repositório. Apresenta componentes específicos para telefonia, autorizações de cartão de crédito, compressão de imagens e

códigos de barras, por exemplo. Possui também componentes para outras linguagens de programação e outras tecnologias, como Java e bancos de dados;

- www.planetsourcecode.com : é um repositório com o propósito de fornecer exemplos de código fonte em algumas linguagens de programação (Java, C++, Visual Basic), sem no entanto especificar a área de aplicação do código. Exemplos: função para concatenar arquivos de imagem, função para gerar números primos. O repositório está dividido em categorias como Algoritmos, Arquivos, Ordenação, Banco de dados.

Alguns desses repositórios são especializados em artigos e tutoriais, outros em componentes Java, outros somente em componentes ActiveX, e há ainda os repositórios especializados em aplicativos completos (sejam eles *open-source* ou não). Pode-se observar que eles diferem na organização e mecanismos de busca dos componentes, mas o que todos têm em comum são descrições de seus componentes e mecanismos para se realizar o *download* desses componentes. A importância desses repositórios públicos está no fato de que qualquer empresa poder desenvolver o seu componente e fazer negócios com isso, disponibilizando-o ao mundo inteiro de forma imediata, bem como se utilizar desses componentes a fim de reduzir o *time-to-market* no desenvolvimento de seus produtos.

No caso de repositórios públicos que forneçam trechos de código isolados, como macros e funções (citados no exemplo do www.planetsourcecode.com), o usuário desenvolvedor deve se policiar a fim de adaptar a facilidade do repositório público com a política de reuso adotada no projeto, criando internamente novos componentes reusáveis. Como a granularidade dos artefatos do repositório é bastante baixa, o desenvolvedor pode facilmente adaptar o código encontrado à sua aplicação e, nesse caso, a busca pura e simples por trechos de código em repositórios públicos passa a caracterizar o reuso oportunístico, que deve ser evitado.

Em sua tese, Robert Norton analisa o problema de como os desenvolvedores podem organizar de forma efetiva os seus componentes de forma a resgatá-los com um mínimo de tempo e esforço, quando um componente reusável se faz necessário [Norton03]. Robert Norton faz uma análise de cerca de 20 ferramentas de repositório existentes no mercado, sob os pontos de vista de privacidade, acessibilidade via Internet, descrição de metadados de componentes personalizáveis, mecanismos de navegação flexíveis, suporte a grande variedade de tipos de componentes, possibilidade de se criar relacionamentos entre os componentes do repositório e possibilidade de se buscar componentes através de consultas em linguagem natural. Ele verifica que nenhuma das ferramentas oferece todas essas facilidades, mas conclui que todas são importantes para um gerenciamento eficiente dos repositórios de componentes e que um repositório dedicado é mais efetivo que a simples busca de um componente num sistema de arquivos hierárquico.

Capítulo 5

Processos

Processo é um conjunto de tarefas definidas para se criar um produto. Os processos são compostos, isto é, podem ser sub-divididos em sub-processos até chegar a tarefas atômicas, também chamadas de atividades, realizadas por uma pessoa num local específico. Os processos de software se referem a todas as tarefas necessárias para se produzir e gerenciar o software. Os processos de reuso correspondem ao sub-conjunto de tarefas necessárias ao reuso de software dentro da empresa [Ezran02] e a introdução de um processo de reuso tem se mostrado como o maior desafio na disseminação do conceito de reuso de componentes [Biggs04].

Os processos se fazem necessários a fim de garantir que atividades similares sejam realizadas da mesma maneira por equipes diferentes em épocas diferentes. A adoção de novos processos ou a customização de processos existentes é uma condição essencial para o sucesso do reuso.

A produção de componentes reusáveis é um processo de reuso que pode ser dividido em várias etapas menores como:

- identificação dos componentes reusáveis. O primeiro passo é pesquisar os projetos da empresa e identificar os componentes que ocorrem freqüentemente;
- desenvolvimento dos componentes reusáveis. Identificados os componentes, o próximo passo é torná-los reusáveis a longo prazo,

desenvolvendo os componentes completamente ou adquirindo-os de terceiros;

- validação dos componentes reusáveis. Com os componentes prontos, o terceiro passo é testá-los, corrigir o maior número de falhas possível e disponibilizá-los num repositório.

Além da produção de componentes reusáveis, outro passo é a transição para o processo de reuso, que pode ser considerada das seguintes formas, com o intuito de se minimizarem os riscos [Ezran02]:

- introduzir o reuso como uma série de pequenas mudanças, cada uma aplicada a uma pequena parte do processo de desenvolvimento, e limitada a um pequeno número de componentes (exemplos: limitar o reuso a reuso de componentes no domínio técnico, que são mais fáceis de identificar e reusar, comparados aos componentes no domínio de aplicação; limitar o reuso a reuso de componentes de baixa granularidade, mais fáceis de especificar que os de alta granularidade);

- numa organização com várias unidades, implantar as mudanças numa unidade piloto, antes de disseminar o processo às outras unidades;

- implantar as mudanças em apenas um projeto piloto, um domínio de aplicação, uma linha de produtos ou um cliente, antes de estender aos demais.

Porém, antes de se pensar na implantação do processo de reuso, o primeiro e mais importante passo é obter suporte e comprometimento da alta gerência da organização para o programa de reuso. Isso é crucial pois o programa de reuso pode afetar todas as áreas do processo de produção de software na empresa e o suporte da alta gerência precisa ser obtido para permitir que mudanças nos processos existentes sejam implantadas (ou até mesmo mudanças na política da empresa, se necessário) [Biggs04].

Em seu trabalho sobre práticas envolvidas no processo de desenvolvimento de software, Steve McConnell cita o reuso como uma de suas *best practices* e, resumidamente, classifica a eficácia do processo de reuso da seguinte forma [McConnell96]:

- Possibilidade de redução do cronograma original: alta
- Melhorias na visibilidade do progresso do projeto: muito baixa
- Possibilidade de redução dos riscos do cronograma: alta
- Chances de sucesso imediato: baixa
- Chances de sucesso a longo prazo: muito alta

Junto com o processo de reuso, surgem outros pontos a considerar a fim de se obter sucesso:

- definir novos papéis para coordenar o reuso entre equipes diferentes;
- permitir o treinamento das equipes envolvidas;
- prover incentivos apropriados para o reuso, a fim de evitar argumentos do tipo: “Por que prejudicar o cronograma do meu projeto apenas para que o que eu faço seja reutilizado por outra equipe depois?” [Facetas04].

5.1. Classificação dos processos

Em seu trabalho, Michel Ezran dividiu o processo de reuso dentro de uma empresa em 5 processos principais [Ezran02]:

- **Produção:** identificação, desenvolvimento e classificação dos componentes;
- **Utilização:** localização e avaliação dos componentes e obtenção do seu

reuso através da integração desses componentes ao projeto sendo desenvolvido. Essa é a etapa do processo de desenvolvimento tradicional afetada pelo reuso;

- **Gerenciamento dos componentes:** armazenamento dos componentes, gerenciamento do repositório (se existir) e disseminação dos componentes disponíveis para reuso;

- **Manutenção e suporte:** suporte aos clientes de reuso na integração e utilização dos componentes; suporte à correção e evolução dos componentes reusáveis existentes;

- **Gerenciamento do reuso:** introdução e monitoramento do reuso dentro da empresa ou departamento, análise de resultados do reuso, coordenação de atividades de reuso e definição e melhorias nos processos de reuso. Esse processo compreende também a definição de como o processo de reuso irá impactar os processos existentes e definir o processo de como novas aplicações serão desenvolvidas a partir de componentes existentes.

Estes cinco processos, juntamente com seus sub-processos, são apresentados na Tabela 5-1. Os processos e sub-processos podem assumir diferentes granularidades, desde simples atividades como publicação de catálogo, até as mais complexas como análise de domínio.

Processo principal	**Sub-processos / atividades**
Produção de componentes	• Análise de custo/benefício • Análise de domínio • Desenvolvimento de componentes • Aquisição de componentes • Reengenharia de aplicações • Testes e qualificações • Documentação
Utilização	• Busca de componentes • Avaliação • Análise de risco • Integração
Gerenciamento dos componentes	• Publicação de catálogo • Gerenciamento do repositório • Inserção de componentes • Remoção de componentes • Captura de requisitos
Manutenção e suporte	• Suporte aos componentes • Suporte ao repositório • Suporte aos processos • Correção dos componentes • Evolução (melhoria) dos componentes
Gerenciamento do reuso	• Introdução do reuso • Coordenação de atividades de reuso • Métricas • Relatórios de reuso • Melhorias no processo

Tabela 5-1 - Processos de reuso

5.2. Classificação dos papéis e responsabilidades

Uma vez definidos os processos de reuso, novas responsabilidades devem ser atribuídas de forma a assegurar que os processos serão seguidos e monitorados corretamente. Novos papéis devem ser criados e distribuídos entre as pessoas dentro da organização. É importante ressaltar que uma pessoa pode assumir mais de um papel dentro do processo de reuso [Ezran02]. Alguns dos novos papéis específicos ao processo de reuso são:

- **gerente de reuso:** responsável por monitorar o programa de reuso de uma forma geral;
- **gerente de repositório:** gerencia o armazenamento e configuração dos componentes e assegura o seu acesso aos desenvolvedores;
- **membro da equipe de suporte:** fornece suporte à aplicação dos métodos e processos de reuso;
- **criador de componentes:** é o idealizador e responsável pelo componente. Também fornece suporte à sua manutenção;
- **desenvolvedor de componentes:** desenvolve componentes reusáveis. Nem sempre o criador é o desenvolvedor do componente.

Os papéis tradicionais impactados pelo processo de reuso são:

- **gerente de projeto:** considera os objetivos de reuso dentro do projeto, coordenando também outros projetos se necessário;
- **arquiteto de software:** define a arquitetura de forma a facilitar o reuso de componentes;
- **engenheiro de software:** participa do programa de reuso junto aos criadores e desenvolvedores de componentes;

- **especialista ou analista de negócios:** ajuda a definir os modelos de domínio e classificar os componentes.

A definição e atribuição desses papéis são influenciadas pela relação entre o paradigma do desenvolvimento integrado e o paradigma do desenvolvimento separado de componentes reusáveis, em relação às atividades normais de desenvolvimento de software dentro da empresa. O desenvolvimento separado de componentes consiste no modelo em que exista uma equipe de desenvolvedores dedicada exclusivamente ao desenvolvimento de componentes reusáveis. O desenvolvimento integrado de componentes é o modelo em que desenvolvedores de um projeto são responsáveis pelo desenvolvimento de componentes reusáveis para outros projetos (além do projeto original para o qual o componente foi criado). Michel Ezran propõe que o desenvolvimento separado de componentes de software reusáveis seja recomendado se [Ezran02]:

- o conhecimento técnico é escasso. O desenvolvedor de tal componente é altamente especializado e seu conhecimento não pode ser diluído entre outras atividades;

- é possível prever com antecedência o desenvolvimento de componentes com ou sem potencial de reuso;

- é definido que componentes reusáveis devem ser desenvolvidos com um padrão de qualidade maior, ou diferente, dos padrões aplicados aos componentes não reusáveis;

- os componentes serão reusados em diferentes projetos, e esses projetos não possuem recursos extras para o desenvolvimento de componentes;

- o custo do trabalho entre o desenvolvimento de componentes reusáveis e o desenvolvimento de componentes não-reusáveis é relativamente balanceado (se o desenvolvimento de componentes reusáveis predomina, então o desenvolvimento para reuso deveria ser instituído como norma

entre todos os desenvolvedores, e o desenvolvimento de componentes não reusáveis seria exceção);

- o volume total do trabalho envolvido é grande o suficiente para justificar a criação de estruturas organizacionais separadas para o desenvolvimento normal de aplicações e o desenvolvimento voltado ao reuso. Isso é também chamado de condição de massa crítica;

- o estilo de gerenciamento favorece a especialização, criação de vários níveis na hierarquia organizacional e forte controle do processo de gerenciamento.

5.3. O fator humano

Supondo que um processo de reuso tenha sido implantado e existam componentes reusáveis disponíveis para serem utilizados por um desenvolvedor ou uma equipe de desenvolvimento, a probabilidade desse desenvolvedor ou dessa equipe de escolher reusar um componente em vez de desenvolvê-lo do início ainda pode ser baixa. Terry Coata afirma que o desenvolvimento de software ainda é mais arte do que engenharia [Coata00]. Os desenvolvedores possuem seus próprios estilos de codificação e em muitos casos eles irão estimar que gastarão mais tempo estudando como se usa um componente existente do que simplesmente recriando o componente. Essa estimativa pode ser verdadeira se o componente em questão for mal projetado, documentado ou testado.

O ponto chave para fazer com que um processo de reuso seja amplamente adotado é facilitar o trabalho do desenvolvedor o máximo possível. Facilitar a descoberta de componentes reusáveis através de um repositório corretamente indexado e atualizado. Para cada componente reusável possuir um responsável técnico, um especialista no componente que seja capaz de explicar ao desenvolvedor como o componente funciona e como integrá-lo no projeto sendo desenvolvido. Assegurar que exista um processo padrão para reportar

falhas nos componentes reusáveis, e assegurar que essas falhas sejam corrigidas num tempo aceitável, ou que soluções alternativas (*workarounds*) sejam fornecidas.

Um outro ponto de vista com relação ao fator humano dentro do processo de reuso é apresentado por [McConnell96]. Às vezes a aplicação do reuso é planejada apenas para código, sendo que é possível aplicar o reuso a partir de qualquer esforço de desenvolvimento anterior, seja código, *design* ou documentos. O que geralmente não é considerado no processo de reuso é que o reuso de pessoas que participaram em projetos similares é uma das formas mais fáceis e eficazes de reuso dentro da organização.

5.4. Análise crítica

Poucas empresas possuem uma visão clara de negócios a respeito dos benefícios do reuso. Em geral, as empresas não definem o escopo do programa de reuso nem sua prioridade em relação aos objetivos do negócio. Não associam o reuso de software aos requisitos de mercado e não analisam profundamente o potencial ganho de mercado através do reuso de componentes. A implantação de um programa de reuso é geralmente uma decisão técnica (de software) e não uma decisão de negócios [Ezran02].

A maturidade dos processos de software existentes tem se mostrado um importante facilitador na implantação de processos de reuso. Muitos casos de implantação de processos de reuso mal sucedidos se devem ao fato que eles foram baseados em processos de software não suficientemente estáveis antes da introdução do reuso. O reuso deve ser planejado com antecedência (no momento de migrar para um novo grupo de tecnologias, por exemplo), mas sua introdução não deve acontecer antes que os processos de software existentes tenham atingido um nível de maturidade suficiente [Ezran02].

Capítulo 6

Métricas

O ponto de partida para qualquer programa de métricas de reuso é definir exatamente o que medir. Sem ter essa resposta plenamente consolidada o que será medido serão apenas números com pouca ou nenhuma importância. Coletar números é fácil, produzir medidas úteis não, e muitos programas de métricas falham porque não conseguem diferenciar essas duas situações [Ezran02]. Aqui se aplica claramente uma frase dita por Watts Humphrey: “Você pode obter praticamente qualquer número que você queira (...) mudando apenas a maneira como você conta.”

Para a introdução de métricas em reuso não se deve pensar em redução de custos, mas na redução no tempo de implementação ou na melhoria da qualidade do software produzido.

O nível de medida de reuso requer um entendimento preciso do que é contado como reuso ou não. Não existe um padrão consolidado, cada organização deve adotar um modelo e usá-lo consistentemente. Por exemplo: somente componentes vindos do repositório são contados como reuso. Isso marca a diferença entre reuso oportunístico e reuso planejado. Às vezes valores absolutos de medidas podem não ser significativos, mas as variações desses valores observados ao longo dos projetos podem representar informações úteis e significativas para o processo de reuso [Ezran02].

Para coletar os dados, é importante uma definição clara de como contar o que é reusado e como é reusado. Essa definição poderia ter a seguinte

forma: um componente reusável é qualquer componente vindo do repositório e utilizado sem modificações para construir, direta ou indiretamente, um produto desenvolvido num dado projeto. Esse componente deve ser contado apenas uma vez por projeto, mesmo que tenha sido reusado mais de uma vez no mesmo projeto [Ezran02]. Dessa definição extraem-se pontos importantes a serem considerados:

- "utilizado sem modificações": se o componente precisar ser modificado, então há um trabalho de desenvolvimento envolvido e isso não deve ser considerado como reuso.

- "vindo do repositório": somente componentes qualificados, documentados e desenvolvidos para reuso podem ser contados oficialmente. Serviços ou componentes disponíveis no sistema operacional não são contados como reuso, pois estão disponíveis livremente e não foram desenvolvidos visando um repositório para reuso. Por outro lado, componentes ou aplicativos avaliados e adquiridos de terceiros, especificamente para o projeto em desenvolvimento e inseridos no repositório para posterior reuso devem ser considerados nas métricas. Os componentes vindos de dentro do próprio projeto não são representativos para as métricas. A lógica para isso é que a identificação, o desenvolvimento e a reutilização de módulos dentro de um mesmo projeto é apenas uma boa prática de *design*, que cada projeto deveria adotar de forma padrão, sem a necessidade de técnicas de reuso planejado ou repositórios.

- "contado apenas uma vez": assim como internamente ao projeto um componente reusado várias vezes é desenvolvido apenas uma vez, o componente vindo do repositório deve ser contado apenas uma vez.

6.1. Exemplos de métricas

Um programa de métricas de reuso deve identificar que atributos de

quais entidades (ou componentes) serão medidos. Até mesmo simples projetos podem ser compostos de centenas de atributos e medi-los todos pode não ser possível devido a limitações de prazo e custo e também não necessário porque nem todos os atributos podem ser úteis.

Em seu trabalho, Michel Ezran adota uma variação da abordagem GQM (*Goal-Question-Metric*) proposta por Basili, que é uma ferramenta conceitual largamente aceita para guiar a escolha de medidas úteis dentre o grande número de possibilidades disponíveis [Ezran02] [Basili88]. Nessa abordagem o programa de métricas é iniciado a partir de certos objetivos, que são definidos a partir de um modelo baseado nas seguintes questões: propósito, atributo, entidade, ponto de vista e contexto. O programa de métricas deve começar com poucos objetivos, que vão sendo refinados e estendidos ao longo do tempo.

Dessa forma, pode-se definir os objetivos iniciais para o programa de métricas de reuso. Esses objetivos são apresentados na Tabela 6-1.

Objetivo	Propósito	Atributo	Entidade	Ponto de vista	Contexto
O-1	Caracterizar	A utilização	Do repositório	Do ponto de vista do gerenciador de reuso	No contexto de um departamento da empresa
O-2	Avaliar	A extensão do reuso	Em um projeto	Do ponto de vista do gerente de projeto e do gerenciador de reuso	No contexto de um departamento da empresa

Tabela 6-1 - Objetivos para métricas de reuso

Definidos os objetivos, para cada um definem-se uma ou mais métricas, correspondentes aos objetivos apresentados na Tabela 6-1. Essas métricas são apresentadas na Tabela 6-2.

Objetivo	Métrica	
O-1	M-11	Número de componentes no repositório
	M-12	Número de acessos (sem *downloads*) ao repositório, por período
	M-13	Número de *downloads* do repositório, por período
	M-14	Número de buscas por palavras chave, sem sucesso, por período
	M-15	Número acumulado de reuso por componente
	M-16	Número acumulado de reuso para todos os componentes no repositório
O-2	M-21	Nível de reuso por projeto
	M-22	Número de acessos (sem *downloads*) ao repositório, por projeto
	M-23	Número de *downloads* do repositório, por projeto

Tabela 6-2 - Métricas de reuso

6.2. Métricas do repositório

O objetivo O-1 define a necessidade do gerenciador de reuso monitorar a atividade de reuso através da utilização do repositório.

O número de componentes no repositório (métrica M-11) é usado para prever o potencial de reuso existente. Um número crescente indica que a produção de componentes para reuso está em pleno andamento, mas um valor muito alto não é isoladamente um indicador de sucesso. Estudos mostram que repositórios pequenos funcionam melhor que aqueles com enormes quantidades de componentes [Ezran02]. Além disso, se essa métrica for utilizada como o principal indicador de sucesso do programa de reuso, isso irá incentivar apenas a produção de componentes e não o seu reuso efetivo.

Os desenvolvedores acessam o repositório a fim de estudar a lista de

componentes existentes (métrica M-12) e possivelmente fazer o *download* de componentes para reuso (métrica M-13). Uma redução significativa em qualquer uma dessas métricas indica que algo não está funcionando bem no processo de reuso. Um valor baixo da métrica M-13 (número de *downloads*) combinado com um valor alto da métrica M-12 (número de acessos) indica que os componentes não estão sendo encontrados, ou que esteja faltando algum componente importante. Nesse caso o valor da métrica M-14 tende a ser alto também e deve ser monitorado em conjunto. Uma outra razão pode ser a falta de informação aos usuários a respeito do conteúdo do repositório e das funções disponíveis. Um número baixo de M-12 pode explicar um correspondente número baixo de M-13 devido à falta de motivação em se acessar o repositório em busca de componentes.

O número acumulado de reuso por componente (métrica M-15) e de todos os componentes (métrica M-16) são bons indicadores do sucesso do programa de reuso sob o ponto de vista técnico. O gerenciador de reuso deve prestar atenção nesses números e atentar para os componentes que não são reusados por um longo período, ou que nunca foram reusados. As razões podem ser várias: o componente está no fim de sua vida útil (nesse caso o gerenciador de reuso pode decidir removê-lo do repositório); o componente é novo e está no começo de sua vida útil (nesse caso é esperado que seu reuso aumente no futuro); o componente pertence a um domínio no qual projetos não estão sendo desenvolvidos no momento. Porém, se a grande maioria dos componentes nunca são reusados, pode haver um grave problema no processo de reuso, tanto do lado produtor quanto do lado consumidor. Considerando esses números, o gerenciador de reuso pode decidir manter ou não certos componentes no repositório, tendo em vista o passado de reuso do componente, sua expectativa de reuso no futuro, o custo de manter um componente no repositório e a política de se possuir grandes repositórios ou não [Ezran02].

6.3. Métricas do projeto

O objetivo O-2 se propõe a compreender como um projeto reusa os componentes do repositório e é de interesse tanto do gerenciador de reuso quanto do gerente de projeto.

O nível de reuso por projeto, ou fator de reuso, indicado pela métrica M-21, é definido como a razão entre o software reusado e o total de software produzido no projeto. Deve-se considerar que essa medida se aplica somente a código reusado, não considerando aqui que outros artefatos, como procedimentos de teste, também podem ser reusados no projeto. Para essa medida é importante definir exatamente como a quantidade de código será contabilizada. As medidas mais comuns são linhas de código e pontos por função. Não há uma escolha ótima, mas a forma definida deve ser usada em toda a empresa, por todos os projetos, para que se possa formar um histórico dos dados. Essa métrica deve ser computada ao fim do projeto com a ajuda de toda a equipe de desenvolvimento. Um baixo fator de reuso pode indicar tanto um problema do lado do projeto quanto do lado do repositório. Talvez a equipe do projeto não tenha se esforçado em procurar componentes no repositório, ou o repositório não contenha componentes úteis ao projeto. Para se chegar a uma conclusão da causa do problema, o valor da métrica M-22 (número de acessos ao repositório por projeto) deve ser utilizado. Um valor baixo indica que a primeira hipótese é verdadeira e é claramente um problema do projeto. A segunda hipótese não é um problema se o projeto é um caso especial, ou o primeiro dentro de um novo domínio, para o qual ainda não existam componentes disponíveis. Torna-se um problema se é um projeto típico da empresa. [Ezran02]

Capítulo 7

Riscos

No nível de projeto, o simples reuso tende a minimizar os riscos de projeto porque menos partes do produto necessitam serem codificadas pelo desenvolvedor e a qualidade dos componentes reusáveis é geralmente maior que a dos componentes ditos não-reusáveis. No nível organizacional, entretanto, um conjunto grande de riscos existe sobretudo devido à dificuldade em se prever quais e como os componentes serão reusados. Tais riscos, apresentados abaixo, existem e devem ser minimizados através da implantação de processos de reuso conforme visto no Capítulo 5.

- **nenhum componente reusável é produzido.** A produção de componentes reusáveis requer esforço adicional no sentido de identificar dependências externas, além do *design* e implementação com técnicas mais avançadas. Essas atividades devem ser diferenciadas do processo normal de produção de software, caso contrário corre-se o risco de não se produzir componentes reusáveis [Ezran02].

- **componentes existem mas não são encontrados.** Os componentes devem ser catalogados e armazenados de forma centralizada, sob o risco dos desenvolvedores não encontrarem o que procuram [Ezran02]. Esse risco aumenta proporcionalmente ao aumento do número de desenvolvedores e de componentes.

- **perde-se muito tempo para se compreender e avaliar o componente reusável.** Os usuários (desenvolvedores) devem ser capazes de avaliar

rapidamente os componentes (sob o ponto de vista técnico e funcional) a fim de decidirem se irão reusá-los ou não. Devem também possuir suporte para realizar tal análise, já que não necessariamente o usuário precisa entender os detalhes de implementação de cada componente. Para isso uma boa ficha cadastral do componente se faz necessária e, se possível, também o suporte dentro da empresa fornecido pelos responsáveis pelo componente reusável [Ezran02].

- **componentes não são reusáveis devido a sua baixa qualidade.** Uma vez produzidos, a qualidade e reusabilidade dos componentes devem ser garantidas aos futuros desenvolvedores. Isso implica em boa documentação (atualizada), adequação aos padrões da empresa e adequação a normas técnicas vigentes. Conforme previsto por William Frakes e representado na Figura 4-2, um processo de certificação para homologar o componente reusável pode se fazer necessário [Frakes96].

- **componentes não são reusáveis porque não atendem a requisitos funcionais.** Os componentes não devem ser criados para atender a um requisito específico, mas sim a um conjunto de requisitos que podem evoluir ou se alterar ao longo do tempo. Uma boa análise de domínio de aplicação dos componentes antecipa os requisitos funcionais comuns, diminuindo esse risco. Nesse caso uma arquitetura funcional comum pode ser implementada [Ezran02].

- **componentes não são reusáveis porque não atendem a requisitos técnicos.** O componente deve ser facilmente integrável à arquitetura técnica de uma nova aplicação. Nesse caso é imprescindível que se defina uma arquitetura técnica que seja comum a várias aplicações antes de se desenvolver um componente pensando no reuso. O uso de *guidelines* e *standards,* como ferramentas e normas técnicas padronizadas, ajuda a garantir um nível mínimo de consistência entre os requisitos técnicos de diversas aplicações [Ezran02].

- **a empresa não está preparada para o reuso, ou o reuso pode não ser apropriado para sua linha de negócios.** Antes de começar o desenvolvimento voltado ao reuso, a empresa deve definir se há um potencial suficiente para reuso e como o reuso será praticado nas diversas áreas da empresa. Nesse caso recomenda-se uma introdução progressiva do processo de reuso [Ezran02].

- **não é possível definir se o reuso está sendo vantajoso, ou se o desenvolvimento do reuso está no caminho certo.** No nível departamental é difícil determinar como o reuso está melhorando os negócios da empresa. Os processos de reuso, custos e benefícios devem ser monitorados a fim de permitir uma melhoria contínua no processo e nos investimentos em reuso, ou seja, um processo de gerenciamento do reuso deve ser definido [Ezran02].

- **necessidade de aplicar novos processos e técnicas e métodos avançados** (*design* de objetos, análise de domínio e *frameworks*, por exemplo) para se alcançar o reuso. Os times de desenvolvimento devem ter o suporte adequado a essas atividades. Um processo de suporte ao reuso é interessante a fim de centralizar os projetos, prover suporte técnico e gerenciar os componentes comuns [Ezran02].

- **desperdício de esforço.** A criação de um componente reusável custa cerca de 2 a 3 vezes mais que a criação de um componente não-reusável [Tracz95]. É a chamada “Regra dos Três”: antes de se receber algum benefício do reuso de um componente, deve-se reusá-lo pelo menos três vezes. Michel Ezran levantou estudos que apontam para um custo de desenvolvimento para reuso 50% maior em relação ao desenvolvimento não voltado ao reuso, porém afirma que o custo de reusar um componente chega a 70% do custo de desenvolvimento do mesmo componente, o que indica a necessidade de se reusar o componente no mínimo 5 vezes para que o reuso seja viável economicamente [Ezran02]. Poulin apresenta, em

seus estudos, valores variando de 1,1 a 3,7 como a quantidade de reusos necessários [Poulin96]. Mesmo que não se tenha certeza do número futuro de reuso, pode ser interessante desenvolver o componente, mas deve-se estar ciente de que isso terá um custo maior que o desenvolvimento de um componente não reusável, e que esse custo poderá nunca ser recuperado [Frakes94]

- **esforço mal direcionado.** Se existir um grupo separado para o desenvolvimento de componentes reusáveis, corre-se o risco do grupo chegar a desenvolver componentes que nunca sejam reusados. Exemplo: se o grupo tiver desenvolvido 4 componentes e tiver se enganado acerca dos requisitos e modelagem de 1 componente, de modo que este nunca seja reusado, então seguindo a Regra dos Três, mostrada no item anterior, cada componente deverá ser reusado pelo menos 5 vezes para que o custo seja compensado [McConnell96].

- **Evolução da tecnologia.** O próprio reuso planejado se torna um risco no sentido de que ele é uma estratégia de médio-longo prazo. Além de ter que reusar um componente várias vezes para que ele seja vantajoso, o componente deve ser reusado antes que a tecnologia no qual ele seja baseado se torne obsoleta. Num mundo onde a tecnologia evolui rápida e substancialmente, deve-se levar em consideração que o investimento num componente reusável hoje poderá se perder em cerca de 2 anos [McConnell96]. Esse risco se agrava se o componente tiver alguma dependência de algum hardware ou software específicos. Exemplos: uma biblioteca de componentes gráficos de jogos para o hardware existente há dois anos atrás pode ter pouca ou nenhuma utilidade hoje, considerando-se o hardware atual e as exigências de mercado. Um componente desenvolvido especificamente para o Microsoft FoxPro possui hoje um potencial de reuso praticamente igual a zero, considerando-se que o Microsoft FoxPro é um produto descontinuado e que novas aplicações de bancos de dados são desenvolvidas através de softwares mais modernos.

- **Superestimar os ganhos de tempo proporcionados pelo reuso.** Não se deve assumir que o reuso de código irá proporcionar grandes reduções de tempo no cronograma. Se somente código for reusado, somente o tempo de codificação será reduzido. Se o reuso for estendido a outros artefatos então outras etapas do processo podem ser reduzidas também. Deve-se observar que o reuso de um componente também demanda tempo para encontrá-lo e entender como usá-lo. Recomenda-se planejar um quinto do tempo para o reuso de um componente, comparado ao tempo que seria usado para desenvolvê-lo completamente [Tracz95].

- **Síndrome NIH.** O sucesso do reuso pode estar em risco caso o desenvolvedor seja acometido pela síndrome NIH, do inglês *Not Invented Here*, cujo sintoma é acreditar de forma irracional que o software desenvolvido por terceiros não é tão eficiente nem possui tão boa qualidade quanto o software desenvolvido pessoalmente [Norton03]. O software desenvolvido por terceiros deve ser avaliado e, caso passe nos critérios de aceitação, deve ser utilizado no projeto, pois seguramente os riscos inerentes ao desenvolvimento interno serão maiores que os riscos assumidos ao se adotar o software externo.

Como em todos os produtos de software que se adquire externamente, ao investir no reuso através da aquisição de componentes de software, deve-se atentar à qualidade do produto, em termos de funcionalidade e número de falhas, ao seu histórico no mercado e ao seu contrato legal, isto é, se nenhuma propriedade intelectual está sendo infringida ao se utilizar ou redistribuir o componente, e se o componente é plenamente confiável do ponto de vista da segurança, isto é, não seja um vírus ou não contenha nenhum software do tipo Cavalo de Tróia, capaz de utilizar recursos do sistema para fins ilícitos sem o consentimento do usuário [Tracz04].

O reuso de componentes pode melhorar a performance do sistema de

duas maneiras. Os componentes reusáveis são geralmente melhor projetados (possuem um *design* melhor) e mais otimizados que os componentes não-reusáveis, o que pode significar mais rápidos. Eles também permitem que os sistemas sejam montados mais rapidamente, permitindo que os testes se iniciem antes e fazendo com que os gargalos de desempenho sejam detectados mais rapidamente. Mas por outro lado, devido aos componentes reusáveis serem mais genéricos, eles podem às vezes ser mais lentos que os componentes não-reusáveis, sendo assim, é importante que testes de desempenho sejam feitos caso a caso, antes de se concluir que um componente seja rápido ou lento.

Em determinados projetos, dependo dos componentes a serem reusados, o sucesso do reuso está associado a ferramentas e linguagens de programação. Por exemplo, o usuário pode reusar uma ferramenta que gera rapidamente códigos de interface gráfica em Delphi ou reusar uma biblioteca de componentes de banco de dados escrita em C, mas não as duas ao mesmo tempo, assumindo-se que, nesse caso, a interoperabilidade entre Delphi e C não seja possível [McConnell96].

Componentes devem ser usados da forma como são, caso contrário não há vantagens no reuso. Comprar um componente pronto, ao invés de desenvolvê-lo completamente, deve ser analisado criteriosamente, pois se cria uma dependência da empresa com o fornecedor do componente. Além disso, os componentes existentes no mercado geralmente sofrem constantes atualizações, seja para corrigir falhas ou para incorporar novas funcionalidades, e substituir um componente existente por uma nova versão pode requerer a re-escrita de parte do código de integração ou a substituição de outros componentes que não sejam compatíveis com a nova versão [Ezran02].

Muito se questiona a respeito das vantagens do reuso oportunístico frente ao reuso planejado. O maior risco enfrentado pelo reuso oportunístico é superestimar os ganhos alcançados com a cópia simples e rápida de grandes partes de código. Reuso de 80% de código já existente não significa redução de

80% no cronograma. Além da codificação, existem no projeto as fases de análise, *design*, testes, documentação e suporte. Se isso não puder ser aproveitado, o reuso de 80% do código pode se transformar em redução de apenas 20% no esforço, e até menos em termos de redução do cronograma. Além disso, parte da atividade de reuso de 80% do código será descobrir quais 80% reusar e, como o código existente pode não estar preparado para o reuso, o desenvolvedor pode correr o risco de começar a analisar e corrigir falhas em cima de um código com o qual não está familiarizado, demandando tempo e esforço extras, podendo acabar anulando até mesmo a redução de 20% no esforço originalmente previsto [McConnell96].

O reuso não é gratuito. Somente pedaços triviais de código podem ser extraídos de um contexto e reutilizados em outro contexto sem nenhum esforço adicional. Para extrair um pedaço significativo de código de um produto existente e transformá-lo num componente reusável é necessário tempo, esforço e conhecimento técnico. Entretanto, o reuso de software mal implementado é pior do que o desenvolvimento independente de software (não voltado ao reuso) [Coata00]. Tempo e esforço serão despendidos em componentes supostamente reusáveis que serão eventualmente descartados porque o desenvolvedor ou não consegue usar o componente, ou não quer usar o componente. Uma vez descartados os componentes reusáveis, o desenvolvedor acaba desenvolvendo o código a partir do início e todos os envolvidos no processo acabam interpretando que o reuso de código é uma perda de tempo. Deve-se observar que reuso de software é uma prática compensadora, mas não é uma prática simples.

Enquanto que repositórios públicos na Internet de fato existem e contêm muitas classes, funções e aplicações úteis, disponíveis gratuitamente, a localização do componente correto, no tempo correto e por um preço acessível é uma tarefa complicada [Norton03]. Supondo que o desenvolvedor precise de uma biblioteca de componentes de rede para Unix, contendo funções de *sockets* que retornem um número do tipo *long* para ser integrado ao seu projeto, uma procura no *site* de pesquisas Google (http://www.google.com.br) pelas palavras “*Unix tcp*

socket long" retorna mais de 30 mil itens. Num outro exemplo, uma procura por "*.NET diagram component*" retorna mais de 650 mil itens. É sabido que tais buscas podem ainda ser refinadas para retornar resultados mais precisos, mas conclui-se claramente que a sobrecarga de informações na Internet se constitui num fator considerável na busca por componentes reusáveis na Internet.

7.1. Recomendações para o sucesso do reuso

- concentrar no reuso de componentes de um domínio específico. Se a linha de negócios da empresa se concentra em aplicativos financeiros, o programa de reuso deve se concentrar no desenvolvimento de componentes financeiros que sejam reusáveis [Frakes94]

- é melhor focar o esforço na criação de componentes pequenos, bastante especializados, ao invés de se tentar criar grandes componentes, super genéricos, tentando abranger o maior número de aplicações possíveis. O desenvolvedor de grandes componentes raramente consegue prever todos os requisitos futuros e quem olhar esses grandes componentes e ver que eles não atendem a todas as suas necessidades vai acabar desistindo de reusá-lo. Dados do Laboratório de Engenharia de Software da NASA sugerem que se um componente, para ser reusado, tiver que ser modificado em mais de 25%, então é mais barato desenvolvê-lo completamente [McConnell96].

- durante o desenvolvimento, focar em encapsulamento e abstração da informação, a base do *design* orientado a objetos. Estudos mostram que o simples uso de linguagens orientadas a objeto não faz com que o reuso fique mais fácil de ser alcançado. Muitos projetos bem sucedidos em termos de reuso foram feitos com linguagens tradicionais de programação, como C e Cobol [McConnell96].

- prover uma boa documentação sobre o componente, sobretudo acerca de

suas limitações, ou lista de falhas conhecidas. É normal que um software seja liberado com algumas falhas de baixa severidade, segundo o contexto em que ele é usado, mas no caso de componentes reusáveis, uma falha de baixa severidade pode se tornar de altíssima severidade se o componente for reusado num contexto diferente [McConnell96].

- desenvolver componentes praticamente livres de falhas. Se o desenvolvedor passa a encontrar falhas nos componentes a serem reusados, ele vai desistir de reusar e todo o programa de reuso da empresa pode ficar comprometido devido a falta de confiança nos produtos. Mesmo que o desenvolvedor corrija a falha, isso pode encarecer o custo de desenvolvimento de software pois, ao invés do desenvolvedor original estar depurando o componente, a empresa estará remunerando um outro desenvolvedor para fazer o mesmo serviço de forma bem menos eficiente, já que ele não está familiarizado com o código do componente [McConnell96].

- focar na qualidade dos componentes, e não na quantidade. É melhor possuir uma biblioteca pequena de componentes, e bastante confiável, do que uma grande biblioteca, com componentes de baixa qualidade [McConnell96].

Conclusão

Através da elaboração e conclusão desse trabalho conseguiu-se reunir informações suficientes para uma análise crítica em relação aos riscos e benefícios associados ao reuso de componentes aplicado no processo de desenvolvimento de software de uma determinada organização. Essas informações constituem a base para a elaboração de novas pesquisas e projetos em áreas específicas relacionadas a reuso de software, como repositórios, por exemplo.

Nesse trabalho foram levantadas características de componentes reusáveis, tecnologias que promovem a aplicação do reuso, modelos e funções de repositórios. Esses conceitos possuem muitos detalhes técnicos que irão influenciar no grau de reuso alcançado pela organização. Por isso recomenda-se que o corpo técnico envolvido em todo o processo de reuso, desde a criação de componentes até o gerenciamento de repositórios, seja bastante qualificado e tenha recursos para serem aplicados no estudo de tecnologias emergentes, de forma a se garantir o processo de reuso a longo prazo.

Dependendo da natureza do negócio da empresa, do tamanho de sua equipe de desenvolvimento e também de seus recursos financeiros, pode-se concluir que em algumas situações não seja vantajosa a implantação interna de um repositório de componentes. Devido à grande quantidade e variedade dos repositórios públicos existentes na Internet, muitas vezes a busca e armazenamento de informações nesses repositórios é a melhor opção. Por outro lado, o grande volume de informações na Internet pode dificultar a localização desses componentes. Cada empresa ou projeto deve analisar o seu caso especificamente.

Mesmo sendo possível mudanças radicais no processo de desenvolvimento de software de forma a se alcançar o reuso, não significa que essa seja a melhor decisão a ser tomada. Independentemente da ocorrência de

melhorias na qualidade e produtividade, com a redução de custos e do *time-to-market*, os reflexos nos negócios da empresa podem não justificar os esforços investidos e, no pior caso, os resultados podem ser negativos, quando o custo de implantação e manutenção do reuso se tornam maiores que os ganhos de capital proporcionados pelo reuso.

Um fator de sucesso absolutamente necessário em qualquer iniciativa de melhoria de processos, incluindo-se a implantação do reuso, é o comprometimento total tanto dos gerentes quanto dos desenvolvedores com relação ao sucesso do programa. Isso significa que todos devem estar cientes das mudanças, suas motivações e o retorno esperado. Percebe-se que é mais fácil alcançar esse comprometimento em pequenas empresas ou departamentos do que em grandes organizações e observa-se que geralmente o fracasso do reuso está associado ao comprometimento de apenas parte de uma equipe. Em função da necessidade desse comprometimento, a cultura do reuso deve estar sendo constantemente disseminada no ambiente em questão. Toda empresa possui uma rotatividade de pessoas e funções em maior ou menor grau, a entrada de novos membros numa equipe precisa ser acompanhada de um treinamento no processo de reuso, e a saída de um especialista em componentes, por exemplo, precisa ser reposta rapidamente.

No mundo corporativo observa-se que em determinadas áreas o reuso de componentes enfrenta muito mais barreiras políticas do que técnicas para a sua implementação. Há casos em que componentes são identificados e modelados para o reuso, mas não são reusados devido a restrições de cooperação entre equipes dentro da empresa. Muitos departamentos precisam garantir um *know-how* exclusivo para se manter dentro uma organização e não têm interesse em disponibilizar seus componentes a outros projetos. Esses fatores precisam ser identificados a fim de não ocultarem os benefícios do reuso e reforçam a importância do comprometimento da alta gerência, que deve promover a integração entre equipes de forma a garantir o sucesso do reuso.

Ao longo desse trabalho foram estudados conceitos importantes para a obtenção do reuso, como documentação de componentes, acesso ao repositório, processos de criação e atualização de componentes, novos papéis e responsabilidades, relatórios de métricas e adoção de novas tecnologias e ferramentas de desenvolvimento. Na prática observa-se que a manutenção desses conceitos muitas vezes traz uma carga excessiva de burocracia nas atividades do desenvolvedor, que passa a gastar mais tempo em atividades secundárias, não relacionadas ao desenvolvimento, e tende a se desmotivar ao longo do tempo. O *feedback* das pessoas envolvidas no processo de reuso é muito importante e, em alguns casos, a redução do controle do processo em prol de uma maior liberdade para o desenvolvedor tomar decisões técnicas a respeito do reuso é um fator relevante na redução dos riscos do projeto.

Na prática, durante a realização desse trabalho, observaram-se alguns casos em que o reuso de componentes se mostrou importante na melhoria da qualidade do software, proporcionando significativa redução na quantidade de falhas encontradas. Numa empresa A, havia o desenvolvimento de um grande projeto de software por uma equipe de desenvolvedores com diferentes especialidades. O desenvolvimento desse projeto seguia um modelo incremental, onde novas versões do software eram lançadas a intervalos de tempo regulares. Um grande problema enfrentado pela equipe, após o lançamento de cada versão, era o grande número de falhas recorrentes encontradas nos componentes do projeto, o que estava onerando os custos desse projeto para as versões seguintes, já que um número cada vez maior de desenvolvedores estava sendo alocado para resolver as falhas da versão anterior. Em função desse problema houve um estudo para se identificar a origem das falhas e se planejar uma estratégia de trabalho para as versões seguintes. Foram levantadas as seguintes informações:

- a maioria das falhas eram muito semelhantes, apenas ocorrendo em componentes diferentes;

- havia um grande número de componentes duplicados. As correções feitas num determinado componente não eram aplicadas nas réplicas desse componente;

- algumas falhas eram decorrentes do uso incorreto das interfaces dos componentes, devido à falta de documentação existente;

- algumas falhas eram decorrentes de funcionalidades de componentes implementadas de forma incompleta, devido à inexperiência do desenvolvedor no domínio de aplicação do componente.

Com base nessas informações, observou-se que o que estava ocorrendo era uma forma de reuso oportunístico, trazendo prejuízos ao desenvolvimento do projeto. A solução adotada então pela equipe do projeto foi:

- identificar os componentes do projeto que estavam sendo duplicados e modificá-los de tal forma que se tornassem componentes reusáveis;

- remodelar partes do projeto utilizando técnicas de orientação a objetos de forma que correções efetuadas em determinados componentes fossem replicadas automaticamente a todos os seus componentes derivados;

- para cada componente do projeto, definir um desenvolvedor responsável, já com experiência no domínio de aplicação daquele componente, de forma que se tornasse um especialista no componente, responsável pela sua correta documentação e o único autorizado a realizar correções e modificações no componente.

Após implantada, essa solução teve um impacto negativo na próxima versão do software, que foi lançada com atraso em função do retrabalho feito e também devido ao processo de adaptação dos desenvolvedores a suas novas funções e responsabilidades. Porém, nas versões posteriores desse projeto, observou-se uma redução significativa no número de falhas encontradas (da ordem de 80% em alguns componentes) e ganhos consideráveis no desempenho

do software (redução da ordem de 50% no tempo de execução de algumas funções). Observou-se que a estratégia de se utilizar desenvolvedores especializados produziu componentes reusáveis mais eficientes e com menos falhas. Nesse estudo de caso concluiu-se que a criação de componentes reusáveis, o uso apropriado de tecnologias voltadas ao reuso e o reuso de pessoas foram fatores importantes na melhoria da qualidade e redução de custos desse projeto da empresa A.

A implantação do reuso planejado é uma atividade arriscada, mas não implantar nenhuma forma de reuso também é arriscado. Tentativas mal sucedidas de implantar o reuso planejado custam recursos e tempo preciosos e podem tornar a gerência do projeto ou organização cética em tentar a implantação novamente. Mas se as empresas concorrentes conseguirem implantar o reuso de forma bem sucedida e a organização não, ela pode perder participação no mercado, ou até mesmo perder o mercado completamente devido à maior agilidade da concorrência em desenvolver novos produtos.

Conclui-se que o reuso é uma estratégia de médio-longo prazo no qual a empresa constrói uma biblioteca de componentes usados freqüentemente de modo a permitir que novos programas sejam construídos mais rapidamente a partir dos componentes existentes nessa biblioteca. Existem muitos riscos envolvidos nesse processo, mas quando suportado por uma gerência de processos de longo prazo, o reuso pode trazer grandes benefícios em termos de redução de prazos e custos, maior que outras práticas de desenvolvimento de software. Além disso, o reuso pode ser aplicado teoricamente por qualquer organização para qualquer tipo de software ou projeto de software.

Trabalhos futuros

Os seguintes temas são apresentados como sugestões de trabalhos futuros, baseados no estudo realizado sobre reuso de componentes de software.

Definição e aprimoramento de outras métricas de reuso

As métricas apresentadas podem ser detalhadas e refinadas a fim de se adequarem a processos em diferentes domínios de aplicação, bem como novas métricas para avaliação do reuso podem ser definidas de acordo com a política de reuso implantada na organização.

Análise da implantação de um repositório

Analisar o impacto da implantação de um repositório com relação à qualidade do trabalho, desempenho e motivação dos desenvolvedores. Para essa análise é importante dispor de dados quantitativos antes da implantação do repositório, e aplicar as mesmas métricas após a mesma equipe de desenvolvedores ter utilizado o repositório para o desenvolvimento de pelo menos três projetos. Como existem cinco fases distintas, esse trabalho é extenso e seus resultados são conhecidos somente a longo prazo. As fases desse trabalho compreendem:

- coletar dados antes da implantação do repositório;
- implementar o repositório;
- popular o repositório com componentes reusáveis;
- coletar os dados após a utilização do repositório por cada projeto;
- consolidar e avaliar o conjunto de dados disponíveis.

Integração do ambiente de reuso

Integrar o ambiente de reuso junto ao ambiente de desenvolvimento de software, proporcionando acesso simples e rápido a todas as ferramentas e documentos utilizados no processo de reuso. Exemplos:

- integrar o mecanismo de busca do repositório no ambiente de desenvolvimento tem a vantagem de apresentar os componentes disponíveis de forma imediata, sem a necessidade do desenvolvedor iniciar um processo separado, podendo perder eficiência devido à interrupção da sua linha de pensamento.

- o acesso a modelos de documentos e documentação do processo de reuso deve ser rápido e integrado ao ambiente de desenvolvimento, de forma a evitar que o desenvolvedor se utilize de modelos ou informações não originais simplesmente por estarem disponíveis mais facilmente.

Bibliografia

[ACE] ACE – The Adaptive Communication Environment. Disponível em http://www.cs.wustl.edu/~schmidt/ACE.html (Acessada em 19 de março de 2005)

[ACM97] Communications of ACM. Object-Oriented Application Frameworks, Vol. 40, No. 10, October 1997

[ADL] ADL – The Advanced Distributed Learning. Disponível em http://www.adlnet.org/ (Acessada em 02 de abril de 2005)

[Basili88] Basili, V. B.; Rombach, H.D. The TAME Project: Towards Improvement-Oriented Software Environments. IEEE Trans. on software engineering, 14(6), June 1988

[Bean03] Bean, James. XML for Data Architects - Designing for Reuse and Integration. Morgan Kaufmann Publishers. 2003

[Bellur97] Bellur, Umesh. The Role of Components & Standards in Software Reuse. Oracle Corporation. Workshop on Compositional Software Architectures. 1997. Disponível em http://www.objs.com/workshops/ws9801/papers/paper012.pdf (Acessada em 18 de fevereiro de 2005)

[Biggs04] Biggs, Peter. A Case Study in Small Company Software Reuse. Disponível em http://www.flashline.com/Content/Study/biggs.jsp (Acessada em 30 de março de 2005)

[Booch93] Booch, Grady. Object-Oriented Analysis and Design with Applications, 2nd edition. Benjamin Cummings, Redwood City. 1993

[Booch98] Booch, Grady; Jacobson, Ivar; Rumbaugh, James. The Unified Modelling Language User Guide. Addison Wesley, 1998.

[Bosch97] Bosch, Jan. Szyperski, Clemens, Weck, Wolfgang. Summary of the Second International Workshop on Component-Oriented Programming (WCOP'97) Disponível em http://sky.fit.qut.edu.au/~szypersk/WCOP97/Summary.html (Acessada em 27 de fevereiro de 2005)

[Braun] Braun, David; Silvis, Jeff; Shapiro, Alex; Versteegh. History of UML Kennesaw State University. Disponível em http://pigseye.kennesaw.edu/~dbraun/csis4650/A&D/UML_tutorial/history_of_uml.htm (Acessada em 03 de abril de 2005)

[Büchi] Büchi, Martin. Weck, Wolfgang. Compound Types for Java. Turku Centre for Computer Science (TUCS). Åbo Akademi University. Disponível em http://pllab.cs.nthu.edu.tw/cs5403/Papers/JavaBean/buchi98compound.pdf (Acessada em 08 de março de 2005)

[Coata00] Coata, Terry. To Reuse or Not to Reuse, That Is the Question. Open Text Corporation. E-ssentials, July 12, 2000, Issue 23. Disponível em http://www.spc.ca/essentials/jul1200.htm (Acessada em 19 de março de 2005)

[DCMI] DCMI – The Dublin Core Metadata Initiative. Disponível em http://dublincore.org/ (Acessada em 02 de abril de 2005)

[Eclipse] The Eclipse Project. Disponível em http://www.eclipse.org/eclipse/ (Acessada em 01 de abril de 2005)

[Ezran02] Ezran, Michel; Morisio, Maurizio; Tully, Colin. Practical Software Reuse. Springer Verlag 2002

[Facetas04] Facetas da Reusabilidade de Software. Disponível em http://www.dsc.ufpb.br/~jacques/cursos/map/html/intro/reusabilidade.html (Acessada em 20 de dezembro de 2004)

[Frakes94] Frakes, William B. Success Factors of Systematic Reuse. IEEE Software, September 1994.

[Frakes96] Frakes, William B.; Terry, Carol. Software Reuse: Metrics and Models ACM Computing Surveys, Vol. 28, No. 2, June 1996

[Gamma94] Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John. Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley 1994

[Guerra04] Guerra, Ana C.; Alves, Angela M. Aquisição de Produtos e Serviços de Software. Editora Campus. 2004.

[Guimaraes04] Um Modelo de Componentes para Aplicações Telemáticas e Ubiqüas. Guimarães, Eliane G. Tese de Doutorado. Faculdade de Engenharia Elétrica e de Computação. Unicamp. 2004.

[IntelligentLab] The Intelligent Software Agents Lab. Disponível em http://www-2.cs.cmu.edu/~softagents/index.html (Acessada em 29 de janeiro de 2005)

[Kirtland00] Kirtland, Mary. Projetando Soluções Baseadas em Componentes. Editora Campus 2000

[Klabunde99] Klabunde, Charles C. Um Estudo Sobre Componentes de Software. Curso de Pós-Graduação em Ciência da Computação – UFRGS – 1999.

[Klabunde00] Klabunde, Charles C. Tratando Frameworks como Componentes. Curso de Pós-Graduação em Ciência da Computação – UFRGS – 2000.

[Kotula98] Kotula, Jeffrey S. Using patterns to create component documentation. IEEE Software, March/April 1998.

[Krueger92] Krueger, Charles W. Software Reuse. ACM Computing Surveys, Vol. 24, No. 2, June 1992

[McConnell96] McConnell, Steve. Rapid Development: Taming Wild Software Schedules. Microsoft Press 1996

[Norton03] Norton, Robert J. Reuse of Personal Software Assets: Theories, Practices and Tools. Thesis. The Florida State University. College of Arts and Sciences. Disponível em http://etd.lib.fsu.edu/theses/available/etd-08282003-183319/unrestricted/01_rjn_thesis.pdf (Acessada em 01 de abril de 2005)

[OSGi] The OSGi Alliance. Disponível em http://www.osgi.org (Acessada em 10 de abril de 2005)

[Radding98] Radding, Alan. Hidden Costs of Software Reuse. Disponível em http://www.informationweek.com/shared/printHTMLArticle.jhtml?article=/708/08prhid.htm (Acessada em 21 de dezembro de 2004)

[RAS] RAS – Reusable Asset Specification Consortium. Disponível em http://www-306.ibm.com/software/rational/advantage/stnd-body.html#ras (Acessada em 01 de abril de 2005)

[Rational] IBM Rational Software. Disponível em http://www-306.ibm.com/software/rational/ (Acessada em 09 de abril de 2005)

[Poulin96] Poulin, J. S. Measuring Software Reuse. Addison Wesley 1996

[Rose] Rational Rose. Família de produtos para design e construção de software. Disponível em http://www-306.ibm.com/software/sw-bycategory/subcategory/SW710.html (Acessada em 18 de fevereiro de 2005)

[Rumbaugh91] Rumbaugh, James R.; Blaha, Michael R.; Lorensen, William; Eddy, Frederick; Premerlani, William. Object-Oriented Modeling and Design. Prentice-Hall. 1991.

[Schmidt02] Douglas C. Schmidt; Huston, Stephen D. C++ Network Programming, Vol. 2: Systematic Reuse with ACE and Frameworks. Addison Wesley 2002

[Silva96] Silva, Mônica F. da. Werner, Cláudia M. L. Packaging Reusable Components Using Patterns and Hypermedia. 4th International Conference on Software Reuse (ICSR '96) Disponível em http://csdl.computer.org/comp/proceedings/icsr/1996/7301/00/73010146abs.htm (Acessada em 17 de fevereiro de 2005)

[Silva00] Silva, Ricardo Pereira e. Suporte ao desenvolvimento e uso de frameworks e componentes. PPGC da UFRGS, 2000. Tese de Doutorado. Disponível em http://www.inf.ufsc.br/~ricardo/download/tese.pdf (Acessada em 18 de março de 2005)

[Sycara96] Sycara, K.; Decker, K.; Pannu, A., Williamson; M. and Zeng, D. Distributed Intelligent Agents. IEEE Expert, Dec., 1996. Disponível em http://www-2.cs.cmu.edu/~softagents/papers/ieee-agents96.pdf (Acessada em 01 de fevereiro de 2005)

[Szyperski03] Szyperski, Clemens. Component Software: Beyond Object-Oriented

Programming. 2nd Edition. Addison Wesley 2003

[Tracz90] Tracz, Will. The 3 Cons of Software Reuse. Proceedings of the 4th Workshop on Software Reuse. July, 1990.

[Tracz95] Tracz, Will. Confessions of a Used Program Salesman. Addison-Wesley 1995

[Tracz04] Tracz, Will. Software Reuse - Trick or Treat? Disponível em http://www.flashline.com/Content/Tracz/trick_or_treat.jsp (Acessada em 20 de dezembro de 2004)

[Vairagade] Vairagade, Mugdha. Developing Efficient Network and Distributed Applications with ACE. Disponível em http://www.developer.com/open/article.php/10930_3392461_1 (Acessada em 19 de março de 2005)

[WebDAV] WebDAV Resources. Disponível em http://www.webdav.org (Acessada em 10 de abril de 2005)

Apêndice A - Observer pattern

Conforme descrito no item 3.2 - Design Patterns, os *design patterns* são soluções documentadas para problemas comuns num determinado contexto. É apresentada a seguir, de forma resumida, a documentação do *Observer pattern* de acordo com o modelo proposto por [Gamma94] para a documentação de sua biblioteca de *patterns*.

A.1 Contexto

Definir um relacionamento de um para múltiplos objetos de forma que, quando um objeto altere seu estado, todos os objetos dependentes sejam notificados e se atualizem de forma automática.

A.2 Problema

Um efeito colateral bastante comum ao se dividir um sistema numa coleção de classes que interagem entre si é a necessidade de se manter uma consistência entre os objetos relacionados. Não é possível atingir essa consistência através do forte acoplamento entre as classes porque isso reduziria a sua reusabilidade. O *Observer pattern* descreve como obter essa consistência. Os objetos principais nesse pattern são o sujeito e o observador. O sujeito pode possuir um número qualquer de observadores independentes. Todos os observadores são notificados sempre que o sujeito sofre alguma mudança em seu estado. Em resposta a essa notificação, cada observador irá fazer uma consulta ao sujeito a fim de sincronizar o seu estado com o estado atual do sujeito.

A.3 Aplicação

O *Observer pattern* deve ser usado em alguma das seguintes situações:

- quando uma abstração possui dois aspectos, um dependente do outro. Encapsulando esses aspectos em objetos separados permite alterá-los e reusá-los de forma independente;

- quando uma alteração num objeto requer alterações em outros objetos e não se sabe *a priori* quantos objetos necessitam serem alterados;

- quando um objeto precisa notificar outros objetos sem precisar saber detalhes de implementação desses objetos. Em outras palavras, quando é necessário existir um fraco acoplamento entre os objetos.

A.4 Estrutura

O diagrama de classes que compõem o *pattern* é apresentado. A Figura A-1 descreve o relacionamento entre as classes que compõem o *Observer pattern*.

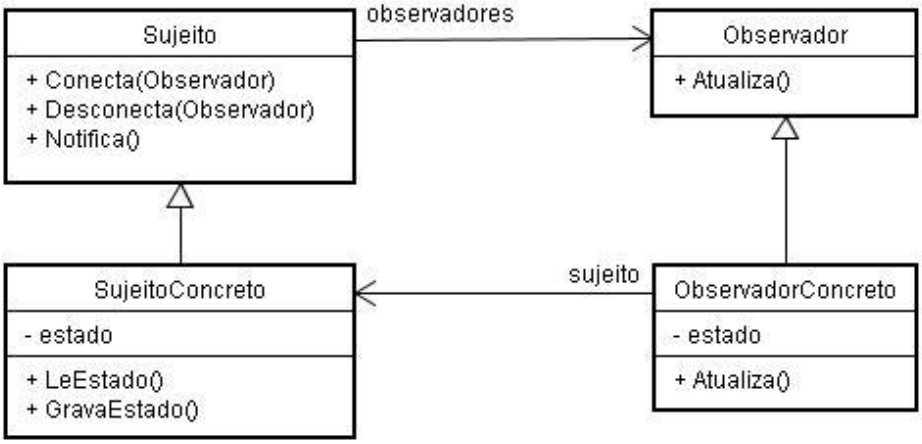

Figura A-1 - Diagrama de classes do *Observer pattern*

A.5 Participantes

É apresentada a descrição e a funcionalidade de cada classe constante na estrutura do *pattern*.

A.6 Colaboração

É apresentado um diagrama de seqüência que define a colaboração entre as classes definidas pelo *pattern*. A Figura A-2 descreve a interação entre as classes que compõem o *Observer pattern*.

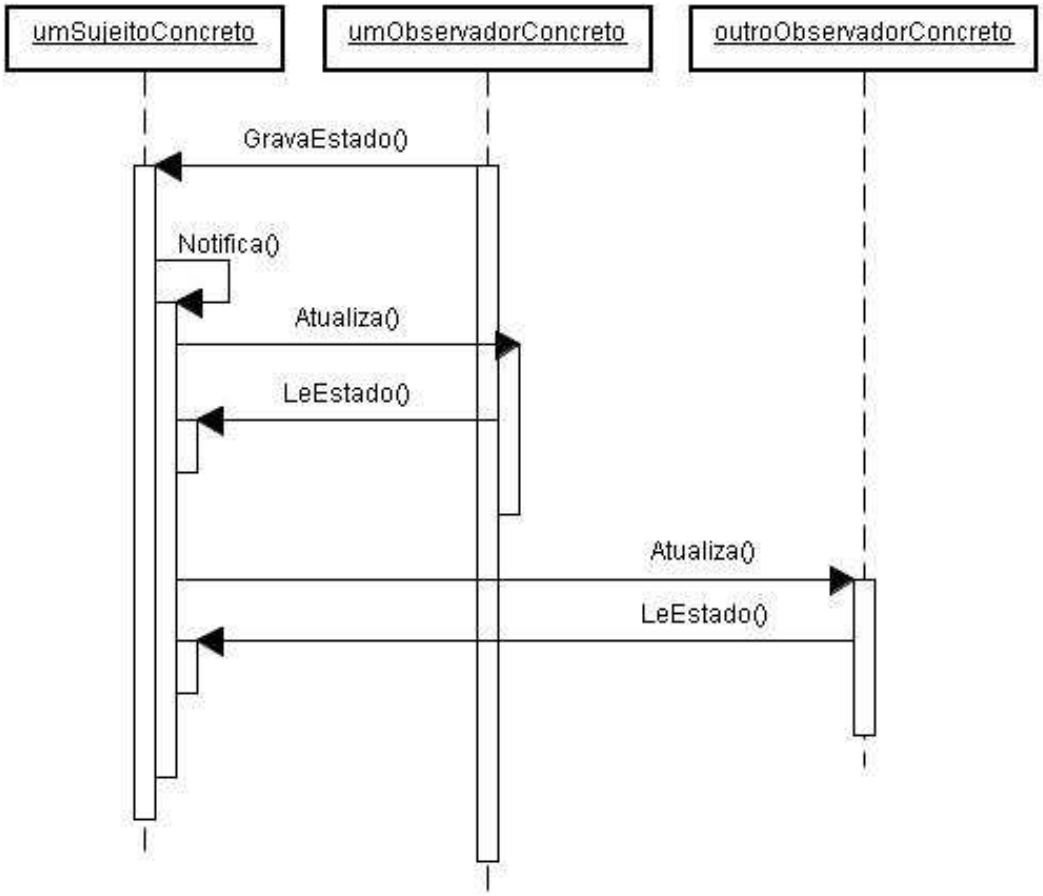

Figura A-2 - Diagrama de seqüência entre classes do *Observer pattern*

A.7 Vantagens

O *Observer pattern* permite a modificação de sujeitos e observadores de forma independente. É possível reusar o sujeito sem reusar seus observadores, e vice-versa. É possível também adicionar observadores sem modificar o sujeito ou outros observadores.

A.8 Implementação

Nessa seção são descritos todos os problemas ou questões importantes relacionadas à implementação e à dependência entre as classes do O*bserver pattern*.

A.9 Código exemplo

Nessa seção é fornecido um código exemplo para a implementação do *pattern*. No caso do *Observer pattern*, Erich Gamma fornece exemplos de código em C++ para a implementação das classes sujeito e observador, e apresenta o código exemplo de uma aplicação de relógio que se utiliza do *Observer pattern* para resolver o problema de atualização de horas e minutos na interface com o usuário [Gamma94].

A.10 Aplicações

Nessa seção são apresentadas aplicações já existentes que se utilizam do referido *pattern*. No caso do *Observer pattern*, Erich Gamma menciona o modelo MVC (*Model-View-Controller*) utilizado no ambiente Smalltalk [Gamma94].

A.11 *Patterns* relacionados

Referências para outros *patterns* são relacionadas. No caso do *Observer pattern*, Erich Gamma se refere ao *Mediator pattern* e ao *Singleton pattern* [Gamma94].