UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO

(Bacharelado)

FERRAMENTA DE SELEÇÃO E CLASSIFICAÇÃO DE COMPONENTES DE CÓDIGO EM DELPHI

TRABALHO DE CONCLUSÃO DE CURSO SUBMETIDO À UNIVERSIDADE REGIONAL DE BLUMENAU PARA A OBTENÇÃO DOS CRÉDITOS DE DISCIPLINA COM NOME EQUIVALENTE NO CURSO DE CIÊNCIAS DA COMPUTAÇÃO - BACHARELADO

JOSIANE GIANISINI

BLUMENAU, JUNHO/2000.

2000/1-35

REPOSITÓRIO DE COMPONENTES EM DELPHI

JOSIANE GIANISINI

ESTE TRABALHO DE CONCLUSÃO DE CURSO FOI JULGADO ADEQUADO PARA OBTENÇÃO DOS CRÉDITOS DA DISCIPLINA DE TRABALHO DE CONCLUSÃO DE CURSO OBRIGATÓRIO PARA OBTENÇÃO DO TÍTULO DE:

BACHAREL EM CIÊNCIAS DA COMPUTAÇÃO

Prof. Marcel Hugo -ORIENTADOR

Prof. José Roque Voltolini da Silva - COORDENADOR DO TCC

BANCA EXAMINADORA

Prof. Marcel Hugo

Prof. Everaldo Artur Grahl

Prof. Maurício Capobianco Lopes

DEDICATÓRIA

Dedico este trabalho a minha família, ao meu noivo e à Senior Sistemas em Turismo Informática Ltda., por tudo que fizeram por mim.

AGRADECIMENTOS

Agradeço aos professores de Bacharelado em Ciências da Computação da Universidade Regional de Blumenau, por todo conhecimento que adquiri ao longo do curso. Em especial ao professor Marcel Hugo pela orientação e incentivo na condução deste trabalho.

Aos amigos do ambiente de trabalho que ajudaram a crescer e a florescer o meu conhecimento acadêmico aliado a experiência profissional.

As pessoas que se dispuseram a ajudar e a aconselhar no desenvolvimento deste trabalho.

A minha família que sempre me incentivou e apoiou, e compreendeu a minha ausência devido aos compromissos acadêmicos.

Ao meu noivo, Ricardo Pereira de Oliveira, por me encorajar, incentivar e gerar uma auto-confiança capaz de enfrentar quaisquer obstáculos e medos, para seguir em frente e vencer.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE TABELAS

RESUMO

O trabalho tem por objetivo o estudo de técnicas de reusabilidade e a especificação e a implementação de um protótipo de um software – ferramenta de seleção e classificação de componentes de código - que será agregada ao Ambiente de Programação Visual Delphi, utilizando-se das técnicas de Orientação a Objetos.

ABSTRACT

This work is aimed to study of techniques of reusability and specification and implementation of a software prototype – tool for selection and classification of code components – that will be aggregated to Delphi Visual Programming Environment by using the Object Oriented Programming.

1. INTRODUÇÃO

1.1 ORIGEM

Desde longa data os desenvolvedores de software vêm sofrendo arduamente com o desenvolvimento de novos projetos - baixa produtividade e qualidade no desenvolvimento - e com a manutenção demorada e complicada que gera altos custos de desenvolvimento dos projetos já existentes ([MCC1993]).

Em função desses fatores e das grandes mudanças no contexto de desenvolvimento de sistemas, os desenvolvedores de softwares estão reavaliando os seus métodos e ambientes tradicionais onde vinham desenvolvendo seus projetos.

Muitos desenvolvedores estão pondo em prática o reuso de software, que segundo [KUT1997], é um dos caminhos mais eficazes para melhorar o processo de desenvolvimento, a qualidade e a consistência dos softwares, além de diminuir os custos com futuras manutenções. A prática de reutilização de softwares é considerada um dos maiores fatores para amenizar alguns dos problemas resultantes da crescente complexidade e da abrangência que os sistemas estão atingindo ([RAD1995]).

Existem diversas maneiras de se contextualizar a reutilização de componentes, porém todas elas têm como objetivo ressaltar os inúmeros benefícios trazidos pela reutilização. Dentre estes benefícios de acordo com [MCC1993], [CHE1994] e [FUR1995], pode-se enfatizar a redução dos riscos a falhas, redução dos custos de desenvolvimento, melhoria da qualidade do software, melhoria da produtividade, melhoria da manutenção, aumento de confiabilidade no sistema, redução do tempo de desenvolvimento, compartilhamento do conhecimento adquirido no planejamento do software, aceleração e simplificação do processo de desenvolvimento.

Para suprir essas necessidades, surgiu a idéia de se desenvolver uma ferramenta específica para ser agregada ao ambiente de programação visual Delphi, com a finalidade de gerenciar a reutilização de componentes de código fornecendo informações detalhadas sobre os mesmos, como por exemplo sua funcionalidade e seu contexto de inserção, e solicitando informações adicionais quando da criação de novos

componentes. Desta forma pode-se ter a melhor opção de um componente a ser reutilizado em um novo desenvolvimento, ou para a manutenção de um sistema já existente de acordo com as necessidades do projeto que está sendo desenvolvido.

A modelagem para a especificação do protótipo foi utilizada a *Unified Modeling Language (UML)* - largamente utilizada para modelagem de sistemas orientados a objetos - e para a implementação do projeto foi utilizado o ambiente de programação visual Delphi.

1.2 OBJETIVOS

O principal objetivo do trabalho é desenvolver uma ferramenta de seleção e classificação de componentes de código no Ambiente de Programação Visual Delphi.

Os objetivos secundários do trabalho são:

a) realizar um estudo sobre a reusabilidade e suas principais aplicações;

b) determinar as principais características de componentes reutilizáveis de código para o Ambiente Delphi.

1.3 ORGANIZAÇÃO DO TEXTO

A seguir serão descritos brevemente cada capítulo do trabalho.

Este capítulo de introdução apresenta uma visão geral da origem deste trabalho, sua relevância e objetivos.

O segundo capítulo apresenta a contextualização da reutilização de software, os seus benefícios, a avaliação da qualidade de software e os inibidores da reutilização de componentes.

O terceiro capítulo apresenta a contextualização de repositório, o conceito principal de repositório, a fundamentação de repositório em ferramentas CASE, os benefícios e o conteúdo do repositório, os tipos de informações reutilizáveis e, por fim, as principais tendências.

O quarto capítulo apresenta um estudo sobre os componentes reutilizáveis, sua criação, desejáveis propriedades, sua identificação, qualificação, classificação e seleção. Também neste capítulo é apresentado um projeto de reutilização de componentes.

No quinto capítulo é apresentado o ambiente de desenvolvimento Delphi, um conceito dentro do contexto do ambiente, de componentes e um estudo sobre as *packages.*

O sexto capítulo trata do desenvolvimento do protótipo, sua especificação, implementação e apresentação.

As conclusões do trabalho e algumas sugestões para futuros trabalhos na área de reusabilidade e aprimoramento do repositório, encontram-se no sétimo capítulo.

2. REUTILIZAÇÃO DE SOFTWARE

A Reutilização e Reengenharia de Software são técnicas que podem ser utilizadas para reduzir os esforços dispendidos no desenvolvimento e manutenção de programas. Em outras palavras, reutilização e reengenharia de software é a reaplicação de informações e artefatos de um sistema já definido, em outros sistemas semelhantes [PUJ1995].

A Reutilização e a Reengenharia de Software trazem como principais motivações o aumento da produtividade e a qualidade de software. A produtividade e qualidade são aspectos críticos no desenvolvimento de software. Desenvolvedores de software são cada vez mais solicitados a fazer mais com menos recursos, a entregar os sistemas solicitados em prazos menores, reduzir custos e tempo de manutenção, aumentar os níveis de desempenho e confiabilidade, aumentar a segurança dos sistemas e etc. Neste contexto são imprescindíveis mudanças significativas na forma como o software é produzido atualmente. Na abordagem do problema o aumento da qualidade e produtividade pode ser resumido em três pontos principais: otimizar a eficiência do processo, reduzir a quantidade de trabalho refeito e reutilizar produtos do ciclo de vida.

A Reengenharia de Software, também conhecida como “renovação e recuperação de software”, é o estudo e alteração de um determinado sistema para reconstituí-lo numa nova forma e subsequente implementação dessa nova forma. A Reengenharia de Software é a modificação em código e estrutura de dados existentes usando os princípios de engenharia de software atuais para aumentar manutenabilidade e adaptatividade do sistema. A Reengenharia geralmente inclui alguma forma de Engenharia Reversa (para se chegar a um nível mais alto de abstração), seguida de alguma forma de “Forward Engineering” ou re-estruturação. Define-se Engenharia Reversa como um processo capaz de extrair informações e documentos de um produto de software de tal forma que possa ser enquadrado num nível de abstração mais alto do que o código e que pertença as fases do ciclo de vida do sistema mais conceituais [LEI1992].

A reutilização de componentes de software existentes em novos sistemas implica numa menor produção de software novo, causando um aumento da produtividade bem como da qualidade e confiabilidade. O aumento da produtividade é devido a diminuição

do esforço necessário para a produção de código novo. O aumento da qualidade e confiabilidade advém do fato do código reutilizado já ter sido amplamente usado, modificado e testado em outros sistemas [COR1994].

Existem vários modelos para a reutilização de software, e que são classificados em modelos de composição e de geração. Modelo de composição é um programa composto a partir de componentes, constituídos geralmente por procedimentos codificados em alguma linguagem de programação, formando bibliotecas de componentes, ou então por classes, formando hierarquias de objetos [PUJ1995].

Os sistemas desenvolvidos segundo o paradigma de objetos são construídos como objetos complexos constituídos pela composição de outros objetos, cada um com propriedades e comportamentos próprios. É possível que esses objetos sejam projetados especificamente para a aplicação em desenvolvimento. Entretanto, a principal potencialidade do paradigma, para o incremento da qualidade e produtividade de software, reside nas suas facilidades para reutilizar o software, como instanciação ou especialização de classes predefinidas. Por isso, a maioria dos objetos que compõe as aplicações deveriam ser projetados e implementados reutilizando classes predefinidas numa biblioteca de software. A manutenção, tanto corretiva como evolutiva é simplificada pelo uso de classes padronizadas e pela total modularidade da arquitetura dos sistemas orientados a objetos. A maioria dos sistemas com suporte para reutilização de software promovem a reutilização de abstrações funcionais. Os conceitos do paradigma de objetos (objetos, classes e herança) constituem outro mecanismo de abstração de componentes de software que conjuga abstração de dados com herança de classes; é promovida a reutilização tanto de estruturas de dados como de funções [GIR1990].

O processo de reutilização de classes é abordado através de três subsistemas: um sistema de recuperação de classes, um sistema de avaliação de classes e um sistema de aquisição de conhecimento.

O sistema de recuperação e seleção de classes é suportado por dois mecanismos complementares, um mecanismo de recuperação sistemática, onde o usuário especifica descritores da classe procurada, da qual através desses descritores é recuperado um

conjunto de classes candidatas, e mecanismo de busca exploratória, onde o próprio usuário procura a classe inspecionando a hierarquia de classes da biblioteca.

No sistema de avaliação, as classes candidatas obtidas através do processo de recuperação sistemática e as classes selecionadas pelo mecanismo de busca exploratória são avaliadas pelo usuário, de forma a selecionar aquela mais adequada para a aplicação em desenvolvimento. O processo de avaliação é assistido por um browser de classes que permite ao usuário analisar a estrutura de informação na biblioteca de uma determinada classe.

No sistema de aquisição de conhecimento, o conhecimento do sistema é incrementado e/ou atualizado, em função da avaliação das classes candidatas recuperadas; dois mecanismos de aquisição de conhecimento são considerados: aquisição de conhecimento implícito (ocorre quando uma ou mais classes são recuperadas através do mecanismo de recuperação sistemática), e aquisição de conhecimento explícita (ocorre quando o mecanismo de recuperação sistemática falha) [GIR1990].

O esquema de classificação é a base de acessibilidade ao código, e está baseado em atributos de reutilização, sendo auxiliado por um mecanismo de seleção automático.

Para reduzir o esforço de entender e adaptar o código, é conveniente que se faça a classificação da coleção de componentes; se a coleção estiver organizada por atributos, que definam de forma unívoca os requisitos, a probabilidade de recuperar-se componentes não relevantes torna-se reduzida.

O processo de reutilização leva em conta a existência de um conjunto de especificações fornecido pelo programador e de uma Biblioteca de Candidatos à Reutilização, onde devem ser encontrados os módulos que satisfaçam as especificações; assim se existe um componente que satisfaça a todas as especificações, a reutilização se torna trivial; e se uma vez selecionada uma lista ordenada de candidatos similares, o usuário seleciona o mais fácil de ser reutilizado, e o passo seguinte é fazer-se a adaptação.

A classificação é o agrupamento de elementos similares, onde todos os membros do grupo compartilham pelo menos uma característica. Uma classificação mostra,

assim, uma relação entre elementos e entre classes de elementos. O resultado é uma rede de relacionamentos. Exemplificando, na definição de objeto, faz-se uma referência para a classe que contém o objeto sendo definido, e incorporam-se referências para as características que diferenciam este objeto de outros membros da mesma classe [PUJ1995].

2.1 BENEFÍCIOS DA REUTILIZAÇÃO

A reutilização de componentes de software é uma técnica de aproveitamento de informações produzidas durante o desenvolvimento de software anteriores, com o objetivo de reduzir o esforço necessário para o desenvolvimento de um novo projeto. A idéia geral da reutilização é reaproveitar componentes de software previamente desenvolvidos em novos projetos, reduzindo o tempo de desenvolvimento destes projetos e aumentando sua qualidade, produtividade, confiabilidade e segurança. Outros benefícios podem ser observados na manutenção e evolução de um produto de software [COR1994][BAR1996].

Mas a razão pela qual atualmente está havendo um aumento no uso da reutilização são os benefícios oferecidos segundo [MCC1993], que incluem :

a) redução do risco a falhas;

b) melhoria da qualidade do software;

c) melhoria da produtividade;

d) melhoria da manutenção;

e) os custo no desenvolvimento de software são reduzidos. Com a reusabilidade, menos componentes de software precisam ser especificados, projetados, implementados e testados;

f) aumento da confiabilidade no sistema;

g) tempo de desenvolvimento do software é reduzido. O reuso do software pode agilizar a prototipação de sistemas;

h) conhecimento adquirido em planejamento de sistemas de software pode ser compartilhado;

i) aceleração e simplificação do processo de desenvolvimento.

j) otimização da eficiência do processo;

k) redução da quantidade de trabalho refeito;

l) reutilização dos produtos do ciclo de vida.

2.2 AVALIAÇÃO DE QUALIDADE DE SOFTWARE

Foi demonstrado em [COR1994], um modelo para avaliação da qualidade de software em relação a reusabilidade de componentes, baseado nos seguintes conceitos:

a) objetos de qualidade: são as propriedades gerais, que o produto deve possuir;

b) fatores de qualidade: determinam a qualidade na visão dos diferentes usuários do produto – usuário final e outros;

c) critérios: são atributos primitivos, possíveis de serem avaliados;

d) processo de avaliação: determinam as métricas a serem utilizadas, de forma a se medir o grau de presença, no produto, de um determinado critério;

e) medidas: são o resultado da avaliação do produto, segundo os critérios;

f) medidas agregadas: são o resultado da agregação das medidas obtidas ao se avaliar de acordo com os critérios, e qualificam os fatores.

Este modelo pode ser utilizado para a avaliação da qualidade de produtos ao longo de todo o processo de desenvolvimento. Na Tabela 1, foi tratado exclusivamente da avaliação em nível de código, especificamente de atributos de qualidade relacionados à reusabilidade em componentes de código.

Considerando a estrutura do modelo para a avaliação da qualidade, na Tabela 1 estão identificados conforme [COR1994], os subfatores e critérios de qualidade, como fundamentais para a avaliação da reutilizabilidade de código.

Tabela 1 - Subfatores e critérios de qualidade

Subfator	Critérios	Descrições
Estilo	**Documentação Interna**	Refere-se a característica do código fonte do componente apresentar informações significantes através de comentários
	Organização Visual	É a característica de um componente ter uma boa apresentação quanto ao posicionamento de nomes, comandos, comentários, linhas em branco e na constatação de que foram utilizadas boas práticas de programação na sua implementação.
	Padronização	O componente obedece às normas e padrões estabelecidos pelo ambiente de programação da organização
	Programação Estruturada	O componente obedece às normas da técnica de programação estruturada.
Generalidade	**Independência do Tipo de Dados**	A aptidão de um programa operar com vários tipos de dados da linguagem utilizada.
	Independência de Qualidade de Dados	Característica de um programa não possuir restrições, para utilização de qualquer volume de dados.
	Independência de Compilador	É a codificação de um componente não incluir particularidades de um determinado compilador.
	Independência de Hardware	É o grau de independência do componente de software em relação ao hardware para o qual foi, originalmente, desenvolvido.
Maturidade	**Confiabilidade**	A probabilidade de um componente executar satisfatoriamente sua função durante um período de tempo.
	Número de Utilizações	Medida do número de vezes que o componente foi (re)utilizado.
	Vida útil	É o período de tempo entre cada (re)utilização do componente.

Simplicidade	**Complexidade**	Número de caminhos lógicos em um programa ou componente.
	Regularidade	É a razão entre a extensão estimada e a extensão real de um componente.
	Tamanho	Os componentes tem sua extensão compreendida entre valores mínimo e máximo estabelecidos como padrão.
Modularidade	**Fan-in**	Número de módulos superiores (chamantes)
	Fan-out	Número de módulos inferiores (chamados)
	Acoplamento	É a relação de interdependência entre componentes.
	Não Memorização	O componente não possui memória de existência prévia, executando, a cada ativação, como se fosse a primeira vez, ou seja, sem memória de estados anteriores.

Fonte: [COR1994].

2.3 INIBIDORES DA REUTILIZAÇÃO DE COMPONENTES

A falta de ambientes de desenvolvimento que ofereçam ferramentas para facilitar a reutilização é um dos principais empecilhos tecnológicos a reutilização. A dificuldade de se selecionar um componente de uma biblioteca, de entender o seu funcionamento e de modificar este componente podem inibir sua utilização. Outros fatores inibidores são falhas na gerência das bibliotecas de componentes e fatores não-tecnológicos, ligados à psicologia, aspectos econômicos, gerenciais ou legais [(BAR1996)].

3. REPOSITÓRIO

A seguir serão dados conceitos sobre repositório, porque o protótipo realizado, possui algumas das funcionalidades e alguns conceitos de repositório.

3.1 REPOSITÓRIO BASEADO EM FERRAMENTAS CASE

Segundo [MCC1993], a principal tecnologia de software nos anos 90 são as ferramentas de softwares baseadas em microcomputadores. A tecnologia CASE (*Computer-Aided Software Engineering*) introduziu e trouxe esta tecnologia à vanguarda. Mas, os desenvolvedores atuais esperam que as ferramentas CASE não atendam somente à fase do desenvolvimento do software, mas que unam e dêem um suporte automatizado a todas as fases do ciclo de vida do software. O repositório, em função das expectativas dos desenvolvedores, veio suprir essa necessidade, tornando-se uns dos componentes de maior importância em um ambiente integrado de ferramentas CASE. Também, pelo fato do mesmo ser a base para a integração das ferramentas de software e por gerar ganho da produtividade, quando do seu uso incorporado às ferramentas CASE.

Nos anos oitenta, ambientes de software e em particular ferramentas CASE foram vistos de fora para dentro, isto é, criou-se um apelo visual muito forte, dando ênfase na interface com o usuário e nas representações gráficas dos sistemas. Já nos anos 90, as perspectivas mudaram, as ferramentas CASE foram analisadas de dentro para fora, dando-se maior valor aos níveis de automatização de software, e por consequência para os repositórios [MC1993].

3.2 CONCEITO PRINCIPAL

De acordo com [MCC1993][MAR1996], o repositório é o **cerne** de uma ferramenta CASE integrada ao ambiente de desenvolvimento. Ele é a base para:

a) integração das ferramentas;

b) padronização das descrições e modelagens dos sistemas;

c) compartilhamento das informações dos sistemas;

d) reusabilidade do software.

O repositório é o mecanismo para definição, armazenamento, acesso e para a administração de toda informação de um projeto, isto é, seus dados e seu sistema. O repositório administra e armazena componentes de software, informações a respeito do software para serem compartilhadas e reutilizadas, fases do ciclo de vida, pessoas envolvidas no projeto desde os desenvolvedores até a parte administrativa [MAR1996][MCC1993].

O repositório é a chave para que a produtividade do software seja crescente, pois ele provê informação rápida e de forma fácil, na fase do desenvolvimento e da manutenção do projeto.

Antes do termo **repositório**, ser assim denominado, ele era conhecido como um dicionário de dados/banco de dados, ou como base de conhecimento ou até mesmo de enciclopédia. Independentemente de como era chamado, o propósito deste componente continua o mesmo [MAR1996][MCC1993].

3.3 BENEFÍCIOS DO REPOSITÓRIO

Em geral, um repositório aumenta a comunicação e compartilhamento da informação do software, das atividades das fases do seu ciclo de vida, dos usuários e das aplicações. Ele simplifica o desenvolvimento dos sistemas e melhora a qualidade da informação do sistema, controlando o acesso à informação e eliminando definições redundantes. Em suma, o repositório ajuda sobremaneira a melhorar a consistência, a validez e a integridade dos softwares [MCC1993].

Os benefícios do repositório que foram mais enfatizados por [MC1993] são:

a) simplificação da manutenção do software, devido aos componentes dos softwares e as suas informações serem administradas pelo repositório;

b) reuso de software, porque o repositório provê um catálogo dos componentes reutilizáveis, com meios fáceis de acessá-los e entendê-los, simplificando a manutenção e disponibilizando a reusabilidade de software para redução direta dos grandes custos do software.

A seguir estão relacionados os principais benefícios do repositório, segundo [MCC1993]:

a) compartilha informações do sistema através das aplicações, ferramentas e do ciclo de vida do sistema;

b) ambiente multi-usuário, integrado com todas as ferramentas de software;

c) aumenta a comunicação entre os usuários e o compartilhamento das informações;

d) consolida e elimina redundâncias dos dados incorporados;

e) aumenta a integridade do sistema;

f) simplifica a manutenção do sistema;

g) tem capacidade de combinar ferramentas de múltiplos vendedores;

h) reutiliza informação através das fases do ciclo de vida dos sistemas;

i) simplifica as conversões/migrações.

3.4 CONTEÚDOS DO REPOSITÓRIO

O repositório suporta todo o ciclo de vida do software, desde o planejamento até a manutenção, assegurando a informação e os componentes usados pelos sistemas durante a produção de cada fase do ciclo de vida. Provê o planejamento das informações necessárias da corporação – estrutura organizacional, metas e processos empresariais, - designs do software, incluindo modelos dos processos e dos dados, telas, definições de relatórios e algoritmos. Durante a fase da implementação, ele controla também os componentes do software (códigos de programa/módulos), seqüência dos dados, tabelas, arquivos e registros. Ajuda também na fase da manutenção do software, revalidando-os. Controla informações relacionadas com os objetivos dos projetos, horários, estimativas de custo e métricas de garantia da qualidade das fases do ciclo de vida, além do descritivo dos processos do ciclo de vida do software, como passos do processo e validações para o controle de qualidade dos processos [MCC1993].

A seguir estão relacionados os vários tipos de informações armazenadas no repositório [MAR1996][MCC1993]:

a) informação sobre a corporação (empreendimento);

b) informação que descreve o software em vários níveis de abstrações para cada fase do ciclo de vida;

c) informação que descreve o ambiente operacional para o sistema;

d) informação para a administração do projeto;

e) informação para a administração dos processos do ciclo de vida do software;

f) dados lógicos e modelos de processo;

g) definições físicas e linhas de código;

h) modelos do projeto;

i) dados empresariais;

j) regras empresariais;

k) relacionamentos;

l) validação de regras;

m) administração do projeto;

n) modelo do processo do ciclo de vida.

O repositório não contém apenas muitos tipos de informação, mas também contém as relações entre os vários componentes de informação e as regras para validação e uso destes componentes. Não é incomum para um repositório manter centenas de tipos de componentes de informação e as suas relações [MCC1993].

Cada componente de repositório é descrito em termos de suas propriedades ou atributos. As propriedades incluem tipicamente ID, pseudônimos dos nomes, tipos, uma narrativa, sub-componentes, tamanho, faixa de valores, edição e derivação das regras, tamanho, e idioma, como também informação de auditoria. As propriedades usadas para descrever um componente variam, dependendo do seu tipo [MCC1993].

Também, existe a preocupação de se armazenar a identificação do componente e o histórico das suas alterações. Por exemplo, quem criou o componente, o projeto no qual o componente foi criado, quando o componente foi criado originalmente, e quando o componente sofreu a sua última atualização [MCC1993].

3.5 INFORMAÇÃO REUTILIZÁVEL

Duas observações levam a uma ampla definição do que está sendo reutilizado na engenharia de software. Primeiro, muitas das motivações econômicas para a reutilização em programação se relacionam com uma redução nos componentes de trabalho que são necessários para fazer surgir um novo sistema de software. Segundo, geralmente todos concordam que a programação, restritamente definida, é apenas uma pequena porção do custo de criação de um sistema. Isto leva a definir o objeto de reutilização como sendo qualquer informação que um desenvolvedor possa precisar no processo de criação de software [FRE1987][FUR1995].

O interesse não está na reutilização de programas por muitos usuários finais em várias ocasiões (por exemplo, quando um sistema operacional é utilizado em várias máquinas diferentes ou um programa intenso de análise é utilizado muitas vezes para diferentes cálculos). Desta maneira, o foco está na informação necessária ao desenvolvedor (ou o mantenedor, que está acrescentado esta informação a um sistema já existente). Porém, esta definição é bastante ampla para auxiliar na compreensão e para impelir a uma pesquisa específica. Os tipos de informação que os desenvolvedores de software tipicamente necessitam, segundo [FRE1987][FUR1995], serão explicados a seguir.

3.5.1 FRAGMENTOS DE CÓDIGO

O código executável é, com freqüência, visto como um produto primário do programador, e sua reutilização eficiente é um dos objetivos mais antigos da tecnologia de software. As sub-rotinas permitem a reutilização do código e aparentemente, isto é o que muitas pessoas entendem por “reutilização”. Preservar o produto do código numa forma em que ele possa ser utilizado novamente, em novas situações (através do fornecimento de bibliotecas grandes e estruturadas de sub-rotinas), é a meta atual de muitas organizações.

As características essenciais desta área de reutilização são as seguintes [FRE1987][FUR1995] :

a) item a ser reutilizado é uma parte do código executável;

b) a definição da peça de código é freqüentemente específica da organização ou do sistema (por exemplo, utilizar um quadro específico de contas ou presumir que ele rodará num sistema operacional em particular);

c) centro organizacional está na redução do número de linhas do código que um programador deve criar, de maneira a construir uma nova aplicação;

d) qualquer parte única do código na coleção, que possa ser portavelmente reduzida, possui pouco ou nenhum significado operacional, sem estar agrupada com outras peças de código (portanto, utilizamos o termo “fragmentos”).

Já que existem diferenças pragmáticas óbvias, aqui não faz-se quaisquer distinções entre linguagem de alto nível ou formas diretamente executáveis de “código”. Na definição de código inclui-se estruturas internas de dados. Também deveria ser observado que a definição acima concentra-se em bibliotecas de pequenas peças de código; as peças maiores freqüentemente são reutilizadas, incluindo sub-sistemas completos, que são reutilizados como um complemento a alguns sistemas novos. É claro que esta forma de reutilização é bastante valiosa, de forma que a definição pode super-enfatizar o aspecto fragmentado. Porém, o aspecto central é o nível de descrição da informação que está sendo reutilizada, e não o seu tamanho.

3.5.2 ESTRUTURAS LÓGICAS

Este nível de informação normalmente é constituído pela arquitetura do processo e pela arquitetura de dados do software. Estas arquiteturas são caracterizadas através da identificação de suas partes (módulos, coleções de dados) e os relacionamentos entre estas partes (chamadas, passagem de parâmetros, inclusão). As partes são funcionalmente ou semanticamente caracterizadas, mas não estão ligadas a qualquer implementação específica. Freqüentemente, isto é chamado de **projeto interno** do sistema.

3.5.3 ARQUITETURAS FUNCIONAIS

O projeto externo de um sistema inclui aqueles aspectos vistos pelo usuário ou outros aspectos que não estão familiarizados com sua estrutura interna. Este nível de informação é normalmente uma especificação de funções e de objetos de dados - uma descrição destas entidades sem os detalhes de como elas são realizadas. Percebe-se dois tipos de arquiteturas funcionais que se relacionam especificamente aos objetivos de reutilização: coleções funcionais e sistemas genéricos [FUR1995].

Uma **coleção funcional reutilizável** é um conjunto de funções de aplicação que:

a) são compactadas como uma unidade;

b) todas pertencem a uma área de aplicação dada;

c) cobrem substancialmente a área;

d) podem ser utilizadas separadamente.

Pacotes matemáticos de sub-rotinas são exemplos de coleções funcionais reutilizáveis. O processamento de dados estatísticos ou um conjunto de testes padronizados pode, com freqüência, ser manuseado pela aplicação de uma coleção de funções para os dados.

Outras tarefas - por exemplo, o controle de um processo complexo de fabricação - pode exigir um sistema mais integrado de funções. Nesta situação, um **sistema genérico** pode ser suficiente para uma aplicação dada e, assim, diminuir a necessidade de um novo desenvolvimento. Um sistema genérico é caracterizado por:

a) ser constituído por um número de funções subsidiárias;

b) estar organizado de maneira que todas as funções, em conjunto, se apliquem a uma função única e principal;

c) ser aplicável a um número de situações diferentes, sem modificação do código.

Uma diferença principal nestas duas arquiteturas funcionais é que uma coleção omite a ordem de aplicação de funções, enquanto um sistema genérico a inclui.

3.5.4 CONHECIMENTO EXTERNO

Conforme [FUR1995], o próximo nível de informação que pode ser reutilizado é externo, porque é uma informação que pertence a coisas que estão fora da composição técnica do próprio software. Aqui, existem dois tipos diferentes de informação: conhecimento da área de aplicação e conhecimento de desenvolvimento.

O **conhecimento da área de aplicação** é tudo o que é conhecido sobre uma área efetiva, na qual se está utilizando um computador. Por exemplo, as leis científicas, os sistemas matemáticos e as regras de contabilidade, são modelos de conhecimento da área de aplicação. Em áreas experientes e bem compreendidas (tal como a matemática), o fornecimento de conhecimento da área de aplicação pode ser extensivo, bem registrado e facilmente reutilizado. Em outras áreas (tal como o gerenciamento de um hotel), o fornecimento de conhecimento pode ser amplamente informal, não-registrado e, conseqüentemente, difícil de ser reutilizado.

O **desenvolvimento do conhecimento** é a informação que se considera para o processo de desenvolvimento de um sistema de software. Isto pode incluir modelos de ciclo de vida, definições de um produto de trabalho, planos de teste e listas de verificação de garantia da qualidade. Freqüentemente, não se pensa a respeito da reutilização desta informação, mas conforme são formalizados os processos de desenvolvimento, existirão oportunidades crescentes nas quais tais itens poderão ser reutilizados.

Embora pareça um disparate, estes dois tipos de informação vem em conjunto no contexto de uma atividade de desenvolvimento. Visto de forma abstrata, o conhecimento da área de aplicação fornece a informação necessária para serem tomadas as decisões a respeito do projeto e as decisões a respeito do desenvolvimento das estruturas do conhecimento da seqüência daquelas decisões.

3.5.5 INFORMAÇÃO DO NÍVEL DE AMBIENTE.

Segundo [FUR1995], o nível mais alto de informação também contém dois tipos diferentes de informação e ambas lidam com o relacionamento de todos os níveis mais

baixos de informação, para categorias cada vez mais extensas de informação que não possuem nada de específico a fazer com o software.

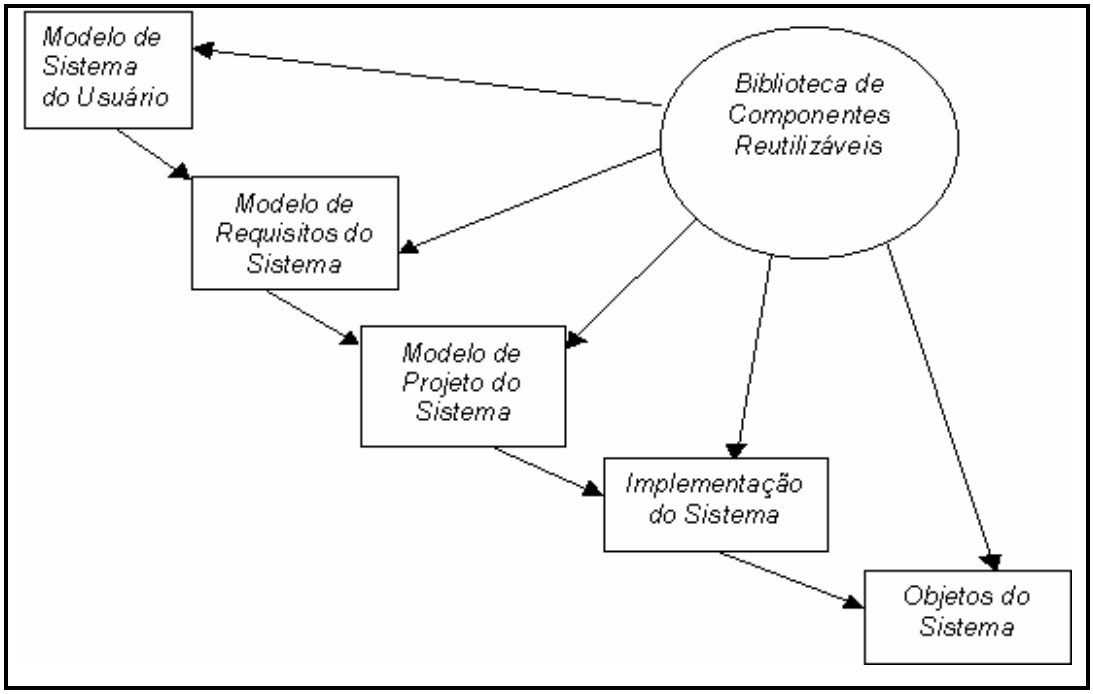
O **conhecimento de utilização** consiste daquilo que se conhece a respeito de como um sistema de software é utilizado na prática. Por exemplo, a informação sobre a maneira na qual um sistema de gerenciamento de um hotel se adapta dentro de toda a operação de um hotel e o conhecimento sobre a forma na qual uma teoria matemática particular é utilizada em alguns ramos da engenharia, são exemplos de conhecimento da utilização. A distinção entre o conhecimento de utilização e o conhecimento da área de aplicação pode ser mais de orientação do que qualquer diferença funcional. Porém, em muitas situações, pode-se distinguir os dois através da eliminação da informação do conhecimento da área de aplicação que descreve o relacionamento entre a aplicação e o ambiente no qual ela é utilizada. É claro que os dois tipos de informação devem ser utilizados em conjunto, mas para os propósitos de compreensão e captura, é mais vantajoso separá-los.

O **conhecimento de transferência de tecnologia** é aquilo que se conhece a respeito da transferência de novas tecnologias de software para aquelas pessoas e situações envolvidas com o desenvolvimento de software. De fato, é a utilização do conhecimento relativo ao conhecimento do desenvolvimento.

Observa-se que a comunicação entre as pessoas interessadas na reutilização deveria ser aumentada através de um conjunto comum de termos. Sugere-se que os termos baseados no tipo de informação que está sendo primariamente reutilizada poderia ser um esquema útil. Desta forma, poderia ser falado a respeito da **reutilização de código, reutilização do projeto, reutilização da especificação, reutilização do teste de plano, reutilização do modelo de aplicação**, etc.

A Figura 1 ilustra que se pode empregar componentes reutilizáveis em todos os níveis de abstração, percorrendo pelo modelo do uso do sistema ao sistema de objetos que executam em um computador. Componentes reutilizáveis podem reduzir o esforço exigido materialmente em todos os níveis, e pode até mesmo ser possível evitar o desígnio e implementação de todo o projeto [REE1996].

Figura 1 - Componentes reutilizáveis que ocorrem em todos os níveis de modelagem



Fonte: [REE1996].

A figura 1 ilustra vários oportunidades para quem planejou reuso:

a) um modelo do uso do sistema pode ser composto de padrões mais gerais que pode-se criar como parte do projeto atual ou que pode-se achar em uma biblioteca de componentes reutilizáveis.

b) um modelo de exigências de sistemas pode ser composto de padrões mais gerais criado como parte do projeto atual ou que pode-se achar em uma biblioteca de componentes reutilizáveis.

c) um modelo de design do sistema pode estar baseado em vários padrões ou estruturas achados em uma biblioteca de componentes reutilizáveis.

d) uma implementação de sistema pode ser derivada de uma ou mais classes ou estruturas achadas em uma biblioteca de componentes reutilizáveis.

e) um sistema de objetos pode ser composto de biblioteca predefinida de objetos.

3.5.6 TENDÊNCIAS

Segundo [MAR1996], os repositórios tenderão a uma crescente inteligência para armazenamento de uma grande quantidade de projetos reusáveis a partir dos quais procedimentos possam ser construídos. Muitos desenvolvedores usarão grandes repositórios centrais, bem como repositórios baseados em servidores de rede.

Bibliotecas de software serão construídas para diferentes classes de aplicações. Alguns exemplos de classes de aplicações que precisam de suas próprias bibliotecas são:

a) procedimentos comerciais;

b) aplicações financeiras;

c) projeto de sistemas operacionais;

d) navegação automática em bancos de dados;

e) linguagens de consulta;

f) projeto de circuitos;

g) controle de robôs;

h) análise criptográfica;

i) controle de mísseis;

j) arquitetura de construção;

k) desenho de gráficos;

l) projeto de Redes;

m) aplicações de controle de vôos;

n) CAD/CAM, projeto e manufatura auxiliados por computador;

o) controle de produção;

p) administração de projetos;

q) gráficos de apoio à decisão.

É essencial para o futuro do software que existam padrões para o repositório. Não ter nenhum padrão aberto para o repositório e suas interfaces com ferramentas seria

equivalente a não ter nenhum padrão para os CDs musicais. Um padrão aberto para o repositório I-CASE e suas interfaces é tão importante para o desenvolvimento de software quanto o padrão CD é para a indústria musical.

Padrões abertos devem incorporar o seguinte:

a) um repositório com seu conteúdo definido em termos dos tipos de objetos que ele armazena.

b) serviços de repositórios que usem métodos precisamente definidos para verificar a coerência e integridade das informações armazenadas no repositório.

c) controle de versão para gerenciar as distintas versões de objetos que são armazenadas.

d) serviços de ferramentas definindo os objetos que são criados ou modificados pelas ferramentas e depois armazenados no repositório.

e) formatos padrões para solicitações e respostas do objeto. Técnicas padronizadas para a interoperabilidade do objeto.

f) uma interface gráfica padrão para fazer as ferramentas CASE e seus diagramas parecerem semelhantes e fáceis de usar.

g) serviços de gestão de trabalho para possibilitar que os computadores de mesa interajam com o repositório de um servidor LAN ou mainframe.

h) uso total dos padrões existentes de sistemas abertos.

As ferramentas de desenvolvimento evoluirão e mudarão. Diversas ferramentas serão construídas por muitas corporações, freqüentemente pequenas, inventivas. Os repositórios devem ser compatíveis com todas essas ferramentas. Os repositórios corporativos já estão crescendo a uma taxa formidável e estão tornando-se um recurso estratégico vital, em alguns casos ajudando a corporação a manter-se à frente de sua concorrência. Os repositórios de componentes reusáveis também conterão uma grande quantidade de conhecimento. Se esses repositórios seguirem um formato padrão, eles poderão ser usados com muitas ferramentas diferentes de projeto.

4. COMPONENTES

4.1 CRIAÇÃO DE COMPONENTES REUTILIZÁVEIS

Com a utilização da reusabilidade, pode-se criar sistemas grandes em projetos pequenos. Componentes reutilizáveis são produtos criados por um fornecedor e aplicados por vários consumidores. O critério de sucesso para um componente reutilizável é que seu uso efetivo e a chave para o seu sucesso é a comunicação efetiva entre o fornecedor e os consumidores [(REE1996)].

Conforme [REE1996], projetos grandes são notoriamente difíceis para estarem certos. Eles são difíceis de planejar e controlar; eles são caros em tempo e recursos; e existem numerosos casos de insucessos. Em contraste, projetos pequenos são simples à plano e controle; e normalmente tem êxito; e os possíveis fracassos são baratos e fáceis de retificar. Mas, como projetos pequenos podem produzir resultados grandes? Uma resposta importante é o reuso. Um programa com atividade de cem meses pode ser reduzido a um mês apenas, com a criação de 99% em solução de componentes. Alguns dos grandes sucessos de tecnologia orientada a objetos são baseados em reuso, mas alguns dos fracassos também são da área do reuso. Todas as prósperas operações empresariais confiam/ram pesadamente em reuso. A primeira reação quando pede-se para resolver um problema ou produzir um resultado é procurar a experiência acumulada por soluções aplicáveis. Se é necessário produzir uma proposta de projeto, começa-se de uma proposta velha para um projeto semelhante. Se é necessário produzir um pedaço novo de código, procura-se soluções provadas a problemas semelhantes [REE1996].

Os empresários não aceitam problemas/falhas do reuso incidental. Eles querem formalizar a experiência deles e empacotar isto de tal modo que possa ser reutilizado confiantemente e constantemente. Eles criam procedimentos empresariais que descrevem modos de executar operações críticas; estabelecem bibliotecas de idéias já avaliadas, modelos e componentes de programa. Pode ser herdada uma experiência até mesmo em um programa de computação: uma proposta de projeto pode ser gerada automaticamente de parâmetros providos pelo usuário. O tema é planejar reutilizando componentes de objeto, por isso que os componentes reutilizáveis são criados com o

mesmo cuidado e dedicação como aplicações de usuário. Um componente reutilizável é um produto que resolve uma classe específica de problemas para uma comunidade de consumidores identificada. Como qualquer outro produto, a criação de um componente reutilizável leva um investimento significante em tempo e dinheiro que deve ser analisado em relação aos benefícios futuros [REE1996].

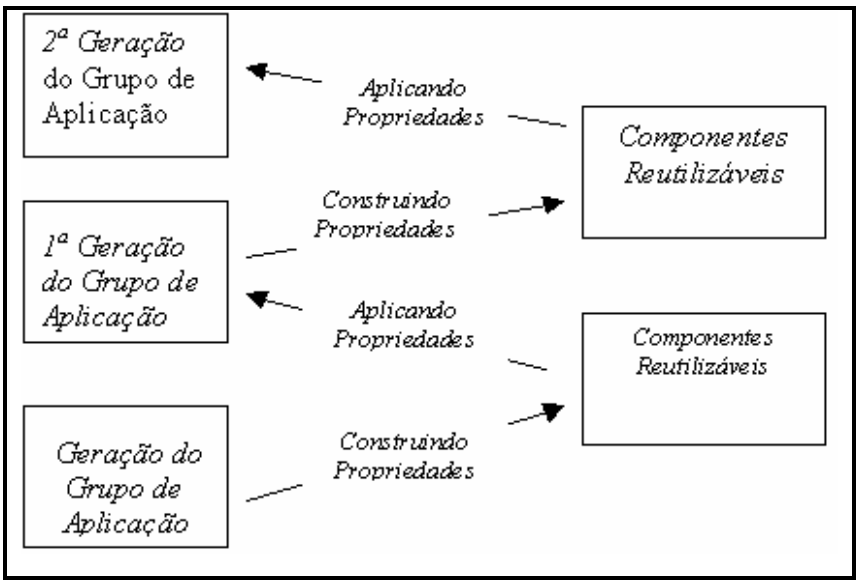
Existem várias vantagens para reuso planejado, como por exemplo, custo reduzido e tempo conduzido. Os componentes reutilizáveis devem ser conferidos, testados cuidadosa e completamente, assim o uso deles melhorará a qualidade de software e a consistência. Precisa-se freqüentemente proteger recursos críticos, como dados empresariais importantes e acesso a sistemas compartilhados. O reuso obrigatório de componentes pode ajudar a manter integridade de sistema, se eles incluem mecanismos que asseguram a aplicação correta deles.

A especificação de um componente reutilizável deve ser baseado em uma análise cuidadosa de soluções existentes criada pela comunidade de consumidores. Tenta-se identificar periodicamente os problemas (erros) e consertá-los, de modo que os mesmos não ocorram novamente. Tenta-se entender o comércio envolvido e criar uma solução geral à situação do trabalho do consumidor, e ter certeza que o novo componente será aceitável e verdadeiramente útil [REE1996].

A criação de um componente reutilizável não para na sua implementação. Cria-se uma solução inicial, usa-se, e continua-se aperfeiçoando-a.

Esta alternação entre uso e recurso de construção é ilustrado na figura 2.

Figura 2 - Uso alternativo de aplicações e melhoria de reusabilidade



Fonte: [REE1996].

Contato pessoal é o meio supremo para saber como comunicar-se tecnicamente. Profissionais que destacam competência complementar e que trabalham próximo ao consumidor, experimentam um processo de aprendizagem contínuo. Se os gerentes querem encorajar o aprendizado, eles formarão grupos com a finalidade de transferência de conhecimento, e eles recompensarão o desempenho do grupo em lugar de realizações individuais. Até mesmo se a comunicação verbal for o melhor canal, isto não siginifica ser suficiente. Documentação produzida cuidadosamente ajuda o consumidor a aplicar componentes reutilizáveis correta e efetivamente [REE1996].

4.2 PROPRIEDADES DESEJÁVEIS DE COMPONENTES REUSÁVEIS

Segundo [MAR1996], a seguir estão relacionadas uma série de propriedades desejáveis de componentes reutilizáveis, para que possibilitem qualidade elevada e rapidez no desenvolvimento:

a) base semântica formal: deve-se usar um formalismo que descreva precisamente o componente;

b) expressividade: o formalismo deve expressar todos os tipos de componentes possíveis;

c) facilidade de entendimento;

d) facilidade de adicionar ou eliminar detalhes;

e) projetado de maneira gráfica: para satisfazer as três propriedades anteriores, o projeto deve ser representado com a capacidade gráfica de um conjunto de ferramentas CASE que integre múltiplas representações diagramaticais;

f) interfaces claras, simples e precisas: o formalismo deve definir precisamente as interfaces entre os componentes;

g) independência de componentes (*self-contained*): os componentes devem ser independentes e ter um comportamento previsível;

h) isolamento da causas e do efeito: o componente deve executar seu comportamento independentemente de causas precedentes e efeitos subsequentes;

i) auto-organização: os componentes devem saber de quais outros componentes eles precisam, de forma que o programador não tenha de lembrar-se deles ou procurá-los;

j) verificável: técnicas para verificar o comportamento dos componentes devem ser usadas quando isso for prático;

k) flexível sem comprometer a eficiência: deve ser possível usar componentes padronizados sem causar problemas de desempenho;

l) independência da linguagem de programação: o formalismo deve ser independente de linguagem de programação;

m) padrões de aplicação: padrões devem ser usados tão amplamente quanto possível para a interface com redes, com bancos de dados, com sistemas operacionais, diálogo com o usuário final, operação cliente-servidor e assim por diante;

n) padrões de documentos eletrônicos: padrões para documentos eletronicamente representados devem ser usados – por exemplo, os padrões ANSI X.12/ISO, Edifact EDI;

o) protocolos definidos: devem existir protocolos para o uso de componentes, de forma que um construtor de software possa empregar componentes licenciados de diversos fornecedores.

4.3 IDENTIFICAÇÃO E QUALIFICAÇÃO DE COMPONENTES

O primeiro problema que se encontra é que o reuso de software surge da natureza do objeto de ser reutilizado. O conceito é simples - usar o mesmo objeto mais de uma vez. Mas com software é difícil de definir qual objeto faz parte de um contexto. Tem-se programas, partes de programas, especificações, exigências, arquiteturas, casos de teste, e planos, onde tudo se relacionam um com o outro. O reuso de cada objeto de software insinua o simultâneo reuso dos objetos associados a ele, e informações informais que viajam com os objetos. Assim, precisa-se reusar mais que código: objetos de software e seus relacionamentos incorporados a uma grande soma de experiência do desenvolvimento passado. Precisa-se reusar esta experiência na produção do novo software. A experiência torna possível reusar objetos de software.

Um segundo maior problema em reuso de código é a falta de um conjunto de componentes reutilizáveis, apesar da quantia grande de software que já existe na memória de muitos produtores de software. Eficiência em reuso e redução de custo requerem um catálogo grande de objetos reutilizáveis disponíveis [ARN1994].

[ARN1994] esboçou um modo para reusar experiência de desenvolvimento junto com os objetos de software produzidos. Focaliza-se um problema no desenvolvimento de um catálogo de componentes reutilizáveis: como analisar componentes existentes e identificar uns satisfatórios para reusar. Depois que eles são identificados, as partes poderiam ser empacotadas apropriadamente para reusar, e armazenadas em um repositório de componentes. O modelo de [ARN1994] para reusar partes dos componentes de software é o tradicional modelo de ciclo de vida em duas partes: uma parte, o projeto integra sistemas de software, enquanto a outra parte, o desenvolvimento, fornece objetos de software reutilizáveis para o projeto. As preocupações primárias do desenvolvimento são a extração e o empacotamento de componentes reutilizávéis,

trabalhando com um conhecimento detalhado do domínio de aplicação do qual um componente é extraído.

4.4 CLASSIFICAÇÃO DOS COMPONENTES

O reuso se torna impossível, segundo [MCC1993] quando não se têm absolutamente nada a reutilizar, ou quando se tem muitas coisas a reutilizar. Isto porque, o esforço despendido para localizar um componente reutilizável pode ser maior do que o esforço necessário para criar o componente a partir do nada.

A menos que se tenha uma maneira rápida e fácil de localizar componentes, o reuso jamais será posto em prática. Aparece aqui, mais um problema para se pôr em prática a reutilização. Um esquema de classificação apropriado para componentes reutilizáveis pode resolver tal problema.

O propósito de um esquema de classificação é proporcionar a localização e recuperação, fácil e rápida, de um componente apropriado da biblioteca onde estão armazenados e são gerenciados.

[MCC1993] propõe um esquema de classificação de componentes, onde cada componente armazenado para futuro reuso, é descrito em termos de descrições e classificações fixas. A classificação de um componente é um conjunto de cinco identificadores, incluindo o seu nome. São eles:

a) nome: nome do componente;

b) tipo: tipo do componente;

c) área de aplicação: é o tipo de família da aplicação, ou área empresarial para que o componente foi criado;

d) mídia: trata-se da especificação do componente, se é um código, um protótipo, uma especificação de projeto etc...

e) tecnologia: é o ambiente técnico utilizado para o desenvolvimento do componente.

A pessoa que realizar a busca por um possível componente a ser reutilizado, utiliza-se da classificação para localizar possíveis componentes que atendam a sua necessidade atual, atribuindo valores a cada um dos identificadores e procurando por esta especificação de componente na biblioteca de componentes reusáveis. Nesta biblioteca, segundo [MCC1993], cada um dos componentes apresentará um grupo de cinco características que o identificarão. A busca por um componente que atenda determinada necessidade se dá pela comparação das características fornecidas pela pessoa que deseja reutilizar um componente com as características de cada componente.

No caso desta proposta de classificação, conforme [MCC1993], o valor de cada característica é pré definida. Para isso um grupo de apoio à reutilização é responsável em defini-los, atualizar as listas de opções disponíveis e classificar cada componente quando o mesmo for inserido na biblioteca de componentes reusáveis.

Segundo [RAM1998][HAM1999], os esquemas de classificação são tipicamente baseados em três tipos de métodos de representação: biblioteca e informação científica, inteligência artificial e sistemas de hipertextos. Destes três tipos de métodos pode-se citar métodos que:

a) consideram textos livres: utilizando-se o método de texto livre, os componentes são descritos por palavras chaves ou frases. Esta descrição pode ser similar a utilizada na descrição do componente quando se está classificando-o . As palavras chaves que indexam termos e descrições podem ser extraídas diretamente da documentação do componente, ou diretamente da pessoa que o classificou. Este é o processo mais barato e rápido e pode ser automatizado pelas ferramentas utilizadas para fazer a documentação do componente. O problema deste esquema de classificação é que componentes que realizam funções diferentes podem ter a mesma descrição ou componentes que exercem a mesma função podem apresentar descrições diferentes.

b) consideram a classificação enumerada: Neste tipo de classificação os componentes são organizados em níveis hierárquicos. Este tipo de classificação entretanto é difícil de ser alterado, uma vez que ele é dividido em classificações exclusivas. Outra desvantagem é que a ambigüidade de componentes ainda pode existir dentro de diferentes hierarquias.

c) consideram a classificação de facetas: Em um esquema de classificação de facetas a classificação se dá através de um conjunto de facetas ordenadas. Uma faceta descreve várias propriedades dos componentes . Cada faceta possui um conjunto de termos e valores finitos que pode assumir. Por exemplo, para classificação de um componente em relação ao sistema operacional que o mesmo suporta, pode se ter os valores: DOS,WIN,OS2 ou ainda o valor nulo ou 0 se o componente não for aplicável a nenhum componente da lista. As vantagens deste esquema de classificação é que ele é fácil de ser modificado e o problema de ambigüidade é solucionado através do número limitado de facetas válidas. A sua desvantagem é que a lista de facetas deve ser criada e mantida.

d) consideram atributos - valores: este método de classificação é similar ao método de classificação das facetas mas ao invés de se utilizar de facetas ele se utiliza de pares de atributos e valores. Cada atributo do componentes é identificado por um número. Esta classificação é simples de usar e pode ser parcialmente automatizada. Um componente é então localizado na biblioteca pelo valor ou pela soma dos valores de cada atributo na classificação. A desvantagem é que não existem maneiras de controlar o valor dos atributos, diferentes valores podem ser usados para o mesmo atributo causando ambigüidade e redundância.

Estudos ainda não conseguiram determinar qual destes métodos é mais vantajoso. Isto significa que a organização deverá selecionar o método de acordo com suas necessidades, preferências e ferramentas disponíveis.

4.5 SELEÇÃO DOS COMPONENTES REUTILIZÁVEIS

A seleção dos componentes a serem reutilizados é a primeira etapa do processo de reutilização. Esta etapa visa encontrar na biblioteca os componentes que atendam a certos requisitos funcionais, necessários em um produto de software em desenvolvimento. Se a biblioteca não possuir um componente que atenda exatamente estes requisitos, o algoritmo de seleção deve retornar componentes que atendam a requisitos similares [BAR1996].

Diversas abordagens já foram exploradas para a seleção de componentes reutilizáveis, entre elas a classificação por palavras-chave, a classificação facetada, o processamento automático de linguagem natural e a descrição formal do comportamento dos componentes. Existe uma outra técnica de recuperação de componentes que está sendo recentemente pesquisada e utilizada por projetistas, a rede neural, baseada na camada auto-organizável de Kohonen e na classificação facetada. A rede é treinada com o objetivo de corrigir as relações de similaridade entre termos de facetas, aprimorando a precisão do processo de seleção evolutivamente, de acordo com o resultado de buscas anteriores [BAR1996] [HAM1999].

5. DELPHI

Será utilizado o ambiente de programação visual Delphi para o desenvolvimento do repositório. O Delphi é um ambiente que já possui uma biblioteca composta de pacotes (*packages*) de componentes comuns, que são incorporados, conforme a necessidade, aos projetos que são desenvolvidos[CAN1998].

O repositório será incorporado ao ambiente para armazenar, classificar e incorporar ao projeto pacotes de componentes e os componentes.

5.1 COMPONENTES DELPHI

Os componentes são os elementos fundamentais dos aplicativos Delphi. Ao escrever um programa, basicamente escolhe-se um número de componentes e define-se suas interações. Isso é tudo que existe na programação visual Delphi [CAN2000].

Existem diferentes tipos de componentes no Delphi. A maioria dos componentes está incluída na paleta de componentes, mas alguns deles (incluindo Tform e Tapplication) não estão. Tecnicamente, os componentes são subclasses da classe Tcomponent [CAN2000].

5.2 PACKAGES

5.2.1 INTRODUÇÃO

Conforme informação retirada em [DEL1997], o *package* é um arquivo que possui código executável que pode ser usado por mais de um processo rodando na máquina, ou que pode ser usado em separado de outro código executável. Em relação a um aplicativo Delphi, um package tem dentro dele várias *units*. Quem desenvolve o package escolhe quais units ele deve conter.

Packages são arquivos executáveis comuns, no formato DLL. A única diferença é a extensão, que ao invés de DLL é BPL, pois os packages foram feitos para serem usados por aplicativos Delphi apenas [DEL1997][CAN1998].

Os packages trazem os seguintes benefícios:

a) economia no espaço em disco necessário para colocar os arquivos executáveis.

b) em certas circunstâncias, ganho de memória para rodar os aplicativos.

c) maior agilidade para atualizar arquivos executáveis.

d) grande facilidade de um sistema atualizar as rotinas que ele utiliza.

e) dramática diminuição no tempo de compilação de um aplicativo.

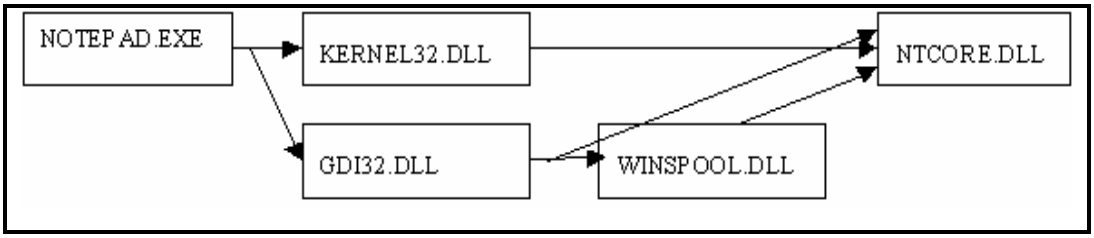
f) grande aumento na organização do código, na hora de criar um aplicativo ou uma rotina.

5.2.2 COMO FUNCIONAM OS PACKAGES

A grande maioria dos arquivos executáveis do Windows dependem de outros arquivos, também executáveis, chamados de arquivos de suporte (mas também podem ser chamados de bibliotecas). Os arquivos de suporte fornecem recursos que muitos aplicativos precisam, como desenhar coisas na tela, enviar bytes para a impressora, acessar o disco para que o aplicativo possa acessar arquivos, e outras coisas comuns a vários aplicativos. Cada uma destas funcionalidades, apesar de parecer simples, possui vários aspectos complexos. Por exemplo, desenhar uma janela com um título e com botões de maximizar e minimizar, exige uma série de comandos gráficos, como desenhar um retângulo preenchido, escrever o título usando uma fonte, desenhar os botões, etc. Como muitos aplicativos precisam fazer isto, é conveniente colocar toda esta funcionalidade no mesmo lugar, de maneira que os aplicativos não precisem fazer isto, e de maneira que o código deles fique menor. Portanto, existe uma biblioteca chamada GDI (*Graphics Device Interface* ou Interface para Dispositivo Gráfico), que possui esta programação. É comum encontrar executáveis que dependem da GDI [DEL1997].

Por exemplo: quando é executado o arquivo NOTEPAD.EXE, a primeira coisa que o Windows faz é carregar os arquivos dos quais o NOTEPAD.EXE depende, sendo eles “KERNEL32.DLL” e “GDI32.DLL” . Somente após isto ter sido feito, o Windows inicia a execução do Notepad. O GDI32.DLL também é um arquivo executável, e também pode depender de outros arquivos de suporte. Portanto, os arquivos das quais o GDI32.DLL depende, também são carregados quando o NOTEPAD.EXE é executado. Isto é feito de forma recursiva, e permite a utilização de vários arquivos, conforme a figura 3.

Figura 3 – Exemplo de execução do aplicativo NOTEPAD.EXE



Do ponto de vista do sistema operacional, os *packages* nada mais são do que arquivos de suporte.

A utilização de *packages* força a organização do código. Outra grande vantagem é que se o sistema tiver um erro, e ser for detectado que o erro é no *package* “MASBAH.BPL”, basta corrigir este *package*, e enviá-lo ao cliente. Não é necessário sequer compilar o sistema novamente, o que representa uma grande economia de tempo [DEL1997].

5.3 REGRAS PARA CRIAÇÃO E MANUTENÇÃO DE PACKAGES

As seguintes regras devem ser seguidas para criar ou utilizar *packages* [DEL1997]:

1. Se um *package* **A** utilizar um *package* **B**, o *package* **B** não pode utilizar o *package* **A**. Ou seja, não são permitidas referências circulares entre os *packages*. Não há sentido nisto ocorrer, pois sempre que qualquer um dos *packages* for carregado, o outro seria carregado também. Ou seja, os dois

packages iriam funcionar como um só. Eles apenas estariam em arquivos separados, e isto não representa grandes benefícios. Por isso, o Delphi não permite que esta situação ocorra. Também não são permitidas referências circulares indiretas (por exemplo: **A** usa **B**, **B** usa **C** e **C** usa **A**). Quando ocorrerem referências circulares, é necessário criar um *package* apenas, que contenha as *units* que estão em **A**, e também as units que estão em **B**. Se o programa estiver muito desorganizado, esta situação será comum, e vão surgir poucos e grandes *packages*, o que iria acabar com os principais benefícios dos *packages*.

2. Durante a execução, dois *packages* não podem conter a mesma unit. Por exemplo, se a unit CADASTRO.PAS estiver nos *packages* **A**.BPL e **B**.BPL, os dois não podem ser carregados ao mesmo tempo durante a execução. Não faria sentido permitir que dois *packages* tenham uma mesma *unit*, pois o código estaria duplicado em memória. Também não é permitido que dois *packages* tenham *units* com o mesmo nome, ainda que elas não sejam iguais. Isto acontece porque o nome da *unit* serve para identificar a funcionalidade dela. Esta regra força uma certa organização nos programas. No entanto, existem situações em que a mesma *unit*, ou *units* com o mesmo nome, estejam contidas em *packages* diferentes. Isto permite um eficiente controle de versões dos *packages*. O importante é que mesmo que isto ocorra, os *packages* jamais poderão ser carregados ao mesmo tempo.

3. Um *package* deve ser 100% compatível com versões anteriores. Seja o *package* "**A**". Seja um arquivo **A**.BPL na versão 15, e outro arquivo **A**.BPL na versão 34. Caso o arquivo **A**.BPL 34 seja simplesmente copiado por cima do arquivo **A**.BPL 15, deve-se garantir que qualquer aplicativo que utilizava o **A**.BPL 15 continue funcionando, mesmo utilizando o **A**.BPL 34. O critério para compatibilidade é o nome do arquivo. Isto significa que sempre que for dado manutenção em um *package* qualquer, não é permitido retirar um recurso, ou alterar a funcionalidade padrão do *package*. Nos casos em que isto for necessário, deve-se criar um outro

package, com um nome diferente do atual. No exemplo dado, caso seja necessário retirar um método do *package* "**A**", deve-se criar um *package* "**B**", sem o método. Para isto, será necessário copiar as *units* do *package* "**A**" para o *package* "**B**", e deve-se manter os dois conjuntos de *units*, um para cada *package*, ainda que os dois contenham as mesmas *units*, ou *units* com o mesmo nome. Isto é preciso, pois no futuro pode ser necessário efetuar correções ou alterações no *package* "**A**". Mesmo assim, os *packages* "**A**" e "**B**" jamais poderão ser utilizados ao mesmo tempo.

5.4 COMO CRIAR PACKAGES

Abaixo estão relacionados alguns arquivos usados no manuseio de *packages* [DEL1997][CAN1998]:

a) arquivo com extensão DPK (sigla de *Delphi Package*). Este arquivo é o arquivo-fonte do *package*, que pode ser editado pelo Delphi ou por um editor de textos qualquer.

b) arquivo com extensão DCP (sigla de *Delphi Compiled Package*). Este arquivo é usado pelo Delphi para compilar um projeto ou um outro *package* que utilize o *package* em questão.

c) arquivo com extensão BPL (sigla de *Borland Package Library*). Este é o arquivo de suporte que o Windows procura quando o aplicativo for executado. Ele é o único arquivo que precisa existir na máquina quando o aplicativo estiver rodando.

Em primeiro lugar, o *package* é um conjunto de *units* compiladas no mesmo arquivo executável (que tem a extensão BPL). Isto significa que é necessário ter os arquivos das *units*. Isto inclui os arquivos DCU ou PAS, e os arquivos adicionais (DFM, RES, OBJ, etc.). É fortemente recomendado ter estes arquivos no mesmo diretório do arquivo DPK.

Em segundo lugar, qualquer *package* que será criada usará um ou mais *packages* que já existem. Para compilar o *package*, é preciso ter os arquivos DCP de cada *package* que o *package* usar, no caminho da Biblioteca. O caminho da biblioteca pode ser visto ou editado no Delphi, na opção "*Tools*", "Environment Options", na guia "*Library*", no

campo "*Library Path*". Para saber quais *packages* podem ser usadas, deve haver em algum lugar uma documentação que mostre a lista dos *packages* e seu conteúdo.

Em terceiro lugar, sempre que o *package* for compilado, o Delphi vai gerar um arquivo com extensão DCP, para ser usado em compilações dos projetos que usam o *package*. Este arquivo será gerado no diretório especificado no campo "DCP output directory", que está junto com o campo "Library Path". É fortemente recomendado que o diretório especificado em "DCP output directory" esteja presente também na lista de diretórios especificada em "Library Path".

Após a configuração dos diretórios, pode-se criar e compilar *packages*. Os passos para a criação de um *package* são estes [DEL1997][CAN1998]:

a) pensar em um bom nome para o *package* que se deseja criar.

b) criar ou obter um diretório vazio para o *package*, que tenha o mesmo nome do *package*.

c) identificar quais *units* estarão dentro do novo *package.*

d) copiar os arquivos para o diretório criado no item “b”.

e) identificar quais *packages* serão usados pelo novo *package.*

f) no Delphi, usar o comando "File", "New...", e escolher a opção "*Package*". O Delphi vai exibir o editor de *packages*.

g) selecionar "Contains", e apertar no botão "Add", para adicionar as *units* que o novo *package* vai conter.

h) selecionar "Requires", e apertar no botão "Add", para adicionar os *packages* que o novo *package* vai utilizar.

i) salvar o *package*, usando a opção "File", "Save".

j) o *package* está criado, e agora o mesmo poderá ser compilado e utilizado em outros projetos.

5.5 AMBIENTE PARA COMPILAÇÃO, DEPURAÇÃO, E LIBERAÇÃO DE PACKAGES

O trabalho com *packages* exige que o desenvolvedor conheça os conceitos acima listados.

Como o trabalho com *packages* usa mais conceitos do que o trabalho com apenas as *units*, o trabalho com *packages* tem mais suscetibilidade a provocar erros. Mas estes erros só ocorrem se o desenvolvedor que estiver trabalhando com *packages* não tiver competência para entender e utilizar os conceitos envolvidos. Se o desenvolvedor tiver esta competência, o trabalho com *packages* só trará benefícios.

Um problema que contribui para que os desenvolvedores fiquem confusos é que o próprio Delphi não possui uma estrutura adequada para o trabalho com *packages*. Os arquivos VCL??.DCP, por exemplo, ficam junto com as *units* que estão contidas nos *packages*. E ainda, o Delphi mistura *packages runtime-only* com *designtime-only*.

Segue uma lista de afirmações corretas sobre a utilização de *packages* [DEL1997][DEL2000]:

A) **Packages são arquivos executáveis simples.**

O *package* é um arquivo executável, e portanto, precisa de um projeto separado. Isto significa que um *package* precisa ter um diretório só dele, com as *units* que ele contém, com o arquivo DPK dele. As *units* de um *package* devem estar junto com o arquivo DPK (que representa o projeto do *package*). Não há qualquer motivo em especial para que as *units* estejam em outro local, muito menos no diretório "Library Path" (ou Lib).

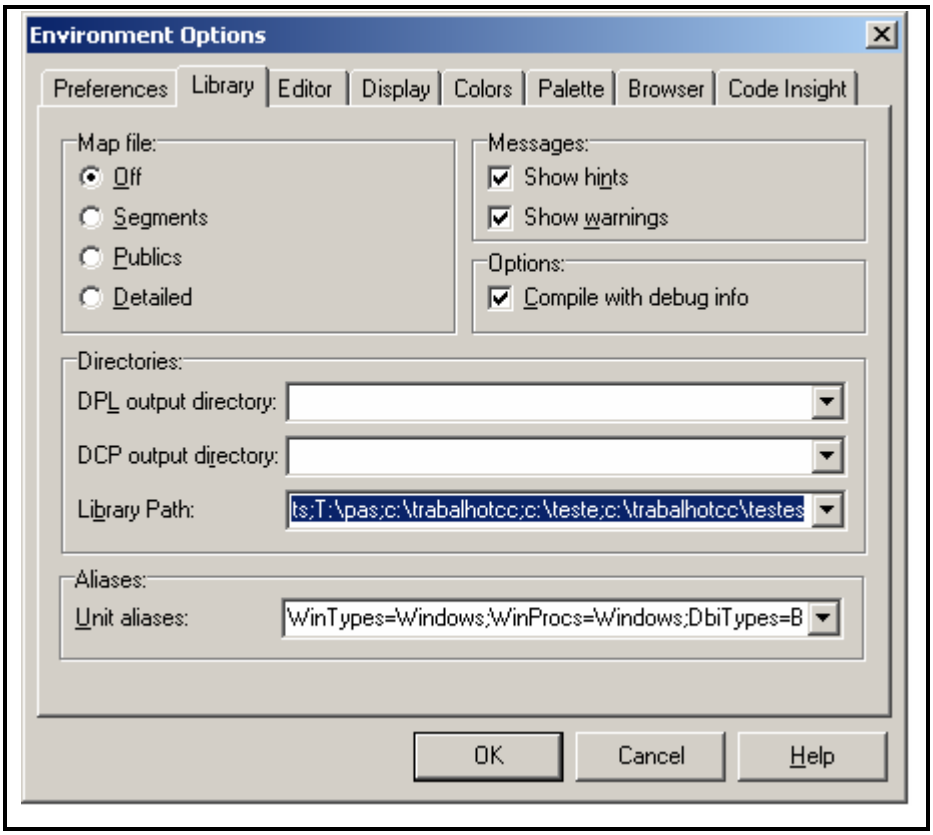
B) **Packages podem ser usados por outros projetos.**

Os *packages* são arquivos executáveis simples, mas podem ser usados por outros executáveis. Quando um outro projeto utilizar um *package*, o compilador vai precisar do DCP do *package*. Por exemplo, se o projeto FOLHA.DPR usar o *package* SYS40, quando o FOLHA.DPR for compilado, o compilador vai precisar do arquivo SYS40.DCP. E onde o compilador vai procurar os arquivos DCPs? Resposta: No mesmo lugar onde estariam as *units*, ou seja, no "Library Path", conforme figura 4. Como fazer com que o DCP esteja em um diretório apontado pelo "Library Path"? Existem duas formas:

1) colocando o diretório do *package* dentro do "Library Path".

2) colocando o arquivo DCP do *package*, dentro de um diretório que já esteja no "Library Path".

Figura 4 - Library Path



A segunda solução é muito superior, por dois motivos: primeiro porque quanto menos for necessário mexer no parâmetro "Library Path", melhor. Segundo porque caso uma *unit* esteja presente em dois pacotes, o Delphi pode escolher o arquivo-fonte errado para exibir no editor. Para colocar um DCP em um diretório qualquer, basta especificar este diretório em "DCP output directory", ou usar o parâmetro /LE, do aplicativo para compilação de packaegs, denominado DCC32.

C) **O parâmetro "Library Path" só precisa ter um diretório.**

Muitos erros de compilação e confusão com arquivos-fonte são gerados por alterações incorretas no parametro "Library Path". Geralmente, este parâmetro aponta para mais de 200 arquivos-fonte, que podem estar repetidos em vários diretórios

especificados em "Library Path". A alteração deste parâmetro não causa muitos problemas quando não se usa *packages*, mas quando os *packages* são utilizados, muitos problemas podem surgir.

É absolutamente desnecessário ter vários diretórios em "Library Path", e também é desnecessário ter vários arquivos DCU ou PAS apontados por este parâmetro. Para que a organização e a facilidade de trabalho sejam máximas, o "Library Path" precisa ter apenas um diretório especificado, e este diretório deve ter nada mais do que os seguintes arquivos:

1) o arquivo SYSTEM.DCU.

2) o arquivo SYSINIT.DCU.

3) os arquivos DCP.

Os arquivos SYSTEM e SYSINIT são usados pelo Delphi em qualquer projeto, mesmo que eles já estejam em outro *package*. Por isso, eles precisam ser apontados pelo "Library Path". Já os arquivos DCPs são usados por outros *projetos*, e vários projetos podem usar o mesmo arquivo DCP. Portanto, é conveniente colocar todos os arquivos DCP no diretório apontado por "Library Path". Para colocar um DCP em um diretório qualquer, basta especificar este diretório em "DCP output directory", ou usar o parâmetro /LE, do DCC32.

É interessante que não existam outras units além da SYSTEM e da SYSINIT no diretório apontado pelo "Library Path". O problema de existirem outras units é que o Delphi pode escolher o arquivo-fonte errado para exibir no editor.

D) **Se um projeto usa um package, as units do package são desnecessárias.**

Por exemplo, caso um projeto esteja um usando um *package* que contenha as *units* CLASSES e FORMS, os arquivos destas *units* (CLASSES.PAS ou CLASSES.DCU, e FORMS.PAS ou FORMS.DCU) não são necessários para compilar o projeto. Basta que exista o arquivo DCP do *package*. No entanto, os arquivos-fonte (extensão PAS) podem ser necessários para depurar o *package*, mas não para compilar um projeto que use o *package*.

E) **Os packages não impedem que projetos sejam compilados com units "à moda antiga".**

Para compilar projetos sem usar *packages*, mas usando *units* quaisquer, como CLASSES, SYSUTILS, SENUTI, RTUTIL, ou qualquer outra *unit*, é muito fácil, e não representa qualquer problema. Basta ter um diretório separado para as *units* que não pertencem à qualquer *package*.

F) **Packages podem ser depurados facilmente.**

Não é possível e não tem sentido depurar um *package* por si só. Para depurar um *package*, é necessário depurar um projeto que utilize o *package*. Basta abrir o projeto que utiliza o *package* e colocar no parâmetro "Debug source path" (que é um parâmetro local do projeto), o caminho dos arquivos-fonte do *package*. Todos os arquivos-fonte do *package* podem ser depurados facilmente.

6. DESENVOLVIMENTO DO PROTÓTIPO

6.1 FERRAMENTA DE SELEÇÃO E CLASSIFICAÇÃO DE COMPONENTES

A ferramenta é capaz de armazenar, classificar, consultar de forma fácil e rápida componentes e *packages*, como uma biblioteca, e com a funcionalidade de após uma consulta, adicioná-los e removê-los do ambiente, conforme a necessidade de uso do desenvolvedor.

O método para classificação dos componentes e *packages* é o mesmo já utilizado e comprovado por [RAM1999] a “classificação facetada”.

A ferramenta foi integrada ao ambiente, isto é, ela faz parte do menu principal do Delphi, juntamente com as outras ferramentas disponibilizadas pelo ambiente. Como por exemplo: “ImageEditor”, DataBaseDesktop, e etc.

Como a ferramenta é uma *package*, a instalação da mesma ao ambiente será de fácil operação, porque o Delphi possui um instalador de *packages*.

Logo, o desenvolvedor não precisará instalar a ferramenta sempre que necessária, pois a mesma estará agregada ao Delphi, sempre que o mesmo entrar no ambiente. Bastará somente selecioná-lo no menu principal.

Com uma ferramenta com tal funcionalidade, agregada ao ambiente, será de suma importância no processo de reuso de componentes e *packages*, a da motivação do desenvolvedor ao criá-los.

6.2 ESPECIFICAÇÃO DO PROTÓTIPO

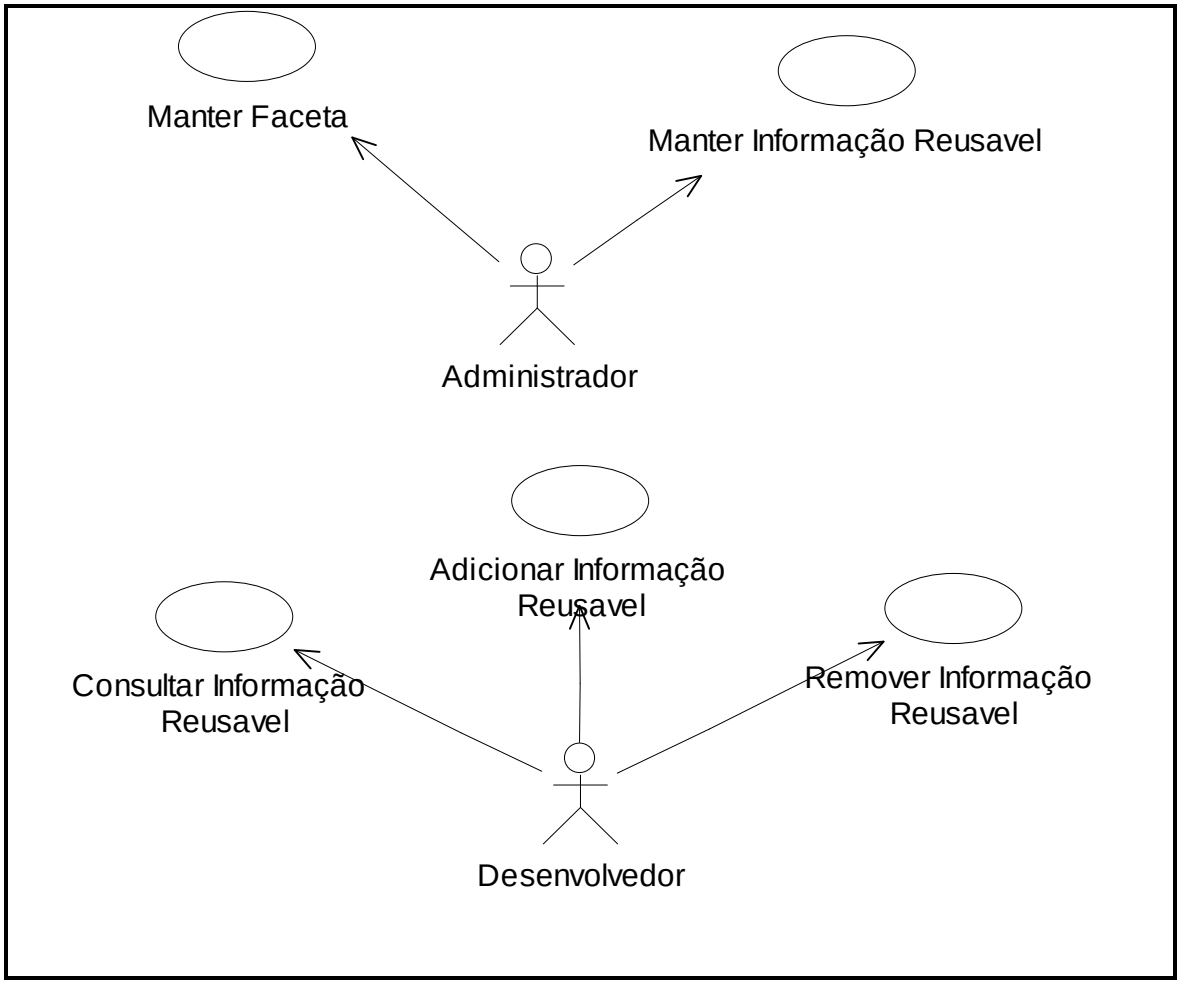
A modelagem do protótipo foi feita utilizando-se da *Unified Modeling Language (UML)* - técnica largamente utilizada para modelagem de sistemas orientados a objetos [FUR1995]. E, na prática, foi utilizada a ferramenta *Rational Rose* para

realizar a modelagem, pois possui a tecnologia necessária para o desenvolvimento dos diagramas abaixo especificados.

6.2.1 USE CASE

O Diagrama Use Case, conforme Figura 5, foi criado com o intuito de dar uma visão global dos processos principais e da interação com os atores.

Figura 5 - Diagrama Use Case



O ator, denominado por Administrador, é responsável para dar manutenção nas facetas, nos seus respectivos valores e também, nas informações reutilizáveis. Já o desenvolvedor será o usuário que fará uma consulta e escolherá, do conjunto de informações reutilizáveis recuperadas, aquela informação que mais se adequa as suas necessidades, para adicionar ou remover do ambiente de desenvolvimento.

A tabela 2, apresenta uma descrição detalhada de cada caso de uso demonstrado na figura 5.

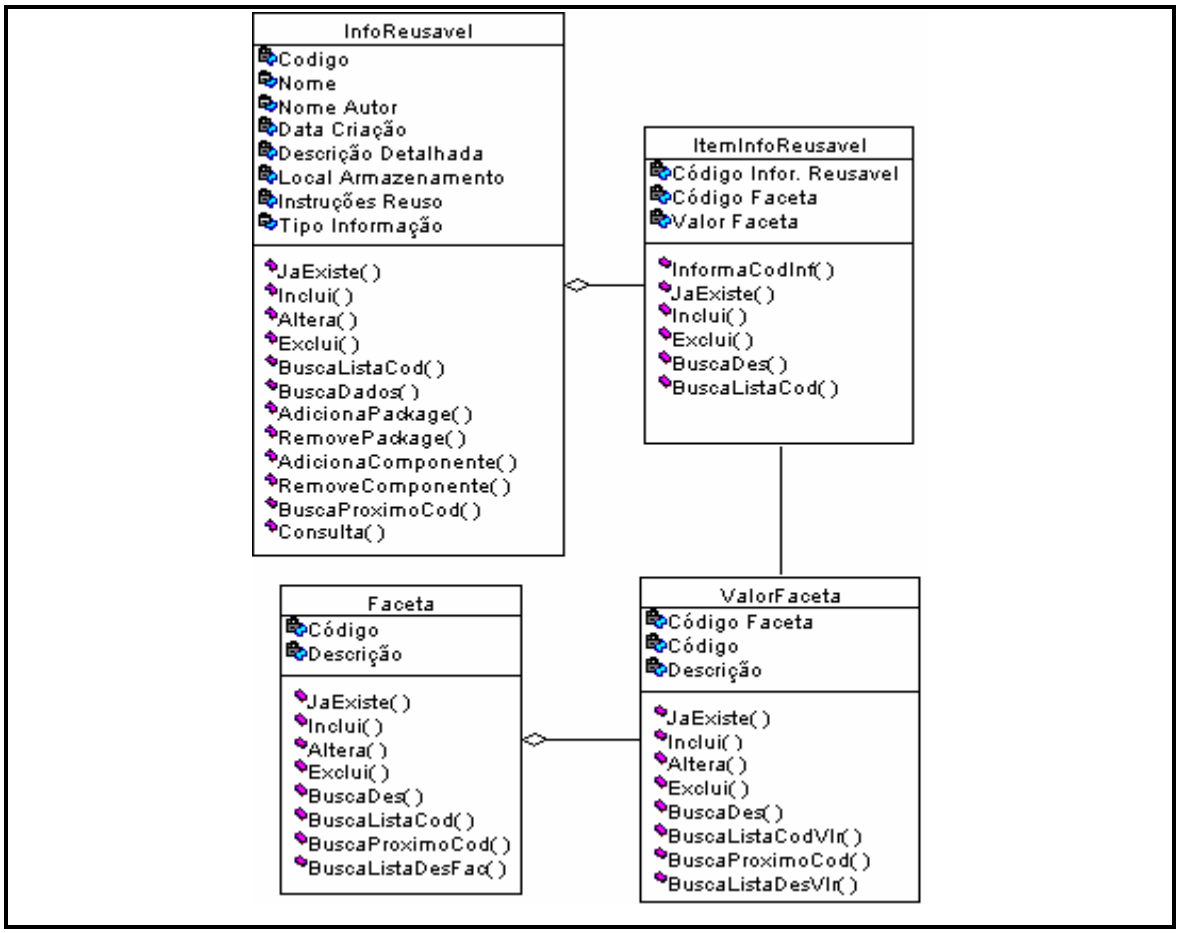
Tabela 2 - Descrição detalhadas de cada caso de uso

Caso de Uso	Descrição
Manter Faceta	Inclusão, Alteração e Exclusão de Faceta e dos seus respectivos valores.
Manter InformaçãoReusável	Inclusão, Alteração e Exclusão de Informação Reusável e dos seus respectivos itens de classificação.
Consultar Informação Reusável	Consulta de Informação Reusável, na qual o desenvolvedor definirá os parâmetros através das facetas e dos seus respectivos valores, com o intuito de adicionar ao seu projeto.
Adicionar Informação Reusável	Adição de componente ou *package* ao projeto, pré-consultados.
Remover Informação Reusável	Remoção de componente ou *package* que não fazem parte do projeto.

6.2.2 DIAGRAMA DE CLASSES NO NÍVEL LÓGICO

Conforme a Figura 6, no diagrama de classes no nível lógico é visualizado os atributos e as operações das classes utilizadas no desenvolvimento do protótipo.

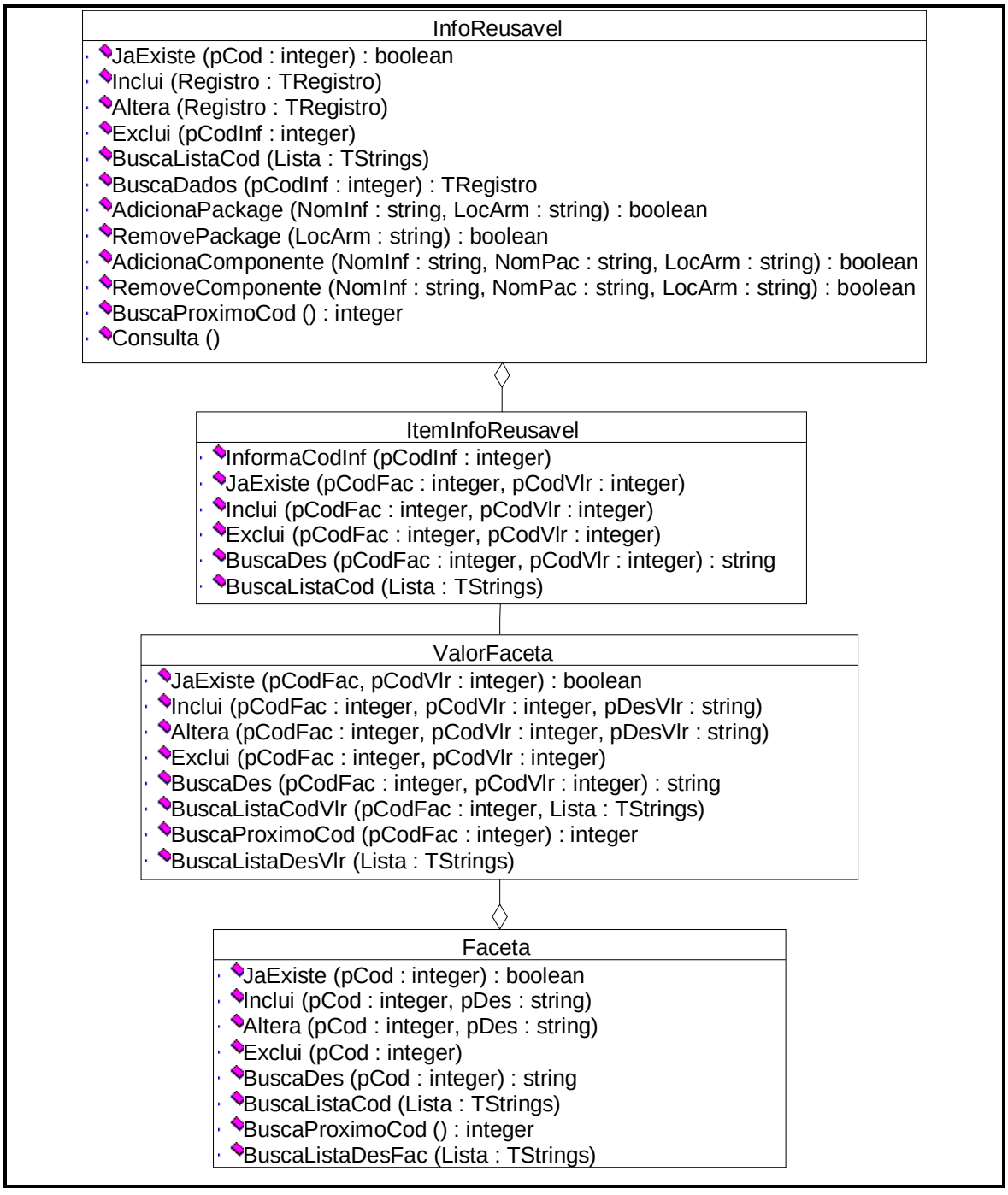
Figura 6 - Diagrama de Classes no Nível Lógico



6.2.3 DIAGRAMA DE CLASSES NO NÍVEL FÍSICO

Conforme a Figura 7, o diagrama de classes é composto por quatro classes, modeladas para servirem como uma interface entre as tabelas utilizadas e a aplicação.

Figura 7 - Diagrama de Classes no Nível Físico



6.2.4 MODELO ENTIDADE RELACIONAMENTO

O Modelo Entidade Relacionamento demonstra as tabelas utilizadas para armazenar as informações reutilizáveis, as facetas e os seus respectivos valores.

Este modelo foi criado na ferramenta ERWIN, pois o Rational Rose, não modela esse tipo de diagrama. As tabelas apresentadas na Figura 8, foram criadas no Paradox.

Figura 8 – Modelo Entidade Relacionamento

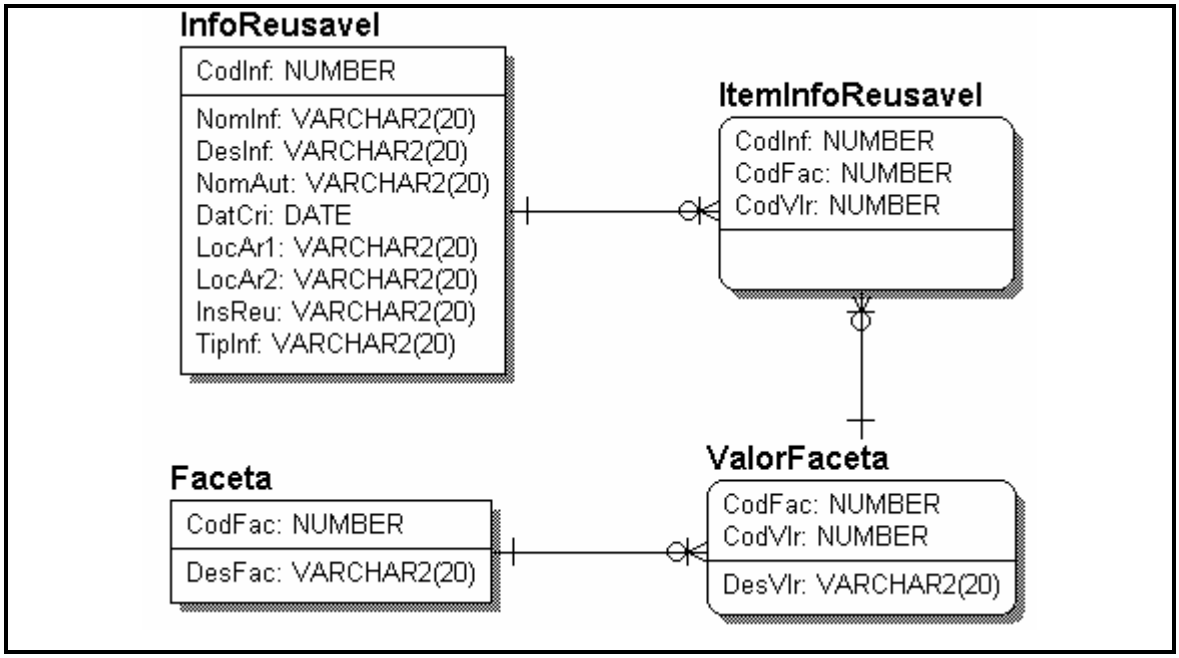


Tabela 3 - Descrição das colunas das tabelas

Nome Campo	Descrição Campo
CodFac	Código Faceta
CodInf	Código Informação Reusável
CodVlr	Código Valor Faceta
DatCri	Data da Criação da Informação Reusável
DesFac	Descrição da faceta
DesInf	Descrição detalhada

DesVlr	Descrição do Valor da Faceta
InsReu	Instruções de Reuso da Infor. Reusável
LocAr1	Local de armazenamento da package/compononente
LocAr2	Local de armazenamento da package
NomAut	Nome do Autor da Informação Reusável
NomInf	Nome da Informação Reusável
TipInf	Tipo da Informação Reusável

6.2.5 DIAGRAMA DE SEQÜÊNCIA

Segue os diagramas de seqüência modelados com o intuito de visualizar as operações principais das classes acima vistas:

Figura 9 - Diagramas de Sequência - Inclusão e Alteração Faceta

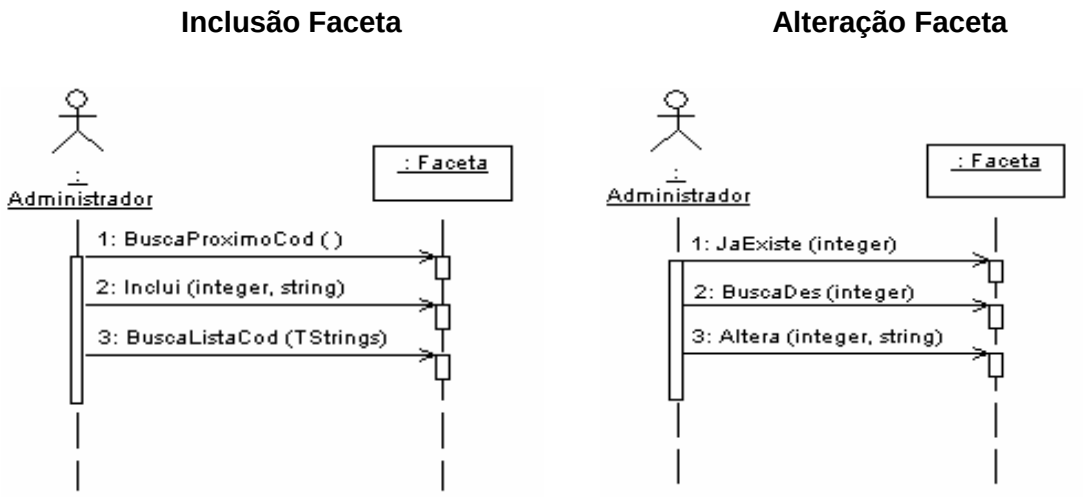


Figura 10 - Diagrama de Sequência - Exclusão Faceta e Inclusão Valor Faceta

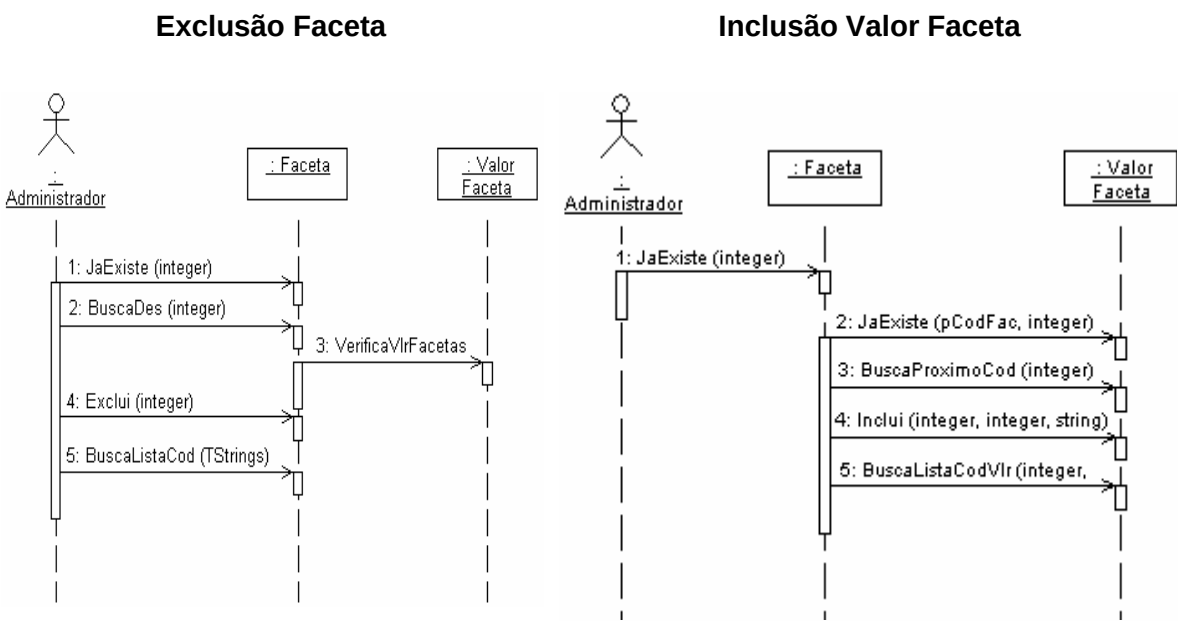


Figura 11 - Diagrama de Sequência - Alteração e Exclusão Valor Faceta

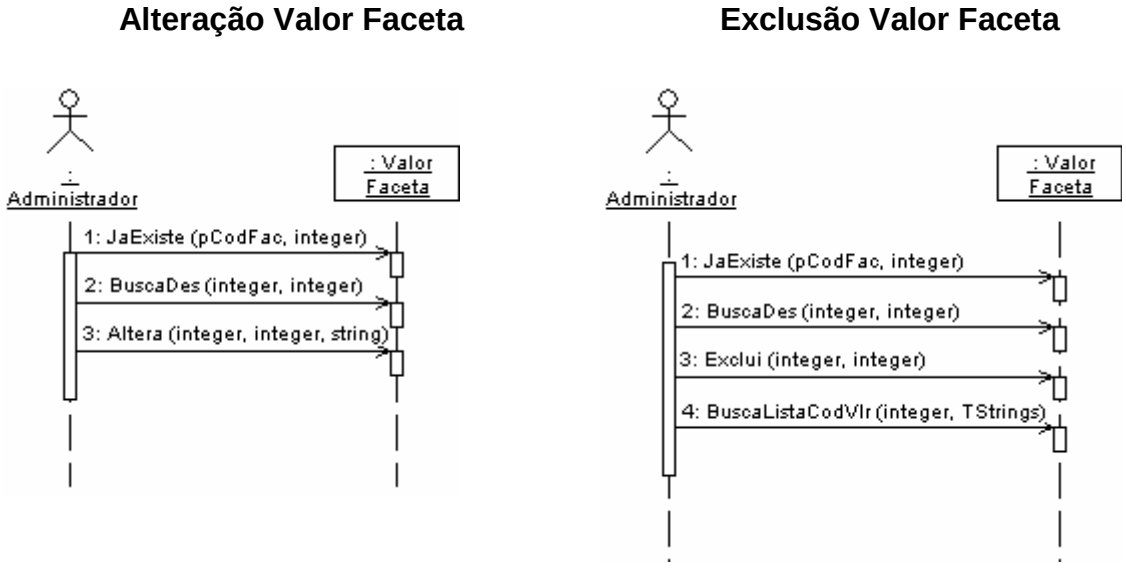


Figura 12 - Diagrama de Sequência - Inclusão e Alteração Informação Reusável

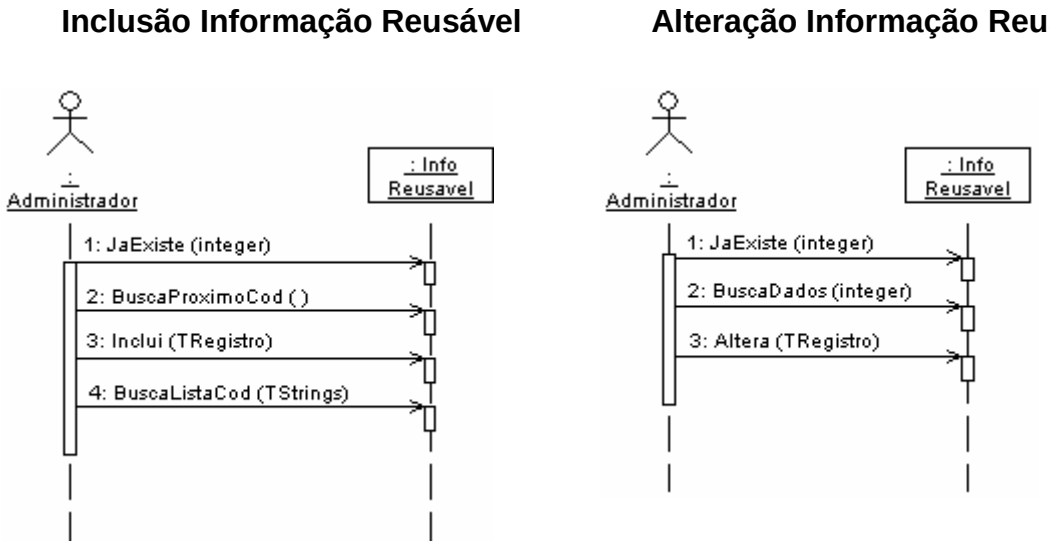


Figura 13 - Diagrama de Sequência - Exclusão Informação Reusável e Inclusão Item Informação Reusável

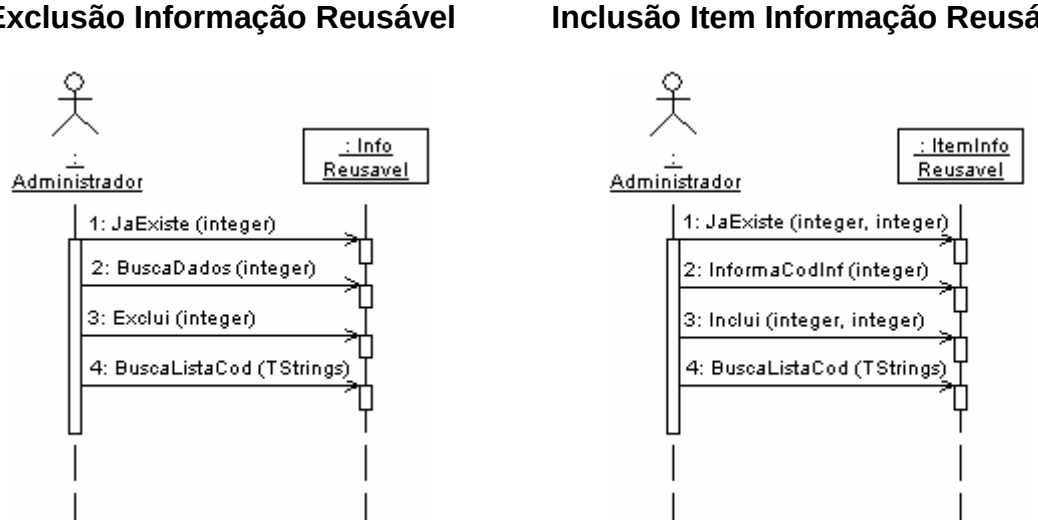
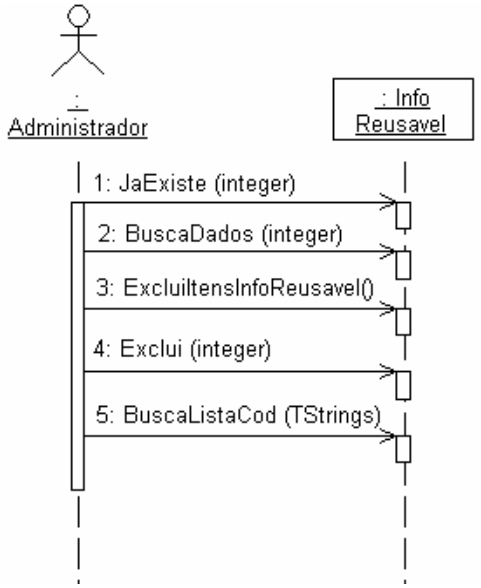


Figura 14 - Diagrama de Sequência - Exclusão Item Informação Reusável e Adição/Remoção de Packages/Componentes

Exclusão Item Inform. Reusável



Adição/Remoção de Packages/Compon.

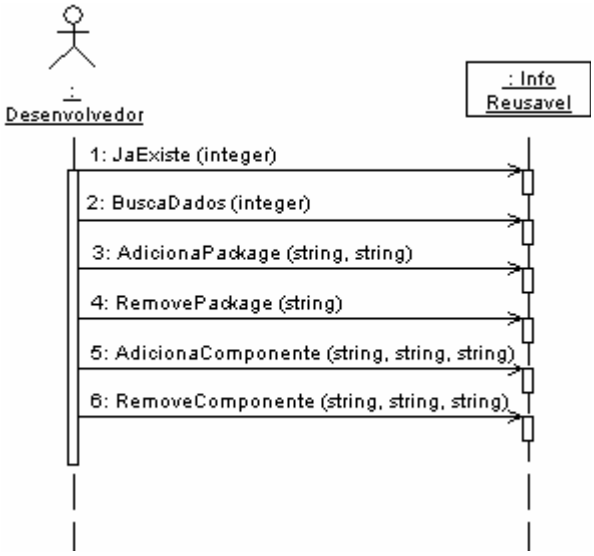
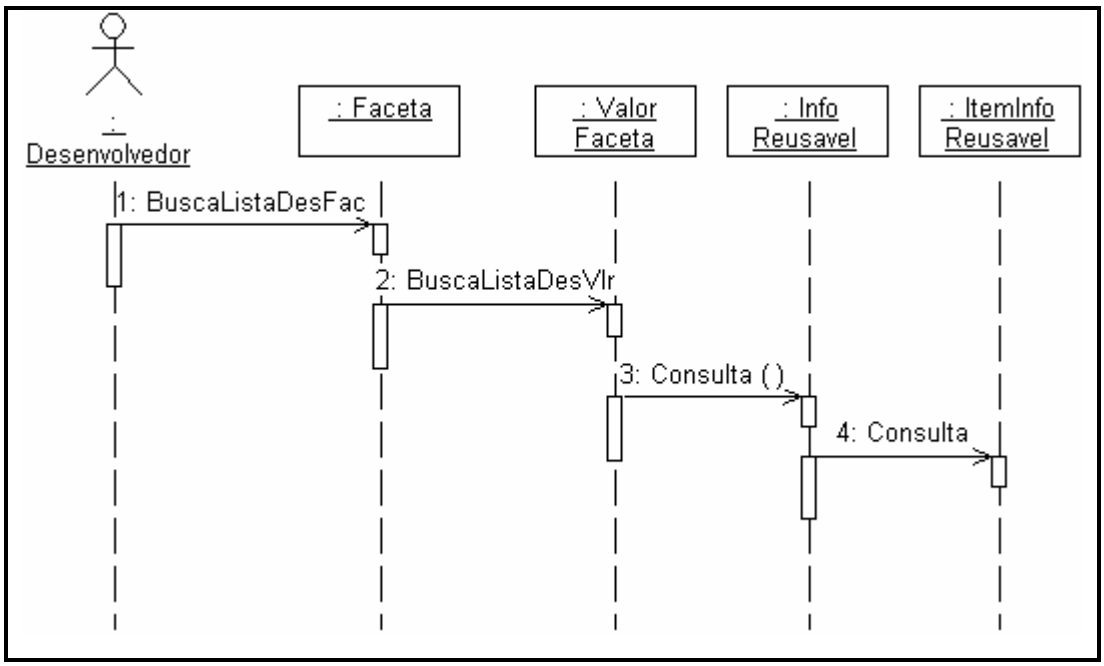


Figura 15 - Diagrama de Sequência - Consulta Informação Reusável



6.3 IMPLEMENTAÇÃO DO PROTÓTIPO

O protótipo foi implementado na linguagem de desenvolvimento visual Delphi, pois o mesmo será incorporado a mesma. Também, como a ferramenta de seleção e classificação de componentes utiliza a própria interface interna do Delphi para agregá-la ao ambiente, seria inapropriado utilizar outra linguagem.

Foram encontrados muitos problemas no desenvolvimento deste protótipo, devido ao fato que o Delphi possui uma interface limitada para os desenvolvedores que necessitam manipular dados do próprio ambiente, como por exemplo, o já citado anteriormente, adição e remoção de packages e tecnologia proprietária.

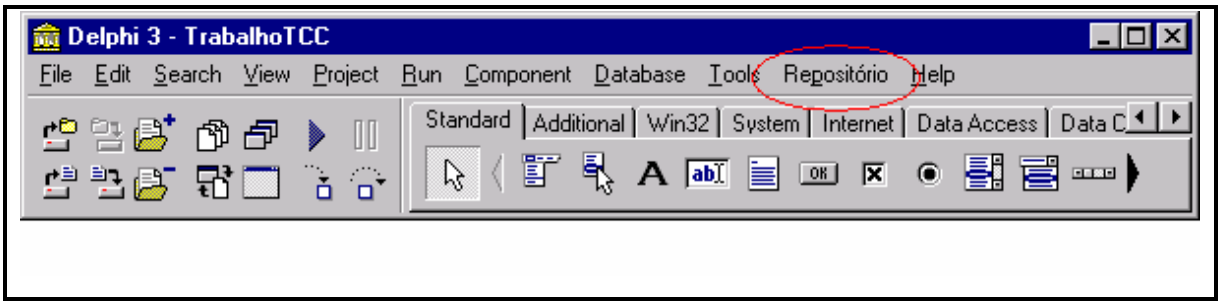
Foram feitos muitos trabalhos de pesquisa para que a ferramenta tivesse o mesmo nível de operação com *packages* e componentes que o Delphi disponibiliza. Mas conforme o descoberto, o ambiente possui uma interface para o desenvolvimento somente para consulta das *packages* e componentes instalados com a sua própria ferramenta.

Então, foram procuradas outras formas para garantir que a ferramenta fosse capaz de além de armazenar e consultar componentes e *packages*, de adicioná-los e

removê-los do ambiente. Uma solução encontrada foi trabalhar com o *registry* do Windows, pois é lá que o Delphi registra todas as *packages* que são instaladas, e as verifica quando o desenvolvedor entra no ambiente. Logo, a ferramenta disponibilizará a funcionalidade de adicionar e remover *packages*, mas com a limitação de ter que reinicializar o ambiente para que as alterações sejam aplicadas com sucesso ao ambiente.

A ferramenta foi integrada ao ambiente de desenvolvimento Delphi, conforme visualizado na figura 16.

Figura 16 – A ferramenta integrada ao ambiente de desenvolvimento



O quadro 1 é um exemplo do código fonte utilizado para a incorporação da ferramenta ao ambiente. Foi utilizada uma *unit* auxiliar, somente para adicionar a ferramenta no menu principal do Delphi. Esta unit não está no contexto das classes anteriormente visualizadas.

Quadro 1 - Código Fonte do Protótipo (Incorporação ao Delphi)

```
procedure ChamaRepositorio(Self: TObject; Sender: TIMenuItemIntf);
begin
     if FormPrincipal = nil
     then  FormPrincipal := TFrmRepositorio.Create(nil);
     FormPrincipal.Show;
end;

procedure SetupMenu; // adiciona item repositório ao menu do Delphi
var
     Method: TMethod;
begin
     Method.Data := nil;
     Method.Code := @ChamaRepositorio;
     MainMenu := ToolServices.GetMainMenu.GetMenuItems.InsertItem(
          9,'Re&pository','MenuRepositorio','', 0, 0, 0,
          [mfVisible, mfEnabled],nil);
     MainMenu.InsertItem(-1,'Executar...','MenuExecutar','', 0, 0, 0,
          [mfVisible,mfEnabled],TIMenuClickEvent(Method));
end;

procedure CleanupMenu; // retira item repositório ao menu do Delphi
     function DestroyMenuItem(var MenuItem: TIMenuItemIntf): Boolean;
     begin
     Result := False;
     if   (MenuItem <> nil) then
          begin
          Result := MenuItem.DestroyMenuItem;
          if   (Result) then
               MenuItem := nil;
          end;
     end;
begin
     FormPrincipal.Destroy;
     DestroyMenuItem(MainMenu);
end;
```

O quadro 2 é um exemplo do código fonte utilizado para adicionar e remover *packages*.

Quadro 2 - Código Fonte do Protótipo (Adição e Remoção de Packages)

```
function TInfoReusavel.AdicionaPackage(NomInf, LocArm: string):
boolean;
var
     Registry: TRegistry;
     Adicionou: boolean;
begin
if not JaExisteChaveRegistry(LocArm)
then begin
  Registry := TRegistry.Create;
  Adicionou := true;
  try
       Registry.RootKey := HKEY_CURRENT_USER;
       If Registry.OpenKey('\Software\Borland\Delphi\3.0\
            KnownPackages', True)
       then Registry.WriteString(LocArm, NomInf)
       else Adicionou := false;
       Registry.CloseKey;
  finally
       Registry.Destroy;
  end;
end;
     Result := Adicionou;
end;


function TInfoReusavel.RemovePackage(LocArm: string): boolean;
var
     Registry: TRegistry;
     Removeu: boolean;
begin
Registry := TRegistry.Create;
if JaExisteChaveRegistry(LocArm)
then begin
  try
       Registry.RootKey := HKEY_CURRENT_USER;
     If Registry.OpenKey('\Software\Borland\Delphi\3.0\Known
Packages', True)
       then Removeu := Registry.DeleteValue(LocArm)
       else Removeu := false;
       Registry.CloseKey;
  finally
       Registry.Destroy;
     end;
end;
     Result := Removeu;
end;
```

Os códigos fontes a seguir, demonstram as interfaces das classes implementadas.

Quadro 3 - Interface da classe TFaceta

```
TFaceta = class(TObject)
    private
         TabelaFaceta: TTable;
    public
         constructor Create;
         destructor Destroy;
         function JaExiste(pCod: integer): boolean;
         procedure Inclui(pCod: integer; pDes: string);
         procedure Altera(pCod: integer; pDes: string);
         function Exclui(pCod: integer): boolean;
         function BuscaDes(pCod: integer): string;
         procedure BuscaListaCod(Lista: TStrings);
         procedure BuscaListaDes(Lista: TStrings);
         function BuscaProximoCod: integer;
    end;
```

Quadro 4 - Interface da classe TValorFaceta

```
TValorFaceta = class(TObject)
    private
         TabelaValorFaceta: TTable;
    public
         constructor Create;
         destructor Destroy;
         function JaExiste(pCodFac, pCodVlr: integer): boolean;
         procedure Inclui(pCodFac, pCodVlr: integer; pDesVlr: string);
         procedure Altera(pCodFac, pCodVlr: integer; pDesVlr: string);
         procedure Exclui(pCodFac, pCodVlr: integer);
         function BuscaDes(pCodFac, pCodVlr: integer): string;
         procedure BuscaListaCodVlr(pCodFac: integer; Lista:
TStrings);
         procedure BuscaListaDesVlr(pCodFac: integer; Lista:
TStrings);
         procedure BuscaListaCodFac(Lista: TStrings);
         function BuscaProximoCod(pCodFac: integer): integer;
    end;
```

Quadro 5 - Interface da classe TInfoReusavel

```
TInfoReusavel = class(TObject)
      private
          TabelaInfoReusavel: TTable;
          function JaExisteChaveRegistry(LocArm: string): boolean;
      public
          constructor Create;
          destructor Destroy;
          function JaExiste(pCod: integer): boolean;
          procedure Inclui(Registro: TRegistro);
          procedure Altera(Registro : TRegistro);
          procedure Exclui(pCodInf: integer);
          procedure BuscaListaCod(Lista: TStrings);
          function BuscaTipInf(pCodInf: integer): string;
          function BuscaDados(pCodInf: integer): TRegistro;
          function BuscaProximoCod: integer;
          function AdicionaPackage(NomInf, LocArm: string): boolean;
          function RemovePackage(LocArm: string): boolean;
          function AdicionaComponente(LocArmPac, LocArmCom: string):
boolean;
          function RemoveComponente(LocArmPac, LocArmCom: string):
boolean;
      end;
```

Quadro 6 - Interface da classe TitemInfoReusavel

```
TItemInfoReusavel = class
     private
          CodInf: integer;
          TabelaItemInfoReusavel: TTable;
     public
         constructor Create;
         destructor Destroy;
         procedure InformaCodInf(pCodInf: integer);
         function JaExiste(pCodFac, pCodVlr: integer): boolean;
         procedure Inclui(pCodFac, pCodVlr: integer);
         procedure Exclui(pCodFac, pCodVlr: integer);
         function BuscaDes(pCodFac, pCodVlr: integer): string;
         procedure BuscaItems(Items: TStrings);
      end;
```

6.4 APRESENTAÇÃO DA FERRAMENTA

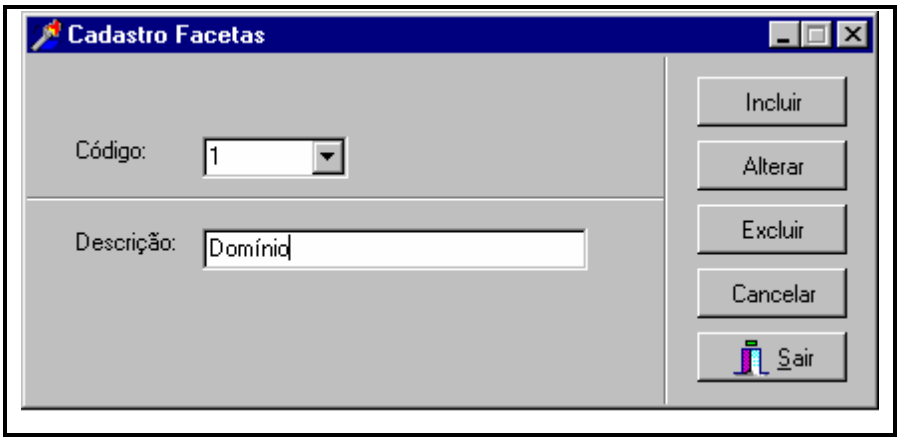
Este tópico apresenta uma descrição da ferramenta de seleção e classificação de *packages* e componentes reutilizáveis em Delphi.

A figura 17 mostra como é simples o acesso da ferramenta dentro do ambiente de desenvolvimento Delphi. Como pôde ser visto na figura 16 a ferramenta está integrada ao ambiente de desenvolvimento para facilitar a interação no uso pelo desenvolvedor.

A ferramenta, para dar maior liberdade ao administrador do mesmo, e para ser um sistema aberto, possui um cadastro das facetas (características) e dos seus respectivos valores para classificação dos componentes ou *packages*. Desta maneira, as facetas e os seus valores são cadastrados conforme a necessidade do administrador e também, para poderem ser reutilizados por outros componentes ou *packages* que possuem algo em comum, facilitando desta maneira a manutenção dos dados. Isto é, eles também fazem parte da biblioteca de dados reutilizáveis da ferramenta.

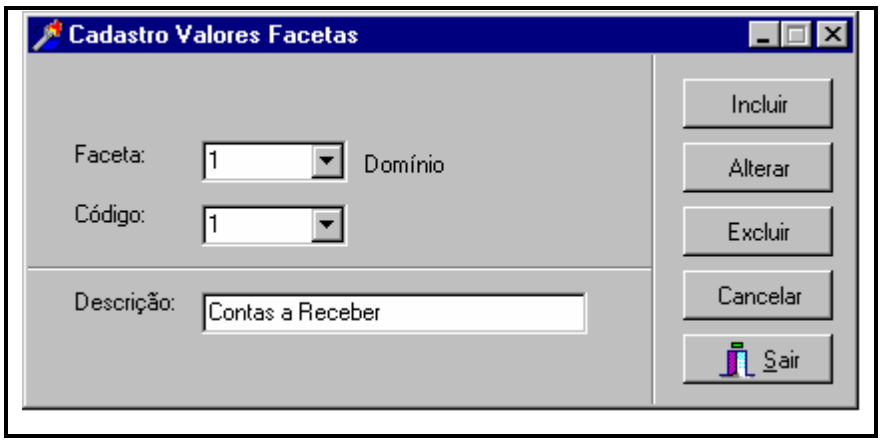
A figura 17 é um demonstrativo da interface de manutenção das facetas:

Figura 17 - Cadastro de Facetas



A figura 18 é um demonstrativo da interface de manutenção dos valores das facetas:

Figura 18 - Cadastro dos Valores das Facetas



A parte principal da ferramenta, está no cadastro dos componentes e *packages*. Conforme a figura 19, o administrador cadastrará componentes ou *packages*, informando os dados referentes a criação dos mesmos, informará instruções necessárias sobre o seu reuso, onde está armazenado fisicamente e qual o tipo de informação reusável com abrangência de, "componente" a "*package*". E, o desenvolvedor adicionará a informação reusável, facetas e valores correspondentes conforme figura 20, que melhor definam a informação. Também, a interface disponibiliza ao desenvolvedor a adição e remoção da informação reutilizável ao ambiente de desenvolvimento.

Figura 19 - Cadastro Informação Reusável

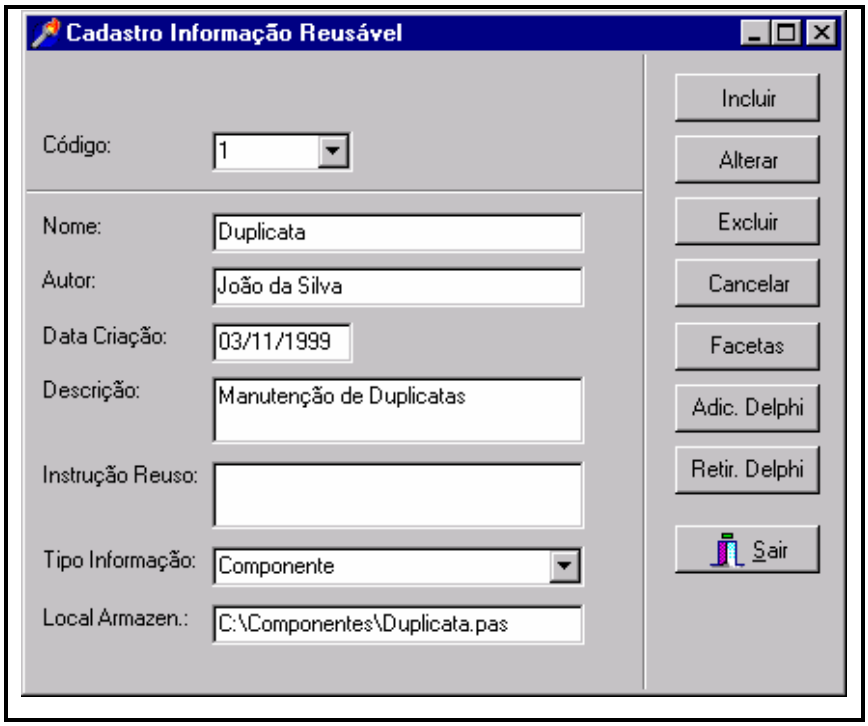
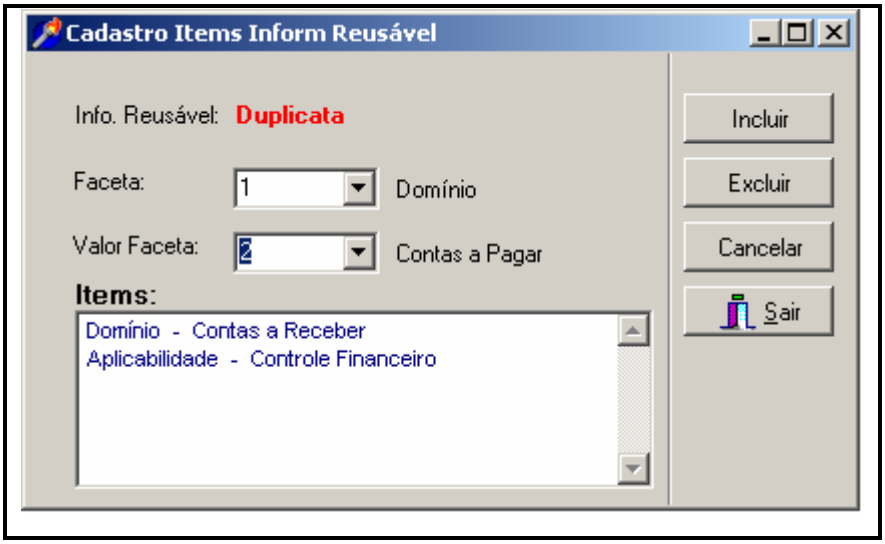
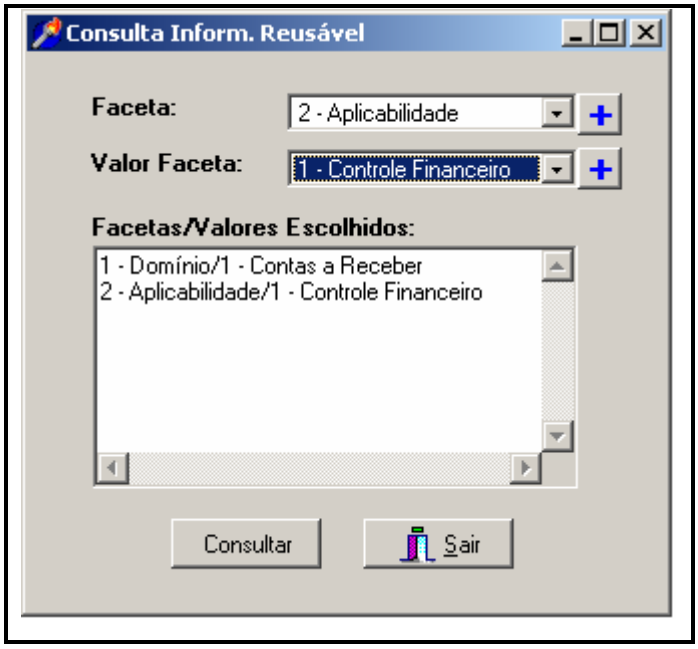


Figura 20 - Facetas e Valores correspondentes a Informação Reusável



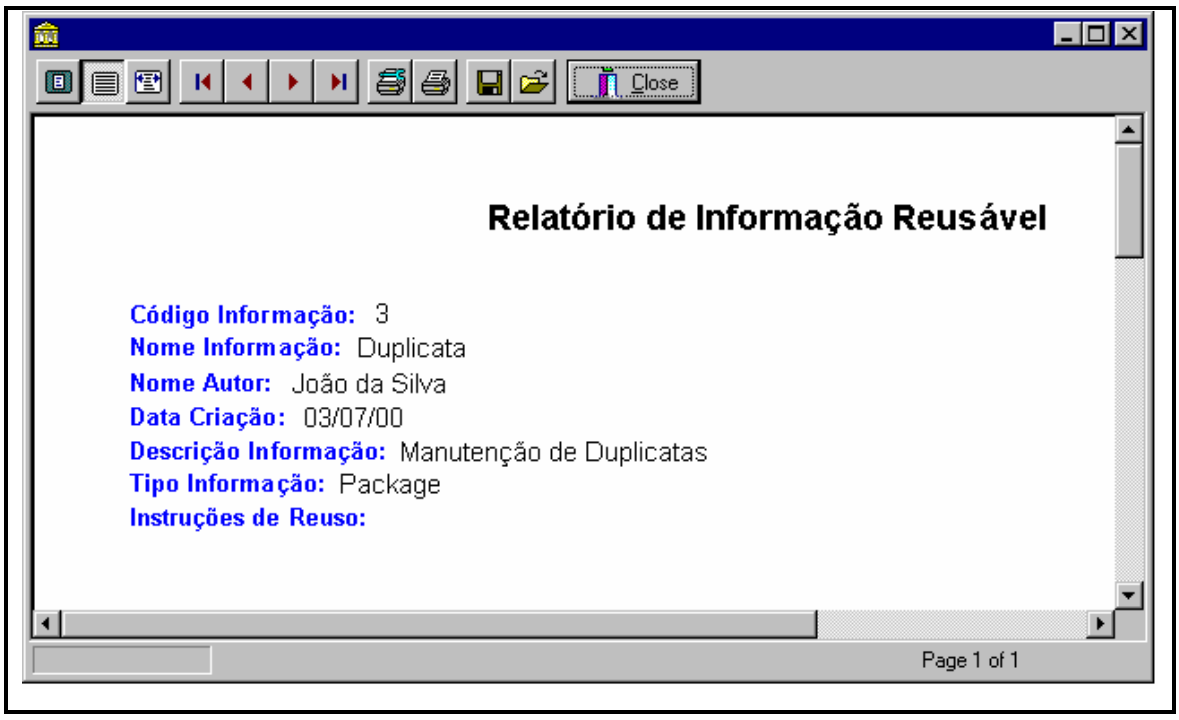
A ferramenta possui uma consulta dinâmica às informações reusáveis incorporadas a ela. Desta maneira o desenvolvedor fará uma consulta informando dados chaves para encontrar o componente ou a *package* específica ou próximos do seu objetivo. Serão procuradas informações reutilizáveis que atendam a todas as facetas e os seus respectivos valores selecionados. A figura 21 demonstra a interface criada para ajudar neste processo.

Figura 21 - Consulta Informação Reusável



Após o desenvolvedor definir os dados de consulta, o repositório fornece um relatório, conforme figura 22, com todos os componentes ou *packages* encontradas que se identificam com os dados por ele informados. Então, o desenvolvedor poderá acessar a interface de cadastro e manutenção de dados das informações reusáveis e adicionar ao ambiente Delphi para ser utilizado no seu projeto.

Figura 22 - Relatório de Informação Reusável



7. CONCLUSÃO

7.1 CONSIDERAÇÕES FINAIS

A ferramenta de seleção e classificação de componentes implementada teve como principal objetivo ser uma alternativa para solucionar um dos problemas de produtividade e manutenção de sistemas, visando o desenvolvimento e a adaptação de projetos desenvolvidos no ambiente de programação visual Delphi, constituindo-os de componentes de código reutilizáveis. Assim, a ferramenta, na sua total aplicação pelos desenvolvedores de softwares, poderá contribuir para o desenvolvimento dos seus projetos.

Bem aplicado nas empresas suprirá a carência nas organizações, com baixo nível de reusabilidade, de utilizar uma ferramenta adequada e efetiva na prática do desenvolvimento do seus sistemas.

A ferramenta foi incorporada de forma integrada ao ambiente de desenvolvimento Delphi, muito utilizado atualmente pelos desenvolvedores de softwares, no mesmo nível de adequação como as demais ferramentas de apoio ao desenvolvimento de projetos, disponibilizadas pelo ambiente. Possui limitações, já discutidas no tópico 6.3, em função dos problemas encontrados na implementação do repositório e interfaces de comunicação com a linguagem Delphi.

O uso efetivo da ferramenta não dependerá exclusivamente do seu nível de funcionalidade dos processos ou da sua interface, mas em primeiro lugar na mudança do modelo mental para que o desenvolvedor adquira o hábito de projetar componentes imaginando aplicações futuras e principalmente na diminuição da resistência a mudanças dos desenvolvedores.

7.2 SUGESTÕES

As principais sugestões para futuros trabalhos nesta área são:

a) desenvolvimento de ferramentas especializadas para realizar testes com alto grau de depuração de componentes para serem reutilizados com maior segurança pelo desenvolvedor;

b) desenvolvimento de uma ferramenta de classificação de componentes com diversas formas de classificação, como por exemplo técnicas da área de inteligência artificial, para facilitar a consulta.

As sugestões para o aprimoramento da ferramenta são:

a) segurança na manutenção do repositório, como por exemplo, um controle de usuários e das suas respectivas senhas;

b) compartilhamento do repositório, pois o mesmo foi implementado para ser utilizado localmente;

c) resolução dos problemas de limitações do repositório, utilizando para desenvolvimento do mesmo uma nova versão do ambiente Delphi;

d) busca por aproximação, usando técnicas de inteligência artificial, como lógica difusa.

REFERÊNCIAS BIBLIOGRÁFICAS

[ARN1994] ARNOLD, Robert S. **Software reengineering**. Los Alamitos : IEEE Computer Society, 1994.

[BAR1996] BARROS, Márcio de Oliveira, Cláudia Maria Lima Werner, Luiz Pereira Caloba. **Recuperação de componentes em bibliotecas de software: uma abordagem conexionista. Anais do X Simpósio Brasileiro de Engenharia de Software**. São Carlos, 1996.

[CAN1998] CANTU, Marco. **Dominando o delphi 3: a bíblia**. São Paulo : Makron Books, 1998.

[CAN2000] CANTU, Marco. **Dominando o delphi 5: a bíblia**. São Paulo: Makron Books, 2000.

[CHE1994] CHENG, Jingewn. **A Reusability-Based Software Development Enviornment**. Austrália : University Monash, 1994.

[COR1994] CORMELATO, Cesar A, Geraldo B. Xexeo, Ana Regina C. da Rocha. **Avaliação da reutilizabilidade de componentes de softwares**. Anais do VIII Simpósio Brasileiro de Engenharia de Software. Curitiba, 1994

[DEL1997] **DELPHI32. 3.0.** Ambiente para desenvolvimento de programas. Borland International Inc. Borland International Inc. Scotts Valley, 1997. 1 CD. Total do CD kilobytes.

[FRA1991] FRANKES, W. B. and P.B.Gandel, **Representing reusable software. Information and Software Technology** : Butterworth-Heinemann, 1991.

[FRE1987] FREEMANN, Peter. **Tutorial: Software Reusability**. Washington: IEEE Computer Soc., 1987.

[FUR1995] FURLAN, José Davi. **Reengenharia da informação: do mito a realidade**. São Paulo : Makron Books, 1995.

[GIR1990] GIRARDI, R. M. and Price. **Um sistema para recuperação de classes reutilizáveis no desenvolvimento de objetos-orientados**. Instituto de Informática da Universidade Fed. do Rio Grande do Sul: Porto Alegre, 1990.

[HAM1999] HAMANN, Kátia Simone. **Comparativo de Esquemas de Classificação de Componentes Reusáveis**. Blumenau, 1999. Trabalho de Conclusão de Curso – Ciencias Exatas e Naturais, Universidade Regional de Blumenau.

[KUT1997] KUTOVA, Marcos André Silveira; MACEDO, Alessandra Alaniz. **Reuso de Software**. São Carlos, 1997. Trabalho de Conclusão de Curso – Ciências Exatas, Universidade de São Paulo.

[LEI1992] LEITE, J. C. S. do Prado. **Reengenharia de software um novo enfoque para um velho problema**. IPESI Negócios & Informática: Rio de Janeiro, 1992.

[MAR1196] MARTIN, James; Odell James J. **Análise projeto orientados a objeto**. São Paulo : Makron Books, 1996.

[MCC1993] MCCLURE, Carma. **The three rs of the software automation: re-engineering, reusability, repository**. New Jersey : Prentece Hall, 1993.

[PUJ1995] PUJATTI, L. Ferreira M. A G. V. **Ambiente de reutilização de software: classificação de módulos em tecnologia de composição**. Escola Politécnica da USP: São Paulo, 1995.

[RAD1995] RADA, Roy. **Software Reuse : Principles, Methodologies and Pratices**. England : Oxford, 1995.

[RAM1998] RAMOS, Débora Cristina Leira. **Ferramenta para reutilização de componentes reutilizáveis em Access.** Trabalho de Conclusão de Curso da FURB: Blumenau, 1998.

[REE1996] REENSKAUG, Trygve. **Working with objects: the ooram software engineering method**. Greenwich : Manning, 1996.

[ROC1883] ROCHA, A R. C. **Um modelo para avaliação da qualidade de especificações**. Tese de Doutorado PUC: Rio de Janeiro, 1983.

[ROC1996] ROCHA, Ana Regina C. da, Cláudia M. L. Werner, Guilherme H. Travassos, et al. Hajdu. **Memphis – um ambiente de desenvolvimento de software baseado em reutilização**. Caderno de Ferramentas do X Simpósio Brasileiro de Engenharia de Software. São Carlos, 1996.

[WER1997] WERNER, Cláudia Maria Lima, Mônica Lyra Barreto de Sá, Ilan Goldman. **Introdução da reutilização em uma empresa brasileira de produção de software**. Anais do XI Simpósio Brasileiro de Engenharia de Software. Fortaleza, 1997.