Pedro Jorge E. Hübscher

Um Processo de Desenvolvimento de Software Utilizando TALISMAN e MFC

Dissertação de Mestrado

Departamento de Informática

Rio de Janeiro, 11 de Outubro de 1995.

Pedro Jorge E. Hübscher

Um Processo de Desenvolvimento de Software Utilizando TALISMAN e MFC

Dissertação apresentada ao Departamento de Informática da Pontifícia Universidade Católica do Rio de Janeiro como parte dos requisitos para a obtenção do título de Mestre em Informática: Engenharia de Software.

Orientador: Arndt von Staa.

Departamento de Informática

Pontifícia Universidade Católica do Rio de Janeiro

Rio de Janeiro, 11 de Outubro de 1995.

A Renata e a Júlia

Meus agradecimentos

- ao professor Arndt von Staa, orientador da dissertação, pelo apoio e confiança depositada.
- a minha esposa Renata pelo apoio.
- a minha filha Júlia pelo estímulo.
- aos meus pais.
- a Carlos Frederico pela ajuda na obtenção de diversos artigos.
- a Lorenzo Ridolfi pela ajuda na pesquisa bibliográfica.

Resumo

A adoção de sistemas operacionais que utilizam uma interface gráfica de usuário, aumentou enormemente a dificuldade no desenvolvimento de software. Novas ferramentas têm surgido para tentar contrabalançar este aumento, sem no entanto alcançar o efeito desejado. A grande deficiência destas ferramentas é não permitir a especificação dos sistemas, restringindo-se a automatização de certas tarefas.

Este texto apresenta um *processo de desenvolvimento de software* que integra TALISMAN – um ambiente para especificação e composição de código –, a biblioteca MFC (*Microsoft Foundation Classes*) e outras ferramentas auxiliares. O processo de desenvolvimento focaliza as etapas de especificação de interfaces, codificação e composição de módulos. São utilizadas *linguagens de representação* que permitem combinar especificações com um alto grau de abstração e fragmentos de código. Deste modo, é possível a *geração automática de código fonte* diretamente a partir das especificações. São definidas *normas de especificações* para cada classe de objetos utilizada no processo. *Validadores* automáticos, escritos na linguagem interna de TALISMAN, permitem criticar as especificações contribuindo para a melhoria da qualidade dos softwares produzidos.

Abstract

The adoption of operating systems which use a graphical user interface, has greatly increased the difficulty in developing software. New tools have been developed in the effort to counterbalance this increase, however not being successful. The great deficiency of this tools is that they do not allow the specification of the systems being developed, restricting themselves to the automation of specific tasks.

This text presents a *software development process* that integrates TALISMAN – an environment for specification and source code composition – the MFC (Microsoft Foundation Classes) library and other auxiliary tools. The development process focus on the interface specification, coding and module composition steps. Representation languages that permit the combination of specification with a high degree of abstraction and source code fragments are employed. This allows the *automatic generation of source code* directly from the specifications. *Specification standards* are defined for each object class used throughout the process. *Automatic "validators,"* special programs written in TALISMAN's internal language, allow the critics of specification contributing to the increase in quality of the final software.

Sumário

Lista de Figuras e Tabelas V
1. Introdução 1
1.1 Motivação 1
1.2 Histórico 1
1.2.1 Restrições 1
1.2.2 Requisitos 1
1.3 Objetivo 2
1.4 Principais características 2
1.5 Forma de apresentação 2
2. O Processo de Desenvolvimento Genérico 4
2.1 Etapas de um processo de desenvolvimento genérico 5
2.1.1 Concepção 6
2.1.2 Análise de domínio 6
2.1.3 Análise do processo 6
2.1.4 Análise de requisitos 6
2.1.5 Especificação da essência 6
2.1.6 Arquitetura 7
2.1.7 Modularização dos programas 7
2.1.8 Projeto estruturado e codificação 7
2.1.9 Teste de unidade 7
2.1.10 Composição de módulos 8
2.1.11 Teste incremental 8
2.1.12 Teste de sistema 8
2.1.13 Instalação 8
2.1.14 Auditoria funcional 8
2.1.15 Conversão 8
2.1.16 Auditoria física 8
3. O Processo de Desenvolvimento Específico 9
3.1 Definição da terminologia utilizada 9
3.2 Etapas do processo de desenvolvimento específico 10
3.2.1 Modularização dos programas 10
3.2.2 Projeto estruturado 11
3.2.3 Codificação 12
3.2.4 Teste de unidade 12
3.2.5 Composição de módulos 13
3.2.6 Teste de sistema 13
3.3 As Estruturas Utilizadas 14
3.4 A combinação de Código e Estrutura 16
3.5 Os Procedimentos Básicos 16
3.5.1 A criação de módulos 17
3.5.2 A criação de classes 19
3.5.3 A criação de funções ou métodos 19

3.5.4 A criação de tipos de dados 21
3.5.5 A criação de atributos 22
3.5.6 A criação de diagramas de estados e eventos 23
3.5.7 A Geração de Código Fonte 26
4. Normas de Especificação 27
4.1 Conteúdo de uma especificação 27
4.2 Descrição da especificação das classe de objetos utilizadas 30
4.2.1 Módulos 30
4.2.2 Classes 30
4.2.3 Funções ou Métodos 31
4.2.4 Tipos 33
4.2.5 Atributos 33
4.2.6 Diagramas de estados e eventos 34
5. Ferramentas Utilizadas 35
5.1 TALISMAN 35
5.2 Visual C++ e MFC 35
5.3 Ferramentas auxiliares 37
5.3.1 Gerador de *scripts* para o programa *make* 37
5.3.2 Geradores de tabelas de definição 37
5.4 A integração das ferramentas 38
5.5 Desenvolvimento em equipe 38
6. Avaliação, Conclusões e Perspectivas 39
6.1 Avaliação 39
6.2 Conclusões 40
6.3 Perspectivas 40
Apêndice: Esquema de geração de código para diagramas de estados e eventos 41
Bibliografia 54

Lista de Figuras e Tabelas

FIGURA 2-1 O PROCESSO DE DESENVOLVIMENTO. 4
FIGURA 3-1 ESTRUTURA DE DECOMPOSIÇÃO 11
FIGURA 3-2 MÁQUINA DE ESTADOS UTILIZANDO A REPRESENTAÇÃO *FLUXO DE DADOS* 12
FIGURA 3-3 DIAGRAMA DE ESTADOS E EVENTOS 13
FIGURA 3-4 FRAGMENTO DE ESPECIFICAÇÃO HIPER-TEXTO 15
FIGURA 3-5 ARQUITETURA DE UM MÓDULO 18
FIGURA 3-6 CHAMADA DE FUNÇÕES UTILIZANDO PACOTES. 20
FIGURA 3-7 RELAÇÃO ENTRE TIPOS E BLOCOS DE DADOS. 22
FIGURA 3-8 TIPOS DE EVENTOS 24
FIGURA 3-9 CICLO TÍPICO PARA MANUTENÇÃO OU CRIAÇÃO DE PROGRAMAS 26
FIGURA 4-1 EXEMPLO DE OMISSÃO DE DESCRIÇÃO. 29
FIGURA 5-1 FRAGMENTO DA HIERARQUIA DE CLASSES DA BIBLIOTECA MFC. 36
FIGURA 6-1 CICLO CONVENCIONAL DE TESTE-DIAGNOSE. 39
FIGURA 6-2 CICLO DE TESTE-DIAGNOSE UTILIZANDO O PROCESSO DE DESENVOLVIMENTO. 40

TABELA 3-1 TIPOS DE ESTRUTURA UTILIZADAS 14
TABELA 3-2 CORRESPONDÊNCIA ENTRE TIPOS C++ E "TIPOS TALISMAN" 21

1. Introdução

1.1 Motivação

O processo de desenvolvimento de sistemas de software tem deixado muito a desejar, tanto no que tange à produtividade das equipes, quanto à qualidade dos produtos desenvolvidos. A causa primária para estas deficiências reside no processo quase artesanal de desenvolvimento. Ambientes de desenvolvimento de software tem sido procurados e utilizados para auxiliar na especificação, arquitetura, projeto, construção, controle de qualidade, manutenção e gerência do processo de desenvolvimento destes sistemas. Por outro lado, os atuais sistemas operacionais, fortemente dirigidos por interfaces gráficas (Windows, OS/2, UNIX/Motif, Mac, etc.), têm aumentado enormemente a complexidade dos sistemas desenvolvidos para estes ambientes. Novas ferramentas têm surgido para tentar contrabalançar este aumento, sem no entanto alcançar o efeito desejado. A grande deficiência destas ferramentas é que elas não apoiam a especificação dos sistemas utilizando formas de abstração de alto nível, restringindo-se apenas à automação de algumas tarefas.

O processo de desenvolvimento apresentado, associado ao ambiente de desenvolvimento TALISMAN [STAA92], permite a especificação do sistema utilizando linguagens de representação com um alto grau de abstração, e a geração de código fonte a partir destas especificações.

1.2 Histórico

Este processo é fruto da experiência adquirida ao desenvolver e manter o ambiente TALISMAN e da fase inicial de desenvolvimento do ambiente TOTEM [Staa93]. As restrições descritas a seguir determinaram diversos requisitos para o processo em questão:

1.2.1 Restrições

- recursos escassos – financeiros, humanos e equipamento.
- membros da equipe permanecem pouco tempo no projeto (mão de obra volátil).
- membros da equipe dispersos pela cidade do Rio de Janeiro.

1.2.2 Requisitos

- deve assegurar qualidade por construção.
- robustez – o sistema deve atingir a qualidade esperada sem exigir testes sistemáticos.
- a manutenção deve ser rápida, simples e independente do indivíduo que a realiza.
- deve fornecer normas de especificação bem definidas.
- as interfaces dos diversos componentes devem ser muito bem definidas.
- deve se concentrar no projeto detalhado.

Para o desenvolvimento de TALISMAN, utilizou-se, fundamentalmente, a linguagem C e a representação *Estruturas Modulares*. Ao iniciar o desenvolvimento de TOTEM, adotou-se a linguagem C++, a biblioteca de classes MFC [MFC_25] (*Microsoft Foundation Classes*) e as linguagens de representação

Estruturas Modulares e *Estados e Eventos*. A representação estados e eventos utiliza uma versão simplificada dos diagramas de estados (*Statecharts*) descritos em [Harel88].

1.3 Objetivo

O objetivo principal do processo é minimizar o tempo de propagação de alterações entre as diversas etapas do desenvolvimento. Isto é possível, devido à utilização de ferramentas que:

- permitem especificar o sistema utilizando representações com um alto grau de abstração e gerar código a partir das mesmas.
- auxiliam a manter a consistência das especificações durante as diversas etapas.

A simples utilização das ferramentas não é suficiente. É necessário definir uma metodologia que governe a sua utilização, defina normas de especificação e as diversas etapas que devem ser realizadas.

1.4 Principais características

1. O processo está focalizado no desenvolvimento de sistemas de médio porte[1].
2. Utilização de ferramentas.
    - TALISMAN
    - Visual C++
    - Geradores de *scripts* para *make*
    - Geradores de tabelas de definição
3. Flexibilidade na seqüência de execução das etapas.

    A seqüência de execução das etapas é restrita apenas pela necessidade de existência de um objeto antes de que possa ser referenciado por outros.
4. Ponto de partida – o conjunto de módulos que devem ser implementados é conhecido.

1.5 Forma de apresentação

O processo de desenvolvimento apresentado está baseado na experiência passada. Isto sugere a sua apresentação de modo *descritivo*. Os seguintes argumentos, entretanto, levaram à sua apresentação de modo *prescritivo* [Dow91]:

- facilita a sua aplicação em outros contextos.
- evita sobrecarregar o leitor com detalhes sobre o ambiente aonde foi aplicado.

O restante deste texto está organizado do seguinte modo:

No capítulo 2 é apresentado um processo de desenvolvimento genérico. Isto tem como objetivo estabelecer o contexto para a definição do processo de desenvolvimento específico, apresentado no capítulo 3. O processo é particionado em etapas, definidas em termos dos resultados que devem ser produzidos pelas mesmas.

---

[1] Definir o "tamanho" de um sistema de software pode ser bastante subjetivo. Para fins comparativos, podemos definir um sistema de médio-porte como sendo aquele que é desenvolvido por uma equipe com menos de 6 pessoas e que possua entre 50.000 e 250.000 linhas de código.

O capítulo 3 apresenta o processo de desenvolvimento específico, as linguagens de representação, as estruturas e classes de objetos utilizadas, bem como sua correspondência com os elementos da linguagem C++ e da biblioteca MFC. São apresentados, ainda, os procedimentos necessários para criar as diversas estruturas e objetos que compõem as representações utilizadas.

No capítulo 4 são definidas as normas de especificação para as classes de objeto utilizadas durante o processo de desenvolvimento. Para cada classe de objeto são definidos os elementos que compõem a sua especificação, detalhando o conteúdo de cada um e as condições para incluí-lo ou não.

As ferramentas utilizadas, o modo como são utilizadas e as dificuldades em inetgrá-las são apresentadas no capítulo 5.

Finalmente, o capítulo 6 avalia o processo, destacando os benefícios obtidos e apresentando as deficiências percebidas. As conclusões obtidas servem como motivação para continuar aprimorando o processo e desenvolvendo ferramentas mais modernas.

2. O Processo de Desenvolvimento Genérico

Para desenvolver sistemas com qualidade assegurada, é necessária uma metodologia. A metodologia determina:

- a seqüência de trabalho genérica, ou seja, a maneira como cada etapa do processo de desenvolvimento deve ser realizada (método)
- as normas técnicas a serem obedecidas em cada etapa
- os resultados a serem produzidos em cada etapa

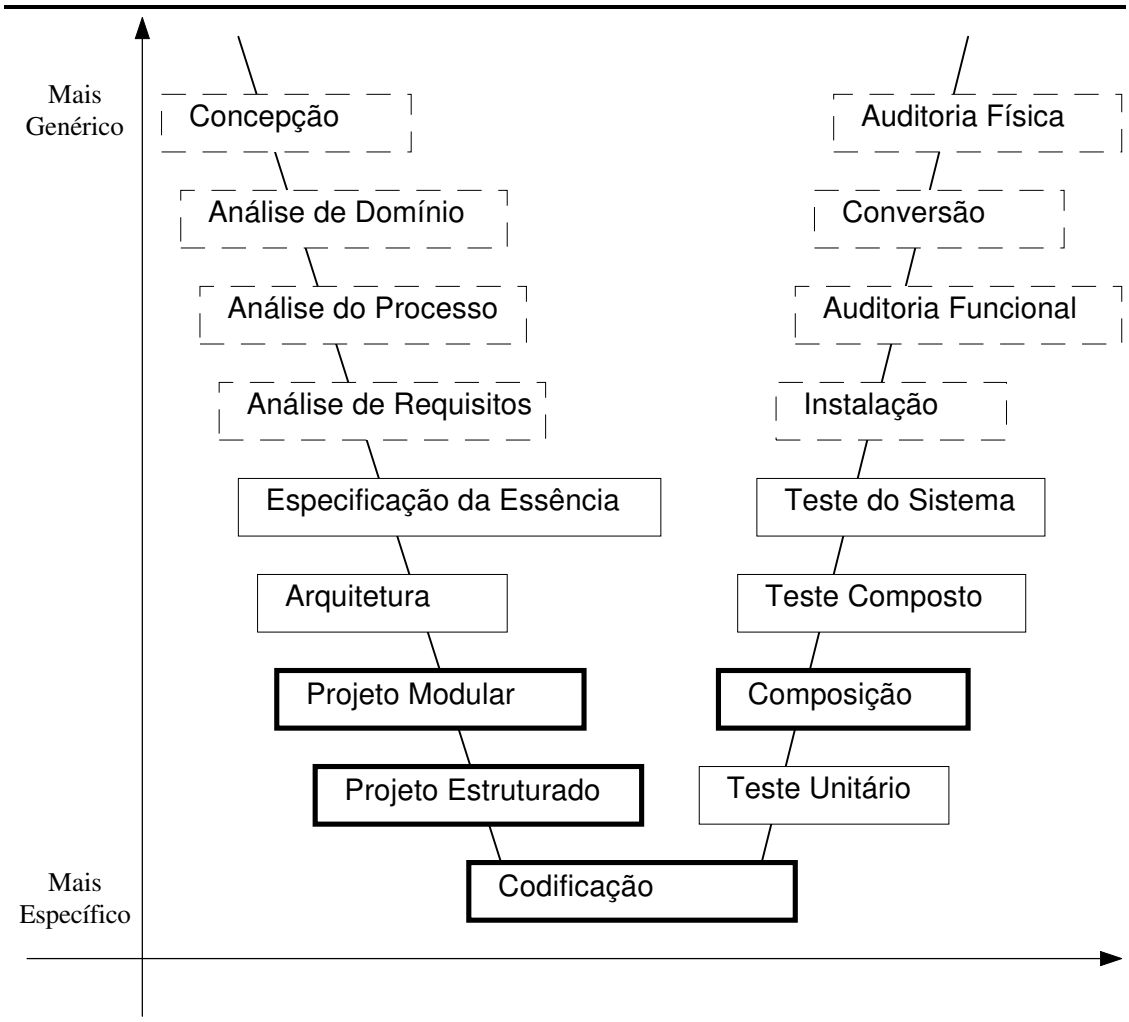


Figura 2-1 O processo de desenvolvimento.

Existem três instâncias para processos de desenvolvimento.

1. Definição do processo de modo genérico (é o assunto deste capítulo). Nesta instância são definidas as etapas, os dados necessários para a sua execução e a natureza dos produtos resultantes em cada uma delas. A figura 2-1 ilustra a seqüência de atividades típica do desenvolvimento de sistemas de software. Nesta figura não aparecem as atividades de suporte e gerência do processo.

2. Instanciação do processo específico para uma determinada classe de problemas, levando a um processo específico. É o objeto desta dissertação para o caso de aplicativos desenvolvidos para a plataforma Windows utilizando a biblioteca de classes MFC.
3. Instanciação do processo específico, levando a um plano de desenvolvimento de um determinado software.

O processo de desenvolvimento específico proposto por esta dissertação não cobre todas as etapas da figura 2-1 com a mesma profundidade. As etapas indicadas com borda tracejada não recebem apoio explícito e fogem ao escopo do processo em questão. As etapas com borda fina recebem algum apoio enquanto que as etapas com borda grossa são fortemente apoiadas e formam o núcleo do processo específico descrito no capítulo 3.

Existem basicamente três estratégias para definir a seqüência de trabalho.

- *"Grand Design"* Essencialmente, representa um processo no qual cada etapa é realizada uma única vez em seqüência. Os requisitos do sistema devem estar bem definidos e entendidos para obter um bom resultado. A vantagem desta estratégia é que os sistemas antigos são substituídos pelos novos de uma só vez.
- *Incremental* Nesta abordagem são realizadas todas as etapas, até a análise de requisitos, inclusive, uma única vez. O restante do desenvolvimento é repetido para cada subsistema identificado. As vantagens são uma melhor distribuição dos recursos ao longo do tempo e um prazo menor para que os primeiros subsistemas estejam completos.
- *Evolutiva* Esta estratégia deve ser utilizada quando os requisitos não puderem ser completamente definidos no início. Neste caso, os requisitos são parcialmente definidos. O sistema resultante é utilizado para aprimorar os requisitos. Este ciclo é repetido adicionando-se novos requisitos e os serviços correspondentes, até que se atinja um sistema satisfatoriamente completo.

O processo de desenvolvimento apresentado pode ser utilizado seguindo qualquer uma destas estratégias. O sistema evolutivo é o mais interessante para a maioria das aplicações ([Boehm86] e [Brooks86]). Para que seja viável, é necessário que o tempo e o esforço para completar um ciclo do processo sejam pequenos (quando comparados ao tempo e esforço totais de desenvolvimento). Isto não quer dizer que o processo se reduz ao velho paradigma de desenvolvimento por tentativa e erro. O que temos é um conjunto de ferramentas dirigidas por uma metodologia que visa alcançar os seguintes objetivos:

- minimizar o tempo de propagação de uma alteração nos requisitos para o restante do sistema
- maximizar a consistência entre os diferentes componentes do sistema (especificações, código fonte, *help on-line*, documentação técnica, manuais)

Existem diversas propostas de processos de desenvolvimento, por exemplo: CMM do SEI/CMU [PCCW93], processo cascata [Boehm76], processo espiral [Boehm86] e os processos orientados a objetos de Booch [Booch93] e Rumbaugh [RBP+91]. Todos estes processo, de uma ou outra forma, utilizam as etapas descritas a seguir. A descrição aqui seguirá o modelo do processo cascata, meramente por razões de simplicidade. O objetivo desta descrição é estabelecer o contexto no qual estará inserido o processo de desenvolvimento específico definido no capítulo 3.

2.1 Etapas de um processo de desenvolvimento genérico

Cada etapa do processo de desenvolvimento deve gerar um conjunto de produtos intermediários bem definido. A seguir é descrita a natureza destes produtos, bem como as ferramentas utilizadas para gerá-los. O ambiente de desenvolvimento TALISMAN [STAA92] é utilizado para editar as diversas

representações. No entanto, precisa ser cuidadosamente configurado e programado para que os resultados de uma etapa possam servir efetivamente como entrada para a próxima.

2.1.1 Concepção

O objetivo da concepção é registrar uma idéia de desenvolvimento ou evolução (manutenção). Usualmente é um texto sucinto contendo uma pré-avaliação quanto à oportunidade de se conduzir o desenvolvimento ou uma evolução de um sistema.

2.1.2 Análise de domínio

Resultado: Identificação dos elementos genéricos que figurarão em todos os sistemas semelhantes ou em evoluções de um mesmo sistema.

A especificação de domínio fatora tudo o que é comum às instâncias de implementação, separando tudo que pode variar de uma instância para outra. A análise de domínio conduz à padronização de uma parcela significativa do sistema, reduzindo em muito os custos de evolução do sistema e/ou de desenvolvimento de novos sistemas.

A análise de domínio pode ser realizada em diversos níveis de abstração. Por exemplo, ao projetar uma estrutura de classes, pode ser utilizada para identificar classes abstratas.

2.1.3 Análise do processo

Resultado: Identificação do processo (industrial ou gerencial) que receberá apoio de um possível sistema automatizado.

Em linhas gerais, deseja-se identificar os organismos da empresa, os processos empresariais e as funções das pessoas que participam do sistema. Com relação às funções deseja-se saber porque são relevantes, que resultados (não informacionais) produzem, que dados (sistema de informação) necessitam e que dados (sistema de informação) produzem.

No caso de sistemas administrativos, pode-se utilizar a linguagem de representação *Análise da Organização.*

2.1.4 Análise de requisitos

Resultado: Identificação dos objetivos e dos requisitos essenciais (o que o sistema fará) e de transformação (que dificuldades e/ou deficiências o sistema deverá eliminar).

Os requisitos identificados deverão ser atingidos para que o sistema possa ser aceito. Cabe salientar que uma das grandes dificuldades de desenvolver sistemas satisfatórios é a ausência da definição do que vem a ser satisfatório, ou seja, falta a definição dos requisitos.

Nesta etapa pode-se utilizar a linguagem de representação *Análise de Requisitos*.

2.1.5 Especificação da essência

Resultado: Especificação do sistema sem preocupação com a sua implementação.

Ao especificar sistemas, são produzidos:

- Especificação estática: define as estruturas de dados utilizadas (tabelas, listas, arquivos, bancos de dados, etc.). Esta especificação é estática, uma vez que define os estados válidos que os dados e os periféricos de um sistema podem assumir. Para gerar a especificação estática, TALISMAN utiliza a linguagem de representação *Entidades e Relacionamentos*.
- Especificação funcional: define como um estado válido (dados de entrada satisfazendo determinadas regras, assertivas de entrada, pré-condições) é transformado em outro (dados de saída satisfazendo determinadas regras, assertivas de saída, pós-condições), sem se preocupar com possíveis outras transformações realizadas em paralelo, ou em múltiplas ocorrências de um determinado processador. Esta especificação é funcional, uma vez que determina os resultados

a serem produzidos para os diversos dados de entrada de um processo (função). Define, ainda, a composição (seqüenciação) de funções. Para gerar a especificação funcional, TALISMAN utiliza a linguagem de representação *Fluxo de Dados*.

- Especificação dinâmica: define a ativação e seqüenciação temporal da execução dos processos. Em particular, são definidos os eventos que disparam as funções a serem efetuadas. Para gerar a especificação dinâmica, TALISMAN utiliza a linguagem de representação *Estados e Eventos*. Para sistemas simples, sem paralelismo efetivo, TALISMAN permite utilizar a linguagem *Fluxo de Dados* através dos mecanismos de geração e processamento de eventos, ou, até, do uso de informação de controle contidas nos fluxos.

2.1.6 Arquitetura

Um dos objetivos da arquitetura é identificar o que será implementado, viabilizar a implementação com o mínimo de compromissos, e definir o processo de implementação. Ao definir o que será implementado, identifica-se os programas a serem desenvolvidos, aloca-se recursos computacionais aos programas, determina-se a existência multiprogramação, multiprocessamento e/ou de compartilhamento de recursos com outros programas, e definem-se todas as interfaces dos diversos programas. Estas informações são adicionadas às especificações usando as linguagens nas quais as especificações foram redigidas. Outro objetivo da arquitetura é a definição do processo de desenvolvimento, identificando os diversos subprodutos a serem gerados e as atividades que conduzem a sua produção. TALISMAN não oferece ferramentas especializadas para esta etapa. Pode-se, entretanto, utilizar a linguagem de representação *Gerência de Projeto* para definir a estruturação e precedência de produtos e tarefas. Estas estruturas podem ser fornecidas a aplicativos de planejamento e acompanhamento de projetos.

2.1.7 Modularização dos programas

Resultado: O conjunto de módulos que implementam os programas.

Tipicamente a especificação é transformada em um projeto estruturado que, por sua vez, é acrescido de operações de controle produzindo módulos independentes. O volume de esforço despendido nesta etapa depende do tamanho do sistema, da linguagem de programação que virá a ser utilizada, da disponibilidade ou possibilidade de se poder criar bibliotecas de módulos. O resultado deve definir com rigor os módulos a serem desenvolvidos, maximizando a reutilização de módulos já conhecidos e maximizando a possibilidade de reúso de módulos a serem desenvolvidos. Quanto maior a possibilidade de reúso, tão mais barato será desenvolver e manter outros sistemas. A linguagem de representação utilizada é *Estrutura Modular*.

2.1.8 Projeto estruturado e codificação

Resultados: A organização interna dos módulos e classes. A estrutura de código de cada componente dos módulos e classes. Diagramas de estados e eventos descrevendo o comportamento dinâmico de classes complexas. Máquinas de estados descrevendo a seqüênciação das etapas de funções complexas. Posteriormente, são acrescidos fragmentos de código aos componentes destas estruturas. As estruturas podem então ser linearizadas produzindo arquivos compiláveis contendo código fonte.

Nesta etapa também deve ser gerada a versão inicial tanto da documentação para o usuário, composta pelo manual do usuário, auxílio *on-line* e tutoriais eletrônicos, bem como da documentação técnica.

2.1.9 Teste de unidade

Resultados: Assegurar que cada módulo esteja corretamente implementado, tanto do ponto de vista estrutura (critérios de qualidade) com do ponto de vista operacional (teste).

O processo de desenvolvimento apresentado não estabelece nenhuma metodologia específica para testes. Recomenda-se, entretanto, um sistema padrão de teste por cobertura e teste funcional [Staa87].

2.1.10 Composição de módulos

Resultados: Agregar módulos formando um conjunto relevante do ponto de vista do desenvolvimento.

Em geral, são utilizadas ferramentas, tais como, *make* e *link* para criar e controlar a composição de programas formados a partir de arquivos de código fonte, tabelas de definições e arquivos com a especificação de diálogos, menus, ícones e outros recursos do sistema operacional.

2.1.11 Teste incremental

Resultados: Deve assegurar que os módulos foram compostos de modo correto e satisfazem os requisitos de funcionalidade estabelecidos nas especificações.

2.1.12 Teste de sistema

Resultados: Deve assegurar que cada programa opera conforme especificado, que o conjunto de programas se compõe conforme especificado e que a documentação, tanto do usuário como técnica, exista e esteja completa.

2.1.13 Instalação

Resultados: Deve garantir que o sistema foi transferido do ambiente de desenvolvimento para o de produção. O ambiente de produção pode ser muito diferente do ambiente de desenvolvimento. Aqui o sistema é testado nas condições em que será efetivamente utilizado. Se for um software de distribuição comercial, deve-se garantir que o sistema e o processo de instalação funcionam na variedade de plataformas e configurações para as quais foi projetado.

Esta etapa não é coberta de modo explícito pelo processo apresentado.

2.1.14 Auditoria funcional

Resultados: Deve assegurar que o sistema atende às necessidades do usuário, verificar se os benefícios esperados estão sendo alcançados, assegurar que o usuário consegue utilizar o sistema sem requerer assistência continuada, assegurar que a documentação de uso está em versão definitiva e corresponde exatamente ao sistema implementado.

Esta etapa não é coberta de modo explícito pelo processo apresentado.

2.1.15 Conversão

Resultados: Deve garantir que o sistema novo, ou evolução do sistema existente, esteja operando normalmente.

Esta etapa não é coberta de modo explícito pelo processo apresentado.

2.1.16 Auditoria física

Resultados: Deve assegurar que tudo o que foi prometido, foi efetivamente fornecido, assegurar que o controle de qualidade foi realizado e que a qualidade esperada foi alcançada, assegurar que a sistemática de manutenção está operacional, assegurar que o histórico do projeto foi fornecido, assegurar que toda a documentação e todo o código está catalogado e acessível à equipe de manutenção, e assegurar que a documentação técnica corresponde exatamente ao que está implementado.

Esta etapa pode se beneficiar do fato de que boa parte das especificações se encontram disponíveis de modo eletrônico e em formato de hiper-texto, facilitando muito sua inspeção.

3. O Processo de Desenvolvimento Específico

O processo específico está voltado para o desenvolvimento e a manutenção de programas em C++, podendo ser facilmente adaptada a programas C. O processo específico utiliza as ferramentas do ambiente TALISMAN e da biblioteca MFC (Microsoft Foundation Classes [MFC_25]). O processo foi experimentado com as versões 1.0, 2.0 e 2.5 de MFC. A classe de sistemas que podem ser desenvolvidos está restrita apenas pelo próprio ambiente Windows. Já que a maioria dos sistemas desenvolvidos para Windows dedica uma parcela significativa à interface humana, procurou-se dotar este processo de recursos para lidar com este tipo de projeto. Em particular, utilizam-se *diagramas de estados e eventos* para representar o comportamento de classes que manipulam mensagens e comandos.

3.1 Definição da terminologia utilizada

base de software (BSW)	é o nome genérico dos repositórios utilizados por TALISMAN. Os objetos armazenados neste repositório podem ser consultados a qualquer momento pela exploração de representações ou por meio de um dicionário de dados.
módulo	é uma unidade de compilação. Cada módulo implementa uma abstração bem definida. Idealmente esta implementação estará completamente incluída em um único módulo. Um módulo é implementado utilizando cinco classes de arquivos: módulo de especificação, módulo de documentação, módulo de definição, módulo de implementação e módulo objeto, todos definidos a seguir.
módulo de especificação	contém a documentação técnica necessária para que projetistas e programadores possam utilizar corretamente o módulo ao desenvolver um determinado programa. Não participa do processo de compilação.
módulo de documentação	contém informações das quais podem-se extrair componentes destinados a gerar arquivos de auxílio (*help files*) e componentes destinados a gerar documentos impressos (manuais do usuário).
módulo de definição	contém o código de interface do módulo. Pode conter comandos de inclusão de arquivos definidos pelo compilador, de módulos de definição de outros módulos – módulos servidores – e de arquivos de declarações de constantes.
módulo de implementação	contém o código encapsulado do módulo. Contém ainda comandos de inclusão de arquivos do compilador, do módulo de definição próprio, de módulos de definição de outros módulos e de arquivos de declarações de constantes. O módulo de implementação é fornecido ao compilador para gerar o correspondente módulo objeto.
módulo objeto	é o resultado da compilação de um módulo de implementação.
componente	é um conjunto de módulos que disponibiliza um serviço através de uma interface bem definida.

Deve-se salientar aqui a relação entre um módulo em um componente. Existem autores que se referem a componentes como unidades binárias que disponibilizam algum serviço por meio de um protocolo bem definido, por exemplo: CORBA, SOM, COM (OLE), OPENDOC, etc. Preferimos, aqui, denominar estes últimos como *objetos interoperáveis* [Valdés95]. As ferramentas utilizadas não apoiam diretamente a criação de componentes. Pode-se, entretanto, particionar o sistema de tal forma que cada base de software represente um componente. Para tal armazenam-se todos os dados referentes aos

módulos que formam o componente em uma mesma *BSW*. Desta forma pode-se facilmente estabelecer relações entre objetos de módulos diferentes que pertençam a um mesmo componente. Na atual versão de TALISMAN, não é possível estabelecer relações entre objetos que não pertençam ao mesmo repositório.

Não é possível dissociar *processo de desenvolvimento* de *trabalho em equipe*. O esquema utilizado para trabalhar com uma equipe de pessoas consiste no particionamento de módulos ou componentes em repositórios (BSW) diferentes. Para cada módulo ou componente é criado um repositório individual. Como TALISMAN não suporta controle de versões ou configurações, procura-se atribuir a cada elemento da equipe o controle sobre um conjunto de repositórios fixo. Quando um indivíduo necessita alterar um objeto contido em um repositório pelo qual não seja responsável, é gerada uma cópia do mesmo e as alterações são realizadas nesta cópia. As inconsistências podem ser muitas vezes resolvidas por meio de um mecanismo de importação e exportação disponível no ambiente TALISMAN. O capítulo 5 explica como isto pode ser feito.

3.2 Etapas do processo de desenvolvimento específico

Como já havia sido mencionado, este processo cobre somente algumas das etapas do processo genérico descrito no capítulo anterior. Dependendo da complexidade do sistema sendo desenvolvido, pode ser necessário realizar de forma cuidadosa todas as etapas inicias do processo genérico ou simplesmente produzir um documento com os requisitos básicos do sistema. Em qualquer destes casos, o ponto de partida para a utilização do processo em questão é o conhecimento do conjunto de módulos que devem ser implementados.

Assim como no processo genérico, cada etapa do processo de desenvolvimento específico deve gerar um conjunto de produtos intermediários bem definido. A ordem na qual estas etapas estão apresentadas, representa a seqüência ideal de execução. Na prática, as etapas podem ser realizas em qualquer seqüência. A experiência demonstra que as etapas são executadas várias vezes, formando ciclos. Nem todas as etapas são executadas em cada ciclo. Quando uma etapa é executada mais de uma vez, deve produzir resultados com qualidade cada vez melhor, até alcançar o patamar desejado.

A seguir é dada uma visão geral das etapas e dos produtos gerados, bem como das ferramentas utilizadas para gerá-los.

3.2.1 Modularização dos programas

Esta etapa pressupõe que o conjunto de módulos e classes que devem ser criados, já existe. Como os sistemas a serem desenvolvidos possuem, a princípio, uma parcela significativa dedicada à interface humana, e como o processo apresentado permite a utilização de diagramas de estados e eventos para o projeto de classes que implementam esta interface, sugere-se o seguinte plano genérico, dividido em três níveis, para a modularização do sistema e para o projeto de classes:

- No primeiro nível, defina as classes que realizam processamento de dados e independem de interface de usuário. Estas classes tipicamente criam e mantém estruturas de dados e arquivos. Caem nesta categoria também as funções de cálculo por ventura utilizados (por exemplo processamento numérico).
- No segundo nível, defina as funções e classes que efetuam processamento de apoio à interface do usuário. As classes neste nível tipicamente ativam diálogos Windows. As funções tipicamente calculam valores em função do estado da interface de usuário, por exemplo: o número de linhas que cabem na janela.
- No terceiro nível, defina as classes que implementam a parte principal da interface. Estas classes são, geralmente, derivadas das classes **CWnd**, **CFrameWnd** ou **CView** (da biblioteca

MFC) e fazem uso das funções e classes definidas no nível anterior. Estas classes podem ser implementadas utilizando diagramas de estados e eventos.

- Separe as classes do primeiro nível em módulos próprios. Caso não sejam muito complexas, as classes e funções do segundo nível podem ficar no mesmo módulo que as classes implementadas utilizando diagramas de estados e eventos.

A atividade principal a ser realizada nesta etapa, constitui em especificar determinados aspectos destes módulos e classes, produzindo os resultados descritos a seguir:

Resultados:

1. A estrutura inicial dos módulos que implementam os programas. Esta estrutura é composta pela definição (e não pela implementação) dos diversos elementos que compõem cada módulo. Devem ser definidas as classes, as estruturas de dados e as funções de cada módulo. Para cada um destes elementos deve ser determinado se ele pertence à interface do módulo ou se está encapsulado. Isto é, deve se definir a interface de cada módulo.
2. A estrutura inicial das classes utilizadas. Devem ser definidas as estruturas de dados associadas, o membros e os métodos de cada classe. Deve ser definida a estrutura de herança das classes, indicando como estas se relacionam com as bibliotecas de classes utilizadas. Devem ser identificadas as classes abstratas. O escopo de cada elemento das classes deve ser identificada, caracterizando deste modo a interface das mesmas.

A linguagem de representação utilizada é *Estrutura Modular*.

3.2.2 Projeto estruturado

Resultados:

Ao contrário da etapa anterior, esta está voltada para aspectos da implementação dos módulos e classes.

1. Utilizando a linguagem de representação *Estruturas Modulares* devem ser definidas as estruturas de decomposição que implementam os tipos, as estruturas de dados, e as funções de cada módulo. A figura 3-1 ilustra este tipo de estrutura.
2. Devem ser criados diagramas, utilizando a linguagem de representação *Fluxo de dados*, para as

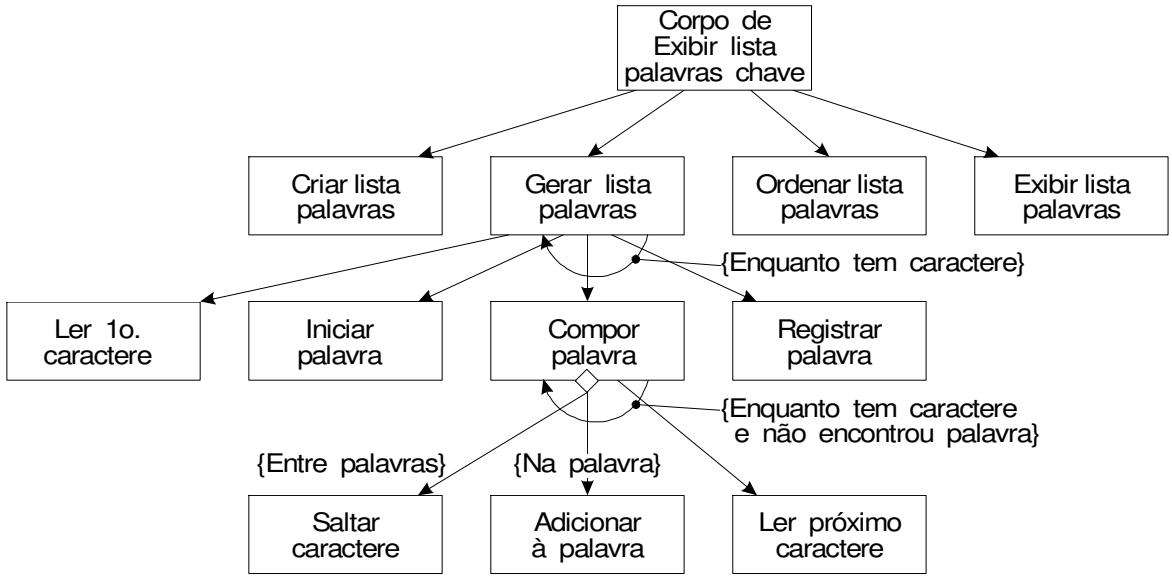


Figura 3-1 Estrutura de decomposição

funções cujo comportamento possa ser descrito por uma máquina de estados; veja a figura 3-2.

3. As estruturas de decomposição dos elementos que formam as classes também devem ser definidas.
4. Devem ser criados diagramas, utilizando a linguagem de representação *Estados e Eventos* para as classes derivadas a partir da biblioteca MFC que possam receber comandos (i.e.: derivadas de **CCmdTarget**; veja a figura 5-1) e cujo comportamento possa ser descrito por um diagrama de estados e eventos; veja a figura 3-3.

3.2.3 Codificação

Resultados: Os elementos criados nas etapas anteriores devem ser acrescidos de fragmentos de código. Deste modo, pode-se *linearizar* os diversos elementos para produzir:

- código compilável
- documentação técnica
- tabelas de definição de constantes
- *scripts* para a geração de arquivos *make*

3.2.4 Teste de unidade

Resultados: Assegurar que cada módulo esteja corretamente implementado, tanto do ponto de vista estrutural (critérios de qualidade) como do ponto de vista operacional (teste).

Para garantir a corretude do ponto de vista estrutural, são utilizados *validadores.* Estes são programas escritos em uma linguagem interna de TALISMAN.

O processo de desenvolvimento apresentado não estabelece nenhuma metodologia específica para testes. Tem-se utilizado com sucesso instrumentos de validação estrutural [NP-08], tal como especificado na figura 3-2, e testes de cobertura de estruturas de dados [Staa87]. Uma referência abrangente é [Jorg95].

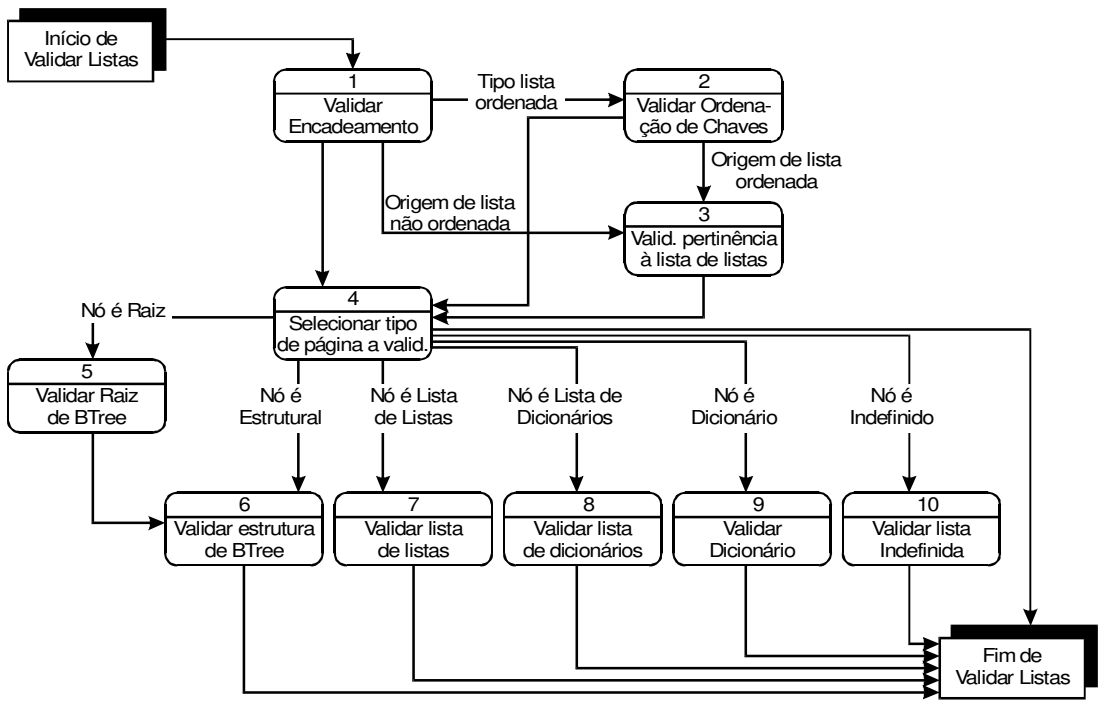


Figura 3-2 Máquina de estados utilizando a representação *Fluxo de Dados*

3.2.5 Composição de módulos

Resultados: Agregar módulos formando um conjunto relevante do ponto de vista do desenvolvimento.

Esta etapa pode parecer pouco relevante a primeira vista. Entretanto, uma das falhas mais comuns em programas (principalmente utilizando as linguagens C e C++) é a incorreta composição dos módulos. Por exemplo: módulos de definição não podem ser incluídos mais de uma vez em uma unidade de compilação. O particionamento inadequado de módulos de implementação e definição pode levar a erros de inconsistência de interface difíceis de serem detectados, mais precisamente, o módulo de implementação deve **incluir** o módulo definição (como se fosse um cliente) e não **duplicar** a interface no seu interior.

Para evitar estes erros:

- É definido um esquema robusto para o particionamento dos módulos em *definição* e *implementação*.
- O arquivo de entrada para o programa *make* é gerado automaticamente a partir da lista de arquivos de código fonte, tabelas e arquivos com a especificações de diálogos, menus, ícones e outros recursos do sistema operacional.

3.2.6 Teste de sistema

Resultados: Deve assegurar que cada programa opera conforme especificado, que o conjunto de programas se compõem conforme especificado e que a documentação de uso existe e está completa.

Como já foi mencionado, o processo de desenvolvimento apresentado não estabelece nenhuma metodologia específica para testes. A biblioteca MFC oferece ferramentas para gerar versões instrumentais dos sistemas com ela gerados. Estas incluem a emissão de mensagens para uma janela de inspeção (trace), validação de assertivas em tempo de execução, proteção de memória alocada dinamicamente e detecção de vazamento de memória. A maioria dos compiladores permitem instalar temporariamente a versão instrumentada do próprio ambiente Windows. Esta versão efetua uma rigorosa validação dos parâmetros fornecidos e, usada em combinação com ferramentas de depuração (por exemplo *CodeView*), permite interceptar GPF's (*General Protection Fault's*) retornando o controle ao *debugger*.

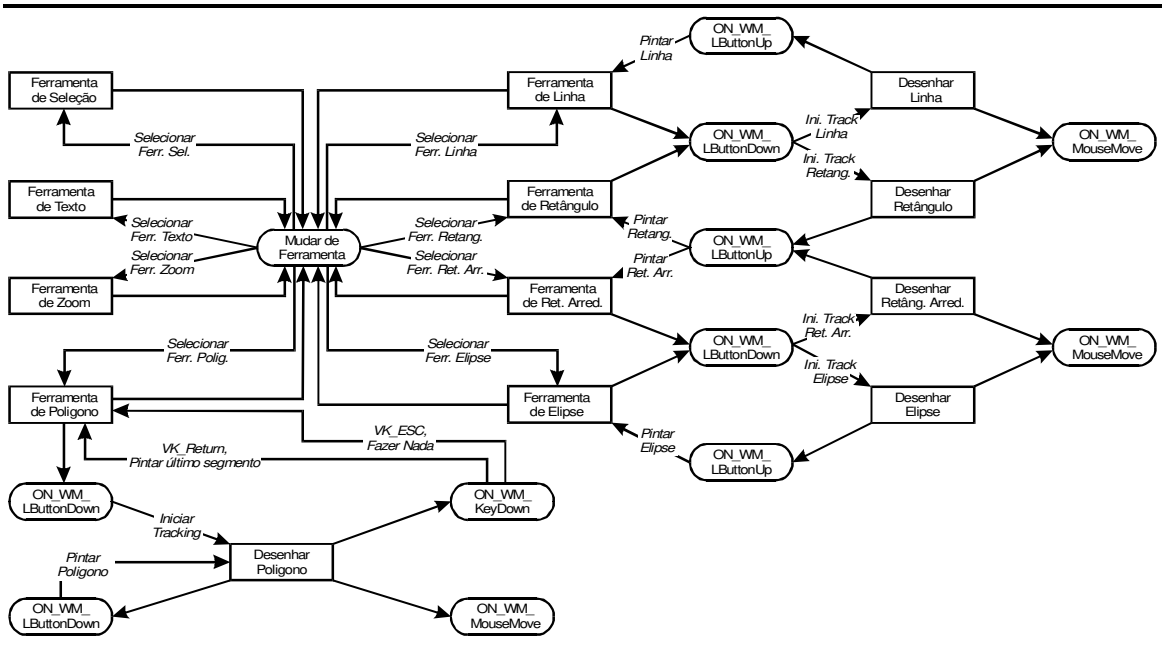


Figura 3-3 Diagrama de Estados e Eventos

3.3 As Estruturas Utilizadas

O processo de desenvolvimento específico está centralizado no uso de três tipos de estruturas para representar diversos aspectos dos sistemas que podem ser construídos. São elas: *Estruturas de decomposição sucessivas*, *Diagramas de estados e eventos* e *Máquinas de estados*. Estas estruturas são construídas e mantidas por meio de editores gráficos utilizando TALISMAN. Como será visto mais tarde, estes editores também permitem a combinação destas estruturas com código fonte, permitindo assim, compor os arquivos compiláveis a partir do repositório.

Tipo de estrutura	Objetivo	Quando deve ser utilizada
Estruturas de Decomposição Sucessivas	♦ especificar a estrutura do código de funções (veja a figura 3-1) ♦ especificar a estrutura de tipos de dados compostos	sempre
Máquinas de Estados	especificar a seqüência de tomada de decisões em funções	quando a seqüência de tomada de decisão for complexa e interdependente
Diagramas de Estados e Eventos	especificar o comportamento de classes derivadas a partir da biblioteca MFC e que possam receber comandos (i.e.: derivadas de **CCmdTarget**; veja a figura 5-1)	quando a complexidade[2] da classe for suficiente para justificar o esforço de se criar e manter um diagrama

Tabela 3-1 Tipos de estrutura utilizadas

A utilização de estruturas que tenham uma fácil interpretação gráfica para especificar sistemas tem como benefícios:

- melhorar a legibilidade das especificações
  Especificações puramente textuais não tem como expressar muitos dos aspectos importantes em sistemas de software.
- aumentar a produtividade das equipes de desenvolvimento
  a utilização de estruturas permite desenvolver sistemas usando técnicas conhecidas, por exemplo:
  - ♦ refinamentos sucessivos (top-down). A partir de um objeto genérico cria-se um conjunto de objetos mais especializados que detalham o objeto abstrato original. Este processo é repetido para os novos objetos, até que cada um seja suficientemente simples.
  - ♦ agregação (bottom-up). Procura-se identificar semelhanças em um conjunto de objetos de modo que estes possam ser agregados em um conjunto de objetos mais abstratos. Este processo é repetido para o conjunto de objetos resultantes, até que se identifique um conjunto em que cada objeto seja independente.
- aumentar a manutenibilidade dos sistemas
  Ao utilizar diagramas, esquemas ou outras representações para especificar o sistema, sem a utilização de ferramentas apropriadas, pode-se prejudicar a manutenibilidade. Entretanto, se for utilizada uma ferramenta que permita
  - ♦ criar e manter as representações em qualquer seqüência
  - ♦ manter a consistência entre as diversas representações

[2] Um critério possivel para medir esta complexidade é o número de estados necessário para descrever a classe. Por exemplo: se forem necessários três ou mais estados, então a classe será considerada suficientemente complexa para a criação de uma diagrama.

- navegar nas relações entre os objetos que compõe o sistema
- compor código compilável a partir das estruturas

então estaremos aumentando enormemente a manutenibilidade do sistema.

- fomentar o reaproveitamento de componentes
  Para reaproveitar é necessário identificar os componentes. A utilização de estruturas permite esta identificação.

```
Nome do evento: ON_WM_LBUTTONDOWN
Sigla

Descrição
   Mensagem do ambiente Windows.
Especificação dos dados e estados de entrada
   Parâmetros recebidos
      CPoint Ponto: contem as coordenadas que definem a
      posição aonde se encontrava o cursor do mouse no
      instante que o botão da direita foi pressionado.
   Variáveis globais do módulo recebidas

   Variáveis globais de outros módulos recebidas

   Arquivos de entrada

   Variáveis globais encapsuladas recebidas

Especificação da interface com usuário
   Diálogos realizados com o usuário

   Dados exibidos para o usuário

   Mensagens exibidas para o usuário

   Feedback de estado exibido para o usuário

Especificação dos dados e estados de saída
   Variáveis globais do módulo retornadas

   Variáveis globais de outros módulos retornadas

   Arquivos de saída gerados ou atualizados

   Variáveis globais encapsuladas alteradas

Requisitos do evento

Hipóteses do evento

Observações

Declaração: protótipo da função que implementa o evento
   afx_msg void OnLButtonDown( UINT, CPoint )
Declaração: cabeçalho da função que implementa o evento
   void JEM_JanEdtMold::OnLButtonDown( UINT, CPoint Ponto )

Nome da macro no mapa de mensagens
   ON_WM_LBUTTONDOWN()
Classe à qual pertence a função associada ao evento
   JEM - Janela de edição do editor de molduras
Cabeçalho da classe
   class JEM_JanEdtMold : public CWnd
```

Figura 3-4 Fragmento de especificação hiper-texto

3.4 A combinação de Código e Estrutura

A utilização eficaz de estruturas requer um mecanismo simples para associar especificações textuais aos objetos que compõem estas estruturas. TALISMAN permite definir *formulários* de preenchimento para cada classe de objeto que compõe cada representação. Estes formulários são formados por:

- campos constantes (não editáveis)
- textos curtos (uma única linha)
- textos longos
- referências simples a outros objetos
- referências a um conjunto de outros objetos (relações)

A figura 3-4 mostra um fragmento do formulário utilizado para especificar um objeto da classe *Eventos*. Nesta figura, os campos do tipo referência estão indicados em fundo cinza e os campos constantes em negrito. Formulários de especificação, assim como outros tipos de formulário que serão vistos mais tarde, são programáveis. Para programar formulários é utilizada uma linguagem própria de TALISMAN. O conjunto de *programas de formulário* constitui uma parte fundamental do processo de desenvolvimento apresentado.

3.5 Os Procedimentos Básicos

A descrição das etapas do processo de desenvolvimento específico (seção 3.2) fornece uma visão do tipo de atividade realizado em cada etapa e dos resultados que devem ser produzidos. Esta visão não captura os *procedimentos* necessários para criar e manter os objetos que compõem os sistemas desenvolvidos. Estes procedimentos podem ser divididos em tarefas. Assim como as etapas anteriormente vistas, as tarefas destes procedimentos, não possuem uma seqüência de execução fixa. Na prática, tem-se um processo bastante flexível, limitado apenas pela precedência de criar um objeto antes de poder referência-lo por outro.

Não é objetivo deste texto indicar como utilizar o ambiente TALISMAN para efetuar as referidas tarefas. Esta descrição encontra-se de forma detalhada no texto *"Ferramentas TALISMAN para o desenvolvimento de programas Windows, utilizando C++ e MFC"* [Hübsch95]. É necessário, entretanto, descrever a correspondência entre as representações, estruturas e classes de objeto utilizadas com os elementos da linguagem C++, do ambiente Windows e da biblioteca MFC.

O processo de construção de objetos no ambiente TALISMAN pode ser dividido em dois tipos de operação:

1. *Definição de atributos do objeto*. Estes atributos podem ser descrições, fragmentos de código, comentários, etc.
2. *Definição de relações com outros objetos*. Uma parte substancial da especificação de um objeto refere-se aos seus relacionamentos com outros objetos. Um módulo, por exemplo, é composto fundamentalmente por referências aos objetos que o compõem. A combinação de texto e referências toma a forma de um agregado semelhante a um hipertexto no qual as relações podem ser editadas dinamicamente. Ao contrário de hipertextos convencionais [Stratt89], todas estas relações possuem semântica própria. A classe de objetos *módulos*, por exemplo, possui duas relações diferentes com a classe de objetos *tipos de dados*, uma para os tipos de dados exportados e outra para os tipos de dados encapsulados.

A seguir são descritas tarefas que são comuns a todos os procedimentos.

Especificar A especificação é completa e não se preocupa com a forma de implementação. Cada classe de objeto possui um conjunto de ítens que

devem constar na sua especificação. Alguns destes ítens são comuns a todas as classes de objetos:

- descrição (objetivo)
- requisitos assegurados
- hipóteses de uso assumidas
- restrições de implementação

Para uma visão completa sobre como especificar módulos e seus componentes, veja: *"Regras e recomendações para a inclusão de especificações no código de programas C ou C++"* [NP-05].

Definir a Arquitetura — A arquitetura da função descreve o modo como a função é implementada. Ao especificar dizemos "o que fazer", ao definir a arquitetura dizemos "como fazer".

Referenciar <objeto> — É a ação pela qual se estabelece uma relação entre dois objetos. Geralmente, temos um *objeto corrente* sendo editado e desejamos relacioná-lo a outro objeto. Por exemplo, ao editar um módulo deseja-se referenciar uma função que deve pertencer ao módulo.

Verificar — A verificação de um objeto consiste em examinar a qualidade da representação em si. Ou seja, examinar se está corretamente redigida, se obedece às normas (veja o capítulo 4 para uma descrição destas normas), se é legível e se é inteligível. TALISMAN oferece recursos para linearizar uma função e permitir a edição dos fragmentos de cada bloco como se formassem um texto contínuo, facilitando deste modo a verificação. Uma etapa obrigatória no processo de verificação é a correta compilação do objeto sendo verificado.

Validar — Como já foi dito anteriormente, o ambiente TALISMAN deve ser configurado e programado para que possa ser utilizando no processo de desenvolvimento apresentado. A validação é a ação de um programa escrito na linguagem interna de TALISMAN cujo objetivo é aplicar regras de validação a um objeto, gerando um laudo. Estas regras não são capazes de examinar a qualidade dos atributos de um objeto (ver item anterior), mas podem detectar a ausência de atributos obrigatórios e testar a consistência entre relações.

3.5.1 A criação de módulos

Não existe o *conceito de módulo* nas linguagens C ou C++, na forma como é utilizado neste texto.

> *Um programa escrito em C++ consiste em um ou mais "arquivos fonte", cada um contendo parte do texto que forma o programa. Um arquivo fonte, juntamente com os seus "arquivos de inclusão" (arquivos incluídos utilizando a diretiva* ***#include*** *do pré-processador), excluindo qualquer seção de código removida por diretivas de compilação condicional (p.ex.:* ***#if****), é denominado uma "unidade de tradução".* [C++_93]

Módulos constituem, entretanto, a classe de objeto central do processo de desenvolvimento apresentado. Módulos representam as "unidades de tradução" de C++, introduzindo restrições à sua utilização.

Módulos são sempre particionados em:

módulo de definição — contém as declarações das classes, tipos e dados exportados. Isto é, define a interface do módulo. Deve ser implementado sob a forma de um arquivo de inclusão com a extensão ".hpp".

módulo de implementação contém a implementação tanto dos elementos declarados na interface, como dos elementos internos (encapsulados). Como pode ser observado na figura 3-5, os elementos de interface não são redeclarados na implementação. Ao invés disto, o próprio módulo de definição é incluído na implementação, como foi explicado na seção 3.2.5.

As tarefas necessárias para criar um módulo são:

1. Especificar o módulo.

   Alem dos ítens genéricos, para módulos de definição, deve ser incluído um texto descrevendo como utilizar o módulo.

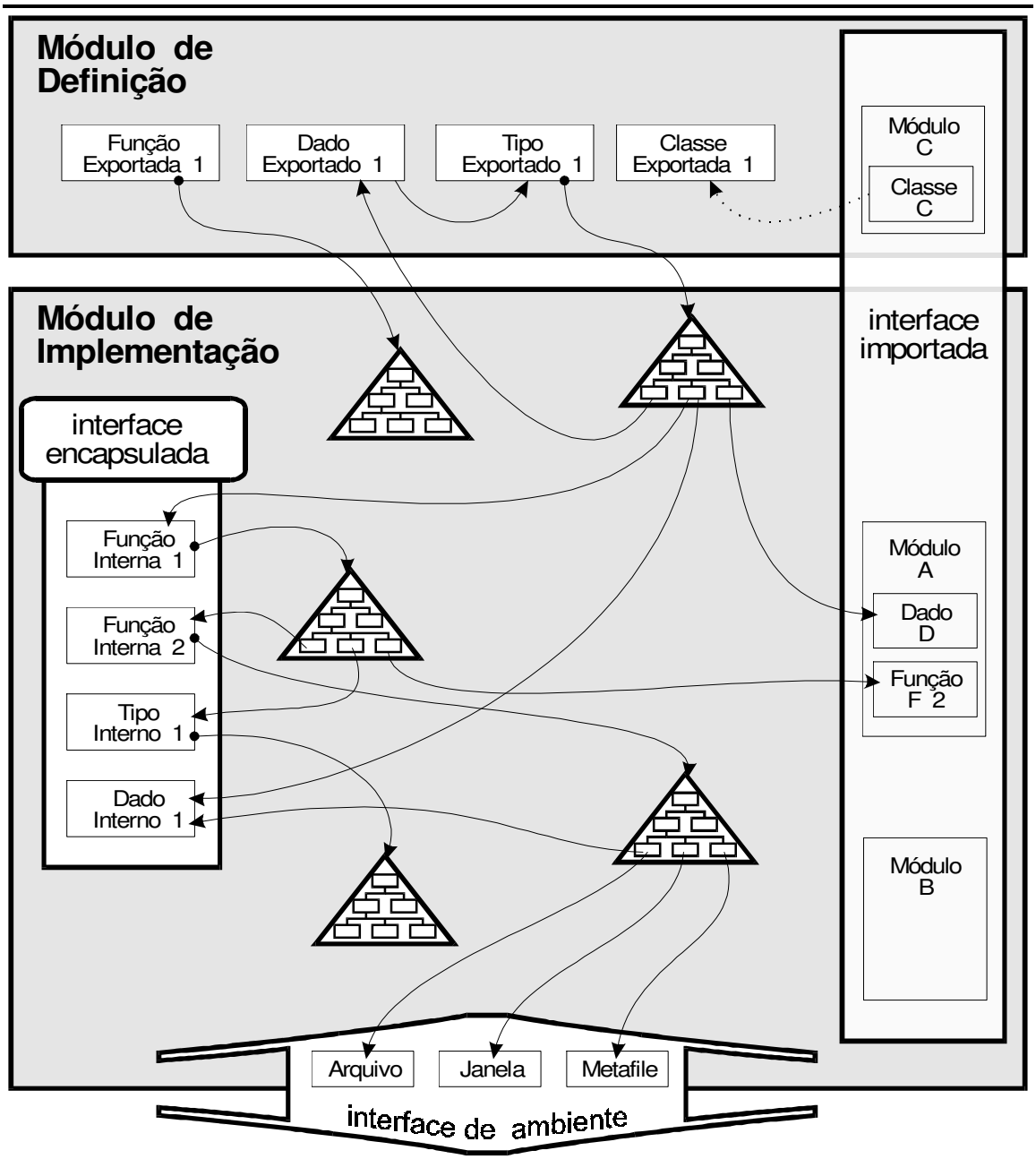


Figura 3-5 Arquitetura de um módulo

2. Definir a arquitetura do módulo.
   2.1. Definir a interface do módulo (módulo de definição)
      - definir os módulos de definição requeridos pela interface,
      - definir as constantes exportadas,
      - referenciar os tipos de dados, atributos, classes e função exportadas pelo módulo.
   2.2. Definir a composição interna do módulo (módulo de implementação)
      - definir os módulos de definição requeridos pela implementação,
      - definir as constantes internas,
      - referenciar os tipos de dados, atributos, classes e função internas ao módulo.
3. Linearizar o módulo de definição e de implementação e verificar o código.
4. Validar o módulo.

3.5.2 A criação de classes

Ao contrário de módulos, a classe de objeto *classes* possui uma correspondência direta com o conceito de classes da linguagem C++.

As tarefas necessárias para criar uma classe são:

1. Especificar a classe.

   Alem dos ítens genéricos, deve ser incluído um texto descrevendo como utilizar a classe. Por exemplo, deve-se indicar se a classe foi projetada para instanciação direta de objetos, ou se o cliente deve inicialmente derivar uma nova classe.
2. Definir a arquitetura da classe.
   2.1. Definir o cabeçalho físico.
   2.2. Referenciar os tipos de dados, atributos e métodos definidos pela classe.
   2.3. Para classes cujo comportamento possa ser descrito por um diagrama de estados e eventos deve-se estabelecer a relação com este diagrama (veja a seção 3.5.6).
3. Linearizar a classe e verificar o código.
4. Validar a classe.

3.5.3 A criação de funções ou métodos

Funções ou métodos (considerados aqui como funções pertencentes a uma classe) são representadas em TALISMAN utilizando duas classes de objetos: *pacote* e *bloco de código*. Isto tem como objetivo aumentar a flexibilidade do modo como são representadas chamadas de funções. A figura 3-6 ilustra este mecanismo. Uma outra forma de representação será vista na seção 3.5.6.

As tarefas necessárias para criar uma função ou método são:

1. Especificar a função.

   Além dos ítens genéricos, devem ser incluídos textos descrevendo:
      - a especificação dos dados e estados de entrada,
      - a especificação dos dados e estados de saída,
      - a descrição da interface com o usuário.

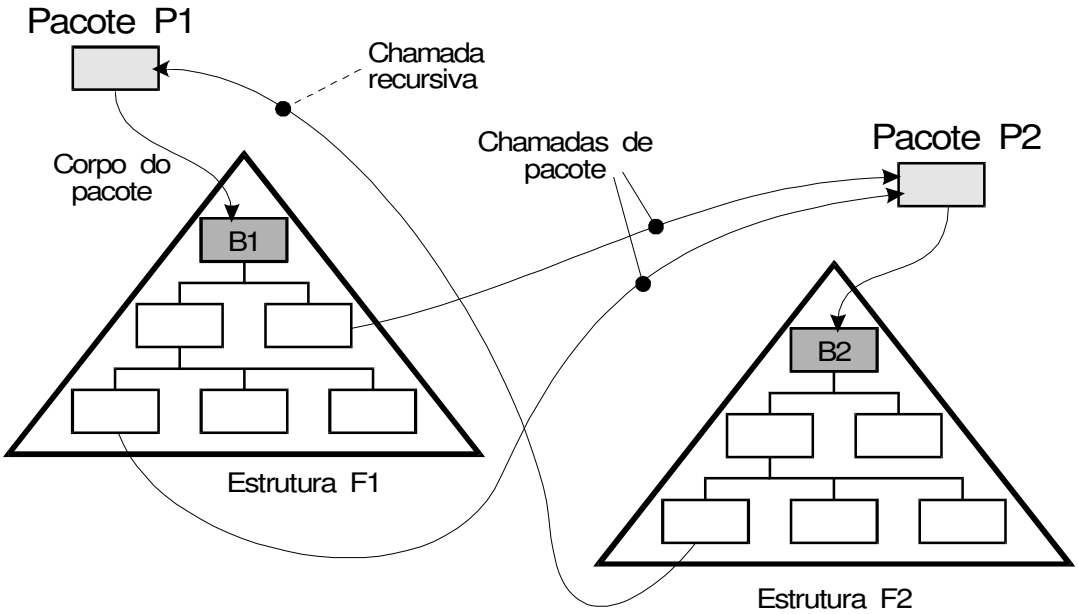


Figura 3-6 Chamada de funções utilizando pacotes.

métodos de classes devem incluir ainda descrições especiais caso:

- redefinam um método de uma classe base (reescrita),
- possuam o mesmo nome que outro método (sobrecarga),
- sejam virtuais.

2. Definir a arquitetura da função.
   2.1. Definir o cabeçalho físico.
   2.2. Definir ou referenciar as estruturas de dados a utilizar.
   2.3. Criar a estrutura de decomposição do algoritmo como na figura 3-1.
   2.4. Caso seja necessário, podem ser definidas *assertivas* [NP-08] de entrada e saída para cada bloco criado.
   2.5. Para cada bloco criado devem ser adicionados fragmentos de código que implementam a abstração que representam.
   2.6. Quando um *bloco de código* contiver um fragmento de código que representa uma chamada a uma função e esta estiver representada por um *pacote*, a relação entre o *bloco de código* e o *pacote* deve ser estabelecida, veja a figura 3-6. Esta "chamada" não é necessariamente no sentido literal da palavra, isto é: passagem imediata do controle de execução. Ela pode representar, por exemplo, o envio de uma mensagem. O pacote, neste caso, representaria a função que trata a mensagem (*message handler*).
3. Validar os objetos que compõem a estrutura de decomposição.
4. Verificar o código.

3.5.4 A criação de tipos de dados

Em C++ podemos classificar tipos de dados em:

1. Tipos fundamentais, por exemplo: *int*, *float* ou *double*
2. Tipos derivados diretamente
   2.1. Vetores
   2.2. Funções
   2.3. Ponteiros
   2.4. Referências
3. Tipos derivados compostos
   3.1. Classes
   3.2. *Structs*
   3.3. *Unions*
   3.4. *Enums*
4. *Typedefs* declara tipos (mas não é um tipo)

Ao criar tipos em TALISMAN utiliza-se uma classificação diferente. Tem-se tipos simples e tipos estruturados. Esta classificação corresponde a existência ou não de uma estrutura de decomposição para o tipo. A tabela 3-2 indica esta correspondência. Tipos estruturados possuem uma raiz que contém o cabeçalho e folhas que contem os elementos que compõem o tipo. Tipos simples possuem apenas a raiz.

Assim como funções são representadas utilizando duas classes de objetos, pacotes e blocos de código, tipos de dados são representados utilizando as classes de objeto *tipo* e *bloco de dados*. O objetivo é o mesmo que para funções, facilitar a representação de "chamadas" entre tipos e blocos de dados. A "chamada", neste caso, representa a referência (utilização) do tipo por um fragmento de código contido no bloco de dado, veja a figura 3-7.

As tarefas necessárias para construir um tipo de dados são:

1. Especificar o tipo

Tipos fundamentais	**São representados como atributos (veja a seção 3.5.5)**
Tipos derivados diretamente	
Vetores	São representados como atributos (veja a seção 3.5.5)
Funções	São representadas por uma estrutura própria (veja a seção 3.5.3)
Ponteiros	São representados como atributos (veja a seção 3.5.5)
Referências	São representados como atributos (veja a seção 3.5.5)
Tipos derivados compostos	
Classes	São representadas por uma estrutura própria (veja a seção 3.5.2)
Structs	São representados como tipos estruturados
Unions	São representados como tipos estruturados
Enums	São representados como tipos estruturados
Typedefs	**São representados como tipos simples**

Tabela 3-2 Correspondência entre tipos C++ e "tipos TALISMAN"

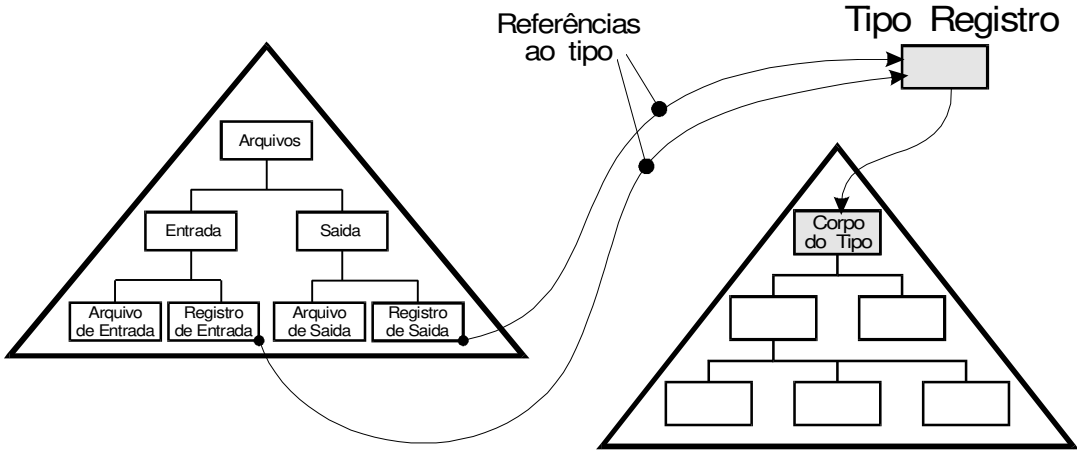


Figura 3-7 Relação entre tipos e blocos de dados.

2. Definir a arquitetura do tipo
   2.1. Definir o cabeçalho físico
   2.2. Para tipos estruturados deve ser definida a estrutura de decomposição, isto é, os elementos que compõem o tipo.
   2.3. Para tipos complexos (tipos que representam estruturas de dados complexas), devem ser incluídas as *assertivas estruturais* [NP-08] do tipo.
   2.4. Quando um objeto da classe *bloco de dados* contiver um fragmento de código que referencia um tipo de dados, e este estiver representado por um objeto da classe *tipo*, a relação entre o *bloco de dados* e o *tipo* deve ser estabelecida.
3. Validar o tipo de dados.
4. Verificar o código.

3.5.5 A criação de atributos

O termo atributo é utilizado para denominar de modo genérico os seguintes elementos da linguagem C++: variáveis de tipos fundamentais ou cujo tipo já foi declarado, vetores, ponteiros e referências (veja a tabela 3-2). A classe de objetos utilizada em TALISMAN para representar atributos é *bloco de dados*. Assim como tipos de dados, atributos podem ser representados por uma estrutura de blocos de dados. Para tipos de dados, esta estrutura reflete a estrutura do tipo, enquanto que para atributos a estrutura de decomposição serve meramente para fins de organização, como em uma estrutura de diretórios. O código reside apenas nas folhas da estrutura. Outra diferença fundamental é que a raiz da estrutura que representa um tipo de dados está relacionada a um objeto da classe *tipo,* veja a figura 3-7.

As tarefas necessárias para construir um atributo são:

1. Especificar o atributo.

   Além dos ítens genéricos, devem ser incluídos textos descrevendo a *medida* e a *forma de validação* do atributo.
2. Definir a arquitetura do atributo
   2.1. Definir o cabeçalho físico (declaração e definição do atributo).
   2.2. Se o atributo for baseado em um tipo de dados definido pelo usuário, este deve ser referenciado.
3. Validar o atributo.
4. Verificar o código.

3.5.6 A criação de diagramas de estados e eventos

Diagramas de estados e eventos são utilizados para representar o comportamento de classes derivadas a partir da biblioteca MFC e que possam receber comandos. O esforço necessário para criar e manter estes diagramas é considerável, por isto, sua utilização só é recomendada quando a complexidade da classe for suficiente para justificar este esforço.

Ao contrário dos elementos apresentados até este ponto (módulos, classes, pacotes, etc.), diagramas de estados e eventos não possuem qualquer conexão com a linguagem C++. As seguintes classes de objetos são utilizadas para construir os diagramas:

Eventos[3] Correspondem a ações externas (*triggers*), provocando mudanças de estado e desencadeando ações (processos). Geralmente, mas não necessariamente, estão associadas a mensagens do ambiente Windows. Por exemplo: click do mouse, seleção de um item em um menu, interrupção disparada por um *timer*, etc.

Estados Cada estado identifica um conjunto de condições garantidamente válidas – *invariante do estado* – sempre que este estado for o estado corrente. É obrigação do desenvolvedor definir esta invariante e assegurar que cada evento garanta a validade de cada possível estado destino.

Processos Estão associados às ligações entre eventos e estados. Quando um evento possuir mais de um estado destino, condições de seleção definidas nos processos associados às ligações de saída, estabelecem qual dos estados destino será o novo estado corrente.

Um dos benefícios decorrentes da utilização destes diagramas é a facilidade de manutenção. Para alterar o comportamento de uma classe descrita por um diagrama, pode ser necessário apenas modificar o diagrama, religando estados e eventos.

Como foi mencionado na seção 3.5.3, funções ou métodos podem ser representados utilizando as classes de objeto *pacote* e *bloco de código*. Ao utilizar diagramas de estados e eventos para descrever o comportamento de uma determinada classe, representam-se os seus métodos de um dos seguintes modos:

forma pacote Utiliza-se um pacote relacionado à classe e uma estrutura de blocos de código associada ao pacote. Esta é a forma "convencional".

forma evento Utiliza-se um diagrama de estados e eventos relacionado à classe e instanciam-se eventos (que representam os métodos) neste diagrama.

Cada instância de evento pode ser ligada a zero ou mais estados origem e a zero ou mais estados destino. Estados origem e destino de uma instância de evento podem ser os mesmos. A direção da ligação determina se o estado é origem – ligação dirigida do estado para o evento – ou destino – ligação dirigida do evento para o estado –.

---

[3] Na atual versão de TALISMAN, utiliza-se o nome *Transição* no lugar de *Evento*.

Uma instância de evento pode ser:

1. *normal*: a instância de evento está ligada a pelo menos um estado origem e a pelo menos um estado destino. O processamento de uma instância de evento normal depende do estado corrente quando a mensagem correspondente ao evento for recebida. Uma instância de evento normal redefine o estado corrente em função das condições de seleção definidas nas ligações para estados destino.
2. *incondicional*: a instância de evento está ligada a nenhum estado origem e está ligada a pelo menos um estado destino. Uma instância de evento incondicional independe do estado corrente quando a mensagem correspondente ao evento for recebida. Uma instância de evento incondicional redefine o estado corrente em função das condições de seleção definidas nas ligações para estados destino.
3. *final*: a instância de evento está ligada a pelo menos um estado origem e está ligada a nenhum estado destino. O processamento de uma instância de evento final depende do estado corrente quando a mensagem correspondente ao evento for recebida. Uma instância de evento final não altera o estado corrente quando a mensagem correspondente ao evento for processada. Note que uma instância de evento que tenha como estado origem e estado destino o mesmo estado, não é uma instância de evento final, já que o valor do estado corrente será efetivamente redefinido, embora para o mesmo valor.
4. *pura*: a instância de evento está ligada a nenhum estado origem e a nenhum estado destino. Uma instância de evento pura independe do estado corrente quando a mensagem correspondente ao evento for recebida. Uma instância de evento pura não altera o estado corrente quando a mensagem correspondente ao evento for processada. Instâncias de eventos puras podem ser utilizadas para tratar mensagens cujo processamento independa do estado corrente e que não alteram o contexto requerido para o estado corrente. Finalmente, somente as mensagens processadas por eventos possuindo apenas instâncias puras, podem ser emitidas ao processar outras mensagens.

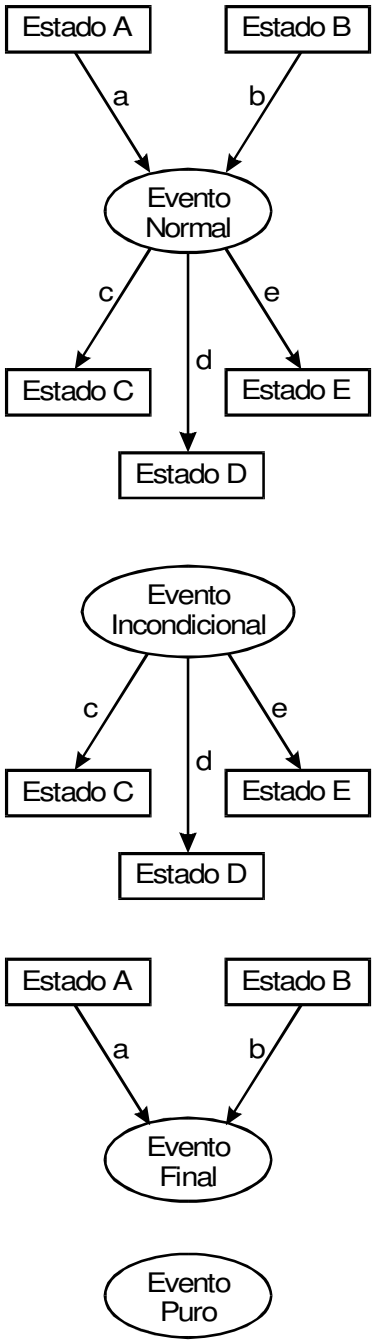


Figura 3-8 Tipos de eventos

Um mesmo evento pode ser instanciado de diversas maneiras em um mesmo diagrama. Ou seja, qualquer evento pode ter uma instância final e uma instância normal, ou outra combinação qualquer. Caso coexistam instâncias de eventos puras e/ou incondicionais com outras finais e/ou normais, as puras e/ou incondicionais somente serão ativadas caso o estado corrente for nenhum dos estados origens das instâncias finais e/ou normais. Adicionalmente, o estado destino ao terminar a execução do evento somente será redefinido pelas instâncias de evento incondicionais e/ou normais.

Cada ligação pode estar associada a um processo. No caso de ligações para estados destino, este processo deve definir as condições de seleção do estado destino. Para cada instância normal, somente uma das ligações poderá não ter processo ou código de seleção associado. Esta ligação será utilizada

para transitar para o estado destino, caso nenhuma das outras condições seja verdadeira. No conjunto de todas as ligações que emanam de uma ou mais instâncias de evento incondicionais, somente uma poderá não ter processo ou código de seleção associado.

Na *forma evento*, o código do método é derivado a partir da combinação de

- fragmentos de código contidos no evento
- fragmentos de código contidos nos estados ligados ao evento
- fragmentos de código contidos nos processos associados às ligações entre os estados e o evento
- configuração destas ligações.

O esquema de código gerado por TALISMAN, descrito a seguir, define o modo como os diversos fragmentos de código se combinam.

1. *código preâmbulo do evento*. Este código deve ser utilizado com cuidado. Tipicamente contém processamento global incondicional necessário antes de iniciar a execução do evento propriamente dito.
2. *código de declaração de dados do evento*. Tipicamente deve conter construtores e declarações de dados.
3. *seleção do estado origem*. Este código é gerado em função do estado origem de cada ligação que atinja alguma instância de evento. Caso a ligação correspondente esteja associada a uma instância de processo contendo código executável, este será incorporado ao código gerado. Caso não exista instância de evento pura ou incondicional, será gerada um retorno incondicional, que será executado caso o estado corrente ao entrar seja nenhum dos estados origem conhecidos.
4. *código do evento*. Este código realiza o processamento do evento válido para todas as instâncias e condições de ativação. Note que todas as instâncias efetuarão este código, independentemente do estado origem.
5. *código de seleção do estado destino*. Aqui será selecionada a instância e, para cada instância selecionada, será gerada a correspondente seqüência de seleção de estado destino. A seleção de instância é realizada considerando os estados origem. Caso a ligação correspondente esteja associada a uma instância de processo contendo código executável, este será incorporado ao código gerado. Da mesma forma será incorporado o código do estado destino. Desta forma, assegura-se a semântica definida no diagrama.
6. *código de liberação dos recursos do estado origem*, caso este seja diferente do estado destino
7. *código de estabelecimento da invariante do estado destino*, caso este seja diferente do estado origem
8. *código do estado destino*, independentemente dos estados origem e destino serem iguais ou diferentes.
9. *código "pós-âmbulo" do evento*. Este código deve ser utilizado com cuidado. Tipicamente deve conter destrutores e comandos **return** caso o tipo retornado não seja **void**.

As tarefas necessárias para construir um diagrama de estados e eventos são:

1. Criar o diagrama
   1.1. Criar a classe cujo comportamento será descrita pelo diagrama (veja a seção 3.5.2).
   1.2. Criar um evento origem para o diagrama e relacioná-lo à classe.
   1.3. Criar o diagrama a partir do evento criado no item anterior.
2. Especificar o diagrama

2.1. Criar os estados e os eventos.

2.2. Criar as ligações entre os eventos e os estados. Quando houver ambigüidade no estado destino, deve-se associar um processo a cada ligação de saída do evento (exceto possivelmente pelo estado *default*).

2.3. Especificar os eventos, estados e processo criados.

3. Definir a arquitetura.

3.1. Definir o cabeçalho físico de cada evento.

3.2. Escrever o código associado aos evento, estados e processos.

4. Validar o diagrama.

5. Verificar o código. Isto pode ser feito *linearizando* a classe associada ao diagrama.

3.5.7 A Geração de Código Fonte

Após definir a estrutura inicial dos módulos e classes (seção 3.2.1), deve-se detalhar a implementação destas estruturas (seção 3.2.2) e inserir o código fonte e outras especificações textuais utilizando os formulários adequados a cada classe de objeto (seção 3.2.3). Resta gerar os arquivos de código fonte correspondentes. Para isto utilizam-se programas de formulários específicos para esta finalidade. O funcionamento destes programas, denominados *linearizadores*, será descrito no capítulo 5. Uma vez gerados os arquivos contendo o código fonte, monta-se o programa executável utilizando um programa de composição (por exemplo: make).

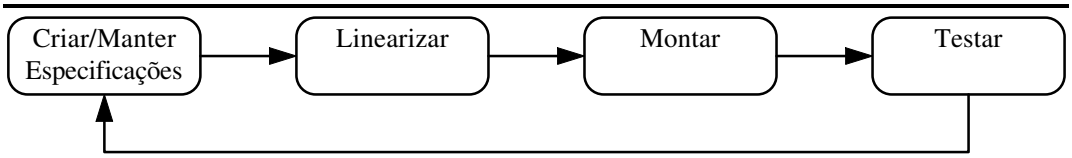


Figura 3-9 Ciclo típico para manutenção ou criação de programas

Ao realizar qualquer manutenção no sistema, os arquivos contendo código fonte não devem ser editados diretamente. Em vez disto, alteram-se as especificações que deram origem a estes arquivos e efetua-se um novo processo de *linearização* e composição. Deste modo, evitam-se inconsistências entre as especificações e a instanciação do sistema.

4. Normas de Especificação

O capítulo 3 descreveu, entre outros ítens, os procedimentos necessários para criar as diversas estruturas e objetos que compõem as representações utilizadas. Em conjunto com esta descrição são apresentados, de forma superficial, elementos de especificação destes objetos. Este capítulo descreve de modo detalhado estas especificações, definindo:

a sua "natureza"	**Porquê e quando** deve ser incluída determinada especificação. Qual deve ser o benefício alcançado ao se incluir uma determinada especificação.
o seu conteúdo	**O que** deve ser especificado, isto é, a descrição precisa do que deve ser especificado.
o procedimento	**Como** deve ser especificado. Que critérios, regras ou dependências devem ser observadas ao especificar.

Estes ítens, apesar de não serem independentes, orientam o modo como serão apresentados os diversos elementos que compõem as especificações de cada classe de objetos utilizada.

Deve-se distinguir aqui entre duas definições de especificação para as classes de objeto utilizadas:

especificação textual	refere-se a um conjunto de textos descritivos pertencentes a um objeto. Este é o tipo de especificação de interesse neste capítulo.
especificação completa	é a especificação textual acrescida de fragmentos de código e das relações com outros objetos. Estas últimas se manifestam na forma de: • referências • instâncias em diagramas • ligações entre instâncias em diagramas Estas relações já foram descritas no capítulo anterior.

Inicialmente, serão descritos os elementos de especificação comuns a todas as classes de objetos utilizadas, para então descrever os elementos particulares de cada classe.

4.1 Conteúdo de uma especificação

Qualquer que seja o sistema ou componente do sistema precisamos saber:

- o que deve fazer
- porque deve fazê-lo

Precisamos então:

- identificar o que queremos (descrição)
- identificar porque queremos isto (objetivos)
- identificar as condições de aceitação do resultado. Estas são dadas por:
  - *requisitos*: condições, atributos, propriedades ou características necessárias para que a implementação do elemento seja aceitável. Em última análise, requisitos definem critérios de aceitação que precisam ser satisfeitos pela construção do elemento sendo especificado.
  - *hipóteses*: condições, atributos, propriedades ou características necessárias assumidas como dadas antes de se construir o elemento sendo especificado.

- *restrições*: condições que somente restringem a liberdade de escolha de alternativas de construção do elemento sendo especificado.

Especificações definem o que um determinado componente deve realizar. Elas devem ser redigidas em uma linguagem precisa e não ambígua para que diferentes leitores extraiam exatamente a mesma informação que a que o redator queria transmitir.

Na apresentação informal, especificações aparecem como de texto redigido em linguagem natural, por exemplo em português. Esta forma é excessivamente sujeita a interpretações errôneas. Em uma apresentação um pouco mais formal, utilizamos uma coletânea de fragmentos de texto composta por **descrição**, **requisitos**, **hipóteses** e **restrições**.

A descrição é informal e destina-se a dar uma visão geral da intenção, ou objetivo, do componente sendo especificado. Requisitos, hipóteses e restrições são redigidos em linguagem natural restrita – português ou inglês estruturado –, que procura minimizar os problemas de ambigüidade.

Exemplos de requisitos bem formulados:

- Diariamente, antes de iniciar o expediente será gerado um arquivo contendo:
- todos os ítens de estoque em falta.
- todos os ítens de estoque com quantidade abaixo do nível mínimo
- todos os ítens de estoque cujas vendas estão aceleradas com relação ao esperado
- todos os ítens de estoque encomendados e cujo fornecimento está atrasado
- A qualidade dos dados transcritos será função exclusiva dos valores destes dados, independendo do processo de transcrição e do número de correções destes dados

Exemplos de requisitos mal formulados

- Deverá ser assegurado bom tempo de resposta. — o que é bom tempo de resposta? Como medir?
- Na medida do possível o sistema deverá proteger-se contra erros de operação. — Como é que se sabe se foi ou não possível?
- Os procedimentos devem ser adequados ao usuário. — O que se entende por adequação ao usuário?

Exemplos de hipóteses bem formuladas

- O crescimento da demanda de serviço prestado pelo sistema ao entrar em funcionamento será:
- 5% no decorrer dos 2 primeiros meses
- 20% no decorrer dos 6 meses a seguir
- 3% daí em diante
- Caso uma mensagem seja recebida, o sistema de transmissão garante que ela está correta. — Quando, ao utilizar um componente, se verificar que uma ou mais das hipóteses definidas não valem, é necessário reformular o componente.

Exemplos de hipóteses mal formuladas

- O crescimento da demanda será grande. — O que é grande?
- Existirão suficientes terminais para utilizar o sistema. — O que é suficiente?

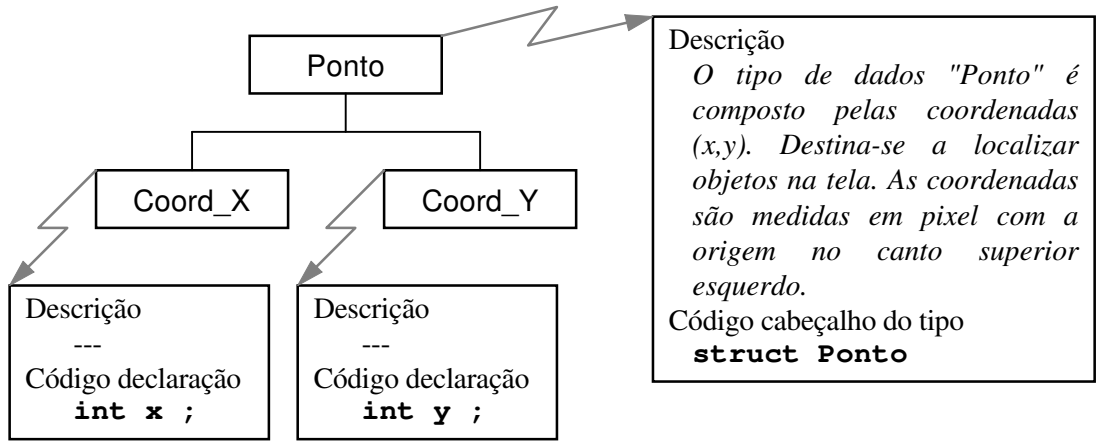


Figura 4-1 Exemplo de omissão de descrição.

Exemplos de restrições bem formuladas

- O sistema deverá operar na plataforma XYZ-123.
- O sistema será desenvolvido em ANSI-C obedecendo às normas NP-01 a NP-10

Exemplos de restrições mal formuladas

- o sistema deverá ser seguro. — Não é restrição, é requisito mal formulado, o que se entende por seguro?
- o sistema será desenvolvido em C. — Que C? A linguagem definida por algum compilador? Ou ANSI irrestrito? Ou ANSI-C restrito por normas internas?

Quando devem ser utilizados os elementos de especificação apresentados? A descrição deve ser incluída sempre. A única exceção é para *objetos simples* em estruturas de decomposição. Entenda-se por um objeto simples:

1. o objeto não é a raiz da estrutura de decomposição
2. a descrição do objeto é óbvia. Por exemplo:

   considere o tipo de dados *Ponto* composto pelos elementos *coordenada_x* e *coordenada_y*. Este tipo deve ser descrito por três objetos: a raiz, que contém a descrição e a declaração do tipo de dados e as duas coordenadas. A descrição deve ser incluída apenas no objeto raiz. Veja a figura 4-1.

Requisitos, restrições e hipóteses seguem regra semelhante. De modo genérico pode-se considerar que, sempre que a descrição não for suficiente para caracterizar de forma completa o objeto, devem ser incluídos requisitos, restrições e hipóteses para completar a especificação.

Qualquer objeto do sistema sofre evoluções no tempo. Estas evoluções devem ser registradas em um *histórico de alterações*. Isto permite avaliar, por exemplo, se um elemento está sendo alterado com muita freqüência, indicando uma possível falha de projeto. Um item no histórico de alterações é composto pela data da alteração e por um comentário descrevendo a alteração. Não se deve registrar toda e qualquer alteração sem nenhum critério. Isto teria um custo muito alto e prejudicaria a legibilidade do histórico. Alterações devem ser registradas sempre que afetem:

- a funcionalidade do componente sendo alterado.
- as restrições, requisitos ou hipóteses do componente sendo alterado.

- de modo significativo a implementação do componente sendo alterado (por exemplo, ao utilizar um algoritmo diferente).

4.2 Descrição da especificação das classe de objetos utilizadas

Além dos elementos genéricos descritos anteriormente, cada classe de objetos possui elementos próprios que são descritos a seguir.

4.2.1 Módulos

A principal finalidade de um módulo é a de agregar diversos elementos (classes, funções, tipos de dados, constantes, etc.) segundo normas bem definidas. Na linguagem C++, módulos tem um papel menos relevante do que em C. Em C++, o papel de organizar os diversos elementos é em grande parte transferida para as classes.

As características a seguir **não** devem ser descritas nos componentes do módulo, mas sim no próprio módulo.

- Descrição. Um módulo bem projetado implementa completamente uma abstração. Esta abstração deve ser completa e corretamente descrita.
- Seqüência de ativação de funções. Deve ser definida sempre que um conjunto de funções (provavelmente uma API) possua uma seqüência de ativação, ou protocolo de utilização bem definido. Por exemplo:
  - `API_ObterContextoDispositivo(...)`
  - `API_DesenharObjeto(...)`
  - `API_LiberarContexto(...)`

  O protocolo que define como estas funções devem ser utilizadas não deve ser repetido na especificação de cada função. Sem este tipo de especificação, torna-se impossível fazer uso correto destas funções.
- Modo de utilização de classes. Um módulo pode conter um conjunto de classes que formam um *Framework* ou parte de um *Framework*. A biblioteca de classes MFC da Microsoft, por exemplo, é composta por mais de 150 arquivos de implementação, aproximadamente 20 arquivos de inclusão e 90 classes. O comportamento deste conjunto de classes não pode ser descrito de modo separado em cada classe. É claro que no caso da MFC, ou qualquer biblioteca desse porte, a descrição completa das classes também não se encontra nos módulos de definição, mas sim em um manual de referência.
- Descrição das estruturas de dados. Módulos de definição devem descrever as estruturas de dados de um modo abrangente, sem se preocupar com sua implementação. Por exemplo: uma estrutura tipo pilha será descrita pelas suas propriedades (*first in last out*), ocultando se a sua implementação é realizada por uma lista ou por um vetor. Já em módulos de implementação, as estruturas devem ser descritas do ponto de vista da sua implementação específica.

4.2.2 Classes

Assim como módulos, classes agregam elementos de modo a implementar uma abstração bem definida. Fazem parte da interface de uma classe os seus métodos, tipos e membros. Classes possuem propriedades adicionais se comparadas a módulos. Elas podem ser instanciadas (na forma de objetos) e possuem estrutura de herança. Seus componentes também possuem maior detalhe. Por exemplo: membros e métodos possuem escopo (privados, públicos ou protegidos), podem ser estáticos ou não, e métodos podem ser virtuais ou não.

Estes fatores tendem a transformar bibliotecas de classes em uma estrutura muito difícil de compreender e manter. Tanto é assim que a maioria dos "fabricantes" de *frameworks* baseados em classes fornece, além de manuais de referência, o código fonte completo da biblioteca. Este fato alerta para o cuidado que se deve ter ao especificar classes.

Do mesmo modo que módulos, classes devem descrever o modo como seus componentes se relacionam. Classes podem ser exportadas ou não. Caso sejam exportadas, sua definição residirá no módulo de definição e sua implementação no módulo de implementação. Caso contrário, residirá totalmente no módulo de implementação.

Os seguintes tópicos devem constar na especificação de uma classe:

- Objetivo da classe. Uma classe bem projetada implementa completamente uma abstração. Esta abstração deve estar completa e corretamente descrita na sua especificação.
- Modo de utilização da classe. Deve indicar, por exemplo, se a classe foi projetada para instanciação direta de objetos, ou se o cliente deve inicialmente derivar uma nova classe. No caso de classes cuja utilização pressupõe uma derivação, deve estar claramente definido quais métodos (se houver) devem ser redefinidos e de que modo. Métodos virtuais merecem especial atenção neste contexto.
- Seqüência de ativação dos métodos. Muitas vezes temos um conjunto de métodos cuja seqüência de ativação deva obedecer padrões (protocolos) bem definidos. Por exemplo:
  - `MetaFile->Abrir(...)`
  - `MetaFile->GravarPrimitiva(...)`
  - `MetaFile->Fechar(...)`

  Deve-se indicar o modo como estas funções interagem entre si e com estruturas de dados encapsuladas na classe ou de outras classes.
- Descrição das estruturas de dados. Muitas classes implementam métodos que interagem através de estruturas de dados encapsuladas. Quando estas estruturas possuírem alguma complexidade é importante que as suas propriedades sejam externadas. No módulo de definição são definidas as propriedades que o cliente da classe necessita conhecer. No módulo de implementação são registradas as propriedades específicas da implementação

4.2.3 Funções ou Métodos

Funções de módulos ou métodos de classes, possuem interfaces muito semelhantes. Os ítens a seguir, combinados com os ítens genéricos já apresentados (descrição, requisitos, hipóteses e restrições), descrevem esta interface. As diferenças entre a interface de uma função e de um método também estão indicadas. Funções e métodos requerem grande atenção e detalhe ao serem especificados, já que são eles que alteram estruturas de dados e arquivos e respondem à eventos do ambiente e do usuário.

- Sobrecarga (Overloading). Utilizado apenas em métodos. Deve indicar as diferenças entre este método e outros com o mesmo nome. A utilização de sobrecarga de nomes permite solucionar, de modo muito elegante do ponto de vista sintático, questões de nomenclatura. Entretanto, se não houver uma documentação apropriada, esta duplicidade de nomes pode levar a erros difíceis de serem encontrados, principalmente ao combinar sobrecarga e reescrita de operadores.
- Reescrita. Utilizado apenas em métodos. Deve indicar as classes base que contenham métodos redefinidos pelo presente método. Este poderoso mecanismo, especialmente quando utilizado em funções virtuais deve ser cuidadosamente documentado. Isto é reforçado pelo fato de ser sintáticamente correto redefinir uma função declarada como virtual, omitindo a palavra-chave *virtual*.

Para especificar o que uma função ou método faz, devemos definir tudo o que ela necessita e/ou modifica. Esta modificação é dada pela diferença entre vários ítens antes e depois da função ter executado. Temos então:

- Especificação dos parâmetros, dados e estados de entrada
  - ◊ *Parâmetros recebidos*. Discrimina e explica todos os parâmetros recebidos. Deve especificar o nível de qualidade esperado para cada parâmetro. Isto permite verificar se a função está criticando de modo adequado os parâmetros recebidos.
  - ◊ *Variáveis globais próprias ao entrar*. Discrimina, explica e define o conteúdo de:
    - ∗ cada variável global pública do módulo que será consultada ou modificada.
    - ∗ cada variável pública ou restrita da classe que contém o método que será consultada ou modificada.
  - ◊ *Variáveis globais de outros ao entrar*. Discrimina, explica e define o conteúdo de:
    - ∗ cada variável global pública de outros módulos que será consultada ou modificada.
    - ∗ cada variável pública de outras classes que será consultada ou modificada.
    - ∗ cada variável restrita ou privada de outras classes que será consultada ou modificada e cujo escopo foi concedido pelo comando ***friend***.
  - ◊ *Arquivos ao entrar*. Discrimina os arquivos que serão consultados ou alterados.
  - ◊ *Estados ao entrar*. Discrimina os estados de equipamento e das estruturas de dados antes de ativar a função.
  - ◊ *Variáveis globais encapsuladas ao entrar*. Discrimina, explica e define o conteúdo de:
    - ∗ cada variável global encapsulada (definida como ***static***) do módulo que será consultada ou modificada.
    - ∗ cada variável privada da classe que será consultada ou modificada.
- Especificação dos parâmetros, dados e estados de saída
  - ◊ *Valor retornado*. Descreve a natureza do valor retornado. Quando existirem condições de retorno, isto é, valores pré-estabelecidos indicando o resultado da execução da função, estas deverão ser discriminadas.
  - ◊ *Parâmetros alterados*. Discrimina os parâmetros cujo valor é (potencialmente) alterado pela função, descrevendo qual será a possível alteração.
  - ◊ *Variáveis globais do módulo ao sair*. Descreve as alterações que serão (potencialmente) feitas em:
    - ∗ cada variável global pública do módulo.
    - ∗ cada variável pública ou restrita da classe que contém o método.
  - ◊ *Variáveis globais de outros módulo ao sair*. Descreve as alterações que serão (potencialmente) feitas em:
    - ∗ cada variável global pública de outros módulos.
    - ∗ cada variável pública de outras classes.
    - ∗ cada variável restrita ou privada de outras classes cujo escopo foi concedido pelo comando ***friend***.
  - ◊ *Arquivos de saída*. Discrimina os arquivos que podem ter sido alterados pela função e descreve as possíveis alterações feitas.
  - ◊ *Estados ao sair*. Discrimina os estados de equipamento e das estruturas de dados após o termino da função.
  - ◊ *Variáveis globais encapsuladas ao sair*. Descreve as alterações que serão (potencialmente) feitas em:

    - cada variável global encapsulada (definida como ***static***) no módulo.
    - cada variável privada da classe.

- Especificação da interface com o usuário

  O "ponto" no qual diálogos são ativados e comandos, eventos ou mensagens são tratados ou disparados é sempre em alguma função ou método. Deve ser possível então examinar a especificação de uma função e determinar:
    - *Diálogos ativados*. Para cada diálogo ativado deve descrever o diálogo, informar o seu nome, recurso associado ou arquivo onde está definido.
    - *Mensagens disparadas*. Qualquer mensagem diretamente disparada (*PostMessage* ou *SendMessage*) pela função deve ser documentada.
    - *Eventos, comandos ou mensagens tratados*. Quando uma função ou método estiver associado a um evento, comando ou mensagem, isto deve ser explicitamente informado. Também deve ser indicado se o *binding* entre o evento e a função é estático (em tempo de compilação) ou dinâmico (em tempo de execução).
    - *Mensagens exibidas*. Discrimina as mensagens exibidas, indicando a causa para a exibição. Entende-se por mensagens neste contexto, diálogos simples com a finalidade de advertir ou informar o usuário sobre diversas situações ou ainda consulta-lo com opções simples (do tipo: [OK, CANCELAR, CONTINUAR] ou [SIM, NÃO]).

- Definição de assertivas

  Assertivas determinam as condições que os dados devem satisfazer ao entrar em uma função ou fragmento de código — *assertivas de entrada* —, bem como as condições a serem satisfeitas ao sair desta função ou fragmento — *assertivas de saída*. Em outras palavras, as assertivas de entrada definem as pre-condições necessárias para que o código possa operar corretamente – *hipóteses* – e as assertivas de saída definem as pós-condições que deverão ser verdadeiras para que se possa dizer que o programa foi implementado corretamente – *requisitos*. Assertivas devem ser utilizadas sempre que a complexidade de uma função ou fragmento de código impeça que este seja compreendido por simples inspeção.

4.2.4 Tipos

Tipos de dados, principalmente e C mas também em C++, são a forma mais comum de declarar estruturas de dados complexas. Além dos elementos de especificação genéricos, é necessário especificar também as *assertivas estruturais* do tipo. Assertivas estruturais definem a correção da organização de uma estrutura de dados. Ao entrar em uma função de manipulação da estrutura, as assertivas estruturais desta estrutura devem estar válidas, pois, de outra forma, o resultado da função ficará comprometido. Ao sair é evidente que as assertivas devem valer, pois senão a função não teria operado corretamente. É claro que durante o processamento da função as estruturas de dados podem passar por estados em que as assertivas estruturais não valem. Por exemplo, ao inserir um elemento em uma lista duplamente encadeada, vai ocorrer um momento em que a lista estará parcialmente encadeada.

4.2.5 Atributos

Como já foi explicado no capítulo anterior, atributos representam constantes, variáveis de tipos fundamentais ou cujo tipo já foi declarado, vetores, ponteiros e referências. Além dos elementos de especificação genéricos, devem ser especificados:

- medida do atributo. Por exemplo:
    - velocidade do escoamento em metros por segundo,

- nome completo na forma [<sobre nome>, <nome>].

- critérios de validade do atributo. Deve descrever os valores válidos (domínio) do atributo ou então definir critérios para controlar a validade do atributo.

4.2.6 Diagramas de estados e eventos

Como foi visto no capítulo anterior, diagramas de estados e eventos estão sempre relacionados a classes. A classe de objeto "diagrama de estados e eventos" não possui qualquer especificação. A classe associada ao diagrama e os objetos que pertencem ao diagrama é que devem ser especificados. As seguintes classes de objetos devem ser especificadas:

- Eventos. Esta classe de objeto representa métodos da classe associada ao diagrama, por isto possui o mesmo tipo de especificação que métodos (veja a seção 4.2.3).
- Estados. Alem dos ítens de especificação genéricos, estados devem definir com precisão as invariantes que representam. Estas devem ser especificadas sob a forma de assertivas. Tem-se então:
  - Assertivas invariantes das variáveis globais do módulo
  - Assertivas invariantes das variáveis globais de outros módulos
  - Assertivas invariantes de arquivos
  - Assertivas invariantes das variáveis globais encapsuladas

  É muito comum que, ao entrar em um estado, seja fornecido algum tipo de *feedback* ao usuário. Este *feedback* pode representar desde a alteração da forma do cursor até uma completa mudança na forma como um documento está sendo apresentado. Qualquer *feedback* desencadeado sempre que se entre ou saia de um estado deve ser documentado.
- Processos. Idealmente devem possuir apenas o código para uma condição de seleção, quando eventos possuírem mais de um estado destino. Podem, entretanto, possuir fragmentos de código ou chamar funções. Devem possuir portanto descrição, requisitos, hipóteses e restrições.

5. Ferramentas Utilizadas

Não é possível dissociar o processo de desenvolvimento apresentado das ferramentas requeridas para criar e manter os produtos resultantes de cada etapa. Se por um lado o processo foi moldado pela existência das ferramentas, pelo outro as ferramentas foram configuradas de modo a adapta-las ao processo. Teoricamente seria factível definir os produtos intermediários de cada etapa sem mencionar como gera-los. Na prática, entretanto, seria extremamente difícil e caro gera-los ou manter sua consistência, inviabilizando assim o processo.

5.1 TALISMAN

TALISMAN é um *Ambiente de Engenharia de Software Assistido por Computador*. Em outra palavras, é uma coletânea *integrada* de ferramentas mecanizadas, utilizadas para desenvolver, controlar a qualidade e manter planos, especificações, projetos, código e documentação de sistemas de programação quaisquer. Em adição, TALISMAN é adaptável, permitindo que os usuários (analistas, desenvolvedores, documentadores, gerentes de desenvolvimento, inspetores de qualidade, etc.) adaptem o ambiente às condições particulares da empresa, do sistema sendo desenvolvido, do estágio no processo de desenvolvimento e, finalmente, às necessidade particulares do usuário.

Para adaptar o ambiente TALISMAN ao processo de desenvolvimento apresentado, foram escritas aproximadamente 16.000 linhas de código, divididas em nove programas, na linguagem interna do ambiente. Estes programas foram desenvolvidos de modo evolutivo ao longo de 5 anos. Podemos particionar estes programas em:

formulários de especificação — Definem a extração (*view*) do repositório que será utilizada para especificar cada classe de objetos.

linearizadores — Do mesmo modo que formulários de especificação, definem uma extração do repositório. O objetivo da extração, neste caso, não é o de especificar cada objeto, mas sim de gerar código compilável. O resultado da linearização pode ser dirigido tanto para um arquivo como para a tela. Quando dirigido para a tela, é possível editar o código de modo semelhante a um editor de texto.

validadores — Destinam-se a aplicar *regras de validação* a um objeto ou conjunto de objetos. Geram *laudos de validação* que devem ser examinado a fim de detectar erros ou falhas nas especificações.

exportadores — Extraem fragmentos selecionados de informação contida no repositório em um formato que permita ser *importado* para outro repositório. É utilizado, por exemplo, para implementar um esquema de trabalho em equipe.

5.2 Visual C++ e MFC

A classe de sistemas que podem ser desenvolvidos utilizando o processo de desenvolvimento apresentado está restrita apenas pelos ambientes e plataformas que suportam a API de Windows e da biblioteca MFC. Os ambientes atualmente cobertos diretamente pela Microsoft são: Windows 3.x, Windows 95, Windows NT e Mac. Outros fabricantes (por exemplo Bristol Wind/U) permitem o desenvolvimento para plataformas UNIX. Conclui-se que isto não constitui uma restrição muito

importante, já que a combinação “MFC + API Windows” está se tornando um padrão “de fato” na industria.

O produto *Visual C++* é formado um conjunto de ferramentas integradas que visam o desenvolvimento de aplicativos de modo “rápido”. Uma série de “assistentes” automatiza tarefas como:

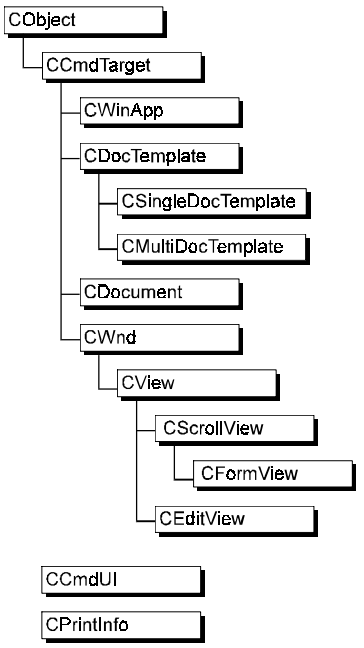


Figura 5-1 Fragmento da hierarquia de classes da biblioteca MFC.

- gerar o “esqueleto” inicial de programas utilizando a biblioteca MFC (AppWizard)
- criar classes derivadas de MFC (ClassWizard)
- associar código (métodos de classes) à mensagens do ambiente Windows (ClassWizard)
- associar variáveis (membros de classes) à controles em diálogos (ClassWizard)

O produto possui um ambiente integrado no qual é possível:

- administrar os arquivos que compõe o programa (incluir, excluir, definir parâmetros de compilação, etc.).
- editar o código fonte dos arquivos que compõe o programa utilizando um editor dirigido por sintaxe.
- * editar recursos Windows (menus, bitmaps, diálogos, barras de botões, tabelas de strings, aceleradores de teclado, etc.).
- utilizar os assistentes disponíveis.
- criar e consultar bases de dados com informações sobre a hierarquia de classes, chamada de funções, associação entre funções e mensagens Windows, etc.
- * compilar arquivos C, C++, auxílio e recursos, tanto de modo individual como de acordo com a data de alteração (*make*).
- * depurar programas.

Somente as ferramentas marcadas com * são utilizadas no processo de desenvolvimento apresentado. O compilador e o “montador” de programas (*make*) são utilizado fora do ambiente integrado.

A biblioteca de classes MFC (incluindo o código fonte), bem como a documentação completa (em CD-ROM) são parte integrante do produto.

O elemento mais relevante do produto é sem dúvida a biblioteca de classes MFC. A utilização da biblioteca MFC pode simplificar muito o desenvolvimento de programas para o ambiente Windows. Principalmente para aqueles que fazem uso de aspectos complexos da API, como OLE (Object Linking and Embedding), DDE (Dynamic Data Exchange), MDI (Multiple Document Interface) ou aplicativos para acesso a bancos de dados.

Os recursos de instrumentação incluídos na biblioteca constituem instrumentos poderosos para identificar falhas usualmente difíceis de serem localizadas. Estão disponíveis recursos para:

- detectar “vazamentos” ou corrupção de memória nos seguintes instantes:
  - ◊ a cada operação que manipule memória.
  - ◊ ao final do programa.
  - ◊ definido pelo programador.
- identificar erros de utilização das classes. Por exemplo, ao criar um objeto da classe **CBrush** será emitida uma mensagem de erro (indicando o erro, a causa e a localização) caso seja utilizado antes de ser associado à um recurso Windows. Isto é realizado por meio de assertivas inseridas do código.

Tem-se na realidade duas versões da biblioteca: a versão sem instrumentação (denominada de produção) e a versão instrumentada. O conceito de versão instrumentada nada tem a ver com a inclusão de informações para o depurador. De fato, apenas alguns arquivos da versão instrumentada possuem informação para o depurador.

Adicionalmente, o produto fornece versões instrumentadas de algumas bibliotecas (*DLL's*) do próprio ambiente Windows. A utilização destas bibliotecas penaliza o desempenho de todo o sistema em até 30%, em compensação, as funções contidas em estas bibliotecas efetuam uma rigorosa validação dos parâmetros recebidos, passando o controle ao depurador ou interrompendo o programa antes que todo o sistema se torne instável.

5.3 Ferramentas auxiliares

Algumas ferramentas foram desenvolvidas (em C e C++) especialmente para automatizar certas tarefas repetitivas e geradoras (em potencial) de erros difíceis de detectar.

5.3.1 Gerador de *scripts* para o programa *make*

Uma tarefa aparentemente simples, mas sujeita a falhas com conseqüências graves, é montagem de um programa para gerar o código executável. Mesmo com a evolução dos microcomputadores, não é viável recompilar todos os arquivos que formam um programa cada vez que alguns poucos destes sejam alterados. Para tentar resolver este problema, a maioria dos fabricantes de compiladores fornece um programa conhecido como *make*. Este programa recebe um roteiro (*script*) que define as relações de dependência entre os arquivos do projeto. Ao montar o programa, apenas os arquivos alterados desde a última montagem, e aqueles que dependem destes arquivos, são processados. O problema é que nenhum destes fabricantes fornece uma ferramenta para gerar o roteiro de entrada com as relações de dependência.

Foi desenvolvida uma ferramenta que, a partir de uma lista de arquivos (e alguns outros parâmetros), gera um roteiro para o programa *make*. Os arquivos da lista devem conter programas escritos em C ou C++. A ferramenta inspeciona o código de cada programa na lista, procurando arquivos dos quais ele dependa. Cada um destes arquivos é examinado de modo recursivo gerado, deste modo, uma lista de dependência. A partir da lista de dependência de cada arquivo é gerado o roteiro.

5.3.2 Geradores de tabelas de definição

Programas que interagem com o usuário, principalmente aqueles que o fazem por meio de interfaces gráficas, tendem a utilizar uma grande quantidade de seqüências de caracteres (*strings*) constantes. Estas seqüências são utilizadas para várias finalidades. Por exemplo:

- mensagens de erro ou advertência
- textos em barras de estado
- *tooltips* em barras de ferramentas
- textos de diálogos montados dinamicamente

Agregar todas estas seqüências em um único local oferece algumas vantagens. Por exemplo:

- visão de conjunto
- possibilidade de realizar edições em bloco (por exemplo, para traduzir as seqüências para outro idioma).

Para isto criam-se arquivos contendo tabelas de definição de seqüências. Estes arquivos podem ser incluídos (utilizando a diretiva **#include**) nos módulos que necessitem utilizar as seqüências.

Por outro lado, o fato da mensagem residir em um ponto diferente do que é utilizado, dificulta a criação, manutenção e legibilidade do código.

A utilização de TALISMAN oferece uma solução sem comprometimento. O formulário de especificação de cada bloco de código inclui um campo no qual é possível definir seqüências de caracteres. Isto permite definir a seqüência em local próximo ao qual será utilizada. Um linearizador permite extrair estas seqüências e direciona-las para a tela ou para um arquivo. Isto fornece uma visão de conjunto e a possibilidade de realizar edições em bloco.

5.4 A integração das ferramentas

Antes de poder utilizar as ferramentas apresentadas de modo integrado no processo de desenvolvimento, foi necessário configura-las e adapta-las. Para utilizar a biblioteca MFC, por exemplo, houve que alterar substancialmente os programas de formulário de TALISMAN. A cada mudança de versão no ambiente Visual C++ são necessários ajustes nas diversas ferramentas. Existe uma tendência no mercado (provavelmente por razões de concorrência comercial) em tornar alguns produtos cada vez mais monolíticos. Isto dificulta a sua integração em qualquer processo de desenvolvimento.

5.5 Desenvolvimento em equipe

O processo de desenvolvimento apresentado não faz nenhuma referência explícita a trabalho cooperativo. Como TALISMAN não suporta controle de versões ou configurações, devem ser utilizados mecanismos de controle externos. Equipes pequenas podem atribuir a cada membro o controle sobre um conjunto de repositórios fixo. Quando um indivíduo necessita alterar um objeto contido em um repositório pelo qual não seja responsável, é gerada uma cópia do mesmo e as alterações são realizadas nesta cópia. Para equipes maiores é recomendável a utilização de um modelo de *check-in* e *check-out* com cópia de arquivos. Em qualquer caso, deve-se armazenar apenas um módulo em cada repositório, a fim de minimizar a "granularidade" da informação.

Para resolver conflitos, devem ser utilizadas ferramentas convencionais para comparação de arquivos aplicadas ao código fonte (não aos repositórios). Quando as diferenças forem de natureza incremental, é possível utilizar o mecanismo de importação e exportação disponível no ambiente TALISMAN para realizar a atualização nos repositórios. Caso contrário, recomenda-se atualizar os repositórios manualmente.

6. Avaliação, Conclusões e Perspectivas

6.1 Avaliação

A grande capacidade de adaptação de TALISMAN tem sido um fator decisivo na construção e evolução do processo de desenvolvimento apresentado. A linguagem de programação interna, permite criar e alterar com grande facilidade formulários de especificação, linearizadores e validadores. Como TALISMAN é um "quase" meta-ambiente, é relativamente simples manter as linguagens de representação, por exemplo, adicionando classes de objeto ou relações.

A geração automática de código a partir de linguagens de representação com elevado grau de abstração, em particular as que requerem a manipulação de representações gráficas (por exemplo, a representação estados e eventos), evidenciou algumas limitações do ambiente TALISMAN, como por exemplo, a incapacidade de manter duas janelas abertas, lado a lado, uma contendo um diagrama e a outra especificações referentes a este diagrama.

Outra deficiência observada, refere-se à ausência de mecanismos de controle de versão e configuração e à impossibilidade de acessar os repositórios por mais de um usuário simultaneamente. Isto dificulta a aplicação do processo em um ambiente de trabalho cooperativo.

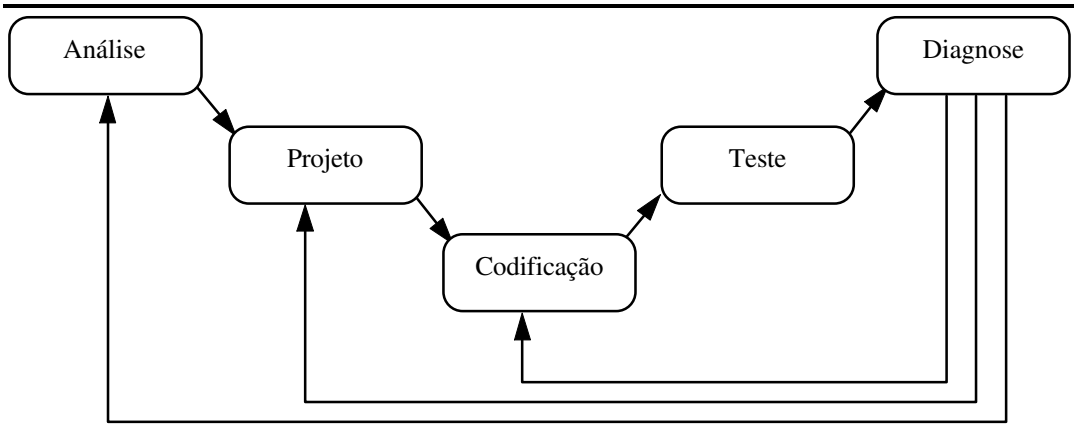


Figura 6-1 Ciclo convencional de teste-diagnose.

O objetivo principal do processo – minimizar o tempo de propagação de alterações entre as diversas etapas do desenvolvimento – foi alcançado. Os métodos de trabalho convencionais, apesar de dedicarem uma grande parte do esforço total na análise e projeto do sistema, não impedem o surgimento de "ciclos de programação por atrito", na fase final de desenvolvimento. É comum que defeitos no software sejam corrigidos de forma incompleta ou incorreta, introduzindo novos defeitos, provavelmente mais difíceis de serem encontrados que os originais. Isto se deve, em muitos casos, à não observação das especificações iniciais. Esta "não observação" está relacionada com o fato de especificações e código residirem em documentos diferentes. A utilização de uma ferramenta como TALISMAN, que permite a inclusão de código junto as especificações do sistema, obriga o desenvolvedor a rever estas especificações sempre que consultar o código. Isto não significa que o ciclo descrito na figura 6-1 deixa de existir, mas sim que a ele se transforma como descrito na figura 6-2.

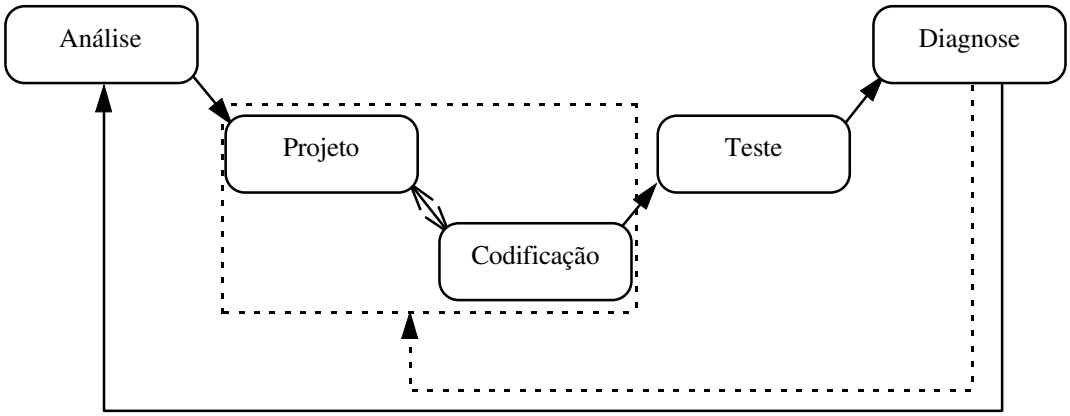


Figura 6-2 Ciclo de teste-diagnose utilizando o processo de desenvolvimento.

6.2 Conclusões

Mesmo não tendo realizado medições de forma sistemática, concluímos que o processo de desenvolvimento apresentado (e outras versões mais antigas) tem sido utilizado com grande sucesso no desenvolvimento dos ambientes TALISMAN e TOTEM (em fase inicial). Isto nos leva a crer, que os objetivos propostos pelo processo estão, pelo menos em parte, sendo alcançados.

A tarefa de criar, utilizar e evoluir um processo de desenvolvimento de software é extremamente onerosa. Acreditamos, entretanto, que a ausência de um processo de desenvolvimento bem definido não só eleva muito os custos de desenvolvimento, como também diminui substancialmente a qualidade dos sistemas produzidos.

6.3 Perspectivas

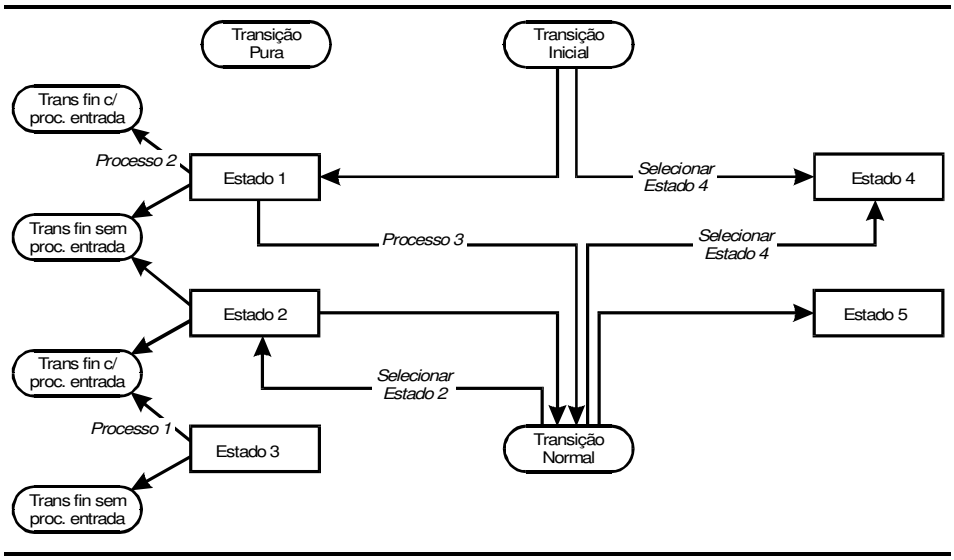
A experiência adquirida com o desenvolvimento e utilização do processo apresentado serve como motivação para continuar seu aprimoramento.

1. Acreditamos ter alcançado os limites de utilização do ambiente TALISMAN. As condições para o desenvolvimento de um novo ambiente estão dadas. Podemos adiantar algumas das características deste novo ambiente:
   - possui mecanismos de controle de versão e configuração.
   - permite o acesso simultâneo por vários usuários ao repositório.
   - permite relacionamentos entre objetos contidos em repositórios distintos.
   - permite criar e manter representações (meta ambiente) de modo controlado.
   - possui mecanismos para gerenciar o processo de desenvolvimento.
   - permitirá a especificação e a geração ou composição de código a partir de representação com maior grau de abstração.

2. A utilização do processo em outros contextos permitirá uma avaliação mais precisa.
3. A introdução de métricas permitirá avaliar de modo rigoroso a eficiência e efetividade do processo.

Apêndice: Esquema de geração de código para diagramas de estados e eventos

O objetivo deste apêndice é exemplificar a composição de código a partir de diagramas de estados e eventos. A figura a seguir ilustra um diagrama de estados e eventos genérico.



O texto que inicia na página 43 representa o esquema de código gerado a partir de especificações textuais (criadas por meio de formulários apropriados não ilustrados neste apêndice) e do diagrama acima.

Com a finalidade de diferenciar o código gerado pela ferramenta, dos fragmentos de código inseridos pelo desenvolvedor, representamos estes últimos como mostra a figura abaixo

```
- Classe 1 -
!CODIGO!
Código: cabeçalho da classe
```

Em um programa "real", este fragmento seria substituido pela declaração de uma classe. Por exemplo:

```
DI_CMinhaJan::CWnd
```

A figura abaixo mostra a diferença entre o código definido pelo desenvolvedor (em negito e itálico), e o código gerado automáticamente pela ferramenta.

```
if ( Estado != Novo_Estado )
             {
                switch ( Novo_Estado )
                {

                   case ESTADO_4 :
                      Estado 4
                      Código fonte para estabelecer
                      a invariante do estado
                      break ;

                   case ESTADO_1 :
                      Estado 1
                      Código fonte para estabelecer
                      a invariante do estado
                      break ;
                } // switch
             } // if
```

A geração de código por meio da combinação de especificações textuais e diagramáticas, permite que manutenções complexas sejam realizadas pela simples alteração de algumas ligações em um diagrama.

```
////////////////////////////////////////////////////////////////////////
//
// Módulo de definição: Modulo 1
//
// Arquivo gerado:      exmaq.hpp
//
// Letras de identificação: xx
// Numero de identificação: 100
//
// Nome lógico da base:     Exemplos de diagrams de transição
// Nome do arquivo base:    D:\PROJETOS\CPP\FORM\ESQUEMA.BSW
//
// Projeto que mantém a base
//
// Direitos autorais da base
//
// Estado corrente de distribuição da base
//    Versão: 1    Modificação: 0    Alteração: 0    Data liberação:
//
//    Autorizou liberação
//
// Histórico do projeto
//    Vers Modf Altr  Autor    Data     Observações
//
// Alterações de emergência realizadas e ainda não passadas a limpo
// Alterações e correções realizadas
//
////////////////////////////////////////////////////////////////////////

// Controle de inclusão múltipla

#if ! defined( _exmaq_ )
#define _exmaq_

// Controle de escopo do arquivo de definição

#if defined( _exmaq_OWN )
   #define _exmaq_CLASS
   #define _exmaq_TEST 1
#else
   #define _exmaq_CLASS extern
   #define _exmaq_TEST 0
#endif

//--- Classes exportadas pelo módulo ---

////////////////////////////////////////////////////////////////////////
//
// Classe: Classe 1
//
// Descrição da classe
//    - Classe 1 -
//    Esta classe ilustra as formas básicas de instâncias de transição.
//
////////////////////////////////////////////////////////////////////////

   - Classe 1 -
   !CODIGO!
   Código: cabeçalho da classe
   {

   public:

      // Transição: Transição final, com um processo na entrada
      //    - Transição final, com um processo na entrada -
      //    Esta transição é executada somente se estiver em um dos estados
      //    de entrada.
```

```
      //    No diagrama figuram processos em algumas das arestas de entrada.
      //    Como não tem saída, não altera o estado corrente.

         - Transição final, com um processo na entrada -
         !CODIGO!
         Declaração: protótipo da função que implementa a transição ;

      // Transição: Transição final, sem processos na transição
      //    - Transição final, sem processos na transição -
      //    Esta transição é executada somente se estiver em um dos estados
      //    de entrada.
      //    No diagrama nao figuram processos nas arestas de entrada.
      //    Como não tem saída, não altera o estado corrente

         - Transição final, sem processos na transição -
         !CODIGO!
         Declaração: protótipo da função que implementa a transição ;

      // Transição: Transição inicial
      //    - Transição inicial -
      //    Esta transição é incondicional
      //    Dependendo de condições definidas na aresta de saída, seleciona
      //       o estado destino

         - Transição inicial -
         !CODIGO!
         Declaração: protótipo da função que implementa a transição ;

      // Transição: Transição normal
      //    - Transição normal -
      //    Esta transição depende do estado ao iniciar
      //    Define o novo estado

         - Transição normal -
         !CODIGO!
         Declaração: protótipo da função que implementa a transição ;

      // Transição: Transicao pura
      //    - Transicao pura -
      //    Esta transição executa incondicionalmente.
      //    Como não tem saída, não altera o estado corrente

         - Transicao pura -
         !CODIGO!
         Declaração: protótipo da função que implementa a transição ;

      //--- Declaração do mapa de mensagens
         DECLARE_MESSAGE_MAP()

      //--- Declaração para diagrama de transições

   private:

         int Estado ;

   } ; // Fim da declaração da classe: Classe 1

#undef _exmaq_CLASS

#endif

////// Fim da definição: Modulo 1 ////
```

```
////////////////////////////////////////////////////////////////////////
//
// Módulo de implementação: Modulo 1
//
// Arquivo gerado:          exmaq.cpp
//
// Letras de identificação: xx
// Numero de identificação: 100
//
// Nome lógico da base:     Exemplos de diagrams de transição
// Nome do arquivo base:    D:\PROJETOS\CPP\FORM\ESQUEMA.BSW
//
// Projeto que mantém a base
//
// Direitos autorais da base
//
// Estado corrente de distribuição da base
//    Versão: 1    Modificação: 0    Alteração: 0    Data liberação:
//
//    Autorizou liberação
//
// Histórico do projeto
//    Vers Modf Altr  Autor    Data     Observações
//
// Alterações de emergência realizadas e ainda não passadas a limpo
// Alterações e correções realizadas
//
////////////////////////////////////////////////////////////////////////

#include  <afxwin.h>

// controle de debug, usado pela macro ASSERT

   #ifdef _DEBUG
      #undef   THIS_FILE
      static   char BASED_CODE THIS_FILE[] = __FILE__ ;
      #define  new  DEBUG_NEW
   #endif

//--- Inclusão do módulo de definição próprio ---

   #define  _exmaq_OWN
   #include "exmaq.hpp"
   #undef   _exmaq_OWN

//--- Declaração dos membros estáticos das classes ---

////////////////////////////////////////////////////////////////////////
//
// Implementação da classe: Classe 1
//
// Descrição da classe
//    - Classe 1 -
//    Esta classe ilustra as formas básicas de instâncias de transição.
//
////////////////////////////////////////////////////////////////////////

//--- Implementação da classe: Classe 1

//--- Mapa de mensagens da classe ---

   - Classe 1 -
   !CODIGO!
   Código: cabeçalho do mapa de mensagens

   //--- Mapa de mensagens gerado a partir do diagrama ---

      // Transição: Transição final, com um processo na entrada
                - Transição final, com um processo na entrada -
```

```
               !CODIGO!
               Nome da macro no mapa de mensagens
      // Transição: Transição final, sem processos na transição
               - Transição final, sem processos na transição -
               !CODIGO!
               Nome da macro no mapa de mensagens
      // Transição: Transição inicial
               - Transição inicial -
               !CODIGO!
               Nome da macro no mapa de mensagens
      // Transição: Transição normal
               - Transição normal -
               !CODIGO!
               Nome da macro no mapa de mensagens
      // Transição: Transicao pura
               - Transicao pura -
               !CODIGO!
               Nome da macro no mapa de mensagens

   //--- Mapa de mensagens gerado a partir dos pacotes ---

   //--- Extensão ao mapa de mensagens definido na classe ---

      - Classe 1 -
      !CODIGO!
      Código: adições ao mapa de mensagens

   END_MESSAGE_MAP()

//--- Define's da máquina de estados ---

   #ifndef      ESTADO_ERRO
      #define   ESTADO_ERRO         10
   #endif
   #define   ESTADO_1              11
   #define   ESTADO_2              12
   #define   ESTADO_3              13
   #define   ESTADO_4              14
   #define   ESTADO_5              15
   #define   FIM_Estado            16
   #define   NIL_Estado            17

////////////////////////////////////////////////////////////////////////////
//
// Função transição: Transição final, com um processo na entrada
//
// Descrição da função transição
//    - Transição final, com um processo na entrada -
//    Esta transição é executada somente se estiver em um dos estados
//    de entrada.
//    No diagrama figuram processos em algumas das arestas de entrada.
//    Como não tem saída, não altera o estado corrente.
//
////////////////////////////////////////////////////////////////////////////

   - Transição final, com um processo na entrada -
   !CODIGO!
   Declaração: cabeçalho da função que implementa a transição
   {

   // controle de validade do estado corrente

      ASSERT(( ESTADO_ERRO < Estado ) && ( Estado < FIM_Estado )) ;

   // Declarações locais da transição

      - Transição final, com um processo na entrada -
      !CODIGO!
      Código de declaração da transição
```

```
   // Código pre-âmbulo da transição

      - Transição final, com um processo na entrada -
      !CODIGO!
      Código fonte pre-âmbulo da transição

   // Seleção de condição de entrada

      switch ( Estado )
      {

         case ESTADO_3 :
            - Processo 1 -
            !CODIGO!
            Código do processo
            break ;

         case ESTADO_1 :
            - Processo 2 -
            !CODIGO!
            Código do processo
            break ;

         case ESTADO_2 :
            break ;

         default :
            - Transição final, com um processo na entrada -
            !CODIGO!
            Código fonte de escape de ativação

      } // Fim de seleção de condição de entrada

   // Código da função transição

      - Transição final, com um processo na entrada -
      !CODIGO!
      Código fonte da transição
      // Transição sem sucessor näo altera o estado corrente


   // Código final da transição

      - Transição final, com um processo na entrada -
      !CODIGO!
      Código fonte final transição

   } // Fim da função transição: Transição final, com um processo na entrada

////////////////////////////////////////////////////////////////////////////
//
// Função transição: Transição final, sem processos na transição
//
// Descrição da função transição
//    - Transição final, sem processos na transição -
//    Esta transição é executada somente se estiver em um dos estados
//    de entrada.
//    No diagrama nao figuram processos nas arestas de entrada.
//    Como näo tem saída, näo altera o estado corrente
//
////////////////////////////////////////////////////////////////////////////

   - Transição final, sem processos na transição -
   !CODIGO!
   Declaraçäo: cabeçalho da função que implementa a transição
   {

   // controle de validade do estado corrente
```

```
      ASSERT(( ESTADO_ERRO < Estado ) && ( Estado < FIM_Estado )) ;

   // Código pre-âmbulo da transição

      - Transição final, sem processos na transição -
      !CODIGO!
      Código fonte pre-âmbulo da transição

   // Seleção de condição de entrada

      switch ( Estado )
      {

         case ESTADO_1 :
         case ESTADO_2 :
         case ESTADO_3 :
            break ;

         default :
            return ;

      } // Fim de seleção de condição de entrada

   // Código da função transição

      - Transição final, sem processos na transição -
      !CODIGO!
      Código fonte da transição
      // Transição sem sucessor não altera o estado corrente


   // Código final da transição

      - Transição final, sem processos na transição -
      !CODIGO!
      Código fonte final transição

   } // Fim da função transição: Transição final, sem processos na transição

////////////////////////////////////////////////////////////////////////
//
// Função transição: Transição inicial
//
// Descrição da função transição
//    - Transição inicial -
//    Esta transição é incondicional
//    Dependendo de condições definidas na aresta de saída, seleciona
//       o estado destino
//
////////////////////////////////////////////////////////////////////////

   - Transição inicial -
   !CODIGO!
   Declaração: cabeçalho da função que implementa a transição
   {

   // Código pre-âmbulo da transição

      - Transição inicial -
      !CODIGO!
      Código fonte pre-âmbulo da transição
   // Transição incondicional


   // Código da função transição

      - Transição inicial -
      !CODIGO!
```

```
   Código fonte da transição

// Determinar próximo estado corrente

   {
      int Novo_Estado ;

   // Instância de transição incondicional


      if ( - Seleciona 4 -
           !CODIGO!
           Condição de seleção do estado destino )
      {
         Novo_Estado = ESTADO_4 ;
         - Seleciona 4 -
         !CODIGO!
         Código do processo
      } else {
         Novo_Estado = ESTADO_1 ;
      } // if

   // Ativa estado destino

      if ( Novo_Estado != NIL_Estado )
      {

      // Estabelecer invariante do estado sucessor

         if ( Estado != Novo_Estado )
         {
            switch ( Novo_Estado )
            {

               case ESTADO_4 :
                  Estado 4
                  Código fonte para estabelecer a invariante do estado
                  break ;

               case ESTADO_1 :
                  Estado 1
                  Código fonte para estabelecer a invariante do estado
                  break ;
            } // switch
         } // if

      // Ativar estado sucessor

         switch ( Novo_Estado )
         {

            case ESTADO_4 :

               Estado 4
               Código fonte do estado
               break ;

            case ESTADO_1 :

               Estado 1
               Código fonte do estado
               break ;
         } // switch

         Estado = Novo_Estado ;

      } // Fim seleção de ativar estado destino

   } // Fim determinar próximo estado corrente
```

```
   // Código final da transição

      - Transição inicial -
      !CODIGO!
      Código fonte final transição

   } // Fim da função transição: Transição inicial

////////////////////////////////////////////////////////////////////////
//
// Função transição: Transição normal
//
// Descrição da função transição
//    - Transição normal -
//    Esta transição depende do estado ao iniciar
//    Define o novo estado
//
////////////////////////////////////////////////////////////////////////

   - Transição normal -
   !CODIGO!
   Declaração: cabeçalho da função que implementa a transição
   {

   // controle de validade do estado corrente

      ASSERT(( ESTADO_ERRO < Estado ) && ( Estado < FIM_Estado )) ;

   // Declarações locais da transição

      - Transição normal -
      !CODIGO!
      Código de declaração da transição

   // Código pre-âmbulo da transição

      - Transição normal -
      !CODIGO!
      Código fonte pre-âmbulo da transição

   // Seleção de condição de entrada

      switch ( Estado )
      {

         case ESTADO_1 :
            - Processo 3 -
            !CODIGO!
            Código do processo
            break ;

         case ESTADO_2 :
            break ;

         default :
            - Transição normal -
            !CODIGO!
            Código fonte de escape de ativação

      } // Fim de seleção de condição de entrada

   // Código da função transição

      - Transição normal -
      !CODIGO!
      Código fonte da transição

   // Determinar próximo estado corrente
```

```
{
   int Novo_Estado ;

// Definir destino por instância da transição

   switch ( Estado )
   {

      case ESTADO_2 :
      case ESTADO_1 :

         if ( - Seleção 2 -
              !CODIGO!
              Condição de seleção do estado destino )
         {
            Novo_Estado = ESTADO_2 ;
            - Seleção 2 -
            !CODIGO!
            Código do processo
         } else if ( - Seleciona 4 -
                     !CODIGO!
                     Condição de seleção do estado destino )
         {
            Novo_Estado = ESTADO_4 ;
            - Seleciona 4 -
            !CODIGO!
            Código do processo
         } else {
            Novo_Estado = ESTADO_5 ;
         } // if
         break ;
      } // Fim selecionar instâncias da transição

// Ativa estado destino

   if ( Novo_Estado != NIL_Estado )
   {

   // Liberar recursos do estado anterior

      if ( Estado != Novo_Estado )
      {
         switch ( Estado )
         {

            case ESTADO_2 :
               Estado 2
               Código fonte para liberar recursos do estado
               break ;

            case ESTADO_1 :
               Estado 1
               Código fonte para liberar recursos do estado
               break ;
         } // switch
      } // if

   // Estabelecer invariante do estado sucessor

      if ( Estado != Novo_Estado )
      {
         switch ( Novo_Estado )
         {

            case ESTADO_2 :
               Estado 2
               Código fonte para estabelecer a invariante do estado
               break ;
```

```
                  case ESTADO_4 :
                     Estado 4
                     Código fonte para estabelecer a invariante do estado
                     break ;

                  case ESTADO_5 :
                     Estado 5
                     Código fonte para estabelecer a invariante do estado
                     break ;
               } // switch
            } // if

         // Ativar estado sucessor

            switch ( Novo_Estado )
            {

               case ESTADO_2 :

                  Estado 2
                  Código fonte do estado
                  break ;

               case ESTADO_4 :

                  Estado 4
                  Código fonte do estado
                  break ;

               case ESTADO_5 :

                  Estado 5
                  Código fonte do estado
                  break ;
            } // switch

            Estado = Novo_Estado ;

         } // Fim seleção de ativar estado destino

      } // Fim determinar próximo estado corrente

   // Código final da transição

      - Transição normal -
      !CODIGO!
      Código fonte final transição

   } // Fim da função transição: Transição normal

////////////////////////////////////////////////////////////////////////////
//
// Função transição: Transicao pura
//
// Descrição da função transição
//    - Transicao pura -
//    Esta transição executa incondicionalmente.
//    Como não tem saída, não altera o estado corrente
//
////////////////////////////////////////////////////////////////////////////

   - Transicao pura -
   !CODIGO!
   Declaração: cabeçalho da função que implementa a transição
   {

   // Código pre-âmbulo da transição
```

```
      - Transicao pura -
      !CODIGO!
      Código fonte pre-âmbulo da transição
   // Transição incondicional


   // Código da função transição

      - Transicao pura -
      !CODIGO!
      Código fonte da transição
      // Transição sem sucessor não altera o estado corrente


   // Código final da transição

      - Transicao pura -
      !CODIGO!
      Código fonte final transição

   } // Fim da função transição: Transicao pura

//--- Fim da classe: Classe 1

////// Fim do módulo: Modulo 1 ////
```

Bibliografia

[Alpert92] Alpert S. R.; “Graceful interaction with graphical constrainsts”, Yorktown Heights, N.Y., IBM Research Division, 1992.

[Boehm76] Boehm B. W.; “Software Engineering”, *IEEE Transactions on Computers*, C25(12):1226-1241, 1976.

[Boehm86] Boehm B. W.; “A spiral model of software development and enhancement”, *ACM SIGSOFT Software Engineering Notes*, vol. 11, no. 4, pags. 14-24, 1986.

[Booch93] Booch G.; *Object-oriented Analysis and Design with Applications*, 2nd edition, Benjamin Cummings, Redwood City. ISBN 0-8053-5340-2, 1993.

[Brooks86] Brooks, F. P.; “No Silver Bullet - Essence and Accidents of Software Engineering”; *IEEE Computer*; pags. 10-19; Outubro 1986.

[Chen76] Chen P. P.; “The Entity-Relationship Model: Toward a Unified View of Data”, *ACM Transactions on Database Systems*, vol. 1, no. 1, pags. 9-36, 1976.

[Cole92] Coleman D., Hayes F., Bear S.; “Introducing Objectcharts or How to Use Statecharts in Object-Oriented Design”; *IEEE Trans. Software Eng.*, vol. 18, no. 1, pags. 9-18, 1992.

[C++_93] *Microsoft C++ Language Reference*; Visual C++ 1.5 Books Online, 1993.

[Dow91] Dowson M.; “Process and Project Management”; *Proceedings of the 7th International Software Process Workshop*, pags. 57-61, Outubro de 1991, Yountville, California, EUA.

[DeMa78] DeMarco T.; *Structured Analysis and System Specification*; New York: Yourdon Press, 1978.

[Deme82] Demetrovics J., Knuth E., Radó P.; “Specification Meta Systems”, *IEEE Computer*, vol. 15, no. 5, pags. 29-35, 1982.

[Foley89] Foley J. D., Kim W.C., Kovacevic S., Murray K.; “Defining interfaces at a high level of abstraction”, *IEEE Software*, vol. 6, no. 1, pags. 25-32, 1989.

[Gane79] Gane C., Sarson T.; *Structured Systems Analysis*; Englewood Cliffs, NJ: Prentice-Hall, 1979.

[Grag90] Garg P. K., Scacchi W.; “A hypertext system to manage software life-cycle documents”, *IEEE Software*, vol. 7, no. 3, pags. 90-98, 1990.

[Gibson89] Gibson M. L., Orr K., Gane C., Yourdon E., Chen P., Constantine L., McClure C.; “In Depth: CASE”; *BYTE*, vol. 14, no. 4, pags. 206-246, 1989.

[Green86] Green, M.; “A survey of three dialogue models”, *ACM Transactions on Graphics*, vol. 5, no. 3, pags. 244-275, 1986.

[Harel88] Harel D.; “On Visual Formalisms”, *Communications of the ACM*, vol. 31, no. 5, pags. 514-530, 1988.

[Hart89a] Hartson R.; “User-interface management control and communication”, *IEEE Software*, vol. 6, no. 1, pags. 62-70, 1989.

[Hart89b] Hartson R., Hix D.; “Human-Computer Interface Development: Concepts and Systems for its Management”, *ACM Computing Surveys*, vol. 21, no. 1, pags. 5-92, 1989.

[Hix90] Hix D.; “Generations of user-interface management systems”, *IEEE Software*, vol. 7, no. 5, pags. 77-87, 1990.

[Hopc79] Hopcroft J. E., Ullman J. D.; *Introduction to Automata Theory, Languages, and Computation*; Addison-Wesley, 1979.

[Hübsch95] Hübscher P. J. E., Sant'Anna M., Staa A. v.; *Ferramentas TALISMAN para o desenvolvimento de programas Windows, utilizando C++ e MFC*; a ser publicado.

[Huds87] Hudson S. E.; "UIMS support for direct manipulation interfaces", *ACM SIGGRAPH Computer Graphics*, vol. 21, no. 2, pags. 120-124, 1987.

[Jacob85] Jacob R. J. K.; "A state transition diagram language for visual programming", *IEEE Computer*, vol. 18, no. 8, pags. 51-59, 1985.

[Jacob86] Jacob R. J. K.; "A specification language for direct-manipulation interfaces", *ACM Transactions on Graphics*, vol. 5, no. 4, pags. 283-317, 1986.

[Jorg95] Jorgensen P. C.; *Software Testing: A Craftsman's Approach*; CRC Press, ISBN 0-8493-7345-X, 1995.

[MFC_25] *Microsoft Foundation Class Library User's Guide*; Visual C++ 1.5 Books Online, 1993.

[NP-05] Staa A. v., Costa G. M., Kanamori L. N., Giovani P. A. O., Correia R.; *Regras e recomendações para a inclusão de especificações no código de programas C ou C++*; NP-05, Projeto de Normas e processos de desenvolvimento, PUC-Rio, 1995.

[NP-08] Staa A. v., Costa G. M., Kanamori L. N., Giovani P. A. O., Correia R.; *Regras e recomendações para o uso de instrumentação em programas C e C++*; NP-08, Projeto de Normas e processos de desenvolvimento, PUC-Rio, 1995.

[Oman90] Oman P. W.; "CASE analysis and design tools", *IEEE Software*, vol. 7, no. 3, pags. 37-43, 1990.

[PCCW93] Paulk M. C., Curtis B., Chrissis M. B., Weber C. W.; *Capability Maturity Model for Software, version 1.1*; Carnegie Mellon Univeristy, Software Engineering Institute; Technical Report CMU/SEI-93-TR-24, 1993.

[RBP+91] Rumbaugh J., Blaha M., Premberlani W., Eddy F., Lorensen W.; *Object-Oriented Modeling and Design*; Prentice-Hall Inc., 1991.

[Shne83] Shneiderman B.; "Direct manipulation: a step beyond programming languages", *IEEE Computer*, vol. 16, no. 8, pags. 57-69, 1983.

[Sore88] Sorenson P.G., Tremblay J.P., McAllister A.J.; "The Metaview system for many specification environments", *IEEE Software*, vol. 5, no. 2, pags. 30-38, 1988.

[Staa87] Staa A. v.; *Engenharia de Programas*, 2ª edição; Livros Técnicos e Científicos, Rio de Janeiro; 1987.

[Staa93] Staa A.v.; *Projeto TOTEM, Meta-Ambiente de Desenvolvimento de Software, Definição de Projeto*, PUC-Rio, 1993.

[STAA92] STAA Informática; *Ambiente de Desenvolvimento de Software TALISMAN - Manual de Referência*; Versão 4.2; Rio de Janeiro; 1992.

[Stratt89] Stratton S. D., Dunsmore H. E.; *The Use of Hypertext in Software Development*; Purdue University, Software Engineering Reserach Center, Technical Report SERC TR-36-P, 1989.

[Valdés95] Valdés R.; "Introducing Interoperable Objects"; *Dr. Dobb's Journal*; #225, vol. 19, issue. 16, pags. 4-6, Winter 1994/95.

[Wasse85] Wasserman A.I.; "Extending state transition diagrams for the specification of human-cumputer interaction"; *IEEE Trans. Software Eng.*, vol. SE-11, no. 8, pags. 699-713, 1985.

[Wasse87] Wasserman A.I.; "A Graphical, extensible integrated environment for software development", *ACM SIGPLAN Notices*, vol. 22, no. 1, pags. 131-142, 1987.

[Wu89] Wu P.Y.F; "User interface management systems: survey and assessment", *Yorktown Heights, N.Y., IBM Research Division*, 1989.

"UM PROCESSO DE DESENVOLVIMENTO DE SOFTWARE UTILIZANDO TALISMAN E MFC". Dissertação de Mestrado apresentada por Pedro Jorge Eugenio Hübscher em 11 de Outubro de 1995 ao Departamento de Informática da PUC-Rio e aprovada pela Comissão Julgadora, formada pelos seguintes professores:

Prof. Arndt von Staa
Orientador
Depto. de Informática – PUC-Rio

Júlio César Sampaio do Prado Leite.
Depto. de Informática – PUC-Rio

Marcelo Gattass
Depto. de Informática – PUC-Rio

Visto e permitida a impressão
Rio de Janeiro, / /

Coordenador dos Programas de Pós-Graduação
e Pesquisa do Centro Técnico Científico