Índice | Próximo

1 Introdução à Engenharia de Software

Objetivos

Apresentar as motivações pela qual a disciplina e a profissão de engenharia de software faz-se necessária. Veremos algumas definições e uma visão geral da área. Discutiremos o que é software, diferenciando-o de programa e vendo-o como um artefato conceitual. Discutiremos ainda as diferenças entre programação e engenharia de software.

Algumas questões deverão ser discutidas:

- Qual o contexto de um software?
- Software e programa são a mesma coisa?
- Engenharia de Software é o mesmo que programação?

1.1 Sistemas baseados em computador

Um **sistema baseado em computador** é aquele que automatiza ou apóia a realização de atividades humanas através do processamento de informações.

Um sistema baseado em computador é caracterizado por alguns elementos fundamentais.

- Hardware
- Software
- Informações
- Usuários
- Tarefas
- Documentação

O **hardware** corresponde às partes eletrônicas e mecânicas (rígidas) que possibilitam a existência do software, o armazenamento de informações e a interação com o usuário. A CPU, as memórias primária e secundária, os periféricos, os componentes de redes de computadores, são exemplos de elementos de hardware. Um único computador pode possibilitar a existência de diversos sistemas e um sistema pode requisitar diversos computadores.

O **software** é a parte abstrata do sistema computacional que funciona num hardware a partir de instruções codificadas numa linguagem de programação. Estas instruções permitem o processamento e armazenamento de informações na forma de dados codificados e podem ser controladas pelo usuário. Este controle, bem como a troca de informações entre o usuário e o sistema é feita através da interface de usuário, composta por hardware e software.

A informação é um componente fundamental nos sistemas baseados em computador. Por isto eles podem também ser chamados de sistemas baseados em **informação**. Sistemas processam e armazenam dados que são interpretados como informações pelos usuários através da interface. São os dados que representam elementos do domínio que tornam o sistema útil para os usuários.

Os **usuários** são também elementos centrais no desenvolvimento de um sistema baseado em computador. As metas de cada usuário, de acordo com o papel que cada um desempenha no domínio, devem poder ser satisfeita pelo sistema.

As **tarefas** ou procedimentos compreendem as atividades que o sistema realiza ou permite realizar. As tarefas caracterizam a funcionalidade do sistema e devem permitir aos usuários satisfazer as suas metas.

A **documentação** do sistema envolve os *manuais de usuário*, que contém informações para o usuário utilizar o sistema (*documentação do sistema*) que descrevem a sua estrutura e o funcionamento. Estes últimos são fundamentais durante o desenvolvimento do sistema para a comunicação entre a equipe de desenvolvimento e para a transição entre as suas diversas etapas e durante a manutenção de um sistema em sua fase operacional.

Um sistema baseado em computador funciona num determinado *domínio de aplicação* que corresponde a um tipo de ambiente ou organização onde o sistema é utilizado.

Exemplos de sistemas baseados em computador

- Sistema de Automação Bancária
- Sistema de Folha de Pagamento
- Sistema de Controle Acadêmico
- Sistema de Biblioteca
- Sistema de Controle de Tráfego Urbano
- Sistema de Controle de Elevadores
- Sistema de Editoração de Jornais e Revistas

Engenharia de Sistemas baseados em computador

O desenvolvimento do sistema deve ser pensado como um todo. Os problemas que o sistema deve resolver devem ser analisados e uma solução envolvendo todos os componentes deve ser proposta. O desenvolvimento de cada componente do sistema pode ser conduzido utilizando um "engenharia" específica. É importante ressaltar que o termo engenharia está sendo utilizado de forma imprecisa.

- **Engenharia de Hardware** - construção dos diversos equipamento de hardware, engenharia de redes, etc.
- **Engenharia de Software** - desenvolvimento dos diversos componentes de software que compõe o sistema
- **Engenharia de Informações** - modelagem e estruturação das informações para que possam ser armazenadas na forma de dados relacionados entre si.
- **Engenharia de Usabilidade (ou de Fatores Humanos)** - Os fatores humanos devem ser analisados para que as atividades humanas sejam desempenhada com qualidade.
- **Engenharia de Procedimentos ou Métodos** - Novas tarefas dos usuários devem surgir e outras podem ser extintas. Outras atividades podem ser automatizadas pelo sistema

Todas estas "engenharias" devem ser concebidas de forma integrada uma vez que o seus elementos estão bastante relacionados entre si. A ênfase em apenas um dos aspectos pode levar a deficiências do sistema em alguns outros componentes.

Na construção de um sistema não podemos nos concentrar apenas na engenharia de software. É preciso considerar que o hardware, bem como as informações e os procedimentos do domínio precisam ser analisados e construídos de forma integrada ao sistema.

Os três níveis de um sistema baseado em computador

Podemos identificar três níveis distintos de visualização de um sistema baseado em computador:

- Nível do domínio ou da organização

No **nível do domínio** todos o componentes podem ser visualizados de forma integrada. Pode-se observar como cada um deles estão relacionados. Neste nível o hardware e o software são visto como um componente único que interage com os diversos usuários de maneira a permitir que as tarefas do domínio possam ser realizadas. Neste nível avalia-se o quanto o sistema resolve os problemas da organização.

Neste nível podemos identificar duas perspectivas distintas. Nas **perspectiva de automação** os componentes hardware/software e usuários são vistos como agentes das tarefas que o sistema pode realizar. Na **perspectiva de mídia** o sistema é visto como a plataforma de suporte às atividades de comunicação entre os usuários através do apoio no processamento, armazenamento e veiculação das informações. Enquanto a primeira perspectiva considera usuários como processadores de informação ao lado do computador, na segunda eles são considerados seres inteligentes e a função do computador é aumentar a sua capacidade.

- Nível da interação usuário-computador

Neste nível a ênfase está nas atividades que os usuários devem fazer e na forma como as informações são percebidas ou produzidas pelos usuários. Neste nível, as tarefas do domínio devem determinar metas para cada usuário que devem poder ser realizadas utilizando o sistema. Hardware e software são vistos através da interface de usuário que pode ser compostas por elementos como, tela, teclado, mouse, menus, comandos, ícones, etc.

É neste nível que se avalia a usabilidade do sistema - facilidade de uso e de aprendizado, produtividade, satisfação, etc.)

Também podemos distinguir duas perspectivas. Na **antropomórfica** busca-se construir um sistema que tenha características cognitivas humanas. Neste caso o sistema deveria interagir na mesma língua do usuário e ter um comportamento inteligente. A outra perspectiva é a de **ferramenta intelectual** que considera que o sistema deve expandir a capacidade cognitiva do usuário, apoiando-o em suas atividades.

- O nível da computação

Este é o nível da tecnologia. A ênfase está no hardware e software do sistema. É preciso definir como os procedimentos podem ser automatizados e como as informações podem ser codificadas. Neste nível podemos identificar os elementos de hardware - CPU, memórias, periféricos - e de software - sistema operacional, sistemas de janelas e aplicação (ou aplicativo) de software. Este último compreende o software que implementa uma funcionalidade que pode ser aplicada num certo domínio.

Neste nível é importante avaliar a qualidade dos diversos elementos do computador: confiabilidade, segurança, eficiência, etc.

Diferentes perspectivas podem ser identificadas. A perspectiva *orientada-a-funções* enfatiza nas funções que o sistema deve realizar. A perspectiva *orientada-a-dados* centraliza todo o sistema na codificação, estruturação, armazenamento e processamento dos dados. Por último, a perspectiva *orientada-a-objetos* considera que o sistema deve ser composto por objetos que encapsulam dados e funções.

1.2 O Software

Existem problemas que podem ser resolvidos por um algoritmo. **Algoritmo** é uma seqüência de operações *bem-definidas* de uma máquina (abstrata ou real) que a partir de dados de entrada pára e pode fornecer resultados (saída).

Um **programa** é um conjunto de soluções algorítmicas que operam sobre estruturas de dados codificadas numa linguagem de programação. Um programa pode está num estado estático e dinâmico - programa executando num computador.

Um programa **estático** pode estar em diferentes formas. O programa fonte é aquele escrito por um programador na linguagem de programação antes de ser traduzido para o código de máquina que a linguagem vai executar. O programa executável é o conjunto de instruções em um código que pode ser executado diretamente pelo processador. Temos ainda o programa em código objeto que é aquele compilado a partir do programa fonte e antes de ser associado a bibliotecas estáticas.

O **software** é normalmente visto como um programa que pode ser executado num computador. Entretanto esta visão é limitada. Precisamos ressaltar a importância de que o software é mais do que um programa. Embora em sua essência software e programa sejam a mesma coisa, podemos adquirir uma nova visão para o seu desenvolvimento se considerarmos as diferenças que podem ser identificadas.

O software deve ser visto como um **artefato virtual** e como tal ele é um produto a ser aplicado com utilidade num certo domínio, sendo chamado de **aplicação de software**. Neste sentido, o software como produto é algo que possui um **modelo conceitual** próprio - a sua funcionalidade e a sua interatividade. A

funcionalidade determina aquilo que ele faz e que o torna útil para resolver problemas dos usuários. A **interatividade** determina a maneira como o usuário deverá utilizar o software. O software é, portanto, um produto conceitual ou lógico.

Precisamos também chamar a atenção para uma diferença de perspectiva. Num programa a perspectiva está em como um certa solução é resolvida por algoritmos. A perspectiva de aplicação de software, o software como produto, está em como ele pode resolver os problemas do usuário.

Um outro importante aspecto que diferencia programa de software é a complexidade. Um software pode ser composto por diversos programas que interagem entre si, bem como podem acessar arquivos de dados e utilizar bibliotecas dinâmicas. Por fim, a atividade de construir programas requer técnicas e ferramentas distintas daquelas que são necessárias para construir um software.

Exemplo de um software como artefato virtual

Para exemplificar esta perspectiva de software como artefato, vejamos o exemplo exemplo de um programa trivial que, com pequenas alterações no programa, mas sem alterar a essência do algoritmo, pode ser visto como aplicações de software de distintas.

Vejamos o seguinte trecho de interação entre o usuário (U) e o sistema (S).

```
S: Forneça um número:
U: 5.3
S: Você deve digitar apenas números inteiros:
U: 5
S: Forneça outro número:
U: 8
S: O resultado é -3
```

Observando a interação do usuário com o sistema podemos concluir que o software realizou uma subtração. Assim, a funcionalidade deste software é realizar subtrações de dois números inteiros. Vejamos, agora o algoritmo que está por trás deste software.

```
escreva("Forneça um Número:");
leia A;
enquanto A não for inteiro faça
        escreva(Você deve digitar apenas números inteiros:)
        leia A;
escreva("Forneça outro Número:");
leia B;
C = A - B;
escreva("O resultado é ", C);
```

Vejamos agora o que acontece se modificarmos as linhas 1, 6 e 9 deste algoritmo para

```
escreva("Forneça o valor de venda");
escreva("Forneça o valor de compra");
escreva("O lucro é",C);
```

Para o programador muito pouca coisa mudou, mas para o usuário deste sistema este software agora é visto como uma aplicação de cálculo de lucros de vendas. As alterações no programa foram bastante pequenas, mas produziu um efeito significativo. A perspectiva de software como artefato deve considerar como o usuário vê a aplicação. Os modelos conceituais da funcionalidade do software são distintos. Desta forma, temos software distintos com aplicações em domínios distintos. O programa nos fornece uma visão de funcionamento. O usuário possui apenas a visão de funcionalidade.

Podemos modificar as mesmas linhas do programa para:

```
escreva("Salário Bruto");
escreva("Descontos");
escreva("Salário líquido,C);
```

que produz um novo software.

Como este exemplo podemos entender melhor que embora o software seja um programa, ele precisa ser considerado como um artefato. Nestes exemplos, temos artefatos virtuais diferentes para programas com um mesmo algoritmo.

Componentes do Software

O software pode ser visto nos estados **dinâmico** e **estático**. No estado estático podemos ver o software formado por componentes lógicos como variáveis de dados, funções, procedimentos, módulos, classes, etc., que podem variar dependendo da linguagem que foi utilizada na sua implementação. Este componentes podem estar na forma de programa fonte, objeto, executável ou em bibliotecas estáticas a serem ligadas ao código objeto.

No estado dinâmico os componentes que formam um software podem ser objetos, agentes, processos, arquivos de dados e outros, que são gerenciados por sistemas operacionais ou sistemas de middleware. Estes componentes podem está agrupados em arquivos executáveis, bibliotecas de ligação dinâmica ou outros. Estes componentes podem interagir por chamada de função, troca de mensagens, acesso compartilhado.

A visão de software formado por componentes interconectados entre si caracteriza a **arquitetura do software**.

Evolução do software

Para que um software possua uma ampla utilidade ele necessita de centenas ou milhares de algoritmos. Atualmente os softwares são bastante complexos e podem ser formados por diversos programas.

O software evoluiu bastante na segunda metade do século 20 quando surgiram os primeiros computadores. Nos **primeiros anos**, de 1950 a meados dos anos 60, os softwares eram executados em computadores de baixa capacidade (embora ocupassem salas enormes), com o processamento de dados em lote (sem interação

com o usuário no decorrer da interação), e feitos para clientes específicos com distribuição limitada.

Na **segunda era**, de meados dos 60's a meados dos 70's surgiram sistemas de grande porte multi-usuários e aplicações de banco de dados. Começou-se a comercializar software genéricos que atendiam a necessidades de diversos clientes. Surgiram também os sistemas de tempo-real. No Brasil software com estas características forma utilizados até meados dos anos 80. Linguagens de programação de alto-nível facilitou em parte a construção de software nesta segunda era.

A **terceira era** começou em meados do anos 70 com o surgimento de hardware de baixo custo baseados em microprocessadores e estendeu-se até início dos anos 90. Com o hardware mais barato, mini e microcomputadores foram adquiridos por diversas empresas e, conseqüentemente, houve uma grande demanda por software. A falta de mão-de-obra especializada, de técnicas e ferramentas de desenvolvimento de software, e de planejamento e gerenciamento do processo para atender a este impacto de consumo levaram à **crise do software**. Este crise do software levou ao surgimento de técnicas de programação como modularização e refinamento sucessivos e metodologias para desenvolvimento de software como a Análise e Projeto Estruturados e o Método de Jackson que buscavam resolver os problemas de desenvolvimento de software.

A **quarta era**, a partir de meados dos anos 80, caracteriza-se pelo surgimento de estações de trabalho "desktop" bastante poderosas interligadas em redes e de baixo custo - os sistemas distribuídos. Grandes empresas optaram por trocar computadores de grande porte por sistemas distribuídos num fenômeno denominado de *downsize*. A tecnologia de orientação-a-objetos e a de interfaces gráficas baseadas em janelas caracterizam a produção de software nesta era. A popularização dos microcomputadores de baixo custo levou ao surgimento de inúmeros softwares aplicativos "de prateleira" comprados por pequenas e micro-empresas e também por usuários comuns.

O surgimento da *World Wide Web*, no início da década de 90, popularizou bastante a utilização da Internet e vem causando uma revolução na forma como as aplicações são desenvolvidas, utilizadas e comercializadas. O surgimento de linguagens voltadas para estes ambientes, como Java, e de plataformas de interoperabilidade para sistemas de software distribuídos, como *CORBA*, vem indicando que estamos entrando numa nova era.

Tamanho do software

Muitas pessoas não têm noção da complexidade de um software, do seu tamanho e da quantidade de pessoas envolvidas no seu desenvolvimento. A tabela abaixo classifica diferentes tamanhos de software.

Categoria	Tamanho da equipe	Duração do desenvolvimento	Tamanho em linhas de código
trivial	1	1-4 semanas	500 linhas
pequeno	1	1-6 meses	1000 a 2000 linhas
médio	2-5	1-2 anos	5K a 50K
grande	5-20	2-3 anos	50K a 100 K
muito grande	100-1000	4-5 anos	1 M
extremamente grande	2000-5000	5-10 anos	1M a 10M

O Windows 95 da Microsoft foi desenvolvido por uma equipe de 200 pessoas e possui 11 milhões de linhas de código fonte.

Qualidades do software

O software como um produto deve ter qualidade. Diversas são as qualidades do software a serem avaliadas. É preciso avaliar tanto a qualidade do produto em si com a do processo de desenvolvimento. Vejamos algumas das qualidades que podem ser avaliadas.

- **Corretude** - um software precisa funcionar corretamente. Um software correto é aquele que satisfaz a sua especificação e que não possui falhas ou erros.
- **Validade** - um software válido é aquele cuja especificação satisfaz aos requisitos dos usuários e da organização, isto é, está de acordo com as necessidades dos usuários.
- **Robustez** - o software deve prever que o usuário de agir de forma não esperada e deve ser capaz de resistir a estas eventuais situações incomuns sem apresentar falhas.
- **Confiabilidade** - um software correto e robusto ganha a confiança dos usuários uma vez que ele deve se comportar como esperado e não falha em situações inesperadas.
- **Eficiência** - o software deve realizar suas tarefas em um tempo adequando à complexidade de cada uma delas. A utilização dos recursos de hardware (memória, disco, tráfego de rede) também deve ser feita de forma eficiente.
- **Usabilidade** - o software precisa ser fácil de aprender e de usar, permitir maior produtividade do usuário, flexibilidade de utilização, flexibilidade de aplicação e proporcionar satisfação de uso.
- **Manutenibilidade** - todo software precisa de manutenção, seja para corrigir erros ou atender a novos requisitos. O software deve ser fácil de manter para que estas correções ou atualizações sejam feitas com sucesso.
- **Evolutibilidade** - todo software precisa evoluir para atender novos requisitos, para incorporar novas tecnologias ou para expansão de sua funcionalidade.
- **Portabilidade** - o software deve poder ser executado no maior número possível de equipamentos de hardware.
- **Interoperabilidade** - software em diferentes plataformas devem poder interagir entre si. Esta qualidade é essencial em sistemas distribuídos uma vez que o software pode estar sendo executado em diferentes computadores e sistemas operacionais. É interessante que diferentes elementos de software distintos possam ser utilizados em ambos. Por exemplo, uma certo arquivo com uma imagem feita num aplicativo deve poder ser vista em outros aplicativos.
- **Reusabilidade** - diversos componentes de um software devem poder ser reutilizados por outras aplicações. O reuso de funções e objetos facilita bastante o desenvolvimento de software.

Existem diversas outras qualidades do software que podem avaliadas. Nosso objetivo foi apenas introduzir algumas delas, mesmo que de maneira superficial. Para um estudo inicial veja alguns livros na nossa bibliografia.

1.3 Engenharia de Software

Vimos nas seções anteriores que, embora o software seja um programa em sua essência, ele precisa ser visto como um artefato virtual que tem associado a si um modelo conceitual. Vamos discutir agora as diferenças entre construir um programa, isto é, programar, e o que vem a ser engenharia de software.

Programar é engenharia?

O desenvolvimento de um software requer pelo menos a construção de um programa. Programar envolve as atividades de **codificar** um algoritmo numa determinada linguagem de programação. O resultado disto é o programa ou **código fonte**. Este programa precisa ser **traduzido** por um interpretador ou compilador em linguagem de máquina (**código executável**) para que possa ser executado pelo processador. Um programa pode reutilizar ou não funções de bibliotecas disponíveis pelo tradutor. No processo de **compilação**, o código fonte é traduzido em um **código objeto** para em seguida ser "**ligado**" a eventuais funções de bibliotecas (processo de *linking*) quando então é gerado o **programa executável** em linguagem de máquina. Alguns sistemas operacionais permitem que funções (ou classes, no caso de programas orientado a objetos) possam ser ligadas durante a execução do programa. Estas funções são armazenadas em **bibliotecas de ligação dinâmica (DLL)**.

Entretanto, a visão do software como artefato requer algo mais que programação. É preciso pensar no software considerando os requisitos dos usuários e a visão que ele terá deste produto. A interface com o usuário é um componente essencial do software junto com o modelo conceitual que determina a sua funcionalidade e a interatividade. (Para maiores detalhes sobre estes conceitos veja Conceitos Básicos. Eles também serão discutidos posteriormente no capítulo 4).

Com esta perspectiva temos que considerar que o desenvolvimento de software é algo mais do que programar. É pensar na aplicação do programa na resolução dos problemas dos usuário. É incorporar no programa elementos do domínio onde ele está inserido, representando-os como informações e oferecendo funções para que os usuários possam utilizá-las.

Além destas considerações, para que este desenvolvimento possa ser considerado uma engenharia outros fatores precisam ser considerados.

O que é engenharia?

O desenvolvimento de um artefato pode ser conduzido de forma **artesanal**, por um processo de *tentativa-e-erro,* manipulando-se diretamente o material com o qual o produto será construído.

Os erros são identificados através da **avaliação experimental** da qualidade do produto. A avaliação deve verificar se o produto está funcionando adequadamente, se ele é útil aos seus usuários, e várias outras qualidades.

Este processo artesanal pode evoluir através da utilização de **ferramentas** específicas e de **técnicas** desenvolvidas a partir de experiências anteriores.

Este processo pode ainda evoluir e incorporar outras atividades. O **design** (desenho ou projeto) consiste na atividade de conceber e descrever o produto a ser construído. O design permite uma visualização antecipada do produto final permitindo que se possa fazer alguma avaliação antes da sua construção.

Para que o produto tenha sucesso ele deve estar adequado às necessidades do usuário. Atividades de **análise de requisitos** devem anteceder o design e a construção do artefato para que estes objetivos sejam atingidos.

Com tantas atividades é preciso um **modelo do processo** que descreva em qual momento cada uma será realizada. Análise, especificação, design, implementação e avaliação devem ser realizadas no momento adequado. Para cada etapa do processo **métodos** devem descrever com detalhes como estas atividades devem ser realizadas.

Cada uma destas atividades requer pessoal com conhecimento especializado. Diversos **princípios** e **modelos teóricos** são conhecimentos indispensáveis para o desempenho das atividades do desenvolvimento.

Os especialistas nas diversas atividades devem atuar em **equipe**. Existem diversas maneiras de estruturar e gerenciar uma equipe de desenvolvimento.

A alocação de tarefas específicas que foram determinadas pelos métodos do modelo de desenvolvimento para os membros da equipe em um determinado período de tempo faz parte do processo de **planejamento**. O planejamento requer ainda que sejam **estimados os custos e prazos** de cada uma destas atividades. Tudo o que foi estimado e planejado deve ser **gerenciado** para que possa ser cumprido.

Quando o desenvolvimento de um artefato incorpora as atividades de análise, design, implementação e avaliação de acordo com princípios, modelos, técnicas, ferramentas e métodos específicos, realizados por uma equipe de especialistas e obedecendo a um planejamento e gerenciamento de custos e prazos, podemos caracterizá-lo como uma **engenharia**.

Veremos, a seguir, as características destas atividades no desenvolvimento de software.

Objetivos da Engenharia de Software

- A engenharia de software tem por objetivos a aplicação de teoria, modelos, formalismos e técnicas e ferramentas da ciência da computação e áreas afins para o **desenvolvimento** sistemático de software.
- Associado ao desenvolvimento, é preciso também aplicar métodos, técnicas e ferramentas para o **gerenciamento** do processo de desenvolvimento.
- Finalmente, a engenharia de software visa a produção da **documentação** formal do software, do processo de desenvolvimento e do gerenciamento destinada a comunicação entre os membros da equipe de desenvolvimento bem como aos usuários finais.

Definições de Engenharia de Software

Os autores apresentam diversas definições para engenharia de software. Vamos apresentar três que consideramos complementares.

- A engenharia de software é a disciplina envolvida com a produção e manutenção sistemática de software que são desenvolvidos com custos e prazos estimados.
- Disciplina que aborda a construção de software complexo - com muitas partes interconectadas e diferentes versões - por uma equipe de analistas, projetistas, programadores, gerentes, "testadores", etc.
- O estabelecimento e uso de princípios de engenharia para a produção economicamente viável de software de qualidade que funcione em máquinas reais.

A primeira destas definições enfatiza que a engenharia visa não apenas o desenvolvimento, mas também a manutenção do produto. Além disso, ela ressalta a importância da estimativa de custos e prazos de desenvolvimento. A segunda definição enfatiza a complexidade do produto e do processo. O software é formado por diversos componentes interconectados e o seu desenvolvimento é realizado por uma equipe que precisa ser gerenciada. A terceira ressalta que o desenvolvimento de software deve seguir os princípios de uma *engenharia* e deve visar a qualidade.

Mitos do Software

Segundo [Pressman], diversos mitos difundidos entre programadores escondem a importância de um desenvolvimento de software de acordo com os princípios de uma engenharia. Vejamos algumas delas:

- O estabelecimento de objetivos gerais é suficiente para se começar a escrever programas.
- Uma vez que o programa esteja escrito e funcionando, nosso trabalho está feito.
- Mudanças no software podem ser feitas facilmente porque ele é "flexível".
- Dê a uma pessoa técnica um bom livro de programação e você terá um programador.
- Até que o programa esteja "rodando" não é possível verificarmos a sua qualidade.
- Um projeto é bem sucedido se conseguirmos um programa funcionando corretamente.

Índice | Próximo

 Última atualização: 22/04/00

2. O Processo de Desenvolvimento de Software

Objetivos:

Este capítulo aborda as diversas maneiras de como um software deve ser desenvolvido. Veremos o conceito de *ciclo de vida*, identificando suas principais fases e as atividades do ciclo de vida do software. Finalizaremos com o estudo de diversas propostas e modelos para o processo de desenvolvimento de software.

2.1 Processo de desenvolvimento

Vimos na introdução que uma engenharia de software requer que as atividades para desenvolver o software sejam feitas de forma planejada, gerenciada, com pessoal capacitado, custos e prazos estimados e utilizando teorias, métodos, técnicas e ferramentas adequadas.

Elaborar um **processo de desenvolvimento de software** significa determinar de forma precisa e detalhada quem faz o que, quando e como. Um processo pode ser visto como uma instância de um método com suas técnicas e ferramentas associadas, elaborado durante a etapa de planejamento, no qual as atividades que o compõem foram alocadas aos membros da equipe de desenvolvimento, com prazos definidos e métricas para se avaliar como elas estão sendo realizadas (veja conceitos básicos).

Enquanto um método é algo teórico, o processo deve determinar ações práticas a serem realizadas pela equipe como prazos definidos. O processo é o resultado do planejamento e precisa ser gerenciado no decorrer de sua execução.

Não é objetivo deste capítulo a elaboração de processos de desenvolvimento. Apenas podemos fazê-lo após estudarmos técnicas de planejamento e gerenciamento. Neste capítulo vamos nos limitar a estudar alguns modelos de processo que nos indique como as diversas etapas e atividades do desenvolvimento podem ser estruturadas.

2.2 O ciclo de vida do software

O ciclo de vida de um artefato diz respeito às diversas fases pelas quais ele passa desde o seu surgimento até a o momento no qual ele não será mais útil. No sistema computacional todos os componentes possuem um ciclo de vida próprio. Embora eles sejam relacionados entre si, eles possuem ciclos de vida

independentes. O hardware pode continuar existindo mesmo que o software seja destruído. O software, em sua forma estática, pode também continuar existindo mesmo que um computador específico torne-se inoperante. Quando o software é construído para um hardware específico é que o ciclo de vida de ambos pode ser mesmo. Neste curso, vamos nos limitar a estudar e discutir o ciclo de vida do software.

No ciclo de vida do software identificamos três fases:

- **Definição**
- **Desenvolvimento**
- **Operação**

Na fase de **definição** os requisitos do software são determinados, a sua viabilidade é estudada e o planejamento das atividades é elaborado.

Na fase de **desenvolvimento** são realizadas as atividades destinadas a produção do software. Ela envolve atividades de concepção, especificação, design da interface, prototipação (do inglês *prototyping,* traduzido também por prototipagem), design da arquitetura, codificação e verificação, dentre outras.

Na fase de **operação** o sistema deverá efetivamente ser utilizado pelos seus usuários produzindo os resultados desejados. Nesta fase devem ocorrer as atividades de manutenção, seja para que se façam correções, ou seja para a sua evolução, isto é, para que o software satisfaça novos requisitos.

Nas próximas seções, vamos descrever algumas das principais atividades do ciclo de vida do software. O nosso objetivo é essencialmente didático. Vamos apresentá-las para que possamos compreender os diferentes modelos do processo que serão apresentados na próxima seção. Algumas destas atividades podem ou não aparecer em um determinado modelo, bem como pode existir alguma atividade especifica de um certo modelo que não será mencionada aqui.

Optamos por a apresentar as atividades do desenvolvimento classificando-as por fase do ciclo de vida. Entretanto, dependendo do modelo de processo algumas atividades também podem estender-se por mais de uma fase do ciclo de vida.

2.2.1 As atividades da fase de definição

Como normalmente o software está inserido num contexto mais amplo - o sistema - a fase de definição do software está inserida na definição do sistema. Definir o sistema é definir todos os seus componentes (ver capítulo 1).

Na fase de definição são tomadas as decisões de construir ou não o software. Nela são definidos os **requisitos do software**, determinando-se o que o cliente quer, o que a organização necessita, quais os problemas nas atividades dos usuários, etc. O cliente pode, por exemplo, necessitar de um software que faça o controle de vendas e compras da sua empresa. Um cliente de uma empresa de publicidade pode necessitar de um software de editoração eletrônica para a *Web*.

Também devem ser definidas algumas restrições ao software. Um exemplo de restrição técnica é: "o software deve ser executado no ambiente *Unix/XWindow*, uma vez que esta é a plataforma instalada na empresa". Outro caso seria: "o software deve ser executado num sistema remotamente distribuído uma vez que a empresa possui diversos pontos de venda". Existem restrições econômicas: "o orçamento de desenvolvimento não pode ultrapassar R$ 10.000,00".

A definição dos requisitos é denominada de **análise e especificação de requisitos** indicando que existe uma atividade de observação e uma descrição rigorosa dos problemas e da proposta de soluções. Alguns autores argumentam que devido à sua complexidade esta atividade deve ser considerada uma **engenharia de requisitos.** O objetivo deles também é ressaltar que os requisitos devem ser gerenciados durante todo o ciclo de vida do software.

O termo especificação está associado a diversas atividades do ciclo de vida: especificação de requisitos, especificação do software, especificação da arquitetura, especificação de dados e algoritmos, etc. e se refere a descrição, através de alguma notação, de algo que foi concebido ou idealizado. A especificação de requisitos tem por objetivo descrever aquilo que clientes ou futuros usuários necessitam e que será resolvido pela construção de um software. Como esta especificação precisa ser validada por clientes e usuários, normalmente ela é feita através de uma notação informal, como a linguagem natural, ou usando uma linguagem gráfica semi-formal como UML, DFD, DER, etc. Uma técnica bastante utilizada para especificação informal de requisitos são os **cenários** (ver capítulo 3).

O resultado desta atividade é a descrição dos **requisitos funcionais** - que dizem respeito àquilo que se quer que o software faça - e os **não-funcionais** - que dizem respeito a requisitos de ordem técnica, econômica, da organização, etc.

Nesta fase também deve haver um **estudo de viabilidade** do software. Este estudo visa verificar se o software é viável **técnica** e **economicamente**, e se os benefícios trazidos serão compensadores. O estudo de viabilidade requer que tenham sido definidos alguns requisitos para que se possa ter idéia do que será o sistema. Conforme veremos mais adiante, alguns modelos de processo podem determinar que o estudo de viabilidade seja feito apenas após a análise completa dos requisitos. Em outros, ele pode ser realizado simultaneamente ou num processo iterativo.

Associada a esta atividade devem ser realizadas **estimativas de custos e prazos** e a **análise de riscos**. A primeira visa determinar gastos e prazos aproximados a partir de dados de experiências anteriores. A análise de riscos visa verificar se existem possibilidades de que algo possa sair errado, como por exemplo, o orçamento estourar ou se haverá dificuldades técnicas.

O resultado do estudo de viabilidade é a decisão de que o software deve ou não ser construído com base nos requisitos, nas restrições técnicas, nas estimativas de custos e análise de riscos, dentre outros fatores, em relação aos benefícios que o sistema deverá proporcionar.

Também na fase de definição, caso o software seja viável, deve ser feito o **planejamento** de como o desenvolvimento será conduzido, isto é, deve-se elaborar um processo de desenvolvimento. O planejamento não necessariamente precisa ser feito completamente nesta fase. Veremos modelos do processo no qual o planejamento é revisto diversas vezes ao longo do desenvolvimento. O resultado do planejamento é uma descrição precisa do processo de desenvolvimento de software.

Existem diversas propostas sobre quem deve realizar as atividades na fase de definição. Na maior parte dos casos a definição dos requisitos iniciais é conduzida por **analistas de sistemas** como parte da definição do sistema. Isto significa que a fase de definição do software pode ser considerada como parte da análise de sistema que visa definir o sistema computacional do qual o software é parte. Para contribuir como definições mais específicas sobre o software, como estimativas de custos e prazos e o planejamento do desenvolvimento de software, **engenheiros de software** e analistas de sistemas devem trabalhar conjuntamente. Em diversas empresas analistas de sistemas realizam mais do que a definição dos requisitos do sistemas. Eles também definem e projetam o software.

2.2.2 Atividades da fase de desenvolvimento

Um **modelo do processo de desenvolvimento** ou **método** compreende algumas atividades. Cada proposta de modelo apresentado na literatura acadêmica ou elaborado por empresas de desenvolvimento devem determinar quais as atividades compõem o método. Embora estas atividades variem de uma proposta para outra, algumas são comuns a vários métodos. Além disso, em alguns modelos de processo, algumas atividades da fase de definição também podem estar inseridas nesta fase para que possam ser revisadas. Existem modelos que propõem ciclos cujas atividades da fase de desenvolvimento são alternadas com atividades da definição.

Design

Design é a atividade de concepção e especificação de um produto. A concepção é a atividade mental de criação do produto que satisfaça aos requisitos. Aquilo que foi concebido deve ser concretizado na forma de uma especificação. Muitos autores utilizam o termo especificação no sentido de design. Design pode ser traduzido como projeto ou como desenho. Consideramos que ambas não são traduções ideais e vamos optar por usar a palavra em inglês.

Diversos autores argumentam que se o software deve ser visto como um produto a atividade de design de software é imprescindível [Winograd 96]. Assim, **design de software** é atividade do desenvolvimento na qual o software deve ser concebido e especificado do ponto de vista do usuário e não do desenvolvedor. O foco está na visão externa do software que é a aquela que será percebida pelos usuários. Fazendo uma analogia com a engenharia civil, o design de software seria uma atividade equivalente à arquitetura de uma edificação.

O design deve determinar o que o software deve fazer - a sua **funcionalidade** - e como o usuário irá interagir como ele - a sua **interatividade** ou modelo de interação. Estes dois elementos compõem o **modelo conceitual da aplicação**. Desta forma o design de software envolve a concepção e especificação da funcionalidade e do modelo de interação.

O design toma como base a especificação de requisitos elaborada na fase definição. É importante ressaltar que, dependendo do processo de desenvolvimento, a especificação dos requisitos pode estender-se pela fase de desenvolvimento, como veremos mais adiante. Enquanto que a especificação dos requisitos visa descrever o que clientes, usuários e organização necessitam, o design de software visa especificar o que ele oferecerá para satisfazer estas necessidades.

Existem decisões importantes que são tomadas durante o design de software que fogem ao escopo de uma especificação de requisitos. Por exemplo, para o requisito funcional de "o software deve realizar o cálculo do total de vendas e do lucro obtido", o designer do software deve decidir os dois cálculos serão feitos simultaneamente por uma única função ou se por duas funções independentes. Ele deve decidir se o usuário fornece os dados todos inicialmente e apenas ao final os cálculos são feitos ou se os dados são fornecidos para cada cálculo que se deseja fazer.

Uma vez que o designer tenha especificado a funcionalidade e o modelo de interação ele pode apresentar aos clientes e usuários para as suas opiniões ou rever os requisitos. Quando os requisitos são refinados a partir da especificação do software as atividades de especificação de requisitos e de software se confundem. Nestas condições nas quais o design é realizado com a participação dos clientes e usuários ele é chamado de **design participativo**.

A engenharia de software tradicionalmente enfatiza apenas na especificação da funcionalidade. A proposta de design centrado no usuário visa chamar a atenção para o modelo de interação e para a interface de usuário como elementos importantes na qualidade de um software.

O **design da interface de usuário** visa a concepção e especificação da parte do software que possibilita que o usuário interaja com o sistema de acordo com o modelo de interação especificado. Enquanto o modelo de interação determina os protocolos de comunicação entre o usuário e o sistema, a interface deve apresentar os menus, janelas, ícones, botões que permitem as ações do usuário de acordo com o modelo. O design da interface é a concretização de um modelo de interação especificado durante o design do software.

A maioria dos autores de engenharia de software utilizam o termo de design do software para se referir à especificação da arquitetura de software, isto é, da configuração de componentes do software - funções, classes, objetos e suas interconexões. Neste caso, estamos realizando design do ponto de vista do desenvolvedor e chamaremos de **design da arquitetura do software**. O design da arquitetura visa determinar de maneira abstrata como a funcionalidade será implementada.

O design da arquitetura deve resultar na especificação da abstrata da macro-estrutura dos componentes do software e de como eles interagem entre si, sem preocupação com detalhes a respeito dos algoritmos que descreverão o funcionamento de cada componente. No design arquitetural é importante que se especifique o que cada componente deve fazer - **especificação funcional de componentes**.

O **design de algoritmos e dados** , também chamado de design detalhado, tem por objetivo a concepção e especificação das estruturas de dados e dos algoritmos que realizam aquilo que foi especificado para cada componente do software. A especificação funcional dos componentes deve se feita a nível do que chamamos de *problemas algorítmicos*, isto é que podem ser resolvidos pela construção de um algoritmo. As soluções algorítmicas são normalmente encontradas na literatura especializada em design de algoritmos e de estruturas de dados.

Em alguns casos, pode-se encontrar componentes de software já desenvolvidos em código fonte ou executável - em bibliotecas, por exemplo - e incorporá-los durante a programação.

O resultado de todas as atividades de design são especificações da funcionalidade, do modelo de interação, da interface de usuário, da arquitetura de software e dos algoritmos e estruturas de dados.

Prototipação

Uma outra forma de concretizar a concepção de um software é a através de um protótipo. Um protótipo é um produto construído utilizando materiais mais baratos e com dimensões reduzidas. A **prototipação** (ou prototipagem) é atividade de construir protótipos. Existem modelos de processo de desenvolvimento que são baseados em prototipação como veremos a seguir.

Um protótipo do software é construído utilizando ferramentas que permitem que apenas partes do software sejam construídas com o objetivo de verificar suas qualidades antes que o produto final venha a ser construídos.

A prototipação, uma técnica usada freqüentemente nas engenharias, tem sido adotada na engenharia de software para aperfeiçoar as previsões e diminuir os riscos envolvidos no desenvolvimento de um novo projeto. Os termos protótipos e prototipação têm sido usados na literatura com diversos significados e não se definiu uma classificação ou critérios para tal, de maneira convincente.

Programação

A programação consiste na atividade de construção de um programa que implementa uma determinada solução para um problema algorítmico. Esta solução pode ser descrita na forma de especificação de algoritmos e estruturas de dados e deve então ser codificada uma linguagem de programação. Especialistas em programação podem escrever a solução algorítmica diretamente na linguagem de programação.

Esta atividade é também chamada de implementação, construção ou codificação do software. A codificação enfatiza a que o software deve ser construído utilizando uma linguagem de programação. Além da codificação propriamente, a programação deve envolver ainda a sua tradução num código de máquina executável.

A programação deve resultar num programa que implemente tudo o que foi especificado durante o design. No caso de software formado por vários componentes, estes devem ser implementados a partir da especificação de cada componente e **integrados** (interconectados) conforme a arquitetura interna do software.

Avaliação ou Verificação

A avaliação ou verificação visa assegurar algumas das principais qualidades do software. Dentre as atividades de avaliação vamos destacar a correção, validação e usabilidade do software.

O software é considerado correto quando o programa implementado satisfaz à sua especificação. Existem diversas formas de verificarmos se um software está correto. Elas são o **teste de programa**, a **inspeção** e a **prova de programas**.

O **teste de programa** permite identificarmos se um programa tem erros a partir da execução do programa de acordo com técnicas específicas.

A **inspeção** pode ser feita pela análise do código fonte e a sua execução mental com o auxílio de lápis e papel (**rastreamento**). Outra forma de inspeção é utilizando o *debbuger* que permite acompanhar passo a passo a execução de cada instrução do programa e verificar o estado das variáveis de dados.

A **prova de programa** utiliza técnicas de lógica matemática que permite provar se um programa está correto em relação à especificação formal deste programa. Em um software complexo, se a sua arquitetura for especificada formalmente pode-se provar que cada componente individualmente está correto e que a sua integração também está correta.

A avaliação da **validação** do software visa determinar se a especificação do software (funcionalidade, arquitetura, interface, etc) satisfaz aos requisitos do usuário. Ela deve ser realizada durante toda a fase de desenvolvimento na medida que as especificações forem elaboradas.

A **avaliação da usabilidade** visa identificar as qualidades relacionadas com a interação entre o usuário e o software. Dentre estas qualidades estão a facilidade de aprendizado, facilidade de uso, produtividade, etc. A usabilidade é verificada de forma empírica através de **testes de usabilidade** que analisam o comportamento do usuário durante a interação.

Outras propriedades do software também precisam ser verificadas tais como **desempenho** e **robustez** e devem ser estudadas na literatura especializada.

2.2.3 As atividades da fase de operação

Durante a fase de operação do ciclo de vida o software deve ser instalado, utilizado e feita a sua manutenção.

A **instalação** e **configuração** é a atividade visa implantar o software no computador para que ele possa ser utilizado pelos usuários. A instalação requer que o software seja armazenado no lugar correto e que os *drivers* de dispositivos de hardware sejam configurados adequadamente para que o software funcione corretamente. Esta atividade pode ser realizada tanto por usuários como pelos engenheiros ou programadores do software.

A **utilização** é a atividade fim do software, uma vez que ele foi construído para auxiliar pessoas na realização de suas tarefas. A utilização deve seguir o modelo de interação especificado durante o seu design que deve está descrito no **manual de usuário**.

A **manutenção** é a atividade destinada a assegurar a qualidade do software durante a fase de operação. A manutenção se faz necessária para que o software possa ser transportado para outras plataformas de hardware e de sistemas operacionais, para satisfazer a novos requisitos dos usuários ou para corrigir eventuais erros. Ela pode envolver a **correção** de erros não detectados durante os testes, o **re-design** da funcionalidade, da interface de usuário, da arquitetura do software ou dos algoritmos de forma que novos requisitos sejam satisfeitos, e várias outras atividades.

2.3 Modelos do processo de desenvolvimento

Um modelo para um processo de desenvolvimento é uma proposta teórica que junto com o planejamento deve determinar quais atividades devem ser realizadas, quando, como e por quem. Nesta seção vamos apresentar alguns dos mais conhecidos modelos do processo de desenvolvimento. Eles são o modelo **Cascata**, o modelo **Evolutivo** ou Incremental, o modelo **Espiral** e o modelo **Transformação**.

A escolha de um destes modelos envolve diversos fatores. Um deles é o tipo de software - se é um software de banco de dados, um software de tempo-real, um software embutido, um sistema especialista. Outro fator importante é se a equipe de desenvolvimento é uma empresa de desenvolvimento (software house) ou se é a equipe de engenheiros da própria organização que utilizará o produto. A maneira como o produto será vendido e instalado é um outro fator importante - se o software é para ser vendido para um público amplo ou se será instalado em uma única empresa, sob encomenda.

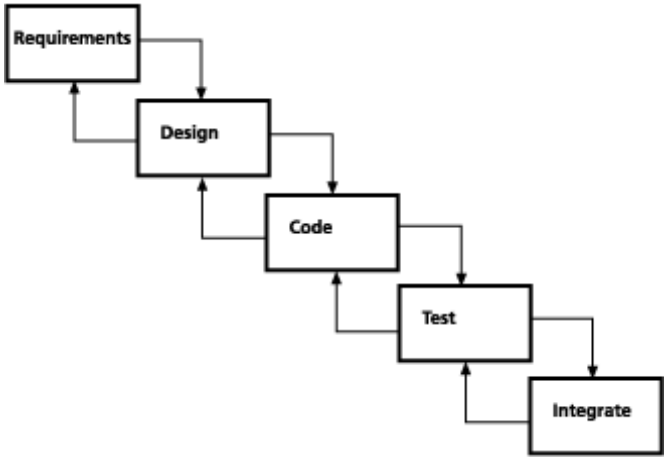
Para cada uma das atividades **métodos** específicos devem ser elaborados. Um conjunto de métodos relacionados entre si num esquema comum de acordo com um modelo é chamado de **metodologia** de desenvolvimento.

O Modelo Cascata

O modelo cascata tornou-se conhecido na década de 70 e é referenciado na maioria dos livros de engenharia de software ou manuais de padrões de software. Nele as atividades do processo de desenvolvimento são estruturadas numa cascata onde a saída de uma é a entrada para a próxima. As suas principais atividades são:

- estudo de viabilidade
- análise e especificação de requisitos
- design e especificação
- codificação e testes de componentes
- teste do sistema e integração
- entrega e instalação
- manutenção

Existem muitas variantes deste modelo propostas por diferentes pesquisadores ou empresas de desenvolvimento e adaptadas a diferentes tipos de software. A característica comum é um fluxo linear e seqüencial de atividades semelhantes a descritas anteriormente. A figura a seguir, mostra uma variante do modelo em cascata.



Este modelo, quando proposto, introduziu importantes qualidades ao desenvolvimento. A primeira chama a atenção de que o processo de desenvolvimento deve ser conduzido de forma disciplinada, com atividades claramente definidas, determinada a partir de um planejamento e sujeitas a gerenciamento durante a realização. Outra qualidade define de maneira clara quais são estas atividades e quais os requisitos para desempenhá-las. Por fim, o modelo introduz a separação das atividades da definição e design da atividade de programação que era o centro das atenções no desenvolvimento de software.

O modelo Cascata também é criticado por ser linear, rígido e monolítico. Inspirados em modelos de outras atividades de engenharia, este modelo argumenta que cada atividade apenas deve ser iniciada quando a outra estiver terminada e verificada. Ele é considerado monolítico por não introduzir a participação de clientes

e usuário durante as atividades do desenvolvimento, mas apenas o software ter sido implementado e entregue. Não existe como o cliente verificar antecipadamente qual o produto final para detectar eventuais problemas.

Características particulares do software (ser conceitual, por exemplo) e a falta de modelos teóricos, técnicas e ferramentas adequadas mostram que é necessário haver maior flexibilidade neste fluxo seqüencial, permitindo volta atrás para eventuais modificações. Veremos mais adiante modelos que propõem maior flexibilidade no fluxo de execução.

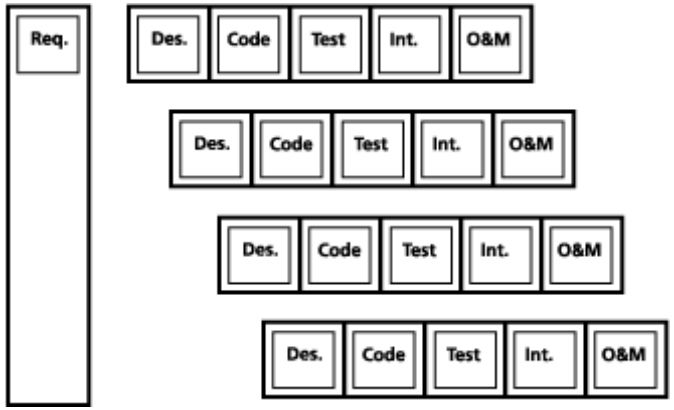
As métricas utilizadas nas estimativas de prazos e recursos humanos são ainda bastante imprecisas e quase sempre o planejamento de atividades precisa ser revisto. Normalmente os prazos não são cumpridos pois o planejamento, neste modelo, é feito nas etapas iniciais do desenvolvimento. A estrutura seqüencial e rígida também não permite que o planejamento seja refeito para corrigir falhas nas atividades de desenvolvimento.

O Modelo Evolutivo

O modelo evolutivo descreve um processo na qual o software deve ser desenvolvido de forma a evoluir a partir de protótipos iniciais. Para entender melhor este modelo é importante entender o que é prototipação. A prototipação também aparece em outros modelo de processo.

O processo de desenvolvimento pode evoluir de diversas maneiras. Uma destas formas consiste em ir desenvolvendo partes funcionais de um software, isto é, que funcionam, mas não atendem a todos os requisitos por completo, e ir incrementando com novos pedaços até que todos os requisitos sejam atendidos. Para cada novo componente incorporado, as qualidades do protótipo devem ser verificadas experimentalmente.

Outra forma de evolução é iniciar o desenvolvimento com um protótipo rudimentar não funcional e torná-lo funcional ao longo do processo através do refinamento de seus componentes.



O fluxo de atividades do modelo evolutivo caracteriza-se por ser cíclico ou iterativo. Ele começa com o design e desenvolvimento de um protótipo inicial, que deve ser mostrado aos usuários e avaliado. Durante a avaliação novos requisitos são definidos e alterações e incrementos ao protótipo inicial devem ser feitas.

Este ciclo deve ser repetido em direção ao produto final.

A grande vantagem deste modelo está em permitir a verificação antecipada do produto final por engenheiros, clientes e usuários, permitindo a correção dos problemas detectados.

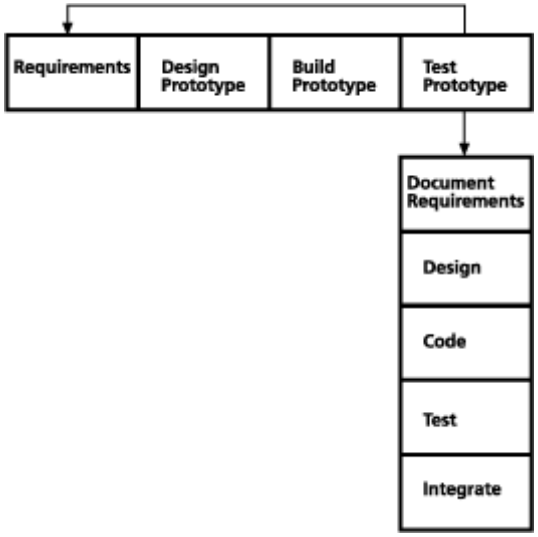
A extrema flexibilidade deste modelo e a falta de rigor leva a software que embora satisfaça aos requisitos dos usuários têm deficiências de desempenho, portabilidade, manutenção e outras qualidades internas.

Embora a prototipação tenha enormes vantagens e deva ser incentivada, basear o desenvolvimento no incremento de protótipos pode levar a software mal documentados e com arquiteturas mal definidas. Como os requisitos estão sempre sendo revistos a cada ciclo de desenvolvimento, torna-se praticamente impossível estimar custos e prazos e planejar as atividades de desenvolvimento.

A prototipação é mais adequada como método auxiliar à análise de requisitos e ao design do software, pois permite a validação antecipada do produto. Entretanto, não se deve deixar de lado o design da arquitetura de software e dos algoritmos para que ele possa ser construído utilizando uma linguagem de programação mais adequada.

Prototipação

Prototipação é uma abordagem baseada numa visão evolutiva do desenvolvimento de software, afetando o processo como um todo. Esta abordagem envolve a produção de versões iniciais - "protótipos" - de um sistema futuro com o qual pode-se realizar verificações e experimentações para se avaliar algumas de suas qualidades antes que o sistema venha realmente a ser construído.



Tipos de Protótipos

O relacionamento entre um protótipo e as atividades do processo de desenvolvimento - início do projeto e análise de requisitos, design da interface e da aplicação, e implementação - permite a identificação de quatro tipos de protótipos:

- **Protótipo de Apresentação** - oferece suporte ao início do projeto e é usado para convencer o cliente de que o futuro sistema é viável e que a interface do usuário se adequa aos requisitos. Na maioria dos casos é usado para mostrar visão que o usuário têm do sistema e revelar aspectos importantes da interface.
- **Protótipo Autêntico** - é um sistema de software provisório e funcional, geralmente projetado para ilustrar aspectos específicos da interface de usuários ou parte da funcionalidade, ajudando na compreensão dos problemas envolvidos.
- **Protótipo Funcional** -- é derivado do modelo do domínio do problema ou da especificação do software e serve para ajudar à equipe de desenvolvimento compreender questões relacionadas com a construção do sistema. Esse protótipo não interessa aos usuários.
- **Sistema Piloto** - é usado não apenas com propósitos ilustrativos, mas como um núcleo básico operacional do sistema. Esse sistema deve ser instalado no ambiente de aplicação e experimentado com os usuários.

Objetivos da Prototipação

Num projeto de software várias questões podem ser respondida com a construcão de protótipos. Nas situações típicas de desenvolvimento podemos distinguir entre diferentes objetivos na prototipação:

- **Exploratória** - é quando o protótipo é usado para ajudar a esclarecer requisitos dos usuários com respeito ao sistema futuro. Uma prototipação também é exploratória quando se busca examinar uma variedade de opções de design de maneira a evitar a escolha de uma abordagem específica não adequada. Com esses objetivos os desenvolvedores adquirem informações sobre o domínio, os usuário e tarefas. Os usuários podem emitir informações e sugestões mais precisas, tornando-se parceiro das decisões que envolvem o desenvolvimento.

- **Experimental** - é quando a prototipação foca aspectos técnicos do desenvolvimento, oferecendo aos desenvolvedores resultados experimentais para tomada de decisões de design e implementação. Um aspecto essencial é a viabilização de uma base de comunicação entre os usuários e desenvolvedores para soluções de problemas técnicos de viabilidade e usabilidade, dentre outros. As principais vantagens para os desenvolvedores são a verificação e validação das decisões tomadas e soluções apresentadas.

- **Evolutiva** - A prototipação pode ser aplicada de maneira bastante proveitosa num processo de reengenharia em organizações, para avaliar o impacto que a introdução de novas tecnologias pode trazer. Nesse caso o protótipo não é visto apenas como uma ferramenta em projetos individuais, mas como parte de um processo contínuo de evolução dos processos organizacionais. Os desenvolvedores não são mais os protagonistas da prototipação, mas consultores que trabalham em cooperação com os usuários no processo de reengenharia.

Técnicas de prototipação

Do ponto de vista técnico, a prototipação pode ser conduzida segundo duas abordagens, relacionadas com técnicas específicas de construção. Experiências em desenvolvimento de software têm apontado inúmeras qualidades no design e implementação em diversas camadas. Essas camadas podem ir desde a interface de

usuário à camadas mais básicas como uma base de dados e o sistema operacional. Dessa forma, podemos identificar duas formar de subdividir a prototipação.

- Prototipação horizontal - Apenas uma camada específica do sistema é construída. Por exemplo, a interface do usuário com suas janelas e *widgets*, ou camadas da aplicação como as funções para transação em bancos de dados.
- Prototipação vertical - Uma parte da funcionalidade do sistema é escolhida para ser implementada completamente. Esta técnica é apropriada quando aspectos da funcionalidade ainda estão em aberto.

Um outro critério para se classificar protótipo é de acordo com o relacionamento entre o protótipo e o sistema final. Em certos casos, o protótipo serve apenas para propósitos de especificação do sistema final e não será usado como parte integrante do sistema final. Ele é jogado fora. Um outro caso, é quando o protótipo vai sendo incrementado e se torna parte integrante (*building block*) do sistema. Por fim, o protótipo pode ser construído apenas para a compreensão de certos problemas que envolvem determinados interesses. Não existe a intenção de transformá-lo num sistema.

O Modelo Transformação

Um programa é uma descrição formal, isto é, ele é descrito por uma linguagem de programação cuja sintaxe e semântica são definidos formalmente. Apenas desta forma é que temos a garantia de que o programa será sempre executado da mesma forma pelo computador.

O modelo Tranformação tem suas raízes na abordagem de métodos formais para o desenvolvimento de software. A idéia é que o desenvolvimento deve ser visto como uma seqüência de passos que gradualmente transforma uma especificação formal num programa. O passo inicial é transformar os requisitos informais numa especificação funcional formal. Esta descrição formal é então transformada numa outra descrição formal mais detalhada, e assim, sucessivamente, até chegar no ponto em que se tenha componentes de programas que satisfaçam a especificação. Durante este processo de transformações sucessivas - chamado de *otimização* - as especifição formais produzidas podem ser executadas por um processador abstrato, permitindo uma verificação formal da sua validação antes mesmo que o programa venha a ser implementado. Antes de serem transformadas cada especificação formal deve ser verificada em relação às expectativas dos clientes e usuários.

As transformações devem ser realizadas pelo engenheiro de software. A natureza formal da transformação possibilitam a aplicação de verificações matemáticas como prova, consistência e completude da especificação. As transformações podem ser realizadas de maneira automática por ferramentas que traduzem especificações formais mais abstratas em especificações mais detalhadas.

Este modelo prevê que o engenheiro de software deve ter a sua disposição uma biblioteca de componentes reutilizáveis que satisfaça especificações formais. Na *otimização*, à medida que as especificações de mais baixo nível vão sendo obtidas verifica-se quais componentes da biblioteca implementam parte desta especificação. Um ambiente automatizado de desenvolvimento (uma ferramenta CASE - Computer Aided Software Engineering) pode auxiliar este processo armazenando o histórico de decisões dos engenheiros de software.

O Modelo Espiral

O objetivo do modelo espiral é prover um *metamodelo* que pode acomodar diversos processos específicos. Isto significa que podemos encaixar nele as principais características dos modelos vistos anteriormente, adaptando-os a necessidades específicas de desenvolvedores ou às particularidades do software a ser desenvolvido. Este modelo prevê prototipação, desenvolvimento evolutivo e cíclico, e as principais atividades do modelo cascata.

Sua principal inovação é guiar o processo de desenvolvimento gerado a partir deste metamodelo com base em **análise de riscos** e um **planejamento** que é realizado durante toda a evolução do desenvolvimento. *Riscos* são circunstâncias adversas que podem surgir durante o desenvolvimento de software impedindo o processo ou diminuindo a qualidade do produto. São exemplos de riscos: pessoas que abandonam a equipe de desenvolvimento, ferramentas que não podem ser utilizadas, falha em equipamentos usados no desenvolvimento ou que serão utilizados no produto final, etc. A identificação e o gerenciamento de riscos é hoje uma atividade importantíssima no desenvolvimento de software devido à imaturidade da área e à falta de conhecimento, técnicas e ferramentas adequadas.

O modelo espiral descreve um fluxo de atividades cíclico e evolutivo constituído de quatro estágios.

- No estágio 1 devem ser determinados objetivos, soluções alternativas e restrições.
- No estágio 2, devem ser analisados os riscos das decisões do estágio anterior. Durante este estágio podem ser construídos protótipos ou realizar-se simulações do software.
- O estágio 3 consiste nas atividades da fase de desenvolvimento, incluindo design, especificação, codificação e verificação. A principal característica é que a cada especificação que vai surgindo a cada ciclo - especificação de requisitos, do produto, da arquitetura, da interface de usuário e dos algoritmos e dados - deve ser feita a verificação apropriadamente.
- O estágio 4 compreende a revisão das etapas anteriores o planejamento da próxima fase. Neste planejamento, dependendo dos resultados obtidos nos estágios anteriores - decisões, análise de riscos e verificação, pode-se optar por seguir o desenvolvimento num modelo Cascata (linear), Evolutivo ou Transformação. Por exemplo, se já no primeiro ciclo, os requisitos forem completamente especificados e validados pode-se optar por seguir o modelo Cascata. Caso contrário, pode-se optar pela construção de novos protótipos, incrementando-o, avaliando novos riscos e re-planejando o processo.

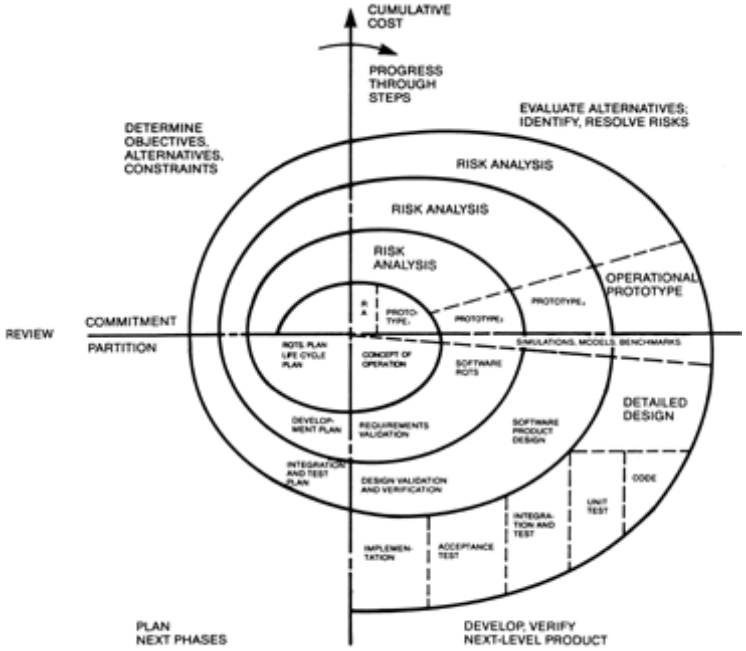


Figure 5. The Spiral model.

Anterior | Índice | Próximo

 Última atualização: 08/03/00

3. Análise e Especificação de Requisitos

Os objetivos deste capítulo são:

- Definir o que são requisitos de software
- Introduzir os objetivos da Engenharia de Requisitos
- Apresentar técnicas de comunicação para obter informações dos clientes e usuários
- Apresentar técnicas para descrever o domínio, usuários e tarefas.
- Especificar requisitos funcionais utilizando Casos de Uso
- Especificar requisitos não funcionais

3.1 Introdução

Vimos que o software é parte de um sistema computacional mais abrangente e que a Análise de Sistemas é a atividade de identificar os problemas do domínio, apresentar alternativas de soluções e o estudo da viabilidade de cada uma delas. Uma vez que se tenha feito a análise do sistema computacional, e delimitado o escopo do software, os requisitos do software devem ser analisados e especificados.

A **análise e especificação de requisitos de software** envolve as atividades de determinar os objetivos de um software e as restrições associadas a ele. Ela deve também estabelecer o relacionamento entre estes objetivos e restrições e a especificação precisa do software.

A análise e especificação dos requisitos de software deve ser vista como uma sub-atividade da análise de sistemas. Normalmente ela é iniciada juntamente com a análise do sistema, podendo se estender após a elaboração do documento de especificação do sistema e do planejamento do desenvolvimento, quando serão refinados os requisitos do software.

Análise e especificação são atividades inter-dependentes e devem ser realizadas conjuntamente. A **análise** é o processo de observação e levantamento dos elementos do domínio no qual o sistema será introduzido. Deve-se identificar as pessoas, atividades, informações do domínio para que se possa decidir o que deverá ser informatizado ou não. Pessoas e as atividades que não serão informatizadas deverão ser consideradas entidades externas ao software.

A **especificação** é a descrição sistemática e abstrata do que o software deve fazer, a partir daquilo que foi analisado. Ela apresenta a solução de como os problemas levantados na análise serão resolvidos pelo software do sistema computacional. Visa descrever de maneira sistemática quais as propriedades

funcionais são necessárias para resolver o problema do domínio. A especificação é também a forma de comunicação sistemática entre analistas e projetistas do software.

O objetivo da definição dos requisitos é especificar o que o sistema deverá fazer e determinar os critérios de validação que serão utilizados para que se possa avaliar se o sistema cumpre o que foi definido.

3.2 O que são requisitos?

Como sistemas computacionais são construídos para terem utilidade no mundo real. Modelar uma parte do mundo real, o domínio de aplicação é uma atividade extremamente importante para se compreender a necessidade e a importância do sistema a ser construído e definir os requisitos que tornam o sistema útil.

Requisitos são objetivos ou restrições estabelecidas por clientes e usuários do sistema que definem as diversas propriedades do sistema. Os requisitos de software são, obviamente, aqueles dentre os requisitos de sistema que dizem respeito a propriedades do software.

Um conjunto de requisitos pode ser definido como uma condição ou capacidade necessária que o software deve possuir (1) para que o usuário possa resolver um problema ou atingir um objetivo ou (2) para atender as necessidades ou restrições da organização ou dos outros componentes do sistema.

Tradicionalmente, os requisitos de software são separados em requisitos funcionais e não-funcionais. Os **requisitos funcionais** são a descrição das diversas funções que clientes e usuários querem ou precisam que o software ofereça. Eles definem a funcionalidade desejada do software. O termo função é usado no sentido genérico de operação que pode ser realizada pelo sistema, seja através comandos dos usuários ou seja pela ocorrência de eventos internos ou externos ao sistema.

São exemplos de requisitos funcionais:

- "o software deve possibilitar o cálculo dos gastos diários, semanais, mensais e anuais com pessoal".
- "o software deve emitir relatórios de compras a cada quinze dias"
- "os usuários devem poder obter o número de aprovações, reprovações e trancamentos em todas as disciplinas por um determinado período de tempo.

A especificação de um requisito funcional deve determinar o que se espera que o software faça, sem a preocupação de como ele faz. É importante diferenciar a atividade de especificar requisitos da atividade de especificação que ocorre durante o design do software. No design do software deve-se tomar a decisão de quais a funções o sistema efetivamente terá para satisfazer àquilo que os usuários querem.

Requisitos não-funcionais são as qualidades globais de um software, como manutenibilidade, usabilidade, desempenho, custos e várias outras. Normalmente estes requisitos são descritos de maneira informal, de maneira controversa (por exemplo, o gerente quer segurança mas os usuários querem facilidade de uso) e são difíceis de validar.

São exemplos de requisitos não-funcionais:

- "a base de dados deve ser protegida para acesso apenas de usuários autorizados".
- "o tempo de resposta do sistema não deve ultrapassar 30 segundo".
- "o software deve ser operacionalizado no sistema Linux"
- "o tempo de desenvolvimento não deve ultrapassar seis meses".

A necessidade de se estabelecer os requisitos de forma precisa é crítica na medida que o tamanho e a complexidade do software aumentam. Os requisitos exercem influência uns sobre os outros. Por exemplo, o requisito de que o software deve ter grande *portabilidade* (por exemplo, ser implementado em Java) pode implicar em que o requisito *desempenho* não seja satisfeito (programas em Java são, em geral, mais lentos).

3.3 A Engenharia de Requisitos

Os diversos relacionamento e restrições que os requisitos exercem uns sobre os outros são muito difíceis de ser controlados. Principalmente se considerarmos que algumas decisões de design que afetam um ou mais requisitos só serão tomadas mais adiante do desenvolvimento. Por exemplo, a decisão de implementar em Java pode ser decidida apenas após o design da arquitetura. Desta forma, os requisitos precisam ser gerenciados durante todo o desenvolvimento.

A importância e complexidade desta atividade levou ao surgimento, no início dos anos 90, da **Engenharia de Requisitos.** O objetivo desta denominação é ressaltar que o processo de definir os requisitos de software é uma atividade extremamente importante e independente das outras atividades da engenharia de software. Ela requer fundamentação e processos próprios, e que devem ser planejados e gerenciados ao longo de todo o ciclo de vida.

Os objetivos da Engenharia de Requisitos podem ser categorizados em três grupos principais [Zave]:

- Investigação de objetivos, funções e restrições de uma aplicação de software
  - Ultrapassar as barreiras de comunicação
  - Gerar estratégias para converter objetivos vagos em propriedades ou restrições específicas
  - Compreender prioridades e graus de satisfação
  - Associar requisitos com os vários agentes do domínio
  - Estimar custos, riscos e cronogramas
  - Assegurar completude
- Especificação do software
  - Integrar representações e visões múltiplas
  - Avaliar estratégias de soluções alternativas que satisfaçam os requisitos
  - Obter uma especificação completa, consistente e não ambígua
  - Validar a especificação - verificar que ela satisfaz aos requisitos
  - Obter especificações que sejam apropriadas para as atividades de design e implementação
- Gerenciamento da evolução e das famílias do software
  - Reutilização de requisitos durante o processo de evolução
  - Reutilização de requisitos no desenvolvimento de software similares

    - Reconstrução de requisitos

A Engenharia de Requisitos deve envolver a documentação das funções, do desempenho, interfaces externas e internas e atributos de qualidade do Software.

Esta área é inerentemente abrangente, interdisciplinar e aberta. Ela lida com a descrição de observações do mundo real através de notações apropriadas.

Os benefícios da Engenharia de Requisitos são:

- Concordância entre desenvolvedores, clientes e usuário sobre o trabalho a ser feito e quais os critérios de aceitação do sistema.
- Uma base precisa para a estimativa dos recursos (custo, pessoal, prazos, ferramentas e equipamentos)
- Melhoria na usabilidade, manutenibilidade e outras qualidades do sistema.
- Atingir os objetivos com o mínimo de desperdício

Uma boa especificação de requisitos deve ser:

- Clara e não-ambígua
- Completa
- Correta
- Compreensível
- Consistente
- Concisa
- Confiável

(Veja mais em Dorfman, Merlin - Requirements Engineering)

3.4 Modelos de documentos de especificação de requisitos

O resultado final da análise e especificação de requisitos e das outras atividades da fase de defição devem ser apresentados aos clientes para que eles possam validá-lo. Este documento oferece a concordância entre clientes, analistas e desenvolvedores sobre o que deve ser desenvolvido. É utilizando este documento que as atividades da fase de desenvolvimento (design e programação) serão validadas.

Cada desenvolvedor utiliza um modelo específico para elaborar este documento. A sua estrutura muitas vezes depende do método que está sendo utilizado. Em linhas gerais este modelo deve ser basicamente textual, utilizando o mínimo de termos técnicos, e ilustrados como modelos gráficos que demonstrem mais claramente a visão que os analistas tiveram dos problemas e dos requisitos para o futuro sistema.

Vamos apresentar a seguir dois modelos de documentos encontrados na literatura. Estes modelos descrevem não apenas a especificação dos requisitos, mas os resultados do estudo de viabilidade e o processo de desenvolvimento.

Pressman apresenta o seguinte documento de especificação de requisitos de software:

I. Introdução

1. Referências do Sistema
2. Descrição Geral
3. Restrições de projeto do software

II. Descrição da Informação

1. Representação do fluxo de informação
    a. Fluxo de Dados
    b. Fluxo de Controle
2. Representação do conteúdo de informação
3. Descrição da interface com o sistema

III Descrição Funcional

1. Divisão funcional em partições
2. Descrição funcional
    a. Narativas
    b. Restrições/limitações
    c. Exigências de desempenho
    d. Restrições de projeto
    e. Diagramas de apoio
3. Descrição do controle
    a. Especificação do controle
    b. Restrições de projeto

IV. Descrição Comportamental

1. Estados do Sistema
2. Eventos e ações

V. Critérios de Validação

1. Limites de desempenho
2. Classes de testes

3. Reação esperada do software
4. Considerações especiais

VI. Bibliografia
VII Apêndice

Uma proposta alternativa é a de Farley. De acordo com ele, um documento de especificação de requisitos deve possui as seguintes seções:

- Visão geral do produto e Sumário
- Ambientes de desenvolvimento, operação e manutenção
- Interfaces Externas e Fluxo de Dados
- Requisitos Funcionais
- Requisitos de Desempenho
- Tratamento de Exceções
- Prioridades de Implementação
- Antecipação de mudanças e extensões
- Dicas e diretrizes de Design
- Critérios de aceitação
- Índice Remissivo
- Glossário

3.5 Técnicas de comunicação com clientes e usuários

Um dos objetivos da Engenharia de Requisitos é ultrapassar barreiras de comunicação entre os clientes e usuários e os analistas para que os requisitos possam capturados e modelados corretamente

Dentre as técnicas mais importantes para a comunicação podemos citar três:

- **Entrevistas**
- **Observação *in loco***
- **Encontros**

Estas três técnicas são complementares e podem todas ser usadas numa mesma análise de requisitos. A **entrevista** é normalmente a primeira técnica utilizada. Analistas entrevistam clientes para definir os objetivos gerais e restrições que o software deverá ter. A entrevista deve ser feita de forma objetiva visando obter o máximo de informações do cliente. Diversas seções de entrevistas podem ser marcadas.

Na **observação *in loco*** os analista devem estar inseridos na rotina de trabalho da organização tentando entender e descrever as principais atividades que são realizadas. Na observação devem ser identificadas quais as atividades podem ser automatizadas, quem são os potenciais usuários, quais tarefas eles querem realizar com a ajuda do novo sistema, etc. A observação deve ser complementada com entrevistas específicas com os futuros usuários.

Os **encontros** são reuniões envolvendo analistas, clientes e usuários destinadas exclusivamente ao levantamento de informações, descrição dos problemas atuais e de metas futuras. Os encontros devem ser realizados em um local neutro (fora da organização) e normalmente duram poucos alguns dias. Nos encontros as informações que vão sendo levantadas vão sendo afixadas em paineis na sala de encontro para que possam ser analisadas e validadas pelos clientes e usuários. Ao final do encontro os analistas devem elaborar um relatório descrevendo os requisitos analisados.

3.6 Usando Cenários para Análise do Domínio

Do ponto de vista de usabilidade, o desenvolvimento de um novo sistema implica na transformação das tarefas e práticas atuais dos usuários e da organização. Conhecer a situação atual e antecipar o impacto que um novo sistema deve ter é fundamental para o seu sucesso. Isso ocorre porque quando se introduz novas tecnologias, parte do contexto de tarefa de uma organização é alterado. Nessa reengenharia, novas tarefas e práticas são incorporadas enquanto que outras são eliminadas.

O levantamento de informações sobre as tarefas e os usuários pode ser melhor realizado quando os analistas procuram descrever situações do processo de trabalho. Os métodos baseados em **cenários** consistem de uma coleção de narrativas de situações no domínio que favorecem o levantamento de informações, a identificação de problemas e a antecipação das soluções. Cenários são uma maneira excelente de representar, para clientes e usuários, os problemas atuais e as possibilidades que podem surgir.

Os cenários têm como foco as atividades que as pessoas realizam nas organizações possibilitando uma perspectiva mais ampla dos problemas atuais onde o sistema será inserido, explicando porque ele é necessário. Eles proporcionam um desenvolvimento orientado a tarefas possibilitando maior usabilidade do sistema.

Não é objetivo dos cenários oferecer uma descrição precisa, mas provocar a discussão e estimular novos questionamentos. eles permitem também documentar o levantamento de informações a respeito dos problemas atuais, possíveis eventos, oportunidades de ações e riscos.

Por sua simplicidade, cenários são um meio de representação de fácil compreensão para os clientes e usuários envolvidos (muitas vezes de formação bastante heterogênea) que podem avaliar, criticar e fazer sugestões, proporcionando a reorganização de componentes e tarefas do domínio. Cenários oferecem um "meio-intermediário" entre a realidade e os modelos que serão especificados possibilitando que clientes, usuários e desenvolvedores participem do levantamento das informações e especificação dos requisitos. Em resumo, os cenários permitem compreensão dos problemas atuais pelos analistas e antecipação da situação futura pelo clientes e desenvolvedores.

Elaborando Cenários

Como toda atividade de análise e especificação de requisitos, a descrição do domínio através de cenários requer técnicas de comunicação e modelagem.

A descrição de cenários deve ser apoiada pelas três técnicas de comunicação vistas anteriormente. Antes de descrever os cenários, os analistas devem entrevistar clientes para entender os problemas e requisitos iniciais. A entrevista com usuários permite que cada um descreva as suas tarefas e os problemas associados a cada uma delas. A observação direta *in loco* é fundamental para que os analistas possam descrever a situação de uso como ela realmente vem ocorrendo na prática.

Após a elaboração dos cenários, clientes, usuários e analistas podem participar de **encontros** para que possam discutir cada um destes cenários. Eles podem ser afixados em quadros na parede onde os participantes possa analisá-los e fazer comentários, possivelmente afixando pequenos pedaços de papel a cada uma das cenas.

Cenários podem ser descritos em narrativas textuais ou através de *storyboards*. As narrativas textuais podem ser descritas livremente, identificando os agentes e as ações que eles participam. É possível utilizar notações para descrever roteiros de cinemas ou notações mais precisas como aquelas utilizadas pela Inteligência Artificial para representação de conhecimento.

Uma técnica que está bastante relacionada com cenários, por permitir descrever situações de uso, é o **storyboarding**. Essa técnica envolve a descrição através de quadros com imagens que ilustram as situações do domínio. Cada quadro deve apresentar a cena que descreve a situação, os atores e as ações que cada um deve desempenhar. Abaixo de cada quadro devem estar descritas ações que os atores desempenham. Os storyboards são bastante utilizados em cinemas como forma de comunicação entre roteiristas, diretores, atores e técnicos.

Existem ferramentas computacionais que podem ser utilizadas para a descrição de cenários como o *Scenario's Browser* [Rosson 99].

Exemplos de cenários

Como exemplo vamos considerar o domínio de uma locadora de fitas de vídeo.

Cena 1: *Cliente procura uma fita com uma certa atriz*
Agentes: Cliente, Atendente, Balconista
Ações:

> Cliente entra na loja e dirige-se até a atendente.
> **Cliente**: - *Eu gostaria de alugar um filme com a atriz que ganhou o oscar este ano.*
> **Antendente**: - *Algum específico?*
> **Cliente**: - *Não, mas que não seja policial ou ação.*
> **Atendente**: *Você sabe o nome dela?*
> **Cliente**: *Não*.
> A atendente pergunta à balconista.
> **Antendente**: - *Você sabe o nome da atriz que ganhou o Oscar 99?*
> **Balconista**: - *Ih. É um nome bem complicado. Só sei que começa com G e o sobrenome é algo parecido com Paltrow.*
> **Cliente**: *É isto mesmo.*

A atendente então procura no fichário de atrizes as que começam com G. Não encontra nenhum nome parecido com o que a balconista falou. Ela então lembra-se de consultar uma revista especializada com os ganhadores do Oscar 99 e lá descobre o nome correto da atriz. Entretanto, não existe realmente ficha alguma com os filmes estrelados por esta atriz. O fichário está desatualizado.
A atendente procura nas estantes alguns filmes e lembra-se de dois: *O Crime Perfeito* e *Seven* e mostra-os ao cliente. O cliente recusa-se dizendo que não gosta do gênero destes filmes e pergunta pelo filme vencedor do Oscar.
A antendente responde: - *Shakespeare Apaixonado?* Ainda não saiu em vídeo.

Cena 2: *O cliente procura por filmes de um certo gênero*
Agentes: Cliente, Atendente, Balconista
Ações:

Cliente: - *Eu gostaria de comprar um filme de ação*.
Atendente: - *Nós apenas alugamos.*
Cliente: - *Tudo bem. Então, por favor, me dê algumas dicas de filmes de ação.*
Atendente: *Com algum ator especial.*
Cliente: *Pode ser com Chuck Norris, Van Dame, Statellone, Charles Bronson*
Atendente: *Temos estes aqui.*
A atendente apresenta dez filmes. O cliente escolhe cinco e fica em dúvida se já assistiu outros três. Ele também pergunta à atendente se os outros dois filmes são bons. Após conversar durante alguns minutos com a atendente que entende muito do gênero, decide ficar com seis fitas. A atendente encaminha o cliente à balconista para calcular o valor da locação e o prazo de devolução. Após consultar as tabelas de preços e prazos, a balconista apresenta três planos de pagamento.
Balconista: - *Se você devolver em um dia, paga apenas R$ 6,00. Se devolver em seis dias paga R$ 12,00 e se devolver em uma semana paga R$ 15,00. Após este prazo paga mais R$ 2,00 por fita, por dia*.

Cena 3: *Cliente procura por filme usando o sistema de consulta*
Agentes: Cliente e Sistema de Consulta
Ações:

João gostaria de assistir a um filme de guerra. Ele vai a uma locadora de fitas e, chegando lá, utiliza um sistema de consulta. Ele não lembra o título nem o diretor, mas sabe que se passava na guerra do Vietnã e lembra bastante da trilha sonora. Utilizando o sistema ele solicita as opções de filmes de guerra. Os sistema oferece a ele as possibilidades de escolher os filmes por local da guerra ou por época. João escolhe a sua opção (guerra) e obtem uma lista com dezenas de filmes sobre o vietnam. Ele pode organizar esta lista, por diretor, atores e título. Na tela do sistema aparecem a ilustração com o cartaz de divulgação do filme e uma opção para ele visualizar o trailler. O sistema, entretanto, não possibilita ele ouvir a trilha sonora.

Após analisar algumas opções ele finalmente encontra o filme desejado, mas uma informação na tela do sistema indica que o filme está alugado. João quer saber quando o filme será devolvido, mas esta informação não consta no sistema. Ele entretanto pode reservar e o sistema enviará para ele um e-mail avisando quando o filme estiver disponível. Ele sai da loja pensando "Que tempo perdido. Seria melhor que eu pudesse consultar o sistema de casa".

Obtendo informações dos cenários através do *Questionamento Sistemático*

A descrição de informações do domínio através de narrativas só é efetivamente realizada se o processo de compreensão por parte dos analistas e projetista for realizado de maneira sistemática [J. Carroll et al. 94].

O **questionamento sistemático** é uma técnica psico-linguística que permite a psicólogos e lingüistas examinar o conteúdo e a estrutura de informações contidas numa narrativa. Uma narrativa é um sumário de um conjunto de eventos e ações envolvendo agentes e objetos do mundo. Nem todas as informações integrantes do contexto são passadas através da narrativa. Muitas dessas informações são inferidas do conhecimento cultural de cada indivíduo. Outras, entretanto, precisam ser obtidas diretamente na fonte, isto é, junto ao autor da narrativa. Essa técnica foi desenvolvida para se entender melhor o processo de compreensão de estórias em narrativas. O objetivo é compreender tudo o que envolve o contexto daquilo que está sendo passado na narrativa.

Aplicando essa técnica no processo de análise de domínios, os especialistas devem descrever em narrativas seu conhecimento do domínio. Entretanto esse conhecimento é muito mais abragente. O questionamento sistemático permite obter todo o conhecimento que está além do que foi comunicado na narrativa. Assim, analistas e projetista podem utilizar essa técnica para adquirir mais eficazmente conhecimento sobre o domínio e inferir objetos e interações que não estão descritos na narrativa. Isto ocorre usando-se cada sentença ou afirmação da narrativa como ponto de entrada na complexidade do problema.

Nessa técnica, cada questão é uma ponte entre uma idéia e outra. Uma resposta a uma questão sobre um componente da narrativa revela outras conexões que são críticas para o seu entendimento. Realizando, sistematica e exaustivamente muitos tipos de questões sobre componentes da narrativa e iterativamente fazendo perguntas sobre as respostas geradas, o analista elabora um mapa conceitual (rede de proposições) que representa as estruturas do conhecimento do domínio.

Os passos do método consistem de:

1. **Geração do cenário** - as narrativas que compõe o cenário devem ser decritas pelo especialista no domínio. O analista pode motivá-lo fazendo perguntas como num processo convencional de entrevista (questões de elicitação do cenário).
2. **Elaboração da rede de proposições** - as narrativas devem ser simplificadas e expressas através de proposições.
3. **Análise** - a partir das proposições pode-se determinar as **tarefas** (ações e objetos) e **usuários** (agentes das ações).
4. **Questionamento sistemático** - novas proposições podem ser elaboradas através de questões que são feitas sobre elementos das proposições anteriores, num processo iterativo.

Nos passos iniciais obtem-se informações sobre agentes, interações e objetos abstratos do domínio. Usando a técnica, pode-se chegar a um conhecimento mais profundo do domínio que permite aos analistas e projetista a elaboração de modelos funcionais do sistema.

Exemplo
Considere o seguinte cenário sobre um caixa eletrônico.
Questão de elicitação do cenário:

> *Como posso sacar R$ 100,00 do caixa eletrônico?*

Cenário:

> *Primeiro ponha o seu cartão do banco no caixa eletrônico, digite a sua senha e pressione o botão "saque rápido". Depois pegue o dinheiro*.

As duas sentenças do cenário acima contém quatro proposições:

> CLIENTE -- põe -- CARTÃO
> CLIENTE -- digita -- SENHA
> CLIENTE -- pressiona -- BOTÃO-DE-SAQUE-RÁPIDO
> CLIENTE -- pega -- DINHEIRO

A partir dessas proposições, o analista determina que o cartão e o cliente são agentes de ações. Numa análise voltada para a elicitação de requisitos da interação, seria determinado como usuário do sistema, o cliente. A ações são portanto: digita, pressiona e pega. São objetos da interação a senha, o botão e o dinheiro. Outros objetos são o caixa eletrônico e o cartão. É preciso determinar que tipo de objetos são esses. Uma outra dúvida é a respeito do cartão ser objeto ou agente.

Obviamente, como esse exemplo é bastante simples e a aplicação também é muito conhecida, parece desnecessário obter mais conhecimentos a respeito. Entretanto, como o objetivo aqui é exemplificar a técnica, mostraremos como pode-se questionar a respeito dessa aplicação.

O analista deve então realizar uma série de questões sobre as proposições. Nesse questionamento o analista precisa determinar qual o relacionamento entre a resposta e a proposição que originou a pergunta.

As questões da categoria por que, visam responder "razões" (metas) e "causas" a respeito de eventos da narrativa. As questões da categoria como oferecem maiores detalhes a respeito de determinados eventos, permitindo determinar sub-tarefas e maiores detalhes sobre a interação. Questões da categoria o que podem ser feitas sobre objetos, e revelam atributos e hieraquias de objetos. Perguntas de verificação podem ser feitas para se saber se as proposições estão sendo entendidas corretamente. As perguntas de verificação são as que têm resposta sim/não. Uma taxonomia mais completa ainda está sendo pesquisada pelos autores do método.

Continuando o exemplo anterior a tabela abaixo descreve uma seção de questionamento sistemático:

Questões "por que"

> Por que o cartão entra no caixa eletrônico?
> _ Para iniciar. (evento conseqüente)
> _ E então a máquina saberá quem é o cliente. (estado conseqüente)
> Por que o cliente digita sua senha?
> _ Para provar que ele é o usuário autorizado. (meta)

Por que o cliente pressiona o botão de saque rápido?
_ Porque é o botão para saques de R$ 100,00 (critério de descriminação).
_ Para evitar a digitação (cenário alternativo).
_ Porque ele está com pressa (causa)

Questões "como"

Como o cliente põe o cartão?
_ O cliente tira o cartão de sua bolsa e
_ insere no local apropriado no caixa eletrônico..
Como o cliente digita a senha?
_ Pressionando os botões adequados.
Como o cliente pressiona o botão de saque rápido?
_ Colocando o seu dedo nele.

Questões "o que"

O que é um botão de saque rápido?
_ É um tipo de botão que se pode pressionar.
O que é um botão?
_ É um dispositivo de controle no painel do caixa eletrônico.

Observe que se o analista estivesse utilizando essa técnica para num método orientado a objetos, outras questões poderiam ser realizadas com outros objetivos de acordo com as necessidades do método, como, por exemplo, para determinar classes e hierarquia de classes.

Após os cenários estarem desenvolvidos, os analistas devem trabalhar em conjunto com os usuários para avaliá-los e refiná-los. Isto pode ser feito, por exemplo, colocando-se *posters* numa sala pela qual os usuários podem circular livremente e observar os diversos cenários. Cada cenário deve apresentar quadros (desenhos, gráficos, fotografias, etc.), usando *storyboards* por exemplo, e uma descrição narrativa da tarefa. Os usuários, munidos de papeis e lápis, podem fazer anotações (críticas e sugestões) e anexá-las a cada *poster*.

Considerações finais sobre cenários

O resultado da modelagem através de cenários é uma base para a compreensão de quem são os usuários, quais a tarefas envolvidas e quais funções a interface e a aplicação devem provê, de maneira que, já se possa ter meios de garantir a usabilidade do sistema.

A idéia de cenários em análise não está necessariamente associada à técnica de questionamento sistemático. Os cenários são extremamente úteis para descrição do domínio. A técnica sistematiza o processo de compreensão do conhecimento adquirido.

Os métodos em geral, e esse não deve fugir à regra, devem ser usados, não como uma camisa-de-força que limite o processo de análise, mas como ferramentas que orientam os analistas e aumentam sua capacidade intelectual.

3.7 Análise de Usuários

Para que o sistema seja construído como uma ferramenta de apoio às atividades de usuários no domínio de aplicação, é preciso conhecer quem são os usuários e quais as suas necessidades, isto é quais tarefas eles necessitam realizar. A análise de usuários deve determinar quem eles são e quais tarefas eles normalmente fazem no domínio. Ela envolve a descrição dos diferentes papéis de usuários e qual o conhecimento, capacidade e cultura possuem os futuros usuários do sistema.

3.7.1 Análise de Perfis de Usuários

A análise do perfil dos diversos usuários do sistema descreve as várias características que podem influenciar as decisões dos projetistas no desenvolvimento do sistema. Os objetivos são assegurar que certas propriedades do sistema estejam adequadas ao conhecimento, cultura e capacidades do usuário, e que potenciais deficiências sejam levadas em consideração. Por exemplo, para um software de acompanhamento de pacientes em hospital, certos termos específicos da medicina devem ser incluídos nas telas do sistema e devem ser evitados termos técnicos de informática (*forneça informações patológicas* ao invés de *entrar dados da doença*). Para usuários com alguma deficiência física ou motora, elemento da interface devem ser modificados, como por exemplo, tela de maior tamanho e letras maiores para deficientes visuais.

Os perfil do usuário pode ser analisado através de formulários do tipo:

Perfil do Usuário

Nome do Sistema: _____
Função do Usuário: _____

Conhecimento e Experiência do Usuário

Nível educacional	**Língua Nativa**	**Nível de leitura e expressão**
( ) Ensino fundamental	( ) Português	( ) Excelente
( ) Ensino médio	( ) Inglês	( ) Bom
( ) Graduação	( ) outra: _____	( ) Regular
( ) Pós-Graduação		( ) Ruim
Experiência com computadores	**Experiência com sistema similar**	**Conhecimento sobre o domínio**

( ) Iniciante
( ) Intermediário
( ) Experiente

( ) Utiliza bastante um similar
( ) Já utilizou um similar
( ) Nunca utilizou um similar

( ) Elementar
( ) Intermediário
( ) Especialista no domínio

Características Físicas

Manipulação
( ) Canhoto
( ) Destro
( ) Ambidestro

Deficiências
( ) Auditiva
( ) Visual
( ) Corporal
( ) Vocal

O perfil deve dar as informações necessárias que os desenvolvedores precisam para tomar as suas decisões. A análise do perfil pode ser adaptada de acordo com as características do sistema e com as necessidades de analistas e desenvolvedores. Por exemplo, pode ser interesse dos designers saber se os usuários têm algumas experiência com interfaces gráficas e qual o padrão (Windows, Motif, Macintosh, etc.) eles estão acostumados a utilizar.

3.7.2 Papéis de Usuários

O papel (ou função) específico de cada usuário é definido pelas tarefas que eles realizam. Numa organização, as pessoas trabalham juntas, de maneira estruturada. A estrutura da organização define relacionamentos entre as pessoas. A implicação imediata dos diferentes papéis de cada usuário são as diferentes tarefas que eles irão realizar. Algumas tarefas podem ser comuns a diferentes papéis de usuários, enquanto outras podem ser exclusivas de papéis específicos.

O conceito de papel de usuário permite abstrairmos as características específicas de um usuário e enfatizar nas diferentes necessidades associadas a função que ele exerce. Para cada papel devemos associar um conjunto de funções, como veremos mais adiante.

No domínio do departamento de informática da UFRN podemos identificar como papéis de usuários: *secretário do departamento, chefe, coordenador de graduação, secretário da coordenação, coordenador de pós-graduação, professor, aluno, funcionário de administração de laboratórios e usuário externo*.

3.8 Análise e Modelagem de Tarefas

Os cenários permitem o levantamento e a descrição mais global das informações, das tarefas e dos problemas do domínio. O perfil de usuário descreve as características de usuários em termo de conhecimento, cultura, capacidades e limitações. A análise de tarefas visa determinar quais as atividades que os usuários (ou cada papel de usuário) devem realizar. Esta informação é essencial para se especificar os requisitos funcionais, determinando a funcionalidade do software. Para que o sistema possa ser construído para que estes problemas sejam resolvidos, ele deve ser uma ferramenta auxiliar na realização das tarefas de cada usuário.

Uma tarefa é, genericamente, uma atividade na qual um ou mais agentes tentam atingir uma meta especificada, através de uma mudança de estado em uma ou mais entidades do mundo. Num domínio de aplicação, as tarefas são as atividades que modificam estados de elementos deste domínio. A construção de um sistema computacional em um certo domínio tem por objetivo apoiar a realização de algumas destas atividades. Durante o processo de análise, deve-se determinar quais as do domínio e identificar quais delas podem ser auxiliadas pelo sistema.

A análise e modelagem de tarefas visa descrever as principais as tarefas que cada usuário do sistema terá de realizar para que os projetistas possam elaborar quais funções o sistema deve oferecer para que elas possam ser desempenhadas. Estas tarefas são descritas em termos de metas e um planejamento de possíveis atividades necessárias para atingi-las. As tarefas podem ser descritas a partir das informações obtidas nos cenários e devem ser agrupadas por **papéis de usuário**.

A análise de tarefas pode ser utilizadas nas diversas fases do ciclo de vida do software. Na fase de análise de requisitos ela pode ser utilizada para identificar problemas na maneira de como as tarefas vêm sendo realizadas. Os modelos de tarefas também podem descrever quais tarefas podem ser realizadas com o auxílio do sistema e como os usuários gostariam que ela fosse realizada. A análise de tarefa também é utilizada na avaliação do sistema para se determinar se como os usuários estão utilizando o sistema e se os objetivos foram atingidos. Nestes casos, a análise de tarefas ajuda ao projetista da interface ter uma visão da aplicação sob a perspectiva do usuário, isto é, um modelo das tarefas do usuário quando executando sessões da aplicação.

3.8.1 Modelo de tarefas como base para a especificação de requisitos funcionais

A análise e modelagem de tarefas pode ser utilizada como base para a especificação de requisitos funcionais. Para isto é preciso descrever as metas associadas a cada papel de usuário, que permitirão saber o que os usuário querem.

Resultados da psicologia cognitiva mostram que as pessoas realizam tarefas estabelecendo **metas** e elaborando um **plano** para cada uma delas. O **planejamento** consiste numa decomposição hierárquica de metas em **sub-metas** até que elas possam ser atingidas por **operações**. O **plano** ou **método** é, portanto, uma estrutura de sub-metas e operações para se atingir um certa meta. As **operações** correspondem a *ações básicas humanas*, isto é, aquelas que qualquer pessoa pode e sabe como realizar. São exemplos de operações *escrever uma palavra ou frase, ler uma informação, falar uma palavra ou frase, andar, relembrar, mover um objeto, pressionar um botão de controle* e várias outras.

Por exemplo, suponhamos que uma pessoa tem como meta *avisar a um amigo, através de uma carta, que sua filha nasceu*. Para atingir seu objetivo a pessoa deve estabelecer duas sub-metas: E*screver a carta e Colocá-la no correio*. A sub-meta escrever carta pode ser atingida pelo método: *Conseguir papel e lápis* e *Escrever mensagem*. Escrever mensagem já pode ser considerada uma operação, enquanto que conseguir papel e lápis requer um novo planejamento que determine as seguintes operações: *ir até o escritório, abrir o armário, pegar o papel e o lápis, levá-los até a mesa*.
O grão de refinamento do que podemos considerar com sendo uma operação é bastante subjetivo. Vai depender das dificuldades de quem realiza o plano. Na prática o plano é necessário quando a pessoa que vai realizar as ações não sabe ainda qual o método. Com a experiência o método torna-se automático e podemos considerar uma sub-meta como uma operação

Na utilização de um sistema computacional, os usuários realizam tarefas com o objetivo de atingir certas metas. Uma meta é um determinado estado do sistema ou de objetos do sistema ao qual o usuário quer chegar. Ao estabelecer a meta o usuário deve realizar um planejamento decompondo a meta em sub-metas até

que ele saiba que existe uma determinada função do sistema que permita que sua meta seja atingida. O caso agora é um pouco diferente do planejamento anterior, pois não é o usuário quem vai realizar a operação desejada, mas o sistema. A decomposição deve ocorrer até que ele identifique que o sistema tem uma determinada função que quando executada realiza a operação necessária para que sua meta seja atingida. Chamamos estas operações que o sistema executa para satisfazer as metas do usuário de **função da aplicação**. O conjunto de funções da aplicação determina a funcionalidade do sistema.

Vejamos um exemplo. Suponha que o usuário esteja escrevendo uma carta utilizando um editor de texto e tenha como meta *formatar um documento*.Para atingir esta meta ele estabelece as seguintes sub-metas: *Formatar página, Formatar parágrafos, Formatar fontes*. Para cada uma destas sub-metas ele estabelece novas sub-metas até que ele identifique no software funções que o sistema pode realizar que permitam que as sub-metas sejam atingidas. Por exemplo, formatar página pode ser decomposta na função da aplicação *especificar tamanho da página* e na sub-meta *definir margens*. Esta última por sua vez pode ser atingida pelas operações *especificar valor da margem superior, especificar valor da margem inferior, especificar valor da margem esquerda e especificar valor da margem direita.*

Vejamos o plano deste usuário

META: Formatar documento
PLANO:
Formatar página (sub-meta)
especificar tamanho da página (função da aplicação)
Definir margens (sub-meta)
especificar valor da margem superior (função da aplicação)
especificar valor da margem inferior (função da aplicação)
especificar valor da margem esquerda (função da aplicação)
especificar valor da margem direita (função da aplicação)
Formatar parágrafos (sub-meta)
selecionar parágrado (função da aplicação)
Especificar atributos do parágrafo (sub-meta)
especificar espaçamento (função da aplicação)
especificar recuos (função da aplicação)
...
Formatar fontes (sub-meta)
...

O modelo de tarefas é extremamente útil para determinarmos as metas dos usuários e quais funções da aplicação eles gostariam que o sistema oferecesse. Por exemplo, a especificação dos requisitos pode determinar que deve existir uma função da aplicação para formatar documento de maneira que a meta do usuário pudesse ser atingida pelo sistema sem a necessidade de planejamento algum.

É importante ressaltar que uma meta pode ser satisfeita por uma única ou por várias funções da aplicação e que também mais de um método pode ser atingido uma mesma função da aplicação.Por exemplo, ao utilizar o Word 7.0, o usuário pode ter como meta *formatar um estilo*. Ao construir o seu plano o usuário em

determinado momento pode estabelecer a sub-meta Especificar atributos do parágrafo. Esta sub-meta requer as mesmas funções de aplicação do plano para a meta formatar parágrafo. Assim, temos um grupo de funções da aplicação que são utilizadas para duas (ou mais) metas distintas.

Para que o usuário solicite ao sistema que execute uma determinada função de usuário, ele deve realizar operações que correspondam a um **comando de função**. Por exemplo, para o usuário solicitar ao sistema operacional que realize a função de copiar um arquivo de um diretório para outro ele deve escrever um comando do tipo **copy nome1 dir1 dir2** ou se estiver numa interface gráfica, mover o ícone correspondente ao arquivo da janela do diretório para a do outro diretório. Ao realizar o comando, o usuário precisa realizar operações com os dispositivos de interface do sistema, como *pressionar mouse, digitar número, teclar enter, etc.*

Apenas com a descrição das operações do usuário é que um plano para atingir uma meta fica completo. Quando o sistema está pronto, o plano tem que determinar exatamente as operações necessárias para comandar a função e, conseqüentemente, ter a meta atingida com o auxílio do sistema.

Na especificação de requisitos é suficiente decompor as metas que o usuário quer atingir nas correspondentes sub-metas. Caberá ao designer do software determinar qual o conjunto de funções que permita atingir o maior número possível de metas para cada papel de usuário e quais devem ser os comandos de interface para cada uma das funções.

3.8.2 Modelagem de Tarefas usando GOMS

Neste curso, utilizaremos a análise de tarefas na especificação de requisitos para determinar as tarefas que os usuários necessitam realizar com o sistema a ser construído. Para isto utilizaremos um método específico que utiliza o modelo **GOMS** simplificado. O modelo **GOMS** (*Goals, Operators, Methods, and Selection Rules*) oferece uma abodagem de análise da tarefa baseada num modelo do comportamento humano que possui três subsistemas de interação: o **perceptual** (auditivo e visual), o **motor** (movimentos braço-mão-dedo e cabeça-olho), e **cognitivo** (tomadas de decisão e acesso a memória). O modelos GOMS descreve o comportamento dinâmico da interação com um computador, especificando-se:

Metas - Uma aplicação é desenvolvida para auxiliar os usuários atingirem metas específicas. Isso requer uma série de etapas. Dessa forma, uma meta pode ser decomposta em várias submetas, formando uma hierarquia.

Operadores - São as ações humanas básicas que os usuários executam.

perceptual - operações visuais e auditivas (*olhar a tela, escutar um beep*).

motor - movimentos braço-mão-dedo e cabeça-olho (*pressionar uma tecla).*

cognitivo - tomadas de decisão, armazenar e relembrar um item da memória de trabalho.

Métodos - Um método é uma sequência de passos para se atingir uma meta. Dependendo do nível da hierarquia, os passos num método podem ser submetas, operadores ou a combinação de ambos.

Regras de Seleção - Pode existir mais de um método para se atingir uma meta. Uma regra de seleção especifica certas condições que devem ser satisfeitas antes que um método possa ser aplicado. Uma regra de seleção é uma expressão do tipo "condição-ação".

O GOMS permite que se construa modelos de tarefas bem mais complexos e detalhados do que o necessário numa tarefa de análise para a especificação de requisitos. Usaremos uma versão simplificada do GOMS, pois:

- o modelo da tarefa não deverá descrever informações de design da interface, uma vez que ela ainda não foi construída;
- o analista não é um especialista em psicologia cognitiva;
- o modelo simplificado pode ser expandido para o original, o que é bastante útil.

1. ***Diretrizes do Modelo de Tarefas Simplificado***

Uma vez que a hierarquia de metas representa um aumento no nível de detalhe, se nos limitarmos à descrição de metas e submetas de mais alto nível, nenhuma decisão de design será envolvida.

- **Faça a análise "top-down"** - comece das metas mais gerais em direção as mais específicas.
- **Use termos gerais para descrever metas** - não use termos específicos da interface.
- Examine todas as metas antes de descer para um nível mais baixo - isso facilita reuso de metas.
- **Considere todas as alternativas de tarefas** - as regras de seleção possibilitam representar alternativas.
- **Use sentenças simples para especificar as metas** - estruturas complexas indicam a necessidade de decomposição da meta em submetas.
- **Retire os passos de um método que sejam operadores** - os operadores são dependentes da interface.
- **Pare a decomposição quando as descrições estiverem muito detalhadas** - quando os métodos são operadores ou envolvem pressuposições de design.

1. ***Modelagem da Tarefa para aplicações com múltiplas funções de usuários.***

   Se, para uma determinada aplicação, a função do usuário for um fator crítico dominante na análise de usuários, deveremos ter modelos de tarefas diferentes para cada função de usuário. No GOMS simplificado, veremos como representar as tarefas para cada usuário num só modelo. Antes de estudarmos a notação do modelo, vejamos algumas regras para modelos com múltiplas funções de usuários:

   - Inicie especificando metas de alto nível para cada função de usuário.
   - Se mais de um compartilham a mesma meta, agrupe-os sob uma só.
   - Se todos compartilham a mesma meta, retire as referências a funções de usuário.

2. ***Notação***

1. Notação para Funções de Usuários.

- Funções de usuários distintas serão denotadas pelo símbolo **FU** seguido por um número.

    ex.: Gerente de vendas (**FU1**), balconista (**FU2**), caixa (**FU3**)

- A descrição de cada função de usuário é a primeira parte do modelo de tarefas.

    ex.: Gerente de vendas (**FU1**): responsável pela vendas nas lojas. Tem acesso a todos os dados do sistema.

- Um ponto-e-vírgula (;) é usado para separar o símbolo da função do usuário do restante da notação do modelo de tarefas.

    ex.: FU1;...

- Se uma meta é usada para mais de uma função de usuário, elas devem ser separadas por uma vírgula (,).

    ex.: FU1, FU2; ...

- Um asterisco (*) é usado se uma meta é aplicada a todas as funções de usuários.

    ex.: *;...

1. **Notação para especificação de metas**

- Cada passo num método é numerado numa ordem seqüencial, com cada nível de decomposição separado por um ponto (.), e com a endentação apropriada para reforçar a noção de hierarquia. Após o último número usa-se dois-pontos (:).

    ex.: FU2; 2.1: Anotar correções.

- Um comentário começa com duas barras inclinadas para direita (//) e acaba ao final da linha.

    ex.: // Para fazer as anotações o balconista deverá examinar as
    // listagens produzida durante as vendas do dia.

    FU2; 2.1: Anotar correções

- **Notação para Métodos e Regras de Seleção**

- Se uma meta possui mais de um método para ser atingida, uma letra do alfabeto é usada de forma ordenada após o número que descreve a meta.

  ex.:

  FU2; 2: Fazer relatório de vendas (meta)

  FU2; 2A: ... (método A)

  FU2; 2B: ... (método B)

- As regras de seleção e o método associado são descritos como pares "condição-ação", logo após a notação numérica da meta.

  ex.: FU2; 2A: se (dia de hoje for sábado)

  então (fazer relatório semanal)

1. **Reutilizando Metas**

   Um aspecto importante dessa notação é que pode-se reutilizar metas, simplesmente usando o mesmo número de uma meta preexistente.

   ex.:

   FU2; 2.1: Anotar correções. (meta preexistente)

   ...

   FU1, FU3; 3: Modificar livro-caixa.

   FU1, FU3; 3.1: Procurar lotes em aberto.

   FU1, FU3; 2.1: Anotar correções. (meta reusada)

   FU1, FU3; 3.3: Recalcular valores.

2. **Diretrizes adicionais**

- Delimite os passos de um método entre chaves: {}.
- Em aplicações com apenas uma função de usuário, não é necessário incluir a notação de função usuário antes de cada meta.
- Num nível de decomposição onde todas as metas têm o mesmo número-prefixo, apenas o número que indica a seqüência naquele nível é necessário.
- A diretriz anterior se aplica também à notação de função de usuário.
- Ao reutilizar uma meta anterior é necessário usar a notação completa para ela.

ex.: FU1, FU3; 3: Modificar livro-caixa.

1: Procurar lotes em aberto.

2.1: Anotar correções. (meta reusada)

3: Recalcular valores.

3.9 Especificando Requisitos utilizando Casos de Uso

Após saber quais as tarefas associadas a cada papel de usuário, é hora de elaborar os **casos de uso (use cases)** que permitem definir as funções de aplicação que o sistema deverá oferecer para o usuário. Os casos de uso podem ser utilizadas durante a análise e especificação dos requisitos para descrever a funcionalidade do sistema.

Os casos de uso foram definidos como parte da metodologia de Jacobson: *Object-oriented Analysis and Design - The User Case Driven Approach*. A linguagem de modelagem UML (*Unified Modeling Language*) apresenta notações para a representação de casos de uso.

Um caso de uso especifica o comportamento do sistema a ser desenvolvido sem, no entanto, especificar com este comportamento será implementado. Os comportamentos descrevem as funções da aplicação que caracterizam a funcionalidade do sistema. Um caso de uso representa o que o sistema faz e não como o sistema faz, proporcionando uma visão externa e não interna do sistema.

Cada caso de uso define um **requisito funcional** do sistema. Por exemplo, num sistema bancário, consulta de saldo, empréstimos e saques de dinheiro são exemplos de casos de uso.

O caso de uso descreve um conjunto de seqüências de ações que o sistema desempenha para produzir um resultado esperado pelo usuário. Cada seqüência representa a interação de entidades externas e o sistema. Estas entidades são chamadas de **atores** e que podem ser usuários ou outros sistemas. No caso de

usuários, um ator representa na verdade uma função de usuários.

Os casos de uso podem ser compostos por outros casos de uso mais específicos. Esta estrutura deve estar em conformidade com o particionamento do sistema em sub-sistemas. Desta forma tanto é possível descrever casos de uso que aplicam-se ao sistema global, quanto casos que são específicos para cada uma das partes do sistema.

Casos de uso também podem ser especializados, por exemplo uma consulta pode ser especializada em consulta de saldo e consulta de extrato, que por sua vez podem ser especializados, cada um , em consultas a conta-corrente ou a conta-poupança.

Ao definir os requisitos funcionais, o caso de uso define o comportamento, as respostas esperadas e os casos de testes que devem validar a implementação do sistema.

3.9.1 A representação de casos de uso usando UML

A notação UML será utilizada neste curso como linguagem de especificação para a maioria das atividades do ciclo de vida do sistema. A partir de agora vamos introduzir esta notação necessária para representar casos de uso. Ao longo do curso outros aspectos da linguagem serão introduzidos.

Um caso de uso é representado por uma **elipse**. Cada caso de uso disntigue-se de um outro por um **nome** que normalmente é um verbo seguido do seu objeto.

Um ator que pode ser uma ou mais função de usuário é representado por uma figura simples de um indivíduo (um ***boneco-de-palitos***).

Os atores são conectados a casos de uso por uma associação representadas por uma **linha**.

O comportamento associado a cada caso de uso pode ser descrito como um **cenário**. Cada cenário descreve textualmente o fluxo de eventos ou seqüência que caracteriza o comportamento do ator e as respostas do sistema. Um cenário é uma instância do caso de uso.

Por exemplo num caixa eletrônico bancário, o caso de uso *validar cliente* pode ser descrito pelo seguinte cenário:

> *Fluxo de eventos principal*: O casos de uso inicia quando o sistema solicita ao cliente (do banco) a sua senha. O cliente fornece os números através do teclado. O cliente confirma a senha pressionando a tecla *entre*. O sistema checa este número e verifica se ele é válido.
>
> *Fluxo de evento excepcional*: O cliente pode cancelar a transação a qualquer momento pressionando o botão *cancele*, reiniciando o caso de uso. Não é feita nenhuma mudança na conta do usuário.
>
> *Fluxo de evento excepcional*: O cliente pode corrigir a senha a qualquer momento antes de confirmar com a tecla *entre*.
>
> *Fluxo de evento excepcional*: Se o cliente fornece um número de senha inválido o caso de uso é reiniciado. Se isto acontecer três vezes seguidas, o sistema cancela o caso de uso e bloqueia por até uma hora.

Os casos de uso também podem ser descritos de maneira mais estruturada e formal, por exemplo, usando **pré- e pós-condições**. Diversas técnicas e notações para especificações formais permitem descrever o comportamento da funções da aplicação em termos de pré- e pós-condições. As pré-condições estabelecem as condições que devem estar satisfeita antes que a função seja executada pelo sistema, enquanto que as pós-condições determinam o que deve ser válido ao final da execução. Estes estados iniciais e finais do sistema descrevem o comportamento independente de como será implementado, isto é, quais os algoritmos, estrutura de dados e linguagem de programação a serem utilizados. As especificações formais não são abordadas neste curso.

Os casos de uso podem ainda ser descritos através de outros diagramas UML. Os **diagramas de atividades** descrevem os diversos estados e as transições entre cada um deles. Os diagramas de interação permitem descrever as interações entre o ator e o componente do sistema que implementa o caso de uso. Estes diagramas serão estudados mais adiante no capítulo 4.
Veja o exemplo acima especificado através de diagramas de estados e de atividades na seção 4.2.7.

3.9.2 Diagrama de Casos de Uso em UML

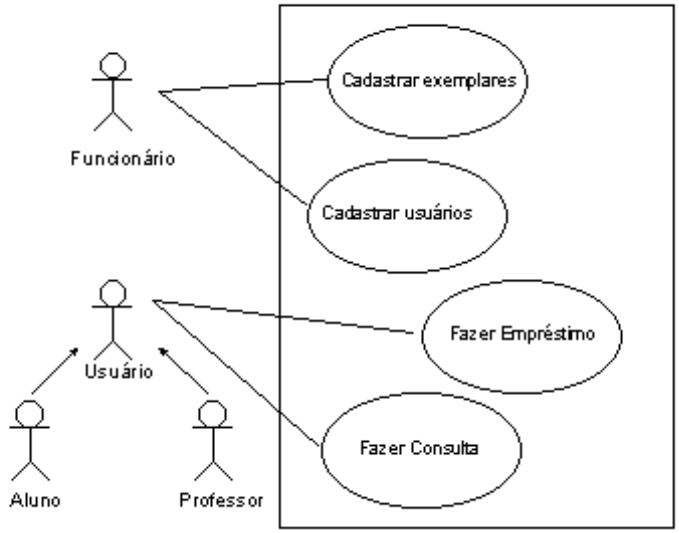
A UML permite elaborar diversos diagramas para visualização, especificação, construção e documentação de diversas partes do sistema em diversas etapas do ciclo de vida. Existem cinco tipos de diagramas que permitem modelar aspectos dinâmicos do sistema através da UML. O **diagrama de casos de uso** é um destes diagramas e pode ser utilizado para visualização, especificação e documentação de requisitos do sistema.

A especificação dos requisitos visa descrever o que o sistema deve fazer para satisfazer as metas dos usuários (requisitos funcionais) e quais outras propriedades é desejável que o sistema possua (requisitos não-funcionais). Vimos que casos de usos, individualmente, descrevem requisitos funcionais. Acrescentandos **Notas** aos diagramas de casos de uso podemos especificar também os requisitos não-funcionais.

Um diagrama de casos de uso contém:

- Casos de Uso
- Atores
- Relacionamentos de dependência, generalização e associações

Podem ser acrescentadas **Notas** com em qualquer outro diagrama UML.



Para modelar os requisitos de software de um sistema podemos seguir as seguintes regras.

- Estabeleça o contexto do sistema identificando os **atores** (usuários e sistemas externos) associados a ele.
- Para cada ator, considere o comportamento que cada um deles espera ou requer que o sistema possua, para que as suas metas sejam satisfeitas.
- Considere cada um destes comportamentos esperados como **casos de uso**, dando nomes para cada um deles.
- Analise os casos de uso tentando identificar comportamentos comuns a mais de um deles. Defina esta parte comum como caso de uso genérico, estabelecendo um relacionamento de **generalização**. Tente identificar outros **relacionamentos**, como a dependência entre casos de uso que incluem o comportamento de outros .
- Modele estes casos de uso, atores e seus relacionamentos em **diagramas de casos de uso**.
- Complemente estes casos de uso com **notas** que descrevam requisitos não-funcionais.

Além destas regras básicas, algumas dicas ajudam a construir diagramas mais claros.

- Dê nomes que comuniquem o seu propósito.
- Quando existirem relacionamentos, distribua os casos de uso de maneira a minimizar os cruzamentos de linhas.
- Organize os elementos espacialmente de maneira que aqueles que descrevem comportamento associados fiquem mais próximos.
- Use notas para chamar a atenção de características importantes - as notas não são apenas para os requisitos não funcionais.
- Não complique o diagrama, coloque apenas o que é essencial para descrever os requisitos. O diagrama deve ser simples sem deixar de mostrar o que é relevante.

 Última atualização: 16/05/00

4. Análise e Especificação de Requisitos

Os objetivos deste capítulo são:

- Definir o que são requisitos de software
- Introduzir os objetivos da Engenharia de Requisitos
- Apresentar técnicas de comunicação para obter informações dos clientes e usuários
- Apresentar técnicas para descrever o domínio, usuários e tarefas.
- Especificar requisitos funcionais utilizando Casos de Uso
- Especificar requisitos não funcionais

4.1 Engenharia de Requisitos

Vimos que o software é parte de um sistema computacional mais abrangente e que a Análise de Sistemas é a atividade de identificar os problemas do domínio, apresentar alternativas de soluções e o estudo da viabilidade de cada uma delas. Uma vez que se tenha feito a análise do sistema computacional, e delimitado o escopo do software, os requisitos do software devem ser analisados e especificados.

A **análise e especificação de requisitos de software** envolve as atividades de determinar os objetivos de um software e as restrições associadas a ele. Ela deve também estabelecer o relacionamento entre estes objetivos e restrições e a especificação precisa do software.

A análise e especificação dos requisitos de software deve ser vista como uma sub-atividade da análise de sistemas. Normalmente ela é iniciada juntamente com a análise do sistema, podendo se estender após a elaboração do documento de especificação do sistema e do planejamento do desenvolvimento, quando serão refinados os requisitos do software.

Análise e especificação são atividades inter-dependentes e devem ser realizadas conjuntamente. A **análise** é o processo de observação e levantamento dos elementos do domínio no qual o sistema será introduzido. Deve-se identificar as pessoas, atividades, informações do domínio para que se possa decidir o que deverá ser informatizado ou não. Pessoas e as atividades que não serão informatizadas deverão ser consideradas entidades externas ao software.

A **especificação** é a descrição sistemática e abstrata do que o software deve fazer, a partir daquilo que foi analisado. Ela apresenta a solução de como os problemas levantados na análise serão resolvidos pelo software do sistema computacional. Visa descrever de maneira sistemática quais as propriedades

funcionais são necessárias para resolver o problema do domínio. A especificação é também a forma de comunicação sistemática entre analistas e projetistas do software.

O objetivo da definição dos requisitos é especificar o que o sistema deverá fazer e determinar os critérios de validação que serão utilizados para que se possa avaliar se o sistema cumpre o que foi definido.

4.1.1 O que são requisitos?

Como sistemas computacionais são construídos para terem utilidade no mundo real. Modelar uma parte do mundo real, o domínio de aplicação é uma atividade extremamente importante para se compreender a necessidade e a importância do sistema a ser construído e definir os requisitos que tornam o sistema útil.

Requisitos são objetivos ou restrições estabelecidas por clientes e usuários do sistema que definem as diversas propriedades do sistema. Os requisitos de software são, obviamente, aqueles dentre os requisitos de sistema que dizem respeito a propriedades do software.

Um conjunto de requisitos pode ser definido como uma condição ou capacidade necessária que o software deve possuir (1) para que o usuário possa resolver um problema ou atingir um objetivo ou (2) para atender as necessidades ou restrições da organização ou dos outros componentes do sistema.

Tradicionalmente, os requisitos de software são separados em requisitos funcionais e não-funcionais. Os **requisitos funcionais** são a descrição das diversas funções que clientes e usuários querem ou precisam que o software ofereça. Eles definem a funcionalidade desejada do software. O termo função é usado no sentido genérico de operação que pode ser realizada pelo sistema, seja através comandos dos usuários ou seja pela ocorrência de eventos internos ou externos ao sistema.

São exemplos de requisitos funcionais:

- "o software deve possibilitar o cálculo dos gastos diários, semanais, mensais e anuais com pessoal".
- "o software deve emitir relatórios de compras a cada quinze dias"
- "os usuários devem poder obter o número de aprovações, reprovações e trancamentos em todas as disciplinas por um determinado período de tempo.

A especificação de um requisito funcional deve determinar o que se espera que o software faça, sem a preocupação de como ele faz. É importante diferenciar a atividade de especificar requisitos da atividade de especificação que ocorre durante o design do software. No design do software deve-se tomar a decisão de quais a funções o sistema efetivamente terá para satisfazer àquilo que os usuários querem.

Requisitos não-funcionais são as qualidades globais de um software, como manutenibilidade, usabilidade, desempenho, custos e várias outras. Normalmente estes requisitos são descritos de maneira informal, de maneira controversa (por exemplo, o gerente quer segurança mas os usuários querem facilidade de uso) e são difíceis de validar.

São exemplos de requisitos não-funcionais:

- "a base de dados deve ser protegida para acesso apenas de usuários autorizados".
- "o tempo de resposta do sistema não deve ultrapassar 30 segundo".
- "o software deve ser operacionalizado no sistema Linux"
- "o tempo de desenvolvimento não deve ultrapassar seis meses".

A necessidade de se estabelecer os requisitos de forma precisa é crítica na medida que o tamanho e a complexidade do software aumentam. Os requisitos exercem influência uns sobre os outros. Por exemplo, o requisito de que o software deve ter grande *portabilidade* (por exemplo, ser implementado em Java) pode implicar em que o requisito *desempenho* não seja satisfeito (programas em Java são, em geral, mais lentos).

4.1.2 A Engenharia de Requisitos

Os diversos relacionamento e restrições que os requisitos exercem uns sobre os outros são muito difíceis de ser controlados. Principalmente se considerarmos que algumas decisões de design que afetam um ou mais requisitos só serão tomadas mais adiante do desenvolvimento. Por exemplo, a decisão de implementar em Java pode ser decidida apenas após o design da arquitetura. Desta forma, os requisitos precisam ser gerenciados durante todo o desenvolvimento.

A importância e complexidade desta atividade levou ao surgimento, no início dos anos 90, da **Engenharia de Requisitos.** O objetivo desta denominação é ressaltar que o processo de definir os requisitos de software é uma atividade extremamente importante e independente das outras atividades da engenharia de software. Ela requer fundamentação e processos próprios, e que devem ser planejados e gerenciados ao longo de todo o ciclo de vida.

Os objetivos da Engenharia de Requisitos podem ser categorizados em três grupos principais [Zave]:

- Investigação de objetivos, funções e restrições de uma aplicação de software
  - Ultrapassar as barreiras de comunicação
  - Gerar estratégias para converter objetivos vagos em propriedades ou restrições específicas
  - Compreender prioridades e graus de satisfação
  - Associar requisitos com os vários agentes do domínio
  - Estimar custos, riscos e cronogramas
  - Assegurar completude
- Especificação do software
  - Integrar representações e visões múltiplas
  - Avaliar estratégias de soluções alternativas que satisfaçam os requisitos
  - Obter uma especificação completa, consistente e não ambígua
  - Validar a especificação - verificar que ela satisfaz aos requisitos
  - Obter especificações que sejam apropriadas para as atividades de design e implementação
- Gerenciamento da evolução e das famílias do software
  - Reutilização de requisitos durante o processo de evolução
  - Reutilização de requisitos no desenvolvimento de software similares

- Reconstrução de requisitos

A Engenharia de Requisitos deve envolver a documentação das funções, do desempenho, interfaces externas e internas e atributos de qualidade do Software.

Esta área é inerentemente abrangente, interdisciplinar e aberta. Ela lida com a descrição de observações do mundo real através de notações apropriadas.

Os benefícios da Engenharia de Requisitos são:

- Concordância entre desenvolvedores, clientes e usuário sobre o trabalho a ser feito e quais os critérios de aceitação do sistema.
- Uma base precisa para a estimativa dos recursos (custo, pessoal, prazos, ferramentas e equipamentos)
- Melhoria na usabilidade, manutenibilidade e outras qualidades do sistema.
- Atingir os objetivos com o mínimo de desperdício

Uma boa especificação de requisitos deve ser:

- Clara e não-ambígua
- Completa
- Correta
- Compreensível
- Consistente
- Concisa
- Confiável

(Veja mais em Dorfman, Merlin - Requirements Engineering)

4.1.3 Técnicas de levantamento de requisitos

Um dos objetivos da Engenharia de Requisitos é ultrapassar barreiras de comunicação entre os clientes e usuários e os analistas para que os requisitos possam capturados e modelados corretamente

Dentre as técnicas mais importantes para a comunicação podemos citar três:

- **Entrevistas**
- **Observação *in loco***
- **Encontros**

Estas três técnicas são complementares e podem todas ser usadas numa mesma análise de requisitos. A **entrevista** é normalmente a primeira técnica utilizada. Analistas entrevistam clientes para definir os objetivos gerais e restrições que o software deverá ter. A entrevista deve ser feita de forma objetiva visando obter o

máximo de informações do cliente. Diversas seções de entrevistas podem ser marcadas.

Na **observação *in loco*** os analista devem estar inseridos na rotina de trabalho da organização tentando entender e descrever as principais atividades que são realizadas. Na observação devem ser identificadas quais as atividades podem ser automatizadas, quem são os potenciais usuários, quais tarefas eles querem realizar com a ajuda do novo sistema, etc. A observação deve ser complementada com entrevistas específicas com os futuros usuários.

Os **encontros** são reuniões envolvendo analistas, clientes e usuários destinadas exclusivamente ao levantamento de informações, descrição dos problemas atuais e de metas futuras. Os encontros devem ser realizados em um local neutro (fora da organização) e normalmente duram poucos alguns dias. Nos encontros as informações que vão sendo levantadas vão sendo afixadas em paineis na sala de encontro para que possam ser analisadas e validadas pelos clientes e usuários. Ao final do encontro os analistas devem elaborar um relatório descrevendo os requisitos analisados.

4.1.4 Modelos de documentos de especificação de requisitos

O resultado final da análise e especificação de requisitos e das outras atividades da fase de definção devem ser apresentados aos clientes para que eles possam validá-lo. Este documento oferece a concordância entre clientes, analistas e desenvolvedores sobre o que deve ser desenvolvido. É utilizando este documento que as atividades da fase de desenvolvimento (design e programação) serão validadas.

Cada desenvolvedor utiliza um modelo específico para elaborar este documento. A sua estrutura muitas vezes depende do método que está sendo utilizado. Em linhas gerais este modelo deve ser basicamente textual, utilizando o mínimo de termos técnicos, e ilustrados como modelos gráficos que demonstrem mais claramente a visão que os analistas tiveram dos problemas e dos requisitos para o futuro sistema.

Vamos apresentar a seguir dois modelos de documentos encontrados na literatura. Estes modelos descrevem não apenas a especificação dos requisitos, mas os resultados do estudo de viabilidade e o processo de desenvolvimento.

Pressman apresenta o seguinte documento de especificação de requisitos de software:
I. Introdução

1. Referências do Sistema
2. Descrição Geral
3. Restrições de projeto do software

II. Descrição da Informação

1. Representação do fluxo de informação
    a. Fluxo de Dados
    b. Fluxo de Controle
2. Representação do conteúdo de informação
3. Descrição da interface com o sistema

III Descrição Funcional

1. Divisão funcional em partições
2. Descrição funcional
    a. Narativas
    b. Restrições/limitações
    c. Exigências de desempenho
    d. Restrições de projeto
    e. Diagramas de apoio
3. Descrição do controle
    a. Especificação do controle
    b. Restrições de projeto

IV. Descrição Comportamental

1. Estados do Sistema
2. Eventos e ações

V. Critérios de Validação

1. Limites de desempenho
2. Classes de testes
3. Reação esperada do software
4. Considerações especiais

VI. Bibliografia
VII Apêndice

Uma proposta alternativa é a de Farley. De acordo com ele, um documento de especificação de requisitos deve possui as seguintes seções:

- Visão geral do produto e Sumário
- Ambientes de desenvolvimento, operação e manutenção
- Interfaces Externas e Fluxo de Dados
- Requisitos Funcionais
- Requisitos de Desempenho
- Tratamento de Exceções
- Prioridades de Implementação
- Antecipação de mudanças e extensões

- Dicas e diretrizes de Design
- Critérios de aceitação
- Índice Remissivo
- Glossário

4.2 Especificando Requisitos utilizando Casos de Uso

Após saber quais as tarefas associadas a cada papel de usuário, é hora de elaborar os **casos de uso (use cases)** que permitem definir as funções de aplicação que o sistema deverá oferecer para o usuário. Os casos de uso podem ser utilizadas durante a análise e especificação dos requisitos para descrever a funcionalidade do sistema.

Os casos de uso foram definidos como parte da metodologia de Jacobson: *Object-oriented Analysis and Design - The User Case Driven Approach*. A linguagem de modelagem UML (*Unified Modeling Language*) apresenta notações para a representação de casos de uso.

Um caso de uso especifica o comportamento do sistema a ser desenvolvido sem, no entanto, especificar com este comportamento será implementado. Os comportamentos descrevem as funções da aplicação que caracterizam a funcionalidade do sistema. Um caso de uso representa o que o sistema faz e não como o sistema faz, proporcionando uma visão externa e não interna do sistema.

Cada caso de uso define um **requisito funcional** do sistema. Por exemplo, num sistema bancário, consulta de saldo, empréstimos e saques de dinheiro são exemplos de casos de uso.

O caso de uso descreve um conjunto de seqüências de ações que o sistema desempenha para produzir um resultado esperado pelo usuário. Cada seqüência representa a interação de entidades externas e o sistema. Estas entidades são chamadas de **atores** e que podem ser usuários ou outros sistemas. No caso de usuários, um ator representa na verdade uma função de usuários.

Os casos de uso podem ser compostos por outros casos de uso mais específicos. Esta estrutura deve estar em conformidade com o particionamento do sistema em sub-sistemas. Desta forma tanto é possível descrever casos de uso que aplicam-se ao sistema global, quanto casos que são específicos para cada uma das partes do sistema.

Casos de uso também podem ser especializados, por exemplo uma consulta pode ser especializada em consulta de saldo e consulta de extrato, que por sua vez podem ser especializados, cada um , em consultas a conta-corrente ou a conta-poupança.

Ao definir os requisitos funcionais, o caso de uso define o comportamento, as respostas esperadas e os casos de testes que devem validar a implementação do sistema.

4.2.1 A representação de casos de uso usando UML

A notação UML será utilizada neste curso como linguagem de especificação para a maioria das atividades do ciclo de vida do sistema. A partir de agora vamos introduzir esta notação necessária para representar casos de uso. Ao longo do curso outros aspectos da linguagem serão introduzidos.

Um caso de uso é representado por uma **elipse**. Cada caso de uso disntigue-se de um outro por um **nome** que normalmente é um verbo seguido do seu objeto.

Um ator que pode ser uma ou mais função de usuário é representado por uma figura simples de um indivíduo (um ***boneco-de-palitos***).

Os atores são conectados a casos de uso por uma associação representadas por uma **linha**.

O comportamento associado a cada caso de uso pode ser descrito como um **cenário**. Cada cenário descreve textualmente o fluxo de eventos ou seqüência que caracteriza o comportamento do ator e as respostas do sistema. Um cenário é uma instância do caso de uso.

Por exemplo num caixa eletrônico bancário, o caso de uso *validar cliente* pode ser descrito pelo seguinte cenário:

> *Fluxo de eventos principal*: O casos de uso inicia quando o sistema solicita ao cliente (do banco) a sua senha. O cliente fornece os números através do teclado. O cliente confirma a senha pressionando a tecla *entre*. O sistema checa este número e verifica se ele é válido.
>
> *Fluxo de evento excepcional*: O cliente pode cancelar a transação a qualquer momento pressionando o botão *cancele*, reiniciando o caso de uso. Não é feita nenhuma mudança na conta do usuário.
>
> *Fluxo de evento excepcional*: O cliente pode corrigir a senha a qualquer momento antes de confirmar com a tecla *entre*.
>
> *Fluxo de evento excepcional*: Se o cliente fornece um número de senha inválido o caso de uso é reiniciado. Se isto acontecer três vezes seguidas, o sistema cancela o caso de uso e bloqueia por até uma hora.

Os casos de uso também podem ser descritos de maneira mais estruturada e formal, por exemplo, usando **pré- e pós-condições**. Diversas técnicas e notações para especificações formais permitem descrever o comportamento da funções da aplicação em termos de pré- e pós-condições. As pré-condições estabelecem as condições que devem estar satisfeita antes que a função seja executada pelo sistema, enquanto que as pós-condições determinam o que deve ser válido ao final da execução. Estes estados iniciais e finais do sistema descrevem o comportamento independente de como será implementado, isto é, quais os algoritmos, estrutura de dados e linguagem de programação a serem utilizados. As especificações formais não são abordadas neste curso.

Os casos de uso podem ainda ser descritos através de outros diagramas UML. Os **diagramas de atividades** descrevem os diversos estados e as transições entre cada um deles. Os diagramas de interação permitem descrever as interações entre o ator e o componente do sistema que implementa o caso de uso. Estes diagramas serão estudados mais adiante no capítulo 5.
Veja o exemplo acima especificado através de diagramas de estados e de atividades na seção 5.2.7.

4.2.3 Diagrama de Casos de Uso em UML

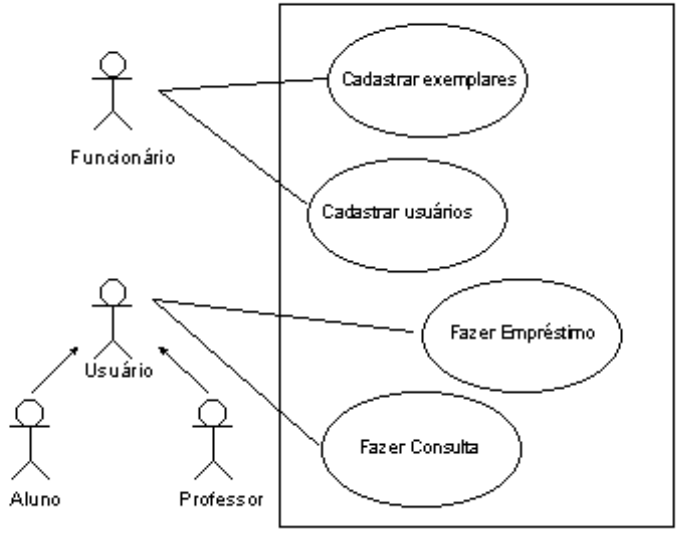
A UML permite elaborar diversos diagramas para visualização, especificação, construção e documentação de diversas partes do sistema em diversas etapas do ciclo de vida. Existem cinco tipos de diagramas que permitem modelar aspectos dinâmicos do sistema através da UML. O **diagrama de casos de uso** é um destes diagramas e pode ser utilizado para visualização, especificação e documentação de requisitos do sistema.

A especificação dos requisitos visa descrever o que o sistema deve fazer para satisfazer as metas dos usuários (requisitos funcionais) e quais outras propriedades é desejável que o sistema possua (requisitos não-funcionais). Vimos que casos de usos, individualmente, descrevem requisitos funcionais. Acrescentandos **Notas** aos diagramas de casos de uso podemos especificar também os requisitos não-funcionais.

Um diagrama de casos de uso contém:

- Casos de Uso
- Atores
- Relacionamentos de dependência, generalização e associações

Podem ser acrescentadas **Notas** com em qualquer outro diagrama UML.



Para modelar os requisitos de software de um sistema podemos seguir as seguintes regras.

- Estabeleça o contexto do sistema identificando os **atores** (usuários e sistemas externos) associados a ele.
- Para cada ator, considere o comportamento que cada um deles espera ou requer que o sistema possua, para que as suas metas sejam satisfeitas.

- Considere cada um destes comportamentos esperados como **casos de uso**, dando nomes para cada um deles.
- Analise os casos de uso tentando identificar comportamentos comuns a mais de um deles. Defina esta parte comum como caso de uso genérico, estabelecendo um relacionamento de **generalização**. Tente identificar outros **relacionamentos**, como a dependência entre casos de uso que incluem o comportamento de outros .
- Modele estes casos de uso, atores e seus relacionamentos em **diagramas de casos de uso**.
- Complemente estes casos de uso com **notas** que descrevam requisitos não-funcionais.

Além destas regras básicas, algumas dicas ajudam a construir diagramas mais claros.

- Dê nomes que comuniquem o seu propósito.
- Quando existirem relacionamentos, distribua os casos de uso de maneira a minimizar os cruzamentos de linhas.
- Organize os elementos espacialmente de maneira que aqueles que descrevem comportamento associados fiquem mais próximos.
- Use notas para chamar a atenção de características importantes - as notas não são apenas para os requisitos não funcionais.
- Não complique o diagrama, coloque apenas o que é essencial para descrever os requisitos. O diagrama deve ser simples sem deixar de mostrar o que é relevante.

4.3 Técnicas complementares para análise de requisitos

Definir casos a partir do nada pode ser bastante difícil. Antes de começar a pensar em casos de uso o analista pode descrever cenários com situações do domínio. Estes cenários contêm informações que podem ser extraídas mais detalhadamente com o objetivo de detalhar os cenários. Além dos cenários, a análise do perfil dos usuário e das tarefas que eles executam permitem um maior conhecimento do domínio, possibilitando uma melhor especificação dos requisitos.

4.3.1 Cenários

Do ponto de vista de usabilidade, o desenvolvimento de um novo sistema implica na transformação das tarefas e práticas atuais dos usuários e da organização. Conhecer a situação atual e antecipar o impacto que um novo sistema deve ter é fundamental para o seu sucesso. Isso ocorre porque quando se introduz novas tecnologias, parte do contexto de tarefa de uma organização é alterado. Nessa reengenharia, novas tarefas e práticas são incorporadas enquanto que outras são eliminadas.

O levantamento de informações sobre as tarefas e os usuários pode ser melhor realizado quando os analistas procuram descrever situações do processo de trabalho. Os métodos baseados em **cenários** consistem de uma coleção de narrativas de situações no domínio que favorecem o levantamento de informações, a identificação de problemas e a antecipação das soluções. Cenários são uma maneira excelente de representar, para clientes e usuários, os problemas atuais e as possibilidades que podem surgir.

Os cenários têm como foco as atividades que as pessoas realizam nas organizações possibilitando uma perspectiva mais ampla dos problemas atuais onde o sistema será inserido, explicando porque ele é necessário. Eles proporcionam um desenvolvimento orientado a tarefas possibilitando maior usabilidade do sistema.

Não é objetivo dos cenários oferecer uma descrição precisa, mas provocar a discussão e estimular novos questionamentos. eles permitem também documentar o levantamento de informações a respeito dos problemas atuais, possíveis eventos, oportunidades de ações e riscos.

Por sua simplicidade, cenários são um meio de representação de fácil compreensão para os clientes e usuários envolvidos (muitas vezes de formação bastante heterogênea) que podem avaliar, criticar e fazer sugestões, proporcionando a reorganização de componentes e tarefas do domínio. Cenários oferecem um "meio-intermediário" entre a realidade e os modelos que serão especificados possibilitando que clientes, usuários e desenvolvedores participem do levantamento das informações e especificação dos requisitos. Em resumo, os cenários permitem compreensão dos problemas atuais pelos analistas e antecipação da situação futura pelo clientes e desenvolvedores.

Elaborando Cenários

Como toda atividade de análise e especificação de requisitos, a descrição do domínio através de cenários requer técnicas de comunicação e modelagem.

A descrição de cenários deve ser apoiada pelas três técnicas de comunicação vistas anteriormente. Antes de descrever os cenários, os analistas devem entrevistar clientes para entender os problemas e requisitos iniciais. A entrevista com usuários permite que cada um descreva as suas tarefas e os problemas associados a cada uma delas. A observação direta *in loco* é fundamental para que os analistas possam descrever a situação de uso como ela realmente vem ocorrendo na prática.

Após a elaboração dos cenários, clientes, usuários e analistas podem participar de **encontros** para que possam discutir cada um destes cenários. Eles podem ser afixados em quadros na parede onde os participantes possa analisá-los e fazer comentários, possivelmente afixando pequenos pedaços de papel a cada uma das cenas.

Cenários podem ser descritos em narrativas textuais ou através de *storyboards*. As narrativas textuais podem ser descritas livremente, identificando os agentes e as ações que eles participam. É possível utilizar notações para descrever roteiros de cinemas ou notações mais precisas como aquelas utilizadas pela Inteligência Artificial para representação de conhecimento.

Uma técnica que está bastante relacionada com cenários, por permitir descrever situações de uso, é o **storyboarding**. Essa técnica envolve a descrição através de quadros com imagens que ilustram as situações do domínio. Cada quadro deve apresentar a cena que descreve a situação, os atores e as ações que cada um deve desempenhar. Abaixo de cada quadro devem estar descritas ações que os atores desempenham. Os storyboards são bastante utilizados em cinemas como forma de comunicação entre roteiristas, diretores, atores e técnicos.

Existem ferramentas computacionais que podem ser utilizadas para a descrição de cenários como o *Scenario's Browser* [Rosson 99].

Exemplos de cenários

Como exemplo vamos considerar o domínio de uma locadora de fitas de vídeo.

Cena 1: *Cliente procura uma fita com uma certa atriz*
Agentes: Cliente, Atendente, Balconista
Ações:

Cliente entra na loja e dirige-se até a atendente.
Cliente: - *Eu gostaria de alugar um filme com a atriz que ganhou o oscar este ano.*
Atendente: - *Algum específico?*
Cliente: - *Não, mas que não seja policial ou ação.*
Atendente: *Você sabe o nome dela?*
Cliente: *Não*.
A atendente pergunta à balconista.
Atendente: - *Você sabe o nome da atriz que ganhou o Oscar 99?*
Balconista: - *Ih. É um nome bem complicado. Só sei que começa com G e o sobrenome é algo parecido com Paltrow.*
Cliente: *É isto mesmo.*
A atendente então procura no fichário de atrizes as que começam com G. Não encontra nenhum nome parecido com o que a balconista falou. Ela então lembra-se de consultar uma revista especializada com os ganhadores do Oscar 99 e lá descobre o nome correto da atriz. Entretanto, não existe realmente ficha alguma com os filmes estrelados por esta atriz. O fichário está desatualizado.
A atendente procura nas estantes alguns filmes e lembra-se de dois: *O Crime Perfeito* e *Seven* e mostra-os ao cliente. O cliente recusa-se dizendo que não gosta do gênero destes filmes e pergunta pelo filme vencedor do Oscar.
A atendente responde: - *Shakespeare Apaixonado?* Ainda não saiu em vídeo.

Cena 2: *O cliente procura por filmes de um certo gênero*
Agentes: Cliente, Atendente, Balconista
Ações:

Cliente: - *Eu gostaria de comprar um filme de ação*.
Atendente: - *Nós apenas alugamos.*
Cliente: - *Tudo bem. Então, por favor, me dê algumas dicas de filmes de ação.*
Atendente: *Com algum ator especial.*
Cliente: *Pode ser com Chuck Norris, Van Dame, Statellone, Charles Bronson*
Atendente: *Temos estes aqui.*
A atendente apresenta dez filmes. O cliente escolhe cinco e fica em dúvida se já assistiu outros três. Ele também pergunta à atendente se os outros dois filmes são bons. Após conversar durante alguns minutos com a atendente que entende muito do gênero, decide ficar com seis fitas. A atendente encaminha o cliente à balconista para calcular o valor da locação e o prazo de devolução. Após consultar as tabelas de preços e prazos, a balconista apresenta três planos de pagamento.
Balconista: - *Se você devolver em um dia, paga apenas R$ 6,00. Se devolver em seis dias paga R$ 12,00 e se devolver em uma semana paga R$ 15,00. Após este prazo paga mais R$ 2,00 por fita, por dia*.

Cena 3: *Cliente procura por filme usando o sistema de consulta*
Agentes: Cliente e Sistema de Consulta
Ações:

> João gostaria de assistir a um filme de guerra. Ele vai a uma locadora de fitas e, chegando lá, utiliza um sistema de consulta. Ele não lembra o título nem o diretor, mas sabe que se passava na guerra do Vietnã e lembra bastante da trilha sonora. Utilizando o sistema ele solicita as opções de filmes de guerra. Os sistema oferece a ele as possibilidades de escolher os filmes por local da guerra ou por época. João escolhe a sua opção (guerra) e obtem uma lista com dezenas de filmes sobre o vietnam. Ele pode organizar esta lista, por diretor, atores e título. Na tela do sistema aparecem a ilustração com o cartaz de divulgação do filme e uma opção para ele visualizar o trailler. O sistema, entretanto, não possibilita ele ouvir a trilha sonora.
>
> Após analisar algumas opções ele finalmente encontra o filme desejado, mas uma informação na tela do sistema indica que o filme está alugado. João quer saber quando o filme será devolvido, mas esta informação não consta no sistema. Ele entretanto pode reservar e o sistema enviará para ele um e-mail avisando quando o filme estiver disponível. Ele sai da loja pensando "Que tempo perdido. Seria melhor que eu pudesse consultar o sistema de casa".

4.3.2 Questionamento Sistemático

A descrição de informações do domínio através de narrativas só é efetivamente realizada se o processo de compreensão por parte dos analistas e projetista for realizado de maneira sistemática [J. Carroll et al. 94]. O **questionamento sistemático** é uma técnica que permite ao analista obter, a partir dos cenários, uma rede de proposições na qual identifica-se **agentes (atores), ações (processos de casos de uso), e objetos (informações).**

O **questionamento sistemático** uma técnica psico-linguística que permite a psicólogos e lingüistas examinar o conteúdo e a estrutura de informações contidas numa narrativa. Uma narrativa é um sumário de um conjunto de eventos e ações envolvendo agentes e objetos do mundo. Nem todas as informações integrantes do contexto são passadas através da narrativa. Muitas dessas informações são inferidas do conhecimento cultural de cada indivíduo. Outras, entretanto, precisam ser obtidas diretamente na fonte, isto é, junto ao autor da narrativa. Essa técnica foi desenvolvida para se entender melhor o processo de compreensão de estórias em narrativas. O objetivo é compreender tudo o que envolve o contexto daquilo que está sendo passado na narrativa.

Aplicando essa técnica no processo de análise de domínios, os especialistas devem descrever em narrativas seu conhecimento do domínio. Entretanto, esse conhecimento é muito mais abrangente. O questionamento sistemático permite obter todo o conhecimento que está além do que foi comunicado na narrativa. Assim, analistas e projetista podem utilizar essa técnica para adquirir mais eficazmente conhecimento sobre o domínio e inferir objetos e interações que não estão descritos na narrativa. Isto ocorre usando-se cada sentença ou afirmação da narrativa como ponto de entrada na complexidade do problema.

Nessa técnica, cada questão é uma ponte entre uma idéia e outra. Uma resposta a uma questão sobre um componente da narrativa revela outras conexões que são críticas para o seu entendimento. Realizando, sistematica e exaustivamente muitos tipos de questões sobre componentes da narrativa e iterativamente fazendo perguntas sobre as respostas geradas, o analista elabora um mapa conceitual (rede de proposições) que representa as estruturas do conhecimento do domínio.

Os passos do método consistem de:

1. **Geração do cenário** - as narrativas que compõe o cenário devem ser decritas pelo especialista no domínio. O analista pode motivá-lo fazendo perguntas como num processo convencional de entrevista (questões de elicitação do cenário).
2. **Elaboração da rede de proposições** - as narrativas devem ser simplificadas e expressas através de proposições.
3. **Análise** - a partir das proposições pode-se determinar as **tarefas** (ações e objetos) e **usuários** (agentes das ações).
4. **Questionamento sistemático** - novas proposições podem ser elaboradas através de questões que são feitas sobre elementos das proposições anteriores, num processo iterativo.

Nos passos iniciais obtem-se informações sobre agentes, interações e objetos abstratos do domínio. Usando a técnica, pode-se chegar a um conhecimento mais profundo do domínio que permite aos analistas e projetista a elaboração de modelos funcionais do sistema.

Exemplo
Considere o seguinte cenário sobre um caixa eletrônico.
Questão de elicitação do cenário:

> *Como posso sacar R$ 100,00 do caixa eletrônico?*

Cenário:

> *Primeiro ponha o seu cartão do banco no caixa eletrônico, digite a sua senha e pressione o botão "saque rápido". Depois pegue o dinheiro*.

As duas sentenças do cenário acima contém quatro proposições:

> CLIENTE -- põe -- CARTÃO
> CLIENTE -- digita -- SENHA
> CLIENTE -- pressiona -- BOTÃO-DE-SAQUE-RÁPIDO
> CLIENTE -- pega -- DINHEIRO

A partir dessas proposições, o analista determina que o cartão e o cliente são agentes de ações. Numa análise voltada para a elicitação de requisitos da interação, seria determinado como usuário do sistema, o cliente. A ações são portanto: digita, pressiona e pega. São objetos da interação a senha, o botão e o dinheiro. Outros objetos são o caixa eletrônico e o cartão. É preciso determinar que tipo de objetos são esses. Uma outra dúvida é a respeito do cartão ser objeto ou agente.

Obviamente, como esse exemplo é bastante simples e a aplicação também é muito conhecida, parece desnecessário obter mais conhecimentos a respeito. Entretanto, como o objetivo aqui é exemplificar a técnica, mostraremos como pode-se questionar a respeito dessa aplicação.

O analista deve então realizar uma série de questões sobre as proposições. Nesse questionamento o analista precisa determinar qual o relacionamento entre a resposta e a proposição que originou a pergunta.

As questões da categoria por que, visam responder "razões" (metas) e "causas" a respeito de eventos da narrativa. As questões da categoria como oferecem maiores detalhes a respeito de determinados eventos, permitindo determinar sub-tarefas e maiores detalhes sobre a interação. Questões da categoria o que podem ser feitas sobre objetos, e revelam atributos e hieraquias de objetos. Perguntas de verificação podem ser feitas para se saber se as proposições estão sendo entendidas corretamente. As perguntas de verificação são as que têm resposta sim/não. Uma taxonomia mais completa ainda está sendo pesquisada pelos autores do método.

Continuando o exemplo anterior a tabela abaixo descreve uma seção de questionamento sistemático:

Questões "por que"

Por que o cartão entra no caixa eletrônico?
_ Para iniciar. (evento conseqüente)
_ E então a máquina saberá quem é o cliente. (estado conseqüente)
Por que o cliente digita sua senha?
_ Para provar que ele é o usuário autorizado. (meta)
Por que o cliente pressiona o botão de saque rápido?
_ Porque é o botão para saques de R$ 100,00 (critério de descriminação).
_ Para evitar a digitação (cenário alternativo).
_ Porque ele está com pressa (causa)

Questões "como"

Como o cliente põe o cartão?
_ O cliente tira o cartão de sua bolsa e
_ insere no local apropriado no caixa eletrônico..
Como o cliente digita a senha?
_ Pressionando os botões adequados.
Como o cliente pressiona o botão de saque rápido?
_ Colocando o seu dedo nele.

Questões "o que"

O que é um botão de saque rápido?
_ É um tipo de botão que se pode pressionar.

O que é um botão?
_ É um dispositivo de controle no painel do caixa eletrônico.

Observe que se o analista estivesse utilizando essa técnica para num método orientado a objetos, outras questões poderiam ser realizadas com outros objetivos de acordo com as necessidades do método, como, por exemplo, para determinar classes e hierarquia de classes.

Após os cenários estarem desenvolvidos, os analistas devem trabalhar em conjunto com os usuários para avaliá-los e refiná-los. Isto pode ser feito, por exemplo, colocando-se *posters* numa sala pela qual os usuários podem circular livremente e observar os diversos cenários. Cada cenário deve apresentar quadros (desenhos, gráficos, fotografias, etc.), usando *storyboards* por exemplo, e uma descrição narrativa da tarefa. Os usuários, munidos de papeis e lápis, podem fazer anotações (críticas e sugestões) e anexá-las a cada *poster*.

Considerações finais sobre cenários

O resultado da modelagem através de cenários é uma base para a compreensão de quem são os usuários, quais a tarefas envolvidas e quais funções a interface e a aplicação devem prover, de maneira que, já se possa ter meios de garantir a usabilidade do sistema.

A idéia de cenários em análise não está necessariamente associada à técnica de questionamento sistemático. Os cenários são extremamente úteis para descrição do domínio. A técnica sistematiza o processo de compreensão do conhecimento adquirido.

Os métodos em geral, e esse não deve fugir à regra, devem ser usados, não como uma camisa-de-força que limite o processo de análise, mas como ferramentas que orientam os analistas e aumentam sua capacidade intelectual.

4.4 Análise de Usuários

Para que o sistema seja construído como uma ferramenta de apoio às atividades de usuários no domínio de aplicação, é preciso conhecer quem são os usuários e quais as suas necessidades, isto é quais tarefas eles necessitam realizar. A análise de usuários deve determinar quem eles são e quais tarefas eles normalmente fazem no domínio. Ela envolve a descrição dos diferentes papéis de usuários e qual o conhecimento, capacidade e cultura possuem os futuros usuários do sistema.

4.4.1 Análise de Perfis de Usuários

A análise do perfil dos diversos usuários do sistema descreve as várias características que podem influenciar as decisões dos projetistas no desenvolvimento do sistema. Os objetivos são assegurar que certas propriedades do sistema estejam adequadas ao conhecimento, cultura e capacidades do usuário, e que potenciais deficiências sejam levadas em consideração. Por exemplo, para um software de acompanhamento de pacientes em hospital, certos termos específicos da medicina devem ser incluídos nas telas do sistema e devem ser evitados termos técnicos de informática (*forneça informações patológicas* ao invés de *entrar*

dados da doença). Para usuários com alguma deficiência física ou motora, elemento da interface devem ser modificados, como por exemplo, tela de maior tamanho e letras maiores para deficientes visuais.

Os perfil do usuário pode ser analisado através de formulários do tipo:

Perfil do Usuário

Nome do Sistema: _____
Função do Usuário: _____

Conhecimento e Experiência do Usuário

Nível educacional
( ) Ensino fundamental
( ) Ensino médio
( ) Graduação
( ) Pós-Graduação

Língua Nativa
( ) Português
( ) Inglês
( ) outra: _____

Nível de leitura e expressão
( ) Excelente
( ) Bom
( ) Regular
( ) Ruim

Experiência com computadores
( ) Iniciante
( ) Intermediário
( ) Experiente

Experiência com sistema similar
( ) Utiliza bastante um similar
( ) Já utilizou um similar
( ) Nunca utilizou um similar

Conhecimento sobre o domínio
( ) Elementar
( ) Intermediário
( ) Especialista no domínio

Características Físicas

Manipulação
( ) Canhoto
( ) Destro
( ) Ambidestro

Deficiências
( ) Auditiva
( ) Visual
( ) Corporal
( ) Vocal

O perfil deve dar as informações necessárias que os desenvolvedores precisam para tomar as suas decisões. A análise do perfil pode ser adaptada de acordo com as características do sistema e com as necessidades de analistas e desenvolvedores. Por exemplo, pode ser interesse dos designers saber se os usuários têm algumas experiência com interfaces gráficas e qual o padrão (Windows, Motif, Macintosh, etc.) eles estão acostumados a utilizar.

4.4.2 Papéis de Usuários

O papel (ou função) específico de cada usuário é definido pelas tarefas que eles realizam. Numa organização, as pessoas trabalham juntas, de maneira estruturada. A estrutura da organização define relacionamentos entre as pessoas. A implicação imediata dos diferentes papéis de cada usuário são as diferentes

tarefas que eles irão realizar. Algumas tarefas podem ser comuns a diferentes papéis de usuários, enquanto outras podem ser exclusivas de papéis específicos.

O conceito de papel de usuário permite abstrairmos as características específicas de um usuário e enfatizar nas diferentes necessidades associadas a função que ele exerce. Para cada papel devemos associar um conjunto de funções, como veremos mais adiante.

No domínio do departamento de informática da UFRN podemos identificar como papéis de usuários: *secretário do departamento, chefe, coordenador de graduação, secretário da coordenação, coordenador de pós-graduação, professor, aluno, funcionário de administração de laboratórios e usuário externo*.

4.5 Análise e Modelagem de Tarefas

Os cenários permitem o levantamento e a descrição mais global das informações, das tarefas e dos problemas do domínio. O perfil de usuário descreve as características de usuários em termo de conhecimento, cultura, capacidades e limitações. A análise de tarefas visa determinar quais as atividades que os usuários (ou cada papel de usuário) devem realizar. Esta informação é essencial para se especificar os requisitos funcionais, determinando a funcionalidade do software. Para que o sistema possa ser construído para que estes problemas sejam resolvidos, ele deve ser uma ferramenta auxiliar na realização das tarefas de cada usuário.

Uma tarefa é, genericamente, uma atividade na qual um ou mais agentes tentam atingir uma meta especificada, através de uma mudança de estado em uma ou mais entidades do mundo. Num domínio de aplicação, as tarefas são as atividades que modificam estados de elementos deste domínio. A construção de um sistema computacional em um certo domínio tem por objetivo apoiar a realização de algumas destas atividades. Durante o processo de análise, deve-se determinar quais as do domínio e identificar quais delas podem ser auxiliadas pelo sistema.

A análise e modelagem de tarefas visa descrever as principais as tarefas que cada usuário do sistema terá de realizar para que os projetistas possam elaborar quais funções o sistema deve oferecer para que elas possam ser desempenhadas. Estas tarefas são descritas em termos de metas e um planejamento de possíveis atividades necessárias para atingi-las. As tarefas podem ser descritas a partir das informações obtidas nos cenários e devem ser agrupadas por **papéis de usuário**.

A análise de tarefas pode ser utilizadas nas diversas fases do ciclo de vida do software. Na fase de análise de requisitos ela pode ser utilizada para identificar problemas na maneira de como as tarefas vêm sendo realizadas. Os modelos de tarefas também podem descrever quais tarefas podem ser realizadas com o auxílio do sistema e como os usuários gostariam que ela fosse realizada. A análise de tarefa também é utilizada na avaliação do sistema para se determinar se como os usuários estão utilizando o sistema e se os objetivos foram atingidos. Nestes casos, a análise de tarefas ajuda ao projetista da interface ter uma visão da aplicação sob a perspectiva do usuário, isto é, um modelo das tarefas do usuário quando executando sessões da aplicação.

4.5.1 Modelo de tarefas como base para a especificação de requisitos funcionais

A análise e modelagem de tarefas pode ser utilizada como base para a especificação de requisitos funcionais. Para isto é preciso descrever as metas associadas a cada papel de usuário, que permitirão saber o que os usuário querem.

Resultados da psicologia cognitiva mostram que as pessoas realizam tarefas estabelecendo **metas** e elaborando um **plano** para cada uma delas. O **planejamento** consiste numa decomposição hierárquica de metas em **sub-metas** até que elas possam ser atingidas por **operações**. O **plano** ou **método** é, portanto, uma estrutura de sub-metas e operações para se atingir um certa meta. As **operações** correspondem a *ações básicas humanas*, isto é, aquelas que qualquer pessoa pode e sabe como realizar. São exemplos de operações *escrever uma palavra ou frase, ler uma informação, falar uma palavra ou frase, andar, relembrar, mover um objeto, pressionar um botão de controle* e várias outras.

Por exemplo, suponhamos que uma pessoa tem como meta *avisar a um amigo, através de uma carta, que sua filha nasceu*. Para atingir seu objetivo a pessoa deve estabelecer duas sub-metas: E*screver a carta e Colocá-la no correio*. A sub-meta escrever carta pode ser atingida pelo método: *Conseguir papel e lápis* e *Escrever mensagem*. Escrever mensagem já pode ser considerada uma operação, enquanto que conseguir papel e lápis requer um novo planejamento que determine as seguintes operações: *ir até o escritório, abrir o armário, pegar o papel e o lápis, levá-los até a mesa*.
O grão de refinamento do que podemos considerar com sendo uma operação é bastante subjetivo. Vai depender das dificuldades de quem realiza o plano. Na prática o plano é necessário quando a pessoa que vai realizar as ações não sabe ainda qual o método. Com a experiência o método torna-se automático e podemos considerar uma sub-meta como uma operação

Na utilização de um sistema computacional, os usuários realizam tarefas com o objetivo de atingir certas metas. Uma meta é um determinado estado do sistema ou de objetos do sistema ao qual o usuário quer chegar. Ao estabelecer a meta o usuário deve realizar um planejamento decompondo a meta em sub-metas até que ele saiba que existe uma determinada função do sistema que permita que sua meta seja atingida. O caso agora é um pouco diferente do planejamento anterior, pois não é o usuário quem vai realizar a operação desejada, mas o sistema. A decomposição deve ocorrer até que ele identifique que o sistema tem uma determinada função que quando executada realiza a operação necessária para que sua meta seja atingida. Chamamos estas operações que o sistema executa para satisfazer as metas do usuário de **função da aplicação**. O conjunto de funções da aplicação determina a funcionalidade do sistema.

Vejamos um exemplo. Suponha que o usuário esteja escrevendo uma carta utilizando um editor de texto e tenha como meta *formatar um documento*.Para atingir esta meta ele estabelece as seguintes sub-metas: *Formatar página, Formatar parágrafos, Formatar fontes*. Para cada uma destas sub-metas ele estabelece novas sub-metas até que ele identifique no software funções que o sistema pode realizar que permitam que as sub-metas sejam atingidas. Por exemplo, formatar página pode ser decomposta na função da aplicação *especificar tamanho da página* e na sub-meta *definir margens*. Esta última por sua vez pode ser atingida pelas operações *especificar valor da margem superior, especificar valor da margem inferior, especificar valor da margem esquerda e especificar valor da margem direita.*

Vejamos o plano deste usuário

META: Formatar documento
PLANO:
Formatar página (sub-meta)
especificar tamanho da página (função da aplicação)
Definir margens (sub-meta)
especificar valor da margem superior (função da aplicação)
especificar valor da margem inferior (função da aplicação)

especificar valor da margem esquerda (função da aplicação)
especificar valor da margem direita (função da aplicação)
Formatar parágrafos (sub-meta)
selecionar parágrado (função da aplicação)
Especificar atributos do parágrafo (sub-meta)
especificar espaçamento (função da aplicação)
especificar recuos (função da aplicação)
...
Formatar fontes (sub-meta)
...

O modelo de tarefas é extremamente útil para determinarmos as metas dos usuários e quais funções da aplicação eles gostariam que o sistema oferecesse. Por exemplo, a especificação dos requisitos pode determinar que deve existir uma função da aplicação para formatar documento de maneira que a meta do usuário pudesse ser atingida pelo sistema sem a necessidade de planejamento algum.

É importante ressaltar que uma meta pode ser satisfeita por uma única ou por várias funções da aplicação e que também mais de um método pode ser atingido uma mesma função da aplicação.Por exemplo, ao utilizar o Word 7.0, o usuário pode ter como meta *formatar um estilo*. Ao construir o seu plano o usuário em determinado momento pode estabelecer a sub-meta Especificar atributos do parágrafo. Esta sub-meta requer as mesmas funções de aplicação do plano para a meta formatar parágrafo. Assim, temos um grupo de funções da aplicação que são utilizadas para duas (ou mais) metas distintas.

Para que o usuário solicite ao sistema que execute uma determinada função de usuário, ele deve realizar operações que correspondam a um **comando de função**. Por exemplo, para o usuário solicitar ao sistema operacional que realize a função de copiar um arquivo de um diretório para outro ele deve escrever um comando do tipo **copy nome1 dir1 dir2** ou se estiver numa interface gráfica, mover o ícone correspondente ao arquivo da janela do diretório para a do outro diretório. Ao realizar o comando, o usuário precisa realizar operações com os dispositivos de interface do sistema, como *pressionar mouse, digitar número, teclar enter, etc.*

Apenas com a descrição das operações do usuário é que um plano para atingir uma meta fica completo. Quando o sistema está pronto, o plano tem que determinar exatamente as operações necessárias para comandar a função e, conseqüentemente, ter a meta atingida com o auxílio do sistema.

Na especificação de requisitos é suficiente decompor as metas que o usuário quer atingir nas correspondentes sub-metas. Caberá ao designer do software determinar qual o conjunto de funções que permita atingir o maior número possível de metas para cada papel de usuário e quais devem ser os comandos de interface para cada uma das funções.

4.5.2 Modelagem de Tarefas usando GOMS

Neste curso, utilizaremos a análise de tarefas na especificação de requisitos para determinar as tarefas que os usuários necessitam realizar com o sistema a ser construído. Para isto utilizaremos um método específico que utiliza o modelo **GOMS** simplificado. O modelo **GOMS** (*Goals, Operators, Methods, and*

Selection Rules) oferece uma abodagem de análise da tarefa baseada num modelo do comportamento humano que possui três subsistemas de interação: o **perceptual** (auditivo e visual), o **motor** (movimentos braço-mão-dedo e cabeça-olho), e **cognitivo** (tomadas de decisão e acesso a memória). O modelos GOMS descreve o comportamento dinâmico da interação com um computador, especificando-se:

Metas - Uma aplicação é desenvolvida para auxiliar os usuários atingirem metas específicas. Isso requer uma série de etapas. Dessa forma, uma meta pode ser decomposta em várias submetas, formando uma hierarquia.

Operadores - São as ações humanas básicas que os usuários executam.

perceptual - operações visuais e auditivas (*olhar a tela, escutar um beep*).

motor - movimentos braço-mão-dedo e cabeça-olho (*pressionar uma tecla).*

cognitivo - tomadas de decisão, armazenar e relembrar um item da memória de trabalho.

Métodos - Um método é uma sequência de passos para se atingir uma meta. Dependendo do nível da hierarquia, os passos num método podem ser submetas, operadores ou a combinação de ambos.

Regras de Seleção - Pode existir mais de um método para se atingir uma meta. Uma regra de seleção especifica certas condições que devem ser satisfeitas antes que um método possa ser aplicado. Uma regra de seleção é uma expressão do tipo "condição-ação".

O GOMS permite que se construa modelos de tarefas bem mais complexos e detalhados do que o necessário numa tarefa de análise para a especificação de requisitos. Usaremos uma versão simplificada do GOMS, pois:

- o modelo da tarefa não deverá descrever informações de design da interface, uma vez que ela ainda não foi construída;
- o analista não é um especialista em psicologia cognitiva;
- o modelo simplificado pode ser expandido para o original, o que é bastante útil.

1. ***Diretrizes do Modelo de Tarefas Simplificado***

Uma vez que a hierarquia de metas representa um aumento no nível de detalhe, se nos limitarmos à descrição de metas e submetas de mais alto nível, nenhuma decisão de design será envolvida.

- **Faça a análise "top-down"** - comece das metas mais gerais em direção as mais específicas.
- **Use termos gerais para descrever metas** - não use termos específicos da interface.
- Examine todas as metas antes de descer para um nível mais baixo - isso facilita reuso de metas.
- **Considere todas as alternativas de tarefas** - as regras de seleção possibilitam representar alternativas.
- **Use sentenças simples para especificar as metas** - estruturas complexas indicam a necessidade de decomposição da meta em submetas.

- **Retire os passos de um método que sejam operadores** - os operadores são dependentes da interface.
- **Pare a decomposição quando as descrições estiverem muito detalhadas** - quando os métodos são operadores ou envolvem pressuposições de design.

1. ***Modelagem da Tarefa para aplicações com múltiplas funções de usuários.***

   Se, para uma determinada aplicação, a função do usuário for um fator crítico dominante na análise de usuários, deveremos ter modelos de tarefas diferentes para cada função de usuário. No GOMS simplificado, veremos como representar as tarefas para cada usuário num só modelo. Antes de estudarmos a notação do modelo, vejamos algumas regras para modelos com múltiplas funções de usuários:

   - Inicie especificando metas de alto nível para cada função de usuário.
   - Se mais de um compartilham a mesma meta, agrupe-os sob uma só.
   - Se todos compartilham a mesma meta, retire as referências a funções de usuário.

2. ***Notação***
   1. Notação para Funções de Usuários.

   - Funções de usuários distintas serão denotadas pelo símbolo **FU** seguido por um número.

     ex.: Gerente de vendas (**FU1**), balconista (**FU2**), caixa (**FU3**)

   - A descrição de cada função de usuário é a primeira parte do modelo de tarefas.

     ex.: Gerente de vendas (**FU1**): responsável pela vendas nas lojas. Tem acesso a todos os dados do sistema.

   - Um ponto-e-vírgula (;) é usado para separar o símbolo da função do usuário do restante da notação do modelo de tarefas.

     ex.: FU1;...

   - Se uma meta é usada para mais de uma função de usuário, elas devem ser separadas por uma vírgula (,).

     ex.: FU1, FU2; ...

   - Um asterisco (*) é usado se uma meta é aplicada a todas as funções de usuários.

     ex.: *;...

1. **Notação para especificação de metas**

- Cada passo num método é numerado numa ordem seqüencial, com cada nível de decomposição separado por um ponto (.), e com a endentação apropriada para reforçar a noção de hierarquia. Após o último número usa-se dois-pontos (:).

    ex.: FU2; 2.1: Anotar correções.

- Um comentário começa com duas barras inclinadas para direita (//) e acaba ao final da linha.

    ex.: // Para fazer as anotações o balconista deverá examinar as
    // listagens produzida durante as vendas do dia.

    FU2; 2.1: Anotar correções

- **Notação para Métodos e Regras de Seleção**

- Se uma meta possui mais de um método para ser atingida, uma letra do alfabeto é usada de forma ordenada após o número que descreve a meta.

    ex.:

    FU2; 2: Fazer relatório de vendas (meta)

    FU2; 2A: ... (método A)

    FU2; 2B: ... (método B)

- As regras de seleção e o método associado são descritos como pares "condição-ação", logo após a notação numérica da meta.

    ex.: FU2; 2A: se (dia de hoje for sábado)

    então (fazer relatório semanal)

1. **Reutilizando Metas**

   Um aspecto importante dessa notação é que pode-se reutilizar metas, simplesmente usando o mesmo número de uma meta preexistente.

ex.:

FU2; 2.1: Anotar correções. (meta preexistente)

...

FU1, FU3; 3: Modificar livro-caixa.

FU1, FU3; 3.1: Procurar lotes em aberto.

FU1, FU3; 2.1: Anotar correções. (meta reusada)

FU1, FU3; 3.3: Recalcular valores.

2. **Diretrizes adicionais**

- Delimite os passos de um método entre chaves: {}.
- Em aplicações com apenas uma função de usuário, não é necessário incluir a notação de função usuário antes de cada meta.
- Num nível de decomposição onde todas as metas têm o mesmo número-prefixo, apenas o número que indica a seqüência naquele nível é necessário.
- A diretriz anterior se aplica também à notação de função de usuário.
- Ao reutilizar uma meta anterior é necessário usar a notação completa para ela.

ex.: FU1, FU3; 3: Modificar livro-caixa.

1: Procurar lotes em aberto.

2.1: Anotar correções. (meta reusada)

3: Recalcular valores.

5. Design Conceitual de Software

Neste capítulo discutiremos:

5.1 O que é *design* de software?

No capítulo anterior vimos como especificar os requisitos de software. Os requisitos funcionais determinam quais funções os usuários necessitam que o sistema ofereça. Vimos também como estas funções de aplicação podem ser especificadas como Casos de Usos na linguagem UML.

Para que os casos de uso tornem-se funções do software é preciso determinar como o usuário irá interagir com elas e como elas devem ser implementadas em um programa. Por exemplo, vamos considerar que o *cálculo do lucro de venda* é um caso de uso que faz parte dos requisitos funcionais de um sistema comercial qualquer. O caso de uso deve descrever, por exemplo, que o usuário deve fornecer o valor de compra e o valor de venda, acionar a função de cálculo e o sistema deve então fornecer o resultado calculado. Para que este (e outros) caso de uso seja concretizado numa função de aplicação do software é preciso projetar e implementar como os usuários fornecerão os valores e obterão os resultados e quais são os algoritmos que realizam tudo isto. No capítulo 1, mostramos o exemplo de um software bastante simples para o cálculo do lucro de vendas. Lá apresentamos como este caso de uso ocorre num software descrevendo a interface com a qual o usuário deve interagir e o algoritmo que implementa este programa.

A partir de agora vamos começar a estudar como a especificação de requisitos pode ser transformada na especificação do software. Devemos determinar como casos de uso devem ser transformados em funções da aplicação com as quais os usuários possam interagir e que possam ser implementadas através de uma linguagem de programação. A esta atividade de criar o software que satisfaz os requisitos dos usuários denominamos de **design de software**.

Design compreende as atividades de concepção, especificação e prototipação de um artefato. O software como um produto - um artefato virtual - precisa ser concebido, especificado e prototipado. Precisamos trazer para o desenvolvimento de software a atividade de *design* que é realizada no desenvolvimento de qualquer produto industrial.

Design poderia ser traduzido tanto por projeto como por desenho. Entretanto, estes dois termos não expressam exatamente o que é design. Projeto é um termo mais abrangente do que design, pois se aplica a projeto de pesquisa, projeto de desenvolvimento de um produto e envolve planejamento, metodologia,

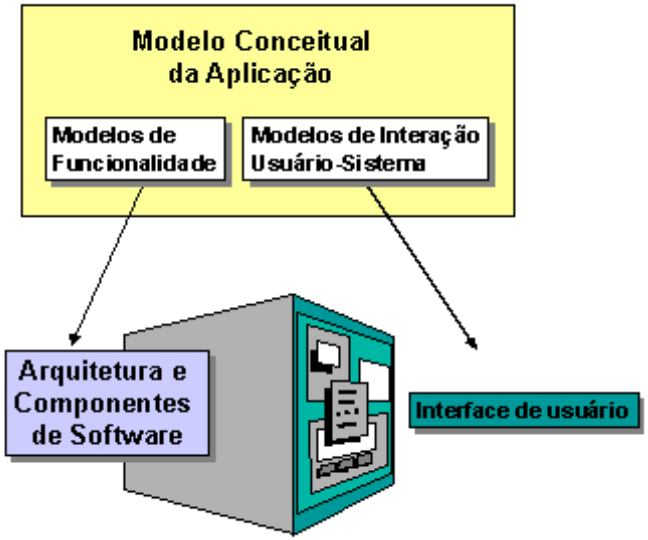
cronograma, recursos, etc. Desenho é uma tradução utilizada no sentido de Desenho Industrial (industrial design), mas leva a conotação de que a atividade resume-se a elaborar os diagramas que descrevem os modelos do produto. Por estes motivos vamos utilizar o termo em inglês: *design*. Para quem faz o design, vamos usar os termos *designer,* projetista ou desenhista. Segundo David Liddle,

> "*o design de software é o ato de determinar a experiência do usuário com um pedaço de software. Não tem nada a ver sobre como o código opera internamente ou se ele é grande ou pequeno. A tarefa do designer é especificar de forma completa e não ambígua a experiência global do usuário*". [David Liddle, *Design of The Conceptual Model*, em Winograd, T.]

O *design* de software compreende a concepção, especificação e prototipação da partes "externas" e "internas" do software. A parte externa compreende o **modelo conceitual da aplicação** e a **interface de usuário**. A parte interna compreende a **arquitetura de componentes de software** e os **algoritmos e estruturas de dados** que implementam estes componentes.

O design de software envolve as atividades de concepção, especificação e prototipação:

- do **modelo conceitual da aplicação**,
- da **interface de usuário**,
- da **arquitetura de componentes de software** e
- dos **algoritmos e estruturas de dados**



A concepção e especificação da arquitetura dos componentes de software determinará toda a estrutura interna do software: os objetos, funções e estruturas de dados. Eles são responsáveis pelo funcionamento do software.

Este e o próximo capítulo abordam o design da parte "externa" do software: o modelo conceitual e a interface de usuário. O design da parte "interna", a arquitetura de componentes de software e os algoritmos e estruturas de dados serão vistos no capítulo 6.

5.2 Modelo Conceitual da Aplicação

Vimos que o software deve ser visto como um artefato virtual que compreende as soluções que foram concebidas para resolver os problemas dos usuários, isto é, auxiliá-los na realização de suas tarefas no domínio. Este artefato virtual é uma entidade abstrata que existe na mente dos usuários quando eles estão interagindo com o usuário.

A esta entidade virtual que os usuários imaginam damos o nome de **Modelo Conceitual da Aplicação** que ele adquire. Este modelo conceitual determina, para o usuário, quais os **conceitos (ou objetos do domínio)** que estão representados no software, o que ele pode fazer com a aplicação - a **funcionalidade** - e como ele pode interagir com a aplicação - o **modelo de interação**.

A idéia de modelo conceitual da aplicação é denominado diferentemente por diversos autores. Ele é chamado de *virtualidade*, *modelo conceitual do usuário, modelo de usabilidade, metáfora de interface* e várias outras denominações.

Num editor de documentos como o MS Word, por exemplo, temos diversos conceitos que permitem ao usuário entender como está estruturado um documento e o que ele pode fazer com ele. No Word encontramos os conceitos de *página, bordas de página, margens, parágrafos, recuos de parágrafos, espaçamento entre linhas, distância-da-primeira-linha, distância-antes-do-parágrafo, tamanho da letra, estilo da letra* e diversos outros. Mais adiante apresentaremos um exemplo de um modelo conceitual para um editor de texto.

A elaboração de um modelo conceitual é fator determinante no sucesso de uma aplicação. O modelo conceitual precisa ser concebido e especificado adequadamente pelo designer, de acordo, com as necessidades, capacidade e limitações dos usuário. Para que o modelo seja adequado aos usuários, a especificação dos requisitos é fundamental. Nela encontramos, os papéis de usuários, o modelo de tarefas, os casos de uso, os conceitos do domínio e outras informações. A partir delas temos condições de elaborar um modelo conceitual que seja adequado aos requisitos dos usuários.

Para que a aplicação seja utilizada adequadamente é preciso que o usuário conheça o modelo conceitual da aplicação que foi elaborado pelo designer. Para isto é preciso que ele seja comunicado adequadamente ao usuário. Uma das formas de comunicar este modelo é através dos manuais de usuário ou de treinamento feito por pessoas especializadas. A outra forma é tentar expressar os conceitos através da interface de usuário. A interface é quem oferece a imagem externa do sistema e que deve ser interpretada de maneira correta pelo usuário. Ela deve comunicar para o usuário quais os objetos do domínio estão representados, o que ele pode fazer (a funcionalidade) com este conceitos e como ele pode interagir (o modelo de interação) com esta funcionalidade. No capítulo 5 veremos como o design da interface de usuário deve ser feito para expressar o modelo conceitual da aplicação.

5.2.1 Exemplo: o modelo conceitual de um sistema operacional

Ao utilizarmos um sistema operacional como o MS DOS sabemos que podemos construir, armazenar e organizar arquivos de software. Um arquivo é na verdade uma seqüência de bits como começo e fim definidos. Para o usuário o arquivo é algo conceitual (um conceito da aplicação) que se refere a documentos, programas, dados, etc. Esta entidade conceitual surge na mente do usuário por que ele tem um nome, um tipo, uma data de criação e uma tamanho em bytes. Arquivos são armazenados em discos em trilhas e setores e podem ser localizados a partir de uma tabela que associa o seu nome ao local físico. Para o usuário, arquivos são colocados "dentro" de uma outra entidade conceitual: o *diretório*.

Arquivos e diretórios são os conceitos da aplicação e podem ser criados, destruídos ou modificados por diversas funções da aplicação. Estas funções são, por exemplo, *copiar arquivo, apagar arquivos, construir diretórios, remover diretórios, colocar arquivos em um diretório, mover arquivo de um diretório para outro,* etc. Esta funções são também entidades virtuais representadas por uma nome que devem indicar, para o usuário, o que o sistema faz. Por exemplo, o sistema operacional não coloca arquivos dentro de um diretório. Isto só existe na mente do usuário. O que o sistema faz é trocar o endereço físico do arquivo na tabela de arquivos e diretórios. Entretanto, a visão de que diretórios são recipientes que armazenam arquivos é muito mais útil para o usuário. Ela permite ao usuário criar na sua mente conceitos da aplicação que permitem a ele entender o que fazer com o sistema e raciocinar melhor sobre ele.

O sucesso de uma aplicação vai depender justamente da criação deste modelo conceitual. Quanto mais claros forem o conceitos e as operações que se pode fazer com eles, mais o usuário vai saber como aplicá-los na resolução de seus problemas. O sucesso de sistemas como o Macintosh ou o MSWindows (que copiou o primeiro) deve-se à construção de modelos conceituais mais familiares para o usuário. Os conceitos de pastas e objetos e a representação deles através de janelas e ícones permite ao usuário criar imagens mentais mais claras e que são familiares as atividades que eles realizam num escritório. O conceito de pasta como recipiente é mais claro que o de diretório (que na verdade é uma lista). No ambiente Windows o usuário move objetos (documentos, aplicativos, figuras, etc.) de uma pasta para outra e para o *topo-de-mesa* (*desktop*) como faria no seus escritório.

No MS DOS, o usuário adquire este modelo conceitual (as noções sobre diretórios e arquivos) ao ler o manual de usuário. No sistema Unix, da mesma forma, é o manual do usuário quem mostra para o usuário quais os conceitos da aplicação. Veja como exemplo um trecho do manual do Unix onde são apresentados os conceitos de arquivos e diretórios.

A interface de usuário é fundamental para que este modelo conceitual seja comunicado ao usuário. Ela é o melhor meio para que expressar os conceitos da aplicação e as funções que podem ser utilizadas. No MS DOS, a interface é muito pouco expressiva. Na interface o conceito de diretório é visualizado por ele como uma lista de *nomes, tipos, tamanhos e datas*, que representam cada arquivo do diretório. Cada arquivo é referenciado individualmente pelo seu nome. No Windows 95, o modelo conceitual é representado por desenhos gráficos que representam os conceitos da aplicação de maneira mais clara.

A interface de usuário também tem o papel de permitir ao usuário interagir com o sistema, comandando as funções da aplicação e visualizando o estado dos conceitos do domínio. No caso do MS DOS, a interface oferece comandos como **copy, del, mkdir** e vários outros para que o usuário possa interagir com os conceitos do domínio através das funções da aplicação.

A concepção do modelo conceitual como vimos é fundamental para o sucesso da aplicação. A criação de uma interface de usuário que expresse uma imagem deste modelo é também outro fator importantíssimo para que usuários compreendam melhor o que podem fazer com o sistema. No capítulo 5, veremos como realizar o design de interfaces que expressem melhor o modelo conceitual da aplicação

5.2.2 Um modelo conceitual básico para sistemas interativos

Qualquer artefato quando utilizado por alguém permite que ele elabore mentalmente um modelo conceitual do artefato. Mesmo que este modelo não tenha sido especificado explicitamente, o usuário, ao utilizá-lo, cria na sua mente os conceitos a respeito daquela aplicação. É tarefa do designer, ao fazer o design de um produto, determinar quais os conceitos que estão associados a ele. Alguns produtos têm conceitos que são comuns a maioria deles. Os conceitos de *volume, brilho, cor e canal* são comuns a televisores. *Encher, lavar, enxaguar e centrifugar* são conceitos que se referem a operações que podem ser feitas em máquinas de lavar. Entretanto, não existe um obrigação de que todos os produtos possuam os mesmos conceitos.

Nos sistemas computacionais interativos o modelo conceitual da aplicação é formado por alguns tipos de conceitos que são comuns a maioria deles. Evidentemente, cabe ao designer determinar quais os conceitos são úteis para a aplicação que ele está projetando. Vamos utilizar alguns tipos pré-definidos de **conceitos**. Eles são os **objetos** e as **funções da aplicação** e os **comandos de função**. Os conceitos do domínio como objetos, propriedades e relacionamentos devem ser representados no sistema para que os usuários possam operar sobre eles. Estas operações - as funções da aplicação - permitem que os conceitos sejam criados, transformados, destruídos, etc. Os comandos de funções permitem ao usuário controlar a execução de uma função pelo computador.

O conceito de objeto da aplicação refere-se às representações abstratas das diversas coisas que podemos distinguir num domínio e que são utilizadas pelos usuários para realizar as suas atividades. Num domínio de biblioteca são exemplos de objetos do domínio *os livros, as revistas, o título do livro, os autores, o assunto, os números e volumes das revistas, o número de cadastro do livro, o nome do usuário, o número de cadastro do usuário* e diversos outros. Num domínio de edição de textos estes conceitos podem ser *documento, página, parágrafo, fonte*, etc. Os objetos do domínio possui propriedades que os caracterizam e podem estar relacionados entre si.

Ao fazermos o design de um software devemos determinar quais os objetos do domínio que serão representados como objetos da aplicação e quais novos poderão ser criados. Um sistema de biblioteca deve poder representar os objetos do domínio biblioteca listados acima. O objeto livro pode ser representado pelos conceitos da aplicação *informações-de-livro* de título, autor, assunto e número de referência.

Atenção! Na especificação do modelo conceitual não estamos interessados ainda em como eles vão ser representados computacionalmente, isto é, qual estrutura de dados será utilizada para representar o conceito numa linguagem de programação. Isto só deverá ser feito no design da arquitetura e dos componentes do software. Também não vamos representar ainda como o conceito será visto pelos usuário - se na forma de um ícone, de uma tabela, de uma lista, etc. Por enquanto estamos interessado em fazer uma especificação abstrata dos conceitos.

As **funções da aplicação** são responsáveis por operações com os conceitos do domínio. Elas devem determinar quais transformações ocorrem no sistema e quais são os conceitos que são modificados, criados ou destruídos. A funções modificam o estado do sistema. Um determinada função da aplicação **FA1** pode levar o sistema de um estado **E1** de para um estado **E2**. Os estado inicial que o sistema está antes que uma determinada função seja executado é também chamado de **pré-condição**. O estado final, isto é, após a função da aplicação ter sido executada é chamado de **pós-condição**.

Por exemplo, a função da aplicação *armazenar nome-do-cliente na lista-de-nomes* deve ter como pré-condição que o *nome-do-cliente* tenha sido fornecido e que a *lista-de-nomes* não possua o *nome-do-cliente* já armazenado. A pós-condição é que *nome-do-cliente* esteja armazenado na *lista-de-nomes* ao final da operação.

A execução de cada função da aplicação deve ser controlada por **comandos de função**. Este conceito refere-se ao conjunto de ações que o usuário deve desempenhar para que uma determinada função seja iniciada, interrompida ou cancelada, por exemplo. As ações que o usuário deve fazer são aquelas que ele desempenha com os dispositivos da interface de usuário, como *pressionar tecla X, escrever um nome com o teclado, pressionar com o mouse, mover com o*

mouse, etc. Existem diferentes formas de interação que determinam comandos de função de diversas natureza. No MS DOS, os comandos são elaborados pela digitação de sentenças. Este estilo é chamado de *linguagem de comando*. No Windows 95/98, os comandos são desempenhados pelo acionamento de objetos da interface num estilo chamado de *manipulação direta*.

Veremos, a seguir, como especificar os conceitos do domínio. Mais adiante veremos a especificação de funções da aplicação e na seção 5.3 veremos a especificação do modelo de interação que inclui os comandos de função.

5.2.3 Especificando o modelo conceitual da aplicação utilizando a UML

Modelos conceituais podem representar qualquer sistema na natureza. No sistema solar temos os conceitos de planetas, órbita, força gravitacional e outros. A física utiliza de notações matemáticas para representar os conceitos que descrevem o sistema solar. A língua portuguesa também pode ser utilizada para descrever conceitos de um domínio. Na biologia, os conceitos referentes a *espécies, gênero, família, classes, ordem, filo e reino* dos seres vivos são descritos em português e latim.

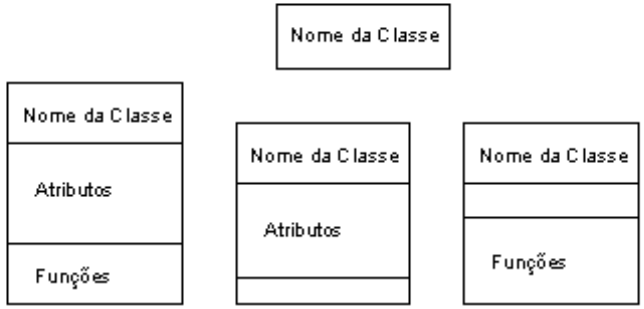
Não existem muitas técnicas ou notações específicas para o design do modelo conceitual de sistemas interativos. Neste modelo o que estamos querendo representar são os conceitos que se formam na mente do usuário ao utilizar o sistema. Algumas das técnicas e notações de inteligência artificial como *redes semânticas, frames e scripts* podem ser utilizadas. Seria interessante que a linguagem utilizada permitisse-nos construir um modelo que pudesse ser "traduzido" tanto em elementos da interface como em componentes de programação.

Utilizaremos a notação UML (Linguagem de Modelagem Unificada) como forma de representar conceitos de objetos e relações da aplicação. Com os diagramas de classes também podemos mostrar como os objetos estão associados com as funções da aplicação - que haviam sido representadas de maneira simplificada como **casos de uso** na UML.

Classes

Tradicionalmente, representamos um objeto dando-lhe um **nome** e descrevendo as **propriedades** ou **atributos** que os distinguem de outros objetos. As classes em UML permitem representar conceitos descrevendo o seu nome, seus atributos e relacionamentos. A classe descreve o conceito abstraindo os valores de propriedades específicos, permitindo a referência genérica a objetos do domínio. A noção de classes como o agrupamento de objetos está presente em nossa linguagem e em diversas ciências. O processo de classificação é estudado desde a filosofia da Grécia antiga.

Em computação existem diversas metodologias para análise e design orientado-a-objetos que permitem a representação de classes. Elas são bastante parecidas entre si. A UML apresenta sua própria notação para a representação de classes através de **diagramas de classes**. Estas notações tem por objetivo a modelagem de programas orientados-a-objeto. Entretanto, na atividade de design utilizaremos diagramas de classes como propósito de modelar objetos da aplicação da forma como são vistos pelo usuário e não da forma como são implementados.



Diversas formas de desenhar classes

Uma **classe** é uma descrição de abstrata de objetos que possuem **atributos** comuns ou estão **relacionadas** a outros objetos. Cada classe possui um **nome** que a distingue de outras classes. Este nome é qualquer seqüência de letras, de preferência um termo do vocabulário do domínio.

Um **atributo** é o nome dado a uma propriedade do conceito representado pela classe. Cada objeto que pertence a uma determinada classe possui **valores** específicos para os atributos que são comuns a uma determinada classe. Por exemplo, a classe *cliente* possui os atributos *nome, endereço, telefone e data-de-nascimento*. Um certo *cliente* (um objeto do domínio) é representado pelos valores específicos dos atributos que descrevem as propriedades. Os clientes *João da Silva, R. Salgado Filho, 2126666, 12/2/55* e *João da Silva, R. Hermes da Fonseca, 2122121, 29/2/65* são distintos, pois possuem valores específicos para os atributos que caracterizam a classe cliente. Nos diagramas de classes os atributos (o nome da propriedade) são identificados por um nome enquanto que os seus valores podem ser representados por números, nomes, caracteres especiais e outros símbolos que forem apropriados.

Um conceito pode ser representado por uma classe **simples**, isto é, que não possui algum atributo específico. A identificação numérica única de um aluno é um conceito que pode ser representado simplesmente pela classe *matrícula-do-aluno* sem nenhum atributo. Este conceito pode, eventualmente, ser um atributo de uma outra classe (a classe que representa o conceito *aluno*, por exemplo). Entretanto, podemos representar também a identificação por uma classe cujos atributos sejam *ano de ingresso, semestre, número do centro, número do curso, número pessoal* e os valores sejam, respectivamente *92-1-8-62-2321*.

Atenção! Os diagramas de classes da linguagem UML são mais genéricos do que precisamos para representar classes e objetos da aplicação. Eles foram projetados para permitir também a modelagem de classes de uma linguagem orientada a objetos como C++, Smalltalk ou Java. Por enquanto utilizaremos este diagrama apenas para representar objetos da aplicação. Utilizaremos diagramas de classes para representar classes de linguagens de programação mais adiante quando estivermos modelando a arquitetura de componentes de software.

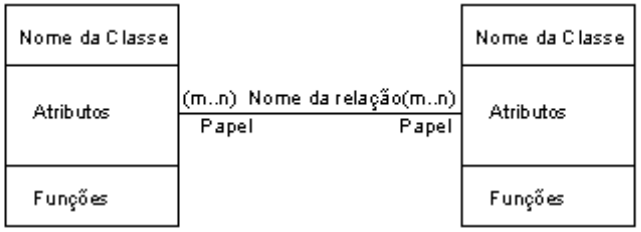
Objetos

Objetos são instâncias de classes. Objetos possuem existência num domínio. No exemplo visto anteriormente, *João da Silva, R. Salgado Filho, 2126666, 12/2/55* é um objeto da classe cliente. Os valores de atributos representam a existência de um cliente específico. *João da Silva, R. Hermes da Fonseca, 2122121, 29/2/65* é um outro objeto e é representado pelos valores de suas propriedades diferenciais.

Na UML um objeto é representado por retângulo, com o nome do objeto seguido por um ":"(dois pontos) e o nome da classe ao qual ele pertence.

Relacionamento

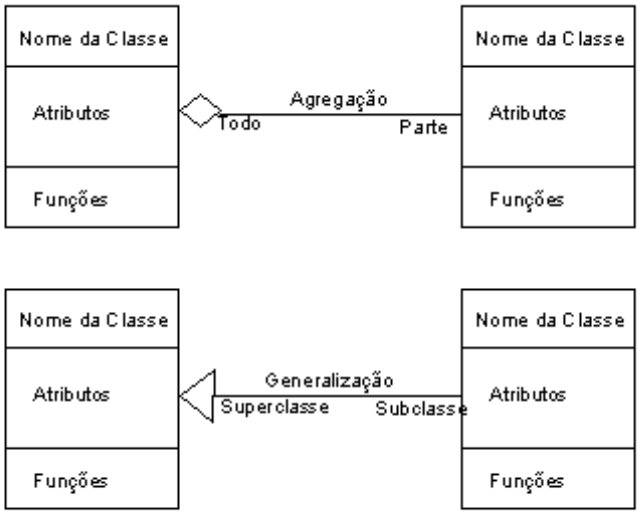
Uma classe pode estar relacionada com outras classes. Isto significa que os objetos representados pela classe estão relacionados de alguma forma com os objetos da outra classe. Uma relação genérica é representada por uma **associação**. Cada associação é identificada por um **nome**. As classes que estão ligadas a uma associação assumem **papéis** específicos. Por exemplo, a relação entre os conceitos *pessoa* e *empresa* pode ser representada pela associação *trabalha-em*. Nesta associação a pessoa assume o papel de *empregado* e a empresa o papel de *empregador*. Numa associação é preciso especificar a **multiplicidade**. Por exemplo, uma pessoa pode *trabalhar-em* uma única empresa (1) e uma empresa pode empregar uma ou mais pessoas (1 . . *).



Relacionamentos: Associação

Um importante tipo de relacionamento entre conceitos, bastante comum na maioria dos modelos, é a **generalização**. A generalização relaciona um conceito mais genérico a um conceito que seja mais específico. O primeiro é chamado de **superclasse** enquanto o conceito mais específico é a **subclasse**. A generalização é também conhecida como *é-um-tipo-de*. Por exemplo, um *retângulo* é-um-tipo-de *figura*. A figura é a superclasse e o retângulo a subclasse. *Círculo* e *polígono* são conceitos que também são subclasses do conceito *figura*. *Quadrado* é uma subclasse de retângulo e, conseqüentemente, também é subclasse de *figura*. A relação **especialização** é oposta a generalização. Ela indica que uma subclasse é uma especialização da superclasse. *Quadrado* é uma

especialização de *figura*.

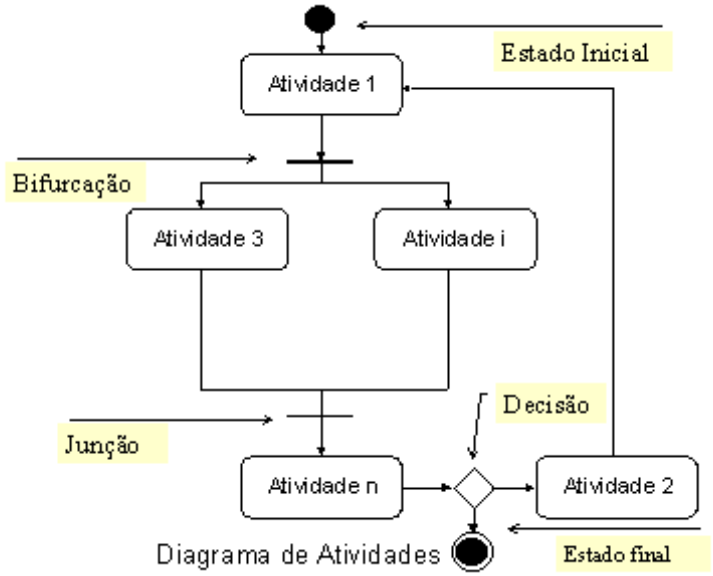


Um outro relacionamento bastante comum é a **agregação**, também conhecida como **parte/todo**. Esta relação ocorre entre conceitos que são parte de um outro (o todo) e pode ser representado pela associação *é-parte-de* ou pela sua relação inversa *tem*. Por exemplo, a *roda* *é-parte-de* *carro* ou, usando a associação inversa, *carro* *tem* *roda*. Nesta associação os papéis desempenhados pelas classes relacionados são, justamente, *parte* e *todo*. A multiplicidade também pode ser representada na associação de agregação. Uma *empresa* *tem* (0 . . *) *departamentos*, isto é uma empresa pode ter nenhum ou vários departamentos. Um *departamento* *é-parte-de* (1) *empresa*, significa que o departamento pertence a uma única empresa.

5.2.4 Especificando Funções da Aplicação usando Diagramas de Atividades

Os diagramas de **casos de uso** descrevem quais as funções da aplicação são de interesse de cada papel de usuário. Veremos agora como os **diagramas de atividades** permitem descrever o comportamento de cada função da aplicação. mais adiante mostraremos como os diagramas de classes permitem representar como as funções da aplicação estão relacionadas com os objetos da aplicação.

O diagrama de atividades é mais um diagrama da UML. Ele permite modelar aspectos dinâmicos de um sistema. Em particular, ele é um gráfico de fluxo (*flowchart)* que permite mostrar o fluxo de controle de atividade para atividade descrevendo a seqüência de passos no processo computacional.



Graficamente, um diagrama de atividades é um grafo composto por nós e vértices. Eles são utilizados para:

- Modelar uma função ou operação
- Modelar um fluxo de trabalho (*workflow*)

Por enquanto, vamos utilizar os diagramas para modelar apenas as funções da aplicação.
Normalmente, um diagrama de atividades é composto por:

- Estados de atividades e estados de ações
- Transições
- Objetos

Um diagrama de atividades visa mostrar com as coisas acontecem no sistema. Um **estado de ação** representa uma situação na qual o sistema encontra-se quando está realizando computações. Cada estado de ação é representado por uma figura específica com uma descrição da computação realizada. Esta descrição é o predicado de uma sentença do tipo *calcule valor, imprima documento*, etc.

Estados de ação são atômicos, isto é, eles não podem ser decompostos. Isto significa que quando a ação esta sendo executada ela não pode ser interrompida. Considera-se que o tempo de execução da ação é insignificante.

Já os **estados de atividades** podem ser decompostos e, eventualmente, representados em diagramas de atividades independentes se for necessário (normalmente por questões de clareza). Os estados de atividades não são atômicos e podem ser interrompidos durante a ocorrência de eventos. O sistema permanece num

estado de atividade durante um certo intervalo de tempo.

Um estado de atividade é composto por outros estados de atividades ou de ações. O estado de ações é, portanto, um estado de atividade atômico que não pode ser mais decomposto. O diagrama de atividades descreve o fluxo de controle do sistema nos diversos estados de atividade e ações.

A figura para representar o estado de atividade é a mesma do estado de ação, exceto que o primeiro pode ter partes adicionais tal como ações de entrada e saída (ações que estão envolvidas quando se entra ou sai do estado) e especificação de sub-máquinas.

Quando a ação ou atividade de um estado termina o fluxo de controle passa imediatamente para o próximo estado de ação ou de atividade. Este fluxo é especificado através de **transições** que mostram o caminho de uma ação ou atividade para o próximo estado de ação ou de atividade. Na UML, a transição é representada por uma linha com uma seta indicando a direção do fluxo.

Num diagrama de atividade existe uma transição que identifica a passagem do estado inicial para um estado de ação ou de atividade e uma outra que identifica a passagem para o estado final. O estado inicial é identificado por uma bola pequenina de cor preta e o estado final é representado por um círculo um pouco maior com a bola preta no seu interior.

A transição seqüencial entre os estados nem sempre ocorre por um caminho único. Caminhos alternativos são representados por uma **decisão** *(branching)*. Na UML a bifurcação é representada por um losângulo conectando linhas que representam as transições. No diagrama uma expressão lógica pode ser acrescentada para indicar em quais condições cada caminho pode ser seguido.

Dois estados de ação ou atividades podem ocorrer simultaneamente. Para indicar a simultaneidade ou concorrência a UML oferece as estruturas **bifurcação** e **junção**. Eles são representados por uma barra de sincronização que indicam quando uma transição se divide em duas ou mais ou quando duas ou mais juntam-se em uma. Uma bifurcação possui uma transição chegando e duas ou mais saindo. A junção, por sua vez, possui duas ou mais transições chegando e uma saindo para um outro estado de ação ou atividade.

Conceitualmente a bifurcação e a junção permitem representar dois estados de atividades ou ações simultâneos, isto é, sendo executando ao mesmo tempo. Na implementação esta concorrência pode ser real ou uma simulação com processos intercalados.

Raias são utilizadas para agrupar estados de atividades ou de ações de maneira a representar de quem é a responsabilidade por aquela ação.Cada raia representa um *locus* de atividade e é identificada por um nome.

Para modelar uma função da aplicação (ou uma operação) usando diagrama de estados você deve:

- Identificar os conceitos que estão envolvidos com esta função. Isto inclui os parâmetros da função e os atributos da classes que está associada à função. Os conceitos envolvidos com a função são denominados de **operandos**.
- Identificar as **pré-condições** no estado inicial da função e as **pós-condições** no seu estado final. Também identifique os **invariantes**, isto é, as condições que permanecem constantes durante a realização das atividades.

- Comece identificando as atividades iniciais que o sistema deve realizar para a função que está sendo modelada. Represente cada atividade que o sistema deve desempenhar como um estado de atividade, se ela puder ser decomposta, ou de ação se ela for atômica.
- Use decisão (*branching*) se necessário para especificar caminhos alternativos e repetições (*loop*).
- Use bifurcação e junção para especificar fluxos de controles paralelos (atividades simultâneas), se for necessário.

O diagrama de atividades é um tipo de máquina de estados. Ele tem por objetivo descrever os estados de atividades e o fluxo de controle, isto é, os caminhos para passar por cada um dos estados. A máquina de estados tem por objetivo descrever como o sistema (ou uma função em particular) muda de estados na ocorrência de eventos externos a ele. No diagrama de atividades descreve-se como o sistema muda de um estado de atividade para outro quando termina a atividade que está sendo realizada, independente da ocorrência de um evento externo.

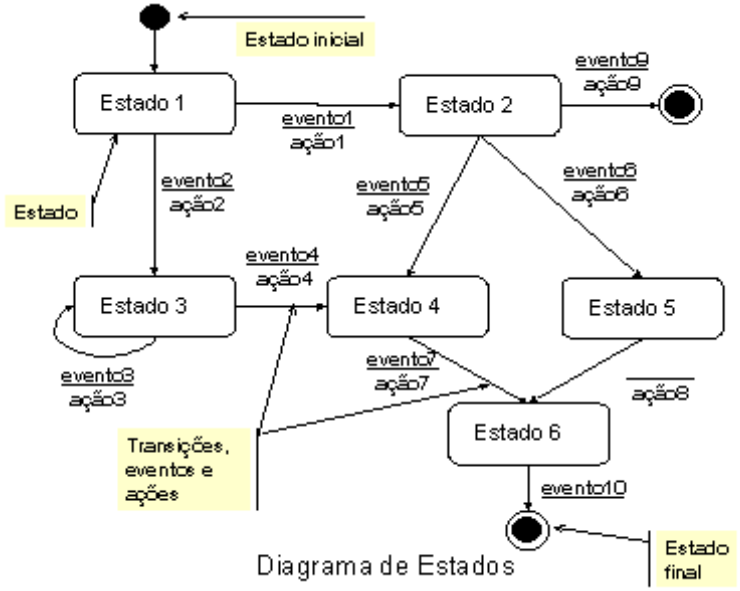
5.2.5 Especificando funções da aplicação usando Diagrama de Estados

Uma **máquina de estados** permite modelar os aspectos dinâmicos de um sistema. Ela mostra o comportamento do sistema em termos de estados pelos quais ele passa ao longo de sua existência. A máquina de estados mostra com o sistema reage a eventos externos. Além disso, ela mostra a ordem na qual o sistema assume determinados estados.

Máquinas de estados podem ser utilizadas para modelar o comportamento de componentes individuais de software, funções de aplicação (ou casos de uso, como são chamadas na UML) ou um sistema inteiro.

A máquina de estados é uma abstração bastante utilizada em computação. A teoria da computação nos mostra como descrever matematicamente uma máquina de estados. Também podem ser encontradas notações com o mesmo potencial em diversas metodologia de desenvolvimento de software.

A UML proporciona uma representação gráfica para máquinas de estados: o **diagrama de estados**. Esta notação permite representar estados, transições, eventos e ações utilizando símbolos específicos, como mostra a figura.



Uma desvantagem do diagrama de estado é ter de definir todos os possíveis estados de um sistema. Isto não é problema quando se está modelando uma única função, mas quando se descreve o sistema completo quando o sistema assume dezenas ou até centenas de possíveis estado o diagrama pode se tornar inviável. Neste cursos vamos usar máquinas de estados para representar o comportamento de cada função da aplicação ou caso de uso.

Um **estado** é uma condição ou situação na qual um objeto do sistema (ou o próprio sistema) está durante o seu tempo de "vida". O sistema pode permanecer num estado até que ocorra um evento externo, até que uma determinada atividade que ele esteja executando termine ou que alguma condição seja satisfeita. O sistema normalmente passa por diversos estados. Graficamente um estado é representado por um retângulo com as bordas arredondadas.

Um estado tem várias partes:

- **Nome** - uma palavra ou sentença simples que distingue um estado de outro.
- **Ações de entrada e saída** - as ações que são executadas quando o sistema entra e sai, respectivamente, de um estado.
- **Sub-estados** - Um estado pode ter sub-estados aninhados em sua estrutura. Eles podem ser concorrente ou seqüenciais.
- **Transições internas** - transições entre os sub-estados internos que não mudam o estado.

Cada máquina de estado tem um **estado inicial** que indica o estado no qual o sistema encontra-se quando a função é iniciada. Quando se está modelando funções, este estado inicial é também chamado de pré-condição. Uma máquina de estado pode ter um ou mais **estados finais** que indica as situações, também chamada de **pós-condições**, na qual o sistema deve estar para considerarmos que a função foi executada como esperado.

Um **evento** é a especificação de uma ocorrência significante no espaço e no tempo. No contexto de máquina de estados o evento é uma ocorrência que leva a uma transição. A ocorrência pode ser externa ao sistema, interna (outra parte do sistema) ou um instante de tempo atingido.

Uma **transição** é um relacionamento entre dois estados indicando que o sistema quando está num determinado estado irá para um outro estado na ocorrência de um certo evento e realizará uma certa ação. A transição representa a mudança de estado.

Uma transição tem cinco partes (algumas em comum com os estados):

- **Estado origem** - o estado no qual o sistema está antes que a transição seja percorrida
- **Estado destino** - o estado para o qual o sistema vai quando a transição é percorrida
- **Evento de ativação** - a ocorrência que faz com que a transição seja ativada indicando a mudança de estado
- **Condição de guarda** - é uma expressão *booleana* que é avaliada quando a transição é ativada. Se a expressão for avaliada como verdadeira a transição é realizada, caso contrário o evento é ignorado e o sistema permanece no mesmo estado.
- **Ação** - a ação que deve ser executada pelo sistema na ocorrência do evento. Normalmente, a execução da ação leva ao sistema ficar no estado "executando a ação". Em outro casos, quando a ação tem um tempo curto e quando acaba o sistema vai direto para um outro estado não é preciso representar este estado de execução das ações.

Uma **ação** ou **atividade** é uma execução realizada pelo sistema. A ação é uma computação atômica, indivisível, enquanto a atividade pode ser sub-dividida em outras atividades ou ações.

5.2.6 Exemplo: especificando uma função da aplicação

No capítulo 4, apresentamos a especificação informal de um caso de uso *validar cliente*. Vejamos agora como a função da aplicação que implementa este caso pode ser especificada usando diagrama de atividades e de estados.

As figuras abaixo mostram os diagramas de estados que descrevem o comportamento do sistema em decorrência dos eventos do usuário. Na primeira figura estão descritos os estados de ler e de verificar a senha. Na figura imediatamente abaixo apresentamos os sub-estados do estado *lendo senha* do diagrama de cima. É importante ressaltar que estes não são os únicos diagramas possíveis.

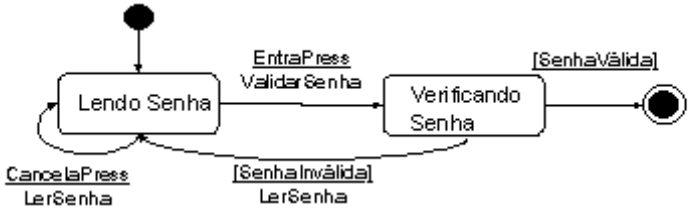
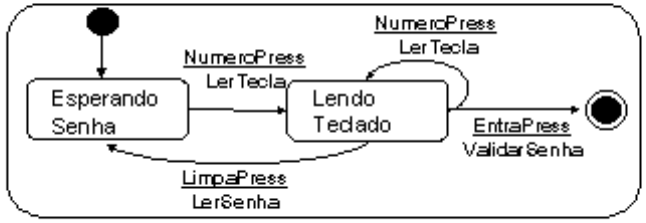


Diagrama de Estados para função *Validar Cliente*



Sub-estados para o estado *Lendo Senha*

A próxima figura mostra o diagrama de atividades para a função *validar cliente*.

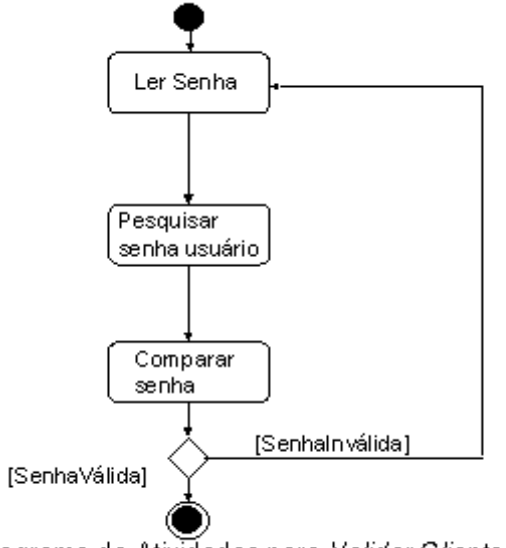


Diagrama de Atividades para *Validar Cliente*

A seguir, mostraremos um diagrama alternativo para a função validar cliente cujas atividades *ler senha* e *pesquisar senha usuário* são realizadas em paralelo.

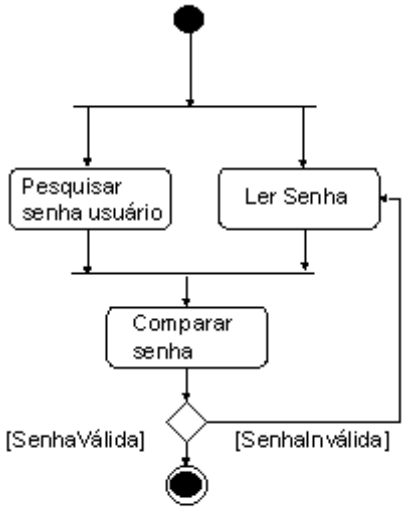


Diagrama de Atividades alternativo para *Validar Cliente*

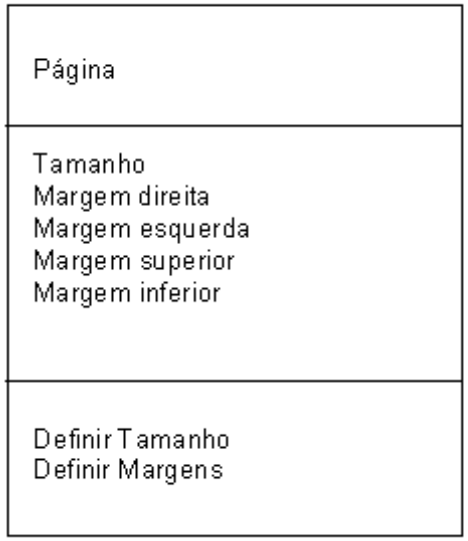
Estes diagramas podem ser modificados. Você pode fazer o seu próprio design e modelar do jeito que você quiser. Como exercício, experimente modelar o fato de que cada cliente poder apenas fazer três tentativas.

5.2.7 Associando Funções da Aplicação a Objetos do Domínio usando Diagramas de Classes.

Vimos que uma Função da Aplicação opera sobre objetos, modificando seu estado ou comportamento. As classes e os relacionamentos permitem elaboramos um modelo dos objetos do domínio com um potencial equivalente aos Diagramas Entidade-Relacionamento utilizados nas metodologias estruturadas e no projeto de banco de dados. Os diagramas de classes, entretanto, permitem representarmos também as **operações ou funções** que são associadas a uma classe. Com isto podemos descrever quais as funções da aplicação que podem ser realizadas com os objetos representados por uma classe.

Na modelagem conceitual pretendemos representar como as funções da aplicação (os casos de uso) estão associados com alguns dos objetos da aplicação representados como classes. Por exemplo, num *editor de texto* podemos modelar as páginas de um documento através da classe *página*, cujos atributos são *tamanho* e *margem direita, esquerda, superior e inferior*. Para que o usuário possam modificar as propriedades da página do documento o editor de texto deve oferecer as funções *definir tamanho* e *definir distância das margens*.

Usando o diagrama de classes podemos representar a classe *página*, da seguinte forma:



É importante ressaltar que no modelo conceitual nosso objetivo é definir quais funções o sistema oferece para o usuário modificar o objeto representado pela classe. A função *definir tamanho* permite ao usuário modificar tamanho da página. Da mesma forma *definir distância das margens* modifica a distância entre a borda e a área onde o texto vai ser escrito. Na modelagem conceitual devemos representar numa classe apenas as funções da aplicação que podem ser controladas pelo usuário e que modificam os objetos da classe à qual ela está relacionada.

5.3 O Modelo de Interação Usuário-Sistema

Vimos que o modelo conceitual da aplicação deve especificar de maneira abstrata o modelo de interação e de funcionalidade do sistema. Na seção 5.2 mostramos como especificar o modelo conceitual de uma aplicação em termos de conceitos do domínio e funções de aplicação. Este modelo deve ser derivado da especificação de requisitos como forma de garantir que a aplicação satisfaz as necessidades dos usuários. Os casos de uso são um ponto chave neste processo. Eles associam as tarefas que os usuários querem realizar para atingir as suas metas às funções da aplicação que são oferecidas pelo software. Estas funções da aplicação operam sobre conceitos do domínio. A especificação das funções e dos objetos da aplicação determinam a funcionalidade do software. Vimos como ela pode ser especificada através da UML (Linguagem de Modelagem Unificada) que permite descrever o comportamento de cada caso de uso através de diagramas de estados e de atividades, bem como outros diagramas que não foram estudados.

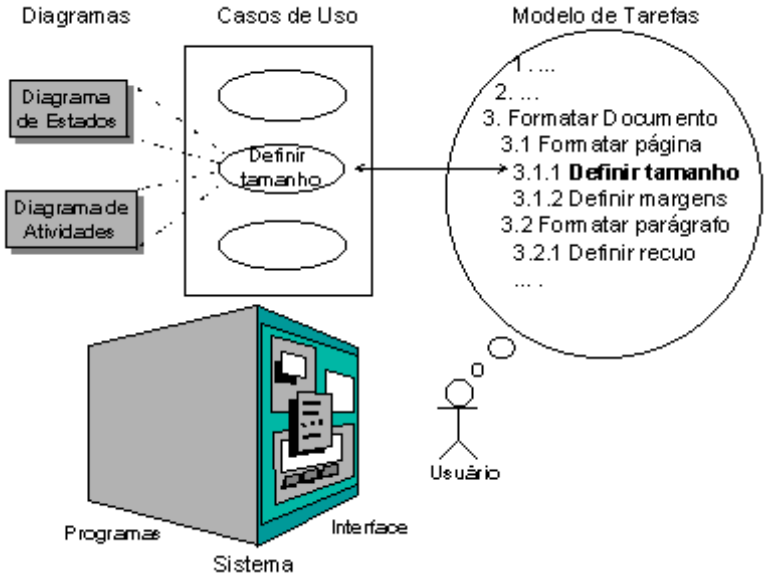
Entretanto, a especificação da funcionalidade do software que determina **o que** se pode fazer com o sistema ainda não é suficiente. É preciso ainda especificar **como** o usuário vai interagir com o sistema. Esta especificação também deve ser feita de maneira conceitual e abstrata da mesma forma que na funcionalidade. Neste caso, estamos especificando a forma como o usuário interage com o sistema independente de uma interface de usuário específica.

Nesta seção vamos mostrar como especificar o modelo de interação da aplicação. Para cada caso de uso vamos descrever o processo de interação entre o usuário e o sistema.

5.3.1 O processo de interação usuário-sistema

Vimos que as metas do usuário são satisfeitas pela execução de tarefas pelo usuário que façam o sistema executar cada função da aplicação. O modelo de tarefas, visto no capítulo 3, mostram os planos do usuário para atingir cada uma das suas metas. Na especificação dos requisitos os modelos de tarefas descrevem as hierarquias de sub-metas até o caso de uso -- a função da aplicação que o sistema deve oferecer. Ao detalharmos mais ainda este plano, descreveremos as ações (ou operações, como são chamadas no GOMS) que o usuário deve realizar com o sistema. Este conjunto de ações é denominado de **comando de função**.

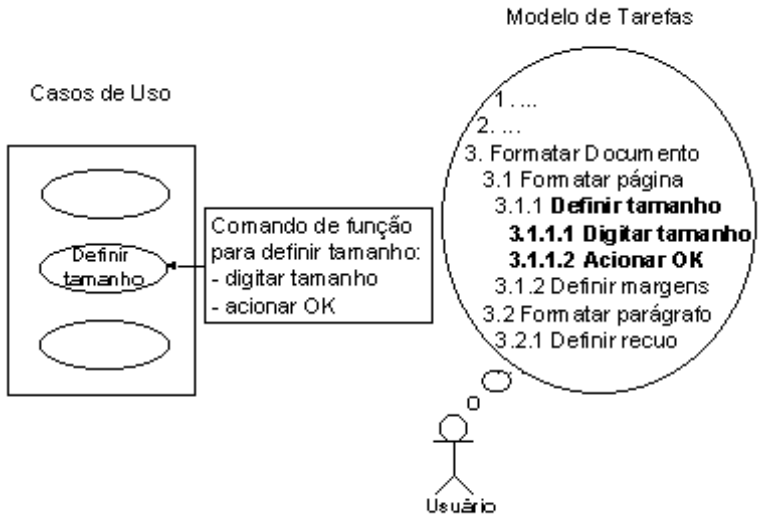
A figura abaixo ilustra a associação entre os casos de uso, o modelo de tarefa e os diagramas utilizados para descrever o comportamento de cada função da aplicação. Ela também situa o que estamos falando em todo o contexto do desenvolvimento de software que temos abordado desde a etapa de análise e especificação de requisitos. Vimos que precisamos analisar as metas dos usuários e elaborar o modelo de tarefas para decidir quais casos de uso o sistema deve oferecer para o usuário. Cada caso de uso determina uma função da aplicação. O conjunto de funções da aplicação definem a funcionalidade do sistema e podem ser especificados em diagramas de classes, de estados, de atividades e outros diagramas. Mais adiante veremos como modelar a interação a através de diagramas de seqüências.



O modelo de interação é composto pelo conjunto de **comandos de funções** necessários para o controle de cada função da aplicação determinada pelos casos de uso. Ele deve também determinar quais as ações que o sistema deve fazer apresentar as informação ou resultados ao usuário associados com cada função da aplicação.

Por exemplo, se a meta do usuário de um editor de texto for *Formatar documento*, vimos no capítulo 3 que o modelo de tarefas especifica sub-metas como *formatar página, formatar parágrafo* e *formatar fonte*, que por sua vez podem ser decompostas, cada uma, em outras sub-metas. A sub-meta *definir tamanho da página* pode ser atingida diretamente pela função da aplicação de mesmo nome, caracterizando um caso de uso. Até então não dissemos o que o usuário tem que fazer para mandar o sistema executar esta função.

O usuário, neste caso, tem que fornecer o valor do tamanho da página e mandar o sistema modificar o tamanho atual para o tamanho desejado. Ele pode fazer isto, por exemplo, digitando o *valor do tamanho da página* desejado numa caixa de texto e acionando um botão de *modificar tamanho*. O comando de função seria formado por estas duas ações básicas. Ele também poderia elaborar um comando do tipo *"modifique tamanho para A4"*.



O comando de função deve determinar quais as ações que o usuário deve fazer para que uma função da aplicação seja executada pelo sistema. Podemos identificar alguns tipos de ações básicas que o usuário costuma realizar ao interagir com o sistema. Elas são chamadas de **interações básicas** e podem ser:

- Digitar valores de informação
- Escolher uma informação de uma lista
- Acionar um botão físico ou imagem na tela (*widget*)
- Mover um botão físico ou imagem na tela (*widget*)
- Visualizar uma informação
- Escutar uma informação

Na especificação conceitual do modelo de interação o designer pode limitar-se a uma descrição informal, como mostrado acima.

É importante ressaltar que estamos chamando de comando as interações necessárias para um único comando. Existe uma associação direta de um comando para cada função da aplicação. O comando requer um conjunto de interações para que uma certa função seja controlada. Ele pode ter diversos efeitos sobre a função como veremos mais adiante. Se para uma determinada meta que o usuário tenha ele necessite de várias funções da aplicação, ele precisará realizar as interações de cada comando de função.

As diversas interações que o usuário tem de realizar normalmente são composições de interações básicas. Ele pode realizar, por exemplo, uma seqüência de acionamentos para comandar uma determinada função. Em outra situação o comando de função poderia ser composto pela repetição de uma digitação de valor mais o acionamento de um *widget*. Podemos identificar alguns tipos de **estruturas de interações**. São elas:

- **Seqüência** - quando as interações básicas são realizadas numa seqüência específica
- **Seleção** - quando o usuário seleciona uma dentre várias opções de interação
- **Repetição** - quando as interações básicas são realizadas de forma repetitiva
- **Agrupamento** - quando as interações básicas podem ser realizadas em qualquer ordem de seqüência
- **Combinação** - quando as interações básicas devem ser realizadas de forma interdependente ou simultânea (dependência temporal)

A seqüência é uma das estruturas mais utilizadas. Ela determina que o usuário deve seguir uma seqüência específica de interações. Por exemplo, se para acionar a função *definir tamanho da página* o usuário precisa informar primeiro o tamanho e depois mandar iniciar a execução o designer pode especificar que o comando tem uma estrutura sequencial composta pelas interações *digitar tamanho* e *acionar botão iniciar*.

A seleção pode ser utilizada quando o designer quer dar ao usuário a opção de escolher alguma interação. Para a função do exemplo anterior o usuário poderia ter a opção de escolher entre *acionar o botão iniciar* e *acionar o botão cancelar* no comando de função.

A repetição deve ser utilizada quando o comando requer que uma interação ou uma estrutura de interações seja repetida para que a função seja executada. O exemplo mais comum é a repetição do *clique com o mouse* (*duplo clique*). Um outro exemplo de repetição seria um comando para a função *definir margens* no qual o usuário tivesses que repetir a interação *digitar distância* para cada uma das quatro margens da página.

O agrupamento ocorre quando o usuário tem a sua disposição diversas interações e ele pode realizá-las sem preocupar com a ordem. Por exemplo, para um comando para a função *imprimir* o usuário pode fazer as interações *digitar número de cópias* e *digitar o nome do documento*, em qualquer ordem.

A combinação determina, por exemplo, que o usuário deve ter que realizar as interações *pressionar tecla SHIFT* e *mover o mouse* para comandar uma função.

Atenção! A grande maioria das aplicações existentes foram concebidas e especificadas sem estas noções em mente. Estamos aqui fazendo uma proposta de como as coisas poderiam ser para que o modelo conceitual da aplicação fosse melhor estruturado e mais lógico, possibilitando que os sistemas pudessem ser mais facilmente utilizados e aprendidos.

5.3.2 A interação com funções da aplicação

O nosso modelo de interação é descrito em função dos casos de uso. Para cada caso de uso temos associada a ele uma função da aplicação que deve ser implementada pelo software. A função da aplicação é uma entidade conceitual que pode ser implementada por um ou mais componentes de software -- funções, procedimentos, métodos de objetos ou qualquer forma de implementação computacional. A função da aplicação é algo que o sistema oferece para que o usuário realize uma determinada tarefa para atingir as suas metas.

A função da aplicação deve estar sob o controle do usuário. É ele quem deve tomar a iniciativa de quando a execução da função deve ser iniciada, interrompida, cancelada, terminada e outras opções de **controle operacional**. O designer deve determinar a maneira como o usuário pode controlar cada função e quais as opções de controle serão fornecidas. São exemplos de controle operacional **iniciar, interromper, continuar, concluir, cancelar, desistir** e outros.

- **Iniciar** refere-se a ativação da função.
- **Interromper** é usado quando é possível fazer uma pausa na execução e o **continuar** permite que a execução seja continuada do ponto em que foi interrompida.
- **Concluir** (ou finalizar) pode ser usado em funções que ficam repetidamente executando algo até que o usuário deseje terminar.
- **Desistir** é usado para permitir ao usuário desistir da execução da função, antes que ela seja iniciada.
- **Cancelar** é usado para o usuário cancelar a execução. (Atenção! No Windows o cancelar não é usado com este propósito, mas como desistir).

O designer pode especificar outros controles que sejam convenientes para a função da aplicação que ele esteja projetando. Ele deve ainda determinar como cada uma destas opções ele deve ser controlada pelo usuário, isto é, qual ação o usuário deve fazer.

Por exemplo, na função da aplicação *sacar dinheiro* o usuário pode ter as opções de *iniciar* ou *desistir*, antes de iniciar e *cancelar* após ele ter mandado iniciar. Estas opções podem ser acionadas através de teclas ou de botões.

Além do controle operacional, o usuário deve fornecer as informações necessárias para a execução da função. Estas informações são os **operandos** da função da aplicação. Existem diversas formas do usuário fornecer informações (os operandos) para cada função da aplicação. O designer deve verificar quais operandos são necessários e de que maneira o usuário irá fornecer cada informação.

São exemplos de operandos *senha do cliente* e o *número da conta* e o *valor de saque* na função *sacar dinheiro*. Os valores de vendas e compra necessários para o *cálculo de lucro* são também exemplos de operandos.

Cada função da aplicação pode estar em um ou mais **estados** como vimos na seção anterior. O designer pode optar por mostrar ao usuário cada estado no qual a função está. O resultado final produzido por cada função também deve ser mostrado ao usuário.

5.3.3 Exemplo de especificação de modelos de interação

Utilizando os conceitos introduzidos acima, vamos fazer a especificação de comandos para três funções da aplicação. Esta especificação será feita de maneira informação utilizando os conceitos de comando de função e interação, estruturas de interação. Cada uma das especificações é apenas uma possibilidade dentre várias possíveis. Ela reflete a decisão do projetista.

a. Comando para a função da aplicação *consultar livro* para um sistema de biblioteca.

Para o usuário consultar um livro ele deve primeiro fornecer informações sobre o *autor,* o *título* e o *código ISBN*. Estas informações são os conceitos do domínio necessários para que a busca seja realizada. Eles são os operandos da função.
Após fornecer os valores dos operandos, o usuário pode escolher entre comandar o *início da busca*, *desistir* de fazer a consulta, ou *interromper* a busca na base de dados.

Utilizando as estruturas vistas acima e algumas da interações básicas, podemos descrever este comando informalmente como:

```
Comando Consultar Livro {
    Seqüência {
        Agrupamento {
            Fornecer informação Autor
            Fornecer informação Título
            Fornecer informação ISBN
        }
        Seleção {
            Acionar Iniciar
            Acionar Desistir
            Acionar Parar
        }
    }
}
```

Na seção 5.3, veremos como este modelo de interação pode ser traduzido em diferentes interfaces.

b. Comando para a função *imprimir arquivos* para um gerenciador de arquivos.

Para comandar a função que imprime arquivos (num suposto gerenciador de arquivo) o usuário deve fornecer os operandos da função para em seguida selecionar o controle de execução da função. Isto deve ser feito em seqüência. O projetista inicialmente optou por enviar uma mensagem direta para o usuário indicando o que ele fazer (Veja). A seqüência e a mensagem são estruturados num agrupamento, pois não importa a ordem que o usuário faça isto.

O comando requer que o usuário forneça o *nome do arquivo* a ser impresso, o *nome da impressora* e o *número de cópias*. O usuário deve poder escolher dentre as impressoras disponíveis e também configurá-la. O designer decidiu que o usuário apenas pode realizar os comandos de controle após ter fornecidos os dados. Neste caso os grupos de interações para fornecimentos dos dados e controle operacional devem estar estruturados com uma seqüência. O fornecimento dos dados pode ser realizado em qualquer ordem (agrupamento). Para fornecer o nome do arquivo o usuário pode escolher entre digitar o nome diretamente ou selecionar de uma lista de nomes de arquivos (seleção). Para configurar uma impressora é necessário escolhê-la para ativar a mensagem de comando configurar impressora. Esta dependência entre as duas ações do usuário requer a utilização da estrutura combinação. O número de cópias deve ser fornecido diretamente.

Após ter fornecido os dados, o usuário pode escolher o controle operacional da função desejado. A função imprimir permite os controles *iniciar, parar, interromper, continuar e desistir*. O usuário deve selecionar (seleção) cada controle e acionar a opção correspondente.

```
Comando de função Imprimir
  Agrupamento {
      Veja "Para imprimir você deve fornecer os dados, e, em seguida selecionar o controle da função que você deseja acionar."
      Sequencia {
          Agrupamento {
              Seleção {
                  Fornecer Informação Nome do Arquivo
                  Seleção Informação Nome do Arquivo
              }
              Combinação {
                  Seleção Informação Nome da impressora
                  Acione Mostrar Comando de Função Configurar impressora
              }
              Fornecer Informação Número de cópias
          }
          Seleção {
              Acione Iniciar
              Acione Parar
              Acione Interromper
              Acione Continuar
              Acione Desistir
          }
      }
}
```

Também na seção 5.3 apresentaremos uma interface que satisfaça esta especificação.

5.3.4 Especificando o modelo de interação com diagrama de interação UML simplificado

O **diagrama de interação** é mais um dos diagramas da UML (Linguagem de Modelagem Unificada) que utilizaremos. Para a especificação do modelo de interação usuário-sistema utilizaremos uma versão simplificada que poderá ser extendida mais adiante quando quisermos expressar a interação entre os componentes (internos) do software. Na versão simplificada, modelaremos apenas a interação (externa) entre usuário e as funções da aplicação.

Além disto, utilizaremos apenas o **diagrama de seqüência** de interação que é uma das duas formas de diagramas de interação. O outro, o **diagrama de colaboração**, é desnecessário para a interação usuário-sistema uma vez que eles são apenas os dois únicos agentes colaboradores.

O diagrama de seqüência de interação tem por objetivo descrever, como o próprio nome indica, a seqüência de interações entre o usuário e o sistema ao longo do tempo. Podemos utilizar o diagrama para descrever a seqüência de interações básicas que compõem um comando de função, bem como para modelar toda a interação com as diversas funções da aplicação.

Uma limitação do diagrama de seqüência é não permtir descrever a estrutura de interações que vimos na seção 5.3.1. Também não é possível diferenciar as formas de mensagens que o sistema envia para o usuário. É possível fazer adaptações no diagrama para especificar as estruturas.

O diagrama representa o "tempo de vida" de cada uma dos agentes por uma linha de tempo vertical. A interação entre os dois agentes (usuário e sistema) é representada por setas na direção da informação que é enviada.

Por exemplo, o comando de função para *definir tamanho*, composto pela seqüência de interações básica *digitar tamanho* e *acionar (botão) OK*. No diagrama estas duas interações são representadas por setas na direção do usuário para o sistema. No diagrama estamos representando por uma barra vertical o "tempo de vida" do sistema.

Anterior | Índice | Próximo

 Última atualização: 12/04/01

6 Design de Interfaces de Usuário

6.1 Interfaces de Usuário

Antes de estudarmos como conceber e projetar as interfaces de usuário de um sistema computacional, é preciso conhecermos alguns conceitos importantes relacionados com a área. Esta seção define o que são as interfaces de usuário, seus objetivos e apresenta alguns estilos de interação que podem ser utilizados para concretizarmos o modelo conceitual de interação visto na seção anterior.

6.1.1 Para que serve a Interface de Usuário?

O termo interface é aplicado normalmente àquilo que interliga dois sistemas. Tradicionalmente, considera-se que uma interface homem-máquina é a parte de um artefato que permite a um usuário controlar e avaliar o funcionamento do mesmo através de dispositivos sensíveis às suas ações e capazes de estimular sua percepção. No processo de interação usuário-sistema a interface é o combinado de software e hardware necessário para viabilizar e facilitar os processos de comunicação entre o usuário e a aplicação. A interface entre usuários e sistemas computacionais diferencia-se das interfaces de máquinas convencionais por exigir dos usuários um maior esforço cognitivo em atividades de interpretação e expressão das informações que o sistema processa [Norman 86].

Segundo Moran, a interface de usuário deve ser entendida como sendo a parte de um sistema computacional com a qual uma pessoa entra em contato física, perceptiva e conceitualmente [Moran 81]. Esta definição de Moran caracteriza uma perspectiva para a interface de usuário como tendo um componente físico, que o usuário percebe e manipula, e outro conceitual, que o usuário interpreta, processa e raciocina. Moran e outros denominam este componente de modelo conceitual do usuário.

A interface é tanto um meio para a interação usuário-sistema, quanto uma ferramenta que oferece os instrumentos para este processo comunicativo. Desta forma a interface é um sistema de comunicação. Quando se considera a aplicação como máquina(s) virtual(is), a interface pode ser considerada ainda como um ambiente virtual para ações.

6.1.2 Objetivos das Interfaces de Usuário

Para que o usuário possa utilizar o sistema com sucesso ele deve saber quais as funções da aplicação são oferecidas pelo sistema e como ele pode interagir com cada uma delas, isto é, qual o modelo conceitual da aplicação o designer concebeu para ele.

A interface de usuário tem dois objetivos fundamentais:

1. Determinar como o usuário pode efetivamente interagir com o sistema, desenvolvendo uma interface que permita ao usuário interagir de acordo com o modelo (conceitual) de interação.
2. Mostrar para o usuário o que ele pode fazer, isto é, quais as funções da aplicação o sistema oferece, e quais os comandos de funções e mensagens auxiliares que compõem o modelo de interação.

O design de interfaces de usuário é um dos pontos da área de pesquisa **Interação Humano-Computador** (*Human-Computer Interaction*). Tradicionalmente, os pesquisadores desta área preocupam apenas com o primeiro destes dois objetivos. Existe uma abordagem completar, chamada de **Engenharia Semiótica**, que se preocupa em como **comunicar** o modelo de interação para o usuário.

Para atingir estes dois objetivos, o design de interfaces de usuário é a etapa do desenvolvimento de software que deve:

1. traduzir o modelo de interação - os comandos de função e mensagens auxiliares - de cada função de aplicação numa interface de usuário.
2. comunicar a funcionalidade e o modelo de interação associado a cada função da aplicação através da interface de usuário

A seguir vamos apresentar diretrizes para o design de interfaces de maneira que estes dois objetivos possam ser atingidos. Antes, porém, vamos discutir as diferentes maneiras de interação entre o usuário e o sistema.

6.1.3 Software e Hardware da Interface

A interface possui componentes de software e hardware. Os componentes de **hardware** compreendem os dispositivos com os quais os usuários realizam as atividades motoras e perceptivas. Entre eles estão a tela, o teclado, o mouse e vários outros. É com o hardware da interface que o usuário entra em contato direto. Alguns dipositivos destinam-se a veicular mensagens do sistema para o usuário e são chamados de dispositivos de saída. Os dispositivo de entrada permitem ao usuário fornecer mensagens para o sistema.

O **software da interface** é a parte do sistema que implementa os processos computacionais necessários para:

- controle dos dispositivos de hardware,
- construção dos objetos de interfaces (os widgets) com os quais o usuário também pode interagir,
- geração dos diversos símbolos e mensagens que representam as informações do sistema, e
- interpretação dos comandos dos usuários.

As interfaces gráficas, também chamadas de GUI (*Graphical User Interface*)ou de WIMP (*Windows, Icons, Menus, Pointer*), requerem rotinas de software bastante complexas para que os widgets possam ser desenhados e as ações do usuário movendo e clicando com o mouse sobre a tela possam ser implementadas. Os programas que implementam estas interfaces gráficas podem ser construídos com **ferramentas de interfaces**.

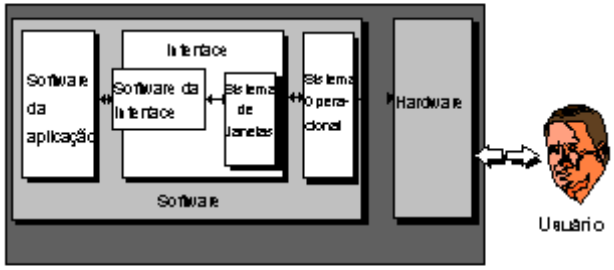
O principal exemplo de ferramentas de interfaces são os **sistemas de janelas**. Eles são os responsáveis, junto com o sistema operacional, a controlar os dispositivos de hardware e fazer o gerenciamento das janelas. É o gerenciador de janelas que controlam a sobreposição de janelas que exibem diversas aplicações independentes. O usuário pode minimizar ou maximizar, trazer para frente ou enviar para trás. Quem desenvolveu cada aplicação não precisa ficar se preocupando com isto. O gerenciador controla estas tarefas. Ele também é o responsável em verificar se ocorreram eventos nos dispositivos de entrada e repassar cada ocorrência para a aplicação específica.

Um outro exemplo de ferramenta são as rotinas de bibliotecas (classes ou funções) que fazem o desenho de botões, janelas, menus, ícones, texto e diversos outros widgets de interfaces. Estas rotinas devem também interpretar as ações que o usuário realiza e dar o tratamento adequado a elas. Estas bibliotecas, chamadas de **toolkits**, são ferramentas de programação. O programadores precisam conhecer a **Interface com o Programa da Aplicação** (API) para utilizar as diversas classes, funções e estruturas de dados disponíveis e incorporá-las ao programa

Existem ferramentas mais avançadas que permitem ao usuário montar os widgets arrastando-os para janelas da interface. Com estas **ferramentas de design** o projetista não precisa programar a interface diretamente. A ferramenta gera o código a ser executado.

A utilização de ferramentas permite que as interfaces mantenham a mesma aparência e comportamento mantendo o mesmo padrão, como veremos na próxima seção.

A figura abaixo mostra a arquitetura global de um sistema interativo. Nela podemos observar o hardware e o software da interface e como eles interagem com o restante do sistema.



Sistema Interativo

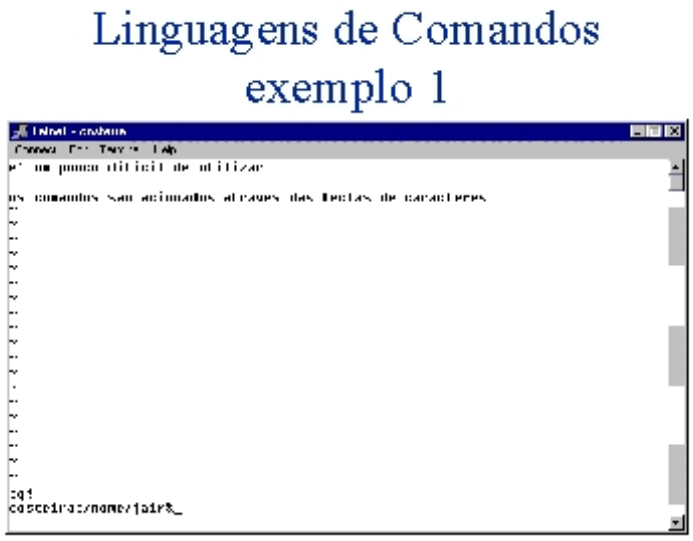
Para maiores detalhes sobre ferramentas de software de interfaces de usuário veja notas de aula do curso de projeto de interfaces de usuário - cap 3.

6.1.4 Estilos de interação

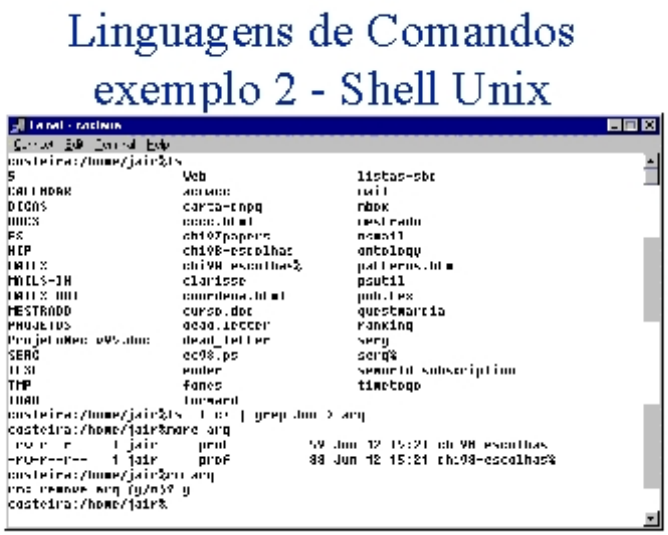
Existem diversas maneiras pela qual o usuário pode interagir com o sistema e diversos tipos de interfaces que podem viabilizar estes modelo de interação. Na literatura estas diferenças são chamadas de **estilos ou paradigmas de interação**. Diversos estilos são identificados pela literatura.

Linguagens de comandos

A interfaces baseadas em linguagens de comandos proporcionam ao usuário a possibilidade de enviar instruções diretamente ao sistema através de comandos específicos. As linguagens de comandos foram o primeiro estilo de interação a ser usado amplamente. Este estilo caracteriza-se por possibilitar ao usuário construir comandos através do teclado (hardware da interface) que devem poder ser interpretados pelo software da interface para que funções específicas da aplicação sejam ativadas. Os comandos podem ser produzidos pelo acionamento de teclas de funções especiais, ou pelo acionamento de uma tecla de caractere, ou pela estruturação delas. Um tipo de estruturação especial é aquele no qual teclas são acionadas para produzir palavras de comandos que podem ser combinadas em sentenças de acordo com a gramática que define a linguagem. Este estilo de interação visa possibilitar que a linguagem de comandos aproxime-se daquela falada pelos usuários.
As linguagens de comandos podem ser extremamente simples ou ter complexidade equivalente a de linguagens de programação. Linguagens de comandos elementares associam um vocabulário de comandos a funções específicas do sistema sem permitir combinações dos mesmos. Um exemplo de linguagem de comando elementar é a do editor de texto vi, mostrado na figura abaixo.



Para permitir uma maior flexibilidade e um maior poder expressivo os comandos podem ser estruturados de acordo com regras de gramáticas regulares, livre-de-contexto, sensível ao contexto ou irrestritas. O grau de expressividade do usuário é diretamente proporcional à capacidade do software da interface de interpretar estes comandos de acordo com a gramática e a semântica que define a linguagem. A interface com o sistema operacional Unix através do shell é exemplo deste estilo no qual a linguagem de comandos é bastante sofisticada permitindo ao usuário construir comandos que podem ser combinados de maneira bastante flexível. A figura abaixo ilustra a utilização de comandos do shell do Unix.

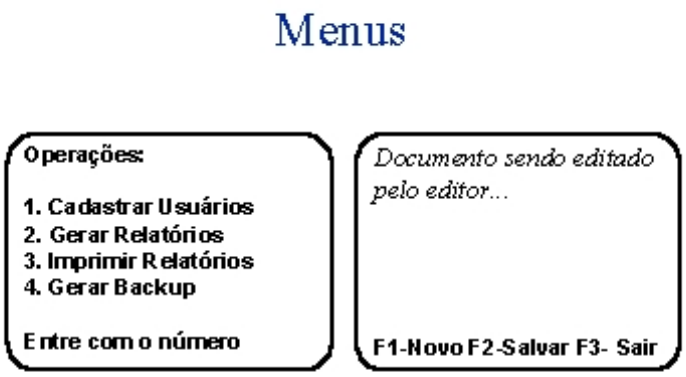


As linguagens de comandos determinam apenas qual devem ser os comandos que o usuário deve utilizar para interagir com o sistema sem especificar como deve ser o estilo de interação a ser utilizado pelo sistema para se comunicar com o usuário. Esta comunicação sistema-usuário é feita através da tela do computador (hardware da interface) que possibilita também um feedback imediato das ações do usuário. Nas interfaces baseadas em linguagens de comandos o usuário é quem toma a iniciativa da interação cabendo ao sistema fornecer o resultados dos comandos.

As linguagens de comandos são poderosas em oferecer acesso direto à funcionalidade do sistema e em permitir maior iniciativa do usuário e uma maior flexibilidade na construção dos comandos através da variação de parâmetros e combinação de palavras e sentenças. Contudo, este poder e flexibilidade implicam em uma maior dificuldade dos iniciantes em aprender e utilizar o sistema. Os comandos e a sintaxe da linguagem precisam ser relembrados e erros de digitação são comuns mesmo nos mais experientes. A falta de padronização nos diversos sistemas é um fator importante na dificuldade de utilização deste estilo. Usuários especialistas, no entanto, conseguem maior controle do sistema e produtividade através de interfaces baseadas em linguagens de comandos.

Menus

Nas interfaces orientadas por menus o conjunto de comandos de funções oferecidas pela aplicação é mostrada ao usuário através da tela e cabe ao usuário selecionar uma delas através do mouse, de teclas alfanuméricas ou de teclas especiais. Como as funções e a maneira de acioná-las estão visíveis na forma de opções para o usuário selecionar existe uma demanda maior pelo processo de reconhecimento ao invés de recordação com no caso das interfaces baseadas em comandos.



Normalmente o número de opções de comandos é grande o suficiente para ser mostrado de uma única vez na tela da interface. Existem diversas técnicas para se agrupar e apresentar as opções de menus. A mais comum é a categorização hierárquica na qual as opções são agrupadas em categorias de maneira hierárquica. A escolha dos nomes dos grupos e de cada opção do menu é fundamental para que o usuário possa escolher aquelas que permite-o atingir as suas metas.

As interfaces orientadas por menus podem ser textuais ou gráficas. Nas interfaces textuais (ou baseadas em caracteres) as opções são descritas através de palavras na linguagem do usuário que representem as funções do sistema correspondentes ao comandos. O comando é acionado através de uma tecla ou número associado à opção do menu. Nas interfaces gráficas os menus podem ser compostos por palavras ou por signos gráficos específicos. Existem alguns tipos específicos de menus nas interfaces gráficas como menus de barra, pull-down, e pop-up. Discutiremos estes estilos mais adiante quando apresentarmos os principais tipos de widgets.

Na medida que aumenta a funcionalidade de um sistema aumentam o número de opções que devem ser colocadas num menu. A estratégia do designer para evitar que os menus fiquem muito saturados de opções é subdividi-los e organizá-los em grupos. Os menus podem ser organizados de maneira hierárquica, linear ou em rede. Uma das principais desvantagens do uso de menus nestes casos é que o usuário pode ficar perdido ao navegar por inúmeras estruturas de menus.

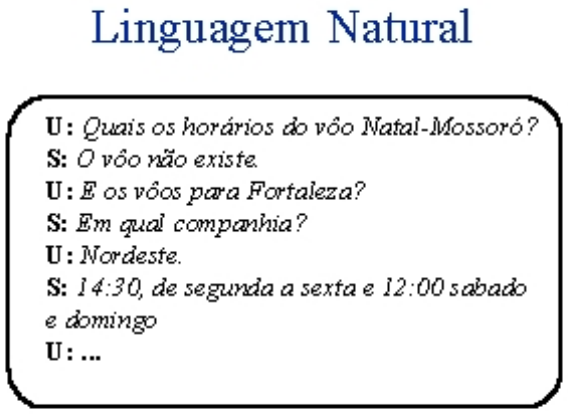
Linguagem Natural

Linguagem Natural (LN) é a expressão utilizada para se referir de maneira genérica à língua que o usuário domina, podendo ser o português, o francês, o inglês ou qualquer das linguagens existentes. Através de interfaces em linguagem natural o usuário pode interagir com o sistema na sua própria linguagem. Ao

contrário das linguagens por comandos quando o usuário tem que aprender uma linguagem artificial que seja mais facilmente processada por computadores, nas interfaces que processam linguagem natural o maior esforço fica a cargo do computador que deve interpretar e gerar sentenças na linguagem natural do usuário.

A interação em LN é bastante atrativa para usuário com pouco ou nenhum conhecimento em computação. Entretanto ela não se aplica a todos os tipos de sistemas. Sistemas de consulta a informações e sistemas baseados em conhecimento são exemplos onde a utilização de interfaces em LN é bastante interessante. No primeiro caso por possibilitar que usuário não especialistas possam fazer consultas em sua própria língua. No segundo caso para que o sistema gere explicações a partir da sua base de conhecimento, uma vez que a LN é expressiva o suficiente para a descrição do raciocínio artificial do programa, o que não seria possível com outros estilos de interação.

Um programa que processa linguagem natural é bastante complexo. Esta é justamente uma das maiores limitações deste tipo de interface. O exemplo mostrado na figura abaixo apresenta um trecho de interação onde o usuário faz referências anafóricas e utiliza-se de elipses dificultando a interpretação e a geração das respostas.



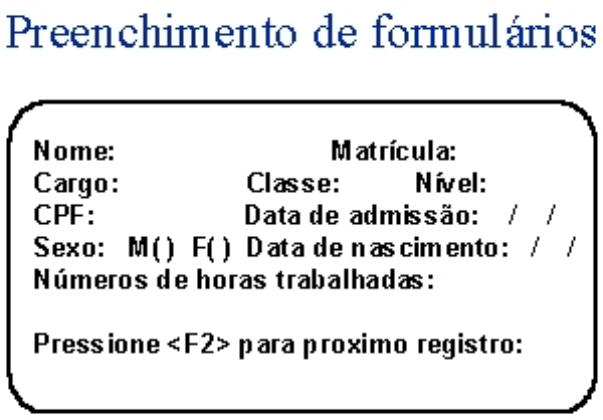
A maioria das interfaces em LN existentes têm sido utilizadas para alguns sistemas específicos de maneira que o escopo do vocabulário e das sentenças seja bastante limitado, reduzindo a complexidade do processo de interpretação e geração das sentenças.

Menus podem ser associados à LN na construção de interfaces mais simples. Nestes casos, menus de palavras são apresentados ao usuário de maneira que ele possa ir construindo sentenças em LN. Isto garante que usuário forme apenas as sentenças que possam ser interpretadas pelo sistema. Além disso, os menus motivam o usuário a formular suas perguntas apresentando termos específicos do domínio que o usuário possa não ter memorizado.

Uma importante desvantagem deste estilo de interação está na imprecisão e na ambigüidade da própria LN que dificulta a sua aplicação em sistemas onde se necessita de precisão e determinismo na interação com o usuário.

Preenchimento de formulário

Interfaces no estilo de preenchimento de formulário são utilizadas principalmente para entrada de dados em sistemas de informação. Estas interfaces apresentam para o usuário uma tela que lembra um formulário em papel solicitando informações específica do domínio da aplicação. Nestes casos o modelo de interação básico é o fornecimento de dados para os campos ou registros de uma base de dados. Os modelos de formulários muitas vezes estão baseados nos formulários em papel que os usuário estavam acostumados a utilizar antes da implantação do sistema, facilitando o aprendizado do modelo de interação.
Este estilo de interação também pode ser utilizado para consulta a base de informações, onde o usuário preenche um formulário que serve como máscara para a busca de dados na base.



Estas interfaces são, em geral, fáceis de aprender e não requerem flexibilidade na funcionalidade. Os aspectos principais que vão influenciar na usabilidade do sistema são a produtividade do usuário, a sua satisfação e o esforço físico provocado pelo sistema, uma vez que estes sistemas são projetados para que os usuários forneçam um grande número de dados ao longo de um dia de trabalho. Estas interfaces devem facilitar a correção de erros de digitação e a verificação dos dados digitados através de técnicas como dígitos verificadores e totalização de valores.

Manipulação Direta

Interfaces de manipulação direta são aquelas que permitem ao usuário interagir diretamente com os objetos da aplicação (dados ou representações de objetos do domínio) sem a necessidade de comandos de uma linguagem específica. Na manipulação direta os comandos são ações que o usuário desempenha diretamente com objeto do sistema.

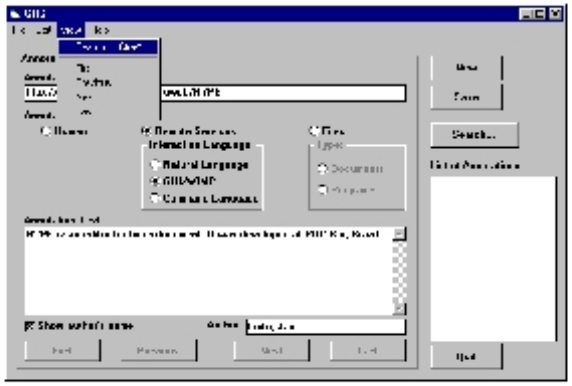
As interfaces gráficas que se utilizam da metáfora do desktop, como as do Apple Macintosh, baseada no Xerox Star, e o Microsoft Windows proporcionam um estilo no qual os usuários podem interagir com o gerenciador de arquivos do sistema operacional através de manipulação de ícones que representam arquivos, diretórios, discos e outros componentes computacionais. O usuário comanda através de ações de arrastar e soltar (drag-and-drop) os ícones utilizando o mouse ou outro dispositivo equivalente.

Embora os primeiros exemplos do que convencionou-se a se chamar de manipulação direta surgiram com a tecnologia de interfaces gráficas e dispositivos apontadores como o mouse, a manipulação direta pode ocorrer em interfaces com tecnologias de telas de caracteres.

WIMP - (Windows, Icons, Menus e Pointers)

O estilo de interação WIMP, um acrônimo em inglês para Janelas, Ícones, Menus e Apontadores, permite a interação através de componentes de interação virtuais denominado de Widgets. Este estilo é implementado com o auxílio das tecnologias de interfaces gráficas, que proporcionam o desenho de janelas e do controle de entrada através do teclado e do mouse em cada uma destas janelas. Os softwares de interfaces que implementam estes estilos permitem a construção de ícones que permite a interação através do mouse, comportando-se como dispositivos virtuais de interação.

Interfaces WIMP



O WIMP não deve ser considerado um único estilo, mas a junção de uma tecnologia de hardware e software, associado aos conceitos de janelas e de widgets que permite a implementação de vários estilos. Nas interfaces WIMP é possível ter os estilos de menus, manipulação direta, preenchimento de formulário e linguagem de comandos. WIMP pode ser considerado um modelo ou um framework de interface apoiado pela tecnologia de interfaces gráficas (GUI).

É possível implementar o estilo WIMP usando tecnologia de telas não gráficas. Entretanto, têm-se a limitação de não se poder ícones gráficos e se ter um baixo nível de resolução que limita a quantidade de objetos de interface que pode ser mostrada para o usuário.

Padrões para Interfaces WIMP

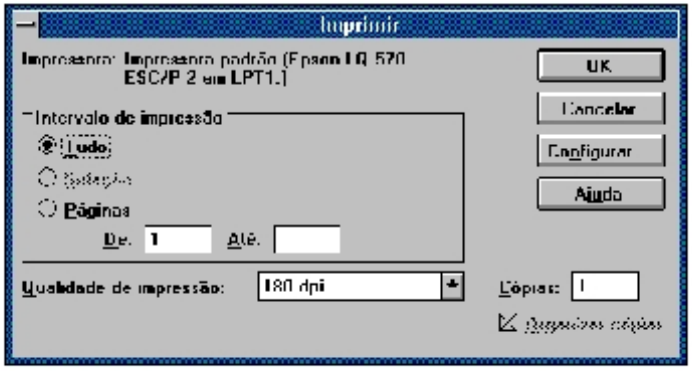
Padrões de interfaces são um conjunto de normas e regras que foram propostas de maneira a se uniformizar e aumentar a consistência da aparência e comportamento (“look and feel”) das interfaces gráficas WIMP.

A aparência e o comportamento das interfaces WIMP é a expressão utilizada para o padrão visual que cada widget de um interface deve obedecer de maneira que eles possam ser facilmente interpretados pelo usuário. Por exemplo, um botão de acionamento deve ter a mesma aparência em todas as instâncias de interfaces. Da mesma maneira, cada widget deve obedecer a um comportamento padrão, ou seja, para cada tipo de acionamento do usuário ou evento do sistema ele deve se comportar de forma previsível. Este comportamento refere-se às modificações na aparência do widget durante o processo de interação.

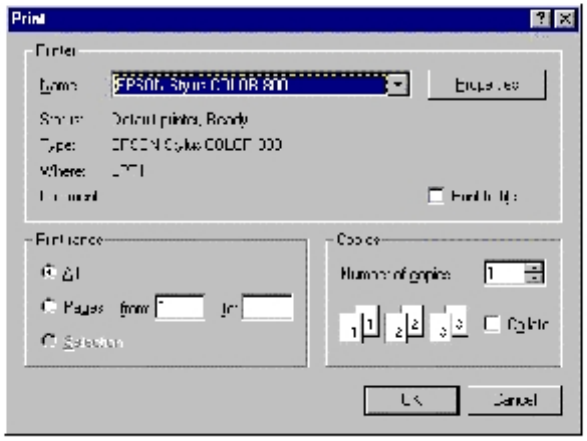
São exemplos de padrões o OSF/Motif, o OpenLook, o Windows 3.11 e o Windows 95. A grande vantagem é que as diversas interfaces construídas obedecendo a um padrão serão consistentes umas com as outras. Entretanto, o uso de determinado padrão não garante que a interface tenha alta usabilidade.

O padrão OpenLook foi proposto pela Sun MicroSystems para as interfaces gráficas das estações gráficas no ambiente SunOS (uma versão da Sun para o Unix). Até metade da década de 90 era o padrão adotado pela Sun, quando foi substituído pelo Motif. O Motif foi proposto pela Open System Foundation (OSF) para ser um padrão internacional. Diversos fabricantes aderiram a esta proposta, inclusive a Sun Microsystems. Os produtos da Microsoft e de outros fabricantes que utilizam o ambiente operacional Microsoft Windows seguem o padrão de mesmo nome. As diferentes versões do MS Windows 3.1 e 95 apresentam padrões de interfaces bastante diferentes. Entretanto cada aplicação que é executada em cada uma destas versões possuem aparência e comportamento padronizados, como está ilustrado nas caixas de diálogo para a mesma função imprimir do Netscape. Nas figuras a seguir.

Padrão Windows 3.1



Padrão Windows 95



O padrão não diz respeito a como o software da interface é implementado. Apenas aspectos da interação, justamente a aparência e comportamento, que determinam o que o usuário deve interpretar e como deve agir. Uma das grandes vantagens de um padrão é manter consistente a aparência e o comportamento em diversas aplicações facilitando este processo de interação usuário-sistema.

Um padrão deve ser implementado pelo software da interface. A implementação pode utilizar-se de ferramentas de software que auxiliam na codificação dos widgets que seguem um determinado padrão. Mais adiante, veremos os diversos tipos de ferramentas para interfaces que oferecem apoio à implementaçao de interfaces de um determinado padrão. Existem diferentes ferramentas que permitem implementar um mesmo padrão.

6.2 Design de Interfaces de Usuário

Nesta seção, vamos abordar o design de interfaces de usuário. Este assunto é bastante extenso para ser abordado de maneira exaustiva nesta seção. Longe disso. Vamos apenas discutir alguns pontos principais com o objetivo de dar uma visão geral da área.

6.2.1 A Engenharia Semiótica de Interfaces de Usuário

O foco principal de nossa abordagem para o desenvolvimento de software tem sido as funções da aplicação, chamadas de casos de uso na especificação de requisitos. Cada função opera sobre conceitos do domínio e sua execução é controlada por comandos de função.

Vamos abordar o design de interfaces com a perspectiva que o designer deve comunicar qual foi a solução que ele projetou para o usuário para que ele possa atingir as suas metas. Esta abordagem recebe o nome de **Engenharia Semiótica**, pois aborda o processo de construir uma mensagem utilizando diversos tipos de signos: textos, ícones, símbolos gráficos, sons, gestos, etc. A interface deve ser vista como a mensagem que o designer construiu para comunicar ao usuário o modelo conceitual da aplicação.

A Engenharia Semiótica é, portanto, uma abordagem para o design de interfaces de usuários na qual o sistema computacional é visto como um meio de comunicação através do qual o *designer* envia, para o(s) usuário(s) uma **mensagem** cuja expressão é a interface e o conteúdo é (1) o que o usuário pode fazer com o sistema e (2) como ele pode interagir com o sistema. Esta mensagem não é uma mensagem simples como uma palavra ou frase, mas uma mensagem bastante complexa, pois ela é dinâmica, interativa, multimídia, multi-código e metacomunicativa. Ela é também unidirecional, pois vai sempre do designer para o usuário.

Por exemplo, quando o designer quer que o usuário pressione com o mouse uma determinada área da tela para acionar a execução da função *imprimir* ele pode utilizar o widget *botão de acionamento* (também conhecido como *command button*) com um rótulo escrito *Imprimir*. Este widget tem uma função comunicativa que diz algo equivalente a *"pressione aqui para ativar a função Imprimir"*. Da mesma forma o widget *caixa de texto* comunica ao usuário *"digite um texto"*. Uma moldura pode ser utilizada para comunicar que os elementos inseridos em seu interior pertencem a uma mesma categoria. A janela *caixa de diálogo* pode comunicar quais são as interações que compõem um comando e que necessárias para a execução de função da aplicação.

A perspectiva da Engenharia Semiótica é que cada elemento presente na interface, ícones, botões, sons, palavras ou qualquer outro **signo** tem o potencial de comunicar algo. Cada decisão de design que o designer toma tem um impacto na maneira como o usuário interpreta aquilo que ele quis dizer. **Semiótica** é a disciplina que estuda os signos e os sistemas de significação e de comunicação. A Engenharia Semiótica tem por objetivo apresentar técnicas, métodos e formalismos que apóiem o usuário a construir a interface como sendo uma mensagem formada por signos de diversas natureza (um sistema de signos ou sistema semiótico).

O design de interfaces na abordagem da Engenharia Semiótica consiste em:

- projetar os comandos de função para cada uma das funções,
- comunicar para o usuário quais as ações ele deve fazer em cada comando,
- elaborar representações para os conceitos do domínio,
- comunicar para o usuário quais as funções o sistema oferece e como acessar cada comando.

Vejamos a seguir algumas diretrizes que auxiliam o design de interface nesta abordagem. esta diretrizes não são regras ou leis, mas dicas que auxiliam o designer. O princípio geral que fundamenta toda esta abordagem é que o designer tem que escolher qual o melhor signo para comunicar aquilo que ele quer dizer ao usuário. Isto significa ainda que o designer precisa conhecer o usuário para saber quais tipos de signos serão melhor interpretados por ele.

Nas seções seguintes veremos conceitos e técnicas para realizar cada uma destas atividades.

6.2.2 Projetando comandos de função

Para cada função da aplicação devemos ter um comando de função. Vimos na seção 4.2 que o modelo de interação determina o conjunto de comando de funções da aplicação. Cada comando é formado por uma estrutura de interações básicas. O modelo conceitual de interação descreve esta estrutura de interações de maneira abstrata. Veremos agora como projetar cada comando de função utilizando os objetos de interfaces (widgets) da maioria das ferramentas de interfaces gráficas.

6.2.3 Comunicando comandos de função

Uma vez que o designer tenha escolhido qual função ele quer utilizar, a interface deve permitir que ele controle a execução de cada uma delas apresentando o comando de função.Cada comando deve comunicar as ações para o usuário fornecer operandos, controlar a execução e visualizar resultados. Vejamos cada um deles separadamente.

Em princípio, toda função tem associada a si um comando de função. O comando mais simples possível é aquele que necessário para ativar a execução de uma função que não necessite de operando algum. Um comando assim precisa ter uma única interação básica que pode ser *pressionar uma tecla* ou *pressionar um widget botão de comando*.

6.2.4 Representando conceitos do domínio

O usuário quer que o sistema execute funções para modificar propriedades ou relaciomentos de conceitos do domínio. Estes conceitos precisam ser representados para que o usuário possa percebê-los. Por exemplo, o conceito *cliente* deve ser representado na interface pelo seu nome completo (um texto). Uma outra interface pode representá-lo por uma fotografia associada ao pré-nome.

O conceito de *margem esquerda* de uma página no Word é representado por uma linha pontilhada sobre a página indicando a distância para a borda esquerda. Este conceito também é representado pelo nome *margem esquerda* seguido de um número que representa o valor da *distância para a borda*.

O conceito *pasta* no Windows é representado por um ícone amarelo representando uma pasta e por uma janela, quando o objetivo é mostrar o seu "conteúdo". O conceito de arquivo é representado por ícones que representam o tipo do arquivo e pelo nome do arquivo. No MS-DOS, os arquivos são representados pelo nome com até 8 letras seguido por um ponto e mais três letras representado o seu tipo.

A escolha do conceito não segue nenhuma regra específica. A regra geral é que se deve escolher uma representação que seja mais familiar ao usuário, isto é, que permita ele entender o conceito associando-o a seus conhecimentos prévios. Um recurso bastante utilizado é o de **metáfora**. A metáfora é um recurso utilizado, por exemplo, quando se quer falar sobre algo utilizando um conceito que já seja conhecido do usuário. O *desktop, as pastas e arquivos* do Windows (originalmente do Xerox Star) fazem parte de uma metáfora para induzir o usuário a realizar com o sistema as atividades que ele faz normalmente na sua mesa do escritório.

6.2.5 Organizando e Comunicando funções da aplicação

Normalmente um sistema apresenta dezenas ou centenas de funções da aplicação em um software. Para que o designer comunique todos eles, ele precisa lançar mão de certos recursos de apresentação.

O acesso aos comandos de funções podem ser organizados em menus. Existem diversas formas de menus. Simples, Pull-down, Pop-up que podem ser organizados de forma linear, hierárquica ou com um hipertexto.

Anterior | Índice | Próximo

 Última atualização: 12/04/01

7 Design da Arquitetura de Componentes de Software

Cada função da aplicação que teve o seu comportamento descrito através modelos conceituais deve ser agora descrita em termos de funções, classes, estruturas de dados, etc., chamados de **componentes de software**. Estes componentes que implementam cada função interagem entre si e com os componentes de outras funções da aplicação. Esta estrutura de componentes interconectados entre si que formam o software recebe o nome de **arquitetura de componentes de software**, ou simplesmente arquitetura de software.

Neste capítulo veremos como o modelo conceitual da aplicação pode ser implementado em termos de componentes de software e como estruturar estes componentes de modo a definir a arquitetura do software.

O que será visto neste capítulo:

Os conceitos teóricos apresentados nas diversas seções deste capítulo serão ilustrados a partir de exemplos que apresentam arquiteturas diferentes para uma mesma função da aplicação. Utilizaremos a função da aplicação *Consulta Livro* já introduzida em exemplos do capítulo anterior. Os exemplos estão descritos em páginas independentes para que possamos ter uma visão global e fazer referências diretas através de *links* a partir de cada seção. Estaremos sempre sugerindo uma visita às páginas do exemplo.

7.1 Arquitetura de Software

A arquitetura de software é uma especificação abstrata do **funcionamento** do software em termos de **componentes** que estão **interconectados** entre si. Ela permite especificar, visualizar e documentar a estrutura e o funcionamento do software independente da linguagem de programação na qual ele será implementado. Isto é possível se considerarmos que o software pode ser composto por componentes abstratos - independentes da linguagem - que juntos

formam um software completo que satisfaz os requisitos especificados (a funcionalidade). Estes componentes estão interligados ou interconectados de maneira a interagir e cooperar entre si.

Alguns autores utilizam o termo **configuração** para se referir à maneira como os componentes estão interconectados. **Estrutura** e **topologia** são termos utilizados no mesmo sentido. Para diferenciar a programação do design da arquitetura de software pode-se utilizar, respectivamente, os termos **programação-em-ponto-pequeno** (*programming-in-the-small*) e **programação-em-ponto-grande** (*programming-in-the-large*).

O conceito de arquitetura de software começou a surgir desde que foram desenvolvidas as primeiras técnicas para o particionamento de programas. As técnicas de modularização e decomposição funcional, por exemplo, introduzem os conceitos de funções, procedimentos, módulos e classes que permitem particionar um programa em unidades menores. Estes pedaços, ou componentes, interagem entre si através de chamadas de função ou troca de mensagens, por exemplo, para que o software funcione corretamente.

O particionamento do software em componentes oferece vários benefícios.

- Permite ao programador compreender melhor o software
- Possibilita que estas partes possam ser reutilizadas mais de uma vez dentro do mesmo programa ou por outro programa
- Podem ser gerenciadas mais facilmente quanto estiverem sendo executadas.

Os conceitos e a tecnologia de orientação a objetos consolidaram o conceito de componentes e, conseqüentemente, o de arquitetura de software. Nos anos 90, a arquitetura de software passou a ser um importante tópico de pesquisa. Atualmente, ela faz parte do processo de desenvolvimento de software e dos programas dos cursos de engenharia de software.

A arquitetura não é descrita por um único diagrama. Diversos diagramas descrevem a arquitetura proporcionando diferentes visões do software. Estas visões podem ser estáticas ou dinâmicas, físicas ou lógicas. Os exemplos que apresentamos neste capítulo apresentam diversos diagramas que descrevem a arquitetura do software.

A arquitetura pode descrever tanto a estrutura lógica do funcionamento do software quanto a arquitetura física de componentes físicos que formam o software. A arquitetura lógica descreve o funcionamento lógico do software em termos de funções, variáveis e classes. A arquitetura física descreve o conjunto de arquivos fontes, arquivos de dados, bibliotecas, executáveis e outros que compõem fisicamente o software. A seguir, veremos como os diversos tipos de componentes formam o software.

7.2 Componentes de Software

O termo componente de software está bastante popular atualmente. Diversos autores e desenvolvedores têm usado o termo com propósitos diferentes. Alguns autores utilizam o termo componente para se referir a pedaços do código fonte, como funções, estruturas de dados e classes. Outros chamam de componentes instâncias de classes (objetos) de um programa que podem ser utilizadas por outras instâncias. Há ainda a denominação de componentes para as bibliotecas de funções e bibliotecas de ligação dinâmica.

Para entendermos melhor os diversos tipos de componentes utilizados em engenharia de software vamos identificar alguns critérios de classificação. Na nossa classificação utilizaremos os critérios de componentes *lógicos* (ou funcionais) e *físicos,* de *tempo-de-desenvolvimento e de tempo-de-execução*.

O componente físico é aquele existe para o sistema operacional e para outras ferramentas do sistema, normalmente na forma de arquivos. Eles podem ser armazenados, transferidos de uma lugar para outro, compilados, etc. O componente lógico ou funcional é aquele que possui uma utilidade para o funcionamento do programa.

O componente de tempo-de-desenvolvimento é aquele utilizado durante o desenvolvimento do software, enquanto o de tempo-de-execução é aquele pronto para ser executado pelo sistema ou que está sendo executado. Existem componentes lógicos e físicos tanto de desenvolvimento quanto de execução.

Com base nestes critérios podemos categorizar os componentes em:

- **componentes de programa** - são componentes lógicos de tempo-de-desenvolvimento fornecidos pelas linguagens de programação e que utilizamos para construir um programa.

  Ex.: *tipos de dados, variáveis, procedimentos, funções, classes, módulos, pacotes - dependem da linguagem de programação.*
  Descrevemos as soluções nos nossos exemplos usando componentes lógicos. As soluções A e B utilizam variáveis e funções. A solução C utiliza classes.
- **componentes físicos de desenvolvimento** - são componentes físicos tempo-de-desenvolvimento que contêm os componentes lógicos. Eles são manipulados pelas ferramentas de desenvolvimento (editores e compiladores) e pelo sistema operacional.

  Ex.: *arquivos de código fonte, arquivos de código objeto, arquivos de declarações (.h), bibliotecas de componentes de programa (de ligação estática)*.
  As funções Ler( ) e Escrever( ) da solução A dos exemplos fazem parte de uma biblioteca de funções oferecida pelo compilador. Da mesma forma todas as funções do Xview -- xv_create( ), xv_set( ), xv_get( ), xv_init( ), xv_destroy( ), xv_main_loop( ) -- fazem parte da biblioteca *libxview.a*. O programa da solução B utiliza os arquivos .h *xview.h, frame.h, panel.h* que também são componentes físicos.
- **componentes físicos de tempo-de-execução** - São os componentes instalação e execução que compõem o sistema antes que ele seja executado. São os componentes que obtemos ao adquirir o software.

  Ex.: *arquivos executáveis, arquivos de configuração, arquivos de dados, bibliotecas de ligação dinâmica (DLL)*.
  A biblioteca XView utiliza funções oferecidas por uma outra biblioteca que precisa estar sendo executado ao mesmo tempo que a aplicação. Esta biblioteca de ligação dinâmica é o XServer que oferece os serviços necessários para o desenho de janelas e leitura dos eventos de entrada.
- **componentes lógicos de tempo-de-execução** - São os componentes lógicos que existem quando o sistema está sendo executado ou que são criados a partir da execução de outros componentes. Podem ser de dois tipos:
  - **intraoperáveis -** quando são visíveis apenas por componentes do mesmo programa

    Ex.: *variáveis, funções, objetos de programa.*
  - **interoperáveis -** quando são visíveis por componentes de diferentes programa

    Ex.: *processos, threads, objetos CORBA, objetos COM*.

O notificador é uma função do XServer incorporado ao programa durante a execução função xv_main_loop( ).
O xv_main_loop é um componente lógico de tempo-de-desenvolvimento que faz parte da biblioteca libxview.a -- componente físico de tempo de desenvolvimento -- que é responsável pela interação em tempo-de-execução da função notificador -- componente lógico de tempo de execução -- que faz parte do XServer, uma biblioteca de ligação dinâmica -- componente físico de tempo de execução.

Esta classificação não é a única possível. Além disso, existe um estreito relacionamento entre os diversos tipos de componentes. Os componentes de programa são "manipulados" e referenciados pelo programador durante o processo de programação. Após o processo de compilação e a sua posterior execução eles se transformam em componentes lógicos de tempo-de-execução que são manipulados e referenciados por outros componentes dinâmicos. Os componentes de programas, ditos lógicos, precisam ser armazenados em arquivos de código fonte ou em bibliotecas - componentes físicos de tempo-de-desenvolvimento.

Os componentes lógicos ou funcionais são aqueles que são utilizados para que o software funcione como desejado. Eles são os componentes que permitem que o software faça aquilo que foi especificado. Os componentes físicos não têm papel para a lógica do programa, i.e., para o seu funcionamento. Eles oferecem o apoio necessário para que o software possa ser gerenciado pelo sistema operacional e pelas ferramentas de desenvolvimento e instalação. Daqui por diante concentraremos nossas atenções aos componentes lógicos do software.

7.3 Componentes lógicos

Grande parte dos componentes lógicos de um programa é dependente da linguagem de programação na qual ele será escrito. Na etapa de design, entretanto, nosso objetivo é descrever o funcionamento do software independente de qual linguagem ele será implementado. Vamos utilizar componentes lógicos de programas num nível mais abstrato, independente da linguagem de programação.

No nível abstrato do design da arquitetura, consideremos apenas como tipos de componentes lógicos, as variáveis, as funções ou procedimentos, e as classes (e os módulos). Consideramos que eles são suficientes para projetarmos a arquitetura de componentes de software.

A **variável** é o componente responsável pelo armazenamento de dados. Cada variável tem um nome que a identifica e o tipo de dados que determina o que pode ser armazenado pela variável.

A **função** é a unidade lógica que realiza uma ou mais tarefas específicas modificando o estado de outros componentes de software, especialmente as variáveis. Cada função deve se caracterizar por aquilo que ela faz, isto é, pelo serviço que ela oferece. Cada função é referenciada por um **nome**. A função pode ter **argumentos** que funcionam como interface de comunicação com outros componentes. Estes argumentos podem ser para recebimento e/ou para o fornecimento de dados. A **interface** de uma função é formada pelos seu nome e seus argumentos. Uma função pode **agregar** outras funções. Neste caso as funções agregadas apenas existem para a função que as contêm. Uma função pode **utilizar** outras funções para poder realizar as suas tarefas. A utilização de uma função por outra é realizada através da sua interface.

A **classe** é o componente que agrega variáveis e funções num único componente lógico. Esta agregação deve ser feita seguindo alguns critérios específicos. As funções da classe operam sobre as variáveis da classe e estas por sua vez só podem ser alteradas pelo grupo de funções associadas. Para os componentes externos à classe apenas as funções são visíveis. Podemos dizer que a classe oferece uma série de serviços para operar sobre os dados armazenados nas suas

variáveis. Cada classe é identificada por um nome e possui como interface uma série de funções que são oferecidas. As funções de uma classe podem utilizar serviços das funções de outra classe, mas não podem operar diretamente sobre suas variáveis.

O **módulo** ou **pacote** agrega funções ou classes de acordo com categorias específicas. Um módulo funcional é aquele no qual todos os seus componentes estão unidos visando atingir um aspecto da funcionalidade. Eles possuem dependência funcional. Um módulo de serviços é aquele que agrega componentes que oferecem serviços que não têm necessariamente dependência funcional. Por exemplo, um módulo com funções matemáticas, um módulos com classes para objetos de interfaces gráficas, etc. Cada módulo possui um nome e uma interface que contém as interface de todas as funções e classes que estão nele contidas.

Os componentes lógicos oferecem recursos e restrições para decidirmos o que podemos construir para implementar o software. Na prática estes recursos podem variar de linguagem para linguagem, mas em geral, eles são comuns a maioria das linguagens. Com os componentes abstratos podemos nos concentrar na solução sem preocupação com as particularidades de cada linguagem. Entretanto, alguns componentes não poderão ser implementados diretamente em uma determinada linguagem.

Existem diversos paradigmas de programação. Podemos citar os paradigmas de programação procedimental, funcional, orientado a objetos e lógica. No paradigma de orientação a objetos o programador tem recursos que não encontra em linguagens no paradigma de programação procedimental e funcional. Ao realizar o design da arquitetura em termos de componentes o engenheiro pode optar apenas pelos componentes lógicos que poderão ser implementados pela linguagem de programação a ser utilizada. Assim, pode-se optar por utilizar apenas funções se a programação será no paradigma procedimental. Da mesma forma, pode-se considerar classes como componente lógico se o paradigma de programação for orientado a objetos. Não vamos considerar o paradigma de programação lógica por não ser muito utilizado em desenvolvimento de software.

7.4 Princípios para o design da arquitetura e seus componentes

Uma engenharia se faz com métodos, técnicas, ferramentas, formalismos e **princípios**. Diversos conceitos e princípios ligados a estes conceitos têm sido introduzidos ao longo da história da engenharia de software. Um princípio gira em torno de uma idéia elaborada a partir de experiências anteriores e que aponta um caminho ou solução para os problemas no desenvolvimento. Seguir um princípio não é uma obrigação, mas uma grande chance de se obter sucesso.

Vamos apresentar alguns princípios para o design da arquitetura do software e de seus componentes que foram introduzidos na engenharia de software e que são incorporados em diversas linguagens e formalismos.

Abstração

A abstração é a capacidade psicológica que os seres humanos têm de se concentrar num certo nível de um problema, sem levar em consideração detalhes irrelevantes de menor nível. A abstração deve ser utilizada como técnicas de resolução de problemas nas diversas áreas de engenharia. As linguagens de modelagem e de programação oferecem recursos para que possamos trabalhar a abstração.

No design da arquitetura esta técnica é fundamental. Nosso objetivo é encontrar uma solução funcional de software em termos de componentes abstratos, sem levar em consideração detalhes das linguagens de programação.

A abstração está presente em quase tudo na computação. O sistema operacional oferece recursos que abstraem o funcionamento do computador. Um programa escrito numa linguagem de programação algorítmica (Pascal ou C) oferece comandos que abstraem a maneira como o computador executaria o comando em linguagem de máquina. O conceito de função permite abstrair os detalhes de implementação da função para que possamos nos concentrar apenas naquilo que a função faz. O uso de diagramas que descrevem o software oferece uma visão abstrata do funcionamento, independente de como ele está implementado.

Modularização

A modularização é uma técnica utilizada em diversas áreas da engenharia para se construir um produto que seja formado por componentes, os módulos, que possam ser montados ou integrados. Esta técnica permite que o esforço intelectual para a construção de um programa possa ser diminuído. Ela também facilita o processo de compilação e execução de um programa. Pode-se compilar componentes separadamente, bem como interligá-los apenas durante a execução quando for necessário.

A modularização é um dos caminhos para garantir a abstração. Podemos nos referir a componentes específicos e utilizar alguns dos seus serviços sem preocupação de como ele foi implementado. Com o conceito de componente, estamos dizendo como uma certa unidade computacional que abstrai certos detalhes pode ser utilizada na composição de outros componentes. Para isto o componente precisa ser referenciado por um nome e precisa ter uma interface que diga como ele pode ser "interconectado" a outros componentes.

Diversos tipos de componentes podem ser encontrados na maioria das linguagens e ferramentas de programação. As variáveis de dados, as funções e as classes de um programa; as bibliotecas de funções e de classes; os arquivos fontes, executáveis e de dados; e diversos outros.

Encapsulamento

Para que a abstração seja implementada com maior rigor, cada componente (módulo) do software deve encapsular todos os detalhes internos de implementação e deixar visível apenas a sua interface. A interface do componente deve dizer aquilo que ele faz, o que ele precisa para se interconectar com outros componentes, e o que ele pode oferecer para os outros componentes.

Este princípio, também chamado de *ocultação de informação (information hiding)* sugere que os componentes sejam projetados de tal modo que os seus elementos internos sejam inacessíveis a outros componentes. O acesso apenas deve ser feito pela interface. Isto garante a integridade do componentes, uma vez que evita que seus elementos sejam alterados por outros componentes.

Reutilização

Além de facilitar o processo de desenvolvimento ao diminuir o esforço intelectual ou facilitar a compilação, os componentes podem ser reutilizados em diferentes software. Uma função que tenha sido construída para um software específico pode ser reutilizada em um outro software. A funcionalidade específica de cada componentes será utilizada para determinar a funcionalidade global do software. Software com diferentes funcionalidades globais podem ser construídos alguns componentes específicos comuns.

Normalmente os componentes reutilizáveis (tipos de dados, funções ou classes) são armazenados em um outro componentes não-funcional chamados de bibliotecas. Os componentes podem ser incorporados durante a compilação ou durante a execução. No primeiro caso os componentes ficam em bibliotecas de compilação ou de ligação estática e no segundo nas bibliotecas de ligação dinâmicas.

Generalização

A construção de componentes ou módulos específicos que facilitem o processo de desenvolvimento de software deve seguir um outro princípio importante: a generalização. Para que um componente de software tenha utilidade em diversos programas diferentes ele deve ser o mais genérico possível. Para isto ele precisa ser construído com o objetivo de oferecer serviços de propósito geral.

Por exemplo, uma função que desenha na tela um retângulo de qualquer tamanho é mais genérica do que uma função que desenha um retângulo com tamanho fixo. Esta função poderia ser ainda mais genérica se permitisse que o desenho de um retângulo com bordas com diversas espessuras e cores.

7.5 Modelando a arquitetura usando a UML

A arquitetura é essencialmente um modelo abstrato que descreve a estrutura de componentes que compõe um software e o seu funcionamento. Este modelo deve ser descrito através de notações e linguagens apropriadas. Diversas **linguagens de descrição de arquitetura** têm sido propostas, entretanto nenhuma ainda firmou-se como uma linguagem padrão.

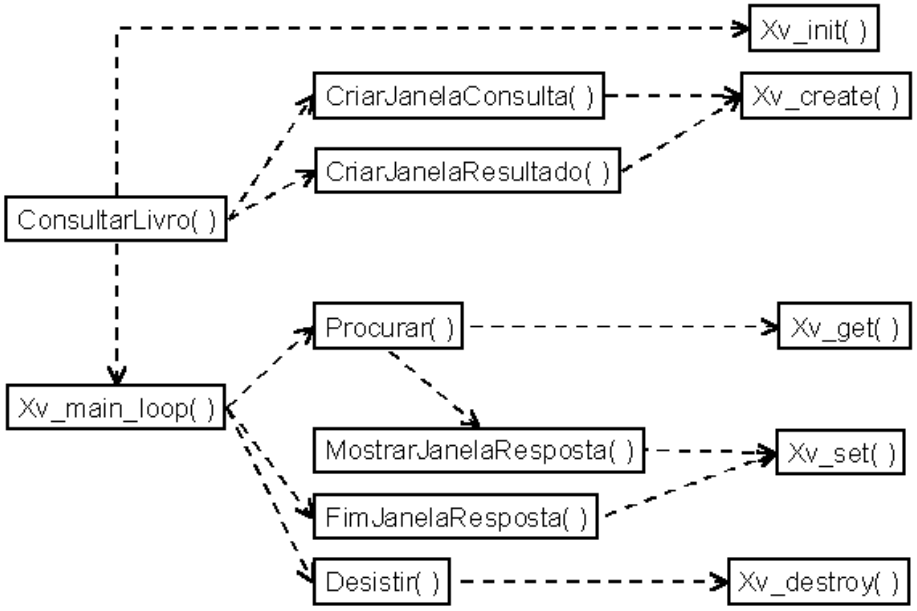
Vimos que a UML foi proposta como uma linguagem para especificação, visualização, construção e documentação de sistemas de software e pode ser utilizada em diversas etapas do desenvolvimento. A UML não foi elaborada com a finalidade de descrever a arquitetura do software. Entretanto, ela apresenta diversos diagramas que permitem representar a estrutura lógica e física do software em termos de seus componentes, bem como a descrição do seu comportamento.

Vamos utilizar alguns dos diagramas UML para descrever a arquitetura lógica do software. É preciso ressaltar, porém, que a UML foi desenvolvida para a modelagem de software desenvolvido no paradigma de orientação a objetos. Para representarmos componentes funções é preciso fazer algumas adaptações. Outro aspecto importante é que na UML o termo componente refere-se a componentes físicos.Na UML existe uma notação específica para a representação de componentes físicos e diagramas que descrevem a arquitetura em termos destes componentes. Os componentes lógicos são representados como através de diagramas de classes e de objetos.

7.5.1 Modelando a dependência de funções

As funções que compõem um software podem ser modeladas usando um diagrama de objetos simplificado. O diagrama de funções é semelhante ao diagrama de objetos e descreve a relação de dependência entre as funções. Neste diagrama cada função é representada por um retângulo e a dependência entre elas é indicada por uma seta tracejada. Esta dependência indica que uma determinada função **USA** uma outra função.

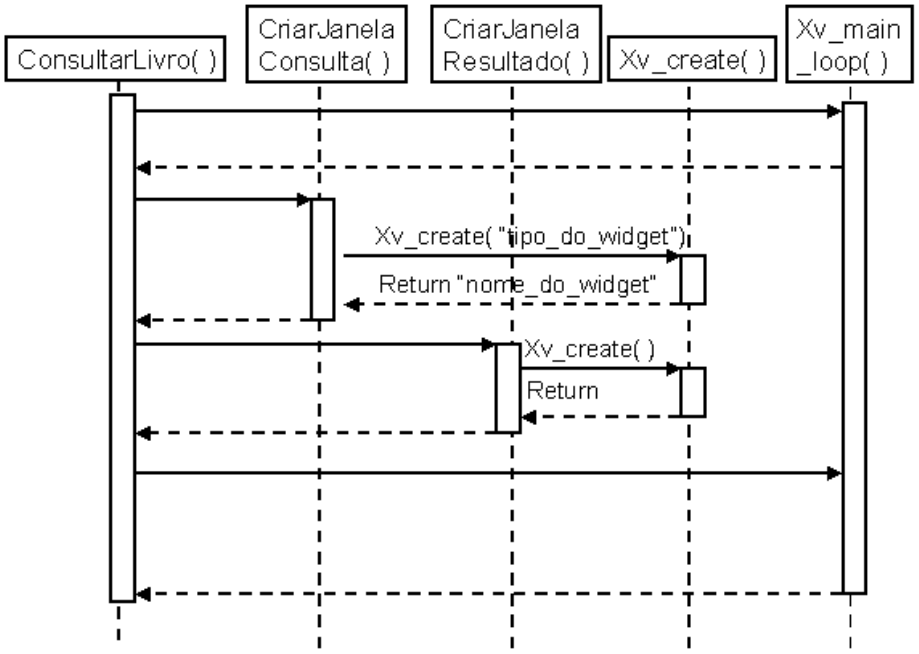
A solução B dos exemplos é composta por diversas funções que possuem dependência entre si, como mostra a figura abaixo.



Este diagrama mostra apenas que uma função usa uma ou outras funções. A função ConsultarLivro( ) usa as funções xv_init( ), xv_main_loop( ), CriarJanelaConsulta( ) e CriarJanelaResposta( ). Estas duas últimas funções usam xv_create( ).

7.5.2 Usando o diagrama de interação

Para mostrar como as funções interagem entre si podemos utilizar o diagrama de interação da UML. Este diagrama mostra como as funções interagem entre si. Este diagrama possui duas formas: o **diagrama de seqüência** e o **diagrama de colaboração**. o diagrama de seqüência mostrar como as interações entre as funções ocorrem ao longo do tempo. A figura abaixo ilustra um diagrama de seqüência .

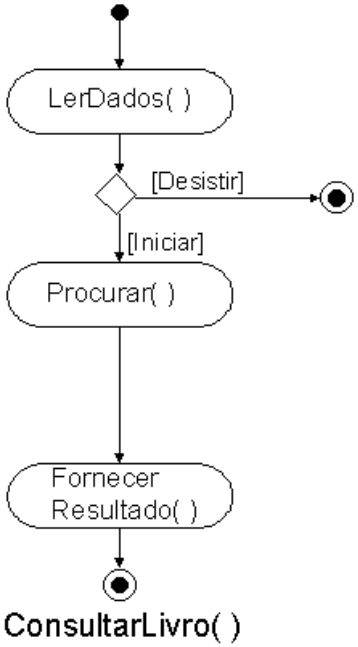


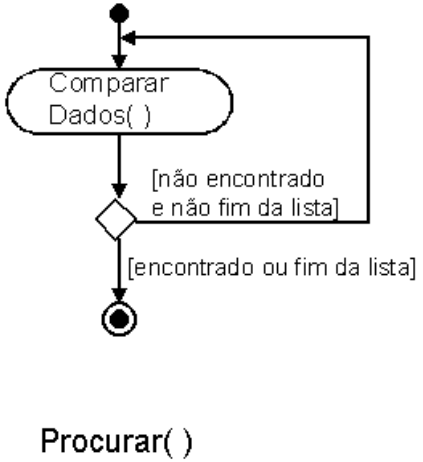
Cada função é representada por um retângulo e são alinhadas na parte superior. De cada retângulo parte uma linha pontilhada indicando a passagem do tempo de cima para baixo. O tempo no qual cada função está *ativa* é representado por um retângulo vertical sobre a linha do tempo. Quando uma função *chama* uma outra função ela *ativa* esta função. Isto é representado no diagrama por uma linha saindo da função que chama e chegando na linha de tempo da função chama. O retêngulo vertical que indica a ativação da função começa a partir deste ponto. O retorno à função que fez a chamada é representado por uma seta tracejada. Cada uma das setas horizontais de chamada e retorno pode indicar quais valores (dados) são passados ou retornados entre as funções.

7.5.3 Usando diagrama de atividades

As atividades internas a cada função pode ser descrita através de diagramas de atividades. Estes diagramas descrevem cada passo da função e permitem representar estruturas de decisões e repetições internas a cada função. As figuras abaixo descrevem os diagrama sde atividades para as funções ConsultarLivro(

) e Procurar( ), da solução A dos exemplos.





7.6 Estilos e Padrões de Design Arquitetural

Padrões são soluções para problemas específicos que ocorrem de forma recorrente em um determinado contexto que foram identificados a partir da experiência coletiva de desenvolvedores de software. A proposta original de padrões veio do trabalho de Christopher Alexander na área de arquitetura (para construções). Sua definição para padrões:

> *Cada padrão é uma regra (esquema) de três partes que expressa uma relação entre um certo contexto, um problema, e uma solução.*

O **contexto** descreve uma situação no desenvolvimento na qual existe um problema. O **problema**, que ocorre repetidamente no contexto, deve também ser descrito bem como as **forças** (requisitos, restrições e propriedades) associadas a ele. A **solução** descreve uma configuração ou estrutura de componentes e suas interconexões, obedecendo às forças do problema.
As **forças**, denominação dada por [Alexander], descrevem os requisitos que caracterizam o problema e que a solução deve satisfazer, as restrições que devem ser aplicadas às soluções e propriedades desejáveis que a solução deve ter.

7.6.1 Padrões no desenvolvimento de software

Os padrões são desenvolvidos ao longo dos anos por desenvolvedores de sistemas que reconheceram o valor de certos princípios e estruturas organizacionais de certas classes de software.

O uso de padrões e estilos de design é comum em várias disciplinas da engenharia. Eles são codificados tipicamente em manuais de engenharia ou em padrões ou normas

Em software podemos ter padrões a nível conceitual, a nível de arquitetura de software ou a nível de algoritmos e estruturas de dados.O padrões podem ser vistos como tijolos-de-construção mentais no desenvolvimento de software. Eles são entidades abstratas que apresentam soluções para o desenvolvimento e podem ser instanciadas quando se encontra problemas específicos num determinado contexto.

Os padrões se diferenciam de métodos de desenvolvimento de software por serem dependentes-do-problema ao passo que os métodos são independentes-do-problema. Os métodos apresentam passos o desenvolvimento e notações para a descrição do sistema, mas não abordam como solucionar certos problemas específicos.

Um sistema de software não deve ser descrito com apenas um padrão, mas por diversos padrões relacionados entre si, compondo uma arquitetura heterogênea.

Os padrões podem ser descritos em sistemas ou catálogos de padrões. Num sistema os padrões são descritos de maneira uniforme, são classificados. Também são descritos os seus relacionamentos. Num catálogo os padrões são descritos de maneira isolada.

7.6.2 Categorias de padrões e Relacionamentos

Cada padrão depende de padrões menores que ele contém e de padrões maiores no qual ele está contido.

- **Padrões arquiteturais** - expressam o esquema de organização estrutural fundamental para um sistema de software. Assemelham-se aos Estilos Arquiteturais descritos por [Shaw & Garlan 96].
- **Padrões de design** - provê um esquema para refinamento dos subsistemas ou componentes de um sistema de software.
- **Idiomas** - são padrões de baixo nível, específicos para o desenvolvimento em uma determinada linguagem de programação, descrevendo como implementar aspectos particulares de cada componente.

Os diversos padrões podem ser relacionados entre si. Três relações são identificados: *refinamento, variante e combinação*.

- **Refinamento**: Um padrão que resolve um determinado problema pode, por sua vez, gerar novos problemas. Componentes isolados de um padrão específico podem ser descritos por padrões mais detalhados.
- **Variante**: Um padrão pode ser uma variante de um outro. De uma perspectiva geral um padrão e suas variantes descrevem soluções para problemas bastante similares. Estes problemas normalmente variam em algumas das forças envolvidas e não na sua caracterísitica geral.

- **Combinação**: Padrões podem ser combinados em uma estrutura mais complexa no mesmo nível de abstração. Tal situação pode ocorrer quando o problema original inclui mais forças do que podem ser balanceadas em um único padrão. Neste caso, combinar vários padrões pode resolver o problema, cada um satisfazendo um conjunto particular de forças.

7.6.3 Descrição de padrões

Cada padrão pode ser descrito através do seguinte esquema:

- **Nome** - o nome do padrão e uma breve descrição.
- **Também conhecido como** - outros nomes, se algum for conhecido.
- **Exemplo** - um exemplo ilustrativo do padrão.
- **Contexto** - as situações nas quais o padrão pode ser aplicado.
- **Problema** - o problema que o padrão aborda, incluindo uma discussão das forças associadas.
- **Solução** - o princípio fundamental da solução por trás do padrão
- **Estrutura** - um especificação detalhada dos aspectos estruturais.
- **Dinâmica** - cenários descrevendo o comportamento em tempo-de-execução.
- **Implementação** - diretrizes para implementar o padrão.
- **Exemplo resolvido** - discussão de algum aspecto importante que não é coberto nas descrições da Solução, Estrutura, Dinâmica e Implementação.
- **Variantes** - Uma breve descrição das variantes ou especializações de um padrão.
- **Usos conhecidos** - Exemplos do uso obtidos de sistemas existentes.
- **Conseqüências** - Os benefícios que o padrão proporciona e potenciais vantagens.
- **Ver também** - Referências a padrões que resolvem problemas similares e a padrões que ajudam refinar os padrões que está sendo descrito.

7.6.4 Exemplos

Vamos apresentar, de forma sucinta, alguns exemplos de padrões de cada uma das categorias. A descrição detalhada de cada padrão pode ser vista em [Buchmann et al. 96].

Model-View-Controller

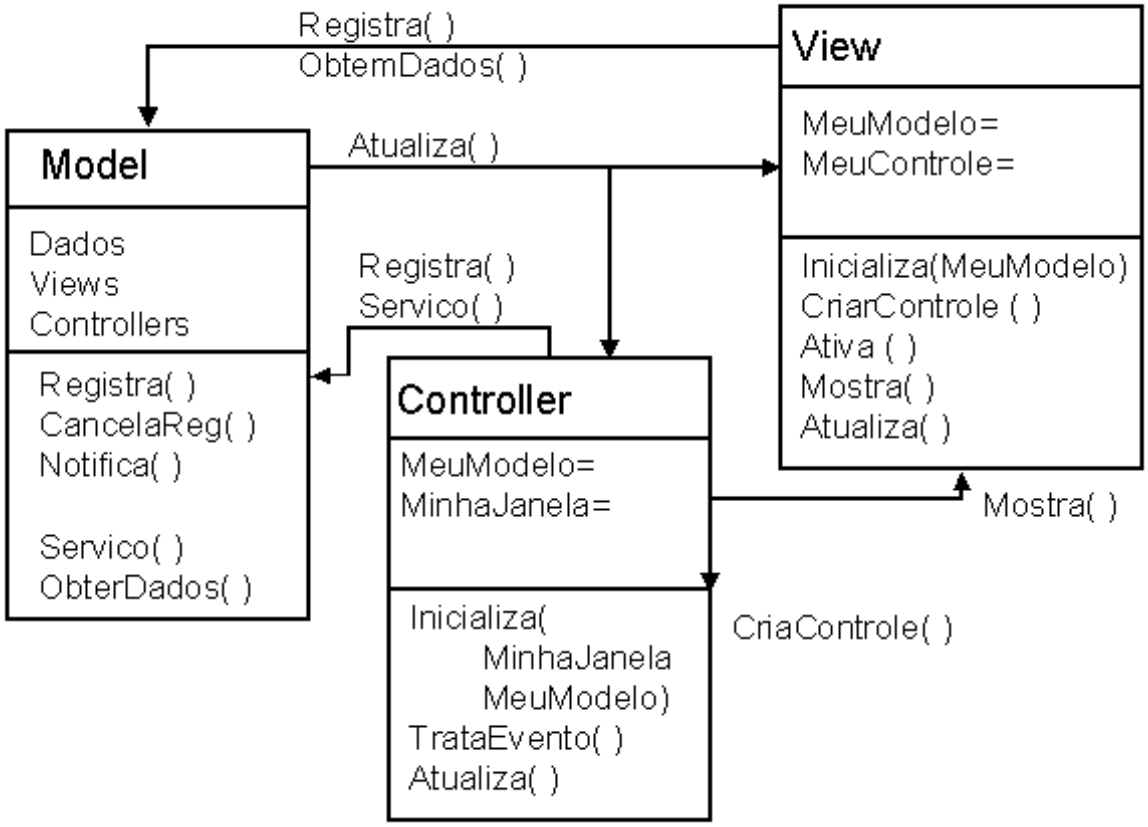
Contexto: Aplicações interativas com interface de usuário gráfica (IUG) flexível.

Problema: As IUGs mudam bastante, sejam por modificações na funcionalidade que requer uma mudança nos comandos ou menus, na necessidades de alterações na representação dos dados por necessidades dos usuários, ou na utilização de uma nova plataforma com um padrão diferente de look & feel. As seguintes forças influenciam a solução.

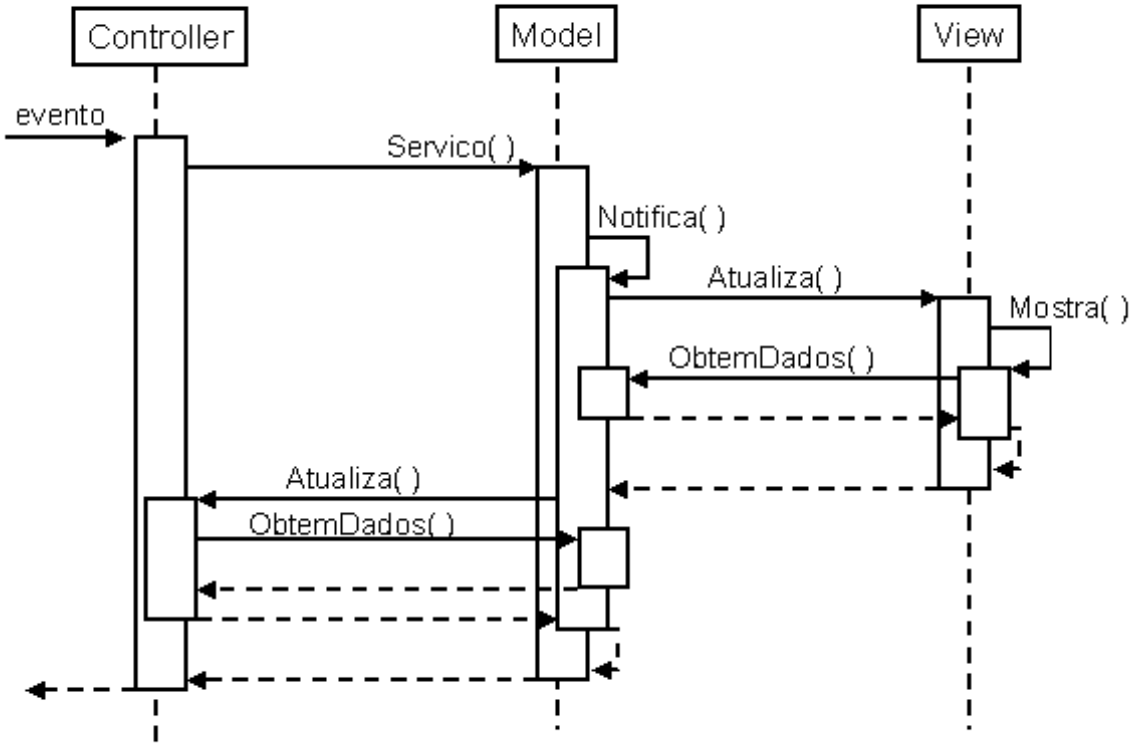
- Uma mesma informação pode ser representada de maneira diferente e em janelas diferentes.

- As representações das informações devem refletir na tela imediatamente as mudanças que ocorrem no sistema.
- As mudanças na IUG devem ser fáceis de fazer e em tempo-de-execução.
- O suporte a diferentes padrões de look & feel não devem afetar o código do núcleo funcional.

Solução: MVC divide a aplicação em três áreas: processamento, saída e entrada.



O componente **Model** encapsula o núcleo da funcionalidade e os dados que ela manipula. Ele é independente das representações específicas mostradas na saída. O componente **View** mostra informações ao usuário a partir de dados obtidos do Model. Podem existir múltiplas visualizações do modelo. Cada View possui um **Controller** associado. Cada Controller recebe eventos da entrada e traduz em serviços para o Model ou para o View.



Estrutura: O Model encapsula os dados e "exporta" as funções que oferecem serviços específicos. Controllers chamam estas funções dependendo dos comandos do usuário. É papel do Model prover funções para o acesso ao seus dados que são usadas pelos componentes View para poderem obter os dados a serem mostrados.

O mecanismo de propagação de mudanças mantém um registro de todos os componentes que são dependentes dos componentes do modelo. Todos os componentes View e alguns Controllers selecionados registram que querem ser informados das mudanças no Model. O mecanismo de propagação de mudanças é a única ligação do Model com Views e Controllers. Cada View define um procedimento de update que é ativado pelo mecanismo de propagação de mudanças. Quando este procedimento é chamado, o componente View recupera os valores atuais do modelo e mostra-os na tela. Existem uma relação um-para-um entre Views e Controllers para que seja possível a manipulação destes últimos sem alteração Models.

Se o comportamento de um Controller é dependente do Model ele deve registrar-se no mecanismo de propagação de mudanças. Isto pode ocorrer quando, por exemplo, as opções de um menu podem ser habilitadas ou desabilitadas de acordo o estado do sistema.

Utilizando este padrão elaboramos uma terceira arquitetura - solução C - para a implementação da função Consultar Livro( ).

Baseado em eventos (ou chamada implícita)

Na solução B dos exemplos que utiliza as funções do XView para utilizar o sistemas de janelas XWindows elaboramos uma arquitetura de software que segue um padrão chamado de arquitetura **baseada em eventos**. Este padrão caracteriza-se pela presença de um componente, chamado notificador ou gerente de eventos, que associa eventos do sistemas a funções ou procedimentos do sistema (que podem ser funções ou métodos de um objeto). Ao invés de chamar um procedimento ou função explicitamente, um componente anuncia um ou mais eventos. Estes componentes são chamados de anunciantes. Outros componentes do sistema registram no notificador um procedimento ou função a ser associada a ele. Quando um evento é anunciado o notificador chama todos os procedimentos que tenham sido registrado com aquele evento. Uma característica deste padrão é que não se tem um fluxo de execução pré-definido, a ordem de execução é depende de quais eventos aconteceram.

As principais vantagens são reuso, evolução e manutenção de componentes, um vez que os componentes podem ser alterados realizando-se apenas modificações nos registros do notificador.

Desvantagens:

- Difícil correção uma vez que não existe um fluxo pré-definido, nem se sabe com precisão quais as pré- e pós-condições quando uma função ou precidemento é chamado pelo notificador.
- Torna-se difícil a passagem de dados dos componentes que geram eventos para os que serão chamados pelo notificador.
- Quando um componente anuncia um evento ele não pode garantir que os outro componentes irão responder.

Sistemas em camadas

- Um sistema em camadas é organizado hierarquicamente com cada camada oferecendo serviços para camadas superiores e utilizando serviços de camadas inferiores.
- Alguns sistemas apenas permitem que os serviços de uma camada apenas sejam acessados por camadas imediatamente superior, i.e., a interação apenas ocorre entre duas camadas adjacentes.
- O modelo computacional é o de que cada camada, a partir da inferior, implementa uma máquina virtual que oferece serviços, a partir de serviços oferecidos pelas máquinas de níveis inferiores. São considerados componentes as diversas máquinas virtuais oferecidas em cada camada.
- Os conectores são definidos pelos protocolos que determinam com as camadas irão interagir. A maioria do sistemas computacionais com este estilo implementam conexões através de chamadas a funções oferecidas por camadas anteriores.
- Os principais exemplos de aplicação incluem sistemas operacionais e sistemas de bancos de dados.
- Podemos destacar algumas propriedades interessantes:
    - Possibilita um processo de **design incremental** - um problema complexo pode ser particionado em níveis crescente de abstração, resolvendo-os num seqüência do mais baixo para o mais alto nível.

    - Oferecem suporte para **reuso.** Pode-se projetar um sistema reusando camadas inferiores e implementando apenas camadas de mais alto nível de abstração.
    - Oferecem suporte para **evolução e manutenção.** Uma vez definidas as interfaces (os protocolos de interação) entre as diversas camadas pode-se acrescentar facilmente novas camadas superiores a um sistema, ou modificar os componentes de camadas existentes por outras mais atualizadas.
- Como desvantagens temos:
    - Nem todo problemas pode ser facilmente particionado em camadas. Muitas vezes é difícl encontrar o nível certo de abstração para cada componente.
    - A performance pode ser afetada quando camadas de mais alto nível precisa interagir com camadas de níveis inferiores.

Tubos e Filtros

- Cada componente, o filtro, tem um conjunto de entradas e saídas.
- Um filtro lê um stream de dados de suas entradas e produz streams de dados em suas saídas.
- Cada filtro realiza uma computação que transforma o stream de dados de entrada no stream de dados de saída
- O filtro gera dados na saída antes mesmo dos dados da entrada terem sido totalmente consumidos
- Os tubos são conectores que interligam a saída de um filtro da entrada de outro.
- Uma configuração de filtros e tubos determinam um modelo computacional formado por elementos que transformam (filtram) dados recebidos nas suas entradas e geram dados nas suas saídas, podendo ser executados sequencialmente ou concorrentemente..
- Os invariantes mais importantes deste estilos determinam que filtros sejam entidades independentes, não compartilhando estados com outros componentes.
- Um outro invariante determina que um filtro não tem conhecimento de quais filtros que lhe fornecem dados ou quais os que recebem seus dados.
- Pipelines são uma especialização deste estilo no qual os filtros são conectados numa seqüência linear.
- O exemplo mais conhecido de aplicação deste estilo são configurações definidas por usuários de shell do unix.
- Propriedades:
    - É fácil entender o comportamento geral do sistema, conhecendo-se o comportamento de cada filtro.
    - Incentiva reusabilidade dos componentes.
    - Fácil evolução e manutenção - filtros podem ser trocados e modificados sem alterar outros componentes do sistema.
    - Suporte a execução corrente - um filtro pode ir gerando dados na saída enquanto outro está lendo estes dados através da sua entrada conectada por um tubo.
- Desvantagens: tende a gerar processamento em lote; não adequado para aplicações interativas; necessidade de adequar dados do stream para um determinado filtro.

Estratégia

Frequentemente é necessário usar algoritmos alternativos num programa. Um exemplo disto pode ser um programa para compressão de dados que oferece mais de uma alternativa de compressão com diferentes utilização de espaço e tempo dos recursos do sistema. Uma solução elegante é encapsular os algoritmos em diferentes classes intercambiáveis que podem ser instanciadas quando necessárias.

O padrão **Estratégia** encapsula algoritmos alternativos para uma mesma funcionalidade. Neste padrão, o componente EstratégiaAbstrata é uma classe abstrata que define uma estratégia comum para todas as estratégias. Parte da sua interface é o metódo **InterfaceDoAlgoritmo()**, que deve ser implementado por todas as subclasses. Cada **EstratégiaConcreta** é uma subclasse de **EstratégiaAbstrata** e implementa cada algoritmo específico na especialização do método **InterfaceDoAlgoritmo**().

Contexto() é o procedimento que utiliza os múltiplos algoritmos e contém uma instância de uma das estratégias. Quando o algoritmo precisa ser aplicado, **Contexto**() chama o método **Interface-do-Contexto**(), que por sua vez chama a instância correspondente do método **InterfaceDoAlgoritmo**() da classe EstratégiaAbstrata.

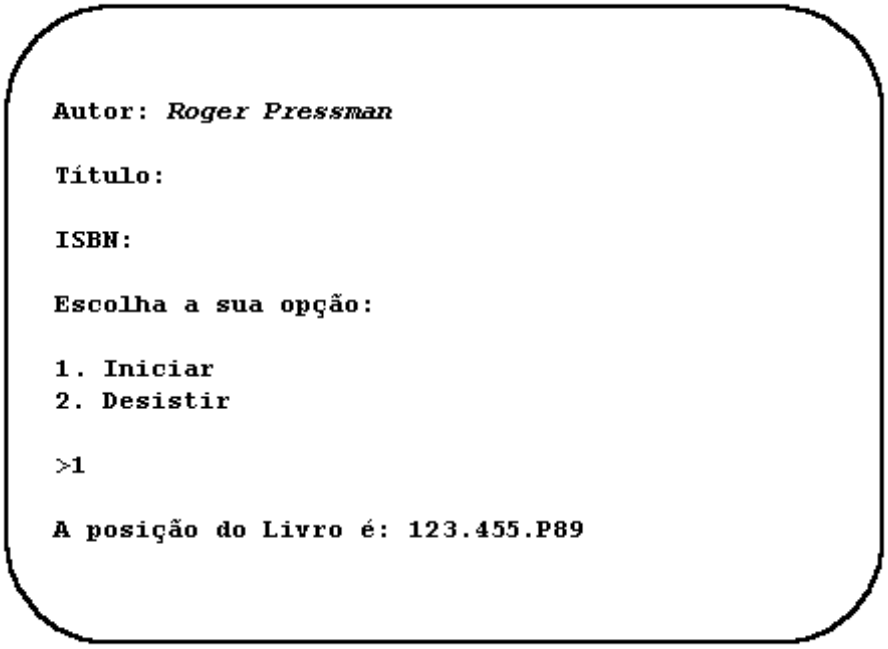
7.7 Exemplos de arquiteturas

Para ilustrar os conceitos teóricos deste capítulo vamos apresentar duas arquiteturas distintas que implementam uma única função da aplicação de um sistema de biblioteca. Vamos utilizar a especificação conceitual para a função *Consultar Livro* apresentada no capítulo 4. Para esta função vamos apresentar duas soluções diferentes. Evidentemente, as soluções apresentadas são bastante simples e estão num escopo bastante pequeno.

Devido à esta simplicidade, pode parecer que o design da arquitetura de software é perda de tempo no desenvolvimento e que seria melhor partirmos direto para a implementação. Num caso simples, talvez sim, mas o objetivo da arquitetura de software é oferecer recursos para a especificação, visualização e documentação de software complexo com centenas ou milhares de linhas de código. Nosso objetivo é ilustrar concretamente como o design da arquitetura de componentes permite abstrair detalhes de programação para chegarmos a solução de software desejada.

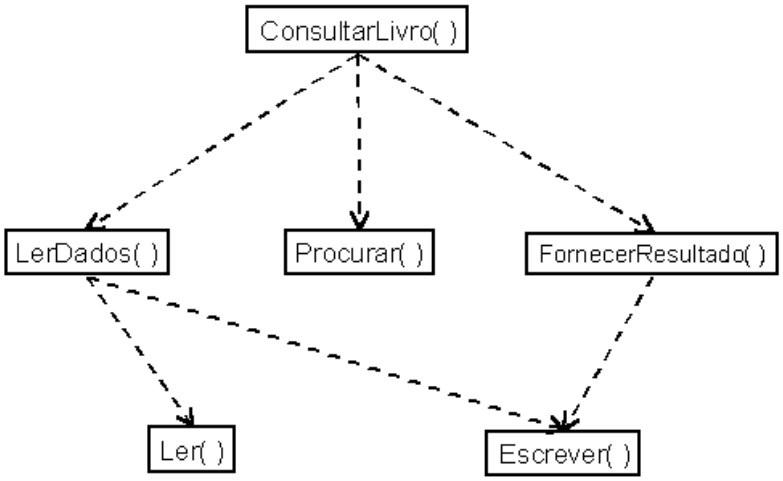
7.7.1 Solução A

Vejamos uma solução bastante simples de arquitetura para implementar a função da aplicação *Consultar Livro*. Este programa poderia ser formado pelas funções: *LerDados*( ), *Procurar*( ), *FornecerResultado*( ). Esta solução não utiliza interfaces WIMP e produz uma interface que tem a seguinte aparência.

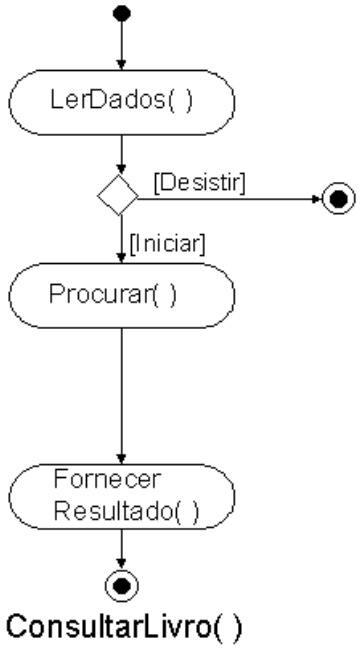


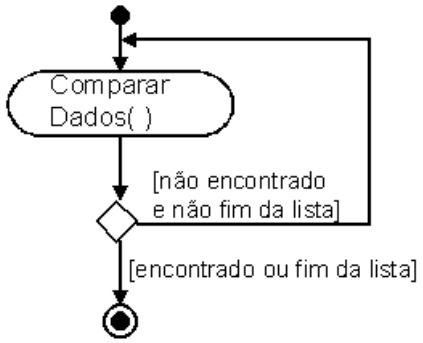
A função *LerDados*( ) deve solicitar ao usuário os dados de *autor*, *título* e *ISBN* do livro. A função *Procurar*( ) deve procurar na base de dados a localização do livro na biblioteca. *FornecerResultado*( ) deve mostrar na tela a localização do livro para o usuário. Estes três componentes podem ser agrupados em uma função principal responsável por fazer a chamada a cada uma das funções em seqüência. Estas funções utilizam outros dois componentes: *Ler*( ) e *Escrever*( ), que fazem parte de uma biblioteca de funções de entrada e saída disponíveis no compilador utilizado.

Representando cada componentes função por um quadrado e a relação de dependência entre uma função que utiliza uma outra podemos ter a seguinte diagrama que descreve arquitetura estática (configuração) de componentes.



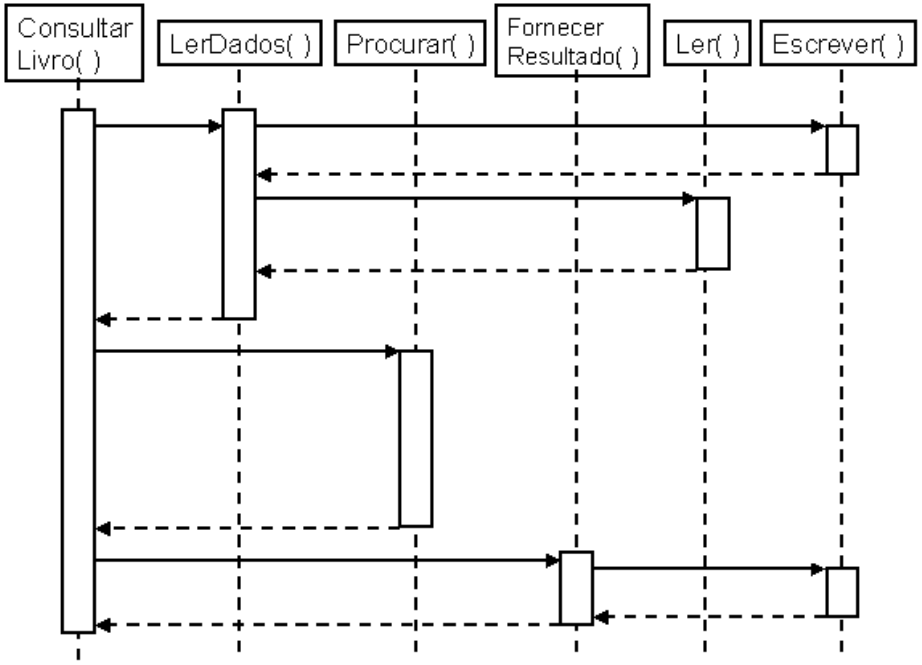
As funções ConsultarLivro( ) e Procurar( ) podem ser descrita pelos seguintes diagrama de atividades.





Procurar( )

Estes componentes interagem entre si da seguinte forma.



Vejamos os algoritmos que implementam estes componentes funções.

Estrutura de dados

```
struct lista_info_livro {

        char autor[30];
        char titulo[30];
        char isbn[30];
        char posicao[10];
        lista_info_livro *proximo;

}
```

Funções

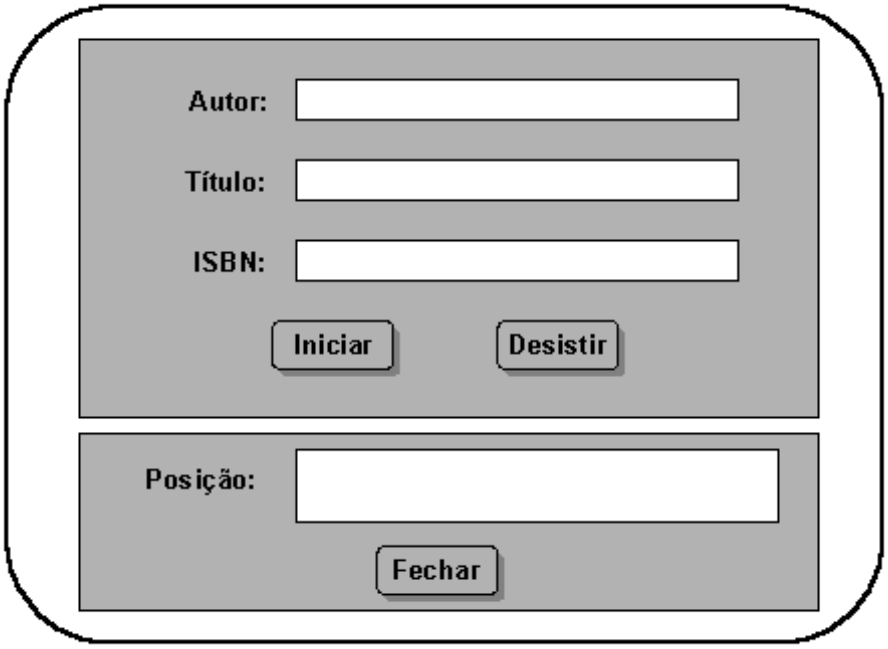
```
ConsultaLivro( ) {

        string autor,titulo,isbn,posicao;
        int escolha;

        LerDados(autor,titulo,isbn,escolha);
        if (escolha = 1) {

                Procurar(autor,titulo,isbn,posicao);
                FornecerResultado(posicao);
        }
        LimparTela( );

}

LerDados(out:autor,out:titulo,out:isbn,out:escolha) {

        Escreva("Autor:");
        Leia(autor);
        Escreva("Titulo:");
        Leia(titulo);
        Escreva("ISBN:");
        Leia(isbn");
        Escreva("1. Iniciar");
        Escreva("2. Desistir");
        Leia(escolha);

}

Procurar (in:autor, in:titulo, in:isbn, out: posicao) {

        info_livros *lista_info_livros;

        info_livros = inicio_lista;
        while (info_livros != NULL)

                if (autor = = info_livros.autor) ||
                   (titulo = = info_livros.titulo) ||
```

```
            (isbn = = info_livros.isbn)
          then
            posicao = info_livros.posicao
          else
            posicao = NULL;
      }

}

FornecerResultado(in:posicao) {

      if (posicao != NULL)
      then
            Escreva("A posicao do livro na  estante é:", posicao);
      else
            Escreva("Livro não encontrado");

}
```

Esta solução não satisfaz complementamente o modelo de interação que foi especificado no capítulo 4. Isto ocorre por limitações da linguagem, dos componentes de bibliotecas que foram utilizados e da maneira como eles podem ser interligados.

7.7.2 Solução B

Vamos apresentar agora uma solução envolvendo interfaces gráficas no estilo WIMP. O usuário vai poder fornecer as informações em uma janela de diálogo (*janela consulta*) na qual ele pode interagir com qualquer elemento independente de ordem. O resultado é apresentado em uma outra janela (*janela resultado*). As duas janelas podem ser vistas na figura abaixo.

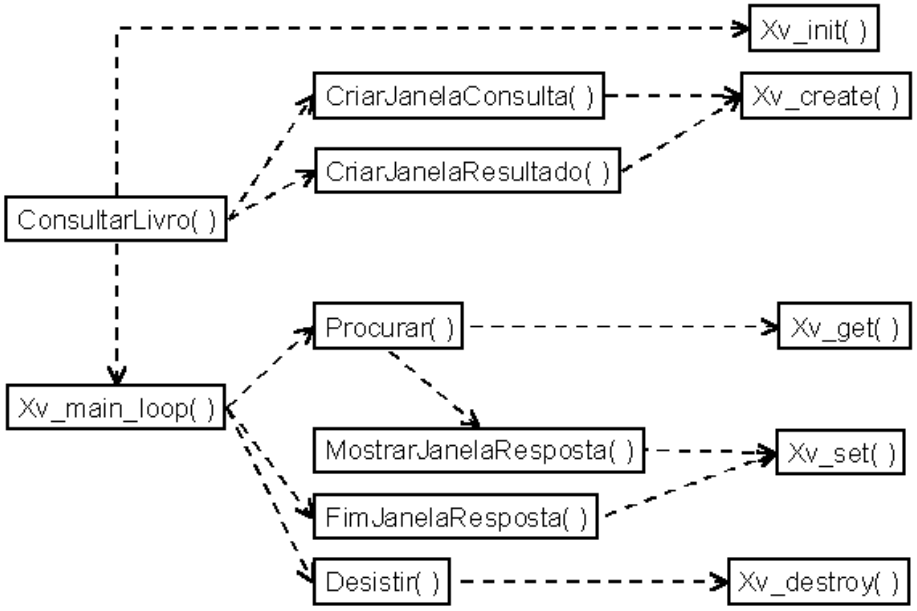


Uma arquitetura para esta mesma função da aplicação poderia ser construídas pelos componentes: *ConsultarLivro*( ), *DesenharJanelaConsulta*( ), *JanelaResultado*( ), *Procurar*( ), *Desistir*( ) que seriam construídas numa linguagem procedimental (como C, por exemplo) e utilizariam funções de uma biblioteca para a construção de widgets de interfaces gráficas, um *toolkit* (veja seção 4.3), chamada de XView. As funções desta biblioteca são incorporadas ao programa em tempo-de-execução. Ela é portanto uma biblioteca de ligação dinâmica (DLL) que é um componente físico de tempo-de-execução(veja seção 7.4).

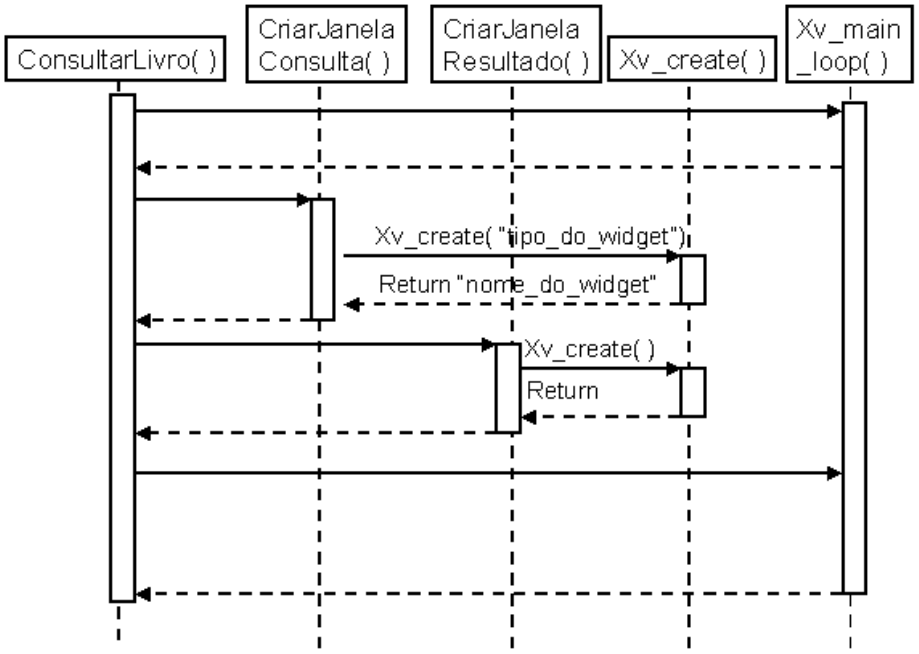
Na biblioteca XView vamos utilizar os seguintes componentes lógicos (funções):

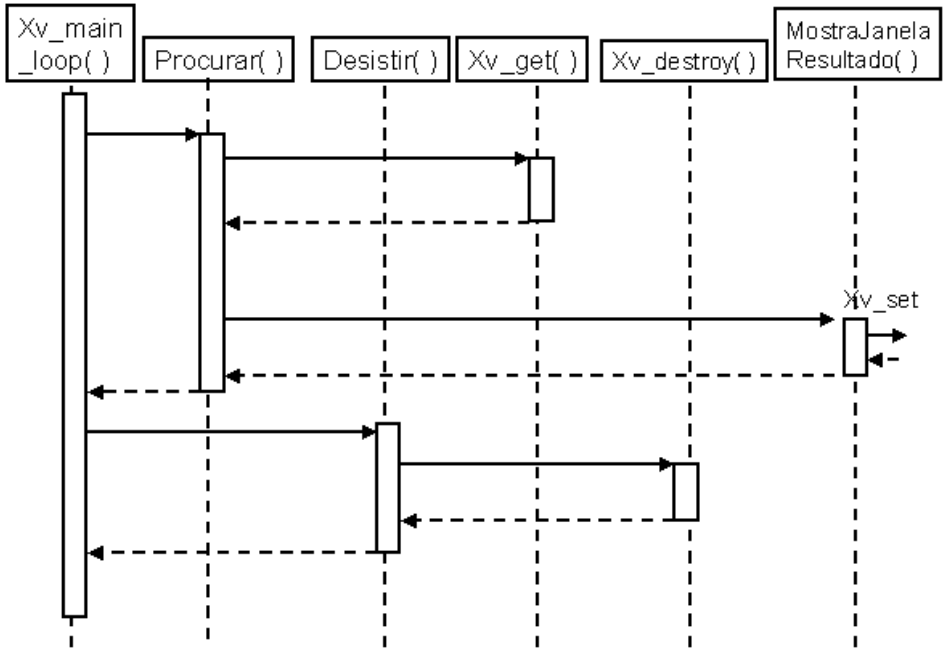
- xv_init( ) - que inicia a execução da biblioteca
- xv_create( ) - que cria os diversos widgets
- xv_get( ) - que obtem valores de atributos de widgets
- xv_set( ) - que modifica valores de atributos
- xv_main_loop( ) - que realiza o papel de *notificador*

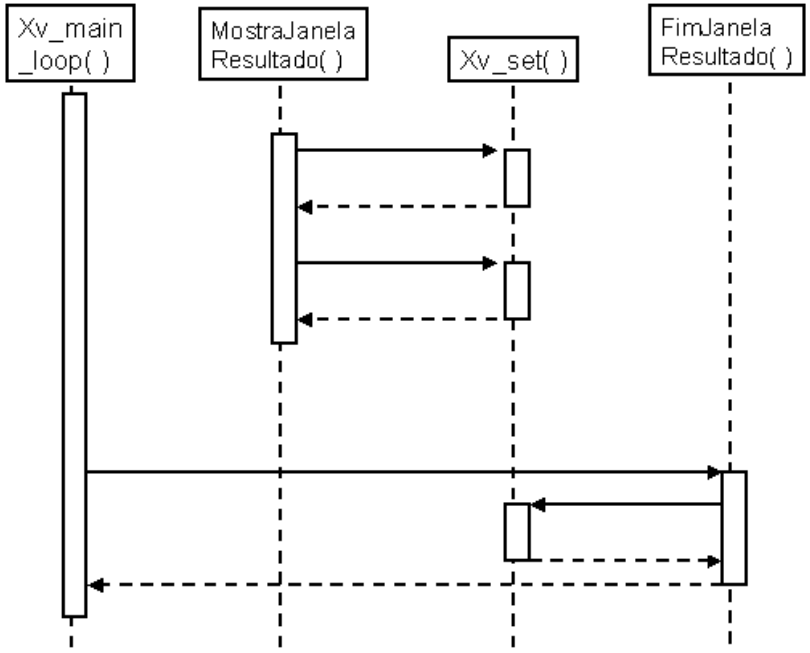
A relação de dependência entre estas funções pode ser vista na figura abaixo.



Estas funções interagem entre si de acordo com o seguinte diagrama de seqüência.

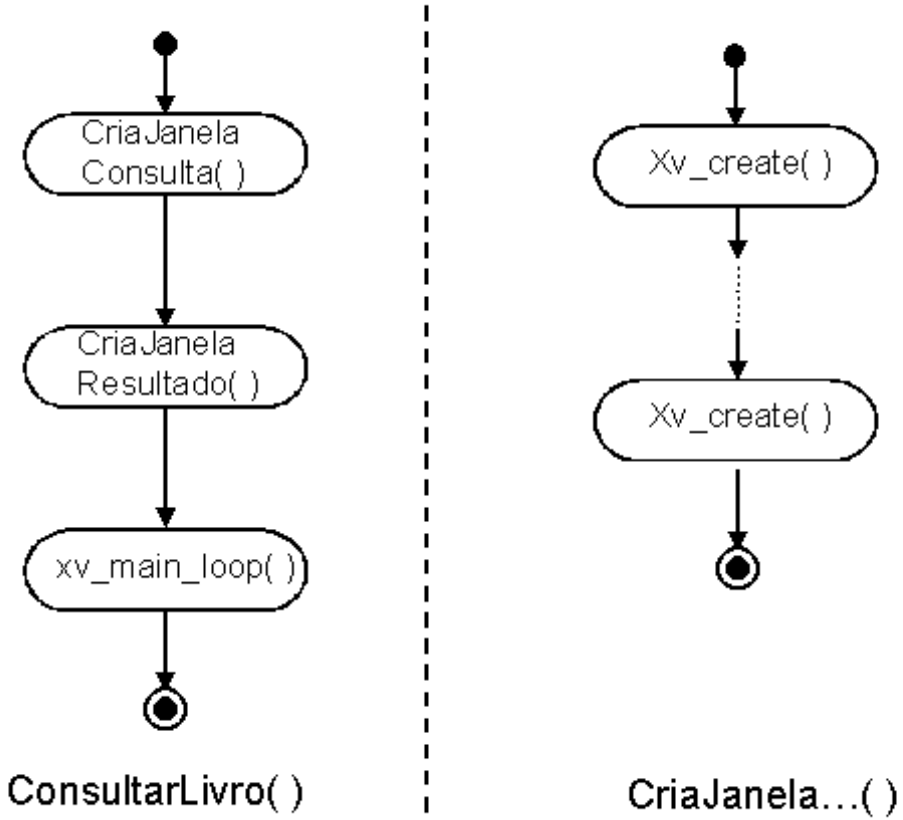




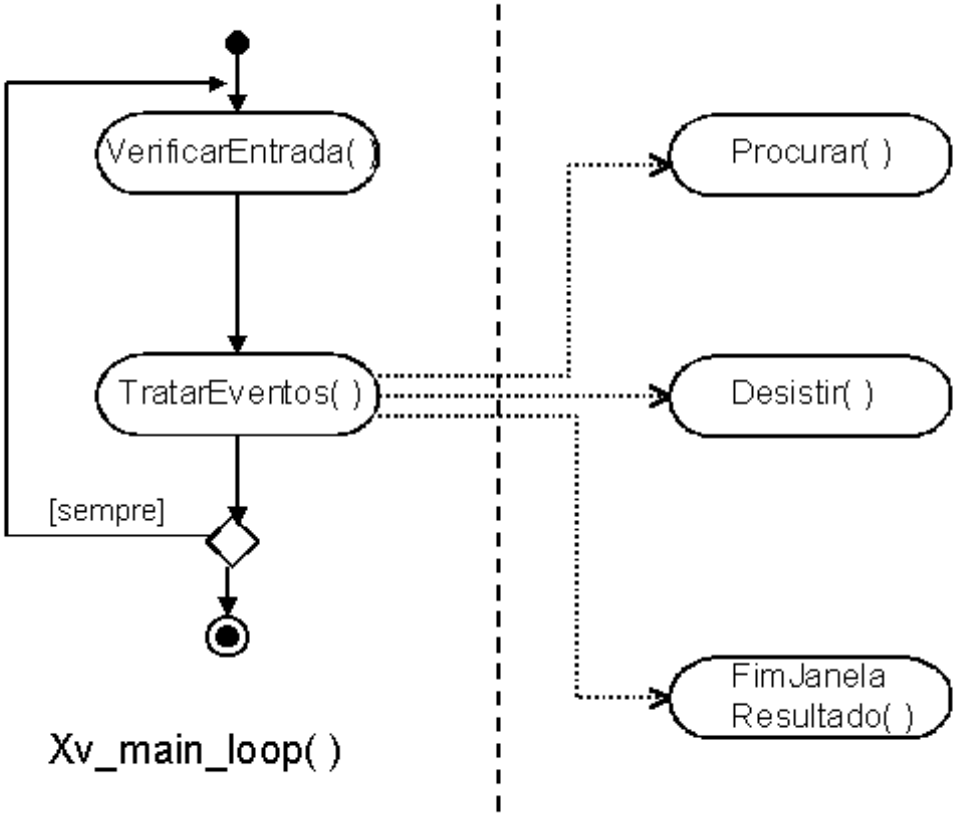


Diagramas de Atividades

Na parte esquerda da figura abaixo temos o diagrama de atividades para a função ConsultarLivro( ). As funções CriaJanelaConsulta( ) e CriaJanelaResultado( ) não estão descritas em detalhes. Entretanto, como podemos ver no algoritmo logo adiante, elas são compostas por seqüências de chamadas à função xv_create( ). Desta forma consideramos desnecessário entrar em detalhes.

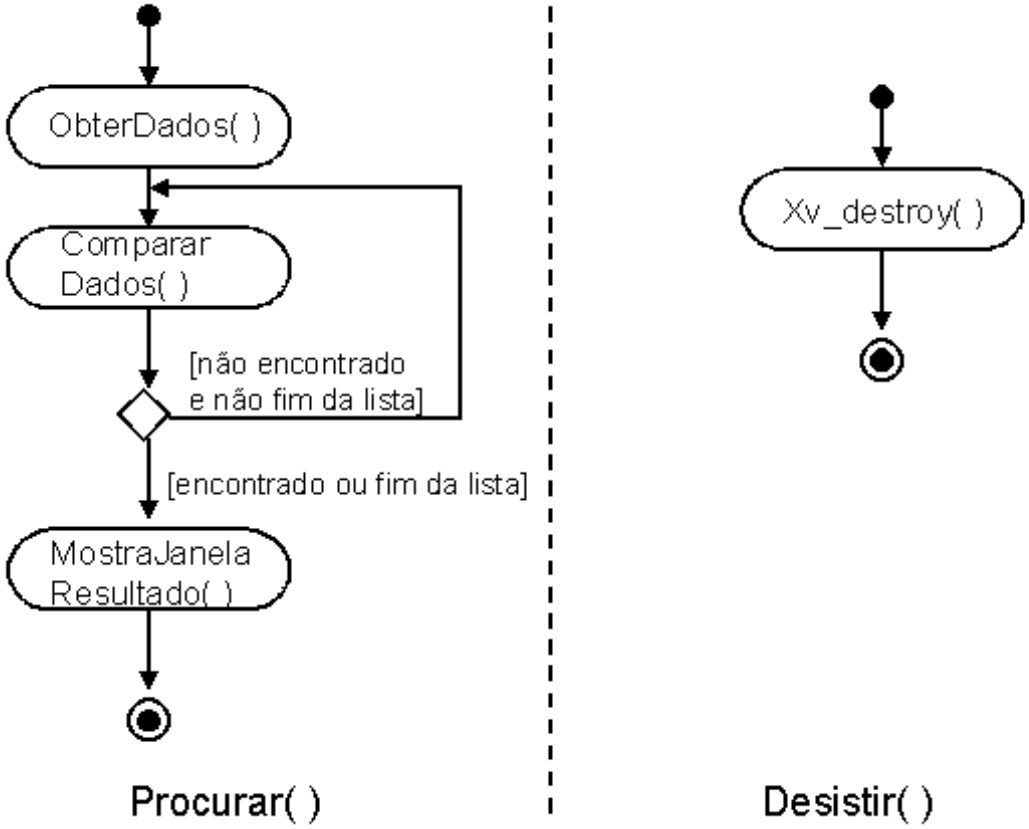


A seguir temos o diagrama de atividades para a função xv_main_loop( ). Na parte direita da figura indicamos apenas que *tratar eventos* chama as funções Procurar( ), Desistir( ) e FimJanelaResultado( ).



A seguir, temos os diagramas de atividades para funções Procurar( ) e Desistir( ). A função procurar( ) é semelhante à função de mesmo nome da solução A. Entretanto, devido à limitação do notificador (xv_main_loop( )) não poder passar parâmetros. esta função deve obter as informações fornecidas pelo usuário diretamente dos widgets caixa de texto. Isto é feito pela função ObterDados( ). Da mesma forma, para fornecer o resultado, esta função deve ativar MostraJanelaResultado( ) que se encarregara de fornecer a posição ao usuário. Esta função apenas ativa a janela que havia sido criada anteriormente. Após isto, MostraJanelaResultado( ) retorna o controle para Procurar( ) que por sua vez passa o controle de volta para o xv_main_loop( ).

A função xv_destroy( ) recebe como argumento o objeto JanelaConsulta para destruir o programa e encerrar a execução.



Algoritmos para as funções

A biblioteca XView também oferece algumas estruturas de dados para construirmos os widgets:

- Frame - moldura de uma janela.
- Panel - interior da janela onde podem ser colocados outros widgets.
- Panel_item - os widgets *botões, caixa de texto,* etc.

Estruturas de dados globais

```
struct lista_info_livro {

        char autor[30];
        char titulo[30];
        char isbn[30];
        char posicao[10];
        lista_info_livro *proximo;

}

// janelas e widgets

Frame    JanelaConsulta,  JanelaResultado;
Panel     PanelConsulta, PanelResultado;
Panel_item  text_autor, text_titulo, text_isbn, text_resultado, button_iniciar, button_desistir, button_fechar;
```

Funções

ConsultarLivro ( ) {

```
        xv_init( );
        JanelaConsulta( );
        JanelaResultado( );
        xv_main_loop( );

}
```

DesenharJanelaConsulta( ) {

```
        JanelaConsulta = xv_create(NULL,FRAME,
                                   FRAME_LABEL, "Consultar Livro",
                                   XV_WIDTH, 200,
                                   XV_HEIGHT, 100,
                                   NULL);
        PanelConsulta = xv_create(JanelaConsulta, PANEL, NULL);
```

```
        text_autor = xv_create(PanelConsulta, PANEL_TEXT,

                                        PANEL_LABEL_STRING, "Autor:",
                                        PANEL_VALUE,"",
                                        NULL);
        text_titulo = xv_create(PanelConsulta, PANEL_TEXT,
                                        PANEL_LABEL_STRING, "Título:",
                                        PANEL_VALUE,"",
                                        NULL);
        text_isbn = xv_create(PanelConsulta, PANEL_TEXT,
                                        PANEL_LABEL_STRING, "Código ISBN:",
                                        PANEL_VALUE,"",
                                        NULL);
        button_iniciar = xv_create(PanelConsulta, PANEL_BUTTON,
                                        PANEL_LABEL_STRING, "Iniciar",
                                        PANEL_NOTIFY_PROC, procurar,
                                        NULL);
        button_desistir = xv_create(PanelConsulta, PANEL_BUTTON,
                                        PANEL_LABEL_STRING, "Desistir",
                                        PANEL_NOTIFY_PROC, desistir,
                                        NULL);

}
```

DesenharJanelaResultado( ) {

```
        JanelaResultado = xv_create(JanelaConsulta ,FRAME_CMD,
                                        FRAME_LABEL, "Resultado",
                                        NULL);
        PanelResultado = xv_create(JanelaResultado, PANEL, NULL);

        text_resultado = xv_create(PanelResultado, PANEL_TEXT,

                                        PANEL_LABEL_STRING, "A posicao do livro é:",
                                        PANEL_VALUE,"",
                                        NULL);
```

```
        button_fechar = xv_create(PanelResultado, PANEL_BUTTON,
                                  PANEL_LABEL_STRING, "Fechar",
                                  PANEL_NOTIFY_PROC, FimJanelaResultado,
                                  NULL);
```

```
FimJanelaResultado( ) {

        xv_set(JanelaResultado, XV_SHOW, FALSE, NULL);

}
```

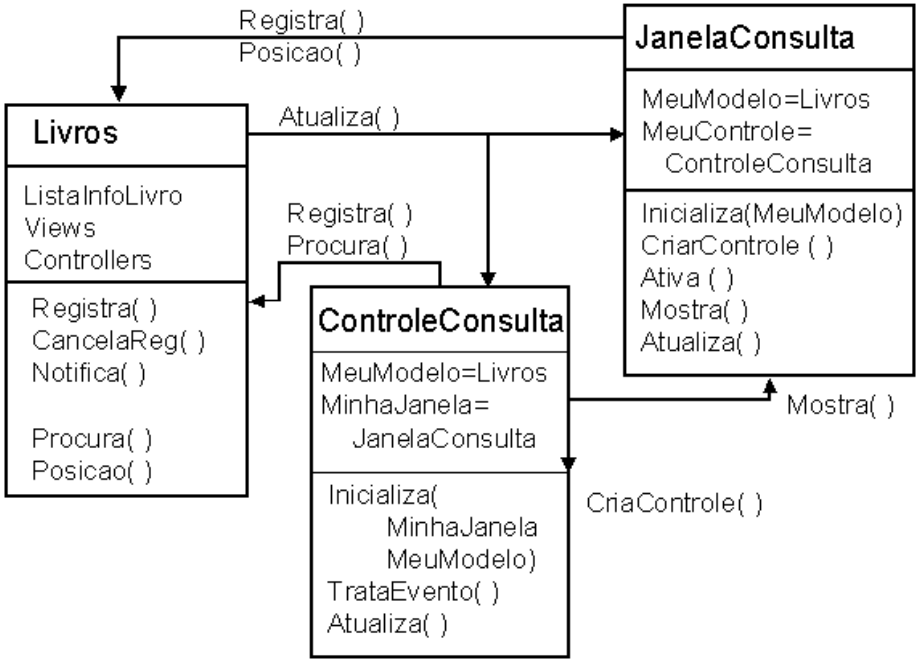
```
Procurar( ) {

        string autor,titulo,isbn,posicao;
        info_livros *lista_info_livros;

        autor = xv_get(text_autor,PANEL_VALUE);
        titulo = xv_get(text_titulo,PANEL_VALUE);
        isbn = xv_get(text_isbn,PANEL_VALUE);

        info_livros = inicio_lista;
        while (info_livros != NULL)
                if (autor = = info_livros.autor) ||
                   (titulo = = info_livros.titulo) ||
                   (isbn = = info_livros.isbn)
                then
                  posicao = info_livros.posicao
                else
                  posicao = NULL;
        }
        MostraJanelaResultado(posicao)

}
```

```
Desistir( ) {

        xv_destroy_frame(JanelaConsulta);
        exit( );

}
```

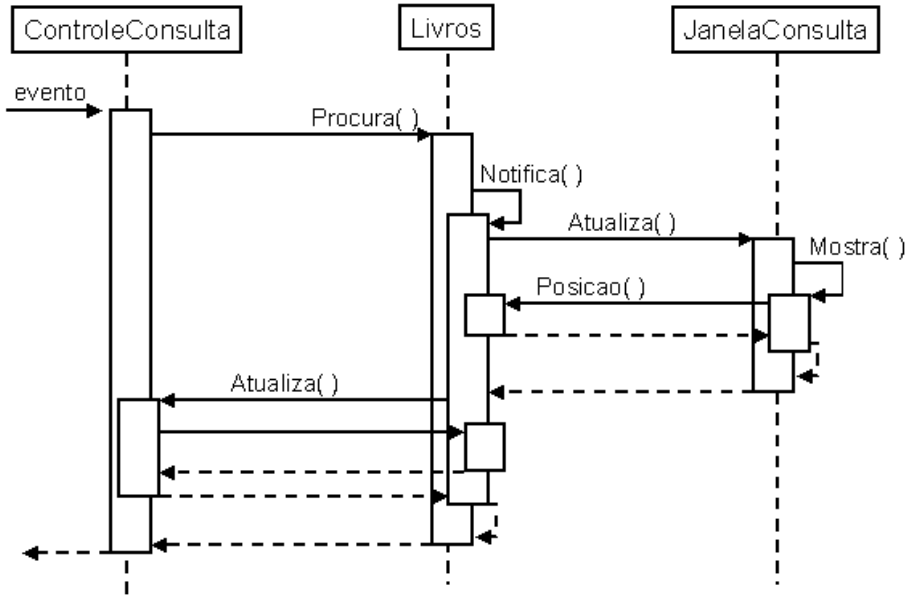
```
MostraJanelaResultado(in:posicao) {

        xv_set(text_resultado,posicao);
        xv_set(JanelaResultado, XV_SHOW, TRUE, NULL);

 }
```

7.7.3 Solução C

Usando o padrão MVC, apresentado na seção 7.6, podemos elaborar uma terceira arquitetura para a função Consultar Livro( ). Esta solução utiliza o paradigma de orientação a objetos e é descrita pelo seguinte diagrama de classes.



A interação entre os objetos destas classes é descrita no diagrama abaixo.



Anterior | Índice | Próximo

 Última atualização: 12/04/01

8 Verificação e Testes de Programas

A etapa de design de software (do modelo conceitual, da interface e da arquitetura) visa encontrar as soluções que satisfaçam os requisitos do software. No design as soluções são especificadas através de formalismos e modelos que possibilitam diferentes visões do software. Entretanto, para que o software funcione é preciso que as idéias concebidas durante do design seja codificadas em uma linguagem de programação para que possam ser traduzidas e executadas num computador.

O programa produzido durante esta etapa pode conter falhas ou erros. Os erros podem ser ocasionados por diversos fatores. Um deles pode ser a interpretação errada dos modelos produzidos durante o design. Em outros casos o programador interpretou erradamente os comandos de um programa. A complexidade de um programa, seja pela complexidade de um algoritmo ou pelo seu tamanho, pode levar ao programador a gerar um software com bastante erros de funcionamento.

Definir o que é um erro ou falha num software não é uma tarefa simples. Podemos simplificar dizendo que um erro é quando o software não se comporta conforme foi especificado. Isto pode ocorrer quando o software não realiza um função esperada, fornece resultados não esperados, não consegue terminar quando esperado, etc.

Para se avaliar o funcionamento do teste é preciso definir quais os critérios necessários para consideramos que o mesmo está correto ou não, isto é, o que pode ser considerado um erro ou falha. Além disso, é preciso determinar quais métodos ou técnicas para verificação do funcionamento serão empregadas.

Este capítulo aborda as várias formas de verificar a correção do funcionamento de um software, ou simplesmente verificação de programas. Serão vistos alguns métodos e técnicas que permitem avaliar um programa tentando verificar se o mesmo está correto ou não.

8.1 Verificação de Programas e Qualidade de Software

Vimos no capítulo 1 que vários são os fatores que determinam a qualidade de software. Cada um destes fatores precisam ser considerados para garantirmos a qualidade do software. Para isto deve-se utilizar técnicas para avaliação e análise de cada um dos fatores de qualidade do software.

Os fatores de qualidade de software podem ser classificados em do *processo* ou do *produto* e *internos* e *externos*. Os fatores de qualidade do processo visam assegurar a qualidade do desenvolvimento do produto em todas as suas etapas. Os fatores relacionados ao produto permitem analisar a qualidade do software em si.

Embora todos os fatores sejam importante para a qualidade final do software, podemos destacar alguns deles como fundamentais para que o software exista com um produto que funcione em máquinas reais e resolva problemas dos usuários.

McCall classifica os fatores de qualidade do produto em três categorias:

- fatores operacionais - aspectos relacionados com o funcionamento e utilização do software
- fatores de revisão - relacionados com a manutenção, evolução e avaliação do software
- fatores de transição - relacionados com a instalação, reutilização e interação com outros produtos

A categoria dos fatores operacionais relaciona alguns daqueles que são essenciais e indispensáveis para o funcionamento e utilização do software. Dentre os fatores que se enquadram nesta categoria podem citar:

- correção
- eficiência ou desempenho
- robustez
- integridade
- confiabilidade
- validade
- usabilidade

A usabilidade engloba outros fatores relacionados com a utilização do software pelo seus usuários. Dentre os fatores da usabilidade podemos citar a utilidade, facilidade de uso, facilidade de aprendizado, a produtividade do usuário, a flexibilidade no uso e a satisfação do usuário. A avaliação da usabilidade de um software está diretamente relacionada à sua interface de usuário. Existem métodos e técnicas especificas para avaliação da usabilidade. Podem ser realizados, por exemplo, testes com os usuário que permitam medir alguns fatores de usabilidade ou aplicados questionários que avaliem a opinião dos usuário. A avaliação da usabilidade é objeto do Design de Interfaces de Usuário e não será considerado aqui.

A validade de um software é o fator que considera que o software está de acordo com os requisitos funcionais e não-funcionais que foram determinados durante a etapa de análise e especificação de requisitos. Um software correto não é necessariamente um software válido, mas a validação requer que ele esteja correto.

A confiabilidade de um software é uma qualidade subjetiva que depende de outros fatores como a correção, eficiência, robustez e integridade. Um software passa a ser confiável para os clientes e usuários quando o mesmo esta correto, realiza as tarefas de forma eficiente, funciona bem em situações adversas e que não permite violações de acesso ou danos a seus componentes.

A correção de um software é, portanto, apenas um dentre os muitos fatores de qualidade de um software que precisam ser avaliados. Entretanto ela é fundamental para que várias outros fatores de qualidade possam ser avaliados. A correção avalia o programa que é quem faz com que o software funcione e seja útil para alguém.

8.2 Formas de verificação de programas

A correção pode ser verificada de diversas maneiras. Mas não basta apenas verificar. É preciso identificar qual a falha ou erro e corrigir. Mais ainda, é preciso garantir que o software está livre de erros ou, pelo menos, que contenha um número muito pequeno de erros que não foram detectados.

Na correção de um programa é importante diferenciar erros de **sintaxe** de erros de **semântica**. Os erros de sintaxe ocorrem quando o programa não escrito corretamente, isto é, de acordo com a gramática formal da linguagem de programação e com as palavras chaves (comandos, tipos, identificadores, etc.) escritos corretamente. Os erros de sintaxe são normalmente detectados pelo analisador sintático do compilador ou interpretador da linguagem.

Os erros de semântica ocorrem por problemas de interpretação das instruções da linguagem. Estes erros são também conhecidos como erros de lógica de programação e tem uma interferência direta no funcionamento do programa. Os erros de semântica quase sempre ocorrem por uma interpretação equivocada do programador. Ele espera que o computador execute as instruções de uma determinada maneira quando na verdade ele está realizando de outra maneira. Como a semântica de uma linguagem de programação é definida de forma rigorosa, é normalmente o programador quem causa o erro de semântica. A verificação de programas tem por objetivo detectar os erros de semântica de um programa.

Existem diversas formas de verificar se o funcionamento de um programa está correto ou não. Vamos considerar aqui três categorias de verificação:

- **inspeção de programas**
- **testes de funcionamento**
- **prova formal**

A **inspeção** é uma forma analítica de verificação da correção na qual o código fonte do programa é analisado diretamente. Ela pode ser estática ou dinâmica. A **inspeção estática** é feita mentalmente, num processo no qual o inspetor (que pode ser o próprio programador) percorre o código executando mentalmente cada uma das instruções. Este processo é denominado de **rastreamento** do código fonte.

A **inspeção dinâmica** é realizada com o auxílio de uma ferramenta de depuração (***debugger***). Estas ferramentas permitem a execução passo-a-passo de trechos do programa e a visualização de variáveis do programa.

Normalmente estas técnicas de inspeção são aplicadas para corrigir um erro já detectado em um programa, pela aplicação de testes de funcionamento.

Os **testes de funcionamento** têm por objetivo a detectar o maior número possível de erros. Deve-se executar o programa de maneira a forçar certos limites para os erros possam ser encontrados. Os testes não podem garantir que não existam erros num software, exceto em casos muito de programas muito simples. Em programas que possuam estrutura de controle como a repetição o número de caminhos possíveis de execução aumenta bastante de maneira a tornar inviável o teste exaustivo de todas as situações. É necessário definir os casos de teste de maneira que seja possível que os caminhos possíveis possam ser testados.

Os teste de funcionamento podem ser divididos em **testes em ponto pequeno** ou **teste de unidades** e em **teste em ponto grande** ou **teste de integração**. Os teste de unidades visam verificar o funcionamento dos componentes, módulos ou unidades funcionais isoladamente. Os testes de integração verificam se os componentes funcionam corretamente após a integração com os outros componentes.

A **prova de programa** é a forma mais precisa de verificar se um programa está correto ou não. Ela pode garantir que um programa está livre de erros. Isto é possível por um programa é um objeto formal, isto é, definido por uma linguagem de programação formal de base matemática. Podemos explicar esta idéia de forma simples.

Cada sentença de um programa é uma instrução para o computador realizar uma certa atividade. Este comportamento é o significado ou semântica da sentença do programa. para que o comportamento do computador seja sempre o mesmo para cada instrução, a semântica dos elementos de uma linguagem de programação deve ser definida formalmente. Desta forma se espera que cada elemento de um programa provoque uma transformação precisa em outros elementos. Analogamente, a combinação das instruções deve possibilitar mudanças também previsíveis no programa. Assim, espera-se todo um programa altere sempre o estado de outros elementos do programa da mesma maneira. Pode-se então provar que para uma dada condição inicial chega-se a uma condição final se as instruções forem executadas.

Exemplificando, vamos considerar inicialmente uma instrução de um programa Pascal

```
{x=5}
x := x + 1;
{x=6}
```

A expressão {x=5} indica a condição da variável antes da execução da instrução, enquanto que {x=6} mostra a situação após a execução da instrução. De acordo com a semântica da instrução pode-se garantir a condição final da variável é sempre a inicial incrementada de 1. Isto pode parecer óbvio numa única e simples instrução, mas torna-se um recurso bastante valioso quando temos um conjunto complexo de instruções. Por outro lado, a complexidade e os custos da aplicação de métodos de prova formal em programas muito grande é enorme.

8.3 Uma introdução à prova de programas

Para aplicarmos os métodos de prova é necessário recorrer às notações matemáticas baseadas em lógica. O cálculo de predicados, por exemplo, permite expressar as condições de programa e as regras de prova com base na semântica das instruções de uma linguagem. Linguagens de especificações formais com Z, VDM ou B, são também bastante utilizadas para prova de programas. Aliás, uma das grandes vantagens da aplicações de métodos formais em engenharia de software é que se pode provar matematicamente que um programa está correto em relação à sua especificação formal. Algumas ferramentas possibilitam a realização de provas automáticas, diminuindo a complexidade e os custos do método.

De uma forma geral a prova de programa requer **condições** que devem ser expressas numa notação matemática e **regras de prova** que podem ser aplicadas de acordo com as instruções de um programas. Expressando as instruções por S (statements) e as condições por {C} podemos dizer que:

```
{C1} S1 {C2}  e   {C2} S2 {C3}
```

Que indica a instrução S1 modifica o programa de uma condição C1 para uma condição C2 e as instrução S2 de C2 para C3.

Quando queremos mostrar que um programa ou trecho de programa está correto indicamos qual a condição final (ou pós-condição) que deve ser atingida para uma certa condição inicial (pré-condição)

```
{Pré} programa {Pós}
```

No caso de um programa que realiza a divisão de dois números positivos a expressão de prova seria:

```
{y>0 & x>0} programa-divisão {y = x*q + r}
```

onde q é o quociente e r o resto. Mais adiante mostraremos

Uma regra de prova é expressa por uma notação do tipo:

$$\frac{E1,\ E2}{E3}$$

Indica que se E1 e E2 tenham sido provados, então deduzimos que E3 também é verdade. Por exemplo, a regra que vale para uma seqüência de instruções é a seguinte:

$$\frac{\{C1\}\ S1\ \{C2\},\ \{C2\}\ S2\ \{C3\}}{\{C1\}\ S1;S2\ \{C3\}}$$

Da mesma forma teríamos regras para as outras principais estruturas de controle como repetição e seleção. Vejamos o caso de uma repetição. Seja o trecho de um programa e as condições iniciais de sua variável:

```
{x >= 0}
while (x>0) do
        x:= x - 1;
end while
{x = 0}
```

Aplicando as regras de prova da repetição e da atribuição podemos provar que o programa está correto, isto é que para a condição inicial {x>=0} teremos a condição final {x=0}. A regra da repetição pode ser descrita como

$$\frac{\{Invariante\ \&\ ExprCond\}\ S\ \{Invariante\}}{\{Invariante\}\ \textbf{while}\ (ExprCond)\ S\ \{Invariante\ \&\ not\ ExprCond\}}$$

Onde ExprCond é a expressão condicional do while e Invariante é uma expressão que deve ser verdade em todo o loop do while, e também antes e depois da sua execução. A idéia básica é que se o corpo S do while não torna o invariante falso -- {Invariante & ExprCond} S {Invariante} -- caso a ExprCond seja verdadeira, então deduz-se que a expressão inferior é verdadeira. Aplicando ao programa acima temos que o invariante é {x>=0} e a regra fica:

$$\frac{\{x>=0 \ \&\ x>0\}\ \mathbf{x := x - 1};\ \{x>=0\}}{\{x>=0\}\mathbf{while}\ (x>0)\ \mathbf{x := x - 1}\ \{x>=0\ \&\ \text{not } x>0\}}$$

Podemos verificar facilmente que a expressão de cima é verdadeira. Com base na regra podemos deduzir que a expressão inferior é verdadeira. Como {x>=0 & not x>0} é o mesmo que {x>=0 & x<=0}, temos que {x=0} o que está de acordo como o nosso programa.

Usando o mesmo método podemos provar que o programa abaixo realiza corretamente uma divisão de y por x:

```
{y>0 & x>0}
r=y;
q=0;
while (r >= x) begin
        q := q + 1;
        r := r - x;
end;
{y = x*q + r & x>r>=0}
```

O invariante deste programa é y = x*q + r, que é a expressão que indica uma divisão e que gostaríamos que fosse válida ao final. A prova fica como exercício.

8.4 Testes de correção

O teste é a maneira mais comum de fazermos a verificação da correção de um programa. Os testes consistem na execução controlada do programa com o objetivo de analisar o seu comportamento para verificar se ele comporta-se como esperado. A limitação de um teste está em ele poder identificar que existem erros, mas não poder garantir que erros não existem.

Vimos que podemos diferenciar os testes em testes de ponto pequeno e teste de ponto grande. Os **teste de ponto pequeno** ou de unidades possibilitam verificar trecho isolados de um programa, normalmente um componentes ou módulo. Os principais tipos de teste de ponto pequeno são os **testes da caixa branca** e os **testes da caixa preta**.

Os **testes em ponto grande** tem por objetivo verificar a correção de conjuntos de componentes interconectados. Eles são aplicados a componentes como um todo.

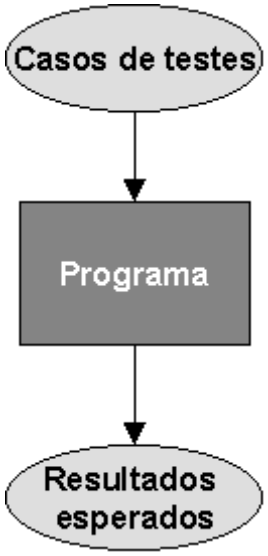
8.4.1 Fundamentos

A idéia fundamental por trás de um teste pode ser explicada de forma simples. Seja P um programa e D e I o domínio e a imagem, respectivamente. O domínio refere-se aos valores iniciais ou de entrada de D e I os valores resultantes ao final da execução de P. Por simplicidade, vamos considerar P que se comporta com uma função com domínio D e I. Obviamente, em casos reais, D e I podem conter seqüências de dados e P pode ter várias funções de entrada/saída.

Seja R os valores resultantes desejáveis determinados durante a especificação. Seja d dados do domínio D, dizemos que P(d) refere-se aos possíveis resultados de P para os dados iniciais d. Dizemos que um programa está **correto** se P(d) satisfaz R. P está correto se e somente se para todo dado d, P(d) satisfaz R. Dizemos que um programa **não** está correto se o programa não termina de executar, ou se termina mas P(d) não satisfaz R.

A presença de um erro é demonstrada pela existência de dados d, tal que P(d) não satisfaz R. Muitas vezes é impossível verificar P(d) para todos os dados d do domínio D. Nestas situações é importante elaborar **casos de testes**.

Um caso de teste é um elemento d de D. Um conjunto de casos de teste T é um subconjunto de D. Um conjunto de teste T é dito ideal se, sempre que P for incorreto, existe um d pertencente a T que faz P(d) não satisfazer a R.



A escolha dos casos de teste é a etapa mais importante durante a realização de um teste. Como o objetivo é encontrar erros, deve-se escolher os casos de teste que force a detecção deles. Esta escolha depende do tipo de teste que podemos aplicar.

Vejamos um exemplo de como a escolha de casos de testes é deve ser criteriosa. Suponha o programa que verifica o valor máximo de dois números.

```
read(x); read(y);
if (x>y) then
        max := x;
else
        max := x;
end if;
write(max);
```

Se escolhermos como caso de teste os valores {x=3, y=2; x=4, y=3; x=5, y=1; x=10, y=5} nenhum deles revelará a existência de erro. Entretanto, com apenas dois casos de testes escolhidos com base na lógica do programa {x=3, y=2; x=2, y=3}, que explora o fato de que um teste comparativo é feito, revela o erro no segundo caso de teste. Ter muitos casos de testes não é garantia de testes bem-sucedidos. Mais importante é escolher casos de testes que ajudem a identificar erros.

Resumindo, as regras gerais para se realizar um teste:

1. Executar um programa com a intenção de descobrir um erro
2. Um bom caso de teste é aquele que tem uma elevada probabilidade de revelar um erro ainda não descoberto.
3. Um teste bem-sucedido é aquele que revela um erro ainda não descoberto.

8.4.2 Testes da caixa branca

Os teste da caixa branca estão baseados nos elementos internos de um trecho de programa.O objetivo é encontrar o menor número de casos de teste que permita que todos os comandos de um programa seja executado pelo menos uma vez. Os casos de teste são determinados a partir das estruturas de controle do programa. A idéia é escolher os casos de teste de forma a forçar que todos os caminho possíveis do fluxo de controle do programa sejam percorridos durante os testes. O teste do caminho básico, o teste de loops (laços) e o teste de estruturas de controles são tipos de teste da caixa branca.

No teste do caminho básico, o passo inicial é determinar todos os caminhos possíveis. Para isto costuma-se elaborar um **grafo de fluxo de controle** do programa. O gráfico permite identificar os caminhos possíveis para que se possa elaborar os casos de uso. Como cada caminho é definido pelas expressões condicionais das estruturas de controle, deve-se determinar os casos de teste escolhendo valores de variáveis para os casos nos quais cada uma das expressões sejam verdadeiras ou não.

Vejamos um exemplo para um programa que calcula o máximo divisor comum (MDC) de dois inteiros.

```
read(x); read(y);
while x≠y do
        if (x>y) then
                x := x - y;
        else
                y := y - x;
        end if;
end while;
write("MDC=",x);
```

Observe que a estrutura de controle while possui dois caminhos possíveis. A condição do while sugere dois possíveis casos de teste. Um deles é para os valores de x=y e o outro é para valores de x≠y. No primeiro caso testa-se o programa para que o while não seja executado. No segundo, testa-se para que ele seja executado pelo menos uma vez.

No corpo do while temos uma estrutura de controle de seleção (if ... then ... else). Esta estrutura possibilita dois caminhos possíveis. A expressão condicional desta estrutura sugere mais dois casos de teste para os dois caminhos. É preciso escolher pelo menos um caso de teste no qual tenhamos x>y e outro no qual tenhamos x≤y. Combinando todas as possibilidades podemos definir quatro casos de teste.

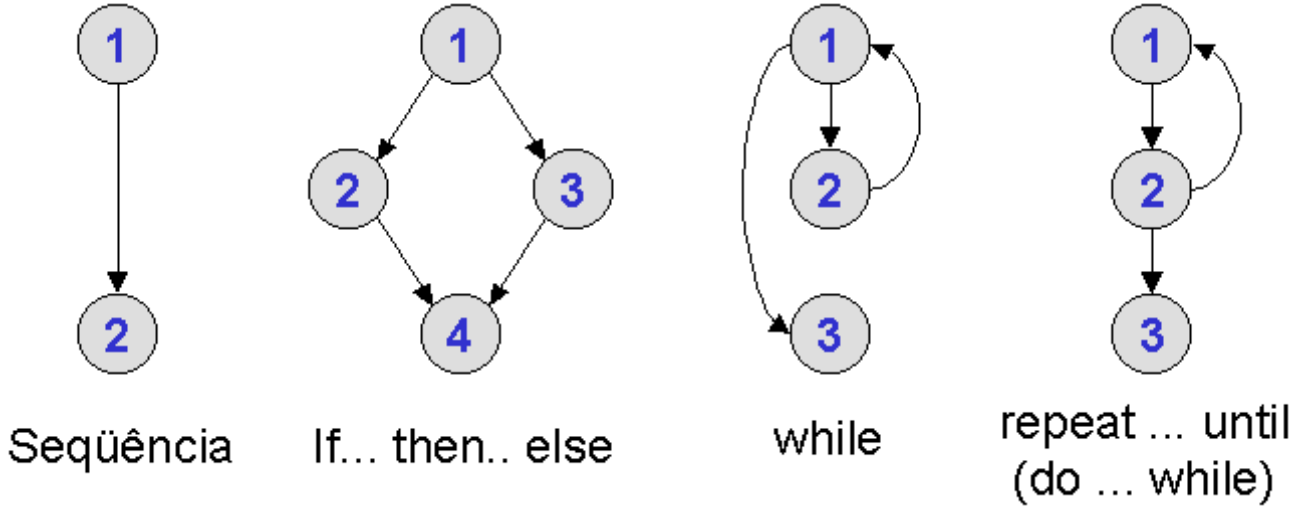
1. x=y e x>y
2. x=y e x≤y
3. x≠y e x>y, que pode ser reduzido x>y
4. x≠y e x≤y, que pode ser reduzido x<y

Que podem ser reduzidos a três casos:

1. x=y
2. x>y
3. x<y

Com isto cobrimos todos os caminhos possíveis.

Para casos de programas mais complexos podemos usar o **grafo de fluxo de controle** para determinarmos a o número de caminhos possíveis.



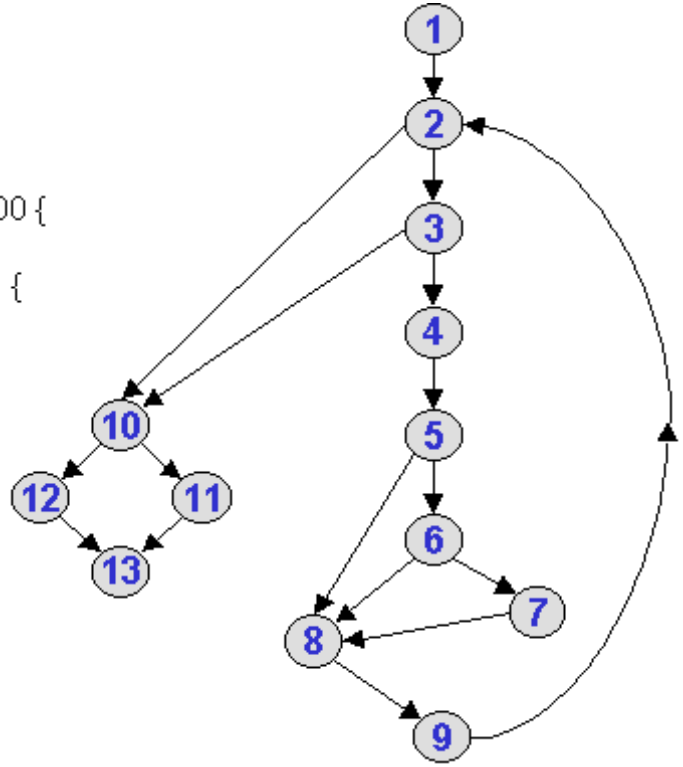
Seja um trecho de programa que calcula a média aritmética de 100 números ou menos que se situem entre valores-limite mínimo e máximo. Ele também computa a soma e o número total válido.

```
i = 1;
total.entrada = total.valido = 0;
soma= 0;
while valor[i] != -999 && total.entrada < 100 {
        total.entrada++;
        if valor[i] ≥ minimo && valor[i] ≤ maximo {
                total.valido++;
                soma = soma + valor[i];
        }
        i++;
}
if total.valido > 0
        media = soma/total.valido;
else
        media = -999;
```

O grafo de fluxo é derivado da seguinte maneira. Cada comando simples é representado por um nó. Cada estrutura de controle é representada por um grafo como mostra a figura a seguir. Para as estruturas de controle que possuem expressões condicionais, cada expressão é representada por um nó do grafo. Caso a expressão condicional seja composta por várias expressões condicionais, um nó para cada expressão deve ser representado no grafo.

Para o programa da média temos, portanto, o seguinte grafo de fluxo.

```
1  i = 1;
   total_entrada = total_valido = 0;
   soma= 0;
2  while valor[i] != -999 && total_entrada < 100 {
3     total_entrada++;
4     if valor[i] ? minimo && valor[i] ? maximo {
5        total.valido++;
6        soma = soma + valor[i];
7     }
8     i++;
   }
9  if total_valido > 0
10    media = soma/total_valido;
   else
11    media = -999;
12 }
```



O número de caminhos possíveis pode ser determinado a partir do grafo de fluxo de várias maneiras:

- Pelo número de regiões do grafo
- Pela fórmula E - N + 2, onde E é o número de elos e N o número de nós (N)

No exemplo temos que o número de caminhos possíveis é 4.

- O grafo tem 6 regiões
- O grafo tem 17 elos e 13 nós, portanto, 17 - 13 + 2 = 6.

Os caminhos possíveis são:

- Caminho 1: 1-2-10-12-13
- Caminho 2: 1-2-10-11-13
- Caminho 3: 1-2-3-10-11-13...
- Caminho 4: 1-2-3-4-5-8-9-2...

- Caminho 5: 1-2-3-4-5-6-8-9-2...
- Caminho 6: 1-2-3-4-5-6-7-8-9-2

As reticências depois dos caminhos 4 a 6 indicam que qualquer caminho ao longo do restante já foi coberto pelos caminhos possíveis.

Os casos de teste devem ser escolhidos com base nas expressões condicionais que forcem o programa a percorrer cada um dos caminhos possíveis.

- Casos de teste do caminho 1:
  - valor[1] = -999
  - Resultado esperado: média = -999 e outros totais com os valores iniciais
- Casos de teste do caminho 2:
  - Para testar este caminho é preciso forçar que a média seja calculada
  - Utilizar:
    - $0 < k1, ..., kn < i \leq 100$
    - $minimo \leq valor[k1],..., valor[kn] \leq maximo$
    - valor[i] = -999
  - Resultado esperado: valor da média calculado corretamente baseado nos k valores e totais com valores apropriados
- Casos de teste do caminho 3:
  - Utilizar:
    - $0 < k1, ..., kn \leq 100 < i$ e
    - $minimo \leq valor[k1],..., valor[kn] \leq maximo$ e
    - valor[i] = -999
  - Resultado esperado: valor da média calculado corretamente baseado nos 100 primeiros valores e total.valido=n e total.entrada=n
- Casos de teste do caminho 4:
  - Utilizar:
    - $0 < k1, ..., kn < i \leq 100$ e
    - $valor[k1],...,valor[kj] < minimo$ e
    - $minimo \leq valor[kj],..., valor[kn] \leq maximo$ e
    - valor[i] = -999
  - Resultados esperados: valor da média calculado para os valores de kj a kn e total.valido=j e total.entrada=n
- Casos de teste do caminho 5:
  - Utilizar:
    - $0 < k1, ..., kn < i \leq 100$ e
    - $minimo \leq valor[k1],..., valor[kj] \leq maximo$ e
    - $valor[kj],..., valor[kn] > maximo$ e
    - valor[i] = -999
  - Resultados esperados: valor da média calculado para os valores de k1 a kj e total.valido=n-j e total.entrada=n
- Casos de teste do caminho 6:

- Utilizar:
  - $0 < k1, ..., kn < i \leq 100$ e
  - $maximo \leq valor[k1],..., valor[kn]$ ou $valor[k1],..., valor[kn] \leq minimo$ e
  - $valor[i] = -999$
- Resultados esperados: valor da média = -999 e total.valido=0 e total.entrada=n

8.4.3 Testes da caixa preta

Os testes da caixa preta são chamados de *testes funcionais* por estarem baseados nos requisitos funcionais do software. Espera-se poder verificar aquilo que se pretende que o programa faça. Exemplos de técnicas de teste da caixa são os *grafos de causa-efeito*, *testes baseados em tabela de decisão* e *teste do valor limite.*

O teste da caixa preta não é uma alternativa ao teste da caixa branca, mas uma abordagem complementar.Ele permite verificar a validade do software em relação aos requisitos. Por exemplo, suponha que o cliente solicitou que o software calculasse o valor médio das três notas dos alunos. O desenvolvedor construiu uma função que calcula a média aritmética. Os teste da caixa-preta devem permitir verificar que a função implementada não é a função correta.

De acordo com os requisitos, a média deve ser calculada pela fórmula **media = (a*4 + b*5 + c*6)/15**. Desta forma, para o caso de teste cujos valores sejam {a = 2, b=3, c=4}, o valor esperado é 3,1333, e para o caso {a = 5, b=6, c=8}, o valor esperado é 6,5333.

Aplicando alguma técnica de teste da caixa preta chega-se ao valor media = 3, para o primeiro caso de teste e media = 6,333 uma vez que o desenvolvedor implementou uma função que calcula a média aritmética e não a média ponderada.

Uma das técnicas utilizadas para o teste da caixa-preta utiliza **grafos de causa-efeito**. Esta técnica oferece um representação concisa das condições lógicas e das ações correspondentes. A técnica segue 4 passos:

1. Causas (condições de entrada) e efeitos (ações) são relacionados para um módulo e um identificador é atribuído a cada um.
2. Um grafo de causa-efeito (descrito a seguir) é desenvolvido.
3. O grafo é convertido numa tabela de decisão.
4. As regras da tabela são convertidas em casos de teste.

Considere como exemplo um programa que realiza a cobrança de chamadas telefônicas. Os valores de cada chamadas são contabilizados de acordo com a duração, local de destino (para onde for feita a chamada) e faixa de horário.

- Se o local de destino for o mesmo da origem (chamada local) e a faixa de horário for das 6:00 às 23:59 então valor do minuto é R$ 1,00.
- Se chamada local e a faixa for 0:00 às 5:59 o valor do minuto de cada chamada é R0,50.
- Se o local for um outro estado no país (chamada interurbana) e a faixa de horário for das 9:00 às 21:00 então o valor do minuto é calculado de acordo com o valor básico por estado.
- Se chamada interurbana e faixa de horário for das 21:00 às 9:00 o valor do minuto é fixo, sendo R$1,00.

- Se chamada internacional o valor não depende de faixa de horário e é calculado de acordo com o valor básico por país.

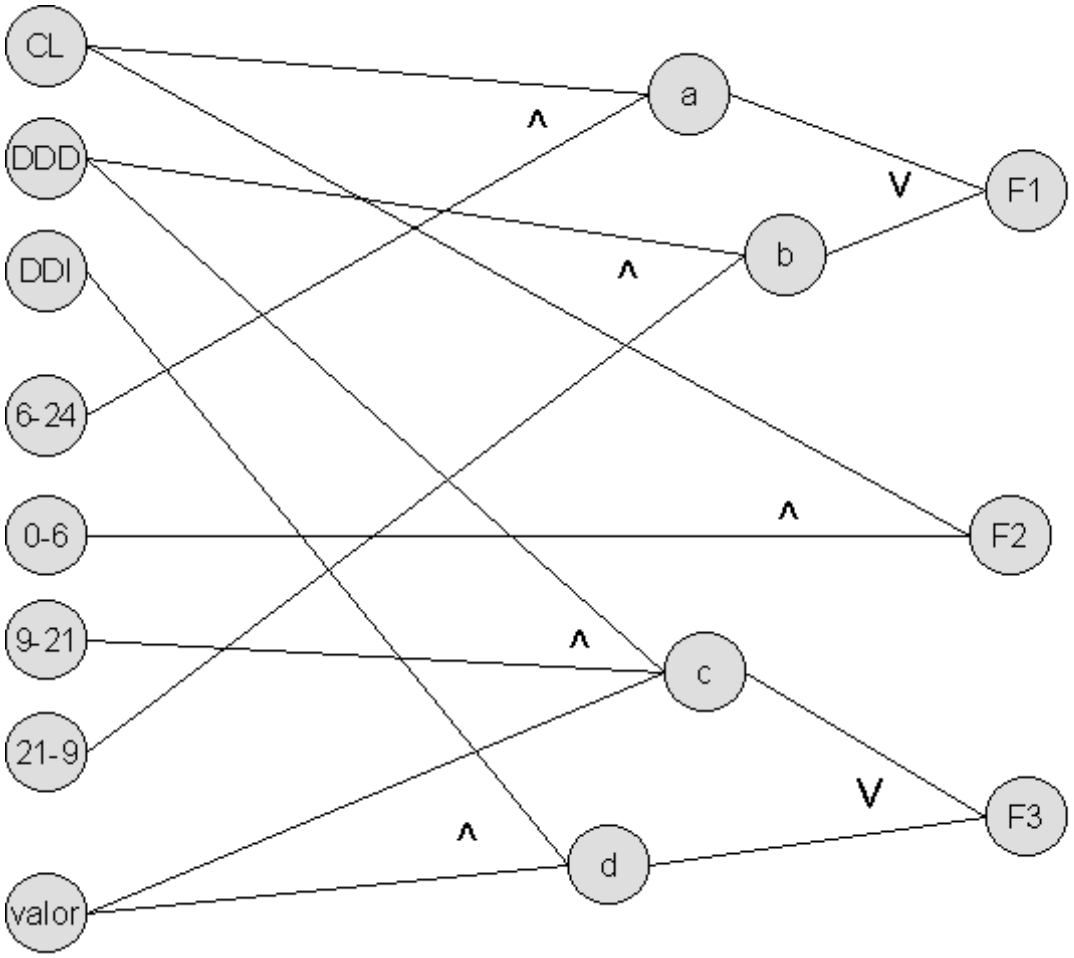
Para o programa cobrança temos então as causas

- tipo da chamada: local (CL), interurbana (DDD) e internacional (DDI)
- faixa de horário: 6-24, 0-6, 9-21, 21-9
- estado ou país: AC,AM, AP, ... (estados do Brasil), EUA, RU, JP, ...

Os efeitos esperados são os cálculos de cobrança conforme especificado

- F1= duração*R$1,00
- F2= R$0,50
- F3= duração*valor_localidade

O grafo de causa-efeito para a especificação acima é o seguinte:



Com base no grafo de causa-efeito a seguinte tabela de decisão pode ser elaborada.

CL	X		X		
DDD		X		X	
DDI					X
6-24	X				
0-6			X		
9-21				X	

21-9		X			
valor_localidade				X	X
	F1	F1	F2	F3	F3

Com base no grafo ou na tabela acima podemos derivar cinco casos de teste nos quais podemos verificar se para as entradas correspondentes (causas) o programa realiza os cálculos correspondentes (efeitos).

- Caso1: {tipo_da_chamada=CL, faixa_de_horário=6-24, valor_localidade=X}
- Caso2: {tipo_da_chamada=DDD, faixa_de_horário=21-9, valor_localidade=X}
- Caso3: {tipo_da_chamada=CL, faixa_de_horário=0-6, valor_localidade=X}
- Caso4: {tipo_da_chamada=DDD, faixa_de_horário=9-21, valor_localidade=X}
- Caso5: {tipo_da_chamada=DDI, faixa_de_horário=Y, valor_localidade=X}

Um outro tipo de técnica para os teste da caixa-preta pode ser utilizada para completar a técnica de causa-efeito. A técnica de **análise do valor limite** visa estabelecer casos de testes nos quais os valores de limites extremos serão avaliados.

Para o exemplo anterior poderíamos criar casos que testassem valores de faixa de horário próximos ao limite. Por exemplo, 5:59, 6:00, 6:01, 23:59, 0:00, 0:01, e assim da mesma forma para as outras faixas de horário.

8.4.4 Testes em ponto grande

 Última atualização: 12/04/01