Engenharia de Software

Sumário

Engenharia de Software

Capítulo 1

Engenharia de Software — Conceitos Básicos

1. Introdução

Nos anos 40, quando se iniciou a evolução dos sistemas computadorizados, grande parte dos esforços, e conseqüentes custos, era concentrada no desenvolvimento do hardware, em razão, principalmente das limitações e dificuldades encontradas na época. À medida que a tecnologia de hardware foi sendo dominada, as preocupações se voltaram, no início dos anos 50, para o desenvolvimento dos sistemas operacionais, onde surgiram então as primeiras realizações destes sistemas, assim como das chamadas linguagens de programação de alto nível, como FORTRAN e COBOL, e dos respectivos compiladores. A tendência da época foi de poupar cada vez mais o usuário de um computador de conhecer profundamente as questões relacionadas ao funcionamento interno da máquina, permitindo que este pudesse concentrar seus esforços na resolução dos problemas computacionais em lugar de preocupar-se com os problemas relacionados ao funcionamento do hardware.

Já no início dos anos 60, com o surgimento dos sistemas operacionais com características de multiprogramação, a eficiência e utilidade dos sistemas computacionais tiveram um considerável crescimento, para o que contribuíram também, de forma bastante significativa, as constantes quedas de preço do hardware.

Uma conseqüência deste crescimento foi a necessidade, cada vez maior, de desenvolver grandes sistemas de software em substituição aos pequenos programas aplicativos utilizados até então. Desta necessidade, surgiu um problema nada trivial devido à falta de experiência e à não adequação dos métodos de desenvolvimento existentes para pequenos programas, o que foi caracterizado, ainda na década de 60 como a "crise do software", mas que, por outro lado, permitiu o nascimento do termo "Engenharia de Software".

Atualmente, apesar da constante queda dos preços dos equipamentos, o custo de desenvolvimento de software não obedece a esta mesma tendência. Pelo contrário, corresponde a uma percentagem cada vez maior no custo global de um sistema informatizado. A principal razão para isto é que a tecnologia de desenvolvimento de software implica, ainda, em grande carga de trabalho, os projetos de grandes sistemas de software envolvendo em regra geral um grande número de pessoas num prazo relativamente longo de desenvolvimento. O desenvolvimento destes sistemas é realizado, na maior parte das vezes, de forma "ad-hoc", conduzindo a freqüentes desrespeitos de cronogramas e acréscimos de custos de desenvolvimento.

O objetivo deste curso é apresentar propostas de soluções às questões relacionadas ao desenvolvimento de software, através da transferência dos principais conceitos relativos à Engenharia de Software, particularmente no que diz respeito ao uso de técnicas, metodologias e ferramentas de desenvolvimento de software.

2. PRINCIPAIS ASPECTOS DO SOFTWARE

2.1. Software: definição e características

Pode-se definir o **software**, numa forma clássica, como sendo: "um **conjunto de instruções** que, quando executadas, produzem a função e o desempenho desejados, **estruturas de dados** que permitam que as informações relativas ao problema a resolver sejam manipuladas adequadamente e a **documentação** necessária para um melhor entendimento da sua operação e uso".

Entretanto, no contexto da Engenharia de Software, o software deve ser visto como um **produto** a ser "vendido". É importante dar esta ênfase, diferenciando os "programas" que são concebidos num contexto mais restrito, onde o usuário ou "cliente" é o próprio autor. No caso destes programas, a documentação associada é pequena ou (na maior parte das vezes) inexistente e a preocupação com a existência de erros de execução não é um fator maior, considerando que o principal usuário é o próprio autor do programa, este não terá dificuldades, em princípio, na detecção e correção de um eventual "bug". Além do aspecto da correção, outras boas características não são também objeto de preocupação como a portabilidade, a flexibilidade e a possibilidade de reutilização.

Um **produto de software** (ou software, como vamos chamar ao longo do curso), por outro lado, é sistematicamente destinado ao uso por pessoas outras que os seus programadores. Os eventuais usuários podem, ainda, ter formações e experiências diferentes, o que significa que uma grande preocupação no que diz respeito ao desenvolvimento do produto deve ser a sua interface, reforçada com uma documentação rica em informações para que todos os recursos oferecidos possam ser explorados de forma eficiente. Ainda, os produtos de software devem passar normalmente por uma exaustiva bateria de testes, dado que os usuários não estarão interessados (e nem terão capacidade) de detectar e corrigir os eventuais erros de execução.

Resumindo, um programa desenvolvido para resolver um dado problema e um produto de software destinado à resolução do mesmo problema são duas coisas totalmente diferentes. É óbvio que o esforço e o conseqüente custo associado ao desenvolvimento de um produto serão muito superiores.

Dentro desta ótica, é importante, para melhor caracterizar o significado de software, é importante levantar algumas particularidades do software, quando comparadas a outros produtos, considerando que o software é um elemento lógico e não físico como a maioria dos produtos:

- ***o software é concebido e desenvolvido como resultado de um trabalho de engenharia*** e não manufaturado no sentido clássico;
- ***o software não se desgasta***, ou seja, ao contrário da maioria dos produtos, o software não se caracteriza por um aumento na possibilidade de falhas à medida que o tempo passa (como acontece com a maioria dos produtos manufaturados);
- ***a maioria dos produtos de software é concebida inteiramente sob medida***, sem a utilização de componentes pré-existentes.

Em função destas características diferenciais, o processo de desenvolvimento de software pode desembocar um conjunto de problemas, os quais terão influência direta na **qualidade do produto**.

Tudo isto porque, desde os primórdios da computação, o desenvolvimento dos programas (ou, a programação) era visto como uma forma de arte, sem utilização de metodologias formais e sem qualquer preocupação com a documentação, entre outros fatores importantes. A experiência do programador era adquirida através de tentativa e erro. A verdade é que esta tendência ainda se verifica. Com o crescimento dos custos de software (em relação aos de hardware) no custo total de um sistema computacional, o

processo de desenvolvimento de software tornou-se um item de fundamental importância na produção de tais sistemas.

A nível industrial, algumas questões que caracterizaram as preocupações com o processo de desenvolvimento de software foram:

- por que o software demora tanto para ser concluído?
- por que os custos de produção têm sido tão elevados?
- por que não é possível detectar todos os erros antes que o software seja entregue ao cliente?
- por que é tão difícil medir o progresso durante o processo de desenvolvimento de software?

Estas são algumas das questões que a **Engenharia de Software** pode ajudar a resolver. Enquanto não respondemos definitivamente a estas questões, podemos levantar alguns dos problemas que as originam:

- raramente, durante o desenvolvimento de um software, é dedicado tempo para coletar dados sobre o processo de desenvolvimento em sí; devido à pouca quantidade deste tipo de informação, as tentativas em estimar a duração/custo de produção de um software têm conduzido a resultados bastante insatisfatórios; além disso, a falta destas informações impede uma avaliação eficiente das técnicas e metodologias empregadas no desenvolvimento;
- a insatisfação do cliente com o sistema "concluído" ocorre freqüentemente, devido, principalmente, ao fato de que os projetos de desenvolvimento são baseados em informações vagas sobre as necessidades e desejos do cliente (problema de comunicação entre cliente e fornecedor);
- a qualidade do software é quase sempre suspeita, problema resultante da pouca atenção que foi dada, historicamente, às técnicas de teste de software (até porque o conceito de **qualidade de software** é algo relativamente recente);
- o software existente é normalmente muito difícil de manter em operação, o que significa que o custo do software acaba sendo incrementado significativamente devido às atividades relacionadas à manutenção; isto é um reflexo da pouca importância dada à manutenibilidade no momento da concepção dos sistemas.

2.2. Software: Mitos e Realidade

É possível apontar, como causas principais dos problemas levantados na seção anterior, três principais pontos:

- a falta de experiência dos profissionais na condução de projetos de software;
- a falta de treinamento no que diz respeito ao uso de técnicas e métodos formais para o desenvolvimento de software;
- a “cultura de programação” que ainda é difundida e facilmente aceita por estudantes e profissionais de Ciências da Computação;
- a incrível "resistência" às mudanças (particularmente, no que diz respeito ao uso de novas técnicas de desenvolvimento de software) que os profissionais normalmente apresentam.

Entretanto, é importante ressaltar e discutir os chamados "mitos e realidades" do software, o que, de certo modo, explicam alguns dos problemas de desenvolvimento de software apresentados.

2.2.1. Mitos de Gerenciamento

Mito 1. *"Se a equipe dispõe de um manual repleto de padrões e procedimentos de desenvolvimento de software, então a equipe está apta a encaminhar bem o desenvolvimento."*

Realidade 1. Isto verdadeiramente não é o suficiente... é preciso que a equipe aplique efetivamente os conhecimentos apresentados no manual... é necessário que o que conste no dado manual reflita a moderna prática de desenvolvimento de software e que este seja exaustivo com relação a todos os problemas de desenvolvimento que poderão aparecer no percurso...

Mito 2. *"A equipe tem ferramentas de desenvolvimento de software de última geração, uma vez que eles dispõem de computadores de última geração."*

Realidade 2. Ter à sua disposição o último modelo de computador (seja ele um mainframe, estação de trabalho ou PC) pode ser bastante confortável para o desenvolvedor do software, mas não oferece nenhuma garantia quanto à qualidade do software desenvolvido. Mais importante do que ter um hardware de última geração é ter ferramentas para a automatização do desenvolvimento de software (as ferramentas CASE)...

Mito 3. *"Se o desenvolvimento do software estiver atrasado, basta aumentar a equipe para honrar o prazo de desenvolvimento."*

Realidade 3. Isto também dificilmente vai ocorrer na realidade... alguém disse um dia que "... acrescentar pessoas em um projeto atrasado vai torná-lo ainda mais atrasado...". De fato, a introdução de novos profissionais numa equipe em fase de condução de um projeto vai requerer uma etapa de treinamento dos novos elementos da equipe; para isto, serão utilizados elementos que estão envolvidos diretamente no desenvolvimento, o que vai, conseqüentemente, implicar em maiores atrasos no cronograma.

2.2.2. Mitos do Cliente

Mito 4. *"Uma descrição breve e geral dos requisitos do software é o suficiente para iniciar o seu projeto... maiores detalhes podem ser definidos posteriormente."*

Realidade 4. Este é um dos problemas que podem conduzir um projeto ao fracasso, o cliente deve procurar definir o mais precisamente possível todos os requisitos importantes para o software: funções, desempenho, interfaces, restrições de projeto e critérios de validação são alguns dos pontos determinantes do sucesso de um projeto.

Mito 5. *"Os requisitos de projeto mudam continuamente durante o seu desenvolvimento, mas isto não representa um problema, uma vez que o software é flexível e poderá suportar facilmente as alterações."*

Realidade 5. É verdade que o software é flexível (pelo menos mais flexível do que a maioria dos produtos manufaturados). Entretanto, não existe software, por mais flexível que suporte alterações de requisitos significativas com adicional zero em relação ao custo de desenvolvimento. O fator de multiplicação nos custos de desenvolvimento do software devido a alterações nos requisitos cresce em função do estágio de evolução do projeto, como mostra a figura 1.1.

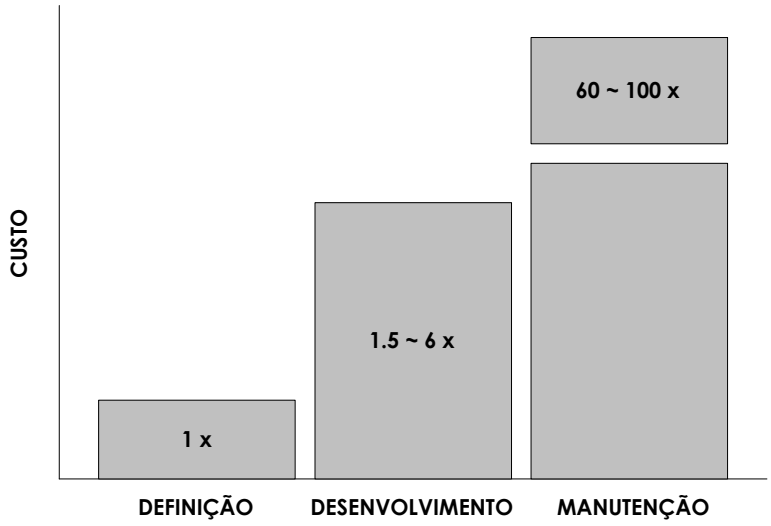


Figura 1.1 - Influência das alterações de requisitos no custo de um sistema.

2.2.2. Mitos do Profissional

Mito 6. *"Após a edição do programa e a sua colocação em funcionamento, o trabalho está terminado."*

Realidade 6. O que ocorre na realidade é completamente diferente disto. Segundo dados obtidos a partir de experiências anteriores, 50 a 70% do esforço de desenvolvimento de um software é despendido após a sua entrega ao cliente (manutenção).

Mito 7. *"Enquanto o programa não entrar em funcionamento, é impossível avaliar a sua qualidade."*

Realidade 7. Na realidade, a preocupação com a garantia do software deve fazer parte de todas as etapas do desenvolvimento, sendo que, ao fim de cada uma destas etapas, os documentos de projeto devem ser revisados observando critérios de qualidade.

Mito 8. *"O produto a ser entregue no final do projeto é o programa funcionando."*

Realidade 8. O programa em funcionamento é uma das componentes do software...além do software, um bom projeto deve ser caracterizado pela produção de um conjunto importante de documentos, os quais podem ser identificados com auxílio da figura 1.2.

3. PARADIGMAS DA ENGENHARIA DE SOFTWARE

3.1 Definição de Engenharia de Software

Os problemas apontados nas seções anteriores não serão, é claro, resolvidos da noite para o dia, mas reconhecer a existência dos problemas e definí-los da forma mais precisa e eficaz são, sem dúvida, um primeiro passo para a sua solução.

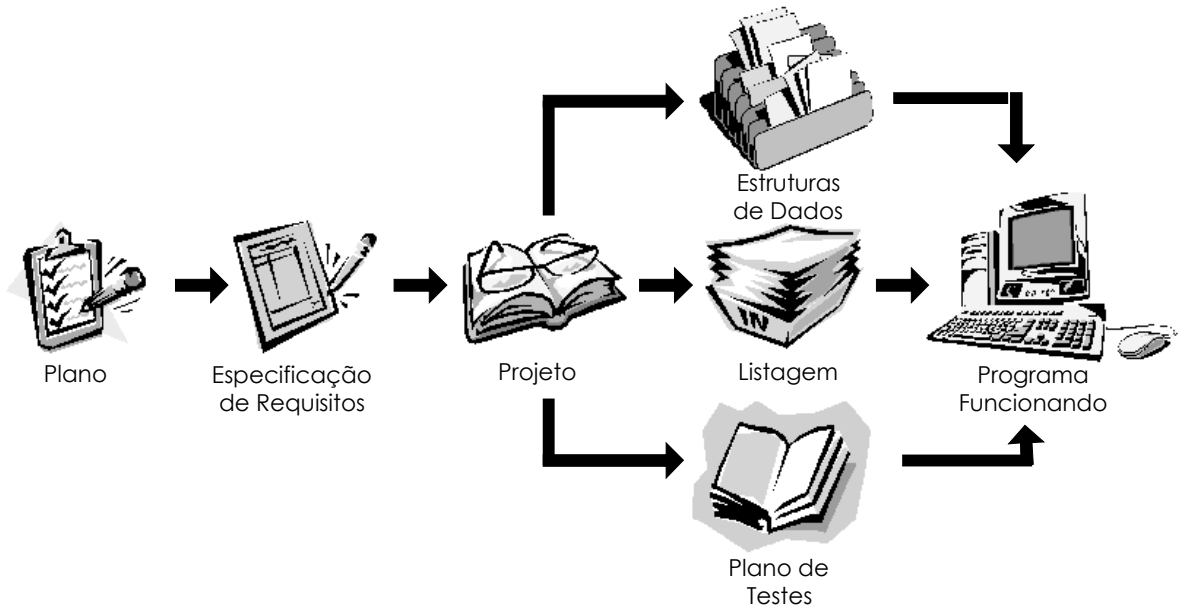


Figura 1.2 - Componentes do software.

Em primeiro lugar, é preciso estar ciente também de que não existe uma abordagem mágica que seja a melhor para a solução destes problemas, mas uma combinação de métodos que sejam abrangentes a todas as etapas do desenvolvimento de um software.

Além disto, é importante e desejável que estes métodos sejam suportados por um conjunto de ferramentas que permita automatizar o desenrolar destas etapas, juntamente com uma definição clara de critérios de qualidade e produtividade de software. São estes aspectos que caracterizam de maneira mais influente a disciplina de **Engenharia de Software**.

Na literatura, pode-se encontrar diversas definições da Engenharia de Software:

"O estabelecimento e uso de sólidos princípios de engenharia para que se possa obter economicamente um software que seja confiável e que funcione eficientemente em máquinas reais" [NAU 69].

"A aplicação prática do conhecimento científico para o projeto e a construção de programas computacionais e a documentação necessária à sua operação e manutenção." [Boehm, 76]

"Abordagem sistemática para o desenvolvimento, a operação e a manutenção de software" [Afnor, 83]

"Conjunto de métodos, técnicas e ferramentas necessárias à produção de software de qualidade para todas as etapas do ciclo de vida do produto." [Krakowiak, 85]

Num ponto de vista mais formal, a Engenharia de Software pode ser definida como sendo a aplicação da ciência e da matemática através das quais os equipamentos computacionais são colocados à disposição do homem por meio de programas, procedimentos e documentação associada. De modo mais objetivo, pode-se dizer que a Engenharia de Software busca prover a tecnologia necessária para produzir software de alta qualidade a um baixo custo. Os dois fatores motivadores são essencialmente a qualidade e o custo. A qualidade de um produto de software é um parâmetro cuja quantificação não é trivial, apesar dos esforços desenvolvidos nesta direção. Por outro lado, o fator custo pode

ser facilmente quantificado desde que os procedimentos de contabilidade tenham sido corretamente efetuados.

3.2. Modelos de Desenvolvimento de software

O resultado de um esforço de desenvolvimento deve resultar normalmente num **produto**. O **processo de desenvolvimento** corresponde ao conjunto de atividades e um ordenamento destas de modo a que o produto desejado seja obtido.

Um **modelo de desenvolvimento** corresponde a uma representação abstrata do processo de desenvolvimento que vai, em geral, definir como as etapas relativas ao desenvolvimento do software serão conduzidas e interrelacionadas para atingir o objetivo do desenvolvimento que é a obtenção de um produto de software de alta qualidade a um custo relativamente baixo.

As seções que seguem vão descrever alguns dos modelos conhecidos e utilizados em desenvolvimento de software.

3.2.1. O Modelo Queda d'Água

Este é o modelo mais simples de desenvolvimento de software, estabelecendo uma ordenação linear no que diz respeito à realização das diferentes etapas. Como mostra a figura 1.3, o ponto de partida do modelo é uma etapa de **Engenharia de Sistemas**, onde o objetivo é ter uma visão global do sistema como um todo (incluindo hardware, software, equipamentos e as pessoas envolvidas) como forma de definir precisamente o papel do software neste contexto. Em seguida, a etapa de **Análise de Requisitos** vai permitir uma clara definição dos requisitos de software, sendo que o resultado será utilizado como referência para as etapas posteriores de **Projeto**, **Codificação**, **Teste** e **Manutenção**.

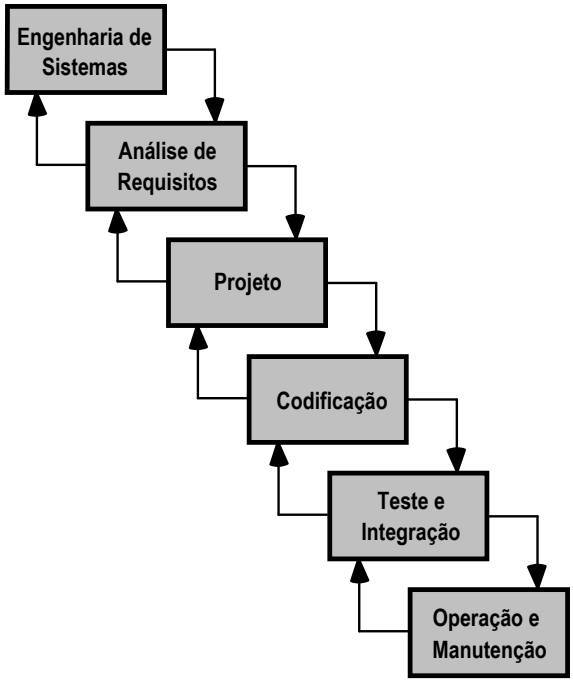


Figura 1.3 - Ilustração do modelo Queda d'Água.

O modelo Queda d'Água apresenta características interessantes, particularmente em razão da definição de um ordenamento linear das etapas de desenvolvimento. Primeiramente, como forma de identificar precisamente o fim de uma etapa de o início da seguinte, um mecanismo de certificação (ou revisão) é implementado ao final de cada

etapa; isto é feito normalmente através da aplicação de algum método de validação ou verificação, cujo objetivo será garantir de que a saída de uma dada etapa é coerente com a sua entrada (a qual já é a saída da etapa precedente). Isto significa que ao final de cada etapa realizada, deve existir um resultado (ou saída) a qual possa ser submetida à atividade de certificação.

Estas saídas, obtidas ao final de cada etapa, são vistas como produtos intermediários e apresentam-se, normalmente, na forma de documentos (documento de especificação de requisitos, documento de projeto do sistema, etc...).

Outra característica importante deste modelo é que as saídas de uma etapa são as entradas da seguinte, o que significa que uma vez definidas, elas não devem, em hipótese alguma ser modificadas.

Duas diretivas importantes que norteiam o desenvolvimento segundo o modelo Queda d'Água, são:

- todas as etapas definidas no modelo devem ser realizadas, isto porque, em projetos de grande complexidade, a realização formal destas vai determinar o sucesso ou não do desenvolvimento; a realização informal e implícita de algumas destas etapas poderia ser feita apenas no caso de projetos de pequeno porte;
- a ordenação das etapas na forma como foi apresentada deve ser rigorosamente respeitada; apesar de que esta diretiva poderia ser questionada, a ordenação proposta pelo modelo, por ser a forma mais simples de desenvolver, tem sido também a mais adotada a nível de projetos de software.

É importante lembrar, também que os resultados de um processo de desenvolvimento de software não devem ser exclusivamente o programa executável e a documentação associada. Existe uma quantidade importante de resultados (ou produtos intermediários) os quais são importantes para o sucesso do desenvolvimento. Baseados nas etapas apresentadas na figura 1.3, é possível listar os resultados mínimos esperados de um processo de desenvolvimento baseado neste modelo: Documento de Especificação de Requisitos, Projeto do Sistema, Plano de Teste e Relatório de Testes, Código Final, Manuais de Utilização, Relatórios de Revisões.

Apesar de ser um modelo bastante popular, pode-se apontar algumas limitações apresentadas por este modelo:

- o modelo assume que os requisitos são inalterados ao longo do desenvolvimento; isto em boa parte dos casos não é verdadeira, uma vez que nem todos os requisitos são completamente definidos na etapa de análise;
- muitas vezes, a definição dos requisitos pode conduzir à definição do hardware sobre o qual o sistema vai funcionar; dado que muitos projetos podem levar diversos anos para serem concluídos, estabelecer os requisitos em termos de hardware é um tanto temeroso, dadas as freqüentes evoluções no hardware;
- o modelo impõe que todos os requisitos sejam completamente especificados antes do prosseguimento das etapas seguintes; em alguns projetos, é às vezes mais interessante poder especificar completamente somente parte do sistema, prosseguir com o desenvolvimento do sistema, e só então encaminhar os requisitos de outras partes; isto não é previsto a nível do modelo;
- as primeiras versões operacionais do software são obtidas nas etapas mais tardias do processo, o que na maioria das vezes inquieta o cliente, uma vez que ele quer ter acesso rápido ao seu produto.

De todo modo, pode vir a ser mais interessante a utilização deste modelo para o desenvolvimento de um dado sistema do que realizar um desenvolvimento de maneira totalmente anárquica e informal.

3.2.2. Prototipação

O objetivo da Prototipação é um modelo de processo de desenvolvimento que busca contornar algumas das limitações existentes no modelo Queda d'Água. A idéia por trás deste modelo é eliminar a política de "congelamento" dos requisitos antes do projeto do sistema ou da codificação.

Isto é feito através da obtenção de um protótipo, com base no conhecimento dos requisitos iniciais para o sistema. O desenvolvimento deste protótipo é feito obedecendo à realização das diferentes etapas já mencionadas, a saber, a análise de requisitos, o projeto, a codificação e os testes, sendo que não necessariamente estas etapas sejam realizadas de modo muito explícito ou formal.

Este protótipo pode ser oferecido ao cliente em diferentes formas, a saber:

- protótipo em papel ou modelo executável em PC retratando a interface homem-máquina capacitando o cliente a compreender a forma de interação com o software;
- um protótipo de trabalho que implemente um subconjunto dos requisitos indicados;
- um programa existente (pacote) que permita representar todas ou parte das funções desejadas para o software a construir.

Colocado à disposição do cliente, o protótipo vai ajudá-lo a melhor compreender o que será o sistema desenvolvido. Além disso, através da manipulação deste protótipo, é possível validar ou reformular os requisitos para as etapas seguintes do sistema.

Este modelo, ilustrado na figura 1.4, apresenta algumas características interessantes, tais como:

- é um modelo de desenvolvimento interessante para alguns sistemas de grande porte os quais representem um certo grau de dificuldade para exprimir rigorosamente os requisitos;
- através da construção de um protótipo do sistema, é possível demonstrar a realizabilidade do mesmo;
- é possível obter uma versão, mesmo simplificada do que será o sistema, com um pequeno investimento inicial.

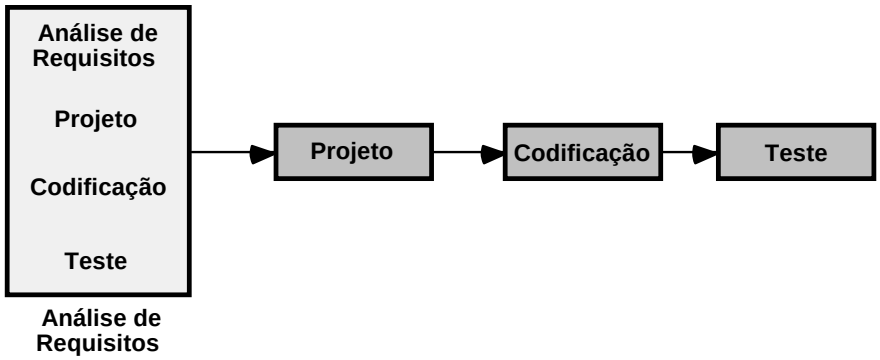


Figura 1.4 - Esquema de evolução da prototipação.

Os protótipos não são sistemas completos e deixam, normalmente, a desejar em alguns aspectos. Um destes aspectos é normalmente a interface com o usuário. Os esforços de desenvolvimento são concentrados principalmente nos algoritmos que implementem as principais funções associadas aos requisitos apresentados, a interface

sendo, a este nível parte supérflua do desenvolvimento, o que permite caracterizar esta etapa por um custo relativamente baixo.

Por outro lado, a experiência adquirida no desenvolvimento do protótipo vai ser de extrema utilidade nas etapas posteriores do desenvolvimento do sistema real, permitindo reduzir certamente o seu custo, resultando também num sistema melhor concebido.

3.2.3. Desenvolvimento Iterativo

Este modelo também foi concebido com base numa das limitações do modelo Queda d'Água e combinar as vantagens deste modelo com as do modelo Prototipação. A idéia principal deste modelo, ilustrada na figura 1.5, é a de que um sistema deve ser desenvolvido de forma incremental, sendo que cada incremento vai adicionando ao sistema novas capacidades funcionais, até a obtenção do sistema final, sendo que, a cada passo realizado, modificações podem ser introduzidas.

Uma vantagem desta abordagem é a facilidade em testar o sistema, uma vez que a realização de testes em cada nível de desenvolvimento é, sem dúvida, mais fácil do que testar o sistema final. Além disso, como na Prototipação, a obtenção de um sistema, mesmo incompleto num dado nível, pode oferecer ao cliente interessantes informações que sirvam de subsídio para a melhor definição de futuros requisitos do sistema.

No primeiro passo deste modelo uma implementação inicial do sistema é obtida, na forma de um subconjunto da solução do problema global. Este primeiro nível de sistema deve contemplar os principais aspectos que sejam facilmente identificáveis no que diz respeito ao problema a ser resolvido.

Um aspecto importante deste modelo é a criação de uma **lista de controle de projeto**, a qual deve apresentar todos os passos a serem realizados para a obtenção do sistema final. Ela vai servir também para se medir, num dado nível, o quão distante se está da última iteração. Cada iteração do modelo de Desenvolvimento Iterativo consiste em retirar um passo da lista de controle de projeto através da realização de três etapas: o projeto, a implementação e a análise. O processo avança, sendo que a cada etapa de avaliação, um passo é retirado da lista, até que a lista esteja completamente vazia. A lista de controle de projeto gerencia todo o desenvolvimento, definindo quais tarefas devem ser realizadas a cada iteração, sendo que as tarefas na lista podem representar, inclusive, redefinições de componentes já implementados, em razão de erros ou problemas detectados numa eventual etapa de análise.

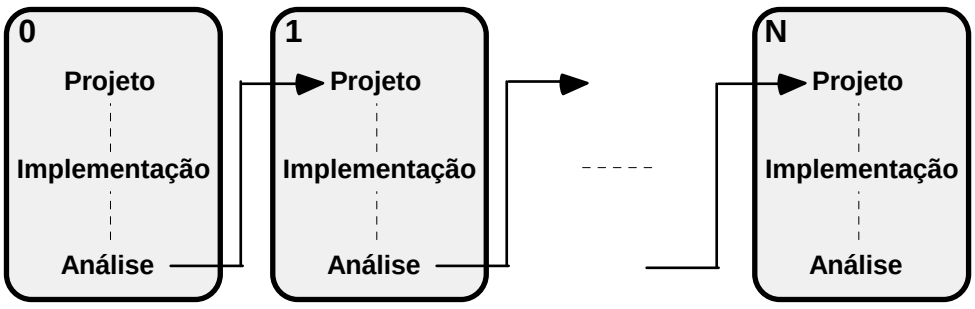


Figura 1.5 - O modelo Desenvolvimento Iterativo.

3.2.4. O Modelo Espiral

Este modelo, proposto em 1988, sugere uma organização das atividades em espiral, a qual é composta de diversos ciclos. Como mostrado na figura 1.6, a dimensão vertical representa o custo acumulado na realização das diversas etapas; a dimensão angular representa o avanço do desenvolvimento ao longo das etapas.

Cada ciclo na espiral inicia com a identificação dos objetivos e as diferentes alternativas para se atingir aqueles objetivos assim como as restrições impostas. O próximo passo no ciclo é a avaliação das diferentes alternativas com base nos objetivos fixados, o que vai permitir também definir incertezas e riscos de cada alternativa. No passo seguinte, o desenvolvimento de estratégias permitindo resolver ou eliminar as incertezas levantadas anteriormente, o que pode envolver atividades de prototipação, simulação, avaliação de desempenho, etc... Finalmente, o software é desenvolvido e o planejamento dos próximos passos é realizado.

A continuidade do processo de desenvolvimento é definida como função dos riscos remanescentes, como por exemplo, a decisão se os riscos relacionados ao desempenho ou à interface são mais importantes do que aqueles relacionados ao desenvolvimento do programa. Com base nas decisões tomadas, o próximo passo pode ser o desenvolvimento de um novo protótipo que elimine os riscos considerados.

Por outro lado, caso os riscos de desenvolvimento de programa sejam considerados os mais importantes e se o protótipo obtido no passo corrente já resolve boa parte dos riscos ligados a desempenho e interface, então o próximo passo pode ser simplesmente a evolução segundo o modelo Queda d'Água.

Como se pode ver, o elemento que conduz este processo é essencialmente a consideração sobre os riscos, o que permite, de certo modo, a adequação a qualquer política de desenvolvimento (baseada em especificação, baseada em simulação, baseada em protótipo, etc...).

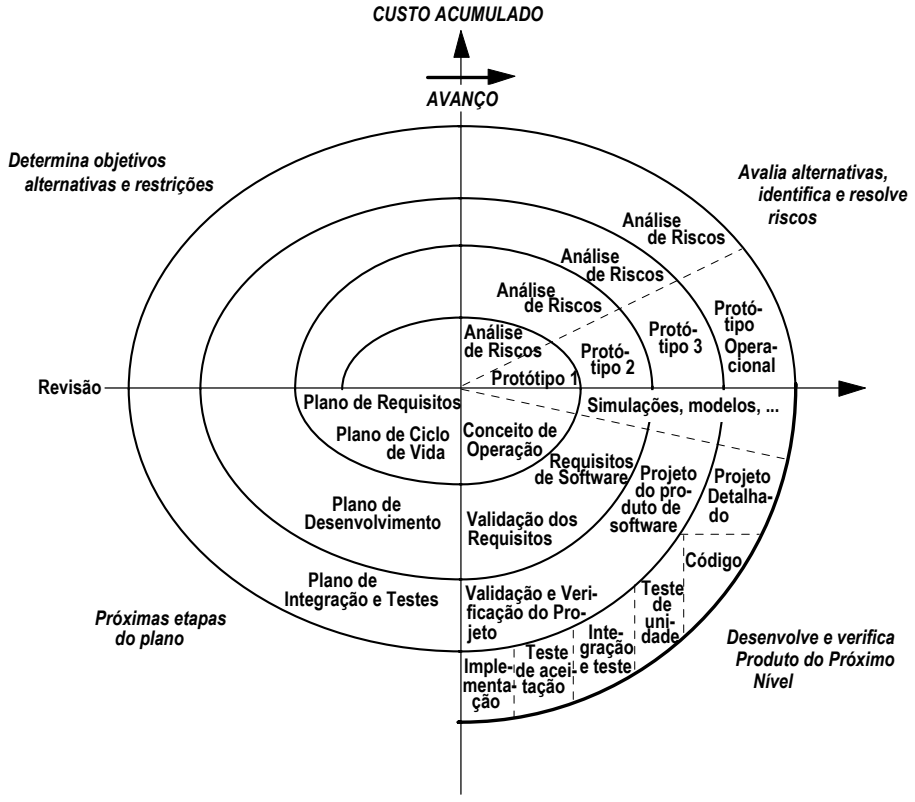


Figura 1.7 - O modelo espiral.

Uma característica importante deste modelo é o fato de que cada ciclo é encerrado por uma atividade de revisão, onde todos os produtos do ciclo são avaliados, incluindo o plano para o próximo passo (ou ciclo). Numa aplicação típica do modelo, pode-se imaginar a realização de um ciclo zero, onde se avalie a realizabilidade do projeto, o resultado devendo ser a conclusão de que será possível implementar ou não o projeto de desenvolvimento. As alternativas consideradas neste caso são de muito alto nível, como por

exemplo, se a organização deve desenvolver o sistema ela própria ou se deve contratar o desenvolvimento junto a uma empresa especializada.

O modelo se adequa principalmente a sistemas que representem um alto risco de investimento para o cliente.

4. VISÃO GERAL DA ENGENHARIA DE SOFTWARE

Analisando os modelos apresentados na seção precedente, é possível observar que, apesar de apresentar denominações às vezes diferentes e de estarem associadas de modo relativamente distinto, as etapas apresentadas são caracterizadas por atividades similares.

De um modo geral, pode-se organizar o processo de desenvolvimento de um software a partir de três grandes fases: a **fase de definição**, a **fase de desenvolvimento** e a **fase de manutenção** as quais serão discutidas nas seções abaixo.

4.1. Fase de Definição

A fase de definição está associada à determinação do **que** vai ser feito. Nesta fase, o profissional encarregado do desenvolvimento do software deve identificar as informações que deverão ser manipuladas, as funções a serem processadas, qual o nível de desempenho desejado, que interfaces devem ser oferecidas, as restrições do projeto e os critérios de validação. Isto terá de ser feito não importando o modelo de desenvolvimento adotado para o software e independente da técnica utilizada pra fazê-lo.

Esta fase é caracterizada pela realização de três etapas específicas:

- a **Análise (ou Definição) do Sistema**, a qual vai permitir determinar o papel de cada elemento (hardware, software, equipamentos, pessoas) no sistema, cujo objetivo é determinar, como resultado principal, as funções atribuídas ao software;
- o **Planejamento do Projeto de Software**, no qual, a partir da definição do escopo do software, será feita uma análise de riscos e a definição dos recursos, custos e a programação do processo de desenvolvimento;
- a **Análise de Requisitos**, que vai permitir determinar o conjunto das funções a serem realizadas assim como as principais estruturas de informação a serem processadas.

4.2. Fase de Desenvolvimento

Nesta fase, será determinado **como** realizar as funções do software. Aspectos como a arquitetura do software, as estruturas de dados, os procedimentos a serem implementados, a forma como o projeto será transformado em linguagem de programação, a geração de código e os procedimentos de teste devem ser encaminhados nesta fase.

Normalmente, esta fase é também organizada em três principais etapas:

- o **Projeto de Software**, o qual traduz, num conjunto de representações gráficas, tabulares ou textuais, os requisitos do software definidos na fase anterior; estas representações (diversas técnicas de representação podem ser adotadas num mesmo projeto) permitirão definir, com um alto grau de abstração, aspectos do software como a arquitetura, os dados, lógicas de comportamento (algoritmos) e características da interface;
- a **Codificação**, onde as representações realizadas na etapa de projeto serão mapeadas numa ou em várias linguagens de programação, a qual será caracterizada por um conjunto de instruções executáveis no computador; nesta etapa, considera-se também a geração de código de implementação, aquele obtido a partir do uso de ferramentas (compiladores, linkers, etc...) e que será executado pelo hardware do sistema;

- os **Testes de Software**, onde o programa obtido será submetido a uma bateria de testes para verificar (e corrigir) defeitos relativos às funções, lógica de execução, interfaces, etc...

4.3. Fase de Manutenção

A fase de manutenção, que se inicia a partir da entrega do software, é caracterizada pela realização de alterações de naturezas as mais diversas, seja para corrigir erros residuais da fase anterior, para incluir novas funções exigidas pelo cliente, ou para adaptar o software a novas configurações de hardware.

Sendo assim, pode-se caracterizar esta fase pelas seguintes atividades:

- a **Correção** ou **Manutenção Corretiva**, a qual consiste da atividade de correção de erros observados durante a operação do sistema;
- a **Adaptação** ou **Manutenção Adaptativa**, a qual realiza alterações no software para que ele possa ser executado sobre um novo ambiente (CPU, arquitetura, novos dispositivos de hardware, novo sistema operacional, etc...);
- o **Melhoramento Funcional** ou **Manutenção Perfectiva**, onde são realizadas alterações para melhorar alguns aspectos do software, como por exemplo, o seu desempenho, a sua interface, a introdução de novas funções, etc...

A manutenção do software envolve, normalmente, etapas de análise do sistema existente (entendimento do código e dos documentos associados), teste das mudanças, teste das partes já existentes, o que a torna uma etapa complexa e de alto custo.

Além disso, considerando a atual situação industrial, foi criado, mais recentemente, o conceito de **Engenharia Reversa**, onde, através do uso das técnicas e ferramentas da Engenharia de Software, o software existente sofre uma "reforma geral", cujo objetivo é aumentar a sua qualidade e atualizá-lo com respeito às novas tecnologias de interface e de hardware.

Engenharia de Software

Capítulo 2

Qualidade de Software

1. Introdução

Como foi mencionado no capítulo anterior, o papel da Engenharia de Software é, principalmente, fornecer métodos e ferramentas para o desenvolvimento do software de qualidade e a baixo custo.

O fator qualidade é um dos aspectos importantes que deve ser levado em conta quando do desenvolvimento do software. Para isto, é necessário que se tenha uma definição precisa do que é um software de qualidade ou, pelo menos, quais são as propriedades que devem caracterizar um software desenvolvido segundo os princípios da Engenharia de Software.

Um outro aspecto importante é aquele relacionado à avaliação e ao aprimoramento do processo de desenvolvimento de software de uma organização. Nesta linha, foi desenvolvido pelo SEI (Software Engineering Institute) um modelo que permite definir parâmetros para a análise desta questão nas corporações, o modelo CMM (Capability and Maturity Model), cujas linhas gerais serão descritas na seção 5.

2. Definição de Software de Qualidade

Primeiramente, é importante discutir o conceito de software de qualidade. Segundo a Associação Francesa de Normalização, AFNOR, a qualidade é definida como *"a capacidade de um produto ou serviço de satisfazer às necessidades dos seus usuários"*.

Esta definição, de certa forma, é coerente com as metas da Engenharia de Software, particularmente quando algumas definições são apresentadas. É o caso das definições de Verificação e Validação introduzidas por Boehm, que associa a estas definições as seguintes questões:

- **Verificação:** "Será que o produto foi construído corretamente?"
- **Validação:** "Será que este é o produto que o cliente solicitou?"

O problema que surge quando se reflete em termos de qualidade é a dificuldade em se quantificar este fator.

2.1. Fatores de Qualidade Externos e Internos

Algumas das propriedades que poderíamos apontar de imediato são a correção, a facilidade de uso, o desempenho, a legibilidade, etc... Na verdade, analizando estas propriedades, é possível organizá-las em dois grupos importantes de fatores, que vamos denominar fatores externos e internos.

Os fatores de qualidade **externos**, são aqueles que podem ser detectados principalmente pelo cliente ou eventuais usuários. A partir da observação destes fatores, o

cliente pode concluir sobre a qualidade do software, do seu ponto de vista. Enquadram-se nesta classe fatores tais como: o desempenho, a facilidade de uso, a correção, a confiabilidade, a extensibilidade, etc...

Já os fatores de qualidade **internos** são aqueles que estão mais relacionados à visão de um programador, particularmente aquele que vai assumir as tarefas de manutenção do software. Nesta classe, encontram-se fatores como: modularidade, legibilidade, portabilidade, etc...

Não é difícil verificar que, normalmente, os fatores mais considerados quando do desenvolvimento do software são os externos. Isto porque, uma vez que o objetivo do desenvolvimento do software é satisfazer ao cliente, são estes fatores que vão assumir um papel importante na avaliação do produto (da parte do cliente, é claro!!!).

No entanto, também não é difícil concluir que são os fatores internos que vão garantir o alcance dos fatores externos.

2.2. Fatores de Qualidade

2.2.1. Correção

É a capacidade dos produtos de software de realizarem suas tarefas de forma precisa, conforme definido nos requisitos e na especificação. É um fator de suma importância em qualquer categoria de software. Nenhum outro fator poderá compensar a ausência de correção. Não é interessante produzir um software extremamente desenvolvido do ponto de vista da interface homem-máquina, por exemplo, se as suas funções são executadas de forma incorreta. É preciso dizer, porém, que a correção é um fator mais facilmente afirmado do que alcançado. O alcance de um nível satisfatório de correção vai depender, principalmente, da formalização dos requisitos do software e do uso de métodos de desenvolvimento que explorem esta formalização.

2.2.2. Robustez

A robustez é a capacidade do sistema funcionar mesmo em condições anormais. É um fator diferente da correção. Um sistema pode ser correto sem ser robusto, ou seja, o seu funcionamento vai ocorrer somente em determinadas condições. O aspecto mais importante relacionado à robustez é a obtenção de um nível de funcionamento do sistema que suporte mesmo situações que não foram previstas na especificação dos requisitos. Na pior das hipóteses, é importante garantir que o software não vai provocar conseqüências catastróficas em situações anormais. Resultados esperados são que, em tais situações, o software apresente comportamentos tais como: a sinalização da situação anormal, a terminação "ordenada", a continuidade do funcionamento "correto" mesmo de maneira degradada, etc... Na literatura, estas características podem ser associadas também ao conceito de **confiabilidade**.

2.2.3. Extensibilidade

É a facilidade com a qual se pode introduzir modificações nos produtos de software. Todo software é considerado, em princípio, "flexível" e, portanto, passível de modificações. No entanto, este fator nem sempre é muito bem entendido, principalmente quando se trata de pequenos programas. Por outro lado, para softwares de grande porte, este fator atinge uma importância considerável, e pode ser atingido a partir de dois critérios importantes:

- a **simplicidade de projeto**, ou seja, quanto mais simples e clara a arquitetura do software, mais facilmente as modificações poderão ser realizadas;
- a **descentralização**, que implica na maior autonomia dos diferentes componentes de software, de modo que a modificação ou a retirada de um componente não

implique numa reação em cadeia que altere todo o comportamento do sistema, podendo inclusive introduzir erros antes inexistentes.

2.2.4. Reusabilidade

É a capacidade dos produtos de software serem reutilizados, totalmente ou em parte, para novas aplicações. A necessidade vem da constatação de que muitos componentes de software obedecem a um padrão comum, o que permite então que estas similaridades possam ser exploradas para a obtenção de soluções para outras classes de problemas.

Este fator permite, principalmente, atingir uma grande economia e um nível de qualidade satisfatórios na produção de novos softwares, dado que menos programa precisa ser escrito, o que significa menos esforço e menor risco de ocorrência de erros. Isto significa de fato que a reusabilidade pode influir em outros fatores de qualidade importantes, tais como a correção e a robustez.

2.2.5. Compatibilidade

A compatibilidade corresponde à facilidade com a qual produtos de software podem ser combinados com outros. Este é um fator relativamente importante, dado que um produto de software é construído (e adquirido) para trabalhar convivendo com outros softwares. A impossibilidade de interação com outros produtos pode ser, sem dúvida, uma característica que resultará na não escolha do software ou no abandono de sua utilização. Os contra-exemplos de compatibilidade, infelizmente, são maiores que os exemplos, mas podemos citar, nesta classe, principalmente, a definição de formatos de arquivos padronizados (ASCII, PostScript, ...), a padronização de estruturas de dados, a padronização de interfaces homem-máquina (sistema do Macintosh, ambiente Windows, ...), etc...

2.2.6. Eficiência

A eficiência está relacionada com a utilização racional dos recursos de hardware e de sistema operacional da plataforma onde o software será instalado. Recursos tais como memória, processador e co-processador, memória cache, recursos gráficos, bibliotecas (por exemplo, primitivas de sistema operacional) devem ser explorados de forma adequada em espaço e tempo.

2.2.7. Portabilidade

A portabilidade consiste na capacidade de um software em ser instalado para diversos ambientes de software e hardware. Esta nem sempre é uma característica facilmente atingida, devido principalmente às diversidades existentes nas diferentes plataformas em termos de processador, composição dos periféricos, sistema operacional, etc...

Alguns softwares, por outro lado, são construídos em diversas versões, cada uma delas orientada para execução num ambiente particular. Exemplos disto são alguns programas da Microsoft, como por exemplo o pacote Microsoft Office, que existe para plataformas Macintosh e microcomputadores compatíveis IBM.

2.2.8. Facilidade de uso

Este fator é certamente um dos mais fortemente detectados pelos usuários do software. Atualmente, com o grande desenvolvimento dos ambientes gráficos como Windows, X-Windows e o Sistema Macintosh, a obtenção de softwares de fácil utilização tornou-se mais freqüente. A adoção de procedimentos de auxílio em linha (help on line) é, sem dúvida, uma grande contribuição neste sentido. Dada a definição de software feita no

início do curso, porém, não se pode deixar de registrar a importância de uma documentação adequada (manual de usuário) capaz de orientar o usuário em sua navegação pelo software considerado.

3. A METROLOGIA DA QUALIDADE DO SOFTWARE

Apresentados alguns fatores de qualidade de software, a dificuldade que se apresenta é como medir a qualidade do software. Ao contrário de outras disciplinas de engenharia, onde os produtos gerados apresentam características físicas como o peso, a altura, tensão de entrada, tolerância a erros, etc..., a medida da qualidade do software é algo relativamente novo e alguns dos critérios utilizados são questionáveis.

A possibilidade de estabelecer uma medida da qualidade é um aspecto importante para a garantia de um produto de software com algumas das características definidas anteriormente.

O Metrologia do Software corresponde ao conjunto de teorias e práticas relacionadas com as medidas, a qual permite estimar o desempenho e o custo do software, a comparação de projetos e a fixação dos critérios de qualidade a atingir.

As medidas de um software podem ser as mais variadas, a escolha da medida devendo obedecer a determinados critérios: a pertinência da medida (interpretabilidade); o custo da medida (percentual reduzido do custo do software); utilidade (possibilidade de comparação de medidas).

As medidas de um software podem ser organizadas segundo duas grandes categorias:

3.1. Medidas dinâmicas

São as medidas obtidas mediante a execução do software, como, por exemplo, os testes, as medidas de desempenho em velocidade e ocupação de memória, os traços, etc... Estas medidas podem assumir um interesse relativamente grande pois elas permitem quantificar fatores que podem implicar em retorno econômico ou que representem informações sobre a correção do programa; por outro lado, estas medidas só podem ser obtidas num momento relativamente tardio do ciclo de desenvolvimento de um software, ou seja, a partir da etapa de testes.

3.2. Medidas estáticas

São aquelas obtidas a partir do documento fonte do software, sem a necessidade de execução do software. Dentre as medidas estáticas, podemos destacar:

- as medidas de complexidade textual, a qual é baseada na computação do número de operadores e do número de operandos contidos no texto do software;
- a complexidade estrutural, a qual se baseia na análise dos grafos de controle associadas a cada componente do software;
- as medidas baseadas no texto, como o tamanho do programa, a taxa de linhas de comentários, etc...

4. QUESTÕES IMPORTANTES À QUALIDADE DE SOFTWARE

Uma vez que alguns parâmetros relacionados à qualidade foram detectados na seção 2.2, cabe aqui discutir alguns princípios que devem fazer parte das preocupações de uma equipe de desenvolvimento de software...

4.1. O PRINCÍPIO DO USO: O SOFTWARE DEVERÁ SER UTILIZADO POR OUTROS

Este princípio implica num cuidado em tornar o software acessível não apenas para o usuário imediato, mas também para futuros programadores que irão trabalhar no código fonte, seja para eliminação de erros, seja para introdução ou aprimoramento de funções.

Com base nestes princípios, o programador ou a equipe de programação deverá prover a necessária flexibilidade e robustez ao programa desde o início do desenvolvimento e não inserir estes aspectos à medida que os problemas vão surgindo.

Um outro ponto importante é que muitas vezes um programador julga que determinadas soluções encontradas a nível de um programa são tão específicas que mais ninguém, incluindo o próprio autor, irá utilizar tal solução novamente. As diversas experiências mostram que isto podem ser um grave erro de avaliação, pois conduz, na maioria dos casos, à geração de código incompreensível, representando um grande obstáculo à reutilização.

Algumas providências no desenvolvimento de um software suportam o respeito a este princípio:

- raciocinar, desde o início do desenvolvimento, em termos de um software público; isto vai proporcionar uma economia relativamente grande no que diz respeito ao tempo necessário para a depuração ou modificação de partes do código;
- documentar suficientemente o programa, o que vai permitir uma economia em horas de explicação a outras pessoas de como funciona o programa ou como ele deve ser utilizado;
- projetar o software para que ele seja utilizável por iniciantes;
- rotular todas as saídas, salvar ou documentar todas as entradas, o que pode facilitar o trabalho de interpretação de resultados obtidos durante a execução do software.

4.2. O PRINCÍPIO DO ABUSO: O SOFTWARE SERÁ UTILIZADO DE MANEIRA INDEVIDA

Este princípio diz respeito às manipulações incorretas que os usuários imprimem ao programa, seja por desconhecimento do modo correto de utilização, seja por simples curiosidade. Exemplos destas manipulações são a manipulação de arquivos de entrada de um programa nas mais diversas condições (arquivos vazios, arquivos binários, arquivos muito grandes, etc...) ou incorreções sintáticas. Muitas vezes, entradas sintaticamente corretas podem provocar erros de execução (valores fora de faixa, erros de overflow, divisão por zero, etc...).

Um outro problema é a tentativa, por parte de usuários, de utilização de um dado software para uma outra finalidade que não aquela para a qual o software foi desenvolvido.

Para levar em conta este princípio, os cuidados que o programador deve ter são os seguintes:

- garantir que o software ofereça alguma saída satisfatória para qualquer tipo de entrada;
- evitar que o programa termine de forma anormal;
- chamar a atenção para entradas alteradas ou saídas incorretas.

4.3. O PRINCÍPIO DA EVOLUÇÃO: O SOFTWARE SERÁ MODIFICADO

Este é outro aspecto de importância no desenvolvimento de software, o qual está fortemente ligado com o aspecto da facilidade de manutenção (ou extensibilidade). É fato reconhecido que os softwares deverão sofrer modificações, estas pelas razões mais diversas; seja devido à deteção de um erro residual do desenvolvimento; seja por novos requisitos surgidos após a instalação do software.

Isto torna evidente a necessidade de desenvolver o projeto e o código de modo a facilitar as modificações necessárias. A existência de um código bem estruturado e documentado é pelo menos 50% do trabalho resolvido.

Dentro deste princípio, os cuidados a serem tomados deverão ser:

- o desenvolvimento de programas com bom nível de legibilidade;
- a estruturação em componentes de software facilmente modificáveis;
- agrupar alterações visíveis ao usuário e eventuais correções em versões numeradas;
- manter a documentação atualizada.

4.4. O princípio da migração: o software será instalado em outras máquinas

Este último princípio leva em conta o fato de que a vida de muitos softwares deve ultrapassar a vida da própria máquina em que ele está executando. A velocidade com a qual as arquiteturas de computador têm evoluido nos últimos anos acentua a importância deste princípio.

Um problema relacionado a este princípio é que às vezes os programas são desenvolvidos explorando algumas particularidades das arquiteturas das máquinas para as quais eles estão sendo concebidos. Isto cria uma forte dependência do hardware, o que dificulta a futura migração para outras arquiteturas.

Isto significa, na verdade, que para um programa apresentar um nível satisfatório de portabilidade, o programador deve evitar amarrar-se a detalhes de hardware da máquina ao invés de "aproveitá-los".

Assim, podemos sintetizar as ações a serem preparadas com relação a este princípio:

- pensar os programas como meio para solucionar problemas e não para instruir determinado processador;
- utilizar as construções padronizadas das linguagens de programação ou sistemas operacionais;
- isolar e comentar todos os aspectos do programa que envolvam a dependência do hardware.

5. O modelo CMM

A causa mais comum do insucesso dos projetos de desenvolvimento de software é a má utilização ou a completa indiferença aos métodos e ferramentas orientados à concepção. É possível que, em empresas de desenvolvimento de software onde o uso de metodologias consistentes não seja prática adotada, os projetos possam ter bons resultados. Mas, em geral, estes bons resultados são muito mais uma conseqüência de esforços individuais do que propriamente causadas pela existência de uma política e de uma infra-estrutura adequada à produção de software.

O modelo CMM — Capability Maturity Model — foi definido pelo SEI — Software Engineering Institute — com o objetivo de estabelecer conceitos relacionados aos níveis de maturidade das empresas de desenvolvimento de software, com respeito ao grau de evolução que estas se encontram nos seus processos de desenvolvimento.

O modelo estabelece também que providências as empresas podem tomar para aumentarem, gradualmente o seu grau de maturidade, melhorando, por conseqüência, sua produtividade e a qualidade do produto de software.

5.1. Conceitos básicos do CMM

Um **Processo de Desenvolvimento de Software** corresponde ao conjunto de atividades, métodos, práticas e transformações que uma equipe utiliza para desenvolver e manter software e seus produtos associados (planos de projeto, documentos de projeto, código, casos de teste e manuais de usuário). Uma empresa é considerada num maior grau de maturidade quanto mais evoluído for o seu processo de desenvolvimento de software.

A **Capabilidade** de um processo de software está relacionada aos resultados que podem ser obtidos pela sua utilização num ou em vários projetos. Esta definição permite estabelecer uma estimação de resultados em futuros projetos.

O **Desempenho** de um processo de software representa os resultados que são correntemente obtidos pela sua utilização. A diferença básica entre estes dois conceitos está no fato de que, enquanto o primeiro, está relacionado aos resultados "esperados", o segundo, relaciona-se aos resultados que foram efetivamente obtidos.

A **Maturidade** de um processo de software estabelece os meios pelos quais ele é definido, gerenciado, medido, controlado e efetivo, implicando num potencial de evolução da capabilidade. Numa empresa com alto grau de maturidade, o processo de desenvolvimento de software é bem entendido por todo o staff técnico, graças à existência de documentação e políticas de treinamento, e que este é continuamente monitorado e aperfeiçoado por seus usuários.

À medida que uma organização cresce em termos de maturidade, ela institucionaliza seu processo de desenvolvimento de software através de políticas, normas e estruturas organizacionais, as quais geram uma infra-estrutura e uma cultura de suporte aos métodos e procedimentos de desenvolvimento.

5.2. Níveis de maturidade no processo de desenvolvimento de software

O modelo CMM define cinco níveis de maturidade no que diz respeito ao processo de desenvolvimento de software adotado a nível das empresas, estabelecendo uma escala ordinal que conduz as empresas ao longo de seu aperfeiçoamento.

Um nível de maturidade é um patamar de evolução de um processo de desenvolvimento de software, correspondendo a um degrau na evolução contínua de cada organização. A cada nível corresponde um conjunto de objetivos que, uma vez atingidos, estabilizam um componente fundamental do processo de desenvolvimento de software, tendo como conseqüência direta o aumento da capabilidade da empresa.

A figura 2.1 apresenta os cinco níveis de maturidade propostos no modelo CMM, na qual se pode observar também o estabelecimento de um conjunto de ações que permitirão a uma empresa subir de um degrau para o outro nesta escala.

5.2.1. Nível Inicial

No ***nível inicial***, o desenvolvimento de software é realizado de forma totalmente "ad hoc", sem uma definição de processos. No caso de problemas que venham a ocorrer durante a realização de um projeto, a organização tem uma tendência a abandonar totalmente os procedimentos planejados e passa a um processo de codificação e testes, onde o produto obtido pode apresentar um nível de qualidade suspeito.

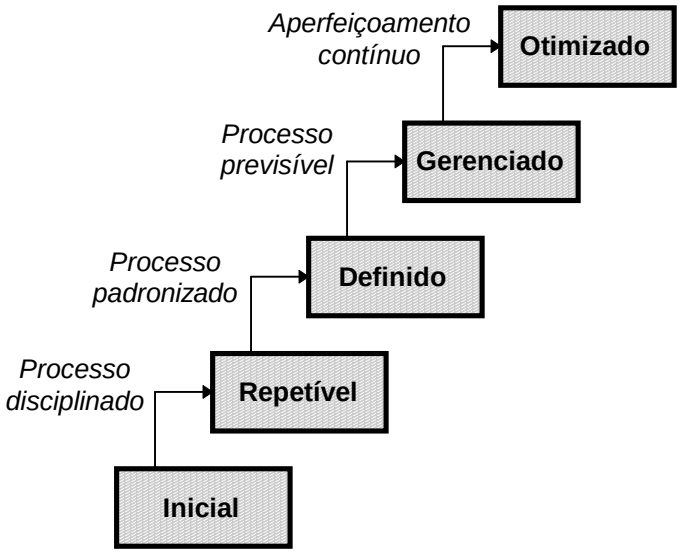


Figura 2.1 - Os níveis de maturidade de um processo de desenvolvimento de software.

A capabilidade de uma empresa caracterizada como nível 1 é totalmente imprevisível, uma vez que o processo de desenvolvimento de software é instável, sujeito a mudanças radicais freqüentes, não apenas de um projeto a outro, mas também durante a realização de um mesmo projeto.

Neste nível, estimação de custos, prazos e qualidade do produto é algo totalmente fora do contexto e da política de desenvolvimento (que política?).

Embora não se possa "assegurar" o fracasso de um projeto desenvolvido por uma empresa situada neste nível, é possível dizer que o sucesso é, geralmente, resultado de esforços individuais, variando com as habilidades naturais, o conhecimento e as motivações dos profissionais envolvidos no projeto.

5.2.2. Nível Repetível

Neste nível, políticas de desenvolvimento de software e tarefas de suporte a estas políticas são estabelecidas, o planejamento de novos projetos sendo baseado na experiência obtida com projetos anteriores.

Para que uma empresa possa atingir este nível, é imprescindível institucionalizar o gerenciamento efetivo dos seus projetos de software, de modo que o sucesso de projetos anteriores possam ser repetidos nos projetos em curso.

Neste nível, os requisitos do software e o trabalho a ser feito para satisfazê-los são planejados e supervisionados ao longo da realização do projeto. São definidos padrões de projeto, e a instituição deve garantir a sua efetiva implementação.

A capabilidade de uma empresa situada neste nível pode ser caracterizada como disciplina, em razão dos esforços de gerenciamento e acompanhamento do projeto de software.

5.2.3. Nível Definido

No ***nível definido***, o processo de desenvolvimento de software é consolidado tanto do ponto de vista do gerenciamento quanto das tarefas de engenharia a realizar; isto é feito através de documentação, padronização e integração no contexto da organização, que adota esta versão para produzir e manter o software.

Os processos definidos nas organizações situadas neste nível são utilizados como referência para os gerentes de projeto e os membros do staff técnico, sendo baseado em práticas propostas pela Engenharia de Software.

Programas de treinamento são promovidos ao nível da organização, como forma de difundir e padronizar as práticas adotadas no processo definido.

As características particulares de cada projeto podem influir no aprimoramento de um processo de desenvolvimento, sendo que para cada projeto em desenvolvimento, este pode ser instanciado.

Um processo de desenvolvimento bem definido deve conter padrões, procedimentos para o desenvolvimento das atividades envolvidas, mecanismos de validação e critérios de avaliação.

A capabilidade de uma empresa no nível 3 é caracterizada pela padronização e consistência, uma vez que as políticas de gerenciamento e as práticas da Engenharia de Software são aplicadas de forma efetiva e repetida.

5.2.4. Nível Gerenciado

No ***nível gerenciado***, é realizada a coleta de medidas do processo e do produto obtido, o que vai permitir um controle sobre a produtividade (do processo) e a qualidade (do produto).

É definida uma base de dados para coletar e analisar os dados disponíveis dos projetos de software. Medidas consistentes e bem definidas são, então, uma característica das organizações situadas neste nível, as quais estabelecem uma referência para a avaliação dos processos de desenvolvimento e dos produtos.

Os processos de desenvolvimento exercem um alto controle sobre os produtos obtidos; as variações de desempenho do processo podem ser separadas das variações ocasionais (ruídos), principalmente no contexto de linhas de produção definidas. Os riscos relacionados ao aprendizado de novas tecnologias ou sobre um novo domínio de aplicação são conhecidos e gerenciados cuidadosamente.

A capabilidade de uma organização situada este nível é caracterizada pela previsibilidade, uma vez que os processos são medidos e operam em limites conhecidos.

5.2.5. Nível Otimizado

No ***nível otimizado***, a organização promove contínuos aperfeiçoamentos no processo de desenvolvimento, utilizando para isto uma realimentação quantitativa do processo e aplicando novas idéias e tecnologias. Os aperfeiçoamentos são definidos a partir da identificação dos pontos fracos e imperfeições do processo corrente e do estabelecimento das alterações necessárias para evitar a ocorrência de falhas. Análises de custo/benefício são efetuadas sobre o processo de desenvolvimento com base em dados extraídos de experiências passadas.

Quando os problemas relacionados à adoção de um dado processo de desenvolvimento não podem ser totalmente eliminados, os projetos são cuidadosamente acompanhados para evitar a ocorrência de problemas inerentes do processo.

5.3. Definição operacional do modelo CMM

O modelo CMM, além de definir os níveis de maturidade acima descritos, detalha, cada um deles, com respeito aos objetivos essenciais de cada um e das tarefas chave a serem implementadas para que estes objetivos sejam atingidos.

Para isto, foi realizado, à exceção do nível 1, um detalhamento nos diferentes níveis, estabelecendo uma estrutura que permitisse caracterizar a maturidade e a capabilidade do processo de desenvolvimento de software. A figura 2.2 ilustra os componentes relacionados a este detalhamento.

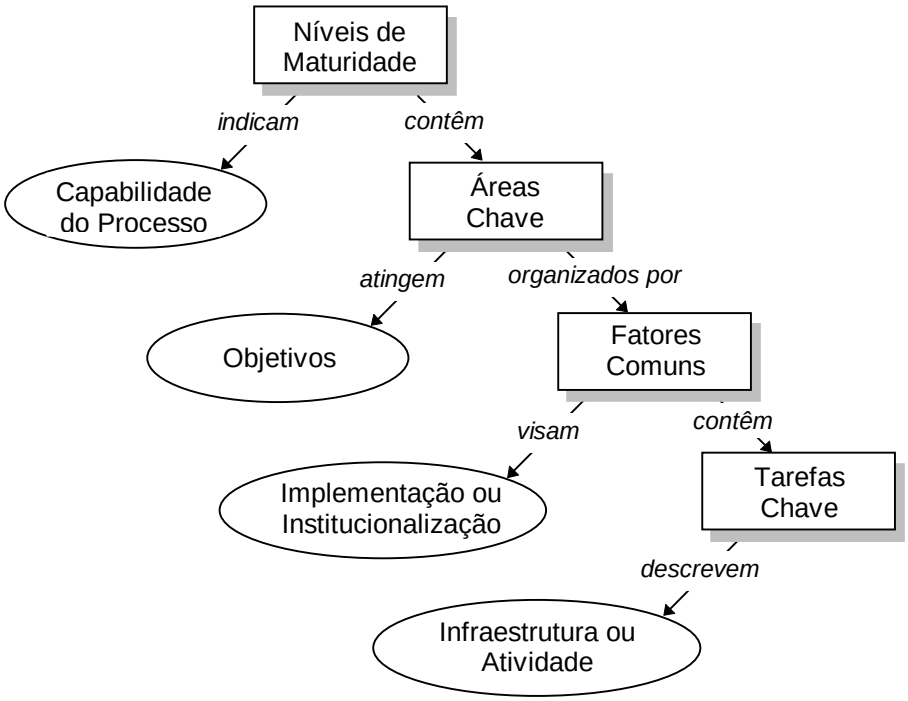


Figura 2.2 - Estrutura dos níveis CMM.

Como se pode notar na figura, cada nível de maturidade é composto por diversas **Áreas Chave**, as quais identificam um conjunto de atividades que, quando realizadas conjuntamente, permitem atingir os objetivos essenciais do nível considerado, aumentando a capabilidade do processo de desenvolvimento de software.

A Tabela 2.1 apresenta as áreas chave associadas a cada um dos níveis do CMM (à exceção do nível 1, como já foi explicado).

Os **Fatores Comuns** indicam o grau de implementação ou institucionalização de uma dada Área Chave. No modelo, os Fatores Comuns foram definidos em número de cinco, como mostra a Tabela 2.2.

As **Tarefas Chave** correspondem às atividades que, uma vez realizadas de modo conjunto, contribuirão para o alcance dos objetivos da área chave. As tarefas chave descrevem a infra-estrutura e as atividades que deverão ser implementadas para a efetiva implementação ou institucionalização da área chave. As tarefas chave correspondem a uma sentença simples, seguida por uma descrição detalhada a qual pode incluir exemplos.

A figura 2.3 apresenta um exemplo de estrutura de uma tarefa chave para a Área Chave de Planejamento do Projeto de Software.

5.4. Utilização do modelo CMM

O objetivo fundamental do estabelecimento deste modelo é fornecer parâmetros para:

- de um lado, que as instituições que pretendam contratar fornecedores de software possam avaliar as capacidades destes fornecedores;
- por outro lado, para que as empresas de desenvolvimento de software possam identificar quais os procedimentos a serem implementados ou institucionalizados no âmbito da organização de modo a aumentar a capabilidade do seu processo de software.

Nível	Áreas Chave
Repetível (2)	• *Gerenciamento de requisitos* • *Planejamento do processo de desenvolvimento* • *Acompanhamento do projeto* • *Gerenciamento de subcontratação* • *Garantia de qualidade* • *Gerenciamento de configuração*
Definido (3)	• *Ênfase ao processo na organização* • *Definição do processo na organização* • *Programa de treinamento* • *Gerenciamento integrado* • *Engenharia de software* • *Coordenação intergrupo* • *Atividades de revisão*
Gerenciado (4)	• *Gerenciamento da qualidade de software* • *Gerenciamento quantitativo do processo*
Otimizado (5)	• *Prevenção de falhas* • *Gerenciamento de mudança de tecnologia* • *Gerenciamento de mudança do processo*

Tabela 2.1 - Áreas Chave por nível de maturidade.

Fator Comum	Objetivos
Passível de realização	*Descreve as ações a realizar para definir e estabilizar um processo*
Capacidade de realização	*Define pré-condições necessárias no projeto ou organização para implementar o processo de modo competente*
Atividades realizadas	*Descreve os procedimentos necessários para implementar uma área chave*
Medidas e análises	*Indica a necessidade de realização de atividades de medição e análise das medidas*
Verificação da implementação	*Corresponde aos passos necessários para assegurar que todas as tarefas foram realizadas adequadamente*

Tabela 2.2 - Fatores comuns.

Embora os dois objetivos sejam diferentes, é definido no modelo um procedimento similar para a realização dos dois tipos de análise. As etapas deste procedimento, descritas a seguir, deverão ser realizadas num esquema seqüencial:

- seleção de uma equipe, cujos elementos serão treinados segundo os conceitos básicos do modelo CMM, sendo que estes já deverão ter conhecimentos de engenharia de software e gerenciamento de projetos;
- selecionar profissionais representantes da organização, para os quais será aplicado um questionário de maturidade;
- analisar as respostas obtidas do questionário de maturidade, procurando identificar as áreas chave;

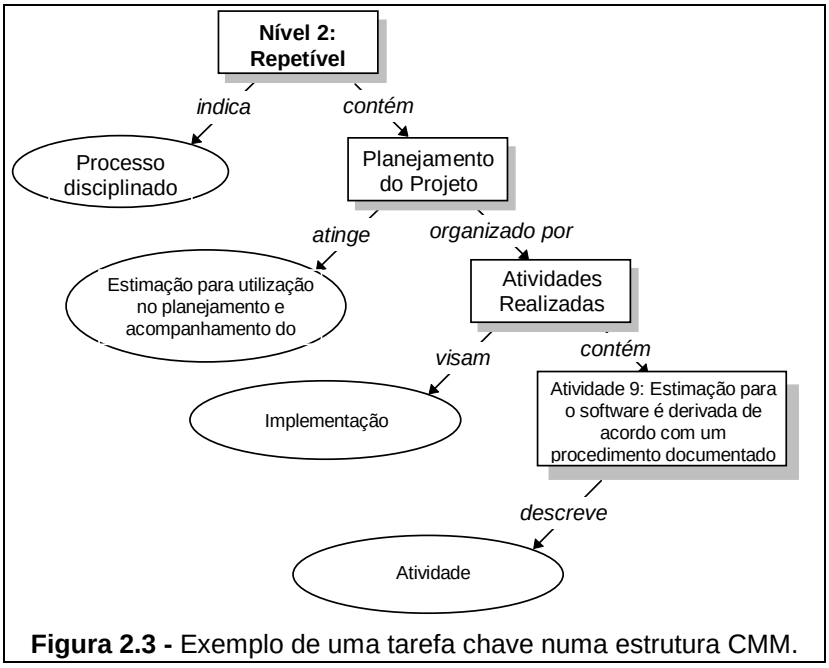


Figura 2.3 - Exemplo de uma tarefa chave numa estrutura CMM.

- realizar uma visita (da equipe) à organização para a realização de entrevistas e revisões de documentação relacionadas ao processo de desenvolvimento; as entrevistas e revisões deverão ser conduzidas pelas áreas chave do CMM e pela análise realizada sobre o questionário de maturidade;
- ao fim da visita, a equipe elabora um relatório enumerando os pontos fortes e os pontos fracos da organização em termos de processo de desenvolvimento de software;
- finalmente, a equipe prepara um perfil de áreas chave, que indica as áreas onde a organização atingiu/não atingiu os objetivos de cada área.

Engenharia de Software

Capítulo 3

Engenharia de Sistemas Computacionais

1. Introdução

Embora este curso tenha por objetivo a discussão dos aspectos mais importantes relacionados ao desenvolvimento do software, é importante reconhecer o fato óbvio de que o software não consiste de um elemento autônomo. Para que suas funções possam ser realizadas, ele deverá ser integrado a outros componentes, especialmente elementos de hardware, como processador, circuitos de memória e outros.

Por esta razão, o software deve, antes de tudo, ser visto como um elemento de um sistema mais amplo, ao qual denominaremos um **Sistema Computacional**. No contexto deste curso, um Sistema Computacional deverá ser entendido como o resultado da união de diferentes componentes, entre os quais se enquadram o software, o hardware e elementos de outras naturezas como por exemplo o elemento humano, representado pelo usuário do sistema.

A **Engenharia de Sistemas Computacionais** corresponde ao conjunto de atividades que deverão ser realizadas para definir o sistema computacional como um todo. Dentre as atividades a serem conduzidas, pode-se citar a especificação, o projeto, a implementação, a validação, a instalação e a manutenção.

Cada uma das disciplinas componentes da Engenharia de Sistemas está relacionada a uma tentativa de estabelecimento de uma ordem no desenvolvimento de sistemas computacionais.

Nestes sistemas, o software vem, há muitos anos, substituindo o hardware como o elemento de mais difícil concepção, com menor probabilidade de sucesso em termos de prazos e custo e de mais difícil administração. Além disso, a demanda por software cresce a cada dia, até como conseqüência do grande desenvolvimento do hardware dos sistemas computacionais.

O objetivo deste capítulo é discutir as principais definições e aspectos relacionados com a Engenharia de Sistemas Computacionais, procurando definir o papel da Engenharia de Software neste contexto.

2. Aspectos da Engenharia de Sistemas Computacionais

2.1. Sistema Computacional

Antes de discutir os principais pontos relativos à Engenharia de Sistemas Computacionais, é imprescindível estabelecer uma definição do que é um sistema computacional. Isto é particularmente importante, uma vez que a palavra "sistema" é uma das mais empregadas em qualquer área da atividade humana (sistema político, sistema educacional, sistema de avaliação, etc...).

Dentro do contexto do que será discutido aqui, pode-se definir um **Sistema Computacional** como sendo *um conjunto de elementos interrelacionados que operam juntos para o alcance de algum objetivo.*

Um sistema de processamento de texto, por exemplo, integra funções de entrada e edição de texto com funções de produção de documentos (software e hardware) manipuladas por um usuário (um elemento humano) para transformar um dado texto (uma entrada) num documento (a saída).

Pode-se listar os elementos de um sistema computacional como sendo:

- ***software***, no caso, programas de computador, estruturas de dados, e documentação associada que serve a efetivar os métodos, procedimentos ou processo de controle lógico;
- ***hardware***, que são os dispositivos eletrônicos que definem as capacidades de um computador e dispositivos eletromecânicos que oferecem funcionalidades ao ambiente externo;
- ***pessoal***, usuários/operadores do hardware e do software;
- ***bancos de dados***, uma coleção de informações organizada sistematicamente, acessada através do software;
- ***documentação***, manuais, formulários e outros documentos que podem auxiliar no conhecimento do uso e operação do sistema;
- ***procedimentos***, as regras que especificam o uso específico de cada elemento do sistema computacional.

As combinações destes elementos podem ser as mais diversas para permitir a transformação das informações. Um robô, por exemplo, é um sistema computacional que transforma um arquivo contendo instruções específicas num conjunto de movimentos e ações.

2.2. Composição dos Sistemas Computacionais e as Propriedades Emergentes

É importante reconhecer o papel das atividades relacionadas à Engenharia de Sistemas Computacionais como determinantes para as tarefas que se desenvolveram com base em seus resultados. Embora tenhamos dito que um Sistema Computacional é composto de diferentes elementos, não se deve negligenciar o fato de que a integração destes diferentes elementos é uma tarefa longe de ser trivial. O fato de se ter elementos concebidos com exemplar nível de qualidade não garante que a reunião destes vá originar um sistema excepcional.

Em tais sistemas, as dependências de um elemento do sistema com relação a outros podem se dar em muitos níveis. Por exemplo, o software só pode automatizar a solução de um problema se o processador estiver funcionando de forma adequada. Por outro lado, uma máquina de comando numérico só irá realizar corretamente o seu trabalho se o software de pilotagem tiver sido adequadamente concebido.

Entender o papel da Engenharia de Sistemas Computacionais é constatar que um sistema é algo mais do que a simples união de seus diversos elementos.

Num sistema de grande porte, existe um conjunto de propriedades que são verificados somente a partir do momento em que os diferentes elementos do mesmo são integrados. São as chamadas Propriedades Emergentes. Alguns exemplos destas propriedades são:

- o peso global do sistema, é um exemplo de propriedade que pode ser previamente determinada a partir de propriedades individuais dos componentes;
- a confiabilidade do sistema, que está fortemente dependente da confiabilidade de cada elemento;

- a facilidade de uso, que vai estar relacionada aos diferentes elementos do sistema como o hardware o software, mas também aos usuários do mesmo.

2.3. Sistemas e Subsistemas

Pode-se observar ainda que elementos de um determinado sistema podem operar de forma autônoma, constituindo-se num sistema independente. No entanto, ao serem incorporados ao sistema, suas funcionalidades vão estar dependendo de outros elementos deste. Uma câmera de vídeo, por exemplo, pode ser vista como um sistema independente. Por outro lado, quando ela está embutida num sistema de segurança automatizado, a maior parte de seus ajustes (enquadramento da imagem, foco, ativação e desativação) serão completamente dependentes de outros componentes do sistema de segurança.

Isto significa que, nos sistemas computacionais é comum um dado sistema compor um outro sistema computacional de nível hierárquico superior. Diz-se então que o sistema considerado é um **macroelemento** ou **subsistema** do sistema de nível superior.

Um exemplo típico desta situação é aquela dos equipamentos de manufatura, como por exemplo os *robôs*, *máquinas de comando numérico* e *controladores lógicos programáveis*, que são sistemas computacionais e ao mesmo tempo subsistemas de um sistema computacional de nível hierárquico superior: a *célula de manufatura*. A célula de manufatura pode ser definida como um sistema computacional, uma vez que esta é caracterizada por todos os elementos definidos acima. Se for feita uma análise considerando outros níveis hierárquicos, é possível visualizar a célula como um macroelemento de outro sistema computacional: o *sistema de manufatura*. A figura 3.1 ilustra esta relação entre os diferentes sistemas computacionais descritos.

2.4. Hierarquia e ambiente de um sistema

Reconhecido o fato de que um sistema pode ser obtido a partir de outros sistemas, como o exemplo ilustrado à figura 3.1, surge então o conceito de hierarquia de sistemas, que vai permitir utilizar a definição de subsistema, introduzida anteriormente.

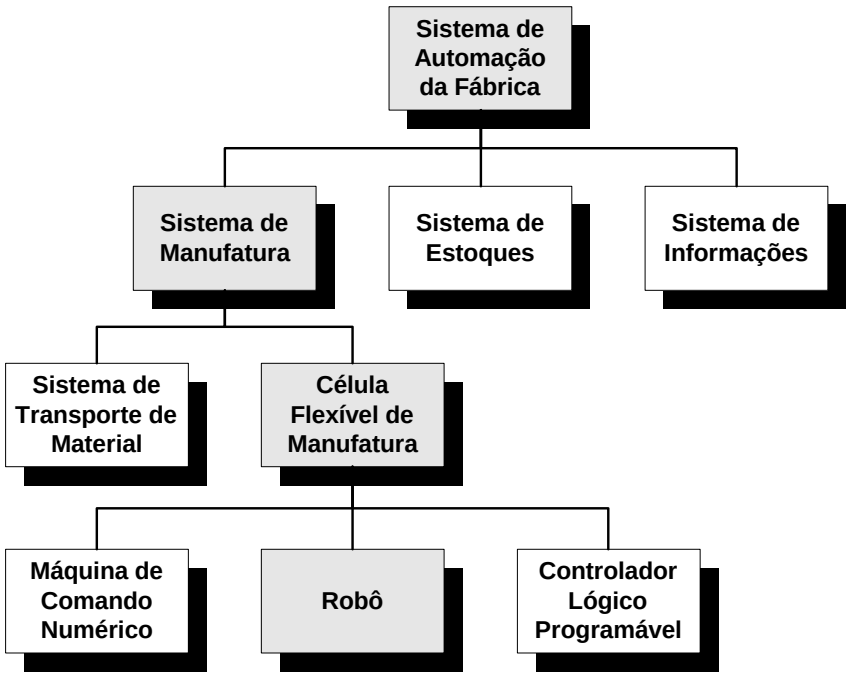


Figura 3.1 - Relação hierárquica entre sistemas computacionais.

Por esta razão, o ambiente de um sistema deve ser cuidadosamente estudado nesta etapa de concepção. A importância do entendimento completo do ambiente de um sistema é justificada pelos fatores seguintes:

- em grande parte das vezes, o sistema é concebido para provocar alterações no ambiente; sendo assim, a compreensão dos principais aspectos de funcionamento do ambiente serão fundamentais para especificar de que forma o sistema deve agir para provocar as alterações de forma precisa; exemplos de tais sistemas são um sistema de iluminação, um sistema de aquecimento (ou refrigeração) de ambiente;
- mesmo quando a função do sistema não está relacionada a alterações de seu ambiente, é importante se conhecer o quanto o ambiente pode influenciar no funcionamento do sistema; por exemplo, numa rede de computadores a ser instalada numa unidade fabricação de uma empresa, o fato de naquela unidade existir um grande número de motores e inversores operando pode ser determinante para decidir que tipos de protocolos e meios de transmissão deverão ser empregados, devido à alta incidência de fontes de interferência que podem causar erros de comunicação.

Um outro aspecto a ser considerado é o conjunto de requisitos originado no próprio ambiente. Os exemplos deste aspecto são os mais diversos, mas pode-se citar normas de qualidade ou segurança de produtos, requisitos de interface numa dada plataforma, requisitos de vedação, para um sistema computacional que irá operar num ambiente úmido ou sujeito a poeira.

Por todas estas razões, é fundamental que um sistema seja visto sempre como um componente de um sistema maior, o estudo aprofundado das características mais importantes relativas ao ambiente e dos requisitos impostos por este sendo uma tarefa essencial.

2.5. A Aquisição de Sistemas

No desenvolvimento de sistemas, principalmente daqueles mais complexos, é comum que determinados componentes sejam adquiridos em lugar de serem completamente desenvolvidos pela equipe ou empresa envolvida no projeto. Em alguns casos, o próprio sistema pode ser completamente adquirido num fabricante específico e, em outros, partes do sistema serão obtidas de um mesmo ou de diferentes fornecedores. Por outro lado, pode haver sistemas onde todos os componentes podem ser completamente desenvolvidos pela equipe.

O processo de decisão sobre a compra ou o desenvolvimento de um sistema ou de parte dele é uma tarefa que requer um esforço e um tempo considerável, principalmente se o grau de complexidade do mesmo for elevado. Uma decisão baseada numa análise superficial pode conduzir ao fracasso do projeto.

Normalmente, para servir de referência à decisão, uma especificação do sistema e um projeto arquitetural devem ser conduzidos. Os resultados obtidos nesta tarefa são importantes pelas seguintes razões:

- para que o processo de aquisição seja encaminhado com sucesso, é necessária a existência de uma especificação e de uma visão arquitetural do sistema;
- o custo de aquisição de um sistema é, na maioria dos casos, menor que do desenvolvimento deste; por esta razão, o projeto arquitetural é importante como forma de decidir quais subsistemas serão adquiridos e quais serão desenvolvidos.

2.6. A Subcontratação

As situações em que uma empresa desenvolve completamente todos os componentes de um sistema são extremamente raras. Geralmente, a organização "usuária" vai encomendar o sistema a uma organização "fornecedora". A organização fornecedora, eventualmente, vai "terceirizar" o desenvolvimento do sistema, contratando outras empresas fornecedoras dos diferentes subsistemas. A figura 3.2 ilustra este processo.

Um aspecto interessante deste processo é a otimização, no que diz respeito aos contatos, entre a organização do usuário e as organizações desenvolvedoras. Na realidade, o único contato mantido pela organização usuária é com a principal contratante. As subcontratantes vão projetar e construir os subsistemas cujos requisitos foram especificados pela principal contratante que exerce o papel de aglutinadora de todas as empresas envolvidas no desenvolvimento.

2.7. A Importância do Software

No desenvolvimento de sistemas computacionais, é comum a utilização de componentes customizados (construídos sob medida) e componentes de "prateleira". Por esta razão, o componente de software assume um papel bastante importante no desenvolvimento do sistema, isto porque é este que pode assumir a função de elo de ligação entre os diferentes componentes, permitindo a adequação do funcionamento dos diferentes componentes de hardware (principalmente aqueles de "prateleira") aos requisitos do sistema.

Um exemplo deste fato está num dos sistemas computacionais mais utilizados nos dias atuais — os microcomputadores da linha PC. O PC da IBM, quando foi concebido, foi totalmente construído a partir de componentes de hardware disponíveis no mercado. Esta decisão foi tomada devido à urgência em ocupar o espaço do mercado no setor de computadores pessoais. Entretanto, a fórmula mágica da IBM para fazer com que todos estes elementos funcionassem harmonicamente foi buscada no software, no caso o ROM-BIOS, que continha todas as rotinas de programação e acesso aos dispositivos de hardware que compunham a arquitetura do PC.

Só a partir do momento em que as outras empresas conseguiram copiar o conteúdo do ROM-BIOS é que surgiram os conhecidos "clones" do PC que difundiram a tecnologia pelo mundo todo.

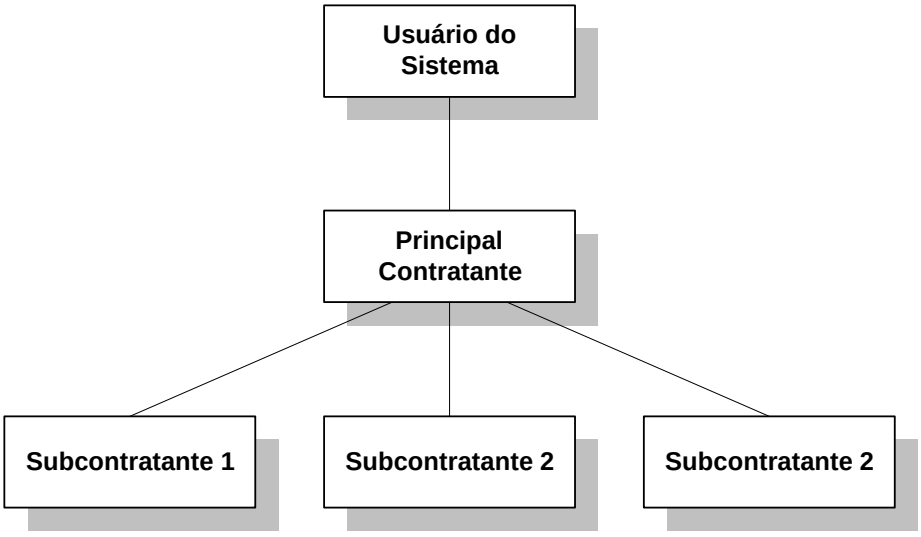


Figura 3.2 - Ilustração do processo de aquisição de um sistema.

3. ENGENHARIA DE SISTEMAS COMPUTACIONAIS

O papel essencial da Engenharia de Sistemas Computacionais é a resolução de problemas, onde a atividade de base é a identificação, análise e atribuição aos elementos constituintes do sistema das funções desejadas para o sistema.

A palavra que melhor define a Engenharia de Sistemas é a **interdisciplinaridade**, uma vez que, na maior parte dos sistemas a serem concebidos, devem entrar em jogo diferentes competências. Além dos Engenheiros de Hardware e de Software, devem intervir nesta etapa especialistas como Engenheiros Eletricistas, Engenheiros Hidráulicos, Engenheiros de Telecomunicações, Químicos, Físicos, Psicólogos, Médicos, etc.

O ponto de partida do trabalho do Engenheiro de Sistemas são os requisitos e restrições formuladas pelo cliente, que vão permitir um primeiro delineamento das funções a serem executadas pelo sistema, de requisitos de desempenho, das interfaces a serem oferecidas, das restrições de projeto e das estruturas de informação que serão processadas por cada um dos elementos constituintes do sistema (software, hardware, pessoal, etc...).

Neste trabalho, é importante que se estabeleça da forma mais precisa possível as funções a serem realizadas. Por exemplo, no caso do sistema de controle de um robô, não é suficiente dizer que *"ele deve reagir com rapidez se a bandeja de ferramentas estiver vazia"*... na realidade, é importante especificar cuidadosamente os diferentes aspectos deste requisito:

- o que vai indicar a bandeja vazia para o robô;
- os limites de tempo relacionados com a reação;
- de que forma deve ser esta reação.

Uma vez identificados e analizadas todas as funções, requisitos e restrições, passa-se à fase de **alocação**, onde o objetivo é atribuir, aos diferentes elementos que constituem o sistema, as funções analizadas anteriormente.

Nesta tarefa, é comum estabelecer alternativas de alocação para posterior definição. Consideremos, por exemplo, um sistema de tratamento gráfico, onde uma das funções principais é a transformação tridimensional de imagens. Deste exemplo, é possível imaginar algumas opções de soluções existentes:

① todas as transformações serão resolvidas por software;
② as transformações mais simples (redimensionamento e translação) serão realizadas em hardware e as mais complexas (rotação, perspectivas, etc...) em software;
③ todas as transformações serão realizadas por um processador geométrico implementado em hardware.

A escolha da solução deve ser baseada num conjunto de critérios, onde entram fatores como custo, realizabilidade, desempenho, padronização de interfaces, portabilidade, etc.

Nas seções a seguir, serão descritas atividades encaminhadas no contexto da Engenharia de Sistemas Computacionais.

3.1. Definição dos Requisitos

O objetivo das tarefas a serem desenvolvidas nesta etapa é identificar os requisitos do Sistema Computacional a ser concebido. Considerando que grande parte dos sistemas computacionais são construídos sob demanda, a forma usual de encaminhar esta etapa é caracterizada por um conjunto de reuniões estabelecidas entre a equipe de desenvolvimento do sistema e os clientes ou usuários.

Nesta etapa, pode-se considerar basicamente três categorias de requisitos:

- os **requisitos básicos**, que estão associados às funções a serem desempenhadas pelo sistema; neste momento do desenvolvimento, estes requisitos são definidos num nível de abstração relativamente elevado, uma vez que o detalhamento será necessariamente realizado numa etapa posterior;

- as **propriedades do sistema**, que estão relacionadas às propriedades emergentes do sistema; desempenho, segurança, disponibilidade, são alguns exemplos desta categoria de requisitos;

- as **características indesejáveis**, que são propriedades que o sistema não deve possuir; é tão importante, muitas vezes, definir que características não devem ser encontradas num dado sistema, quanto definir aquelas que são consideradas essenciais; num sistema de comunicação de dados, por exemplo, algumas características indesejáveis são os erros, as perdas e as duplicações de mensagens.

De forma geral, é importante que sejam estabelecidos, da forma mais completa possível, os objetivos e as funções a serem providas pelo sistema. Neste nível, deve-se enfatizar o papel do sistema computacional no ambiente em que ele será instalado, podendo-se deixar para etapas mais à frente, a especificação dos aspectos funcionais do mesmo.

Observemos a distinção através de um exemplo, onde são apresentadas duas versões de especificação dos requisitos de um sistema de segurança para um edifício comercial:

(a) O objetivo do sistema é prover um sistema de proteção contra incêndio e intrusão que vai ativar um alarme interno e externo em caso de princípio de incêndio ou invasão por pessoas não autorizadas;

(b) O objetivo do sistema é assegurar a continuidade do funcionamento normal conduzido no edifício, mesmo na ocorrência de eventos como o princípio de incêndio ou intrusão.

Neste nível do desenvolvimento do sistema, a segunda versão apresentada é a mais adequada, uma vez que ela é, por um lado, mais abrangente e, por outro, limitante em alguns aspectos. A forma como está estabelecido o objetivo do sistema está mais compatível com as necessidades do ambiente, e dá uma maior abertura à adoção de técnicas mais evoluídas de prevenção de incêndio e intrusão. Por outro lado, o fato de estabelecer como requisitos a garantia de continuidade das atividades realizadas no escritório, elimina algumas soluções como o uso de alarme interno ou irrigadores contra incêndio.

3.2. O projeto do sistema

Os principais passos envolvidos no projeto de um sistema computacional estão esquematizados na figura 3.3 e serão descritos nos parágrafos que seguem.

Particionamento dos Requisitos. Uma vez estabelecidos os requisitos do sistema na etapa anterior, o objetivo deste primeiro passo de projeto é particioná-los segundo algum critério lógico. A escolha do critério de particionamento é importante pois deste vai depender as diferentes categorias de requisitos que serão criadas.

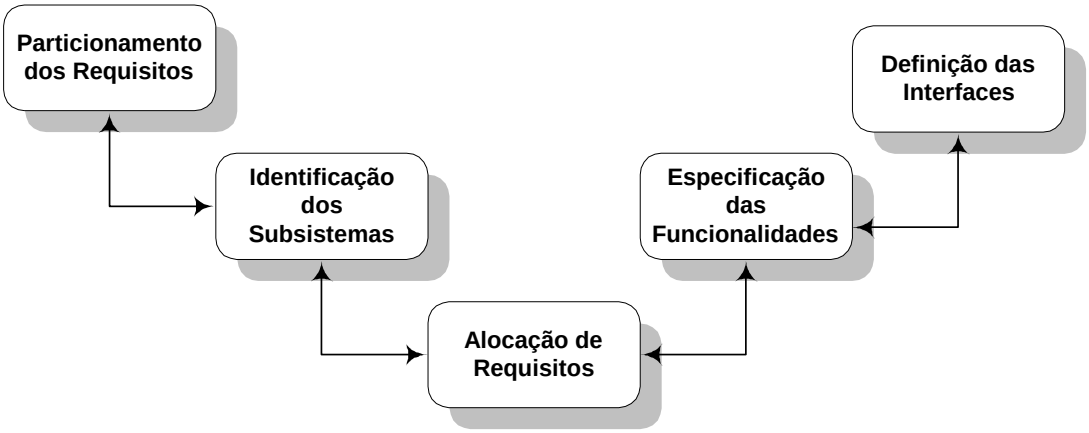


Figura 3.3 - Passos a serem conduzidos no projeto de um sistema computacional.

Identificação dos Subsistemas. Este passo busca definir os diferentes subsistemas que irão compor o sistema computacional. O particionamento realizado no passo anterior será determinante para a obtenção dos subsistemas, uma vez que, dependendo do critério utilizado, cada grupo de requisitos poderá definir um subsistema. Por outro lado, outros aspectos podem ser considerados para a definição dos subsistemas, particularmente aqueles relacionados a questões do ambiente onde o sistema será instalado.

Alocação dos Requisitos. O próximo passo a ser conduzido é a alocação dos requisitos aos subsistemas estabelecidos no passo anterior. Pode parecer estranho considerar tal etapa, quando se utiliza o resultado obtido no Particionamento dos Requisitos para realizar a Identificação dos Subsistemas, mas, na verdade, isto não ocorre de forma tão imediata. Por exemplo, no caso da adoção de sistemas de prateleira para implementar algum subsistema, as limitações do sistema adotado podem impor a realocação de requisitos.

Especificação das Funcionalidades. Neste passo, serão definidas as funções que serão implementadas por cada subsistema. No caso de um subsistema de software, esta atividade pode ser encaminhada como parte da análise de requisitos do mesmo.

Definição das Interfaces dos Subsistemas. Este é um passo de extrema importância na condução do projeto do sistema. A definição, de forma precisa das interfaces entre os subsistemas é fundamental para as etapas posteriores de integração dos diferentes componentes. Não se deve negligenciar o fato de que muitos subsistemas podem ser adquiridos em lugar de serem desenvolvidos e mesmo aqueles que são especialmente construídos no contexto de um projeto, usualmente são desenvolvidos por diferentes equipes ou por diferentes empresas. Qualquer má definição neste item pode comprometer o sucesso do projeto ou implicar num aumento de custo ou de prazo de desenvolvimento.

As linhas contendo setas nas duas direções entre cada par de passos sugere a necessidade de revisões e realimentações em cada passo. Na maioria dos projetos, o número de soluções possíveis é bastante grande, podendo-se chegar a diferentes visões de como as funções são alocadas a subsistemas de hardware, de software ou aos elementos humanos que vão interagir com o sistema.

A escolha da solução a ser encaminhada vai ser feita em função de critérios técnicos, mas também terá influência de critérios políticos e organizacionais da instituição. No caso de projetos desenvolvidos para instituições militares, por exemplo, fornecedores nacionais serão escolhidos em lugar de fornecedores estrangeiros, mesmo que em termos técnicos, a solução oferecida não seja a ideal.

3.3. Desenvolvimento dos Subsistemas

Esta etapa agrupa as atividades relacionadas ao desenvolvimento de cada subsistema definido no projeto. Cada subsistema terá suas funções bem definidas e suas especificidades quanto às competências necessárias ao seu desenvolvimento. No caso de um subsistema de software, um processo de desenvolvimento caracterizado pelas etapas clássicas de desenvolvimento será iniciado.

Em raras situações, o desenvolvimento de um sistema impõe a construção, a partir do zero, de todos os subsistemas que o compõem. O mais comum é que alguns subsistemas sejam adquiridos e incorporados ao sistema. Embora seja o caso mais comum, nem sempre é fácil integrar subsistemas "de prateleira" a um sistema; em muitos casos, são necessárias modificações no sentido de adaptar o subsistema adquirido às necessidades do sistema a ser desenvolvido. Além disso, nem sempre é possível prever a disponibilidade de um subsistema de prateleira no momento em que ele é necessário.

A **Engenharia de Hardware**, para efeito de definição de um dado sistema, consiste no trabalho de seleção dos componentes que vão compor o hardware do sistema e que serão capazes de assumir as funções atribuídas a este elemento. Atualmente, o trabalho de seleção dos componentes de hardware é mais simples do que o de software, sendo facilitado pelos seguintes fatores:

- os componentes são montados como blocos de construção individuais;
- as interfaces entre os componentes são padronizadas;
- existem numerosas alternativas "de prateleira" à disposição;
- aspectos como o desempenho, custo e disponibilidade são de fácil obtenção.

A Engenharia de Hardware é baseada na execução de três principais fases:

- o **Planejamento e a Especificação**, cujo objetivo é estabelecer a dimensão do esforço necessário ao desenvolvimento, assim como o estabelecimento de um roteiro para o projeto e implementação do hardware; ainda nesta fase, é conduzida a análise de requisitos, cujo resultado deve ser uma especificação a mais completa possível das funções e outros requisitos para o hardware e das suas restrições;

- o **Projeto e Prototipação**, a qual consiste na fase que, com base nos requisitos e restrições especificadas na fase anterior, define uma configuração preliminar do hardware; atualmente, as tarefas relativas a esta fase são semi ou completamente automatizadas com o auxílio de ferramentas de software CAE/CAD; o resultado final desta fase é um protótipo montado que será testado para garantir a satisfação de todos os requisitos;

- a **Produção, Distribuição e Instalação**, onde o protótipo obtido na fase anterior vai sofrer as evoluções para tornar-se verdadeiramente um produto de hardware; alterações na embalagem, interfaces, componentes, etc..., são então realizadas; definição de métodos de controle de qualidade é um ponto fundamental nesta fase; a criação de um estoque de peças de reposição é uma outra preocupação conduzida neste momento, assim como a definição de esquemas de instalação e manutenção.

Um dos resultados da Engenharia de Sistemas é a definição de aspectos como funcionalidade e desempenho do software. O trabalho essencial do engenheiro de software é acomodar os requisitos de funcionalidade e desempenho da forma mais eficiente possível, tendo que, para isto, adquirir e/ou desenvolver os componentes de software. A dificuldade que surge na Engenharia de Software é a falta de padronização nos componentes de

software (o que não ocorre no caso do hardware), uma vez que, na maior parte dos projetos, estes componentes são "customizados" para atender às exigências de software do sistema a ser desenvolvido.

Como já foi discutido anteriormente, a Engenharia de Software envolve as seguintes fases:

- **Definição**, que é iniciada com o Planejamento do Desenvolvimento do Software, englobando todas as tarefas discutidas no capítulo III do curso, obtendo um documento de Plano do Software, o qual será produzido e revisado pelo Gerente do Projeto; ainda nesta fase é realizada a Análise de Requisitos do Softwa, a qual vai permitir que funções, dentro do sistema como um todo serão atribuídas ao software; a última tarefa relacionada a esta fase é a revisão da Especificação de Requisitos do Software, o qual é o documento resultante da Análise de Requisitos;

- **Desenvolvimento do Software**, que é a fase que tem, como ponto de partida, os documentos produzidos na fase anterior, particularmente, o Plano do Software e a Especificação de Requisitos do Software; com base nestes documentos, inicia-se a etapa de Projeto do Software, onde serão descritos aspectos relacionados ao funcionamento do software como a sua arquitetura e as estruturas de dados; após avaliada esta definição, inicia-se a etapa de Projeto Detalhado, onde os aspectos algorítmicos e comportamentais do software são definidos; finalmente, a etapa de codificação é encaminhada, seja com base no uso de uma linguagem de programação clássica ou com o auxílio de uma ferramenta CASE, o resultado desta etapa sendo a listagem dos programas-fonte do software em desenvolvimento;

- **Verificação, Entrega e Manutenção**, é a última fase do processo, a qual envolve as atividades de teste do software, preparando-o para a entrega; uma vez entregue, inicia-se, ao longo de toda a vida útil do software a etapa de manutenção, a qual permitirá manter o software em funcionamento a partir da correção de novos erros que tenham sido detectados com o software em funcionamento, da introdução ou melhorias de funções do software, da adaptação do software para novas plataformas de hardware existentes.

Por razões óbvias, os diferentes subsistemas definidos na etapa de projeto são desenvolvidos em paralelo. No caso de ocorrência de algum problema relacionado à interface dos diferentes subsistemas, uma solicitação de modificação no sistema pode ser necessária. Em sistemas envolvendo a utilização de muitos subsistemas de hardware, a realização de modificações após o término da implementação podem representar um aumento considerável no custo de desenvolvimento. Uma forma bastante utilizada nos projetos, neste caso, é utilizar os subsistemas de software para facilitar estas modificações, devido, particularmente, à flexibilidade inerente dos sistemas de software. É lógico que, para que as modificações nos subsistemas de software não representem custos tão elevados ou maiores quanto o dos subsistemas de hardware, é necessário que estes subsistemas tenham sido concebidos tendo como princípio sua manutenibilidade.

3.4. Integração do Sistema

O conjunto de atividades a ser desenvolvido nesta etapa é o de conexão dos diferentes subsistemas construídos ou adquiridos para compor o sistema. É uma atividade bastante complexa, devido principalmente, à grande diversidade de tecnologias envolvidas na concepção dos diferentes sistemas.

Um problema comumente encontrado nesta etapa é o mal funcionamento de um subsistema como conseqüência de uma definição imprecisa de funcionalidade de outro subsistema.

Uma forma de realizar a integração de um sistema é o processo "big-bang", onde todos os seus subsistemas são conectados num passo único. Entretanto, por razões técnicas e gerenciais, a forma mais adequada é a integração incremental dos diferentes subsistemas. As principais razões para adoção desta estratégia são:

1) *Na maior parte dos casos, é impossível sincronizar o fim do desenvolvimento de todos os subsistemas;*

2) *A integração incremental reduz os custos de identificação e correção de erros de integração. Quando um número muito grande de subsistemas são integrados simultaneamente, uma falha no sistema pode ser conseqüência de um ou mais erros presentes nos diferentes subsistemas. No caso da integração incremental, a ocorrência de uma falha pode ser mais facilmente controlada, pois ela será, muito provavelmente, conseqüência de um erro no subsistema mais recentemente integrado.*

No caso de sistemas em que os subsistemas foram construídos por diferentes fornecedores, é muito comum ocorrer desentendimentos quando um problema de integração acontece. Os fornecedores tendem a acusar um ao outro como o responsável do problema detectado. Este tipo de ocorrência é bastante prejudicial ao desenvolvimento do sistema, podendo tomar muito tempo para que o problema seja resolvido.

3.5. Instalação do Sistema

Esta etapa envolve todas as atividades relacionadas à colocação do sistema em funcionamento no ambiente para o qual ele foi concebido. Embora possa parecer um problema menor, a instalação de um sistema pode ser caracterizada pela ocorrência de muitos problemas que coloquem em risco o cumprimento de estimativas de prazo ou de custo.

Alguns problemas típicos desta etapa são:

- O ambiente para o qual o sistema foi concebido não é precisamente o mesmo que foi especificado no início do projeto; este é um problema bastante comum em sistemas de software, particularmente no que diz respeito à utilização de certas facilidades do sistema operacional; quando a versão do sistema operacional do ambiente não é a mesma que foi utilizada para o desenvolvimento do sistema, o resultado pode ser catastrófico;

- A resistência dos usuários à implantação do novo sistema é um outro problema importante; nem sempre a adoção de um sistema computacional para a realização de uma tarefa que vinha sendo realizada de outra forma é bem vista pelos funcionários, pois este vai, na quase totalidade dos casos, na necessidade de treinamento ou de alteração na sistemática de trabalho; um exemplo bastante óbvio é o da utilização de sistemas de software para a declaração de imposto de renda; quando os primeiros programas foram lançados, muitas pessoas que tinham computadores ainda insistiam em utilizar os formulários impressos para realizar suas declarações;

- A necessidade de coexistência entre um sistema antigo e o novo sistema é outra fonte de problemas de instalação. No caso dos sistemas compartilharem algum recurso, pode ser impossível instalar completamente o novo sistema sem desativar o antigo;

- Obstáculos físicos à instalação do sistema são problemas muito comuns nesta etapa; ausência de pontos de energia ou tubulação apropriada, falta de sistema

de ar-condicionado, são exemplos de obstáculos comumente encontrados na instalação de sistemas computacionais.

3.6. Ativação do Sistema

Uma vez instalado, o sistema deverá ser ativado. Uma etapa de apresentação e treinamento relativo à utilização pode estar associada à ativação do sistema. Os problemas que podem ocorrer nesta etapa são os mais diversos. No caso de sistemas que deverão operar conjuntamente a outros sistemas existentes, problemas de incompatibilidade podem ocorrer e a dificuldade de resolução destes problemas pode provocar grandes atrasos à sua colocação em operação.

Outra fonte de problemas pode ser a inadequação ou a má utilização das interfaces de operador do novo sistema. Muitos erros de operação poderão ser cometidos até que o operador crie intimidade com o novo sistema.

3.7. Evolução do Sistema

Os grandes sistemas computacionais são desenvolvidos para funcionar durante longos períodos tempo. É inevitável que, durante este período, ocorram problemas que imponham modificações no sistema. As razões que podem conduzir a esta necessidade podem ser erros do próprio sistema, novas necessidades em termos de função ou alterações ambientais.

As evoluções a serem promovidas num dado sistema podem representar um custo bastante alto, o que é justificado pelas razões abaixo:

- Necessidade de estudos relativos às modificações a serem feitas para não comprometer o bom funcionamento do sistema;
- O alto grau de dependência dos diversos subsistemas pode conduzir à necessidade de alterações em muitos subsistemas, como conseqüência da alteração de um componente;
- A falta de documentação relativa ao desenvolvimento do sistema original vai ser, sem dúvida, um grande obstáculo para o desenvolvimento do sistema;
- Á medida que o tempo passa e que alterações vão sendo realizadas, a estrutura original dos sistemas vai sendo modificada, de modo que futuras mudanças podem ser fortemente comprometidas.

3.8. Desativação do Sistema

Esta etapa consiste em retirar o sistema de operação, quando seu tempo previsto de funcionamento esgota ou quando ele será substituído por um novo sistema. A desativação deve ser feita de forma bastante rigorosa, principalmente no caso de sistemas cujos componentes podem ser nocivos ao ambiente. Sistemas que utilizam componentes radioativos são os exemplos mais comuns de sistemas cuja desativação deve ser feita com o maior cuidado possível.

Por outro lado, no caso dos sistemas de software, a desativação pode ser extremamente simples. A maior parte dos sistemas de software construídos atualmente já vem dotada de um utilitário de desativação (ou "desinstalação") que retira automaticamente todos os componentes do sistema e as referências a estes componentes do sistema computacional.

Outro aspecto importante da desativação dos sistemas são as informações que este manipulava ou gerava. Esta, normalmente, deverão ser armazenadas em mídia específica para posterior adaptação à utilização num novo sistema.

4. PROJETO ARQUITETURAL

4.1. O MODELO DO SISTEMA

Um importante resultado da especificação de requisitos e do projeto de um sistema computacional é a sua organização em termos de subsistemas e dos relacionamentos entre estes. A forma mais usual de representar esta organização é, sem dúvida, os modelos baseados em linguagem gráfica.

Invariavelmente, a utilização de diagramas em blocos destacando os principais subsistemas e como estes estão relacionados é bastante aceita nos projetos de sistemas em geral, incluindo os sistemas computacionais. A figura 3.4 ilustra tal mecanismo, através do exemplo de projeto arquitetural de um sistema de controle de tráfego aéreo. O objetivo aqui não é descrever o funcionamento de um tal sistema, mas mostrar como se pode representar os diferentes componentes deste e como estes relacionam-se. As setas ligando os diferentes subsistemas indicam os fluxos de informação que existem no sistema.

No caso de sistemas relativamente complexos, é comum que os subsistemas definidos neste primeiro nível sejam, por si só, sistemas complexos. Isto significa que, para se ter uma idéia mais completa de como o sistema deve operar, os subsistemas podem, eventualmente, ser representados utilizando a mesma linguagem.

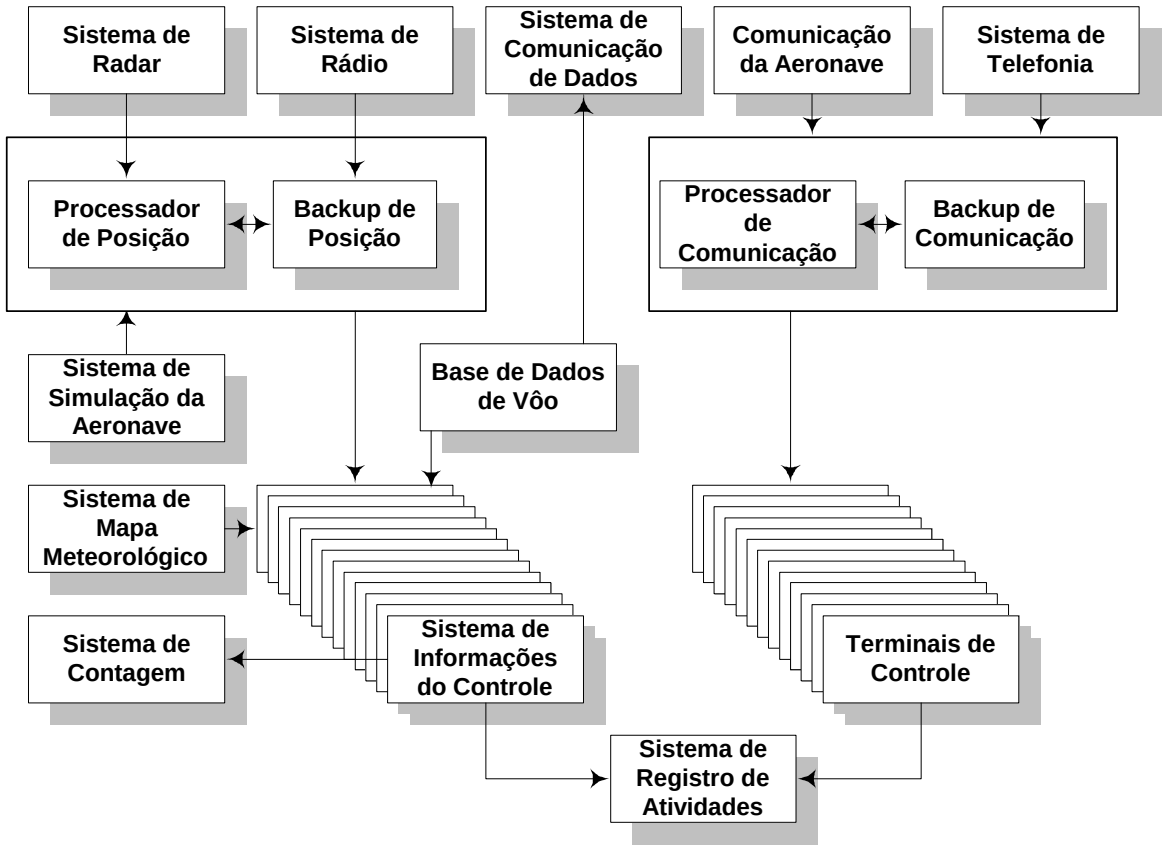


Figura 3.4 - Exemplo de projeto arquitetural: um sistema de controle de tráfego aéreo.

Num sistema computacional, os diagramas de arquitetura destacam, na forma de subsistemas, os elementos que deverão ser desenvolvidos em hardware e em software, permitindo o encaminhamento das atividades relativas às diferentes engenharias, normalmente conduzidas em paralelo.

Na maior parte dos sistemas atuais, é comum que os componentes de hardware incluam um ou mais computadores e o software associado. Isto se justifica pela queda constante nos preços de equipamentos computacionais, permitindo a substituição de outras alternativas pelos sistemas computacionais, além das vantagens de tais equipamentos, particularmente para tarefas de controle e supervisão.

No nível de projeto arquitetural, nem sempre é importante definir que funções serão executadas por software ou por hardware. O mais importante é definir subsistemas com base nas funções a serem executadas pelo sistema. A definição sobre a forma como cada função será implementada (em software ou em hardware) pode ser feita numa etapa posterior, levando em conta não apenas fatores técnicos. Por exemplo, a existência ou não de um componente de hardware que implemente adequadamente uma dada função pode ser determinante para definir se esta função deverá ser implementada em hardware ou software.

4.2. Componentes funcionais

Durante a realização das diferentes etapas que caracterizam o desenvolvimento de um sistema computacional, particularmente na etapa de definição dos subsistemas, diferentes categorias de componentes podem ser identificadas. Abaixo serão descritas aquelas mais freqüentemente encontradas nos sistemas:

Sensores. São os elementos capazes de coletar informações a respeito do ambiente onde o sistema está instalado. Exemplos deste tipo de componente são os radares do sistema de controle de tráfego aéreo ou os sensores de posicionamento de papel numa impressora a laser ou jato de tinta.

Atuadores. Estes componentes tem por função realizar ações no sistema ou em seu ambiente. Válvulas para abertura ou fechamento de fluxo hidráulico, os flaps das asas do avião, o mecanismo de alimentação de papel numa impressora, são exemplos desta classe de componentes.

Componentes computacionais. São os elementos capazes de exercer algum processamento de uma dada informação de entrada para gerar um conjunto de informações de saída. Alguns exemplos são processadores gráficos, processadores aritméticos, um elemento que implemente um algoritmo de controle, etc.

Componentes de comunicação. São os componentes que irão viabilizar a comunicação entre partes de um sistema ou entre o sistema e seu ambiente. Uma placa Ethernet, uma porta serial, um modem, são instâncias de tal classe.

Componentes de Coordenação. São os componentes que executam as funções de coordenação do sistema. Um algoritmo de escalonamento num sistema tempo-real é um bom exemplo deste tipo de componente.

Componentes de interface. São os componentes capazes de adaptar uma forma de representação de informação na forma de representação utilizada pelo componente que vai processá-la. Exemplos deste tipo de componente são uma placa de vídeo, um conversor analógico-digital, etc.

A figura 3.5 mostra, através de um sistema de segurança doméstica, como podem ser classificados os diferentes componentes, o que é explicitado na tabela a seguir.

Classe	Componente	Função
Sensor	Sensor de movimento e da porta	Detectar intrusão

Atuador	Sirene	Gerar um sinal de alarme
Comunicação	Ativador de chamada telefônica	Realizar chamada a um sistema externo
Coordenação	Controlador do alarme	Coordenar os demais componentes
Interface	Sintetizador de voz	Gerar mensagem de localização

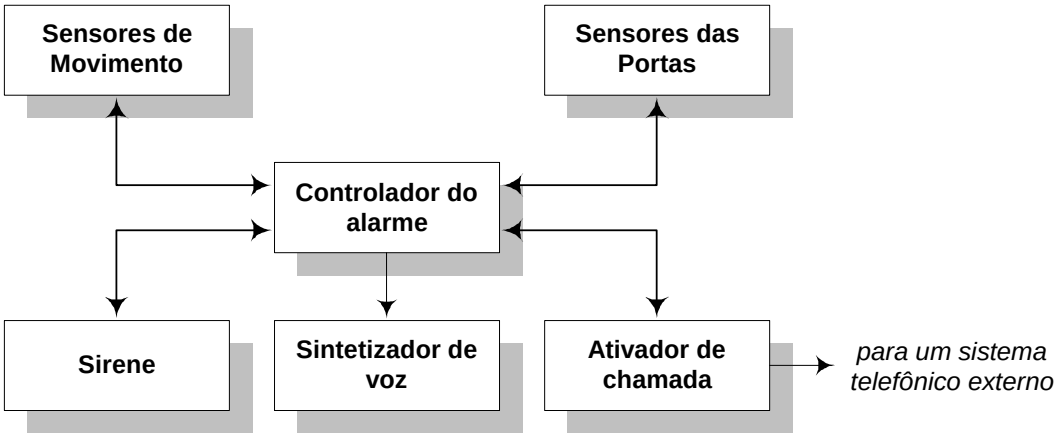


Figura 3.5 - Sistema de segurança doméstica.

Engenharia de Software

Capítulo 4

Planejamento do Desenvolvimento de Software

1. Introdução

Na maior parte dos trabalhos de engenharia (e os trabalhos de Engenharia de Software não fazem exceção), o tempo é um fator preponderante. Isto é conseqüência de uma questão cultural (todos nós aprendemos a trabalhar sob o efeito da pressão de tempo), resultante de uma "pressa" nem sempre justificada, conduzindo, na maior parte das vezes, a prazos totalmente irrealísticos, definidos por quem não tem um grau de envolvimento significativo no projeto.

Na prática, a definição de cronogramas dos projetos é feita de maneira arbitrária; os riscos do projeto só são considerados quando eles transformaram-se em problemas reais; a organização da equipe nem sempre é clara e feita de forma consciente.

Neste capítulo serão discutidos alguns pontos fundamentais para que o projeto de desenvolvimento de um software seja conduzido de forma a obter resultados satisfatórios em termos de produtividade (do processo) e qualidade (do produto).

2. Análise de Riscos

A análise dos riscos é uma das atividades essenciais para o bom encaminhamento de um projeto de software. Esta atividade está baseada na realização de quatro tarefas, conduzidas de forma seqüencial: a identificação, a projeção, a avaliação e a administração.

2.1. A Identificação dos Riscos

Nesta primeira tarefa, o objetivo é que sejam levantados, da parte do gerente e dos profissionais envolvidos no projeto, todos os eventuais riscos aos quais este será submetido. Nesta identificação, riscos de diferentes naturezas podem ser detectados:

- **riscos de projeto**, os quais estão associados a problemas relacionados ao próprio processo de desenvolvimento (orçamento, cronograma, pessoal, etc...);
- **riscos técnicos**, que consistem dos problemas de projeto efetivamente (implementação, manutenção, interfaces, plataformas de implementação, etc...);
- **riscos de produto**, os quais estão mais relacionados aos problemas que vão surgir para a inserção do software como produto no mercado (oferecimento de um produto que ninguém está interessado; oferecer um produto ultrapassado; produto inadequado à venda; etc...).

A categorização dos riscos, apesar de interessante, não garante a obtenção de resultados satisfatórios, uma vez que nem todos os riscos podem ser identificados facilmente. Uma boa técnica para conduzir a identificação dos riscos de forma sistemática é o estabelecimento de um conjunto de questões (*checklist*) relacionado a algum fator de

risco. Por exemplo, com relação aos riscos de composição da equipe de desenvolvimento, as seguintes questões poderiam ser formuladas:

- são os melhores profissionais disponíveis?
- os profissionais apresentam a combinação certa de capacidades?
- há pessoas suficientes na equipe?
- os profissionais estão comprometidos durante toda a duração do projeto?
- algum membro da equipe estará em tempo parcial sobre o projeto?
- os membros da equipe estão adequadamente treinados?

Em função das respostas a estas perguntas, será possível ao planejador estabelecer qual será o impacto dos riscos sobre o projeto.

2.2. Projeção dos Riscos

A projeção ou estimativa de riscos permite definir basicamente duas questões:

- Qual a probabilidade de que o risco ocorra durante o projeto?
- Quais as conseqüências dos problemas associados ao risco no caso de ocorrência do mesmo?

As respostas a estas questões podem ser obtidas basicamente a partir de quatro atividades:

- estabelecimento de uma escala que reflita a probabilidade estimada de ocorrência de um risco;
- estabelecimento das conseqüências do risco;
- estimativa do impacto do risco sobre o projeto e sobre o software (produtividade e qualidade);
- anotação da precisão global da projeção de riscos.

A escala pode ser definida segundo várias representações (booleana, qualitativa ou quantitativa). No limite cada pergunta de uma dada checklist pode ser respondida com "sim" ou "não", mas nem sempre esta representação permite representar as incertezas de modo realístico.

Uma outra forma de se representar seria utilizar uma escala de probabilidades qualitativas, onde os valores seriam: altamente improvável, improvável, moderado, provável, altamente provável. A partir destes "valores" qualitativos, seria possível realizar cálculos que melhor expressassem as probabilidades matemáticas de que certo risco viesse a ocorrer.

O próximo passo é então o levantamento do impacto que os problemas associados ao risco terão sobre o projeto e sobre o produto, o que permitirá priorizar os riscos. Pode-se destacar três fatores que influenciam no impacto de um determinado risco: a sua natureza, o seu escopo e o período da sua ocorrência.

A natureza do risco permite indicar os problemas prováveis se ele ocorrer (por exemplo, um risco técnico como uma mal definição de interface entre o software e o hardware do cliente levará certamente a problemas de teste e integração). O seu escopo permite indicar de um lado qual a gravidade dos problemas que serão originados e qual a parcela do projeto que será atingida. O período de ocorrência de um risco permite uma reflexão sobre quando ele poderá ocorrer e por quanto tempo.

Estes três fatores definirão a importância do risco para efeito de priorização...um fator de risco de elevado impacto pode ser pouco considerado a nível do gerenciamento de projeto se a sua probabilidade de ocorrência for baixa. Por outro lado fatores de risco com alto peso de impacto e com probabilidade de ocorrência de moderada a alta e fatores de risco com baixo peso de impacto com elevada probabilidade de ocorrência não devem ser desconsiderados, devendo ser processados por todas atividades de análise de risco.

2.3. Avaliação dos Riscos

O objetivo da atividade de avaliação dos riscos é processar as informações sobre o fator de risco, o impacto do risco e a probabilidade de ocorrência. Nesta avaliação, serão checadas as informações obtidas na projeção de riscos, buscando priorizá-los e definir formas de controle destes ou de evitar a ocorrência daqueles com alta probabilidade de ocorrência.

Para tornar a avaliação eficiente, deve ser definido um **nível de risco referente**. Exemplos de níveis referentes típicos em projetos de Engenharia de Software são: o custo, o prazo e o desempenho. Isto significa que se vai ter um nível para o excesso de custo, para a ultrapassagem de prazo e para a degradação do desempenho ou qualquer combinação dos três. Desta forma, caso os problemas originados por uma combinação de determinados riscos provoquem a ultrapassagem de um ou mais desses níveis, o projeto poderá ser suspenso. Normalmente, é possível estabelecer um limite, denominado de ponto referente (*breakpoint*) onde tanto a decisão de continuar o projeto ou de abandoná-lo podem ser tomadas.

A figura 4.1 ilustra esta situação, na qual dois níveis de risco referente são considerados (o custo e o prazo). Se uma conbinação de riscos conduzir a uma situação onde os limites de ultrapassagem de custo e/ou de prazo estejam acima do limite (delimitado pela curva na figura), o projeto será abandonado (área em cinza escuro). No breakpoint, as decisões de continuar o projeto ou abandoná-lo têm igual peso.

2.4. Administração e Monitoração dos Riscos

Uma vez avaliados os riscos de desenvolvimento, é importante que medidas sejam tomadas para evitar a ocorrência dos riscos ou que ações sejam definidas para a eventualidade da ocorrência dos riscos. Este é o objetivo da tarefa de Administração e monitoração dos riscos. Para isto, as informações mais importantes são aquelas obtidas na tarefa anterior, relativa à *descrição*, *probabilidade de ocorrência* e *impacto sobre o processo*, associadas a cada fator de risco.

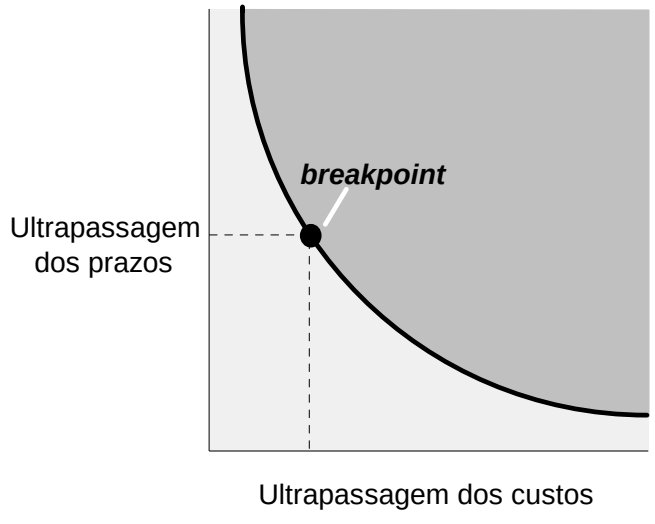


Figura 4.1 - Ilustração de uma situação de definição de dois níveis de riscos.

Por exemplo, considerando a alta rotatividade de pessoal numa equipe um fator de risco, com base em dados de projetos passados, obtém-se que a probabilidade de ocorrência deste risco é de 0,70 (muito elevada) e que a sua ocorrência pode aumentar o prazo do projeto em 15% e o seu custo global em 12%.
Sendo assim, pode-se propor as seguintes ações de administração deste fator de risco:

- reuniões com os membros da equipe para determinar as causas da rotatividade de pessoal (más condições de trabalho, baixos salários, mercado de trabalho competitivo, etc...);
- tomada de providências para eliminar ou reduzir as causas "controláveis" antes do início do projeto;
- no início do projeto, pressupor que a rotatividade vai ocorrer e prever a possibilidade de substituição de pessoas quando estas deixarem a equipe;
- organizar equipes de projeto de forma que as informações sobre cada atividade sejam amplamente difundidas;
- definir padrões de documentação para garantir a produção de documentos de forma adequada;
- realizar revisões do trabalho entre colegas de modo que mais de uma pessoa esteja informado sobre as atividades desenvolvidas;
- definir um membro da equipe que possa servir de *backup* para o profissional mais crítico.

É importante observar que a implementação destas ações pode afetar também os prazos e o custo global do projeto. Isto significa que é necessário poder avaliar quando os custos destas ações podem ser ultrapassados pela ocorrência dos fatores de risco.

Num projeto de grande dimensão, de 30 a 40 fatores de risco podem ser identificados; se para cada fator de risco, sete ações forem definidas, a administração dos riscos pode tornar-se um projeto ela mesma. Por esta razão, é necessário que uma priorização dos riscos seja efetuada, atingindo normalmente, a 20% de todos os fatores levantados.

Todo o trabalho efetuado nesta tarefa é registrado num documento denominado **Plano de Administração e Monitoração de Riscos**, o qual será utilizado posteriormente pelo gerente de projetos (particularmente, para a definição do Plano de Projeto, que é gerado ao final da etapa de planejamento). Uma ilustração sintetizando os passos essenciais desta tarefa é apresentada à figura 4.2.

3. DEFINIÇÃO DE UM CRONOGRAMA

A definição de um cronograma pode ser obtida segundo duas diferentes abordagens:

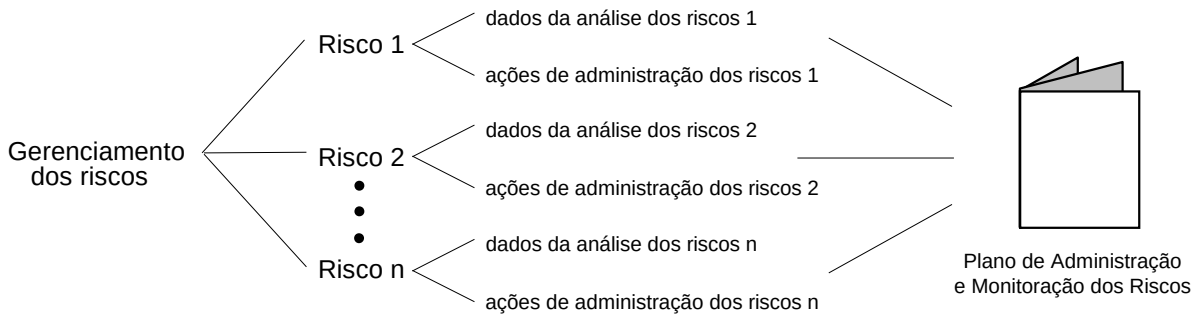


Figura 4.2 - Síntese da tarefa de Administração e Monitoração dos Riscos.

- a primeira, é baseada na definição prévia de um prazo de entrega do software; neste caso, o planejamento deve ser feito de modo a distribuir os esforços ao longo do prazo estabelecido;
- a segunda, está relacionada a uma discussão de limites cronológicos aproximados para cada etapa do desenvolvimento, sendo que o prazo de entrega do software seja estabelecido a partir de técnicas de planejamento da Engenharia de Software.

É evidente que a primeira abordagem é a mais encontrada nos projetos de software.

Um cronograma bem definido pode trazer enormes benefícios a um projeto de software, sendo às vezes mais importante que a própria definição de custos. Numa visão de desenvolvimento de software como produto, um adicional nos custos de produção pode ser absorvido por uma redefinição nos preços ou pela amortização em função de um elevado número de vendas. Já, um acréscimo imprevisto no prazo de entrega de um software pode provocar grandes prejuízos ao produto, como por exemplo: a queda no impacto de mercado, insatisfação dos clientes, elevação de custos internos, etc...

Quando a tarefa é fixar prazos para os projetos de software, diversas questões podem ser formuladas:

- como relacionar o tempo cronológico com o esforço humano?
- que tarefas e que grau de paralelismo podem ser obtidos?
- como medir o progresso do processo de desenvolvimento (indicadores de progresso)?
- como o esforço pode ser distribuído ao longo do processo de Engenharia de Software?
- que métodos estão disponíveis para a determinação de prazos?
- como representar fisicamente o cronograma e como acompanhar o progresso a partir do início do projeto?

As seções que seguem discutirão algumas destas questões.

3.1. As relações pessoas-trabalho

Em primeiro lugar, é preciso dizer que, à medida que um projeto ganha dimensão, um maior número de pessoas deve estar envolvida. Além disso, deve-se tomar o cuidado de não levar a sério o mito, apresentado no capítulo I, de que *"Se o desenvolvimento do software estiver atrasado, basta aumentar a equipe para honrar o prazo de desenvolvimento."*

Deve-se considerar, ainda que, quanto maior o número de pessoas, maior o número de canais de comunicação, o que normalmente requer esforço adicional e, conseqüentemente, tempo adicional.

A experiência tem mostrado que pode ser mais interessante realizar o desenvolvimento de um software com uma equipe com menos pessoas por um período maior de tempo (quando as restrições de prazo de entrega assim o permitem) do que o contrário.

3.2. A Definição de tarefas

Uma das vantagens de se ter uma equipe com mais de uma pessoa são as possibilidades de se realizar determinadas tarefas em paralelo. O paralelismo de tarefas é um mecanismo interessante como forma de economia de tempo na realização de qualquer trabalho de engenharia. Para isto, basta que um gerenciamento dos recursos necessários a cada tarefa (recursos humanos, recursos de software, etc...) seja feito de forma eficiente.

Sendo assim, as tarefas a serem realizadas durante um projeto de desenvolvimento de software podem ser expressas na forma de uma rede (rede de tarefas) a qual apresenta todas as tarefas do projeto, assim como as suas relações em termos de realização (paralelismo, seqüência, pontos de encontro, etc... Um exemplo desta representação é apresentado na figura 4.3.

Nesta figura, é possível verificar que existem indicadores de progresso (representados por um símbolo "❋") situados ao longo do processo e que permitem ao gerente uma avaliação do estado de evolução do processo. Um indicador de progresso é considerado atingido ou satisfeito quando a documentação produzida com relação a este indicador é revisada e aprovada.

3.3. A Distribuição do Esforço

Normalmente, as técnicas de estimativas para projetos de software conduzem a uma definição do esforço em termos do número de homens-mês ou homens-ano necessárias para realizar o desenvolvimento do projeto. Uma proposta de distribuição de esforço bastante utilizada nos projetos é aquela ilustrada na figura 4.4, a qual é baseada numa regra anteriormente denominada *Regra 40-20-40*, a qual sugere, como etapas onde o esforço deve ser maior, as dos extremos do processo de desenvolvimento (análise/projeto e testes), a codificação sendo a tarefa que deve envolver a menor concentração de esforço. Isto pode ir contra o grau de importância em termos de esforço que muitos desenvolvedores dão a cada uma destas atividades.

É evidente que estes valores não podem ser levados à risca para todos os projetos de software, sendo que as características de cada projeto vai influenciar na parcela de esforço a ser dedicada a cada etapa.

No entanto, é importante entender a razão pela qual o esforço dedicado à etapa de codificação aparece com menor intensidade que os demais.

Na realidade, se a etapa de projeto foi realizada utilizando as boas regras da Engenharia de Software, a etapa de codificação será, consequentemente, minimizada em termos de esforço.

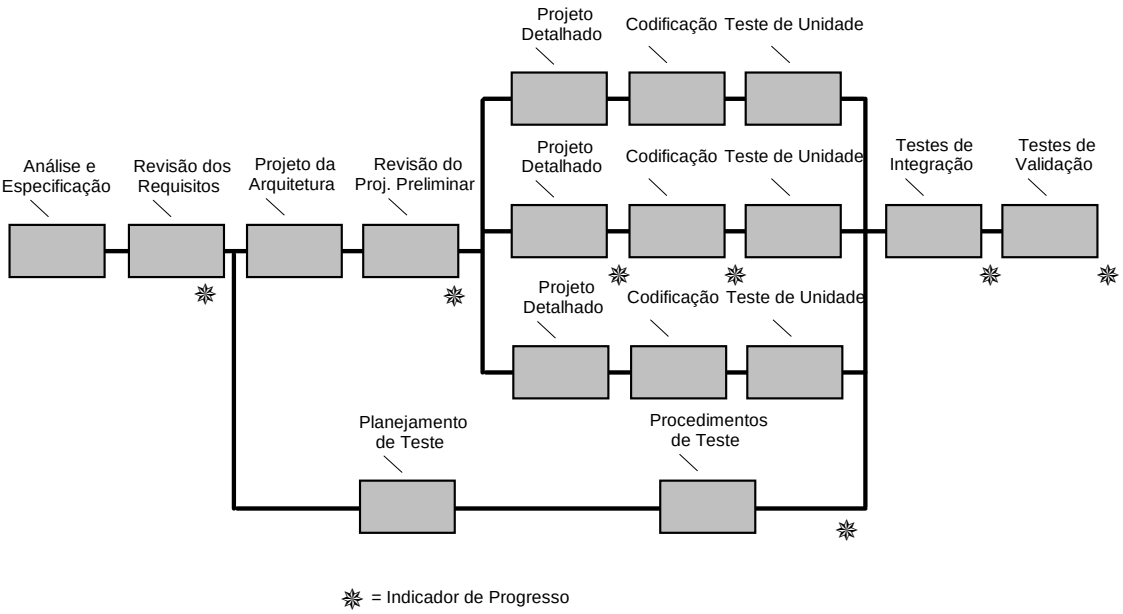


Figura 4.3 - Exemplo de uma Rede de Tarefas.

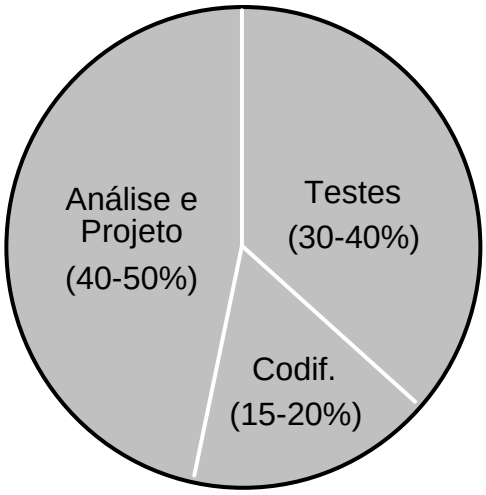


Figura 4.4 - Proposta de distribuição dos esforços de desenvolvimento.

Nos processos de desenvolvimento em que as etapas de projeto/análise não são enfatizadas, existe uma razão porque a etapa de codificação acaba exigindo maior esforço que estas etapas... atividades de análise e decisões de projeto acabam sendo efetuadas na etapa de codificação, o que certamente pode corresponder a um custo de desenvolvimento superior ao de um processo onde isto fosse feito segundo uma metodologia mais condizente com o que prega a Engenharia de Software.

A etapa de testes, por sua vez, pode representar de 30 a 40% do esforço total do projeto. Na realidade, o que vai determinar o esforço a ser dispendido com os testes serão os requisitos do próprio software. Softwares concebidos para aplicações críticas, onde as conseqüências dos erros podem ser catastróficas (seja em termos de perdas humanas ou materiais), vão exigir um esforço muito maior em termos de teste do que as aplicações mais clássicas (de automação de escritório, por exemplo).

3.3. A Representação do Cronograma

A definição do cronograma é uma das tarefas mais difíceis de se definir na etapa de planejamento do software. O planejador deve levar em conta diversos aspectos relacionados ao desenvolvimento, como: a disponibilidade de recursos no momento da execução de uma dada tarefa, as interdependências das diferentes tarefas, a ocorrência de possíveis estrangulamentos do processo de desenvolvimento e as operações necessárias para agilizar o processo, identificação das principais atividades, revisões e indicadores de progresso do processo de desenvolvimento, etc...

A forma de representação dos cronogramas depende da política de desenvolvimento adotada, mas pode ser apresentada, numa forma geral por um documento do tipo apresentado na figura 4.5.

As unidades de tempo consideradas no projeto (dias, semanas, meses, etc...) são anotadas na linha superior da folha de cronograma. As tarefas do projeto, as atividades e os indicadores de progresso são definidos no coluna da esquerda. O traço horizontal permite indicar o período durante o qual determinada atividade será realizada, o tempo necessário sendo medido em unidades de tempo consideradas. Quando atividades puderem ser realizadas em paralelo, os traços vão ocupar unidades de tempo comuns.

O cronograma deve explicitar as atividades relevantes do desenvolvimento e os indicadores de progresso associados. É importante que os indicadores de progresso sejam representados por resultados concretos (por exemplo, um documento).

A disponibilidade de recursos deve também ser representada ao longo do cronograma. O impacto da indisponibilidade dos recursos no momento em que eles são necessários deve também ser representado, se possível, no cronograma.

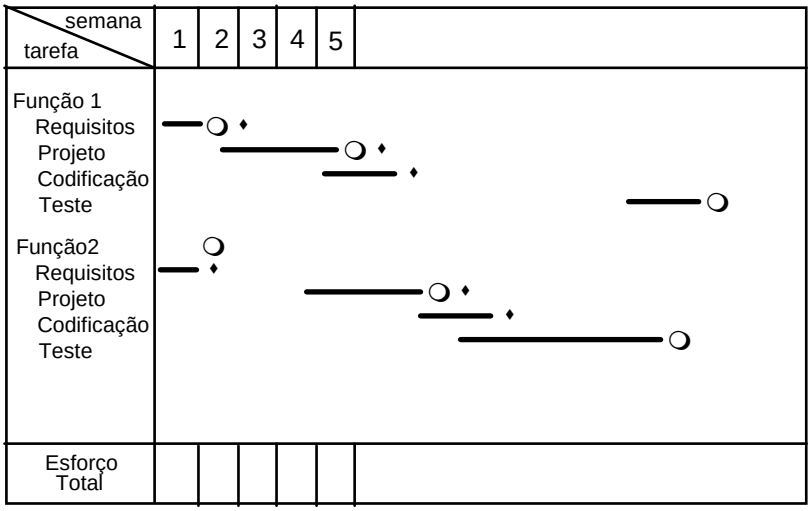


Figura 4.5 - Representação geral de um cronograma de desenvolvimento.

4. AQUISIÇÃO DE SOFTWARE

Embora a Engenharia de Software sugira o desenvolvimento de software, em alguns casos pode ser mais interessante adquirir o software do que desenvolvê-lo. Esta é uma decisão a qual o gerente de um projeto pode ter de tomar no contexto de um projeto. Com relação à aquisição de software, diversas ações podem ser tomadas:

- adquirir (ou licenciar) um pacote comercial que atenda às especificações estabelecidas;
- adquirir um pacote comercial e modificá-lo de forma a que o novo software atenda às especificações de projeto;
- encomendar o software a terceiros para que este atenda às especificações.

Os procedimentos a efetuar para a aquisição de um software vão depender da criticalidade das especificações e do seu custo. No caso de um software de custo relativamente baixo (um software para PC, por exemplo), pode ser mais interessante adquirí-lo e fazer uma análise de adequação às especificações de projeto.

No caso de softwares mais caros, uma análise mais cuidadosa se faz necessária, sendo que os procedimentos podem ser os seguintes:

- desenvolver uma especificação funcional e de desempenho para o software a projetar, estabelecendo, quando possível, valores mensuráveis;
- realizar a estimação do custo interno de desenvolvimento;
- escolher um conjunto de pacotes comerciais que poderiam atender às especificações;
- desenvolver um esquema de análise comparativa que permita confrontar as funções-chave das soluções alternativas;
- avaliar cada pacote com base na qualidade do produto, no suporte do vendedor (garantia), reputação do vendedor e direcionamento do produto;
- entrevistar usuários do software, colhendo suas opiniões sobre os pacotes.

A partir dos resultados obtidos, a decisão entre comprar ou desenvolver o software vai estar baseada nos seguintes critérios:

- a data de entrega do produto precede a data de finalização do produto se desenvolvido internamente?

- o custo de aquisição mais o custo de "customização" será inferior ao custo de desenvolvimento interno?
- o custo de suporte externo (contrato de manutenção) é inferior ao custo do suporte interno?

5. REENGENXARIA

Um outro problema importante a ser tratado pela Engenharia de Software são os antigos pacotes de software que são fundamentais à realização de negócios de uma empresa e que oferecem grandes dificuldades de manutenção. Historicamente, a manutenção destes pacotes foi realizada com base em verdadeiros "remendos", sem nenhuma documentação, resultando muitas vezes em programas ineficientes e com alta taxa de falhas. Estes fatores conduziram a uma situação na qual os custos de manutenção de tais sistemas não justificam mais os benefícios que tais alterações podem trazer.

Por outro lado, um processo de reengenharia do software pode aparecer como uma alternativa de baixo custo à manutenção do software. Para isto, é importante que os seguintes procedimentos sejam encaminhados:

- selecionar os programas que estejam sendo bastante utilizados no momento e que continuarão a ser utilizados nos próximos 5 a 10 anos;
- estimar o custo anual de manutenção dos programas selecionados, incluindo correção de erros, adaptação e melhorias funcionais;
- organizar, por ordem de prioridade, os programas selecionados, registrando os custos levantados no procedimento anterior;
- estimar os custos para realizar a reengenharia dos programas selecionados e estimar os custos de manutenção do programa após realizada a reengenharia;
- realizar uma análise comparativa de custos para cada programa;
- calcular o tempo necessário para o retorno de investimento da reengenharia;
- levar em consideração algumas questões importantes que resultam da reengenharia, como a melhor confiabilidade do sistema, melhor desempenho e melhores interfaces;
- obter a aprovação da empresa para realizar a reengenharia de um programa;
- com base nos resultados obtidos desta experiência, desenvolver uma estratégia de reengenharia dos demais programas.

6. PLANEJAMENTO ORGANIZACIONAL

Existe uma grande diversidade no que diz respeito aos modos de organização das equipes de desenvolvimento de software. A escolha da forma como a equipe de desenvolvimento vai ser organizada é um dos pontos que deve ser definido nesta etapa. Considerando que um processo de desenvolvimento de software vai ocupar uma equipe de n pessoas com uma duração de k anos, é possível comentar algumas opções organizativas:

① n indivíduos são alocados a m diferentes tarefas com pequeno grau de interação, sendo que a coordenação da equipe fica a cargo do gerente de projeto;
② n indivíduos são alocados a m diferentes tarefas, com $m \leq n$, formando equipes informais de desenvolvimento, com um responsável ad-hoc de cada equipe, sendo que a coordenação entre as equipes é da responsabilidade do gerente do projeto;
③ n indivíduos são organizados em k equipes, cada equipe sendo alocada para uma ou mais tarefas; a organização de cada equipe é específica a ela própria, a coordenação ficando a cargo da equipe e do gerente do projeto.

Sem discutir em detalhes os argumentos contrários ou a favor de cada uma das opções, é importante dizer que a constituição em equipes formais de desenvolvimento (como sugere a opção ③) é mais produtiva.

O principal objetivo da formação de equipes é o desenvolvimento de um conceito de projeto como sendo o resultado de uma reunião de esforços. A criação de equipes evita o sentimento de "ego-programação" que pode atingir as pessoas envolvidas num desenvolvimento, transformando o "meu programa" em "nosso programa".

De um ponto de vista geral, pode-se estabelecer uma referência no que diz respeito à organização de uma equipe de desenvolvimento de software, sendo que o número de equipes e o número de membros de cada equipe vai variar em função da grandeza do projeto.

O núcleo de uma equipe vai ser composto dos seguintes elementos:

- um **engenheiro sênior** (ou programador chefe), responsável do planejamento, coordenação e supervisão de todas as atividades relativas ao desenvolvimento do software;
- o **pessoal técnico**, de dois a cinco membros que realizam as atividades de análise e desenvolvimento;
- um **engenheiro substituto**, que atua no apoio ao engenheiro sênior e que pode, eventualmente, substituí-lo sem grandes prejuízos ao desenvolvimento do software.

Além deste núcleo, a equipe de desenvolvimento pode ainda receber a colaboração dos seguintes elementos:

- um **conjunto de especialistas** (telecomunicações, bancos de dados, interface homem-máquina, etc...);
- **pessoal de apoio** (secretárias, editores técnicos, desenhistas, etc...);
- um **bibliotecário**, o qual será responsável da organização e catalogação de todos os componentes do produto de software (documentos, listagens, mídia magnética, coleta de dados relativos ao projeto, módulos reutilizáveis, etc...).

7. MÉTRICA DO SOFTWARE

7.1. IMPORTÂNCIA DA MÉTRICA

Em primeiro lugar, é importante estar ciente que as medidas são uma forma clara de avaliação da produtividade no desenvolvimento de software.

Sem informações quantitativas a respeito do processo de desenvolvimento de softwares por parte de uma empresa ou equipe de desenvolvedores de produto, é impossível tirar qualquer conclusão sobre de que forma está evoluindo a produtividade (se é que está evoluindo!?!).

Através da obtenção de medidas relativas à produtividade e à qualidade, é possível que metas de melhorias no processo de desenvolvimento sejam estabelecidas como forma de incrementar estes dois importantes fatores da Engenharia de Software.

Particularmente no que diz respeito à qualidade, é possível, com uma avaliação quantitativa deste parâmetro, promover-se pequenos ajustes no processo de desenvolvimento como forma de eliminar ou reduzir as causas de problemas que afetam de forma significativa o projeto de software.

No que diz respeito aos desenvolvedores, a obtenção de medidas podem auxiliar a responder com precisão uma série de perguntas que estes elementos se fazem a cada projeto:

- que tipos de requisitos são os mais passíveis de mudanças?
- quais módulos do sistema são os mais propensos a erros?
- quanto de teste deve ser planejado para cada módulo?

- quantos erros (de tipos específicos) pode-se esperar quando o teste se iniciar?

7.2. Medidas de Software

Na área de engenharia, a medição tem sido um aspecto de grande importância, sendo que poderíamos desfilar uma fila interminável de grandezas as quais sofrem este tipo de tratamento: dimensões físicas, peso (ou massa), temperatura, tensões e correntes elétrica, etc... No mundo dos computadores, alguns parâmetros são quantificados como forma de expressar as potencialidades de determinadas máquinas, tais como a capacidade de um processador de executar um certo número de instruções por segundo (MIPS), as capacidades de armazenamento (Mbytes), a freqüência do clock do processador (MHz), etc...

No caso particular do software, existem diversas razões para que a realização de medições seja um item de importância:

- quantizar a qualidade do software como produto;
- avaliar a produtividade dos elementos envolvidos no desenvolvimento do produto;
- avaliar os benefícios de métodos e ferramentas para o desenvolvimento de software;
- formar uma base de dados para as estimativas;
- justificar o pleito e aquisição de novas ferramentas e/ou treinamento adicional para membros da equipe de desenvolvimento.

De forma análoga a outras grandezas do mundo físico, as medições de software podem ser classificadas em duas categorias principais:

- as **medições diretas**, por exemplo, o número de linhas de código (LOC) produzidas, o tamanho de memória ocupado, a velocidade de execução, o número de erros registrados num dado período de tempo, etc...

- as **medições indiretas**, as quais permitem quantizar aspectos como a funcionalidade, complexidade, eficiência, manutenibilidade, etc...

As medições diretas, tais quais aquelas exemplificadas acima, são de obtenção relativamente simples, desde que estabelecidas as convenções específicas para isto. Por outro lado, aspectos como funcionalidade, complexidade, eficiência, etc..., são bastante difíceis a quantizar.

As medições de software podem ser organizadas em outras classes, as quais serão definidas a seguir:

- **métricas da produtividade**, baseadas na saída do processo de desenvolvimento do software com o objetivo de avaliar o próprio processo;
- **métricas da qualidade**, que permitem indicar o nível de resposta do software às exigências explícitas e implícitas do cliente;
- **métricas técnicas**, nas quais encaixam-se aspectos como funcionalidade, modularidade, manutenibilidade, etc...

Sob uma outra ótica, é possível definir uma nova classificação das medições:

- **métricas orientadas ao tamanho**, baseadas nas medições diretas da Engenharia de Software;
- **métricas orientadas à função**, que oferecem medidas indiretas;
- **métricas orientadas às pessoas**, as quais dão indicações sobre a forma como as pessoas desenvolvem os programas de computador.

7.3. Métricas orientadas ao tamanho

Esta classe abrange todas as possíveis medidas obtidas diretamente do software. Um exemplo de tal classe de medidas é mostrado na tabela a seguir. Como pode ser verificado na tabela desta figura, cada projeto é caracterizado por um conjunto de parâmetros que permite obter diversas informações tanto sobre a produtividade do processo quanto sobre a qualidade do software obtido. Pode-se, então definir algumas grandezas tais como:

Produtividade = KLOC/pessoa-mês
Qualidade = erros/KLOC
Custo = $/KLOC
Documentação = págs/KLOC

A despeito da facilidade em sua obtenção, existe muita discussão em torno das métricas orientadas ao tamanho. Um caso bastante discutido é o da utilização do número de linhas de código como medida de dimensão. Os que defendem o uso desta métrica, afirmam que é um aspecto de fácil obtenção e, portanto, de fácil aplicação a qualquer projeto de software, além de destacar a quantidade de informação existente com base nesta métrica.

Projeto	Esforço	Custo	KLOC	Págs	Erros	Pessoas
AAA-01	24	168	12,1	365	29	3
CCC-04	62	440	27,2	1224	86	5
FFF-03	43	314	20,2	1050	64	6

Os que discutem o uso desta informação, alertam para a forte dependência da linguagem no uso desta técnica. Além disso, a métrica orientada ao tamanho tende a penalizar os programas bem estruturados e que tenham feito economia de software. É uma métrica que, na verdade, deixa de ser precisa principalmente por causa dos diversos novos fatores introduzidos pelas linguagens de programação mais recentes, além de ser de difícil aplicação para efeito de estimativas, uma vez que é necessário descer a um nível de detalhe que não é desejável na etapa de planejamento do projeto.

7.4. Métricas orientadas à função

Esta classe é baseada em medidas indiretas do software e do processo utilizado para obtê-lo. Em lugar da contagem do número de linhas de código, esta métrica leva em conta aspectos como a funcionalidade e a utilidade do programa.

Uma abordagem proposta nesta classe é a do **ponto-por-função** (*function point*), a qual é baseada em medidas indiretas sobre a complexidade do software. A figura 2.2 mostra uma tabela que deve ser preenchida para que se possa obter uma medida sobre a complexidade do software. Os valores a serem computados são os seguintes: número de entradas do usuário, número de saídas do usuário, número de consultas do usuário, número de arquivos e número de interfaces externas.

Uma vez computados todos os dados, um valor de complexidade é associado a cada contagem. Feito isto, obtém-se o valor de pontos-por-função utilizando-se a seguinte fórmula:

$$FP = Contagem\ Total * [0,65 + 0,01 * SOMA(Fi)]$$

Na fórmula acima, além da *Contagem Total*, obtida diretamente da tabela da figura 4.6, *Fi* (para *i = 1* a *14*) corresponde a um conjunto de valores de ajuste de complexidade.

			Fator de ponderação			
Parâmetro	**Contagem**		**Simples**	**Médio**	**Complexo**	
Ent. usuário		**x**	**3**	**4**	**5**	
Saídas usuário		**x**	**4**	**5**	**7**	
Consult. usuário		**x**	**3**	**4**	**6**	
Arquivos		**x**	**7**	**10**	**15**	
Interfaces ext.		**x**	**5**	**7**	**10**	
Contagem total =						

Figura 4.6 - Computação da métrica *ponto-por-função*.

Estes valores são obtidos a partir de respostas dadas a perguntas feitas a respeito da complexidade do software. Os parâmetros da equação acima, assim como os pesos apresentados na figura 4.6 (relativos aos níveis de complexidade) são obtidos empiricamente.

Uma vez computados, os pontos por função podem ser utilizados de forma análoga ao número de linhas de código (LOC) para a obtenção de medidas de qualidade, produtividade e outros atributos:

Produtividade = FP/pessoa-mês
Qualidade = erros/FP
Custo = $/FP
Documentação = págs/FP

8. O PLANO DE SOFTWARE

Ao final desta etapa, um documento de **Plano de Software** deverá ser gerado, o qual deverá ser revisto para servir de referência às etapas posteriores. Ele vai apresentar as informações iniciais de custo e cronograma que vão nortear o desenvolvimento do software. Ele consiste de um documento relativamente breve, o qual é encaminhado às diversas pessoas envolvidas no desenvolvimento do software.

Dentre as informações a serem contidas neste documento, é possível destacar:

- o contexto e os recursos necessários ao gerenciamento do projeto, à equipe técnica e ao cliente;
- a definição de custos e cronograma que serão acompanhados para efeito de gerenciamento;
- dá uma visão global do processo de desenvolvimento do software a todos os envolvidos.

A figura 4.7 apresenta uma possível estrutura para este documento. A apresentação dos custos e cronograma pode diferir dependendo de quem será o leitor do documento.

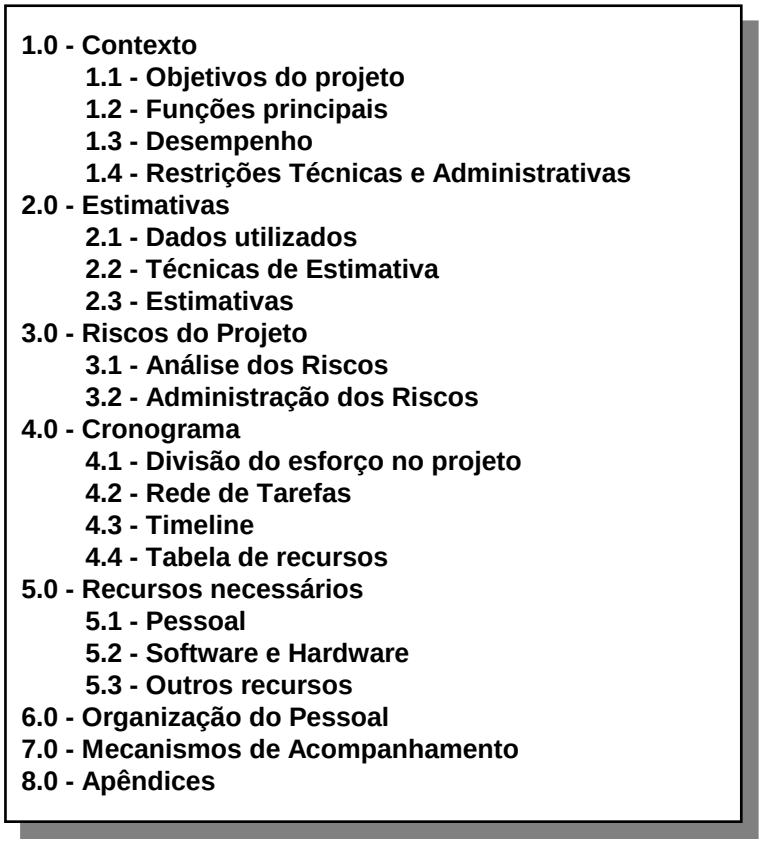


Figura 4.7 - Proposta de estrutura para o documento de Plano do Software.

O Plano de Software não necessita ser um documento extenso e complexo. O seu objetivo é auxiliar a análise de viabilidade dos esforços de desenvolvimento. Este documento está associado aos conceitos de *"o que"*, *"quanto"* e *"quão longo"* associados ao desenvolvimento. As etapas mais à frente vão estar associadas ao conceito de *"como"*.

Engenharia de Software

Capítulo 5

Análise de Requisitos

1. Introdução

O completo entendimento dos requisitos de software é um ponto fundamental para o sucesso de um projeto de software. Independente da precisão com a qual um software venha a ser projetado e implementado, ele certamente trará problemas ao cliente/usuário se a sua análise de requisitos foi mal realizada.

A Análise de Requisitos é uma tarefa que envolve, antes de tudo um trabalho de descoberta, refinamento, modelagem e especificação das necessidades e desejos relativos ao software que deverá ser desenvolvido. Nesta tarefa, tanto o cliente como o desenvolvedor vão desempenhar um papel de grande importância, uma vez que caberá ao primeiro a formulação (de modo concreto) das necessidades em termos de funções e desempenho, enquanto o segundo atua como indagador, consultor e solucionador de problemas.

Esta etapa é de suma importância no processo de desenvolvimento de um software, principalmente porque ela estabelece o elo de ligação entre a alocação do software a nível de sistema (realizada na etapa de Engenharia de Sistema) e o projeto do software. Desta forma, ela permite que o engenheiro de sistemas especifique as necessidades do software em termos de funções e de desempenho, estabeleça as interfaces do software com os demais elementos do sistema e especifique as restrições de projeto. Ao engenheiro de software (ou analista), a análise de requisitos permite uma alocação mais precisa do software no sistema e a construção de modelos do processo, dos dados e dos aspectos comportamentais que serão tratados pelo software. Ao projetista, esta etapa proporciona a obtenção de uma representação da informação e das funções que poderá ser traduzida em projeto procedimental, arquitetônico e de dados. Além disso, é possível definir os critérios de avaliação da qualidade do software a serem verificados uma vez que o software esteja concluído.

2. As Atividades da Análise de Requisitos

A etapa de Análise de Requisitos é caracterizada basicamente pela realização de um conjunto de tarefas, as quais serão discutidas nas seções que seguem.

2.1. A Análise do Problema

Nesta tarefa inicial, o analista estuda os documentos de Especificação do Sistema e o Plano do Software, como forma de entender o posicionamento do software no sistema e revisar o escopo do software utilizado para definir as estimativas de projeto. Um elo de comunicação entre o analista e o pessoal da organização cliente deve ser estabelecido, sendo que o gerente de projetos pode atuar na coordenação dos contatos. A meta do

analista neste contexto é identificar os principais fatores do problema a ser resolvido, pela ótica do cliente.

2.2. A Avaliação e Síntese

Esta segunda tarefa envolve principalmente uma análise dos fluxos de informação e a elaboração de todas as funções de tratamento e os aspectos de comportamento do software. Ainda, é importante que uma definição de todas as questões relacionadas à interface com o sistema, além de uma especificação das restrições de projeto.

Terminada a análise, o analista pode iniciar a síntese de uma ou mais soluções para o problema. Na síntese das eventuais soluções, o analista deve levar em conta as estimativas e as restrições de projeto. Este processo de avaliação e síntese prossegue até que o analista e o cliente estejam de acordo sobre a adequação das especificações realizadas para a continuidade do processo.

2.3. A Modelagem

A partir da tarefa de avaliação e síntese, o analista pode estabelecer um modelo do sistema, o qual permitirá uma melhor compreensão dos fluxos de informação e de controle, assim como dos aspectos funcionais e de comportamento. Este modelo, ainda distante de um projeto detalhado, servirá de referência às atividades de projeto, assim como para a criação da especificação de requisitos.

Em muitas situações, como forma de reforçar o conhecimento sobre a viabilidade do software a ser desenvolvido, pode ser necessário o desenvolvimento de um protótipo de software como alternativa ou como trabalho paralelo à análise de requisitos. Este ponto será discutido mais adiante neste documento.

2.4. Especificação dos Requisitos e Revisão

A etapa de Análise de Requisitos culmina com a produção de um documento de **Especificação de Requisitos de Software**, o qual registra os resultados das tarefas realizadas. Eventualmente, pode ser produzido como documento adicional um Manual Preliminar do Usuário. Embora pareça estranho, a produção deste manual permite que o analista passe a olhar para o software da ótica do cliente/usuário, o que pode ser bastante interessante, principalmente em sistemas interativos. Além disso, a posse de um Manual de Usuário, mesmo em estágio preliminar permite ao cliente uma revisão dos requisitos (de interface, pelo menos) ainda num estágio bastante prematuro do desenvolvimento de software. Desta forma, algumas decepções resultante de uma má definição de alguns aspectos do software podem ser evitadas.

3. PROCESSOS DE COMUNICAÇÃO

O desenvolvimento de um software é, na maior parte dos casos, motivado pelas necessidades de um cliente, que deseje automatizar um sistema existente ou obter um novo sistema completamente automatizado. O software, porém, é desenvolvido por um desenvolvedor ou por uma equipe de desenvolvedores. Uma vez desenvolvido, o software será provavelmente utilizado por usuários finais, os quais não são necessariamente os clientes que originaram o sistema.

Isto significa que, de fato, existem três partes envolvidas no processo de desenvolvimento de um produto de software: o cliente, o desenvolvedor e os usuários. Para que o processo de desenvolvimento seja conduzido com sucesso, é necessário que os desejos do cliente e as expectativas dos eventuais usuários finais do sistema sejam precisamente transmitidos ao desenvolvedor.

Este processo de transmissão de requisitos, do cliente e dos usuários ao desenvolvedor, invariavelmente apresenta relativa dificuldade, considerando alguns aspectos:

- geralmente, os clientes não entendem de software ou do processo de desenvolvimento de um programa;
- o desenvolvedor, usualmente, não entende do sistema no qual o software vai executar.

Estes dois aspectos provam que existem, efetivamente, um problema de comunicação a ser resolvido para que o processo de desenvolvimento seja bem sucedido. A importância desta etapa está no fato de que é através dela que as idéias do cliente sobre o problema a ser resolvido pelo sistema a desenvolver são expressas na forma de um documento, se possível, que utilize ferramentas formais.

Uma Análise de Requisitos bem sucedida deve, normalmente, representar corretamente as necessidades do cliente e dos usuários, satisfazendo, porém às três partes envolvidas (incluindo o desenvolvedor). O que é verificado, em boa parte dos projetos de software é que o cliente nem sempre entende perfeitamente quais são as suas reais necessidades, assim como os usuários têm dificuldades para exprimir as suas expectativas com relação ao que está sendo desenvolvido. Um primeiro resultado desta etapa deve ser, sem dúvida, o esclarecimento a respeito do que são estas necessidades e expectativas.

A obtenção bem sucedida de informações é um fator preponderante para o sucesso da Análise de Requisitos, particularmente nas duas primeiras tarefas descritas anteriormente. A compilação das informações relevantes para o processo de desenvolvimento é uma tarefa bastante complexa, principalmente porque entram, muitas vezes, em jogo um conjunto de informações conflitantes. Ainda, quanto mais complexo é o sistema a ser desenvolvido, mais inconsistência haverá com relação às informações obtidas.

Neste contexto, o analista deve ter bom senso e experiência para extrair de todo o processo de comunicação as "boas" informações.

4. PRINCÍPIOS DE ANÁLISE

Independente do método utilizado para realizar a análise de requisitos, existem algumas preocupações que são comuns a todos eles, os quais discutiremos nos parágrafos que seguem.

4.1. O Domínio de Informação

Todo software é construído com a função básica de processar dados, não importa a área de atuação considerada. Como já discutimos no capítulo anterior, é possível representar um software na sua concepção mais clássica como um sistema que recebe um conjunto de informações (ou dados) de entrada, realiza o tratamento ou processamento desta informação e produz informações de saída.

Além dos dados, um software é capaz de processar eventos, onde cada evento está relacionado a um aspecto de controle do sistema, como por exemplo, um sensor que é capaz de detectar a ultrapassagem de um determinado limite de pressão ou temperatura pode gerar um evento (um sinal de alarme) para que o software de monitoração alerte o operador ou efetue alguma ação previamente definida.

Isto significa que os dados e os eventos fazem parte do domínio de informação do software, sendo que basicamente três pontos de vista podem ser considerados para abordar esta questão:

- o **fluxo da informação**, o qual representa a forma pela qual os dados e os eventos se modificam ao longo do sistema. o que pode ajudar a determinar quais

são as transformações essenciais às quais os itens de informação são submetidos;
- o **conteúdo da informação**, que permite representar os dados e os eventos que estejam relacionados a um determinado item de informação (semântica da informação);
- a **estrutura da informação**, a qual permite expressar a forma como itens de dados e/ou eventos estão organizados (sintaxe da informação); esta informação pode vir a ser importante para a etapa de projeto e implementação das estruturas de informação via software (linguagens de programação).

4.2. Modelagem

A modelagem é um outro aspecto de importância no processo de análise de requisitos de um software, uma vez que ela permite uma melhor compreensão das questões arquiteturais e comportamentais do problema a ser resolvido com o auxílio do software. Uma boa modelagem de software deve permitir a representação da informação a ser transformada pelo software, das funções (ou sub-funções) responsáveis das transformações e do comportamento do sistema durante a ocorrência destas transformações.

Um modelo realizado durante a etapa de Análise de Requisitos deve concentrar-se na representação do **que** o software deve realizar e não em **como** ele o realiza. Normalmente, é desejável que os modelos sejam expressos através de uma notação gráfica que permita descrever os diferentes aspectos citados no parágrafo anterior. Nada impede, porém, que um modelo seja dotado de descrições textuais complementares, sendo que normalmente, estas descrições devem ser realizadas ou por meio de linguagem natural ou de uma linguagem especializada que permita expressar, sem ambigüidades, os requisitos estabelecidos.

A obtenção de um modelo representativo dos requisitos do software pode ser útil às diferentes partes envolvidas no desenvolvimento do software:

- ao **analista**, para uma melhor compreensão da informação, as funções e o comportamento do sistema, o que pode tornar a tarefa de análise mais sistemática;
- ao **pessoal técnico** como um todo, uma vez que ele pode ser uma referência de revisão, permitindo a verificação de algumas propriedades, como a completude, a coerência e a precisão da especificação;
- ao **projetista**, servindo de base para o projeto através da representação dos aspectos essenciais do software.

4.3. Particionamento

Em boa parte das vezes, os problemas a serem resolvidos são excessivamente complexos, apresentando grande dificuldade para a sua compreensão (e conseqüente resolução) como um todo. Com a finalidade de dominar de forma completa os problemas sob análise, um princípio fundamental é a decomposição do mesmo em partes menores, o que denominaremos de particionamento. A partir do particionamento de um problema e a partir da análise de cada parte estabelecida, o entendimento fica mais facilitado. Desta forma, é possível estabelecer as interfaces de cada parte do problema de modo a que a função global do software seja realizada.

Segundo este princípio, as funções, os aspectos de comportamento e as informações a serem manipuladas pelo programa poderão ser alocadas às diferentes partes.

De um ponto de vista genérico, o procedimento básico de particionamento é o estabelecimento de uma estrutura hierarquizada de representação da função ou da

informação dividindo em partições o elemento superior, esta divisão podendo ser efetuada segundo uma abordagem vertical (deslocando-se verticalmente na hierarquia) ou horizontal. As figuras 5.1 e 5.2 representam a aplicação sobre um exemplo das abordagens horizontal e vertical, respectivamente.

No caso dos exemplos ilustrativos, o particionamento é realizado sobre o aspecto funcional do software, mas poderia ser aplicado, segundo uma ou outra abordagem, sobre os aspectos informacionais ou comportamentais.

4.4. Concepções essenciais e de implementação

Os requisitos de software podem ser especificados segundo dois critérios, dependendo do grau de conhecimento que se tem do sistema ou dependendo do estágio de análise no qual nos encontramos.

O primeiro critério é o da concepção essencial, onde o objetivo é contemplar os aspectos essenciais do problema sob análise, sem preocupação com detalhes de implementação. Considerando o exemplo de problema ilustrado nas figuras 5.1 e 5.2, a concepção essencial da função **ler status** ignora o formato dos dados de status ou o tipo de sensor que será utilizado. A vantagem de realizar a concepção essencial é de deixar em aberto as alternativas de implementação possíveis, o que é adequado para as atividades iniciais do desenvolvimento.

O segundo critério é o da concepção de implementação, o qual permite expressar os requisitos do software segundo as funções de processamento e as estruturas de informação do sistema real.

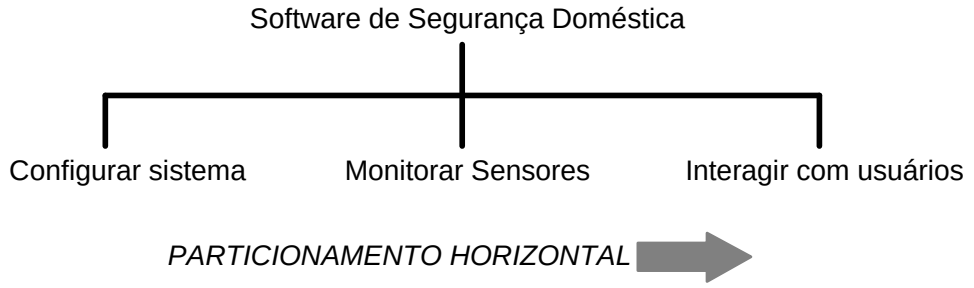


Figura 5.1 - Ilustração da abordagem horizontal de particionamento.

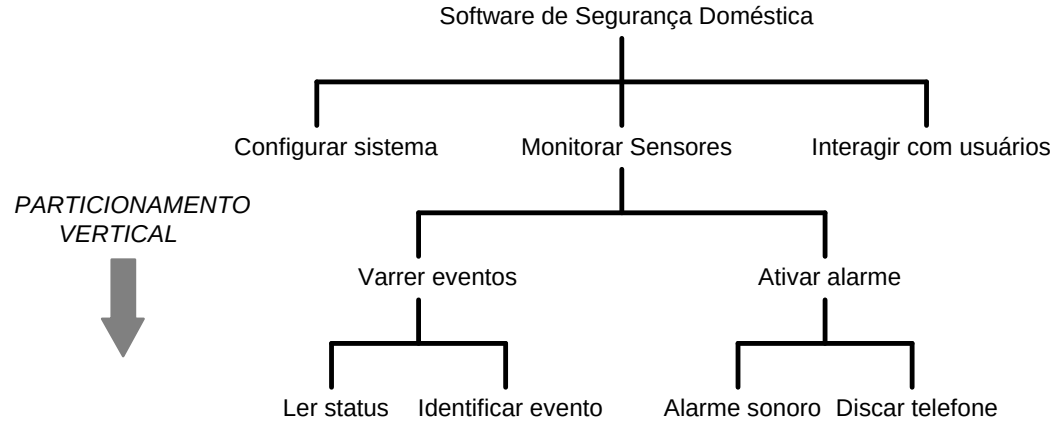


Figura 5.2 - Ilustração da abordagem vertical de particionamento.

Em alguns casos, uma representação física é o ponto de partida da etapa de projeto do software mas, na maioria dos casos, grande parte dos sistemas computacionais é especificada acomodando alguns detalhes de implementação. O conhecimento de alguns destes detalhes pode auxiliar o analista (e, em seguida, o projetista) na definição de restrições impostas ao software pelo sistema.

5. PROTOTIPAÇÃO DE SOFTWARE

5.1. Motivações e Etapas da Prototipação

A etapa de Análise de Requisitos engloba um conjunto de atividades de fundamental importância para o processo de desenvolvimento de um software e deve ser, portanto, realizada independentemente da abordagem e técnica utilizadas.

Em muitos casos, é possível aplicar-se técnicas de análise de modo a derivar uma representação em papel (ou em arquivo, no caso da utilização de uma ferramenta CASE) que sirva de referência para o projeto do software.

Em outras situações, a partir da coleta de requisitos um modelo do software — um **protótipo** — pode ser construído de modo a permitir uma melhor avaliação por parte do desenvolvedor e do cliente.

O paradigma da prototipação tem como ponto de partida uma **Solicitação de Proposta** encaminhada pelo cliente. A partir desta solicitação, os seguintes passos são realizados:

- análise da solicitação para identificar a necessidade da prototipação; normalmente, softwares interativos e/ou gráficos ou softwares que exijam a utilização de algoritmos combinatórios podem ser objeto de prototipação; no entanto, o fator complexidade deve permitir determinar a realização ou não do protótipo; outro ponto que deve ser pesado nesta decisão é se a equipe de desenvolvimento tem experiência e ferramentas adequadas à prototipação;
- especificação resumida dos requisitos, realizada pelo analista, de modo a representar o domínio da informação e os domínios funcionais e comportamentais do software, de modo que a atividade de particionamento do problema possa ser efetuada;
- revisada a especificação resumida dos requisitos, é realizado um projeto do protótipo, concentrando-se principalmente nos aspectos arquitetônicos e de dados e deixando de lado os aspectos procedimentais;
- desenvolvimento do protótipo, se possível a partir da utilização de blocos de construção de software preexistentes (o que na maioria dos casos são de difícil obtenção); por outro lado, a construção de um protótipo pode ser facilitada graças à existência de diversas ferramentas orientadas a esta atividade;
- apresentação do protótipo (testado) ao cliente, para que este possa efetuar sua avaliação; a partir desta avaliação, o cliente pode sugerir extensões ou reformular alguns requisitos de modo a que o software possa melhor corresponder às reais necessidades;
- repetição iterativa dos dois passos anteriores, até que todos os requisitos tenham sido formalizados ou até que o protótipo tenha evoluído na direção de um sistema de produção.

5.2. Métodos e Ferramentas de Prototipação

O desenvolvimento de um protótipo é uma atividade que deve ser realizada num tempo relativamente curto, mas de forma a representar fielmente os requisitos essenciais do software a ser construído. Para conduzir de modo eficiente esta atividade, já se dispõe de três classes de métodos e ferramentas, as quais são apresentadas a seguir:

- as **técnicas de quarta geração** (4GT), as quais englobam conjuntos de linguagens para geração de relatórios e de consulta a bancos de dados e derivação de aplicações e programas; a área de atuação destas técnicas é

atualmente limitada aos sistemas de informação comerciais, embora estejam aparecendo algumas ferramentas orientadas a aplicações de engenharia;
- os **componentes de software reusáveis**, os quais permitem a "montagem" do protótipo a partir dos "blocos de construção" (*building blocks*), dos quais não se conhece necessariamente o seu funcionamento interno, mas as suas funções e interfaces são dominadas pelo analista; o uso desta técnica só é possível a partir da construção (ou existência) de uma biblioteca de componentes reusáveis, catalogados para posterior recuperação; em alguns casos, um software existente pode ser adotado como "protótipo" para a construção de um novo produto, mais competitivo e eficiente, o que define uma forma de reusabilidade de software;
- as **especificações formais** e os **ambientes de prototipação**, que surgiram em substituição às técnicas baseadas em linguagem natural; as vantagens da utilização destas técnicas são, basicamente, a representação dos requisitos de software numa linguagem padrão, a geração de código executável a partir da especificação e a possibilidade de validação (da parte do cliente) de modo a refinar os requisitos do software.

6. ESPECIFICAÇÃO DOS REQUISITOS DE SOFTWARE

Como foi exposto no início do capítulo (item 2.4), a etapa de Análise de Requisitos culmina com a produção de um documento de Especificação dos Requisitos de Software. Este resultado pode vir na forma de um documento em papel utilizando linguagem natural ou gráfica, pode ter sido produzida com o auxílio de uma ferramenta CASE, ou ainda pode ser complementada ou expressa a partir de um protótipo construído nesta etapa.

Este documento é basicamente o resultado de um trabalho de representação das necessidades para a solução do problema analisado, devendo expressar os aspectos funcionais, informacionais e comportamentais do software a ser construído.

De modo a obter uma especificação eficiente, alguns princípios podem ser considerados:

- a separação entre funcionalidade e implementação;
- utilização de uma linguagem de especificação orientada ao processo;
- a especificação deve levar em conta o sistema do qual o software faz parte e o ambiente no qual o sistema vai operar;
- a especificação deve representar a visão que os usuários terão do sistema;
- uma especificação deve ser operacional, no sentido de que ela deve permitir, mesmo num nível de abstração elevado, algum tipo de validação;
- uma especificação deve ser localizada e fracamente acoplada, o que são requisitos fundamentais para permitir a realização de modificações durante a análise de requisitos.

A figura 5.3 apresenta uma proposta de estrutura para o documento de Especificação dos Requisitos do Software. O documento inicia com uma Introdução, que apresenta as principais metas do software, descrevendo o seu papel no contexto do sistema global. As informações prestadas nesta seção podem ser, inclusive, inteiramente extraídas do documento de Plano de Software.

1.0 - Introdução
2.0 - Descrição da Informação
- **2.1 - Diagramas de Fluxos de Dados**
- **2.2 - Representação da Estrutura dos Dados**
- **2.3 - Dicionários de Dados**
- **2.4 - Descrição das Interfaces do Sistema**
- **2.5 - Interfaces Internas**

3.0 - Descrição Funcional
- **3.1 - Funções**
- **3.2 - Descrição do Processamento**
- **3.3 - Restrições de Projeto**

4.0 - Critérios de Validação
- **4.1 - Limites de Validação**
- **4.2 - Classes de Testes**
- **4.3 - Expectativas de Resposta do Software**
- **4.4 - Considerações Especiais**

5.0 - Bibliografia
6.0 - Apêndices

Figura 5.3 - Estrutura do documento de Especificação de Requisitos.

A seção de Descrição da Informação deve apresentar informações sobre o problema que o software deve resolver. Os fluxos e a estrutura das informações devem ser descritos nesta seção (utilizando um formalismo adequado, como, por exemplo, os DFDs e os Dicionários de Dados), assim como os elementos relacionados às interfaces internas e externas do software (incluindo hardware, elementos humanos, outros softwares, etc...).

A seção seguinte, Descrição Funcional, apresenta as informações relativas às funções a serem providas pelo software, incluindo uma descrição dos processamentos envolvidos no funcionamento do software. Ainda nesta seção, devem ser apresentadas as restrições de projeto, detectadas nesta etapa.

A seção de Critérios de Validação, que apesar de ser a mais importante é aquela sobre a qual menor atenção é dispensada, vai estabelecer os parâmetros que permitirão avaliar se a implementação do software vai corresponder à solução desejada para o problema. Definir os critérios de validação significa ter um entendimento completo dos requisitos funcionais e de processamento de informação do software. Uma definição importante a ser encaminhada nesta seção é o conjunto de testes que deverá ser aplicado à implementação para que se tenha uma garantia do funcionamento do software nos moldes estabelecidos pela especificação dos requisitos.

Finalmente, devem ser registrados neste documento a lista de documentos relacionados ao software a ser desenvolvido (Plano de Software, por exemplo) e uma seção de Apêndices, onde podem ser apresentadas informações mais detalhadas sobre a especificação do software (descrição detalhada de algoritmos, diagramas, gráficos, etc...).

Um outro aspecto interessante, não apresentado na figura, refere-se a quando um software terá características de interatividade com usuários. Neste caso, é interessante que um Manual de Usuário (em versão preliminar) seja elaborado, o qual vai conduzir a duas conseqüências positivas: forçar o analista a enxergar o software do ponto de vista do usuário; permitir ao cliente uma avaliação do que será o software nesta etapa de desenvolvimento.

7. ANÁLISE ESTRUTURADA

A Análise Estruturada ou SSA (para *Structured System Analysis*) foi desenvolvida em meados dos anos 70 por Gane, Sarson e De Marco. A técnica SSA é baseada na utilização de uma linguagem gráfica para construir modelos de um sistema, incorporando

também conceitos relacionados às estruturas de dados. Os elementos básicos da Análise Estruturada são: o Diagrama de Fluxo de Dados, o Dicionário de Dados, as Especificações de Processos e as Técnicas de Estruturação de Bases de Dados.

Nos parágrafos que seguem, apresentaremos os principais fatores e notações que regem a Análise Estruturada, ilustrando por meio de exemplos, quando necessário.

7.1. Os Diagramas de Fluxos de Dados

Em primeira instância, um sistema computacional pode ser representado segundo as informações que ele manipula e o fluxo destas informações ao longo do sistema. À medida que se deslocam ao longo de um software, as informações de entrada vão sofrendo transformações no sentido da obtenção das informações de saída.

Um Diagrama de Fluxo de Dados (DFD) é uma técnica gráfica de representação que permite explicitar os fluxos de informação e as transformações que são aplicadas à medida que os dados se deslocam da entrada em direção à saída. Um DFD assume o formato de um esquema como o ilustrado na figura 5.4.

Num DFD, os processos ou atividades de transformação são caracterizados por **círculos** ou **bolhas** identificadas por uma expressão que descreva precisamente o processo ou o tipo de transformação realizada sobre os dados. O fluxo de dados é expresso por **setas rotuladas** que interligam os processos e que permitem indicar o caminho seguido pelos dados. **Retângulos rotulados** vão definir as origens ou destinos dos dados envolvidos no sistema, representando os geradores ou consumidores das informações. Os geradores ou consumidores das informações não são considerados objeto de análise do problema. Finalmente, uma **linha dupla** preenchida com um rótulo permite expressar depósitos de dados. A figura 5.5 apresenta os símbolos básicos de um DFD.

Esta representação pode ser adotada seja para o sistema seja para o software como elemento de um sistema, sendo possível definir diversas partições como forma de apresentar diferentes níveis de detalhamento do software ou do sistema.

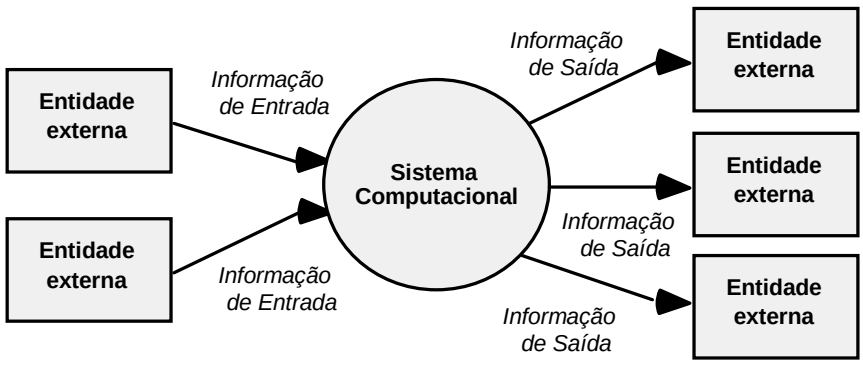


Figura 5.4 - Formato genérico de um DFD.

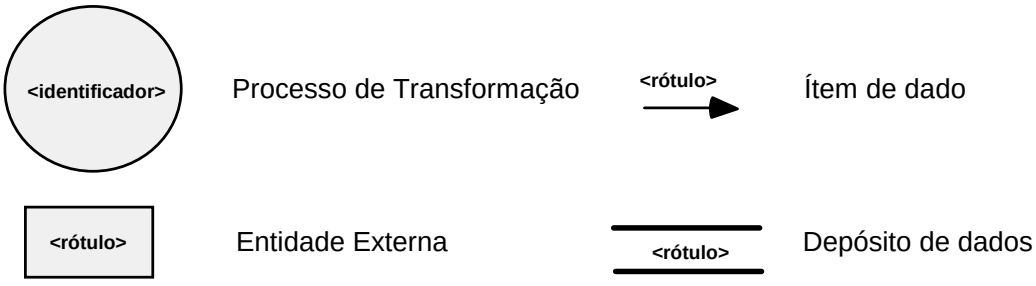


Figura 5.5 - Símbolos gráficos de um DFD.

O primeiro nível de representação, ou nível 0 do DFD é o **modelo fundamental do sistema** (ou **modelo de conteúdo**), utilizado para representar o elemento de software global como um único círculo e as informações de entrada e saída representadas pelas setas rotuladas entrando e saindo do círculo. Neste primeiro nível são representados por retângulos os elementos externos ao software que geram e consomem as informações. À medida que se vai evoluindo na análise, novos processos de transformação vão sendo acrescentados nos DFDs de níveis inferiores, correspondendo aos detalhamentos realizados em processos de um dado nível. A figura 5.6 ilustra esta situação.

Um exemplo de DFD, resultante da análise de um sistema de folha de pagamento é apresentado na figura 5.7. Quando múltiplos dados são necessários por um processo, um asterisco (*) é introduzido junto aos diferentes fluxos de dados (representando um relacionamento "E").

Um relacionamento do tipo "OU" pode também ser definido, se necessário, pela introdução de um sinal de adição (+) entre os fluxos de dados.

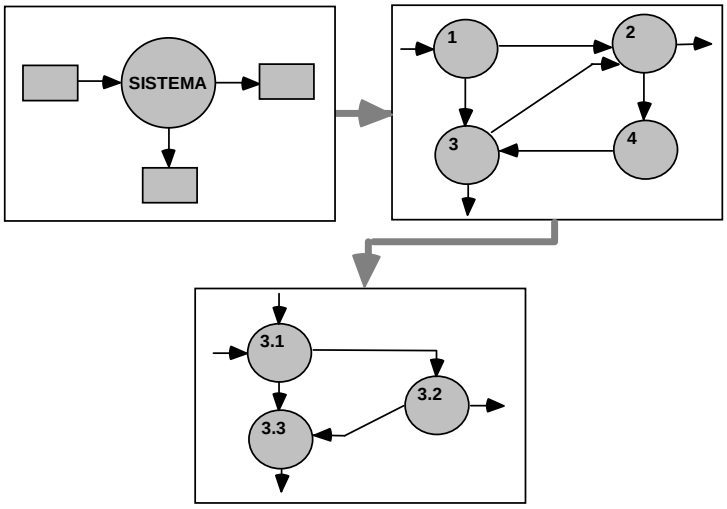


Figura 5.6 - Refinamento de um DFD.

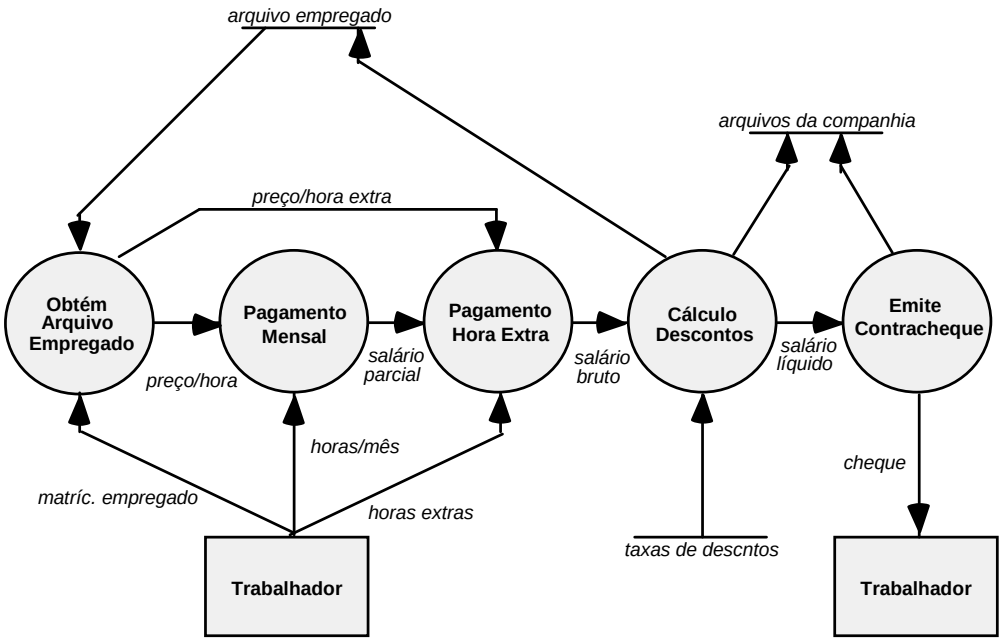


Figura 5.7 - DFD de um sistema de folha de pagamento.

É importante ressaltar que o DFD da figura 5.7 corresponde a um sistema de folha de pagamento, independente da forma como este é implementado (manual ou automático), o que sugere o caracter de abstração deste método.

Nesta primeira visão do sistema exemplo, detalhes relativos ao seu funcionamento não são apresentados (por exemplo, o que fazer se um erro na ficha de horas/mês é detectado?). Isto permite ter uma visão mais simples do sistema global. Caso alguns detalhes sejam considerados importantes, o DFD poderá ser refinado numa etapa posterior.

É preciso destacar aqui que um DFD não é um fluxograma. Um DFD exprime apenas o fluxo de dados, enquanto o fluxograma exprime o fluxo de controle. Num DFD, questões relacionadas a procedimentos de execução devem ser omitidas sempre; aspectos como decisões, malhas de controle, etc... não fazem parte de um DFD.

O objetivo principal de um DFD é expressar **quais** as transformações são aplicadas aos dados, sem preocupação em representar **como** são processadas estas transformações.

O primeiro passo para a construção de um DFD para um dado problema é a identificação das principais entradas e saídas. Informações de entrada e saída de menor importância (como, por exemplo, mensagens de erro) podem ser ignoradas num primeiro tempo. Uma vez identificadas as entradas, uma abordagem pode ser partir das entradas e identificar as principais transformações pelas quais estas passam, até atingirem (ou transformarem-se em) as saídas. Um outro princípio seria caminhar no sentido inverso, ou seja, das saídas em direção às entradas.

Os passos a seguir correspondem a algumas sugestões para a construção de um DFD:

- iniciar a construção do DFD, partindo das entradas para as saídas ou vice-versa, procurando definir as principais (poucas, em princípio) transformações de dados; refinando, em seguida, estas transformações em conjuntos de transformações mais específicas;
- evitar sempre a expressão de fluxo de controle; eliminar da análise qualquer raciocínio que sugira decisão ou malhas;
- rotular as setas com os nomes dos dados apropriados; entradas e saídas de cada transformação devem ser cuidadosamente identificadas;
- utilizar os operadores * e + sempre que necessário;
- quando possível, traçar diversos DFDs de um mesmo problema, ao invés de adotar o primeiro construído.

7.2. O Dicionário de Dados

Num DFD, os dados são identificados por um rótulo único, que represente claramente o significado da informação. Por outro lado, a definição da estrutura de dados que vai representar a informação não é feita a nível do DFD. Para isto, é construído um **Dicionário de Dados**, o qual se constitui de um depósito de todos os fluxos de dados explicitados no DFD associado. Os componentes de um fluxo de dados do DFD podem ser então especificados no dicionário de dados, assim como a estrutura dos eventuais arquivos definidos no diagrama. As notações mais diversas podem ser utilizadas para definir as estruturas dos dados, sendo que uma proposta de notação possível inclui as seguintes informações:

- ***nome***, o identificador principal do item de dados, do depósito de dados ou de uma entidade externa;
- ***alias***, eventualmente outros nomes utilizados para o mesmo item;
- ***utilização***, em que contexto (onde e como) o item de informação é utilizado;
- ***descrição***, uma notação que permita explicitar o conteúdo do item;

- ***informações complementares*** a respeito do item de dados, como valores iniciais, restrições, etc...

Atualmente, a obtenção do dicionário de dados é resultante do uso de uma ferramenta CASE, sendo que esta pode inclusive checar a consistência da especificação com relação a esta aspecto da definição. Por exemplo, caso o analista nomeie um item de dados recém derivado com um identificador que já faça parte do dicionário, a ferramenta devolve uma mensagem de erro indicando a duplicação do identificador. Outro aspecto positivo do uso de uma ferramenta CASE é a geração de algumas informações de forma automatizada. A informação ***utilização*** é um exemplo disto, uma vez que ela pode ser derivada a partir dos fluxos de dados.

Apresentamos abaixo o dicionário de dados definido para alguns dos itens de dados apresentados no DFD da figura 5.7.

```
nome        : matrícula_empregado
alias       : nenhum
utilização  : obtém arquivo empregado (entrada)
descrição   : matricula_empregado = dígito + dígito + dígito + dígito

nome        : preço/hora
alias       : nenhum
utilização  : obtém arquivo empregado (entrada)
descrição   : preço/hora = valor em dolar
```

Para formalizar a descrição dos itens de dados, utiliza-se um conjunto de operadores que permita compor representações que poderão ser mapeadas futuramente em estruturas de dados de uma dada linguagem de programação. Alguns dos operadores e construtores utilizados são:

`x = a + b`	**x** possui os elementos de dados **a** e **b**
`x = [a | b]`	**x** possui **a** ou **b** (escolha)
`x = (a)`	**x** possui um elemento de dados opcional **a**
`x = {a}`	**x** possui de **zero** a mais ocorrências de **a**
`x = y(a)`	**x** possui **y** ou mais ocorrências de **a**
`x = (a)z`	**x** possui **z** ou menos ocorrências de **a**
`x = y(a)z`	**x** possui entre **y** e **z** ocorrências de **a**

7.3. A Descrição de Processos

Um outro aspecto importante na análise de requisitos é a Especificação de Processos. O objetivo da especificação de processo é auxiliar o analista a descrever, de forma precisa, o comportamento de alguns componentes do sistema (ou processos) definidos nos DFDs de nível mais baixo. A especificação de processos pode ser realizada segundo diversas técnicas, mas uma das mais interessantes é o texto estruturado, o qual é baseado num grupo limitado de verbos e substantivos organizados de modo a representar um compromisso entre a legibilidade e a precisão da especificação. O texto estruturado é baseado, principalmente, nos seguintes elementos:

- um conjunto limitado de verbos de ação (por exemplo, calcular, encontrar, imprimir, etc...);
- estruturas de controle já conhecidas da programação estruturada (IF-THEN-ELSE, DO-WHILE, CASE, etc...);
- elementos de dados definidos no Dicionário de Dados.

Um exemplo de especificação de processo é apresentada abaixo, representando um dos processos definidos no DFD da figura 5.7.

```
PROCESSO Calcula_Desconto

IF Empregado é isento de descontos
     THEN Salário Líquido é igual a Salário Bruto
     ELSE
        IF Salário Bruto é maior que X
           THEN Desconto é de 25%
           ELSE
              IF Salário Bruto é maior que Y
                 THEN Desconto é de 20%
                 ELSE Desconto é de 15%

FIM Calcula_Desconto
```

Outras técnicas podem ser utilizadas para a especificação de um processo, como os fluxogramas, as tabelas de decisão, etc...

7.4. Representação da Relação entre os Dados

Um quarto elemento relacionado à análise estruturada objetiva o estabelecimento de uma relação entre os diferentes dados definidos no DFD e no dicionário de dados. Uma forma de representar estas relações é através dos diagramas Entidade-Relação. É uma ferramenta gráfica que permite definir as relações entre as diferentes entidades definidas num dado sistema. Fugindo um pouco do exemplo da figura 5.7, é apresentado, na figura 5.8, um exemplo de diagrama Entidade-Relação para um sistema de tráfego aéreo.

Como se pode notar no exemplo, os retângulos representam as entidades do sistema e os losangos as possíveis relações entre estas.

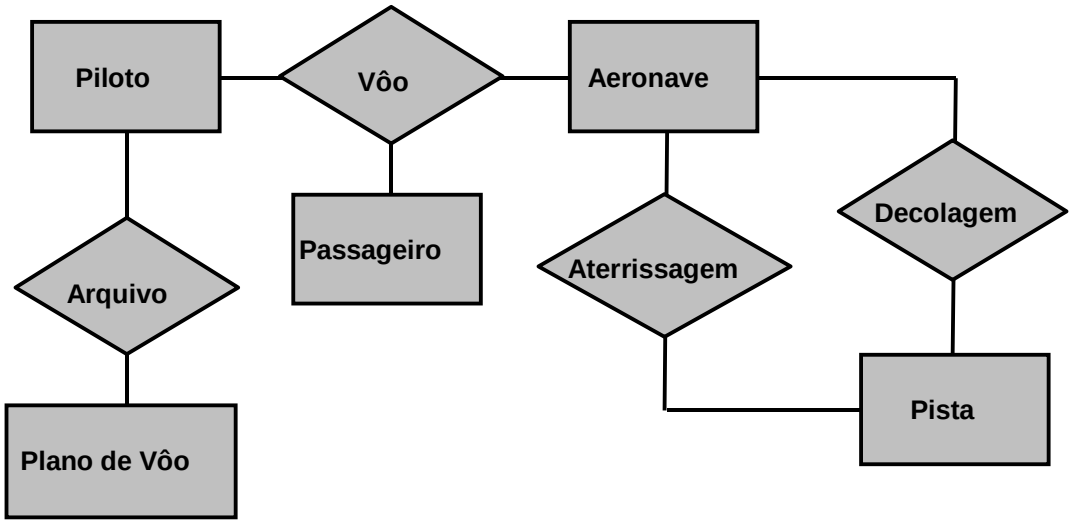


Figura 5.8 - Exemplo de Diagrama Entidade-Relação.

8. TÉCNICAS DE ANÁLISE

8.1. Técnicas Orientadas Estado

Apesar de apresentarem aspectos interessantes no que diz respeito ao desenvolvimento da tarefa de Análise de Requisitos, as ferramentas da técnica de Análise Estruturada de Sistemas não permitem cobrir todos os aspectos importantes de determinadas classes de sistemas.

Um exemplo disto são os sistemas reativos e de tempo-real, onde a ordenação de eventos e os aspectos de temporização podem assumir uma importância fundamental no entendimento do problema a ser resolvido.

Por esta razão, nestes casos, é, sem dúvida, mais interessante realizar a formalização do problema utilizando uma técnica baseada no conceito de estados. Os parágrafos a seguir apresentarão algumas das técnicas orientadas a estados que são úteis na etapa de Análise de Requisitos.

8.1.1. As Tabelas de Decisão

As tabelas de decisão são utilizadas para expressar decisões lógicas de alta complexidade. Estas são caracterizadas por quatro quadrantes: o quadrante de condições, o quadrante de ações, as entradas de condições e as entradas de ações.

O quadrante de condições contém todas as condições que serão examinadas na análise de um problema. As entradas de condições servem para combinar as condições com as regras de decisão estabelecidas na parte superior da tabela.

O quadrante de ações define todas as ações que deverão ser tomadas como resposta às regras de decisão. As entradas de ação relaciona as regras de decisão às ações.

A figura 5.9 ilustra o formato padrão de uma tabela de decisão.

A figura 5.10 apresenta um exemplo de uma tabela de decisões com entradas limitadas. As entradas possíveis são ***S***, ***N***, **-** e ***X***, que denotam, respectivamente, SIM, NÃO, TANTO FAZ e REALIZE A AÇÃO. Como se pode verificar na figura, um determinado empréstimo é autorizado nas seguintes situações: se o limite de crédito não foi ultrapassado, se o limite foi ultrapassado mas a experiência de pagamentos é positiva, se uma liberação especial de crédito pode ser obtida. Em outras condições, a ordem é rejeitada.

As entradas ***S***, ***N*** e **–** em cada coluna caracterizam uma *regra de decisão*. Caso duas ou mais regras tenham as entradas idênticas, a tabela é considerada *ambígua*.

	Regras de Decisão			
	Regra 1	*Regra 2*	*Regra 3*	*Regra 4*
Quadr. de Condições		*Entradas de Condições*		
Quadrante de Ações		*Entradas de Ações*		

Figura 5.9 - Formato geral de uma tabela de decisões.

	Regras de Decisão			
	Regra 1	*Regra 2*	*Regra 3*	*Regra 4*
Limite de crédito é satisfatório	***S***	***N***	***N***	***N***
Experiência de pagamentos é favorável	**–**	***S***	***N***	***N***
Liberação especial é obtida	**–**	**–**	***S***	***N***
Autorize empréstimo	***X***	***X***	***X***	
Rejeite empréstimo				***X***

Figura 5.10 - Uma tabela de decisões com entradas limitadas.

Caso duas ou mais regras idênticas conduzam às mesmas ações, elas são consideradas *redundantes*. Caso elas conduzam a ações diferentes, elas são consideradas *contraditórias*. Regras contraditórias não são necessariamente indesejáveis; Elas podem ser utilizadas para expressar o não-determinismo de um dado problema.

Uma tabela de decisões é considerada completa se todas as possíveis combinações de condições podem ser associadas a uma ação. Numa tabela com *N* entradas de condição, vão existir 2^N possíveis combinações. A não especificação de uma ou mais combinações vai resultar numa tabela de decisões incompleta.

8.1.2. As tabelas de transição

As tabelas de transição são utilizadas para especificar mudanças no estado de um sistema como resultado da ação de eventos característicos. O estado de um sistema caracteriza as condições de todas as entidades do sistema num dado instante. Para um estado S_i, a ocorrência de uma condição C_j vai provocar a passagem ou transição ao estado S_k.

Um exemplo de tabela de transição é apresentada na figura 5.11, que representa um sistema composta de dois estados (S_0,S_1) nos quais duas entradas (*a* e *b*) podem ocorrer. Como se pode notar na figura, as entradas rotuladas por *a* fazem com que o sistema permaneça no seu estado atual (S_0 ou S_1). As entradas rotuladas por *b* vão causar uma transição para S_1, se o sistema estiver em S_0 ou para S_0 se o sistema estiver em S_1.

8.1.3. Os diagramas de transição de estado

Uma técnica também interessante é o **diagrama de transição de estados**. Neste diagrama, *retângulos* ou *círculos* representam os **estados** de um processo ou sistema. A passagem ou **transição** de um estado para o outro é representada, neste diagrama, por *setas* que definem a direção da transição. Tanto os estados quanto as transições são rotuladas; os primeiros indicam nomes para os estados; os rótulos associados às setas definem os eventos que provocam as transições de estados, assim como as ações resultantes destas transições. A figura 5.12 ilustra um diagrama de transição de estado para o sistema representado pela tabela de transições da figura 5.6.

ESTADO CORRENTE	*ENTRADA CORRENTE*	
	a	*b*
S0	***S0***	***S1***
S1	***S1***	***S0***

Figura 5.11 - Uma tabela de transição com entradas limitadas.

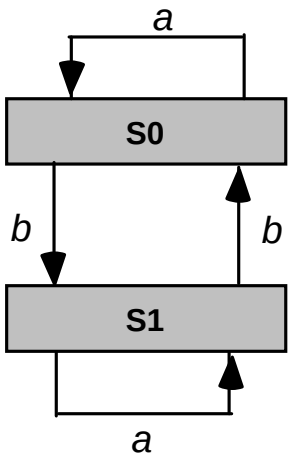


Figura 5.12 - Diagrama de Transição de Estados.

8.2. SADT

Uma das ferramentas mais difundidas para a realização da análise de requisitos é, sem dúvida, o **SADT** ou ***Structured Analysis and Design Technique***. A técnica SADT vem sendo utilizada há mais de 15 anos por um grande conjunto de organizações em todo o mundo.

O SADT foi concebido de modo a apresentar as principais características desejáveis num modelo orientado à Análise e Especificação de Requisitos. Dentre estas características, podemos relacionar:

- o uso de uma linguagem gráfica de comunicação;
- a possibilidade de representação de um problema obedecendo a uma política de decomposição estruturada, de cima para baixo (top-down);
- limitar a quantidade de informações podendo conter num dado nível de representação de um problema, segundo os limites que a mente humana pode absorver;
- permitir a representação de um problema segundo dois pontos de vista (o ponto de vista das atividades e ponto de vista dos dados) o que satisfaz a um dos princípios importantes da Análise de Requisitos que é o princípio da projeção.

Um modelo **SADT** consiste de um conjunto ordenado de diagramas, onde cada diagrama é traçado numa página única. Um diagrama é composto em média de três a seis blocos, interconectados por um conjunto de arcos dirigidos.

8.2.1. A Linguagem de Representação Gráfica

O princípio que rege o modelo SADT é o fato de que o conhecimento sobre o mundo e seus sistemas é fundamento sobre **coisas** e **acontecimentos**, ou melhor, de **objetos** e **eventos**, ou ainda, de **dados** e **atividades**.

A linguagem de representação proposta no SADT é baseada na construção de um conjunto de diagramas, onde os elementos básicos são os **blocos** e as **setas**. O tipo de informação representado pelas setas depende do tipo de diagrama construído no modelo.

Existem basicamente dois tipos de diagramas definidos no modelo SADT: os actigramas e os datagramas. Nos **actigramas**, os blocos designam as atividades relacionadas ao sistema sob análise, sendo que os arcos representam os fluxos de dados entre as atividades. Os actigramas são, de certo modo, um equivalente dos diagramas de fluxos de dados dentro do modelo SADT. Nos **datagramas**, os blocos especificam objetos de dados e os arcos as atividades ou ações definidas sobre estes dados. Os actigramas e os datagramas têm, estabelecida entre si, uma relação de dualidade.

Na prática, os actigramas são utilizados com muito mais freqüência que os datagramas. Entretanto, o uso de datagramas assume um nível elevado de importância por duas razões principais:

- para especificar todas as atividades relacionadas com um dado objeto de dado;
- para verificar a coerência de um modelo SADT pela construção de datagramas a partir de actigramas.

A forma geral dos blocos de um actigrama e de um datagrama é apresentada na figura 5.13. Em 5.13.(a) é mostrado o bloco relacionado a um actigrama. Na sua forma geral, os possíveis fluxos de dados são os de entrada, de saída, de controle e o mecanismo. As **saídas** de um bloco de actigrama podem representar entradas ou controles de um outro bloco do mesmo actigrama e deverão, desta forma, ser conectadas a outros blocos do actigrama ou ao ambiente externo.

As **entradas** e **controles** de um bloco de actigrama deverão, similarmente, ser saídas de outros blocos ou dados do ambiente externo. Os **controles** de uma atividade são dados utilizados na atividade mas que não são modificados por ela. Os **mecanismos** são os elementos necessários à realização daquela atividade, como por exemplo, os processadores.

Um exemplo de representação por actigrama é apresentado na figura 5.14. No exemplo, o processo de escrita de um texto, a **atividade** é representada por um bloco rotulado pelo verbo **redigir**. O dado de entrada da atividade é a idéia e a saída é o texto.

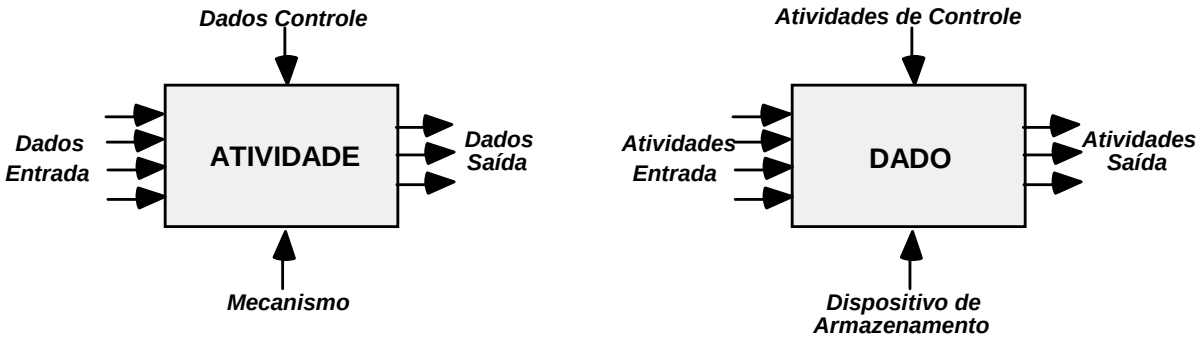


Figura 5.13 - Blocos de composição de um actigrama (a) e de um datagrama (b).

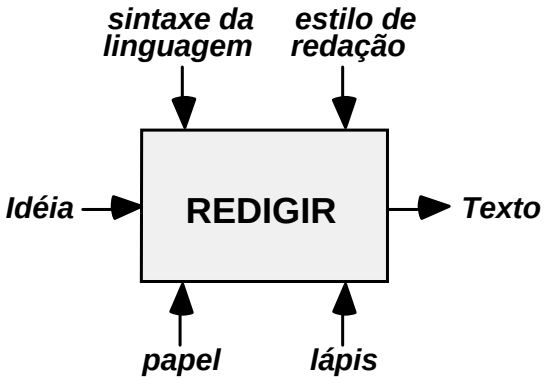


Figura 5.14 - Exemplo de representação em SADT: o processo de redação de um texto.

Como controle, tem-se a **sintaxe da linguagem** e o **estilo de redação**. Finalmente, os **mecanismos** considerados são o **lápis** e o **papel**.

Num datagrama, as entradas de um bloco são as atividades que geraram o dado representado pelo bloco, as saídas são as atividades que vão utilizar o objeto de dado. Os controles são as condições nas quais o objeto de dado será ativado. Os mecanismos são os elementos necessários ao armazenamento do dado.

8.2.2. Refinamento de Diagramas

De forma similar ao modelo do Diagrama de Fluxo de Dados, o nível de complexidade de um sistema sob análise pode conduzir a construção de um conjunto de diagramas SADT, onde um diagrama construído num dado nível representa um refinamento de um bloco de um diagrama de nível superior.

Com exceção do primeiro diagrama criado, os demais diagramas correspondem a detalhamentos de um bloco de diagrama de nível superior – o **bloco pai** o qual pertence ao **diagrama pai**. Os diagramas que representam os detalhamentos de blocos de um dado diagrama são denominados **diagramas filhos**.

A figura 5.15 apresenta um exemplo de refinamento de um bloco de um actigrama. No exemplo, o bloco **A1**, que é conectado ao sistema pelos fluxos de dados ***I1*** (entrada), ***O1*** e ***O2*** (saídas), ***C1*** (controle) e ***m*** (mecanismo), é refinado em três blocos, no caso **A11**, **A12** e **A13**. Um aspecto interessante de ser observado é o aparecimento, além dos fluxos de dados relativos aos blocos representados no novo actigrama, dos fluxos de dados originais do bloco **A1**, o que permite verificar a coerência do refinamento efetuado.

Como no caso dos DFDs, o refinamento dos diagramas não é infinito, mas composto de tantos níveis quantos se julguem necessários para representar precisamente o sistema. O processo de refinamento termina quando considera-se que o processo pode ser descrito precisamente numa outra linguagem (por exemplo, texto estruturado, diagramas de estado, etc...) e que o espaço ocupado pela descrição não ultrapasse uma única página.

8.2.3. Referência aos Diagramas

A referência de cada diagrama é localizada no canto esquerdo inferior da folha de cada diagrama, sendo que cada referência é definida como sendo um **nó**. O nó é especificado pela letra **A**, seguida de um número que indique o bloco pai que está sendo detalhado. Cada bloco num diagrama apresenta um número de referência escrito no seu canto superior direito, o qual o identifica de forma única para aquele diagrama. A figura 5.16 ilustra a forma de referência aos diagramas e aos blocos num diagrama.

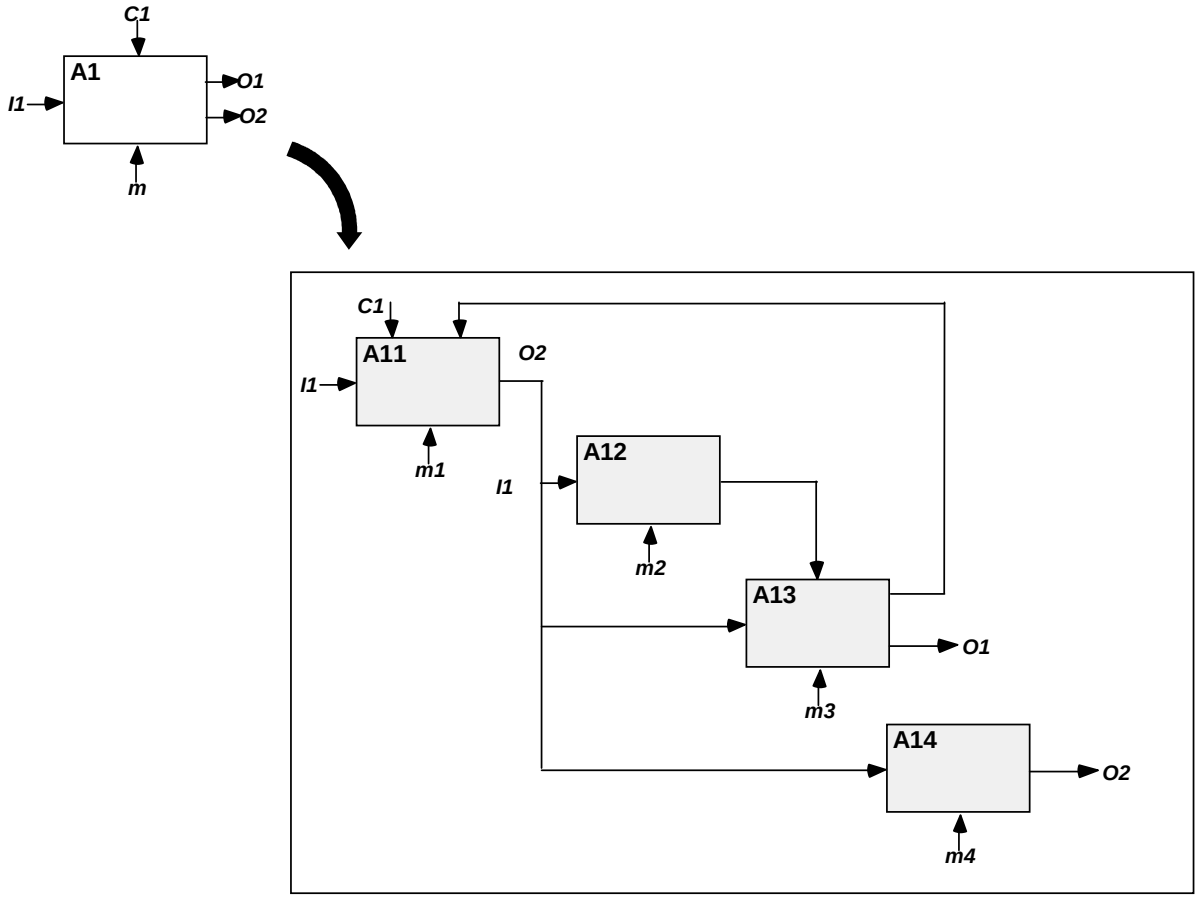


Figura 5.15 - Exemplo de refinamento de um bloco num actigrama.

A representação da estrutura hierárquica do modelo completo do sistema pode ser representada em duas formas distintas:

- um índice de nós, como mostra a figura 5.17;
- uma tabela de nós, como mostrado na figura 5.18.

8.2.4. Etiquetagem das Setas

Uma regra importante a ser seguida, a qual reflete fortemente na consistência do modelo em realização, é que, durante o refinamento de uma caixa, nenhuma adição ou supressão de informação deve ser feita, particularmente no que diz respeito às interfaces (entradas, saídas, controles e mecanismos) e ao nome do bloco pai.

As setas que apresentam as suas extremidades livres num dado diagrama correspondem às interfaces do bloco pai. Após a verificação de que todas as setas com extremidades livres correspondem à totalidade das interfaces do bloco pai, elas devem ser rotuladas com as letras ***I***, ***C***, ***O*** e ***M*** (designando uma **entrada**, um **controle**, uma **saída** ou um **mecanismo**, respectivamente), sendo que a letra deve ser seguida de um número que indique a sua posição geométrica relativa no bloco pai. A ordenação das setas é feita, num diagrama, de cima para baixo e da esquerda para e direita. Uma seta rotulada por **C3** corresponde ao terceiro controle do bloco pai.

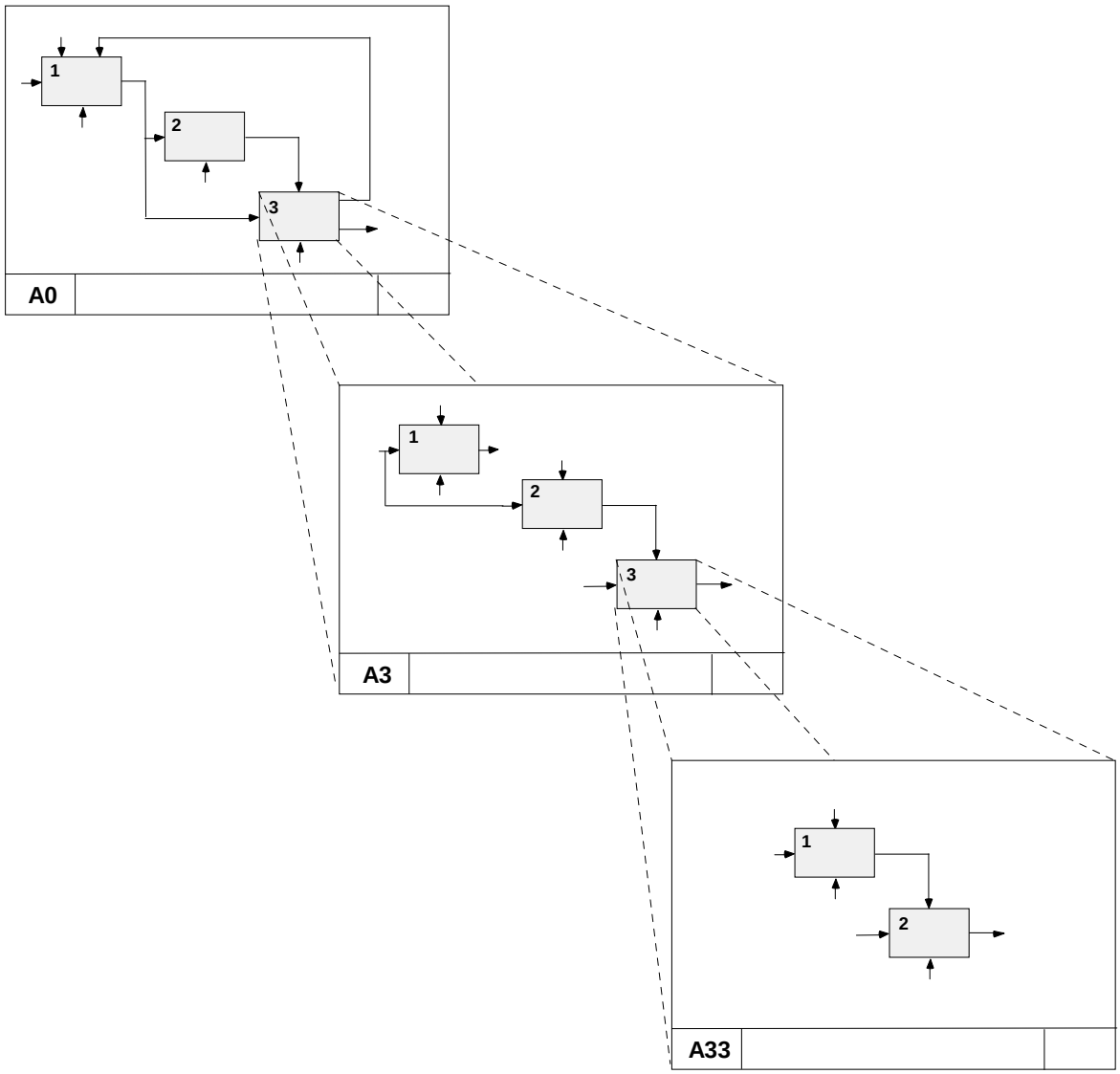
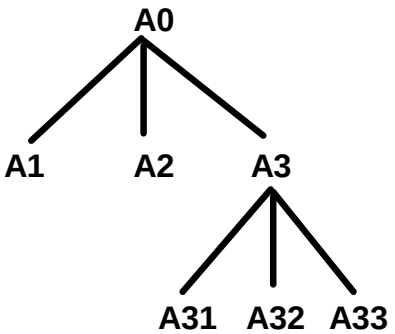


Figura 5.16 - Ilustração das referências aos diagramas e aos blocos.

Por outro lado, quando é realizado o refinamento de um dado bloco (bloco pai), as interfaces não serão necessariamente vistas da mesma forma que no bloco pai. Um exemplo disto é apresentado à figura 5.19, onde um controle (**C2**) no bloco pai corresponde a uma entrada de um dos blocos do diagrama filho.



A0 título diagrama
A1 título diagrama
A2 título diagrama
A3 título diagrama
A31 título diagrama
A32 título diagrama
A33 título diagrama

Figura 5.17 - Exemplo de um Índice de Nós.

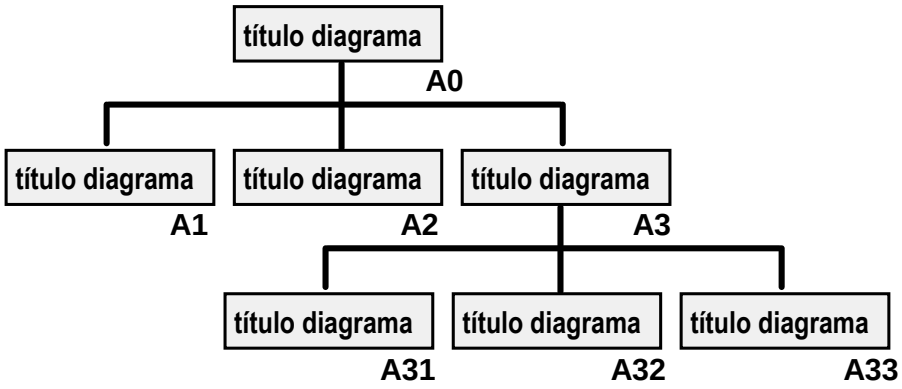


Figura 5.18 - Exemplo de uma Tabela de Nós.

8.2.5. A Análise de um Diagrama

A leitura de um diagrama deve ser efetuada segundo um **caminho principal** que parte da seta de entrada ou de controle não conectada mais importante e chega na seta de saída não conectada mais importante. As seguintes regras devem ser observada quando da análise de uma diagrama:

① examinar apenas os blocos do diagrama sob análise;
② retornar ao diagrama pai para observar como as setas I, C, O e M estão conectadas no bloco pai e qual a sua configuração no diagrama filho;
③ identificar a seta de entrada (ou de controle) e a seta de saída mais importantes do diagrama;
④ examinar o diagrama percorrendo o caminho principal;

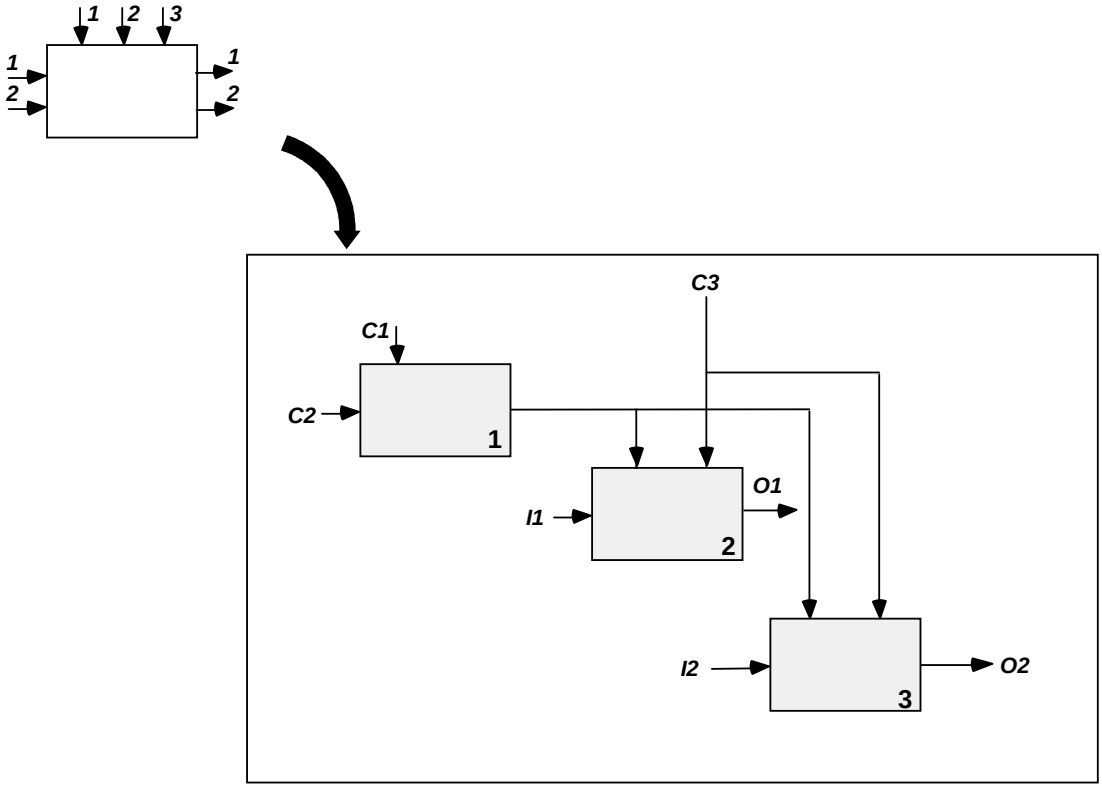


Figura 5.19 - Exemplo de etiquetagem das setas num diagrama.

⑤ percorrer o diagrama, observando como as diferentes setas intervêm em cada caixa e definindo os caminhos secundários;
⑥ ler as notas associadas ao diagrama para um melhor entendimento.

8.2.6. Normas de representação de um diagrama

Os diagramas deverão ser apresentados em folhas de formato normalizado, observando a configuração mostrada na figura 5.20.

Dentro de um diagrama (pai), um bloco (pai) é referenciado da seguinte forma (ver figura 5.21):

- o número do bloco é posicionado em baixo à direita de cada bloco;
- a referência do diagrama filho que representa o refinamento do bloco é posicionado abaixo do bloco (esta referência representa a página do documento de diagramas, onde o refinamento do bloco está apresentado; a ausência de referência implica na não existência de um refinamento daquele bloco).

UFSC	Nome do Projeto: Autor: Data: Versão:		ETAPAS	LEITOR	DATA
			proposta		
			aprovação		
			publicação		

Nó:	Título:	Referência:
Nº do nó	TÍTULO DO BLOCO PAI	Ref. do Diag. filho

Figura 5.20 - Folha padrão para construção do diagrama.

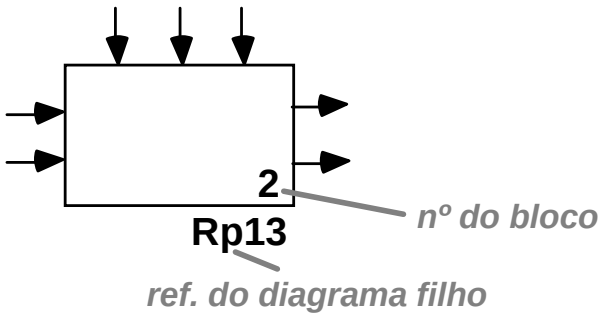


Figura 5.21 - Referências dos blocos num diagrama.

Engenharia de Software

Capítulo 6

Projeto de Software

1. Introdução

A etapa de Projeto é o passo inicial de uma nova fase do ciclo de vida do software, a fase de Desenvolvimento. O Projeto consiste na aplicação de um conjunto de técnicas e princípios, de modo a definir um sistema num nível de detalhe suficiente à sua realização física. A tarefa do projetista nada mais é do que produzir um modelo de representação do software que será implementado. Nesta etapa, os requisitos definidos na etapa anterior devem servir de referência para a obtenção da representação do software.

Neste capítulo, serão apresentados os principais aspectos relativos ao projeto de software, como etapa fundamental para a obtenção de um software de qualidade. Serão apresentadas algumas das técnicas clássicas de projeto, assim como a aplicação de técnicas de projeto a aspectos mais inovadores do desenvolvimento de software como as interfaces homem-máquina e os sistemas de tempo-real.

2. O Processo de Projeto

2.1. Projetos Preliminar e Detalhado

A etapa de projeto é caracterizada pelo conjunto de atividades que vai permitir traduzir os requisitos definidos na etapa anterior em uma representação do software a ser construído. Na sua forma mais clássica, o primeiro resultado obtido no projeto é uma visão da estrutura do software em termos de componentes, sendo que, a partir de procedimentos de refinamento, chega-se a um nível de especificação bastante próxima da codificação do programa.

Do ponto de vista do gerenciamento do processo de desenvolvimento, a etapa de projeto é conduzida basicamente em dois principais estágios:

- o ***projeto preliminar***, o qual permite estabelecer, a partir dos requisitos, a arquitetura do software e da informação relacionada;
- o ***projeto detalhado***, o qual permite aperfeiçoar a estrutura do software e definir representações algorítmicas de seus componentes.

No contexto dos projetos preliminar e detalhado, uma conjunto de atividades técnicas de projeto são desenvolvidas. Num ponto de vista mais genérico, pode-se destacar os projetos *arquiteturais*, *procedimentais* e *de dados*. Em projetos mais recentes, e, particularmente, naqueles envolvendo interatividade com o usuário, o *projeto de interfaces* tem assumido um papel de fundamental importância, sendo considerada uma atividade do mesmo nível dos demais projetos.

2.2. Características desejáveis num projeto de software

A obtenção de um software de boa qualidade está relacionada ao sucesso de cada uma das etapas do seu desenvolvimento. No que diz respeito à etapa de projeto, é possível detectar algumas das características desejáveis:

- deve apresentar um organização hierárquica, definida de modo a utilizar racionalmente todos os elementos de software;
- deve basear-se em princípios de modularidade, ou seja, propor um particionamento do software em elementos que implementem funções ou subfunções do software;
- deve conduzir a uma definição em módulos com alto grau de independência, como forma de facilitar as tarefas de manutenção ao longo da vida do software;
- deve ser fiel à especificação dos requisitos definida na etapa anterior.

3. Aspectos fundamentais do projeto

Durante esta etapa, é importante que alguns conceitos sejam levados em conta para que se possa derivar um projeto contendo as características citadas acima. As seções que seguem discutirão alguns destes conceitos.

3.1. Abstração

O princípio de abstração está fortemente relacionado com as características de modularidade que um software pode apresentar. Quando se desenvolve um software que vai apresentar estas características, é comum que suas representações sejam feitas considerando vários níveis de abstração.

No que diz respeito à linguagem de representação, nos níveis mais altos de abstração, a linguagem utilizada é bastante orientada à aplicação e ao ambiente onde o software irá ser executado. À medida que se vai "descendo" nos níveis de abstração, a linguagem de representação vai se aproximando de questões de implementação, até que, no nível mais baixo, a solução é representada de modo a que possa ser derivada diretamente para uma linguagem de implementação.

Na realidade, o próprio processo de Engenharia de Software constitui-se num aprimoramento de níveis de abstração. Na Engenharia de Sistemas, por exemplo, parte das tarefas do sistema a desenvolver são atribuídas ao software, como elemento constituinte de um sistema computacional. Na Análise de Requisitos, a solução em termos de software é apresentada de forma conceitual. Durante o Projeto, partindo do Projeto Preliminar para o Projeto Detalhado, múltiplos níveis de abstração vão sendo definidos, aproximando-se cada vez mais da Implementação, que é o nível mais baixo de abstração.

3.2. Refinamento

O refinamento surgiu na forma de uma técnica de projeto (a técnica de Refinamentos Sucessivos, proposta por Wirth em 1971). Esta técnica sugere como ponto de partida a definição da arquitetura do software a ser desenvolvido, sendo que esta vai sendo refinada sucessivamente até atingir níveis de detalhes procedimentais. Este processo dá origem a uma hierarquia de representações, onde uma descrição macroscópica de cada função vai sendo decomposta passo-a-passo até se obter representações bastante próximas de uma linguagem de implementação. Este processo é bastante similar ao processo utilizado em diversas técnicas de Análise de Requisitos (como o DFD e o SADT), sendo que a diferença fundamental está nos níveis de detalhamento atingidos nesta etapa.

3.3. Modularidade

O conceito de modularidade tem sido utilizado já há bastante tempo, como forma de obtenção de um software que apresente algumas características interessantes como a facilidade de manutenção. Este conceito apareceu como uma solução aos antigos softwares "monolíticos", os quais representavam grandes dificuldades de entendimento e, conseqüentemente para qualquer atividade de manutenção. A utilização do conceito de modularidade oferece resultados a curto prazo, uma vez que, ao dividir-se um grande problema em problemas menores, as soluções são encontradas com esforço relativamente menor. Isto significa que, quanto maior o número de módulos definidos num software, menor será o esforço necessário para desenvolvê-lo, uma vez que o esforço de desenvolvimento de cada módulo será menor. Por outro lado, quanto maior o número de módulos, maior será o esforço no desenvolvimento das interfaces, o que permite concluir que esta regra deve ser utilizada com moderação.

Finalmente, é importante distinguir o conceito de modularidade de projeto com o de modularidade de implementação. Nada impede que um software seja projetado sob a ótica da modularidade e que sua implementação seja monolítica. Em alguns casos, como forma de evitar desperdício de tempo de processamento e de memória em chamadas de procedimentos (salvamento e recuperação de contexto), impõe-se o desenvolvimento de um programa sem a utilização de funções e procedimentos. Ainda assim, uma vez que o projeto seguiu uma filosofia de modularidade, o software deverá apresentar alguns benefícios resultantes da adoção deste princípio.

3.4. A Arquitetura de Software

O conceito de arquitetura de software está ligado aos dois principais aspectos do funcionamento de um software: a estrutura hierárquica de seus componentes (ou módulos) e as estruturas de dados. A arquitetura de software resulta do desenvolvimento de atividades de particionamento de um problema, encaminhadas desde a etapa de Análise de Requisitos. Naquela etapa, é dado o pontapé inicial para a definição das estruturas de dados e dos componentes de software. A solução é encaminhada ao longo do projeto, através da definição de um ou mais elementos de software que solucionarão uma parte do problema global.

É importante lembrar que não existe técnica de projeto que garanta a unicidade de solução a um dado problema. Diferentes soluções em termos de arquitetura podem ser derivadas a partir de um mesmo conjunto de requisitos de software. A grande dificuldade concentra-se em definir qual a melhor opção em termos de solução.

3.5. Hierarquia de Controle

A hierarquia de controle nada mais é do que a representação, usualmente sob a forma hierarquizada, da estrutura do software no que diz respeito aos seus componentes. O objetivo não é apresentar detalhes procedimentais ou de sequenciamento entre processos, mas de estabelecer as relações entre os diferentes componentes do software, explicitando os níveis de abstração (refinamento) aos quais eles pertencem.

O modo mais usual de apresentar a hierarquia de controle utiliza uma linguagem gráfica, normalmente em forma de árvore, como mostra a figura 6.1. Com relação à estrutura de controle é importante apresentar algumas definições de "medição".

Utilizando a figura 6.1 como referência, é possível extrair alguns conceitos:

- a ***profundidade***, a qual está associada ao número de níveis de abstração definidos para a representação do software;
- a ***largura***, que permite concluir sobre a abrangência do controle global do software;

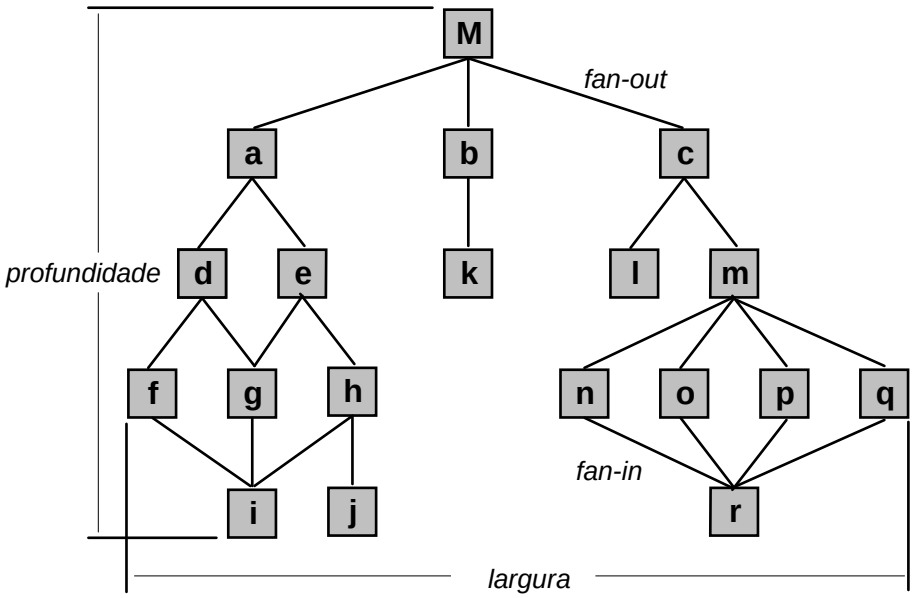


Figura 6.1 - Representação da hierarquia de controle.

- o ***fan-out***, que permite definir a quantidade de módulos controlados por um dado módulo;
- o ***fan-in***, que indica quantos módulos controlam um dado módulo.

Ainda, é possível estabelecer as relações de controle entre os módulos. Um módulo que exerce controle sobre outros módulos é denominado o módulo ***superordenado***. Um módulo que é controlado por outro é dito um módulo ***subordinado*** a ele.

3.6. A Estrutura dos Dados

A estrutura dos dados representa os relacionamentos lógicos entre os diferentes elementos de dados individuais. À medida que o projeto se aproxima da implementação, esta representação assume fundamental importância, já que a estrutura da informação vai exercer um impacto significativo no projeto procedimental final.

A estrutura dos dados define a forma como estão organizados, os métodos de acesso, o grau de associatividade e as alternativas de processamento das informações. Apesar de que a forma de organizar os elementos de dados e a complexidade de cada estrutura dependam do tipo de aplicação a desenvolver e da criatividade do projetista, existe um número limitado de componentes clássicos que funcionam como blocos de construção (building-blocks) de estruturas mais complexas. A figura 6.2 ilustra alguns destes construtores. Os construtores ilustrados podem ser combinados das mais diversas maneiras para obter estruturas de dados com grau de complexidade relativamente complexo.

3.7. Procedimentos de Software

Os procedimentos de software têm por objetivo expressar em detalhes a operação de cada componente de software (módulo) individualmente. Os aspectos a serem expressos nos procedimentos de software são o processamento da informação, o seqüenciamento de eventos, os pontos de decisão, as operações repetitivas e, eventualmente (se houver necessidade) algumas estruturas de dados.

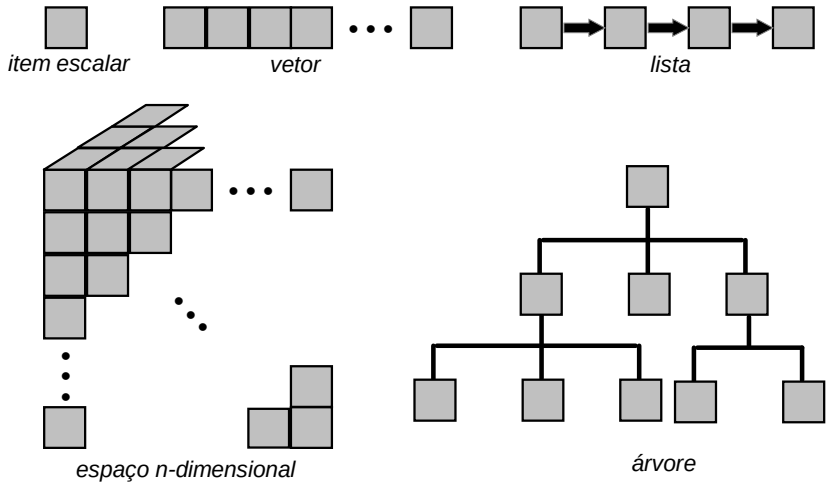


Figura 6.2 - Alguns construtores clássicos de estruturas de dados.

É evidente que a construção de procedimentos de software é uma atividade que deve ser feita tendo como referência a hierarquia de controle. Isto porque, na definição do procedimento de um dado módulo de software, deve ser feita referência a todos os módulos subordinados a ele.

Normalmente, esta referência será desenvolvida, num nível mais baixo de detalhamento, de modo a se obter o procedimento de software do módulo subordinado.

3.8. Ocultação de Informação

O princípio da ocultação de informações propõe um caminho para decompor sistematicamente um problema de modo a obter, de modo eficiente, os diferentes módulos do software a construir.

Segundo este princípio, os módulos devem ser decompostos de modo que as informações (os procedimentos e dados) contidas em cada módulo sejam inacessíveis aos módulos que não tenham necessidade destas informações. Ao realizar a decomposição segundo este princípio, o projetista proporciona um grau relativamente alto de independência entre os módulos, o que é altamente desejável num tal projeto.

Os benefícios da adoção deste princípio vão aparecer no momento em que modificações deverão ser encaminhadas a nível da implementação, por exemplo, por conseqüência de uma atividade de teste do software. Graças à ocultação de informação, erros introduzidos como resultado de alguma modificação num dado módulo não serão propagados a outros módulos do software.

4. CLASSES DE PROJETO

Olhando para os diversos aspectos apresentados na seção anterior, pode-se verificar que, de fato, a etapa de projeto de software é caracterizada por um conjunto de projetos, que desenrolam-se paralelamente. Nos itens abaixo, serão discutidos estes diferentes projetos.

4.1. O Projeto Modular

O princípio da modularidade é, sem dúvida, um dos mais difundidos e adotados em qualquer abordagem de desenvolvimento de software conhecida. A razão para isto reside nos benefícios da aplicação de tal princípio, tanto durante o desenvolvimento quanto ao longo da fase de manutenção do software.

Durante o projeto, os princípios de abstração e de ocultação de informação são aplicados como forma de obtenção dos módulos que constituem a arquitetura de um

programa. A aplicação destes dois princípios vai se refletir em características de operação do módulo, como:

- o ***tempo de incorporação***, que indica o momento da introdução do módulo no código-fonte do software (por exemplo, um macro de compilação, um subprograma, etc...);
- o ***mecanismo de ativação***, que indica a forma como o módulo será invocado durante a execução do programa (por exemplo, por meio de uma referência — instrução call — ou por interrupção);
- o ***padrão de controle***, o qual descreve como o módulo é executado internamente.

No que diz respeito aos tipos de módulos que podem ser definidos na arquitetura de um software, pode-se encontrar basicamente três categorias:

- os ***módulos seqüenciais***, que são ativados e sua execução ocorre sem qualquer interrupção;
- os ***módulos incrementais***, que podem ser interrompidos antes da conclusão do aplicativo e terem sua execução retomada a partir do ponto de interrupção;
- os ***módulos paralelos***, que executam simultaneamente a outros módulos em ambientes multiprocessadores concorrentes.

No que diz respeito ao projeto dos módulos, pode-se ainda apresentar algumas definições importantes:

- a ***independência funcional***, que surge como conseqüência da aplicação dos princípios de abstração e ocultação de informação; segundo alguns pesquisadores da área, a independência funcional pode ser obtida a partir da definição de módulos de "propósito único" e evitando-se excessivas interações com outros módulos;
- a ***coesão***, que está fortemente ligada ao princípio de ocultação e que sugere que um módulo pode realizar a sua função com um mínimo de interação com os demais módulos do sistema; é desejável que os módulos num software apresentem um alto grau de coesão;
- o ***acoplamento***, que permite exprimir o grau de conexão entre os módulos; os módulos de um software devem apresentar um baixo coeficiente de acoplamento.

4.2. O Projeto dos Dados

À medida que a etapa de projeto evolui, as estruturas de dados vão assumindo um papel mais importante nesta atividade, uma vez que estas vão definir a complexidade procedimental do software (ou de seus componentes). O projeto dos dados nada mais é do que a seleção das representações lógicas dos objetos de dados identificados na etapa de Análise e Especificação dos Requisitos.

Como forma de obter resultados satisfatórios no que diz respeito ao projeto dos dados no contexto de um software, alguns princípios podem ser adotados:

- a realização de uma análise sistemática no que diz respeito aos dados, a exemplo do que é feito com os aspectos funcionais e comportamentais do software;
- identificação exaustiva das estruturas de dados e das operações a serem realizadas sobre elas;
- estabelecimento de um dicionário de dados (eventualmente, o mesmo definido na etapa de Análise de Requisitos, incluindo refinamentos);
- adiar decisões de baixa prioridade no que diz respeito ao projeto de dados (aplicação do princípio de refinamentos sucessivos);

- limitar a representação das estruturas de dados aos módulos que as utilizarão;
- estabelecimento de uma biblioteca de estruturas de dados úteis e das operações a serem aplicadas a elas (reusabilidade);
- adoção de uma linguagem de programação e projeto que suporte tipos abstratos de dados.

4.3. O Projeto Arquitetural

O projeto arquitetural visa a obtenção de uma estrutura modular de programa, dotada de uma representação dos relacionamentos de controle entre os módulos. Ainda, esta atividade permitem efetuar a fusão dos aspectos funcionais com os aspectos informacionais, a partir da definição de interfaces que irão definir o fluxo de dados através do programa.

Uma questão importante no que diz respeito ao projeto arquitetural é que ele deve ser encaminhado prioritariamente a outros projetos. É importante, antes que outras decisões possam ter tomadas, que se tenha uma visão global da arquitetura do software. O que estiver definido a nível do projeto da arquitetura do software vai ter, sem dúvida, grande impacto nas definições relativas aos demais projetos.

4.4. O Projeto Procedimental

O projeto procedimental é encaminhado a partir da definição da estrutura do software. Devido à riqueza de detalhes que pode caracterizar o projeto procedimental, a adoção de notações adequadas ao projeto é uma necessidade, como forma de evitar a indesejável ambigüidade que poderia ser resultante da utilização, por exemplo, da linguagem natural.

4.4.1. A Programação estruturada

A programação estruturada, introduzida no final dos anos 60, foi uma interessante forma de se encaminhar projetos e implementações de software oferecendo facilidade de programação e de entendimento. A idéia por trás da programação estruturada é a utilização de um conjunto limitado de construções lógicas simples a partir das quais qualquer programa poderia ser construído. Cada construção apresentava um comportamento lógico previsível, iniciando no topo e finalizando na base, o que propiciava ao leitor um melhor acompanhamento do fluxo procedimental.

Independente da linguagem utilizada para a implementação, ela era composta basicamente de três classes de construções: as seqüências, as condições e as repetições.

O uso de um número limitado de construções simples (as quais podem ser combinadas das mais variadas formas) permite obter, sem dúvida, programas mais simples. Um programador que domine efetivamente as construções básicas da programação estruturada, não encontra grandes dificuldades para combiná-las durante a atividade de síntese de um dado procedimento. A mesma observação pode ser feita para as atividades de análise.

4.4.2. O pseudocódigo

Esta notação pode ser aplicada tanto para o projeto arquitetural quanto para o projeto detalhado e a qualquer nível de abstração do projeto. O projetista representa os aspectos estruturais e de comportamento utilizando uma linguagem de síntese, com expressões de sua língua (Inglês, Português, etc...), e estruturada através de construções tais como: IF-THEN-ELSE, WHILE-DO, REPEAT-UNTIL, END, etc... Esta política permite uma representação e uma análise dos fluxos de controle que determinam o comportamento dos componentes do software bastante adequadas à posterior derivação do código de implementação.

O uso do pesudocódigo pode suportar o desenvolvimento do projeto segundo uma política "top-down" ou "bottom-up". No caso do desenvolvimento "top-down", uma frase definida num dado nível de projeto é substituída por seqüências de frases que representam o refinamento da frase original.

O pseudocódigo, apesar de não apresentar uma visão gráfica da estrutura do software ou de seu comportamento, é uma técnica bastante interessante pela sua facilidade de uso e pela sua similaridade com algumas das linguagens de implementação conhecidas, como, por exemplo, Pascal, C, etc... A seguir, é apresentada uma listagem exemplo de uso do pseudocódigo para o projeto detalhado de um componente de software.

```
INICIALIZA tabelas e contadores;
ABRE arquivos;
LÊ o primeiro registro de texto;
ENQUANTO houver registros de texto no arquivo FAZER
     ENQUANTO houver palavras no registro de texto FAZER
       LÊ a próxima palavra
       PROCURA na tabela de palavras a palavra lida
         SE a palavra lida é encontrada
           ENTÃO
             INCREMENTA o contador de ocorrências da palavra lida
           SENÃO
             INSERE a palavra lida na tabela de palavras
        INCREMENTA o contador de palavras processadas
     FIM_ENQUANTO
FIM_ENQUANTO
IMPRIME a tabela de palavras e o contador de palavras processadas
FECHA arquivos
FIM do programa
```

4.4.3. O uso de notações gráficas

Outra contribuição importante às atividades de projeto pode ser adoção de notações gráficas para representar os procedimentos a serem implementados. O fluxograma é uma conhecida técnica para a representação do fluxo de controle de programas, particularmente, os programas seqüenciais. Os fluxogramas estruturados são aqueles baseados na existência de um conjunto limitado de fluxogramas básicos que, combinados, compõem uma representação de comportamento de um elemento de software.

O resultado obtido é uma representação gráfica de comportamento que pode, eventualmente, ser representada por um pseudocódigo. Exemplos de fluxogramas básicos para a composição de fluxogramas mais complexos são apresentados na figura 6.3.

Uma técnica gráfica também definida para a representação de comportamento de componentes de um software é o diagrama de Nassi-Schneiderman. Esta técnica é baseada na representação através de caixas, das estruturas de controle clássicas.

De forma similar aos fluxogramas estruturados, os diagramas de Nassi-Schneiderman são baseados na existência de blocos básicos representando estruturas elementares de controle que serão combinados de modo a compor a representação total do software (ou do componente de software) projetado.

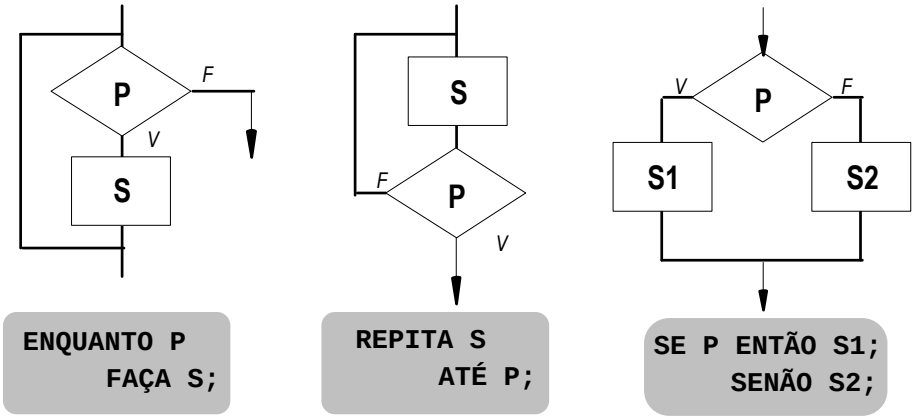


Figura 6.3 - Exemplos de blocos básicos para fluxogramas estruturados.

A figura 6.4 apresenta alguns exemplos de blocos básicos definidos nesta técnica. A representação de comportamento é obtida a partir do encaixe das estruturas básicas, formando uma caixa como mostra o exemplo da figura 6.5.

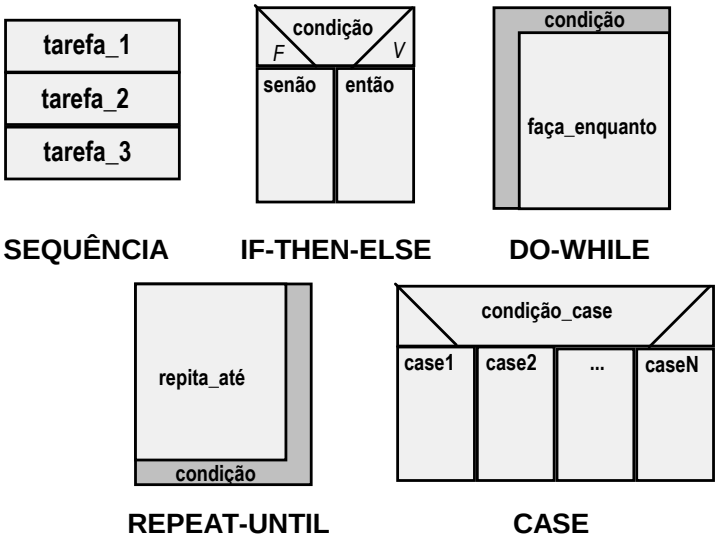


Figura 6.4 - Exemplos de blocos básicos dos diagramas de Nassi-Schneiderman.

5. DOCUMENTAÇÃO

A figura 6.6 apresenta uma proposta de estrutura para o documento de projeto de software. O objetivo deste documento é apresentar uma descrição completa do software, sendo que as seções vão sendo geradas à medida que o projetista avança no seu trabalho de refinamento da representação do software.

Na seção I, é apresentado o escopo global do software, sendo que grande parte do que vai conter esta seção é derivada diretamente do documento de Especificação de Requisitos obtido na etapa precedente, assim como de outros documentos gerados na fase de definição do software. A seção II apresenta as referências específicas à documentação de apoio utilizada.

A *Descrição de Projeto*, objeto da seção III, apresenta uma visão de projeto preliminar. Nesta parte do documento, é apresentada a estrutura do software, obtida a partir de diagramas de fluxos de dados ou outras técnicas de representação utilizadas durante a etapa de Análise de Requisitos. Juntamente com a estrutura do software, são apresentadas as diferentes interfaces de cada componente de software.

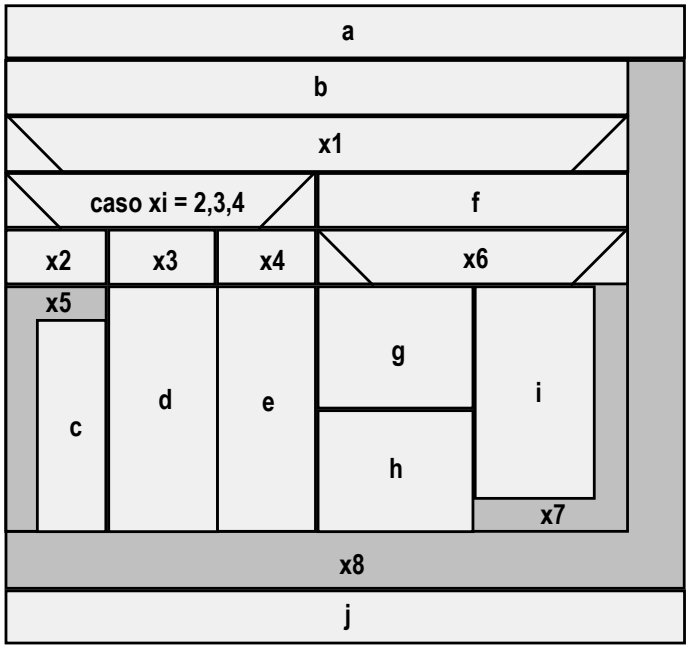


Figura 6.5 - Exemplo de representação de comportamento utilizando Nassi-Schneiderman.

Nas seções IV e V, são apresentados os resultados das atividades de refinamento, que conduzem aos níveis mais detalhados de projeto. Inicialmente, é apresentada uma narrativa (em linguagem natural) da operação de cada componente de software (módulos, procedimentos, funções, etc...). Esta narrativa deve concentrar-se na especificação da função a ser provida pelo componente, evitando detalhes algorítmicos.

A partir desta narrativa, com o auxílio de uma técnica de projeto procedimental, obtém-se uma descrição estruturada de cada componente. A seção V apresenta uma descrição da organização dos dados (arquivos mantidos em meios de armazenagem, dados globais, referência cruzada, etc..).

I - Escopo
II - Documentos de Referência
III - Descrição do Projeto
- **3.1 - Descrição dos Dados**
- **3.2 - Estrutura de Software**
- **3.3 - Interfaces dos Componentes**

IV - Descrição dos Módulos
- **4.1 - Narrativa de Processamento**
- **4.2 - Descrição da Interface**
- **4.3 - Descrição numa Técnica de Projeto**
- **4.4 - Outros**

V - Estrutura de Arquivos e Dados Globais
VI - Referência Cruzada dos Requisitos
VII - Procedimentos de Teste
VIII - Requisitos Especiais
IX - Observações
X - Apêndices

Figura 6.6 - Estrutura do documento de Projeto de Software.

Na seção VI, é elaborada uma referência cruzada dos requisitos, cujo objetivo é determinar que aspectos do projeto desenvolvido estão satisfazendo os requisitos especificados.

De uma forma mais concreta, deve ser feita a indicação de que conjunto de módulos é determinante para a implementação dos requisitos especificados.

Na seção VII é apresentada a especificação dos procedimentos de teste, o que é possível efetuar graças à definição já estabelecida da estrutura do software e das interfaces.

Na seção VIII, são apresentadas as restrições e requisitos especiais para o desenvolvimento do software (necessidade de overlay, gerenciamento de memória virtual, processamento de alta velocidade, interface gráfica específica, etc...).

As seções IX e X apresentam dados complementares, como a descrição de algoritmos, procedimentos alternativos, dados tabulares. Finalmente, pode ser encaminhado o desenvolvimento de um Manual de Instalação/Operações Preliminares, a ser incluído como apêndice ao documento.

6. PROJETO DE INTERFACES HOMEM-MÁQUINA

À medida que os sistemas computacionais foram evoluindo e conquistando um número cada vez maior de usuários, os aspectos de interface com o usuário passaram a assumir um papel de fundamental importância na construção de softwares.

Os computadores pessoais, que apareceram no final dos anos 70, são um exemplo típico desta evolução. Basta olhar a forma como evoluiu a interface de seu sistema operacional, do DOS, em sua versão baseada em linguagens de comando, passando pela construção de uma interface gráfica executando sobre o DOS (o Windows) até chegar, nos dias atuais, ao Windows 95, um sistema inerentemente gráfico que permite ao usuário manipular todos os objetos e eventos de seu sistema através de ícones e movimentos e ações com o mouse. O uso do teclado, num tal sistema, restringe-se à digitação de nomes de arquivos e poucos parâmetros de configuração.

Esta evolução tem sido uma conseqüência, por um lado, da evolução do hardware dos computadores e, por outro lado, da necessidade em eliminar o "terror tecnológico" criado graças à existência de interfaces de difícil aprendizado, complexas no uso e totalmente frustantes em boa parte dos casos.

6.1. Os fatores humanos

Isto faz com que o projeto das interfaces homem-máquina deixe de ser visto como mais uma componente do projeto do software como um todo, mas que passe a ser considerada uma atividade onde uma série de fatores técnicos e, principalmente, humanos sejam levados em conta de modo a que o usuário possa explorar completamente, e da forma mais amigável possível, as potencialidades do software.

O projetista de interfaces homem-máquina deve conhecer alguns conceitos que serão fundamentais para o encaminhamento de suas atividades:

- ***o mecanismo da percepção humana***, particularmente com relação aos sentidos de visão, audição e de tato, exprimem a capacidade do ser humano em adquirir, armazenar e utilizar as informações; na maior parte das operações realizadas num computador, o usuário faz uso dos olhos e do cérebro para o processamento das informações, sendo capaz de diferenciar os diferentes tipos de informação segundo diversos parâmetros visuais (a cor, a forma, as dimensões, etc...); a leitura é um outro processo importante na comunicação usuário-software, apesar da grande tendência em se utilizar recursos gráficos nas interfaces homem-máquina;
- ***os diferentes níveis de habilidades*** das pessoas são um outro aspecto de importância no projeto de uma interface de software; o projetista deve ter a

preocupação de dirigir a interface para o usuário típico do software; uma interface que seja orientada e adequada ao uso por um engenheiro pode representar grandes dificuldades de utilização por parte de um trabalhador inexperiente; no caso de softwares de aplicação vertical (destinados a áreas de aplicação específicas), é importante que a linguagem utilizada suporte termos e conceitos próprios da área;

- ***as tarefas a realizar*** com a introdução do software são, em grande parte dos casos, as mesmas que eram realizadas no sistema não-automatizado; a introdução do software para automatizar um sistema raramente viabiliza a realização de novas tarefas, mas permite que as tarefas antes realizadas manualmente possam vir a ser realizadas de forma mais eficiente e menos dispendiosa; em muitos casos, o software assume tarefas que eram realizadas manualmente; sendo assim, as tarefas essenciais do sistema deve ser um outro aspecto a ser considerado no projeto da interface.

6.2. Aspectos dos projetos de interface

6.2.1. Os diferentes modelos utilizados no projeto

Dado que as interfaces homem-máquina vão envolver diferentes elementos do sistema computacional (pelo menos o software e pessoas), o projetista deve manipular diferentes modelos para obter um resultado em termos de interface que seja compatível com os fatores técnicos e humanos desejados.

Um primeiro modelo a ser definido e utilizado pelo projetista é o **modelo de projeto**, que está relacionado à representação de todos os aspectos do software (procedimentos, arquitetura, dados, etc...).

Um outro modelo a ser desenvolvido pelo projetista é um **modelo de usuário**, o qual permite descrever o perfil típico do usuário do software. Parâmetros como idade, sexo, formação, motivação, capacidades físicas, personalidade, etc..., são levados em conta na construção deste modelo. Além destas características, o grau de experiência e de utilização de um software pode intervir, o que permite classificar os usuários em *principiantes*, *instruídos e intermitentes* e *instruídos e freqüentes*. A localização nesta classificação corresponde ao grau de conhecimento semântico (o conhecimento das funções básicas relativas à aplicação) e conhecimento sintático (a forma de interação com o software) apresentado pelo usuário.

A **percepção do sistema** corresponde a um modelo estabelecido pelo usuário que representa a sua visão do que será o software. Neste modelo, o usuário descreve como ele pensa que deverá ser a sua interação com o software: os objetos da aplicação que ele considera importantes (e que devem, portanto, ser ressaltados), as operações que ele desejará realizar sobre os objetos do sistema, como ele pretende visualizar estes objetos na interface do software, etc...

A **imagem do sistema** é uma representação do implementador do que o software vai oferecer como interface e todo o material de apoio à utilização (manuais, videotapes, livros, etc...). A coincidência entre a percepção do sistema e a imagem do sistema não é um acontecimento óbvio, mas quanto mais próximos estiverem estes modelos, mais chances se terá de projetar uma interface que venha satisfazer aos anseios de seus usuários e que viabilize a utilização efetiva do software.

O trabalho do projetista é o de definir os aspectos de interface à luz dos modelos produzidos, procurando harmonizar os desejos dos usuários com as limitações impostas pelas técnicas e ferramentas disponíveis e pelos critérios de projeto do software.

6.2.2. Análise e Modelagem de Tarefa

Como se pôde verificar nas discussões relativas ao projeto num contexto mais geral, as atividades de síntese (no caso, as atividades de projeto) são antecedidas por atividades de análise (no caso, a análise de requisitos).

Ainda, como já foi citado, os softwares são introduzidos nos sistemas como forma de automatizar um conjunto de tarefas que vêm sendo realizadas de forma inadequada ou ineficiente (ou utilizando equipamentos obsoletos ou realizados artesanalmente). Sendo assim, uma análise do conjunto destas tarefas é primordial ao projeto da interface.

O objetivo desta análise é viabilizar um mapeamento da interface não necessariamente idêntico, mas próximo do conjunto de tarefas que vem sendo realizado pelos profissionais envolvidos na operação do sistema (e que serão os usuários finais do software).

O processo de análise das tarefas é normalmente conduzido segundo uma abordagem por refinamentos sucessivos. O projetista identifica as principais tarefas executadas no contexto da aplicação, sendo que cada tarefa pode ser, eventualmente, refinada em duas ou mais subtarefas.

A partir desta identificação e refinamento das tarefas, o projeto da interface pode ser encaminhado, observando-se os seguintes passos iniciais:

- estabelecimento dos objetivos de cada tarefa;
- definição da seqüência de ações necessárias para cada tarefa;
- especificar como as ações de cada tarefa estarão acessíveis a nível de interface;
- indicar o aspecto visual da interface para cada acesso a uma ação da seqüência especificada (estado do sistema);
- definir os mecanismos disponíveis para que o usuário possa alterar o estado do sistema;
- especificar o efeito de cada mecanismo de controle no estado do sistema;
- indicar o significado de cada estado do sistema (que informações deverão ser oferecidas pela interface em cada estado).

6.2.3. Parâmetros de Projeto

Durante o desenvolvimento de uma interface, existem alguns parâmetros visíveis pelo usuário que devem fazer parte das preocupações do projetista:

- o ***tempo de resposta*** é um parâmetro que, se mal definido (ou omitido num projeto) pode conduzir a resultados frustantes a despeito do cuidado com o qual tenham sido tratados outros fatores; o tempo de resposta a comandos do usuário não tem de ser necessariamente o mais curto possível; em muitas aplicações, pode ser interessante se ter um tempo de resposta mai elevado (sem que o usuário venha a se aborrecer de esperar), de forma a permitir que o usuário reflita sobre as operações que está realizando; o tempo de resposta deve ser projetado observando dois aspectos: a *extensão* (o espaço de tempo entre um comando do usuário e a resposta do software) e a *variabilidade* (que se refere ao desvio de tempo de resposta do tempo de resposta médio de todas as funções do sistema);
- a ***ajuda ao usuário*** é um outro parâmetro importante, uma vez que é muito comum a ocorrência de dúvidas durante a utilização de um software; na maioria dos softwares, dois tipos de facilidades de ajuda podem ser definidas: a ajuda integrada, que é projetada juntamente com o software e que permite que o usuário obtenha respostas rapidamente sobre tópicos relativos à ação que ele está realizando, e a ajuda "add-on", a qual é incorporada após a instalação do software e que exige que o usuário percorra uma lista relativamente longa de tópicos para encontrar a resposta à sua dúvida; é óbvio que o primeiro tipo de ajuda pode tornar a interface mais amigável;

- o **tratamento de mensagens de erro** é um outro aspecto com o qual o projetista deve preocupar-se; a forma como as mensagens são apresentadas para o usuário deve ser cuidadosamente estudada; em muitos casos, as mensagens de erro indicam situações (anormalidades) às quais os usuários não têm nenhum controle (por exemplo, memória insuficiente para executar uma dada função); em outros casos, elas sinalizam uma manipulação incorreta por parte do usuário, a qual pode ser corrigida (por exemplo, indicação de um parâmetro fora dos limites válidos); neste caso, a mensagem deve apresentar informações suficientemente completas e que permitam que o usuário recupere a origem do erro; no caso em que o erro provoque conseqüências danosas à execução de uma dada tarefa (por exemplo, adulteração de conteúdo de um arquivo), estas deverão ser explicitadas na mensagem; resumindo, quanto melhor elaboradas forem as mansagens de erro, menor poderá ser a frustação do usuário quando elas ocorrerem;
- a ***rotulação de comandos*** (ou de menus e opções) corresponde a um outro aspecto que deve ser tratado com especial atenção pelo projetista; os softwares onde a interação é feita através de uma linguagem de comandos correspondem a uma abordagem que tende a desaparecer dando lugar ao uso de técnicas de "point-and-pick", baseadas no uso de janelas e tendo o mouse como ferramenta essencial de entrada de dados e comandos; de todo modo, mesmo softwares com interfaces gráficas de última geração podem oferecer ao usuário uma opção de uso do teclado para todas (ou pelo menos para as mais importantes) funções disponíveis no software (os atalhos de teclado); um outro aspecto presente em aplicações mais recentes é o conceito de macros, que podem ser definidos pelo usuário para armazenar as seqüências mais comuns de uso do software; segundo este conceito, ao invés de digitar todos os comandos necessários à realização de uma dada tarefa, o usuário digita simplesmente o nome do macro; um último aspecto relacionado aos comandos é o uso de padronizações em relação aos rótulos ou atalhos de teclado; um exemplo claro deste tipo de padronização está na maior parte das aplicações construídas para os ambientes gráficos, que utilizam o mesmo rótulo para ações compatíveis como as ações de cópia e reprodução de blocos de texto ou gráfico (copy/paste), as ações de salvamento de arquivos (save/save as...), o abandono de um programa (exit).

6.2.4. A Implementação de Interfaces

O projeto de uma interface homem-máquina consiste invariavelmente num processo iterativo, onde um modelo de projeto é criado, implementado na forma de protótipo e vai sendo refinado e aproximando-se da versão definitiva à medida que recebe realimentações por parte de usuários típicos do software.

Um aspecto importante neste processo é a disponibilidade de ferramentas de suporte à abordagem que permitam obter, com certa agilidade, os modelos de projeto e realizar as modificações necessárias de modo a obter as versões finais.

Atualmente, existem diversos conjuntos de ferramentas de suporte ao desenvolvimento de interfaces, as quais oferecem bibliotecas para a construção dos elementos básicos como janelas, menus, caixas de diálogos, mensagens de erros, etc...Alguns ambientes de programação mais recentes incluem a possibilidade do programador "desenhar" o aspecto visual das telas da interface e obter automaticamente o código dos módulos de interface a partir dos desenhos.

Uma vantagem do uso de tais ferramentas, além da redução do esforço de desenvolvimento, é a obtenção de uma interface que respeite a determinados padrões de aparência e acesso a comandos, já adotados por outras aplicações de uma mesma plataforma.

6.3. Princípios do Projeto de Interfaces

De modo a cumprir o objetivo de gerar uma interface que permita ao usuário explorar completa e eficientemente as potencialidades do software, um conjunto de princípios deve ser observado. Não existe uma fórmula ótima para o desenvolvimento das interfaces, mas, em linhas gerais, pode-se classificar os princípios sob três diferentes pontos de vista: a *interação*, a *exibição de informações* e a *entrada de dados*.

6.3.1. A Interação

No que diz respeito à interação, pode-se relacionar as seguintes diretrizes:

- a ***coerência***, na definição das escolhas do menu, entradas de comandos e outras funções importantes;
- utilização de ***recursos visuais e auditivos*** nas respostas a comandos do usuário;
- exigir ***confirmação*** de qualquer ação destrutiva não trivial (eliminação de arquivos, abandono do software, alterações substanciais de configuração, etc...), proteger eventualmente com *senha* ações que possam acarretar em conseqüências catastróficas para o sistema;
- possibilitar a ***reversão*** (*"undo"*) da maior parte das ações (por exemplo, um usuário que selecione um bloco de texto a ser copiado e que digite acidentalmente uma tecla qualquer antes do comando de cópia pode perder parte do seu trabalho se não houver a possibilidade de anulação da digitação);
- ***redução da quantidade de informações*** a ser memorizada no intervalo entre ações;
- ***otimização de diálogo***, definindo cuidadosamente o layout das telas e dos atalhos de teclado;
- admissão de erros do usuário (atalhos de teclado incorretos, comandos e dados inadequados, etc...), evitando a ocorrência de funcionamento anormal (relacionado à robustez);
- ***categorização das atividades***, organizando a tela segundo a classificação escolhida (operações de arquivos, operações de edição, ajuda, formatação, configurações, etc...);
- oferecimento de ***facilidades de ajuda sensíveis ao contexto***;
- ***nomeação de comandos*** através de verbos ou expressões verbais simples.

6.3.2. A Exibição de Informações

Com relação à exibição de informações, os seguintes aspectos devem ser observados:

- a ***concisão***, ou seja, mostrar apenas informações relevantes ao contexto do sistema;
- priorizar a utilização de ***símbolos gráficos*** para a expressão de informações, quando houver possibilidade;
- ***precisão com relação às mensagens de erro***, permitindo que o usuário possa tomar alguma ação que possa recuperar o erro ou que ele possa evitar uma situação semelhante;
- em ***informações textuais***, o uso de letras maiúsculas e minúsculas pode facilitar o entendimento;
- uso de ***janelas*** para separar diferentes tipos de informação;
- utilizar ***displays análogos*** para representar valores de grandezas reais de um dado sistema (por exemplo, a utilização da figura de um termômetro para indicar o valor da temperatura de uma determinada substância é mais significativa para um

usuário do que um valor numérico desfilando na tela; eventualmente, uma combinação dos dois pode ser uma boa solução — redundância);
- ***uso eficiente do espaço da tela***, particularmente em interfaces baseadas no uso de múltiplas janelas; a disposição destas deve ser tal que pelo menos uma parte de cada janela continue sendo visualizada; caso contrário, definir um mecanismo que permita fazer rapidamente o chaveamento entre janelas (menu Janelas, ou comandos de teclado — ALT+TAB, por exemplo);

6.3.3. A Entrada de Dados

Durante a utilização de um software com alto nível de interatividade, o usuário passa boa parte do seu tempo fazendo a escolha de comandos e inserindo dados de processamento. De forma a melhorar as condições nas quais estas interações ocorrem, os seguintes princípios devem ser considerados:

- ***minimização do número de ações necessárias à entrada de dados***, reduzindo, principalmente a quantidade de texto a ser digitada; um recurso interessante para esta redução é o uso de recursos gráficos sensíveis ao mouse, como por exemplo, escalas variáveis;
- a ***coerência visual da interface em relação às entradas***, mantendo cores, tamanhos e formas compatíveis com o restante da interface;
- ***configuração da entrada por parte do usuário***, permitindo minimizar, em função das habilidades do usuário, etapas na entrada dos dados (por exemplo, eliminar quadros de informação sobre como o usuário deve proceder para executar determinado comando ou definir um dado);
- ***desativação de comandos*** inadequados ao contexto das ações realizadas num dado momento, o que pode evitar que o usuário gere erros de execução pela introdução de um comando inválido (por exemplo, invalidar o comando Reduzir Zoom, quando a redução chegou ao seu limite);
- ***fornecimento de poder de interação ao usuário***, permitindo que ele controle a navagação aolongo do software, eliminando ações de comando e recuperando-se de erros sem a necessidade de abandonar o software;
- ***minimizar o esforço do usuário para a entrada de dados*** (evitar que o usuário tenha de digitar unidades na entrada de grandezas físicas, utilizando uma unidade default modificável; eliminar a necessidade de digitação de ",00" quando os valores não contiverem componentes decimais, etc...).

Engenharia de Software

Capítulo 7

Linguagens de Programação

1. Introdução

Todas as etapas discutidas até o momento no contexto do curso são desenvolvidas visando a obtenção de um código executável que será o elemento central do produto de software. Sob esta ótica, o código do programa nada mais é do que uma formalização, numa dada linguagem, das idéias analisadas na etapa de engenharia do sistema, dos requisitos estabelecidos na etapa de análise e dos modelos gerados na etapa de projeto de modo a que o sistema computacional alvo possa executar as tarefas definidas e refinadas nas etapas anteriores.

Na etapa de Codificação, que é o nome que daremos ao conjunto de atividades relacionadas a esta formalização, é feito uso de linguagens de programação como forma de expressar, numa forma mais próxima da linguagem processada pela máquina, os aspectos definidos nas etapas precedentes.

Num processo de desenvolvimento de software onde os princípios e metodologias da Engenharia de Software sejam aplicados corretamente, a codificação é vista como uma conseqüência natural do projeto. Entretanto, não se deve deixar de lado a importância das características da linguagem de programação escolhida nesta etapa nem o estilo de codificação adotado.

2. A Tradução

Na etapa de codificação, o programador procura mapear corretamente as representações (ou modelos) obtidos no projeto detalhado nas construções suportadas por uma dada linguagem de programação. O resultado deste mapeamento constitui o **código-fonte**, o qual será, na seqüência, traduzido por um compilador gerando o **código-objeto**. Finalmente, o código-objeto será mapeado nas instruções executáveis pelo microprocessador que compõe o sistema computacional, resultando no **código de máquina**.

Ainda em grande parte dos casos, o primeiro nível de mapeamento é obtido de forma manual ou semi-automatizada, estando sujeito, portanto, a "ruídos" inerentes do processo. Alguns exemplos de ruídos são:

- interpretação inadequada de aspectos da especificação de projeto;
- restrições ou complexidades características da linguagem de programação adotada;
- etc...

As restrições impostas por uma dada linguagem de programação ou o conhecimento incompleto das suas potencialidades pode conduzir a raciocínios (e conseqüentes projetos) relativamente limitados.

3. CARACTERÍSTICAS DAS LINGUAGENS DE PROGRAMAÇÃO

Como já foi mencionado, as linguagens de programação funcionam no processo de desenvolvimento de software como o agente de formalização das representações de projeto perante o sistema computacional. As características de uma linguagem de programação podem exercer um impacto significativo no processo de codificação segundo diferentes ângulos, os quais serão abordados nos parágrafos que seguem.

3.1. Características Psicológicas

Sob a ótica psicológica, aspectos tais como a facilidade de uso, simplicidade de aprendizagem, confiabilidade e baixa freqüência de erros são características que devem ser apresentadas pelas linguagens de programação; estes fatores podem causar impacto significativo no processo de desenvolvimento de um software, considerando a codificação como uma atividade intrinsecamente humana. As características de ordem psicológica são apresentadas abaixo.

3.1.1. Uniformidade

Esta característica está relacionada à consistência (coerência) da notação definida para a linguagem, a restrições arbitrárias e ao suporte a exceções sintáticas e semânticas. Um exemplo (ou contra-exemplo) significativo desta característica está na linguagem Fortran, que utiliza os parênteses para expressar dois aspectos completamente diferentes na linguagem — a precedência aritmética e a delimitação de argumentos de subprogramas).

3.1.2. Ambigüidade

Esta é uma característica observada pelo programador na manipulação da linguagem. Embora o compilador interprete a instrução de um único modo, o leitor (programador) poderá ter diferentes interpretações para a mesma expressão. Um caso típico de problema relacionado a esta característica é a precedência aritmética implícita. Por exemplo, a expressão **`X = X1/X2*X3`** pode levar a duas interpretações por parte do leitor: **`X = (X1/X2)*X3`** ou **`X = X1/(X2*X3).`**

3.1.3. Concisão

Esta é uma outra característica psicológica desejável nas linguagens de programação que está relacionada à quantidade de informações orientadas ao código que deverão ser recuperadas da memória humana. Os aspectos que influem nesta característica são a capacidade de suporte a construções estruturadas, as palavras-chave e abreviações permitidas, a diversidade com relação aos tipos de dados, a quantidade de operadores aritméticos e lógicos, entre outros.

3.1.4. Localidade

A localidade está relacionada ao suporte a construções que possam ser identificadas em blocos. O leitor identifica um conjunto de instruções a partir da visualização de um de seus componentes. Por exemplo, as construções do tipo ***IF-THEN-ELSE***, REPEAT-UNTIL, entre outras, são exemplos de construções que aumentam a característica de localidade numa linguagem.

3.1.5. Linearidade

A linearidade está relacionada à manutenção do domínio funcional, ou seja, a percepção é melhorada quando uma seqüência de instruções lógicas é encontrada. Programas caracterizados por grandes ramificações violam esta característica.

3.2. Características de Engenharia

Numa visão de engenharia de software, as características das linguagens de programação estão relacionadas às necessidades do processo de desenvolvimento do software. Algumas características que podem ser derivadas segundo esta ótica são:

3.2.1. Facilidade de derivação do código-fonte

Esta característica está ligada à proximidade da sintaxe da linguagem de programação com as linguagens de representação de projeto. Na maior parte dos casos, as linguagens que suportam construções estruturadas, estruturas de dados relativamente complexas, possibilidade de manipulação de bits, construções orientadas a objetos e construções de entrada/saída especializadas permitem mapear mais facilmente as representações de projeto

3.2.2. Eficiência do compilador

Esta característica contempla os requisitos de tempo de resposta e concisão (exigência de pouco espaço de memória) que podem caracterizar determinados projetos.

A maioria dos compiladores de linguagens de alto nível apresenta o inconveniente de geração de código ineficiente.

3.2.3. Portabilidade

Este aspecto está relacionado ao fato do código-fonte poder migrar de ambiente para ambiente, com pouca ou nenhuma alteração. Por ambiente, pode-se entender o processador, o compilador, o sistema operacional ou diferentes pacotes de software.

3.2.4. Ambiente de Desenvolvimento

Um aspecto fundamental é, sem dúvida, a disponibilidade de ferramentas de desenvolvimento para a linguagem considerada. Neste caso, deve ser avaliada a disponibilidade de outras ferramentas que as de tradução, principalmente, ferramentas de auxílio à depuração, edição de código-fonte, bibliotecas de subrotinas para uma ampla gama de aplicações (interfaces gráficas, por exemplo).

3.2.5. Manutenibilidade

Outro item de importância é a facilidade em alterar o código-fonte para efeito de manutenção. Esta etapa, em muitos casos negligenciada nos projetos de desenvolvimento de software, só poderá ser conduzida eficientemente a partir de uma completa compreensão do software.

Embora a existência de documentação relativa ao desenvolvimento do software seja de fundamental importância, a clareza do código-fonte vai ser fator determinante para o sucesso das tarefas de manutenção.

3.3. Características técnicas

As duas classes de características apresentadas anteriormente são, de certo modo, conseqüência da forma como as linguagens de programação foram concebidas, tanto do ponto de vista sintático como semântico. A seguir, serão apresentadas as principais características "técnicas" das linguagens de programação, as quais exercem forte impacto na qualidade do software e nas condições sob as quais este é desenvolvido.

3.3.1. Suporte a tipos de dados

Quanto mais poderosos são os programas atuais, mais complexas são as estruturas de dados que estes manipulam. Nas linguagens de programação atuais, o suporte à representação dos dados é caracterizado por diferentes categorias de dados, desde estruturas extremamente simples até estruturas com alto grau de complexidade. A seguir são enumeradas algumas destas categorias:

- tipos numéricos (inteiros, complexos, reais, bit, etc...);
- tipos enumerados (valores definidos pelos usuários... cores, formas, etc...);
- cadeias de caracteres (strings);
- tipos booleanos (verdadeiro/falso);
- vetores (uni ou multidimensionais);
- listas, filas, pilhas;
- registros heterogêneos.

Na maior parte dos casos, a manipulação lógica e aritmética de dados é controlada por mecanismos de verificação de tipos embutidos nos compiladores ou interpretadores. O objetivo deste mecanismo é garantir a coerência dos programas com relação às estruturas de dados e às operações que podem ser realizadas sobre eles.

3.3.2. Os subprogramas

Os subprogramas correspondem a componentes de programa contendo estruturas de controle e dados que podem ser compilados separadamente. Com relação aos elementos de um software definidos na etapa de projeto, pode-se, eventualmente, considerar subprogramas como o mapeamento natural dos módulos de um software, embora não necessariamente os módulos (que correspondem a uma definição mais genérica) serão mapeados em subprogramas. Os subprogramas recebem diferentes denominações, dependendo da linguagem de programação considerada (subrotinas, procedimentos, funções, módulos, agentes, processos, objetos).

Independentemente da linguagem de programação considerada, os subprogramas são caracterizados pelos seguintes elementos:

- uma seção de especificação (declaração), contendo o seu identificador e uma descrição da interface;
- uma seção de implementação, onde é descrito o seu comportamento através das estruturas de controle e dados;
- um mecanismo de ativação, o qual permite invocar o subprograma a partir de um ponto qualquer do programa (por exemplo, uma instrução ***call***).

3.3.3. Estruturas de controle

Praticamente todas as linguagens de programação oferecem suporte à representação das estruturas de controle mais clássicas, como seqüências, condições e repetições. A maior parte destas linguagens oferece um conjunto de estruturas sintáticas

quase que padronizadas que permitem exprimir estruturas como IF-THEN-ELSE, DO-WHILE, REPEAT-UNTIL e CASE.

Além das estruturas clássicas mencionadas, outros fatores podem ser interessantes como suporte a aspectos de programação não contemplados nestas estruturas. Abaixo são citados alguns destes fatores:

- a ***recursividade***, que permite que um subprograma contenha em seu código uma ativação a si próprio;
- a ***concorrência***, que permite a criação de múltiplas tarefas e oferece suporte à comunicação e sincronização destas tarefas;
- o ***tratamento de exceções***, que permite ativar a execução de um conjunto de instruções a partir da ocorrência de um evento significativo do sistema computacional (erro de execução, relógio de tempo-real, etc...) ou definido pela aplicação (interrupção causada pela pressão de uma tecla, chegada de uma mensagem, etc...).

3.3.4. Suporte a abordagens orientadas a objeto

Não obstante o resultado da aplicação de uma abordagem orientada a objeto possa ser codificada a partir de qualquer linguagem de programa, o mapeamento dos elementos especificados no projeto fica mais simples se a linguagem de programação oferece suporte a esta abordagem.

Sendo assim, o suporte a conceitos como classe, herança, encapsulamento e troca de mensagens pode ser um aspecto interessante numa linguagem utilizada na implementação de um projeto orientado a objetos.

3.4. Critérios de Escolha de uma Linguagem de Programação

As características apresentadas acima devem ser levadas em conta no momento da escolha de uma linguagem de programação no contexto de um projeto de software. Por outro lado, outros critérios podem auxiliar na decisão, entre eles:

- a área de aplicação para a qual o software está sendo construído;
- a complexidade computacional e algorítmica do software;
- o ambiente no qual o software vai executar;
- os requisitos de desempenho;
- a complexidade da estrutura de dados;
- o conhecimento da equipe de desenvolvimento;
- a disponibilidade de boas ferramentas de desenvolvimento.

4. Classes de Linguagens

Devido à grande diversidade das linguagens de programação existentes nos dias atuais, é interessante estabelecer uma classificação que permita situá-las no contexto do desenvolvimento de programas. Embora os critérios de classificação possam ser os mais diversos, adotou-se aqui uma ótica cronológica, onde as linguagens posicionam-se, de certo modo, em função de suas características técnicas.

4.1. Linguagens de Primeira Geração

Nesta classe de linguagens, desenvolvidas nos primórdios da computação, enquadram-se as linguagens em nível de máquina. O código de máquina representa a linguagem interpretada pelos microprocessadores, sendo que cada microprocessador é caracterizado por uma linguagem de máquina própria, podendo, eventualmente,

estabelecer-se certa compatibilidade em software entre microprocessadores de uma mesma família (fabricante).

Embora tendo seu uso bastante limitado, alguns programas ainda são realizados utilizando o código de máquina ou, mais precisamente, seu equivalente "legível", a linguagem Assembly.

Numa ótica de Engenharia de Software, a programação através de linguagens desta classe limitam-se a situações onde a linguagem de alto nível adotada para a programação não suporte alguns requisitos especificados.

4.2. Linguagens de Segunda Geração

As linguagens de segunda geração constituem as primeiras linguagens de "alto nível", que surgiram entre o final da década de 50 e início dos anos 60. Linguagens como Fortran, Cobol, Algol e Basic, com todas as deficiências que se pode apontar atualmente, foram linguagens que marcaram presença no desenvolvimento de programas, sendo que algumas delas têm resistido ao tempo e às críticas, como por exemplo Fortran que ainda é visto como uma linguagem de implementação para muitas aplicações de engenharia. Cobol, por outro lado, ainda é uma linguagem bastante utilizada no desenvolvimento de aplicações comerciais.

4.3. Linguagens de Terceira Geração

Nesta classe, encaixam-se as chamadas linguagens de programação estruturada, surgidas em meados dos anos 60. O período compreendido entre a década de 60 e a de 80 foi bastante produtivo no que diz respeito ao surgimento de linguagens de programação, o que permitiu o aparecimento de uma grande quantidade de linguagens as quais podem ser organizadas da seguinte forma:

- as ***linguagens de uso geral***, as quais podem ser utilizadas para implementação de programas com as mais diversas características e independente da área de aplicação considerada; encaixam-se nesta categoria linguagens como Pascal, Modula-2 e C;
- as ***linguagens especializadas***, as quais são orientadas ao desenvolvimento de aplicações específicas; algumas das linguagens que ilustram esta categoria são Prolog, Lisp e Forth;
- as ***linguagens orientadas a objeto***, que oferecem mecanismos sintáticos e semânticos de suporte aos conceitos da programação orientada a objetos; alguns exemplos destas linguagens são Smalltalk, Eiffel e C++.

4.4. Linguagens de Quarta Geração

As linguagens de quarta geração surgiram como forma de aumentar o grau de abstração na construção de programas. Uma característica comum nestas linguagens, é a especificação do programa sem a necessidade de expressão de detalhes algorítmicos de processamento.

É evidente que, devido a esta característica, é impossível que uma linguagem possa ser utilizada para a programação de uso geral, sendo normalmente destinadas a aplicações em domínios específicos.

Dentro desta classe, é possível encontrar diferentes categorias de linguagens:

4.4.1. Linguagens de consulta

Nesta categoria, encaixam-se as linguagens orientadas a aplicações conjuntas de bancos de dados, permitindo que o usuário especifique operações sobre os registros num alto nível de sofisticação. Algumas linguagens nesta categoria incorporam recursos de

interpretação de linguagem natural, de modo que o usuário pode manipular a informação no nível mais alto de abstração possível.

4.4.2. Linguagens geradoras de programas

Nesta categoria estão as linguagens que permitem que o programador especifique o programa considerando construções e instruções num elevado nível de abstração, sendo que os programas serão obtidos em código-fonte de uma linguagem de programação de terceira geração.

4.4.3. Outras linguagens

Nesta categoria, pode-se considerar as linguagens desenvolvidas nos últimos anos como por exemplo as linguagens de especificação formal (referenciadas no capítulo 6), as linguagens de prototipação, e os ambientes de programação presentes em computadores pessoais (planilhas, bancos de dados, hypertexto, etc...).

5. ESTILO DE CODIFICAÇÃO

A escolha de uma linguagem de programação adequada, que suporte os requisitos de projeto estabelecidos e que apresente boas características entre o conjunto apresentado anteriormente é, sem dúvida, um grande passo na direção da obtenção de um software de qualidade.

Entretanto, um fator que é essencial para a obtenção de um código claro e que ofereça facilidade de manutenção é a boa utilização dos mecanismos presentes na linguagem adotada, o que vamos rotular aqui por “estilo de codificação”.

Abaixo, serão discutidos os principais fatores determinantes à obtenção de código-fonte bem escrito.

5.1. A Documentação de Código

O código-fonte não deve ser considerado, em nenhuma hipótese, como um resultado intermediário irrelevante no processo de desenvolvimento de um software. Ele se constitui num elemento essencial tanto para atividades de validação do software como (e principalmente) para as tarefas de manutenção.

Desta forma, o aspecto de documentação é um aspecto que deve ser bastante considerado na etapa de codificação.

Um primeiro passo na documentação do código fonte é a escolha dos identificadores de variáveis e procedimentos. O uso de “siglas” ou caracteres para identificar os elementos de um programa é uma tendência herdada das linguagens de segunda geração que não encontra mais lugar nos dias atuais. Assim, é importante que os identificadores escolhidos tenham significado explícito para a aplicação considerada.

Outro aspecto bastante considerado é a introdução de comentários ao longo do código-fonte. Na realidade este é um aspecto bastante polêmico, mas este mecanismo pode ser bastante útil se utilizado eficientemente, podendo transformar-se no principal aliado às tarefas de manutenção. Uma sistemática interessante de uso dos comentários é a compatibilização destes com a documentação de projeto do software.

Um último aspecto importante na documentação é a formatação do código-fonte. Um mecanismo importante neste contexto é o uso da endentação como meio de melhor posicionar as instruções no contexto de trechos significativos do código. A endentação permite explicitar melhor a combinação dos blocos básicos de uma linguagem de programação (as estruturas clássicas de controle) para a composição de componentes mais complexos. No que diz respeito a este aspecto muitas ferramentas CASE oferecem os chamados formatadores automáticos de código, os quais liberam o programador desta preocupação.

5.2. A Declaração de Dados

Este aspecto nem sempre é objeto de preocupação por parte dos programadores, mas, à medida que o programa vai se tornando complexo no que diz respeito à definição de estruturas de dados, o estabelecimento de uma sistemática padronizada de declaração de tipos de dados e variáveis vai assumindo um nível cada vez maior de importância.

A linguagem Pascal deu um impulso importante neste aspecto, com a possibilidade do usuário construir tipos de dados mais complexos a partir dos tipos básicos e de seus construtores. Este mecanismo está presente em praticamente todas as linguagens em uso nos dias de hoje.

5.3. A Construção de Instruções

O fluxo lógico de instruções é definido normalmente durante a etapa de projeto (projeto detalhado). Entretanto, o mapeamento deste fluxo lógico em instruções individuais faz parte do trabalho de codificação.

Um aspecto que deve ser explorado durante a construção das instruções é a simplicidade. Utilizar as possibilidades que algumas linguagens oferecem de escrever mais de uma instrução por linha, por exemplo, é sem dúvida uma decisão que vai contra a clareza do código, independente da economia de espaço (e de papel) que isto pode representar.

Abaixo, são apresentadas algumas regras importantes no que diz respeito a este aspecto:

- evitar o uso de testes condicionais complicados ou que verifiquem condições negativas;
- evitar o intenso aninhamento de laços ou condições;
- utilizar parênteses para evitar a ambigüidade na representação de expressões lógicas ou aritméticas;
- utilizar símbolos de espaçamento para esclarecer o conteúdo de uma instrução.

5.4. Entradas/Saídas

As entradas de dados num software podem ser realizadas de forma interativa ou em batch. Independente da forma como as entradas serão efetuadas, é importante considerar algumas regras:

- validação de todas as entradas de dados;
- simplicidade e padronização no formato da entrada;
- no caso de entradas envolvendo múltiplos dados, estabelecer um indicador de final de entrada (ENTER, ponto final, etc...);
- entradas de dados interativas devem ser rotuladas, indicando-se eventuais parâmetros, opções e valores default;
- rotulação de relatório de saída e projeto.

6. CODIFICAÇÃO E EFICIÊNCIA

Embora os fatores determinantes da eficiência na execução de software possam ser determinados fundamentalmente por aspectos de hardware como o processador utilizado e o espaço de memória disponível, a codificação pode ter uma participação, mesmo modesta, na redução do tempo de processamento de um programa ou na menor ocupação de espaço na memória.

De modo geral, existem basicamente três regras a serem observadas no tocante à eficiência:

- a eficiência é um requisito de desempenho e deve, portanto, ser contemplada desde a etapa de análise e definição dos requisitos;
- a eficiência pode ser melhorada com um bom projeto;
- a eficiência e a simplicidade do código devem ser requisitos bem equilibrados; em hipótese alguma deve-se sacrificar a clareza e a legibilidade do código em nome da eficiência.

A codificação pode ser encaminhada de modo a obter eficiência segundo diferentes visões, as quais serão discutidas a seguir.

6.1. Eficiência de Código

Este aspecto está relacionado diretamente à eficiência dos algoritmos que serão implementados nos componentes do software. Embora estes sejam especificados na fase de projeto detalhado, o estilo de codificação pode trazer uma contribuição importante, particularmente com a observação das seguintes regras:

- expressões aritméticas e lógicas devem ser simplificadas como uma tarefa que anteceda a codificação;
- malhas aninhadas devem ser cuidadosamente analisadas para verificar a possibilidade de moves instruções para fora delas;
- quando possível, privilegiar o uso de estruturas de dados simples, evitando estruturas do tipo vetores multidimensionais, ponteiros e listas complexas;
- adotar operações aritméticas rápidas;
- evitar a mistura de diferentes tipos de dados (mesmo que a linguagem utilizada permita isto);
- utilizar expressões booleanas e aritmética de números inteiros sempre que possível.

6.2. Eficiência de Memória

Este item refere-se ao fato do programa ocupar o menor espaço possível na memória da máquina. Embora no que se refere a grandes computadores e mesmo em computadores pessoais (em alguns casos) a eficiência de memória não seja uma questão fundamental, a preocupação é verdadeira no caso de sistemas dedicados.

Não existem propriamente regras explícitas para se atingir a eficiência de memória, mas o parâmetro fundamental para obter isto é a simplicidade de software. Além disso, algumas das regras seguidas para obtenção da eficiência de código podem resultar igualmente em eficiência de memória. O caso típico para isto é a utilização de estruturas de dados simples.

6.3. Eficiência de Entradas/Saídas

Este é outro aspecto importante a ser considerado no momento da codificação. As principais regras a serem seguidas para se atingir este parâmetro são:

- minimizar a quantidade de solicitações de E/S;
- fazer uso de buffers para reduzir o acúmulo de operações de comunicação;
- no caso de E/S para dispositivos secundários deve adotar uma organização dos dados em blocos e métodos de acesso simples;
- as E/S para terminais e impressoras devem levar em conta características dos dispositivos que melhorem sua velocidade e qualidade;
- a clareza da E/S deve prevalecer sobre as questões de eficiência.

Engenharia de Software

Capítulo 8

Teste de Software

1. Introdução

O desenvolvimento de software utilizando as metodologias, técnicas e ferramentas da Engenharia de Software não oferece a total garantia de qualidade do produto obtido, apesar de melhorá-la significativamente. Por esta razão, uma etapa fundamental na obtenção de um alto nível de qualidade do software a ser produzido é aquela onde são realizados os procedimentos de teste, uma vez que esta é a última etapa de revisão da especificação, do projeto e da codificação.

A realização, de forma cuidadosa e criteriosa, dos procedimentos associados ao teste de um software assume uma importância cada vez maior dado o impacto sobre o funcionamento (e o custo) que este componente tem assumido nos últimos anos. Por esta razão, o esforço despendido para realizar a etapa de teste pode chegar a 40% do esforço total empregado no desenvolvimento do software.

No caso de programas que serão utilizados em sistemas críticos (aqueles sistemas dos quais dependem vidas humanas, como controle de vôo e a supervisão de reatores nucleares), a atividade de teste pode custar de 3 a 5 vezes o valor gasto nas demais atividades de desenvolvimento do software.

O objetivo deste capítulo é apresentar, de forma breve, os principais conceitos e técnicas relacionados ao teste de software.

2. Fundamentos do teste de software

2.1. Objetivos

Os objetivos do teste de software podem ser expressos, de forma mais clara, pela observação das três regras definidas por Myers:

- A atividade de teste é o processo de executar um programa com a intenção de descobrir um erro;
- Um bom caso de teste é aquele que apresenta uma elevada probabilidade de revelar um erro ainda não descoberto;
- Um teste bem sucedido é aquele que revela um erro ainda não descoberto.

As três regras expressam o objetivo primordial do teste que é o de encontrar erro, contrariando a falsa idéia de que uma atividade de teste bem sucedida é aquela em que nenhum erro foi encontrado.

A etapa de teste deve ser conduzida de modo que o maior número de erros possível seja encontrado com um menor dispêndio de tempo e esforço.

2.2. O projeto de casos de teste

A realização, com sucesso, da etapa de teste de um software deve ter, como ponto de partida, uma atividade de projeto dos casos de teste deste software. Projetar casos de teste para um software pode ser uma atividade tão complexa quanto a de projeto do próprio software, mas ela é necessária como única forma de conduzir, de forma eficiente e eficaz, o processo de teste.

Os princípios básicos do teste de qualquer produto resultante de uma tarefa de engenharia são:

- conhecida a função a ser desempenhada pelo produto, testes são executados para demonstrar que cada função é completamente operacional, este primeiro princípio deu origem a uma importante abordagem de teste, conhecida como o *teste de caixa preta (black box)*;
- com base no conhecimento do funcionamento interno do produto, realiza-se testes para assegurar de que todas as peças destes estão completamente ajustadas e realizando a contento sua função; à abordagem originada por este segundo princípio, foi dado o nome de *teste de caixa branca (white box)*, devido ao fato de que maior ênfase é dada ao desempenho interno do sistema (ou do produto).

2.2.1. O software e o teste de caixa preta

Quando o procedimento de teste está relacionado ao produto de software, o teste de caixa preta refere-se a todo teste que implica na verificação do funcionamento do software através de suas interfaces, o que, geralmente, permite verificar a operacionalidade de todas as suas funções. É importante observar que, no teste de caixa preta, a forma como o software está organizado internamente não tem real importância, mesmo que isto possa ter algum impacto na operação de alguma função observada em sua interface.

2.2.2. O software e o teste de caixa branca

Um teste de caixa branca num produto de software está relacionado a um exame minucioso de sua estrutura interna e detalhes procedimentais. Os caminhos lógicos definidos no software são exaustivamente testados, pondo à prova conjuntos bem definidos de condições ou laços. Durante o teste, o "status" do programa pode ser examinado diversas vezes para eventual comparação com condições de estado esperadas para aquela situação.

Apesar da importância do teste de caixa branca, não se deve guardar a falsa idéia de que a realização de testes de caixa branca num produto de software vai oferecer a garantia de 100% de correção deste ao seu final. Isto porque, mesmo no caso de programas de pequeno e médio porte, a diversidade de caminhos lógicos pode atingir um número bastante elevado, representando um grande obstáculo para o sucesso completo desta atividade..

3. MODALIDADES DE TESTE

3.1. Testes estáticos e dinâmicos

Uma primeira divisão que pode ser estabelecida em relação ao teste de software corresponde à forma de utilização do código obtido na etapa de implementação (ou codificação). Segundo esta ótica, pode-se organizar os testes em **estáticos** e **dinâmicos**.

3.1.1. Os Testes Estáticos

Os testes estáticos são aqueles realizados sobre o código-fonte do software, utilizando como técnica básica a inspeção visual. Este tipo de teste é de simples implementação, uma vez que não há necessidade de execução do programa para obter-se resultados. Eles podem ser utilizados utilizando a técnica de leitura cruzada, onde um leitor é atribuído para avaliar o trabalho de cada programador do software. Uma outra forma de realizar o teste é por inspeção, onde uma equipe designada analisa o código à luz de um questionário especialmente concebido — a check list.

Nos dois casos, os responsáveis pela realização do teste não realizam nenhum tipo de correção no código, limitando-se a assinalar os erros encontrados.

Eventualmente, o teste estático pode ser automatizado com o auxílio de ferramentas de análise estática, que podem ser simples geradores de referências cruzadas ou, no caso de ferramentas mais sofisticadas, serem dotadas de funções de análise do fluxo de dados do programa.

3.1.2. Os Testes Dinâmicos

Os testes dinâmicos são os procedimentos baseados na execução do código binário do programa, sendo esta execução realizada com base em subconjuntos de dados — o jogo de teste.

A escolha do subconjunto de dados a ser utilizado para o teste pode ser feita com base em aspectos estruturais do software (obtido a partir do código-fonte) ou em aspectos funcionais (a partir da especificação do programa).

3.2. Testes de unidade, integração, validação e sistema

Uma outra forma de subdividir o teste de software é quanto ao seu objetivo na busca por erros do programa. Sob este ponto de vista, encontra-se os seguintes tipos de teste:

3.2.1. O Teste de Unidade

O teste de unidade objetiva a verificação de erros existentes nas unidades de projeto do mesmo, à qual daremos o nome de **módulo**. Nesta modalidade de teste, é importante utilizar as informações contidas no documento de projeto detalhado do software, as quais servirão de guia para sua aplicação. O teste de unidade é, de certa forma, uma técnica de teste de caixa branca, podendo ser realizado em paralelo sobre diferentes módulos.

3.2.2. O Teste de Integração

O Teste de Integração, como o nome indica, objetiva a busca de erros surgidos quando da integração das diferentes unidades componentes do software. É importante lembrar que, o fato de se ter analisado os módulos do software de forma exaustiva (através de procedimentos de teste de unidade), não há nenhuma garantia de que estes, uma vez colocados em conjunto para funcionar, não apresentarão anomalias de comportamento.

Uma das maiores causas de erros encontrados durante o teste de integração são os chamados erros de interface, devido, principalmente, às incompatibilidades de interface entre módulos que deverão trabalhar de forma cooperativa.

3.2.3. O Teste de Validação

Ao final do teste de integração, o software é finalmente estruturado na forma de um pacote ou sistema. A forma mais simples de definição do teste de validação é a verificação de que o software como um todo cumpre corretamente a função para a qual ele foi especificado. É importante lembrar que, no documento de especificação do software,

quando no início da sua concepção, são definidos os chamados critérios de validação, que servirão de guia para o julgamento de aprovação ou não do software nesta etapa de testes.

3.2.4. O Teste de Sistema

Neste tipo de teste, são verificados os aspectos de funcionamento do software, integrado aos demais elementos do sistema, como o hardware e outros elementos. Dependendo do destino do software, é somente neste momento do teste em que alguns peris de funcionamento do software podem ser efetivamente testados (particularmente, aqueles aspectos do funcionamento que dependem da interação dos demais elementos do sistema).

Apresentada uma breve descrição destes tipos de teste, informações mais detalhadas serão apresentadas nas seções que seguem.

4. TESTE DE UNIDADE

4.1. ASPECTOS A SEREM TESTADOS

A figura 8.1 ilustra a forma como são desenvolvidos os testes de unidade. Os aspectos verificados no contexto deste teste, normalmente são:

- a *interface*, em busca de erros de passagem de dados para dentro e para fora do módulo;
- as *estruturas de dados locais*, para se prover a garantia de que todos os dados armazenados localmente mantêm sua integridade ao longo da execução do módulo;

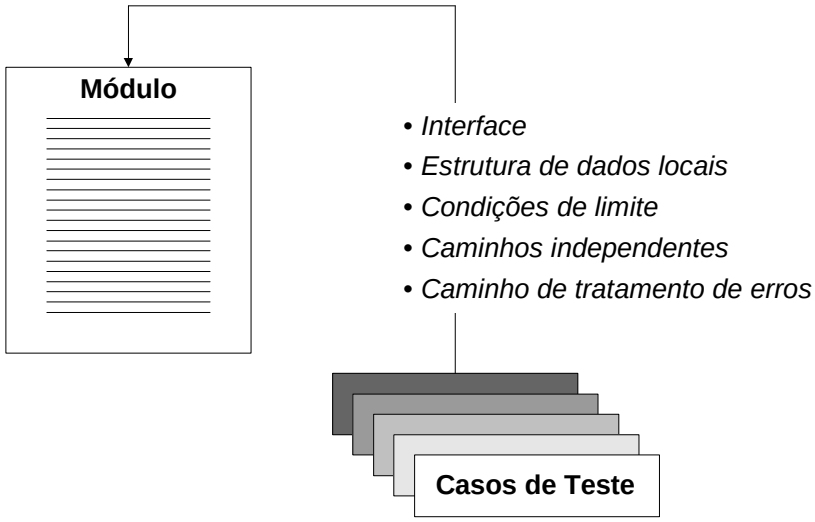


Figura 8.1 - Teste de Unidade

- as *condições limite*, que permitem verificar que o módulo executa respeitando os valores máximos e mínimos estabelecidos para seu processamento;
- os caminhos independentes (ou caminhos básicos) da estrutura de controle são analisados para se ter garantia de que todas as instruções do módulo foram executadas pelo menos uma vez;
- finalmente, os *caminhos de tratamento de erros* (se existirem), serão testados para observar a robustez do módulo.

4.1.1. O teste de interface

O teste de interface é o primeiro e o mais importante teste a ser aplicado sobre uma unidade de software. Ele deve ser realizado em primeiro lugar, pois, qualquer anomalia observada durante a realização deste teste põe em dúvida os resultados obtidos nos demais testes sobre a mesma unidade. Durante a realização deste teste, os seguintes aspectos devem ser observados:

- coerência (em termos de números, atributos e sistemas de unidades) entre argumentos de um módulo e os parâmetros de entrada;
- coerência (em termos de números, atributos e sistemas de unidades) entre argumentos passados aos módulos chamados e os parâmetros de entrada;
- verificação sobre a consistência dos argumentos e a ordem de passagem destes nas funções embutidas;
- existência de referências a parâmetros não associados ao ponto de entrada atual;
- alteração de argumentos de entrada/saída;
- consistência de variáveis globais ao longo dos módulos;
- passagem de restrições sobre argumentos.

No caso de unidades que envolvam tratamento de entradas/saídas, todas as operações envolvendo tratamento de arquivos ou programação de dispositivos periféricos devem ser cuidadosamente (ou exaustivamente) verificadas.

4.1.2. O teste de estruturas de dados

Com relação as estruturas de dados, devem ser verificadas não apenas aquelas que serão utilizadas exclusivamente no contexto da unidade, mas também aquelas definidas em termos globais. Praticamente todas as linguagens de programação oferecem mecanismos para a definição de estruturas de dados com diferentes escopos. Neste teste, deve ser verificado o bom uso destes mecanismos e se esta utilização está sendo feita de modo coerente com o que foi estabelecido no projeto.

Os erros mais comuns detectados neste tipo de teste são:

- digitação inconsistente ou imprópria (por exemplo, identificadores de uma variável ou tipo de dados digitados de forma incorreta);
- iniciação incorreta de variáveis (valores iniciais incorretos);
- inconsistência nos tipos de dados;
- *underflow* e *overflow*.

4.1.3. O teste de condições limite

Este teste busca verificar que o que foi definido a nível de projeto para as condições limite de uma unidade está sendo respeitado a nível de implementação. Por exemplo, um módulo encarregado de implementar um mecanismo de temporização deve ser testado para observar se o tempo máximo de espera está compatível com o que foi especificado.

4.1.4. O teste de caminhos de execução

O sucesso da realização deste tipo de teste depende fortemente da complexidade do algoritmo implementado a nível da unidade sob teste. É que neste caso, é necessário que os casos de teste a serem aplicados permitam percorrer os diversos caminhos de execução da unidade, os quais serão em maior número quanto mais complexa for a unidade. Os erros mais comuns verificados neste tipo de teste são:

- precedência aritmética incorreta;
- operações envolvendo dados de tipos diferentes;
- inicialização incorreta;
- erros deprecisão;
- representação incorreta de uma expressão;
- laços mal definidos;
- comparação de diferentes tipos de dados;
- operadores lógicos utilizados incorretamente;
- etc.

4.1.5. O teste de caminhos de tratamento de erros

É desejável que um projeto de software estabeleça caminhos de tratamento de erros para as unidades que o compõem, permitindo que os problemas que venham a ocorrer durante a utilização de um programa, especialmente no caso de programas interativos, possam ser recuperados e a execução do programa possa ser retomada normalmente. Entretanto, embora muitos projetistas e programadores tenham o hábito de inserir este tipo de tratamento, raramente eles são testados.

É importante que estes caminhos venham a ser testados também, para detectar os erros mais comuns deste tipo de tratamento que são:

- descrição incompreensível do erro (por exemplo, **erro 1356**);
- descrição incorreta do erro (o erro indicado não corresponde ao erro ocorrido);
- ocorrência de intervenção do sistema antes da indicação ou tratamento do erro pela unidade;
- processamento incorreto das condições de exceção (por exemplo, tratar ou idicar a ocorrência de um erro que não aconteceu);
- descrição imprecisa do erro (não informa como localizar ou evitar que o erro volte a ocorrer).

4.2. Procedimento do teste de unidade

O teste de unidade é considerado uma atividade vinculada à codificação. Uma vez desenvolvido o código-fonte, os casos de teste devem ser projetados. O projeto dos casos de teste pode ser uma conseqüência da revisão do código-fonte da unidade, o que vai permitir estabelecer as condições para analizar a unidade segundo os diferentes pontos de vista citados acima. Parte do projeto dos casos de teste deve ser também a especificação dos resultados esperados em cada um deles.

As unidades de software não são programas autônomos, sendo dependentes de outras unidades. Por esta razão, o teste de unidade exige a definição de módulos de software específicos, denominados *Drivers* ou *Stubs*. Como pode ser observado na figura 8.2, um *driver* faz o papel (numa versão bastante simplificada) do programa principal ou de uma unidade de nível superior que ativa a unidade a ser testada comunicando-se com ela através da interface, imprimindo dados que sejam importantes para a tarefa de análise daquela unidade. Os *stubs* são construídos para desempenhar o papel das unidades de nível mais baixo (ou mesmo, de componentes de hardware do sistema) com o fim de viabilizar a execução da unidade sob teste. Tanto os drivers quanto os stubs são programas adicionais a serem desenvolvidos e, por esta razão, devem ser escritos da forma mais simples possível para minimizar o esforço adicional de desenvolvimento de software. Entretanto, esta "simplicidade" nem sempre é facilmente obtida, pois determinadas unidades exigem a existência de drivers ou stubs mais sofisticados para que a atividade de teste seja eficaz.

5. Teste de Integração

O teste de integração é realizado no sentido de assegurar que, uma vez que as unidades foram testadas e seu bom comportamento foi verificado individualmente, estas vão trabalhar de forma cooperativa e harmoniosa quando forem associadas. O problema é colocá-los juntos, considerando que cada um deles foi definido com uma interface. Quem ainda não tentou ligar um plug de três pinos numa tomada com dois furos?

Os erros mais comuns de integração são:

- perdas de dados através das interfaces;
- efeitos inesperados da combinação de duas ou mais funções;
- estruturas de dados globais apresentando problemas;
- imprecisões individualmente aceitáveis podem gerar imprecisões absurdas;
- etc.

5.1. Sistemática do teste de integração

A idéia básica por trás do teste de integração é a construção, passo-a-passo, do programa, associando suas unidades e testando esta associação, até que se atinja a totalidade do programa projetado.

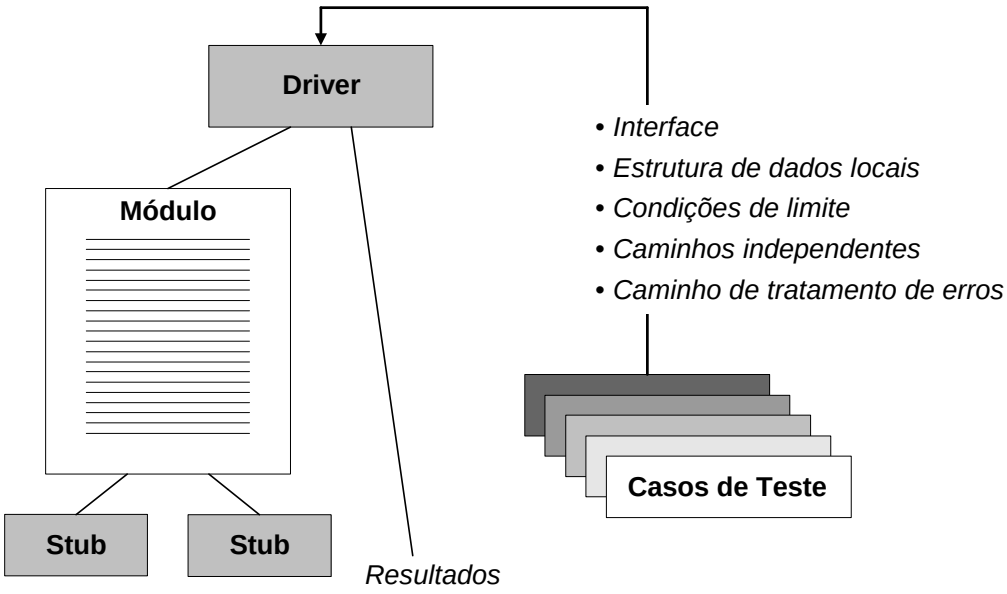


Figura 8.2 - Ambiente de teste de unidade.

Uma abordagem muitas vezes adotada é a abordagem *big bang*, um processo de integração não-incremental na qual todos os módulos são associados e o programa é testado como um todo. Na maioria dos testes realizados segundo esta abordagem, o resultado não é outro senão catastrófico. A correção dos erros torna-se uma tarefa extremamente complexa, principalmente porque é bastante difícil isolar as causas dos erros dada a complexidade do programa completo. Mesmo quando alguns erros são detectados e corrigidos, outros aparecem e o processo caótico recomeça.

A integração incremental tem-se mostrado mais eficiente neste contexto, uma vez que, segundo esta abordagem, o programa vai sendo "construído" aos poucos e testado por partes. Neste tipo de integração, o isolamento das causas de erros e suas correções são obtidos mais facilmente. Além disso, o teste das interfaces pode ser feito com maior eficácia.

As seções a seguir vão apresentar a descrição de duas estratégias de teste de integração bastante clássicas: a integração *top-down* e a integração *bottom-up*.

5.2. Integração Top-Down

A integração *top-down*, ilustrada na figura 8.3, é uma estratégia de integração incremental onde os módulos são integrados segundo um movimento de cima para baixo. O processo de integração inicia-se no módulo de controle principal (representado na figura 8.3 pelo módulo $\mathbf{M_1}$), sendo que o caminho de descida pode ser definido de duas formas: *depth-first* (descida em profundidade) ou *breadth-first* (descida em largura).

Na ilustração da figura 8.3, a descida em depth-first englobaria inicialmente todos os módulos relacionados a um caminho de controle principal. A escolha deste caminho depende, principalmente, das características do software em desenvolvimento. Supondo que o caminho principal fosse aquele liderado pelo módulo $\mathbf{M_2}$, a integração seria feita abrangendo os módulos $\mathbf{M_5}$, $\mathbf{M_6}$ e $\mathbf{M_8}$. Só então os módulos mais à direita seriam integrados. No caso da integração breadth-first, todos os módulos subordinados a um dado nível seriam integrados primeiro. No caso do exemplo, a integração partiria de $\mathbf{M_1}$ e atingiria inicialmente os módulos $\mathbf{M_2}$, $\mathbf{M_3}$ e $\mathbf{M_4}$.

O processo de integração é conduzido considerando 5 passos:

- o módulo principal atua como um *Driver*, e os módulos de nível superior à porção sob teste são substituídos por stubs especialmente construídos;

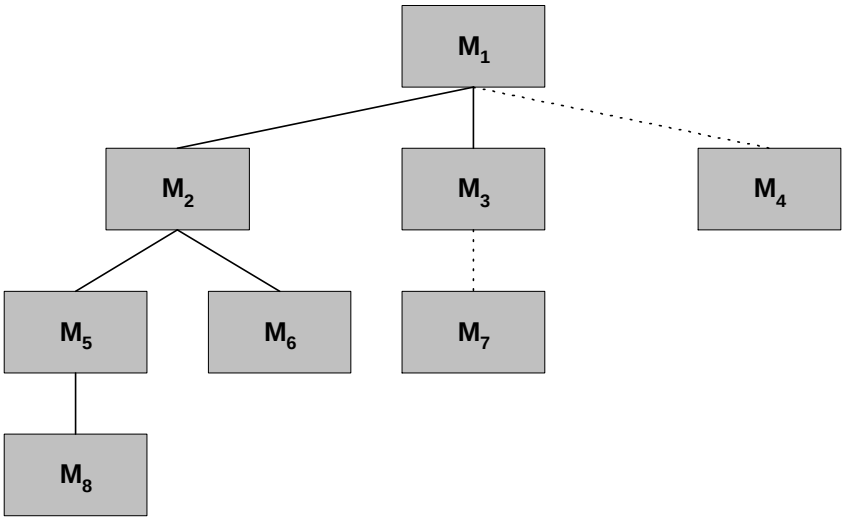


Figura 8.3 - Integração *Top-Down*.

- dependendo do tipo de descida escolhida (*depth-first* ou *breadth-first*), os stubs podem ir sendo substituídos, gradualmente, por módulos reais, já testados;
- à medida que os módulos são integrados, os testes vão sendo realizados;
- quando uma bateria de testes é terminada, os stubs vão sendo substituídos por módulos reais;
- finalmente, pode ser realizado um teste de regressão, que tem por objetivo garantir que novos erros não tenham sido introduzidos durante o processo de descida.

5.3. Integração Bottom-Up

O teste de integração segundo a abordagem *bottom-up* sugere uma integração igualmente incremental, mas no sentido inverso, ou seja, dos níveis mais baixos para o nível mais alto.

Como ilustrado pela figura 8.4, a construção se inicia a partir dos módulos ditos *atômicos* (os módulos do nível mais baixo), facilitando, de certa forma, integração pois, quando se para de um nível para um superior, os módulos subordinados necessários para testar a integração neste novo nível estão disponíveis (e testados).

Os passos normalmente executados nesta estratégia de teste são:

- os módulos de nível mais baixo são agrupados em *clusters* (ou *construções*), que executam uma função específica do software;
- um driver é construído para coordenar a entrada e a saída dos casos de teste;
- o *cluster* é testado;
- os drivers são removidos e os *clusters* são combinados com módulos dos níveis superiores para uma nova etapa, até que todo o programa tenha sido remontado.

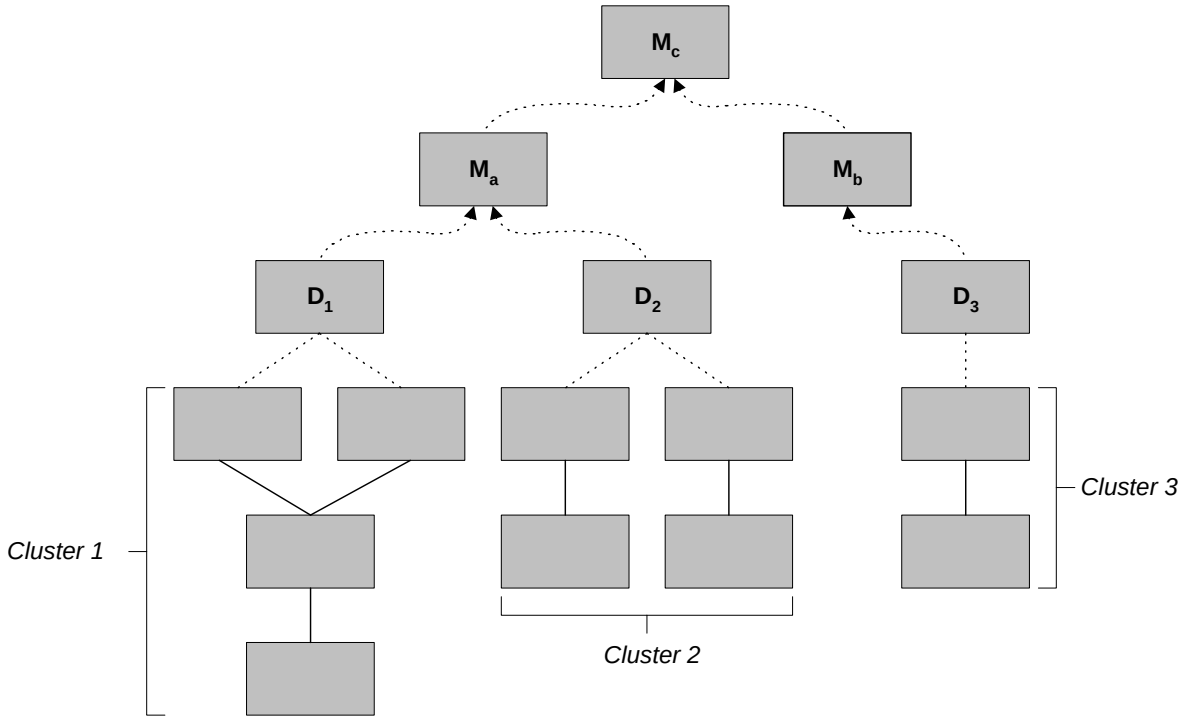


Figura 8.4 - Integração *Bottom-Up*.

5.4. Comparação entre as duas estratégias

Quando se tenta confrontar as duas estratégias descritas nas seções precedentes, não é difícil antecipar que a vantagem de um pode constituir-se na desvantagem do outro. No caso da abordagem *top-down*, a maior desvantagem é, sem dúvida, a necessidade de se construir *stubs* para representar módulos subordinados que não foram suficientemente testados (por razões óbvias). Por outro lado, a possibilidade de se ter resultados sobre as principais de controle (uma vez que são os módulos de mais alto nível que suportam estas funções), constitui-se num ponto a favor desta estratégia.

Já no caso da estratégia *bottom-up*, pode-se dizer que o programa é completamente inexistente até que o último módulo seja finalmente integrado. Mas esta deficiência é largamente compensada pelo fato de haver muito maior facilidade para projetar os casos de teste e pela não exigência na criação de *stubs* para a realização dos testes dos módulos dos níveis mais altos.

6. Teste de Validação

O teste de validação tem por objetivo determinar se o programa, mesmo que funcionando corretamente (em conseqüência das etapas de teste anteriores... de unidade e de integração) apresenta as propriedades e a funcionalidade definidas na etapa de especificação dos requisitos.

Pode-se dizer que um teste de validação bem sucedido é aquele que dá um máximo de respostas sobre a capacidade do software apresentar (ou não) um funcionamento o mais próximo possível daquele esperado pelo cliente (ou usuário). O problema é como medir esta proximidade. Nesta medida, um conjunto de informações importantes são aquelas definidas no documento de especificação de requisitos. Esta é uma das razões pelas quais é importante que se busque, na fase de especificação dos requisitos, utilizar parâmetros mensuráveis para definir os requisitos do software (rever o capítulo 5).

6.1. O processo de teste de validação

O teste de validação é realizado através de um conjunto de testes de caixa preta cujo objetivo é determinar a conformidade do software com os requisitos definidos nas fases preliminares da concepção. Um documento de plano de testes (eventualmente definido na etapa de Análise de Requisitos) define quantos e quais testes serão realizados.

Os requisitos são analizados sob vários pontos de vista: funcionais, desempenho, documentação, interface, e outros requisitos (portabilidade, compatibilidade, remoção de erros, etc.).

O resultado do teste de validação pode apresentar um dos dois resultados:

1. as características de função, desempenho e outros aspectos enquadram-se nos parâmetros estabelecidos na especificação de requisitos;

2. é descoberto um "desvio" das especificações... neste caso, uma lista de deficiências é criada para relacionar de que forma (ou quanto) as características do software obtido afasta-se dos requisitos.

No caso de descoberta de erros, estes dificilmente serão corrigidos antes do período definido para a conclusão do programa. Eventualmente, uma negociação (extensão de prazo de entrega, por exemplo) deverá ser realizada em conjunto com o cliente para possibilitar a eliminação destas deficiências.

6.2. Revisão de configuração

Um aspecto de importância no processo de validação é a chamada *revisão de configuração*. Ela consiste em garantir que todos os elementos de configuração do software tenham sido adequadamente desenvolvidos e catalogados corretamente com todos os detalhes necessários que deverão servir de apoio à fase de manutenção do software. Esta revisão, algumas vezes denominada auditoria, está ilustrada na figura 8.5.

6.3. Testes Alfa e Beta

Apesar do esforço na validação de um software, na maioria das vezes é extremamente difícil para o desenvolvedor ou programador prever as diferentes maneiras como o software será utilizado. Principalmente no caso de softwares interativos, os usuários muitas vezes experimentam comandos e entradas de dados os mais diversos, o que pode ser um verdadeiro atentado à robustez de um sistema. Da mesma forma, dados de saída ou formatos de apresentação de resultados podem parecer ótimos para o analista, mas completamente inadequados para o usuário.

Quando um software é construído sob demanda, uma bateria de testes de aceitação é conduzida, envolvendo o cliente. A duração dos testes de aceitação pode variar entre algumas semanas e vários meses, dependendo da complexidade ou natureza da aplicação.

No caso de um software de prateleira, aquele software que é construído para um segmento do mercado de software e que é destinado à utilização por um grande conjunto grande de usuários, os editores de software fazem uso de um processo denominado *teste alfa* e *teste beta*.

6.3.1. Testes Alfa

Os testes alfa são realizados com base no envolvimento de um cliente, ou numa amostra de clientes, sendo realizado nas instalações do desenvolvedor do software. O software é utilizado num ambiente natural, sendo que o desenvolvedor observa o comportamento do cliente e vai registrando as anomalias detectadas durante a utilização

6.3.2. Testes Beta

Já os testes beta são conduzidos em uma ou mais instalações do cliente pelo usuário final do software.

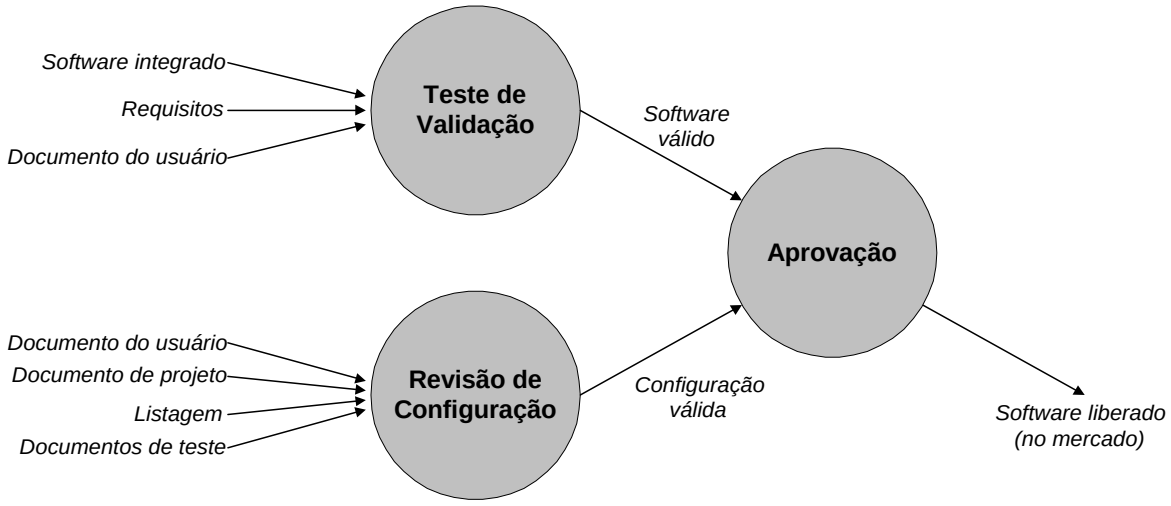


Figura 8.5 - A revisão de configuração.

Neste caso, é praticamente impossível haver um controle da parte do desenvolvedor sobre os erros encontrados. Estes normalmente são registrados pelo próprio usuário e encaminhados ao desenvolvedor na forma de um relatório escrito. Atualmente, muitas empresas fazem uso deste processo, encaminhando (ou disponibilizando via Internet) versões beta de seus softwares, sendo que as informações sobre erros encontrados podem ser fornecidas ao desenvolvedor através de correio eletrônico ou através do acesso a páginas Web especialmente desenvolvidas.

Infelizmente porém, muitos usuários de versões beta não têm a preocupação de contribuir efetivamente para a validação, apenas fazendo uso do software para seu próprio interesse.

7. Teste de Sistema

Como já foi mencionado, o software definido apenas como um elemento de um sistemas que pode envolver hardware, elementos humanos e outros. Por esta razão, uma vez validado, ele será incorporado aos demais elementos.

Um problema comumente encontrado neste tipo de teste é o clássico "apontar o dedo", onde o responsável pelo desenvolvimento de um elemento do sistema tenta livrar-se da responsabilidade do erro, acusando outro elemento do sistema como causador do erro.

Por esta razão, é importante que o Engenheiro de Software tente prever potenciais problemas de integração a outros elementos do sistema e inclua no software caminhos de tratamento de erros para este sistema. Por exemplo, num software de comunicação podem ser incluídos módulos de tratamento de erros que informem quando um determinado dispositivo de E/S não está respondendo corretamente.

O teste de sistema inclui diversas modalidades de teste, cujo objetivo é testar o sistema computacional como um todo. Embora cada teste tenha uma finalidade diferente, o objetivo global acaba sendo atingido, uma vez que estes abrangem todos os elementos constituintes do sistema, verificando se estes foram adequadamente integrados.

7.1. O teste de recuperação

Neste tipo de teste, o objetivo é observar a capacidade do sistema para recuperar-se da ocorrência de falhas, num tempo previamente determinado. Em alguns casos (cada vez mais freqüentes), exige-se que o sistema seja tolerante a falhas, ou seja, que nele seja previsto processamento que permita retomar seu funcionamento mesmo no caso de ocorrência de algumas falhas.

Neste tipo de teste, falhas são provocadas artificialmente (por uma técnica denominada *injeção de falhas*), de modo a analisar a capacidade do sistema do ponto de vista da recuperação. Os valores de tempo exigido para recuperação do sistema (seja por ação automática ou devido à intervenção de um operador humano) devem ser registrados e confrontados aos valores especificados.

7.2. O teste de segurança

O teste de segurança visa garantir que o sistema não vai provocar danos recuperáveis ou não ao sistema pela sua própria ação. Por exemplo, num sistema de informação acadêmica, um aluno deve ter autorização de acesso para a visualização das notas obtidas nas disciplinas que cursou (histórico escolar), mas não deve ter a possibilidade de alterar seus valores.

No teste de segurança, o analista deve desempenhar um papel semelhante ao de um *hacker*, tentando contornar todos os mecanismos de segurança implementados no mesmo. As ações a experimentar são as mais diversas:

- derrubar o sistema como um todo;
- acessar informações confidenciais;
- modificar informações de bases de dados;
- interferir no funcionamento do sistema;
- introduzir vírus de computador no sistema;
- sobrecarregar o sistema pela multiplicação de processos executando;
- etc.

7.3. O teste de estresse

O teste de estresse, como o nome indica, consiste em verificar como o sistema vai se comportar em situações limite. Este limite pode ser verificado sob diferentes pontos de vista, dependendo das características do software. Alguns aspectos que podem ser observados neste contexto são:

- limite em termos de quantidade de usuários conectados a um determinado sistema servidor;
- quantidade de utilização de memória;
- uso em diferentes versões de processadores;
- quantidade de bloqueios encontrados num dado período de utilização;
- etc.

7.4. O teste de desempenho

Este tipo de teste consiste em verificar se os requisitos de desempenho estão sendo atendidos para o sistema como um todo. Este aspecto, que pode não Ter grande importância em alguns sistemas (quais?), torna-se crítico em aplicações envolvendo sistemas embarcados e sistemas multimídias, que pertencem à classe dos sistemas tempo-real. Nestes sistemas, o não atendimento a um requisito de tempo pode afetar de forma irreversível a função do sistema e, no caso de alguns sistemas (os chamados sistemas críticos), a não realização desta função ou o não atendimento a estes requisitos temporais pode resultar em prejuízos catastróficos (risco a vidas humanas ou grandes prejuízos financeiros).

Os testes de desempenho são realizados, normalmente, com o auxílio de instrumentação de hardware e de software que permitam medir como os recursos do sistema estão sendo utilizados. O uso da instrumentação pode facilitar a coleta de dados e o registro de status do sistema nos seus diferentes pontos de funcionamento.

Engenharia de Software

Capítulo 9

Análise e Projeto Orientados a Objetos

1. Introdução

Existem muitas definições para o que se chama, em desenvolvimento de software, de Orientação a Objetos. Estudando a bibliografia da área, observa-se que cada autor apresenta a sua visão do que entende por esta abordagem. Aí vão alguns conceitos:

- a orientação a objeto pode ser vista como a abordagem de modelagem e desenvolvimento que facilita a construção de sistemas complexos a partir de componentes individuais;
- o desenvolvimento orientado a objetos é a técnica de construção de software na forma de uma coleção estruturada de implementações de tipos abstratos de dados;
- desenvolvimento de sistemas orientado a objetos é um estilo de desenvolvimento de aplicações onde a encapsulação potencial e real de processos e dados é reconhecida num estagio inicial de desenvolvimento e num alto nível de abstração, com vistas a construir de forma econômica o que imita o mundo real mais fielmente;
- a orientação a objetos é uma forma de organizar o software como uma coleção de objetos discretos que incorporam estrutura de dados e comportamento.

Visando a que cada leitor passe a ter a sua própria definição do que significa Orientação a Objetos, vamos apresentar aqui os principais conceitos relacionados a esta tecnologia.

2. Orientação a objetos: conceitos

2.1. Características de Objetos

Um **objeto** é algo distinguível que contém *atributos* (ou propriedades) e possui um *comportamento*. Cada objeto tem uma identidade e é distinguível de outro mesmo que seus atributos sejam idênticos. Exemplos de objetos: o parágrafo de um documento, a janela num computador, o aluno Pedro neste curso, o carro do João. O conjunto de valores associados às propriedades do objeto definem o estado deste; o comportamento descreve as mudanças do estado do objeto interagindo com o seu mundo externo, através das *operações* realizadas pelo objeto.

As seções que seguem apresentam outras definições relacionadas à tecnologia de objetos que são de extrema importância à compreensão das atividades e metodologias baseadas em objetos.

2.2. Classe

Uma **classe** é o agrupamento de objetos com a mesma estrutura de dados (definida pelos *atributos* ou *propriedades*) e comportamento (*operações*) [RBP91]. Uma classe é uma abstração que descreve as propriedades importantes para uma aplicação e não leva em conta as outras. Exemplos de classes: Parágrafo, Janela, Aluno, Carro. Cada classe descreve um conjunto possivelmente infinito de objetos individuais. Cada objeto é uma *instância* de classe. Cada instância de classe tem seu próprio valor para cada um dos atributos da classe mas compartilha os nomes e as operações com as outras instâncias de classe.

2.3. Orientação a objetos

Ela se caracteriza principalmente pela abstração, encapsulamento, herança e polimorfismo.

2.4. Abstração

A abstração consiste em enfocar os aspectos mais importantes de um objeto (visão externa; o que é e o que ele faz), ignorando suas características internas (visão interna; como ele deve ser implementado).

2.5. Encapsulamento

O encapsulamento é o empacotamento de dados (atributos) e de operações sobre estes (métodos). No caso da orientação a objetos, os dados não podem ser acessados diretamente mas através de mensagens enviadas para as operações. A implementação de um objeto pode ser mudada sem modificar a forma de acessa-lo.

2.6. Herança

A herança consiste no compartilhamento de atributos e operações entre as classes numa relação hierárquica. Este mecanismo permite a uma classe ser gerada a partir de classes já existentes; por exemplo a classe **automóvel** herda da classe **veículo** algumas propriedades e operações. Uma classe pode ser definida de forma abrangente (como no caso do exemplo anterior) e posteriormente refinada em termos de sub-classes e assim sucessivamente. Cada subclasse *herda* todas as propriedades e operações da sua superclasse, (que não precisam ser repetidas) adicionando apenas as suas especificas.

Esta relação entre classes é uma das grandes vantagens de sistemas orientados a objetos por causa da redução de trabalho resultante durante o projeto e a programação destes.

Existem dois tipos de herança:

- herança simples, ondeuma subclasse tem somente uma superclasse;
- herança múltipla, na qual uma subclasse herda simultaneamente de várias superclasses.

2.7. Polimorfismo

O polimorfismo significa que uma mesma operação pode se comportar de forma diferente em classes diferentes. Exemplo de polimorfismo, a operação **calcular o perímetro** que é diferente para as instâncias de classe circulo e polígono ou a operação **mover** diferente para janela de computador e peça de xadrez. Uma operação que tem mais de um método que a implementa é dita **polimórfica**.

Nos sistemas orientados a objetos, o suporte seleciona automaticamente o método que implementa uma operação correto a partir do nome da operação e da classe do objeto no qual esta se operando, da mesma forma que no mundo real onde o objeto real "tem conhecimento" intrínseco do significado da operação a realizar. Essa associação em tempo de execução é chamada de ligação dinâmica (ou "dynamic binding").

3. Desenvolvimento orientado a objetos

O desenvolvimento orientado a objetos diz respeito aos procedimentos de concepção de sistemas a partir dos conceitos básicos anteriores. Ele corresponde às principais fases do ciclo de vida de software: análise, projeto e implementação.Existem várias técnicas para o desenvolvimento orientado a objetos cujas características serão citadas a seguir.

3.1. OMt — Object Modelling technique

A técnica OMT se distingue pela sua divisão em 4 fases: análise, projeto de sistema, projeto de objetos e implementação. Como principais características, destacam-se a separação clara entre análise e projeto, a inclusão de todos os conceitos da orientação a objetos e de alguns específicos do método. A fase de análise é baseada de 3 diagramas relacionados entre si e que representam os modelos de objetos, dinâmico e funcional.

3.2. técnica de Boocx

A técnica Booch é dividida em 3 fases: análise de requisitos, análise de domínios e projeto, com ênfase maior no projeto. Os diagramas seguintes são providos: de classes, de transição de estados, de objetos, temporais, de módulos e de processos.

3.3. técnica de Coad/Yourdon

Esta técnica utiliza um modelo único para todas as fases (OOA, OOD e OOP), o que torna mais simples e compreensível o desenvolvimento.

3.4. técnica de Sxlaer/Mellor

Ela fornece um conjunto integrado de modelos de análise e que depois são traduzidos (Recursive Design) durante o projeto.

3.5. técnica OOSE (Jacobson)

A técnica OOSE é centrada em casos-de-uso (use-cases) e permite durante a análise aprofundar o entendimento de como o sistema deve ser realmente utilizado.

3.6. técnica Fusão

A Fusão corresponde a uma integração das técnicas OMT e de Booch na qual os dois modelos de objeto e de interface visam representar os aspectos estáticos e dinâmicos do problema.

3.6. técnica UML

UML ou *Unified Modeling Language* é uma unificação dos métodos OMT, Booch e OOSE que está sendo submetido a OMG para a padronização.

A técnica OMT será descrita mais detalhadamente nos itens seguintes.

4. MODELAGEM ORIENTADA A OBJETOS

Um modelo é uma abstração que tem como propósito entender um problema antes de solucioná-lo. A partir de modelos, é possível simular e testar sistemas antes de construí-los, facilitar a comunicação com os usuários e os outros membros da equipe de desenvolvimento, visualizar e reduzir a complexidade dos problemas a tratar.

No modelo OMT, temos 3 visões combinadas que permitem representar o sistema:

- um **modelo objeto** para representar os aspectos estáticos, estruturais, de "dados" de um sistema;
- um **modelo dinâmico** para representar os aspectos temporal, comportamental e de "controle" de um sistema;
- um **modelo funcional** para representar os aspectos transformacionais e de "função" de um sistema.

Esses modelos não são independentes, existem interconexões limitadas e explícitas.

4.1. O modelo objeto

O modelo objeto descreve a estrutura de objetos no sistema: sua identidade, suas relações com os outros objetos, seus atributos e suas operações. Este modelo tenta capturar a estrutura estática dos componentes do mundo real que pretende-se representar. O modelo objeto tem uma representação gráfica na forma de diagramas contendo as classes de objetos com seus atributos e operações, e organizados segundo hierarquias e associações com os diagramas de outras classes.

4.1.1. Objetos e classes

Todos os objetos tem uma identidade e são distinguíveis. Eles são instancias de classes de objetos que descrevem grupos de objetos com similaridade nas propriedades (atributos), comportamento (operações), relações com os outros objetos e semântica. A notação gráfica para representar objetos, classes e suas relações é composta de dois tipos de diagramas, mostrados na figura 9.1:

- um **diagrama de classe**, que representa classes de objetos e tem a função de ser um esquema, um padrão ou um "template" para os dados;
- um **diagrama de instância**, utilizado para representar instâncias de classes e tem o objetivo de descrever como um conjunto particular de objetos se relaciona com outro.

O atributo é colocado na segunda parte da caixa (ver figura 9.2). Cada nome de atributo pode ser seguido por detalhes opcionais tais como tipo (precedido por "**:**") e valor default (precedido de "**=**"). Identificadores (explícitos) de objeto não são necessários no modelo objeto pois cada objeto já tem a sua propria identidade (a partir de seus valores).

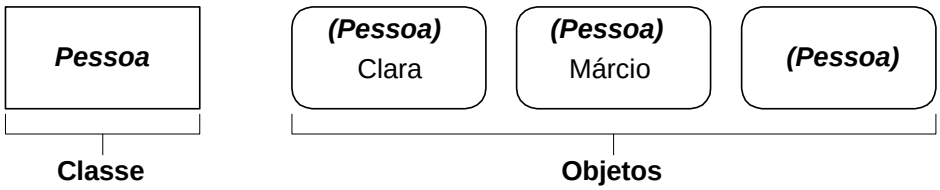


Figura 9.1 – Ilustração de Classes e Objetos.

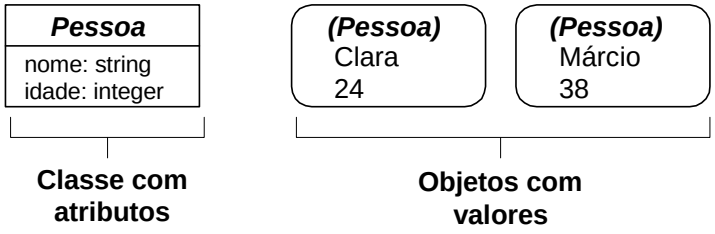


Figura 9.2 – Ilustração de Atributos e Valores.

Uma operação pode ser aplicada a ou por objetos numa classe. A mesma operação pode se aplicar a várias classes diferentes (polimorfismo). Um método é a implementação de uma operação para uma classe. Quando uma operação tem métodos em várias classes, eles devem ter a mesma assinatura (em número e tipos de argumentos). As operações se encontram na terceira parte da caixa, como ilustrado na figura 9.3. Cada operação pode ser seguida de detalhes opcionais tais como lista de argumentos (colocada entre parênteses após o nome, separados por "**,**") e tipo de resultado (precedido por "**:**"). A notação generalizada para classes de objetos se encontra na figura 9.4.

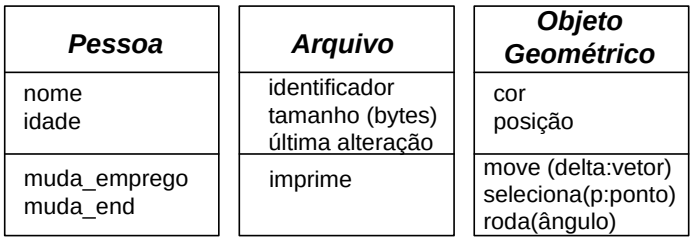


Figura 9.3 – Operações com Objetos.

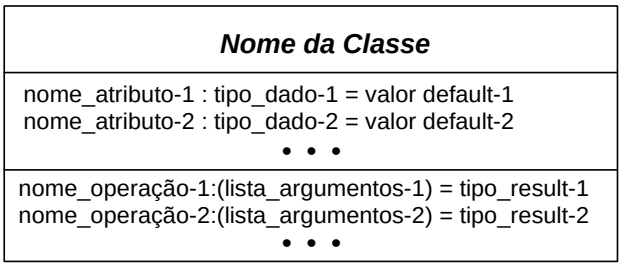


Figura 9.4 – Generalização da notação de modelagem de objetos.

4.1.2. Ligações e associações

A *ligação* ou *link* é uma conexão física ou conceitual entre instâncias de objetos. Por exemplo: Vitório Mazzola **trabalha para** UFSC. Uma ligação é uma túpla matemática correspondente a uma lista ordenada de instâncias de objetos. Uma ligação é uma instância de uma associação.

Uma *associação* ou *association* descreve um grupo de ligações com estrutura e semântica comuns. Por exemplo: uma pessoa **trabalha para** uma instituição. Todas as ligações de uma associação conectam objetos das mesmas classes. O conceito de associação é similar aquele usado na área de base de dados.

Associações e ligações são descritas por verbos. Associações são inerentemente bidirecionais e podem ser lidas nos dois sentidos. Por exemplo: num primeiro sentido pode-se ler Vitório Mazzola **trabalha para** UFSC e no sentido inverso, UFSC **emprega** Vitório Mazzola). Associações são muitas vezes implementadas em linguagens de programação como apontadores (atributo de um objeto que contêm referencia explícita a um outro) de um objeto para outro. Por exemplo: objeto *Pessoa* pode conter atributo *empregado* que aponta para um conjunto de objetos *Instituição*.

Uma ligação mostra a relação entre 2 ou mais objetos. A figura 9.5 mostra uma associação um-a-um e as ligações correspondentes. Observa-se, na notação OMT, que a associação corresponde a uma linha entre classes e que os nomes de associação são em itálico.

As associações podem ser binárias, terçárias ou de ordem maior. De fato, a maior parte é binária ou qualificada (i.e., é uma forma especial de terçária que será comentado a seguir). A simbologia OMT para terçária ou n-ário é um losango com linhas conectando este as classes relacionadas. O nome da associação é escrito dentre do losango; entretanto, os nomes de associação são opcionais. A figura 9.6 mostra exemplos de associações e ligações terçárias.

A **multiplicidade** especifica quantas instâncias de uma classe podem se relacionar a uma única instância de classe associada. A multiplicidade restringe o número de objetos relacionados. A multiplicidade depende de hipóteses e de como se define os limites do problema. Requisitos vagos tornam a multiplicidade incerta. Não é possível de determinar muito cedo a multiplicidade no processo de desenvolvimento de software; num primeiro tempo, determina-se objetos, classes, ligações e associações e depois decide-se a respeito de multiplicidade. A distinção mais importante é entre “um” e “vários”; entretanto existem símbolos especiais para “exatamente um”, “um ou mais”, “intervalos”, “”zero ou mais”, “zero ou um”, conforme mostra a figura 9.7.

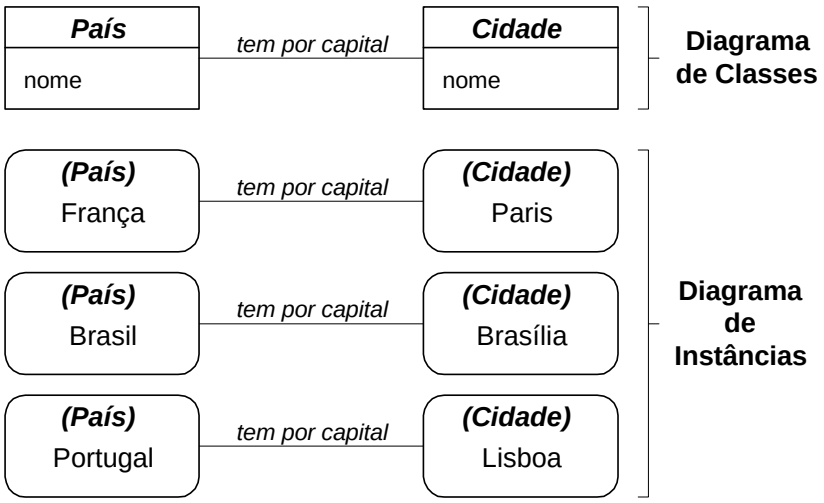


Figura 9.5 – Associação um-para-um e links.

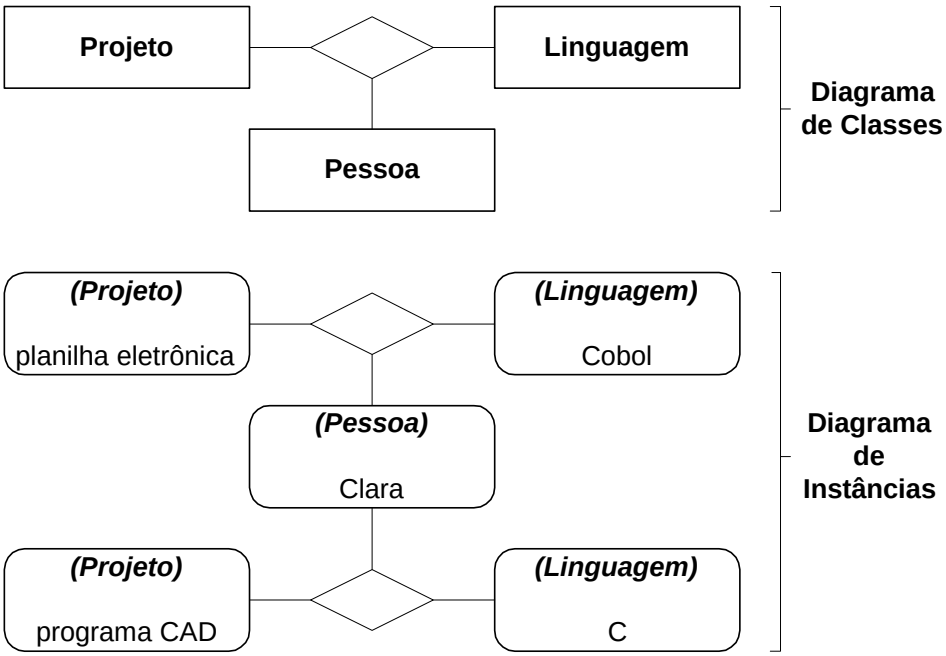


Figura 9.6 – Associação ternária e links.

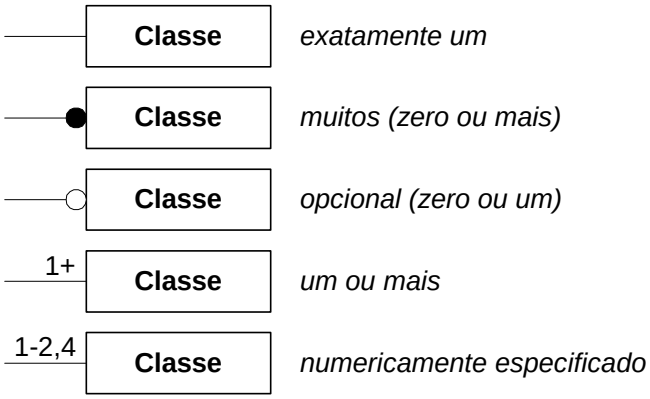


Figura 9.7 – Notação para associações múltiplas.

4.1.3. Ligações e associações avançadas

Atributo de ligação. Similarmente ao caso da classe onde um atributo é uma propriedade dos objetos desta, um atributo de ligação é uma propriedade das ligações numa associação. A figura 9.8 apresenta o caso onde *permissão-de-acesso* é um atributo da associação *Acessível-por*. Cada atributo de ligação tem um valor para cada ligação, conforme pode ser visto nesta figura ainda. É ainda possível ter atributos de ligações para associações vários-a-um conforme visto na figura 9.9 e também para ligações terçárias. Apesar de poder substituir os atributos de ligação por atributos de objetos da associação, é preferível manter atributos de ligação.

Modelagem de associação como classe. Às vezes, é útil modelar uma associação como classe. Cada ligação se torna uma instância da classe.

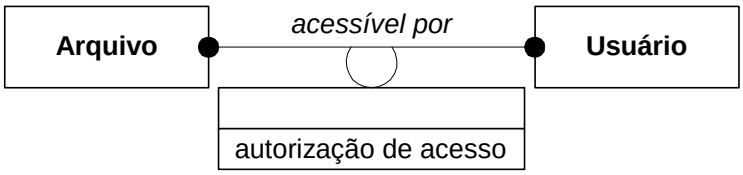


Figura 9.8 – Atributo de link para associação muitos-para-muitos.

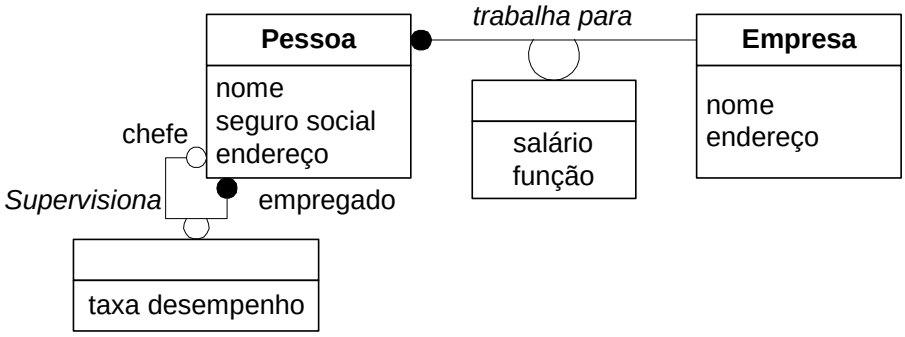


Figura 9.9 – Atributos de link para associação um para vários (papéis).

A figura 9.10 mostra um exemplo disto. Esta opção é particularmente útil quando ligações podem participar em associações com outros objetos ou quando ligações são sujeitas a operações.

Papel. Um *papel* ou *role* é uma extremidade de uma associação. Uma associação binária tem dois papeis, um em cada extremidade, cada um deles podendo ter seu nome. Cada papel numa associação binária identifica um objeto ou um conjunto de objetos associado com um objeto na outra extremidade. Nomes de papel são necessários em associações entre objetos da mesma classe, conforma consta na figura 9.9.

Ordenamento. Usualmente, do lado de uma associação "vários", os objetos não têm ordem explícita e podem ser vistos como um conjunto. Entretanto, às vezes, os objetos são explicitamente ordenados, o que é representado por *{ordered}* perto da indicação de multiplicidade.

Qualificação. Uma associação qualificada relaciona duas classes de objetos e um qualificador. O qualificador é um atributo especial que reduz a multiplicidade efetiva de uma associação, distinguindo objetos dentro do conjunto na extremidade "vários" da associação.

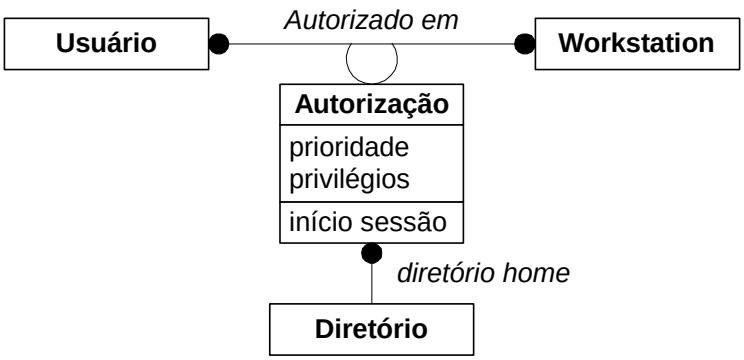


Figura 9.10 – Modelando uma associação como Classe.

4.1.4. Agregação

A agregação é uma relação "parte-todo" ou "uma-parte-de" na qual os objetos representando os componentes de algo são associados com um objeto representando a junção deles ("*assembly*"). Algumas propriedades da classe junção se propagam às classes componentes, possivelmente com alguma modificação local.

A agregação é definida como uma relação entre uma classe junção e uma classe componente. A agregação é então uma forma especial de associação. A simbologia da agregação é similar a da associação (linha), exceto pelo fato de um pequeno losango indicar o lado da junção da relação, conforme a figura 9.11.

4.1.5. Generalização e herança

Generalizações e herança são abstrações poderosas que permitem compartilhar similaridades entre as classes, apesar de preservar suas diferenças.

Generalização é uma relação do tipo "é um" entre uma classe e uma ou mais versões refinadas. A classe que está sendo refinada é uma superclasse e a classe refinada é uma subclasse. Por exemplo, **máquina de comando númerico** é uma superclasse de **torno**, **fresadora**. Atributos e operações comuns a um grupo de subclasses são reagrupados na superclasse e compartilhado por cada subclasse. Diz-se que cada subclasse herda as características de sua superclasse.

A notação para a generalização é um triângulo conectando uma superclasse a suas subclasses, conforme visto na figura 9.12. As palavras próximas ao triângulo no diagrama são discriminadores (i.e. atributos de um tipo enumeração que indica qual propriedade de um objeto está sendo abstraído por uma relação de generalização particular). Os discriminadores estão inerentemente em correspondência um-a-um com as subclasses da generalização.

É possível relação de herança em vários níveis. A herança em 2 ou 3 níveis (até 5 em alguns casos) de profundidade é aceitável; já a mais de 10 níveis é excessiva.

A generalização facilita a modelagem a partir de classes estruturadas e da captura do que é similar e do que é diferente nas classes. A herança de operação é de grande ajuda durante a implementação para facilitar a reutilização de código.

Utiliza-se a generalização para se referenciar à relação entre classes, enquanto a herança diz respeito ao mecanismo de compartilhar atributos e operações usando a relação de generalização. Generalização e especialização são dois pontos de vista da mesma relação, no primeiro caso vista a partir da superclasse e no segundo da subclasse: a superclasse generaliza a subclasse e a subclasse refina ou especializa a superclasse.

4.1.6. Módulo

Um módulo é uma construção lógica para reagrupar classes, associações e generalizações. Um módulo captura uma perspectiva ou uma vista de uma situação, como no caso de um **prédio**, existem várias visões correspondentes a **planta elétrica**, **hidráulica**, **de calefação**. Um modelo objeto consiste de um ou vários módulos. A noção de módulo força a particionar um modelo objeto em peças gerenciaveis. Uma mesma classe pode ser referenciada em módulos diferentes. De fato, referenciando a mesma classe em múltiplos módulos é o mecanismo para interligar módulos entre sí.

Figura 9.11 – Agregação.

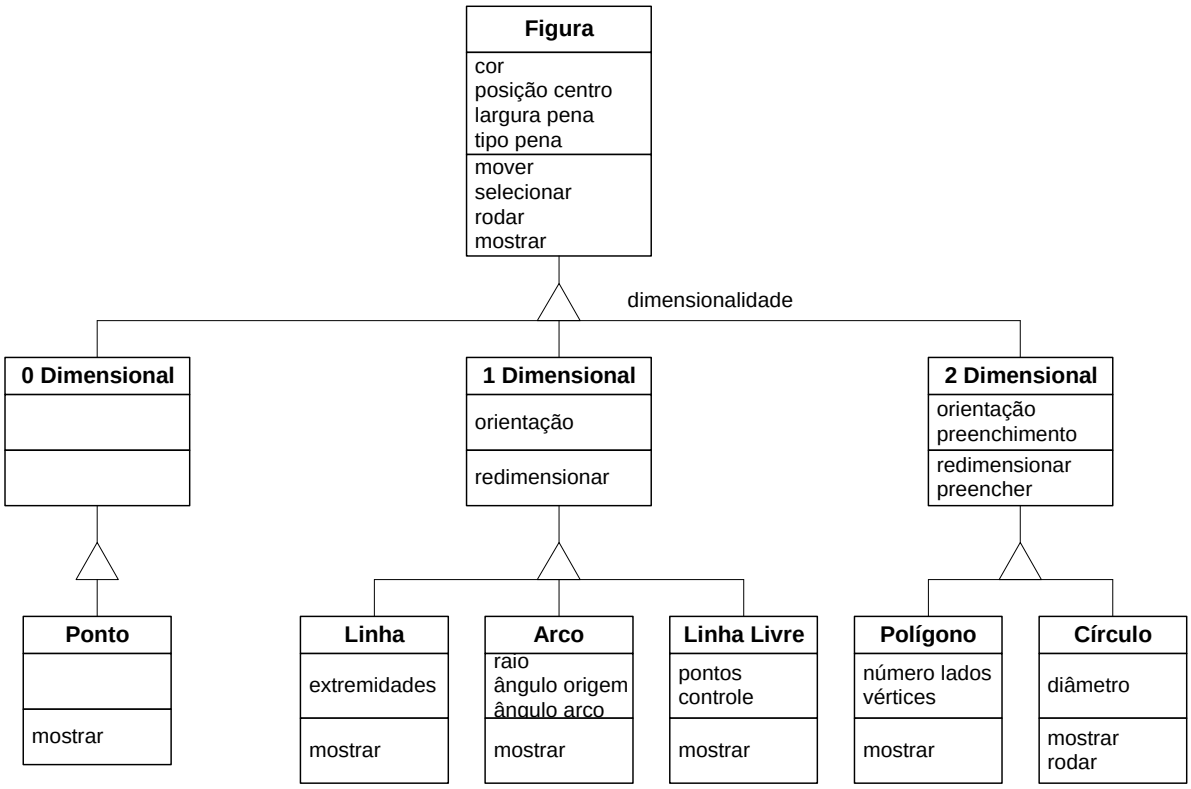


Figura 9.12 – Herança aplicada a figuras gráficas.

4.2. O modelo dinâmico

O modelo dinâmico descreve os aspectos do sistema que dizem respeito ao tempo e à seqüência de eventos (operações). Este modelo tenta capturar o controle, aspecto de um sistema que descreve as seqüências de operação que ocorrem em resposta a estímulos externos, sem levar em conta o que as operações fazem, quem as ativa e como são implementadas. Os conceitos utilizados nesta modelagem dinâmica são os de eventos que representam os estímulos externos e de estados que representam os valores de objetos.

A representação gráfica é um diagrama de estados que representa os estados e a seqüência de eventos permitidos num sistema para uma classe de objetos. Os estados e eventos podem ainda serem organizados de forma hierárquica e representados num diagrama de estados estruturado.

4.2.1. Eventos e estados

Um **estado** é caracterizado pelos valores dos atributos e pelas ligações mantidas por um objeto. Um **evento** corresponde a um estimulo individual de um objeto a um outro. O **diagrama de estados** representa o modelo de eventos, estados e transições de estado para um a classe dada. O **modelo dinâmico** consiste de vários diagramas de estados, um para cada classe com comportamento dinâmico importante; ele mostra o modelo de atividade para um sistema completo. Cada maquina de estados se executa de forma concorrente e pode mudar de estado independentemente. Os diagramas de estado para as várias classes combinam num modelo dinâmico único através de eventos compartilhados.

Um **evento** é algo que ocorre num instante de tempo e que não tem duração. Um evento pode preceder ou seguir outro evento ou pode não ter relação entre eventos (neste caso, são ditos concorrentes). Cada evento é uma ocorrência única; entretanto é possível reagrupa-los em classes de eventos e dar a cada uma delas um nome que indica uma estrutura e um comportamento comuns. Alguns eventos são simples sinais mas muito

outros tem atributos indicando a informação que eles transportam. O tempo no qual o evento ocorre é um atributo implícito de todos os eventos. Um evento transporta a informação de um objeto a outro; os valores de dados transportados por um evento são seus atributos. Os eventos incluem as condições de erro e as ocorrências normais.

Um **cenário** é uma seqüência de eventos que ocorre durante uma execução particular de um sistema. Ele pode incluir todos os eventos do sistema ou apenas eventos gerados por certos objetos no sistema. A seqüência de eventos e os objetos que trocam eventos podem ser mostrados juntos num diagrama de **rastro de eventos**. A figura 9.13 mostra o cenário e o rastro de eventos para uma chamada telefônica.

Um **estado** é uma abstração dos valores dos atributos e das ligações de um objeto. Um estado especifica a resposta do objeto à eventos de entrada. A resposta de um objeto à um evento pode incluir uma ação ou uma mudança de estado pelo objeto.
Um estado tem uma duração; ele ocupa um intervalo de tempo entre dois eventos. Na definição de estados, pode se ignorar atributes que não afetam o comportamento do objeto. O estado é caracterizado por um nome e uma descrição contendo a seqüência de eventos que leva ao estado, a condição que o caracteriza e os eventos aceitos neste estado com a ação que ocorre e o estado futuro. O estado pode incluir os valores de suas ligações.

O **diagrama de estados** relaciona estados e eventos. A mudança de estado causada por um evento é chamada de **transição**. A figura 9.14 mostra o diagrama de estados de uma linha telefônica. Os diagramas de estado podem representar ciclos de vida uma-vez (com um estado inicial e um estado final) que representam objetos com vida finita ou malhas continuas como na figura 9.14. Um modelo dinâmico é uma coleção de diagramas de estado que interagem entre si através de eventos compartilhados.

Uma **condição** é uma função booleana de valores objetos. Condições podem serem usados como guardas nas transições, sendo que uma transição guardada dispara quando o evento ocorre e que a condição de guarda é verdadeira.

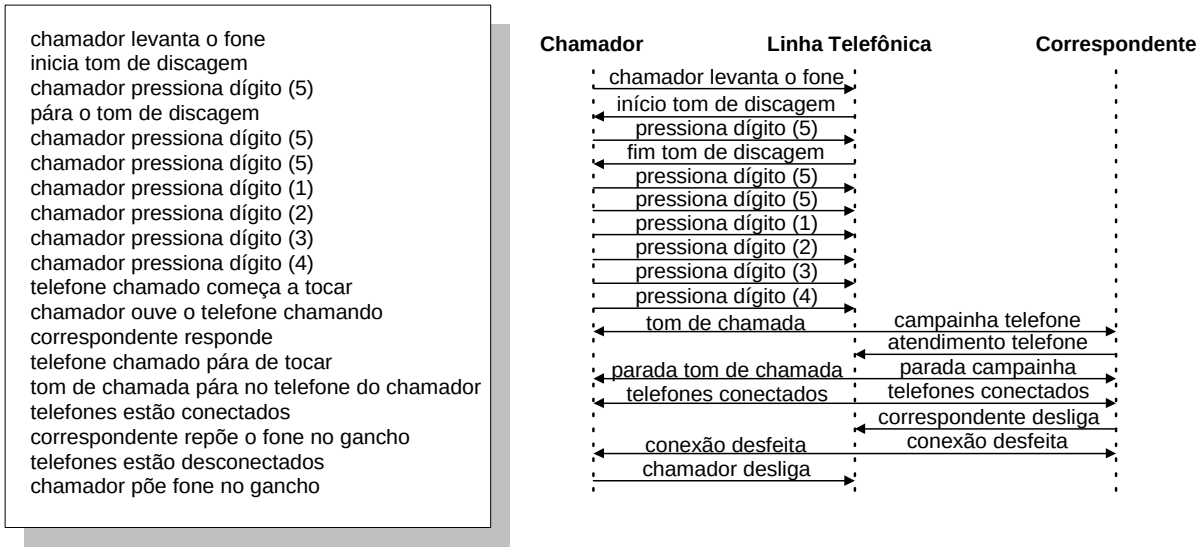


Figura 9.13 – Cenário e traço de eventos para uma chamada telefônica.

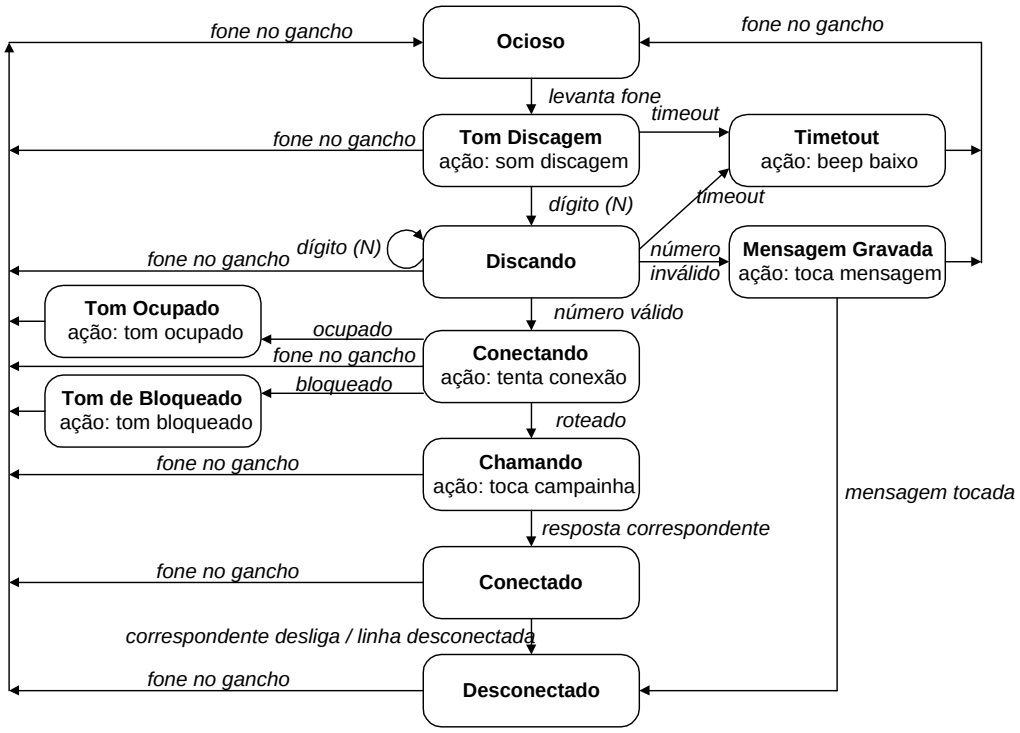


Figura 9.14 – Diagrama de estados de uma linha telefônica.

4.2.2. Operações

Uma descrição do comportamento de um objeto deve especificar o que o objeto faz em resposta a eventos. Operações associadas à estados ou transições são realizadas em resposta aos estados correspondentes ou a eventos.

Uma **atividade** é uma operação que leva tempo para se completar. Ela é associada a um estado. A notação "*do: A*" dentro de um caixa de estado indica que a atividade A inicia na entrada no estado e para na saída.

Uma **ação** é uma operação instantâneo e é associada a um evento. Uma ação representa uma operação cujo a duração é pequena comparada com a resolução do diagrama de estados. Ações podem também representar operações de controle interno. A notação para uma ação numa transição é um "/" seguido do nome da ação, após o evento que a causa. A figura 9.15 representa os diagramas de estados com operações.

4.2.3. Diagramas de estado aninhados

Os diagramas de estado podem ser estruturados para permitir descrições concisas de sistemas complexos. Através de uma **generalização**, é possível expandir, num nível mais baixo, adicionando detalhes, uma atividade descrita num nível superior. É possível organizar estados e eventos em hierarquias com herança de estrutura e comportamento comuns, como no caso da herança de atributos e operações em classes de objetos.

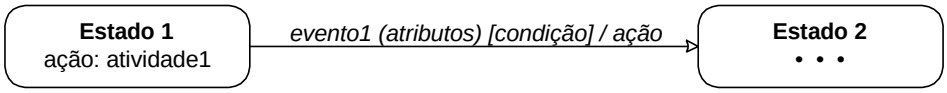


Figura 9.15 – Diagrama de estados (generalização).

A **agregação** permite quebrar um estado em componentes ortogonais, com interação limitada entre eles, da mesma foram como se tem uma hierarquia de agregação de objetos; a agregação é equivalente à concorrência de estados (estados concorrentes correspondem geralmente a agregações de objetos).

Uma **atividade** num estado pode ser expandida como um diagrama de estados de baixo-nível, cada estado representando um degrau da atividade. Atividades aninhadas são diagramas de estados *uma-vez* com transições de entrada e saída, de forma similar a subrotinas. Um diagrama de estados aninhado é uma forma de generalização (relação do tipo "*or*") sobre os estados. Os estados num diagrama aninhado são todos refinamentos (sub-estados) de um estado (super-estado) de um diagrama de alto-nível; a simbologia utilizada corresponde a uma caixa arredondada representando o super-estado e contendo todos seus sub-estados.

Eventos podem também ser expandidos em diagramas de estados subordinados. Os eventos podem ser organizados numa hierarquia de generalização com herança dos atributos de eventos. A hierarquia de eventos permite que diferentes níveis de abstração sejam usados em diferentes lugares num modelo.

4.2.4. Concorrência

Um modelo dinâmico descreve um conjunto de objetos concorrentes, cada um com seu próprio diagrama de estados. Os objetos num sistema são inerentemente concorrentes e podem mudar de estados de forma independente. O estado do sistema total é o resultado dos estados de todos os seus objetos. Um diagrama de estados no caso da junção ("*assembly*") de objetos através de agregação (relação "*and*") é uma coleção de diagrama de estados, concorrentes, um para cada componente. Entretanto, em vários casos, os estados dos componentes interagem; transições guardadas para um objeto podem depender do estado de um outro objeto.

A concorrência pode também ocorrer dentro do estado de um único objeto, quando o objeto pode ser particionado em subconjuntos de atributos ou ligações, cada um com seu próprio sub-diagrama; na notação adotada, os sub-diagramas são separados por linhas pontilhadas.

4.2.5. Modelagem dinâmica avançada

Ações de entrada e saída. Como alternativa a mostra ações nas transições, pode-se associar ações com a entrada ou a saída de um estado. O nome da ação de entrada seguindo *entry/* na caixa representando o estado e o nome da ação de saída seguindo *exit/* nesta são as notações adotadas.

Ações internas. Um evento pode forçar uma ação a ser realizada sem causar uma mudança de estado; o nome do evento é escrito na caixa representando o estado, seguido por um "/" e o nome da ação. A figura 9.16 representa a notação utilizada para os diagramas de estado.

Transição automática. Existe a possibilidade de representar transições automáticas que disparam quando a atividade associada ao estado fonte é completado; é representado por uma seta sem nome de evento. No caso do estado não ter atividade associada, a transição sem nome dispara logo após ter entrado no estado. Chama-se este tipo de transição automática de lambda (λ) transição.

Eventos de envio. Um objeto pode realizar uma ação de enviar um evento a um outro objeto. Um sistema de objetos interage entre si. Na notação OMT, pode se utilizar a palavra *Send* ou uma linha pontilhada que leva da transição a um objeto, conforme indicado na figura 9.16.

Sincronização de atividades concorrentes. Quando um objeto deve realizar duas ou mais atividades de forma concorrente, é necessário dividir (“split”) o controle em partes concorrentes e depois juntá-las (“merge”), completando as atividades antes da progressão do objeto para um próximo estado. Uma seta que se ramifica para a divisão e uma seta com a extremidade ramificada para a junção são as representações na notação OMT.

4.3. O modelo funcional

O modelo funcional descreve os aspectos do sistema que dizem respeito com as transformações de valores: funções, mapeamentos, restrições e dependências funcionais. Este modelo captura *o que* o sistema faz sem levar em conta o *como* e o *quando* ele faz.

O modelo funcional é representado por vários diagramas de fluxo de dados (DFDs) que mostram as dependências entre valores e o calculo de valores de saída a partir de valores de entrada e de funções. O modelo funcional inclui também as restrições entre valores no modelo objeto.

4.3.1. Diagramas de Fluxo de Dados

O modelo funcional consiste de múltiplos diagramas de fluxo de dados que especificam o significado de operações e restrições. Um diagrama de fluxo de dados (DFD) mostra a relação funcional dos valores calculados pelo sistema, incluindo valores de entrada, saída e armazenamento de dados internos.

Um DFD contem *processos* que transformam dados, *fluxos de dados* que movimenta dados, objetos *atores* que produzem e consomem dados, objetos *armazenamento de dados* que estocam os dados.

Processos. Um processo transforma valores de dados. Um processo é implementado como um método de uma operação de uma classe de objetos. O objeto alvo é usualmente um dos fluxos de entrada, especialmente se a mesma classe de objeto é também um fluxo de saída. Em alguns casos, o objeto alvo é implícito.

Fluxos de dados. Um fluxo de dados conecta a saída de um objeto ou de um processo a entrada de um outro objeto ou processo Ele representa um valor de daos intermediário num calculo. O valor permanece sem mudança no fluxo de dados.

Atores. Um ator é um objeto ativo que conduz o diagrama de fluxo de dados produzindo ou consumindo valores. Atores são ligados as entradas e saídas de um diagrama de fluxo de dados. Eles podem serem vistos como fontes e receptores de dados.

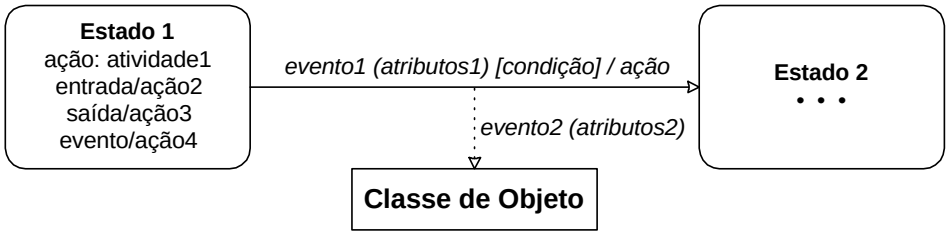


Figura 9.16 –Ilustração da notação extendida dos diagramas de estado.

Armazenadores de dados. Um armazenador de dados é um objeto passivo do diagrama de fluxo de dados que armazena dados para uma cesso futuro.> como no caso de um ator, um armazenador não gera operações sobre ele mesmo mas simplesmente responde a pedidos para armazenar e acessar dados. O acesso pode ser feito em ordem diferente do

armazenamento. Destaca-se que atores e armazenadores de dados são objetos que se diferenciam pelo seu comportamento e uso; atores podem ainda serem implementados como dispositivos externos e armazenadores como arquivos.

Diagramas de fluxo de dados aninhados. Um DFD é particularmente útil para mostrar a funcionalidade de alto-nível de um sistema e sua quebra em unidades funcionais menores. Um processo pode ser expandido num outro DFD no qual as entradas e saídas do processos o são também no novo diagrama. Eventualmente, o anhinhamento de diagramas termina com funções simples que devem ser especificadas como operações.

Fluxos de controle. Um DFD não mostra quais caminhos são executados e em que ordem. Decisões e sequenciamento são questões de controle que fazem parte do modelo dinâmico. As vezes, pode ser útil introduzir o fluxo de controle no DFD. O fluxo de controle é uma variável booleana que indica quando um processo pode ser realizado; o fluxo de controle (representado por uma linha pontilhada no DFD que vai de um processo que gera uma variável booleana ao processo a ser controlado) não é um valor de entrada para o processo ele mesmo.

4.3.2. Especificando operações

Processos em DFD devem eventualmente ser implementados como operações sobre objetos. Para cada nível baixo, um processo atômico é uma operação. Processos de nível elevado podem também ser considerados operações. Apesar que uma implementação possa ser organizada de forma diferente da que o DFD representa por causa de otimização. Cada operação pode ser especificada de várias formas como por exemplo: funções matemáticas, tabelas de valores de entrada e saída, equações especificando saída em termos de entrada, pré e pós condições, tabelas de decisão, pseudo-código e linguagem natural.

4.4. Relações entre modelos

Cada modelo descreve um aspecto do sistema mas contem referencias aos outros modelos. O modelo objeto descreve a estrutura de dados sobre a qual os modelos dinâmico e funcional operam. As operações no modelo objeto correspondem aos eventos no modelo dinâmico e as funções no modelo funcional.

O modelo dinâmico descreve a estrutura de controle dos objetos. As ações no diagrama de estados correspondem as funções no diagrama funcional. Os eventos no diagrama de estados se tornam as operações no modelo objeto.

O modelo funcional descreve as funções invocadas pelas operações no modelo objeto e ações no modelo dinâmico. As funções operam sobre as valores de dados especificados pelo modelo objeto. O modelo funcional mostra ainda as restrições sobre os valores objeto

5. ANÁLISE ORIENTADA A OBJETOS

A fase de análise diz respeito ao entendimento e à modelagem da aplicação e do domínio no qual ela opera. Esta fase focaliza *o que* necessita ser feito e não *como* deve ser feito. A etapa inicial da fase de análise consiste no estabelecimento dos requisitos do problema a ser resolvido, fornecendo uma visão conceptual do sistema proposto. O dialogo subsequente com o usuário, o conhecimento da área e a experiência adquirida do mundo real são elementos adicionais que servem de entrada para a análise. A saída desta fase de análise é um modelo “formal” que captura os aspectos essenciais do sistema: objetos e suas relações, fluxo dinâmico de controle e transformação funcional de dados. No caso da metodologia OMT, obtém-se os modelos objeto, dinâmico e funcional.

5.1. Modelo objeto

Para construir o modelo objeto, é necessário seguir os seguintes passos:

- identificar objetos e classes;
- preparar um dicionário de dados;
- identificar associações (incluindo agregações) entre objetos;
- identificar atributos de objetos e ligações;
- organizar e simplificar classes de objetos utilizando herança;
- verificar os caminhos de acesso;
- iterar e refinar o modelo;
- agrupar as classes em módulos.

5.1.1. Modelo dinâmico

Para construir o modelo dinâmico, é necessário seguir os seguintes passos:

- preparar cenários de seqüências de interação típicas;
- identificar eventos entre objetos;
- preparar um rastro de eventos para cada cenário;
- construir um diagrama de estados;
- equiparar eventos entre objetos para verificar a consistência.

5.1.2. Modelo funcional

Para construir o modelo funcional, é necessário seguir os seguintes passos:

- identificar valores de entrada e saída;
- construir um DFD mostrando dependências funcionais;
- descrever funções;
- identificar restrições (i.e dependências funcionais) entre objetos;
- especificar um critério de otimização

5.1.3. Acrescentando as operações

O estilo de análise apresentado não da ênfase nas operações; entretanto é necessário distinguir os vários tipos de operações durante a fase de análise: operações a partir do modelo objeto, de eventos, de ações e atividades no diagrama de estados, de funções (DFD).

6. PROJETO ORIENTADO A OBJETOS

O projeto orientado a objetos se divide em projeto do sistema e projeto do objeto, descritos sucintamente a seguir.

6.1. Projeto do Sistema

O projeto do sistema consiste em tomar decisões de alto nível sobre o *como* o problema será resolvido e a solução construída. O projeto de sistemas inclui decisões sobre a arquitetura do sistema i.e. sobre a organização do sistema em subsistemas , a alocação de subsistemas a componentes de hardware ou de software e sobre a política e a estratégia que formam o quadro no qual o projeto detalhada poderá ser desenvolvido.

O projetista do sistema deve tomar as seguintes decisões:

- organizar o sistema em subsistemas, a partir das noções de camadas (subdivisão horizontal) e de partições (subdivisão vertical);
- identificar a concorrência inerente ao problema e definir as tarefas concorrentes;
- alocar os subsistemas a processadores e tarefas;
- escolher uma abordagem para gerenciar os armazenadores de dados;
- determinar os mecanismos para manusear o acesso aos recursos globais;
- escolher a implementação do controle no software;
- determinar o manuseio das condições limites;
- estabelecer as prioridades entre os compromissos.

6.2. Projeto do objeto

A fase de projeto de objeto determina as definições completas das classes e associações utilizadas na implementação, bem como as interfaces e os algoritmos dos métodos utilizados par implementar as operações. Nesta fase, são adicionados objetos internos para a implementação e otimizado as estruturas de dados e os algoritmos. O projetista deverá escolher entre várias formas de implementação, levando em conta questões como minimização do tempo de execução, da memória e outras medidas de custo. A otimização do projeto deve ainda levar em conta as facilidades de implementação, manutenção e expansão.

Durante o projeto do objeto, o projetista deve realizar os passos seguintes:

- combinar os três modelos para obter as operações sobre as classes;
- projetar os algoritmos para implementar as operações;
- otimizar os caminhos de acesso aos dados;
- implementar o controle para as interações externas;
- ajustar a estrutura de classes para adicionar a herança;
- realizar o projeto das associações;
- determinar a representação do objeto;
- empacotar classes e associações nos módulos;
- documentar as decisões de projeto.

Engenharia de Software

BIBLIOGRAFIA

EDWARD YOURDON
"Modern Structured Analysis", Prentice-Hall Inc., New York, 1989.

IAN SOMMERVILLE
"Software Engineering", 2a. Edição, Addison-Wesley, Londres, 1985.

JAMES RUMBAUGH et al.
"Object-Oriented Modeling and Design", Ed. Prentice-Hall, New Jersey, 1991.

PANKAJ JALOTE
"An Integrated Approach to Software Engineering", Springer-Verlag, New York, 1991.

RICHARD FAIRLEY
"Software Engineering Concepts", McGraw-Hill, New York, 1985.

ROGER S. PRESSMAN
"Engenharia de Software ", 3ª. Ed., McGraw-Hill/Makron Books do Brasil, São Paulo, 1995.