Universidade de São Paulo
Instituto de Matemática e Estatística

MAC5701 — Tópicos em Ciência da Computação
Relatório de Estudos

Simulação de Sistemas Dinâmicos Estocásticos

Edgar Szilagyi
Orientador: Prof. Marco Dimas Gubitoso

São Paulo, Junho de 2003

Prefácio

A primeira parte deste texto inicia-se com uma introdução aos sistemas dinâmicos através de uma breve introdução histórica e de sua relevância para as ciências exatas, seguido da apresentação de sua definição formal e de algumas de suas propriedades. No segundo capítulo, é dada uma breve visão sobre máquinas sequenciais e autômatos; e no terceiro, uma introdução aos processos estocásticos. Por fim, é mostrado um estudo de caso em biologia molecular onde é proposto um modelo baseado no sistema dinâmico estocástico discreto e sua representação em termos de estruturas de dados.

Na segunda parte, encontram-se alguns modelos de processo em engenharia de software e alguns tópicos em programação orientada a objetos em C++, bem como a descrição de dois padrões aplicados no *design* do programa de simulação. O último capítulo traz uma breve introdução sobre a arquitetura distribuída CORBA.

De uma maneira geral, esse texto resume os estudos realizados durante um programa de mestrado cujo objetivo principal era o de implementar um programa de simulação para auxiliar no estudo particular do problema das redes de regulação de expressão gênica. Embora a proposta para este trabalho tenha incluído a descrição da linguagem de especificação de modelos para o simulador, esta não foi apresentada por duas razões: A primeira é que no atual estado de desenvolvimento do programa, a mesma não está suficientemente madura, estando ainda sujeita a discussões. O segundo motivo é que se trata de algo muito específico a um problema particular, fugindo do tema geral de "tópicos em ciência da computação".

Acredito que esse texto possa ser útil àqueles que queiram ter um primeiro contato com a vasta área de pesquisa dos sistemas dinâmicos ou interessado em desenvolver algum programa de simulação, seja sobre sistemas estocásticos ou não, discretos ou contínuos. Também pode ser útil para aqueles que desejam ter uma noção introdutória aos modelos de processo em engenharia de software, à programação orientada a objetos em C++ e padrões e finalmente sobre a arquitetura distribuída CORBA.

Sumário

I Matemática e Modelagem **3**

1 Sistemas Dinâmicos **4**
1.1 A astronomia de Aristóteles a Galileu 4
1.2 O que são sistemas dinâmicos e para que servem 5
1.3 Classificações 5
1.3.1 Quanto ao determinismo 6
1.3.2 Quanto à variável temporal 6
1.3.3 Quanto ao espaço ser contínuo ou discreto 7
1.3.4 Quanto ao sistema ser linear ou não-linear 7
1.3.5 Quanto a parâmetros fixos ou variáveis 8
1.3.6 Quanto a parâmetros concentrados ou distribuídos 8
1.3.7 Quanto ao sistema ser dinâmico ou instantâneo 8
1.4 Definição formal e propriedades gerais 9
1.5 A importância das simulações 10

2 Máquinas Sequenciais e Autômatos **11**
2.1 Visão informal 11
2.2 Máquinas sequenciais 12
2.2.1 Completa 12
2.2.2 Não necessariamente completa 12
2.2.3 Infinita 13
2.3 Autômatos 13
2.3.1 Definição 13

3 Processos Estocásticos **14**
3.1 Introdução 14
3.2 Cadeias de Markov finitas 14
3.3 Exemplo 15

4 Construção de um Modelo para Redes de Expressão Gênica **17**
4.1 O problema bioquímico da expressão gênica 17
4.2 Modelagem do problema 18
4.2.1 O modelo estocástico 19
4.3 A representação em estruturas de dados 19

II Engenharia de Software e POO **21**

5 Modelos de Processo de Software **22**
5.1 Introdução . . . 22
5.2 Modelo em cascata . . . 23
5.3 Modelo de prototipagem . . . 24
5.4 Modelo RAD . . . 25
5.5 Modelos evolucionários . . . 26
5.5.1 Modelo incremental . . . 26
5.5.2 Modelo espiral . . . 27
5.5.3 Modelo concorrente . . . 28
5.6 Desenvolvimento baseado em componentes . . . 29
5.7 O modelo de métodos formais . . . 29
5.8 Técnicas de quarta geração . . . 30
5.9 XP — Programação extrema . . . 31

6 Programação Orientada a Objetos e Padrões **33**
6.1 Programação orientada à objetos em C++ . . . 33
6.1.1 O que é um objeto? . . . 33
6.1.2 Encapsulamento e modularidade . . . 34
6.1.3 Sobrecarga . . . 34
6.1.4 Herança e polimorfismo . . . 35
6.1.5 Tratamento de exceções . . . 37
6.1.6 *STL — Standard template library* . . . 38
6.2 Padrões . . . 39
6.2.1 Padrão *MVC — Model/View/Controller* . . . 39
6.2.2 Padrão observador . . . 40

7 Introdução ao CORBA **42**
7.1 Visão geral . . . 42
7.2 A IDL — *Interface Definition Language* . . . 43
7.3 Um exemplo simples de aplicação . . . 44
7.4 Serviços extras do CORBA . . . 46
7.4.1 *Naming service* . . . 46
7.4.2 *Trading service* . . . 46
7.4.3 *Event service* . . . 47
7.4.4 *Security service* . . . 47
7.4.5 *Transaction service* . . . 47

Parte I

Matemática e Modelagem

Capítulo 1

Sistemas Dinâmicos

Neste capítulo serão apresentados uma breve introdução histórica sobre a origem da teoria dos sistemas dinâmicos e a motivação para seu estudo; suas classificações e sua definição formal com algumas propriedades gerais. Por fim, argumenta-se sobre a importância das simulações para esta vasta área de pesquisa.

1.1 A astronomia de Aristóteles a Galileu

Aristóteles (384-322 a.C.) foi um dos principais filósofos clássicos e um dos responsáveis pela estruturação da cultura ocidental. Sua principal contribuição[8] foi a formulação de uma base para o pensamento lógico: uma coleção de regras metodológicas que devem ser empregadas na construção de um discurso coerente. Sua filosofia é dedutiva e apoiada em “verdades evidentes”, indo do particular para o geral, de hipóteses e axiomas para conclusões.

Entretanto, suas proposições sobre as leis que governam os movimentos dos corpos e seu modelo astronômico (geocêntrico) estão longe do atual modelo aceito, mas perduraram até o século XVI. Nessa ocasião, a física sofreu uma revolução. Surgiram vários de seus maiores nomes como T. Brahe, J. Kepler, N. Copérnico, E. Torricelli, G. Galilei e I. Newton.

Brahe dedicou sua vida a acumular dados observacionais do firmamento com precisão ainda não existente até então. Seus dados ajudaram a consolidar o modelo heliocêntrico de Copérnico. Partindo dessa teoria, e juntamente com os dados de Brahe, Kepler formulou suas leis das órbitas planetárias.

Galileu, contemporâneo de Kepler, foi o pioneiro no uso do telescópio para estudos sobre os corpos celestes. Entre 1609 e 1610, observou irregularidades na superfície lunar, as fases de Vênus, a rotação das manchas solares, quatro satélites de Júpiter e uma infinidade de estrelas invisíveis a olho nú. Mas já a alguns anos antes, Galileu começou a se interessar pela teoria dos movimentos. Desconfiado das conclusões aristotélicas, foi o pioneiro a adotar métodos experimentais para comprovar a validade das proposições teóricas. Fez análises quantitativas de experiências realizadas com pêndulos simples, projéteis lançados obliquamente e corpos deslizando sobre planos inclinados ou caindo livremente. Pressupôs os efeitos da resistência do ar e solicitou ajuda a Torricelli, que nas suas tentativas de criar vácuo, acabou inventando o barômetro de mercúrio. Seus estudos refutaram as leis da física de Aristóteles e constituem o ponto de partida da Teoria de Sistemas Dinâmicos.

1.2 O que são sistemas dinâmicos e para que servem

Um sistema pode ser definido como um conjunto de objetos agrupados por alguma interação ou interdependência, de modo que existam relações de causa e efeito nos fenômenos que ocorrem com os elementos desse conjunto. São exemplos de sistemas: um circuito elétrico, Júpiter e seus satélites, a situação financeira de um país, o ecossistema de um lago, o funcionamento de um organismo vivo como uma bactéria ou um mamífero. Um sistema é dinâmico quando algumas grandezas que caracterizam seus objetos constituintes variam com o tempo. Nos exemplos anteriores, algumas das variáveis poderiam ser a corrente elétrica através de um certo componente eletrônico, a posição de Ganimedes (uma das luas de Júpiter), o valor do PIB do país, o número de peixes no lago, a concentração de determinada substância no citoplasma da bactéria ou a concentração hormonal no sangue do animal.

Determinar essa evolução temporal é muito importante, particularmente nas situações a seguir:

- O sistema não existe fisicamente. Por exemplo, na concepção de uma missão espacial, é importante prever tudo o que for possível para garantir ao máximo o sucesso do projeto.

- Tenta-se explicar o funcionamento de sistemas já existentes. Esse é o objetivo da Física, que pretende elucidar as leis que governam o funcionamento do Universo. É também o objetivo da Biologia, nas questões relativas aos seres vivos. Uma simples célula constitui um sistema dinâmico extremamente complexo, que ainda hoje não foi elucidado completamente, mas de fundamental importância para vida.

- O teste experimental é muito caro ou perigoso. Num programa de vacinação no combate a alguma doença como o sarampo, existe sempre um estudo sobre a eficiência da estratégia.

O objetivo desses estudos é prever o futuro de um modo científico[8].

1.3 Classificações

Além de definir a nomenclatura comumente usada na Teoria de Sistemas Dinâmicos, essa classificação serve para distinguir diferentes tipos de situações e, conseqüentemente, diferentes tipos de modelos que podem ser construídos e usados para se estudar a evolução temporal de um sistema.

1.3.1 Quanto ao determinismo

Quando um sistema dinâmico é dito determinístico, a partir de uma determinada condição inicial é possível conhecer-se com certeza todo o seu futuro. Não existe nenhuma variável aleatória ou qualquer tipo incerteza. Porém existe uma subcategoria desses sistemas determinísticos que são os chamados caóticos.

Os sistemas dinâmicos caóticos são determinísticos[9, 10], mas a característica que os tornam particulares é o fato de dependerem fortemente das variações das condições iniciais. Em outras palavras, num sistema não caótico, quando as condições iniciais variam ligeiramente, o comportamento do sistema varia igualmente de forma sutil. Entretanto, nos sistemas caóticos, ínfimas variações nas condições iniciais representam evoluções completamente distintas. Pode haver também a possibilidade de que condições iniciais muito diferentes resultem em comportamentos semelhantes. Podemos citar como exemplo, o sistema composto por duas rolhas soltas juntas na beira de um rio. As condições inicias(velocidade e posição) para dada uma das rolhas são extremamente parecidas, mas com o passar do tempo, uma dela pode ter ficado atracada num rochedo a poucos metros dali, enquanto que a outra acabou no oceano a centenas de quilômetros de distância. Sistemas deste tipo são difíceis de se estudar experimentalmente pois a precisão dos instrumentos de medição é limitada e conseqüentemente as condições iniciais não podem ser determinadas com exatidão. Esses erros inerentes acabam por comprometer a correlação do modelo com os resultados experimentais.

A outra categoria dos sistemas dinâmicos é a estocástica. Nesse caso, uma ou mais de suas variáveis possuem algum tipo de comportamento aleatório. Um exemplo comum, é a modelagem de linhas de transmissão de dados com fonte de ruído externo. O comportamento da linha é determinístico, mas o ruído não. Essa categoria modela não só problemas como esse, mas aqueles que vão desde simples jogos de azar até processos de difusão e reações químicas.

1.3.2 Quanto à variável temporal

Normalmente quando se estuda um processo da natureza, constrói-se um modelo baseado em tempo contínuo. Nesse caso, a variável temporal t é um número real. A descrição do comportamento do sistema dinâmico é dado normalmente por equações diferenciais.

Por outro lado, existem alguns casos em que não faz sentido ou não é conveniente utilizar-se de uma modelagem em tempo contínuo, mas sim em tempo discreto. Nesse caso, a variável t é um número inteiro e a evolução do sistema é dada por equações de diferenças finitas ou qualquer outra função iteradora, ou seja, definida em termos de estados do sistema no instante de tempo discreto anterior. Podemos exemplificar o modelo populacional de um determinado ecossistema, onde o número de indivíduos é conhecido a cada intervalo de uma semana. Nesse exemplo, além do tempo, o espaço(número de indivíduos) são discretos.

Resolução numérica de equações diferenciais

Quando se simula um sistema dinâmico contínuo num computador, de alguma forma o tempo será necessariamente discretizado[7, 10]. Entretanto há uma diferença fundamental em se tratar um caso ou outro. No caso de tempo discreto, as funções são iteradores e estão exatamente definidos em termos de valores anteriores. Entretanto, no caso do tempo contínuo, o fato do mesmo ter que ser discretizado, acarreta em erros que podem comprometer a solução. Dessa forma, quando se soluciona numericamente equações diferenciais é necessário um estudo analítico sobre o método afim de se verificar sob quais condições a solução numérica se aproxima da real. Nenhum método numérico funciona bem para todos os casos, e por esse motivo, existem algumas dezenas deles, cada um para certas classes de problemas.

1.3.3 Quanto ao espaço ser contínuo ou discreto

Os valores que as variáveis do sistema dinâmico podem assumir podem ser reais, no caso do espaço contínuo, ou valores discretos, no caso do espaço discreto. Por exemplo, a temperatura de uma caldeira está contida num espaço contínuo. O número de indivíduos de uma população, por outro lado, está contido num espaço discreto. A saída de um circuito eletrônico conversor analógico-digital também é discreta, embora possa representar um valor não inteiro.

1.3.4 Quanto ao sistema ser linear ou não-linear

A forma mais geral de se escrever uma equação diferencial linear[8] para a variável dependente *x(t)* é:

$$a_n(t)\frac{d^n x(t)}{dt^n} + a_{n-1}(t)\frac{d^{n-1}x(t)}{dt^{n-1}} + \cdots + a_1(t)\frac{dx(t)}{dt} + a_0(t)x(t) = F(t)$$

sendo $t \in \mathbb{R}_+$. Uma equação diferencial é linear se $x(t), dx(t)/dt, ..., d^n x(t)/dt^n$ aparecem como termos de primeiro grau.

Os *coeficientes* $a_i(t)$ e a *entrada* ou *forçamento externo F(t)* são funções apenas da variável independente. Sistemas lineares são aqueles que podem ser escritos da forma acima. Os não-lineares são os que não podem.

Para os sistemas lineares, valem o *princípio da aditividade* e o *princípio da proporcionalidade entre excitação e resposta*. O primeiro princípio estabelece que se para uma entrada $F_1(t)$ o sistema exibe uma resposta $x_1(t)$, e para uma entrada $F_2(t)$ o sistema exibe uma resposta $x_2(t)$, então para uma entrada $F_1(t) + F_2(t)$, a saída do sistema será $x_1(t) + x_2(t)$. O segundo princípio, também conhecido como *princípio da homogeneidade*, afirma que, se para uma entrada *F(t)*, o sistema tem uma saída *x(t)*, então para uma entrada *kF(t)*, sendo k uma constante, a saída será *kx(t)*. Esses dois princípios podem ser combinados em um só, chamado de *princípio da superposição de efeitos*: à entrada $k_1F_1(t) + k_2F_2(t)$, corresponde a saída $k_1x_1(t) + k_2x_2(t)$, sendo k_1 e k_2 constantes.

Em sistemas não-lineares não valem, em geral, esses princípios.

1.3.5 Quanto a parâmetros fixos ou variáveis

Num sistema a parâmetros fixos, os coeficientes $a_i(t)$ são constantes. Assim, o tempo só pode aparecer explicitamente na função de entrada $F(t)$. Num sistema a parâmetros variáveis, um ou mais coeficientes $a_i(t)$ são funções explícitas do tempo.

1.3.6 Quanto a parâmetros concentrados ou distribuídos

Em sistemas a parâmetros concentrados, a variável dependente é função apenas de uma variável independente, em geral o tempo. No caso de sistemas a parâmetros distribuídos, a variável dependente é função do espaço e do tempo. Para ficar mais clara a definição, vamos tomar como exemplo, a equação que descreve a variação da tensão v num capacitor de capacitância C, ligado a um indutor de indutância L:

$$LC\frac{d^2v(t)}{dt^2} + v(t) = 0$$

Note que nesse caso, a tensão v depende apenas do tempo, e portanto o sistema possui parâmetros concentrados. Esse modelo é útil quando se estuda circuitos operando a baixas freqüências, pois o tamanho dos componentes é muito menor que o comprimento de onda. Entretanto, para operações em freqüências mais elevadas, como as utilizadas nas transmissões de rádio, é necessário levar em conta que as ondas eletromagnéticas se propagam com velocidade finita. Nesse caso, temos que utilizar um sistema a parâmetros distribuídos, como mostra a equação abaixo:

$$LC\frac{\partial^2 v(x,t)}{\partial t^2} - \frac{\partial^2 v(x,t)}{\partial x^2} = 0$$

1.3.7 Quanto ao sistema ser dinâmico ou instantâneo

Num sistema instantâneo (sem memória), a resposta num dado instante depende apenas da excitação nesse mesmo instante. Por exemplo, considere um resistor de resistência R, que obedece à lei de Ohm $v(t) = RI(t)$. Suponha que esse resistor esteja ligado a uma fonte que gera uma corrente $I(t)$. Esse sistema é instantâneo, pois o valor da tensão v no instante t, depende apenas do valor da corrente I, no mesmo instante t.

Num sistema dinâmico (com memória), a resposta num dado instante depende dos valores das entradas passadas. Por exemplo, considere um capacitor de capacitância C. A tensão v entre suas placas vale no instante t:

$$v(t) = \frac{1}{C}\int_{-\infty}^{t} I(t)dt$$

sendo $I(t)$ a corrente que chega ao capacitor no instante t. Em princípio, para se calcular v(t), parece ser necessário conhecer a corrente que fluiu para o capacitor em todo o seu passado, isto é, de -∞ até t. Entretanto, essa expressão pode ser reescrita como:

$$v(t) = \frac{1}{C}\int_{-\infty}^{t_0} I(t)dt + \frac{1}{C}\int_{t_0}^{t} I(t)dt = v(t_0) + \frac{1}{C}\int_{t_0}^{t} I(t)dt$$

Assim, conhecendo-se o valor de v em algum instante t_0, e como I varia entre t_0 e t, pode-se determinar $v(t)$. Observe que $v(t_0)$ contém toda a informação sobre o passado do capacitor que é necessária para calcular $v(t)$, para $t \geq t_0$. Esse sistema é dinâmico, pois $v(t)$ depende de $v(t_0)$, que por sua vez é determinado a partir da entrada $I(t)$ entre -∞ e t_0.

1.4 Definição formal e propriedades gerais

Os sistemas dinâmicos podem ser definidos em termos de tempo contínuo ou discreto. Para não nos atermos a esses detalhes, chamaremos o conjunto domínio do tempo simplesmente de $\mathcal{T}$. Formalmente[5], o modelo de um sistema físico é um sistema dinâmico se e somente se pode ser associado a ele cinco objetos ($\mathcal{U}$, Σ, $\mathcal{Y}$, s, r)

- $\mathcal{U}$ é um conjunto de funções. Seus elementos mapeiam $\mathcal{T} \to \mathbb{U}$. Tipicamente, $\mathbb{U}$ é $\mathbb{R}$ ou $\mathbb{R}^n$. Cada elemento u deste conjunto é chamado de uma entrada do sistema. Para todo $t \in \mathcal{T}, u(t)$ é chamado de valor da entrada u no instante t.

- Σ é um conjunto de variáveis. Cada um de seus elementos x é chamado de estado. Tipicamente, Σ é $\mathbb{R}^k$. Para todo $t \in \mathcal{T}, x(t)$ é chamado de estado no instante t.

- $\mathcal{Y}$ é um conjunto de funções. Seus elementos mapeiam $\mathcal{T} \to \mathbb{Y}$. Tipicamente, $\mathbb{U}$ é $\mathbb{R}$ ou $\mathbb{R}^m$. Cada elemento y deste conjunto é chamado de saída do sistema. Para todo $t \in \mathcal{T}, y(t)$ é chamado de valor da saída y no instante t.

- s é a chamada função de transição de estado. Está definida para todo $t_1, t_0 \in \mathcal{T}$ com $t_1 \geq t_0$, para todo $x_0 \in \Sigma$ e para todo $u \in \mathcal{U}$. O estado no instante t_1 atingido a partir do estado x_0 no instante t_0 como resultado da entrada u é definido como
$$x(t_1) := s(t_1, t_0, x_0, u)$$

- r é uma função que mapeia $\mathcal{T} \times \Sigma \times \mathbb{U} \to \mathbb{Y}$. A saída do sistema no instante t quando no estado $x(t)$ no instante t e quando submetido à entrada $u(t)$ é definida como:
$$y(t) := s(t, x(t), u(t))$$

e cuja função s satisfaz os seguintes axiomas:

- Axioma da transição de estado: Para todo $t_1, t_0 \in \mathcal{T}$ com $t_1 \geq t_0$, para todo $x_0 \in \Sigma$, se $u, \tilde{u} \in \mathcal{U}$ e se $u(t) = \tilde{u}(t)$ para todo $t \in [t_0, t_1] \cap \mathcal{T}$ então $s(t_1, t_0, x_0, u) = s(t_1, t_0, x_0, \tilde{u})$

- Axioma do semi-grupo: Para todo $t_0, t_1, t_2 \in \mathcal{T}$ com $t_2 \geq t_1 \geq t_0$, para todo $x_0 \in \Sigma$, e para todo $u \in \mathcal{U}$,

$$s(t_2, t_1, s(t_1, t_0, x_0, u) = s(t_2, t_0, x_0, u)$$

Em resumo, dizemos que um modelo é um sistema dinâmico se e somente se podemos associá-lo a um espaço de entrada, a um espaço de estado, a um espaço de saída e a duas funções s e r, onde para todo $t_1, t_0 \in \mathcal{T}$ com $t_1 \geq t_0$, para todo $x_0 \in \Sigma$, e para todo $u \in \mathcal{U}$,

$$x(t_1) := s(t_1, t_0, x_0, u_{[t_0,t_1]})$$

$$y(t_1) := s(t_1, x(t_1), u(t_1))$$

e cuja função s satisfaz os axiomas de transição de estado e do semi-grupo.

Do axioma de transição de estado concluímos que conhecendo-se o estado inicial do sistema no instante t_0, pode-se determinar qualquer estado futuro num instante t_1 onde $t_1 \geq t_0$. Em outras palavras, o estado x_0 resume todo o passado do sistema. Também pode-se concluir que o sistema é não-antecipativo ou causal.

Do axioma do semi-grupo, concluímos que para se conhecer o estado do sistema num instante t_1 a partir das condições iniciais em t_0, devemos conhecer a evolução do sistema em todo $t \in \mathcal{T}$ com $t_0 < t < t_1$.

1.5 A importância das simulações

As simulações computacionais de sistemas dinâmicos são importantes por três razões principais:

- Soluções analíticas só podem ser obtidas para casos bastante particulares de sistemas dinâmicos. A única forma de se estudar a grande maioria dos problemas é através da simulação. O problema dos três corpos[1] em física por exemplo, só pode ser solucionado numericamente.

- Muitos sistemas possuem comportamentos estranhos, ou de difícil compreensão. A visualização gráfica da evolução do sistema dinâmico tem ajudado à entender como esses sistemas funcionam.

- Ferramentas computacionais são importantes na exploração de sistemas complexos, ajudando a selecionar informações relevantes num conjunto muito grande de dados, a encontrar correlações, etc.

[1] É um problema onde se pretende determinar a posição e velocidade de três corpos onde entre eles exista uma força de atração e/ou repulsão mútua. O Sistema Solar é um problema desse tipo, mas ao invés de três, possui dez corpos.

Capítulo 2

Máquinas Sequenciais e Autômatos

Este breve capítulo apenas apresenta algumas definições acerca das máquinas sequenciais e dos autômatos.

2.1 Visão informal

Suponha um aparelho de vídeo-cassete simples. A cada instante, ele pode estar em apenas um de três estados possíveis: desligado, ligado ou reproduzindo. O aparelho também possui em seu painel apenas três botões: *power*, *play* e *stop*. Por fim, a saída do vídeo-cassete, conectada à uma TV, pode estar em dois estados possíveis: emitindo sinal de vídeo ou não.

Este é um exemplo concreto de uma máquina seqüencial simples. Pela descrição anterior, fica clara a presença de três elementos:

- Entrada: São os três botões do painel: *power*, *play* e *stop*.
- Saída: É a saída do sinal de vídeo do aparelho.
- Estado: É o estado em que o aparelho se encontra: desligado, ligado ou reproduzindo.

Mas apenas esses três elementos não definem como o aparelho funciona. Para isso, precisamos definir mais dois elementos:

- Função de transição: A partir de um determinado estado e uma determinada entrada, podemos dizer qual será o próximo estado do aparelho. Por exemplo, se estiver no estado desligado, e for pressionada a tecla play, o aparelho continuará no estado desligado. Por outro lado, neste mesmo estado, se for pressionada a tecla power, o vídeo-cassete será ligado. Com um conjunto de regras desse tipo, pode-se definir como funciona o aparelho.
- Função de saídaMas só podemos descobrir se o vídeo-cassete está funcionando ou não, reproduzindo ou não, se o conectarmos à uma TV. Esse é um vídeo-cassete simples, sem qualquer painel indicador de funcionamento!

O grafo abaixo resume o funcionamento desse vídeo-cassete:

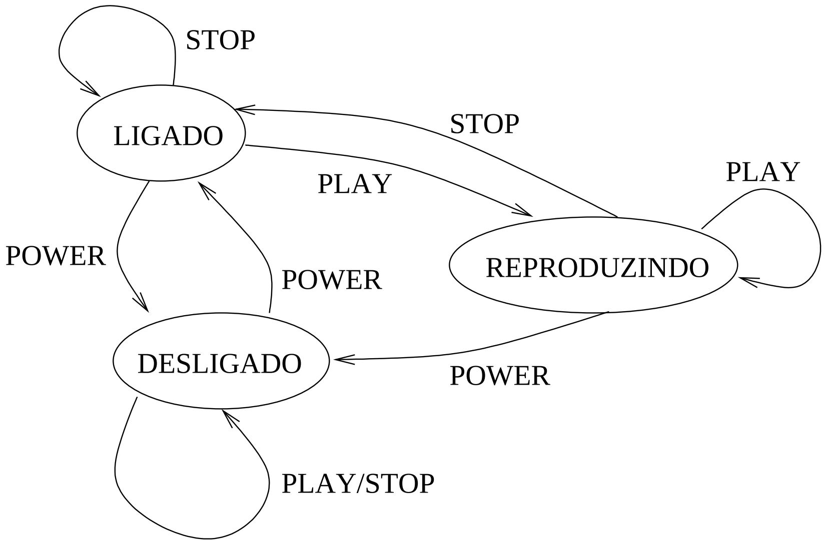

Figura 1 - Uma máquina seqüencial simples

Note que não existe nada especificado na figura acerca da saída do vídeo-cassete. Mas está implícito que apenas no estado reproduzindo é que será emitido um sinal de vídeo que pode ser captado pela TV.

2.2 Máquinas sequenciais

2.2.1 Definição: Máquina seqüencial completa

Uma máquina seqüencial completa[6] é uma tupla $M = (S, \Sigma, \Delta, \delta, \lambda)$, onde

(i) S é um conjunto finito não vazio, chamado conjunto de estados;

(ii) Σ é um conjunto finito não vazio, chamado conjunto de entrada ou alfabeto de entrada;

(iii) Δ é um conjunto finito não vazio, chamado conjunto de saída ou alfabeto de saída;

(iv) δ é uma função, chamada de função de transição ou de função de próximo-estado. Ela mapeia $S \times \Sigma \to S$;

(v) λ é uma função, chamada de função de saída. Ela mapeia $S \times \Sigma \to \Delta$.

2.2.2 Definição: Máquina seqüencial não-necessariamente-completa

Uma máquina seqüencial não-necessariamente-completa[6] é uma tupla $M = (S, \Sigma, \Delta, \delta, \lambda)$, onde

(i) S é um conjunto finito não vazio, chamado conjunto de estados;

(ii) Σ é um conjunto finito não vazio, chamado conjunto de entrada ou alfabeto de entrada;

(iii) Δ é um conjunto finito não vazio, chamado conjunto de saída ou alfabeto de saída;

(iv) δ é uma função, chamada de função de transição ou de função de próximo-estado. Ela mapeia um subconjunto de $S \times \Sigma$ em S;

(v) λ é uma função, chamada de função de saída. Ela mapeia um subconjunto de $S \times \Sigma$ em Δ.

M é dita uma máquina seqüencial incompleta se e somente se o domínio de δ ou de λ ou de ambas é um subconjunto próprio de $S \times \Sigma$.

2.2.3 Definição: Máquina seqüencial infinita

Uma máquina seqüencial é dita infinita quando o conjunto de estados S é infinito. Os conjuntos de entrada Σ e de saída Δ podem ser infinitos, mas não necessariamente.

2.3 Autômatos

2.3.1 Definição: Autômato

Um autômato[6] é uma tripla $A = (S, \Sigma, \delta)$, onde

(i) S é um conjunto, chamado conjunto de estados;

(ii) Σ é um conjunto não vazio, chamado conjunto de entrada ou alfabeto de entrada;

(iv) δ é uma função, chamada de função de transição ou de função de próximo-estado. Ela mapeia $S \times \Sigma \to S$;

A é um autômato infinito se e somente se o conjunto de estados S é infinito. A é um autômato finito se e somente se S é finito.

Note que a definição de autômato é aproximadamente a definição de máquina seqüencial sem a estrutura de saída composta por Δ e λ.

Capítulo 3

Processos Estocásticos

Neste capítulo é dada uma introdução sobre processos estocásticos.

3.1 Introdução

Um processo estocástico é um processo aleatório evoluindo no tempo[12, 11]. Mais precisamente, um processo estocástico é uma coleção de variáveis aleatórias X_t indexadas pelo tempo. Esse tempo pode ser contínuo ou discreto. As variáveis X podem assumir valores de um conjunto chamado de espaço de estados. Esse conjunto também pode ser contínuo ou discreto.

Uma classe importante dos processos estocásticos também possui a propriedade de que a mudança de estado no instante t é determinada apenas pelo estado do processo neste mesmo instante t, e não por estados anteriores a este. Esses processos são os chamados processos de Markov.

3.2 Cadeias de Markov finitas

Considere um processo estocástico em tempo discreto, $X_n, n = 0, 1, 2, ...$, onde X_n recebe valores do conjunto finito $S = \{0, 1, ..., N-1\}$ ou $\{1, ..., N\}$. Chamaremos os possíveis valores de X_n de estados do sistema. Para descrever as probabilidades para este processo, precisamos dos valores de

$$P\{X_0 = i_0, X_1 = i_1, ..., X_n = i_n\}$$

para todo n e toda seqüencia finita de estados $(i_0, i_1, ..., i_n)$. Equivalentemente, precisamos da distribuição de probabilidades inicial

$$\phi(i) = P\{X_0 = i\} \qquad i = 1, ..., N$$

e das probabilidades de transição

$$q_n(i_n|i_0, ..., i_{n-1}) = P\{X_n = i_n|X_0 = i_0, ..., X_{n-1} = i_{n-1}\}$$

para então

$$P\{X_0 = i_0, ..., X_n = i_n\} = \phi(i_0)q_1(i_1|i_0)q_2(i_2|i_0, i_1)\cdots q_n(i_n|i_0, ..., i_{n-1})$$

Considerando-se que se trata de um processo de Markov, podemos fazer as predições futuras olhando-se apenas para o estado atual do sistema, sem haver a necessidade de conhecer seu passado. O estado atual do sistema é necessário para se determinar o próximo estado, mas como esse estado atual foi alcançado é irrelevante. Matematicamente, podemos escrever isso assim

$$P\{X_n = i_n|X_0 = i_0, ..., X_{n-1} = i_{n-1}\} = P\{X_n = i_n|X_{n-1} = i_{n-1}\}$$

Podemos fazer mais uma consideração: A probabilidade de transição de estado se mantém ao longo do processo. Essa é a propriedade de homogeneidade temporal do sistema. Uma cadeia de Markov homogênea no tempo é um processo que

$$P\{X_n = i_n|X_0 = i_0, ..., X_{n-1} = i_{n-1}\} = p(i_{n-1}, i_n),$$

para alguma função $p : S \times S \to [0, 1]$.

Para obtermos as probabilidades da cadeia de Markov, precisamos conhecer a distribuição de probabilidades inicial $\phi(i) = P\{X_0 = i\}$, e as probabilidades de transição $p(i, j)$ para então

$$P\{X_0 = i_0, ..., X_n = i_n\} = \phi(i_0)p(i_0, i_1)p(i_1, i_2)\cdots p(i_{n-1}, i_n)$$

A matriz de transição P para a cadeia de Markov é uma matriz $N \times N$ cujo elemento (i, j) é $P_{ij} = p(i, j)$. A matriz P é estocástica, isto é,

$$0 \leq P_{ij} \leq 1, \qquad 1 \leq i, j \leq N$$

$$\sum_{j=1}^{N} P_{ij} = 1, \qquad 1 \leq i \leq N.$$

Qualquer matriz que satisfaça as duas condições anteriores pode ser uma matriz de transição para uma cadeia de Markov.

3.3 Um exemplo de processo estocástico

Suponhamos o seguinte modelo, chamado de jogo da espera. O processo inicia-se no estado $X_0 = A$. Com probabilidade q continua neste mesmo estado, e com probabilidade $p = 1 - q$ passa para o estado B. A partir deste estado, permanece assim indefinidamente, isto é, a probabilidade de passar do estado B para o estado B é 1. A figura abaixo representa esse processo.

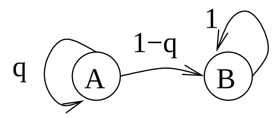

Figura 2 - Um processo estocástico simples

A matriz P de transição para este processo pode ser escrita como

$$P = \begin{bmatrix} q & 1-q \\ 0 & 1 \end{bmatrix} \qquad \text{e por indução finita} \qquad P^n = \begin{bmatrix} q^n & 1-q^n \\ 0 & 1 \end{bmatrix}$$

P^n representa a probabilidade de transição considerando-se n iterações. Observa-se no exemplo anterior, que com o passar do tempo, a probabilidade da transição A$\rightarrow$B tende a 1, enquanto que a probabilidade da transição A$\rightarrow$A tende a zero. Isso quer dizer que o sistema tende ao estado estacionário B. Através de estudos semelhantes a esse, pode-se determinar se um processo estocástico tende a um determinado estado estacionário, se ocorrerão ciclos ou se continuará de forma indefinida.

Capítulo 4

Construção de um Modelo para Redes de Expressão Gênica

Neste capítulo serão apresentados uma teoria sucinta sobre o sistema bioquímico envolvido no controle da expressão gênica. Em seguida, será apresentada uma modelagem desse sistema em termos de sistemas dinâmicos estocásticos. Por fim, será apresentada a representação desse modelo em termos computacionais.

4.1 O problema bioquímico da expressão gênica

Uma célula é o ser vivo mais simples que existe, mas um dos sistemas mais complexos com que o Homem já se deparou. Ela possui vários mecanismos que vão desde a síntese de uma proteína até o processo de divisão celular, diferenciação[1] ou apoptose[2]. Naturalmente, esses sistemas possuem mecanismos de controle e auto-regulação[1, 2, 3, 4], provavelmente todos integrados entre si.

Um grande desafio para a ciência é compreender cada vez mais todos esses mecanismos. Através desse conhecimento, pode-se desenvolver de modo muito mais eficaz técnicas para tratamentos de certas doenças, como o câncer, ou até mesmo encontrar a cura.

Mas muita coisa já se sabe. Todo o funcionamento da célula é regido pelo código genético contido na molécula de DNA. Um gene, é uma pequena fração deste DNA que codifica uma proteína. Em outras palavras, com o auxílio da maquinaria de síntese proteica da célula, fabrica-se a proteína codificada nesse gene. O genoma de uma célula é codificado numa seqüência de quatro diferentes tipos de bases nitrogenadas dispostas na molécula de DNA: adenina, timina, guanina e citosina. Cada trio dessas bases forma o que se chama de códon. Há portanto $4^3 = 64$ códons diferentes.

As proteínas são polímeros[3] de aminoácidos. Existem vinte diferentes tipos de aminoácidos. O que diferencia uma da outra é a seqüência dos aminoáciodos. Embo-

[1] É o processo pelo qual uma célula tronco se diferencia, tornando-se por exemplo, uma célula nervosa ou epitelial.

[2] Processo de morte celular programada.

[3] Uma molécula longa composta de diferentes sub-unidades, no caso os aminoácidos.

ra as proteínas possuam em geral de dezenas a centenas de aminoácidos, a mudança de apenas um deles pode significar uma nova proteína com função bioquímica completamente diferente.

A tradução do código genético, contido na forma de uma seqüência de códons no DNA em proteína é feita pelo sistema de síntese proteica da célula. Cada códon representa um único aminoácido. Nota-se que o código genético é degenerado, pois tem-se 64 códons diferentes para representar apenas 20 aminoácidos. São as proteínas as verdadeiras protagonistas das funcionalidades celulares. Quando a proteína correspondente a um certo gene é sintetizada, dizemos que este gene está sendo expresso.

É evidente que cada uma dessas proteínas é sintetizada de forma controlada. Por exemplo, durante o processo de divisão celular, certos tipos de proteínas são responsáveis pela formação dos fusos[4]. Sintetizá-las em outra ocasião poderia ser um desastre. Em outras palavras, a expressão dos genes é um sistema auto-regulado. Essa regulação pode ocorrer em diversos níveis, de forma direta ou indireta. Do DNA até uma proteína, existem várias etapas, como a síntese de RNA(transcrição) e a síntese proteica(tradução). A regulação da expressão pode ocorrer em qualquer uma dessas etapas, de modo a estimular ou inibir o processo. Essa regulação pode ser causada pelos sub-produtos diretos ou indiretos desse processo ou de qualquer outro processo ocorrendo dentro da célula além dos estímulos externos. É realmente um sistema de controle bastante complexo. Compreendê-lo completamente é ainda um desafio.

4.2 Modelagem do problema

O sistema bioquímico de regulação de expressão gênica é um sistema dinâmico. Podemos associá-lo aos cinco objetos que definem um sistema dinâmico:

- Espaço de entrada: Todas as variáveis externas à célula ou algum processo particular que se estuda. Por exemplo, a concentração de determinada substancia no ambiente em que a célula reside. São unicamente dependentes do tempo.

- Espaço de saída: Todas as variáveis que não influem no processo em questão. Não possuem muita importância em nosso modelo, já que o objeto de estudo são os estados internos do sistema.

- Espaço de estados: Representa a concentração de todas as substâncias que se considera relevantes ao processo em estudo.

- Função de transição: O sistema de auto-regulação pode ser definido em termos da concentração de cada uma das substâncias envolvidas no processo, bem como dos sinais provenientes do exterior da célula[5].

[4]São estruturas em forma de fibra proteica que dão sustentação mecânica ao processo de divisão celular.

[5]Essa é uma teoria bem conhecida e aceita na bioquímica.

- Função de saída: Determina o valor das saídas do sistema. Não tem grande importância em nosso modelo.

A função de transição de estado satisfaz os axiomas de transição de estado e do semi-grupo. Intuitivamente, podemos concluir isso porque o processo é causal, isto é, não-antecipativo e porque o futuro do processo é determinado pelas condições atuais, não importando o passado. Além disso, para se determinar a configuração num certo estado futuro, deve-se antes determinar o estado de todos os instantes intermediários.

Essas argumentações são intuitivas. O formalismo matemático não foi colocado pois seria bastante extenso. Entretanto, justificam a proposta do modelo.

4.2.1 O modelo estocástico

Como os processos bioquímicos estão estritamente relacionados às reações químicas, e estes são processos aleatórios[6], há um certo grau de incerteza na sua predição. Dessa forma, é conveniente acrescentar ao modelo essa característica.

Para não fugir do modelo de sistema dinâmico determinístico, podemos acrescentar uma função de entrada com comportamento aleatório. Cada uma das funções de influência, isto é, cada componente da função de transição de estado, terá uma dependência com relação a esta entrada aleatória, de modo que seu resultado possa ser influenciado pela mesma.

4.3 A representação em estruturas de dados

Podemos representar um sistema dinâmico através de um grafo, onde cada vértice corresponde a uma variável do sistema, seja ela de entrada, saída ou estado interno. Cada uma delas é função de um conjunto de outras. Isso pode ser representado através de arcos, com ponta inicial na variável dependente e ponta final na variável influente. A variável independente t também é representada por um vértice, mas nesse caso ele é especial e possui alguns tratamentos diferenciados. Cada vértice é associado à uma estrutura de dados que representa a função que determina o estado da variável associada à esse vértice. Os argumentos dessa função são os valores das variáveis associadas aos vértices que estão na ponta final dos arcos que partem do vértice em questão. Para ficar mais clara a definição, tomemos como exemplo o sistema dinâmico abaixo:

- Entradas: $u_1(t) = t^2$ e $u_2(t) = 3t$
- Estados: $x_1(t_1) = u_1(t_1) + x_2(t_0) + t_1$ e $x_2(t_1) = u_2(t_1) + x_1(t_0)$
- Saídas: $s(t) = x_1(t) + x_2(t) + u_2(t)$

[6]Uma reação química, depende de uma série de fatores aleatórios tais como a energia e a orientação das moléculas no momento em que se chocam. A partir dessa teoria, chamada de mecânica estatística, pode-se prever diversos comportamentos macroscópicos como por exemplo a aceleração de uma reação através do aquecimento.

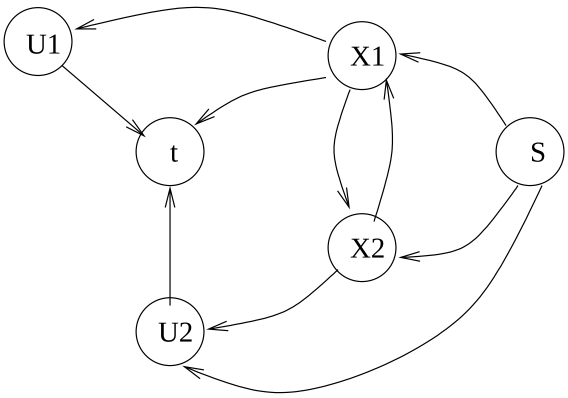

Figura 3 - Um exemplo de representação de sistemas dinâmicos em grafos

Nota-se pela figura que:

- A variável independente t é representada por um vértice especial com grau de saída igual a zero;
- As entradas são representadas por vértices com grau de saída igual a zero, se desconsiderarmos o vértice da variável independente;
- As saídas são representadas por vértices com grau de entrada zero.
- As variáveis de estado do sistema dinâmico são representadas por vértices que não se enquadram em nenhuma das categorias acima.

Além disso, podemos inferir de imediato algumas propriedades:

- Se alguma variável A depende direta ou indiretamente de outra variável B, então existe pelo menos um caminho de A até B. Portanto, o território de um vértice v associado à uma variável a do sistema dinâmico representa o conjunto das variáveis que exercem alguma influência de forma direta ou indireta nessa variável a.
- Desconsiderando-se a variável independente t, cada um dos componentes do grafo representa um sistema dinâmico independente.

Relacionados a essas duas propriedades, já existem algoritmos conhecidos. Dessa forma, pode-se a *priori* fazer uma análise acerca do sistema dinâmico representado, identificando erros na estrutura do mesmo. Isso pode ser interessante se um programa de simulação que utiliza essa estrutura de dados permite que o usuário defina o sistema dinâmico.

Parte II

Engenharia de Software e Programação Orientada a Objetos

Capítulo 5

Modelos de Processo de Software

Neste capítulo são apresentados os principais paradigmas de desenvolvimento em engenharia de software

5.1 Introdução

Os problemas reais na indústria de software são na sua maioria bastante complexos. Além dos problemas relacionados à programação, existem muitos fatores complicadores. Para citar alguns,

- Os clientes normalmente não sabem exatamente o que estão solicitando. É comum não terem um conhecimento preciso acerca da lógica do próprio negócio. Especificar o que o novo sistema deverá ser capaz de fazer torna-se quase impossível.

- Quando o cliente conhece bem seu problema e o que de fato necessita, há o problema da comunicação disso para o desenvolvedor. Mal entendidos nessa fase são comuns e podem ter conseqüências drásticas como o cancelamento de um contrato no meio do processo.

- O software requer manutenção após sua conclusão; seja para atender a novos requisitos de um cliente, seja para produzir uma nova versão competitiva com o mercado. Os requisitos podem mudar até mesmo durante o período de desenvolvimento. Dependendo de como é desenvolvido o software, pode-se tornar mais fácil, difícil ou impossível lidar com esse problema.

- Desenvolver um software, como toda atividade empresarial, dever ser um negócio lucrativo. Para a empresa ser competitiva, deve aliar alta qualidade e baixo custo. Mas esse paradoxo sempre foi um dos desafios da engenharia.

Na busca de amenizar esses problemas é que surgiu a engenharia de software. É uma área de estudo bastante ampla, que aborda as metodologias de desenvolvimento, arquitetura de sistemas, gerência do processo e análise de risco entre vários outros tópicos igualmente importantes.

Para resolver problemas reais numa empresa, um engenheiro de software deve adotar uma estratégia de desenvolvimento, também conhecida como modelo de processo. Esse modelo representa uma tentativa de trazer ordem para uma atividade inerentemente caótica. É importante lembrar que cada um dos modelos existentes foi caracterizado de forma a ajudar no controle e na coordenação de um projeto de software real. A seguir são discutidos os principais modelos de processo para engenharia de software[14, 15].

5.2 Modelo em cascata

Também chamado de modelo seqüencial linear ou clássico sugere uma abordagem sistemática seqüencial para o desenvolvimento de software, que começa no nível de sistema e progride através da análise, projeto, codificação, teste e manutenção. Apesar do modelo original proposto por W. Royce em 1970 prever ciclos de realimentação entre essas etapas, a grande maioria das organizações que o aplica trata-o como se fosse estritamente linear. A figura abaixo resume esse método.

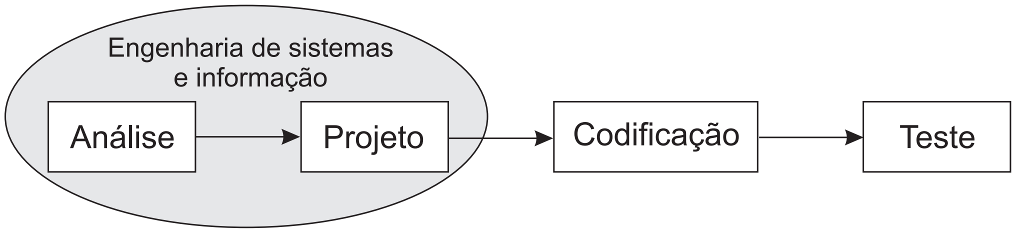

Figura 4 - Modelo cascata ou seqüencial linear

O modelo seqüencial linear é o paradigma mais antigo e é o mais amplamente usado. Entretanto, as críticas sobre esse modelo levaram até mesmo seus fiéis adeptos a questionarem sua eficácia. Entre outros problemas, podemos destacar:

- Projetos reais raramente seguem o fluxo seqüencial que o modelo propõe. Apesar de aceitar iteração, isso é feito de modo indireto neste paradigma. Como resultado, modificações podem causar confusão à medida que o projeto prossegue;

- Em geral, é difícil para o cliente estabelecer todos os requisitos explicitamente. O modelo seqüencial linear exige isso e tem dificuldade de acomodar a incerteza natural que existe no começo de vários projetos;

- A primeira versão executável do programa não fica disponível até o termino do projeto. Um erro grosseiro pode ser desastroso, e isso não é incomum de acontecer na prática.

Embora possua seus pontos fracos, é um modelo de desenvolvimento de software significativamente melhor que uma abordagem aleatória.

5.3 Modelo de prototipagem

O cliente freqüentemente define um conjunto de objetivos gerais para o software, mas não identifica detalhadamente requisitos de entrada, processamento ou saída. Em outros casos, o desenvolvedor pode estar inseguro da eficiência de um algoritmo, da adaptabilidade de um sistema operacional ou da forma que a interação homem/máquina deve assumir. Nessas e em muitas outras situações, um paradigma de prototipagem pode oferecer a melhor abordagem.

Esse paradigma começa com a definição de requisitos. O desenvolvedor e o cliente encontram-se e definem os objetivos gerais do software, identificam as necessidades conhecidas e delineiam áreas que necessitam de mais definições. Um projeto rápido é então realizado, concentrando-se na representação nos aspectos do software que vão ficar visíveis ao cliente/usuário, isto é, na interface. O projeto rápido parte de um protótipo que é então avaliado pelo cliente/usuário e usado para refinar os requisitos do software que será desenvolvido. Interações ocorrem à medida que o protótipo é ajustado para satisfazer às necessidades do cliente, enquanto, ao mesmo tempo, permitem ao desenvolvedor entender melhor o que precisa ser feito. A figura abaixo resume o modelo.

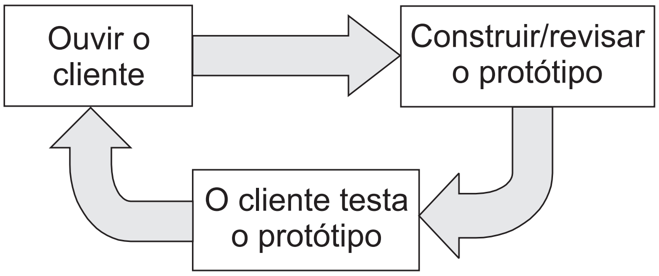

Figura 5 - Modelo de prototipagem

Esse protótipo serve apenas para delinear melhor os requisitos do sistema, incluindo a interface[1] do programa com o cliente/usuário. Só então que o núcleo do sistema com suas funcionalidades é implementado. Essa é a grande vantagem desse modelo para ambos os lados, mas também tem seus inconvenientes:

- O cliente vê o que parece ser uma versão executável do software, ignorando que o protótipo apenas consegue funcionar precariamente, sem saber que sua qualidade global e manutenibilidade são baixíssimas. O cliente geralmente reclama ao saber que o sistema deve ser refeito para que o mesmo atinja a qualidade desejável, já que terá que esperar e pagar por isso;

- Ao desenvolver o protótipo, o objetivo principal é obter a melhor especificação dos requisitos possível através de um *feedback* com o cliente. Não há a preocupação com a qualidade da implementação. Entretanto, como já existe algo feito, mesmo que mal, acaba sendo incorporado ao sistema definitivo, as vezes por razões de tempo ou custo, outras, por descuido dos próprios programadores.

[1]Geralmente uma GUI — *Graphical user interface*

Apesar desses inconvenientes, a prototipagem pode ser um paradigma efetivo para a engenharia de software. O importante é definir as regras do jogo no início; isto é, o cliente e o desenvolvedor devem estar de acordo que o protótipo seja construído apenas para servir como um mecanismo para definição dos requisitos. Depois ele é descartado e o software definitivo é desenvolvido buscando-se a qualidade e manutenibilidade.

5.4 Modelo RAD

O desenvolvimento rápido de aplicação(*rapid application development, RAD*) é um modelo de processo de desenvolvimento de software incremental que enfatiza um ciclo de desenvolvimento extremamente curto. O modelo RAD é uma adaptação de alta velocidade do modelo seqüencial linear, no qual o desenvolvimento rápido é conseguido pelo uso de construção baseada em componentes. Se os requisitos são bem compreendidos e o objetivo do projeto é restrito, o processo RAD permite a uma equipe de desenvolvimento criar um sistema plenamente funcional, dentro de períodos muito curtos. Esse modelo possui cinco fases: modelagem do negócio, modelagem dos dados, modelagem do processo, geração da aplicação, teste e entrega. Se uma aplicação comercial pode ser modularizada de modo a permitir que cada função principal possa ser completada em menos de três meses, é uma forte candidata à RAD. Cada uma dessas funções pode ser tratada por uma equipe RAD distinta e depois integrada para formar um todo. A figura abaixo esquematiza o processo.

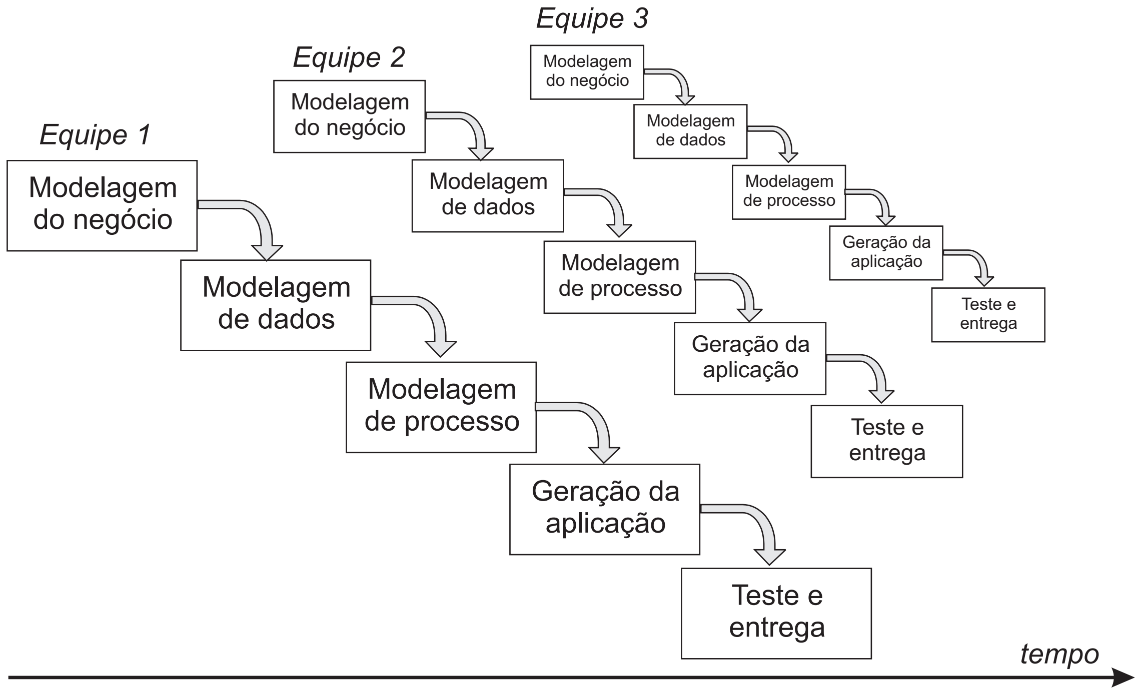

Figura 6 - Modelo RAD

Como todos os modelos de processo, também possui suas desvantagens:

- Para projetos grandes, é necessário um número grande de equipes;

- Tanto os desenvolvedores quanto os clientes estão comprometidos com a rapidez do ciclo de desenvolvimento. Se isso não for seguido à risca, o processo falha, pois o trabalho de uma equipe depende da outra;

- Nem todos os tipos de aplicação podem ser suficientemente modularizados de modo a se tornar apropriada ao RAD. A componentização pode atrapalhar ao invés de ajudar. O processo de integração no final pode ser problemático;

- Quando riscos tecnológicos[2] forem altos, o método não é aconselhável.

5.5 Modelos evolucionários

Torna-se cada vez mais evidente que o software, como todos os sistemas complexos, evolui durante um período de tempo. Requisitos do negócio e do produto mudam freqüentemente à medida que o desenvolvimento prossegue, dificultando um caminho direto para um produto final. Prazos reduzidos de mercado tornam impossível completar um produto de software abrangente, mas uma versão reduzida pode ser elaborada para fazer face à competitividade ou às pressões do negócio; um conjunto de requisitos básicos de um produto ou sistema é bem recebido, desde que os detalhes das extensões do produto ou sistema sejam definidos. Nesse caso, e em situações semelhantes, os engenheiros de software precisam de um modelo de processo que tenha sido explicitamente projetado para acomodar um produto que evolui com o tempo. A natureza evolutiva do software não é considerada em nenhum dos paradigmas clássicos descritos anteriormente. Modelos evolucionários são interativos. Eles são caracterizados de forma a permitir aos engenheiros de software desenvolver versões cada vez mais completas do produto.

5.5.1 Modelo incremental

O modelo incremental combina elementos do modelo seqüencial linear, aplicado repetidamente, com a filosofia interativa da prototipagem. Cada seqüência linear produz um incremento factível do software. Quando um modelo incremental é usado, o primeiro incremento é freqüentemente chamado de núcleo do sistema. Isto é, os requisitos básicos são satisfeitos, mas muitas características suplementares deixam de ser elaboradas. O núcleo do produto é então usado e avaliado pelo cliente. Um plano é desenvolvido para o próximo incremento como resultado desse uso e/ou avaliação. O plano visa à modificação do núcleo do produto para melhor satisfazer às necessidades do cliente e a elaboração de características e funcionalidades adicionais. Esse processo é repetido após a realização de cada incremento, até que o produto completo seja produzido.

[2]Ocorre quando se usa intensamente uma nova tecnologia ou quando o novo software exige um alto grau de interoperabilidade com outros programas já existentes.

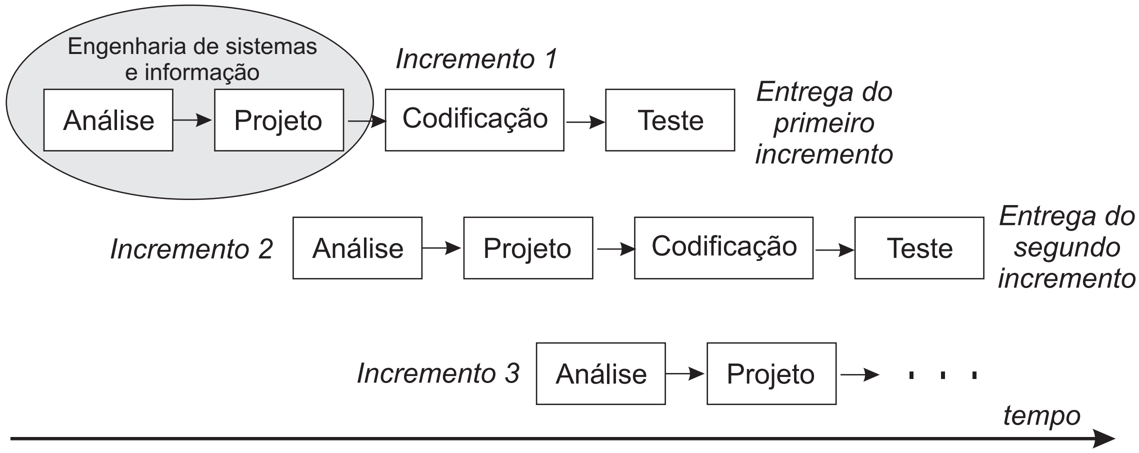

Figura 7 - Modelo incremental

Diferentemente da prototipagem, o modelo incremental objetiva a elaboração de um produto operacional a cada incremento. Os primeiros incrementos são versões simplificadas do produto final, mas oferecem capacidades que servem ao usuário, além de uma plataforma para avaliação pelo mesmo. O desenvolvimento incremental é particularmente útil quando não há mão-de-obra disponível para uma implementação completa, dentro do prazo comercial de entrega estabelecido para o projeto. Os primeiros incrementos podem ser implementados com menos pessoal. Se o núcleo do produto for bem sucedido, então o pessoal extra pode ser adicionado para implementar o próximo incremento.

5.5.2 Modelo espiral

É o modelo mais aceito pelas grandes corporações. É voltado principalmente para grandes projetos, tanto aqueles voltados ao mercado geral quanto os feitos sob encomenda por um cliente em particular. É semelhante ao modelo incremental pois produz uma nova versão funcional do sistema a cada processo iterativo, mas cada ciclo é bem definido e possui uma série de tarefas:

- Comunicação com o cliente: Tarefas necessárias para estabelecer efetiva comunicação entre o desenvolvedor e o cliente;

- Planejamento: Tarefas necessárias para definir recursos, prazos e outras informações relacionadas ao projeto;

- Análise de risco: Tarefas necessárias para avaliar os riscos, tanto técnicos quanto gerenciais;

- Construção e liberação: Tarefas necessárias para construir, testar, instalar e fornecer apoio ao usuário;

- Avaliação pelo cliente: Tarefas necessárias para obter realimentação do cliente, com base na avaliação das representações do software criadas durante o estágio de engenharia e implementadas durante o estágio de instalação.

À medida que esse processo evolutivo tem início, a equipe de engenharia de software move-se em volta da espiral, no sentido horário, a partir do centro. O primeiro circuito em torno da espiral pode resultar no desenvolvimento da especificação do produto; passagens subseqüentes em torno da espiral podem ser usadas para desenvolver um protótipo e depois, progressivamente, versões mais sofisticadas do software. Cada passagem pela região de planejamento resulta em ajustes ao plano do projeto. O custo e o cronograma são ajustados com base na realimentação derivada da avaliação do cliente. Além disso, o gerente do projeto ajusta a quantidade de iterações necessárias planejada para completar o software. A figura abaixo resume o modelo.

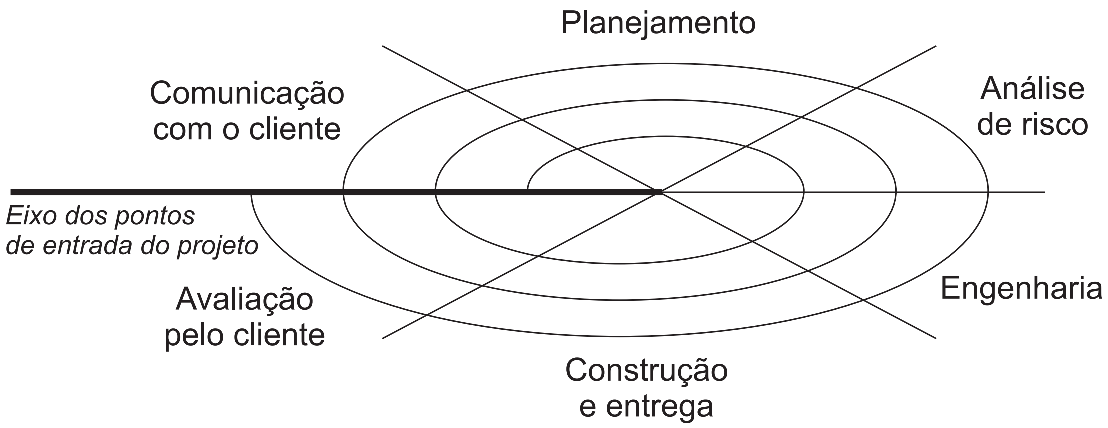

Figura 8 - Modelo espiral

Diferentemente dos modelos clássicos de processo, o modelo espiral pode ser adaptado para aplicação ao longo da vida do software. É uma abordagem realística para o desenvolvimento de sistemas e software de grande porte. Como o software evolui à medida que o processo avança, o desenvolvedor e o cliente entendem melhor e reagem aos riscos em cada nível evolucionário. O modelo espiral usa prototipagem como mecanismo de redução de risco, porém, o mais importante é que permite ao desenvolvedor aplicar a abordagem de prototipagem em qualquer estágio de evolução do produto. Ele mantém a abordagem sistemática passo a passo, sugerida pelo ciclo de vida clássico, ma o incorpora numa estrutura iterativa, que reflete mais realisticamente o mundo real. O modelo espiral exige a consideração direta dos riscos técnicos em todos os estágios do projeto e, se aplicado adequadamente, deve reduzir os riscos antes que eles fiquem problemáticos.

5.5.3 Modelo concorrente

No paradigma concorrente, as fases de desenvolvimento do software ocorrem simultaneamente. Uma determinada tarefa pode influenciar o estado de outra. Por exemplo, durante a codificação ou teste, é descoberta uma inconsistência na especificação. Esta então é refeita, e na seqüência, o novo desenho, a implementação e os testes. Mas isso ocorre apenas na parte do sistema a que isso diz respeito, nas demais o processo corre normalmente de forma paralela. Nos ambientes corporativos, é muito comum os sistemas serem desenvolvidos dessa maneira. Há pessoas definindo requisitos, outras codificando e outras testando, tudo ao mesmo tempo.

Na verdade, o modelo de processo concorrente é aplicável a todos os tipos de desenvolvimento de software e fornece um panorama preciso do estado atual de um projeto. Ao invés de confinar as atividades de engenharia de software a uma seqüência de eventos, ele define uma rede de atividades. Cada atividade da rede existe simultaneamente com outras atividades. Eventos gerados dentro de uma certa atividade ou em algum outro lugar da rede de atividades disparam transições entre estados de uma atividade.

5.6 Desenvolvimento baseado em componentes

As tecnologias de orientação a objetos fornecem o arcabouço técnico para um modelo de processo baseado em componentes para a engenharia de software. O paradigma de orientação a objetos enfatiza a criação de classes que encapsulam tanto os dados quanto os algoritmos usados para manipulá-los. Classes orientadas a objetos, se adequadamente projetadas e implementadas, são reusávies ao longo de diferentes aplicações e arquiteturas de sistemas baseados em computador.

O modelo de desenvolvimento baseado em componentes incorpora muitas das características do modelo espiral. Evolucionário por natureza, demanda uma abordagem iterativa para a criação de software. Todavia, esse modelo conta com componentes de software previamente preparados chamados *classes*.

A atividade de engenharia começa com a identificação das classes adequadas. Isso é conseguido pelo exame dos dados a serem manipulados pela aplicação e dos algoritmos a serem aplicados para efetuar a manipulação.

Classes criadas em projetos anteriores de engenharia de software são armazenadas numa biblioteca ou repositório de classes. Uma vez identificadas as classes adequadas, essa biblioteca é examinada para determinar se as mesmas já existem. Em caso afirmativo, são reusadas. Se uma classe apropriada não é encontrada, é então desenvolvida e depositada no repositório.

Esse modelo leva ao reuso de software e isso fornece aos engenheiros um certo número de benefícios mensuráveis. Com base em estudos de reuso, a QSM Associates, Inc. relata que a montagem de componentes leva a uma redução de 70% no prazo do ciclo de desenvolvimento; uma redução de 84% no custo do projeto e um índice de produtividade de 26,2%, comparado com o padrão de 16,9% para a indústria. Apesar desses resultados se deverem à robustez da biblioteca de componentes, existe alguns questionamentos quanto ao modelo de desenvolvimento baseado em componentes fornecer vantagens significativas para os engenheiros de software.

5.7 O modelo de métodos formais

O modelo de métodos formais abrange um conjunto de atividades que levam à especificação matemática formal do software de computador. Métodos formais permitem ao engenheiro de software especificar, desenvolver e verificar um sistema baseado em computador, pela aplicação de uma rigorosa notação matemática.

Quando são usados métodos formais durante o desenvolvimento, um mecanismo para eliminação de vários problemas difíceis de resolver usando outros paradigmas de engenharia de software é fornecido. Ambigüidade, inconclusão e inconsistência podem ser descobertas e corrigidas mais facilmente, não através das revisões comuns, mas através da aplicação de análise matemática. Quando são usados durante o projeto, métodos formais servem como base para a verificação do programa e assim permitem que o engenheiro de software descubra e corrija erros que poderiam passar despercebidos.

Apesar de não vir a ser uma abordagem de uso geral, o modelo de métodos formais oferece a promessa de software livre de defeitos. Todavia, foram formuladas as seguintes preocupações sobre sua aplicabilidade num ambiente comercial:

- O desenvolvimento de modelos formais é atualmente muito lento e caro;
- Como poucos desenvolvedores de software têm o preparo necessário para aplicar métodos formais, torna-se necessário um treinamento extensivo;
- É difícil usar os modelos como um mecanismo de comunicação, especialmente com clientes despreparados tecnicamente.

Apesar desses inconvenientes, a abordagem de métodos formais tem adeptos entre os desenvolvedores de software crítico em termos de segurança, como em navegação aeroespacial, ou entre desenvolvedores de software que podem sofrer pesadas sanções econômicas se ocorrerem erros no sistema.

5.8 Técnicas de quarta geração

Essas técnicas abrangem um amplo conjunto de ferramentas de software que têm uma coisa em comum: cada ferramenta permite que o engenheiro de software especifique alguma característica do software em alto nível. A ferramenta gera, então, código-fonte automaticamente baseado na especificação do desenvolvedor. Há um consenso de que quanto mais alto o nível de especificação do software para uma máquina, mais rapidamente o programa pode ser construído. Esse paradigma para engenharia de software concentra-se na habilidade de especificar software usando formas especializadas de linguagem, ou uma notação gráfica que descreve o problema a ser resolvido de forma que o cliente possa entender.

Atualmente, um ambiente de desenvolvimento de software que dá apoio a esse paradigma inclui ferramentas como: linguagens não-procedimentais para consulta a bases de dados, geração de relatórios, manipulação de dados, definição e interação com telas, geração de código, capacidade gráfica de alto nível, capacidade de disposição em planilha, geração automática de HTML entre outras.

Como todos os paradigmas de engenharia de software, esse modelo tem vantagens e desvantagens. Os defensores afirmam que para o desenvolvedor há uma drástica redução no tempo de desenvolvimento e uma melhora significativa na produtividade. Os oponentes alegam que as atuais ferramentas não são tão mais fáceis de usar quanto as linguagens de programação. O código-fonte resultante, produzido por tais ferramentas é ineficiente, e sua manutenibilidade é bem mais difícil.

Entretanto, as técnicas de quarta geração já se tornaram uma parte importante da engenharia de software. Quando combinado com as abordagens de desenvolvimento baseado em componentes esse paradigma pode se transformar na abordagem predominante para o desenvolvimento de software.

5.9 XP — Programação extrema

A Programação Extrema é uma metodologia ágil para equipes pequenas e médias desenvolvendo software com requisitos vagos e em constante mudança[18]. Não é adequada para projetos e/ou equipes grandes. Utiliza uma abordagem que permite que o desenvolvimento seja muito mais dinâmico e que o produto se adapte rapidamente a novos requisitos, mesmo que estes surjam no final do processo de desenvolvimento.

A Programação Extrema foi criada em resposta aos problemas que surgem quando os requisitos do sistema mudam. Os consumidores de software sob encomenda raramente possuem uma idéia firme e clara do que realmente necessitam. Para contornar esse problema, o sistema deve ser desenvolvido de tal forma que as funcionalidades possam ser alteradas com bastante freqüência. Em alguns contextos, como no mercado financeiro, as modificações de requisitos é uma constante. É nesse momento que a metodologia XP obtém sucesso, enquanto que as tradicionais, mesmo as evolutivas, fracassam.

A metodologia XP fundamenta-se em quatro pilares: comunicação, simplicidade, retroalimentação(*feedback*), e coragem. Programadores XP produzem seu código de maneira simples e clara, fator importante para a comunicação. O *design* do software é simplificado continuamente. É isso que sustenta a premissa extrema.

Comunicam-se constantemente com seus clientes e com outros programadores, mantendo um *feedback* constante com ambos. Isso permite que erros de *design*, programação ou mesmo de interpretação sejam corrigidos tão cedo quanto for possível. Assim, o desenvolvimento é guiado constantemente e se evita que se distancie da expectativa do cliente.

É uma metodologia muito dinâmica, com ciclos de produção pequenos que atendem às alterações constantes dos requisitos. Para segui-la, é preciso coragem para apontar um problema no projeto, parar quando você está cansado, pedir ajuda quando necessário, simplificar código que já está funcionando, dizer ao cliente que não será possível implementar um requisito no prazo estimado, fazer alterações no processo de desenvolvimento, etc. Ou seja, fazer a coisa certa mesmo que não seja a coisa mais popular naquele momento.

Softwares projetados de forma simples e elegante são muito mais interessantes que um sistema complexo e difícil de se manter. Num projeto típico de desenvolvimento de software, são gastos cerca de vinte vezes mais em recursos humanos que em hardware. Por exemplo, um projeto que gasta 2 milhões de dólares com programadores por ano, deve gastar por volta de 100 mil dólares em equipamentos durante o mesmo período. Suponhamos agora que programadores experientes e engenhosos que, com alguns truques de programação, reduzam em 20% os gastos com hardware. Isso representaria uma economia de 20 mil dólares por ano, porém também resultaria num código fonte bem menos legível. Se por outro lado produzíssemos um sistema mais simples e claro, fácil de se entender e manter, poderíamos conseguir, por exemplo, uma economia de 10% nos custos de programação. Isso representaria 200 mil dólares em um ano, uma soma bem maior, e que certamente iria influenciar o custo final do produto. A simplicidade é um fator fundamental na qualidade e para permitir alterações durante o ciclo de desenvolvimento, e na própria manutenção. Nas metodologias tradicionais entretanto, muitas vezes esse quesito é negligenciado já que, o que o cliente vê é a funcionalidade e não como ela foi implementada.

A metodologia XP lida de forma bastante eficiente com modificações de requisitos, mesmo que surjam no final. A comunicação interna e o *feedback* constante do cliente durante o processo de desenvolvimento, contribui de forma decisiva para isso. Quando surge uma alteração de requisito, esta é passada rapidamente para a equipe de desenvolvimento de modo que as alterações ainda possam ser feitas. Não é necessário esperar a finalização de um longo ciclo de desenvolvimento, quando pode ser tarde demais.

A Programação Extrema enfatiza os testes, sobretudo os testes bem feitos. Testes realizados de forma automática e correta, dão segurança para os programadores e conseqüentemente para os consumidores. Em XP, os testes são criados antes e durante a programação. Fornecem uma importante parcela do *feedback* interno ao grupo de desenvolvimento e permite que os *bugs* sejam rapidamente encontrados e corrigidos, impedindo-os de se propagarem. Um código simples e claro também contribui para isso.

Ao contrário do que muitos, que desconhecem a metodologia, afirmam, a programação extrema é de fato um modelo de processo em engenharia de software. É baseada nos fundamentos de organização e sistemática da engenharia, porém é um modelo muito mais dinâmico que os convencionais; mas que vem provando ser de grande eficiência, se aplicado nas circunstâncias certas e de maneira correta.

Capítulo 6

Programação Orientada a Objetos e Padrões

Neste capítulo são apresentados de forma bem sucinta alguns aspectos da programação orientada a objetos em C++. Esses conceitos podem ser bastante poderosos se aprofundados e usados de maneira correta. Na segunda parte, há uma breve introdução ao conceito de padrões e a descrição de dois deles envolvidos na elaboração de um programa de simulação de sistemas dinâmicos.

6.1 Programação orientada à objetos em C++

A partir da década de 90, a programação orientada à objetos tornou-se o principal paradigma de linguagens de programação[13]. Isso se deve ao fato desse paradigma favorecer o encapsulamento e a modularidade, fatores importantes na reutilização de código e conseqüente redução nos custos de desenvolvimento e manutenção do software.

O C++ foi criado por Bjarne Stroustrup no começo dos anos 80, e tinha dois objetivos principais: Manter a compatibilidade com o C original, extendendo-o para ganhar a característica de orientação a objetos. É uma das linguagens mais eficientes em termos de velocidade de execução, mas que possui uma série de inconvenientes para os programadores. Entre vários problemas, dois de seus pontos mais fracos são a ausência da coleta de lixo e a permissão de misturar indiscriminadamente código orientado a objetos com código sem orientação.

6.1.1 O que é um objeto?

Uma classe é uma entidade que agrupa dados, chamados de atributos, e funções, chamadas de métodos. Esses dois elementos formam um conjunto coeso. A idéia por trás disso, é que toda manipulação desses atributos seja feita através dos métodos da mesma classe. Um objeto é uma instância de uma classe.

Suponhamos, por exemplo, a classe *matriz de números reais*. Ela possui alguns atributos que representam seus elementos e mais dois indicando sua dimensão. Possui métodos para calcular o determinante e a matriz transposta, além daqueles para carregar e recuperar os valores dos elementos. Nota-se que é um conjunto coeso. Os algoritmos utilizam os atributos de dimensão da matriz para realizar as operações de modo seguro, sem exceder os limites da mesma. Esse é um exemplo simples; mas quando existem milhares de dados e centenas de operações num programa, essa característica de empacotamento das classes é de grande auxílio para se manter a ordem no código e se evitar que tudo se transforme num caos.

6.1.2 Encapsulamento e modularidade

Como já foi mencionado no capítulo anterior, a reutilização de código é extremamente benéfica para o desenvolvedor, pois além de reduzir os custos de desenvolvimento e manutenção, auxilia a criação de programas com maior qualidade. Mas para que isso seja possível, é necessário que as funcionalidades providas por esses componentes reutilizáveis sejam independentes do contexto em que são utilizadas. A orientação a objetos favorece isso através de suas características de modularidade e encapsulamento inerentes.

Essas duas características estão muito relacionadas entre si. A modularidade refere-se à independência do objeto em relação ao seu contexto, isto é, podemos trocar um módulo por outro num programa sem grandes ou nenhuma alteração no restante do conjunto. Já o encapsulamento refere-se à característica do objeto especificar a sua funcionalidade sem expor seu funcionamento interno. Um objeto não pode ser modular sem ser encapsulado. Caso não seja encapsulado, outras partes do programa poderiam passar a depender da sua implementação interna, amarrando-o ao conjunto, de modo que a troca por outro não seria efetuada de maneira simples. Por outro lado, um objeto encapsulado possui uma interface bem definida e seu interior ocultado, assim a sua modularidade está implícita.

6.1.3 Sobrecarga

A sobrecarga é a capacidade de se especificar diversas funcionalidades para um mesmo método, ficando a diferenciação a cargo dos argumentos fornecidos na chamada. Por exemplo, poderíamos ter um método *print(int)* e outro *print(String)*, cada um deles com sua codificação própria.

Mas talvez a principal vantagem desta característica seja a sobrecarga de operadores. Por exemplo, podemos fazer com que o operador + quando aplicado à Strings resulta na concatenação das mesmas, ou quando aplicado à matrizes, resulte na soma delas. A palavra chave **operator** que precede o nome do operador, faz com que a declaração seja interpretada da forma correta. Temos a seguir, um exemplo de código que faz isso. No final, a variável *both* conterá a string concatenada.

```
//Sobrecarregando o operador + e o método print
class my_string{
  public:
    my_string();
    void assing(const char* st);
    int len();
```

```
    void print(int n) const{
      for(int i; i<n; i++)
        cout << s << endl;
    }
    void print(String str) const{
      cout << str << s << endl;
    }
    friend my_string operator+(const my_string& a, const my_string& b);
  private:
    char* s;
    int len;
};
...
//Definição do operador +
my_string operator+(const my_string& a, const my_string& b){
  my_string* tmp = new my_string(a.len + b.len);
  strcpy(tmp->s, a.s);
  strcat(tmp->s, b.s);
  return *tmp;
}
...
// Uso
my_string one, two, both;
one.assign(``Minha Terra tem palmeiras '');
two.assign(``onde canta o sabiá.'')
both = one + two;
both.print(1);
```

6.1.4 Herança e polimorfismo

O conceito de herança foi criado com o objetivo de se reutilizar código. Através dela, uma classe(derivada) pode ser construída com base em outra, a super classe. Isso faz com que a classe derivada herde os atributos e métodos da superclasse. A nova classe pode então especializar alguns desses métodos herdados através de sua redefinição ou ainda implementar novos. Isso também pode ser feito com relação aos atributos. Dessa maneira, pode-se construir classes mais complexas e/ou especializadas, a partir de classes mais simples. Por exemplo, numa aplicação poderíamos ter uma classe denominada *pessoa* e derivada dessa, duas classes: *professor* e *aluno*. Os métodos e atributos comuns ficam na classe *pessoa*, enquanto que os específicos ficam nas suas respectivas classes.

Em C++ é permitida a herança múltipla, isto é, uma classe pode ser derivada de outras duas ou mais classes. Embora possa parecer uma característica interessante, é uma péssima prática, e deve ser evitada a todo custo. A maioria das outras linguagens orientadas a objetos, como Java e Smalltalk, não aceitam herança múltipla.

Como conseqüência da herança, tem-se o polimorfismo. Essa característica permite que, em tempo de execução, seja determinado qual método apropriado deve ser invocado. Para ficar mais claro, tomemos o exemplo da hierarquia de pessoa, professor e aluno citados. Suponhamos que desejamos imprimir um crachá para todas as pessoas. Nos crachás dos professores deverá vir impresso "PROFESSOR" e terá também uma foto, enquanto que para os alunos, conterá apenas a especificação "ALUNO". Naturalmente, os métodos que imprimem os crachás são diferentes. Numa linguagem não orientada a objetos, deveria se testar se a pessoa é um aluno ou professor e só então chamar a função correspondente. Mas na programação orientada a objetos, isso é feito de maneira automática quando se chama o método. Em tempo de execução, é determinado se a pessoa é um aluno ou professor, e então o método específico daquela classe é chamado. Essa é uma característica extremamente poderosa se bem explorada. O trecho de código a seguir mostra como fazer isso em C++:

```
//Classe Pessoa
class person{
  public:
    person(String Name, int CPF){
      name = new String(Name);
      cpf = CPF;
    }
    virtual void print();
    ...
  private:
    String name;
    int cpf;
    ...
};

//Classe Professor
class professor : public person{
  public:
    professor(String Name, int CPF, Bitmap Photo):person(Name, CPF){
      photo = new Bitmap(Photo);
    }
    void print();
    ...
  private:
    Bitmap photo;
    ...
};

//Classe Estudante
class student : public person{
  public:
    student(String Name, int CPF):person(Name, CPF){
```

```
      name = new String(Name);
      cpf = CPF;
    }
    void print();
    ...
};
...
// Uso
...
pessoa* ptr_pessoa;
ptr_pessoa = new student(``Edgar'', ``1234567890'');
ptr_pessoa->print();
...
ptr_pessoa = new professor(``Knuth'', ``0987654321'');
ptr_pessoa->print();
```

Nota-se nas quatro últimas linhas que o método *print()* é chamado na classe pessoa. Mas o mesmo é declarado como virtual, portanto, será invocado o método correspondente à classe mais derivada a que a pessoa pertence, isto é, professor ou aluno. Em C++, a palavra reservada **virtual** é necessária para que haja o polimorfismo.

6.1.5 Tratamento de exceções

Em C++, assim como em outras linguagens orientadas a objetos, existe um mecanismo de tratamento de erros chamado *exception handling*. Esse mecanismo permite que o tratamento de erros seja feito de maneira mais simples. Ao invés de se verificar o resultado de cada operação para garantir que tudo corre dentro do esperado, pode-se colocar as operações sujeitas a erros dentro de um bloco (*try*) e associado a este, outros blocos (*catch*) de tratamento de erros. O exemplo a seguir deixa mais claro como isso é feito. A exceção é lançada através da instrução **throw**.

```
// Construtor de uma pilha com exceções
stack::stack(int n){
  if (n<1)          // O tamanho da pilha deve ser positivo
    throw (n);
  p = new char[n];
  if (p==0)         // Quando new falha, retorna
    throw(``Free store exhausted'')
  top = EMPTY;
  max_len = n;
}
...
// Uso da pilha
try{
  stack a(n), b(n);
  ...
```

```
}
catch(int n) { ... }        // Tratamento de erro para argumento inválido
catch(char* error) { ... } // Tratamento de erro para pilha não alocável
...
```

6.1.6 *STL — Standard template library*

Essa biblioteca provê uma série de componentes em estruturas de dados, como vetores, listas de vários tipos, mapas e algoritmos para sua manipulação. Essa biblioteca foi escrita usando-se o conceito de *template*.

Esse conceito permite sejam criados algoritmos capazes de manipular dados genéricos. Para ficar mais claro, tomemos como exemplo a ordenação de números inteiros e de caracteres. Quando se resolve o primeiro problema, usa-se normalmente variáveis auxiliares do tipo inteiro, enquanto que no segundo são usadas variáveis do tipo caractere. Usando uma programação convencional, precisaríamos criar duas funções, uma para cada tipo, embora semanticamente, fazem a mesma coisa, são o mesmo algoritmo. O padrão *template* surgiu exatamente para resolver esse problema. Ao escrever tal função, ao invés de se especificar as variáveis como *int* ou *char*, especifica-se como um tipo T por exemplo. Ao ser instanciado um objeto que usa esse padrão, será então especificado o tipo de variável. Isso aplica-se não apenas a tipos básicos como aqui exemplificados, mas também com objetos. O exemplo abaixo mostra como esse padrão pode ser usado:

```
// Exemplo de template
#include <iostream>
using namespace std;
template<class T> class Array{
    T A[100];
  public:
    T& operator[](int index){
      return A[index];
    }
};
...
int main() {
  Array<int> ia;
  Array<float> fa;
  for(int i = 0; i < 20; i++) {
    ia[i] = i * i;
    fa[i] = float(i) * 1.414;
    cout << ia[i] << '', `` << fa[i] << endl;
  }
}
```

6.2 Padrões

Os programadores tendem a basear a solução para algumas partes de seu código em trabalhos de outros programadores, geralmente mais experientes[16]. Esse processo de imitação, que não é exatamente uma cópia, envolve a percepção de um padrão nesse código que pode ser adaptado ao problema em mãos. Isso ocorre desde os primórdios da computação.

O conceito de padrões pode ser visto como uma abstração desse processo de imitação. Em outras palavras, os *design patterns* constituem um conjunto de regras descrevendo como resolver determinados tipos de problemas no desenvolvimento de software. Os padrões são descritos conforme algumas regras para que sejam claros e objetivos quanto a sua utilização.

Os padrões surgem de problemas práticos. Os desenvolvedores normalmente se deparam com vários problemas em comum. Cada um encontra uma solução diferente para o problema. Mas algumas são melhores que as outras e com o passar do tempo, num processo de seleção natural, apenas a melhor persiste. Surge então um padrão.

Podemos citar um exemplo de padrão: Suponhamos que haja a necessidade de se acessar um objeto remotamente. Isso pode ser feito de várias maneiras, mas o padrão para esse problema sugere a utilização de um desenho específico chamado *proxy*. Basicamente, é instanciado um objeto local que representa o objeto remoto. Toda comunicação remota é feita apenas por esse objeto local que possui toda funcionalidade do objeto remoto. Quando um método é invocado, ele repassa a mensagem para o objeto remoto, toma o resultado da operação e finalmente retorna esse resultado ao objeto que o invocou. Dessa forma, o restante do programa pode acessar um dado remoto como se estivesse acessando-o localmente. Esse padrão é a base da arquitetura CORBA, discutida na próxima seção. A seguir são discutidos com mais detalhes dois padrões muito conhecidos e que foram utilizados na elaboração do programa de simulação de sistemas dinâmicos.

6.2.1 Padrão *MVC* — *Model/View/Controller*

Esse padrão é utilizado para implementação de interfaces gráficas com usuário(*GUI*). O MVC reflete o fato de que uma aplicação com interface gráfica consiste de três componentes: O modelo, a visualização e o controle. Esses componentes impõem uma estrutura às aplicações interativas.

O modelo corresponde às estruturas de dados contendo as informações internas da aplicação, de forma completamente independente de qualquer interface com usuário.

A visualização é o componente responsável por adaptar as informações contidas no modelo e apresentá-las na tela. Podem existir diversas visualizações numa mesma aplicação. Por exemplo, uma tabela ou um gráfico são duas representações possíveis de um mesmo conjunto de dados.

O controle é a parte responsável por obter os comandos do usuário. Cada visualização possui um controle associado. Dessa forma, uma aplicação pode conter vários pares controle/visualização. Uma pequena variante desse padrão é o M/VC, onde a visualização e o controle são tratados como um mesmo componente. A figura a seguir mostra a interdependência desses três componentes. As setas indicam a direção das mensagens(chamadas a métodos).

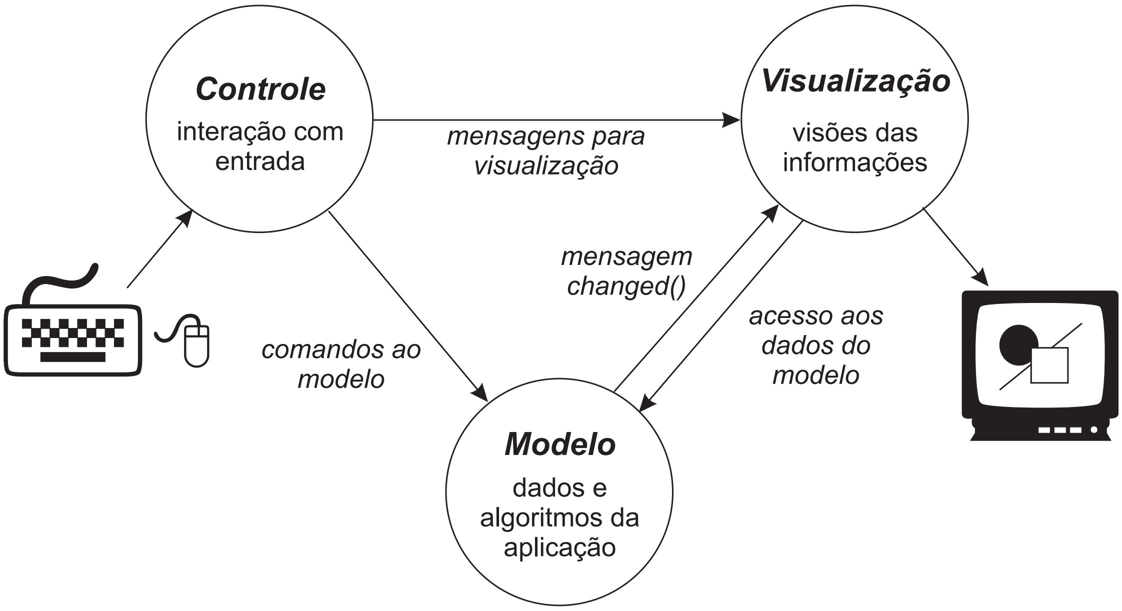

Figura 9 - Arquitetura MVC

O estado da visualização depende do controle. Por exemplo, quando uma janela é redimensionada, a mesma deve ser redesenhada. Os *frameworks* para desenvolvimento de interfaces gráficas normalmente fazem isso de forma automática. Quando um evento de teclado ou mouse acontece, se necessário, são enviadas mensagens como *redraw() ou refreshWindow()* para todas as janelas de visualização. Por outro lado, entretanto, não existe um fluxo de mensagens da visualização para o controle.

Evidentemente que o controle também acessa o modelo. É através dessa interação que o usuário pode dar comandos para a aplicação realizar as tarefas desejadas. Da mesma forma que acontece com a visualização, o modelo não acessa o controle, a menos das inicializações e algumas poucas funções especiais.

Mas o modelo e a visualização estão em constante troca de mensagens. Geralmente, a visualização envia mensagens ao modelo solicitando dados para serem apresentados na tela, por exemplo durante a execução de métodos como *redraw()*. Do outro lado, quando um modelo sofre alguma alteração que deve ser visível, este notifica a visualização, que então envia as mensagens ao modelo solicitando os dados atualizados. Essa interação do modelo com a visualização forma a base de outro padrão, o observador.

6.2.2 Padrão observador

Esse padrão é utilizado quando há um ou mais elementos dependentes da atualização de outro. Por exemplo, várias janelas mostrando simultaneamente os dados de uma aplicação, uma em forma de tabela e outra em forma de gráfico. Quando algum valor da aplicação é alterado, ambas as janelas devem ser atualizadas. Isso é conseguido através do registro na aplicação dos dependentes, no caso as duas janelas. Toda vez que uma alteração na aplicação ocorre, ela envia a mensagem *changed()* ou algo equivalente para todos os dependentes, isto é, os observadores. Esses dependentes, as janelas, solicitam então os novos valores à aplicação. Algumas vezes, a mensagem *changed()* é enviada com parâmetros informando em que aspecto a aplicação mudou. Os observadores podem usar esses parâmetros para decidir então se alguma coisa deve ser feita ou não. Isso pode evitar uma sobrecarga de atualização desnecessária.

Capítulo 7

Introdução ao CORBA

Neste capítulo é dada uma pequena introdução à arquitetura distribuída CORBA.

7.1 Visão geral

O CORBA(*Common Object Request Broker Architecture*) é um *framework*, bem estabelecido e aceito na indústria, voltado para o desenvolvimento de aplicações distribuídas[17]. Não é um produto, mas sim uma especificação da OMG(*Object Management Group*). Existem diversas implementações de ORBs para vários sistemas operacionais, algumas comerciais e outras livres.

Sua principal característica é a flexibilidade de operar em diversas linguagens e plataformas. Permite, por exemplo, que uma parte da aplicação seja desenvolvida em C++ executando num ambiente Windows PC e a outra parte em Java, numa estação da Sun.

Para que isso seja possível, o padrão especifica cinco elementos importantes:

- *IDL — Interface Definition Language*: Para que um programa invoque uma operação de um objeto distribuído, é necessário que ele conheça sua interface. A IDL é uma linguagem declarativa de especificação de interfaces que permite interoperabilidade entre diversas linguagens e plataformas. Ela é descrita com mais detalhes na próxima seção;

- Mapeamento da IDL para várias linguagens: A IDL não é uma linguagem computacionalmente completa, assim, não é possível escrever programas com ela. Seu objetivo é apenas definir as interfaces dos objetos remotos. Mas através de um mapeamento padronizado, os tipos de dados e funções definidas de forma comum na IDL são mapeadas para o tipo e função correspondentes na linguagem alvo, como C++ ou Java. Já foi especificado o mapeamento para uma série de linguagens como C, C++, Smalltalk, Java, COBOL, Ada, entre outras;

- Invocação remota de métodos: É um mecanismo que permite que um programa cliente invoque um método de um objeto remoto da mesma maneira que invoca um método de um objeto local. É possível realizar uma invocação estática, isto é, quando já se conhece em tempo de compilação a classe do objeto alvo. A outra forma é a dinâmica, onde em tempo de execução é determinada a classe do objeto alvo. Essa forma porém requer uma programação extra. A figura 10 mostra como esse processo é realizado. As classes stub e skeleton são geradas automaticamente pelo ORB durante a compilação;

- Adaptadores de objetos: Os objetos no servidor que são instâncias de um objeto remoto são chamados de serventes. Os adaptadores de objetos são intermediários[1] entre o cliente e esse servente com a finalidade de tornar mais robusta e flexível a interface dos objetos remotos. Por exemplo, é possível através desses adaptadores instanciar serventes conforme demanda. Em outras palavras, é possível invocar a operação de um objeto remoto sem que o mesmo esteja carregado em memória. O adaptador de objetos se encarrega de instanciá-lo quando necessário.

- Protocolos Inter-ORB: Toda comunicação entre os ORBs é feita através do protocolo GIOP(*General Inter-ORB Protocol*). Normalmente é utilizado o protocolo IIOP(*Internet Inter-ORB Protocol*) que é a implementação do GIOP sobre TCP/IP. Isso permite que ORBs de diferentes fabricantes se comuniquem.

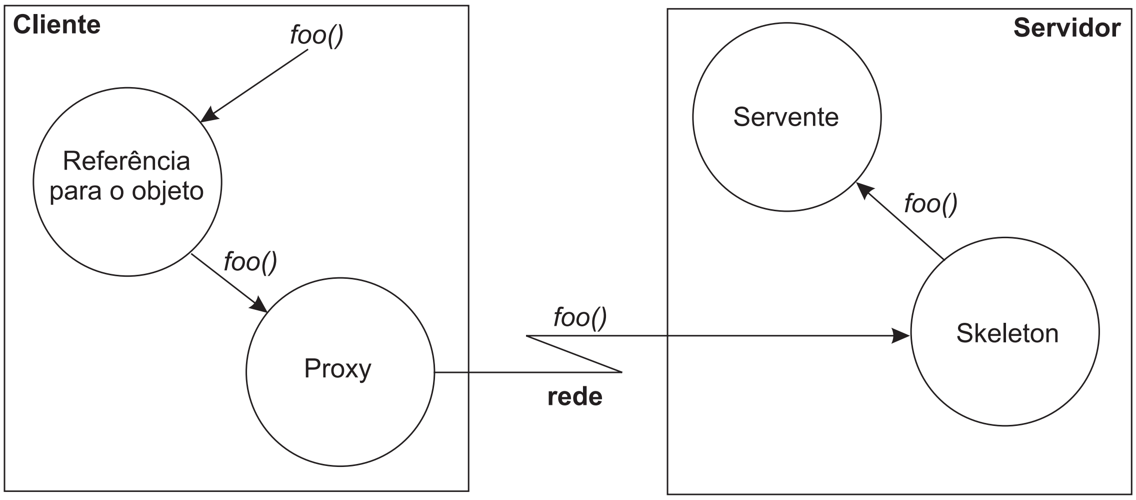

Figura 10 - Invocação remota de método em CORBA

7.2 A IDL — *Interface Definition Language*

A IDL da OMG é o principal mecanismo de abstração para separar as interfaces dos objetos de suas respectivas implementações. A IDL estabelece um contrato entre o cliente e o servidor que descreve os tipos de dados e as interfaces dos objetos utilizados numa aplicação. Esta descrição é independente da linguagem de implementação e da plataforma.

[1] Não confundir esses intermediários com os stubs e skeletons

Durante o processo de compilação, que é específico da linguagem e plataforma, são geradas classes e estruturas de dados que representam aquelas definidas na IDL. Entre elas estão o *stub* e o *skeleton*. Uma vez gerado esse código fonte a partir da IDL de maneira automática, prossegue-se com a implementação e compilação propriamente ditas na linguagem e plataforma escolhidas. Esse processo é feito no lado cliente e no lado servidor.

A IDL é uma linguagem de especificação bastante extensa e não será abordada com detalhes por questões de espaço. Entretanto, na próxima seção há um exemplo de aplicação contendo o código fonte da IDL e da implementação.

7.3 Um exemplo simples de aplicação

hello.idl

```
// IDL para classe HelloWorld
interface HelloWorld {
  void hello ();
};
```

server.cc

```
// HelloWorld Server
#include <fstream.h>
#include "hello.h"
#include <stdlib.h>

// HelloWorld class implementation
class HelloWorld_impl : virtual public POA_HelloWorld {
public:
  void hello ();
};

void HelloWorld_impl::hello () {
  printf ("Hello World!\n");
}

// Server main program
int main (int argc, char *argv[]) {
  // Initialize the ORB
  CORBA::ORB_var orb = CORBA::ORB_init (argc, argv);

  // Obtain a reference to the RootPOA and its Manager
  CORBA::Object_var poaobj = orb->resolve_initial_references ("RootPOA");
  PortableServer::POA_var poa = PortableServer::POA::_narrow (poaobj);
  PortableServer::POAManager_var mgr = poa->the_POAManager();
```

```
  // Create a Hello World object
  HelloWorld_impl * hello = new HelloWorld_impl;

  // Activate the Servant
  PortableServer::ObjectId_var oid = poa->activate_object (hello);

  // Write reference to file
  ofstream of ("hello.ref");
  CORBA::Object_var ref = poa->id_to_reference (oid.in());
  CORBA::String_var str = orb->object_to_string (ref.in());
  of << str.in() << endl;
  of.close ();

  // Activate the POA and start serving requests
  mgr->activate ();
  orb->run();

  // Shutdown (never reached)
  poa->destroy (TRUE, TRUE);
  delete hello;
  return 0;
}
```

client.cc

```
// HelloWorld client program
#include "hello.h"

int main (int argc, char *argv[]){
  // Initialize the ORB
  CORBA::ORB_var orb = CORBA::ORB_init (argc, argv);

  // IOR is in hello.ref in the local directory
  if (CORBA::is_nil(orb)) {
    printf ("oops: could not locate HelloWorld server\n");
    exit (1);
  }
  char pwd[256], uri[300];
  sprintf (uri, "file://%s/hello.ref", getcwd(pwd, 256));

  // Bind to Hello World server
  CORBA::Object_var obj;
  obj = orb->string_to_object (uri);
  HelloWorld_var hello = HelloWorld::_narrow (obj);
```

```
  if (CORBA::is_nil(hello)) {
    printf ("oops: could not locate HelloWorld server\n");
    exit (1);
  }

  // Invoking remote method
  hello->hello ();
  return 0;
}
```

Para compilar esse programa, inicialmente executa-se o compilador IDL $\Rightarrow$ C++ no lado servidor e no lado cliente. Isso irá gerar as classes stub e skeleton. Prossegue-se então com a compilação usual e tem-se os dois programas prontos para serem executados em ambiente distribuído.

Entretanto, observando-se com cuidado o código, nota-se que o cliente obtém a referência para o objeto remoto usando o sistema de arquivos. Mas como obter essa referência se o cliente estiver executando em outra máquina que não possui acesso ao sistema de arquivos da máquina onde está sendo executado o servidor? Na verdade, essa forma de se obter a referência é utilizada apenas para trabalhos simples, geralmente em redes locais. O CORBA provê uma forma muito mais robusta para se obter referências através do serviço de nomes, uma das funcionalidades extras do CORBA discutidas na próxima seção.

7.4 Serviços extras do CORBA

O CORBA provê uma série de serviços que ajudam a tornar mais robusta a aplicação distribuída. A seguir são discutidos sucintamente os cinco serviços principais.

7.4.1 *Naming service*

O serviço de nomes permite que referências a objetos sejam associadas a nomes. Numa analogia à lista telefônica, o número do telefone corresponde à referência enquanto que o nome do assinante corresponde ao nome do objeto. No exemplo mostrado na seção anterior, ao invés de se obter a referência para o objeto remoto HelloWorld através de um arquivo, poderia-se obtê-la através do serviço de nomes consultando, por exemplo, pelo nome HelloWorldServer.

O serviço de nomes possui uma estrutura hierárquica semelhante à estrutura de diretórios num sistema de arquivos. Os "diretórios" são chamados de contextos, possibilitando uma melhor organização das entradas.

7.4.2 *Trading service*

Esse serviço tem uma analogia com as páginas amarelas. Ele permite que as propriedades dos objetos sejam anunciadas para potenciais clientes que o consultam para obter referências para objetos com as propriedades desejadas. Por exemplo, um cliente pode estar solicitando por um serviço de impressão.

O *Trading service* é bem mais complexo que o *Naming service* e não é muito utilizado na prática, ao contrário do serviço de nomes.

7.4.3 *Event service*

Em certas situações, a comunicação desacoplada é mais interessante que um esquema direto de requisição/resposta. É útil no fornecimento de informações sobre condições relativamente raras, como disco cheio por exemplo. Quando um objeto muda de estado, notifica os outros objetos interessados nessa mudança. Note que esse modelo segue o padrão observador discutido no capítulo anterior.

7.4.4 *Security service*

A preocupação com segurança é algo inerente aos sistemas distribuídos. A especificação do CORBA descreve mecanismos de identificação, autenticação, autorização, controle de acesso, auditoria, proteção de mensagem, imputabilidade, autenticação mútua ou one-way, garantias de integridade e confidencialidade. Entretanto, é uma especificação bastante ousada e por esse motivo é implementada apenas em parte pelos ORBs.

7.4.5 *Transaction service*

Outra característica inerente aos sistemas distribuídos é a concorrência. A gerência de transações pode ser facilitada pelo serviço de transações do CORBA. Ele permite que transações possam ser feitas envolvendo diversos objetos distribuídos, garantindo as quatro propriedades ACID: atomicidade, consistência, isolação e durabilidade. CORBA especifica o mecanismo de *two phase commit* para a efetivação das transações distribuídas.

Referências Bibliográficas

[1] Lehninger, Albert L.; Nelson, David L.; Cox, Michael M.; *Principles of Biochemistry, 2nd edition*, Worth Publishing, 1993

[2] Voet & Voet; *Biochemistry, 2nd edition*, John Wiley & Sons, 1995

[3] Stryer, Lubert; *Biochemistry, 4th edition* , W H Freeman and Co, 1995

[4] Lawrence, Peter A.; *The Making of a Fly: The Genetics of Animal Design*, Blackwell Science Inc, 1992

[5] Desoer, C.A.; *A Second Course on Linear Systems*, Van Nostrand Reinhold Co,1969

[6] Bavel, Z.; *Introduction to the Theory of Automata*, Reston Publishing Company, 1983

[7] Holmgren, Richard A.; *A First Course in Discrete Dynamical Systems*, Springer-Verlag, 1994

[8] Monteiro, Luiz H.A.; *Sistemas Dinâmicos*, Editora Livraria da Física, 2002

[9] Alligood, K.T.; Sauer, T.D.; Yorke, J. A.; *Chaos - An Introduction to Dynamical Systems*, Springer-Verlag,1996

[10] Nusse, H.E.; Yorke, J. A.; *Dynamics - Numerical Explorations,* 2^{nd} *edition*, Springer-Verlag, 1997

[11] Goodman, Roe; *Introduction to Stochastic Models*, Benjamin/Cummings, 1988

[12] Lawer, Gregory F.; *Introduction to Stochastic Processes*, Chapman & Hall, 1996

[13] Pohl, Ira; *Object-oriented Programming using C++*, Addison-Wesley Pub Co, 1997

[14] Sommerville, Ian; *Software Engineering, 6th edition*, Addison-Wesley Pub Co., 2000

[15] Pressman, Roger S.; *Engenharia de Software, 5a edição*, McGraw Hill, 2002

[16] Pree, Wolfgang; *Design Patterns for Object-Oriented Software Development*, Addison-Wesley Pub Co, 1995.

[17] Vinoski, M.; Henning, S.; *Advanced CORBA Programming with C++*, Addison-Wesley Pub Co, 1999

[18] Ken Auer and Roy Miller, *Extreme Programming Applied*, Addison Wesley, 2002.