Tutorial

Revista de
Informática
Teórica e Aplicada

O Paradigma de Desenvolvimento por Objetos

María del Rosario Girardi
Roberto Tom Price
PGCC/II - Universidade Federal do Rio Grande do Sul

Sumário

São analisados os conceitos relevantes da Programação Orientada a Objetos desde o ponto de vista da Engenharia de Software. O Paradigma de Desenvolvimento por Objetos é caracterizado e comparado com o estilo de desenvolvimento baseado em decomposição funcional, enfatizando as principais contribuições do paradigma para o incremento da produtividade e confiabilidade de software. Discutem-se as abordagens de desenvolvimento mais relevantes e descreve-se, em detalhe, uma metodologia de desenvolvimento que atende todas as fases do ciclo de vida de software. Identificam-se as limitações existentes na aplicação do paradigma, sugerindo-se a criação de um conjunto de ferramentas de apoio ao desenvolvimento e manutenção de sistemas orientados a objetos.

Abstract

From a Software Engineering point of view, some important concepts of Object-oriented Programming are analysed. The Object-development Paradigm is characterized and compared with functional decomposition-based development, emphasizing their contributions to the increase of software productivity and reliability. Some development approaches are discussed and a development methodology, which covers all the life cycle phases is exposed in detail. Current limitations to the application of the paradigm are identified and a toolkit to support the development and maintenance of object-oriented systems is proposed.

Palavras-chave: Programação Orientada a Objetos, Desenvolvimento Orientado a Objetos, Metodologias de Desenvolvimento Orientadas a Objetos, Ferramentas de Apoio ao Desenvolvimento Orientado a Objetos.

1. Introdução

Os problemas existentes na produção de sistemas de software complexos e confiáveis, com baixo custo de desenvolvimento e manutençãq, têm levado à

pesquisa de novos paradigmas de programação como programação em lógica, funcional e orientada a objetos.

Atualmente as aplicações são de natureza muito diversa. Representar e simular o comportamento dessas aplicações com as linguagens algorítmicas convencionais (Fortran, Algol, C e outras) não é simples nem eficiente, em parte, pela dificuldade no mapeamento de um modelo da realidade (interações entre objetos) num modelo computacional (programas e dados).

A percepção dos aspectos mais importantes do mundo real ou *Espaço de Problemas*, para fins de representação em computador, envolve, fundamentalmente, um processo de identificação de abstrações. Se essas abstrações não tiverem uma expressão direta (ou próxima) no mundo computacional, ou *Espaço de Soluções*, a complexidade da solução será evidentemente aumentada pela distância entre os dois espaços. Costuma-se chamar essa distância de "Gap" semântico [TAK 88]. Partindo da assertiva de que o mundo real é um mundo de objetos que interagem entre si, o *Paradigma de Desenvolvimento por Objetos* (PDO) constitui uma abordagem que minimiza o "gap" semântico entre os *problemas* no mundo real e as *soluções* no mundo computacional.

A proximidade entre os dois espaços permite que os sistemas desenvolvidos segundo o PDO possam ser concebidos e implementados numa forma próxima a como a mente humana percebe o mundo da aplicação: como uma interação entre objetos distintos, cada um possuindo propriedades e comportamento próprio e onde cada objeto pode ter outros objetos como componentes de sua estrutura [JON 87].

Por outra parte, o PDO atende à natureza repetitiva do processo de construção de software. Fornecendo facilidades para a reutilização de componentes, este estilo de desenvolvimento promove o desenvolvimento de aplicações de forma semelhante à construção de sistemas físicos: como objetos complexos constituídos pela reunião de diversos objetos pré-fabricados de uso geral. Essa capacidade de reuso tem conseqtências favoráveis para incrementar a produtividade no desenvolvimento de software: redução de custos e aumento da confiabilidade e manutenibilidade das aplicações.

Este artigo pretende caracterizar o PDO e analisar os conceitos relevantes da Programação Orientada a Objetos desde o ponto de vista da Engenharia de Software. São discutidas as soluções oferecidas pelo paradigma às dificuldades existentes na construção de software e os problemas que ainda existem na sua aplicabilidade. Também são analisadas as abordagens de desenvolvimento mais relevantes e é apresentada uma metodologia de desenvolvimento orientada a objetos que atende todas as fases do ciclo de vida de software.

2. Antecedentes

Diversas áreas têm influenciado a conceitualização do paradigma de objetos: Linguagens e Sistemas Operacionais, Especificação Algébrica e Tipos Abstratos

de Dados, Banco de Dados e Inteligência Artificial (técnicas para representação de conhecimento).

Das linguagens para simulação de sistemas, *Simula* [DAH 66] introduz os conceitos de *classe* e *herança*.

A evolução dos estudos em programação e verificação de programas, nos anos 70, levou à formalização do conceito de *Tipo Abstrato de Dado* (TAD), com a idéia básica de separar a utilização de um tipo de dados (a interface externa) de sua implementação. Paralelamente, ocorreram avanços nas pesquisas em Especificação Algébrica com métodos e linguagens para descrever tipos e garantir a sua implementação correta [GUT 77].

Os conceitos de *classificação* (para abstrair tipos a partir de itens ou instâncias), *generalização* (para abstrair novos tipos a partir de tipos dados) e *agregação* (para criar novos tipos através da composição de tipos ou objetos existentes) e as formas opostas, respectivamente, *instanciação*, *especialização* e *refinamento* surgem dos estudos na área de projeto conceitual de Banco de Dados.

Da pesquisa em modelos para *representação de conhecimento*, na área de Inteligência Artificial, também se encontram antecedentes. Os modelos de Redes Semânticas e Frames introduzem o conceito de *hierarquia de tipos* [TIC 87].

O conceito de passagem de mensagens aparece no modelo de computação por atores [HEW 77]. FLAVORS [CAN 80] tem contribuído com o conceito de herança múltipla.

Finalmente, o termo *Programação Orientada a Objetos* (POO) é introduzido com a linguagem SMALLTALK [GOL 83,GOL 84] e, entre a diversidade de ambientes e linguagens com orientação a objetos existentes, as características dessa linguagem são, em geral, as mais utilizadas para tipificar o paradigma.

3. Conceitos

A seguir são descritos vários conceitos, pretendendo caracterizar, de maneira informal, a POO. Também se apresentam as linguagens e ambientes mais representativos com suporte para este paradigma de desenvolvimento.

Esses conceitos foram sintetizados num conjunto de termos, a partir da diversa bibliografia existente [TAK 88,ROB 81,PAS 86,CAR 85,STR 88,HOR 88] de forma a oferecer, ordenadamente, uma rápida compreensão dos aspectos a serem considerados no resto do trabalho.

Linguagens de Programação Orientadas a Objetos (LPOOs):

Podemos identificar duas grandes linhas de LPOOs: as ditas puras e as ditas híbridas. As LPOOs puras seguem o paradigma proposto por Smalltalk [GOL 83] e suportam os conceitos próprios do paradigma de objetos: objetos, mensagens, classes e herança. Eiffel [MEY 88] e Trellis/Owl [O'B 87], apesar de

suas próprias características, podem-se considerar nesta categoria. As LPOOs híbridas são linguagens que introduzem modificações nas linguagens tradicionais a fim de permitir programar segundo o paradigma de objetos. Entre elas podemos citar: Objective-C [COX 88], C++ [STR 86] a partir da linguagem C, Object-Pascal [SCH 86] a partir do Pascal, Flavors [CAN 80] e Loops [BOB 82] a partir de Lisp. Nestas linguagens pode-se programar na proposta original da linguagem ou com orientação a objetos.

Objeto:

É uma entidade com capacidade de armazenar informação e manipulá-la. É expresso através de sua estrutura de dados e de seu comportamento (descrito por um conjunto de procedimentos ou *métodos*).

Método:

É a descrição do comportamento associado a um ou mais objetos, definindo um conjunto de operações a serem efetuadas sobre seus dados, no momento em que ele receber (de outro objeto) uma *mensagem* explícita, solicitando sua execução. Os componentes de um método são: sua *interface*, onde se descreve o que o método faz, e sua *implementação*, onde se descreve como o faz.

Mensagem:

É o mecanismo de comunicação entre objetos, através do qual se desencadeia a execução de um método específico. A mensagem indica o objeto destinatário, seletor do método e opcionalmente, um conjunto de argumentos. O objeto receptor determina o método a ser executado que por sua vez devolve informações ao objeto solicitante, podendo também enviar mensagens a outros objetos.

Método primitivo:

É um método que não envia mensagens; efetua processamento interno e retorna o resultado desse processamento.

Classe (ou Tipo):

Objetos podem ser agrupados em coleções de objetos similares: classes. Nas classes são descritas a estrutura de dados e o comportamento de um ou mais objetos similares que têm seus dados estruturados da mesma forma e são manipulados pelos mesmos métodos. Cada objeto é *instância* de uma classe. Os objetos representados por determinada classe diferenciam-se entre si pelo seu estado, isto é, pelos valores de seus dados. Em um sistema uniformemente orientado a objetos, uma classe é também um objeto. Desta forma a classe também pode receber mensagens que ativam métodos especiais da classe (métodos fabricantes).

Interface (ou Protocolo):

É o conjunto de mensagens às quais um objeto pode responder. A interface é definida na classe dos objetos e descreve a componente externa dos métodos dos objetos: o que os métodos fazem (nome do método e descrição dos argumentos de entrada e saída). Podem existir diversas classes de objetos com a mesma interface e estas podem ter ou não implementações iguais de seus métodos.

Metaclasse:

É uma classe especial cujas instâncias são classes. Na linguagem Smalltalk, as metaclasses são utilizadas para armazenar a informação necessária para distinguir entre mensagens enviadas à instância de uma classe daquelas enviadas a própria classe.

Instância:

É a materialização de um dos objetos descritos por uma classe. Pode existir uma classe para a qual não foi criada uma instância, mas nunca haverá uma instância que não pertença a uma classe. Todas as instâncias de uma classe usam o mesmo método para responder a um tipo particular de mensagem. Duas instâncias (objetos) diferentes podem responder de forma diferente a uma mesma mensagem, devido aos diferentes valores de suas variáveis de instância.

Variáveis de instância:

São as variáveis que formam a estrutura de dados das instâncias de classes (objetos) com valores que referenciam o estado do objeto ou referências a outros objetos.

Variáveis de classe:

São variáveis de uma classe acessíveis por todas suas instâncias.

Métodos fabricantes:

São métodos especiais das classes que possibilitam, por exemplo, a criação e inicialização de suas instâncias. A criação de uma instância se dá pelo envio à classe de uma mensagem correspondente a um método fabricante. As classes têm uma interface para estes procedimentos de maneira análoga à interface dos objetos.

Herança (ou Subclasseamento) simples:

É o mecanismo que permite definir uma nova classe a partir de uma classe já existente. A classe criada se diz *subclasse* da classe já existente. Por sua vez,

a classe já existente é chamada *superclasse* da classe criada. A subclasse herda a estrutura de dados e os métodos da superclasse, podendo adicionar variáveis na estrutura de dados herdada, bem como adicionar novos métodos e reescrever métodos herdados. Quando uma mensagem é mandada para um objeto, a procura do método correspondente começa pela classe do objeto. Se o método não for encontrado, a procura continua na sua superclasse. Uma subclasse pode ser superclasse de outra classe, podendo assim haver uma *hierarquia de classes*. As instâncias da superclasse não são afetadas pela existência de subclasses. Algumas LPOOs possuem mecanismos que permitem alterar a precedência, na procura de um método, na hierarquia das classes. Como exemplo podemos citar o mecanismo SUPER de Smalltalk que faz com que a procura de determinado método comece na superclasse do objeto.

Herança (ou Subclasseamento) múltipla:

É o mecanismo que permite a uma classe herdar as descrições de várias superclasses e, como na herança simples, adicionar métodos, variáveis à estrutura de dados, assim como reescrever métodos. A procura do método correspondente a uma mensagem começa pela classe do objeto e, se o método não for encontrado, a procura segue a ordem de uma lista de precedências entre as superclasses. Esta lista pode ser fixa ou definida pelo usuário.

Ocultamento de informação ("Information hiding"):

É a característica que descreve o potencial que algumas linguagens de programação têm para reduzir as interdependências entre módulos de software: cada módulo encobre os detalhes de implementação de seus procedimentos e os módulos se comunicam através de interfaces bem definidas. Assim, os programadores podem introduzir mudanças na implementação de um módulo sem afetar o comportamento externo a esse módulo. Dois aspectos complementares do conceito de ocultamento de informação são suportados pelas LPOOs: *encapsulamento de dados* e *abstração de dados*.

Encapsulamento de dados:

Os objetos encapsulam suas estruturas de dados. A estrutura de dados de um objeto só pode ser manipulada pelos métodos ou procedimentos do objeto. O recebimento de uma mensagem é a única forma dos objetos exercerem seu comportamento. O encapsulamento torna inacessível os detalhes da manipulação do objeto de seu meio externo.

Abstração de dados:

Uma classe de objetos é descrita abstraindo seus dados, isto é, ela fica totalmente caracterizada pelas propriedades externas de seus objetos (sua interface) e não

pela sua representação interna (a implementação de sua estrutura de dados). O conceito de abstração de dados é mais amplo que o de encapsulamento de dados: o encapsulamento localiza e oculta os detalhes de um objeto (instância); a abstração localiza e oculta os detalhes de um padrão (classe) [COH 84].

Polimorfismo:

Ao receber uma mensagem para efetuar uma operação, é o objeto quem determina como a operação deve ser efetuada pois ele tem comportamento próprio. Como a responsabilidade é do receptor e não do emissor, pode acontecer que uma mesma mensagem ative métodos ou procedimentos diferentes, dependendo da classe de objeto para onde é enviada a mensagem. Esta propriedade caracteriza o polimorfismo das LPOOs. O polimorfismo permite a criação de várias classes com interfaces idênticas e implementações diferentes, promovendo o desenvolvimento de protocolos padrões.

Protocolo padrão:

É uma interface compartilhada por várias classes de objetos. Estes protocolos facilitam o reuso de classes, minimizando o vocabulário que o projetista necessita conhecer. Da mesma forma como os matemáticos reusam os nomes das operações aritméticas para matrizes, polinômios e outros objetos algébricos, os projetistas de sistemas desenvolvidos segundo o PDO podem usar os mesmos nomes para os métodos de muitos tipos de classes.

Classe abstrata:

É uma classe especial que possivelmente não possui estrutura de dados e somente define um conjunto de métodos em termos de outros métodos não definidos na classe e que serão implementados nas suas subclasses. As classes abstratas são utilizadas como esqueletos e, nas subclasses, são completadas algumas opções reusando o código do esqueleto da classe abstrata. Dessa forma, as subclasses fornecem implementações alternativas para os métodos especificados na classe abstrata. São construídas, em geral, como raízes das hierarquias de classes e obviamente, não são classes instanciáveis.

Acoplamento tardio ("dynamic binding"):

Algumas LPOOS permitem que os endereços e parâmetros relacionados com chamadas de procedimentos sejam resolvidos em tempo de execução e não em uma compilação prévia. Assim, quando uma mensagem é enviada a um objeto, o método que deve responder a essa mensagem é identificado dinamicamente, em tempo de execução, na classe do objeto.

Base de Objetos de Software (BOS):

Uma Base de Objetos de Software (BOS) ou Biblioteca de Classes é um conjunto de classes de objetos potencialmente reutilizáveis na construção de aplicações segundo o PDO. Uma BOS possui um conjunto de classes, chamadas classes originais que proporcionam a funcionalidade básica do ambiente: os tipos de dados básicos, as estruturas de controle e os elementos do ambiente de programação. Com o desenvolvimento de novas aplicações, o projetista vai definindo novas classes que passam a fazer parte da BOS.

Esqueleto de classes ("Framework"):

É o conjunto de classes ou hierarquias de classes que compõem o projeto-solução de uma família de problemas relacionados e suportam reutilização numa granularidade maior que as classes de objetos. Distinguem-se os *esqueletos abertos* dos *esqueletos em caixa preta*. Os esqueletos abertos são criados nas primeiras épocas da história dos sistemas daquela família de problemas. Na sua reutilização se faz forte uso da herança pelo que o projetista necessita conhecer a implementação das classes. Com o tempo, essas hierarquias de classe são reorganizadas, tornando-se mais refinadas, até transformarem-se em esqueletos em caixa preta com maior capacidade de reuso por simples instanciação e, portanto, sem necessidade de conhecer a implementação de suas classes [JOH 88].

Generalização:

É o mecanismo utilizado para classificar e estruturar classes em novas classes mais gerais: do particular ao geral.

Especialização:

É o mecanismo utilizado para classificar e estruturar classes em novas classes mais especializadas: do geral ao particular. O subclasseamento ou herança é um mecanismo de construção de hierarquias de classes por especialização.

4. Exemplos

Apresentamos, a seguir, dois exemplos que ilustram os principais conceitos enunciados na seção 3.

1. Uma classe do domínio de interfaces gráficas poderia definir-se, numa LPOO hipotética, como:

   Classe: *Janela*

   Subclasse de: *Objeto_gráfico*

   Variáveis de instância: *ativa, título, superior_esq, inferior_dir, barra_reversa.*

Interface (mensagens): *mostrar, ocultar, altura:alt, largura:lg.*
Métodos:
mostrar [...implementação do método mostrar...]
ocultar [...implementação do método ocultar...]
...

A descrição da classe diz que cada instância de Janela terá sua estrutura de dados composta pelos atributos (*ativa,..., barra_reversa*) e que pode responder às mensagens (*mostrar,..., largura:lg*). A classe Janela pode ter mais atributos e responder a mais mensagens, dependendo do que tenha herdado da classe *Objeto_gráfico*. Por exemplo, a classe *Objeto_gráfico* pode definir métodos comuns a vários objetos gráficos (ex. *abrir, fechar*) que serão herdados pela classe *Janela*, permitindo que as instâncias da classe *Janela* respondam a mensagens enviados a esses métodos.

Cada atributo de *Janela* faz referência a outro objeto: *ativa* pode ser um objeto booleano, *título* um objeto string, etc.

Os objetos são instanciados, por exemplo, na linguagem Smalltalk, enviando a mensagem *new* à classe. A seguir apresenta-se a criação e dimensionamento de uma janela:

UmaJanela <— Janela new
UmaJanela altura:200 largura:400

Na classe *Janela* estamos fazendo abstração de dados e em qualquer de suas instâncias, por exemplo no objeto *UmaJanela*, temos encapsulamento de dados.

A classe *Janela* é uma especialização da classe *Objeto_gráfico*, que, por sua vez, é uma generalização da classe *Janela*.

Se a classe *Objeto_gráfico* fosse uma classe abstrata, poderia definir-se como:

Classe: *Objeto_gráfico*
Métodos:
mostrar [...esqueleto do método mostrar...]
ocultar [...esqueleto do método ocultar...]

As subclasses de *Objeto_gráfico*, como a classe *Janela*, definiriam implementações alternativas dos métodos mostrar e ocultar, a partir dos esqueletos em *Objeto_gráfico*.

2. O polimorfismo permite que diferentes classes compartilhem o mesmo protocolo, definindo diferentes métodos para executar as mensagens. Por exemplo, operações como "<" podem ser definidas em várias classes: a classe *inteiro* implementará o método de comparação inteira, enquanto a classe *string* implementará o método da ordem lexicográfica. Dessa forma, uma mensagem como,

X A:8 B:5

destinada a fazer com que o objeto referenciado por X "compare" A com B, podera ativar procedimentos diferentes, caso X seja *inteiro* ou *string*.

5. Fatores de Qualidade e Produtividade

O PDO oferece soluções para atender os requisitos de reusabilidade, manutenibilidade e confiabilidade dos produtos de software, permitindo incrementar a qualidade das aplicações e a produtividade no desenvolvimento.

Além disso, as facilidades para prototipação rápida fornecidas por este paradigma contribuem para a adequação das aplicações às necessidades do usuário, permitindo testar rapidamente requisitos especificados.

Descreve-se, a seguir, a forma pela qual o PDO dá suporte a estes aspectos.

5.1 Reusabilidade

No estilo de desenvolvimento por objetos, as aplicações são construídas, basicamente, integrando componentes de software (classes de objetos) pré-fabricados, de uso geral, selecionados desde uma BOS. A necessidade de construir novas classes, pela inexistência de componentes reutilizáveis que possam adaptar-se aos requisitos de novas aplicações, deve minimizar-se, inversamente ao crescimento e evolução da BOS.

As classes de uma BOS podem ser reutilizádas segundo diferentes facilidades disponíveis nas LPOOs [ARA 88,TSI 88]:

- instanciação: novos objetos são obtidos instanciando determinadas classes da BOS;
- herança: novas classes de objetos são criadas herdando funcionalidade de classes já existentes na BOS;
- classes abstratas ou implementação diferida: a funcionalidade das classes abstratas na BOS é especificada sem fornecer total ou parcialmente uma implementação. Novas classes podem ser criadas a partir dessas classes abstratas herdando sua funcionalidade e proporcionando implementação complementar adequada.

As técnicas de desenvolvimento por objetos devem enfatizar o desenvolvimento de classes reusáveis. Reescrever uma nova classe para torná-la mais reusável é tão importante como criar uma nova classe. Normalmente, as classes de uso geral são construídas, primeiro, para a resolução de um problema específico e posteriormente são generalizadas ao descobrir que elas têm uma aplicabilidade mais ampla. Há um problema aí: são necessários alguns desenvolvimentos (três ou mais) para conseguir generalizar uma classe. O ambiente de desenvolvimento deveria dar suporte a dois tipos de projetistas: o engenheiro

de aplicações, com a responsabilidade de gerar aplicações genéricas, e o desenvolvedor de aplicações, com a responsabilidade de gerar aplicações especificas a partir dos componentes de software gerados pelo engenheiro de aplicações.

Por outro lado, no PDO, a reutilização de software não está limitada ao código. A criação de esqueletos de classes e classes abstratas promove a reutilização da informação de projeto [JOH 88].

5.2 Confiabilidade, custos e manutenibilidade

O custo de produção de cada classe de objetos projetada é dividido entre todas as aplicações que a utilizam, com redução considerável do custo por aplicação, uma vez disponível na BOS um conjunto importante de classes reusáveis. Por outro lado, as aplicações podem ser implementadas em menor prazo, devido à redução do esforço de especificação e programação.

As aplicações desenvolvidas segundo este paradigma apresentam um maior grau de confiabilidade já que utilizam classes previamente definidas e, portanto, já testadas.

A manutenção, tanto corretiva como evolutiva, é simplificada pelo uso de componentes padronizados e pela total modularidade da arquitetura do sistema. As facilidades para a reutilização de software oferecidas pelas LPOOs também são aplicáveis à fase de manutenção [JOH 88]:

- a abstração de dados permite localizar o efeito das mudanças nos requisitos do sistema nas classes de objetos a serem modificadas;

- o polimorfismo reduz o número de métodos ou procedimentos que o projetista de manutenção deve compreender;

- o mecanismo de herança permite que novas versões de um programa possam ser construídas sem afetar as antigas.

5.3 Facilidades para construção de protótipos rápidos

As LPOOs são potencialmente úteis para a construção de protótipos rápidos. As linguagens que suportam acoplamento tardio permitem associação dinámica de mensagens a métodos. Isto favorece à prototipação. Além disso, a maioria dos métodos não necessitam alterações ou recompilações a medida que o protótipo evolui. Por outro lado, a reutilização de classes economiza o tempo de desenvolvimento do protótipo. Assim, é possível colocar rapidamente à disposição do usuário um sistema funcionalmente correto para uso experimental e testar realisticamente os requisitos especificados, tornando possível convergir mais rápidamente a um produto que satisfaça as necessidades do usuário.

6. O Estilo de Desenvolvimento

O PDO baseia a decomposição do software nas classes de objetos que o sistema manipula.

O desenvolvimento de sistemas baseado em decomposição funcional constrói módulos segundo as operações do sistema e distribui as estruturas de dados entre os módulos resultantes. No PDO se faz o contrário: as estruturas de dados mais importantes são a base da modularização e as operações são colocadas naquelas estruturas de dados por elas manipuladas. Para [MEY 87] isto permite desenvolver sistemas com maior vida útil, isto é, altamente manuteníveis, pois as categorias de objetos que interagem num sistema são mais permanentes que a sua funcionalidade.

O estilo de desenvolvimento por objetos está baseado nos conceitos de classe, hierarquia de classes e herança.

O conceito de herança ou subclasseamento proporciona um critério adequado para a modelagem e modularização dos objetos do domínio da aplicação. A idéia chave do subclasseamento é que uma aplicação pode ser modelada construindo como classes os conceitos mais gerais do domínio da aplicação para logo tratar os casos mais particulares como subclasses. Sucessivos passos de refinamento introduzem e descrevem conceitos mais especializados. Em cada passo é considerada a informação relevante a esse nível de abstração. O resultado é um modelo de objetos constituído por uma ou mais hierarquias de classe.

Essa metodologia, apesar de ser denominada por alguns autores *refinamento por especialização* [HAL 87,BOR 85], na realidade, não é estritamente "top-down". Nem sempre é modelada uma hieraquia de classes começando com uma superclasse e após construindo as subclasses. Pelo contrário, frequentemente é aplicado o mecanismo de generalização: são criadas várias classes independentes; depois, ao notar que elas estão relacionadas, fatoram-se suas características comuns em uma ou mais superclasses. É melhor uma metodologia "middle-out" [MEY 87]: várias fases "up" (do particular ao geral) e "down" (do geral ao particular) são requeridas para produzir um projeto correto e completo.

O processo de refinamento deve realizar-se conjuntamente com o acesso a uma BOS de forma a selecionar e recuperar (através de alguns dos mecanismos indicados na seção 5.1) classes que possam adaptar-se à aplicação.

Por outro lado, a técnica de desenvolvimento deve enfatizar o projeto de hierarquias com classes potencialmente reutilizáveis, definidas num alto nível de abstração (por exemplo, definindo classes abstratas como raízes das hierarquias).

Outra característica importante deste estilo de desenvolvimento é a tênue distinção entre as fases de projeto e implementação. A única diferença entre as duas fases e o nível de abstração considerado: alguns detalhes no projeto são postergados à fase de implementação. Além disso, a implementação de novas classes tende a minimizar-se e a atividade principal se concentra na fase de projeto, com a seleção e adaptação de classes previamente implementadas.

7. Metodologias de Desenvolvimento e Limitações práticas

Ambientes de desenvolvimento com suporte para o PDO já estão disponíveis comercialmente: *Smalltaltk-80* [GOL 84], *Eiffel* [MEY 88], C^{++} [STR 86]. Outros, como *Trellis* [O'B 87] e *Ithaca* [PRO 89] são importantes protótipos de pesquisa. Eles são compostos, basicamente, de uma LPOO, um suporte de execução, uma BOS (com, pelo menos, um conjunto de classes básicas) e ferramentas de apoio ao desenvolvimento e manutenção de aplicações.

Entretanto, ainda existem dois grandes problemas, alvo das pesquisas mais recentes. Um deles aponta à necessidade de definir metodologias de desenvolvimento que habilitem a aplicabilidade do paradigma a sistemas de grande porte. O outro problema está relacionado com a necessidade de criar ferramentas de apoio que viabilizem a construção de aplicações a partir de um conjunto de classes predefinidas de forma a permitir atingir a produtividade proclamada para este estilo de desenvolvimento.

7.1 Metodologias

Considerando os fatores de qualidade e produtividade acima mencionados, dispor de metodologias de desenvolvimento segundo o PDO, compreensíveis, rigorosas e bem definidas é de especial interesse para a Engenharia de Software.

Em geral, as abordagens principais estão concentradas nas fases de projeto e implementação, com aplicabilidade somente a sistemas de médio e pequeno porte [HAL 87,BOO 86].

Existem algumas abordagens e sugestões que procuram combinar técnicas de desenvolvimento por objetos com diversos métodos clássicos de análise de requisitos: *Análise Estruturada* [ROT 87], *SADT* [BOO 84], *SREM* [BOO 86], *JSD* [MAS 88], *Redes de Petri* [MOR 88,BRU 86]. Estas propostas fornecem heurísticas para o mapeamento da especificação de requisitos, produto da aplicação de alguma das metodologias mencionadas, numa especificação de classes, hierarquias de classe e trocas de mensagens.

Entre essas abordagens podemos citar a [ROT 87] que realiza um importante estudo, apresentando um conjunto de guias e heurísticas a fim de mapear a representação da especificação de requisitos de um Diagrama de Fluxo de Dados (DFD) a um Modelo de Objetos. Em síntese, o método de transformação consiste em:

1. Identificam-se os objetos: os depósitos de dados que aparecem nos DFDs são candidatos a darem origem a objetos;

2. Identificam-se as operações associadas a cada objeto. Para isso se analisam os processos e se identifica a quais objetos potenciais a associação será mais natural, considerando que:

- alguns candidatos a objetos não terão operações que os credenciem a se tornarem objetos. Nesse caso, incorporá-los a outros candidatos a objetos;
- alguns processos correspondem-se com mais de um objeto. Nesse caso, deve-se decompor esses processos;
- haverá processos sem nenhum candidato a objeto. Nesse caso, cria-se um objeto para cada um destes processos.

3. Representam-se os objetos identificados, operações associadas a Entidades Externas, e ajustam-se os DFDs às modificações introduzidas.

4. Definem-se as interfaces dos objetos.

5. Fazer avaliação final.

Estas técnicas têm viabilizado a aplicabilidade do PDO a projetos de grande porte. Porém, na utilização dessas metodologias não se leva em consideração uma das grandes vantagens do paradigma de objetos: a proximidade entre o modelo de objetos do domínio da aplicação e o modelo computacional correspondente.

Por outra parte, no processo de mapeamento podem perder-se certas apreciações da realidade, pois estar-se-á forçando uma estrutura para o modelo de objetos que não terá sempre a correspondência mais natural com a realidade. Isto leva, às vezes, a definir estruturas para o modelo computacional radicalmente diferentes das estruturas do domínio da aplicação [HAL 87].

Considerando a uniformidade conceitual entre o mundo da aplicação e o mundo computacional segundo o paradigma de objetos (a "realidade" é composta de objetos e só objetos que interagem entre si), é de interesse o desenvolvimento de metodologias para especificação de requisitos que permitam representar diretamente o domínio da aplicação num modelo de objetos. Dessa forma, as estruturas do modelo computacional serão mais parecidas às estruturas do mundo real.

Em [GIR 89a] foram abordados estes problemas e apresentada uma metodologia de desenvolvimento onde todos os produtos do processo de desenvolvimento são representados por modelos de objetos, desde a fase de especificação de requisitos até a fase de implementação. Também, recentemente, têm surgido propostas importantes para análise de requisitos [BAI 89], [SHL 89], [KUR 89] que consideram a problemática mencionada. A seguir, descreve-se uma síntese da metodologia proposta: o ciclo de vida adotado e os métodos a serem aplicados em cada uma de suas fases.

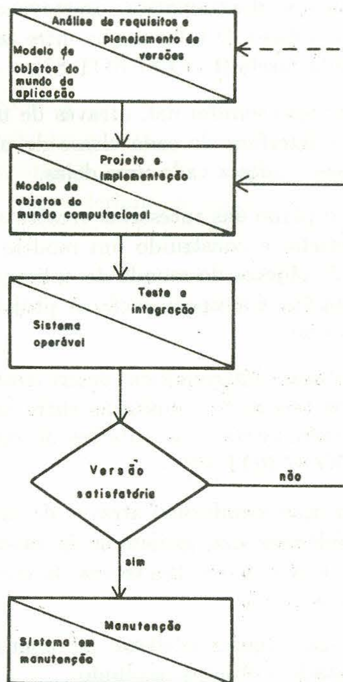
7.1.1 Uma Metodologia de Desenvolvimento Orientada a Objetos

A metodologia proposta tem sido influenciada, fundamentalmente, pelos conceitos de modelagem conceitual de sistemas [BOR 85], pelas abordagens de

[ROT 87,JAC 87,MAT 88] assim como dos estudos sobre projeto orientado a objetos de [HAL 87,JOH 88]. No anexo é apresentado um exemplo de utilização da metodologia.

7.1.1.1 O Ciclo de Vida

O ciclo de vida adotado, apresentado na figura 1, é uma adaptação do ciclo de versões sucessivas apresentado em [FAI 85]. Cada versão é um protótipo do sistema que é refinado em sucessivos incrementos de funcionalidade.



Referência:

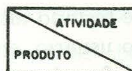


Figura 1: O Ciclo de Vida

Na fase de *análise de requisitos*, o mundo da aplicação é modelado em termos de suas entidades e tarefas. Uma *entidade* é um objeto do mundo da aplicação sobre o qual o sistema necessita produzir ou usar informação. Uma *tarefa* é um conjunto de ações que visa a obtenção de determinado produto ou resultado.

Nesta fase também é planejada a versão a prototipar, podendo ser um protótipo para estabelecer uma conveniente interface com o usuário; um protótipo que permita esclarecer requisitos da aplicação ou um protótipo que permita avaliar o desempenho do sistema em áreas críticas.

O produto desta fase é a especificação de requisitos, um modelo de objetos do domínio da aplicação que contém:

- uma apresentação diagramática, descrevendo as classes de objetos, suas hierarquias e o fluxo de informação entre as classes de objetos que participam em cada tarefa [LOO 87,ROT 87];
- uma apresentação semiformal, através de um dicionário, descrevendo os atributos e a interface de cada classe de objetos, as tarefas e fluxos de mensagens associados a cada uma delas.

Também é obtido o plano das sucessivas versões a serem construídas.

Na fase de *projeto*, é construído um modelo computacional de objetos a partir do modelo de objetos do mundo da aplicação e do plano do protótipo.

O produto desta fase é a especificação de projeto, um modelo computacional de objetos que contém:

- uma apresentação diagramática, descrevendo as classes de objetos, suas hieraquias, as trocas de mensagens entre as classes de objetos que participam de cada tarefa e os critérios de reusabilidade a partir da BOS [CUN 86,LOO 87,ROT 87];
- uma apresentação semiformal através de um dicionário, descrevendo interface e implementação projetada de cada classe de objetos, variáveis de instância e descrições das trocas de mensagens entre as classes que compõem cada tarefa.

Na *implementação* são criadas as classes de objetos projetadas, no ambiente de programação orientado a objetos escolhido.

As fases de projeto e implementação deverão ter forte participação do grupo responsável da BOS, que projetará, implementará e testará novas classes de objetos segundo critérios adequados de reusabilidade para sua inclusão na BOS.

Na fase de *teste e integração* é testada individualmente cada nova classe criada; logo, as classes são integradas e testadas em função das tarefas do sistema e finalmente, o protótipo é integrado e testado integralmente.

Se a versão satifaz os requisitos do usuário, em sua totalidade, o sistema é liberado para operação, começando sua fase de manutenção. Em caso contrário, começa uma nova fase de projeto onde o protótipo é refinado ou incrementado em alguns de seus aspectos funcionais ou operacionais.

As linhas pontilhadas da figura 1 indicam a necessidade de uma maior análise do mundo da aplicação.

A fase de *manutenção* compreende os mesmos passos e métodos que o ciclo de desenvolvimento, para os requisitos de extensão e/ou modificação da funcionalidade do sistema.

7.1.1.2 Os métodos em cada fase

A seguir é apresentado um conjunto de guias e heurísticas a serem aplicadas em cada uma das fases do ciclo de desenvolvimento a fim de construir os modelos de objetos associados a cada uma dessas fases.

Análise de requisitos

1. Identificar entidades e tarefas.

   O analista analisa o mundo da aplicação e considera todas as pessoas e coisas que poderiam ser consideradas como entidades e todas as operações que afetam essas entidades que poderiam ser consideradas tarefas. Também são identificados os atributos de cada entidade. Com essa informação se elabora:

   - uma lista de entidades
   - uma lista de tarefas

   As listas são depuradas com as entidades e tarefas relevantes ao problema segundo os seguintes critérios [ROT 87]:

   - uma entidade será mantida na lista caso o sistema em desenvolvimento necessite produzir ou usar informações sobre ela;
   - as entidades que não têm autonomia e só são relevantes como parte de outras entidades passam a ser atributos destas;

   As entidades identificadas dão origem às classes de objetos do sistema, fatorando as características comuns dos objetos que representam. Os atributos comuns dessas entidades constituem-se na estrutura de dados (variáveis de instância) da classe correspondente. Por exemplo, as entidades João e Pedro, com vários atributos comuns, como nome, idade, endereço, etc podem dar origem à classe Pessoa.

2. Decompor as tarefas e associar os métodos obtidos às classes de objetos correspondentes. Identificar o possível surgimento de novas classes.

   As tarefas podem afetar várias classes de objetos, isto é, as ações ou procedimentos que compõem uma tarefa podem ser sofridas ou realizadas por diferentes objetos. Exemplo: Um livro é emprestado a um aluno e o aluno recebe um livro em empréstimo. Nestes casos, a tarefa deve ser

decomposta em pacotes de operações indivisíveis (métodos) que possam ser acomodadas nas classes de objetos envolvidos.

Com essa informação, prepara-se uma lista das classes com seus correspondentes métodos.

No processo de associar ações ou procedimentos a classes de objetos pode acontecer que determinada tarefa, ou conjunto de procedimentos, capture em si mesma um conceito abstrato do domínio da aplicação e, em conseqtência, torne mais conveniente modelar essa tarefa como uma classe de objetos independente. Exemplo: Ao modelar a matrícula de um estudante num curso as possíveis alternativas são:

(a) atribuir a tarefa-ação à classe estudante ou à classe curso;

(b) criar uma nova classe matrícula que abstraia a informação dos cursos que cada aluno pretende assistir.

A segunda alternativa pode ser mais adequada se for desejado capturar alguma informação adicional, tal como o conceito obtido pelo estudante no curso.

Não é tarefa fácil decidir se um procedimento será implementado como método de uma classe ou como uma classe independente. As seguintes guias [JOH 88] podem ajudar na decisão de criar uma classe independente:

- o procedimento é em si mesmo uma abstração significativa;
- o procedimento será compartilhado por várias classes;
- o procedimento é complexo;
- o procedimento faz pouco uso da representação de seus operandos;
- o procedimento, como método de uma classe particular, é pouco utilizado.

Por outro lado, não é simples estabelecer regras que indiquem em que classe implementar o procedimento. Procedimentos com vários argumentos podem ser implementados como métodos em qualquer classe que possua na sua estrutura de dados algum desses argumentos.

Com as considerações mencionadas, se sugere particionar a atividade a realizar nesta etapa, nas subatividades a seguir, de forma repetitiva até obter os métodos associados a cada classe de objetos:

(a) Decompor as tarefas;

(b) Identificar o surgimento de novas classes;

(c) Associar os procedimentos identificados às classes de objetos.

3. Descrever, num dicionário, a funcionalidade dos métodos de cada classe de objetos.

4. Verificar a existência de atributos e/ou procedimentos comuns a mais de uma classe de objetos, definindo uma hierarquia entre elas de acordo com o seguinte critério [MAT 88]:

   - existindo uma intersecção dos conjuntos de atributos e/ou procedimentos de mais de uma classe, deve ser definida uma superclasse que as generalize, passando as classes originais a serem subclasses desta, herdando as características comuns;

   - existindo uma classe de objetos cujo conjunto de atributos e/ou procedimentos seja subconjunto dos atributos e/ou operações de uma outra classe qualquer, deve ser definida uma hierarquia entre elas de tal forma que a segunda classe assuma o papel de subclasse da primeira, herdando as características comuns.

5. Representa-se, num diagrama, o modelo de objetos do mundo da aplicação com o fluxo de informação entre cada um dos procedimentos-classe identificados.

Projeto

1. Completar o modelo de objetos obtido na especificação de requisitos com a descrição das classes de objetos próprias do mundo computacional (exemplo: janelas) e introduzir as modificações necessárias para satisfazer requisitos de comportamento (tempo de resposta, confiabilidade, eficiência, etc).

2. Para cada classe:

   - verificar se já existe na BOS, algum esqueleto de classe (classes ou hierarquias de classe) com funcionalidade similar que possa ser reutilizado;

   - caso exista, se indicará a classe a ser reusada e os critérios de reutilização (seção 5.1);

   - caso não exista, projetam-se os procedimentos (métodos) da classe especificando a sua lógica.

3. Encontrar protocolos padrão.

   Como foi apresentado na seção 3, é importante a identificação de protocolos padrão, isto é, várias classes com interfaces similares, de forma a reduzir a informação que o projetista necessita conhecer para a tarefa de reuso de software.

4. Refinar o modelo de objetos obtido até o passo (3), isto é, modificar classes e hierarquias, atendendo aos seguintes critérios [HAL 87]:

- reusabilidade: se um determinado comportamento (procedimento) pode ser compartilhado ou reusado por várias classes, será de maior utilidade associar esse procedimento (método) a uma superclasse de forma que aquelas classes possam reusar o procedimento;
- complexidade: se a implementação de um determinado método é muito complexa, pode ser conveniente associar esse método a uma outra classe, especial, a fim de introduzir simplicidade na classe original;
- aplicabilidade: deve considerar-se a aplicabilidade geral do método. Quanto mais especializado é um comportamento, mais adequada sua colocação numa classe especializada;

e as seguintes sugestões [JOH 88]:

- dividir classes com um número muito grande de métodos em classes disjuntas, pois, geralmente, elas representam mais de uma abstração;
- dividir classes onde subconjuntos disjuntos de métodos acessam subconjuntos disjuntos de variáveis de instância;
- desenvolver hierarquias de classes profundas e estreitas, pois, vários níveis de generalidade/especialidade facilitarão a reutilização de classes;

5. Representam-se, num diagrama, o modelo computacional de objetos, com as trocas de mensagens.

Implementação

São criadas as classes de objetos (variáveis de instância e métodos) a partir da especificação de projeto e da BOS no ambiente de programação com orientação a objetos escolhido.

Integração e teste

1. Depurar cada uma das novas classes e aquelas modificadas a partir da BOS [MAT 88]:
   - substituir as mensagens enviadas a outras classes por uma mensagem externa (exibição para o depurador), e pela inicialização dos parâmetros que estariam na mensagem de volta;
   - ativar cada um dos procedimentos da classe através do envio de mensagens e verificar os resultados obtidos.
2. Para cada tarefa identificada:
   - as classes de objetos que implementam cada tarefa são integradas e testadas para assegurar que a tarefa satisfaz os requisitos do usuário;

- são integradas e testadas conjuntamente as classes de objetos que compõem a totalidade das tarefas do sistema.

7.2 Ferramentas

A aplicabilidade do PDO junto com o aproveitamento prático de sua potencialidade estão limitados à disponibilidade de ferramentas adequadas que facilitem as tarefas de construção, reutilização e integração de classes de objetos a partir de uma BOS [GIR 89b,ARA 88]. Mais explicitamente, é necessário estabelecer critérios e construir ferramentas de apoio que permitam:

- classificar, armazenar, recuperar, adaptar e documentar as classes de objetos da BOS;
- gerenciar a evolução da BOS de forma a manter a sua consistência (modificações na interface de uma classe invalidam referências entre classes de objetos) e garantir o princípio "darwiniano" de sobrevivência das classes que melhor se adaptam ao desenvolvimento de novas aplicações;
- selecionar da BOS classes adequadas para a aplicação em desenvolvimento. O acesso por palavras chaves não parece suficiente: o projetista, normalmente, somente dispõe de um critério aproximado de seleção. Por outro lado, os objetos estão encapsulados e existe pouca informação disponível de seu conteúdo e comportamento. São necessárias ferramentas para "browsing" que permitam, a partir de um determinado critério de seleção, visualizar hierarquias de classe, exemplos de utilização, interfaces, implementações, classes relacionadas, etc;
- apoiar o desenvolvimento de novas aplicações, facilitando o processo de decomposição, projetando-as a partir de classes pré-fabricadas e promovendo o desenvolvimento de classes com capacidade de serem reutilizadas em outras aplicações;
- definir os mecanismos de configuração e controle de versões das aplicações desenvolvidas segundo este paradigma.

8. Conclusões e Perspectivas

Foi caracterizado o PDO e suas contribuições principais para Engenharia de Software. Também foram apresentadas metododologias de desenvolvimento e as limitações existentes na aplicação deste estilo de desenvolvimento, na construção de sistemas de grande porte e com bibliotecas contendo um grande número de classes.

"...Quando construimos um novo sistema, deveríamos ordenar componentes desde (esses) catálogos e combiná-los, em lugar de re-inventar a roda cada vez..."

[MEY 87]. O PDO oferece solução a esse problema, através de sua capacidade para a reutilização de componentes. Reduz tanto o tempo de desenvolvimento como o custo de manutenção, simplificando a criação de novos sistemas e de novas versões de sistemas já existentes.

Por outro lado, em seu atual estágio, este estilo de desenvolvimento não é a panacéia para os problemas de desenvolvimento de software, principalmente porque:

- os componentes de software devem ser projetados considerando sua futura capacidade de reuso e isto não é tarefa simples porque requer experiência do domínio da aplicação para poder generalizar conceitos que permitam uma modelagem adequada de classes e operações associadas;

- são necessárias ferramentas de apoio ao processo de reuso na atividade de projeto (classificação, selecão e integração) e isto é fundamental porque se os projetistas não tiverem acesso adequado à BOS não poderão tirar o máximo proveito das potencialidades do paradigma;

- deve-se introduzir rigor às metodologias existentes e desenvolver técnicas de análise de requisitos mais orientadas ao mundo da aplicação. É de especial interesse a formalização dos princípios da POO. Em [LAD 88] é sugerida a utilização do Método de Desenvolvimento de Viena (VDM) para introduzir rigorosidade na POO já que existem analogias entre uma especificação VDM e os princípios da POO. A descrição de um sistema usando VDM tem dois componentes principais: a descrição de alguns tipos de dados que compreendem o estado interno do sistema e definição de operaç ões que são usadas para manipular esse estado;

- finalmente, é necessário superar a força da cultura tradicional da programação, que produz resistências na forma de conceber e realizar sistemas de software.

Referências

[ARA 88] ARAPIS, C. & KAPPEL, G. Organizing Objects in an Object Software Base. In: TSICHRITZIS, Dennis C. ed. Active Object Environments, Genève, Centre Universitaire d'Informatique / Université de Genève, Juin 1988, p.32-50.

[BAI 89] BAILIN, Sidney. An Object-Oriented Requirements Specification Method. Communications of the ACM, New York, 32(5): 608-623, May. 1989.

[BOB 82] BOBROW, D.G. & SEFIK, M.J. Loops: An Object- Oriented Programming System for Interlisp, Xerox PARC, Palo Alto, 1982.

[BOO 84] BOOCH, Grady. Object-Oriented Design. In: Tutorial on Software Design Techniques, Silver Spring, IEEE Computer Society Press, 1984, p. 420-436.

[BOO 86] BOOCH, Grady. Object-Oriented Development. IEEE Transactions on Software Engineering, New York, 12(2): 211-221, Feb. 1986.

[BOR 85] BORGIDA, Alexander et alii. Knowledge Representation as the Basis for Requirements Specification, Computer, Los Alamitos, 18(4):82-91, Apr. 1985.

[BRU 86] BRUNO, G. & BALSAMO, A. Petri Net-based Object-oriented Modelling of Distributed Systems. In: Object-Oriented Programming Systems, Languages and Applications (OOPSLA' 86), Portland, Sep. 29 - Oct. 2, 1986. Proceedings. Publicado em SIGPLAN Notices, New York, v. 21, n. 11, Nov. 1986.

[CAN 80] CANNON, J. I. Flavors, MIT Artificial Intelligence Lab, Cambridge, 1980.

[CAR 85] CARDELLI, Luca & WEGNER, Peter. On Underestanding Types, Data Abstraction and Polymorphism. ACM Computing Surveys, 17(4):471-522, Dec. 1985.

[COH 84] COHEN, A. Toni. Data abstraction, data encapsulation and Object-Oriented Programming, SIGPLAN Notices, 19(1): 31-35, Jan. 1984.

[COX 88] COX, B. Objective-C: The Need for Specification and Testing Languages. Journal of Object-Oriented Programming, New York, 1(2): 44-7, Jun/Jul. 1988.

[CUN 86] CUNNINGHAM, Ward & BECK, Kent. A Diagram for Object-Oriented-Programs. In: Object-Oriented Programming Systems Languages and Applications (OOPSLA' 86), Portland, Sept. 29 - Oct. 2, 1986. Proceedings. Publicado em SIGPLAN Notices, New York, 21(11): 361-367, Nov. 1986.

[DAH 66] DAHL, O. & NYGAARD, K. Simula: an Algol-Based Simulation Language, ACM Communications, 9(9):671-78, Set. 1966.

[FAI 85] FAIRLEY, R. Software Engineering Concepts, New York, Mc.Graw Hill, 1985.

[GIR 89a] GIRARDI, María del Rosario. Proposta de uma Metodologia de Desenvolvimento de Software Orientada a Objetos, Porto Alegre, PGCC, 1989.

[GIR 89b] GIRARDI, María del Rosario. Uma Ferramenta de Apoio ao Desenvolvimento de Software segundo o Paradigma de Objetos, Dissertação de Mestrado, PGCC, 1989 (em andamento).

[GOL 83] GOLDBERG, A. & ROBSON, D. Smalltalk-80: The Language and Its Implementation, Reading, Addison - Wesley, 1983.

[GOL 84] GOLDBERG, A. Smalltalk-80: The Interactive Programming Environment, Reading, Addison - Wesley, 1984.

[GUT 77] GUTTAG,J. Abstract Data Types and the Development of Data Structures, Communications of the ACM, 20(6):396-404, Jun. 77.

[HAL 87] HALBERT, D. & O' BRIEN, P.D. Using Types and Inheritance in Object-Oriented Programming, IEEE Software, 4(5): 71-79, Sep. 1987.

[HEW 77] HEWITT, C. Viewing Control Structures as Patterns of Passing Messages, Artificial Intelligence, 8(3):323-364, 1977.

[HOR 88] HORN, Bruce L. An Introduction to Object-Oriented Programming, Inheritance and Method Combination, Relatório de Pesquisa, Carnegie Mellon University, Jan. 1988.

[JAC 87] JACOBSON, Ivar. Object-Oriented Development in an Industrial Environment. In: Object-Oriented Programming Systems Languages and Applications (OOPSLA' 87), Oct 4-8, 1987. Proceedings. Publicado em SIGPLAN Notices, New York, 22(12): 183-191, Dec. 1987.

[JOH 88] JOHNSON, Ralph E. & FOOTE, Brian. Designing Reusable Classes. Journal of Object-Oriented Programming, New York, 1(2):22-35, Jun./Jul. 1988.

[JON 87] JONATHAN, Miguel. Orientação a Objetos em Smalltalk-80: uma abordagem eficaz para a construção de sistemas de software. In: I Simpósio Brasileiro de Engenharia de Software, Petrópolis, 22-23 Outubro, 1987. Anais. p. 32-44.

[KUR 89] KURTZ, Barry D. et alii. An Object-Oriented Methodology for System Analysis and Specification, Hewlett-Packard Journal, p.86-90, Apr. 1989.

[LAD 88] LADDEN, Richard M. A Survey of Issues to be considered in the Development of an Object-Oriented Development for ADA. Software Engineering Notes, New York, 13(3):24-30, Jul. 1988.

[LOO 87] LOOMIS, M.E.S. & SHAH, A. V. & RUMBAUGH, J.E. An Object Modeling Technique for Conceptual Design. In: European Conference on Object-Oriented Programming, Paris, June 15-17, 1987. Proceedings. Publicado em Lecture Notes in Computer Science, Berlin, Springer-Verlag, 1987. S. 276, p. 192-202.

[MAT 88] MATTOSO, Adriana & BLUM, Hélcio. Proposta de Desenvolvimento de Software com Orientação a Objetos. In: II Simpósio Brasileiro de Engenharia de Software, Canela, 27-28 Outubro, 1988. Anais. p.7-15.

[MAS 88] MASIERO, P & GERMANO, F. S. R. JSD as an Object- Oriented Design Method. Software Engineering Notes, New York, 13(3):22-23, Jul. 1988.

[MEY 87] MEYER, Bertrand. Reusability: The Case for Object-Oriented Design. IEEE Software, Los Alamitos,4(2): 50-63, Mar. 1987

[MEY 88] MEYER, B. Object-Oriented Software Construction, New York, Prentice-Hall, 1988.

[MOR 88] MOREIRA, Roberto Normandia. Programação Orientada a Objetos para Aplicações em Tempo Real, Porto Alegre, PGCC da UFRGS, 1988.

[O'B 87] O'BRIEN, Patrick D. et alii. The Trellis Programming Environment, In: Object-Oriented Programming Systems Languages and Applications (OOPSLA' 87), Oct 4-8, 1987. Proceedings. Publicado em SIGPLAN Notices, New York, v.22, n.12, Dec. 1987, p. 91-102.

[PAS 86] PASCOE, Geoffrey A. Elements of Object-Oriented Programming. Byte, Peterborough, 11(8): 139- 144, Aug. 1986.

[PRO 89] PROFROCK, A. et alii. ITHACA: An Integrated Toolkit for Highly Advanced Computer Applications. In: OBJECT-ORIENTED DEVELOPMENT, TSICHRITZIS Dennis C. ed., Genève, Centre Universitaire d' Informatique / Université de Genève, 1989, p. 321-44.

[ROB 81] ROBSON, David. Object-Oriented Systems. Byte, Peterborough, 6(8): 74-86, Aug. 1981.

[ROT 87] ROTENBERG, Hélio B. Programação Orientada a Objetos: Um Enfoque da Engenharia de Software, Dissertação de Mestrado, PUC/RJ, 1987.

[SCH 86] SCHMUCKER, K. Object-Oriented Languages for the Macintosh. Byte, 11(8): 177-185, Ago. 1986.

[SHL 89] SHLAER, S. & MELLOR, S. J. An Object-Oriented Approach to Domain Analysis. Software Engineering Notes, 14(5):66-77, Jul. 89.

[STR 86] STROUSTRUP, B. The C++ Programming Language, Addison Wesley, Menlo Park, 1986.

[STR 88] STROUSTRUP, B. What is "Object-Oriented Programming"?, IEEE Software, Los Alamitos, 5(3):10-20, May. 1988.

[TAK 88] TAKAHASHI, Tadao. Introdução à Programação Orientada a Objetos, Curitiba, III EBAI, 1988.

[TIC 87] TICHY, W. What Can Software Engineerings Learn from Artificial Intelligence?, Computer, 20(11):43-54, Nov. 1987.

[TSI 88] TSICHRITZIS, D. & NIERSTRASZ, O. Application Development Using Objects. In: TSICHRITZIS Dennis C. ed. Active Object Environments, Genève, Centre Universitaire d' Informatique / Université de Genève, Juin 1988, p. 18-31.

ANEXO

Um exemplo de Utilização da Metodologia Proposta

A seguir são apresentados os modelos de objetos resultantes da aplicação da metodologia proposta na análise e projeto de um sistema de biblioteca de livros e periódicos. A aplicação dos métodos propostos em cada uma das fases é descrita, em detalhe, em [GIR 89a].

Na figura A-1 é apresentada a notação gráfica utilizada nos modelos de objetos (adaptada de [ROT 87]).

A figura A-2 mostra o modelo de objetos do domínio da aplicação, resultado da fase de análise de requisitos.

A figura A-3 mostra o modelo de objetos do mundo computacional, resultado da fase de projeto.

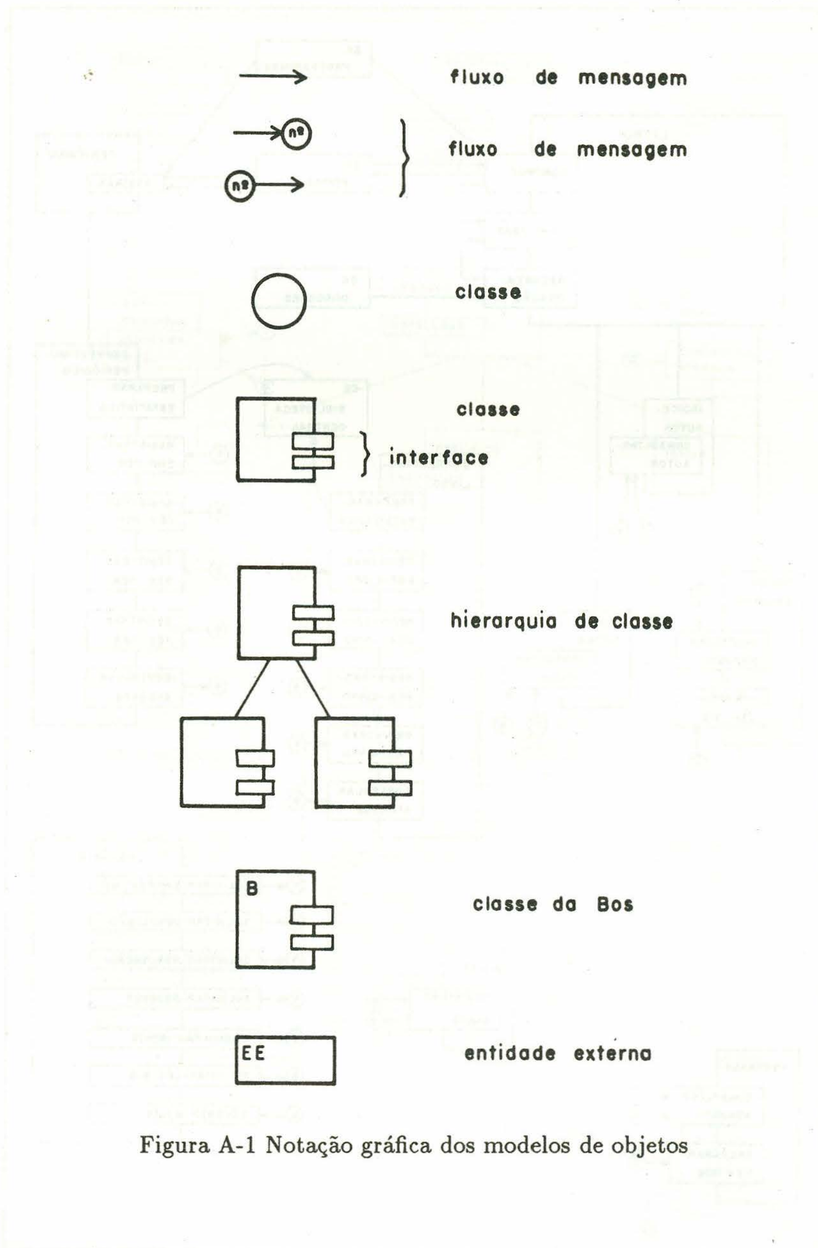


Figura A-1 Notação gráfica dos modelos de objetos

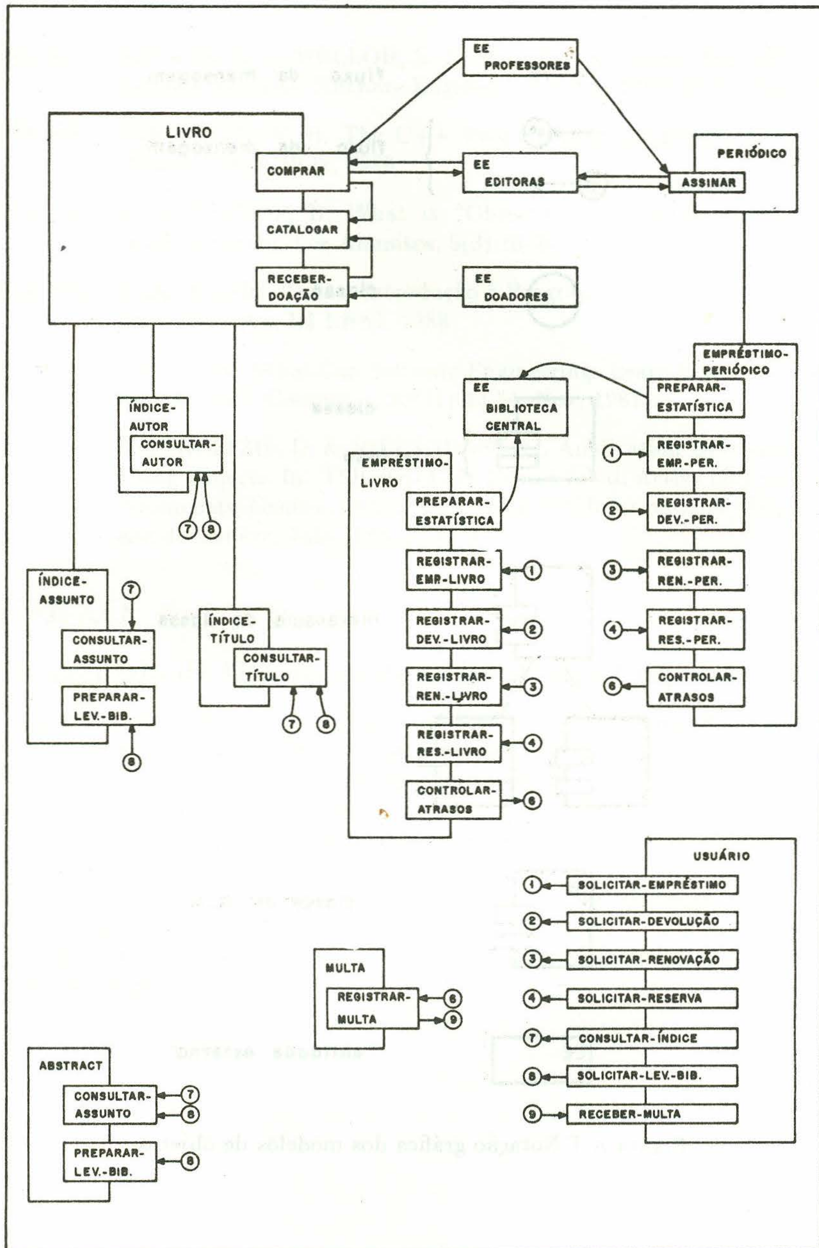


Figura A-2 Modelo de objetos do domínio da aplicação

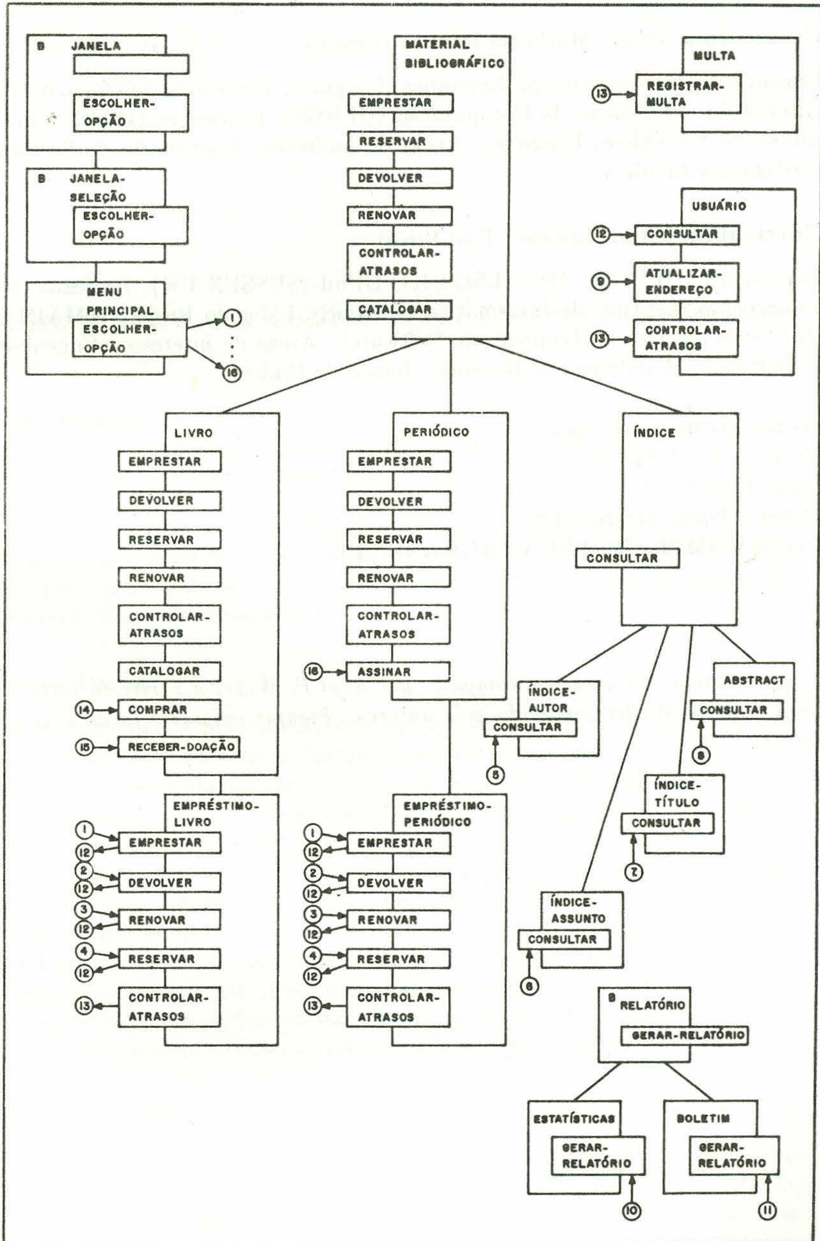


Figura A-3 Modelo de objetos do mundo computacional

Curriculum vitae (María del Rosario Girardi)

Engenheira (Universidade da República, Uruguai); Mestranda do Curso de Pós-Graduação em Ciência da Computação (UFRGS). Professora (INCO, Universidade da República, Uruguai). Áreas de interesse: Engenharia de Software, Inteligência Artificial.

Curriculum vitae (Roberto Tom Price)

Engenheiro (UFRGS), MSc (LSE-UK), DPhil (SUSSEX-UK). Professor Pesquisador do Instituto de Informática (UFRGS). Líder do Projeto AMADEUS (Ambiente de Desenvolvimento de Software). Áreas de interesse: Engenharia de Software, Modelagem de Sistemas, Banco de Dados.

Endereço dos autores:
CPGCC/II - UFRGS
Caixa Postal 1501
90.001 - Porto Alegre - RS
bitnet: TOMPRICE@SBU.UFRGS.ANRS.BR

Convertido, formatado e composto por Raul F. Weber a partir do texto original em Chi-Writer fornecido pelo autores. Figuras originais fornecidas pelo autores.