Considerações sobre Projeto Orientado a Objetos: Manipulação Formal das Técnicas

Francisco Reinaldo[1,2], Marcus Siqueira[1,2]

[1] FEUP - Faculdade de Engenharia da Universidade do Porto
LIACC - Laboratório de Inteligência Artificial e Ciência de Computadores
Rua Dr. Roberto Frias, sn, 4200-465
Porto, Portugal
reifeup@fe.up.pt, marvinsiq@gmail.com
[2] UnilesteMG - Centro Universitário do Leste de Minas Gerais
GIC - Grupo de Inteligência Computacional
Av. Presidente Tancredo Neves n. 3500, Bairro Universitário, CEP 35170-056
Coronel Fabriciano, MG, Brasil

Resumo

Este artigo apresenta e comenta as principais técnicas que estão sendo utilizadas para o desenvolvimento de esquemas(frameworks). Um esquema define o comportamento de uma coleção de objetos, provendo uma nova maneira para reusa-lo. Através da reutilização de código, da definição dos tipos de esquemas e do procedimento necessário para desenvolvimento de um projeto, poderemos construir um sistema mais robusto, flexível e extensível.
Palavras-chave: engenharia de programas, esquemas, padrão de projeto.

Abstract

This article presents the main usable techniques to develop frameworks. Framework appoints the basic behaviours in a object collection to be reusable. In addition, the reuse of code, definition of many kinds of frameworks and the necessary steps to develop the project will design a robust, flexible and extensible system.
Key-words: software engineering, frameworks, design patterns.

1 Introdução

A Orientação a Objetos (OO) é um paradigma de programação que oferece técnicas e formas para representar soluções vindas de problemas do mundo real em um sistema. Para isso utiliza-se de uma coleção de unidades individuais de programa denominada objeto. Um objeto pode armazenar estados e reagir ou enviar as chamadas do métodos do objeto através de ações. Ao invés de serem criadas variáveis e funções que manipulam estes dados como nas linguagens procedurais, as linguagens baseadas em objetos possuem propriedades e métodos que definem e controlam o comportamento do objeto. A técnica de OO simplifica a vida do programador/projetista, quando referida a diminuição de tempo e custos de manutenção. Conseqüentemente, simplifica a construção de novos sistemas em relação aos sistemas completamente estruturados. Usando OO, o programador consegue atingir a simplificação e modularização no sistema. A orientação a objetos permite técnicas reusáveis para a construção de código, na linguagem de programação.

Este artigo descreve a importância do objeto e seu protocolo, para trabalhar com o reuso de código. Esta importância é tanta que no curso de Ciência da Computação da UFSC (Universidade Federal de Santa Catarina) Waslawick preparou um excelente livro sobre o assunto [13]. Também descreve e discute as técnicas para se alcançar o desenvolvimento de esquemas(*frameworks*) através de um conjunto de classes que incorpora as soluções para problemas comuns com grande reuso; enfatizando a utilização de padrões de projeto para facilitar e viabilizar a construção de classes.

2 O Objeto e seu Protocolo

A possibilidade de um objeto encapsular os dados e as operações, permite atingir a modularidade através de informações ocultas [2]. Assim, há duas características que diferem da linguagem de programação OO e da baseada em tipos de dados abstratos: o polimorfismo e a herança [1]. O polimorfismo se dá quando um método de uma classe apresenta várias formas, isto é, apresenta comportamentos diferentes de acordo com o tipo de ob-

jeto. O conceito de polimorfismo é diferente da sobrecarga de métodos. A sobrecarga ocorre quando em uma mesma classe existe métodos com nomes iguais e parâmetros diferentes enquanto que o polimorfismo precisa de vários objetos com implementações diferentes para o mesmo tipo. Com o uso do polimorfismo, as operações são efetuadas através de envio de chamadas dos métodos do objeto. Uma mensagem invoca o método, ou seja, o envio de chamadas do métodos do objeto causa polimorfismo. Já na herança, têm-se várias vantagens: reuso, classificação e criação de protocolos padrão. A herança facilita o desenvolvimento de protocolos que posteriormente são usados no polimorfismo. A classe herdada pode ser aumentada. Nem todas as linguagens de programação orientada a objeto permitem que a herança e os protocolos sejam separados. A linguagem C++ pede que um objeto tenha acesso a uma super classe para receber uma mensagem.

Todos os objetos são especificados por seus protocolos. Conhecendo o protocolo, determina-se o conjunto de chamadas do métodos do objeto a enviar. Com isso, declara-se que tipo é um protocolo e não uma classe. Objetos com protocolos idênticos são intercambiáveis, podendo gerar objetos complexos através de um conjunto de componentes compatíveis [7]. Todos os protocolos são importantes para definirem interface dentro do programa. Compartilhar protocolo é necessário pois só assim um programador pode saber as características de muitos dos componentes disponíveis. A linguagem Pascal não tem polimorfismo, o que ocasiona leves mudanças de nomes.

Protocolos padrão são freqüentemente representados por classes abstratas. Uma classe abstrata não tem instâncias, só sua subclasse. As raízes de uma hierarquia de classes são as classes abstratas, onde as classes-folha nunca são classes abstratas. Uma classe abstrata define poucos métodos que podem ser redefinidos pelas subclasses, assim pode ser usada como uma classe esqueleto, onde o usuário preenche em certas opções e reusa código no esqueleto. As classes abstratas podem conter métodos abstratos ou concretos. Os métodos abstratos não são implementados, só possuem uma declaração e seu comportamento é definido pela subclasse. A utilização de métodos abstratos obriga o programador a sempre implementar estes métodos nas classes derivadas. Uma classe não abstrata é concreta. Uma classe concreta provê uma definição para seus dados e algumas subclasses precisam de diferentes representações. Para fazer uma classe abstrata deve-se observar que ela será reusada várias vezes [7].

3 Esquemas

Esquema(*framework*) é uma simplificação dos projetos orientados a objeto que consiste de classes abstratas para cada componente específico. Estas classes podem ser direcionadas para uma biblioteca de classes, ou pode ser até uma aplicação orientada a objetos específica. Esquemas podem se unir, compartilhando classes. Ao se construir esquemas é necessário garantir a sua flexibilidade e generalização mantendo sempre o desempenho e eficiência das aplicações geradas por ele. De acordo com Evandro [4], essa é uma das dificuldades de se construir esquemas: manter a generalidade e, ao mesmo tempo, permitir a adaptação às especificidades de cada aplicação. Para se projetar um esquema, necessita-se de muita experiência e testes, tal como projetar componentes. Deve-se tomar o cuidado também de faze-lo simples o suficiente para ser aprendido. Construir um bom esquema custa caro e pode levar muito tempo, no entanto este tempo será recompensado na fase em que o esquema estiver sendo utilizado na construção das aplicações. Dentre as vantagens em se utilizar esquemas podemos citar a extensibilidade, modularidade, facilidade de manutenção, aumento de produtividade, qualidade do programa, e requisição de pouco esforço no desenvolvimento de aplicações devido ao reuso não somente de código mas também de elaboração.

3.1 Definição

Dentre as várias definições de esquema, este artigo sugere as mais utilizadas. Assim, Reinaldo [10] afirma: "[...] esquemas como um conjunto de classes-base (esqueleto) sobre o qual uma ferramenta é construída". Larman [8] define esquema como um conjunto coeso de classes que colaboram para fornecer serviços para o núcleo invariante de um sistema lógico. Contém classes concretas e abstratas que definem interfaces a serem seguidas, interações entre objetos das quais a aplicação deve participar como também outros invariantes. Silva [12] explica esquema como um conjunto de estruturas de classes que constituem implementações incompletas que quando estendidas, permitem produzir diferentes artefatos de um sistema.

3.2 Tipos e Modelos

De acordo com Reinaldo [10], para identificar a estrutura de um esquema, o usuário deve perguntar **como o esquema é usado?** Logo, existem três estruturas de esquema: caixa branca, caixa preta e híbrido.

Esquema caixa-branca é caracterizado pela necessidade de entendimento por parte do usuário de como as classes funcionam internamente. Para esse tipo direcionado a arquitetura, a implementação deve ser entendida para que seja possível modificar ou estender funcionalidades. Focado na herança, o problema principal desse esquema é que toda aplicação requer a criação de muitas subdivisões de novas classes, mas o elevado número delas pode se tornar difícil para um programador novo aprender bem o projeto.

Esquema caixa-preta é particularizado pela agregação de componentes. Essa agregação provê um comportamento específico direcionado a dados para a aplicação que está sendo gerada. Não é necessário entender sua estrutura interna pois esta não poderá ser alterada ou vista pelo usuário. O usuário deverá usar as interfaces fornecidas e funcionalidades já presentes. Focado na composição, cada componente requer um protocolo. Quando o usuário precisar entender só a parte de interface externa, então será um esquema caixa-preta. Os esquemas

de caixa-preta são mais fáceis de aprender e manusear, mas são poucos flexíveis.

Esquema híbrido ou esquema caixa-cinza é um resultado originário da união de duas classificações: caixa-branca e caixa-preta. Na maioria dos esquemas, sua classificação tem sido apontada na herança com algumas funcionalidades prontas na composição.

Dentre os esquemas caixa branca e preta, vale apresentar as seguintes diferenças: na branca, os estados de cada instância estão implicitamente avaliados para todos os métodos no esquema, tal como variáveis globais. Na preta, qualquer informação passada para o cliente deve ser passada explicitamente. O conjunto de chamadas do métodos do objeto que um objeto pode entender é essencialmente de seu tipo. De fato, determinando quais classes podem ser usadas em um esquema é possível determinar a compatibilidade de tipo.

Reinaldo [10] aponta outro item de classificação que deve ser compreendido em **onde o esquema é usado?** Abaixo, apresentam-se três modelos de esquemas.

Esquema de Suporte dedica a assuntos que envolvam serviços em nível de sistema operacional e não de aplicação, por exemplo: comunicação, acesso a arquivos, computação distribuída e linguagens de programação.

Esquema de Aplicação trata de assuntos que envolvam o projeto da construção de aplicações específicas. Encapsula conhecimento aplicável a uma vasta gama de aplicações, por exemplo: construção de interface gráfica para usuário, telecomunicações, finanças, produção e educação, geo-processamento etc.

Esquema de Integração é responsável por integrar aplicações distribuídas e componentes em uma mesma arquitetura. Encapsula conhecimento aplicável a aplicações pertencendo a um domínio particular de problema, por exemplo: ORB (*Object Request Broker*), CORBA (*Common Object Request Broker Architecture*) e o RMI (*Remote Method Invocation*).

3.3 Busca por Classes Abstratas (Genéricas)

A busca por um conjunto de classes simples é o grande objetivo para se chegar ao reuso. A diminuição de código visa quebrar grandes classes em pequenas classes. Para isso, deve-se verificar trechos de códigos comuns, criando assim novas classes, a qual serão herdadas futuramente. Deve-se tomar cuidado para que um esquema caixa-branca não se torne caixa-preta. Cada hierarquia de classe oferece a possibilidade de gerar um esquema. Simplificar classes é muito importante. Um projetista deveria ficar muito feliz quando uma abstração de classe é achada, não importando como foi achada.

3.4 Projeto

As definições de classes estabelecem o comportamento do projeto. Um projeto torna-se completo se o objeto e a função referenciada já foi definida. Usando de procedimentos, cada verbo é uma operação implementada por uma classe [1]. Johnson [7] comenta que é difícil distinguir novas classes de funções existentes, mas perguntas são feitas para dar um rumo e assim esclarecer. É necessário um julgamento humano para decidir quando e como uma classe será reorganizada. Um protocolo padrão deve ter um nome escolhido com cuidado.

Johnson [7] e Gamma [5] estabelecem uma lista de regras que ajudam no projeto:

- Introduzir recursão: mesmo se não existir uma real recursividade, se o método aparece como recursivo, então é chamado esta regra;
- Eliminar análise casual: trocar por chamadas do métodos do objeto, separar a classe para cada tipo de variável;
- Reduzir o número de argumentos: uma mensagem com muitos argumentos deve ser refinada;
- Reduzir o número de métodos: os métodos bem projetados são sempre pequenos. Isto facilita fazer uma subclasse de uma classe com métodos pequenos.

De acordo com Johnson [7], as regras para encontrar uma classe genérica são:

- Definir as hierarquias de classes para que se tornem finas e compridas: deverá ter vários níveis de profundidade;
- Declarar o topo da hierarquia da classe como abstrato: facilita a herança;
- Minimizar o acesso a variáveis: mínimos envios das chamadas do métodos do objeto;
- Especializar as subclasses: adicionar métodos que as caracterizam.

Johnson [7] relata as regras para encontrar esquemas:

- Dividir grandes classes: classes largas devem ser quebradas em classes menores a medida que elas crescem. Isso é necessário para se aproximar mais da simplicidade;
- Diferenciar o fator de implementação nos subcomponentes: algumas subclasses implementam métodos que diferem entre classes. Logo, deve-se criar uma classe de componente;
- Separar métodos que não se comunicam;
- Mandar chamadas do métodos do objeto para componentes em vez de para ele mesmo: um esquema baseado em herança pode ser convertido dentro de um esquema baseado em componente de estrutura caixa-preta, pela sobreposição de métodos usando as chamadas do métodos do objeto enviadas aos componentes;
- Reduzir a quantidade de parâmetros que são passados: desta forma pode-se dividir uma classe.

3.5 Desenvolvimento

Esquemas devem definir o comportamento de seus objetos. Aproveitando do reuso, projeto e código, o usuário pode usar, ampliar ou alterar para suas próprias soluções limitadas a aplicações específicas no qual o esquema foi projetado. A empresa Taligent [6] estipula as seguintes fases para desenvolvimento de esquemas:

- Gerar esquemas a partir da existência de problemas e soluções existentes;
- Desenvolver esquemas pequenos e específicos;
- Provê alteração por subclasses;
- Chama as funções do cliente;
- Controla o fluxo de execução;
- Define a interação dos objetos;
- Provê comportamento padrão;
- Construir esquemas usando um processo iterativo direcionado a prototipação e as decisões do cliente;
- Fornecer manutenção, suporte e outras características que um produto deve conter.

De acordo com Taligent, para ser de alta qualidade, esquemas devem ser classificados como:

- Completos: deve suportar as características dos clientes, provendo implementações, sendo built-in quando possível. Gerar concretas derivações de classes abstratas, de fácil entendimento e permitindo áreas alteráveis;
- Flexíveis: diferentes exemplos podem ser usados em diferentes contextos;
- Extensíveis: clientes podem facilmente adicionar ou modificar as funcionalidades. Provendo *hooks* é possível mudar comportamentos pela derivação de novas classes;
- Legíveis: a interação com o cliente deve acontecer de forma clara e esquemas devem estar bem documentados.

Esquemas precisam de investimento. Os benefícios não são necessariamente imediatos. Taligent [6] apresenta os benefícios e custos para se desenvolver um esquema. Assim, os benefícios são:

- Evolução rápida;
- Estimular experiências especificas do domínio;
- Promove consistência e uma melhor integração ao longo do sistema;
- Manutenção reduzida;
- Produtividade avançada.

Já os custos são:

- Requer mais esforços para construir e aprender;
- Programas podem ser difíceis de verificar erros;
- Requer documentação, manutenção e suporte;
- Projetos de esquemas precisam de mais tempo para criar que uma biblioteca procedural
- Clientes precisam de mais tempo para aprender a usar um esquema do que uma biblioteca procedural.

Em seu livro *"Design Patterns for Object-Oriented Software Development"*, Pree [9] enfatiza o uso de padrões para a construção de uma base forte. Neste livro, o autor cita que *"Design Patterns recently emerged as a glimmer of hope on the horizon for supporting the development and reuse of frameworks"*.

4 Padrões de Projeto

De acordo com Larman [8], Taligent [6] e Gamma [5] e Fowler [3], Padrões de Projetos vem identificar os nome e temas de abstrações comuns em projetos OO. Os padrões de projetos capturam além da intenção do projeto pela identificação dos objetos, também os objetos que interagem e como as responsabilidades são distribuídas. Eles constituem a base de experiência para construção de programas reusáveis que e atuam como construtores de blocos no qual o mais complexo projeto pode ser construído.

O primeiro a introduzir o conceito de padrões foi o arquiteto Christopher Alexander, descrevendo os elementos e suas regras [6]. Projetistas de OO tem aceitado o conceito de padrões e uso de linguagem para planejar, discutir e documentar projetos [3][11]. Da mesma forma que os objetos são decompostos para ter um alto nível de projeto, uma implementação orientação a objetos também pede a decomposição de um esquema em padrões recorrentes [10]. Alguns padrões são genéricos e alguns específicos a um domínio de problemas, sendo caracterizado por:

- Pré-condição: os padrões devem ser satisfeitos para este padrão para serem validados;
- Problema: o problema indicado por um padrão;
- Obstáculo: a conflitante força agindo para qualquer solução para o problema e as prioridades destes obstáculos; "Solução: a solução do problema.

Durante o processo de construção do esquema, o projetista analisará várias vezes o domínio de problema e assim refinará seu projeto. Esquemas pequenos são reusados mais freqüentemente. Silva cita que a melhor solução para se desenvolver esquemas é torna-los suficientemente pequenos, pois resultaria em ser usado em outros contextos [12].

5 Conclusão

Para o desenvolvimento de sistemas com alto nível de complexidade, torna-se o reuso de classes e módulos, polimorfismo e herança, imprescindíveis para a obtenção de um alto grau probatório na construção de um esquema orientado a objetos. A escolha do uso de padrões de projeto deve ser decisiva na tomada de decisão para a construção de um bom projeto orientado a objetos. Diferentes sistemas, com mesmo enfoque, podem ser construídos a partir das classes pré-estabelecidas do esquema. As técnicas documentadas nesse artigo vêm esclarecer estudantes e desenvolvedores que não dominam o conceito e a técnica para se construir um bom esquema.

Referências

[1] DEITEL, H. M. & DEITEL, P. J. *C++ How To Program*, 3 ed. Prentice Hall, Englewood Cliffs, New Jersey 07632, 2001.

[2] FAYAD, M. E. & SCHIMIDT, D. C. Surveying current research in object-oriented design. *Communications of ACM 40*, 10 (1997), 32–38.

[3] FOWLER, M. & SCOTT, K. *UML essencial: um breve guia para a linguagem padrão de modelagem de objetos*. Bookman, Porto Alegre, 2000.

[4] FREIBERGER, E. C. & E SILVA, R. P. Suporte ao uso de frameworks orientados a objetos com base no histórico do desenvolvimento de aplicações. *SBQS III* (2004), 2.

[5] GAMMA, E.; HELM, R.; JOHNSON, R. & VLISSIDES, J. *Padrões de projeto: soluções reutilizáveis de software orientado a objetos*. Bookman, Porto Alegre, 2000.

[6] IBM-TALIGENT(ED.). Building object-oriented frameworks. White paper, Fev. 1994.

[7] JOHNSON, R. E. & FOOTE, B. Designing reusable classes. *Journal of Object-Oriented Programming 1*, 2 (1998), 22–35.

[8] LARMAN, C. *Utilizando UML e padrões*. Bookman, Porto Alegre, 2000.

[9] PREE, W. *Design patterns for object oriented software development*. Addison-Wesley, 1994.

[10] REINALDO, F. Projecting a framework and programming a system for development of modular and heterogeneous artificial neural networks. Msc thesis. department of computer science, Federal University of Santa Catarina, Florianopolis, Brazil, Feb 2003.

[11] SAUVÉ, J. P. Definições de frameworks. `http://www.dsc.ufpb.br/~jacques/cursos/1999.2/map/material/frame/deffw.htm`, Fev. 2002.

[12] SILVA, R. P. *Suporte ao desenvolvimento e uso de frameworks e componentes*. Dissertação de doutorado em ciência da computação, Universidade Federal do Rio Grande do Sul, Porto Alegre, Brazil, 2000.

[13] WAZLAWICK, R. S. *Análise e Projeto de Sistemas de Informação Orientados a Objetos*. Elsevier, 2004.