SOFTWARE PATTERNS: FUNDAMENTOS, TIPOS E DESCRIÇÃO

Sérgio Teixeira de Carvalho[1]

Resumo
Especialistas, em cada projeto de software, adquirem experiência e conhecimento que naturalmente são empregados em novos projetos. Para casos onde boas soluções são encontradas, testadas e aplicadas, existem grandes chances de serem novamente empregadas com sucesso em contextos semelhantes. Um software *pattern* descreve de forma padronizada um problema recorrente e apresenta um esquema para a sua solução, tornando possível a reutilização da experiência adquirida na resolução do problema. Este artigo trata dos fundamentos em torno de software patterns, apresentando seus tipos e a forma de descrevê-los.

Palavras-chave: reutilização, padrões, forma, arquitetura

SOFTWARE PATTERNS: BASES, TYPES AND DESCRIPTION

Abstract
Specialists, in each project of software, acquire experience and knowledge that eventually are used in new projects. In the cases where good solutions are found, tested and applied, there are great possibilities for these methodologies are employed successfully in similar contexts. A software pattern describes in a standardized way a recurrent problem and presents a outline for its solution, making possible the reuse of the experience acquired in the solution of the problem. This article deals with the foundation of software patterns presenting its types and description.

Key-words: reuse, standard, form, architecture

Introdução

Reutilização, modularidade, abstração e separação de interesses (SZTAJNBERG, 1999) constituem-se em alguns conceitos há muito tempo investigados. Tais conceitos são entrelaçados e têm como fundamento a

[1] Mestre em Computação, Professor do curso de Processamento de Dados do Uni-ANHANGÜERA. Email: sergiocarvalho@anhanguera.edu.br

concepção de sistemas de software de qualidade, compostos de elementos reaproveitáveis em outros contextos e projetos.

Uma forma de se conseguir bons projetos arquiteturais é o reaproveitamento da experiência adquirida de outras arquiteturas concebidas. A questão central é como reaproveitar esta experiência, sem utilizar-se de práticas relacionadas à transmissão informal de idiomas arquiteturais. Por exemplo, um sistema pode ser definido, no nível da arquitetura, como um "sistema cliente-servidor", um *"pipeline"*, ou um "sistema em camadas". Todos estes termos são bastante genéricos e não conseguem expressar com exatidão as propriedades e soluções que poderiam ser reaproveitadas em outras arquiteturas.

Para documentar as soluções a ponto de reaproveitá-las, deve-se formalizá-las, transformando-as em *patterns* (padrões) que possam ser compreendidos e reutilizados em outros projetos. Esse artigo tem o objetivo de apresentar os fundamentos dos *patterns*, seus tipos e formas de descrição.

Fundamentos

Os *patterns* são modelos de organização de hierarquias de classes, protocolos e distribuição de responsabilidades entre classes e objetos, que caracterizam construções elementares do projeto orientado a objetos, por meio da reutilização.

Os princípios que envolvem os *patterns* têm relação direta com a reutilização de software, mais especificamente com a reutilização da experiência no desenvolvimento de software. A cada projeto criado, especialistas adquirem experiência e conhecimento que naturalmente são empregados em novos projetos. Para casos em que boas soluções são encontradas, testadas e aplicadas, existem grandes chances de serem novamente empregadas com sucesso em contextos semelhantes àqueles já experimentados.

Boas soluções podem ser compartilhadas, transformando-se em padrões utilizados de forma recorrente. Há livros sobre algoritmos, por exemplo, que destacam soluções eficientes para problemas relacionados à classificação de elementos (e.g., *sort*), ou ainda que mostram como realizar, da melhor maneira possível, a recuperação de determinado item de um

conjunto de elementos. Tais algoritmos foram exaustivamente analisados, implementados e testados por vários programadores, e formaram padrões que podem ser reutilizados por outros profissionais.

A idéia exposta pode ser aplicada em todos os níveis do desenvolvimento de um sistema de software, desde a concepção de sua arquitetura até a implementação. Entretanto, um padrão, ou *pattern*, deve ser descrito de forma padronizada, apresentando o problema e sua respectiva solução. Assim, o projetista ao descrever um *pattern*, pode compartilhar sua *expertise* com outros, tornando possível a reutilização de sua experiência quanto ao problema-solução descrito.

A documentação de um *pattern* é feita a partir da observação de sua ocorrência em diversos projetos de software. Esse tipo de documentação é interessante pois o conhecimento, restrito apenas a alguns especialistas, é capturado e moldado de uma forma que possa ser facilmente compartilhado.

Importante frisar que *patterns* não são construções apenas teóricas, mas sim artefatos que são descobertos em múltiplos sistemas (RISING, 1999). Existe uma regra quanto à documentação de determinado *pattern*: sua solução deve ter sido aplicada em pelos menos três sistemas – chamada "Regra de Três" (APPLETON, 2000). Tal regra garante que um *pattern* documenta uma solução já utilizada em situações reais, e não apenas uma boa idéia ainda sem aplicação.

Uma definição para software *pattern*, ou simplesmente *pattern*, está em (BUSCHMANN, 1996): um *pattern* descreve de forma padronizada um problema recorrente dentro de contextos específicos e apresenta um esquema genérico para sua solução. A solução é especificada por meio da descrição de seus componentes, incluindo suas responsabilidades, relacionamentos e a forma com a qual colaboram uns com os outros.

Em resumo, *patterns* possuem as seguintes características (BUSCHMANN, 1996):

(1) Um *pattern* trata um problema recorrente que aparece em situações específicas. Por exemplo, o problema de processar fluxos de dados (tratado pelo *pattern Pipes and Filters*) é recorrente e pode surgir quando do

desenvolvimento de sistemas de software nos quais a interação entre processos é necessária. Processos podem interagir por intermédio de *pipes* que conduzem os dados de um processo para o outro.

(2) *Patterns* documentam a experiência empregada em projetos de software, tornando-a disponível a outros projetistas. Eles portanto não são invenções, mas meios de reutilizar o conhecimento obtido por projetistas experientes quando do desenvolvimento de sistemas de software.

(3) *Patterns* identificam e especificam abstrações que estão acima do nível de classes, instâncias ou componentes individuais. Normalmente, um *pattern* descreve vários componentes, classes ou objetos e detalha suas responsabilidades e relacionamentos, bem como suas cooperações. Componentes em conjunto, portanto, resolvem o problema tratado por um *pattern*.

(4) A descrição de um *pattern* deve seguir o formato Contexto-Problema-Solução, apresentando o contexto no qual o problema deve ser tratado, o problema a ser resolvido e a correspondente solução.

O livro *Design Patterns: Elements of Reusable Object-Oriented Software* (GAMMA, 1995) foi o primeiro a ter aceitação como uma forma padronizada de descrição de *patterns* (APPLETON, 2000). Ele documenta vinte e três *design patterns* (um tipo de *pattern*) que podem ser aplicados no desenvolvimento de sistemas orientados a objeto.

Tipos de Patterns

Patterns podem ser descritos nas diversas fases do desenvolvimento de um sistema de software. Alguns deles podem auxiliar na estruturação de um sistema em subsistemas; outros suportam o refinamento de subsistemas ou componentes. Os *patterns* também auxiliam na fase de implementação, por meio da descrição de problemas e soluções relacionados a determinadas linguagens de programação.

Uma forma de categorizar *patterns* relaciona-se às diferentes fases de desenvolvimento de um sistema de software (RIEHLE, 1996;

BUSCHMANN, 1996). As categorias são: *architectural patterns*, *design patterns* e idiomas.

Architectural Patterns

Architectural Patterns são *patterns* relacionados à estruturação do sistema de software partindo-se da concepção de sua arquitetura. Eles são modelos *(templates)* utilizados para a definição de arquiteturas de software concretas. Expressam um fundamento estrutural para sistemas de software, fornecendo um conjunto de subsistemas pré-definidos, especificando suas responsabilidades e incluindo regras para a organização dos relacionamentos entre eles, de acordo com Buschmann (1996).

Quando da definição de uma arquitetura de software em particular, esse tipo de *pattern* pode ser empregado como um passo inicial. Ele pode ser analisado quanto à sua solução e um mapeamento pode ser feito entre os subsistemas do *pattern* e os subsistemas necessários à arquitetura em definição. Subsistemas desta, que podem ser componentes, são associados com os descritos no *pattern* utilizado, fazendo com que a arquitetura tenha sua concepção fundamentada no mesmo.

Um exemplo de *architectural pattern* é o Pipes and Filters, já citado. Ele fornece uma estrutura para sistemas que processam um fluxo de dados. Cada passo do processamento é encapsulado em um componente chamado filtro, e os dados são passados por meio de pipes entre filtros adjacentes. Há no *pattern* uma estrutura definida com propriedades, variantes, conseqüências de utilização etc. que são naturalmente reaproveitadas por novas arquiteturas. Layers também é um exemplo de *architectural pattern.* O livro POSA2, segundo Schmidt (2000), um dos mais recentes catálogos de *patterns*, apresenta várias descrições de *architectural patterns.*

Design Patterns

Design Patterns caracterizam-se como os mais conhecidos e empregados no desenvolvimento de sistemas de software. Tais *patterns* têm relação com a

fase de projeto *(design)* do sistema, no qual subsistemas de arquiteturas de software e seus relacionamentos são refinados ou detalhados em unidades arquiteturais menores, representados, por exemplo, por modelos de classes e objetos.

Um *design pattern* fornece um esquema para o refinamento de subsistemas (ou componentes) e seus relacionamentos. Ele descreve uma estrutura recorrente de comunicação entre componentes que resolva um problema geral dentro de um contexto particular (GAMMA, 1995).

Apesar de atuarem em um nível mais baixo que os *architectural patterns*, *design patterns* são independentes de linguagens ou paradigmas de programação segundo Buschmann (1996). Eles podem, entretanto, ter a descrição de técnicas de implementação e características específicas relacionadas a alguma linguagem de programação, quando for o caso. Fragmentos de código podem também ser descritos, com o objetivo de ilustrar como poderia ser sua implementação em determinada linguagem de programação.

O *design pattern* Observer, descrito em Gamma (1995), apresenta uma solução para o problema de implementar dependências entre objetos de um sistema.

Nome: Observer

Contexto: Um componente usa dados ou informações fornecidas por outro componente.

Problema: A mudança do estado interno de um componente pode introduzir inconsistências entre componentes que cooperam entre si. Para restaurar a consistência, precisa-se de um mecanismo para trocar dados ou informações de estado entre tais componentes.

Duas restrições (forças) estão associadas com este problema:

- Os componentes devem ser fracamente acoplados; o fornecedor

de informações não deve depender de detalhes de seus colaboradores.

- Os componentes que dependem do fornecedor de informações não são conhecidos a priori.

Solução: Implementar um mecanismo de propagação de modificações entre o fornecedor de informações (o observado ou subject) e os componentes dependentes dele (os observadores ou observers). Observadores podem, dinamicamente, registrarem-se nesse mecanismo. Sempre que o observado modifica seu estado, inicia o mecanismo de propagação de modificações para restaurar a consistência com todos os observadores registrados. Modificações são propagadas por intermédio da invocação de uma função especial de atualização, comum a todos os observadores.

Idiomas

Idiomas são *patterns* diretamente relacionados a aspectos de implementação específicos de uma linguagem de programação, segundo Buschmann (1996). Eles descrevem como implementar aspectos particulares de componentes utilizando características de uma dada linguagem.

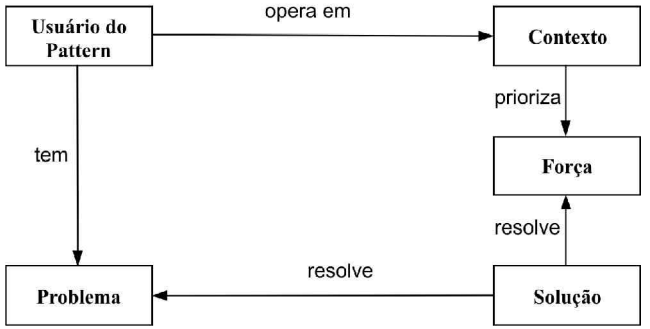
Idiomas capturam a experiência relacionada à programação e vinculada a linguagens específicas. Algumas vezes um idioma útil para uma linguagem de programação não faz sentido para outra (BUSCHMANN, 1996). Programadores C++, por exemplo, usam idiomas para gerenciar dinamicamente a alocação de recursos, o que em Java não há emprego, uma vez que ela fornece um mecanismo automático para a coleta de lixo (garbage collection). Idiomas para a linguagem C++ podem ser encontrados em (COPLIEN, 1992).

Elementos da Descrição de um Pattern

Para que um *pattern* possa ser utilizado no processo de desenvolvimento

de sistemas de software é necessário que seja compreendido quanto à sua solução para determinado problema. *Patterns*, portanto devem ser apresentados de uma forma apropriada no sentido de fornecer uma descrição clara do problema tratado e da solução proposta. Sua descrição também deve contemplar os detalhes necessários à sua implementação e considerar as conseqüências de sua aplicação. Descrições de *patterns* são informais, mas ao mesmo tempo estruturadas e precisas. Devem descrever por que e como usá-los, em vez de apenas documentar o que são.

O formato de descrição de *patterns* mais conhecido é chamado de "formato GoF", utilizado por Gamma (1995). Independentemente do formato utilizado na sua descrição, um *pattern* deve conter certos elementos essenciais. Dois *patterns* (*Mandatory Elements Present* e *Optional Elements When Helpful*) publicados em Martin (1997) apresentam, respectivamente, os elementos considerados obrigatórios e os considerados opcionais na descrição de um *pattern*. O primeiro mostra ainda a relação entre os elementos obrigatórios, apresentada na Figura 1. Os elementos obrigatórios são: Nome (*Name*), Contexto (*Context*), Problema (*Problem*), Forças (*Forces*) e Solução (*Solution*). Os elementos opcionais são, dentre outros: Conseqüências (*Consequences*), Patterns Relacionados (*Related Patterns*), Usos Conhecidos (*Know Uses*) e Código-Exemplo (*Sample Code*). Os nomes exatos desses elementos podem variar de uma descrição para outra, bem como a ordem de apresentação dos mesmos.



Fonte: Martin (1997)

Figura 1 - Relacionamento entre elementos da descrição de um *pattern*.

Nome (*Name*)

Nome pelo qual o par problema/solução pode ser referenciado. Um nome significativo permite a formação de um vocabulário que auxilie nas discussões conceituais do *pattern* em questão. Em casos onde o *pattern* possui outro(s) nome(s) pelo(s) qual(is) é conhecido, o elemento Também Conhecido Como (*Also Known As*) é utilizado para referenciá-los.

Contexto (*Context*)

Circunstâncias nas quais o *pattern* pode ser aplicado. Este elemento mostra a aplicabilidade do *pattern* e pode ser visto como a configuração inicial do sistema antes da aplicação do mesmo. Contexto é substituído por Aplicabilidade (*Applicability*) no formato GoF.

Problema (*Problem*)

O problema específico que precisa ser tratado e resolvido. Um exemplo concreto do problema pode ser descrito por meio da ilustração de modelos de classes, objetos etc. Tal exemplo pode aparecer como uma seção distinta chamada Exemplo (*Example*), auxiliando no entendimento dos demais elementos da descrição do *pattern*. No formato GoF, Problema é substituído por Intento (*Intent*) acrescido da seção Motivação (*Motivation*) que enfoca um exemplo ilustrativo.

Forças (*Forces*)

Descrevem forças (restrições) que devem ser levadas em conta quando da escolha de uma solução para o problema e como interagem ou entram em conflito com os objetivos almejados pelo *pattern*. Alguns *patterns* têm as suas forças descritas junto com Problema.

Solução (*Solution*)

Solução proposta para o problema. A descrição da solução pode incluir

figuras, diagramas, textos que identifiquem a estrutura do *pattern*, seus participantes e colaborações entre eles. A solução deve ser descrita levando-se em conta as forças definidas para o *pattern*.

Na descrição de *design patterns* utilizando o formato GoF, as seções Estrutura (*Structure*), Participantes (*Participants*), Colaborações (*Collaborations*) e Implementação (*Implementation*) substituem o elemento Solução.

A seção Estrutura mostra uma representação gráfica de classes baseada em UML (BOOCH, 1999). Participantes, por sua vez, é uma seção que mostra as classes e/ou objetos do *pattern* e suas responsabilidades. Em alguns *patterns*, esta seção traz cartões CRC (*Class-Responsibility-Collaborator*) ou CRC *cards* (BECK e CUNNINGHAM, 1989; AMBLER, 2002), descrevendo as classes com suas responsabilidades e colaboradores. A seção Colaborações apresenta como os participantes colaboram entre si de acordo com suas responsabilidades, e finalmente, Implementação mostra as técnicas envolvidas na implementação do *pattern*.

Conseqüências (*Consequences*)

Esse elemento descreve as conseqüências geradas e os benefícios obtidos a partir do emprego do *pattern* que está sendo definido. Essa descrição é feita a partir da análise da composição do sistema resultante após a aplicação do respectivo *pattern*. A comparação entre os elementos Conseqüências e Forças permite verificar se os objetivos do *pattern* foram atendidos. O nome Contexto Resultante (*Resultant Context*) substitui Conseqüências em muitas descrições de *patterns*.

Patterns Relacionados (*Related Patterns*)

Outros *patterns* de interesse são aqueles que tratam problemas similares ou ainda que auxiliem na composição do *pattern* que está sendo descrito. O elemento *Patterns* Relacionados pode ainda apresentar *patterns* que resolvam problemas mostrados em Conseqüências, quando for o caso.

Usos Conhecidos (*Known Uses*)

Descreve ocorrências conhecidas do *pattern* e suas aplicações em sistemas existentes. Auxilia na validação do *pattern* no sentido de apresentar uma solução para um problema recorrente.

Código-Exemplo (*Sample Code*)

Descreve um exemplo de código que apresente como implementar o *pattern*. Fragmentos de código são mostrados em uma determinada linguagem de programação. Os *design patterns* apresentados em Gamma (1995) têm os códigos escritos na linguagem C++ ou *Smalltalk.*

Conclusão

Este artigo mostrou conceitos essenciais e terminologias sobre software *patterns*, apresentando seus fundamentos, tipos e elementos que compõem a sua descrição.

A utilização de *patterns* têm proporcionado soluções coesas, estruturadas e, o mais importante, reutilizáveis aos problemas encontrados durante o desenvolvimento de software. Um vocabulário comum está sendo formado, facilitando a comunicação e, por conseguinte, o reaproveitamento da experiência e do conhecimento entre projetistas, desenvolvedores e implementadores de sistemas de software.

Referências Bibliográficas

AMBLER, S. W. CRC Modeling: bridging the communication gap between developers and users. White Paper Ambysoft, 2002. Disponível em http://www.ambysoft.com/crcModeling.html. Acesso em 24 jun. 2004.

APPLETON, B. Patterns and Software: essential concepts and terminology, 2000. Disponível em http://www.enteract.com/~bradapp/docs/patterns-intro.html Acesso em 24 jun. 2004.

BECK, K.; CUNNINGHAM, W. A Laboratory for teaching object-oriented thinking. OOPSLA'89, New Orleans, Louisiana. Outubro, 1989.

BOOCH, G; RUMBAUGH, J.; JACOBSON, I. **The unified modeling language user guide**. Object Technology Series: Addison-Wesley, 1999.

BUSCHMANN, F.; MEUNIER, R.; ROHNERT, H.; SOMMERLAD, P.; STAL, M. A **System of Patterns** - Pattern-Oriented Software Architecture. John Wiley & Sons, 1996.

COPLIEN, J. O. Advanced C++ - **Programming Styles and Idioms**. Addison-Wesley, Reading, Massachusetts, 1992.

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. **Design Patterns**: elements of reusable object-oriented software. Addison-Wesley, 1995.

MARTIN, R. C.; RIEHLE, D.; BUSCHMANN, F. **Pattern Languages of Program Design 3.** Software Pattern Series. Addison-Wesley, 1997.

RIEHLE, D.; ZÜLLIGHOVEN, H. Understanding and Using Patterns in Software Development. **Theory and Practice of Object Systems,** vol. 2, no. 1, p. 3-13, 1996.

RISING, L. Patterns: **A Way to Reuse Expertise**. IEEE Communications, p. 34-36, abril 1999.

SZTAJNBERG, A. Flexibilidade e Separação de Interesses para Concepção e Evolução de Sistemas Distribuídos. Exame de Tese de Doutorado. COPPE-UFRJ, março, 1999.

SCHMIDT, D.; STAL, M.; ROHNERT, H.; BUSCHMANN, F. **Pattern-Oriented Software Architecture**, Patterns for Concurrent and Networked Objects, John Wiley & Sons, 2000.