Componentes, Frameworks e Design Patterns

Ricardo R. Gudwin
DCA-FEEC-UNICAMP
09/11/2010

Componentes

Define-se o conceito de **componente**, como uma unidade modular com um conjunto de interfaces bem definidas, que pode ser substituído dentro de seu ambiente. O conceito de componente é oriundo da área de desenvolvimento baseado em componentes, onde um componente é modelado durante o ciclo de desenvolvimento e refinado sucessivamente durante a instalação e execução do sistema. Um aspecto importante do desenvolvimento baseado em componentes é o reuso de componentes previamente construídos. Um componente pode ser considerado como uma unidade autônoma dentro de um sistema ou subsistema. Ele deve possuir uma ou mais **interfaces**, que podem ser potencialmente disponibilizadas por meio de **portas**, e seu interior é normalmente inacessível. O acesso a um componente deve ocorrer única e exclusivamente por meio de suas interfaces. Apesar disso, um componente pode ser dependente de outros componentes, e a linguagem UML provê mecanismos para representar essa dependência, indicando as interfaces que um componente demanda de outros componentes. Esse mecanismo de representação de dependências torna o componente uma unidade encapsulada, de forma que o componente pode ser tratado de maneira independente. Como resultado disso, componentes e subsistemas podem ser reutilizados de maneira bastante flexível, sendo substituídos por meio da conexão de diversos componentes por meio de suas interfaces e dependências.

Um componente é uma unidade auto-contida que encapsula o estado e o comportamento de um grande número de objetos. Um componente especifica um contrato formal dos serviços que provê a seus clientes e dos serviços que necessita de outros componentes em termos de suas interfaces disponibilizadas e requeridas.

Um componente é uma unidade substituível que pode ser remanejada tanto durante o design como na implementação, por outro componente que lhe seja funcionalmente equivalente, baseado na compatibilidade entre suas interfaces. De modo similar, um sistema pode ser estendido adicionando-se novos componentes que tragam novas funcionalidades. As interfaces disponibilizadas e requeridas podem ser organizadas opcionalmente por meio de portas. Portas definem um conjunto de interfaces disponibilizadas e requeridas que são encapsuladas de maneira conjunta.

Frameworks

Outro conceito importante, relacionado ao reuso de implementações é o conceito de framework. Frameworks de software são mini-arquiteturas reutilizáveis que provêm a estrutura genérica e comportamento para famílias de abstrações de software, junto com um contexto de metáforas que especificam sua aplicação e uso dentro de um dado domínio. Em termos de software orientado a objetos, frameworks correspondem a um conjunto de classes cooperantes que permitem uma reutilização de design para uma classe de software específica. Assim, um framework provê uma orientação arquitetural, definindo quais as responsabilidades e colaborações entre objetos são necessárias para implementar uma determinada funcionalidade padrão. Para utilizar um framework, um designer normalmente faz o sub-classeamento de classes do framework, e utiliza suas funcionalidades por meio de herança. Assim, por exemplo, em Java, temos diversos frameworks que são

disponibilizados na API padrão do Java. Quando programamos janelas ou outros tipos de interfaces com o usuário, o que fazemos é utilizar o framework de janelas do Java. Quando utilizamos programação distribuída, utilizamos outro framework, fazendo uso de classes como *Socket* e *ServerSocket* pertencentes ao framework de acesso a rede do Java.

Observe que a reutilização de implementações por meio de frameworks é fundamentalmente diferente da reutilização de implementações por meio de componentes. Quando reutilizamos um framework, normalmente precisamos conhecer um certo número de diferentes classes que devem trabalhar em conjunto para que uma determinada funcionalidade seja reutilizada. Quando reutilizamos componentes, ao contrário, basta criar o(s) componente(s) e acessá-lo(s) diretamente por meio de sua(s) interface(s).

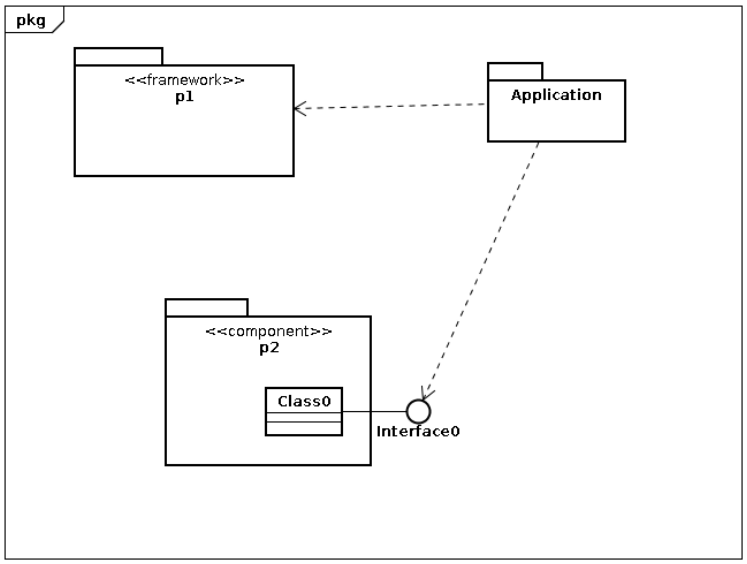

Figura 1: Uso de Frameworks e Componentes

A figura 1 acima demonstra o uso de frameworks e componentes. O pacote **p1** é um framework que é utilizado pelo pacote **Application**. O pacote **p2** é um componente, que é utilizado pelo mesmo pacote. Pode-se depreender se a arquitetura de um sistema é orientada a frameworks ou a componentes, verificando-se se a dependência que ocorre entre pacotes é direta no pacote (como no caso de **p1**) ou a alguma interface do pacote (como no caso de **p2**). Quando vamos realizar o design do sistema, temos que adotar uma postura. Ou realizamos o design do sistema utilizando-se um paradigma orientado a frameworks ou orientado a componentes. Podemos ainda adotar um paradigma híbrido, utilizando simultaneamente frameworks e componentes. Há vantagens e desvantagens em cada uma das alternativas. O uso de componentes normalmente facilita o reuso, pois basta criarmos o componente e utilizarmos o mesmo por meio de sua(s) interface(s). Entretanto, o reuso de componentes só é possível quando encontramos um componente que satisfaça nossas necessidades. Nem sempre podemos ter um componente que tenha um funcionamento exatamente da maneira que desejamos. Nestes casos, é melhor utilizar um framework, pois o mesmo é mais flexível, proporcionando uma adequação a nossas necessidades. Entretanto, o uso de frameworks normalmente é mais complicado, pois para utilizar um framework temos que conhecer detalhadamente todas as classes que o compõem, o que nem sempre é uma tarefa fácil. Por exemplo, se necessitamos fazer um acesso ao banco de dados, poderíamos utilizar um componente de acesso ao banco de dados, que abstraísse todo o processo de conexão ao banco, montagem de queries, etc. Caso este componente esteja disponível, esse seria o processo mais adequado. Entretanto, caso o componente não esteja disponível, poderia-se utilizar o framework do Java para acesso a banco de dados, o JDBC. Neste caso, teríamos que conhecer todas as classes envolvidas no uso do framework, por exemplo as classes *DriverManager*, *Connection*, *PreparedStatement*, *ResultSet*, *ResultSetMetaData*, etc., bem como seus métodos principais. Os

frameworks, dessa forma, são bem mais poderosos do que os componentes (em termos de flexibilidade), mas são bem mais complexos de serem utilizados. Os componentes, por outro lado, são bem mais simples de serem utilizados, mas podem simplesmente não estarem disponíveis ou não atenderem às demandas de especificação do sistema. De maneira reversa, é bem mais simples criarmos um novo framework do que criarmos um novo componente. Para a criação de um framework, basta criarmos um conjunto de classes cooperantes que implementem uma funcionalidade. Já para criarmos um novo componente, é necessário um passo adicional que é a criação das interfaces, e sua ligação com as classes internas ao componente, o que resulta em um pouco mais de trabalho.

Durante a fase de design, diversos fatores diferentes podem levar à escolha de uma arquitetura orientada a frameworks ou uma arquitetura orientada a componentes. Caso existam componentes disponíveis que atendam as demandas de funcionalidade, deve-se dar preferência ao uso de componentes. Caso não seja possível reutilizar diretamente um componente, deve-se considerar se vale a pena desenvolver um novo componente, reutilizar algum framework que exista disponível ou desenvolver um novo framework. Caso seja possível se abstrair a funcionalidade, de tal forma que exista grande chance futura de reuso, deve-se considerar o desenvolvimento de um novo componente. Caso esse componente possa ser reutilizado no futuro, isso pode fazer valer a pena o custo adicional em seu desenvolvimento. Caso não seja possível depreender um potencial de reuso que valha a pena, deve-se considerar um framework. Caso haja algum framework maduro que possa ser reutilizado (principalmente se esse framework já foi extensivamente testado e validado), deve-se promover a reutilização do framework. Caso o framework existente esteja ainda em um nível experimental, ou não se tenha muita confiança em seu funcionamento, deve-se optar por desenvolver um novo framework.

Design Patterns

O conceito de **design patterns** é hoje fundamental para que um design seja de boa qualidade. Mas … afinal … o que exatamente significa isso ?

A idéia toda sobre *design patterns* começa na experiência que os desenvolvedores de software adquiriram ao longo dos anos durante sua convivência com o desenvolvimento de software. Até mesmo programadores menos experientes já passaram pelo processo de reutilização de uma idéia boa que funcionou muito bem em um programa e que foi "reciclada" em programas posteriores. Imaginemos agora, se todas essas boas idéias fossem organizadas e sistematizadas em um grande "banco de idéias", e disponibilizadas para consumo em larga escala. Essa é a idéia por trás dos design patterns.

Ao longo dos anos, os desenvolvedores de software orientado a objetos experientes acumularam um grande repertório de princípios gerais e soluções que os guiam frequentemente em suas decisões no desenvolvimento de novos softwares. Esses princípios, uma vez identificados e isolados, podem ser formalizados/compilados, dando origem aos chamados patterns (padrões). Assim, os patterns codificam um conhecimento comum sobre princípios de como resolver problemas que aparecem repetidamente em uma dada atividade criativa. Observe que essa idéia de patterns é maior do que simplesmente sua aplicação em desenvolvimento de sistemas de software. Ela pode ser generalizada para qualquer tipo de atividade criativa.

Qual é o formato em que esses patterns são codificados ? Veremos que os patterns aparecem sempre na forma de um par problema/solução, que pode ser aplicado em novos contextos, acompanhados de conselhos sobre como devem ser aplicados. Em geral, os patterns têm um nome sugestivo, o que de certa forma enraíza o conceito do pattern em nossa memória, promovendo seu uso sempre que possível.

Em engenharia de software, a origem dos chamados design patterns se deu a partir da publicação do livro: "Design Patterns: Elements of Reusable Object-Oriented Software" de Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides (também conhecidos pela alcunha de "gangue dos quatro", ou

em inglês Gang of Four - GoF). Nesse livro, os autores isolaram um grande número de patterns significativos para o design de software e os disponibilizaram para uso público. Entretanto, o uso de patterns não se restringe à engenharia de software. Ele pode ser utilizado (e é) em qualquer tipo de atividade que envolva o design, como por exemplo em organizações, processos, no ensino, na arquitetura, etc. Mesmo com relação ao desenvolvimento de software, o uso mais famoso de patterns é na fase de design, mas podem haver outros tipos de uso. Alguns autores sugerem a utilização de patterns na fase de análise (o que seria uma espécie de patterns de análise, ao invés de design patterns). Diversos outros livros foram lançados depois do livro da gangue dos quatro. Dentre eles, o livro "Pattern-Oriented Software Architecture: A System of Patterns" (também chamado de POSA book, devido a suas iniciais), de Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Somerlad e Michael Stal (que em contraposição à gangue dos quatro acabou recebendo a alcunha da "Gangue dos Cinco da Siemens" ou Gang of Five - GoV). Outra referência importante nesse sentido foi "Pattern Languages of Program Design and PLPD 2 - selected papers from 1st and 2nd conferences on Patterns Languages of Program Design", que faz uma compilação dos principais trabalhos apresentados nas primeiras conferências realizadas cujo tema foi exclusivamente os patterns.

Na verdade, a origem do termo "pattern" é derivada dos trabalhos de um arquiteto, chamado Christopher Alexander, que escreveu vários livros em tópicos relacionados ao planejamento urbano e arquitetura de construção civil. Assim, a origem primeira do termo, está relacionada ao uso de soluções comuns em arquitetura, que acabaram por criar os chamados "estilos arquitetônicos". Por exemplo, o uso de grandes colunas: um estilo grego-clássico, ou então o uso de arcos - um colonial mexicano. Em 1987, Ward Cunningham e Kent Beck decidiram utilizar algumas das idéias de Alexander no desenvolvimento de conselhos para a geração de estruturas eficientes de código em Smalltalk, que se repetiam em diversos programas.

Era o primeiro uso dos patterns relacionado a programação de sistemas. Em 1991, Jim Coplien publicou "Advanced C++ Programming Styles and Idioms" (idiomas são um tipo especial de pattern). De 1990 a 1992, os membros da GoF iniciaram a compilação de um catálogo de patterns. Em 1993 e 94, novas conferências sobre patterns surgiram e logo a seguir apareceu o livro de Design Patterns da GoF, que acabaram por criar essa área de estudos dentro da engenharia de software chamada de "design patterns".

Mas, … , afinal de contas, … o que são patterns ? Diversas respostas podem ser dadas a esta pergunta. Por exemplo: Dirk Riehle e Heinz Zullighoven definem um pattern da seguinte maneira: "Um Pattern corresponde a uma abstração de uma forma concreta que aparece recorrentemente em contextos específicos e não-arbitrários". Puxa vida … complicado, não ? Vamos ler de novo … "abstração" …. "forma concreta" …. "aparece recorrentemente" … "contextos específicos e não arbitrários". Assim vemos em primeiro lugar que um pattern não é algo concreto, mas sim uma abstração. Abstração de quê ? Ora, de formas concretas. Mas que formas concretas ? Aquelas que aparecem recorrentemente, ou seja, repetidas vezes. E onde ? Em contextos específicos e não arbitrários, ou seja, de formas que não podemos prever a princípio, e sempre de uma maneira um pouco diferente. Assim vemos que a noção de pattern está voltada para a solução de problemas de design que se repetem em diferentes contextos. Vejamos uma outra definição: "um pattern corresponde ao encapsulamento de uma informação instrutiva que captura a estrutura essencial e a efetividade de uma família de soluções de sucesso em problemas recorrentes que surgem em determinados contextos e sistemas de forças". Aqui vemos uma complementação da definição anterior, ou seja, a idéia de que os patterns capturam soluções completas e não simplesmente princípios gerais ou estratégias. Essa é uma característica muito importante de um pattern. Outra característica importante é que os patterns são conceitos testados e aprovados e não teorias ou especulações. Assim, dizemos que os patterns são descobertos, e não inventados ! Para que um pattern seja descoberto, é necessário que ele apareça repetidas vezes dentro do código de diferentes desenvolvedores, e sempre como uma solução de sucesso. Caso a solução seja um fracasso, teremos os

chamados *anti-patterns*.

Podemos dizer que o uso de frameworks corresponde a uma reutilização do design por meio da reutilização direta do código (no caso, por meio de herança). Assim, um framework nada mais é do que a implementação de um sistema de design patterns.

Entretanto, é necessário fazermos uma diferenciação bem explícita entre o que é um pattern e o que é um framework. Design patterns são entidades mais abstratas que frameworks. Frameworks estão de certa forma embutidos no código, ao passo que somente exemplos (ou instâncias) de patterns estarão embutidas no código. Uma vantagem dos frameworks é que estes podem ser formalizados diretamente em uma linguagem de programação e não somente estudados, sendo executados e reutilizados diretamente. Design patterns, ao contrário, necessitam ser implementados a cada vez que são utilizados.

Observe que um framework pode conter diversos design patterns, mas o contrário nunca é verdadeiro. Design patterns são menos especializados que frameworks. Frameworks sempre têm um domínio particular de aplicações em vista. Design patterns podem ser utilizados, em princípio, em qualquer tipo de aplicação ao contrário.

GRASP Patterns

Um conjunto de patterns que é particularmente importante para um design bem feito inclui os chamados GRASP Patterns (General Responsibility ASsignement Principles), que determinam a atribuição de responsabilidades a objetos durante a fase de design. Essa atribuição normalmente é feita durante a atividade de design de casos de uso e aparece durante a criação dos diagramas de interação (diagramas de comunicação e diagramas de sequência) que detalham o caso de uso. Os GRASP patterns descrevem os princípios gerais para a atribuição de responsabilidades a objetos, expressos na forma de patterns. O conhecimento desses patterns é importante durante a fase de criação dos diagramas de interação, pois permitem a criação de programas de melhor qualidade. A assimilação de GRASP patterns é especialmente útil com relação a programadores com pouca experiência, pois com seu uso, rapidamente passarão a gerar um código que imbui um conjunto de princípios básicos com os quais somente programadores experientes estão acostumados a lidar.

Os principais GRASP patterns são os identificados pelos seguintes nomes: *Expert*, *Creator*, *High Cohesion, Low Coupling* e *Controller*. Para compreendê-los melhor, é necessário antes analisarmos o que significa atribuirmos responsabilidade a objetos.

Na programação orientada a objetos, criamos o design de um caso de uso distribuindo tarefas ou responsabilidades entre múltiplos objetos participantes do sistema. Façamos uma metáfora com relação à construção de uma casa. Utilizando os princípios da programação estruturada, identificaríamos as tarefas necessárias à construção de uma casa e as executaríamos uma após a outra. Utilizando os princípios da programação estruturada, o que fazemos é contratar uma equipe de objetos e atribuir responsabilidades a cada um deles. Assim, teremos o pedreiro que terá como responsabilidade assentar tijolos, o azulejista que terá como responsabilidade assentar os azulejos, o encanador que terá como responsabilidade a parte hidráulica, o eletricista, a parte elétrica, o marceneiro a montagem do telhado e por aí vai. Uma vez que a equipe esteja formada, criamos então uma dinâmica de trabalho em que cada objeto sabe quais são as suas responsabilidades, e passam então a interagir entre si para que a casa seja montada. Um programa orientado a objetos funciona exatamente desta maneira. Criamos um conjunto de objetos, atribuimos as responsabilidades aos objetos e depois colocamos eles para interagir. Façamos agora uma abstração dessa história para ver que tipos de responsabilidades podem existir. Basicamente existem os seguintes tipos de responsabilidade:

- Responsabilidade de Saber
    - Guardando o conhecimento consigo
    - Sabendo quem sabe
    - Sabendo como derivar um conhecimento de outros conhecimentos
- Responsabilidade de Fazer
    - Fazendo alguma coisa sozinho
    - Criando novos objetos e delegando-lhes sub-responsabilidades
    - Controlando e coordenando atividades em objetos que já existem

Observemos que existem dois tipos básicos de responsabilidades: a responsabilidade de saber e a responsabilidade de fazer. No primeiro caso, os objetos têm apenas a missão de informar, quando requisitados, alguma informação pela qual são responsáveis. No segundo caso, os objetos têm como responsabilidade exercer alguma atividade. No primeiro caso ainda, um objeto pode guardar o conhecimento em si próprio (em seus atributos), ou então ele deve fazer referências a outros objetos, que detêm o conhecimento desejado. Ou ainda, podem, a partir de outros conhecimentos, derivar o conhecimento em questão. No segundo caso, (a responsabilidade de fazer), o objeto pode fazer alguma coisa por si só (por meio de seus métodos), ou então utilizar outros objetos para ajudá-lo. Nesse caso, ele tanto pode criar por si só esses objetos, como pode controlar e coordenar um conjunto de objetos já existentes. Veremos que a maneira como atribuímos essas responsabilidades entre objetos pode fazer toda a diferença entre um design bem feito e um design mal feito. Vejamos os GRASP patterns:

PATTERN: EXPERT

PROBLEMA: Quem deve ser o responsável em prover algum tipo de informação ?

SOLUÇÃO: Devemos atribuir a responsabilidade ao objeto **expert** (ou especialista) - uma classe que tem a informação necessária para informar os dados em questão - seja por sabê-los por si próprio, ou sabendo quem sabe ou sabendo calculá-lo.

Esse pattern nos diz que mesmo que optemos por distribuir o conhecimento em diversos objetos, devemos sempre ter um objeto do tipo expert, que sabe destrinchar o conhecimento e nos oferecê-lo. Assim, sabemos sempre em quem confiar e não nos perdemos fazendo procuras ou buscas em estruturas de dados que podem ser muitas vezes confusas. Assim, ao invés de fazermos a busca pelos dados nós mesmos, devemos utilizar um expert.

Os benefícios desse pattern são vários. Em primeiro lugar, o uso de um expert mantém o encapsulamento no programa, fazendo com que o código fique inteligível. Além disso o expert suporta um outro pattern (a ser visto a seguir), chamado low-coupling, que em geral leva a sistemas mais robustos e gerenciáveis. Da mesma maneira, como o comportamento normalmente é distribuído em múltiplas classes, encoraja-se a criação de classes do tipo "lightweight", que são mais coesas. Isso está de acordo com um outro pattern chamado de high-cohesion, que portanto também é suportado.

PATTERN: CREATOR

PROBLEMA: Quem deve ser o responsável pela criação de novas instâncias de alguma classe ?

SOLUÇÃO: Devemos atribuir a uma classe B a responsabilidade pela criação de uma instância da classe A se uma das condições abaixo for verdadeira:

- B agrega objetos do tipo A
- B contém objetos do tipo A

- B referencia objetos do tipo A
- B utiliza objetos do tipo A
- B possui os dados de inicialização que devem ser passados a A quando este é criado

---

Este pattern nos dá portanto as condições em que um objeto deve ser o creator de outros objetos. Seu principal benefício é que suporta o pattern low-coupling, o que implica em menores dependências de gerenciamento e maiores oportunidades de reuso.

PATTERN: LOW COUPLING

PROBLEMA: Como suportar baixa dependência e maiores potencialidades de reutilização entre classes ?

SOLUÇÃO: Devemos atribuir a responsabilidade de tal forma que o acoplamento permaneça baixo.

Para compreendermos o pattern acima, é necessário compreendermos o que significa acoplamento. Acoplamento é uma medida de quão fortemente uma classe está conectada a outras classes. Essa conexão pode ser na forma da necessidade de referenciar ou ter que utilizar a outra classe para seu próprio funcionamento. Uma classe com baixo acoplamento não é dependente de muitas outras classes. Uma classe com alto acoplamento, ao contrário, depende de diversas outras classes para seu funcionamento. Classes com esse perfil são indesejáveis, pois mudanças em alguma das classes de que depende implicam mudanças na classe em questão. Da mesma forma, classes desse tipo são mais difíceis de se compreender de forma isolada e por conseguinte mais difíceis de serem reutilizadas, pois seu uso implica na necessidade da presença adicional das classes de que depende.

Devemos observar que o low-coupling é um princípio que devemos ter em mente a cada instante, quando tomamos nossas decisões de design. É uma meta que deve estar sendo continuamente considerada. O low-coupling suporta o design de classes que são mais independentes entre si e mais re-utilizáveis. Ele não deve ser utilizado, entretanto, de maneira isolada, mas sim conjuntamente com outros patterns. Dentre seus benefícios, temos que classes que implementam o low-coupling são mais dificilmente afetadas por mudanças em outros componentes, são mais simples de se entender e mais convenientes para o reuso.

PATTERN: HIGH COHESION

PROBLEMA: Como manter a complexidade de um sistema gerenciável ?

SOLUÇÃO: Devemos atribuir responsabilidades de tal forma que a coesão entre os objetos permaneça alta.

De novo, para compreendermos esse pattern, é necessário que compreendamos o que significa coesão. Coesão é uma medida de quão fortemente relacionadas e focalizadas estão as responsabilidades atribuidas a uma certa classe. Uma classe com responsabilidades afins, e que não se encontram sobrecarregadas de responsabilidades, possuem uma alta coesão. Uma classe com baixa coesão ou tem responsabilidades que não são afins ou então tem muitas responsabilidades. Tais classes são indesejadas pois são difíceis de compreender, difíceis de reutilizar, difíceis de manter, ao mesmo tempo que são muito frágeis, ou seja, são constantemente afetadas por mudanças no sistema.

Da mesma maneira que o low-coupling o high-cohesion é um princípio que deve ser mantido em vista

durante todas as decisões de design. A delegação de responsabilidades para classes filhas pode ser uma maneira de aumentar a coesão, ao mesmo tempo que mantém o acoplamento baixo. Os benefícios da alta coesão são uma maior claridade e maior facilidade de compreensão de nosso design, ao mesmo momento em que manutenções e melhorias são simplificadas. O baixo acoplamento é sempre suportado também. Classes que suportam responsabilidades afins suportam um maior potencial de reuso, pois podem ser reutilizadas quase que diretamente em outras situações. Isso ocorre pois classes de alta coesão possuem um propósito bem específico.

PATTERN: CONTROLLER

PROBLEMA: Quem deve ser o responsável por gerenciar eventos do sistema ?

SOLUÇÃO: Devemos atribuir a responsabilidade a uma classe representando a entidade que pretensamente deveria estar reagindo ao evento (**controller**).

Esse GRASP pattern nos diz que devemos concentrar o tratamento de eventos em objetos do tipo controller, ou seja, que sejam dedicados a essa finalidade. Em princípio, teremos um número de controllers proporcional aos objetos de interface recebendo eventos do sistema operacional. Algumas linguagens, como o Java por exemplo, em seu framework de janelas favorece e recomenda o uso de controllers. Dentre outros benefícios, o uso de controllers aumenta o potencial de reuso e compreensão de componentes.

De uma maneira geral, durante o design do sistema, devemos estar considerando o uso dos GRASP patterns como diretivas na concepção de interação entre objetos. Ao fazer isso, estaremos melhorando a qualidade de nosso software, independentemente da linguagem de programação que iremos adotar para implementação.

GoF Patterns

A seguir, temos uma visão resumida dos GoF Patterns, conforme o livro "Design Patterns: Elements of Reusable Object-Oriented Software" de Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides. Um resumo desses patterns pode ser visto na figura a seguir:

		Propósito		
		Criacional	Estrutural	Comportamental
Escopo	Classe	Factory Method	Adapter (classe)	Interpreter Template Method
	Objeto	Abstract Factory Builder Prototype Singleton	Adapter (objeto) Bridge Composite Decorator Façade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

A descrição completa destes patterns está além dos propósitos deste curso. Entretanto, uma versão sumarizada dos mesmos é apresentada em ordem alfabética, a seguir.

Abstract Factory

- Provê uma interface para criar famílias de objetos relacionados ou dependentes, sem especificar suas classes concretas

Adapter

- Converte a interface de uma classe em outra interface que os clientes esperam. O Adapter permite que classes com interfaces incompatíveis trabalhem entre si

Bridge

- Desacopla uma abstração de sua implementação, de tal forma que ambas podem variar de maneira independente

Chain of Responsibility

- Evita o acoplamento entre o sender de uma requisição e o receiver da mesma, dando a chance a mais de um objeto de responder a uma requisição. Encadeia o objeto sendo recebido e o passa ao longo da cadeia até que um objeto o processe

Command

- Encapsula uma requisição na forma de um objeto, permitindo que se parametrize clientes com diferentes requisições, requisições de filas ou logs, suportando o undo de operações

Composite

- Compõe objetos em estruturas do tipo árvore, de forma a representar hierarquias do tipo parte-todo. Clientes podem tratar objetos individuais ou objetos compostos de maneira uniforme.

Decorator

- Atribui dinamicamente novas responsabilidades a um objeto. Constitui-se de uma alternativa flexível ao subclasseamento para estender funcionalidades

Façade

- Provê uma interface unificada a um conjunto de outras interfaces em um subsistema, de forma a tornar o subsistema mais fácil de se utilizar

Factory Method

- Define uma interface para a criação de objetos, mas delega a subclasses a decisão de que classe instanciar

Flyweight

- Usa compartilhamento para suportar um vasto número de objetos de menor granularidade de maneira mais eficiente

Interpreter

- Dada uma linguagem, define uma representação para sua gramática, em conjunto com um interpretador que utiliza a representação para interpretar sentenças da linguagem

Iterator

- Provê uma maneira para acessar elementos em um objeto agregado, de forma sequencial, sem expor sua representação interna

Mediator

- Define um objeto que encapsula a maneira como um conjunto de objetos interagem entre si. O Mediator promove um acoplamento fraco, ao evitar que os objetos se referenciem mutuamente de maneira explícita, o que permite que se varie a interação de maneira independente

Memento

- Sem violar o encapsulamento, captura e externaliza o estado interno de um objeto, de tal forma que o objeto possa ser restaurado a este estado posteriormente

Observer

- Define uma dependência do tipo um-para-muitos entre objetos, de tal maneira que quando um objeto tem seu estado alterado, todos os objetos que dependem dele são notificados e atualizados automaticamente

Prototype

- Especifica que tipo de objeto criar usando uma instância prototípica, e cria novos objetos fazendo uma cópia do protótipo

Proxy

- Provê um substituto para outro objeto, que promove o acesso ao objeto desejado indiretamente

Singleton

- Garante que uma determinada classe possui somente uma única instância, e provê um ponto de acesso global a ela

State

- Permite que um objeto possa alterar seu comportamento, quando seu estado interno muda. O objeto parecerá ter modificado sua classe

Strategy

- Define uma família de algoritmos, encapsula cada um deles e torna-os intercambiáveis. O Strategy permite que o algoritmo varie independente dos clientes que o usam

Template Method

- Define o esqueleto de um algoritmo em uma operação, delegando alguns passos para subclasses. O Template Method permite que subclasses redefinam certos passos de um algoritmo, sem modificar a estrutura do algoritmo

Visitor

- Representa uma operação a ser realizada nos elementos de uma estrutura. O Visitor permite que se definam novas operações sem que seja necessário se mudar as classes dos elementos nos quais opera.