Padrões de Projeto

A T2Ti

Apresentação

A T2Ti nasce do sonho de três colegas que trabalhavam no maior banco da américa latina.

Tudo começa em 2007 com o lançamento do curso Java Starter. Logo depois veio o Siscom Java Desktop seguido de outros treinamentos.

Desde então a Equipe T2Ti se esforça para produzir material de qualidade que possa formar profissionais para o mercado, ensinando como desenvolver sistemas de pequeno, médio e grande porte.

Um dos maiores sucessos da Equipe T2Ti foi o Projeto T2Ti ERP que reuniu milhares de profissionais num treinamento dinâmico onde o participante aprendia na prática como desenvolver um ERP desde o levantamento de requisitos. Foi através desse treinamento que centenas de desenvolvedores iniciaram seu negócio próprio e/ou entraram no mercado de trabalho.

Em 2010 a T2Ti lança sua primeira aplicação para produção, o Controle Financeiro Pessoal. O sucesso foi tanto que saiu até em matéria no site Exame, ficando entre os 10 aplicativos mais baixados da semana.

Começa então a era de desenvolvimento de sistemas para alguns clientes exclusivos, pois o foco ainda era em desenvolvimento de treinamentos. A T2Ti desenvolve sistemas para o mercado nacional e internacional.

Atualmente a T2Ti se concentra nas duas vertentes: desenvolver sistemas e produzir treinamentos.

_____

Sumário

Padrões de Projeto

Padrões de Projeto

Introdução;

Conceitos;

Características Obrigatórias;

Template;

Solução de Problemas;

Como Selecionar um Padrão;

Catálogos | GoF | GRASP;

Estudando os Padrões.

Padrões de Projeto

Introdução

Projetar software orientado a objetos não é uma tarefa fácil, porém projetar software orientado a objetos reutilizável é ainda uma tarefa mais árdua. Para isso você deve identificar os objetos pertinentes, decompô-los em classes no nível correto de granularidade, definir as interfaces das classes, as hierarquias de herança e ainda estabelecer as relações-chave entre eles.

O seu projeto deve ser específico para o problema a ser resolvido, mas também genérico a ponto de ser reaproveitável para atender demandas futuras. Também deve evitar ou pelo menos minimizar as chances de um reprojeto.

Os projetistas de software orientado a objetos mais experientes lhes dirão que é muito difícil, senão impossível, se obter corretamente na primeira vez um projeto reutilizável e flexível. Antes que um projeto esteja terminado, eles normalmente tentam reutilizá-lo várias vezes, modificando-o a cada tentativa.

Padrões de Projeto

Introdução

Novos projetistas de software orientado a objetos são sobrecarregados pelas opções disponíveis, tendendo a recair em técnicas não orientadas a objetos que já usavam antes, ao passo que os projetistas com mais experiência realizam com mais facilidade bons projetos orientados a objetos. Leva-se um longo tempo para que os novatos aprendam o que é realmente um bom projeto orientado a objetos. Os projetistas experientes evidentemente sabem algo que os inexperientes não sabem.

Uma coisa que os projetistas experientes sabem que não devem fazer é resolver cada problema a partir de princípios elementares ou do zero. Em vez disso, eles reutilizam soluções que funcionaram no passado. Quando encontram uma boa solução, eles a utilizam repetidamente. Consequentemente, você encontrará padrões de classes e de comunicação entre objetos. Esses padrões resolvem problemas específicos de projetos e tornam os projetos orientados a objetos mais flexíveis e, em última instância, reutilizáveis ajudando os projetistas a reutilizar projetos bem-sucedidos.

Padrões de Projeto

Conceitos

O conceito de padrão de projeto foi criado na década de 70 pelo Engenheiro Civil Christopher Alexander que afirmou que "cada padrão descreve um problema no nosso ambiente e o cerne da sua solução, de tal forma que você possa usar essa solução mais de um milhão de vezes, sem nunca fazê-lo da mesma maneira".

Porém o conceito de padrões de projeto só foi amplamente difundido em 1995 com a publicação do livro Design Patterns: Elements of Reusable Object-Oriented Software escrito pela Gangue dos Quatro - GoF (Erich Gamma, Richard Helm, Ralphx Johnson e John Vlissides), catálogo que definiu 23 padrões de projeto. Posteriormente, vários outros catálogos de padrões de projeto foram publicados.

Padrões de Projeto

Conceitos

Os Padrões de Projeto de software, também conhecido como Design Patterns, descrevem soluções para problemas recorrentes no desenvolvimento de sistemas de software orientados a objetos visando uma melhor reutilização de software. Estes padrões tornam mais fácil a reutilização de projetos e arquiteturas bem sucedidas. Os padrões de projeto ajudam a escolher alternativas de projeto que tornam um sistema reutilizável e a evitar alternativas que comprometam a reutilização.

Segundo Marinescu (2002), um padrão de projeto é definido como uma solução desenvolvida utilizando boas práticas para um problema comum que ocorre várias vezes. Um padrão de projeto documenta e explana um problema importante que pode ocorrer no projeto ou implementação de uma aplicação e então discute a melhor solução prática para este problema.

Em termos de OO, padrões de projeto identificam classes, instâncias, seus papéis, colaborações e a distribuição de responsabilidades. São descrições de classes e objetos que se comunicam, implementados para solucionar um problema comum num contexto específico.

Padrões de Projeto

Conceitos

Cada padrão de projeto sistematicamente nomeia, explica e avalia um aspecto de projeto importante e recorrente em sistemas orientados a objetos. O nome do padrão é uma referência que podemos usar para descrever em uma ou duas palavras um problema de um projeto ou suas soluções e consequências.

Dar nome a um padrão aumenta imediatamente o nosso vocabulário de projeto. Isso nos permite projetar em um nível mais alto de abstração. Ter um vocabulário para padrões permite-nos conversar sobre eles com nossos colegas, em nossa documentação e até com nós mesmos. O nome torna mais fácil pensar sobre projetos e a comunicá-los, bem como os custos e benefícios envolvidos, a outras pessoas.

Os padrões de projetos são concebidos com o objetivo de capturar a experiências de projetos antigos de uma forma que as pessoas possam usá-las efetivamente tornando mais fácil a reutilização de ideias e arquiteturas bem sucedidas.

Características Obrigatórias

Embora um padrão seja a descrição de um problema, de uma solução genérica e sua justificativa, isso não significa que qualquer solução conhecida para um problema possa constituir um padrão, pois existem características obrigatórias que devem ser atendidas pelos padrões:

- Possuir um nome, que descreva o problema, as soluções e consequências. O nome permite definir o vocabulário utilizado pelos projetistas e programadores no nível mais alto de abstração.
- Todo padrão deve relatar de maneira clara quais são os problemas que quando inserido em um determinado contexto o padrão conseguirá resolver.
- Solução descreve os elementos que compõem o projeto, seus relacionamentos, responsabilidades e colaborações. Um padrão deve ser uma solução concreta, ele deve ser exprimido em forma de gabarito (algoritmo) que, no entanto, pode ser aplicado de maneiras diferentes.
- Todo padrão deve relatar quais são as suas consequências para que possa ser analisada a solução alternativa de projetos e para a compreensão dos benefícios da aplicação do projeto.

Template

Com base nas características obrigatórias, chegamos num template para a documentação de um padrão de projeto.

- Nome do Padrão: Deve ser facilmente lembrado, reflete o conteúdo do padrão.
- Problema: Uma descrição do problema que pode ser escrito em forma de pergunta.
- Contexto: Descreve o contexto do problema. As circunstâncias ou pré-condições sob as quais o problema pode ocorrer.
- Forças: As restrições ou características que devem ser seguidas pela solução. As forças podem interagir e ter conflitos umas com as outras.
- Solução: Deve ser direta e precisa.

Os padrões de projeto podem melhorar a documentação e a manutenção de sistemas ao fornecer uma especificação explícita de interações de classes e objetos e o seu objetivo subjacente. Em suma, ajudam um projetista a obter mais rapidamente um projeto adequado.

Padrões de Projeto

Solução de Problemas

Os padrões de projeto, de várias formas, ajudam os projetistas a solucionarem muitos dos problemas que encontram no dia a dia. Veremos alguns desses problemas e como os padrões de projeto tratam da sua solução.

Buscando Objetos Apropriados

Programas orientados a objetos são feitos de objetos. Um objeto empacota tanto os dados quanto os procedimentos que operam sobre eles. Os procedimentos são tipicamente chamados de métodos ou operações. Um objeto executa uma operação quando ele recebe uma solicitação ou uma mensagem de um cliente.

As solicitações são as únicas maneiras de mudar os dados internos de um objeto. Por causa destas restrições, diz-se que o estado interno de um objeto está encapsulado, ou seja, ele não pode ser acessado diretamente e sua representação é invisível do mundo exterior ao objeto.

Solução de Problemas

Buscando Objetos Apropriados

A parte difícil sobre projetos orientados a objetos é decompor um sistema em objetos. A tarefa é difícil porque muitos fatores entram em jogo: encapsulamento, granularidade, dependência, flexibilidade, desempenho, evolução, reutilização e assim por diante. Todos influenciam a decomposição, frequentemente de formas conflitantes.

As metodologias de projetos orientados a objetos favorecem muitas abordagens diferentes. Você pode escrever uma descrição de um problema, separar os substantivos e verbos e criar as classes e operações correspondentes. Ou você pode se concentrar sobre as colaborações e responsabilidades no seu sistema. Ou ainda, poderá modelar o mundo real e, na fase de projeto, traduzir os objetos encontrados durante a análise. Sempre haverá desacordo sobre qual é a melhora abordagem.

Solução de Problemas

Buscando Objetos Apropriados

Muitos objetos em um projeto provêm do modelo de análise. Porém, projetos orientados a objetos frequentemente acabam tendo classes de baixo nível, outras estão em um nível muito mais alto. A modelagem estrita do mundo real conduz a um sistema que reflete as realidades atuais, mas não necessariamente as futuras. As abstrações que surgem durante um projeto são as chaves para torná-lo flexível.

Os padrões de projeto ajudam a identificar abstrações menos óbvias bem como os objetos que podem capturá-las. Por exemplo, objetos que representam um processo ou algoritmo não ocorrem na natureza, no entanto, eles são uma parte crucial de projetos flexíveis. Esses objetos são raramente encontrados durante a análise ou mesmo durante os estágios iniciais de um projeto, eles são descobertos mais tarde durante o processo de tornar um projeto mais flexível e reutilizável.

Solução de Problemas

Indicando a Granularidade dos Objetos

Os objetos podem variar em tamanho e número. Podem representar qualquer coisa indo para baixo até o nível do hardware ou seguindo todo o caminho para cima até chegarmos a aplicações inteiras. Como decidirmos o que deve ser um objeto?

Os padrões de projeto tratam desse tópico. Há padrões que descrevem como representar subsistemas completos como objetos, como suportar enormes quantidades de objetos nos níveis de granularidade mais finos, descrevendo as maneiras específicas de decompor um objeto em objetos menores.

Solução de Problemas

Especificando a Interface dos Objetos

Cada operação declarada por um objeto especifica o nome da operação, os objetos que ela aceita como parâmetros e o valor retornado por ela. Isso é conhecido como a assinatura da operação. O conjunto de todas as assinaturas definido pelas operações de um objeto é chamado interface do objeto. A interface de um objeto caracteriza o conjunto completo de solicitações que lhe podem ser enviado. Qualquer solicitação que corresponde a uma assinatura na interface do objeto pode ser enviada para ele mesmo.

Um tipo é um nome usado para denotar uma interface específica. Quando dizemos que um objeto tem o tipo "Janela", significa que ele aceita todas as solicitações para as operações definidas na interface chamada "Janela". Um objeto pode ter muitos tipos, assim como objetos muito diferentes podem compartilhar um mesmo tipo.

Solução de Problemas

Especificando a Interface dos Objetos

Parte da interface de um objeto pode ser caracterizada por um tipo, e outras partes por outros tipos. Dois objetos do mesmo tipo necessitam compartilhar somente partes de suas interfaces. As interfaces podem conter outras interfaces como subconjuntos. Dizemos que um tipo é um subtipo de outro se sua interface contém a interface do seu supertipo. Frequentemente dizemos que um subtipo herda a interface do seu supertipo.

As interfaces são fundamentais em sistemas orientados a objetos. Os objetos são conhecidos somente através das suas interfaces. Não existe nenhuma maneira de saber algo sobre um objeto ou de pedir que faça algo sem intermédio de sua interface. A interface de um objeto nada diz sobre sua implementação - diferentes objetos estão livres para implementar as solicitações de diferentes maneiras. Isso significa que dois objetos que tenham implementações completamente diferentes podem ter interfaces idênticas.

Solução de Problemas

Especificando a Interface dos Objetos

Quando uma mensagem é enviada a um objeto a operação específica que será executada depende de ambos - mensagem e objeto receptor. Diferentes objetos que suportam solicitações idênticas, podem ter diferentes implementações das operações que atendem a estas solicitações. A associação em tempo de execução de uma solicitação a um objeto e a uma das suas operações é conhecida como ligação dinâmica.

Os padrões de projeto ajudam a definir interfaces pela identificação de seus elementos-chave e pelos tipos de dados que são enviados através de uma interface. Um padrão de projeto também pode lhe dizer o que não colocar na interface. Os padrões de projeto também especificam relacionamentos entre interfaces. Em particular, frequentemente exigem que algumas classes tenham interfaces similares, ou colocam restrições sobre interfaces de algumas classes.

Solução de Problemas

Herança de Classe Versus Herança de Interface

É importante compreender a diferença entre a classe de um objeto e seu tipo. A classe de um objeto define como ele é implementado. A classe define o estado interno do objeto e a implementação de suas operações. Em contraste a isso, o tipo de um objeto se refere somente à sua interface - o conjunto de solicitações às quais ele pode responder. Um objeto pode ter muitos tipos, e objetos de diferentes classes podem ter o mesmo tipo.

Naturalmente, existe um forte relacionamento entre classe e tipo. Uma vez que uma classe define as operações que um objeto pode executar, ela também define o tipo do objeto. Quando dizemos que o objeto é uma instância de uma classe, queremos dizer que o objeto suporta a interface definida pela classe. Enviar uma mensagem exige a verificação de que a classe do receptor implementa a mensagem, mas não exige a verificação de que o receptor seja uma instância de uma classe específica.

Solução de Problemas

Herança de Classe Versus Herança de Interface

É também importante compreender a diferença entre herança de classe e herança de interface (ou sub tipificação). A herança de classe define a implementação de um objeto em termos da implementação de outro objeto. Resumidamente, é um mecanismo para compartilhamento de código e de representação. Diferentemente disso, a herança de interface descreve quando um objeto pode ser usado no lugar de outro.

Muitos dos padrões de projeto dependem desta distinção. Por exemplo, os objetos no padrão Chain of Responsibility devem ter um tipo em comum, mas usualmente não compartilham uma implementação. Os padrões Command, Observer, State e Strategy são frequentemente implementados com classes abstratas que são puramente interfaces.

Solução de Problemas

Herança Versus Composição

As duas técnicas mais comuns para a reutilização de funcionalidade em sistemas orientados a objetos são herança de classe e composição de objetos. Como já explicado, a herança de classe permite definir a implementação de uma classe em termos da implementação de outra. A reutilização por meio de subclasses é frequentemente chamada de reutilização de caixa branca (ou aberta). O termo "caixa branca" se refere à visibilidade: com herança, os interiores das classes ancestrais são frequentemente visíveis para as subclasses.

A composição de objetos é uma alternativa à herança de classe. Aqui, a nova funcionalidade é obtida pela montagem e/ou composição de objetos, para obter funcionalidades mais complexas. A composição de objetos requer que os objetos que estão sendo compostos tenham interfaces bem definidas. Esse estilo de reutilização é chamado reutilização de caixa preta, porque os detalhes internos dos objetos não são visíveis. Os objetos aparecem somente como "caixas pretas".

Solução de Problemas

Herança Versus Composição

Cada uma possui vantagens e desvantagens. A herança de classes é definida estaticamente em tempo de compilação e é simples de usar, uma vez que é suportada diretamente pela linguagem de programação. Com a herança de classe também se torna mais fácil modificar a implementação que está sendo reutilizada.

Porém, a herança de classe tem também algumas desvantagens. Você não pode mudar as implementações herdadas das classes ancestrais em tempo de execução, porque a herança é definida em tempo de compilação. Além disso, as classes ancestrais frequentemente definem pelo menos parte da representação física das suas subclasses. A implementação de uma subclasse, dessa forma, torna-se tão amarrada à implementação da sua classe-mãe que qualquer mudança na implementação desta forçará uma mudança naquela.

Solução de Problemas

Herança Versus Composição

A composição de objetos é definida dinamicamente em tempo de execução pela obtenção de referências a outros objetos através de um determinado objeto. A composição requer que os objetos respeitem as interfaces uns dos outros, o que por sua vez exige interfaces cuidadosamente projetadas, que não impeçam você de usar um objeto com muitos outros.

Porém, existe um ganho. Como os objetos são acessados exclusivamente através de suas interfaces, nós não violamos o encapsulamento. Qualquer objeto pode ser substituído por outro em tempo de execução, contanto que tenha o mesmo tipo. Além do mais, como a implementação de um objeto será escrita em termos de interfaces de objetos, existirão substancialmente menos dependências de implementação.

Solução de Problemas

Herança Versus Composição

A composição de objetos tem outro efeito sobre o projeto do sistema. Dar preferência à composição de objetos à herança de classes ajuda a manter cada classe encapsulada e focalizada em uma única tarefa.

Suas classes e hierarquias de classes se manterão pequenas, com menor probabilidade de crescerem até se tornarem monstros intratáveis. Por outro lado, um projeto baseado na composição de objetos terá mais objetos (embora menos classes), e o comportamento do sistema dependerá de seus inter-relacionamentos ao invés de ser definido em uma classe.

Sendo assim, prefira a composição no lugar da herança.

Solução de Problemas

Delegação

Delegação é uma maneira de tornar a composição tão poderosa para fins de reutilização quanto à herança. Na delegação, dois objetos são envolvidos no tratamento de uma solicitação: um objeto receptor delega operações para o seu delegado. Isto é análogo à postergação de solicitações enviadas às subclasses para as suas classes-mãe.

Por exemplo, em vez de fazer da classe Janela uma subclasse de Retângulo (porque janelas são retangulares), a classe Janela deve reutilizar o comportamento de Retângulo, conservando uma variável de instância de Retângulo, e delegando o comportamento específico de Retângulo para ela. Em outras palavras, ao invés de uma Janela ser um Retângulo ela teria um Retângulo. Agora, Janela deve encaminhar as solicitações para sua instância Retângulo explicitamente, ao passo que antes ela teria herdado essas operações.

Padrões de Projeto

Gang Of Four Design Patterns

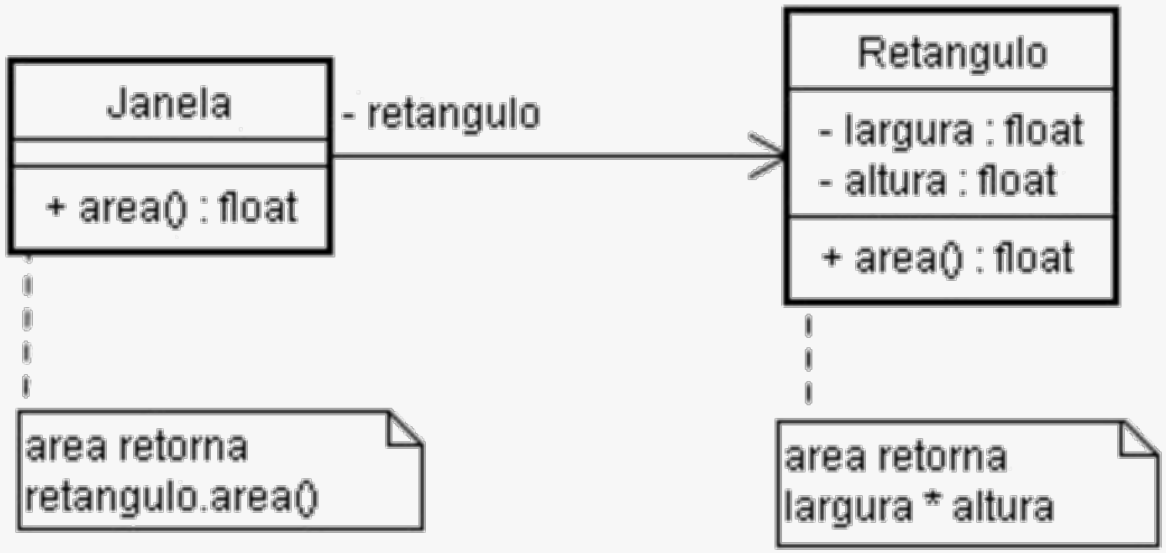
Solução de Problemas

Delegação

O diagrama ao lado ilustra a classe Janela delegando sua operação área a uma instância de Retângulo.

Uma flecha com uma linha cheia indica que um objeto de uma classe mantém uma referência para uma instância de outra classe.

A referência tem um nome opcional, nesse caso, "retângulo".



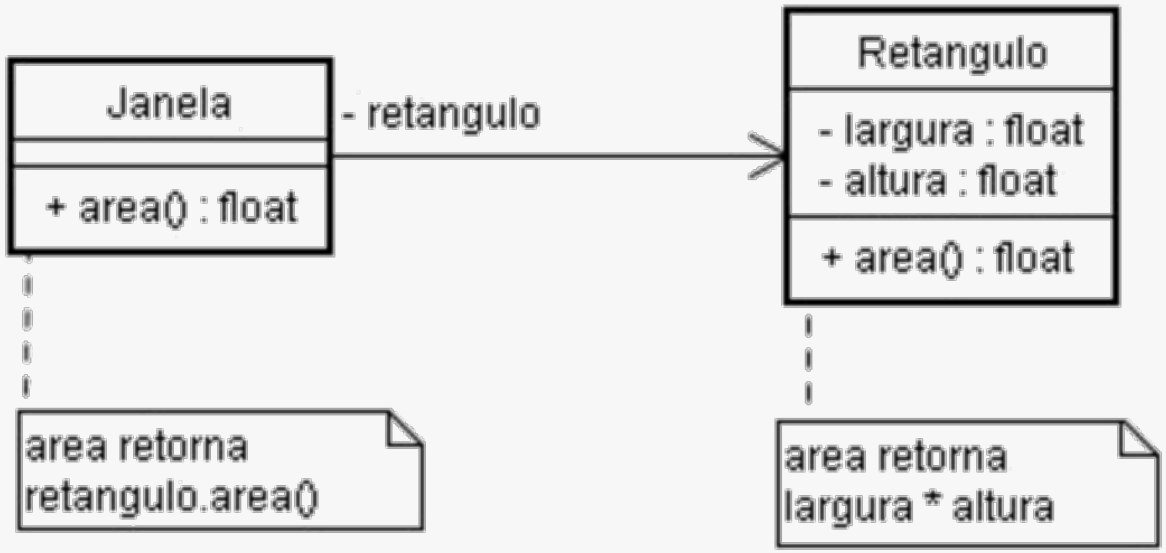
Padrões de Projeto

Gang Of Four Design Patterns

Solução de Problemas

Delegação

A principal vantagem da delegação é que ela torna fácil compor comportamentos em tempo de execução e mudar a forma como são compostos. A nossa Janela pode se tornar circular em tempo de execução, simplesmente pela substituição da sua instância Retângulo por uma instância de Circulo, assumindo-se que Retângulo e Circulo tenham o mesmo tipo.



Solução de Problemas

Delegação

A delegação tem uma desvantagem que ela compartilha com outras técnicas que tornam o software mais flexível através da composição de objetos: o software dinâmico, altamente parametrizado, é mais difícil de compreender do que o software mais estático.

Há também ineficiências de tempo de execução, mas as ineficiências humanas são mais importantes a longo prazo. A delegação é uma boa escolha de projeto somente quando ela simplifica mais do que complica.

Não é fácil estabelecer regras que lhe digam exatamente quando usar delegação, porque o quanto ela será efetiva dependerá das condições do contexto e de quanta experiência você tem com o seu uso.

Solução de Problemas

Delegação

A delegação é um exemplo extremo da composição de objetos. Ela mostra que você pode sempre substituir a herança pela composição de objetos como um mecanismo para a reutilização de código.

Diversos padrões de projeto usam delegação. Um exemplo é o padrão Strategy, onde um objeto delega uma solicitação específica para um objeto que representa uma estratégia para executar a solicitação. Um objeto terá somente um estado, mas ele pode ter muitas estratégias para diferentes solicitações.

A finalidade é mudar o comportamento de um objeto pela mudança dos objetos para os quais ele delega solicitações.

Padrões de Projeto

Solução de Problemas

Projetando Visando a Evolução do Sistema

A chave para maximização da reutilização está na antecipação de novos requisitos e alteração nos requisitos existentes e em projetar sistemas de modo que eles possam evoluir de acordo.

Para projetar o sistema de maneira que seja robusto em face de tais mudanças, você deve levar em conta como o sistema pode necessitar mudar ao longo de sua vida. Um projeto que não leva em consideração a possibilidade de mudanças está sujeito ao risco de uma grande reformulação no futuro. Essas mudanças podem envolver redefinições e novas implementações de classes, modificação de clientes e novos testes do sistema. A reformulação afeta muitas partes de um sistema de software e, invariavelmente, mudanças não-antecipadas são caras.

Se você conhece o PAF-ECF desde sua origem, sabe muito bem do que estamos falando. Um sistema que evolui constantemente, se mal projetado, deverá ser refeito mais cedo ou mais tarde.

Solução de Problemas

Projetando Visando a Evolução do Sistema

Os padrões de projeto ajudam a evitar esses problemas ao garantirem que o sistema possa mudar segundo maneiras específicas.

Cada padrão de projeto permite a algum aspecto da estrutura do sistema variar independentemente de outros aspectos, desta forma tornando um sistema mais robusto em relação a um tipo particular de mudança.

Veremos agora algumas causas comuns que fazem com que um sistema seja refeito e quais padrões poderiam ser utilizados para evitar isso.

Não se preocupe de antemão com os nomes dos padrões que serão apresentados. Posteriormente descreverei para que serve cada um deles.

Solução de Problemas

Projetando Visando a Evolução do Sistema

Criando um objeto pela especificação explícita de uma classe.

Especificar um nome de uma classe quando você cria um objeto faz com que se comprometa com uma implementação em particular, em vez de se comprometer com uma determinada interface. Este compromisso pode complicar futuras mudanças. Para evitá-lo, crie objetos indiretamente.

Padrões de projeto: Abstract Factory, Factory Method, Prototype.

Solução de Problemas

Projetando Visando a Evolução do Sistema

Dependência de operações específicas.

Quando você especifica uma operação em particular, se compromete com uma determinada maneira de atender a uma solicitação. Evitando solicitações codificadas inflexivelmente (hard-coded), você torna mais fácil mudar a maneira como uma solicitação é atendida, tanto em tempo de compilação como em tempo de execução.

Padrões de projeto: Chain of Responsibility, Command.

Solução de Problemas

Projetando Visando a Evolução do Sistema

Dependência da plataforma de hardware e software.

As interfaces externas do sistema operacional e as interfaces de programação de aplicações (APIs) são diferentes para diferentes plataformas de hardware e software. O software que depende de uma plataforma específica será mais difícil de portar para outras plataformas. Pode ser até mesmo difícil mantê-lo atualizado na sua plataforma nativa. Portanto, é importante projetar o seu sistema para a limitar suas dependências de plataformas.

Padrões de projeto: Abstract Factory, Bridge.

Solução de Problemas

Projetando Visando a Evolução do Sistema

Dependência de representações ou implementações de objetos.

Clientes que precisam saber como um objeto é representado, armazenado, localizado ou implementado podem necessitar ser alterados quando esse objeto muda. Ocultar essas informações dos clientes evita a propagação de mudanças em cadeia.

Padrões de projeto: Abstract Factory, Bridge, Memento, Proxy.

Solução de Problemas

Projetando Visando a Evolução do Sistema

Dependências algorítmicas.

Os algoritmos são frequentemente estendidos, otimizados e substituídos durante desenvolvimento e reutilização. Os objetos que dependem de algoritmos terão que mudar quando o algoritmo mudar. Portanto os algoritmos que provavelmente mudarão deveriam ser isolados.

Padrões de projeto: Builder, Iterator, Strategy, Template, Method, Visitor.

Padrões de Projeto

Solução de Problemas

Projetando Visando a Evolução do Sistema

Acoplamento forte.

Classes que são fortemente acopladas são difíceis de reutilizar isoladamente, uma vez que dependem umas das outras. O acoplamento forte leva a sistemas monolíticos, nos quais você não pode mudar ou remover uma classe sem compreender e mudar muitas outras classes. O sistema torna-se uma massa densa difícil de aprender, portar e manter. Um acoplamento fraco aumenta a probabilidade de que uma classe possa ser usada por si mesma e de que um sistema possa ser aprendido, portado, modificado e estendido mais facilmente. Os padrões de projeto usam técnicas como acoplamento abstrato e projeto em camadas para obter sistemas fracamente acoplados.

Padrões de projeto: Abstract Factory, Bridge, Chain of Responsibility, Command, Façade, Mediator, Observer.

Solução de Problemas

Projetando Visando a Evolução do Sistema

Estendendo a funcionalidade pelo uso de subclasses.

A composição de objetos, em geral, e a delegação, em particular, fornecem alternativas flexíveis à herança para a combinação de comportamentos. Novas funcionalidades podem ser acrescentadas a uma aplicação pela composição de objetos existentes de novas maneiras, em vez de definir novas subclasses a partir das classes existentes. Por outro lado, o uso intenso da composição de objetos pode tornar os projetos menos compreensíveis. Muitos padrões de projeto produzem arquiteturas (designs) nas quais você pode introduzir uma funcionalidade customizada simplesmente pela definição de uma subclasse e pela composição de suas instâncias com as existentes.

Padrões de projeto: Bridge, Chain of Responsibility, Composite, Decorator, Observer, Strategy.

Solução de Problemas

Projetando Visando a Evolução do Sistema

Incapacidade para alterar classes de modo conveniente.

Algumas vezes você tem que modificar uma classe que não pode ser convenientemente modificada. Talvez necessite do código-fonte e não disponha do mesmo (como pode ser o caso em se tratando de uma biblioteca comercial de classes). Ou, talvez, qualquer mudança possa requerer a modificação de muitas subclasses existentes. Padrões de projeto oferecem maneiras para modificações de classes em tais circunstâncias.

Padrões de projeto: Adapter, Decorator, Visitor.

Como Selecionar um Padrão

Com vários padrões de projeto catalogados para se escolher, pode ser difícil encontrar aquele que trata um problema de projeto particular. Seguem algumas abordagens que ajudam a encontrar o padrão de projeto correto para o problema em questão.

- Comece examinando as intenções de cada padrão. Leia a intenção de cada padrão para encontrar um ou mais padrões que pareçam relevantes para o seu problema.
- Estude como os padrões se inter-relacionam O estudo desses relacionamentos pode ajudar a direcioná-lo para o padrão, ou grupo de padrões mais adequado à seu problema.
- Estude padrões de finalidades semelhantes.
- Considere o que deveria ser variável no seu projeto. Esta abordagem é o oposto de se focalizar nas causas de reformulação. Ao invés de considerar o que pode forçar uma mudança em um projeto, considere o que você quer seja capaz de mudar sem reprojetar. O foco, aqui, é posto sobre o encapsulamento do conceito que varia, um tema de muitos padrões de projeto. Dessa forma, eles podem ser mudados sem necessidade de reformulação de projeto.

Padrões de Projeto

Catálogos

Os padrões costumam ser organizados em catálogos. Um dos mais conhecidos é o catálogo de padrões da gangue dos quatro (GoF). No entanto, existem muitos outros.

Iremos analisar, além dos padrões GoF, os padrões GRASP (padrões que descrevem os princípios fundamentais da atribuição de responsabilidades a objetos).

Os padrões de projeto variam na sua granularidade e no seu nível de abstração. Como existem muitos padrões de projeto, é necessário organizá-los. Veremos como classificar os padrões de projeto de maneira que possamos nos referir a famílias de padrões relacionados.

A classificação ajuda a aprender os padrões mais rapidamente, bem como direcionar esforços na descoberta de novos.

Padrões de Projeto

Catálogos | GoF

GoFPatterns

Os padrões de projeto GoF são soluções genéricas para os problemas mais comuns do desenvolvimento de software orientado a objetos. Foram coletados de experiências de sucesso na indústria de software, principalmente de projetos em C++ e SmallTalk. Seus padrões são classificados por dois critérios:

O primeiro critério é a finalidade - reflete o que um padrão faz. Os padrões podem ter finalidades de criação, comportamento e estrutura. Os padrões de criação descrevem as técnicas para instanciar objetos (ou grupos de objetos), e possibilitam organizar classes e objetos em estrutura maiores, os padrões de comportamento se caracterizam pela maneira pelas quais classes ou objetos interagem e distribuem responsabilidades e por fim os padrões de estrutura lidam com a composição de classes ou objetos.

Padrões de Projeto

Catálogos | GoF

O segundo critério, chamado escopo, especifica se o padrão se aplica primariamente a classes ou a objetos. Os padrões para classes lidam como os relacionamentos entre classes e suas subclasses. Esses relacionamentos são estabelecidos através do mecanismo de herança, assim eles são estáticos fixados em tempo de compilação. Os padrões para objetos lidam com relacionamentos entre objetos que podem ser mudados em tempo de execução e são mais dinâmicos. Quase todos utilizam a herança em certa medida. Note que a maioria está no escopo de objeto.

Padrões de Criação

São aqueles que abstraem e ou adiam o processo de criação dos objetos. Eles ajudam a tornar um sistema independente de como seus objetos são criados, compostos e representados. Um padrão de criação de classe usa a herança para variar a classe que é instanciada, enquanto que um padrão de criação de objeto delegará a instanciação para outro objeto.

Padrões de Projeto

Catálogos | GoF

GoFPatterns

Padrões de Criação

Os padrões de criação tornam-se importantes à medida que os sistemas evoluem no sentido de dependerem mais da composição de objetos do que a herança de classes. O desenvolvimento baseado na composição de objetos possibilita que os objetos sejam compostos sem a necessidade de expor o seu interior como acontece na herança de classe, o que possibilita a definição do comportamento dinamicamente e a ênfase desloca-se da codificação de maneira rígida de um conjunto fixo de comportamentos, para a definição de um conjunto menor de comportamentos que podem ser compostos em qualquer número para definir comportamentos mais complexos.

Há dois temas recorrentes nesses padrões. Primeiro todos encapsulam conhecimento sobre quais classes concretas são usadas pelo sistema. Segundo ocultam o modo como essas classes são criadas e montadas. Tudo que o sistema sabe no geral sobre os objetos é que suas classes são definidas por classes abstratas.

Padrões de Projeto

Catálogos | GoF

GoFPatterns

Padrões Estruturais

Tais padrões se preocupam com a forma como as classes e os objetos são compostos para formar estruturas maiores.

Os padrões estruturais de classes utilizam a herança para compor interfaces ou implementações, e os padrões estruturais de objeto ao invés de compor interfaces ou implementações, descrevem maneiras de compor objetos para obter novas funcionalidades.

A flexibilidade obtida pela composição de objetos provém da capacidade de mudar a composição em tempo de execução o que não é possível com a composição estática (herança de classes).

Padrões de Projeto

Catálogos | GoF

Padrões Comportamentais

Os padrões de comportamento se concentram nos algoritmos e atribuições de responsabilidades entre os objetos. Eles não descrevem apenas padrões de objetos ou de classes, mas também os padrões de comunicação entre os objetos.

Os padrões comportamentais de classes utilizam a herança para distribuir o comportamento entre classes, e os padrões de comportamento de objeto utilizam a composição de objetos em contrapartida a herança. Alguns descrevem como grupos de objetos orientados a objetos e esperam para a execução de uma tarefa que não poderia ser executada por um objeto sozinho.

A tabela na próxima página ilustra como os padrões de projetos GoF foram organizados.

Padrões de Projeto

Catálogos | GoF

Gang Of Four Design Patterns

GoFPatterns

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

Padrões de Projeto

Catálogos | GoF | Padrões de Criação

GoFPatterns

- Abstract Factory - Provê uma interface para criar famílias de objetos relacionados ou interdependentes sem especificar suas classes concretas.
- Builder - Separa a construção de um objeto complexo da sua representação de forma que o mesmo processo de construção possa criar representações diferentes.
- Factory Method - Define uma interface para criar um objeto, mas deixa as subclasses decidirem qual classe instanciar. O padrão Factory Method deixa uma classe repassar a responsabilidade de instanciação para subclasses.
- Prototype - Especifica os tipos de objetos a criar usando uma instancia protótipo e cria novos objetos copiando este protótipo.
- Singleton - Garante que uma classe tenha uma única instância e provê um ponto global de acesso à instância.

Padrões de Projeto

Catálogos | GoF | Padrões Estruturais

- Adapter - Converte a interface de uma classe em outra interface com a qual os clientes estão prontos para lidar. Permite que classes trabalhem juntas apesar de interfaces incompatíveis.
- Bridge - Desacopla uma abstração de sua implementação de forma que as duas possam mudar independentemente uma da outra.
- Composite - Compõe objetos em estruturas de árvore para representar hierarquias do tipo Parte-Todo. Permite que clientes tratem objetos individuais e composições de objetos de maneira uniforme.
- Decorator - Adiciona responsabilidades a um objeto dinamicamente. Decoradores proveem uma alternativa flexível à herança para estender funcionalidade.
- Façade - Provê uma interface unificada para um conjunto de interfaces num subsistema. O padrão Façade define uma interface de mais alto nível, deixando o subsistema mais fácil de usar.
- Flyweight - Usa o compartilhamento para dar suporte eficiente ao uso de um grande número de objetos de granularidade pequena.
- Proxy - Provê um objeto procurador para outro objeto controlar o acesso a ele.

Padrões de Projeto

Catálogos | GoF | Padrões Comportamentais

- Chain of Responsibility - Evita acoplar o remetente de um pedido ao receptor dando oportunidade a vários objetos para tratarem do pedido. Os objetos receptores são encadeados e o pedido é passado na cadeia até que um objeto o trate.
- Command - Encapsula um pedido num objeto, permitindo assim parametrizar clientes com pedidos diferentes, enfileirar pedidos, fazer log de pedidos, e dar suporte a operações de undo.
- Interpreter - Dada uma linguagem, define uma representação de sua gramática e um interpretador que usa a representação da gramática para interpretar sentenças da linguagem.
- Iterator - Provê uma forma de acessar os elementos de uma coleção de objetos sequencialmente sem expor sua representação subjacente.
- Mediator - Define um objeto que encapsule a forma com a qual um conjunto de objetos interagem. O padrão Mediator promove o acoplamento fraco evitando que objetos referenciem uns aos outros explicitamente e permite que suas interações variem independentemente.
- Memento - Sem violar o princípio de encapsulamento, captura e externaliza o estado interno de um objeto de forma a poder restaurar o objeto a este estado mais tarde.

Padrões de Projeto

Catálogos | GoF | Padrões Comportamentais

GoFPatterns

- Observer - Define uma dependência um-para-muitos entre objetos de forma a avisar e atualizar vários objetos quando o estado de um objeto muda.
- State - Permite que um objeto altere seu comportamento quando seu estado interno muda. O objeto estará aparentemente mudando de classe com a mudança de estado.
- Strategy - Define uma família de algoritmos, encapsula cada um, e os deixa intercambiáveis permitindo que o algoritmo varie independentemente dos clientes que o usem.
- Template Method - Define o esqueleto de um algoritmo numa operação, deixando que as subclasses completem algumas das etapas do algoritmo sem alterar a sua estrutura.
- Visitor – Representa uma operação a ser realizada nos elementos de uma estrutura de objetos. O padrão Visitor permite que se defina uma nova operação sem alterar as classes dos elementos nos quais a operação age.

A imagem na página seguinte podemos observar os relacionamentos existentes entre os padrões.

Padrões de Projeto

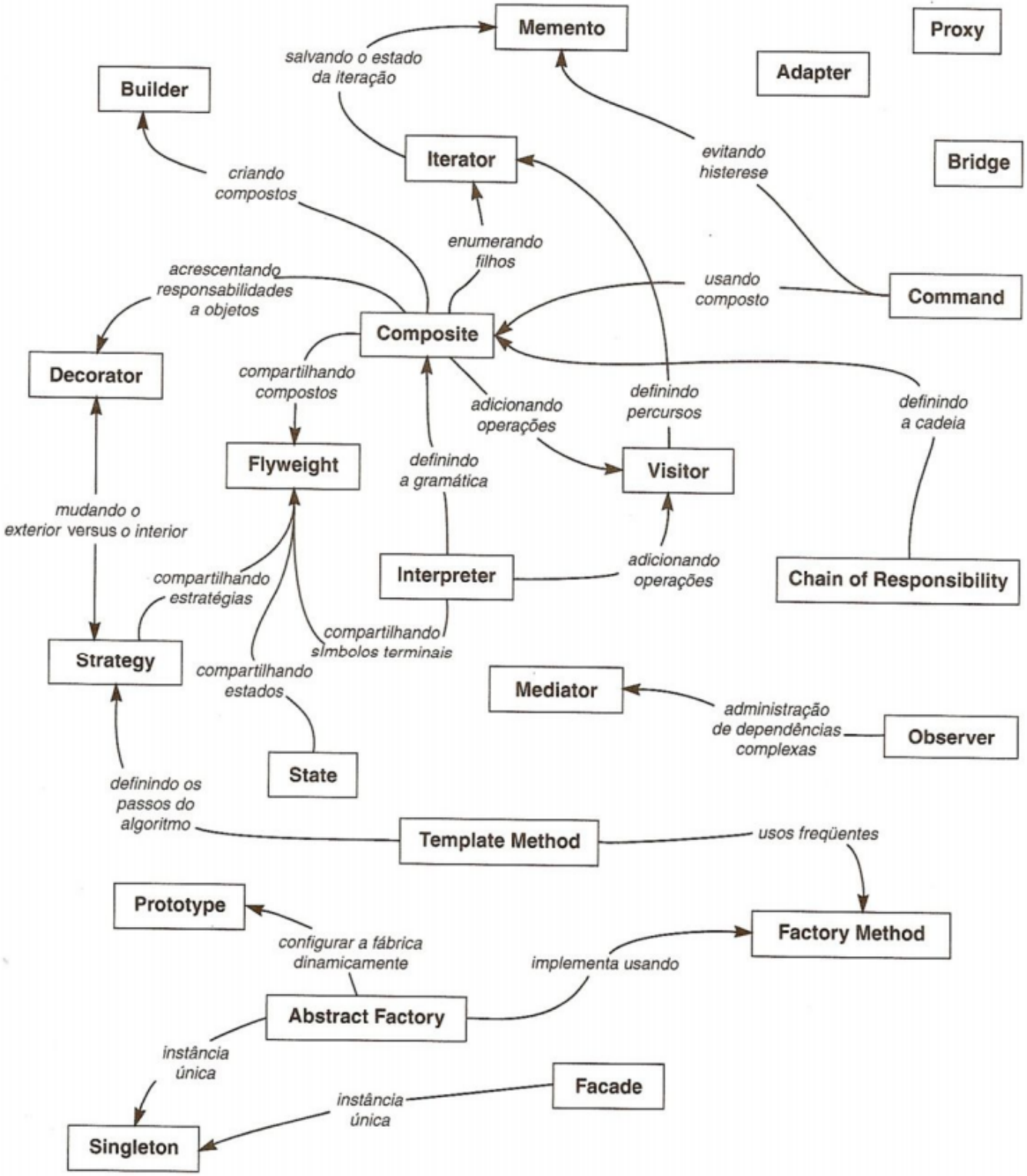
Catálogos | GoF

Alguns padrões são frequentemente usados em conjunto.

Por exemplo, o Composite é frequentemente usado com o Iterator ou o Visitor.

Alguns padrões são alternativos: o Prototype é frequentemente uma alternativa para o Abstract Factory.

Alguns padrões resultam em projetos semelhantes, embora tenham intenções diferentes.



Padrões de Projeto

Catálogos | GRASP

Os padrões GRASP (General Responsibility and Assignment Software Patterns - Padrões Genéricos de Software para Atribuição de Responsabilidade), foram catalogados por Craig Larman em seu livro Utilizando UML e Padrões.

Conforme o nome já diz, tais padrões têm o objetivo de tornar a distribuição de responsabilidade entre as classes uma tarefa mais criteriosa e eficiente, melhorando de forma significativa a qualidade de seu projeto e reduzindo os problemas de arquitetura mais comuns.

Enquanto o GoF explora as soluções mais especificas o GRASP refletem as praticas mais pontuais da aplicação das técnicas de OO.

O seu estudo ajuda definir responsabilidades às classes do projeto, que ocorre frequentemente durante a fase de criação dos diagramas de iteração e o entendimento dos princípios de um bom projeto orientado a objetos.

Catálogos | GRASP

A maioria dos programadores aprendeu que em POO devemos mapear objetos reais para objetos em seu modelo de classes.

Isto muitas vezes é seguido à risca, mas nem sempre é a melhor maneira de modelar suas classes, pois muitos detalhes cruciais só são visíveis ao se considerar o contexto geral, ou seja, a forma como esses objetos vão interagir e como eles vão colaborar uns com os outros.

A qualidade de um projeto orientado a objeto está fortemente relacionada com a distribuição de responsabilidade. Os padrões GRASP surgiram exatamente para auxiliar nesta questão.

As responsabilidades podem ser descritas como obrigações e podem ser divididas em conhecer e fazer.

Catálogos | GRASP

As obrigações de conhecimento estão relacionadas à distribuição das características do sistema entre as classes:

- Conhecer algo sobre dados privativos e encapsulados.
- Conhecer algo sobre objetos relacionados.
- Conhecer algo sobre coisas que pode calcular ou derivar.

As obrigações de realização estão relacionadas à distribuição das características do sistema entre as classes:

- Fazer alguma coisa em si mesmo.
- Iniciar uma ação em outro objeto.
- Controlar e coordenar atividades em outros objetos.

Catálogos | GRASP

Exemplo: uma classe Venda pode ter obrigação de gerar pedidos (faz algo a si mesma), ou pode utilizar uma classe Pedido para isso (inicia uma ação em outro objeto) e controlar essas ações mantendo um contador de pedidos (informações encapsuladas). Para isso, a Venda deve conhecer a classe Pedido (objetos relacionados).

A principal característica da atribuição de responsabilidades está em não sobrecarregar os objetos com responsabilidades que poderiam ser delegadas. O objeto só deve fazer o que está relacionado com a sua abstração. Para isso, delega as demais atribuições para quem está mais apto a fazer. Quando o objeto não sabe quem é o mais apto, pergunta para algum outro objeto que saiba.

Os padrões GRASP são classificados como fundamentais e avançados.

Catálogos | GRASP | Padrões Fundamentais

Especialista (Expert)

Um princípio geral de projetos de objetos e de atribuição de responsabilidades. Atribua uma responsabilidade ao especialista na informação – a classe que tem a informação necessária para satisfazê-la.

Criador (Creator)

Quem cria? (Note que a fábrica é uma alternativa comum). Atribua à classe B a responsabilidade de criar uma instância da classe A se uma das seguintes cinco condições for verdadeira:

[B contém A] [B agrega A] [B tem os dados de iniciação de A]
[B registra A]
[B usa A de maneira muito próxima]

Catálogos | GRASP | Padrões Fundamentais

Controlador (Controller)

Quem trata de um evento do sistema? Atribua a responsabilidade pelo tratamento de uma mensagem de evento do sistema a uma classe que represente uma das seguintes opções:

- Representa todo o sistema, dispositivo ou subsistemas (controlador de fachada)
- Representa um cenário de um caso de uso dentro do qual ocorra
- evento do sistema (controlador de caso de uso ou sessão)

Catálogos | GRASP | Padrões Fundamentais

Acoplamento Fraco (de avaliação) (Low Coupling)

Como favorecer a dependência baixa e aumentar a reutilização? Atribua responsabilidade de modo que o acoplamento (desnecessário) permaneça baixo.

Coesão Alta (de avaliação) (High Cohesion)

Como manter a complexidade controlável? Atribua responsabilidade de modo que a coesão permaneça alta.

Catálogos | GRASP | Padrões Avançados

Polimorfismo (Polymorphism)

Quem é responsável quando o comportamento varia segundo o tipo? Quando alternativas ou comportamentos relacionados variam segundo o tipo (classe), atribua a responsabilidade pelo comportamento – usando operações polimórficas – aos tipos para os quais o comportamento varia.

Invenção Pura (Pure Fabrication)

Qual objeto é responsável quando você está desesperado e não quer violar a coesão alta e o acoplamento baixo?

Atribua um conjunto de responsabilidades altamente coeso a uma classe de comportamento artificial ou de conveniência, que não represente um conceito no domínio do problema – algo inventado, para suportar a coesão alta, o acoplamento baixo e a reutilização.

Catálogos | GRASP | Padrões Avançados

Indireção (Indirection)

Como atribuir responsabilidades para evitar o acoplamento direto?

Atribua a responsabilidade a um objeto intermediário para ser o mediador entre outros componentes ou serviços, para que eles não sejam diretamente acoplados.

Variações protegidas (Protected Variations)

Como atribuir responsabilidades a objetos, subsistemas e sistemas, de modo que as variações ou a instabilidade nesses elementos não tenham um impacto indesejável sobre outros elementos?

Identifique pontos de variação ou instabilidades prevista, atribua responsabilidades para criar uma interface estável em torno deles.

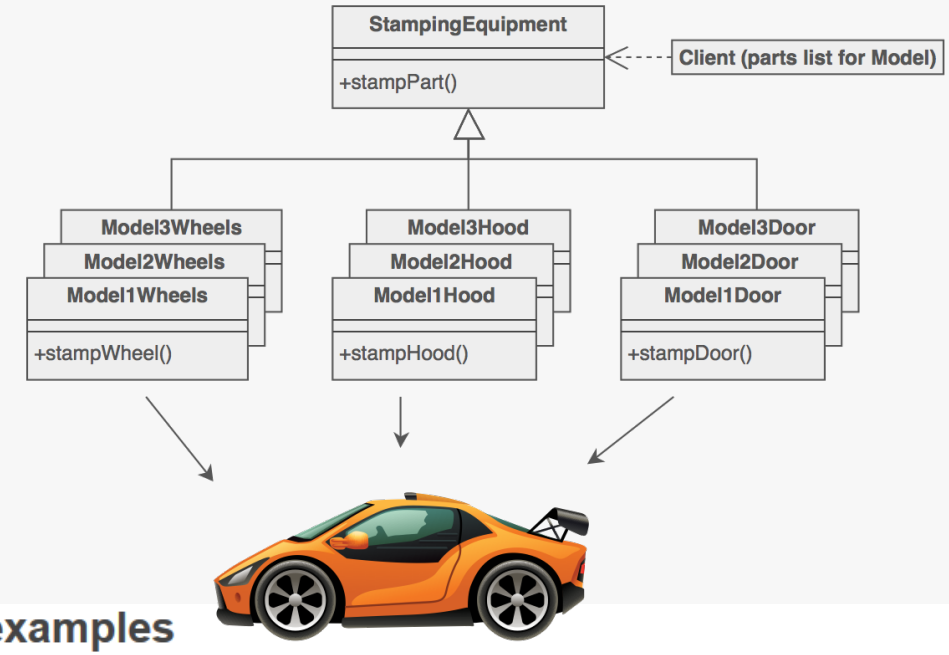
Padrões de Projeto

Estudando os Padrões

Agora que você teve uma visão geral dos padrões de projeto, já deve ter percebido que eles não são triviais. É preciso estudá-los com afinco para compreender como utilizá-los de acordo com a necessidade nos projetos.

Para ajudar nos estudos, vou sugerir um site bem interessante: o SourceMaking. Clique na imagem abaixo para acessar o site.

Quando acessar o site, você verá que existe um link para cada tipo de padrão. Ao clicar no link, você será direcionado para uma página que descreve o padrão, que mostra um exemplo com um diagrama de classes e no final traz códigos em várias linguagens. Divirta-se!



Code examples

Java	Abstract Factory in Java	Abstract Factory in Java
C++	Abstract Factory in C++	Abstract Factory in C++: Before and after
PHP	Abstract Factory in PHP	Abstract Factory in PHP
Delphi	Abstract Factory in Delphi	

Referências

Craig Larman, 1999, “Utilizando UML e Padrões”, 1ª ed., Prentice-Hall
Craig Larman, 2007, “Utilizando UML e Padrões”, 3ª ed., Bookman.
Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (1995). Design Patterns. Elements of Reusable Object-Oriented Software 1 ed. Addison-Wesley
Marinescu, Floyd (2002). EJB Design Patterns: Advanced Patterns, Processes and Idioms. John Wiley & Sons.
Martin, Robert C. "Princípios de Projeto e Padrões de Projeto". 2000.
SOMMERVILLE, IAN. Engenharia de Software. 8. ed. Addison Wesley, 2007.
WIKIPEDIA, Acessos em Mar/2019.
W3C, Acesso em Mar/2019.