SUMMARY SERIES

direto ao ponto

Padrões de Projeto

Soluções reutilizáveis de software orientado a objetos

SERGIO CABRAL

sergiocabral.com

Índice

O que segue é um resumo do livr *Padrões de Projeto: Solucoes Reutilizaveis de Software Orientado a Objetos*, por Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides. Sendo esses conhecidos na internet como *Gang of Four* (GoF), ou *Gang dos Quatro*. Foi utilizado a versão brasileira, ano 2000, reimpressão de 2008, traduzida por Luiz A. Meirelles Salgado. Editora *Bookman*.

O texto tem **dois níveis de detalhe**. Para uma leitura ainda mais resumida pule por entre os destaques em amarelo. Eles são conectados um ao outro mantendo a fluidez da leitura. E caso sinta falta de algum entendimento poderá ler a matéria circundante.

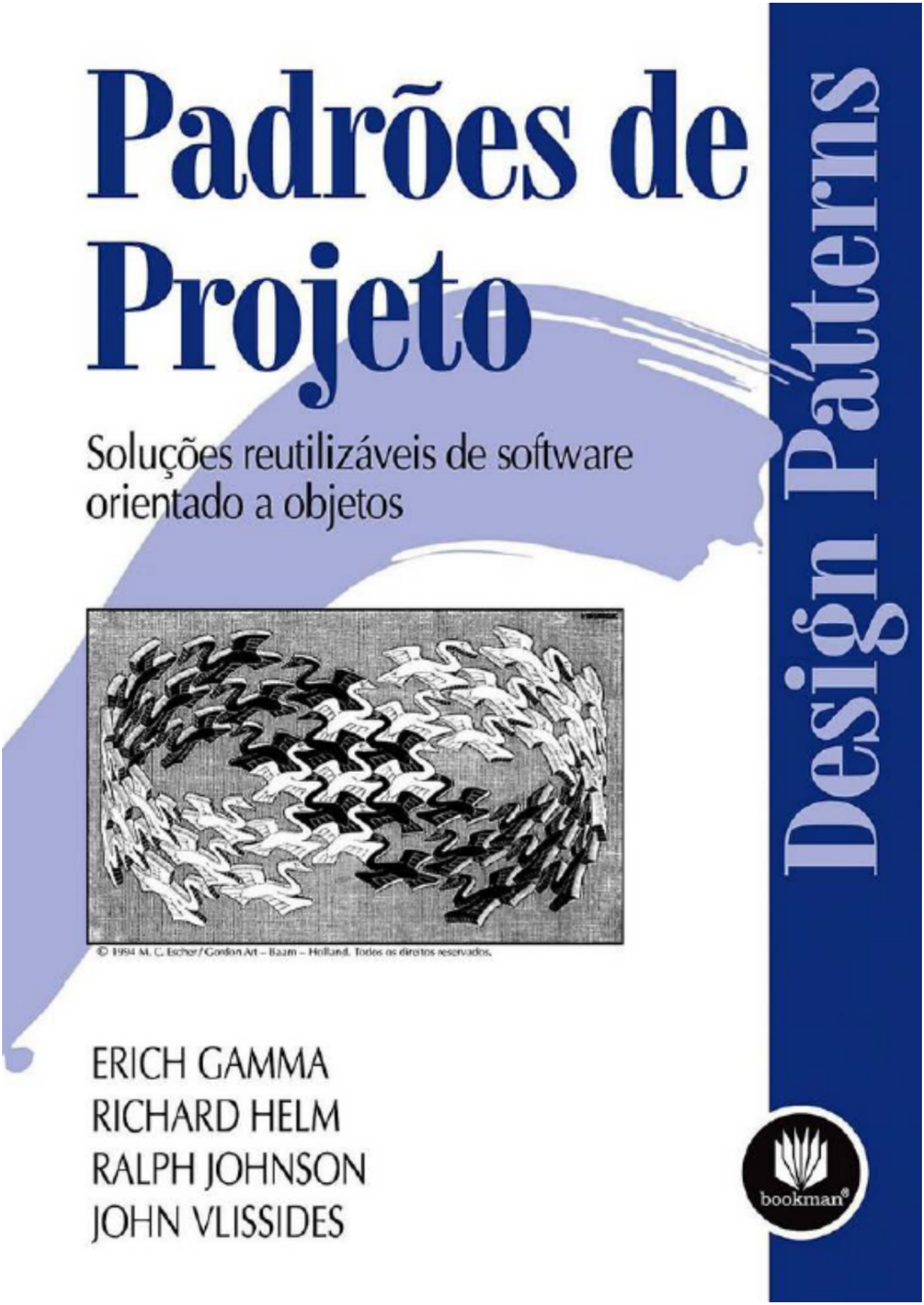

Figura 1. Capa do livro original

Figura 2. Capa interna do livro original

G193 Gamma, Erich
Padrões de projeto: soluções reutilizáveis de software orientado a objetos / Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides; trad. Luiz A. Meirelles Salgado. – Porto Alegre: Bookman, 2000

ISBN 978-85-7307-610-3

1. Engenharia de sistemas – Programação de computadores. I. Helm, Richard. II. Johnson, Ralph. III. Vlissides, John. IV. Título.

CDU 681.3.02

Figura 3. Informações ISBN do livro original

Figura 4. Dados de copyright do livro original

Obra originalmente publicada sob o título
Design patterns – elements of reusable object-oriented software

ISBN 0 - 201- 63361- 2

Capa: Mario Röhnelt (com ilustração de M. C. Escher/Cordon Art/Holanda)

Preparação do original: *Rafael Corsetti*

Supervisão editorial: *Arysinha Jacques Affonso*

Editoração eletrônica: *Laser House*

Figura 5. Dados da publicações do livro original em inglês

Prefácio

O livro não é para iniciantes na programação, mas é destinado à quem já tem vivência com orientação a objetos.

Não tem por objetivo mostrar código-fonte, mas explicar, com um grau de abstração, soluções para problemas conhecidos.

Nos seus sistemas de software os padrões de projeto vão contribuir com visam maior reutilização e flexibilidade. Vai facilitar a modularização e compreensão do projeto.

Apresentação

> Arquiteturas, seja na construção de cidades ou softwares, quando bem construídas fazem uso de padrões. Tais padrões contribuem para uma arquitetura menor, mais simples e compreensível.
>
> Este livro faz duas importantes contribuições: 1) mostrar que padrões influenciam positivamente na arquitetura de sistemas complexos e 2) servir de catálogo para padrões bem concebidos que o desenvolvedor poderá aplicar na criação de suas próprias aplicações.
>
> — Grady Booch, Cientista-Chefe, Rational Software Corporation

Guia para os leitores

- **Capítulos 1 e 2** (primeira parte)
  - O que são padrões de projeto?
  - Como ajudam a projetas software orientado a objetos?
  - Exemplos práticos
- **Capítulos 3, 4 e 5** (segunda parte)
  - Compõe a maior parte do livro com o catálogo de padrões de projetos.

1. Introdução

Projetar um software reutilizável é muito difícil. Ele deve resolver o problema em específico, mas também deve ser genérico o suficiente para atender requisitos futuros sem que seja necessário refazer o projeto original.

Novos e experientes projetistas sabem realizar um projeto. Ambos estão diante das mesma opções a escolher, mas os novos acabam escolhendo mal.

Os melhores projetistas reutilizam soluções que deram certo no passado, sem ter que recomeçar do zero. Se valem da experiência anterior.

Os padrões de projeto apresentam técnicas testadas e aprovadas. Escolher a eles evitam comprometer a reutilização. Favorecem a documentação e manutenção porque fornecem uma especificação clara das interações entre classes, objetos e seus objetivos.

Todos os padrões de projeto neste livro foram testados mais de uma vez. Alguns nunca haviam sido documentados antes, embora fizesse parte do folclore da comunidade de desenvolvedores.

1.1. O que é um padrão de projeto?

Foi dito a respeito de padrões em construções e cidades, mas se aplica aos padrões orientados a objetos:

> Cada padrão de projeto 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.
>
> — Christopher Alexandre, AIS+77, pág. x

Um padrão tem quatro elementos essenciais:

1. **Nome**
    a. O nome facilita a comunicação entre colegas, já que reúne em uma palavra os três elementos a seguir: problema, solução e consequência.
2. **Problema**
    a. Descreve o contexto da situação em que se aplica o padrão para solucionar um dado problema.
    b. Em alguns casos incluirá uma lista de condições que devem ser satisfeitas para que se faça sentido aplicar a solução.
3. **Solução**
    a. Não descreve algo concreto ou uma implementação específica, pois se trata de um gabarito que pode ser aplicado em muitas situações.
    b. Apresenta como um arranjo de classes e objetos é capaz de resolver o problema.
4. **Consequências**

a. Avaliação das vantagens e desvantagens, *trade-offs*. Tem relação com o impacto na flexibilidade, extensibilidade ou portabilidade de um sistema.

O conceito de padrão de projeto pode variar de uma pessoa para outra, mas neste liv ***"são descrições de objetos e classes comunicantes que precisam ser personalizadas para resolver um problema geral de projeto num contexto particular"*** .

Os exemplos neste livro são em C++ ou Smalltalk porque fazem parte da experiência prática dos autores. Os padrões neste livro não cabem numa implementação através de linguagens procedurais como C, Pascal, Ada, etc.

A escolha da linguagem por quem vai usar o padrão de projeto influencia o que pode, ou não, ser implementado facilmente. O uso linguagens orientadas a objetos torna desnecessário incluir padrões de projeto como *Herança*, *Encapsulamento* e *Polimorfismo*, já que fazem parte da base dessas linguagens, diferente das linguagens procedurais.

1.2. Padrões de projeto no MVC de Smalltalk

MVC é a tríade de classes abaixo.

1. ***Model*** **(Modelo)**: É o objeto de aplicação.
2. ***View*** **(Visão)**: É a apresentação na tela.
3. ***Controller*** **(Controlador)**: É a maneira como interface do usuário reage às entradas do mesmo.

Essa separação aumenta a flexibilidade e reutilização. Sem esse padrão os projetos de interface tendem a agrupar tudo em único lugar.

Como mostra a figura a seguir, a separação de visão e modelo permite que uma certa estrutura de dados (Model) possa ter uma ou mais apresentações (View).

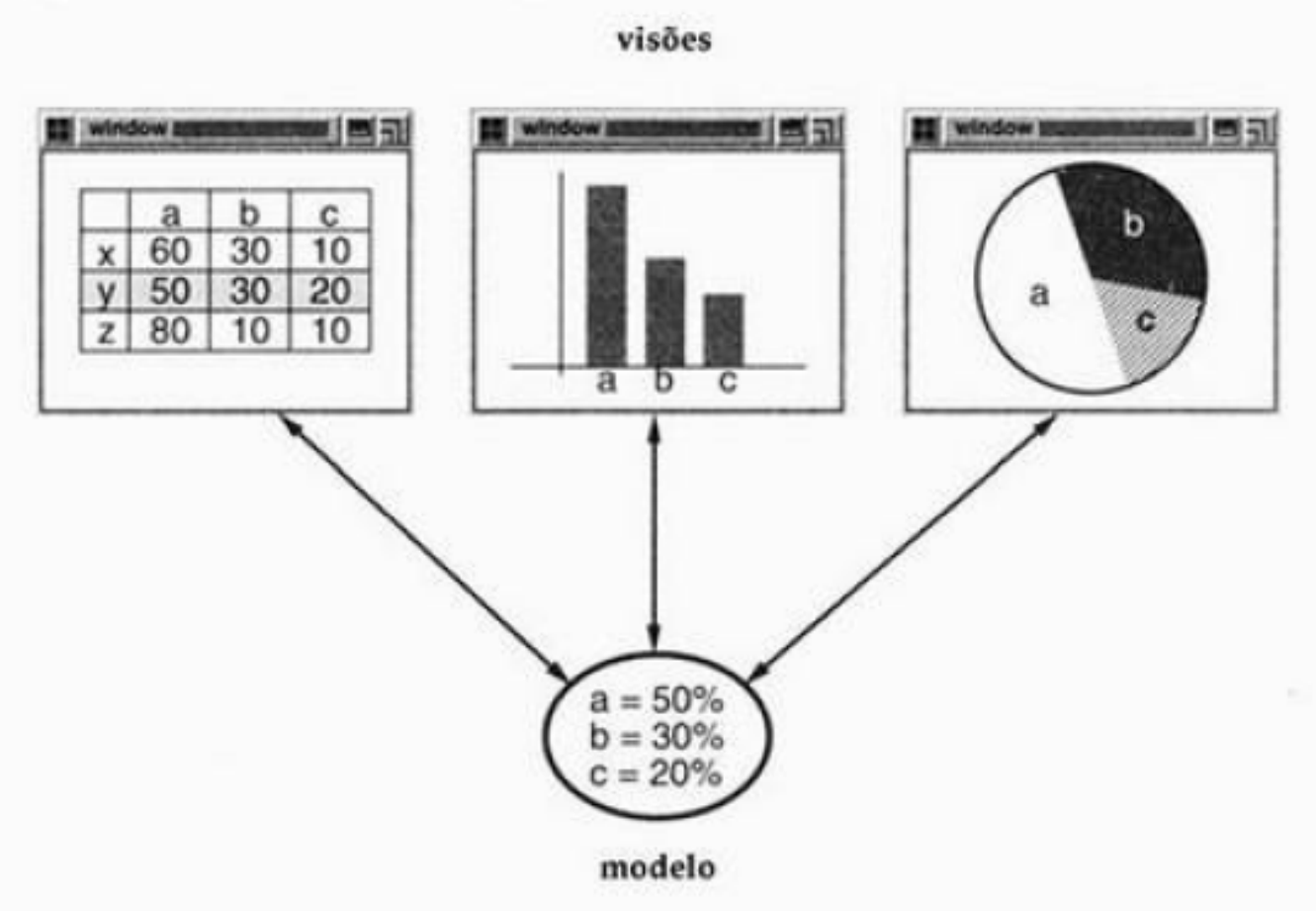

Figura 6. Modelos e Visões

Essa separação é possível através de um protocolo do tipo inscrição/notificação (*subscribe/notify*) entre eles. A *View* se inscreve para receber atualizações do *Model* . O *Model* , quando atualizado, notifica os inscritos, as *Views* . Isso permite que várias *Views* sejam conectadas ao mesmo *Model* .

O padrão *Observer* (visto adiante) faz o mesmo , ao pertimir que mudança em um objeto possam ser refletidas em vários outros objetos, mas sem exigir que o objeto alterado conheça detalhes dos objetos notificados.

Outra característica do MVC é que Views podem ser encaixadas em um CompositeView. Este último funciona como uma View, mas ela contem e administra Views internas, encaixadas.

O padrão *Composite* (visto adiante) tem essa abordagem , quando queremos agrupar objetos e tratar o grupo como um objeto individual. Permite criar uma hierarquia, já que um CompositeView pode conter tanto Views como outros CompositeViews.

No MVC, uma *View* usa um *Controller* para implementar a maneira como responde ao usuário. Para mudar a estratégia de resposta substitua o *Controller* por outro. Por exemplo, um *Controller* aceita entrada através do teclado ao passo que outro através de um menu pop-up. De modo que pode haver vários *Controllers* para uma mesma *View* .

O relacionamento *View - Controller* é um exemplo do padrão *Strategy* (visto adiante). Um *Strategy* é um objeto que representa um algoritmo. No MVC, cada *Controller* atua como um *Strategy* quando são utilizados pela mesma *View*.

O MVC pode utilizar outros padrões que serão vistos mais adiante. Por exemplo o *Factory Method*

ao ter que especificar um *Controller* padrão para uma *View* que não especificou um. Também o *Decorator* para acrescentar capacidades a uma *View*, por exemplo, incluir barra de rolagem.

Em resumo, os relacionamentos no MVC são fornecidos pelos padrões:

- Principalmente:
  - *Observer*
  - *Composite*
  - *Strategy*
- Também:
  - *Factory Method*
  - *Decorator*

1.3. Descrevendo os padrões de projeto

Descrever um padrão com desenhos são importantes, mas não o suficiente. Isso porque capturam o produto final quanto ao relacionamento entre classes e objetos. É necessário mais do que isso. Este livro segue o gabarito a seguir para descrever cada padrão de projeto abordado:

Nome e classificação

Expressa a essência do padrão de forma resumida. Trata-se de um nome que será incorporado ao seu vocabulário.

Intenção e objetivo

Uma curta declaração para responder:

- O que faz?
- Quais seus princípios e intenções?
- Que problema tenta tratar?

Também conhecido como

Outros nomes bem conhecidos para o padrão, se existirem.

Motivação

Um cenário que concretiza as descrições mais abstratas a seguir. Trata-se de um exemplo que mostra como um problema é resolvido com a estrutura de classes e objetos do padrão.

Aplicabilidade

- Onde pode ser aplicado?
- Que maus projetos ele pode tratar?
- Como reconhecer essas situações?

Estrutura

Uma representação gráfica das classes do padrão.

Participantes

As classes e objetos que participam do padrão e suas responsabilidades.

Colaborações

Como as classes colaboram para executar suas responsabilidades.

Consequências

- Como o padrão suporta a realização de seus objetivos?
- Custos versus Benefícios
- Que aspecto da estrutura de um sistema ele permite variar de forma independente?

Implementação

Quais são as armadilhas, sugestões ou técnicas que você precisa saber? Existem considerações específicas de linguagem?

Exemplo de código

Exemplos de código-fonte em C++ ou Smalltalk.

Usos conhecidos

Exemplos do padrão em sistemas reais. São incluídos pelos menos dois exemplos de domínios diferentes.

Padrões relacionados

Outros padrões intimamente relacionados. Quais diferenças entre eles? Com quais outros padrões este deveria ser utilizado?

1.4. O catálogo de padrões de projeto

Segue uma visão geral do catálogo neste livro com seus 23 padrões de projetos:

Abstract Factory

Interface para criação de famílias de objetos relacionados ou dependentes sem especificar suas classes concretas.

Adapter

Converte a interface de uma classe para que seja compatível com outra classe que a utilizará. Ela permite que classes incompatíveis, por conta da interface, possam trabalhar em conjunto.

Bridge

Separa uma abstração da sua implementação, de modo que as duas possam variar independentemente.

Builder

Separa a construção de um objeto complexo da sua representação, de modo que o mesmo processo de construção possa criar diferentes representações.

Chain of Responsability

Evita o acomplamento do remetente de uma solicitação ao seu destinatário. Dá a mais de um objeto a chance de tratar a mesma solicitação. Encadeia os objetos receptores e passa a solicitação ao longo da cadeia até que um objeto a trate.

Command

Encapsula uma solicitação como um objeto. Isso permite 1) parametrizar clientes com diferentes solicitações, 2) enfileirar ou registrar (tipo *log*) solicitações ou 3) suportar operações que podem ser desfeitas.

Composite

Compõe objetos em estrutura de árvore para representar hierarquias do tipo Partes-Todo. Permite que os clientes tratem tanto objetos individuais como composições de objetos da mesma maneira.

Decorator

Atribui responsabilidades adicionais a um objeto dinamicamente. Fornecem uma alternativa flexível para estender funcionalidades de subclasses.

Facade

Fornece uma interface unificada para um conjunto de interfaces em um subsistema. Define uma interface de nível mais alto que torna o subsistema mais fácil de usar.

Factory Method

Define uma interface para criar um objeto, mas deixa as subclasses decidirem qual classe será instanciada. Permite a uma classe delegar para as subclasses a criação da instância.

Flyweight

Usa compartilhamento para suportar grandes quantidades de objetos, de granularidade fina, de maneira eficiente.

Intrpreter

Dada uma linguagem, define uma representação para sua gramática juntamente com um interpretador que usa a representação para interpretar sentenças nessa linguagem.

Iterator

Fornece uma maneira de acessar sequencialmente os elementos de uma agregação de objetos sem expor sua representação.

Mediator

Define um objeto que encapsula a forma como um conjunto de objeto interage. Promove um fraco acoplamento visto que cada objeto não faz referência ao outro de maneira explicita, tornando possível varias suas interações de forma independente.

Memento

Sem violar o encapsulamento, captura e externaliza um estado interno de um objeto, de modo que o mesmo possa posteriormente ser restaurado para este estado.

Observer

Define uma dependência de uma-para-muitos de modo que quando um objeto muda de estado todos os seus dependentes são notificados e atualizados.

Prototype

Especifica os tipos de objetos a serem criados usando uma instância que serve de protótipo. Assim novos objetos são criados como cópias do protótipo.

Proxy

Fornece um objeto representante (surrogate), ou um marcador de outro objeto, para controlar o acesso ao mesmo.

Singleton

Garante que uma classe tenha apenas uma instância e fornece um ponto global de acesso a ela

State

Garante que uma classe altere seu comportamento quando seu estado interno muda. O objeto parecerá ter mudado de classe.

Strategy

Define uma família de algoritmos, encapsula cada um deles e os torna intercambiáveis, independentemente dos clientes que o utilizam.

Template Method

Define o esqueleto (estrutura externa) de um algoritmo em uma operação, mas delega para subclasses a implementação de alguns passos (estrutura interna).

Visitor

Representa uma operação a ser executada sobre os elementos de uma estrutura de objetos. Permite definir uma nova operação sem modificar as classes dos elementos.

1.5. Organizando o catálogo

Padrões variam na sua granularidade e nível de abstração. Eles serão agrupados em famílias com base em dois critérios:

1. **Propósito**, ou finalidade
   a. **Criação**: de objetos
   b. **Estrutural**: composição de classes ou objetos
   c. **Comportamental**: interação entre classes e objetos, distribuição de responsabilidades.
2. **Escopo**
   a. **Classe**: relacionamento entre classes e subclasses definidos no tempo de compilação, por exemplo, pela herança.
   b. **Objeto**: mais dinâmicos; modificáveis durante o tempo de execução

		Propósito		
		De criação	Estrutural	Comportamental
Escopo	Classe	Factory Method (112)	Adapter (class) (140)	Interpreter (231) Template Method (301)
	Objeto	Abstract Factory (95) Builder (104) Prototype (121) Singleton (130)	Adapter (object) (140) Bridge (151) Composite (160) Decorator (170) Façade (179) Flyweight (187) Proxy (198)	Chain of Responsibility (212) Command (222) Iterator (244) Mediator (257) Memento (266) Observer (274) State (284) Strategy (292) Visitor (305)

Figura 7. Organização dos padrões de projeto

Existem outras maneiras de organizar os padrões, já que eles relacionam-se entre si. Por exemplo, *Composite* é usado com *Iterator* ou *Visitor*. *Prototype* é uma alternativa para *Abstract Factory*. Os diagrama de estruturas do *Composite* e *Decorator* são semelhantes, mas com intenções diferentes. Também, a parte "Padrões Relacionados", usada no gabarito explicado na seção Descrevendo os padrões de projeto é ilustrada a seguir.

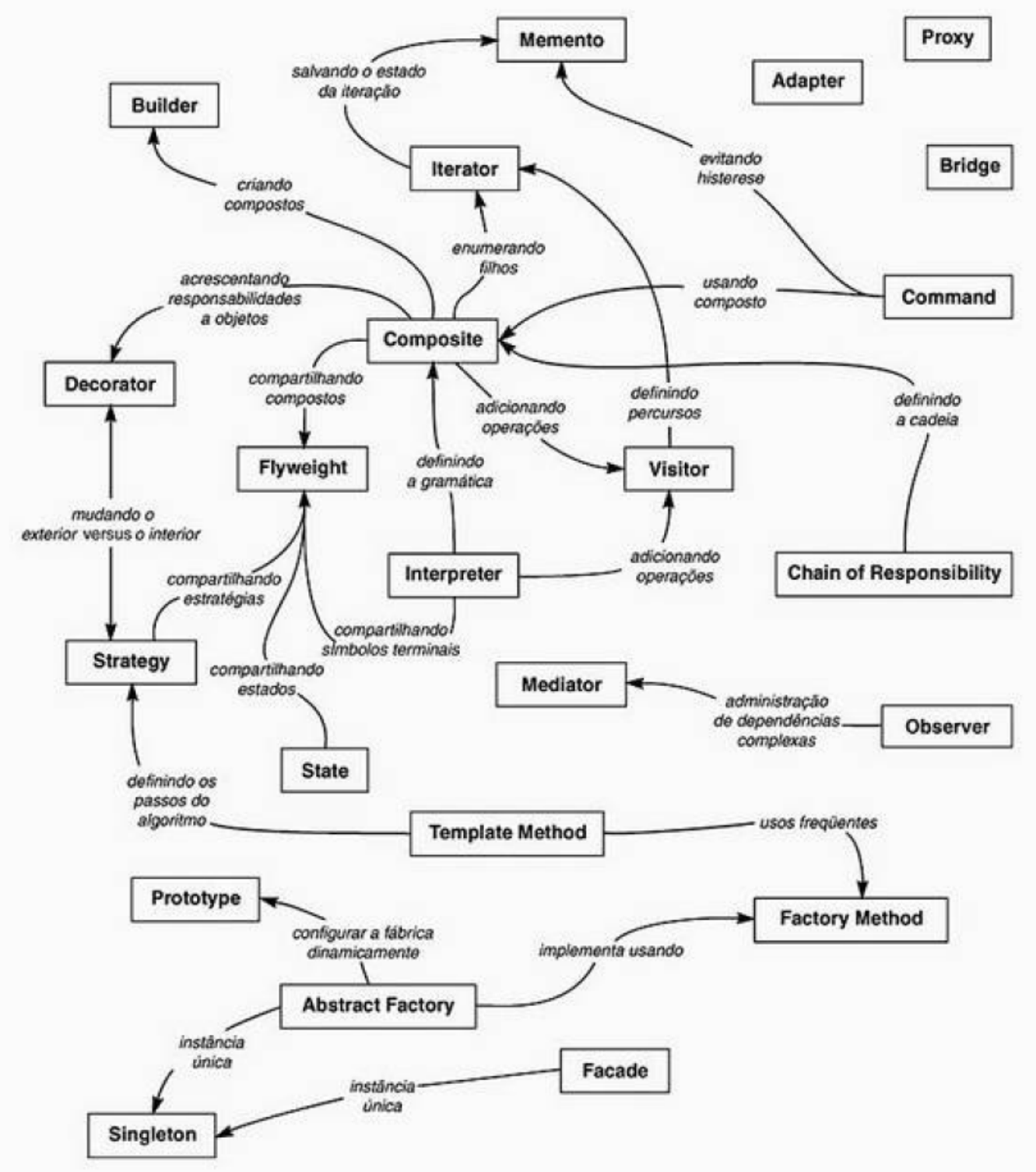

Figura 8. Relacionamento entre padrões de projeto

1.6. Como os padrões solucionam problemas de projeto

1.6.1. Procurando objetos apropriados

Um objeto empacota **dados** e **operações** (métodos, procedimentos ou funções) que operam sobre eles. Um objeto executa tal operação quando recebe u**requisição** (request, solicitação, mensagem) de um **cliente** . As requisições são a *única maneira* de executar operações, que por sua vez são a *única maneira* de modificar os dados. Chama-se isso de **encapsulamento** , onde os dados

não podem ser acessados diretamente e são invisíveis do exterior do objeto.

Decompor um sistema em objetos não é fácil por conta de fatores como 1) encapsulamento, 2) granularidade, 3) dependência, 4) flexibilidade, 5) desempenho, 6) evolução, 7) reutilização, e assim por diante. Eles influenciam a decomposição e, com frequência, de formas conflitantes entre si.

A decomposição pode seguir abordagens diferentes, sabendo que sempre haverá desacordo sobre qual é a melhor:

- Descrever o problema e, então, fazer dos substantivos (as coisas) classes, e dos verbos (ações) operações nessas classes.
- Focar na colaboração e responsabilidades do sistema.
- Na *fase de análise*, modelar os objetos do mundo real e, *na fase de projeto*, traduzir esses objetos para classes de programação.

Muitos objetos num projeto vem da *fase de análise* , mas novos objetos são necessários pois só são percebidos na *fase de projeto* . Tratam-se de objetos de mais baixo nível sem correspondência com o mundo real, tal como vetores. Outros podem estar num nível muito alto, como no padrão *Composite*, que introduz uma abstração para tratamento uniforme de objetos. A modelagem estrita do mundo real gera um sistema que condiz com a realidade atual, mas não necessariamente futura. Por isso, abstrações são a chave para torná-lo flexivel.

Os padrões de projeto ajudam a identificar tais abstrações e os objetos que podem capturá-lasO padrão *Strategy* implementa famílias de algoritmos intercambiáveis, mas esses não ocorrem na natureza. Também, o padrão *State* representa cada estado de uma entidade como um objeto, que não são percebidos durante a análise ou mesmo no estado inicial do projeto. Só ficam evidentes ao tentar tornar o projeto mais flexível e reutilizável.

1.6.2. Determinando a granularidade dos objetos

Podemos usar objetos para representar coisas num nível micro, como peças do hardware, ou num nível macro como uma aplicação inteira. Padrões podem implementar a granularidade necessária. O *Facade* pode representar subsistemas completos como objetos. O *Flyweight* descreve como suportar enormes quantidades de objetos com granularidade mais fina. Para decompor um objeto em objetos menores, o *Abstract Factory* e o *Builder* disponibilizam objetos com a única responsabilidade de criar outros objetos. O *Visitor* e o *Command* disponibilizam objetos com a única responsabilidade de implementar uma solicitação em outro objeto ou grupo de objetos.

1.6.3. Especificando interfaces de objetos

Uma operação (dependendo da linguagem*method* , *function* , *procedure* , etc.) tem um nome, parâmetros e retorno. Isso é o que chamamos de**assinatura** . O conjunto de todas as assinaturas define a **interface** .

Um **tipo** é um nome que identifica uma interface. Vários objetos podem compartilhar o mesmo tipo e um objeto pode acumular vários tipos. Nesse último caso significa que a interface do objeto terá um trecho com assinaturas esperadas por um tipo e outros trechos com assinaturas esperadas por outros tipos. Um tipo pode ser chamado de **subtipo** se tiver a mesma interface de um **supertipo** estabelecendo uma herança, onde o subtipo *herda* a interface do supertipo.

As interfaces são fundamentais em sistemas orientados a objetos porque é a única coisa que você saberá de um objeto. Não se sabe como é a implementação de cada objeto, de modo que dois objetos com mesma interface poderão ter implementações completamente diferentes.

Quando uma solicitação (mensagem) é enviada a um objeto ela tem como destino uma das operações presentes na interface dele. Dois objetos podem aceitar solicitações idênticas. É chamado de **ligação dinâmica** (*dynamic binding*) quando a definição de qual dos objetos será o receptor da solicitação ocorre durante o tempo de execução (*runtime*).

A ligação dinâmica permite que uma implementação em particular não fique presa a certo tipo de solicitação, porque ela poderá ser substituída por outra implementação. A isso chamamos de **polimorfismo**. Assim, você escreve programas que esperam por interfaces de objetos ao invés de objetos especificamente. Isso desacopla e simplifica a relação entre objetos e permite que eles variem seus inter-relacionamentos durante a execução do programa.

Os padrões de projetos ajudam a saber o que colocar e não colocar em uma interface. Também especificam o relacionamento entre interfaces.

1.6.4. Especificando implementações de objetos

Uma implementação de um objeto é definida por sua *classe*. Usando a modelagem *OMT* (*Object Modeling Technique*), uma classe é dada por um retângulo tendo o nome em negrito, as operações e os dados, cada um separado por linhas. Os tipos de retorno e das variáveis serão opcionais porque não é definida neste livro nenhuma linguagem de implementação.

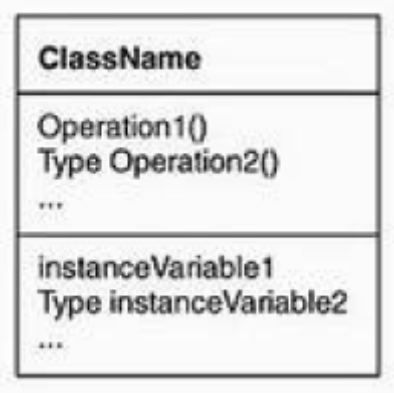

Figura 9. Uma classe de objeto

Um objeto é uma *instância* de uma classe. Esse processo de instanciação aloca memória para os dados internos do objeto, *variáveis da instância*. Também associa as operações a estes dados.

Uma flecha tracejada indica que uma classe instancia objetos de outra classe. Ela aponta para a classe dos objetos instanciados.

Figura 10. Uma classe instancia outra classe

Em uma *herança de classe*, temos na parte de cima a *classe-mãe* e embaixo a *subclasse* que conterá todas as operações e dados das classes ancestrais (mãe, avó, etc.) além dos que ela mesma definir. A

representação se dá por uma linha vertical com um triângulo.

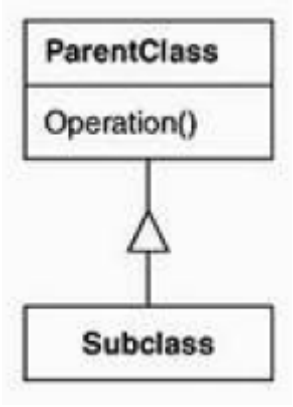

Figura 11. Herança de classes

Uma *classe abstrata* é uma classe cuja finalidade principal é definir uma interface comum para suas subclasses. Ela pode delegar parte de, ou toda, sua implementação para as subclasses, de modo que ela mesma não poderá ser instanciada. As operações que uma classe abstrata declara mas não implementa são chamadas de *operações abstratas* . Classes que não são abstratas são chamadas de *classes concretas* .

Um comportamento implementado em uma classe abstrata pod *redefinido* , isto é, reimplementado, na subclasse.

Classes e operações abstratas tem seus nomes em itálico. Um pseudocódigo pode ser associado a uma operação através de uma caixa com canto dobrado conectada por uma linha pontilhada.

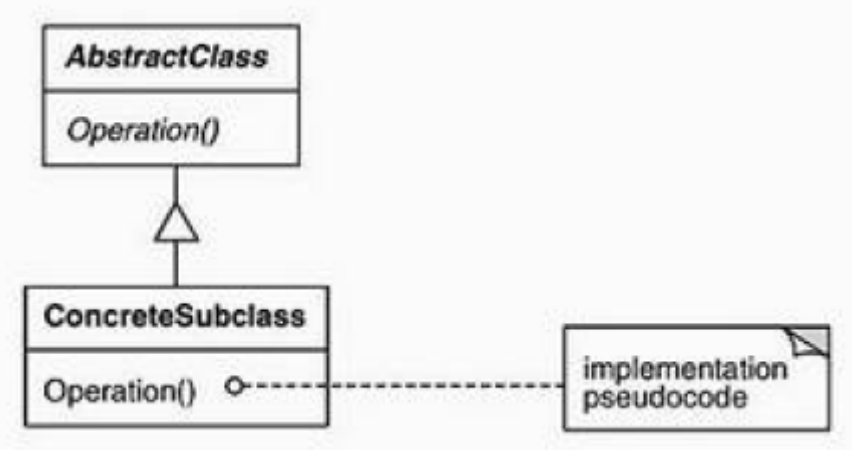

Figura 12. Classe e operação abstratas e pseudocódigo para implementação

Uma *classe mixin* tem a intenção de oferecer uma interface ou funcionalidade opcional a outras classes. São semelhantes a classes abstratas por não poderem ser instanciadas. É exigido que a linguagem de programação suporte herança múltipla.

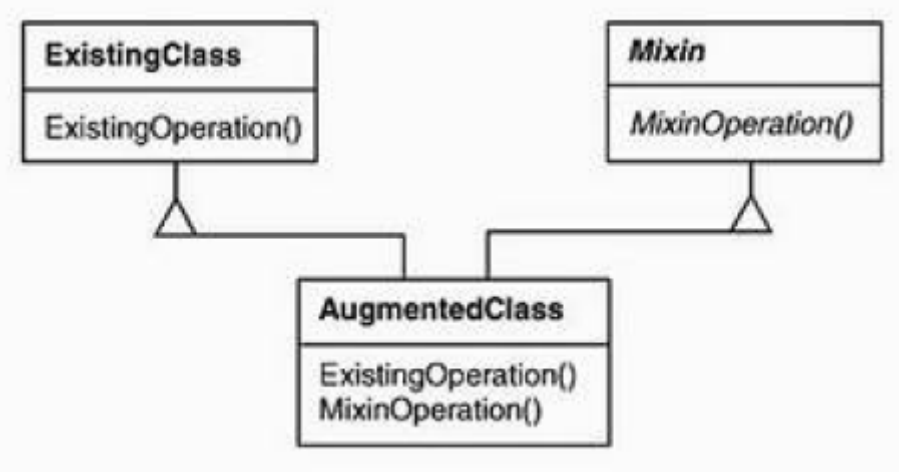

Figura 13. Classe mixin

1.6.4.1. Herança de classe *versus* herança de interface

Existe uma diferença entre *classe* e *tipo*. A classe de um objeto define como ele é implementado ao passo que o tipo se refere apenas a sua interface. Um objeto tem uma única classe que possui a implementação de suas operações e o estado interno dos dados. Mas um objeto pode ter vários tipos dependendo da interface que um cliente venha a precisar. Existem um forte relacionamento entre classe e tipo, mesmo porque a classe é um tipo, mas não é o único tipo do objeto.

Outra diferença é a *herança de classe* e a *herança de interface* (ou subtipificação). No primeiro caso temos um reaproveitamento de código porque a implementação é feita na classe ancestral mas é acessível na subclasse. No segundo caso não há implementação de código, apenas uma especialização onde a subclasse define assinaturas adicionais.

1.6.4.2. Programando para uma interface, não para uma implementação

Este subtítulo anuncia um*princípio* para projetos reutilizáveis orientados a objetos. Significa preferir declarar variáveis como instâncias da classe abstrata ao invés da classe concreta. Dessa forma os clientes permanecem sem conhecimento do objeto quanto ao seu *tipo específico* e *classe que o implementa*.

Quando utilizada apropriadamente a herança permite que qualquer subclasse possa responder a solicitações na interface da classe abstrata, tornando-se subtipo desta. Por exemplo, os *padrões de projeto de criação*, apresentados em Organizando o catálogo, instanciam classes concretas, mas ao invés de entregar essas implementações específicas entregam interfaces generalistas. Com isso o sistema é escrito em termos de interfaces, não de implementações.

1.6.5. Colocando os mecanismos de reutilização para funcionar

Compreender conceitos como objetos, interfaces, classes e herança é fácil quando comparado a aplicá-los na prática, construindo softwares flexíveis e reutilizáveis. Os padrões de projeto ajudam nisso.

1.6.5.1. Herança *versus* Composição

Essas são as duas técnicas mais comuns para a reutilização de funcionalidades em sistemas.

- **Herança de classe**, ou *reutilização de caixa branca*
  - A implementação da subclasse é definida com o código da classe ancestral.
  - A subclasse **pode visualizar** os aspectos internos da classe ancestral.
- **Composição de objetos**, ou *reutilização de caixa preta*
  - Novas funcionalidades são obtidas pela montagem de objetos que entregam uma interface bem definida.
  - Os detalhes internos desses objetos **não são visíveis**.

Seguem as vantagens e desvantagens da herança e composição.

- **Herança de classe**
  - Começando com as vantagens
    - Já que é suportada diretamente pela linguagem de programação, é mais simples de usar.
    - É mais fácil modificar uma implementação que será reutilizada.
  - Desvantagens
    - Quando uma subclasse redefine algumas operações ela pode afetar o funcionamento das demais operações herdadas da classe mãe, porque esta última chamará as operações redefinidas.
    - Como é definida em tempo de compilação, não é possível mudar as operações herdadas das classes ancestrais, mas, antes, elas é quem definem pelo menos parte da representação física das suas subclasses.
    - Frequentemente é dito qu **"a herança viola o encapsulament** porque a implementação da classe ancestral é exposta para a subclasse. Isso faz com que a classe-filha fique tão amarrada a classe-mãe que qualquer mudança na implementação desta forçará uma mudança naquela.
    - Reutilizar uma subclasse pode causar problemas se algum aspecto na classe-mãe não for apropriado a novos domínios de problema. Isso limita a flexibilidade e reusabilidade. Uma cura para isso é herdar somente de classes abstratas, uma vez que elas normalmente fornecem pouca ou nenhuma implementação.
- **Composição de objetos**
  - Começando com as desvantagens
    - Sua utilização é mais trabalhosa, pois é definida dinamicamente em tempo de execução através de algum objeto que entregue as referências dos objetos que farão a composição.
    - Requer que os objetos utilizadores respeitem as interfaces dos objetos utilizadosTais interfaces devem ser cuidadosamente projetadas para que não impeçam você de usar um objeto com muitos outros.
  - Vantagens
    - 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, contato que

tenham o mesmo tipo.

- Como a implementação de um objeto será escrita em termos de interfaces, existirão menos dependências de implementação.

Prefira a composição de objetos à herança de classe Isso favorecerá que cada classe fique encapsulada e focada em uma única tarefa. Suas classes e hierarquias de classes se manterão pequenas com menor probabilidade de crescerem até se tornarem monstros intratáveis.

A experiência mostra que os projetos são mais reutilizáveis e mais simples ao preferir a composição de objetos. Um projeto baseado na composição de objetos fará o comportamento depender do inter-relacionamento entre os objetos, ao invés de ser definido em uma classe. Os padrões de projeto seguem essa abordagem repetidas vezes.

1.6.5.2. Delegação

Delegação é uma maneira de usar composição, onde dois objetos estão envolvidos. Um é o receptor de solicitações que encaminha essas para o seu objeto delegado. Algo semelhante ocorre na herança, quando as solicitações para uma classe filha são encaminhadas para a classe-mãe.

Ao passo que na herança o acesso ao objeto receptor se dá por `this` (algumas linguagens usam `self` ou algo de significado semelhante), na delegação é necessário passar o objeto receptor como parâmetro ao objeto delegado.

No exemplo abaixo é possível ver que `Window` poderia ser uma subclasse de `Rectangle` e disponibilizar a operação `Area()`, mas com delegação ele deve criar uma instância de `Rectangle` e passar a si mesmo como parâmetro. Dessa forma, quando `Area()` for requisitado em `Window` a chamada será delegada, ou encaminhada, para `Rectangle` que poderá realizar operações já que pode ter acesso a `Window`.

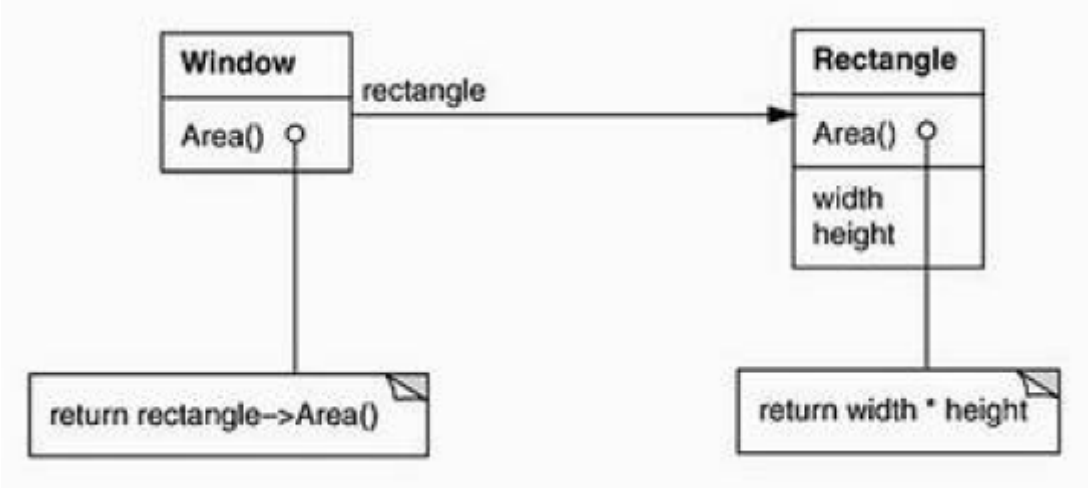

Figura 14. Classe `Window` delega `Area()` para a classe `Rectangle`

A principal **vantagem** da delegação é facilitar definir ou trocar um comportamento em tempo de execução. Por exemplo, se `Window` precisa ser um circulo bastaria trocar `Rectangle` por `Cicle`, desde que tivessem o mesmo tipo.

A **desvantagem** da delegação ocorre por tornar o software mais difícil de compreender do que se fosse estático, isto é, definido em tempo de compilação pela herança. Também pode haver

ineficiências no tempo de execução das operações, mas as ineficiências humanas são mais indesejáveis a longo prazo.

Você sempre pode substituir a herança pela delegação para reutilizar código, mas não necessariamente deve fazer isso. Requer bom critério. Use delegação se isso simplifica mais do que complica.

Diversos padrões de projeto utilizam delegação. Alguns mais, como *State*, *Strategy* e *Visitor*. Outros menos, como *Mediator*, *Chain of Responsability* e *Bridge*.

1.6.5.3. Herança *versus* Tipos Parametrizados

Tipos parametrizados (também conhecido como *generics* ou *templates*) é uma técnica de reutilização de código que define um tipo sem precisar especificar todos os tipos que ele usa. Como exemplo um `List` define no momento da sua declaração qual tipo de objeto será armazenado, seja `string`, `integer`, etc.

Além da *herança* e *composição de objetos*, os *tipos parametrizados* são uma terceira maneira de fazer *composição de comportamentos* em sistemas orientados a objetos. Considere uma rotina de classificação `sort`. Usando...

1. **Herança**: a sub-classe implementa a lógica de ordenação.
2. **Composição de objetos**: o objeto receptor implementa a lógica de ordenação e se passa como parâmetro para o objeto delegado que faz a ordenação de fato.
3. **Tipos parametrizados**: define como parâmetro do tipo uma função que possui a logica de ordenação.

Nos casos 1 e 3 o comportamento está sendo definido em tempo de compilação, ao passo que no caso 2 será em tempo de execução.

Nenhum dos padrões deste livro trata de tipos parametrizados, embora ele possa ocorrer em uns poucos exemplos de código em C++.

1.6.6. Relacionando estruturas de tempo de execução e de tempo de compilação

Considere a diferença entre **agregação** e **associação** (em inglês, acquaintance, significando "conhecimento ou relacionamento social", mas sendo aqui a ideia de conhecimento de um objeto por outro).

A **agregação** consiste na relação *um objeto tem*, ou *um objeto é parte de* outro objeto. Exige que ambos nessa relação tenha o mesmo tempo de vida.

A **associação** implica na relação *um objeto usa*, ou seja, tem conhecimento de outro objeto. Embora um objeto que use outro possa solicitar operações, ele não é responsável pelo outro. Trata-se de um relacionamento mais fraco do que a agregação, com *menor acomplamento* entre os objetos.

Nos diagramas desse livro, *associação* é representada por uma flecha de linha cheia como na última figura, sobre composição de objetos. Logo abaixo você vê a representação de uma

agregação, com uma flecha tendo um losango em sua base.

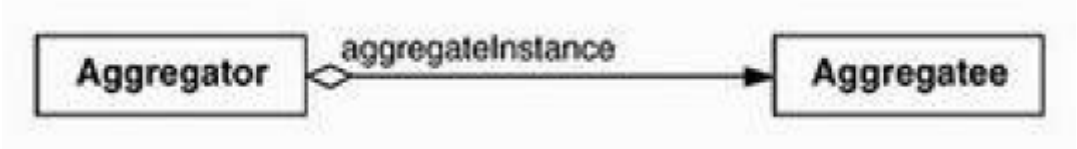

Figura 15. Representação de agregação pelo losango na base da flecha

Associação e *agregação* são determinadas mais pela intenção do projetista do que por mecanismos da linguagem, visto que muitas delas não fazem distinção entre esses tipos de definição. Relacionamentos de agregação são em menor número e mais permanentes que associações. Associações são feitas e refeitas com muito mais frequência, algumas vezes existindo somente para a duração de uma operação.

O código não revelará tudo acerca de como o sistema funciona. Embora *agregações* sejam mais fáceis de serem percebidas por estarem no tempo de compilação, a *associações* , impostas pelo projetista ao invés de evidenciadas pela linguagem, vão requerer uma visão mais profunda.

Um bom ou mau projeto no quesito de relacionamento entre objetos é o que faz de uma estrutura algo bom ou ruim no tempo de execução. Muitos padrões (em especial os que tem escopos de objeto), capturam explicitamente a distinção entre estruturas de tempo de compilação e execução, por exemplo, *Composite*, *Decorator*, *Observer*, *Chain of Responsability*. As estruturas de tempo de execução estabelecidas por eles não serão claras a menos que você conheça tais padrões.

1.6.7. Projetando para mudanças

A chave para reutilização está na capacidade de antecipar as futuras mudanças que surgirão, para que o então sistema possa ir evoluindo atendendo os novos requisitos. Projetos que não levam isso em conta acabam tendo que passar por grandes reformulações, que acarretam em alto custo e exige uma retestagem de todo o sistema.

Padrões de projeto ajudam nesse ponto porque permitem variar algum aspecto da estrutura sem variar as demais. Seguem algumas causas comuns que levam a uma reformulação e quais padrões tratam essas causas

1. **Criação de objeto pelo nome direto da classe**:

   Isso resulta em acomplamento de uma objeto com uma implementação em particular, quando o ideal é depender de uma interface. Crie objetos indiretamente.

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

2. **Dependência de operações específicas**:

   Especificar uma operação em particular faz com que haja uma dependência com uma determinada maneira de atender uma solicitação. Solicitações codificadas inflexivelmente (*hard-coded*) devem ser evitadas para torna mais fácil mudar a maneira como uma solicitação é atendida, seja no tempo de compilação ou execução.

   Padrões de projeto: *Chain of Responsability* e *Command*.

3. **Dependência da plataforma de hardware ou software**:

   APIs (*Interfaces de Programação de Aplicações*, em inglês, *Application Programming Interface*) externas podem sofrer modificações que vão comprometer as partes internas do sistema que dependem dela. É importante projetar de tal forma a limitar essa dependência.

   Padrões de projeto: *Abstract Factory* e *Bridge*.

1. **Dependência de representações ou implementações de objetos**:

   Quando um cliente precisa saber como um objeto é representado, armazenado, localizado ou implementado ele pode necessitar de alteração quando este objeto muda. Ocultar essas informações dos clientes evita esse tipo de propagação de mudanças em cadeia.

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

2. **Dependências de algoritmos**:

   Algoritmos são frequentemente estendidos, otimizados e substituídos durante o desenvolvimento. Por isso eles devem ser isolados para evitar que um objeto tenha dependência dele.

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

3. **Acomplamento forte**:

   Classes que são fortemente acopladas são difíceis de serem usadas isoladamente porque dependem uma das outras. Ou usa todas ou não usa nenhuma. Isso evidencia um sistema monolítico, onde a mudança de uma classe leva a mudança ou revisão das classes ao seu redor. Por outro lado, um acomplamento fraco torna mais fácil que uma classe possa ser usada por si mesma.

   Padrões de projeto: *Abstract Factory*, *Bridge*, *Chain of Responsability*, *Command*, *Facade*, *Mediator* e *Observer*.

4. **Estendendo funcionalidade através de sub-classes**:

   Escrever uma sub-classe exige conhecimento profundo da classe-mãe. Caso implementações da classe-mãe sejam redefinidas pela sub-classe isso pode causar comportamentos inesperados. Enquanto que a composição de objetos pela delegação fornece uma alternativa mais flexível para combinar comportamentos. Mas existe desvantagens nos dois casos, quando a *herança* pode levar a uma explosão de sub-classes e a *composição de objetos* pode levar a projetos menos compreensíveis.

   Por outro lado, muitos padrões produzem arquiteturas, que são designs conhecidos, nas quais você pode introduzir uma funcionalidade usando herança ou composição de objetos.

   Padrões de projeto: *Bridge*, *Chain of Responsability*, *Composite*, *Decorator*, *Observer* e *Strategy*.

1. **Incapacidade de alterar classes de modo conveniente**:

   Bibliotecas comerciais, por exemplo, são fechadas para modificação. Ou uma modificação no sistema pode exigir muitas outras. Há padrões que oferecem formas de realizar mudanças mesmo em tais circunstâncias.

   Padrões de projeto: *Adapter*, *Decorator* e *Visitor*.

Ao usar padrões de projeto para contornar os casos acima você deve avaliar quão crucial a flexibilidade é para seu sistema. A seguir veremos o papel dos padrões em três grandes classes de software: 1) programas de aplicação, 2) toolkits e 3) frameworks.

1.6.7.1. Programas de Aplicação

Ao construir uma aplicação, como um editor de texto ou qualquer coisa do tipo, sua prioridade será garantir uma *reusabilidade interna* para facilitar a manutenção, extensão e não fazer você projetar ou implementar mais do que o necessário.

Padrões de projeto reduzem a dependência interna mantendo um acomplamento mais fraco, o que torna mais fácil reutilizar classes e operações em contextos diferentes. Também podem ajudar a limitar a dependência da plataforma em que a aplicação é executada e dar a ela uma estrutura de camadas.

1.6.7.2. Toolkits (Bibliotecas de Classes)

Tratam-se de um conjunto de classes relacionadas que visam *reusabilidade externa* . Toolkits não impõe uma modelagem de projeto específica à aplicação que o utiliza, mas se propõe apenas a fornecer um conjunto de funcionalidades para uso geral.

O autor do toolkit não está numa posição em que possa saber quais aplicação o utilizarão. Por isso deve evitar suposições e dependências para torna-lo alinhado ao seu propósito, isto é, a reutilização de código.

1.6.7.3. Frameworks (Arcabouços de Classes)

Um framework é um conjunto de classes que cooperam entre si para construir um projeto reutilizável em uma determinada categoria de software. Trata-se de algo genérico, ao passo que a aplicação será a concretização das abstrações do framework. De forma geral, a aplicação implementa subclasses que herdam das classes abstratas do framework.

O framework define a arquitetura da aplicação. Dessa forma, o implementador se concentra mais nos aspectos específicos à aplicação do que a como ela será estruturada. Ou seja, o framework já terá decidido o que é comum ao domínio da aplicação, porque que ela vai pertencer a uma categoria no qual o framework se baseia.

Frameworks enfatizam a *reutilização de projeto* , ao invés de *reutilização de código*, como seria o caso dos toolkits visto antes. Ainda que os frameworks possam ter classes concretas, haverá uma inversão de controle porque não será a aplicação que vai instanciar tais classes ou chamar suas operações, mas sim o framework. Mais uma vez, isso será inverso ao uso de toolkits, onde a aplicação chama o que o toolkit tem a oferecer. A aplicação baseada em um framework escreve

operações com nomes e convenções já definidas pelo framework.

Com essas decisões já tomadas a construção da aplicação acaba sendo mais rápida. Mesmo aplicações diferentes utilizando o mesmo framework terão estruturas similares. Isso vai passar ao usuário uma experiência de uso comum e mais consistente. Fica claro, então, que se perde liberdade criativa porque qualquer decisão do implementador deverá estar dentro dos limites do arcabouço das classes do framework.

Se aplicações são difíceis e toolkits ainda mais, os frameworks ultrapassam em dificuldade a todos os dois. Isso porque deverá prever uma arquitetura que funcionará para todas as aplicações do seu domínio. Uma mudança no framework pode obrigar mudanças nas aplicações já construídas, o que não é desejável. Por isso, ao serem construídos é imperativo que ele seja tão flexível e extensível quanto possível. Desta forma, o acomplamento fraco é ainda mais importante aqui, porque, do contrário, mesmo pequenas mudanças no framework afetaria as aplicações construídas em cima dele.

Por tudo considerado acima, frameworks que usam padrões de projetos tem maior probabilidade de atingir altos níveis de reusabilidade de projeto e código. Outro benefício é tido quando o framework é documentado com citação aos padrões de projeto utilizados, porque pessoas que os conhecem terão uma compreensão mais rápida do funcionamento. Ainda mais do que para aplicações e toolkits, uma boa documentação é um ponto crítico dos frameworks. Embora os padrões não possam achatar a curva de aprendizado, podem torná-la mais suave ao explicitar os elementos-chaves do framework.

Padrões e framework tem similaridades, mas há pelo menos três principais diferenças:

1. **Padrões de projeto são mais abstratos que frameworks**: Frameworks são materializados em código-fonte, mas padrões são apenas conceitos que usam exemplos de código-fonte para serem explicados. Frameworks são feitos em uma linguagem específicas, padrões não. Padrões não tomam decisões, mas apresentam os custos e benefícios (trade-offs) entre a escolha de um a outro.
2. **Padrões de projetos são elementos de arquitetura menores que framew**: Um framework típico terá muitos padrões de projetos, mas a recíproca nunca é verdadeira.
3. **Padrões de projeto são menos especializados que frameworks**: Os frameworks sempre tem um particular domínio de aplicação, fazendo com que todas as aplicações tenham propósitos semelhantes. Já os padrões podem ser usados em quase qualquer domínio de aplicação e não estabelecem uma estrutura a ser seguida.

Os frameworks estão se tornando cada vez mais comuns e importantes. Isso porque são a maneira pela qual aplicações conseguem maior reutilização. Aplicações maiores podem inclusive ter camadas de frameworks que cooperam entre si. Em todos os casos o projeto da aplicação será definida ou pelo menos fortemente influenciadas pelos frameworks.

1.7. Como selecionar um padrão de projeto

Temos nesse livro mais de 20 padrões de projeto. Abordagens que ajudam na escola:

Como padrões solucionam problemas de projeto

Para isso considere a seção Como os padrões solucionam problemas de projeto.

Quais as intenções dos padrões

Na seção O catálogo de padrões de projeto vemos a intenção de cada padrão. A classificação apresentada na tabela Organização dos padrões de projeto também vai ajudar nisso.

Como os padrões se interrelacionam

Veja a imagem Relacionamento entre padrões de projeto.

Estude padrões de finalidades semelhantes

O [Catálogo de padrões de projeto] tem três capítulos focados em padrões 1) de criação, 2) estruturais e 3) comportamentais. Cada capítulo inicia com uma introdução e finaliza com as semelhanças e diferenças.

Examinar a causa da reformulação do projeto

Veja as causas de reformulação apresentadas em Programas de Aplicação, Toolkits (Bibliotecas de Classes) e Frameworks (Arcabouços de Classes). Procure saber se o seu problema envolve alguma delas.

Considere o que devere ser variável no seu projeto

Essa abordagem é o inveso da anterior, porque aqui você quer deixar seu projeto aberto para mudanças sem ter que reformulá-lo. Veja a tabela Aspectos do projeto que o uso de padrões permite variar. O **encapsulamento do conceito que varia** é um tema comum para muitos padrões de projeto.

1.8. Como usar um padrão de projeto

Depois de escolhido o padrão, como usar:

1. Tenha uma visão geral lendo o padrão por inteiro. Dê atenção às seções *Aplicabilidade* e *Consequências* para ter certeza que é o padrão correto para o seu problema.
2. Estude as seções *Estrutura*, *Participantes* e *Colaborações* e compreenda as classes e objetos e seus relacionamentos.
3. Olhe a seção *Exemplo de Código* e veja um exemplo concreto de código para ter ideia de como implementá-lo.
4. Escolha os nomes para os participantes do padrão que façam sentido no contexto da sua aplicação. Isso porque os nomes para os participantes no padrão serão tão genéricos quanto ele.
5. Defina as classes. Declare suas interfaces, estabeleça seus relacionamentos de herança, defina as variáveis de instâncias que representam dados ou referenciam objetos. Então, identifique as classes já existentes na sua aplicação que serão afetadas pelo padrão e modifique-as de acordo.
6. Defina os nomes das operações com base no contexto da sua aplicação. Seja consistente na nomeclatura. Por exemplo, ao usar o padrão *Factory Method*, poderá usar `Create` como prefixo das operações.
7. Implemente as operações para suportar as responsabilidades e colaborações presentes no

padrão. As seções *Implementação* e *Exemplo de Código* vão ser de ajuda.

Propósito	Padrão	Aspecto(s) que pode(m) variar
De Criação	Abstract Factory (95)	famílias de objetos-produto
	Builder (104)	como um objeto composto é criado
	Factory Method (112)	subclasse de objeto que é instanciada
	Prototype (121)	classe de objeto que é instanciada
	Singleton (130)	a única instância de uma classe
Estruturais	Adapter (140)	interface para um objeto
	Bridge (151)	implementação de um objeto
	Composite (160)	estrutura e composição de um objeto
	Decorator (170)	responsabilidade de um objeto sem usar subclasses
	Façade (179)	interface para um subsistema
	Flyweight (187)	custos de armazenamento de objetos
	Proxy (198)	como um objeto é acessado; sua localização
Comporta-mentais	Chain of Responsibility (212)	objeto que pode atender a uma solicitação
	Command (222)	quando e como uma solicitação é atendida
	Interpreter (231)	gramática e interpretação de uma linguagem
	Iterator (244)	como os elementos de um agregado são acessados, percorridos
	Mediator (257)	como e quais objetos interagem uns com os outros
	Memento (266)	que informação privada é armazenada fora de um objeto e quando
	Observer (274)	número de objetos que dependem de um outro objeto; como os objetos dependentes se mantêm atualizados
	State (284)	estados de um objeto
	Strategy (292)	um algoritmo
	Template Method (301)	passos de um algoritmo
	Visitor (305)	operações que podem ser aplicadas a (um) objeto(s) sem mudar sua(s) classe(s)

Figura 16. Aspectos do projeto que o uso de padrões permite variar

Tão importante quanto saber *quando usar* é saber *quando não usar* um padrão. Eles não devem ser aplicados indiscriminadamente porque, embora tragam flexibilidade, aumentam a complexidade. Use um padrão apenas se a flexibilidade oferecida for realmente necessária.

2. Um estudo de caso: projetando um editor de documentos

continuar na p. 47