Padrões de Projeto

Julio Zancan

Programa do Curso

1. Introdução
   - O que são Padrões
   - Catálogo de Padrões
   - Seleção de Padrões
2. Padrões de Criação
   - Apresentação
   - Padrões
3. Padrões Estruturais
   - Apresentação
   - Padrões
4. Padrões Comportamentais
   - Apresentação
   - Padrões

Padrões de Projeto

- “Projetar software OO é difícil ...”
- “Projetar software OO **reutilizável** é ainda mais difícil”
  - Encontrar objetos relevantes
  - Fatorar classes
  - Estabelecer hierarquias de herança
  - Limites entre especifidade / generalidade
- É difícil obter projetos flexíveis / reutilizáveis na primeira vez
  - Várias ‘iterações’ de reutilização para **refatoramentos** e reorganização do código

Padrões de Projeto

- Projetistas experientes não partem do zero na resolução de problemas de projeto:
  - **Partem de soluções que funcionaram no passado**
  - **Reusam boas soluções repetidamente**
- Esta prática resulta em **padrões de soluções** aplicados sucessivamente em software orientado a objetos
- Os padrões de soluções resolvem problemas específicos
  - Tornam os projetos mais flexíveis e reusáveis

Padrões de Projeto

- Projetistas familiarizados com os padrões podem aplicá-los sem a necessidade de redescobri-los

- Os padrões de soluções mais importantes foram catalogados e denominados ‘Padrões de Projeto’

Padrões de Projeto

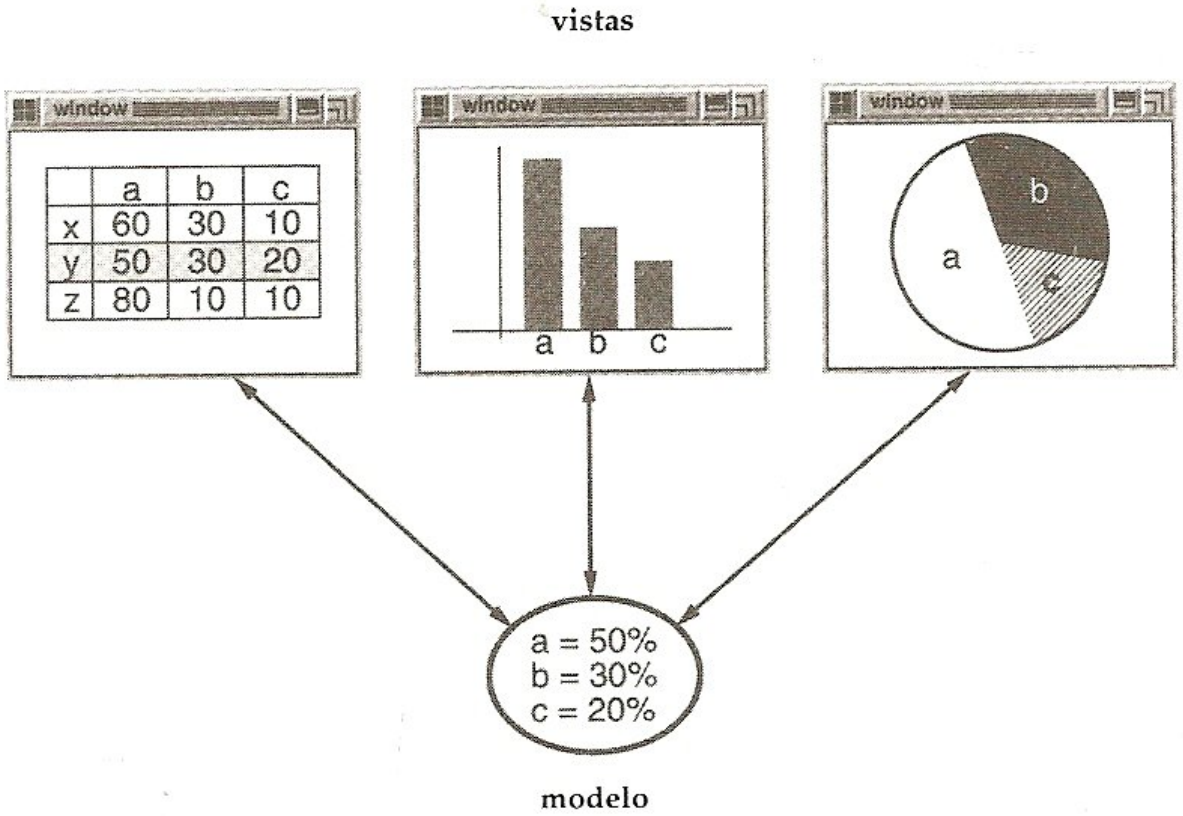
- O que é um padrão?
  - Christopher Alexander : “cada padrão descreve um problema no nosso ambiente e o núcleo de solução, de tal forma que você possa usar esta solução mais de um milhão de vezes”
  - Christopher Alexander é um arquiteto e utiliza os padrões para soluções em construções, porém a mesma definição vale para software OO

Padrões de Projeto

- Padrões são descritos por quatro elementos essenciais ou básicos:
  - Nome do padrão
  - Problema
  - Solução
  - Conseqüências

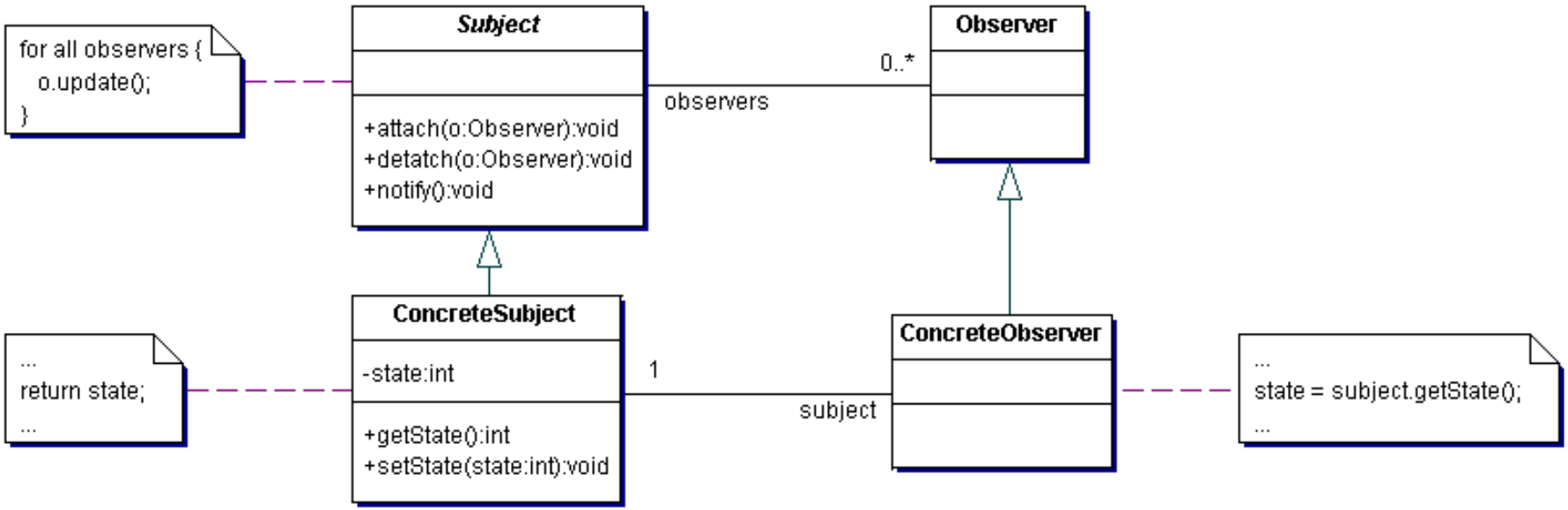
Exemplo: Observer

- Nome: Observer
- Problema: como manter consistência em entre objetos relacionados?



Exemplo

- Solução: Definir uma dependência de um para muitos, de maneira que quando um objeto muda de estado todos os dependentes são notificados



Exemplo: Observer

- Conseqüências: o padrão Observer permite variar ‘subjects’ sem reutilizar seus observadores e vice-versa

...

Catálogo de Padrões

- Captura os padrões de projeto, organizando de acordo com sua finalidade
- São quatro finalidades
  - Criação
  - Estruturação
  - Comportamento
- O catálogo é descrito de acordo com o seguinte gabarito:
  - Nome e classificação do padrão
  - Intenção e objetivo
  - Também conhecido como

Catálogo de Padrões

- Motivação
- Aplicabilidade
- Estrutura
- Participantes
- Colaborações
- Conseqüências
- Implementação
- Exemplo de Código
- Usos Conhecidos
- Padrões relacionados

Padrões de Criação

- Abstraem o processo de instanciação
- Ajudam a promover a independência de um sistema da forma como os objetos são criados, compostos e representados
  - Encapsulam conhecimento sobre quais classes concretas são usadas pelo sistema
  - Ocultam o modo como as instâncias são criadas e associadas
    - Ex. Template Method, Abstract Factory

Padrões de Criação

Padrão	Aspecto(s) variável(is)
Abstract Factory	Famílias de objetos-produto
Builder	Como um objeto composto é criado
Factory Method	Subclasse do objeto que é instanciada
Prototype	Classe de objeto que é instanciada
Sigleton	A única instância de uma classe

Padrões Estruturais

- Se preocupam com a forma como classes e objetos são compostos para formar estruturas maiores.
  - Padrões estruturais de classe: usam herança para compor interfaces ou implementações (foco na composição de interfaces ou implementações)
    - Ex. Adapter, Facade
  - Padrões estruturais de objetos descrevem maneiras de compor objetos para obter novas funcionalidades – permitem mudar a composição em tempo de execução.
    - Ex. Composite

Padrões Comportamentais

- Preocupam-se com algoritmos e a atribuição de responsabilidades entre os objetos
  - Não descrevem apenas padrões de objetos ou classes – descrevem também os padrões de comunicação entre eles
- Padrões comportamentais de classe utilizam herança para distribuir comportamento entre classes
  - Ex. Template Method, Interpreter

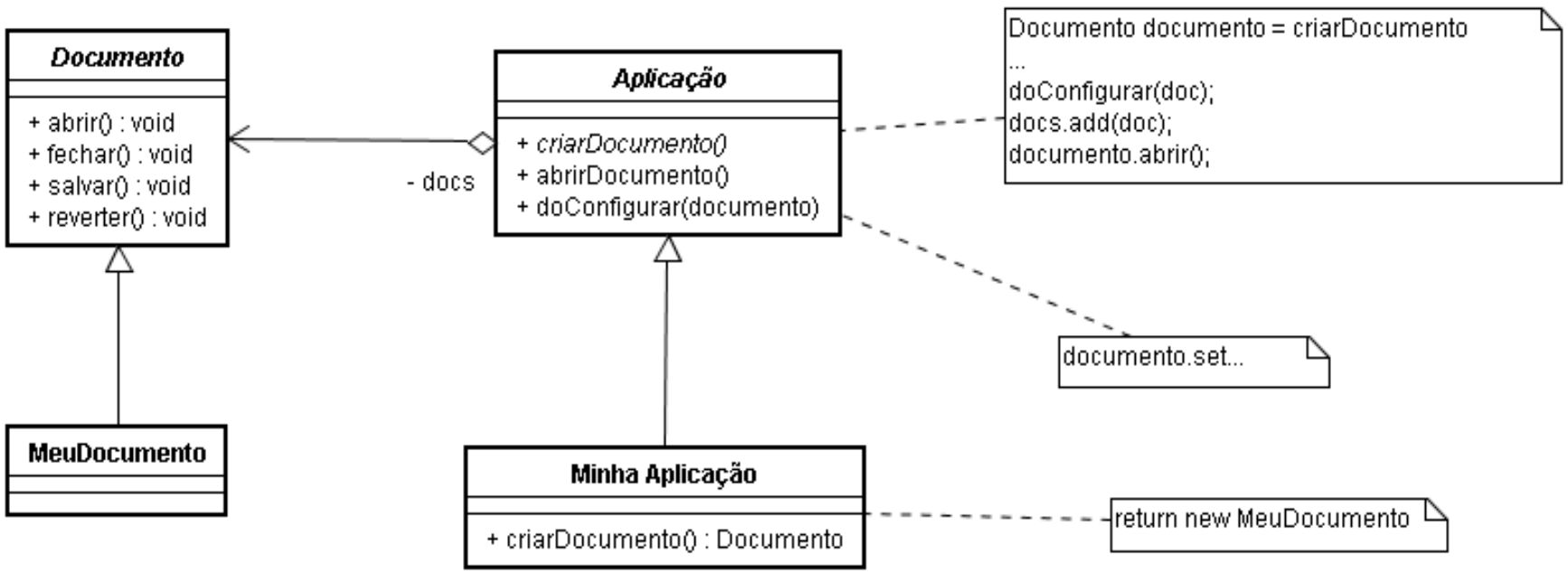
Padrões Comportamentais

- Padrões comportamentais de objetos utilizam a composição de objetos.
  - Descrevem como objetos conhecem uns aos outros: através de referência explicita (mais acoplamento) ou referência indireta (acoplamento fraco)
    - Ex.: Chain of responsibility, Observer, Command, State

Template Method

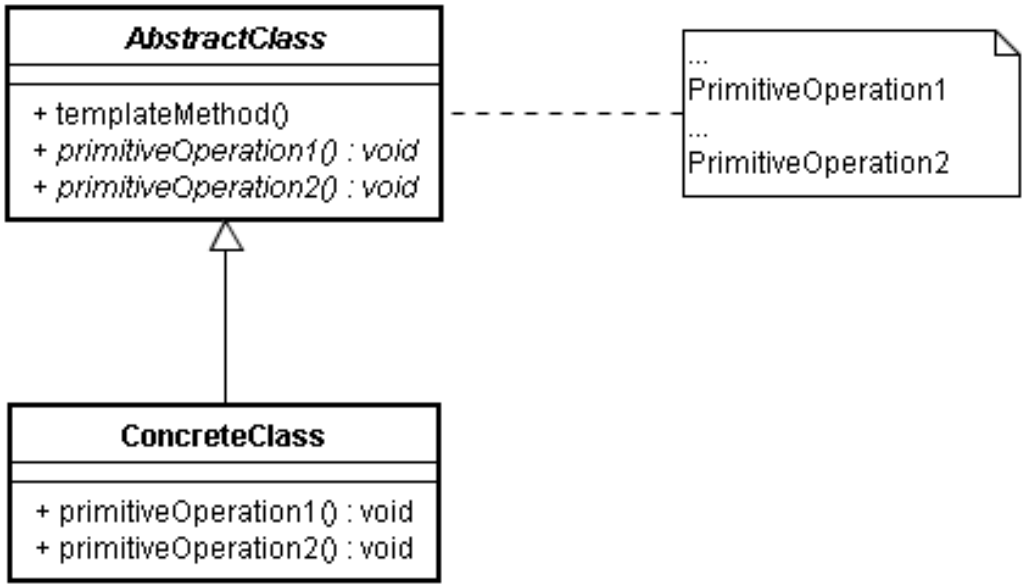
- Intenção: Definir o esqueleto de um algoritmo em uma operação, postergando (deferring) alguns passos para as subclasses. Template Method permite que subclasses redefinam certos passos de um algoritmo sem mudar a estrutura do mesmo

Template Method



- Aplicabilidade
  - Implementar as partes invariantes de um algoritmo e deixar para subclasses o comportamento variável
  - Evitar duplicação de código na refatoração de código em subclasses a fim de generalizar comportamentos
  - Para controlar os pontos de extensões de subclasses. O método template determina os locais **customizáveis** por subclasses

Estrutura abstrata

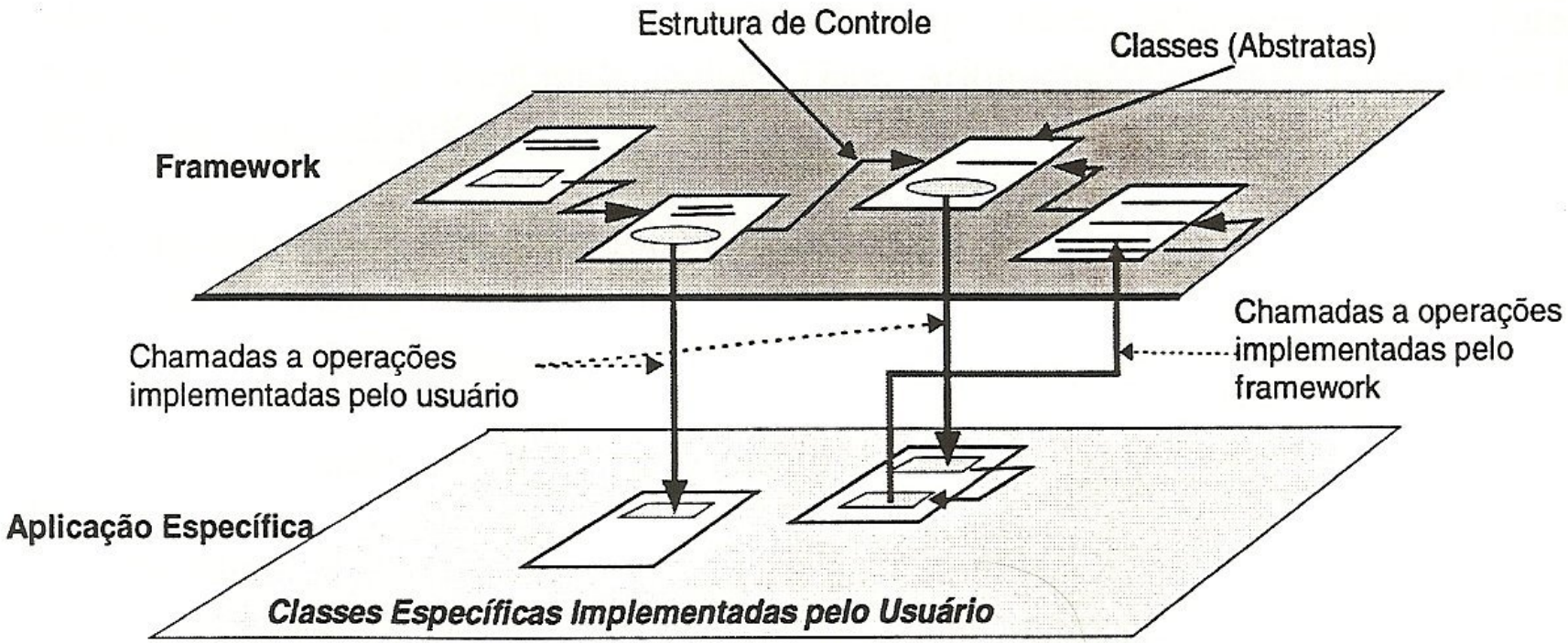


Template Method

- Participantes
  - Abstract Class
    - define operações primitivas abstratas para serem redefinidas nas subclasses
    - Implementa o método template que define o esqueleto de um algoritmo, o qual invoca as operações primitivas
  - Concrete Class
    - Implementa as operações primitivas para executarem os passos específicos do algoritmo da subclasse

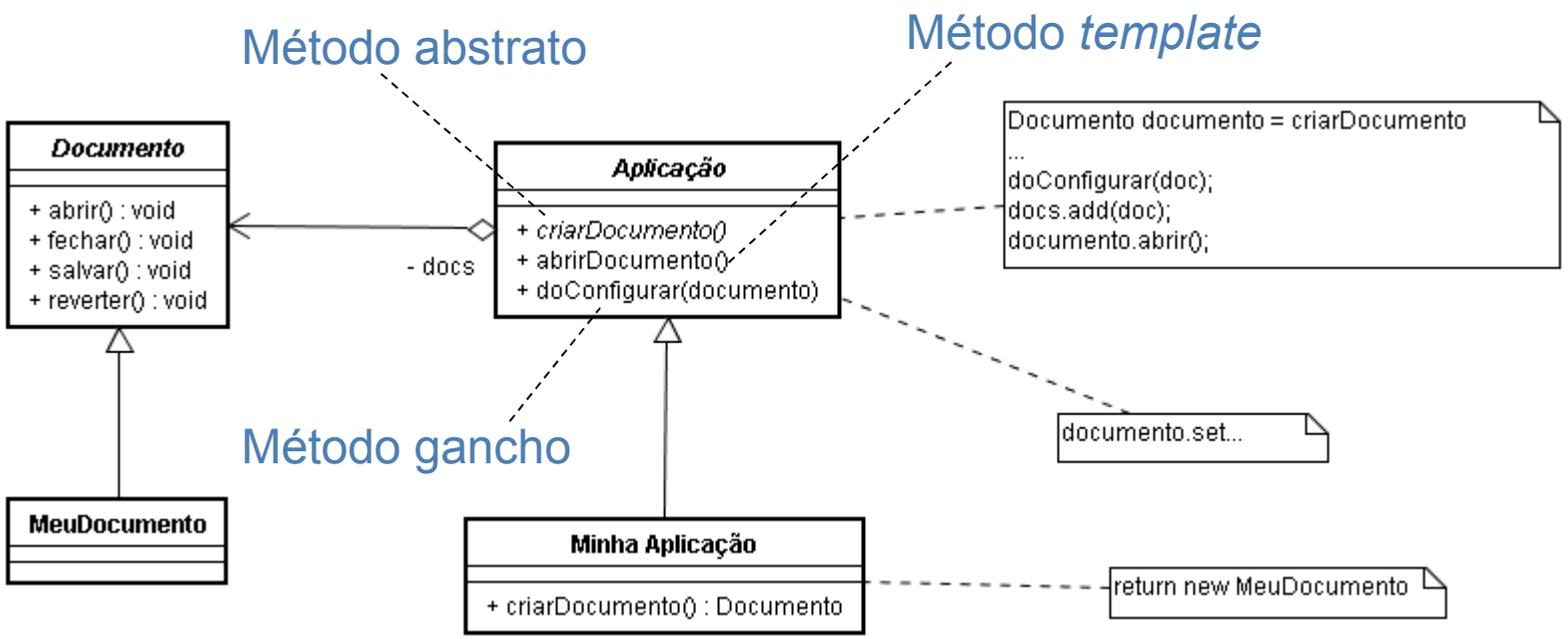
- Conseqüências
  - Os métodos template são um técnica fundamental para a reutilização
  - São a base da **inversão de controle** ou "princípio de Hollywood": "*Não nos chame, nós chamaremos você!*"
  - As classes abstratas e seus métodos template formam o esqueleto básico na construção de *frameworks orientados a objetos*.

Template Method



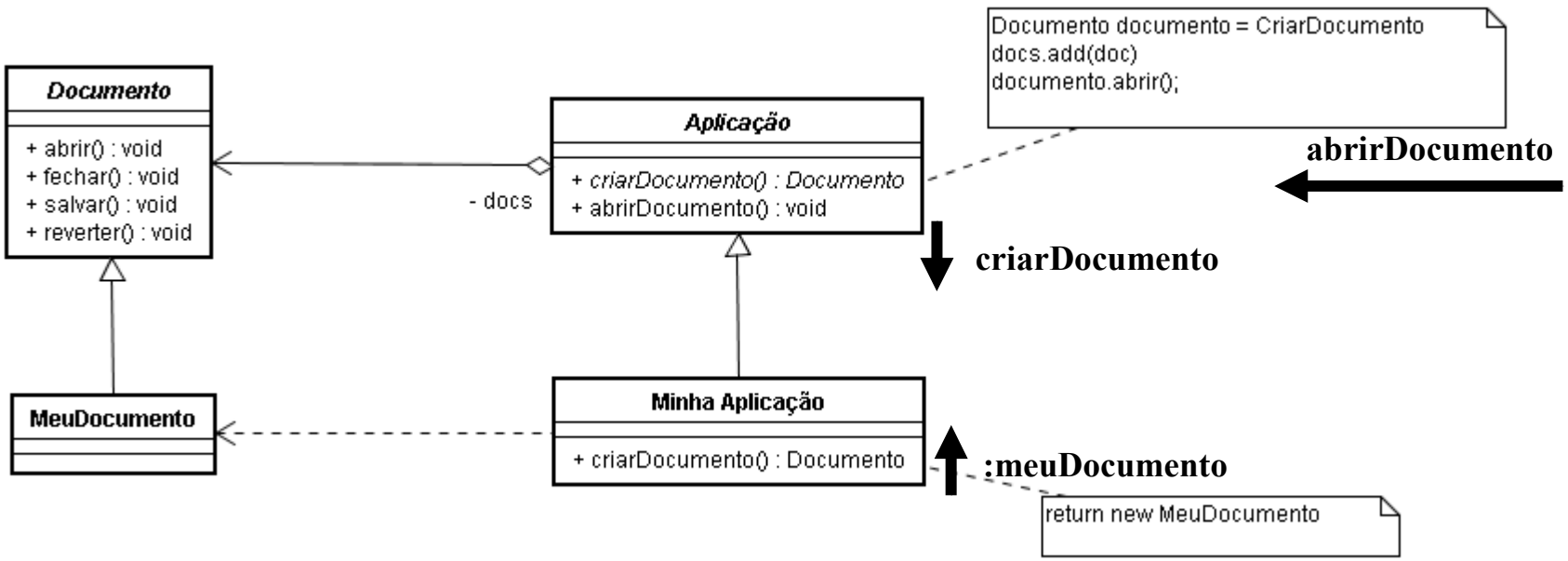
Template Method

- Métodos template especificam operações 'ganchos' (*hooks*), as quais **podem** ser redefinidas e são operações abstratas as quais **devem** ser redefinidas



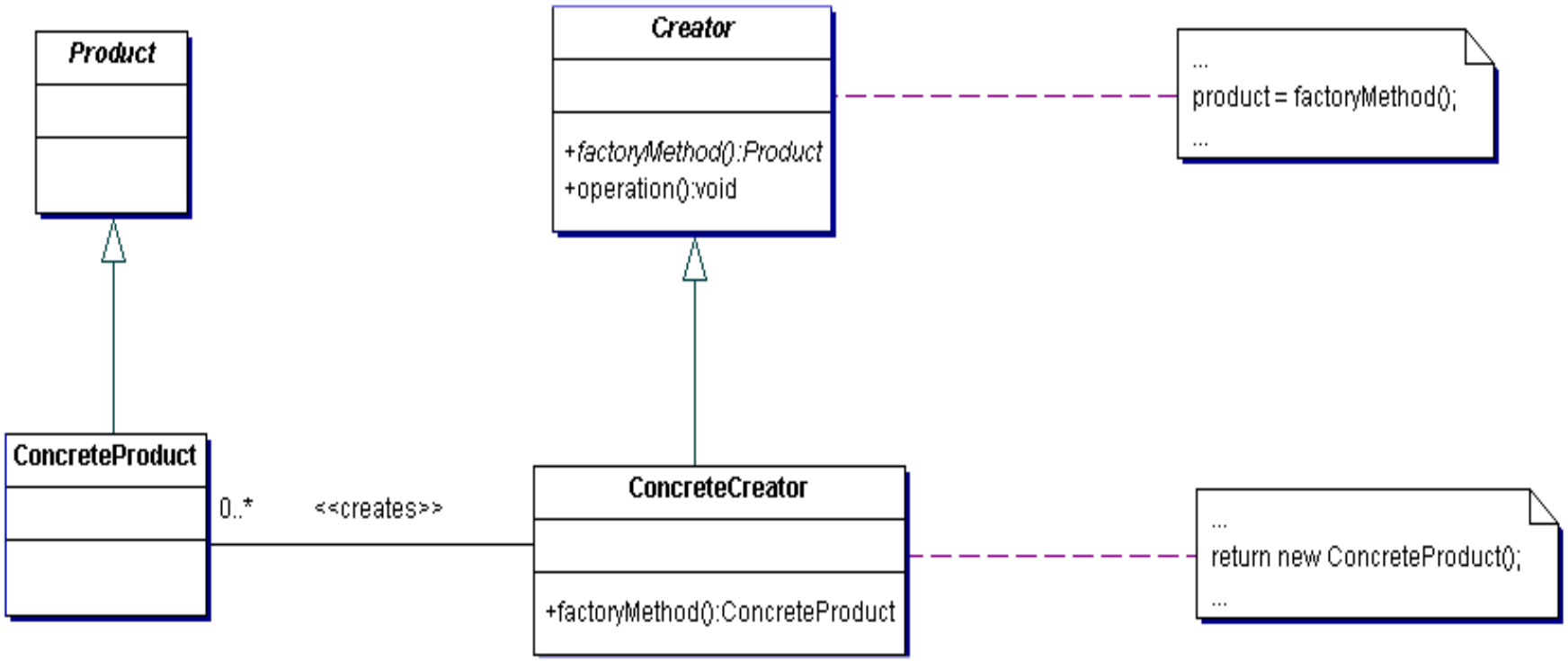
Factory Method

- Intenção: Definir uma interface para criar um objeto, mas deixar as subclasses decidirem que classes instanciar



- Aplicabilidade
  - Uma classe não pode antecipar a classe dos objetos que devem ser criados
    - Delega a uma subclasse a especificação dos objetos que devem ser criados

- Estrutura Abstrata



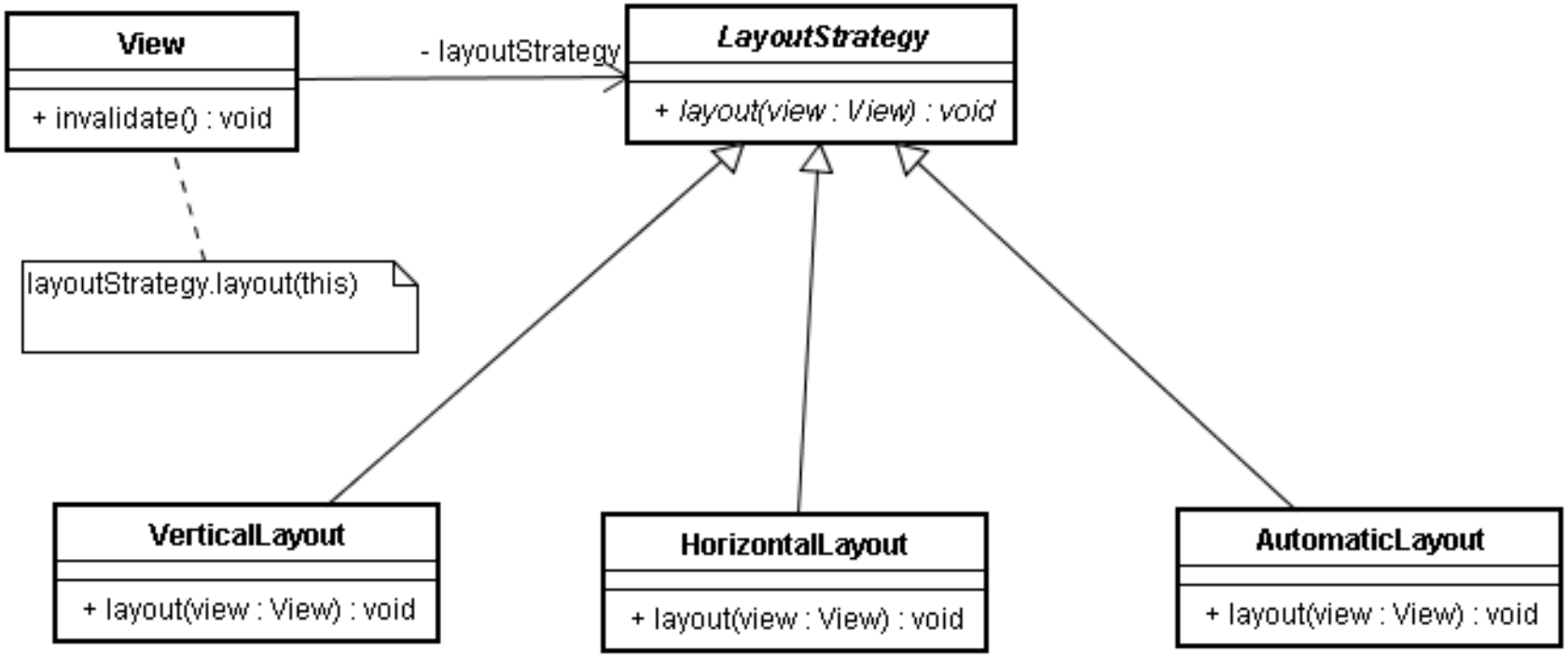
Factory Method

- Participantes
  - Produto
    - Define a interface de objetos criados pelo método fábrica
  - ProdutoConcreto
    - Implementa a interface de Produto
  - Criador
    - Declara o método fábrica o qual retorna um objeto do tipo Produto . Pode definir implementação por default.
  - CriadorConcreto
    - Redefine – 'overrides' o método fábrica para retornar um ProdutoConcreto

Strategy

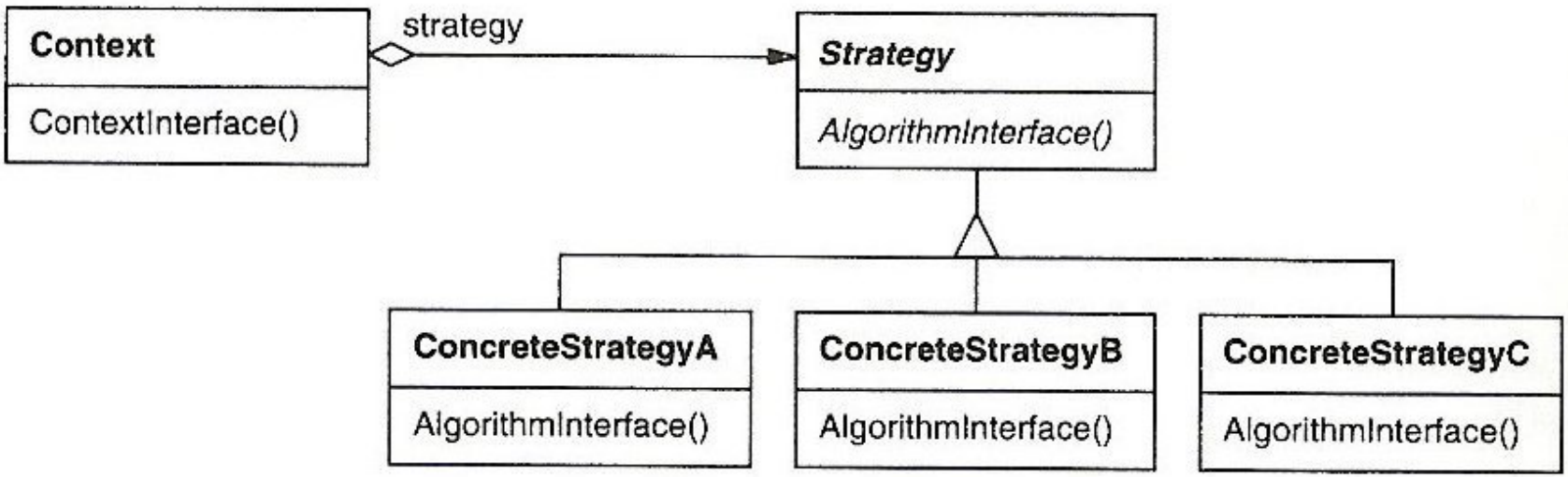
- Intenção: Definir uma família de algoritmos, encapsular cada uma das famílias e torná-las intercambiáveis. Strategy permite que o algoritmo varie independentemente dos clientes que o utilizam.

Strategy


View
+ invalidate() : void
- layoutStrategy
LayoutStrategy
+ layout(view : View) : void
layoutStrategy.layout(this)
VerticalLayout
+ layout(view : View) : void
HorizontalLayout
+ layout(view : View) : void
AutomaticLayout
+ layout(view : View) : void

- Aplicabilidade
  - Muitas classes relacionadas diferem somente no seu comportamento. As estratégias fornecem uma maneira de configurar um comportamento dentre muitos.
  - São necessárias variantes de um algoritmo
  - Um algoritmo usa dados de que os clientes não deveriam ter conhecimento
  - Uma classe define muitos comportamentos e estes aparecem como múltiplos comandos condicionais.

- Estrutura Abstrata

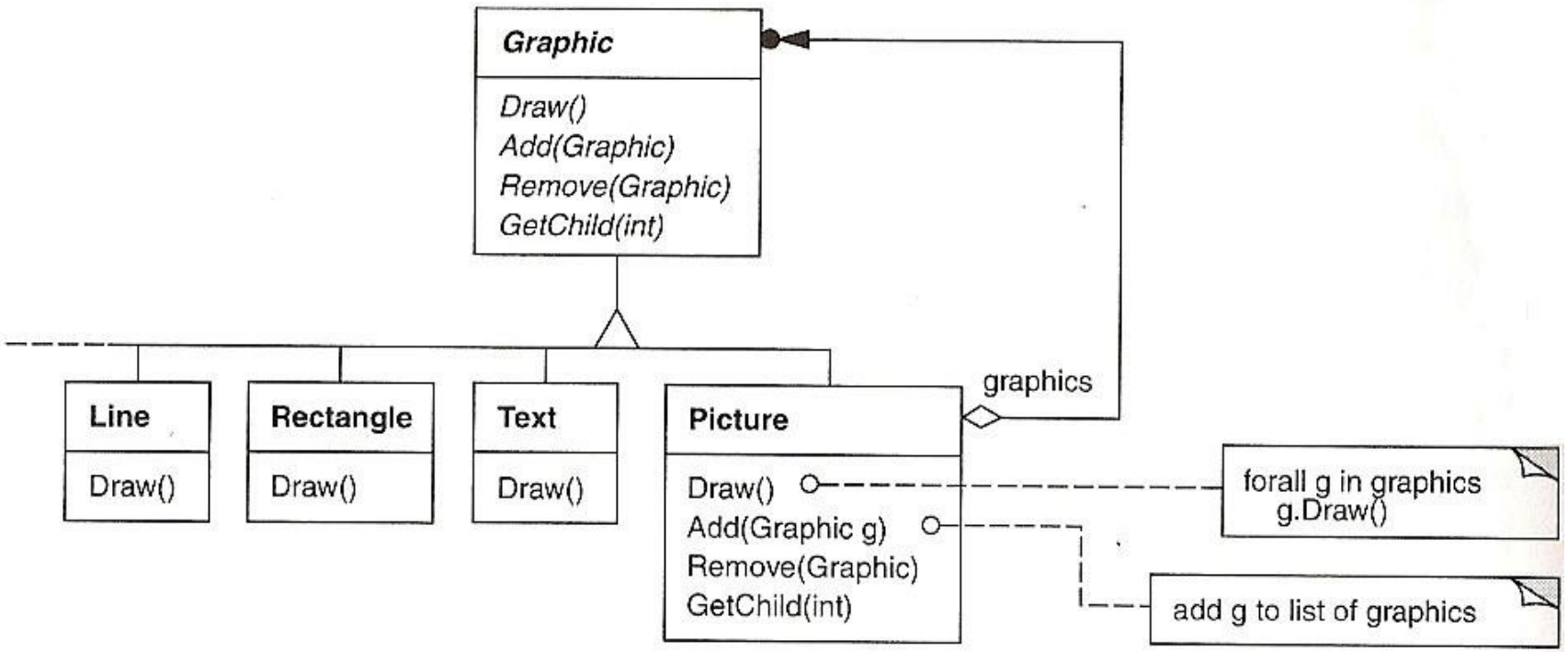


- Participantes
  - Strategy – define uma interface comum para todos dos algoritmos suportados
  - ConcreteStrategy – implementa o algoritmo usando a interface de Strategy
  - Context
    - É configurado com um objeto do ConcreteStrategy
    - Mantém referência para Strategy
    - Pode definir uma interface que permita strategy acessar seus dados

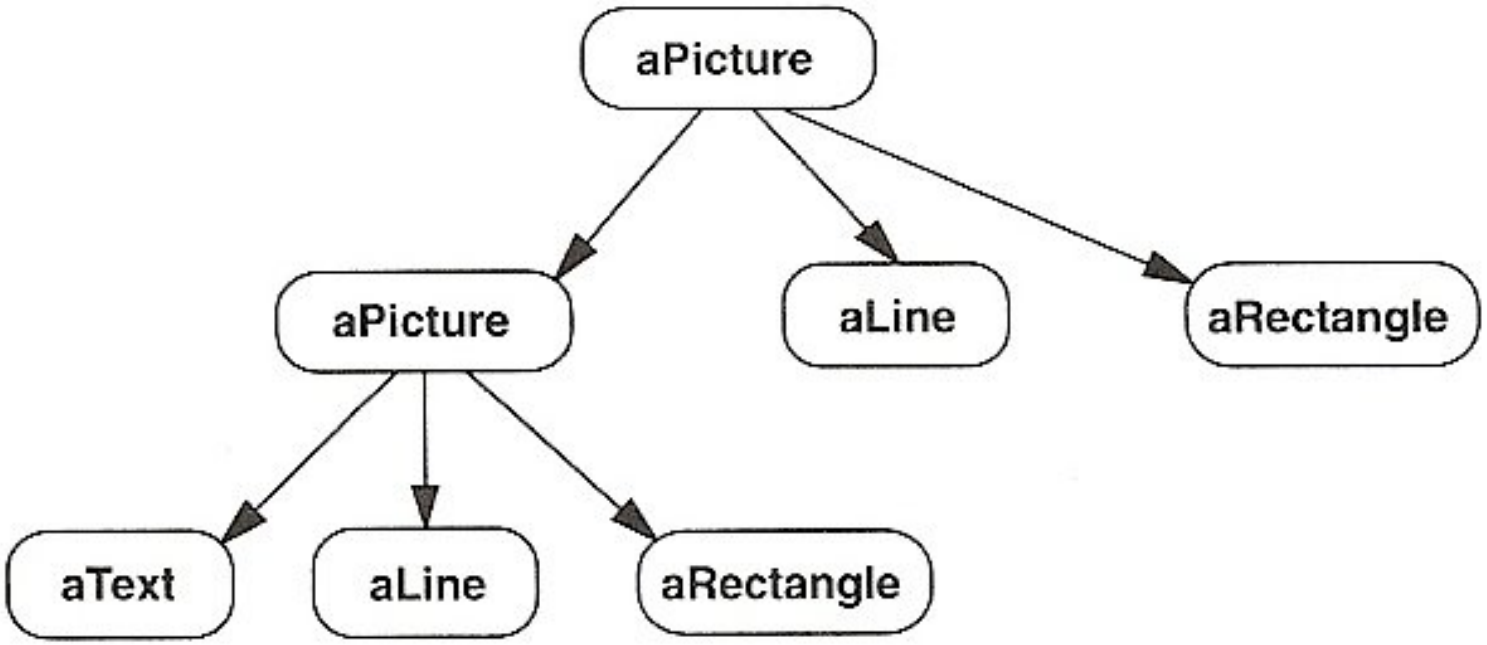
Composite

- Intenção: Compor objetos em estruturas em árvore para representarem hierarquias todo-partes. Composite permite aos clientes tratarem de maneira uniforme os objetos individuais e as composição de objetos.

Composite

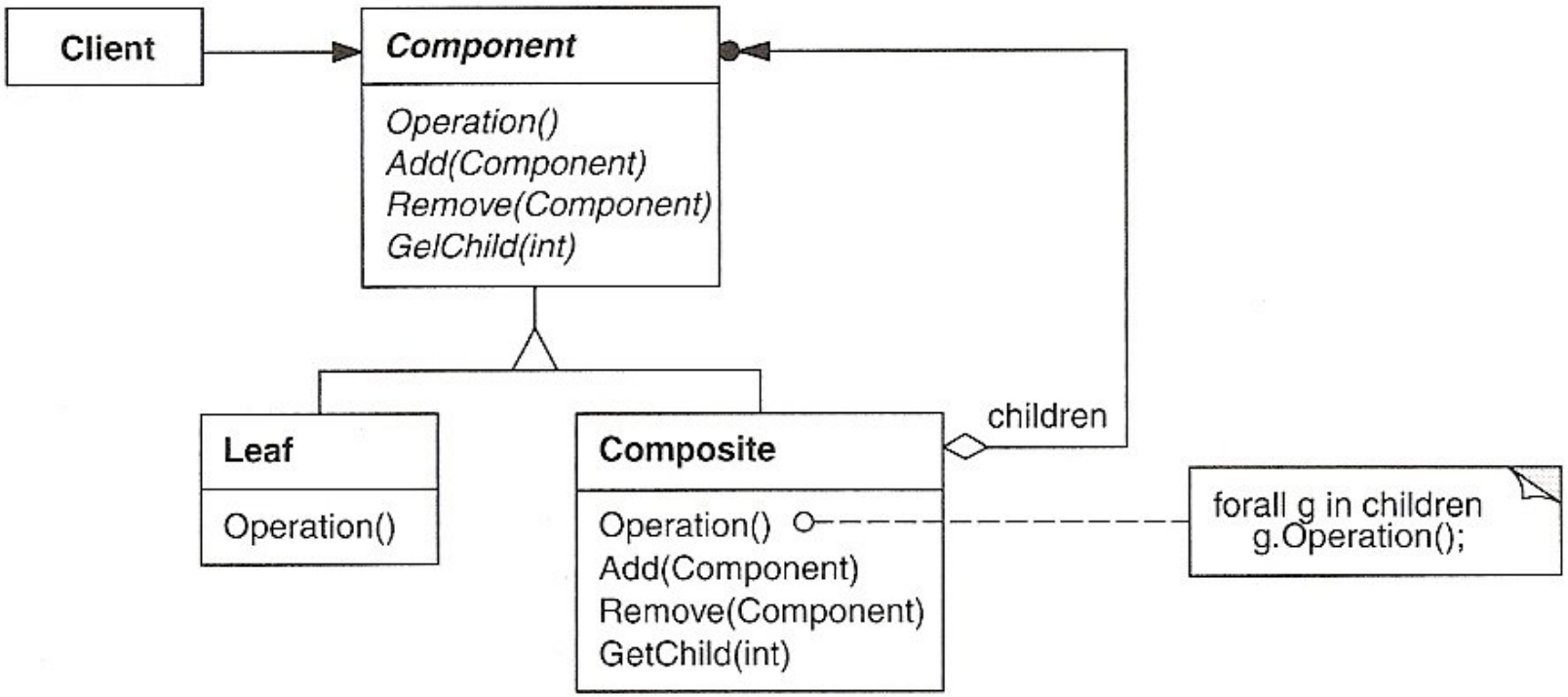

Graphic
Draw()
Add(Graphic)
Remove(Graphic)
GetChild(int)
Line
Draw()
Rectangle
Draw()
Text
Draw()
Picture
Draw()
Add(Graphic g)
Remove(Graphic)
GetChild(int)
graphics
forall g in graphics
g.Draw()
add g to list of graphics

Composite


aPicture
aPicture
aLine
aRectangle
aText
aLine
aRectangle

- Aplicabilidade:
  - Representação de hierarquias todo-partes de objetos
  - Os clientes devem ser capazes de ignorar a diferença entre composições de objetos e objetos individuais. Os clientes tratarão todos os objetos na estrutura de maneira uniforme.

- Estrutura Abstrata



- Partipantes
  - Component
    - Declara a interface para objetos na composição
    - Implementa o comportamento *default* para interface comum
    - Declara uma interface para acessar e gerencias seus componentes-filhos
    - (opcional) define uma interface para acessar o pai de um componente na estrutura recursiva e a implementa se isto for necessário

Composite

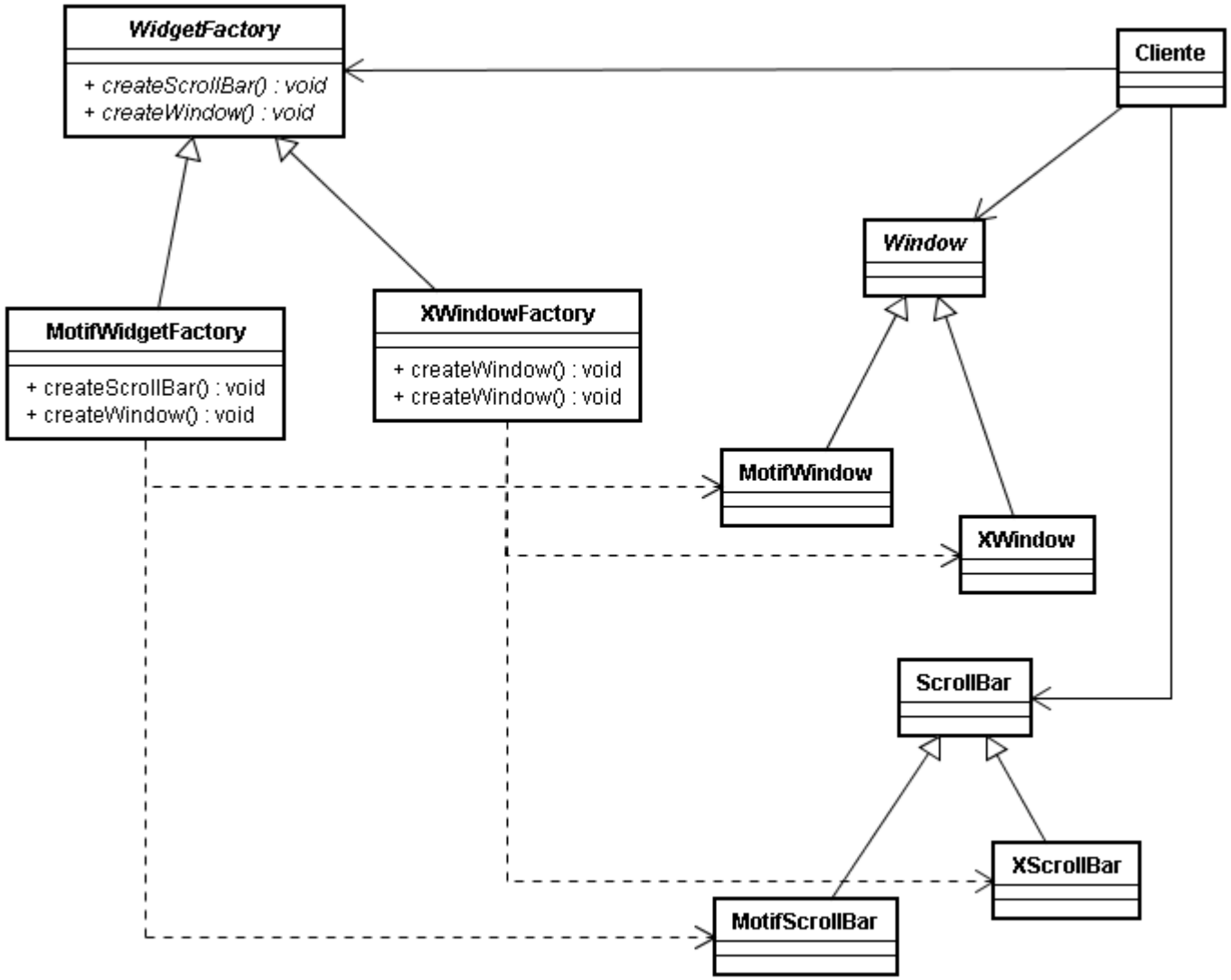
- Partipantes
  - Leaf
    - Representa objetos-folha (não tem filhos) na composição
    - Define o comportamento para objetos primitivos na composição
  - Composite
    - Define comportamento para componentes que tem filhos
    - Armazena os **componentes-filhos** (folhas e compostos)
    - Implementa as operações relacionadas com os filhos presentes na interface de Component

- Partipantes
  - Client
    - Manipula os objetos na composição através da interface de Component

Abstract Factory

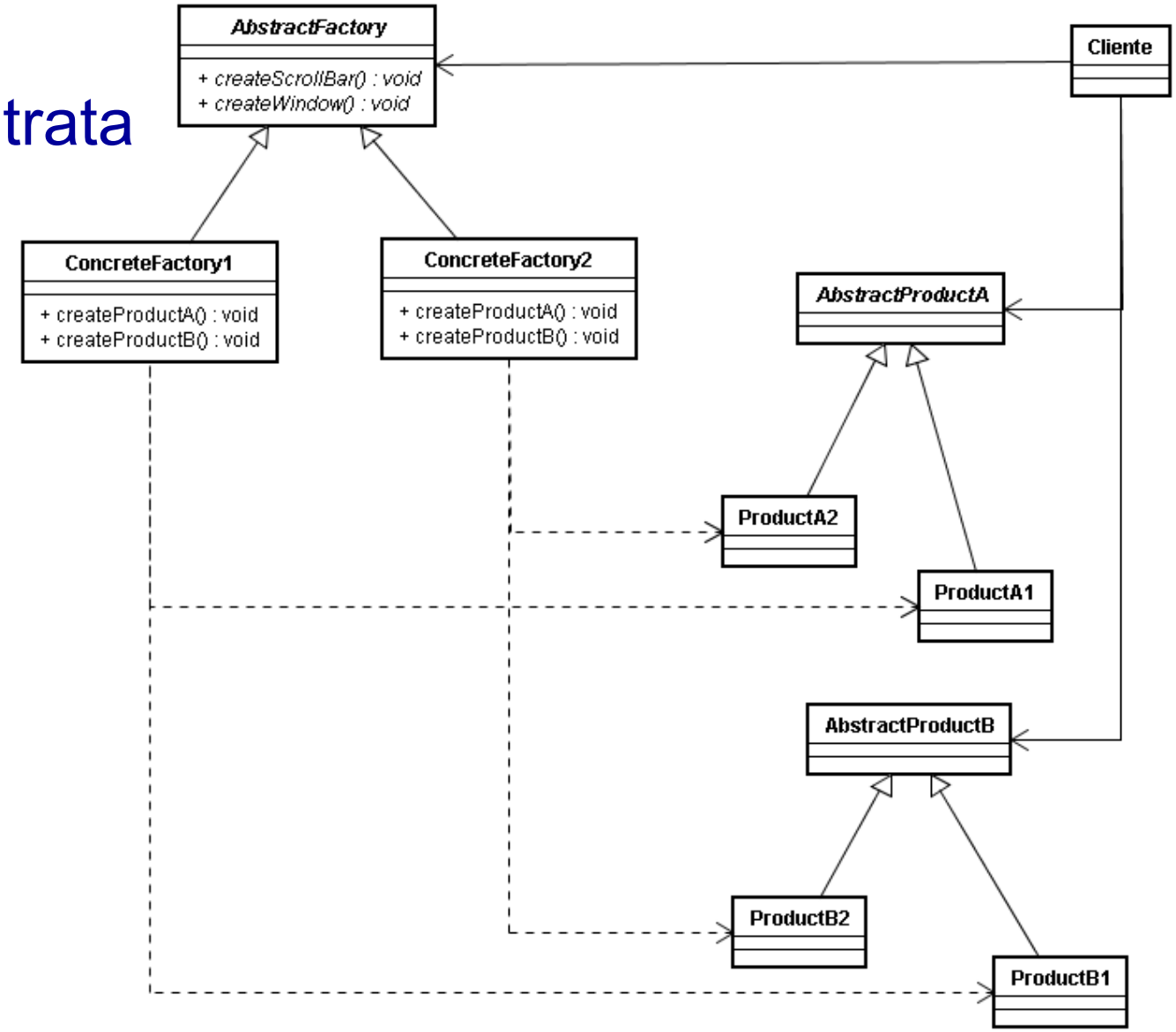
- Intenção: Fornecer uma interface para criação de famílias de objetos relacionados ou dependentes sem especificar suas classes concretas

Abstract Factory



- Aplicabilidade
  - Um sistema deve ser independente de como os seus produtos são criados, compostos ou representados
  - Um sistema deve ser configurado como um produto de uma família de múltiplos produtos
  - Uma família de objetos-produto for projetada para ser usada em conjunto e é necessário garantir esta restrição
  - Deseja-se fornecer uma biblioteca de classes de produtos e somente suas interfaces serão reveladas – a implementação deve ser ocultada

Estrutura abstrata



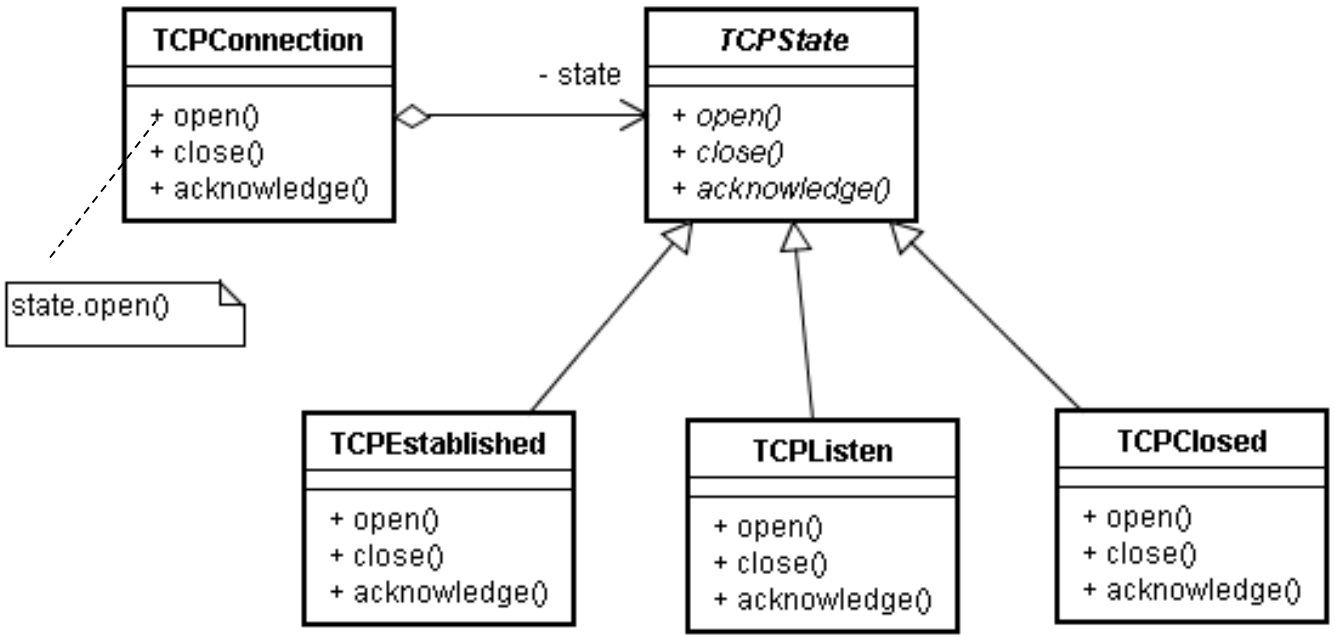
- Participantes
  - AbstractFactory
    - Declara uma interface para operações de criação de objetos-produtos abstratos
  - ConcreteFactory
    - Implementa a interface que cria objetos produtos concretos
  - AbstractProduct
    - Declara uma interface para um tipo de objeto-produto

Abstract Factory

- ConcreteProduct
  - Define um objeto-produto a ser criado pela correspondente da fábrica concreta
- Client
  - Usa somente interfaces declaradas pelas classes AbstractFactory e AbstractProduct

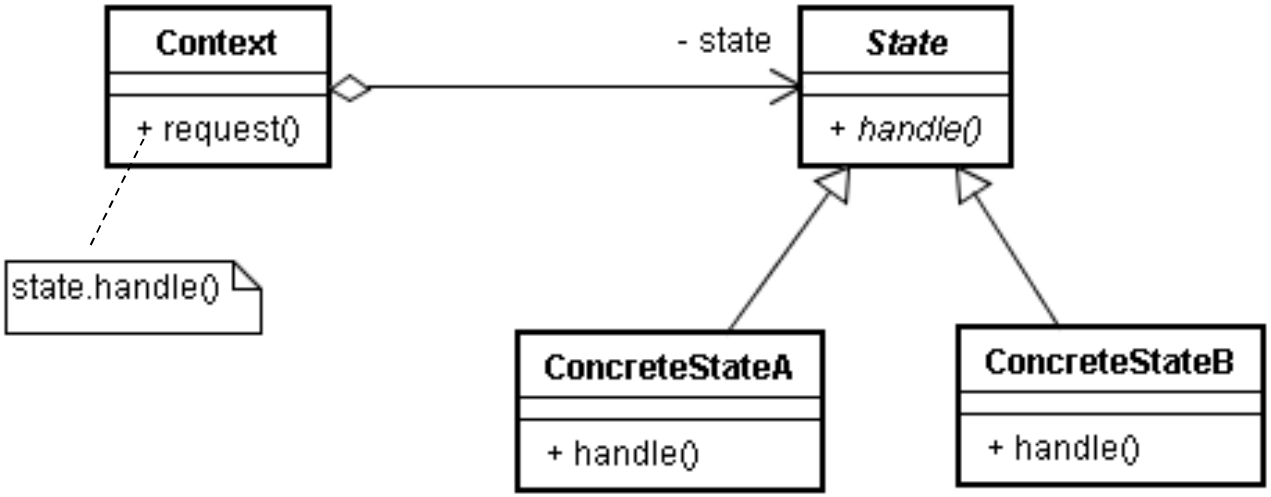
State

- Intenção: Permite um objeto alterar seu comportamento quando o seu estado interno muda. O objeto parecerá ter mudado de classe



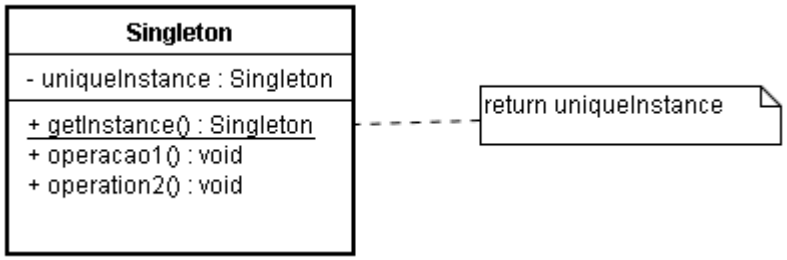
- Aplicabilidade:
  - O comportamento de um objeto depende do seu estado e ele pode mudar seu comportamento em tempo de execução
  - Operações têm comandos condicionais grandes, com várias alternativas que dependem do estado do objeto, normalmente representado por constantes enumeradas

Estrutura abstrata



- Participantes
  - Context
    - Define a interface de interesse para os clientes
    - Mantém uma instância de uma subclasse de ConcreteState a qual define o estado corrente
  - State
    - Define uma interface para encapsulamento de um determinado estado de Context
      - ConcreteState: cada classe implementa um possível estado de State

- Intenção: Garantir que uma classe tenha somente uma instância e fornecer um ponto global de acesso para a mesma

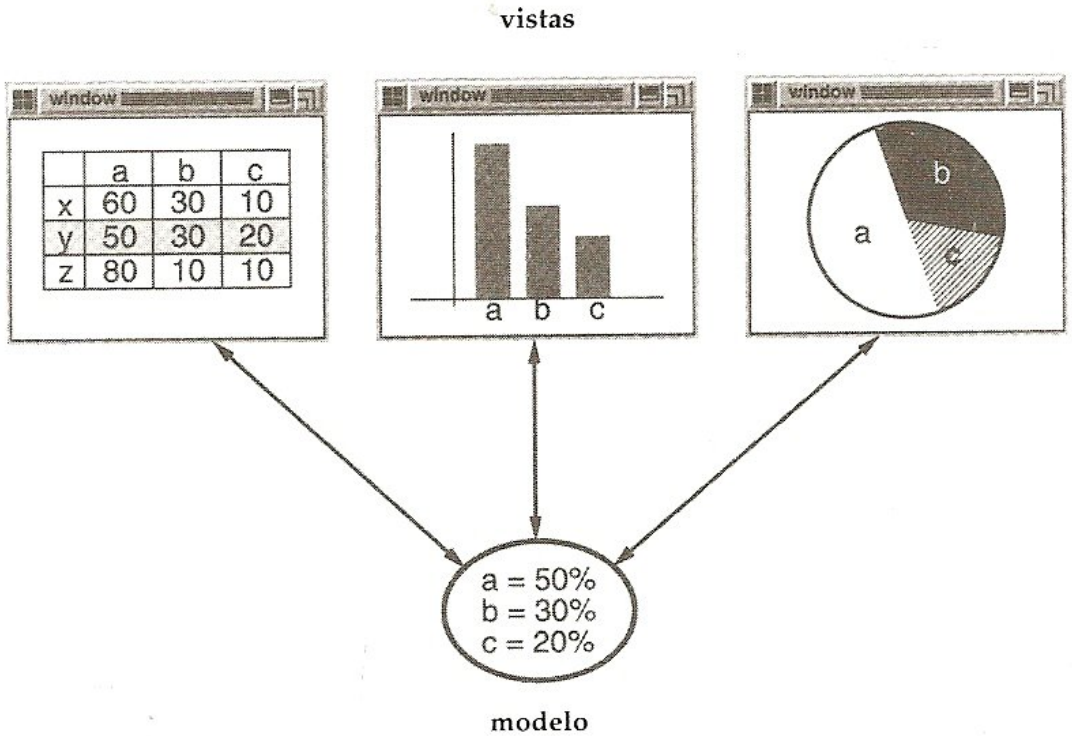


- Aplicabilidade
  - Deve haver apenas uma instância de uma classe e essa instância deve ser acessível através de um ponto bem conhecido
  - Quando uma única instância for extensível através de subclasses permitindo variar o comportamento da instância sem afetar o código dos clientes

- Participantes
  - Singleton
    - Define uma operação getInstance que permite aos clientes acessarem sua única instância. getInstance é uma operação estática ou de classe
    - Pode ser responsável pela criação da sua própria instância única (lazy creation)

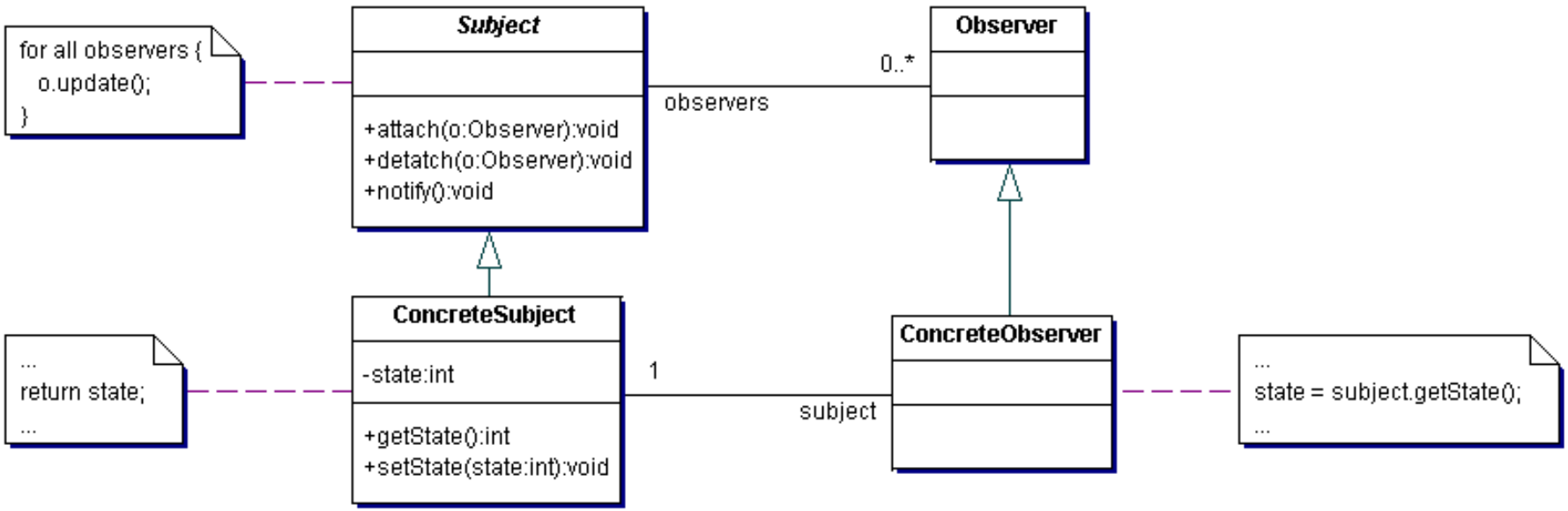
Observer

- Intenção: Definir uma dependência de um para muitos entre objetos, de maneira que quando um objeto muda de estado, todos os seus dependentes são notificados e atualizados automaticamente



- Aplicabilidade
  - Quando uma abstração tem dois aspectos e um depende do outro (ex. modelo e visão)
  - Quando uma mudança em um objeto exige mudança em outro e não é possível saber quantos objetos precisam ser mudados
  - Quando um objeto deveria ser capaz de notificar outros objetos sem saber quem são estes objetos (desacoplamento)

- Estrutura Abstrata



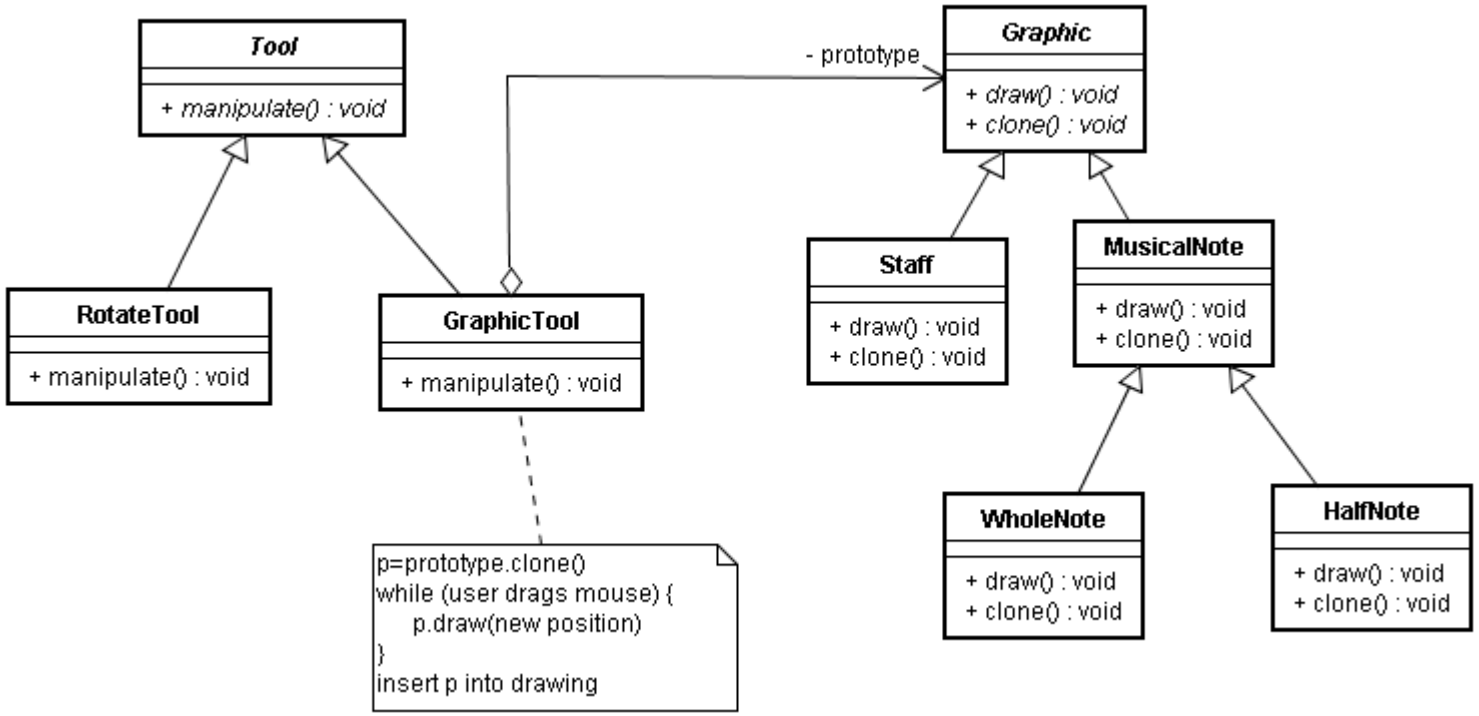
Observer

- Participantes
  - Subject
    - Conhece seus observadores
    - Fornece interface para acrescentar e remover observadores
  - Observer
    - Define uma interface de atualização para objetos que deveriam ser notificados sobre mudanças em subject
  - Concrete Subject
    - Armazena estados de interesse para objetos ConcreteObserver
    - Envia notificação quando seu estado muda

- ConcreteObserver
  - Mantém uma referência para um objeto ConcreteSubject
  - Armazena estados que deveriam ser consistentes com os de Subject
  - Implementa a interface de atualização de Observer

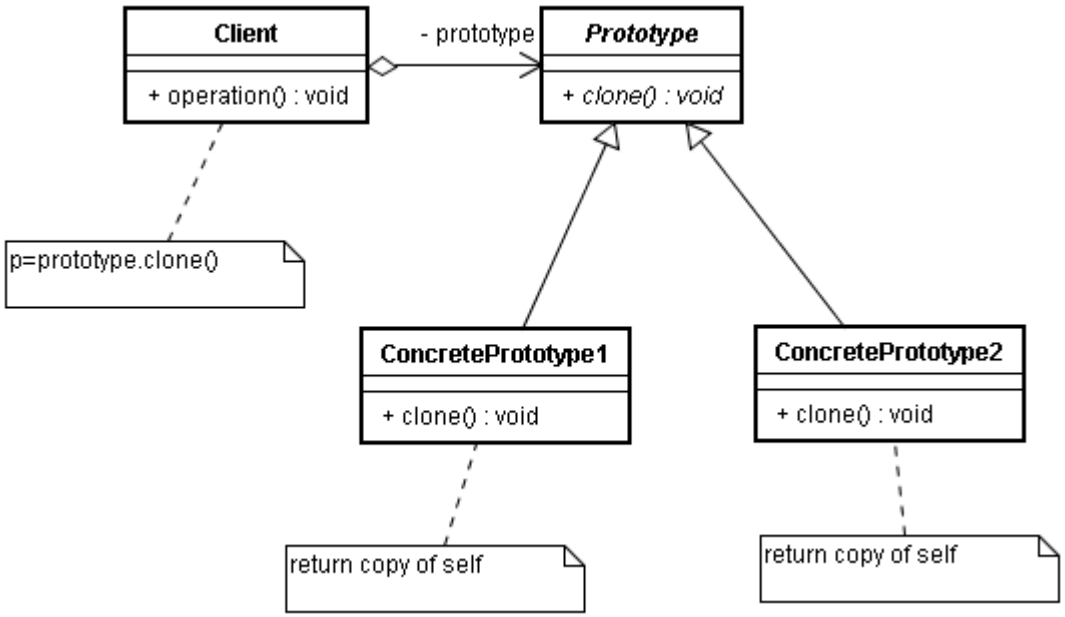
Prototype

- Intenção: Especificar tipos de objetos a serem criados usando uma instância protótipo e permitir a criação de novos objetos pela cópia do protótipo



- Aplicabilidade
  - Criação de instâncias que são especificadas em tempo de execução
  - Evitar a criação de uma hierarquia de classes fábrica paralela a hierarquias de classes produtos
  - Quando as instâncias de uma classe puderem ter uma dentre poucas combinações diferentes de estados – pode ser mais mais conveniente um número de protótipos correspondente e cloná-los ao invés de instanciar a classe manualmente, e inicializar o estado apropriado

- Estrutura Abstrata



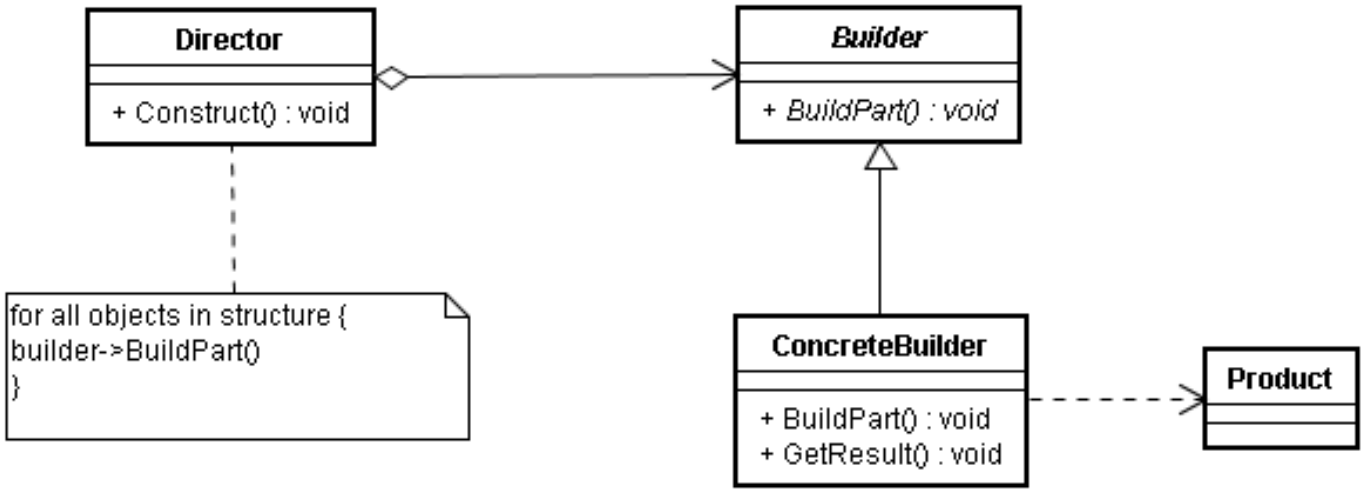
- Participantes
  - Prototype
    - Declara uma interface para clonar a si próprio
  - ConcretePrototype
    - Implementa uma operação para clonar a si próprio
  - Client
    - Cria um novo objeto solicitando a um protótipo que clone a si próprio

Builder

- Intenção: Separar a construção de um objeto complexo da sua representação de modo que o processo de construção possa criar diferentes representações

- Aplicabilidade
  - Algoritmo para criação de um objeto complexo deve ser independente das partes que compõem o objeto e de como elas são montadas
  - O processo de construção deve permitir diferentes representações para o objeto que é construído

- Estrutura Abstrata



Builder

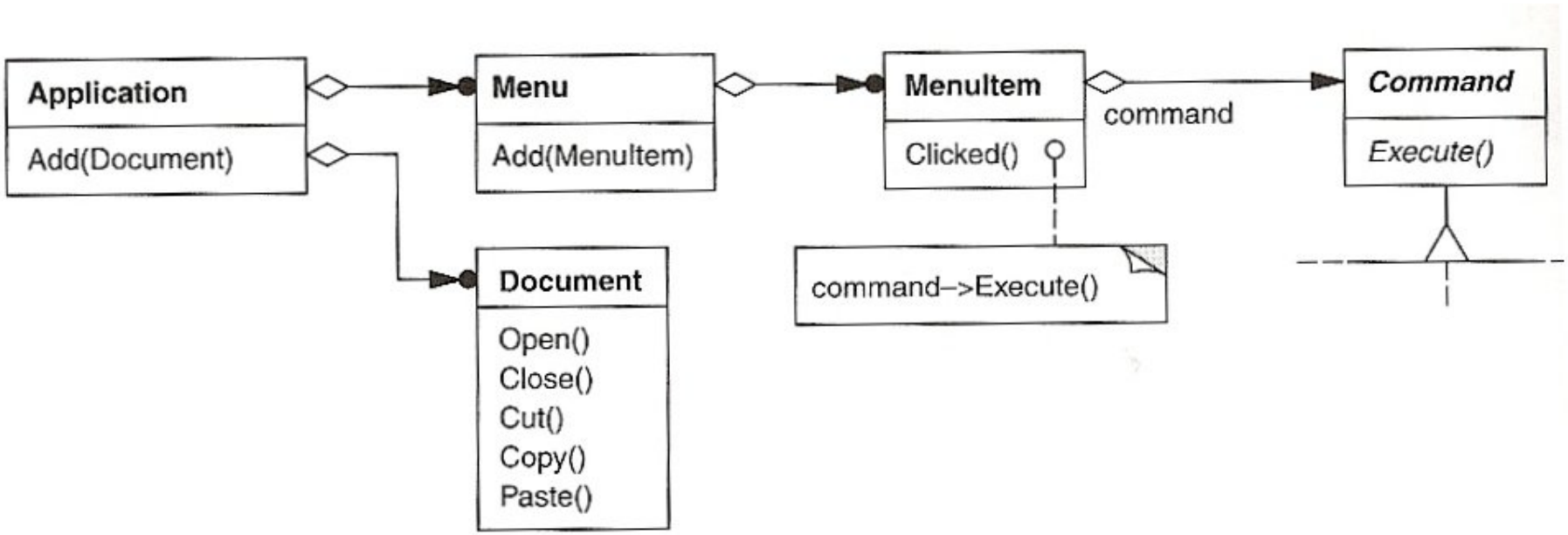
- Participantes
  - Builder
    - Especifica uma interface abstrata para a criação de partes de um objeto-produto
  - ConcreteBuilder
    - Constrói e monta as partes do produto pela implementação da interface de Builder
    - Define e mantém a representação que cria
    - Fornece uma interface para recuperação do produto

Builder

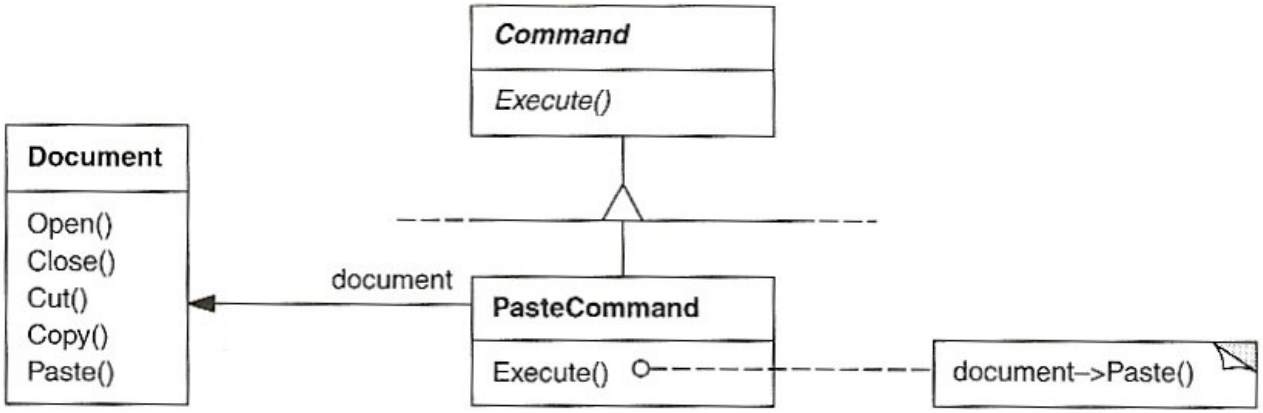
- Director
  - Constrói um objeto usando a interface de Builder
- Product
  - Representa o objeto complexo em construção. ConcreteBuilder constrói a representação interna do produto e define o processo pelo qual ele é montado
  - Inclui classes que definem as partes constituintes , inclusive as interfaces para a montagem das partes no resultado final

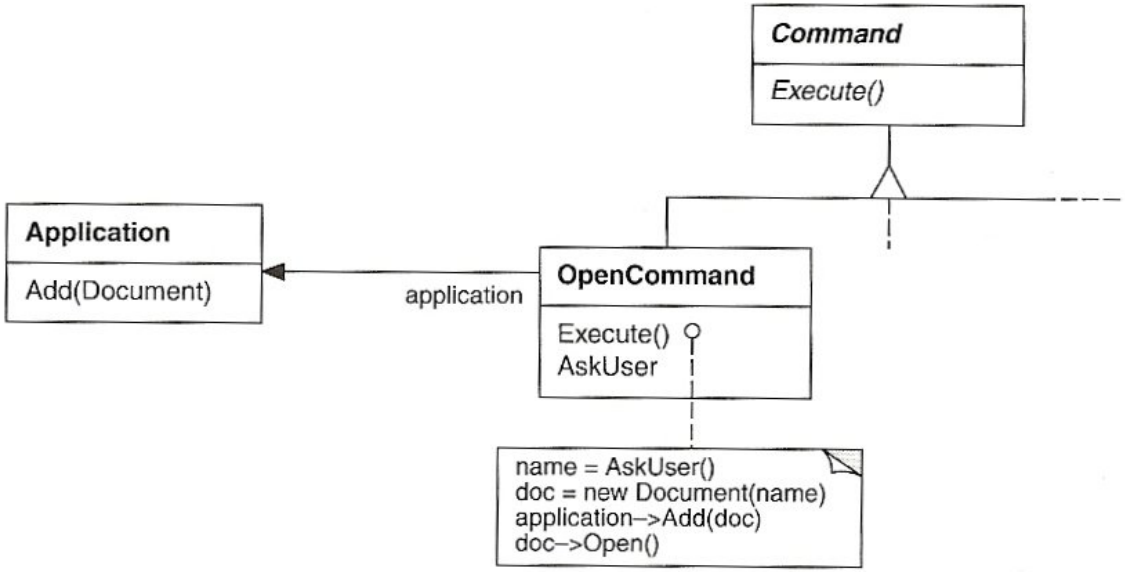
Command

- Intenção: Encapsular a solicitação de um objeto permitindo parametrizar clientes com diferentes solicitações, enfileirar e fazer registro (log) de solicitações e suportar operações de podem ser desfeitas (undo)



Command

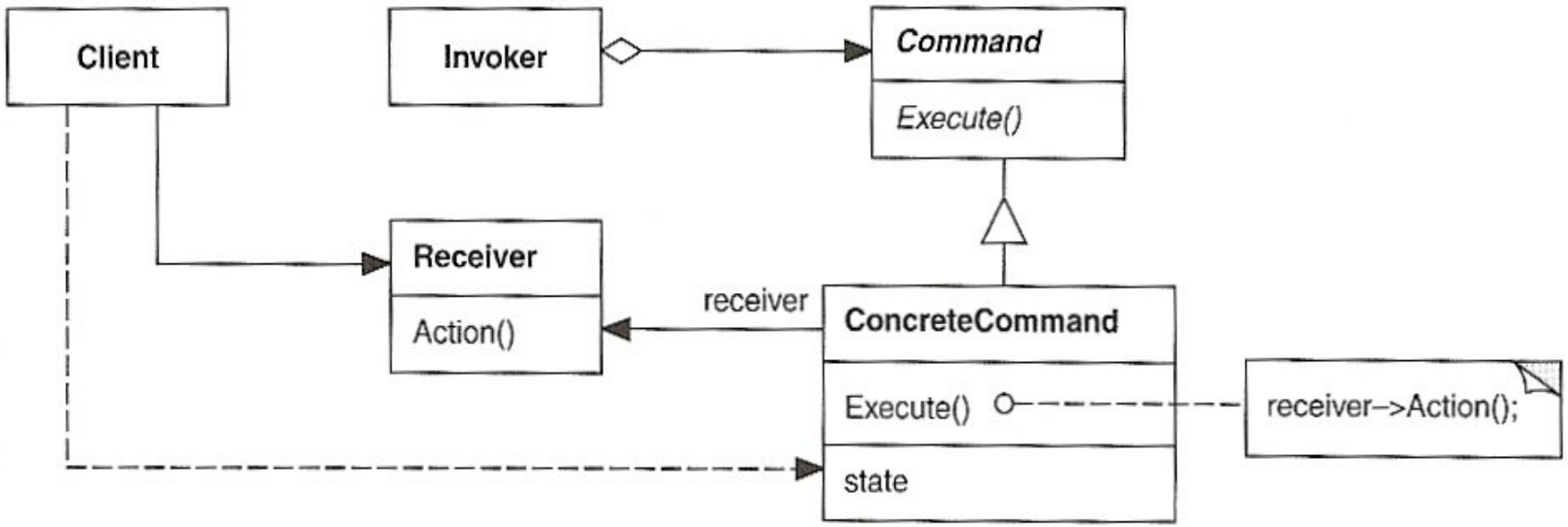

Command
Execute()
Document
Open()
Close()
Cut()
Copy()
Paste()
document
PasteCommand
Execute()
document->Paste()


Command
Execute()
Application
Add(Document)
application
OpenCommand
Execute()
AskUser
name = AskUser()
doc = new Document(name)
application->Add(doc)
doc->Open()

- Aplicação
  - Parametrizar objetos por uma ação a ser executada. Permitem substituir os *callbacks* ...
  - Suportar *undo* de operações. A operação *execute* pode armazenar estados para reverter seus efeitos no próprio comando. Uma operação *unexecute* pode reverter os efeitos de execute.
  - Suportar o registro (logging) de mudanças – Uma interface de command com operações para carregar e armazenar permite persistir mudanças. Na recuperação de uma queda os comandos persistidos podem ser reaplicados

- Aplicação
  - Estruturar um sistema em torno de operações de alto nível construídas sobre primitivas. Esta característica é comum em sistemas que suportam transações

- Estrutura


Client
Invoker
Command
Execute()
Receiver
Action()
receiver
ConcreteCommand
Execute()
state
receiver->Action();

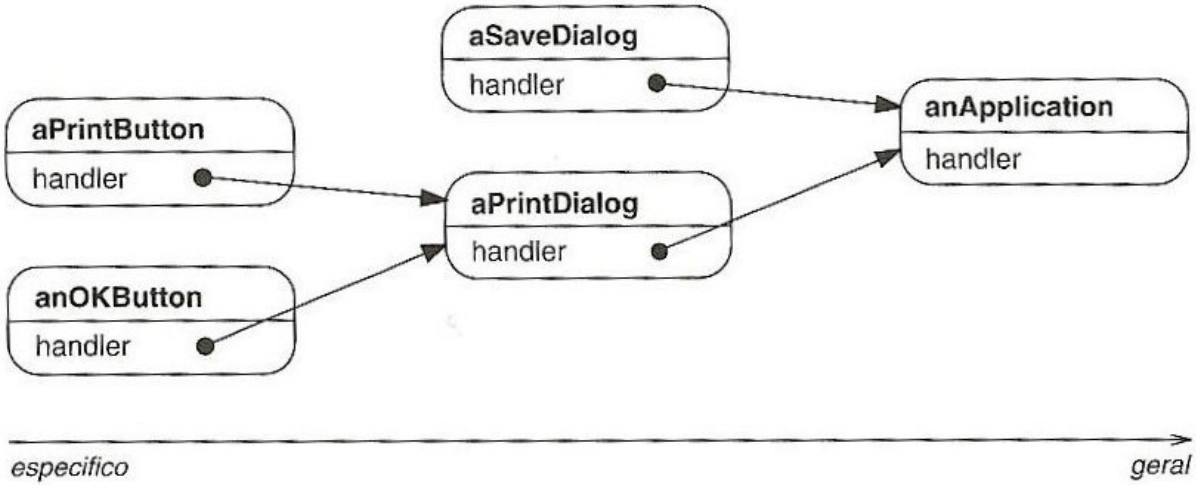
Command

- Participantes
  - Command
    - Declara uma interface para a execução de uma operação
  - ConcreteCommand
    - Define uma vinculação entre um objeto Receiver e uma ação
    - Implementa Execute através da invocação da(s) correspondente(s) operação (ões) no Receiver

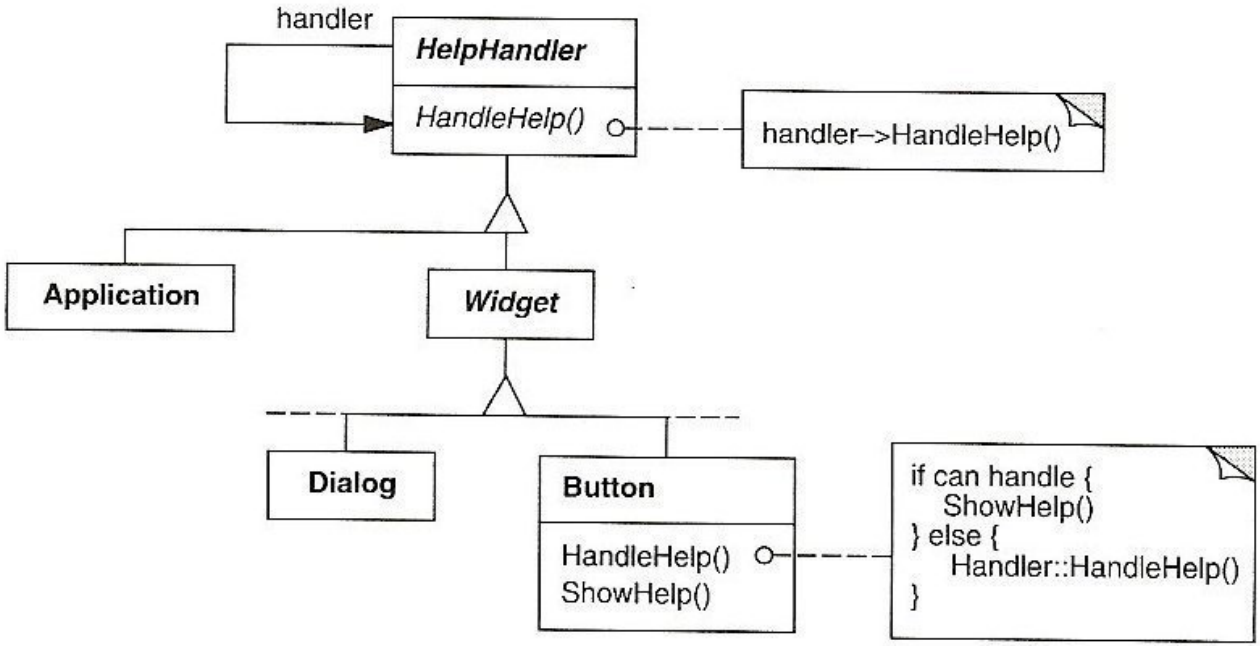
- Participantes
  - Client
    - Cria um objeto *ConcreteCommand* e estabelece o seu receptor (reviver)
  - Invoker
    - Solicita ao Command a execução da solicitação
  - Receiver
    - Sabe como executar as operações associadas a uma solicitação. Qualquer classe pode funcionar como um receiver

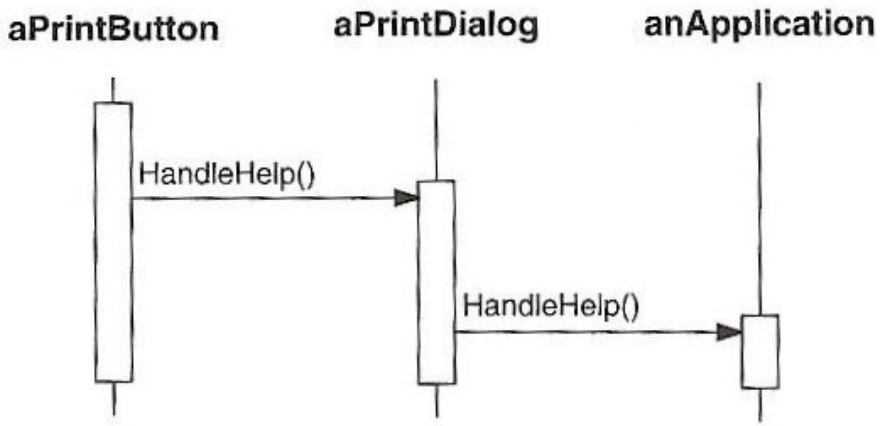
Chain of Responsibility

- Intenção: Evitar o acoplamento do remetente de uma solicitação e seu receptor, ao dar a mais de um objeto a oportunidade de tratar a solicitação. Encadear os objetos receptores, passando a solicitação ao longo da cadeia até que um objeto a trate.



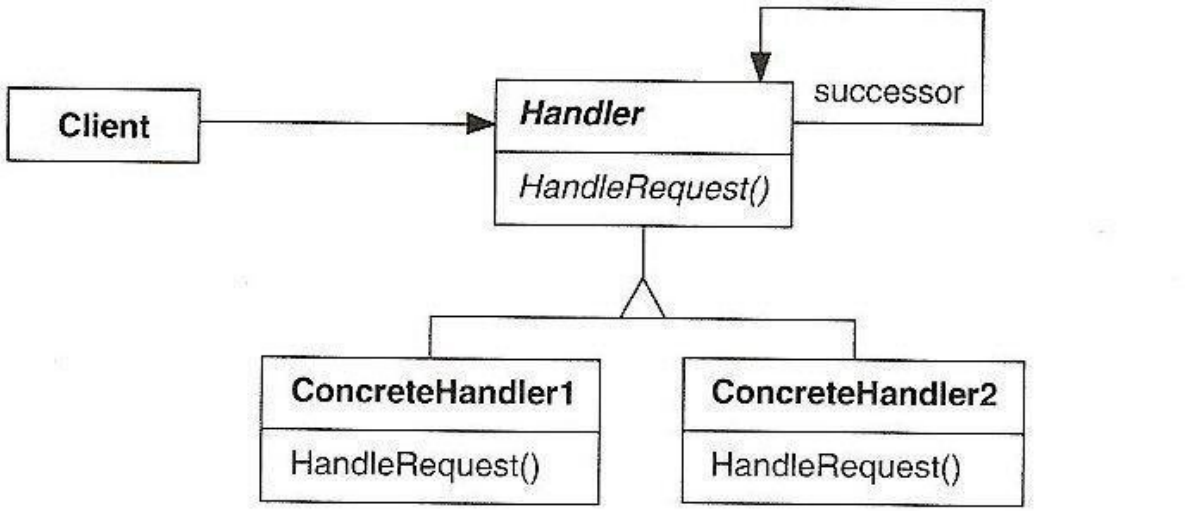
Chain of Responsibility

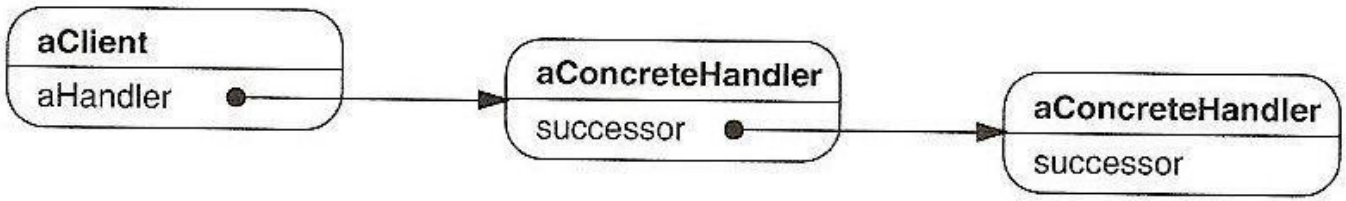




- Aplicação
  - Mais de um objeto pode tratar uma solicitação e o objeto que a tratará não é conhecido a priori
  - Deseja-se emitir uma solicitação para um dentre vários objetos, sem especificar explicitamente o receptor
  - O conjunto de objetos que pode tratar uma solicitação pode ser especificado dinamicamente.

- Estrutura


Client
Handler
HandleRequest()
successor
ConcreteHandler1
HandleRequest()
ConcreteHandler2
HandleRequest()

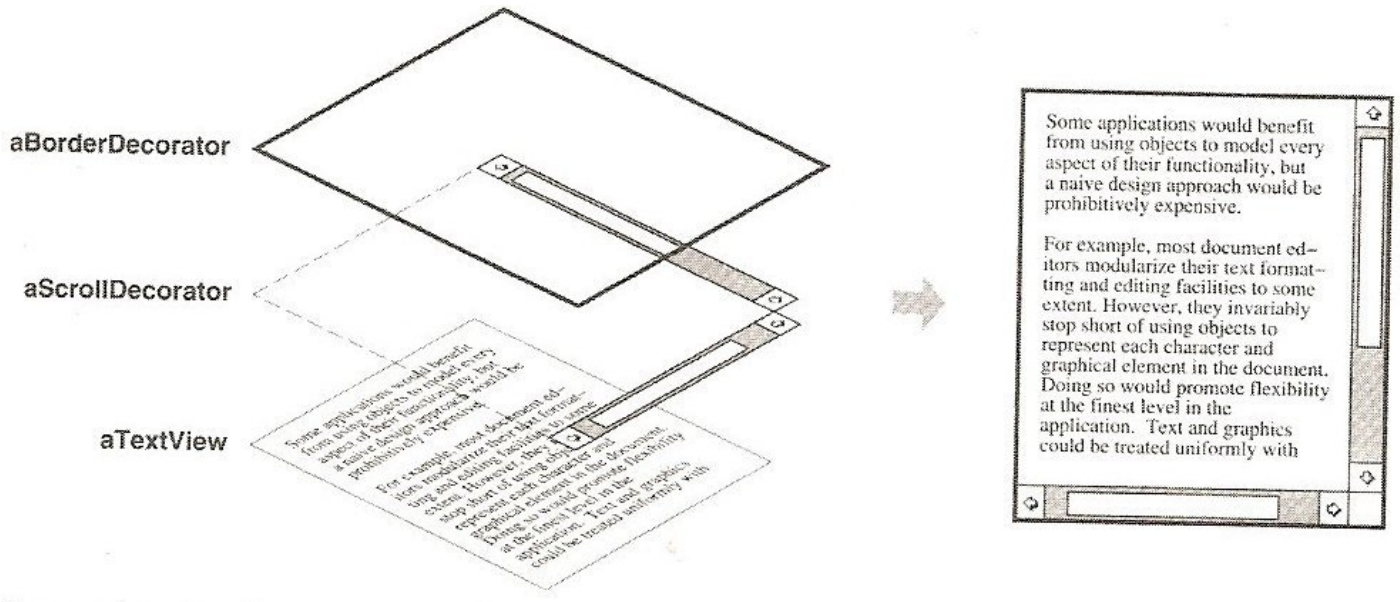

aClient
aHandler
aConcreteHandler
successor
aConcreteHandler
successor

- Participantes
  - Handler
    - Define uma interface para tratar solicitações
    - (opcional) o elo (link) ao sucessor
  - ConcreteHandler
    - Trata as solicitações pelas quais é responsável
    - Pode acessar seu sucessor
    - Se o concreteHandler pode tratar a solicitação, ele assim o faz, caso contrário ele repassa a solicitação para seu sucessor
  - Cliente
    - Inicia a solicitação para um objeto ConcreteHandler da cadeia

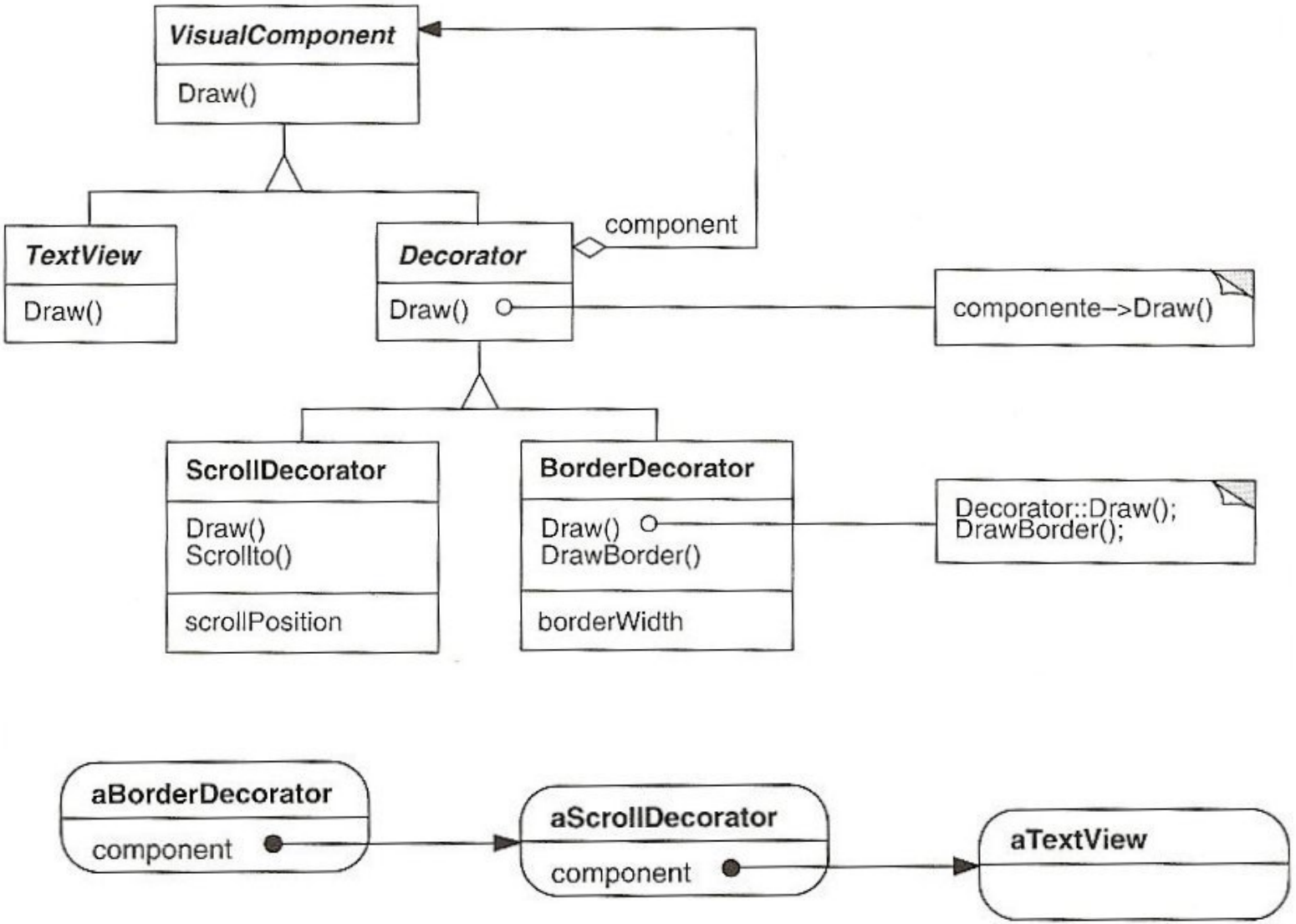
- Participantes
  - Client
    - Cria um objeto *ConcreteCommand* e estabelece o seu receptor (reviver)
  - Invoker
    - Solicita ao Command a execução da solicitação
  - Receiver
    - Sabe como executar as operações associadas a uma solicitação. Qualquer classe pode funcionar como um receiver

Decorator

- Intenção: Dinamicamente agregar responsabilidades adicionais a um objeto. Os *decorators* fornecem uma alternativa flexível ao uso de subclasses para extensão de funcionalidades.
  - *Também conhecido como Wrapper*



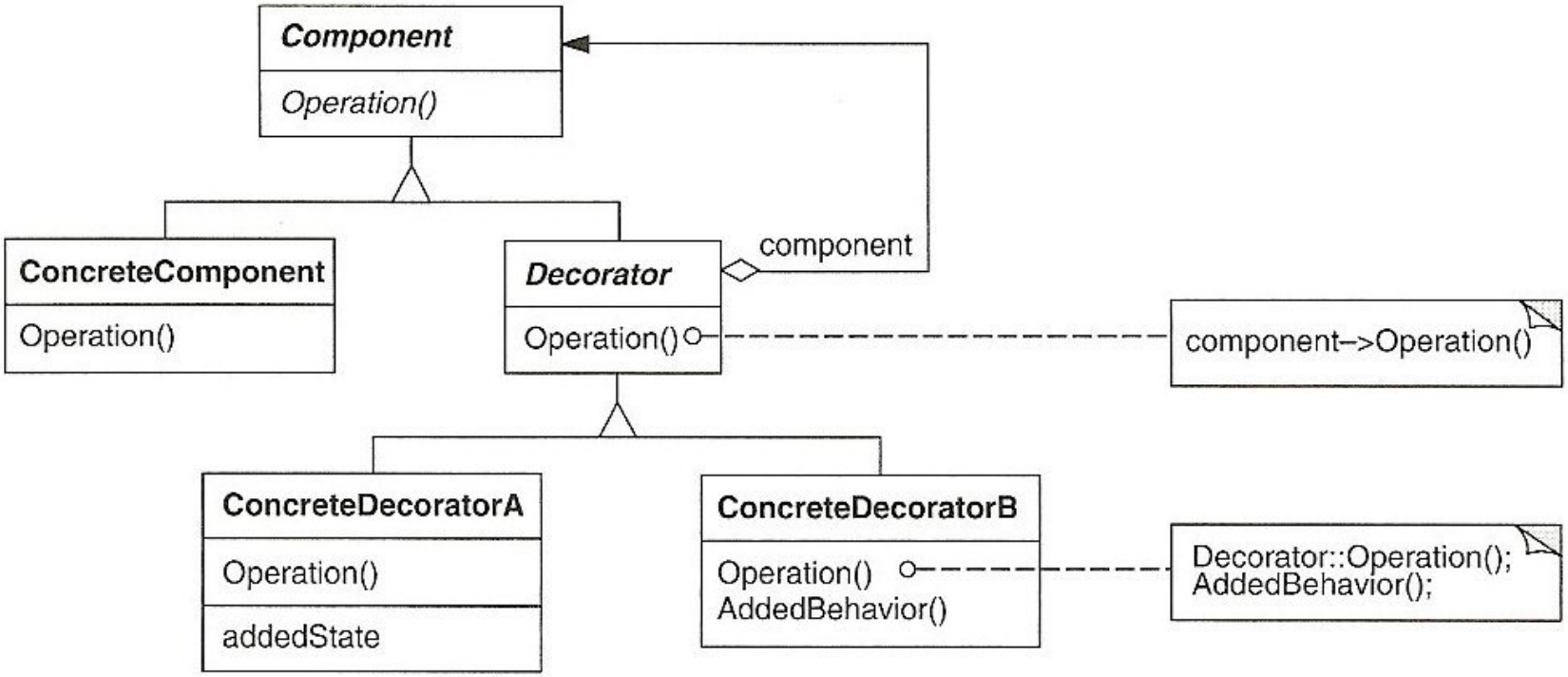
Decorator


VisualComponent
Draw()
TextView
Draw()
Decorator
Draw()
component
componente->Draw()
ScrollDecorator
Draw()
ScrollTo()
scrollPosition
BorderDecorator
Draw()
DrawBorder()
borderWidth
Decorator::Draw();
DrawBorder();
aBorderDecorator
component
aScrollDecorator
component
aTextView

- Aplicação
  - Acrescentar responsabilidades a objetos individuais de forma dinâmica e transparente, ou seja, sem afetar outros objetos
  - Para responsabilidades que podem ser removidas
  - Quando a extensão através do uso de subclasses não é prática. Às vezes um grande número de extensões independentes é possível e poderia produzir explosão de subclasses para cada combinação.

Decorator

- Estrutura



Decorator

- Participantes
  - Component
    - Define a interface para objetos que podem ter responsabilidades acrescentadas aos mesmos dinamicamente
  - ConcreteComponent
    - Define um objeto para o qual responsabilidades adicionais podem ser atribuídas
  - Decorator
    - Mantém uma referência para um objeto Componente e define uma interface que segue a interface de Component

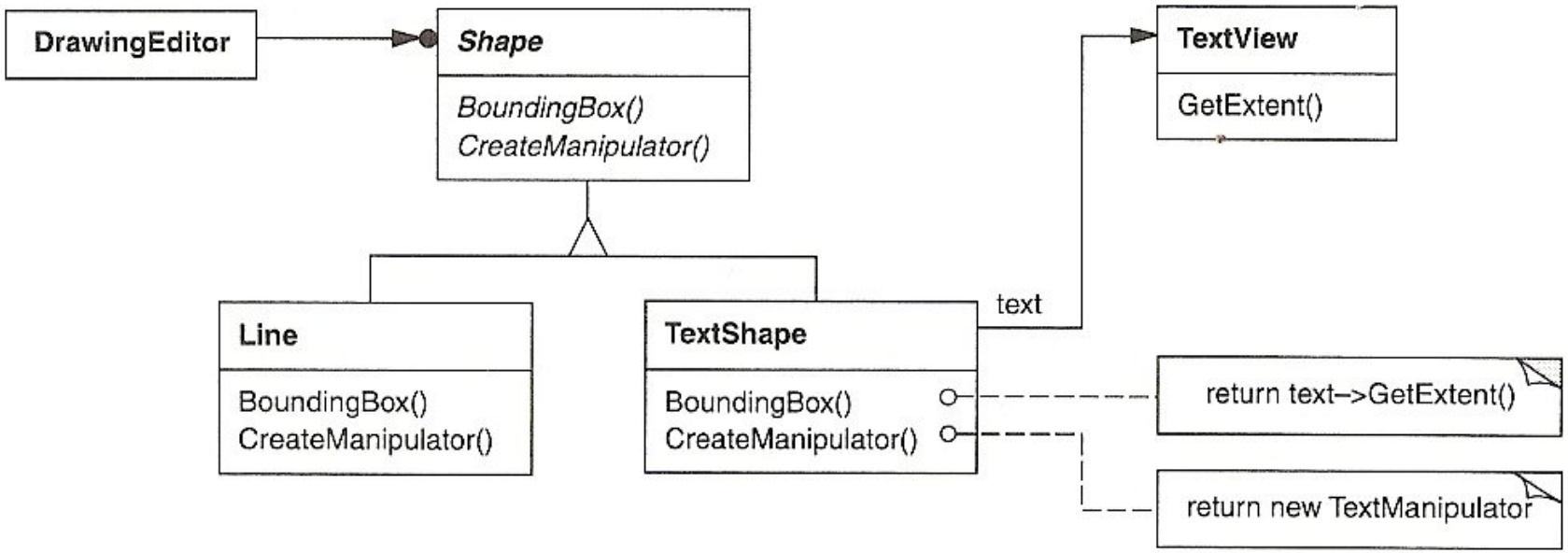
Decorator

- Participantes
  - ConcreteDecorator
    - Acrescenta responsabilidades ao componente

Adapter

- Intenção: Converter a interface de uma classe em outra interface, esperada pelos clientes. O Adapter permite que classes com interfaces incompatíveis trabalhem em conjunto – o que, de outra forma, seria impossível
  - *Também conhecido como Wrapper*

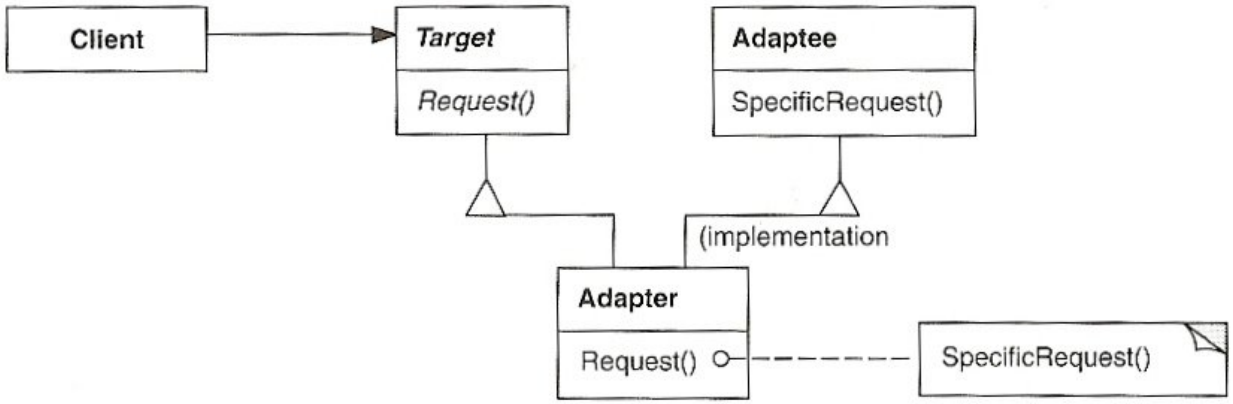
Adapter


DrawingEditor
Shape
BoundingBox()
CreateManipulator()
TextView
GetExtent()
Line
BoundingBox()
CreateManipulator()
TextShape
BoundingBox()
CreateManipulator()
text
return text->GetExtent()
return new TextManipulator

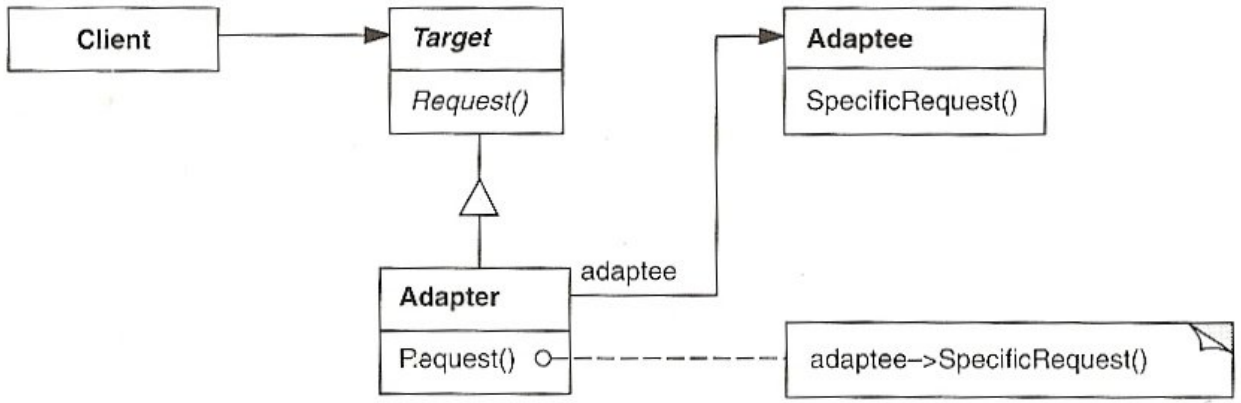
- Aplicação
  - É preciso utilizar uma classe existente, mas sua interface não corresponde a interface esperada
  - É necessário criar uma classe reutilizável que coopera com classes não-relacionadas ou previsíveis – ou seja, que não tem interfaces compatíveis
  - (adaptadores de objetos) – é necessário usar várias subclasses existentes, porém é impraticável adaptar estas interfaces através da criação de subclasses. Um adaptador de objeto pode ser usado para adaptar a interface de sua classe mãe.

- Estrutura

Um adaptador de classe usa a herança múltipla para adaptar uma interface à outra:

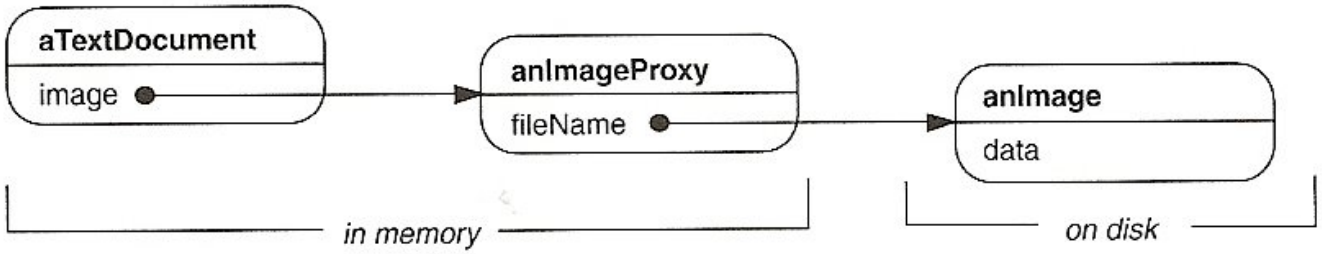


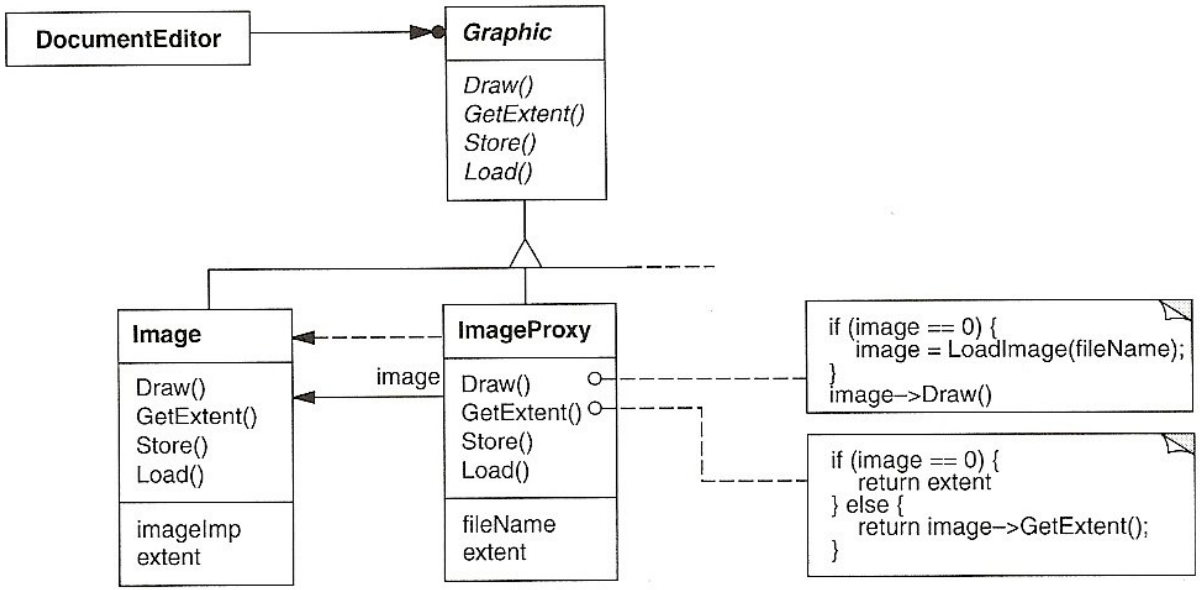
Um adaptador de objeto depende da composição de objetos:



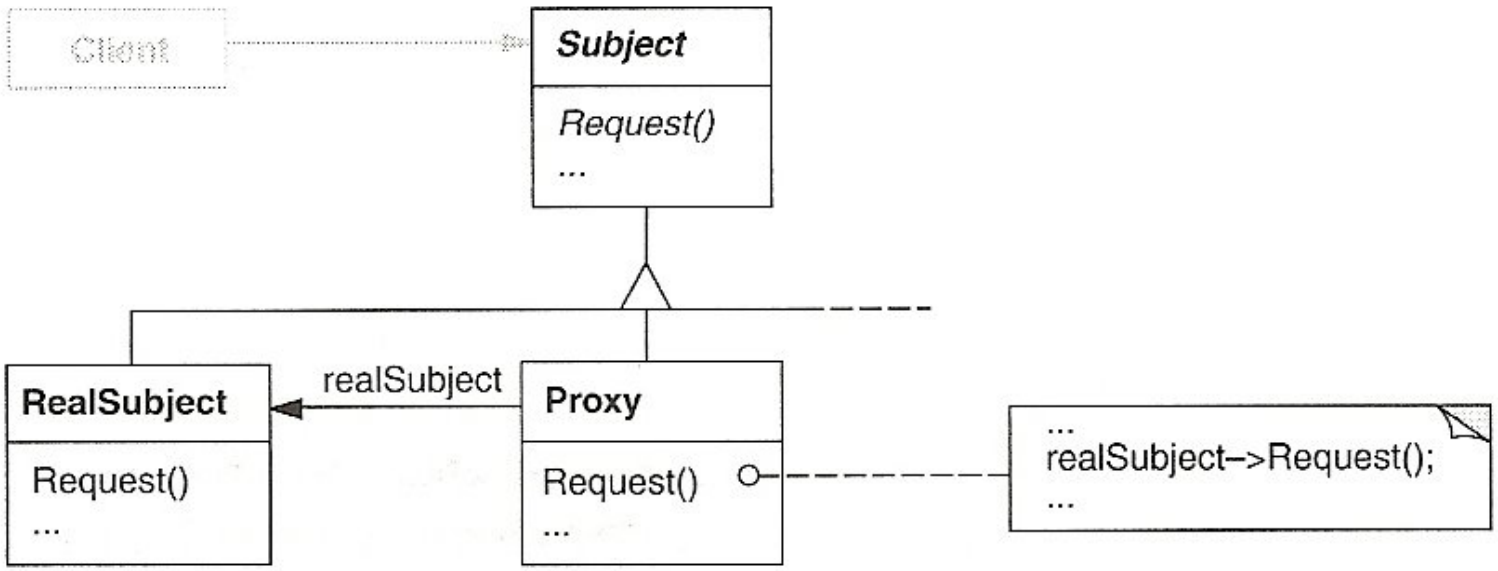
- Participantes
  - Target
    - Define a interface específica do domínio que o cliente usa.
  - Client
    - Colabora com objetos compatíveis com a interface de Target
  - Adaptee
    - Define uma interface existente que necessita ser adaptada
  - Adapter
    - Adapta a interface ao Adaptee à interface de Target

Proxy

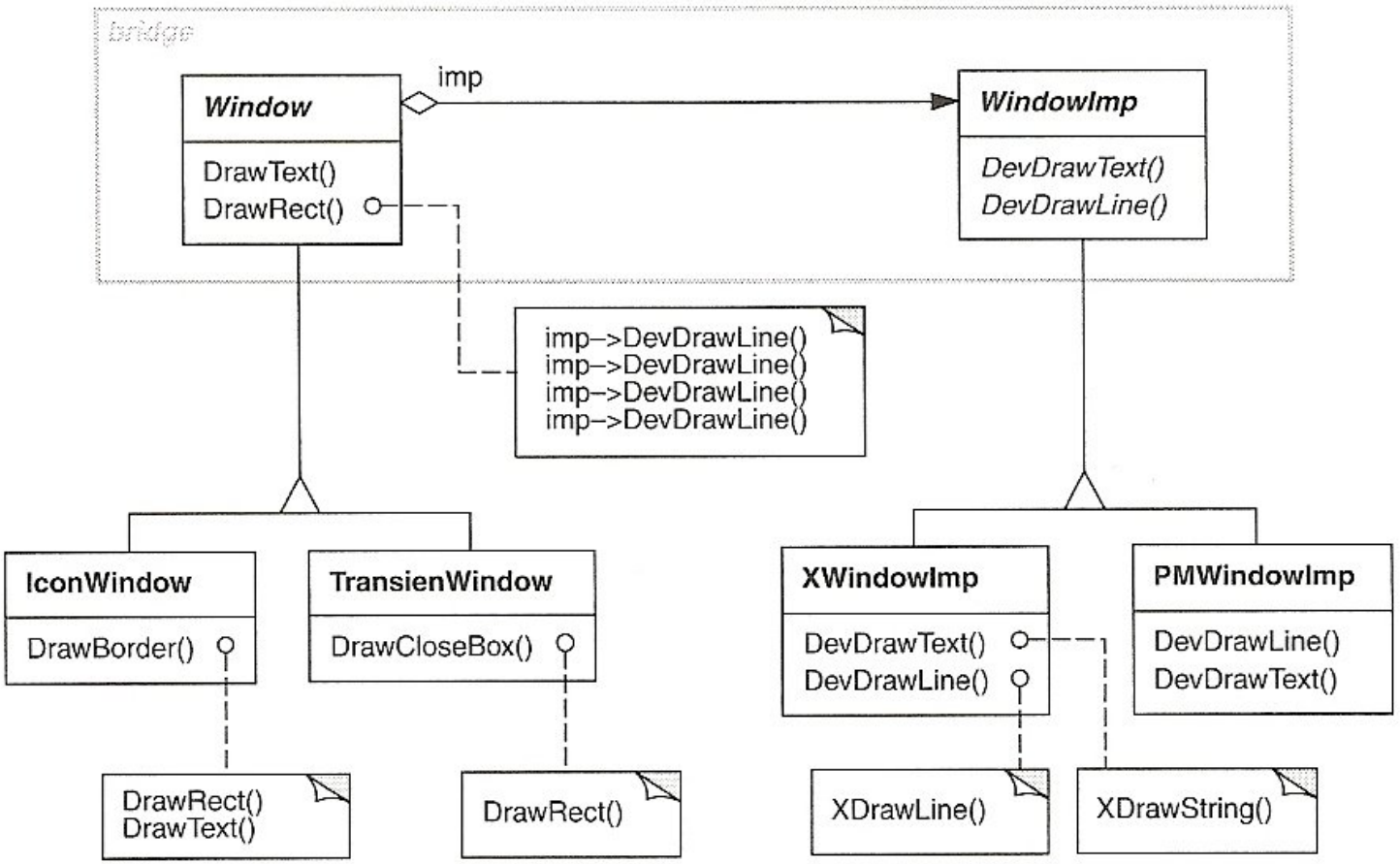

aTextDocument
image
anImageProxy
fileName
anImage
data
in memory
on disk


DocumentEditor
Graphic
Draw()
GetExtent()
Store()
Load()
Image
Draw()
GetExtent()
Store()
Load()
imageImp
extent
image
ImageProxy
Draw()
GetExtent()
Store()
Load()
fileName
extent
if (image == 0) {
image = LoadImage(fileName);
}
image->Draw()
if (image == 0) {
return extent
} else {
return image->GetExtent();
}

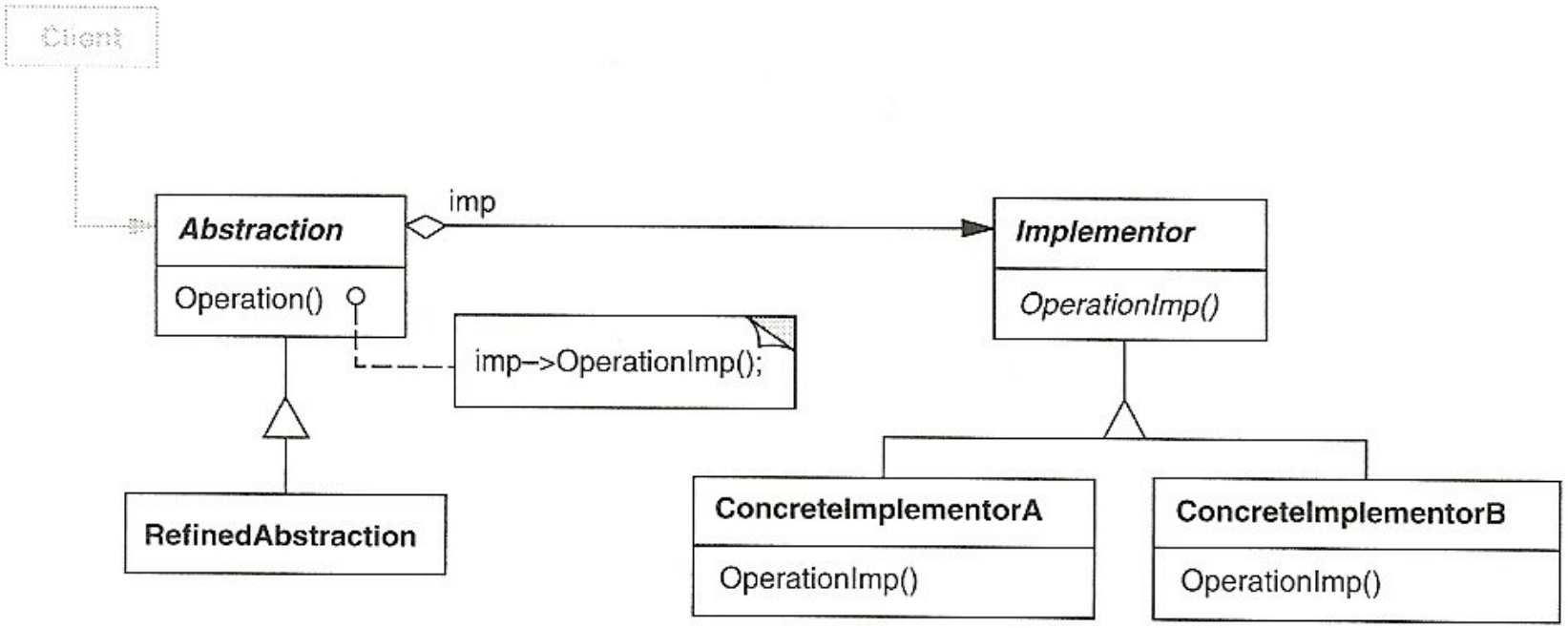
Proxy


Client
Subject
Request()
...
RealSubject
Request()
...
realSubject
Proxy
Request()
...
...
realSubject->Request();
...

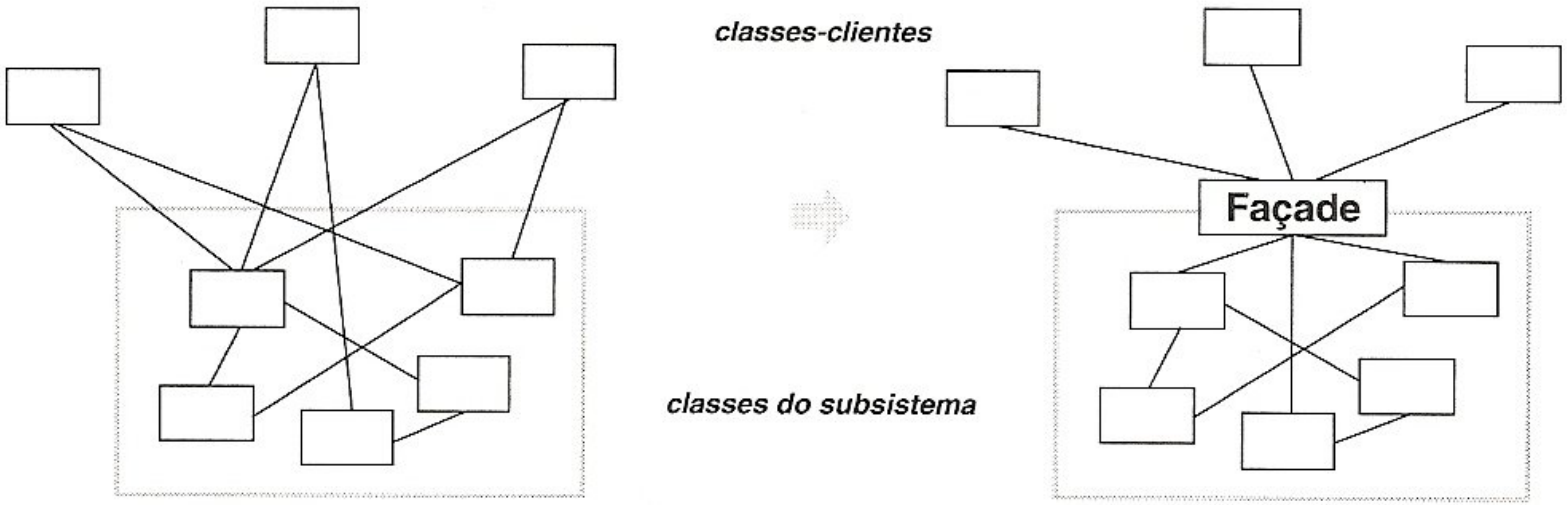
Bridge


Bridge
Window
DrawText()
DrawRect()
imp
WindowImp
DevDrawText()
DevDrawLine()
imp->DevDrawLine()
imp->DevDrawLine()
imp->DevDrawLine()
imp->DevDrawLine()
IconWindow
DrawBorder()
TransienWindow
DrawCloseBox()
XWindowImp
DevDrawText()
DevDrawLine()
PMWindowImp
DevDrawLine()
DevDrawText()
DrawRect()
DrawText()
DrawRect()
XDrawLine()
XDrawString()

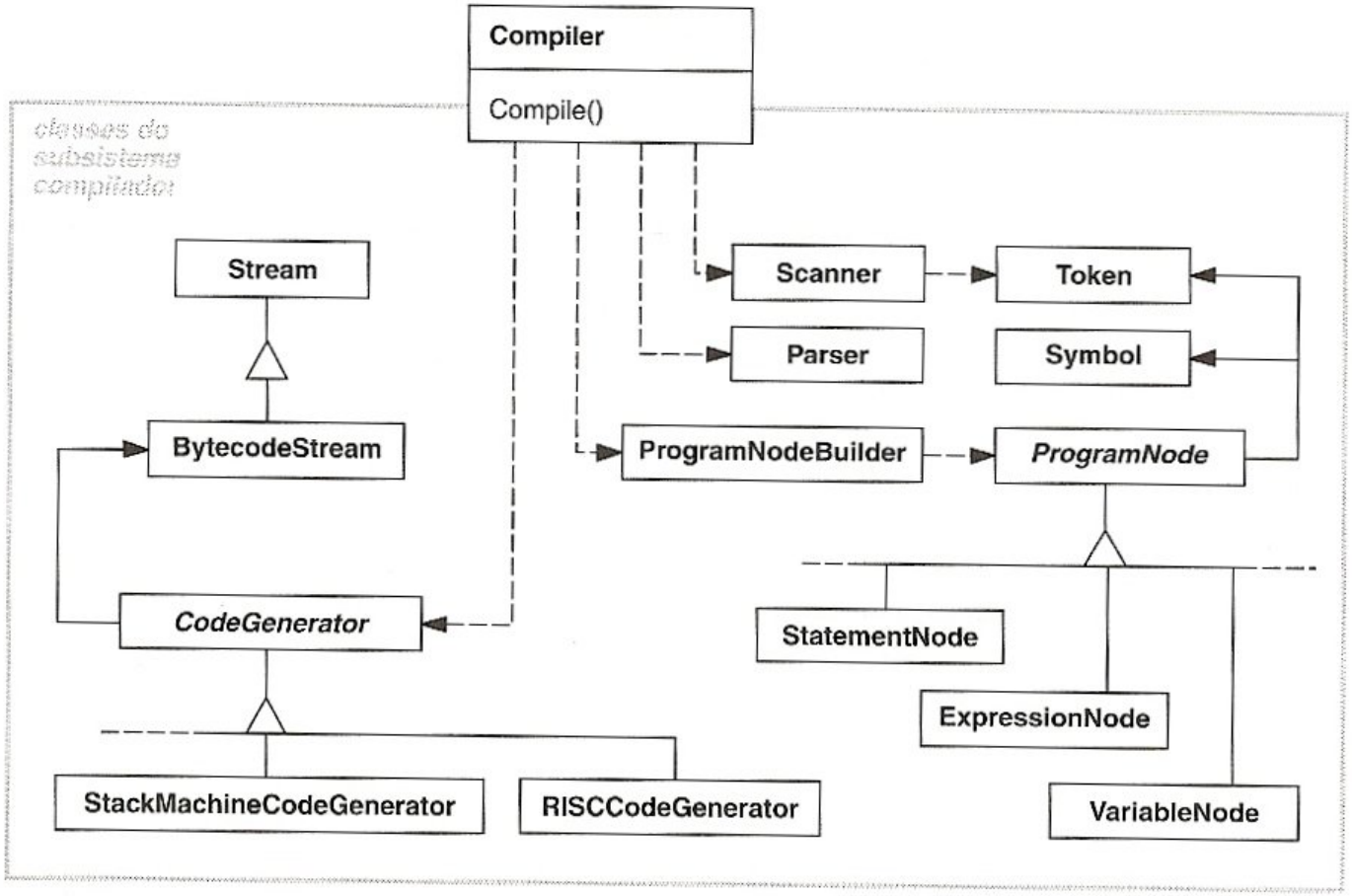
Bridge


Client
Abstraction
Operation()
imp
imp->OperationImp();
Implementor
OperationImp()
RefinedAbstraction
ConcreteImplementorA
OperationImp()
ConcreteImplementorB
OperationImp()

Facade



Facade


Compiler
Compile()
classes do subsistema compilador
Stream
Scanner
Token
Parser
Symbol
BytecodeStream
ProgramNodeBuilder
ProgramNode
CodeGenerator
StatementNode
ExpressionNode
StackMachineCodeGenerator
RISCCodeGenerator
VariableNode

Referências

- 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

- Site: http://www.vincehuston.org/dp/