Padrões de Software

GoF Patterns

Roni Fabio Banaszewski

UTFPR – Universidade Tecnológica Federal do Paraná

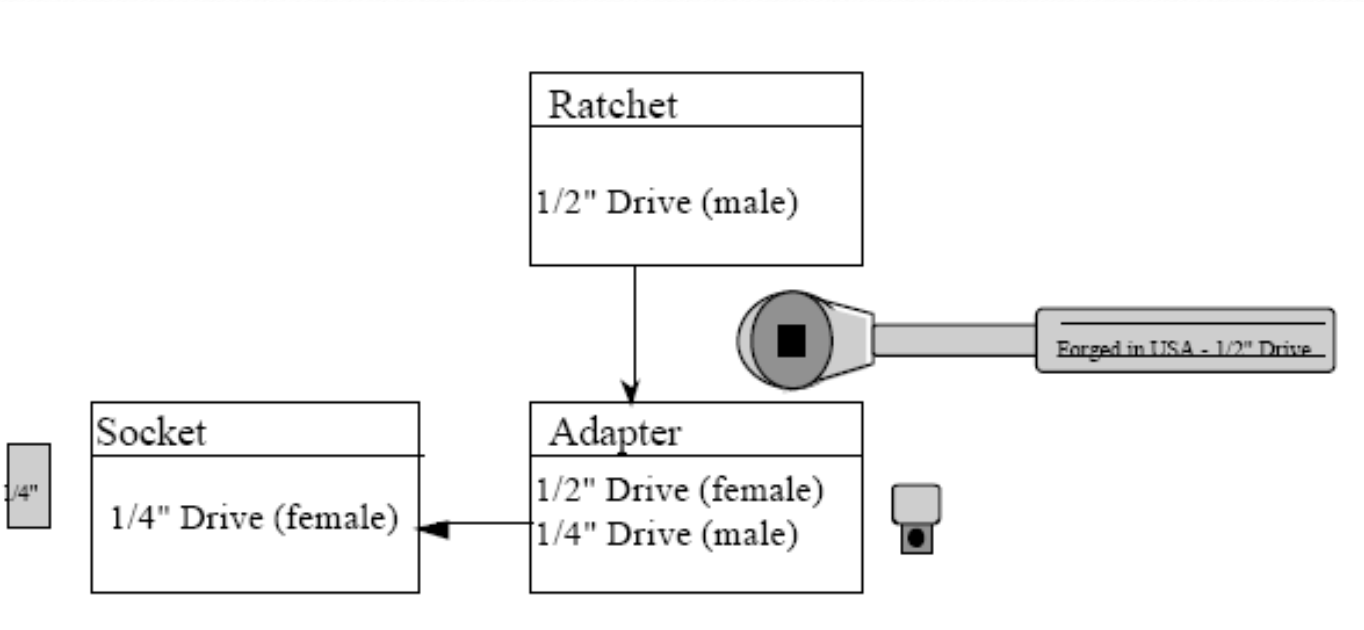
1

Adapter

"Objetivo:

Converter a interface de uma classe em outra interface esperada pelos clientes. Adapter permite a comunicação entre classes que não poderiam trabalhar juntas devido à incompatibilidade de suas interfaces." [GoF]

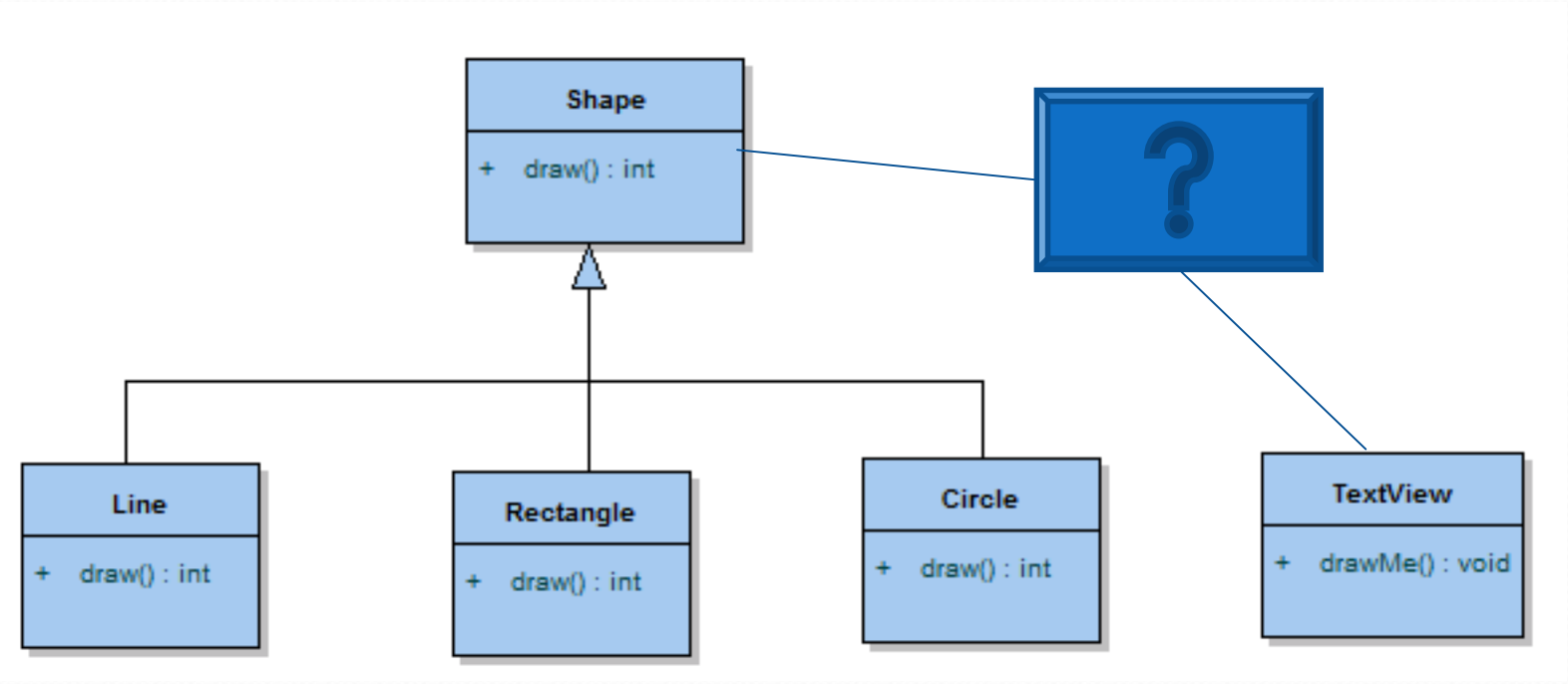
Analogia


Ratchet
1/2" Drive (male)
Forged in USA - 1/2" Drive
Socket
1/4" Drive (female)
Adapter
1/2" Drive (female)
1/4" Drive (male)
1/4"

Motivação

- Certos componentes reusáveis não são reutilizados com sucesso
  - Incompatibilidade de interfaces
- Por exemplo:
  - Um editor de desenho
    - Permite desenhar e organizar objetos gráficos
    - Objetos: Linhas, Polígonos, Caixas de Texto...
    - Cada objeto herda uma única interface: Shape
    - Linhas são fáceis de implementar
    - Caixa de Texto são mais complexas
      - Atualização de Tela e Gerenciamento de Buffer
    - Solução: Reutilizar um componente pronto: TextView

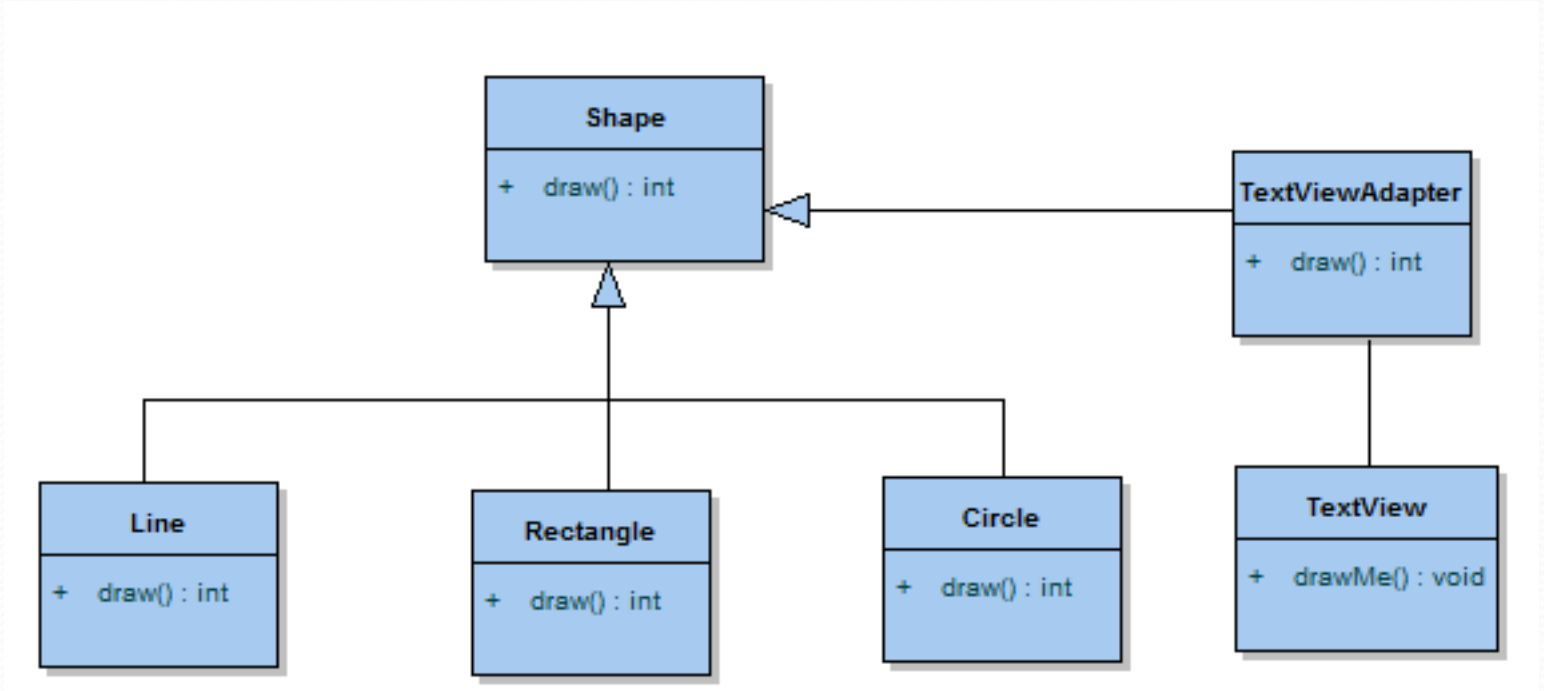
Motivação


Shape
+ draw() : int
?
Line
+ draw() : int
Rectangle
+ draw() : int
Circle
+ draw() : int
TextView
+ drawMe() : void

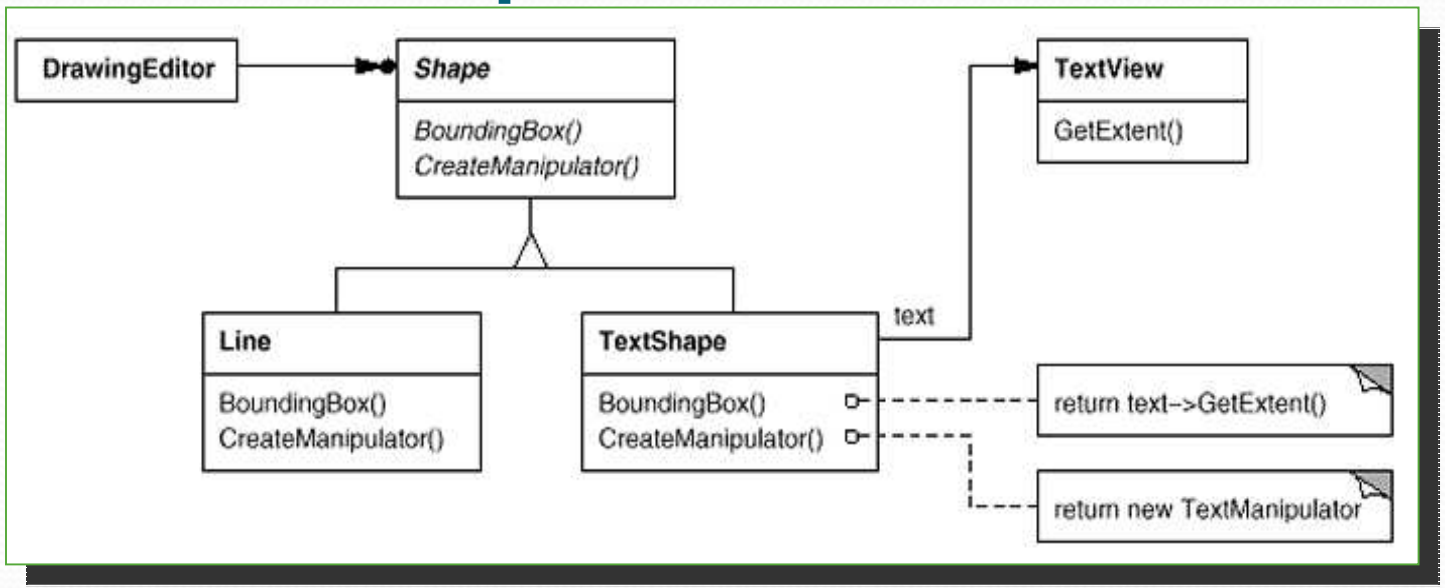
Motivação

- Problema:
  - O TextView não foi projetado para herdar a classe Shape
- O que fazer???????
- Como permitir que classes com interfaces incompatíveis possam se relacionar????
  - Alternativa: mexer na classe TextView e adaptá-la à aplicação
    - É preciso ter acesso ao código do TextView... Tem acesso?
    - Não. Então esqueça esta alternativa.
    - Sim. Cria um forte acoplamento com uma única aplicação.

Motivação

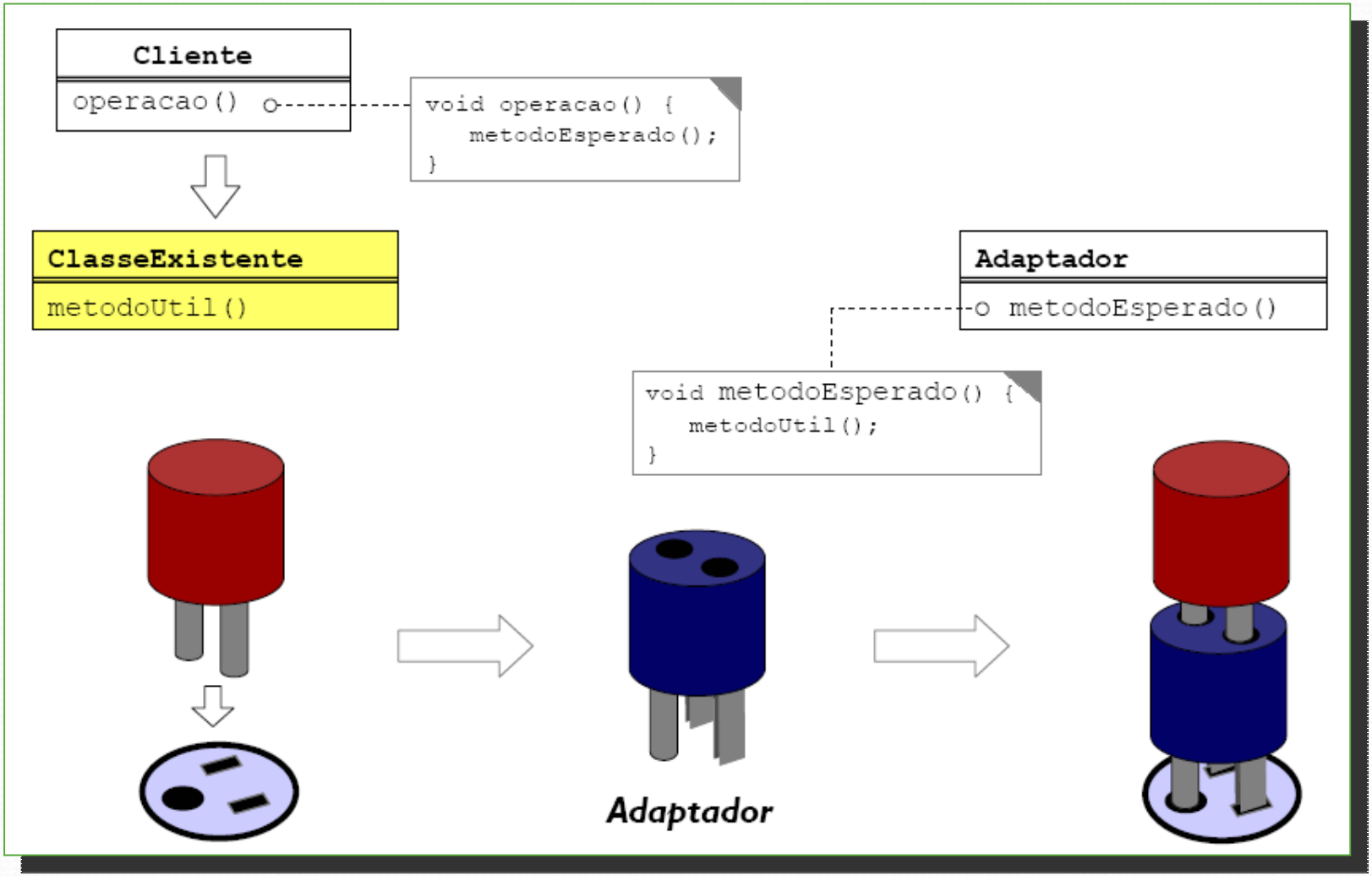

Shape
+ draw() : int
TextViewAdapter
+ draw() : int
Line
+ draw() : int
Rectangle
+ draw() : int
Circle
+ draw() : int
TextView
+ drawMe() : void

Solução - Adapter



- Define TextShape para adaptar a aplicação ao TextView
- Classes Adapter são responsáveis por funcionalidades que as classes adaptadas não provêem
  - Arrastar um objeto na tela
  - Manipulator implementa a animação dos objetos Shape: “drag”

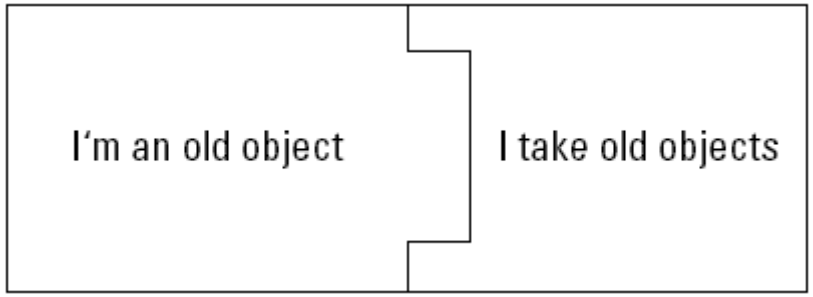
Solução - Adapter



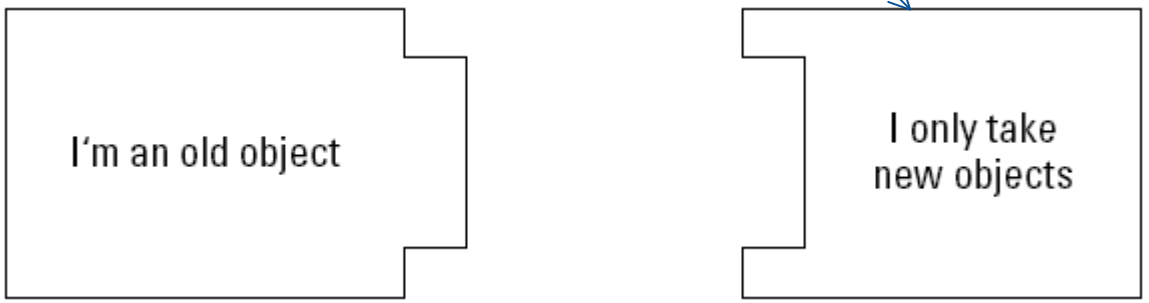
Aplicabilidade

- Você deseja reusar uma classe existente, mas a interface não é compatível com o que você necessita.
- Você quer criar uma classe reutilizável que coopera com classes inesperadas e não relacionadas, isto é, classes que não possuem interfaces compatíveis.
- Você precisa reutilizar muitas subclasses, mas é impraticável adaptar a sua interface para cada uma das subclasses. Um Object Adapter pode adaptar a interface da classe Base delas.

Exemplo: Adaptação de um objeto com interface desatualizada

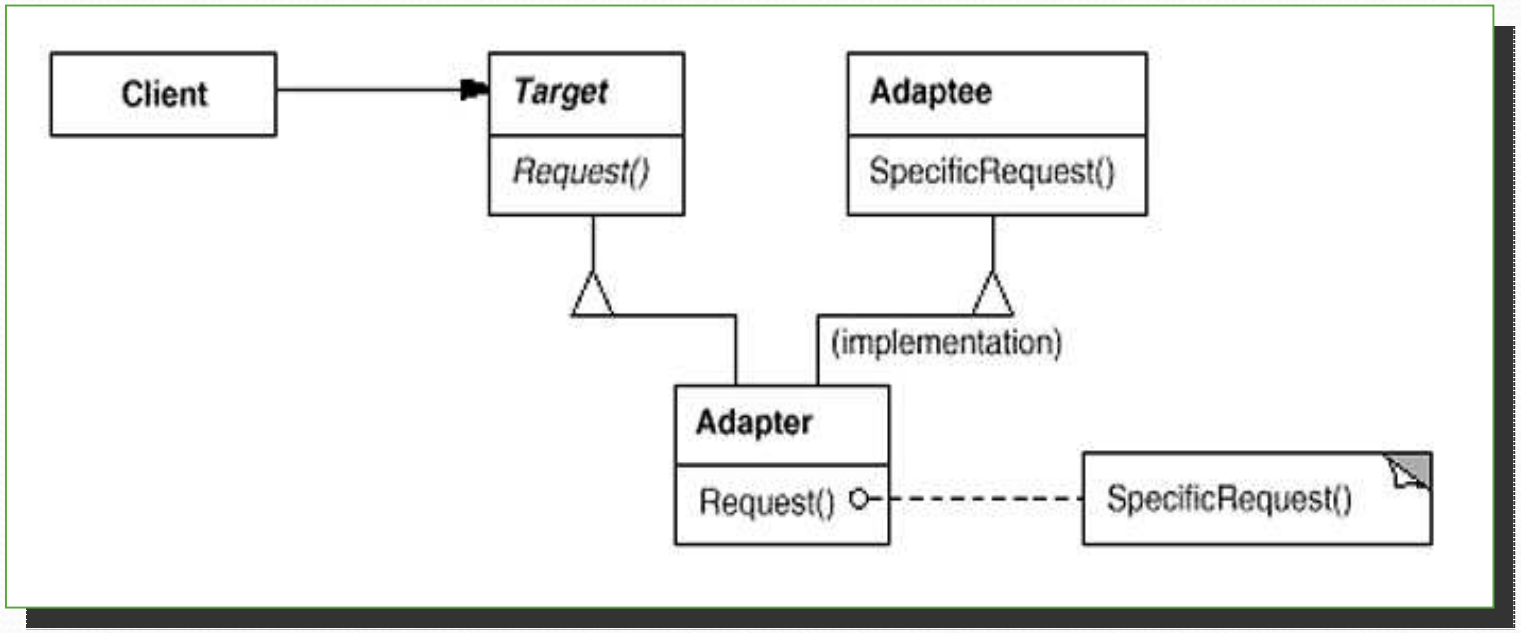


Foi atualizado

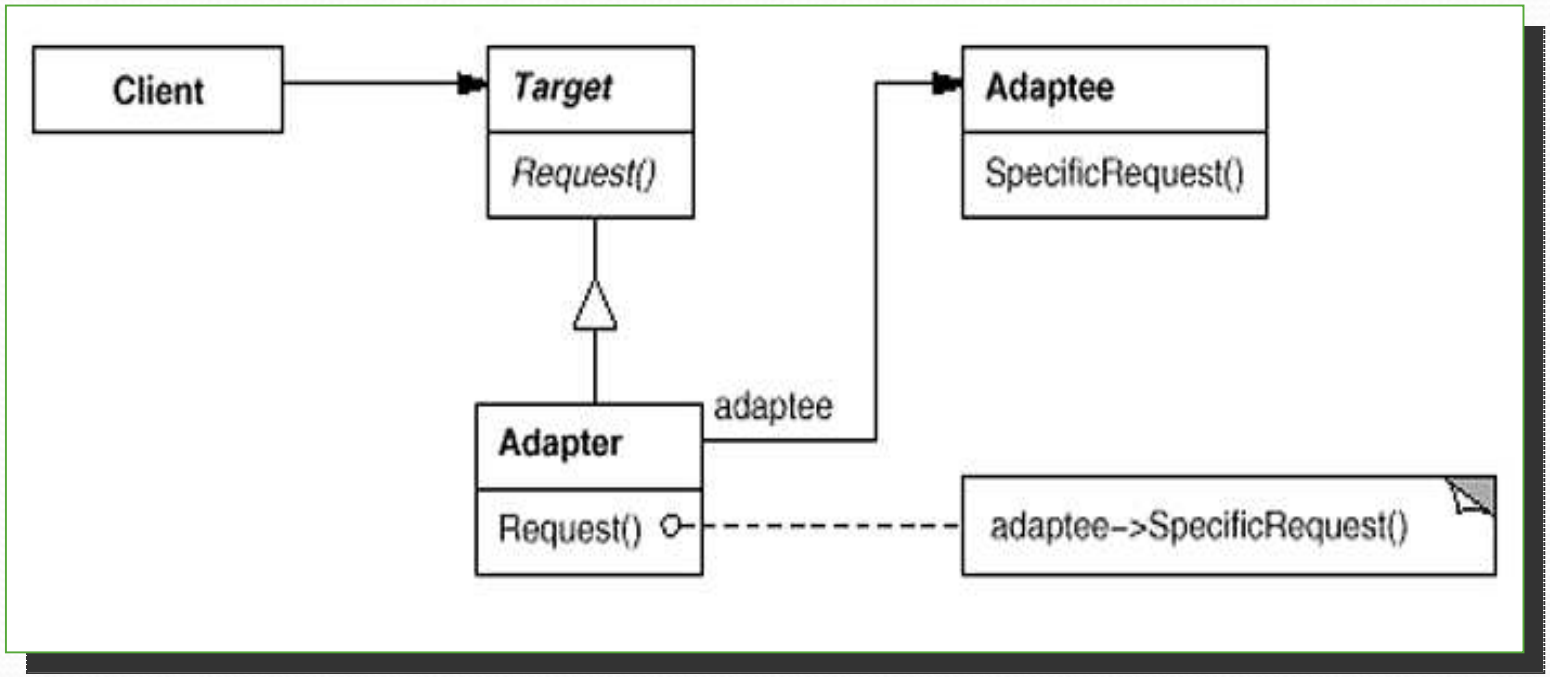


Solução

Class Adapter



Object Adapter



Participantes

- Target (Shape)
  - Define a interface que o cliente usa
- Client (Editor de Imagens)
  - Interage com os objetos de acordo a interface Target
- Adaptee (TextView)
  - Define uma interface existente que necessita ser adaptada
- Adapter (TextShape)
  - Adapta a interface de Adaptee para a interface Target

Conseqüências

- Class Adapter
  - Não funciona para adaptar uma classe Adaptee e algumas de suas subclasses – estende classe concreta
  - Permite ao Adapter subscrever algum comportamento de Adaptee – feito por herança
  - Introduz somente um objeto
- Object Adapter
  - Um Adapter pode trabalhar com muitos Adaptees
    - O Adaptee e todas suas subclasses
    - Torna difícil subscrever o comportamento do Adaptee
      - É preciso estender Adaptee, implementar uma subclasse e fazer Adapter apontar para esta subclasse

Implementação

- Classes Adapters em C++:
  - Adapter pode herdar publicamente a classe Target e privadamente a classe Adaptee
  - Assim, Adapter é subtipo de Target, mas não de Adaptee

Implementação – Código C++

```
class Shape {
public:
    Shape();
    virtual void BoundingBox(
        Point& bottomLeft, Point& topRight
    ) const;
    virtual Manipulator* CreateManipulator() const;
};

class TextView {
public:
    TextView();
    void GetOrigin(Coord& x, Coord& y) const;
    void GetExtent(Coord& width, Coord& height) const;
    virtual bool IsEmpty() const;
};
```

Implementação – Código C++

```
class TextShape : public Shape, private TextView {
public:
    TextShape();

    virtual void BoundingBox(
        Point& bottomLeft, Point& topRight
    ) const;
    virtual bool IsEmpty() const;
    virtual Manipulator* CreateManipulator() const;
};
```

```
bool TextShape::IsEmpty () const {
    return TextView::IsEmpty();
}
```

Class Adapter

Object Adapter

```
class TextShape : public Shape {
public:
    TextShape(TextView*);

    virtual void BoundingBox(
        Point& bottomLeft, Point& topRight
    ) const;
    virtual bool IsEmpty() const;
    virtual Manipulator* CreateManipulator() const;
private:
    TextView* _text;
};
```

```
bool TextShape::IsEmpty () const {
    return _text->IsEmpty();
}
```

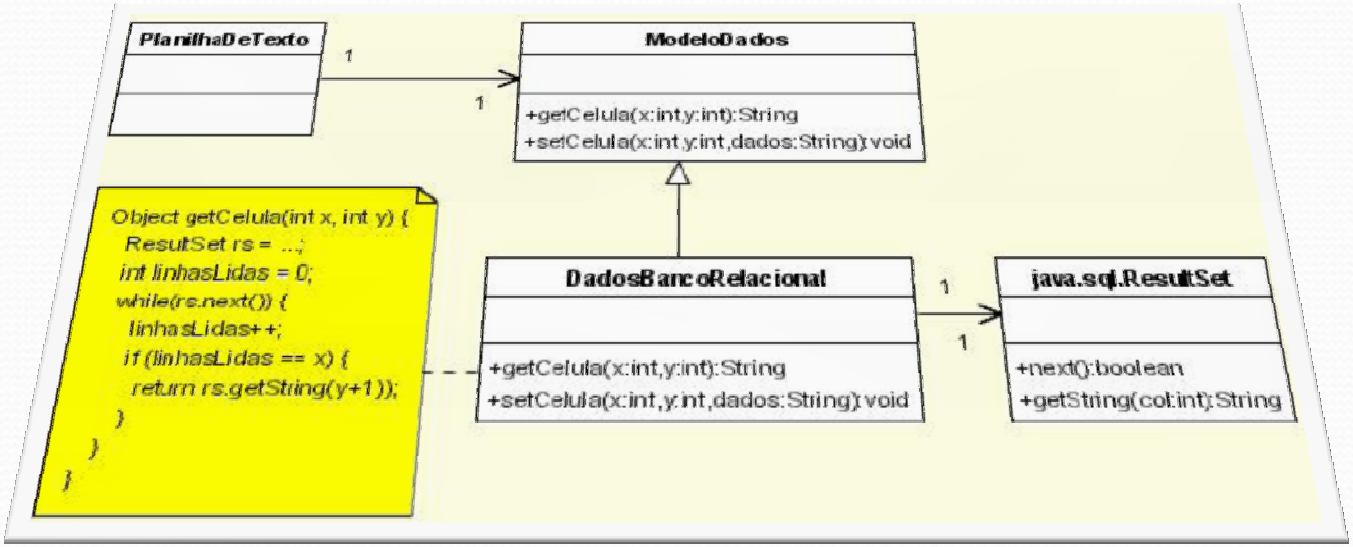
Classe x Objeto

```
void TextShape::BoundingBox (
    Point& bottomLeft, Point& topRight
) const {
    Coord bottom, left, width, height;

    GetOrigin(bottom, left);
    GetExtent(width, height);

    bottomLeft = Point(bottom, left);
    topRight = Point(bottom + height, left + width);
}
```

```
TextShape::TextShape (TextView* t) {
    _text = t;
}

void TextShape::BoundingBox (
    Point& bottomLeft, Point& topRight
) const {
    Coord bottom, left, width, height;

    _text->GetOrigin(bottom, left);
    _text->GetExtent(width, height);

    bottomLeft = Point(bottom, left);
    topRight = Point(bottom + height, left + width);
}

bool TextShape::IsEmpty () const {
    return _text->IsEmpty();
}
```

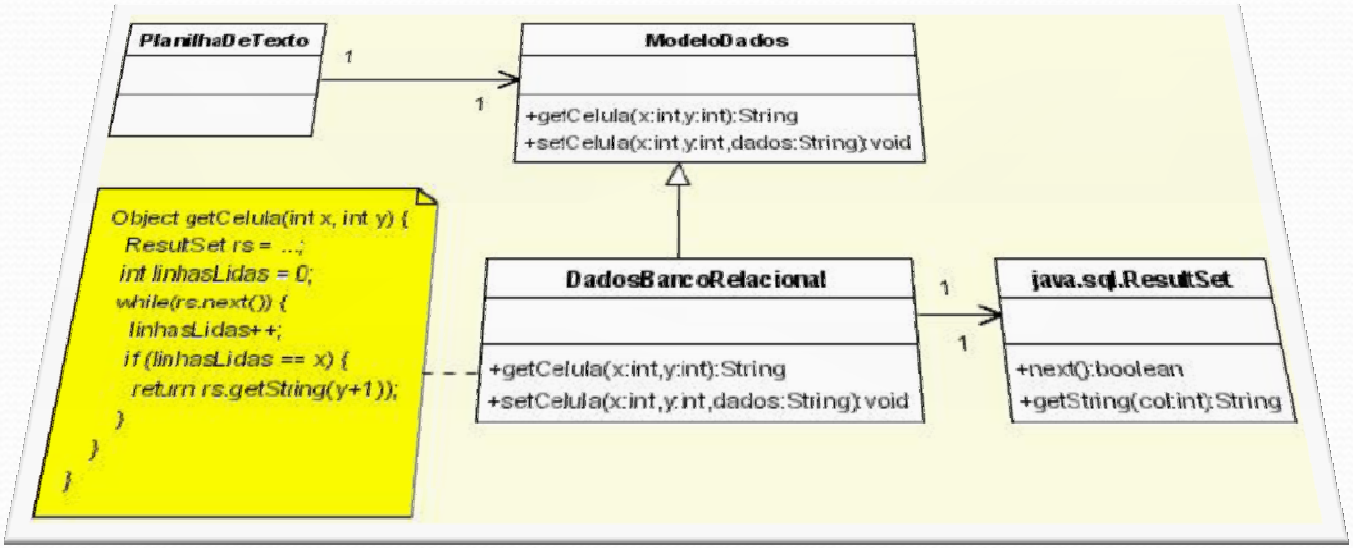
Que padrão de design é representado pela classe DadosBancoRelacional no diagrama UML abaixo?

a) Proxy
b) Decorator
c) Adapter
d) Composite
e) Façade

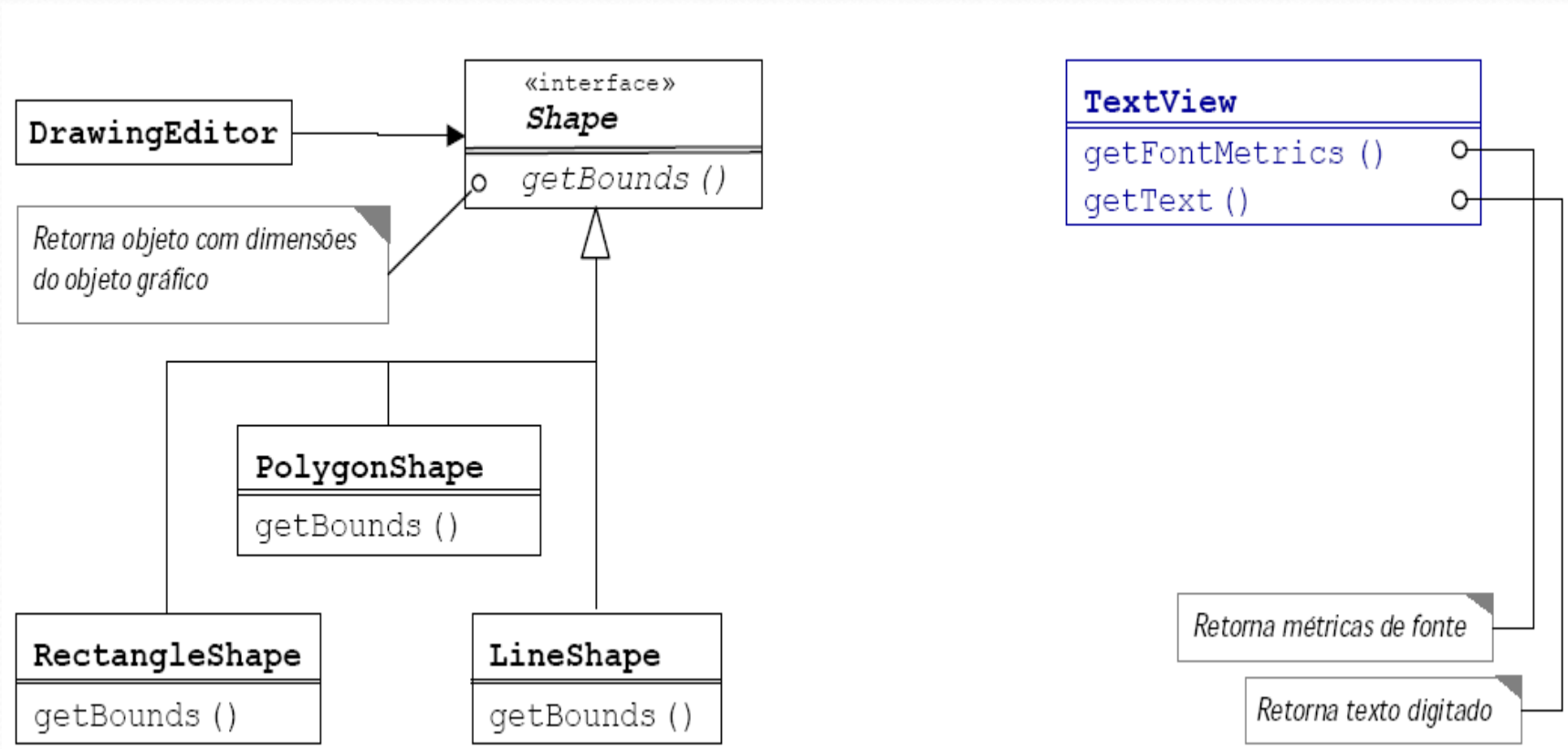


Que padrão de design é representado pela classe DadosBancoRelacional no diagrama UML abaixo?

a) Proxy
b) Decorator
c) Adapter
d) Composite
e) Façade



Complete o diagrama de classes abaixo para que um objeto TextView possa participar da aplicação DrawingEditor



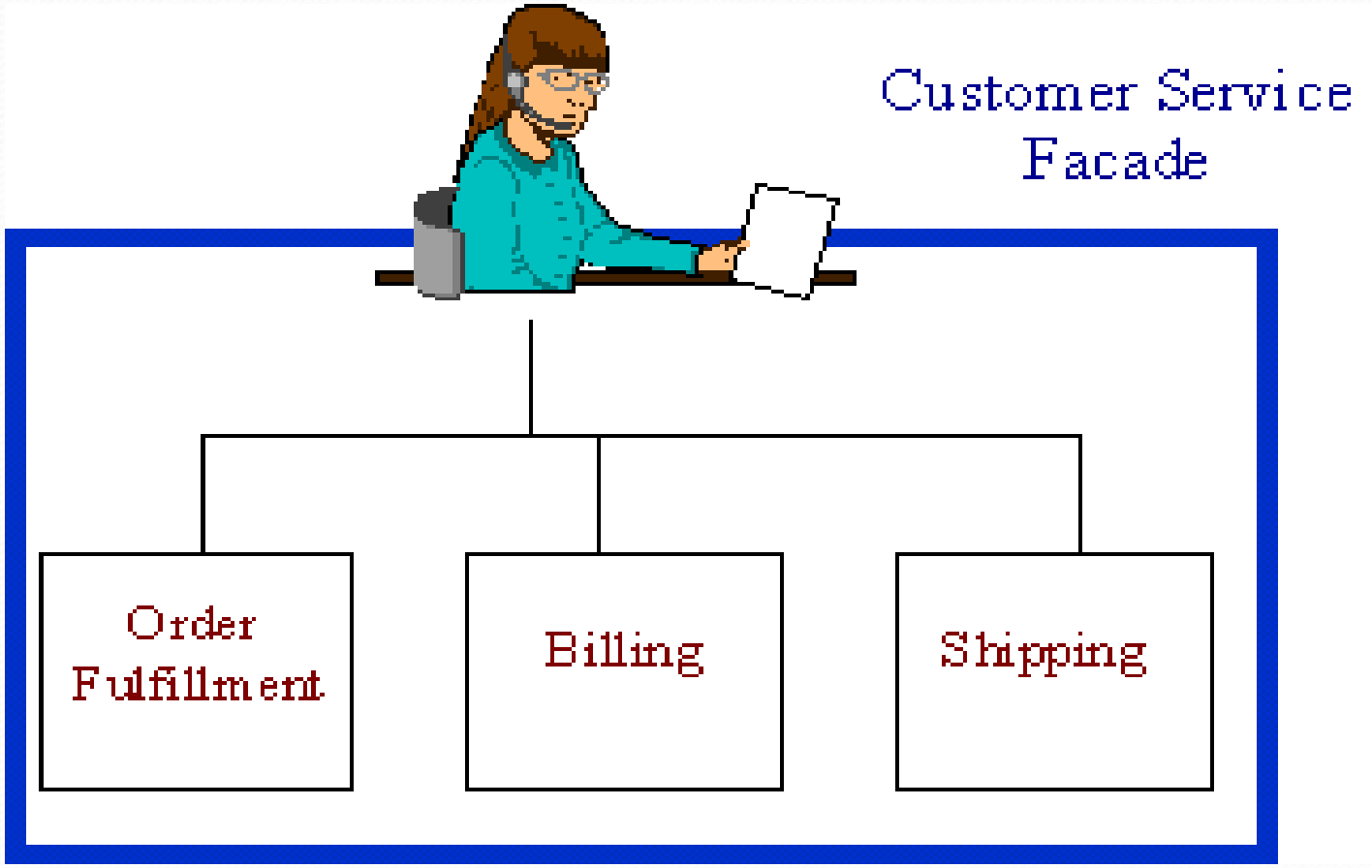
2

Facade

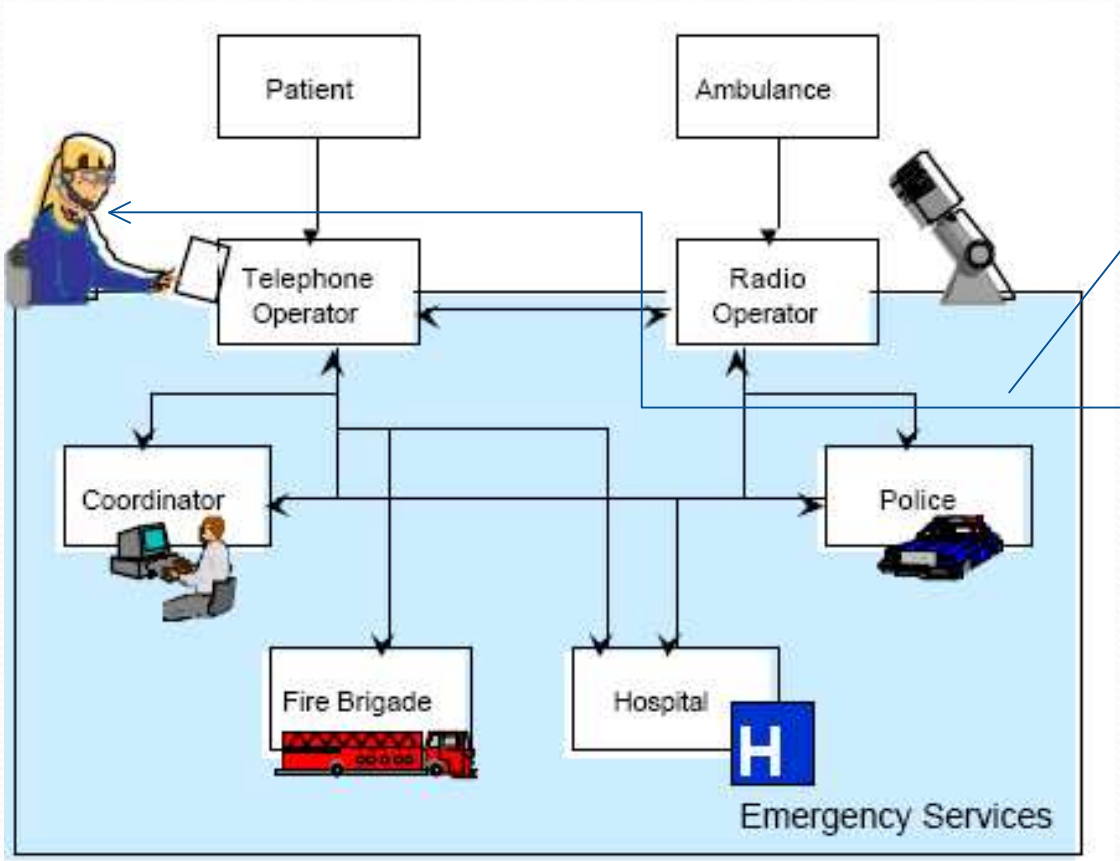
Objetivo:

"Oferecer uma interface única para um conjunto de interfaces de um subsistema. Façade define uma interface de nível mais elevado que torna o subsistema mais fácil de usar." [GoF]

Analogia


Customer Service
Facade
Order
Fulfillment
Billing
Shipping

Analogia

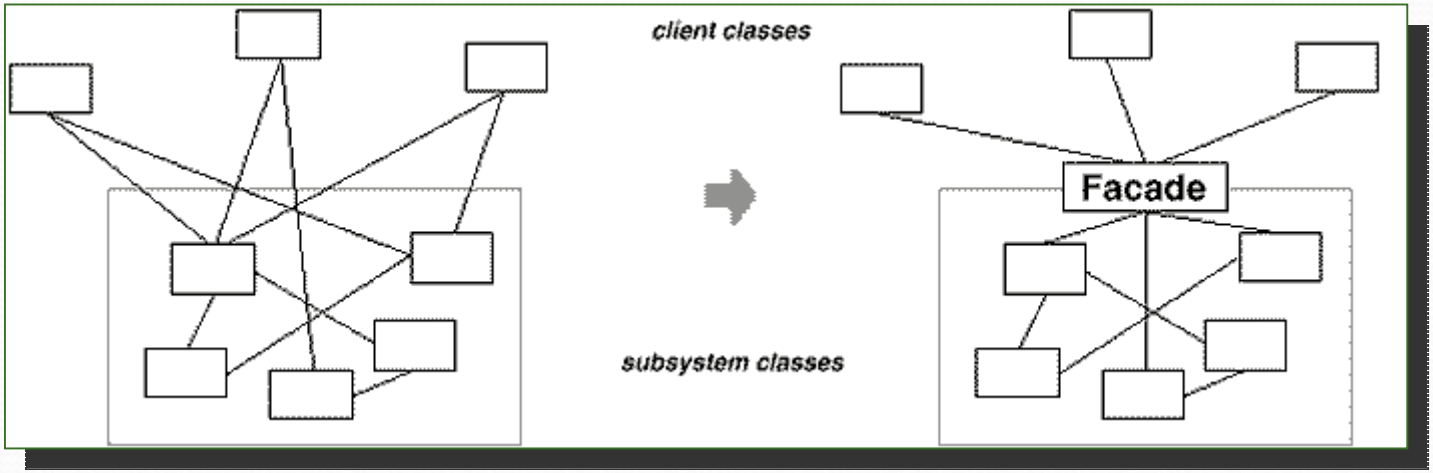


Vários tipos de serviços

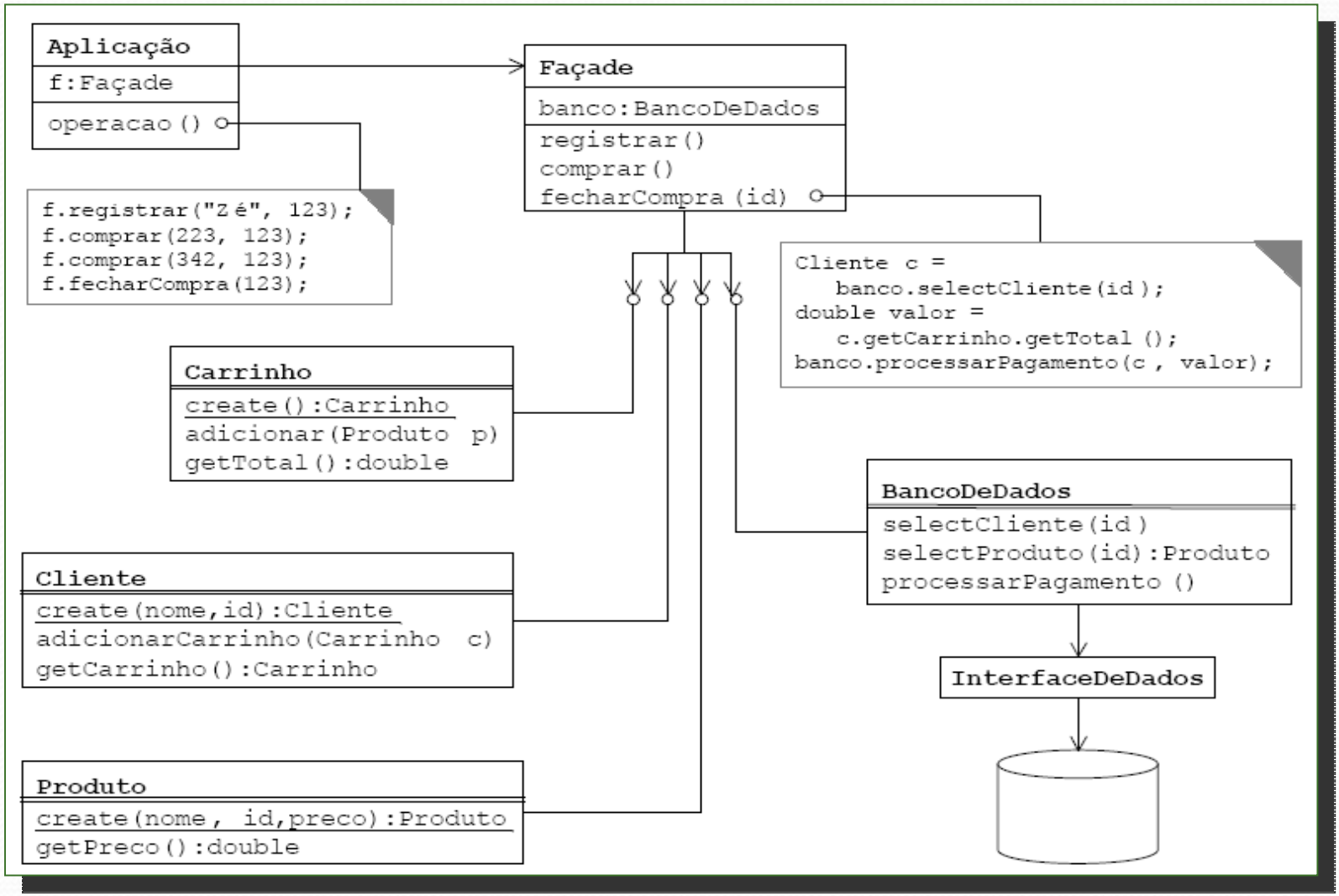
Não é preciso ligar diretamente para cada serviço (Polícia, bombeiros...). Basta pedir ajuda ao serviço de emergência da sua localidade comforme o acontecido (Roubo)

Motivação

- Estruturar um sistema em vários subsistemas ajuda a reduzir a complexidade
- É preciso diminuir a comunicação e dependência entre subsistemas
- Solução: Facade
  - Interface única
  - Simplifica a interface para interagir com vários tipos de serviços



Exemplo

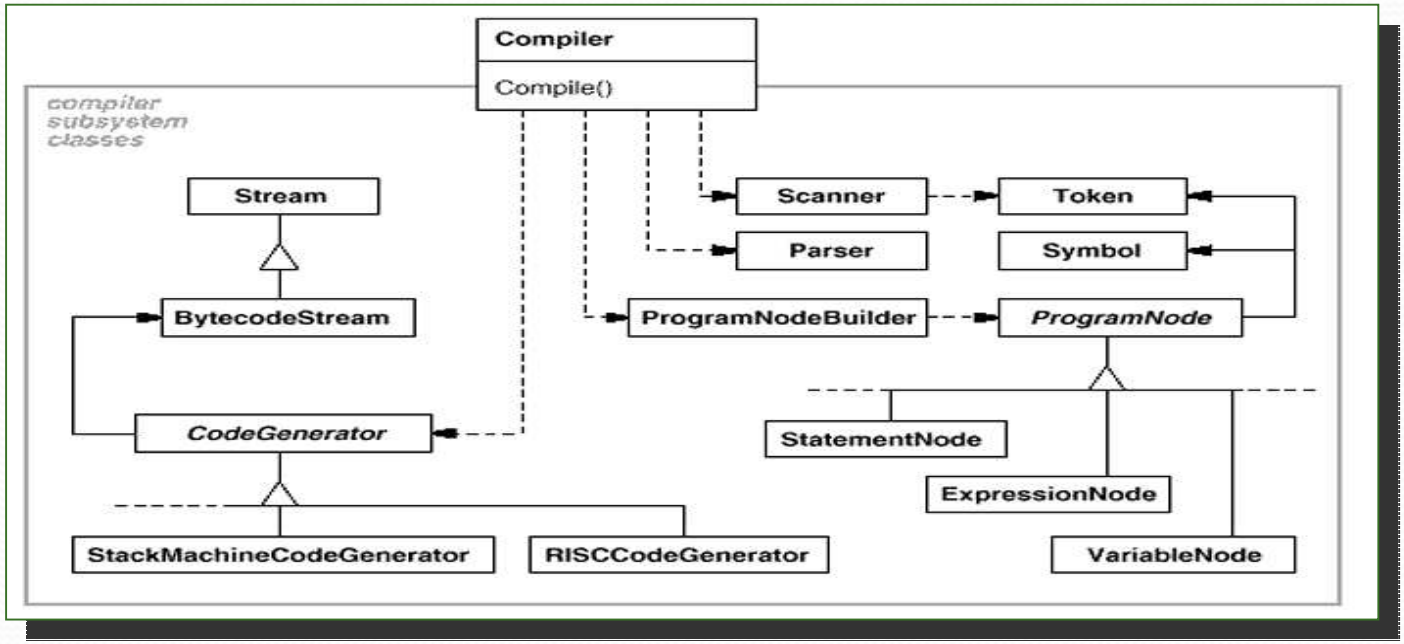

Aplicação
f:Façade
operacao()
f.registrar("Zé", 123);
f.comprar(223, 123);
f.comprar(342, 123);
f.fecharCompra(123);
Façade
banco:BancoDeDados
registrar()
comprar()
fecharCompra(id)
Cliente c =
banco.selectCliente(id);
double valor =
c.getCarrinho.getTotal();
banco.processarPagamento(c , valor);
Carrinho
create():Carrinho
adicionar(Produto p)
getTotal():double
BancoDeDados
selectCliente(id)
selectProduto(id):Produto
processarPagamento()
Cliente
create(nome,id):Cliente
adicionarCarrinho(Carrinho c)
getCarrinho():Carrinho
InterfaceDeDados
Produto
create(nome, id,preco):Produto
getPreco():double

Exemplo

- Um ambiente de programação que permite acesso ao subsistema de compilação
  - Classes do Subsistema: Scanner, Parser...
- Algumas aplicações necessitam interagir diretamente com partes destes objetos
- A maioria das aplicações, somente precisam compilar algum código de forma genérica
  - Precisam de uma interface simples
  - Não querem se aborrecer com os detalhes de compilação
  - Neste caso, o poder das interfaces de baixo nível somente complicam suas tarefas

Solução - Facade

- Classe Compiler = Facade
  - Interface de alto nível
  - Interface única
  - Ele relaciona as classes necessárias para compilação sem esconder as interfaces de baixo nível
    - Elas podem continuar sendo utilizadas

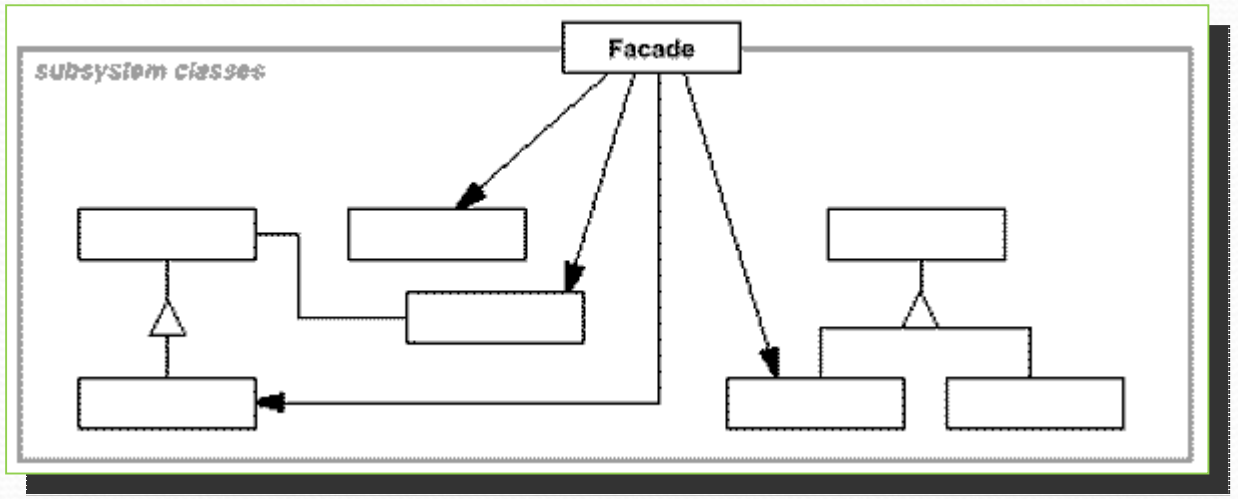


Aplicabilidade

Quando:

- Precisar prover uma interface simples para um subsistema complexo
  - Clientes que necessitam funcionalidades mais avançadas poderão ignorar o Facade
- Há muita dependência entre o código cliente e um subsistema
  - Promove independência do subsistema e portabilidade
- Se deseja uma arquitetura em camadas
  - Faca de como ponto de entrada para cada nível
  - Subsistemas dependentes podem se relacionar através de seus Facades

Estrutura e Participantes



- Facade:
  - Conhece quais classes do subsistemas são responsáveis pela requisição
  - Delega as requisições de clientes para os apropriados objetos de um subsistema
- Classes do Subsistema
  - Implementam as funcionalidades do subsistema
  - Não possuem conhecimento sobre o Facade

Conseqüências

- Torna o subsistema mais fácil de usar
  - Protege o cliente dos detalhes do subsistema
- Fraco acoplamento entre Cliente e Subsistema
  - Pode mudar os componentes no subsistema sem afetar o cliente
  - Ex.: o departamento de envio pode mudar de responsáveis sem impacto no relacionamento do cliente com a companhia
- Não impede, quando necessário, as aplicações fazerem uso das classes do subsistema

Implementação

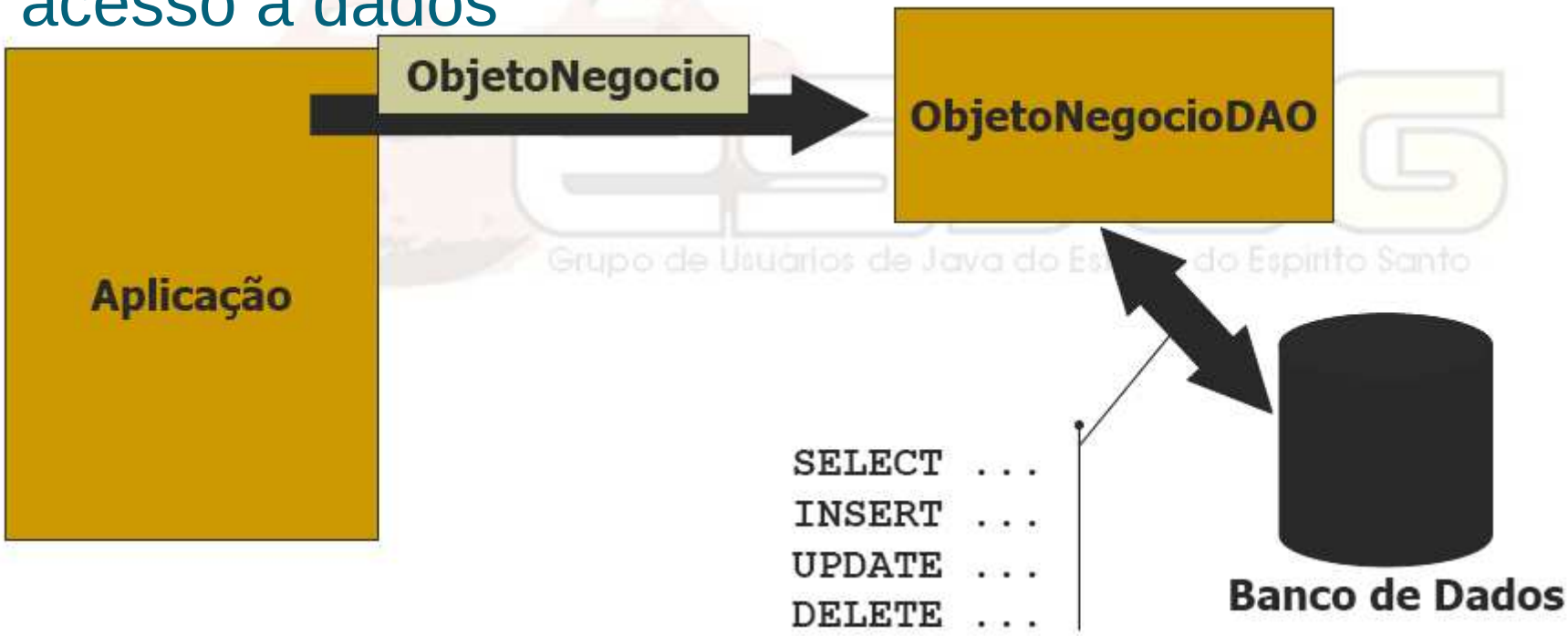
```
class Compiler {
public:
    Compiler();

    virtual void Compile(istream&, BytecodeStream&);
};

void Compiler::Compile (
    istream& input, BytecodeStream& output
) {
    Scanner scanner(input);
    ProgramNodeBuilder builder;
    Parser parser;

    parser.Parse(scanner, builder);

    RISCCodeGenerator generator(output);
    ProgramNode* parseTree = builder.GetRootNode();
    parseTree->Traverse(generator);
}
```

```
class Scanner {
public:
    Scanner(istream&);
    virtual ~Scanner();

    virtual Token& Scan();
private:
    istream& _inputStream;
};
```

```
class Parser {
public:
    Parser();
    virtual ~Parser();

    virtual void Parse(Scanner&, ProgramNodeBuilder&);
};
```

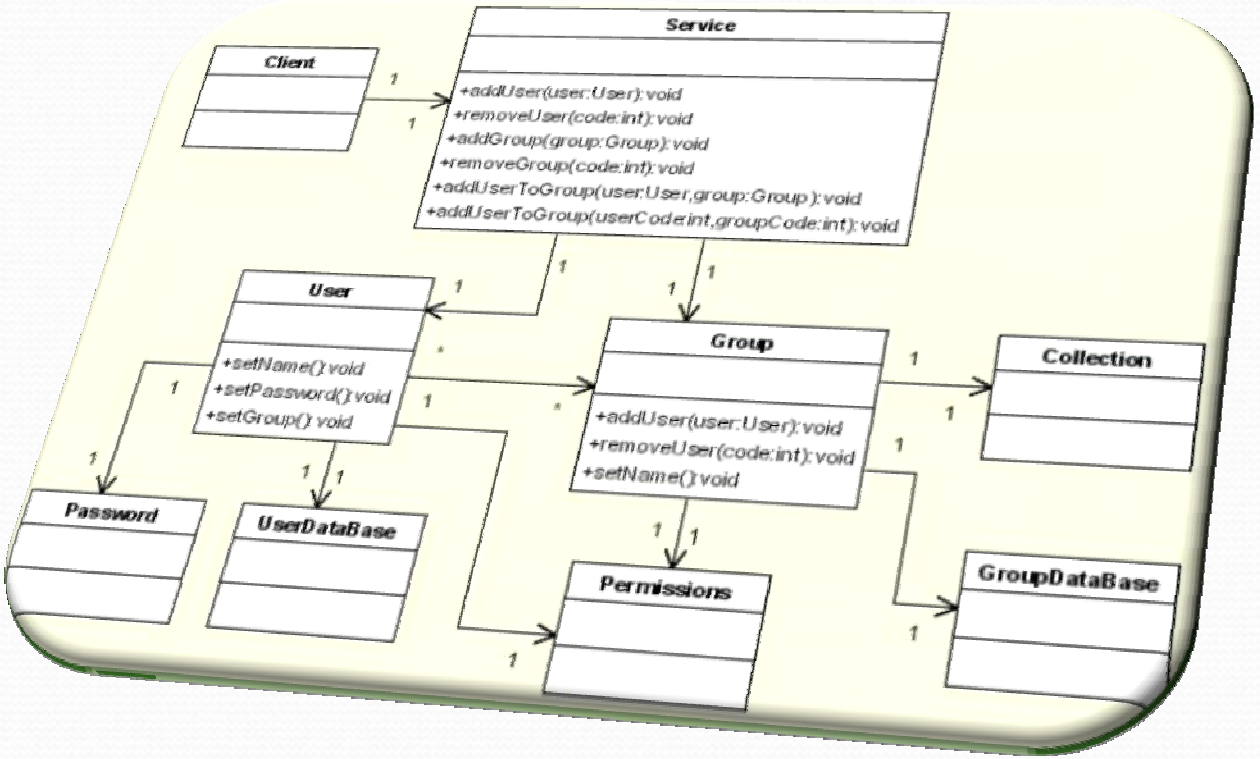
Data Acess Object (DAO)

O padrão DAO pode ser considerado uma fachada para acesso a dados



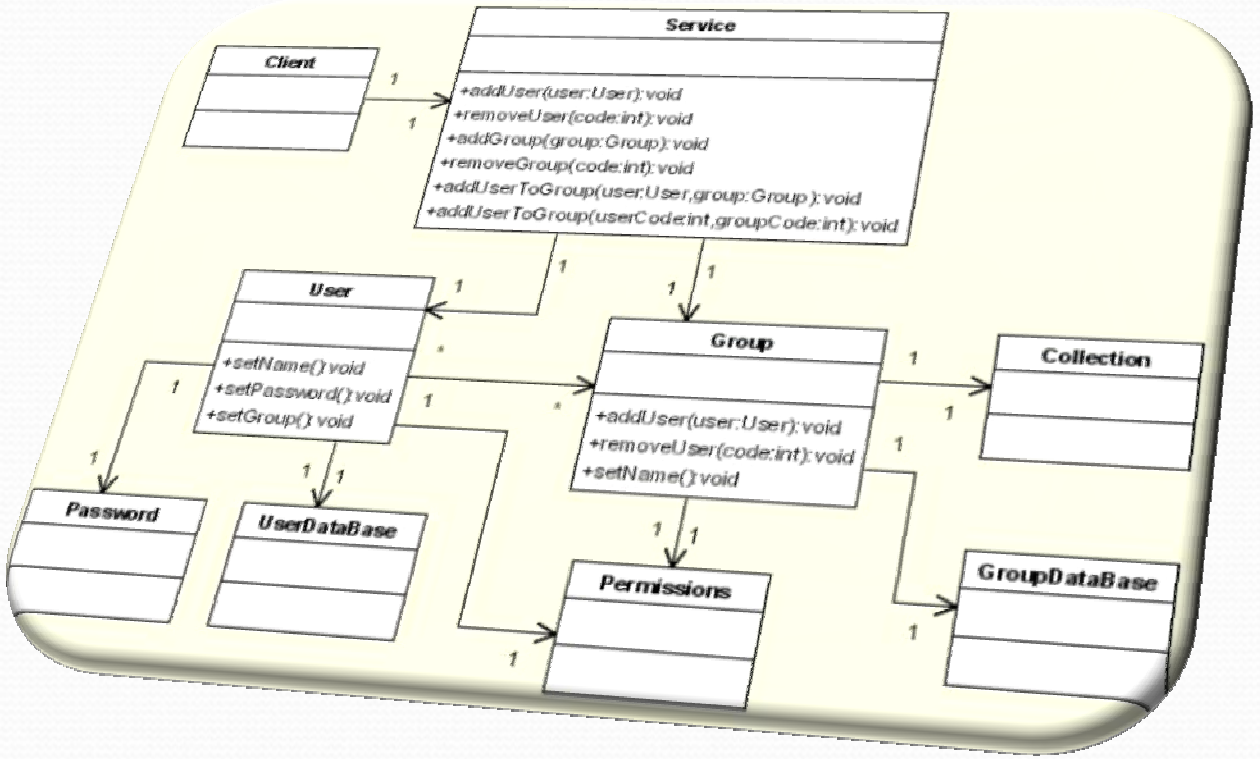
A classe Service abaixo concentra todas as operações que podem ser executadas pela classe Client, simplificando a interface da aplicação. Que padrão de design é representado por esta classe?

a) Adapter
b) Command
c) Strategy
d) Singleton
e) Façade



A classe Service abaixo concentra todas as operações que podem ser executadas pela classe Client, simplificando a interface da aplicação. Que padrão de design é representado por esta classe?

a) Adapter
b) Command
c) Strategy
d) Singleton
e) Façade

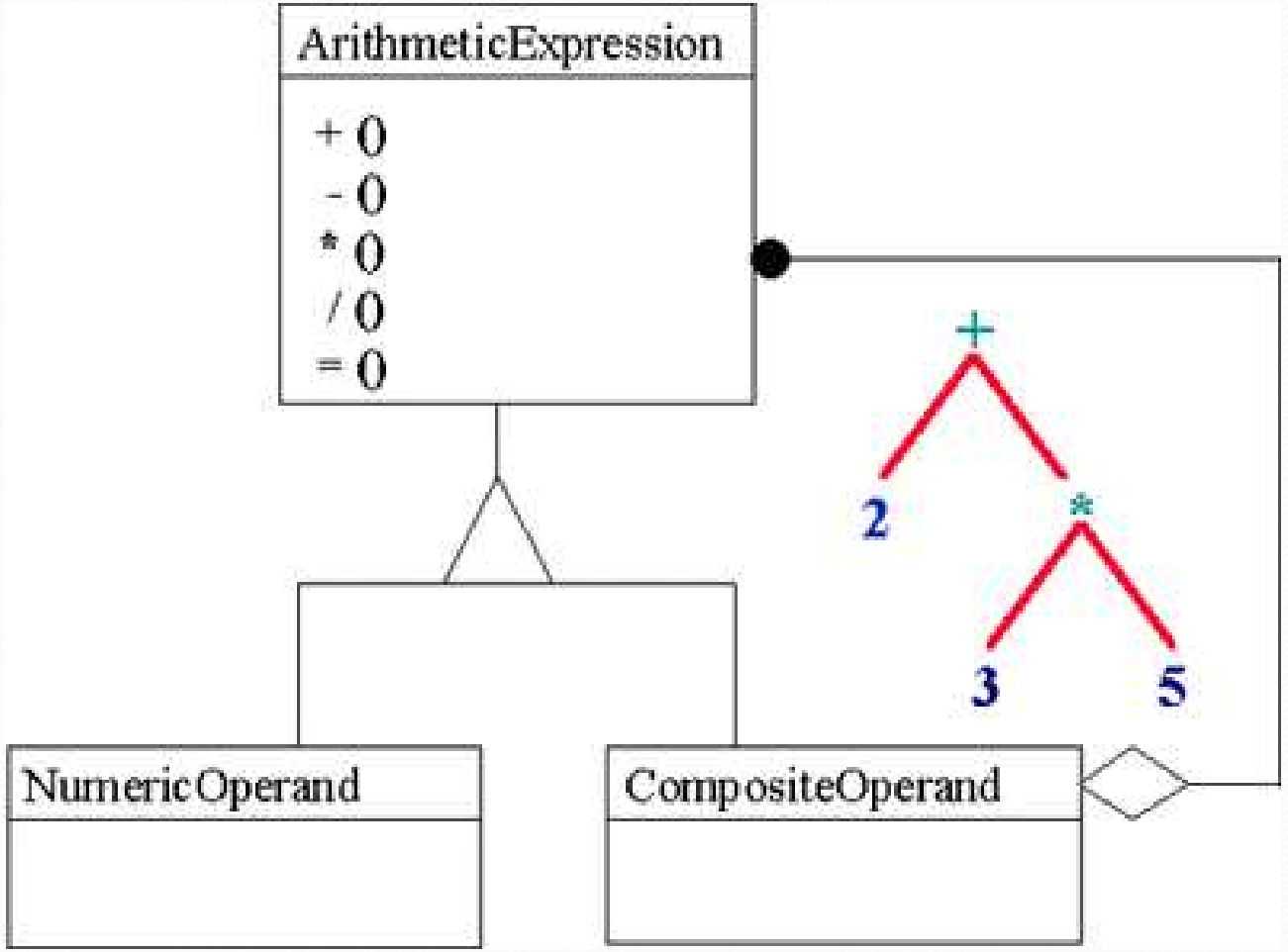


Composite

Objetivo:

"Compor objetos em estruturas de árvore para representar hierarquias todo-parte. Composite permite que clientes tratem objetos individuais e composições de objetos de maneira uniforme." [GoF]

Analogia


ArithmeticExpression
+ 0
- 0
* 0
/ 0
= 0
+
2
*
3
5
NumericOperand
CompositeOperand

Analogia

White Roux

1 tablespoon flour
1 tablespoon butter

To make roux, melt the butter in a saucepan over medium heat. When the butter starts to froth, stir in flour, combining well.

Continue cooking the roux over heat until it turns a pale brown and has a nutty fragrance.

Pasley Sauce

1 portion of white roux
1 cup milk
2 tablespoons of parsley

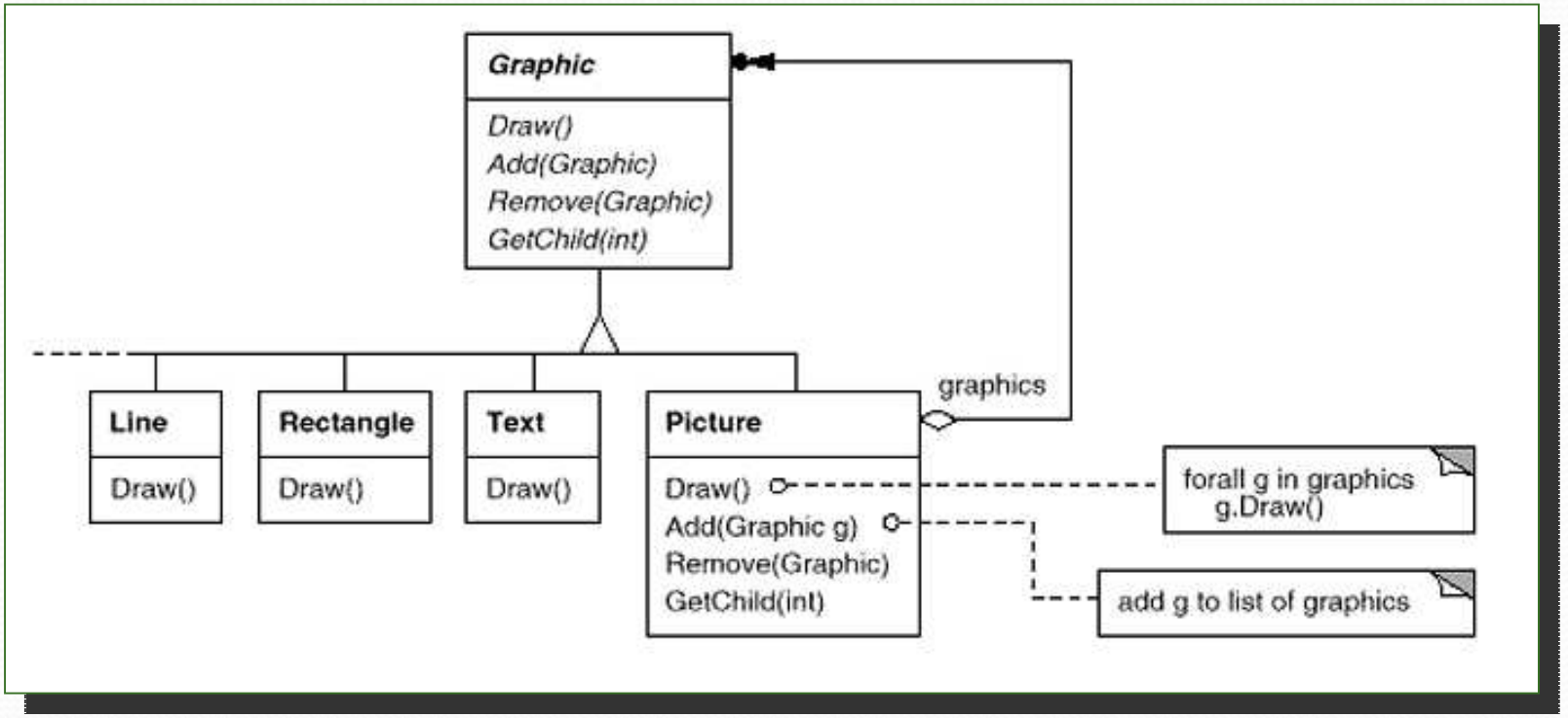
To make sauce, over heat, add a little of the milk to the roux, stirring until combined. Continue adding milk slowly until you have a smooth liquid. Stir over medium heat for 2 minutes. Then stir in parsley, and cook for another 30 seconds. Remove from heat, and leave sauce standing to thicken.

Motivação

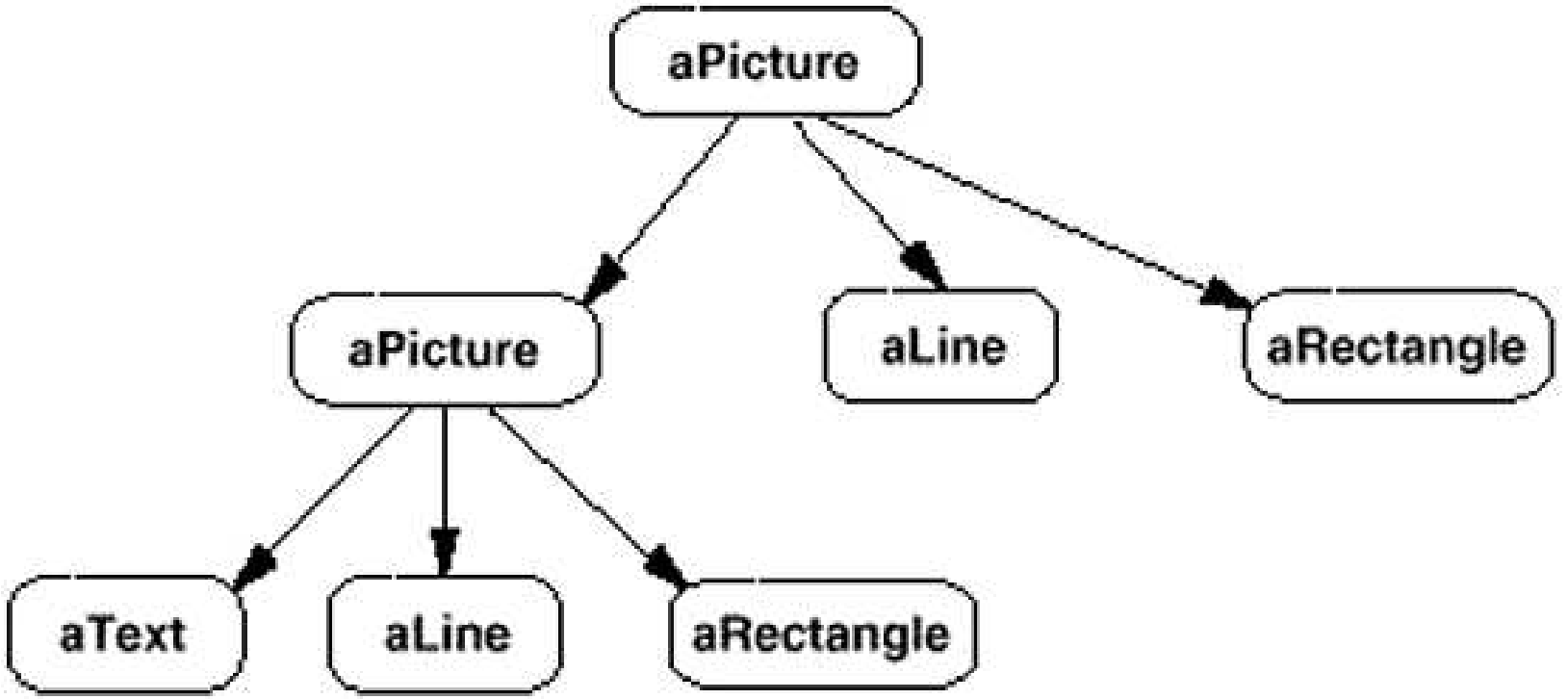
- Editores de desenho utilizam simples componentes para formar componentes complexos
  - Agrupamento de componentes primitivos
- Componentes primitivos: Linha, Retângulo, Texto...
- Componentes complexos: Figura
- Problema:
  - O código da aplicação deve tratar os dois tipos de componentes de forma diferente
  - O usuário da aplicação deve trabalhar com os dois de forma idêntica
  - Diferenciar estes componentes torna a aplicação mais complexa

Solução - Composite

- Com Composite – a distinção entre os componentes não é mais necessário



Estrutura de um objeto Composite

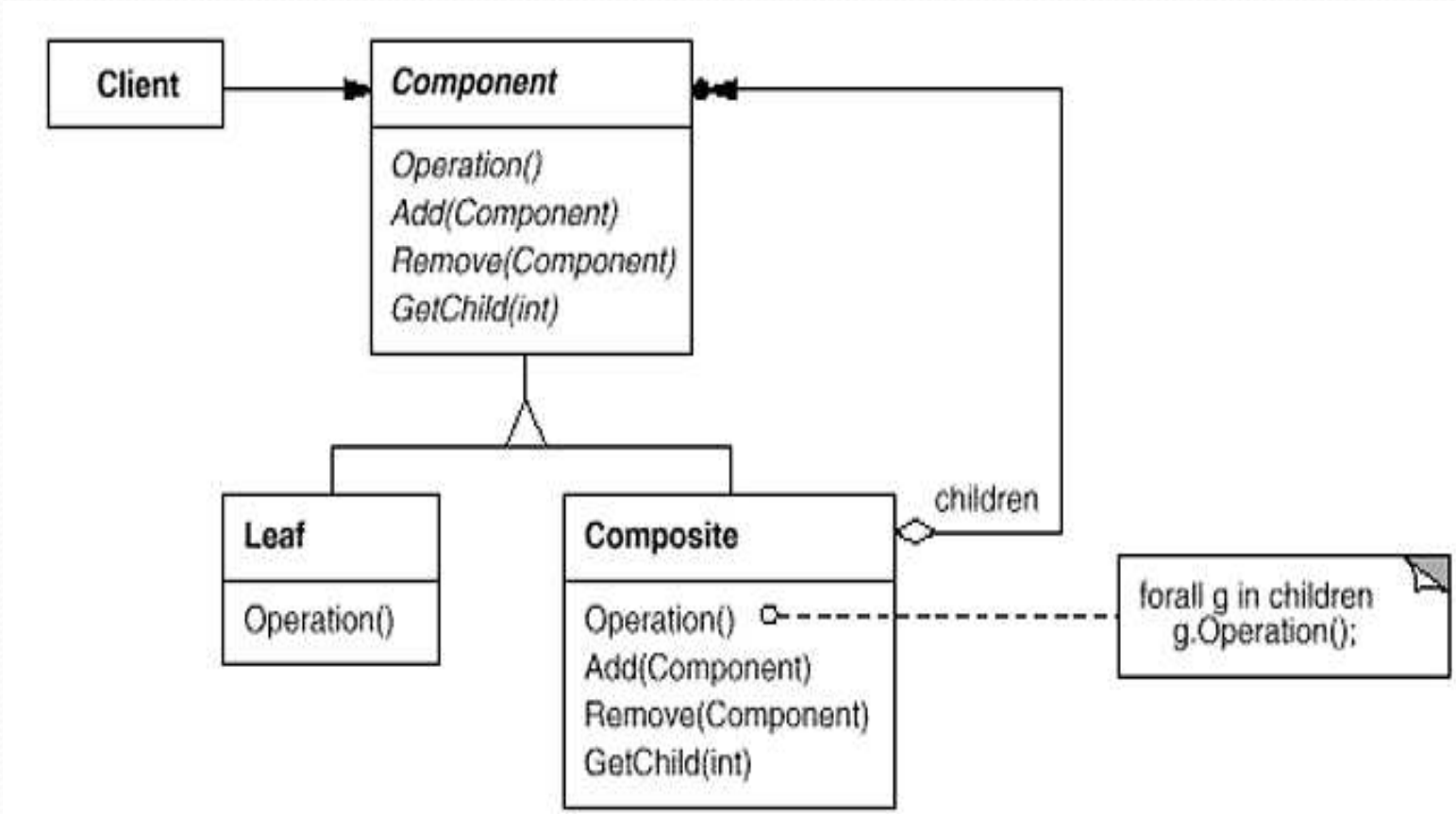


Aplicabilidade

Usar quando:

- Deseja-se representar uma hierarquia de objetos Todo-Parte
- Deseja-se que clientes ignorem a diferença entre componentes individuais e composições

Estrutura


Client
Component
Operation()
Add(Component)
Remove(Component)
GetChild(int)
Leaf
Operation()
Composite
Operation()
Add(Component)
Remove(Component)
GetChild(int)
children
forall g in children
g.Operation();

Participantes

- Component (Graphic)
  - Declara a interface para objetos na composição
  - Implementa comportamentos default para a interface comum a todas as classes
  - Declara uma interface para acesso e gerenciamento dos nós folhas da composição
- Leaf (Linha, Texto...)
  - Não possui nó filho
  - Define o comportamento para os componentes primitivos
- Composite (Figura)
  - Possui filhos
- Cliente
  - Manipula os objetos na composição através da interface Component

Conseqüências

- Hierarquia de objetos primitivos e compostos
- Torna o código cliente mais simples
  - Trata ambos de forma uniforme
  - Não é necessário escrever códigos de seleção para determinar qual o tipo de componente
- Facilidade de adicionar novos tipos de componentes
  - Cliente não é modificado

Exemplo

- O hardware de computadores são organizados como Todo-Parte
  - Um gabinete pode conter cabos, luzes, auto-falantes...
  - Uma placa-mãe contém circuitos, processador....

```
class Equipment {
public:
    virtual ~Equipment();

    const char* Name() { return _name; }

    virtual Watt Power();
    virtual Currency NetPrice();
    virtual Currency DiscountPrice();

    virtual void Add(Equipment*);
    virtual void Remove(Equipment*);
    virtual Iterator* CreateIterator();
protected:
    Equipment(const char*);
private:
    const char* _name;
};
```

Exemplo

```
class FloppyDisk : public Equipment {
public:
    FloppyDisk(const char*);
    virtual ~FloppyDisk();

    virtual Watt Power();
    virtual Currency NetPrice();
    virtual Currency DiscountPrice();
};
```

Nó Folha

```
class CompositeEquipment : public Equipment {
public:
    virtual ~CompositeEquipment();

    virtual Watt Power();
    virtual Currency NetPrice();
    virtual Currency DiscountPrice();

    virtual void Add(Equipment*);
    virtual void Remove(Equipment*);
    virtual Iterator* CreateIterator();

protected:
    CompositeEquipment(const char*);
private:
    List _equipment;
};
```

Nó Composto

```
class Chassis : public CompositeEquipment {
public:
    Chassis(const char*);
    virtual ~Chassis();

    virtual Watt Power();
    virtual Currency NetPrice();
    virtual Currency DiscountPrice();
};
```

Nó Folha

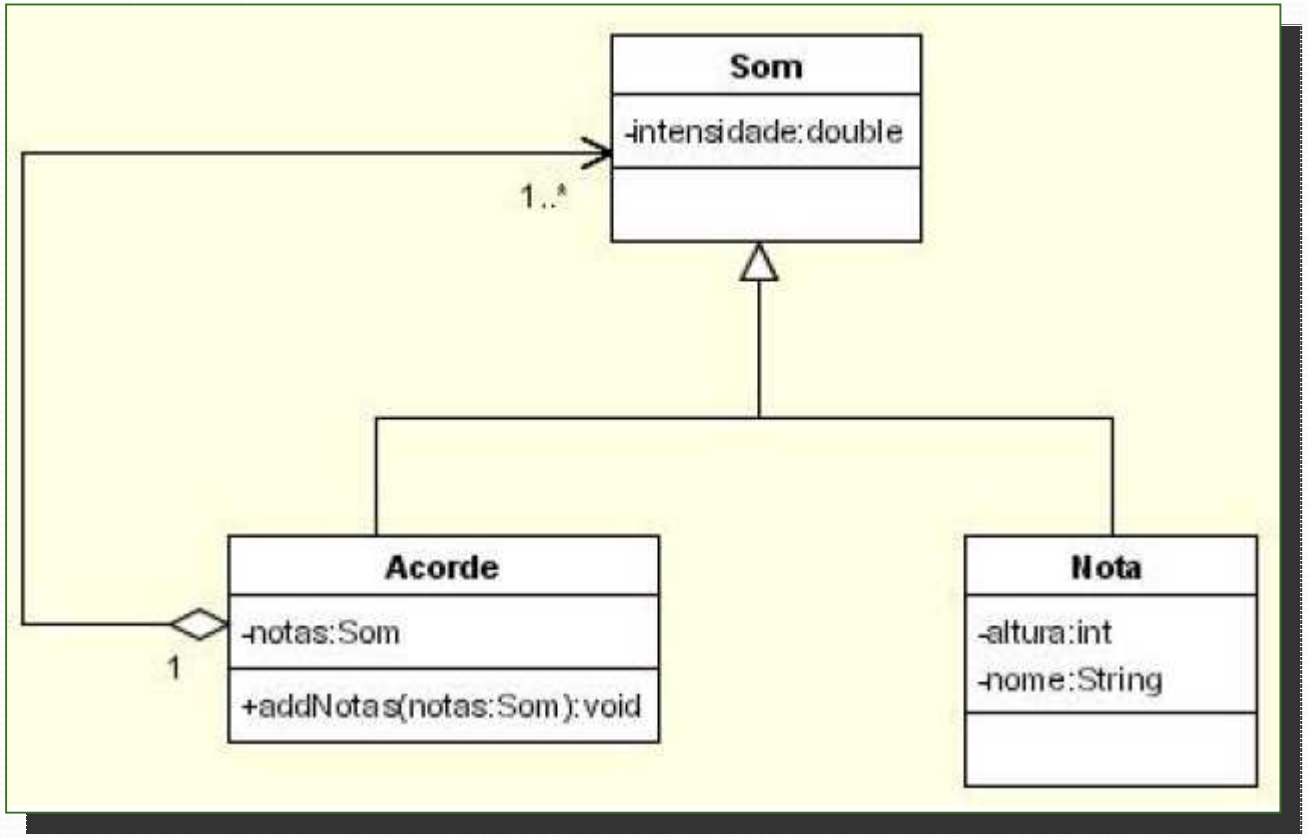
Exemplo

```
Currency CompositeEquipment::NetPrice () {
    Iterator* i = CreateIterator();
    Currency total = 0;

    for (i->First(); !i->IsDone(); i->Next()) {
        total += i->CurrentItem()->NetPrice();
    }
    delete i;
    return total;
}
```

```
Cabinet* cabinet = new Cabinet("PC Cabinet");
Chassis* chassis = new Chassis("PC Chassis");

cabinet->Add(chassis);

Bus* bus = new Bus("MCA Bus");
bus->Add(new Card("16Mbs Token Ring"));

chassis->Add(bus);
chassis->Add(new FloppyDisk("3.5in Floppy"));

cout << "The net price is " << chassis->NetPrice() << endl;
```

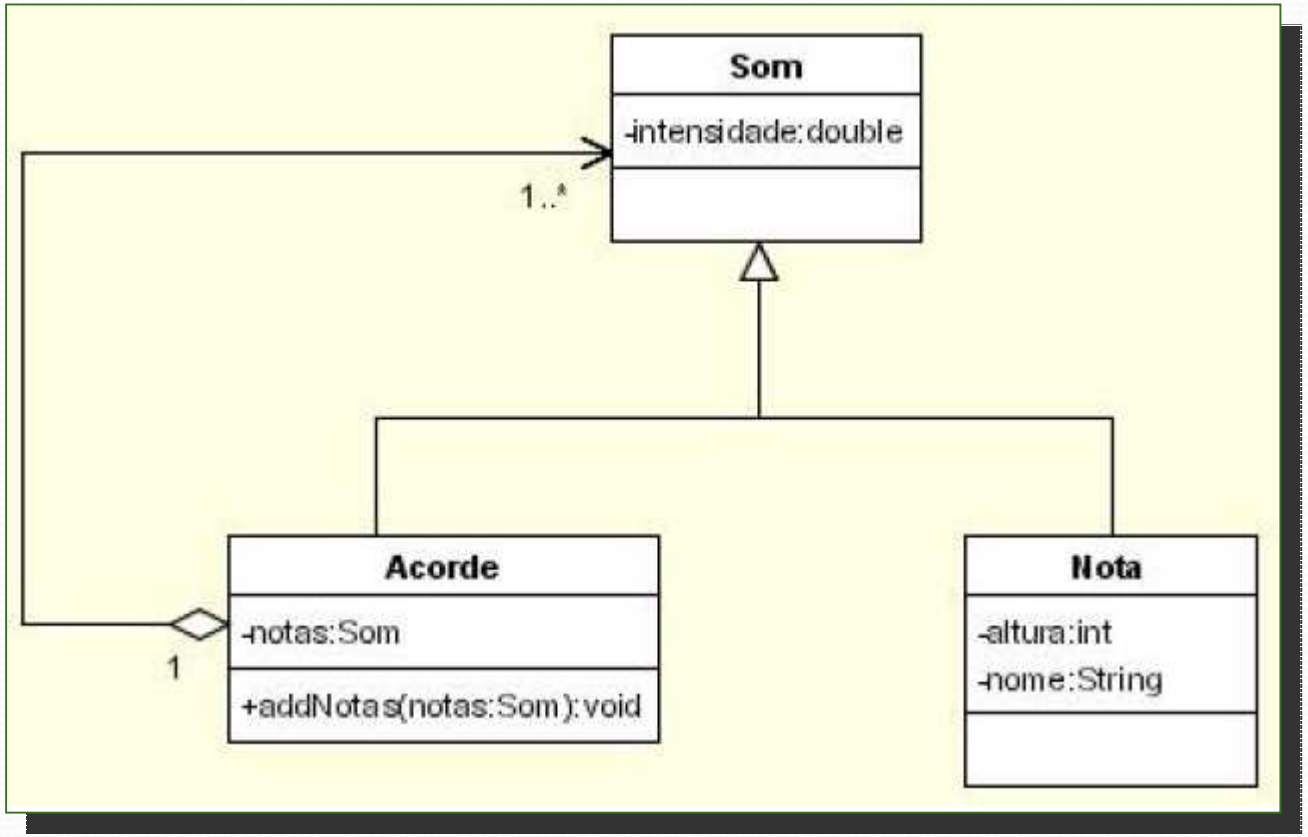
Qual padrão de design está representado no diagrama UML abaixo?

a) Adapter
b) Proxy
c) Mediator
d) Composite
e) Façade



Qual padrão de design está representado no diagrama UML abaixo?

a) Adapter
b) Proxy
c) Mediator
d) Composite
e) Façade



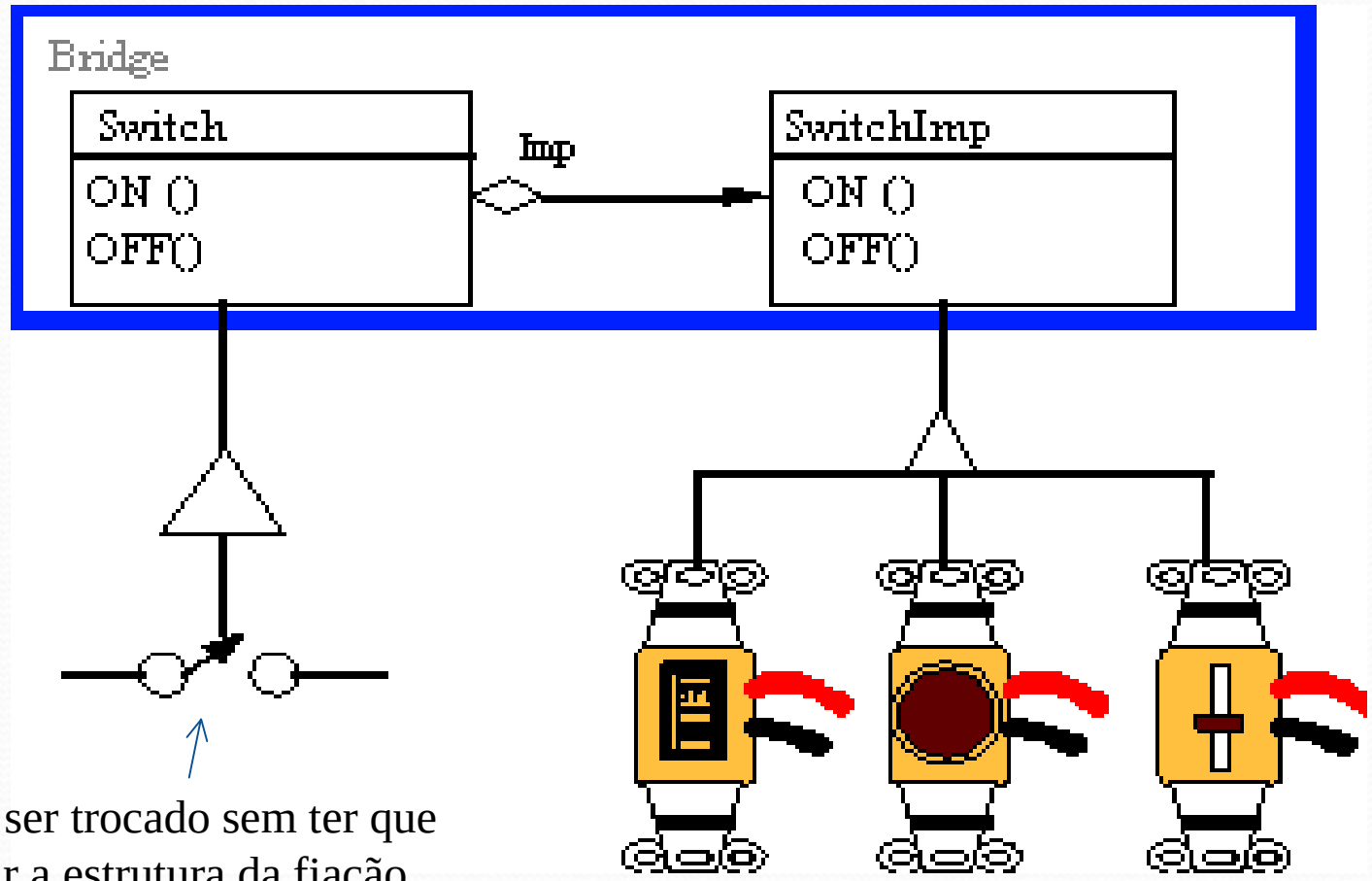
4

Bridge

Objetivo:

"Desacoplar uma abstração de sua implementação para que os dois possam variar independentemente." [GoF]

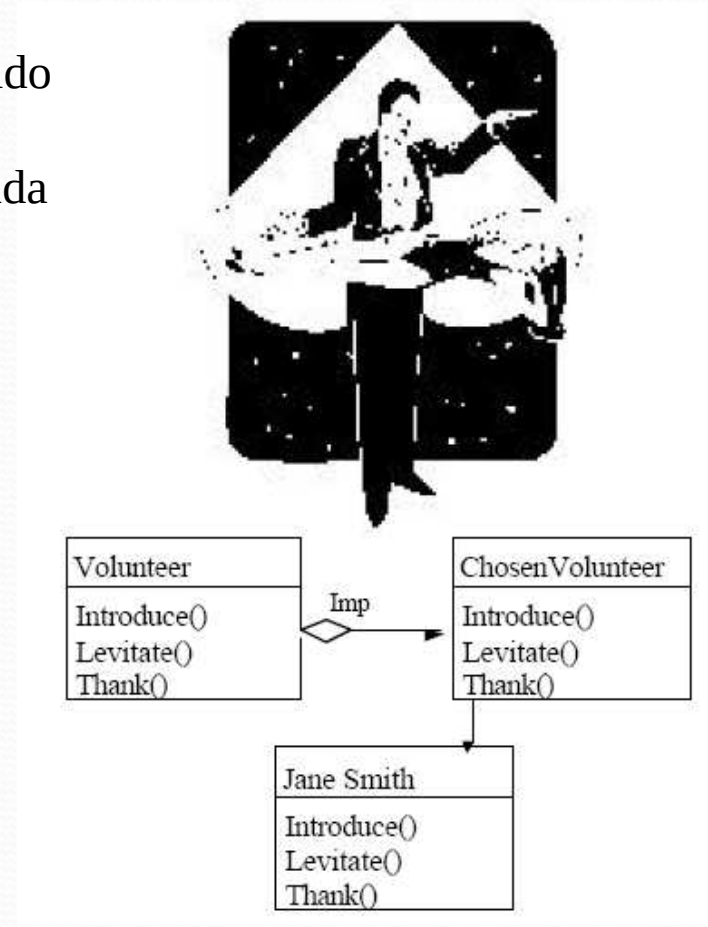
Analogia



Pode ser trocado sem ter que mudar a estrutura da fiação

Analogia

- O voluntário só é conhecido na hora da performance
- Ele pode ser alterado a cada performance

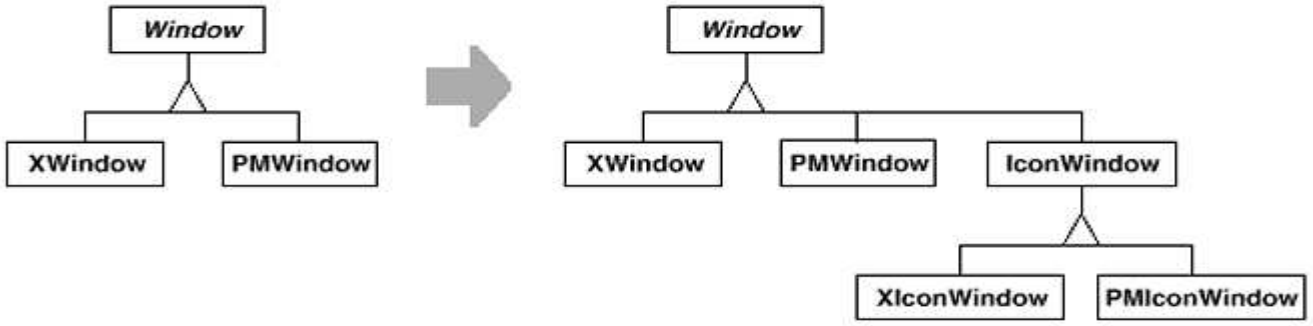


Motivação

- Quando uma abstração pode ter mais de uma implementação, usa-se herança
  - Não é muito flexível
  - Acopla implementacao e abstracao permanentemente
  - Dificulta: modificação e reuso de forma independente

Exemplo

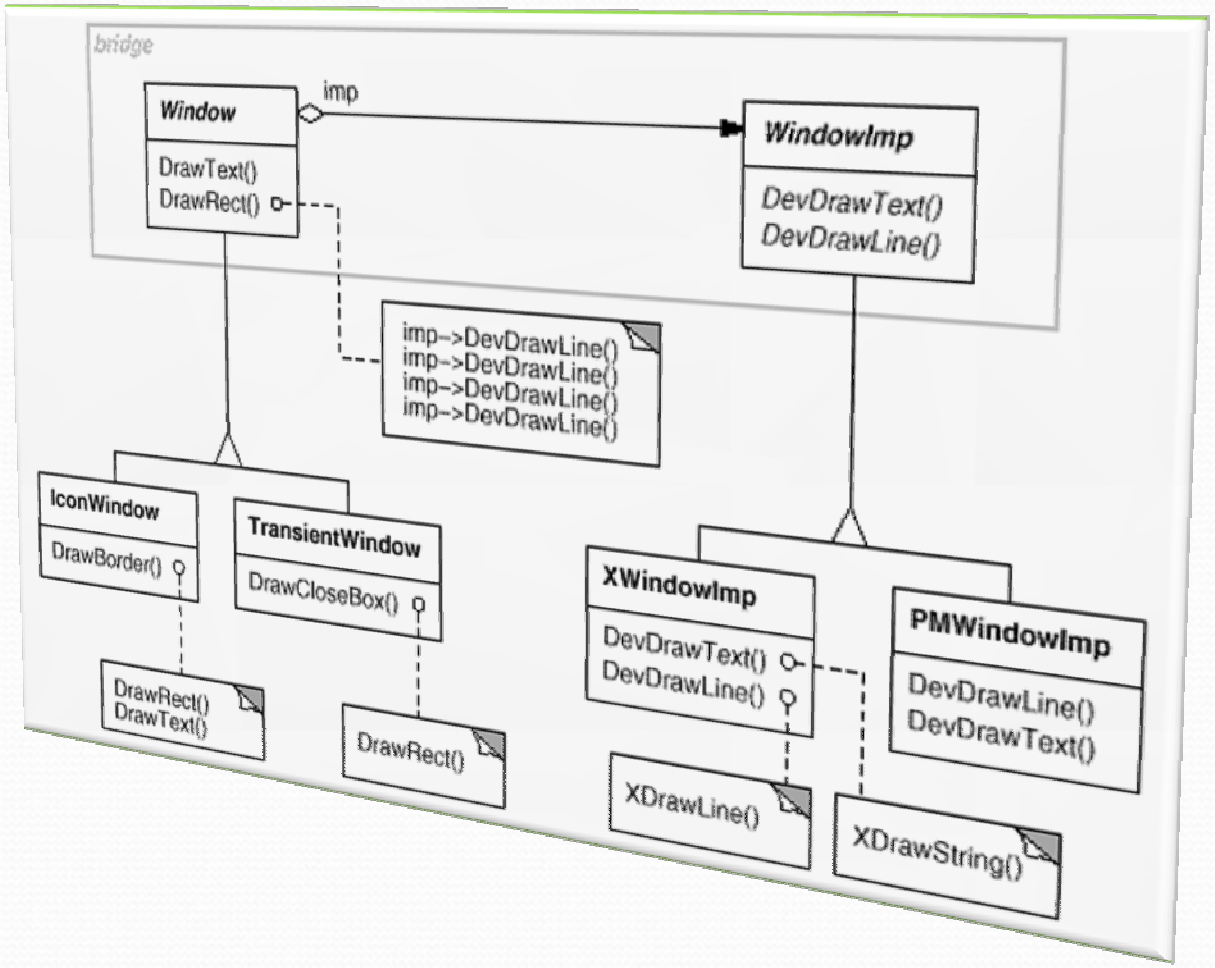
- Desenvolver um componente Window para atuar em diferentes plataformas
  - X Window System
  - IBM`s Presentation Manager
- Esquema com herança:



- Cliente se torna dependente de plataforma

Solução - Bridge

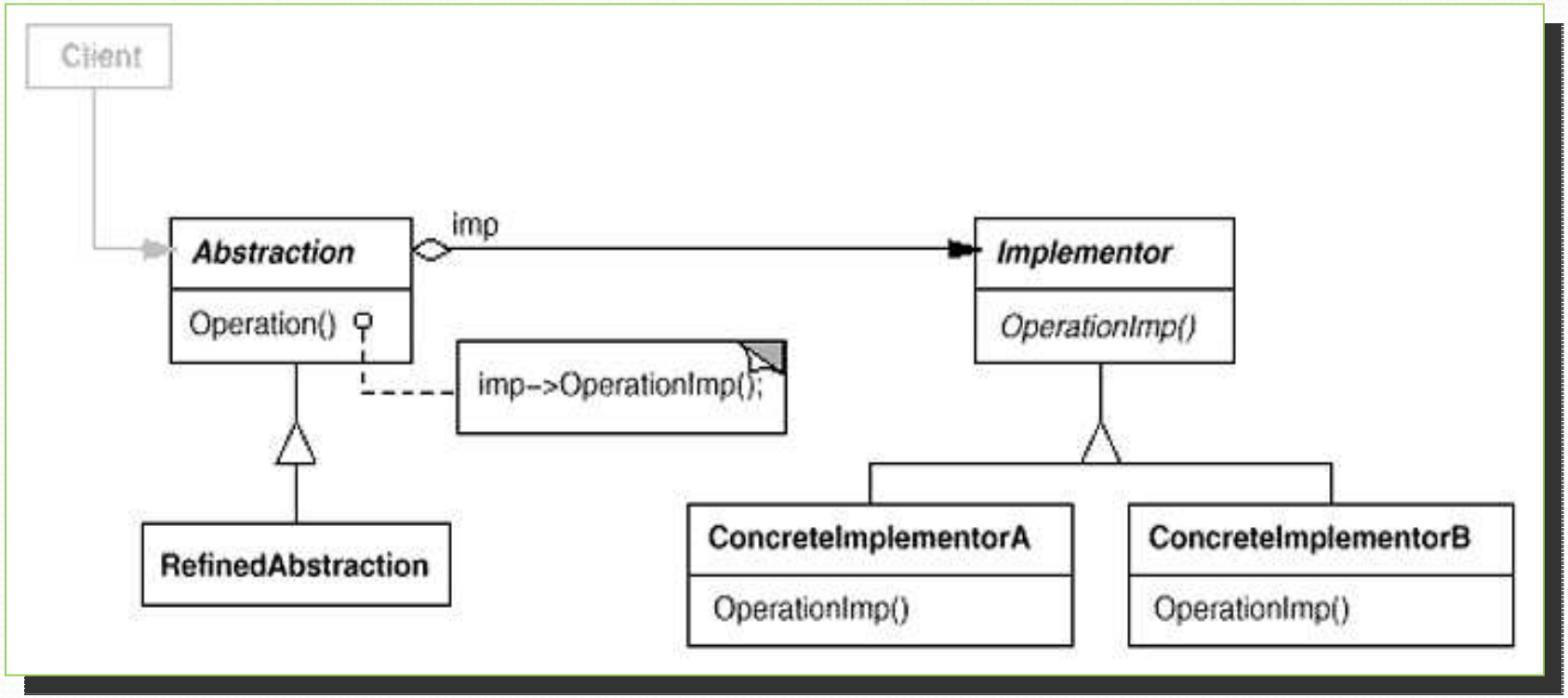
- Hierarquias de classes separadas
- Relação: Window – WindowImp = bridge
  - Os dois lados podem variar independentemente



Aplicabilidade

- Para evitar acoplamento permanente entre a abstração e a implementação
  - Casos em que a implementação deve ser escolhida em tempo de execução
- Abstrações e implementações devem ser estendidas de forma independente
- Mudanças na implementação de uma abstração não deve afetar o cliente
  - Código cliente não deve ser re-compilado

Estrutura


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

Participantes

- Abstraction (Window)
  - Define a interface de abstração
  - Matem uma referencia para o objeto implementador
- RefinedAbstraction (IconWindow)
  - Estende Abstraction
- Implementor (WindowImp)
  - Define uma interface para implementação das classes
  - Esta interface não precisa corresponder exatamente a interface da classe de abstração
    - Imp: operações primitivas
    - Abs: operações de alto-nível
- ConcreteImplementor (XWindowImp, PMWindowImp)
  - Implementa a interface de Implementor

Conseqüências

- Desacoplamento entre abstração e implementação
  - A implementação de uma abstração pode ser configurada em tempo de execução
  - Um objeto pode mudar sua implementação em tempo de execução
- Elimina dependências em tempo de compilação
  - Pode mudar uma implementação sem necessitar re-compilar a abstração
- Sistema melhor estruturado
- Melhor poder de extensão
  - Pode-se estender ambas hierarquias de forma independente

Exemplo

```
class WindowImp {
public:
    virtual void ImpTop() = 0;
    virtual void ImpBottom() = 0;
    virtual void ImpSetExtent(const Point&) = 0;
    virtual void ImpSetOrigin(const Point&) = 0;

    virtual void DeviceRect(Coord, Coord, Coord, Coord) = 0;
    virtual void DeviceText(const char*, Coord, Coord) = 0;
    virtual void DeviceBitmap(const char*, Coord, Coord) = 0;
    // lots more functions for drawing on windows...
protected:
    WindowImp();
};
```

```
void Window::DrawRect (const Point& p1, const Point& p2) {
    WindowImp* imp = GetWindowImp();
    imp->DeviceRect(p1.X(), p1.Y(), p2.X(), p2.Y());
}
```

```
class Window {
public:
    Window(View* contents);

    // requests handled by window
    virtual void DrawContents();

    virtual void Open();
    virtual void Close();
    virtual void Iconify();
    virtual void Deiconify();

    // requests forwarded to implementation
    virtual void SetOrigin(const Point& at);
    virtual void SetExtent(const Point& extent);
    virtual void Raise();
    virtual void Lower();

    virtual void DrawLine(const Point&, const Point&);
    virtual void DrawRect(const Point&, const Point&);
    virtual void DrawPolygon(const Point[], int n);
    virtual void DrawText(const char*, const Point&);

protected:
    WindowImp* GetWindowImp();
    View* GetView();

private:
    WindowImp* _imp;
    View* _contents; // the window's contents
};
```

Exemplo

```
class IconWindow : public Window {
public:
    // ...
    virtual void DrawContents();
private:
    const char* _bitmapName;
};
```

```
void IconWindow::DrawContents() {
    WindowImp* imp = GetWindowImp();
    if (imp != 0) {
        imp->DeviceBitmap(_bitmapName, 0.0, 0.0);
    }
}
```

```
class ApplicationWindow : public Window {
public:
    // ...
    virtual void DrawContents();
};

void ApplicationWindow::DrawContents () {
    GetView()->DrawOn(this);
}
```

Exemplo

```
class XWindowImp : public WindowImp {
public:
    XWindowImp();

    virtual void DeviceRect(Coord, Coord, Coord, Coord);
    // remainder of public interface...
private:
    // lots of X window system-specific state, including:
    Display* _dpy;
    Drawable _winid;  // window id
    GC _gc;           // window graphic context
};
```

```
void XWindowImp::DeviceRect (
    Coord x0, Coord y0, Coord x1, Coord y1
) {
    int x = round(min(x0, x1));
    int y = round(min(y0, y1));
    int w = round(abs(x0 - x1));
    int h = round(abs(y0 - y1));
    XDrawRectangle(_dpy, _winid, _gc, x, y, w, h);
}
```

```
class PMWindowImp : public WindowImp {
public:
    PMWindowImp();
    virtual void DeviceRect(Coord, Coord, Coord, Coord);

    // remainder of public interface...
private:
    // lots of PM window system-specific state, including:
    HPS _hps;
};
```

Como a classe Window obtém uma instância da implementação certa?

```
WindowImp* Window::GetWindowImp () {
    if (_imp == 0) {
        _imp = WindowSystemFactory::Instance()->MakeWindowImp();
    }
    return _imp;
}
```

Ver Abstractory Factory

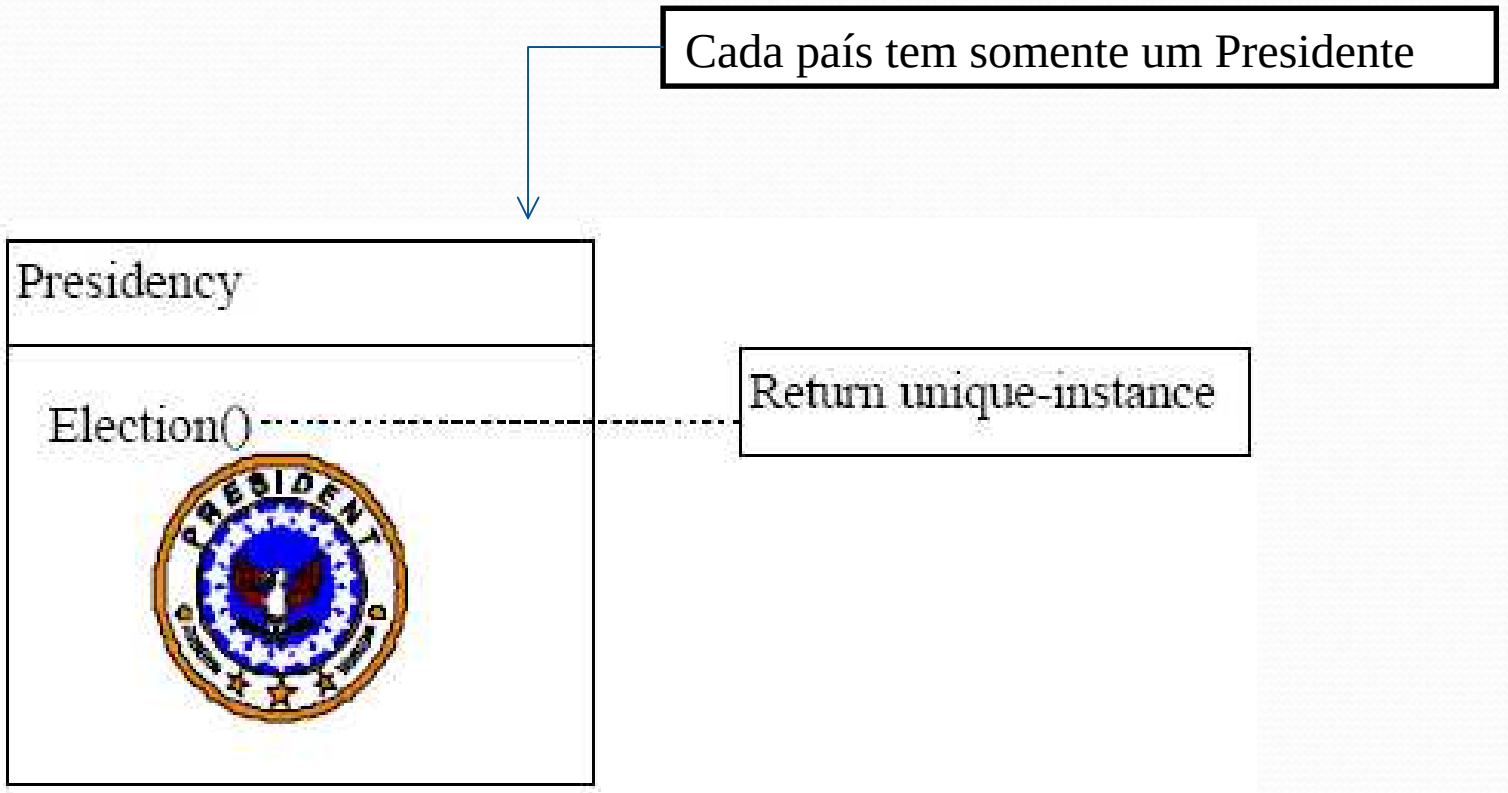
5

Singleton

Objetivo:

"Garantir que uma classe só tenha uma única instância, e prover um ponto de acesso global a ela." [GoF]

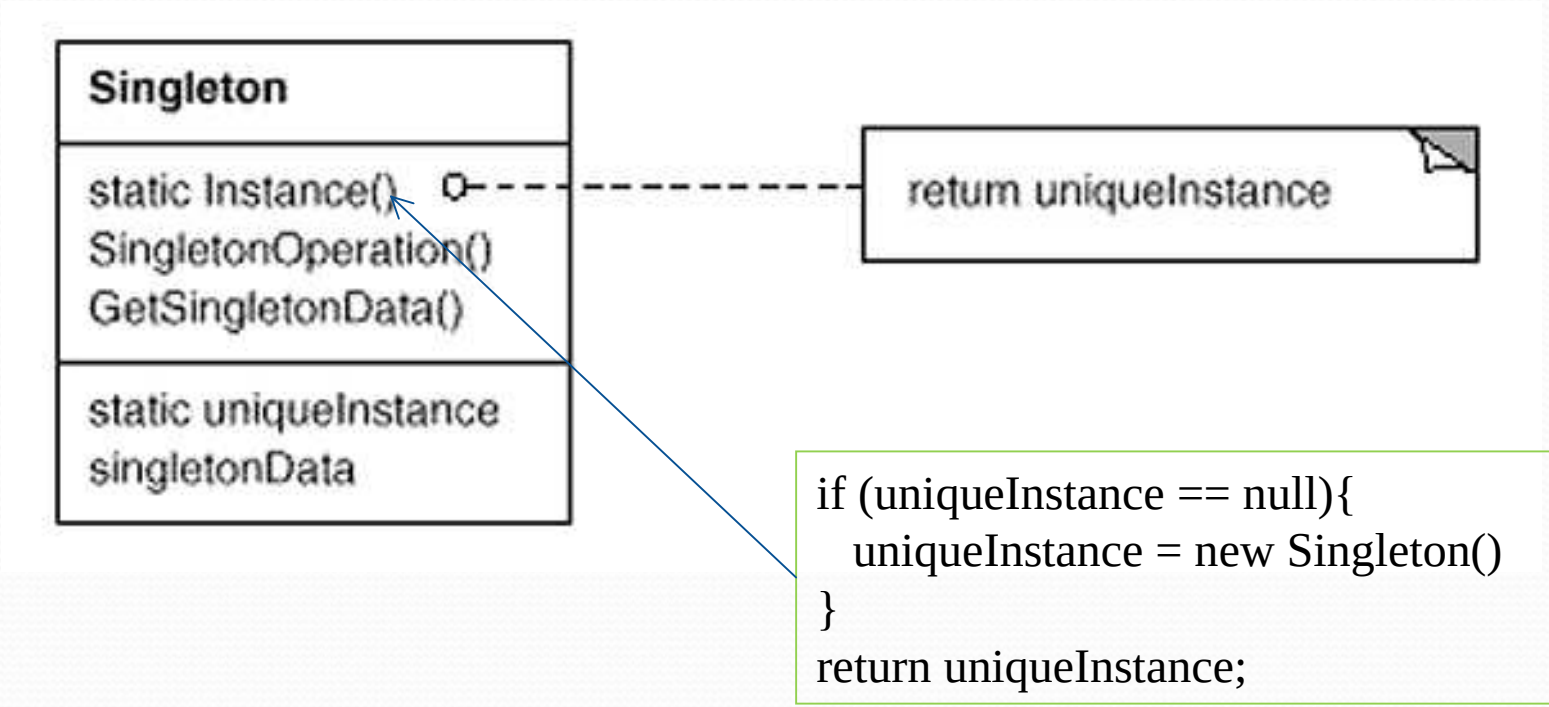
Analogia


Cada país tem somente um Presidente
Presidency
Election()
Return unique-instance
PRESIDENT

Motivação

- Assegura que cada objeto só terá uma única instância
  - Spooler de impressão
  - Um único acesso ao arquivo de log
  - Uma única classe Facade
  - Único banco de dados
- Como assegurar que uma classe terá somente uma única instância e que esta instância seja facilmente acessível?
  - Usando Singleton
  - A própria classe controla a criação de uma única instância

Solução - Singleton



Aplicabilidade

- Quando necessário somente uma instância de uma classe
  - Clientes devem acessar de maneira fácil esta instância
- Quando somente uma instância deve ser derivada por subclasses

Participantes

- Singleton
  - É responsável por criar sua própria única instância
  - Define a operação instance()

Conseqüências

- Acesso controlado a somente uma instância
  - Singleton pode controlar como e quando ter acesso a sua instância
- Permite um variável número de instâncias
  - Pode-se limitar o número de instâncias de um certo tipo numa aplicação
- Pode-se ter subclasses
  - Isto seria impossível se todas as operações fossem estáticas
  - Só antes de ser instanciada pela primeira vez
- Difícil ou impossível de implementar em ambientes distribuídos

Implementação

```
class Singleton {
public:
    static Singleton* Instance();
protected:
    Singleton();
private:
    static Singleton* _instance;
};
```

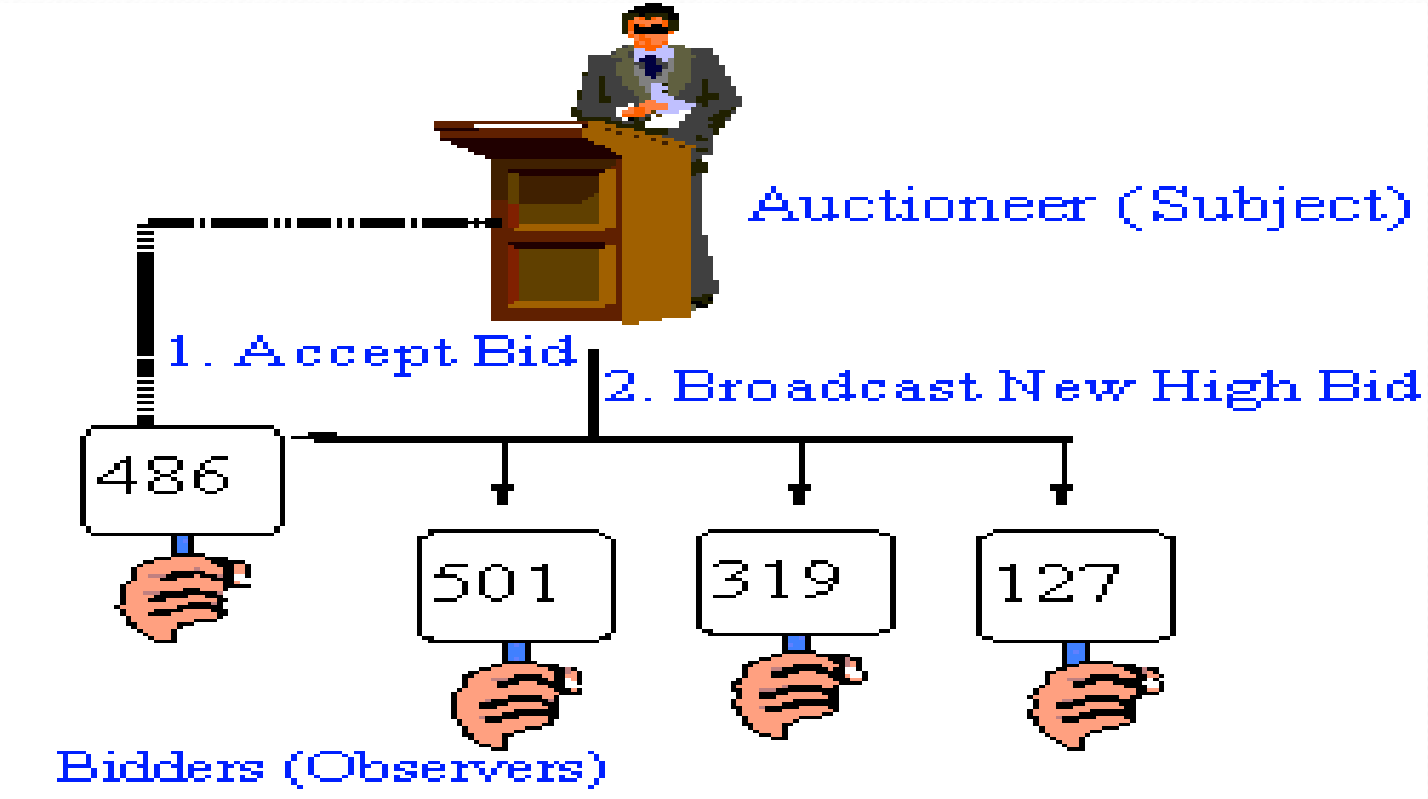
```
Singleton* Singleton::_instance = 0;

Singleton* Singleton::Instance () {
    if (_instance == 0) {
        _instance = new Singleton;
    }
    return _instance;
}
```

Observer

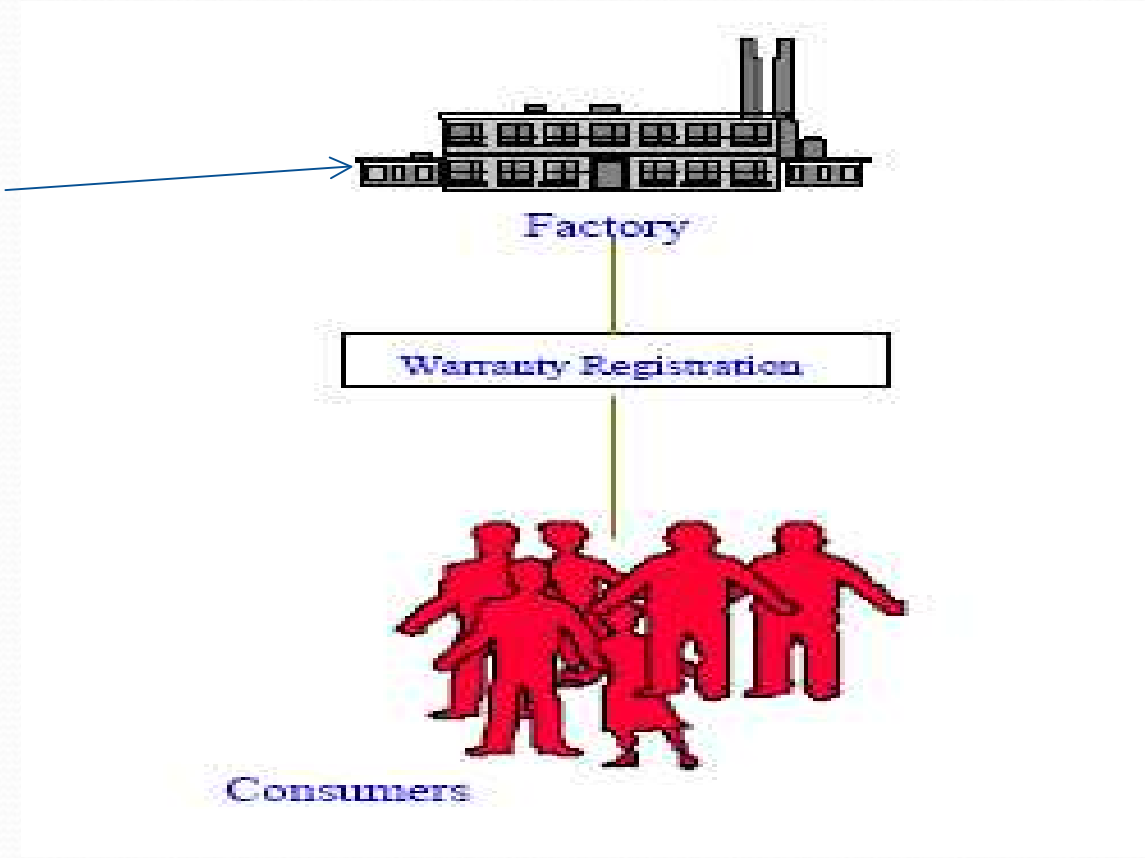
Objetivo:

"Definir uma dependência um-para-muitos entre objetos para que quando um objeto mudar de estado, todos os seus dependentes sejam notificados e atualizados automaticamente." [GoF]

Analogia


Auctioneer (Subject)
1. Accept Bid
2. Broadcast New High Bid
486
501
319
127
Bidders (Observers)

Analogia



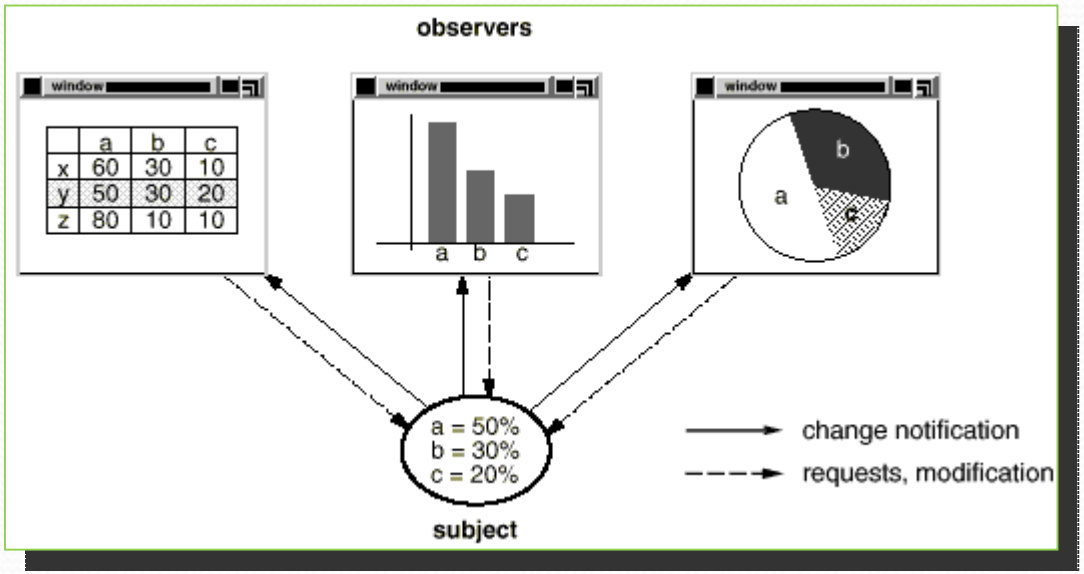
Motivação

- Particionar um sistema em classes colaborativas
  - É preciso consistência
  - Alternativa: classes fortemente acopladas
    - Prejudica o reuso
- Aplicações separam a GUI dos dados da aplicação
  - Classes GUI ou classes que definem dados podem ser reutilizadas livremente
  - Ambas podem trabalhar juntas
  - Tanto uma planilha eletrônica quanto um gráfico de barras podem representar de forma diferente os mesmos dados
    - Eles não se conhecem mas se comportam como se conhecessem
    - Podem ser reutilizados independentemente

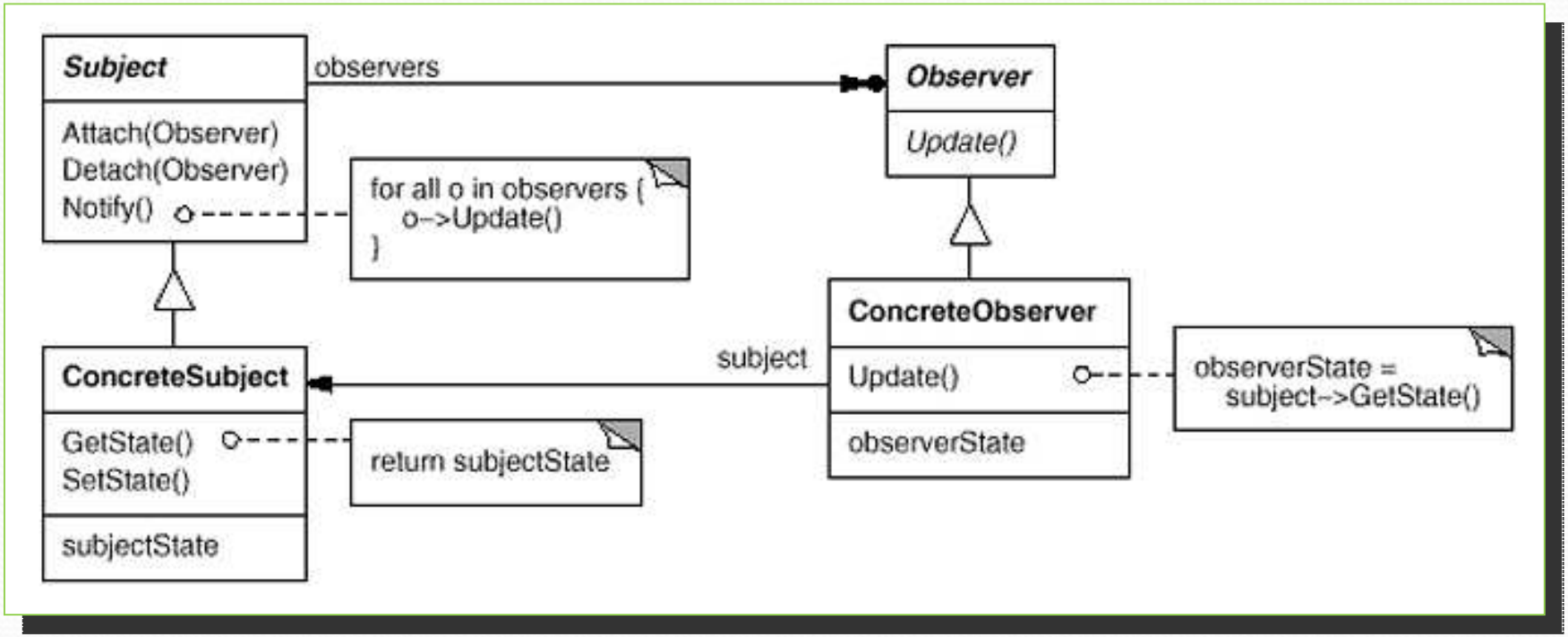
Motivação

- Quando um dado é modificado na planilha, o gráfico de barras reflete a mudança imediatamente
- Ambos são dependentes dos dados
- Não há limites para o número de objetos dependentes
- Como manter esta relação???

Observer



Estrutura


Subject
observers
Observer
Attach(Observer)
Detach(Observer)
Notify()
for all o in observers {
o->Update()
}
Update()
ConcreteObserver
subject
ConcreteSubject
Update()
observerState =
subject->GetState()
GetState()
SetState()
return subjectState
observerState
subjectState

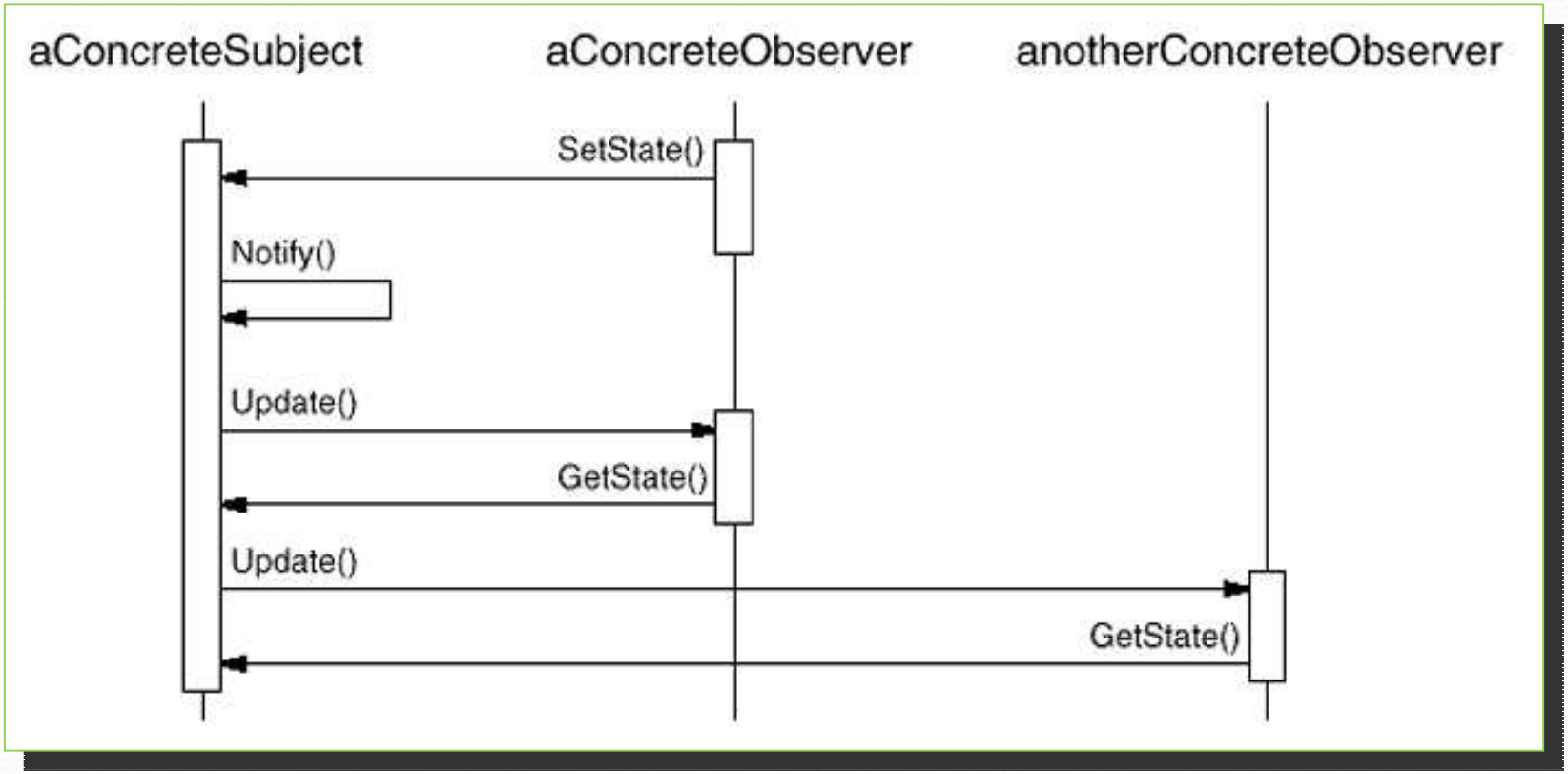
Aplicabilidade

- Quando uma abstração possui um aspecto dependente de outro. Ao encapsular estes aspectos em objetos diferentes, eles podem variar e serem reutilizados livremente
- Quando a mudança do estado de um objeto implica na mudança de estados de outros objetos
  - Não se conhece quantos são estes objetos
- Quando um objeto deve notificar outros objetos sem saber quem eles são
  - Sem forte acoplamento

Participantes

- Subject
  - Conhece seus observadores
  - Um número de observadores podem observar um subject
  - Provê uma interface para atar e dasatar observadores
- Observer
  - Define a interface para que possa ser notificado sobre mudanças ocorridas no subject
- ConcreteSubject
  - Armazena estados de interesse do ConcreteObserver
  - Notifica observadores quando muda de estado
- ConcreteObserver
  - Mantém referência para o ConcreteSubject
  - Armazena estados que devem estar consistente com o subject

Colaboração



Conseqüências

- Cada subject somente conhece a interface abstrata de Observer
  - Acoplamento mínimo
  - Subject não conhece as classes concretas de Observer
- Notificação por broadcast
  - Não é preciso cuidar quantos interessados existem
- O abuso pode causar sério impacto na performance. Sistemas onde todos notificam todos a cada mudança ficam inundados de requisições ("tempestade de eventos")

Implementação

```
class Subject;

class Observer {
public:
    virtual ~ Observer();
    virtual void Update(Subject* theChangedSubject) = 0;
protected:
    Observer();
};
```

```
class Subject {
public:
    virtual ~Subject();

    virtual void Attach(Observer*);
    virtual void Detach(Observer*);
    virtual void Notify();
protected:
    Subject();
private:
    List<Observer*> *_observers;
};

void Subject::Attach (Observer* o) {
    _observers->Append(o);
}

void Subject::Detach (Observer* o) {
    _observers->Remove(o);
}

void Subject::Notify () {
    ListIterator<Observer*> i(_observers);

    for (i.First(); !i.IsDone(); i.Next()) {
        i.CurrentItem()->Update(this);
    }
}
```

Implementação

```
class ClockTimer : public Subject {
public:
    ClockTimer();

    virtual int GetHour();
    virtual int GetMinute();
    virtual int GetSecond();

    void Tick();
};
```

```
void ClockTimer::Tick () {
    // update internal time-keeping state
    // ...
    Notify();
}
```

```
class DigitalClock: public Widget, public Observer {
public:
    DigitalClock(ClockTimer*);
    virtual ~DigitalClock();

    virtual void Update(Subject*);
        // overrides Observer operation

    virtual void Draw();
        // overrides Widget operation;
        // defines how to draw the digital clock
private:
    ClockTimer* _subject;
};

DigitalClock::DigitalClock (ClockTimer* s) {
    _subject = s;
    _subject->Attach(this);
}

DigitalClock:: DigitalClock () {
    _subject->Detach(this);
}
```

Implementação

```
void DigitalClock::Update (Subject* theChangedSubject) {
    if (theChangedSubject == _subject) {
        Draw();
    }
}

void DigitalClock::Draw () {
    // get the new values from the subject

    int hour = _subject->GetHour();
    int minute = _subject->GetMinute();
    // etc.

    // draw the digital clock
}
```

```
class AnalogClock : public Widget, public Observer {
public:
    AnalogClock(ClockTimer*);
    virtual void Update(Subject*);
    virtual void Draw();
    // ...
};
```

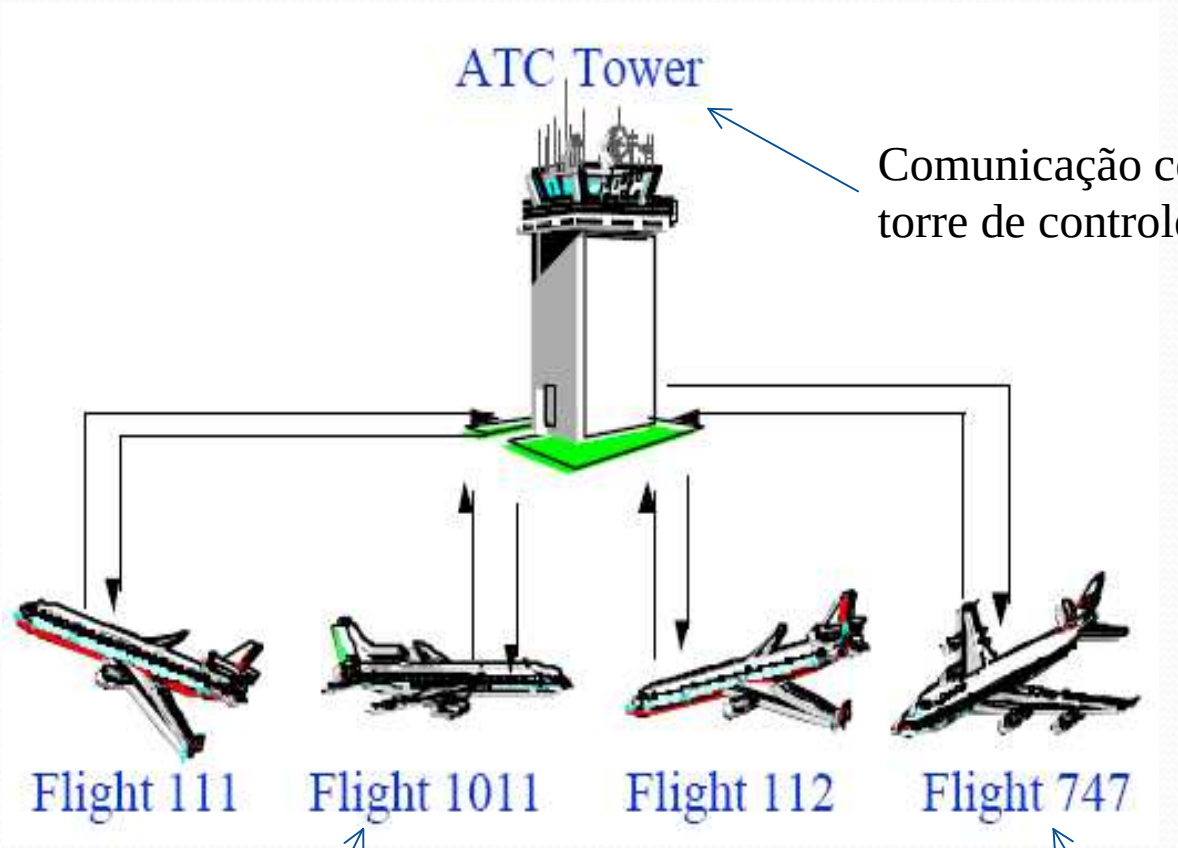
7

Mediator

Objetivo:

"Definir um objeto que encapsula como um conjunto de objetos interagem. Mediator promove acoplamento fraco ao manter objetos que não se referem um ao outro explicitamente, permitindo variar sua interação independentemente." [GoF]

Analogia

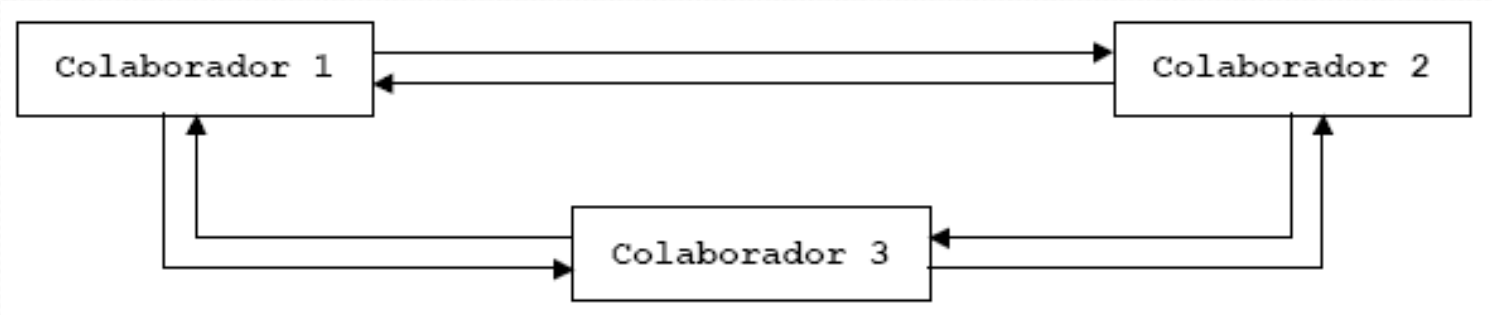

ATC Tower
Comunicação centralizada na torre de controle
Flight 111
Flight 1011
Flight 112
Flight 747
Saída
Chegada

Motivação

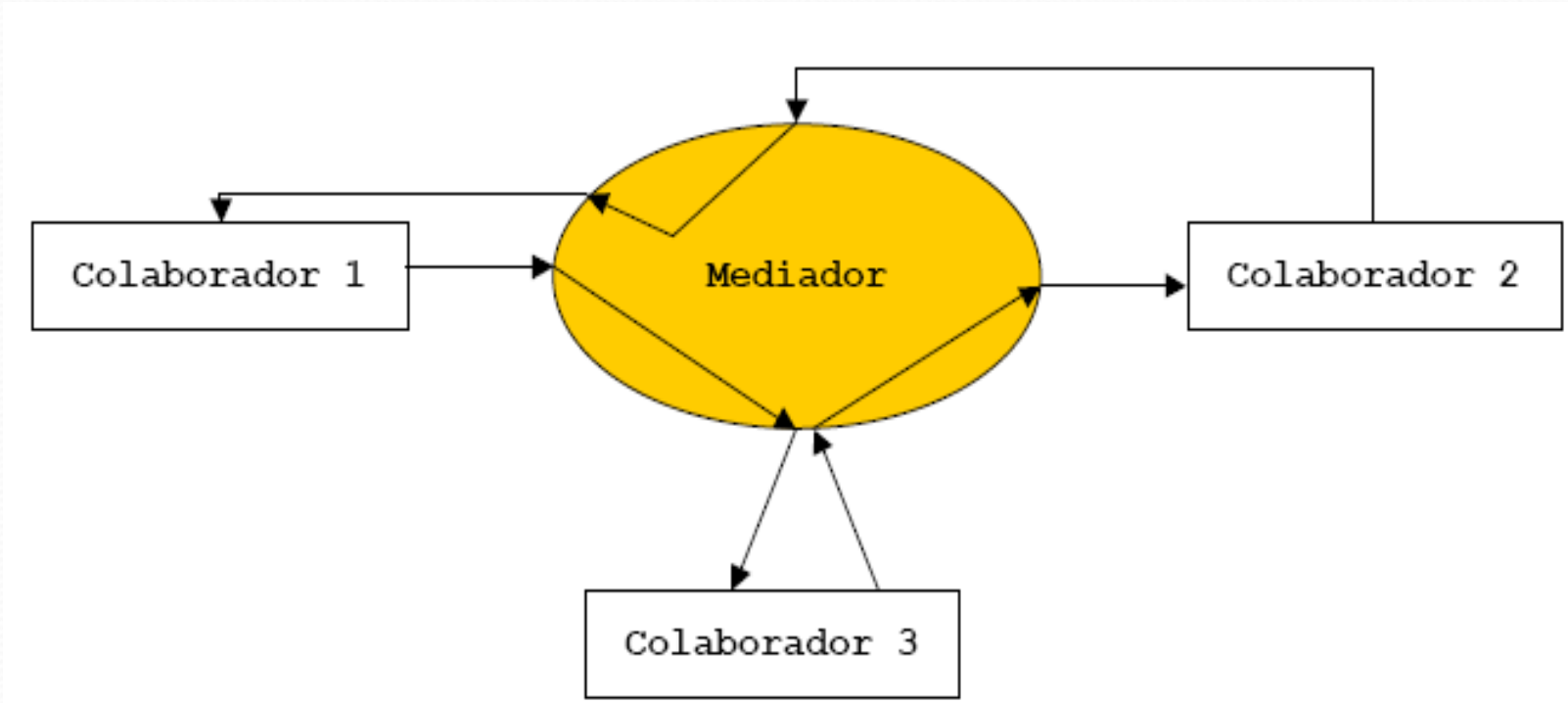
- OO encoraja a distribuição de comportamentos entre objetos
  - Há muita conexão entre objetos
  - No pior, caso cada objeto conhece cada outro
- O particionamento de um sistema em muitos objetos aumenta a reusabilidade
  - Muitas conexões entre eles prejudica a reusabilidade
  - O sistema parece monolítico (indivisível)
  - Alterações precisam ser feitas em muitos objetos – o comportamento está distribuído entre os objetos
  - Forte acoplamento

Motivação

- Como permitir que um grupo de objetos se comunique entre si sem que haja acoplamento entre eles?
- Como remover o forte acoplamento presente em relacionamentos muitos para muitos?
  - Ex.: Professores x Alunos
- Como permitir que novos participantes sejam ligados ao grupo facilmente?



Solução - Mediator



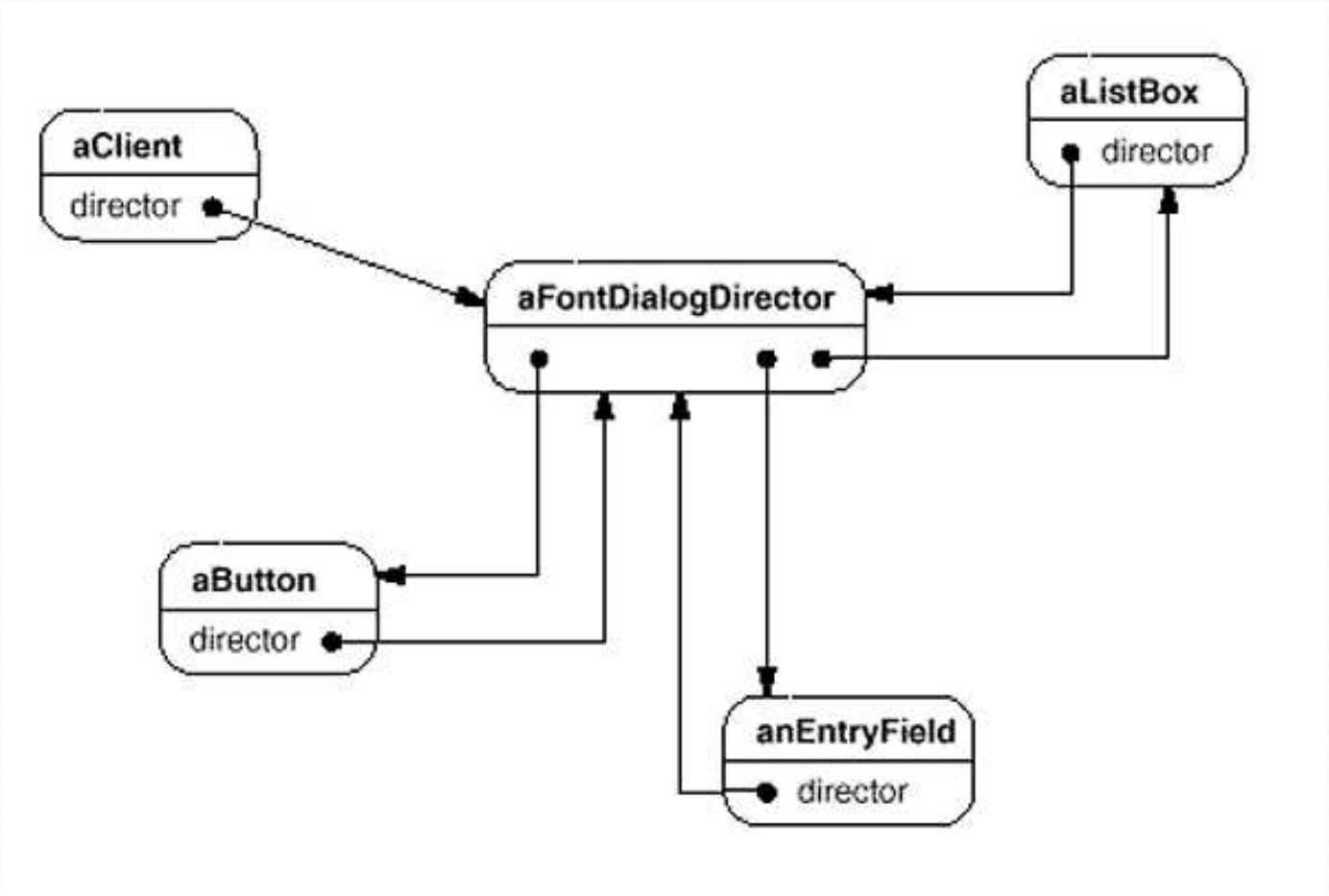
Motivação - Exemplo

- Objetos widgets são dependentes
- Exemplos:
  - O botão é desabilitado quando o campo texto está vazio
  - Ao selecionar um elemento na lista, muda o valor do campo texto
  - Entrando com um valor no campo de texto, um ou vários valores podem ser selecionados na lista

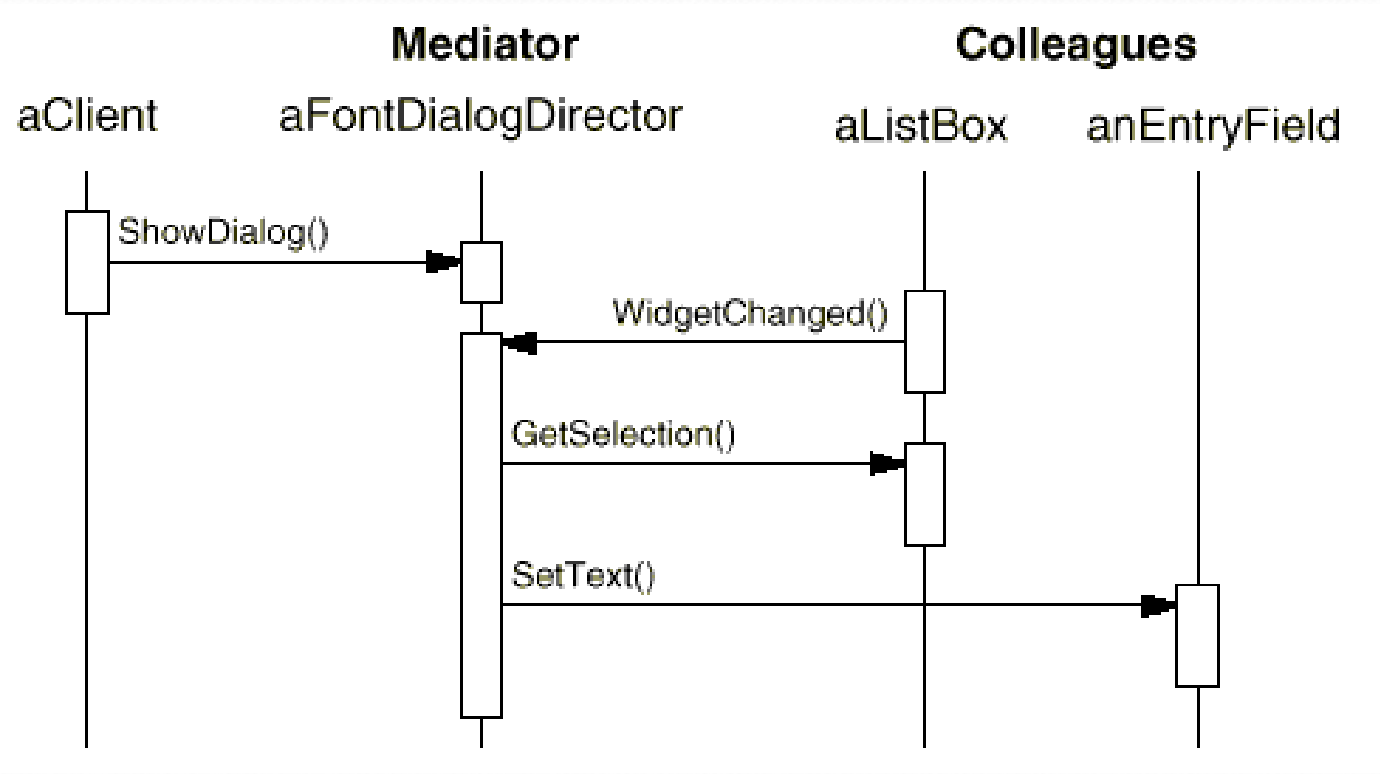
Problema

- Dificuldade em reusar o Diálogo
  - É preciso estender cada Widget que se deseja mudar o comportamento
  - Nem todo diálogo precisa ter as mesmas dependências
- Solução
  - Encapsular um conjunto de comportamentos em um único objeto – Mediator
  - Mediator :
    - é responsável pelo controle e coordenação da interação entre um grupo de objetos
    - Elemento intermediário
    - Objetos somente conhecem o mediator
  - Para mudar o comportamento dos Widgets, basta alterar o comportamento de uma única classe (herança)

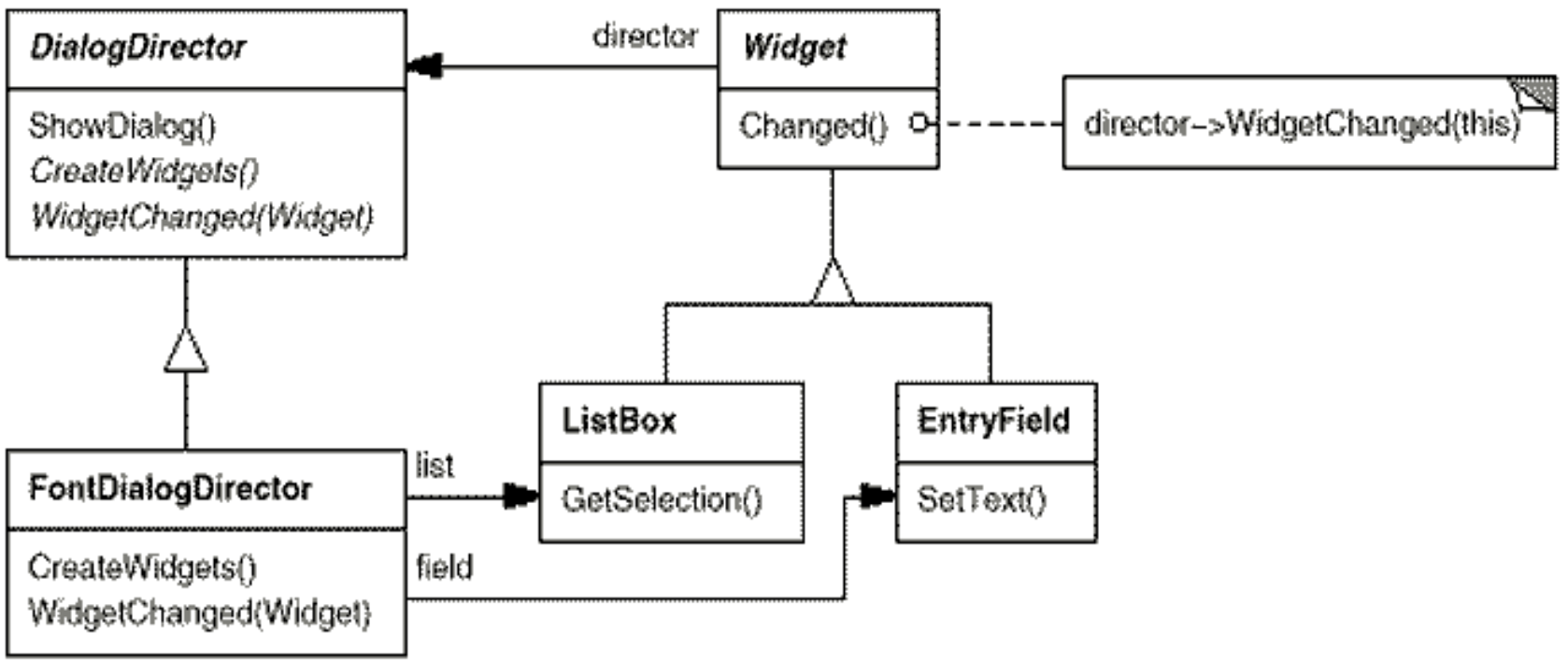
Solução - Mediator



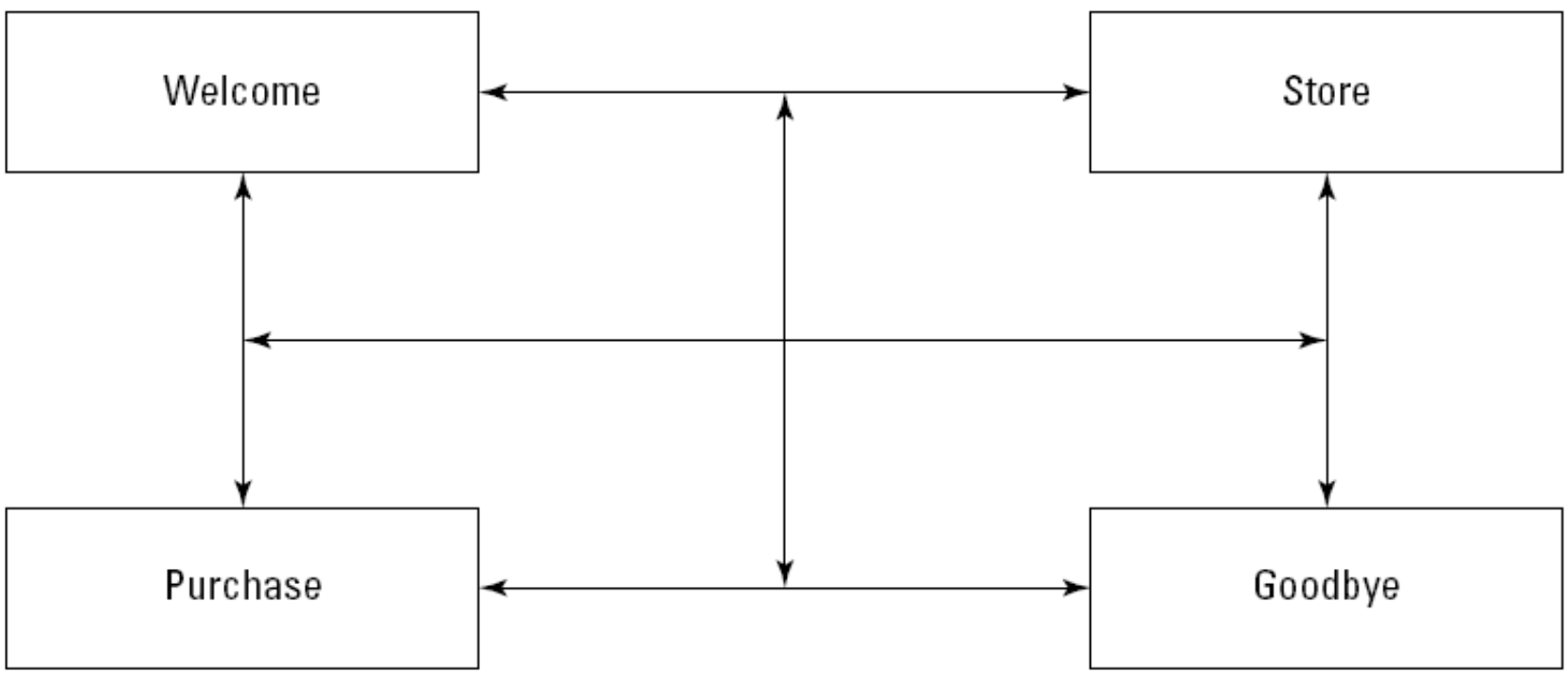
Colaboração



FontDialogDirector

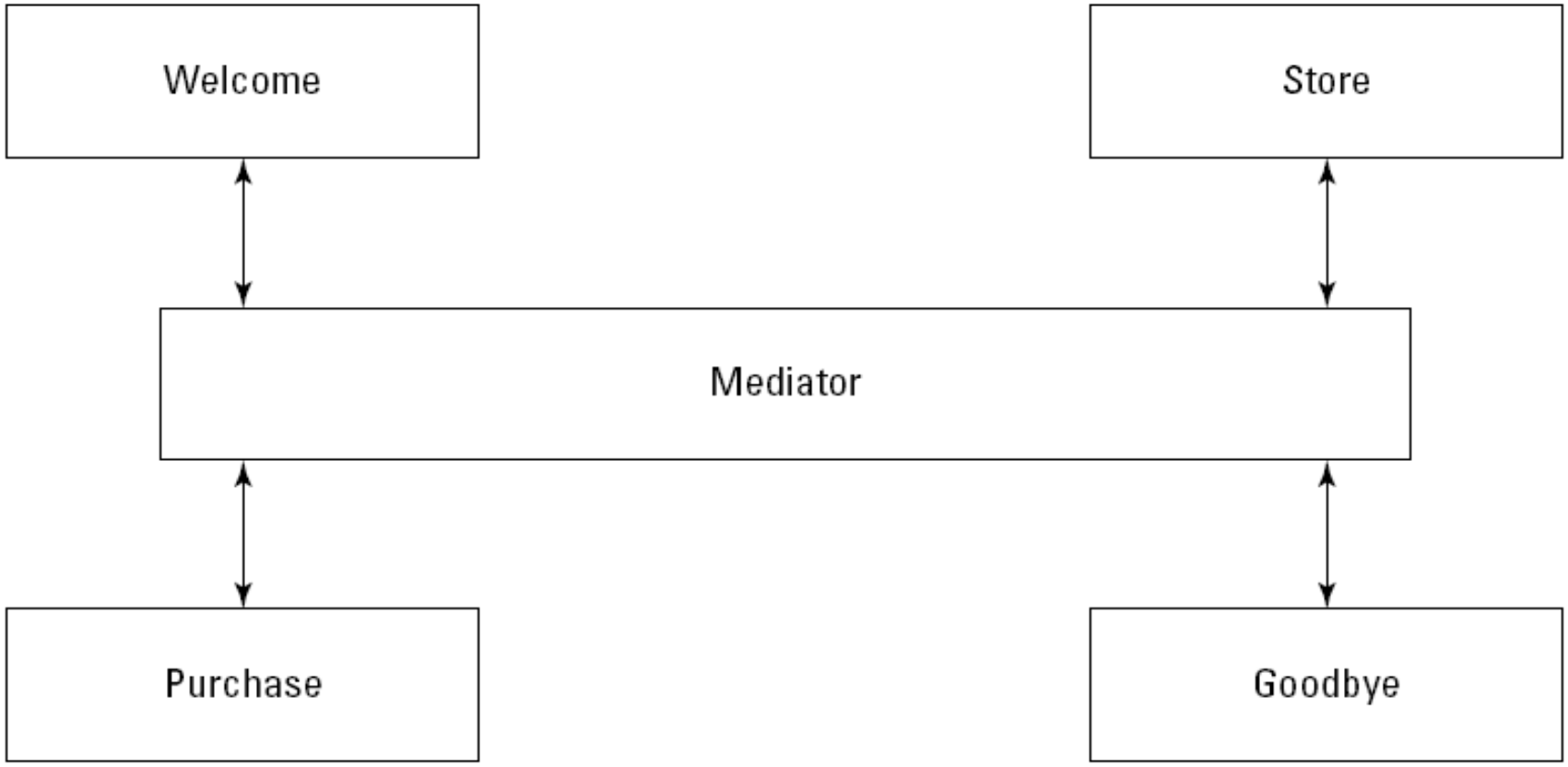


Exemplo: Sistemas de compras Online - Sem Mediator



Cada página carrega a informação de acesso a outras páginas

Exemplo: Sistemas de compras Online

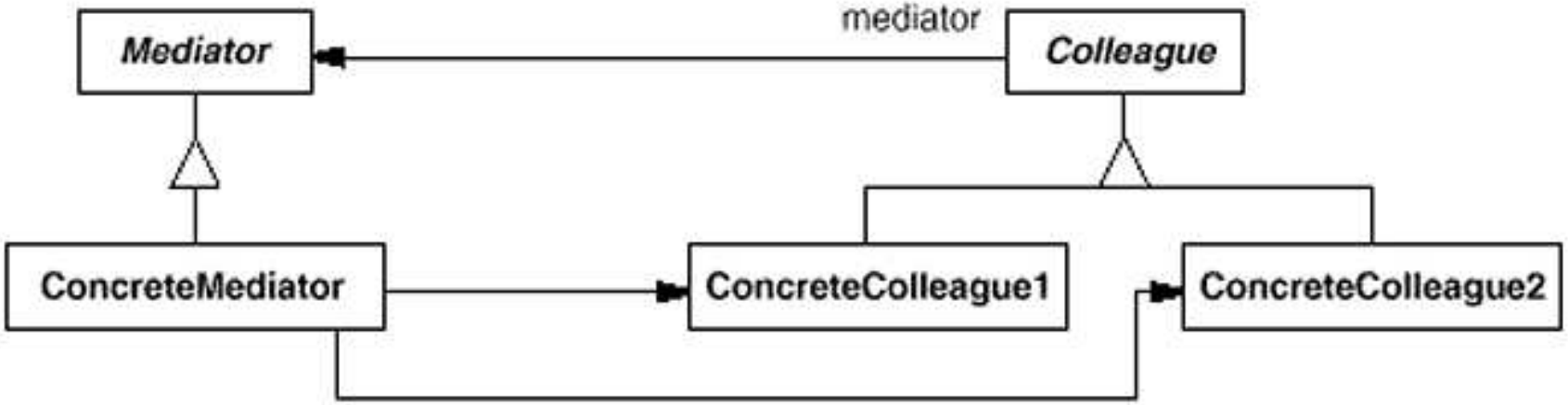


Código de navegação encapsulado

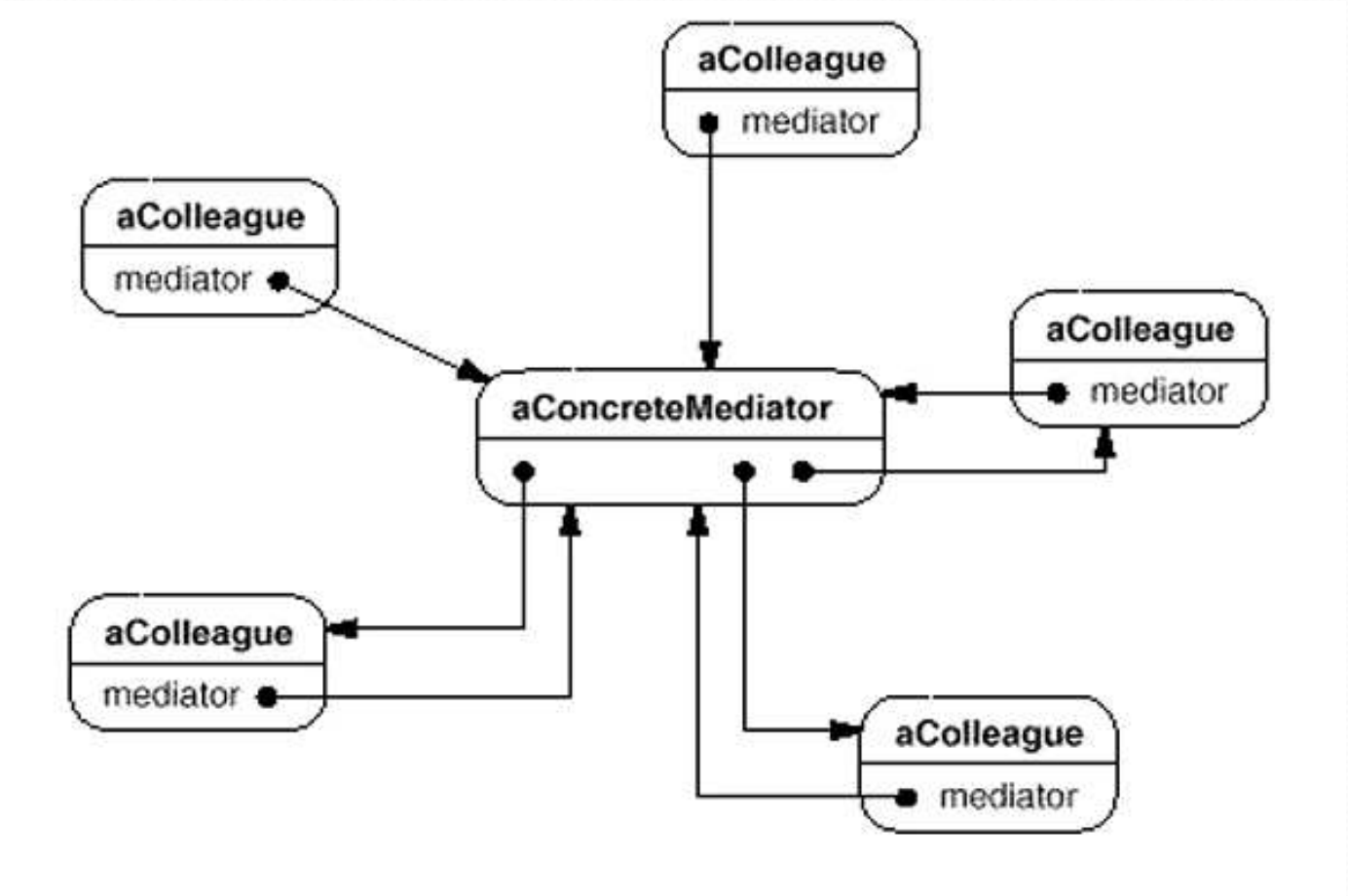
Aplicabilidade

- Quando um conjunto de objetos se comunicam de uma forma complexa
  - Relacionamentos difíceis de entender
- Quando está difícil de reusar um objeto devido ao alto grau de comunicação com vários objetos
- Quando um comportamento que está distribuído entre várias classes deve ser customizado sem uma grande quantidade de heranças

Estrutura


Mediator
mediator
Colleague
ConcreteMediator
ConcreteColleague1
ConcreteColleague2

Estrutura


aColleague
mediator
aColleague
mediator
aColleague
mediator
aConcreteMediator
aColleague
mediator
aColleague
mediator

Participantes

- **Mediator** (DialogDirector)
  - Define uma interface para comunicar com muitos objetos Colleagues
- **ConcreteMediator** (FontDialogDirector)
  - Implementa comportamentos cooperativos para coordenação
  - Conhece e mantém seus Colleagues
- **Colleague classes** (ListBox, EntryField)
  - Cada um conhece o Mediator
  - Toda vez que deseja se comunicar com outro Colleague, ele faz através do Mediator

Conseqüências

- Desacoplamento entre os diversos participantes da rede de comunicação: participantes não se conhecem.
- Diminui o numero de heranças na customização
- Eliminação de relacionamentos muitos para muitos (são todos substituídos por relacionamentos um para muitos)
- A política de comunicações está centralizada no mediador e pode ser alterada sem mexer nos colaboradores.
- O mediator pode ficar sobrecarregado de responsabilidades
  - Periodicamente, rever as responsabilidades do mediator
  - Ele deve somente se preocupar com as iterações entre os objetos
- Dificil reusar um mediator em outro projeto
  - Mas, são fáceis de escrever em comparação com a distribuição de responsabilidade em vários objetos

Implementação - Abstrações

```
class DialogDirector {
public:
    virtual ~DialogDirector();

    virtual void ShowDialog();
    virtual void WidgetChanged(Widget*) = 0;

protected:
    DialogDirector();
    virtual void CreateWidgets() = 0;
};
```

```
void Widget::Changed () {
    _director->WidgetChanged(this);
}
```

```
class Widget {
public:
    Widget(DialogDirector*);
    virtual void Changed();

    virtual void HandleMouse(MouseEvent& event);
    // ...
private:
    DialogDirector* _director;
};
```

Implementação - Widgets

```
class ListBox : public Widget {
public:
    ListBox(DialogDirector*);

    virtual const char* GetSelection();
    virtual void SetList(List<char*>* listItems);
    virtual void HandleMouse(MouseEvent& event);
    // ...
};

class EntryField : public Widget {
public:
    EntryField(DialogDirector*);

    virtual void SetText(const char* text);
    virtual const char* GetText();
    virtual void HandleMouse(MouseEvent& event);
    // ...
};
```

```
class Button : public Widget {
public:
    Button(DialogDirector*);

    virtual void SetText(const char* text);
    virtual void HandleMouse(MouseEvent& event);
    // ...
};

void Button::HandleMouse (MouseEvent& event) {
    // ...
    Changed();
}
```

Implementação - Mediator

```
class FontDialogDirector : public DialogDirector {
public:
    FontDialogDirector();
    virtual ~FontDialogDirector();
    virtual void WidgetChanged(Widget*);

protected:
    virtual void CreateWidgets();

private:
    Button* _ok;
    Button* _cancel;
    ListBox* _fontList;
    EntryField* _fontName;
};
```

```
void FontDialogDirector::CreateWidgets () {
    _ok = new Button(this);
    _cancel = new Button(this);
    _fontList = new ListBox(this);
    _fontName = new EntryField(this);

    // fill the listBox with the available font names

    // assemble the widgets in the dialog
}
```

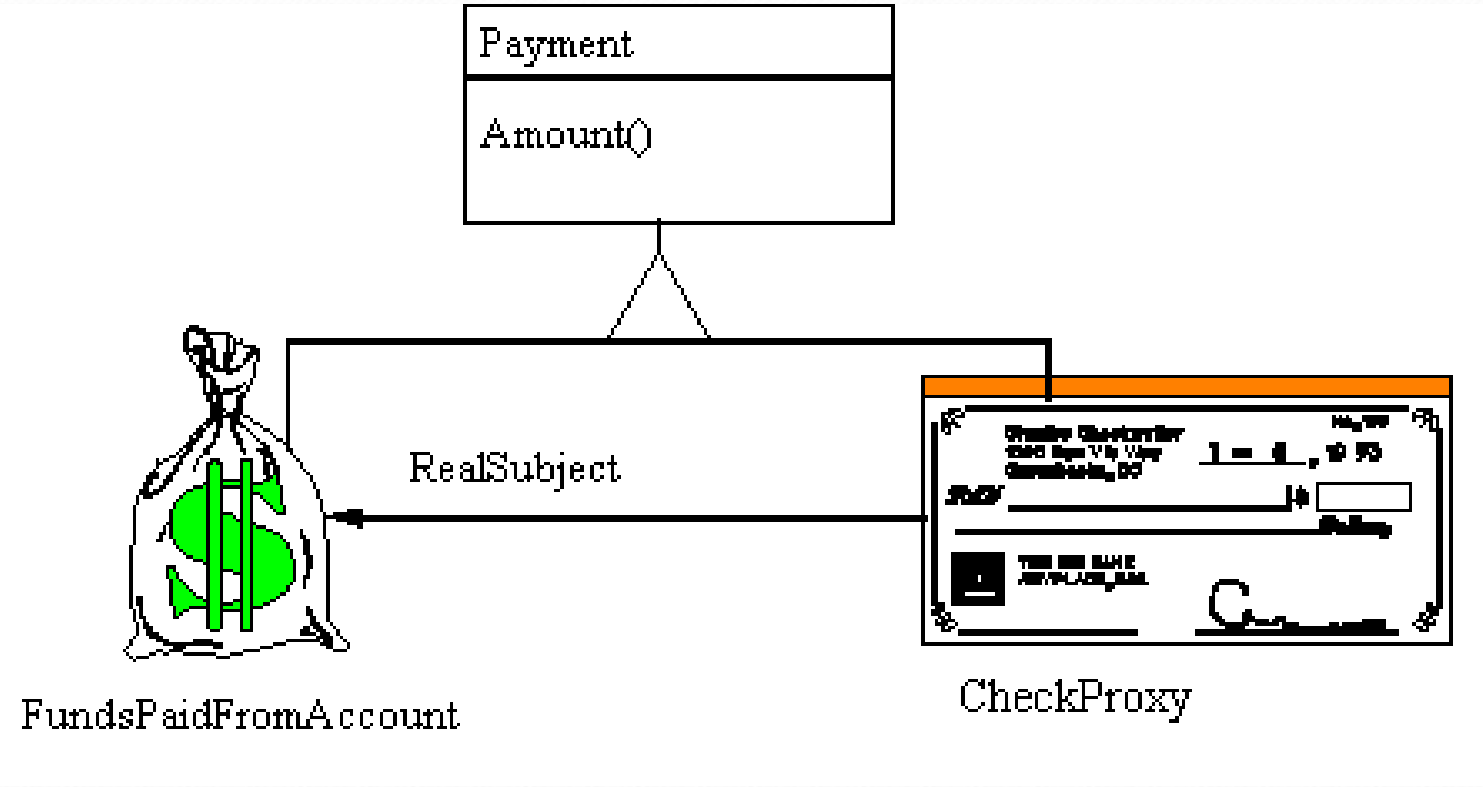
```
void FontDialogDirector::WidgetChanged (
    Widget* theChangedWidget
) {
    if (theChangedWidget == _fontList) {
        _fontName->SetText(_fontList->GetSelection());

    } else if (theChangedWidget == _ok) {
        // apply font change and dismiss dialog
        // ...

    } else if (theChangedWidget == _cancel) {
        // dismiss dialog
    }
}
```

Proxy

Objetivo:

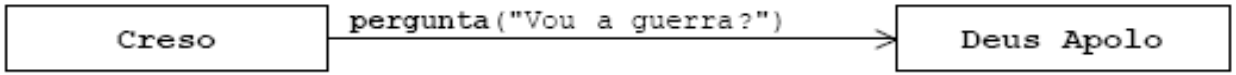
"Prover um substituto ou ponto através do qual um objeto possa controlar o acesso a outro." [GoF]

Analogia

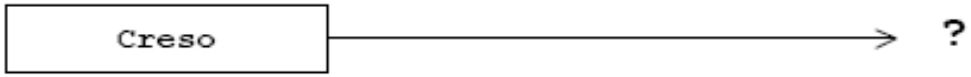

Payment
Amount()
RealSubject
FundsPaidFromAccount
CheckProxy

Motivação

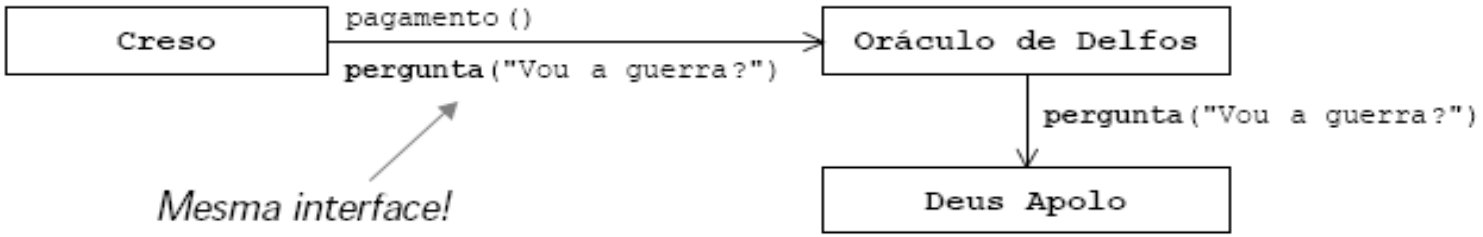
Sistema quer utilizar objeto real...



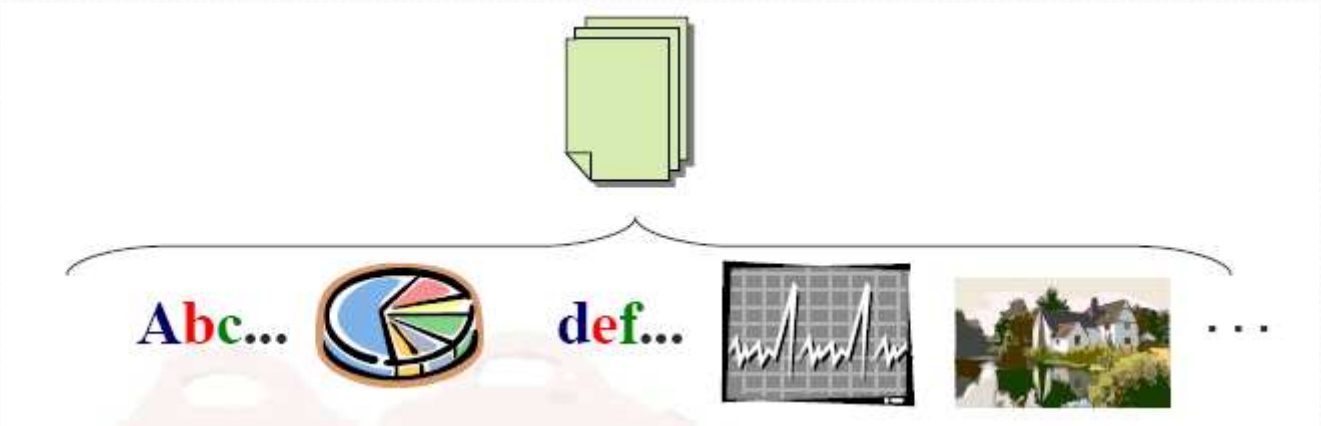
Mas ele não está disponível (remoto, inaccessível, ...)



*Solução: arranjar um **intermediário** que saiba se comunicar com ele eficientemente*



Motivação



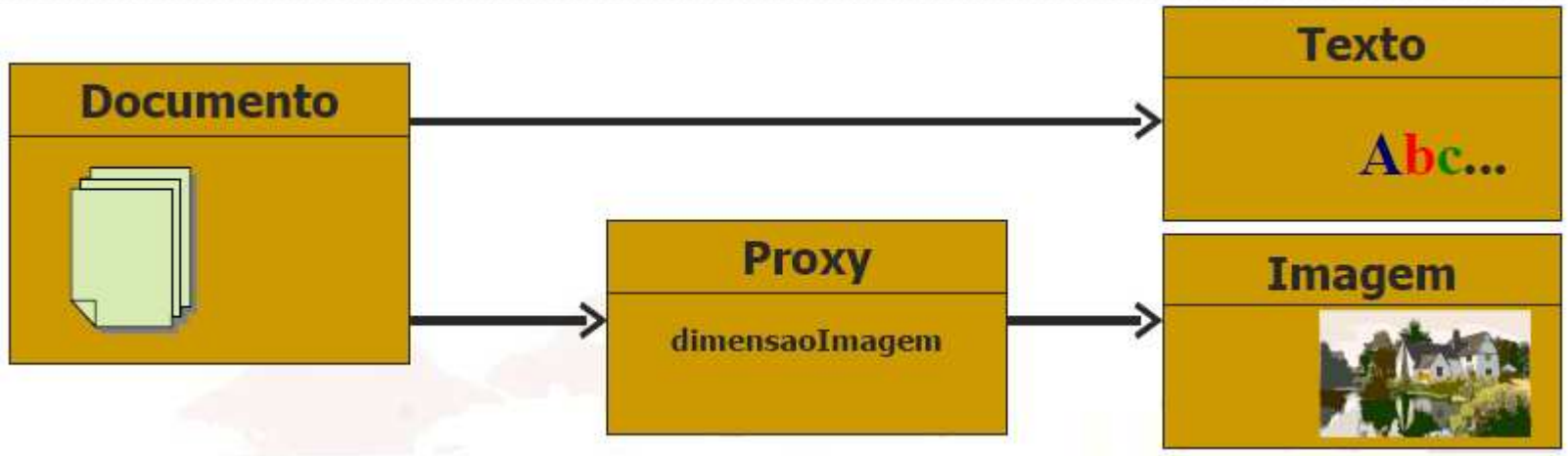
- Considere um editor de texto multimídia (texto e imagens):
- Carregar todas as imagens do texto assim que ele é aberto pode demorar;
  - Deve-se evitar
  - O documento deve ser aberto rapidamente – sem demora
- Nem todas as imagens aparecem na primeira página, muitas estão “escondidas” mais abaixo.
  - Serão apresentadas por demanda

Problemas

- Mas o que se coloca no documento no lugar da imagem?
- Como esconder o fato de que a imagem será criada por demanda para que não complique a implementação do editor?
- Isto não implicará em problemas de desenho e na formatação da imagem na página??
- Como solucionar ???

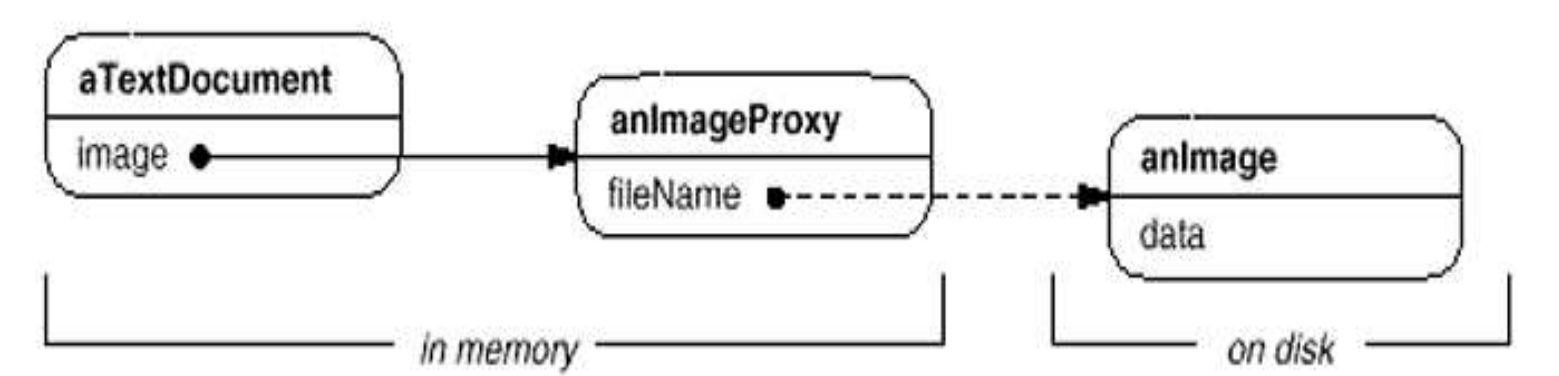
Solução - Proxy

- Documento instancia um objeto Proxy, que possui referência à imagem;
  - É o substituto da imagem
- Assim que necessário, Proxy carrega a imagem – lazy loading.

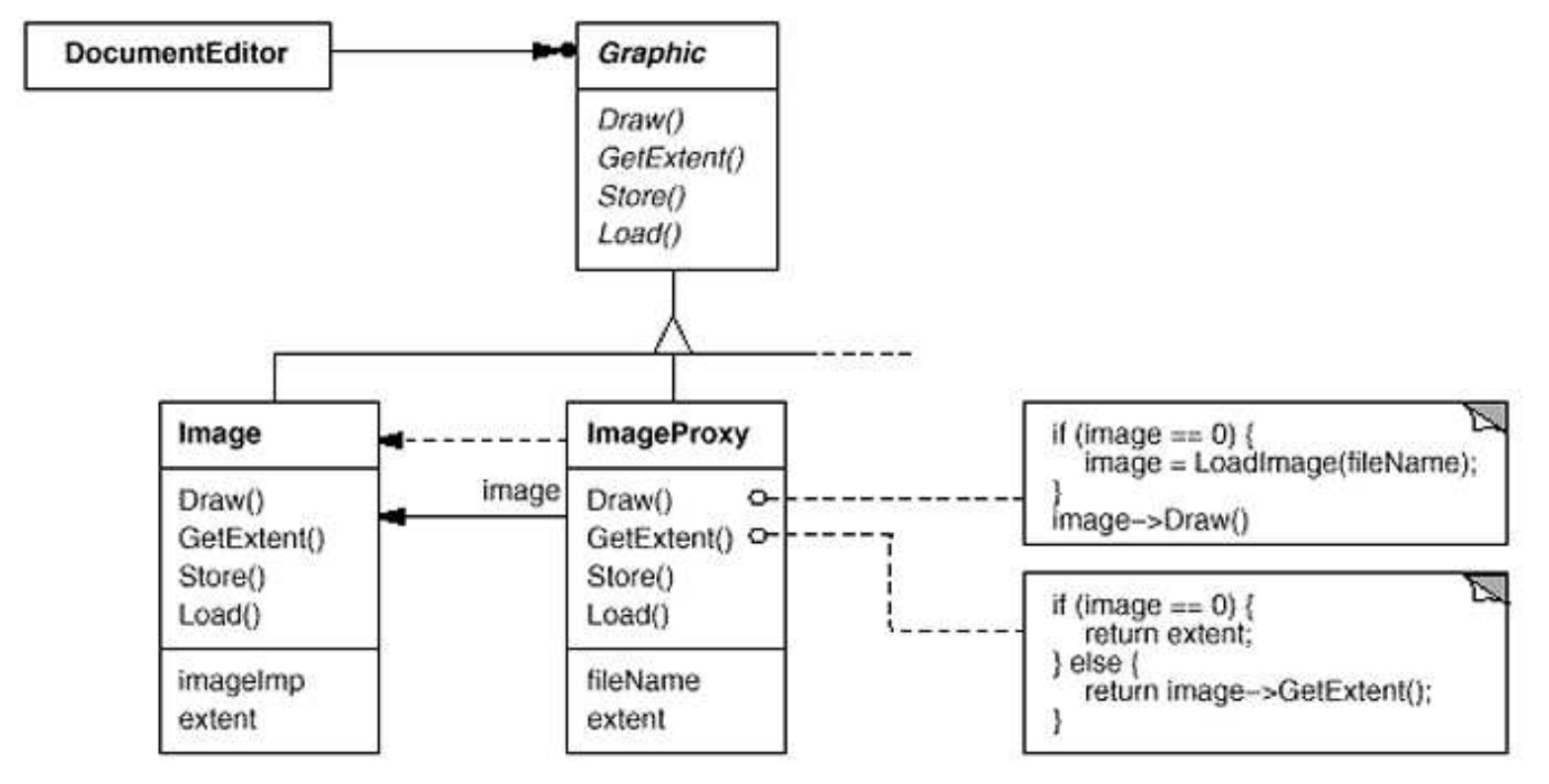


Proxy

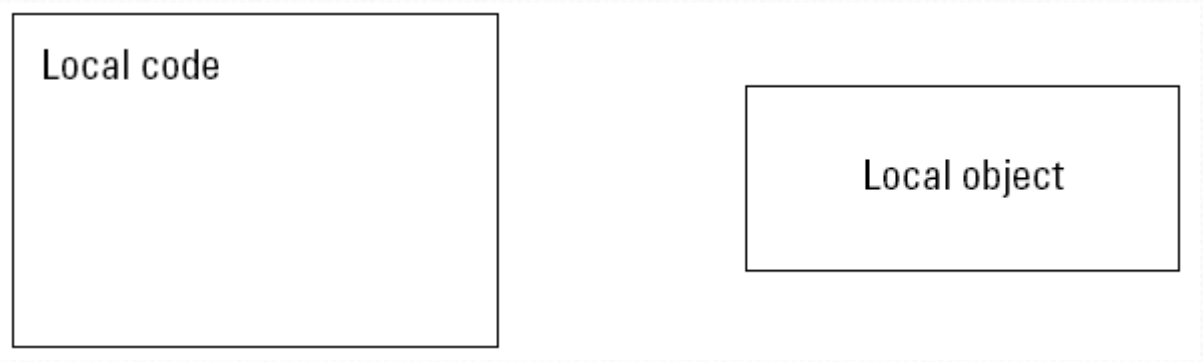
- Proxy possui referencia a imagem no disco
- Proxy possui informação sobre a dimensão da imagem
  - Responde o tamanho da imagem sem precisar instanciar

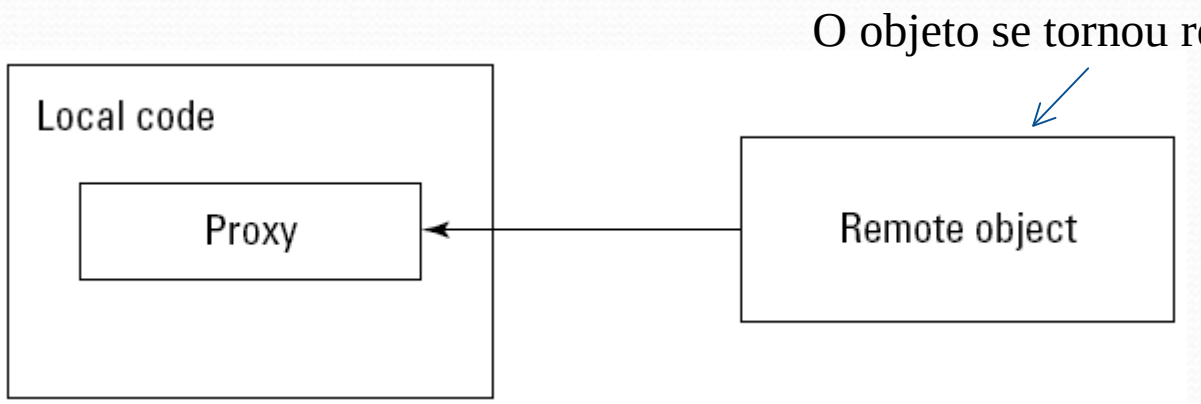


Exemplo – Editor de documentos



Exemplo: Acesso remoto





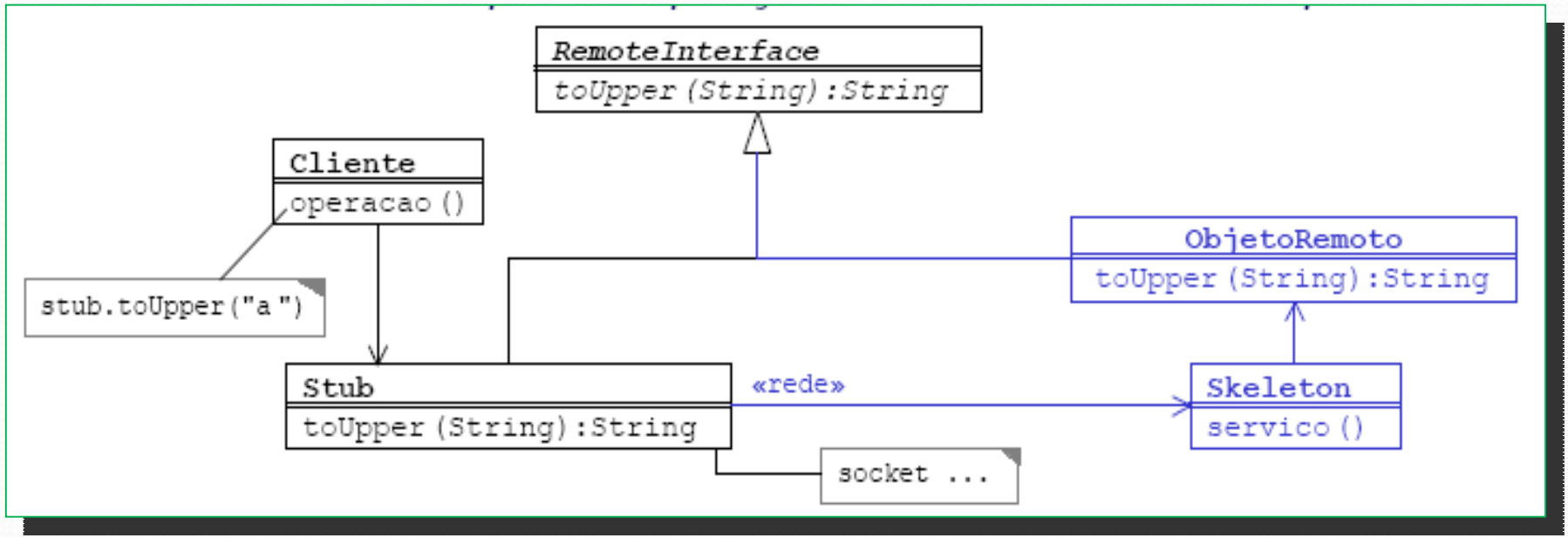
Proxy: permite tratar o objeto remoto como se ele fosse local

Aplicabilidade

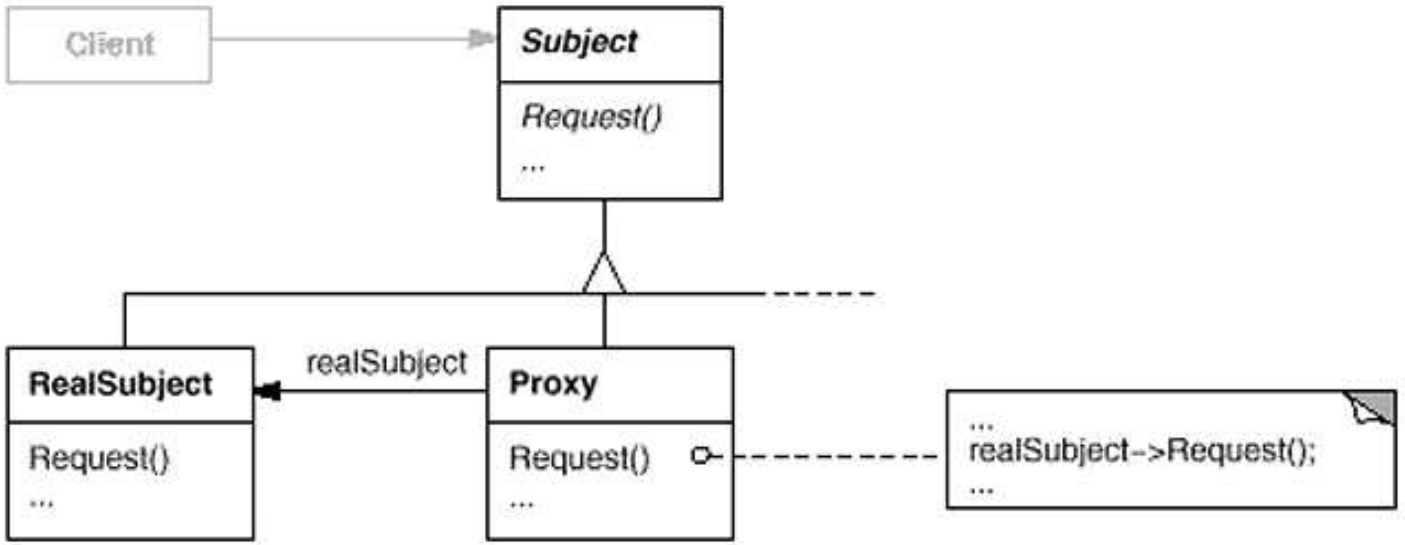
- Quando precisar de um acesso mais versátil a um objeto do que um ponteiro:
  - Remote proxy (acesso remoto);
  - Virtual proxy (exemplo da imagem);
  - Protection proxy (controla acesso)
    - Controla permissões de acesso

Aplicabilidade

- Mais comum em objetos distribuídos
- O Stub é proxy do cliente para o objeto remoto
- O Skeleton é parte do Proxy: cliente remoto chamado pelo Stub



Estrutura


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

Participantes

- Proxy (image Proxy)
  - Matem referência ao objeto real
  - Possui interface idêntica ao objeto real para ser substituído pelo mesmo
  - Controla o acesso ao objeto real
    - Pode ser responsável por criar ou excluir este objeto
- Subject (Graphic)
  - Define uma interface comum para o Proxy e o objeto real
- Realsubject (Image)
  - Define o real objeto que o Proxy representa

Conseqüências

- Permite maior eficiência com caching no cliente
- Transparência: mesma sintaxe usada na comunicação entre o cliente e sujeito real é usada no proxy
- Permite o tratamento inteligente dos dados no cliente
- Possível impacto na performance
- Transparência nem sempre é 100% (fatores externos como queda da rede podem tornar o proxy inoperante ou desatualizado)

Implementação

```
class Graphic {
public:
    virtual ~Graphic();

    virtual void Draw(const Point& at) = 0;
    virtual void HandleMouse(Event& event) = 0;

    virtual const Point& GetExtent() = 0;

    virtual void Load(istream& from) = 0;
    virtual void Save(ostream& to) = 0;
protected:
    Graphic();
};
```

```
class Image : public Graphic {
public:
    Image(const char* file);  // loads image from a file
    virtual ~Image();

    virtual void Draw(const Point& at);
    virtual void HandleMouse(Event& event);

    virtual const Point& GetExtent();

    virtual void Load(istream& from);
    virtual void Save(ostream& to);
private:
    // ...
};
```

Implementaçã

```
class ImageProxy : public Graphic {
public:
    ImageProxy(const char* imageFile);
    virtual ~ImageProxy();

    virtual void Draw(const Point& at);
    virtual void HandleMouse(Event& event);

    virtual const Point& GetExtent();

    virtual void Load(istream& from);
    virtual void Save(ostream& to);
protected:
    Image* GetImage();
private:
    Image* _image;
    Point _extent;
    char* _fileName;
};
```

```
ImageProxy::ImageProxy (const char* fileName)  {
    _fileName = strdup(fileName);
    _extent = Point::Zero;  // don't know extent yet
    _image = 0;
}

Image* ImageProxy::GetImage() {
    if (_image == 0) {
        _image = new Image(_fileName);
    }
    return _image;
}
```

```
const Point& ImageProxy::GetExtent () {
    if (_extent == Point::Zero) {
        _extent = GetImage()->GetExtent();
    }
    return _extent;
}

void ImageProxy::Draw (const Point& at) {
    GetImage()->Draw(at);
}

void ImageProxy::HandleMouse (Event& event) {
    GetImage()->HandleMouse(event);
}
```