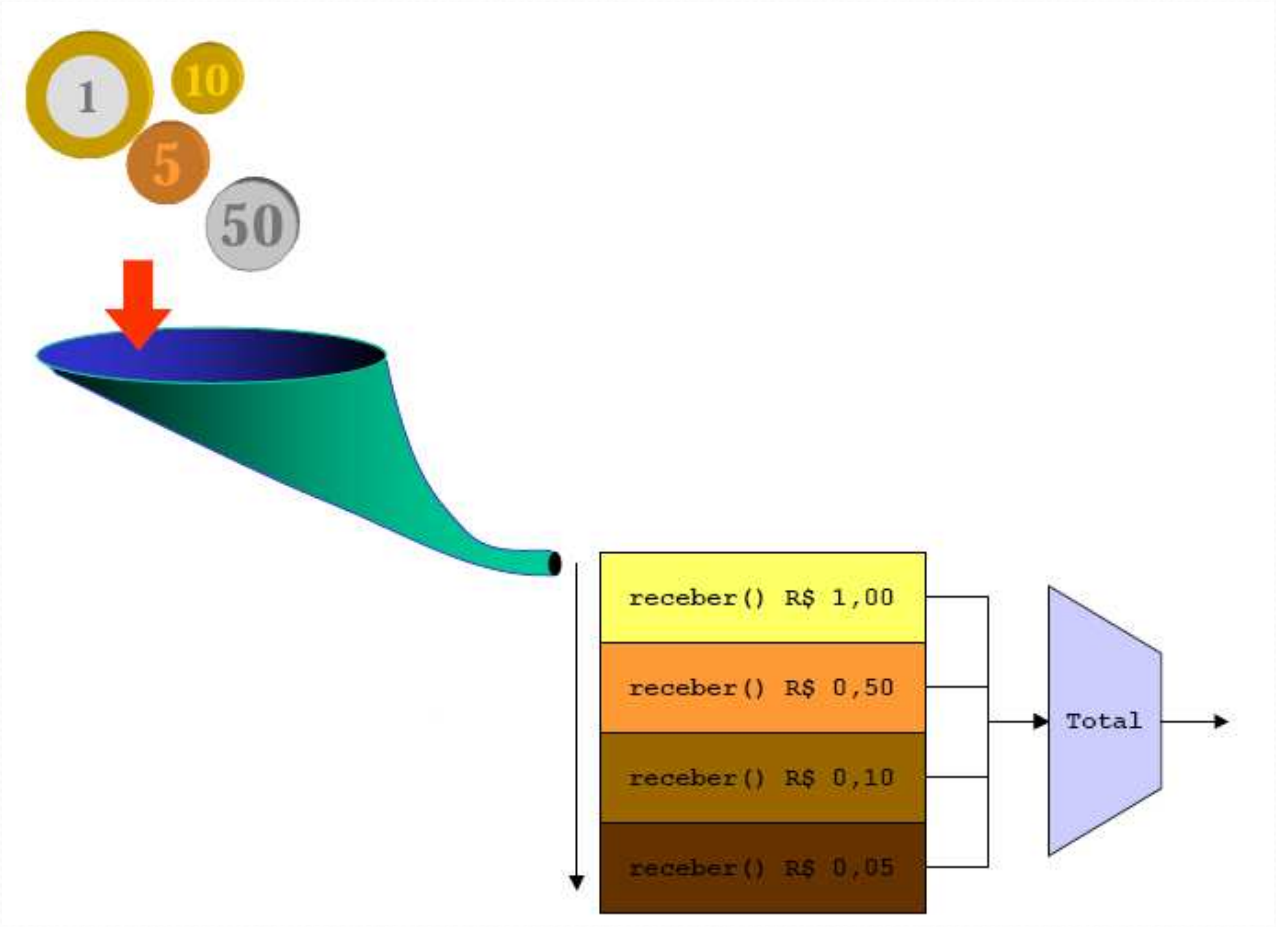
Chain of Responsability 9

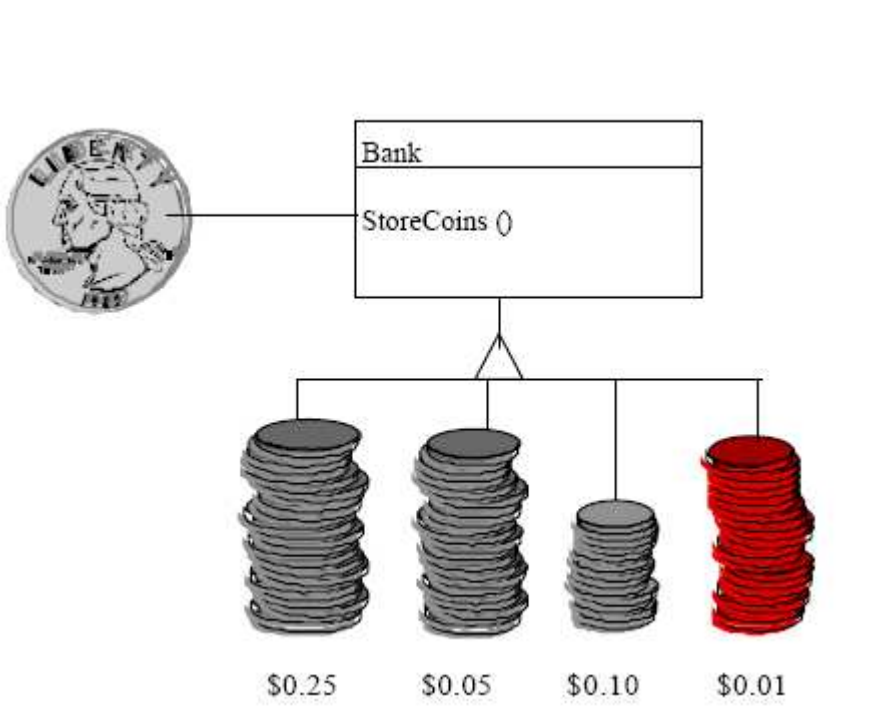
Objetivo:

“Evita acoplar o remetente de uma requisição ao seu destinatário ao dar a mais de um objeto a chance de servir a requisição. Compõe os objetos em cascata e passa a requisição pela corrente até que um objeto a sirva." [GoF]

Analogia


1
10
5
50
receber() R$ 1,00
receber() R$ 0,50
receber() R$ 0,10
receber() R$ 0,05
Total

Analogia

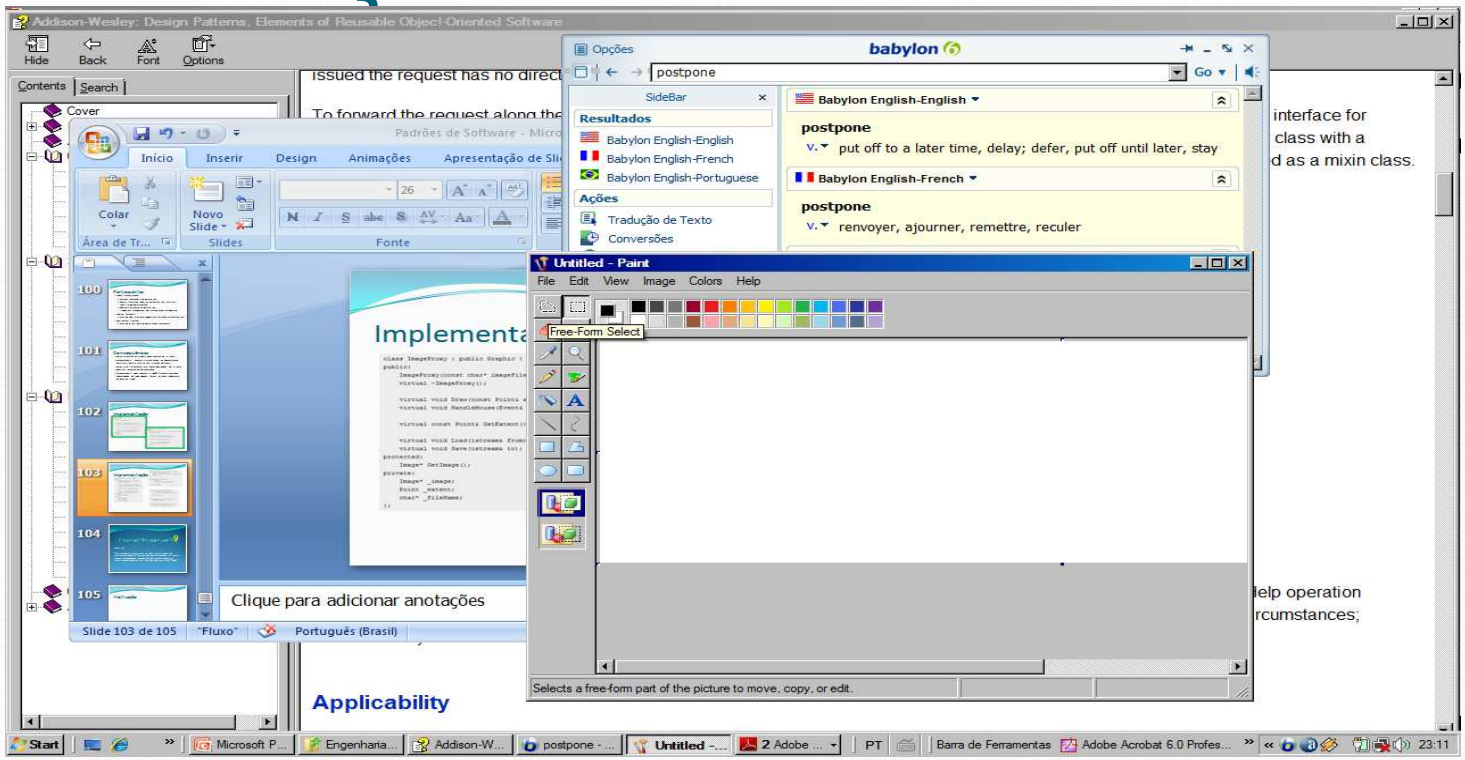


Analogia

Motivação

- Considerem as facilidades de ajuda em uma aplicação de interface gráfica
- Ao selecionar um botão + F1 = poderá ter acesso a informações de ajuda
  - A ajuda depende do objeto selecionado
- E se o objeto (botão)não tiver um arquivo de ajuda relacionado
  - Deve ser mostrada um arquivo de ajuda de um objeto mais abstrato – Ex.: janela da aplicação
- Organização: mais específico → mais geral

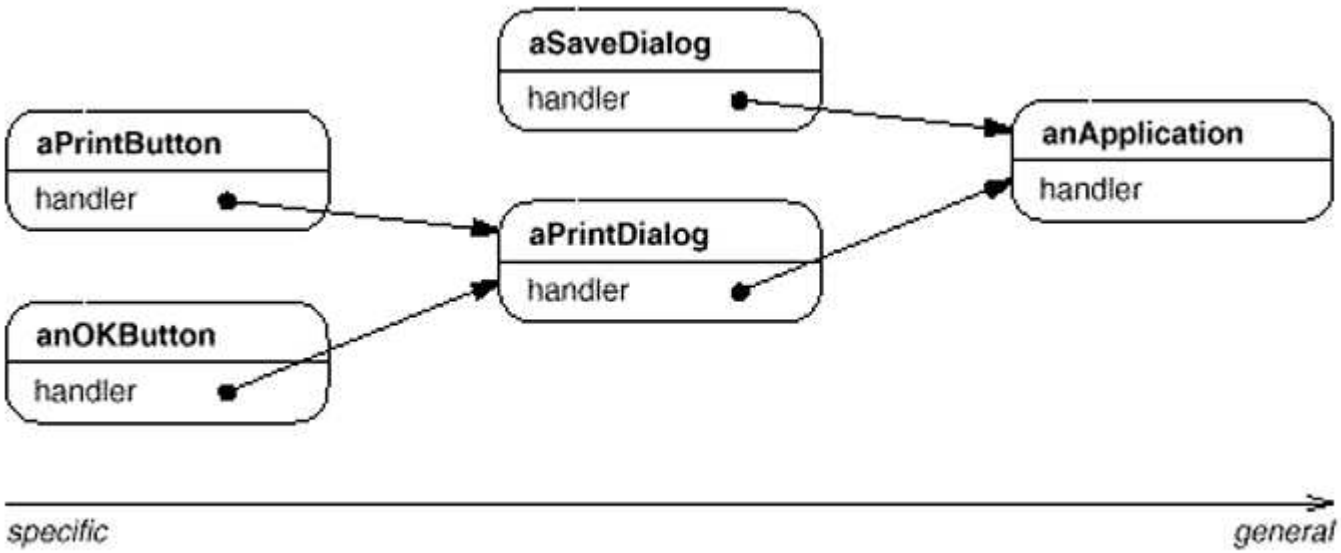
Motivação



- Ajuda de contexto: ao pressionar F1 com o foco em um componente gráfico, qual componente acionará a ajuda?

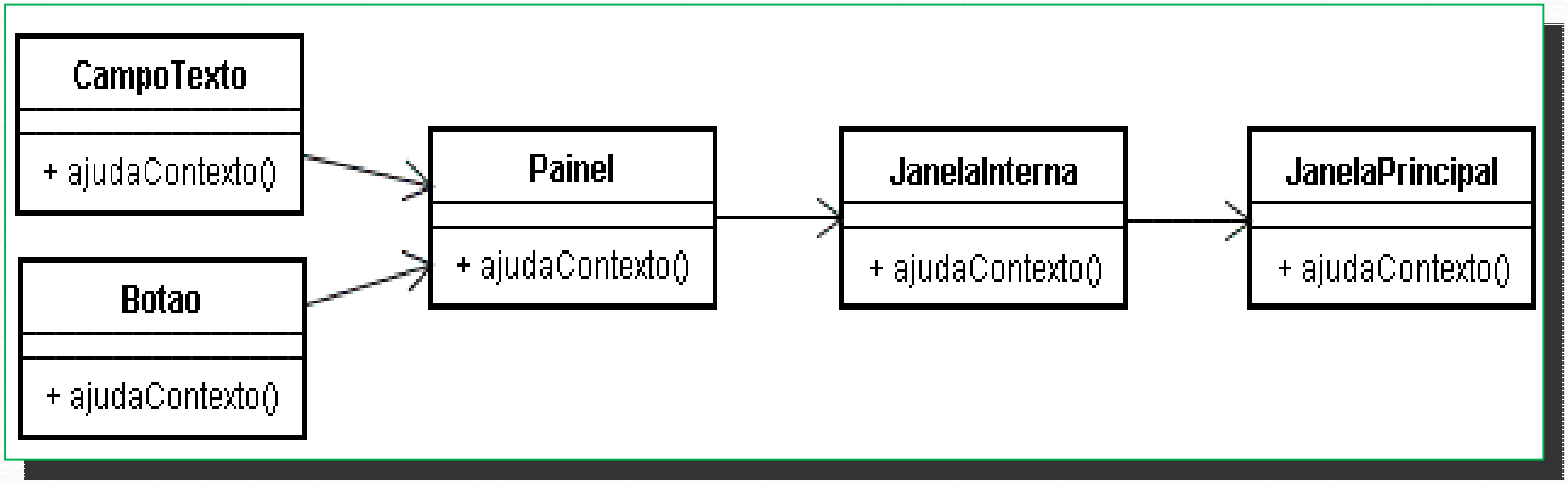
Solução – Chain of Responsability

- Desacopla transmissores e receptores para dar a chance para muitos objetos tratarem a requisição
- A requisição é passada por uma cadeia de objetos até que algum deles possa tratá-la



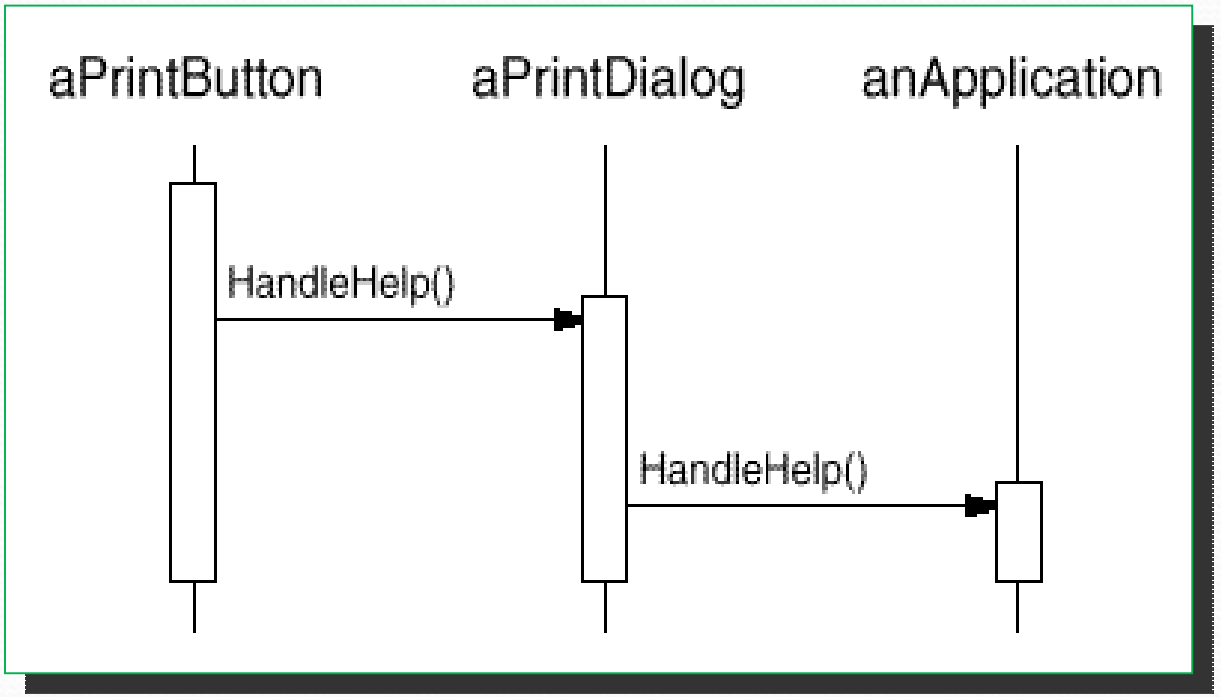
Solução – Chain of Responsability

- Componentes colocados em cadeia na ordem filho -> pai;
- Se não há ajuda de contexto para o filho, ele delega ao pai e assim sucessivamente.



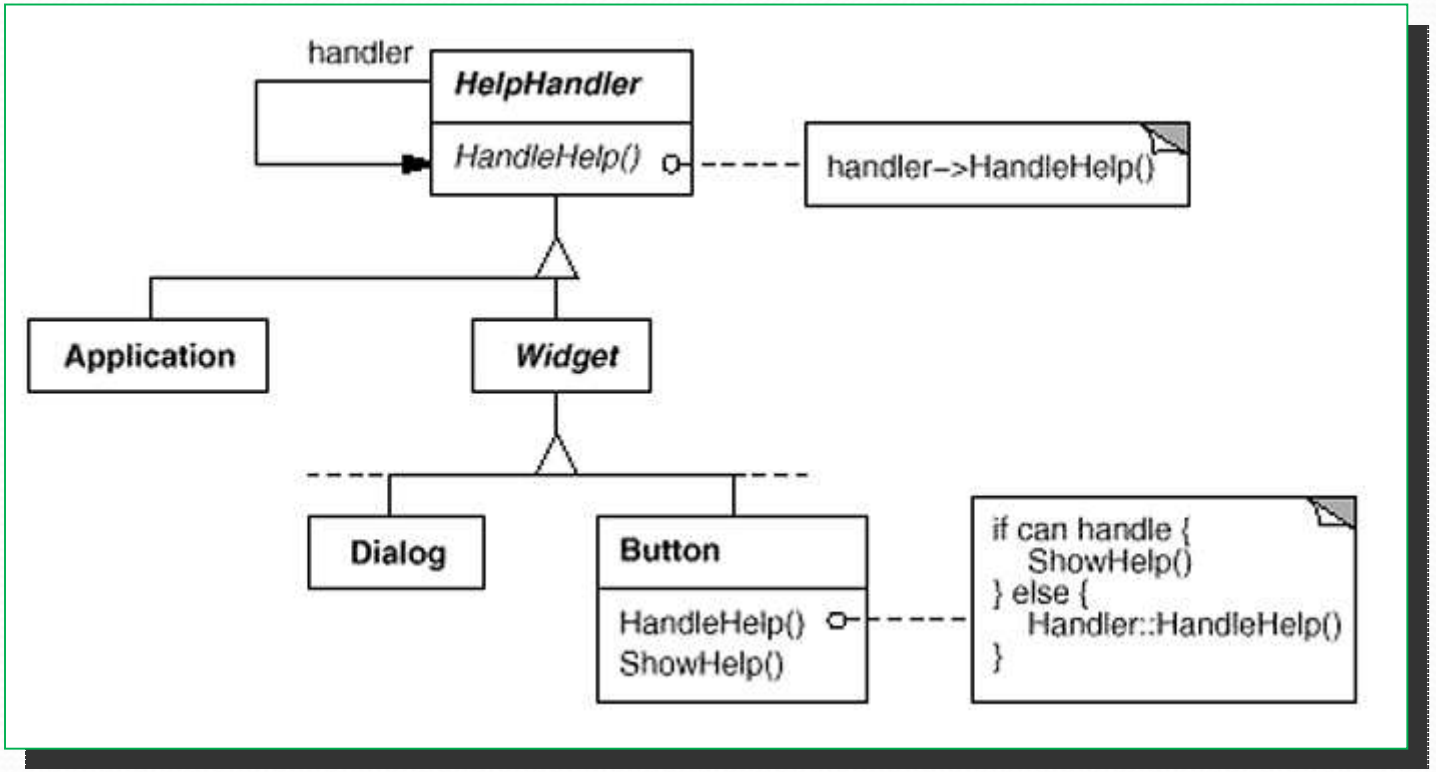
Colaboração

- A aplicação pode tratar ou ignorar a requisição de ajuda



Estrutura

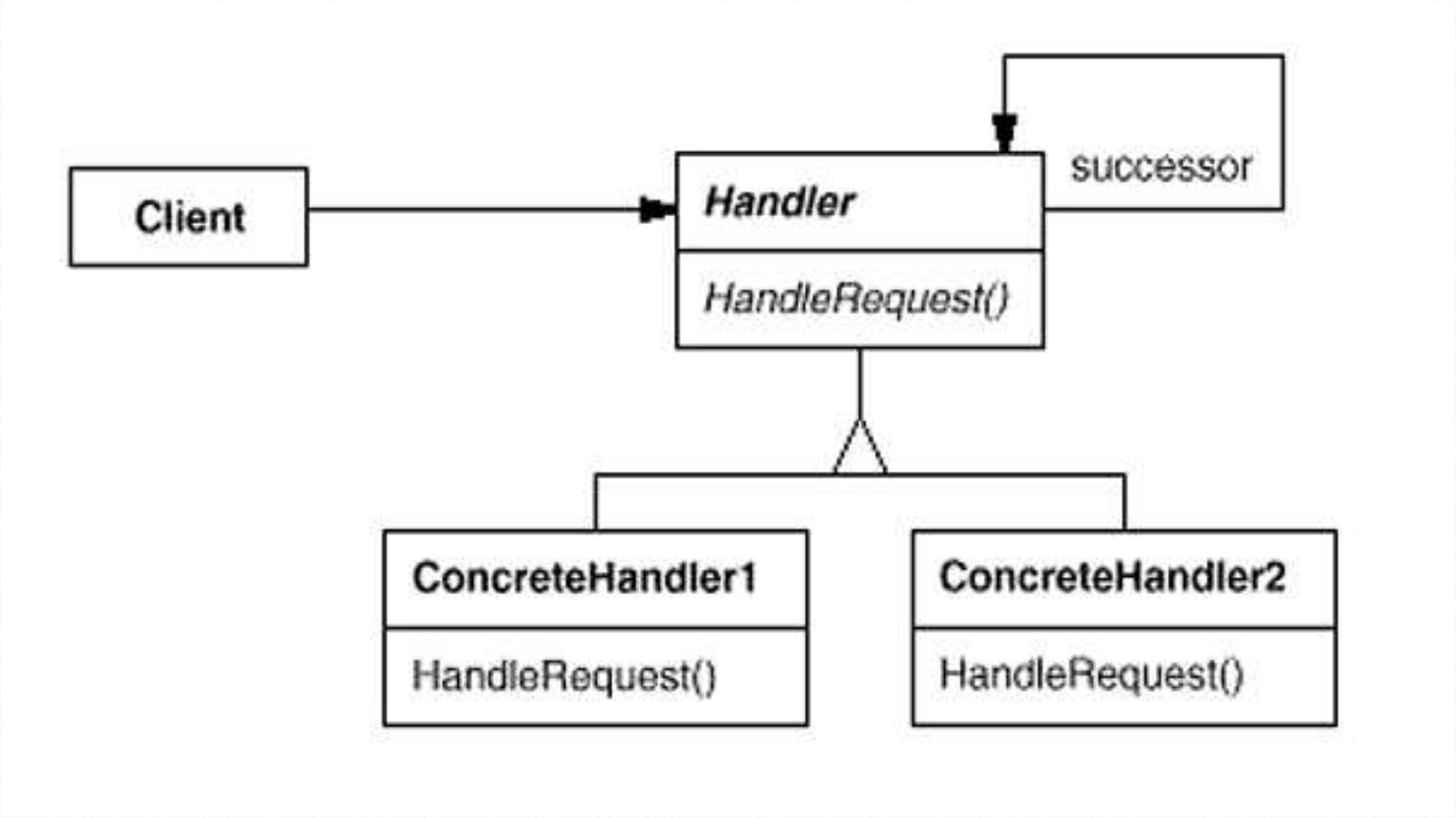
- HelpHandler contém o sucessor
- HandleHelp() pode ser derivado para indicar o sucessor na subclasse



Aplicabilidade

- Mais de um objeto pode tratar a requisição
- Não se conhece qual objeto tratará a requisição
- Quando se deseja atribuir uma requisição para um objeto sem determinar qual deles
  - Escolha dinâmica

Estrutura


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

Participantes

- **Handler** (HelpHandler)
  - Define uma interface para tratar requisições
  - Pode implementar o link para o sucessor
- **ConcreteHandler** (PrintButton, PrintDialog)
  - Trata as requisições para quais é responsável
  - Pode acessar seu sucessor
  - Se puder tratar a requisição, ele faz, caso contrário, encaminha
- **Client**
  - Inicializa a requisição

Conseqüências

- Acoplamento reduzido:
  - Não se sabe a classe ou estrutura interna dos participantes.
  - Pode usar Mediator para desacoplar ainda mais.
- Delegação de responsabilidade:
  - Flexível, em tempo de execução.
- Garantia de resposta:
  - Deve ser uma preocupação do desenvolvedor!
  - Não utilizar identidade de objetos.

Implementaçã

```
typedef int Topic;
const Topic NO_HELP_TOPIC = -1;

class HelpHandler {
public:
    HelpHandler(HelpHandler* = 0, Topic = NO_HELP_TOPIC);
    virtual bool HasHelp();
    virtual void SetHandler(HelpHandler*, Topic);
    virtual void HandleHelp();
private:
    HelpHandler* _successor;
    Topic _topic;
};

HelpHandler::HelpHandler (
    HelpHandler* h, Topic t
) : _successor(h), _topic(t) { }

bool HelpHandler::HasHelp () {
    return _topic != NO_HELP_TOPIC;
}

void HelpHandler::HandleHelp () {
    if (_successor != 0) {
        _successor->HandleHelp();
    }
}
```

```
class Widget : public HelpHandler {
protected:
    Widget(Widget* parent, Topic t = NO_HELP_TOPIC);
private:
    Widget* _parent;
};

Widget::Widget (Widget* w, Topic t) : HelpHandler(w, t) {
    _parent = w;
}
```

```
class Button : public Widget {
public:
    Button(Widget* d, Topic t = NO_HELP_TOPIC);

    virtual void HandleHelp();
    // Widget operations that Button overrides...
};
```

```
Button::Button (Widget* h, Topic t) : Widget(h, t) { }

void Button::HandleHelp () {
    if (HasHelp()) {
        // offer help on the button
    } else {
        HelpHandler::HandleHelp();
    }
}
```

Implementação

```
class Dialog : public Widget {
public:
    Dialog(HelpHandler* h, Topic t = NO_HELP_TOPIC);
    virtual void HandleHelp();

    // Widget operations that Dialog overrides...
    // ...
};

Dialog::Dialog (HelpHandler* h,  Topic t) : Widget(0) {
    SetHandler(h, t);
}

void Dialog::HandleHelp () {
    if (HasHelp()) {
        // offer help on the dialog
    } else {
        HelpHandler::HandleHelp();
    }
}
```

```
class Application : public HelpHandler {
public:
    Application(Topic t) : HelpHandler(0, t) { }

    virtual void HandleHelp();
    // application-specific operations...
};

void Application::HandleHelp () {
    // show a list of help topics
}
```

Implementação

```
const Topic PRINT_TOPIC = 1;
const Topic PAPER_ORIENTATION_TOPIC = 2;
const Topic APPLICATION_TOPIC = 3;

Application* application = new Application(APPLICATION_TOPIC);
Dialog* dialog = new Dialog(application, PRINT_TOPIC);
Button* button = new Button(dialog, PAPER_ORIENTATION_TOPIC);
```

```
button->HandleHelp();
```

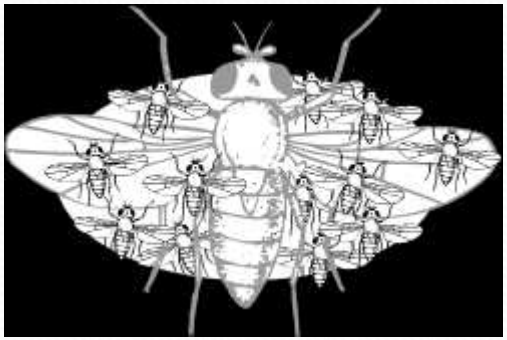
Exercício

- Escreva uma aplicação em Java que receba um texto ou arquivo de texto da linha de comando
- O texto deve ser lido e estatísticas devem ser impressas sobre: a) o número de espaços encontrados, b) o número de letras 'a' e c) o número de pontos.
- Use Chain of Responsibility e faça com que cada tipo de caractere seja tratado por um elo diferente da corrente.

Flyweight (Peso Mosca)

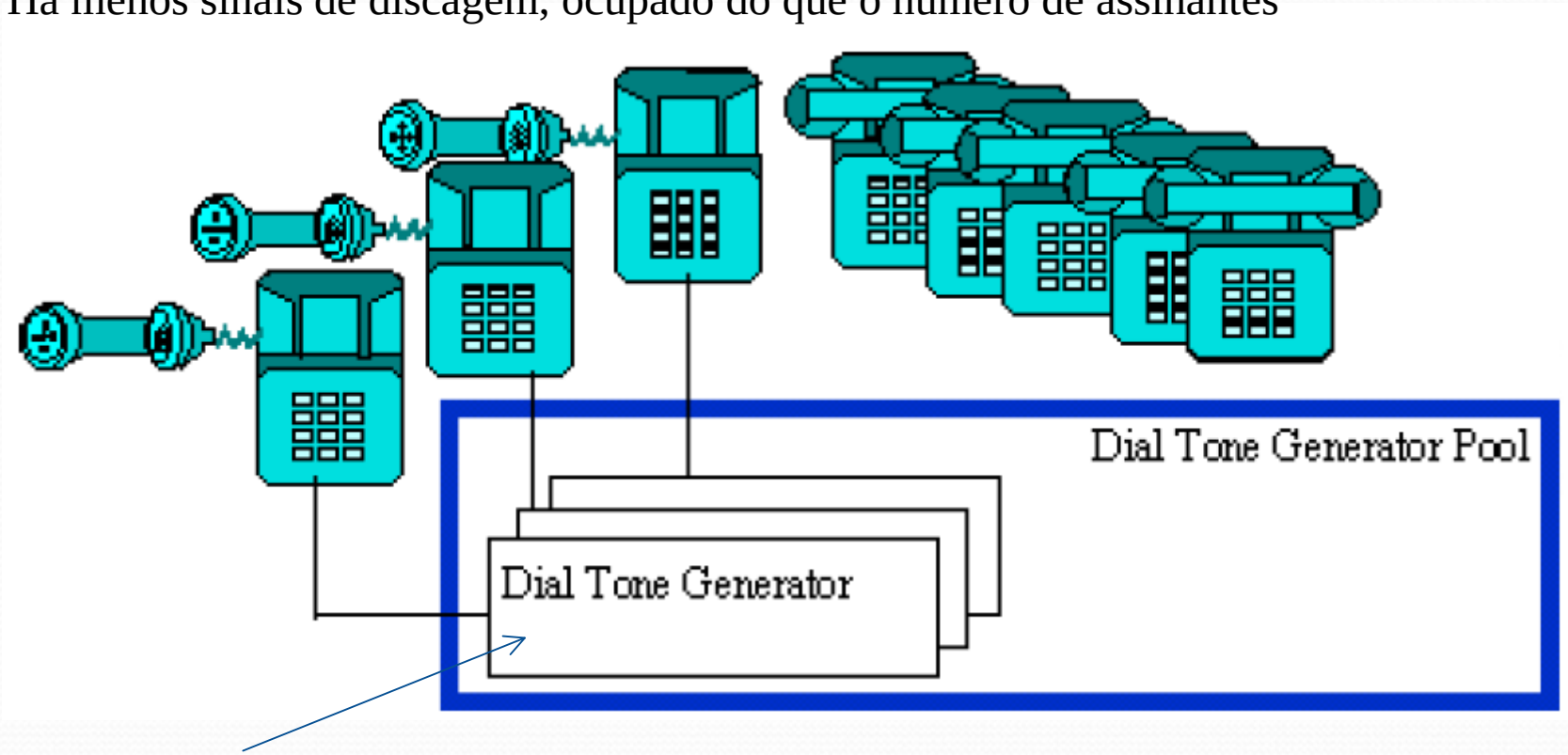
Objetivo:

"Usar compartilhamento para suportar grandes quantidades de objetos refinados eficientemente." [GoF]



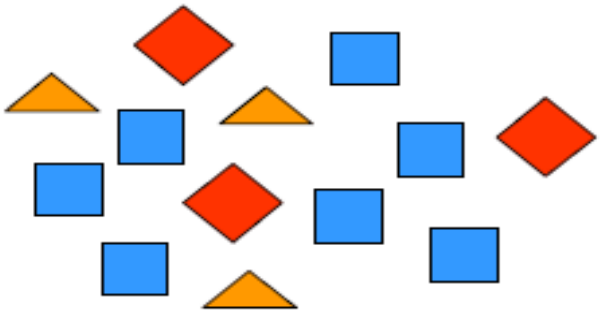
Analogia

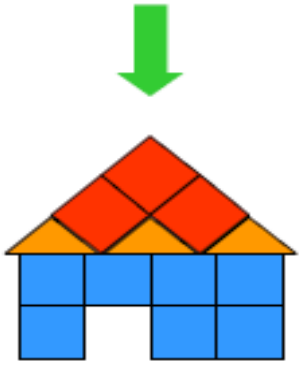
Há menos sinais de discagem, ocupado do que o número de assinantes

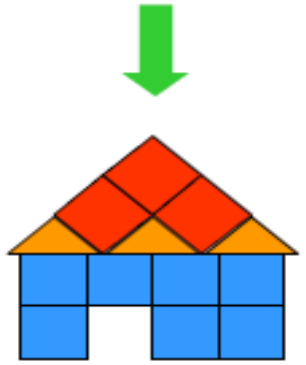


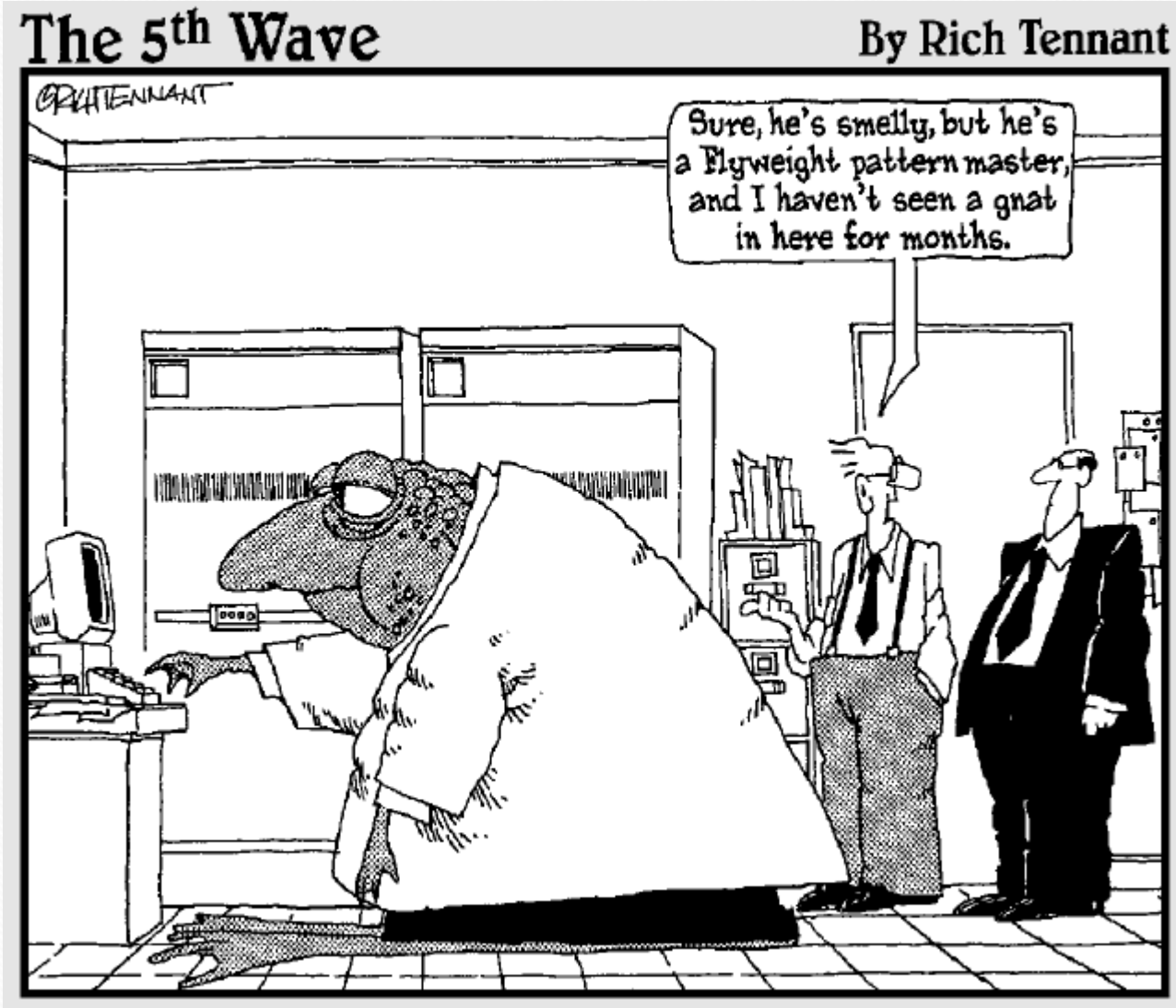
Compartilhamento de tons

Analogia





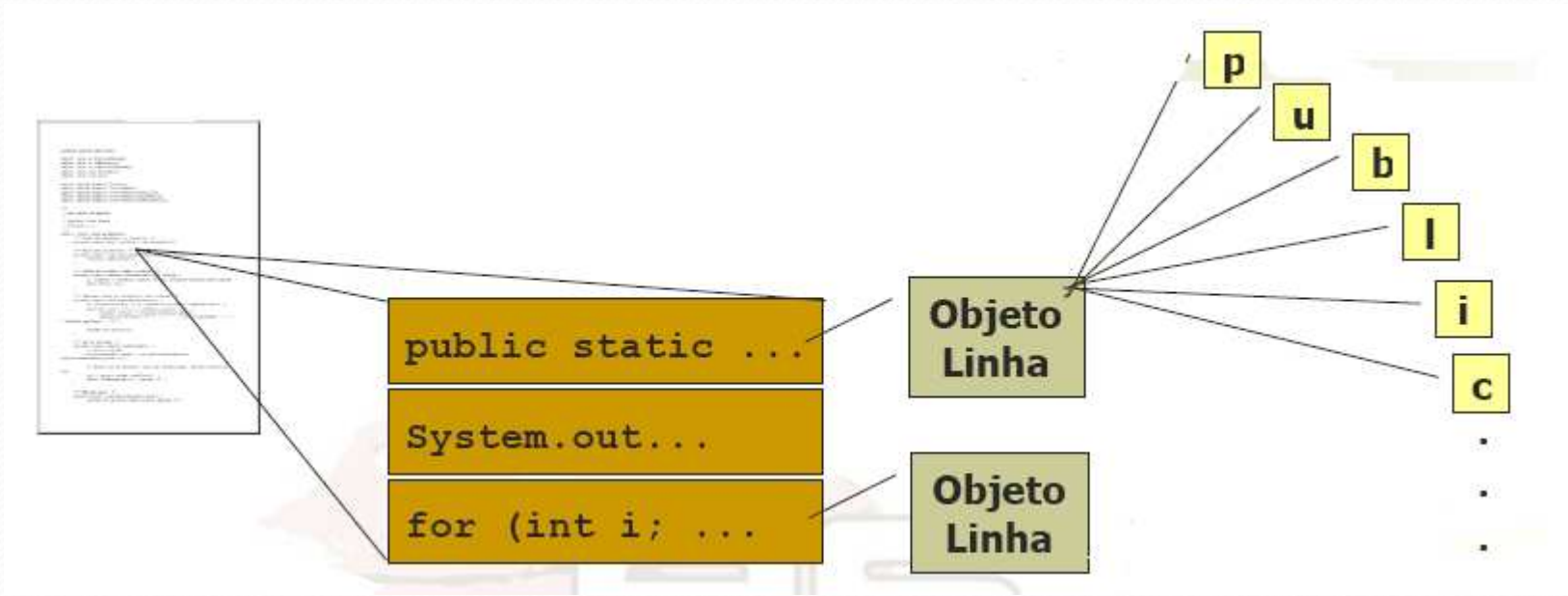



The 5th Wave
By Rich Tennant
RICH TENNANT
Sure, he's smelly, but he's a Flyweight pattern master, and I haven't seen a gnat in here for months.

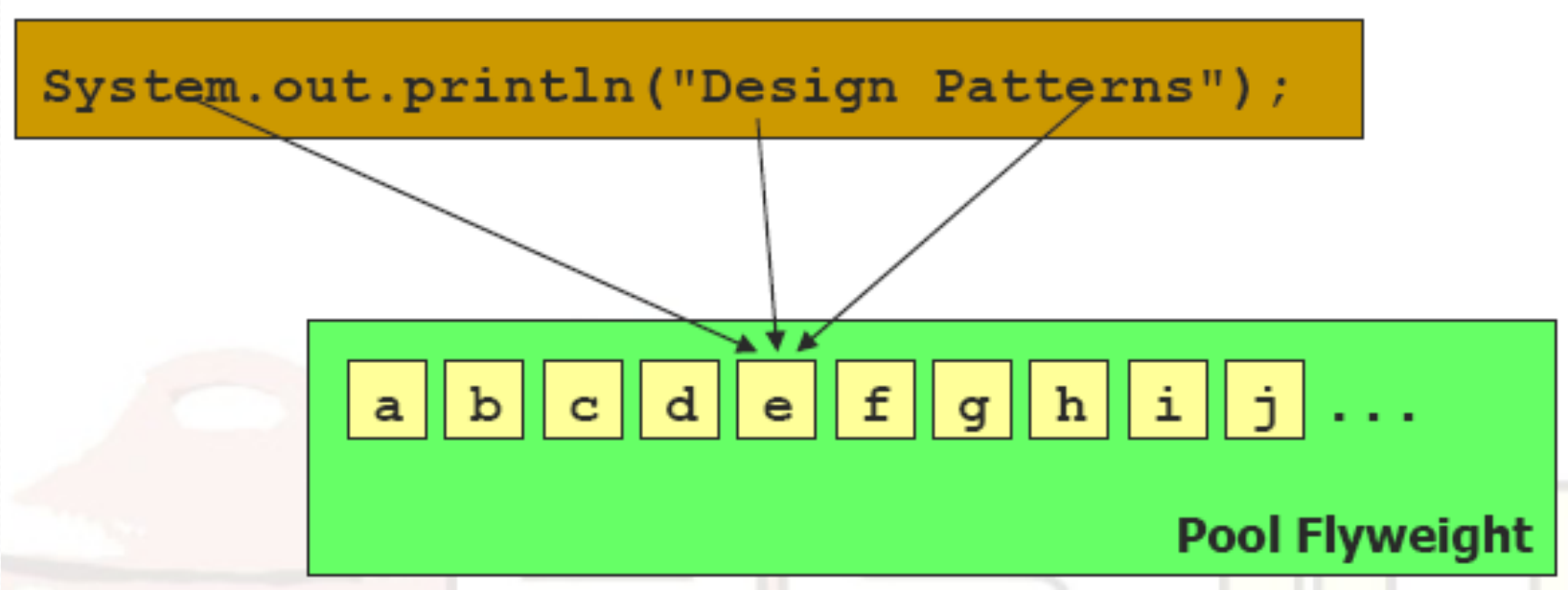
Motivação

- Muitas aplicações podem se beneficiar do uso de OO
  - O uso deve ser cauteloso
- Por exemplo:
  - Editores de Texto OO poderiam tratar cada caracter como sendo um objeto
  - Tornaria a aplicação extensível (novos símbolos)
  - A implementação é inviável
    - Necessita de muito recurso
    - Problemas com textos muito extensos

Motivação


public static ...
System.out...
for (int i; ...
Objeto Linha
Objeto Linha
p
u
b
l
i
c

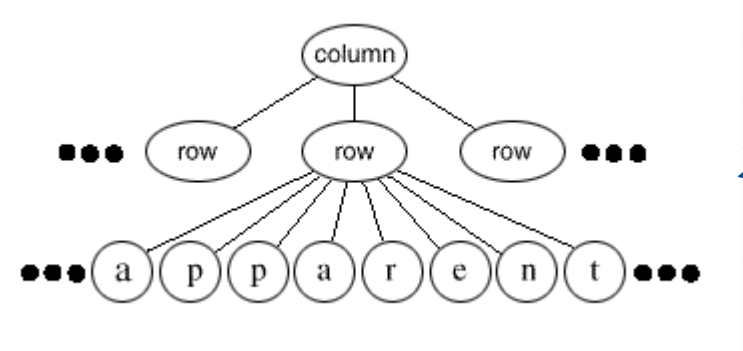
Solução - Flyweight



Solução - Flyweight

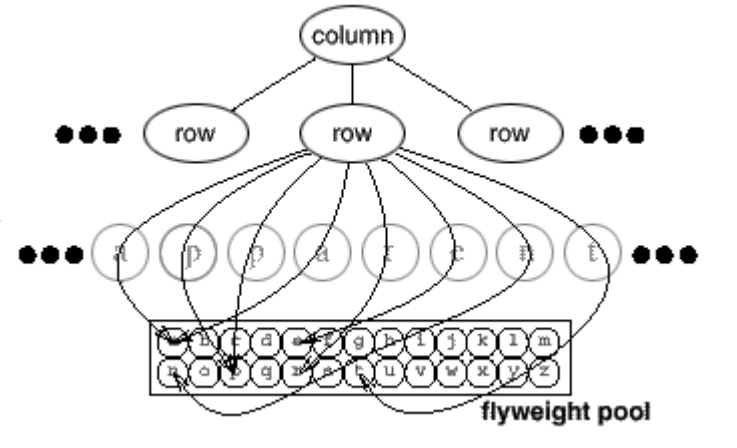
- Descreve como compartilhar objetos
  - Permite o uso sem custos proibitivos
- O conceito chave é: Estado Intrínseco X Estado Extrínseco
  - **Estado Intrínseco**: é armazenado na instância do Flyweight , portanto é **compartilhado** (código do caracter)
  - **Estado Extrinseco**: São informações dependentes do contexto, portanto **não é compartilhado**. Os clientes enviam o estado extrínseco para o objeto compartilhado (detalhes de formatação e posição)

Solução - Flyweight



Sem Flyweight

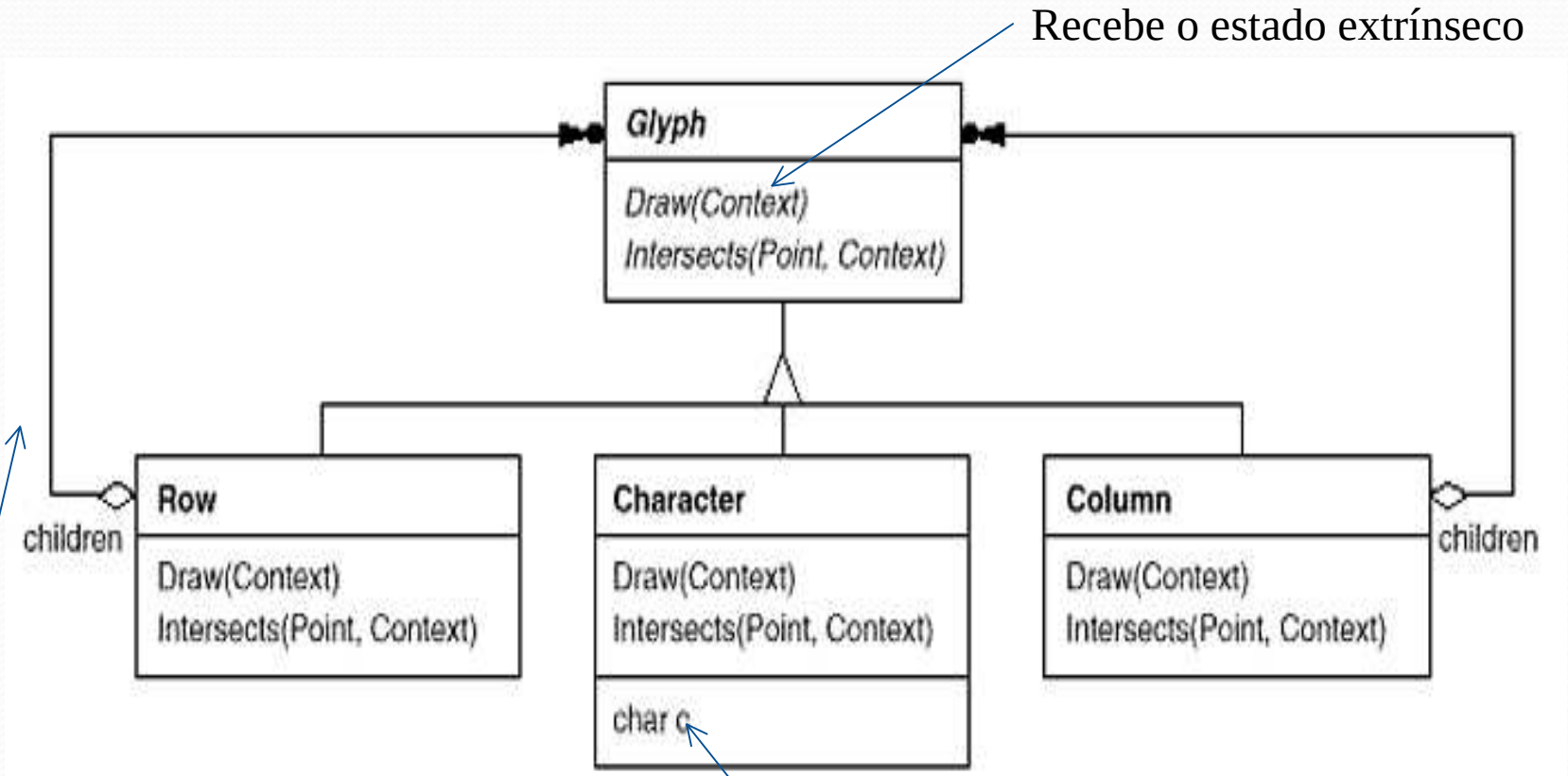
Com Flyweight



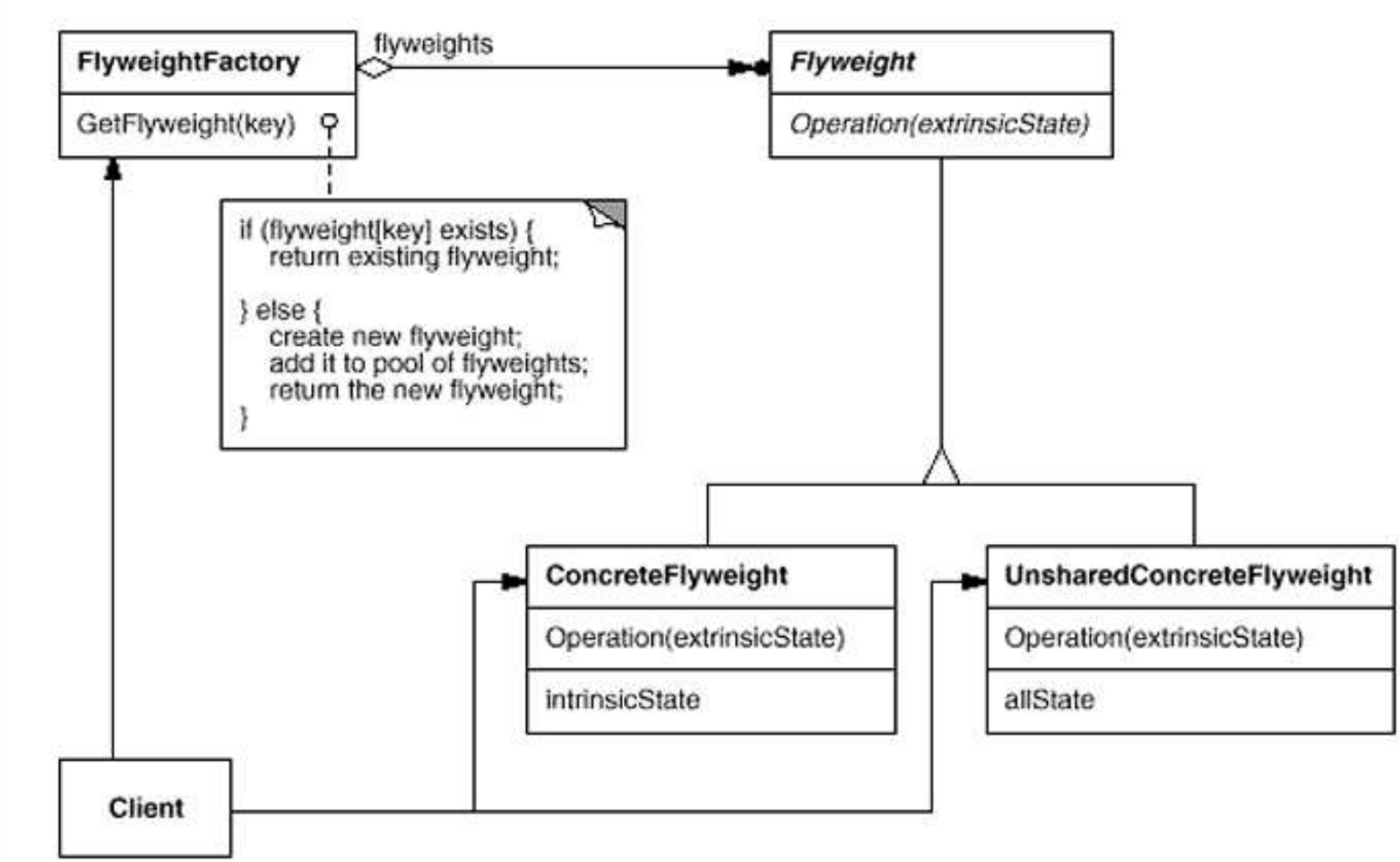
Aplicabilidade

- Uma aplicação usa um grande número de objetos;
- Os custos de armazenamento são altos por causa do grande número de objetos;
- A maioria dos estados podem se tornar extrínsecos;
- Muitos grupos de objetos podem ser substituídos por relativamente poucos compartilhados;
- A aplicação não depende da identidade dos objetos
  - Testes de igualdade sempre retornam verdadeiro

Estrutura


Recebe o estado extrínseco
Glyph
Draw(Context)
Intersects(Point, Context)
children
Row
Draw(Context)
Intersects(Point, Context)
Character
Draw(Context)
Intersects(Point, Context)
char c
Column
Draw(Context)
Intersects(Point, Context)
children
Passa a localização para desenho aos filhos
Código do caractere

Estrutura



A instanciação direta deve ser evitada – garantia de compartilhamento com o Factory

Participantes

- FlyWeight(Peso-Mosca) (Glyph):
  - Declara a interface através da qual objetos Flyweight podem receber e atuar sobre estados extrínsecos.
- ConcreteFlyWeight (Character):
  - Implementa a interface de Flyweight e adiciona armazenamento para estados intrínsecos, se houver. O ConcreteFlyWeight deve ser compartilhavel.
- UnsharedConcreteFlyWeight (Row, Colum):
  - Nem todas as subclasses de Flyweight precisam ser compartilhadas, como as instâncias de linha e coluna.
- FlyWeightFactory:
  - Cria e gerência instâncias de Flyweight .
- Client:
  - Mantém as referências para os objetos Flyweight e computa e armazena o estado extrínseco dos objetos Peso-Mosca.

Conseqüências

- Os **Pesos-Mosca** podem introduzir custo de tempo com a transferência e computação de estados extrínsecos. Contudo, tais custos são compensados com a economia de espaço
- A eficiência do Flyweight deve-se:
  - A redução do número total de instâncias, pelo compartilhamento;
    - Quanto maior a quantidade de objetos compartilhados, mais se poupa recursos
  - A quantidade de estados intrínsecos por objeto;
  - O estado extrínseco é apenas computado ou armazenado.

Implementação

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

    virtual void Draw(Window*, GlyphContext&);

    virtual void SetFont(Font*, GlyphContext&);
    virtual Font* GetFont(GlyphContext&);

    virtual void First(GlyphContext&);
    virtual void Next(GlyphContext&);
    virtual bool IsDone(GlyphContext&);
    virtual Glyph* Current(GlyphContext&);

    virtual void Insert(Glyph*, GlyphContext&);
    virtual void Remove(GlyphContext&);
protected:
    Glyph();
};
```

```
class Character : public Glyph {
public:
    Character(char);

    virtual void Draw(Window*, GlyphContext&);
private:
    char _charcode;
};
```

```
Character* GlyphFactory::CreateCharacter (char c) {
    if (!_character[c]) {
        _character[c] = new Character(c);
    }

    return _character[c];
}
```

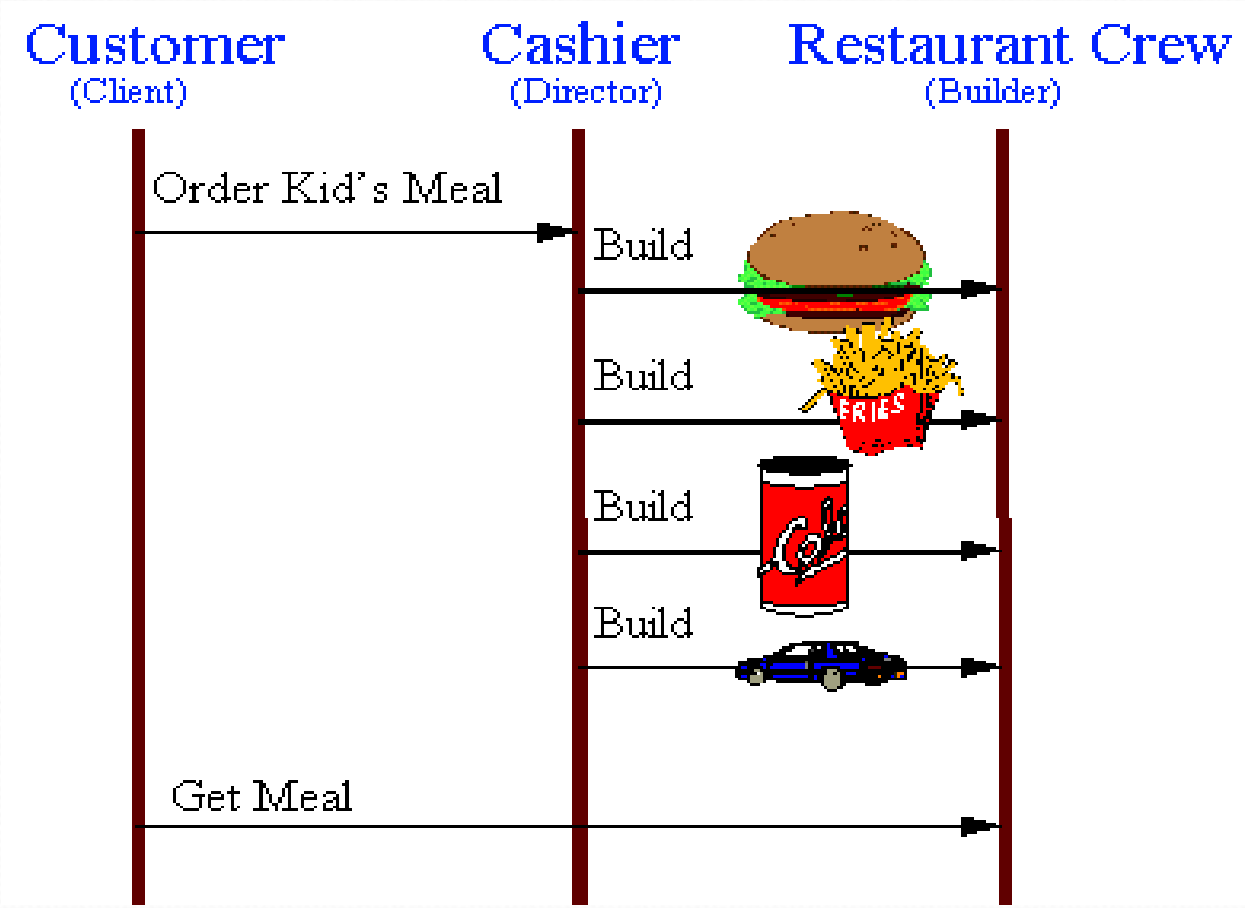
```
const int NCHARCODES = 128;

class GlyphFactory {
public:
    GlyphFactory();
    virtual ~GlyphFactory();

    virtual Character* CreateCharacter(char);
    virtual Row* CreateRow();
    virtual Column* CreateColumn();
    // ...
private:
    Character* _character[NCHARCODES];
};
```

Builder

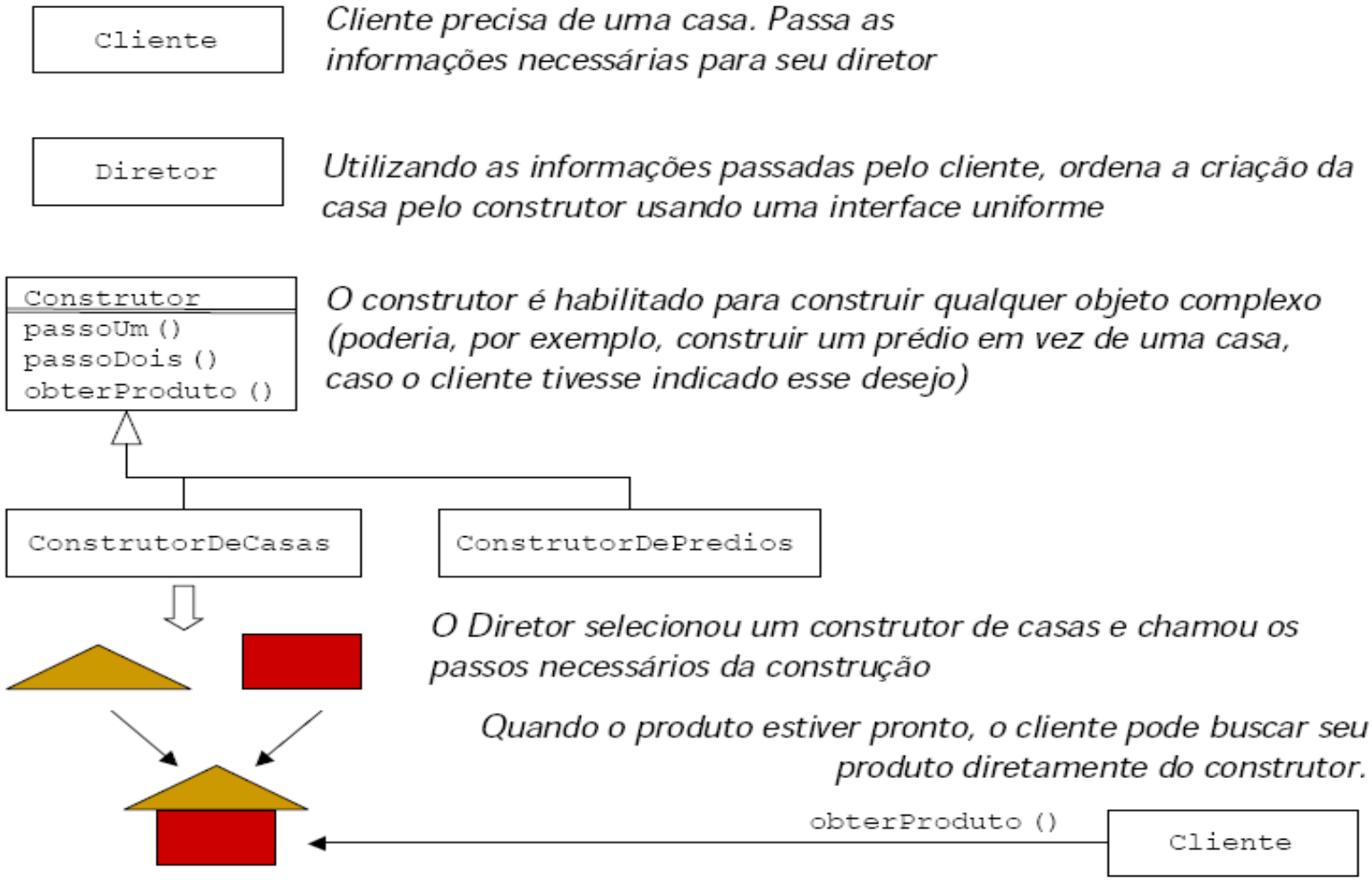
Objetivo:

"Separar a construção de um objeto complexo de sua representação para que o mesmo processo de construção possa criar representações diferentes." [GoF]

Analogia


Customer
(Client)
Cashier
(Director)
Restaurant Crew
(Builder)
Order Kid's Meal
Build
Build
FRIES
Build
Build
Get Meal

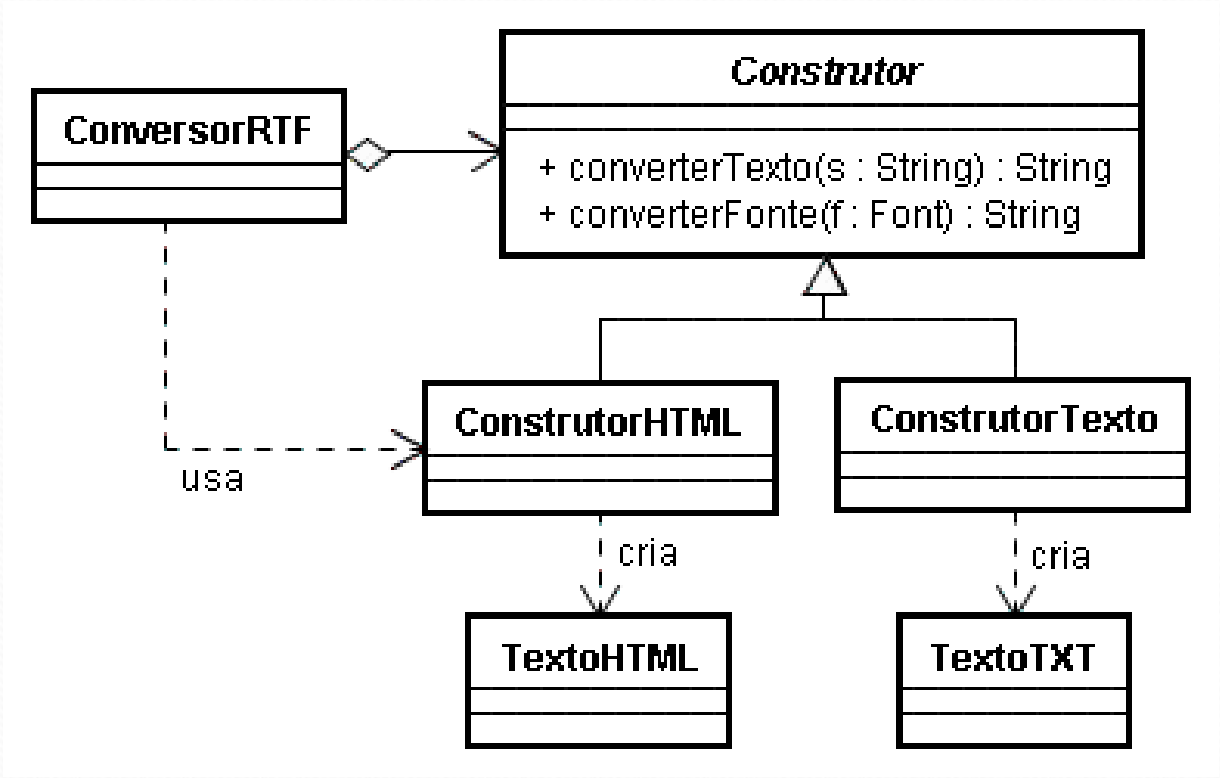
Analogia


Cliente
Cliente precisa de uma casa. Passa as informações necessárias para seu diretor
Diretor
Utilizando as informações passadas pelo cliente, ordena a criação da casa pelo construtor usando uma interface uniforme
Construtor
passoUm ()
passoDois ()
obterProduto ()
O construtor é habilitado para construir qualquer objeto complexo (poderia, por exemplo, construir um prédio em vez de uma casa, caso o cliente tivesse indicado esse desejo)
ConstrutorDeCasas
ConstrutorDePredios
O Diretor selecionou um construtor de casas e chamou os passos necessários da construção
Quando o produto estiver pronto, o cliente pode buscar seu produto diretamente do construtor.
obterProduto ()
Cliente

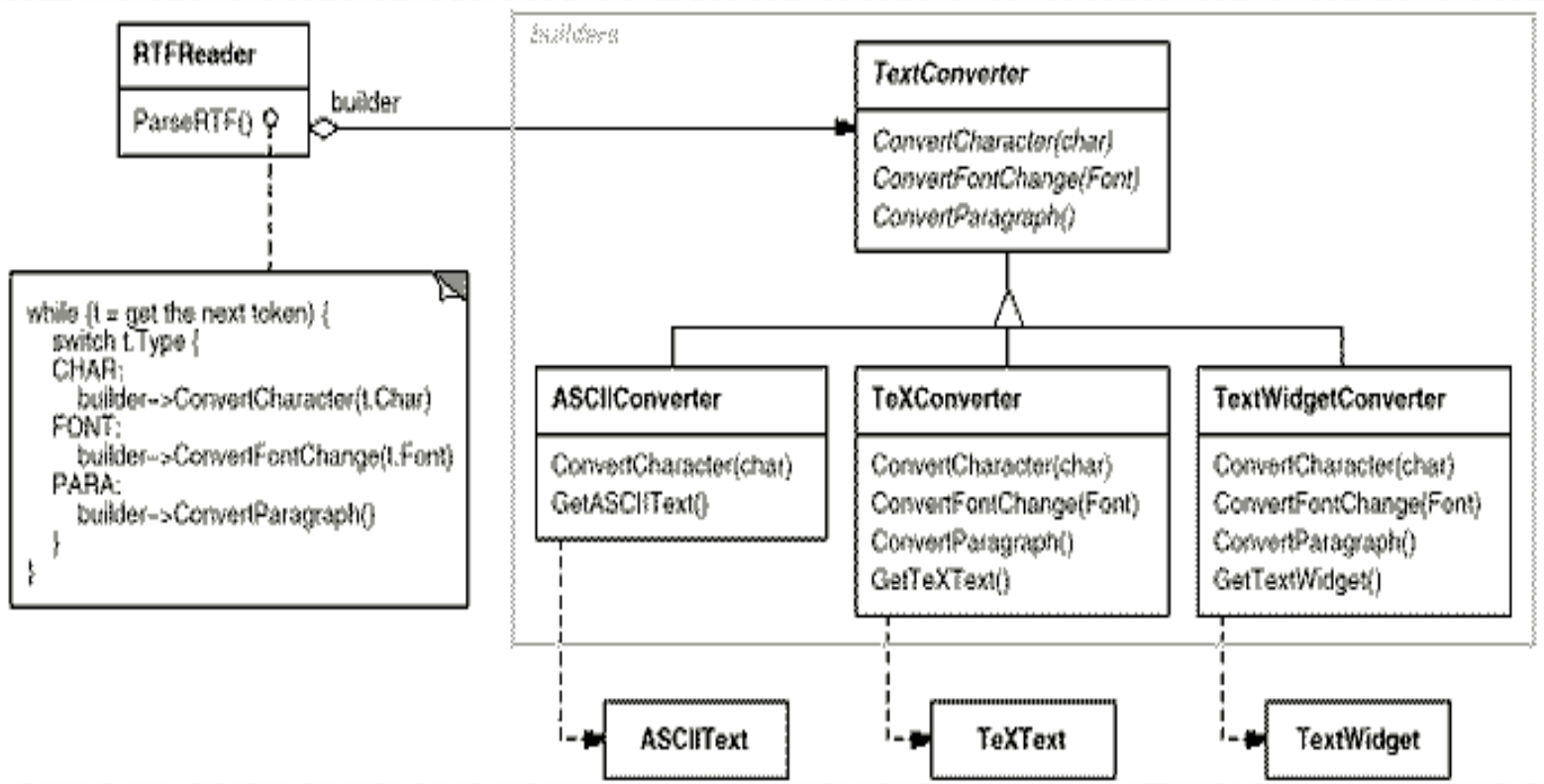
Motivação

- Um leitor de arquivos RTF
  - Conversão para outros formatos (pdf, txt, html...)
- Deve ser fácil adicionar novos conversores sem precisar modificar o leitor de RTF
- Exemplo:
  - Um objeto lê um texto em RTF e converte para HTML como produto final de um processamento;
  - Como fazer para converter para outro formato?

Solução - Builder



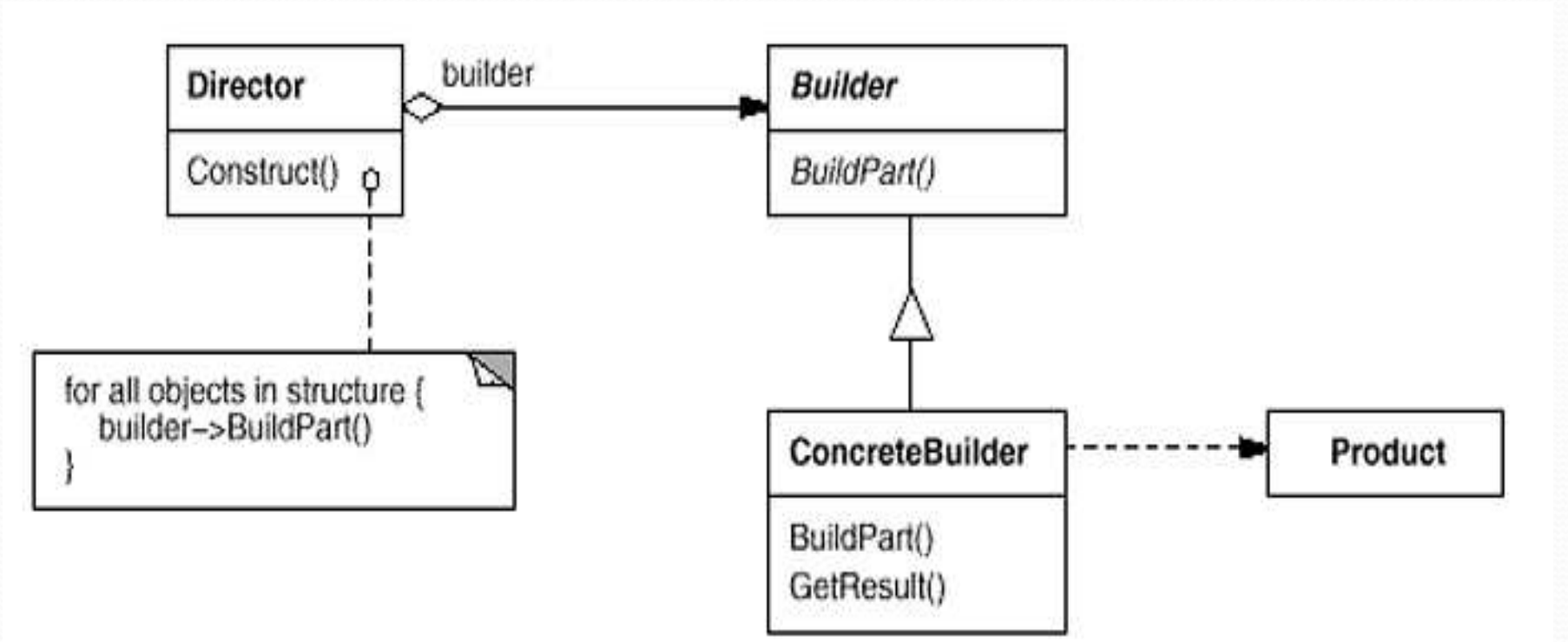
Solução - Builder



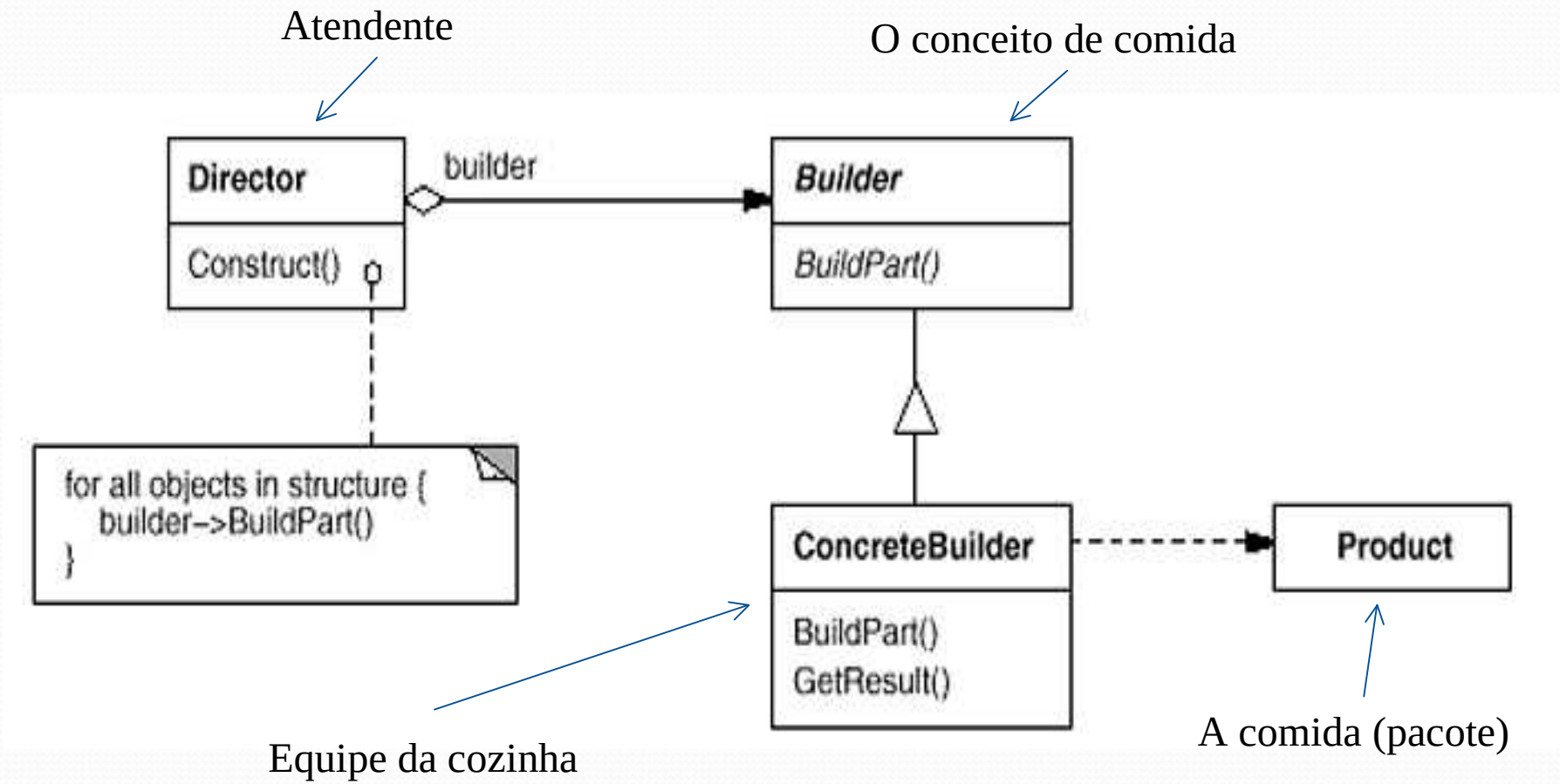
Aplicabilidade

- Quando o algoritmo para criação de objetos complexos tiver que ser independente das partes que compõem o objeto e como elas são unidas;
- Quando o processo de construção tiver que permitir diferentes representações do objeto construído.

Estrutura


Director
builder
Builder
Construct()
BuildPart()
for all objects in structure {
builder->BuildPart()
}
ConcreteBuilder
Product
BuildPart()
GetResult()

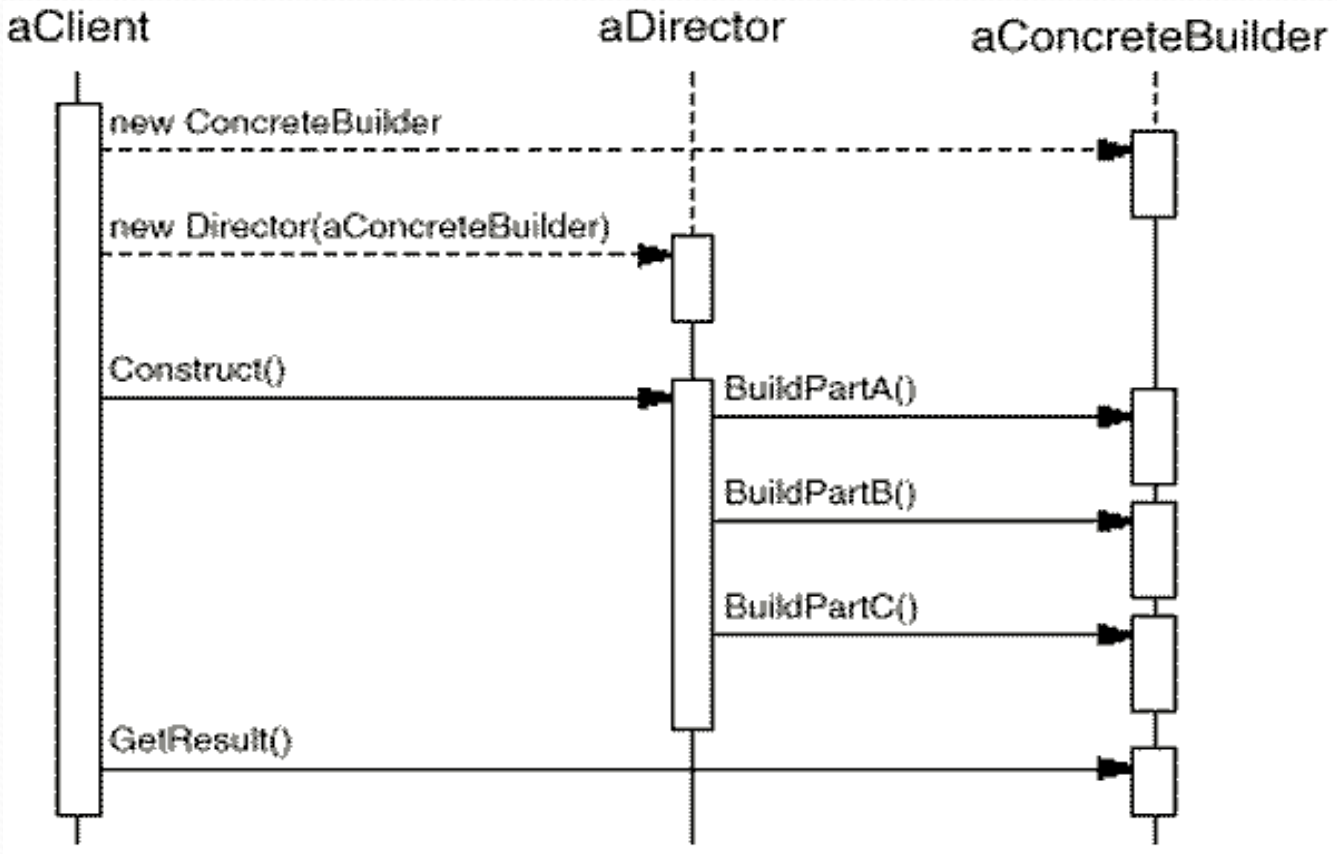
Estrutura


Atendente
O conceito de comida
Director
builder
Construct()
Builder
BuildPart()
for all objects in structure {
builder->BuildPart()
}
ConcreteBuilder
BuildPart()
GetResult()
Product
Equipe da cozinha
A comida (pacote)

Participantes

- Builder (TextConverter)
  - Especifica uma interface para criar partes de um produto
- ConcreteBuilder (ASCIIConverter, TeXConverter, TextWidgetConverter)
  - Constrói as partes do produto
  - Fornece uma interface para retornar o produto
- Director (RTFReader)
  - Constrói um objeto através do Builder
- Product (ASCIIText, TeXText, TextWidget)
  - Representa o objeto depois de construído

Colaboração



Conseqüência

- Permite que varie a representação interna de um produto:
  - Basta construir um novo builder.
- Separa o código de construção:
  - Melhora a modularidade, pois o cliente não precisa saber da representação interna do produto.
- Maior controle do processo de construção:
  - Constrói o produto passo a passo, permitindo o controle de detalhes do processo de construção.

Implementação

```
class MazeBuilder {
public:
    virtual void BuildMaze() { }
    virtual void BuildRoom(int room) { }
    virtual void BuildDoor(int roomFrom, int roomTo) { }

    virtual Maze* GetMaze() { return 0; }
protected:
    MazeBuilder();
};
```

```
Maze* MazeGame::CreateMaze (MazeBuilder& builder) {
    builder.BuildMaze();

    builder.BuildRoom(1);
    builder.BuildRoom(2);
    builder.BuildDoor(1, 2);

    return builder.GetMaze();
}
```

```
Maze* MazeGame::CreateComplexMaze (MazeBuilder& builder) {
    builder.BuildRoom(1);
    // ...
    builder.BuildRoom(1001);

    return builder.GetMaze();
}
```

Implementação

```
class StandardMazeBuilder : public MazeBuilder {
public:
    StandardMazeBuilder();

    virtual void BuildMaze();
    virtual void BuildRoom(int);
    virtual void BuildDoor(int, int);

    virtual Maze* GetMaze();
private:
    Direction CommonWall(Room*, Room*);
    Maze* _currentMaze;
};
```

```
Maze* maze;
MazeGame game;
StandardMazeBuilder builder;

game.CreateMaze(builder);
maze = builder.GetMaze();
```

```
StandardMazeBuilder::StandardMazeBuilder () {
    _currentMaze = 0;
}
```

```
void StandardMazeBuilder::BuildMaze () {
    _currentMaze = new Maze;
}

Maze* StandardMazeBuilder::GetMaze () {
    return _currentMaze;
}
```

```
void StandardMazeBuilder::BuildDoor (int n1, int n2) {
    Room* r1 = _currentMaze->RoomNo(n1);
    Room* r2 = _currentMaze->RoomNo(n2);
    Door* d = new Door(r1, r2);

    r1->SetSide(CommonWall(r1,r2), d);
    r2->SetSide(CommonWall(r2,r1), d);
}
```

```
void StandardMazeBuilder::BuildRoom (int n) {
    if (!_currentMaze->RoomNo(n)) {
        Room* room = new Room(n);
        _currentMaze->AddRoom(room);

        room->SetSide(North, new Wall);
        room->SetSide(South, new Wall);
        room->SetSide(East, new Wall);
        room->SetSide(West, new Wall);
    }
}
```

Implementação

```
class CountingMazeBuilder : public MazeBuilder {
public:
    CountingMazeBuilder();

    virtual void BuildMaze();
    virtual void BuildRoom(int);
    virtual void BuildDoor(int, int);
    virtual void AddWall(int, Direction);

    void GetCounts(int&, int&) const;
private:
    int _doors;
    int _rooms;
};
```

```
int rooms, doors;
MazeGame game;
CountingMazeBuilder builder;

game.CreateMaze(builder);
builder.GetCounts(rooms, doors);

cout << "The maze has "
     << rooms << " rooms and "
     << doors << " doors" << endl;
```

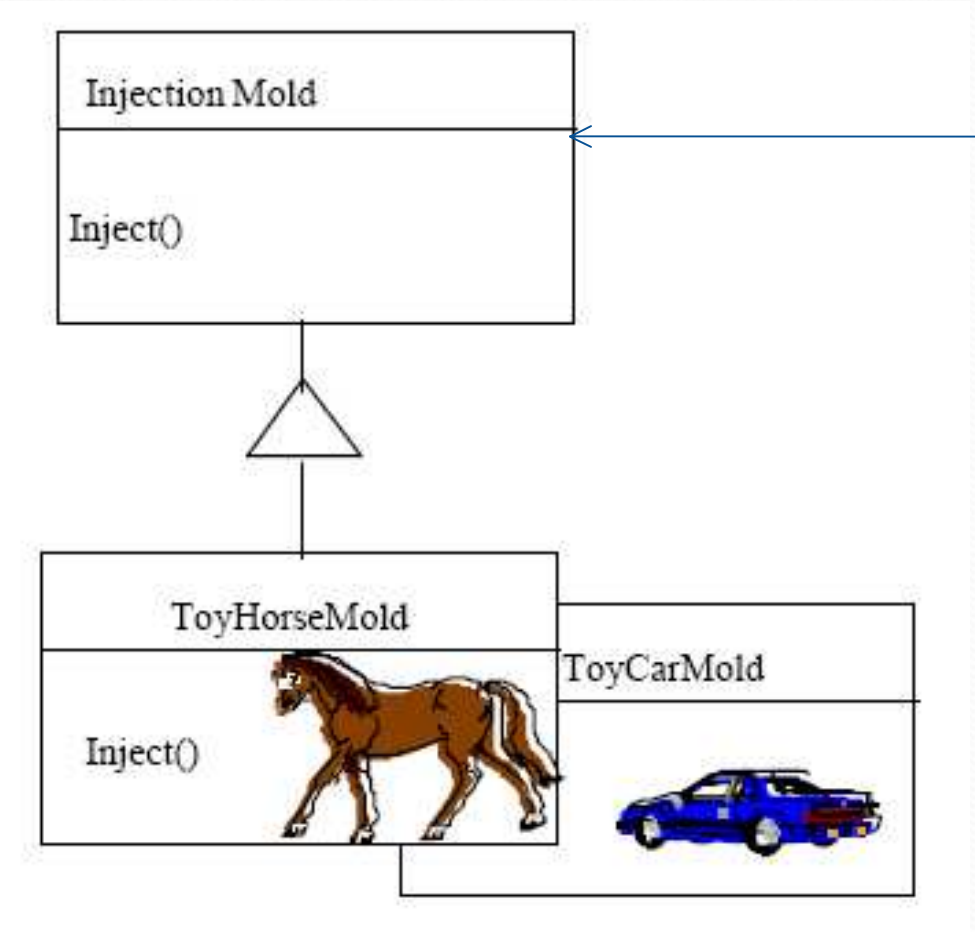
```
CountingMazeBuilder::CountingMazeBuilder () {
    _rooms = _doors = 0;
}

void CountingMazeBuilder::BuildRoom (int) {
    _rooms++;
}

void CountingMazeBuilder::BuildDoor (int, int) {
    _doors++;
}

void CountingMazeBuilder::GetCounts (
    int& rooms, int& doors
) const {
    rooms = _rooms;
    doors = _doors;
}
```

Factory Method

Objetivo:

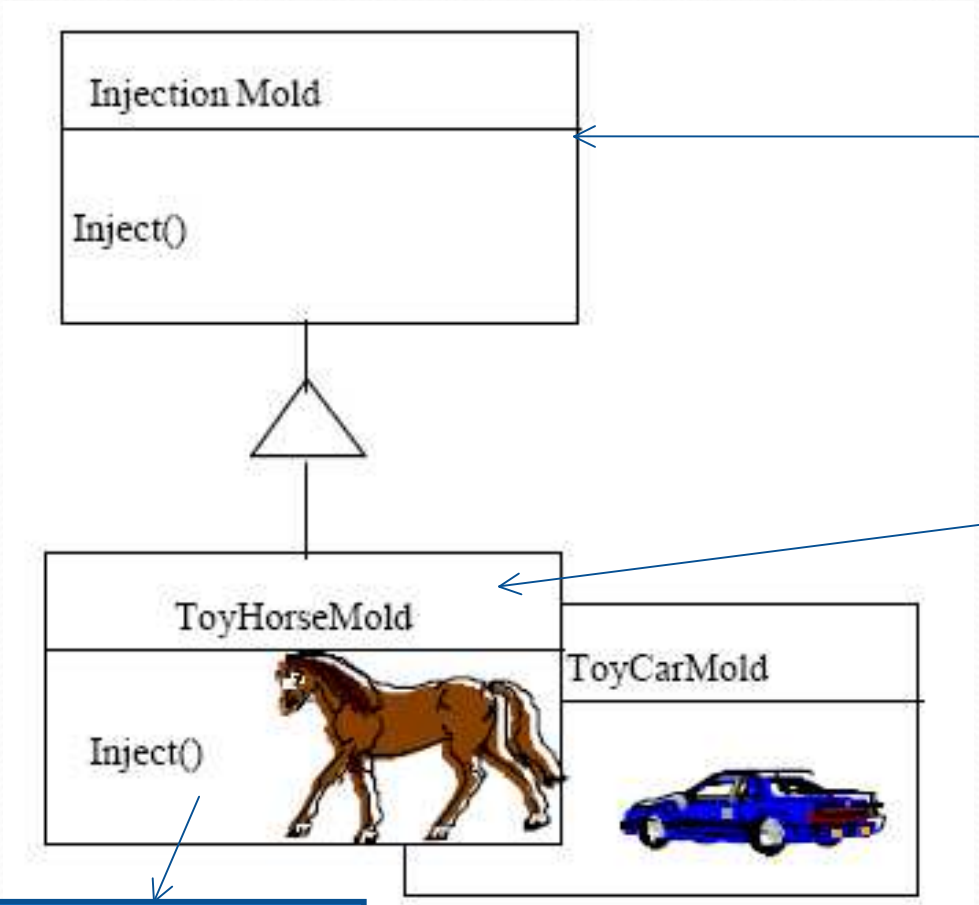
“Definir uma interface para criação de objetos mas deixar as subclasses decidirem qual classe instanciar. Em outras palavras, delega a instanciação para as subclasses.”[GoF]

Analogia

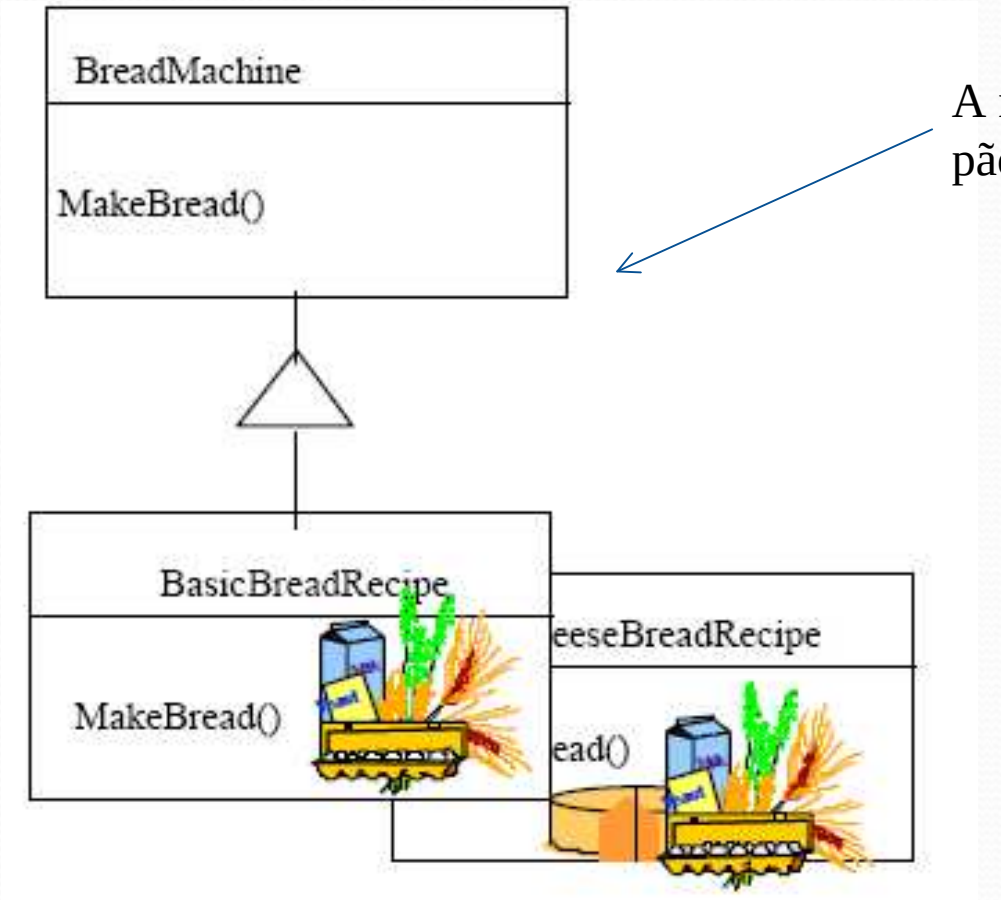


Empresa de Brinquedos

Analogia


Injection Mold
Inject()
ToyHorseMold
Inject()
ToyCarMold
Empresa de Brinquedos
Departamento Especializado
Unicórnio

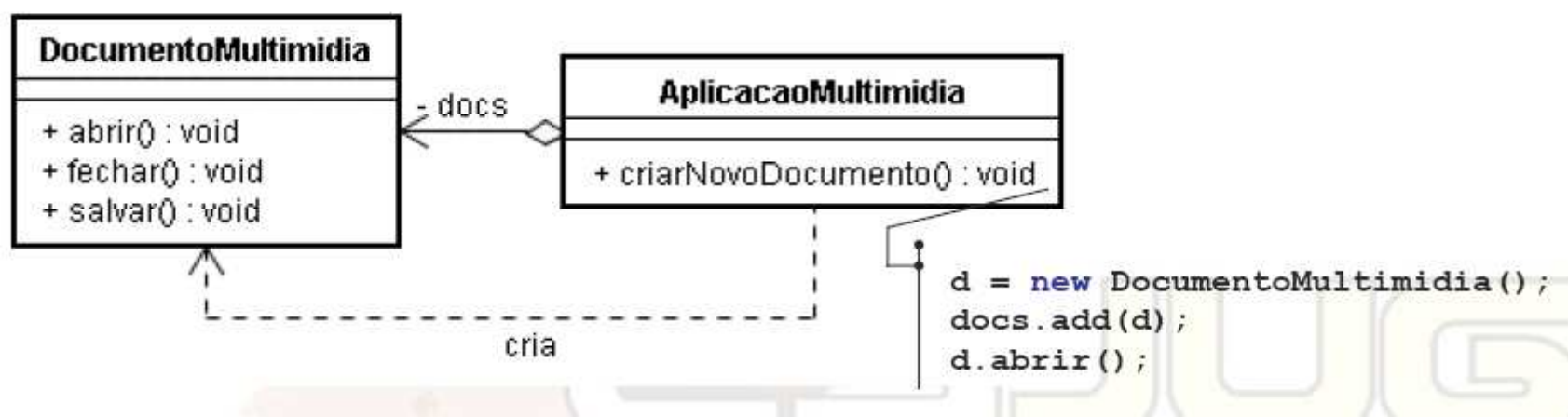
Analogia



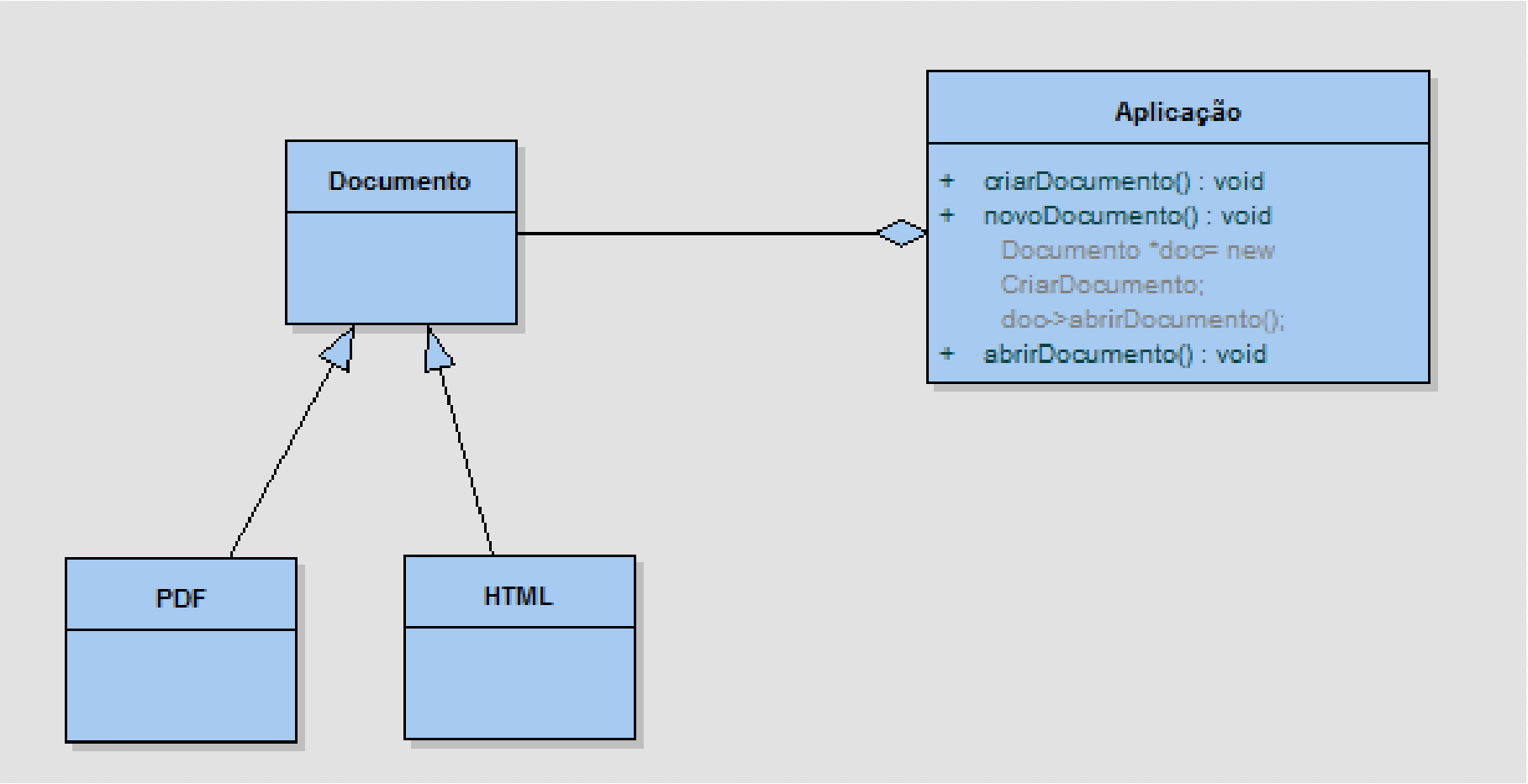
A receita usada determina o tipo de pão que será feito

Motivação

- Framework genérico:
  - Document0 e Aplicação
- Framework específico para uma aplicação que manipula documentos multimídia;
- É possível criar um framework mais genérico, para qualquer aplicação de manipulação de documentos?



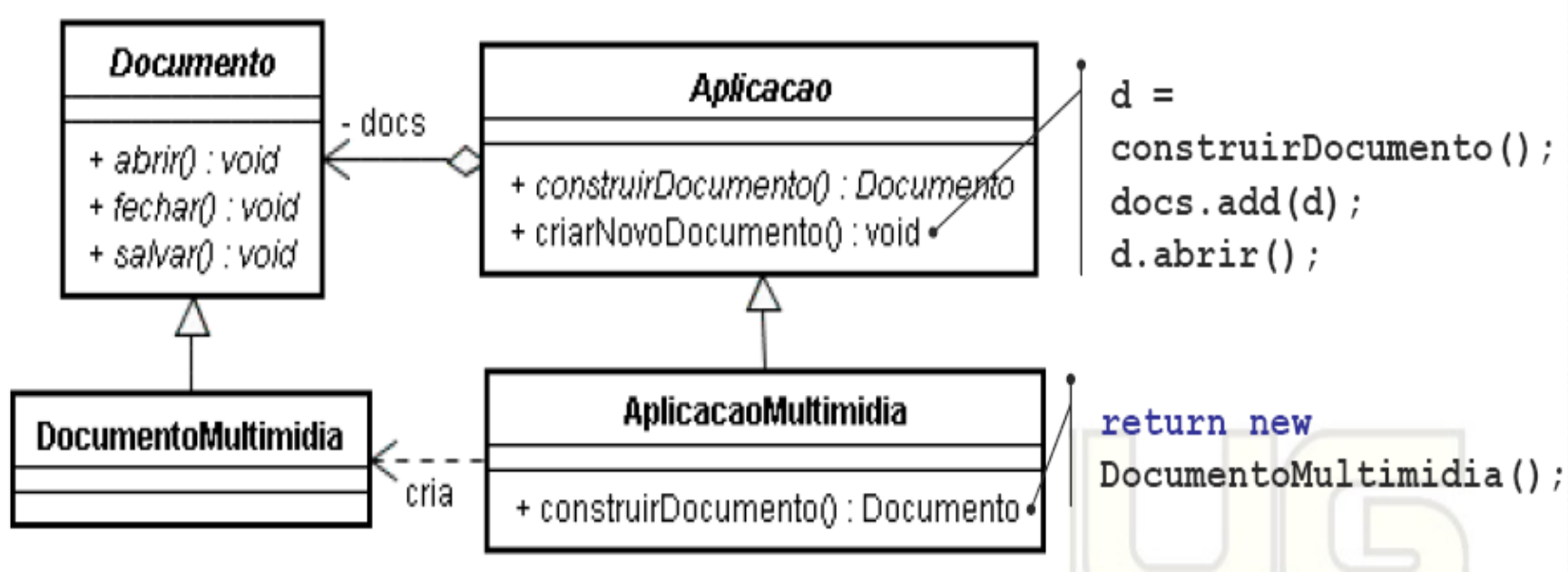
Motivação


Aplicação
+ criarDocumento() : void
+ novoDocumento() : void
Documento *doc= new
CriarDocumento;
doc->abrirDocumento();
+ abrirDocumento() : void
Documento
PDF
HTML

Motivação

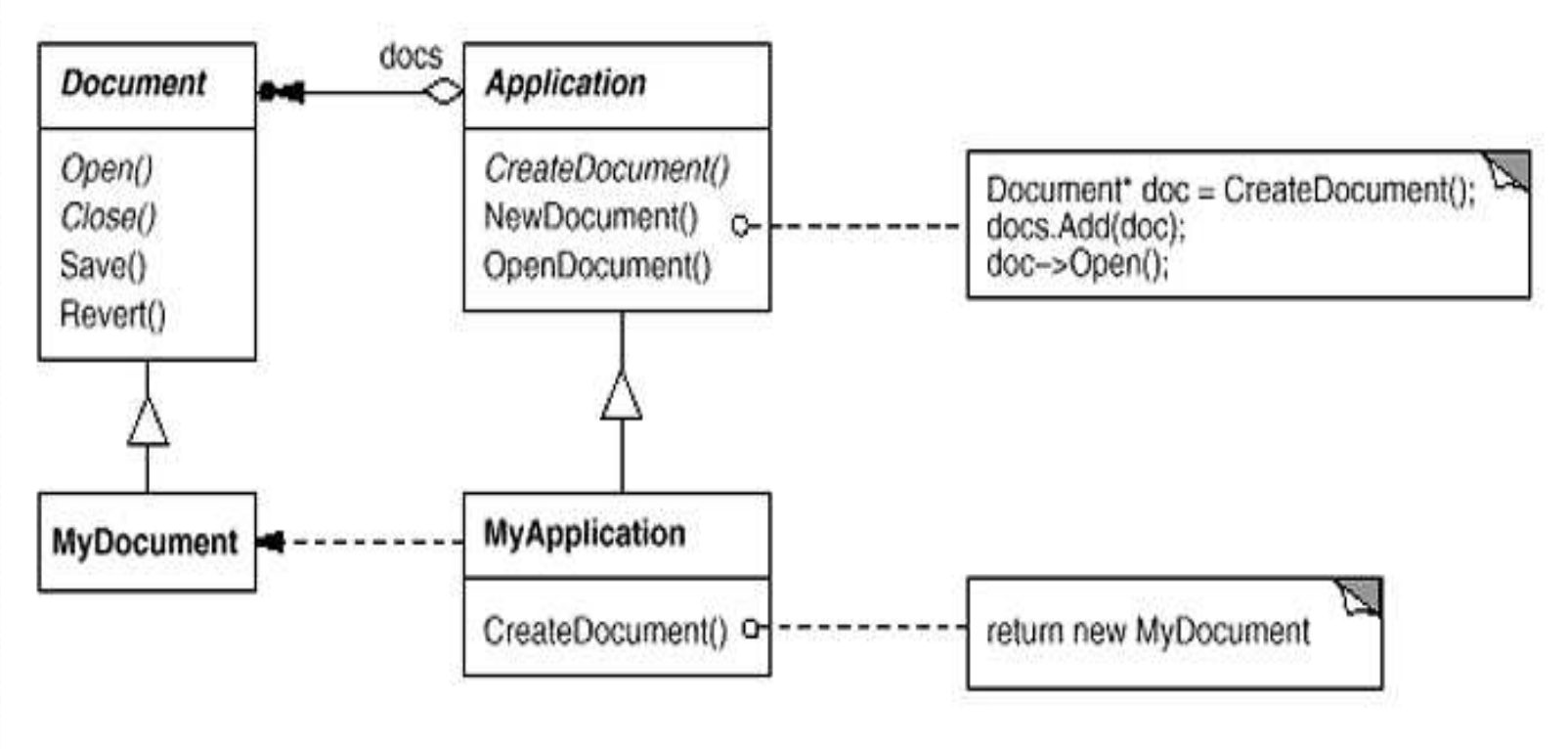
- Frameworks usam classes abstratas para definir objetos
  - Um framework para representar muitos documentos numa aplicação
  - Uma alternativa: o framework manipular alguns formatos predefinidos
    - Inflexível e incompleta
  - Usa duas classes abstratas: Document e Application
    - Aplicação de desenho: DrawingApplication e DrawingDocument
    - Aplicação gerencia os documentos (comando new, open...)

Solução – Factory Method

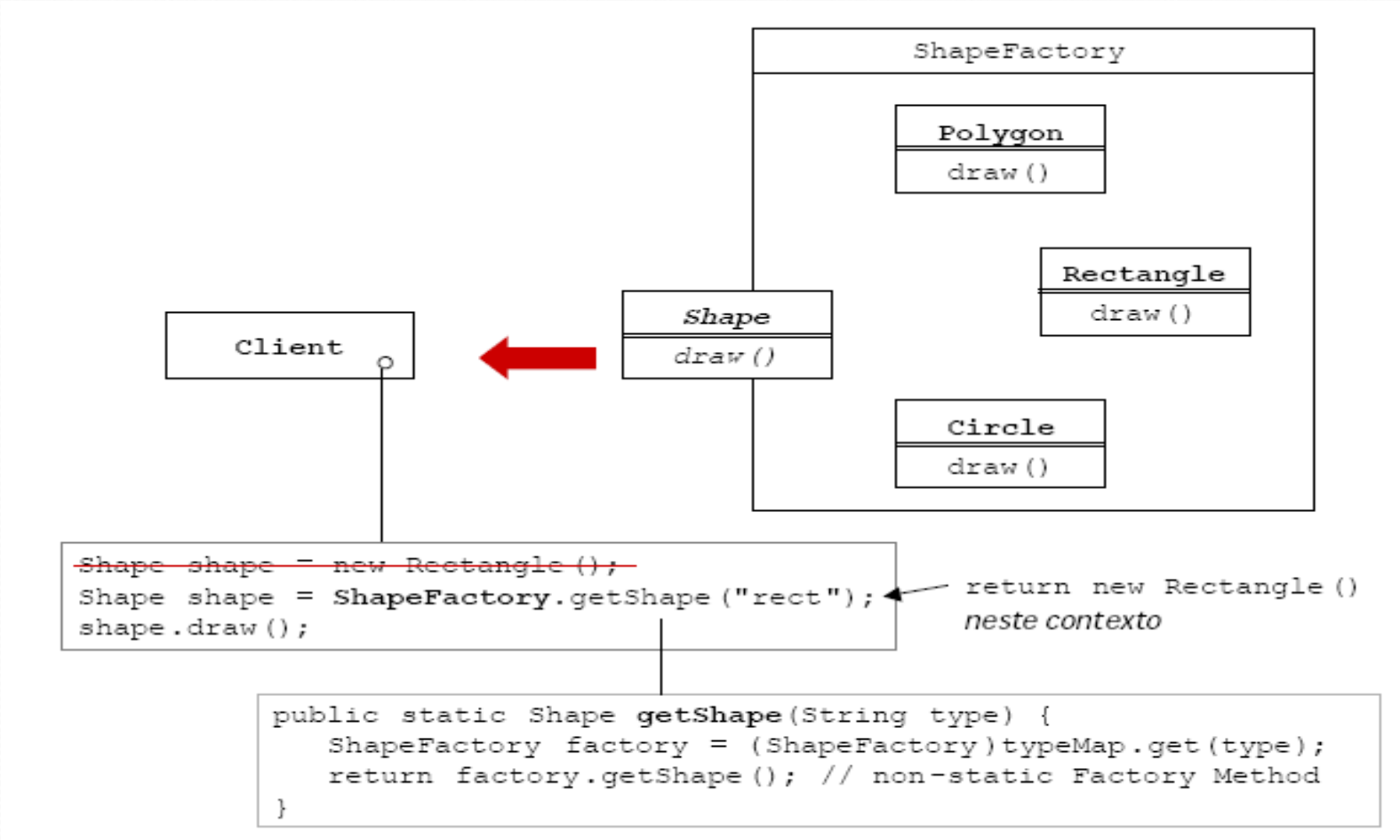


- Classes abstratas implementam as funções comuns a todo tipo de documento;
- Método fábrica é definido na superclasse e implementado na subclasse.
- Cliente implementa os formatos necessários e o framework é responsável pela abstração

Solução – Factory Method



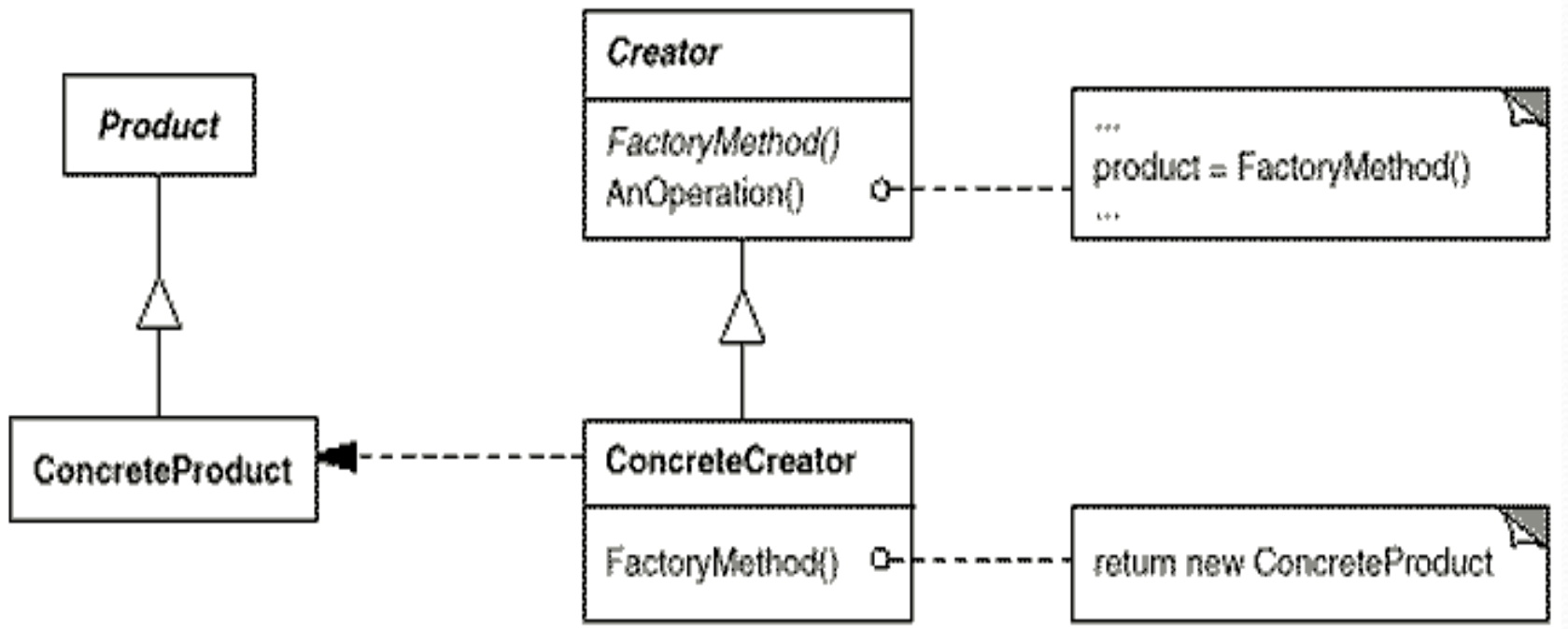
Motivação


ShapeFactory
Polygon
draw()
Rectangle
draw()
Shape
draw()
Client
Circle
draw()
Shape shape = new Rectangle();
Shape shape = ShapeFactory.getShape("rect");
shape.draw();
return new Rectangle()
neste contexto
public static Shape getShape(String type) {
ShapeFactory factory = (ShapeFactory)typeMap.get(type);
return factory.getShape(); // non-static Factory Method
}

Aplicabilidade

- Quando uma classe não tem como saber a classe dos objetos que precisará criar;
- Quando uma classe quer que suas subclasses especifiquem o objeto a ser criado.

Estrutura


Product
Creator
FactoryMethod()
AnOperation()
...
product = FactoryMethod()
...
ConcreteProduct
ConcreteCreator
FactoryMethod()
return new ConcreteProduct

Participantes

- **Product** (Document)
  - Define a interface de objetos que o Factory Method criará
- **ConcreteProduct** (MyDocument)
  - Implementa a interface de Product
- **Creator** (Application)
  - Declara o Factory Method. Retorna um objeto do tipo de produto
  - Pode definir um ConcreteCreator por default
- **ConcreteCreator** (MyApplication)
  - Sobrescreve o Factory Method para retornar um objeto

Conseqüências

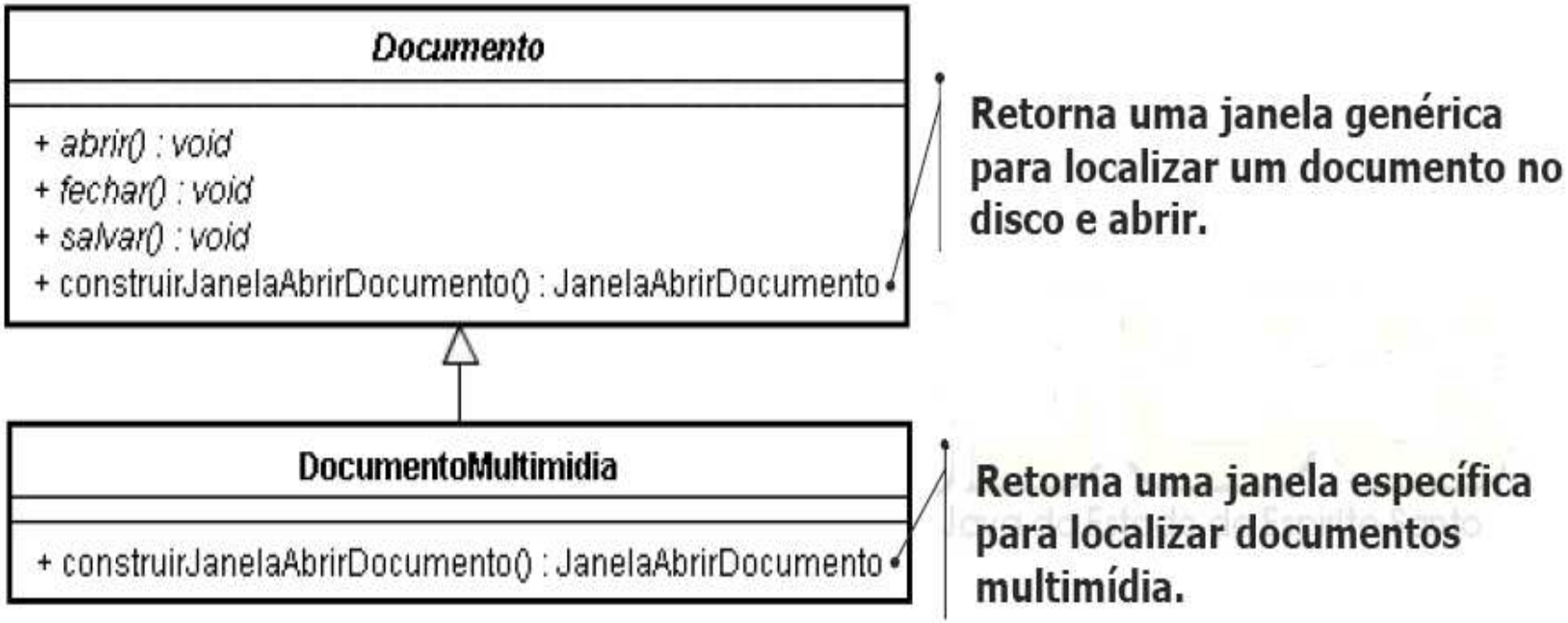
- Melhor extensibilidade:
  - Não é necessário saber a classe concreta do objeto para criá-lo.
- Obrigatoriedade da subclasse fábrica:
  - É preciso conhecer uma factory para o produto
  - Pode ser uma factory genérica(parametrizada)

Templates para Evitar Especialização

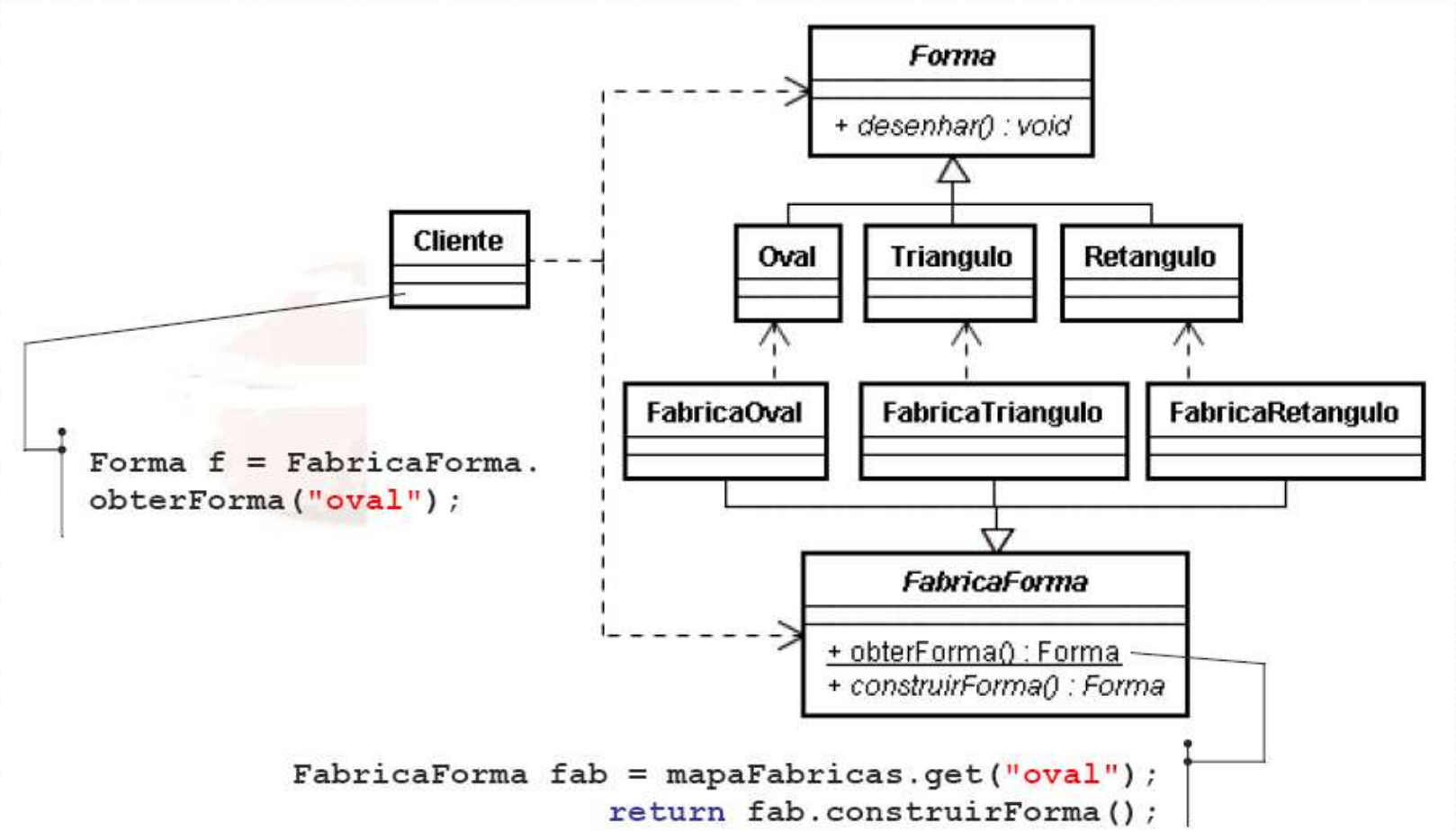
```
class Creator {
public:
    virtual Product* CreateProduct() = 0;
};

template <class TheProduct>
class StandardCreator: public Creator {
public:
    virtual Product* CreateProduct();
};

template <class TheProduct>
Product* StandardCreator<TheProduct>::CreateProduct () {
    return new TheProduct;
}
```

```
class MyProduct : public Product {
public:
    MyProduct();
    // ...
};

StandardCreator<MyProduct> myCreator;
```

Extensão pela subclasse



Método Fábrica Parametrizado



Com Templates

```
class Creator {
public:
    virtual Product* CreateProduct() = 0;
};

template <class TheProduct>
class StandardCreator: public Creator {
public:
    virtual Product* CreateProduct();
};

template <class TheProduct>
Product* StandardCreator<TheProduct>::CreateProduct () {
    return new TheProduct;
}
```

Não precisa herdar o Creator toda vez que deseja criar um produto diferente

```
class MyProduct : public Product {
public:
    MyProduct();
    // ...
};

StandardCreator<MyProduct> myCreator;
```

Implementação

```
class MazeGame {
public:
    Maze* CreateMaze();

// factory methods:

    virtual Maze* MakeMaze() const
        { return new Maze; }
    virtual Room* MakeRoom(int n) const
        { return new Room(n); }
    virtual Wall* MakeWall() const
        { return new Wall; }
    virtual Door* MakeDoor(Room* r1, Room* r2) const
        { return new Door(r1, r2); }
};
```

```
Maze* MazeGame::CreateMaze () {
    Maze* aMaze = MakeMaze();

    Room* r1 = MakeRoom(1);
    Room* r2 = MakeRoom(2);
    Door* theDoor = MakeDoor(r1, r2);

    aMaze->AddRoom(r1);
    aMaze->AddRoom(r2);

    r1->SetSide(North, MakeWall());
    r1->SetSide(East, theDoor);
    r1->SetSide(South, MakeWall());
    r1->SetSide(West, MakeWall());

    r2->SetSide(North, MakeWall());
    r2->SetSide(East, MakeWall());
    r2->SetSide(South, MakeWall());
    r2->SetSide(West, theDoor);

    return aMaze;
}
```

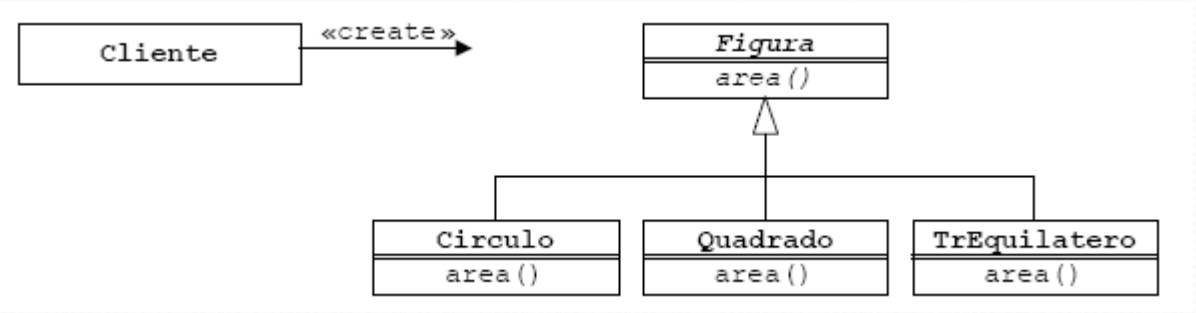
Implementação

```
class BombedMazeGame : public MazeGame {
public:
    BombedMazeGame();

    virtual Wall* MakeWall() const
        { return new BombedWall; }

    virtual Room* MakeRoom(int n) const
        { return new RoomWithABomb(n); }
};
```

```
class EnchantedMazeGame : public MazeGame {
public:
    EnchantedMazeGame();

    virtual Room* MakeRoom(int n) const
        { return new EnchantedRoom(n, CastSpell()); }

    virtual Door* MakeDoor(Room* r1, Room* r2) const
        { return new DoorNeedingSpell(r1, r2); }
protected:
    Spell* CastSpell() const;
};
```

Exercício

- Implemente a aplicação abaixo usando Factory Method para criar os objetos:
  - Crie um objeto construtor para cada tipo de objeto (XXXFactory para criar figuras do tipo XXX)
  - Utilize a fachada Figuras que contém um HashMap, onde os construtores são guardados, e um método estático que seleciona o construtor desejado com uma chave

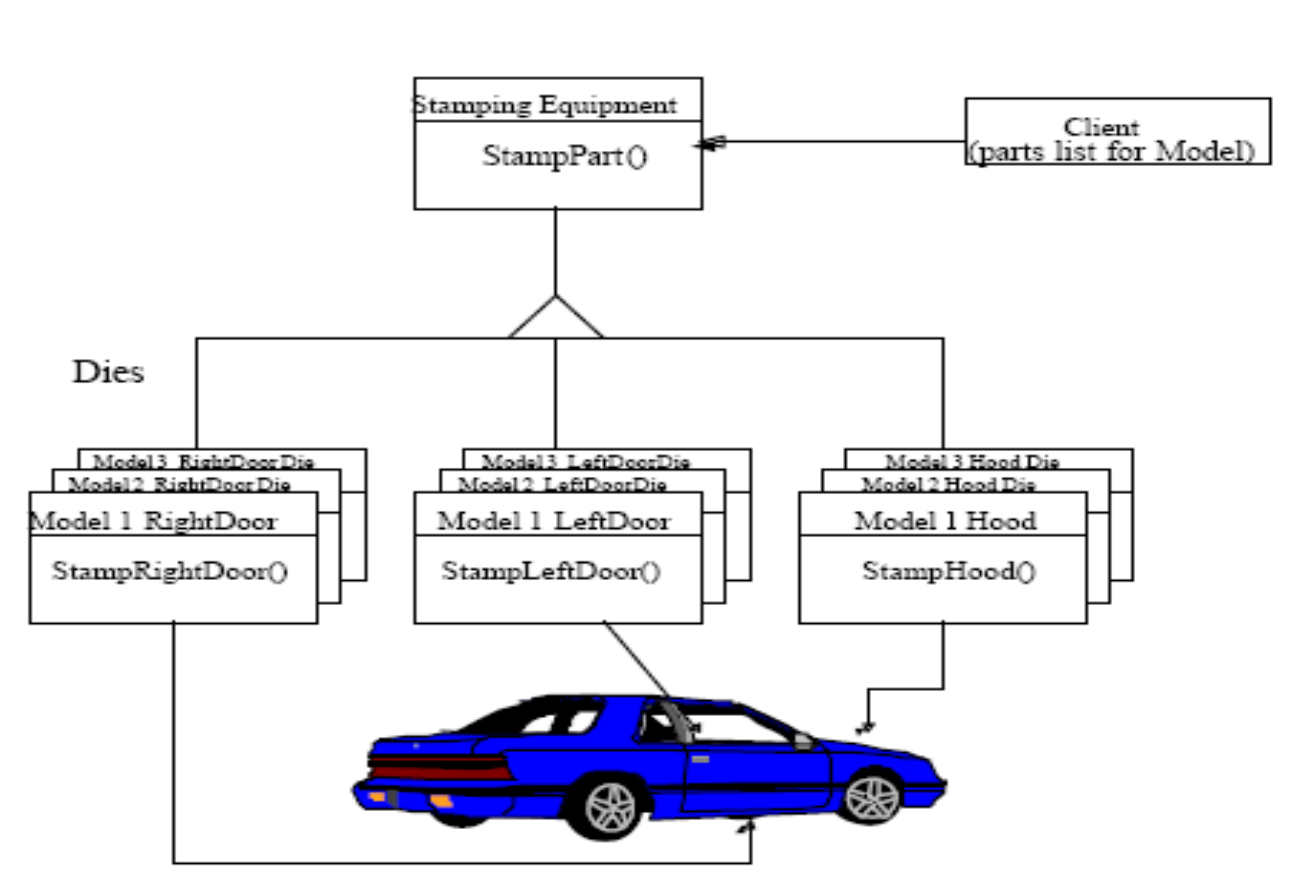


Abstract Factory

Objetivo:

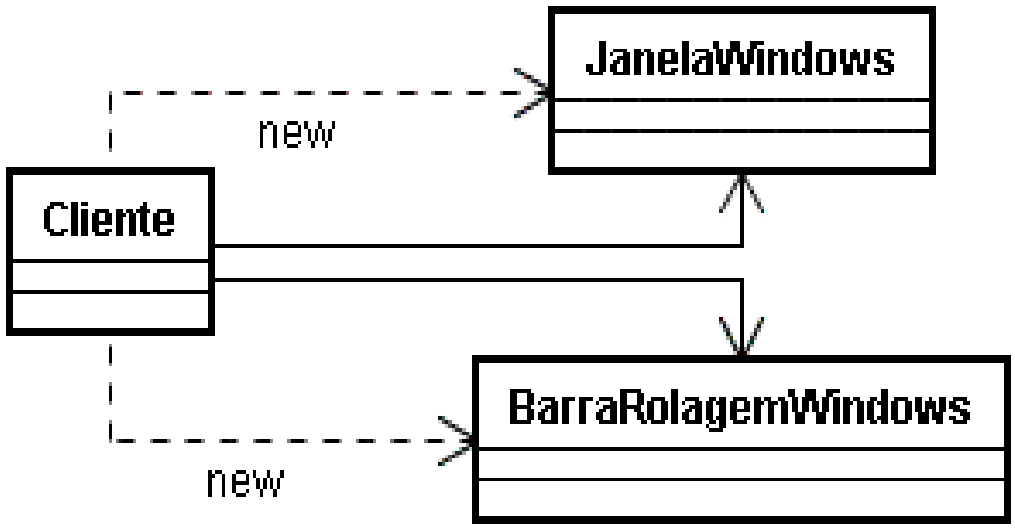
"Prover uma interface para criar famílias de objetos relacionados ou dependentes sem especificar suas classes concretas." [GoF]

Analogia

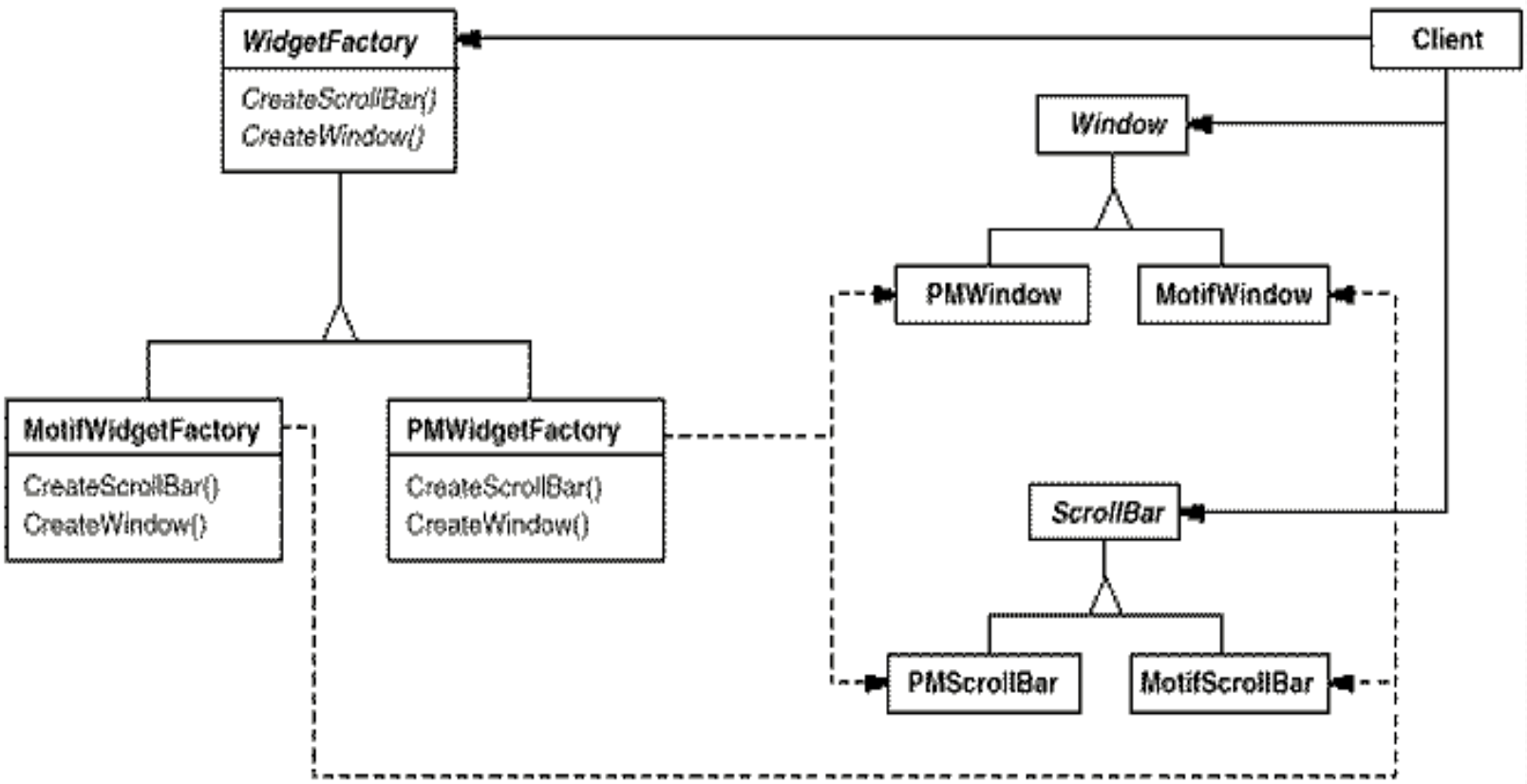

Stamping Equipment
StampPart()
Client
(parts list for Model)
Dies
Model 1 RightDoor
StampRightDoor()
Model 1 LeftDoor
StampLeftDoor()
Model 1 Hood
StampHood()

Motivação

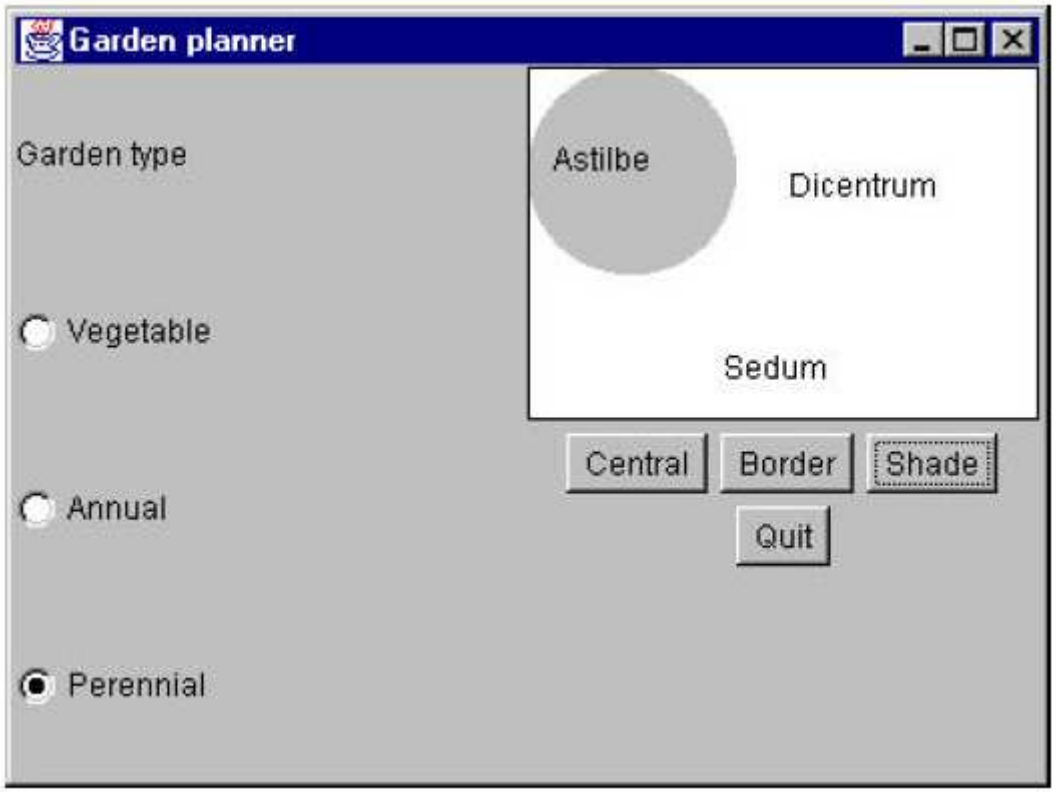
- GUI deve suportar diferentes look-and-feels
- Não se deve instanciar classes específicas para look-and-feel – dificulta a mudança



Solução – Abstract Factory

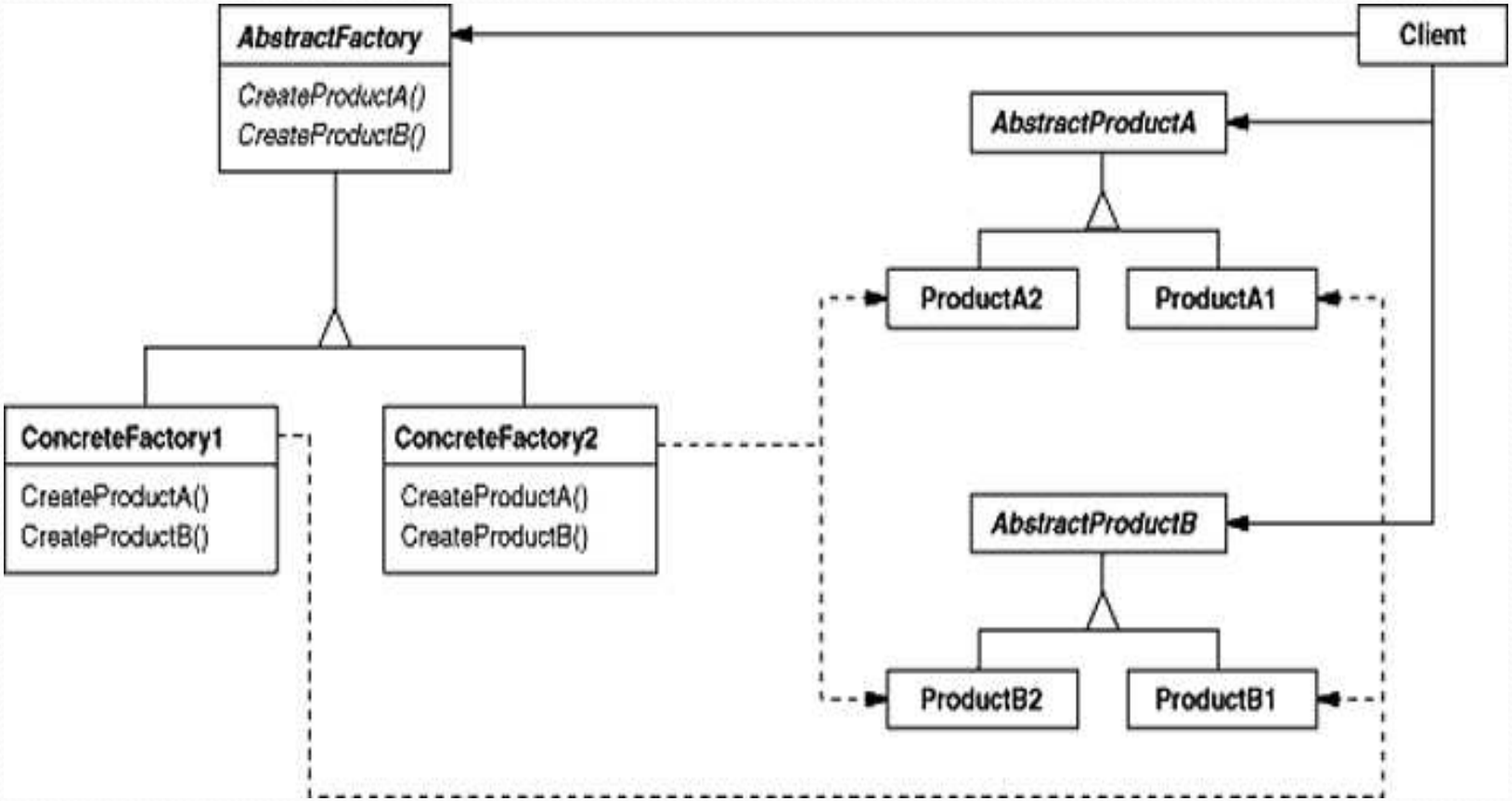


Motivação



Uma aplicação para planejar o layout de jardins

Estrutura


AbstractFactory
CreateProductA()
CreateProductB()
Client
AbstractProductA
ProductA2
ProductA1
ConcreteFactory1
CreateProductA()
CreateProductB()
ConcreteFactory2
CreateProductA()
CreateProductB()
AbstractProductB
ProductB2
ProductB1

Participantes

- **AbstractFactory** (WidgetFactory)
  - Declara a interface para criar objetos
- **ConcreteFactory** (MotifWidgetFactory, PMWidgetFactory)
  - Implementa as operações para criar objetos
- **AbstractProduct** (Window, ScrollBar)
  - Declara a interface para os tipos de produtos
- **ConcreteProduct** (MotifWindow, MotifScrollBar)
  - Define o produto a ser criado
- **Client**
  - Usa as interfaces de AbstractProduct e AbstractFactory

Aplicabilidade

- Quando o sistema tiver que ser independente de como seus produtos são criados, compostos ou representados;
- Quando o sistema tiver que ser configurado com uma ou mais famílias de produtos (classes que devem ser usadas sempre em conjunto);
- Quando você quiser construir uma biblioteca de produtos e quiser revelar apenas suas interfaces e não suas implementações.

Conseqüências

- Isola classes concretas:
  - Fábricas cuidam da instanciação, o cliente só conhece interfaces.
- Facilita a troca de famílias de classes:
  - Basta trocar de fábrica concreta.
- Promove consistência interna:
  - Não dá pra usar um produto de uma família com um de outra.
- Criar novos produtos é trabalhoso:
  - É necessário alterar as implementações de todas as fábricas para suportar o novo produto.
    - Ex.: inclusão de aparelhos de DVD nos carros

Implementação

```
Maze* MazeGame::CreateMaze (MazeFactory& factory) {
    Maze* aMaze = factory.MakeMaze();
    Room* r1 = factory.MakeRoom(1);
    Room* r2 = factory.MakeRoom(2);
    Door* aDoor = factory.MakeDoor(r1, r2);

    aMaze->AddRoom(r1);
    aMaze->AddRoom(r2);

    r1->SetSide(North, factory.MakeWall());
    r1->SetSide(East, aDoor);
    r1->SetSide(South, factory.MakeWall());
    r1->SetSide(West, factory.MakeWall());

    r2->SetSide(North, factory.MakeWall());
    r2->SetSide(East, factory.MakeWall());
    r2->SetSide(South, factory.MakeWall());
    r2->SetSide(West, aDoor);

    return aMaze;
}
```

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

    virtual Maze* MakeMaze() const
        { return new Maze; }
    virtual Wall* MakeWall() const
        { return new Wall; }
    virtual Room* MakeRoom(int n) const
        { return new Room(n); }
    virtual Door* MakeDoor(Room* r1, Room* r2) const
        { return new Door(r1, r2); }
};
```

Implementação

```
class EnchantedMazeFactory : public MazeFactory {
public:
    EnchantedMazeFactory();

    virtual Room* MakeRoom(int n)  const
        { return new EnchantedRoom(n, CastSpell()); }

    virtual Door* MakeDoor(Room* r1, Room* r2)  const
        { return new DoorNeedingSpell(r1, r2); }

protected:
    Spell* CastSpell() const;
};
```

```
Wall* BombedMazeFactory::MakeWall () const {
    return new BombedWall;
}

Room* BombedMazeFactory::MakeRoom(int n) const {
    return new RoomWithABomb(n);
}
```

```
MazeGame game;
BombedMazeFactory factory;

game.CreateMaze(factory);
```

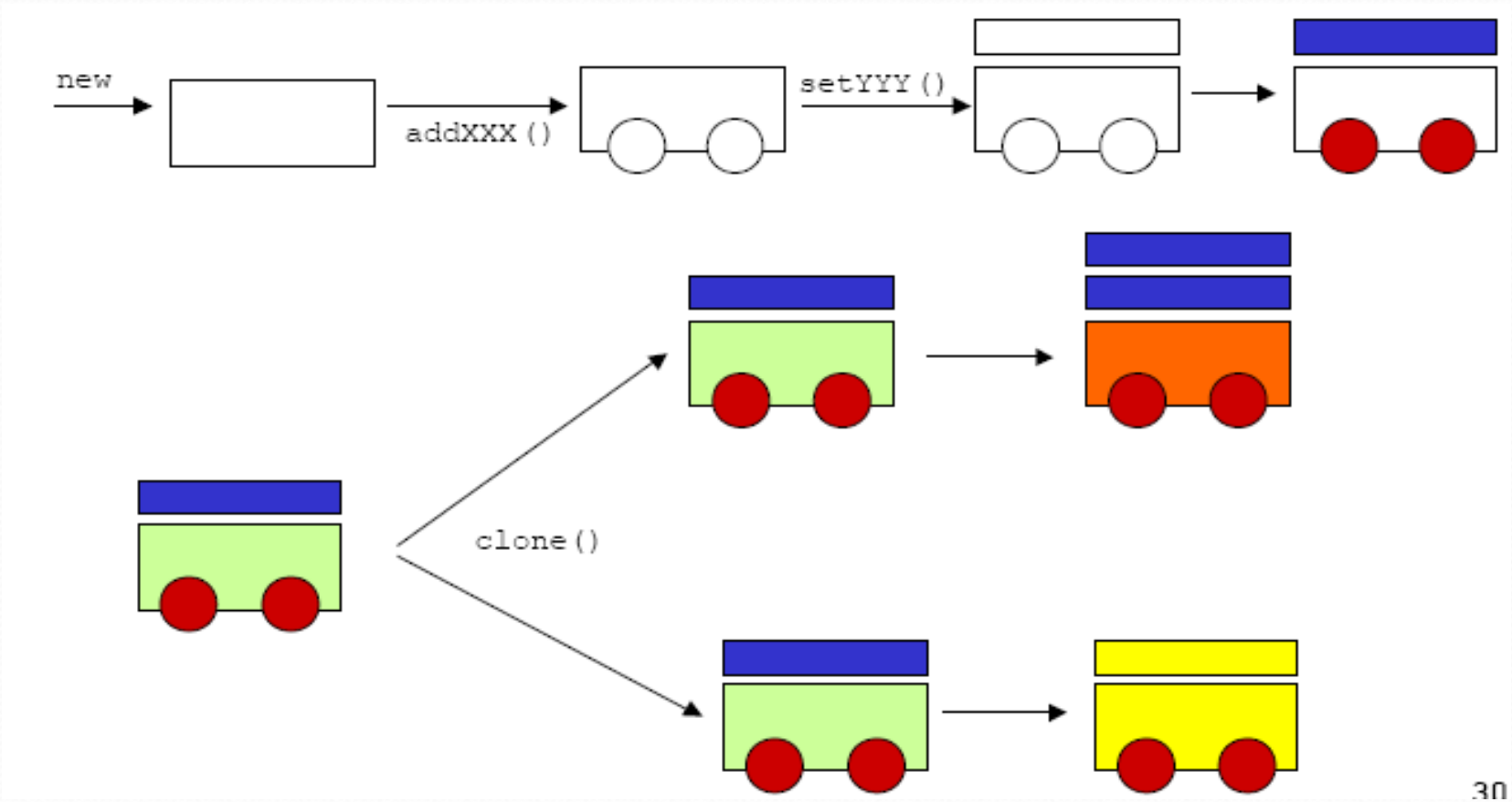
Prototype

14

Objetivo:

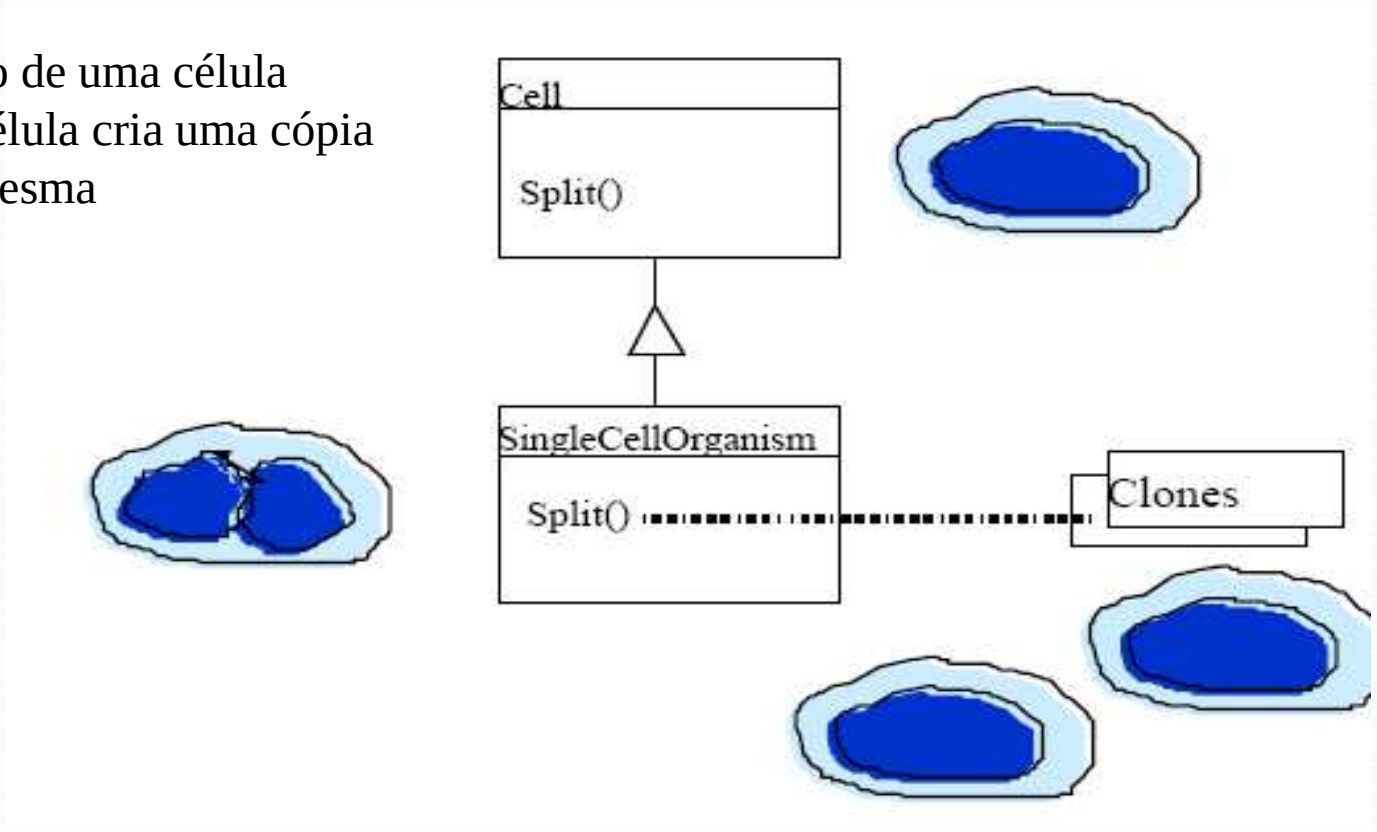
"Especificar os tipos de objetos a serem criados usando uma instância como protótipo e criar novos objetos ao copiar este protótipo." [GoF]

Analogia



Analogia

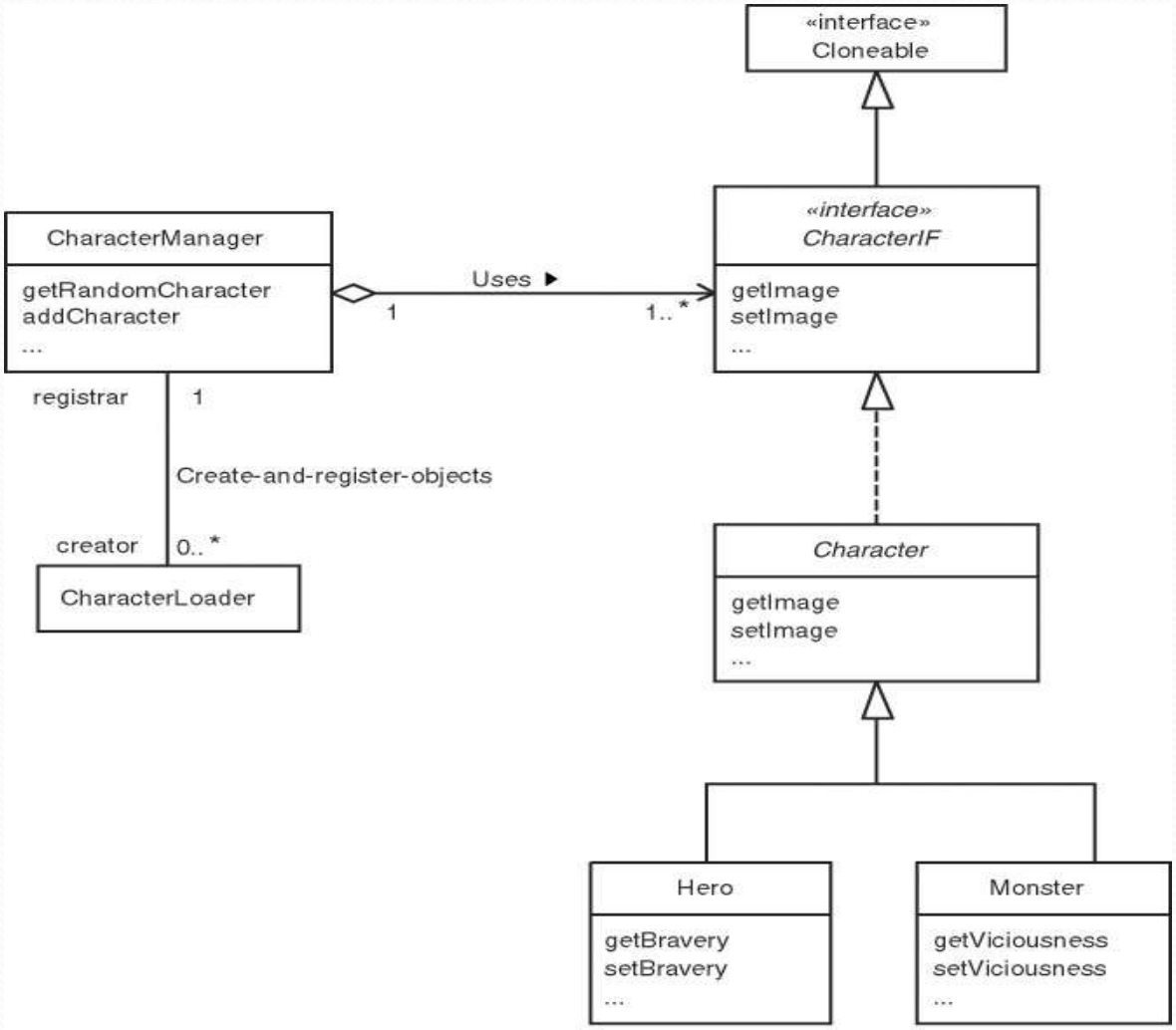
Divisão de uma célula
Uma célula cria uma cópia de si mesma



Motivação

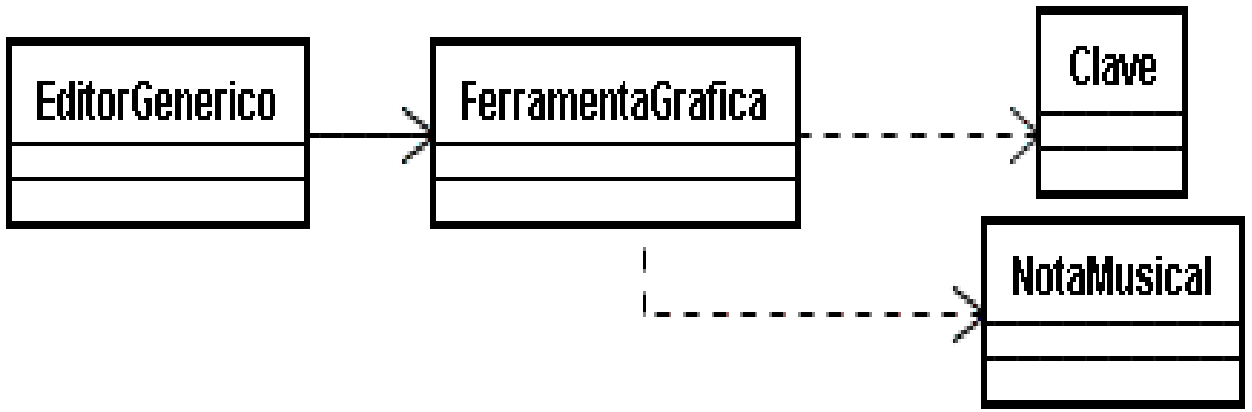
- Objetivo: construir um jogo de RPG interativo
  - Jogadores cansam dos velhos personagens e desejam novos;
  - Construir um add-on para gerar alguns personagens padrão e gerar novos
- Classes: monstro, vilão...
  - Os atributos os fazem diferentes
    - Imagem, nome, inteligência...

Estrutura da Aplicação



Motivação

- Construir uma Ferramenta de Edição de Partituras reutilizando uma ferramenta de edição gráfica genérica (framework)
  - Adição de objetos que representam elementos musicais
  - Ferramenta deve ter paleta – mover
  - Problema: as classes estão fora do framework – especifica para a aplicação
    - Framework não conhece estas classes
    - Alternativa: especialização – subclasses para invocar cada objeto musical

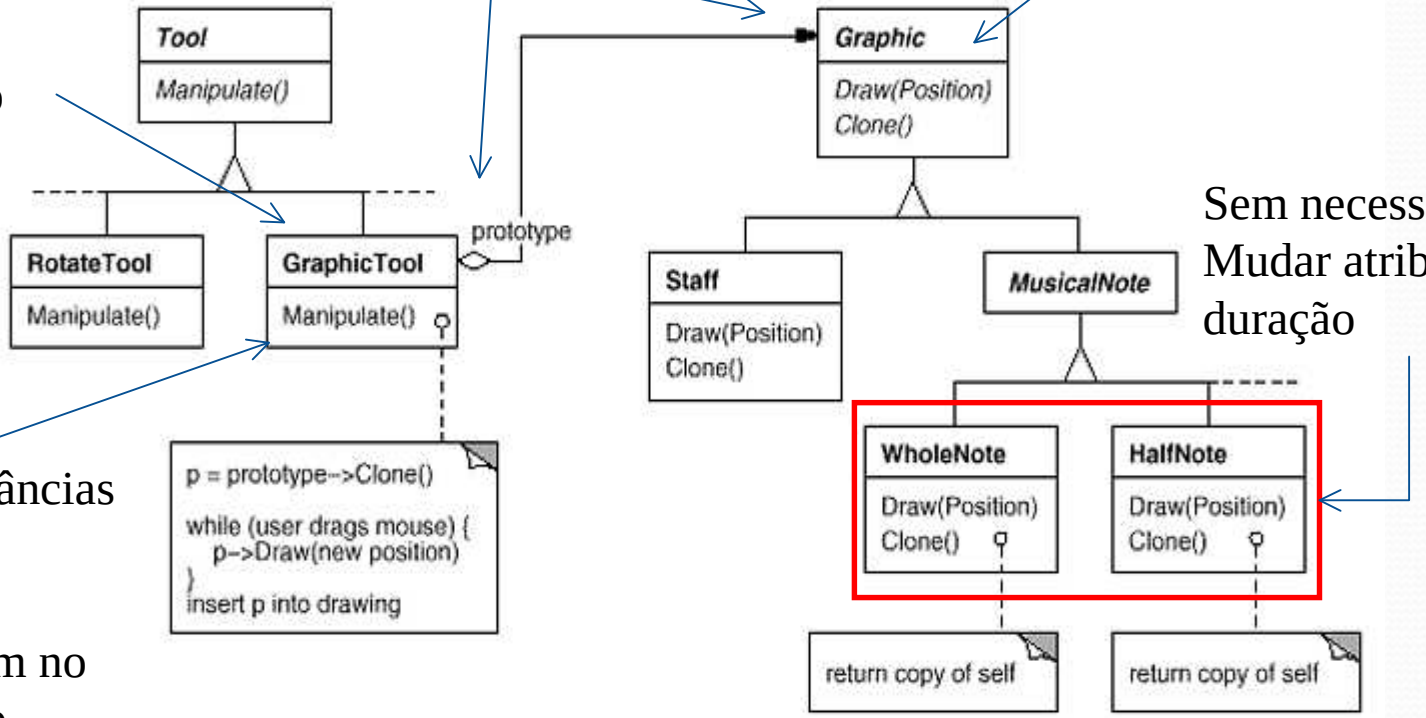


Estrutura da Aplicação

GT clona componentes gráficos

Componentes gráficos

O protótipo é passado como parâmetro



Sem necessidade
Mudar atributos de duração

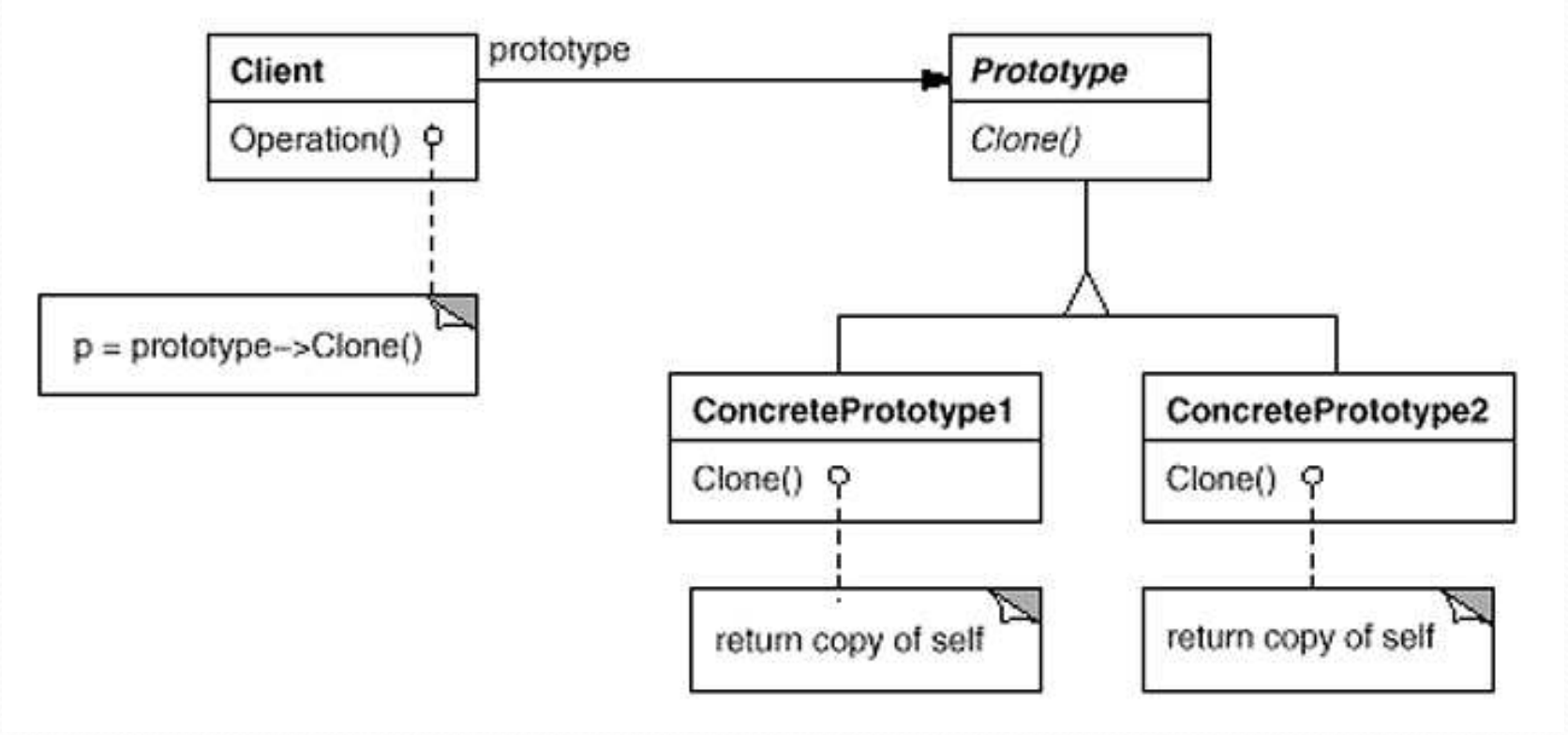
Criam instâncias de objetos gráficos e adicionam no documento

Ocorre em run-time, ao mover um objeto da paleta
Há uma instância de GT para cada componente gráfico

Aplicabilidade

- Quando o sistema precisar ser independente de como os produtos serão criados
- Quando as classes a serem instanciadas são decididas em tempo de execução;
- Para evitar construir uma hierarquia de factorys para cada produto;
- Quando objetos podem ter somente uma das poucas combinações de estados de um objeto
  - É mais apropriado instalar um numero de protótipos e cloná-los, do que instanciar cada classe toda vez que seja necessário

Estrutura


Client
Operation()
prototype
Prototype
Clone()
p = prototype->Clone()
ConcretePrototype1
Clone()
ConcretePrototype2
Clone()
return copy of self
return copy of self

Participantes

- **Prototype** (Graphic)
  - Declara uma interface para clonar a si mesmo
- **ConcretePrototype** (Staff, WholeNote, HalfNote)
  - Implementa um operação para clonar a si mesmo
- **Client** (GraphicTool)
  - Cria um novo objeto ao pedir a um objeto para se auto clonar

Conseqüências

- Esconde a implementação do produto
  - Registrando protótipos com o cliente
- Permite adicionar e remover produtos em tempo de execução (configuração dinâmica da aplicação)
  - Basta alterar o valor dos atributos
- Não necessita de uma fábrica para cada hierarquia de objetos – reduz o número de subclasses
- Implementar clone() pode ser complicado
  - Caso da redundância – um objeto referencia outro que referencia outro que acaba referenciando o primeiro.
  - Objeto pré-existente não oferece o método clone()

Implementação

```
class MazePrototypeFactory : public MazeFactory {
public:
    MazePrototypeFactory(Maze*, Wall*, Room*, Door*);

    virtual Maze* MakeMaze() const;
    virtual Room* MakeRoom(int) const;
    virtual Wall* MakeWall() const;
    virtual Door* MakeDoor(Room*, Room*) const;

private:
    Maze* _prototypeMaze;
    Room* _prototypeRoom;
    Wall* _prototypeWall;
    Door* _prototypeDoor;
};
```

```
MazePrototypeFactory::MazePrototypeFactory (
    Maze* m, Wall* w, Room* r, Door* d
) {
    _prototypeMaze = m;
    _prototypeWall = w;
    _prototypeRoom = r;
    _prototypeDoor = d;
}
```

```
Wall* MazePrototypeFactory::MakeWall () const {
    return _prototypeWall->Clone();
}

Door* MazePrototypeFactory::MakeDoor (Room* r1, Room *r2) const {
    Door* door = _prototypeDoor->Clone();
    door->Initialize(r1, r2);
    return door;
}
```

Implementação

```
MazeGame game;
MazePrototypeFactory simpleMazeFactory(
    new Maze, new Wall, new Room, new Door
);

Maze* maze = game.CreateMaze(simpleMazeFactory);
```

```
MazePrototypeFactory bombedMazeFactory(
    new Maze, new BombedWall,
    new RoomWithABomb, new Door
);
```

```
class Door : public MapSite {
public:
    Door();
    Door(const Door&);

    virtual void Initialize(Room*, Room*);
    virtual Door* Clone() const;

    virtual void Enter();
    Room* OtherSideFrom(Room*);
private:
    Room* _room1;
    Room* _room2;
};

Door::Door (const Door& other) {
    _room1 = other._room1;
    _room2 = other._room2;
}

void Door::Initialize (Room* r1, Room* r2) {
    _room1 = r1;
    _room2 = r2;
}

Door* Door::Clone () const {
    return new Door(*this);
}
```

Implementação

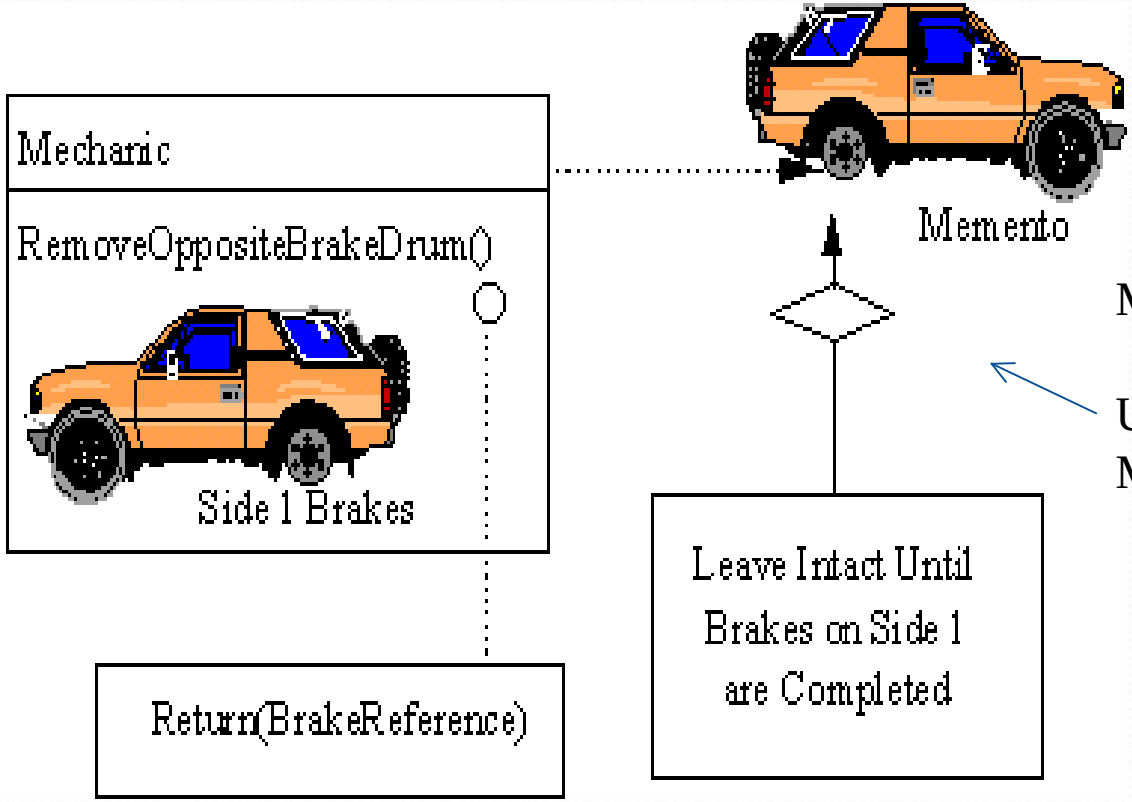
```
class BombedWall : public Wall {
public:
    BombedWall();
    BombedWall(const BombedWall&);

    virtual Wall* Clone() const;
    bool HasBomb();
private:
    bool _bomb;
};

BombedWall::BombedWall (const BombedWall& other) : Wall(other) {
    _bomb = other._bomb;
}

Wall* BombedWall::Clone () const {
    return new BombedWall(*this);
}
```

Memento

Objetivo:

"Sem violar o encapsulamento, capturar e externalizar o estado interno de um objeto para que o objeto possa ter esse estado restaurado posteriormente." [GoF]

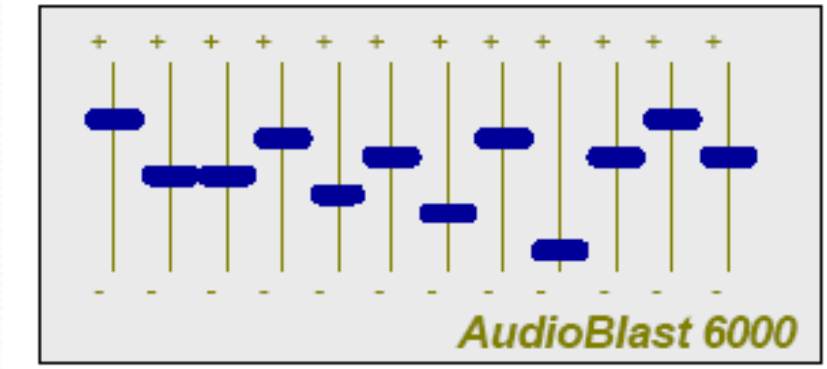
Analogia

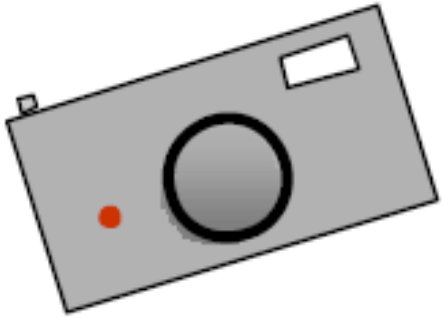


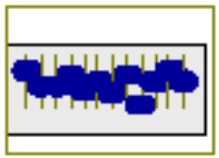
Mudando a pastilha de freio.

Um lado do carro serve como Memento para o outro lado.

Analogia





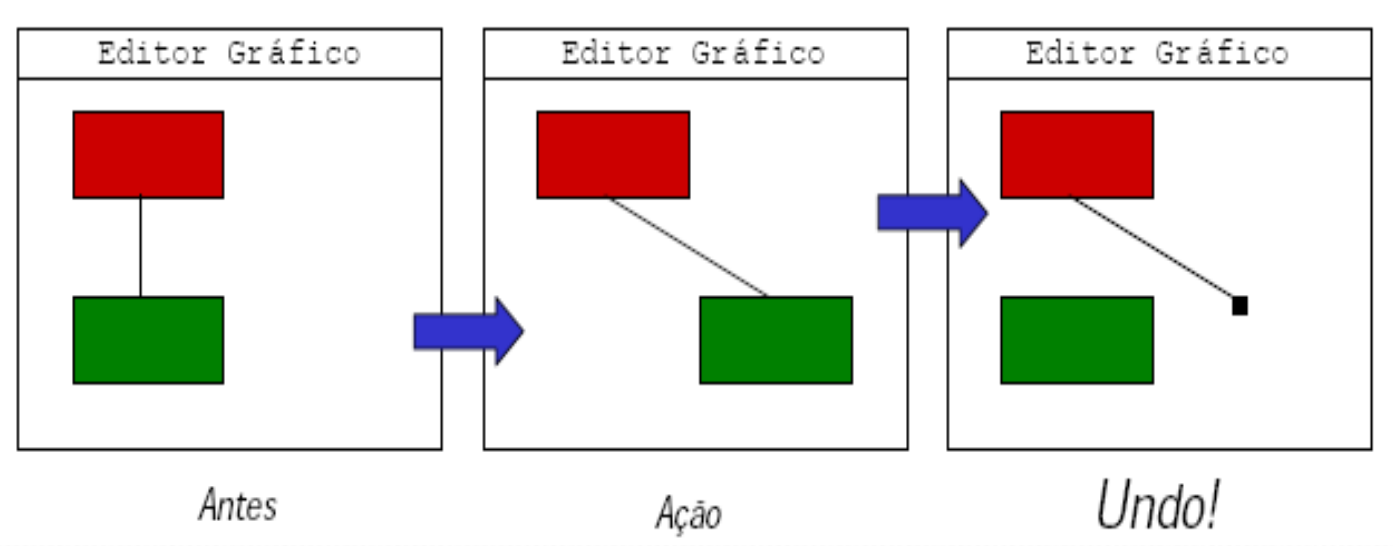


Bater uma foto da configuração
Da caixa de som, para poder retornar
A este estado

Analogia

Motivação

- Certas vezes é necessário armazenar os estados internos de um objeto
  - Voltar a um estado ou se recuperar de erros
- É preciso guardar informações sobre um objeto para poder desfazer uma operação

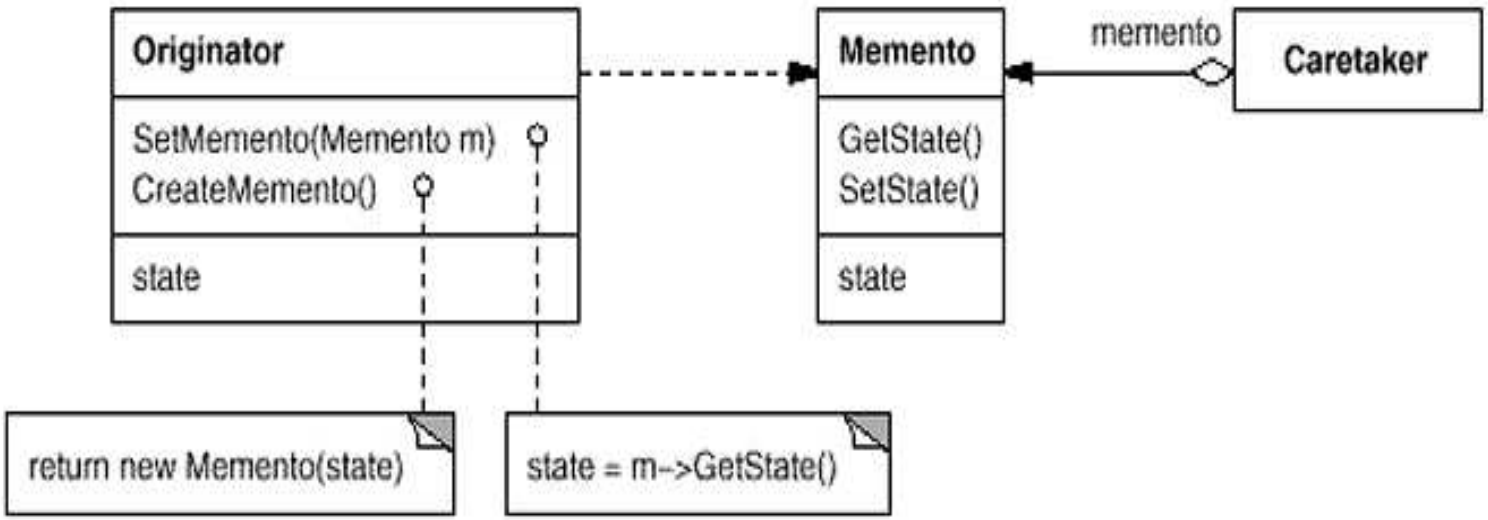


Motivação

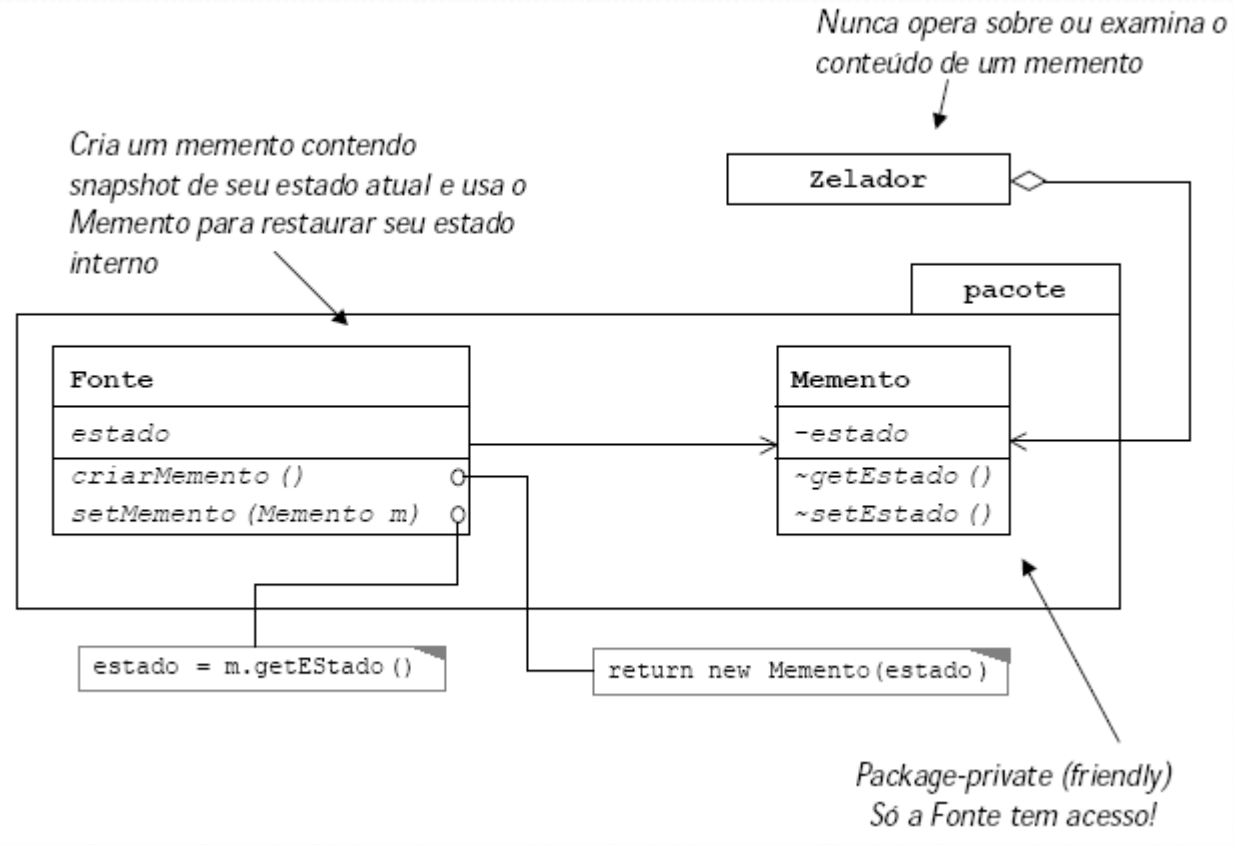
Solução - Memento

- Memento armazena o estado interno de um objeto – originator
- A aplicação solicita um memento a um originator quando ele necessita fazer o checkpoint do estado do originator
- O originator inicializa o memento com a informação do seu estado corrente
- Somente o originator pode armazenar e retornar informações do memento

Estrutura


Originator
SetMemento(Memento m)
CreateMemento()
state
Memento
GetState()
SetState()
state
memento
Caretaker
return new Memento(state)
state = m->GetState()

Estrutura II



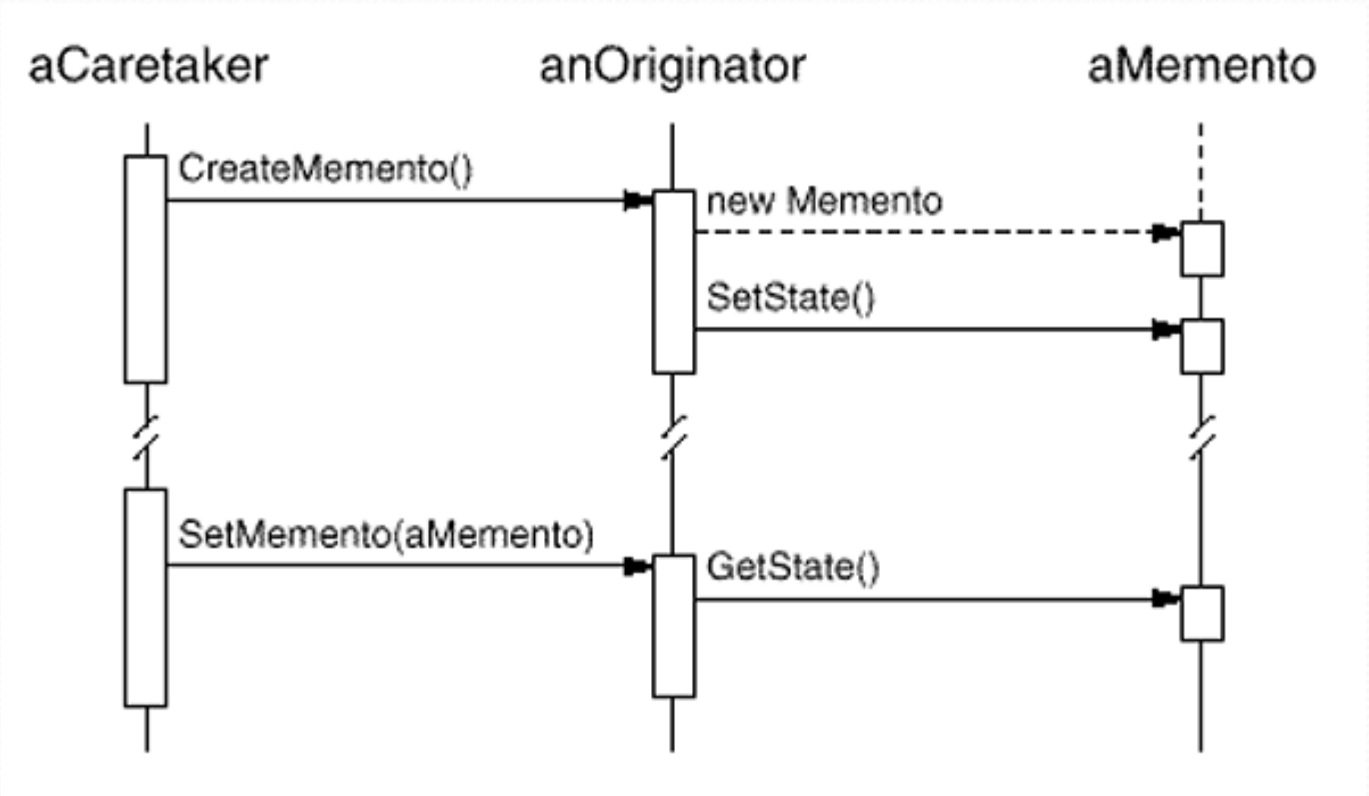
Participantes

- **Memento** (SolverState)
  - Armazena o estado interno de um objeto Originator
  - Protege contra o acesso de outros objetos, tirando o Originator
- **Originator** (ConstraintSolver)
  - Usa o memento para restaurar seu estado
- **Caretaker** (undo mechanism)
  - Passa o memento a outros objetos
  - Nunca opera em ou examina o conteúdo de um memento
  - Responsável por deletar mementos que estão a seu cuidado

Aplicabilidade

- Um snapshot do (parte do) estado de um objeto precisa ser armazenada para que ele possa ser restaurado ao seu estado original posteriormente
- Uma interface direta para se obter esse estado iria expor detalhes de implementação e quebrar o encapsulamento do objeto (i.e. atributos públicos)

Colaboração



Conseqüências

- Preserva o encapsulamento:
  - O pattern evita expor informações que somente o Originator deve gerenciar, mas é armazenado fora do Originator;
  - Retira do Originator a tarefa de armazenar estados anteriores;
  - No entanto pode ser difícil esconder este estado (somente o Originator pode ter acesso ao memento) em algumas linguagens.
- Pode ser caro:
  - Dependendo da quantidade de estado (informação) a ser armazenado no memento, pode custar caro.

Persistência

- Memento persistente?
  - Em banco de dados?
  - Serializado em disco?
  - Convertido em texto (XML)?
  - Etc.
- Memento só em memória?

Implementação

```
class Graphic;
    // base class for graphical objects in the graphic

class MoveCommand {
public:
    MoveCommand(Graphic* target, const Point& delta);
    void Execute();
    void Unexecute();
private:
    ConstraintSolverMemento* _state;
    Point _delta;
    Graphic* _target;
};
```

Move o objeto

Desfaz o movimento

```
void MoveCommand::Execute () {
    ConstraintSolver* solver = ConstraintSolver::Instance();
    _state = solver->CreateMemento(); // create a memento
    _target->Move(_delta);
    solver->Solve();
}

void MoveCommand::Unexecute () {
    ConstraintSolver* solver = ConstraintSolver::Instance();
    _target->Move(-_delta);
    solver->SetMemento(_state); // restore solver state
    solver->Solve();
}
```

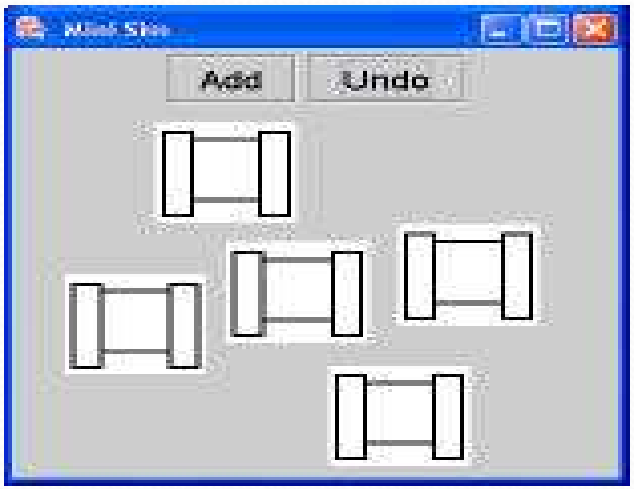
```
class ConstraintSolver {
public:
    static ConstraintSolver* Instance();

    void Solve();
    void AddConstraint(
        Graphic* startConnection, Graphic* endConnection
    );
    void RemoveConstraint(
        Graphic* startConnection, Graphic* endConnection
    );

    ConstraintSolverMemento* CreateMemento();
    void SetMemento(ConstraintSolverMemento*);
private:
    // nontrivial state and operations for enforcing
    // connectivity semantics
};

class ConstraintSolverMemento {
public:
    virtual ~ConstraintSolverMemento();
private:
    friend class ConstraintSolver;
    ConstraintSolverMemento();

    // private constraint solver state
};
```

Implementação

```
void MoveCommand::Execute () {
    ConstraintSolver* solver = ConstraintSolver::Instance();
    _state = solver->CreateMemento(); // create a memento
    _target->Move(_delta);
    solver->Solve();
}

void MoveCommand::Unexecute () {
    ConstraintSolver* solver = ConstraintSolver::Instance();
    _target->Move(-_delta);
    solver->SetMemento(_state); // restore solver state
    solver->Solve();
}
```

Exercício

- Cada vez que um objeto for criado ou movido, o sistema criará um memento do objeto e o guardará em uma pilha
- Cada vez que o usuário clicar no botão Undo, o código irá recuperar o memento mais recente e restaurar a simulação ao estado armazenado no topo da pilha



Exercício

- Escreva uma aplicação gráfica simples que permite digitar texto em um TextField que é copiado para um TextArea (um objeto em cada linha) quando o usuário aperta o botão gravar. Em seguida o TextField é esvaziado
  - Crie um botão "Desfazer"
  - Implemente uma operação de Undo que permita desfazer todas as operações (recuperar o texto anterior no TextField e mostrar o TextArea sem o texto)
- Grave as alterações em disco de forma que, se a aplicação fechar, quando ela reiniciar, ela "lembre” do estado em que estava antes de fechar.

Resumo

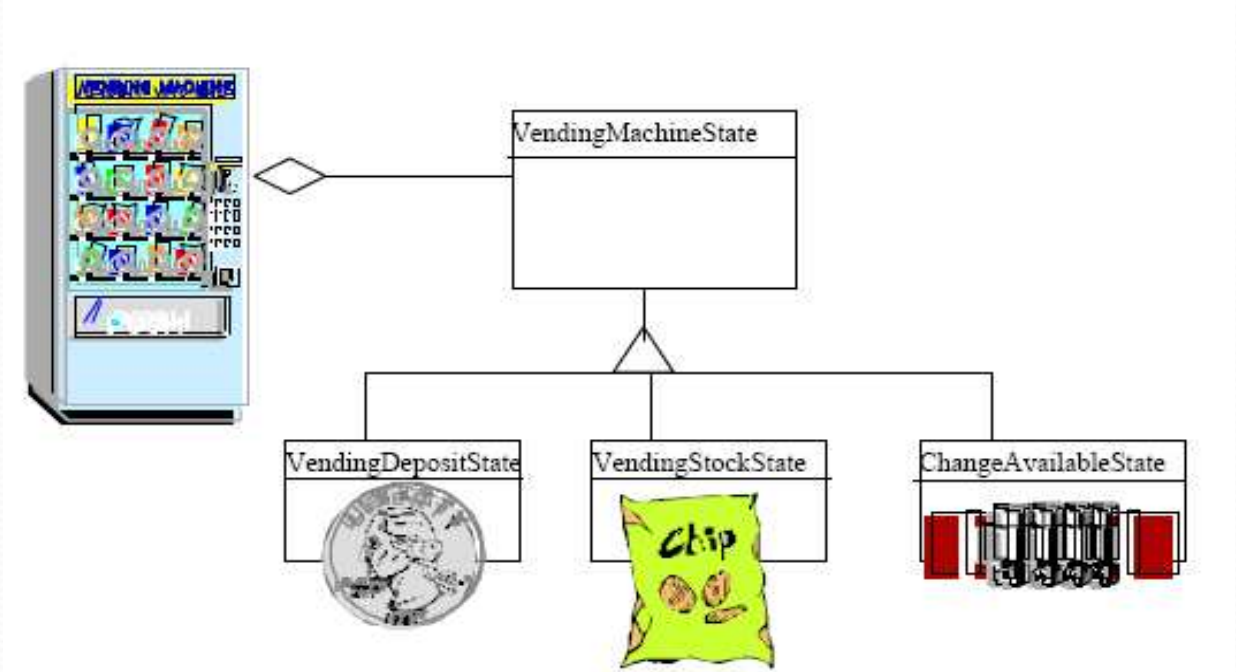
- *Builder*
  - *Para construir objetos complexos em várias etapas e/ou que possuem representações diferentes*
- *Factory Method*
  - *Para isolar a classe concreta do produto criado da interface usada pelo cliente*
- *Abstract Factory*
  - *Para criar famílias inteiras de objetos que têm algo em comum sem especificar suas interfaces.*
- *Prototype*
  - *Para criar objetos usando outro como base*
- *Memento*
  - *Para armazenar o estado de um objeto sem quebrar o encapsulamento. O uso típico deste padrão é na implementação de operações de Undo.*

State

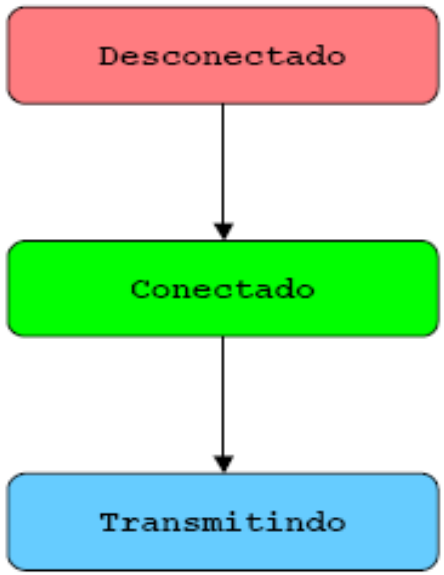
Objetivo:

"Permitir a um objeto alterar o seu comportamento quanto o seu estado interno mudar. O objeto irá aparentar mudar de classe." [GoF]

Analogia


VendingMachineState
VendingDepositState
VendingStockState
ChangeAvailableState
Chip

Motivação



:Objeto

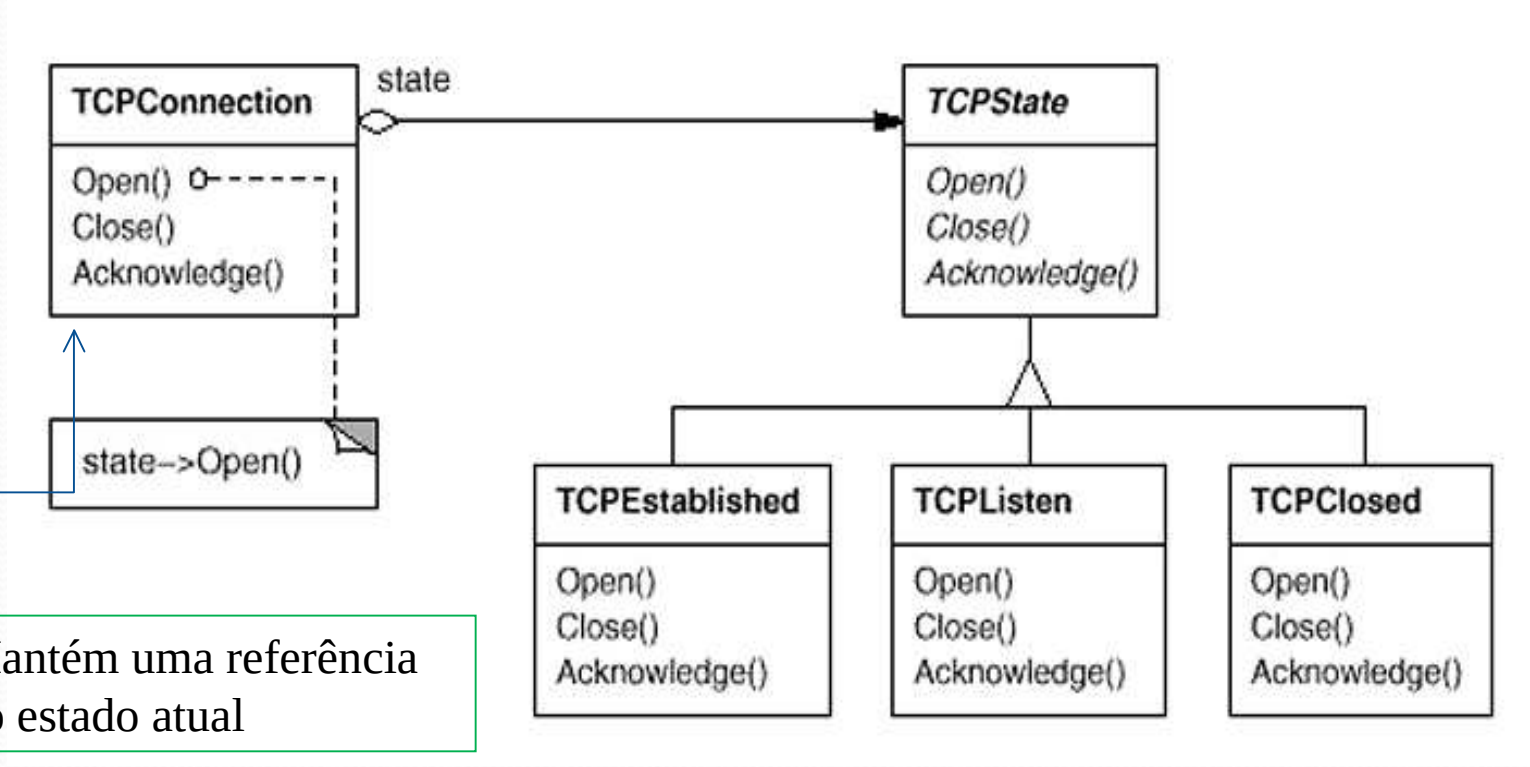
operação

```
if (estado == desconectado) {
  façaIsto();
} else if (estado == conectado) {
  façaAquilo();
} else {
  faça();
}
```

operação

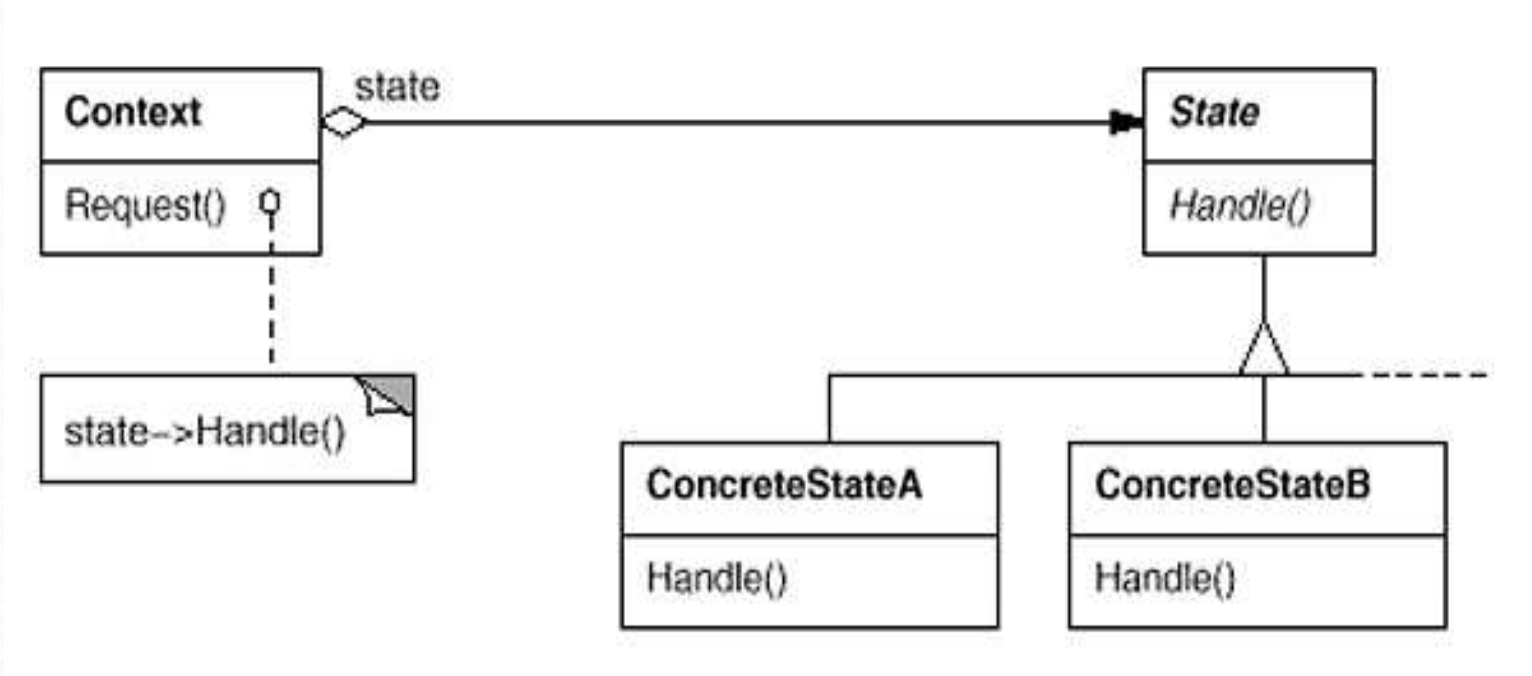
```
estado.faça()
```

Solução - State



Mantém uma referência ao estado atual

Estrutura


Context
Request()
state
State
Handle()
state->Handle()
ConcreteStateA
Handle()
ConcreteStateB
Handle()

Participantes

- **Context** (TCPConnection)
  - Mantém uma instancia de um ConcreteState que define o estado atual
- **State** (TCPState)
  - Define uma interface para encapsular o comportamento associado com um estado particular de um Contexto
- **ConcreteState subclasses** (TCPEstablished, TCPListen, TCPClosed)
  - Cada subclasse implementa um comportamento associado ao estado do Contexto

Aplicabilidade

- Quando o comportamento de um objeto depende do seu estado, que é alterado em tempo de execução;
- Quando operações de um objeto possuem condicionais grandes e com muitas partes (sintoma do caso anterior).

Conseqüências

- Separa comportamento dependente de estado:
  - Novos estados/comportamentos podem ser facilmente adicionados.
- Transição de estados é explícita:
  - Fica claro no diagrama de classes os estados possíveis de um objeto.
- States podem ser compartilhados:
  - Somente se eles não armazenarem estado em atributos.

Implementação

```
class TCPOctetStream;
class TCPState;

class TCPConnection {
public:
    TCPConnection();

    void ActiveOpen();
    void PassiveOpen();
    void Close();

    void Send();
    void Acknowledge();
    void Synchronize();

    void ProcessOctet(TCPOctetStream*);
private:
    friend class TCPState;
    void ChangeState(TCPState*);
private:
    TCPState* _state;
};
```

```
class TCPState {
public:
    virtual void Transmit(TCPConnection*, TCPOctetStream*);
    virtual void ActiveOpen(TCPConnection*);
    virtual void PassiveOpen(TCPConnection*);
    virtual void Close(TCPConnection*);
    virtual void Synchronize(TCPConnection*);
    virtual void Acknowledge(TCPConnection*);
    virtual void Send(TCPConnection*);
protected:
    void ChangeState(TCPConnection*, TCPState*);
};
```

Implementação

```
TCPConnection::TCPConnection () {
    _state = TCPClosed::Instance();
}

void TCPConnection::ChangeState (TCPState* s) {
    _state = s;
}

void TCPConnection::ActiveOpen () {
    _state->ActiveOpen(this);
}

void TCPConnection::PassiveOpen () {
    _state->PassiveOpen(this);
}

void TCPConnection::Close () {
    _state->Close(this);
}

void TCPConnection::Acknowledge () {
    _state->Acknowledge(this);
}

void TCPConnection::Synchronize () {
    _state->Synchronize(this);
}
```

```
void TCPState::Transmit (TCPConnection*, TCPOctetStream*) { }
void TCPState::ActiveOpen (TCPConnection*) { }
void TCPState::PassiveOpen (TCPConnection*) { }
void TCPState::Close (TCPConnection*) { }
void TCPState::Synchronize (TCPConnection*) { }

void TCPState::ChangeState (TCPConnection* t, TCPState* s) {
    t->ChangeState(s);
}
```

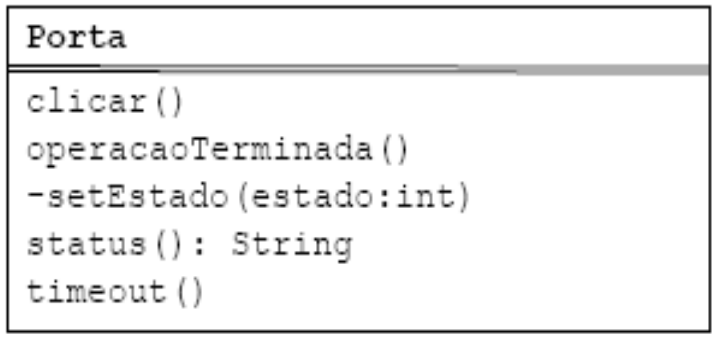
Implementação

```
class TCPEstablished : public TCPState {
public:
    static TCPState* Instance();

    virtual void Transmit(TCPConnection*, TCPOctetStream*);
    virtual void Close(TCPConnection*);
};

class TCPListen : public TCPState {
public:
    static TCPState* Instance();

    virtual void Send(TCPConnection*);
    // ...
};

class TCPClosed : public TCPState {
public:
    static TCPState* Instance();

    virtual void ActiveOpen(TCPConnection*);
    virtual void PassiveOpen(TCPConnection*);
    // ...
};
```

```
void TCPClosed::ActiveOpen (TCPConnection* t) {
    // send SYN, receive SYN, ACK, etc.

    ChangeState(t, TCPEstablished::Instance());
}

void TCPClosed::PassiveOpen (TCPConnection* t) {
    ChangeState(t, TCPListen::Instance());
}

void TCPEstablished::Close (TCPConnection* t) {
    // send FIN, receive ACK of FIN

    ChangeState(t, TCPListen::Instance());
}

void TCPEstablished::Transmit (
    TCPConnection* t, TCPOctetStream* o
) {
    t->ProcessOctet(o);
}

void TCPListen::Send (TCPConnection* t) {
    // send SYN, receive SYN, ACK, etc.

    ChangeState(t, TCPEstablished::Instance());
}
```

Exercício

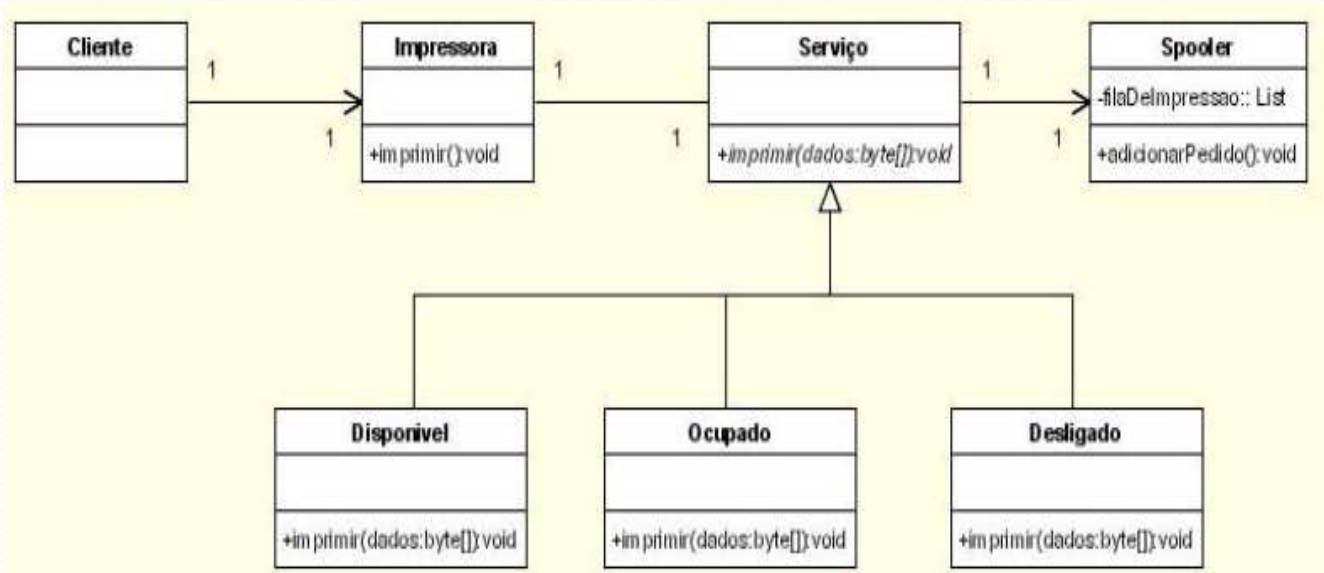
- *Refatore a classe Porta (representada em UML abaixo) para representar seus estados usando o State pattern*
- *A porta funciona com um botão que alterna os estados de aberta, abrindo, fechada, fechando, manter aberta.*
- *Execute a aplicação e teste seu funcionamento*



Exercício

Um sistema de impressão utiliza o objeto Serviço para controlar quando, ao receber uma ordem de impressão, uma tarefa será enviada diretamente para a impressora ou para a fila de impressão. O diagrama UML abaixo ilustra o modelo de implementação usado. Que padrão de design foi utilizado?

a)Command
b)Interpreter
c)State
d)Strategy
e)Chain of Responsibility



Exercício

Um sistema de impressão utiliza o objeto Serviço para controlar quando, ao receber uma ordem de impressão, uma tarefa será enviada diretamente para a impressora ou para a fila de impressão. O diagrama UML abaixo ilustra o modelo de implementação usado. Que padrão de design foi utilizado?

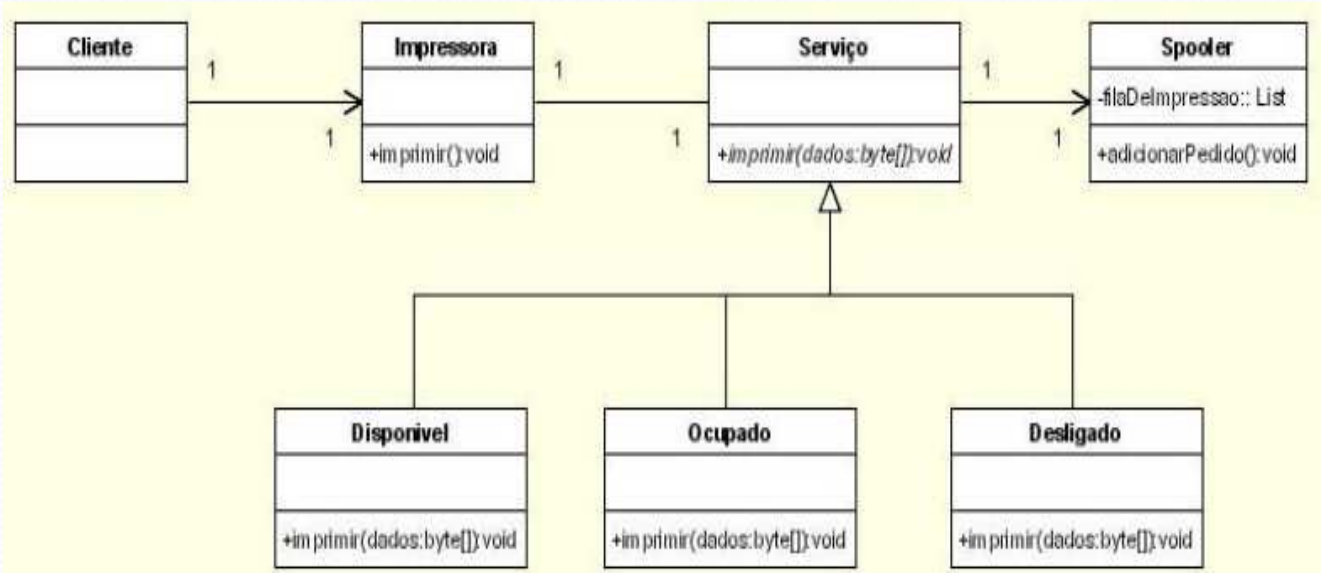
a)Command

b)Interpreter

c)State

d)Strategy

e)Chain of Responsibility

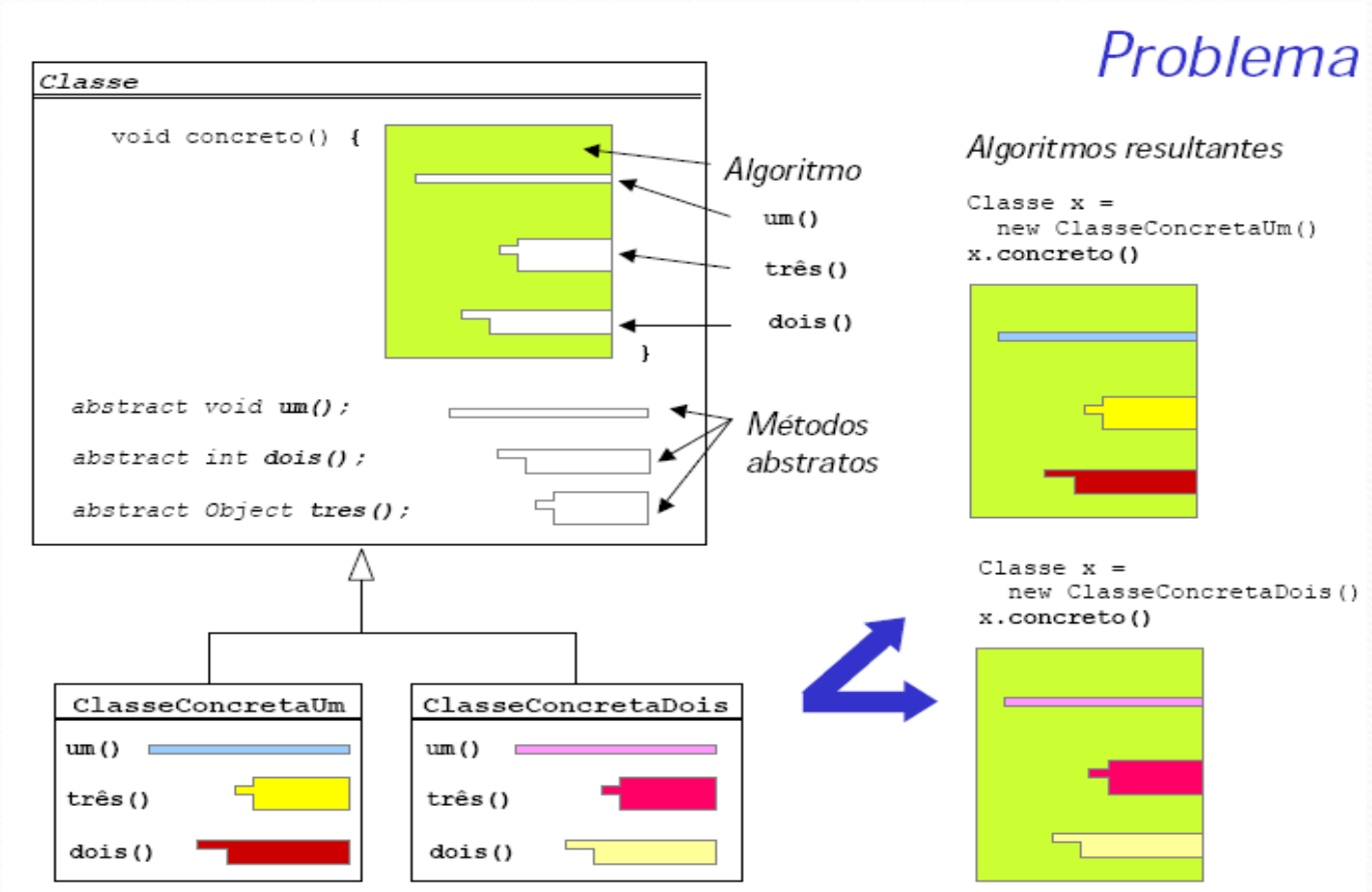


Template Method

Objetivo:

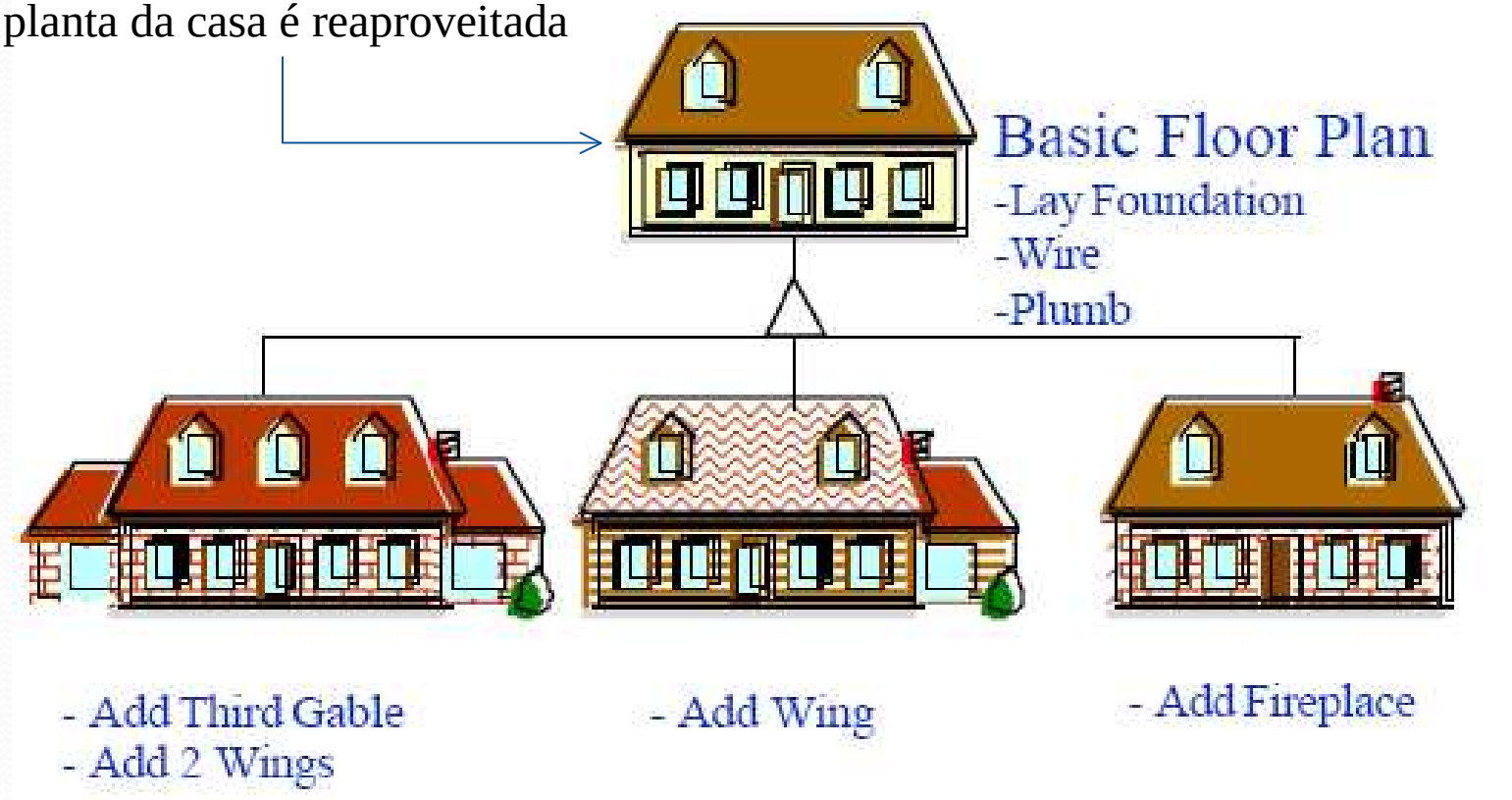
"Definir o esqueleto de um algoritmo dentro de uma operação, deixando alguns passos a serem preenchidos pelas subclasses. Template Method permite que suas subclasses redefinam certos passos de um algoritmo sem mudar sua estrutura." [GoF]

Analogia

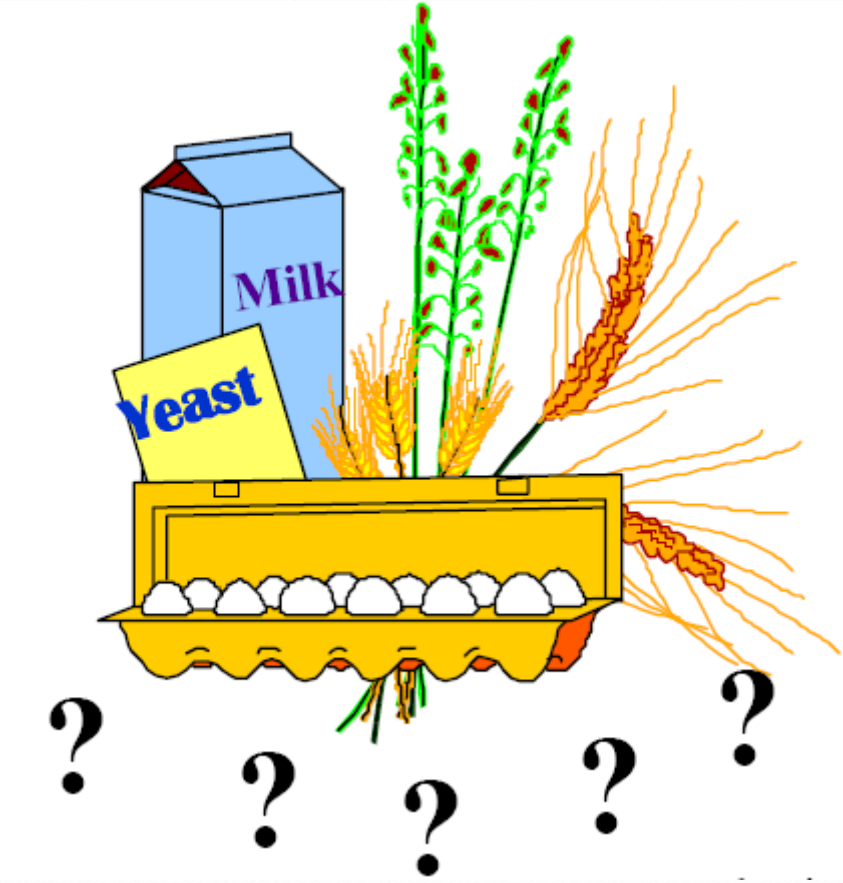

Problema
Classe
void concreto() {
Algoritmo
um()
três()
dois()
}
abstract void um();
abstract int dois();
abstract Object tres();
Métodos abstratos
Algoritmos resultantes
Classe x =
new ClasseConcretaUm()
x.concreto()
Classe x =
new ClasseConcretaDois()
x.concreto()
ClasseConcretaUm
um()
três()
dois()
ClasseConcretaDois
um()
três()
dois()

Analogia

A planta da casa é reaproveitada



Analogia

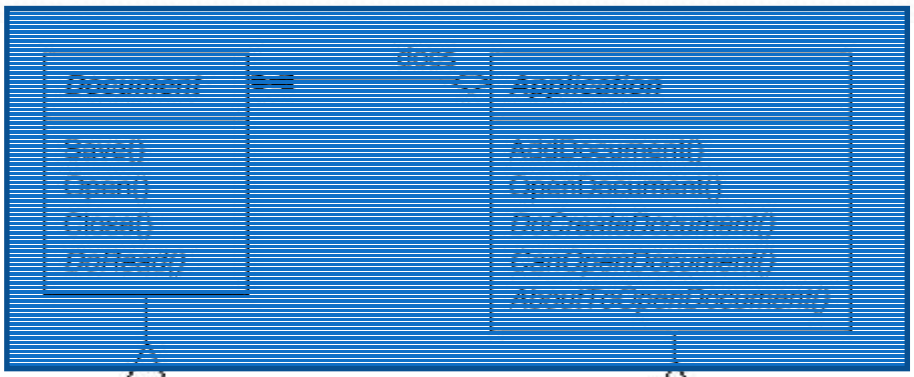


Receitas de pães são reutilizadas para inventar novas receitas

- Adicionar queijo, nozes, canela...
- Todas elas compartilham uma receita em comum

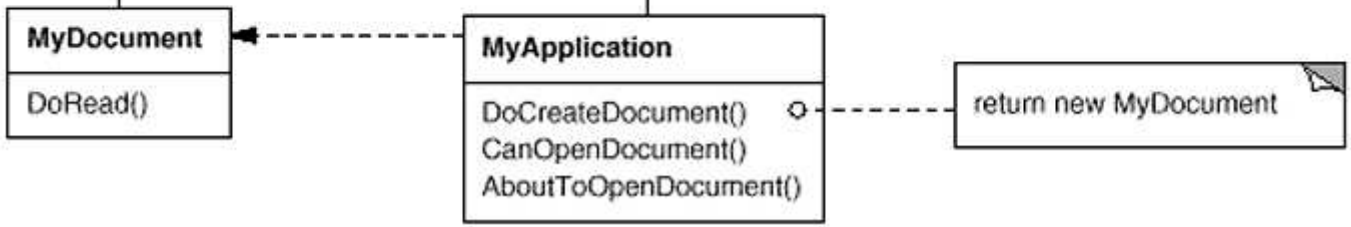
Motivação

- Considere um *framework* com as classes:
  - Application e Document



Aplicações específicas podem instanciar este framework

- Planilha eletrônica
- Aplicações de desenho



Motivação

Define o esqueleto do método

Definido na classe abstrata

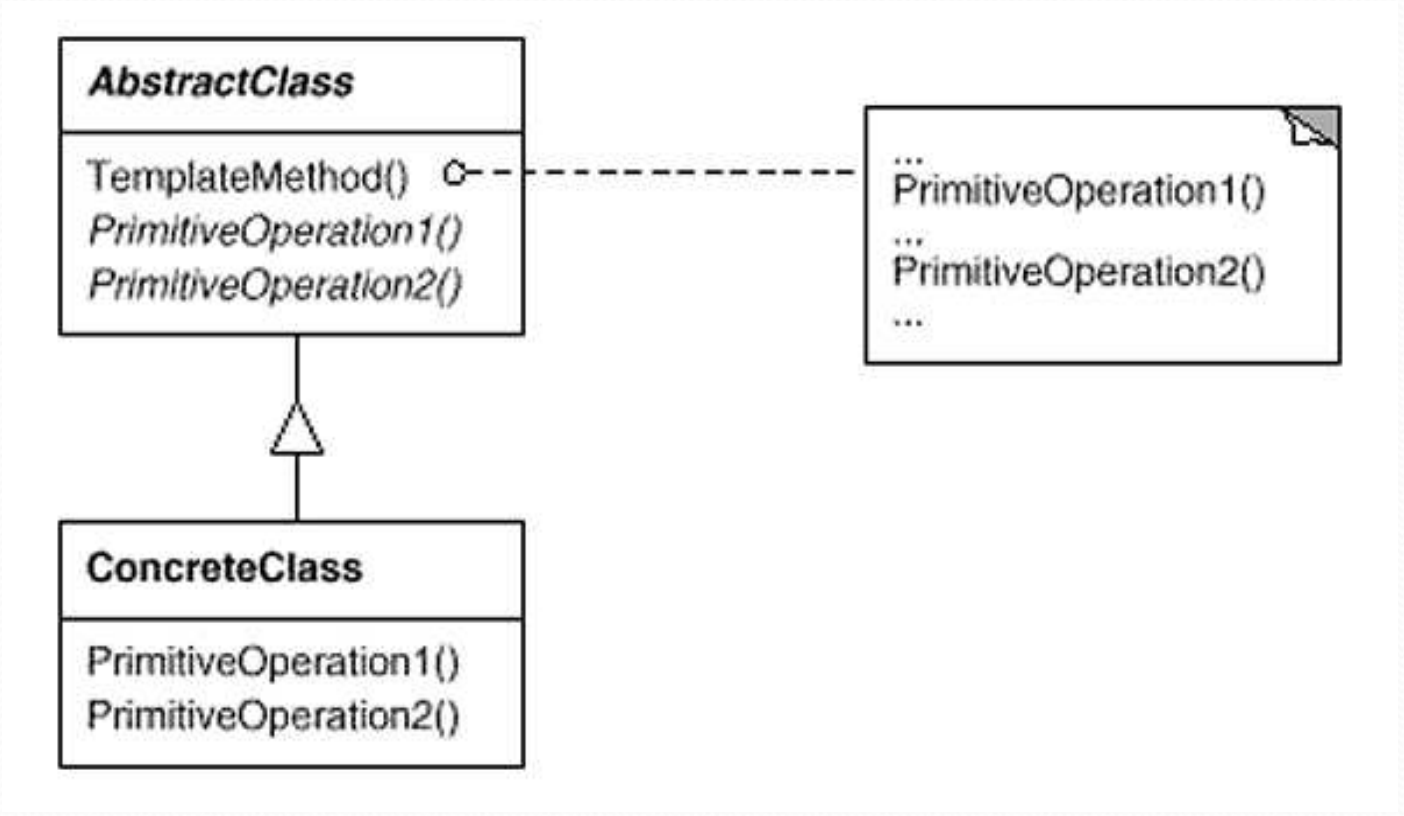
Métodos abstratos que deverão ser implementados pela subclasse

```
void Application::OpenDocument (const char* name) {
    if (!CanOpenDocument(name)) {
        // cannot handle this document
        return;
    }

    Document* doc = DoCreateDocument();

    if (doc) {
        _docs->AddDocument(doc);
        AboutToOpenDocument(doc);
        doc->Open();
        doc->DoRead();
    }
}
```

Solução – Template Method

- O que é um Template Method
  - Um Template Method define um algoritmo em termos de operações abstratas que subclasses sobrepõem para oferecer comportamento concreto
- Quando usar?
  - Quando a estrutura fixa de um algoritmo puder ser definida pela superclasse deixando certas partes para serem preenchidos por implementações que podem variar

Estrutura


AbstractClass
TemplateMethod()
PrimitiveOperation1()
PrimitiveOperation2()
...
PrimitiveOperation1()
...
PrimitiveOperation2()
...
ConcreteClass
PrimitiveOperation1()
PrimitiveOperation2()

Participantes

- **AbstractClass** (Application)
  - Define o esqueleto de um algoritmo
  - Define operações primitivas abstratas
    - Classes concretas implementarão estes métodos para seguir os passos do algoritmo
- **ConcreteClass** (MyApplication)
  - Implementa as operações primitivas

Aplicabilidade

- Quando quiser implementar partes invariantes de um algoritmo na superclasse e deixar o restante para as subclasses;
- Quando o comportamento comum entre subclasses precisa ser generalizado para evitar duplicidade de código;
- Quando quiser controlar o que as subclasses podem estender (métodos hooks ou virtuais).

Métodos hooks

Sobrecarga não-obrigatória

```
void DerivedClass::Operation () {
    // DerivedClass extended behavior
    ParentClass::Operation();
}
```

Pode-se esquecer de chamar o método pai

```
void ParentClass::Operation () {
    // ParentClass behavior
    HookOperation();
}
```

Deixar métodos a serem implementados por subclasses

```
void ParentClass::HookOperation () { }
```

Não faz nada

```
void DerivedClass::HookOperation () {
     // derived class extension
}
```

Sobrecarga

Conseqüências

- Reuso de código:
  - Partes de um algoritmo são reutilizadas por todas as subclasses.
- Controle:
  - É possível controlar o que as subclasses podem estender (métodos finais).
- Comportamento padrão extensível:
  - Superclasse pode definir o comportamento padrão e permitir sobrescrita.

Implementação

```
void View::Display () {
    SetFocus();
    DoDisplay();
    ResetFocus();
}
```

Já implementado. Organiza as fontes e cores.

Método hook

```
void View::DoDisplay () { }
```

```
void MyView::DoDisplay () {
    // render the view's contents
}
```

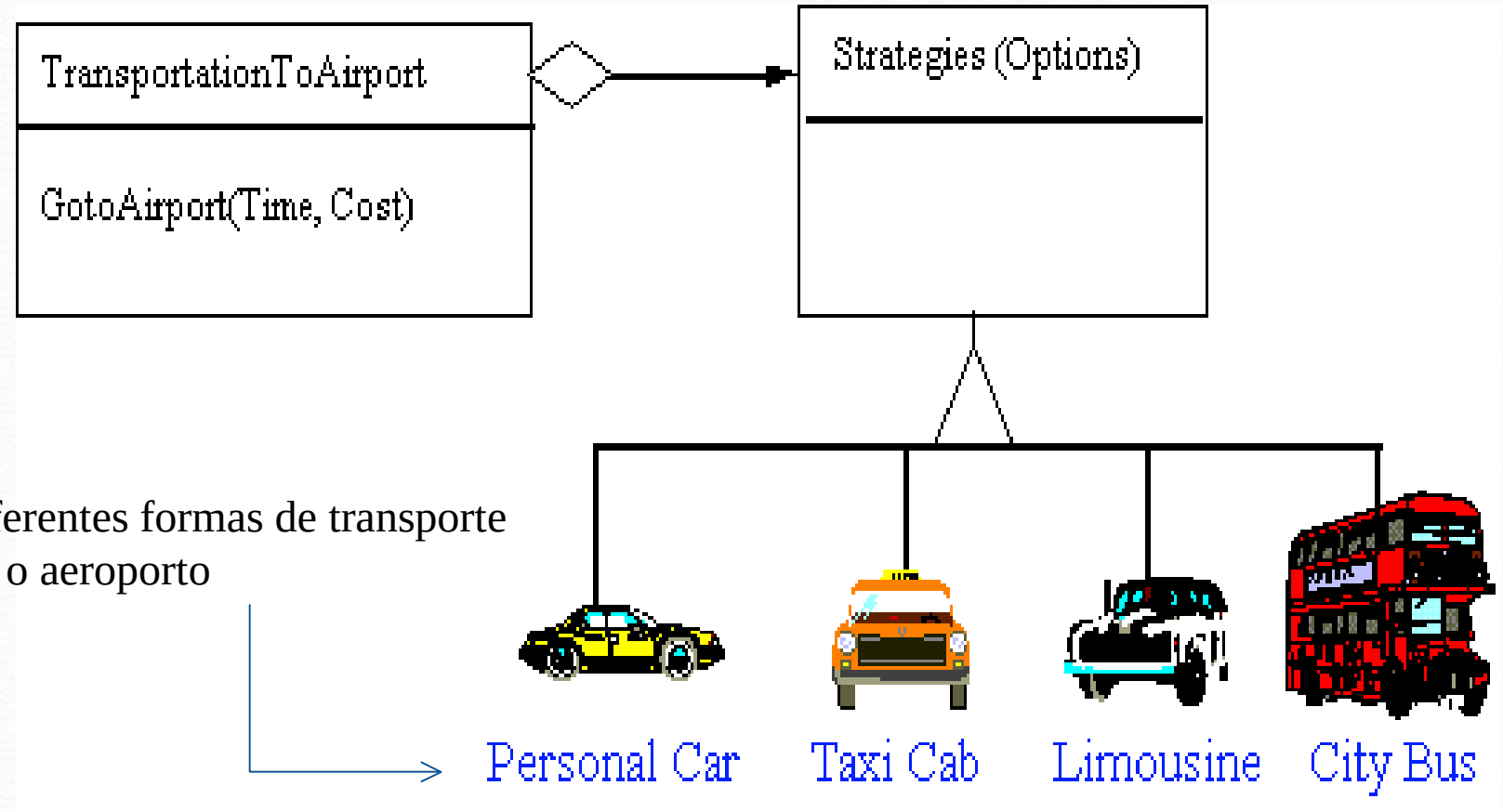
Implementação pela subclasse MyView

Strategy

Objetivo:

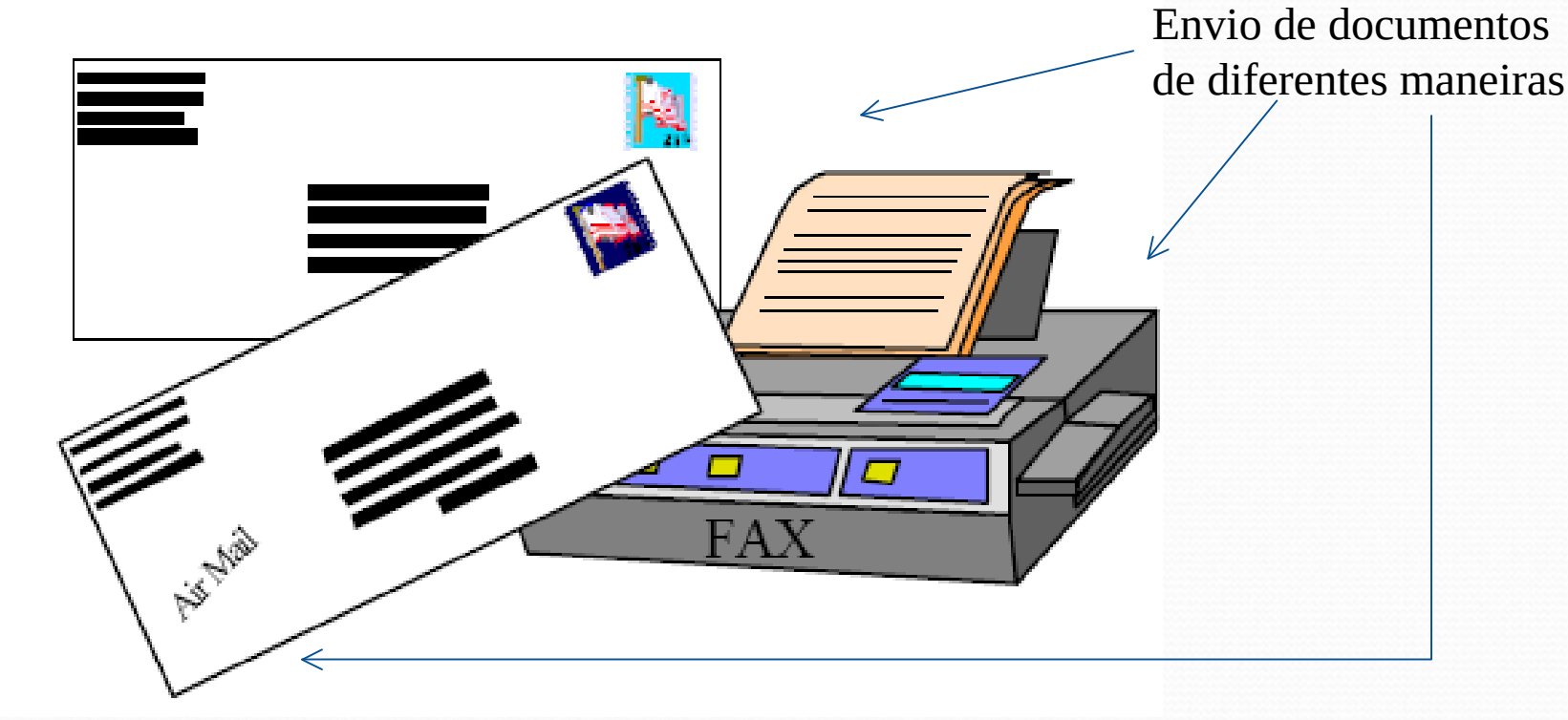
"Definir uma família de algoritmos, encapsular cada um, e fazê-los intercambiáveis. Strategy permite que algoritmos mudem independentemente entre clientes que os utilizam." [GoF]

Analogia



Diferentes formas de transporte até o aeroporto

Analogia



Algoritmos são independentes de contexto: a proximidade não dita a estratégia a ser usada – enviar a um vizinho

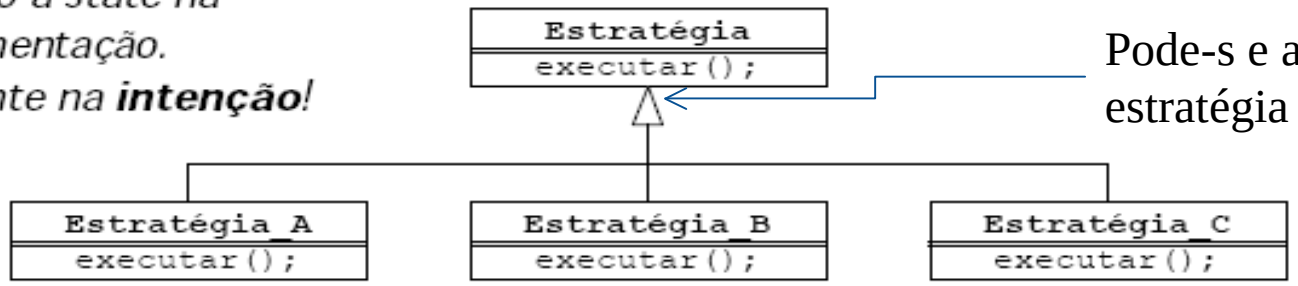
Exemplo

Várias estratégias, escolhidas de acordo com opções ou condições

```
if (guerra && inflação > META) {
   doPlanoB();
else if (guerra && recessão) {
   doPlanoC();
} else {
   doPlanejado();
}
```

```
if (guerra && inflação > META) {
   plano = new Estrategia_C();
} else if (guerra && recessão) {
   plano = new Estrategia_B();
} else {
   plano = new Estrategia_A();
}

plano.executar();
```

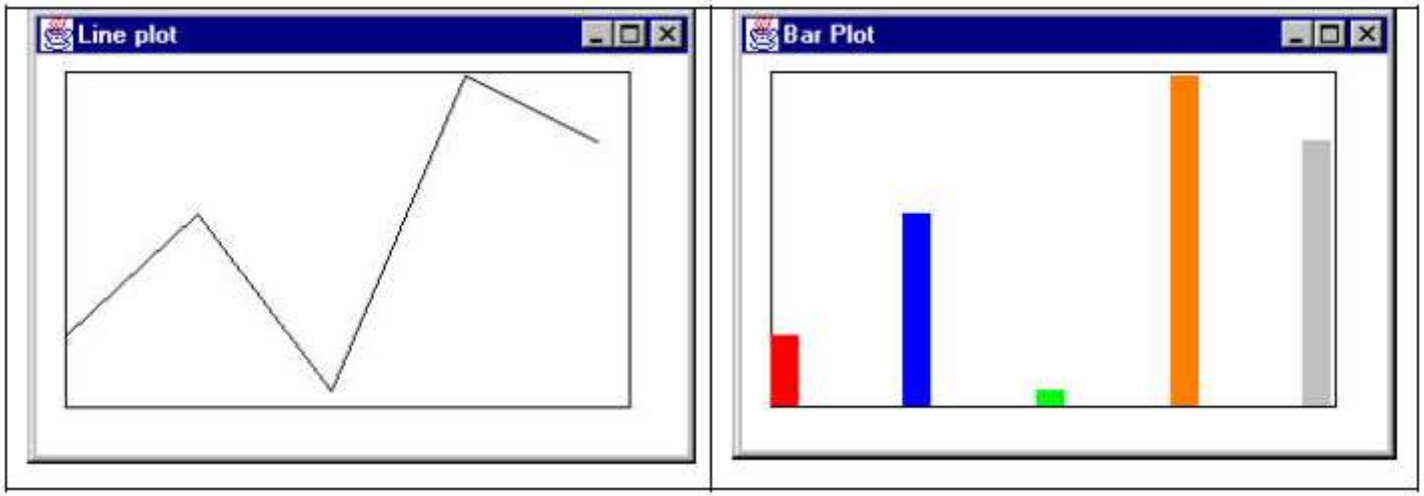
Idêntico a state na implementação.
*Diferente na **intenção**!*



Pode-s e adicionar novas estratégia facilmente

Alguns usos...

- Salvar um arquivo em diferentes formatos...
- Comprimir arquivos usando diferentes algoritmos...
- Apresentar o mesmo dado em diferentes tipos de gráficos: gráfico de setores, de barras, linhas...



- Usar diferentes modos de quebra de linha para mostrar um texto

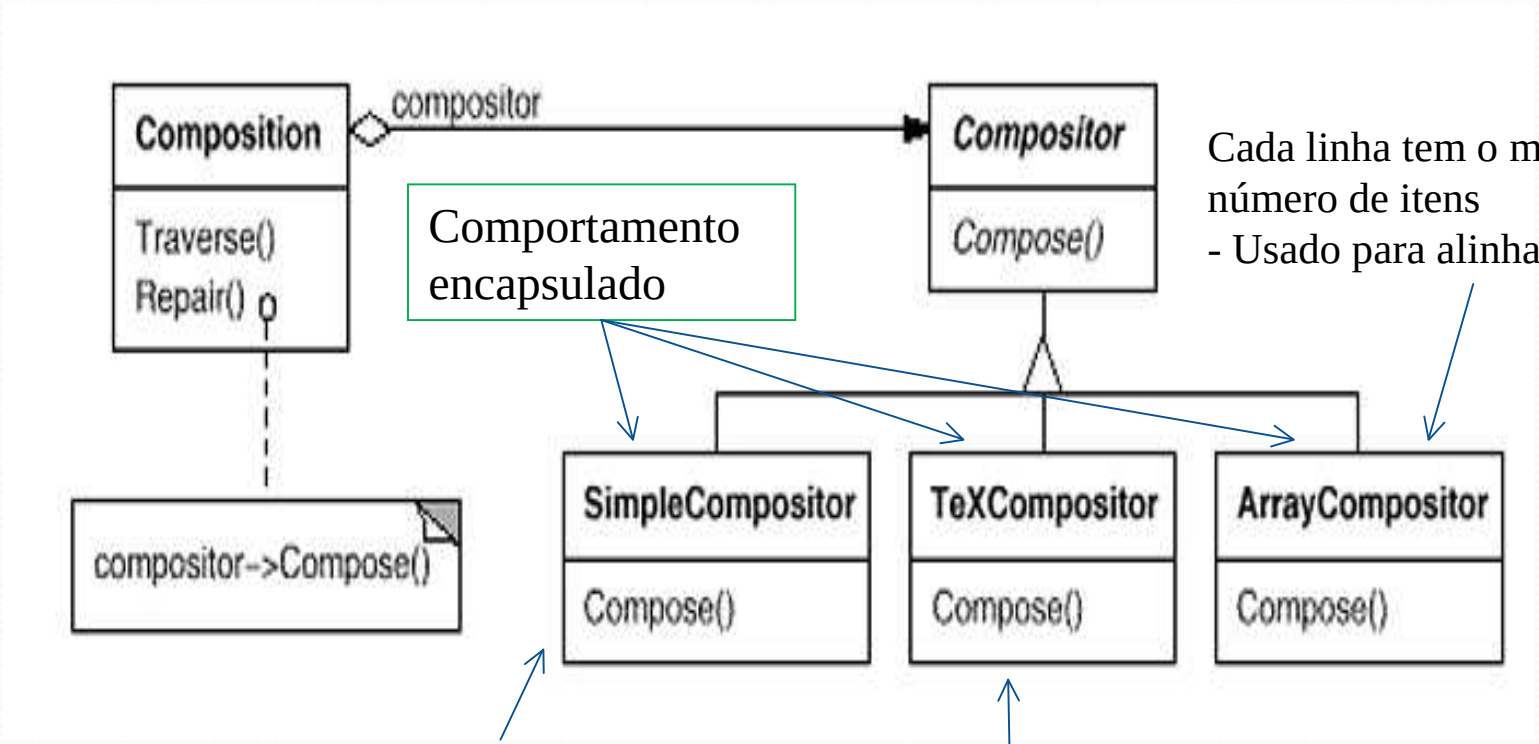
Motivação

- Existem problemas que possuem vários algoritmos que os solucionam;
  - Ex.: quebrar um texto em linhas.
- Seria interessante:
  - Separar estes algoritmos em classes específicas para serem reutilizados;
  - Permitir que sejam intercambiados e que novos algoritmos sejam adicionados com facilidade.

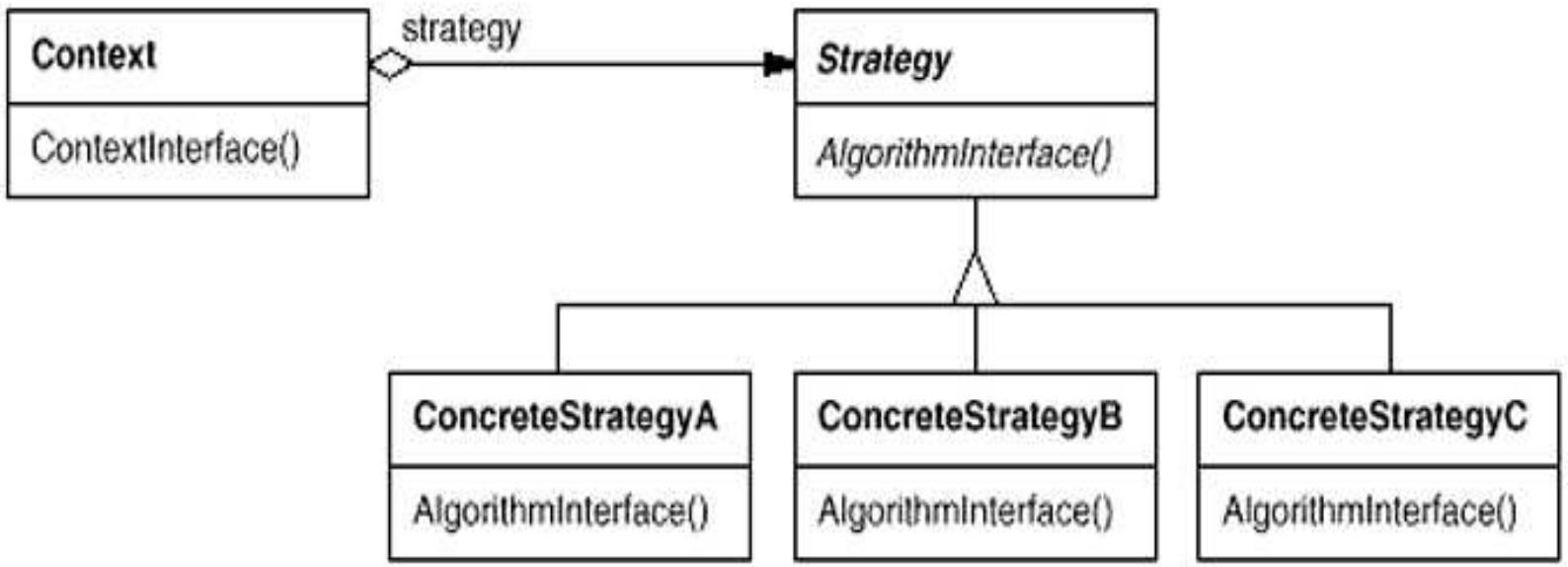
Motivação

- Quebrar um texto em linhas
- Clientes ficam mais complexos se manterem vários algoritmos de quebra de texto
  - Dificulta a manutenção
- É inviável manter vários algoritmos se usa-se somente um deles por vez
- Se todas as estratégias estão no cliente
  - Torna-se difícil modificar o comportamento ou adicionar novos algoritmos

Solução - Strategy



Estrutura


Context
ContextInterface()
strategy
Strategy
AlgorithmInterface()
ConcreteStrategyA
AlgorithmInterface()
ConcreteStrategyB
AlgorithmInterface()
ConcreteStrategyC
AlgorithmInterface()

Participantes

- **Strategy** (Compositor)
  - Declara uma interface comum a todos os algoritmos suportados
  - Context usa esta interface para invocar os algoritmos
- **ConcreteStrategy** (SimpleCompositor, TeXCompositor, ArrayCompositor)
  - Implementa os algoritmos usando a interface Strategy
- **Context** (Composition)
  - Possui referencia a um objeto Strategy
  - É configurado com um ConcreteStrategy

Aplicabilidade

- Quando você precisa de variantes de um mesmo algoritmo;
- Quando um algoritmo utiliza dados que o cliente não precisa conhecer;
- Quando uma classe define múltiplos comportamentos, escolhidos num grande condicional.

Conseqüências

- Famílias de algoritmos:
  - Beneficiam-se de herança e polimorfismo.
  - Reuso
- Alternativa para herança:
  - Especializar o Contexto o torna difícil de entender
    - Não dá para variar de algoritmo dinamicamente
  - Com Strategy: Comportamento é a única coisa que varia.
- Eliminam os grandes condicionais:
  - Evita código monolítico.
- Escolha de implementações:
  - Pode alterar a estratégia em runtime.

Eliminação de declarações condicionais

```
void Composition::Repair () {
    switch (_breakingStrategy) {
    case SimpleStrategy:
        ComposeWithSimpleCompositor();
        break;
    case TeXStrategy:
        ComposeWithTeXCompositor();
        break;
    // ...
    }
    // merge results with existing composition, if necessary
}
```

Sem Strategy

Com Strategy

```
void Composition::Repair () {
    _compositor->Compose();
    // merge results with existing composition, if necessary
}
```

Conseqüências

- Clientes devem conhecer as estratégias:
  - Eles que escolhem qual usar a cada momento.
  - É preciso entender a implementação do algoritmo
- Parâmetros diferentes para algoritmos diferentes:
  - Algoritmos simples x complexos
  - Todos implementam a interface Strategy
  - Algoritmos simples não utilizam todos os parâmetros da interface Strategy
- Aumenta o número de objetos:
  - Este padrão aumenta a quantidade de objetos pequenos presentes na aplicação.

Implementação

```
void Composition::Repair () {
    Coord* natural;
    Coord* stretchability;
    Coord* shrinkability;
    int componentCount;
    int* breaks;

    // prepare the arrays with the desired
    // ...

    // determine where the breaks are:
    int breakCount;
    breakCount = _compositor->Compose(
        natural, stretchability, shrinkability,
        componentCount, _lineWidth, breaks
    );

    // lay out components according to breaks
    // ...
}
```

```
class Composition {
public:
    Composition(Compositor*);
    void Repair();
private:
    Compositor* _compositor;
    Component* _components;      // the list of components
    int _componentCount;         // the number of components
    int _lineWidth;              // the Composition's line width
    int* _lineBreaks;            // the position of linebreaks
                                 // in components
    int _lineCount;              // the number of lines
};
```

Texto e Figuras

Implementação

```
class ArrayCompositor : public Compositor {
public:
    ArrayCompositor(int interval);

    virtual int Compose(
        Coord natural[], Coord stretch[], Coord shrink[],
        int componentCount, int lineWidth, int breaks[]
    );
    // ...
};
```

```
Composition* quick = new Composition(new SimpleCompositor);
Composition* slick = new Composition(new TeXCompositor);
Composition* iconic = new Composition(new ArrayCompositor(100));
```

```
class SimpleCompositor : public Compositor {
public:
    SimpleCompositor();

    virtual int Compose(
        Coord natural[], Coord stretch[], Coord shrink[],
        int componentCount, int lineWidth, int breaks[]
    );
    // ...
};
```

```
class TeXCompositor : public Compositor {
public:
    TeXCompositor();

    virtual int Compose(
        Coord natural[], Coord stretch[], Coord shrink[],
        int componentCount, int lineWidth, int breaks[]
    );
    // ...
};
```

```
class Compositor {
public:
    virtual int Compose(
        Coord natural[], Coord stretch[], Coord shrink[],
        int componentCount, int lineWidth, int breaks[]
    ) = 0;
protected:
    Compositor();
};
```

Strategy em Java

```
public class Guerra {
    Estrategia acao;
    public void definirEstrategia() {
      if (inimigo.exercito() > 10000) {
         acao = new AliancaVizinho();
      } else if (inimigo.isNuclear()) {
         acao = new Diplomacia();
      } else if (inimigo.hasNoChance()) {
         acao = new AtacarSozinho();
      }
    }
    public void declararGuerra() {
       acao.atacar();
    }
    public void encerrarGuerra() {
       acao.concluir();
    }
}
```

```
public interface Estrategia {
    public void atacar();
    public void concluir();
}
```

```
public class AtacarSozinho
         implements Estrategia {
    public void atacar() {
       plantarEvidenciasFalsas();
       soltarBombas();
       derrubarGoverno();
    }
    public void concluir() {
       estabelecerGovernoAmigo();
    }
}
```

```
public class AliancaVizinho
         implements Estrategia {
    public void atacar() {
       vizinhoPeloNorte();
       atacarPeloSul();
       ...
    }
    public void concluir() {
       dividirBeneficios(...);
       dividirReconstrução(...);
    }
}
```

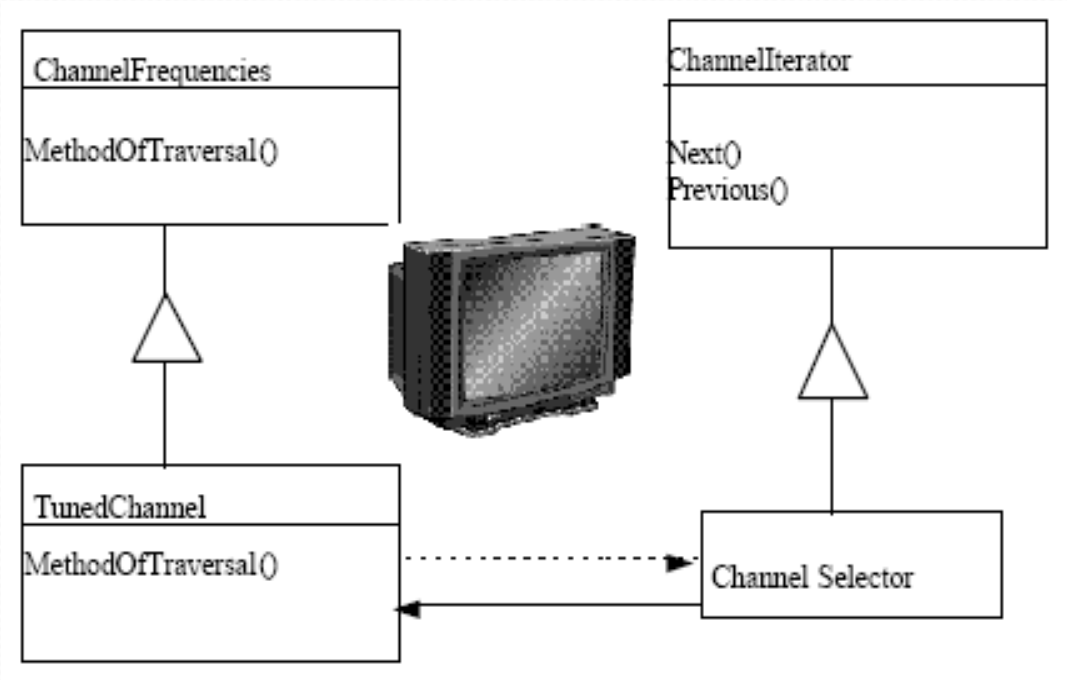
```
public class Diplomacia
         implements Estrategia {
    public void atacar() {
       recuarTropas();
       proporCooperacaoEconomica();
       ...
    }
    public void concluir() {
       desarmarInimigo();
    }
}
```

Iterator

Objetivo:

"Prover uma maneira de acessar os elementos de um objeto agregado seqüencialmente sem expor sua representação interna." [GoF]

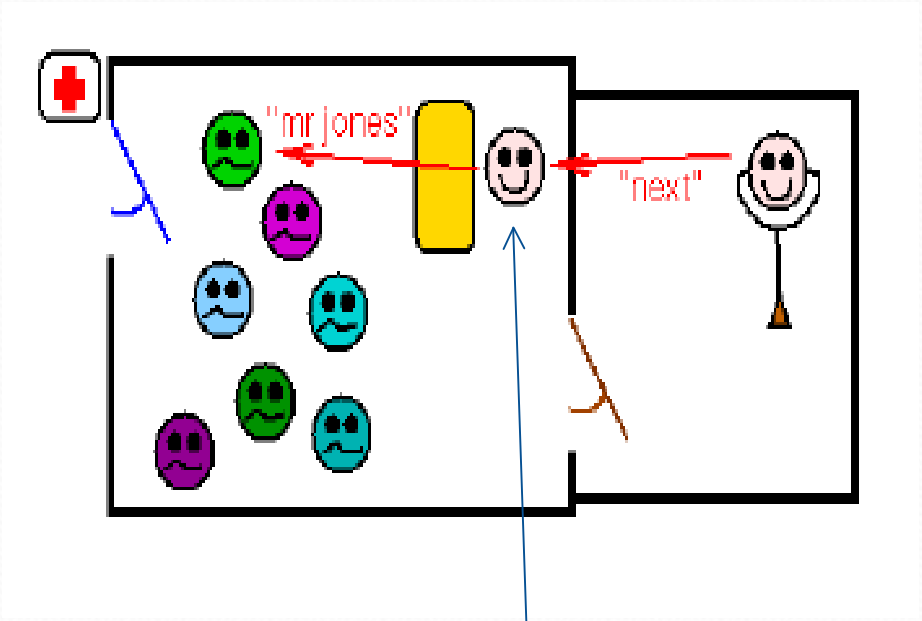
Analogia



O seletor de canais de televisão permite ao usuário percorrer os canais

• Filtra somente os canais disponíveis na região, diferente de antigamente

Analogia

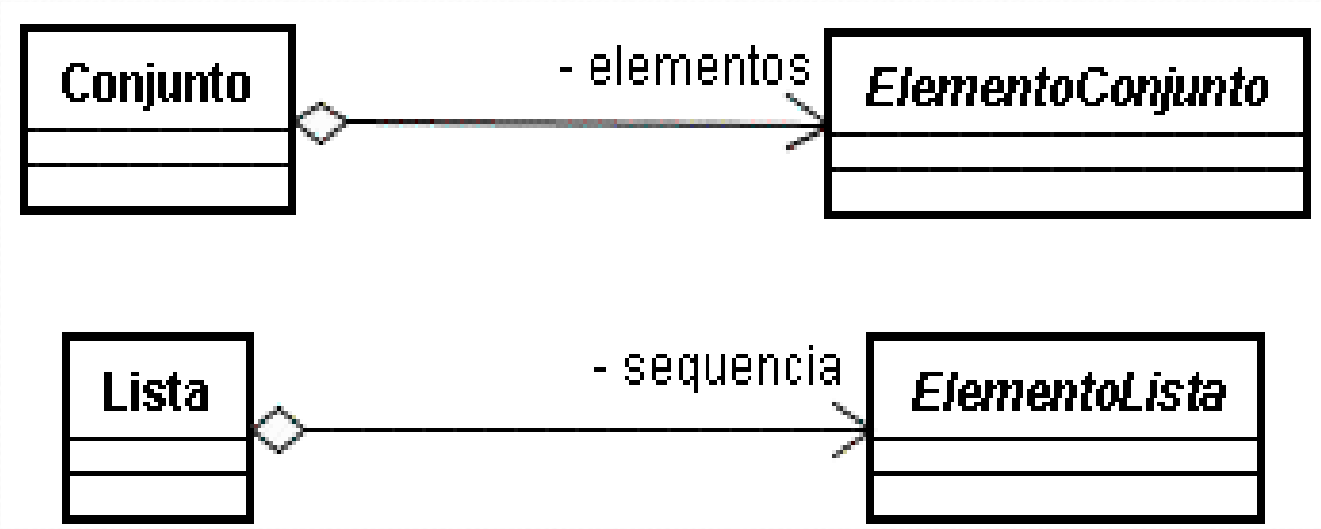


A secretária de um médico organiza a chamada dos pacientes que estão sentados espalhados pela sala. Ela sabe o próximo paciente.

Pode adotar diferentes políticas:
Ordem de chegada, estado do paciente...

Motivação

- Cliente precisa acessar os elementos;
- Cada coleção é diferente e não se quer expor a estrutura interna de cada um para o Cliente.



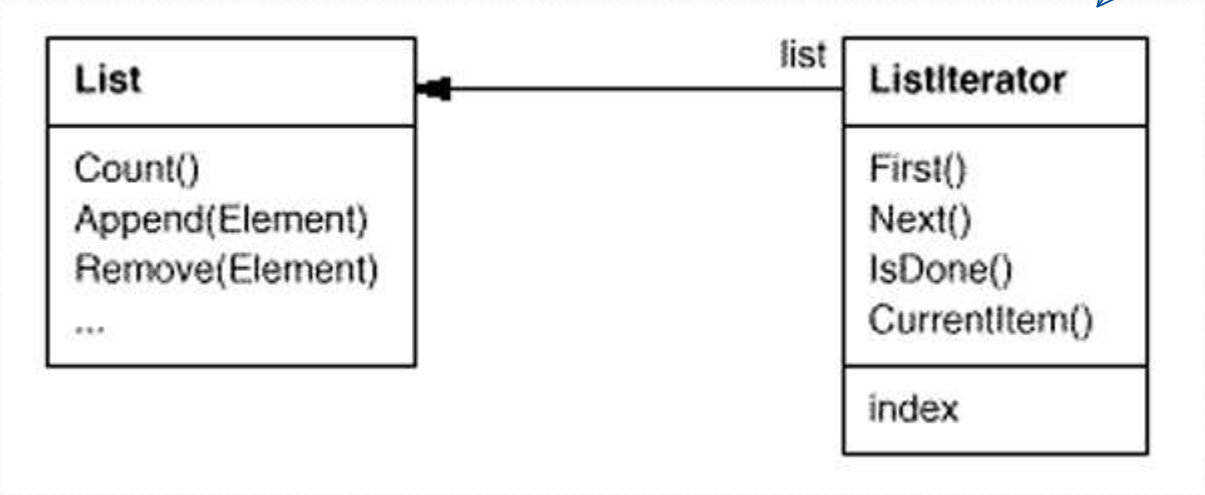
Motivação

- Cliente precisa acessar os elementos de uma coleção
- Não quer expor o tipo de coleção (i.e. lista, pilha, arvore...)
- Quer percorrer uma lista de diferentes formas (início-fim, fim-início)
- Não quer implementar os modos de percorrer dentro da lista
  - Não se pode antecipar todas as formas
- Quer acessar uma lista por dois ou mais objetos ao mesmo tempo

Solução - Iterator

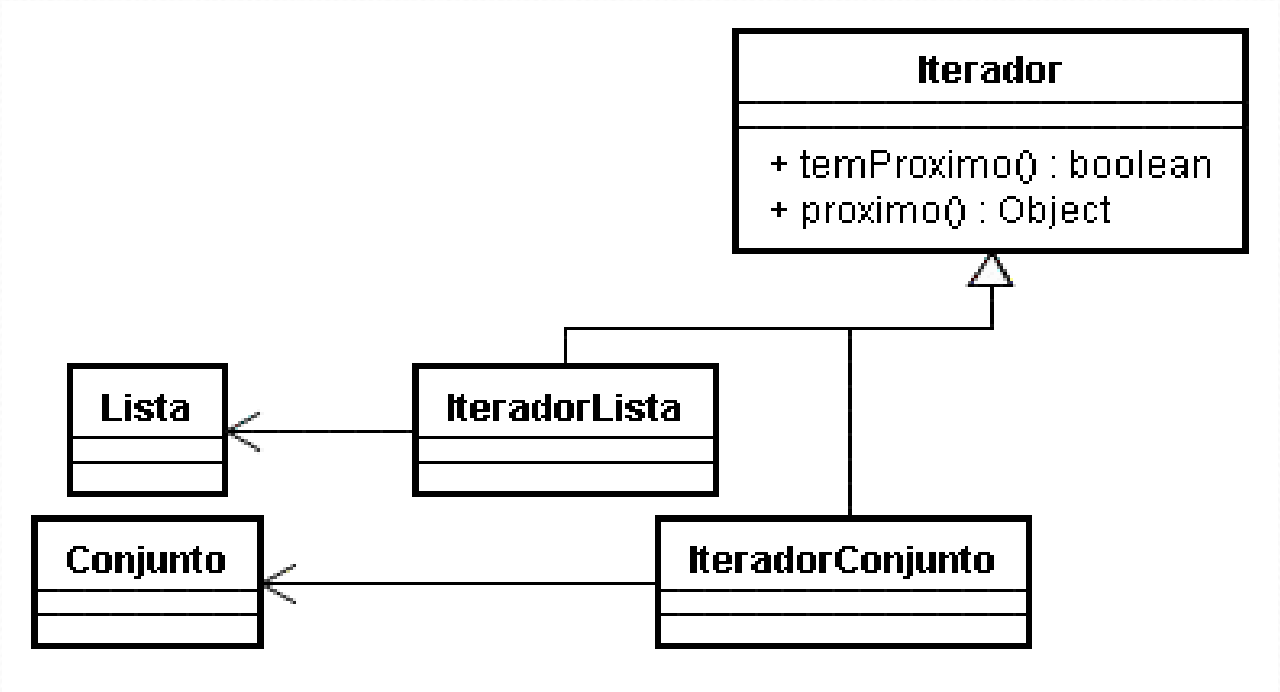
- Forte acoplamento
  - Cliente deve ser independente do tipo do Iterator

Diferentes implementações
Ex.: filtro

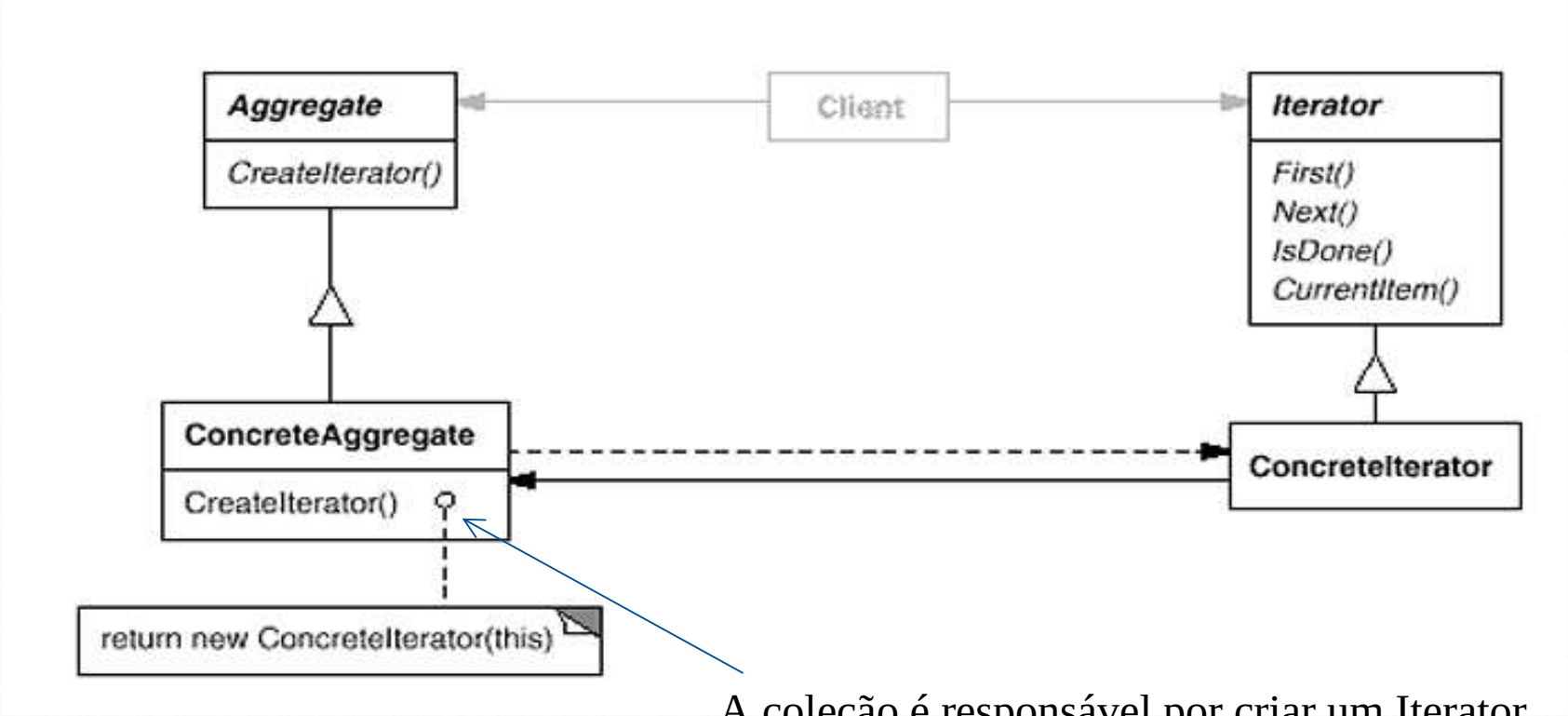


Solução - Iterator

- Iterator provê acesso seqüencial aos elementos, independente da coleção.
  - Pode trabalhar com as duas coleções sem mudar a interface

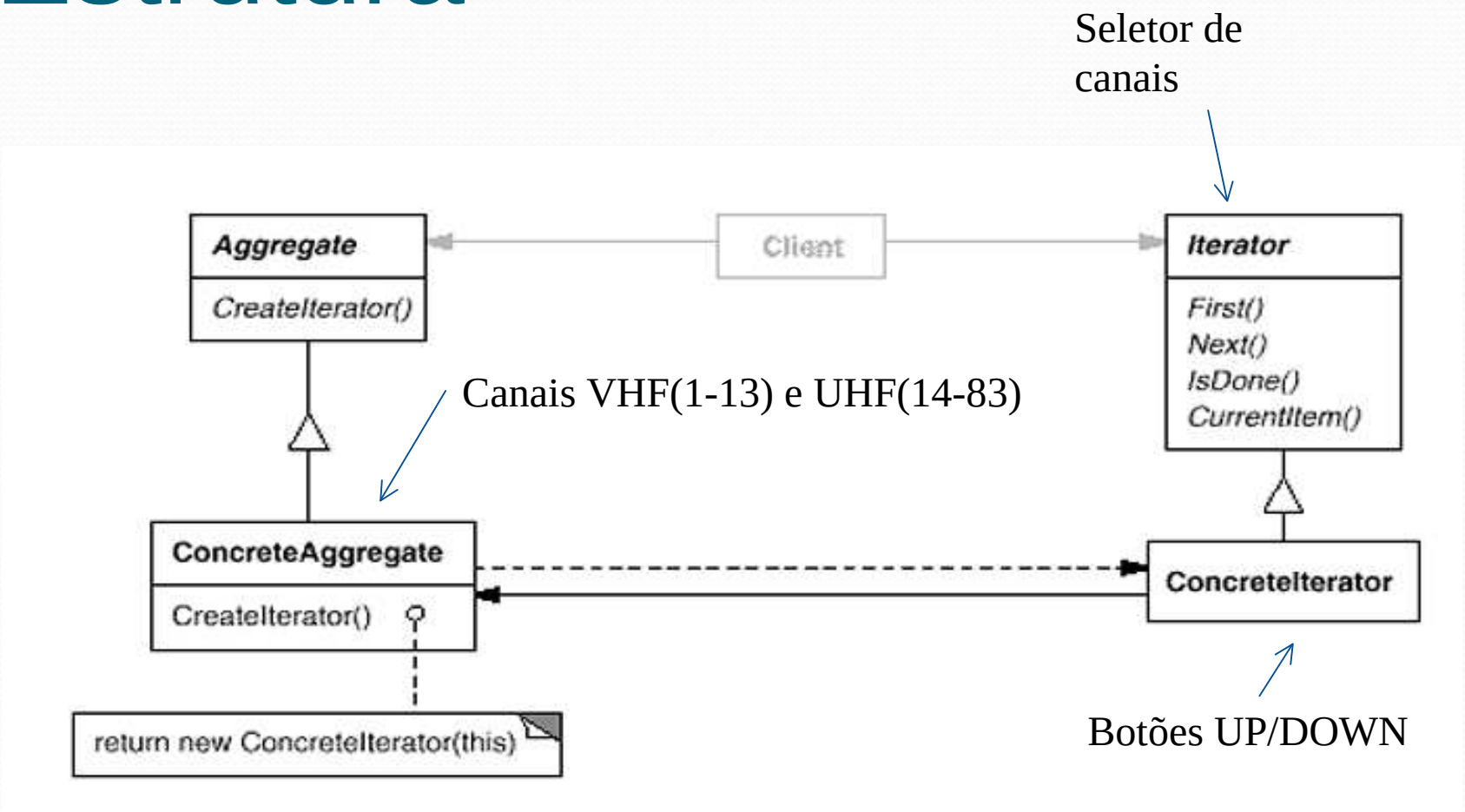


Estrutura



A coleção é responsável por criar um Iterator

Estrutura


Seletor de canais
Aggregate
CreateIterator()
Client
Iterator
First()
Next()
IsDone()
CurrentItem()
Canais VHF(1-13) e UHF(14-83)
ConcreteAggregate
CreateIterator()
ConcreteIterator
return new ConcreteIterator(this)
Botões UP/DOWN

Participantes

- Iterator
  - Define uma interface para acessar e percorrer elementos
- ConcreteIterator
  - Implementa a interface Iterator
  - Cuida da posição atual ao percorrer uma coleção
- Aggregate
  - Define uma interface para criar objetos Iterators
- ConcreteAggregate
  - Retorna uma instancia de um Iterator

Aplicabilidade

- Quando quiser acessar objetos agregados (coleções) sem expor a estrutura interna;
- Quando quiser prover diferentes meios de acessar tais objetos;
- Quando quiser especificar uma interface única e uniforme para este acesso.

Conseqüências

- Múltiplas formas de acesso:
  - Basta implementar um novo iterador com uma nova lógica de acesso.
- Interface simplificada:
  - Acesso é simples e uniforme para todos os tipos de coleções.
- Mais de um iterador:
  - É possível ter mais de um acesso à coleção em pontos diferentes.

Implementação

```
template <class Item>
class List {
public:
    List(long size = DEFAULT_LIST_CAPACITY);

    long Count() const;
    Item& Get(long index) const;
    // ...
};
```

```
template <class Item>
class Iterator {
public:
    virtual void First() = 0;
    virtual void Next() = 0;
    virtual bool IsDone() const = 0;
    virtual Item CurrentItem() const = 0;
protected:
    Iterator();
};
```

```
template <class Item>
class ListIterator : public Iterator<Item> {
public:
    ListIterator(const List<Item>* aList);
    virtual void First();
    virtual void Next();
    virtual bool IsDone() const;
    virtual Item CurrentItem() const;

private:
    const List<Item>* _list;
    long _current;
};
```

Implementação

```
template <class Item>
ListIterator<Item>::ListIterator (
    const List<Item>* aList
) : _list(aList), _current(0) {
}
```

```
template <class Item>
void ListIterator<Item>::First () {
    _current = 0;
}
```

```
template <class Item>
void ListIterator<Item>::Next () {
    _current++;
}
```

```
template <class Item>
bool ListIterator<Item>::IsDone () const {
    return _current >= _list->Count();
}
```

```
template <class Item>
Item ListIterator<Item>::CurrentItem () const {
    if (IsDone()) {
        throw IteratorOutOfBounds;
    }
    return _list->Get(_current);
}
```

```
template <class Item>
class ListIterator : public Iterator<Item> {
public:
    ListIterator(const List<Item>* aList);
    virtual void First();
    virtual void Next();
    virtual bool IsDone() const;
    virtual Item CurrentItem() const;

private:
    const List<Item>* _list;
    long _current;
};
```

Implementação

- Usando Iterators

```
void PrintEmployees (Iterator<Employee*>& i) {
    for (i.First(); !i.IsDone(); i.Next()) {
        i.CurrentItem()->Print();
    }
}
```

Imprimir uma lista de empregados

```
List<Employee*>* employees;
// ...
ListIterator<Employee*> forward(employees);
ReverseListIterator<Employee*> backward(employees);

PrintEmployees(forward);
PrintEmployees(backward);
```

Percorrer a lista em duas ordens:
Início–Fim
Fim-Início

Implementação

- Usando outros tipos de listas

Ruim -> Solução

```
SkipList<Employee*>* employees;
// ...

SkipListIterator<Employee*> iterator(employees);
PrintEmployees(iterator);
```

```
template <class Item>
class AbstractList {
public:
    virtual Iterator<Item>* CreateIterator() const = 0;
    // ...
};
```

Reutilizando o método

```
template <class Item>
Iterator<Item>* List<Item>::CreateIterator () const {
    return new ListIterator<Item>(this);
}
```

```
// we know only that we have an AbstractList
AbstractList<Employee*>* employees;
// ...

Iterator<Employee*>* iterator = employees->CreateIterator();
PrintEmployees(*iterator);
delete iterator;
```