

Design Patterns Estruturais e Comportamentais

USJT – 2019 – ECP06ANMCA – Arquitetura e Desenvolvimento de
Sistemas Multicamadas – Prof. Bonato

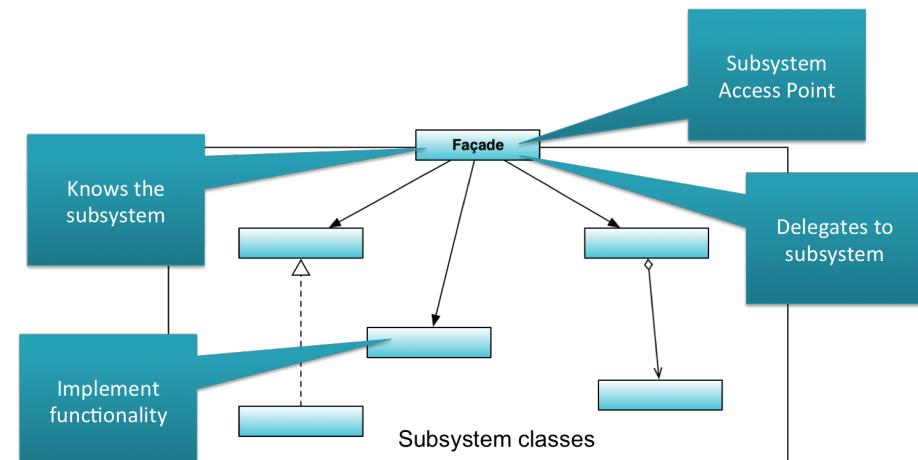
Padrões Estruturais

Objetivos e Estrutura

Facade

- **Objetivo**

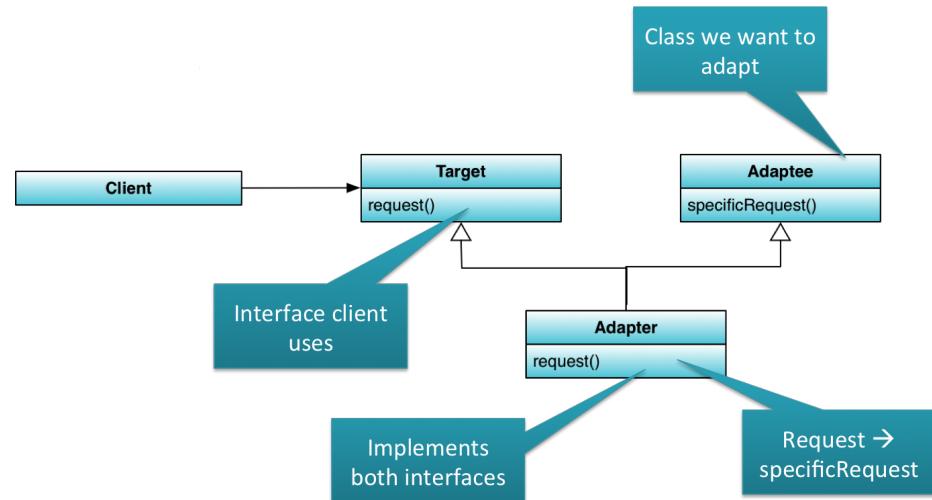
- desacoplar clientes dos subsistemas
- fornecer uma interface simplificada
- colocar subsistemas em camadas (negócio, dados e serviços clientes)



Adapter

- **Objetivo**

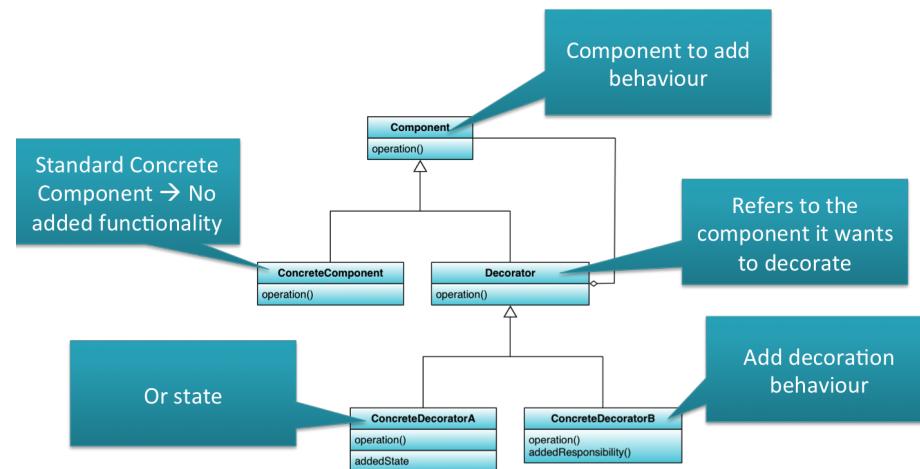
- reusar uma classe existente
- combinar classes não relacionadas que possuam interfaces incompatíveis



Decorator

- **Objetivo**

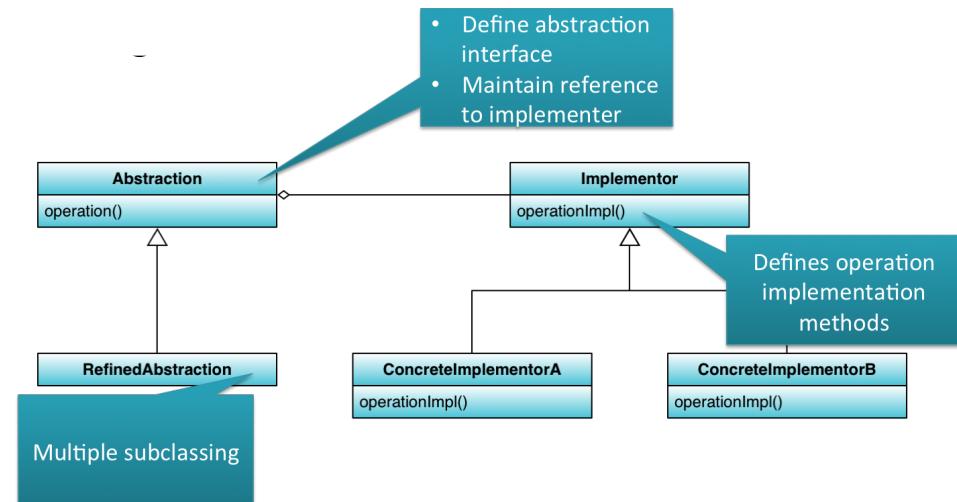
- adicionar funcionalidades a objetos sem afetar outros objetos
- poder remover funcionalidades no futuro
- extensão via subclasses é difícil



Bridge

- **Objetivo**

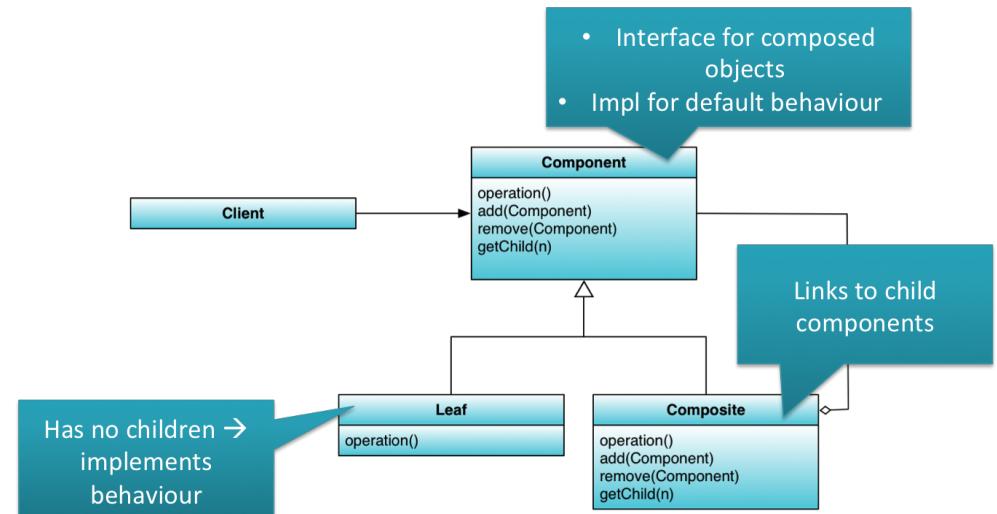
- evitar a vinculação entre interface e implementação
- Tornar possível o uso de subclasses para abstração e implementação
- Deve ser possível mudar a implementação em tempo de execução sem afetar os clientes



Composite

- **Objetivo**

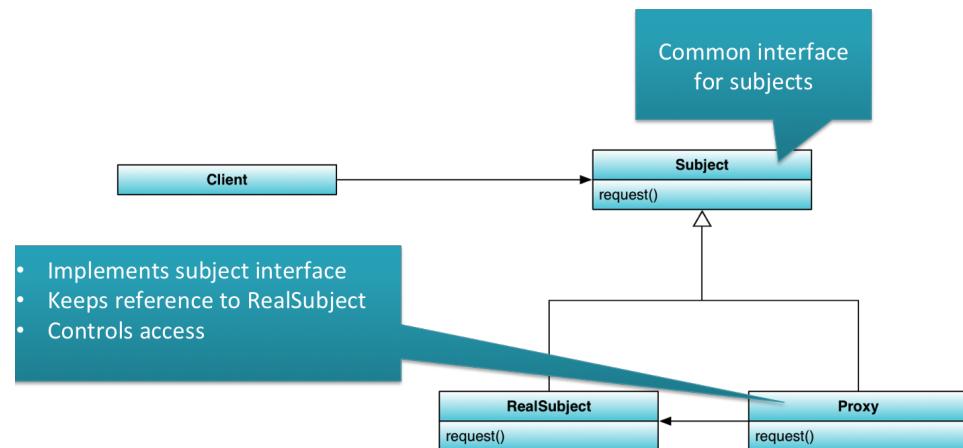
- ignorar diferenças entre composições e itens individuais
- representar hierarquias de objetos parte-todo



Proxy

- **Objetivo**

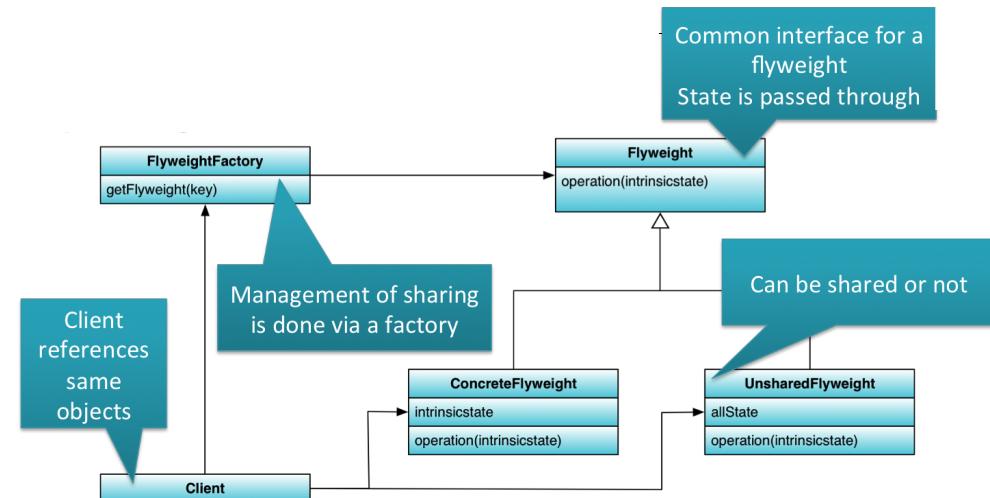
- Funcionalidade extra é requerida
 - Transparência
 - Mais do que apenas uma referência



Flyweight

- **Objetivo**

- Grande número de objetos
- Altos custos de armazenamento
- Estado compartilhado
- Substituir muitos objetos por poucos
- Identidade do objeto não é necessária



Facade

Facade: Motivação

- Sua empresa é uma empresa baseada em produtos e lançou um produto no mercado, chamado Schedule Server. É um tipo de servidor em si e é usado para gerenciar tarefas. Os trabalhos poderiam ser qualquer tipo de trabalho, como enviar uma lista de e-mails, sms, ler ou gravar arquivos de um destino, ou simplesmente transferir arquivos de uma fonte para o destino. O produto é usado pelos desenvolvedores para gerenciar esse tipo de trabalho e serem capazes de se concentrar mais em seu objetivo de negócios. O servidor executa cada trabalho no horário especificado e também gerencia todos os problemas de subjacentes, como o problema de concorrência e o de segurança. Como desenvolvedor, basta codificar apenas os requisitos de negócios relevantes e uma boa quantidade de chamadas de API é fornecida para agendar um trabalho de acordo com as necessidades deles.
- Tudo estava indo bem, até que os clientes começaram a reclamar sobre iniciar e parar o processo do servidor. Eles disseram que, embora o servidor esteja funcionando muito bem, os processos de inicialização e encerramento são muito complexos e eles querem uma maneira fácil de fazer isso. O servidor expôs uma interface complexa para os clientes que parece um pouco complicado para eles.
- Precisamos fornecer uma maneira fácil de iniciar e parar o servidor. Sem ter que refazer a codificação a partir do zero. Precisamos de uma maneira de resolver esse problema e tornar a interface fácil de acessar.

Facade: Motivação

Código para iniciar o servidor

```
ScheduleServer scheduleServer = new ScheduleServer();
scheduleServer.startBooting();
scheduleServer.readSystemConfigFile();
scheduleServer.init();
scheduleServer.initializeContext();
scheduleServer.initializeListeners();
scheduleServer.createSystemObjects();

System.out.println("Start working.....");
System.out.println("After work done.....");
```

Facade: Motivação

Código para parar o servidor

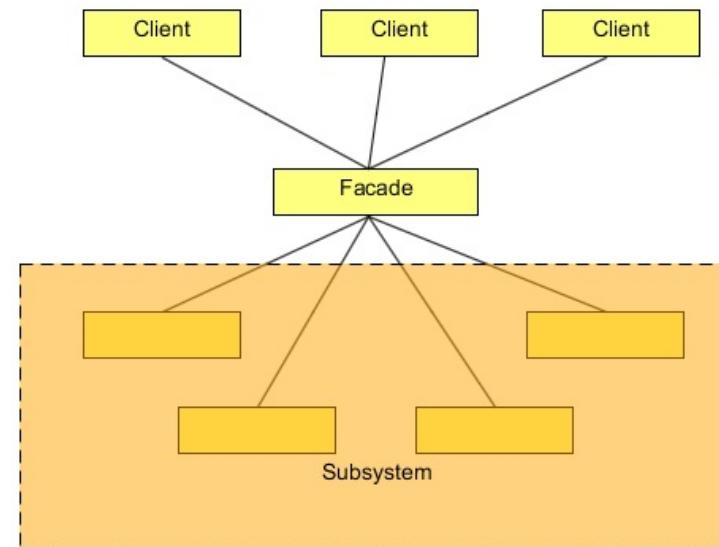
```
scheduleServer.releaseProcesses();
scheduleServer.destory();
scheduleServer.destroySystemObjects();
scheduleServer.destoryListeners();
scheduleServer.destoryContext();
scheduleServer.shutdown();
```

Facade: Intenção

- O Facade Pattern torna uma interface complexa mais fácil de usar, usando uma classe Facade. O Facade Pattern fornece uma interface unificada para um conjunto de interface em um subsistema. O Facade define uma interface de nível superior que facilita o uso do subsistema.
- O Facade unifica as complexas interfaces de baixo nível de um subsistema em ordem para fornecer uma maneira simples de acessar essa interface. Ele apenas fornece uma camada para as interfaces complexas do subsistema, o que facilita o uso.
- A Facade não encapsula as classes ou interfaces do subsistema; Ele apenas fornece uma interface simplificada para sua funcionalidade. Um cliente pode acessar essas classes diretamente. Ele ainda expõe a funcionalidade completa do sistema para os clientes que podem precisar dele.
- Uma Facade não é apenas capaz de simplificar uma interface, mas também separa um cliente de um subsistema. Ele adere ao Princípio do Mínimo Conhecimento, que evita o acoplamento entre o cliente e o subsistema. Isso proporciona flexibilidade: suponha que, no problema relatado, a empresa deseje adicionar mais algumas etapas para iniciar ou parar o Schedule Server, que possuem suas próprias interfaces diferentes. Se você codificou seu código de cliente para a fachada em vez do subsistema, seu código de cliente não precisa ser alterado, apenas a fachada necessária para ser alterada, que seria entregue com uma nova versão para o cliente.

Facade: Estrutura

- Os clientes se comunicam com o subsistema enviando solicitações ao Facade, que os encaminha ao(s) objeto(s) do subsistema apropriado(s).
- Embora os objetos do subsistema executem o trabalho real, a fachada pode ter que fazer o trabalho próprio para converter sua interface em interfaces do subsistema.
- Os clientes que usam a fachada não precisam acessar diretamente os objetos do subsistema.



Facade: Código Exemplo

```
package com.javacodegeeks.patterns.facadepattern;

public class ScheduleServerFacade {

    private final ScheduleServer scheduleServer;

    public ScheduleServerFacade(ScheduleServer scheduleServer) {
        this.scheduleServer = scheduleServer;
    }

    public void startServer() {

        scheduleServer.startBooting();
        scheduleServer.readSystemConfigFile();
        scheduleServer.init();
        scheduleServer.initializeContext();
        scheduleServer.initializeListeners();
        scheduleServer.createSystemObjects();
    }

    public void stopServer() {

        scheduleServer.releaseProcesses();
        scheduleServer.destory();
        scheduleServer.destroySystemObjects();
        scheduleServer.destroyListeners();
        scheduleServer.destroyContext();
        scheduleServer.shutdown();
    }
}
```

Fachada para iniciar e parar o servidor

Facade: Código Exemplo

Testando: iniciando e parando o servidor a partir da Facade

```
package com.javacodegeeks.patterns.facadepattern;

public class TestFacade {

    public static void main(String[] args) {

        ScheduleServer scheduleServer = new ScheduleServer();
        ScheduleServerFacade facadeServer = new ScheduleServerFacade(scheduleServer ←
            );
        facadeServer.startServer();

        System.out.println("Start working.....");
        System.out.println("After work done.....");

        facadeServer.stopServer();
    }
}
```

Adapter

Adapter: Motivação

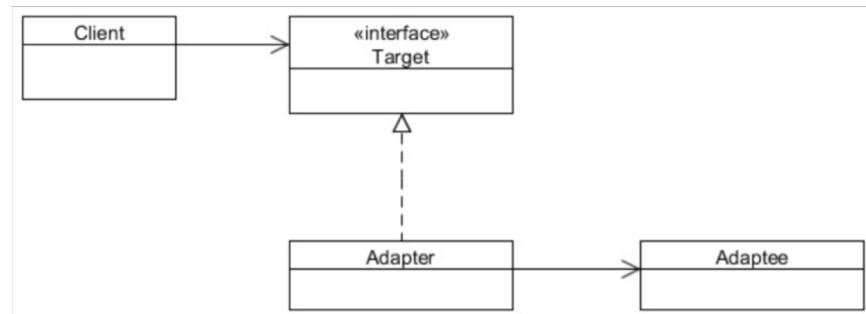
- Um desenvolvedor de software, Max, trabalhou em um site de comércio eletrônico. O site permite que os usuários façam compras e paguem on-line. O site é integrado a um portal de pagamento de terceiros, através do qual os usuários podem pagar suas contas usando seu cartão de crédito. Tudo estava indo bem, até que seu gerente o chamou para uma mudança no projeto.
- O gerente disse a ele que está planejando alterar o fornecedor do gateway de pagamento e precisa implementá-lo no código.
- O problema que surge aqui é que o site é anexado ao gateway de pagamento Xpay, que recebe um tipo de objeto Xpay. O novo fornecedor, PayD, permite apenas que o tipo de objeto PayD acesse o processo. Max não deseja alterar o conjunto de 100 classes que fazem referência a um objeto do tipo XPay. Isso também aumenta o risco do projeto, que já está sendo executado na produção. Nem ele pode alterar a ferramenta de terceiros do gateway de pagamento. O problema ocorreu devido às interfaces incompatíveis entre as duas partes diferentes do código. Para que o processo funcione, Max precisa encontrar uma maneira de tornar o código compatível com a API fornecida pelo fornecedor.

Adapter: Intenção

- O que o Max precisa aqui é um adaptador que pode se encaixar entre o código e a API do fornecedor e permitir que o processo flua. Mas antes da solução, vamos primeiro ver o que é um adaptador e como ele funciona.
- Às vezes, pode haver um cenário em que dois objetos não se encaixam, como devem ser feitos para que o trabalho seja feito. Essa situação pode surgir quando estamos tentando integrar um código legado a um novo código ou ao alterar uma API de terceiros no código. Isto é devido a interfaces incompatíveis dos dois objetos que não se encaixam.
- O padrão do adaptador permite adaptar o que um objeto ou uma classe expõe ao que outro objeto ou classe espera. Converte a interface de uma classe em outra interface que o cliente espera. Ele permite que as classes trabalhem juntas, o que não poderia ocorrer devido a interfaces incompatíveis. Permite fixar a interface entre os objetos e as classes sem modificar os objetos e as classes diretamente.
- Você pode pensar em um adaptador como um adaptador do mundo real que é usado para conectar duas peças diferentes de equipamento que não podem ser conectadas diretamente. Um adaptador fica entre esses equipamentos, obtém o fluxo do equipamento e o fornece para o outro equipamento na forma desejada, o que, de outra forma, é impossível devido às suas interfaces incompatíveis.

Adapter: Estrutura

- Um adaptador usa composição para armazenar o objeto que deve se adaptar e, quando os métodos do adaptador são chamados, ele converte essas chamadas em algo que o objeto adaptado pode entender e passa as chamadas para o objeto adaptado. O código que chama o adaptador nunca precisa saber que ele não está lidando com o tipo de objeto que ele considera, mas sim com um objeto adaptado.



Adapter: Código Exemplo

Adaptee: interface da classe concreta XPay

```
package com.javacodegeeks.patterns.adapterpattern.xpay;

public interface Xpay {

    public String getCreditCardNo();
    public String getCustomerName();
    public String getCardExpMonth();
    public String getCardExpYear();
    public Short getCardCVVNo();
    public Double getAmount();

    public void setCreditCardNo(String creditCardNo);
    public void setCustomerName(String customerName);
    public void setCardExpMonth(String cardExpMonth);
    public void setCardExpYear(String cardExpYear);
    public void setCardCVVNo(Short cardCVVNo);
    public void setAmount(Double amount);

}
```

Adapter: Código Exemplo

Adaptee: Código da Classe Concreta XPay

```
package com.javacodegeeks.patterns.adapterpattern.site;

import com.javacodegeeks.patterns.adapterpattern.xpay.Xpay;

public class XpayImpl implements Xpay{

    private String creditCardNo;
    private String customerName;
    private String cardExpMonth;
    private String cardExpYear;
    private Short cardCVVNo;
    private Double amount;

    @Override
    public String getCreditCardNo() {
        return creditCardNo;
    }
}
```

Adaptee: continuação do código da classe concreta XPay

```
@Override  
public String getCustomerName() {  
    return customerName;  
}  
  
@Override  
public String getCardExpMonth() {  
    return cardExpMonth;  
}  
  
@Override  
public String getCardExpYear() {  
    return cardExpYear;  
}
```

```
@Override  
public Short getCardCVVNo() {  
    return cardCVVNo;  
}  
  
@Override  
public Double getAmount() {  
    return amount;  
}  
  
@Override  
public void setCreditCardNo(String creditCardNo) {  
    this.creditCardNo = creditCardNo;  
}  
  
@Override  
public void setCustomerName(String customerName) {  
    this.customerName = customerName;  
}  
  
@Override  
public void setCardExpMonth(String cardExpMonth) {  
    this.cardExpMonth = cardExpMonth;  
}  
  
@Override  
public void setCardExpYear(String cardExpYear) {  
    this.cardExpYear = cardExpYear;  
}  
  
@Override  
public void setCardCVVNo(Short cardCVVNo) {  
    this.cardCVVNo = cardCVVNo;  
}  
  
@Override  
public void setAmount(Double amount) {  
    this.amount = amount;  
}  
}
```

Adapter: Código Exemplo

Target Interface: interface do novo gateway de pagamentos PayD

```
package com.javacodegeeks.patterns.adapterpattern.payd;

public interface PayD {

    public String getCustCardNo();
    public String getCardOwnerName();
    public String getCardExpMonthDate();
    public Integer getCVVNo();
    public Double getTotalAmount();

    public void setCustCardNo(String custCardNo);
    public void setCardOwnerName(String cardOwnerName);
    public void setCardExpMonthDate(String cardExpMonthDate);
    public void setCVVNo(Integer cvvNo);
    public void setTotalAmount(Double totalAmount);
}
```

Note que a diferença está na data de expiração, que usar mês e ano juntos. Na interface XPay, são métodos separados.

Adapter: Código Exemplo

Adapter: Código da Classe Concreta XPayToPayDAdapter, que usa o código existente do Adaptee XPay para implementar os métodos de PayD

```
package com.javacodegeeks.patterns.adapterpattern.site;

import com.javacodegeeks.patterns.adapterpattern.payd.PayD;
import com.javacodegeeks.patterns.adapterpattern.xpay.Xpay;

public class XpayToPayDAdapter implements PayD{

    private String custCardNo;
    private String cardOwnerName;
    private String cardExpMonthDate;
    private Integer cVVNo;
    private Double totalAmount;

    private final Xpay xpay;

    public XpayToPayDAdapter(Xpay xpay) {
        this.xpay = xpay;
        setProp();
    }
}
```

```

@Override
public String getCustCardNo() {
    return custCardNo;
}

@Override
public String getCardOwnerName() {
    return cardOwnerName;
}

@Override
public String getCardExpMonthDate() {
    return cardExpMonthDate;
}

@Override
public Integer getCVVNo() {
    return cVVNo;
}

@Override
public Double getTotalAmount() {
    return totalAmount;
}

@Override
public void setCustCardNo(String custCardNo) {
    this.custCardNo = custCardNo;
}

@Override
public void setCardOwnerName(String cardOwnerName) {
    this.cardOwnerName = cardOwnerName;
}

```

continuação do código da Classe Concreta XPayToPayDAdapter

```

@Override
public void setCardExpMonthDate(String cardExpMonthDate) {
    this.cardExpMonthDate = cardExpMonthDate;
}

@Override
public void setCVVNo(Integer cVVNo) {
    this.cVVNo = cVVNo;
}

@Override
public void setTotalAmount(Double totalAmount) {
    this.totalAmount = totalAmount;
}

private void setProp(){
    setCardOwnerName(this.xpay.getCustomerName());
    setCustCardNo(this.xpay.getCreditCardNo());
    setCardExpMonthDate(this.xpay.getCardExpMonth() + "/" + this.xpay. ←
        getCardExpYear());
    setCVVNo(this.xpay.getCardCVVNo().intValue());
    setTotalAmount(this.xpay.getAmount());
}

```

Adapter: Código Exemplo

Client: roda o exemplo de adapter

```
package com.javacodegeeks.patterns.adapterpattern.site;

import com.javacodegeeks.patterns.adapterpattern.payd.PayD;
import com.javacodegeeks.patterns.adapterpattern.xpay.Xpay;

public class RunAdapterExample {

    public static void main(String[] args) {

        // Object for Xpay
        Xpay xpay = new XpayImpl();
        xpay.setCreditCardNo("4789565874102365");
        xpay.setCustomerName("Max Warner");
        xpay.setCardExpMonth("09");
        xpay.setCardExpYear("25");
        xpay.setCardCVVNo((short)235);
        xpay.setAmount(2565.23);

        PayD payD = new XpayToPayDAdapter(xpay);
        testPayD(payD);
    }
}
```

continuação do código cliente

```
}

private static void testPayD(PayD payD) {

    System.out.println(payD.getCardOwnerName());
    System.out.println(payD.getCustCardNo());
    System.out.println(payD.getCardExpMonthDate());
    System.out.println(payD.getCVVNo());
    System.out.println(payD.getTotalAmount());
}

}
```

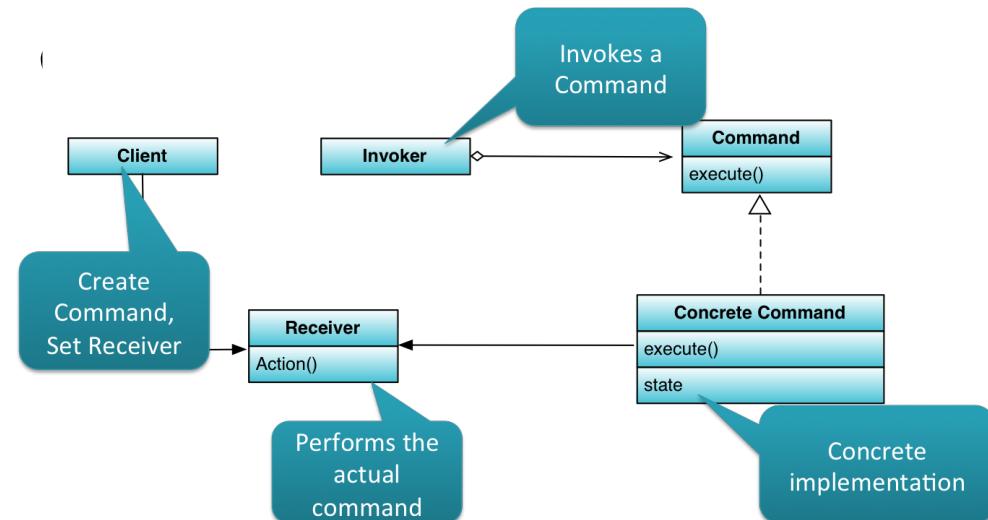
Padrões Comportamentais

Objetivos e Estrutura

Command

- **Objetivo**

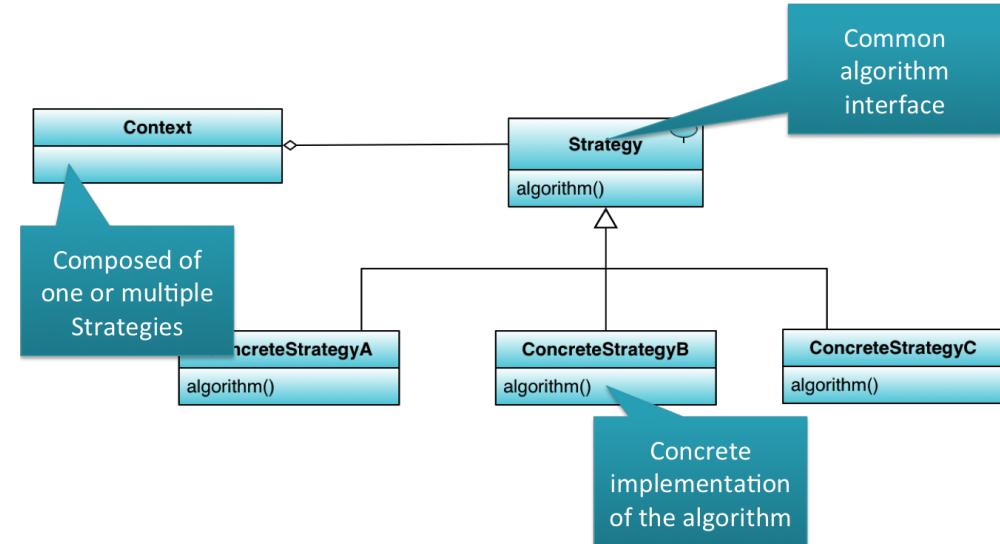
- Command como parâmetro
- Passar command como objeto geral
- Fila de Requisições
- Salvar estado da requisição
- Funcionalidade de undo
- Fornecer métodos de fazer e desfazer
- Suportar logs
- Reexecutar em caso de falha



Strategy

- **Objetivo**

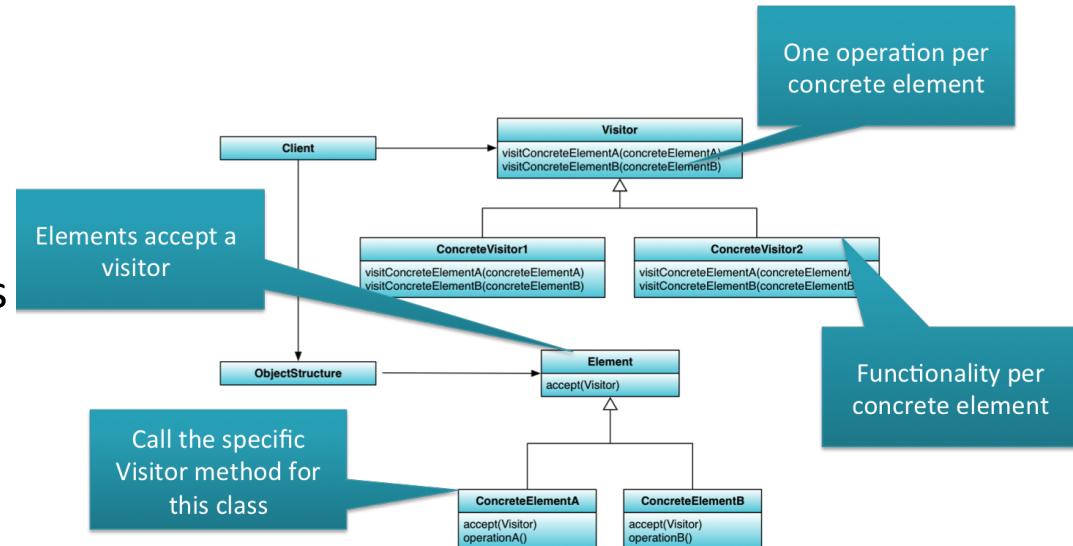
- Classes mudam apenas em comportamento
- Diferentes variações de um algoritmo
- Algoritmos que usam dados complexos dos quais os clientes não devem estar cientes



Visitor

- **Objetivo**

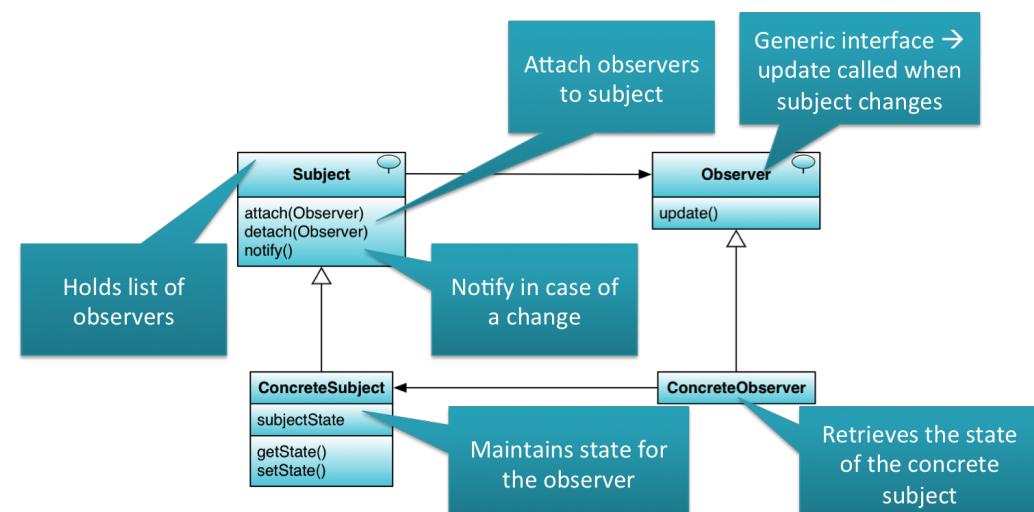
- Visitar uma estrutura de objetos complexa (herança)
 - executar operações baseado nas classes concretas
- Evitar a poluição de classes concretas com muitas operações diferentes
 - Visitor agrupa funcionalidades
- Habilidade de facilmente definir novas operações sem mudar as classes concretas.



Observer

- **Objetivo**

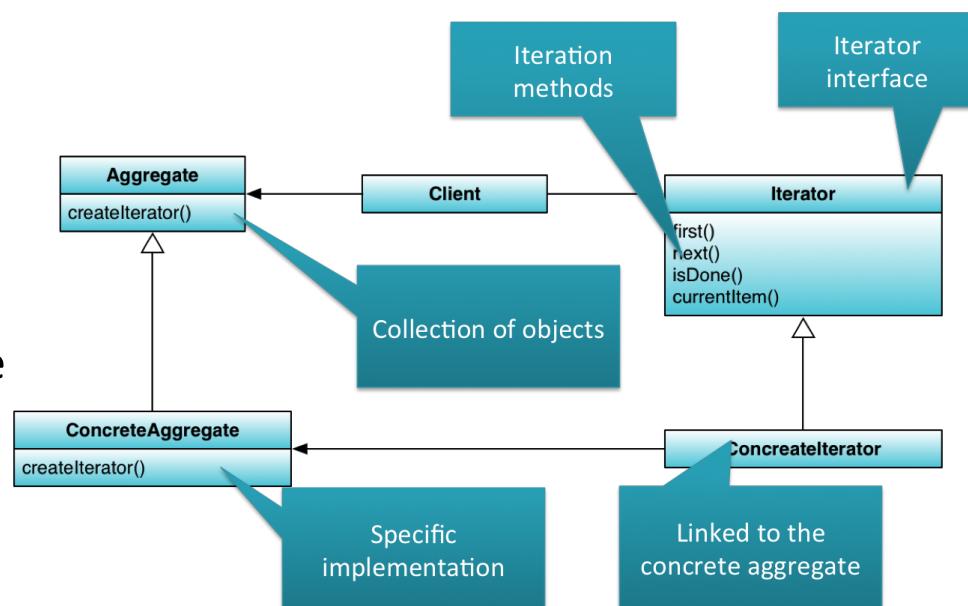
- Mudar um objeto implica em mudar outros
- Nenhuma ideia de quantos objetos precisam mudar
- Notificação de mudança de objetos
- Um objeto notifica outros sem saber quem eles são



Iterator

- **Objetivo**

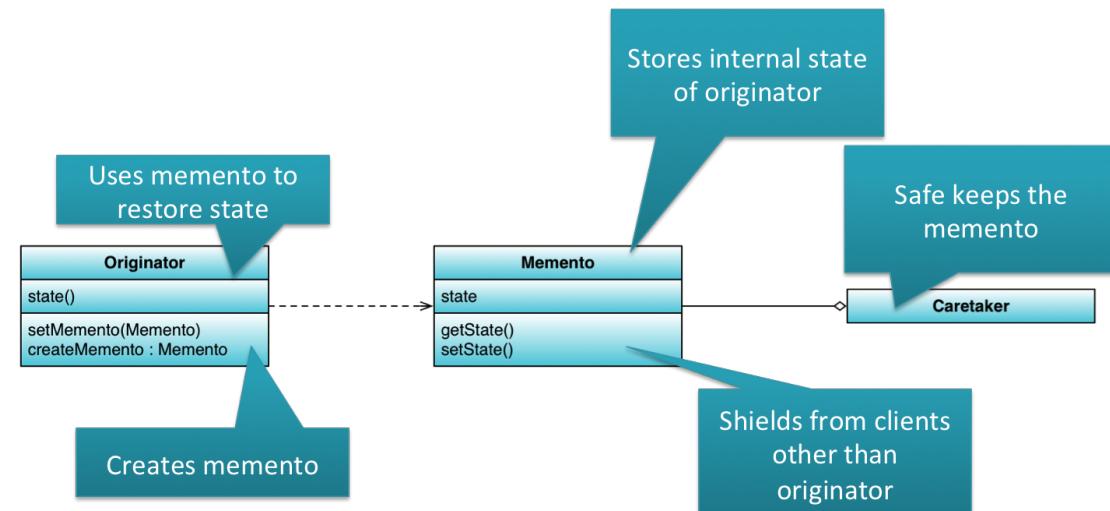
- Acessar o conteúdo de objetos segregados sem expor sua representação
- Suportar múltiplos percursos de objetos agregados
- Fornecer uma interface uniforme para:
 - Percorrer objetos agregados
 - Podem ser de classes diferentes



Memento

- **Objetivo**

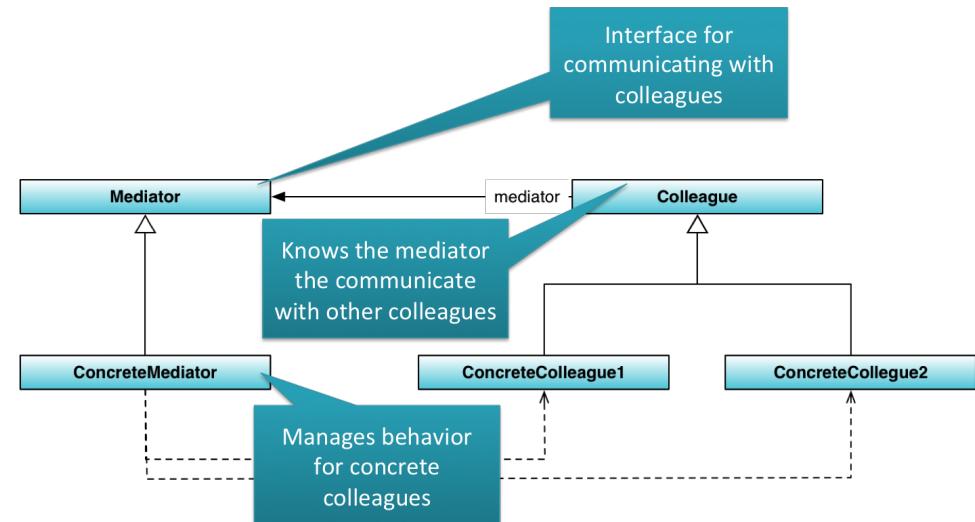
- Salvar uma foto do estado de um objeto sem violar o encapsulamento



Mediator

- **Objetivo**

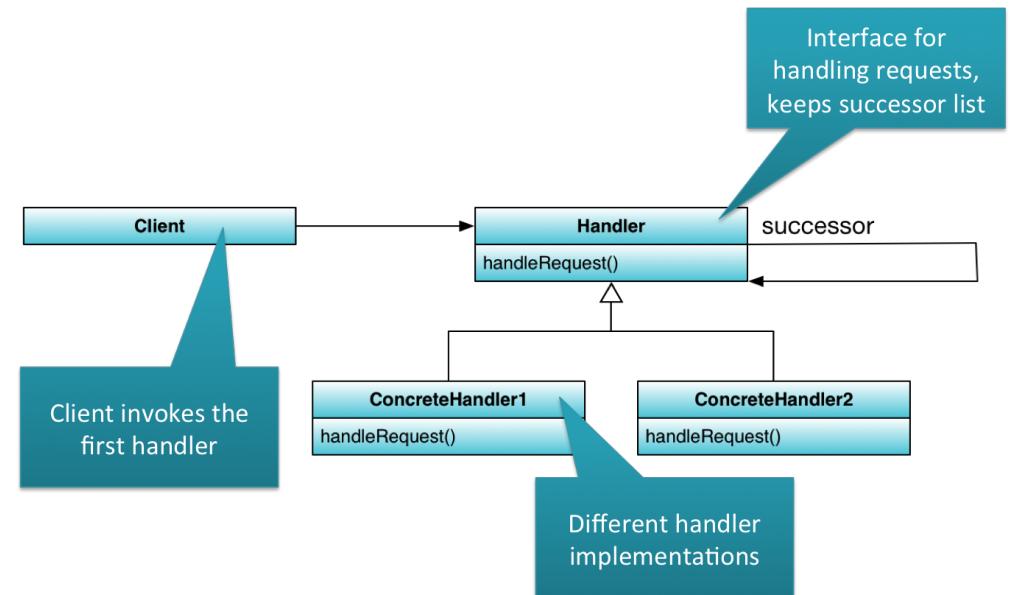
- Objetos tem comunicação complexa mas são bem definidos
- Difícil de identificar como a comunicação funciona na verdade
- Reuso de objetos é difícil
- Centralizar comportamento entre classes



Chain of responsibility

- **Objetivo**

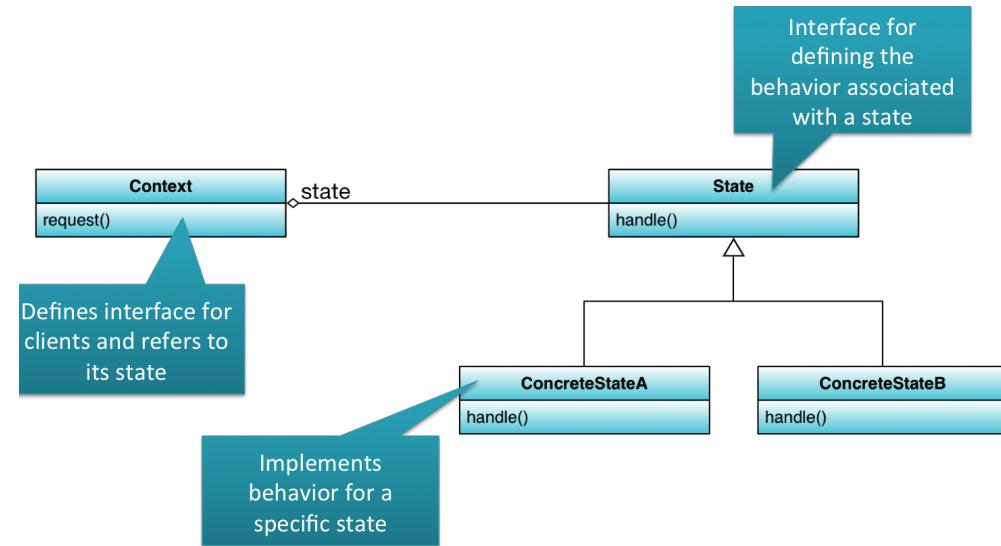
- Múltiplos objetos precisam lidar com uma requisição
- Não é claro, a princípio, quem tem que lidar com ela
- Quem pode lidar com a requisição deveria ser dinâmico



State

- **Objetivo**

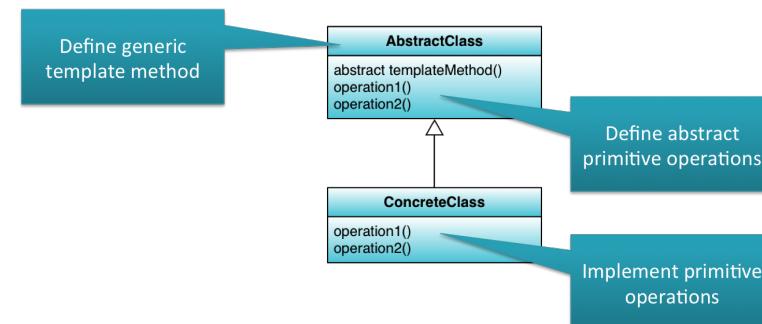
- O comportamento do objeto depende do seu estado
- Evitar estruturas complexas de if-else



Template method

- **Objetivo**

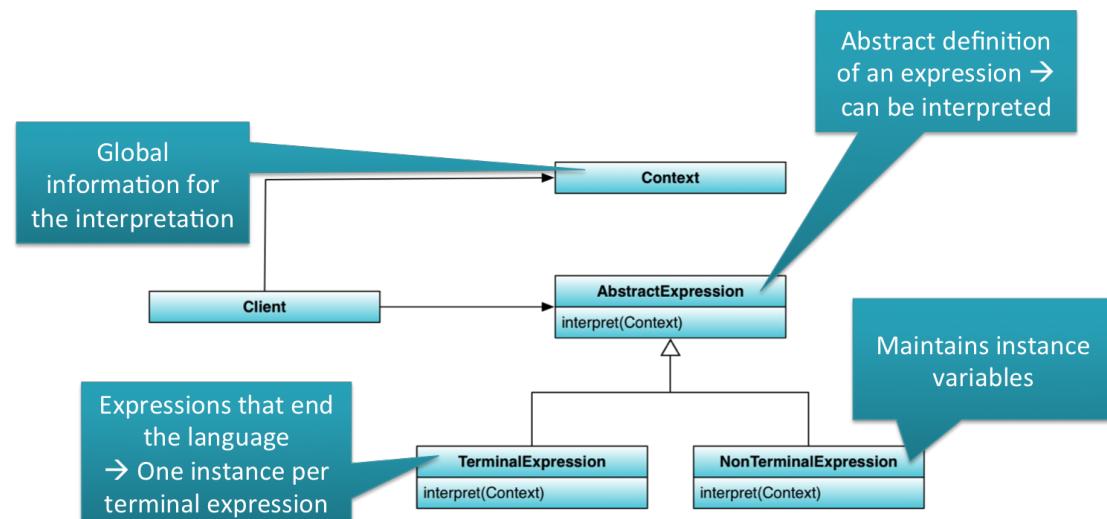
- Implementar um algoritmo uma vez
- Evitar duplicação de código
- Definir como uma classe deve ser estendida



Interpreter

- **Objetivo**

- A linguagem precisa ser interpretada
- Eficiência não é crítica



Strategy

Strategy: Motivação

- O Strategy Design Pattern parece ser o mais simples de todos os padrões de projeto, mas oferece grande flexibilidade ao seu código. Esse padrão é usado em quase todos os lugares, mesmo em conjunto com outros padrões de projeto.
- Para entender o padrão de design strategy, vamos criar um formatador de texto para um editor de texto. Um editor de texto pode ter diferentes formatadores de texto para formatar o texto. Podemos criar diferentes formatadores de texto e depois passar o necessário para o editor de texto, para que o editor possa formatar o texto conforme o exigido.
- O editor de texto manterá uma referência a uma interface comum para o formatador de texto e o trabalho do editor será passar o texto ao formatador para formatar o texto.

Strategy: Intenção

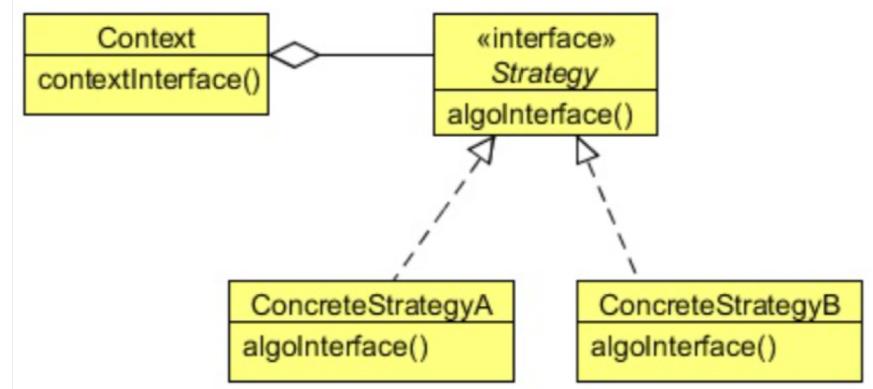
- O Strategy Design Pattern define uma família de algoritmos, encapsulando cada um deles, e tornando-os intercambiáveis. A estratégia permite que o algoritmo varie independentemente dos clientes que o usam.
- O padrão Strategy é útil quando há um conjunto de algoritmos relacionados e um objeto cliente precisa ser capaz de escolher e escolher dinamicamente um algoritmo desse conjunto que atenda à sua necessidade atual. O padrão Strategy sugere manter a implementação de cada um dos algoritmos em uma classe separada. Cada um desses algoritmos encapsulados em uma classe separada é chamado de estratégia. Um objeto que usa um objeto Strategy é geralmente chamado de objeto de contexto.

Strategy: Intenção

- Com diferentes objetos de Estratégia, alterar o comportamento de um objeto Contexto é simplesmente uma questão de mudar seu objeto Estratégia para aquele que implementa o algoritmo requerido. Para permitir que um objeto Contexto acesse diferentes objetos de Estratégia de maneira transparente, todos os objetos de Estratégia devem ser projetados para oferecer a mesma interface. Na linguagem de programação Java, isso pode ser feito projetando cada objeto de Estratégia como um implementador de uma interface comum ou como uma subclasse de uma classe abstrata comum que declara a interface comum requerida.
- Depois que o grupo de algoritmos relacionados é encapsulado em um conjunto de classes de estratégia em uma hierarquia de classes, um cliente pode escolher entre esses algoritmos selecionando e instanciando uma classe de estratégia apropriada. Para alterar o comportamento do contexto, um objeto cliente precisa configurar o contexto com a instância de estratégia selecionada. Esse tipo de arranjo separa completamente a implementação de um algoritmo do contexto que o utiliza. Como resultado, quando uma implementação de algoritmo existente é alterada ou um novo algoritmo é adicionado ao grupo, tanto o contexto quanto o objeto cliente (que usa o contexto) permanecem inalterados.

Strategy: Estrutura

- **Strategy**
 - Declara uma interface comum a todos os algoritmos suportados. O Context usa essa interface para chamar o algoritmo definido por um ConcreteStrategy.
- **ConcreteStrategy**
 - Implementa o algoritmo usando a interface Strategy.
- **Context**
 - Está configurado com um objeto ConcreteStrategy.
 - Mantém uma referência a um objeto Strategy.
 - Pode definir uma interface que permita à Strategy acessar seus dados.



Strategy: Código Exemplo

Strategy: a interface de um formatador de textos

```
package com.javacodegeeks.patterns.strategypattern;

public interface TextFormatter {
    public void format(String text);
}
```

Strategy: Código Exemplo

ConcreteStrategy: 2 diferentes formatações de texto

Todas Maiúsculas

```
package com.javacodegeeks.patterns.strategypattern;

public class CapTextFormatter implements TextFormatter{

    @Override
    public void format(String text) {
        System.out.println("[CapTextFormatter]: "+text.toUpperCase());
    }
}
```

Strategy: Código Exemplo

ConcreteStrategy: 2 diferentes formatações de texto

Todas Minúsculas

```
package com.javacodegeeks.patterns.strategypattern;

public class LowerTextFormatter implements TextFormatter{

    @Override
    public void format(String text) {
        System.out.println("[LowerTextFormatter]: "+text.toLowerCase());
    }
}
```

Strategy: Código Exemplo

Context: guarda a referência para a Estratégia – o formatador

```
package com.javacodegeeks.patterns.strategypattern;

public class TextEditor {

    private final TextFormatter textFormatter;

    public TextEditor(TextFormatter textFormatter){
        this.textFormatter = textFormatter;
    }

    public void publishText(String text){
        textFormatter.format(text);
    }

}
```

Strategy: Código Exemplo

Testando

```
package com.javacodegeeks.patterns.strategypattern;

public class TestStrategyPattern {

    public static void main(String[] args) {
        TextFormatter formatter = new CapTextFormatter();
        TextEditor editor = new TextEditor(formatter);
        editor.publishText("Testing text in caps formatter");

        formatter = new LowerTextFormatter();
        editor = new TextEditor(formatter);
        editor.publishText("Testing text in lower formatter");
    }
}
```

Strategy: Código Exemplo

Resultado

```
[CapTextFormatter]: TESTING TEXT IN CAPS FORMATTER  
[LowerTextFormatter]: testing text in lower formatter
```

Template

Template: Motivação

- Você já se conectou a um banco de dados de relacionamentos usando seu aplicativo Java? Vamos lembrar de algumas etapas importantes necessárias para conectar e inserir dados no banco de dados. Primeiro, precisamos de um driver de acordo com o banco de dados com o qual queremos nos conectar. Em seguida, passamos algumas credenciais para o banco de dados e, em seguida, preparamos uma instrução, definimos os dados na instrução de inserção e os inserimos usando o comando insert. Mais tarde, fechamos todas as conexões e, opcionalmente, destruímos todos os objetos de conexão.
- Você precisa escrever todas essas etapas, independentemente do banco de dados relacional de qualquer fornecedor. Considere um problema em que você precisa inserir alguns dados nos diferentes bancos de dados. Você precisa buscar alguns dados de um arquivo CSV e inseri-los em um banco de dados MySQL. Alguns dados vêm de um arquivo de texto e devem ser inseridos em um banco de dados Oracle. A única diferença é o driver e os dados, o restante das etapas deve ser o mesmo, já que o JDBC fornece um conjunto comum de interfaces para se comunicar com o banco de dados de relação específico de qualquer fornecedor.
- Podemos criar um modelo, que irá realizar algumas etapas para o cliente, e deixaremos algumas etapas para permitir que o cliente as implemente de maneira específica. Opcionalmente, um cliente pode substituir o comportamento padrão de algumas etapas já definidas.

Template: Intenção

- O Template Design Pattern é um padrão de comportamento e, como o nome sugere, ele fornece um modelo ou uma estrutura de um algoritmo que é usado pelos usuários. Um usuário fornece sua própria implementação sem alterar a estrutura do algoritmo.
- O Template Pattern define o esqueleto de um algoritmo em uma operação, adiando algumas etapas para subclasses. O Template Method permite que as subclasses redefinam certas etapas de um algoritmo sem alterar a estrutura do algoritmo.

Template: Intenção

- O padrão Template Method pode ser usado em situações em que há um algoritmo, e algumas etapas podem ser implementadas de várias maneiras diferentes. Em tais cenários, o padrão Template Method sugere manter o contorno do algoritmo em um método separado chamado de template em uma classe, que pode ser referida como uma classe template, deixando de fora as implementações específicas das partes variantes (passos que podem ser implementados de várias maneiras diferentes) do algoritmo para diferentes subclasses dessa classe.
- A classe Template não precisa necessariamente deixar a implementação para subclasses em sua totalidade. Em vez disso, como parte do fornecimento do esboço do algoritmo, a classe Template também pode fornecer uma certa quantidade de implementação que pode ser considerada invariante em diferentes implementações. Ele pode até fornecer implementação padrão para as partes variantes, se apropriado. Apenas detalhes específicos serão implementados dentro de diferentes subclasses. Esse tipo de implementação elimina a necessidade de código duplicado, o que significa uma quantidade mínima de código a ser gravada.

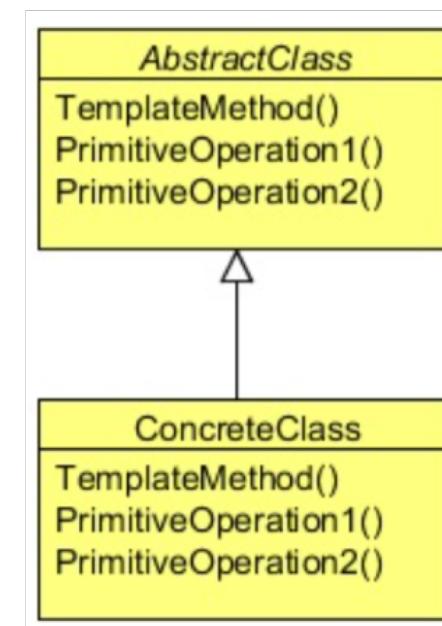
Template: Estrutura

- **AbstractClass**

- Define operações primitivas abstratas que subclasses concretas definem para implementar etapas de um algoritmo.
- Implementa um método de modelo que define o esqueleto de um algoritmo. O método de modelo também chama operações primitivas como operações definidas no AbstractClass ou de outros objetos.

- **ConcreteClass**

- Implementa as operações primitivas para transportar.



Template: Código Exemplo

AbstractClass: operações genéricas de JDBC

```
package com.javacodegeeks.patterns.templatepattern;

public abstract class ConnectionTemplate {

    public final void run() {
        setDBDriver();
        setCredentials();
        connect();
        prepareStatement();
        setData();
        insert();
        close();
        destroy();
    }
}
```

continuação da AbstractClass: operações genéricas de JDBC

```
public abstract void setDBDriver();

public abstract void setCredentials();

public void connect() {
    System.out.println("Setting connection...");
}

public void prepareStatement() {
    System.out.println("Preparing insert statement...");
}

public abstract void setData();

public void insert() {
    System.out.println("Inserting data...");
}

public void close() {
    System.out.println("Closing connections...");
}

public void destroy() {
    System.out.println("Destroying connection objects...");
}
}
```

Template: Código Exemplo

ConcreteClass: implementação para MySQL

MySQL

```
package com.javacodegeeks.patterns.templatepattern;

public class MySqLCSVCon extends ConnectionTemplate {

    @Override
    public void setDBDriver() {
        System.out.println("Setting MySQL DB drivers...");
    }

    @Override
    public void setCredentials() {
        System.out.println("Setting credentials for MySQL DB...");
    }

    @Override
    public void setData() {
        System.out.println("Setting up data from csv file....");
    }
}
```

Template: Código Exemplo

ConcreteClass: implementação para Oracle

Oracle

```
package com.javacodegeeks.patterns.templatepattern;

public class OracleTxtCon extends ConnectionTemplate {

    @Override
    public void setDBDriver() {
        System.out.println("Setting Oracle DB drivers...");
    }

    @Override
    public void setCredentials() {
        System.out.println("Setting credentials for Oracle DB...");
    }

    @Override
    public void setData() {
        System.out.println("Setting up data from txt file....");
    }
}
```

Template: Código Exemplo

Testando

```
package com.javacodegeeks.patterns.templatepattern;

public class TestTemplatePattern {

    public static void main(String[] args) {
        System.out.println("For MYSQL....");
        ConnectionTemplate template = new MySqLCSVCon();
        template.run();
        System.out.println("For Oracle...");
        template = new OracleTxtCon();
        template.run();
    }
}
```

Template: Código Exemplo

Resultado

```
For MYSQL....  
Setting MySQL DB drivers...  
Setting credentials for MySQL DB...  
Setting connection...  
Preparing insert statement...  
Setting up data from csv file....  
Inserting data...  
Closing connections...  
Destroying connection objects...  
  
For Oracle...  
Setting Oracle DB drivers...  
Setting credentials for Oracle DB...  
Setting connection...  
Preparing insert statement...  
Setting up data from txt file....  
Inserting data...  
Closing connections...  
Destroying connection objects...
```