

Padrões de projeto de software orientado a objetos

Proxy, Decorator e Adapter

Profa. Thienne Johnson



Conteúdo

- ▶ E. Gamma and R. Helm and R. Johnson and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
 - Conhecido como GoF (Gang of Four)
 - Versão em português disponível na biblioteca da EACH. (nome: Padrões de Projeto)

- ▶ Java Design Patterns At a Glance
 - <http://www.javacamp.org/designPattern>
- ▶ Java Design Patterns Resource Center
 - <http://www.deitel.com/ResourceCenters/Programming/JavaDesignPatterns/tabid/1103/Default.aspx>

Design Patterns

Elements of Reusable
Object-Oriented Software

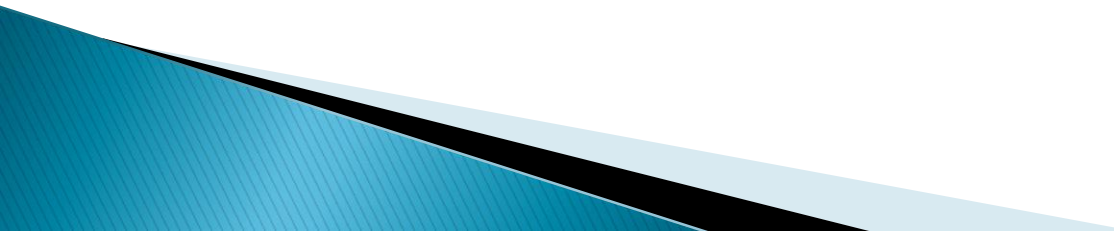
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides



Foreword by Grady Booch



Padrões estruturais

- ▶ Descrevem como objetos e classes podem ser combinados para formar estruturas.
 - ▶ Padrões de Objetos e Classes
 - Classe: descrevem relações e estruturas com auxílio de herança
 - Objetos: descrevem como objetos podem ser associados e agregados para formar estruturas maiores e mais complexas.
- 

Proxy

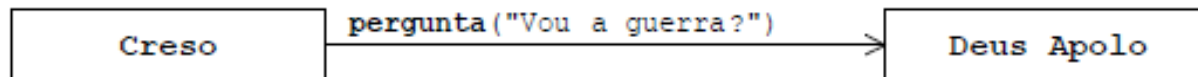
Padrão estrutural

Definição

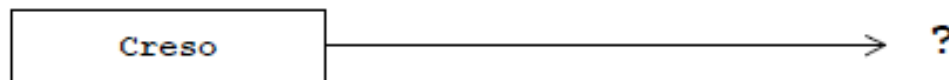
- ▶ Proxy é um padrão estrutural que fornece um representante para outro objeto de forma a controlar o acesso a ele.

Motivação

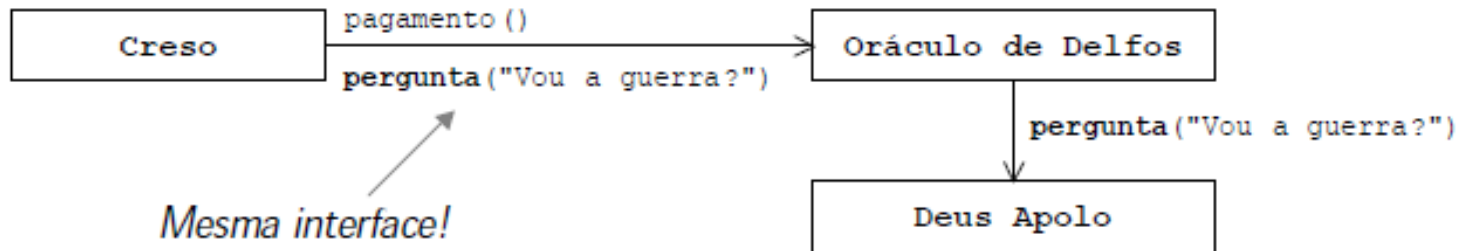
- *Sistema quer utilizar objeto real...*



- *Mas ele não está disponível (remoto, inacessível, ...)*



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

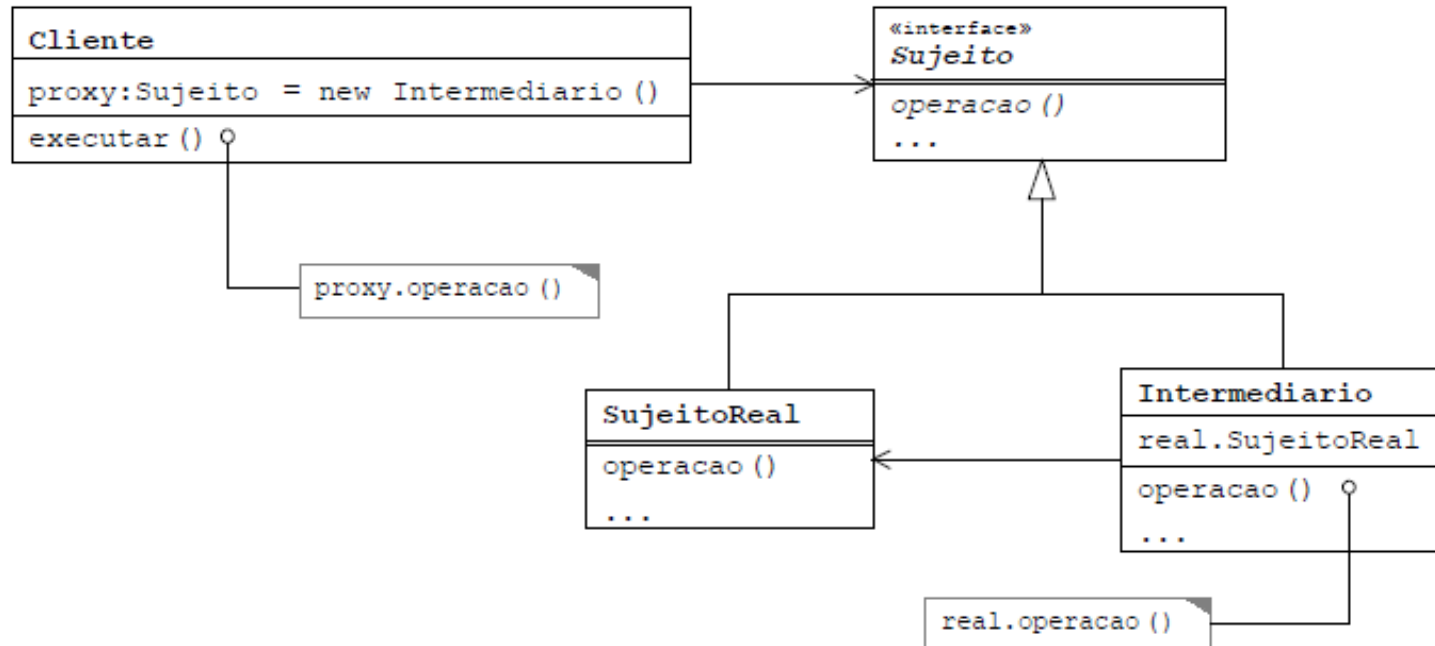


Alguns tipos de Proxy

Table 23.1 List of Different Proxy Types

<i>Proxy Type</i>	<i>Purpose</i>
Remote Proxy	To provide access to an object located in a different address space.
Virtual Proxy	To provide the required functionality to allow the on-demand creation of a memory intensive object (until required).
Cache Proxy/Server Proxy	To provide the functionality required to store the results of most frequently used target operations. The proxy object stores these results in some kind of a repository. When a client object requests the same operation, the proxy returns the operation results from the storage area without actually accessing the target object.
Firewall Proxy	The primary use of a firewall proxy is to protect target objects from bad clients. A firewall proxy can also be used to provide the functionality required to prevent clients from accessing harmful targets.
Protection Proxy	To provide the functionality required for allowing different clients to access the target object at different levels. A set of permissions is defined at the time of creation of the proxy. Subsequently, those permissions are used to restrict access to specific parts of the proxy (in turn of the target object). A client object is not allowed to access a particular method if it does not have a specific right to execute the method.

Estrutura



Participantes

▶ Cliente

- Usa intermediário em vez de sujeito real

▶ Intermediário

- Suporta a mesma interface que sujeito real
- Contém uma referência para o sujeito real e repassa chamadas, possivelmente, acrescentando informações ou filtrando dados no processo

Proxy em Java

```
public class Creso {  
    ...  
    Sujeito apolo = Fabrica.getSujeito();  
    apolo.operacao();  
    ...  
}
```

```
public class SujeitoReal implements Sujeito {  
    public Object operacao() {  
        return coisaUtil;  
    }  
}
```

```
public class Intermediario implements Sujeito {  
    private SujeitoReal real;  
    public Object operacao() {  
        cobraTaxa();  
        return real.operacao();  
    }  
}
```

```
public interface Sujeito {  
    public Object operacao();  
}
```

*inacessível
pelo cliente*



*cliente comunica-se
com este objeto*



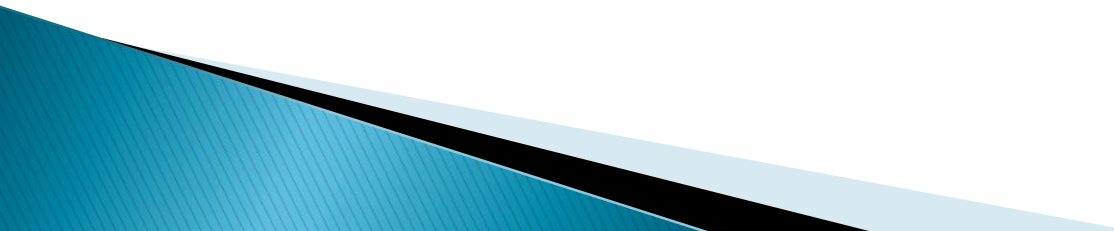
Vantagens

- ▶ **Transparência:** mesma sintaxe usada na comunicação entre o cliente e sujeito real é usada no proxy
- ▶ **Segurança**

▶ Desvantagens

- Possível impacto na performance
- Transparência nem sempre é 100% (fatores externos como queda da rede podem tornar o proxy inoperante ou desatualizado)

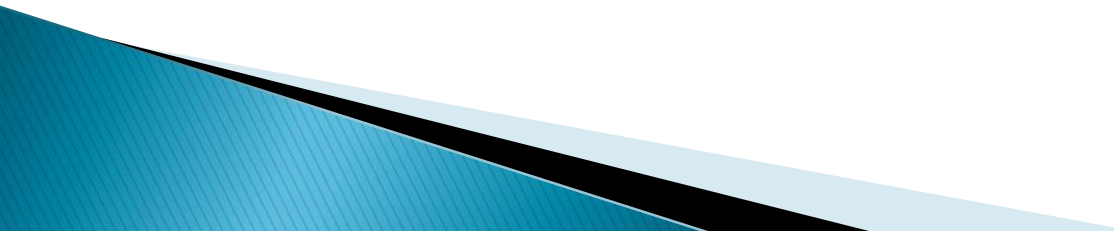
Aplicabilidade

- ▶ Quando a criação de um objeto é relativamente cara, pode ser uma boa idéia substituí-lo com um Proxy que possa fazer a instanciação cara do objeto ser mantida em um mínimo.
 - ▶ Permite login e autorização antes que alguém alcance o objeto requisitado
 - ▶ Pode fornecer uma representação local de um objeto em uma localização remota.
- 

Benefícios

- ▶ Dá habilidade para controlar o acesso a um objeto, se for por causa de um processo de criação caro ou aspectos de segurança.

Consequências

- ▶ Introduce outro nível de abstração para um objeto
 - ▶ Se alguns objetos acessam o objeto alvo diretamente e outros via o proxy, existe uma chance que eles consigam comportamentos diferentes. Isto pode ou não ser a intenção do criador.
- 

Proxy versus Façade

▶ Proxy

- Representa um único objeto.
- O objeto do cliente não acessa o objeto alvo diretamente.
- O objeto Proxy fornece controle de acesso ao objeto alvo único.

▶ Façade

- Representa um subsistema de objetos.
- O objeto do cliente tem habilidade de acessar o subsistema diretamente, se necessário.
- Um objeto Façade fornece uma interface de alto nível simplificada para os componentes do subsistema.

Exemplo

```
package cachedLogging;  
    public interface ICachedLogging {  
        public void logRequest(String logString);  
    }
```

```
package cachedLogging;  
public class CachedLogger implements ICachedLogging {  
    public void logRequest(String logString) {  
        System.out.println("CachedLogger logging to some  
expensive resource: " + logString + "\n");  
    }  
}
```


Exemplo (cont.)

```
package cachedLogging;
import java.util.ArrayList;
import java.util.List;
public class CachedLoggingProxy implements ICachedLogging {
    List<String> cachedLogEntries = new ArrayList<String>();
    CachedLogger cachedLogger = new CachedLogger();
    public void logRequest(String logString) {
        addLogRequest(logString);
    }

    private void addLogRequest(String logString) {
        cachedLogEntries.add(logString);
        if(cachedLogEntries.size() >= 4)
            performLogging();
    }
}
```

Exemplo (cont.)

```
private void performLogging() {
    StringBuffer accumulatedLogString = new StringBuffer();
    for (String logString : cachedLogEntries) {
        accumulatedLogString.append("\n"+logString);
        System.out.println("CachedLoggingProxy: adding logString \"\"
+logString + "\" to cached log entries.");
    }
    System.out.println("CachedLoggingProxy: sends accumulated logstring
to CachedLogger.");
    cachedLogger.logRequest(accumulatedLogString.toString());
    cachedLogEntries.clear();
    System.out.println("CachedLoggingProxy: cachedLogEntries cleared.");
}
}
```

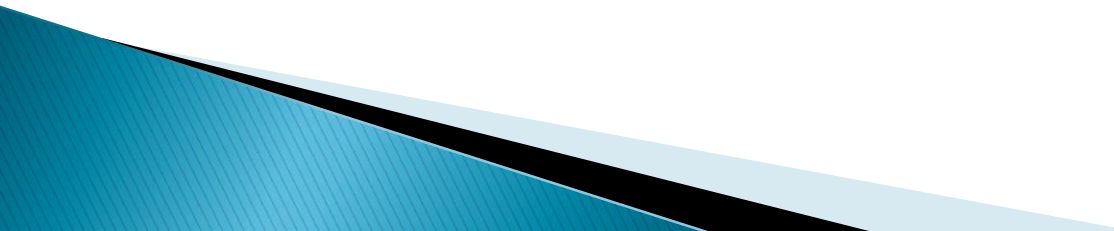
Exemplo (cont.)

```
package client;
import cachedLogging.*;
public class Client {
    public static void main(String[] args) {
        ICachedLogging logger = new CachedLoggingProxy();
        for (int i = 1; i < 5; i++) {
            logger.logRequest("logString "+i);
        }
    }
}
```

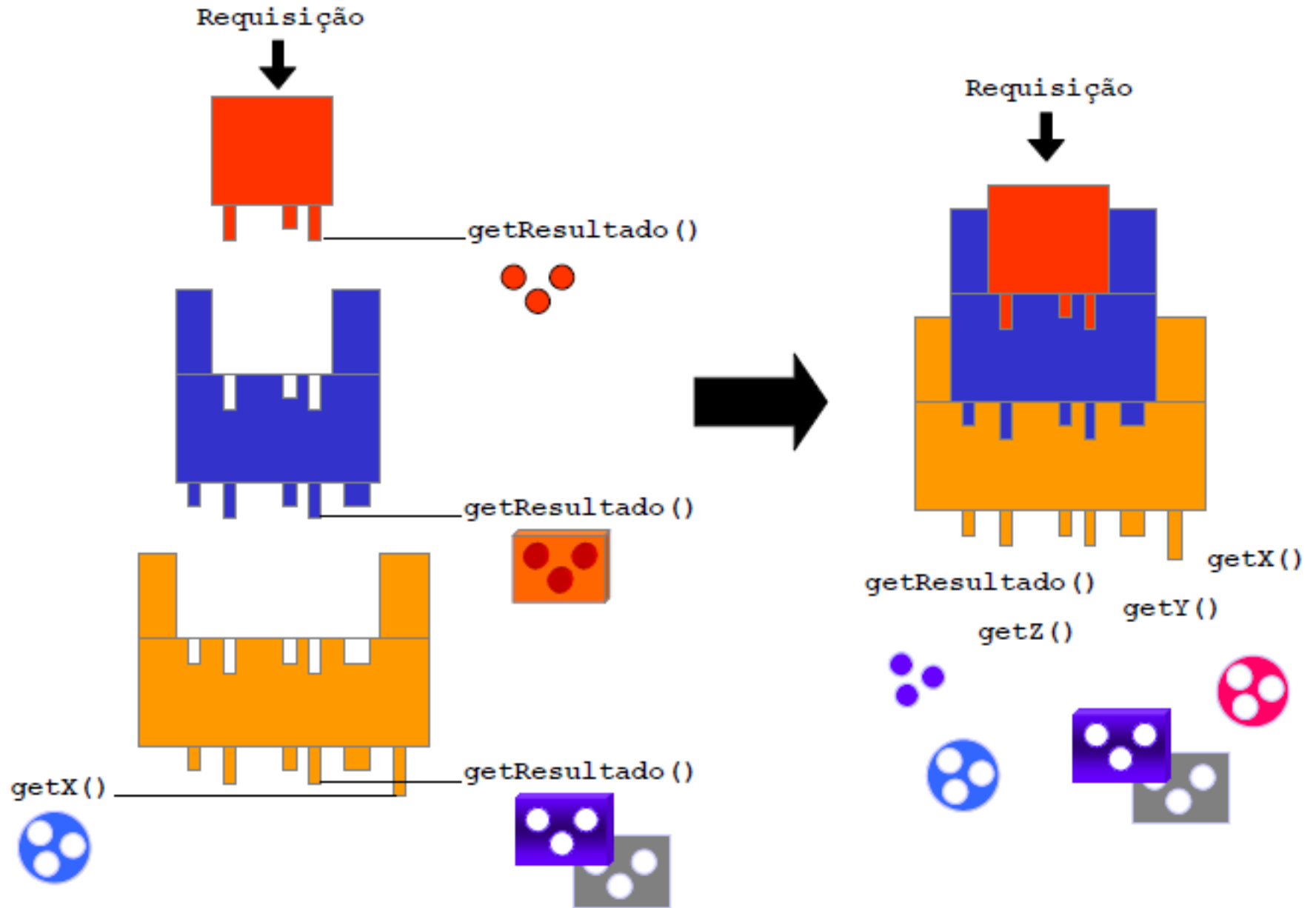
Decorator

Padrão estrutural

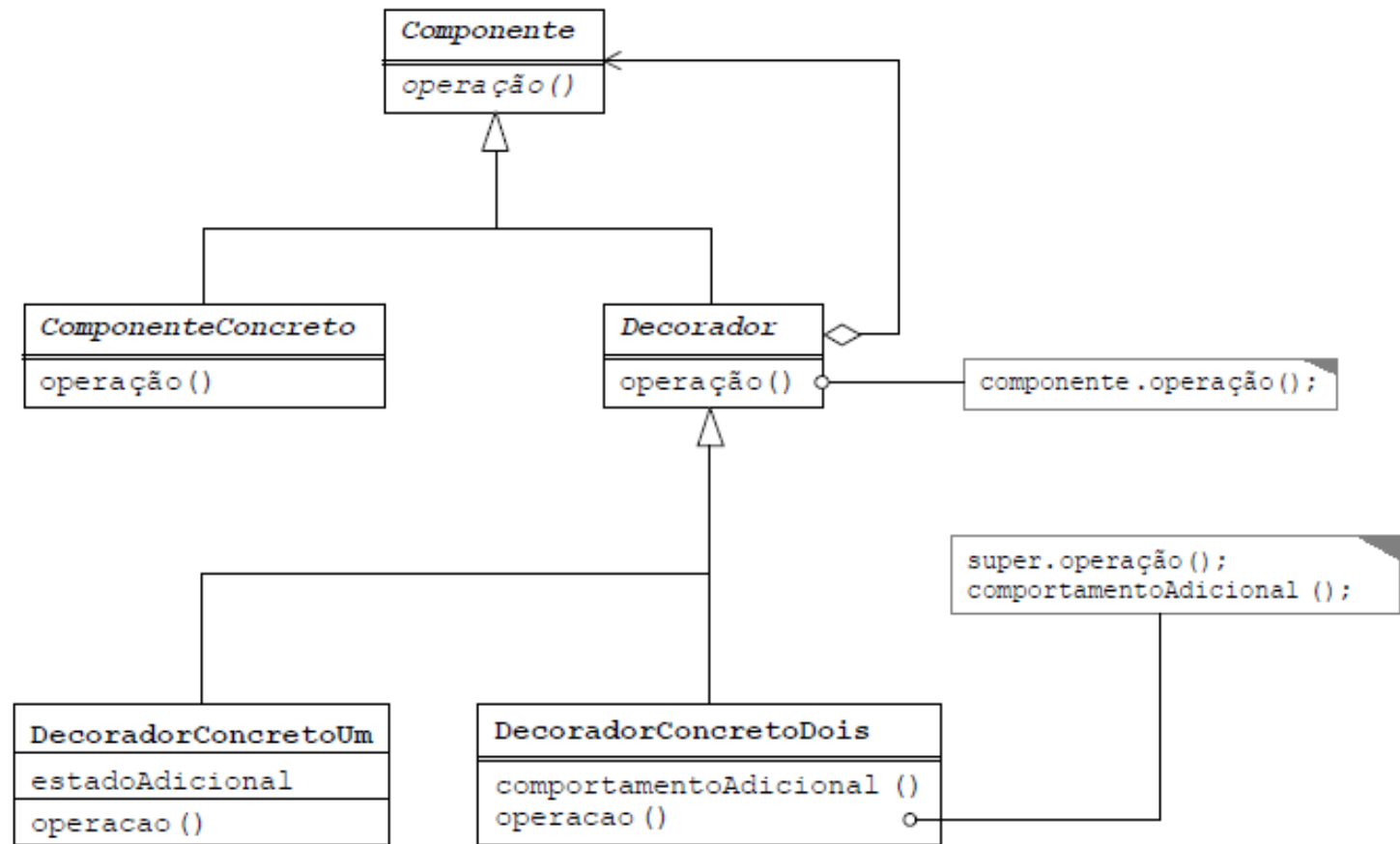
Definição

- ▶ Permite adicionar responsabilidade e modificar a funcionalidade de uma instância dinamicamente.
 - ▶ Fornecem uma alternativa flexível a subclassing ao estender funcionalidades usando composição ao invés de herança.
- 

Motivação



Estrutura



Decorator em Java

Decorator em Java

```
public abstract class DecoradorConcretoUm extends Decorador {
    public DecoradorConcretoUm (Componente componente) {
        super(componente);
    }
    public String getDadosComoString() {
        return getDados().toString();
    }
    private Object transformar(Object o) {
        ...
    }
    public Object getDados() {
        return transformar(getDados());
    }
    public void operacao(Object arg) {
        // ... comportamento adicional
        componente.operacao(arg);
    }
}
```

```
public abstract class DecoradorConcretoUm
    extends Decorador {
    private Object estado;
    public DecoradorConcretoUm (Componente comp,
        Object estado) {
        super(comp);
        this.estado = estado;
    }
    ...
    public void operacao(Object arg) {
        // ... comportamento adicional
        super.operacao(estado);
        // ...
    }
}
```

```
public class ComponenteConcreto implements Componente {
    private Object dados;
    public Object getDados() {
        return dados;
    }
    public void operacao(Object arg) {
        ...
    }
}
```

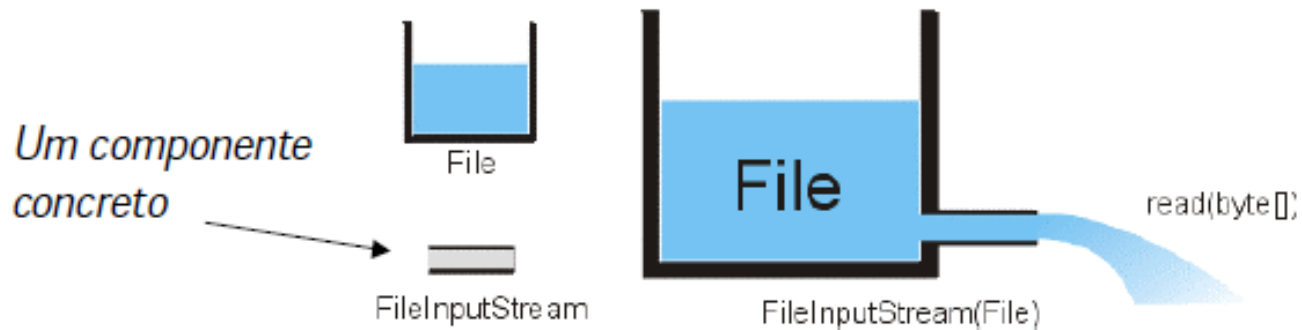
```
public interface Componente {
    Object getDados();
    void operacao(Object arg);
}
```

```
public abstract class Decorador implements Componente {
    private Componente componente;
    public Decorador(Componente componente) {
        this.componente = componente;
    }
    public Object getDados() {
        return componente.getDados();
    }
    public void operacao(Object arg) {
        componente.operacao(arg);
    }
}
```

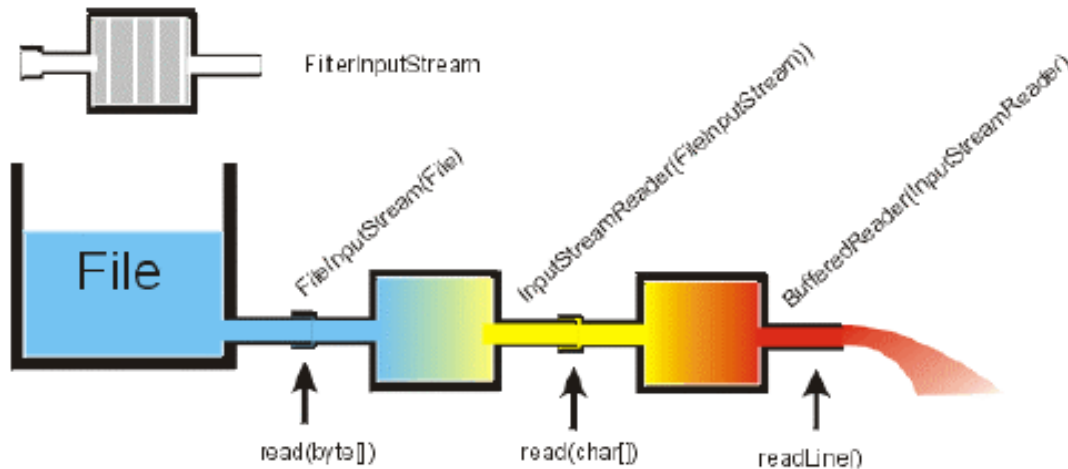

Decorator no J2SDK

- ▶ Embora na literatura sobre design patterns (GoF) a maior parte das aplicações apresentadas para uso de decoradores seja em aplicações gráficas, em Java o Swing usa outras abordagens
 - Ex: ScrollPane "decora" um TextArea, mas as chamadas não são feitas através do ScrollPane
- ▶ Em Java, o uso mais comum de decoradores é nos objetos que representam fluxos de entrada e saída (I/O streams)
 - java.io: InputStream, OutputStream, Reader, Writer, etc.

I/O Streams



```
// objeto do tipo File
File tanque = new File("agua.txt");
// componente FileInputStream
FileInputStream cano = new FileInputStream(tanque);
// read() lê um byte a partir do cano
byte octeto = cano.read();
```



Concatenação de I/O streams

Concatenação do decorador

```
// partindo do cano (componente concreto)
FileInputSteam cano = new FileInputSteam(tanque);

// decorador chf conectado no componente
InputStreamReader chf = new InputStreamReader(cano);
```

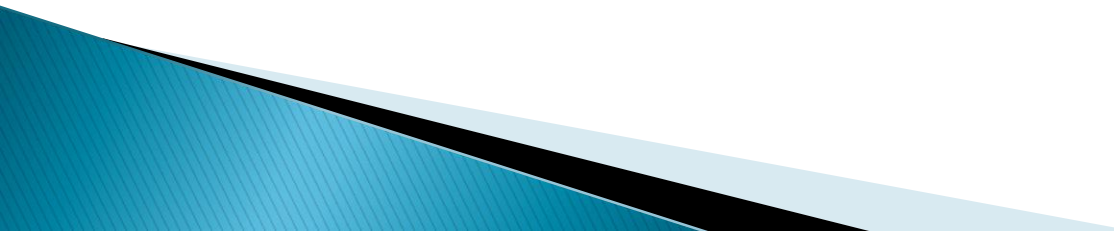
```
// pode-se ler um char a partir de chf (mas isto impede que
// o char chegue ao fim da linha: há um vazamento no cano!)
char letra = chf.read();
```

Uso de método com comportamento alterado

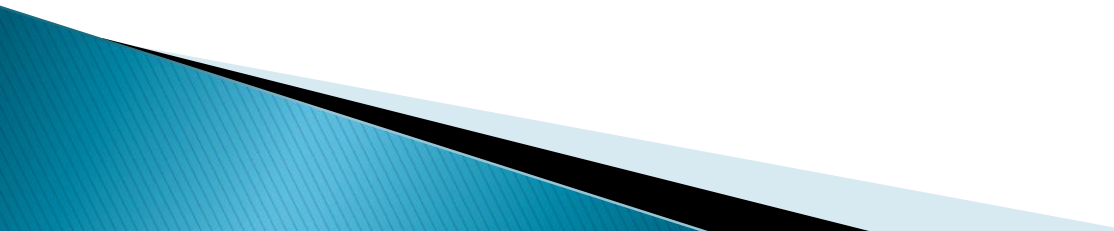
```
// decorador br conectado no decorador chf
BufferedReader br = new BufferedReader (chf);
// lê linha de texto a de br
String linha = br.readLine();
```

Comportamento adicional

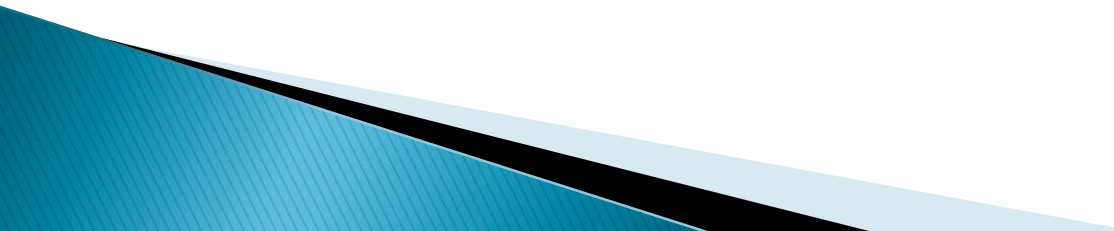
Aplicabilidade

- ▶ Quando você quer adicionar responsabilidades a objetos individuais dinamicamente e transparentemente, sem afetar o objeto original ou outros objetos.
 - ▶ Quando você quer adicionar responsabilidades ao objeto que você possa querer mudar no futuro.
 - ▶ Quando fazer extensão por subclassing estático for impraticável.
- 

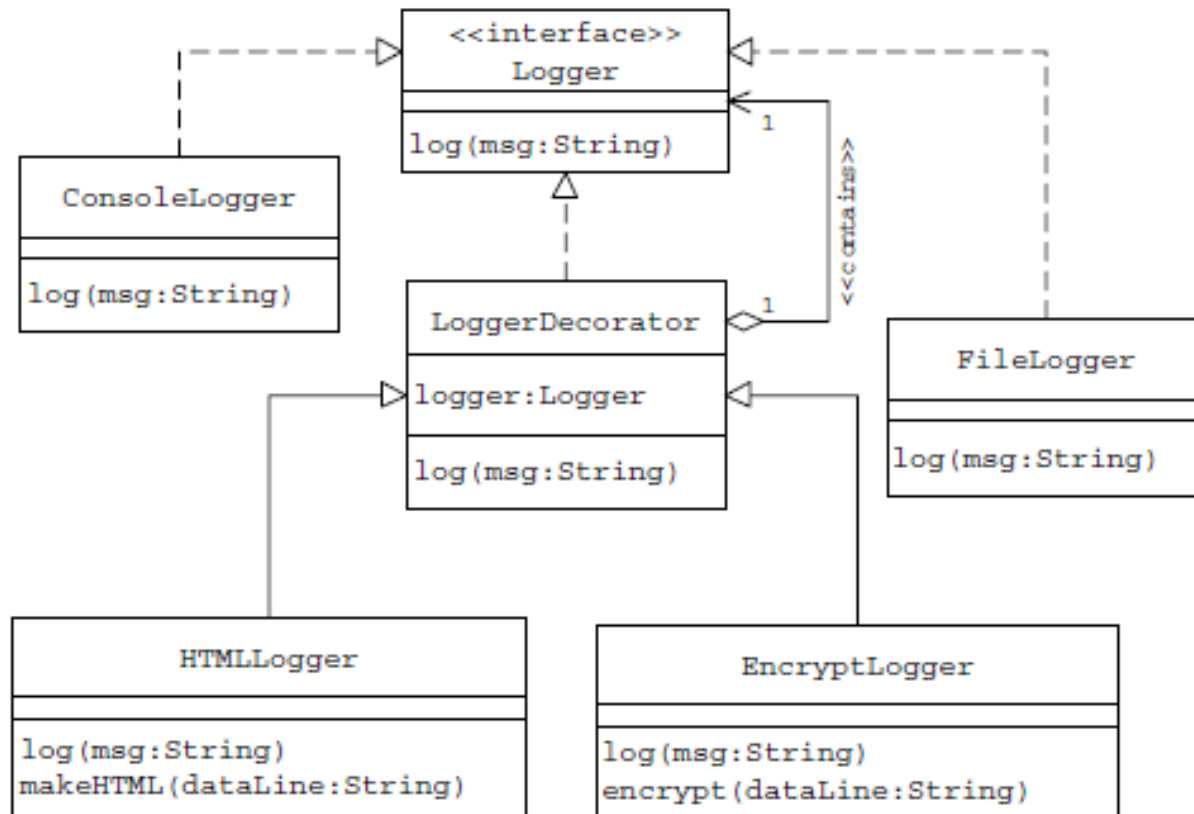
Benefícios

- ▶ Mais flexível do que herança estática
 - ▶ Evita classes pesadas (com muitas características) mais altas na hierarquia
 - ▶ Simplifica codificação porque escrevemos uma série de classes, cada uma direcionada para uma parte específica da funcionalidade ao invés de codificar todo o comportamento no objeto.
 - ▶ Aumenta a extensibilidade do objeto porque fazemos as mudanças ao codificar novas classes.
- 

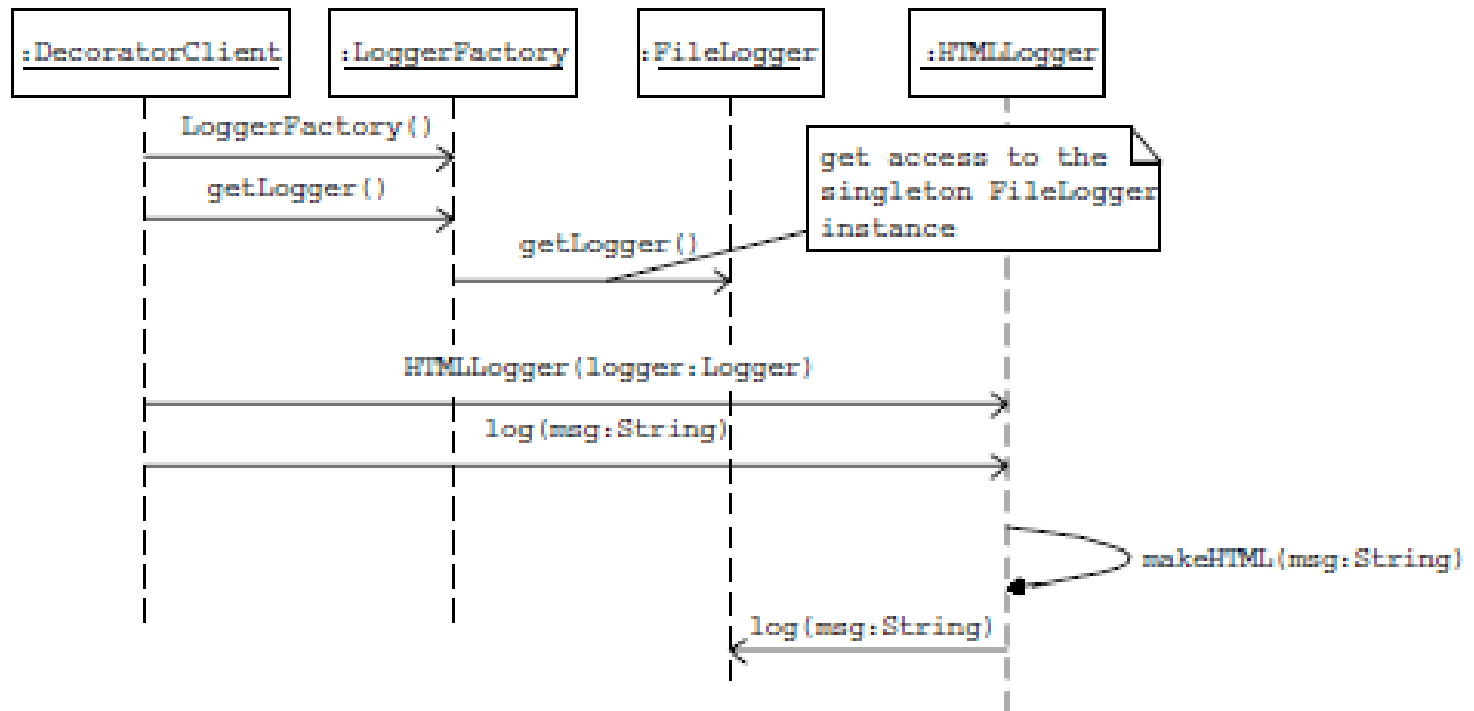
Consequências

- ▶ É necessário manter a interface do componente simples. Devemos evitar criar uma interface complexa, porque isso fará ficar muito mais difícil criar os Decorators corretos.
 - ▶ Overhead de desempenho associado a uma corrente longa de Decorators.
- 

Exemplo



Exemplo (cont.)



Exemplo (cont.)

```
package com.logica.decorator;  
public interface Logger {  
    public void log(String msg);  
}
```

```
package com.logica.decorator;  
public class LoggerDecorator implements Logger {  
    Logger logger;  
    public LoggerDecorator(Logger logger) {  
        super();  
        this.logger = logger;  
    }  
    public void log(String msg) {  
        logger.log(msg);  
    }  
}
```

Exemplo (cont.)

```
package com.logica.decorator;  
public class ConsoleLogger implements Logger {  
    public void log(String msg) {  
        System.out.println("Salva os dados: "+msg);  
    }  
}
```

```
package com.logica.decorator;  
    public class EncryptLogger extends LoggerDecorator {  
        public EncryptLogger(Logger logger) {  
            super(logger);  
        }  
        public void log(String msg) {  
            msg = encrypt(msg);  
            logger.log(msg);  
        }  
        private String encrypt(String msg) {  
            msg = msg.substring(msg.length()-1) + msg.substring(0,  
            msg.length() -1);  
            return msg;  
        }  
    }  
}
```

Exemplo (cont.)

```
package com.logica.decorator;
public class HTMLLogger extends LoggerDecorator {
    public HTMLLogger(Logger logger) {
        super(logger);
    }
    public void log(String msg) {
        msg = makeHTML(msg);
        logger.log(msg);
    }
    private String makeHTML(String msg) {
        msg = "<html><body>" + "<b>" + msg + "</b>"
        + "</body></html>";
        return msg;
    }
}
```

Exemplo (cont.)

```
package com.logica.decorator;
public class LoggerFactory {
    public static final String TYPE_CONSOL_LOGGER = "console";
    public static final String TYPE_FILE_LOGGER = "file";
    public Logger getLogger(String type) {
        if(TYPE_CONSOL_LOGGER.equals(type)) {
            return new ConsoleLogger();
        } else {
            return new FileLogger();
        }
    }
}
```

Exemplo (cont.)

```
package com.logica.decorator;
public class DecoratorClient {
    public static void main(String[] args) {
        LoggerFactory factory = new LoggerFactory();
        Logger logger =
factory.getLogger(LoggerFactory.TYPE_FILE_LOGGER);
        HTMLLogger htmlLogger = new HTMLLogger(logger);
        htmlLogger.log("A message to log");
        EncryptLogger encryptLogger = new EncryptLogger(logger);
        encryptLogger.log("A message to log");
    }
}
```

Adapter

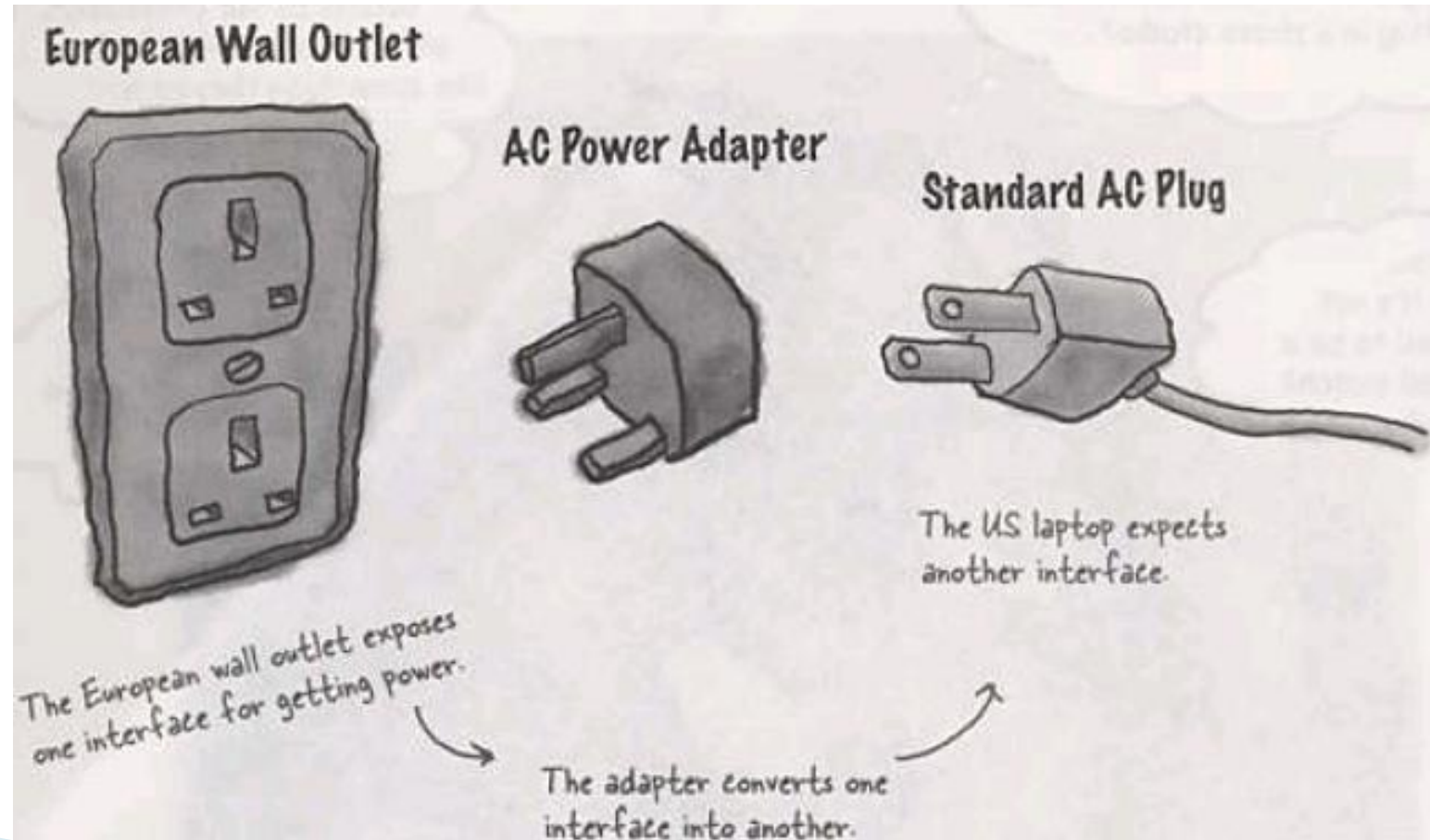
Padrão estrutural



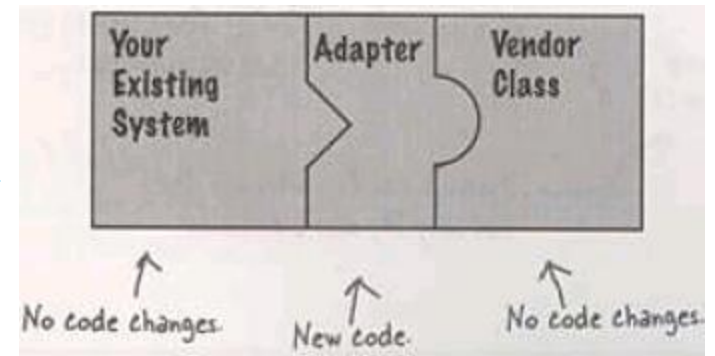
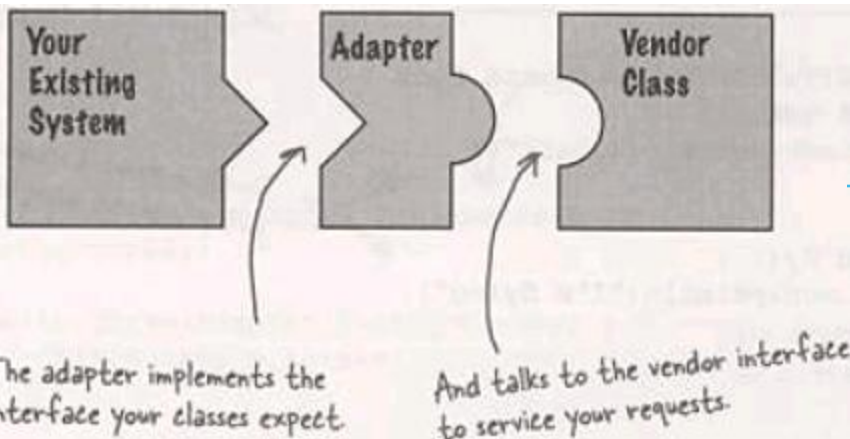
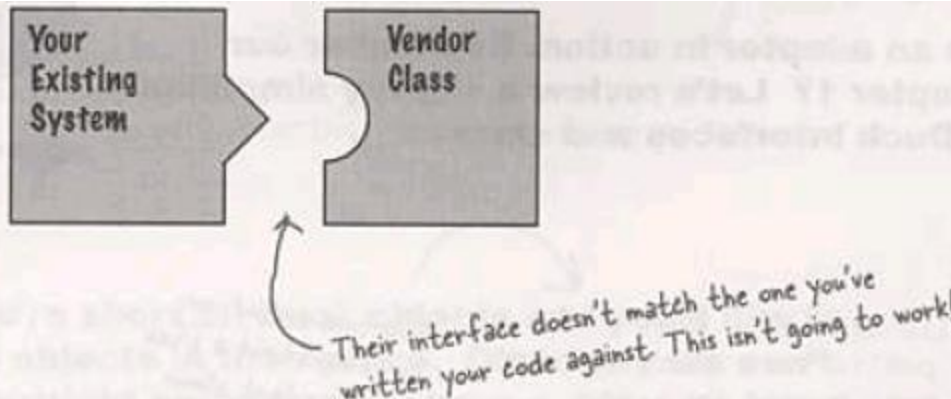
Definição

- ▶ É usado para traduzir a interface de uma classe em outra interface.
- ▶ Podemos fazer com que classes trabalhem juntas – o que não poderiam fazer devido a interfaces incompatíveis.
- ▶ Um adaptador de classe usa herança múltipla (ao estender uma classe e/ou implementar uma ou mais classes) para adaptar uma interface em outra.
- ▶ Também conhecido como
 - Wrapper.

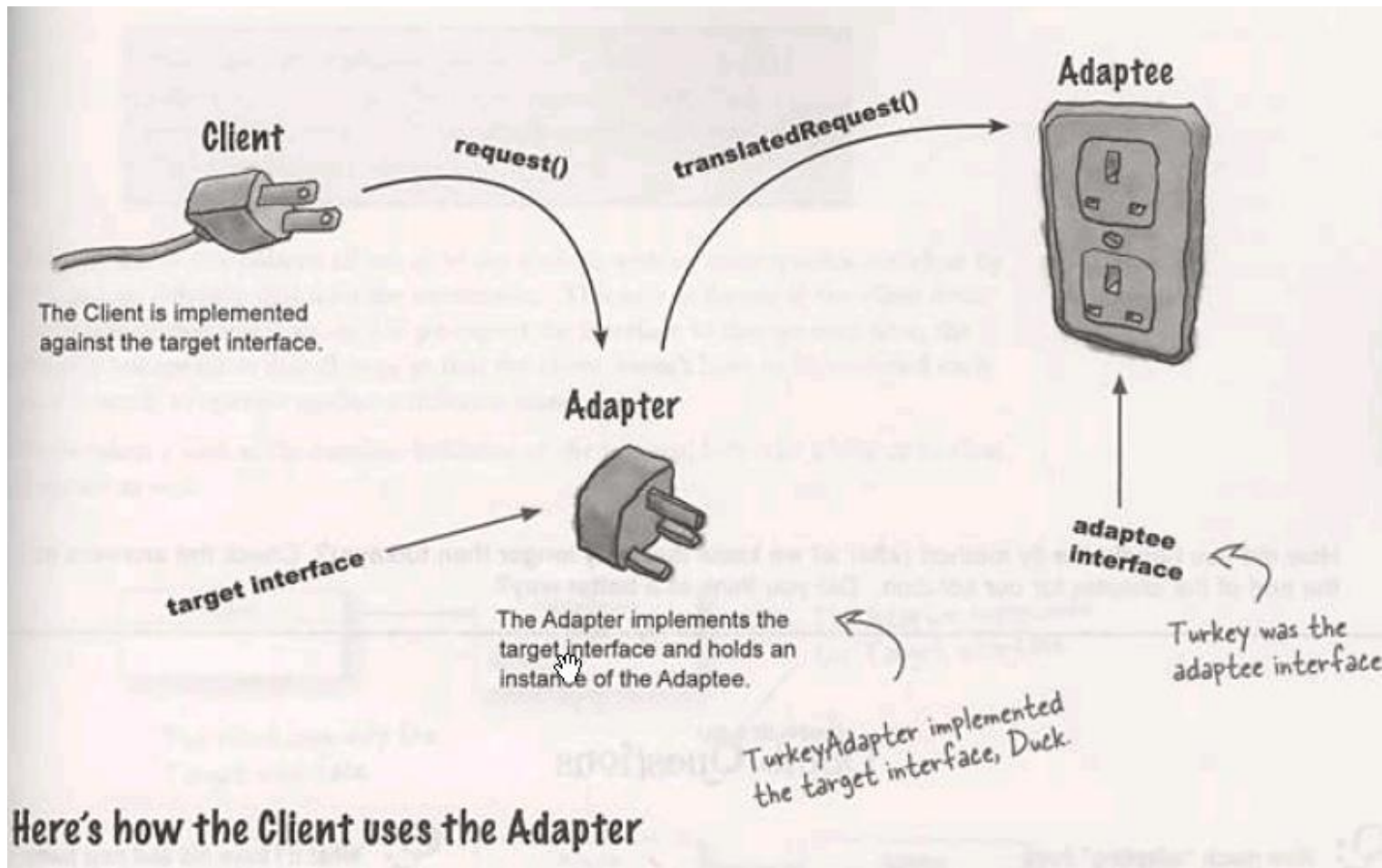
Motivação



Motivação



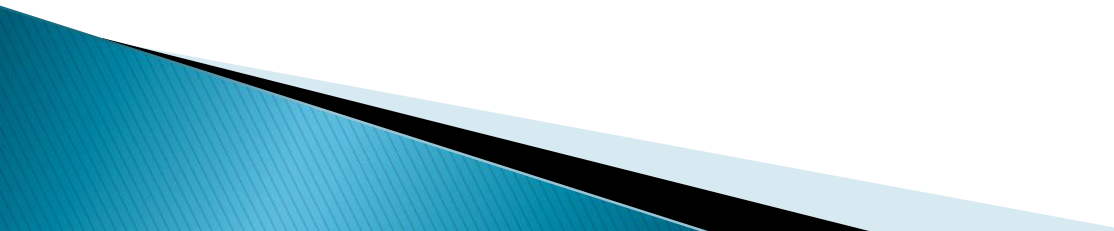
Motivação



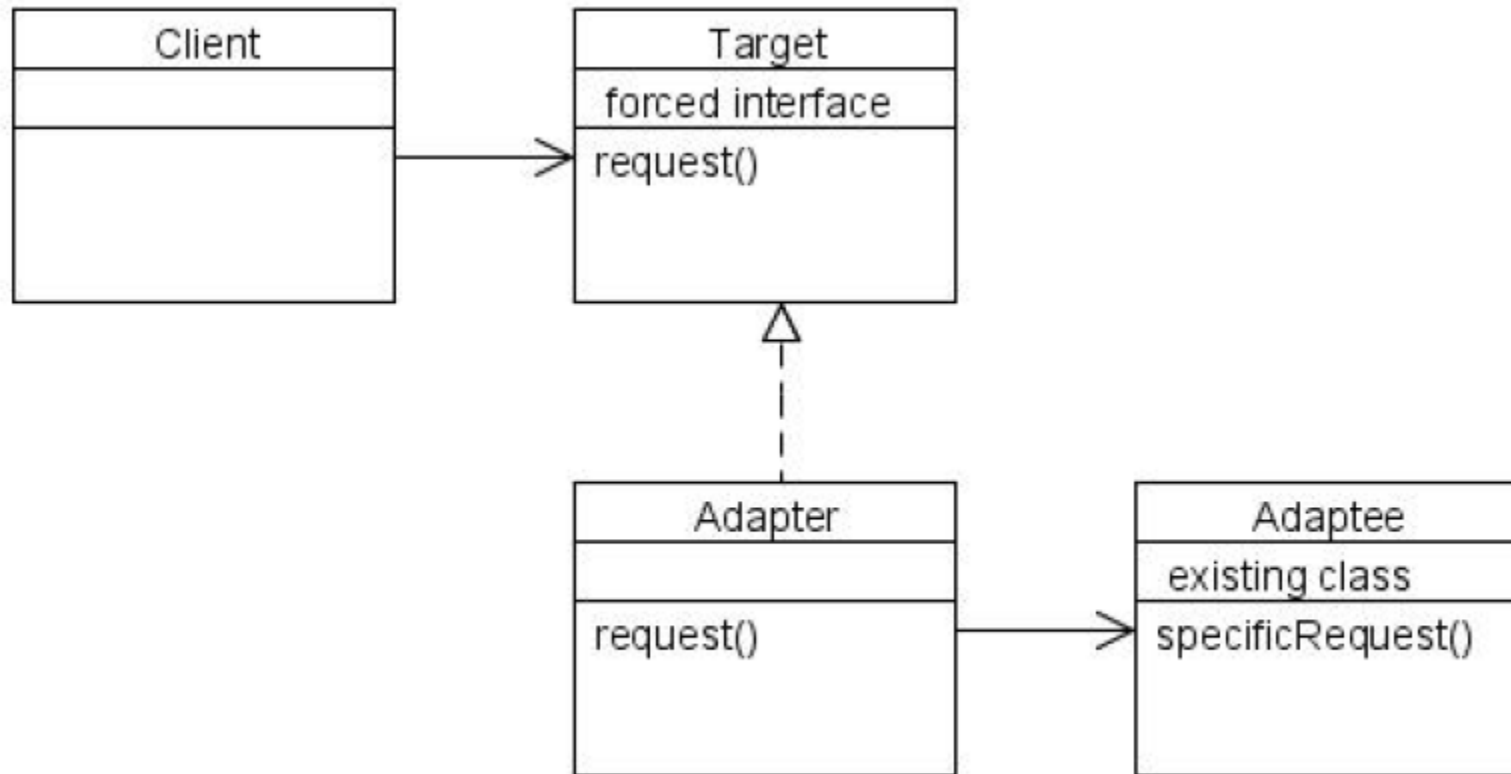
Exemplos de Adapter

- ▶ JSDK API: Tratamento de eventos (java.awt.event)
 - MouseAdapter, WindowAdapter, etc. são stubs para implementação de adapters
- ▶ JSDK API: Wrappers de tipos em Java
 - Double, Integer, Character, etc. "Adaptam" tipos primitivos à interface de java.lang.Object.
- ▶ Uso de JTable, JTree, JList (javax.swing)
 - A interface TableModel e as classes AbstractTableModel e DefaultTableModel oferecem uma interface para o acesso aos campos de uma Jtable
 - Um adapter é útil para traduzir operações específicas do domínio dos dados (planilha, banco de dados, etc.) às operações da tabela.

Aplicabilidade

- ▶ Quando você quiser usar uma classe existente, e sua interface não combina com a que você necessita
 - ▶ Quando você quer criar uma classe reutilizável que coopera com classes não relacionadas ou inesperadas, isto é, que não possuem interfaces compatíveis.
 - ▶ Quando você quer fazer um kit plugável.
- 

Estrutura



Participantes

▶ Interface Alvo

- A classe cliente espera uma certa interface (chamada de interface Alvo)
- A interface disponível não combina com a interface Alvo

▶ Adapter

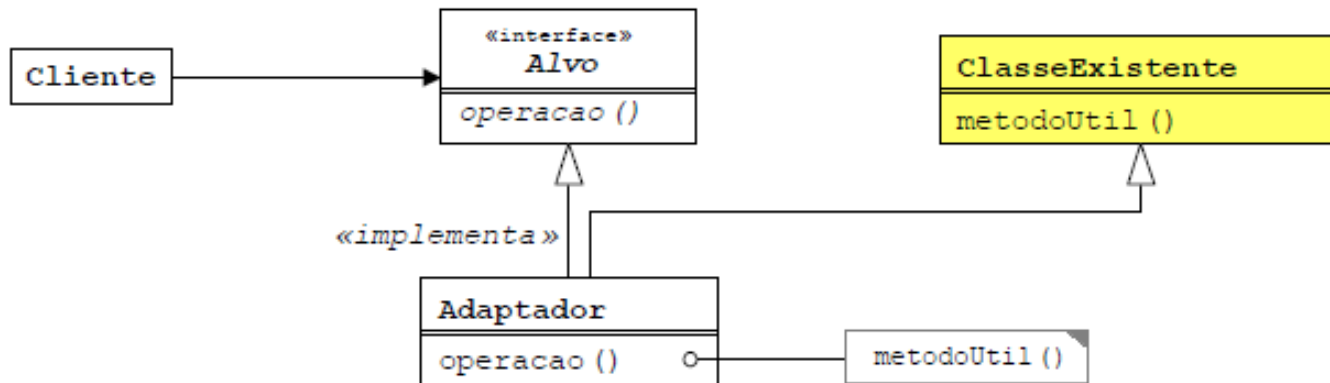
- A classe Adapter faz a ponte entre a interface alvo e a interface disponível

▶ Adaptee

- É a interface disponível

Duas formas de Adapter

- ▶ Class Adapter: usa herança múltipla



- Cliente: aplicação que colabora com objetos aderentes à interface Alvo
- Alvo: define a interface requerida pelo Cliente
- Adaptee – ClasseExistente: interface que requer adaptação
- Adaptador (Adapter): adapta a interface do recurso à interface Alvo

Class Adapter em Java

```
public class ClienteExemplo {
    Alvo[] alvos = new Alvo[10];
    public void inicializaAlvos() {
        alvos[0] = new AlvoExistente();
        alvos[1] = new Adaptador();
        // ...
    }
    public void executaAlvos() {
        for (int i = 0; i < alvos.length; i++) {
            alvo.operacao();
        }
    }
}
```

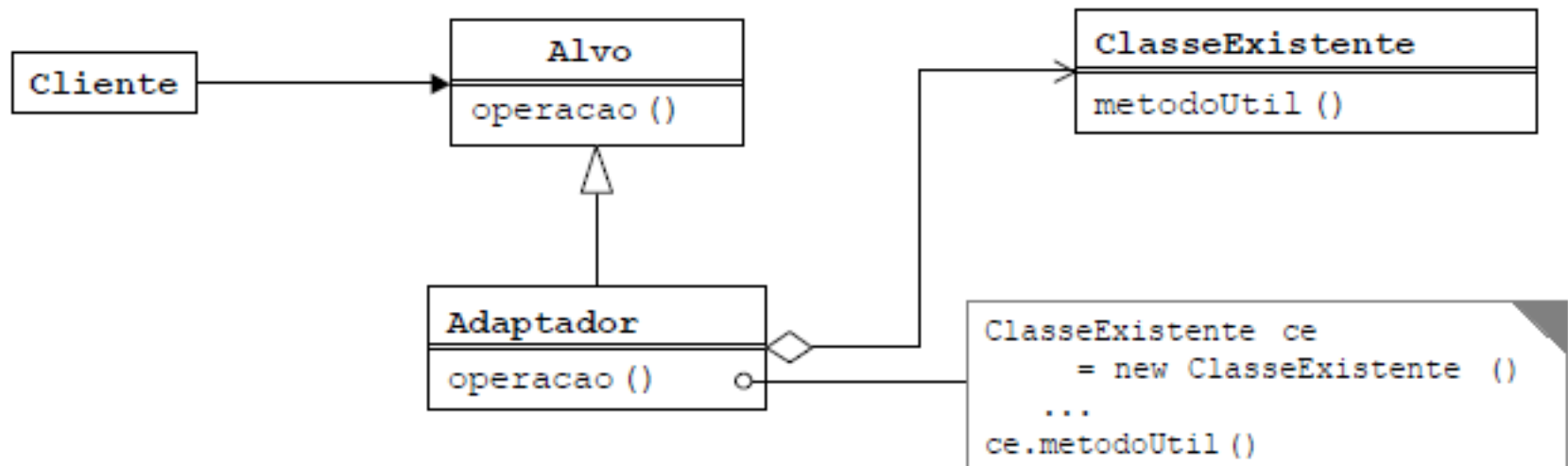
```
public interface Alvo {
    void operacao();
}
```

```
public class Adaptador extends ClasseExistente implements Alvo {
    public void operacao() {
        String texto = metodoUtilDois("Operação Realizada.");
        metodoUtilUm(texto);
    }
}
```

```
public class ClasseExistente {
    public void metodoUtilUm(String texto) {
        System.out.println(texto);
    }
    public String metodoUtilDois(String texto) {
        return texto.toUpperCase();
    }
}
```


Duas formas de Adapter (cont.)

- ▶ Object Adapter: usa agregação



- ▶ Cada método de Alvo chama o(s) método(s) correspondente(s) na interface adaptada.

Object Adapter em Java

```
public class ClienteExemplo {
    Alvo[] alvos = new Alvo[10];
    public void inicializaAlvos() {
        alvos[0] = new AlvoExistente();
        alvos[1] = new Adaptador();
        // ...
    }
    public void executaAlvos() {
        for (int i = 0; i < alvos.length; i++) {
            alvos[i].operacao();
        }
    }
}
```

```
public abstract class Alvo {
    public abstract void operacao();
    // ... resto da classe
}
```

```
public class Adaptador extends Alvo {
    ClasseExistente existente = new ClasseExistente();
    public void operacao() {
        String texto = existente.metodoUtilDois("Operação Realizada.");
        existente.metodoUtilUm(texto);
    }
}
```

```
public class ClasseExistente {
    public void metodoUtilUm(String texto) {
        System.out.println(texto);
    }
    public String metodoUtilDois(String texto) {
        return texto.toUpperCase();
    }
}
```

Quando usar?

- ▶ **Class Adapter**
 - Quando houver uma interface que permita a implementação estática
- ▶ **Object Adapter**
 - Quando menor acoplamento for desejado
 - Quando o cliente não usa uma interface Java ou classe abstrata que possa ser estendida

Benefícios

- ▶ Altamente reutilizável.
- ▶ Introduz somente 1 objeto.

- ▶ **Consequências**
 - Quando usar Java, o Alvo deve ser uma interface.

Exemplo

```
public interface FileManager {  
    public String open(String s);  
    public String close();  
    public String read(int pos, int amount, byte[] data);  
    public String write(int pos, int amount, byte[] data);  
}
```

```
import java.util.*;  
import java.io.*;  
public class FileManagerUtil {  
    private RandomAccessFile f;  
    public boolean openFile(String fileName) {  
        System.out.println("Opening file: "+fileName);  
        boolean success=true;  
        return success;  
    }  
    public boolean closeFile() {  
        System.out.println("Closing file");  
        boolean success=true;  
        return success;  
    }  
}
```

Exemplo (cont.)

```
public boolean writeToFile(String d, long pos, long amount) {
    System.out.print("Writing "+amount+" chars from string: "+d);
    System.out.println(" to pos: "+pos+" in file");
    boolean success=true;
    return success;
}
public String readFromFile(long pos, long amount) {
    System.out.print("Reading "+amount+" chars from pos: "+pos+" in file");
    return new String("dynamite");
}
}
```

Exemplo (cont.)

```
public class FileManagerImpl extends FileManagerUtil implements FileManager {
    public String close() {
        return new Boolean(closeFile()).toString();
    }
    public String open(String s) {
        return new Boolean(openFile(s)).toString();
    }
    public String read(int pos, int amount, byte[] data) {
        return readFromFile(pos, amount);
    }
    public String write(int pos, int amount, byte[] data) {
        boolean tmp= writeToFile(new String(data), pos, amount);
        return String.valueOf(tmp);
    }
}
```

Exemplo (cont.)

```
public class FileManagerClient {
    public static void main(String[] args) {
        FileManager f = null;
        String dummyData = "dynamite";
        f = new FileManagerImpl();
        System.out.println("Using filemanager: "+f.getClass().toString());
        f.open("dummyfile.dat");
        f.write(0, dummyData.length(), dummyData.getBytes());
        String test = f.read(0,dummyData.length(),dummyData.getBytes());
        System.out.println("Data written and read: "+test);
        f.close();
    }
}
```


Exercício 1

- ▶ Qual a diferença entre
 - Adapter e Decorator
 - Adapter e Proxy