

Laboratório de Engenharia de Software

INF1636 – PROGRAMAÇÃO ORIENTADA A OBJETOS

Departamento de Informática – PUC-Rio

Ivan Mathias Filho
ivan@inf.puc-rio.br

Programa – Capítulo 17



Laboratório de Engenharia de Software

- Padrões de Design
- Singleton
- Facade
- Factory Method
- Observer
- Strategy
- Adapter

© LES/PUC-Rio

Programa – Capítulo 17



- **Padrões de Design**
- Singleton
- Facade
- Factory Method
- Observer
- Strategy
- Adapter

Definições



- Os Padrões de Design representam soluções reutilizáveis para problemas recorrentes [Grand, 1998];
- Os Padrões de Design formam um conjunto de regras que descrevem como realizar certas tarefas no âmbito do desenvolvimento de software [Pree, 1994];
- Um Padrão de Design visa resolver um problema recorrente de design que surge em determinadas situações [Buschmann et al., 1996];
- Os Padrões de Design identificam e definem abstrações que estão acima do nível de uma única classe e de suas instâncias, ou de componentes [Gamma et al., 1994].

Princípios



- Novos problemas são geralmente similares a problemas já resolvidos anteriormente;
- As soluções para problemas similares seguem padrões recorrentes;
- Definem um vocabulário comum entre os projetistas, o que ajuda na disseminação de soluções bem sucedidas.

Breve Histórico



- Os Padrões de Design foram baseados nos trabalhos publicados pelo arquiteto Christopher Alexander no final da década de 1970;
- Em 1987, Ward Cunningham e Kent Beck usaram algumas das idéias de Alexander no trabalho sobre GUI intitulado *"Using Pattern Languages for Object-Oriented Programs"* [OOPSLA-87];
- Em 1994, Erich Gamma, Richard Helm, John Vlissides e Ralph Johnson publicaram um dos livros mais importantes de Engenharia de Software da década de 90: *"Design Patterns: Elements of Reusable Object-Oriented Software"* [GoF].

Herança vs Composição (1)



- A herança é um mecanismo de reutilização *caixa branca*, pois expõe freqüentemente a estrutura das classes ancestrais;
- A herança é um mecanismo estático, não permitindo assim a *reconfiguração* dinâmica de um sistema;
- A herança cria um forte acoplamento entre uma classe e as suas classes ancestrais, diminuindo assim a possibilidade de reutilização de uma classe em um outro projeto.

Herança vs Composição (2)



- A composição nos permite obter funcionalidades complexas através da colaboração de vários objetos que implementam funções mais simples;
- A composição é um mecanismo de reutilização *caixa preta*, pois os detalhes internos dos objetos não precisam ser expostos;
- A composição permite a *reconfiguração* dinâmica de um sistema.

Herança vs Composição (3)



- A composição aumenta as chances da reutilização de classes, pois favorece a criação de classes menores e mais coesas;
- O poder da composição aumenta ainda mais quando usado conjuntamente com o mecanismo de polimorfismo;
- Para tal, prefira o mecanismos de herança de interface em relação à de implementação.

Delegação



- A **delegação** é uma maneira de tornar a composição um mecanismo de reutilização extremamente poderoso;
- Na **delegação**, um objeto recebe uma solicitação e delega a sua execução a um ou mais objetos;
- O objeto receptor atua freqüentemente como coordenador da execução de uma solicitação;
- Deste modo, muitas vezes é necessário que os objetos delegados consultem o estado do objeto receptor (*callback*);
- Para tal, o objeto receptor passa uma referência para si próprio (*this*) quando envia mensagens para os objetos delegados.

Descrição dos Padrões



- Os **Padrões de Design** são usualmente descritos através de um formato que inclui os seguintes itens:
 - Uma descrição do problema, que inclui um exemplo concreto e uma solução para tal problema;
 - Considerações que levam à formulação de uma solução geral;
 - Uma solução geral;
 - As consequências, boas e más, do uso de uma dada solução para o problema em questão;
 - Uma lista de **Padrões** relacionados.

© LES/PUC-Rio

Programa – Capítulo 17



- Padrões de Design
- **Singleton**
- Facade
- Factory Method
- Observer
- Strategy
- Adapter

© LES/PUC-Rio

O Padrão Singleton – Objetivo



- Garantir que uma classe tenha somente uma única instância e fornecer um ponto global de acesso para tal instância;
- Algumas classes devem possuir exatamente uma instância;
- Tais classes geralmente estão envolvidas no gerenciamento de algum recurso, ou controlando alguma atividade (controller);
- O recurso pode ser externo (ex: uma conexão com um gerenciador de banco de dados) ou interno (ex: um objeto que mantém estatísticas de erro para um compilador).


O Padrão Singleton – Estrutura (1)



- O padrão Singleton é relativamente simples, uma vez que ele envolve uma única classe;
- A classe unitária possui uma variável estática que mantém uma referência para a única instância que se deseja manipular;
- Esta instância é criada quando a classe é carregada na memória ou quando ocorrer a primeira tentativa de acesso à instância.

Laboratório de Engenharia de Software

O Padrão Singleton – Estrutura (2)




- A classe unitária não pode permitir a criação de instâncias adicionais;
- Para tal, devemos nos assegurar que todos os construtores da classe unitária sejam declarados como privados.

Singleton
- <u>instance</u> : Singleton
- Singleton() : + <u>getInstance()</u> : Singleton

© LES/PUC-Rio

Laboratório de Engenharia de Software

O Padrão Singleton – Exemplo



```

public class CtrlVenda {

    private static CtrlVenda ctrl=null;
    private CtrlVenda()
    { }
    public static CtrlVenda getCtrlVenda()
    {
        if(ctrl==null)
            ctrl=new CtrlVenda();
        return ctrl;
    }
    public void encerraVenda()
    { }
}

public class UmaClasse {

    public void umMetodo()
    {
        CtrlVenda.getCtrlVenda().encerraVenda();
    }
}

```

CtrlVenda
- <u>ctrl</u> : CtrlVenda
- CtrlVenda() : + <u>getCtrlVenda()</u> : CtrlVenda + <u>encerraVenda()</u> : void

© LES/PUC-Rio

Programa – Capítulo 17



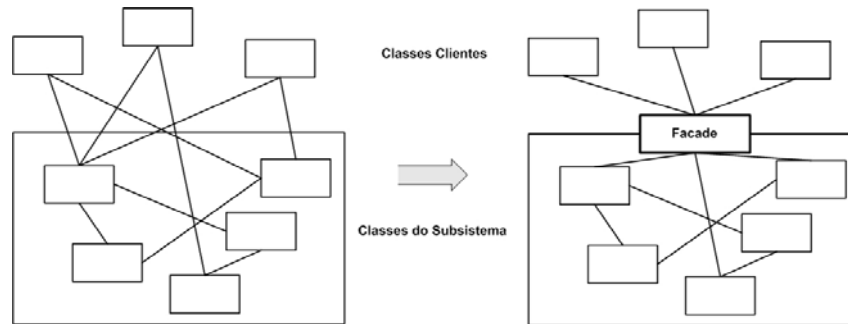
- Padrões de Design
- Singleton
- **Facade**
- Factory Method
- Observer
- Strategy
- Adapter

O Padrão Façade – Objetivo



- Um objetivo comum a todos os projetos é minimizar a comunicação e as dependências entre os subsistemas que compõem uma aplicação;
- Estruturar um sistema em subsistemas ajuda a reduzir a complexidade;
- Pode-se alcançar este objetivo introduzindo um objeto facade (fachada), que fornece uma interface única para os recursos e facilidades mais gerais de um subsistema.

O Padrão Façade – Estrutura



© LES/PUC-Rio

Programa – Capítulo 17



- Padrões de Design
- Singleton
- Facade
- **Factory Method**
- Observer
- Strategy
- Adapter

© LES/PUC-Rio

O Padrão Factory Method – Objetivo



- Criar uma classe que possa instanciar outras classes, de modo que o solicitante não dependa diretamente das classes instanciadas;
- O solicitante delega a instanciação a outro objeto e referencia as instâncias criadas através de uma interface (ou de uma classe abstrata).

O Padrão Factory Method – Exemplo

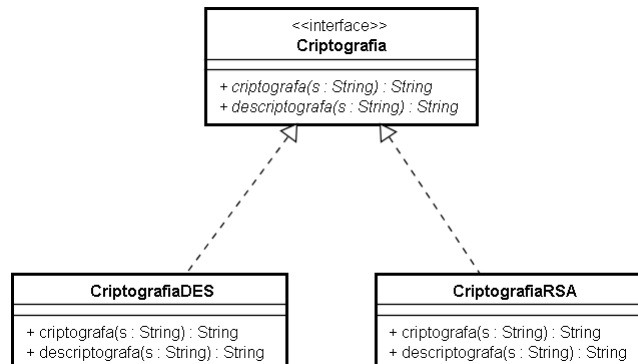


- Uma aplicação precisa criptografar suas mensagens.
- Existem vários algoritmos de criptografia que podem ser utilizados.
- O algoritmo escolhido deve ser definido em um arquivo de configuração.
- Deseja-se que a escolha do algoritmo de criptografia seja transparente às classes que irão usá-lo.

Definição da Interface



- Todas as classes que possuírem algoritmos de criptografia deverão implementar a interface Criptografia.



© LES/PUC-Rio

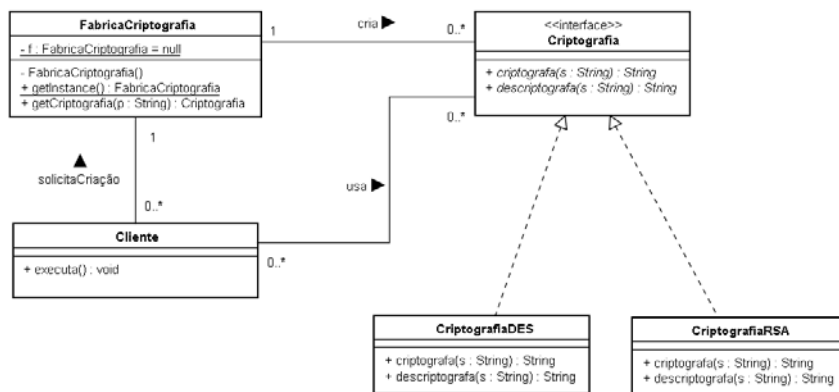
Descrição dos Passos



- Uma classe cliente precisa criptografar mensagens.
- Para tal, ela irá solicitar a instanciação de um objeto à classe-fábrica.
- O método-fábrica irá analisar o arquivo de configuração para definir qual algoritmo deverá ser usado.
- Definido o algoritmo, uma instância da classe correspondente será criada e devolvida ao cliente.
- O cliente recebe uma instância da classe adequada e a utiliza através da interface **Criptografia**.
- A verdadeira classe desse objeto é transparente ao cliente.
- A classe-fábrica será implementada como um **Singleton**.

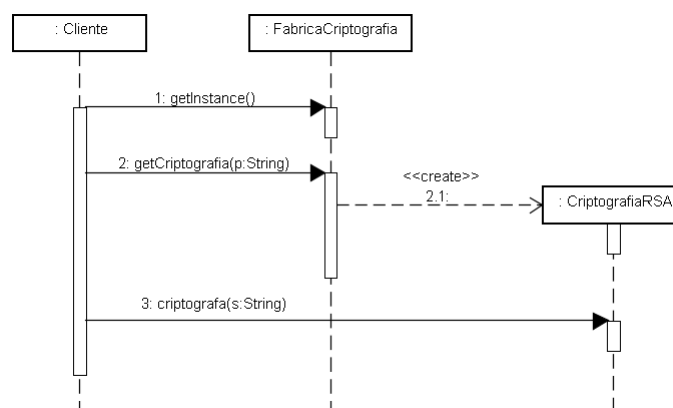
© LES/PUC-Rio

Visão Geral



© LES/PUC-Rio

Aspectos Dinâmicos



© LES/PUC-Rio

Solicitação do Cliente



```
public class Cliente {
    public void executa()
    {
        Criptografia c;
        FabricaCriptografia f;

        f=FabricaCriptografia.getInstance();
        c=f.getCriptografia("e:\\cap03.cfg");
        if(c==null)
        {
            System.out.println("Erro no arquivo de configuracao");
            System.exit(-1);
        }
        System.out.println(c.criptografa("Teste"));
        System.out.println(c.descriptografa("Teste"));
    }
}
```

Instanciação da Classe Escolhida



```
public class FabricaCriptografia {
    ...
    public Criptografia getCriptografia(String p)
    {
        Scanner s=null;
        try {
            s=new Scanner(new File(p));
            String cmd;
            while(s.hasNext()) {
                cmd=s.nextLine();
                if(cmd.lastIndexOf("CRIPTOGRAFIA")==0)
                    if((cmd.lastIndexOf("DES")>-1))
                        return new CriptografiaDES();
                else
                    if((cmd.lastIndexOf("RSA")>-1))
                        return new CriptografiaRSA();
                else
                    return null;
            }
            s.close();
        }
        catch(FileNotFoundException e)
        { return null; }
        return null; }
}
```

Programa – Capítulo 17



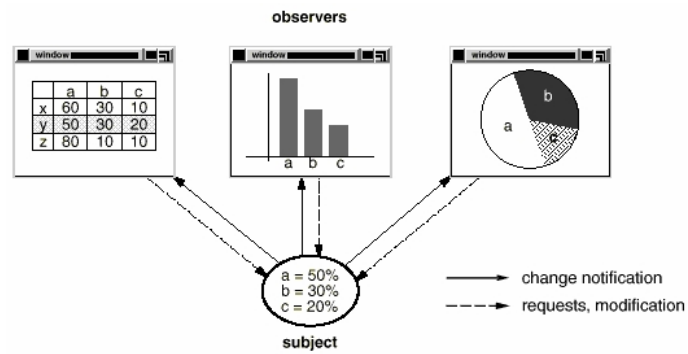
- Padrões de Design
- Singleton
- Facade
- Factory Method
- **Observer**
- Strategy
- Adapter

O Padrão Observer



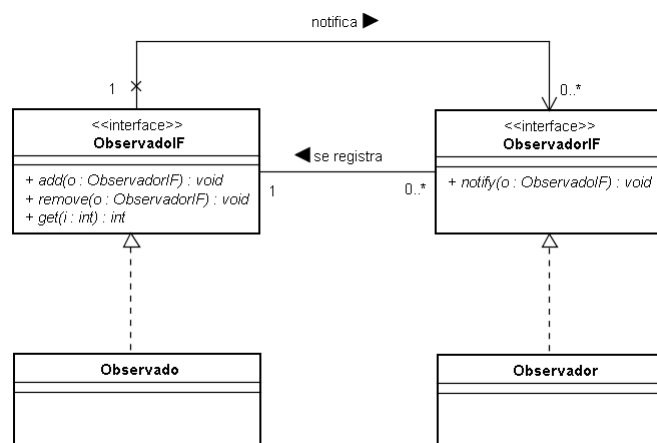
- Permite que um objeto (observado) registre dinamicamente outros objetos que dele dependem (observadores);
- Tais objetos são notificados quando houver mudanças no estado do observado;
- Isso permite que os observadores fiquem consistentes com o estado do objeto observado;
- Este padrão fornece um mecanismo flexível para a notificação de eventos.

Exemplo



© LES/PUC-Rio

Estrutura

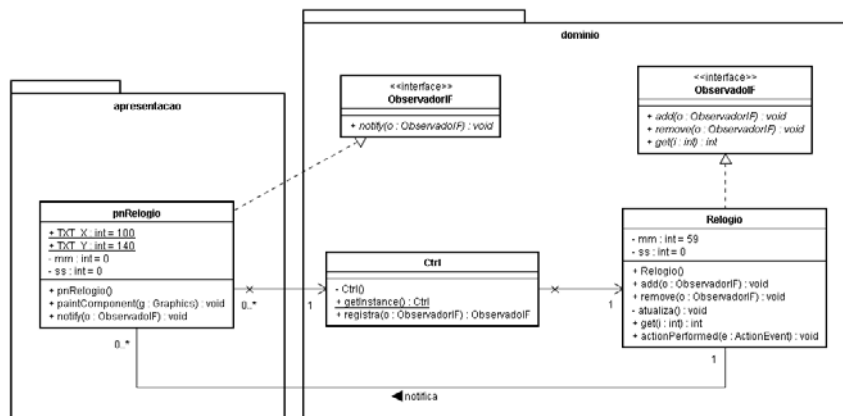


© LES/PUC-Rio

Aplicação (1)



- A figura a seguir mostra o diagrama de classes de design de um relógio que exibe as horas em uma interface gráfica:



© LES/PUC-Rio

Aplicação (2)



- O painel de exibição será notificado quando houver modificação no tempo marcado pelo relógio interno.
- Para tal, o painel irá se registrar, através de um **Controlador**, junto ao **Relógio**;
- O **Controlador**, então, repassa a solicitação para o **Relógio**, que registra o painel como observador.

© LES/PUC-Rio

Aplicação (3)



```
public class pnRelogio extends JPanel implements ObservadorIF
{
    public static final int TXT_X=100;
    public static final int TXT_Y=140;
    private int mm=0,ss=0;

    public pnRelogio()
    {
        Ctrl.getInstance().registra(this);
    }
    ...
    public void notify(ObservadorIF o)
    {
        mm=o.get(1);
        ss=o.get(2);
        repaint();
    }
}
```

Aplicação (4)



```
class Relogio implements ObservadorIF,ActionListener {

    private List<ObservadorIF> lst=new ArrayList<ObservadorIF>();
    private int mm=59,ss=0;

    public void add(ObservadorIF o)
    {
        lst.add(o);
    }
    private void atualiza()
    {
        ListIterator<ObservadorIF> li=lst.listIterator();

        while(li.hasNext())
        {
            li.next().notify(this);
        }
    }
    ...
}
```

Aplicação (5)



- Após o recebimento da notificação, o observador envia mensagens (callback) para o objeto observado com o objetivo de obter os dados desejados.

```
public class pnRelogio extends JPanel implements ObservadorIF
{
    public static final int TXT_X=100;
    public static final int TXT_Y=140;
    private int mm=0,ss=0;

    public pnRelogio()
    {
        Ctrl.getInstance().registra(this);
    }
    ...
    public void notify(ObservadoIF o)
    {
        mm=o.get(1);
        ss=o.get(2);
        repaint();
    }
}
```

© LES/PUC-Rio

Aplicação (6)



- O objeto observado, então, retorna as informações solicitadas.

```
class Relogio implements ObservadoIF,ActionListener {
    ...
    public int get(int i)
    {
        if(i==1)
            return mm;
        else
            return ss;
    }
    ...
}
```

© LES/PUC-Rio

Programa – Capítulo 17




Laboratório de Engenharia de Software

- Padrões de Design
- Singleton
- Facade
- Factory Method
- Observer
- **Strategy**
- Adapter

© LES/PUC-Rio

O Padrão Strategy

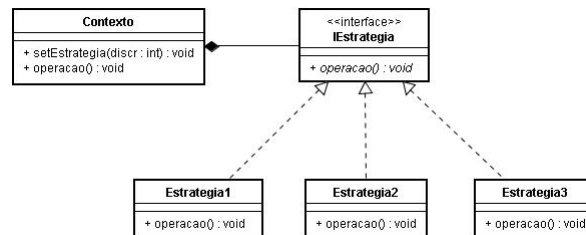


Laboratório de Engenharia de Software

- Tem por objetivo definir uma família de algoritmos para uma certa operação;
- Cada algoritmo da família é encapsulado em um objeto;
- Permite a mudança dinâmica dos algoritmos, independentemente dos clientes que os utilizam.

© LES/PUC-Rio

Estrutura Geral



© LES/PUC-Rio

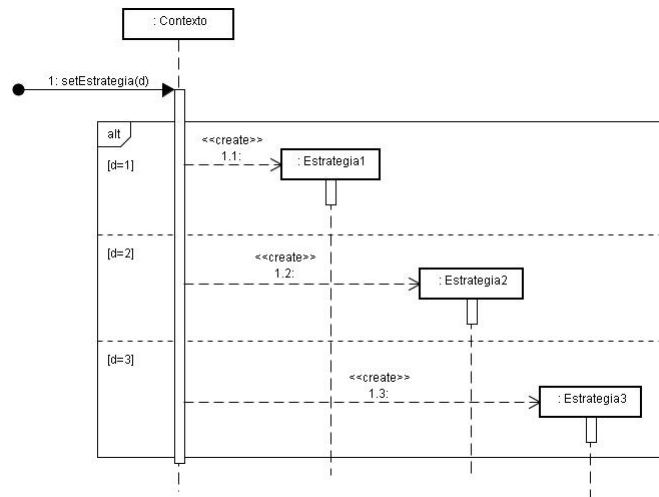
O Padrão – Comentários



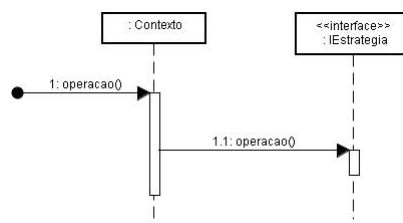
- O padrão **Strategy** utiliza a composição em vez de herança;
- Isso permite uma melhor separação entre o comportamento e as classes que usam o comportamento;
- Os clientes podem solicitar a mudança de do comportamento em tempo de execução;
- Novos comportamentos podem ser acrescentados sem alterações significativas no código existente.

© LES/PUC-Rio

Definição da Estratégia



Definição da Estratégia



Programa – Capítulo 17



- Padrões de Design
- Singleton
- Facade
- Factory Method
- Observer
- Strategy
- **Adapter**

O Padrão Adapter – Objetivos

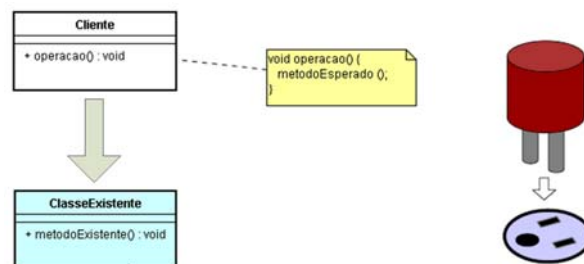


- Converter a interface de uma classe em outra interface, esperada pelos clientes.
- Adapter permite que classes trabalhem juntas, o que não poderia ser feito sem o uso desse padrão, por causa da incompatibilidade de interfaces.
- Envolve uma classe existente em uma nova interface.
- Ajusta um componente antigo para operar em um novo sistema.

Problemas

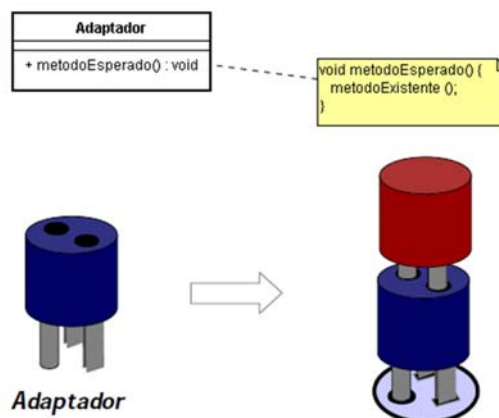


- Componentes de prateleira oferecem funcionalidades interessantes poderiam ser usadas em novos sistemas.
- Entretanto, concepções sobre o ambiente operacional tornam esses componentes incompatíveis com a filosofia e arquitetura do sistema a ser desenvolvido.



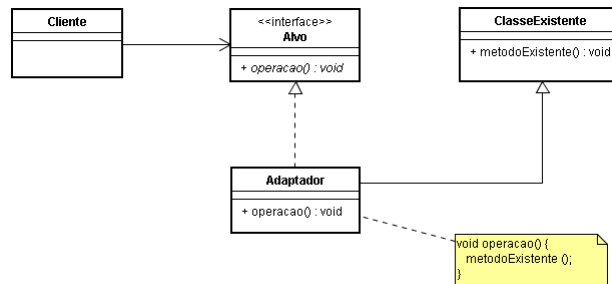
© LES/PUC-Rio

Solução



© LES/PUC-Rio

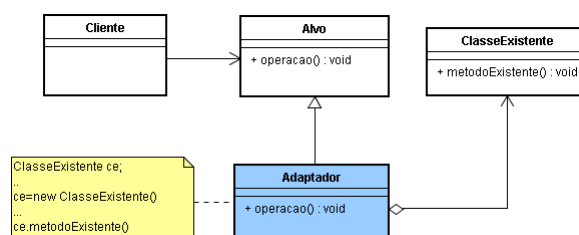
Class Adapter: Herança + Implementação de Interface



- **Cliente:** aplicação que colabora com objetos aderentes à interface Alvo.
- **Alvo:** define a interface requerida pelo Cliente.
- **ClasseExistente:** classe que requer adaptação.
- **Adaptador (Adapter):** adapta a interface do componente existente à interface Alvo.

© LES/PUC-Rio

Object Adapter: Herança + Composição



- Alvo pode ser uma classe ou uma interface.
- Adaptador possui referência para objeto que terá sua interface adaptada (instância de ClasseExistente).
- Cada método de Adaptador chama o(s) método(s) adequado(s) na(s) classe(s) existente(s).
- Permite utilizar vários componentes pré-existentes.

© LES/PUC-Rio