

## Padrões de Design Abstract Factory

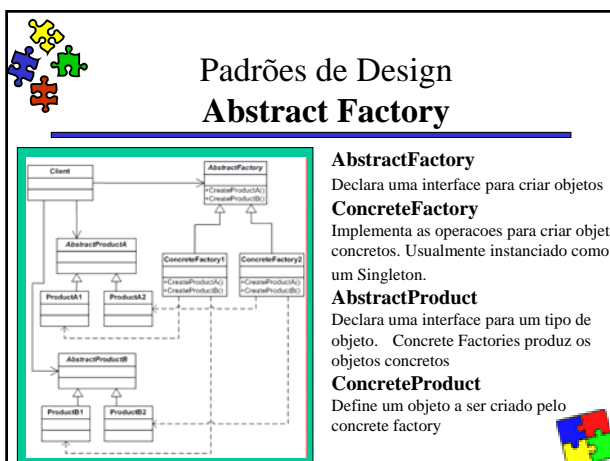
- Provê uma interface para criação de **famílias de objetos** relacionados ou dependentes
- Responsabilidade: Criar Objetos
- Benefícios
  - Há limites de QUEM pode criar objetos, COMO eles são criados e QUANDO eles são criados (Restrições de Criação)
  - Código de criação localizado em uma classe → menos chance de erros
  - Cliente desacoplado

## Padrões de Design Abstract Factory

- **Problema:** Família de Objetos precisam ser criados
- **Solução:** Coordena a criação de famílias de objetos
- **Participantes e Colaboradores:**
  - Abstract Factory: define a interface de como criar cada membro da família
  - Concrete Factory (CF): cria cada família (um CF para cada família criada)
- **Implementação:**
  - Define uma classe abstrata que especifica quais objetos serão criados
  - Implementa uma classe concreta para cada família


## Padrões de Design Abstract Factory

- **Aplicabilidade:**
  - Um sistema deve ser independente de como seus objetos são criados, compostos ou representados
  - Uma família de objetos foi projetada para ser usada em conjuntos e é necessário garantir essa restrição



## Abstract Factory Exemplo (from Brent Ramerth)

- Gerenciar endereços e número de telefones
  - Fixo para endereços no USA
  - Se você deseja ampliá-lo para incorporar qualquer endereço/fone
- Subclasses
  - DutchAddress, JapanesePhoneNumber, etc.



## Abstract Factory Exemplo

---



- Defina um *Factory* e especifique os objetos abstraindo os *factories*.

```

public class USAddressFactory implements
AddressFactory{
    public Address createAddress(){
        return new USAddress();
    }

    public PhoneNumber createPhoneNumber(){
        return new USPhoneNumber();
    }
}



```

## Padrões de Design Singleton


---



- Problema:** Uma classe tem apenas **uma instância** e provê um ponto de acesso global a essa instância
- Solução:** Criar um método estático *getInstance()*. Quando o método for acessado pela primeira vez, cria a instancia do objeto e retorna uma referencia para o objeto criado. Em acessos subsequentes nenhuma instancia é criada e a referencia ao objeto existente é retornada.
- Participantes e Colaboradores:**
  - Singleton: define uma operação *Instance* que permite que o cliente acesse apenas sua única instância. Instance é uma operação da classe.

## Padrões de Design Singleton

---



Singleton
Static Instance()  Retorna instancia única SingletonOperation() GetSingletonData() <hr/> Static uniqueInstance singletonData

## Padrões de Design Adapter

---



- Problema:** Uma classe muitas vezes não é reusada porque sua interface não corresponde à interface específica de uma aplicação
- Solução:** Converter a interface de uma classe em outra interface, permitindo que classes incompatíveis trabalhem em conjunto
- Participantes e Colaboradores:**
  - Target (Destino): define a interface específica que o cliente usa
  - Cliente: colabora com objetos compatíveis com a interface de Target
  - Adaptee (Adaptado): define uma interface que vai ser adaptada
  - Adapter (Adaptador): adapta a interface do Adaptee à interface Target

## Padrões de Design Adapter

---


- Aplicabilidade:**
  - Usado para prover uma nova interface para sistemas existentes (sistemas legados)
  - Quando se quer usar uma classe existente mas sua interface não corresponde a interface que necessita
  - Quando se quer criar uma classe reusável que coopera com classes não previstas

## Padrões de Design Adapter

---

- Dois padrões Adapter:**
  - Adaptador de Classes (Class Adapter)
    - Usa Herança Multipla para adaptar uma interface a outra
  - Adaptador de Objeto (Object Adapter)
    - Usa Herança Simples e Delegação



## Padrões de Design Adapter

---

- Exemplo

```

Class NewTime
{
public:
int getTime() {
    return ad_oldTime.getTime * 100 + 50
}
private:
OldTime ad_oldTime;
};
        
```

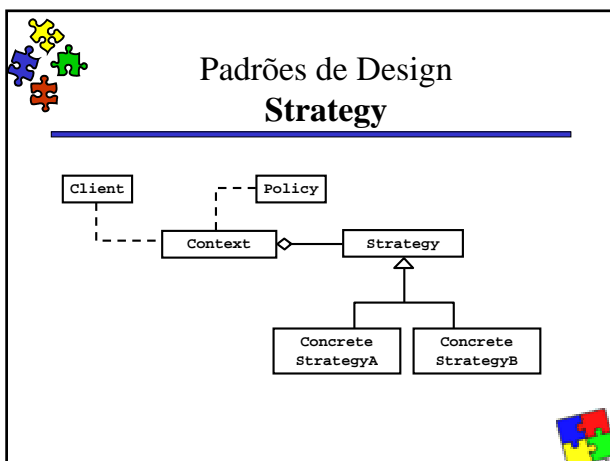
Adaptee

Adapter

## Padrões de Design Strategy

---

- Problema:** Uma classe muitas vezes não é reusada porque sua interface não corresponde à interface específica de uma aplicação
- Solução:** Definir uma família de algoritmos, encapsular cada uma delas e torná-las intercambiáveis. O Strategy permite que o algoritmo varie independentemente dos clientes que o usam.
- Participantes e Colaboradores:**
  - Strategy: define uma interface comum para todos os algoritmos suportados
  - ConcreteStrategy: implementa o algoritmo usando a interface de Strategy
  - Context: é configurado com um objeto ConcreteStrategy e mantém uma referência para o objeto Strategy.



## Padrões de Design Strategy

---

- Aplicabilidade:**
  - Usado quando muitas classes relacionadas diferem somente no seu comportamento. As estratégias fornecem uma maneira de configurar uma classe com um, dentre muitos comportamentos.
  - Há necessidade de variantes de um algoritmo.
  - Quando se quer criar uma classe reusável que coopera com classes não previstas

## Padrões de Design Strategy

---

- Benefícios:**
  - Famílias de algoritmos relacionados
  - Uma alternativa ao uso de subclasses
  - Eliminam comandos condicionais da linguagem de programação
  - A possibilidade de escolha de implementações (diferentes implementações do mesmo comportamento)

## Padrões de Design Strategy

---

- Desvantagens:**
  - O cliente tem de entender a diferença entre as estratégias

## Padrões de Design Strategy

- Exemplo:
  - Interface entre redes com fio e sem fio (diferentes estratégias para cada)
  - Compressão de Arquivos
  - Algoritmos inteligentes para jogos

## Padrões de Design Decorator

*Objeto que adiciona características a outros objetos*

Some applications would benefit from using objects to model every aspect of their functionality, but a naive design approach would be prohibitively expensive.

For example, most document editors modularize their text formatting and editing facilities to some extent. However, they eventually stop short of using objects to represent each character and graphical element in the document. Doing so would promote flexibility at the time based in the application. Text and graphics could be treated uniformly with

## Padrões de Design Decorator

- Problema:** As vezes é necessario acrescentar responsabilidades a objetos individuais e nao a toda uma classe.
- Solução:** Dinamicamente agregar responsabilidades adicionais a um objeto. O Decorator permite extensão de funcionalidades.
- Participantes e Colaboradores:**
  - Component: define a interface para objetos que podem ter responsabilidades acrescentadas aos mesmos dinamicamente
  - ConcreteComponent: define um objeto para o qual responsabilidades adicionais podem ser atribuidas
  - Decorator: mantem uma referencia para um objeto Component e define uma interface que segue a interface de Component (opcionalmente pode executar operações adicionais antes e depois de repassar a solicitação).
  - ConcreteDecorator: acrescenta responsabilidades ao componente.

## Padrões de Design Decorator

```
public class Decorator extends Component {
    public Decorator(Component c){
        setLayout(new BorderLayout());
        add(c,BorderLayout.CENTER);
    }
}
```

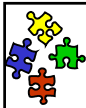
## Padrões de Design Decorator

- Pizza Decorator adiciona ingredientes a Pizza
- Original
  - Subclasses de Pizza
  - Explosão Combinatorial em numero de subclasses
- Usando padrão decorator
  - A classe Pizza decorator adiciona ingredientes a objetos Pizza dinamicamente
  - Pode criar diferentes combinações de ingredientes sem modificar a classe Pizza

## Padrões de Design Decorator

- Exemplo:
  - Adicionar scroll bars e bordas a controles de Interfaces de Usuário
  - JScrollPane é a container com scroll bars a qual pode-se adicionar qualquer componente

```
// JScrollPane decora GUI components
JTextArea area = new JTextArea(20, 30);
JScrollPane scrollPane =
    new JScrollPane(area);
contentPane.add(scrollPane);
```



## Padrões de Design Decorator

---

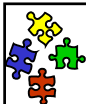
- Strategy X Decorator
  - Duas formas de mudar um objeto: Strategy permite mudar o interior de um objeto. Decorator permite mudar a superfície do objeto.
- Adapter X Decorator
  - Decorator somente muda as responsabilidades de um objeto e não sua interface. O adapter dá uma interface nova.



## Padrões de Design Resumo

---

1. Abstract Factory- Cria uma instancia de várias famílias de classe
2. Builder - Separa construção de objetos da sua representação
3. Factory Method – Cria uma instância de várias classes derivadas
4. Prototype – Uma instância é copiada ou clonada
5. Singleton – Uma classe que tem uma única instância



## Padrões de Design Resumo

---

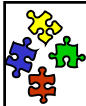
6. Adapter – Compatibiliza interfaces de diferentes classes Match
7. Bridge – Separa a interface de um objeto de sua implementação
8. Composite – Uma estrutura de árvore de objetos simples ou compostos
9. Decorator – Adiciona responsabilidades a objetos dinamicamente
10. Façade – Classe única que representa um sistema inteiro
11. Flyweight – Instancia usada para compartilhamento
12. Proxy – Objecto representando outro objeto



## Padrões de Design Resumo

---

13. Chain of Responsibility – uma forma de passar uma solicitação entre uma cadeia de objetos
14. Command – Encapsula um comando
15. Interpreter – Uma maneira de incluir elementos da linguagem em um programa
16. Iterator – Acesso sequencial a elementos de uma coleção
17. Mediator – Define comunicação simplificada entre classes
18. Memento – Captura e recupera o estado interno de um objeto



## Padrões de Design Resumo

---

19. Observer – Maneira de notificar mudança em várias classes
20. State – Altera o comportamento de um objeto quando seu estado muda.
21. Strategy – Encapsula um algoritmo dentro da classe
22. Template Method – Permite que subclasses definam passos de um algoritmo sem mudar a estrutura do mesmo
23. Visitor – Define uma nova operação para uma classe sem mudar a classe

