Conceitos de Programação Orientada ao Objecto

João Luís Ferreira Sobral
Departamento do Informática
Universidade do Minho

Outubro 2005

Padrões de Desenvolvimento

Soluções a determinados problemas que surgem em aplicações orientadas ao objecto e visam o desenvolvimento de software orientado ao objecto **Reutilizável**

- O desenvolvimento de aplicações baseadas em padrões consiste em escolher um padrão adequado ao problema (ou criar um novo), o que acelera o desenvolvimento e facilita a compreensão de sistemas existentes, pois baseia-se em soluções conhecidas.
- **Projectar para a mudança =>** programar para a interface (*interface*) e não para a implementação (*class*)

Organização dos padrões de acordo com a sua finalidade

- **de criação (cr):** abstraem o processo de criação de objectos
- **estruturais (es):** lidam com a composição de classes ou de objectos
- **de comportamento (cp):** caracterizam a forma como os objectos ou classes interagem e com a distribuição de responsabilidades

Padrões de Desenvolvimento

Catálogo de padrões (cf. Gamma e Al.)

- **Fábrica Abstracta (cr) – encapsular a criação de famílias de objectos**
- **Método de Fabrico (cr) - encapsular num método a criação de objectos**
- **Construtor (cr) – criar um objecto por fases**
- **Protótipo (cr) – criar novos objectos através de cópias dos existentes**
- **Solteiro (cr) – desenvolver uma classe só com uma instância**
- **Adaptador (e) – adaptar uma classe existente a uma interface**
- **Ponte (e) – criar uma separação entre a classe e a sua implementação**
- **Composto (e) – tratar um agregado da mesma forma que um só objecto**
- **Decorador (e) – adicionar características a um objecto durante a execução**
- **Fachada (e) – unificar um conjunto de interfaces num só interface**
- **Peso-leve (e) – suportar a partilha eficiente de objectos**
- **Procurador (e) – criar um representante de um objecto**

Padrões de Desenvolvimento

Catálogo de padrões (*continuação*)

- **Interpretador (cp) – criar um interpretador de uma linguagem**
- **Cadeia de responsabilidades (cp) – delegar o tratamento de um pedido a um objecto de uma cadeia**
- **Comando (cp) – encapsular um pedido num objecto parametrizando os pedidos**
- **Iterador (cp) – Percorrer um agregado de objectos sem expor a sua representação**
- **Mediador (cp) – mediar a comunicação entre vários objectos**
- ***Memento* (cp) – guardar o estado de um objecto para o repor posteriormente**
- **Observador (cp) – implementar um esquema de notificações automáticas**
- **Estado (cp) – tornar o comportamento de um objecto dependente do seu estado**
- **Estratégia (cp) – encapsular um algoritmo para que possa variar de forma independente**
- **Visitante (cp) – representar uma operação numa estrutura**
- **Método padrão (cp) – definir um esqueleto dum algoritmo, refinado por herança**

Padrões de Desenvolvimento

Padrões frequentes na computação paralela

- **Procurador – Utilizado para encapsular o facto de um objecto ser remoto**
- **Fábrica de objectos (Método de fábrica) – Utilizada para implementar a criação remota de objectos**
- **Solteiro – as fábricas de objectos são normalmente instâncias solteiras**

- **Cadeia de responsabilidades – utilizadas nos algoritmos com processamento encadeado de informação**
- **Composto – semelhante ao utilizado nos algoritmos do tipo *master-slave* e *split&merge***
- **Comando – utilizado para enviar tarefas entre objectos**

Padrões de Desenvolvimento

Observador

- **Utilizado para implementar a notificação de eventos do tipo 1 -> n , por forma a que quando o estado de um objecto é alterado os dependentes (observadores) são automaticamente notificados**

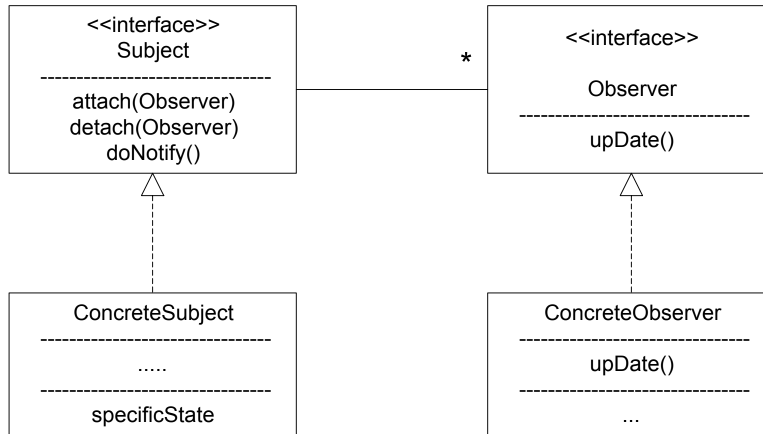
- **Descrição**
  - **Existe um objecto que gera eventos (o *assunto*) e um conjunto de observadores que pretendem ser notificados desses eventos. O *assunto* mantém uma lista dos observadores registados, incluindo métodos para registar e remover observadores**

- **Exemplo (interface da classe)**

```
public interface Subject {
            public void attach(Observer o);
            public void detach(Observer o);
            public void doNotify();
}
public interface Observer {
            public void upDate();
}
```

Padrões de Desenvolvimento

Observador (cont)



- **Opções de implementação**
  - Quem inicia o processo de notificação? O cliente que gerou uma alteração do assunto, ou o próprio assunto? (i.e., invocando um método do observador do tipo *upDate(newState)*)
  - Como é notificada especificidade da alteração? Passando o estado do *Subject* aos observadores ou obrigando os observadores a efectuar uma chamada de retorno para obter o estado (i.e., adicionando um *getState()* ao *Subject*)

Padrões de Desenvolvimento

Iterador

- **Fornece uma forma de aceder aos elementos de um agregado, sem expor a representação interna do agregado**

- **Descrição**

  - **O agregado possui um método para criar um iterador, objecto que regista o elemento actual do agregado, possuindo métodos para obter o elemento actual e verificar se já foram percorridos todos os elementos**

- **Exemplo (interface da classe)**

```
public interface Iterator {
            public Object next();
            public Boolean hasNext();
}
public interface Aggregate {
            public Iterator createIterator();
}
```

Padrões de Desenvolvimento

Cadeia da responsabilidades

- **Evita o acoplamento do emissor de um pedido do seu receptor, dando oportunidade a mais do que um objecto de tratar o evento**

- **Descrição**

  - **Existe uma cadeia de objectos que tratam eventos. Cada evento é passado ao longo da cadeia até ser tratado por um objecto. Cada objecto deve ter um método que trata o evento, ou passa o evento ao elemento seguinte.**

- **Exemplo (interface da classe)**

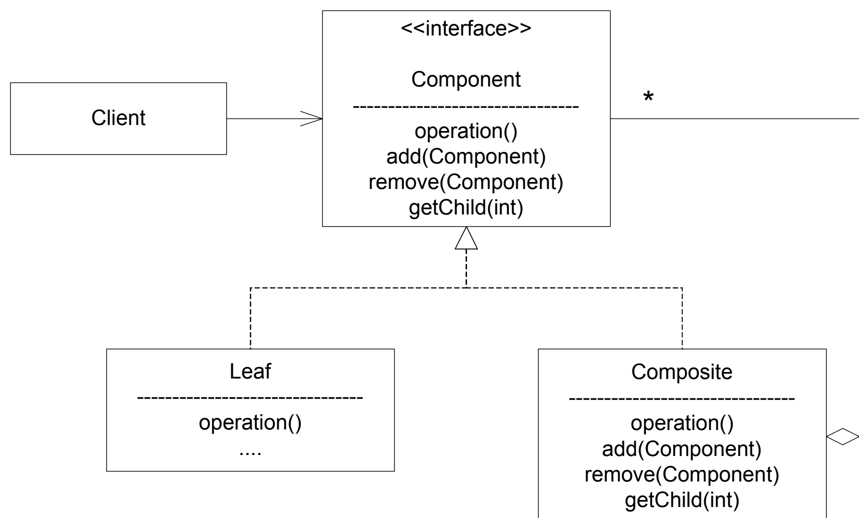
```
public interface Handler {
            public void handleRequest(int);
}
public class TestZero implements Handler {
            private Handler next;
            public Handler(Handler n) { next = n; }
            public handleRequest(int i) {
                        if (i!=0) next.handleRequest(i);    // passa ao seguinte
                        else ...    // trato eu e se next = null???
            }
}
```

Padrões de Desenvolvimento

Composto

- **Utilizado para encapsular um conjunto de objectos de forma transparente para o cliente, por forma a ser possível lidar com um só objecto ou com o conjunto utilizando a mesma interface**

- **Descrição**
  - **O objecto composto implementa a mesma interface que os elementos que o compõem, desta forma, pode ser utilizado nos mesmos contextos. Cada elemento do composto pode ser também um composto, formando uma estrutura em árvore**



Composto (cont.)

- **Exemplo (simplificado)**

```
public class CompositeRunnable implements Runnable {
     protected Vector agenda = new Vector();

      public synchronized void add(Runnable r) {
              agenda.addElement(r);
      }
     public synchronized run() {
           Iterator e = agenda.iterator();
           while (e.hasNext())
           ( (runnable) e.next() ).run();
     }
 }
```

Opções de implementação

- Onde devem ser declaradas as operações para lidar com os elementos de um composto (*add*, *remove*)? No *Component* ou apenas no *Composite*? Transparência *versus* segurança.

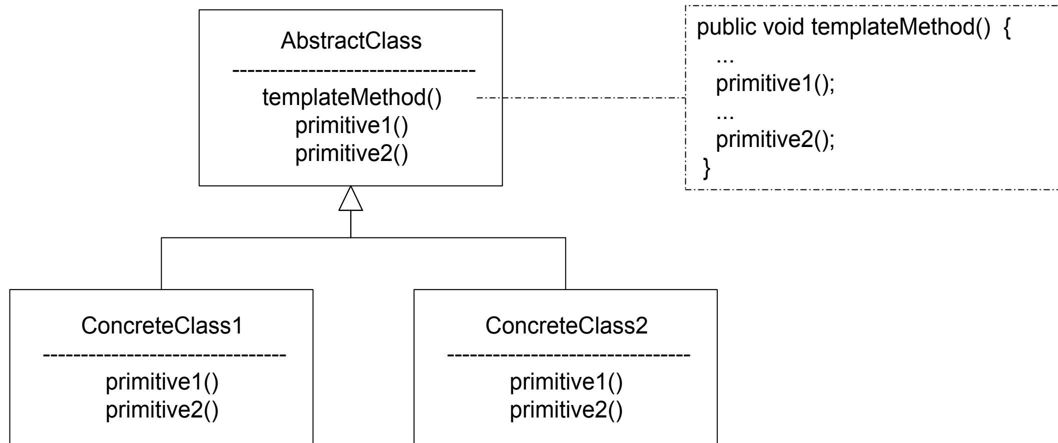
Padrões de Desenvolvimento

Método Padrão

- Utilizado para implementar um esqueleto de um algoritmo, delegando alguns detalhes de implementação para as classes derivadas

- **Descrição**

  - Os detalhes do algoritmo são encapsulados em métodos do objecto, invocados pelo esqueleto, mas não implementados. Estes métodos são implementados nas classes derivadas.

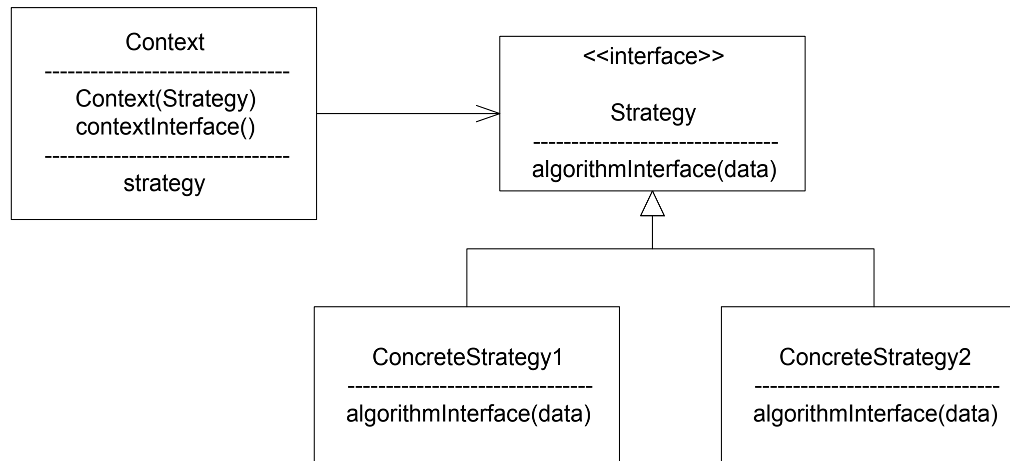


Padrões de Desenvolvimento

Estratégia

- Utilizado para definir uma família de algoritmos, encapsulando cada algoritmo num objecto, tornando-os permutáveis.

- **Descrição**
  - Um objecto (o contexto) utiliza um objecto estratégia para implementar o algoritmo pretendido. O algoritmo pode ser definido pelo cliente



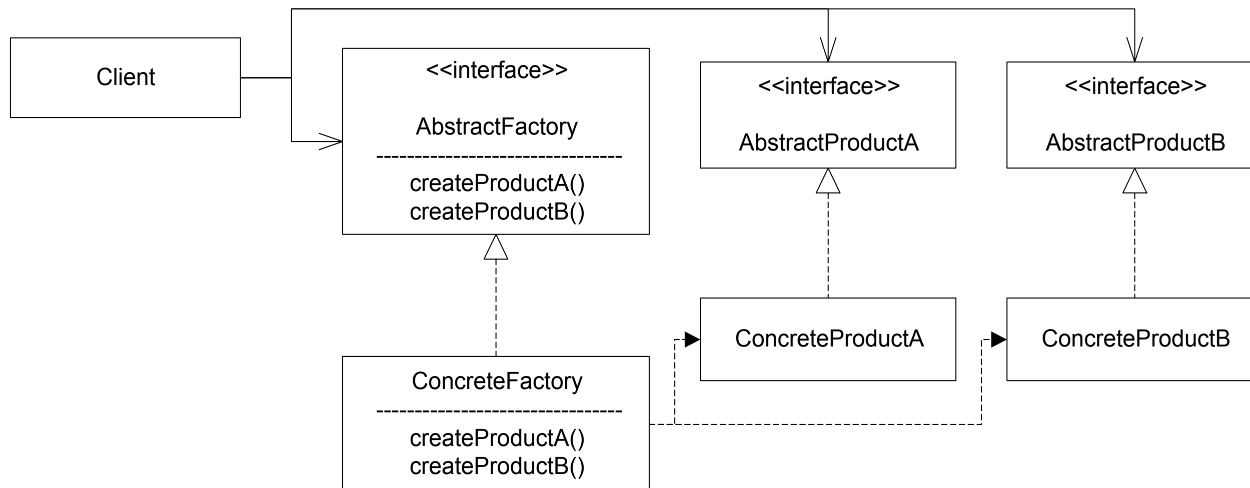
Padrões de Desenvolvimento

Fábrica abstracta

- Fornece uma interface para a criação de famílias de objectos ou objectos dependentes, sem especificar a classe concreta dos objectos

- **Descrição**

  - Normalmente é criada, durante a execução, uma só instância de cada fábrica. Cada fábrica concreta é utilizada para criar uma família de produtos



Fábrica abstracta (cont.)

- **Exemplo (simplificado)**
  - sem fábrica de objectos:

```
objA = new ConcreteProductA();
objB = new ConcreteProductB();
```

  - com fábrica de objectos:

```
AbstractFactory factory = new ConcreteFactory();
// ...
objA = factory.createProductA();
objB = factory.createProductB();
```

Opções de implementação

- As instâncias de fábricas abstractas são normalmente instâncias solteiras
- A forma mais frequente de implementar as fábricas de objectos é utilizando métodos de fabrico (no exemplo anterior *createProductA()* e *createProductB()* são métodos de fabrico)

Padrões de Desenvolvimento

Exercício

- Desenvolver uma versão de um contador, onde seja possível vários objectos serem notificados sempre que o valor do contador é alterado. Pode registar os vários observadores utilizando a classe Vector:

```
import java.util.*;
public class Vector {
        public Vector();
        public void addElement(Object o);
        public void removeElement(Objecto o);
        public Object elementAt(int index);
        public int size();
        ...
}
```

- A classe *Vector*, utilizada na alínea anterior é uma especialização da classe *AbstractList*, herdando o seguinte método, que devolve um iterador nessa lista de elementos:

```
        public Iterator iterator();
```

  Este iterador implementa os seguintes métodos:

```
        public boolean hasNext();
        public Object next();
```

  Implemente novamente o método *doNotify()*, agora utilizando um iterador.