Abstração e encapsulamento

CS-214 – Orientação a Objetos
Prof. Fábio R. de Miranda
Outubro de 2003
(capítulo 6 do Object-Oriented Software Constructon)

Abstração procedural

- Quando organizamos uma seqüência de instruções em um procedimento com um dado nome, digamos *empilha(pilha* p, int i)*, estamos definindo um novo tipo de operação. Ao invocá-la, é necessário conhecer apenas *o que* ela faz, não *como ela faz*.

Abstração procedural

- Em um momento posterior, poderemos retornar ao programa e substituir a implementação deste procedimento por uma versão melhor, o que altera o modo *como* o resultado é atingido, sem alterar *o que* o procedimento computa.

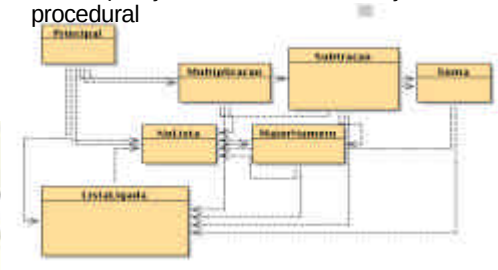
Abstração procedural

Separar o *o que* do *como* é um ato de abstração. Provê dois benefícios:

- *Facilidade de uso – precisamos saber somento o que o procedimento faz.*
- *Facilidade de modificação – Se quisermos mudar como o procedimento é escrito, não precisaremos fazer mudanças ao longo do programa.*

Abstração procedural: Exemplos de alunos

- Decomposição baseada em abstração procedural



Abstração de dados

- A abstração de dados é o mecanismo de abstração mais importante. Ele permite que se abstraiam os detalhes de como os objetos de dados são implementados, nos concentrando em como os objetos devem se comportar.
- A entidade que geramos através deste processo de abstração é o ADT (*abstract data type – tipo abstrato de dados*).

Tipos abstratos de dados (ADT)

- Um tipo abstrato de dados é uma coleção de dados e operações que apresentam propriedades bem definidas aos usuários, escondendo a maneira em que estas operações são implementadas em termos de representações de mais baixo nível.

Exemplo de ADT: Pilha

Operações visíveis:

- Empilhar um elemento na pilha
- Remover um elemento da pilha
- Espiar o topo da pilha
- Checar se a pilha está vazia

Pouco importa como a pilha está implementada (arrays, listas ligadas, anõezinhos com cartazes, etc)

ADT como base formal da POO

- A Programação Orientada a Objetos fornece elementos de modularidade e encapsulamento propícios à implementação de tipos abstratos de dados de forma eficiente.
- Os ADTs fornecem o que há de mais próximo a uma base teórica da POO (e da especificação formal de software)

Modularidade

- Um módulo é uma unidade de organização de um sistema de software que empacota em si uma coleção de entidades (como dados e operações) e que controla cuidadosamente o que usuários externos do módulo podem ver e usar. Ferramentas de modularização da POO:
  - Classes, Interfaces e Pacotes

Encapsulamento (ocultação de informações)

- Um módulo bem projetado distingue-se de um mal projetado pelo grau com que oculta dados internos e detalhes de implementação dos outros módulos.
- Os bons módulos ocultam todos os detalhes de implementação, separando interface (*API*) da implementação.

Encapsulamento

- É importante porque:
- Desacopla os módulos que compõem o sistema, permitindo que sejam desenvolvidos, testados, usados, compreendidos, otimizados e modificados individualmente.
- Acelera o desenvolvimento, pois módulos podem ser desenvolvidos em paralelo.

Encapsulamento

- Aumenta a chance de reuso, porque módulos individuais não dependem de detalhes de implementação uns dos outros, e podem ser utilizados fora do programa para o qual foram desenvolvidos.

Encapsulamento – implementação

- De maneira ideal, cada membro de uma classe ou pacote deve estar tão protegido quanto possível.

Encapsulamento: Interface de um módulo

- Interessamo-nos por um módulo na medida em que seja capaz de nos oferecer operações interessantes, independente de como seja sua implementação interna.

Exercício

- Defina as operações a serem aplicadas a um tipo abstrato de dados que implemente uma fila de banco
- Crie o esqueleto de uma classe em Java que contenha a informação acima

Tipos de métodos

CS-214 – Orientação a Objetos
Prof. Fábio R. de Miranda
Abril de 2003
(ainda capítulo 6 do Object-Oriented Software Constructon)

ADTs e métodos

- Numa abordagem ortodoxa de encapsulamento dentro de orientação a objetos, não é bom expor os elementos internos, de implementação, de um objeto.

ADTs e métodos (cont)

- Para que não seja necessário expor nenhum detalhe de implementação de um objeto, todas as operações em que este objeto precisar estar envolvido devem ser feitas através de métodos.

ADTs e métodos (cont)

- Estes métodos poderão ser de três tipos
- Criadores
- Métodos de acesso
- Mutadores

Métodos

- Criadores– Servem para incializar e criar novos objetos. Idealmente devem entregar o objeto criado tão "pronto para usar" quanto possível, para que o usuário não tenha de ficar configurando o objeto depois

Métodos

- Métodos de acesso – servem para fornecer informações sobre o estado do objeto

Mutadores

- São métodos que alteram o estado atual de um objeto

Exemplo

- Vamos converter a classe *Ponto* dos primeiros exemplos para uma abordagem que favoreça o encapsulamento e classificar os métodos resultantes.

Herança II

Programação Orientada a Objetos
2003
Fábio Roberto de Miranda

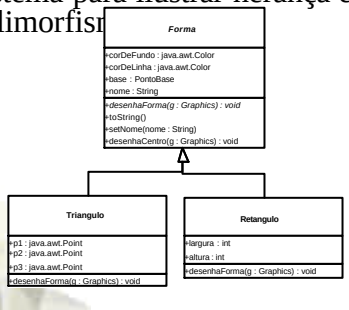
Herança

- Inicialmente havíamos visto a herança como uma prática de reuso
- Outro uso tão importante quanto é combinar uma interface entre módulos, *realizado através da implementação de classes abstratas*

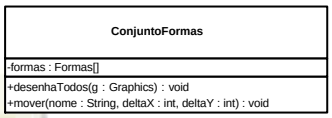
Sistema para ilustrar herança e polimorfismo

- O exemplo modela uma aplicação gráfica simples
- Será preciso criar objetos gráficos de diversos tipos, como retângulos, triângulos, etc.

Sistema para ilustrar herança e polimorfis



ConjuntoFormas é a classe onde se invoca o desenhaForma()



Polimorfismo universal de inclusão (classe ConjuntoFormas)

```
public void desenhaTodos(Graphics g){
    for (int i = 0; i < formas.length; i++) {
        if (formas[i]!=null){
            formas[i].desenhaForma(g);
            formas[i].desenhaCentro(g);
        }
    }
}
```

Observações

- No exemplo anterior, a classe ConjuntoFormas não precisa saber que tipos de *Forma existem, porque todo objeto de uma subclasse de Forma também será uma Forma, implementando o método desenhaForma(Graphics g)*

Classes abstratas ou deferidas

- Classes abstratas são aquelas que definem alguns métodos que não contém implementação. Estas classes não podem ser instanciadas, e as operações precisam ser definidas por suas classes filhas.
- São também chamadas de deferidas porque a implementação é postergada (deferida) para suas classes filhas.

Classes abstratas

- Por suas características, as classes abstratas da orientação a objetos são um mapeamento direto da definição dos tipos abstratos de dados – permitem que se especifique somente a interface de um módulo, sem fornecer implementação.

Classes abstratas x concretas

- As classes abstratas precisam necessariamente incluir a palavra-chave abstract antes do nome da classe e antes dos métodos abstratos
- As classes filhas serão avisadas pelo compilador de que precisam implementar os métodos cuja implementação não havia sido fornecida.

Herança múltipla

- Algumas linguagens orientadas a objeto suportam herança múltipla. Este tipo de característica traz muitas complicações a uma linguagem OO.
- Java adota um modelo mais simples de herança múltipla, através do uso de interfaces. Uma classe pode herdar de apenas uma classe, mas de múltiplas interfaces

INTERFACES

Uma interface é similar a uma classe abstrata, com uma lista de métodos e atributos constantes. Os métodos nunca estão implementados

- os métodos podem ser implementados por qualquer classe, independente da hierarquia.

Interfaces

- uma mesma classe pode implementar métodos de várias interfaces diferentes.
- interfaces podem generalizar outras interfaces, inclusive com herança múltipla.

Declaração de interfaces

```
public interface <nome>
    extends interface1,...,interfaceN{
      <tipo> atributo1 = inicialização1;
      <tipo> atributo2 = inicialização2;
        ...
      <tipo> public metodo1(parâmetros);
        ...
}
```

Os atributos de uma interface fazem papel de constantes para a classe que implementa a interface, e funcionam sempre como se fossem **static** e final.

Utilização de interfaces

```
class <nome>
   implements interface1,...,interfaceN{

    <tipo> public metodo1(parâmetros){
          Implementação...
       }
}
```

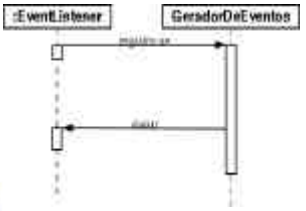
Eventos

- Freqüentemente precisamos, em um programa, descobrir quando ocorre um dado acontecimento, por exemplo: clique de um botão, chegada de pacotes através da rede, fechamento de uma janela, etc...
- Temos duas abordagens para descobrir esta informação – verificá-la de tempos em tempo, checando se mudou, ou ser informado por alguma entidade externa quando isto acontece

Programação baseada em eventos

- A herança a partir de interfaces é bastante comum quando desejamos realizar programação baseada em eventos
- Num programa baseado em eventos, em geral temos um objeto que é o gerador dos eventos, e outros objetos que se registram junto a ele, interessados naquele evento

Esquema de programas baseados em eventos



Classes geradoras de eventos

- As classes que são geradoras de eventos guardam referências para os objetos que precisam ser avisados.
- Estas referências são inicializadas quando os objetos se registram como candidatos a ouvir tais eventos
- Quando acontece o evento, um método combinado entre geradora e interessados é chamado nos interessados

Lista de interessados nos eventos

- *De que tipo deve ser a lista de interessados no evento, mantida pelo objeto gerador?*
- *Object* é inadequado, pois não terá métodos adequados à notificação da ocorrência do evento (a classe geradora não terá como avisá-lo)
- É inviável definir uma subclasse a ser herdada pelos interessados – teriam de abrir mão de serem subclasses de qualquer outra classe

Eventos e interfaces

- Para este tipo de tarefa, costuma-se recorrer a uma interface, que é herdada pelos interessados no evento
- Este é o tipo de herança “leve” para que as interfaces se mostram bastante adequadas
- Vejamos um exemplo prático para entender melhor como isto funciona, implementando interfaces para ouvir eventos em RoboMain.*java* e *ImplementaSensorListener.java*

Exemplo de implementação de interface e tratamento de eventos

```
import josx.platform.rcx.*;

public class ImplementaSensorListener
  implements SensorListener {
     public void stateChanged(Sensor s,
  int oldValue, int newValue){
          Sound.beep();
          LCD.showNumber(newValue);
     }
}
```

Exemplo de implementação de interface e tratamento de eventos

```
import josx.platform.rcx.*;
public class RoboMain
{
    public static void main(String[] args) throws
  InterruptedException {
        Sensor botao1 = Sensor.S1;
  botao1.setTypeAndMode(SensorConstants.SENSOR_TYPE_TOUCH,
  SensorConstants.SENSOR_MODE_BOOL);
        ImplementaSensorListener listener = new
  ImplementaSensorListener();
        botao1.addSensorListener(listener);
        while(true){
            Thread.sleep(20);
        }
    }
}
```