MC302 – Programação Orientada a Objetos 14/06/2016
Instituto de Computação – UNICAMP
Profa.: Cecília Mary Fischer Rubira

Lab14: Reflexão e Padrão de Projeto Observer

Atenção: as questões 1 e 2 (Observer) devem ser entregues até o fim do dia de hoje, e a questão 3 (EXTRA - Reflection) pode ser entregue até o final de quinta-feira. *Fim do dia = 23:59:59*.

Padrão de Projeto Observer:

Considere o seguinte problema: Como acoplar objetos entre si, de forma que estes objetos não se conheçam em tempo de compilação (i.e., não queremos fazer "referênciaObjetoConhecido.método()") e de forma a criar o acoplamento e desfazê-lo a qualquer momento em tempo de execução?

O padrão de projetos ***Observer*** soluciona esta questão fornecendo uma implementação muito flexível de acoplamento de abstrações. O padrão ***Observer*** permite que objetos interessados sejam avisados da mudança de estado ou outros eventos ocorrendo num outro objeto. O objeto sendo observado é normalmente chamado de *Source*, *Subject* ou *Observable*. O objeto que observa é chamado de *Observer* ou *Listener.*

Em Java, um objeto (*Source*) envia informação para outro objeto (*Listener*) pela chamada de um método do *Listener*. Mas, para que isso seja possível, *Source* deve ter uma referência ao *Listener*. O tipo desta referência deve ser uma classe ou interface que declare ou herde o método a chamar. Fazer com que o tipo da referência seja a classe (concreta) do *Listener* não funciona bem porque o número e tipos dos *Listeners* podem não ser conhecidos em tempo de compilação. Os vários *listeners* poderão não fazer parte de uma mesma hierarquia de objetos. Não queremos criar um acoplamento forte entre *Source* e *Listeners*. A solução baseia-se primordialmente em *interfaces* para resolver o problema. Este é um *excelente* exemplo do poder de interfaces para prover polimorfismo envolvendo classes não relacionadas por herança (de implementação).

Exemplo:

Como projetar um sistema que modele um telefone e todos os objetos que poderiam estar interessados quando ele toca? Os objetos interessados poderiam ser pessoas que estejam perto (na mesma sala); uma secretária eletrônica; um FAX ou ainda um dispositivo de escuta clandestina! Os objetos interessados podem mudar dinamicamente, i.e., pessoas entram e saem da sala onde o telefone está; secretárias eletrônicas, FAX, etc., podem ser adicionados ou removidos durante a execução do programa; novos dispositivos poderão ser inventados e adicionados em versões futuras do programa. Qual é a solução básica de projeto? – Faça do telefone um *Event Source* !

Pacote *br.unicamp.ic.mc302.*`telefone`:

1. Abra os arquivos *Pessoa.java*, *SecretariaEletronica.java*, *Telefone.java*, *TelefoneEvent.java*, *TelefoneListener.java*, *TelefoneAdapter.java* e *ExemploFone.java*. Compile e execute a classe Inicial. Estude a aplicação. Observe que o ***source*** fica no arquivo *Telefone.java*. Leia com atenção o código das operações *disparaTelefoneTocou( )* e *disparaTelefoneAtendido* da classe *Telefone*. Examine agora o arquivo *TelefoneEvent.java*. Observe que ***source*** é passado como parâmetro e armazenado no objeto (***super(source)*** faz isso). Isso permite que quem recebe o evento faça *java.util.EventObject.getSource()* para saber qual objeto gerou o evento. Permite que um mesmo objeto seja *Listener* de vários objetos *Source*. Também permite que, com esta referência ao *Source*, o *Listener* acione outros métodos do objeto para obter informação. Este modelo chama-se "***Pull model***" (em outro modelo, "***Push model***", toda a informação necessária está presente dentro do evento). No exemplo, não se está encapsulando dados do

evento, mas seria possível. Como exercicio, inclua o número de telefone que está chamando, a data e a hora.

2. Observe que a *SecretariaEletronica* implementa a interface *TelefoneListener* diretamente, sem usar *TelefoneAdapter*. Por outro lado, o objeto *Pessoa* instancia uma "inner class" anônima que estende *TelefoneAdapter* e faz *override* apenas do método que interessa (*telefoneTocou()*). Quisemos mostrar com isso, formas diferentes de implementar a interface *TelefoneListener*. Implemente uma classe de escuta clandestina que faz uso da interface acima (*TelefoneAdapter*).

Conseqüências do uso do padrão:

- Permite que se variem objetos *Source* e *Listeners* independentemente;
- Pode-se reusar objetos *Source* sem reusar seus *Listeners* e vice-versa;
- Pode-se adicionar *Listeners* sem modificar o *Source* ou os outros *Listeners;*
- O acoplamento entre *Source* e *Listeners* é mínimo;
- Os objetos envolvidos poderiam até pertencer a camadas diferentes de software;
- Suporte para comunicação em broadcast. O *Source* faz broadcast do aviso. Os *Listeners* podem fazer o que quiserem com o aviso, incluindo ignorá-lo;

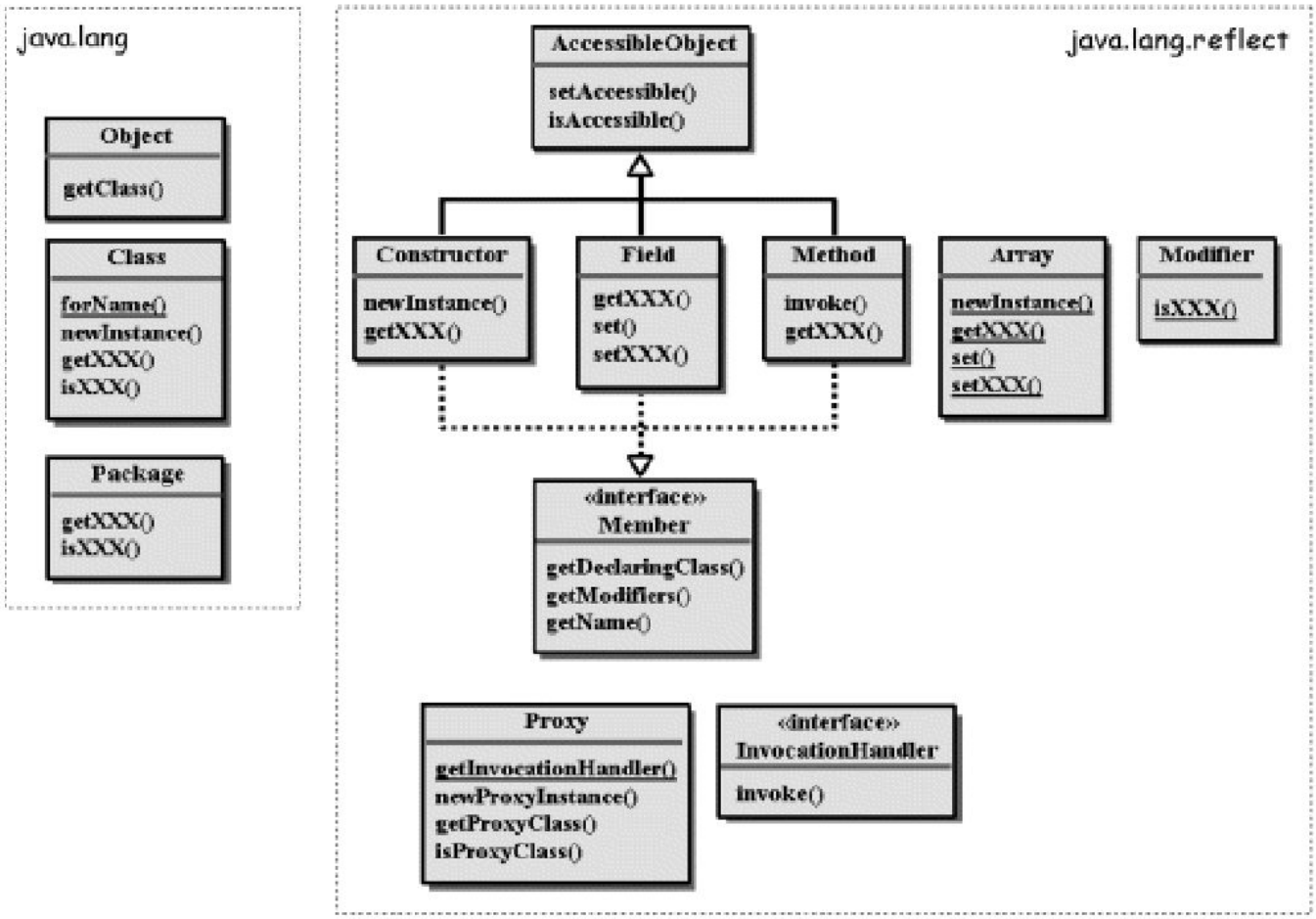
Veja também:

- http://www.javaworld.com/javaworld/javaqa/2001-05/04-qa-0525-observer.html

---

Reflexão:

Reflexão é a capacidade de um programa de investigar fatos sobre si próprio. Por exemplo, um objeto pode "perguntar" a outro quais os métodos que ele possui. A linguagem de programação precisa oferecer recursos para que a reflexão seja possível. A linguagem Java oferece esses recursos através da API ***Java Reflection***:



Exemplos:

- Carregando uma classe
  Class cls = Class.forName("className")
- Determinado a superclasse
  Class superClass = cls.getSuperclass()
- Determinado as interfaces
  Class[] interfaces = cls.getInterfaces()
- Determinado o pacote
  Package pack = cls.get Package()
- Determinado os modificadores
  Modifier.isXXX(member.getModifiers())
  Onde XXX pode ser: public, private, protected, static, final, synchronized, volatile, transient,...
- Determinando campos
  - Determinar todos os campos declarados
    Field[] f1 = cls.getDeclaredFields();
  - Determinar todos os campos públicos
    Field[] f2 = cls.getFields();
  - Determinar um campo especifico
    Field f3 = cls.getField(fieldName)

Field f4 = cls.getDeclaredField(fieldName)

- Determinando métodos
  - Determinar todos os métodos declarados
    Method[] m1 = cls.getDeclaredMethods();
  - Determinar todos os métodos públicos
    Method[] m2 = cls.getMethods();
  - Determinar um método especifico
    Method m3 = cls.getMethod(methodName,params)
    Method m4 = cls.getDeclaredMethod(methodName,params)
- Invocando métodos dinamicamente
  method.invoke(target, params);
- Determinando construtores
  - Determinar todos os construtores declarados
    Constructor[] c1 = cls.getDeclaredConstructors();
  - Determinar todos os construtores públicos
    Constructor [] c2 = cls.getConstructors();

etc.

Pacote `sistemaCaixaAutomatico`:

Você pode utilizar o código disponibilizado no arquivo *codigoCaixa19Abr2016.zip*, ou o seu material entregue no laboratório 8. **Inclua o código no mesmo projeto no eclipse, ao zipar a pasta final com o resultado do laboratório.**

3 (EXTRA). Você deverá construir uma nova classe TrmCxAut (chame-a de TerminalReflexivo) similar à classe existente, com as seguintes opções:

a. Imprimir a hierarquia de classes da classe ***ContaCor***;
b. Imprimir todos atributos das classes de conta;
c. Imprimir todos os métodos públicos da classe ***ControladorCaixa***;
d. Efetuar um saque pelo meta-nível ao invés do nível base.