Observer

- Não vamos seguir a apresentação do livro GoF aqui, pois há críticas sobre a solução dada
  - Falaremos das críticas à frente
- Seguiremos a apresentação dada por Bill Venners em http://www.javaworld.com/topicalindex/jw-ti-techniques.html (The 'event generator' idiom)
- Em particular, apresentaremos como este padrão é implementado em Java
  - Portanto, além de um Design Pattern (que não depende de linguagem), apresentaremos um "Idioma Java" que mostra como implementar um Design Pattern numa linguagem particular

Problema

- Como acoplar objetos entre si:
  - De forma a que não se conheçam em tempo de compilação
    - Não queremos fazer "referênciaAUmObjetoConhecido.método()"
  - De forma a criar o acoplamento e desfazê-lo a qualquer momento em tempo de execução
- Solucionar isso fornece uma implementação *muito flexível* de acoplamento de abstrações

Objetivo

- 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 chamado de:
    - "Subject" (GoF)
    - "Observable" (java.util)
    - "Source" ou "Event Source" (java.swing e java.beans)

- Provedor de informação (Bill Venners)
- Gerador de eventos (Bill Venners)

- O objeto que observa é chamado de
  - Observer (GoF e java.util)
  - Listener (java.swing)
    - Java usa este padrão em 2 lugares mas de formas diferentes!
    - A forma java.util não é boa (ver críticas adiante)
- Usaremos as palavras *Source* e *Listener*

Também chamado de

- Publisher-Subscriber, Event Generator, Dependents

Exemplo

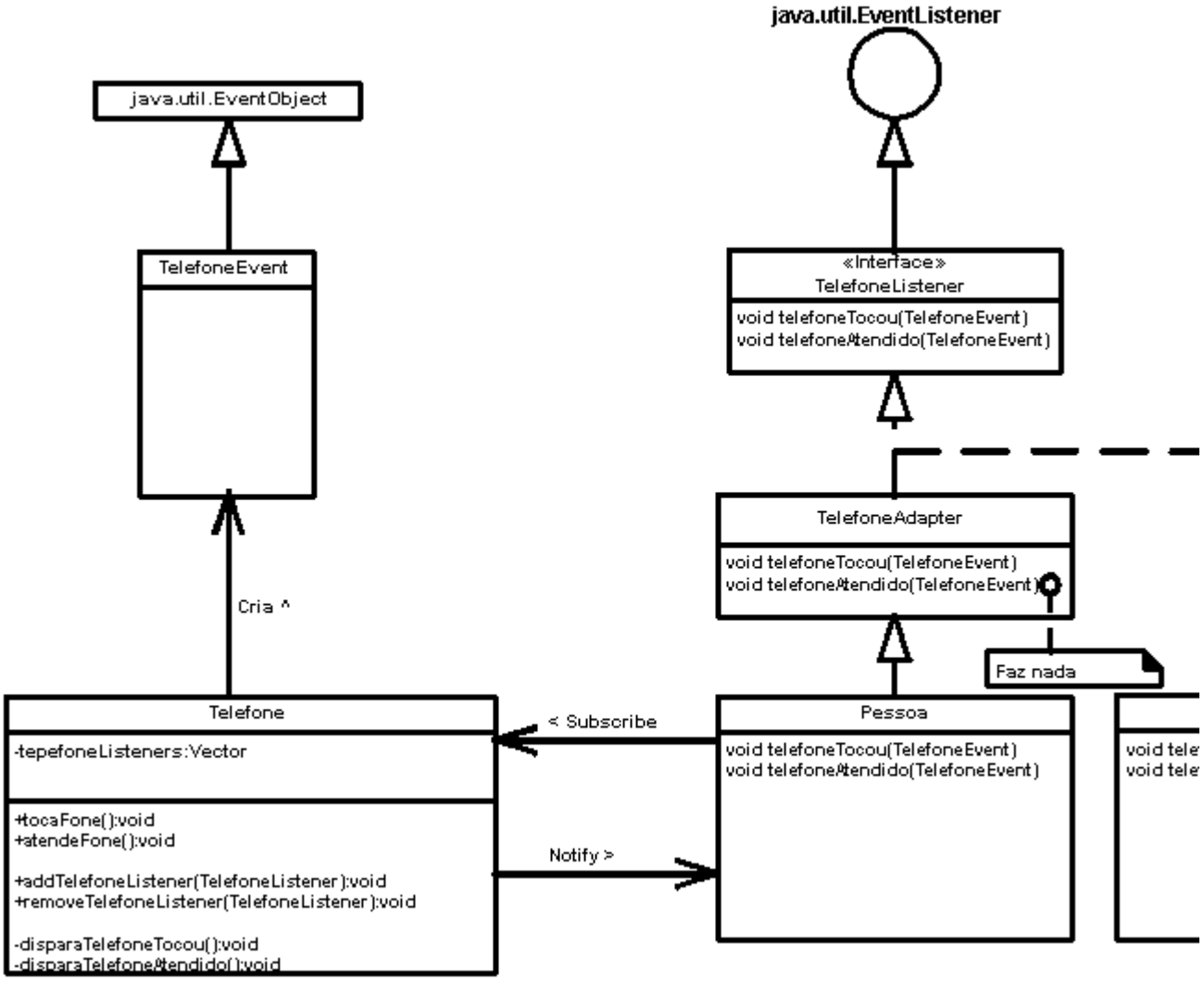
- Como projetar um sistema que modele um telefone e todos os objetos que poderiam estar interessados quando ele toca?
- Os objetos interessados poderiam ser:
  - Pessoas que estejam perto (na mesma sala)
  - Uma secretária eletrônica
  - Um FAX
  - Até um dispositivo de escuta clandestina :-)
- Os objetos interessados podem mudar dinamicamente
  - 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*

O problema

- Em Java, um objeto (o Source) envia informação para outro objeto (o Listener) pela chamada de um método do Listener
- Mas, para que isso seja possível:
  - O 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 não é conhecido 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 vai se basear primordialmente em *interfaces* para resolver o problema
  - Aliás, este é um *excelente* exemplo do poder de interfaces para prover polimorfismo envolvendo classes não relacionadas por herança (de implementação)

A solução idiomática em Java

Estrutura



Etapa 1: Definir classes de categorias de eventos

- Defina uma classe separada de evento para cada categoria principal de eventos que poderão ser gerados pelo Source
  - Nós teremos uma única categoria: "Telefone"
- Faça com que cada classe de eventos estenda java.util.EventObject
- Faça com que cada classe encapsule a informação a ser propagada para os Listeners
- Use nomes de classe que terminem em Event (exemplo: TelefoneEvent)

Etapa 2: Define interfaces de Listener

- Para cada categoria de eventos, defina uma interface

que estenda java.util.EventListener e que contenha um método para cada evento (da categoria) que vai gatilhar a propagação de informação do Source para os Listeners

- Chame a interface como chamou a classe de eventos, com terminação Listener em vez de Event
  - Exemplo: Para a classe TelefoneEvent, a interface de Listener seria TelefoneListener
- Dê nomes aos métodos da interface para descrever a situação que causou o evento. Use um verbo no pretérito
  - Exemplo: método telefoneTocou
- Cada método deve retornar void e aceitar um parâmetro, uma referência a uma instância da classe de eventos apropriada
  - Exemplo de uma assinatura completa:

```
void telefoneTocou(TelefoneEvent e);
```

Etapa 3: Define classes de adaptação (opcional)

- Observação:
  - É muito comum querer uma classe que implemente uma interface fazendo nada para a maioria dos métodos e fazendo algo útil apenas para alguns poucos métodos
  - Para ajudar o programador, é comum, em Java, criar uma classe "adapter" que implemente todos os métodos de uma interface com métodos que nada fazem
    - As classes que devem implementar a interface podem herdar do adapter e fazer override de alguns poucos métodos
    - Isso nada tem a ver com o Design Pattern "Adapter"
- Para cada interface de Listener que contenha mais do que um método, defina uma classe adapter que

implemente a interface por inteiro com métodos que nada fazem
- Dê um nome à classe com terminação Adapter em vez de Listener
  - Exemplo, para a interface TelefoneListener, a classe seria TelefoneAdapter

Etapa 4: Defina a classe observável Source

- Para cada categoria de eventos que serão propagados a partir de instâncias desta classe, define um par de métodos para adicionar/remover Listeners
  - Chame os métodos add<nome-da-interface-listener> e remove<nome-da-interface-listener>
    - Exemplo: addTelefoneListener() e removeTelefoneListener()
- Para cada método em cada interface de Listener, define um método privado de propagação de eventos. O método não aceita parâmetros e retorna void. Este método propaga o evento para os Listeners
  - Chame o método dispara<nome-do-método-listener>
    - Exemplo: disparaTelefoneTocou()
- Chamadas ao método de disparo de eventos devem ser adicionas em lugares apropriados da classe Source
  - Nos lugares onde há mudança de estado ou ocorrência de outros eventos interessantes

Etapa 5: Defina objetos Listener

- Para ser um Listener de uma certa categoria de eventos, basta implementar a interface de Listener da categoria de eventos
  - Isso é normalmente feito estendendo a classe Adapter, mas não é obrigatório

Exemplo de código

- No arquivo TelefoneEvent.java:

```
public class TelefoneEvent
    extends java.util.EventObject {

    public TelefoneEvent(Telefone source) {
        super(source);
    }
}
```

- 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
    - Chama-se este modelo de "Pull model"
    - No "Push model", toda a informação necessária está presente dentro do evento
- Por simplicidade, não se está encapsulando dados no evento aqui mas seria possível incluir:
  - O número de telefone que está chamando
  - A data e as horas
  - etc.
- No arquivo TelefoneListener.java (interface de Listener):

```
public interface TelefoneListener
    extends java.util.EventListener {

    void telefoneTocou(TelefoneEvent e);
    void telefoneAtendido(TelefoneEvent e);
}
```

- No arquivo TelefoneAdapter.java (classe Adapter):

```
public class TelefoneAdapter
    implements TelefoneListener {
```

```
    void telefoneTocou(TelefoneEvent e) {}
    void telefoneAtendido(TelefoneEvent e) {}
}
```

- A definição do source fica no arquivo Telefone.java:

```
import java.util.*;

public class Telefone {
    private Collection telefoneListeners = new Vector();

    // método de suporte para testar a solução
    public void tocaFone() {
        disparaTelefoneTocou();
    }

    // método de suporte para testar a solução
    public void atendeFone() {
        disparaTelefoneAtendido();
    }

    public synchronized void addTelefoneListener(
                TelefoneListener l) {

        if(!telefoneListeners.contains(l)) {
            telefoneListeners.add(l);
        }
    }

    public synchronized void
            removeTelefoneListener(TelefoneListener l) {

        telefoneListeners.remove(l);
    }

    private void disparaTelefoneTocou() {
        Collection tl;
        synchronized (this) {
            // Clonar para evitar problemas de sincronizaçã
            // durante a propagação
            tl = (Collection)(((Vector)telefoneListeners).cl
        }
        TelefoneEvent evento = new TelefoneEvent(this);
        Iterator it = tl.iterator();
        while(it.hasNext()) {
            ((TelefoneListener)(it.next())).telefoneTocou(ev
```

```
        }
    }

    // disparaTelefoneAtendido() é semelhante a disparaTelef
    // Exercício: Que design pattern poderia ser usado para
    // o código comum?
    private void disparaTelefoneAtendido() {
        Collection tl;
        synchronized (this) {
            tl = (Collection)(((Vector)telefoneListeners).cl
        }
        TelefoneEvent evento = new TelefoneEvent(this);
        Iterator it = tl.iterator();
        while(it.hasNext()) {
            ((TelefoneListener)(it.next())).telefoneAtendido
        }
    }
}
```

- Agora, precisamos de classes para *usar* o esquema acima
- Primeiro, os Listeners
- No arquivo SecretariaEletronica.java, temos:

```
public class SecretariaEletronica
    implements TelefoneListener {

    public void telefoneTocou(TelefoneEvent e) {
        System.out.println("Secretaria escuta o telefone toc
    }

    public void telefoneAtendido(TelefoneEvent e) {
        System.out.println("Secretaria sabe que o telefone f
    }
}
```

- No arquivo Pessoa.java

```
public class Pessoa {
    public void escutaTelefone(Telefone t) {
        t.addTelefoneListener(
            new TelefoneAdapter() {
                public void telefoneTocou(TelefoneEvent e) {
                    System.out.println("Eu pego!");
                    ((Telefone)(e.getSource())).atendeFone()
                }
```

```
            }
        );
    }
}
```

- 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 apenas mostrar formas diferentes de implementar a interface TelefoneListener
- Finalmente, precisamos de uma aplicação (arquivo ExemploFone.java)

```
public class ExemploFone {
    public static void main(String[] args) {
        Telefone fone = new Telefone();
        Pessoa fulano = new Pessoa();
        SecretariaEletronica se = new SecretariaEletronica()

        fone.addTelefoneListener(se);
        fulano.escutaTelefone(fone);

        fone.tocaFone(); // começa a brincadeira
    }
}
```

- A execução de ExemploFone imprime:

```
Secretaria escuta o telefone tocando.
Eu pego!
Secretaria sabe que o telefone foi atendido.
```

Quando usar o padrão Observer?

- Quando uma abstração tem dois aspectos, um dependente do outro. Encapsular tais aspectos em objetos separados permite que variem e sejam reusados separadamente
- Quando uma mudança a um objeto requer mudanças

a outros e você não sabe quantos outros objetos devem mudar
- Quando um objeto deve ser capaz de avisar outros sem fazer suposições sobre quem são os objetos. Em outras palavras, sem criar um acoplamento forte entre os objetos

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
  - Basta que os Listeners implementem uma interface simples
  - 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
- Do lado negativo: o custo de uma mudança ao estado de um Source pode ser grande se houver muitos Listeners

Considerações de implementação

- Um Listener pode estar cadastrado junto a vários objetos Source
  - Ele pode descobrir quem o esta notificando se o objeto evento contiver uma referência ao source (como temos no idioma Java)
- Quem dispara o evento original?
  - Se cada método que muda o estado do Source disparar um evento, pode haver eventos demais se

houver mudanças de estado demais
- Neste caso, pode-se deixar um método público do Source que clientes ativam para disparar um evento depois que todas as mudanças ao estado forem feitas
  - O problema é que o cliente pode "esquecer" de chamar este método

- Assegurar a consistência do estado do objeto antes de disparar o evento
  - Particularmente perigoso se um método do Source fizer:

```
super(novoValor); // Pai dispara o evento
var = novoValor;  // atualiza estado do objeto (é tarde!
```

- Diferenças entre modelos Push e Pull
  - O modelo Pull acopla os objetos menos pois o modelo Push supõe que o Source sabe das necessidades de informação dos Listeners
  - O modelo Push pode ser mais eficiente pois o Source pode indicar o que mudou dentro do evento, facilitando a vida para que os listeners saibam o que mudou
    - Usar eventos diferentes para cada situação pode resolver o problema
- Cadastro de interesses
  - Pode-se mudar o protocolo de cadastro (Subscribe) para que o Listener indique as coisas que o interessam

Críticas sobre outras soluções do mesmo padrão

- Gamma apresenta este padrão de uma forma um pouco diferente e menos interessante
  - Esta forma (fraca) do padrão Observer também é usada nas classes Observer/Observable de java.util
- Esses dois exemplos são mais fracos do que o

apresentado aqui (baseado no Java Swing) pelos seguintes motivos:

- Observable é uma classe da qual você *deve* herdar para fazer seu objeto um Source
  - Java não tem herança múltipla e isso queima o (único) cartucho de herança
  - Usar interfaces (herança de tipo) em vez de herança de implementação é melhor
- Para implementar um Observer (equivalente a Listener), você tem que implementar a interface Observer que tem um único método update (Observable, Object)
  - A solução acima é muito melhor pois podemos ter vários eventos, e vários métodos associados, o que torna o código mais claro
    - Você prefere entrar no código do método para descobrir que ele tratar de um telefone que está tocando ou é melhor chamar o método telefoneTocou() como fizemos??
  - A solução acima permite descobrir mais facilmente o que mudou no estado do Source pois podemos usar vários eventos
- De forma geral, o Observer/Observable do Java não é bem visto hoje

Perguntas finais para discussão

- O design clássico Model-View-Controller é explicado na nota de implementação #8: *Encapsulamento de semântica complexa de atualização*. Poderia fazer sentido às vezes um Observer (ou View) falar diretamente com o Subject (ou Model)?
- Quais são as propriedades de um sistema que usa o padrão Observer muito? Como fazer para depurar código em tal sistema?
- Como tratar o problema de concorrência com este padrão? Considere, por exemplo, uma mensagem Unregister() sendo enviada para um objeto antes que este envie uma mensagem Notify() para o Gerente de

Mudança (ou Controlador).

- Examine a tecnologia Infobus do Java e relacione-a com o padrão Observer.

Ver também

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

programa