Java Swing – Tratarea evenimentelor

Arhitectura bazata pe evenimente

EDA – Event-driven Arhitectura – reprezinta un model architectural utilizat in multe aplicatii software datorita flexibilitatii si capacitatii de a reprezenta facil lumea reala. EDA reprezinta un tipar arhitectural bazat pe entitati de tip eveniment si actiuni de tip: generare/creare de evenimente, detectie de evenimente, consumare de evenimente si inceperea unui set de activitati in momentul detectiei unui eveniment.

O aplicatie dezvoltata pe baza acestui model arhitectural va defini si construi urmatoarele entitati:

- evenimente
- surse sau producatori de evenimente – *sources*
- consumatori de evenimente – *sinks*
- protocol de transmitere si interpretare a evenimentelor dinspre sursa spre consummator.

Java Swing API este un framework GUI dezvolvat pe baza acestui tip de arhitectura.

Pentru o viziune mai ampla asupra acestui model arhitectural se poate consulta articolul: Event-Driven Architecture Overview de B. M. Michelson.

Observer Pattern

Din perspectiva design-ului aplicatiilor obiect orientate, pe baza modelului arhitectural MVC – Model View Controller, o paradigma des utilizata este Observer Pattern. (Alte modele similare: Publish-Subscribe, Mediator).

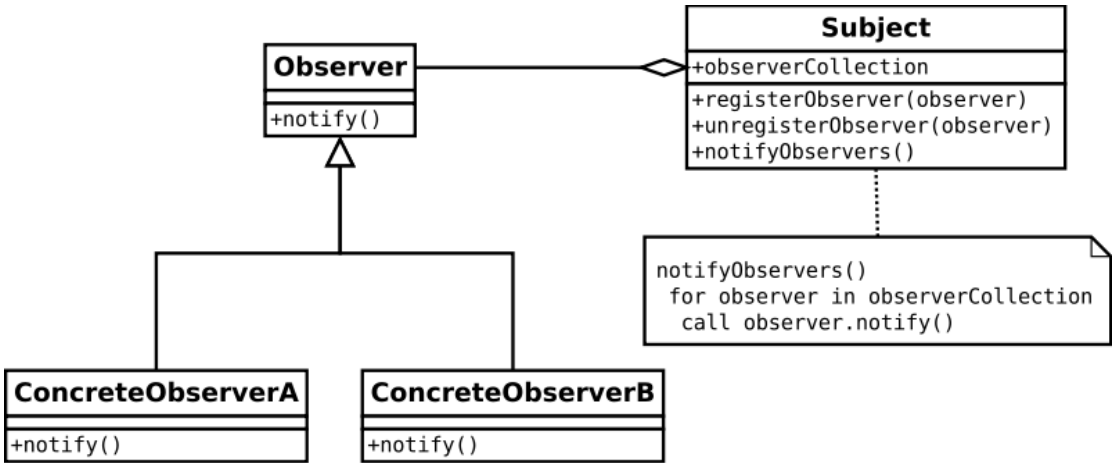


Diagrama UML a claselor pentru Observer Pattern
[http://en.wikipedia.org/wiki/Observer_pattern]

Entitatile de tip Observer reprezinta clasa de baza pentru orice obiect care vrea sa se inscrie ca si observator fata de un anumit Subject.

Subject reprezinta clasa de baza pentru orice entitate care defineste elemente de interes pentru observator.

Subject mentine o lista de entitati Observer si un API (i.e. metode) care permit:

- inregistrarea/eliminarea entitatilor de tip Observer in lista respectiva
- notificarea observatorilor inregistrati la momentul producerii unui eveniment de interes in ciclul de viata al subiectului.

O entitate de tip Observer trebuie sa defineasca metoda notify(). Aceasta este apelata de catre Subject in momentul in care doreste sa anunte o notificare catre observatori. Metoda este suprascrisa in clasele specializate ale clasei de baza Observer pentru a defini reactia potrivita in momentul in care entitatile de tip Subject notifica evenimentele de interes.

O simpla implementare in limbajul Java:

```
import java.util.ArrayList;
import java.util.Calendar;
import java.util.List;


public abstract class Observer {
	public abstract void doNotify();

	public abstract <T> void doNotify(T message);
}

class ConcreteObserver extends Observer {
	String history = "";

	public void doNotify() {
		history +=
				"Notified at: " +

	Calendar.getInstance().getTimeInMillis() + "\n";
	}

	public <T> void doNotify(T message) {
		history +=
				"Notified at: " +
				Calendar.getInstance().getTimeInMillis() + ";
Message received: " + message + "\n";
	}
}

class Subject implements Runnable {
	private List<Observer> observers;

	Subject() {
		observers = new ArrayList<Observer>();
	}
```

```
	public void registerObserver(Observer o) {
		observers.add(o);
	}

	public void unregisterObserver(Observer o) {
		observers.remove(o);
	}

	public void notifyAllObservers() {
		for(Observer o: observers)
			o.doNotify(); // notice the polymorphic call
	}

	@Override
	public void run() {
		// compute and eval something

		try {
			Thread.sleep(5000);
		} catch (InterruptedException e) {}
		/* ...
		 * at this point something interesting happens and we n
			need to let all the observes know about it
		 */
		notifyAllObservers(); // mind you that you could
				      // pass some info back to the
observes by calling doNotify(some_argument);
	}
}
```

Java Swing - event listeners

Pe baza modelului arhitectural EDA si a design pattern-ului Observer Java Swing defineste notiunea de Event Listener. Orice componenta grafica poate genera evenimente atunci cand:

- utilizatorul interactioneaza prin intermediul mouse-ului aflat deasupra componentei
- componenta detine focus-ul iar utilizatorul interactioneaza prin intermediul tastaturii.

Exista si alte tipuri de evenimente care pot fi generate si vor fi discutate in sectiunea urmatoare.

Pentru ca aplicatia sa poata reactiona la evenimentele generate Java Swing permite inregistrarea unor observatori la nivelul componentelor grafice. Acesti observatori sunt denumiti in Java Swing API: *EventListener*. Orice tip specializat de listener/observer extinde clasa EventListener.

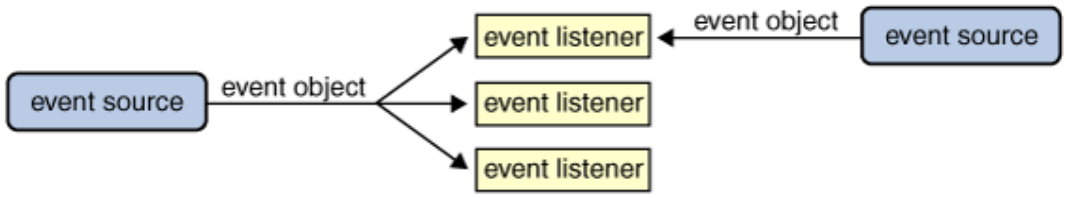
Entitatile de tip Listener ce sunt suportate de componentele grafice definite la nivelul Java Swing API sunt de doua tipuri:

- entitati Listener recunoscute de toate componentele grafice:
    - ComponentListener – observa si reactioneaza la modificarile ce apar la nivelul dimensiunii, pozitiei sau vizibilitatii unei componente

- FocusListener – observa si reactioneata la detinerea/eliberarea focusului pentru o componenta
- KeyListener – observa si reactioneaza la apasarea taselor (doar in cazul in care componenta detine focusul)
- MouseListener – observa si reactioneaza la evenimente de tip mouse (click, miscare, etc)
- MouseMotionListener – reactioneaza la schimbarea pozitiei cursorului mouse-ului deasupra componentei
- MouseWheelListener – reactioneaza la miscarea rotitei mouse-ului deasupra componentei
- HierarchyListener – reactioneaza si observa la evenimente din ierarhia de includere fata de alte componente grafice (i.e. fiecare componenta grafica are un context grafic inregistrat ca si parinte)
- HierarchyBoundsListener – similar HierarchyListener dar relativ la evenimente de mutare si redimensionare.

- entitati Listener recunoscute doar de anumite componente grafice (e.g. ActionListener).

Model bazat pe evenimente



Modelul bazat pe evenimente implementat pe baza EDA si Observer Pattern in Java Swing induce mecanisme puternice si flexibile de dezvoltare a aplicatiilor GUI.

Astfel orice numar de entitati event listener pot sa asculte/observe orice tip de eveniment de la oricate entitati de tip sursa de evenimente (i.e. orice aritate de asoricere intre entitatile sursa si listener se poate implementa; e.g. un event listener poate primi evenimente de la multiple surse, o multime de event listeners pot primi evenimente de la o singura sursa, etc).

Spre deosebire de tiparul clasic al Observer Pattern in cazul entitatilor event listener din Java Swing informatie este mereu transmisa intre sursa si listener. Informatia este incapsulata intr-un obiect de tip Event si contine informatie relevanta despre evenimentul produs s despre sursa care la emis.

Pentru majoritatea evenimentelor ce pot fi regasite la nivelul unei aplicatii GUI exista o serie de event listeners si exemple/pattern-uri de manipulare si

reactionare la aceste evenimente. In acest sens se pot consulat sectiunile de tip how-to de la Sun.

Dezvoltarea unui event listener

Un event listener trebuie sa se execute rapid deoarece toate apelurile catre medotele de redare grafica si evcent-listening sunt executate la nivelul aceluiasi thread. In cazul in care un anumit evenimen determina executia unei sectiuni indelungate atunci este indicata executia ecesteia intr-un nou thread. Consultati aceasta sectiune pentru detalii privind pornirea si executia thread-urilor.

Obiecte de tip Event

Informatia primita – sub forma de argument – la nivelul metodelor unui event listener este impachetata intr-un obiect ce este de tipul sau extinde `EventObject`. (e.g. clase care extind EventObject: KeyEvent, MouseEvent, AWTEvent, InputEvent, etc).

EventObject defineste metoda getSource() ce returneaza un Object care incapsuleaza sursa care a generat respectivul eveniment.

In cazul evenimentelor specializate se pot obtine informatii suplimentare prin API-ul asociat (e.g. KeyEvent defineste getKeyCode() care returneaza un intreg asociat cu key-ul acestui eveniment; apoi se poate invoca metoda getKeyText(int keyCode) cu argument intregul returnat de getKeyCode() si astfel se obtine o reprezentare String a tastei apasate – e.g. ‘HOME’).

Evenimente de nivel primar si evenimente semantice

Un eveniment poate fi:

- de nivel primar – low level event: acele evenimente ce sunt lansate la nivelul sistemului de ferestre grafice sau device-uri de tip input (e.g. mouse, tastatura).
- semantic: acele evenimente generate fie ca urmare a unui low-level event sau ca urmare a unei alte actiuni sau activitati sesizate in fluxul de executie al aplicatiei. (e.g. un ActionEvent lansat in urma unui click peste o comoponenta JButton).

Este incurajata dezvoltarea folosind evenimente semantice in favoarea celor low-level.

Clase adaptor – Event Adapters

Majoritatea entitatilor de tip Listener sunt implementate in Java Swing API sub forma de interfete ce trebuie implementate in clase concrete la nivelul aplicatiei.

Unele interfete de tip listener definesc o multiudine de metode, de exemplu MouseListener defineste:

- `mouseClicked(MouseEvent e)`
- `mouseEntered(MouseEvent e)`
- `mouseExited(MouseEvent e)`
- `mousePressed(MouseEvent e)`

- `mouseReleased(MouseEvent e)`

Implementarea tuturor acestor metode intr-o clasa concreta care implementeaza interfata MouseListener, clasa care trateaza doar cazul de mouse clicked implica:

- implementarea unor metode empty-body (i.e. {})
- reducerea lizibilitatii codului si a mentenatei.

Din acest motiv Java Swing API defineste pentru fiecare interfata de tip Listener – ce defineste mai mult de o metoda – o clasa adaptor. In cazul MouseListener exista clasa MouseAdapater care implementeaza toate metodele interfetei in format empty-body. In tabelul Listener API Table sunt identificate toate interfetele Listener care prezinta o clasa adaptor.

Pentru a utiliza clasele adaptor trebuie definita o clasa proprie care extine clasa adaptor corespunzatoare.

Exemplu:

Pentru o clasa care nu ar folosi MouseAdapter si implementeaza MouseListener codul trebuie sa contina minim urmatoarele linii:

```
public class SomeMouseListenerImpl implements MouseListener {
    ...
        someObject.addMouseListener(this);
    ...
    /* Empty-body method definition. */
    public void mousePressed(MouseEvent e) {}

    /* Empty method definition. */
    public void mouseReleased(MouseEvent e) {}

    /* Empty method definition. */
    public void mouseEntered(MouseEvent e) {}

    /* Empty method definition. */
    public void mouseExited(MouseEvent e) {}

    public void mouseClicked(MouseEvent e) {
        ...//Event listener implementation goes here...
    }
}
```

Prin utilizarea clasei MouseAdapter codul de mai sus se transforma in:

```
public class SomeMouseListenerImpl extends MouseAdapter {
    ...
        someObject.addMouseListener(this);
    ...
    public void mouseClicked(MouseEvent e) {
        ...//Event listener implementation goes here...
    }
}
```

Implementarea si extinderea conceputului de Event si Listener

In exemplul de mai jos se creeaza un nou tip de Listener pentru a reactiona la un caz special/eveniment generat in executia aplicatiei (i.e. un eveniment semantic).

```
package ag.main.frame;

import java.awt.BorderLayout;
import java.awt.Dimension;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;

public class MainJFrame extends JFrame {
	public static void main(String[] args) {

		// 1. init the frame container
		MainJFrame frame = new MainJFrame();

		frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

		frame.setSize(new Dimension(300,300));

		// 2. generate an ActionEvent object

		final ActionEvent ae = new ActionEvent(frame, 100,
"myEvent");


		// 3. create a JButton and register MyListener to it

		final JButton myButton = new JButton("my button");

		myButton.addActionListener(new MyListener());

		// 4. add the button to the frame and show the GUI
		frame.getContentPane().add(myButton);

		frame.setVisible(true);

		// simulate the emission/generation of the ActionEvent
		Thread t = new Thread(new Runnable() {

			@Override
			public void run() {
				try {
					Thread.sleep(5000);
				} catch (InterruptedException e) {
					e.printStackTrace();
				}
					for(ActionListener a:
myButton.getActionListeners())
					a.actionPerformed(ae);
			}

		});
```

```
			t.start();
		}
}

class MyListener implements ActionListener {

		@Override
		public void actionPerformed(ActionEvent arg0) {
			// TODO Auto-generated method stub
			System.out.println(arg0.getActionCommand());

		}
}
```

In aceasi maniera se pot crea evenimente noi (i.e. extinderea unei clase care reprezinta un event). De asemenea se pot creaza noi metode de notificare la nivelul unui Listener (i.e. in loc de actionPerformed(..) se poate adauga o noua metoda – e.g. *newCustomActionPerformed*(..) – ce va fi cunoscuta doar de aplicatia care utilizeaza acest nou API specific.

Exercitiu:
Implementati un nou tip de eveniment: SpecialActionEvent care extinde ActionEvent. Implementati o noua metoda specialActionPerformed(SpecialActionEvent…) in clasa MyListener. Signatura metodei sa fie identica cu cea de la actionPerformed(…).
Simulati la nivelul main urmatorul scenariu:

- actionPerformed(…) actionata de ActionEvent(…,"myEvent") lanseaza un thread
- thread-ul genereaza dupa 5 secunde un SpecialActionEvent
- la generarea SpecialActionEvent se apeleaza specialActionPerformed().