Patrones de Diseño

...

75.10 - Técnicas de Diseño

Aplicación

Estamos trabajando en una aplicación de trading. La misma, entre otras cosas, maneja la notificación de los cambios de precios en las acciones. Queremos poder realizar distintas operaciones cada vez que se produce un cambio en el precio de las acciones. Queremos que sea fácil poder agregar nuevas operaciones a realizar antes las actualizaciones de precios. Cómo podemos diseñar nuestra aplicación para cumplir con este requerimiento?

1- Primera aproximación

```
class Application {

    public static void main(final String[] args) {
        StockPriceManager manager = new StockPriceManager(0);

        manager.setPrice(10);
        manager.setPrice(5);
        manager.setPrice(15);
        manager.setPrice(7);
    }

}
```

1- Primera aproximación

```
class StockPriceManager {
	private int currentPrice;

	public StockPriceHandler(final int initialPrice) {
		this.currentPrice = initialPrice;
	}

	public void getPrice() {
		return currentPrice;
	}

	public void setPrice(final int newPrice) {
		if (newPrice > currentPrice) {
			System.out.println("Price went up!");
		} else if (newPrice < currentPrice) {
			System.out.println("Price went down!");
		}
	}
}
```

2- Crear Abstracciones

```
class abstract StockPriceHandler {
	private int previousPrice;
	private StockPriceManager manager;

	public StockPriceHandler(final StockPriceManager manager) {
		this.manager = manager;
		updatePreviousPrice();
	}

	public void handle() {
		doHandle();
		updatePreviousPrice();
	}

	private void updatePreviousPrice() {
		this.previousPrice = manager.getPrice();
	}

	protected void doHandle();
}
```

2- Crear Abstracciones

```
public IncreaseStockPriceHandler extends StockPriceHandler {

    public IncreaseStockPriceHandler(final StockPriceManager manager) {
        super(manager);
    }

    public void doHandle() {
        if (manager.getPrice() > this.previousPrice) {
            System.out.println("Price went up!");
        }
    }
}
```

2- Crear Abstracciones

```
public DecreaseStockPriceHandler extends StockPriceHandler {

    public DecreaseStockPriceHandler(final StockPriceManager manager) {
        super(manager);
    }

    public void doHandle() {
        if (manager.getPrice() < this.previousPrice) {
            System.out.println("Price went down!");
        }
    }
}
```

3- Cómo cambió el price manager?

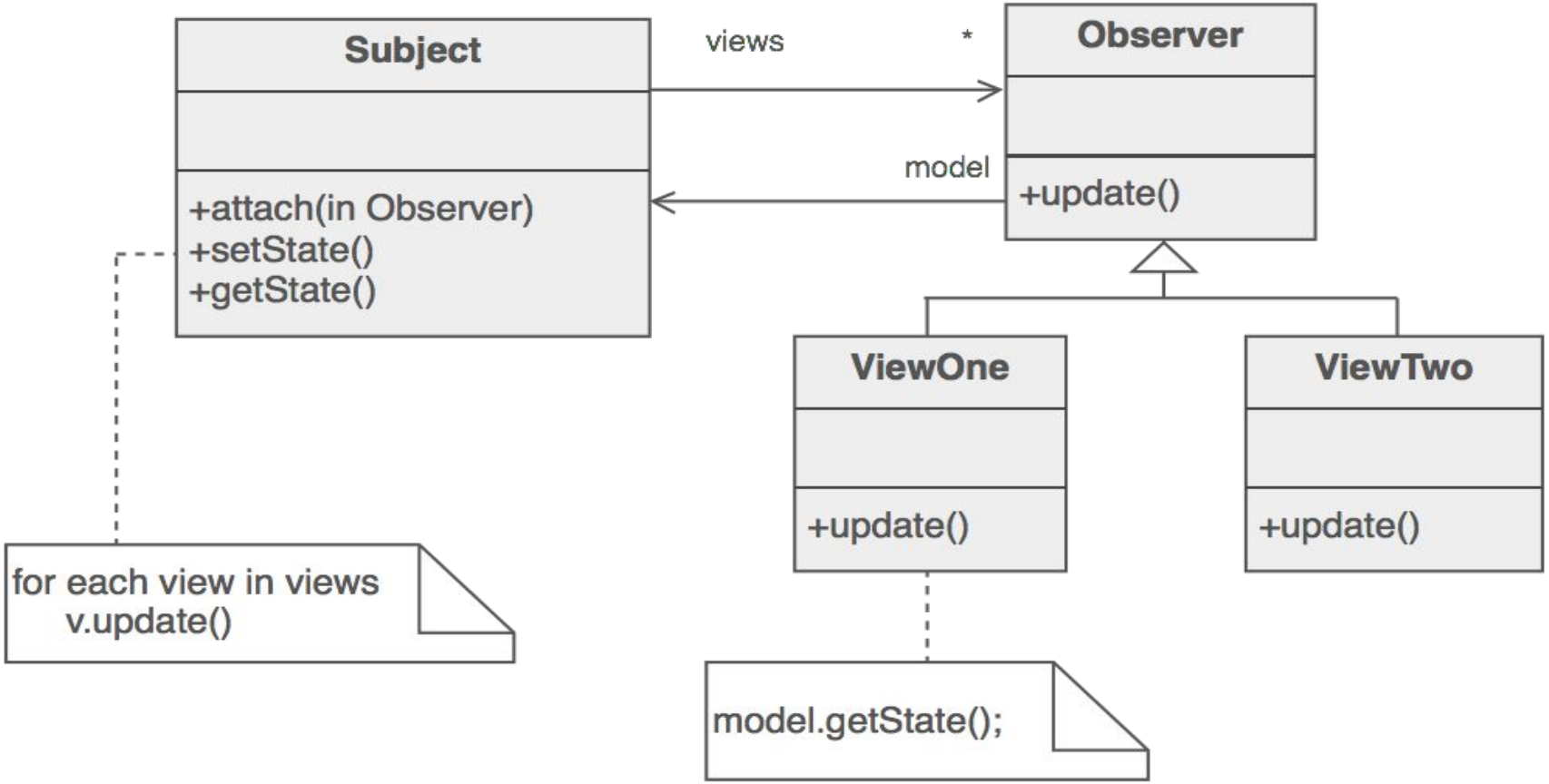
```
class StockPriceManager {
	private int currentPrice;
	private Collection<StockPriceHandler> handlers = new
ArrayList<>();

	public StockPriceHandler(final int initialPrice) {
		this.currentPrice = initialPrice;
	}

	public void getPrice() {
		return currentPrice;
	}

	public void registerHandler(final StockPriceHandler handler) {
		this.handlers.add(handler);
	}
```

```
	private void notifyHandlers() {
		for (StockPriceHandler handler : handlers)
			handler.handle();
		}
	}

	public void setPrice(final int newPrice) {
		this.currentPrice = newPrice;
		this.notifyHandlers();
	}
}
```

4- Cómo cambió mi aplicación?

```
class Application {

    public static void main(final String args[]) {
        StockPriceManager manager = new StockPriceManager(0);

        DecreaseStockPriceHandler decreaseHandler = new DecreaseStockPriceHandler(manager);
        IncreaseStockPriceHandler increaseHandler = new IncreaseStockPriceHandler(manager);

        manager.registerHandler(decreaseHandler);
        manager.registerHandler(increaseHandler);

        manager.setPrice(10);
        manager.setPrice(5);
        manager.setPrice(15);
        manager.setPrice(7);
    }

}
```

Observer

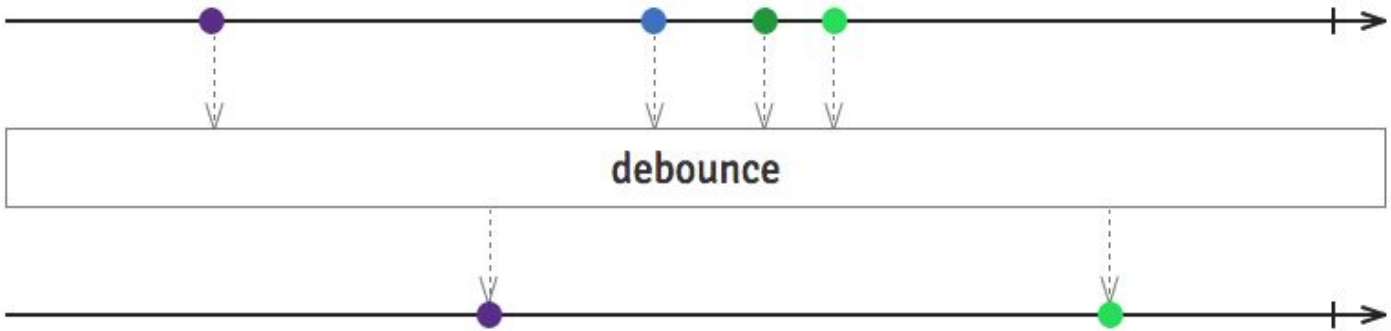

Subject
views
*
Observer
model
+attach(in Observer)
+setState()
+getState()
+update()
ViewOne
ViewTwo
+update()
+update()
for each view in views
v.update()
model.getState();

Reactive Extensions

ReactiveX Introduction Docs Languages Resources Community

The Observer pattern done right

ReactiveX is a combination of the best ideas from
the Observer pattern, the Iterator pattern, and functional programming



?