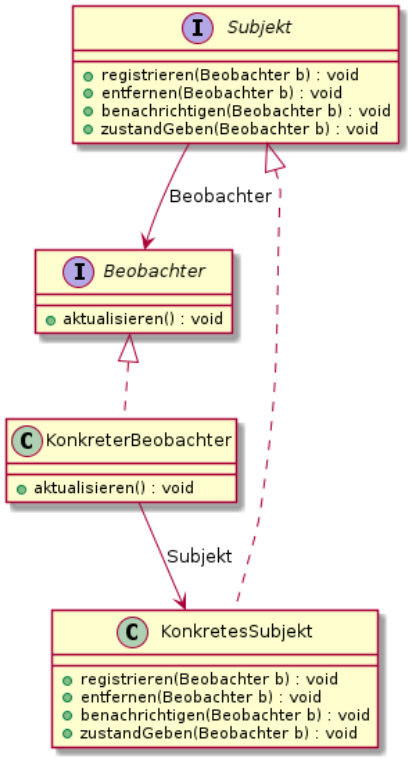
Merkblatt - Observer/Observable Pattern

Kurzbeschreibung/Zweck

Observer/Observable ist ein Verhaltens-Entwurfsmuster. Es spezifiziert die Kommunikation zwischen Objekten: Beobachter und Subjekte. Ein Subjekt ist ein Objekt, das Beobachter über Änderungen seines Zustands benachrichtigt.

UML



Problem

Das Observer/Observable Pattern befasst sich mit den folgenden Problemen:

- Eine 1:n-Abhängigkeit zwischen Objekten sollte definiert werden, ohne die Objekte eng miteinander zu koppeln.
- Es sollte sichergestellt werden, dass bei einer Zustandsänderung eines Objekts eine offene Anzahl von abhängigen Objekten automatisch aktualisiert wird.
- Es sollte möglich sein, dass ein Objekt eine unbefristete Anzahl anderer Objekte melden kann.

Die Definition einer 1:n-Abhängigkeit zwischen Objekten durch die Definition eines Objekts (Subjekts), das den Zustand abhängiger Objekte direkt aktualisiert, ist unflexibel, da es das Subjekt an bestimmte abhängige Objekte koppelt. Dennoch kann es aus Sicht der Leistung oder wenn die Objektimplementierung eng gekoppelt ist, sinnvoll sein. Eng gekoppelte Objekte können in manchen Szenarien schwer zu implementieren und schwer wiederzuverwenden sein, weil sie sich auf viele verschiedene Objekte mit unterschiedlichen Schnittstellen beziehen und darüber Bescheid wissen.

Lösung

Definieren Sie Subjekt- und Beobachterobjekte, so dass bei einer Zustandsänderung eines Subjekts alle registrierten Beobachter automatisch benachrichtigt und aktualisiert werden. Die alleinige Verantwortung eines Subjekts besteht darin, eine Liste von Beobachtern zu führen und sie über Zustandsänderungen durch Aufruf ihrer update()-Operation zu benachrichtigen. Dadurch sind Subjekt und Beobachter lose gekoppelt. Subjekt und Beobachter haben keine explizite Kenntnis voneinander. Beobachter können zur Laufzeit unabhängig voneinander hinzugefügt und entfernt werden.

Beispiel in Java

//-----Subjekt-----

```
public interface Subjekt {

	public void registrieren(Beobachter b);
	public void entfernen(Beobachter b);
	public void benachrichtigen(String s);
	public void zustandGeben(Beobachter b);
}
```

//-----Beobachter-----

```
public interface Beobachter {
	public void aktualisieren(String s);
	public String getTest();
}
```

//-----KonkretesSubjekt-----

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

public class KonkretesSubjekt implements Subjekt {

	public KonkretesSubjekt(String test) {
		this.test = test;
		List<Beobachter> list = new ArrayList<>();
		System.out.println("success");
	}

	private String test;
	private List<Beobachter> list = new ArrayList<>();

	public void setTest(String test) {
		this.test = test;
		benachrichtigen(this.test);
	}

	public String getTest() {
		return test;
	}

	@Override
	public void registrieren(Beobachter b) {
		this.list.add(b);

	}
```

```
	@Override
	public void entfernen(Beobachter b) {
		this.list.remove(b);

	}

	@Override
	public void benachrichtigen(String test) {
		for(Beobachter b : list) {
			b.aktualisieren(test);
		}

	}

	@Override
	public void zustandGeben(Beobachter b) {
		System.out.println(b.getTest());
	}

}
```

//----- KonkreterBeobachter -----

```
public class KonkreterBeobachter implements Beobachter {
	private String test;

	@Override
	public void aktualisieren(String s) {
		this.test = s;

	}

	public String getTest() {
		return test;
	}

}
```