Zuständigkeitskette

aus der Reihe:
Objektbasierte Verhaltensmuster

Stefan Belitz, Hannes Mühlenberg, Martin Oppelt

(Chain of Responsibility)

„Vermeide die Kopplung des Auslösers einer Anfrage mit seinem Empfänger, indem mehr als ein Objekt die Möglichkeit erhält, die Aufgabe zu erledigen. Verkette die empfangenden Objekte und leite die Anfrage an der Kette entlang, bis ein Objekt sie erledigt.“

Funktionsweise

- organisiere mögliche Empfänger einer Anfrage hinsichtlich ihrer Allgemeinheit als eine Kette von Objekten
- jeder Empfänger kennt nur seinen Nachfolger
- Auslöser und Bearbeiter sind einander nicht bekannt
- der Bearbeiter einer Anfrage wird zur Laufzeit bestimmt (impliziter Empfänger)

Teilnehmer

Bearbeiter:

- definiert eine Schnittstelle zur Bearbeitung einer Anfrage
- implementiert eine Verbindung zum Nachfolgeobjekt

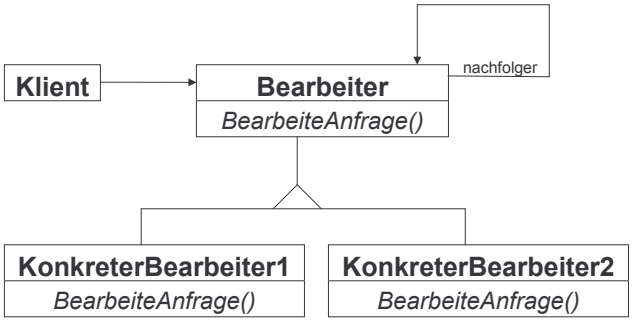
konkreter Bearbeiter:

- bearbeitet eine Anfrage oder leitet sie weiter, wenn er nicht für sie zuständig ist

Klient:

- stößt mit einer Anfrage die Kette von Bearbeiterobjekten an

Struktur



Konsequenzen

- reduzierte Kopplung befreit ein Objekt davon, zu wissen welches Objekt eine Anfrage beantworten wird; Objektbeziehungen werden vereinfacht
- einfache Abänderung der Kette (Struktur, neue Bearbeiterobjekte), da flexibel und dynamisch
- keine Abarbeitungsgarantie, da es keinen expliziten Empfänger gibt

Aspekte der Implementierung

Implementierung der Nachfolgerkette:

- Verwendung existierender Verbindungen (Kompositionsmuster)
- Definition neuer Verbindungen

Aspekte der Implementierung

Verbindung von Nachfolgeobjekten:

- bei Nichtexistenz einer Referenz zur Definition der Kette verwaltet der Bearbeiter die Nachfolgeobjekte
- der Bearbeiter stellt eine Defaultimplementierung von BearbeiteAnfrage bereit, welches die Anfrage weiterleitet

Bsp.:

```
class Hilfebearbeiter{
public:   Hilfebearbeiter ( HilfeBearbeiter* nf ) : _nachfolger ( nf ) { }
virtual void BearbeiteHilfsAnfrage ( );
private:  HilfeBearbeiter* _nachfolger;
};

void Hilfebearbeiter::BearbeiteHilfsAnfrage ( ) {
          if (_nachfolger) {
          _nachfolger->BearbeiteHilfsAnfrage ( );
          }
}
```

Aspekte der Implementierung

Repräsentation von Anfragen:

- fest als Operationsaufruf codiert
  -> festgelegte Anzahl von Anfragen
- eine einzige Bearbeitungsoperation mit Parameterübergabe
  -> unbegrenzte Menge von Anfragen
- anlegen einer Anfrage-Klasse zur expliziten Repräsentation jeder Anfrage, Unterklassen für unterschiedliche Parameter

Aspekte der Implementierung

Bsp.:

```
void Bearbeiter::BearbeiteAnfrage ( Anfrage* dieAnfrage ) {
        switch ( dieAnfrage->GibTyp ( ) )
        case Hilfe:
        BearbeiteHilfsAnfrage ( ( HilfsAnfrage* ) dieAnfrage );
        break;

        case Drucken:
        BearbeiteDruckAnfrage ( ( DruckAnfrage* ) dieAnfrage );
        break;

        default:
        .... break;
        }
}
```

Aspekte der Implementierung

Automatische Weiterleitung in Smalltalk

- verwendung des doesNotUnderstand-Mechanismus aus Smalltalk
- Anfragen mit dem ein Objekt nichts anfangen kann werden im dNU-Mechanismus abgefangen
- er muß also nur überschrieben werden, so das er die Anfrage an das FolgeObjekt weitergibt

Implementierung

Beispiel in Java:

- Arbeiter stellt an seine Vorgesetzten die Anfrage nach Geld
- Vorgesetzte sind hierarchisch in einer Kette organisiert

Code:

```
public abstract class Geldhahn { // Bearbeiter

    Geldhahn next;

    void gibGeld() { // default-Implementierung der Beabeitungsmethode
          if (next!=null) next.gibGeld();
    }
}
-------------------------------------------------------------------------------------
public class Abteilungsleiter extends Geldhahn { // konkreter Bearbeiter 1

    Abteilungsleiter(Geldhahn next) {       // im Konstruktor wird der jeweilige
    Nachfolger
          this.next = next;           // übergeben
    }                                 // Bearbeitungsmethode wird nicht überschrieben
}

public class Filialleiter extends Geldhahn { // konkreter Bearbeiter 2

    Filialleiter(Geldhahn next) {
          this.next = next;
    }

    void gibGeld() { // Bearbeitungsmethode wird überschrieben,
          System.out.println("Hier haste ne Maak"); // Anfrage ist abgearbeitet
    }
}
```

Code:

```
public class Obermotz extends Geldhahn { // konkreter Bearbeiter 3

    Obermotz(Geldhahn next) {
          this.next = next;
    }

    void gibGeld() { // überschreibt ebenfalls die Bearbeitungsmethode, die Anfrage
    // wurde aber schon vom Filialleiter bearbeitet, und wird daher nicht aufgerufen
          System.out.println("Hier haste zehntausend Maak");
    }
}
-------------------------------------------------------------------------------------
public class Arbeiter { // Klient, der die Anfrage stellt

    public static void main(String[] args) {
          // erst mal wird jedes Kettenglied mit Nachfolger erzeugt
          Geldhahn Obermotz = new Obermotz(null);
          Geldhahn Filialleiter = new Filialleiter(Obermotz);
          Geldhahn Abteilungsleiter = new Abteilungsleiter(Filialleiter);

          Abteilungsleiter.gibGeld(); // stellt die Anfrage an das erste Kettenglied
    }
}
```

Verwendung

- mehr als ein Objekt sollen eine Anfrage beantworten können; welches das tut, ist nicht von vorn herein bekannt
- eine Anfrage soll an eines von mehreren Objekten gerichtet werden, ohne den Empfänger explizit anzugeben
- die Menge der Objekte, die eine Anfrage bearbeiten sollen, wird dynamisch festgelegt

Fragen zur Zuständigkeitskette an

Martin Oppelt: moppelt@freenet.de; ICQ-101816998