Objektorientiertes Design

Verantwortung, Rolleneinhaltung und Zusammenarbeit! Aber welche Klasse oder welches Objekt sollte welche Verantwortung haben?

Wichtige Punkte, die als Eingabe für das objektorientierte Design verwendet werden können / können:

- Erstellung eines Domänenmodell mit Analyse und Konzeption
- Beschreibungen von Anwendungsfällen in Form von User Storys, die sich im aktuellen iterativen Schritt in der Entwicklung befinden

Darauf aufbauend müssen Sie Interaktionsdiagramme für den Systembetrieb der vorliegenden Anwendungsfälle erstellen, indem Sie Richtlinien und Grundsätze für die Zuweisung von Verantwortlichkeiten anwenden.

Während der Systemverhaltensanalyse wie an unserem Beispiel des POS-Systems, werden Systemoperationen einer konzeptionellen Klasse beispielsweise dem System zugewiesen. Dies Bedeutet nicht zwangsläufig, dass im Design ein Klassensystem enthalten sein muss. Eine Controller-Klasse wird zugewiesen, um die Systemoperationen auszuführen.

Wer ist für den Systembetrieb verantwortlich? Welches Objekt außerhalb der UI-Schicht empfängt und koordiniert die Systemoperationen?

- Erstellen Sie für jede Systemoperation im Entwicklungszyklus ein separates Diagramm
- Verwenden Sie die Systemoperation, als Startmeldung
- Wenn ein Diagramm zu komplex wird, teilen Sie es in kleinere Diagramme auf:
    - teilen Sie die Zuständigkeiten zwischen den Klassen möglicherweise während der Objektgestaltung oder beim Hinzugefügen
    - oder auf der Beschreibung des Verhaltens basierend auf

Verantwortung

Robert C. Martin (Uncle Bob):

> Jede Verantwortung ist eine Achse der Veränderung. Wenn sich die Anforderungen ändern, manifestiert sich diese Änderung durch Änderungen der Verantwortung zwischen den einzelnen Klassen. Wenn eine Klasse mehrere Zuständigkeiten hat, gibt es mehrere Gründe dafür diese zu ändern.

Die Zuordnung von Verantwortung von Klassen ist eine der wichtigsten Aktivitäten während des Designs. Pattern, Idiome, Prinzipien helfen bei der Zuordnung der Verantwortung. In Verantwortungsvollem Design (RDD Responsibility-driven Design) denken wir, dass Software-Objekte Verantwortlichkeiten haben. Diese Verantwortung werden während der Objektgestaltung dem Objektklassen zugewiesen. Doch wie bestimmt man die Zuordnung von Verantwortlichkeiten zu verschiedenen Objekten?

In der Verantwortungszuweisung gibt es eine große Unterschiede, es gibt „gute“ und „schlechte“, „schöne“ und „hässliche“, „effiziente“ und „ineffiziente“ Designs, doch wie macht man diese aus und

wie effizient ist ein „schöner“ Code? Schlechte Entscheidungen führen zu instabilen Systemen, welche schwer zu warten und zu verstehen sind oder deren Erweiterung und auch Wiederzuverwendung einer neuen Softwareentwicklung gleicht.

Kopplung oder Coupling

Die Kopplung misst die Abhängigkeit von Klassen und Paketen. Die Klasse A ist an die Klasse B gekoppelt, wenn A direkt oder indirekt B erfordert. Oder eine Klasse, die von zwei anderen Klassen abhängig ist, hat eine niedrigere Kopplung als eine Klasse, die von 8 anderen Klassen abhängig ist.

Typ X hat ein Attribut, das sich auf eine Instanz des Typs Y oder den Typ Y selbst bezieht

```
class X{ private Y y = …}
class X{ private Object o = new Y(); }
```

Ein Objekt vom Typ X ruft Methoden eines Typs Y-Objekt auf

```
class Y{f(){;}}
class X{ X(){new Y.f();}}
```

Typ X verfügt über eine Methode, die auf eine Instanz des Typs Y verweist, beispielsweise mittels Parameter, lokaler Variable oder Rückgabetyp

```
class Y{}
class X{ X(y Y){…}}
class X{ Y f(){…}}
class X{ void f(){Object y = new Y();}}
```

Typ X ist ein Subtyp vom Typ Y

```
class Y{}
class X extends Y{}
```

Kopplung in Java

```
package it.haberland.testpackage;

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

public class QuitAction implements ActionListener {
  @Override
  public void actionPerformed(ActionEvent e)
  {
    System.exit(0)
  }
```

```
}
```

Die Klasse QuitAction ist an folgendes gekoppelt:

- ActionListener
- ActionEvent
- java.lang.Override
- java.lang.System
- java.lang.Object

Hohe Kopplung

Eine Klasse mit hoher Kopplung ist unerwünscht, da:

- Änderungen in verwandten Klassen lokale Änderungen erzwingen
- in isolierter Form schwieriger zu verstehen sind
- Die Wiederverwendung schwer ist, da für die Verwendung mehrere abhängig Klassen erforderlich sind.

Niedrige Kopplung

Niedrige Kopplung unterstützen das objektorientierte Design durch unabhängige und wiederverwendbare Klassen.

- Generische Klassen mit hoher Wahrscheinlichkeit für eine Wiederverwendung sollten eine besonders geringe Kopplung haben.
- ABER, gar keine Kopplung ist auch nicht wünschenswert, da bei der Objektorientierung ein System mit den verbundenen Objekten über Nachrichten kommuniziert.
- zu geringe Kopplung führt zu übermäßigen Auslastung bei aktiven Objekten und bremsen dadurch das gesamte System aus.

Hohe Kopplung stabilisiert Elemente und stellt damit selten ein Problem dar. Jedoch kann das zwanghafte Streben nach einer niedrigeren Kopplung zu Gunsten der Wiederverwendbarkeit in zukünftigen Projekten unnötig komplex werden und damit zu hohen Projektkosten führen. Wie oft verwenden Sie Code in neuen Projekten wieder?

Verschiedene Kopplungsarten

Die Kopplung an „stabile Klassen“, welche beispielsweise durch das JDK stattfinden sind normalerweise kein Problem, da diese:

- sich nicht ändern
- die Wiederverwendung nicht Verhindern

Eine Kopplung an Schnittstellen ist besser als eine Kopplung der Klassen

- denn dadurch wird die Testbarkeit verbessert
- die Wiederverwendung wird erleichtert

- Kopplung an Schnittstellene sind (oft) stabiler

Cohesion oder Kohäsion

Kohäsion misst die Stärke der Beziehung zwischen Elementen einer Klasse. Alle Operationen und Daten innerhalb einer Klasse sollten „natürlich“ zu dem Konzept gehören, das die Klasse modelliert.

Kohäsion in Java

```
public class SimpleLinkedList {
  private final Object value;
  private final SimpleLinkedList next;
  public SimpleLinkedList(Object value, SimpleLinkedList next) {
    this.value = value; this.next = next;
  }

  public Object head() {
    return value;
  }

  public SimpleLinkedList tail() {
    return next;
  }
}
```

Analyse der Kohäsion der SimpleLinkedList:

- Der Konstruktor verwendet beide Felder value und next
- head verwendet nur das Feld value
- tail benutzt nur next
- head und tail sind einfache Getter Methoden
- head und tail mutieren nicht

```
import java.awt.Color;

abstract class ColorableFigure implements Figure {
  private Color lineColor = Color.BLACK;
  private Color fillColor = Color.BLACK;

  public Color getLineColor() {
    return lineColor;
  }

  public void setLineColor(Color c) {
    lineColor = c;
  }
```

```
  public Color getFillColor() {
    return fillColor;
  }

  public void setFillColor(Color c) {
    this.fillColor = c;
  }
}
```

Analyse der Kohäsion von ColorableFigure:

- lineColor wird nur vom Getter und Setter verwendet
- fillColor wird nur vom Getter und Setter verwendet
- lineColor und fillColor haben keine Abhängigkeiten

Arten der Kohäsion

- Zufällige: Keine sinnvolle Beziehung zwischen Elementen einer Klasse.
- Logische: durch funktionale Kohäsion, Elemente einer Klasse führen eine Art logische Funktion aus. Zum Beispiel eine Schnittstelle mit POST-Hardware.
- Temporär: Alle Elemente einer Klasse werden „zusammen“ ausgeführt.

Um die Designkomplexität überschaubar zu halten, weisen Sie die Zuständigkeiten unter Beibehaltung der hohen Kohäsion zu.

Geringe Kohäsion

Klassen mit niedriger Kohäsion sind unerwünscht, da diese

- schwer zu verstehen sind
- schwer wiederverwendbar sind
- und schwer zu warten da leicht von Veränderungen betroffen sind

Klassen mit hoher Kohäsion können oft durch einen einfachen Satz beschrieben werden. Klassen mit niedriger Kohäsion stellen oft eine sehr weiträumige Abstraktion dar und haben Verantwortung übernommen, die an andere Objekte hätte delegiert werden sollen.

Messung der Kohäsion

Der Mangel an Kohäsion in Methoden (LCOM) misst die Ungleichheit zwischen Methoden in einer Klasse.

- Ein hoher Wert zeigt „schlechte“ Kohäsion an
- Ein niedriger Wert zeigt „hohe“ Kohäsivität an

Definition von LCOM = Anzahl unzusammenhängender Sets lokaler Methoden. Jedes Set hat eine oder mehrere lokale Methoden der Klasse und zwei beliebige Methoden im Set greifen auf mindestens ein

gemeinsames Attribut der Klasse zu. Die Anzahl der gemeinsamen Attribute im Bereich von 0 bis N (wobei N eine positive ganze Zahl ist).

Im Allgemeinen berücksichtigen Ich keine Vererbung und Konstruktoren!

Beispiel am Code:

```
public class Rectangle extends Figure {
  private int id;
  private String name;
  private Color color;

  public Color getColor() { return color; }
  public void setColor(Color color) { this.color = color; }
  public int getID() { return id; }
  public int getName() { return name; }
  public String toString() {
   return "Rectangle(" + id + ":"+name+")";
  }
}
```

- 3 Attribute (id, name, color)
- Zugriff auf das color-Attribut mit getColor()
- Zugriff auf das id-Attribut mit getID()
- Zugriff auf das name-Attribut mit getName()
- Zugriff auf das name-Attribut und id-Attribut mit toString()
- Anzahl disjunkten Sets: 2

Würde man nun den oberen Abschnitt ändern in:

```
public Rectangle(int id, String name, Color color) {
 this.id = id;
 this.name = name;
 this.color = color;
 }
```

Wäre eine Trennung unmöglich, da die Anzahl der getrennten Sets unbedingt, mindestens 1 sein muss. Daher sollten Sie es generell vermeiden, Konstruktoren zu berücksichtigen.

Würde man den unteren Code-Abschnitt ändern in:

```
public String toString() {
 return "Rectangle(" + getID() + “:”+getName()+")";
 }
```

Folglich erhöht sich die Anzahl der disjunkten Sets auf 4. Daher erfordert die LCOM-Metriken immer ein detailliertes Verständnis.

Design braucht Prinzipien

Eine häufige Falle im objektorientierten Design sind Vererbungen und deren Beziehungen.

Nehmen wir an, wir wollen unsere Bibliothek für Vektorgrafikanwendungen erweitern, und unsere Bibliothek definiert bereits Klassen für Kreise und Quadrate. Nehmen wir an, wir wollen unsere Bibliothek weiterentwickeln und für Rechtecke erweitern Soll dann das Rechteck vom Quadrat erben oder sollte Quadrat von Rechteck erben? Wir prüfen nach: „Ist ein Rechteck ein Quadrat? – „NEIN!“ also muss folglich das Quadrat von Rechteck erben.

Zusammenfassung

Zusammengefasst möchten Sie folgendes erreichen:

- niedrige Kopplung
- hohe Kohäsion
- grundsätzlich nur Einzelverantwortung (Single Responsibility)
- keine Dopplung, auch wenn diese geringfügig abweicht
- keine zyklische Abhängigkeit
- Liskov-Substitutionsprinzip
- Open-Closed Prinzip, (Erweiterbarkeit und Wartbarkeit)

Und ganz Wichtig: Eine Klasse sollte nur einen Grund haben geändert zu werden.

Orthogonalität

Laut Andrew Hunt and David Thomas, The Pragmatic Programmer, Addison-Wesley, 2000:

> Zwei oder mehre Dinge sind orthogonal, wenn Änderungen an einem die anderen Elemente nicht verändern. z.B. Wenn sich eine Änderung des Codes zur Verknüpfung zu einer Datenbank keine Auswirkungen auf Ihren GUI hat, dann wird dies als orthogonal bezeichnet.

Design Heuristiken oder Design Heuristics

Designheuristiken helfen, ein Design zu bewerten, be es gut, schlecht oder irgendwo dazwischen ist. Objektorientierte Designheuristiken bieten Einblick in objektorientierte Designverbesserungen. Dabei sind die Richtlinien sprachunabhängig und ermöglichen dadurch die Beurteilung der Integrität eines Softwaredesigns. Heuristiken sind jedoch keine harten Regeln. Sondern müssen viel mehr als Warnmechanismen betrachtet werden, die es ermöglichen, die Heuristik nach Bedarf oder Projekthintergund zu ignorieren. Viele Heuristiken sind auch nur kleine Änderungen am Design und damit nur lokal.

Ein einzelner Verstoß gegen eine Heuristik verursacht selten große Auswirkungen auf die gesamte Anwendung.

Es gibt zwei Bereiche, in denen das objektorientierte Paradigma Design Ihr Projekt in eine gefährliche Richtungen lenken kann:

- schlecht verteilte Systemintelligenz oder auch The God Class Problem, bedeutet, dass Sie die Systemintelligenz so einheitlich wie möglich verteilen, genauer, die Klassen der obersten Ebene in Ihrem Entwurf sollten die Aufgaben einheitlich verteilen. Vermeiden Sie Klassen, in deren public interfaces viele accessor Methoden definiert sind. Das führt dazu, dass Daten und Verhalten nicht an einem Ort bleiben. Vermeiden Sie ausserdem Klassen, die zu wenig kommunizierendes Verhalten zeigen, d.H. Methoden, die mit einer richtigen Teilmenge der Datenelemente einer Klasse arbeiten. God Classes zeichnen sich oft durch diese nicht kommunizierendes Verhalten aus.
- zu viele Klassen für die Größe des Designproblems, wodurch Ihr Projekt an Transparenz verliert und unübersichtlich wird.

Alle Daten in einer Basisklasse müssen privat sein. Versuchen Sie nicht privaten Daten zu vermeiden, Definieren Sie stattdessen (geschützte) Zugriffsmethoden. Wenn Sie gegen diese Heuristik verstoßen, ist Ihr Design tendenziell gefährdet.

Beispiel

Ein Beispeil an Hand einer Punk-Klasse:

punkt
– x : int [private int xKoordinate] – y : int [private int yKoordinate]
punkt() + getX() [public int getX()] + setX(int) [public void setX(int)] + getY() [public int getY()] + setY(int) [public void setY(int)]

Die Klasse Punkt verfügt über Zugriffsoperationen über öffentliche Schnittstellen. Könnte es ein Problem mit diesem Design geben? Nein, da Accessor-Methoden nicht notwendigerweise Implementierungsdetails offen legen.

Accessor-Methoden weisen auf eine unzureichende Kapselung von Daten und Verhalten hin. Beispielsweise ruft man die x- und y-Werte des Punkt-Objektes nur ab um damit etwas zu berechnen. Dieses Verhalten, bezieht sich dabei auf die Punkte.

Notwendigkeit von Accessor-Methoden

- Eine Klasse mit Getter- und Setter-Methoden, implementiert eine Verfahrensweise.
- UI-Schicht muss in der Lage sein, die Daten zur Visualisierung zu erhalten.

Implementieren von Richtlinien / Verfahrensweisen zwischen zwei oder mehr Klassen Beispiel aus der Kursplanungsdomäne einer Universität. Dabei betrachten wir die folgenden Klassen:

Die Klasse Student erfasst statische Informationen über den Studenten, z.B. Name, Matrikelnummer,

eine Liste der von ihm belegten Kurse, usw.

Die Klasse Kurs erfasst ebenso statische Informationen der Kursobjekten, z.B. Kursnummer, Beschreibung, Dauer, Mindest- und Höchstzahl der Teilnehmer, Liste der Voraussetzungen usw.

Die Klasse Vorlesungsverzeichnis erfasst statische und dynamische Informationen, die sich auf einen bestimmten Abschnitt eines bestimmten Kurses beziehen, z. B. den angebotenen Kurs, den Raum und den Zeitplan, den Kursleiter, eine Teilnehmerliste usw.

Möchte nun eine Person eine Vorlesung besuchen, dann Sieht das Design volgendes Vorgehen vor:

- überprüfe ob die Person ein Student ist (z.B. ob er gültig eingeschrieben ist)
- überprüfe ob der Student die Voraussetzungen für diesen Kurs erfüllt (z.B. ob der Student die Grundlagen schon besitzt für diesen Kurs)
- überprüfe ob der Kurs die Voraussetzungen für diesen Studenten erfüllt (z.B. ob noch ein Platz frei ist)

Versuchen Sie im Allgemeinen immer, die reale Welt zu modellieren. Geringe Repräsentationslücke erleichtert dabei die Wartung und die Entwicklung ungemein. Achten Sie bei der Modellierung der realen Welt auf Heuristiken. Zum Beispiel zeigt ein Raum in der realen Welt kein Verhalten, jedoch ist für die Implementierung eines Heizsystem das Aufheizen oder das Runterkühlen des Raumes zu beachten und über Methoden oder Klassen zu implementieren. Grundsätzlich ist eine god class eine Klasse, die zu viel tut, also auf das Verhalten bezogen.

Durch systematische Anwendung der untersuchten Prinzipien wird die Impelemntierung von god classes weniger wahrscheinlich.

Stellen Sie sicher, dass die von Ihnen modellierten Abstraktionen Klassen sind und nicht nur Rollen, die Objekte spielen. Stellen Sie vor dem Erstellen neuer Klassen sicher, dass das Verhalten wirklich anders ist und dass es keine Situation gibt, in der die Rolle eine Teilmenge der eine andere Funktionalität verwendet.

Zusammenfassung

Weisen Sie den Klassen Verantwortlichkeiten so zu, dass die Kopplung so gering wie möglich, die Kohäsion ist so hoch wie möglich und die Repräsentationslücke so gering wie möglich ist.

Kopplung und Kohäsion sind Bewertungskriterien, anhand derer Sie das OO-Design beurteilen können.

Designheuristiken sind keine harten Regeln, sondern helfen Ihnen, Schwachstellen in Ihrem Code zu erkennen, um potenzielle (zukünftige) Probleme zu erkennen.