Objektorientierte Programmierung

11 - Entwurf

Alexander Stuckenholz

Version 2022-05-04

Inhalt

HOCHSCHULE HAMM-LIPPSTADT

Analyse

HOCHSCHULE HAMM-LIPPSTADT

In den letzten Abschnitten haben wir uns mit der **Analysephase** der Softwareentwicklung befasst.

- Das Ziel dieser Phase ist es, ein erstes **Informationsmodell** einer Anwendung zu erstellen.
- Zudem wird meist auch die elementare **Verarbeitungslogik** festgelegt.

Beide Facetten (Struktur und Dynamik) können mithilfe der UML modelliert werden.

- Das Informationsmodell wird dabei meist als Klassendiagramm erstellt.
- Für die Verarbeitungslogik werden z.B. Aktivitäts- oder Sequenzdiagramme genutzt.

In der Analysephase geht es hauptsächlich um die **Fachlichkeit** des Systems.

- Die entstehenden Modelle sollen die Frage beantworten, was genau das System leisten soll und welche Strukturen dazu nötig sind.
- Technische Details zur Umsetzung (ob als Web-Anwendung, oder als App) werden dabei zunächst (soweit es geht) ausgeblendet.

Entwurf

HOCHSCHULE HAMM-LIPPSTADT

In der **Entwurfsphase** werden die Ergebnisse der Analyse aufgegriffen und erweitert.

- Der Entwurf befasst sich nun mit allen technischen Details, z.B.:
- um welche Art von Anwendungssystem es sich handeln soll (Web, App, Desktop, ...),
- wie die Benutzeroberfläche gestaltet wird,
- wie und wo Daten gespeichert werden,
- wie Benutzer authentifiziert und autorisiert werden, ...

Durch die Festlegung all dieser Details entsteht letztendlich die **Softwarearchitektur**.

- Meistens wird der Begriff der Softwarearchitektur so definiert, dass es sich dabei um die fundamentale Struktur/Organisation des Softwaresystems handelt.
- Martin Fowler, ein anerkannter Experte, bezeichnet Softwarearchitektur eher als die Summe aller wichtigen und schwer änderbaren Entscheidungen[1].

[1] https://martinfowler.com/architecture/

Qualität des Entwurfs

HOCHSCHULE HAMM-LIPPSTADT

In dieser Veranstaltung soll es nicht primär um spezielle Technologien gehen.

- Wir wollen uns hier keine speziellen Frameworks ansehen, die z.B. bei der Entwicklung von Web-Anwendungen oder Apps für mobile Endgeräte wichtig sind.

Wir wollen aber lernen, wie man zu einem gelungenen, objektorientierten Entwurf kommt.

- Dabei muss entschieden werden, wie die fachlichen und technischen Anforderungen auf Klassen, Schnittstellen und Beziehungen abgebildet werden sollen.
- In der Realität existieren dafür beliebig viele Varianten.
- Welche Entwurfsvariante ist aber besser, welche schlechter geeignet?

Der Entwurf muss nach bestimmten Qualitätskriterien bewertet werden.

- Hochwertige Software ist aber nicht nur funktional, effizient und zuverlässig.

Evolvierbarkeit

Da sich die Umgebung von Software über die Zeit ändert, muss langlebige Software kontinuierlich angepasst werden.

- Die Software muss anpassbar, weiterentwickelbar, evolvierbar sein.

Manche Entwürfe zeigen beim Versuch der Weiterentwicklung aber eine gewisse *Alterung*.

- Das sind Eigenschaften, die auch alternde Lebewesen besitzen, siehe auch [3, S. 2-6]:
- Starrheit, Zerbrechlichkeit, Unbeweglichkeit, Zähigkeit.

Robert C. Martin spricht dann davon, dass Software verrottet, vermodert (siehe [2]).

- Verrottende Software entwickelt dann gewisse Gerüche (*engl. smells*).
- Eine ganze Reihe von Werken befassen sich mit sog. Code Smells und ihrer Vermeidung, siehe z.B. [4].

Kohäsion und Kopplung

Ob ein Entwurf evolvierbar ist, wird insbesondere auch durch die strukturelle Aufteilung auf Klassen und Schnittstellen beeinflusst.

- Die Qualität des Entwurfs schlägt sich dabei in zwei Eigenschaften nieder: **Kohäsion** und **Kopplung**.

Kohäsion meint, wie gut Programmelemente eine einzelne logische Aufgabe abdecken.

- In einem System mit starker Kohäsion ist z.B. eine einzelne Klasse für genau eine Aufgabe zuständig.

Kopplung dagegen ist ein Gradmesser, wie stark die Abhängigkeiten zwischen Klassen sind.

- In einem System mit geringer Kopplung können einzelne Klassen leicht gegen andere Varianten ausgetauscht werden.
- Eine Änderung an einer Klasse führt auch nicht zwangsläufig dazu, dass an vielen anderen Klassen Änderungen vorgenommen werden müssen.

SOLID

Eine Reihe von **Entwurfsprinzipien** zielen darauf ab, die Kohäsion zu erhöhen und die Kopplung zu reduzieren.

- Solche Prinzipien sind keine starren Regeln, sondern Empfehlungen.
- Sie helfen aber dabei, qualitativ hochwertige Entwürfe zu erstellen.

Ein bekannter Satz solcher Prinzipien ist unter der Abkürzung **SOLID** bekannt, siehe [1].

- Die einzelnen Buchstaben dieses Begriffs stehen für:
- **S**: Single Responsibility Principle (SRP)
- **O**: Open Closed Principle (OCP)
- **L**: Liskov Substitution Principle (LSP)
- **I**: Interface Segregation Principle (ISP)
- **D**: Dependency Inversion Principle (DIP)

Wir werden uns im Folgenden alle diese Prinzipien ansehen.

Single Responsibility Principle

HOCHSCHULE HAMM-LIPPSTADT

Das **Prinzip der eindeutigen Verantwortung** (engl. Single Responsibility Principle) zielt direkt auf die Erhöhung der Kohäsion ab.

- Es besagt, dass Programmelemente nur eine Anforderung eines Akteurs umsetzen sollen.
- Unterschiedliche Aufgaben und Aspekte sollen getrennt voneinander implementiert werden (engl. separation of concerns).
- Anders gesagt, darf es nur einen Grund geben, ein Programmelement ändern zu müssen, siehe [2, S. 114-119].

Aus diesem Prinzip lassen sich viele weitere Heuristiken direkt oder indirekt ableiten, z.B.:

- Redundanzen im Programmcode müssen vermieden werden (engl. Don't repeat yourself).
- Eine Methode sollte entweder andere Methoden integrieren, oder selber Logik implementieren, aber nicht beides.
- Eine Methode sollte entweder ein Kommando oder eine Abfrage umsetzen, aber nicht beides (engl. Command-Query-Separation).

CSV-Daten

Wir wollen uns an einem Beispiel ansehen, wie man dieses Prinzip umsetzen kann.

- In einem Unternehmen werden alle Verkäufe in sog. CSV-Dateien abgelegt.
- CSV-Dateien sind Text-Dateien, die tabellarische Daten beinhalten.
- Die Spalten sind durch ein Semikolon voneinander getrennt.
- Die erste Zeile beinhaltet die Spaltenüberschriften.

Eine solche Datei könnte wie folgt aussehen:

```
1  Produkt;Preis;Anzahl
2  SSD Festplatte;48,9;3
3  Monitor;129,3;5
4  Tastatur;10,49;3
5  Maus;62,99;7
```

Unsere Aufgabe ist es nun, ein Programm zu entwickeln, welches die Datei einliest und den Gesamtumsatz berechnet und ausgibt.

Umsatzleser

HOCHSCHULE HAMM-LIPPSTADT

Eine erste, sehr einfache Lösung könnte wie folgt aussehen:

```
1  class UmsatzLeser
2  {
3      public static void Main()
4      {
5          string dateiname = "Daten.csv";
6          var reader = new StreamReader(dateiname);
7          reader.ReadLine();
8
9          while (!reader.EndOfStream)
10         {
11             string line = reader.ReadLine();
12             var parts = line.Split(';');
13             var preis = Convert.ToDouble(parts[1]);
14             var anzahl = Convert.ToDouble(parts[2]);
15             gesamtumsatz += anzahl * preis;
16         }
17         reader.Close();
18     }
19 }
```

Bewertung

Diese Lösung funktioniert zwar, ist aber kein wirklich guter Entwurf.

- Wir haben in der Klasse *UmsatzLeser* zu viele unterschiedliche Dinge getan.
- Die Klasse liest eine Datenquelle (Datei), die Daten werden in das gewünschte Format transformiert und Berechnungen werden angestellt.

Keiner der einzelnen Aspekte kann aktuell eigenständig getestet werden.

- Die Transformation der Daten oder die Berechnung funktioniert nicht ohne eine CSV-Datei.

Wir wollen daher den Entwurf verbessern.

- Dazu verteilen wir die einzelnen Aspekte auf mehrere eigene Klassen und Methoden.

Nach der Änderung

HOCHSCHULE HAMM-LIPPSTADT

Wir erstellen einige neue Klassen:

- Ein Objekt der Klasse *SalesPosition* bildet genau eine Zeile der Datei ab.
- Die Klasse *CsvParser* transformiert die Datei in eine Menge solcher Objekte.
- Die Klasse *SalesCalculator* berechnet dann den Umsatz aus den Objekten.

Die Lösung sieht danach wie folgt aus:

```
1  string[] file_content = File.ReadAllLines("Data.csv");
2  IEnumerable<SalesPosition> sales = CsvParser.Parse(file_content);
3  double turnover = SalesCalculator.GetTurnover(sales);
4  Console.WriteLine(turnover);
```

Alle diese Klassen können unabhängig voneinander genutzt und getestet werden.

- Die Kohäsion der neuen Klassen ist viel höher, als in der ersten Lösung.
- Die Anwendung ist deutlich anpass-, portier- und wartbarer als vorher.

Open Closed Principle

Das Prinzip der **Offen- und Verschlossenheit (engl. Open Closed Principle)** ist das nächste Entwurfsprinzip, dass wir uns ansehen wollen.

- Das Prinzip besagt, dass Programmelemente für Modifikationen verschlossen sein sollten.
- Für Erweiterungen sollten Sie aber offen sein, siehe [2, S. 120-132].

Den bestehenden Programmcode einer Klasse zu verändern ist eine Modifikation.

- Eine Modifikation ändert das Verhalten von bestehenden Systemen.
- Entsprechend müssen Modifikationen sehr vorsichtig vorgenommen werden.

Eine Erweiterung jedoch fügt dem System zusätzliches Verhalten hinzu.

- Dies kann z.B. durch Ableiten von einer bestehenden Klasse bzw. Implementieren einer Schnittstelle geschehen.
- Es muss dann nur diese neue Klasse getestet werden, nicht jedoch der bestehende Programmcode.

Beispiel

Ein Beispiel für das Open Closed Principle haben wir bereits gesehen.

- In unserem Tic Tac Toe Spiel haben wir eine abstrakte Basisklasse *Spieler* definiert.

```
1  abstract class Spieler
2  {
3      public char Spielstein { get; set; } 4
4      public Spieler(char spielstein)
5      {
6          Spielstein = spielstein;
7      }
8
9      public abstract void Ziehe(Spielfeld feld);
10 }
```

Die Klasse gibt das Verhalten vor, das eine Klasse anbieten muss, um als Spieler zu gelten.

- Daraus haben wir dann zwei unterschiedliche Arten von Spielern abgeleitet.
- Eine Klasse *ComputerSpieler* und einen *MenschlichenSpieler*.

Netzwerkspieler

HOCHSCHULE HAMM-LIPPSTADT

Wir können nun eine weitere Art von Spieler einführen, den *Netzwerkspieler*.

- Diese könnte dann den Zug von einem entfernten Rechner empfangen.
- Wenn wir die neue Klasse von der Klasse *Spieler* ableiten, müssen wir lediglich die Methode *Ziehe()* entsprechend implementieren.

Wir können unser System dadurch ganz einfach um weitere Funktionalität erweitern.

- Die abstrakte Klasse Spieler abstrahiert von den konkreten Spielerarten.
- Mithilfe der Polymorphie realisieren wir eine Art Steckdose, in der wir unterschiedliche Arten von Spielern einstecken können.

Wir erweitern dadurch die Anwendung um neue Funktionalität, ohne bestehende Klassen verändern zu müssen.

Netzwerkspieler

HOCHSCHULE HAMM-LIPPSTADT

Betrachten wir nun einen ersten Entwurf unserer Klasse *Netzwerkspieler*:

```
1  public class Netzwerkspieler : Spieler
2  {
3      public bool IstVerbunden { get; private set; }
4
5      public void Verbinden() {
6          // Hier wird die Verbindung aufgebaut
7          IstVerbunden = true;
8      }
9
10     public override void Ziehe(Spielfeld feld) {
11         if (!IstVerbunden)
12             throw new Exception("Verbinung ist nicht aufgebaut!");
13
14         // Nun wird der Zug vom Server geholt...
15     }
16 }
```

Liskov Substitution Principle

HOCHSCHULE HAMM-LIPPSTADT

Bei dieser Implementierung gibt es leider ein Problem.

- Bevor ein Zug ausgeführt werden kann, muss mit der Methode *Verbinden()* zunächst eine Verbindung zu einem Server aufgebaut worden sein.

Ein Objekt der Klasse *Spiel* macht dies natürlich nicht.

- Das Spiel kennt die konkrete Implementierung der abstrakten Klasse *Spiel* nicht und weiß nichts über die Methode *Verbinde()*.
- Der Netzwerkspieler kann daher nicht ohne Weiteres in unserem Spiel eingesetzt werden.

Das sog. **Liskovsche Substitutionsprinzip (engl. Liskov Substitution Principle)** befasst sich genau mit dieser Problematik.

- Es beschreibt, wie Kindklassen aufgebaut sein müssen, damit sie in Systemen als gleichwertiger Ersatz von Basisklassen genutzt werden können.

Offenbar wurde in unserem ersten Entwurf gegen dieses Prinzip verstoßen.

Design by Contract

HOCHSCHULE HAMM-LIPPSTADT

Der Netzwerkspieler funktioniert nur dann vernünftig, wenn auf dem Objekt vorher einmal *Verbinden()* aufgerufen wurde.

- Eine solche Forderung wird auch als **Vorbedingung** bezeichnet.

Der Begriff der Vorbedingung stammt aus dem Konzept des **Design by Contract**.

- Damit eine Methode vernünftig arbeiten kann, müssen alle Vorbedingungen erfüllt sein.
- Bei der Methode *Ziehe()* ist das die Vorbedingung, dass die Verbindung aufgebaut wurde.

Nach der Erfüllung einer Aufgabe sind dann i.d.R. verschiedene **Nachbedingungen** erfüllt.

- Für die Methode *Ziehe()* gilt, dass auf dem Spielfeld nach dem Zug ein Spielstein mehr liegen muss als vorher.

Zudem müssen durchgängig gewisse Grundannahmen (sog. **Invarianten**) gelten.

- So bleibt die Dimension des Spielfelds durchgängig gleich (3x3).

Problemlösung

HOCHSCHULE HAMM-LIPPSTADT

Ob in einer Vererbungshierarchie das Liskovsche Substitutionsprinzip eingehalten werden kann, entscheidet sich anhand folgender Regeln:

- Die Kindklasse darf die **Vorbedingungen** der Basisklasse höchstens **abschwächen**.
- Die Kindklasse darf die **Nachbedingungen** der Basisklasse höchstens **verstärken**.

In unserem Fall wurden die Vorbedingungen der Kindklasse Netzwerkspieler allerdings erhöht.

- Entsprechend konnten wir den *Netzwerkspieler* nicht einfach in unser Spiel integrieren.

Wir können das Problem aber lösen.

- Die Vorbedingung der aufgebauten Verbindung muss immer dann erfüllt sein, wenn wir die Methode *Ziehe()* aufrufen.
- Wir können die Verbindung z.B. bereits im Konstruktor der Klasse aufbauen, oder bei jedem Zug prüfen, ob die Verbindung besteht und diese im Zweifel aufbauen.

Abhängigkeiten

HOCHSCHULE HAMM-LIPPSTADT

Abhängigkeiten entstehen dadurch, dass sich Objekte unterschiedlicher Klassen in einem Geflecht gegenseitig nutzen (Client-Server-Prinzip).

- Diese gegenseitige Nutzung von Objekten ist ein Kernprinzip der Objektorientierung.
- Das TicTacToe-Spiel kann z.B. ohne Spieler nicht funktionieren.

Viele Abhängigkeiten (ein hoher Grad der Kopplung) können aber zu einer deutlich schlechteren Anpassbarkeit und Wartbarkeit eines Systems führen.

- Selbst kleine Änderungen können dann einen erheblichen Arbeitsaufwand erzeugen.

In manchen Systemen kommt man an dem Punkt an, dass Änderungen unüberschaubare Effekte mit sich bringen.

- Es darf möglichst nichts mehr geändert werden.

Beispiel

In einem vorhergehenden Abschnitt haben wir eine Anwendung entwickelt, um einen Sportverein verwalten zu können.

```
1  class Sportverein
2  {
3      private Mitglied vorsitzender;
4      private List<Mitglied> mitglieder = new List<Mitglied>();
5
6      // Hier fehlt einiges an Programmcode...
7
8      public IEnumerable<Mitglied> AlleMitglieder()
9      {
10         return mitglieder;
11     }
12 }
```

Ein Objekt der Klasse *Sportverein* verwaltet einen Vorsitzenden und mehrere Mitglieder.

- Die Mitglieder werden in einer Objektvariablen vom Typ `List<Mitglied>` verwaltet.

List oder IEnumerable

HOCHSCHULE HAMM-LIPPSTADT

In der Methode *AlleMitglieder()* liefern wir als Ergebnis ein Objekt vom Typ `IEnumerable<Mitglied>` zurück, nicht die Liste selbst.

- Dies hat insbesondere etwas mit **Abhängigkeiten** zwischen Klassen zu tun.

Die Schnittstelle *IEnumerable* abstrahiert von allen möglichen Arten von Sammlungen.

- Dadurch könnte in der Klasse *MitgliederListe* die Verwaltung der Mitglieder z.B. auf ein Array umgestellt werden.
- Der Nutzer der Klasse würde dies nicht bemerken.

Abhängigkeiten sollten möglichst nur zu **Abstraktionen** (z.B. Schnittstellen), nicht aber zu konkreten Klassen aufgebaut werden.

- Variablen sollte keine Referenz auf eine konkrete Klasse halten, lediglich zu Abstraktionen.
- Klassen sollten nicht von konkreten Klassen ableiten.

Interface Segregation Principle

HOCHSCHULE HAMM-LIPPSTADT

Schnittstellen sind das wichtigste Mittel, um von konkreten Klassen zu abstrahieren.

- Schnittstellen sollten dabei so schlank wie möglich sein.

Ein Modul sollte über eine Schnittstelle nur diejenigen Methoden präsentiert bekommen, die es für eine bestimmte Aufgabe auch wirklich braucht.

- Aber eben nicht mehr.

Diese Forderung ist auch als das **Schnittstellen-Segregationsprinzips (engl. Interface Segregartion Principle, ISP)** bekannt, siehe [2, S. 166].

- Zu schwergewichtige Schnittstellen sollten in mehrere Schnittstellen aufgetrennt werden.
- Dieses Prinzip ist eng verwandt mit dem Prinzip der eindeutigen Verantwortung.
- Auch Abstraktionen (Schnittstellen) sollten dieser eindeutigen Verantwortung gerecht werden.

Dependency Inversion Principle

HOCHSCHULE HAMM-LIPPSTADT

In unserem TicTacToe-Spiel benutzt das Spiel unterschiedliche Arten von Spielern.

- Es entsteht eine Abhängigkeitsbeziehung zwischen diesen Elementen.
- Die Klasse *Spiel* kann darin als das übergeordnete Modul bezeichnet werden.

Das **Abhängigkeitsumkehr-Prinzip (engl. Dependency Inversion Principle, DIP)** stellt in einer solchen Beziehung die folgenden Regeln auf, siehe [2, S. 151-161]:

- Module höherer Ebenen sollten nicht von Modulen niedrigerer Ebenen abhängen.
- Beide sollten von Abstraktionen abhängen.
- Abstraktionen sollten nicht von Details abhängen.
- Details sollten von Abstraktionen abhängen.

Robert C. Martin bezeichnet die Einhaltung dieses Prinzips auch als das Markenzeichen guten objektorientierten Designs, siehe [2, S. 161].

Dependency Inversion Principle

Die Abhängigkeit zwischen Spiel und Spieler haben wir bereits über eine Abstraktion geregelt.

- Wir haben gesehen, dass dazu entweder eine Schnittstelle *ISpieler* oder eine abstrakte Basisklasse eingesetzt werden kann.

Das Abhängigkeitsumkehr-Prinzip benennt dabei auch, wer für die Definition dieser Abstraktion verantwortlich sein sollte.

- Die höhere Ebene (das Spiel) sollte diese Abstraktion vorgeben.
- Das Spiel sollte definieren, was es sich unter einem Spieler vorstellt.
- Die Spieler müssen diese Vorgabe dann einhalten.

Würden Spiel und die Spieler in unterschiedlichen Bibliotheken implementiert, hätte die Spieler-Bibliothek dann eine Abhängigkeit zu der Spiel-Bibliothek.

- Nicht anders herum.
- Dadurch entsteht auch der Name des Prinzips: Abhängigkeiten-Umkehrprinzip.

Zusammenfassung

Wir haben heute gelernt, ...

- was die Aufgabe des objektorientierten Entwurfs ist.
- welche Qualitätskriterien einen guten von einem schlechten Entwurf unterscheiden.
- was mit Kohäsion und Kopplung gemeint ist.
- welche Entwurfsprinzipien dabei helfen, die Qualität eines Entwurfs zu verbessern.
- wofür der Begriff SOLID steht und wie die einzelnen Prinzipien darin umgesetzt werden.

Aufgaben

HOCHSCHULE HAMM-LIPPSTADT

In einem Unternehmen sind Informationen zu allen Mitarbeitern in einer Text-Datei abgelegt:

```
1  last_name, first_name, date_of_birth, email
2  Doe, John, 1982/10/08, john.doe@foobar.com
3  Ann, Mary, 1975/09/11, mary.ann@foobar.com
```

Schreiben Sie eine Konsolen-Anwendung in C#, die täglich eine E-Mail mit Geburtstagsgrüßen an diejenigen Mitarbeiter verschickt, die an diesem Tag Geburtstag haben.

- Berücksichtigen Sie dabei die SOLID Entwurfsprinzipien.

Erweitern Sie das Programm so, dass diejenigen Mitarbeiter, die an einem 29. Februar Geburtstag haben und es diesen Tag im Jahr nicht gibt, die Mail am Tag zuvor erhalten.

- Berücksichtigen Sie dabei das Prinzip der Offen- und Verschlossenheit.

Quellen I

[1] Robert C. Martin. *Design Principles and Design Patterns*. 2000. URL: https://fi.ort.edu.uy/innovaportal/file/2032/1/design_principles.pdf.

[2] Micah Martin und Robert C. Martin. *Agile Principles, Patterns, and Practices in C#*. 1. Aufl. Pearson, 20. Juli 2006. 691 S.

[3] Joachim Goll. *Architektur- und Entwurfsmuster der Softwaretechnik: Mit lauffähigen Beispielen in Java*. 2., aktualisierte Aufl. 2014 Edition. Wiesbaden: Springer Vieweg, 2014. 432 S. ISBN: 978-3-658-05531-8.

[4] Uwe Post. *Besser coden: Best Practices für Clean Code. Das ideale Buch für die professionelle Softwareentwicklung*. 2. Aufl. Bonn: Rheinwerk Computing, 31. Aug. 2021. ISBN: 978-3-8362-8492-9.