Objektorientierte Programmierung

12 - Entwurfsmuster

Alexander Stuckenholz

Version 2022-05-04

Inhalt

HOCHSCHULE HAMM-LIPPSTADT

Entwurfsmuster

HOCHSCHULE HAMM-LIPPSTADT

Die in der Entwurfsphase eingesetzten Muster werden als **Entwurfsmuster** bezeichnet.

- Sie zielen darauf ab, die Architektur der Anwendung positiv zu beeinflussen, so dass die Kohäsion steigt und die Koppelung sinkt.

Für große Verbreitung von objektorientierten Entwurfsmustern hat dabei das Buch *Design Patterns - Elements of Reusable Object-Oriented Software* gesorgt, siehe: [1].

- Die Autoren (Erich Gamma, Richard Helms, Ralph Johnson und John Vlissides) werden heute auch als die *Gang-of-Four* (GoF) bezeichnet.

In dem Buch werden 23 Muster in drei unterschiedliche Kategorien unterschieden:

- Die **erzeugende Muster** befassen sich mit der Erzeugung von Objekten aus Klassen.
- **Strukturmuster** fassen Objekte zu gewissen statischen Strukturen zusammen.
- **Verhaltensmuster** beschreiben, wie Objekte durch Zusammenwirken ein bestimmtes Verhalten erzeugen können.

Objekterzeugung

HOCHSCHULE HAMM-LIPPSTADT

Objekte werden mit Hilfe des `new` Operators aus den Klassen erzeugt.

- Dazu wird dann der Konstruktor der Klasse aufgerufen, der das Objekt initialisieren kann.
- Dabei kann es aber zu Problemen kommen, die ein Konstruktor allein nicht lösen kann:

Ein Konstruktor kann nur genau den Typ erzeugen, der durch die Klasse festgelegt ist.

- Ein Konstruktor der Klasse *Person* kann nur ein *Person*-Objekt erzeugen.
- Mitunter soll aber in Abhängigkeit der übergebenen Parameter ein Subtyp geliefert werden, z.B. *Student*.

Die Erzeugung eines Objekts kann zudem recht kompliziert sein.

- Diese Mechanik dann im Konstruktor abzubilden, verstößt gegen das Prinzip der einzigen Verantwortung.

Fabrik

Diese Probleme können durch das Entwurfsmuster **Fabrik (engl. Factory)** gelöst werden.

- Die Fabrik ist ein erzeugendes Muster, es befasst sich mit der Erzeugung von Objekten.

Eine Fabrik wird durch eine eigene Klasse abgebildet.

- Der Name der Klasse sollte das Wort *Fabrik* im Namen tragen, z.B. *PersonFactory*.
- Eine solche Fabrikklasse bietet eine oder mehrere, meist statische Methoden an, die Objekte erzeugen und zurückliefern.

Der Rückgabetyp dieser Methoden ist oft eine Abstraktion.

- Also eine Schnittstelle oder eine (abstrakte) Basisklasse.
- Dadurch können unterschiedliche Typen zurückgeliefert werden.

Beispiel

Im folgenden Beispiel soll aus einem übergebenen Text ein passendes Objekt erzeugt werden.

```
1  class PersonFactory
2  {
3      public static Person GetPerson(string data)
4      {
5          var elements = data.Split(';');
6          if (elements.Count == 3)
7              return new Student(elements[0], elements[1], Convert.ToInt32(elements[2]));
8          else
9              return new Person(elements[0], elements[1]);
10     }
11 }
```

Je nachdem, wie die Parameter beschaffen sind, werden unterschiedliche Typen erzeugt:

- `var obj1 = PersonFactory.GetPerson("Achim;Schäfer");`
- `var obj2 = PersonFactory.GetPerson("Elvira;Öztürk;1234567");`
- In diesem Fall ist `obj1` eine Person, `obj2` jedoch ein Student.

Einzelstück

Das **Einzelstück** (*engl. Singleton*) ist ebenfalls ein erzeugendes Entwurfsmuster.

- Es sorgt dafür, dass zur Laufzeit einer Anwendung nur ein einziges Objekt einer bestimmten Klasse existieren kann.
- Es nutzt dazu eine Fabrikmethode, liefert aber immer dasselbe Objekt zurück.
- Das Einzelstück ist in vielen Szenarien hilfreich, z.B. beim Logging oder einer Datenbankverbindung.

Um sicherzustellen, dass nur ein einziges Objekt einer Klasse existieren kann, darf der Konstruktor der Klasse nicht öffentlich sein.

- Das einzige Exemplar der Klasse wird dann über eine statische Fabrikmethode beschafft.

Beispiel

HOCHSCHULE HAMM-LIPPSTADT

```
1  class DatabaseConnectionAsSingleton
2  {
3      private static DatabaseConnectionAsSingleton instance = null;
4
5      private DatabaseConnectionAsSingleton()
6      {
7          // Hier kann das Objekt noch weiter initialisiert werden
8      }
9
10     public static DatabaseConnectionAsSingleton Instance
11     {
12         get
13         {
14             if (instance == null)
15                 instance = new DatabaseConnectionAsSingleton();
16
17             return instance;
18         }
19     }
20 }
```

Nutzung des Einzelstücks

Ein Objekt der Klasse *DatabaseConnectionAsSingleton* kann nun nicht mehr mit dem Operator `new` erzeugt werden.

- Der Compiler würde wegen des privaten Konstruktors einen Fehler melden.
- Das einzige Objekt der Klasse wird über die Fabrikmethode *Instance* beschafft:
- `var instance = DatabaseConnectionAsSingleton.Instance;`
- Auf dem Objekt `instance` können dann alle öffentlichen Methoden aufgerufen werden.

Achtung: Die Nutzung eines Einzelstücks verbirgt Abhängigkeit im Code.

- Abhängigkeiten sollten aber immer explizit sein.
- Hängt A von B ab, sollte man B an A übergeben müssen.
- Entsprechend wird davor gewarnt, das Einzelstück übermäßig einzusetzen.
- Stattdessen sollte besser *Dependency Injection* eingesetzt werden.

Iterator

Das Entwurfsmuster **Iterator** ist ein Beispiel eines Verhaltensmusters.

- Der Iterator wird im Zusammenhang mit Datenstrukturen eingesetzt.
- Bekanntlich können Datenstrukturen ihre Daten auf ganz unterschiedliche Weise organisieren, z.B. in einem Array, als einfach-verkettete Liste, als Baum, ...

Der Iterator abstrahiert von dieser Organisation.

- Egal, wie die Daten abgelegt sind, mit dem Iterator kann man über die Elemente iterieren.

Im .Net-Framework wird der Iterator über die Schnittstelle *IEnumerator* abgebildet.

- Die Klasse, welche diese Schnittstelle implementiert, muss die Eigenschaft *Current* und die Methoden *MoveNext()* und *Reset()* anbieten.

Beispiel

HOCHSCHULE HAMM-LIPPSTADT

Zu unserer einfach-verketteten Liste können wir nun eine eigene Iterator-Klasse konstruieren.

- Die Klasse *LinkedListEnumerator* implementiert die Schnittstelle *IEnumerator*.
- Sie erhält im Konstruktor ein Objekt einer einfach-verketteten List.
- Die wichtigste Methode *MoveNext()* schiebt eine Art Zeiger-Objekt durch die Liste.

```
1  public class LinkedListEnumerator : IEnumerator
2  {
3      private LinkedList list;
4      private ListNode current;
5
6      // Der Rest fehlt hier
7
8      public bool MoveNext()
9      {
10         current = current != null ? current.next : null;
11         return current != null;
12     }
13 }
```

IEnumerable

Datenstrukturen, zu denen ein Iterator existiert, implementieren im .Net-Framework die Schnittstelle *IEnumerable*.

- Die einzige Methode *GetEnumerator()* liefert dann einen entsprechenden *Iterator* zurück

```
1  public IEnumerator GetEnumerator()
2  {
3      return new LinkedListEnumerator(this);
4  }
```

So ausgestattet kann die einfach-verkettete Liste dann auch in einer `foreach`-Schleife genutzt werden:

```
1  var list = new LinkedList();
2  list.pushBack(1);
3  list.pushBack(2);
4
5  foreach (object o in list)
6      Console.WriteLine(o);
```

TicTacToe

Erinnern wir uns an unser TicTacToe-Spiel.

- Unser Entwurf sollte offen für Erweiterungen, aber verschlossen für Änderungen sein.
- Daher haben wir mit einer abstrakten Basisklasse bzw. einer Schnittstelle eine Abstraktion für die unterschiedlichen Arten von Spielern geschaffen.
- So können zur Laufzeit dann unterschiedliche Arten von Spielern gegeneinander antreten.
- Zudem können neue Arten von Spielern auch nachträglich noch hinzugefügt werden, ohne das Spiel selbst verändern zu müssen.

Diese Herangehensweise ähnelt dem Entwurfsmuster **Strategie (engl. strategy)**, siehe [2].

- Dieses Verhaltensmuster sorgt dafür, dass in einem Kontext ein Algorithmus durch einen anderen austauschbar wird.

Routenplaner

Stellen wir uns vor wir müssten einen Routenplaner erstellen.

- Der Routenplaner kann aber unterschiedliche Routen berechnen, z.B. die schnellste oder die kürzeste Route.

Wir müssen also unterschiedliche Strategien vorgeben können.

- Über eine Schnittstelle definieren wir eine Abstraktion dieser Strategien.

```
1  public interface IRouteStrategy
2  {
3      Route GetRoute();
4  }
```

Wir können dann unterschiedliche solcher Strategien implementieren.

- z.B. die Klasse *FastestRoutingStrategy* bzw. die *ShortesRoutingtStrategy*.

Strategien anwenden

Der Routenplaner bekommt genaue eine konkrete Strategie übergeben.

- Er nutzt diese, um die Route zu berechnen:

```
1  var fastest = new FastestRoutingStrategy();
2  var planer = new Routeplaner(fastest);
3  var route = planer.ExecuteStrategy();
4  Console.WriteLine(route.Path + " --> " + route.Duration);
5
6  // Eine andere Routing-Strategie erzeugen:
7  planer.Strategy = new ShortestRoutingStrategy();
8  route = planer.ExecuteStrategy();
9  Console.WriteLine(route.Path + " --> " + route.Duration);
```

Dokument

HOCHSCHULE HAMM-LIPPSTADT

Stellen wir uns vor, wir müssten die Struktur eines Textdokuments auf Klassen abbilden.

- Ein Dokument besteht aus eine Menge von einzelnen Bestandteilen.
- Jeder Bestandteil kann z.B. normaler Text, Text in Fettdruck, oder ein Hyperlink sein.

Eine solche Struktur könnte z.B. wie folgt aussehen (siehe [3]):

```
1  public abstract class DocumentPart {
2      public string Text { get; set; }
3  }
4
5  public class PlainText : DocumentPart { }
6  public class BoldText : DocumentPart { }
7  public class Hyperlink : DocumentPart {
8      public string Url { get; set; }
9  }
10
11 public class Document {
12   public List<DocumentPart> Parts { get; private set; } = new List<DocumentPart>();
13 }
```

Dokumentenformate

HOCHSCHULE HAMM-LIPPSTADT

Mit Hilfe dieser objektorientierten Klassenstruktur können nun Dokumente als Objekte abgebildet werden:

```
1  var doc = new Document();
2  doc.Parts.Add(new BoldText("Überschrift"));
3  doc.Parts.Add(new PlainText("Dies ist ein normaler Textabschnitt"));
4  doc.Parts.Add(new Hyperlink("Dies ist ein Link", "https://www.hshl.de"));
```

Diese Datenstruktur wollen wir oft in unterschiedlichen Formaten ausgeben können.

- z.B. als Html- oder als Latex-Dokument.

Ein einfache Herangehensweise wäre dann, alle Klassen mit entsprechenden Methoden auszustatten, z.B. *ToHtml()*:

```
1  // So könnte z.B. die Methode ToString() für die Klasse Hyperlink aussehen:
2  public string ToHtml() {
3      return "<a href=\" + docPart.Url + "\">" + docPart.Text + "</a>";
4  }
```

Besucher

Für jedes neue Datenformat müssten wir die bestehenden Klassen *Document*, *PlainText*, *BoldText* und *Hyperlink* um zusätzliche Methoden erweitern.

- Ein Entwurf sollte aber für Modifikationen verschlossen, für Erweiterung aber offen sein (Prinzip der Offen- und Verschlossenheit).

Zusätzliche Dokumentenformate sollten also als eigene Klassen implementiert werden.

- Dazu bietet sich das Entwurfsmuster **Besucher (engl. Visitor)** an.
- Dieses Verhaltensmuster dient der Kapselung von Operationen, die auf Elementen einer Objektstruktur angewandt werden sollen.

Für jedes Datenformat wird dabei eine Besucherklasse eingeführt, die eine bestimmte Schnittstelle implementiert.

- Die Elemente der Datenstruktur, in unserem Fall die Dokumentenbestandteile, können von diesen Besucherobjekten dann besucht werden.

IDocumentConverterVisitor

HOCHSCHULE HAMM-LIPPSTADT

Die Schnittstelle *IDocumentConverterVisitor* definiert, welche Methoden eine Besucherklasse implementieren muss, um die Datenstruktur in ein konkretes Format zu überführen:

```
1  public interface IDocumentConverterVisitor
2  {
3    void Visit(PlainText docPart);
4    void Visit(BoldText docPart);
5    void Visit(Hyperlink docPart);
6  }
```

Ein konkreter Besucher implementiert nun diese Schnittstelle:

```
1  public class HtmlDocumentConverter : IDocumentConverterVisitor
2  {
3      // Der Rest fehlt hier...
4
5      public void Visit(Hyperlink docPart)
6      {
7          output.Append($"<a href=\"{docPart.Url}\">{docPart.Text}</a>");
8      }
9  }
```

Besucher empfangen

HOCHSCHULE HAMM-LIPPSTADT

Die Elemente der Dokumentenstruktur müssen nun in der Lage sein, einen Besucher zu empfangen.

- Dazu bieten sie die Methode *Accept* an, die als Parameter ein Objekt vom Typ *IDocumentConverterVisitor* empfängt.

```
1  public class Document
2  {
3    public List<DocumentPart> Parts { get; private set; } = new List<DocumentPart>();
4
5    public void Accept(IDocumentConverterVisitor visitor)
6    {
7      foreach (var part in Parts)
8        part.Accept(visitor);
9    }
10 }
```

Besucher benutzen

HOCHSCHULE HAMM-LIPPSTADT

Der *HtmlDocumentConverter* kann die Dokumentenstruktur nun besuchen und dies in ein Html-Dokument konvertieren.

```
1  var doc = new Document();
2  doc.Parts.Add(new BoldText("Überschrift"));
3  doc.Parts.Add(new PlainText("Dies ist ein normaler Textabschnitt"));
4  doc.Parts.Add(new Hyperlink("Dies ist ein Link", "https://www.hshl.de"));
5
6  var html = new HtmlDocumentConverter();
7  doc.Accept(html);
8  Console.WriteLine("Als Html:\n" + html.ToString() + "\n");
```

Es können zudem leicht neue Klassen eingeführt werden, welche die Schnittstelle *IDocumentConverterVisitor* implementieren.

- Diese bilden dann zusätzliche Formate ab.
- Die bestehenden Klassen der Dokumentenstruktur müssen dazu nicht verändert werden.

Ereignisse

Objekte stellen anderen Objekten ihre Dienste über Methoden zur Verfügung.

- Dies ähnelt einer klassischen Client-Server-Beziehung.
- Die Interaktion geht dabei immer vom Client-Objekt aus.

Objekte verwalten aber auch ihren eigenen inneren Zustand.

- z.B. der Kontostand eines Bankkontos, oder ob ein Button geklickt wurde.
- Der Zustand kann sich im Prinzip jederzeit ändern.
- Ist ein Client an diesen Änderungen interessiert, ist es wenig performant, den Server immer wieder zu fragen, ob der Zustand sich geändert hat (polling).

Die Zustandsänderung eines Objekts kann man sich als **Ereignis** vorstellen.

- In vielen Situationen ist es hilfreich, wenn Objekte über solche Ereignisse informieren können.

Beobachter

HOCHSCHULE HAMM-LIPPSTADT

Das Entwurfsmuster **Beobachter (engl. Observer)** ist ein Verhaltensmuster.

- Es sorgt dafür, dass Objekte eine Zustandsänderung in Form von Ereignissen melden können.

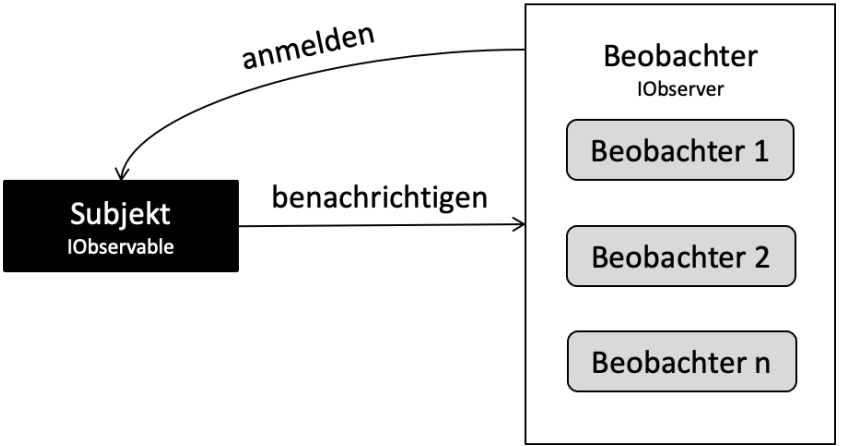
An dem Muster sind zwei Arten von Objekten beteiligt.

- Das Objekt, welches Ereignisse melden kann, wird als **Subjekt** bezeichnet.
- Ein Subjekt implementiert die Schnittstelle *IObservable*.
- Sie ermöglicht es den Beobachtern, sich für die Benachrichtigung an- und abzumelden.

Ein **Beobachter** wiederum implementiert die Schnittstelle *IObserver*.

- Sie zwingt den Beobachter dazu, eine Methode zu implementieren, die das Subjekt zur Benachrichtigung aufrufen kann.

Beobachter


anmelden
Beobachter
IObserver
Beobachter 1
Subjekt
IObservable
benachrichtigen
Beobachter 2
Beobachter n

IObservable und IObserver

HOCHSCHULE HAMM-LIPPSTADT

Die Schnittstelle *IObservable* muss durch das zu beobachtende Subjekt implementiert werden:

```
1  interface IObservable
2  {
3      void RegistriereBeobachter(IObserver b);
4      void EntferneBeobachter(IObserver b);
5      void BenachrichtigeAlleBeobachter();
6  }
```

Die Schnittstelle *IObserver* wird durch alle Beobachter implementiert:

```
1  interface IObserver
2  {
3      void AenderungIstEingetreten(IObservable quelle);
4  }
```

Bankkonto 1/2

HOCHSCHULE HAMM-LIPPSTADT

Die Änderungen an einem Bankkonto sollen beobachtbar sein:

```
1  public class Bankkonto : IObservable
2  {
3      private double kontostand;
4      private List<IObserver> beobachter = new List<IObserver>();
5
6      public void RegistriereBeobachter(IObserver b)
7      {
8          beobachter.Add(b);
9      }
10
11     public void EntferneBeobachter(IObserver b)
12     {
13         beobachter.Remove(b);
14     }
15
16     public void BenachrichtigeAlleBeobachter()
17     {
18         foreach (var b in beobachter)
19             b.AenderungIstEingetreten(this);
```

Bankkonto 2/2

HOCHSCHULE HAMM-LIPPSTADT

```
1       public double Kontostand
2       {
3           get { return kontostand; }
4       }
5
6       public void Einzahlen(double betrag)
7       {
8           kontostand += betrag;
9           BenachrichtigeAlleBeobachter();
10      }
11
12      public void Auszahlen(double betrag)
13      {
14          kontostand -= betrag;
15          BenachrichtigeAlleBeobachter();
16      }
17  }
```

Bank

Die Bank hat die Fähigkeit, Änderungen zu beobachten:

```
1  public class Bank : IObserver
2  {
3      public void AenderungIstEingetreten(IObservable quelle)
4      {
5          var konto = quelle as Bankkonto;
6
7          if (konto != null)
8          {
9              Console.WriteLine("Eine Änderung an einem Bankkonto ist eingetreten!");
10             Console.WriteLine("Neuer Kontostand: " + konto.Kontostand);
11         }
12     }
13 }
```

Zusammenspiel

Wir können nun an Objekten der Klasse Bankkonto die Bank als Beobachter anmelden.

- Tritt eine Änderung an einem Konto ein, wird die Bank darüber informiert.

```
1  static void Main(string[] args)
2  {
3      var bank = new Bank();
4      var konto1 = new Bankkonto();
5      var konto2 = new Bankkonto();
6
7      konto1.RegistriereBeobachter(bank);
8      konto2.RegistriereBeobachter(bank);
9
10     // Hier wird nun die Bank informiert
11     konto1.Einzahlen(1000);
12     konto2.Auszahlen(1000);
13 }
```

Sonstige hilfreiche Muster

HOCHSCHULE HAMM-LIPPSTADT

Es existieren noch viele weitere Muster, die sich im Alltag der Softwareentwicklung als sehr hilfreich erweisen.

- Ein Profi sollte sich alle Muster der Gang-of-Four einmal genauer angesehen haben.
- z.B. Facade, Proxy, Zustand, Strategie, ...

Neben den Analyse- und Entwurfsmustern existieren zudem auch Architekturmuster.

- Diese helfen dabei, geeignete Strukturen aus Komponenten bzw. Schichten zu schaffen.
- z.B. Model-View-Controller (MVC) oder Model-View-Viewmodel (MVVM).
- Solche Schichten widmen sich dann bestimmten Aufgaben in einem Softwaresystem, z.B. der Benutzerführung (engl. user interface), oder der Datenspeicherung.

Innerhalb dieser Schichten spielen dann auch speziellere Muster eine Rolle.

- z.B. bei der Datenspeicherung: Datentransferobjekt, Repositorium, ...

Zusammenfassung

HOCHSCHULE HAMM-LIPPSTADT

Wir haben heute gelernt, dass...

- auch in der Entwurfsphase eine ganze Reihe von Mustern hilfreich sind.
- dass bei den Entwurfsmustern zwischen Erzeugungs-, Struktur- und Verhaltensmustern unterschieden wird.
- dass die Fabrik und das Einzelstück Erzeugungsmuster sind.
- wozu der Iterator genutzt wird.
- wie man Datenstrukturen mithilfe des Besuchermusters um zusätzliche Operationen erweitern kann.
- wie Ereignisse mithilfe des Beobachtermusters umgesetzt werden können.

Aufgabe 1

HOCHSCHULE HAMM-LIPPSTADT

Im Begleitprojekt der Veranstaltung *Algorithmen und Datenstrukturen* ist die Klasse *ArrayList* zu finden[1].

- Erstellen Sie eine geeignete Iterator-Klasse zu dieser Datenstruktur.
- Sorgen Sie dafür, dass die Klasse *ArrayList* die Schnittstelle *IEnumerable* implementiert.

[1] `https://github.com/LosWochos76/AUD`

Aufgabe 2

HOCHSCHULE HAMM-LIPPSTADT

Erstellen Sie eine neue Klasse *ObervableArrayList*, die Sie von der *ArrayList* aus Aufgabe 1 durch Vererbung ableiten.

- Die neue Klasse soll die Schnittstelle *IObservable* implementieren.
- Sobald die Liste geändert wird, z.B. wenn ein neues Element eingefügt wird, sollen die Beobachter über die Änderung informiert werden.
- Realisieren Sie Unit-Tests, um die neue Funktionalität zu teste.

Quellen I

[1] Erich Gamma u. a. *Design Patterns. Elements of Reusable Object-Oriented Software*. 1st ed., Reprint Edition. Reading, Mass: Prentice Hall, 1994. 395 S. ISBN: 978-0-201-63361-0.

[2] Tiago Martins. *Strategy Pattern - C#*. 10. Nov. 2020. URL: https://medium.com/@martinstm/strategy-pattern-c-24b8ca1e4a8 (besucht am 17. 11. 2021).

[3] Sebastian Krysmanski. *The Visitor-Pattern Explained*. URL: https://manski.net/2013/05/the-visitor-pattern-explained/.