Modul 223

Dieses Dokument stellt eine Zusammenfassung des Lernstoffs von Modul 223 dar, der für die richtugsspezifische Lehrabschlussprüfung 2007 relevant ist. Die Zusammenfassung gliedert sich in die folgenden Teile:

Im ersten Teil wird auf die Grundlagen der objektorientierten Programmierung (OOP) eingegangen. Hier werden Begriffe wie Klassen, Objekte, Methoden usw. erklärt sowie die drei Säulen der objektorientierten Programmierung näher erläutert.

Entwurfsmuster sind das Thema des zweiten Teils. Neben den Entwurfsmustern Singleton, Strategy und Observer wird auch das zusammengesetzte Muster MVC betrachtet. Weiter geht es im zweiten Teil um allgemeine Entwurfsprinzipien der objektorientierten Programmierung.

Der dritte Teil widmet sich dem Multitasking. Hier geht es vor allem um die Theorie der nebenläufigen Programmierung, wobei das Prozess- und Thread-Scheduling vertieft betrachtet wird.

Im vierten und somit letzten Teil geht es um mehrschichtige Architekturen. Dieser Teil ist ebenfalls sehr theoretisch gehalten und betrachtet die ganze Thematik nur oberflächlich.

Freigabe

Version	***Datum***	***Bearbeiter***	***Änderungen***
0.01	15.05.2007	Patrick Bucher	- Dokument aufgesetzt - Theorie objektorientierter Programmierung eingefügt - Multitasking zusammengefasst - Mehrschichtige Architekturen zusammengefasst
0.02	18.05.2007	Patrick Bucher	- Verbesserungen und Korrekturen vorgenommen - Das Singleton-Entwurfsmuster zusammengefasst
0.03	01.06.2007	Flavio Hüsler Patrick Bucher	- Verbesserungen und Korrekturen vorgenommen
0.04	01.06.2007	Patrick Bucher	- Strategy und Observer zusammengefasst
0.05	02.06.2007	Patrick Bucher	- MVC zusammengefasst
1.00	02.06.2007	Patrick Bucher	- Kleinere Korrekturen vorgenommen - Freigabe

Inhaltsverzeichnis

1 Objektorientierte Programmierung....4
1.1 Klassen....4
1.2 Objekte....4
1.3 Eigenschaften....4
1.3.1 Objektstatus....4
1.4 Methoden....4
1.4.1 Methodensignatur....5
1.5 Vererbung....5
1.6 Kapselung....5
1.6.1 Sichtbarkeit....5
1.7 Die drei Säulen der objektorientierten Programmierung....5
2 Entwurfsmuster....7
2.1 Singleton....7
2.1.1 Aufbau....7
2.1.2 Definition....8
2.1.3 Implementierung....8
2.1.4 Probleme mit mehreren Threads....8
2.1.4.1 Lösungsvorschlag 1 – Eine synchronized-Methode....9
2.1.4.2 Lösungsvorschlag 2 – Synchronisierung beim ersten Durchlauf....9
2.1.4.3 Lösungsvorschlag 3 – Die Instanz von Anfang an erstellen....10
2.1.5 Vor- und Nachteile des Singleton-Musters....11
2.2 Strategy....11
2.2.1 Aufbau....11
2.2.2 Definition....12
2.2.3 Implementierung....12
2.2.3.1 Die Schnittstelle Berechnung....12
2.2.3.2 Die konkreten Strategien Addition und Subtraktion....13
2.2.3.3 Der Kontext Taschenrechner....13
2.2.3.4 Der Klient....14
2.2.4 Vor- und Nachteile des Strategy-Musters....14
2.3 Observer....15
2.3.1 Aufbau....15
2.3.2 Definition....16
2.3.3 Implementierung....16
2.3.3.1 Die Schnittstelle Subjekt....16
2.3.3.2 Die Schnittstelle Beobachter....17
2.3.3.3 Die Klasse Redaktion....17
2.3.3.4 Die Klasse Leser....18
2.3.3.5 Die Klasse Programm....19

2.3.4 Vor- und Nachteile des Observer-Musters....19
2.4 Zusammengesetzte Muster: MVC....19
2.4.1 Komponenten des MVC....20
2.4.2 Beispiel....20
2.4.2.1 View....21
2.4.2.2 Model....21
2.4.2.3 Controller....21
2.4.2.4 Ablauf....21
2.4.3 Verwendete Entwurfsmuster....21
2.5 Allgemeine Entwurfsprinzipien....22
3 Multitasking....23
3.1 Prozesse und Threads....23
3.2 Kontextwechsel....23
3.3 Thread-Prioritäten....24
3.4 Scheduling-Verfahren....24
3.4.1 Prozess-Umschaltung....24
3.4.2 Round-Robin-Verfahren....25
3.4.3 Prioritäts-Scheduling....25
3.4.4 Mehrere Schlangen....25
3.4.5 Shortest-Job-First....26
3.4.6 Zweistufiges Scheduling....26
3.5 Thread-Zustände....26
3.5.1 Wechsel von Thread-Zuständen....27
3.6 Threads programmieren....27
4 Mehrschichtige Architektur....29
4.1 Entwurfsentscheidungen....29
4.2 Schichten-Architektur....30
4.2.1 Die Drei-Schichten-Architektur....30
4.2.2 Zugriffsschichten....31
4.2.2.1 Zugriffsschicht auf das Fachkonzept....31
4.2.2.2 Zugriffsschicht auf die Datenhaltung....31
4.2.3 Weitere Architekturen....31
4.3 Entwurfsziele....32
4.4 Entwurfsheuristiken....32

1 Objektorientierte Programmierung

Strukturierte Programmiersprachen wie C sehen eine strikte Trennung von Daten und Anweisungen vor. Daten werden durch Variablen und Strukturen repräsentiert, Anweisungen können zu Funktionen zusammengefasst werden. Man hat also zwei Teile, die zwar von einander abhängig sind, jedoch nicht als eine logische Einheit implementiert werden können.

Aus dieser Problematik ist die objektorientierte Programmierung (kurz: OOP) entstanden. Daten und Anweisungen können nun in sog. Klassen zusammengehalten werden und bilden so nicht nur eine logische, sondern auch eine physische Einheit. Der Zugriff auf die Daten kann dabei eingeschränkt bzw. geschützt werden. Dies bezeichnet man als *Datenkapselung*. Hier bei gilt der folgende, wichtige Merksatz:

> *In einer objektorientierten Anwendung sollten alle Eigenschaften vor Zugriffen von aussen geschützt sein und nur über die öffentlichen Methoden (Schnittstelle) verändert werden können!*

Man spricht dann von Eigenschaften oder Membervariablen (Variablen die zu einem Objekt gehören) und Methoden oder Memberfunktionen (Funktionen die zu einem Objekt gehören).

Als die populärsten Objektorientierten Programmiersprachen gelten Java, C++ und C#. Aber auch viele Skriptsprachen bieten (mehr oder weniger) objektorientierte Möglichkeiten, so z.B. Python, PHP und Perl.

1.1 Klassen

Das wichtigste Konstrukt in der OOP ist die Klasse. In einer Klasse wird angegeben, welche Daten diese beinhaltet und wie diese verarbeitet werden können. Eine Klasse ist oftmals eine *Abstraktion* eines Sachverhalts aus der Umwelt. Mit Klassen kann man z.B. die Funktionalität eines Geräts oder einer Schnittstelle definieren, den Zugriff auf eine Datei verwalten oder auch triviale Dinge aus der realen Welt wie z.B. Katzen und Autos besser abbilden (Abstraktion).

Eine Klasse wird nicht direkt vom Programmierer verwendet (ausgenommen sind statische Elemente einer Klasse), sondern ist nur ein Bauplan für Objekte.

1.2 Objekte

Damit ein Programmierer eine Klasse auch wirklich verwenden kann, muss aus dieser zuerst ein Objekt erstellt werden. Diesen Vorgang bezeichnet man als *Instanziierung* – ein Objekt ist eine *Instanz* einer Klasse. Aus einer Klasse können beliebig viele Objekte erstellt werden, ein Objekt kann jedoch nur aus einer einzigen Klasse erstellt werden.

1.3 Eigenschaften

Die Daten eines Objekts werden durch Eigenschaften repräsentiert. Eine Eigenschaft ist dabei nichts anderes, als eine einfache Variable, welche von der ganzen Klasse verwendet werden kann und somit nicht auf eine einzige Funktion beschränkt ist.

Eine Eigenschaft kann gegen aussen geschützt werden, es ist jedoch auch möglich, dass verschiedene Klassen auf eine Eigenschaft einer Klasse zugreifen können. Es generell zu empfehlen, dass Eigenschaften gekapselt werden.

1.3.1 Objektstatus

Die Gesamtheit aller Eigenschaften eines Objekts bezeichnet man als dessen *Status* oder *Objektstatus*.

1.4 Methoden

Um die Funktionalität einer Klasse zu definieren, werden Methoden verwendet. Eine Methode ist nichts anderes, als eine Funktion, die sich innerhalb einer Klasse befindet.

Methoden können auch in ihrer Sichtbarkeit eingeschränkt werden, sodass diese nur von der Klasse aus sichtbar ist, in welcher sie auch tatsächlich steht. Schränkt man Methoden in ihrer Sichtbarkeit hingegen nicht ein, so kann sie auch von anderen Klassen her aufgerufen werden.

1.4.1 Methodensignatur

Die Signatur einer Methode besteht aus dem Namen der Methode und der Liste der Parameter mit ihren Datentypen. Sie identifiziert eine Methode innerhalb einer Klasse eindeutig. Der Rückgabewert einer Methode ist nicht Bestandteil der Methodensignatur!

1.5 Vererbung

Klassen können von anderen Klassen abgeleitet werden. Dies bezeichnet man als *Vererbung*. Die erbende Klasse (Unterklasse) erbt die gesamte Funktionalität (ausgenommen private Elemente) und die Daten der vererbenden Klasse (Oberklasse, Superklasse) und kann dieser weitere Funktionalität hinzufügen. Sie kann jedoch auch Funktionalität aus der Oberklasse überschreiben.

Vererbung wird genutzt, um eine bestimmte Klasse (Oberklasse) zu erweitern und/oder zu spezialisieren.

1.6 Kapselung

Die Daten eines Objekts (Eigenschaften) werden in der OOP für den Zugriff nach aussen geschützt. Damit man Daten aus einem Objekt lesen und/oder manipulieren kann, werden Methoden verwendet, welche den Zugriff auf die Daten *kapseln*.

In der Programmiersprache Java verwendet man dazu sog. „getter"- und „setter"-Methoden, welche lesend bzw. schreibend auf eine Eigenschaft zugreifen. C# verfolgt ein ähnliches Konzept mit sog. Properties. Hierbei bleibt es für den Anwender einer Klasse jedoch transparent, ob er direkt auf eine Eigenschaft oder via Properties auf diese zugreift.

Eine Klasse hat also genaue Kontrolle darüber, von wem und vor allem wie ihre Daten verwendet werden.

1.6.1 Sichtbarkeit

Klassen, Eigenschaften, Methoden usw. werden mit einer bestimmten Sichtbarkeit definiert. Diese bestimmt, von wo aus auf ein Element zugegriffen werden kann. Dabei wird grundsätzlich zwischen drei Sichtbarkeiten unterschieden:

1. `public`
   - Das Element kann von überall her gesehen und verwendet werden
2. `private`
   - Das Element ist nur innerhalb der aktuellen Klasse sichtbar
3. `protected`
   - Das Element ist innerhalb der aktuellen Klasse sowie von vererbten Klassen sichtbar

Verschiedene Programmiersprachen stellen weitere Sichtbarkeiten zur Verfügung, auf diese wird aber hier nicht näher eingegangen.

1.7 Die drei Säulen der objektorientierten Programmierung

Die Eigenschaften der objektorientierten Programmierung lassen sich auf die folgenden drei Kriterien zusammenfassen:

1. **Kapselung**
   - Eigenschaften werden innerhalb einer Klasse verborgen.
   - Auf bestimmte Eigenschaften wird eine öffentliche Schnittstelle angeboten.

2. **Vererbung**
   - Klassen können auf Basis bestehender Klassen erstellt werden.
   - Eine Klasse kann die Funktionalität einer anderen Klasse erben und diese erweitern.
3. **Polymorphie**
   - Sprachelemente können vielgestaltig sein.
   - Objekte erfüllen Aufgaben für verschiedene Anwendungszusammenhänge.
   - Methoden können beispielsweise den gleichen Namen, aber unterschiedliche Parameter haben.

2 Entwurfsmuster

Ein *Entwurfsmuster* (*design pattern*) beschreibt eine bewährte Schablone für ein Entwurfsproblem. Es stellt damit eine wiederverwendbare Vorlage zur Problemlösung dar.

Die Verwendung von Entwurfsmustern hat folgende Vorteile:

- Ein Entwurfsmuster beschreibt eine Problemlösung für eine bestimmte Klasse von Entwurfsproblemen, und nicht nur für ein einziges, konkretes Problem.
- Jedes Entwurfsmuster ist unter einem Namen bekannt. Dies erleichtert die Kommunikation zwischen Entwicklern.
- Entwurfsmuster sind nicht von einer bestimmten Programmiersprache abhängig, somit können (die meisten) Entwurfsmuster für alle objektorientierten Programmiersprachen eingesetzt werden.
- Die meisten Probleme in der Softwareentwicklung wurden bereits einmal gelöst. Entwurfsmuster beschreiben Lösungen für häufig auftretende Entwurfsprobleme. Der Entwickler kann also eine bereits bestehende Lösung aufgreifen und für sein konkretes Problem verwenden.
- Entwurfsmuster lösen eine Klasse von Problemen äusserst zuverlässig. Der Einsatz der richtigen Entwurfsmuster für die entsprechenden Probleme führt zu gut wartbarer und stabiler Software mit einem sauberen, objektorientierten Design.

Ist in der Softwareentwicklung von Entwrufsmustern die Rede, so bezieht man sich häufig auf die sog. „GoF-Muster"[1]. Diese Sammlung umfasst 25 Entwurfsmuster. In den folgenden Abschnitten werden einige dieser Entwurfsmuster beschrieben, es handelt sich dabei um die Muster *Singleton*, *Strategy* und *Observer*. Weiter wird auf das zusammengesetzte Muster *Model-View-Controller* und auf allgemeine, objektorientierte *Entwurfsprinzipien* eingegangen.

2.1 Singleton

Das *Singleton*-Entwurfsmuster gilt im Bezug auf die Komplexität als das einfachste Entwurfsmuster, da es nur aus einer einzigen Klasse besteht. Eine Klasse wird immer dann als Singleton implementiert, wenn es nur **eine einzige Instanz** von ihr geben darf. Singleton garantiert, dass es von einer Klasse höchstens eine Instanz geben kann und bietet dazu einen globalen Zugriffspunkt auf diese Instanz. Dies kann bei verschiedenen Verwendungszwecken sehr hilfreich sein:

- Datenbankverbindungen
- Dialoge
- Logging-Objekte
- Objekte, die globale Daten und Einstellungen verwalten

Bei all diesen Anwendungsbeispielen geht es darum, dass die Objekte nur eine einzige Instanz haben dürfen, um so Inkonsistenzen zu vermeiden.

2.1.1 Aufbau

Wie bereits erwähnt, besteht das Singleton-Entwurfsmuster nur aus einer einzigen Klasse. Diese verfügt im Wesentlichen über die folgenden Merkmale:

- Eine **statische Eigenschaft** des gleichen Typs (Typ der Klasse)
- Einen **privaten Konstruktor**
- Einer **statischen Methode**, welche die eine Instanz der Klasse zurück gibt

Oder als UML2-Klassendiagramm ausgedrückt:

1 Gang of Four: http://de.wikipedia.org/wiki/Viererbande_%28Softwareentwicklung%29

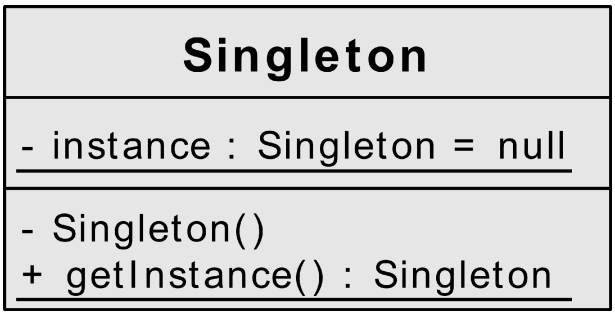

Abbildung 1: Das Singleton-Entwurfsmuster (UML2-Klassendiagramm)

2.1.2 Definition

Das Singleton-Muster *sichert, dass es nur eine Instanz einer Klasse gibt und bietet einen globalen Zugriffspunkt für diese Instanz.*

2.1.3 Implementierung

```
(01) package singleton;
(02)
(03) public class Singleton
(04) {
(05)   private static Singleton instance = null;
(06)
(07)   private Singleton() {}
(08)
(09)   public static Singleton getInstance()
(10)   {
(11)     if (Singleton.instance == null)
(12)     {
(13)       instance = new Singleton();
(14)     }
(15)     return Singleton.instance;
(16)   }
(17) }
```

Die Implementierung kann eins zu eins vom Klassendiagramm übernommen werden. Das einzige Interessante ist die Methode `getInstance()` ab Zeile (09). Zunächst wird geprüft, ob die statische Eigenschaft `instance` bereits instantiiert wurde, sodass sie nicht mehr auf `null` verweist. Falls `instance` immer noch auf `null` verweist, so wird eine neue Instanz unserer Singleton-Klasse erstellt. Dies bezeichnet man auch als *lazy instantiation.* Die Rückgabe der Instanz erfolgt in jedem Fall.

Der ganze Trick am Singleton-Entwurfsmuster besteht also darin, dass die Klasse ihre Instanzen selbst verwaltet. Der private Konstruktor verhindert, dass eine Instanz der Singleton-Klasse aus einer anderen Klasse heraus erstellt werden kann.

2.1.4 Probleme mit mehreren Threads

Die beschriebene Implementierung ist grundsätzlich richtig. Dies gilt aber nur bei der Verwendung der Klasse durch einen einzigen Thread. Verwendet man die Singleton-Klasse (genauer die Methode `getInstance()`) aus mehreren Threads heraus, so kann die Einzigartigkeit des Singleton-Objekts nicht länger gewährleistet werden!

Die Ursache dieses Problems ist einfach. Betrachten wir folgendes Szenario:

- `instance` verweist am Anfang auf `null`.
- Thread 1 betritt die Methode `getInstance()`.

    - Auf Zeile (11) wird geprüft, ob die statische Instanz bereits auf ein existierendes Objekt verweist.
    - Da `instance` immer noch auf `null` verweist, wird der bedingte Block betreten.
- Thread 2 kommt an die Reihe und betritt nun ebenfalls die Methode `getInstance()`.
    - Da Thread 1 `instance` noch nicht instanziieren konnte, verweist es immer noch auf `null`.
    - Auch in Thread 2 wird der bedingte Block betreten.
- Thread 1 ist wieder an die Reihe und instantiiert die Singleton-Klasse (Zeile (13)).
- Thread 2 erhält Rechenzeit und erstellt ebenfalls eine Instanz der Singleton-Klasse.

Wir verfügen also nun über **zwei** Instanzen unserer Singleton-Klasse!

2.1.4.1 Lösungsvorschlag 1 – Eine `synchronized`-Methode

Die erste (und einfachste) Möglichkeit, unser Synchronisationsproblem zu lösen besteht darin, dass die Methode `getInstance()` einfach mit dem Schlüsselwort `synchronized` definiert wird:

```
(01) package singleton;
(02)
(03) public class Singleton
(04) {
(05)   private static Singleton instance = null;
(06)
(07)   private Singleton() {}
(08)
(09)   public static synchronized Singleton getInstance()
(10)   {
(11)     if (Singleton.instance == null)
(12)     {
(13)       instance = new Singleton();
(14)     }
(15)     return Singleton.instance;
(16)   }
(17) }
```

Nun kann immer nur ein Thread gleichzeitig unsere Methode abarbeiten. Das Problem ist also soweit gelöst. Das gleichzeitige Ausführen der Methode `getInstance()` von mehreren Threads aus ist jedoch nur beim ersten Aufruf der Methode problematisch. Beim zweiten Aufruf existiert die Instanz garantiert. Synchronisationsbedarf besteht also nur beim allerersten Durchlauf der Methode. Das Multi-Threading wird so nur unnötig eingeschränkt!

Die Methode `getInstance()` ist offensichtlich sehr klein. Für die meisten Anwendungen ist die Performance-Einbusse durch die synchronisierte Methode verschwindend gering. Eine Lösung mit `synchronized` ist also durchaus vertretbar.

2.1.4.2 Lösungsvorschlag 2 – Synchronisierung beim ersten Durchlauf

Der zweite Lösungsansatz stellt eigentlich nur eine Verfeinerung des ersten Lösungsansatzes dar. Die Methode `getInstance()` wird nicht in jedem Fall gesperrt, sondern nur beim ersten Durchlauf.

```
(01) package singleton;
(02)
(03) public class Singleton
(04) {
(05)   private volatile static Singleton instance = null;
(06)
(07)   private Singleton() {}
(08)
(09)   public static Singleton getInstance()
(10)   {
(11)     if (Singleton.instance == null)
(12)     {
(13)       synchronized (Singleton.class)
(14)       {
(15)         if (Singleton.instance == null)
(16)         {
(17)           Singleton.instance = new Singleton();
(18)         }
(19)       }
(20)     }
(21)     return instance;
(22)   }
(23) }
```

Auf Zeile (13) wird ein **synchronisierter Codeblock** eingeleitet. Statt die Methode nur für alle anderen Threads zu sperren, wird nur ein einzelner Befehlsblock innerhalb der Methode synchronisiert. Und diese Synchronisierung ist auch nur dann notwendig, wenn unsere Instanz noch nicht erstellt wurde. Dies wird auf Zeile (11) geprüft.

Befinden wir uns erst einmal im synchronisierten Block, so müssen wir auf Zeile (15) diese Prüfung noch einmal durchführen, um sicherzustellen, dass vor dem Sperren des Objekts (mithilfe des `synchronized`-Schlüsselwortes) noch keine Instanz erzeugt wurde. Erst dann wird auf Zeile (17) unsere Instanz erzeugt.

Auf Zeile (5) bei der Deklaration von `instance` ist ein neues Schlüsselwort hinzu gekommen; **`volatile`**. Dieses Schlüsselwort sichert, dass mehrere Threads richtig mit der Eigenschaft umgehen, wenn sie dann auf Zeile (17) instantiiert wird.

Achtung:
Das zweifach-geprüfte Sperren funktioniert erst ab Java 5! Wer die JDK-Version 1.4 oder eine noch ältere Version einsetzt, der muss auf eine andere Variante zurückgreifen (Lösungsvorschlag 1 oder 3).

2.1.4.3 Lösungsvorschlag 3 – Die Instanz von Anfang an erstellen

Statt der bisher verwendeten lazy instantiation können wir unsere Instanz auch von Anfang an erstellen:

```
(01) package singleton;
(02)
(03) public class Singleton
(04) {
(05)   private static Singleton instance = new Singleton();
(06)
(07)   private Singleton() {}
(08)
(09)   public static Singleton getInstance()
(10)   {
(11)     return Singleton.instance;
(12)   }
(13) }
```

Statt also `instance` am Anfang nur mit `null` zu initialisieren, erstellen wir nun gleich eine Instanz unserer Singleton-Klasse. Die Java-Laufzeitumgebung garantiert uns also, dass wir immer eine Instanz haben - garantiert Thread-sicher. Das Problem an dieser Lösung besteht jedoch darin, dass wir auch eine Instanz erstellen, wenn diese in der Applikation möglicherweise gar nicht verwendet wird. Wir verschenken also unnötig Speicher und nehmen eine etwas längere Startzeit unserer Anwendung in Kauf.

2.1.5 Vor- und Nachteile des Singleton-Musters

- Vorteile
  - Es kann garantiert werden, dass ein Objekt nur ein einziges mal existiert.
  - Es wird ein globaler Zugriffspunkt auf dieses Objekt zur Verfügung gestellt.
    - ➔ Das Singleton-Muster stellt eine saubere, objektorientierte Umsetzung für globale Variablen dar.
- Nachteile
  - Ob die Instanz wirklich einzigartig sein kann, hängt von der jeweiligen Plattform ab.
    - ➔ Manche Programmiersprachen unterstützen keine statischen Eigenschaften und Methoden, der Singleton kann nicht implementiert werden.
    - ➔ Verschiedene Plattformen können unterschiedliche Gültigkeitsbereiche für Instanzen haben. Somit kann sich ein Singleton auf verschiedenen Plattformen unterschiedlich verhalten.
  - Der Programmierer muss sich selbst um die Thread-Sicherheit kümmern.
  - Ein Singleton-Objekt kann zwar zur Laufzeit zerstört werden, der Programmierer kann aber nicht wissen, wann eine Singleton-Instanz nicht mehr verwendet wird.
    - ➔ Ein sinnvoller Zeitpunkt für die Zerstörung einer Singleton-Instanz kann in der Regel nicht ermittelt werden.

2.2 Strategy

Das *Strategy*-Muster definiert eine Familie von austauschbaren Algorithmen. Diese Algorithmen erfüllen gegen aussen einen ähnlichen Zweck, sie unterscheiden sich jedoch in ihrer Implementierung. Das Stragety-Muster wird verwendet, wenn:

- ein Programm verschiedene Algorithmen für eine Aufgabe verwendet
- das Verhalten von Algorithmen gekapselt werden soll
- erst zur Laufzeit klar wird, welche Implementierung eines Algorithmus verwendet werden soll

Beim Strategy-Muster geht es somit immer um eine Familie von Algorithmen mit der gleichen Schnittstelle, die sich nur in ihrer Implementierung unterscheiden.

2.2.1 Aufbau

Es wird eine Schnittstelle definiert, welche die Definition aller unterstützten Algorithmen enthält. Diese Schnittstelle wird dann als „Strategie" bezeichnet. Die Algorithmen werden dann von verschiedenen Klassen implementiert, jede Implementierung der Strategie stellt eine „konkrete Strategie" dar.

Die konkreten Strategien werden dann von einem Kontext verwendet. Dieser ruft die Algorithmen jedoch nicht direkt, sondern über die Schnittstelle „Strategie" auf. Der Kontext stellt oftmals Methoden zur Verfügung, womit der Algorithmus ausgetauscht werden kann (z.B. durch setter-Methoden in Java). Die Referenz auf den konkreten Algorithmus wird dann im Kontext als Eigenschaft gespeichert.

Der Kontext wird immer von einem Klienten verwendet. Dieser ruft bestimmte Methoden auf dem Kontext auf. Dieser Aufruf wird dann innerhalb des Kontext weiter an die entsprechende

konkrete Strategie delegiert.

Das ganze präsentiert sich dann folgendermassen als UML2-Klassendiagramm:

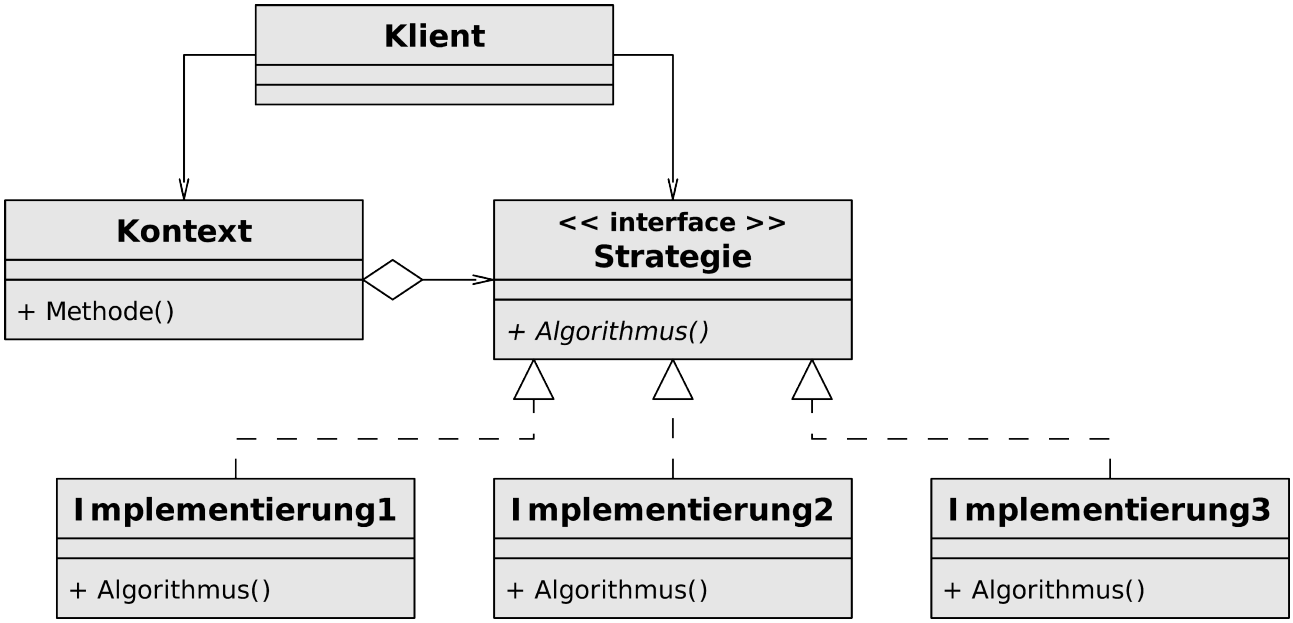

Abbildung 2: Das Strategy-Entwurfsmuster (UML2-Klassendiagramm)

2.2.2 Definition

Das Strategy-Muster *definiert eine Familie von Algorithmen, kapselt sie einzeln und macht sie austauschbar. Das Strategy-Muster ermöglicht es, den Algorithmus unabhängig von den Clients die ihn einsetzen, variieren zu lassen.*

2.2.3 Implementierung

Für einen Taschenrechner sind verschiedene Rechenarten zu implementieren, unter anderem sind dies die Addition und die Subtraktion. Es sollen immer zwei Zahlen angegeben werden können, die Berechnung ergibt dann ein Resultat.

Die Schnittstelle der beiden Rechenarten Addition und Subtraktion ist somit gleich (es werden immer zwei Zahlen erwartet), der konkrete Algorithmus unterscheidet sich jedoch (bei der Addition sollen die Zahlen addiert werden, bei der Subtraktion soll die eine Zahl von der anderen subtrahiert werden). Somit müssen folgende Klassen und Schnittstellen erstellt werden:

- Die Schnittstelle `Berechnung`
- Die konkreten Strategien `Addition` und `Subtraktion`
- Ein `Taschenrechner` als Kontext und ein `Klient`

2.2.3.1 Die Schnittstelle `Berechnung`

Die Schnittstelle Berechnung definiert eine einzige Methode, die Methode `berechne()`. Diese erwartet zwei ganzzahlige Werte.

```
(01) package strategy;
(02)
(03) public interface Berechnung
(04) {
(05)   public int berechne(int a, int b);
(06) }
```

2.2.3.2 Die konkreten Strategien `Addition` und `Subtraktion`

Die beiden Klassen `Addition` und `Subtraktion` stellen die konkreten Strategien dar und implementieren somit die Schnittstelle `Berechnung`.

Die Klasse `Addition`:

```
(01) package strategy;
(02)
(03) public class Addition implements Berechnung
(04) {
(05)   public int berechne(int summandA, int summandB)
(06)   {
(07)     return summandA + summandB;
(08)   }
(09) }
```

Die Klasse `Subtraktion`:

```
(01) package strategy;
(02)
(03) public class Subtraktion implements Berechnung
(04) {
(05)   public int berechne(int minuend, int subtrahend)
(06)   {
(07)     return minuend - subtrahend;
(08)   }
(09) }
```

Diese beiden Klassen unterscheiden sich nur in ihrem Algorithmus; die Klasse `Addition` addiert die beiden Parameter während die Klasse `Subtraktion` den Wert des zweiten Parameters vom Wert des ersten Parameters subtrahiert. Das Resultat wird in beiden Fällen zurückgegeben.

2.2.3.3 Der Kontext `Taschenrechner`

Der Taschenrechner stellt den eigentlichen Kontext der Anwendung dar:

```
(01) package strategy;
(02)
(03) public class Taschenrechner
(04) {
(05)   private Berechnung rechenart;
(06)
(07)   public Taschenrechner()
(08)   {
(09)     rechenart = new Addition();
(10)   }
(11)
(12)   public void setRechenart(Berechnung rechenart)
(13)   {
(14)     this.rechenart = rechenart;
(15)   }
(16)
(17)   public int berechne(int a, int b)
(18)   {
(19)     return rechenart.berechne(a, b);
(20)   }
(21) }
```

Die Klasse `Taschenrechner` verfügt über eine Eigenschaft namens `rechenart` des Typs `Be-`

`rechnung`, diese ist auf Zeile (05) definiert.

Der Konstruktor belegt diese Eigenschaft mit der Rechenart `Addition`, dies geschieht auf Zeile (09). Es wird somit standardmässig die Addition verwendet. Der Algorithmus lässt sich doch mithilfe der Methode `setRechenart()` austauschen, diese ist von Zeile (12) bis und mit Zeile (15) definiert.

Mit der Methode `berechne()`, die auf den Zeilen (17) bis und mit (20) definiert ist, wird die eigentliche Berechnung durchgeführt. Die beiden Parameter werden dabei zur Berechnung an die jeweilige Rechenart delegiert, dieser Aufruf ist auf Zeile (19) ersichtlich.

2.2.3.4 Der Klient

Der Klient wird in diesem Beispiel mit der gleichnamigen Klasse `Klient` implementiert:

```
(01) package strategy;
(02)
(03) public class Klient
(04) {
(05)   public static void main(String[] args)
(06)   {
(07)     Taschenrechner rechner = new Taschenrechner();
(08)     System.out.println(rechner.berechne(20, 15));
(09)
(10)     // Wechsel der Strategie
(11)     rechner.setRechenart(new Subtraktion());
(12)     System.out.println(rechner.berechne(20, 15));
(13)   }
(14) }
```

Die Klasse `Klient` enthält in diesem Fall die Methode `main()`, welche den Einstiegspunkt in unser Programm darstellt.

Auf Zeile (07) wird eine neue Instanz des Taschenrechners erstellt. Diese Instanz wird dann auf der Zeile (08) verwendet, hier soll eine Berechnung mit den beiden Zahlen 20 und 15 durchgeführt werden. Da der Konstruktor der Klasse `Taschenrechner` standardmässig die Addition als Berechnungsstrategie verwendet, lautet die Ausgabe 35.

Auf Zeile (11) wird diese Strategie jedoch verändert. Dazu wird eine neue Instanz der konkreten Strategie `Subtraktion` erstellt, welche dann dem Taschenrechner als neue Rechenart übergeben wird. Anschliessend wird erneut eine Berechnung mit den beiden Zahlen 20 und 15 durchgeführt. Da der Taschenrechner nun mit der Subtraktions-Strategie arbeitet, lautet das Ergebnis nun 5.

2.2.4 Vor- und Nachteile des Strategy-Musters

- Vorteile
  - Strategien stellen eine Alternative zur Bildung von Unterklassen dar.
    - ➔ Es wird Komposition und nicht Vererbung verwendet.
  - Die einzelnen Strategien verbessern die Wiederverwendung des Programmcodes.
  - Ähnliche Algorithmen werden in einer Familie von Algorithmen gruppiert.
  - Algorithmen können zur Laufzeit ausgetauscht werden.
    - ➔ Dies steigert die Flexibilität.
- Nachteile
  - Klienten müssen die unterschiedlichen Strategien kennen.
    - ➔ Dies stellt eine feste Bindung zwischen Klienten und konkreten Strategien dar.
  - Es müssen sehr viele (kleine) Klassen implementiert werden.

  - Die Anzahl der Objekte zur Laufzeit wird erhöht.

2.3 Observer

Beim *Observer*-Entwurfsmuster geht es darum, eine Anzahl von Objekten darauf aufmerksam zu machen, das ein bestimmtes Objekt geändert wurde. Dabei wird ein bestimmtes Objekt (das Subjekt) von einer variablen Anzahl von anderen Objekten (den Beobachtern) „beobachtet". Der Einsatz des Observer-Musters ist somit sinnvoll, wenn:

- eine Anzahl von Objekten über den Status eines Objekts auf dem Laufenden gehalten werden soll
- ein Objekt bei einer Änderung eines anderen Objekts benachrichtigt werden soll, ohne dieses im Detail zu kennen
  - Beispiel: Event-Handling (so verwendet bei Swing und SWT)
- eine MVC-Architektur (siehe Abschnitt 2.4, Zusammengesetzte Muster: MVC) realisiert werden soll

2.3.1 Aufbau

Grundsätzlich geht es im Observer-Muster um zwei Parteien; das Subjekt und den Beobachter (engl. Observer). Diese werden jeweils als Schnittstelle umgesetzt. Die Implementierung erfolgt dann in einem konkreten Subjekt bzw. in einem konkreten Beobachter.

Ein Subjekt enthält jeweils eine Liste mit Referenzen auf verschiedene Beobachter, letztere können sich dieser Liste selbständig hinzufügen oder sich von dieser Liste entfernen. Das Subjekt muss entsprechende Methoden für diesen Zweck zur Verfügung stellen. Die Beobachter können die Änderungen an einem Subjekt somit „abonnieren".

Ein Beobachter enthält eine Methode zur Aktualisierung. Ändert sich der Status des Subjekts, benachrichtigt dieses sämtliche Beobachter in der Liste.

Es gibt zwei Möglichkeiten, wie die Beobachter an den geänderten Wert eines Subjekts heran kommen können:

1. Das „Push"-Verfahren
   - Der Beobachter erwartet den geänderten Wert in seiner Aktualisierungs-Methode als Parameter.
   - Das Subjekt gibt dem Beobachter den geänderten Wert bei der Aktualisierung mit.
2. Das „Pull"-Verfahren
   - Der Beobachter erwartet eine Referenz auf das Subjekt in seiner Aktualisierungs-Methode.
   - Das Subjekt gibt eine Referenz auf sich selbst bei der Aktualisierung mit.
   - Weiter stellt das Subjekt eine Methode zur Verfügung, mit der die Beobachter den geänderten Wert vom Subjekt heraus lesen können.
   - Die Beobachter lesen den Wert mithilfe genannter Methode aus dem Subjekt heraus.

Das folgende UML2-Klassendiagramm beschreibt das Observer-Muster mit dem zweiten Verfahren, dem „Pull"-Verfahren:

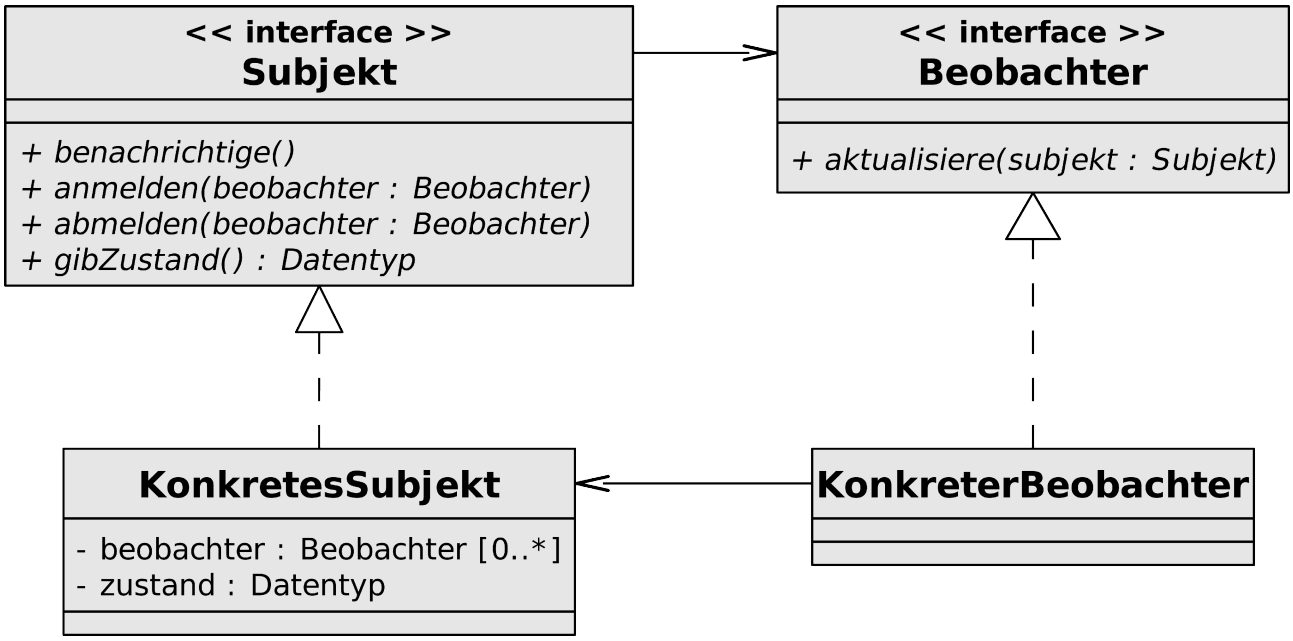

Abbildung 3: Das Observer-Entwurfsmuster (UML2-Klassendiagramm)

2.3.2 Definition

Das Observer-Muster *definiert eine Eins-zu-viele-Abhängigkeit zwischen Objekten in der Art, dass alle abhängigen Objekte benachrichtigt werden, wenn sich der Zustand des einen Objekts ändert.*

2.3.3 Implementierung

Das Observer-Muster soll anhand eines kleinen Beispiels veranschaulicht werden. Dabei geht es um eine Zeitungsredaktion, die Nachrichten entgegen nimmt und diese an ihre Abonnenten weiter verteilt. Ein Abonnent kann sich bei der Zeitungsredaktion registrieren lassen und erhält fortan sämtliche Nachrichten dieser Redaktion. Möchte er diese Nachrichten nicht mehr empfangen, so kann er sich von der Abonnenten-Liste streichen lassen.

2.3.3.1 Die Schnittstelle `Subjekt`

Die Schnittstelle `Subjekt` definiert fünf Methoden:

1. die Methode `registriere()`, womit ein Beobachter die Änderung des Subjekts abonnieren kann
2. die Methode `entferne()`, womit sich ein Beobachter aus der „Abonnenten-Liste" des Subjekts entfernen lassen kann
3. die Methode `benachrichtige()`, welche alle registrieren Beobachter über eine Änderung informiert
4. die Methode `zeigeNachricht()`, über welche die Beobachter bei der Aktualisierung den geänderten Wert herauslesen können
5. und die Methode `schreibeNachricht()`, womit ein Klient den Status des Subjekts von aussen ändern kann

Die Schnittstelle sieht dann folgendermassen aus:

```
(01) package observer;
(02)
(03) public interface Subjekt
(04) {
(05)   public void registriere(Beobachter beobachter);
(06)
(07)   public void entferne(Beobachter beobachter);
(08)
(09)   public void benachrichtige();
(10)
(11)   public String zeigeNachricht();
(12)
(13)   public void schreibeNachricht(String nachricht);
(14) }
```

Die zu versendenden Nachrichten bestehen jeweils aus einem einfachen String.

2.3.3.2 Die Schnittstelle `Beobachter`

Die Schnittstelle Beobachter besteht nur aus einer einzigen Methode, der Methode `aktualisiere()`.

```
(01) package observer;
(02)
(03) public interface Beobachter
(04) {
(05)   public void aktualisiere(Subjekt subjekt);
(06) }
```

In diesem Beispiel soll das „Pull"-Verfahren verwendet werden, der Beobachter erhält somit eine Referenz auf das Subjekt und liest den geänderten Wert dann mit der Methode `zeigeNachricht()` aus dem Subjekt heraus.

2.3.3.3 Die Klasse `Redaktion`

Die Klasse Redaktion stellt die eigentliche Implementierung des Subjekts dar.

- Auf den Zeilen (07) und (08) ist eine `ArrayList` für die Abonnenten des Subjekts definiert, diese hat die Startgrösse 0.
  - Auf Zeile (14) wird der Liste der Beobachter hinzugefügt, der als Parameter übergeben wurde.
  - Auf Zeile (19) wird der als Parameter übergebene Beobachter wieder aus der Liste entfernt.
- Auf Zeile (24) beginnt eine `foreach`-Schleife, welche über sämtliche registrierten Beobachter iteriert.
  - Jeder registrierte Beobachter wird auf Zeile (26) jeweils aktualisiert.
- Beim Aufruf der Methode `schreibeNachricht()` (Zeile (35)), wird die aktuelle Nachricht der Redaktion mit der übergebenen Nachricht überschrieben (Zeile (37)). Es folgt die Benachrichtigung sämtlicher Abonnenten auf Zeile (38).

```
(01) package observer;
(02)
(03) import java.util.ArrayList;
(04)
(05) public class Redaktion implements Subjekt
(06) {
(07)   private ArrayList<Beobachter> abonnenten = new
(08)     ArrayList<Beobachter>(0);
(09)
(10)   private String nachricht;
(11)
(12)   public void registriere(Beobachter beobachter)
(13)   {
(14)     abonnenten.add(beobachter);
(15)   }
(16)
(17)   public void entferne(Beobachter beobachter)
(18)   {
(19)     abonnenten.remove(beobachter);
(20)   }
(21)
(22)   public void benachrichtige()
(23)   {
(24)     for (Beobachter beobachter : abonnenten)
(25)     {
(26)       beobachter.aktualisiere(this);
(27)     }
(28)   }
(29)
(30)   public String zeigeNachricht()
(31)   {
(32)     return nachricht;
(33)   }
(34)
(35)   public void schreibeNachricht(String nachricht)
(36)   {
(37)     this.nachricht = nachricht;
(38)     benachrichtige();
(39)   }
(40) }
```

2.3.3.4 Die Klasse `Leser`

Die Klasse `Leser` stellt die Implementierung der Schnittstelle `Beobachter` dar.

```
(01) package observer;
(02)
(03) public class Leser implements Beobachter
(04) {
(05)   public void aktualisiere(Subjekt subjekt)
(06)   {
(07)     System.out.println(subjekt.zeigeNachricht());
(08)   }
(09) }
```

Wird der `Leser` auf eine Änderung des Subjekts aufmerksam gemacht (Aufruf der Methode `aktualisiere()` auf Zeile (05)), holt dieser die aktuelle Nachricht aus dem Subjekt heraus und gibt diese auf die Standardausgabe aus (Zeile (07)).

2.3.3.5 Die Klasse `Programm`

Die Klasse `Programm` demonstriert das Observer-Muster anhand eines einfachen Ablaufs.

```
(01) package observer;
(02)
(03) public class Programm
(04) {
(05)   public static void main(String[] args)
(06)   {
(07)     Subjekt nzz = new Redaktion();
(08)     Beobachter meier = new Leser();
(09)     nzz.registriere(meier);
(10)
(11)     nzz.schreibeNachricht("22.11.1963: John F. Kennedy ermordet");
(12)     nzz.schreibeNachricht("20.07.1969: Erste Mondlandung");
(13)
(14)     nzz.entferne(meier);
(15)
(16)     nzz.schreibeNachricht("26.12.1991: Untergang der Sowjetunion");
(17)   }
(18) }
```

Auf Zeile (07) wird ein neues `Subjekt`, die `Redaktion`, erstellt. Es folgt die Erstellung eines neuen `Beobachters` auf Zeile (08).

Dieser `Leser` wird dann auf Zeile (09) bei der `Redaktion` registriert, er „abonniert" die Nachrichten aus der `Redaktion`. Auf den Zeilen (11) und (12) werden nun Neuigkeiten an die `Redaktion` gemeldet. Diese werden sogleich an den `Leser meier` weitergegeben, es erfolgt die Ausgabe der Nachrichten auf die Standardausgabe (siehe Klasse `Leser`, Zeile (07)).

In Zeile (14) wird der `Leser meier` von der Abonnenten-Liste entfernt. Somit erhält er die Meldung von Zeile (16) nicht mehr.

2.3.4 Vor- und Nachteile des Observer-Musters

- Vorteile
  - Ein Subjekt kann mehrere Beobachter auf dem Laufenden halten.
  - Die Beobacher erhalten die Aktualisierungen automatisch.
    ➔ Es muss kein Polling verwendet werden.
  - Die Beobachter sind locker an das Subjekt gekoppelt.
    ➔ Beobachter können sich bei verschiedenen Subjekten registrieren.
- Nachteile
  - Bei einer grossen Anzahl von Beobachtern leidet die Performance.
  - Hat die Aktualisierung eines Beobachters eine Änderung des Subjekts zur Folge, kann eine Endlosschleife entstehen.
  - Ein Beobachter erhält eine Information, **dass** sich etwas geändert hat, er weiss jedoch nicht, **wie** die Änderung im Detail aussieht.

2.4 Zusammengesetzte Muster: MVC

Die Abkürzung *MVC* steht für *Model View Controller*. MVC stellt ein Architekturmuster zur Aufteilung eines Softwaresystems in drei Einheiten dar; Model, View und Controller.

Mithilfe der MVC-Architektur wird ein flexibles Programmdesign erreicht, die Aufteilung in drei verschiedene Einheiten fördert die Wiederverwendbarkeit der einzelnen Einheiten. Software, die nach der MVC-Architektur entworfen wurde, lässt sich sehr einfach erweitern. So kann man

beispielsweise eine neue View entwickeln und diese dann ohne jegliche Änderungen an Model oder Controller in die Software integrieren.

2.4.1 Komponenten des MVC

Verschiedene MVC-Architekturen können in einem gewissen Rahmen voneinander abweichen. Alle MVC-Architekturen haben jedoch etwas gemeinsam; sie bestehen immer aus den Komponenten Model, View und Controller.

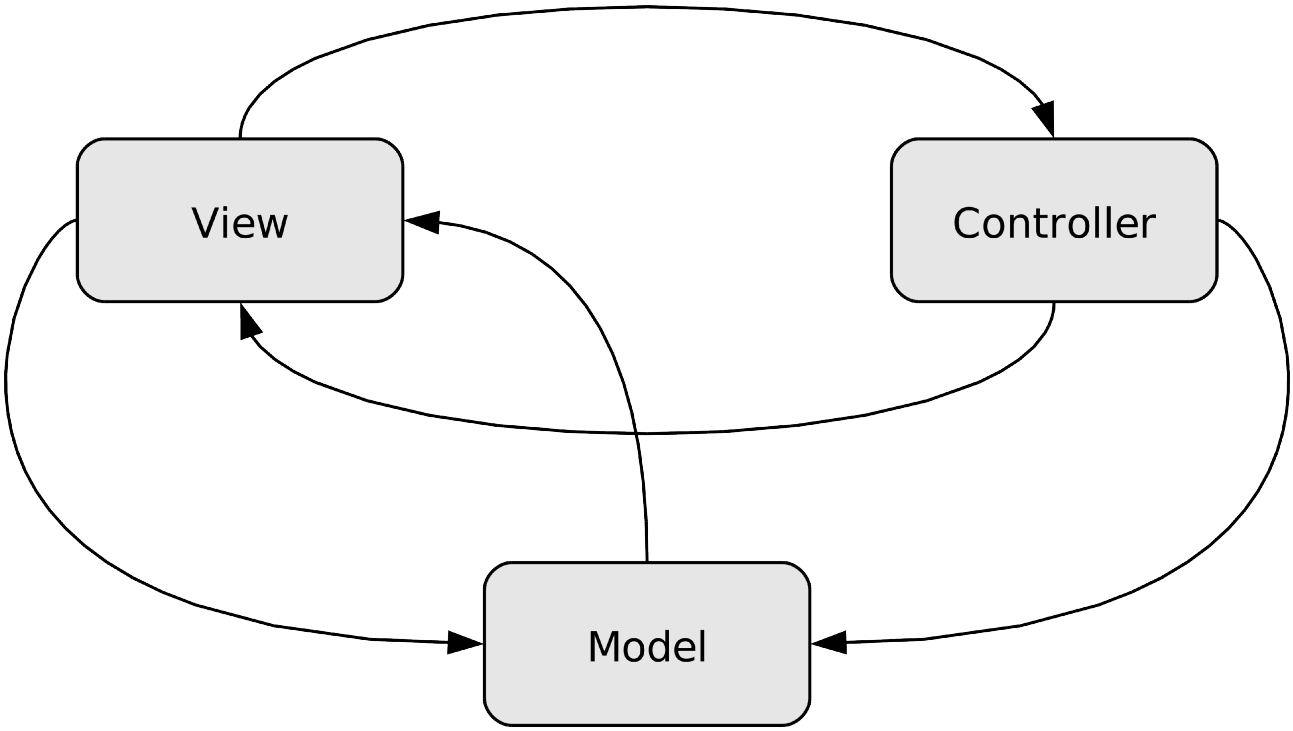

Abbildung 4: Komponenten einer MVC-Architektur

Erklärung der Komponenten:

- Model (Modell)
  - Das Model enthält die darzustellenden Daten. Für das Model spielt es keine Rolle, woher diese Daten kommen.
  - Dem Model ist nicht bekannt, wie die Präsentation der enthaltenen Daten zu erfolgen hat oder wie die Steuerung der Anwendung abläuft.
- View (Präsentation)
  - Die View stellt die Daten aus dem Model dar.
  - Ändern sich die Daten des Models, muss die View aktualisiert werden.
  - Interaktionen werden an den Controller weitergeleitet.
- Controller (Steuerung)
  - Der Controller ist für die Interaktion mit dem Benutzer zuständig.
  - Aktionen vom Benutzer werden entgegengenommen und ausgewertet.
  - Erfordert die Benutzeraktion eine Manipulation der Daten, so wird dies an das Model delegiert.
  - Der Controller steuert den Ablauf der Anwendung.

2.4.2 Beispiel

Zum besseren Verständnis der MVC-Architektur folgt nun ein kleines Beispiel, das die Interaktion zwischen den einzelnen Komponenten zur Laufzeit aufzeigt. Es soll ein Taschenrechner

nach der MVC-Architektur entworfen werden:

2.4.2.1 View

Es wird von einem Taschenrechner ausgegangen, die View wird dementsprechend durch eine grafische Oberfläche (GUI) in Form eines Taschenrechners dargestellt. Es existieren verschiedene Views:

- ein Taschenrechner für die Grundrechenarten (+, -, x, :)
- ein erweiterter Taschenrechner, der Wurzelfunktionen und die Potenzierung von Zahlen unterstützt
- ein wissenschaftlicher Taschenrechner, der Funktionen wie Sinus und Cosinuns unterstützt

Eine View kann die Daten auf unterschiedliche Art und Weise repräsentieren. So werden beim einfachsten Taschenrechner nur drei Nachkommastellen angegeben, der erweiterte Taschenrechner zeigt zehn Nachkommastellen an und der wissenschaftliche Taschenrechner verwendet immer die Exponentialdarstellung ($7.21 \cdot 10^3$).

2.4.2.2 Model

Die eigentliche Berechnung der Werte findet im Model statt. Dieses enthält sämtliche Funktionen zur Ausführung von Berechnungen, ob nun Grundrechenarten oder wissenschaftliche Funktionen verwendet werden sollen.

2.4.2.3 Controller

Der Controller hat die Aufgabe, die von der View gesendeten Aufgaben entsprechend an das Model weiterzuleiten und somit die Berechnungen in Auftrag zu geben.

Nebenbei „kontrolliert" der Controller auch die Benutzerschnittstellen (Views), so darf z.B. der Button mit dem Gleichheitszeichen (ein Klick auf diesen Button löst die Berechnung aus) nur angezeigt werden, wenn eine vollständige Rechnung in den Taschenrechner eingegeben wurde.

Möchte der Benutzer die Ansicht des Taschenrechners verändern (z.B. von einfach nach wissenschaftlich), muss der Controller die entsprechende View anzeigen.

2.4.2.4 Ablauf

Zum besseren Verständnis soll nun der Ablauf einer einfachen Berechnung aufgezeigt werden:

1. Der Benutzer gibt eine Rechnung in die View ein.
   - Sobald die Rechnung fertig formuliert ist, zeigt der Controller den Button mit dem Gleichheitszeichen an.
2. Der Benutzer möchte die Berechnung ausführen und klickt auf den Button mit dem Gleichheitszeichen.
3. Der Controller empfängt die Anfrage der View.
   - Es soll eine Berechnung ausgeführt werden, die eingegebenen Daten können aus der View entfernt werden.
   - Die Berechnung wird an das Model delegiert.
4. Das Model führt die Berechnung durch und aktualisiert die View mit dem entsprechenden Resultat.
5. Die View zeigt das Resultat an.

2.4.3 Verwendete Entwurfsmuster

MVC ist kein Entwurfsmuster im eigentlichen Sinne, es ist vielmehr ein allgemein gültiges Paradigma zum Entwurfs von Anwendungen mit einer grafischen Oberfläche. Für die Implementierung einer MVC-Architektur werden jedoch mehrere Entwurfsmuster eingesetzt, MVC kann so-

mit als „zusammengesetztes Muster" bezeichnet werden.

Ein zusammengesetztes Muster *kombiniert zwei oder mehrere Muster zu einer Lösung für ein wiederkehrendes oder allgemeines Problem.*

Eine MVC-Implementierung enthält in der Regel die folgenden Entwurfsmuster:

- Strategy
  - Zwischen View und Controller wird ein klassisches Strategy-Muster verwendet.
  - Eine View kann verschiedene Controller-Strategien verwenden.
  - Ein Controller kann verschiedene View-Strategien verwenden.
  - View und Controller sind locker aneinander gekoppelt.
- Observer
  - Das Model stellt das Subjekt dar.
  - Die einzelnen Views stellen die Beobachter dar.
  - Ein Model aktualisiert sämtliche Views, sobald sich die Daten geändert haben.
- Composite
  - Eine View besteht aus verschiedenen GUI-Elementen.
  - Diese GUI-Elemente werden in Gruppen (Composites) angeordnet.

Wie bereits erwähnt, können sich einzelne MVC-Implementierungen stark voneinander unterscheiden. So ist es durchaus möglich, dass für die Implementierung ein Entwurfsmuster weggelassen werden kann (z.B. Composite) oder dass noch weitere Entwurfsmuster eingesetzt werden.

2.5 Allgemeine Entwurfsprinzipien

Entwurfsmuster lösen eine bestimmte Klasse von Problemen. Neben den Entwurfsmustern gibt es auch allgemein gültige Entwurfsprinzipien für die objektorientierte Programmierung, welche nicht nur für ein bestimmtes Problem eingesetzt werden können, sondern bei jedem objektorientierten Entwurfs angewendet werden sollten. Einige dieser Entwurfsprinzipien lauten:

1. Identifizieren Sie die Aspekte Ihrer Anwendung, die sich ändern können und trennen Sie sie von denen, die konstant bleiben.
2. Programmieren Sie auf eine Schnittstelle, nicht auf eine Implementierung.
3. Ziehen Sie Komposition der Vererbung vor.
4. Streben Sie bei Entwürfen mit interagierenden Objekten nach lockerer Kopplung.
5. Klassen sollten für Erweiterungen offen, aber für Veränderung geschlossen sein.
6. Stützen Sie sich auf Abstraktionen und nicht auf konkrete Klassen.

Es existieren noch zahlreiche weitere Entwurfsprinzipien. Die hier genannten Prinzipien lassen sich jedoch am besten auf die in diesem Kapitel beschriebenen Entwurfsmuster (vor allem Strategy und Observer) anwenden.

3 Multitasking

Als PC-Benutzer ist man es sich gewohnt, dass mehrere Programme und Dienste **z**ur gleichen Zeit laufen. Für den Anwender ist dieses gleichzeitige Ausführen von Anwendungen eine Selbstverständlichkeit, für ein Betriebssystem stellt es eine Menge Arbeit dar.

Man unterscheidet zwischen zwei Arten, wie Programme gleichzeitig ausgeführt werden können:

1. Echt gleichzeitig
2. Scheinbar gleichzeitig

Ersteres ist dann der Fall, wenn der Rechner mit mehreren Prozessoren bestückt ist. Das Betriebssystem kann die Aufgaben so auf mehrere Prozessoren verteilen, die Aufgaben werden gleichzeitig von verschiedenen Prozessoren abgearbeitet. Steht dem Betriebssystem jedoch nur ein Prozessor zur Verfügung, so kann er diese nicht wirklich zur gleichen Zeit ausführen. In diesem Falle wird in sehr hoher Frequenz zwischen den Aufgaben umgeschaltet, sodass sich die Wahrnehmung der Parallelität einstellt.

3.1 Prozesse und Threads

In der nebenläufigen Programmierung unterscheiden wir zwischen den zwei Ausführungseinheiten *Prozess* und *Thread*.

Ein Prozess wird dann gestartet, wenn ein neues Programm ausgeführt werden soll. Jeder Prozess hat seinen eigenen Adressraum, sodass Prozesse sich nicht gegenseitig in die Quere kommen können. Das Betriebssystem stellt jedoch verschiedene Mechanismen zur Verfügung, damit Prozesse trotzdem miteinander kommunizieren können.

Ein Thread ist ein Programmfaden, also eine Sequenz von auszuführenden Befehlen. Ein Prozess kann mehrere Threads besitzen. Dabei teilen sich sämtliche Threads eines Prozesses den gleichen Adressraum (jeder Thread besitzt seinen eigenen Stack, sie verwenden jedoch alle den selben Heap).

Jeder Prozess und jeder Thread besitzt seine eigene ID. Dies ist eine Nummer, die den Prozess/Thread systemweit und eindeutig identifiziert. Das Betriebssystem hat die Aufgabe, die verschiedenen Prozesse und Threads zu verwalten und ihnen die notwendigen Ressourcen zur Verfügung zu stellen.

Da sich alle Threads eines Prozesses einen gemeinsamen Heap teilen, kann das Betriebssystem hier keine Schutzfunktion übernehmen. Dies ist die Aufgabe des Programmierers. Instanzen sind bekanntermassen auf dem Heap abgespeichert, somit können sie nicht einem bestimmten Thread zugeordnet werden. Threads besitzen jedoch eigene Stacks, von welchen aus Speicherobjekte referenziert werden können. Instanzen sind also nicht an einen Thread, sondern an einen Prozess gebunden. Kann ein Objekt von verschiedenen Threads verwendet werden, ohne dass es dabei zu Inkonsistenzen kommt, so wird das Objekt als *threadsicher* oder *threadsafe* bezeichnet, was ein erstrebenswerter Zustand ist.

3.2 Kontextwechsel

Die Zuteilung von Ressourcen, insbesondere der CPU, ist eine der wichtigsten Aufgaben eines jeden Betriebssystems. Dabei sollen alle Prozesse genügend Rechenzeit erhalten. Es gibt jedoch die Möglichkeit, Prozesse zu priorisieren. So erhalten Prozesse mehr oder weniger Rechenzeit bzw. erhält die Rechenzeit mehr oder weniger oft, mehr oder weniger schnell. Jeder Thread erhält ein bestimmtes Zeitintervall, in welcher er die CPU beanspruchen darf. Ist diese Zeit abgelaufen, so ist der nächste Thread an der Reihe und wird ausgeführt. Dieses Konzept nennt man *Kontextwechsel*.

Bei diesem Kontextwechsel wird entweder auf einen anderen Threads des selben Prozesses, oder auf einen Thread eines anderen Prozesses umgeschaltet. Im ersten Fall ist die Rede von einem Thread-Kontextwechsel. Wird zu einem Thread eines anderen Prozesses gewechselt, so muss auch der Prozess umgeschaltet werden. Diese Prozess-Kontextwechsel sind wesentlich aufwändiger, da auf einen anderen Adressraum umgeschaltet werden muss. Sämtliche Kon-

textwechsel werden vom *Scheduler* übernommen. Ein Scheduler entscheidet, welcher Thread zu welcher Zeit Ressourcen erhält. Dies geschieht unter der Berücksichtigung der Thread-Prioritäten.

3.3 Thread-Prioritäten

Prozesse und Threads sind prinzipiell gleichberechtigt, sodass sie alle genügend Ressourcen zu ihrer Ausführung erhalten. Damit wichtigere Aufgaben etwas schneller ausgeführt werden können als weniger wichtige, wurde das Konzept der *Prioritäten* eingeführt. So kann man einem Thread eine höhere oder eine tiefere Priorität zuweisen.

Threads werden vom Scheduler nach ihrer Priorität gruppiert. Nun werden die Threads der Gruppe mit der höchsten Priorität nacheinander abgearbeitet, bis entweder alle Threads beendet sind oder ein Thread einer nicht leeren Gruppe einer höheren Priorität auftaucht. Dann wird die Gruppe mit der nächst tieferen Priorität abgearbeitet. Tauchen Threads in einer höheren Prioritätsgruppe auf, so schaltet der Scheduler auf diese Gruppe um.

Innerhalb einer Prioritätsgruppe wird die CPU-Zeit gerecht aufgeteilt. Damit Threads mit einer höheren Priorität nicht die ganze CPU-Zeit in Anspruch nehmen können, kann der Scheduler z.B. einen gelaufenen Thread um eine Prioritätsstufe senken. Der Scheduler kann dabei keine bestimmte Ausführungsreihenfolge garantieren, dies ist Aufgabe des Programmierers.

Es gibt unterschiedliche Scheduling-Strategien mit, welche auf verschiedene Arten implementiert werden können. Auf diese Scheduling-Verfahren wird im folgenden Abschnitt etwas näher eingegangen.

3.4 Scheduling-Verfahren

Ein Scheduling-Verfahren kann unter Berücksichtigung der folgenden Kriterien bewertet werden:

- **Fairness**
  - Prozesse sollen einen gerechten Teil der Prozessorzeit erhalten
- **Effizienz**
  - Der Prozessor soll nach Möglichkeit immer vollständig ausgelastet sein
- **Antwortzeit**
  - Interaktive Prozesse sollen eine möglichst kurze Antwortzeit haben
- **Verweilzeit**
  - Die Wartezeit von Stapelaufträgen soll minimiert werden
- **Durchsatz**
  - Die Anzahl bearbeiteter Aufträge in einem Zeitintervall soll maximiert werden

Einige dieser genannten Kriterien sind offensichtlich widersprüchlich. So soll z.B. die Antwortzeit von interaktiven Prozessen und die Verweilzeit von Stapelaufträgen möglichst kurz gehalten werden. Hier hat der Scheduler die Aufgabe, einen optimalen Kompromiss zu finden. Wird eine Klasse von Prozessen bevorzugt, so wird automatisch eine andere benachteiligt. Dies liegt in der Natur der Sache, da immer gleich viel Ressourcen zur Verfügung stehen.

3.4.1 Prozess-Umschaltung

Computer besitzen eine elektronische Stoppuhr. Diese löst in regelmässigen Abständen ein Unterbrechungssignal aus. Das Intervall kann vom Betriebssystem bestimmt werden. Bei jedem Unterbrechungssignal erhält der Scheduler den Fokus und kann entscheiden, ob er den laufenden Prozess unterbrechen oder ihn noch ein wenig laufen lassen soll. Kann ein Scheduler einen laufenden Prozess unterbrechen und ihn zu einem späteren Zeitpunkt wieder aufnehmen, so ist die Rede von *präemptivem Scheduling*.

Wenn ein Scheduler nicht über den genannten Mechanismus verfügt, so ist die Rede von *nicht-präemptivem Scheduling*. Jeder Prozess wird also komplett abgearbeitet, bis der nächste an die Reihe kommt. Dies ist bei älteren Stapelsystemen der Fall.

3.4.2 Round-Robin-Verfahren

Des Round-Robin-Verfahren ist das älteste Scheduling-Verfahren und gilt gleichzeitig als das einfachste und fairste. Jeder Prozess erhält ein gewisses Zeitintervall, das *Quantum*, für seine Ausführung. Ist dieses Quantum abgelaufen, wird auf den nächsten Prozess gewechselt. Der Prozesswechsel kann auch bereits vor dem Ablauf des Quantums stattfinden, so kann ein blockierender Prozess vorzeitig unterbrochen werden.

Alle Prozesse stellen sich in eine Warteschlange (Queue) ein (**A**-B-C-D). Nun wird Prozess A ausgeführt. Ist sein Quantum abgelaufen, kommt Prozess B an die Reihe. Prozess A stellt sich nun wieder hinten an (B-C-D-**A**).

Jeder Prozesswechsel erfordert etwas Prozessorzeit, so muss z.B. bei der Prozessumschaltung immer auf einen anderen Speicherbereich gewechselt werden. Angenommen, das Quantum beträgt 50 Millisekunden und die Umschaltung nimmt 10 Millisekunden in Anspruch, so wird die Prozessorzeit zu 20% für die Prozessumschaltung verbraucht. Um dieses Verhältnis zu verbessern, könnte man das Quantum auf eine Sekunde erhöhen. Der Prozessor verwendet nun 99% für die Prozessausführung und nur noch 1% für die Umschaltung. Dies mag im ersten Moment sehr gut klingen. Vergessen wir aber nicht, was dies für Auswirkungen auf Interaktive Systeme hätte. Wenn ein Benutzer z.B. auf dem Terminal drei mal die Return-Taste betätigen würde, so vergingen geschlagene drei Sekunden, bis das letzte Return angekommen wäre. Eine unzumutbare Wartezeit. Bei der Bestimmung der Länge des Quantums gilt es also, einen **optimalen Kompromiss** zu finden.

3.4.3 Prioritäts-Scheduling

Das Round-Robin-Verfahren betrachtet alle Prozesse als gleich wichtig. Oftmals empfindet ein Benutzer jedoch einen Prozess als wichtiger als den anderen. Um dem gerecht zu werden, wurde das *Prioritäts-Scheduling* entwickelt.

Jeder Prozess erhält dabei eine Priorität. Der ausführbare Prozess mit der höchsten Priorität wird als erster ausgeführt. Damit priorisierte Prozesse nicht einfach die gesamte Prozessorzeit in Beschlag nehmen können, wird die Priorität eines Prozesses nach jedem Durchlauf verringert. So werden bei der Ausführung trotzdem alle Prozesse berücksichtigt.

Diese Prioritäten können *statisch* oder *dynamisch* zugewiesen werden. Bei der statischen Zuweisung wird einfach ein fixer Wert vorgegeben. Die dynamische Priorisierung betrachtet, wie stark ein Prozess sein Quantum ausnützt. Ein Prozess, der sein Quantum jeweils vollständig ausnutzt, erhält eine tiefere Priorität. Prozesse, die z.B. viel I/O-Operationen ausführen, benötigen anteilsmässig weniger von ihrem Quantum. Dafür sollen sie aber auch schneller reagieren können (interaktive Verarbeitung). Letztere werden also eher ausgeführt, damit der Anwender "flüssig" arbeiten kann.

Prozesse können auch in Prioritätsgruppen eingeteilt werden. Innerhalb dieser Prioritätsgruppen kommt häufig das Round-Robin-Verfahren zum Einsatz. Auch hier müssen die Prioritäten angepasst werden. Ein Prozess wird so nach jeder Ausführung in eine tiefere Prioritätsgruppe verschoben.

3.4.4 Mehrere Schlangen

Ein Prozesswechsel benötigt bekanntermassen einen nicht unwesentlich grossen Teil der Prozessorzeit. Wie wir auch gesehen haben, gibt es Prozesse die ihre Quanten nicht vollständig ausnützen, andere Prozesse lasten ihre Quanten um ein Mehrfaches aus. Um in diesem Falle ein unnötig häufiges Umschalten zwischen Prozessen zu verhindern, kann man den zeitlich anspruchsvolleren Prozessen grössere Quanten zuweisen.

Die Prozesse werden in verschiedene Prioritätsklassen gruppiert. Die Gruppe mit der höchsten Priorität erhält kleinere Quanten, dafür werden sie auch häufiger ausgeführt. Nach jeder Ausführung wandert der Prozess eine Stufe nach unten. Je weiter unten sich ein Prozess befindet, desto seltener wird er ausgeführt und desto grössere Quanten erhält er. Auf diese Weise kann die Anzahl der Kontextwechsel reduziert werden, es wird insgesamt Prozessorzeit gespart. So können interaktive Prozesse oft und kurz ausgeführt werden, länger dauernde Prozessen werden selten, dafür aber auch für eine längere Zeit ausgeführt.

3.4.5 Shortest-Job-First

Bisher stand der Fokus immer auf interaktiven Systemen. Shortest-Job-First ist jedoch speziell für Stapelaufträge geeignet, deren Ausführungszeit bereits am Anfang bekannt ist.

Gehen wir von den Prozessen A, B, C und D aus. A wird 10 Minuten für die Ausführung beanspruchen. Die Prozesse B, C und D benötigen nur jeweils 2 Minuten. Würden die Prozesse in dieser Reihenfolge ausgeführt, betrüge die Verweildauer für Prozess A 10 Minuten, für B 12 Minuten, für C 14 Minuten und für D 16 Minuten. Dies gibt eine durchschnittliche Verweildauer von 13 Minuten. Führt man jedoch die kurzen Prozesse zuerst aus und stellt A ganz nach hinten, so würde die Verweildauer für B 2 Minuten, für C 4 Minuten, für D 6 Minuten und für A 16 Minuten betragen. Dies führt zu einer durchschnittlichen Verweildauer von 7 Minuten. So kann man in einer kurzen Zeit viele Prozesse abarbeiten, einzig ein langwieriger Prozess würde etwas länger auf sich warten lassen.

3.4.6 Zweistufiges Scheduling

Manchmal ist es nicht möglich, alle Prozesse im Arbeitsspeicher zu halten. So müssen Prozesse auf dem Hintergrundspeicher (Festplatte) abgelegt werden. Das holen eines Prozesses von der Festplatte dauert wesentlich länger als das wechseln zwischen zwei Prozessen innerhalb des Arbeitsspeichers. Hier werden also zwei Scheduler benötigt, welche sich die Arbeit teilen.

Der übergeordnete Scheduler entscheidet darüber, wann Prozesse vom Arbeitsspeicher auf die Festplatte und von der Festplatte in den Arbeitsspeicher kommen. Der untergeordnete Scheduler kann nach einem beliebigen, bereits vorgestellten, Verfahren arbeiten und wechselt nur die Prozesse innerhalb des Arbeitsspeichers aus.

3.5 Thread-Zustände

Threads können verschiedene Zustände einnehmen. Dabei hat jeder Thread zu jedem Zeitpunkt exakt einen Zustand. Pro Prozessor kann immer nur ein Thread gleichzeitig ausgeführt werden.

- **Initialized**
  - Der Thread wurde initialisiert, jedoch noch nicht gestartet.
- **Ready**
  - Der Thread ist bereit zur Ausführung und wartet nur noch darauf, dass er Rechenzeit zugeteilt bekommt. Threads des Zustands Ready sind nach ihrer Priorität gruppiert.
- **Standby**
  - Dieser Thread wird gleich Prozessor-Ressourcen erhalten und somit ausgeführt. Nur ein Thread pro Prozessor kann diesen Status besitzen.
- **Running**
  - Dieser Thread wird im Moment abgearbeitet, pro Prozessor kann dies genau ein Thread sein.
- **Wait/Transition**
  - Threads in diesem Zustand wären eigentlich bereit zur Ausführung (Status Ready), müssen jedoch noch ein bestimmtes Ereignis oder eine bestimmte nicht-Prozessor-Ressource abwarten. Sobald diese Ressourcen frei sind oder das Ereignis eingetreten ist, wechseln sie zurück zum Status Ready.
- **Terminated**
  - Der Thread wurde abgearbeitet und ist nun beendet.

Für die Zuteilung der CPU-Ressource werden nur Threads der Zustände Ready, Standby und Running berücksichtigt. Sämtliche Prozesse müssen zunächst den Zustand Ready erreichen, bevor sie ausgeführt werden können. Die anderen Zustände sind für den Programmierer nicht weiter interessant.

3.5.1 Wechsel von Thread-Zuständen

Ein Thread kann von einem Zustand immer nur in bestimmte andere Zustände übergehen. So muss ein Thread zunächst die Zustände Initialized, Ready, Standby und Running durchlaufen haben, bevor er abgeschlossen (also Terminated) sein kann. Dazu ein Beispiel:

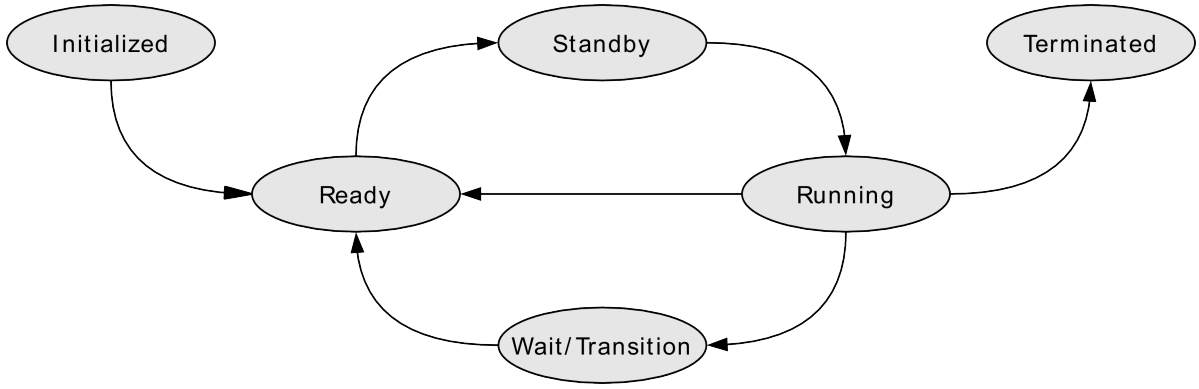

Abbildung 5: Wechsel von Thread-Zuständen

1. Wir initialisieren einen neuen Thread. Dieser hat nun den Status **Initialized**.
2. Nun geben wir diesen Thread zur Ausführung frei, sodass er den Status **Ready** besitzt.
3. Der Scheduler meldet, dass unser Thread als nächstes gestartet werden kann, er wechselt zum Status **Standby**.
4. Die CPU ist nun frei, unser Thread erhält den Zustand **Running** und wird somit ausgeführt.
5. Nun ist die CPU-Zeit schon wieder abgelaufen, unser Thread wurde aber noch nicht ganz abgearbeitet. Er muss sich also wieder hinten anstellen und erhält so wieder den Status **Ready**.
6. Einige Threads werden abgearbeitet, sodass unser Thread als nächstes an die Reihe kommt - Status **Standby**.
7. Unser Thread wird nun wieder ausgeführt und erhält den Status **Running**
8. Wir warten auf eine bestimmte Ressource/auf ein bestimmtes Ereignis, sodass unser Thread nicht mehr arbeiten kann. Der Thread wird zurückgestellt und erhält den Zustand **Wait/Transition**.
9. Das besagte Ereignis ist eingetreten, der Scheduler weckt unseren Thread wieder und vergibt ihm erneut den Status **Ready**.
10. Unser Thread durchläuft erneut die Stati **Standby** und **Running**.
11. Dieses mal hat die CPU-Zeit ausgereicht, um alle Aktionen durchzuführen. Der Thread ist abgearbeitet und erhält somit den Status **Terminated**

Was hier nach einer mühsamen und langen Prozedur aussieht, geschieht auf modernen Rechnern mehrmals pro Sekunde!

3.6 Threads programmieren

Die hier geschilderte Theorie des Multitaskings hört sich zugegebenermassen sehr kompliziert an. In der Praxis ist die Implementierung einer Applikation mit mehreren Threads nicht ganz so kompliziert, da moderne Umgebungen wie Java und .Net die Implementierung von Threads stark vereinfachen. Eine Multithread-Anwendung ist nichts desto trotz komplizierter in der Entwicklung und vor allem in der Wartung als eine Anwendung, die nur einen Thread benötigt. Es gilt also:

Eine Anwendung sollte nur mit mehreren Threads entwickelt werden, wenn das Problem nicht auch mit einem einzigen Thread gelöst werden kann.

Es gibt jedoch einige Gründe, warum eine Anwendung mit mehreren Threads entwickelt werden soll:

1. **Antwortverhalten**
   - Eine Benutzeroberfläche soll auch bedienbar bleiben, wenn im Hintergrund gerade eine rechenintensive Operation durchgeführt wird. Hintergrundaktivitäten müssen so in eigene Threads ausgelagert werden. Hat der Benutzer z.B. eine rechenintensive Suche gestartet, so soll er diese auch wieder abbrechen können. Dazu muss die Benutzeroberfläche ständig bedienbar sein.
   - Bei einer Client-Server-Anwendung läuft die Kommunikation oftmals asynchron ab. Stellt der Benutzer eine Anfrage an den Server, so soll dieser mit seinem Client weiter arbeiten können. Die Anforderung wird also von einem separaten Thread an den Server gesendet, dieser Thread nimmt dann die Antwort vom Server zu einem späteren Zeitpunkt wieder entgegen.
2. **Performance**
   - Verteilt man Programmlogik auf mehrere Threads, so wirkt sich dies grundsätzlich negativ auf die Performance aus. Dies ist damit zu begründen, dass die Verwaltung von mehreren Threads einen gewissen Overhead mit sich zieht. Auch wenn die Logik durch mehrere Threads ausgeführt wird, die Rechenkapazität des Rechners bleibt gleich.
   - Verfügt man über mehrere Prozessoren, so wirkt sich die Verteilung der Programmlogik auf mehrere Threads positiv aus. Das Programm wird nicht mehr von einem einzigen Prozessor sequenziell ausgeführt, sondern parallel von verschiedenen Prozessoren. So kann man z.B. komplexe Algorithmen oder Berechnungen auf mehrere Prozessoren verteilen.
3. **Klarheit des Programmcodes**
   - Unter der Verwendung von mehreren Threads wird die Anwendung komplexer, der Quellcode ist schwieriger zu lesen als bei einem einzigen Thread.
   - Dennoch kann es bei gewissen Anwendungen zu mehr Klarheit führen, wenn man einen Thread als eine bestimmte Aufgabe betrachtet.

Wichtig sind vor allem die ersten beiden Gründe (Antwortverhalten und Performance). Programmcode wird durch die Verwendung von mehreren Threads in der Regel komplexer und nur in den wenigsten Fällen klarer.

4 Mehrschichtige Architektur

Früher wurde Software monolithisch entworfen, von Datenzugriff bis Benutzeroberflächen war alles fest miteinander verzahnt. Mit dem verstärkten Aufkommen der Client-Server-Architektur und objektorientierten Programmiersprachen verschwand das monolithische Entwurfsparadigma aber immer mehr aus der Softwareentwicklung. Software wird heutzutage *mehrschichtig* entworfen.

Für die spätere Wartbarkeit der zu entwickelnden Software ist ein guter Entwurf entscheidend, dieser charakterisiert sich durch die folgenden Eigenschaften:

- Systematisches Treffen von grundlegenden Entwurfsentscheidungen
- Realisierung einer drei- oder mehrschichtigen Architektur
- Einhalten von bewährten Entwurfs-Heuristiken und Standards
- Systematischer Einsatz von Entwurfsmustern
- Verwendung geeigneter Frameworks und Bibliotheken

4.1 Entwurfsentscheidungen

Bevor eine Software entworfen und später auch implementiert werden kann, gilt es einige grundlegende Entwurfsentscheidungen zu treffen. Dazu gehören unter anderem:

- Die Wahl der **Plattform**
  - Microsoft Windows Plattform
  - Sun's Java-Plattform
  - Eine Unix-artige Plattform (Linux, Solaris, BSD)
- Die Wahl der passenden (objektorientierten) **Programmiersprache**
  - Java
  - C#
  - C++
- Die Wahl des GUI-**Frameworks**
  - WindowsForms oder Windows Presentation Foundation für .Net
  - Eclipse RCP, SWT oder Swing für Java
  - Web-Oberfläche
- Die Wahl der **Datenhaltung**
  - Relationale oder objektrelationale Datenbank für grössere Datenmengen
    - ➔ Ermöglicht den gleichzeitigen Zugriff für mehrere Benutzer
    - ➔ Die Daten werden redundanzarm gespeichert
    - ➔ Die Wahrung der Datenkonsistenz wird auch nach Systemfehlern gewährleistet
    - ➔ Ermöglicht die zentrale Verwaltung der Zugriffsrechte
    - ➔ Ermöglicht die sofortige Ausführung von Abfragen
  - Objektserialisierung oder XML-Dateien für kleinere Datenmengen
- **Verteilung** der Anwendung auf verschiedene Systeme
  - Client
  - Server

4.2 Schichten-Architektur

Software wird heute mehrschichtig entworfen und implementiert. Im Folgenden werden einige grundlegenden Schichten-Architekturen erläutert.

4.2.1 Die Drei-Schichten-Architektur

Eine *Drei-Schichten-Architektur* (three-tier architecture) besteht aus den folgenden Schichten:

1. Einer **GUI-Schicht**
   - Benutzeroberfläche einer Anwendung
   - Dialogführung
   - Präsentation der Daten
   - Greift auf die Fachkonzeptschicht zu
2. Einer **Fachkonzeptschicht**
   - Funktionaler Kern der Anwendung
   - Greift auf die Datenhaltungsschicht zu
3. Einer **Datenhaltungsschicht**
   - Realisiert die jeweilige Form der Datenhaltung

Die Drei-Schichten-Architektur kann in zwei Ausprägungen auftreten:

1. Eine **strenge** Drei-Schichten-Architektur
   - Die GUI-Schicht darf nur auf die Fachkonzeptschicht zugreifen
     ➔ Bessere Wartbarkeit, Änderbarkeit und Portablilität
2. Eine **flexible** Drei-Schichten-Architektur
   - Die GUI-Schicht darf auch auf die Datenhaltungsschicht zugreifen
     ➔ Bessere Performance

Abbildung 6 zeigt eine strenge Drei-Schichten-Architektur, Abbildung 7 zeigt die flexible Version. Die Pfeile stellen die möglichen Zugriffe von der einen Schicht auf eine andere dar.

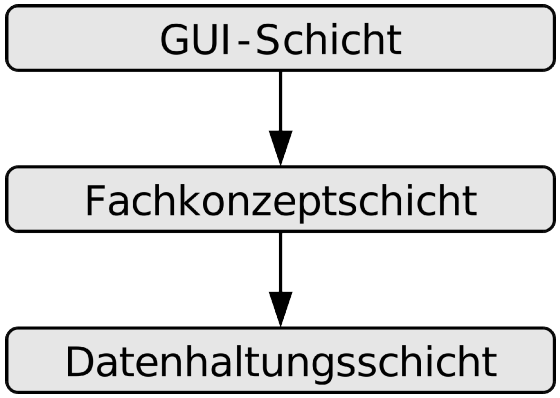

Abbildung 6: Strenge 3-Schichten-Architektur

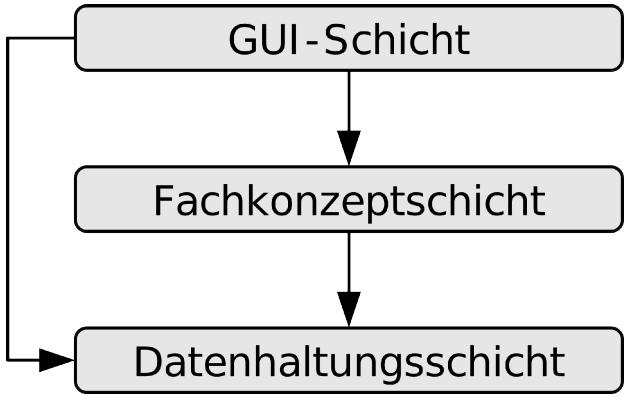

Abbildung 7: Flexible 3-Schichten-Architektur

In beiden Fällen darf eine übergeordnete Schicht nur auf untergeordnete Schichten zugreifen. Das bedeutet, dass die Fachkonzept nie direkt auf die GUI-Schicht zugreifen darf. Nur durch diese strikte Trennung ist es überhaupt möglich, dass die GUI- und die Fachoknzeptschicht getrennt voneinander entwickelt und zu einem späteren Zeitpunkt einfacher ausgetauscht werden können.

Damit die GUI-Schicht aber dennoch an die Daten der Fachkonzeptschicht gelangen kann, gibt es folgende Möglichkeiten:

1. **Polling**
   - Die GUI-Schicht setzt ständig in gleichen Intervallen Abfragen an die Fachkonzeptschicht ab, um diese nach Änderungen abzufragen. Dieses Verfahren wirkt sich negativ auf die Performance aus, da unnötig viele Anfragen gestellt werden.
2. **Indirekte Kommunikation**
   - Die Verbindung zwischen GUI- und Fachkonzeptschicht kann mit dem Beobachter-Muster (observer pattern) realisiert werden. Dabei sind GUI- und Fachkonzeptschicht locker (technisch gesehen über ein Interface) miteinander gekoppelt, die einzelnen Schichten können nach wie vor einfach ausgetauscht werden.

Werden GUI- und Fachkonzeptschicht zu einer einzigen Schicht zusammengeführt, so ist die Rede von einer Zwei-Schichten-Architektur. Der Nachteil daran ist, dass die beiden Schichten fest miteinander verbunden sind und ein Austausch der Benutzeroberfläche oder des Fachkonzepts sehr mühsam durchzuführen ist. Diese Zwei-Schichten-Architektur wird heute in der Praxis kaum noch angewendet.

4.2.2 Zugriffsschichten

Bei grösseren und komplexeren Anwendungen lohnt es sich oftmals, wenn der Zugriff auf eine tiefer liegende Schicht von einer weiteren Zwischenschicht übernommen wird. Hierbei ist die Rede von sog. *Zugriffsschichten*.

Erweitert man eine Drei-Schichten-Architektur um Zugriffsschichten, so ist die Rede von einer *Mehr-Schichten-Architektur* (multi-tier-Architektur).

4.2.2.1 Zugriffsschicht auf das Fachkonzept

In einer Drei-Schichten-Architektur führt die GUI-Schicht streng genommen zwei Aufgaben aus:

1. **Präsentation** der Informationen
2. **Kommunikation** mit der Fachkonzeptschicht

Um diesem Sachverhalt auch im Entwurf gerecht zu werden, kann aus der GUI-Schicht eine weitere Schicht extrahiert werden; die Fachkonzept-Zugriffsschicht. Dabei kümmert sich die GUI-Schicht nur noch um die reine Präsentation der Daten, der Zugriff auf die Fachkonzeptschicht wird von der neuen Fachkonzept-Zugriffsschicht übernommen.

Auf diese Weise können die komplexeren Daten-Beziehungen innerhalb der Fachkonzept-Schicht für die GUI-Schicht verborgen werden. Zudem sind GUI und Fachkonzept so noch weniger aneinander gekoppelt, ein Austausch der GUI-Schicht würde in diesem Fall leichter fallen.

4.2.2.2 Zugriffsschicht auf die Datenhaltung

Die Kernaufgabe der Fachkonzept-Schicht ist die Ausführung der eigentlichen Geschäftslogik. Die Fachkonzept-Schicht muss sich jedoch auch um die Kommunikation mit der Datenhaltungsschicht kümmern. Soll die Datenhaltungsschicht ausgetauscht werden, so kann diese enge Kopplung zu Problemen führen.

Die Lösung dieses Problems liegt darin, dass aus der Fachkonzept-Schicht eine Datenhaltungs-Zugriffsschicht extrahiert wird. Diese kümmert sich um den Zugriff auf die Datenhaltung, die Fachkonzept-Schicht ist so noch lockerer mit der Datenhaltungsschicht gekoppelt, letztere kann somit leichter ausgetauscht werden. Die Fachkonzeptschicht kann die Daten somit in einer höheren Abstraktion verarbeiten, die Geschäftslogik wird weiter von technischen Details isoliert.

4.2.3 Weitere Architekturen

Es gibt eine Unzahl von weiteren Architekturen, den meisten davon liegt aber das Konzept aus der Drei-Schichten-Architektur zu Grunde. Mit zunehmender Grösse kann eine Anwendung immer weiter in Schichten aufgeteilt werden. Ansonsten würden die Schichten unübersichtlich gross werden.

Man sollte es mit der Anzahl der Schichten jedoch auch nicht übertreiben, ansonsten kann auf Dauer die Orientierung verloren gehen. Zudem verschlechtert sich die Performance mit jeder weiteren Schicht, eine Anfrage muss schliesslich jede einzelne Schicht passieren. Bei einfachen Anfragen wirken die einzelnen Schichten so als blosser „Durchlaufserhitzer".

Hier geht es also darum, einen guten Kompromiss zu finden.

4.3 Entwurfsziele

Ob die gewählte Schichten-Architektur optimal für die jeweilige Anwendung ist, kann anhand folgender Entwurfsziele überprüft werden:

- **Wiederverwendbarkeit**
  - Jede Schicht besitzt eine präzise Schnittstelle.
  - Die Kommunikation mit dieser Schicht soll einzig und allein über diese Schnittstelle funktionieren können.
  - Eine Schicht kann einfach in eine andere Anwendung integriert werden.
- **Änderbarkeit/Wartbarkeit**
  - Die internen Mechanismen einer Schicht sollen beliebig stark angepasst werden können, gegen aussen muss die Schicht aber immer das gleiche Verhalten aufweisen.
  - Dies kann nur durch lockere Kopplung und präzise definierte Schnittstellen erfolgen.
- **Portabilität**
  - Eine Schicht soll Daten für die darüber liegende Schicht so gut wie möglich abstrahieren.
  - Auch dieses Ziel kann nur unter der Verwendung von klar definierten und strikt eingehaltenen Schnittstellen erreicht werden.

Insgesamt soll also die Kommunikation zwischen den einzelnen Schichten auf ein Minimum reduziert werden. Diese Kommunikation muss zudem über klar definierte und möglichst schlanke Schnittstellen erfolgen.

4.4 Entwurfsheuristiken

Bei einer sog. *Heuristik* handelt es sich um eine allgemein gültige Vorgehensweise zur Lösung eines bestimmten Problems. Trifft man in der Softwareentwicklung auf ein Problem, so ist in der Regel schon ein anderer Entwickler auf ein vergleichbares Problem gestossen. Aus der Lösung dieser Probleme hat sich über die Jahre ein gewisser Erfahrungsschatz angesammelt.

Auch für den Entwurf einer Anwendung gibt es bestimmte Heuristiken. Wer einen qualitativ hochwertigen Softwareentwurf anstrebt, der sollte die folgenden Heuristiken berücksichtigen:

- Sichtbarkeit
  - Das Geheimnisprinzip realisieren
    - ➔ Öffentliche Attributen können überall benutzt werden.
    - ➔ Was benutzt werden kann, das wird auch benutzt.
    - ➔ Attribute sollen privat gehalten werden!
  - Das Verhalten von internen Operationen sollen verborgen werden
    - ➔ Die Schnittstelle soll möglichst schlank gehalten werden.
    - ➔ Gemeinsame Teilfunktionen sollen in private Operationen ausgelagert werden.
- Vererbung und Generalisierung
  - Generalisierungshierarchien flach halten
    - ➔ Bei einer tiefen Generalisierungshierarchie verschwindet schnell der Überblick.

- ➔ Die Generalisierungshierarchie sollte nicht tiefer als sechs Ebenen sein.

- Oberklassen sollen abstrakt sein
  - ➔ Konkrete Klassen sollen sich nur am Ende der Hierarchie befinden.
  - ➔ Mit der Verwendung von abstrakten Klassen und Interfaces bleibt man flexibler.
- Gemeinsamkeiten so hoch als möglich in der Hierarchie einordnen
  - ➔ Attribute und Operationen, die in einer Oberklasse eingetragen sind, können in sämtlichen Unterklassen verwendet werden.
  - ➔ Man erspart sich viele gleichartige Änderungen.

- Interaktion, Kopplung und Bindung
  - Die GUI-Schicht greift auf die Fachkonzeptschicht zu, aber nicht umgekehrt
    - ➔ Die GUI-Schicht ändert sich wahrscheinlicher, als die Fachkonzeptschicht.
    - ➔ Hier ist lockere Kopplung (z.B. durch das Beobachter-Muster) zu verwenden!
  - Minimierung der Kopplung zwischen den Klassen
    - ➔ Eine Klasse A soll nur die öffentlichen Elemente einer Klasse B verwenden.
  - Maximierung der Bindung innerhalb einer Klasse
    - ➔ Eine Klasse realisiert genau eine Art von Objekten.
  - Vermeidung der Switch-Anweisung
    - ➔ Switch-Anweisungen sind ein Indiz für fehlenden Polymorphismus.
    - ➔ Die Verwendung des Zustand-Musters schafft hier Abhilfe.

Referenzen

- Entwurfsmuster von Kopf bis Fuss
  - Autor: Eric und Elisabeth Freeman
  - Verlag: O'Reilly
  - ISBN: 3-89721421-0
- Entwurfsmuster – Elemente wiederverwendbarer objektorientierter Software
  - Autoren: „Gang of Four"
    - ➔ Erich Gamma
    - ➔ Richard Helm
    - ➔ Ralph Johnson
    - ➔ John Vlissides
  - Verlag: Addison-Wesley
  - ISBN: 3-8273-2199-9
- Parallele Programmierung mit Java Threads
  - Autor: Rainer Oechsle
  - Verlag: Fachbuchverlag Leipzig
  - ISBN: 3-446-21780-0
- Moderne Betriebssysteme
  - Autor: Andrew S. Tannenbaum
  - Verlag: Pearson Studium
  - ISBN: 3-8273-7019-1
- Lehrbuch der Objektmodellierung
  - Autorin: Heide Balzert
  - Verlag: Spektrum Akademischer Verlag
  - ISBN-13: 978-3827402851
- Wikipedia
  - Objektorientierte Programmierung
    - ➔ http://de.wikipedia.org/wiki/Objektorientierte_Programmierung
  - Entwurfsmuster
    - ➔ http://de.wikipedia.org/wiki/Entwurfsmuster
  - MVC
    - ➔ http://de.wikipedia.org/wiki/MVC
  - Multitasking
    - ➔ http://de.wikipedia.org/wiki/Multitasking
  - Threads
    - ➔ http://de.wikipedia.org/wiki/Thread_%28Informatik%29

Abbildungsverzeichnis

Abbildung 1: Das Singleton-Entwurfsmuster (UML2-Klassendiagramm).....8
Abbildung 2: Das Strategy-Entwurfsmuster (UML2-Klassendiagramm)..... 12
Abbildung 3: Das Observer-Entwurfsmuster (UML2-Klassendiagramm)..... 16
Abbildung 4: Komponenten einer MVC-Architektur.....20
Abbildung 5: Wechsel von Thread-Zuständen.....27
Abbildung 6: Strenge 3-Schichten-Architektur.....30
Abbildung 7: Flexible 3-Schichten-Architektur.....30