Ravensburg-Weingarten
University of Applied Science

Fakultät Elektrotechnik
und
Informatik

SS20

Projektarbeit im Rahmen des Informatik Seminars

JavaScript und Python Entwurfsmuster

Singleton, Adapter, Observer

Autor:
Dennis Erdelean
26806

Professor:
Prof. Dr. rer. nat. Martin
Zeller

27. Mai 2020

Inhaltsverzeichnis

1 Einleitung

Durch die wachsende Digitalisierung entstehen mehr Softwareprojekte, es wird mehr Quellcode generiert und der Wettbewerb wird anspruchsvoller. Das bedeutet, dass heutzutage die Effizienz beim Entwickeln von Software essentieller denn je ist.

Das Entwerfen von objektorientiertem Quelltext ist eine recht abstrakte und kniffelige Aufgabe, insbesonders bei größeren Softwareprojekten an denen mehrere Entwickler arbeiten. Möchte man den Programmcode dann wiederverwenden, sollte dieser gut verständlich sein.

Genau wie bei einem Hausbau oder im Allgemeinen der Architektur bestimmter Dinge, steht man meistens vor wiederkehrenden Problemen. Es entsteht sozusagen ein Muster an Problemen. Das bedeutet im Umkehrschluss, dass diese Probleme in der Vergangenheit schon öfter gelöst wurden. Über die Zeit hinweg kamen daher viele Lösungen erfahrener Programmierer, für genau diese bekannten Probleme zusammen. Aus diesen Lösungen kristallisierten sich die besten, welche wir heute als Design Patterns, zu deutsch, Entwurfsmuster kennen. Sie sollen beim Entwicklen von Quelltext helfen, den Programmcode besser zu strukturieren und die Wiederverwendbarkeit zu pflegen.

Schaut man sich die Veränderung der letzten Jahre hinsichtlich der Verwendung von Programmiersprachen an, gewinnen die Programmiersprachen JavaScript und Python immer mehr an Beliebtheit und sind kaum wegzudenken. JavaScript ist eine der meistverwendsten Programmiersprachen.[1][2] Mittlerweile wird sie mit dem populären Framework Node.js[3] auch in der serverseitigen Entwicklung genutzt.

Python ist ein mächtiger Allrounder, der mit seiner verständlichen Syntax den Einstieg in die Programmierung vereinfacht. Sie schafft es im Vergleich zu anderen Programmiersprachen mit viel weniger Zeilen an Code auszukommen.

Sehr viele moderne und innovative IT-Unternehmen nutzen Python und JavaScript in der Entwicklung ihrer Websites und Web Services. Häufig ist auch die Kombination aus Python im Back-End und JavaScript im Front-End zu sehen. Besitzt man gute Kenntisse dieser beiden Sprachen, ist man in der Lage sehr kreative und perfomante Projekte umzusetzen.

Diese Projektarbeit soll jeweils 3 Design Patterns, die es sowohl in der Programmiersprache Python, als auch in JavaScript gibt, beleuchten. Hauptfokus liegt auf 3 wichtige Entwurfsmuster, dem

Singleton, *Adapter* und *Observer*.

1.1 Zielgruppe

An wen wendet sich diese Projektarbeit ?
Addressiert werden Programmierer mit soliden Grundkenntnissen in mindestens einer der beiden Sprachen, JavaScript oder Python. Es werden grundlegende Konzepte vorausgesetzt wie, Schleifen, Bedingte Anweisungen und Verzweigungen, Gültigkeitsbereiche, Objekte, Funktionen und insbesondere objektorientierte Konzepte.

1.2 Struktur

Beginnend mit der Motivation und dem eigentlichen Hauptziel dieser Projektarbeit, wird dann in den weiteren Kapiteln allgemein auf die Programmiersprachen JavaScript und Python selbst eingegangen.

In Kapitel 4 wird generell auf Entwurfsmuster eingegangen. Die Umsetzung des Singleton, Observer und Adapter Pattern in beiden Programmiersprachen, befinden sich in den Kapiteln 5, 6 und 7.

Vor der jeweiligen Umsetzung der Entwurfsmuster werden die teilnehmenden Komponenten des Programmcodes aufgelistet. Nach Vorstellung eines Design Patterns kommt es zum Vergleich, bei denen die Unterschiede und Besonderheiten betrachtet werden.

Im abschließenden Teil wird besonders auf die Unterschiede eingegangen und deren Wichtigkeit diskutiert.

1.3 Motivation

Das Hauptziel ist es, mit der Kenntnis von Entwurfsmustern in einer der beiden Programmiersprachen, einen schnellen Einstieg in die andere Sprache zu bekommen und einen tieferen Einblick in die beiden Programmiersprachen JavaScript und Python zu ermöglichen. Die Tatsache, dass JavaScript und Python sogenannte dynamisch typiisierte Skriptsprachen sind und sich von Grund auf sehr ähneln, lässt Spielraum für die Frage, ob sich die Umsetzung von Entwurfsmustern denn genauso ähnelt. Aus diesem Grund werden drei der dreiundzwanzig bekannten Design Patterns herangezogen und mit den Programmiersprachen JavaScript und Python implementiert. Erstmals vorgestellt im Buch "Design Patterns, Elements of Reusable Object-Oriented Software"[4], der sogenannten Gang of Four.

Da sich die Implementation von diesen allgemeinen Entwurfsmuster je nach Wahl der Programmiersprache stark unterscheiden

kann, wird der Quelltext so entwickelt, um einen verständlichen Übergang von JavaScript zu Python und umgekehrt zu ermöglichen. Besonders durch das bis 2015 fehlende Konzept der Klassen in JavaScript, unterscheidet sich die Implementierung vieler dieser 23 Entwurfsmuster zu anderen objektorientierten Programmiersprachen. Daher wird der von Ecma International eingeführte Standard 2015 (ES6)[5] genutzt. Mit diesem Standard wurde zum ersten Mal das Konzept der Klassen in JavaScript eingeführt. Um auf einen gemeinsamen Nenner mit der Programmiersprache Python zu kommen, werden die Entwurfsmuster mit diesem klassenbasierten Ansatz entwickelt. Für ein klareres Verständnis werden sehr spezielle Spracheneigenheiten beider Programmiersprachen weitestgehend vermieden.

Ein immenser positiver Nebeneffekt der Verwendung von Entwurfsmustern, ist die Lesbarkeit und Wiederverwendbarkeit des Quellcodes.

2 JavaScript

JavaScript ist eine interpretierte und kompilierte Multiparadigmen-Programmiersprache und bekannt als die Sprache des Webs. Sie wurde 1995 von Brendan Eich entwickelt. Sie ist prozedual, funktional und objektorientiert, verwendet aber keine Interfaces. Man ist also nicht an eines dieser Paradigmen gebunden, was JavaScript sehr flexibel macht.

JavaScript wurde zu Beginn nur in der Clientseitigen Entwicklung genutzt, um auf HTML-Elemente zuzugeifen und Webseiten eine gewisse Dynamik zu verleihen.

Ein großer Umschwung kam durch das Framework Node.js und verleiht JavaScript nun auch die Möglichkeit Serverandwendungen zu realisieren. Weitere Verwendung findet die Programmiersprache auch in der Programmierung von Desktop Applikationen, mobilen Anwendungen und Erweiterungen für Web-Browser. Vor allem wegen der beliebten und zahlreichen Frameworks, nimmt die Beliebtheit der Sprache stetig zu.

3 Python

Python ist eine objektorientierte Skriptsprache und wurde 1991 von Guido van Rossum entwickelt. Das Ziel von Guido van Rossum war es, all die störenden und überladenen Merkmale anderer Program-

miersprachen zu beseitigen und auf die Vorlieben der Softwareentwickler Community einzugehen ohne dabei an Effizienz zu verlieren. Python kann nahezu in allen Bereichen der Programmierung genutzt werden. Vor allem die große Anzahl an mitgelieferten Bibliotheken und Funktionen macht die Sprache so nützlich. Vor allem im Bereich Data Science und der Künstlichen Intelligenz hat sich Python als sehr hilfreich erwiesen. Frameworks wie TensorFlow[6] und scikit-learn[7] konnten schon einige revolutionelle Meilensteine erreichen. Python ist auch eine beliebte Programmiersprache im Bereich der Webentwicklung, insbesondere auch wegen der populären Frameworks wie Django[8] und Flask[9].

4 Entwurfsmuster

4.1 Definition und Vorteile

1995 wurden erstmals, im Buch "Design Patterns, Elements of Reusable Object-Oriented Software", dreiundzwanzig Design Patterns vorgestellt. Das sind die allgemein bekannten Entwurfsmuster, die bis heute noch ihre Gültigkeit finden.

Der bekannte Architekt und Entwurfstheoretiker Christopher Alexander sagt, “Jedes Muster beschreibt ein Problem, welches immer und immer wieder in unserer Umgebung auftritt und beschreibt den Kern der Lösung zu diesem Problem auf eine Art und Weise, so dass man diese Lösung über eine Millionen Mal benutzen kann ohne dies doppelt getan zu haben ”.[10]
Was Christopher Alexander über Gebäude und Städte sagte, trifft genau so auch auf das Entwerfen objektorienierter Software zu.

Nehmen wir an, man erkennt beim Programmieren immer wiederkehrende Probleme und entwickelt über die Zeit Herangehensweisen wie man diese Probleme lösen kann. Da jedoch nicht jedes Programmierproblem exakt auf die Zeile gleich ist, greift man jedes Mal zur den erlernten Herangehensweisen und wendet es auf die Probleme an. Diese Lösung ist jedoch zum gegebenen Zeitpunkt nicht dokumentiert und öffentlich anerkannt. Irgendwann kommt man dann eventuell auf die Idee, dass diese Lösung der Öffentlichkeit sehr nützlich sein könnte. Das sind die ersten Schritte zur Geburt eines Design Patterns. Man sollte beachten, dass solche Muster nicht als rein kopierbare Lösungen betrachtet werden sollten. Es geht vielmehr darum, für einen Satz an bekannten Problemen, entsprechende Lösungen zu entwicklen, welche auf diese Probleme anwendbar sind. Dadurch muss das Rad nicht neu erfunden werden

und man spart sich enorm viel Zeit bei der Problemlösung.

Ein weiterer entscheidender Vorteil von Design Patterns ist die Möglichkeit eine Abstraktionsschicht zu realisieren. Bei größeren Projekten kann der Programmcode logisch in Module aufgeteilt werden. Somit widmet sich jedes Team seinen eigenen Modulen und Änderungen beeinflussen dadurch andere Module nicht.

Ist man mit den Konzepten der Design Patterns vertraut, erkennt man relativ schnell verschiedene Problemstellungen und kann die erlernten Entwurfsmuster darauf anwenden. Auch andere Programmierer die an dem gleichen Code arbeiten oder Quellcode weiterentwicklen sollen, finden einen viel schnelleren Einstieg in die Programmlogik. Dieses Konzept kann insofern die Effizienz drastisch steigern.

4.2 Klassifizierung

Entwurfsmuster werden anhand zweier Kriterien klassifiziert. Zum einen dem *Gültigkeitsbereich* und zum anderen die *Absicht*. Das Kriterium des Gültigkeitsbereiches spezifiziert ob das Entwurfsmuster sich auf Beziehungen zwischen Klassen oder Objekten bezieht. Entwurfsmuster bezogen auf Klassen sind statisch und es wird viel mit Vererbung zwischen Klassen gearbeitet. Entwurfsmuster bezogen auf Objekte dagegen sind dynamische Beziehungen unter Objekten welche sich meist zur Laufzeit ändern können. Da fast alle Design Patterns sich in der Kategorie der Beziehung zwischen Objekten wiederfinden, wird in dieser Projektarbeit auf Design Patterns dieser Kategorie eingegangen.
Bekannt sind folgende drei Absichten von Entwurfsmuster:

- **Erzeugung**
- **Struktur**
- **Verhalten**

Erzeugende Muster befassen sich mit der Erzeugung von Objekten.
Strukturierende Muster gehen auf die Zusammensetzung von Klassen und Objekten ein.
Verhaltensmuster zeigen uns die Verantwortung von Klassen und Objekten und wie diese miteinander interagieren.
Bei der Umsetzung in den Kapiteln 5, 6 und 7, wird jeweils auf ein wichtiges Entwurfsmuster der drei Kategorien genauer eingegangen.

4.3 Entwurfsmuster in JavaScript

Wie schon erwähnt führte der ECMAScript Standard 2015 Klassen ein. Noch vor der Einführung von Klassen musste man diese, für einige Design Patterns programmiertechnisch emulieren. Hinsichtlich des neuen Standards, ist man nun in der Lage bestimmte Entwurfsmuster wie in klassenbasierten Programmiersprachen wie Python oder Java umzusetzen. Für Entwurfsmuster ohne den klassenbasierten Ansatz sind die beiden Bücher, JavaScript Patterns[11] und JavaScript Design Patterns[12] von Stefan Stojanov und Addy Osmani, sehr hilfreiche Ressourcen.

Die Einführung von Klassen in JavaScript macht es dem Programmierer nun möglich, eine ähnliche Herangehensweise wie in anderen Programmiersprachen, unter Verwendung herkömmlichen 23 Design Patterns zu verfolgen. Python-Entwickler finden durch die Einführung des neuen Standards, einen schnellen Einstieg zu JavaScript.

Zu beachten ist aber trotzdem, dass Klassen in JavaScript nur eine snytaktische Verschönerung für das bestehende prototypenbasierte Vererbungsmodell ist. Diese Erweiterung führt also kein neues objektorientiertes Modell in die Sprache ein. Mit der Funktionalität der Klassen ist es dem Programmierer möglich, verständlicher Objekte darzustellen und zu erzeugen. In JavaScript ist eine Klasse im Endeffekt eine Funktion, was wiederrum ein Objekt ist. Bei der Umsetzung im nächsten Kapitel wird noch näher darauf eingegangen.

5 Singleton

Das Singleton Entwurfsmuster ist ein Design Pattern in der Kategorie der erzeugenden Muster. Eine Singleton Klasse erlaubt es nur eine Instanz seiner Klasse zu erzeugen. Die Klasse stellt also sicher, dass keine weitere Instanziierung mehr folgt.

Ein sehr häufiges reales Singleton Beispiel wäre eine Verbindung zu einer Datenbank. Hier macht es Sinn nur das eine Objekt zu erzeugen, welche die Datenbankverbindung herstellt. Genau so wie es in einem Betriebssystem mit grafischer Oberfläche nur einen Window Manager geben sollte, welcher alle Fenster verwaltet. Singletons sollte man immer dort einsetzen wo Konfigurationen benötigt werden, die sich während der Programmlaufzeit nicht ändern.

Im nachfolgenden Kapitel wird auf die Umsetzung eines Singleton am Beispiel der Programmiersprache JavaScript eingegangen.

Bevor auf die Erläuterung eingegangen wird, empfiehlt es sich ander dieser Stelle mit dem Glossar vertraut zu machen. Dort befinden sich verschiedene Definitionen, um Spracheneigenheiten aufzuklären. In der Umsetzung der folgenden Entwurfsmuster wird bewusst auf Auslagerung der Klassen in Module verzichtet, da der Quellcode relativ überschaubaur ist. Vorraussetzung für die Ausführung der verschiedenen Programme ist für Python die Installation, mindestens der Version Python3.0[13]. In JavaScript benötigt man die aktuelle Version node.js 12.16.3[14]. Beide Downloads findet man auf den jeweiligen offiziellen Webseiten.

5.1 Herangehensweise

Die folgenden Beispiele erzeugen eine Instanz einer Datenbank und geben die IP-Adresse und den Datenbanknamen auf der Konsole aus.

Zum besseren Verständnis wird das Programmbeispiel so allgemein wie möglich gehalten und stellt deshalb nicht wirklich eine Datenbankverbindung her. Wichtig hierbei ist es das Grundprinzip des Singleton Entwurfsmusters zu verstehen, was mit dieser vereinfachten Darstellung erzielt werden soll.

Die essentielle Aufgabe eines Singleton ist die Überprüfung ob es schon eine Instanz dieser Klasse gibt. Ist das nicht der Fall, wird sie erzeugt und die Kontruktorfunktion gibt die erzeugte Instanz zurück. Bei jedem Versuch eine neue Instanz dieser Klasse zu erzeugen, wird immer wieder die zu Beginn erzeugte Instanz referenziert. Dies ist nur möglich, wenn wir die Referenz zum Singleton Instanzübergreifend speichern. Das bedeutet, diese Eigenschaft ist keine Eigenschaft einer Instanz sondern eine Eigenschaft der Klasse. Das ist notwendig, denn andernfalls würde immer die aktuell erzeugte Instanz zurückgegeben. Dazu aber gleich im Kapitel der Programmerläuterung.

Das folgende Kapitel beschreibt die Teilnehmenden Komponenten des Singleton Entwurfsmusters.

5.1.1 Teilnehmer

- *Singleton* - stellt Nutzern die Möglichkeit zur Verfügung eine Instanz der Klasse Singleton zu erzeugen[15]

5.2 JavaScript

5.2.1 Quelltext

```
1  class DB {
2   constructor(ip, dbName ){
3     if(DB.singletonInstanz  == null){
4       this.ip = ip;
5       this.dbName = dbName;
6       DB.singletonInstanz = this;
7       console.log("Server: " , this.dbName , " mit der
        ↪  IP-Adresse: " , this.ip , " verbunden!" );
8
9       return DB.singletonInstanz
10    }
11    else{
12      console.log("Weitere Instanziierung nicht möglich.
        ↪  Singleton Klasse!");
13
14      return DB.singletonInstanz;
15    }
16  }
17 }
18
19
20 const db = new DB("52.0.14.116" , "NASA");
21 const db2 = new DB("127.0.0.1" , "loopback");
```

5.2.2 Erläuterung

Die Datenbankklasse trägt den Namen DB und der Konstruktor nimmt eine IP-Adresse und den Datenbanknamen entgegen.

Wir beginnen im Programmcode mit der Abfrage ob denn schon eine Instanz erzeugt wurde. Das realisieren wir indem wir die Eigenschaft `DB.singletonInstanz` auf `null` prüfen. Man kann sie als Klassenattribut oder statische Eigenschaft sehen, welche keine Eigenschaft der aktuellen Instanz ist, sondern eine Instanzübergreifende Eigenschaft der Klasse und nur über den Klassennamen `DB` aufrufbar. Sozusagen eine globale Variable. Wichtig zu wissen ist an dieser Stelle, dass in JavaScript bei der Erzeugung von Eigenschaften ohne Wertzuweisung, mit dem vorangestellten Klassennamen

und dem Punktoperator, eine Eigenschaft automatisch den Wert `null` oder `undefined` zugewiesen bekommt. Das bedeutet, wir können mit dem Schlüsselwort `null` oder `undefined`, prüfen ob ein JavaScript-Objekt noch keinen Wert zugewiesen bekommen hat. Das bedeutet, eine Variable ist deklariert und befindet sich ohne einen Wert im Speicher. Dies machen wir uns zu Nutze und erstellen eine explizite Abfrage, ob die Eigenschaft `DB.singletonInstanz` schon einen Wert besitzt. Um auf die Eigenschaft `singletonInstanz` zuzugreifen, nutzen wir den Klassennamen `DB`. Das Attribut `DB.singletonInstanz` ist der zentrale Punkt in unserem Programmcode.

Wie man im Programmcode sehen kann, werden Variablen und Funktionsnamen im camelCase [1] Stil beschrieben, einer JavaScript-üblichen Konvention.

Im nächsten Schritt initialisieren wir drei Eigenschaften. Die ersten beiden sind die Instanz-Eigenschaften `ip` und `dbName` die mit dem vorangestellten Schlüsselwort `this`, die Eigenschaften der aktuellen Instanz wiederspiegeln. Diese beiden Eigenschaften der Klasse kann man als lokale dynamische Variablen betrachten, da sich die Werte in `ip` und `dbName` je nach Parameterübergabe während der Objekterzeugung ändern und deren Gültigkeitsbereich sich auf eine Instanz beschränkt.

Dem dritten Attribut `DB.singletonInstanz`, weisen wir das Schlüsselwort `this` als Wert zu, was die Referenz der aktuellen Instanz ist. Möchte man sich anschauen was `this` an dieser Stelle denn genau bedeutet bzw. was es referenziert, kann man mit der Ausgabe `console.log(this)` in der Konsole sehen, dass `this` dem aktuellen Objekt `DB{ip: '52.0.14.116', dbName: 'NASA'}` mit seinen 2 Eigenschaften entspricht.

```
1  class DB {
2    constructor(ip, dbName ){
3      if(DB.singletonInstanz  == null){
4        this.ip = ip;
5        this.dbName = dbName;
6        DB.singletonInstanz = this;
7        // DB { ip: '52.0.14.116', dbName: 'NASA' }
8        console.log(this);
9      .
```

[1] Die camelCase Scheibweise ist eine Namenskonvention, bei der bei der ein Name aus mehreren Wörtern gebildet wird. Dabei wird jedes neue Wort mit einem Großbuchstaben begonnen. Die Wörter werden direkt aneinandergereiht, es werden keine Leerzeichen oder Unterstriche verwendet.

```
10      .
11      .
```

Hier die Ausgabe des oben aufgeführten verkürzten Codeabschnitts in der Konsole, ausgeführt mit Node.js im Terminal:

```
bash@terminal:~/Patterns$ node singleton.js
DB { ip: '52.0.14.116', db_name: 'NASA' }
```

Wollen wir nun ein weiteres Objekt mit:

```
const db2 = new DB("127.0.0.1" , "loopback");
```

dieser Klasse erzeugen, werden wir gewarnt, dass dies nicht möglich ist. Das passiert, weil beim erneuten Aufruf des Kontruktors die Variable `DB.singletonInstanz` nicht mehr `null` ist. Denn wir haben zuvor schon eine Instanz erzeugt die genau in dieser Variablen gespeichert bleibt. Dementsprechend springt das Programm in die `else`-Verzweigung und die Konstruktorfunktion bekommt als Rückgabewert die zuvor erzeugte Instanz zurück. Jetzt werden wir darüber informiert, dass es sich um eine Singleton Klasse handelt:

```
else{
      console.log("Weitere Instanziierung nicht möglich.
      ↪  Singleton Klasse!");
      return DB.singletonInstanz;
    }
```

Man sollte ein Singleton so Transparent wie möglich gestalten, weil man auf den ersten Blick nicht sofort sieht, dass es sich um eine Singleton Klasse handelt. Daher ist es von Wichtigkeit die entsprechende Stelle zu kommentieren und zusätzlich eine Warnung auszugeben. Genau aus diesem Grund wird in diesem Beispiel bei erneuter Instanziierung eine Warnung auf der Konsole ausgegeben.

Fügen wir nach der Anweisung in Zeile 21 eine Konsolenausgabe ein, kann man nun sehen wie das zweite Objekt `db2` die Eigenschaften des ersten Objekts `db1` besitzt. Der folgende Codeabschitt:

```
const db1 = new DB("52.0.14.116" , "NASA");
console.log(db1);
const db2 = new DB("127.0.0.1" , "loopback");
console.log(db2);
console.log(Object.is(db, db2))
```

erzeugt die Ausgabe:

```
x@x:~/Patterns$ node singleton.js
Server: NASA mit der IP-Adresse: 52.0.14.116  verbunden!
DB { ip: '52.0.14.116', db_name: 'NASA' }
Weitere Instanziierung nicht möglich. Singleton Klasse!
DB { ip: '52.0.14.116', db_name: 'NASA' }
true
```

Wie man sehen kann referenziert das zweite Objekt `db2` das erste Objekt `db1`. Genauer gesagt, sie liegen an der gleichen Stelle im Speicher. Mit der Funktion `Object.is()` kann man zwei Werte auf ihre Gleichheit prüfen. Unter anderem auch auf die Gleichheit des Objekts, sprich der äquivalenten Speicherstelle. In der letzten Zeile der Ausgabe des Terminals ist zu sehen, dass die beiden Instanzen die gleichen sind und das Ergebnis aus der Evaluierung der Funktion ist daher `true`.

5.3 Python

5.3.1 Quelltext

```
1  class DB:
2      singleton_instanz = None
3      def __new__(cls, ip, db_name):
4          if DB.singleton_instanz == None:
5              DB.singleton_instanz = super().__new__(cls)
6              cls.ip = ip
7              cls.db_name = db_name
8              print("Server: ", cls.db_name, " mit der
                   ↪  IP-Adresse: ", cls.ip, " verbunden!" )
9              return DB.singleton_instanz
10         else:
11             print("Weitere Instanziierung nicht möglich.
                   ↪  Singleton Klasse!")
12             return DB.singleton_instanz
13
14
15 db1 = DB("127.0.0.1", "NASA")
16 db2 = DB("asdsd","asddsf")
```

5.3.2 Erläuterung

Im nächsten Beispiel sehen wir die Umsetzung einer Singleton Klasse in der Programmiersprache Python. Ausführbar mit der Python Version ab 3.0 . Hier verwenden wir wieder das Datenbankbeispiel.

Die Klasse trägt den Namen DB und wir erzeugen ausserhalb des Konstruktors eine Klasseneigenschaft `singleton_instanz`. Theoretisch ist dies eine statische Variable. Das ist die Variable in der später die Referenz zum Singleton gespeichert wird. Zunächst weisen wir dieser Variable das Schlüsselwort `None` zu. In Python ist `None` ein Weg einer Variable nicht wirklich einen Wert zuzuweisen. Das Interessante an dem Schlüsselwort `None` in Python ist, dass `None` selbst ein Singleton ist, ein Singleton der Klasse `NoneType`. Das bedeutet, allen Variablen denen `None` zugewiesen wird, referenzieren immer nur das eine `None` Objekt an der gleichen Stelle im Speicher. Doch wir nutzen `None` hier in einem andere Kontext. Man könnte der Variable einen beliebigen Wert zuweisen. Die Variable `singleton_instanz` wird also sozusagen als Flag [2] benutzt, um später zu prüfen ob sich der Wert der Variable verändert hat.

Der `__new__()`-Konstruktor nimmt drei Parameter entgegen. Die Referenz zur Klasse `cls` und die zwei Instanzattribute `ip` und `db_name`. Python besitzt zwei Konstruktoren und zwar den `__new__()`-Konstruktor, eine statische Methode für die Erzeugung eines Objekts und den `__init__()`-Konstruktor, eine Instanzmethode für die Modifikation des Objektes. Die Tatsache, dass der`__init__()`-Konstruktor nur einen impliziten Rückgabewert besitzt, genauer gesagt die aktuelle Instanz, macht es uns nicht möglich dem Konstruktor einen anderen Rückgabewert zu übergeben. Doch genau diese Funktionalität benötigen wir für ein Singleton. Deswegen nutzen wir den `__new__()`-Konstruktor, um explizit das Singelton Objekt zurückzugeben.
Der `__new__()`-Konstruktor nimmt grundsätzlich mindestens ein Parameter entgegen, `cls` und optionale Parameter. Mit dem Bezeichner`cls` kann dann auf die, ausserhalb des Konstruktors deklarierte Klasseneigenschaft `singleton_instanz` zugegriffen werden. In der Python-üblichen Konvention wird der Bezeichner `cls` in Verbindung mit dem `__new__()`-Kontruktor benutzt, um auf Klassen und Instanzattribute zuzugreifen, doch aber nicht auf Instanzattribute die sich in dem `__init__()`-Konstruktor befinden.

Der Konstruktor `__init__()` muss in unserem Quellcode nicht explizit angegeben werden, da dieser implizit nach dem Aufruf des

[2] Ein Flag ist ein Statusindikator. Er wird für die Kennzeichnung bestimmter Zustände benutzt.

__new__() aufgerufen wird und die Eigenschaften ip und dbName automatisch als Parameter übergeben bekommt. Die Alternative hierzu wäre, dass wir den zweiten Kontruktor __init__() nutzen und beiden Eigenschaften ip und db_name dort deklarieren. Folgenden Codeabschitt würden wir für diese Variante nach dem __new__()-Konstruktor einfügen:

```
def __init__(self, ip, dbName):
    self.ip = ip
    self.dbName = dbName
```

Die Instanzmethode __init__() nimmt als ersten Parameter die Referenz zur aktuellen Instanz self und die zusätzlichen Parameter des __new__()-Konstruktor entgegen. Der Bezeichner self ist wieder eine Python-übliche Konvention die in Verbindung mit dem __init__()-Kontruktor verwendet wird, um auf die Eigenschaften der aktuellen Instanz zuzugreifen. Man sollte bei dieser Alternative nicht vergessen, dem __new__()-Konstruktor trotzdem die beiden Parameter ip und db_name zu übergeben, obwohl diese visuell nicht einmal in der Methode verwendet werden. Wie schon erwähnt überträgt der __new__()-Konstruktor implizit seine optionalen Parameter an den zweiten __init__()-Kontruktor. Hier der komplette Quelltext dazu.

```
1  class DB:
2      singleton_instanz = None
3      def __new__(cls, ip, db_name):
4          if DB.singleton_instanz == None:
5              DB.singleton_instanz = super().__new__(cls)
6              print("Server: " + db_name + " mit der IP-Adresse:
                   ↪  " + ip + " verbunden!" )
7              return DB.singleton_instanz
8          else:
9              print("Weitere Instanziierung nicht möglich.
                   ↪  Singleton Klasse!")
10             return DB.singleton_instanz
11
12
13     def __init__(self, ip, db_name):
14         self.ip = ip
15         self.db_name = db_name
```

Kehren wir nun wieder zur ursprünglichen Variante zurück. In Zeile 4 des Programmcodes beginnen wir mit der zentralen Abfra-

ge ob die Klasseneigenschaft `singleton_instanz` den Wert None besitzt, sprich ob denn schon eine Instanz erzeugt wurde. Ist das der Fall, greifen wir mit dem vorangestellten `DB` und dem Punktoperator auf die Klasseneigenschaft `singleton_instanz` zu und erzeugen die Singleton Instanz, indem wir `DB.singleton_instanz` die Referenz der aktuellen Instanz mit dem Befehl `super().__new__(DB)` oder `super().__new__(cls)` als Wert zuweisen. Im Endeffekt wird hier durch die `super()`-Methode, der aktuelle Rückgabewert des `__new__(DB)`-Konstruktors in dem wir uns befinden, mit der Klasse selbst `DB` als Parameter, geerbt. Noch zu erwähnen wäre, dass es ebenfalls möglich ist immer den Bezeichner `cls` statt `DB` zu nutzen. Mit beiden Bezeichnern greift man auf die Klasseneigenschaften zu und mit `cls` zusätzlich auf Instanzattribute des `__new__(DB)`-Konstruktors.

Im nächsten Schritt erstellen wir die weiteren Instanzattribute `ip` und `dbName` mit dem vorangestelltem `cls` . Da der Konstruktor bei einer Objekterzeugung immer automatisch aufgerufen wird, folgt auch die Ausgabe des Servernamen und die IP-Adresse auf der Konsole. Unser Singleton ist schließlich der Rückgabewert der Kontruktorfunktion, die Referenz zur aktuell erzeugten Instanz `DB.singleton_instanz`.

Die `else`-Verzweigung trifft ein, wenn das Klassenattribut `DB.singleton_instanz` nicht mehr den Wert `None` besitzt. Dies geschieht, wenn wir mit:

```
db2 = DB("127.0.0.1","loopback")
```

versuchen eine neue Instanz zu erzeugen. Nun werden wir durch die Konsolenausgabe gewarnt, dass dies nicht möglich ist und die Kontruktorfunktion bekommt als Rückgabewert die zuerst erzeugte Instanz des Programms zurück. Bei jedem weiteren Versuch eine neue Instanz zu erzeugen, referenzieren wir die Singleton Instanz:

```
    else:
        print("Weitere Instanziierung nicht möglich. Singleton
        ↪  Klasse!")
        return cls.singleton_instanz
```

Hier die Ausgabe auf der Konsole bei Ausführung des Programms:

```
bash@terminal:~/Patterns$ python3 singleton.py
Server: NASA mit der IP-Adresse: 52.0.14.116 verbunden!
Weitere Instanziierung nicht möglich. Singleton Klasse!
```

Zur Überprüfung ob wirklich bei jeder Objekterzeugung die Singleton Instanz referenziert wird, können wir in Python mit einer Konsolenausgabe der Funktion `id()` mit der Instanz als Parameter, die Speicherstelle betrachten. Ein Objekt in Python besteht aus einer Identität (seiner einzigartigen Speicheradresse), einem Typ der zur Laufzeit erkannt wird und einem Wert. Mit der Methode `id()` kann man die Identität eines Objektes betrachten. Sie ist die Übersetzung der hexadezimalen Speicheradresse, welche man mit der einfachen Ausgabe der Instanz ermitteln kann. Hier die Ermittlung der Speicheradressen und die Überprüfung ob die zweite Instanz die erste referenziert:

```
db1 = DB("52.0.14.116", "NASA")
print(db1)        #hexadezimal
print(id(db1))    #dezimal
db2 = DB("127.0.0.1","loopback")
print(db2)
print(id(db2))
print(db1 is db2)
```

Die Ausgabe im Terminal sieht wie folgt aus:

```
bash@terminal:~/Patterns$ python3 singleton.py
Server: NASA mit der IP-Adresse: 52.0.14.116  verbunden!
<__main__.DB object at 0x7fc787d0d850>
140494953830480
Weitere Instanziierung nicht möglich. Singleton Klasse!
<__main__.DB object at 0x7f96743cb5d0>
140494953830480
True
```

Wie man sehen kann, haben die Objekte `db1` und `db2` die gleiche Speicheradresse. Ebenso können wir mit dem Schlüsselwort `is` auf Gleichheit prüfen und bekommen als Ergebnis `True`.

5.4 Vergleich

Fassen wir nun die Unterschiede und Besonderheiten der Umsetzung zusammen.

- Eine Klasse in JavaScript ist nicht wirklich eine Datenstruktur, sondern letztendlich eine Funktion. Da alles in JavaScript

ein Objekt ist, ist auch eine Funktion ein Objekt. In Python ist ebenso alles ein Objekt, bestehend aus einer Identität, einem Typ und einem Wert.

- Die Identität in Python steht für die Speicheradresse. Die Programmiersprache JavaScript verzichtet bewusst auf Darstellung oder Manipulation von Speicheradressen, da die Speicheradressen zur Laufzeit vergeben werden und sich jederzeit ändern können.[16]

- In Python haben wir echte Klasseneigenschaften, können diese folglich ausserhalb des Konstruktors erzeugen und problemlos ohne Instanziierung darauf zugreifen.

- In JavaScript kann man nur mit dem Klassennamen auf statisch erzeugte Eigenschaften zugreifen, in Python ist das mit einem zusätzlichen übergebenen Parameter in der Klassenmethode möglich. `cls` ist hierbei die Konvention.

- Python besitzt zwei Konstruktoren, einer davon gibt implizit die aktuelle Instanz zurück und mit dem anderen Konstruktor ist es möglich einen beliebigen Wert entgegenzunehmen. In JavaScript gibt es nur einen Konstruktor mit modifizierbarem Rückgabewert.

- Python's Konstruktor `__new__()`, ist eine spezielle statische Methode die nicht als solche deklariert werden muss.[17]

- Offiziell besitzt JavaScript im Moment noch keine Klasseneigenschaften, diese sind nur eine Konvention. Um Klasseneigenschaften zu erzeugen, kann man dies im globalen Namensraum oder im Konstruktor der Klasse, mit dem vorangestellten Klassennamen tun. Ausserhalb des Konstruktors, doch innerhalb der Klasse wie in Python, ist dies nur bedingt möglich. ECMA International[3] beschäftigt sich jedoch gerade damit solche Klasseneigenschaften wie in Python einzuführen. Die Rede ist hier von *class fields*[19][20] . Mit dem vorangestellten Schlüsselwort `static`, kann man dann reale Klasseneigenschaften erzeugen. Mit dem Konzept der *class fields* ist es nun möglich ausserhalb des Konstruktors, Eigenschaften zu erzeugen. Das

[3] Ecma International ist eine Normungsorganisation, die Standards für Computer-Hardware, Kommunikationstechnik und Programmiersprachen entwickelt. Sie hat die ECMA-262[18] Spezifikation betreut, welche die Kernspezifikation von JavaScript ist.

einzige Problem hierbei ist die Inkompatibilität mit manchen Internet Browsern. Die Einführung von *class fields* in JavaScript ist momentan noch eine experimentelle Funktionalität. Auf der Webseite des Mozilla Developer Network[21], findet man dazu zwei Tabellen welche die Komptatibilität mit private und public class fields aufzeigen.

6 Adapter

Nun führen wir ein Entwurfsmuster in der Kategorie der strukturellen Muster ein.

Strukturelle Entwurfsmuster schlagen Wege vor, wie man Objekte zusammensetzt, um neue Funktionalitäten bereitzustellen.

Das Adapter Pattern, bekannt auch als Wrapper Pattern, hilft uns Schnitstellen die nicht kompatibel sind, kompatibel zu machen. Gründe hierfür sind vielseitig. Nehmen wir an wir wollen eine alte Komponente in ein neues System einbinden oder eine neue Komponente in ein altes System. Das Apapter Pattern realisiert dies ohne den Quelltext der vorhandenen Komponenten zu ändern. Er fungiert, wie es der Name schon sagt, als ein Adapter um die neuen Funktionen einzubinden. Das Adapter Entwurfsmuster übersetzt also Eigenschaften und Methoden eines Objekts zu einem anderen Objekt. Ein sehr effektives Entwurfsmuster das enorm viel Zeit spart, um nicht den bestehenden Code noch komplexer und eventuell unverständlicher zu machen. Auch die Wiederverwendbarkeit ist ein weiterer positiver Aspekt.

6.1 Herangehensweise

Führen wir nun ein Gedankenexperiment zum Verständnis ein. Man stelle man sich ein Smartphone, eine Steckdose und ein Ladegerät vor. Möchte man sein Smartphone aufladen, sollte man in Kenntnis der Ausgangsspannung der Steckdose sein. Diese Spannung variiert jedoch in einigen Ländern. Das bedeutet, wenn man zum Beispiel in Deutschland wohnt, benötigt man ein Ladegerät welches mit einer deutschen Steckdose funktioniert. Entsprechend diesen Anforderungen, benötigen wir einen Adapter/Ladegerät, was uns ermöglicht eine korrekte Eingangsspannung zu erzeugen und unser Smartphone aufzuladen. Je nach Land, gibt es einen speziellen Adapter der sich dieser Aufgabe widmet. Das bedeutet man muss für das Land in das man reist, im Besitz den benötigten Ladegerätes sein. Nehmen wir nun jedoch an, wir möchten einen universellen

Adapter einführen, der die jeweilige Ausgangsspannung des Landes umwandelt. Demnach müssen wir uns nicht mehr darum kümmern, ein Ladegerät des jeweiligen Landes zu erwerben und können somit unser Smartphone in jedem beliebigen Land der Welt aufladen.

Man könnte sich diesen Adapter wie ein Gerät vorstellen, in welches das Ladegerät des Smartphone eingesteckt wird und sich mit enstprechenden mechanischen Vorrichtungen an die jeweilige Steckdose des Landes anpasst.

Wie realisieren wir demnach einen Adapter, der mit allen Spannungen umgehen kann, ohne die Steckdose an sich umzubauen ? Die equivaläntе programmiertechnische Frage dazu lautet, wie fügen wir dem Programmcode die Funktionalität eines universellen Adapters hinzu, ohne die bestehenden Komponenten zu verändern ? Die Lösung wird anhand des Adapter Pattern aufgezeigt. Dazu entwickeln wir auf Basis eines bestehenden Adapters, einen universellen Adapter und fügen diesem die Fähigkeiten hinzu, jede beliebige Spannung umzuwandeln und das Gerät außerdem zu laden.

Das Adapter Pattern setzt die Objekte so geschickt zusammen, ohne die bestehende Hauptfunktionalität zu beeinträchtigen. Im Programmcode wird dazu eine eigene Klasse `UniAdapter` bereitgestellt. Diese Klasse erbt insofern von einem bestehenden Adapter. In Kombination mit den Eigenschaften der Klasse `UniAdapter` und einer Adapterklasse wie `DEAdapter`, können so neue Funktionalitäten hinzugefügt und mit dem existierenden Programmcode kompatibel gemacht werden.

6.1.1 Teilnehmer

- *Ziel* (EUSocket, USSocket) - definiert die Domain-spezifische Schnittstelle welche vom Nutzer verwendet wird
- *Adapter* (UniAdapter) - adaptiert die Schnittstelle der Adaptierenden zur Schnittstelle des Ziels
- *Adaptierender* (EUAdapter, USAdapter) - definiert Schnittstelle die adaptiert werden muss
- *Nutzer* (Smartphone) - interagiert mit Objekten entsprechend der Schnittstelle des Ziels.[22]

6.2 JavaScript

6.2.1 Quelltext

```
class Smartphone{
  constructor(){
    this.maxInputVolt = 5;
  }

  charge(convertedVolt){
    if(convertedVolt == this.maxInputVolt){
      console.log("Eingangsspannung: ", convertedVolt ," Volt, Smartphone lädt...");
    }
    else{
    console.log("Inkorrekte Eingangsspannung. Smartphone wird nicht geladen.");
    }
  }
}

class DESocket{
  constructor(){
    this.outputVolt = 230
  }
}

class USSocket{
  constructor(){
      this.outputVolt = 120
  }
}

class DEAdapter{
  constructor(deSocket, smartphone){
    this.smartphone = smartphone;
    this.deSocket = deSocket;
  }

  convert(){
    this.inputVolt = this.deSocket.outputVolt;
    if(this.inputVolt == 230){
```

```
37        this.convertedVolt = this.smartphone.maxInputVolt;
38        this.smartphone.charge(this.convertedVolt);
39      }
40      else{
41          console.log("Fehler!"};
42      }
43   }
44
45
46  class USAdapter{
47    constructor(usSocket, smartphone){
48      this.smartphone = smartphone;
49      this.usSocket = usSocket;
50    }
51
52    convert(){
53      this.inputVolt = this.usSocket.outputVolt;
54      if(this.inputVolt == 120){
55        this.convertedVolt = this.smartphone.maxInputVolt;
56        this.smartphone.charge(this.convertedVolt);
57      }
58      else{
59          console.log("Fehler!"};
60      }
61   }
62
63  class UniAdapter extends DEAdapter{
64    constructor(socket, smartphone){
65      super(socket, smartphone);
66      this.inputVolt = socket.outputVolt;
67      if(this.inputVolt == 230){
68          console.log("Deutsche Steckdose erkannt: Übersetzung
            ↪  von 230V in Eingangsspannung des Smartphones");
69          this.convert();
70      }
71      else{
72          console.log("Amerikanische Steckdose erkannt:
            ↪  Übersetzung von 120V in Eingangsspannung des
            ↪  Smartphones");
73          this.convert();
74      }
75    }
```

```
76    convert(){
77    this.convertedVolt = this.smartphone.maxInputVolt;
78    this.smartphone.charge(this.convertedVolt);
79    }
80  }
81
82  const smartphone = new Smartphone();
83  const deSocket = new DESocket();
84  const usSocket = new USSocket();
85
86  //const deAdapter = new DEAdapter(deSocket, smartphone);
87  //deAdapter.convert();
88  const uniAdapter  = new UniAdapter(deSocket,smartphone);
```

6.2.2 Erläuterung

Es stehen sechs Klassen zur Verfügung. In der Klasse `Smartphone` speichern wir die maximale Eingangsspannung des Smartphones und die Ausgabe auf dem Bildschirm, dass das Smartphone lädt.

Die Klasse `DESocket` und `USSocket` beinhalten jeweils die Eigenschaft in der die Ausgangsspannung des jeweiligen Landes gespeichert ist. Die Klassen `DEAdapter` und `USAdapter` konvertieren die jeweilige Ausgangsspannung der Steckdose in die korrekte Eingagangsspannung des Smartphones. .

Kommen wir zunächst zur Beschreibung der bestehenden Codebasis ohne Nutzung des Adapter Patterns und somit ohne Funktion eines universellen Adapters. Um das Programm nun auszuführen instanziieren wir zunächst die Klassen `Smartphone`, `DESocket` und `USSocket`. Dazu kreieren wir für jede Instanz, eine Variable mit dem vorrangestellten Schlüsselwort `const`:

```
const smartphone = new Smartphone();
const deSocket = new DESocket();
const usSocket = new USSocket();
```

In JavaScript gibt es drei Varianten Variablen zu erzeugen. Mit `let`, `var` und `const`. Bei `let` und `const` beschränkt sich der Gültigkeitsbereich auf den lokalen Block, genauer gesagt innherhalb eines Paares geschweifter Klammer. Mit `const`, wie es der Name schon vermuten lässt, wird aus der Variable eine Konstante und ist somit unveränderbar. Das Schlüsselwort `var` bezieht sich auf den Gültigkeits-

bereich einer Funktion. In unserem Fall ist es realtiv unbedeutend, welche der drei Varianten wir nutzen, da sich die Erzeugung der Instanzen auf Modulebene und sonach auf globaler Ebene befindet.

Bei der Erzeugung der Instanz `deAdapter`, übergeben wir dem Konstruktor der Klasse `DEAdapter`, mit der Instanz `deSocket`, die Ausgangsspannung einer deutschen Steckdose und mit `smartphone`, die Instanz der Klasse `Smartphone`. Im Konstruktor erzeugen wir zwei Eigenschaften und weisen ihnen die beiden Instanzen zu. Nun können auf alle Eigenschaften beider Klassen zugegriffen werden.

Um die Konvertierung der Ausgangsspannung der deutschen Steckdose in die Eingagsspannung des Smartphones zu beginnen, wird in Zeile 87 die Methode `convert()` aufgerufen. In dieser Methode speichern wir über den Aufruf `this.deSocket.outputVolt` der Klasse `DESocket`, die Ausgangsspannung von 230 Volt, als Eingangsspannung des Adapters in `this.inputVolt`. Jetzt prüft der Adapter ob es sich bei der Eingagsspannung um 230 Volt handelt. Ist das der Fall, holt sich die Eigenschaft `this.convertedVolt` mit dem Aufruf `this.smartphone.maxInputVolt` , die maximale Eingangsspannung des Smartphone und lädt es mit dem Befehl:

```
this.smartphone.charge(this.convertedVolt);
```

Liegt ein anderer Wert vor, wird eine Fehlermeldung auf der Konsole ausgegeben. Genau mit diesen Schritten simulieren wir sozusagen, die Konvertierung von 230 Volt der Steckdose, in 5 Volt des Smartphone.

Die Methode `charge()` der Klasse `Smartphone`, nimmt eine Spannung `inputVolt` entgegen, in diesem Fall die übersetzte Spannung und fängt an das Smartphone mit der korrekten Spannung zu laden. Zuvor wird noch überprüft ob die übersetzte Spannung der maximalen Eingangsspannung des Smartphone entspricht. Trifft dieser Fall ein, wird das Smartphone geladen. Gleicht die übersetzte Spannung nicht der maximalen Eingangsspannung, wird das Smartphone nicht geladen. Folgende Ausgabe ist bei korrekter Spannung zu sehen:

```
bash@terminal:~/Patterns$ node adapter.js
Eingangsspannung: 5 Volt, Smartphone lädt ...
```

Nun erweitern wir das Programm mit der Funktionalität, eine beliebige Ausgangsspannung entgegenzunehmen und diese in die korrekte Eingangsspannung des Smartphone zu übersetzen. Mit Hilfe des Adapter Pattern nutzen wir den bestehenden Adapter `DEAdapter`,

um den universellen Adapter zu erstellen. Dadurch nutzen wir demzufolge die alte Funktionalität des Programms, um eine weitere zum System hinzuzufügen.

Wir erstellen eine neue Klasse `UniAdapter`, welche von einem beliebigen Adapter erbt. In diesem Beispiel erben wir von der Klasse `DEAdapter`. Der Konstruktor nimmt die Instanz einer Steckdose und die Instanz eines Smartphone entgegen. Zusätzlich erbt die Klasse explizit die beiden übergebenen Parameter im Konstruktor der Klasse `DEAdapter`, mit `super(socket, smartphone)`. Um eine beliebige Spannung entgegenzunehmen, nutzen wir die Eigenschaft `this.inputVolt` der Elternklasse `DEAdapter`, welche über die Erzeugung der Instanz in Zeile 88 der Klasse `UniAdapter` geschieht. Fakt ist, dass die Elternklasse durch die Überprüfung auf Spannungshöhe mit der Instanz `deSocket`, sozusagen nur die Ausgangsspannung einer deutschen Steckdose über 230 Volt entgegennimmt. Da wir aber die Methode `charge()` erben, überschreiben wir sie mit unserer eigenen Funktionalität der Überprüfung auf beliebige Spannungen. Das ist die Kernfunktionalität welche durch Anwendung des Adapter Pattern zum Tragen kommt.

Nach der Überprüfung der jeweiligen Spannung, wird die überschriebene Methode `charge()` aufgerufen. Den maximalen Wert für die Eingansspannung des Smartphone wird über die Instanz `smartphone`, welche über die Klasse `DEAdapter` geerbt wurde, in die Eigenschaft `this.convertedVolt` gespeichert und somit übersetzt.

Der letzte Schritt ist schließlich der Aufruf der Methode `charge()` über die geerbte Instanz `smartphone`.

Um den universellen Adapter in das Programm einzubinden kommentieren wir Zeile 86 und 87 aus:

```
//const deAdapter = new DEAdapter(deSocket, smartphone);
//deAdapter.convert();
let uniAdapter  = new UniAdapter(usSocket,smartphone);
```

Wollen wir eine Konvertierung einer amerikanischen Steckdosenspannung in die korrekte Eingangsspannung des Smartphone, übergeben wir dem Adapter die Instanz `usSocket` als Parameter und erhalten als Ausgabe:

```
bash@terminal:~/Patterns$ node adapter.js
Amerikanische Steckdose erkannt: Übersetzung von 120V in
↪  Eingangsspannung des Smartphones
Eingangsspannung: 5 Volt, Smartphone lädt ...
```

Liegt die Eingangsspannung des Adapters bei 230V, folgt der gleiche Prozess, nur als Erkennung einer deutschen Steckdose. Nun kann jede beliebige Spannung in den Adapter fließen und umgewandelt werden. Ausführung des Programmcode mittels Parameterübergabe der Instanz, in der sich die deutsche Ausgangsspannung von 230 Volt befindet, folgt mit:

```
let uniAdapter = new UniAdapter(deSocket,smartphone);
```

ensteht die folgende Ausgabe auf der Konsole :

```
bash@terminal:~/Patterns$ node adapter.js
Deutsche Steckdose erkannt: Übersetzung von 230V in
↪  Eingangsspannung des Smartphones
Eingangsspannung: 5 Volt, Smartphone lädt ...
```

Wie wir sehen können, ist es nun möglich mit jeder beliebigen Ausgangsspannung einer Steckdose, diese zu übersetzen und das Smartphone mit der benötigten maximalen Eigangsspannung zu laden. Genauso kann auch jedes beliebige Smartphone angeschlossen werden, da die Klasse `Smartphone` jederzeit austauschbar ist.

Natürlich kann man das Programm mit mehreren Überprüfungen auf verschiedene Spannungshöhen unterschiedlicher Länder erweitern. Wie bereits erwähnt, sollte es als Gedankenexperiment dienen, um das Entwurfsmuster zu verdeutlichen.

Mit einfacher Vererbung, geschickter Zusammensetzung von Instanzen und das Überschreiben einer Methode, konnte das Adapter Pattern hier angewendet werden.

Im nächsten Kapitel gehen wir auf die Umsetzung mit der Programmiersprache Python ein.

6.3 Python

6.3.1 Quelltext

```
1 class Smartphone:
2     def __init__(self):
3         self.max_input_volt = 5
4
5     def charge(self, converted_volt):
6         if(converted_volt == self.max_input_volt):
7             print("Eingangsspannung: ", converted_volt, "Volt,
              ↪  Smartphone lädt...")
```

```
8           else:
9               print("Inkorrekte Eingangsspannung. Smartphone
                ↪  wird nicht geladen.")
10
11  class DESocket:
12      def __init__(self):
13          self.output_volt = 230
14
15  class USSocket:
16      def __init__(self):
17          self.output_volt = 120
18
19  class DEAdapter:
20      def __init__(self, de_socket, smartphone):
21          self.smartphone = smartphone
22          self.de_socket = de_socket
23
24      def convert(self):
25          self.input_volt = self.de_socket.output_volt
26          if(self.input_volt == 230):
27              self.converted_volt =
                ↪  self.smartphone.max_input_volt
28              self.smartphone.charge(self.converted_volt)
29          else:
30              print("Fehler")
31
32  class USAdapter:
33      def __init__(self, us_socket, smartphone):
34          self.smartphone = smartphone
35          self.us_socket = us_socket
36
37      def convert(self):
38          self.input_volt = self.us_socket.output_volt
39          if(self.input_volt == 120):
40              self.converted_volt =
                ↪  self.smartphone.max_input_volt
41              self.smartphone.charge(self.converted_volt)
42          else:
43              print("Fehler")
44
45
46  class UniAdapter(DEAdapter):
```

```
47      def __init__(self, socket, smartphone):
48          super().__init__(socket, smartphone)
49          self.input_volt = socket.output_volt
50
51          if(self.input_volt == 120):
52              print("Amerikanische Steckdose erkannt:
                ↪  Übersetzung von 120V in Eingangsspannung des
                ↪  Smartphones")
53              self.convert()
54          else:
55              print("Deutsche Steckdose erkannt: Übersetzung von
                ↪  230V in Eingangsspannung des Smartphones")
56              self.convert()
57
58      def convert(self):
59          self.converted_volt = self.smartphone.max_input_volt
60          self.smartphone.charge(self.converted_volt)
61
62  smartphone = Smartphone()
63  de_socket = DESocket()
64  us_socket = USSocket()
65
66  #de_adapter = DEAdapter(de_socket, smartphone)
67  #us_adapter = USAdapter(de_socket, smartphone)
68
69  #de_adapter.convert()
70
71  uni_adapter = UniAdapter(us_socket, smartphone)
```

6.3.2 Erläuterung

Die Implementierung des Adapter Pattern ist nahezu identisch zu JavaScript. Nun zur Erläuterung des Quellcodes.

Es stehen uns sechs Klassen zu Verfügung. Die Klasse `Smartphone` mit der Instanzeigenschaft der maximalen Eingangsspanung des Smartphone `self.max_input_volt` und der Methode `charge()` um das Smartphone zu laden.

Vorab ist noch zu erwähnen, dass es eine Python-übliche Konvention ist, Variablen und Funktionen im snake_case [4] zu benennen.

[4] Bei der snake_case Namenskonvention, werden Wörter mit einem Unterstrich aneinandergereiht.

Die Methode `charge()` nimmt die beiden Parameter `self` und die Eingangsspannung `converted_volt` als Eingangsspannung des Smartphone entgegen. Wird diese Methode aufgerufen, fängt das Smartphone an zu laden und es folgt die entsprechende Meldung auf der Konsole.

In den beiden Klassen `DESocket` und `USSocket`, welche die jeweiligen Steckdosen des Landes darstellen soll, befindet sich der jeweilige Spannungsausgangswert des Landes.

Die Klasse `DEAdapter`beherbergt die Eigenschaften `input_volt` und `converted_volt`. Weiter, nimmt der Konstruktor die Instanzen `de_socket` und `smartphone` entgegen. Dies ist nötig, um auf die maximale Eingangsspannung des Smartphone und die Ausgangsspannung der Steckdose mit 230 Volt zugreifen zu können. In dieser Klasse findet im Endeffekt die Übersetzung der Ausgangsspannung von 230 Volt der Steckdose, zu der erwünschten Eingangsspannung von 5 Volt zum Smartphone statt. Die Eingangspannung für den Adapter befindet sich in der Eigenschaft `input_volt`. Initialisiert wird diese Variable mit dem Wert 230 welche sich in der Klasse `DESocket` befindet. Die Konvertierung der Ausgangsspannung über 230 Volt erfolgt über die Methode `convert()`, die in Zeile 69 aufgerufen wird. Davor wird geprüft ob der Spannungswert bei 230 Volt liegt. Ist diese Bedingung wahr, erfolgt die Initialisierung der Variable `converted_volt`, mit dem Aufruf über die Instanz `smartphone`, dem maximalen Wert für die Eingangspannung des Smartphone `max_input_volt` über 5 Volt. Hiermit findet sozusagen die Umwandlung statt. Letzten Endes wird das Smartphone mit:

```
self.smartphone.charge(self.converted_volt)
```

geladen.

Ist die Bedingung nicht wahr, erfolgt eine Fehlermeldung auf der Konsole.

Die übersetzte Spannung `converted_volt` wird der Methode `charge()` übergeben, das Smartphone nimmt die Spannung entgegen und prüft auf Gleichheit mit seiner maximalen Eingangsspannung von 5 Volt.

Die Klasse `USAdapter` hat dieselbe Funktionalität wie die Klasse `DEAdapter`, nur mit dem Ziel 120 Volt einer amerikanischen Steckdose, in 5 Volt Eingangsspannung des Smartphone zu übersetzen.

Für die Ausführung des Programmes werden die Instanzen `smartphone`, `de_Socket`, `us_Socket` und `de_dapter` erzeugt. Der Adapter `DEAdapter` benötigt die Ausgangsspannung der deutschen Steckdose mit der Instanz `de_socket` als Parameter und das Smartphone um seine Funktion ausführen zu können. Ebenso der Adapter

`USAdapter`, der die Instanz `us_socket` als Parameter benötigt. Der letzte Codeabschnitt muss wie folgt aussehen:

```
smartphone = Smartphone()
de_socket = DESocket()
us_socket = USSocket()

de_adapter = DEAdapter(de_socket, smartphone)
us_adapter = USAdapter(de_socket, smartphone)

de_adapter.convert()
```

Bisher brauchten wir für eine amerikanische und deutsche Steckdose jeweils einen Adapter bzw. Ladegerät. Nun führen wir aber einen universellen Adapter ein, der in jedem Land funktioniert.

Für die erweiterte Funktionalität eines universellen Adapters erstellen wir eine zusätzliche Klasse `UniAdapter`, welche sich dieser Aufgabe widmet. Je nach Eigangsspannung erkennt diese Klasse, um wie viel Volt es sich handelt, wandelt den Wert in die korrekte Spannung des Smartphone um und lädt es auf. Dazu erbt die Klasse `UniAdapter` von der Klasse `DEAdapter`, für Nutzung und Erweiterung der Funktionalität. Der Konstruktor der Klasse `UniAdapter` nimmt im Konstruktor die beiden Parameter `socket` und `smartphone` entgegen und erbt zusätzlich mit dem Aufruf `super().__init__(socket, smartphone)` die Eigenschaften der Elterklasse `UniAdapter`. Nun können die Eigenschaften der beiden Instanzen aufgerufen werden.

Der Zugriff auf die Ausgangsspannung des jeweiligen Landes erfolgt über die Instanz `socket`, welche dann die Eingangsspannung des Adapters in der Eigenschaft `self.input_volt` darstellt. Dieser Wert wird durch eine `if, else`-Anweisung geprüft und dementsprechend wird der Wert umgewandelt.

Liegt der Wert der Eingangsspannung bei 120, wird eine amerikanische Steckdose erkannt und in der Zeile 53 wird die Methode `convert()` aufgerufen, welche folgende beiden Befehle ausführt:

```
self.converted_volt = self.smartphone.max_input_volt
self.smartphone.charge(self.converted_volt)
```

Diese Anweisungen stellen die Haupfunktion des Programmes dar. Mit der ersten Anweisung findet die eigentliche Umwandlung der Ausgangsspannung des jeweiligen Landes, in die Eingangsspannung des Smartphone statt und die zweite Anweisung lädt das Smartphone. Hier wird über die Instanz `smartphone`, die als Parameter über-

geben wurde, auf die maximale Eingangsspannung des Smartphone zugegriffen. Dies ist nun sozusagen die konvertierte Ausgangsspannung des Adapters und wird in der Eigenschaft `converted_volt` gespeichert. Diese Eigenschaft wird der Methode `charge()` übergeben und das Smartphone lädt.

Liegt der Wert bei 230 Volt, trifft die `else`-Verzweigung ein und es folgt die gleiche Prozedur nur als Erkennung einer deutschen Steckdose.

Was wir hier tun ist, die eigentliche Methode `convert()` der Klasse `DEAdapter`, mit der zuvor beschriebenen zusätzlichen Funktionalität zu überschreiben. Das ist notwendig, da der ursprüngliche Adapter nur für die Spannung eines Landes vorgesehen ist. Mit dem Adapter Pattern bringen wir also eine neue Schnittstelle in das System, welches Gebrauch eines alten individuellen Adapter macht und diesen mit der Umwandlung beliebiger Spannungen ausweitet.

Um das Programm wirklich ausführen zu können, sollten folgende Zeilen auskommentiert werden und der letzte Codeabschnitt folgendermaßen aussehen:

```
smartphone = Smartphone()
smartphone = Smartphone()
de_socket = DESocket()
us_socket = USSocket()

#de_adapter = DEAdapter(de_socket, smartphone)
#us_adapter = USAdapter(de_socket, smartphone)

#de_adapter.convert()

uni_adapter = UniAdapter(us_socket, smartphone)
```

Da wir nun einen universellen Adapter hinzugefügt haben, können die Instanzen und Methodenaufrufe der individuellen Adapter auskommentiert werden.

Das Smartphone wird mit Übergabe der Instanzen `de_socket` und `smartphone` geladen. Der entscheidende erste Parameter `socket`, der Klasse `UniAdapter`, nimmt nun jede beliebige Spannung entgegen und lädt das Smartphone. Das bedeuet, jede beliebige Instanz einer Socketklasse kann entgegengenommen werden.

Schließlich wurde das Programm um die Funktionalität erweitert, eine beliebe Spannung zu übersetzen und mit entsprechender Ausgabe auf der Konsole zu erkennen aus welchem Land diese stammt,.

6.4 Vergleich

Bis auf einige syntaktische Unterschiede, ist die Umsetzung des Adapter Pattern zu JavaScript sehr ähnlich.

- In Python muss bei einer Instanzmethode die Referenz der Instanz selbst, als Parameter übergeben werden. Dieser Parameter wird konventionell `self` benannt. Über `self` erzeugt und greift man auf alle Eigenschaften der Instanz zu. In JavaScript muss das Schlüsselwort `this` nicht als Parameter übergeben werden. Der Zugriff und die Erzeugung auf Eigenschaften der Instanz, erfolgt durch das Schlüsselwort `this`.
- Mit `let`, `var` und `const` können in JavaScript Variablen erzeugt werden. In Python werden Variablen einfach nur mit dem Variablennamen deklariert.

7 Observer

Das Observer Pattern gliedert sich in die Kategorie der Verhaltensmuster ein. Verhaltensmuster beschreiben wie sich bestimmte Objekte und Funktionen in verschiedenen Situation verhalten sollen. Dies geschieht durch das Schaffen von Abhängigkeiten und Delegation der Aufgaben. Das Observer Pattern zu deutsch, Beobachtermuster, verwendet ein Konzept, in dem es mehrere Beobachter und einen Beobachtenden hat. Ein Observer, der Beobachter, meldet sich bei einer Funktionalität an, um bei Änderungen oder Neuigkeiten benachrichtigt zu werden. Ein Subject, der Beobachtende, benachrichtigt stets jeden Observer bei Veränderungen. Genau so wie sich Abonennten zu einem Newsletter anmelden können, um regelmäßig mit Neuigkeiten benachrichtigt zu werden. Ein anderes Beispiel wäre eine Software zur visuellen Ausgabe verschiedener statistischer Daten. Nutzt diese Software mehrere Diagramme zur Visualisierung, muss jedes einzelne Diagramm über die Veränderung der Daten benachrichtigt werden. Anhand dieser Benachrichtigungen erfolgt die Aktualisierung der Diagramme mit den neuen Daten.

Die Grundidee des Observer Patterns ist es eine lose Abhängigkeit unter Klassen zu schaffen, welche aufgrund dieser losen Kopplung immer wiederverwendet werden können.

Dieses Entwurfsmuster findet häufige Verwendung in der Praxis. Sehr gute Beispiele des Observer Patterns sind Bibliotheken und Frameworks für grafische Oberflächen wie React.js[23] oder Angular[24].

Sie setzen das Observer Pattern in sehr geschickten Varianten um.

7.1 Herangehensweise

7.1.1 Teilnehmer

- *Subject* - speichert, entfernt und fügt Observer hinzu
- *Observer* - bietet eine Schnittstelle für Objekte, um von Zustandsänderungen des Subject benachrichtigt zu werden
- *ConcreteObserver* (State) - speichert Zustand für konkrete Observer und benachrichtigt alle registrierten Observer bei verändertem Zustand
- *ConcreteSubject* (UserCount, UserList, UserOutput) - setzt die Schnittstelle für Zustandsveränderungen um, speichert Zustandsveränderungen und führt Funktionen mit neuem Zustand aus[25]

Der folgende Quellcode soll eine Implementierung des Observer Pattern in JavaScript veranschaulichen.

7.2 JavaScript

7.2.1 Quelltext

```
1  class Subject{  //Subject
2      constructor(){
3        this.observers = [];
4        }
5
6      addObserver(observer){
7      this.observers.push(observer);
8      }
9
10     removeObserver(observer){
11       this.observers = this.observers.filter(function(item){
12         if(item !== observer)
13             return item;
14       });
15     }
```

```
16
17      notify(data){
18        if (this.observers.length > 0){
19          for(let observer of this.observers){
20            observer.update(data);
21          }
22        }else{
23          console.log("Keine Observer registriert")
24        }
25      }
26    }
27
28  class Observer{
29
30    update(){}
31  }
32
33  class State extends Subject{
34    constructor(){
35      super();
36      this.state = [];
37    }
38
39    addPerson(person){
40      this.state.push(person);
41      this.notify(this.state);
42    }
43    getState(){
44      return this.state;
45    }
46  }
47
48  class PersonOutput extends Observer{
49
50    update(state){
51      console.log("Person ", state[state.length -1].name,"
        ↪  Hinzugefügt");
52    }
53  }
54
55  class PersonCount extends Observer{
56
```

```
57    update(state){
58      console.log("Anzahl der Personen: " , state.length);
59    }
60  }
61   class PersonList extends Observer{
62
63    update(state){
64      for(let person of state)
65        console.log(user.name);
66    }
67   }
68
69
70  const state = new State();
71  const personOutput = new PersonOutput();
72  const personCount = new PersonCount();
73  const personList = new PersonList();
74
75  state.addObserver(personOutput);
76  state.addObserver(personCount);
77  state.addObserver(personList);
78  state.addPerson({name: "Alvin"});
79  state.addPerson({name: "Simon"});
80  state.removeObserver(personCount);
```

7.2.2 Erläuterung

Mit diesem Programmbeispiel können wir Personen zu einer Liste hinzufügen, entfernen und aufzählen. Bei jeder Ausführung einer dieser Funktionen, werden Programmteile benachrichtigt, welche die Änderungen im Bestand der Personen auf der Konsole ausgeben. Diese Programmteile fungieren sozusagen als visuelle Benachrichtigung, wenn die genannten Aktionen ausgeführt werden.

Das Programm besteht aus sechs Klassen.

Die Klasse `Subject` kann Observer registrieren, entfernen und besitzt eine Methode, um alle registrierten Observer mit Änderungen zu benachrichtigen.

Die Klasse `Observer`, implementiert die Schnittstelle, um über alle Änderungen benachrichtigt zu werden.

Die Klasse `State` fügt die Personen hinzu, speichert den aktuellen Zustand mit der Liste der Personen und benachrichtigt alle regi-

strierten Observer mit dem aktuellen Zustand.

Die drei Klassen `UserOutput`, `UserList` und `UserCount` dienen dazu, den hinzugefügten Personen die Anzahl der Personen und alle aktuellen Personen selbst auf der Konsole auszugeben.

Kommen wir nun zur schrittweisen Ausführung des Programmes. Von Zeile 70-73 erzeugen wir jeweils eine Instanz des konkreten Subject und die drei konkreten Observer. Für die Klasse `Subject` und `Observer` erzeugen wir keine Instanz da, wir von diesen zwei Klassen erben:

```
const state = new State();
const personOutput = new PersonOutput();
const personCount = new PersonCount();
const personList = new PersonList();
```

Als erstes fügen wir den Observer `userOutput` hinzu. Das geschieht in dem wir der Methode `addObserver()`, die Instanz `userOutput` der Observer Klasse `UserOutput` als Parameter übergeben. Die Methode `addObserver` um Observer zu registrieren, befindet sich in der Klasse `Subject`. Da aber die Klasse `State` alle Eigenschaften erbt, rufen wir die Methode über die `state` Instanz auf. Die Methode `addObserver()` nimmt als Parameter einen Observer entgegen und fügt diesen mit der `push` Methode an das Ende eines Arrays an. Dieses Array befindet sich im Konstruktor der Klasse `Subject`, in dem alle Observer gespeichert werden. Ebenso registrieren wir die beiden anderen Instanzen `userCount` und `userList` als Observer.

Wir fügen nun mit der Methode `addPerson()`, eine Person in Form eines JavaScript Objektes hinzu. Wir übergeben also der `addPerson()` Methode, die Person `Alvin`. Geschieht dies, wird die Person in der Klasse `State` zu einem Array hinzugefügt, welches die aktuellen Personen speichert. Gleich darauf wird die Methode `notify()` mit dem Array der aktuellen Person `Alvin` aufgerufen. Diese Methode ist eine geerbte Funktion der Klasse `Subject`. Mit dem `extends` Schlüsselwort und der `super()` Methode im Konstruktor, erben wir alle Eigenschaften der Klasse `Subject` und können somit problemlos die `notify()` Methode aufrufen. Diese Methode iteriert über alle registrierten Observer und ruft für jeden Observer die `update()` Methode auf und übergibt mit dem Parameter `state` die aktuelle Liste der Personen. Zuvor fangen wir noch den Fall ab, in welchem noch keine Observer registriert sind, sprich das Array leer ist. Wichtig ist, dass jeder Observer die Methode `update()` implementiert, die von der Observer Klasse geerbt und mit der gewünschten Funktionalität überschrieben bzw. ausgefüllt werden soll. Schließlich wird jede Änderung in

der Liste der Personen, durch die Methode `update()`, an jeden Observer mitgeteilt.

Diedrei Observer haben jeweils eigene Funktionalitäten. Der Observer `userOutput`, also die Instanz der Klasse `UserOutput`, gibt die Person die soeben hinzugefügt wurde, auf der Konsole aus. In JavaScript greift man mit `state[state.length -1]` auf das letzte Element des Arrays zu, was der zuletzt hinzugefügten Person entspricht. Mit dem Punkoperator greifen wir auf den Wert des JavaScript Objektes, dem Namen zu. Direkt danach wird der Observer `userCount` geupdated. Diese Methode `update()`, gibt die Anzahl der Personen auf der Konsole aus. Mit `state.length` geben wir die Länge des Arrays aus, was der Anzahl der Personen entspricht. Der dritte und letzte Observer `userList`, gibt alle Personen auf dem Bildschirm aus. Dies geschieht mit der Iteration über jedes Element des Arrays und der damit aufgerufenen Methode `console.log(user.name)`, welche bei jeder Iteration den Namen der Person ausgibt.

Hier die entsprechende Ausgabe auf der Konsole:

```
bash@terminal:~/Patterns/src$ node observer.js
Person Alvin hinzugefügt
Anzahl der Personen: 1
Liste der Personen:
Alvin
```

Fügen wir in Zeile 79 mit:

```
state.addPerson(name: "Simon");
```

eine neue Person hinzu, werden wieder alle drei Observer benachrichtigt und es folgt die Ausgabe auf der Konsole:

```
bash@terminal:~/Patterns/src$ node observer.js
Person Simon hinzugefügt
Anzahl der Personen: 2
Liste der Personen:
Alvin
Simon
```

Entfernen wir mit:

```
state.removeObserver(personCount);
```

den Observer `personCount` aus dem Array der Observer, ist die Funktionalität mit der die Personen gezählt werden nicht mehr aktiv, da

die Methode `update()` der Klasse `PersonCount` nicht mehr aufgerufen werden kann. Die Methode `removeObserver()` nimmt als Parameter einen Observer entgegen, filtert jedes Element des Arrays mit den Observern, welches nicht dem zu entfernenden Observer entspricht und erzeugt ein neues Array.

Die Funktion `filter()` iteriert implizit durch alle Elements des Arrays `observers` und nimmt eine Callback-Funktion[5] entgegen. Die Callback-Funktion nimmt den Parameter `item` entgegen, welcher einem Element im Array enstpricht. Mit der `if`-Abfrage wird also bei jeder Iteration geprüft, ob das aktuelle Element `item`, dem Element `observer` entspricht, welcher der Methode `removeObserver()` übergeben wurde. Trifft der Fall nicht ein, wird mit dem Rückgabewert das Array neu initialisiert. Ist die Bedingung wahr, gibt es in dieser Iteration keinen Rückgabewert und das Element, der Observer, wird nicht im Array gespeichert.

Die Umsetzung in Python folgt im nächsten Kapitel.

7.3 Python

7.3.1 Quelltext

```
1  class Subject:
2
3      def __init__(self):
4          self.observers = []
5
6      def add_observer(self, observer):
7          self.observers.append(observer)
8
9      def remove_observer(self, observer):
10         self.observers.remove(observer)
11
12     def notify(self, data):
13         if(len(self.observers) > 0):
14             for observer in self.observers:
15                 observer.update(data)
16         else:
17             print("Keine Observer registriert")
18
```

[5] Eine Callback-Funktion ist eine Funktion die als Parameter einer anderen Funktion oder Methode übergeben wird.

```
19  class Observer:
20
21      def update():
22          pass
23
24  class State(Subject):
25      def __init__(self):
26          super().__init__()
27          self.state = []
28
29      def add_person(self, person):
30          self.state.append(person)
31          self.notify(self.state)
32
33      def get_state():
34          return self.state
35
36  class PersonOutput(Observer):
37
38      def update(self, state):
39          print("Person ", state[-1]["name"], " Hinzugefügt")
40
41  class PersonCount(Observer):
42
43      def update(self, state):
44         return print("Anzahl der Personen: ", len(state))
45
46  class PersonList(Observer):
47      def update(self, state):
48          print("Liste der Personen: ")
49          for person in state:
50              print(person["name"])
51
52
53  state = State()
54  person_output = PersonOutput()
55  person_count  = PersonCount()
56  person_list = PersonList()
57
58  state.add_observer(person_output)
59  state.add_observer(person_count)
60  state.add_observer(person_list)
```

```
61 state.add_person({"name" : "Alvin"})
62 state.add_person({"name" : "Simon"})
```

7.3.2 Erläuterung

In diesem Python Programmcode stehen uns wieder die sechs Klassen zu Verfügung. Zusammengesetzt aus dem Subject, Observer, drei konkreten Observer und dem konkreten Subject.

Wir erzeugen von Zeile 57-60 die Instanzen der Klassen `State`, `PersonOutput`, `PersonCount` und `PersonList`. Die Klasse Subject und Observer benötigen keine direkte Instanziierung da von ihnen geerbt wird.

Beginnen wir mit der ersten Funktionalität des Programmes indem drei Observer registriert werden. Diese drei Observer sollen ab dem Zeitpunkt der Registrierung über Zustandsveränderungen benachrichtigt werden. Über die Instanz `state` rufen wir die geerbte Methode `add_observer()` der Klasse `Subject` auf und übergeben dieser Methode nacheinander die Observer `person_output`, `person_count` und `person_list`. Mit dem Befehl `class State(Subject):` geben wir an, von welcher Klasse geerbt werden soll und mit `super().__init__()` erben wir explizit den Konstruktor der Elternklasse `Subject` und somit alle Eigenschaften.

Kommen wir nun zur eigentlichen Funktionalität des Programmes, nämlich Personen zu einer Liste hinzuzufügen:

```
def add_person(self, person):
        self.state.append(person)
        self.notify(self.state)
```

Dies realisieren wir mit der Methode `add_person()`, die sich in der Klasse `State` befindet. Schauen wir uns die Klasse `State` an, können wir sehen, dass sich im Kontruktor ein Array mit dem Namen `state` befindet. Dieses Array hält den aktuellen Zustand des Programmes. Genauer gesagt, die Liste mit den hinzugefügten Personen. Die Methode `add_person()`, nimmt zwei Parameter entgegen und zwar die Referenz zur Instanz selbst und eine Person, in diesem Fall das Objekt `{name:Alvin}`. Mit `self.state.append(person)` wird bei jedem Aufruf, mit einer Person als Parameter, eine Person zur Liste `state` hinzugefügt. Eine Liste in Python verhält sich wie ein dynamisches Array. Eine weitere Funktion der Methode `add_person()`, ist

der zusätzliche Aufruf der Methode `notify(self.state)`, deren Deklaration sich in der Klasse `Subject` befindet. Sie wird mit der Liste an aktuellen Personen als Parameter aufgerufen. Der Aufruf dieser Benachrichtigungs-Methode, muss über das Schlüsselwort `self` geschehen. Da wir in der Klasse `State` alle Eigenschaften der Klasse `Subject` über den Konstruktor erben, erfolgt der Zugriff über `self`, welches die Instanz mit all den Eigenschaften ist, auf die mit `self` zugegriffen werden kann.

Die Methode `notify()` in der Klasse `Subject`, nimmt die Liste mit der neu hinzugefügten Person entgegen, iteriert durch alle Observer und benachrichtigt diese, dass die Person `Alvin` hinzugefügt wurde. Dies ist möglich, da alle Observer die Methode `update()` der Klasse `Observer` erben und bei jeder Iteration aufgerufen wird. In der Methode `notify()` wird zuvor noch geprüft ob es denn schon einen Observer in der Liste `observers` gibt. Ist das der Fall kann die Schleife ausgeführt und alle Observer mit dem neuen Zustand benachrichtigt werden. Ist das nicht der Fall, springt das Programm in die `else`-Verzweigung und es wird eine Warnung auf der Konsole ausgegeben.

Zur Erläuterung der Funktionen der drei Observer schauen wir uns zuerst die Klasse `Observer` an, welche die besagte Methode `update()` implementiert. Diese Methode wird mit der Funktionalität, die zuletzt hinzugefügte Person auf der Konsole auszugeben, überschrieben. Um über eine hinzugefügte Person benachrichtigt zu werden, nimmt die Methode den Parameter `state` entgegen. Zugriff auf das letzte Element einer Liste erfolgt in Python mit dem Index -1 in eckigen Klammern. Mit der zweiten eckigen Klammern greifen wir dann auf den Wert des Python Name/Werte-Paar-Objektes, `name` zu. Dieses Array wird dann bei jeder neu hnzugefügten Person auf der Konsole ausgegeben.

Der zweite Observer `PersonCount` erbt genau so von der Klasse `Observer` und implementiert ebenso die Methode `update()`. Diese Methode nimmt wieder den Zustand entgegen und gibt die Länge der im Zustand befindlichen Liste, mit der Methode `len()` aus. Damit werden bei jedem Aufruf der Methode `update()`, die Anzahl der aktuellen Personen ausgegeben.

Der dritte Observer `PersonList` gibt die Liste der hinzugefügten Personen aus. Der Zustand `state` wird entgegengenommen und es wird durch die Liste iteriert. Die Variable `person` entspricht einem Element in der Liste. Bei jeder Iteration wird der Name des Elements an der aktuellen Stelle in der Liste ausgegeben.

Es wurde gezeigt wie Observer registriert werden, doch nicht wie sie deregistriert werden können. Die Entfernung eines Observer aus

der Liste mit Observer, erfolgt mit der Methode `remove_observer()` die sich in der Klasse `Subject` befindet. Diese rufen wir über die Instanz `state` auf und übergeben der Methode einen Observer als Parameter. `remove_observer()` aus der Klasse `Subject` nimmt einen Observer entgegen und entfernt ihn mit der Methode `remove()` . Diese Methode entfernt ein Element aus der Liste und erzeugt implizit ein neue Liste. Wird zum Beispiel der Observer `personList` aus den Observer entfernt, gibt es keine Ausgabe der aktuellen Personen mehr auf dem Bildschirm. Die Methode `update()` in der Klasse `PersonList`, wird demnach nicht nicht mehr aufgerufen und es folgen keine Benachrichtigungen an diese Klasse.

7.4 Vergleich

Es ist zu sehen, dass die Umsetzung des Observer Enwurfsmusters beider Programmiersprachen ähneln. Mit Kenntnis in nur einer dieser Sprachen, schafft man es, bis auf einige Eigenheiten, das Entwurfsmuster realtiv mühelos in die andere Sprache umzusetzen.

- Die Vererbung aller Instanzeigenschaften erfolgt in JavaScript durch Vererbung des Konstruktors mit dem Schlüsselwort `super()`. In Python muss der Konstruktor mit `super().__init__()` explizit aufgerufen werden.

- Python schafft es mit der einfachen Methode `remove()`, Element aus einer Liste zu entfernen. Eine Liste ist wie ein dynamisches Array, doch mit vielen darauf anwendbaren Methoden. In JavaScript gestaltet sich das Entfernen eines Elements anders. Mit der Methode `filter()`, welche eine weitere Funktion entgegennimmt, werden Elemente aus einem Array ausgewählt. Diese Methode mag komplexer erscheinen, bietet aber durch Entgegennahme einer Funktion mehr Möglichkeiten. Ebenso besitzt auch Python die Methode `filter()`. In unserem Beispiel jedoch bietet sich eher die Methode `remove()` an. Bei komplexeren Algortihmen ist die Methode `filter()` durchaus hilfreich.

- Um das letzte Element einer Liste in Python zu ermitteln, reicht es den Index `-1` in einer eckigen Klammer anzugeben. Python besitzt im Allgemeinen sehr viele Möglichkeiten Listen zu manipulieren. In JavaScript greift man mit `state.length -1` auf das letzte Element zu. Man ruft also die Eigenschaft `length` des Arrays auf, welche der Gesamtlänge des Arrays entspricht. Da ein Array bei Index 0 beginnt, wird mit `length -1` innerhalb der eckigen Klammer auf das Element zugegriffen.

8 Zusammenfassung

In dieser Projektarbeit sind anhand von Entwurfsmuster, Vorgehensweisen aufgezeigt worden, die helfen wartbaren und wiederverwendbaren Quelltext zu entwickeln.

Entwurfsmuster sind Lösungen für wiederkehrende bekannte Probleme.

Für die Umsetzung wurden bewusst die Programmiersprachen JavaScript und Python ausgewählt. Sie sind Bestandteil sehr vieler Unternehmen und in unserer Welt der immer fortschreitenden Digitalisierung nicht wegzudenken.

Besonders in JavaScript wurde eine klassenbasierte Umsetzung verfolgt, um so auf einen gemeinsamen Nenner mit der Programmiersprache Python zu kommen. Ziel war es, mit der Implementierung von Entwurfsmuster beider Sprachen, den Einstieg von einer in die andere Programmiersprache zu vereinfachen.

Die zwei Kriterien an denen sich Entwurfsmuster klassifizieren lassen sind zum einen, ob sich diese auf den Gültigkeitsbereich beziehen und zum anderen, was deren Absicht ist. Die vorgestellten Entwurfsmuster beziehten sich auf die Absicht, welche auch Objekt Pattern genannt werden.

Es wurden drei weitverbreitete Design Patterns vorgestellt. Das **Singleton** Pattern, welches sich in die Kategorie der erzeugenden Verhaltenmuster einreiht, das **Adapter** Pattern, in der Kategorie der strukturellen Entwurfsmuster und das **Observer** Pattern, welches ein Entwurfsmuster in der Kategorie der Verhaltensmuster ist.

Erzeugende Muster beschäftigen sich mit Mechanismen der Objekterzeugung. Strukturelle Muster vereinfachen das Design wie Objekte im Zusammenhang stehen. Verhaltensmuster reaslisieren eine anschauliche Kommunikaton zwischen Klassen und Objekten.

Das Singelton Pattern zielt darauf ab, nur eine einzige Instanzierung der Wichtigsten Klasse im System zu erlauben. Bei weiteren Instanziierungen wird man darauf hingwiesen, dass es sich um eine Singleton Klasse handelt und jede darauffolgend erzeugte Instanz verweist auf die zuerst erzeugte Instanz, dem **Singleton**.

Die *Singleton* Klasse selbst ist die einzige Komponente des Systems.

Beim Vergleich in der Umsetzung des **Singleton** Entwurfsmusters mit JavaScript und Python wurde herausgehoben, dass es sich bei Klassen in JavaScript nur um eine syntaktische Verschönerung handelt. Seit dem ES6 Standart wurden diese eingeführt und passen sich letztendlich an andere objektorientierte Programmierspra-

chen an. Im Gegensatz zu JavaScript besitzt Python zwei Konstruktoren, einen beliebigen und einen festen Rückgabewert. Für die Umsetzung des Singleton Entwurfsmusters benötigt man den `def __new__()` Konstruktor, damit die Referenz zum Singleton als Rückgabewert angegeben werden kann. Echte Klasseneigenschaften gibt es nur in Python, in JavaScript sollen diese bald eingeführt werden. Speicheradressen sind in Python einsehbar, JavaScript vergibt Speicheradressen zur Laufzeit und damit können sich die Adressen sets ändern.

Das **Adapter** Pattern, auch bekannt als Wrapper Pattern, verfolgt den Ansatz ein neues System mit einem alten kompatibel zu machen und vice versa.

Hauptkomponeten sind *Ziel*, *Adapter*, *Adaptierender* und *Nutzer*.

Der Unterschied einer Instanzmethode in Python zu JavaScript ist, die Übergabe der Referenz zur Instanz selbst als Parameter, um Instanzeigenschaften erzeugen und auf diese zugreifen zu können. `let`, `var` und `const` sind die drei Wege in JavaScript Variablen zu deklarieren.

Das **Observer** Pattern ist zuständig für regelmäßige Benachrichtigungen an alle registrierten Teilnehmer bei bestimmten Veränderungen.

Teilnehmende Komponenten sind *Subject*, *ConcreteSubject*, *Observer* und *ConcreteObserver* .

Der Vergleich zeigte, dass JavaScript implizit den Kontruktor miterbt, Python muss diesen bei der Vererbung explizit angeben. Um Elemente aus einem Array zu entfernen, gibt es in Python eine einfache Operation `remove()`, in JavaScript ist das durch die Methode `filter()`, mit etwas mehr Aufwand verbunden. Python bietet viele Möglichkeiten zur Datenmanipulation.

9 Fazit

Deign Patterns sind nicht wirklich eine Revolution oder eine neue Errungenschaft in der Wissenschaft. Viel mehr bündeln Design Patterns Lösungen für Probleme, mit denen Softwareentwickler seit Anbeginn der Programmierung konfrontiert sind. Durch Dokumentation der Herangehensweisen an programmiertechnische Probleme, konnten sich Design Pattern entwickeln und werden sich insbesondere im Bezug auf spezifische Programmiersprachen immer weiterentwickeln. Sie helfen den Quellcode besser zu strukturieren, zu warten und wiederzuverwenden. Je mehr Programmierer in einem Team, Kenntnisse der Design Patterns besitzen, desto effizienter

kann ein Projekt umgesetzt werden. Genau dann, kann man sich darüber unterhalten wann und wo Design Patterns angewendet werden sollten. Die Fähigkeit, mit Verständnis von Design Patterns, sich schneller in bestehenden Programmcode einzulesen, resultiert in Zeitersparnissen . Das ist es was durch diese Projektarbeit erreicht werden soll. Ist man zum Beispiel ein Python Programmierer ohne Kenntnisse der Programmiersprache JavaScript, doch man kennt sich gut mit Design Patterns aus, findet man einen guten Einstieg in JavaScript allein nur durch die Kenntnis der Entwurfsmuster. Die wachsende Beliebtheit von Bibliotheken und Frameworks wie wie Django, Flask, React, AngularJS, Node.js etc., welche alle Design Patterns verwenden, machen es mehr denn je notwendig, Design Patterns zu identifizieren und sie zu nutzen.

Ich möchte Entwickler ermutigen sich mit Enturfsmuster auseinanderzusetzen. Denn der eigentliche Grund dieser Projektarbeit war die Frustration über Programmierprobleme vor denen ich stand, wo ich persönlich im Vorhinhein eigentlich dachte, es wäre trivial diese Probleme zu lösen. Ich verbrachte extrem viel Zeit im Internet nach Lösungen für diese Probleme zu suchen und merkte irgendwann, dass Design Pattern sich genau mit diesen Problemen auseinandersetzen vor denen ich stand. Sie beschreiben wirklich Kernprobleme die immer wieder auftauchen. Durch die Design Patterns fange ich an fremden Code viel besser zu verstehen, da viele Programme inuitiv auf Design Patterns basieren. Zudem konnte ich mir einen tieferen Blick in die Kunst des Programmieren selbst verschaffen. Ich bin der Meinung, dass Entwurfsmuster mir den wahren Kern der objektorientierten Programmierung enthüllen.

Ich hoffe sehr, dass ich mit den aufgezeigten Beispielen einen Einblick in die Welt der Entwurfsmuster und den beiden Programmiersprachen JavaScript und Python geben konnte.

Für diejenigen die sich tiefergehend mit Design Patterns und der Umsetzung in verschiedenen Programmiersprachen beschäftigen wollen, empfehle ich zwei sehr interessante wissenschaftliche Arbeiten von Vladislav Georgiev Alfredov und Kristian Pederson. Sie beschäftigen sich mit der Fragestellung, inwiefern sich die Implementierung verschiedener Programmiersprachen auf Design Patterns auswirken.[26][27]

Literaturverzeichnis

[1]S. Overflow, *Stack Overflow Annual Developer Survey.* (2020) https://insights.stackoverflow.com/survey/2020#most-popular-technologies.

[2]Tiobe, *Tiobe Index.* (2020) https://www.tiobe.com/tiobe-index/.

[3]OpenJS, *Node.js.* (2020) https://nodejs.org/de/.

[4]E. Gamma, R. Helm, R. Johnson und J. Vlissides, *Design Patterns, Elements of Reusable Object-Oriented Software* (Addison-Wesley, 1995).

[5]MDN, *ECMAScript 2015 Sprachspezifikation.* (2015) https://developer.mozilla.org/de/docs/Web/JavaScript/Language_Resources.

[6]Google, *TensorFlow.* (2020) https://www.tensorflow.org/.

[7]D. Cournapeau, *scikit-learn.* (2020) https://scikit-learn.org/stable/.

[8]Django, *Django Project.* (2020) https://www.djangoproject.com/community/.

[9]A. Ronacher, *Flask.* (2020) https://palletsprojects.com/p/flask/.

[10]A. Christopher, „A Pattern Language: Towns, Buildings, Construction.“, in (Oxford University Press, 1977) Kap. x.

[11]S. Stefanov, *JavaScript Patterns* (O'Reilly, 2010).

[12]A. Osmani, *JavaScript Design Patterns* (O'Reilly, 2017).

[13]OpenJS, *Python3 Download.* (2020) https://www.python.org/downloads/.

[14]OpenJS, *Node.js Download.* (2020) https://nodejs.org/en/download/.

[15]E. Gamma, R. Helm, R. Johnson und J. Vlissides, „Design Patterns, Elements of Reusable Object-Oriented Software“, in, Bd. 1, Participants (Addison-Wesley, 1995) Kap. 4, S. 141.

[16]MDN, *Speicherverwaltung.* (2020) https://developer.mozilla.org/de/docs/Web/JavaScript/Speicherverwaltung.

[17]Python, *Data Model.* (2020) https://docs.python.org/3/reference/datamodel.html#object.__new__.

[18]*ECMAScript*, (2019) https://www.ecma-international.org/publications/standards/Ecma-262.htm.

[19]MDN, *Class fields.* (2020) https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/Public_class_fields.

[20]TC39, *Class fields proposal.* (2020) https://tc39.es/proposal-class-fields/#prod-FieldDefinition.

[21]Mozilla, *Mozilla Developer Network.* (2020) https://developer.mozilla.org/de/.

[22]E. Gamma, R. Helm, R. Johnson und J. Vlissides, „Design Patterns, Elements of Reusable Object-Oriented Software“, in, Bd. 1, Participants (Addison-Wesley, 1995) Kap. 3, S. 127.

[23]Facebook, *React.* (2020) https://reactjs.org/.

[24]Google, *Angluar.* (2020) https://angular.io/.

[25]E. Gamma, R. Helm, R. Johnson und J. Vlissides, „Design Patterns, Elements of Reusable Object-Oriented Software“, in, Bd. 1, Participants (Addison-Wesley, 1995) Kap. 5, S. 295.

[26]V. G. Alfredov, „How Programming Languages Affect Design Patterns.“, Magisterarb. (University of Oslo, 2016).

[27]K. Pedersen, „How Implementation Language Affects Design Patterns.“, Magisterarb. (University of Oslo, 2019).

Bücher

[4]E. Gamma, R. Helm, R. Johnson und J. Vlissides, *Design Patterns, Elements of Reusable Object-Oriented Software* (Addison-Wesley, 1995).

[11]S. Stefanov, *JavaScript Patterns* (O'Reilly, 2010).

[12]A. Osmani, *JavaScript Design Patterns* (O'Reilly, 2017).

Referenzen im Buch

[10]A. Christopher, „A Pattern Language: Towns, Buildings, Construction.“, in (Oxford University Press, 1977) Kap. x.

[15]E. Gamma, R. Helm, R. Johnson und J. Vlissides, „Design Patterns, Elements of Reusable Object-Oriented Software“, in, Bd. 1, Participants (Addison-Wesley, 1995) Kap. 4, S. 141.

[22]E. Gamma, R. Helm, R. Johnson und J. Vlissides, „Design Patterns, Elements of Reusable Object-Oriented Software“, in, Bd. 1, Participants (Addison-Wesley, 1995) Kap. 3, S. 127.

[25]E. Gamma, R. Helm, R. Johnson und J. Vlissides, „Design Patterns, Elements of Reusable Object-Oriented Software“, in, Bd. 1, Participants (Addison-Wesley, 1995) Kap. 5, S. 295.

Masterthesen

[26]V. G. Alfredov, „How Programming Languages Affect Design Patterns.“, Magisterarb. (University of Oslo, 2016).

[27]K. Pedersen, „How Implementation Language Affects Design Patterns.“, Magisterarb. (University of Oslo, 2019).

Weblinks

[1]S. Overflow, *Stack Overflow Annual Developer Survey.* (2020) https://insights.stackoverflow.com/survey/2020#most-popular-technologies.

[2]Tiobe, *Tiobe Index.* (2020) https://www.tiobe.com/tiobe-index/.

[3]OpenJS, *Node.js.* (2020) https://nodejs.org/de/.

[5]MDN, *ECMAScript 2015 Sprachspezifikation.* (2015) https://developer.mozilla.org/de/docs/Web/JavaScript/Language_Resources.

[6]Google, *TensorFlow.* (2020) https://www.tensorflow.org/.

[7]D. Cournapeau, *scikit-learn.* (2020) https://scikit-learn.org/stable/.

[8]Django, *Django Project.* (2020) https://www.djangoproject.com/community/.

[9]A. Ronacher, *Flask.* (2020) https://palletsprojects.com/p/flask/.

[13]OpenJS, *Python3 Download.* (2020) https://www.python.org/downloads/.

[14]OpenJS, *Node.js Download.* (2020) https://nodejs.org/en/download/.

[16]MDN, *Speicherverwaltung.* (2020) https://developer.mozilla.org/de/docs/Web/JavaScript/Speicherverwaltung.

[17]Python, *Data Model.* (2020) https://docs.python.org/3/reference/datamodel.html#object.__new__.

[18]*ECMAScript*, (2019) https://www.ecma-international.org/publications/standards/Ecma-262.htm.

[19]MDN, *Class fields.* (2020) https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/Public_class_fields.

[20]TC39, *Class fields proposal.* (2020) https://tc39.es/proposal-class-fields/#prod-FieldDefinition.

[21]Mozilla, *Mozilla Developer Network.* (2020) https://developer.mozilla.org/de/.

[23]Facebook, *React.* (2020) https://reactjs.org/.

[24]Google, *Angluar.* (2020) https://angular.io/.

Glossar

dynamisch typiisiert Dynamisch typiisiert bedeutet, dass man bei Deklaration einer Variablen keinen Datentyp zuweisen muss, denn der Datentyp wird zur Laufzeit geprüft. 4

Eigenschaft Eine Eigenschaft ist eine objektgebundene Assoziation zwischen Name und einem Wert. 11

Framework Ein Framework stellt eine gemeinsame und wiederverwendbare Struktur für Applikationen zur Verfügung. Man baut ein Framework in eine Applikation ein, um es den Anforderung entsprechend zu erweitern. 3

Konstruktor Ein Konstruktor ist eine Operation, welche automatisch aufgerufen wird, um neue Instanzen zu initialisieren. 10

Multiparadigmen-Programmiersprache Der Aufbau von Programmiersprachen ist unterschiedlich. Sie haben verschiedene Ansätze und verfolgen jeweils andere Ziele. Sie können prozedual, objekorientiert oder funktional sein. Eine Multiparadigmen-Programmiersprache ist eine Programmiersprache, die mehr als nur eines dieser Paradigmen besitzt. 5

prototypen-basiert JavaScript basiert auf einem prototypenbasierenden Vererbungsmodell bei dem jedes Objekt eine interne Verbindung zu einem anderen "älteren" Objekt besitzt, welches als sein Prototyp bezeichnet wird. Dieses Prototypobjekt hat selbst einen Prototyp, der wiederum selbst einen Prototyp hat. Dies setzt sich fort, bis ein Objekt erreicht wird, dessen Prototyp null ist. null hat per Definition keinen Prototyp und bildet somit den Abschluß dieser Prototypenkette. 8

scikit-learn scikit-learn ist eine freie Software-Biliothek zum maschinellen Lernen für die Programmiersprache Python. 6

Skriptsprachen ECMAScript ist eine Skriptsprachenspezifikation. Sie wurde entwickelt um JavaScript zu standartisieren. 4

TensorFlow TensofFlow ist eine Open-Source Plattform für maschinelles Lernen, entwickelt von Google Brain. 6