Borrow - Inherit - Mix
Code-Wiederverwendung in JavaScript

Seminararbeit im Fach Informatik
Kursnummer 1919 im WiSe 18/19 an der FernUniversität Hagen

Felix Eckstein*

WiSe 2018/19

Betreuerin: Dr. Daniela Keller
Lehrgebiet Programmiersysteme
Fachbereich Informatik

* *Student im Master of Science Informatik an der FernUniversität Hagen,*
Matr.-#: 8161569, eckstein@embedded-engineering.de

INHALTSVERZEICHNIS

1 EINLEITUNG

Durch die in allen Webbrowsern vorhandene Implementierung einer JavaScript-Laufzeitumgebung ist diese Sprache das Mittel der Wahl, um eine wirklich plattformunabhängige Implementierung von Applikationen zu realisieren. Auf jedem Gerät, auf dem ein moderner Webbrowser verfügbar ist, kann eine in JavaScript geschriebene Applikation ausgeführt werden. Durch die Einbettung des Codes in Webseiten ist es für die Benutzerin nicht notwendig, eine solche Applikation erst mühsam zu installieren. Es ist ausreichend die entsprechende Webseite anzusteuern, damit das Programm abläuft.

Seit einigen Jahren hat sich die Sprache aus ihrem angestammten Feld, nämlich Webapplikationen mit einfach zu bedienenden Benutzeroberflächen, weiterentwickelt und wird zunehmend auch im Server-Umfeld oder sogar in embedded Applikationen auf Mikrocontrollern eingesetzt. Mit JavaScript lässt sich heute die vollständige Applikationsentwicklung von der Serverapplikation inklusive Datenhaltung über die Netzwerkkommunikation bis hin zur Benutzeroberfläche im Browser der Anwenderin abdecken.

Diese Breite an Anwendungsmöglichkeiten und die allgegenwärtige Verfügbarkeit von Laufzeitumgebungen und Entwicklungswerkzeugen hat dazu geführt, dass heute kaum eine Programmiererin an JavaScript vorbei kommt.

JavaScript unterstützt den Einstieg für Benutzerinnen, die diese Sprache nur selten einsetzen durch eine einfache, an C und Java angelehnte Syntax und eine objektorientierte Struktur. Zusammen mit den dynamischen Eigenschaften und dem Verzicht auf strenge Typisierung kommen die meisten Einsteigerinnen sehr schnell zu ersten brauchbaren Ergebnissen.

Sobald man JavaScript nicht mehr nur gelegentlich für Mini-Anwendungen benutzt, sondern größere Anwendungen damit entwickeln möchte, führt jedoch –wie in jeder anderen Programmiersprache auch– kein Weg daran vorbei, sich intensiv mit der Sprache auseinanderzusetzen und deren Eigenheiten zu lernen und zu verstehen.

Der größte Unterschied von JavaScript zu vielen anderen landläufig bekannten Programmiersprachen liegt darin, dass JavaScript zwar objektorientiert ist, dabei aber auf Klassen im Sprachkern verzichtet.[1] Durch den Verzicht auf Klassen als Grundlage der Objektdefinition lassen sich viele bekannte Muster aus der klassischen[2] Objektorientierung nicht eins-zu-eins auf JavaScript übertragen. Im Einzelfall kann das dazu führen,

1 Seit dem Sprachstandard ECMAScript 6 (ES6) wurden in JavaScript klassenähnliche Konstrukte und eine klassenähnliche Syntax eingeführt. Diese bauen intern jedoch auch auf der prototypischen Objektorientierung auf, die JavaScript zu eigen ist. Letztendlich handelt es sich bei den in ES6 eingeführten Klassen (fast) nur um syntaktischen Zucker, der die Einstiegshürde für Umsteigerinnen aus klassischen Sprachen senken soll.

2 Im Weiteren wird meist der Begriff *klassisch* anstelle von *klassenbasiert* verwendet, da in der englischsprachigen Literatur meist von *classical* und nicht von *class based* die Rede ist. Diese Begrifflichkeit soll hier übernommen werden. Damit ist auf keinen Fall *klassisch* im Sinne von *althergebracht* gemeint.

dass Muster in Javascript etwas komplizierter werden, während in vielen anderen Fällen die Objekterzeugung ohne Klassendefinition große Vorteile bringt. Im Einzelfall ist daher zu untersuchen, wie bestimmte Muster nach JavaScript übertragen werden können, oder ob es in JavaScript nicht andere, bessere Möglichkeiten zur Problemlösung gibt.

In der vorliegenden Arbeit soll ein Überblick darüber gegeben werden, wie, aufbauend auf dem JavaScript Objektsystem, eine Code-Wiederverwendung stattfinden kann. Es wird zunächst ein Überblick gegeben, wie Objekte ohne Klassendefinitionen in JavaScript erzeugt werden. Darauf aufbauend werden verschiedene Methoden vorgestellt, wie Verhalten, das für ein Objekt entwickelt wurde, auch für andere und andersartige Objekte wieder verwendet werden kann. Die betrachteten Techniken sind:

- Method Borrowing
- prototypische Vererbung und Objekt-Mixins
- functional Mixins

Abschließend sollen die gezeigten Techniken kritisch betrachtet und bewertet werden.

Da es in dieser Arbeit vor allem darum gehen soll, wie sich die anzuwendenden Techniken der Code-Wiederverwendung in einer prototypischen Sprache wie JavaScript von den bekannteren Techniken in klassischen Sprachen unterscheiden, werden die mit ECMAScript 6 (ES6) eingeführten Klassen nicht weiter betrachtet. Eine kurze Zusammenfassung der Kritik an den ES6-Klassen ist in Anhang A zusammengestellt. Eine kritische Würdigung der ES6-Klassen und ihrer Vor- und Nachteile gäbe genug Stoff für mindestens eine eigenständige Seminararbeit.

2 JAVASCRIPT GRUNDLAGEN

Bevor die JavaScript Spezifika der Code-Wiederverwendung besprochen werden können, müssen einige Besonderheiten der Programmiersprache und des damit einhergehenden Programmierparadigmas besprochen werden.

JavaScript ist eine objektorientierte (OO) Programmiersprache, die jedoch im Gegensatz zur großen Mehrzahl anderer Sprachen nicht *klassenbasiert* sondern *prototypenbasiert* ist. Neben dem objektorientierten Paradigma kann JavaScript auch als funktionale Programmiersprache aufgefasst werden, da Funktionen in JavaScript *normale* Objekte, und damit „1st-class values“ sind. Sie können wie jedes andere Objekt an Variablen gebunden werden und damit sowohl als Parameter als auch als Rückgabewerte von Funktionsaufrufen verwendet werden.

Auf diese beiden Eigenschaften und die daraus resultierenden Konsequenzen für die Wiederverwendung von Code soll im folgenden einleitenden Abschnitt kurz eingegangen werden.

2.1 Prototypische Objektorientierung

Die meisten landläufig bekannten OO-Programmiersprachen basieren auf Klassen[3]. Dazu gehört auch die an der FernUni Hagen als Standard-OO-Sprache gelehrte Sprache Java.

In diesen klassischen OO-Sprachen wird zunächst eine Klasse definiert, die als Blaupause, bzw. abstraktes Modell für Objekte eines bestimmten Typs dient. Wenn die Programmiererin ein konkretes Objekt –eine Instanz– benötigt, so wird es anhand dieses vorher definierten Bauplans erstellt. Es handelt sich damit um die „materialisierte Kopie" des Bauplans. In der klassischen Objektorientierung gibt es kein Objekt, zu dem nicht im Vorfeld eine Klasse definiert wurde (siehe [Simpson, 2014, p. 69]).

Im Gegensatz dazu stehen Objekte in prototypbasierten Sprachen für sich allein. Sie können „out of thin air" ([Simpson, 2014, p. 21]) erzeugt werden, ohne dass vorher ein Modell des Objekts definiert werden muss.

Ein Objekt in JavaScript selber ist eine Sammlung von benannten *Properties*, denen Werte zugeordnet werden können. Die Struktur ist damit vergleichbar mit einer *Dictionary*-Datenstruktur aus `{key: value}` mit dem Property-Namen als `key`. Diesem Dictonary, und damit dem Objekt, können zur Laufzeit weitere Properties hinzugefügt oder auch wieder daraus gelöscht werden.

Die Werte der Properties sind entweder *Referenzen* auf ein weiteres Objekt oder ein *primitiver Wert*. Als primitive Werte sind in JavaScript lediglich die Typen `Undefined`, `Null`, `Boolean`, `Number`, `Symbol`, und `String` definiert (siehe [TC39 und Terlson, 2018, §4.3.2]). Alle anderen Werte, und insbesondere Funktionen, sind selber Objekte, auf die eine Property referenzieren kann. Bei Funktionen, die einer Objekt-Property zugeordnet sind, spricht man von *Methoden*.

2.2 Die Prototype-Chain

JavaScript hat einen eingebauten Delegationsmechanismus bezüglich des Zugriffs auf Objekt-Properties. Dazu hat jedes Objekt eine interne Referenz –in der Sprachspezifikation [TC39 und Terlson, 2018, §9.1] als *[[Prototype]]*-Slot bezeichnet– die entweder auf ein anderes Objekt zeigt oder den primitiven Wert `null` enthält. Das referenzierte Objekt wird *Prototyp* genannt. Da ein so referenziertes Objekt selber wieder eine Referenz auf einen eigenen Prototypen enthält, ergibt sich eine verkette Liste aus Objekt-

3 Eine Übersicht kann hier der Wikipedia Eintrag `https://en.wikipedia.org/wiki/List_of_programming_languages_by_type%23Object-oriented_class-based_languages` liefern.

Referenzen, die als *Prototype-Chain* bezeichnet wird. In der Regel endet die Prototype-Chain im Objekt `Object.prototype`, auf dem einige nützliche Hilfsfunktionen (z. B. `toString() oder valueOf()`) als Properties implementiert sind.

Der Prototyp eines Objekts wird automatisch bei der Objekterzeugung gesetzt und kann über `Object.getPrototypeOf()` abgefragt bzw. über `Object.setPrototypeOf()` geändert werden. Wie die automatische Setzung erfolgt, wird später im Abschnitt zur Objekterzeugung detailliert erläutert.

Bei einem lesenden Zugriff oder einem Funktionsaufruf auf eine Objekt-Property wird zunächst durch die Runtime geprüft, ob das Objekt selber eine Property entsprechenden Namens hat. Ist dies nicht der Fall, so folgt die Runtime der [[Prototype]]-Referenz und sucht im Prototyp-Objekt nach der passenden Property. Wenn diese dort gefunden wird, so erfolgt der Zugriff darauf. Andernfalls wird die Suche nach der Property entlang der Prototype-Chain fortgesetzt, bis sie entweder gefunden wird, oder die Kette endet. Dieser automatische Zugriff auf Properties von Objekten, die in der Prototypen-Hierarchie weiter oben stehen, entspricht einer Delegation an ein anderes Objekt und umfasst insbesondere auch Methodenaufrufe.

Diese automatische Delegation erfolgt nur bei lesenden Zugriffen auf Properties. Da schreibende Zugriffe auf eine Property Auswirkungen auf alle, in der Prototype-Chain tiefer liegenden Objekte haben, werden Änderungen per Delegation von einem tiefer liegenden Objekt aus verhindert. Bei einem schreibenden Zugriff auf eine Property, die weiter oben in der Prototype-Chain definiert und nicht als *read-only* markiert ist, wird auf dem Startobjekt der Prototype-Chain eine neue Property des gleichen Namens angelegt. Diese neue Property verdeckt bei darauffolgenden lesenden Zugriffen die weiter oben liegende Property des Prototypen. Man spricht in diesem Fall von *shadowing*.[4]

Änderungen direkt am Prototypen-Objekt dagegen haben Auswirkungen auf alle darauf referenzierenden Objekte, es sei denn, die geänderte Property wird weiter unten in der Hierarchie verdeckt. Daher ist bei der Zuweisung von Werten darauf zu achten, ob man gerade auf eine eigene Property zugreift oder auf eine in der Prototype-Chain weiter oben liegende. Wenn es darauf ankommt zu wissen, ob ein Objekt eine Property selber besitzt, oder per Delegation entlang der Prototype-Chain darauf zugreift, so kann dies über die Methode `obj.hasOwnProperty(prop)` geprüft werden. Diese ist definiert als `Object.prototype.hasOwnProperty()` und wird daher auf einem Objekt `obj` per Delegation an `Object.prototype` aufgerufen. Seit der Version ES6 können die eigenen Properties eines Objekts über die Funktion `Object.keys(obj)` direkt als Array abgefragt werden (siehe [TC39 und Terlson, 2018, §19.1.2.16]).

4 Der Vollständigkeit halber sei erwähnt, dass eine weiter oben in der Chain liegende Property in Ausnahmefällen verändert werden kann, wenn diese nicht als normale Property, sondern über einen *Setter* definiert ist. Dies ist in der Praxis jedoch die Ausnahme und wird daher hier nicht weiter ausgeführt. Details finden sich in [Simpson, 2014, p. 88f.] und in [TC39 und Terlson, 2018].

```
let father = {
    first: "Darth",
    last: "Skywalker",

    sayName: function () {
        console.log(`My name is ${this.first} ${this.last}.`);
    }
}

let son = {};   //empty object with Prototype set to Object
Object.setPrototypeOf(son, father); //performance penalty!!! Better: let son = Object.create(father)
console.log(`son has own keys: [${Object.keys(son)}]`);   //son has own keys: []

son.first = "Luke"; //becomes an own property of son and will shadow father's prop of the same name
son.side = "light side"; //becomes an own property of son

son.sayName();      //My name is Luke Skywalker.
father.sayName();   //My name is Darth Skywalker.

father.last = "Vader"
father.side = "dark side"; //this becomes an own property of father


son.sayName();      //My name is Luke Vader.    //Oops, the last name changed on his Prototype Object
console.log(son.side);  //light side

father.sayName();   //My name is Darth Vader.
console.log(father.side);   //dark side

console.log(`Object.keys(son): [${Object.keys(son)}]`);   //Object.keys(son): [first,side]
console.log(`Object.keys(father): [${Object.keys(father)}]`); //Object.keys(father):
     [first,last,sayName,side]
```

Listing 2.1: Beispiel von Delegation und Shadowing in der Prototype Chain

Zur Verdeutlichung sei hier ein kleines Beispiel angegeben. In Listing 2.1 wird ein `father` und ein `son`-Objekt erstellt. Die [[Prototype]]-Referenz von `son` wird explizit auf `father` gesetzt und hat damit per Delegation Zugriff auf dessen Properties inklusive der Methode `sayName()`.

Zunächst hat `son` keine eigenen Properties, kann aber per Delegation auf die Properties von `father` zugreifen. Die Zuweisung in Zeile 14 erzeugt eine neue Property `first` auf dem Objekt `son`, welche die gleichnamige Property von `father` überdeckt. In Zeile 20 wird `father.name` geändert. Das hat auch (häufig ungewollte) Auswirkungen auf den Property-Lookup des `son`-Objekts aus. In den Zeilen 24-31 sind weitere Ausgaben zu sehen, die das Verhalten verdeutlichen. In Abbildung 2.1 ist die Prototype-Chain ausgehend von `son` zum Programmende schematisch dargestellt.

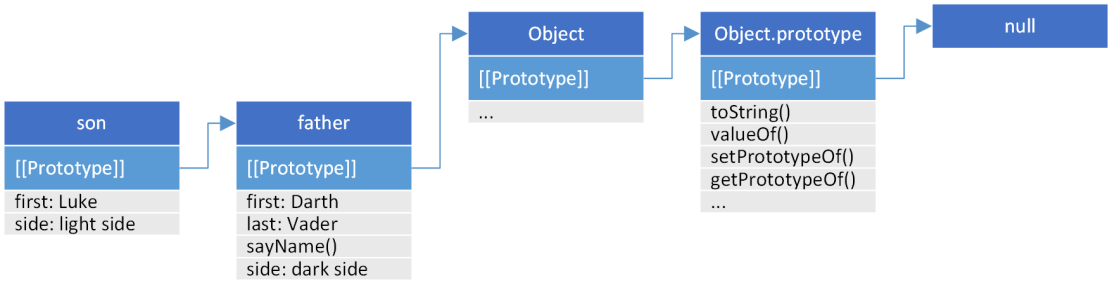

Abbildung 2.1: Die Prototype-Chain der Objekte aus Listing 2.1.

2.3 `this`-Binding

In OO-Sprachen wird eine Möglichkeit benötigt, mit der eine Methode eines Objekts auf die Properties eben dieses Objekts zugreifen kann. In den meisten OO-Programmiersprachen gibt es dazu das Schlüsselwort `this`[5], welches an die gerade aktuelle Objektinstanz gebunden ist.

Auch JavaScript hat das Schlüsselwort `this` definiert, welches bei einem Funktionsaufruf zur Laufzeit an ein Objekt gebunden wird. An welches Objekt `this` gebunden wird, ist davon abhängig wie die Funktion aufgerufen wird. Die konkrete Bindung zur Laufzeit gibt den Kontext vor, in dem die Funktion ausgeführt wird und auf dessen Properties sie Zugriff hat.

Es gibt im Wesentlichen vier Regeln, nach denen die Laufzeitumgebung entscheidet, auf welches Objekt `this` referenziert. Diese Regeln werden der Reihe nach geprüft und die zuerst zutreffende Regel wird verwendet, um die passende Referenz in `this` abzulegen:

1. *new binding*
2. *explicit binding*
3. *implicit binding*
4. *default binding*

Zusätzlich gibt es seit der Sprachversion ES6 noch einen Bindungsmechanismus, welcher die aufgezählten Regeln außer Kraft setzt: ES6 *arrow functions* benutzen einen *lexikalischen Kontext* (*lexical scope*) und damit eine feste `this`-Bindung, die nicht erst zur Laufzeit erstellt wird.

NEW BINDING Wenn eine Funktion als Konstruktorfunktion mit dem Schlüsselwort `new` aufgerufen wird, so erzeugt sie ein neues Objekt (siehe unten 2.4). Innerhalb der Konstruktorfunktion wird `this` an dieses neu erzeugte Objekt gebunden.

5 Der Name des Schlüsselworts variiert in verschiedenen Sprachen.

EXPLICIT BINDING Die Bindung von `this` kann für eine Funktion explizit gesetzt werden. Dazu dienen die eingebauten Funktionen `Function.prototype.call()`, `Function.prototype.apply()` und `Function.prototype.bind()`. Diese Methoden aus `Function.prototype` binden die `this`-Referenz an das als erstes Argument übergebene Objekt. Dies wird am Besten anhand eines kleinen Beispiels deutlich:

```
function speak() {
    var greeting = "Hello, I'm " + this.name;
    console.log(greeting);
}

var me = {
    name: "Felix"
};
var you = {
    name: "Listener"
};

//explizites ad-hoc-Binding
speak.call(me); // Hello, I'm Felix
speak.call(you); // Hello, I'm Listener

//permanentes Binding zur spaeteren Verwendung
var meSpeak = speak.bind(me);
var youSpeak = speak.bind(you);

meSpeak(); // Hello, I'm Felix
youSpeak(); // Hello, I'm Listener
```

Listing 2.2: Beispiel für die Wirkung von `call()` und `bind()`.

In diesem Code wird die `this`-Referenz der Funktion `speak` explizit an verschiedene Objekte gebunden.

IMPLICIT BINDING Wenn eine Funktion direkt auf einem umgebenden Objekt aufgerufen wird, so wird dieses Objekt an `this` gebunden. Diese Bindung geht verloren, wenn die Funktion nicht direkt auf diesem Objekt aufgerufen wird, weil sie z. B. vorher einer anderen Variable zugewiesen wurde (Siehe Beispiel in Listing 2.3).

DEFAULT BINDING Wenn keine dieser drei Regeln zutrifft, dann wird als Binding für `this` entweder `undefined` im *strict mode* oder das *global object* verwendet.

```
var me = {
    speak: function speak() {
        var greeting = "Hello, I'm " + this.name;
        console.log(greeting);
    },
    name: "Felix",
};

//implicit binding
me.speak(); //Hello, I'm Felix
```

```
//default binding
global.name = "Wer?"; //property on global object
var whoSpeak = me.speak;    //binding to me gets lost here for variable whoSpeak
whoSpeak();
//Hello, I'm Wer? //in non-strict mode
//TypeError: this is undefined //in strict-mode
```

Listing 2.3: Beispiel für *implicit binding* und *default binding*.

LEXICAL BINDING In ES6 wurde eine neue kompakte Schreibweise für Funktionsausdrücke eingeführt, die sogenannten *Arrow*-Funktionen. Ihren Namen verdanken sie ihrer Schreibweise mit dem „fat arrow"-Symbol =>. Neben der kompakteren Schreibweise gibt es einen wichtigen semantischen Unterschied zu konventionellen Funktionen da eine neue Regel für die Bindung von `this` hinzu kam: Arrow-Funktionen übernehmen für ihr `this`-Binding das `this`-Binding des sie umgebenden lexikalischen Kontexts zur Definitionszeit (lexical binding). Dadurch ist es möglich ad-hoc definierte Funktionen, wie sie häufig für Callbacks eingesetzt werden, mit einer sinnhaften Bindung zu versehen, ohne dies explizit zu formulieren. Der Unterschied zu traditionellen Funktionen wird in Listing 2.4 deutlich:

```
var obj1 = {
    a: "function",
    callback: function () {
        console.log("this.a = ", this.a);
    },
    foo: function foo() {
        setTimeout(this.callback, 1000) //link to execution context gets lost
    }
}
obj1.foo(); //this.a =  undefined

var obj2 = {
    a: "bind",
    callback: function () {
        console.log("this.a = ", this.a);
    },
    foo: function foo() {
        setTimeout(this.callback.bind(this), 1000) //explicit binding of this
    }
}
obj2.foo(); //this.a =  bind

var obj3 = {
    a: "arrow",
    foo: function foo() {
        setTimeout(() => {
            console.log("this.a = ", this.a); //lexical binding of this
        }, 1000)
    }
}
obj3.foo(); //this.a =  arrow
```

Listing 2.4: Beispiel für das lexical Binding von `this` in Arrow-Funktionen.

2.4 Objekterzeugung

In den vorangegangenen Abschnitten wurde erläutert, was ein Objekt ist, welche Properties es enthält, wie Delegationshierarchien zwischen Objekten über die Prototype-Chain aufgebaut werden können und wie über das Schlüsselwort `this` aus einer Methode auf Properties des Objekts zugegriffen werden kann. Bisher wurden jedoch noch kein Überblick darüber gegeben, wie Objekte erzeugt werden können. Dies soll nun nachgeholt werden.

Auch bei der Objekterzeugung ist es gut, zunächst die weiter verbreiteten klassischen OO-Sprachen zu betrachten, und davon ausgehend die Wege der Objekterzeugung in JavaScript und deren Unterschiede aufzuzeigen.

In klassischen Sprachen ist der Bauplan des Objekts über die Klassendefinition vollständig festgelegt. Zur Laufzeit muss aus diesem Bauplan ein konkretes Objekt instantiiert werden. Dazu wird in den meisten OO-Sprachen eine Konstruktorfunktion zusammen mit dem Schlüsselwort `new` aufgerufen. Daraufhin wird Speicherplatz für eine neue Instanz des in der Klasse definierten Objekts reserviert und die Konstruktorfunktion wird gestartet. In der Konstruktorfunktion können Initialisierungen der Properties des Objekts vorgenommen werden, bevor am Ende eine Referenz auf das neu instantiierte Objekt zur weiteren Verwendung zurückgegeben wird.

Ein so entstandenes Objekt kann zwar in den konkreten Werten seiner Properties jederzeit verändert werden, die Struktur des Objekts bleibt aber bis zum Ende seiner Lebensdauer unverändert. Ein Objekt ist strukturell immer eine genaue Kopie seiner Klassendefinition und damit in seiner Struktur statisch.

In JavaScript dagegen sind Objekte dynamische Gebilde, deren Struktur zur Laufzeit verändert werden kann. Sie können jederzeit angelegt und verändert werden, ohne dass es dazu einer vorher festgelegte Strukturdefinition (in Form einer Klassendefinition) bedarf. Zur Objekterzeugung gibt es verschiedene Methoden, die bezüglich der daraus resultierenden Objekte gleichwertig sind.

OBJEKTLITERALE Die einfachste und am häufigsten eingesetzte Methode zur Objekterzeugung setzt darauf das Objekt einfach aufzuschreiben und ein sogenanntes Objektliteral im Quelltext zu platzieren. Dabei werden die Properties des Objekts als *`{key: value}`-Pairs* im Quelltext in geschweiften Klammern angegeben. Daraus wird automatisch ein Objekt erzeugt, welches sich sofort benutzen lässt. Die per Objektliteral erzeugten Objekte werden vom eingebauten Basisobjekt `Object` abgeleitet; es wird also die [[Prototype]]-Property des neuen Objekts auf `Object.prototype` gesetzt.

```
var empty = {}
console.log(empty); //{}
console.log(Object.getPrototypeOf(empty) === Object.prototype); //true

var foo = {
    p1: "Property 1",
```

```
    p2: empty,
    answer: 42,
    f1: function f1(param1, param2) {
        console.log(`${param1}, ${param2}!`);
    },
    f2: function f2() {
        console.log(`The answer is: ${this.answer}; What was the question?`)
    }
}

console.log(foo.p1); //Property 1
console.log(foo.p2); //{}
foo.f1("Hello", "world"); //Hello, world!
foo.f2(); //The answer is: 42; What was the question?
console.log(Object.getPrototypeOf(foo) === Object.prototype); //true
```

Listing 2.5: Erzeugung von Objekten mit Objektliteralen. Zunächst wird ein leeres Objekt `empty` erzeugt. Das Objekt `foo` dagegen hat mehrere Properties, die entweder primitive Werte oder Objekte sein können.

OBJEKTERZEUGUNG ÜBER KONSTRUKTORFUNKTIONEN In JavaScript kann jede Funktion als Konstruktorfunktion verwendet werden, wenn sie mit dem Schlüsselwort `new` aufgerufen wird. Zudem hat jede Funktion neben ihrem eigenen impliziten [[Prototype]]-Link noch eine explizite `.prototype`-Property, die auf ein Objekt verweist, das, bei Verwendung der Funktion als Konstruktorfunktion, als Prototyp des neu erzeugten Objekts dient. Auf diesem Prototyp-Objekt können Properties definiert werden, die neu erzeugte Objekte per Delegation erben sollen. Standardmäßig ist das `.prototype`-Objekt einer neu definierten Funktion ein leeres Objekt `{}`, dessen eigener `.__proto__`-Link auf `Object.prototype` referenziert.

Bei einem Aufruf einer Funktion mit `new` laufen vor der Funktionsausführung einige vorbereitende Schritte ab: Es wird zunächst ein neues, leeres Objekt erzeugt. Dessen impliziter [[Prototype]]-Link wird auf das durch die `.prototype`-Property der Konstruktorfunktion referenzierte Objekt gesetzt. Das Schlüsselwort `this` wird an dieses neu erzeugte Objekt gebunden (*new binding*), bevor die aufgerufene Konstruktorfunktion tatsächlich ausgeführt wird. Am Ende der Funktionsausführung wird `this` implizit zurückgegeben, falls die Funktion nicht per explizitem `return`-Statement einen anderen Rückgabewert definiert. Während der Funktionsausführung kann auf das neue Objekt über die `this`-Referenz zugegriffen werden und das Objekt kann verändert und seine Properties initialisiert werden.

Eine Konstruktorfunktion unterscheidet sich nicht von einer normalen Funktion. Eine Funktion wird erst durch den Aufruf mit dem Schlüsselwort `new` zu einer Konstruktorfunktion. Daher ist bei der Verwendung von Konstruktorfunktionen zur Objekterzeugung besondere Vorsicht geboten, diese auch tatsächlich mit dem Schlüsselwort `new` aufzurufen. Andernfalls wird zwar die Funktion ausgeführt, es wird jedoch vor der Funktionsausführung kein neues Objekt erzeugt und die `this`-Referenz zeigt per *default binding* auf das globale Objekt (im *non-strict mode*) oder auf `undefined` (im *strict*

mode). In beiden Fällen verhält sich das Programm anders als erwartet, und in komplexeren Anwendungen entstehen dadurch häufig sehr subtile, schwer zu entdeckende Fehler. Per Konvention sollen Konstruktorfunktionen immer mit einem Großbuchstaben beginnen. Es kann per Linter-Regel –also mittels einer rein statischen Codeanalyse[6]– geprüft werden, dass jeder Aufruf einer Funktion, die mit Großbuchstaben beginnt, auch mit einem `new` versehen ist (siehe dazu [Simpson, 2014, p. 96]).

```
var Dog = function (dogName) {
    this.name = dogName;
    this.bark = function () {
        console.log(`${this.name} says: Wuff-Wuff`);
    }
}

var bello = new Dog('Bello');
bello.bark(); //Bello says: Wuff-Wuff
var lumpi = Dog('Lumpi'); //called without new and hence pollutes the global namespace
bark(); //Lumpi says: Wuff-Wuff
lumpi.bark(); //Cannot read property 'bark' of undefined
```

Listing 2.6: Erzeugung eines Objekts über eine Konstruktorfunktion. Da Konstruktorfunktionen gewöhnliche Funktionen sind, führt ein vergessenes `new` zu unerwarteten Effekten.

ERZEUGUNG EINES NEUEN OBJEKTS MIT OBJECT.CREATE() Mit ES6 wurde die neue Funktion `Object.create()` eingeführt. Sie erzeugt ein neues, leeres Objekt und setzt dessen [[Prototype]]-Property auf das als erstes Argument übergebene Objekt. Damit hat das neu erzeugte Objekt über Delegation entlang der Prototype-Chain alle Eigenschaften des übergebenen Prototypen geerbt und kann dann mit weiteren Properties angereichert werden, oder es können bestimmte (default-)Properties überschrieben werden.[7]

```
var protoDog = {
    name: 'Bello', //default value
    bark: function () {
        console.log(`${this.name} says: Wuff-Wuff`);
    },
}

var defaultDog = Object.create(protoDog);
defaultDog.bark(); //Bello says: Wuff-Wuff
var lumpi = Object.create(protoDog);
lumpi.name = 'Lumpi'; //override the prototype 'name'  property with an own 'name' property
lumpi.bark(); //Lumpi says: Wuff-Wuff

//altering the prototype in the old style and without performance optimizations
```

6 siehe dazu https://de.wikipedia.org/wiki/Lint_(Programmierwerkzeug)

7 In pre-ES6 Laufzeitumgebungen kann der Prototyp eines Objekts auch mittels `Object.setPrototypeOf()` verändert werden. Dies ist im dritten Beispiel in Listing 2.7 gezeigt. Das ist jedoch vergleichsweise langsam und sollte vermieden werden, da moderne Laufzeitumgebungen viele Optimierungen darauf nicht anwenden können. Siehe dazu [Mozilla Developer Network, b].

```
var fifi = {
    name: 'Fifi',
}
Object.setPrototypeOf(fifi, protoDog);
fifi.bark(); //Fifi says: Wuff-Wuff
```

Listing 2.7: Erzeugung von Objekten per `Object.create()` mit einem spezifischen `Prototypen` .

FACTORIES Da in JavaScript jederzeit ein Objekt erzeugt werden kann, ist dies natürlich auch innerhalb von Funktionen möglich. Es ist nicht notwendig eine spezielle Konstruktorfunktion zu schreiben und diese mit `new` aufzurufen. Es ist völlig ausreichend eine Funktion zu schreiben, die ein neues Objekt erzeugt, mit den passenden Parametern initialisiert und am Ende über ein explizites `return` Statement zurück gibt. Eine solche Funktion nennt man *Factory*.

Das gleiche Objekt, das in Listing 2.6 per `Dog`-Konstruktor erzeugt wurde, kann auch über eine `dogFactory` wie in Listing 2.8 erzeugt werden. Bei der Verwendung der `dogFactory` ist die Programmiererin davor sicher, dass ein vergessenes `new` Schlüsselwort zu unvorhergesehenen Fehlern führt.

```
var dogFactory = function(dogName) {
  return {
    name: dogName,
    bark() {
      console.log(`${this.name} says: Wuff-Wuff`);
    }
  };
};

var bello = dogFactory("Bello");
bello.bark(); //Bello says: Wuff-Wuff

var lumpi = dogFactory("Lumpi");
lumpi.bark(); //Lumpi says: Wuff-Wuff
bark(); //ReferenceError: bark is not defined; Program Abortion;
```

Listing 2.8: Erzeugung eines Objekts über eine Factory. Im Gegensatz zur Erzeugung per Konstruktorfunktionen führt ein vergessenes `new` *nicht* zu unerwarteten Effekten.

Die Verwendung von Factories anstelle von Funktionen, die mittels `new` als Konstruktorfunktionen aufgerufen werden, wird von vielen Autoren massiv präferiert. Begründet wird dies mit einer größeren Sicherheit und Flexibilität gegenüber Konstruktorfunktionen. Die Sicherheit rührt daher, dass `new` nicht vergessen werden kann, da es niemals benötigt wird. Da der Aufruf einer Factory ohne `new` und den damit verbundenen Automatismen zur Objekterzeugung erfolgt, ist mit Factories auch eine größere Flexibilität gegeben. Die Art der Objekterzeugung kann besser kontrolliert und gleichzeitig vor der Anwenderin verborgen werden. Damit ist es beispielsweise

möglich bei großen und aufwändigen Objekten, deren Erzeugung teuer ist, auf einen Objekt-Pool umzusteigen, in dem nicht mehr benutzte Objekte recycelt werden. Eine solche Änderung ist für die Anwenderin von Factories völlig transparent. Mit Konstruktorfunktionen ist solch eine Änderung von Implementierungsdetails nicht möglich, da neue Objekte vor der Änderung immer mit `new` erzeugt wurden und nach der Änderung nur noch ohne `new` abgerufen werden dürfen.

> Using constructor functions is a clear and strong accent, because they are completely unnecessary in JavaScript. They are a waste of time and energy. [Elliott, 2014, p. 51]

Auf alle vier angesprochenen Arten lassen sich in JavaScript neue Objekte erzeugen. Bei der Verwendung von Konstruktorfunktionen ist Vorsicht geboten, da ein vergessenes `new` zu schwer zu lokalisierenden Fehlern führen kann. Objektliterale sind dagegen ein einfaches Mittel, vor allem solche Objekte zu definieren, welche lediglich einmal benötigt werden. Die Erzeugung mittels `Object.create()` eignet sich bestens, um Objekthierarchien aufzubauen. Diese Methode wird in der Praxis meist innerhalb des *Factory*-Musters eingesetzt.

Eine wichtige Eigenschaft und ein starkes Differenzierungsmerkmal von JavaScript ist die Tatsache, dass Objekte dynamische Gebilde sind, die sich jederzeit zur Laufzeit verändern lassen. In klassischen Sprachen sind Objekte in ihrer Struktur starr festgelegt und es lassen sich lediglich die darin gespeicherten Daten verändern. In JavaScript dagegen kann zu jeder Zeit jedem Objekt eine weitere Property hinzugefügt oder aus dem Objekt gelöscht werden. Sogar die Prototype-Chain kann zur Laufzeit verändert werden. Damit lässt sich das Verhalten eines Objekts unter Beibehaltung der eigenen Daten zur Laufzeit vollständig austauschen. Wie wir später noch sehen werden ist diese Dynamik der Objekte in JavaScript ein mächtiges Werkzeug zur Wiederverwendung von Code.

Diese Flexibilität durch Dynamik hat jedoch auch ihren Preis: In JavaScript ist es schwierig bis unmöglich, einem Objekt anzusehen, welche Eigenschaften und welches Verhalten gerade aktuell sind. Während in stark typisierten Sprachen schon zur Compile-Zeit geprüft werden kann, ob auf einem bestimmten Objekt bestimmte Operationen ausgeführt werden können, so kann in JavaScript in der Regel nicht durch eine Typprüfung festgestellt werden, welche Eigenschaften ein Objekt unterstützt. Daher muss in der Praxis auf das sogenannte *Duck-Typing* zurückgegriffen werden: Getreu dem Motto „if it looks like a duck, and it quacks like a duck, it must be a duck"' ([Simpson, 2014, p. 141]) wird dabei nicht der `Typ` eines Objekts überprüft, sondern es wird lediglich das Vorhandensein der gerade benötigten Eigenschaft getestet. Es ist ausreichend, wenn diese Eigenschaft benutzt werden kann. Der formale Typ des Objekts ist für deren Verwendung nicht wichtig.

3 METHOD BORROWING

JavaScript macht es der Programmiererin sehr leicht, eine bestimmte Methode eines Objekts in einem anderen Kontext (das bedeutet mit einem anderen `this`-Binding) auszuführen. Damit kann eine für ein Objekt entwickelte Methode auf ein anderes Objekt angewendet und so wiederverwendet werden.

Nach den im vorigen Abschnitt erläuterten Regeln ist eine Methode normalerweise mit dem Objekt, auf dem sie aufgerufen wird, über das implizite `this` verbunden. JavaScript bietet die eingebauten Funktionen `call()` und `apply()` an, mit denen sich das `this`-Binding bei einem Funktionsaufruf explizit überschreiben lässt. `call()` und `apply()` unterscheiden sich lediglich darin, wie die Parameter des Funktionsaufrufs übergeben werden. Bei `call()` werden sie als Parameterliste angegeben während sie bei `apply()` als Array übergeben werden. Dies wird an einem Beispiel deutlich:

```
// call() example
notmyobj.doStuff.call(myobj, param1, p2, p3);
// apply() example
notmyobj.doStuff.apply(myobj, [param1, p2, p3]);
```

Listing 3.1: Die Methode `doStuff()` des Objekts `notmyobj` wird aufgerufen und explizit per `call(myobj)` bzw. `call(myobj)` an ein anderes Objekt gebunden. (Beispiel aus [Stefanov, 2010])

.

Man spricht bei der gezeigten Technik von *method borrowing*, da sich ein Objekt eine Methode eines anderen Objekts ausleiht.

Diese Technik wird sehr häufig angewendet, um Funktionen eingebauter Objekte, z. B. des `Array`-Objekts zu benutzen, obwohl das Objekt, auf dem operiert wird, selber kein Array, sondern nur *array-like* ist. Die innerhalb von Funktionen belegte Variable `arguments` ist ein solches array-like Objekt (siehe [Mozilla Developer Network, a]). Um es in ein *echtes* Array umzuwandeln, kann die `slice()` Methode aus `Array.prototype` ausgeliehen werden:

```
var toArray = function () {
    return Array.prototype.slice.call(arguments);
}

console.log(toArray(1, 4, 3, 2)); //[ 1, 4, 3, 2 ]
```

Listing 3.2: Die Methode `Array.prototype.slice` wird ausgeliehen und auf das array-like `arguments` Objekt angewendet.

Wenn eine Funktion in einer Variable gespeichert werden soll oder als Callback übergeben und später aus einem anderen Kontext heraus aufgerufen wird, so verliert sie ihr `this`-Binding. `this` fällt beim Aufruf auf das default-Binding zurück, das im non-strict-Mode auf das *global Object* zeigt und im strict-Mode zu einem Laufzeitfehler

führt. Um eine Funktion fest an einen Kontext zu binden, gibt es seit ES5 die Funktion `Function.prototype.bind()`, die es erlaubt ein festes Objekt als `this`-Binding und optional weitere Funktionsparameter zu fixieren.

```
var one = {
    name: "object",
    say: function (greet) {
        console.log(`${greet}, ${this.name}`);
    }
};

var two = {
    name: "another object"
    };

// test
one.say('hi'); // "hi, object"

// assigning to a variable
// `this` will point to the global object
var say = one.say;
say('hoho'); // "hoho, undefined"   //no name-property on global object
say.call(two, 'hoho'); // "hoho, another object"

//fix explicit binding to two-Object
var twosay1 = one.say.bind(two);
twosay1('yo'); // yo, another object

//fix explicit and parameter assignment
var twosay2 = one.say.bind(two, 'Cheers');
twosay2(); // Cheers, another object
```

Listing 3.3: `bind()` zur festen `this`-Bindung (Beispiel frei nach [Stefanov, 2010])

Die aus dem Objekt `one` geborgte Methode wird zunächst ungebunden in eine Variable gespeichert (l. 12), so dass ihr `this` beim Aufruf auf das globale Objekt zeigt, das keine Property `name` besitzt. Per `call()` kann das `this`-binding explizit gesetzt werden (l. 14). Optional kann auch die Funktion vor dem Speichern in einer Variablen fest an ein bestimmtes Objekt gebunden (l. 17) und zusätzlich mit festen Funktionsparametern (l.21) versehen werden. Eine derart fest gebundene Funktion kann auch problemlos als Parameter für Callbacks verwendet werden.

4 DELEGATION UND MIXINS

Ein Mechanismus zum Code-Reuse in klassischen Sprachen ist die Vererbung. Dazu wird eine Hierarchie aufeinander aufbauender Klassendefinitionen definiert. Die tiefer liegende Subklassen *erben* alle Eigenschaften der darüber liegender Superklassen und können sie verwenden. Die tiefer liegenden Subklassen können eigene Eigenschaften ergänzen und Eigenschaften der darüber liegenden Superklassen verändern. Man spricht davon, dass Methoden oder Properties überschrieben werden. Man kann sich

das vorstellen, wie Baupläne auf transparentem Papier, die übereinandergelegt werden.

Wenn eine Instanz einer Subklasse erzeugt wird, so entsteht ein neues Objekt nach der Definition, die sich aus der Summe der Klassendefinitionen der Klassenhierarchie ergibt. Das neue Objekt ist eine materialisierte Kopie dieser Summendefinition.

4.1 Prototypische Vererbung und Delegation

Da JavaScript nicht auf Klassen basiert, sondern auf Objekten, die zueinander in Beziehung stehen, ist dieser Mechanismus in der Form nicht verfügbar. Javascript bietet anstelle dessen den *Delegation*-Mechanismus entlang der Prototype-Chain.

Der [[Prototype]]-Link eines jeden Objekts ist zugreifbar und kann sowohl gelesen als auch gesetzt werden. Der Prototyp ist über die Funktionen `Object.getPrototypeOf(obj)` und `Object.setPrototypeOf(obj, proto)`, und seit ES6 auch über die eigenen Property `.__proto__` zugreifbar. Bei Objekten, die über einen Konstruktoraufruf erzeugt werden, wird der [[Prototype]]-Link bei der Erzeugung auf das über die Property `.prototype` der Konstruktorfunktion referenzierte Objekt gesetzt.

Mit Hilfe der Prototype-Chain und der Möglichkeit, den Prototypen eines Objekts zu manipulieren, lässt sich klassenähnliches Verhalten in JavaScript simulieren: (Wert-) Properties werden in der Konstruktorfunktion auf dem neu erstellten Objekt definiert. Sie sind für jedes später über den Konstruktor erzeugte Objekt genau einmal vorhanden. Methoden (Verhaltens-Properties) werden auf dem Prototypen des Objekts definiert, auf den `.prototype` der Konstruktorfunktion zeigt. Sie sind lediglich einmal auf dem [[Prototype]]-Objekt vorhanden, auf das alle über den Konstruktor erzeugten Objekte verlinkt sind. Auch auf dem Prototypen können Wert-Properties definiert werden, das funktioniert aber häufig nicht so, wie gewünscht.

Ein Beispiel eines Zählerobjekts ist in Listing 4.1 zu sehen:

```
var Counter = function (name) {
    this.name = name;
}

Counter.prototype = {
    counter: 0,
    counterObj: {
        value: 0,
    },
    inc: function () {
        this.counter++;
        this.counterObj.value++;
    },
    getCount: function () {
        console.log(`${this.name}.counter = ${this.counter};\t${this.name}.counterObj.value =
            ${this.counterObj.value};`);
    }
}

var one = new Counter('one');
```

```
20 console.log(`one.counter? ${one.hasOwnProperty('counter')}`); //one.counter? false
   var two = new Counter('two');

   one.inc();
   console.log(`one.counter? ${one.hasOwnProperty('counter')}`); //one.counter? true
25 one.inc();
   one.inc();
   two.getCount(); //two.counter = 0;  two.counterObj.value = 3;
   two.inc();
   one.getCount(); //one.counter = 3;  one.counterObj.value = 4;
30 two.getCount(); //two.counter = 1;  two.counterObj.value = 4;
```

Listing 4.1: Erzeugung eine `Counter`-„Klasse“ und Instantiierung zweier `Counter`-Objekte.

Zunächst wird eine Konstruktorfunktion definiert, auf deren `.prototype`-Objekt die für einen Zähler notwendigen Properties und Funktionen definiert sind. Sodann werden über Konstruktoraufrufe zwei Zählerobjekte erzeugt. Die Zähler werden mehrfach inkrementiert und das Ergebnis abgefragt. Es ergibt sich folgende (überaschende) Ausgabe:

```
one.counter? false
one.counter? true
two.counter = 0; two.counterObj.value = 3;
one.counter = 3; one.counterObj.value = 4;
two.counter = 1; two.counterObj.value = 4;
```

Diese Ausgabe lässt sich wie folgt erklären: Die Wert-Properties `counter` und `counterObj` sind auf dem Prototypen `Counter.prototype` definiert. Das durch den Konstruktoraufruf `new Counter('one');` erzeugte Objekt hat keine eigene Property mit dem Namen `counter`. Sobald jedoch die Funktion `inc()` aufgerufen wird, wird eine solche Property direkt auf dem Objekt `one`, auf das die `this`-Referenz beim Funktionsaufruf zeigt, angelegt und mit einem Wert belegt. Das liegt daran, das bei schreibenden Zugriffen auf eine Property nicht erst ein passendes Objekt entlang der Prototype-Chain gesucht wird, sondern direkt eine eigene Property angelegt wird, wenn der Schreibzugriff auf eine noch nicht existierende Property des Objekts erfolgt.

Die Zeile `this.counter++;` wird ausgeführt als `one.counter = one.counter + 1`. Es wird die Property `counter` auf dem Objekt `one` gesucht und auf `one.__proto__` gefunden. Dazu wird `1` addiert und es wird der Property `one.counter` zugewiesen. Da diese Property auf `one` noch nicht existiert, wird sie angelegt. Ab nun hat `one` eine eigene Property `counter`, welche `one.__proto__.counter` verdeckt.

Die Zeile `this.counterObj.value` dagegen wird als `one.counterObj.value = one.counterObj.value + 1` ausgeführt. Hier wird bei beiden Zugriffen auf `one.counterObj` nur lesend zugegriffen. Der Schreibzugriff erfolgt auf die darin enthaltene `value` Property. In beiden Fällen wird für `one.counterObj` die Objektreferenz auf `one.__proto__` gefunden. Die in diesem referenzierten Objekt enthaltene Property `value` wird ausgelesen, inkrementiert und gespeichert. Die Objektreferenz `one.counterObj` bleibt dabei

unverändert. Sie ist für beide Objekte `one` und `two` lediglich einmal auf dem durch `Counter.prototype` referenzierten Objekt vorhanden. Die Abbildung 4.1 veranschaulicht die Situation.

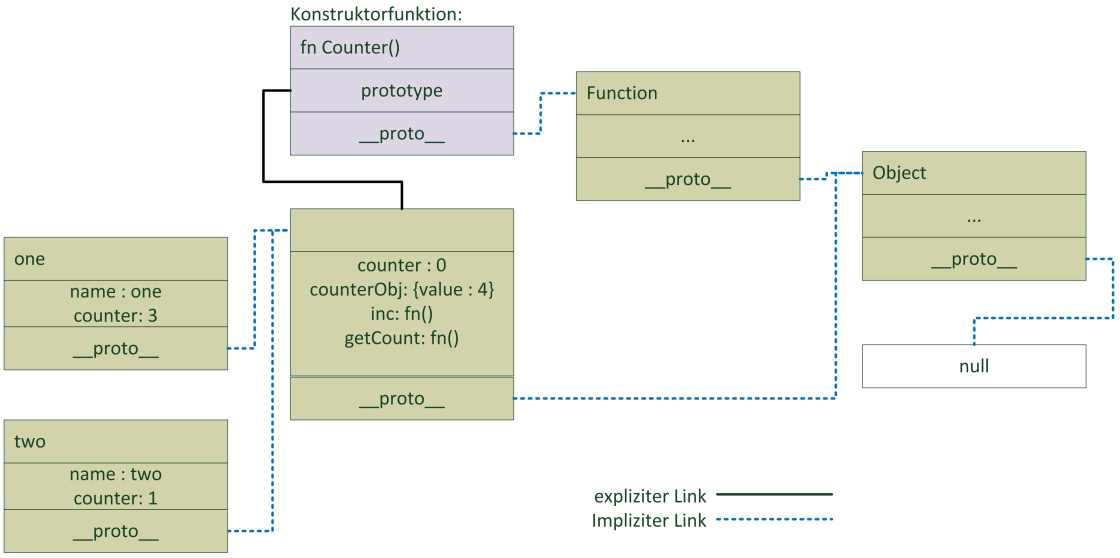

Abbildung 4.1: Objektgeflecht des Zählerbeispiels.

Ausgehend von dem gezeigt Beispiel lassen sich natürlich auch in JavaScript mehrstufige Delegations-Hierarchien erstellen, die einer Vererbungshierarchie in klassischen OO-Sprachen ähnelt. Zur Verdeutlichung des Prinzips sei ein kurzes Beispiel gezeigt. Es sollen, wie in Abbildung 4.2 dargestellt, vier Objekte erstellt werden.

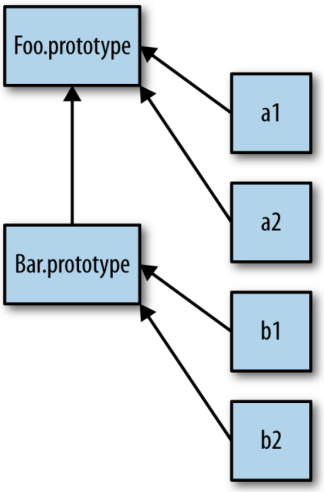

Abbildung 4.2: Prototypische Vererbung / Delegationshierarchie (aus [Simpson, 2014, p. 93])

Dies lässt sich wie im Listing 4.2 gezeigt programmieren. Zunächst wird eine Konstruktorfunktion für `Foo` erstellt, auf deren `.prototpye`-Objekt eine Funktion `identify` definiert wird. Dieser Prototyp wird als zu verlinkendes Delegate-Objekt an die zweite Konstruktorfunktion `Bar` über deren `.prototype`-Link angebunden. Im `Bar`-Konstruktor wird explizit `Foo.call(this)` aufgerufen. Es handelt sich dabei um Method-Borrowing, so dass `Foo` ein explizites `this`-Binding auf das Objekt erhält, das beim Konstruktoraufruf von `Bar` neu erzeugt wurde und auf diesem Objekt Properties setzen kann.

Auch auf dem `Bar.prototype`-Objekt ist eine Funktion `identify` definiert. Diese „überschreibt“ die gleichnamige Funktion auf `Foo.prototype`, da sie in der Prototype-Chain früher gefunden wird. Es ist zu erkennen, dass der in klassichen Sprachen übliche Aufruf der überschriebenen Funktion deutlich umständlicher ist als der klassisch übliche Aufruf `super()`. In JavaScript ist man daher bemüht das Überschreiben zu vermeiden, wenn die überschriebene Property noch benötigt wird.

```
function Foo(name) {
    this.name = name;
}

Foo.prototype.identify = function () {
    return {name: this.name};
};

function Bar(name, label) {
    Foo.call(this, name);   //call the Foo function with explicit this binding
    this.label = label;
}
// here, we make a new 'Bar.prototype' linked to 'Foo.prototype'
Bar.prototype =  Object.create(Foo.prototype);

Bar.prototype.identify = function () {
    return { ...this.__proto__.__proto__.identify.call(this),   //"super call"
            label: this.label
    };
};

var a1 = new Foo("a1");
var a2 = new Foo("a2");
var b1 = new Bar("b1", "obj b1");
var b2 = new Bar("b2", "obj b2");

console.log(a1.identify()); // { name: 'a1' }
console.log(b1.identify()); // { name: 'b1', label: 'obj b1' }
```

Listing 4.2: Erzeugung einer mehrstufigen prototypischen Vererbungs-/Delegations-Hierarchie (frei nach [Simpson, 2014], p. 101)

4.2 Vererbung durch Kopieren

Im vorangehenden Abschnitt wurde gezeigt, wie die in JavaScript enthaltene Delegation zwischen OLOO (Objects Linked to Other Objects) funktioniert, und wie man mit

ihrer Hilfe einen Mechanismus aufsetzen kann, der der klassischen Vererbung sehr nahe kommt. Es wurde dabei im `Counter`-Beispiel auch aufgezeigt, dass bei der Objektinstantiierung keine Kopien von Klassendefinitionen erstellt werden, sondern lediglich Objekte, die über Ihre Prototype-Chain auf andere, bereits bestehende Objekte verlinkt sind. Dies kann zu unerwartetem Verhalten führen, wenn nicht genau darauf geachtet wird, ob eine Property einen primitiven Wert enthält oder auf ein Objekt refrenziert.

In den vorangehenden Beispielen wurde auch deutlich, dass –im Gegensatz zu klassischen Sprachen– Objekte in JavaScript dynamisch sind und jederzeit auch nach Erstellung verändert werden können. So wurden den `.prototype`-Objekten nach ihrer Erstellung Funktions-Properties hinzugefügt.

Die Möglichkeit, Objekte nach ihrer Erstellung zu verändern, lässt sich nutzen, um Code-Wiederverwendung nicht über Delegation zu erreichen, sondern über automatisierte Kopien von Properties. Es lässt sich einfach eine Hilfsfunktion `extend` schreiben, die ein bestehendes Objekt mit den Properties eines anderen Objekts erweitert. Die Vererbung findet hier durch Kopieren statt.

In Listing 4.3 wird ein `kid` Objekt dadurch erzeugt, dass in ein leeres Objekt alle eigenen Properties des `dad`-Objekts einkopiert werden. Wichtig zu sehen ist dabei, dass diese beiden Objekte `nicht` über die Prototype-Chain verbunden sind. Es werden lediglich die im Elternobjekt vorhandenen eigenen Properties kopiert. Es wird eine sogenannte *flache* Kopie (*shallow copy*) des Elternobjekts im Kind-Objekt erzeugt. Die gezeigte Funktion `extend` ist seit ES6 auch im JavaScript Sprachstandard enthalten und es kann einfacher `Object.assign(target, ...sources)` geschrieben werden.

```
function extend(parent, child) {
    var i;
    child = child || {};
    for (i in parent) {
        if (parent.hasOwnProperty(i)) {
            child[i] = parent[i];
        }
    }
    return child;
}

var dad = {
    name: 'Adam',
    counts: [1, 2, 3],
    reads: { paper: true }
};
var kid = extend(dad);
kid.name = 'Phil';
kid.counts.push(4);
console.log(dad.name); //Adam
console.log(kid.name); //Phil
console.log(dad.counts); // [1, 2, 3, 4]
console.log(dad.reads === kid.reads); // true
```

Listing 4.3: „Vererbung“ durch Kopieren (frei nach [Stefanov, 2010], p. 133f.)

Auch hier besteht das Problem wie im `Counter`-Beispiel, dass Properties, die Referenzen auf andere Objekte sind, von beiden Kopien aus auf das gleiche Ursprungsobjekt zeigen und damit keine eigenständigen, gedoppelten Werte enthalten. Das ist in der Regel nicht gewünscht, und es müssen entsprechende Setter-Funktionen programmiert werden, die den Inhalt solcher Referenzen durch eigene Objekte ersetzen, und nicht nur den Wert einzelner Objektproperties verändern. In diesem Beispiel sollte die Funktion `Array.concat()` anstelle von `Array.push()` verwendet werden. `Array.concat()` gibt immer ein neues Array zurück, während `Array.push()` das ursprüngliche Array verändert.

Eine (selten genutzte) Alternative besteht darin, eine sogenannte *tiefe* Kopie (*deep copy*) zu erstellen, die Objekt- und Array-Referenzen rekursiv auflöst, und entsprechend neue Werte im Empfängerobjekt erstellt. Eine solche tiefe Kopierfunktion ist in Listing 4.4 zu sehen.

```
function extendDeep(parent, child) {
    var i,
        toStr = Object.prototype.toString,
        astr = "[object Array]";
    child = child || {};
    for (i in parent) {
        if (parent.hasOwnProperty(i)) {
            if (typeof parent[i] === "object") {
                child[i] = (toStr.call(parent[i]) === astr) ? [] : {};
                extendDeep(parent[i], child[i]);
            } else {
                child[i] = parent[i];
            }
        }
    }
    return child;
}

var dad = {
    counts: [1, 2, 3],
    reads: { paper: true }
};
var kid = extendDeep(dad);
kid.counts.push(4);
console.log(kid.counts); // "1,2,3,4"
console.log(dad.counts); // "1,2,3"
console.log(dad.reads === kid.reads); // false
kid.reads.paper = false;
kid.reads.web = true;
console.log(dad.reads.paper); // true
```

Listing 4.4: „Vererbung“ durch *tiefes* Kopieren (frei nach [Stefanov, 2010], p. 134)

Mit Hilfe der `extendDeep` Funktion wird das `parent`-Objekt im `child`-Objekt vollständig dupliziert. Dadurch können sich die beiden Objekte danach nicht mehr beeinflussen. Eine solche tiefe Kopie ist jedoch eine sehr teure Operation und meist nicht notwendig. Der Einsatz einer tiefen Kopierfunktion für die Wiederverwendung von Code ist nur in Ausnahemefällen sinnvoll, die im Vorfeld genau analysiert werden

müssen. In den meisten Fällen ist die sorgfältige Implementierung geeigneter Setter vorzuziehen, die selektiv Objekte austauschen, wenn darin enthaltene Werte verändert werden sollen.

4.3 Object-Mixins für den orthogonalen Code-Reuse

Durch bisher gezeigten Beispiele der Code-Wiederverwendung per Delegation oder per Kopie wurde eine baumartige Hierarchie aufgebaut, wie sie typischerweise in allen Vererbungshierarchien in OO-Sprachen mit Einfachvererbung entsteht. Eine solch strenge Hierarchie, in der jede Entität genau einen Vorgänger hat, von dem Eigenschaften übernommen werden können, ist in der Praxis häufig nicht ausreichend. Es gibt in der Realität in fast allen Bereichen Anwendungsfälle, die sich mit einer solchen Baumstruktur nicht darstellen lassen.

Als Beispiel sei das Modell einer Firma und ihrer Mitarbeitern genannt: Ein `Entwickler` ist ein `Angestellter`, der eine `Person` ist. Daneben gibt es noch einen `Produktionsmitarbeiter`, der ein `Angestellter` ist, der eine `Person` ist. Dieses auf den ersten Blick einleuchtende Modell bricht, sobald ein externer `Entwickler` in der Firma beschäftigt wird, der selber ein `Freelancer` ist. In diesem Fall kann die Realität nicht über einen einfachen Baum modelliert werden. Es kommen Eigenschaften hinzu, die zu der ursprünglich verwendeten Taxonomie *orthogonal* sind.

Ein Ansatz zur Modellierung solcher orthogonaler Beziehungen, die nicht einer strengen Hierarchie gehorchen, wird als *Mixin* bezeichnet.

Der Begriff des Mix-In wurde laut Wikipedia[8] ursprünglich von einem Eis-Verkäufer geprägt, der aus den immer gleichen Grundsorten (Vanille, Schokolade, ...) und weiteren Zutaten (Smarties, Schoko-Chips, Gummibärchen, Kekse, ...) eine Unmenge an spezialisierten Eiscremesorten anbieten konnte, die individuell nach Kundenwunsch zubereitet wurden. Er benutzte eine Grundsorte als Ausgangsbasis und mixte weitere Zutaten nach Kundenwunsch unter, so dass ein neuer individueller Geschmack entstand.

Übertragen auf klassische OO-Programmierung, stellen die baumartigen Vererbungshierarchien aus Super- und Subklassen die Grundzutaten dar. Die zusätzlichen Bestandteile werden mit dem dazu orthogonalem Verhalten identifiziert, das je nach Bedarf hinzugefügt werden kann. Das zuzufügende Verhalten kann in der Sprache der klassischen Objektorientierung als Definition einer *abstrakten Subklasse* bezeichnet werden.

> A mixin is an abstract subclass; i.e. a subclass definition that may be applied to different superclasses to create a related family of modified classes. [Bracha und Cook, 1990]

Diese abstrakte Subklasse kann für sich gesehen nicht instantiert werden. Sie kann nur auf eine konkrete Basisklasse angewendet werden, und damit ihr Verhalten dem

8 https://en.wikipedia.org/wiki/Mix-in

Verhalten der Basisklasse hinzufügen. Die Basisklasse muss dazu bestimmte Garantien bezüglich ihrer Implementierung geben, damit die erweiterte Funktionalität zur Verfügung gestellt werden kann. Die Verwendung von Mixins in klassischen OO-Sprachen bedarf der Unterstützung der Sprache selber, deren Klassendefinitionen und Objektinstantiierung die Idee einer abstrakten Subklasse, die auf eine konkrete Superklasse angewendet wird, implementieren müssen. Die meisten klassischen OO-Sprachen haben ein Modell implementiert, bei der die Konkretisierung von abstrakten Superklassen zu konkreten Subklassen, also von oben nach unten, geht. Dieses Modell lässt sich nicht ohne weiteres umkehren, weswegen Mixins nicht in vielen Sprachen unterstützt werden.[9]

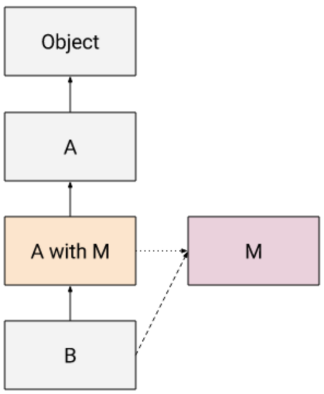

Abbildung 4.3: Die abstrakte Mixinklasse `M` angewandt auf die Basisklasse `A` in der Vererbungshierarchie `class B extends A with M {}`([Fagnani, 2015])

JavaScript hat keine Klassen. Es gibt lediglich Objekte und diese Objekte sind zur Laufzeit dynamisch veränderbar. Damit ergibt sich als direkte Konsequenz aus der im vorigen Abschnitt aufgezeigten einfachen „Wiederverwendung durch Kopieren“ die Möglichkeit, das Verhalten eines Objekts in ein anderes bestehendes Objekt zu mixen. Der in klassischen Sprachen recht aufwändige Mixin-Mechanismus lässt sich in JavaScript auf eine einfache Kopie zurückführen.

In einer Grundform ist eine flache Kopie der Properties des Mixin-Objekts auf das Empfängerobjekt ausreichend, um eine Mixin-Funktionalität in Javascript zur Verfügung zu stellen. Dieser Vorgang wird auch als „concatenative sharing“ ([Braithwaite, 2016]) bezeichnet, da schlicht die Properties des Mixin-Objekts an das zu erweiternde Objekt angehängt werden. In der Vergangenheit wurden dazu meist mehr oder weniger aufwändige `mixin()`-Funktionen aus Bibliotheken verwendet. Seit ES6 ist jedoch die Funktion `Object.assign(target, ...sources)` standardisiert und kann für diesen Zweck genutzt werden.

9 Eine Liste von Sprachen, die Mixins direkt unterstützen, findet sich z. B. in https://en.wikipedia.org/wiki/Mixin%23Programming_languages_that_use_mixins

Im Listing 4.5 wird gezeigt, wie leicht sich bestehende Objekte durch einfache Erweiterung mit Mixin-Verhalten ergänzen lassen. Es wird auch deutlich, dass die Mixin-Objekte nicht für sich allein stehen können, da sie im Beispiel auf die `.name`-Property des Empfängerobjekts zugreifen.

```
var mixDeveloper = {
    languages: [],
    patterns: [],
    addLanguage(lang) {
        // use concat instead of push, to not modify the original array but to replace it!
        this.languages = this.languages.concat(lang);
        return this;
    },
    addPattern(pat) {
        // use concat instead of push, to not modify the original array but to replace it!
        this.patterns = this.patterns.concat(pat);
        return this;
    },
    developerSay() {
        console.log(`My name is ${this.name}, my Languages are ${this.languages} and I master
             ${this.patterns}.`);
        return this;
    },
    initDeveloper(languages = [], patterns = []) {
        this.languages = languages;
        this.patterns = patterns;
        return this;
    }
}

var mixEmployee = {
    personnelNumber: 0,
    annualSalary: 50000,
    setPersonalNumber(num) { this.personnelNumber = num },
    setAnnualSalary(num) { this.annualSalary = num },
    employeeSay() {
        console.log(`My name is ${this.name}, my personnel number is ${this.personnelNumber}.`)
        return this;
    },
    initEmployee(personalNumber = 0, annualSalary = 50000) {
        this.personnelNumber = personalNumber;
        this.annualSalary = annualSalary;
        return this;
    }
}

var mixFreelancer = {
    hourlyRate: 100,
    setHourlyRate(num) { this.hourlyRate = num },
    freelanceSay() {
        console.log(`My name is ${this.name}, my hourly rate is ${this.hourlyRate}.`)
        return this;
    },
    initFreelance(hourlyRate = 100) {
        this.hourlyRate = hourlyRate;
        return this;
    }
}

var protoPerson = {
```

```
55      initPerson(name) {
            this.name = name;
            return this;
        },
        personSay() {
60          console.log(`Hello, my name is ${this.name}`)
        },

    }

65  let felix = Object.create(protoPerson).initPerson('Felix');
    Object.assign(felix, mixDeveloper, mixFreelancer);

    let john = Object.create(protoPerson).initPerson('John');
    Object.assign(john, mixDeveloper, mixEmployee);
70
    console.log(`john.languages === felix.languages? ${john.languages === felix.languages}`);
    //john.languages === felix.languages? true


75  felix.initDeveloper(['JS', 'C'], ['Mixins', 'Decorators']).initFreelance(120);
    felix.developerSay();   //My name is Felix, my Languages are JS,C and I master Mixins,Decorators.
    felix.freelanceSay();   //My name is Felix, my hourly rate is 120.


80  john.initDeveloper(['Java'], ['Singletons', 'Facade']).initEmployee(666, 30000);
    john.addLanguage('C#');
    john.developerSay();    //My name is John, my Languages are Java,C# and I master Singletons,Facade.
    john.employeeSay(); //My name is John, my personnel number is 666.
```

Listing 4.5: Einfache Object-Mixins durch Definition von Mixin-Objekten und Anwenden von `Object.assign(target, ...sources)`

Bei der vorgestellten Mixin-Technik werden die Empfängerobjekte lediglich durch eine flache Kopie der Mixin-Objekte erweitert. Es ist daher genau wie im vorangehenden Beispiel darauf zu achten, dass Referenzen auf Objekte innerhalb verschiedener Empfängerobjekte zunächst auf das gleiche Objekt zeigen. Während dieses Verhalten bei den Referenzen auf Funktionen (und damit auf Verhalten des Mixins) wünschenswert ist, um Speicher zu sparen, ist bei objektwertigen Properties Vorsicht geboten. Nach der Erweiterung der beiden Objekte `felix` und `john` mit dem `mixDeveloper` gilt `john.languages === felix.languages`. Das bedeutet, dass die `.languages`-Property beider Objekte auf dasselbe Array-Objekt referenziert (l. 72).

Im gezeigten Code-Beispiel wurde dafür Sorge getragen, dass dieses Verhalten keine unerwünschten Nebeneffekte hat. Die `languages` und `patterns` Arrays werden bei Veränderung ersetzt und nicht modifiziert. Sowohl die Funktion `initDeveloper` als auch die Funktionen `addLanguage` bzw. `addPattern` ersetzen die Arrays und modifizieren damit die Referenzen der Objekte und nicht nur die Werte innerhalb referenzierter Objekte.

Aufgrund der Verwendung dynamischer Objekte, deren Verhalten und Struktur zur Laufzeit modifiziert werden kann, ist das Mixin-Pattern in JavaScript sehr einfach anzuwenden. Es bedarf keiner zusätzlichen Sprachunterstützung, sondern lässt sich mit

den vorhandenen Mitteln einfach erledigen. Das Muster lässt sich auch in Kombination mit der [[Prototype]]-Delegation einsetzen, so dass nicht nur einzelne Objekte wie im Beispiel mit erweiterter Funktionalität ausgestattet werden können, sondern auch deren [[Prototype]]-Delegates. Dies lässt sich in Factory-Funktionen gewinnbringend einsetzen, wenn eine große Menge erweiterter Objekte erzeugt werden soll.

Trotz des einfachen Einsatzes von Mixins soll nicht verschwiegen werden, dass in der Praxis in größeren Projekten noch einige Details behandelt werden müssen, die den Einsatz etwas komplizierter als im Beispiel gezeigt machen. Dies betrifft insbesondere den Umgang mit namensgleichen Properties. In der gezeigten Methodik „gewinnt" das zuletzt eingemixte Objekt und dessen Implementierung einer Property oder eines Verhaltens bestimmt das Verhalten des erweiterten Objekts. Ohne spezielle Vorkehrungen gibt es keine Möglichkeit, die Methoden oder Properties der Vorgängerobjekte zu erreichen. Es kann jedoch per Method-Borrowing explizit aus der überschreibenden Mixin-Methode die gleichnamige Methode des Empfängerobjekts aufgerufen werden (siehe dazu „explicit pseudopolymorphism" [Simpson, 2014, p.78]). Eine andere Möglichkeit besteht darin eine aufwändigere `mixin()`-Funktion zu benutzen, die bei Namensgleichheit eine automatische Konfliktlösung bietet. Dies wird z. B. in `react-server.js` realisiert. Dort können für bestimmte Properties sogenannte `specPolicies` wie `OVERRIDE_BASE`, `DEFINE_ONCE`, `DEFINE_MANY` oder `DEFINE_MANY_MERGED` angegeben werden. Diese zusätzlichen Angaben werden beim Kopieren der Properties entsprechend berücksichtigt. Die Details dieser Implementierung[10] führen an dieser Stelle jedoch viel zu weit.

5 FUNCTIONAL MIXINS

Im vorherigen Kapitel wurde die Code-Wiederverwendung in JavaScript hauptsächlich durch die klassische Brille in Anlehnung an Vererbungstechniken betrachtet. Über die Techniken, Vererbung per Delegation und per automatischer Kopie zu realisieren, wurde die Idee von Mixins entwickelt, die es ermöglichen aus dem streng hierarchischen Korsett der Vererbung auszubrechen und damit eine flexiblere Möglichkeit der Objektanreicherung bieten. Es wurde gezeigt, dass durch die dynamischen Objekte in JavaScript eine Verwendung von Mixins deutlich einfacher ist als in klassischen Sprachen, die dafür spezielle Mechanismen in der Klassendefinition unterstützen müssen.

Von Detailfragen, wie der Konfliktauflösung bei namensgleichen Properties, abgesehen ist in JavaScript ein Mixin eine simple Erweiterung des Empfängerobjekts zur Laufzeit. Um ein Objekt-Mixin, wie im vorigen Kapitel beschrieben anzuwenden, wird

10 Die Funktion `mixSpecIntoComponent` lohnt eine weitere Betrachtung der Mechanismen: https://github.com/reactjs/react-rails/blob/ec9e1736d4aac87afe4c4e8ba024b49deadc1d3a/lib/assets/react-source/development/react-server.js ab Zeile 2828

zunächst ein Mixin-Objekt erzeugt, dessen Eigenschaften mit einer mehr oder weniger ausgefuchsten Kopierfunktion auf das Empfängerobjekt übertragen werden. Das Mixin-Objekt ist für sich selbst genommen jedoch meist nutzlos, da es lediglich abstrakte Methoden zur Verfügung stellt, die auf bestimmte Properties des Empfängerobjekts angewiesen sind. Es liegt daher die Frage nahe, ob es notwendig ist erst zwei Objekte zu erstellen, die mittels einer speziellen Funktion zusammengeführt werden müssen.

In seinem viel beachteten Blogpost „A fresh look at Javascript Mixins“ ([Croll, 2011]) stellt Angus Croll genau diese Frage und schlägt vor, Mixins eher als Funktion denn als abstraktes Objekt aufzufassen. Ein solches *funktionales Mixin* ist eine Funktion, die auf das zu erweiternden Objekt angewendet wird, um es zielgerichtet zu erweitern. Eric Elliot greift dieses Konzept viele Jahre später wieder auf und erweitert es ([Elliott, 2017a]). Er entwickelt Mixin-Funktionen, die im Sinne der funktionalen Programmierung *composable*, also zusammensetzbar hinereinander ausführbar sind und vergleicht diese funktionalen Mixins mit Arbeitern an einem Fließband, die nacheinander Properties und Verhalten zu einem Objekt hinzufügen. Diese Art der Betrachtung entspricht in wesentlichen Belangen dem sogenannten *Decorator*-Pattern aus [Gamma, 1995, p. 169], das als flexible Alternative zu Subklassenbeziehungen empfohlen wird. Zusammen mit einigen, seit ES6 möglichen syntaktischen Verfeinerungen wird sich zeigen, dass funktionale Mixins als Decorators eine sehr übersichtliche Syntax ermöglichen, die sehr ausdrucksstark und leicht lesbar ist.

Bevor Elliot's snytaktischen ES6-Erweiterungen vorgestellt werden, soll hier das ursprüngliche Beispiel von Croll gezeigt werden, da einige Besonderheiten hier im direkten Vergleich zu den bisherigen Objekt-Mixins sehr gut zutage treten.

Für eine Benutzeroberfläche sollen runde Knöpfe erstellt werden. Dazu werden funktionale Mixins definiert, die ein Objekt mit den „Rundheits“- bzw. „Knopfheits“-Eigenschaften versehen.

```
const asCircle = function (radius) {
    let _radius = radius;
    this.area = function () {
        return Math.PI * _radius * _radius;
    };
    this.grow = function () {
        _radius++;
    };
    this.shrink = function () {
        _radius--;
    };
    return this;
};

const asButton = function (action) {
    let _action = action;
    this.hover = function (bool) {
        bool ? mylib.appendClass('hover') : mylib.removeClass('hover');
    };
```

```
    this.press = function (bool) {
        bool ? mylib.appendClass('pressed') : mylib.removeClass('pressed');
    };
    this.fire = function () {
        return _action();
    };
    this.setAction = function (action) {
        _action = action;
        return this;
    }
    return this;
};

const roundButton = {
    label: "Button1",
}

asCircle.call(roundButton, 5)
asButton.call(roundButton, (function () { console.log(`${this.label} pressed`) }).bind(roundButton))

console.log(roundButton.area());    //78.53981633974483
roundButton.fire(); //Button1 pressed
roundButton.grow();
console.log(roundButton.area()); //113.09733552923255
console.log(roundButton._radius); //undefined
```

Listing 5.1: Erstellung eines runden Buttons durch Anwendung zweier funktionaler Mixins. (Beispiel frei nach [Croll, 2011])

An diesem einfachen Beispiel in Listing 5.1 kann man sehen, wie leicht sich die funktionalen Mixins definieren und anwenden lassen. Als Ausgangspunkt wird ein sehr einfaches Objekt erstellt, das sogar leer sein kann. Auf dieses Objekt werden nacheinander die notwendigen funktionalen Mixins angewendet, um es mit der gewünschten Funktionalität zu versehen. Es wird mit dieser Funktionalität *dekoriert*. Dazu wird die Mixin-Funktion mit einem expliziten `this`-Binding über `.call()` aufgerufen, so dass sie ihre eigenen öffentlichen Funktionen an das übergebene Objekt anhängen kann.

Für künftige Versionen von ECMAScript sind Bestrebungen im Gange solche Decorators mit einer speziellen @-Syntax in die Sprache aufzunehmen. Die Anwendung würde damit vereinfacht werden zu:

```
@asCircle(5)
@asButton((function () { console.log(`${this.label} pressed`) }).bind(roundButton))
const roundButton = {
    label: "Button1",
};
```

Listing 5.2: Anwendung von functional Mixins als Decorators in einer möglichen künftigen ES.next-Syntax

Neben der gefälligen und gut lesbaren Schreibweise haben functional Mixins noch den weiteren großen Vorteil der einfachen Kapselung interner Daten. Wie in der letzten Zeile von Listing 5.1 zu sehen ist, wird `_radius` nicht im Empfängerobjekt des Mixins sichtbar. Dies ist möglich, da eine Funktion immer eine Closure bildet, in der die

innerhalb definierten Daten privat bleiben. Diese Closure bleibt auch erhalten, wenn die Funktion nicht mehr ausgeführt wird, aber noch andere Properties (in diesem Fall die Mixin-Verhaltensfunktionen) darauf zugreifen. Private Daten eines Mixins können so einfach im Scope der Mixin-Funktion definiert werden und sind für alle Verhaltensfunktionen dieses Mixins zugreifbar, während sie nach außen nicht sichtbar sind. Damit wird die Oberfläche des Mixins im Gegensatz zu Objekt-Mixins, bei denen auch alle internen Properties des Mixins auf dem Empfängerobjekt sichtbar wurden deutlich verkleinert.

Weiterhin lassen sich beim Mixin-Funktionsaufruf sehr einfach weitere Parameter an die funktionalen Mixins übergeben, die als Optionen für die Funktionalität dienen. Im gezeigten Fall wird so der initiale Radius und die an den Button gebundene Funktion übergeben.

Da die funktionalen Mixins selber für die Anreicherung der Empfängerobjekte zuständig sind und sich, im Gegensatz zu den Objekt-Mixins, nicht auf eine einheitliche Kopierfunktion stützen, kann für jedes Mixin individuell definiert werden, wie mit Namenskonflikten umgegangen wird. Es ist zum Beispiel problemlos möglich vor dem Einfügen des eigenen Verhaltens zu prüfen, ob das Empfängerobjekt schon eine Property gleichen Namens hat. In einem solchen Fall kann entschieden werden, ob diese Property überschrieben werden soll, oder ob eine Kombination der alten mit der neuen Funktionalität stattfinden soll. Ein Beispiel für solch eine Konfliktauflösung ist in Listing 5.3 skizziert.

```
const withID = function (idNumber) {
    let _idNumber = idNumber;
    let superIdentify = this.identify || (()=>{});
    this.identify = function(){
        return {idNumber: _idNumber, ...superIdentify()}
    }
    return this;
};
const withName = function(name) {
    let _name = name;
    let superIdentify = this.identify || (()=>{});
    this.identify = function(){
        return {name: _name, ...superIdentify()}
    }
    return this;
}
const onlyName = withName.call({}, 'Peter');
const onlyID = withID.call({}, '666');
const iDAndName = withName.call(withID.call({}, '999'), 'Hans');
const nameAndId = withID.call(withName.call({}, 'Tim'), '333');

console.log(onlyName.identify());   //{ name: 'Peter' }
console.log(onlyID.identify());     //{ idNumber: '666' }
console.log(iDAndName.identify());  //{ name: 'Hans', idNumber: '999' }
console.log(nameAndId.identify());  //{ idNumber: '333', name: 'Tim' }
```

Listing 5.3: Die funktionalen Mixins benutzten den Rückgabewert der schon vorhandenen `identify`-Methode als Eingabe der eigenen, überschreibenden `identify`-Methode.

Ein Nachteil der funktionalen Mixins gegenüber den Objekt-Mixins soll hier nicht unerwähnt bleiben: In der vorgestellten Variante bekommt jedes dekorierte Objekt eine eigene Kopie der Mixin-Verhaltensfunktionen und benötigt damit mehr Speicher als die Objekt-Mixins, bei denen lediglich eine Referenz auf die Funktionen des Mixin-Objekts in den Empfängerobjekten gespeichert werden.

Crol zeigt in seinem Blogpost eine Möglichkeit auf, wie die Verhaltensfunktionen ähnlich der privaten Daten in eine Closure gekapselt und damit gecached werden können. Sie werden dann nicht mehr für jedes Empfängerobjekt geklont, haben im Gegenzug aber keinen Zugriff mehr auf eine Mixin-Closure, zur Kapselung privater Daten. Es ist daher im Einzelfall abzuwägen, ob die Verhaltensfunktionen in einer Closure gecached werden sollen, um Seicherplatz zu sparen, oder ob die Daten der Mixin-Logik in einer Closure gekapselt werden sollen, um sie privat zu halten.

Nach dieser ersten Einführung in die Idee der funktionalen Mixins soll die Idee einer Fließband-Fertigung von Objekten mit bestimmten Eigenschaften aufgegriffen werden. In [Elliott, 2017b] beschreibt Eric eine sehr elegante Methode, wie sich Factory-Functions zusammen mit funktionalen Mixins kombinieren lassen. Damit wird auch ohne ES.next-Decorators eine extrem ausdrucksstarke und lesbare Syntax geschaffen, mit der sich ohne großen Aufwand Objekte mit den passenden Mixin-Eigenschaften erstellen lassen.

Eine dazu notwendige Hilfsfunktion höherer Ordnung ist `pipe`. Sie ist in Listing 5.4 angegeben:

```
const pipe = (...fns) => (x => fns.reduce((y, f) => f(y), x));

const add1 = n => n+1;
const double = n => n*2;
const add1AndDouble = pipe(add1, double)

console.log(add1AndDouble(20)); //42
```

Listing 5.4: Die Hilfsfunktion Pipe zur Hintereinanderschaltung mehrerer Funktionen

Die Funktion `pipe` nimmt eine Liste von Funktionen `(...fns)` und gibt selber eine Funktion zurück, die einen Parameter `x` nimmt. Der Aufruf dieser zurückgegebenen Funktion mit einem Parameter liefert als Ergebnis die verkettete Anwendung der übergebenen Funktionen auf diesen Parameter. Im angegebenen Beispiel liefert `add1AndDouble(20)` das Ergebnis `double(add1(20)) = (((20)+1)*2) = 42`.

Mit der Hilfsfunktion `pipe` lassen sich funktionale Mixins zu einer „Gesamtfunktion“ zusammenfassen, die sehr gut in einer Factory verwendet werden kann. Im Beispiel in Listing 5.5 wird deutlich, wie ausdrucksstark und gut lesbar sich auf diese Art neue Objekte erzeugen lassen:

```
const pipe = (...fns) => x => fns.reduce((y, f) => f(y), x);

// Set up some functional mixins
const withFlying = o => {
  let isFlying = false;
  return {
    ...o,
    fly() {
      isFlying = true;
      return this;
    },
    land() {
      isFlying = false;
      return this;
    },
    isFlying: () => isFlying
  };
};

const withBattery = ({ capacity }) => o => {
  let percentCharged = 100;
  return {
    ...o,
    draw(percent) {
      const remaining = percentCharged - percent;
      percentCharged = remaining > 0 ? remaining : 0;
      return this;
    },
    getCharge: () => percentCharged,
    getCapacity() {
      return capacity;
    }
  };
};

const withID = ({ idNumber }) => o => {
  let _idNumber = idNumber;
  let superIdentify = o.identify || (() => {});
  return {
    ...o,
    identify: () => ({ idNumber: _idNumber, ...superIdentify() }),
    setLabel(label) {
      _label = label;
    }
  };
};

const withLabel = ({ label }) => o => {
  let _label = label;
  let superIdentify = o.identify || (() => {});
  return {
    ...o,
    identify: () => ({ label: _label, ...superIdentify() }),
    setLabel(label) {
      _label = label;
    }
  };
};

// define the drone factory by composing the functional mixins
const createDrone = ({ capacity = "3000mAh", label = "genericDrone", idNumber = "00-00" }) =>
```

```
  pipe(
    withFlying,
    withBattery({ capacity }),
    withLabel({ label }),
    withID({ idNumber }),
  )({});

// use factory to create a new drone object
const myDrone = createDrone({
  capacity: "5500mAh",
  label: "My 1st Drone",
  idNumber: "18-19"
});

// check the drones functionality
console.log(`identity: ${JSON.stringify(myDrone.identify())}
can fly:  ${myDrone.fly().isFlying() === true}
can land: ${myDrone.land().isFlying() === false}
battery capacity: ${myDrone.getCapacity()}
battery status: ${myDrone.draw(50).getCharge()}%
battery drained: ${myDrone.draw(75).getCharge()}%
`);

// identity: {"idNumber":"18-19","label":"My 1st Drone"}
// can fly:  true
// can land: true
// battery capacity: 5500mAh
// battery status: 50%
// battery drained: 0%
```

Listing 5.5: Ausdrucksstarke Syntax zur Objekterzeugung mit verketteten funktionalen Mixins in einer Faxctory. (Beispiel aus [Elliott, 2017b])

Im Listing 5.5 wird der Vorteil von funktionalen Mixins gegenüber Objekt-Mixins deutlich: Da es sich um normale Funktionen handelt, die auf ein übergebenes Objekt wirken, lassen sie sich zu einer Gesamtfunktion kombinieren, der sich immer noch leicht Parameter als Optionen übergeben lassen. Die Behandlung von überschriebenen Properties kann in jedem Mixin individuell geregelt werden. Die Anwendung ist dadurch extrem einfach, und Implementierungsdetails der Mixins bleiben in den durch die Mixin-Funktionen gebildeten Closures gekapselt.

6 KRITIK UND AUSBLICK

In den vorangegangenen Kapiteln wurden einige Möglichkeiten vorgestellt, wie sich in JavaScript effektiver Code-Reuse auf Objektebene betreiben lässt. Dazu wurden die Eigenheiten herausgestellt, die Javascript als prototypenbasierte Sprache im Gegensatz zu klassischen OO-Sprachen auszeichnet.

Die hier vorgestellten Mittel sind alle in der Praxis bewährt und werden eingesetzt. Trotzdem ist auch in JavaScript kein *heiliger Gral* unter den Programmiermustern in Sicht. Die Programmiererin muss vielmehr –wie in jeder anderen Programmierspra-

che auch– darauf achten, möglichst viele Methoden zu kennen und zu verstehen. Das schließt auch die Schwächen der Methoden mit ein. Nur so kann in der Programmierpraxis im konkreten Projekt sorgfältig ausgewählt werden, welches Werkzeug für welche Aufgabe geeignet ist. JavaScript bietet besonders viele und flexible Möglichkeiten an.

Jede der vorgestellten Methoden hat ihre Stärken und Schwächen. Während die Stärken in der Vorstellung der Muster schon ausreichend herausgestellt wurden, soll nun kritisch hinterfragt werden, welche Schwächen und Defizite man sich mit der Verwendung der einzelnen Muster einkauft.

Die Technik des Method Borrowing ist in vielen Fällen verlockend, um auf die Schnelle Code wieder zu verwenden, der für ein anderes Objekt schon entwickelt wurde. Wie gezeigt wurde, bietet JavaScript einfach die Möglichkeit eine bestehende Methode in einem anderen Kontext auszuführen, ohne dabei auf „schwere“ Techniken wie Vererbung zurückgreifen zu müssen.

Dabei ist extreme Vorsicht geboten, da sich die Programmiererin, die sich eine Methode ausleiht, auf eine bestimmte Implementierung eines anderen –eigentlich unbeteiligten– Objekts verlässt. Dadurch entsteht eine sehr enge Kopplung der beiden Objekte ohne dass die Autorin des Spenderobjekts davon etwas weiß. Änderungen in der ursprünglichen Implementierung können zu Verhaltensänderungen führen, die in der Praxis auch bei kleinen Projekten schon nicht mehr abzusehen sind. Diese Art der Code-Wiederverwendung steht dem allgemein anerkannten Ratschlag „Program to an interface, not an implementation“ ([Gamma, 1995, p. 18]) genau entgegen und sollte daher nach Möglichkeit vermieden werden.

Eine Ausnahme bilden die eingebauten Objekte mit ihren Methoden, die sich aller Voraussicht nach nicht ändern und damit nicht nur ein stabiles Interface bieten, sondern auch eine stabile Implementierung.

Funktionen die lediglich als wiederverwendbare Utilities dienen, sollten nach Möglichkeit in ES6-Modulen implementiert werden. Diese bieten ein wohldefiniertes Interface und können auch per Namespacing sauber vom eigenen Quelltext getrennt werden. Wenn sie den Objektstatus verändern oder lesen sollen, so ist es zu empfehlen, darauf nicht über das implizite `this`-Binding zuzugreifen, sondern das Objekt explizit als Parameter zu übergeben. Details zu ES6-Modulen finden sich z. B. in [Elliott, 2014, §4]. Darauf soll in dieser Arbeit nicht weiter eingegangen werden.

Bei der prototypischen Vererbung gilt das Gleiche wie bei der klassischen Vererbung. Es wird eine sehr enge Kopplung zwischen erbendem und vererbendem Objekt hergestellt. Das erbende Objekt benötigt intime Kenntnisse über die Implementierung des vererbenden Objekts. Damit hat die prototypische Vererbung dasselbe Fragile-Base-Class Problem wie die klassische Vererbung. Auch in Bezug auf den *richtigen* Aufbau der Vererbungshierarchien unterscheiden sich klassische und prototypische Vererbung nicht. Nach einer Weile der Benutzung erweist sich schlussendlich jede Hierarchie als

falsch. Bei der prototypischen Vererbung per Delegation entlang der Prototype-Chain kommen noch die, schon angesprochenen, Eigenheiten hinzu, wenn eine objektwertige Property benutzt wird, die aufgrund ihrer Lage in der Kette von mehreren Objekten erreicht wird.

Das Object-Mixin Pattern addressiert die Probleme, die durch fehlende Mehrfachvererbung und die dadurch geforderte streng baumartige Objekthierarchie entsteht. Das Problem, dass alle beteiligten Entwickler die Implementierungsdetails aller verwendeten Mixins kennen müssen, wird jedoch eher verstärkt. Es kommen mit der Anzahl der verwendeten Mixins immer mehr (teils implizite) Abhängigkeiten in den Code, die kaum mehr zu überblicken sind. Dabei kann es zu Abhängigkeiten in alle Richtungen kommen: Das Empfängerobjekt ist anhängig von der Implementierung durch ein Mixin. Gleichzeitig muss das Mixin sich auf bestimmte Implementierungsdetails des Empfängerobjekts verlassen. Die Abhängigkeiten können sogar zwischen verschiedenen Mixins entstehen, die gemeinsam auf einem Empfängerobjekt benutzt werden. Durch die von Mixins realisierten *many-to-many*-Beziehungen wird das Fragile-Base-Class-Problem potenziert. Eindrücklich nachlesen lässt sich das in diversen Blogposts, die, mit Praxisbeispielen untermauert, davor warnen, Mixins im Übermaß einzusetzen. (Siehe dazu z. B. [Abramov, 2015] und [Braithwaite, 2016].) Aus diesen Gründen hat Facebook 2016 bekannt gegeben, dass die in früheren Versionen von React üblichen Mixins in Zukunft nicht mehr eingesetzt werden sollen und aus dem React-Interface verschwinden werden ([Abramov, 2016]).

Functional Mixins sind zwar eleganter in der Anwendung, haben aber sehr ähnliche Probleme wie Object-Mixins. Ihr großer Vorteil liegt darin, dass die Kapselung privater Daten sehr viel einfacher realisiert werden kann. Dadurch wird die Oberfläche der Empfängerobjekte nicht so stark vergrößert und es treten weniger Probleme mit impliziten Abhängigkeiten auf. Das Interface lässt sich klarer definieren, da lediglich die eigentliche Mixin-Funktionalität über zusätzliche Properties dem Empfängerobjekt zugefügt wird, während interne Properties und Hilfsfunktionen in der Closure der Mixin-Funktion gekapselt bleiben. Die Problematik der many-to-many-Beziehungen bei Anwendung mehrerer Mixins auf ein Objekt wird dadurch jedoch nicht aufgelöst.

Da es keinen *heiligen Gral* der Code-Reuse Muster gibt, liegt die Verantwortung zur Auswahl des jeweils richtigen Musters bei der Programmiererin. Sie muss ihren Werkzeugkasten gut bestücken und von Fall zu Fall auswählen, welche der bereitstehenden Methoden sinnvoll angewandt werden können. Neben dem allgemeinen „Favour object composition over class inheritance.“ [Gamma, 1995, p. 20] ist der Ratschlag von Eric Elliott ein wenig konkreter:

> Start with the simplest implementation and move to more complex implementations only as required: Functions > objects > factory functions > functional mixins > classes [Elliott, 2017a]

Insbesondere der Blick auf Funktionen lohnt sich in JavaScript noch mehr, als in vielen anderen Sprachen. Durch die Möglichkeiten der funktionalen Programmierung und den Einsatz von *Higher Order Functions* lassen sich viele Probleme sehr einfach und präzise lösen, die in nicht-funktionalen Sprachen einigen Aufwand erfordern. Doch das ist eine weitere Seminararbeit.

LITERATUR

[Abramov 2015] Abramov, Dan: *Mixins Are Dead. Long Live Composition*. März 2015. – URL https://medium.com/@dan_abramov/mixins-are-dead-long-live-higher-order-components-94a0d2f9e750. – Zugriffsdatum: 2018-11-07

[Abramov 2016] Abramov, Dan: *Mixins Considered Harmful – React Blog*. Juli 2016. – URL https://reactjs.org/blog/2016/07/13/mixins-considered-harmful.html. – Zugriffsdatum: 2018-10-15

[Bracha und Cook 1990] Bracha, Gilad ; Cook, William: Mixin-Based Inheritance. In: *Proceedings of the European Conference on Object-Oriented Programming on Object-Oriented Programming Systems, Languages, and Applications*. New York, NY, USA : ACM, 1990 (OOPSLA/ECOOP '90), S. 303–311. – ISBN 978-0-89791-411-6

[Braithwaite 2016] Braithwaite, Reginald: *Why Are Mixins Considered Harmful?* Juli 2016. – URL http://raganwald.com/2016/07/16/why-are-mixins-considered-harmful.html. – Zugriffsdatum: 2018-11-25

[Crockford] Crockford, Douglas: *Classical Inheritance in JavaScript*. – URL https://www.crockford.com/javascript/inheritance.html. – Zugriffsdatum: 2018-12-16

[Crockford 2008] Crockford, Douglas: *JavaScript: The Good Parts ; [Unearthing the Excellence in JavaScript]*. 1. ed. Beijing : O'Reilly, 2008. – OCLC: 255031813. – ISBN 978-0-596-51774-8

[Croll 2011] Croll, Angus: *A Fresh Look at JavaScript Mixins*. Mai 2011. – URL https://javascriptweblog.wordpress.com/2011/05/31/a-fresh-look-at-javascript-mixins/. – Zugriffsdatum: 2018-11-25

[Elliott 2014] Elliott, Eric: *Programming JavaScript Applications*. First edition. Beijing; Sebastopol : O'Reilly, 2014. – OCLC: ocn867765966. – ISBN 978-1-4919-5029-6

[Elliott 2017a] Elliott, Eric: *Functional Mixins*. Juni 2017. – URL https://medium.com/javascript-scene/functional-mixins-composing-software-ffb66d5e731c. – Zugriffsdatum: 2018-12-18

[Elliott 2017b] Elliott, Eric: *JavaScript Factory Functions with ES6+*. Juli 2017. – URL https://medium.com/javascript-scene/javascript-factory-functions-with-es6-4d224591a8b1. – Zugriffsdatum: 2018-12-18

[Fagnani 2015] Fagnani, Justin: *"Real"Mixins with JavaScript Classes*. Dezember 2015. – URL http://justinfagnani.com/2015/12/21/real-mixins-with-javascript-classes/. – Zugriffsdatum: 2018-12-11

[Gamma 1995] Gamma, Erich (Hrsg.): *Design Patterns: Elements of Reusable Object-Oriented Software*. 32. printing. Reading, Mass : Addison-Wesley, 1995 (Addison-Wesley professional computing series). – ISBN 978-0-201-63361-0

[Mozilla Developer Network a] Mozilla Developer Network: *The Arguments Object*. – URL https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/arguments. – Zugriffsdatum: 2018-12-16

[Mozilla Developer Network b] Mozilla Developer Network: *Object.setPrototypeOf()*. – URL https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/setPrototypeOf. – Zugriffsdatum: 2018-12-09

[Seibel 2009] Seibel, Peter: *Coders at Work: Reflections on the Craft of Programming*. New York : Apress, 2009. – OCLC: ocn286534539. – ISBN 978-1-4302-1948-4 978-1-4302-1949-1

[Simpson 2014] Simpson, Kyle: *This & Object Prototypes*. First edition. Beijing; Sebastopol, CA : O'Reilly, 2014 (You don't know JS). – OCLC: ocn891619771. – ISBN 978-1-4919-0415-2

[Stefanov 2010] Stefanov, Stoyan: *JavaScript Patterns : [Build Better Applications with Coding and Design Patterns]*. 1. ed. Beijing [u.a.] : O'Reilly, 2010 (Yahoo! Press). – ISBN 978-0-596-80675-0

[Steimann und Keller 2010] Steimann, Friedrich ; Keller, Daniela: *Objektorientierte Programmierung: Vorlesungsskript Zum Kurs 1814*. Hagen : Fernuniversität Hagen, 2010

[TC39 und Terlson 2018] TC39, ECMA ; Terlson, Brian: *ECMAScript 2018 Language Specification*. 9th. Geneva : ECMA International, 2018. – URL https://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf. – Zugriffsdatum: 2018-11-28

A ANMERKUNGEN ZU ES6 KLASSEN UND VERERBUNG

Die meisten landläufig bekannten objektorientierte-Programmiersprachen (OO) basieren auf Klassen. In klassischen OO-Sprachen wird für jedes Objekt zunächst eine Klasse definiert, die als Blaupause, bzw. abstraktes Modell für Objekte eines bestimmten Typs dient. Wenn nun ein konkretes Objekt –eine Instanz– benötigt wird, so wird es anhand dieses vorher definierten Bauplans erstellt. Es handelt sich damit quasi um die „materialisierte Kopie“ des Bauplans. In der klassischen Objektorientierung gibt es kein Objekt, zu dem nicht im Vorfeld eine Klasse definiert wurde.

Ein weit verbreitetes Code-Reuse-Muster in klassischen Programmiersprachen ist die *Vererbung*. Dabei werden Hierarchien von Klassen erstellt, deren Definitionen hierarchisch aufeinander aufbauen. Die in der Hierarchie weiter unten stehenden Klassen erben dabei Eigenschaften von Elternklassen, die weiter oben definiert wurden.

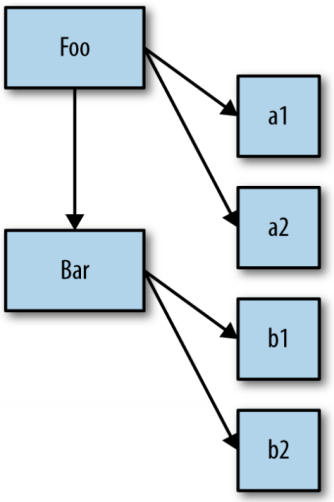

Abbildung A.1: Vererbungshierarchie in einer klassischen OO-Sprache (aus [Simpson, 2014, p. 70]).

In Abbildung A.1 ist eine solche (einfache) Hierarchie dargestellt. Ausgehend von der Klassendefinition `Foo` werden die beiden Objekte `a1` und `a2` instantiiert. Die Klasse `Bar` erbt alle Eigenschaften von `Foo` und kann eigene Spezialisierungen hinzufügen. Die konkreten Objekte `b1` und `b2` haben alle Fähigkeiten, die auch in der Klassendefinition `Foo` definiert wurden und haben zusätzlich noch weitere Eigenschaften, die in der Klassendefinition von `Bar` angegeben sind.

Diese Art der klassischen Objektorientierung ist seit ES6 durch die Benutzung des Schlüsselwortes `class` auch in Javascript möglich. Es werden Klassen wie z. B. `class Foo {...}` als Blaupausen definiert. Diese können mittels `class Bar extends Foo{...}` zu klassischen Vererbungshierarchien ausgebaut werden. Damit lässt sich in Javascript

sowohl syntaktisch als auch semantisch sehr ähnlich programmieren wie in klassischen OO-Sprachen.

Im Kern ist JavaScript eine prototypenbasierte Programmiersprache, in der Objekte für sich alleine stehen und in der andere Techniken der Code-Wiederverwendung zur Verfügung stehen. Einige anerkannte Mitglieder der Javascript Community sind sogar der Meinung, dass die Einführung klassischer Sprachmittel eher nachteilig ist und damit mehr Probleme einhergehen als gelöst werden. Sie kritisieren die mit `class` eingeführte klassische Semantik in Javascript zum Teil heftig:

In [Elliott, 2014, p. 48ff. „Classical Inheritance Is Obsolete"] führt Eric Elliot als Argumente gegen die klassische Vererbung u. a. an:

- „Tight Coupling": Vererbung zwischen Klassen ist eine der engsten Kopplungen zwischen Komponenten, die in der Programmierung möglich ist. Die erbende Subklasse muss intime Kenntnisse der Implementierung der Basisklasse haben. Daraus ergibt sich direkt das *Fragile Base Class Problem*, [Steimann und Keller, 2010, §6.2]).
- „Inflexible Hierarchies": Nach einer Weile der Nutzung und bei steigender Benutzerbasis erweisen sich letztendlich sämtliche Klassenhierarchien als falsch, um bestimmte neue Nutzungsfälle damit zu modellieren. Durch die enge Kopplung ist es aber extrem schwierig bis unmöglich, diese Fehler durch Refactoring zu beheben.
- „Gorilla/Banana Problem": Die Vererbung läuft nach dem „Alles oder Nichts"-Prinzip und es ist nicht möglich nur einzelne Eigenschaften einer Basisklasse zu erben. Elliot zitiert [Seibel, 2009]:

  > The problem with object-oriented languages is they've got all this implicit environment that they carry around with them. You wanted a banana but what you got was a gorilla holding the banana and the entire jungle.

- „Duplicating by necessity": Aufgrund der angesprochenen Probleme der klassischen Vererbung kommt es in vielen Applikationen dazu, dass entgegen aller Design-Prinzipien, Code Wiederverwendung durch Copy/Paste geschieht und damit der Idee der Vererbung zuwider läuft.

Auch Douglas Crockford, Autor des Standardwerks „JavaScript: the good parts" ([Crockford, 2008]) spricht sich letztendlich gegen klassische Vererbung in Javascript aus:

> I have been writing JavaScript for 14 years now, and I have never once found need to use an uber function. The super idea is fairly important in the classical pattern, but it appears to be unnecessary in the prototypal and functional patterns. I now see my early attempts to support the classical model in JavaScript as a mistake. [Crockford]

Als letzter prominenten Vertreter der Kritiker sei hier noch Kyle Simpson erwähnt, der über die mit ES6-eingeführten Klassen in Javascript sagt:

> Bottom line: if the ES6 `class` makes it harder to robustly leverage `[[Prototype]]`, and hides the most important nature of the JS object mechanism –the live delegation links between objects– shouldn't we see class as creating more troubles than it solves, and just relegate it to an antipattern? [Simpson, 2014, p. 153]

Entsprechend dieser kritischen Einschätzung wurde in dieser Arbeit *nicht* weiter auf die Möglichkeiten eingegangen, Javascript wie eine klassische OO-Sprache zu benutzen. Obwohl die mit ES6 eingeführten Klassen viele interessante Eigenschaften haben, ist es wichtig, Javascript zuerst als prototypenbasierte Sprache zu verstehen. Erst dann kann fundiert entschieden werden, welches Sprachmittel gewinnbringend eingesetzt werden soll.

Bei der für viele, aus anderen klassischen Sprachen kommende, Entwicklerinnen verlockenden Möglichkeit, über den ES6 `class`-Mechanismus auch JavaScript wie eine klassische Sprache zu benutzen, wird viel Potential verschwendet, das in der prototypischen und damit flexibleren Objektorientierung von JavaScript liegt.

Aus meiner Sicht ist es daher wichtig, vom prototypischen Ursprung von JavaScript aus zu denken und erst dann, wenn das nicht mehr ausreicht, die Erweiterungen zu betrachten, die der Sprache ein klassisches Gewand überstreifen. Andernfalls besteht die Gefahr, dass mit einen klassischen Hammer in der Hand jedes Problem aussieht wie ein Klassennagel.