Fortgeschrittene Techniken der Objektorientierung

Douglas Crockford, Lichtgestalt der finsteren JavaScript-Welt und Erfinder des JSON-Formats, muss frustriert gewesen sein, als er 2001 einen Artikel verfasste mit dem Titel „JavaScript: The World's Most Misunderstood Programming Language“ [1]. Auch wer kein leidenschaftlicher JavaScript-Entwickler ist, dürfte bei der Lektüre nicht ungerührt bleiben, denn in gewisser Weise ist der Beitrag beißende Polemik und zärtliche Liebeserklärung zugleich. Ein Artikel über eine Sprache, die flexibel und mächtig ist wie kaum eine andere, vielleicht aber aufgrund eben diesen Umstands in der Vergangenheit häufig nur auf Unverständnis und Ratlosigkeit gestoßen ist. Gerade die ersten Bücher zu JavaScript behandeln das Thema Objektorientierung stiefmütterlich - wenn überhaupt.

Auch in Zeiten von Ajax und Web 2.0 und der damit einhergehenden Renaissance der Programmiersprache JavaScript gibt es eine gewaltige Grauzone im Entwicklerlager, die flankiert wird von enthusiastischen Verfechtern auf der einen und dilletantischen Formularprüfern auf der anderen Seite. In der Mitte aber herrscht bange Ratlosigkeit. Die Zerrissenheit wurde JavaScript schon in die Wiege gelegt: Aufgerieben zwischen willfährigen Marketing-Spezialisten kennen wir die Sprache als Mocha, LiveScript, JScript, EcmaScript und ActionScript. JavaScript ist als Programmiersprache die *lingua franca* des Webs, fast jeder PC auf dieser Welt beeinhaltet mindestens einen Interpreter. JavaScript ist so dynamisch, objektorientiert und universell, dass eine moderne JavaScript-Bibliothek wie Prototype [2] kurzerhand den Ruby-Kernel inkl. der Standard-Klassen nach JavaScript portiert hat - elegante Iteratoren-Syntax und zahlreiche Patterns inklusive. Halten wir es für´s Protokoll mal fest: JavaScript hat soviel mit Java zu tun wie Reinhold Messner mit Tiefseetauchen. Dennoch: Der Vergleich wird immer und überall gezogen. Nehmen wir also im Rahmen dieses Artikels die Herausforderung an und klopfen JavaScript auf jene Features ab, die man an einer Hochsprache wie Java rühmt, z.B. Interfaces oder Zugriffsschutz. Geht nicht? Geht doch! Denn JavaScript ist so flexibel, dass man sich diese Features selbst hinzuprogrammieren kann. Und einige mehr dazu, die Java nicht kennt, z.B. die automatische Erstellung von Getter- und Setter-Methoden, wie sie Ruby-Programmierer schätzen. Im Wesentlichen sind drei Features von JavaScript verantwortlich für ein Höchstmaß an Eleganz und Flexibilität: die Erweiterbarkeit von Instanzen zur Laufzeit auf der Grundlage einer Prototypen-basierten Objektorientierung, die Möglichkeit, eine Funktion als Argument an eine andere Funktion zu übergeben und - nicht zuletzt - Closures. Die meisten Beispiele in diesem Beitrag verwenden diese Features auf die eine oder andere Art.

Unverständlich ist mir, warum sich die meisten Entwickler so schwer tun mit JavaScript, wo die Anleihen an andere Sprachen Legion sind: C, Lisp oder Scheme. JavaScript kennt auch Lambda-Ausdrücke, doch dazu später mehr. Fatal auch, dass im Grunde JavaScript noch immer Nachteile zugeschrieben werden, die längst schon Geschichte sind: die Ermangelung von Ausnahmebehandlungen, Vererbung oder innere Funktionen gehören hierzu. Zu Fehlern im Sprachentwurf (welche Sprache hat sie nicht) kamen fehlerhafte

Browser, schlechte Bücher, unverständliche Spezifikationen. Am Ende war JavaScript der Fußabtreter unter den Programmiersprachen. Jeder hatte eine Meinung, wenige jedoch hatten genügend Ahnung, um die Ausdruckskraft von JavaScript hinreichend zu nutzen. Für die Programmiersprache JavaScript ist Web 2.0 ein Segen, weil die damit verbundene Hinwendung zum Browser zur Folge hat, dass man besser entwickeln möchte, eleganter, performanter. Das führt zwangsläufig zur Umsetzung der Prinzipien objektorientierter Programmierung, die im Falle von JavaScript nicht klassenbasiert, sondern Prototypen-basiert ist. JavaScript hat kein Klassenkonzept wie Java, erlaubt aber dennoch Konstruktoren. Methoden, Eigenschaften. Diese objektorientierten Features werden JavaScript häufig abgesprochen - mit Verweis auf mangelnde Fähigkeiten der Kapselung oder der Vererbung. Beides ist möglich - mit einer überraschenden Anzahl an Implementierungsmöglichkeiten. Beginnen wir mit einem Feature, das - typisch für JavaScript! - unterschätzt wird, jedoch die Grundlage darstellt für zahlreiche fortgeschrittene Möglichkeiten objektorientierter Programmierung:

Zugriffsschutz

Listing 1 zeigt, wie sich „private“ Methoden in JavaScript durch das Weglassen eines Methoden-Pointers realisieren lassen. Nur die Methoden, die durch das Schlüsselwort `this` einen Methoden-Pointer erhalten, sind öffentlich sichtbar.

```
function MyClass( parameterA, parameterB ) {
    propertyR = "propertyR is only readable.";

    // private
    propertyA = parameterA;
    propertyB = para meterB;

    // methods
    this.getPropertyA = function() {
        return propertyA;
    };

    this.setPropertyA = function( para ) {
        propertyA = para;
    };

    this.getPropertyB = function() {
        return propertyB;
```

```
    };

    this.setPropertyB = function( para ) {
        propertyB = para;
    };

    this.getPropertyR = function() {
        return propertyR;
    };

    this.doSomeAction1 = function() {
        alert( aPrivateMethod() + " " + this.getPropertyR() );
    };

    function aPrivateMethod() {
        return "aPrivateMethod() is a private method.";
    };
};


var obj = new MyClass( "A", "B" );
alert( obj.getPropertyA() ); // shows "A"
alert( obj.getPropertyB() ); // shows "B"

obj.setPropertyA( "Y" );
obj.setPropertyB( "Z" );

alert( obj.getPropertyA() ); // shows "Y"
alert( obj.getPropertyB() ); // shows "Z"

obj.doSomeAction1();

alert( obj.propertyB ); // undefined
```

```
// error
alert( obj.aPrivateMethod() );
```

Listing 1

Wir sollten diesem Aspekt etwas mehr Aufmerksamkeit zukommen lassen, nicht zuletzt, weil in der Literatur allzu oft kolportiert wird, dass JavaScript über keinen Zugriffsschutz verfügt. Das ist grundlegend falsch. Die Grundfesten der Objektorientierung in Sprachen wie C++ oder Java bilden die Konzepte der Klasse und des Objekts. Eine Klasse ist ein strukturierter komplexer Typ, der als eine Art Vorlage für die zugehörigen Objekte, auch Instanzen genannt, dient. Letztere werden nach dem vorgegebenen Schema der Klasse erzeugt. Weiterhin definieren Klassen lediglich die Datenstruktur und die entsprechenden Methoden, auf ihnen selbst kann jedoch nicht operiert werden. Das ist nur mit den konkreten Ausprägungen, eben den Objekten, möglich (sehen wir einmal von statischen Funktionen ab). JavaScript kennt im Gegensatz zur klassenbasierten Vererbung die Vererbung basierend auf Prototypen. Dieser Ansatz differenziert nicht zwischen Objekten und Klassen: Statt Klassenschablonen gibt es prototypische Objekte. Jedes Objekt kann zur Laufzeit um beliebige Methoden und Attribute erweitert werden. Dieser konzeptionelle Unterschied hat in der Vergangenheit zahlreiche Autoren dazu verleitet, JavaScript ein Geheimnisprinzip (vgl. Zugriffsmodifizierern `public`, `protected`, `private` in Java) abzusprechen. Zu unrecht, da es durchaus möglich ist, in JavaScript nicht-öffentliche Methoden zu erzeugen. Schauen wir JavaScript zunächst mal unter die Haube. In JavaScript ist alles ein Objekt: Arrays sind Objekte, Funktionen sind Objekte, Objekte sind Objekte. Oder besser gesagt: Schlüssel-Wert-Paare. Die Schlüssel sind Strings, die Werte was immer JavaScript an Datentypen hergibt. Lehnen wir uns etwas aus dem Fenster und nennen einen Wert, der eine Funktion darstellt, eine Methode. Mithilfe des Schlüsselwortes `this` greifen wir dann auf Werte einer Instanz zurück. Zunächst einmal sind alle Mitglieder eines Objekts öffentlich, egal ob wir sie im Konstruktor definieren oder sie an die `prototype`-Eigenschaft hängen:

```
function MyClass( val ) {
    this.member = val;
};

var my = new MyClass( 'a' );
```

Listing 2

Die Eigenschaft `my.member` enthält `a`. In der Praxis definiert man häufig Eigenschaften im Konstruktor, während Methoden an die `prototype`-Eigenschaft angehängt werden:

```
MyClass.prototype.getMember() {
    return this.member;
};
```

Listing 3

Private Eigenschaften sind nun jene, die im Konstruktor verwendet werden - und zwar einzig unter Verwendung des Schlüsselworts `var`.

```
function My Class( val ) {
    this.member = val;

    var value = 3
    var self  = this;
};
```

Listing 4

Die Klasse verfügt somit über drei (!) private Eigenschaften: `val`, `value` und `self`. Sie sind an das Objekt gebunden, aber sie sind nicht sichtbar von außen. Sie sind noch nicht mal sichtbar für die öffentlichen Methoden des jeweiligen Objekts. Sie sind einzig sichtbar für die privaten Methoden, die realisiert werden als innere Funktionen des Konstruktors:

```
function MyClass( val ) {
    function say() {
        alert( value );
    };

    this.member = val;

    var value = 3
    var self  = this;
};
```

Listing 5

Die Konvention empfiehlt, immer eine Variable `self` oder auch `that` einzuführen, die die Instanz auch für private Methoden verfügbar macht:

```
function MyClass( val ) {
    function say() {
        alert( self.member );
    };
```

```
    this.member = val;

    var value = 3
    var self  = this;

    say();
};

var test = new MyClass( 5 );
```

Listing 6

Private Methoden können nicht durch öffentliche Methoden aufgerufen werden. Das ist natürlich nur bedingt erstrebenswert. Um das Konzept der privaten Methoden nützlicher zu machen, führen wir privilegierte Methoden ein. Eine privilegierte Methode kann auf private Eigenschaften und Funktionen zugreifen, ist selbst aber sichtbar für öffentliche Methoden von außerhalb. Privilegierte Methoden erzeugt man unter Verwendung von `this` im Konstruktor.

```
function MyClass( val ) {
    function say() {
        alert( self.member );
    };

    this.member = val;

    var value = 3
    var self  = this;

    this.sayHello = function() {
        say();
    };
};

var test = new MyClass( 5 );
test.sayHello();
```

Listing 7

`sayHello` ist eine privilegierte Methode, auf die von außen zugegriffen werden kann, die selbst aber wiederum Zugriff auf die private innere Funktion `say` hat. Sie werden zugeben müssen, dass diese Vorgehensweise einen ähnlichen Zugriffsschutz gewährleistet, wie wir es z.B. von Java, PHP 5, Python oder Ruby kennen. Möglich ist diese Funktionsweise in JavaScript durch Closures. Innere Funktionen haben in JavaScript immer Zugriff auf die Variablen in der umgebenden Funktion. Das ist eine mächtige Funktion von JavaScript, die in der Literatur häufig unterschlagen wird. Eine der wenigen guten Einführungen finden Sie unter http://jibbering.com/faq/faq_notes/closures.html. Bedenken Sie, dass private und privilegierte Eigenschaften und Funktionen eines Objekts nur bei der Instanziierung des Objekts erstellt werden können, öffentliche hingegen zu jeder Zeit. Übrigens können wir auch statische Member privat machen. Die Vorgehensweise baut dabei auf dem oben genannten Prinzip auf: Der Aufruf des Konstruktors definiert ein Closure, das alle Parameter, lokale Variablen und Funktionen mit dem Objekt assoziiert. Innere Funktionen, die an Eigenschaften der Instanz gebunden werden (z.B. mit `this.myMethod = function() {...};`), sind dann „privilegiert“, weil sie direkten Zugriff auf die privaten Eigenschaften des Objekts haben. Nur durch diese privilegierten Methoden kann Zugriff genommen werden auf private Eigenschaften, nicht jedoch durch öffentliche Methoden. Wenn Sie mit anderen objektorientierten Sprachen vertraut sind, wird Ihnen der von Douglas Crockford geprägte Begriff „privileged“ sicher komisch vorkommen, aber er beschreibt auf recht anschauliche Art die Sonderrolle der inneren Funktionen des Konstruktors. Java kennt neben Modifiern wie `private`, `protected` und `public` noch andere, z.B. `static`. Ein statisches Member ist Mitglied der Klasse, nicht eines Objekts. Von diesem Member existiert zur Laufzeit nur eine Kopie. Üblicherweise werden statische Member in JavaScript als Eigenschaften des Konstruktors definiert:

```
function MyClass() {
};


MyClass.counter = 0;
```

Listing 8

Derlei statische Member sind natürlich öffentlich, es ist jedoch auch möglich, Crockfords Ideen zu nutzen, um statische private Eigenschaften zu deklarieren. Wieder bedienen wir uns eines Closures:

```
var MyObject = ( function() {
    // private static class member
    var counter = 0;
```

```
    // private static method
    function incInstanceCount() {
        return counter++;
    };

    // class constructor
    function constructorFunc( id ) {
        this.id = id;
        var self = this;

        // call private static class method
        // and assign the returned index to
        // a private instance member
        var index = incInstanceCount();

        // privileged instance method
        this.getIndex = function() {
            return index;
        };
    };

    // privileged static class method
    // (a property of the constructor)
    constructorFunc.getInstanceCount = function() {
        return counter;
    };

    // public instance method priviliged at the
    // class level
    constructorFunc.prototype.pubInstMethod = function() {
    };

    // return the constructor
    return constructorFunc;
```

```
} )();


// public static member
MyObject.pubStatic = "anything"


// public instance member
MyObject.prototype.pubInstVar = 8;
```

Listing 9

Die gleichzeitige Definition und der Aufruf in Form eines Closures wird hier verwendet, um den Konstruktor der Klasse zurückzugeben. Dieser Konstruktor enthält nun auch private statische Eigenschaften sowie privilegierte statische Methoden - eben als Eigenschaften des Konstruktors. Das ist im Grunde die natürliche Weiterentwicklung der Vorgehensweise von Crockford - nur auf Klassenebene. In JavaScript haben innere Funktionen direkten Zugriff auf Parameter und lokale Variablen in der Funktion, in der sie enthalten sind. Im obigen Beispiel kann Code z.B. im Konstruktor die Funktion `incInstanceCount()` aufrufen. Öffentliche Instanzmethoden (also Eigenschaften von prototype) und statische Methoden (also Eigenschaften des Konstruktors, der von der inneren Funktion zurückgegeben wird) haben keinen Zugriff auf die privaten statischen Eigenschaften einer Klasse. Private statische Member funktionieren, weil alle Instanzen einer Klasse den gleichen Konstruktor haben. So können sie auch ein Closure teilen, welches den Konstruktor definiert und zurückgibt.

Eingedenk dieser Tatsache, gehen wir nun einen wesentlichen Schritt voran. Wenn Crockfords Idee bei Instanzen und Klassen funktioniert, warum nicht auch bei Gruppen von Klassen (packages in der Terminologie klassenbasierter Sprachen)? Obwohl der Ausdruck Packages sicher etwas zu hoch gegriffen ist... Dennoch: Wir können ungewöhnliche Effekte erzielen, wenn wir uns der Möglichkeiten verschachtelter Closures klar werden. Die Vorgehensweise ist recht ungewöhnlich, hat aber bereits ihre Fans gefunden, z.B. die Entwickler von TrimPath (http://www.trimpath.com). Bauen wir uns eine (anonyme) Closure zur Klassengruppierung. Beachten Sie dabei die Funktion `privateToClassGroup()`, die eine Utility-Funktion für alle Klassen dieser Gruppe enthalten könnte:

```
var global = this;


( function() {
    var classGroupMember = 3;


    function privateToClassGroup(){
```

```
    };

    global.MyObject1 = function() {
        var privteStaticMember = 4;

        function privateStaticMethod() {
        };

        function constructorFunc( id ) {
        };

        return constructorFunc;
    }();

    global.MyObject2 = function() {
        function constructorFunc( id ) {
        };

        return constructorFunc;
    }();

    global.MyObject3 = function() {
        function constructorFunc( id ) {
        };

        return constructorFunc;
    }();
} )();
```

Listing 10

Alle Instanzen einer Klasse teilen sich einen Konstruktor. Auch teilen sich alle Instanzen einer Klasse ein `prototype`-Objekt. Könnte demnach auch ein Closure assoziiert mit einem `prototype`-Objekt als Repository privater statischer Member dienen? Versuchen wir es:

```
function MyClass() {
```

```
};

MyClass.prototype = ( function() {
    // private static class member
    var privateStaticProp = "whatever";

    // private static method
    function privateStaticMethod = function() {
    };

    return ( {
        // These functions objects are shared by
        // all instances that uses this prototype
        // and they have access to the private static
        // members within the closure that returns
        // this object
        publicInstanceMethod: function() {
        },

        setSomething: function( s ) {
            privateStaticProp = s;
        }
    } );
} )();

// public instance member
MyObject.prototype.pubInstVar = 8;
```

Listing 11

Funktioniert! Und ist besonders dann empfehlenswert, wenn private Instanzvariablen nicht benötigt werden und es auch keine inneren Funktionen des Konstruktors gibt, die auf die privaten statischen Eigenschaften der Klasse zugreifen wollen. Wie auch immer, Sie kennen nun zwei Möglichkeiten, private statische Member zu definieren.

Vererbung

Aufgrund der Prototypen-basierten Objektorientierung ist Vererbung in JavaScript etwas anders gelöst als in bekannten objektorientierten Sprachen wie z.B. Java. Damit einher gehen gewisse Fallstricke. Nehmen wir uns einmal ein klassisches Beispiel vor:

```
function Animal( name ){
    this.name = name;
    this.offspring = [];
};

Animal.prototype.haveABaby = function() {
    var newBaby = new Animal( "Baby " + this.name );
    this.offspring.push( newBaby );

    return newBaby;
};

Animal.prototype.toString = function() {
    return '[Animal "' + this.name + '"]';
};



function Dog( name ) {
    this.name = name;
};

// Here's where the inheritance occurs
Dog.prototype = new Animal();

// Otherwise instances of Dog would have a constructor of Animal
Dog.prototype.constructor = Dog;

Dog.prototype.toString = function() {
    return '[Dog "' + this.name + '"]';
};
```

```
var someAnimal = new Animal( 'Sumo' );
var myPet = new Dog( 'Spinky Bilane' );

// results in 'someAnimal is [Animal "Sumo"]'
alert( 'someAnimal is ' + someAnimal );

// results in 'myPet is [Dog "Spinky Bilane"]'
alert( 'myPet is ' + myPet );

// calls a method inherited from Animal
myPet.haveABaby();

// shows that the dog has one baby now
alert( myPet.offspring.length );

// results in '[Animal "Baby Spinky Bilane"]'
alert( myPet.offspring[0] );
```

Listing 12

Schauen Sie sich noch einmal die letzte Zeile an. Das Baby eines Hundes sollte doch auch ein Hund sein, nicht wahr? Die `haveABaby`-Methode hat ihre Arbeit korrekt verrichtet, weil sie explizit eine neue Instanz von Animal erzeugt hat. Wir könnten nun natürlich eine `haveABaby`-Methode innerhalb der `Dog`-Klasse implementieren, elegant wäre es allerdings nicht. Viel besser wäre es, wenn die Methode der Basisklasse gleich ein Objekt vom richtigen Typ erzeugen würde, z.B. mithilfe dieser Methode:

```
Animal.prototype.haveABaby = function() {
    var newBaby = new this.constructor( "Baby " + this.name );
    this.offspring.push( newBaby );

    return newBaby;
}

// ...
```

```
// same as before: calls the method inherited from Animal
myPet.haveABaby();

// now results in '[Dog "Spinky Bilane"]'
alert( myPet.offspring[0] );
```

Listing 13

Jede Instanz in JavaScript kennt eine Eigenschaft `constructor`, die auf den eigenen Konstruktor verweist. Mit diesem Wissen, haben wir nun eine Methode, die immer den korrekten Konstruktor aufruft. Was nun, wenn man aber explizit den Konstruktor der Elternklasse aufrufen möchte? Derzeit kennt JavaScript noch keine Eigenschaft `super`, die auf die Elternklasse verweist, stattdessen können wir aber die `call`-Methode des `Function`-Objektes verwenden, die uns diese Funktionalität bietet.

```
Dog.prototype.haveABaby = function() {
    Animal.prototype.haveABaby.call( this );
    alert( "I am a dog" );
};
```

Listing 14

Wenn Sie dem Methodenaufruf Parameter hinzufügen wollen, so können Sie diese nach dem `this` platzieren. Die oben gezeigte Konstruktion ist ziemlich gewöhnungsbedürftig. Da wir als JavaScript-Entwickler natürlich für die Flexibilität der Sprache Reklame machen wollen, schlage ich einen alternativen Weg vor:

```
// ...
Dog.prototype = new Animal();
Dog.prototype.constructor = Dog;
Dog.prototype.superclass  = Animal.prototype;
// ...
Dog.prototype.haveABaby = function(){
    var theDoggy = this.superclass.haveABaby.call( this );
    alert( "I am a dog" );

    return theDoggy;
};
```

Listing 15

Hier speichern wir die Elternklasse in der Eigenschaft `superclass`, damit wir auf sie jederzeit zugreifen können. Das ist ein bisschen eleganter, aber immer noch viel zu kompliziert. Bauen wir uns ein Helferlein:

```
/**
 * Helper method for easy handling of inheritance.
 *
 * @access public
 */
Function.prototype.extend = function( parentConstructor,
                                      className ) {
    var f = new Function();

    if ( parentConstructor ) {
        f.prototype = parentConstructor.prototype;
        proto = this.prototype = new f;
        proto.superclass = parentConstructor;
    } else {
        proto = this.prototype;
        proto.superclass = null;
    }

    proto.constructor = this;
    proto._prototype  = this.prototype;

    if ( className ) {
        proto.classname = className;
    }

    return proto;
};
```

Listing 16

Diese Funktion hilft uns dabei, das Ableiten von Klassen künftig einfacher zu lösen. Über die unbedingt notwendige Funktionalität heraus speichert die Funktion auch den Namen der Klasse als String um auf Reflection basierende Features zu erleichtern. Damit sieht Vererbung für uns künftig so aus:

```
function BaseClass() {
    // Some code here
};

_pt = BaseClass.extend( null, "BaseClass" );



function SubClass() {
    BaseClass.call( this );
};

_pt = SubClass.extend( BaseClass, "SubClass" );


_pt.someMethod() {
    // Some code here
};
```

Listing 17

Bedeutend einfacher! Und kürzer, denn das `[Klasse].prototype` können wir uns in Zukunft sparen. Wie in der JavaScript-Welt üblich, reicht bereits ein kleines Code-Snippet, um die Gemüter zu erhitzen. Was könnte man gegen diese einfache Erweiterung des nativen Function-Objekts einwenden? Na, zum Beispiel, dass es überhaupt eine unschöne Sache ist, native Objekte zu erweitern. Das ist Geschmackssache - und es ist ganz nach meinem Geschmack, allerdings mit einer Ausnahme: Erweiterungen der `prototype`-Eigenschaft von `Object` führen zu unerwünschten Seiteneffekten, weil dadurch die Schlüssel-Wert-Zuordnung aufgebrochen wird. Das folgende kurze Beispiel zeigt das ganze Dilemma:

```
var my_obj = {
    'cars': ['Audi', 'BMW', 'Volkswagen'],
    'foo':  'bar'
};
```

```
Object.prototype.dump = function() {
    for ( o in this ) {
        alert( o + ':' + this[o] );
    }
};

my_obj.dump();
```

Listing 18

Wir erzeugen eine Variable `my_obj`, die ein Objekt mit mehreren Schlüssel-Wert-Zuordnungen enthält. Anschließend erweitern wir die `prototype`-Eigenschaft von `Object` um eine triviale Dump-Methode, die wir anschließend aufrufen, damit sie durch unser zuvor angelegtes Objekt iteriert und die einzelnen Schlüssel-Wert-Zuordnungen anzeigt. Das Problem: Die Methode `dump` ist nun auch ein Mitglied des Objektes `my_obj`, was natürlich auf keinen Fall wünschenswert ist. Viele Autoren haben große Anstrengungen unternommen, dieses Verhalten zu umgehen, indem sie z.B. neue Methoden einem speziellen Stack hinzugefügt haben, der bei jeder Verarbeitung eines Objekts befragt wurde. Ein guter Rat: Unterlassen Sie es einfach, `Object.prototype` zu erweitern, sie ersparen sich eine Menge Probleme.

Lassen Sie uns aber noch einmal das Hundebaby-Beispiel anschauen. Wir haben nämlich ein Problem erzeugt, dass sich nicht gleich erschließt. Erschaffen wir zunächst einmal ein zweites Hundebaby:

```
var myPet2 = new Dog( 'Pinsel' );

// results in 'myPet2 is [Dog "Pinsel"]'
alert( 'myPet2 is ' + myPet2 );

myPet2.haveABaby();

// results in '2'
alert( myPet2.offspring.length );
```

Listing 19

Das Objekt `myPet2` hat zwei Babies gespeichert, obwohl es nur eins haben sollte! Warum? Weil `Dog.prototype = new Animal();` eine einzelne Instanz von Animal in die `prototype`-Chain von Dog eingebracht hat. Jede Instanz von `Dog` verändert die gleiche Eigenschaft `offspring` einer einzigen Instanz von Animal. Die Eigenschaften der Instanz der Elternklasse sind `prototype`-Eigenschaften geworden und

werden somit von allen Instanzen verwendet. Es gibt mehrere Möglichkeiten, dieses Verhalten zu umgehen. Die trivialste Möglichkeit ist die „Maskierung“: Dabei definieren wir einfach alle Eigenschaften erneut im Konstruktor der abgeleiteten Klasse, beispielsweise so:

```
function Dog( name ) {
    this.name = name;
    this.offspring = [];
};
```

Listing 20

Das ist natürlich kein ernst zu nehmender Vorschlag, weil diese Vorgehensweise dem Wesen der Ableitung zuwiderläuft. Wenn wir zur Implementierung der Klasse `Dog` die interne Datenstruktur von `Animal` wissen müssen, dann müssen wir uns vorwerfen lassen, das Prinzip objektorientierter Programmierung nicht verstanden zu haben, indem wir Kapselung verhindern. Hinzu kommt, dass im obigen Beispiel der Wert `name` nicht an den Konstruktor der Klasse `Animal` weitergereicht werden. Wenden wir uns also einer eleganteren Lösung zu und formulieren unser Beispiel neu:

```
function Animal( name ){
    this.name = name;
    this.offspring = [];
};

Animal.prototype.haveABaby = function() {
    var newBaby = new Animal( "Baby " + this.name );
    this.offspring.push( newBaby );

    return newBaby;
};

Animal.prototype.toString = function() {
    return '[Animal "' + this.name + '"]';
};

function Dog( name ) {
    Animal.call( this, name );
};
```

```
// Here's where the inheritance occurs
Dog.prototype = new Animal();

// Otherwise instances of Dog would have a constructor of Animal
Dog.prototype.constructor = Dog;

Dog.prototype.toString = function() {
    return '[Dog "' + this.name + '"]';
};

var someAnimal = new Animal( 'Sumo' );
var myPet = new Dog( 'Spinky Bilane' );

// results in 'someAnimal is [Animal "Sumo"]'
alert( 'someAnimal is ' + someAnimal );

// results in 'myPet is [Dog "Spinky Bilane"]'
alert( 'myPet is ' + myPet );

// calls a method inherited from Animal
myPet.haveABaby();

// shows that the dog has one baby now
alert( myPet.offspring.length );

// results in '[Animal "Baby Spinky Bilane"]'
alert( myPet.offspring[0] );

var myPet2 = new Dog( 'Pinsel' );

// results in 'myPet2 is [Dog "Pinsel"]'
alert( 'myPet2 is ' + myPet2 );
```

```
myPet2.haveABaby();

// results in '1'
alert( myPet2.offspring.length );
```

Listing 21

Der Konstruktor der Klasse `Animal` wird aufgerufen. Im Kontext der `Dog`-Klasse (`this`) werden alle Elemente der `Animal`-Klasse in der Klasse `Dog` angelegt. Durch die Verwendung der Methode `Function.call` stellen wir zudem die Datenkapselung sicher. Vererbung lässt sich - Lob der syntaktischen Flexibilität von JavaScript! - aber auch anders realisieren. Douglas Crockford hat hierzu drei Sugar-Methoden geschrieben, die JavaScript ein wenig versüßen und z.B. parasitäre Vererbung ermöglichen [3].

Virtuelle Klassen

Einige objektorientierte Sprachen kennen das Konzept virtueller Klassen, also Klassen, die nicht selbst instanziiert werden können, von denen man jedoch ableiten kann. Das lässt sich in JavaScript sehr einfach implementieren, indem wir die virtuelle Klasse als Objekt anlegen und nicht als Funktion. Wenn die Klasse keine Funktion ist, kann sie auch nicht als Konstruktor verwendet werden:

```
LivingThing = {
    beBorn : function() {
        this.alive = true;
    }
};

function Animal( name ) {
    this.name = name;
    this.offspring = [];
};

Animal.prototype = LivingThing;
// Note: not 'LivingThing.prototype'
Animal.prototype.superclass = LivingThing;

Animal.prototype.haveABaby = function() {
    this.parent.beBorn.call( this );
```

```
    var newBaby = new this.constructor( "Baby " + this.name );
    this.offspring.push( newBaby );

    return newBaby;
};

Animal.prototype.toString = function() {
    return '[Animal "' + this.name + '"]';
};

// results in 'someAnimal is [Animal "Sumo"]'
alert( 'someAnimal is ' + new Animal( 'Sumo' ) );

// error!
new LivingThing();
```

Listing 22

Interfaces

In der objektorientierten Programmierung vereinbaren Schnittstellen (engl. Interface) gemeinsame Signaturen von Klassen. Das heißt, eine Schnittstelle vereinbart die Signatur einer Klasse, die diese Schnittstelle implementiert. Das Implementieren einer Schnittstelle stellt eine Art Vererbung dar. Die Schnittstelle gibt an, welche Methoden vorhanden sind, bzw. vorhanden sein müssen. Schnittstellen repräsentieren eine Garantie bezüglich der in einer Klasse vorhandenen Methoden. Sie geben an, dass alle Objekte, die diese Schnittstelle besitzen, gleich behandelt werden können. Gleich vorweg: Nehmen Sie die folgende Möglichkeit, Interfaces in JavaScript zu verwenden, nicht allzu ernst. Da es sich hierbei nicht um ein Sprachfeature handelt, ist die Verwendung von der Disziplin des Entwicklers abhängig - was dem Wesen eines Vertrages, den ein Interface ja darstellt, eigentlich zuwiderläuft. Bohren wir ein weiteres Mal das `Function`-Objekt auf.

```
/**
 * Ensures that a function fulfills an interface.
 *
 * Since with ECMA 262 (3rd edition) interfaces
 * are not supported yet, this function will
 * simulate the functionality. The arguments for
```

```
 * the function are all classes that the current
 * class will implement. The function checks whether
 * the current class fulfills the interface of the
 * given classes or not.
 *
 * @throws Error
 * @access public
 */
Function.prototype.fulfills = function() {
    var I;

    for ( var i = 0; i < arguments.length; ++i ) {
        I = arguments[i];

        if ( typeof I != "function" || !I.prototype ) {
            throw new Error( "Not an interface." );
        }

        if ( !this.prototype ) {
            throw new Error( "Current instance is " +
                "not a function definition." );
        }

        for ( var f in I.prototype ) {
            // don't take properties into consideration
            // which were added in Function.extend
            if ( f.toString() == "classname" ||
                 f.toString() == "superclass" ) {
                continue;
            }

            if ( typeof I.prototype[f] != "function" ) {
                throw new Error( f.toString() +
                    " is not a method in Interface " +
```

```
                    I.toString() );
            }

            if ( typeof this.prototype[f] != "function" &&
                 typeof this[f] != "function" ) {
                if ( typeof this.prototype[f] == "undefined" &&
                     typeof this[f] == "undefined" ) {
                    throw new Error( f.toString() +
                        " is not defined" );
                } else {
                    throw new Error( f.toString() +
                        " is not a function" );
                }
            }
        }
    }
};
```

Listing 23

Dazu schreiben wir uns ein kleines Beispiel, das die Verwendung von - naja - Interfaces in JavaScript illustriert:

```
function MyInterface() {
    this.constructor.fulfills( MyInterface );
};

MyInterface.prototype.requiredMethod = function() {};

function MyClass() {
    MyInterface.call( this );
};

var cls = new MyClass();
```

Listing 24

`MyInterface` stellt in diesem Beispiel das Interface dar, das eine Methode `requiredMethod` enthält, die es in Klassen, die dieses Interface implementieren, auszuformulieren gilt. Falls dies nicht der Fall ist, erfolgt eine Fehlermeldung. Sieht ganz nach einem echten Interface aus, finden Sie nicht? Bitte beachten Sie hierzu auch den Beitrag von Jörg Schaible über Unit-Tests in JavaScript.

Getter- und Setter-Methoden

Sind wir mal ehrlich: Entwickler objektorientierter Programme verrichten einen großen Teil ihrer Zeit mit der immer gleichen Aufgabe, nämlich dem Implementieren von Getter- und Setter-Methoden (auch Accessor- und Mutator-Methoden genannt). Das sieht dann z.B. so aus:

```
function Rectangle() {
    this._width  = 100;
    this._height = 100;
};

_pt = Rectangle.extend( null, "Rectangle" );

_pt.setWidth = function( width ) {
    this._width = width;
};

_pt.getWidth = function() {
    return this._width;
};

_pt.setHeight = function( height ) {
    this._height = height;
};

_pt.getHeight = function() {
    return this._height;
};
```

Listing 25

Langweilig? Langweilig! Und fehleranfällig! Bauen wir uns wieder ein kleines Helferlein:

```
Function.READ       = 1;
Function.WRITE      = 2;
Function.READ_WRITE = 3;

/**
 * @access public
 */
Function.prototype.addProperty = function( sName, nRdWr, v ) {
    var p           = this.prototype;
    nRdWr           = nRdWr || Function.READ_WRITE;
    var capitalized = sName.charAt( 0 ).toUpperCase() +
                      sName.substr( 1 );

    if ( nRdWr & Function.READ ) {
        p["get" + capitalized] = function() {
            return this["_" + sName];
        };
    }

    if ( nRdWr & Function.WRITE ) {
        p["set" + capitalized] = function( v ) {
            this["_" + sName] = v;
        };
    }

    if ( v ) {
        p["_" + sName] = v;
    }
};
```

```
function Rectangle() {
};

Rectangle.addProperty( 'width',  Function.READ_WRITE, 100 );
Rectangle.addProperty( 'height', Function.READ_WRITE, 100 );



var rect = new Rectangle();
alert( rect.getHeight() );
```

Listing 26

Eleganter, nicht wahr? Mithilfe der `addProperty`-Methode haben wir eine einfache Möglichkeit, Getter-, Setter- oder beide Methoden dynamisch zu erzeugen, so wie es aus Ruby bekannt ist, Default-Wert inklusive. Das Qooxdoo-Projekt geht sogar noch einen Schritt weiter und erweitert dieses Feature um zahlreiche zusätzliche Funktionen, z.B. Validierungsmechanismen oder Prüfmethoden à la `isAvailable()`.

Namensräume

Mit Namensräumen kann ein Entwickler große Programmpakete mit vielen definierten Namen schreiben, ohne sich Gedanken machen zu müssen, ob die neu eingeführten Namen in Konflikt zu anderen Namen stehen. Im Gegensatz zu der Situation ohne Namensräume wird hier nicht der ganze Name neu eingeführt, sondern nur ein Teil des Namens, nämlich der des Namensraumes. Ein Namensraum ist ein deklaratorischer Bereich, der einen zusätzlichen Bezeichner an jeden Namen anheftet, der darin deklariert wurde. Dieser zusätzliche Bezeichner macht es weniger wahrscheinlich, das ein Namenskonflikt auftritt mit Namen, die anderswo im Programm deklariert wurden. Es ist möglich den gleichen Namen in unterschiedlichen Namensräumen ohne Konflikt zu verwenden, auch wenn der gleiche Namen in der gleichen Übersetzungseinheit vorkommt. Solange sie in unterschiedlichen Namensräumen erscheinen, ist jeder Name eindeutig aufgrund des zugefügten Namensraumbezeichners.

Namensräume in JavaScript 1.x sind nicht mit Namensräumen zu vergleichen, wie es sie z.B. in C++ gibt. Wie so häufig handelt es sich hierbei eher um eine programmiertechnische Konvention, beruhend auf der einfachen Erweiterbarkeit von Objektstrukturen. Bekannte Projekte wie die Yahoo UI-Bibliothek (YUI), Qooxdoo oder Prototype nutzen diese Konvention, um Namenskonflikte zu vermeiden und Code sauber zu organisieren. So findet sich die Drag & Drop-Klasse der Yahoo UI-Bibliothek etwa hier:

```
var myDDobj = new YAHOO.util.DD( "myDiv" );
```

Der Kalender hier:

```
var myCal = new YAHOO.widget.Calendar( "calEl", "container" );
```

Ein solcher Namensraum lässt sich ganz einfach aufbauen:

```
if ( typeof MyNamespace == "undefined" ) {
    var MyNamespace = {};
}

MyNamespace.SomeClass = function() {
};
```

Listing 27

Dies ist deutlich eleganter als Namensungetüme wie `Base_Security_Passwd_Generator`, finden Sie nicht? Sollten Sie in Ihrem Projekt von zahlreichen anderen Bibliotheken Gebrauch machen, sind Sie gut beraten, Ihre Skripte in einem eigenen Namensraum anzusiedeln, um Namensüberschneidungen von vornherein auszuschließen. Bitte beachten Sie im obigen Beispiel die explizite Prüfung auf die Existenz von `MyNamespace`. Im Gegensatz zu C# darf ein Namespace nur ein einziges Mal definiert werden. Wird diese Anweisung ein weiteres Mal ausgeführt, so werden alle definierten Member gelöscht.

Es gibt sogar Hardliner, die mit Ihrem Code, den globalen Namensraum überhaupt nicht „verschmutzen“ (*namespace pollution*). Das lässt sich ganz einfach mit Hilfe eines Closures bewerkstelligen:

```
( function() {
    function MyClass() {

    }

    MyClass.prototype.sayHello = function() {
        alert( 'hello' );
    };

    // example code
    var test = new MyClass();
    test.sayHello();
})();
```

```
alert( MyClass ); // -> undefined
```

Listing 28

Mehrfachvererbung

In der Objektorientierung ist Vererbung eine Methode, neue Klassen unter Verwendung von bestehenden aufzubauen. Zu unterscheiden ist dabei Schnittstellenvererbung und Klassenvererbung. Bei der Schnittstellenvererbung „erbt“ eine abgeleitete Klasse die Signaturen von Methoden, muss die Methoden aber selbst implementieren. Bei der Klassenvererbung erbt die abgeleitete Klasse auch die Implementierung von einer oder mehreren Basisklassen. JavaScript bietet keine Klassenvererbung „out of the box“. Wir können dies aber simulieren (wenn sie nicht allzu streng mit dem Beispiel sind):

```
/**
 * Emulation of multiple inheritance for JavaScript.
 *
 * @param  mixed    arg                 Either an object or array
 *                                      of objects
 * @param  Boolean  bInheritPrototype   Whether to use prototype
 *                                      members as well
 * @access public
 */
Function.prototype.inherits = function( arg, bInheritPrototype ) {
    if ( !arg ) {
        return;
    }

    if ( typeof arg != 'object' || arg.constructor != Array ) {
        arg = [arg];
    }

    function getMembers( obj ) {
        var result = [], member;

        for ( member in obj ) {
            result[member] = obj[member];
        }
```

```
        return result;
    };

    var member, members, i;

    for ( i = 0; i < arg.length; i++ ) {
        // static members
        if ( arg[i].constructor ) {
            members = getMembers( arg[i] );

            for ( member in members ) {
                this[member] = members[member];
            }
        }

        // prototype members
        if ( bInheritPrototype && arg[i].prototype ) {
            members = getMembers( arg[i].prototype );

            for ( member in members ) {
                this.prototype[member] = members[member];
            }
        }
    }
};
```

Listing 29

Das folgende Beispiel zeigt eine Klasse `Test`, die Methoden und Eigenschaften der Klassen `Car` und `Bus` erbt.

```
Car = function() {
    Car.counter++;
};
```

```
Car.counter = 0;
Car.prototype.stop  = function() { alert( 'Stop' );  };
Car.prototype.drive = function() { alert( 'Drive' ); };

Bus = function() {
    Bus.counter++;
};

Bus.counter = 0;
Bus.crash = function() { alert( 'Crash' ); };
Bus.prototype.stopAtSchool = function() { alert( 'Pick up' ); };



Test = function() {
};

Test.inherits( [Car, Bus], true );

var ts = new Test();
ts.drive();
ts.stop();
ts.stopAtSchool();
Test.crash();
```

Listing 30

Überladen von Funktionen

Überladen bezeichnet in der Welt der Programmiersprachen die Erstellung von zwei oder mehr Funktionen mit dem selben Namen. Welche Funktion aufgerufen wird, wird anhand der deklarierten Datentypen der Parameter entschieden. Eine ähnliche Funktionalität lässt sich auch in JavaScript erreichen. Erweitern wir dazu einfach den Sprachkern:

```
/**
 * Allows functions to be overloaded (different versions of the
 * same function are called based on the arguments types).
 *
 * @access public
```

```
 * @static
 */
Function.overload = function() {
    var f = function( args ) {
        var i, l, h = "";

        for ( i = -1,
              l = ( args = [].slice.call( arguments ) ).length;
              ++i < l; h += args[i].constructor );

        if ( !( h = f._methods[h] ) ) {
            var x, j, k, m = -1;

            for ( i in f._methods ) {
                for ( k = 0, j = -1, l = Math.max( args.length,
                      x = f._methods[i][1] ); ++j < l;
                      ( args[j] instanceof x[j] ||
                        args[j].constructor == x[j] ) && ++k );
                k > m && ( h = f._methods[i], m = k );
            }
        }

        return h? h[0].apply( f, args ) : undefined;
    };

    f._methods = {
    };

    f.overload = function( f, args ) {
        this._methods[( args = [].slice.call( arguments, 1 ) ).
        join( "" )] = [f, args];
    };
```

```
    f.unoverload = function( args ) {
        return delete this._methods[[].slice.call( arguments ).
        join( "" )];
    };

    return f;
};
```

Listing 31

Die Funktion speichert verschiedene Signaturen ab in Abhängigkeit der Anzahl der Argumente sowie deren Konstruktoren. JavaScript ist nicht typlos, es ist dynamisch typisiert. Wir können also zur Laufzeit den Typ einer Variable ermitteln:

```
alert( [].constructor == Array ); // -> true
```

Listing 32

Zurück zu unserem Beispiel. Wir erzeugen nun eine Funktion `ol`, die sich je nach Anzahl und Art der übergebenen Argumente anders verhält:

```
ol = new Function.overload;

// one parameter which is a Number
ol.overload( function( x ) {
    document.write( "NUMBER<br />" );
}, Number );

// one parameter which is a String
ol.overload( function( x ) {
    document.write( "STRING<br />" );
}, String );

// two parameters, a Function and a Number
ol.overload( function( x, y ) {
    document.write( "FUNCTION, NUMBER<br />" );
}, Function, Number );

// two parameters, a Number and a String
ol.overload( function( x, y ) {
```

```
    document.write("NUMBER, STRING<br />" );
}, Number, String );

// tests
ol( function() {}, 123 );
ol( 123 );
ol( "ABC" );
ol( 123, "ABC" );
ol( {} );

// remove function with Number parameter
ol.unoverload( Number );

ol( {} );
```

Listing 33

Design Patterns

Je mächtiger die Objektorientierung einer Sprache wird, umso wichtiger und notwendiger wird die Planung objektorientierter Softwareentwicklung. Design Patterns ebnen hier einen Weg, um wiederkehrende Entwurfsprobleme bei Softwareentwicklungsprozessen zu unterbinden und Lösungen in Form von bewährten Mustern (*Patterns*) bereitzustellen, um somit die Problemsituation zu erkennen und so effizient wie möglich zu lösen. Die Intention und der Grundgedanke zur Verwendung von objektorientierter Software besteht in der Wiederverwendbarkeit (*Code Reuse*), um auch bei zukünftigen Anforderungen und Problemen zu bestehen.

Design Patterns repräsentieren Programmteilstrukturen, die einfache und elegante Lösungen für spezifische Probleme des Softwareentwurfs beschreiben. Entwurfsmuster sind Beschreibungen von Lösungen für Software-Design-Probleme. Pattern-Beschreibungen müssen in einer festgelegten Form erfolgen, damit man die Patterns miteinander vergleichen und in ein Schema einordnen kann. Für diesen Zweck haben in den letzten Jahren viele Autoren Kataloge entwickelt, die auf die Anforderungen von Design Patterns für die Softwareentwicklung abgestimmt sind. Die Struktur des bekanntesten Katalogs wurde von Erich Gamma und seinen Kollegen Richard Helm, Ralph Johnson und John Vlissides (Gang of Four, kurz GoF) entwickelt. Die meisten GoF-Patterns sind objektbasiert, d.h. sie beziehen sich auf Objekte und ihre Beziehung zueinander.

Erich Gamma und seine Mitautoren teilen Design Patterns nach zwei Klassifikationskriterien ein. Dies sind der Gültigkeitsbereich der Pattern und ihre Aufgabe. Nach dem Gül-

tigkeitsbereich werden Patterns für Klassen (klassenbasiertes Muster) und Objekte (objektbasiertes Muster) unterschieden, da ausschließlich objektorientierte Patterns beschrieben werden. Nach der Aufgabe werden Patterns in die Kategorien „Creational“ (Erzeugungsmuster), „Structural“ (Strukturmuster) und „Behavioral“ (Verhaltensmuster) eingeteilt. Erzeugungsmuster beschäftigen sich mit der Erzeugung von Objekten, Strukturmuster beschreiben die statische Zusammensetzung von Objekten und Klassen, Verhaltensmuster charakterisieren das dynamische Verhalten von Objekten und Klassen. Die Gang of Four hat insgesamt 23 Patterns beschrieben, von denen wir hier einige Umsetzungen betrachten wollen.

Factory

Die Fabrikmethode ist den klassenbasierten Erzeugungsmustern zugeordnet. Sie ist ein effizienter Mechanismus zur Kapselung der Objekterzeugung - mit der Option, Unterklassen entscheiden zu lassen, von welcher Klasse das Objekt ist. Es handelt sich um ein häufig verwendetes Muster, da die Objekterzeugung zu den Standardaufgaben in der objektorientierten Programmierung gehört. Listing 34 zeigt, wie sich mit Hilfe der BrowserAbstractionFactory-Klasse Objektinstanzen erzeugen lassen, die auf den verwendeten Browser abgestimmt sind. Ältere DHTML-Bibiliotheken schleppen noch den Ballast für alle Browser mit sich herum. Es ist auch denkbar, die factory-Methode als statische Methode an das Browser-Objekt anzuhängen und mit Browser eine Default-Implementierung zu schaffen.

```
// Base class for our Browser classes
function Browser() {
};

Browser.prototype.getBrowser = function() {
    return this.browser_type;
};

// file BrowserMoz.js
function BrowserMoz() {
    Browser.call( this );

    this.browser_type = "Mozilla";
};

_pt = BrowserMoz.extend( Browser, "BrowserMoz" );
```

```
// file BrowserOpera.js
function BrowserOpera() {
    Browser.call( this );

    this.browser_type = "Opera";
};

_pt = BrowserOpera.extend( Browser, "BrowserMoz" );

// file BrowserIE.js
function BrowserIE() {
    Browser.call( this );

    this.browser_type = "Internet Explorer";
};

_pt = BrowserIE.extend( Browser, "BrowserIE" );

function BrowserAbstractionFactory() {
    this.className  = null;
    var isSupported = true;

    var agt      = navigator.userAgent.toLowerCase();
    var is_moz   = agt.indexOf( 'gecko'  ) != -1;
    var is_opera = agt.indexOf( 'opera'  ) != -1;
    var is_ie    = ( agt.indexOf( 'msie' ) != -1 ) &&
                   ( agt.indexOf( 'opera' ) == -1 );

    if ( is_moz ) {
        this.className = "BrowserMoz";
    } else if ( is_opera ) {
        this.className = "BrowserOpera";
    } else if ( is_ie ) {
```

```
        this.className = "BrowserIE";
    }

    if ( this.className != null ) {
        document.write( "<script language=\"JavaScript\" src=\"" +
                        this.className + ".js\"><" + "/script>" );
    } else {
        isSupported = false;
        alert( "Browser not supported" );
    }

    // define factory method
    this.factory = function() {
        return eval( 'new ' + this.className + '()' );
    };
};

var browser_factory = new BrowserAbstractionFactory();
var dhtml_obj = browser_factory.factory();
alert( dhtml_obj.getBrowser() );
```

Listing 34

Das „Nachladen“ der benötigten Klassen mittels `document.write()` ist nicht sehr elegant, aber ein gangbarer Weg. Natürlich liesse sich hier das `XmlHttp`-Objekt nutzen, um einen Loader-Mechanismus zu implementieren, es würde allerdings den Rahmen dieses Beitrages sprengen.

Singleton

Das Singleton-Pattern kommt immer dann zum Einsatz, wenn von einer Klasse nur jeweils eine Instanz zur selben Zeit existieren darf. Listing 35 zeigt, wie sich das Singleton-Pattern in JavaScript realisieren lässt.

```
function SingletonExample() {
    if ( SingletonExample._singleton ) {
        return SingletonExample._singleton;
    }
```

```
    SingletonExample._singleton = this;
};

_pt = SingletonExample.extend( null, "SingletonExample" );

_pt.setValue = function( value ) {
    this.value = value;
};

_pt.getValue = function() {
    return this.value;
};

var singleton_obj = new SingletonExample();
singleton_obj.setValue( 5 ); // returns 5
alert( singleton_obj.getValue() );

var singleton_obj2 = new SingletonExample();
alert( singleton_obj2.getValue() ); // returns 5 as well
```

Listing 35

Proxy

Das Entwurfsmuster Proxy (proxy = Stellvertreter) liefert ein schönes Beispiel für die Kategorie der objektbasierten Strukturmuster. Hinter diesem Entwurfsmuster verbirgt sich eine einfache Idee: eine Stellvertreter-Klasse verbirgt eine nachgeladene Klasse. Damit kann der Zugriff auf die nachgelagerte Klasse kontrolliert werden. Dies ist z. B. sinnvoll, wenn auf die nachgelagerte Klasse nur beschränkter Zugriff gewährt werden soll. Die Arbeit verrichtet die nachgelagerte Klasse, die Schnittstelle wird durch die `Proxy`-Klasse definiert. Listing 36 zeigt die generische Umsetzung dieses Konzepts in einfachster Form.

```
function Implementation() {
    this.a = function() {
        return "a";
    }
```

```
    this.b = function() {
        return "b";
    }

    this.c = function() {
        return "c";
    }
};

function Proxy() {
    this.impl = new Implementation();

    this.get = function( which ) {
        if ( this.impl[which] ) {
            return this.impl[which]();
        } else {
            return null;
        }
    }
};

var proxy_obj = new Proxy();
alert( proxy_obj.get( "a" ) ); // returns "a"
alert( proxy_obj.get( "z" ) ); // returns null
```

Listing 36

Template Method

Das Entwurfsmuster Template Method gehört zur Kategorie der objektbasierten Verhaltensmuster und beschreibt eine grundlegende Technik zur Wiederverwendung von Code. Mit dieser Technik wird das Gerüst eines Algorithmus in einer Operation aufgebaut. So ist es möglich, bestimmte Schritte des Algorithmus jederzeit auf unterer Ebene zu ändern (durch Überschreiben der Methoden in den Unterklassen), ohne die Struktur des Algorithmus anzutasten. Es wird also von oben eine bestimmte Struktur vorgegeben, die in den Unterklassen beliebig angepasst wird. Charakteristisch für Template Method ist, dass in der Basisklasse eine Methode definiert wird, die wiederum Methoden der Basisklasse aufruft, welche in der Unterklasse überschrieben werden. Manchmal verfügen die be-

schriebenen Methoden über ein Default-Verhalten (also schon eine Implementierung in der Basisklasse), das trotzdem auf unterer Ebene durch Überschreiben entsprechend modifiziert werden kann. Die Methode `templateMethod` delegiert die Aufgabe demzufolge an die abstrakten Methoden der Basisklasse, die ihrerseits in der Unterklasse überschreiben werden. Listing 37 zeigt die JavaScript-Version dieses Patterns:

```
function Template() {
};

_pt = Template.extend( null, "Template" );

_pt.templateMethod = function() {
    this.method1();
    this.method2();
};

_pt.method1 = function() {
    throw new Error( "Not implemented" );
};

_pt.method2 = function() {
    throw new Error( "Not implemented" );
};

function Application() {
    Template.call( this );
};

_pt = Application.extend( Template, "Application" );

_pt.method1 = function() {
    alert( "Method 1" );
};
```

```
_pt.method2 = function() {
    alert( "Method 2" );
};

var app_obj = new Application();
app_obj.templateMethod();
```

Listing 37

Iterator

Objekte, die als Behälter für andere Objekte auftreten, wie z.B. `Array` oder `Object`, sollten in der Regel Methoden bereitstellen, die ein einfaches Iterieren über dieses Element erlauben. Listing 38 zeigt einen einfachen Array-Iterator:

```
function ArrayIterator( arr ) {
    this.array = ( arr != null )? arr : [];
    this.arrayCount = 0;
};

_pt = ArrayIterator.extend( null, "ArrayIterator" );

_pt.reset = function() {
    this.arrayCount = 0;
};

_pt.hasMore = function() {
    if ( this.arrayCount == this.array.length ) {
        return false;
    }

    return true;
};

_pt.next = function() {
    return this.array[this.arrayCount++];
};
```

```
_pt.count = function() {
    return this.array.length;
};

var it = new ArrayIterator( new Array( 1, 2, 4, 8, 16, 32, 64 ) );
it.reset();

while ( it.hasMore() ) {
    alert( it.next() );
}
```

Listing 38

Das obige Beispiel für die Implementierung des Iterator-Patterns ist zugegebenermaßen ziemlich trivial. Wir verdanken es in erster Linie Sam Stephenson, dem Autor des Prototype-Frameworks, dass wir heute elegantere Mechanismen haben, um Ruby-ähnliche Iteratoren zu verwenden. Prototype, streng genommen eine Portierung des Ruby-Kerns nach JavaScript, macht alle generischen JavaScript-Objekte kurzerhand zu Unterklassen einer neu eingeführten Enumerable-Klasse, die zahlreiche neue Funktionen bereithält, darunter Iteratoren und Reflection-Funktionen.

Patterns beschreiben Problem-Lösungs-Beziehungen und haben einige gewichtige Vorteile. Schon durch ihren Namen bietet sich eine einfache Möglichkeit zur Kommunikation und Dokumentation. Kennen mehrere Programmierer bestimmte Patterns, reicht ihr Name aus, um über Design-Alternativen zu sprechen. Design Patterns helfen, immer wieder vorkommende Probleme bei der Softwareentwicklung mit erprobten Lösungen in den Griff zu bekommen und die Software leichter verständlich zu machen. Sie sind damit ein Mittel, kostengünstig und zeitsparend Lösungen zu erarbeiten. Software, die lange verwendet werden soll, steht irgendwann vor dem Problem, dass sich mit der Zeit die Anforderungen, die sie zu erfüllen hat, ändern. Oft werden zusätzliche Anforderungen durch Erweiterungen der Software befriedigt. Dies führt aber zu einem inflexiblen System, das für spätere Anpassungen ungeeignet ist. Um die Software weiter zu entwickeln, muss sie umorganisiert werden. Dieser Prozess heißt Refactoring. Design Patterns beschreiben Strukturen, die das Resultat von Refactoring sind. Dadurch zeigen sie die Richtung, in die ein inflexibles Programm umorganisiert werden soll, um auch zukünftigen Anforderungen gewachsen zu sein. Verwendet man Design Patterns von Beginn an, wird der Bedarf für Refactoring zu einem späteren Zeitpunkt verringert oder sogar ganz vermieden. Patterns definieren die Strukturen, derer es bedarf, um das Zusammenspiel großer Klassen-Hierarchien zu regeln.

Fazit

Vielleicht liegt es am Suffix „script“, dass man JavaScript allenfalls kleinere Spielereien zutraut. Dabei hat sich JavaScript längst jene Bereiche erobert, die Hochsprachen so „hochnäsig“ machen. In den Browsern dieser Welt erbringt JavaScript Sekunde für Sekunde Höchstleistungen beim Rendern von Navigationsbäumen, bei der Manipulation von Datensätzen, bei der Kommunikation mit dem Server oder - vergessen wir nicht die Ursprünge - beim Validieren von Formulareingaben. Dass man JavaScript mehr und mehr zutraut, sieht man an den zahlreichen typischen Entwicklerwerkzeugen, die es immer häufiger auch für die JavaScript-Welt gibt: Unittest-Bibliotheken wie das in diesem Buch beschriebene JsUnit, Fenstersysteme wie Qooxdoo [4] oder Documentation-Tools wie JSDoc [5] nach dem Vorbild von JavaDoc. Die ECMAScript-Spezifikation, allzu häufig Quelle von Mißverständnissen, reserviert schon heute Schlüsselwörter wie `class`, `import`, `super` und `extends`. Das deutet darauf hin, dass sich künftige JavaScript-Versionen in Richtung Java und klassenbasierter Objektorientierung bewegen werden. Das Proposal für JavaScript 2.0 [6] dümpelt schon seit 2003 im Netz, ist jedoch bislang nur teilweise Realität geworden - und auch nur für Actionscript-Programmierer. Warum auch warten auf JavaScript 2.0? Die Sprache kennt jetzt schon Duck Typing, Closures, Lisp-ähnliche Features, Metaprogrammierung, Aspektorientierte Programmierung und vieles mehr. Für mich die ideale Lösung für die Webanwendungen der Zukunft.

[1] http://javascript.crockford.com/javascript.html

[2] http://prototype.conio.net/

[3] http://javascript.crockford.com/inheritance.html

[4] http://qooxdoo.org/

[5] http://jsdoc.sourceforge.net/

[6] http://www.mozilla.org/js/language/js20/