MotionJS
A JavaScript Framework for large applications

A dissertation submitted in partial fulfilment of the requirements
for the MSc in Information and Web Technologies

by Michael Sauter

Department of Computer Science and Information Systems
Birkbeck College, University of London

May 2011

Academic Declaration

This report is substantially the result of my own work except where explicitly indicated in the text. I give my permission for it to be submitted to the JISC Plagiarism Detection Service. I have read and understood the sections on plagiarism in the Programme booklet and the School's website.

The report may be freely copied and distributed provided the source is explicitly acknowledged.

London, XXth May 2011

Michael Sauter

Table of Contents

Abstract iv
1 Introduction 1
2 Background 3
2.1 JavaScript Language Features 3
2.1.1 Dynamicity 3
2.1.2 Objects and Inhertiance 3
2.1.3 Execution Context and Function Invocation 4
2.1.4 Closures 4
2.2 Server-side JavaScript 5
2.3 Dependency Injection 5
3 Analysis 7
3.1 PHP and Python 7
3.1.1 Modules and Namespaces 8
3.1.2 Classes 8
3.1.3 Access Modifiers 8
3.1.4 Interfaces 9
3.1.5 Inheritance 9
3.2 Yahoo! User Interface Library 3 (YUI3) 10
3.3 Joose 11
3.4 Requirements 13
4 Design 15
4.1 Architecture 15
4.2 Module Loading 15
4.3 Module Content 17
4.4 Object Configuration 17
4.4.1 Dependencies 17
4.4.2 Inherits 17
4.4.3 Requirements 18
4.5 Object Creation 19
4.6 Object Management 19
5 Implementation 20
5.1 Architecture 20
5.2 Object creation 20
5.3 Module Loading and Development Server 24
5.4 Build System and Deploying 28
6 Evaluation 29
7 Conclusion 37
References 39
8 Literaturverzeichnis 39

Abstract

Dorem ipsum dolor sit amet,consectetuer adipiscing elit,sed diam nonummy nibh euismod tinet unt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, qui nostrud exercitation ullamcorper suscipit lobortis nisl utaliquip ex ea commod con. Duis autem vel eum iriure dolor in hendrerit in vulputate velit essent molestie quat,vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iustos odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feug ait nulla facilisi. Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed di am nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Uti wisi enim ad minim veniam,quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat. Duis autem vel eum iriure dolor in hendrerit init. Dorem ipsum dolor sit amet,consectetuer adipiscing elit, sed diam nonummy nibh euismod tinet unt ut laoreet dolore magna aliquam erat volutpat.

1 Introduction

The JavaScript programming language was designed in 1995 to allow non-developers to add behaviour to websites [**1**]. Since then, it became more and more widely used, to the point where the vast majority of the most popular sites use JavaScript [**2**]. Especially in recent years, there has been an increased interest in the language as some websites turn into JavaScript-powered applications (e.g. Google Docs[1]). This lead to increased performance in browsers and renewed research efforts, investigating in areas such as security and correctness [2]. Next to its usage in browsers, JavaScript also plays an important role in the mobile space, as it can be executed on all systems (through browsers), or even being the main development language as in webOS [**3**]. Another area where JavaScript is getting to be used is in server-side environments [2]. New platforms like NodeJS[2] or Rhino[3] have attracted developers in recent years.

As a consequence thereof, JavaScript programs have been used by developers for more ambitious projects, pushing the language to its limits. The general assumption that JavaScript programs are little scripts used for DOM manipulation does not hold anymore [2]. This development exhibits the shortcomings of the language, which are more obvious in large systems that are composed of many scripts, containing a large amount of objects. One such lacking feature is the ability to import scripts, which are taken for granted in server-side scripting languages like PHP or Python. Other problematic aspects of the language in the sight of large systems are the missing support for encapsulation or the weak type system (see chapter 2 for details). Despite these shortcomings, developers have not abandoned JavaScript, as it is the only language natively supported in all major browsers [**4**]. Instead, it was tried to overcome the problems. Broadly speaking, three approaches can be distinguished:

1. Writing code in other languages and translating it to JavaScript
2. Extending JavaScript and compiling to pure JavaScript (e.g. Objective-J[4])
3. Using JavaScript's dynamic features to write JavaScript frameworks that provide missing features and/or ease development

While the first two approaches have been popular, they need additional steps in the development process and may miss the features that only JavaScript has to offer (e.g. the prototypal inheritance, more on this in chapter 2). The third approach has been quite popular as well, leading to a myriad of JavaScript frameworks, some of which will be discussed later on in chapter 3. Some of these frameworks try to make JavaScript more like some other language (e.g. Java) providing class-based inheritance etc, while others believe that JavaScript's concepts are not inferior to other languages, but only miss some features, which are then tried to be added.

[1] http://docs.google.com/
[2] http://nodejs.org
[3] http://www.mozilla.org/rhino/
[4] http://cappuccino.org/learn/tutorials/objective-j-tutorial.php

Most of the frameworks developed so far offer help in building interfaces (e.g. by easing DOM manipulation), as client-side JavaScript is (still) the dominant usage. Very few frameworks have tackled the problem that arise on the server-side and especially in larger systems. Throughout this work, we use the terms "large systems" or "large software" in the meaning of software that is composed of many scripts which likely make use of a big number of objects, not necessarily of software that is written by many developers or used by many people. Typically though, large systems consist of many components, that might not have been specifically written for a single project, but might be reused from previous projects, possibly from different authors. These systems put demands on a framework that differ from the current offerings: The organisation of code becomes very important, dependencies between scripts (and objects) have to be managed and maintainability is one of the biggest concerns.

The framework developed during this project aims to aid the developer in the development of systems with these requirements. The goal is to create a framework (which we'll call MotionJS) which provides features missing in JavaScript to efficiently support code organisation and reuse, and add features necessary for developing large systems, such as dependency injection. As these additions are helpful both client and server-side, MotionJS should run without modification in both environments (more on this in chapter 4).

In the following, we're going to describe the development of the framework, beginning from its analysis and design process through to its realisation and the evaluation of the results. Chapter 2 starts with a short introduction to JavaScript's main features and server-side capabilities necessary to understand the implementation details later on. Chapter 3 lays the foundation for the design by analysing the scripting languages PHP and Python and identifying features available there but missing in JavaScript. Furthermore, it presents two JavaScript frameworks to elaborate what these provide in order to help developers write large-scale applications. Based on this analysis, requirements for the frameworks are distilled, which lead to the proposed design for our framework in chapter 4. Following that, chapter 5 describes the implementation of this design, before we evaluate the developed framework in chapter 6 by demonstrating its features and comparing it to the existing solutions. In the final chapter, conclusions from the project are drawn and possible future work is indicated.

2 Background

This chapter will start with a look at JavaScript language features that underpin the envisioned framework and are therefore essential to understand the implementation detailed in chapter 5. It is however not intended to be a complete list of all JavaScript language features. Secondly, the basics of JavaScript on the server are explained as this is a fairly recent development and cannot be considered common knowledge. Finally, a general outline of Dependency Injection – and reasons why it will play such an important role in the design process – are given.

2.1 JavaScript Language Features

The features described in this section refer to the ECMAScript 3 Language specification [**5**], finalized in December 1999. ECMAScript 5 [**6**] (version 4 was skipped) has been passed in 2009 and provides several enhancements to the language, but due to the lack of support in some major browsers, it was not considered for the development of the framework and therefore this report only deals with ECMAScript 3.

2.1.1 Dynamicity

JavaScript, as a scripting langauge, is dynamic, which means that types are determined at runtime. Moreover, JavaScript is also a weakly typed language: variables may change their type during runtime and implicit coercion may occur [1, chapter 9]. Objects can be changed after their creation: properties can be added, modified and deleted. This offers great flexibility and is especially helpful for writing frameworks (like the one proposed in this report) that want to improve the object model of JavaScript.

2.1.2 Objects and Inhertiance

In JavaScript, objects can be created with the *object literal* or via a constructor and the `new` keyword, as can be seen in code listing 1:

```
var a = { foo: 'a' }
var B = function () {
  this.bar = 'b'
}
var b = new B() // b = { bar: 'b' }
```

Code listing 1: Objects in JavaScript

There are no classes in JavaScript and inheritance is achieved through the use of prototypes. Every object has a hidden `prototype` property containing the so-called *prototype chain* (a chain of objects the object "inherits" from). When a property of an object is accessed which is not defined on the object, it looks up its *prototype chain* to check if one of the inherited objects define the property. All objects in JavaScript implicitly inherit from the `Object` object, that is, the hidden `prototype` property is set to `Object`. This can be changed to any other object by setting the `prototype` property of a function, as illustrated in code listing 2.

```
var a = { foo: 'a' }
var B = function () {
  this.bar = 'b'
}
B.prototype = a // now every object created from B will inherit from a
var b = new B()
alert(b.foo) // alerts 'a'
```

Code listing 2: Inheritance in JavaScript

It is important to take note of the fact that every object created with new B() will now share the same prototype (object a).

2.1.3 Execution Context and Function Invocation

In JavaScript, there are three execution contexts: global code, eval code and function code [1, chapter 10]. Before any execution context is entered, a unique global object is created (containing built-in objects and host defined properties). When control enters an execution context, a this variable is assigned and its value is determined. In global code, this always refers to the global object, in eval code this always refers to the calling context and in function code it is either the caller object or, if that is not an object, the global object. As JavaScript allows to invoke functions on any object (therefore, functions are not bound to an object), the this variable inside function code will refer to the object on which the function is invoked. To invoke functions on a specific object, JavaScript provides two utility methods on Object, call() and apply(). Both take the object which this in the function code will reference as their first argument. Additionally, call() takes an arbitrary number of arguments, whereas apply() expects only one more argument: an array which will be applied to the function as arguments. The following code listing 3 illustrates how the value of this depends on the caller.

```
var a = {
  name: 'Jane Doe',
  getName: function () {
    return this.name
  }
}
var b = {
  name: 'John Doe'
}
alert(a.getName()) // will alert 'Jane Doe', could be written as a.getName.call(a)
alert(a.getName.call(b)) // will alert 'John Doe'
```

Code listing 3: Function invocation

2.1.4 Closures

Closures are functions that remember the context in which they were created. As seen in the previous section, every function code creates its own execution context. Inside a function, variables in the outer scope can be accessed, even when the function is used in a different context. Code listing 4 shows an example of this.

```
var outerFunction = function () {
  var foo = 'foo'
  return function () {
    return foo
  }
}

var innerFunction = outerFunction()
alert(innerFunction()) // will alert 'foo'
```

Code listing 4: Closure

However, there is no possibility to access the variable foo directly from the global context.

2.2 Server-side JavaScript

JavaScript has been mainly used as a client-side scripting language with a web browser as the host environment. However, the language is not limited to a web browser and can be used on the server-side as well. Since 2008 / 2009, there has been a lot of movement in this area which has been accompanied by standardisation efforts like CommonJS [**7**]. CommonJS is motivated by the lack of a standard library in the official JavaScript specification "that is useful for building a broader range of applications. [It tries to achieve this] by defining APIs that handle many common application needs, ultimately providing a standard library as rich as those of Python, Ruby and Java" [3]. Programs written using the CommonJS API are supposed to run in different host environments. One of them gaining a lot of interest lately is NodeJS[5]. It is a server-side platform built on Google's V8 engine[6] and offers asynchronous I/O, event loop concurrency and support for CommonJS's module system [**8**]. Such a module is shown in the following code listing 5.

```
// file: foo.js
var secret = 1
exports.bar = function () {
  return secret
}
```

Code listing 5: CommonJS module

With NodeJS, CommonJS modules can be used inside other files as shown in code listing 6.

```
var foo = require('./foo.js')
console.log(foo.bar()) // will output '1'
console.log(foo.secret) // will output 'undefined'
```

Code listing 6: CommonJS module usage

The to-be-developed framework will use NodeJS as its server-side platform due to NodeJS's popularity and ease of use, but should run with minor or no modifications on other server-side environments (e.g. narwhal[7]).

2.3 Dependency Injection

Dependency Injection (coined by Martin Fowler [**9**]) is a pattern to achieve *Inversion of Control* in software. The goal of *Inversion of Control* is to move control away from the objects themselves to a container or framework "around" them. Dependency Injection strives to handle the dependencies of objects from the outside. Therefore, objects do not build their dependencies but rather get those dependencies injected by the enclosing framework [5]. The following code listing 7 is a simple example of Dependency Injection without a container.

[5] http://www.nodejs.org
[6] http://code.google.com/p/v8/
[7] http://narwhaljs.org/

```
var House = function (door) {
  // without DI, we would create an instance of Door in here:
  // var door = new Door()

  this.openDoor = function () {
    door.open()
  }
}
var Door = function () {
  this.open = function () {
    alert('Door opened')
  }
}
var door = new Door()
var house = new House(door)
```

Code listing 7: Dependency Injection

Within a container, the creation of an instance of House would automatically create and inject an instance of Door so that the developer does not have to create all the dependencies manually. The wiring is usually done in a configuration file.

The Dependency Injection pattern enables objects to be more loosely coupled (implementation can be easily exchanged), which is especially important in larger applications. Furthermore, it eases testing of the objects that get their dependencies injected [**10**]. Because they do not build the dependencies inside their code, in testing those dependencies can be mocked and set on the object from the outside. However, this is of less importance in JavaScript because properties usually can be accessed from the outside the object and modified anytime. Therefore, it has been questioned whether Dependency Injection is needed in dynamic languages in general and in JavaScript in particular. These objections miss the fact that Dependency Injection offers the possibility to exchange the implementation (or simply the object in languages without interfaces) by changing the configuration (which specifies all the dependencies) without modifying the source code itself. This is crucial because it cannot be assumed that all source code is owned or that it is desirable to modify source code provided by a third party (this would make updates more complicated). Secondly, and this is specific to JavaScript, Dependency Injection can help with the "composition" of the software. JavaScript does not provide a module system or any other way to import code from other source files. Because of this, many frameworks provide additional tools to combine (all) source files into one file which can be executed. Having a configuration file which states the dependencies between objects helps such tools to decide exactly which files are needed when one object is requested. The details of this technique are explained in chapter 4 of this report, but it should be noted here that Dependency Injection enables smart loading of resources, saving bandwidth which is very important in the browser environment in which JavaScript is currently primarily used. Therefore, the principles of the Dependency Injection pattern play a major role in the proposed framework.

3 Analysis

As stated in the introduction, the goal of the framework is to provide missing language features of JavaScript and to help programmers write systems that are composed of many different components. This chapter will analyse which features are needed to achieve this, beginning with a review of other scripting languages. Languages that need to be compiled (like Java) are not considered in this review as they are not directly competing with JavaScript[8]. The languages chosen for the analysis are PHP and Python, because they are the two most popular languages in the script category[9] according to the March 2011 results of the Language Popularity Index tool [**11**]. Besides being popular, PHP is also a good choice due to its use in many big projects on the web (e.g. facebook.com [**12**] or digg.com [**13**]). Python also is especially interesting in the context of this report due to its similarity to JavaScript (e.g. with respect to encapsulation, interfaces, modules etc.).

Furthermore, we must investigate what existing JavaScript frameworks provide to foster development of large applications. We will consider two frameworks which are chosen as representatives of their specific approach. For each of them, we will list the main features and explain why they are not sufficient to achieve the goal outlined in the introduction.

Finally, the requirements for the design, which is detailed in the next chapter, will be drawn from the analysis we have carried out.

3.1 PHP and Python

There are many ways to compare programming languages with each other – e.g. features, syntax, built-in types, libraries or tool support just to name a few. In this section, we'll focus on the features, because this is the part of the language we're aiming to improve. As noted in chapter 2, JavaScript's dynamic nature and flexibility allows us to easily introduce new features by writing a framework, whereas other characteristics of the language cannot be changed by the user, like the syntax[10]. The main features we consider are those concerning the organization of code and the object model of the language in particular. These greatly determine how suited a language is to develop large applications: The bigger the code base grows, the more important it is to organize the code and to maximize reusability and minimize code duplication. In the following section, we will compare how PHP (in version 5.3) and Python (in version 3.2) approach these aspects in contrast to JavaScript.

[8] To a certain extent, JavaScript can even be considered to be a totally unique language because it is the only scripting language running natively in every major browser.

[9] Depending on the metrics used, results vary. Other language popularity measurements such as LangPop.com or the TIOBE Software Index have slightly different results [INSERT CITATION MANUALLY].

[10] That being said, JavaScript's syntax has optional features, e.g. the semicolon to mark the end of each line is not required, see [CITE ECMA STANDARD HERE, chapter 7.9].

3.1.1 Modules and Namespaces

PHP offers namespaces in order to group related classes, functions and constants [**14**]. This logical grouping prevents class name clashes and does not require a specific physical structure of the files involved. PHP does not provide a module system, but the language is able to include other (PHP) files into the program [**15**] and to autoload classes at runtime [**16**]. Python however offers a module system [**17**]. The filename of the module equals the name of the module, by which it can be imported by other scripts. Modules and classes form natural namespaces. JavaScript supports no modules like Python and no namespaces like PHP and therefore no importing of source code into another script. There is however the CommonJS standardization effort mentioned in chapter 2 which defines a module functionality for the server side [**18**], e.g. implemented by NodeJS. As JavaScript is already becoming more like Python in this area, our framework will organize the source code in the CommonJS module standard and bring this functionality to the client side as well.

3.1.2 Classes

The object model of both PHP and Python is centred around classes. A class represents a type in the program and consists of properties, which can be either variables or functions. Classes are instantiated with the new keyword (which calls a constructor function of the class). The created object is of the type of the class and therefore also consists of the variables and functions defined in the class [**19**]. Furthermore, classes might define properties – called *static properties* in PHP and *attribute references* in Python – belonging to the class itself (instead of to the object). These properties are shared between all objects of the same type and can be accessed via the class name. JavaScript however does not have classes at all. Instead, it is part of the prototype family of object-oriented languages that use cloning to facilitate creation of new objects. It is possible though to introduce a type by defining a function. This function (which itself is an object) can create a new object from itself when it is called with the new keyword [**20**]. Hence, the function acts as both the constructor and the type of the object.

Many JavaScript frameworks try to make JavaScript more class-like by introducing class constructs etc (e.g. *Joose*[11], which is described in the next section). It is being questioned though if the class approach offers significant advantages over the JavaScript object model and especially if trying to change the language to be more like some other language results in a better solution [**21**] [**22**]. Our framework will follow the object-only approach of JavaScript, as the great flexibility that comes with this choice can be very helpful for developers.

3.1.3 Access Modifiers

PHP offers access modifiers for the properties of a class. Properties can be public (accessible to everyone), private (accessible only within the class) or protected (accessible within the class and also from any subclass) [**23**]. This approach ensures encapsulation: Developers are able to control who can change properties so that the state of an object cannot be corrupted from the outside. Python however, like JavaScript, does not offer access modifiers

[11] http://code.google.com/p/joose-js/

at all. Instead, Python developers follow a convention to mark variables as private by the use on an underscore prefix [**24**]. To be precise, the underscore signals that developers should not use them from the outside, but objects inside (in Python, that is in the module) can access them like they were public. The reason for not having access modifiers is that a convention is believed to provide enough guidance for a developer which members to use and which to avoid, but still leaves the opportunity to access them if absolutely needed. While this might seem rather fragile for programmers coming from very strict languages, it seems to work well in practise. Furthermore, it is helpful in unit testing, because members can easily be mocked. For the to-be-developed framework, we decided to stay with the current approach to not have access modifiers but adopt the underscore convention of Python[12].

3.1.4 Interfaces

While PHP provides the possibility to define interfaces for classes [**25**], Python does not. An interface specifies the public methods of a class (the API of that class). Interfaces also act as types, so objects of different classes might all have same type if their classes all implement the same interface. The advantage of interfaces is that they enable developers to program against interfaces and not specific classes. Then, the implementation of the interface (the specific class) can be changed without breaking consumer code. JavaScript does not offer interfaces, which is also due to its dynamic nature: A certain interface cannot be guaranteed as the object might have been altered between its creation and its usage. Objects always have to be asked what they can and cannot do (this behaviour is called duck-typing). That being said, it is till desirable to program against some abstract concept of something rather than its actual implementation.

3.1.5 Inheritance

Both PHP and Python allow inheritance of classes. The subclass is a subtype of the superclass and therefore may be used instead of an object of the superclass (this is called *subtype polymorphism*) [**26**]. The subclass has the possibility to override methods of the superclass. PHP allows only single inheritance [**27**], whereas Python allows multiple inheritance as well [**28**]. JavaScript however features prototypal inheritance, as explained in chapter 2. In PHP, overriding methods can access the method defined in the superclass via the *super* keyword. Python also enables developers to access inherited methods that have been overridden by providing a *super* function (delegating method calls to a parent or sibling class [**29**]). This additional level of control is needed for multiple inheritance because it might not be unambiguous which parents' method to use. The following figure 1 demonstrates this problem (known as the *diamond problem*):

[12] However, it is possible in JavaScript to have private variables via closures, see [INSERT CROCKFORD CITATION HERE]. Also, protected variables can be emulated by passing private variables from inherited types to a function. This technique is described in a German article on the technology magazine heise.de [INSERT HEISE CITATION HERE].

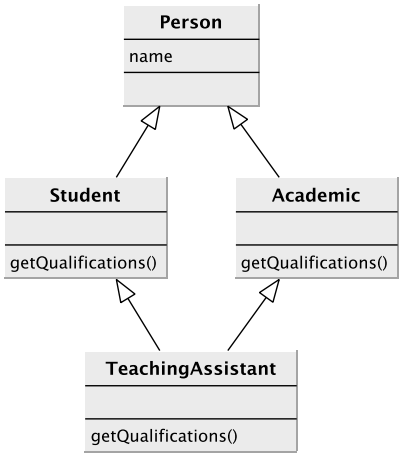

Figure 1: Diamond problem

Python allows to specifically call the getQualifications() of Student or of Teacher. By means of this *super* function it is also possible to re-use methods further up in the hierarchy, not only those in the immediate parent(s), e.g. if Person would define a getQualifications() method as well. As multiple inheritance increases the reusability of code, our framework tries to implement this feature. A similar approach to Python is taken, adding a function to access overridden methods (which is not possible in standard prototypal inheritance). We will also adopt Python's default strategy to inherit properties: The parent classes are searched in the order they are given, "depth-first, left-to-right, not searching twice in the same class where there is an overlap in the hierarchy" [INSERT PYTHON CLASS CITATION HERE].

As a result of the language comparison, we saw that support for modularisation of code, multiple inheritance and some form of interfaces would be beneficial for developers of large applications. Those three requirements will be included in the requirements list at the end of this chapter.

3.2 Yahoo! User Interface Library 3 (YUI3)

Although named "User Interface Library", this library is very interesting for JavaScript developers trying to write large applications as it offers many features related to the object model and the organization of code. YUI3 was designed for the client-side and is in use on many websites with a lot of JavaScript code, e.g. the Yahoo! Homepage and LinkedIn [**30**]. The core of the is framework the *YUI global object*, which mainly offers loading and dependency functionality for other components. When a component is used, it and all its dependencies will be fetched by the *YUI global object* in a single HTTP request before the dependent code is executed [**31**]. YUI3 also helps composing objects by providing several helper functions like extend, augment and mix. These allow the developer to build the notion of class hierarchies, and to dynamically add functionality to objects after their creation. To provide this, YUI3 makes extensive use of the prototypal nature of JavaScript. In the following code listing 8, it is illustrated how to create a two and a three dimensional point (Point and Point3D) with YUI3, using type hierarchy:

```
YUI().use('oop', function(Y) {
  function Point(x, y) {
    this.x = x
    this.y = y
  }

  Point.prototype.clear = function () {
    this.x = 0
    this.y = 0
  }

  function Point3D(x, y, z) {
    Point3D.superclass.constructor.call(this, x, y) // Chain the constructors
    this.z = z
  }

  Y.extend(Point3D, Point) // Point3D is a Point

  Point3D.prototype.clear = function () {
    this.x = 0
    this.y = 0
    this.z = 0
  }

  var point = new Point(1, 1)
  var point3d = new Point3D(1, 1, 1)
  point3d isinstanceof Point // true
})
```

Code listing 8: Point and Point3D in YUI3 (with type hierarchy)

Alternatively, one could achieve the same functionality without type hierarchy by simply augmenting Point, as illustrated in code listing 9.

```
YUI().use('oop', function(Y) {
  function Point(x, y) { /* omitted */ }
  Point.prototype.clear = function () { /* omitted */ }

  function Point3D() { /* omitted */ }
  Point3D.prototype.clear = function () { /* omitted */ }
  Y.augment(Point3D, Point)

  var point = new Point(1, 1)
  var point3d = new Point3D(1, 1, 1)
  point3d isinstanceof Point // false
})
```

Code listing 9: Point and Point3D in YUI3 (without type hierarchy)

When we compare YUI3 with the goal set for our framework in the introduction, we notice some shortcomings however. Firstly, it does not work out of the box on the server. By default, the loading functionality only works in an asynchronous environment, loading scripts via HTTP. It is possible to replace the loader though and have the framework run on the server as well [**32**]. Furthermore, dependencies cannot be configured from the outside, which makes it hard to exchange their implementation effortlessly. Also, the dependency system is not a real dependency injection system but simply tries to bring modules (and import thereof) to the client. Finally, while the bundling of components together to be served in a single HTTP request is a desirable feature for performance reasons, it only works if the files are being served from Yahoo!'s content delivery network (CDN) [**33**]. Ideally, our framework would not rely on extra infrastructure.

3.3 Joose

Joose is a meta object system for JavaScript. It uses JavaScript's object literals do describe constructs like classes and to add features like inheritance, mixins and method modifiers [**34**]. Joose provides these features by helper functions that sit in the global namespace like

Class, Module, Role or Prototype. Into each of these helpers, one passes in an object literal that will be turned into the real representation (e.g. in the case of Class, it will assemble a function in the global namespace which can act as the constructor for an object). A simple example that defines a two dimensional point (Point) and a three dimensional point (Point3D) inheriting from Point is shown in code listing 10 below.

```
Class('Point', {
    has: {
        x: {is: 'ro'}, // property is readonly, generates getX method
        y: {is: 'rw'}, // property is read/writeabe, generates getY and setY methods
    },
    methods: {
        clear: function () {
            this.x = 0
            this.setY(0)
        }
    }
})

Class('Point3D', {
    isa: Point, // Point3D subclasses Point
    has: {
        z: {}
    },
    after: {
        clear: function () {
            this.z = 0
        }
    }
})

var point = new Point()
var point3d = new Point3D()
```

Code listing 10: Point and Point3D in Joose

The object with the key has defines the properties of a class, isa specifies the supertype. Methods can be laid out in methods, and later modified e.g. with the after keyword. Joose transfers these meta descriptions and creates a function Point and Point3D in the global namespace.

Generally, it could be said that Joose's approach is to make JavaScript more like a class-based language. Other frameworks with similar feature sets are for example Qooxdoo[13] or the Dojo toolkit[14]. Out of the box, Joose cannot be used on the server side and does not provide any form of dependency management, but there are CommonJS modules available to add support for both[15]. These modules need to be imported into every script that wants to use these features. In our tests, these did not work though, as the dependencies between modules were not resolved. This seemed to be a configuration problem, but the documentation was sparse and the error messages did not point to the problem. Even if the dependency management would work flawlessly, the approach taken to load dependencies asynchronously is quite simplistic and inefficient compared to YUI3's approach, because the

[13] http://qooxdoo.org/

[14] http://dojotoolkit.org/

[15] There is a module for NodeJS:
http://openjsan.org/doc/s/sa/samuraijack/Task/Joose/NodeJS/0.07/lib/Task/Joose/NodeJS.html and one to add dependency management between modules:
http://openjsan.org/doc/s/sa/samuraijack/JooseX/Namespace/Depended/0.10/lib/JooseX/Namespace/Depended.html.

extension needs as many HTTP requests as there are dependencies[16]. When judging Joose's feature set, it lacks dependency injection and out-of-the-box server side compatibility.

From our evaluation of JavaScript based frameworks it is clear that there is no framework which is built from the start for both client- and server side currently available; and no framework that truly uses dependency injection. Additionally, every of the available options misses some of the goals we intend to achieve. That being said, we gained valuable insight into which features might be useful for our framework, which we'll present in the next section.

3.4 Requirements

To summarise the analysis carried out so far, the goals of our framework are as follows:

1. **Testability**
   Although not mentioned in the analysis above, testability is a must-have for large applications: As the code base grows, it becomes more and more important to run repeatable tests to ensure the functionality works as expected and that changes made to one part of the application do not negatively affect other parts of the application. Furthermore, in the context of dynamic languages, tests can help to spot errors that would otherwise be detected by a compiler **[35]**.

2. **Modularity**
   The source code must be organized in small chunks (to increase maintainability) which then can be composed by the framework as needed. The chosen format is the CommonJS module system, which can be used on the server side without modification.

3. **Multiple Inheritance**
   To maximize reusability, objects should be able to inherit from more than one parent. The solution must provide a way to deal with the *diamond problem* and offer a convenient way to access overridden methods.

4. **Dependency Injection**
   Objects that depend on other objects should not resolve those dependencies on their own, but rely on the framework to inject them. As a side-effect, Dependency Injection should provide a way to exchange the implementation which is injected. This exchange should be possible without touching the consuming code so that the framework gains some of the features interfaces provide and enables developers to program against abstractions, not implementations.

5. **Server-side and client-side compatibility**
   The code should work with no modification by the developer in both environments. This will affect the design of our module system: On the client, code needs to be

[16] However, there has been a discussion in May 2009 to improve this situation, see http://www.sencha.com/forum/showthread.php?69161-Grouped-dependencies-loading-suitable-for-production.

loaded asynchronously and dependent code must not be executed before all dependencies are resolved.

6. **Minimum overhead in production context**
   On the client, dependencies should be served in a single request, and no special server should be needed. Furthermore, if the framework is used on the server, no code concerning only client side issues should be loaded (and vice versa).

With those requirements in place, the next chapter will develop the design for the framework.

4 Design

The design of the framework will adhere to the requirements specified in the previous chapter. Therefore, this chapter will not only give a broad overview of the design, but will also address how each requirement is met by the concept.

4.1 Architecture

Code written based upon the framework will be organised in so-called *bundles*. A *bundle* provides a coherent piece of functionality, which is mainly provided by JavaScript modules, which are accessible by their *identifiers* (their name, determined by the physical structure). Furthermore, a *bundle* may contain tests and provide static assets like HTML files, images, etc.

Next to the bundles directory, the framework will have a `web` directory, which should be the only directory accessible from the client side (more on how the loading works will be explained in the next chapter) and a global configuration file to control how the *bundles* and *identifiers* work together. The basic file structure therefore looks like the following:

```
build/
bundles/
web/
configuration.js
```

4.2 Module Loading

One requirement is that the framework should work equally well on both server and client side. This affects the way we organize our code. As already stated, we would like to write normal CommonJS modules that look like those explained in chapter 2. However we cannot directly use those modules on the client, because they are designed for an synchronous environment. If a module is imported in a synchronous environment, the execution stops until the code has loaded. On an asynchronous host however, execution does not stop. Therefore, we need to wrap the dependent code in a function and call this function once the modules have finished loading (usually via HTTP). This difference in coding results in the incompatibility of server and client side modules. Our framework will have to unify this in order to use the same modules both sides without code modification, but first we need to determine how we are going to load scripts in an asynchronous environment.

The website of the simple module loader *RequireJS*[17] gives a good overview of which methods are available to load scripts on the client [**36**]. The seemingly best approach is to create new `script` tags on demand and add them to the HTML `head` tag, which will cause the script to load asynchronously. Once it is loaded, we can call our dependent code. The easiest way to detect load completion would be to listen for the *onload* event of the `script` tag, but unfortunately this event is handled very differently across browsers. An alternative

[17] http://requirejs.org/

method working in all browsers is to call a function at the end of every script to be loaded [**37**]. When this function is executed, we know that loading is done. The main disadvantage of this method is that you have to control the content of the JavaScript file you want to load. In our framework we do have that, so we will use this approach to script loading.

With that in place, we have to investigate how we can use the exact same code for both server and client side, as we saw that we need to wrap dependent code on the client and need to call a function after each loading (which we do not need on the server). Also, on the client we would like to combine all files into one HTTP response (similar to YUI3). The architecture of our framework to achieve this is shown in figure 2.

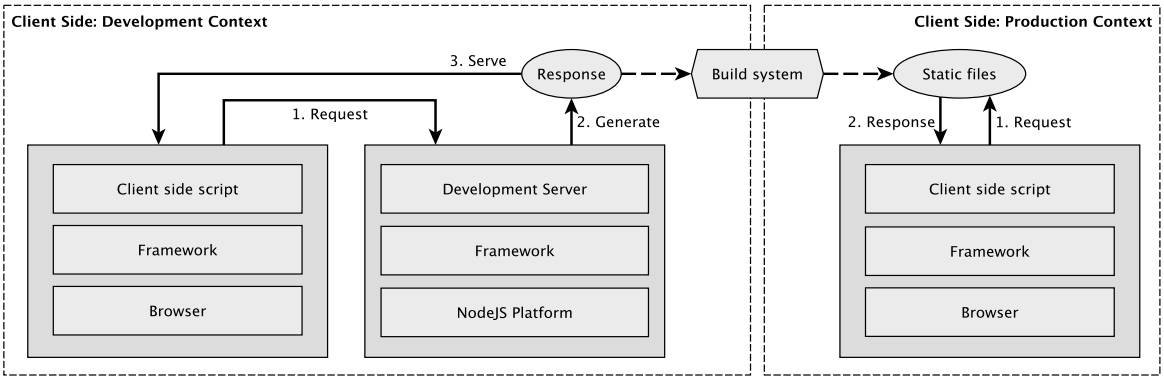

Figure 2: Loading Architecture

The architecture is based on the idea that there are two contexts in which the frameworks may run: development and production. In development context (left hand side in figure 2), changes to the code happen very often and we want the framework to automatically adapt to them. In production (right hand side in figure 2) however, performance is the highest priority and changes do not happen as frequently. Therefore, we will have a development server (using NodeJS) running in development context, dynamically creating the responses. When a client requests a module (step 1), the server loads the required modules and finds their dependencies by executing the code (this is possible because the server itself is written in JavaScript). When all dependencies for the request are determined, the server will dynamically generate a single HTTP response by reading the module files, wrapping the code in functions and adding a function call at the end to signal that loading is done (step 2). This response is then send back to the client (step 3), but also stored for later use by the build system. The build system can generate static JavaScript files for each response (reusing the functionality of the development server) recorded in the development context. These static files are then used in production context, in which we do not want to rely on an additional NodeJS server. Therefore, if a client requests a module in production context (step 1), that request directly accesses one of the static files (step 2).

This approach also ensures a great level of security: Some parts of the application might be only suitable for use on the server side (e.g. database access with passwords) and should never be accessible though the web server. With our approach, only those modules requested from the client by the developer in development context will be made available to the web server in production context.

4.3 Module Content

In the previous section, we defined how modules will be written and how they are going to be loaded, but we did not define what content they will have. In our framework, each module represents an object, which is referred to by its *identifier*. It is possible to re-configure which module will be used for a particular *identifier* in a global configuration file. This enables developers to exchange the implementation easily, but it does not guarantee that the substituting object is actually a valid replacement (like an interface would). Unfortunately, JavaScript dynamic nature makes it impossible to achieve this.
The module content is made out of two parts: an object definition and an object configuration. See code listing 11 for an example.

```
exports.definition =
{ colour: ''
, init: function (colour) {
    this.colour = colour
  }
}

exports.configuration =
{ inherits: ['org.motionjs.demo/model/vehicle']
, dependencies: { 'org.motionjs.demo/model/engine': '_engine' }
}
```

Code listing 11: Module content describing a car

In the definition, the properties – variables and functions – of the object are defined. If the definition contains an init method, the object is instantiable later on through the framework, if not, the definition is like an abstract class. The configuration is responsible for specifying the objects *dependencies*, the *identifiers* it *inherits* and variable *requirements* that need to be met, which will be explained in more detail in the following section.

4.4 Object Configuration

4.4.1 Dependencies

As noted earlier, we want to use dependency injection in our framework. In the *dependencies* part of the configuration, developers can note which *dependency* the object described in the definition has (and which variable should reference this dependency). The *dependency* will be loaded by the framework and injected into the specified variable immediately after the object is created.

4.4.2 Inherits

As noted in chapter 3, we want to support multiple inheritance and use a similar approach to Python. When multiple *identifiers* are inherited, the properties of the *identifiers* are searched depth-first, from left to right. Therefore, the most left *identifiers* possible override properties defined in *identifiers* to their right. While the idea is very similar to Python, the implementation is completely different. Remembering JavaScript's prototypal nature, we could merge the functions of the inherited identifiers into one object and set that as the prototype of object to be created. The variables of the inherited objects would need to be collected and added to the created object as well. Figure 3 shows this setup.

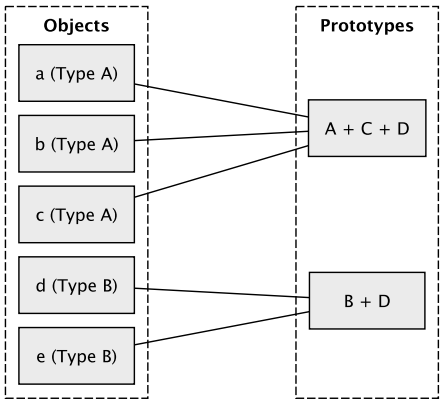

Figure 3: Object Inheritance, first approach

Unfortunately, this does not allow us to access functions that have been overridden. As this is crucial for the provision of multiple inheritance though, we need to further refine the concept. In chapter 2 we saw that JavaScript functions are not bound to objects but can be invoked on any object. Therefore, we can extract all functions from the *object definition* in a module and store them in an *ability object*. For each *identifier*, there is exactly one such object in the framework. The function calls any object in the framework then need to be delegated to that *ability object*. We accomplish this by adding function wrappers to the object and invoking the *ability* function on our current object. This approach allows us to access (by means of a helper function) functions of specific *abilities* so that using overridden methods is possible. One final improvement is to share the wrapper functions in the prototype for each object, as they are the same for all objects of one type. This design is illustrated in figure 4.

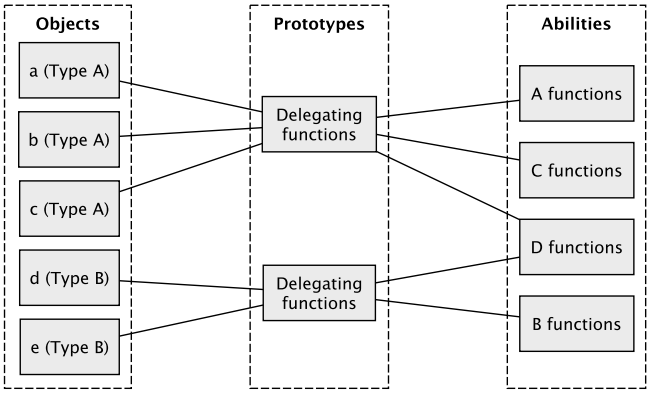

Figure 4: Object inheritance, second approach

By using this approach we reduce memory usage by sharing common functions, and introduce a very powerful multiple inheritance mechanism.

4.4.3 Requirements

An object configuration can specify required properties. For example, the *identifier* might need a certain property to exist for its functions to work. When an object is created, it is checked whether all the *requirements* of the *identifier* of the created object are met. This is helpful for example, if an identifier wants to delegate the creation of a dependency to the inheriting object.

4.5 Object Creation

The object concept described above forces us to create objects only through the framework. The basic mechanism of object creation is illustrated in figure 5.

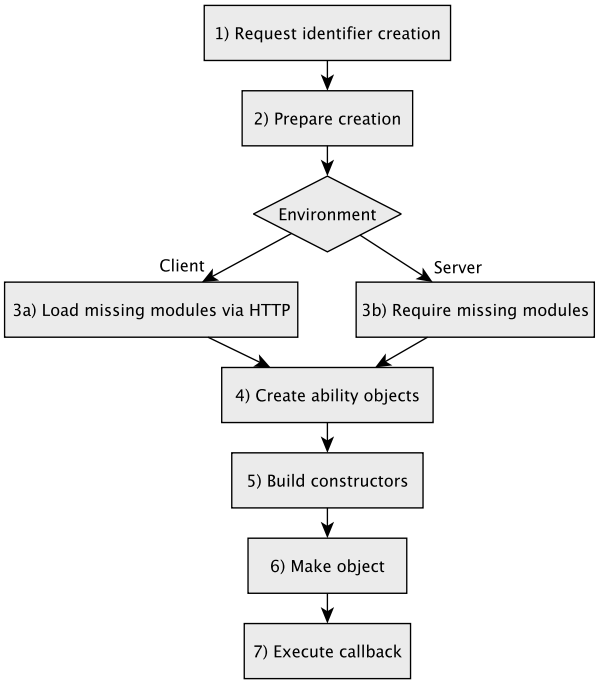

Figure 5: Object creation procedure

When we want to create an object, we ask the framework to give us an object of a certain *identifier* (step 1). The framework then prepares the creation of the object (step 2) by determining which dependencies the identifier has. The missing modules are added to the framework (step 3a or step 3b depending on the environment) before we extract the functions of the inherited identifiers according to the design detailed in section 4.4.2 (step 4). Afterwards, the constructors are built (step 5), which are then used to create the object (step 6). Finally, the object is given back to consumer code by passing it to the callback (step 7).

4.6 Object Management

The framework is also responsible for managing objects. Sometimes, it is desirable to have only one instance of an object in the entire program. One solution to achieve this is to use the Singleton pattern, which hides the constructor and provides a static method to create an instance [**38**]. This method returns always the same instance on subsequent calls. This pattern is often criticised because it introduces global state and impedes testing [**39**] [**40**]. Furthermore, it would not be possible to implement this exact pattern, as JavaScript does not have static classes. Therefore, our framework provides an alternative way, which does not require these objects to be in the globally accessible, to achieve object sharing. If an *identifier* is configured as shared, our framework will always return the same instance when it is asked to create a new instance of the *identifier*.

5 Implementation

The implementation is based on the design described in the previous chapter, but the design is, to some extent, also a result of a partial implementation. In order to get the design right, initially a quick prototype was developed to refine the basic concept that derived from the requirements. This prototype then led to the detailed design described earlier. The following chapter will focus on the development of the framework as it was carried out from that point on.

5.1 Architecture

The framework presents itself as a single object to the user, called motionjs. This object has a very simple public interface, consisting only of two methods: create and clone. clone is creates an exact copy of an object, whereas create is responsible for creating new objects (how this works internally is covered in the next sections). Objects can be created on the server and on the client side only in asynchronous style, hence we pass a callback to the create function, and the object is passed to the callback once it is created. While we could return the object from the create function on the server (where modules can be loaded synchronously), this would not be practical, as two different API's would prevent us from using the same script in both environments.

Although the API is the same in both environments, the inner working is very different. This would result in an unnecessary overhead if we had the code for both environments in the motionjs object. To minimize this overhead as much as possible, the framework instead extracts all server specific code into its own file (called adapter/server.js) and all client side code into another file (adapter/client.js). When the framework is build on each platform, the respective adapter is loaded and merged into the core object (motionjs), which is possible through JavaScript's dynamic nature.

5.2 Object creation

Before an object can be created, the framework needs to make sure that it is ready to create the object. The framework is prepared to create an object, if:

- All modules of its dependencies are loaded
- The ability objects are created
- A constructor (complete with the prototype) is available

In order to achieve this state, firstly, a list of *identifiers* that need to be loaded has to be created (see code listing 1 of the function _buildInheritanceChains). This list of *identifiers* (which is stored as an object called _inheritanceChains in the framework) can be consulted by the framework to determine which identifiers are inherited by a given *identifier*. To build _inheritanceChains, the function is called with the *identifier* for which to determine the inherited identifiers. A flag loadModules indicates whether the module of the *identifier*

should be loaded from the disk and stored as an object in the framework or not. This depends on the environment:

- On the client, we only want to build the *inheritance chains*, as the modules are added via a loaded script. This script consists of all modules required, which in turn are determined in development context by the framework in the server environment. The next section on module loading and the development server will explain in detail how this works.
- On the server, we load the modules here (2), as the function _buildInheritanceChains will have to visit each *identifier* required, and therefore it is ensured that all source code needed to create an object from any *identifier* in _inheritanceChains is present afterwards.

The *identifier* given to _buildInheritanceChains is only processed if it has not been entered into _inheritanceChains previously (1). We also store the *identifier* into its own *inheritance chain*, as this will simplify the steps needed to create an instance of an identifier later on (3). We then iterate over the inherited *identifiers* specified in the *object configuration*, retrieve the real *identifier* for each and call _buildInheritanceChains again. This will recursively build the _inheritanceChains of all inherited *identifiers*. The newly build chain will get concatenated to the current *inheritance chain* (4). To be able to create the dependencies, we need to determine their inherited *identifiers* as well, which happens in (5).

```
, _buildInheritanceChains: function (identifier, loadModules) {
    if (typeof this._inheritanceChains[identifier] == 'undefined') {  1

      // identifier itself
      if (loadModules) {
        this._requireModule(identifier)  2
      }
      this._inheritanceChains[identifier] = [identifier]  3

      // inherited identifiers
      if (this._modules[identifier].configuration.inherits) {
        for (var i = 0; i < this._modules[identifier].configuration.inherits.length; i++) {
          var inheritedIdentifier = this._getRealIdentifier(this._modules[identifier].configuration.inherits[i])  4
          this._buildInheritanceChains(inheritedIdentifier, loadModules)
          this._inheritanceChains[identifier] =
            this._inheritanceChains[identifier].concat(this._inheritanceChains[inheritedIdentifier])
        }
      }

      // dependencies
      if (this._modules[identifier].configuration.dependencies) {
        for (var dependencyIdentifier in this._modules[identifier].configuration.dependencies) {  5
          this._buildInheritanceChains(this._getRealIdentifier(dependencyIdentifier), loadModules)
        }
      }
    }
  }
```

Code listing 12: _buildInheritanceChains

Secondly, the *ability objects* (as detailed in chapter 4) need to be created, which is done in _buildAbilitiyObjects. This function iterates over the previously created *inheritance chain* for the *identifier*. In each iteration (for each inherited *identifier*), a new *ability object* is created if it does not exist yet, containing just the functions defined in the *object definition*.

Thirdly, the constructor needs to be built. This is done in _buildConstructor, which first part is shown in code listing 2:

```
// constructor (contains all properties)
var source = {}
for (var i = this._inheritanceChains[identifier].length - 1; i >= 0 ; i--) {
  var identifierModule = this._modules[this._inheritanceChains[identifier][i]]
  // set properties specified in the definition
  for (var key in identifierModule.definition) {
    var objectProperty = identifierModule.definition[key]
    // only set properties that are not functions (functions are wrapped in prototype and
    // delegate to ability functions)
    if (typeof objectProperty != 'function') {
      source[key] = objectProperty
    }
  }
}
this._constructors[identifier] = function () {
  var temp = self.clone(source)
  for (var key in temp) {
    this[key] = temp[key]
  }
}
```

(1) (2) (3)

Code listing 13: _buildConstructor, part 1

The constructor needs to set all properties (except the dependencies which are injected later on) defined in the *identifier* to be created and its inherited *identifiers*. Therefore, in (1), we iterate over all *object definitions* in the *inheritance chain* and collect all properties which are not functions (they have already been added to the *ability objects* in the previous step) into a source object (2). This object is cloned on each instantiation, as can be seen in (3).

After the constructor is defined, its prototype needs to be attached in the second part of _buildConstructor (see code listing 3).

```
// prototype (contains all methods)
var self = this
this._constructors[identifier].prototype =
{ _type: identifier
, isInstanceOf: function (typeName) {
    for (var i = 0; i < self._inheritanceChains[this._type].length; i++) {
      if (self._inheritanceChains[this._type][i] == typeName) return true
    }
    return false
  }
, _uber: function () {
    var methodArguments = Array.prototype.slice.call(arguments)
    if (methodArguments.length >= 2) {
      var abilityIdentifier = self._getRealIdentifier(methodArguments.shift())
      var methodName = methodArguments.shift()
      // ensure typeName is one of the abilities
      if (this.isInstanceOf(abilityIdentifier)) {
        return self._abilityObjects[abilityIdentifier][methodName].apply(this, methodArguments)
      } else {
        throw new Error('Cannot call ' + methodName + ' on ' + abilityIdentifier + ' because ' + this._type
          + ' is not a subtype of ' + abilityIdentifier)
      }
    } else {
      throw new Error('You need to provide at least a method name and an identifier.')
    }
  }
}
this._applyAbilityFunctionWrappers(identifier, this._constructors[identifier].prototype)
```

(1) (2) (3) (4) (5) (6)

Code listing 14: _buildConstructor, part 2

The prototype is used to store the function wrappers (applied in (6)) that delegate to the *ability objects*. Furthermore, the prototype provides helper properties and functions to deal with type information and inheritance:

- A _type property is added to easily retrieve the type (the *identifier*) of an object (2).

- As we do not store all the inherited objects in the prototype, the instanceof operator does not work anymore, so we provide an alternative isInstanceOf function in (3) . This function checks whether the given *identifier* is in the *inheritance chain* we created earlier. The *inheritance* chain is stored in the motionjs object though, and not in the created object. Therefore we make use of a closure (see chapter 2) to access this information from the isInstanceOf function. In (1) we store the reference to the motionjs object in the variable self. We then use this variable in the isInstanceOf function to access the _inheritanceChains property.
- In (3), we create the _uber function which provides access to overridden methods (see chapter 4). It is given an *identifier*, a method name and an arbitrary number of arguments. We can retrieve these from the arguments variable automatically provided by JavaScript which holds all arguments passed to a function. Before we can use it, we first need to convert it into an array, which is done in (4). Afterwards, we can either call the function on the *ability object* with the given arguments, or, if the given *identifier* has not been inherited, throw an error (5).

At this point, the framework is ready to create the object, so we call the _makeObject function (code listing 4).

```
, _makeObject: function (identifier, initArguments, buildContainer) {                    1
    // if object is already in buildContainer, return it right away (enables cyclic dependencies)
    if (typeof buildContainer[identifier] != 'undefined') {
      return buildContainer[identifier]                                                  2
    }

    // if the object is configured as shared and already in the framework, return it right away
    if (this._modules[identifier].configuration.shared && typeof this._sharedObjects[identifier] != 'undefined') {
      return this._sharedObjects[identifier]                                                                        3
    }
```

Code listing 15: _makeObject, part 1

The function has three parameters (1): The *identifier* to create, any arguments given to initialize the object and a buildContainer.

In code listing 5, we'll see that dependencies are created via recursive calls to _makeObject. If there were no safeguard, a cyclic dependency would result in endless loops and could never be built. To prevent this, we store each *identifier* build during one create call in the buildContainer (again we'll see this shortly in listing 5). If the identifier to make is already present, we return it instead of building a new object (and calling _makeObject again) (2).

Another case in which to return an object instead of making a new one is if the *identifier* is configured as shared (see chapter 4). If the object has already been created, we access the _sharedObjects property and return the shared object (3).

If the object has not been returned, we need to create a new object. This process is shown in code listing 5.

```
// create new object
var obj = buildContainer[identifier] = new this._constructors[identifier]() (1)
if (this._modules[identifier].configuration.shared) {
  this._sharedObjects[identifier] = obj (2)
}
// set dependencies of object
for (var i = 0; i < this._inheritanceChains[identifier].length; i++) { (3)
  var inheritedIdentifier = this._inheritanceChains[identifier][i]
  for (var dependencyIdentifier in this._modules[inheritedIdentifier].configuration.dependencies) {
    var dependencyObjectKey = this._modules[inheritedIdentifier].configuration.dependencies[dependencyIdentifier] (4)
    dependencyIdentifier = this._getRealIdentifier(dependencyIdentifier)
    obj[dependencyObjectKey] = this._makeObject(dependencyIdentifier, [], buildContainer) (5)
  }
}
```

Code listing 16: _makeObject, part 2

First, we create a new object by using the new operator on the constructor stored in _constructors (1). We store the object in the buildContainer to prevent endless loops, and, if appropriate, also in _sharedObjects (2). Then we set the dependencies, for which we have to iterate over all *identifiers* in the *inheritance chain* (3). For each *identifier*, we need to retrieve the property name for which to set the dependency (4), and then store the requested dependency object in there. We create this object by recursively calling _makeObject (5).

After the object has been created, in code listing 6, we ensure that all requirements are met (1). This is the case if all properties requested in the *identifier* itself or in any of the inherited *identifiers* are present. If no exception is thrown, we call the init function of the newly created object (3). We take into account that there might be defaultArguments defined in the *global configuration*, so we merge them into the initArguments given (as argument to _makeObject), giving precedence to the initArguments. (2). After we have called the init function, we can delete it as it should really be called only once (3).

```
// check if the constructor will set all properties that are required by the identifiers
this._checkRequirements(identifier, obj) (1)

// initialize object
if (typeof initArguments == 'undefined') {
  initArguments = []
}
if (typeof this._configuration.objects[identifier] != 'undefined'
  && typeof this._configuration.objects[identifier].defaultArguments != 'undefined'
  && this._configuration.objects[identifier].defaultArguments.length > initArguments.length) {
  var startIndex = initArguments.length
  var endIndex = this._configuration.objects[identifier].defaultArguments.length (2)
  for (var i = startIndex; i < endIndex; i++) {
    initArguments.push(this._configuration.objects[identifier].defaultArguments[i])
  }
}
obj.init.apply(obj, initArguments) (3)
delete obj.init
```

Code listing 17: _makeObject, part 3

5.3 Module Loading and Development Server

In this section, we'll see how modules are loaded in a browser environment and how the framework actually serves itself in this procedure. In the browser, before _prepareObjectCreation can be called, the code has to be loaded from the server. This is started in _loadModules in code listing 7.

```
motionjs._loadModules = function (identifier, originalCallback, createCallback) {
  var self = this
  if (typeof this._loadModulesCallbacks[identifier] == 'undefined') {
    this._loadModulesCallbacks[identifier] = []
  }
  this._loadModulesCallbacks[identifier].push(
    { success: function () {
        self._servedIdentifiers.push(identifier)          (1)
        self._prepareObjectCreation(identifier, false)
        createCallback.call(self)
      }
    , error: originalCallback                             (2)
    }
  )

  var servedIdentifiers = this._servedIdentifiers.join(',')   (3)

  var url = ''
  if (this._configuration.mode == 'production') {
    url = 'scripts/' + this._getHash(identifier, '+', servedIdentifiers) + '.min.js'   (4)
  } else {
    // _developmentPort is set via the development server
    url = 'http://localhost:' + this._developmentPort + '/modules?identifier=' + identifier + '&served=' +   (5)
servedIdentifiers
  }
  this._appendScript(url)   (6)
}
```

Code listing 18: _loadModules

For each load procedure, we need to store callbacks which can be executed once the loading is done (the reasons for this technique have been outlined in chapter 4). We provide a callback if the loading succeeds (1) and one if the loading fails (2). While the callback in case of an error is simply the function the user provided, the success callback is more complicated. After the scripts have been loaded from the server, we need to mark the current identifier as being served and then start the object creation of the identifier (1). Finally, we can resume the create procedure that was interrupted by the loading by calling the given callback.

As discussed in chapter 4, script loading depends on the context the framework runs in. If the mode is production, we do not access the development server but rather access the static files already created. Their name consists of a hash over the combination of the *identifier* and previously served *identifiers* (4). If we're running in development context, we access the development server and pass *identifier* and served *identifiers* in the *query string* of the URL (5). The served identifiers will minimize the amount of code to load as they tell the development server which identifiers have been loaded already to prevent modules from being loaded twice. We then call _appendScript (6), which is shown in listing 8:

```
motionjs._appendScript = function (url) {
  var head = document.getElementsByTagName('head')[0];   (1)
  var script = document.createElement('script');
  script.type = 'text/javascript';                        (2)
  script.src = url;
  head.appendChild(script);   (3)
}
```

Code listing 19: _appendScript

We retrieve the head element of the page (1) and create a new script tag with the url attribute set to the given URL (2). This script tag is then appended to the head tag (3), which will cause the script to load. At the end of this script is a call to _executeSuccessCallback (code listing 9) or _executeErrorCallback (code listing 10):

```
motionjs._executeSuccessCallback = function (identifier) {
  this._loadModulesCallbacks[identifier].shift().success.call(null)
}
```

Code listing 20: _executeSuccessCallback

```
motionjs._executeErrorCallback = function (identifier, errorMessage) {
  this._loadModulesCallbacks[identifier].shift().error.call(null, errorMessage, undefined)
}
```

Code listing 21: _executeErrorCallback

Both will retrieve the respective callbacks stored in listing 9 and execute them (and therefore resume the create procedure).

The assembling of the scripts to be loaded (e.g. adding the function calls at the end) needs to be done on the development server. How this is accomplished is described in the following.

The development server is a simple NodeJS server as shown in listing 11:

```
var http = require('http')
  , url = require('url')                    (1)
  , port = process.argv[2]
  , assembler = require('./../assembler')

http.createServer(function(request, response) {   (2)
  var parsedUrl = url.parse(request.url, true)

  var callback = function (error, data) {
    if (error) {
      response.writeHead(500)
      response.end('Internal error: ' + error)    (3)
    } else {
      response.writeHead(200)
      response.end(data, 'binary')
    }
  }

  if (parsedUrl.pathname == '/motion') {          (4)
    var port = parsedUrl.query.port || ''
    assembler.getMotionResponse(port, callback)
  } else if (parsedUrl.pathname == '/modules') {
    var identifier = parsedUrl.query.identifier || ''   (5)
    var servedString = parsedUrl.query.served || ''
    assembler.getModulesResponse(identifier, servedString, callback)
  } else if (parsedUrl.pathname == '/file') {
    var name = parsedUrl.query.name || ''         (6)
    assembler.getFileResponse(name, callback)
  }
}).listen(port)
console.log('Development server running at http://localhost:' + port)
```

Code listing 22: Development Server

In order to set up a server, we need to import the built-in modules http and url (1). The server will run on the port we passed via the command line (we can rely that a port is given because actually, the port is passed via our build system, which we'll discuss in the next section). The logic to handle the incoming requests is stored in our module assembler, which therefore is included as well before we create our NodeJS server. This server is instantiated via http.createServer (2). The server is then instructed to listen for requests on the given port (7). Upon each request, the anonymous function passed to the server instance is called with the parameters request and response (2). According to the pathname part of the URL (retrievable from the request), each request will be mapped to a function in the assembler module. This function will call the passed function callback upon its completion. We do this instead of returning a string as I/O operations in NodeJS are non-

blocking and therefore require us to work with callbacks. The callback executed is the same for all requests and is defined in (3). It receives the arguments error and data. If error is set, we respond with an internal server error (code 500). If no error occurred, we serve a response with the given data and response code 200 (successful). If the pathname of the URL does not match any of the defined options, we respond with error code 404 (not found).

The 3 requests for which the development server sends a response are:

- **/motion**
  This will respond with a complete motionjs object to be used by the client. It does this by combining the files bundles/org.motionjs.core/lib/core.js and bundles/org.motionjs.core/lib/adapter/client.js (which modifies the object defined in the core module). Finally, the configuration object from configuration.js is given to the _setConfiguration function of the motionjs object.

- **/modules**
  Given an identifier, it will combine the modules required into one response. If previously served identifiers are given as well, the modules that have been loaded already are not served twice. At the end of the served script is a call to either _executeSuccessCallback (see code listing 11) or _executeErrorCallback (see code listing 12). A sample response might look like code listing 12:

```
motionjs._addModule("org.motionjs.demo/model/engine", (function () { 1
var exports = {}; 2
exports.definition =
{ manufactorer: ''                    3
, init: function (manufactorer) {
    this.manufactorer = manufactorer
  }
}

exports.configuration = {}
return exports; 4
})());

motionjs._executeSuccessCallback("org.motionjs.demo/model/engine"); 5
```

Code listing 23: /modules response

The module is added to the motionjs object via the _addModule function. The function receives two arguments: the *identifier* of the module, and a function (1) that wraps around the original module content (3). The function defines an object exports (2), so the definition and the configuration (3) are being set as properties on that object. Finally, the exports object is returned. The wrapper function is executed immediately, so that we actually pass an object as the second argument to _addModule. This object is then stored in _modules and used by the framework. In (5) we see how at the end of the /modules response, we call the success callback stored for the *identifier* in _loadModules (see code listing 9) which started the loading initially. With this wrapping done by the development server, we can use the CommonJS module system without modification by the developer on the client side as well, which is a requirement for our framework.

- **/file**
  For this request, the response is simply the file which name is given in the query string. The development server reads the file content from the disk and sends it back to the client.

5.4 Build System and Deploying

In the previous section we described how the module loading works in development context. However, we need to prepare the static files for production context to not need a NodeJS to serve the modules there. For this task and others like starting the development server, we use a build system. This build system is based on Jake[18], a build tool written in JavaScript which is similar to build systems like Make or Rake. The tasks to executre are defined in a Jakefile. Each task can be called from the command line by executing jake <task name> in the directory of the Jakefile. The following tasks are available:

- **devserver**
  Starts the development server. The port can be given as an optional argument and defaults to 8080.

- **deploy**
  Generates the files needed for the production context from the *cached requests*.

- **cache**
  This task will need a second parameter to run. The argument show will display all *cached requests*. delete and either a hash as third parameter to delete a single cached requests or all to delete all requests. New requests can be added via add.

- **demo**
  This command will run the server side demo file.

- **interface**
  Given an *identifier*, interface will show the public methods of the *identifier*

- **test**
  Our framework uses nodeunit[19] as testing framework. nodeunit has the advantage of being usable on both client and server side. This enables the developer to write unit tests for both sides in the same style. With this task, server-side tests are executable via jake test server path/to/file (multiple files can be executed if a folder is specified containing many unit tests). On the client, test need to run in the browser. The framework provides a simple test server (again a simple NodeJS server) which sole purpose it is to dynamically serve a HTML response composed of the unit tests to run. The test server can be invoked via jake test client path/to/file. The test will run on each request to the testserver (e.g. running on port 8081).

[18] https://github.com/mde/node-jake
[19] https://github.com/caolan/nodeunit

6 Evaluation

To evaluate how our frameworks performs compared to the YUI3 and Joose framework discussed in chapter 3, we will pick up the point example again. We'll see how it can be implemented in MotionJS and what advantages this offers the developer. In the second part of this chapter, we're going to implement a simple event system to showcase other features of our framework not touched by the point example.

In our framework, each object is defined in its own module and therefore in its own file. The modules will be part of a bundle, for this example, we'll call it org.motionjs.demo. The following code listing 1 shows the equivalent of a Point class, stored in the file point.js:

```
exports.definition =
{ x: 0
, y: 0
, init: function (x, y) {
    this.x = x
    this.y = y
  }
, clear: function () {
    this.x = 0
    this.y = 0
  }
}
exports.configuration = {}
```

Code listing 24: Content of bundles/org.motionjs.demo/lib/point.js

One thing to notice here is that we do not specify the *identifier* in the module. The *identifier* is composed of parts of the filename, in our case org.motionjs.demo/point (the corresponding filename is bundles/org.motionjs.demo/lib/point.js). The *configuration* is empty as we do not inherit other *identifiers* or depend on other objects.

This is different for Point3D, which inherits from Point. Code listing 2 shows point3d.js:

```
exports.definition =
{ z: 0
, init: function (x, y, z) {
    this._uber('org.motionjs.demo/point', 'init', x, y)
    this.z = z
  }
, clear: function () {
    this._uber('org.motionjs.demo/point', 'clear')
    this.z = 0
  }
}
exports.configuration =
{ inherits: ['org.motionjs.demo/point'] }
```

Code listing 25: Content of bundles/org.motionjs.demo/lib/point3d.js

We only add one additional variable, z, as the variables x and y are inherited from org.motionjs.demo/point. The inheritance is specified in the *configuration* part. Through the _uber function in init and clear, we have access to the overridden methods. Note that when we would inherit from multiple *identifiers* we could access the methods of each specifically.

Now that the *identifiers* are defined it is time to use them. For our demonstration purposes, we will create a single JavaScript file which we'll run on the server first and then show that

it can be used on the client as well with no modification. Code listing 3 shows the file example1.js.

```
if (typeof window == 'undefined') {
  var motionjs = require('./../../org.motionjs.core/lib/core').motionjs
}

motionjs.create('org.motionjs.demo/point', 1, 2, function (error, point) {
  if (error) throw error
  console.log(point)
  point.clear()
  console.log(point)
})

motionjs.create('org.motionjs.demo/point3d', 1, 2, 3, function (error, point3d) {
  if (error) throw error
  console.log(point3d)
  point3d.clear()
  console.log(point3d)
})
```

Code listing 26: Content of bundles/org.motionjs.demo/lib/example1.js

Note that we perform a simple check to determine if we're in a server-side environment (where there is no window object). In that case, we need to import the motionjs object. On the client, motionjs is provided by another script, as we'll see shortly. After the motionjs object is present, we can use it to create objects from the *identifiers*. We pass the *identifier* to create and the arguments that the init function of point (point3d) can take. The last argument is the callback to execute when the object has been created. The callback receives an error argument (undefined in the case of successful creation) and the object created. To run the code on the server, we execute node example1.js in bundles/org.motionjs.demo/lib. This will give us the output shown in figure 1:

```
[redacted]:lib [redacted]$ node example1.js
{ x: 1, y: 2 }
{ x: 0, y: 0 }
{ x: 1, y: 2, z: 3 }
{ x: 0, y: 0, z: 0 }
```

Figure 6: Output of running example1.js

On the client, we create a simple HTML file (code listing 4) which includes two scripts: The first is the motionjs core dynamically assembled by the development server (running at localhost:8080). The second is the example1.js file we just created[20].

```
<!DOCTYPE HTML>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>MotionJS Demo</title>
    <script src="http://localhost:8080/motion?port=8080" type="text/javascript" charset="utf-8"></script>
    <script src="../bundles/org.motionjs.demo/lib/example1.js" type="text/javascript" charset="utf-8"></script>
  </head>
  <body>
    MotionJS Demo
  </body>
</html>
```

Code listing 27: Content of web/example1.html

Before we access the HTML file, we need to start the development server. Figure 2 shows how the server can be started via a Jake task from the root directory of the project:

[20] Note that in a real setup, a user should not be able to access files outside the web directory for security reasons: The JavaScript files inside the bundles directory might contain sensible information like passwords for databases etc.

Figure 7: Starting the development server

Important to note here is that the script does not terminate (see the cursor at the end of the output in figure 2), the NodeJS server waits for requests until the process is ended.

With the development server running, we can now open the HTML file in a browser. As we use console.log() calls to display the information in example1.js, we need to open the JavaScript console of the browser (e.g. the Firebug[21] console in Mozilla Firefox[22]) to see the results (see figure 3).

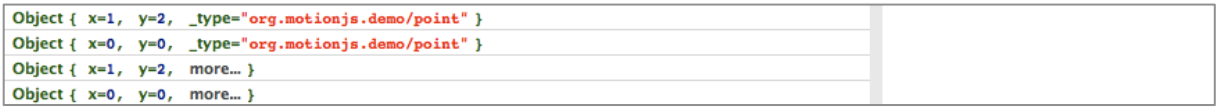

Figure 8: Output of example 1 in the Firebug console

We see that example1.js has been executed and created the same result (though formatted differently), but this time the *identifiers* were loaded via the development server. With a tool like Firebug, we can trace what happened exactly. Figure 4 shows the requests made.

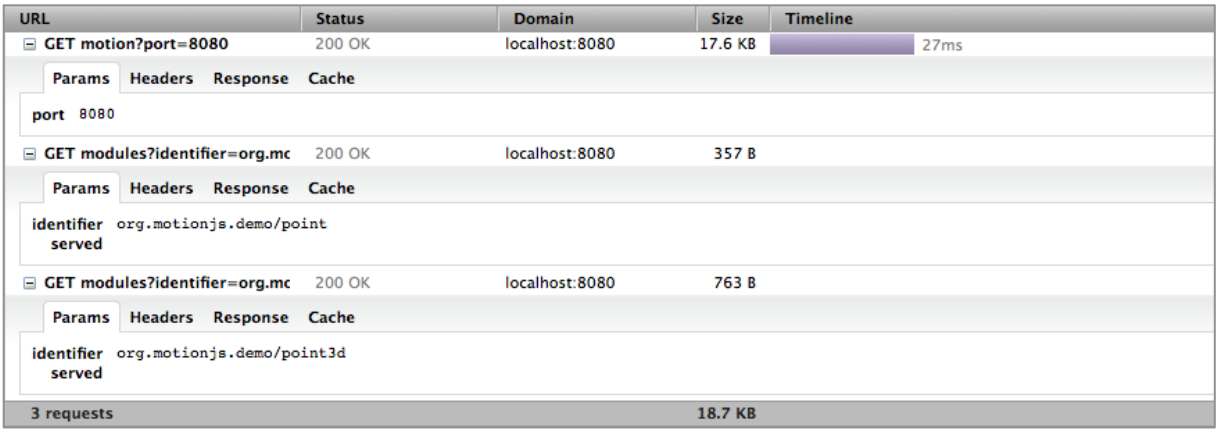

Figure 9: Request log in Firebug

We see that the first request was issued to receive the MotionJS core. Then we see two modules requests. The first loads the *identifier* org.motionjs.demo/point, the second the *identifier* org.motionjs.demo/point3d. Both times the served parameter is empty. In chapter 4, we saw that the framework is supposed to not send module code twice to the client. The second request for org.motionjs.demo/point3d should not have to download org.motionjs.demo/point, as this module is already loaded in the first request. However, as no served parameter is set, the response for org.motionjs.demo/point3d contains the module org.motionjs.demo/point as well (see figure 5).

[21] https://addons.mozilla.org/de/firefox/addon/firebug/
[22] http://www.mozilla.com/firefox/

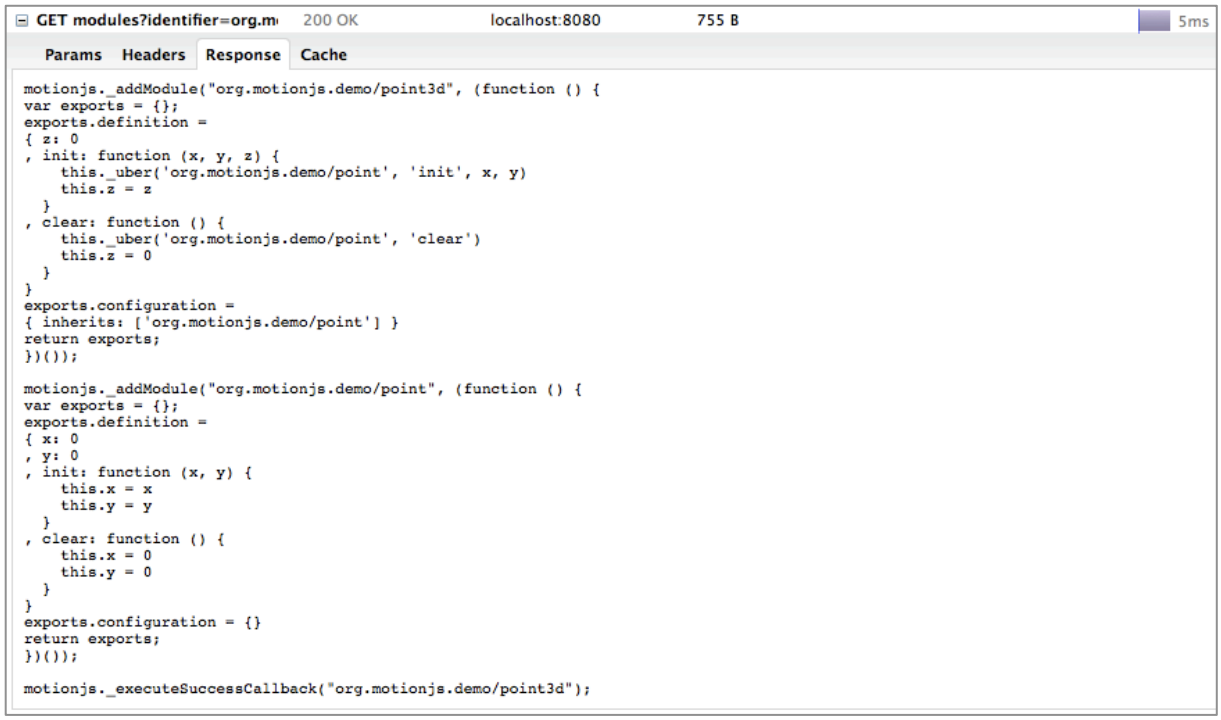

Figure 10: Response for request of org.motionjs.demo/point3d identifier

The reason for this is that the two requests happen directly after each after. At the point the second request is issued, the first one has not returned yet, so the framework does not have org.motionjs.demo/point served at this time. Of course, when org.motionjs.demo/point3d is eventually constructed, org.motionjs.demo/point might have been downloaded, resulting in unnecessary duplication. We can prevent this if we create org.motionjs.demo/point3d only after we know that org.motionjs.demo/point3d has loaded. To ensure this, we change example1.js as shown in code listing 5, nesting the two requests.

```
if (typeof window == 'undefined') {
  var motionjs = require('./../../org.motionjs.core/lib/core').motionjs
}

motionjs.create('org.motionjs.demo/point', 1, 2, function (error, point) {
  if (error) throw error
  console.log(point)
  point.clear()
  console.log(point)

  motionjs.create('org.motionjs.demo/point3d', 1, 2, 3, function (error, point3d) {
    if (error) throw error
    console.log(point3d)
    point3d.clear()
    console.log(point3d)
  })
})
```

Code listing 28: Modified content of bundles/org.motionjs.demo/example1.js

The output on the server and the client stays exactly the same[23], but the client sends a different request for org.motionjs.demo/point3d, as can be seen in figure 6.

[23] Actually, in the previous example it would have been possible that point3d was created before point as there is no guarantee that both requests take exactly the same time. With the nested approach the order is determined.

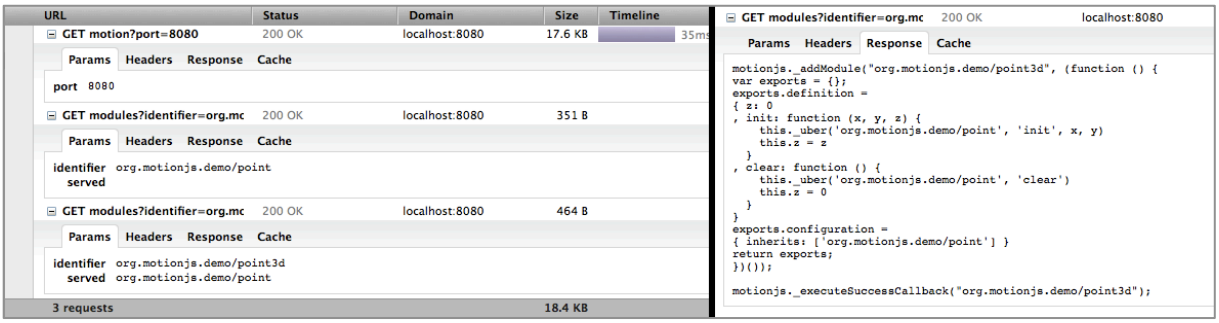

Figure 11: Firebug request log

This time, org.motionjs.demo/point is added as a served *identifier* (left of figure 6) and not added to the loaded script (right of figure 6).

We have now demonstrated most of the basic functionality of MotionJS. The more powerful features like dependency injection will be introduced in the next example. Let's imagine we want to be able to send events between some objects. These objects – we'll call them observable objects – should be able to add and remove event listeners, and to fire events themselves. When an event is fired, all objects that listen for this event should be notified. In the following, we'll see how this scenario could be implemented in MotionJS.

We will create a new identifier called motionjs.event/observable which defines the methods addEventListener, removeEventListener and fireEvent as shown in code listing 6. Every object that wants to participate in the event system needs to inherit this identifier. Our support of multiple inheritance makes it easy to do this as we don't interfere with a possibly existing type hierarchy.

```
exports.definition =
{ addEventListener: function (eventName, eventHandler) {
    this._eventManager.addEventListener(this, eventName, eventHandler)
  }
, removeEventListener: function (eventName, eventHandler) {
    this._eventManager.removeEventListener(this, eventName, eventHandler)
  }
, fireEvent: function (eventName) {
    this._eventManager.fireEvent(this, eventName)
  }
}

exports.configuration = {
  dependencies: {'org.motionjs.event/manager': '_eventManager'}
}
```

Code listing 29: Content of bundles/org.motionjs.event/observable.js

Every object inheriting the observable object is now able to call these three methods (via wrapper functions on the prototype which delegate to the functions in the observable *ability object*). The methods in the observable object however just delegate to an event manager (see code listing 6). This event manager is configured as a dependency and will be automatically injected whenever an observable object is created. Of such an event manager, there should be only one, in order that all event listeners are registered on the same object and therefore all listeners can be called when the corresponding event is fired. Code listing 7 shows the event manager:

```
exports.definition =
{ _listeners: {}

, init: function () {}

  /**
   * Adds listener eventHandler on target for eventName
   */
, addEventListener: function (target, eventName, eventHandler) {
    if (typeof this._listeners[eventName] == 'undefined') {
      this._listeners[eventName] = []
    }
    this._listeners[eventName].push({obj: target, handler: eventHandler})
  }

  /**
   * Removes listeners calling eventHandler on target for eventName
   */
, removeEventListener: function (target, eventName, eventHandler) {
    var indexesToRemove = []
    if (typeof this._listeners[eventName] == 'object') {
      for (var i = 0; i < this._listeners[eventName].length; i++) {
        if ((typeof target == 'undefined' || this._listeners[eventName][i].obj == target) && (typeof eventHandler ==
'undefined' || this._listeners[eventName][i].handler == eventHandler)) {
          indexesToRemove.push(i)
        }
      }
    }
    for (var j = 0; j < indexesToRemove.length; j++) {
      this._listeners[eventName].splice(j, 1)
    }
  }

  /**
   * Fires eventName from source
   */
, fireEvent:function (source, eventName) {
    if (typeof this._listeners[eventName] == 'object') {
      for (var i = 0; i < this._listeners[eventName].length; i++) {
        this._listeners[eventName][i].handler.call(this._listeners[eventName][i].obj, eventName, source)
      }
    }
  }
}

exports.configuration = { shared: true }
```

Code listing 30: Content of bundles/org.motionjs.event/manager.js

To demonstrate this simple event system we'll extend our previous point/point3d example. For example, assume we want to listen to an add event in point3d. Whenever a new point3d is created, all existing point3d objects should be notified. For this to work, we modify the org.motionjs.demo/point3d identifier to code listing 8:

```
exports.definition =
{ z: 0
, init: function (x, y, z) {
    this._uber('org.motionjs.demo/point', 'init', x, y)
    this.z = z
    console.log('created a point3d with ' + x + ',' + y + ',' + z)
    this.addEventListener('add', this.onAdd)
    this.fireEvent('add')
  }
, clear: function () {
    this._uber('org.motionjs.demo/point', 'clear')
    this.z = 0
  }
, onAdd: function () {
    console.log('onAdd called')
  }
}
exports.configuration =
{ inherits: ['org.motionjs.demo/point', 'org.motionjs.event/observable'] }
```

Code listing 31: Modified content of bundles/org.motionjs.demo/lib/point3d.js

We can simply add the org.motionjs.event/observable identifier to the inheritance chain to enable every instance of org.motionjs.demo/point3d to be observable. Note that if org.motionjs.event/observable would define a clear function (e.g. for removing all event listeners of the object), we would still have the possibility to access both the org.motionjs.event/observable's clear function and the org.motionjs.event/point's clear function via the helper _uber.

Now the event manager is a really naive implementation. For example, there is no indication which events are available. How would a class know which events might be fired? This is a serious flaw and should be improved. However, assume that the event manager is part of a bundle provided by someone else. The bundle itself uses the event manager quite heavily and we don't want to change the source code because this would stop us from simply updating the bundle (e.g. via a version control system). We can solve this problem elegantly with MotionJS as it offers the possibility to provide an alternative implementation for an *identifier*. Whenever the *identifier* is used, the implementation we provide is chosen by the framework, even in code we do not own. To see how this works, we instruct the framework in the *global configuration* that the event manager should be provided by another implementation. A sample configuration is shown in code listing 9.

```
exports.configuration =
{ mode: 'development'
, objects: {
    'org.motionjs.event/manager':
    { providedBy: 'org.motionjs.demo/eventManager'  }
  }
}
```

Code listing 32: Content of configuration.js

For the purpose of this example, we simply want to demonstrate the mechanism rather than providing a different functionality, so we just copy the previous event manager and only add console.log() calls to each function. To prove that the event system now works as expected, we create another file called example2.js, which is shown in code listing 10.

```
var motionjs = require('./../../org.motionjs.core/lib/core').motionjs

motionjs.create('org.motionjs.demo/point3d', 1, 1, 3, function (error, point1) {
  if (error) throw error
  motionjs.create('org.motionjs.demo/point3d', 2, 2, 3, function (error, point2) {
    if (error) throw error
  })
})
```

Code listing 33: Content of bundles/org.motionjs.demo/lib/example2.js

When we run this code via node example2.js, we will see that our own event manager has been used indeed, as shown in figure 7. The log calls have not been present in the original implementation (see code listing 7 for reference).

```
[redacted]:lib [redacted]$ node example2.js
created a point3d with 1,1,3
demo event manager - addEventListener
demo event manager - fireEvent
onAdd called
created a point3d with 2,2,3
demo event manager - addEventListener
demo event manager - fireEvent
onAdd called
onAdd called
```

Figure 12: Output of running example2.js

The two created objects add an event listener in their init function and subsequently, fire the add event. When the first point is created, only its own event listener is notified.

However, when the second point is created, two objects have been registered at the event manager to be notified, therefore "onAdd called" is printed twice. This proves that our setup with the shared event manager that is injected into the org.motionjs.demo/point3d instances and accessed via the inherited observable object works.

To summarise the evaluation of our framework, we'll contrast our framework to the alternatives (see the analysis in chapter 3) in table 1. The rows are composed of the features that have been demonstrated in the two examples carried out above. The support of each feature is indicated in the respective column of the framework.

	YUI3	Joose	MotionJS
Access overridden methods	Only in superclass	Only in superclass	In full hierarchy
Use on both client and server side without modification	No	-[24]	Yes
Single HTTP response	Only on CDN	No	Yes
Shared objects	Partial[25]	Via singleton	Yes
Dependency Injection	No	No	Yes
Exchange implementation	No	No	Yes

Table 1: Feature comparison of YUI3, Joose and MotionJS

[24] Could not be verified, see chapter 3 for details

[25] Possible only in one YUI instance, which is not always practical

7 Conclusion

In the introduction, we laid out how JavaScript has increased in importance and popularity over the last years. With this growth, the need for better support of large systems has risen as well. The goal of this project therefore was to develop a framework which enables composition of many scripts, fosters reuse and especially minimizes the effort to switch between client and server side development. The exact requirements chosen for the project where determined in chapter 3.

In the last chapter, we have evaluated how the developed framework MotionJS can be used to build applications and which features support the developer in his work. In this final chapter, we are going to reflect upon our work and investigate where improvement is needed and which steps could be taken next.

The basis of our framework really is the system to compose software from different scripts and whole bundles (see chapter 4 and 5), built upon the module standard proposed by CommonJS. While this definitely improves organisation of code, it does not fully alleviate the lack of a native module system. One reason for this is that the CommonJS standard has been designed for a synchronous environment, making it hard to be used on the client. We solved this problem by wrapping dependent code in functions, which in turn requires a development server to dynamically create the correct files. Although we believe our solution is superior to current alternative solutions, it is definitely not ideal. A native module system (like in Python, see chapter 3) is needed. This module system would have to work both server and client side (removing the workaround to load JavaScript files by appending script tags to the head of an HTML file), ideally allowing static loading (before any code is executed) and dynamic loading (supporting a callback mechanism to determine when to execute dependent code). Current discussion on the next version of JavaScript hints at a such a module system, see David Herman's talk at the Mountan View JavaScript meetup in February 2011 [**41**].

Another problem we tackled stems from JavaScript's dynamic nature. Best practices in other languages have shown that programming against abstractions (interfaces), not against particular implementations lead to better maintainability, making it easier to (ex)change code. Our framework provides a way to exhange the implementation of an identifier, which allows the developer to swap implementations without interface support of the language itself. However, this shifts the responsibility of keeping the contract to the developer: The framework does not ensure alternative implementations really provide all required functions; and that the signature of the functions are the same as well. There are two reasons for this: Firstly, objects can be modified at runtime in JavaScript, and secondly, the type of parameters in JavaScript cannot be retrieved and/or ensured. While this offers great flexibility often desirable in small projects, it becomes increasingly difficult to manage software of larger scale without type safety. Unfortunately, this problem has to be solved by the language itself and therefore, our framework cannot provide a solution. Again, recent development in JavaScript has realized this problem and ECMAScript 5

allows the developer to control the flexibility of the objects in his program. For example, it is possible to seal objects, preventing properties to be added or existing ones to be removed (for more on this, see [INSERT CITATION MANUALLY].

Apart from these shortcomings of our framework (or rather, JavaScript itself), we also identified some possible addtions that could be developed in the future. One of them is to support getters and setters to improve encapsulation. Instead of having publicly accessible properties, we could hide all variables and expose selected ones by automatically assigning getters and, if wanted, setters to them. JavaScript introduced support for this via property descriptors in ECMAScript 5 [INSERT CITATION MANUALLY].

Another addition could be to implement support for aspect oriented programming (AOP). AOP allows a developer to keep concerns separate and to weave so called aspects into code, e.g. add logging or security to functions or whole classes. JavaScript's dynamic nature would allow us to implement AOP during runtime quite easily (e.g. it is possible to replace functions). Furthermore, the next version of JavaScript might feature proxies [**42**] [**43**], which could provide an alternative, maybe more performant way to implement this.

That being said, the current state of the framework already solves many problems developers face when building large software which JavaScript, especially fostering code reuse and organisation. Therefore, the project was able to meet its original objectives. During the development, especially NodeJS turned out to be a well-suited platform The best example for this is its capability to develop servers very easily, which led to our solution to let the framework serve itself in the development context. As this technology is just starting to be used more widely, it is safe to assume that other frameworks and ideas with similar objectives to our project will spread up. Alongside, the language is getting more mature to become better suited outside its original scope. This process has started with the release of ECMAScript 5 in 2010 and will be continued with the next version. All in all, it will be interesting to see in which directions JavaScript evolves and if it is going to be used indeed for larger software, both client and server side.

References

8 Literaturverzeichnis

[1] Stephen Chapman. A Brief History of Javascript. [Online]. http://javascript.about.com/od/reference/a/history.htm

[2] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek, An Analysis of the Dynamic Behavior of JavaScript Programs.

[3] Mitch Allen, *Palm webOS*. Sebastopol, USA: O'Reilly Media, 2009.

[4] Douglas Crockford, *JavaScript: The Good Parts*. Sebastopol, USA: O'Reilly Media, 2008.

[5] ECMAScript Comittee. (1999, December) ECMAScript Language Specification, 3rd Edition. [Online]. http://www.ecma-international.org/publications/files/ECMA-ST-ARCH/ECMA-262,%203rd%20edition,%20December%201999.pdf

[6] ECMAScript Comittee. ECMAScript Language Specification, 5th Edition. [Online]. http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-262.pdf

[7] CommonJS: JavaScript Standard Library. [Online]. http://www.commonjs.org/

[8] modules - Node.js Manual & Documentation. [Online]. http://nodejs.org/docs/v0.4.3/api/modules.html

[9] Martin Fowler. (2004, January) Inversion of Control Containers and the Dependency Injection pattern. [Online]. http://martinfowler.com/articles/injection.html

[10] Misko Hevery. (2008, July) How to Think About the "new" Operator in Respect to Unit Testing. [Online]. http://misko.hevery.com/2008/07/08/how-to-think-about-the-new-operator

[11] The Transparent Language Popularity Index. [Online]. http://lang-index.sourceforge.net/

[12] Brian Shire. (2007, May) Facebook. [Online]. http://blog.facebook.com/blog.php?post=2356432130

[13] Brian Fioca. (2006, April) O'Reilly ONLamp Blog. [Online]. http://www.oreillynet.com/onlamp/blog/2006/04/digg_phps_scalability_and_perf.html

[14] (2011, March) PHP: Namespaces overview. [Online]. http://www.php.net/manual/en/language.namespaces.rationale.php

[15] (2011, March) PHP: require. [Online]. http://php.net/manual/en/function.require.php

[16] (2011, March) PHP: Autoloading. [Online]. http://php.net/manual/de/language.oop5.autoload.php

[17] Modules - Python 3.2 documentation. [Online]. http://docs.python.org/py3k/tutorial/modules.html

[18] (2011, February) Modules/1.1 - CommonJS Spec Wiki. [Online]. http://wiki.commonjs.org/wiki/Modules/1.1

[19] Swaroop C H, *A Byte of Python.*, 2008. [Online]. http://www.swaroopch.com/notes/Python_en:Object_Oriented_Programming

[20] (2010, December) Special Operators: new. [Online].
https://developer.mozilla.org/en/JavaScript/Reference/Operators/Special/new

[21] Douglas Crockford. Classical Inheritance in JavaScript. [Online].
http://www.crockford.com/javascript/inheritance.html

[22] Douglas Crockford. (2008, April) Prototypal Inheritance in JavaScrip. [Online].
http://javascript.crockford.com/prototypal.html

[23] (2011, March) PHP: Visibility. [Online].
http://www.php.net/manual/en/language.oop5.visibility.php

[24] Classes - Python v3.2 documentation. [Online].
http://docs.python.org/py3k/tutorial/classes.html#private-variables

[25] (2011, March) PHP: Interfaces. [Online].
http://www.php.net/manual/en/language.oop5.interfaces.php

[26] Matt Zandstra, *PHP Objects, Patterns, and Practice*. New York, America: Springer Verlag, 2008.

[27] (2011, March) PHP: Object Inheritance. [Online].
http://www.php.net/manual/en/language.oop5.inheritance.php

[28] Classes - Python v3.2 documentation. [Online].
http://docs.python.org/py3k/tutorial/classes.html#inheritance

[29] Built-In Functions - Python v3.2 documentation. [Online].
http://docs.python.org/py3k/library/functions.html#super

[30] Yahoo! User Interface Library: Sites powered by YUI. [Online].
http://developer.yahoo.com/yui/poweredby/

[31] YUI 3: YUI Global Object. [Online]. http://developer.yahoo.com/yui/3/yui/

[32] Dav Glass. (2010, April) Running YUI 3 Server-Side with Node.js. [Online].
http://www.yuiblog.com/blog/2010/04/05/running-yui-3-server-side-with-node-js/

[33] YUI Configurator. [Online]. http://developer.yahoo.com/yui/3/configurator/

[34] joose-js. [Online]. http://code.google.com/p/joose-js/

[35] Misko Hevery. (2010, April) Move over Java, I have fallen in love with JavaScript. [Online]. http://misko.hevery.com/2010/04/07/move-over-java-i-have-fallen-in-love-with-javascript/

[36] Why RequireJS? [Online]. http://requirejs.org/docs/why.html

[37] Jan Wolter. (2007, March) JavaScript Madness: Dynamic Script Loading. [Online].
http://unixpapa.com/js/dyna.html

[38] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Upper Saddle River: Addison-Wesley, 1995.

[39] Misko Hevery. (2008, August) Root Cause of Singletons. [Online].
http://misko.hevery.com/2008/08/25/root-cause-of-singletons/

[40] (2008, May) TotT: Using Dependancy Injection to Avoid Singletons. [Online].
http://googletesting.blogspot.com/2008/05/tott-using-dependancy-injection-to.html

[41] David Herman. (2011, February) ECMAScript.Next with David Herman of Mozilla. [Online]. http://www.youtube.com/watch?v=hs6tF-RDX4U&feature=player_embedded

[42] (2011, March) Proxy. [Online]. https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Proxy

[43] (2011, March) Catch-all Proxies. [Online]. http://wiki.ecmascript.org/doku.php?id=harmony:proxies

[44] (2011) Programming Language Popularity. [Online]. http://langpop.com/

[45] Douglas Crockford. (2001) Private Members in JavaScript. [Online]. http://www.crockford.com/javascript/private.html

[46] Andreas Junghans and Tilman Schneider. (2008, November) heise Developer. [Online]. http://www.heise.de/developer/artikel/Closures-227494.html

[47] (2011) TIOBE Programming Community Index for February 2011. [Online]. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

[48] Chuck Esterbrook. (2001, April) Using Mix-ins with Python. [Online]. http://www.linuxjournal.com/node/4540/print